



Terms and Conditions of Use of Digitised Theses from Trinity College Library Dublin

Copyright statement

All material supplied by Trinity College Library is protected by copyright (under the Copyright and Related Rights Act, 2000 as amended) and other relevant Intellectual Property Rights. By accessing and using a Digitised Thesis from Trinity College Library you acknowledge that all Intellectual Property Rights in any Works supplied are the sole and exclusive property of the copyright and/or other IPR holder. Specific copyright holders may not be explicitly identified. Use of materials from other sources within a thesis should not be construed as a claim over them.

A non-exclusive, non-transferable licence is hereby granted to those using or reproducing, in whole or in part, the material for valid purposes, providing the copyright owners are acknowledged using the normal conventions. Where specific permission to use material is required, this is identified and such permission must be sought from the copyright holder or agency cited.

Liability statement

By using a Digitised Thesis, I accept that Trinity College Dublin bears no legal responsibility for the accuracy, legality or comprehensiveness of materials contained within the thesis, and that Trinity College Dublin accepts no liability for indirect, consequential, or incidental, damages or losses arising from use of the thesis for whatever reason. Information located in a thesis may be subject to specific use constraints, details of which may not be explicitly described. It is the responsibility of potential and actual users to be aware of such constraints and to abide by them. By making use of material from a digitised thesis, you accept these copyright and disclaimer provisions. Where it is brought to the attention of Trinity College Library that there may be a breach of copyright or other restraint, it is the policy to withdraw or take down access to a thesis while the issue is being resolved.

Access Agreement

By using a Digitised Thesis from Trinity College Library you are bound by the following Terms & Conditions. Please read them carefully.

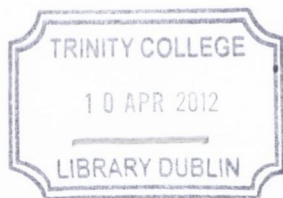
I have read and I understand the following statement: All material supplied via a Digitised Thesis from Trinity College Library is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of a thesis is not permitted, except that material may be duplicated by you for your research use or for educational purposes in electronic or print form providing the copyright owners are acknowledged using the normal conventions. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone. This copy has been supplied on the understanding that it is copyright material and that no quotation from the thesis may be published without proper acknowledgement.

FPGA based solver for Internet Eigenvectors

Séamas McGettrick

A thesis submitted to the University of Dublin in partial fulfilment of the
requirements of the degree of Doctor of Philosophy

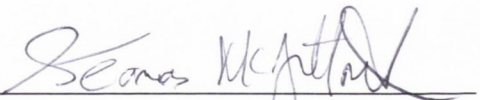
School of Engineering,
Department of Mechanical and Manufacturing Engineering,
University of Dublin, Trinity College.



9430

Declaration

I declare that the work described in this thesis is, unless otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other University. I agree that Trinity College Library may lend or copy this thesis upon request.

Signature of author: 

Séamas McGettrick

May 2009

Abstract

Calculating an Internet Eigenvector for Internet ranking is a massive computational problem dominated by Sparse Matrix by Vector Multiplication (SMVM) where the matrix is very sparse, unsymmetrical and unstructured. The computation presents a serious challenge to general purpose processors (GPP) on which it is run. The GPP only achieves a fraction of its peak performance. The result of this is a very lengthy computation time.

Specialised hardware running on FPGAs has been used in other scientific applications to accelerate algorithms where very sparse SMVM is required. This thesis attempts to apply the knowledge gained, from using FPGAs on these other scientific calculations, to PageRank - the most popular Internet ranking algorithm - for the first time.

This is done firstly by implementing and benchmarking two floating point hardware architectures (SPAR and SCAR) designed for finite element analysis against a state of the art GPP. Two variations of the SCAR SMVM unit are presented. One of which has a single MAC pipeline and the second which has a dual MAC pipeline and thus fully utilises the available memory bandwidth. A third architecture is proposed and implemented which is designed to work especially with the PageRank algorithm. Two variations of this hardware architecture are also created, one that has three SMVM units with 1024 vector entries in the X and Y cache and the other which has two SMVM units with 2048 vector words stored in the cache. The five variations of the hardware architecture are targeted to a Virtex-II V6000 FPGA with four independent memory channels.

With the exception of the SPAR architecture, all of the FPGA based architectures achieved over 50% of the performance of the GPP despite having a clock rate of

about 100 MHz which is 30 times slower than the 3 GHz clock rate of the GPP. The DDR-266 SDRAMs used by the FPGA are also less than half the speed of the DDR2-667 SDRAM modules used in the GPP. The Specialised PageRank hardware with two SMVM units and 2048 Vector entries in its cache is the highest performing FPGA architecture achieving approximately 80% of the GPP performance.

The FPGA architectures pad their streams with NOPs to avoid Read After Write (RAW) hazards in the adders. NOPs inserted to avoid RAW hazards account for between 12% and 16% of the matrix stream length. It is shown that the number of RAW hazards is directly proportional to adder latency and so reducing adder latency should reduce NOPs in the stream. One method to reduce adder latency is to reduce precision. Thus, a number of tests are presented to estimate the precision needed to calculate the PageRank algorithm and two of the above systems are implemented using fixed point arithmetic. The performance of the fixed point implementations is only marginally better than the floating point implementations. This is due to the SMVM being only part of the PageRank algorithm.

Finally the two best performing designs are targeted at a Virtex-5 FPGA with the result that the generic SMVM architecture performs on average about 1.5 times faster than the GPP and the specialised PageRank HW can run at about 1.8 times the speed of the GPP even though the clock rate is 14 times slower than that of the GPP.

Throughout the thesis a number of limitations of the architectures are discussed together with suggested improvements that could be made to the architectures in future implementations of a hardware solver for Internet Eigenvectors.

Publications Associated with the Thesis

- [16] S. McGettrick, D. Geraghty, and C. McElroy, "Searching the Web with an FPGA Based Search Engine," in *Reconfigurable Computing: Architectures, Tools and Applications*, Brazil, 2007, pp. 350-357.
- [17] D. Gregg, C. Mc Sweeney, C. McElroy, F. Connor, S. McGettrick, D. Moloney, and D. Geraghty, "FPGA Based Sparse Matrix Vector Multiplication using Commodity DRAM Memory," in *Field Programmable Logic and Applications*, 2007. FPL 2007. International Conference on, 2007, pp. 786-791.
- [18] S. McGettrick, D. Geraghty, and C. McElroy, "Towards an FPGA solver for the PageRank EigenVector Problem," in *Parallel Computing: Architectures, Algorithms and Applications*, ParCo 2007, Julich and RWTH Aachen, Germany, 2007, pp. 793-800.
- [19] S. McGettrick, D. Geraghty, and C. McElroy, "An FPGA architecture for the Pagerank eigenvector problem," in *Field Programmable Logic and Applications*, 2008. FPL 2008. International Conference on, 2008, pp. 523-526.

Acknowledgements

It is a pleasure to thank some of the many people who made this work possible. I would like to thank my supervisor, Mr. Dermot Geraghty, for all his support and guidance over the last number of years. Special thanks is also extended to the FIAMMA team, especially Ciarán McElroy who answered all of the many questions I had when I first started working with FPGAs and was always willing to talk through difficulties and discuss possible solutions whenever I needed help. I would like to say thanks to the students and researchers in my lab who always gave support when it was needed.

My family and friends have been most supportive throughout my studies. Particular thanks to my Mam and Dad for giving me the confidence and belief to get this research finished.

Most of all I would like to thank Beth, for all her help.

Table of contents

1	Introduction.....	1
1.1	Introduction.....	1
1.2	FPGA hardware	4
1.3	The FIAMMA Project	5
1.3.1	Accreditation of work	5
1.4	Contribution.....	7
1.4.1	Precision of the PageRank Calculation.....	7
1.4.2	PageRank in hardware	7
1.4.3	Implementation and Benchmarking of two generic Linear Algebra architectures	8
1.4.4	Implementation and Benchmarking of specialised PageRank hardware..	9
1.5	Publications.....	9
1.6	Thesis Guide	10
2	Technology Background.....	13
2.1	Introduction.....	13
2.2	Hardware Design Flow	14
2.3	Field Programmable Gate Arrays (FPGAs).....	15
2.3.1	Development board.....	18
2.3.2	Virtex-5.....	19
2.4	Memory - Double Data Rate SDRAM (SDRAM).....	21
2.4.1	PC Memory Hierarchy.....	22
2.4.2	FPGA Memory Hierarchy	23
2.4.3	FPGA and PC Memory Hierarchy Compared	25
2.5	Number representation.....	27
2.5.1	Floating point.....	27
2.5.2	Fixed point	29
2.6	Summary.....	30
3	Internet Search	33
3.1	Introduction.....	33
3.2	Crawling.....	35
3.3	Indexing	37
3.4	Query Processing.....	39
3.5	Ranking	40
3.5.1	Boolean Search Engines	40
3.5.2	Latent Semantic Analysis	41
3.5.3	Internet Ranking Algorithms	44
3.5.4	Hypertext Induced Topic Search (HITS).....	45
3.5.5	PageRank	49
3.6	Sparse Matrix by Vector Multiplication	55

3.6.1	Compressed Row/Column Storage	56
3.7	Summary	59
4	Literature Review	63
4.1	Introduction	63
4.2	Internet Search.....	63
4.2.1	Alternative PageRank Algorithms	64
4.2.2	Modifications to Power Method for PR	66
4.2.3	Parallel Computation of PageRank	74
4.2.4	Other FPGA Architectures for Internet Ranking	75
4.3	SMVM.....	75
4.3.1	Optimisation for SMVM in software	76
4.3.2	Specialised hardware for SMVM.....	81
4.3.3	SMVM on FPGAs.....	84
4.4	Arithmetic units on FPGA.....	89
4.4.1	Fixed point/Integer arithmetic on FPGA.....	89
4.4.2	Floating point arithmetic on FPGA	91
4.5	Summary	94
5	Design and Implementation	97
5.1	Introduction	97
5.2	Architecture.....	97
5.2.1	MicroBlaze	99
5.2.2	Wishbone Bus	101
5.2.3	Wishbone Bus Interface	103
5.2.4	Wishbone Bus Operations.....	106
5.2.5	PCI Interface	108
5.2.6	Memory Controller.....	109
5.3	Vector Unit.....	110
5.4	SMVM Architecture 1 - SPAR	114
5.4.1	SPAR Modifications	118
5.5	SMVM Architecture 2 - SCAR.....	119
5.5.1	SCAR Data Structure	122
5.5.2	SCAR Hardware.....	125
5.5.3	Dual SCAR architecture.....	127
5.6	SMVM Architecture 3 – PageRank SMVM	129
5.6.1	Removing SMVM from PageRank	130
5.6.2	PageRank Hardware.....	133
5.6.3	Dual path solution	137
5.7	Bus Contention on SMVM architectures	138
5.8	Software	139
5.8.1	Software on Host PC.....	139
5.8.2	Hardware Communication kernels.....	139
5.8.3	Programming the MicroBlaze	140
5.9	Summary	141
6	Results	143
6.1	Introduction	143
6.2	Benchmark Matrices	144
6.3	General Purpose Processor results	146
6.4	Floating Point PageRank Performance on Virtex-II	149
6.4.1	SMVM Architecture 1 – SPAR.....	150
6.4.2	SMVM Architecture 2 – SCAR	152

6.4.3	SMVM Architecture 3 – PageRank HW	157
6.4.4	SMVM Architectures Compared	162
6.5	Performance Limitations.....	165
6.5.1	Bus Contention	165
6.5.2	Load Balancing.....	170
6.5.3	Measuring NOP frequency	172
6.6	Precision.....	176
6.6.1	Fixed point Emulation	176
6.6.2	Zipfian distributions.....	179
6.7	Fixed Point Performance on Virtex-II	181
6.7.1	Fixed Point SCAR implementation	182
6.7.2	Fixed Point PageRank HW implementation.....	184
6.7.3	Fixed Point Architecture compared	185
6.8	Floating Point Performance on Virtex-5.....	186
6.9	Summary	191
7	Discussion and Conclusions	197
7.1	Introduction.....	197
7.2	Contribution of this Thesis	198
7.2.1	Precision of the PageRank calculation.....	198
7.2.2	PageRank in Hardware	199
7.2.3	Implementation and Benchmarking of two Generic Linear Algebra Systems	200
7.2.4	Implementation and Benchmarking of specialised PageRank Hardware	202
7.3	Limitations and Future Work.....	204
7.4	Final Thoughts	208
8	References.....	209
9	Appendix 1: FE Matrices.....	221
10	Appendix 2: Source Code	223

List of figures

Figure 2.1 Design flow for FPGA design, based on [22].....	14
Figure 2.2 FPGA Architecture Overview (Virtex-II) [15].....	16
Figure 2.3 CLB Element (Virtex-II) [15].....	17
Figure 2.4 Slice Configuration (Virtex-II) [15]	18
Figure 2.5 Alpha-Data ADP-DRCII board [28].....	19
Figure 2.6 Virtex-5 Logic Block.....	20
Figure 2.7 Intel Xeon Woodcrest memory architecture, based on [38] (L1 cache is 32KB data & 32KB Instruction)	23
Figure 2.8 Memory Bandwidth of the Xeon Woodcrest and FPGA for various memory block sizes.	26
Figure 2.9 Floating point number.....	28
Figure 3.1 Overview of a Search Engine	35
Figure 3.2 Google Webmaster controls.....	37
Figure 3.3 Link matrix created from crawled pages	38
Figure 3.4 Reverse Term Index creation.....	38
Figure 3.5 The truncation of the S,U and V matrices in LSA.....	43
Figure 3.6 Example of a 10 Node Neighbourhood graph	45
Figure 3.7 An Authority node and a Hub node	45
Figure 3.8 Building a neighbourhood graph in HITS	47
Figure 3.9 Finalised neighbourhood graph for “dog food” query.....	47
Figure 3.10 Link matrix for HITS neighbourhood graph.....	48
Figure 3.11 Matlab code for HITS ranking Algorithm	48
Figure 3.12 Simplified PageRank Calculation [49]	50
Figure 3.13 Sample Web for PageRank adjacency matrix.....	51
Figure 3.14 PageRank adjacency Matrix	52
Figure 3.15 Matlab code for PageRank ranking Algorithm.....	54
Figure 3.16 Sparse Matrix by Vector Multiplication	56
Figure 3.17 CRS vectors of A matrix in Figure 3.16	57
Figure 3.18 CCS vectors of A matrix in Figure 3.16	57
Figure 3.19 Spatial locality of reference in the NZE leads to temporal locality of reference in X and Y vectors.....	58
Figure 3.20 Matrix with poor spatial locality increase the number of cache misses	59
Figure 4.1 Adaptive power method mis-identified convergence	68
Figure 4.2 Example of Internet divided into hosts for Blockrank.....	70
Figure 4.3 Adjacency matrix for Internet shown in Figure 4.2.....	70
Figure 4.4 Local and Global adjacency matrices for Internet shown in Figure 4.2	71
Figure 4.5 Register blocking dense block scheme to increase data reuse.....	77
Figure 4.6 Clovertown memory hierarchy [37, 86]	79
Figure 4.7 AMD Opteron memory hierarchy [86].....	79

Figure 4.8 Webbase matrix performance on Intel Xeon and AMD Opteron [37] (Bars on the graph represent the performance of the IA matrix on the Xeon and AMD processor respectively. The line graphs represent the median performance of Williams test set, showing a large discrepancy between IA matrices and other Sparse matrices.)	80
Figure 4.9 Zhuo and Prasanna's system model [13]	86
Figure 4.10 SMVM architecture used by Zhuo and Prasanna	87
Figure 4.11 Reduction circuit used by Zhuo and Prasanna [96]	87
Figure 5.1 High level view of PageRank hardware architecture	98
Figure 5.2 MicroBlaze controller unit	99
Figure 5.3 Wishbone multiplexed Shared bus with round robin arbiter	102
Figure 5.4 Wishbone slave interface for SMVM unit	104
Figure 5.5 Wishbone Master Interface for SMVM unit	105
Figure 5.6 Wishbone single read timing diagram	107
Figure 5.7 Wishbone block write timing diagram	108
Figure 5.8 Block diagram of PCI interface	109
Figure 5.9 Overview of memory controller	110
Figure 5.10 Block diagram of Hardware Vector unit	112
Figure 5.11 Vector unit command word	113
Figure 5.12 SPAR storage format	115
Figure 5.13 SPAR architecture	116
Figure 5.14 Example of a RAW hazard	118
Figure 5.15 SPAR strip reordering scheme to eliminate Y-cache misses	119
Figure 5.16 NxM matrix divided into blocks for SCAR unit	121
Figure 5.17 SCAR data words A) block update B) matrix data point	122
Figure 5.18 absolute and relative addressing as used in the SCAR data format. The column and row coordinates contained in the block update contain the absolute address of the matrix in the matrix and all other addresses are given relative to the block.	123
Figure 5.19 Simple SCAR reordering (using round robin system between the three queues)	124
Figure 5.20 Opportunistic reordering in SCAR	125
Figure 5.21 Outline of SCAR architecture	126
Figure 5.22 Block diagram of Dual SCAR architecture	129
Figure 5.23 Example of PageRank SMVM	130
Figure 5.24 The new column value vector and pattern matrix	131
Figure 5.25 Replacing SMVM in PageRank	132
Figure 5.26 Pattern adder data word	133
Figure 5.27 Encoding the Pattern adder stream	134
Figure 5.28 Block diagram of Pattern adder	135
Figure 5.29 Internal Structure of X-buffer	136
Figure 5.30 Dual Path Pattern adder	137
Figure 5.31 The C code for the PageRank algorithm run on the MicroBlaze	141
Figure 6.1 General Purpose Processor Benchmark results	147
Figure 6.2 GPP % Peak Computational Performance.	148
Figure 6.3 SPAR system Benchmark results Vs. GPP results. Primary axis: Performance in MFLOPS. Secondary axis: %peak computational performance (shown by circles for single SPAR and triangles for three SPAR system).	151
Figure 6.4 Performance Benchmarks for SCAR architecture	153
Figure 6.5 Performance Benchmarks for Dual MAC SCAR architecture	156
Figure 6.6 Performance Benchmarks for SCAR system Vs. PC	157
Figure 6.7 Performance Benchmark for Dual 1024 line X-Buffer PageRank system	159

Figure 6.8 Performance Benchmark for Dual 2048 line X-Buffer PageRank system ..	161
Figure 6.9 Performance Benchmarks for PageRank system Vs. GPP	162
Figure 6.10 Summary of Floating point Benchmarks on Virtex-II.....	163
Figure 6.11 Average percentage of GPP performance achieved by test architectures .	163
Figure 6.12 % of achievable computational bandwidth or % adder utilisation for the GPP and all the FPGA architectures running the complete PageRank solver	164
Figure 6.13 Histogram of NZE per NZ matrix block.....	167
Figure 6.14 SMVM Performance of PageRank system with two vector buses to alleviate shared bus contention.	169
Figure 6.15 PageRank Algorithm performance of PageRank system with two vector buses to alleviate shared bus contention.	170
Figure 6.16 Load balancing analysis of the IA matrices in the dual MAC SCAR system	171
Figure 6.17 Load balanced calculation of SMVM on IA matrices	172
Figure 6.18 The effect of RCM on the number of Nops in the Dual MAC SCAR stream (lower is better)	174
Figure 6.19 Nops in SCAR stream vs. adder latency	175
Figure 6.20 The fixed-point bits needed vs. percentage correctly ordered	177
Figure 6.21 matrix size Vs. fixed point bits needed	178
Figure 6.22 PageRank Vector following the power law	179
Figure 6.23 Fitting Zipfian distributions to emulation graphs	180
Figure 6.24 Estimating fixed point precision needed for larger systems with Zipfian distribution	181
Figure 6.25 Performance results for the fixed point dual MAC SCAR system on Virtex-II.	183
Figure 6.26 Performance results for fixed point dual path PageRank HW with 2048 line X-buffer on Virtex-II.....	185
Figure 6.27 Fixed and Floating point performance as a percentage of GPP performance	186
Figure 6.28 Performance results for single MAC SCAR system on Virtex-5 with 250MHz clock	188
Figure 6.29 Performance results for dual MAC SCAR system on Virtex-5 with 222MHz clock	189
Figure 6.30 Performance results for dual path PageRank HW system on Virtex-5 with 2048 line cache and 222MHz clock	190
Figure 6.31 Summary of performance of FPGA architectures on Virtex-5 as a % of GPP performance.....	191

List of tables

Table 2.1 Xilinx Virtex family compared (* multipliers replaced with extreme DSP slices, † Slices in Virtex-5 contain four LUT and four flip-flops)	20
Table 2.2 A summary of DDR data rates [35].....	22
Table 2.3 FPGA clock rates needed to utilise maximum memory bandwidth in systems of various data word sizes.....	24
Table 2.4 SDRAM latency of FPGA and Intel Woodcrest.....	25
Table 2.5 double and single precision floating point.....	28
Table 2.6 Fixed point range and precision.....	30
Table 4.1 Summary of Fixed point adders [101]	90
Table 4.2 Summary of Fixed point Multipliers [101].....	90
Table 4.3 Summary of double precision floating point Adders on Virtex-II and Virtex-II pro (N/A = not available).....	92
Table 4.4 Summary of double precision floating point Multipliers on Virtex-II and Virtex-II pro (N/A = not available).....	93
Table 5.1 Vector unit functions	111
Table 5.2 The 2-bit command sub-word.....	124
Table 5.3 Table of Hardware communication kernel functions	140
Table 6.1 Details of Benchmark matrices.....	145
Table 6.2 Finite Element and Internet Adjacency matrices compared	146
Table 6.3 FPGA utilisation for floating point SPAR on Virtex-II.....	150
Table 6.4 FPGA utilisation for floating point SCAR on Virtex-II	153
Table 6.5 FPGA utilisation for floating point 3xDual MAC SCAR on Virtex-II	155
Table 6.6 FPGA utilisation for floating point Dual PageRank system on Virtex-II	158
Table 6.7 FPGA utilisation for modified floating point PageRank system on Virtex-II	160
Table 6.8 Shared Vector Bus utilisation in SCAR architecture.....	166
Table 6.9 Nops in SCAR data stream	173
Table 6.10 A summary of the average percentage of Nops in FPGA architectures data stream using opportunistic reordering.	173
Table 6.11 FPGA utilisation for fixed point Dual MAC SCAR on Virtex-II	183
Table 6.12 FPGA utilisation for fixed point Dual Path PageRank HW with 2048 line X-buffers on Virtex-II.....	184
Table 6.13 Resources of the Virtex-II v6000 FPGA and Virtex-5 lx155 compared	187
Table 7.1 Time spend on operations carried out by the Vector unit and SMVM unit when calculating the PageRank algorithm. (figures in bold indicate vector unit limited architectures).....	206
Table 9.1 Summary of Finite Element Matrices (Bus Utilisation and FPU Utilisation quoted as a single SMVM SCAR unit).....	221

Chapter One

1 Introduction

1.1 Introduction

In 1962, J.C.R. Licklider published a series of memos discussing his idea for a global network [1] which would allow people to access large libraries of information from a computer. The vision that Licklider presented in this paper was quite possibly the humble root of the reality of the Internet today. Licklider became the first head of computer research at the Defence Advanced Research Project Agency (DARPA) in 1962. Here, he was in charge of three newly installed network terminals, one in Santa Monica, one in Berkeley and one in MIT. His need for a global unified network was made evident by the problems he faced with these three different machines. Each of the three terminals had different user commands, making it impossible to communicate with the three computers from the same terminal [2]. Thus, Licklider convinced his successors at DARPA of the importance of his networking concept. Meanwhile, in MIT, Leonard Kleinrock had just developed packet switching theory [3]. Previously, all communications had been done using switching circuits, by which a direct circuit was created between the two people communicating. Packet switching changed this procedure by allowing many people to share a line, thus making a direct electronic circuit unnecessary. Each package of information in Kleinrock's theory contained its own destination address and so could be routed automatically. This process has become normal procedure for all data networks.

In 1972, an electronic engineer by the name of Robert Kahn joined the team at DARPA. He began working on a protocol to unify the way networks communicated with each other [4]. Kahn, with the help of Vinton Cerf, created the protocol which is used on the Internet to this day. This protocol is known as Transmission Control

Protocol/Internet Protocol (TCP/IP) [4]. The backbones of the Internet were then complete, making it possible for computers to communicate across the world.

The Internet continued changing from its initial foundations. In 1989, Tim Berners-Lee created a language that changed the way the Internet is used to this day [5]. Tim Berners-Lee may not have invented the Internet, but his work made the Internet so accessible that he is often praised in this way. Berners-Lee's work on hypertext mark-up language (HTML) made the Internet accessible to the masses. Berners-Lee's HTML sparked the creation of the very first website and the first browsers to view websites. These innovations, coupled with the falling cost of technology, helped create the Internet's current form.

Currently, the Internet is the world's largest document collection, containing over 1 trillion pages; that is 1,000,000,000,000 pages [6]. Licklider's dream of a 'Galactic Network' has now developed into a reality. With a trillion pages to choose from, though, finding one desired page on the Internet can seem a daunting task for any user. In order to remedy this problem, search engines have been designed to wade through the vastness of the Internet and retrieve the most useful documents for any given query.

Internet search engines generally perform two main processes. The first is the user query process which is the view of a search engine that a user gets when they make a query. This process parses the user query and builds a ranked list of pages the search engine deems most relevant to the user's query. This is done by consulting a vast database/index of pages stored on the search engine servers. The page ranking is important since many search terms return a large number of pages and users will usually only check the first few results returned. The user query process culminates in the user being served with a webpage containing this ranked list of these relevant pages. However, long before any user submits a query, the search engine must perform the second process. This second process is independent of user queries and aims to create an extensive index for use in the user query process. This process begins with automated web-page parsing programs, called *spiders*, visiting and gathering information about all the pages that the search engine will later index. This first step in the process is commonly called *crawling*. The pages retrieved are then parsed and vital information is saved to a database, so that a list of related pages can be retrieved quickly when a query is submitted. This second step is called the

indexing step. The third necessary step is to rank the pages, which can be done either at or before query time. In both cases, a ranking algorithm handles this process. Documents with relevant content receive a high rank and irrelevant documents receive low ranking scores. Once the crawling, indexing and ranking (if any) steps are complete the search engine is ready to deal with user queries. Crawling and indexing are on going operations and periodically update the index being used to lookup user queries. This ensures that the search engine responds to deleted pages as well as new content on the Internet.

Before the advent of Google™, malicious computer programmers, often referred to as spammers, had found ways to manipulate the existing ranking algorithms used by search engines. Thus, an Internet user often had to search through pages of useless results (Spam) before finding a page with useful information. It was vital for the future of the Internet as a useful knowledge resource that useful and relevant data be easily accessible. Google made this possible by quickly returning relevant pages for a query every time and managed to avoid including spam. This improved search engine was made possible by a new ranking algorithm called PageRank™. PageRank remains an important part of the Google search engine to this day [7].

PageRank achieved a higher quality result than other search engines of the day by taking advantage of the hyperlinked structure of the Internet [7]. Most pages on the Internet contain hyperlinks to other legitimately relevant pages on the Internet. Each hyperlink to a page is counted as a vote for the content of the page to which the hyperlink leads. Pages that receive more votes are given a higher ranking score, since a page that gets more votes is assumed to have better content. This ranking is a problem in linear algebra, which requires the eigenvector of a very large sparse matrix to be calculated. This PageRank calculation has been dubbed “the largest matrix calculation in the world” [8]. The matrix at the centre of the calculation is of the order of 1 trillion rows and columns and is constantly growing as new pages are added to the Google index. The PageRank calculation is only one of about 200 factors used to rank a webpage to a given search query [7]. It is none the less a vital part of the Google search engine and is by far the most computationally intensive of the ranking scores. The PageRank calculation is query independent. Thus, it is calculated prior to query time. The freshness of the PageRank algorithm is still important since the Internet is a highly dynamic environment. Pages are created,

updated and removed constantly and so the hyperlink structure of the Internet changes and thus the PageRank. Therefore the PageRank of all pages must be kept up to date to avoid returning pages whose content is no longer relevant or useful to the user. The PageRank calculation is currently solved through the use of General Purpose Processors (GPPs). In 2003, L.A. Barroso, a Google engineer, published a paper that stated that over fifteen thousand commodity class PCs were in use by Google worldwide [9]. Barroso's paper is primarily concerned with user query handling and the number of PCs used is probably much larger now, since Google's index has grown considerably in recent years but it does indicate the scale of the problem. Google use commodity PCs as they give better cost/performance than high end servers [9]. One of the major problems that Google face with this strategy is the reliability of the commodity PCs. Commodity PCs are inherently unreliable and so Google use fault tolerant software and redundancy to remove this problem. Another better approach may be possible, however. Many hardware developers are claiming large performance increases are possible by bringing modern FPGA parallelism and size to bear on scientific problems.

1.2 FPGA hardware

The first Field Programmable Gate Array (FPGA) was created by Xilinx in 1984 [10]. This FPGA was a simple device that could be reprogrammed by the user to calculate different functions. The reducing cost of transistors has allowed more modern FPGAs grow in complexity and usefulness. The modern FPGA is a dynamically reconfigurable microchip that can be programmed to become almost any digital circuit. FPGAs now contain large quantities of logic as well as specialised arithmetic units, memory units and a great deal of I/O capabilities. The FPGA can be programmed to become a specialised hardware unit which is optimised to the special needs of an application without the expense and complex design flow that would be necessary to produce a custom Application Specific Integrated Circuit (ASIC).

Traditionally, FPGAs were used in integer and fixed point arithmetic in fields such as Digital Signal Processing (DSP). FPGAs are especially suited to perform largely parallelisable algorithms. The achievable clock rate in the FPGA is much lower than the clock rates of GPPs. However, since the FPGA is programmable a user can create

a highly parallelised version of an algorithm and still obtain accelerations over the GPP performance.

Each generation of FPGA released shows a dramatic improvement over previous generations in terms of I/O pins, clock rate increases and programmable logic increases. Up until a few years ago it was impossible to implement a floating point arithmetic unit on a single FPGA due to the complexity and size. However, this all changed with the introduction of the Virtex generation devices. It is now possible to implement floating point operators on FPGA along with other circuitry. The advent of these larger FPGAs and the ever increasing gap between GPP processor speed and memory I/O has sparked a great deal of research into using FPGAs to accelerate floating-point operations. [11-14]. One such research project was the FIAMMA project which was used as a starting point for this work.

1.3 The FIAMMA Project

The aim of the FIAMMA¹ project was to develop an FPGA based accelerator for Finite Element Analysis (FEA). This was implemented on an FPGA daughterboard connected to the host PC via a PCI connection. The host PC offloaded FEA problems to the FPGA daughterboard via the PCI bus. FEA problems often have large memory requirements and so the FPGA daughter board was populated with 4 separate banks of SDRAM. The daughterboard contained a single Virtex-II FPGA [15] on which the accelerator was built. The accelerator was designed around 4 Wishbone buses [16] to which a MicroBlaze controller [15], three SMVM units, a divider and a multi-purpose Vector Unit were attached. The initial version of the FIAMMA accelerator used a column-based SMVM architecture which was implemented from a legacy architecture called SPAR [17]. This was later replaced with the smaller and more efficient SCAR architecture which was designed and implemented by the FIAMMA team.

1.3.1 Accreditation of work

The FIAMMA project had several team members. In this section the work carried out by each member of the team will be outlined.

¹ Finite Iterative and Matrix Mathematic Accelerator

Séamas McGettrick: The author is responsible for all of the aspects of the search project. This contribution includes creation of additional software for the PageRank algorithm and other software functions. Furthermore, the original FIAMMA architecture was modified to increase performance and to remove bugs that became evident when computing Internet ranking algorithms. The author designed and implemented a double-precision floating-point divider and dual port read-write cache for use in the FIAMMA architecture. He was also responsible for implementing and testing the SPAR architecture. The author implemented the fixed point SCAR unit and carried out all benchmarks on all the architectures unless otherwise stated. The FIAMMA architecture was designed for use with Virtex-II and so it had to be ported to the newer Virtex-5 device. To achieve this, all Intellectual Property (IP) had to be regenerated using Xilinx's core generator and a new SDRAM controller had to be created using Xilinx's Memory Interface Generator (MIG). Finally, the author is responsible for the design and implementation of the PageRank architecture which is discussed in section 5.5.3. This architecture was specially designed to take advantage of the unique structure of the PageRank algorithm.

Ciarán Mc Elroy: Ciarán implemented the initial floating point version of the SCAR architecture and designed the Wishbone bus system used in the FIAMMA and the search system. He also created a design flow to aid with the creation of bit files for the FIAMMA and search systems.

Fergal Connor: Fergal delivered the SDRAM controller design used in this project, as well as developing the hardware floating point cores (multiplier and adder) used in the SPAR system (Chapter 5).

Colm Mc Sweeney: Colm developed the core software library routines used in this project. The ranking software is built to use a number of Colm's libraries.

Dermot Geraghty: Dermot is the Principal Investigator for the FIAMMA and Search project and supervisor for this PhD research project.

1.4 Contribution

This thesis evaluates the viability of using an FPGA based accelerator architecture to calculate the PageRank Internet ranking algorithm. This was done by emulating PageRank precision requirements, designing and implementing hardware and software for the PageRank algorithm and benchmarking this hardware against the GPP. To the best of the author's knowledge this work is the first published work to investigate Internet ranking algorithms being run on an FPGA. The PageRank problem is, in essence, a problem in linear algebra. In recent years, a myriad of work has been published with regard to linear algebra problems running on FPGAs. The work outlined in this thesis expands this research to the area of Internet search for the first time.

1.4.1 Precision of the PageRank Calculation

A system for estimating the precision needed to calculate the PageRank vector is proposed. To this end, a fixed point emulator was developed and the PageRank was calculated for a number of real Internet Adjacency (IA) matrices over a number of precisions. The PageRank vector is known to follow a Zipfian distribution. A Zipfian distribution was thus created that fitted the fixed point emulations. This Zipfian distribution was then used to extrapolate the precision needed for IA matrices of any size.

It is important to know the precision of the calculation. This data can be used to ensure the most suitable number representation is used in the architectures to compute the PageRank vector.

1.4.2 PageRank in hardware

The design and implementation of an FPGA based architecture for calculating the Google PageRank vector (i.e. the power eigenvector method) is presented with implementation on a Virtex-II FPGA. This architecture is specially designed to cater for large sparse matrices like IA matrices. The algorithm was broken down into operations that can be performed at a hardware level. Then a number of software libraries were written to control the hardware, so as to calculate the PageRank algorithm.

A number of variants of the hardware architecture were developed. The first version uses double precision floating point representation with each of the three SMVM architectures (discussed in the next two sections) and the second version uses the information gained in the precision tests to create a fixed point solution for the PageRank architecture. The third version is the same as version one with the exception that the whole design is ported to Virtex-5. These versions of the hardware are benchmarked and compared to one another and the GPP with real IA matrices. The effect of adder latency, bus contention and load balancing on the architecture performance is discussed and the effect of reducing memory bandwidth with Reverse Cuthill McKee (RCM) reordering is also measured in section 6.5.3.

1.4.3 Implementation and Benchmarking of two generic Linear Algebra architectures

This thesis includes details of the implementation of two SMVM architectures which are suitable for use with any large matrix calculations. The SPAR architecture is an existing SMVM architecture [17]. From now on the SPAR architecture will be referred to as just SPAR. The SPAR had not been implemented on an FPGA prior to the FIAMMA project. The architecture computes the SMVM between a X-vector which is held in cache and a matrix which is streamed from memory. On large matrices this process often resulted in a lot of cache misses. The second architecture known as the SCAR architecture was designed to address the problems inherent to the SPAR. The SCAR architecture is a block row based solver; it breaks the matrix into strips and then each strip into multiple tiles. The SMVM of the matrix is calculated on a per tile basis. Since the X-vector and Y-vector entries associated with the block fit in the local cache this method alleviates problems associated with Y cache misses.

The two architectures are benchmarked calculating the PageRank vector of real IA matrices. Results are given for both floating point and fixed point solutions. The performance of these generic architectures is compared with results obtained from a modern GPP.

1.4.4 Implementation and Benchmarking of specialised PageRank hardware

The final architecture presented is a specialised architecture for the PageRank problem. The unique structure of the matrix used in the PageRank calculation allows for some changes in the PageRank algorithm. The multiplication stage of the SMVM can be removed from the SMVM and carried out as a dense vector operation. Doing this means the SMVM can be removed and replaced with a ‘pattern addition’². This reduces the amount of data needed to be streamed from memory for each operation and so can increase the parallelism of the calculation and thus increase architecture performance. This architecture was designed and implemented as part of the work presented in this thesis.

The PageRank architecture was benchmarked by calculating the PageRank vector with real IA matrices. The results were obtained for both floating point and fixed point implementations. These results were used to compare the architecture with a modern GPP.

1.5 Publications

Some of the work presented in this thesis has been published and presented in a number of international peer reviewed conferences.

The initial idea was proposed at the 3rd International Workshop for Applied Reconfigurable Computing in Brazil in 2007. It was subsequently published in Springer Lecture Notes for Computer Science, Vol. 4419 [18].

The hardware system for large linear algebra problems was presented at the Field Programmable Logic 2007 (FPL 2007) conference in Amsterdam [19]. This paper contained results for the SPAR connected to DDR on FPGA.

Some of the benchmarks from Chapter 6 were presented in the Parallel Computing conference 2007 (PARCO 2007) in Aachen, Germany and were subsequently published in [20].

² Pattern addition is the phrase coined for an SMVM unit without a multiplier.

More of the benchmarks from Chapter 6 were presented in the Field Programmable Logic conference 2008 (FPL 2008). This included a discussion on the overall architecture and the novel specialised PageRank architecture [21].

1.6 Thesis Guide

This thesis is presented in 7 Chapters. Chapter 1 is this introduction. In Chapter 2 the technologies being used are discussed. This information is presented to aid understanding of the work performed for this thesis. The chapter starts with a brief review of the hardware design flow. This outlines the process of designing hardware from the concept phase to the generation of the bit file for the FPGA. This is followed by a discussion on the FPGA architecture and logic. The development board used in this project is then introduced. The memory hierarchy of the development platform and the GPP used to benchmark the test architectures is discussed. Finally a brief introduction to numeric representations is offered.

Chapter 3 gives a general overview of Internet Search algorithms. This is broken down into the key operations needed for Internet search, crawling, indexing, ranking and returning a query. A number of Internet ranking algorithms are reviewed and the mathematical operations explained. The chapter concludes with a discussion on Sparse Matrix by Vector Multiplication (SMVM) and matrix storage formats.

Chapter 4 gives an overview of other work that is related to this thesis. This chapter is divided into a number of sections in which the state of the art in Internet search, SMVM and arithmetic units on FPGA is discussed.

In Chapter 5, the implementation of this project is discussed with details of hardware and software developed for this project. Three hardware units for performing SMVM are outlined and the advantages and weaknesses of both are highlighted. One of these SMVM architectures was designed especially for the PageRank algorithm as part of this project. Chapter 5 concludes with a discussion of the PageRank software used to control the hardware units.

In Chapter 6 the results of the benchmark results are presented. This chapter begins by introducing the IA matrices used for benchmarking. The performance of IA matrices on an Intel Xeon Woodcrest is then presented. The FPGA architecture

performance results are split into three main sections. These are floating point performance on Virtex-II, fixed point performance on Virtex-II and floating point performance on Virtex-5. The Chapter also includes a discussion on PageRank precision and performance limitations..

In Chapter 7 the results of this thesis are discussed. The contribution of the work is presented and some ideas for future work are highlighted.

Chapter Two

2 Technology Background

2.1 Introduction

The process of creating a suitable architecture to calculate the PageRank vector requires an understanding of several technical subjects. A working knowledge of these subjects and related technologies is essential in the understanding of the design problems and solutions. In this section, a number of these key technologies are explored. This discussion includes a general overview of the technologies, with special regard given to how they are used in this particular project. The discussion begins with an overview of the hardware design flow in section 2.2. The basics of FPGA based technology, including the details of the development board used to prototype and test the architectures developed for the PageRank problem, are then presented in section 2.3. The PageRank problem has very large storage requirements, due to the scale of the matrices and vectors involved. Thus, commodity memory in the form of Double Data Rate SDRAM (DDR SDRAM), or equivalent, is the only viable option. This memory technology is discussed in section 2.4, with special regard to FPGA compatibility. The memory hierarchy of a state of the art General Purpose Processor (GPP) which uses SDRAM memory is also described as the performance of the FPGA solutions are benchmarked against its performance. The chapter concludes with section 2.5, which provides a brief introduction to number representation in computers.

2.2 Hardware Design Flow

The FPGA hardware design flow is depicted in Figure 2.1 (based on a diagram from [22]). This design process begins with an initial concept or hypothesis. The concept is then modelled at the behavioural level and the model tested to prove viability and function. Behavioural modelling can be performed in any high level language, such as C or Matlab. Modelling the system before implementation provides an opportunity to refine the concept; it also establishes a reference against which the hardware behaviour may be compared in the later stages of the hardware design process. In this project, a fully functional Matlab model was created of the PageRank algorithm. This model was used with the real IA matrices to verify the PageRank algorithm and to identify the hardware units needed to solve the PageRank problem. The Matlab model also was used to estimate the actual hardware performances.

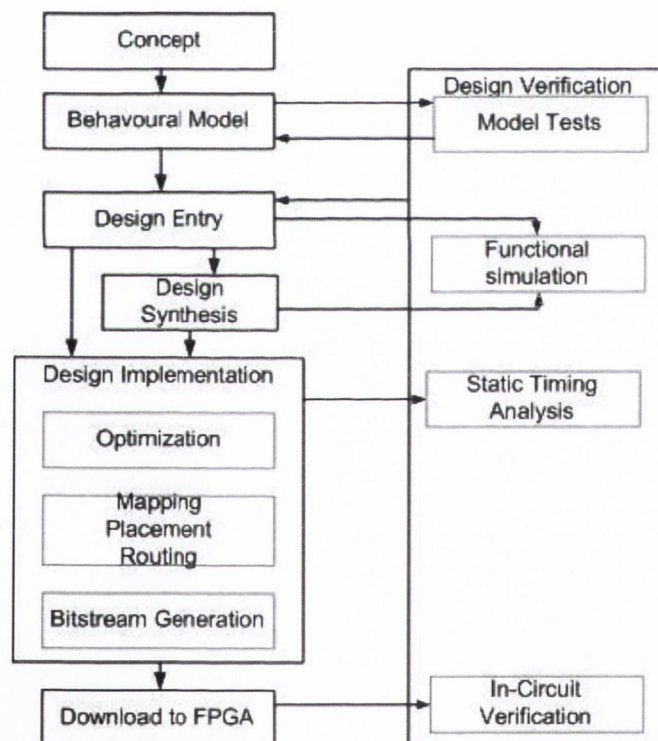


Figure 2.1 Design flow for FPGA design, based on [22]

Every stage of the design process has a corresponding testing phase. The design entry was done using System C [23] and converted to Verilog with the SC2V tool [24]. System C is a C based Register Transfer Level language which allows designs to be fully integrated with C test-benches. RTL simulations and testing were done both using a C test-bench and a Verilog test-bench after the code had been converted

using the SC2V tool. The design was synthesised using Synplify Pro [25] and implemented into logic using the Xilinx ISE [26] and EDK [27] tools version 8.1. The design was then downloaded onto an FPGA and tested. A combination of a Matlab front-end and a C program running on the MicroBlaze on board the FPGA were used for testing. This software is discussed in more detail in section 5.8.

2.3 Field Programmable Gate Arrays (FPGAs)

A Field Programmable Gate Array (FPGA) is a user reconfigurable device which can be programmed to implement a hardware circuit [15]. FPGA based prototype hardware is significantly cheaper and faster to develop than ASIC based prototype hardware. The FPGA will not achieve the same clock rate achievable by ASIC hardware but it does have a number of advantages over ASIC hardware. The FPGA can be reprogrammed in the field or after deployment to compute different functions quickly and easily, unlike ASIC, which can usually only perform a single task. The FPGA also shortens the time taken in the design implementation stage of the hardware design flow since hardware can be written to the FPGA in seconds without expensive equipment. ASIC designs on the other hand must be fabricated from silicon which is a long and costly process. The FPGA also allows for changes to be made to the hardware after it has been configured which means that much of the FPGA testing can be done in hardware. If a problem is found it is easy to reconfigure with the correct hardware. ASIC designs on the other hand cannot be changed once fabricated and thus a great deal of testing must be done before fabrication to ensure that the circuit operates correctly. ASIC fabrication is very expensive and large order quantities are needed to reduce the price. The result is that FPGAs lend themselves to low production prototype circuits. It is for these reasons that the FPGA is used in this project.

All circuits created on an FPGA are comprised of just five main reconfigurable elements as illustrated in Figure 2.2. These elements are the Input/Output Blocks (IOB), the Configurable Logic Blocks (CLBs), the Block RAM memory modules, the multiplier modules and the Digital Clock Manager blocks (DCM) [15]. These elements are connected together in various configurations to create the desired circuit

via programmable switch matrices which are represented by the spaces between blocks in Figure 2.2.

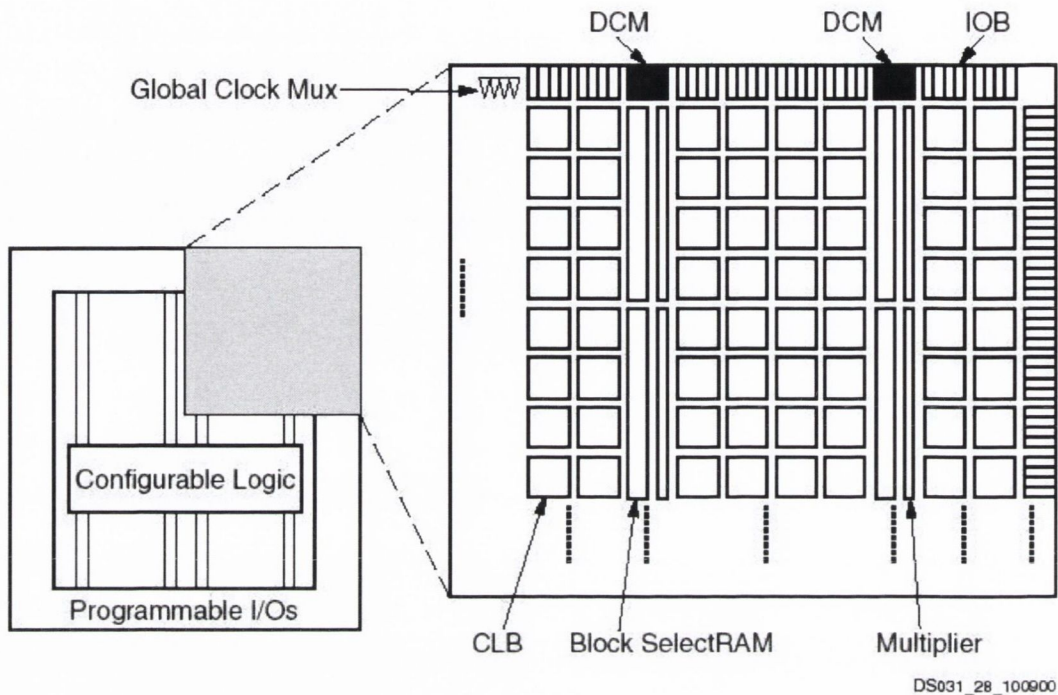


Figure 2.2 FPGA Architecture Overview (Virtex-II) [15]

The IOBs are arguably the most important components of the FPGA as they are used to link the circuit implemented on the FPGA to the pins of the FPGA. The pins can then be connected to external devices, such as memory, Ethernet, or PCI connections. The IOBs are highly versatile and support many different I/O standards allowing the FPGA to communicate with many devices. The IOBs have no knowledge of how to communicate with devices connected to the FPGA until they are configured. The designer's responsibility is to ensure that the IOB is set up correctly to communicate with the device connected to the pins of the FPGA.

The DCMs can be used to correct a delay caused by clock distribution, to multiply or divide the clock signal or to shift the clock phase. This is especially important in large designs to ensure that the registers in the circuit are all clocked at the right time.

The block RAM modules provide small 18Kb of dual-port RAMs. These on board RAMs can be used to store data needed by the design. This feature reduces the number of CLBs being used as memory and thus increases the size of the designs that will fit on the FPGA. Integer Multipliers also consume a lot of FPGA area when

implemented using standard CLBs. For this reason, dedicated onboard 18x18 integer multipliers are provided as part of the FPGA fabric.

The CLBs form the biggest part of the FPGA fabric. Internally, the CLBs in Virtex-II FPGAs contain four slices, two tri-state buffers and a switch matrix; see Figure 2.3. The slices inside the CLB can communicate via a dedicated internal communication channel. If the slices need to connect to slices outside the CLB, they must do this through the switch matrix. The tri-state buffers can be used if a slice in the CLB needs to drive a bus.

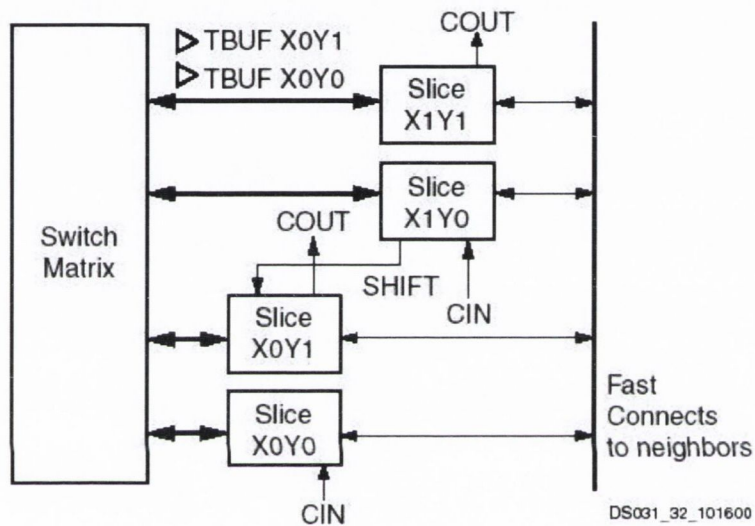


Figure 2.3 CLB Element (Virtex-II) [15]

On a Virtex-II FPGA, each slice contains two registers, two 4-input function generators, Carry logic (CY), arithmetic logic gates and function multipliers, as shown in Figure 2.4. The 4-input function generator can be configured as a 4-input LUT, 16-bit shift register element or 16 bits of distributed RAM. The registers can be used in conjunction with the function generators. They can also be separately configured to use inputs that bypass the function generator. This process does not affect the function generator, which can still be used in the design [15].

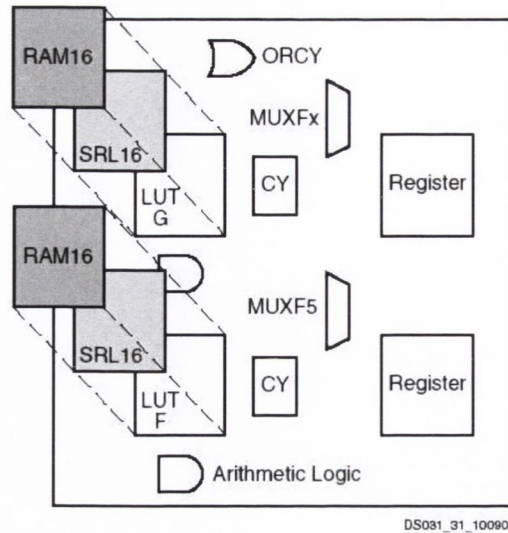


Figure 2.4 Slice Configuration (Virtex-II) [15]

2.3.1 Development board

The development platform used in this work is the Alpha-Data ADP-DRCII [28]. The development platform supports Virtex-II FPGAs and is populated with the Virtex-II 2V6000 (FF1152 package) FPGA device (speed grade 6). Figure 2.5 shows a block diagram of the development board used. Its major features include four independent SDRAM memory banks (populated with DDR-266), programmable clocks, 2MB of DDR SRAM, a PCI Interface Chip (PLX 9656 ASIC) with a 66MHz, 64-bit PCI bus and some programming and control logic.

The development platform plugs into the PCI slot of a PC. While the PLX target device supports a 64 bit PCI interface, the local bus interface (i.e. the interface between the PCI target device and FPGA) is only 32 bits wide. The net result is that the bandwidth of this interface is only 264MB/s. In this implementation this is something of a bottleneck, severely limiting the rate at which data can be transferred from the PC memory to the FPGA platform memory.

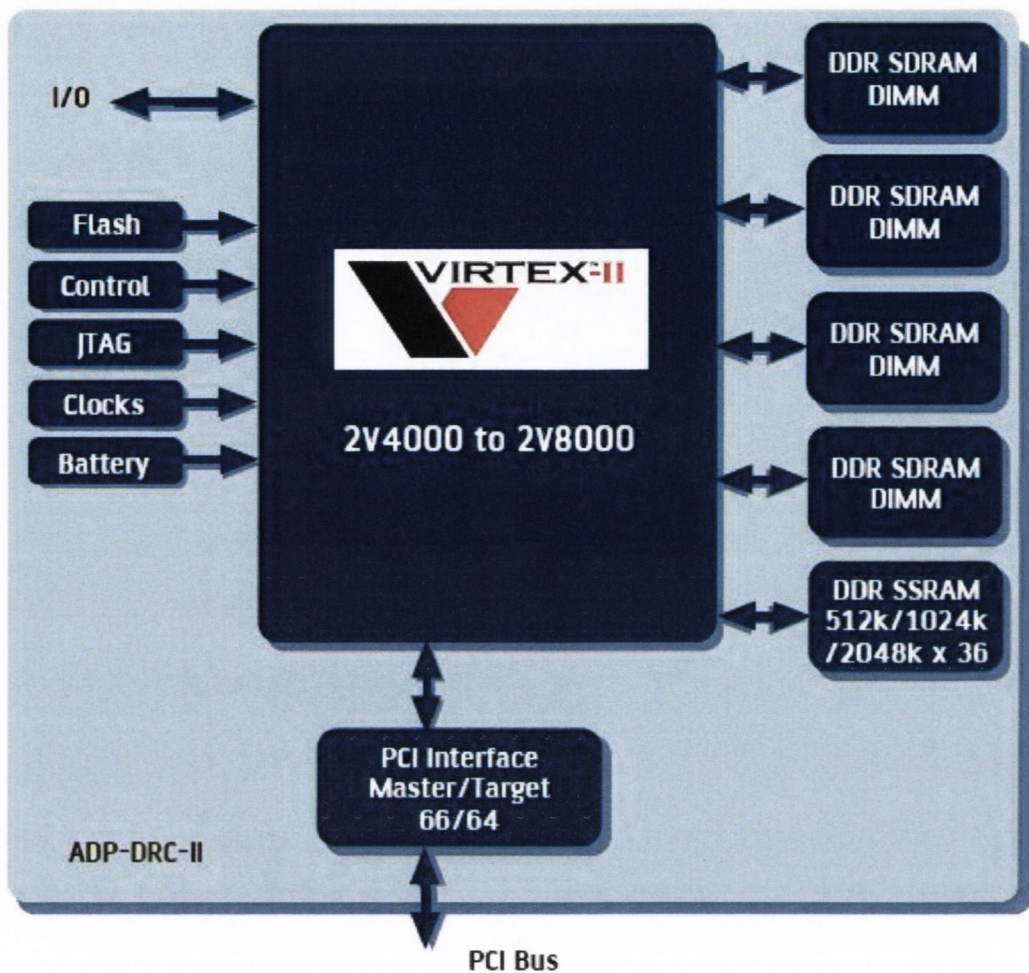


Figure 2.5 Alpha-Data ADP-DRCII board [28]

However, the most important factor in the choice of this platform for this development was the large memory bandwidth – provided by four independent SDRAM memory spaces. Additionally, as the intention was to develop an iterative solver the data transfer time could be amortised over the total solution time and does not prove to be a major hindrance.

A platform supporting PCI-Express would have been ideal but no suitable one was available at the start of the project (or even still as far as the author is aware).

2.3.2 Virtex-5

The Virtex-II FPGA is now several generations old. Xilinx have introduced Virtex-4 and Virtex-5³ devices more recently. These more recently introduced FPGAs have

³ Virtex-6 has just been announced at time of writing

larger capacities, cost less and can achieve higher clock rates. A summary of some of the Xilinx FPGA families introduced since Virtex-II are shown in Table 2.1.

Table 2.1 Xilinx Virtex family compared (* multipliers replaced with extreme DSP slices, † Slices in Virtex-5 contain four LUT and four flip-flops)

	Virtex-II [15]	Virtex-II Pro [29]	Virtex-4 [30]	Virtex-5 [31]
Process Geometries	90 nm	90nm	90nm	65nm
Slices	256-46,592	1,408-44,096	6,000-89,088	4,800-51,840†
Multipliers	4-168	12-444	32-96*	32-192*
RAM (Kbits)	72-3,024	216-7,992	864-6,048	1,152-10,368
I/O	88-1,108	205-1,164	320-960	400-1,200
Introduced	2000	2002	2004	2006

Virtex-5 is the newest of these families and was introduced in 2006 and there are some notable differences between the Virtex-5 and the Virtex-II used on the development platform. The 18x18 multipliers used in Virtex-II have been replaced with Extreme DSP slices which contain an 18x25 multiplier [32]. The basic slice architecture has also been changed. Figure 2.6 shows a Virtex-5 slice which consists of four LUTs and four flip-flops [33].

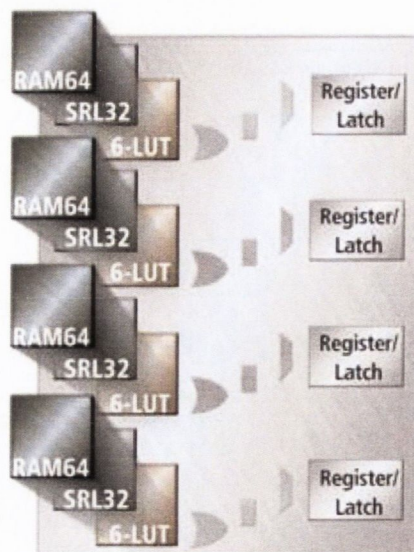


Figure 2.6 Virtex-5 Logic Block

Table 2.1, clearly shows that the Virtex-5 family provides a great deal more on chip Block RAM (BRAM). The Virtex-5 family can support DDR2 and DDR3 memory

and so offers the opportunity to increase memory bandwidth. However, a suitable Virtex-5 (or Virtex-4) development board for the PageRank problem was not available. For this reason the Virtex-II FPGA platform was used.

If a suitable development platform was created for Virtex-5 a number of tasks would need to be carried out to port the hardware designed for the Virtex-II development board onto the newer device. The hardware modules written as part of this project can be ported easily but all Xilinx generated hardware would need to be reconfigured. The hardest module to port would be the memory interface which was created with the Xilinx Memory Interface Generator (MIG) [34]. Time would also need to be taken to reassign pins on floor planning of the design

2.4 Memory - Double Data Rate SDRAM (SDRAM)

The large size of the PageRank problem limits the choice of memory available to use in the problem. Often in FPGA based architectures, on board BRAM is sufficient to store the data. This situation is not the case, however, in the PageRank problem. Static RAM (SRAM) modules could be used to store the data, but they become prohibitively expensive as the problem size grows. Commodity memory in the form of Double Data Rate SDRAM (DDR SDRAM) is scalable and cheap. Many different types of SDRAM memory exist, which are often classified by the amount of data words they output per second. For example, DDR-266 produces about two hundred and sixty six million 64-bit (8 Byte) words per second.

Table 2.2 shows a number of different DDR memories along with their clocking rates, peak transfer rate, latencies and whether or not they are supported by the Virtex-II or Virtex-5 FPGA. DDR2 and DDR3 are newer versions of DDR and can achieve higher clock rates than DDR memory.

Table 2.2 A summary of DDR data rates [35]

Device name	Bus clock rate MHz	data width (bytes)	peak transfer rate MB/s	Memory Latency (cycles)	Virtex-II support	Virtex-5 support
DDR-266	133	8	2128	2-2.5	yes	yes
DDR-333	166	8	2656	2-2.5	yes	yes
DDR2-400	200	8	3200	3	no	yes
DDR2-533	266	8	4264	4	no	yes
DDR2-667	333	8	5336	5	no	yes
DDR3-800	400	8	6400	7	no	yes

The newer DDR memories have a higher clock speed, yet also have an increase in memory latency. The memory latency measures the time taken between a memory request to the memory and data being returned. Memories with larger bandwidths also have larger latencies. These latencies have minimal effects on large data transfers, but decrease the performance on short bursts of data.

The memory controller has a huge effect on performance of SDRAM. The effects are specific to the memory hierarchy and controller being used. In the next two sections, the difference between the way the FPGA uses DDR and the way the PC uses DDR is explored. The effects of the memory hierarchy and controller on SDRAM performance in these specific architectures will be discussed further in section 2.4.3.

2.4.1 PC Memory Hierarchy

The benchmarks presented in this thesis were obtained using a Dell 590 server. At the heart of this PC is the Intel Xeon (Woodcrest) processing chip which is a 65nm technology [36]. Internally the Woodcrest contains two processor cores running at 3 GHz, as shown in Figure 2.7. Each of these cores has a 32 KB L1 data cache. A 32 KB cache is small and translates to approximately 4000 double precision numbers. Thus, the core can compute small problems very quickly. If the problem cannot be stored in L1 cache, the two cores have access to a shared 4 MB L2 cache. Data can be shared between the two processors using this L2 cache [37]. The L2 cache is connected via a 1333 MHz Front Side Bus (FSB) with a bandwidth of 10.6 GB/s to the Northbridge memory controller. The controller is connected to four banks of fully buffered DDR2-667 (FBD). Each FBD has a bandwidth of 5.33 GB/s and so overall, the memory controller has a bandwidth of 21.3 GB/s. This memory bandwidth

cannot be utilised by the processor since the FSB can only achieve a maximum bandwidth of 10.6 GB/s. The FSB is therefore the bottleneck in the PC memory system. The memory controller allows for a second FSB to be connected to it, which would use the memory bandwidth available. However, in the tests run as part of this thesis the second processor socket is not populated.

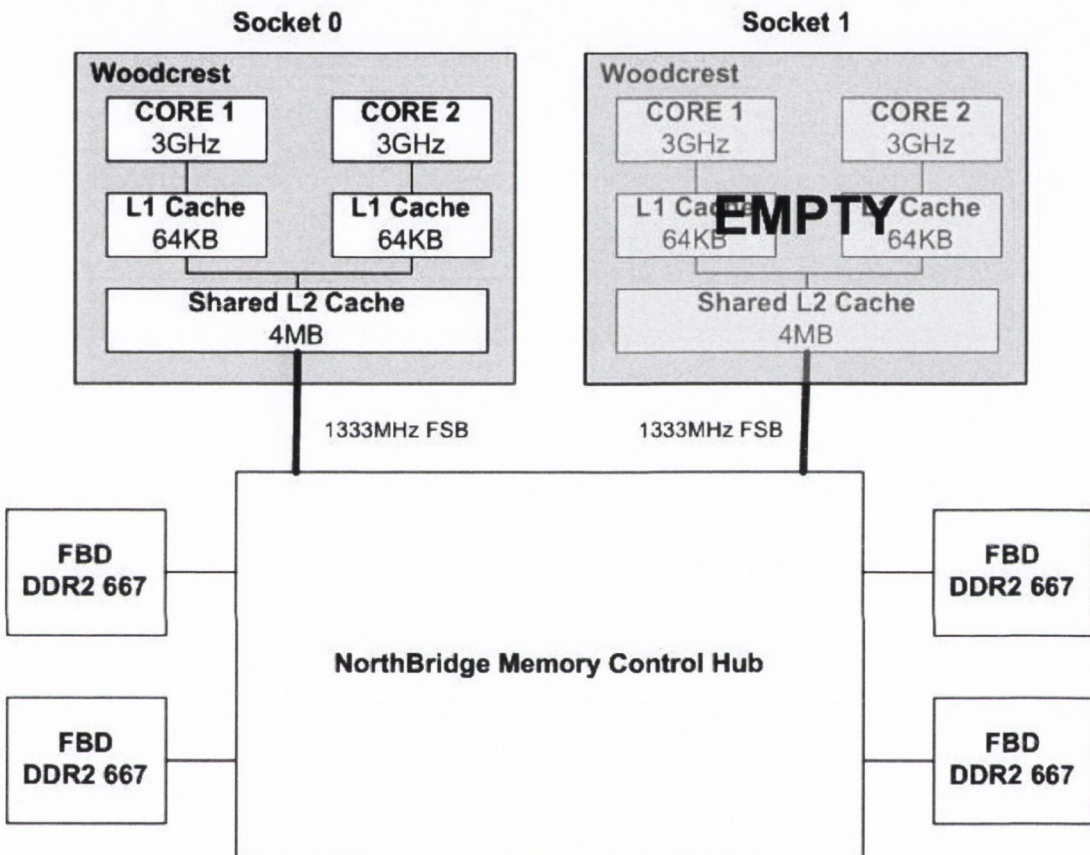


Figure 2.7 Intel Xeon Woodcrest memory architecture, based on [38] (L1 cache is 32KB data & 32KB Instruction)

2.4.2 FPGA Memory Hierarchy

The FPGA has a major advantage over the PC as regards memory interfaces/bandwidth. The FPGA can support multiple banks of SDRAM in parallel. The development platform used in this project implements four independent banks of DDR-266 memory. The FPGA can stream data from all four of these DDRs simultaneously. DDR-266 has a peak data rate of 2128 MB/s (see Table 2.2). Thus,

the FPGA platform has a maximum memory bandwidth of 8512 MB/s. Virtex-5 FPGAs can support DDR-800, which has a memory bandwidth of 6400 MB/s⁴.

Some Virtex-5 FPGAs have enough IO pins to implement four (or more) banks of DDR, yielding a memory bandwidth of 25.6GB/s when used with DDR3-800.

Memory bandwidth, however, is not the bottleneck for FPGA computations, as is the case with general purpose processors. The clock speed is often the bottleneck in FPGA designs. It is often difficult to design hardware for the FPGA logic to run fast enough to fully utilise the memory bandwidth. Table 2.3 shows the clock rates that are required to fully utilise the DDR memory bandwidth. The clock rate varies with the number of bits that the processing elements on the FPGA require per clock cycle. For example, a circuit that needed 64 bits (8 Bytes) of data streamed from DDR-266 memory would require a clock rate of 266 MHz in order to fully utilise the memory bandwidth. However, if the circuit used 96 bits (12 Bytes) per clock cycle the clock rate would fall to 175 MHz.

Table 2.3 FPGA clock rates needed to utilise maximum memory bandwidth in systems of various data word sizes

Device name	peak transfer rate MB/s	Clock rate (MHz) for payload of:		
		8 Byte	12 Byte	24 Byte
DDR-266	2128	266	177	88.6
DDR-333	2656	333	218	109
DDR2-400	3200	400	266	133
DDR2-533	4264	533	355.5	177.8
DDR2-667	5336	666	444	222
DDR3-800	6400	800	533	266

The clock rates for the 8 Byte and 12 Byte data transfers are almost impossible to achieve for large circuits. The clock rates for the 24 Byte transfer look more reasonable. This subject is discussed again in section 5.5.3, when the details of the architecture are discussed.

⁴ It is worth noting that at the time of writing only very high end, highest speed grade devices can support this speed grade of memory.

2.4.3 FPGA and PC Memory Hierarchy Compared

The benchmark PC, like all PCs, has a complex memory hierarchy with three levels of varying speed memories. The L1 and L2 cache have low latencies, high clock rates and thus, have large bandwidths albeit very limited in size, 32 KB for L1 cache and 4 MB for L2 cache. The caches are clocked at the same speed as the processor. The caches are linked by the FSB to a Northbridge chip which is responsible for communicating with the SDRAM. The result of this complex memory hierarchy is a memory system whose performance varies with data size and data reuse. Table 2.4 shows the latency of the PC memory for varying data sizes as measured by the SANDRA benchmarking suite [39]. The latency increases as the data size outgrows the L1 and L2 cache. The larger data forces the PC to fetch data from its slower memory. There is an obvious increase in latency as the data outgrows each cache. 16 KB is the largest data size that will fit in L1 cache. The drop in bandwidth between the L1 and L2 cache is caused by the L2 cache's higher latency. The limit of the L2 cache is also evident after 4 MB. Larger data sizes are accommodated by SDRAM which is much higher latency and lower bandwidth than the L1 and L2 caches. The reduction in bandwidth evident between the L2 cache and the SDRAM is caused by both increased latency, FSB cycles, memory controller cycles and the slower clock rate of the SDRAM.

Table 2.4 SDRAM latency of FPGA and Intel Woodcrest

Block Size	Intel Woodcrest (DDR2-667)		FPGA (DDR-266)	
	Clock cycles 3 GHz	Time (ns)	Clock cycles 100MHz	Time (ns)
1 KB	3	1	33	330
4 KB	3	1	31	310
16 KB	3	1	32	320
64 KB	14	4.7	30	300
256 KB	16	5.3	32	320
1 MB	16	5.4	32	320
4 MB	55	18.6	31	310
16 MB	289	96.7	34	340
64 MB	329	110	32	320

In contrast to the complex PC memory hierarchy the memory hierarchy on the FPGA can be implemented in what ever form best suits the problem. The PC memory hierarchy performs well for problems with a large quantity of data reuse. SMVM has

little data reuse - this will be discussed in greater detail in section 3.6. For this reason a different memory hierarchy was implemented on the FPGA; see section 5.2. The FPGA memory architecture is based on a streaming data system. Data that will not be reused is streamed from memory directly to the processing elements. The BRAM on the FPGA is used as buffers (simple L1 cache) for the vector elements that are reused. This simple streaming architecture reduces the overhead and complexity associated with having L1 and L2 caches. Removing this overhead and complexity result in the memory latency for all data sizes remaining constant; see Table 2.4. The longer latency time of the FPGA is due to the slower clock-rate of the memory.

The memory architectures of the two systems are also evident in the bandwidth measurements. The three tiered memory architecture of the PC is reflected in the three distinct steps in the memory bandwidth; see Figure 2.8. The top most tier of the graph in Figure 2.8 represents the L1 cache. The L2 cache bandwidth is evident for memory blocks between 64 KB and 4 MB. After this point the PC must access information from SDRAM via the Northbridge chip. The memory bandwidth decreases greatly when data is stored in the SDRAM. This decrease in bandwidth can be attributed to lower memory bandwidth of SDRAM when compared with the L1 and L2 caches, FSB limitations and memory controller overhead. The theoretical peak bandwidth of the DDR2-667 SDRAM used in the Intel Woodcrest is 5336 MB/s per socket. The Northbridge supports four banks of memory giving a theoretical memory bandwidth of 21344 MB/s. The PC sustains a mere 2775 MB/s which is 13% of this value in the Sandra bandwidth tests [39].

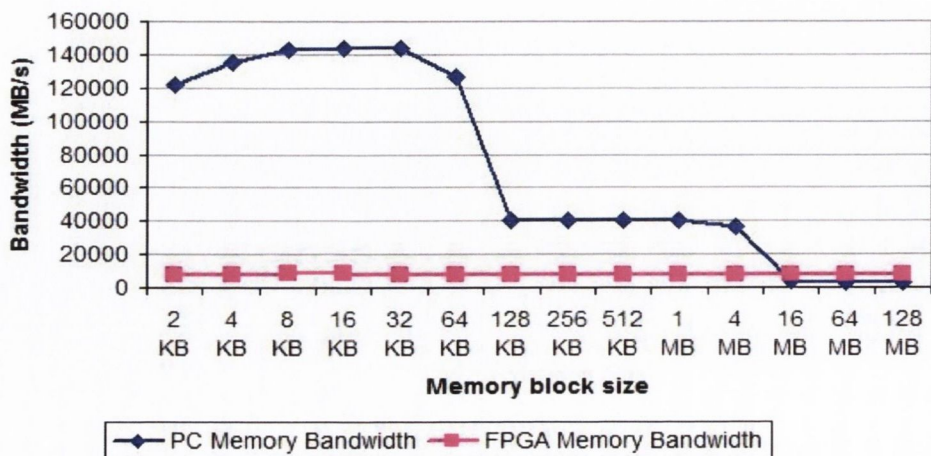


Figure 2.8 Memory Bandwidth of the Xeon Woodcrest and FPGA for various memory block sizes.

The FPGA exhibits a constant memory bandwidth. In memory copy tests carried out on the FPGA with data blocks of varying size, the FPGA achieves a sustained 7660 MB/s or 90% percent of its 8512 MB/s peak. The FPGA, thanks to its efficient use of memory, has a larger sustained memory bandwidth than the PC despite using memory that is less than half the speed. Memory bandwidth is very important since large matrices associated with Sparse Matrix by Vector Multiplication are memory bounded.

2.5 Number representation

There are many ways to represent numbers on computer systems. These include integer/fixed point and floating point. The simplest form is integer/fixed point format. This format can be used to represent any whole number or fractional part. However, it is difficult to achieve a large dynamic range as the number of bits is limited. An integer is simply the number saved in binary format using a certain number of bits. A 32-bit integer uses 32 bits to store the number and a 64-bit integer uses 64 bits to represent the number. The range of a number format is a measure of the amount of numbers it can represent. It is calculated by taking the smallest number from the largest number and adding 1. In the case of an integer the range is simply 2^b where b is the number of bits used to store numbers in the format. The precision of a number is the number of significant bits the representation can accommodate. Integers cannot represent the fractional part of a number. Scientific calculations like the PageRank algorithm need a number system can represent fractions. In this section, two formats for storing these numbers are discussed. They are fixed point and floating point numbers.

2.5.1 Floating point

A Floating Point (FP) number is made up of three parts, namely the sign, the exponent and the mantissa; see Figure 2.9 [40]. If the number is negative, the sign bit is 1. Otherwise, the sign bit is 0. The exponent gives the position of the number in the numerical systems range. The mantissa contains the fractional part of the number. The number is normalised so that the most significant 1 is just to the left hand side of the binary point sign. This 1 does not need to be stored, since it is implied.

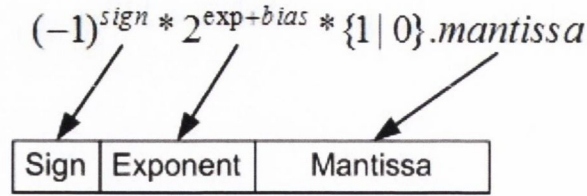


Figure 2.9 Floating point number

The IEEE floating point standard gives a standard approach for handling floating point numbers. This approach includes information on precision, range, rounding modes, exception handling, overflow and underflow [40]. Thus, consistent behaviour is given across compliant processors. Before the standard was introduced in 1985, many different ways of dealing with FP numbers existed. This led to a great deal of confusion as programs run on different machines often gave different answers. A floating point standard is needed to ensure everyone running a calculation will get the same answer.

2.5.1.1 Precision and range

Two main floating point formats are outlined in the IEEE floating point standard. These two formats are single and double precision. Double precision floating point has a larger range and precision than single precision floating point. The range is limited by the number of exponent bits available and the precision is governed by the number of mantissa bits in the format. Table 2.5 shows how the single and double precision formats differ. The double precision format has a larger number of bits in its exponent, and so can cover a greater range than the single precision format. The double precision format also has more digits in its mantissa and so can represent a number to a great precision than the single precision format. The double precision format can represent a number with 53 significant digits. These 53 significant binary digits encompass the 52 fractional binary digits and the implied 1 which is included because the number is normalised to be between 1 and 2. The single precision format gives 24 significant binary digits of precision.

Table 2.5 double and single precision floating point

Type	exponent	mantissa	bias	max value	min value
single	8 bits	23 bits	127	$(1-2^{-23}) * 2^{127}$	2^{-126}
double	11 bits	52 bits	1023	$(1-2^{-52}) * 2^{1023}$	2^{-1022}

Numbers between the minimum value and zero can be achieved using denormalized numbers. Denormalized numbers increase the range, but do so by decreasing precision. A denormalized floating point number is one in which the exponent is zero. This factor signals that the number is no longer normalised, and so no longer contains a 1 left of the binary point. The largest denormalized number is just below the minimum value of the normalized numbers quoted in Table 2.5. Each time the number gets a degree smaller, it loses a bit of precision until the precision finally becomes zero. The IEEE floating point format includes a representation of infinity; thus if a number is too large to be represented by the scheme, it is treated as infinity. The IEEE floating point format allows for other non-arbitrary formats to be used. Any combination of exponent length and mantissa length can be used to suit a user's needs so long as it is made conform to the standard regarding rounding modes, exceptions, and denormal numbers as outlined in the IEEE standard [40].

2.5.1.2 Rounding modes and exceptions

The IEEE floating point format includes 4 rounding modes. These modes are round to zero, round to negative infinity, round to positive infinity and round to nearest even. All IEEE-compliant arithmetic units must support these formats. In order to fully support the format, arithmetic units must also support a number of exceptions. The IEEE floating point format includes an exception code word which must be allowed to propagate unhindered through the arithmetic units. An underflow, overflow and divide by zero flag must also be available. The divide by zero flag is set when a division by zero operation is attempted. The overflow flag is set if a number gets too big to be represented by the format while being operated upon. The underflow flag gets set if precision is lost, due to the number becoming too small to be represented or becoming denormalized when being used in a calculation.

2.5.2 Fixed point

Fixed point arithmetic gets its name from the fact that the binary point always remains in the same fixed place. The digits before the binary point are referred to as the Magnitude (M) and the digits after the binary point are called the Fraction (F). The maximum value that can be represented using unsigned fixed point arithmetic is

given by $2^m - 2^{-f}$ where m and f are the number of bits used to represent M and F respectively. The upper limit for signed fixed point arithmetic is given by $2^{m-1} - 2^{-f}$ and the minimum value is given by -2^{m-1} . Table 2.6 shows the range and precision of four fixed point number systems.

Table 2.6 Fixed point range and precision

Type	Magnitude	Fraction	precision	max value	min value
Unsigned 32	2 bits	30 bits	32 bits	$2^2 - 2^{-30}$	0
Signed 32	2 bits	30 bits	32 bits	$2 - 2^{-30}$	-2
Unsigned 64	2 bits	62 bits	64 bits	$2^2 - 2^{-62}$	0
Signed 64	2 bits	62 bits	64 bits	$2 - 2^{-62}$	-2

The range and precision of the fixed point numbers in Table 2.6 are very different than the range and precision calculated for the floating point numbers in Table 2.5. A floating point number has a much larger range than the fixed point number of the same size. However, the fixed point number has a greater precision than the equivalent floating point number. For example, a single precision floating point number has a dynamic range of 2×2^{127} and a precision of 23 bits. A signed 32-bit fixed point number has a maximum dynamic range of 2×2^{31} (depending on f) and a precision of 32 bits. Floating point numbers are designed to cover a large range, while fixed point numbers cover a much smaller range, albeit with greater precision. Fixed point arithmetic is not as complicated as floating point to implement on FPGA and can usually run at a higher clock speed than its floating point counterpart. If the number used by an algorithm falls into a small range, then fixed point arithmetic is often the best choice of number format. If a large range of numbers is needed, however, then fixed point is limited in its use and floating point arithmetic should be used.

2.6 Summary

In this section a number of key technologies were discussed. First of all, a brief description of the hardware design flow was presented. Details of the fabric of the two FPGAs used in this research were also presented. The development platform uses a Virtex-II FPGA. Although this FPGA is several generations old no suitable replacement development board has been created using the Virtex-5. Virtex-5

performance data can be obtained from the Virtex-II performance results together with post place and route timing information. The performance data is extrapolated to the clock rate on the newer device. Virtex-5 performance data will give a truer picture of the achievable performance for the PageRank architecture on a state of the art FPGA.

Next, the memory issue was addressed. The PageRank vectors and matrix are very large, and so it is essential to use commodity memory. The development platform with four banks of DDR memory provides the necessary memory capacity and bandwidth. The memory bandwidth is often the limiting factor in calculations like the PageRank algorithm. However, modern FPGAs can support very high speed DDR3 memory, and so achieving a clock rate to fully utilise the memory bandwidth is a serious design issue.

Finally, a discussion of number representation in scientific computing was presented. Floating-point arithmetic is complex to implement on FPGA but the large range it offers is often required by applications. Floating point arithmetic also requires the hardware engineer to implement special case exceptions and rounding modes. Historically, FPGAs have implemented fixed point arithmetic. Fixed point is limited in range but offers good precision. The choice between floating and fixed point comes down to the range and precision needed by the PageRank algorithm to distinguish between the multitude of pages it ranks.

Chapter Three

3 Internet Search

3.1 Introduction

All large collections of information are indexed to allow them to be searched effectively. For this reason, books have indices, libraries have catalogues, and bookshops have inventory lists. Likewise, as the Internet began to grow, it became obvious that it too needed to be indexed. Search engineers thus applied well-established search techniques to the Internet so that it could be searched. These techniques were clearly shown to be insufficient in early search engines, as users had to wade through pages of Spam results to get the page they had sought in a web query. These poor results were caused by spammers, who purposely wrote pages to manipulate search engines results. They added hidden text (e.g. white text on white back ground), added fake description tags and stuffed their articles full of commonly used search keywords all in order to bring unhelpful web pages to the fore. The result was that Internet users were frustrated in their searches for helpful information and web pages.

The unique difficulty of the Internet is that it is different from other document collections in ways which make it difficult to search. It has four key distinguishing features that complicate search methods. These four features are that the Internet is huge, dynamic, self-organised and hyperlinked. For these reasons, traditional search methods like boolean search algorithms, were destined to fail on the Internet [41].

First of all, the Internet is huge and thus its size alone prevents many traditional search techniques from being used. Any search engine needs to be scalable to the ever-increasing size of the Internet. The Internet is not only huge, but its size is not accurately known. In 2003, Langville said Google had ten billion pages in its index

[41]. In 2006, Austin claimed Google had twenty five billion pages in its index [42]. In 2007 Google announced that they had indexed their one trillionth page [6]. Even though these estimates are very different, they give an idea of the scale of the Internet and possibly the rate of growth. The Internet is a data collection of billions of documents, many of which have frequently changing content.

Secondly, the Internet is dynamic. Studies have shown that 40% of web pages change weekly and that 23% of the “.com” domain web pages change daily [43, 44]. This feature also represents a big change from traditional document collections, where documents remain unchanged once they are added to the collection.

Thirdly, the Internet is self-organised. In a traditional data collection, one trained body of staff collect and organise the data. With the Internet, however, no one central person or body organises or controls the Internet. Pages are added, removed and updated regularly.

The last and most obvious way that the Internet differs from a traditional data collection is that it is hyperlinked. A hyperlink is a digital link from one webpage to another which when clicked sends the user to the referenced page. Hyperlinks are what makes “web surfing” and modern search engines possible. These four properties of the Internet must be considered when creating a successful search engine.

Figure 3.1 is an overview of the major components of a search engine. The search engine can be broken down into four processes, namely crawling, indexing, ranking and query processing. In this chapter, these four processes are discussed. Although many of the details of crawling, indexing and query processing are outside the scope of this project, a short simplified description of each is given as applicable to this project. The second part of this chapter is focused on the ranking process. A number of ranking algorithms are discussed in detail in this section.

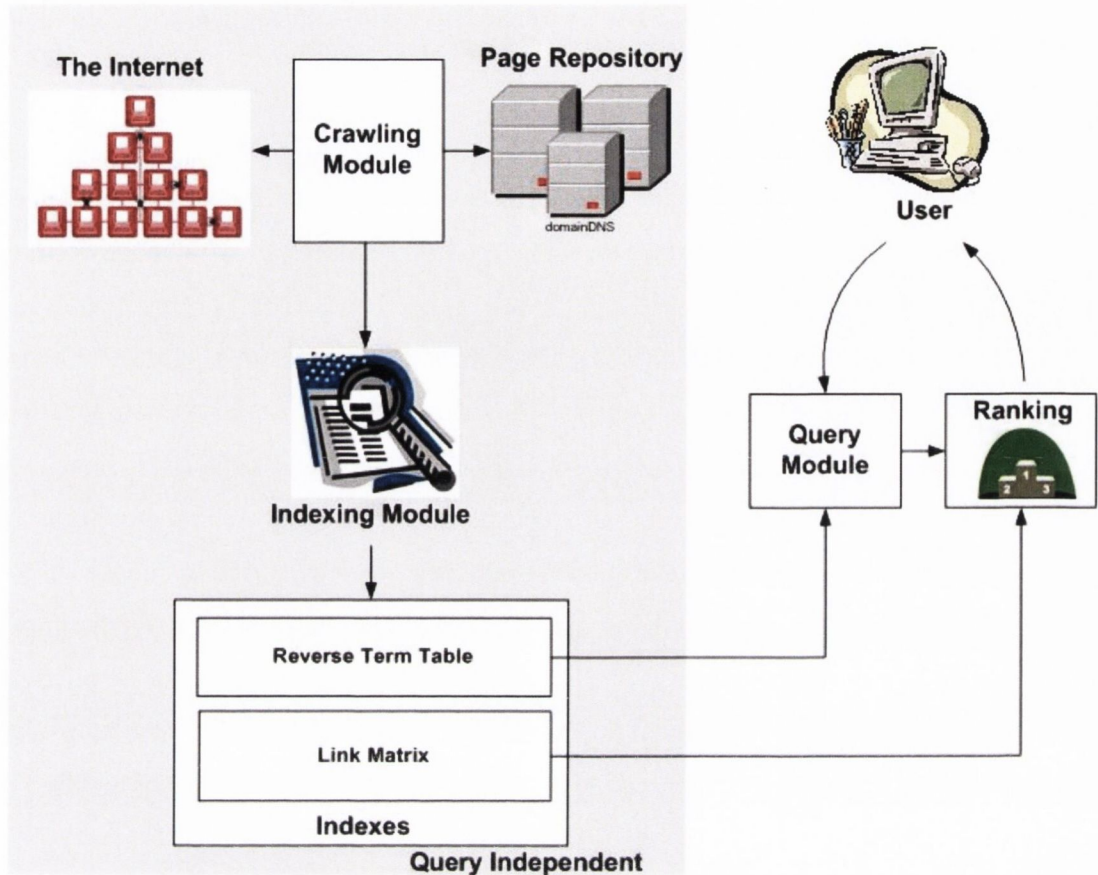


Figure 3.1 Overview of a Search Engine

3.2 Crawling

In the previous section, the Internet was described as a highly volatile environment. The structure and content of the Internet are constantly changing. The scale of the Internet makes it impossible for any human to keep search engines up to date with the newest web pages. No central index exists to tell search engines when a page is updated or a new page is added to the Internet. The search engine indexes thus are kept up to date by Internet crawlers.

The process begins when Internet crawlers, or ‘spiders’, are given lists of URLs to visit by the Crawler Module [42]. The spiders visit the pages and return the text content to the Page Repository and the Indexing Module. The Crawler Module parses the new content looking for URLs. When a URL that has not been visited recently is found, the Crawler Module adds that URL to the list of URLs that the spider has yet to visit. Thus, the spider spreads its accesses out across the Internet, crawling from

one site to another, adding new pages to its index and refreshing its index for pages that have been updated.

Internet crawling is a continuous process. New pages and updated content ensure that the Internet crawlers perpetually have plenty of work. Since the spiders visit pages, they consume the bandwidth of the websites they visit; they take that bandwidth from the “paying public”. Sites are often limited in their bandwidth. This bandwidth can be thought of in two different ways. The bandwidth of a site is the number of page requests that a site’s server can deal with or the maximum amount of data that the server can transfer before it can no longer keep up with page requests. If search engine providers did not limit their web crawling, they could clog up the Internet by requesting too many pages too quickly. Yet, if search engines did not continually crawl websites, the search engine would not have the newest version of pages and not be able to return the most relevant results. Thus, a balance must be found. Search engine providers are continually trying to find the right balance between crawling too much and crawling too little, and thus between index freshness and bandwidth consumption.

The search engine company Google has a novel approach to this problem. In order to take the guesswork out of figuring out when was the right time to crawl, Google created a service called Webmaster Tools [45]. The service simply asks the people who make the sites (webmasters) to estimate how often each page changes on their site. The webmaster uploads an XML file containing information about every page on his or her site, including an estimate of how often it is updated. This service also gives the Webmaster the option of limiting the crawl rate of his/her site (Figure 3.2), to save bandwidth for his/her visitors. Using this service is not mandatory to be included on the Google index, but does help to ensure the Google spiders find your site and crawl it regularly.

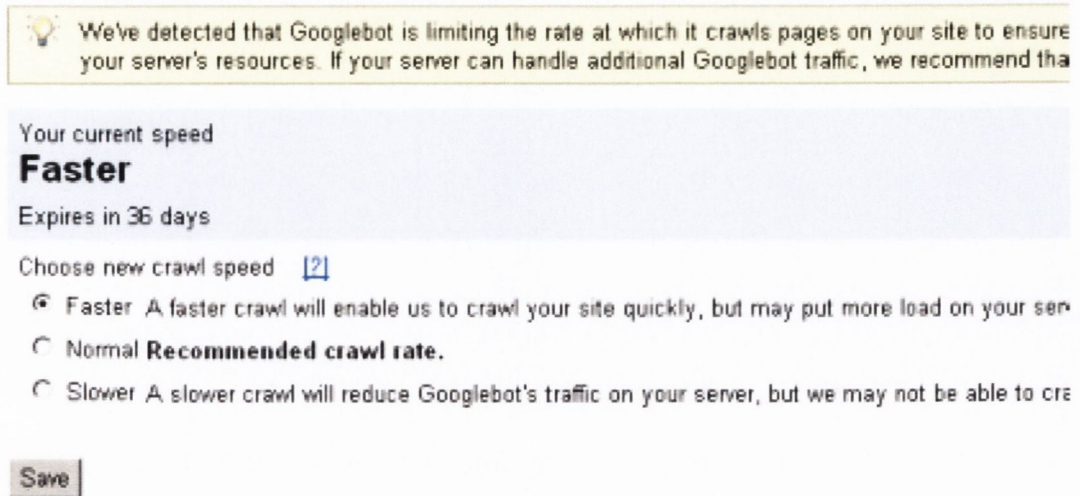


Figure 3.2 Google Webmaster controls

3.3 Indexing

Freshly crawled content is fed into the Indexing Module [41]. This module has two functions. It must both create an index of the page content, referred to as the Reverse Term Index (RTI), and it must create the Internet Adjacency (IA) matrix, as in Figure 3.3. Each new URL crawled is assigned a number to identify it. The IA matrix is created using these identity numbers as co-ordinates. Figure 3.3 shows how the IA matrix is constructed from a simple web. Note that each page is represented with a number. The actual URLs are stored in another index.

Non-zeros in the IA matrix column represent hyperlinks to the page corresponding to the column number (inlinks). Non-zeros in this IA matrix rows represent hyperlinks that page has to another page corresponding to the column number (outlinks). The IA matrix in Figure 3.3 shows that the page with index number 1 contains a hyperlink to pages 2, 3 and 5, and pages 3 and 4 contain hyperlinks to page 1. This IA matrix is central to the ranking algorithms and so will be discussed in more detail later.

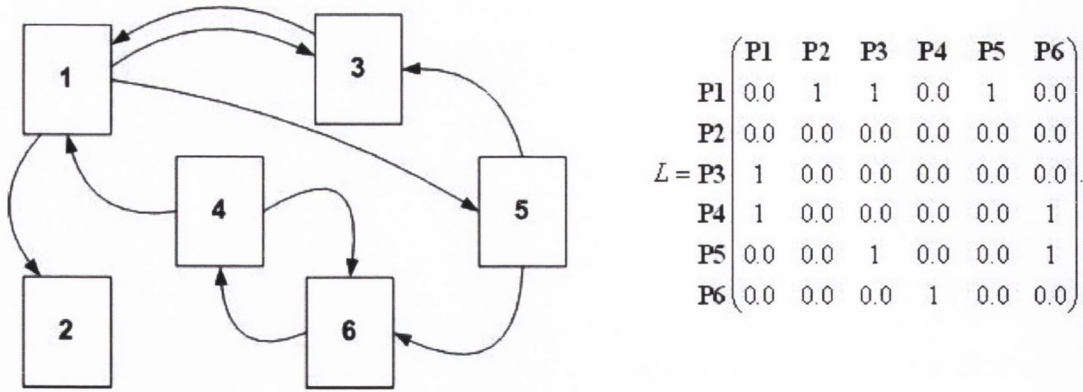


Figure 3.3 Link matrix created from crawled pages

The second function of the Indexing Module is the creation of the Reverse Term Index (RTI). The Indexing Module parses the content of each page that the spiders return, removing unnecessary information and very commonly used words. Figure 3.4 shows a simplified version of what the Indexing Module does to create the RTI. The RTI is similar to an index at the back of a book. Every keyword is listed, along with a list of pages on which that keyword occurs.

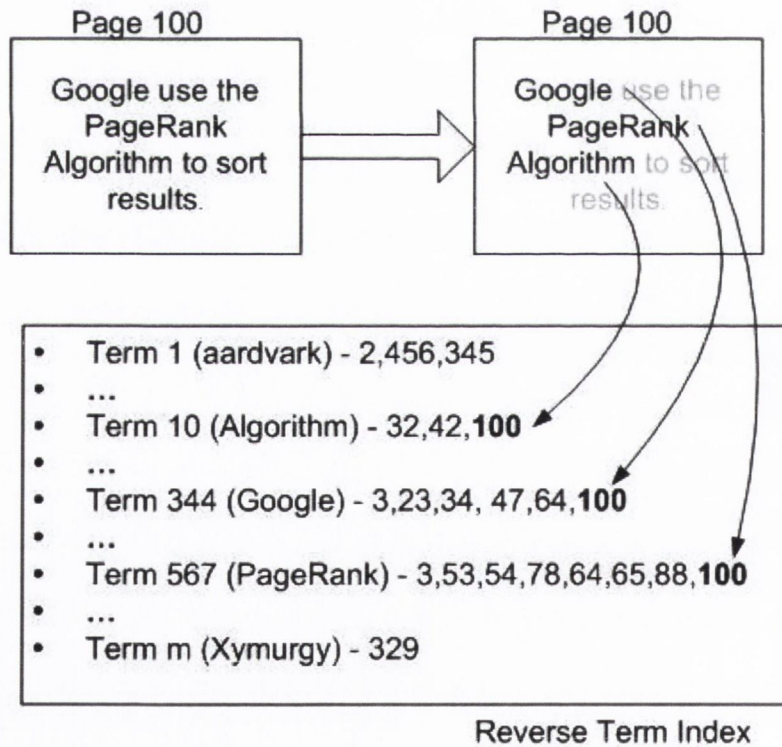


Figure 3.4 Reverse Term Index creation

Every page on the Internet that uses a particular word can be found by consulting the RTI. Figure 3.4 demonstrates that the word “Google” appears in six pages of this simple example. It appears on page 3, 23, 34, 47, 64 and 100. RTIs are commonly

used in information retrieval [41]. The size of the Internet and its variety of content does cause some problems for the RTI creation and management. It consumes a great deal of memory, as it must index numerous languages, names and phrases. The RTI is not as simple as portrayed in this example. Other information regarding the reference is also stored (e.g. is the word in the page title, in a heading or in bold font etc.). Significant time and resources are spent researching the best way to store and implement the Reverse Term Index.

3.4 Query Processing

Internet crawling and indexing occur before the user makes a query. They are query independent. The previous two modules create the indexes that the query module uses to return results to the user. The Query module is query dependent. This means that the results of this process are affected by what the user searches for. It is important that any calculations carried out in this process are done quickly because the user is waiting for the result. The query module is responsible for three duties, the RTI lookup, content score calculation and returning the ranked results to the user [41].

The first step is use the RTI to find all the pages in which the query keywords appear. For example, if we search for “Google AND PageRank” using the RTI from Figure 3.4, we would see we have 6 pages with the word “Google” in them and 8 pages contain the word “PageRank”. However, only 3 pages contain both words. The pages are page 3, 64 and 100. The second step is to create the content score. This score is worked out using the other information in the RTI. Many different factors make up the content score [46]. For example, the frequency and position (title, header or body text) of the keyword in the document affects the overall content score. These other factors are unimportant for the purposes of this example. Some search engines only use the content score to rank pages but others have a separate ranking module. If a ranking module exists these scores are combined with the content score to give the overall result order. The results are then returned in this order to the user.

3.5 Ranking

In this section a number of ranking algorithms will be presented and compared. To aid understanding a number of traditional document collection ranking algorithms will be presented along side the Internet ranking algorithms. The algorithms will be assessed for scalability, complexity of calculation, accuracy of results and the time the user has to wait for results. The traditional ranking algorithms refer to ranking algorithms that are used on non-linked document collections. These include Boolean Search Engines and Latent Semantic Analysis. Traditional ranking-algorithm scores are computed only using the page content. Modern Internet ranking algorithms differ from traditional algorithms as they combine a content score with a popularity score. This popularity score is calculated using the linked structure of the Internet. There are many different ways of doing this and a number of these methods are outlined in this chapter. Salsa [47], HITS [48], and PageRank [49, 50] are examples of this type of ranking algorithm.

3.5.1 Boolean Search Engines

The Boolean search engine is one of the simplest types of ranking algorithms. Many libraries use a version of the Boolean search engine. As the name suggests the Boolean search engine is based on Boolean algebra [51]. The user uses the AND, OR, XOR and NOT operators with keywords to get the results they want. The user can string as many keywords as they want using these operators, thus refining their search. The search engine looks for the presence (absence in the case of the NOT operator) of the inputted keywords and only returns the pages that comply with the user's query statement. The model does not return partial matches. Decisions are based completely on the binary criterion. This can lead to too many or too few results being returned and ultimately poor performance [51]. Users can find these systems counter intuitive, since a search for Boolean meaning of the AND operator and the OR operator are not the same as they are in daily language. A request for "cats and dogs" in daily language can mean the user wants documents about cats and documents about dogs but in Boolean logic it means only documents that contain both terms. A request for "tea or coffee" on the other hand can imply a mutually exclusive decision. You can have tea or you can have coffee but not both. In Boolean logic it means you want any document that contains the word tea or the word coffee.

This can cause confusion and often the results users received were very different from what they expected [51].

Boolean search engines also suffer from another grave disadvantage. They do not understand synonymy (multiple words have the same meaning) and polysemy (one word meaning many different things). Many search engines fail to deal efficiently with this problem. If you searched for “car AND maintenance” a Boolean search engine would fail to return a document called “automobile maintenance”. To help alleviate this problem the idea of a fuzzy Boolean search engine was developed. It uses fuzzy logic to categorise queries like the “car and maintenance” example as partially relevant [41, 51].

The Boolean search engine is scalable and versions of it were used in early Internet search engines. However, it is easy to spam. It has no concept of a document being useful other than that it contains a given keyword. A spammer could include keywords on his page that were not relevant to the page content, making his page appear in the results of high traffic searches. This was a very common spamming method before Google.

3.5.2 Latent Semantic Analysis

Latent Semantic Analysis (LSA) is an example of a vector space search model. It was introduced in 1988 to address the issues that were inherent in summarising documents and queries into a set of keywords, as is done in Boolean search engines, - namely irrelevant documents with the query keywords are returned and relevant documents that do not contain the keywords are not returned [51]. In LSA, documents and query vectors are mapped into a lower dimensional space, which is associated with themes [51]. Simplified, this mapping process works by comparing the structure and word usage in the documents to find the underlying themes of the documents. This process looks at the frequency of words used in the document and their relative proportionality to each other. Based on these factors, documents of similar themes can be identified despite individual discrepancies in exact keywords used. The hypothesis is that results retrieved from this themes based approach are better than those retrieved using keyword lookup model like that used in Boolean search engines. A simple example of this would be to take the case of two almost

identical documents about car engines. Document A uses the keyword “car”.

Document B uses the keyword “automobile”. As discussed earlier, a Boolean search engine would only return document A for a query on “car engines” as it would only analyse exact words used in the query. However, LSA should notice similarities in documents A and B, despite document B not using the query word “car.” LSA would do this by examining other words common to both documents, and noting their frequency and proportionality. Thus, LSA should see the common theme of both documents and return both document A and document B. Since document A and B have relevance to the query, by returning them both LSA performed better than the Boolean search engine.

The latent semantic search engine contains a dictionary of terms, T_1, T_2, \dots, T_n . This dictionary contains words and phrases and is in the order of a couple of thousand entries. The search engine parses each document D_j in its index and records the frequency each dictionary term appears $D_j = (freq_{1j}, freq_{2j}, \dots, freq_{nj})$ where, $freq_{ij}$ is the frequency that term T_i occurs in document D_j . This information is stored in a matrix $A_{n,m}$ see (1). Since most documents do not contain all the words this matrix is sparse. Thus, sparse matrix techniques can be used to solve it [52].

$$A_{n,m} = \begin{matrix} T_1 \\ T_2 \\ \vdots \\ T_n \end{matrix} \begin{pmatrix} D_1 & D_2 & \dots & D_m \\ freq_{11} & freq_{12} & \dots & freq_{1m} \\ freq_{21} & freq_{22} & \dots & freq_{2m} \\ \vdots & \vdots & & \vdots \\ freq_{n1} & freq_{n2} & \dots & freq_{nm} \end{pmatrix} \quad (1)$$

Singular Value Decomposition (SVD) is applied to the $A_{n,m}$ matrix as shown in equation 2 [53].

$$A_{n,m} = \bar{U} \bar{S} \bar{V}^t \quad (2)$$

This yields the three component matrices that make up the $A_{n,m}$ matrix, where \bar{U} is the matrix of eigenvectors derived from the term-to-term correlation matrix given by $A_{n,m} A_{n,m}^t$, \bar{V} is the matrix of eigenvectors derived from the term to term correlation matrix given by $A_{n,m}^t A_{n,m}$, and \bar{S} is the $r \times r$ diagonal matrix of singular values where $r = \min(n, m)$ is the rank of $A_{n,m}$ [51]. The \bar{S} matrix is truncated to the k top entries. The \bar{V} matrix and the \bar{U} matrix are also truncated to correspond to the \bar{S} matrix [53].

Figure 3.5 shows the truncation of the \bar{S} , \bar{U} and \bar{V} matrices.

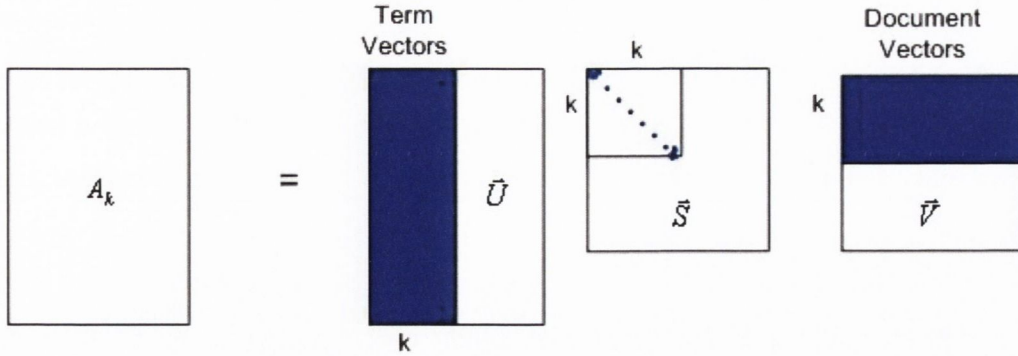


Figure 3.5 The truncation of the S,U and V matrices in LSA

This is the crucial step in LSA. If k includes too few of the S values, returned query results would be inaccurate. The LSA search engine would be unable to tell documents apart. However, if k includes too many rows of the S matrix the results returned for a query would resemble the results of a simple Boolean search engine and will not effectively deal with synonymy or polysemy [52].

A number of ways to run queries exist in LSA. Baeza-Yates et al. suggest modelling the query as a pseudo document [51]. A query vector $Q^T = (q_1, q_2, \dots, q_m)$ is created where q_i is equal to one if term T_i appears in the query string and zero otherwise. This vector is added to the A matrix as the first column. Then SVD is applied to the matrix and the $A_k^T A_k$ matrix is calculated. The first row of this $A_k^T A_k$ matrix is the rank of all documents with respect to this query [51]. This method requires a full SVD to be calculated on the document collection every time a query is made. Berry et al. [53] and Meyers [52] use a method that does not involve calculating the SVD every time a query is made. Instead the SVD is computed every time the document collection changes. This is done to compute the new A_k , which is the reduced term-document matrix. The document ranking for a query is then given by comparing the query vector Q^T to each document vector D_j (Columns of A_k). Equation 3 below shows how the cos function can be used to compare the vectors.

$$\cos \theta_j = \frac{Q^T D_j}{\|Q\|_2 \|D_j\|_2} \quad (3)$$

The user must choose a threshold τ . The results where $|\cos \theta| > \tau$ are returned to the user as relevant documents. The choice of τ is as Meyer puts it “part art part science” [52].

Latent Semantic Analysis is a powerful tool for finding the underlying semantics in a document collection. The quality of the results is high provided the k and τ values have been wisely chosen. LSA, however, is not suitable for huge, dynamic, self-organised and hyperlinked document collections like the Internet. First of all, LSA is computationally expensive for huge document collections as it requires comparing every query to every document. Furthermore, LSA necessitates computing a SVD each time a document changes, further adding to computation costs for a dynamic document collection. The huge computations associated with LSA make it the perfect candidate for hardware acceleration. This avenue of inquiry was not pursued in favour of examining the most popular form of Internet ranking algorithms.

3.5.3 Internet Ranking Algorithms

Modern Internet ranking algorithms exploit the unique linking structure of the Internet when calculating page rankings. Before the advent of these hyperlink dependent-ranking algorithms, searching the Internet for useful information had become a rather frustrating task. The search engines of the day often returned pages and pages of spam results before a single useful page could be found. It was clear that something had to be done or the Internet could never achieve the dreams of its creators of becoming the world's largest useful information collection. So, in the late 1990s the door was open for anyone who could product highly relevant search results and cut down on the spam results being returned by search engines. Solutions came in the form of two ranking algorithms, HITS and PageRank, which both used the Internet's linked structure to determine the order in which pages should be returned for a query. Leading modern day search engines such as Google, MSN search and Ask still use the linking structure of the Internet to compile webpage rankings [50, 54, 55]. The precise details of these algorithms are closely guarded trade secrets. However, the initial versions of PageRank used by the Google search engine and the HITS algorithm used by Ask are available. In the following two sections the HITS and PageRank algorithms will be discussed.

3.5.4 Hypertext Induced Topic Search (HITS)

Jon Kleinberg developed HITS in 1997 [48]. To understand HITS one must visualise the web as a graph. Every page in the web is a node on the graph and directed arrows on the graph represent hyperlinks between pages. Figure 3.6 shows a simple example web. This type of graph is called a neighbourhood graph.

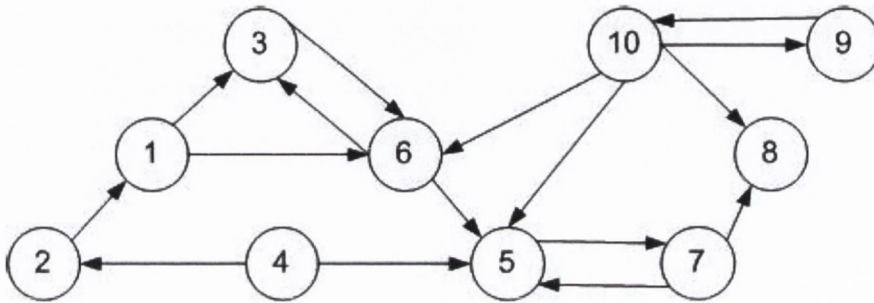


Figure 3.6 Example of a 10 Node Neighbourhood graph

A node that has several inbound links is called an authority. It is thought of as an authority because multiple other websites reference its material. A node with several outbound links is called a hub. A hub directs web users to sites where the information can be found, see Figure 3.7.

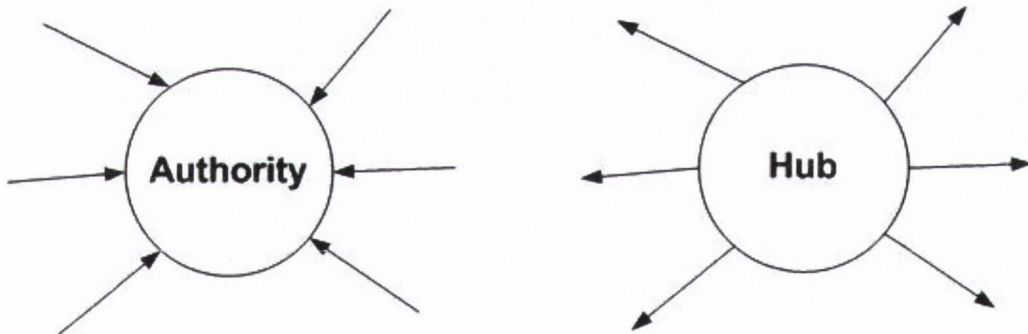


Figure 3.7 An Authority node and a Hub node

The HITS thesis is that good authorities are pointed to by good hubs and good hubs point to good authorities [48]. Many pages on the Internet are both hubs and authorities since they have incoming and outgoing links. In the HITS ranking algorithm a hub and authority score are calculated for every node on the web. Pages with highly relevant content, to a given query, receive a high authority score. Pages that contain many links to useful pages for a given query get a high hub score. Both results are returned to the user and they choose to order the results with either the hub or authority scores. The scores are useful for different things. For example, if one was searching for a camcorder. If the particular model was known, the authority

score would return highly specialised pages about that particular model. However, if one just wanted to conduct a broad search about camcorders in general the hub score would return pages that linked to many relevant informative pages, which would allow the search to be broadened to find what was needed. Equation 4, shows the original equations used by Kleinberg to calculate the HITS ranking algorithm, where every page (i) has both an authority score x_i and a hub score y_i [48]. Let e_{ij} be a directed link going from page i to page j (i.e. a hyperlink going from page i to page j). An initial estimation of authority score $x_i(0)$ and the hub score $y_i(0)$ must be assigned. HITS iteratively refines these estimates using the two equations in Equation 4

$$\begin{aligned}x_i(k) &= \sum_{j:e_{ji} \in E} y_j(k-1) \\y_i(k) &= \sum_{j:e_{ij} \in E} x_j(k-1) \\k &= 1, 2, 3, \dots\end{aligned}\tag{4}$$

It is clear from the equations for HITS that the authority score of a page is the sum of all the hub scores of the pages that link to it. Likewise the hub score of a page can be determined by summing all the authority scores of the pages it links to. These equations can be written in matrix form. This is shown in Equation 5, where L is a partial link matrix obtained as described in the indexing Section 3.3. This will be discussed later.

$$\begin{aligned}X(k) &= L^T Y(k-1) \\Y(k) &= LX(k)\end{aligned}\tag{5}$$

Equation 5 can be simplified to the form shown in Equation 6 by simple substitution. These two new equations define the iterative power method for computing the dominant eigenvector for the matrices LL^T and $L^T L$. Kleinberg suggested that this method could be used to solve the equations [48]. There is no need to solve both equations using the power method. Once one of the scores has been computed the other can be obtained by back-substitution into Equation 5.

$$\begin{aligned}X(k) &= L^T LX(k-1) \\Y(k) &= LL^T Y(k-1)\end{aligned}\tag{6}$$

The HITS algorithm can be divided into three parts. Firstly a neighbourhood graph is created, then the link matrix L is obtained from the neighbourhood graph and finally

the authority and Hub scores are calculated. When a query is made the HITS based search engine must create a neighbourhood graph. There are various ways of doing this but one simple way to do it is to consult the RTI as described in section 3.3.

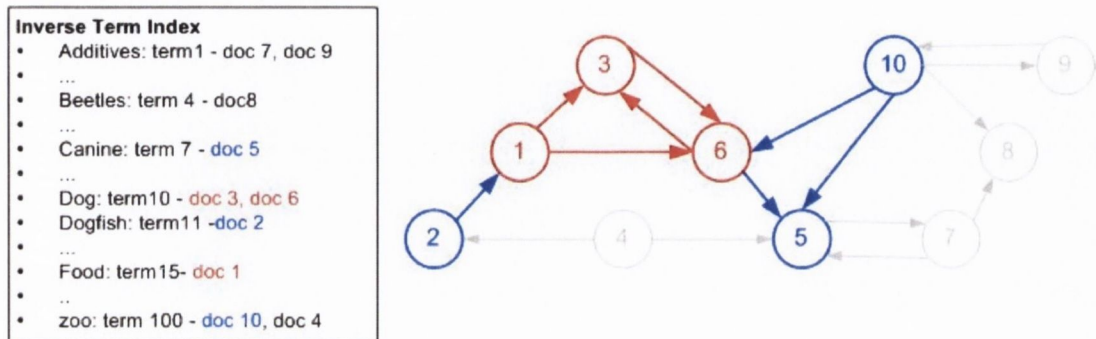


Figure 3.8 Building a neighbourhood graph in HITS

If a query for “dog food” was run on the RTI in Figure 3.8, the pages 1, 3 and 6 would be returned. The initial neighbourhood graph would be created using only these results (shown in red). The neighbourhood graph would then be expanded by adding pages that link directly to the existing pages in the graph. In the example above that adds pages 2, 5 and 10 to the graph (marked in blue). This padding of the neighbourhood graph is done to allow some semantic associations to be made. This usually resolves the problem of synonyms [41]. However, it can also add unrelated pages to the search. This is evident in this example; neighbourhood graph padding has added a page about the zoo (page 10) and a page about dogfish (page 2), which are two unrelated pages to the subject of “dog food”. However by adding page 5 it has added a page on canines which is a related article. A limit is placed on how much the neighbourhood graph can be padded because large graphs would take too long to solve. Figure 3.9 shows the final neighbourhood graph for this example.

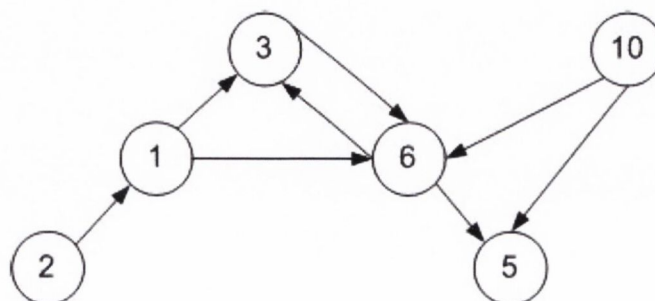


Figure 3.9 Finalised neighbourhood graph for “dog food” query

Once the neighbourhood graph has been created the adjacency matrix L can be created. The adjacency matrix is an $N \times N$ matrix where N is the number of nodes in

the neighbourhood graph. Say e^{ij} is an element in the adjacency matrix on row i and column j . If a hyperlink exists linking node i to node j then e^{ij} is set to one. If node i is not linked to node j then e^{ij} is set to zero. The adjacency matrix shows us how all the nodes in the neighbourhood graph are linked. The adjacency matrix for the neighbourhood graph in Figure 3.9 works out as follows:

$$L = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 5 & 6 & 10 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 5 \\ 6 \\ 10 \end{matrix} & \begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix} \end{matrix}$$

Figure 3.10 Link matrix for HITS neighbourhood graph

This adjacency matrix is used with Equation 6 to calculate the authority and hub scores. The version of the power method used by HITS always converges. The reasons for this are outside the scope of this thesis but it is due to the matrices LL^T and L^TL being symmetric, positive semi definite and nonnegative, so it is not possible to have multiple eigenvalues on the spectral circle [41]. Sample Matlab code for the power method used to solve the HITS ranking algorithm is given in Figure 3.11.

```
while(residual >= epsilon)
    prevx=x;
    x=x*L';
    x=x/sum(x);
    residual=norm(x-prevx,2);
end
y=x*L';
y=y/sum(y)
```

Figure 3.11 Matlab code for HITS ranking Algorithm

The HITS algorithm requires two Sparse Matrix by Vector Multiplications, one memory copy, one vector scale, one division, one vector subtraction, and one vector norm to be calculated per iteration. Research suggests that the HITS algorithm converges in about 10-15 iterations [48, 56].

HITS has a number of disadvantages. It is query-dependent, which means that the neighbourhood graph, the authority scores and the hub scores are calculated at query time (while the user waits). HITS is also susceptible to spamming. By adding links to and from your page you can influence the authority and hub scores. A slight change in score can give a page a big advantage on the Internet. Since most users only ever

view the top 10 to 20 results for a query there is great incentive to try to improve your score by link spamming. Finally, HITS can fall susceptible to topic drift. The process used to expand the neighbourhood graph can sometimes introduce an off topic page that scores very well due to the number of links it has either in or out. However, by weighting the authority and hub scores by how relevant they are to a query one can eliminate this problem. The cosine distance measure between query and page, like the one used in LSA, could be used to calculate this weight factor [57].

3.5.5 PageRank

At the same time Kleinberg was working on his HITS thesis, Sergey Brin and Lawrence Page, two PhD students in Stanford university, had teamed up to create another link-based ranking algorithm. They called their ranking system PageRank and it was the heart of their new search engine. The Google search engine started as four PCs networked together in a Stanford dorm room [58]. PageRank remains at the heart of Google to this day [7]. The PageRank algorithm for ranking hyperlinked documents has been compared to the HITS authority score [41]. However, PageRank avoids the inherent weaknesses of HITS. It is query independent. This means the page rankings are calculated before query time and they are used regardless of the query terms used. There is no need for real time matrix operations at query time. In this section the PageRank algorithm will be discussed as it was described in Page and Brin's initial paper [50].

PageRank is a popularity score for page. It is calculated for every page in Google's index. Hyperlinks to a page are considered votes for that page. However, votes from important pages are worth more than votes from unimportant pages and votes from pages with fewer out-bound links are worth more than votes from pages that link to many pages. Page and Brin described this, in their landmark paper, in terms of a "random surfer". This Internet surfer randomly follows hyperlinks on the Internet surfing from page to page. The PageRank score is the probability that the random surfer will arrive at any given page. Therefore, the PageRank vector sums to one. If this surfer is randomly following links from page to page it makes sense that the more inbound links a page has the higher the probability of the surfer landing on that page, and hence the higher the PageRank will be for that page. If a page has a high probability that the surfer will land on it, all pages to which it links will benefit from

its higher PageRank score, since the surfer must follow a hyperlink to its next page. After landing on a page the surfer must randomly choose a link to follow if there is more than one outbound link and so the chances of it going to any particular page it links to must be modified to reflect the number of out-bound links on the page. Equation 7 shows how PageRank (r) could be calculated using the random surfer model for any page P .

$$r(P_i) = \sum_{P_j \in B_p} \frac{r(P_j)}{|P_j|}$$

$$B_p = \{\text{all pages pointing to } P\}$$

$$|P_j| = \text{number of out links from page } P_j$$
(7)

Equation 7 states that the PageRank of a page is the sum of the PageRanks of all the pages linking to it, divided by the number of pages they link to. This gives each page a vote corresponding to their PageRank. Each page passes on a proportion of its vote to every page to which it links. Figure 3.12 shows how the PageRank of a very simple 3-node web is distributed (Note: sum of all PageRank scores must sum to one). Node A has a PageRank of 0.4. It has two out bound links and so passes on half of its PageRank to each of the pages it links to. Node C has two links to it each passing on 0.2 PageRank and so Node C has a PageRank of 0.4.

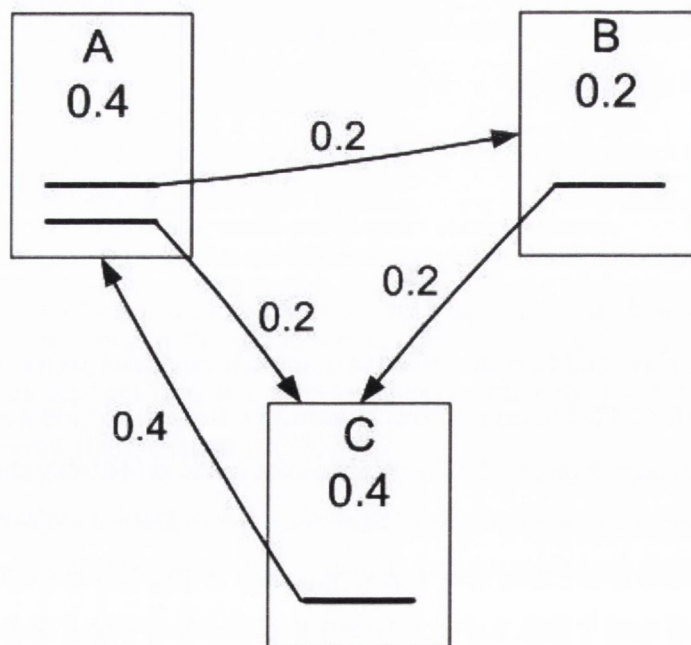


Figure 3.12 Simplified PageRank Calculation [49]

It can be seen in Figure 3.12 and Equation 7 that the PageRank calculation is a recursive method. One must know the value for PageRank to work out the value for PageRank. However, it can be rewritten in an iterative form as in Equation 8. In this process, a new estimation for PageRank is calculated on each iteration, until the PageRank eventually converges to some stable value.

$$r_{k+1}(P_i) = \sum_{P_j \in B_p} \frac{r_k(P_j)}{|P_j|} \quad (8)$$

An initial estimate for PageRank is needed to start the system off. In Brin and Page's initial paper $r_0(P_i) = 1/n$ where n is the number of pages being indexed by the Google search engine [49]. Equation 8 can be rewritten in terms of matrix operations. Firstly an adjacency matrix (H) must be created in much the same way as it was for the HITS neighbourhood graph (section 3.5.4). However, in PageRank an adjacency matrix is made for the entire web and not just a subsection of it as happens in HITS. The PageRank adjacency matrix is also column normalised. A 1 represented a link between two web pages in the HITS adjacency matrix. $1/|p_i|$ represents a link between two pages in the PageRank adjacency matrix, where $|p_i|$ is the number of outbound links on page i .

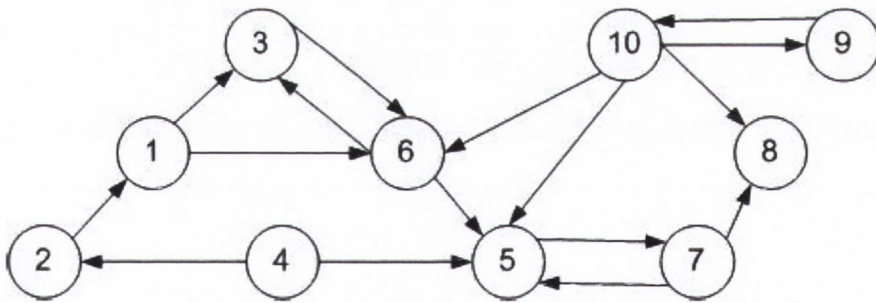


Figure 3.13 Sample Web for PageRank adjacency matrix

Figure 3.14 shows the PageRank adjacency matrix for the sample web shown in Figure 3.13.

$$H = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{matrix} & \left(\begin{array}{cccccccccc} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.5 & 0 & 0 & 0 & 0 & 0.5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & 0 & 0.5 & 0.5 & 0 & 0 & 0.25 \\ 0.5 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0.25 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.5 & 0 & 0 & 0.25 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.25 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{array} \right) \end{matrix}$$

Figure 3.14 PageRank adjacency Matrix

The equation for PageRank can now be expressed in terms of this column normalised adjacency matrix H .

$$R_j = HR_{j-1} \quad (9)$$

The PageRank equation as shown in Equation 9 is easily identified as a power method iteration to find the dominant Eigenvector of matrix H [41]. Brin and Page chose the power method to solve the PageRank problem because it was simple to implement, it converged relatively quickly and it was able to deal with the problem-size. In section 4.2.1, alternate PageRank algorithms are discussed. Brin and Page choose the power method because it was simple and has a very lower storage requirement than the alternatives. This simple form of the PageRank algorithm is not guaranteed to converge. H is a substochastic⁵ matrix; this is to say that the sum of all the elements in each column that contain values is one. However, there are zero columns. These represent pages with no outlinks and are referred to as dangling nodes. On the Internet these nodes might represent Portable Document Format (PDF) files or other non-linking office documents. Node 8 in Figure 3.13 is a dangling node. The zero columns caused by these dangling nodes are the reason why this simple PageRank algorithm is not guaranteed to converge [41]. To circumvent this problem Brin and Page extended the random surfer model to include dangling nodes [49]. They stated that when the random surfer lands on a dangling node, there is no link to follow and so is forced to type an address in the address bar and jump to some

⁵ All columns in the matrix that contain NZE sum to one, but zero columns are also allowed

other random point on the Internet⁶. They decided that the probability of such a jump should be uniformly spread over every page. This adjustment replaced the zero columns in the H matrix with a column with equal probability of visiting any page. This new adjustment to H is shown mathematically in Equation 10. A new N length vector is created called the dangling node vector (dnv). The $dvn(i) = 1$ if Page _{i} is a dangling node, i.e. corresponds to a zero column in the H matrix. The dnv is zero otherwise. In Equation 10, n is the number of rows in the matrix and e is a vector full of ones.

$$S = H + dnv \left(\frac{1}{n} e \right) \quad (10)$$

The matrix S is fully stochastic, i.e. all columns sum to one. The power method calculated using the S matrix will always converge. However, being stochastic alone does not guarantee the PageRank algorithm will converge to a unique solution regardless of initial estimation. It also cannot guarantee that it will converge quickly. The PageRank vector should be the same regardless of the starting estimation for PageRank and quick convergence is also favourable. To do this the matrix would need to be primitive. A primitive matrix is irreducible and aperiodic and thus is the stationary vector of a Markov chain. Page and Brin never discussed the Google matrix in relation to Markov chains but it has been well documented since that it is stationary vector of a Markov chain [41, 59]. Page and Brin again describe this adjustment in terms of the random surfer. The random surfer goes from page to page following hyperlinks. Occasionally the random surfer gets bored with the current page and does not follow any of its links but instead jumps to another page by typing the address in the address bar. The probability that a user will continue to follow the pages' hyperlinks is given by the Google factor α . Equation 11 shows how the S matrix is adjusted to make the new G matrix.

$$G = \alpha S + (1 - \alpha) \frac{1}{n} e^T e \quad (11)$$

The choice of α is important. Since it is a probability factor it must be between 0 and 1. Smaller α values result in the method converging quickly, but large α values give

⁶ Brin and Page did not include back button modelling (i.e. their model suggests people do not use the back button). However, Langville discusses back button modelling in [41] A. N. Langville and C. D. Meyer, *Google's PageRank and Beyond: The Science of Search Engine Rankings*: Princeton University Press, 2006.

better approximations of link popularity. Brin and Page set α to 0.85 in their experiments in 1998. The G matrix will always converge to a unique PageRank vector and so a new expression for PageRank can be given by equation 12.

$$R_j = GR_{j-1} \quad (12)$$

The G matrix is dense. It would take too long to repeatedly calculate a dense matrix by vector calculation on a matrix like this with billions of rows. Luckily the equation can be rewritten in terms of the sparse matrix H and two vector adjustments. This is a prime example of where sparse matrix computation increases performance and reduces storage requirements considerably. Consider a one billion node web. To store the G matrix would require 12×10^9 GB. However, in sparse matrix form with an average of 10 Non-Zero Elements (NZE) per column it would require just over 120GB. Equation 13 shows the final equation for PageRank. This equation always converges and does so in between 50 and 100 iterations [49, 50].

$$\begin{aligned} R_j &= GR_{j-1} \\ &= \alpha SR_{j-1} + (1 - \alpha) \frac{1}{n} ee^T R_{j-1} \\ &= \alpha HR_{j-1} + \alpha(dnv)R_{j-1} \frac{e}{n} + (1 - \alpha) \frac{1}{n} e \\ &= \alpha HR_{j-1} + (\alpha(dnv)R_{j-1} + 1 - \alpha) \frac{1}{n} e \end{aligned} \quad (13)$$

One sparse matrix by vector multiplication, two vector scales, one dot product and four scalar operations are required per iteration to compute the PageRank vector. There is also a vector subtraction and vector norm to check for convergence. Sample Matlab code for PageRank is given in Figure 3.15.

```
while(residual >= epsilon)
    prevR=R;
    R=alpha *H*prevR +(alpha*(prevR*dnv) ...
    +1-alpha)*1/n*ones(1,n)
    residual=norm(R - prevR,2);
end
```

Figure 3.15 Matlab code for PageRank ranking Algorithm

The PageRank ranking algorithm avoids the weaknesses of the HITS algorithm; namely it is query independent and due to the primitivity adjustment is guaranteed to converge to the same vector regardless of the initial estimation. PageRank is also less susceptible to topic drift because it is computed on the whole web graph instead of

just a partial web graph as is the case with HITS. The sheer size of the PageRank calculation is a major disadvantage. It has been reported that it takes a number of days to complete and is only calculated about once a month [41]. A great deal can change on the Internet in 30 days. Pages can be created, deleted and updated. The PageRank algorithm does not take these changes into account until it has re-crawled and recomputed the PageRank ranking vector. Users can often be presented with pages that are out of date. To combat this Google must rely heavily on other scores for fast updating pages like news sites and blogs.

3.6 Sparse Matrix by Vector Multiplication

Sparse Matrix by Vector Multiplication (SMVM) is the key operation in the PageRank, HITS and other Internet Ranking algorithms as seen in the preceding sections. These matrices are often very large. In the case of the PageRank algorithm, the matrix is of the order of billions of rows. The SMVM multiplication, $A\vec{x} = \vec{y}$, can be calculated in many ways [60]. One way to perform SMVM is to take each row of the matrix and perform a dense dot product calculation with the vector x . The result obtained from this dot product is the y value corresponding to the matrix row used in the dot product. Equation 14 shows this method where a_i is row i of the matrix A .

$$\begin{bmatrix} a_1^T \cdot \vec{x} \\ a_2^T \cdot \vec{x} \\ \vdots \\ a_i^T \cdot \vec{x} \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_i \end{bmatrix} \quad (14)$$

This method works well for dense matrices. However, for sparse matrices it is inefficient, since it computes the dot product of all values in the row including the zero entries. It also results in a large number of cache misses.

Another method for computing matrix by vector operations involves multiplying the Non-Zero Elements (NZE) by their corresponding X value and summing the result to the corresponding Y vector entry. Figure 3.16 shows how SMVM is calculated. y_1 is

calculated by summing the products of $x_1 a_{11}$ and $x_3 a_{13}$. This scheme reduces the storage requirements for the matrix and the number of operations being carried out for the SMVM, since zero-entry trivial-operations are not processed.

$$\begin{pmatrix} a_{11} & 0 & a_{13} & 0 \\ 0 & a_{22} & a_{23} & 0 \\ 0 & 0 & a_{33} & a_{34} \\ 0 & a_{42} & 0 & a_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} a_{11}x_1 + a_{13}x_3 \\ a_{22}x_2 + a_{23}x_3 \\ a_{33}x_3 + a_{34}x_4 \\ a_{42}x_2 + a_{44}x_4 \end{pmatrix}$$

Figure 3.16 Sparse Matrix by Vector Multiplication

There are many different schemes for storing sparse matrices in memory. They are known as sparse matrix compression schemes since they compress the matrix information into a much smaller memory space. Duff et al. give a good overview of these compression schemes in [61]. These schemes include Compressed Row Storage (CRS) and Compressed Column Storage (CCS), which are described here, as well as other schemes. These other schemes include block compressed row storage, compressed diagonal storage, jagged diagonal storage, and skyline storage [62]. These schemes usually involve storing a vector of the non-zero elements of a matrix and one or two vectors that give the address of the non-zero in the matrix. In this way, storage requirements are reduced considerably.

3.6.1 Compressed Row/Column Storage

The Compressed Row Storage scheme (CRS) uses three vectors to store the matrix. The NZE are stored in the K vector in row major format. The corresponding column numbers for each NZE are stored in the C vector and finally the L vector contains information on how many NZE are in each row. This reduces the storage requirements from $N*N$ floating point numbers to NNZ floating point numbers and $NNZ+N$ integers, where NNZ is the number of non-zero elements in the matrix. Figure 3.17 shows how the A matrix in Figure 3.16 can be represented using CRS.

$$\begin{aligned}
 K &= [a_{11} \quad a_{13} \quad a_{22} \quad a_{23} \quad a_{33} \quad a_{34} \quad a_{42} \quad a_{44}] \\
 C &= [1 \quad 3 \quad 2 \quad 3 \quad 3 \quad 4 \quad 2 \quad 4] \\
 L &= [2 \quad 2 \quad 2 \quad 2]
 \end{aligned}$$

Figure 3.17 CRS vectors of A matrix in Figure 3.16

Compressed Column Storage (CCS) is very similar to CRS. It uses 3 vectors. The K vector stores the NZE in column major format. The R vector contains the row containing the corresponding NZE and the L vector contains the number of NZEs in each column. CCS reduces the storage requirements of the A matrix in exactly the same way CRS does.

$$\begin{aligned}
 K &= [a_{11} \quad a_{22} \quad a_{42} \quad a_{13} \quad a_{23} \quad a_{33} \quad a_{34} \quad a_{44}] \\
 R &= [1 \quad 2 \quad 4 \quad 1 \quad 2 \quad 3 \quad 2 \quad 4] \\
 L &= [1 \quad 2 \quad 3 \quad 2]
 \end{aligned}$$

Figure 3.18 CCS vectors of A matrix in Figure 3.16

CRS and CCS are widely used compression schemes for sparse matrices; other compression formats can be architecture specific. The SPAR architecture for SMVM, as discussed in section 5.4, is designed to work with the SPAR compression/storage format [17]. The SCAR system for SMVM, discussed in section 5.5, also has a specific storage format.

Splitting the matrix up into multiple vectors has one main disadvantage. Multiple memory reads are now needed to access any matrix information. In the case of CRS three reads are needed. One memory read is needed for the NZE data and two other memory reads are needed to access NZE position information. This reduces the utilisable memory bandwidth.

In CRS and CCS each NZE requires 96 bits of data to be transferred to represent the NZE, a 64 bit double precision word for the NZE data and a 32 bit integer for the NZE row/column address. Furthermore, another 32 bit integer needs to be read from memory at the end of every column or row for CCS and CRS respectively. This reduces the bandwidth available to feed the multiplier and adder used in SMVM by over 33%. Thus, the maximum utilisable bandwidth is therefore only 66% the peak bandwidth available.

It is common for a GPP to achieve less than 10% of its peak performance while computing SMVM [37, 63, 64]. This cannot be entirely accounted for by the bandwidth limitation caused by sparse matrix storage formats and so SMVM performance must be limited by other factors. One major reason for this poor performance is the poor data locality of the matrix in memory [65]. There are two different types of data locality that must be considered when it comes to SMVM performance. They are spatial and temporal locality of reference. Temporal locality of reference refers to the reuse of data in a relatively short time. In SMVM problems there is very little temporal locality of reference since every NZE only gets used once in the calculation. The X and Y vectors do exhibit temporal locality of reference but this is dependent on the level of spatial locality of reference evident in the NZE of the matrix. Spatial locality of reference refers to the use of data relatively close to other data in memory. Figure 3.19 shows how the spatial locality of the matrix data can lead to data reuse of the X and Y values (the X value in this particular case). In the simple system shown in Figure 3.19 the X -cache and Y -cache can only store a single entry of the X or Y vector. The NZE distribution in the A matrix is such that the same X value is used multiple times.

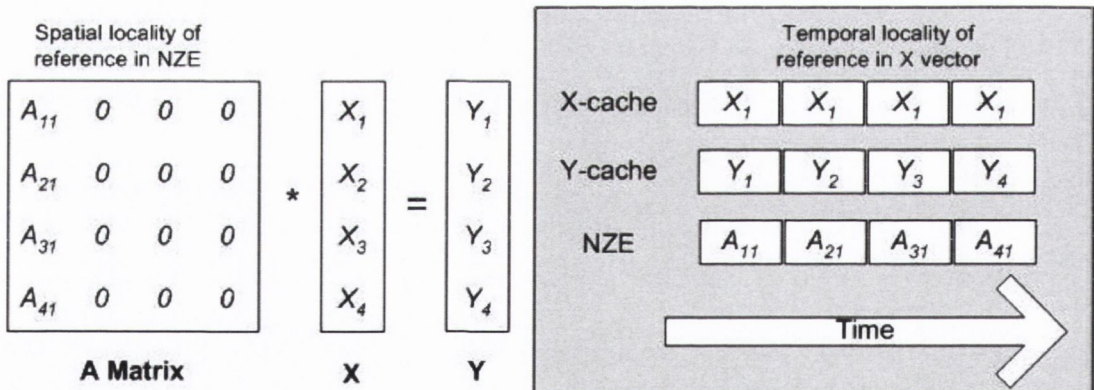


Figure 3.19 Spatial locality of reference in the NZE leads to temporal locality of reference in X and Y vectors.

Figure 3.20 shows a matrix where the NZE elements are more scattered than in Figure 3.19. In this system a new X and Y value is used for every NZE. This shows how the structure of the matrix used in the SMVM can affect the locality of reference of the data in memory. This locality of reference affects performance since more data reuse reduces cache misses and thus reduces the amount of time the system must wait for memory.

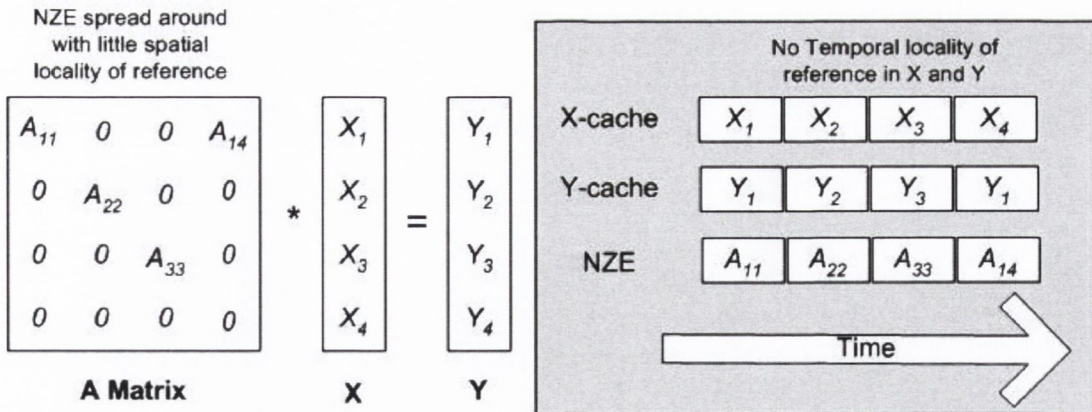


Figure 3.20 Matrix with poor spatial locality increase the number of cache misses

Poor data locality of reference increases cache miss rates, forcing the processor to retrieve data from main memory. Fetching data from commodity memory, like that used in processors today, has quite a large overhead, which greatly reduces the floating-point SMVM performance. In section 2.4.3, the latency of SDRAM in the PC was discussed. The Intel Xeon Woodcrest takes about one hundred times longer to access SDRAM than its L1 cache (see Table 2.4). The custom architectures for SMVM, like the ones described in Chapter 5, aim to alleviate this problem by using storage formats and hardware that increase the potential for data locality.

3.7 Summary

It was clear from the earliest years of the Internet that users would need some sort of index to find information. Users were accustomed to being able to consult indices to find the information they needed in other archives. Libraries and bookshops had inventories kept up to date by their trained workers. However, unlike a book shop, where the owners keep a central catalogue, the Internet has no owner. The Internet is dynamic, self organised, hyperlinked and is growing rapidly. Internet search engineers started crawling the Internet, downloading pages and indexing them to attempt to create a useful index. The Internet is volatile with 40% of its pages being changed weekly. Thus, crawling and indexing is a continuous process. The result of this endless crawling is that search engines now had a list of pages that contained keywords that users were searching for. The next problem is in what order these pages should be returned to the user. In the early days of the Internet search engineers experimented with traditional methods of search. Boolean operator search

engines were simple to build but were very hard for the user to operate efficiently to get the right results. Spamming became a major problem with these early attempts at ranking results. Spammers packed their documents full of high traffic keywords that had little or nothing to do with the page content. The result was frustrated users. Latent semantic analysis techniques would have helped make the search engines user friendly but the Internet's sheer size meant that they were limited in their use.

The answer to Internet search came in 1998 when two papers were published independently but with a similar theme. Both Jon Kleinberg's HITS algorithm and Brin and Page's PageRank algorithm took advantage of a unique feature of the Internet - hyperlinks. They allowed each hyperlink count as a vote for a page. The more web pages that link to a page the more important/useful that page is. It was a revolutionary idea and it returned relevant documents almost every time. The HITS algorithm created two scores, an authority score and a hub score. Documents with higher authority scores are deemed more relevant to a query than documents with lower authority scores. Pages with higher hub scores are more likely to link to a useful document than those pages with low hub scores. HITS is query dependent and is calculated on a subsection of the Internet. The matrix calculations at the heart of the calculations can be computed using the power method and always converge to an answer but the answer depends on the starting vector. Since HITS is only calculated on a subsection of the Internet link matrix it is susceptible to topic drift and link spamming.

PageRank seemed to tackle all the weaknesses inherent in the HITS algorithm. It does not give a relevance score because it is query independent. Instead it gives an importance score to each page on the Internet. The PageRank philosophy is that an important page is linked to by more important pages than unimportant pages are. The PageRank vector can be solved using the power method and it always converges to a single solution regardless of starting estimation. The PageRank algorithm became the heart of Brin and Page's search engine, Google, which is the most widely used search engine to this day. Brin and Page had figured out a way to search the huge, dynamic, self-organised and hyperlinked Internet efficiently and accurately.

Central to the PageRank algorithm is a large SMVM calculation where the IA matrix is multiplied by the current estimate for PageRank to compute a better estimate for PageRank using the simple power method iteration. Storing the IA matrix in its

dense format is unfeasible and so sparse matrix compression schemes are used like CRS and CCS which only store the NZE of the matrix together with NZE position information. These schemes greatly reduce the storage requirement. The performance of this SMVM operator is critical to the performance of the overall PageRank algorithm. The performance of SMVM on GPP is severely limited by memory bandwidth due to the lack of temporal and spatial data locality in the IA matrix. To increase performance of this bottleneck SMVM calculation a solution for the memory bandwidth issues must be found. One way that this could be achieved would be to implement multiple paths to memory to increase memory bandwidth. Specialised hardware connected to these multiple paths could then be used to calculate the SMVM and thus boost the overall SMVM performance.

Chapter Four

4 Literature Review

4.1 Introduction

The PageRank eigenvector problem is one of the world's most important calculations. In the words of Brin and Page, it brings "order to the web," which is otherwise a very difficult environment to search effectively. Google has invested a great deal of time and effort to ensure that the PageRank algorithm is suitable for all their needs. Unfortunately, little of this information is available to the public. No other research into PageRank algorithms running on FPGA devices has been carried out, to the best of the author's knowledge. It is difficult therefore to compare the results of this research to existing work. A number of researchers have been publishing revisions of the PageRank algorithm and a good deal of research is being carried out on SMVM on the FPGA architecture. Much of this research is targeted at structured matrices such as those common to Finite Element Analysis (FEA) problems. In this section, some of this research is presented. This includes publications on the PageRank ranking algorithm, SMVM on FPGA, General Purpose Processor (GPP) benchmarks and other scientific algorithms running on FPGA.

4.2 Internet Search

In this section, research carried out on PageRank efficiency will be reviewed. The matrix at the heart of the PageRank calculation is huge and constantly growing. It currently takes a number of days to compute the PageRank of the Google index [41]. However, as the matrix grows in size, algorithm solving time also increases. Thus, a great deal of research has been carried out with an aim of making the PageRank

algorithm run faster and more efficiently. Brin and Page's original paper used the power method to solve the algorithm. The problem size alone limited their choice of algorithm greatly. The power method is often slow to converge and so the possibility of accelerating such a commercial application sparked new research interest into this well-known algorithm. Researchers investigated many ways in which they could speed up the PageRank computation. These methods can be categorised under three generic fields; researchers could use different algorithms than the power method to compute PR, they could reduce the work being done in each iteration of the power method and finally they could reduce the number of iterations needed by the power method to converge. These methods have been applied to the PageRank problem in many different combinations with varied success.

4.2.1 Alternative PageRank Algorithms

Arasu et al., Gleich et al., Golub et al. and Corso et al. [66-69] were all involved in attempts to use other algorithms to solve the PageRank eigenvector problem. Arasu et al. proposed the use of the Gauss-Seidel method [66]. This method is very like the power method except that it uses the newer approximations for PageRank as soon as they are calculated, rather than waiting until the next iteration to use them. The main drawback with this method is that it requires the rows in the matrix to be sorted. The power method can take the matrix rows in any order. Arasu et al. suggested that the improved performance would more than make up for the time taken to sort the matrix. However, Arasu et al. only carried out tests on a single test matrix, and so this method would need further testing before its performance improvements could be proved. Arasu's paper also explained that his method uses an aggregation method, similar to the one used by Kamvar et al. [70] which will be discussed later.

Gleich et al., [69] went yet another route and took a broader look at the possible algorithms. He compared the ability of many different linear algebra algorithms at calculating the PageRank vector. In his experiments he compared the power method to the Jacobi algorithm and a number of Krylov subspace methods, namely Generalised Minimum Residual (GMRES), Biconjugate gradient (BiCG), Quasi-Minimal Residual (QMR) Conjugate Gradient Squared (CGS), Biconjugate Gradient Stabilized (BiCGSTAB) and Chebyshev Iterations [62]. The Krylov methods' performance can be improved by the use of preconditioners, so Gleich et al. [69]

investigated using parallel Jacobi, block Jacobi and Adaptive Schwarz as preconditioners. Gleich found that QMR, CGS, and Chebyshev algorithms failed to converge for the test data being used. Gleich et al. also found that the Power and Jacobi methods converged at approximately the same rate and were the most stable of all the algorithms. The Power and Jacobi iterations are actually the same for graphs that contain no dangling nodes [69]. Convergence of the Krylov methods was highly dependent on the structure of the IA matrix (see section 3.3). However, Gleich et al. do not go into detail regarding what aspects of IA matrix structure affect the convergence rate. Gleich et al. concluded that GMRES and BiCGSTAB are the best choice for the PageRank algorithm as they converge quickly for most graphs. This acceleration comes at a cost of increased memory requirements. The power method requires the matrix and three vectors of length n to be stored. The GMRES method requires the matrix and over five vectors of length n to be stored. The BiCGSTAB requires the matrix and ten vectors of length n to be stored. This increased memory requirement for a calculation that is already memory bounded may cause problems. Corso et al. [68] also experimented with a large range of algorithms including Jacobi, Gauss-Seidel and Reverse Gauss-Seidel and reordering schemes for the PageRank algorithm. Using these techniques, they managed to reduce the time taken to calculate the PageRank algorithm to 10% of the time needed by the Power method for his test set. However Corso did not take into account the time needed to reorder his test sets when this figure was calculated.

Golub and Greif came up with a way to solve the PageRank problem using the Arnoldi method [67]. They adjusted the restarted refined Arnoldi algorithm to tailor it especially for the PageRank algorithm. This adjustment was made to avoid complex eigenvalues and the sometimes slow and irregular behaviour which occurs as the algorithm eigenvalues converged to 1. Golub found that the Arnoldi method had more computations per iteration than the power method. However, for higher values of the Google factor α , the Arnoldi method required significantly fewer iterations than the Power method. Brin and Page [49] set $\alpha = 0.85$. Golub found that at this setting the Arnoldi method offered no increase in speed over the basic Power method. For $\alpha=0.99$ the power method performed 230% more iterations than the Arnoldi method. This improvement came at the cost of an increased memory

requirement. The PageRank problem is already memory hungry and so further increases in memory requirements are undesirable.

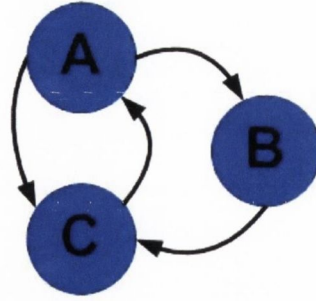
In this section, some research into alternate algorithms for solving the PageRank problem was presented. Many of these algorithms can offer performance increases over the basic power method. However, most of them come with an increase in memory requirements or require some sort of reordering to be carried out on the IA matrix. Arasu shows that Gauss-Seidel can increase performance of an ordered IA matrix. Gleich et al., show that the performance of many of the alternate algorithms is very dependent on matrix structure. Corso et al., show that large performance increases can be achieved using reordering together with alternate algorithms. However, he did not take the reordering time into account when publishing his results. The Power method gives a good overall trade-off between convergence speed, memory requirements and scalability. There has not been any conclusive proof that other algorithms are going to prove better. If performance improvements cannot be gained by changing the algorithm perhaps some modification can be made to the Power method to increase performance. A number of these modifications are discussed in the next section.

4.2.2 Modifications to Power Method for PR

A renewed interest in the Power method arose with the advent of PR, since other algorithms proved to be of limited use for the PageRank eigenvector problem. Kamvar et al's lab in Stanford published three different optimisations that can be applied to the PageRank algorithm [70-72]. The first of these papers described a method called the adaptive power method [70]. In the basic power method, convergence is judged on the aggregated error between this iteration's result vector and the previous result vector. Kamvar et al. noticed that not all pages converge at the same rate. They saw that the PageRank score for the majority of pages converged quickly and that the algorithm continued to run waiting for the last few PageRank scores to converge. They changed the convergence criterion to compare individual elements rather than computing an aggregate error (vector norm) for the whole vector. They then locked these values of PageRank and continued only to compute the values of PageRank that had yet to converge. Locking the converged values of PageRank itself had no effect on the work carried out. In order to reduce the amount

of calculations being carried out, they periodically zeroed rows of the matrix corresponding to the locked entries. This approach reduced the work carried out in every iteration of the algorithm. Using this method, Kamvar et al. reported a reduction of 20% in the time needed to solve the PageRank algorithm. The adaptive power method required more iterations than the normal power method, but due to the decrease in average work per iteration, the adaptive power method still produced a significant reduction in calculation time. Kamvar et al. do not make it clear if this increase in iterations to convergence is to be expected in all instances of the adaptive power method or if they are related to the IA matrix structure.

The major disadvantage with this method, however, is that it is not guaranteed to converge. Often PageRank values level off for a time in the PageRank algorithm before changing and converging to a different number. If values that are momentarily levelled off are locked as being converged, the method will not get the correct PageRank vector and may not converge. Figure 4.1 shows how identifying a temporary levelling off in PageRank score as convergence can lead to an incorrect PageRank vector with adaptive power rank. A simple three node web/Internet, along with its PageRank vector during the first 10 iterations of the PageRank algorithm using both basic power method and adaptive power method are shown in Figure 4.1. When using the basic power method the PageRank vector is normalised throughout the first 10 iterations. The power method in this example will eventually converge to give $PR=[0.400,0.200,0.400]$.



	1	2	3	4	5	6	7	8	9	10
A	0.333	0.333	0.500	0.333	0.417	0.417	0.375	0.417	0.396	0.396
B	0.333	0.167	0.167	0.250	0.167	0.208	0.208	0.188	0.208	0.201
C	0.333	0.500	0.333	0.417	0.417	0.375	0.417	0.396	0.396	0.403

Power Method

	1	2	3	4	5	6	7	8	9	10		
A	0.333	0.333	0.500	0.333	0.417	0.333	0.375	0.333	0.354	0.333		
B	0.333	0.167	0.167	0.167 LOCKED								
C	0.333	0.500	0.333	0.417	0.333	0.375	0.333	0.354	0.333	0.340		

Adaptive Power Method

Figure 4.1 Adaptive power method mis-identified convergence

The adaptive power method however notices that the PageRank for node B does not change between iteration 2 and 3, thus falsely identifying node B as having converged. This PageRank value is then locked and remains at this value for the rest of the calculation. This mistake in the PageRank of node B stops the algorithm from calculating the correct value of PR. The PageRank vector becomes de-normalised. After 10 iterations, $PR=[0.333,0.167, 0.340]$. The algorithm will continue to iterate and finally converges to $PR=[0.333,0.167, 0.333]$. In some cases, the algorithm will not converge at all when this happens. The risk of this problem happening can be reduced by leaving multiple iterations of the power method before allowing a value to be locked. If, for example, a value has not changed over 10 iterations, it is much less likely to be a false positive. However, the risk cannot be totally eliminated since PageRank propagation relies heavily on the structure of the matrix.

The same lab in Stanford published another paper detailing another scheme for accelerating the PageRank algorithm called Quadratic Extrapolation [71]. The rate at which the PageRank algorithm converges is dictated by the second eigenvalue of the Google matrix. If the second eigenvalue is close to one, the PageRank algorithm

converges slowly. Kamvar et al., [71] proposed subtracting estimates of the sub-dominant eigenvectors at intervals during the algorithm. By using this method, they claim to speed up the computation of the PageRank algorithm by up to 300% on some of their data sets. This method reduces the number of iterations of the power method before convergence at the cost of calculating an estimate for the sub-dominant eigenvectors periodically throughout the calculation. The results show that the method works very well at α factors close to 1 for which the power method struggles to converge due to sub-dominant eigenvalues being close to the unit circle. However, at the level of α suggested by Brin and Page in their landmark paper, quadratic extrapolation shows little or no reduction in the number of iterations.

The third and final contribution to PageRank acceleration made by the team at Stanford is called BlockRank [72]. In this paper Kamvar et al. set out to increase locality of reference, reduce computational complexity and increase parallelization by taking advantage of the way that the Internet naturally orders itself, i.e. a web page's host path. They do this by looking at a top down approach to the web page's address. For example the URL for the Mechanical Engineering Department at Trinity College Dublin (TCD) is www.mme.tcd.ie/index.php this would be stored as follows ie.tcd.mme.www/index.php in Kamvar's approach. The top level domain "ie" comes first which states it is a host from Ireland, "tcd" comes next which tells us that the page is part of the TCD webspace. The "mme" shows that the page is on the mechanical engineering servers and the "www" shows that it is a public website on the mme server. The rest of the address is the location on the host "mme". This high level to low level organisation is inherent in Internet URL. Kamvar et al. sorted their crawl data according to this method. Kamvar then showed that above 79.1% of links on websites are intrahost links (links between pages on the same host) and over 83% are intra-domain links (links between pages with the same domain i.e. tcd.ie). This high level of intra-domain and host-linking is to be expected since most links on any given website are to other pages on that website. This statistic led Kamvar et al. to propose the BlockRank algorithm. The BlockRank algorithm is divided into two parts, the local PageRank calculation and the global PageRank calculation. The local PageRank calculation is computed using intra host links and the global PageRank vector is calculated using the inter host links. The algorithm begins by dividing the

web-graph according to hosts, see Figure 4.2. In this simple example, the Internet consists of nine web pages, which are from three different web sites or hosts.

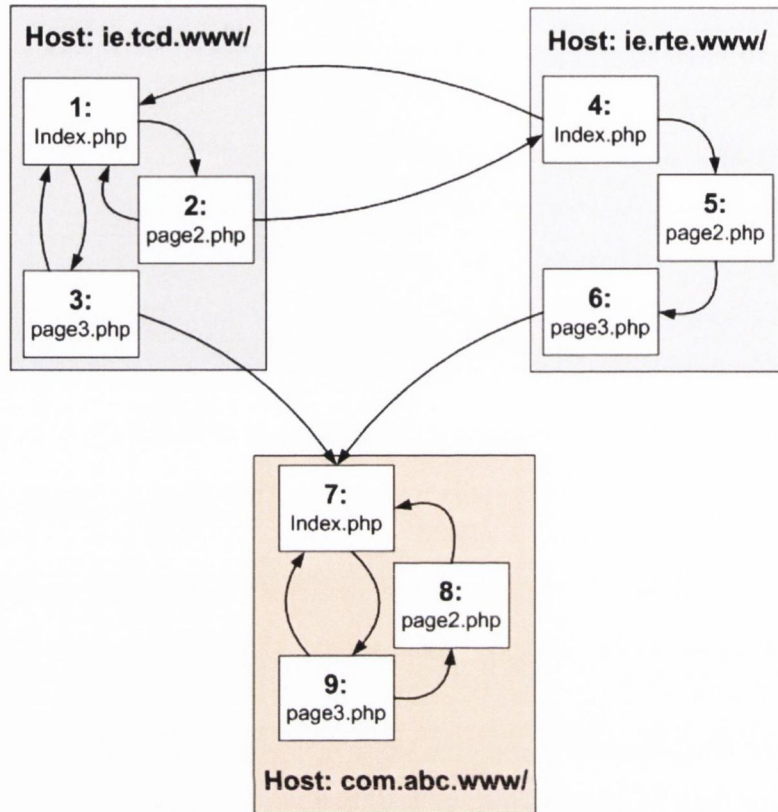


Figure 4.2 Example of Internet divided into hosts for Blockrank

Normally the adjacency matrix for Figure 4.2 would be given by Figure 4.3. The shaded areas of the matrix represent links that are intra-host links. There are many more intra-host links than there are inter-host links (i.e. a page on one host links to a page on another host).

0	1	1	1	0	0	0	0	0
1	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0
0	0	0	0	1	0	0	0	0
0	0	1	0	0	1	0	1	1
0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	1	0	0

Figure 4.3 Adjacency matrix for Internet shown in Figure 4.2

The BlockRank method extracts these local host adjacency matrices and partitions the calculation according to host. Any interhost links are placed in the newly formed global PageRank matrix. The matrix in Figure 4.3 becomes four matrices as shown in Figure 4.4.

$$\begin{array}{c}
 \begin{array}{c} P1 \ P2 \ P3 \\
 \begin{bmatrix} 0 & 1 & 1 \\
 1 & 0 & 0 \\
 1 & 0 & 0 \end{bmatrix} \\
 \text{Host 1}
 \end{array}
 \qquad
 \begin{array}{c}
 \begin{array}{c} P4 \ P5 \ P6 \\
 \begin{bmatrix} 0 & 0 & 0 \\
 1 & 0 & 0 \\
 0 & 1 & 0 \end{bmatrix} \\
 \text{Host 2}
 \end{array}
 \\
 \\
 \begin{array}{c}
 \begin{array}{c} P7 \ P8 \ P9 \\
 \begin{bmatrix} 0 & 1 & 1 \\
 0 & 0 & 1 \\
 1 & 0 & 0 \end{bmatrix} \\
 \text{Host 3}
 \end{array}
 \qquad
 \begin{array}{c}
 \begin{array}{c} H1 \ H2 \ H3 \\
 \begin{bmatrix} 0 & 1 & 0 \\
 1 & 0 & 0 \\
 1 & 1 & 0 \end{bmatrix} \\
 \text{Global}
 \end{array}
 \end{array}
 \end{array}$$

Figure 4.4 Local and Global adjacency matrices for Internet shown in Figure 4.2

The problem has now been split up into multiple matrices, one matrix for each host in the system and a global matrix that shows how the hosts in the system connect. In this example, it is coincidence that all the matrices are the same size. In reality, they would all be different sizes. The size of a local host matrix is dependent on the number of pages on the host and the size of a global matrix is dependent on the number of hosts in the system.

Once the web graph has been divided into local and global link matrices, the PageRank of each local host matrix is calculated in the normal way, see Section 3.5.5. Once the local host PageRank values have been calculated, the relative importance of each block or host must be calculated. The NZE in the global matrix are weighted by the results of the local PageRank calculations, to ensure that multiple links between hosts are weighted correctly. See [72] for details on how this weighting is achieved. The BlockRank is calculated using the modified global matrix. The BlockRank gives the relative importance of the blocks to each other. Once the BlockRank is calculated, each local PageRank vector is weighted by its corresponding BlockRank result. This weighting gives an approximate PageRank score for every page. The PageRank calculated by BlockRank is not the actual

PageRank since some links are ignored in both stages of the algorithm. Kamvar stated that these missing links in the local PageRank calculations cause the algorithm to give too little importance to the root node. This effect could be very problematic, as the root node (main page) should usually have the highest PageRank of a web site. Kamvar also asserted that using the BlockRank PageRank as the starting vector for the normal PageRank calculation would correct all these errors. Only a few iterations would be needed since the starting vector is a very good approximation.

As Kamvar et al pointed out, the BlockRank algorithm can speed up the PageRank calculation by a factor of two or more. It succeeds in improving calculation speed by breaking down the large problem into parallelisable blocks. These blocks can fit in cache and so increase the locality of reference. The smaller problems also converge much quicker than the large PageRank algorithm does. However, the BlockRank method's major disadvantage is that it only creates an approximation of the PageRank vector. Thus, the large PageRank vector must be run for a number of iterations at the end to correct the errors in the BlockRank PageRank vector causing additional time to be used in the calculation.

Yizhou Lu et al. published an algorithm similar to the BlockRank algorithm called PowerRank [73]. Lu sub-divided the Internet into three levels; domain, host and webpage. The PageRank algorithm is first run at a domain level. At this level, each domain is counted as a single node in the web-graph. After a number of iterations, the lowest scoring domains are removed from the calculation. The hosts from the remaining domains are then put forward for the second part of the algorithm. The PageRank algorithm is run another time on this new host graph. Again, after a number of iterations, the lowest scoring hosts are removed from the calculation. The remaining web-graph is expanded to include all remaining webpages and the PageRank algorithm runs for a third time on this reduced web-graph. The final step in the PowerRank algorithm is to assign a PageRank score to the pages that were excluded from the algorithm. This is done using the domain and host scores. Lu et al., claimed an increase in the performance of this method of 30% more than the basic PageRank algorithm. However, the PowerRank algorithm only estimates the PageRank algorithm. The PowerRank vector could be used as a starting point for the PageRank algorithm. Kamvar et al., did this with BlockRank to calculate the exact PageRank vector. Yet, this approach would reduce the performance increase

achieved by PowerRank. The PowerRank algorithm therefore isn't sufficient if an exact PageRank vector is needed.

A different approach is shown by Lee et al., who proved that, by using Markov chain lumping theory, the dangling nodes of the PageRank algorithm could be reduced to a single super node [74]. This change reduced the size of the matrix considerably without compromising the accuracy of the result as the PowerRank algorithm did.

Lee et al., used this theory by lumping all the dangling nodes together. This approach was viable because all the dangling nodes have the same probability distribution ($1/n$ in Brin and Page's original paper). There are many dangling nodes on the web. They are made up of PDFs, pictures, other non linking media and pages that have not been crawled yet. Lumping all these pages together significantly reduces the problem size. Lee then calculated the PageRank score for the reduced web-graph containing the dangling node supernode. He used aggregation to lump the non-dangling node pages together and computed the PageRank vector for the dangling nodes given the result obtained from the first step. These results are concatenated and produced a full PageRank vector which is identical to the PageRank vector produced by using the full matrix. Lee et al., achieved speed ups of up to five times better than the basic power method by using their Markov chain lumping method. Langville and Meyer described the same technique using linear algebra and proposed a reordering scheme that was suitable for it [75]. This reordering scheme does not increase the performance further as it is costly to implement, but they too achieved a factor of five speed-up over the basic power method. Implementing this change to the PageRank algorithm would not require any additional hardware units in an FPGA accelerator. All changes could be made on the software level at IA matrix creation time and runtime.

In this section, many adjustments to the PageRank algorithm were discussed. These adjustments can be used to increase the performance of the PageRank algorithm. However in order to successfully speed up an algorithm of this size on an FPGA, a large amount of parallelisation must be used. In the next section a number of parallel methods will be discussed. Many of these parallel methods can be used in conjunction with the algorithmic improvements discussed in this section.

4.2.3 Parallel Computation of PageRank

Much of the work described in the previous section opened the door to increased parallelisation in the PageRank calculation. BlockRank and PowerRank break the problem down into multiple local PageRank calculations allowing many parallel processors to be used when calculating the PageRank vector. Zhu et al. proposed another similar scheme to BlockRank [76]. However, the emphasis in Zhu's work was on distributed calculation of the PageRank algorithm. The method splits the matrix up into hosts much like the BlockRank algorithm.. However, in this case, each host must calculate its own local PageRank score. This local PageRank score is used as a starting estimate for the rest of the algorithm. Like BlockRank, a controller goes on to calculate the relative rank of all the hosts. Zhu's method differed in this regard from Kamvar's BlockRank. In Zhu's method, each host then had to construct an extended local transition matrix. It is not clear from [76] as to what exactly is included in this new extended matrix, but after PageRank calculations at a local level on this extended matrix, the vectors are normalised and sent back to the controller. This process is iterated until the convergence criteria are met. The majority of work in this algorithm is done by the host itself. The major shortcomings of this method are that it only calculates an approximate PageRank vector and all nodes have to wait for all other nodes to finish the current iteration before they can continue. Furthermore, allowing a host to calculate its own PageRank also leaves the scheme open to malicious attacks. A host could wilfully return the wrong local PageRank in an attempt to increase its own PageRank ranking.

Other work has been done by Bradley et al. on partitioning of the algorithm over parallel processors [77]. He found that using 2D graph partitioning increased the algorithm speed by almost twice that achievable with other techniques. However, there was a substantial overhead due to the initial partitioning. Cevahir et al., suggested using a site-to-page version of the pre-processing algorithm used by Bradley [78]. This method meant that instead of the $n \times n$ matrix used in Bradley's experiments, Cevahir used an $m \times n$ matrix where m is the number of sites being crawled and n is the number of pages. This reduced matrix size and increased the performance of the pre-processing step considerably. Yet, it remains ultimately unhelpful because of the necessary initial overhead.

4.2.4 Other FPGA Architectures for Internet Ranking

To the best of the author's knowledge, no other work has been released to date on FPGA implementations of the Google PageRank algorithm to the best of the author's knowledge. However, recently an FPGA based accelerator for RankBoost was unveiled by Xu et al. [79]. RankBoost [80] is a machine learning algorithm first published in 2003. Traditionally in search engines, many factors have to be taken into account when calculating overall relevancy. In Google, PageRank is only one of the scores used in the overall ranking. Other scores for page content, page freshness, URL, page title and many more must be factored into the final ranking score. This inclusion of other factors can be done by manual tuning the scores. The RankBoost algorithm is an alternate approach. It uses machine learning techniques to combine all the scores to give an overall ranking score. RankBoost must be trained with a large training set. This training is very computationally expensive and its operation is critical to the speed of RankBoost. Xu et al. developed a co-processor architecture to accelerate the training algorithm used by the RankBoost algorithm. The co-processor board connects to the host PC via a PCI link. The board contains a single bank of DRAM which gets filled by the host PC. The host PC then sets the Processing Elements (PE) on the co-processor running. The PE process data streams from memory. The algorithm must be run as many as 200,000 times and so any overheads caused by downloading data across the PCI are definitely merited. Xu found that the FPGA could process the training data almost 168 times quicker than a naïve implementation of the problem and 4 times quicker than an optimised system distributed across 4 threads on a 3GHz Pentium 4. Xu does use floating point arithmetic but it is unclear if double or single precision is used. The results are impressive and show that FPGAs can be used in Internet algorithms to speed up operations. The RankBoost accelerator has been successfully used by research engineers in Microsoft India [79].

4.3 SMVM

Sparse matrix by vector multiplication (SMVM) dominates the PageRank calculation as discussed in section 3.5.5. It is also the central calculation of many other scientific calculations and as such has been the subject of much research. Gropp et al. showed

that operations like SMVM, which are memory bound, are fated to achieve an ever decreasing percentage of the GPPs peak performance [81]. This consequence is due to the ever increasing gap between memory speed and processor performance. Much of the CPU's time is spent waiting for memory in the SMVM instructions. In this section, a number of attempts to accelerate the SMVM calculation will be discussed. These modifications can be grouped in three headings. The first of these three sections deals with ways to reduce the effect of this memory-processor gap on software. These software approaches to accelerating SMVM are presented in section 4.3.1. Next, specialised hardware for SMVM calculation are discussed in section 4.3.2. Finally, specialised FPGA accelerators for SMVM are examined in section 4.3.3. This last group of accelerators is of the most relevance to this project. Developing a powerful SMVM architecture on FPGA will be pivotal to the success of an FPGA based solver for the PageRank eigenvector problem.

4.3.1 Optimisation for SMVM in software

Software optimisation for SMVM calculations has been the subject of much research. Researchers are limited in what they can do by the fixed structure of the CPU architecture. Therefore, several methods exist that centre around the same general theme, namely to increase data reuse by increasing locality of reference. This result can be achieved in numerous ways, such as reordering the matrix, register blocking, cache blocking and multiplying multiple vectors at one time.

Sparse matrices often suffer from poor data locality when stored in the CPU cache. This problem can lead to poor performance as new values need to be read into the cache frequently when computing SMVM. Cuthill and McKee [82] introduced a reordering scheme for symmetric matrices. This scheme, known as Cuthill-McKee reordering, reduced the matrix bandwidth needed to represent the sparse matrix. The matrix bandwidth is a measure of how tightly the non-zero elements of the matrix are banded. George found that the reverse of the Cuthill-McKee reordering often achieved better results and never was inferior to the original Cuthill-McKee reordering [83]. This method is known simply as Reverse Cuthill-McKee (RCM). These reordering schemes were devised for symmetric matrices. They still often work well for non-symmetric cases. In the course of this work a number of experiments were carried out with RCM and are presented in section 6.5.3. Using

RCM reduces the number of the PageRank adjustments described in section 4.2 that can be used, since methods like BlockRank require the matrix to be stored according to its domain and host.

Register blocking attempts to increase data reuse in the highest level memory possible, the CPUs registers. Im and Yelick described a method for register blocking that divided the sparse matrix into dense blocks [63]. The optimal size of the blocks is determined by the structure of the matrix and the number of registers available in the GPP. A simple example of a matrix being partitioned into dense blocks for register blocking is shown in Figure 4.5. This diagram also highlights one of the drawbacks of register blocking. Zero entries have to be added to some of the blocks to make them dense and thus increasing the storage requirement for the matrix. Im and Yelick did find that this method increased performance in certain matrices though. Deciding on the optimal block size is not a trivial problem and requires a good deal of computation [84]. Im devised a method for deciding if register blocking would increase performance and a method to decide the optimal block size. It was made available in the SPARSITY tool box [84].

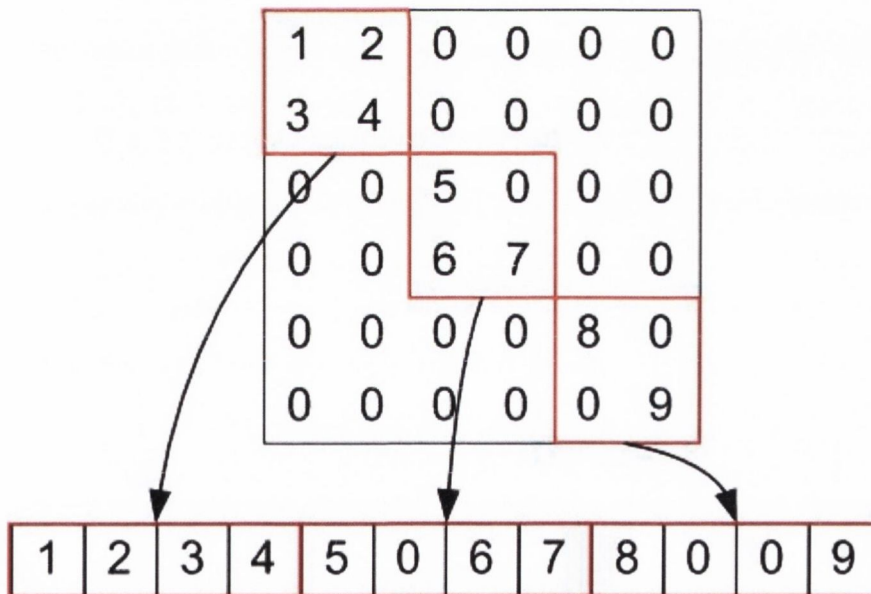


Figure 4.5 Register blocking dense block scheme to increase data reuse

Toledo also used register blocking as part of his optimisations for SMVM [64]. Unlike Im and Yelick, however, Toledo did not limit his search to square dense matrix blocks. He searched for dense rectangular blocks in addition to square blocks. Toledo also investigated RCM and found that with sufficient iterations, the

performance increase justified the cost of the reordering scheme. Furthermore, Toledo used pre-fetching in a novel way to increase the performance of his SMVM computations. Usually, pre-fetching is used to hide latency. This event occurs when a large amount of computations are completed between memory accesses. Pre-fetching removes the time spent waiting for memory to respond to a request. The large amount of data computations gives the pre-fetcher time to start the request to memory before it is needed. However, in sparse systems, a great deal of memory reads are needed and not enough computation time is available to use pre-fetching to hide memory latency. Toledo, however, used pre-fetching to avoid multiple load/store units stalling due to misses on the same cache line [64]. These changes increased the performance of his tests by more than twice the naïve implementation.

Another optimisation suggested by Im in her PhD thesis is the use of cache blocking. Cache blocking is very similar to the tiling technique used by the SCAR architecture in section 5.5. The matrix is divided into blocks. These blocks are processed one by one. The blocks, together with the corresponding segment of the X and Y vector, are stored in the GPP's cache. All values processed inside the block are in cache, thus reducing memory cache misses. This method, as implemented by Im, increased performance for a number of the matrices on the architectures tested [84]. Im also investigated the use of register and cache blocking in unison but found that no further performance increase was achieved by using both methods together. Im's results showed that if multiple vectors were multiplied by a single matrix the performance of the system could be greatly increased. Vuduc [85] and Gropp [81] both published results that suggest multiple vectors in SMVM can greatly increase performance. This process is suitable for some algorithms e.g. conjugate gradient with multiple right hand sides as is sometimes used in finite element problems, but not for the PageRank algorithm where only one vector for sparse matrix multiplication is available at any time. In the SPARSITY package, Im developed a method that chooses the best configuration for registry blocking and cache blocking for any given matrix. This calculation can be costly, but only needs to be done once per matrix.

Williams et al. examined sparse matrix by vector multiplications on multi-core machines [37, 86]. They ran a number of tests on Intel Xeon, AMD Opteron, Sun Niagara2 and IBM Cell processors. The Intel Xeon processor used in the experiments

is the Intel Xeon Clovertown, which is a very similar model to the Intel Xeon Woodcrest used in this project. The Clovertown can hold four core-2 processors sharing two FSBs to memory; see Figure 4.6. The Woodcrest on the other hand only contains two core 2 processors, each of which has its own FSB to memory. Since SMVM is memory bounded and both architectures share the same FSB architecture, the performance results should be similar.

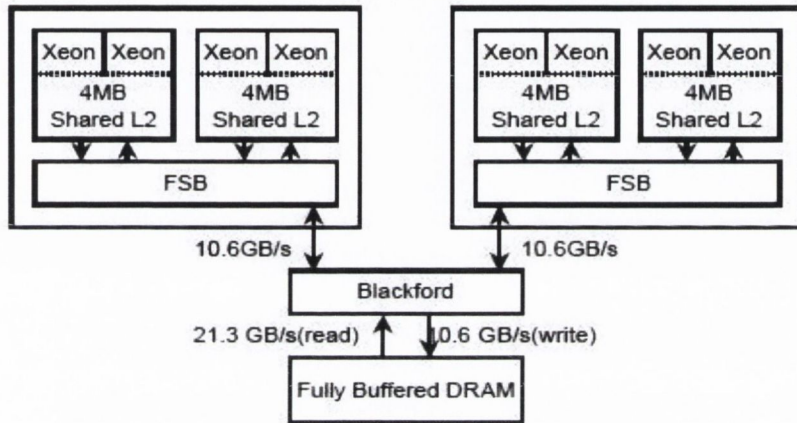


Figure 4.6 Clovertown memory hierarchy [37, 86]

The AMD Opteron machine consists of two sockets with two 2.2 GHz cores each with a 64 KB L1 cache, 1 MB victim cache (L2 cache); see Figure 4.7. Each socket in the Opteron has an independent FSB to DDR-2 667. A hyper-transport connection is used to transfer data between the two sockets [37].

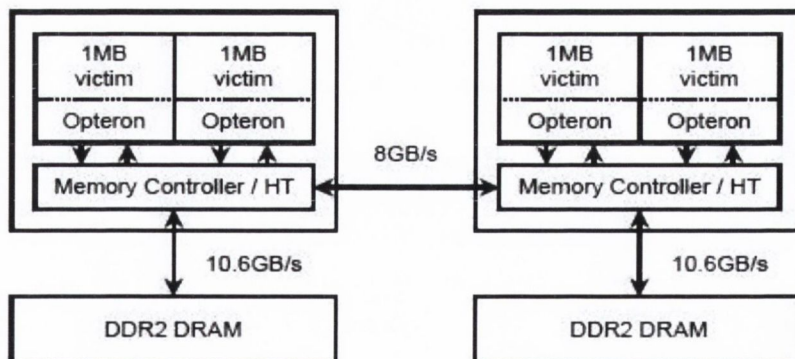


Figure 4.7 AMD Opteron memory hierarchy [86]

Probably the most interesting aspect of the results of this work is that one of the matrices used to test the processor performance is a matrix also used in the benchmarks of this thesis. This result will therefore provide some independent

validation of the benchmarks obtained in the course of this work. The Webbase matrix, used by Williams and in this work, is an Internet link matrix sourced from the Webbase archive (see Table 6.1). It is called webbase-2001 (matrix 13) in this thesis.

The first test run by Williams et al., was a naïve single thread implementation. The Intel Xeon achieved 0.3 GFLOPs performance for the Webbase matrix. It was one of the 3 worst performing matrices in the tests. The average performance of the 13 matrix test set was approximately 0.5 GFLOPs. The AMD Opteron 2214 performed even worse on the Webbase matrix with only 0.2 GFLOPs. Williams then went on to apply optimisations to the naïve SMVM code. The first of these optimisations was to parallelise the code to run on multiple threads. This allowed the computational load to be spread between all processor cores. All further optimisations were applied to the new threaded version of SMVM. Figure 4.8 shows how the performance of the Intel Xeon and AMD Opteron varied with these optimisations for the Webbase matrix. The median results for all the test matrices and all optimisations on both architectures are also included to show how poorly SMVM of Internet adjacency matrices perform, even in comparison to sparse matrices from other application domains.

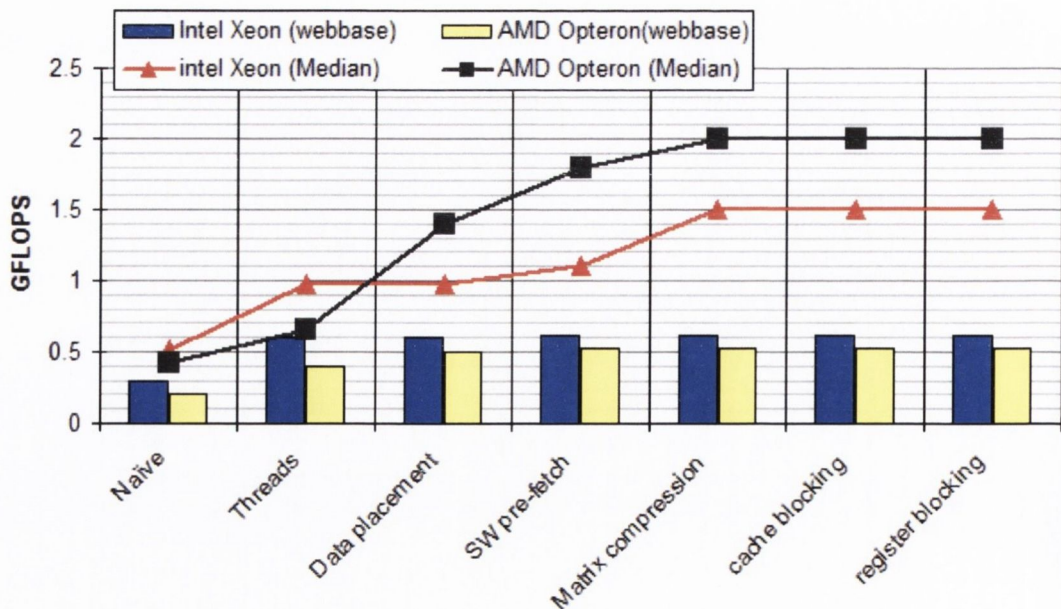


Figure 4.8 Webbase matrix performance on Intel Xeon and AMD Opteron [37] (Bars on the graph represent the performance of the IA matrix on the Xeon and AMD processor respectively. The line graphs represent the median performance of Williams test set, showing a large discrepancy between IA matrices and other Sparse matrices.)

It is clear from Figure 4.8 that rewriting the problem to run on multiple threads resulted in the biggest improvement to performance. Other techniques that appear to work well on other sparse matrices seem to do little to increase the webbase matrices performance further.

There are a limited number of optimisations that one can apply to the fixed architecture of the GPP. The GPP has been designed to perform well over a wide range of tasks and situations. However, some problems, like SMVM, have fallen behind with regard to performance due to the increasing gap between computational bandwidth and memory bandwidth. For this reason, Gropp et al., [81] have condemned the SMVM operator to an ever-decreasing percentage peak of performance. Software optimisations are limited in the increase in performance they can achieve since they are forced to use the underlying hardware. This problem appears to be especially true when looking at the Internet link matrix used in Williams' research. One way, however, to increase the performance greatly would be to consider custom-made hardware for SMVM.

4.3.2 Specialised hardware for SMVM

Reconfigurable hardware offers the ability to customise the hardware for any given task. With poor SMVM performance on GPPs, researchers naturally began to examine ways to remove the bottleneck in SMVM performance by using customised hardware. Back in 1983, Swanson, an employee at Ansys⁷, designed a co-processor system to accelerate SMVM on the Ansys platform on his VAX-11/780 system [87]. The idea was that the VAX would off-load certain calculations to use the dedicated co-processor in an attempt to accelerate solution time. The system reduced the computation time to as little as one-twelfth to one-sixteenth of the original computation time needed for the benchmark calculations. Swanson's scheme became obsolete quite quickly, however, as large strides forward were made in the design and performance of GPP. Despite Swanson's lack of success in creating a market for his accelerator, he did prove that his idea was a good one. Co-processors could be used to solve calculations that ran slowly on the GPP.

⁷ Computer Package for solving Finite Element Analysis problems

More recently Graphics Processor Units (GPUs) have been used in a similar fashion to accelerate graphics applications on PCs. In the past, GPUs were designed only for use as graphic pipelines. They were programmed with fixed algorithms for rendering graphics. Modern GPUs, however, offer a large amount of programmability for use in any application. For example, NVIDIA GPUs are based on the Tesla - Unified Graphics and Computing Architecture [88]. This architecture is a fully programmable parallel processor array. Some research has investigated the use of GPUs in scientific calculations. Garland presents a “how to” paper for mapping SMVM on to manycore GPUs [89] like a modern NVIDIA GPU. He does not implement his technique or test it. Bolz et al., do implement a conjugate gradient solver on their NVIDIA GPU [90]. They showed that the GPU could achieve 66% more SMVMs per second than their 3GHz Pentium 4. They also showed that both the GPU and the GPP were bandwidth limited in this calculation. The main problem with Bolz’s work is that GPUs natively use a version of single precision floating point. GPUs floating point arithmetic is not usually IEEE compliant, as most of them round de-normal numbers to zero and use slightly different rounding schemes than the IEEE standard. Therefore, the problems run on the GPU would get a different answer than problems run on the GPP. This can have serious consequences for linear algebra problems. For example, the Conjugate Gradient algorithm can fail to converge with even the smallest change in rounding. Goddeke noted that many real world scientific calculations need double precision floating point arithmetic [91]. Goddeke describes a mixed precision defect correction algorithm as a solution for the lack of double precision floating point on GPU. The single point GPU result is corrected with a few iterations of double precision correction arithmetic carried out on the GPP. Goddeke’s idea was to attain the full accuracy of the GPP, while keeping the speed of the GPU. The results show that by using this mixed precision error correction technique the GPU could out perform the GPP by 2.3 times for Goddeke test matrices. Goddeke also noted that for small matrices the GPP outperformed the GPU, but as the problem out-grew the GPP’s cache, the GPU actually maintained a much higher performance. The residuals of the GPU method were almost identical to that of the GPP method. This GPU method had to perform many more iterations than the GPP, but due to its efficient operation, the GPU still outperforms the GPP. This correction method is an interesting way to work around the lack of double precision arithmetic available in the GPU. However, results using

this method may still be different from the reference results obtained from the GPP and as such may reduce its usefulness in scientific computing. Both answers would be correct but the residuals in Goddeke’s results sometimes differed. This could cause slightly different numbers being returned for both methods.

Another approach to floating point acceleration was proposed by Wolfe et al. in 1988 when they described their White Dwarf architecture [92]. The white dwarf was a co-processor unit which could be attached to a SUN 3/160 machine. It consisted of a CPU unit which was based on AMD chips, memory and a high speed communication bus. Internally, the data path consisted of separate integer and floating point units. The floating point units were used for computing the single precision numbers needed in the matrix and vector operations. The integer unit was used for all other calculations (the White Dwarf was designed before double precision arithmetic became the norm for scientific calculations). The integer data path also retrieved address pointers from memory for use in the floating point units. The architecture achieved a sustained 80% floating point unit utilisation. It could compute calculations twice as quickly as the stand alone Sun workstation. The result of 17.8MFLOPs may look small in relation to current standards but, for its time the White Dwarf was an impressive machine. It once again showed that specialised hardware can be used to increase performance in floating point calculations.

In this section, a number of historic floating point accelerators were presented. The White Dwarf and the Ansys solver show that specialised hardware has been used to accelerate poor GPP performance for many years. A modern approach to floating point acceleration was also described in the form of GPU accelerators. Historically, GPUs were very difficult to program for non-graphic applications. However, in recent years the GPU API has become much easier to use for all sorts of applications. The major drawback with these systems is their lack of double precision arithmetic. Goddeke shows a way to return a double precision answers using single precision arithmetic. The result, however, would not always be identical to that achieved by the GPP. In search engines, a small change in ranking score could be pivotal for a business, as it could cause them not to show up on the first page of results in a search. In the next section SMVM on other specialised FPGA based hardware platforms will be discussed.

4.3.3 SMVM on FPGAs

In the research presented by Gropp et al. [81] GPPs were shown to have ever-decreasing percentage peak performance for memory bounded calculations like that of SMVM. This is due to the widening gap between computation bandwidth and memory bandwidth. Underwood investigated GPP and FPGA floating point performance in 2004 [12]. He found that the rate of peak achievable floating point performance was growing at a significantly faster rate on FPGAs than it was in GPPs. He compared the most popular FPGA and GPPs from a number of years between 1997 and 2004. 1997 was the first year that FPGAs capable of running floating point cores became readily available. Underwood then extrapolated the floating point performance of these test systems to future years and found that the FPGA would outperform the GPP for all the floating point operators he tested by 2009. The projected results showed FPGA floating point performance would outperform GPP floating point performance.

Craven et al., looked at Underwood's research and using his data analysed the FPGA in terms of cost-performance [93]. His data pointed to FPGA solutions becoming cheaper per GFLOP of performance sometime between 2009 and 2012. This result came with the stipulation that the Virtex 4 results were removed from the test data. Craven claimed that less than expected floating point performance in Virtex 4 was caused by a reduced number of multiplier units. He goes on to state that the introduction of 25x18 bit multipliers in the Virtex-5 family would appear to put the FPGA back on track to reach the processors cost-performance ratio by 2009-2012. For this reason, a good deal of research has been carried out on floating point algorithms on FPGAs in recent years. A great deal of this research has been focused on the SMVM operator, due to its poor performance on the GPP.

Unlike the GPP's single memory configuration which due to a mismatch in memory and computational bandwidth cannot fully exploit the available floating point arithmetic units available in the GPP, the FPGA is flexible and can be programmed to use the very best memory architecture to ensure higher utilisation of the FPGA computational bandwidth. Modern large FPGAs have enough I/O pins and logic to create highly parallelized hardware platforms which can lead to better performance. In this section some of the most noteworthy publications in this field will be reviewed.

DeLorimier described an architecture specially designed for SMVM on a Virtex-2 FPGA [14, 94]. In these papers, Delorimier attempts to take advantage of the large memory bandwidth available from FPGA BRAM to increase the performance of SMVM calculations. Since memory bandwidth limits SMVM calculation speed on GPPs, Delorimier hoped that by using BRAM significant improvements could be made. BRAM's relatively large memory bandwidth when compared with the bandwidth of commodity memory would be responsible for this performance improvement. A large amount of the logic available in the Virtex-2 device used in these experiments is utilised as BRAM. The BRAM is used to store the matrix and vectors needed for SMVM. The matrix is stored in CRS and the SMVM is computed as a series of dot products which are spread across multiple processing elements (PE). DeLorimier experimented with single FPGA solutions. In order to scale the system for larger matrices, he also ran tests on a multiple FPGA platform. In these experiments he found that he could scale up to 16 FPGAs. However, the communication overhead for larger systems became prohibitive if any more FPGAs was added to the system. DeLorimier suggested a special type of FPGA with larger memory units and less logic would allow the system to accommodate larger problems.

The results achieved by DeLorimier's architecture are impressive, with a sustainable 1.2 GFLOPS for a single FPGA system and 12GFLOPS for the 16 FPGA system. The non-linear scaling is caused by the communication overhead. All results from a given iteration of the SMVM need to be broadcast to the other FPGAs before a new iteration can be started. A number of very serious drawbacks do exist with DeLorimier's architecture. The most obvious shortcoming is cost as using FPGA BRAM as the primary source of memory is expensive. In 2005, when he wrote his paper, DeLorimier would have had to pay \$96,000 for 16 Virtex-2 FPGAs [65] which shows that scaling by the addition of extra FPGAs is not a good idea. Another disadvantage of DeLorimier's architecture is that it does not scale past 16 FPGAs. This scaling issue together with the prohibitive cost would be an insuperable problem in dealing with large IA matrices.

Zhou and Prasanna also developed an FPGA architecture based on repeated dot products [13]. Their system is implemented on a Cray XD1 [95]. A block diagram of the Cray XD1 is in Figure 4.9. The FPGAs in the system are programmed with the

architecture described by Zhuo and Prasanna. The FPGA architecture utilises three levels of memory. These are the FPGA's onboard BRAM, SRAM and SDRAM.

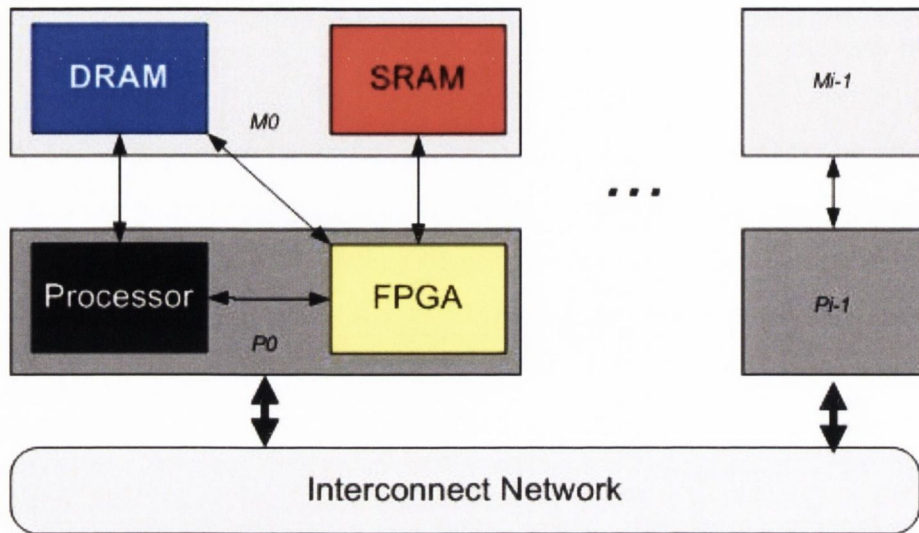


Figure 4.9 Zhuo and Prasanna's system model [13]

The onboard BRAM is used as a cache. In the GPP cache all values read in from memory are stored in the L1 cache. Data that is only going to be used once can replace data that is going to be used multiple times causing multiple costly cache misses. The FPGA has a major advantage over the GPP as only values explicitly written into the BRAM are cached. This set-up allows the designer full control of cache contents. This level of control is impossible on a GPP where the operating system controls the cache contents. In SMVM the matrix entries are not reused and so there is no need to cache them. Instead Zhuo and Prasanna use the BRAM to store values from the X and Y vectors. The second level of memory is the SRAM. This memory technology is slower than BRAM, but has a much higher bandwidth and lower latency than the third type of memory which is SDRAM. Small matrices can be stored in SRAM and their X and Y values can be stored in BRAM. This architecture can scale to any size as the DRAM can be used if the SRAM becomes too small for the problem. For larger problems, the X and Y vector need to be split up and partial dot products are calculated.

Zhuo and Prasanna's architecture used dot products to calculate the SMVM. Four NZE in CRS format were read in on every clock cycle. All four NZE products must be added to the same Y value and so a tree structure is used to accumulate the results.

This gives a partial result. A reduction circuit is used to add the partial results together; see Figure 4.10.

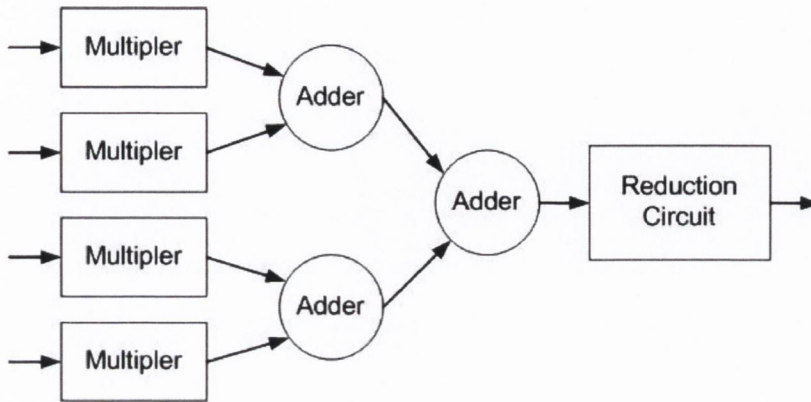


Figure 4.10 SMVM architecture used by Zhuo and Prasanna

Zhuo et al designed a special reduction circuit that uses only one adder for use in this architecture, see Figure 4.11. The reduction circuit contains two a^2 buffers, where a is the adder pipeline depth. The first a partial results from each row are placed in one of the buffers. Any subsequent partial results from that matrix row are fed directly to the adder where they are added to one of the results in the buffer before the new partial result is written back to the buffer. Thus, the number of partial results for each row is reduced to a . This is done for a rows of the matrix to fill one of the buffers.

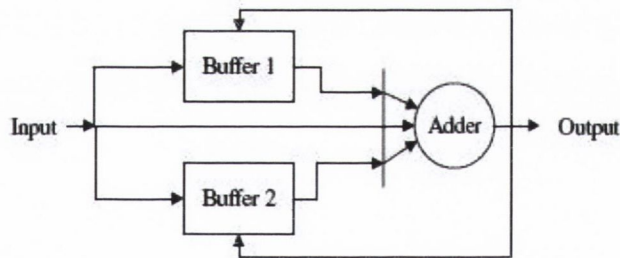


Figure 4.11 Reduction circuit used by Zhuo and Prasanna [96]

The reduction circuitry then begins to fill the other buffer in the same manner, while the other buffer is filling, the first buffer reduces the remaining a partial results of each row to the final result. Since each buffer contains a rows of the matrix, Read After Write hazards can be avoided by servicing these rows in a round robin fashion. This circuitry works because it takes the same time to fill the second buffer as it does to reduce the first buffer to the final answer. Full details of this reduction circuit are available in [96].

The Zhuo and Prasanna architecture could achieve a sustained 262 MFLOPS at a clock rate of 164MHz for matrices that were stored in DRAM. Speeds of up to 1.05 GFLOPS were achieved on matrices that could fit in SRAM. Internet Adjacency matrices, however, would have to be stored in DRAM and so would only achieve a maximum of 164 MFLOPS. Prasanna and Morris ran tests on a SRC06 MAPStation [97]. They presented the same architecture as described above. However, they estimated with post PAR figures that the SRC-06 would perform better than the Cray XD1. They estimated a sustained performance in the region of 167-670 MFLOPs for their test sets. These results have not been verified in hardware.

El-Kurdi et al. [98] described an accelerator for SMVM calculations on FPGA. Their system used the single precision floating point format. El-Kurdi introduced a novel striping approach [99]. This striping method was specially designed for his architecture. The architecture itself consists of eight processing elements which are pipelined together. The matrix stripes are streamed from memory into one of the eight processing elements. Meanwhile the X and Y values are passed along the PE pipeline until the end of the stripe is reached. The eight processing units, X queue, Y queue and resultant Y FIFO all share a single link to memory. A MUX ensures that all units receive their relevant data and that new Y values are written back to memory. The author claimed that sustained performance of 1.5 GFLOPS would be achieved if the architecture had an 8Gb/s link to memory. This architecture would contain multiple FPGA and would utilise the rest of the TM4 development board [100] that the project uses.

The FIAMMA group, the pre-cursor of the current project, also investigated SMVM on the Virtex-II platform [19]. FIAMMA investigated the SPAR architecture, first proposed by Taylor [17] in 1995. They implemented the SPAR on a Virtex-II V6000 FPGA with four independent banks of DDR-266 SDRAM. The SPAR consisted of two vector fetch units, a MAC unit, X cache, Y cache and control logic (see section 5.4). They found a number of the assumptions made by Taylor in her initial paper impossible to achieve on FPGA. Taylor had described a three cycle adder and a memory latency of 10 cycles. The FIAMMA group found they needed to increase the adder pipeline to maintain a high clock rate. They also found 10 cycles to be an unrealistic memory latency. The FPGA implementation of the SPAR showed that Y-cache misses were a major bottleneck in the system. A matrix banding scheme is

proposed to reduce the Y-cache misses at the expense of increasing RAW hazard penalties. The system achieved a maximum performance of 422 MFLOPs which was 74% of the theoretical peak. The SPAR architecture will be discussed again in section 5.4 and its performance with IA matrices is in section 6.4.1.

In this section a number of architectures specially designed for SMVM algorithms have been presented. DeLorimier designed a system with impressive performance, but which was limited in its scalability. El-Kurdi presented a novel striping architecture, but published no real performance data for his architecture. Zhuo and Prasanna's architecture is scalable and performs well, but does not contain the elements needed to do the rest of the PageRank algorithm. In order to be useful in specialised hardware, the system needs to be able to compute SMVM, dot product as well as other vector operations as described in the section on PageRank (§3.5.5).

4.4 Arithmetic units on FPGA

In the previous section, FPGA architectures for SMVM were discussed. A number of multipliers and adders are found at the heart of these units. The precision and speed of these arithmetic units dictate the performance and usefulness of the overall system. In this section the capabilities of fixed point/integer arithmetic and floating point arithmetic on FPGA will be discussed.

4.4.1 Fixed point/Integer arithmetic on FPGA

Traditionally, FPGAs have been known for their good performance on integer arithmetic. The FPGA is well-suited to perform large quantities of parallel integer arithmetic. In this section some of the research carried out on a number of fixed point/integer arithmetic units will be discussed.

In 2005, Becvar et al., published a paper looking at various precision fixed point arithmetic on the Virtex-II FPGA [101]. Becvar et al. implemented a Carry-Ripple adder (CRA), Carry-lookahead adder (CLA), Carry-Skip adder (C-SKIP) and a Carry Select Adder (C-Select). These adders were compared with an adder generated by the Xilinx ISE 5.2 (Generated). The results obtained from this study are summarised in Table 4.1.

Table 4.1 Summary of Fixed point adders [101]

Name	16 bit		32 bit		64 bit	
	Area (Slices)	Delay (ns)	Area (Slices)	Delay (ns)	Area (Slices)	Delay (ns)
CRA	23	13.52	47	23.99	95	40.27
CLA	29	7.97	61	19.06	127	24.84
C-SKIP	23	11.54	47	15.67	95	18.02
C-SELECT	27	13.27	67	16.47	135	23.88
Generated	8	4.05	16	5.09	32	8.38

The results for fixed point addition in Table 4.1 show that the synthesis generated fixed-point adder is both faster and smaller than all the other adders tested. Becvar et al., conclude that this is due to the fact that the generated fixed-point adder is the only adder to use the dedicated carry chain hardware available on the Virtex-II FPGA. The Generated adder has a delay of 8.38ns for the 64 bit operand which means that the adder can be clocked at approximately 120 MHz on the Vitex-II FPGA.

Becvar et al. also ran a number of tests for a number of different fixed-point multipliers and the results of these tests are summarised in Table 4.2.

Table 4.2 Summary of Fixed point Multipliers [101]

Name	16 bit		32 bit		64 bit	
	Area (Slices)	Delay (ns)	Area (Slices)	Delay (ns)	Area (Slices)	Delay (ns)
Array	364	28.96	1536	53.45	4187	99.97
Wallace	390	19.18	1542	27.18	6164	37.00
Generate-C	134	20.32	49	26.27	2109	29.97
Generated	0	14.87	533	18.54	295	28.94

Once again Becvar et al. found the core generated by the synthesis tool to be the most efficient single cycle multiplier. The generated multiplier uses the dedicated 18x18 multipliers available on the FPGA. The 64 bit multiplier has a much slower clock speed than the single cycle 64 bit adder and so would probably need to be pipelined so that they could be used in the same circuit.

Xilinx Core Gen [102] has the ability to create fixed point adders and multipliers. The user sets parameters and the core generator will implement the desired unit. The newest version of the Xilinx fixed point multiplier core shows that a 53x53 multiplier

can operate at a maximum frequency of 450MHz on the Virtex-5 FPGA [103]. This multiplier has a 12 cycle latency which is quite a large latency for a fixed point multiplier. The latency of the multiplier could be reduced if a high clock rate is not needed. The Xilinx LogiCore add/subtract unit [104] is capable of running at a maximum clock frequency of 340 MHz on a Virtex-5 FPGA. This system adds two operands of 100 bits and has a latency of 9 clock cycles. The Xilinx fixed point cores have high clock rates but these are achieved by increasing the latency.

4.4.2 Floating point arithmetic on FPGA

The IEEE standard 754 for floating point numbers was published in 1985 [40]. The IEEE 754 describes a standard approach to floating point arithmetic, including precision, range, rounding modes, exception handling, overflow, underflow and handling of denormalised numbers. Before the IEEE standard was introduced, every manufacturer had their own approach to floating point numbers. Software designers found it increasingly difficult to ensure that their systems worked properly on all platforms. Since then, the IEEE 754 compliant double precision arithmetic has become the standard for scientific applications all over the world. FPGAs traditionally had poor performance with floating point arithmetic before the introduction of the Virtex series FPGA. The advent of large FPGAs, however, has allowed floating point performance to steadily improve. As discussed earlier, Gropp has shown that floating point performance on FPGAs is increasing at a rate quicker than that of floating point on the GPP.

Some of the earliest attempts at implementing floating performance on an FPGA were carried out by Fagin et al. [105], Shirazi et al. [106] and Louca et al. [107]. Fagin et al., implemented a single precision floating point adder and multiplier over 4 FPGAs. Their arithmetic units could run at 4MHz and included logic for rounding and denormal numbers. However their multiplier was not pipelined and thus could only produce a result every 6 clock cycles. A year later, in 1995 Shirazi et al., implemented a half single precision adder and multiplier. The operands were 18 bits wide, but Shirazi managed to fit the design on to a single FPGA. In 1996, Louca et al., implemented a single precision adder and multiplier. Louca's adder was fully pipelined but his multiplier, like that of Fagin et al, could only handle an input every 12 cycles. These attempts at implementing floating point arithmetic clearly show that

FPGA based floating point units were still not viable. In 1998 the first fully pipelined multiplier was implemented on FPGA by Ligon et al. [108]. This single precision floating point pipelined multiplier was made possible by the larger FPGAs available in 1998.

The Virtex-II FPGA first went on the market in 2000. This model was a significant step forward in FPGA technology [15]. The Virtex-II was the first FPGA to contain integrated 18x18 integer multipliers. Researchers quickly noticed that these multipliers could be used to speed up floating point arithmetic units. The hardware multipliers also greatly reduced the amount of logic needed for floating point units. Roesler et al., showed a 77% reduction in resources needed to implement a single precision floating point multiplier when using the inbuilt 18x18 multipliers instead of conventional FPGA logic [109]. Furthermore, the Virtex-II FPGA was also the first FPGA that made FPGA based double precision floating point arithmetic viable. This new capability spurred a great deal of research into the area of double precision arithmetic on FPGAs. This research led to the investigations carried out on SMVM and other scientific calculations that have been mentioned in the previous sections of this chapter. Table 4.3 shows a summary of some of the double precision adders available on Virtex-II.

Table 4.3 Summary of double precision floating point Adders on Virtex-II and Virtex-II pro
(N/A = not available)

Name	No of Slices	no of 18x18 Multipliers	Latency	IEEE STD Compliant	Clk MHz (Speed)	REF
FIAMMA	937	0	6	Yes	110 (-6)	[11]
Underwood	1090	0	14	Yes	125 (-5)	[12]
Zhuo	892	N/A	14	Yes	170 (Pro)	[13]
DeLorimier	790	N/A	13	N/A	140 (-4)	[94]
Xilinx	861	0	12	Yes	134 (-6)	[110]
Dou	738	0	8	No denormals	177(Pro)	[111]
Govindu 1	693	N/A	8	Yes	130 (Pro)	[112]
Govindu 2	1383	N/A	23	Yes	200(Pro)	[112]

Table 4.3 shows the different trade-offs that exist between slice count, latency, IEEE compliance and clock speed. Reducing slice count often comes at the cost of reducing clock speed or losing IEEE compliance. Equally so, increasing clock speed often can only be achieved by lengthening latency and increasing the size of the

circuit. The many different flavours of FP adder, shown in Table 4.3, reflect the many different requirements of the architectures of the designers. The clock speeds of the Virtex-II Pro implementations tend to be higher than those of the Virtex-II. This increased clock speed is to be expected as the Virtex-II pro was designed as the next step up from the Virtex-II. Table 4.4 shows a similar table for double precision multipliers.

Table 4.4 Summary of double precision floating point Multipliers on Virtex-II and Virtex-II pro (N/A = not available)

Name	No of Slices	no of 18x18 Multipliers	Latency	IEEE STD Compliant	Clk MHz (Speed)	REF
FIAMMA	825	9	7	Yes	114 (-6)	[11]
Underwood	1607	9	20	Yes	105 (-5)	[12]
Zhuo	835	N/A	11	Yes	170 (Pro)	[13]
DeLorimier	3276	N/A	26	N/A	140 (-4)	[94]
Xilinx	703	16	8	Yes	105 (-6)	[110]
Dou	585	9	5	No denormals	178(pro)	[111]
Govindu 1	775	10	11	Yes	130(pro)	[112]
Govindu 2	1558	10	23	Yes	200(pro)	[112]

The trade-off between size, latency, IEEE compliancy and clock speed can be seen in Table 4.4 for the multipliers in the same way as occurred with the adders. The sizes of the multiplier and adder cores are much smaller than the devices on which they are implemented. Therefore, for the first time, it was possible to implement multiple cores in parallel. This parallelisation of floating point cores allows FPGA architectures to compete with the GPP despite having a much slower clock rate.

The generations of FPGAs that followed the Virtex-II continued to progress towards faster floating point arithmetic. Ehliar et al. [113] developed non-IEEE compliant single precision floating point cores on Virtex-4. Ehliar's units did not support denormal numbers, Inf code words or NaN code words. His design achieved 250 MHz in a complete system. Xilinx Logicore double precision fully IEEE compliant floating point adder and multiplier achieved a maximum clock rate of 327MHz and 380MHz respectively when implemented on Virtex-4 [110]. Xilinx's high clock rate is achieved by liberally using the extreme DSP blocks. If none of the Virtex-4 extreme DSP blocks are used, the clock rate falls to 181MHz and 301 MHz for the multiplier and the adder respectively. When Gropp et al., [81] was extrapolating

Underwood's results, he found that he had to remove Virtex-4 floating point performance. He claimed that FP arithmetic on Virtex-4 did not follow the trends set by other generations of FPGA. This claim was not made because of the clock rate that Floating point operators on Virtex-4 FPGAs could achieve, which can be seen here to be well above that of the Virtex-II. It was due to the fact that Virtex-4 FPGAs do not have the same proportion of built-in multipliers per slice as previous generations. This was probably caused by Xilinx moving from simple multipliers to extreme DSP slices. Gropp et al., also claimed that this problem was fixed with the release of Virtex-5.

Virtex-5 is the newest generation of Xilinx FPGA [33]. The changes made to the fabric of the device seem to have increased floating point performance a great deal. Wider and more numerous built in extreme DSP units increase multiplier performance. The same LogiCore [110] floating point multiplier and adder described above implemented on the Virtex-4 FPGA can run at a clock rate of 410 MHz and 397 MHz respectively on the Virtex-5 FPGA with liberal use of DSP units. The clock rate drops to 237MHz and 338MHz for the multiplier and adder respectively if no DSP units are used. A number of other multiplier combinations use less DSP units and can be clocked at 386MHz and 368MHz respectively.

A great deal of research has been carried out on FPGA with regards floating and fixed point arithmetic. Floating point is more complex to implement and takes more FPGA logic, but it offers a greater dynamic range. It is for this reason that FPGA floating point has become a large area of research in scientific computing.

4.5 Summary

In this chapter a number of works related to this thesis have been reviewed. It is hoped that this will help put the results of this work in context. There are no results that can be directly compared to the results obtained by this thesis but a good deal of work has been carried out in other related scientific fields like FEA. The chapter started with a look at research into alternative PageRank algorithms. Some of these research ideas did show performance improvements over the basic power method but this increased performance came at the cost of higher memory requirements or a lengthy reordering step. The performance of some of the alternative algorithms

proposed is dependent on matrix structure. However, the power method offers a trade off between convergence speed, memory requirements, stability and scalability that is difficult to beat.

Next, a number of modifications to the power method for PageRank were discussed. Some of these modifications affect the stability of the PageRank algorithm like the first of the three methods proposed by Kamvar et al. [70] where the values of PageRank were locked as they individually converged. Most of these modifications did not require extra hardware and so could be implemented along with any hardware accelerator. This was the case with BlockRank [72], and Markov chain lumping theory [74]. BlockRank creates an estimation of the overall PageRank vector which may limit its use but the Markov chain lumping theory idea proposed by Lee et al. gives the exact PageRank vector.

A great deal of research has been carried out on software optimisations for SMVM on the GPP. Most of these modifications to increase performance revolve around increasing data reuse and smarter use of the cache. The results obtained by William's on the single IA matrix in his test set shows that these optimisations give little or no performance increase for the very sparse IA matrix.

A good deal of research has been carried out on hardware for SMVM. GPUs are examples of stream architectures that specialise in high performance computing. Traditionally it was very difficult to target these devices for non-graphic applications but in newer GPUs advanced APIs have been made available making it much easier to run applications on the GPU. However, the GPUs lack of double precision floating point arithmetic has limited its usefulness in scientific computing.

The final area examined in this chapter was FPGA arithmetic. Modern FPGAs have made it possible to implement complex accelerator architectures. These can include highly parallelised architectures that are more suited to SMVM than the GPP processor. The major bottleneck in FPGA design is the clock rate. Despite this fact the FPGA floating point performance is set to pass the GPP in 2009/2010 [12].

In this section the current state of the art in a number of key technologies was discussed. Although no direct comparison for this project was found there are aspects used in this project that have been used in other areas of scientific computing. This

knowledge will aid with the understanding of the implementation which is presented in the next chapter.

Chapter Five

5 Design and Implementation

5.1 Introduction

In the preceding chapters a background to the PageRank problem has been presented. In chapter 2 an overview of the technologies being used was presented. This was followed in chapter 3 with an investigation into Internet search algorithms including the PageRank algorithm. In chapter 4 a review of related work was presented. In this section the details of the hardware used in this implementation of the PageRank architecture are discussed. Details of the hardware architecture are presented in section 5.2 to section 5.2.1. A discussion on the precision needed to accurately represent the PageRank vector is given in section 6.5 and details of the software architecture are presented in section 5.7. Programming details are also included.

5.2 Architecture

The architecture is designed as a System on a Chip (SoC) for solving problems in linear algebra such as calculation of the PageRank vector. A high level depiction of the system is presented in Figure 5.1. A soft RISC processor, the Microblaze, is used as the system controller and it is interfaced to all peripherals, arithmetic units such as the SMVM, PCI interface and memory controllers via four Wishbone buses.

The arithmetic units consist of three fully fledged SMVM units and a programmable Vector Unit which can compute any vector-vector or vector-scalar operation. Matrix by Vector operations can be computed using one, two or three SMVM units if the operation has been partitioned in a pre-processing stage. A fully populated system comprises three SMVM units and four independent memory spaces. Commodity

DDR/DDR2/DDR3 memory is used. Thus, four memory controllers are provided for the large capacity memory. In most cases, three memory spaces are used for matrix storage with the fourth needed for vector storage. High speed caches/buffers are provided locally in the SMVM units to reduce the number of accesses to the slow DDR memory. The host computer communicates with the FPGA board via a PCI bus. Thus, the system includes a PCI interface. The PCI interface allows the host PC to populate the memory with matrix and vector data. The arithmetic units can be controlled directly by the host PC or via the MicroBlaze processor. There are 3 separate clock domains in the system, the PCI clock, the memory clock and the system clock. The PCI clock domain is used in the PCI interface on the PC side and the memory clock domain is used by the memory controllers on the memory side. The Wishbone buses, arithmetic units, MicroBlaze and Wishbone interfaces use the system clock.

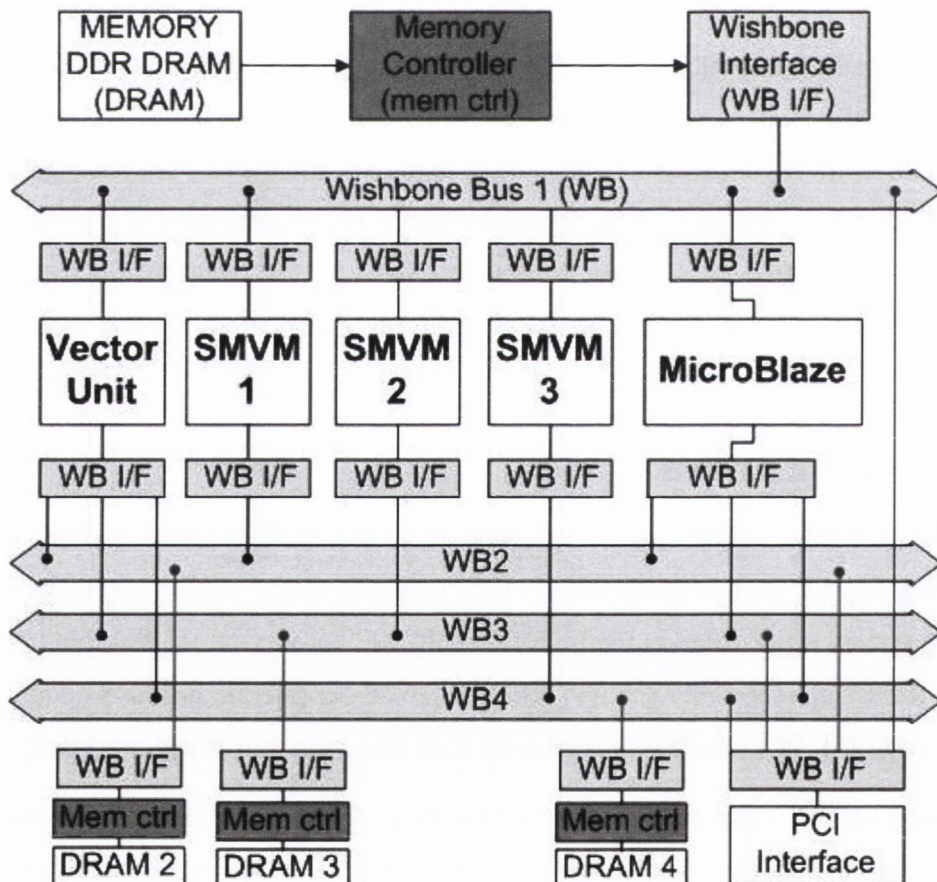


Figure 5.1 High level view of PageRank hardware architecture

5.2.1 MicroBlaze

It was decided that a processor would be used as the system controller. This would allow the hardware units on the FPGA to be controlled by a C program running on the processor. The Virtex-II FPGA can implement a MicroBlaze (MB) soft RISC processor. A soft processor is one that is implemented using the FPGA logic. Thus including a processor uses FPGA resources. Xilinx offer a hard PowerPC processor on a number of their other FPGAs like the Virtex-II pro [114]. The PowerPC is embedded in the FPGA silicon. Thus, it does not utilise any FPGA resources. The PowerPC is not available on the Virtex-II FPGA so the MB was used.

The MB was implemented using the Xilinx Embedded Development Kit (EDK). The MB is a 32 bit processor and is designed to connect to an On-chip Perennial Bus (OPB). It is through the OPB that the MB communicates with all other hardware units. It also is connected to a small RAM module where program code and data are stored. The control program for the system is stored in this memory. The MB can be easily programmed using C. This is a major advantage as it makes emulating operations and reprogramming the system for other algorithms very straightforward. Writing programs for the MB is discussed later in section 5.8.3. Figure 5.2 shows a block diagram of the MB and associated hardware as it was implemented in this architecture.

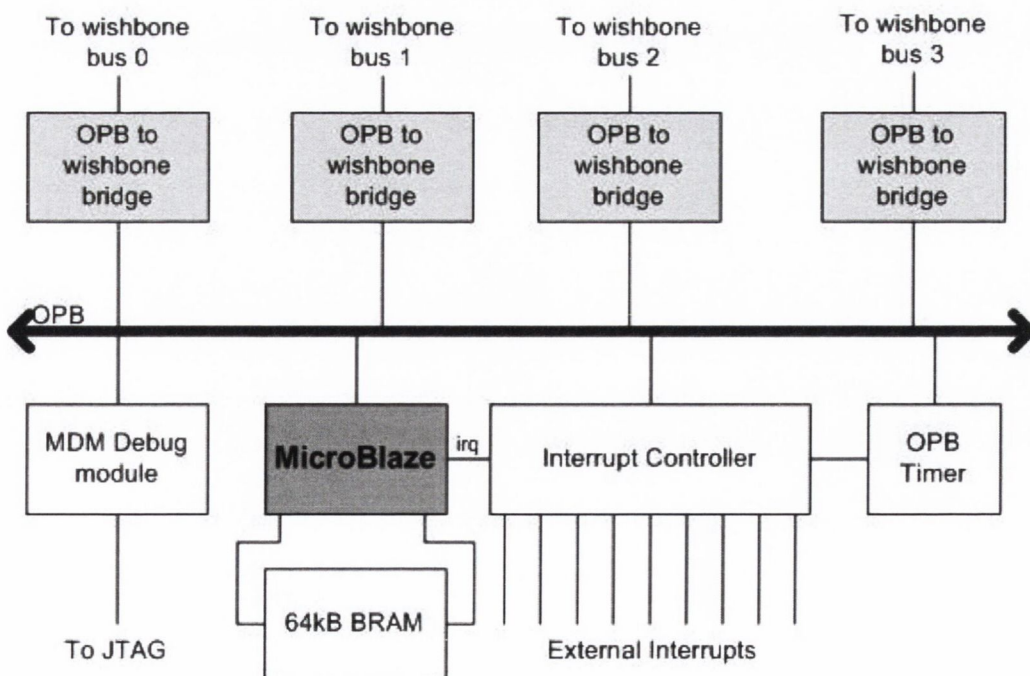


Figure 5.2 MicroBlaze controller unit

The major features of the MB controlling system are the Interrupt controller, the debug module and four OPB to Wishbone bridges.

The interrupt controller is connected to the OPB and allows the PC and the hardware units to issue interrupts to the MB. The interrupt controller has 9 interrupt lines that can be connected to any hardware units on FPGA. If the hardware unit wants to issue an interrupt to the MB the unit must set the corresponding interrupt signal high and an interrupt handler on the MB corresponding to the interrupt line set will carry out the operations required. Once the interrupt handler has been completed the MB will clear the interrupt. This allows hardware units to inform the MB that an operation is complete. The MB also has an interrupt line connected to the host PC which allows it to issue an interrupt when it has completed the operation. This allows the host PC to continue working on other calculation while it waits for the FPGA development platform to complete the PageRank calculation.

The MDM debug module allows the user to connect to the MB using a JTAG cable. This connection can be used to program the MB program memory and to debug systems running on the MB.

Finally, four OPB to Wishbone bridges allow for communication between units connected to the Wishbone buses and the MB. The MB is designed for use with the OPB which is a 32-bit wide bus. The OPB therefore is not wide enough for the data being used by the hardware units. It was for this reason that the WB buses were implemented to allow data to stream to the hardware units. The MB, as the system controller, must be able to communicate with all hardware modules and memory interfaces on the WB buses, thus an OPB to WB Bridge was implemented between the OPB bus and each of the four WB buses. However, connecting the OPB bus to the Wishbone buses is difficult as the OPB bus is only 32 bits wide and uses different control signals than the Wishbone bus system. The bridges therefore need to coordinate communication between the two systems. The bridge converts all data and control signals to the desired width and implements the correct protocol for both buses allowing information to be passed between the two bus systems.

5.2.2 Wishbone Bus

The Wishbone Bus (WB) is an open source hardware communication bus that was designed to provide a standard way for hardware cores to communicate with one another [16]. The Wishbone standard does this by standardising the communication process between cores. The WB is designed to be simple, flexible and portable. It was created to allow designers to release their IP cores with a standard communication interface. This means that functional cores with WB interfaces can be easily implemented as black boxes in any circuit with a WB.

It was decided to use the WB standard in this project when problems with compatibility with other buses became apparent. The Alpha Data development platform came with IP-core buses for use with the power PC available on the FPGA. However, no bus larger than 32 bits was provided for use with the MicroBlaze.

The WB is a master/slave architecture [16]. The masters initiate data transactions to participating slave interfaces. There are a number of ways the WB modules can be interconnected including point-to-point, data flow, shared bus or crossbar switch.

Point-to-point interconnects can be used when a master needs to communicate with exactly one slave device. The Data-Flow interconnects are used when each unit needs to be a slave to one unit and a master to another unit. The Shared Bus topology is used when multiple masters need access to one or more common slave interfaces. Only one connection can be made at a time using this topology so sometimes masters must wait for the bus to become free. Finally the Crossbar Switch allows multiple masters to connect to multiple slaves but it allows more than one connection to be made at a time so long as it is not to the same slave interface. The Shared Bus topology is used in this work since multiple units (masters) need access to multiple slaves. It was decided that the Crossbar Switch would require too much logic to implement and that none of the buses would require multiple accesses to different units. Figure 5.3 shows a high level block diagram of how the shared bus is implemented in this project. The WB standard does not specify how the shared bus should be implemented and so a multiplexed bus is used in this work.

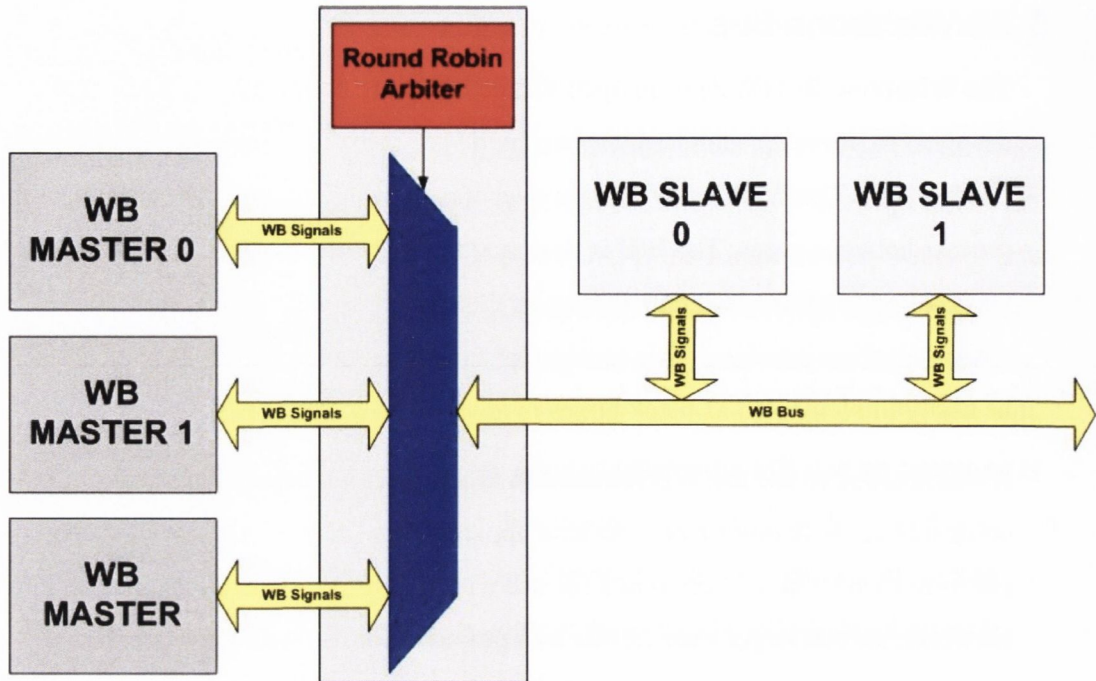


Figure 5.3 Wishbone multiplexed Shared bus with round robin arbiter.

The arbiter uses a round robin scheme to choose which master gets control of the bus. It cycles through each of the masters until it finds a master that needs the bus. The arbiter then multiplexes the signals from the master to the shared slave bus, thus giving the master control of the bus. When the operation is complete, the arbiter starts to cycle through the masters again until it finds another master in need of the bus.

This project has implemented 96, 128 and 192 bit Wishbone buses. The 96 bit bus is used in the SPAR and single MAC SCAR units described in Section 5.4 and Section 5.5 respectively. The NZE of the matrix requires 96 bits to represent them and so this bus delivers a single NZE every clock cycle. The 192 bit bus delivers two NZE per clock cycle to the dual MAC SCAR described in Section 5.5.3. This was implemented to utilize the full memory bandwidth with increasing the FPGA clock rate. Finally, the 128 bit WB is used on the shared vector bus to read/write two 64 bit vector entries to/from the X and Y buffers. The 192 bit bus and 128 bit bus contain an extra signal which tells the slave how many bits to use when transferring data. This is to facilitate writing 32, 64, 96 and 128 bit words across the 128 bit or 192 bit bus. This means the bus can be used to set up the 32 bit registers in the slave units and to stream data from memory which is converted into 192 bit words in the memory interfaces. The Wishbone buses in the system are connected to the

arithmetic units using master and slave interfaces which means that the arithmetic units do not need to deal with implementing the Wishbone standard internally. Instead all aspects of the WB standard are enforced in the WB interfaces.

5.2.3 Wishbone Bus Interface

All hardware units in the system communicate across the Wishbone bus. The Wishbone interface facilitates this communication by implementing the Wishbone protocol. A Wishbone interface can either be a master or a slave. A master Wishbone interface allows the unit to control the bus. Once a unit has control of the bus it can access any unit with a slave interface to read data from it or write data to it. Conversely any unit with a slave interface can be accessed using the Wishbone bus to write data to it or to read data from it. Some of the hardware units only have a slave interface or master interface and others have both. The memory controller is an example of a slave unit. It never needs to control the bus. However, the slave interface allows all the other units with master interfaces to access the memory through the memory controllers slave interface. The arithmetic units have both slave and master bus interfaces. Figure 5.4 and Figure 5.5 show the Wishbone bus slave and master interface for the SMVM unit respectively.

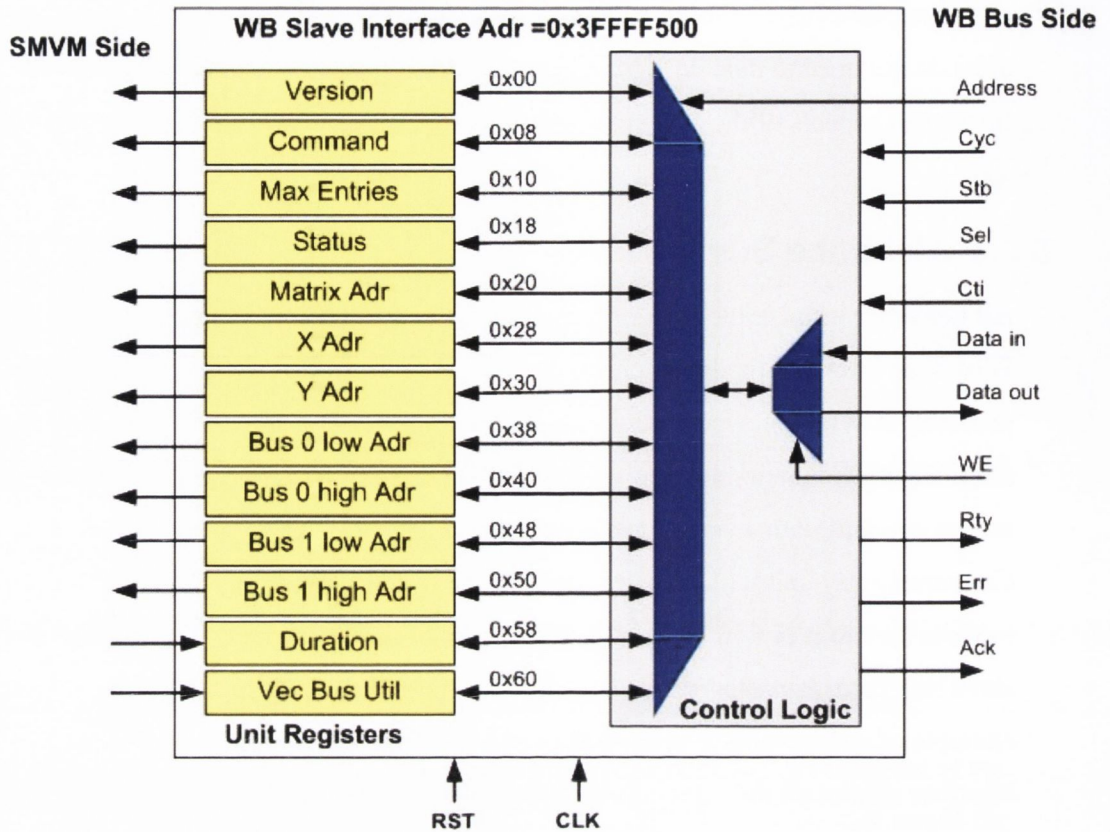


Figure 5.4 Wishbone slave interface for SMVM unit

The WB slave interface in Figure 5.4 has a fixed address space in memory; in this case, the addresses 0x3FFFF500 – 0x3FFFF560 map this slave interface. Any unit connected to the WB bus can access this slave interface by referencing these addresses. If an access cycle begins on the WB bus with a given slave's address, that slave interface responds. After the WB hand shaking is complete (as described in section 5.2.2) the data is either read or written to the unit register corresponding to the address. Multiple registers can be written to, or read from, using the WB burst operation. Data in the unit registers is available to the rest of the unit, in this case the SMVM unit. These registers are used to set up the hardware block (e.g. an SMVM unit) for a given calculation. These registers make the arithmetic units highly flexible in operation. The Vector Unit for example can be programmed to do any one of sixteen operations just by changing a single register entry. The Vector Unit also allows the user to set the position of the data in memory, the rounding mode and whether or not the unit should interrupt the MB when the operation is completed. This is discussed in more detail in Section 5.3.

The MB therefore connects to the units via this slave interface. The MB sets up the calculation by passing parameters to the arithmetic units registers via their slave interfaces. Once the arithmetic unit's command word is written the arithmetic units take control of the bus via their master interface to read in data needed from memory see Figure 5.5.

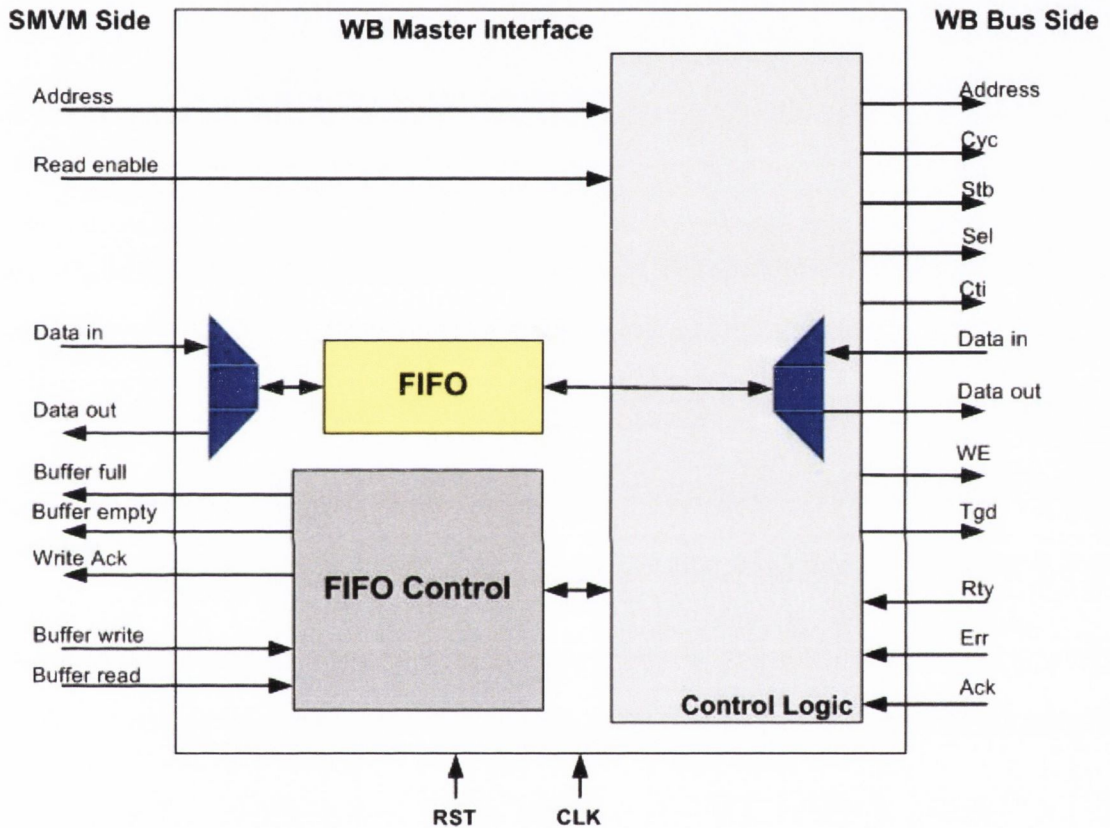


Figure 5.5 Wishbone Master Interface for SMVM unit

The WB master interface allows a unit to control the Wishbone bus. On the unit side (SMVM side in Figure 5.5) the master interface is simply a FIFO with some control signals. There are two modes of operation for the master interface - the read mode and the write mode. If the unit needs to read data from another unit i.e. memory, it must set the read enable signal and put the address of the data on the address input. The control logic then initiates a WB read cycle to get the data from the address provided. The data is sent on the bus and placed into the FIFO which causes the buffer empty signal to go low. This lets the unit know that the data is available and it begins to read the data from the FIFO. Once the required data has been read, the read enable signal is set low and this lets the control logic know it can finish the read

transaction. To write data to another unit or to memory the data is simply placed in the FIFO with a valid address on the address line. Logic in the FIFO signals to the control logic in the WB master interface to begin a write transaction. Once all the data is written the transaction is terminated. In the next section the WB signals and memory transactions are discussed.

5.2.4 Wishbone Bus Operations

The WB standard describes three types of bus cycles - single read/write, block read/write and read-modify-write. In this implementation only single read/write and block read/write operations are needed. The read-modify-write operation is not needed due to the functions of the units in this system. The SMVM or Vector unit never need to make read-modify-write operations. They either are streaming blocks of information in or writing blocks of information out. The SDRAM overhead as discussed in Section 2.4 would also make a read-modify-write operation to SDRAM prohibitively long.

When a master wants to initiate a single memory read cycle it places the address of the data on the address output (ADR_O) and then sets the Strobe (STB) and cycle (CYC) signals high and sets the write enable signal (WE_O) low, see Figure 5.6. In this implementation of the WB an optional cycle tag instruction (CTI_O) is also set. This informs the slave what type of data transfer is being attempted. The CTI_O signal is set to 0x1 for single read/write operations and to 0x3 for block read/write operations. After these signals have been set, the master must wait for the slave unit to reply. This wait state can last numerous clock cycles depending on delays in retrieving data. These delays include delays due to bus arbitration, data availability and bus latency. The slave units constantly monitor the address line of the Wishbone bus and if they see an address in their range on the bus with a valid STB signal they place the data on the data lines of the bus and set the acknowledge signal (ACK). Upon seeing an ACK signal, the master must latch the data on the data in (DAT_I) lines and negate its STB signal in response to the ACK signal. The transaction is now complete so the bus is released by negating the CYC signal. The arbiter knows the transaction is complete when the CYC signal is negated and so gives the bus to the next master in the queue.

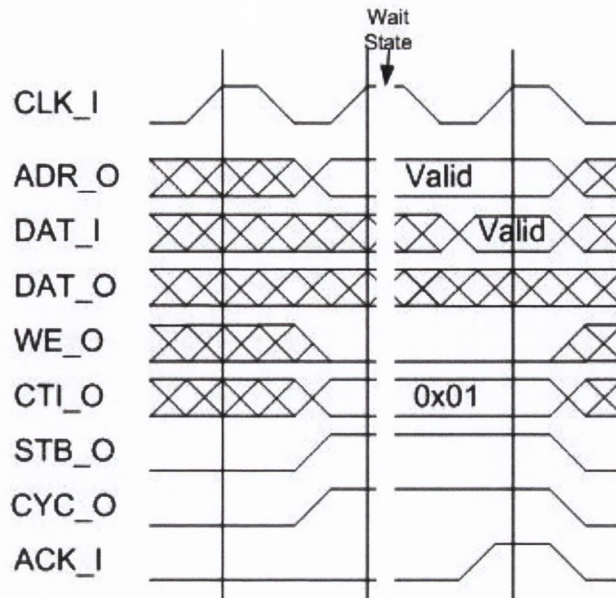


Figure 5.6 Wishbone single read timing diagram

The single write operation is quite similar to the single read. In the write operation the master must set the WE_O signal and put the data to be written on the DAT_O bus lines. The slave sets the ACK_I signal once it has latched the data and so the operation can be ended safely. Burst read/write transactions are quite similar to the single read/write transactions. Figure 5.7 shows a burst write Wishbone transaction. Once again the master must set the STB_O, CYC_O, WE_O and CTI_O signals along with assigning the data output channel (DAT_O) with the first piece of data and setting the address line (ADR_O). The CTI_O signal is set to 0x3 to tell the slave it is a burst transaction. The master unit then waits for the slave unit to acknowledge that it has latched the data on the bus. The slave does this by asserting the ACK_I signal.

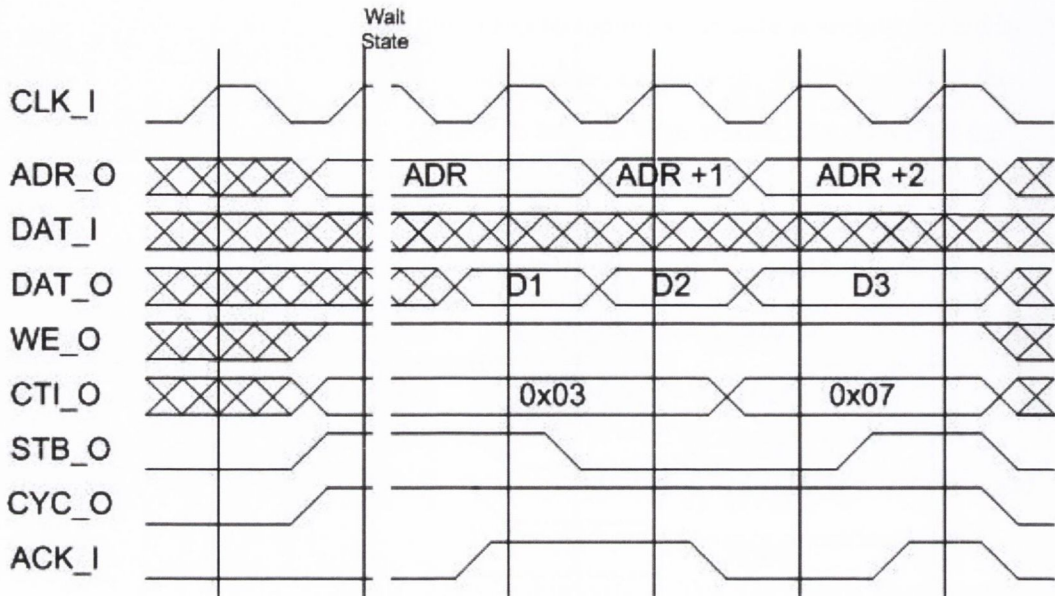


Figure 5.7 Wishbone block write timing diagram.

After receiving the ACK_I signal from the slave, the master negates the STB_O signal as it did in the single read/write transaction to acknowledge that it received the ACK_I signal. The master can then increment the address and place the next piece of data on the DAT_O lines. The master must wait for an acknowledgement before it can move on to the next piece of data. This is why in Figure 5.7 that D3 remains on the data lines for two clock cycles. ACK_I is not set at the end of the first clock cycle and so the master must wait until the slave acknowledges receiving the data before it can move on. Another important note is that the CTI_O signal is set to 0x7 for the last piece of data in the burst, thus warning the slave interface that this will be the last piece of data to be sent in this transaction. When the master receives the ACK_I signal for the last piece of data, the transaction can be terminated by negating the CYC_O signal. This frees the bus for use by another master.

5.2.5 PCI Interface

The PCI interface allows for communication between the SOC and the host PC, see Figure 5.8. The PCI interface is connected to all 4 Wishbone buses to allow it access to all 4 banks of DDR. It contains both master and slave Wishbone interfaces. It can control the Wishbone buses using the master interface when it needs to write data to the banks of memory or if the arithmetic units are being controlled via the host PC. The slave interface allows the MicroBlaze to control the PCI interface. This can be

used to allow the MicroBlaze to communicate with the host PC or to control the downloading of data. The PCI interface operates in two different clock domains. The local bus interface operates at 66 MHz which is the speed at which the PCI bus operates. The Wishbone interfaces, flow control and control registers are clocked with the system clock which is the clock the FPGA architecture uses. A pair of asynchronous FIFOs is used to avoid any errors associated with crossing the clock domain.

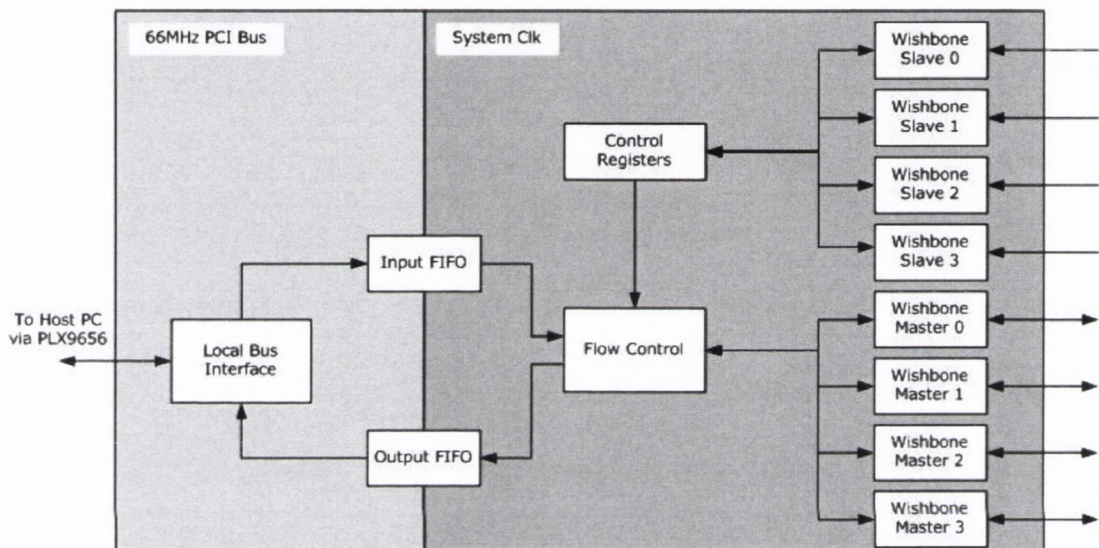


Figure 5.8 Block diagram of PCI interface

5.2.6 Memory Controller

The memory controller is responsible for communication between the Wishbone bus and the DDR. The controller consists of a Wishbone slave interface with associated slave control registers. These can be used to set parameters for control of the memory cycles. Figure 5.9 shows a high level block diagram of the memory controller logic and the Wishbone interface. As discussed in the general architecture description, the SDRAM have a separate clock from the rest of the system. Therefore there are two separate clock domains in the memory controller. The first is clocked at the clock speed required by the SDRAM (e.g. 133 MHz for DDR-266; see Table 2.2). The second clock in the memory controller is the system clock. This is the clock rate that the arithmetic units, Wishbone buses and Wishbone interfaces are capable of achieving. Cross-over of data between these two clock domains is handled by an asynchronous FIFO. Data is latched in the FIFO in one of the clock domains. It then

can be read out of the FIFO in the other clock domain. This asynchronous FIFO is also able to deal with varying data word size. The SDRAM, for example, read and write 128-bits of data per clock cycle. However, some of the Wishbone buses read and write 192-bits of data per clock cycle. This FIFO system can deal with this conversion from 128-bits to 192-bit data words where necessary.

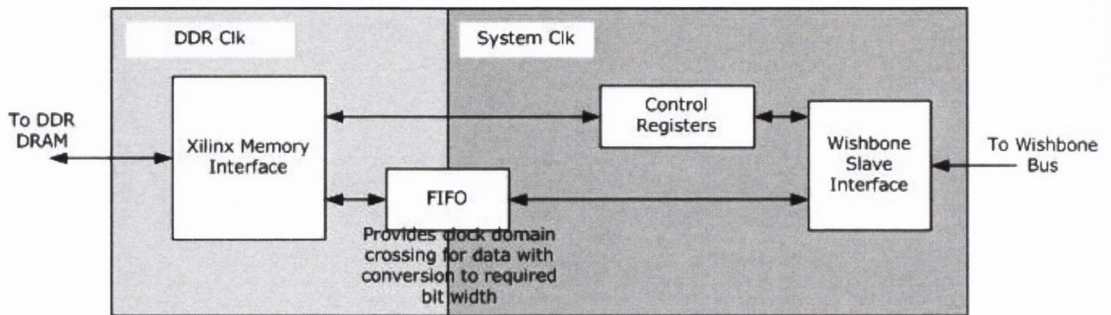


Figure 5.9 Overview of memory controller

The signal level communication with the SDRAM is accomplished using the Xilinx Memory Interface core. This core was generated using the Memory Interface Generator (MIG) tool [34]. The MIG tool generates HDL files for user specified configuration of SDRAM. These files take care of all the read and write protocols of the SDRAM. The memory controller is wired to connect the Xilinx memory interface to the Asynchronous FIFO so that data can be read from and written to the SDRAM. Once the data has been passed through the Asynchronous FIFO to the Wishbone bus it is available to the arithmetic unit that requested the data. In the next few sections the arithmetic units implemented in this project will be described.

5.3 Vector Unit

The Vector Unit is a dedicated arithmetic unit that can compute the operations detailed in Table 5.1 i.e. practically any vector-vector or scalar-vector or scalar-scalar operations. A high level diagram of the Vector Unit is shown in Figure 5.10. The Vector Unit consists of a single multiply-add unit, 3 master Wishbone interfaces, a slave Wishbone interface and a controlling state machine. For simplicity, the details of the state machine have been omitted from Figure 5.10.

Table 5.1 Vector unit functions

Operation	Op-Code	C-Code	Description
NOP	0x0	nop	No Operation
$z=x$	0x1	memcpy	Memory Copy
$z=\langle x,y \rangle$	0x2	dot	Dot Product of vectors x and y
$z=x.*y$	0x3	elem	Element by Element multiplication of vector x and y
$z=ax$	0x4	vscale	Vector scale, scale vector x by scalar a
$z=ax+y$	0x5	axpy	Vector scale and addition. Scale x by a and add vector y
$z=ax-y$	0x6	axmy	Vector scale and addition. Scale x by a and subtract vector y
$z=wx+y$	0x7	wxpy	Element by element multiplication of vector x and w and result added to y
$z=wx-y$	0x8	wxmy	Element by element multiplication of vector x and w and subtract y from result
$z=\langle x,x \rangle$	0x9	norm	Norm squared of x
$z=\alpha*\beta$	0xA	smult	Scalar Multiply
$z=\alpha+\beta$	0xB	sadd	Scalar add
$z=\alpha-\beta$	0xC	ssub	Scalar subtract
$z=\alpha*\beta+\gamma$	0xD	smadd	Scalar multiply and add
$z==\alpha?$	0xE	scomp	Scalar Compare
$z=0$	0xF	zero	Zero memory

Like the slave interface discussed in section 5.2.3, the Wishbone slave interface contains a number of registers which can be used to setup the vector unit. These registers include a command register, a vector dimension register, a status register and 8 general purpose registers. The general purpose registers are used to store the addresses of the vectors in memory for vector operations or the scalar value in scalar operations. The registers are 32 bits wide with a single register being used for address data and a pair of registers for double precision scalars. The state machine controls how the Vector Unit interprets these registers depending on the operation being carried out. Figure 5.10 shows that each input to the multiplier and adder can be connected to one of the slave registers or via a Wishbone interface to memory depending on the operation (the “op” signal in Figure 5.10). If the input is a vector the address of the vector is available in one of the slave registers, otherwise the scalar value is stored in the slave registers. The state machine is responsible for setting the calculation up in the correct way depending on the operation requested via the command word. Table 5.1 shows the range of functions that the Vector Unit can

compute, together with the corresponding C function name used by the MB to set up the calculation. Many of these functions are needed to compute the PageRank algorithm.

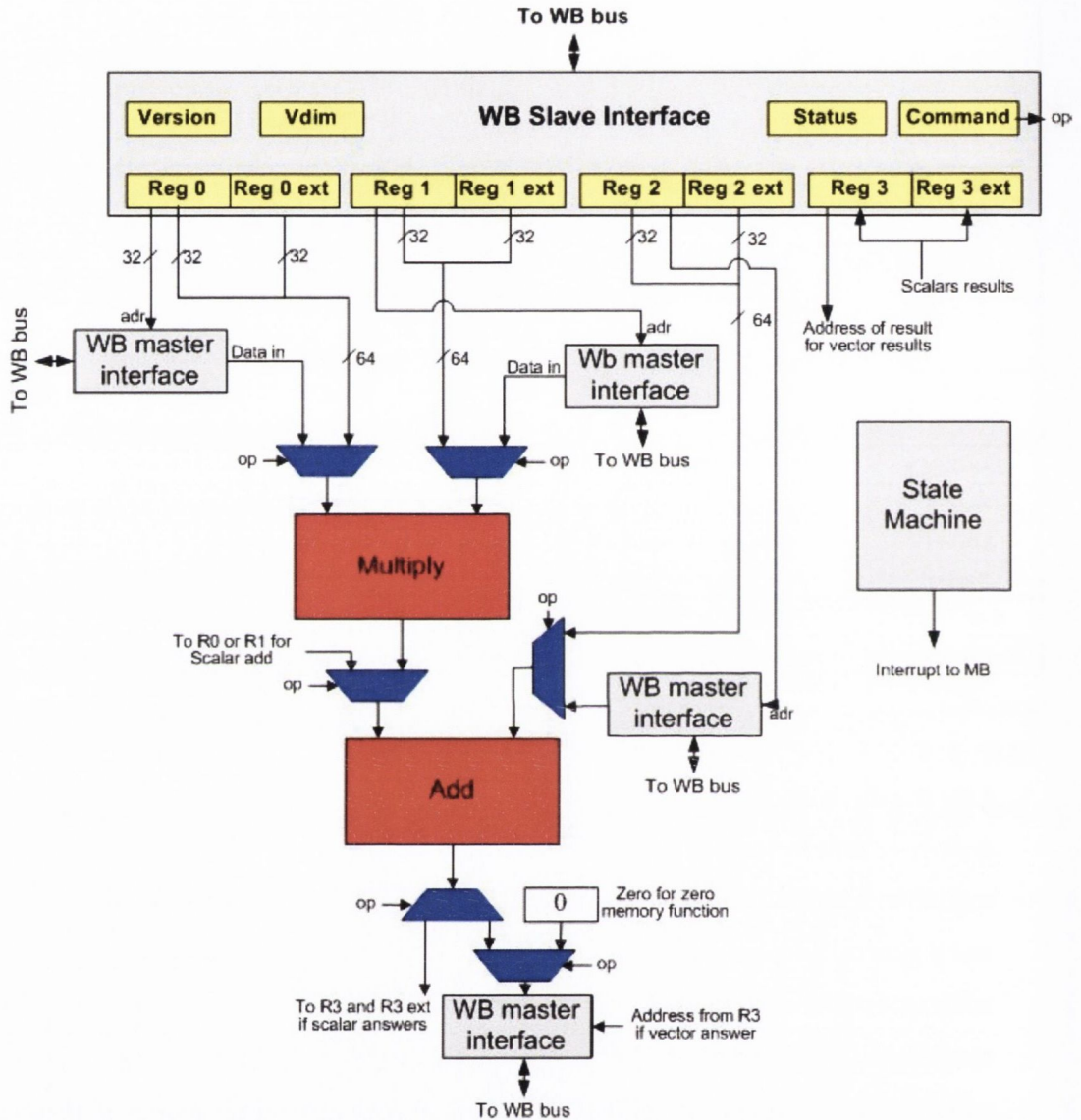


Figure 5.10 Block diagram of Hardware Vector unit

To start the operation, the command word register in the Wishbone slave interface must be set after the other slave registers have been given values. The command word contains all the information the Vector Unit needs to start calculating the answer. The command word is 32 bits long. Each bit in the command word corresponds to some aspect of the calculation and is used to control the hardware; it is formed as in Figure 5.11.

	irq	mult-add rounding	Op-Code	o/p	i/p1	i/p2	i/p3
Bit(s)	31	30-28	27-16	15-12	11-8	7-4	3-0

Figure 5.11 Vector unit command word

If an interrupt is required when the unit has finished bit 31 is set. The IEEE compliance and rounding mode for the adder and multiplier are set in bits 30, 29-28 respectively. There are four rounding modes to choose from: round to zero, round to infinity, round to negative infinity and round to nearest even. Their codes are 00,01,10,11 respectively. One of the op-codes from Table 5.1 is written to bits 27-16 of the command word. This controls which operation is being calculated. The remainder of the bits are used to signify which of the general purpose registers holds the address or the data for the output and input scalars and vectors. There is no need to specify if the input is a scalar or a vector as the state machine can infer it from the requested operation. For example, to compute $z=ax+y$ with an interrupt and IEEE compliance and round to nearest even rounding mode, the command word is worked out as follows:

- Bit[31] =1 – want interrupt
- Bit[30]=0 – IEEE compliant
- Bit[29:28]=0x3 – Round to nearest even
- Bits[27:16]=0x5 – Operation $ax+y$
- Bits[15:12] = 0x8 – Address for result in R3
- Bits[11:8] = 0x1 – a is in R0 and R0 ext
- Bits[7:4] = 0x2 – address for x is in R1
- Bits[3:0] = 0x4 – address for y is in R2
- Complete command word is 0xB0058124

Once the command is given the Vector Unit starts to compute the result. Firstly, if a vector is required for the calculation the Wishbone interface accesses the DDR. The address of the vector in memory is stored in one of the general purpose registers in the slave interface. For scalar operations the value needed is stored in the general purpose registers in the slave interface and so this is applied to the adder or multiplier input. The data is streamed through the adder and multiplier according to the

operation being carried out. If the result is a scalar, such as in a dot product calculation, the result is stored in one of the general purpose registers. If the result is a vector it is streamed out to DDR as it is calculated and a pointer to the vector is stored in the general purpose register. Once the input vectors have finished streaming through the adder and multiplier, the Vector Unit will generate an interrupt to the MicroBlaze, if it has been set to do so in the command word, and then it returns to its idle state.

5.4 SMVM Architecture 1 - SPAR

The SMVM operation performance is pivotal to the performance of the PageRank algorithm. In section 3.6 the SMVM operator was described and a discussion of storage formats was introduced. The SMVM operation can be column based (i.e. all NZE in a column are processed before moving on to another column) or row based (i.e. all NZE in a row are processed before moving on to a new row). In this work three architectures were implemented and tested - one of which is column based and the other two are block row based. The column based architecture, SPAR, is described here and the block row based architecture is discussed in section 5.5.

Taylor et al., first described the SPAR architecture in 1995 [17]. The SPAR architecture consists of a specialised storage format and a hardware unit optimised for SMVM. Taylor found that using specialised hardware together with a specially designed storage format could greatly improve the peak performance of SMVM. Taylor never built the SPAR architecture. Instead the results were obtained from an RTL simulation. The SPAR data format is outlined in Figure 5.12.

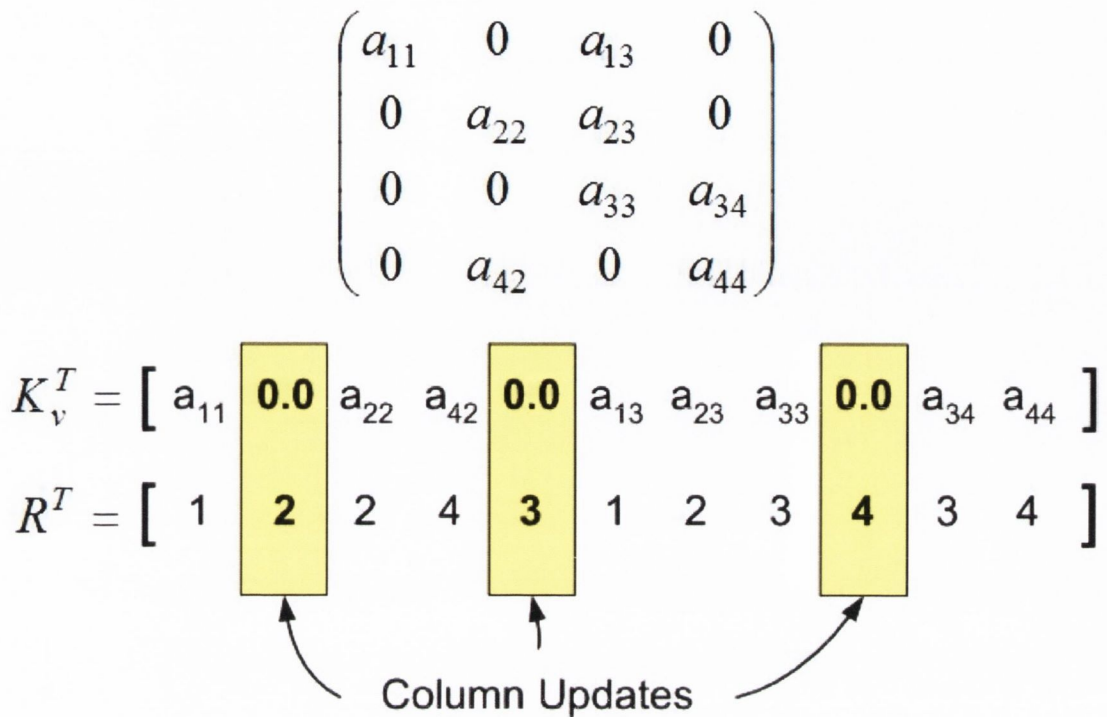


Figure 5.12 SPAR storage format

The matrix is split up into two vectors, the K vector and the R vector. The K vector contains the NZE of the matrix sorted by column. The end of each column is denoted by a zero in the K vector. The R vector stores the row address of the non-zero entries of the matrix. The entries in the R vector that correspond to the column updates (zeros) in the K vectors give the address of the next column being calculated. Taylor embedded the column update information in the K and the R vectors. Other commonly used sparse matrix storage formats like CRS and CCS (see section 3.6.1) use three vectors. Using two vectors instead of three is advantageous to a streaming SMVM architecture as it reduces the number of memory channels needed by the matrix.

Figure 5.13 shows a block diagram of the SPAR architecture which is designed to work with the sparse matrix representation just described. The SPAR unit as described by Taylor consists of a floating point adder and multiplier, a cache, 3 vector fetch units (VFU) and zero detection logic (ZDL).

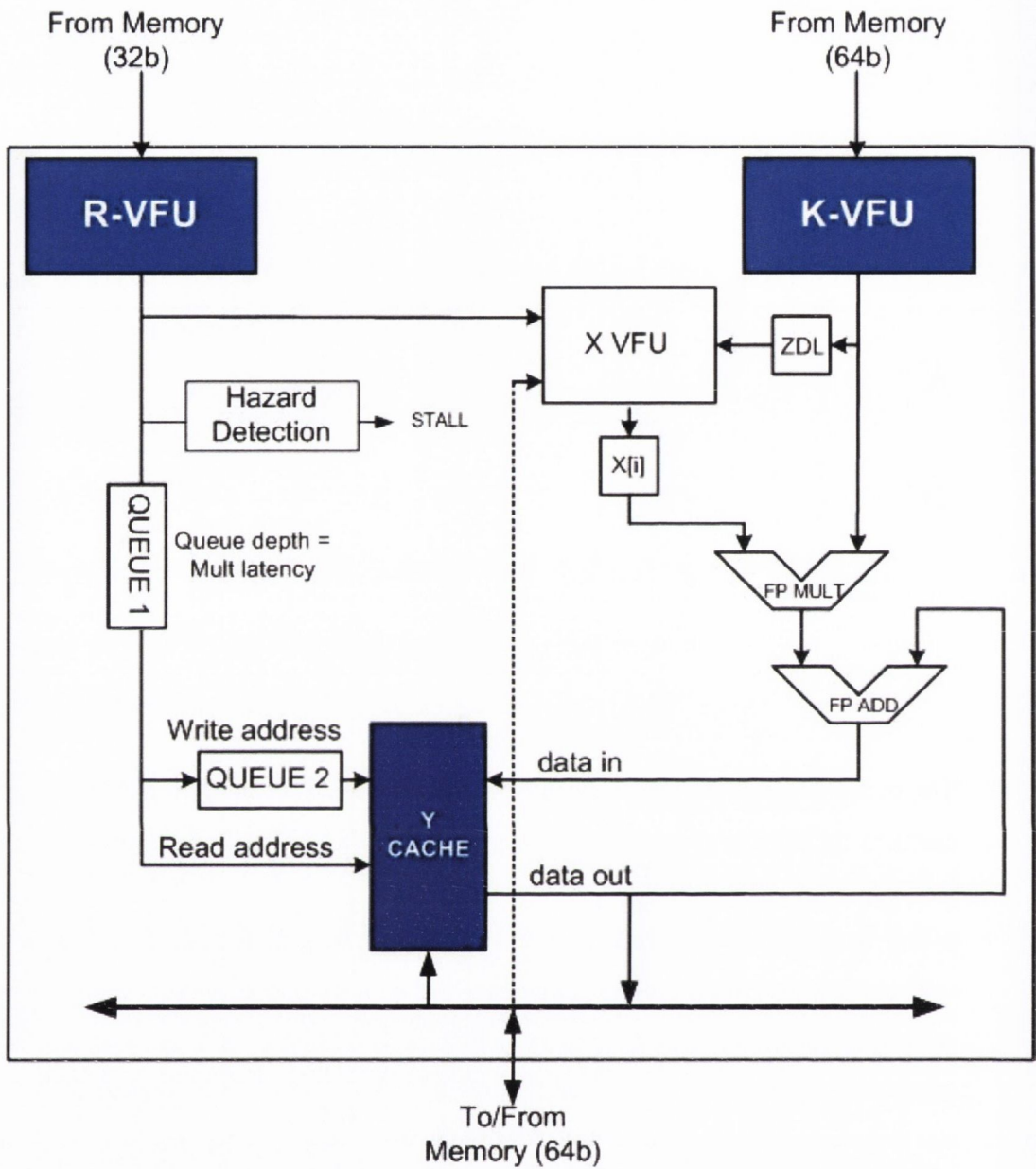


Figure 5.13 SPAR architecture

The Vector Fetch units read in the K , R and X vectors which contain the matrix NZE , the corresponding row address and the X vector entry for the current column respectively. The NZE value is multiplied by the current X value. The result is then added to the corresponding row entry in the Y -cache as denoted by the R vector entry. There are two delay queues in the system. The first delay queue, Queue 1 in Figure 5.13, is used to co-ordinate the multiplier output with the Y -cache output. This ensures that the product from the multiplier is added to the correct value from the Y -cache. The second queue delays the addresses on the write address input of the

Y-cache. This ensures the correct write address is available on the Y-cache write address line when the addition is complete. The ZDL checks the K data stream for zeros which denote the end of a column. If a zero is found, the multiplier and adder are stalled and a new X value corresponding to the column address in the R vector is fetched from memory. Column updates stall the system but once a new value of X -vector has been read in, the system can immediately restart without further operations.

An 8 kB, direct mapped, write back cache is used in this implementation of SPAR. The SPAR cache is a two-cycle cache. The first cycle is used to check if the data needed is in the cache. The data is outputted on the second cycle if the data is in cache. If the data is not available in the cache, a Y-miss signal stalls the multiplier and K and R vector fetch units. The adder is flushed to ensure lines being used by numbers in the adder are not replaced. If the cache line replaced by a Y-cache miss is referenced by a number in the adder pipeline, two further Y cache misses would occur. The first additional Y-cache miss occurs when the number which was in the adder pipeline is written back to the cache. Since the cache line has been replaced a Y-cache miss must occur to fetch the correct (previous) line from memory. By this time, the new data is now in the adder pipeline and so causes a second similar Y-cache miss when the system attempts to write it to the Y-cache. Y-cache misses reduce the performance of the system significantly. Every miss involves a block write to memory to write back the current cache line to memory and a block read to read in the new cache line. The delay caused by flushing the adder is minimal compared to the delay that would be caused by the two additional Y-cache misses. Thus the adder is flushed on every Y-cache miss.

The cache has an in-built flush signal which when set causes the entire contents of the cache to be written out to memory. This operation is used to flush the cache at the end of the SMVM operation. The SPAR's slave register contains the number of NZEs in the matrix. Once the SPAR unit has processed this number of data elements it sets the flush signal to terminate the calculation.

Hazard detection logic was added to SPAR to avoid Read After Write (RAW) hazards. RAW hazards occur when a Y value that is currently in the adder pipeline is inadvertently referenced again. Because the first operation is incomplete at this point, the value in the cache is not up to date when the second operation accesses it. The

update already in the adder pipeline will be lost, thus making the final answer incorrect. Figure 5.14 shows how a RAW hazard could occur in a system with a two cycle adder. In step one address 1 is accessed and incremented. This same operation occurs again in step two. The current value for address 1 is still in the adder pipeline and has not yet been written back to the cache so this second increment instruction gets the old value for address 1 (4 in this example) instead of the new value for address 1 (5 in this example). The value from the first calculation is written back to the cache in step three and overwritten by the incorrect value in step four.

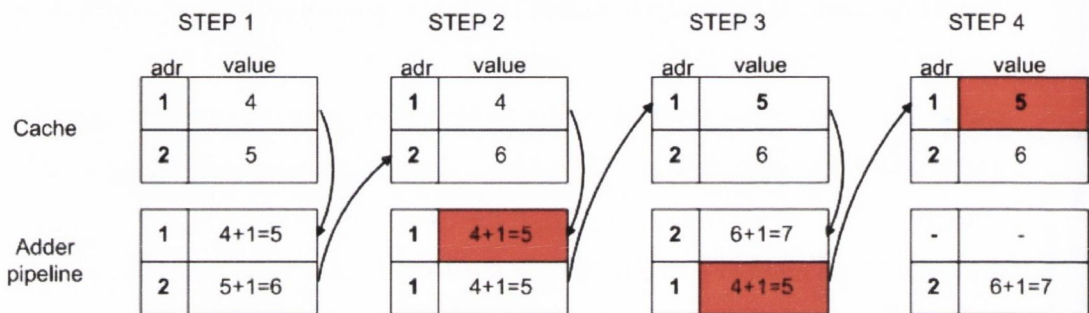


Figure 5.14 Example of a RAW hazard.

The hazard detection logic in SPAR checks for these RAW hazards. Upon finding a possible hazard, it issues a stall command to the input VFU and multipliers. Once the adder has flushed its pipeline, the detection unit clears the stall command and allows the SPAR unit to continue. In this way, the potential RAW hazard is successfully averted.

5.4.1 SPAR Modifications

Initial testing on the SPAR architecture showed that the SPAR as originally described by Taylor performed poorly when implemented on FPGA. These poor results were largely attributed to the overhead involved in reading and writing to SDRAM. A number of modifications were made to Taylor's design to reduce the impact of these overheads.

The first of these changes was the inclusion of an X-buffer to reduce the number of memory reads associated with Column updates. The new system reads in 1024 X-vector values at a time. Column updates only need to access memory if all the values in the X-buffer have been used. This reduces the number of Column updates that access memory and thus reduces the time taken to do the SMVM calculation.

The second modification was implemented to reduce the number of Y-cache misses. In large matrices, where not all the Y values can fit in the cache, multiple Y values are mapped onto the same position in the cache. Initial testing of the SPAR showed that lines in the Y-cache were often replaced only to be read back in again moments later. Cache line replacements are doubly expensive in terms of cycles used as they require the old cache line to be written out before the new line can be read in. To reduce these cache misses a simple reordering scheme is introduced. The matrix is divided into cache size stripes as in Figure 5.15. The SPAR unit then processes these stripes. Since the stripes only contained enough rows to fill the cache, no cache misses occur. This scheme does increase the number of X-buffer misses, but since the X-buffer is a read only buffer there is no need to write the data out to memory before reading the new X values in. This scheme increased the performance of the SPAR architecture considerably.

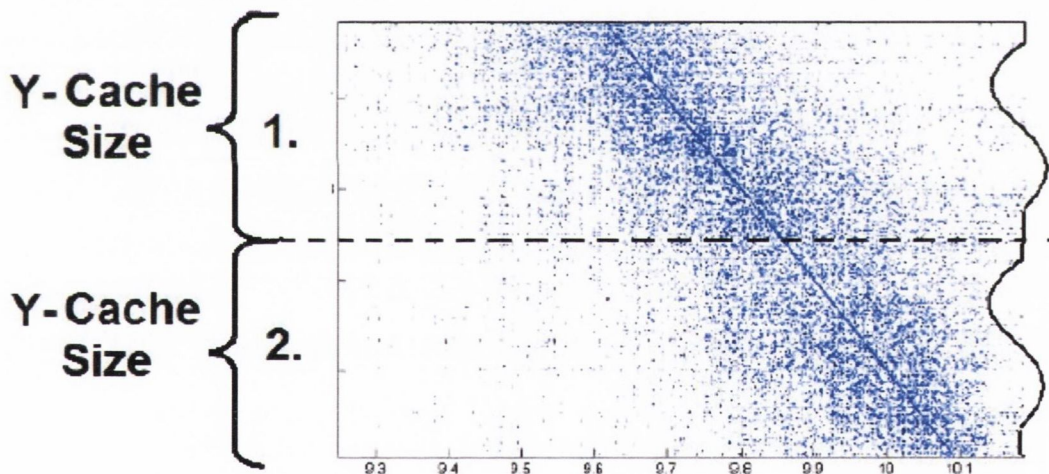


Figure 5.15 SPAR strip reordering scheme to eliminate Y-cache misses

5.5 SMVM Architecture 2 - SCAR

The SPAR architecture is a column based architecture. All the NZEs from a column are processed before moving on to another column. This method gives maximum X usage since each column corresponds to a single X vector entry. In large matrices where the Y-vector cannot fit in cache there can be many Y-cache misses. The Y-cache misses have a large overhead because they are write-read memory cycles and so a great deal of time is wasted reading and writing data to the Y-cache.

In an attempt to reduce the time wasted performing Y -cache misses, the second SMVM architecture implemented is a block-row based architecture. In row based architectures the NZEs are processed in rows instead of columns. This eliminates Y -cache misses since all NZE in a row affect the same Y vector entry. This Y -vector locality is gained at the expense of increased X -buffer reads from memory. The X -buffer is read only and as such memory accesses have less overhead than the write-read cycles of the SPAR unit. The row based architecture is very susceptible to RAW hazards since the NZE in a row affect the same Y value. To avoid RAW hazards a block-row SMVM algorithm called SCAR was implemented.

Software Controlled Arbitrary Reordering (SCAR) is a block row SMVM architecture. The SCAR system uses its own matrix compression/reordering scheme in conjunction with a companion architecture. The reordering scheme used is designed to ensure that the hardware arithmetic units are running at maximum efficiency while avoiding RAW hazards [115]. In order to do this, the SCAR system tiles the matrix in a pre-processing stage as shown in Figure 5.16. These tiles are processed one at a time by the SCAR unit. Each tile is related to a fragment of the X and Y vectors. In Figure 5.16 each tile is colour coded and numbered to show which fragment of the X vector is associated with it. For example, tiles 1, 2 and 3 correspond to the $X1$, $X2$ and $X3$ fragments of the X vector respectively. Each row of tiles is associated with a fragment of the Y vector. Tiles 1, 2 and 3 all are associated with the $Y1$ fragment of the Y vector. The hardware has local memory buffers to act as caches for these fragments of the X and Y vectors.

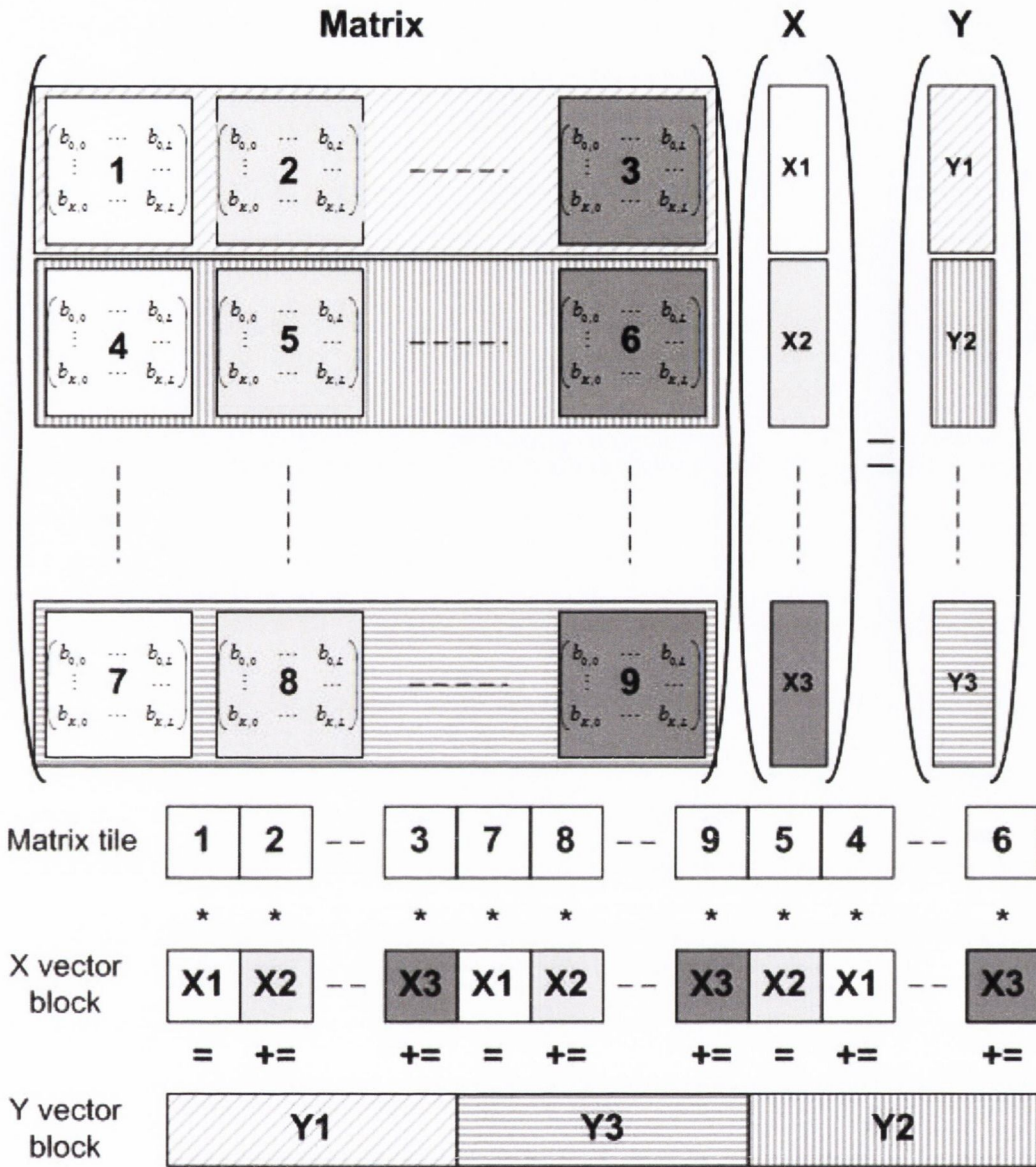


Figure 5.16 $N \times M$ matrix divided into blocks for SCAR unit

Processing of the tiles is done on a stripe by stripe, block by block basis. All the tiles in a stripe are processed before moving to a new row of tiles, since all tiles in the same stripe are associated with the same fragment of Y vector. At the end of a stripe the corresponding Y -vector fragment is complete and is written to memory.

Simultaneously the Y -buffer, which is implemented as a dual port memory, is zeroed and proceeds to the next stripe of matrix tiles. In Figure 5.16 we can see that each tile in any given row is associated with different fragments of the X vector. The associated fragment of the X vector is read from memory and placed in the X -buffer at the beginning of every tile. The X and Y vector fragments are now in the buffers, thus the memory channel is left free to stream the matrix data. Therefore a local copy

of the matrix entries is not needed. The matrix stripes, as well as the tiles within them, can be processed in any order as long as the stripe is completed before the next stripe is started. Maximum Y value reuse is achieved using this method.

The VFUs for the X and Y buffers share a single channel of memory. Since the Y -vector fragment is only written out to memory at the end of a stripe, the Y -vector has very low memory bandwidth requirements. Most of the traffic on the bus to the shared memory is due to fetching X -vector fragments. As discussed previously this is read-only data. The SCAR therefore does not utilise write-then-read memory operations like the SPAR does. Both the read and write operations have a much smaller overhead than the write-then-read operation used by the SPAR and, as such, the SCAR memory operations have a much smaller overhead and bandwidth requirement than that of the Y -cache misses in the SPAR architecture.

5.5.1 SCAR Data Structure

The SCAR system uses a data reordering scheme which was designed in conjunction with the SCAR hardware architecture. The scheme aims to maximise arithmetic unit utilisation which is a measure of efficiency. The SCAR data structure only uses one vector to store all sparse matrix data. This differs from other sparse matrix storage schemes like the SPAR and CRS which use two and three vectors respectively to store the matrix. An entry in the SCAR stream is made up of three fields, a 2-bit command word, an index field and a payload. Two different SCAR stream entries are shown in Figure 5.17. The size of the SCAR word is dependent on precision, X -buffer size and Y -buffer size. In this implementation the SCAR data word was set to 96 bits, 64 bits for double precision NZE, 2 bits for the command and 15 bits each for the X and Y address.

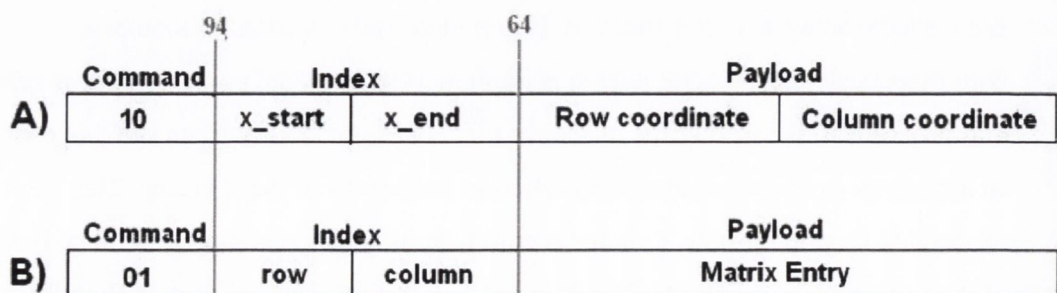


Figure 5.17 SCAR data words A) block update B) matrix data point

The first word in the SCAR data stream must be a block update. A block update command contains the absolute row and column coordinates of the top left hand entry of the block. The matrix blocks of the SCAR architecture are regular and dictated by the X and Y buffer size. However since sparse matrix block may not contain NZE in every column a small optimisation was made. The block update contains the relative addresses to the first and last X entry to be accessed in the block in order to save reading X values from memory for columns that are not needed, thus only x values between these two limits are read into the X-buffer when a new block is loaded.. This information is arranged as shown in Figure 5.17 A. If the word begins with the 01 command as in Figure 5.17 B, then it is a valid matrix NZE entry. In that case, the payload would contain the NZE value and the index sub-word would contain the position of the entry in the block, relative to the top left hand corner of the block. This is also the address of the data in the X and Y buffers. Figure 5.18 shows how the mix of absolute and relative addressing can be used to access the NZE in the matrix.

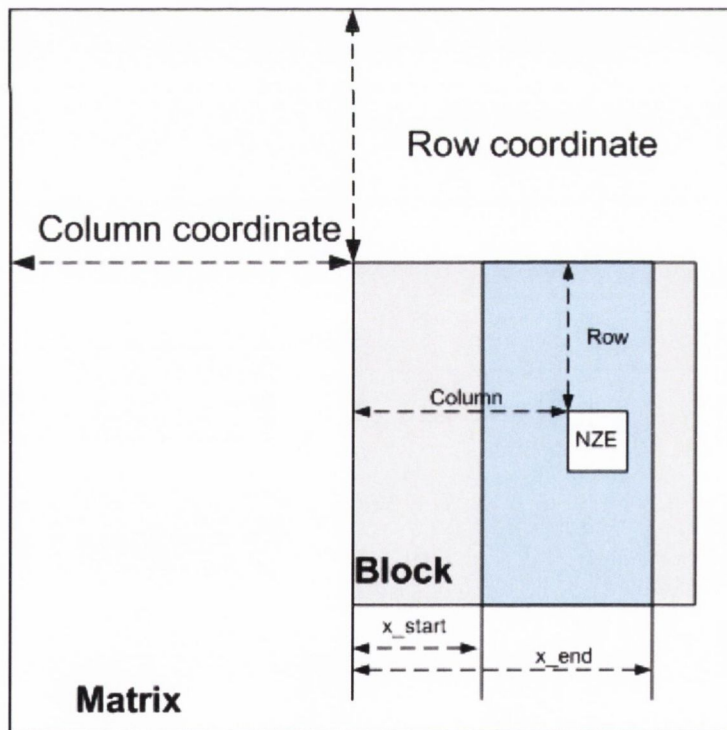


Figure 5.18 absolute and relative addressing as used in the SCAR data format. The column and row coordinates contained in the block update contain the absolute address of the matrix in the matrix and all other addresses are given relative to the block.

If the command word is a NOP (i.e. is equal to 00), however, then the index and payload sub-words can be anything. The data would pass through the data path, but nothing will be added to the Y-buffer. The command word, therefore, is used to decipher the index and payload. Table 5.2 is a summary of possible command words and what they mean.

Table 5.2 The 2-bit command sub-word

Value	Description
00	No-Operation (used for padding)
01	A valid matrix entry
10	A block update
11	Not used

The non-zero elements in a SCAR block can be ordered using any method. They are usually ordered to maximise FPU utilisation. The only goal of the reordering is the prevention of RAW hazards. The initial method used to reorder the SCAR stream was a simple round robin reordering. In this method, a queue is created for each pipeline stage in the adder (e.g. if adder has a 3 stage pipeline, 3 queues are needed). Each row in the block is allocated to one of these queues. The queues are then padded with NOPs to make them all the same length. The SCAR stream is created by taking a value from the top of these queues in a round robin fashion. See Figure 5.19.

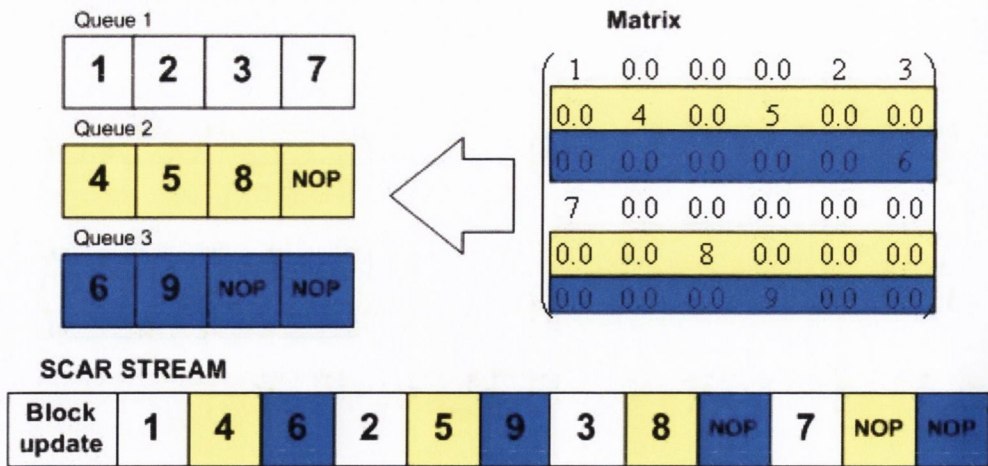


Figure 5.19 Simple SCAR reordering (using round robin system between the three queues)

RAW hazards are avoided since accesses to each row are always adder latency clock cycles apart. This reordering is easily done in software, which proved very efficient when working with Finite Element matrices. The very sparse Internet adjacency

matrices resulted in large amounts of NOPs being added to the data stream. To increase the performance of the SCAR, a new reordering scheme was devised called opportunistic reordering.

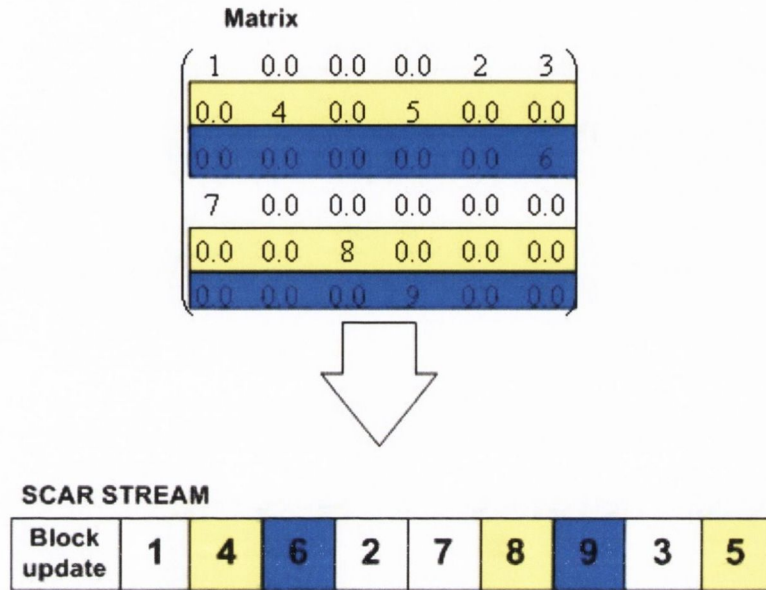


Figure 5.20 Opportunistic reordering in SCAR

Figure 5.20 shows a simple example of how opportunistic reordering works. In this reordering scheme, data is added to the SCAR stream opportunistically so that all data remains at minimum one adder latency away from the last reference to that row. This method loosens the constraint used in the simple reordering, where consecutive row entries must be exactly the adder latency apart. A better load balancing is thereby allowed for between all pipeline positions of the adder, reducing the number of NOPs. In the example in Figure 5.20, the 3 NOPs do not have to be inserted into the stream that would otherwise be needed with the simple reordering (Figure 5.19). The disadvantage of using this scheme is that it takes longer to implement than the simple reordering. The effect on performance of the two ordering schemes will be presented in the results chapter in Section 6.5.3.

5.5.2 SCAR Hardware

Figure 5.21 shows the high level view of the SCAR architecture [115]. Details of the state machine, vector buffers and memory interfaces are removed for simplicity. The SCAR system is composed of 4 main components; the controlling state machine, the X-buffer, the Y-buffer and the arithmetic path. The controlling state machine

coordinates the operations in hardware. It is controlled using embedded codes in the input stream encoding, see section 5.5.1. The Y-buffer is a dual port buffer that stores the fragment of the Y vector currently being calculated. The X-buffer is a cache for the X vector. At the start of each new block, the X-buffer is filled with all the X vector values that will be needed for that block.

The arithmetic data path consists of a multiplier and adder. In this implementation double precision floating point arithmetic units are used. Each matrix entry is multiplied by the appropriate X vector value from the X-buffer and the result is added to the appropriate Y value in the Y- buffer. Equation 15 shows the calculation being carried out on every matrix entry (A_{ij}).

$$y'_i = y_i + A_{ij}x_j \quad (15)$$

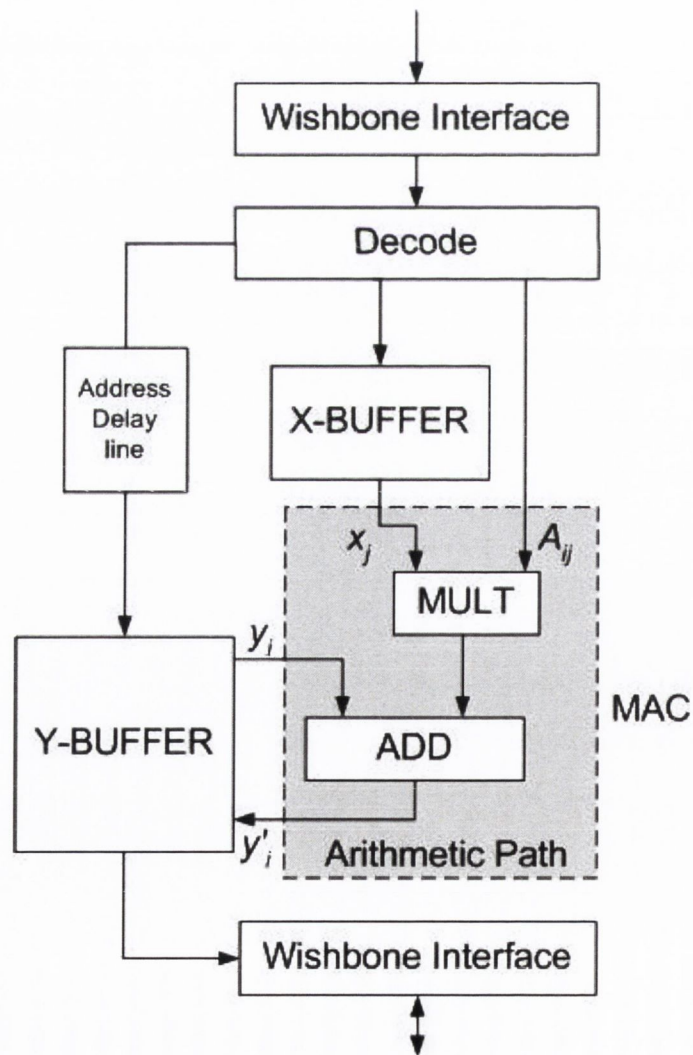


Figure 5.21 Outline of SCAR architecture

At the end of every row, the Y-buffer is flushed to main memory and zeroed, so that it can be used again for the next horizontal ribbon. The delay line is used to synchronise the reading and writing from the Y-buffer, if the multiplier and adder have a latency greater than one clock cycle.

The double precision floating point adders used in the SCAR have a latency of 12 cycles and so the possibility of a Read After Write (RAW) hazard occurring with the Y vector entries exists. Similarly to the SPAR architecture, this hazard happens when a value from the Y vector is accessed before the previous operation on that y value has been returned from the adder pipeline. An incorrect value would then be added to the adder pipeline, causing an incorrect result. No hardware exists to detect such hazards in the SCAR architecture. Thus to avoid this hazard, the reordering software must ensure that no value is accessed until the previous operation on that value has been returned. This reordering can be accomplished with relative ease as described in section 5.5.1. This process inserts NOPs into the SCAR stream if no NZE is found that will not cause a RAW hazard. Unlike the SPAR architecture, the SCAR architecture never stalls in the middle of a block. Once started, a new matrix value or NOP is read on every clock cycle until the end of a block is reached. This simplifies the state machine at the expense of extra storage requirements, since the NOPS make the matrix stream longer (This will be discussed in the results chapter in Section 6.5.3).

5.5.3 Dual SCAR architecture

In Table 2.3 the clock rate required to utilise the full memory bandwidth was presented for an FPGA architecture streaming data from memory. The floating-point double precision SCAR requires a 96 bit data word every clock cycle. The SCAR system would therefore need a clock of 175 MHz to completely use the memory bandwidth provided by a single DDR-266 (Memory used in the development board). In section 4.4.2 a number of floating point arithmetic units implemented on Virtex-II are presented in Table 4.3 and Table 4.4. None of the arithmetic units implemented on Virtex-II could achieve a 175 MHz clock rate. Therefore, increasing the clock rate to use the full memory bandwidth available was not possible. Instead parallelisation was used and thus a second implementation of the SCAR architecture was created with two arithmetic paths, see Figure 5.22.

This system operates in much the same way as the single arithmetic path SCAR system. The Y-buffer is split in two with odd addresses stored in one of the buffers and even address stored in the other. The X-buffer is replicated to allow for multiple accesses simultaneously. The data is streamed in from memory as before. If the address is odd, it is sent to the arithmetic path connected to the Y-buffer full of odd address values and otherwise it is sent to the second arithmetic path.

This system can operate on two SCAR data words simultaneously in every clock cycle. This is twice the data used by the single SCAR unit. By using twice as much data the dual SCAR only needs to go at half the clock rate the single SCAR must operate at to fully utilise memory bandwidth. In the case of double precision floating-point arithmetic this means the dual SCAR unit must be clocked at 87.5 MHz to utilise the entire memory bandwidth provided by the DDR 266. However, since there are now two adder pipelines there is a greater chance of RAW hazards occurring. Using this system effectively doubles the adder latency, thus more NOPs need to be added to the stream. The number of NOPs depends on the matrix structure and the number of NZE.

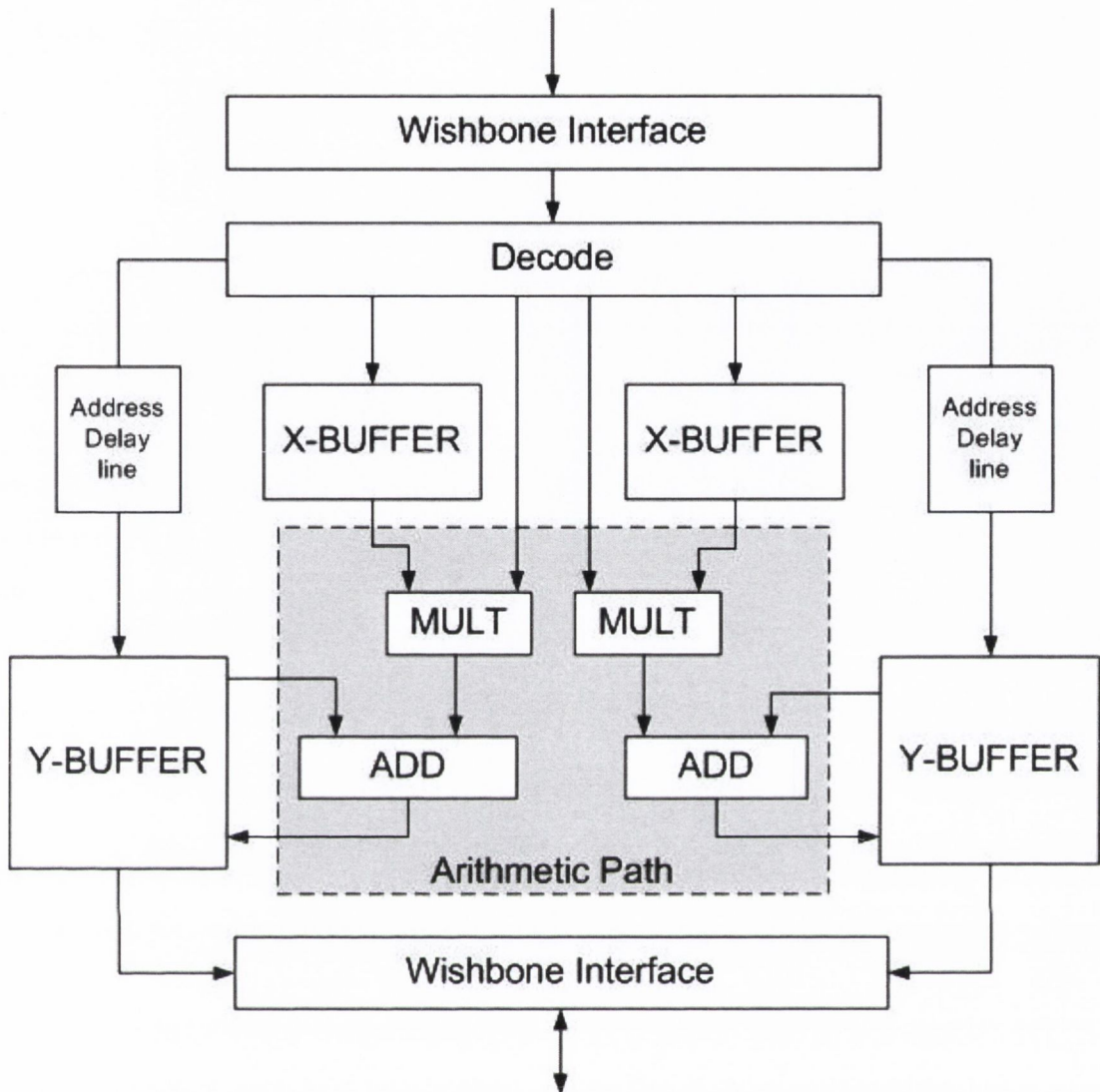


Figure 5.22 Block diagram of Dual SCAR architecture

5.6 SMVM Architecture 3 – PageRank SMVM

The third and final SMVM architecture presented is a special hardware architecture that was developed specifically for the PageRank algorithm as part of this project. It is based on the SCAR architecture (Section 5.5). However a number of changes have been implemented to take full advantage of the unique nature of the PageRank link matrix. In Section 3.5.5 the PageRank link matrix was described. This matrix is always column normalised. This allows the SMVM to be removed from the calculation. It is replaced with a single dense vector operation and a number of additions controlled by a pattern matrix. In this section the equivalency of these

operations to SMVM is discussed and the new hardware needed to solve such a system is presented.

5.6.1 Removing SMVM from PageRank

SMVM of the column normalised PageRank adjacency matrix and the current estimation of the PageRank vector is a central operation in the solving of the PageRank vector using the power method (see Section 3.5.5). Equation 16 states the mathematical expression for the power method which is an SMVM operation. It shows that the new value of PageRank for a page i ($R_k(i)$) is the sum of NZE in row i (A_{ij}) multiplied by the corresponding value of PageRank from the previous iteration ($R_{k-1}(j)$). Figure 5.23 shows a simple example of SMVM in the PageRank algorithm.

$$R_k = HR_{k-1}$$

$$R_k(i) = \sum_{j=1}^n A_{ij} R_{k-1}(j) \quad (16)$$

The H matrix is a sparse matrix and so is stored using a compression scheme. In the SCAR system the NZE of the matrix are stored in a 96 bit word as discussed in section 5.5. This 96 bit word contains the 64 bit NZ value and the other 32 bits are used to store the NZE position in the matrix and the control word. Therefore, the matrix requires $NNZ \cdot 96$ bits of storage space where, NNZ is the number of non-zeros in the matrix.

$$R_k = HR_{k-1}$$

$$\begin{pmatrix} 3/8 \\ 2/8 \\ 2/8 \\ 1/8 \end{pmatrix} = \begin{pmatrix} 0.0 & 1 & 1/2 & 0.0 \\ 1/2 & 0.0 & 0.0 & 1/2 \\ 0 & 0.0 & 1/2 & 1/2 \\ 1/2 & 0.0 & 0.0 & 0.0 \end{pmatrix} \begin{pmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{pmatrix}$$

Figure 5.23 Example of PageRank SMVM

The example in Figure 5.23 would therefore need 7 x 96 bit words to represent it. However, since the PageRank adjacency matrix is column stochastic, all values in a column are equal. Therefore we can represent H with a vector C_v of the column

values and a pattern matrix of H called P (i.e. all the NZE in the matrix are replaced with ones).

$$Cv = (1/2 \quad 1 \quad 1/2 \quad 1/2)$$

$$P = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Figure 5.24 The new column value vector and pattern matrix

This can be expressed mathematically by equation 17 and equation 18. These two equations use the same convention as the PageRank algorithm, $|P_i|$ is the number of outbound links on page i .

$$Cv(i) = \begin{cases} 0 & \text{if } |p_i| = 0 \\ \frac{1}{|p_i|} & \text{if } |p_i| > 0 \end{cases} \quad (17)$$

$$P_{ij} = \frac{A_{ij}}{Cv(j)} \quad \text{for } \begin{matrix} i = 1..n \\ j = 1..n \end{matrix} \quad (18)$$

This system of storing the matrix reduces the number of bits needed to store the matrix by a factor of 3. The matrix P no longer needs to store NZE values since all NZEs are one and so every NZE in the matrix P can be represented just using its position in the matrix which using the SCAR compression scheme takes just 32 bits. An additional vector Cv is created which is $n \times 64$ bits. However since IA matrices have an average of 10 NZE in every column the Cv vector and P matrix combined require less storage space than the H matrix.

The mathematical equivalent of an SMVM of the H matrix and the R vector can be calculated if an element by element vector multiplication of the Cv and the previous R vector is calculated. The result of this calculation is used in the SMVM multiplication with the pattern matrix to give the next estimation for PR, see Figure 5.25.

$$Cv \cdot R_{k-1} = T$$

$$\begin{pmatrix} 1/2 \\ 1 \\ 1/2 \\ 1/2 \end{pmatrix} \begin{pmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{pmatrix} = \begin{pmatrix} 1/8 \\ 1/4 \\ 1/8 \\ 1/8 \end{pmatrix}$$

$$R_k = PT$$

$$\begin{pmatrix} 3/8 \\ 2/8 \\ 2/8 \\ 1/8 \end{pmatrix} = \begin{pmatrix} 0.0 & 1 & 1 & 0.0 \\ 1 & 0.0 & 0.0 & 1 \\ 0 & 0.0 & 1 & 1 \\ 1 & 0.0 & 0.0 & 0.0 \end{pmatrix} \begin{pmatrix} 1/8 \\ 1/4 \\ 1/8 \\ 1/8 \end{pmatrix}$$

Figure 5.25 Replacing SMVM in PageRank

The results of this modified algorithm are the same as that produced originally in Figure 5.23. This can be confirmed mathematically by:

$$R_k = PT$$

$$R_k(i) = \sum_{j=1}^n P_{ij} T(j) \quad (19)$$

Substituting for T into equation 19, we get:

$$R_k(i) = \sum_{j=1}^n P_{ij} Cv(j) R_{k-1}(j) \quad (20)$$

However rewriting equation 18 we can express A_{ij} in terms of Cv and R_{k-1} .

$$A_{ij} = P_{ij} Cv(j)$$

$$\therefore R_k(i) = \sum_{j=1}^n A_{ij} R_{k-1}(j) \quad (21)$$

Equation 21 is the same as Equation 16 and so the two methods are equivalent. The SMVM used in Figure 5.25 between the P pattern matrix and the vector T does not have to be a SMVM. All the values of the P matrix are 1 and so the multiplier can be removed from the SMVM unit and thus the SMVM unit becomes a pattern adder.

The inputs of the adder are dictated by the pattern matrix. There is no longer a multiplication needed in the unit and so it is no longer an SMVM unit.

5.6.2 PageRank Hardware

The SMVM operation is broken up into two operations, an element by element vector operation and a sparse pattern addition as described above. The existing Vector Unit is capable of calculating element by element vector multiplications so no new hardware is needed to compute that operation (see Table 5.1). The SMVM unit could be used to perform the pattern addition. The pattern matrix could be multiplied by 1 and then added to their respective Y value. However, this would not take advantage of the memory bandwidth that was freed up by removing the need to stream NZE values. The pattern adder can work just using the addresses of the NZE since all the entries are one. This allows for compression in the data word. In the SCAR data stream a NZE data word contained a 64 bit NZE value and the remaining 32 bits contain the address and control data. This 64 bit NZE value is redundant and can be replaced with more NZE address information. This effectively increases bandwidth since multiple NZE are being streamed in on every clock cycle. In this solution 2 other NZE addresses are encoded in the data word see Figure 5.26.

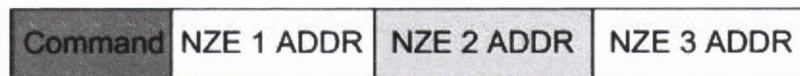


Figure 5.26 Pattern adder data word

The reordering used to create the pattern adder data word is very like the arbitrary reordering used in the SCAR system. Like the SCAR system, the matrix is divided up into blocks. Each of these blocks is then encoded into a SCAR stream. However, the top three NZE from the next available row are taken, see Figure 5.27.

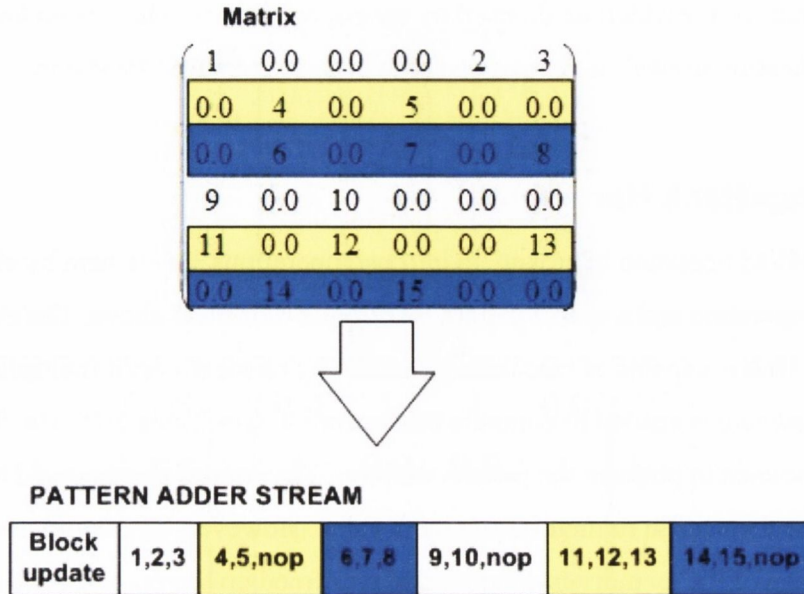


Figure 5.27 Encoding the Pattern adder stream

The three NZE addresses in the pattern adder data word are from the same row in the matrix and thus affect the same Y-buffer value. 3 reads to the Y-buffer would be required if the NZE in the pattern adder data word came from different rows. Taking them from the same row keeps the clock rate of the Y-buffer at a reasonable rate and ensures that the hardware and reordering remain relatively simple. A special NOP command can be used if a row contains less than three values. To use this new encoding system specialised hardware was developed. Figure 5.28 shows a block diagram of the system.

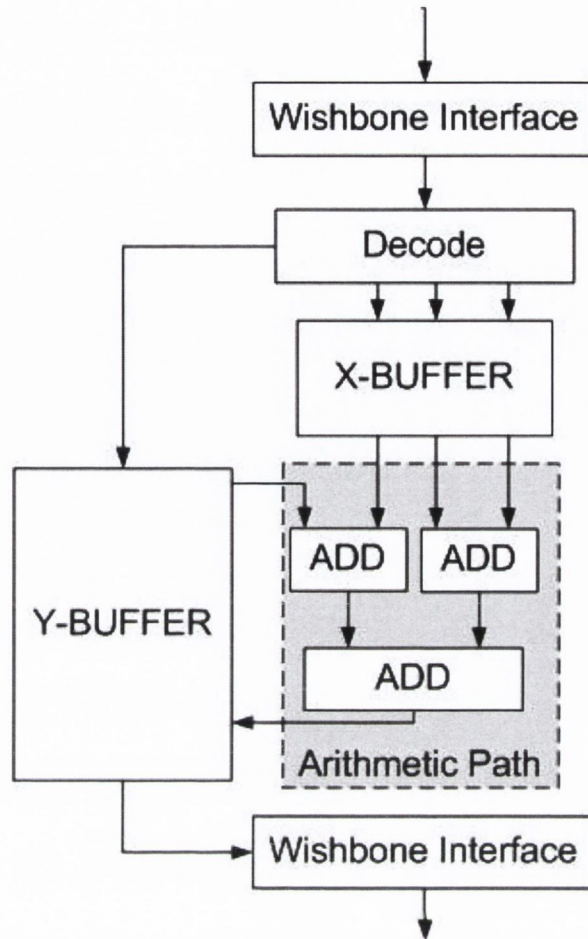


Figure 5.28 Block diagram of Pattern adder

The hardware for the pattern adder is relatively simple. It operates much like the SCAR hardware described in section 5.5. The pattern adder data is streamed from memory through the Wishbone interface. The decode unit breaks the data word up into a Y-buffer address and up to 3 X-buffer addresses. The Y-buffer contains the partial y-sums for all the rows in the block. The X-buffer contains a fragment of the vector obtained from the element by element multiplication of the previous PageRank vector and the matrix column value vector. The X-buffer is queried for three values. One of them is summed with the value returned from the Y-buffer. The other two X-buffer values are summed together. The results of the two additions are then added together with a third adder and the result is written back to the Y-buffer. Since all the NZE values come from the same row of the matrix they all affect the same Y value and thus the reason they can all be summed together. When a block is complete the X-buffer fetches the next section of the element by element multiplication result. If the block was the last in a matrix row then the data in the Y-

buffer is written out to memory. That section of the Y value is complete. The Y -buffer is then zeroed for the next set of blocks.

The X -buffer in this system must return 3 values every clock cycle. In order to achieve this using the FPGA's on board BRAM the X -buffer is internally duplicated as in Figure 5.29. The X -buffer contains 2 dual port rams. The read only port address is connected to the NZE address coming on the data word. This provides two of the three X values needed. The read/write port is used to populate the dual port ram when a block update command is received. However, the Read/write port on one of the rams is also used to read the third X value when the system needs three X values. The R/W signal is set to write if the unit is performing a block update and the R/W signal is set to read if the input data is a NZE data word. A MUX unit decides if the read or write address is needed. The read address comes from the decoded data packet and the write address comes from the block update state machine.

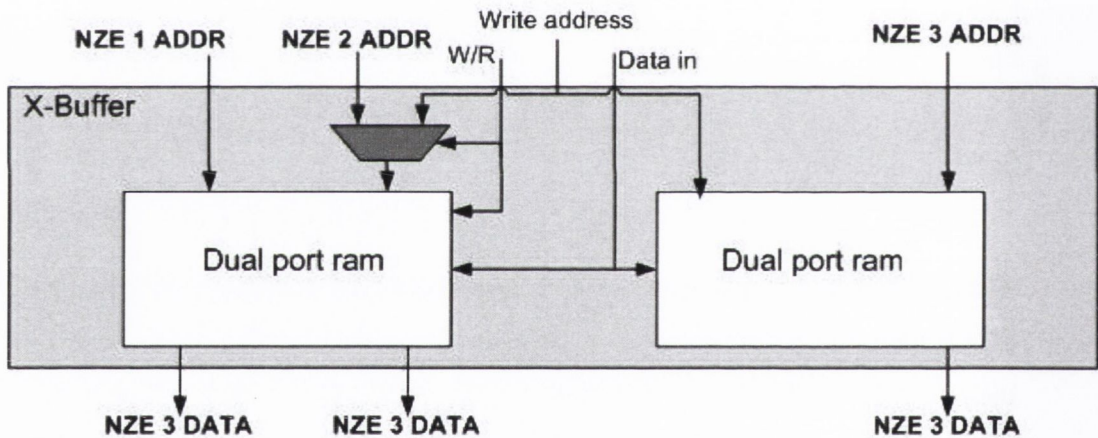


Figure 5.29 Internal Structure of X-buffer

This hardware architecture contains an adder tree. If a value from the Y -buffer is in the adder pipelines it cannot be accessed again, as this would cause a RAW hazard. Long adder latencies increase the likelihood of these errors occurring. The pattern adder must use a shorter latency adder than the SCAR unit because data has to be processed by two adders before being written back to the Y -buffer. Details of the adders used are shown in chapter 6.

5.6.3 Dual path solution

Like the SCAR architecture, the pattern adder hardware can be built with a dual arithmetic path. Doing this effectively halves the clock rate at which the unit must be clocked at to use the memory bandwidth. However, this effectively doubles the adder-tree latency, increasing the number of potential RAW hazards in the stream and thus increasing the NOPs in the stream. A block diagram of the dual path pattern adder is shown in Figure 5.30. The Y-buffer is split into two parts. The even addresses are processed by one arithmetic path and the other arithmetic path processes the odd addresses. The X-buffer must be duplicated to allow access from both arithmetic paths. Both arithmetic paths work on the same block of data but spread the work between the two adder trees. This parallelisation in the calculation effectively doubles the clock rate of the system.

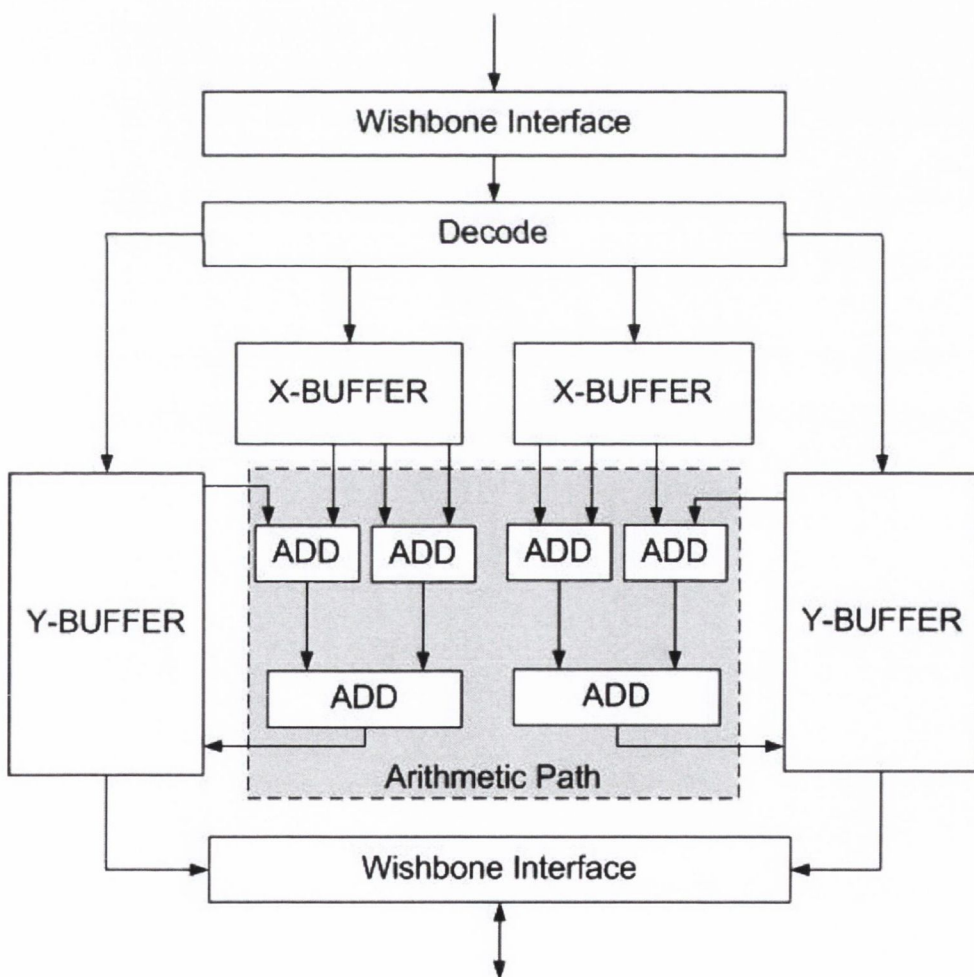


Figure 5.30 Dual Path Pattern adder

The Wishbone bus can deliver two data words per clock cycle. The decoder then splits the calculation over the two arithmetic paths. Since one arithmetic path is used for even buffer address and the other arithmetic path is used for odd addresses there is no contention between the two Y-buffers. Both Y-buffers operate independent of each other. There are two X-buffers in the system but they are identical. Two X-buffers are needed to allow 6 simultaneous queries to the X-buffer. This system can process 6 matrix NZE per clock cycle. At the end of a row of blocks the Y-Buffer is flushed. Since odd address Y values are in one Y-buffer and even address Y values are in the other Y-buffer the Y buffers must be flushed simultaneously, interweaving every second value to ensure a correct Y-vector. In practice this is easy since the Wishbone bus used for this calculation is 128 bits wide, a single value from each Y-buffer can be streamed to memory together.

5.7 Bus Contention on SMVM architectures

Bus contention may arise when two or more units are connected to the same bus and must share the bus resources. This can leave units waiting to use the bus if another unit is already using the bus. Thus, bus contention reduces the performance. The FPGA system described in this chapter contains four Wishbone buses. Three of these buses are used for matrix data in the SMVM units and are not shared. The fourth bus is a shared vector bus and so is liable to bus contention. During the SMVM calculation using a single SMVM unit there is only one situation that could lead to bus contention. In the single SMVM system the X and Y vectors are stored in the memory connected to the shared vector bus. This means that the X and Y vectors in main memory cannot be accessed simultaneously. In the SPAR unit this is an issue any time the Y-cache and X-cache miss at the same time. In the SCAR and PageRank system this only becomes a problem at the end of every stripe of blocks when the Y-vector is written back to main memory. Ideally the X vector for the next stripe could be read in while the Y-buffer is flushing. This does reduce performance but not to a great extent and definitely does not justify the inclusion of another memory channel.

The real problem with bus contention becomes evident when multiple SMVM units are connected to the same shared vector bus. All the SMVM units must now contend

for the bus every time a new X-vector is to be read in or Y-vector needs to be written out. If there are sufficient NZE in each block of the SCAR and PageRank systems this should not be a problem. In previous tests on FEA, the shared vector bus showed very little contention issues when operating with 3 SCAR units. However, bus contention on the shared vector bus is a limiting factor on the number of SMVM units that can be added to FPGA architecture. Internet adjacency matrices are sparser than FE matrices and so bus contention may well be an issue with multiple SMVM units calculating the PageRank algorithm.

5.8 Software

The software for the FPGA base PageRank system can be divided up into three levels, software running on the host PC, the software kernels available on the FPGA and the software running on the Microblaze processor. Each level of software plays its own vital part in the PageRank calculation on FPGA

5.8.1 Software on Host PC

The software running on the host PC is responsible for preparing the data and transferring it to the memory banks on the FPGA daughter board. It is mostly coded in C, in the form of mex files running in Matlab. The host PC software takes the raw adjacency matrix and encodes it into the required format (SPAR or SCAR) format. The matrix is written to the banks of DDR memory on the development board via the PCI bus. The initial estimate of the PageRank vector is also written down to the card along with the dangling node vector and some control data. Once the data is on the card the PC host software interrupts the Microblaze and starts the calculation. The Microblaze issues an interrupt to the host PC when it has finished the calculation. The software then downloads the result to the host PC. The result can then be used or checked for correctness.

5.8.2 Hardware Communication kernels

To facilitate easy reprogramming of the Microblaze a number of low level kernels were written. A hardware communication kernel was written for every hardware unit

available in the system. This code allows a user to use high level C function calls to control hardware units. The user can simply call one of these kernels giving it the addresses of the vectors and matrices being operated upon and the kernel will set up the hardware for the desired calculation. The kernel then starts the hardware. When the hardware calculation has completed the hardware unit interrupts the MB. The kernel then reads the result from the hardware unit and returns it to the user in the form of a variable or address. This allows users to use the system with little or no knowledge of the underlying architecture. Table 5.3 shows the range of kernels available to a user in current system.

Table 5.3 Table of Hardware communication kernel functions

name	function	name	function
smvm	sparse matrix by vector	elem	vector element by element multiply
memcpy	memory copy	sadd	scalar add
smult	scalar multiply	ssub	scalar subtract
vscale	Vector scale	dot	dot product
axpy	vector scale and vector addition	axmy	vector scale and vector subtraction
wxpy	vector element multiply and vector addition	wxmy	vector element multiply and vector subtraction
norm	norm squared	zero	zero memory

5.8.3 Programming the MicroBlaze

The MB processor is used to control the whole system. To implement a particular problem on the system a simple C program needs to be written to run on the Microblaze. This C program will set up all the hardware to carry out the operations needed to calculate the algorithm result. The hardware communication kernels make programming the Microblaze easy. To calculate the dot product of two vectors a user only needs to call the dot product kernel and pass it the two vectors to operate upon. The result will be returned to the Microblaze and stored in one of its local registers for use later on in the algorithm. The C code for the MB running the PageRank algorithm is shown in Figure 5.31. The kernels make the code easy to understand and edit. Any algorithm that uses the functions in Table 5.3 can be programmed quickly and easily on the Microblaze.

```

1 do.
2 {
3 //stochastic and primitivity adjustment.
4 sto_adj = dot (R,dnv);.
5 beta = smult (sto_adj,alpha);.
6 beta = sadd((1-alpha),beta);.
7 adj = vscale(beta,(e/n));.
8 // Sparse matrix by Vector multiplication.
9 prod=smvm(H,R);.
10 // alpha scale and adjust.
11 R_n=saxpy(alpha,prod,adj);.
12 iter++;.
13 check_converge (done,R,R_n);.
14 }.
15 while ( (done ==0) && (iter<=max_iterations));.

```

Figure 5.31 The C code for the PageRank algorithm run on the MicroBlaze

Once the code has been developed it can be compiled and downloaded into the Microblaze program memory via the JTAG cable. This means that no rebuilding of the system is required to change the algorithm running on it.

5.9 Summary

In this section, hardware design and implementation has been discussed. The PageRank solver system was presented. The system comprising of four Wishbone buses that connect via four memory controllers to four banks of DDR. A number of computation units are also connected to the Wishbone buses. A very versatile Vector Unit allows many different vector and scalar operations to be computed on the board. In addition to the Vector Unit, three SMVM units are present in the system.

The PageRank calculation is dominated by a large SMVM calculation. Three different SMVM architectures were presented. The SPAR architecture was designed by Taylor and is a column based architecture. It has been slightly modified to maximise its performance on large matrices being streamed from slow DDR memory. This was done by splitting the matrix up into stripes that fit in the Y-cache and computing the solution for the partial matrix before moving on, thus avoiding lengthy Y-cache stalls. The SCAR architecture is a row based architecture. The matrix is split up into blocks. The X vector associated with the block is read into the X-buffer and the Y buffer is set to zero. The SCAR unit processes the row and thus computes the Y values for the SMVM. At the end of every row the Y value is written

out to memory. The SCAR system only uses X and Y buffers and does not need complicated caches like the SPAR architecture. The NZE are reordered to ensure no RAW hazards occur. If there is not enough NZE entries to avoid RAW hazards the stream is padded with NOPS. The SCAR units therefore do not need to stall, which was an issue that caused problems in the SPAR architecture. The final SMVM architecture takes advantage of the unique attributes of the PageRank adjacency matrix. Since all values in a column are equal, the matrix can be replaced with a pattern matrix and dense vector of column values. The PageRank algorithm can then be reordered to remove the SMVM calculation altogether. It is replaced with a dense vector operation and a series of additions controlled by the pattern matrix. This leads to a decrease in storage requirements and an increase in memory bandwidth. The pattern addition is done by a tree of adders. This scheme has an advantage over SCAR and SPAR because the NZE no longer need to be streamed into the architecture which effectively triples the memory bandwidth.

The system is controlled by a MicroBlaze processor. A number of software kernels have been specially designed to make programming algorithms on the Microblaze easy and efficient. The system can be programmed using a simple C program running on the MicroBlaze using a set of hardware communication libraries. This allows for great flexibility of the system. It can be programmed to calculate many different matrix calculations by adding a software function to the Microblaze software library.

Chapter Six

6 Results

6.1 Introduction

This chapter presents the results of benchmarks carried out on the architecture for computing the PageRank algorithm as described in Section 5. These benchmarks are compared against the performance of a state of the art GPP.

The results presented in this chapter are divided into four main groups. These groups are floating point performance on Virtex-II, fixed point performance on Virtex-II, floating point performance on Virtex-5 and fixed point performance on Virtex-5. The results of all benchmarks for Virtex-II systems are fully implemented designs running on the Virtex-II XCV6000 FPGA. However, as discussed earlier the Virtex-5 results are extrapolated from the Virtex-II results and the post place and route clock information for the system targeted at the Virtex-5 155LX FPGA. The performance is measured in Operations per Second (OPS) for fixed point implementations and Floating Point Operations per Second (FLOPS) for floating point designs. The OPS or FLOPS are calculated by dividing the number of arithmetic operations carried out by the execution time; see equation 22.

$$(FL)OPS = \frac{\text{number of arithmetic operations}}{\text{time}} \quad (22)$$

The host PC reads the system time immediately before and just after the FPGA calculates the PageRank vector. The elapsed time is the execution time of the PageRank algorithm and is used to work out the (FL)OPS. The number of arithmetic operations carried out is calculated from an analysis of the PageRank algorithm. The PageRank algorithm requires one SMVM, five vector operations and two scalar

operations per iteration. The vector operations all require $2*N$ operations, except for the vector-scale operation which only requires N operations. The SMVM requires $2*NNZ$ operations. The total number of operations to calculate the PageRank vector is given by equation (23).

$$operations = (2NNZ + 9N + 2) * iterations \quad (23)$$

The benchmarks were carried out using actual IA matrices. These matrices are presented in the next section.

It is hoped that these results and comparisons with a state of the art GPP will provide insight into the viability of using FPGAs to tackle a full scale Internet ranking problem.

6.2 Benchmark Matrices

The matrices used to benchmark the GPP and the FPGA based hardware are real IA matrices. They were created from web crawls carried out for the WebBase and WebGraph projects [116, 117]. These crawls contained as many as 170 million links. Matrices of this size would not fit in the SDRAM on the development boards and so for this reason, only subsections of the crawls were used. The FPGA hardware itself does not limit the size of the matrix other than that it must fit in the FPGA board's memory.

The Alpha Data development platform used in these experiments was populated with four 256MB DDR SDRAM. Each SDRAM module could hold up to 22 million 96-bit words (the size needed to represent a NZE in SCAR). To increase the matrix size further, the memory could be replaced with larger SDRAM modules. The actual test matrices used in the benchmarks are outlined in Table 6.1 which shows clearly that none of the matrices will fill the SDRAM. The matrices were intentionally made smaller than necessary in order to allow for NOPs to be added to the stream, as well as to leave space for the other vectors needed by the PageRank algorithm to be stored. The IA matrices used have an average of 14 links per node. Langville states that the average node on the Internet contains 10 links [41]. Since the benchmark matrices link count is of this order, the test matrices should correctly represent the average IA matrix.

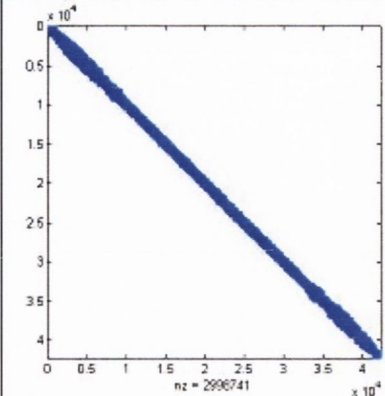
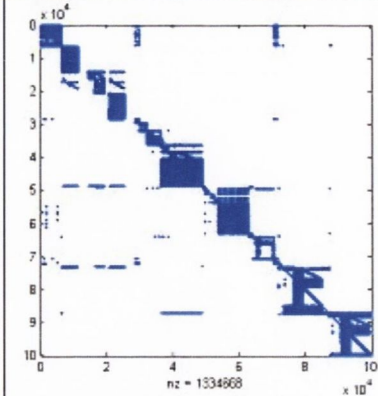
Table 6.1 Details of Benchmark matrices

Num.	Name	Nodes	No. of Links	Source
1	Arabic-2005	500000	9877485	WebGraph
2	cnr-2000	325557	3216152	WebGraph
3	eu-2005	500000	11615380	WebGraph
4	in-2004	500000	5422294	WebGraph
5	Indochina-2004	500000	8147699	WebGraph
6	it-2004	500000	9387335	WebGraph
7	sk-2005	400000	13391888	WebGraph
8	uk-2002	500000	6998368	WebGraph
9	uk-2005	500000	11192060	WebGraph
10	web-mat-1.5M	1500000	12392081	WebBase
11	web-mat-1M	1000000	8686242	WebBase
12	web-Stanford	281903	2312497	WebBase
13	webbase-2001	500000	4214705	WebGraph

In a number of the tests run as part of this work, the results of the IA matrices are compared against results achieved by FE matrices. The details of the FE matrices used are in Appendix One. A graphical representation of the IA matrices in Table 6.1 is also presented for reference.

Table 6.2 outlines the similarity and differences of FE matrices and IA matrices. The major difference between the two matrices is scale. IA matrices are over a thousand times bigger than FE matrices. The size or sparseness of the IA matrices should not in itself have a significantly negative effect on the performance of SMVM. The IA matrices are only loosely banded, however, which will reduce performance due to the lack of locality of reference. This poor locality of reference will, thus, cause increased cache misses in the GPP. Furthermore, it will also affect custom hardware performance, as it will increase the number of blocks with NZEs and thus reduce performance. This loose banding can be seen in the matrix diagrams in Table 6.2. The FE matrix consists of a single tight band along the diagonal of the matrix while the Internet adjacency matrix has many NZE away from the diagonal. The IA matrix is still somewhat banded around the diagonal, which is due to the URL based reordering scheme used to order the matrix (see section 4.2.2).

Table 6.2 Finite Element and Internet Adjacency matrices compared

Finite Element Matrix.	Internet Adjacency matrix
	
Very Sparse (0.02 – 1.1%)	Very Sparse (< 0.01%)
Often Symmetric	Never Symmetric
Millions of rows	Billions of rows
11-72 NZE/columns	4-14 NZE / column
Usually tightly banded	Not tightly banded

6.3 General Purpose Processor results

Obtaining an accurate performance measurement of state-of-the-art technology is crucial when testing new hardware's viability at performing a given task. Currently, the 3GHz Intel Xeon (Woodcrest) processor is a top-of-the-range General-Purpose Processor (GPP) and so is a good benchmark against which to compare any new computational hardware. The Woodcrest is a core-two processor, meaning it has two processing cores internally as shown earlier in Figure 2.7. The Woodcrest processor is capable of computing four floating-point operations per clock cycle [118]. At 3 GHz, the peak performance reached is 12 GFLOPs. However, this level of performance is only sustainable under certain circumstances (e.g. high level of data reuse from L1 cache). This level of performance is not achievable for the PageRank algorithm because of the size of the matrix and vector. The PageRank calculation is too big to fit in cache and so requests to memory via the Front Side Bus (FSB) must be continuously made. The calculation is therefore limited by the FSB speed. The Woodcrest FSB has a theoretical peak memory bandwidth of 10.6 GB/s. If every NZE is represented by 96 bits (12 bytes) and requests to memory for X and Y values

are assumed to be relatively small, the number of NZE transferred across the FSB per second is given by (24).

$$\text{NZE/s} = \frac{\text{FSB BW}}{\text{NZE size}} = \frac{10.6\text{GB}}{12B} = 883\text{MWords/s} \quad (24)$$

Since two floating point operations are carried out per NZE, the maximum performance achievable by the Woodcrest is about 1.76GFLOPs. In reality, the Woodcrest can only achieve about half of this performance, due to X and Y vector fetches and memory controller cycles. Figure 6.1 shows the performance achieved by the GPP for the IA matrices. These tests were done using multi-threaded C code based on the benchmark code used by Sweeney [65] and modified for PageRank. The code was compiled using the Intel compiler. Two versions of the SMVM code were implemented one was a column major SMVM routine and the other was a row major SMVM routine. The highest performance value from these two tests is quoted below. Williams et al. [37] attempted to optimise SMVM routines on the Woodcrest. One of his benchmarking matrices, matrix 13 was also used in this work. The results achieved in this work for matrix 13 are the same as the results obtained by Williams et al [37]. Therefore Williams' results act as an independent verification as to the quality and accuracy of the benchmarking results presented in this thesis.

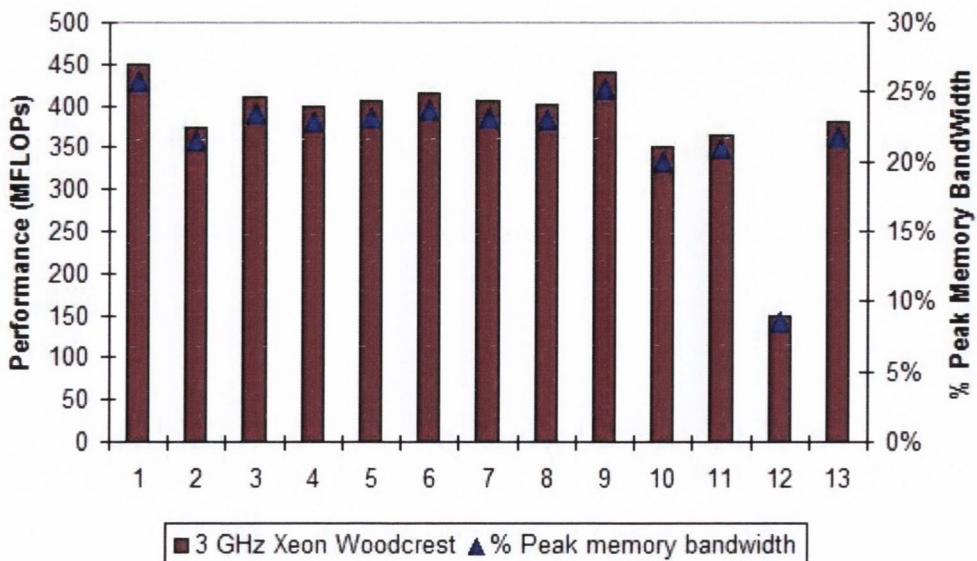


Figure 6.1 General Purpose Processor Benchmark results

The GPP achieves approximately 400 MFLOPs for all matrices except number 12 for which it only achieves 150 MFLOPs. Matrix number 12 is the IA matrix retrieved

from a crawl of the Stanford domain in the WebBase project [116]. This matrix is very sparse with only 4 links per page, which is not standard for an IA matrix. The Stanford matrix does not appear to be reordered like all the other matrices used. The crawler usually reorders matrices based on their URLs as they are produced to reduce computation time [75]. The Stanford crawler does not appear, however, to have reordered the nodes of the matrix to increase performance.

Overall, the Woodcrest achieves between 3% and 4% of its theoretical computational peak due to memory bottlenecks and limited parallelisation; see Figure 6.2. The Woodcrest relies on heavy reuse of its cache to achieve peak performance. However, IA matrices have no data reuse and so the Woodcrest is forced to fetch new data from memory repeatedly. Since the Woodcrest only has one memory channel for both processors to share (see 2.4.1), the speed of the FSB becomes critical to the performance of the system. Even when the limiting FSB is taken into account, the Woodcrest only achieves 25% of its peak performance across the FSB. The results achieved from the GPP show that even the high performance cache structure is of little use with the PageRank algorithm, due to its lack of data reuse. Another approach needs to be taken. The FPGA with its large number of I/O pins gives the opportunity to increase memory bandwidth by parallelisation and thus to improve the performance of the PageRank ranking algorithm.

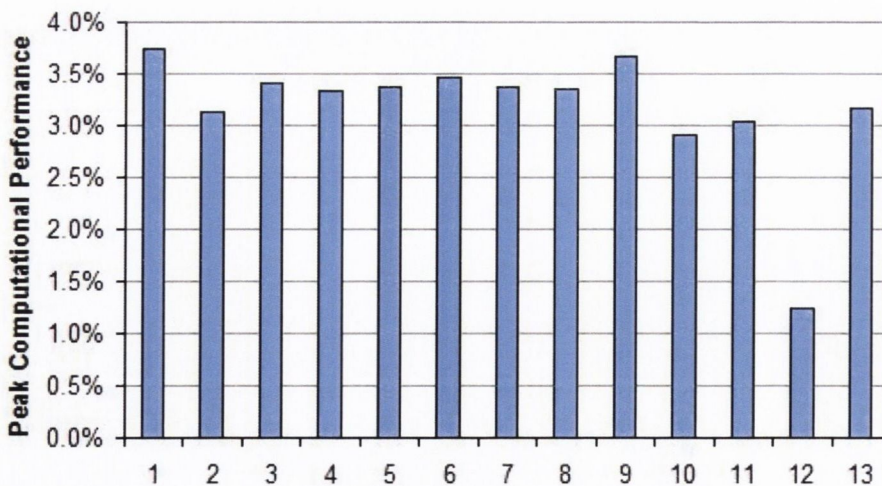


Figure 6.2 GPP % Peak Computational Performance.

6.4 Floating Point PageRank Performance on Virtex-II

The first of the FPGA architecture results to be presented are the floating point PageRank performance benchmarks on a Xilinx Virtex-II XCV6000 FPGA. The system described in chapter 5 was built and implemented on this device. The system was implemented with the three different SMVM architectures discussed in sections 5.4, 5.5 and 5.6. The results of these benchmarks are presented here. All architectures were implemented with double precision floating point multipliers and adders. The Virtex-II FPGA supports DDR type commodity memory. The memory used in these tests is DDR-266. Each bank of DDR-266 has a maximum memory bandwidth of 2128 MB/s, as discussed in section 2.4. The peak performance per memory channel is 355 double precision MFLOPs assuming that a NZE can be represented by a 96 bit word and that to calculate a SMVM each NZE undergoes two floating-point operations. The Virtex-II systems have three memory channels which can be used for streaming the matrix and a fourth channel which is shared between the SMVM units to access the X and Y vectors. The system thus has a peak overall SMVM performance of 1.065 GFLOPS, if memory bandwidth is fully used. This maximum performance is not achievable due to the same memory refresh cycles and memory latency issues that affect the GPP performance. Bus contention on the shared X/Y vector bus is another issue which will be discussed later in section 6.5.1.

The computational bandwidth of the hardware does not always match the bandwidth available from memory. The overall system performance is limited by whichever of the computational or the memory bandwidth is the smaller. A number of factors affect the computational bandwidth for the PageRank algorithm. These factors are the number of vector operations (NV_Ops), the number of sparse matrix computations (NM_Ops), the peak performance of the SMVM unit (p_SMVM), number of SMVM units (m) and the peak performance of the Vector Unit (pV). The computational bandwidth is given by equation (25).

$$\text{Computation BW} = \frac{NM_Ops}{\text{Total Ops}} (p_SMVM * m) + \frac{NV_Ops}{\text{Total Ops}} (pV) \quad (25)$$

Equation (25) will be used throughout the chapter to calculate the peak achievable performance of the architectures. The three SMVM architectures were implemented and tested on Virtex-II FPGA running the PageRank algorithm. The results of these

tests are presented in the next few sections. This discussion starts with the SPAR architecture in section 6.4.1. A single and dual path version of the SCAR architecture is discussed in 6.4.2 and the specialised PageRank architecture is presented in 6.4.3. This section concludes with a comparison of the three architectures.

6.4.1 SMVM Architecture 1 – SPAR

The SPAR architecture is the first of the SMVM architectures to be presented. The SPAR architecture uses a 96 bit word to represent each NZE, as 64 bits contain the NZE value and the other 32 bits contain the NZE row address. A system containing three SPARs was implemented on the Virtex-II FPGA. The FPGA utilisation information is displayed in Table 6.3. The system has a clock speed of 100MHz with a 7 cycle double precision floating point adder. The system utilises 84% of the available slices.

Table 6.3 FPGA utilisation for floating point SPAR on Virtex-II

Logic name.	Utilisation (Available)	% used
Slice Flip-Flops	32470 (67584)	48%
4 input LUTs	42873 (67584)	63%
Multipliers	39 (144)	27%
Block RAM	103 (144)	71%
Gclk	8 (16)	50%
Occupied slices	28389 (33392)	84%

Each SPAR unit completes an addition and multiplication on every clock cycle, and so at 100MHz, each SPAR unit has a maximum performance of 200 MFLOPs. The system contains three SPAR units and so has a peak SMVM performance of 600 MFLOPs. The achievable peak performance of the PageRank algorithm running on this system was calculated using equation (25). It was found to be on average 200 MFLOPs and 500 MFLOPs for the systems with one and three SPARs respectively. The three SPAR system does not triple the computational bandwidth of the PageRank algorithm since the SMVM operator is only part of the calculation.

A large gap exists between the computational BW and the memory BW. The memory system can support up to 1.06 GFLOPs. The gap in BW is caused by the limited clock rate of the SPAR implementation, which can only be clocked at 100MHz. In large systems the inter-connects between logic cells get longer and thus

the clock rate decreases. The critical path in this design lay in the adder pipeline in the SPAR unit. A slightly higher clock rate could possibly be achieved if more floorplanning was implemented on the design or if a higher speed grade FPGA was used. In section 2.4.2 it was shown that an architecture using a 12B word every clock cycle would necessitate a clock rate of 175 MHz in order to fully utilise the memory bandwidth available from the DDR-266 SDRAM. The results achieved by the system are much lower than even the computational peak and are shown in Figure 6.3 together with GPP result for comparisons.

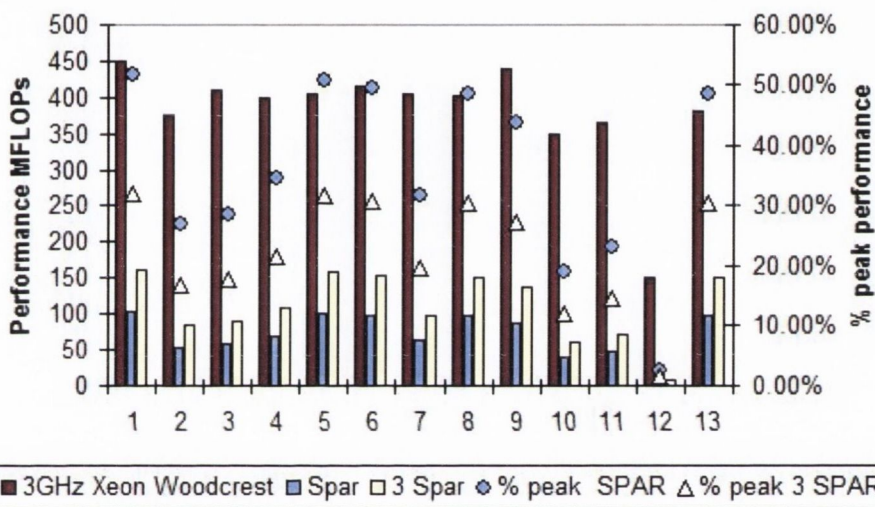


Figure 6.3 SPAR system Benchmark results Vs. GPP results. Primary axis: Performance in MFLOPs. Secondary axis: %peak computational performance (shown by circles for single SPAR and triangles for three SPAR system).

A single SPAR unit achieved an average of 70 MFLOPs when computing the PageRank vector on the test matrices. This result represents an average of about 35% adder utilisation, which is much better than the 3-4% computation BW utilisation achieved by the GPP. However, since the GPP clock rate is 30 times higher than the FPGA clock rate, it easily out-performs the FPGA architecture. The adder utilisation of the SPAR is 35%. Thus, the adder is unused 65% of the time, which can be contributed to RAW hazards, see Section 6.5.3.

The SPAR MAC unit is stalled for almost 50% of the calculation duration due to RAW hazards. RAW hazards arise when a number is requested from the cache that is currently being updated in the adder pipeline. The cache must wait for the adder to return the new value before servicing the cache request. Many clock cycles are

wasted waiting for the updated value to be written back to cache. The SPAR architecture is a column-based architecture, which means that it services NZEs one column at a time. No RAW hazards occur between NZEs in any given column. However, the short column lengths of IA matrices mean that there are not enough NZEs to flush the adder between successive columns and thus stalls due to RAW hazards are unavoidable in the SPAR architecture when using IA matrices.

The three-SPAR system achieved an average of 110 MFLOPs. Three SPAR units did not increase performance by three times the performance achieved by one SPAR unit due to RAW hazards prevention in the Y-cache and bus contention on the shared vector bus. This bus contention once again stems from the sparseness of IA matrices. The SPAR system must use this bus to read in the Y and X vectors. The Y vector is read at the beginning of each stripe of the matrix and written out to memory at the end of the matrix stripe. The X-buffer values are read in every 1024 column updates. In denser matrices, this fetching of the X vector does not have a major impact on performance. IA matrices are so sparse that the SPAR unit finishes columns quickly and thus needs to read in new X values quickly. Only one SPAR can access the shared vector bus at a time and so the other units have to wait for the first SPAR to finish before they can access the shared memory.

6.4.2 SMVM Architecture 2 – SCAR

The second system tested was the SCAR SMVM system. This system has a single arithmetic path with a 1024*DWORD (64 bits) X-buffer and 1024*DWORD Y-buffer. The adder used in the SCAR unit was a 12 cycle double precision floating point Xilinx adder. This adder was used to remove the critical path of the design out of the adder which was the case in the SPAR architecture. In the SCAR architecture the critical path was in the memory interface. The FPGA utilisation for a system containing three SCARs is shown in Table 6.4. The design takes up 79% of the FPGA-slices which makes it slightly smaller than the SPAR unit. The SCAR state machine is simpler than the SPAR state machine and this reduction in complexity is chiefly responsible for the reduction in overall system size. The SCAR architecture contains no RAW hazard detection logic, since this is done by the SCAR reordering software. (see section 5.5.1).

Table 6.4 FPGA utilisation for floating point SCAR on Virtex-II

Logic name.	Utilisation (Available)	% used
Slice Flip-Flops	32470 (67584)	48%
4 input LUTs	42873 (67584)	63%
Multipliers	60 (144)	41%
Block RAM	94 (144)	65%
Gclk	8 (16)	50%
Occupied slices	27032 (33392)	79%

The system can achieve a clock rate of 100 MHz, which gives each SCAR unit a peak performance of 200 MFLOPs. Therefore, using equation (25), the overall computational bandwidth of the system running the PageRank algorithm is on average 200 MFLOPs, 349 MFLOPs and 500 MFLOPs for a system with one, two and three SCAR units respectively. The computational bandwidth is well below the peak memory bandwidth of approximately 1 GFLOP. Some small improvements in clock rate and thus computational BW might be possible through floor planning and handcrafting the layout but these improvements would not be large enough to make them worthwhile. The clock rate of the SCAR system cannot be increased significantly so as to fully utilise the peak memory bandwidth. The performance results of the PageRank calculation running on the SCAR system are on the primary axis of Figure 6.4. The percentage of peak computational bandwidth is on the secondary axis of Figure 6.4.

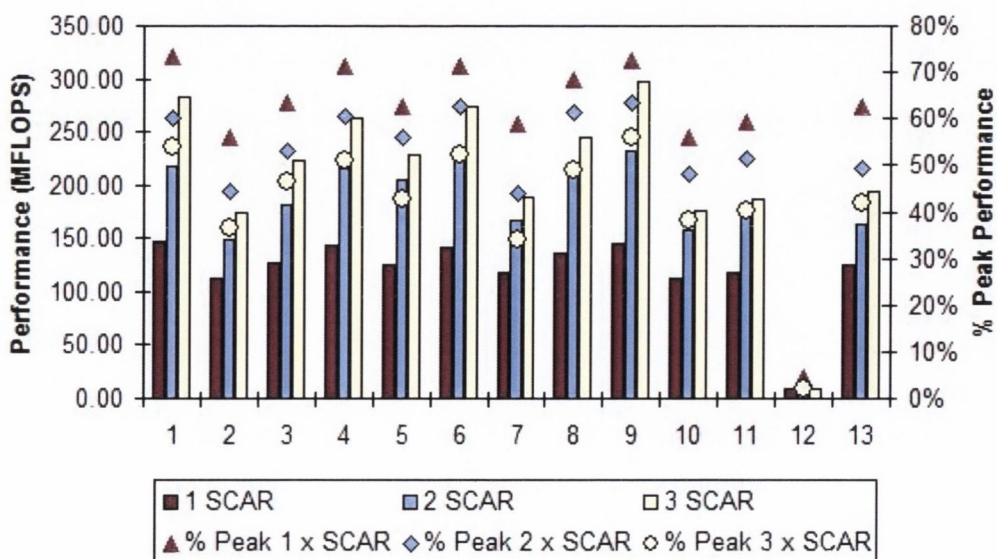


Figure 6.4 Performance Benchmarks for SCAR architecture

The average performance achieved by the SCAR architecture is 126 MFLOPs, 180 MFLOPS and 208 MFLOPs for a system containing one, two and three SCARs respectively. The SCAR unit achieves about 1.85 times the performance achieved by the SPAR architectures. Multiple SCAR units increase performance; however, performance does not scale linearly with the number of SCAR units. This discrepancy is caused by the same bus contention on the shared vector bus that reduces multiple SPAR units' performance. Load balancing between the two/three units may also contribute to the lack of linear scaling performance. The effects of this bus contention and load balancing will be discussed later (section 6.5). Also, the SMVM calculation is only part of the PageRank calculation. On average, it accounts for 77% of the calculation load in our test matrices. Multiple SMVM units increase SMVM performance, but do nothing to increase vector operations performance, which is limited at a peak computation bandwidth of 200 MFLOPs. The SCAR architecture utilises the adder 60%, 50% and 42% of the time for the system with one, two and three SCAR units respectively. These adder utilisation figures are an increase on the utilisation of 3-4% and ~35% achieved by both the GPP and the SPAR units respectively.

6.4.2.1 Matching Computational and Memory Bandwidth

In section 2.4.2, the clock speed needed to use the full memory bandwidth is given in Table 2.3. The SCAR uses a 12 byte or 96-bit data word and so would need to run at 175 MHz to use the memory bandwidth available from the DDR memory. The SCAR system can only achieve 100 MHz on the Virtex-II FPGA, and so does not use the available memory bandwidth. An improved dual MAC SCAR system was implemented to use the available memory bandwidth. The dual MAC SCAR system as described in section 5.5.3 has two arithmetic data paths internally and so can deal with two data words every clock cycle. This system, therefore, uses 2*12 Byte data words every clock cycle, which infers it can be clocked at 88 MHz and still use the entire memory bandwidth available. The dual MAC SCAR system also takes advantage of bigger X and Y buffers than the single SCAR system does. Increasing the buffer size and consequently the matrix tile size decreases the number of X-buffer reads and Y-buffer writes. Thus, the impact of the often large memory latency is reduced; see Table 2.4. The dual MAC SCAR system's buffers can accommodate

2,048 DWORDs. The single SCAR could only accommodate 1,024 DWORDs. This increase in buffer capacity was made possible by the more relaxed timing constraints of the dual MAC SCAR system. Reducing the required clock rate allows more logic and longer inter-connects between registers. Thus, this allows more of the device logic to be used. The 3x dual MAC SCAR system occupies 98% of the available slices on the FPGA. The FPGA utilisation figures for the dual MAC SCAR system containing three SMVM units is shown in Table 6.5.

Table 6.5 FPGA utilisation for floating point 3xDual MAC SCAR on Virtex-II

Logic name.	Utilisation (Available)	% used
Slice Flip-Flops	43720 (67584)	64%
4 input LUTs	53524 (67584)	79%
Multipliers	94 (144)	65%
Block RAM	134 (144)	93%
Gclk	8 (16)	50%
Occupied slices	33179 (33392)	98%

The dual MAC SCAR system can be clocked at up to 96 MHz, but, as discussed earlier, this system only needs to be clocked at 88 MHz. Increasing the clock rate further will not increase performance, since the memory bandwidth is fully utilised at 88 MHz. The dual MAC SCAR system uses the same 12 cycle adder used in the single SCAR system. However, because of the dual arithmetic path there are twice as many values are in the adder pipeline at any time, thus creating an increased chance of RAW hazards. The system has effectively doubled the adder latency and so it becomes more difficult for the SCAR reordering system to use interleaving to reduce RAW hazards. Thus, many more NOPs must be introduced to the stream to avoid these RAW hazards. The peak performance of the dual MAC SCAR is 352 MFLOPS for a single SMVM unit. Using equation (25) the computational peak of the entire PageRank algorithm is calculated as 320 MFLOPs, 575 MFLOPs and 840 MFLOPs for the dual MAC SCAR system with one, two and three SMVM units respectively. The dual MAC SCAR system with two and three SMVM units has a peak SMVM performance of 704 MFLOPS and 1.05GFLOPS respectively. These peak performance figures show clearly that the dual MAC SCAR's SMVM computational bandwidth matches the available memory bandwidth. The Vector Unit still does not utilise the full memory bandwidth and so reduces the overall peak performance

figures. The actual performance results are in Figure 6.5. The percentage of peak computational bandwidth of each of the results is also in Figure 6.5.

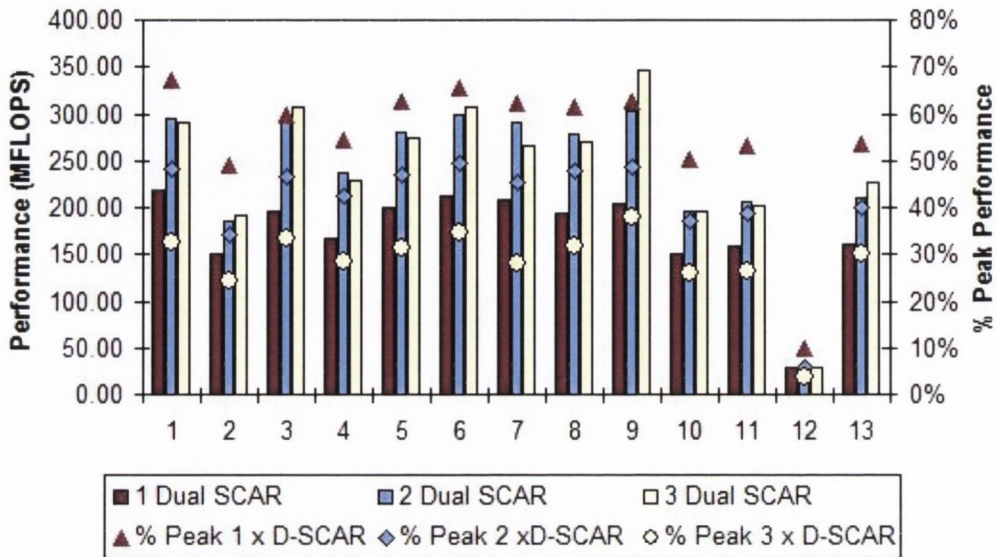


Figure 6.5 Performance Benchmarks for Dual MAC SCAR architecture

The dual MAC SCAR implementations achieved an average performance of 176 MFLOPS, 247 MFLOPS and 259 MFLOPS for a system containing one, two and three SMVM units respectively. The three dual MAC SCAR units performance is about 20% better than three single MAC SCAR units. As discussed in section 5.5.3 the additional MAC in the dual MAC SCAR systems effectively doubles the adder pipeline depth. This increases the probability of RAW hazards (see section 6.5.3). Thus, the number of NOPs added to the SCAR stream increases, which limits the performance of the dual MAC SCAR. The shared vector bus contention is once again evident in the move from two to three SMVM units. The performance of matrices 1, 4, 5, 7, 11 and 12 is lower for three SMVM units than for two SMVM units. The three dual MAC SCAR system still achieves a slightly better average performance than the two dual MAC SCAR. For this reason, the dual MAC SCAR system with 3 SMVM units is compared to the highest performing single SCAR system and the GPP in Figure 6.6. The bus contention issues will be discussed in more detail later.

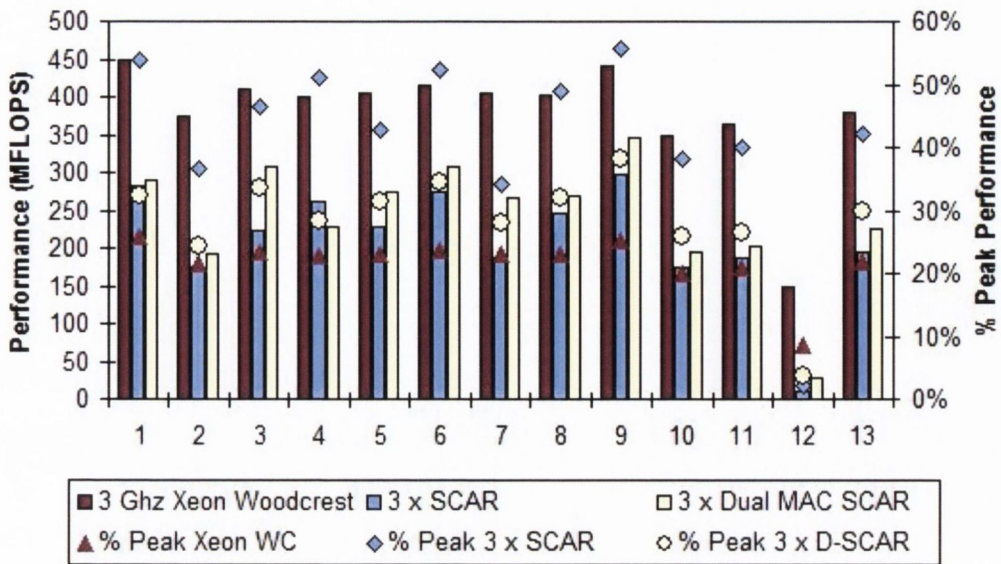


Figure 6.6 Performance Benchmarks for SCAR system Vs. PC

The SCAR and dual MAC SCAR systems perform well against the GPP, achieving 55% and 63% of the performance of the GPP respectively. The two FPGA based systems achieve this performance despite having a clock rate that is 30 times slower than that of the GPP and using memory that is less than 50% the speed of the DDR2-667 memory used by the GPP. The average adder utilisation is also higher in both the SCAR systems than the GPP. The SPAR and the SCAR architectures are examples of generic SMVM architectures designed for use with FE stiffness matrices rather than IA matrices. In the next section hardware specially designed for the PageRank algorithm is benchmarked.

6.4.3 SMVM Architecture 3 – PageRank HW

The PageRank HW architecture is designed specially for use with the PageRank algorithm. It replaces the SMVM with a dense vector element by element multiplication and a pattern addition. The results of this architecture can not be directly compared with others by using MFLOPs, because this architecture reorders the calculation. The results are instead presented as effective MFLOPs using a black box approach. The operations needed to replace the SMVM are timed. Knowing that every NZE needs to be operated on twice for SMVM, the effective MFLOPS rating is then computed. The system was implemented with 3 pattern adders, but due to X-buffer replication, the X-buffer size was limited to 1024 lines. Increasing the X-

buffer size further would inhibit three SMVM units being implemented on the Virtex-II device. Since each SMVM unit uses a dedicated memory channel, three SMVM units use the available matrix memory bandwidth. The Y-buffer contains 2048 lines; each of which can contain a single double precision word. The adder used in the pattern adder is an 8 cycle double precision floating point adder generated by Xilinx Core library V8.2.03. The FPGA utilisation details are given in Table 6.6. On foot of the SCAR results, the decision was made to implement a dual arithmetic path system only, since it used more of the memory bandwidth available to the system. Therefore, the PageRank system utilises 100% of the memory bandwidth, which for DDR-266 is 2128MB/s per channel. The dual PageRank system uses 32 bits of streaming data to represent NZE and so processes six NZE per clock cycles. Thus, the PageRank pattern adder has an effective peak performance of approximately 1 GFLOPS, 2 GFLOPS and 3 GFLOPS for one, two and three PageRank pattern adders respectively. Using equation (25), the peak computational bandwidth of the PageRank HW system for the entire PageRank algorithm is 800 MFLOPs, 1.5 GFLOPs and 2.3 GFLOPs for the PageRank HW with one, two and three pattern adders respectively.

Table 6.6 FPGA utilisation for floating point Dual PageRank system on Virtex-II

Logic name.	Utilisation (Available)	% used
Slice Flip-Flops	34345 (67584)	53%
4 input LUTs	51296 (67584)	75%
Multipliers	12 (144)	8%
Block RAM	122 (144)	84%
Gclk	8 (16)	50%
Occupied slices	31561 (33392)	93%

The PageRank HW system occupies 93% of the available slices and 84% of the Block RAMs. It can run at up to 97MHz and so can easily match the computational bandwidth of the DDR-266. The number of multipliers used by the system is greatly reduced as compared to the dual MAC SCAR system⁸. This reduction in multipliers used is because the PageRank HW Pattern adder removes the multiplication and does it as a dense vector element by element multiplication using the vector unit. The

⁸ Benefits of reducing the number of multipliers include, more freedom for place and route, smaller overall design size and would allow the system to be implemented on Virtex-4 which has a smaller multiplier to logic ratio than the other Xilinx FPGA families.

results for systems using one, two and three PageRank HW pattern adders are in Figure 6.7.

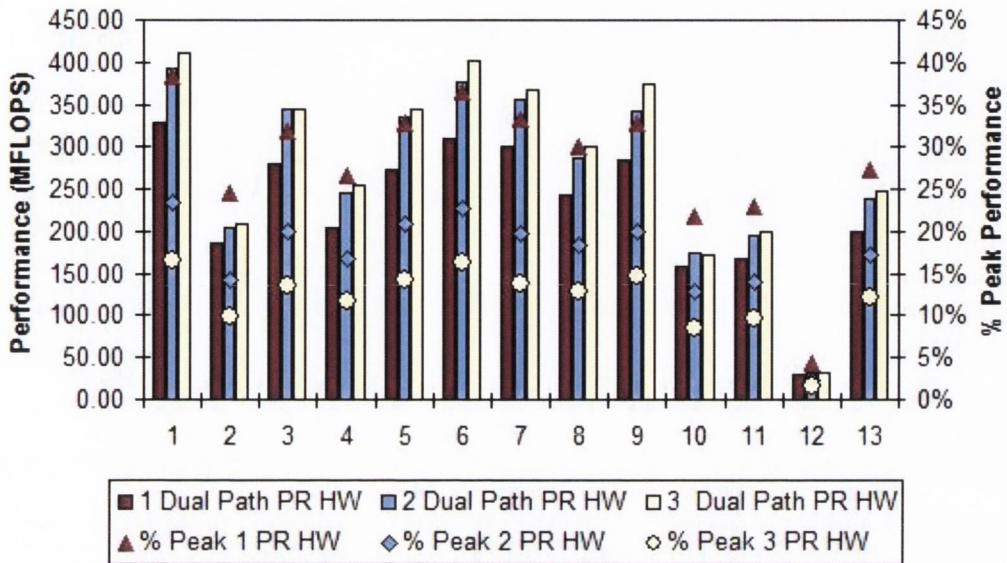


Figure 6.7 Performance Benchmark for Dual 1024 line X-Buffer PageRank system

The PageRank HW shows an increase in performance over the dual MAC SCAR system with an average of 228 MFLOPs, 270 MFLOPs and 281 MFLOPs for the PageRank HW system with one, two and three pattern adders respectively. The performance increase is due to the fact that the pattern adder can deal with 6 NZE per clock cycle. This factor reduces the stream length. However, the performance improvement is very modest due to three factors. The shared bus contention problem discussed earlier is still an issue. This problem is especially evident in matrices 3 and 10, which show a reduction in performance when the third pattern adder is used in the system. The second issue is RAW hazards, which occur very frequently because of the 16 cycle adder tree. This becomes effectively a 32 cycle adder when the dual arithmetic path is taken into account. The final issue faced by the architecture is the sparseness of the matrix. The 1024 entry X-buffer reduces the number of NZE available to make the stream per block, and so NOPS have to be placed in the stream to avoid RAW hazards. The reordering scheme will choose any NZE which will not cause a RAW hazard. However, in small blocks, the choice is greatly reduced and so the reordering software must pad the stream with NOPS to avoid hazards. Since all of the architectures are affected by NOPs details of the NOP frequency and solutions to this problem are discussed in section 6.5.3. The adder utilisation is greatly reduced

in the PageRank HW due to the very sparse matrix. The PageRank HW takes up to three values from the same row on every clock cycle. The IA matrices used in these tests often don't have enough NZE to utilise this scheme to its full potential.

In an attempt to increase performance, a second version of the PageRank HW system was implemented. This system aimed to alleviate bus contention on the shared vector bus and to reduce the number of NOPs being added to the pattern adder's stream.

This system only contained two dual path pattern adder units, but the X-buffer size was doubled to store 2048 double precision numbers. The other details of the system remained the same. Using fewer pattern adders should have reduced bus contention, since there are fewer units accessing the shared bus. The larger X-buffer doubles the size of the blocks and thus should reduce the number of NOPs in the stream by increasing the number of NZEs which the reordering scheme can choose from when trying to avoid RAW hazards. Also, larger blocks should reduce the number of times the pattern adder units need to access the bus to read in new X vector values. The FPGA utilisation of this second PageRank system is given in Table 6.7.

Table 6.7 FPGA utilisation for modified floating point PageRank system on Virtex-II

Logic name.	Utilisation (Available)	% used
Slice Flip-Flops	33962 (67584)	50%
4 input LUTs	49509 (67584)	73%
Multipliers	12 (144)	8%
Block RAM	134 (144)	93%
Gclk	8 (16)	50%
Occupied slices	30874 (33392)	91%

This new system has a slightly smaller number of occupied slices due to the removal of a pattern adder, but the block RAM usage has increased from 84% to 93%. The system can still achieve 97MHz clock rate. The peak performance for the entire PageRank algorithm, calculated by equation (25), remains 800 MFLOPs and 1.5 GFLOPs for the modified PageRank HW with one and two dual path pattern adder units respectively. The performance results for this new system are presented in Figure 6.8.

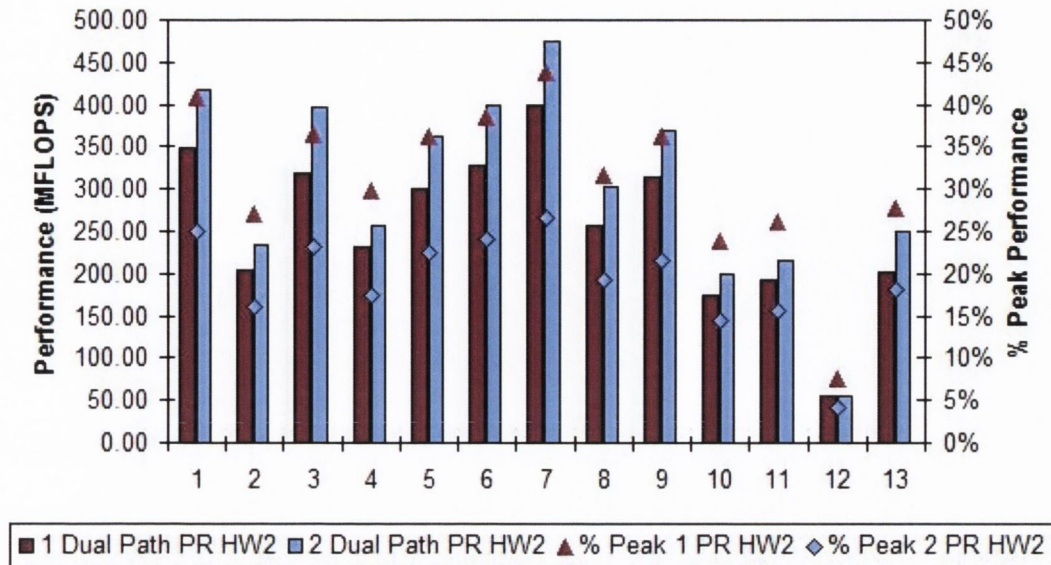


Figure 6.8 Performance Benchmark for Dual 2048 line X-Buffer PageRank system

The second PageRank system with the 2048 double precision word cache has an average performance of 255 MFLOPS and 302 MFLOPs for one and two pattern adders respectively. This system leaves one of the memory channels unused. If a copy of the X vector was made and stored in this unused memory channel, the shared bus contention could be alleviated. This alteration should improve the performance and will be discussed in section 6.5.1. However, the performance increase achieved by this change would be limited, as it would only increase performance in the pattern addition portion of the PageRank algorithm. The time taken to perform the vector operations would remain the same, as they would not be affected by this change. An extra memory copy would also have to be added to the system, which would also reduce the performance boost achieved.

The PageRank system performance is compared with the GPP performance in Figure 6.9. The modified PageRank architecture with the larger X-buffer and only two pattern adder units performs equally well or outperforms the original PageRank HW architecture with 1024 line X-buffers and 3 pattern adders for all the matrices. The PageRank system with the larger cache performs 20% better than the GPP on matrix 7, despite being clocked at over 30 times slower and using memory that is more than half the speed of DDR-667 used by the GPP. In general, however, the GPP performs about 20% better on average than the PageRank HW for the set of matrices.

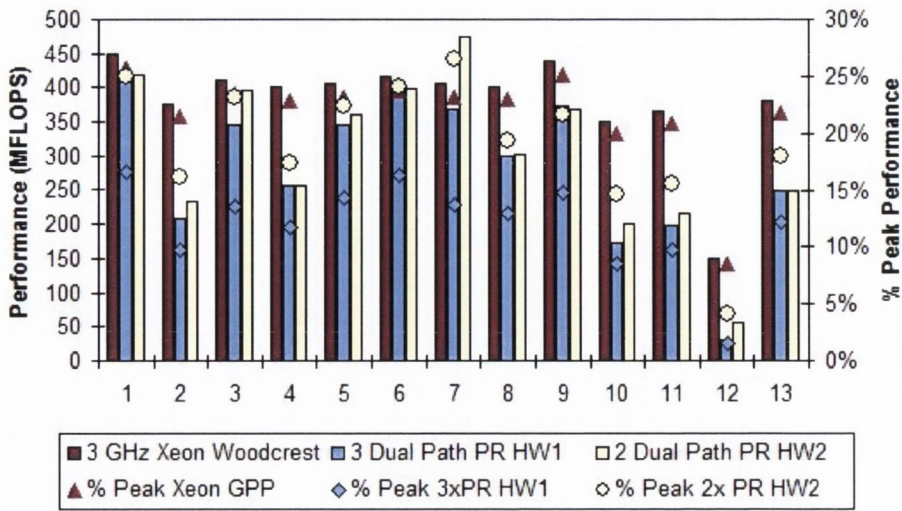


Figure 6.9 Performance Benchmarks for PageRank system Vs. GPP

6.4.4 SMVM Architectures Compared

A summary of the best performance results obtained from each of the FPGA architectures implemented on the Virtex-II FPGA is presented in Figure 6.10. The GPP performance is also included for comparison. The dual PageRank architecture achieved the best performance of all the FPGA architectures. It performs on average about two and half times better than the SPAR architecture and about 25% better than the dual MAC SCAR architecture. The dual PageRank architecture even outperforms the GPP on two of the test matrices despite having a clock that runs 30 times slower than the GPP clock.

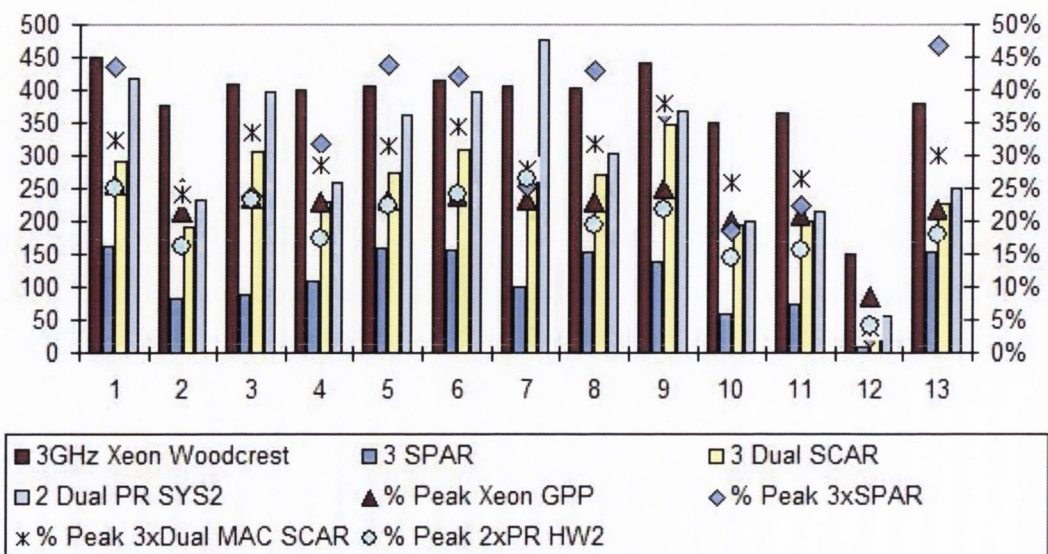
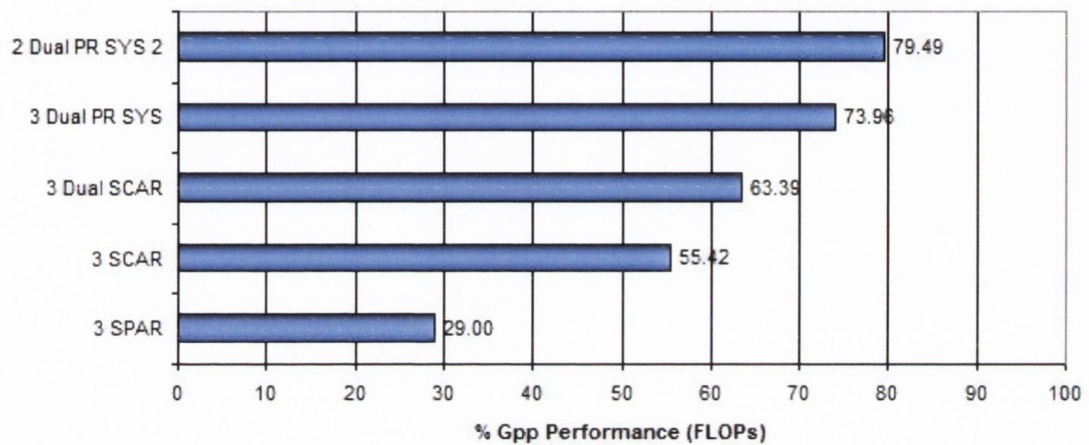


Figure 6.10 Summary of Floating point Benchmarks on Virtex-II

The GPP has a higher average gross FLOPs performance across the range of test matrices. This is achieved with a clock rate of over 30 times the clock rate achievable on the Virtex-II FPGA and using a memory that can supply data at over twice the speed of the DDR-266 used in the development platform. The average performance of the FPGA architectures as a percentage of GPP performance is shown in Figure 6.11.

**Figure 6.11 Average percentage of GPP performance achieved by test architectures**

The Dual PageRank HW system with the 2048 row and column block size achieves 79% of the average performance of the GPP. The Dual MAC SCAR and SCAR architectures achieve an average 63% and 55% of the GPP performance respectively. The SPAR architecture achieves less than 30% of the GPP performance. Gross FLOP performance results like those in Figure 6.11 can only go so far in measuring the performance of a new design. Combining these results with a measure of the overall efficiency of the architecture can be very useful when comparing architectures. In Figure 6.12, the percentage adder utilisation or percentage of the peak achievable computational bandwidth is shown for the GPP and all the FPGA architectures running the PageRank algorithm. The GPP only achieves 3-4% of its computational peak performances but this graph takes into account that limiting factor for the GPP running the PageRank calculation is its FSB speed and so the GPP is expressed as % of peak achievable computational bandwidth for the PageRank algorithm.

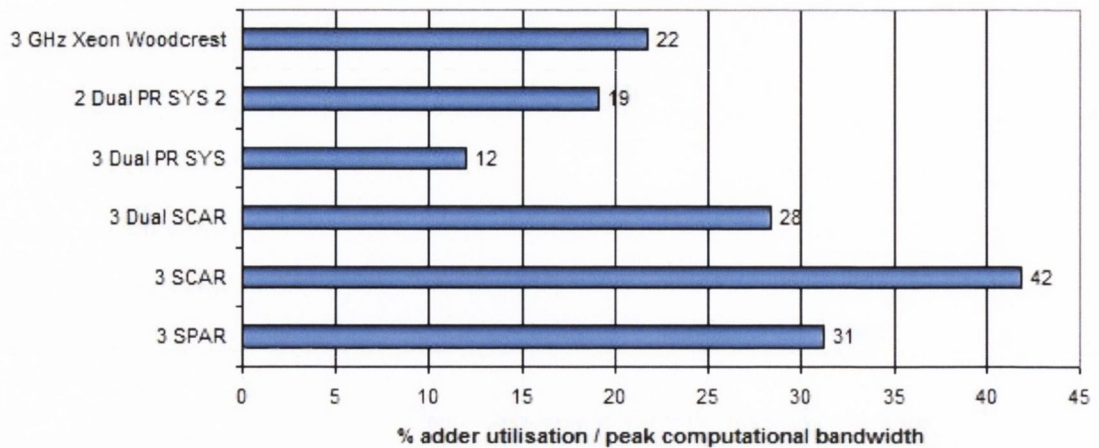


Figure 6.12 % of achievable computational bandwidth or % adder utilisation for the GPP and all the FPGA architectures running the complete PageRank solver

The SCAR and the SPAR architectures have the highest adder utilisations in these tests with adder utilisation of 42% and 31% respectively; see Figure 6.12. The higher adder utilisation of the SCAR unit suggests that row based architectures perform better than column based architectures when calculating the PageRank algorithm on IA matrices. It was decided because of the poor SPAR results that no further tests would be run on the SPAR architecture. The SCAR and the PageRank system both exhibited promising results. The SCAR shows the highest adder utilisation and the dual MAC SCAR achieved 65% of the GPP performance using DDR-266. Upgrading to DDR2-667 should be enough to make the dual MAC SCAR exceed the GPP performance. The PageRank HW exhibits very poor adder utilisation due to the fact that it is a highly paralleled architecture. The IA matrices do not have enough NZE to fully utilise the PageRank HW's bandwidth without causing RAW hazards. The gross FLOPS performance of the PageRank HW is the highest of all the FPGA architecture with an average of 80% of the GPP performance, despite the slower memory used in the development platform and the much slower clock rate.

The performance of all the FPGA architectures could be increased with an upgrade to faster memory or an increase in clock rate. Throughout this chapter, a number of other issues that limited the FPGA architectures performance arose. These issues were bus contention on the shared vector bus, NOPs being added to the stream to avoid RAW hazards and load balancing between multiple SMVM units. In the next section, these issues and the limits they place on the FPGA architecture performance are discussed.

6.5 Performance Limitations

The results thus far have highlighted a number of the problems with solving the PageRank Eigenvector problem on FPGA. These problems include bus contention on the shared vector bus, NOPS in the data stream and very sparse matrices associated with the PageRank problem. NOPS are added to the stream of both SCAR and PageRank architecture, to avoid RAW hazards, as discussed in section 5.5.1 and 5.6.2 respectively. These NOPS reduce the FPU utilisation, since no value is placed in the adder pipeline on a NOP cycle. Thus, the performance of the system is reduced. Using multiple parallel SMVM units also raises the issue of load balancing. For maximum performance, it is important that the NZE data is distributed equally among all SMVM units. In the next sections, the causes and effects of bus contention, load balancing and RAW hazards will be discussed.

6.5.1 Bus Contention

In section 5.7, the causes of bus contention were discussed. Bus contention occurs in this system when more than one SMVM unit requires use of the shared X/Y Vector bus at the same time. The bus is arbitrated in a round robin fashion, and so one of the SMVM units gets to use the bus while the other units are forced to wait for their turn. Sharing a data bus in hardware is common. No problem results if each of the units only uses the bus sparingly, as is the case in FE problems where each SCAR utilises the shared vector bus an average of 11% of the time. A three SCAR unit computing SMVM on FE matrices described in Appendix One only has activity on the shared vector bus 32% of the time. The three-SCAR system performs on average 2.75 times faster than the single SCAR matrix when computing SMVM on FE matrices. This outcome shows that in FE matrices the shared bus is not overly utilised. The shared bus % utilisation for a single SCAR system with X and Y buffers of 2048 lines computing SMVM on IA matrices is presented in Table 6.8. On average, the IA matrices use the shared bus over two times more than the FE matrices.

Table 6.8 Shared Vector Bus utilisation in SCAR architecture

Num.	Name	Duration (cycles)	Bus usage (cycles)	Vector bus Utilisation (bus-usage/duration)
1	Arabic-2005	17627234	1958879	11.11%
2	cnr-2000	7200037	2048316	28.45%
3	eu-2005	25373217	5661907	22.31%
4	in-2004	10628335	1931251	18.17%
5	Indochina-2004	14444211	1798849	12.45%
6	it-2004	17453016	1858245	10.65%
7	sk-2005	32485443	6765244	20.83%
8	uk-2002	11475177	1456032	12.69%
9	uk-2005	15106095	3542367	23.45%
10	web-mat-1.5M	26818645	10639489	39.67%
11	web-mat-1M	16336374	5852331	35.82%
12	web-Stanford	28671556	38905039	81.75%
13	webbase-2001	6677817	1097817	16.44%
	Average:	19170530	6424289	25.68%

The matrices with the highest vector bus utilisation figures are the matrices that showed the smallest increases in performance when migrating from single SMVM systems to multiple SMVM systems. This result is especially apparent in all three SMVM architectures on matrix 12 and less apparent in matrix 2, 10 and 11. These matrices use the vector bus for 81%, 28%, 39% and 35% of the SMVM duration respectively. The major consideration that leads to excessive use of the shared vector bus is the sparsity of the matrix blocks. This sparsity can be caused by overall matrix sparsity or by NZE positioning. A partitioned matrix has two types of block, viz Zero blocks and Non-Zero (NZ) blocks. Zero blocks contain no matrix data and can be ignored. NZ blocks contain NZE and must be processed. A matrix with many NZ blocks with few NZE will use the shared vector bus more than a matrix with the same amount of NZE with fewer NZ blocks. Therefore, the number of NZE per NZ matrix block can affect the amount the shared vector bus is used. The IA matrices are very sparse, and their NZE are not tightly banded together and so have many NZ blocks with few NZE, thus causing excessive use of the shared vector bus. Figure 6.13 is a histogram of the NZE per NZ block of the IA matrices and the FE matrices. The bars in the histogram represent the percentage of NZ matrix blocks that contain a given number of NZE. Blocks with less than 1000 NZE go into the first category,

blocks with more than 1000 but less than 2000 go into the second etc. The first bar of the histogram, therefore, shows that almost 90% of matrix blocks in IA matrices contain less than 1000 entries. This is about 50% more blocks than the FE matrices have in this category.

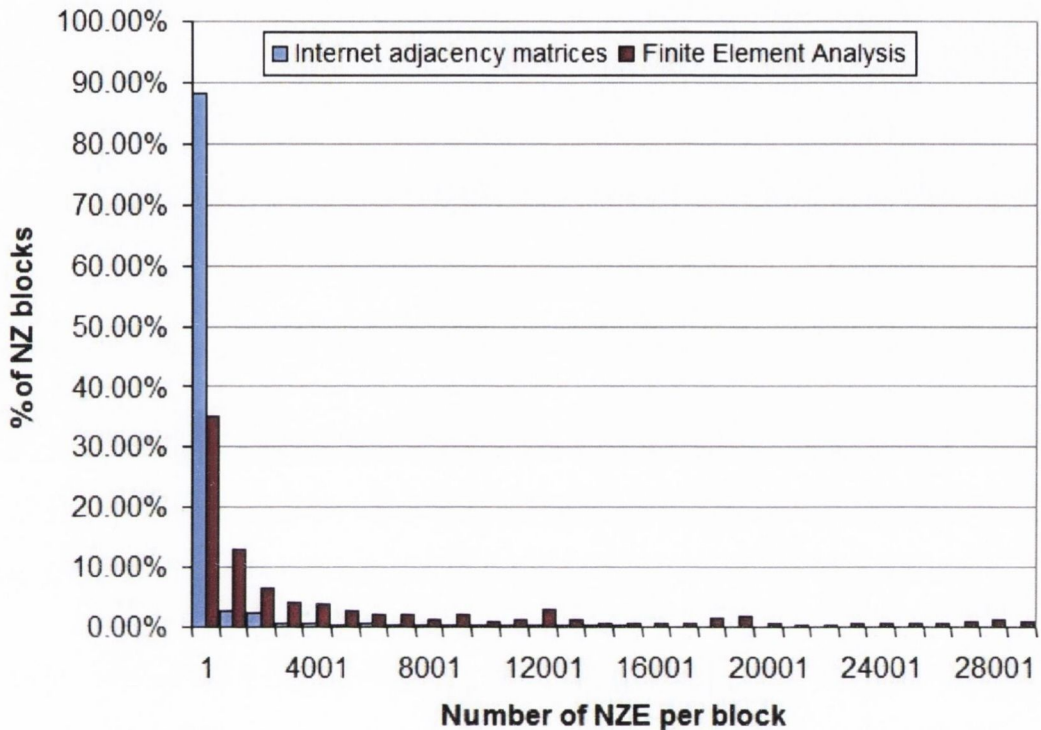


Figure 6.13 Histogram of NZE per NZ matrix block

The FE matrices contain many more relatively dense blocks when compared with the IA matrices. This result occurs because of the loosely banded layout of the IA matrices. The X-buffer in the SCAR can store 2048 X values. To fill this buffer at the beginning of a block can take about 1500 clock cycles if the whole X fragment is needed. According to the histogram in Figure 6.13, almost 90% of the blocks in the IA matrices contain less than 1000 entries. It is therefore possible to process the block quicker than it is to fill the X-buffer on every block update. In a system where this is the case, adding a third SMVM unit will have no effect. The first unit reads in its X-buffer and then starts to process the block. A second SMVM unit would then start filling its X-buffer. However, the first SMVM unit would finish processing its block before the second SMVM unit reads in the X-buffer and so waits for the vector bus to become free. Once it is free, the first block starts reading the X values for its

second block, and so on. A system will always only have one SMVM unit processing information and one SMVM unit filling its X-buffer. A third SMVM unit would not increase performance as the vector bus is already saturated with read requests.

In the second PageRank HW system, the size of each X-buffer was doubled at the expense of the third Pattern adder. One of the four memory interfaces on the FPGA was thus left unused. This unused interface could be used to make a copy of the X-vector to alleviate the bus contention on the shared vector bus. As discussed earlier the performance of the second PageRank HW system would not then be doubled as the Pattern addition is only part of the PageRank calculation. An extra memory copy would also need to be implemented, which would further reduce the performance benefit. This memory copy would make a copy of the *X* vector at the end of each iteration to the second vector memory channel. Thus, each SMVM unit would have exclusive access to a copy of the *X* vector. This system was not implemented in hardware as it would have involved a major redesign of the architecture and the results can be extrapolated from results already obtained from the architecture.

Figure 6.14 shows the extrapolated SMVM performance of the dual path PageRank system with the 2048 line X-buffer and two vector buses to alleviate bus contention. The results include all vector operations needed to achieve a correct SMVM of the matrix and vector. The results for the single pattern adder include the element by element vector multiplication which makes the pattern adder equivalent to a full SMVM unit. A memory copy is added to the second set of results, in addition to the element by element multiplication, since the architecture using both vector buses needs to share their updated vectors at the end of each iteration.

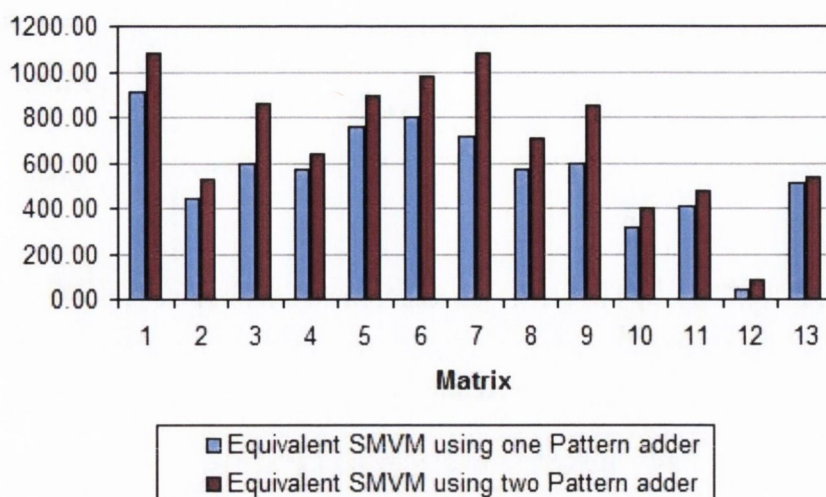


Figure 6.14 SMVM Performance of PageRank system with two vector buses to alleviate shared bus contention.

The architecture with two SMVM units is about 30% faster than the single SMVM architecture. However, if the memory copy used to duplicate the X vector into the second vector memory of this system is ignored the two SMVM unit system performance increases to 70% faster than the single SMVM unit. This result is a great deal bigger than the performance increase achieved between the one and two SMVM unit PageRank systems, which shows that bus contention is really a serious issue with IA matrices. The memory copy needed to allow the SMVM units to use independent copies of the X vector reduces significantly the performance improvements achieved by using two independent memory channels for the X vector. Reducing bus contention has little effect on the overall PageRank algorithm performance, due to the large number of vector operations being carried out per iteration; see Figure 6.15. The two SMVM units now only perform about 10% faster than the single SMVM unit system. Matrix 12 doubles performance in both Figure 6.14 and Figure 6.15 when bus contention is removed. The results are still less than 50% the average performance of the test set of IA matrices. Matrix 12 has caused major problems for all the architectures. Clearly then high performance figures for matrix 12 are very difficult to obtain.

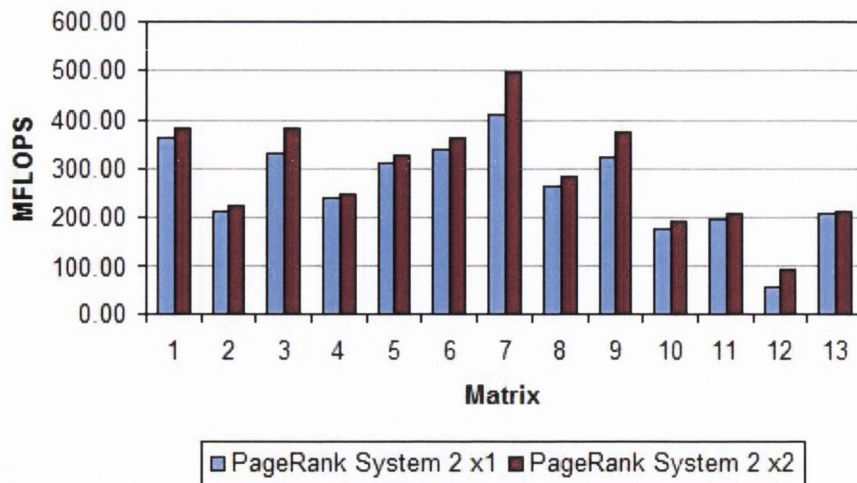


Figure 6.15 PageRank Algorithm performance of PageRank system with two vector buses to alleviate shared bus contention.

Bus contention in this system happens on the shared vector bus and is caused by the very sparse, non-tightly-blocked matrix structure inherent to IA matrices. The performance with FE stiffness matrices proves that the SMVM architectures are efficient given very sparse matrices that have a defined structure. Using multiple memory channels for the shared vector bus only increases performance of the PageRank algorithm by 10%. This is due to the extra memory copy needed in the algorithm. The Vector Unit is now the bottleneck of this algorithm. Parallelising of the Vector Unit would increase the performance of the overall system which is discussed in more detail in section 7.3.

6.5.2 Load Balancing

One of the major challenges in parallelising an algorithm is load balancing the operations between all processing nodes. The results presented so far in this thesis use a naïve matrix partitioning system. The matrix is simply split up into an equal number of rows for each processing element being used. In FE matrices, this process proves to be very satisfactory as the matrices are symmetrical, structured and banded around the diagonal. The stream length of the SCAR data stream running on each of the three dual MAC SCAR units was measured for each of the FE matrices in Appendix One. The average difference between stream lengths on the three SMVM system was approximately 3% with a max difference of 20% on a single test matrix. This means that on average the three SMVM units load for the FE matrix

calculations differed by approximately 3%. These load balancing tests were run on the IA matrices on the dual MAC SCAR system with a 2048x2048 block size and the results are in Figure 6.16.

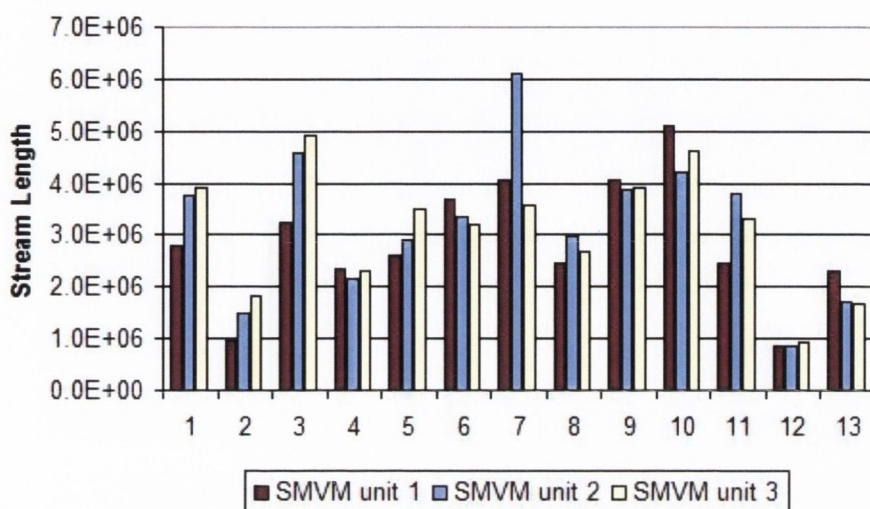


Figure 6.16 Load balancing analysis of the IA matrices in the dual MAC SCAR system

Figure 6.16 clearly shows that IA matrices are not uniform like the FE matrices above. Since the three SMVM units run in parallel, the last one to finish dictates when the SMVM calculation finishes. The majority of the IA matrices show that they could benefit from better load balancing techniques. This is especially evident in matrix 7, where the second SMVM unit must operate on between 1.5 and 1.7 times more data words than the first and third SMVM units. The average load difference between the three SMVM units is 10% of the total stream length.

One method that could be used on IA matrices to more efficiently spread the load across the SMVM units would be to split the matrix into stripes the size of the Y-buffer (i.e 2048 rows). The SCAR stream for this stripe can be calculated and then written to the memory space associated with SMVM unit that currently has the shortest stream. This method should not increase the time to calculate the SCAR stream and could have a dramatic effect on load-balancing of the calculation. Figure 6.17 shows the effect this new load balancing scheme has on the length of the dual MAC SCAR streams for the three SMVM units. This scheme has reduced the difference between the loads on the SMVM units to an average of less than 1%, which should increase the performance of the three SMVM units working in parallel by almost 10% on average.

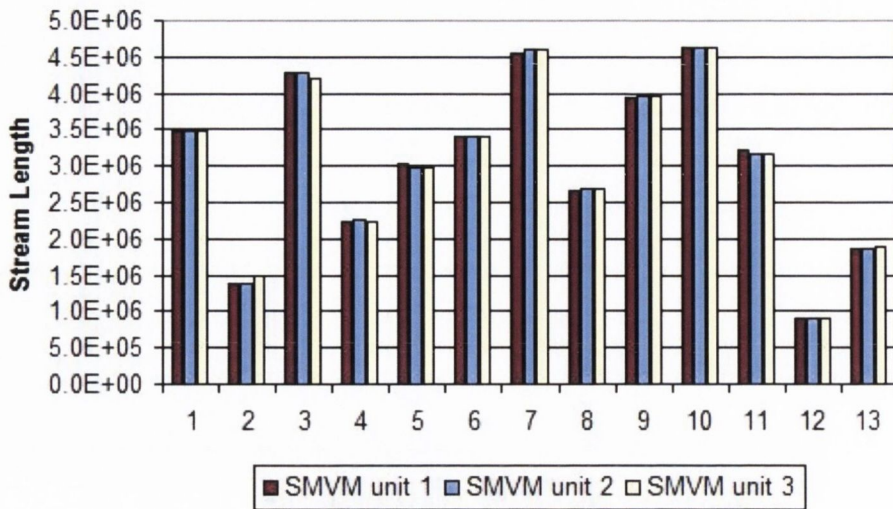


Figure 6.17 Load balanced calculation of SMVM on IA matrices

6.5.3 Measuring NOP frequency

In order to achieve high performance results using the SCAR and PageRank HW architectures, NOPs must be kept to a minimum. NOPs are wasted clock cycles, since no results are computed on a NOP cycle. The FPGA runs at over 30 times slower than the GPP, and thus it is important that every clock cycle in the FPGA is used. In section 5.5.1 two reordering schemes for the SCAR architecture were presented. The first of these is simple modulo- n reordering where n is the adder pipeline depth. The second is opportunistic modulo- n reordering.

The simple reordering scheme ensures that all values in a given row are kept exactly the adder latency apart and thus allows the SCAR to proceed without RAW hazards. This reordering is easily done in software and proved very efficient when working with FE matrices for which it was designed. Table 6.9 shows the average percentage NOPs in the SCAR stream over the FE and IA matrix test sets using simple and opportunistic reordering. These results assume an adder latency of 14 cycles, which is the adder pipeline depth in the single MAC SCAR unit. The FE test set only has ~7% NOPs in the data stream when simple reordering is used. However, when simple reordering is used on the IA matrices, the single MAC SCAR stream contains almost 21% NOPs. This means that one-fifth of the time the SCAR is doing nothing so that it can avoid RAW hazards.

Table 6.9 Nops in SCAR data stream

DATA SET	% NOPs in data stream	
	Simple reordering	Opportunistic reordering
Finite Element	6.76%	0.84%
Internet	20.98%	7.92%

Opportunistic reordering was developed to increase the performance of the SCAR architecture. Opportunistic reordering organises row NZEs to be at least the adder latency apart; see Section 5.5.1. When this reordering scheme is applied to the FE matrices, it reduces NOPs in the stream to less than one percent in the 14 cycle adder single MAC SCAR system. It also reduces the NOPs in the IA matrices streams to an average of around 8%. Performance-wise, the opportunistic reordering boosts the SCAR architectures performance by about 12%. Table 6.10 shows the average amount of NOPs added to the data stream for all the FPGA architectures. The SPAR, as discussed earlier, uses more than half its time avoiding RAW hazards. The SCAR systems have about 8%-12% NOPs in their data streams and so could benefit from a further reduction. The specialised PageRank HW contains an average of 16% NOPs in its data stream. The PageRank HW data word can contain 3 NZE, and so a NOP is counted as a data word that contains no NZE, or a data word that is completely wasted. The quantity of NOPs in the stream for the PageRank HW increases to about 35% if the NZE-positions not filled are counted.

Table 6.10 A summary of the average percentage of Nops in FPGA architectures data stream using opportunistic reordering.

Architecture	% NOPs in data stream (approx)
SPAR	55%
single MAC SCAR	8%
dual MAC SCAR	12%
PageRank HW	16%

The number of NOPS caused by RAW hazards in the stream is affected by two parameters. These are the matrix structure and the adder latency. Matrix reordering schemes can be used to change the order of sparse matrices to increase data locality. These schemes attempt to compact the extremely sparse IA matrices into fewer blocks. The denser NZE blocks allow more choice in the data reordering schemes

and thus reduce the number of RAW hazards. One such matrix reordering scheme is Reverse Cuthill McKee (RCM) [83]. RCM aims to reorder matrix nodes to increase data locality. Figure 6.18 shows the effect that RCM reordering has on the number of NOPs in the dual MAC SCAR data stream. RCM actually increases the number of NOPS in the stream by an average of 4% for the IA matrix set. Matrices 10, 11 and 12 have an increase of more than 10% of NOPS in the dual MAC SCAR stream when RCM is applied. Matrix 4 did see a slight reduction of 2% in the number of NOPs in the stream. It is clear from this data that RCM is of little use for this particular matrix set and so cannot be used to increase performance of the PageRank algorithm further.

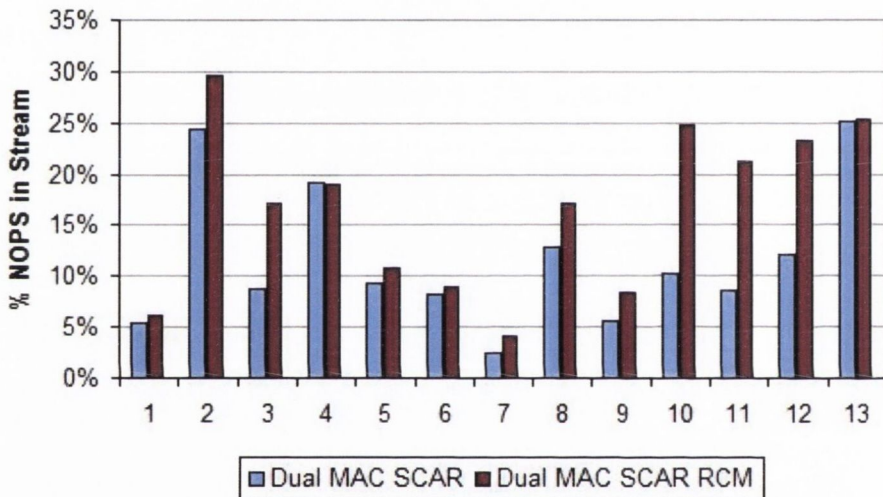


Figure 6.18 The effect of RCM on the number of Nops in the Dual MAC SCAR stream (lower is better)

Since reordering the very sparse structure of the IA matrices did not increase performance an investigation into the effects of adder latency on the number of NOPS in the data stream was carried out. Shorter pipelined adders should have the effect of reducing RAW hazards and thus reducing NOPs in the SCAR stream.

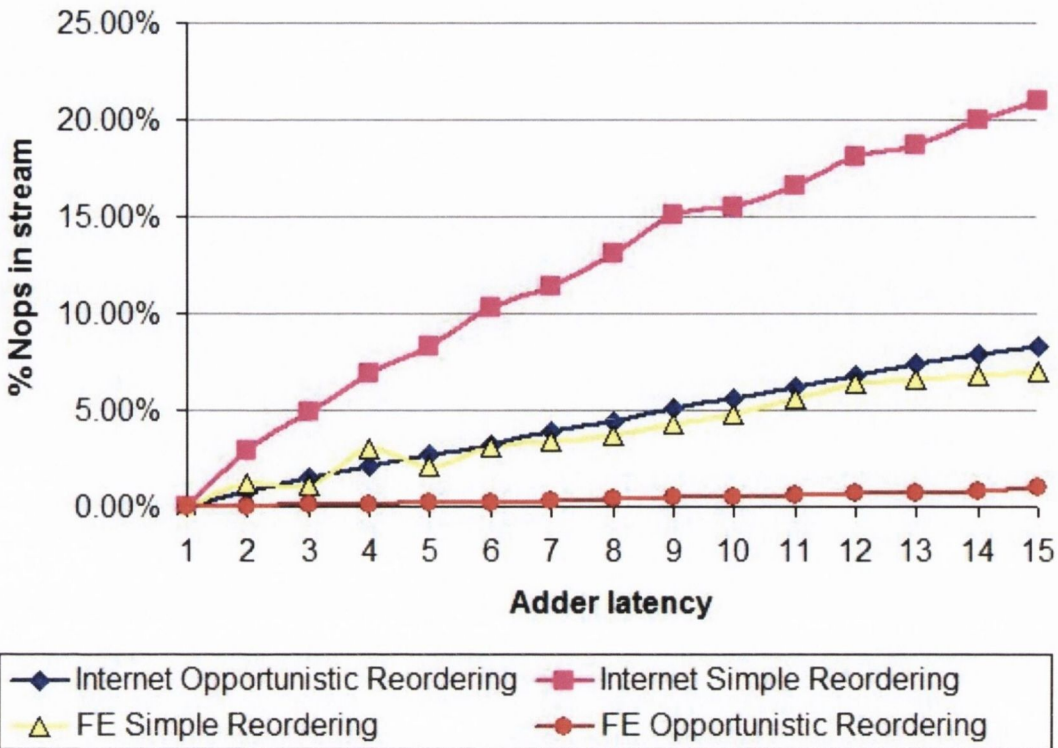


Figure 6.19 Nops in SCAR stream vs. adder latency

Figure 6.19 shows the effects of adder latency on NOP insertion in the SCAR stream. It is clear from the graph that opportunistic reordering reduces the number of NOPs in the stream compared with the simple reordering technique. Figure 6.19 also clearly shows the number of NOPs in the SCAR stream is directly proportional to the adder latency. In order to achieve the most efficient implementation of the SCAR architecture for use with IA matrices the adder latency must be reduced. The adder in the original version of SCAR was a 14-cycle double precision floating point adder. Significant problems were not caused with the FE matrices that have as little as 6% and 1% NOPs in the stream using simple and opportunistic reordering respectively. The SCAR architecture works much better on these FE matrices than for IA matrices. This high-pipelined adder was used to ensure a high clock rate, and thus memory bandwidth utilisation. In order to reduce the adder latency and maintain a high clock rate, the precision of the calculation will also need to be reduced. In section 6.5, the precision considerations for the PageRank calculation will be discussed, including the effects of reducing numeric precision on result accuracy, clock rate and the overall architecture performance.

6.6 Precision

The FPGA architectures described so far have used IEEE double precision floating point arithmetic. Thus, the accuracy of the results and scalability of the system are guaranteed. The floating-point cores used were generated using Xilinx Core Generator [102]. As discussed in the preceding section, RAW hazards reduce the overall performance of the system. To reduce this risk, a lower latency adder could be used. One way to reduce the adder latency is to reduce the precision of the calculation. According to Langville, double precision arithmetic is not needed in the PageRank problem [41]. She argues that a lower precision can be used since only a fraction of the PageRank vector is used for any given query and users usually only look at about the top 20 results of the results returned by the search engine to a query. The argument states that with relatively few pages being returned to the user, the probability of two pages of similar rank being returned are reduced. Most queries return a handful of highly ranked pages and lots of pages with very low ranking scores. Thus it is easy to sort the returned results. However, the weakness of this system comes to light in the exceptional query that returns only low ranking pages and thus makes it difficult to sort the results. In this case it would be important that other scores would help sort the pages effectively. An investigation into arithmetic precision and its effects on the ordering of the PageRank vector was conducted and is presented here.

6.6.1 Fixed point Emulation

Before hardware was designed and built, a simple fixed-point emulator in C was developed to assess the viability and effect of using fixed point arithmetic on the PageRank algorithm. The emulator could be set to emulate any fixed-point precision needed. All inputs were truncated to the required precision and the result of every calculation carried out was truncated to the lower precision. The PageRank calculation was emulated and the results were compared with the results obtained from the double precision calculations. The absolute value of a page's PageRank score is not important. The relative order of the pages is a better measure of how much precision is needed. The % correct order gives the % of pages that appear in the right order when the emulated PageRank vector is ordered and compared with the ordered PageRank vector calculated using double precision arithmetic.

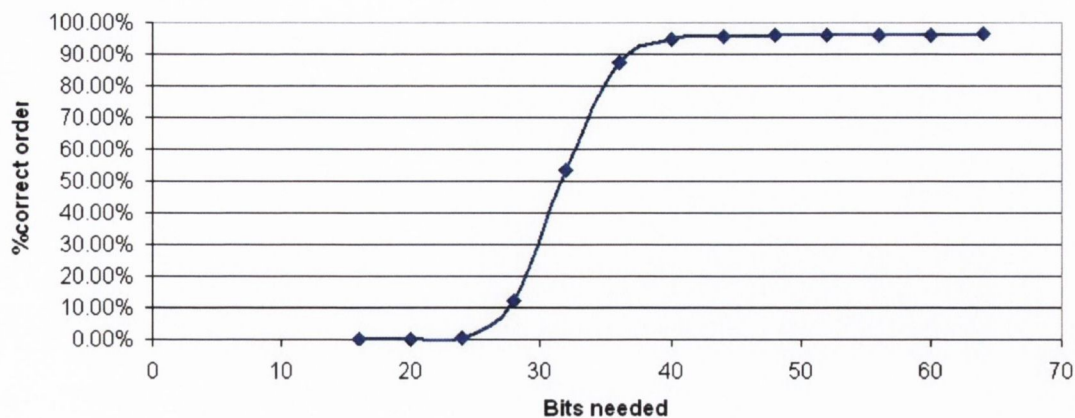


Figure 6.20 The fixed-point bits needed vs. percentage correctly ordered

The tests were run for a variety of fixed point precisions ranging from 10 bits all the way up to 64 bits for 50000 node subsets of the test matrices. The average number of bits needed to correctly order a given percentage of the nodes is shown in Figure 6.20. The matrices all contain 50000 rows and columns. Subsets of the test matrices were used because fixed point emulation was very slow. Figure 6.20 shows that about 40 bits of fixed point precision is needed for these matrices. Only a small increase in the percentage of correctly ordered nodes is obtained by increasing the precision further. Below 25 bits of fixed point precision, the system does not have the resolution to calculate the PageRank vector for these matrices and so 0% of the PageRank vector gets to be in the correct order.

The next test was to find the effect of matrix size on the fixed point precision needed. The PageRank calculation was calculated for test matrices ranging in size from 5000 – 50000 rows over a range of fixed point precisions. The fixed point precision needed to calculate the PageRank vector to 95 % correct order and 70% correct order is in Figure 6.21. Clearly a greater precision is needed to correctly calculate PageRank vector for bigger matrices. To achieve a PageRank vector that is 95% in the correct order 30 bits are needed for a matrix with 5000 rows and 38 bits are needed for a matrix containing 50000 rows. Figure 6.21 and Figure 6.20 both show that a greater precision results in a more correctly ordered PageRank vector.

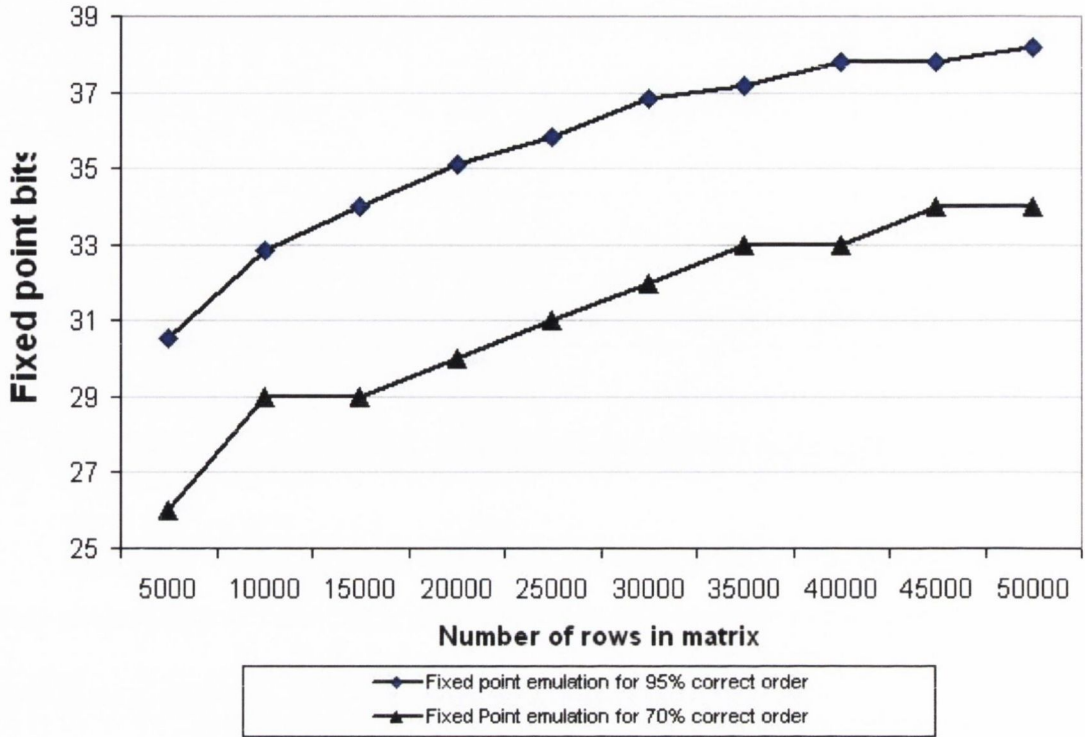


Figure 6.21 matrix size Vs. fixed point bits needed

The matrices used in these tests are relatively small when compared with IA matrices. Fixed point emulation is slow and full size IA matrices are hard to obtain, so an alternative approach was taken. However, the PageRank vector is an example of a Zipfian distribution [41, 119] and the Zipfian distribution can be used to extrapolate the results to a full scale system.

6.6.2 Zipfian distributions

The Zipfian distribution can be used to estimate the value of the components in a power law series. The PageRank vector follows a power law [41]. Figure 6.22 shows the values of an ordered PageRank vector plotted with a Zipfian distribution.

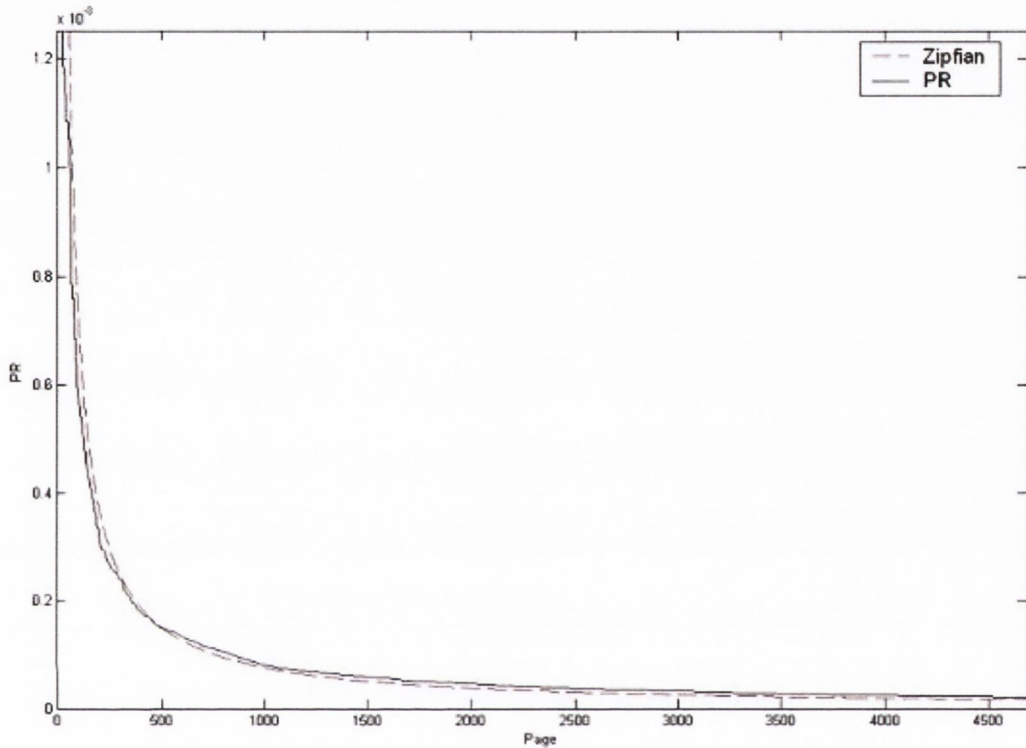


Figure 6.22 PageRank Vector following the power law

In the case of PR, the Zipfian distribution gives an approximate PageRank value for any page of rank k in the system containing N pages. A page of rank k refers to the page with the k^{th} largest PageRank value. The equation to calculate the Zipfian approximation of a page of rank k is shown in equation 26.

$$Z = \frac{1/k^s}{\sum_{n=1}^N 1/n^s} \quad (26)$$

The s factor is a customisation factor which is close to 1. This customisation factor can be used to fit the Zipfian distribution results to the results obtained from the fixed point emulation as shown in Figure 6.23. The Zipfian estimate follows the 95% and 70% correctly ordered fixed point emulation test very accurately when s is set to 1.33 and 0.64 respectively. The Zipfian estimate is obtained by calculating a Zipfian estimate of the PageRank vector of a system of size N . The two smallest values of the

Zipfian estimate are taken and the difference is calculated. The number of bits needed to represent this difference is returned as the Zipfian estimate of precision needed.

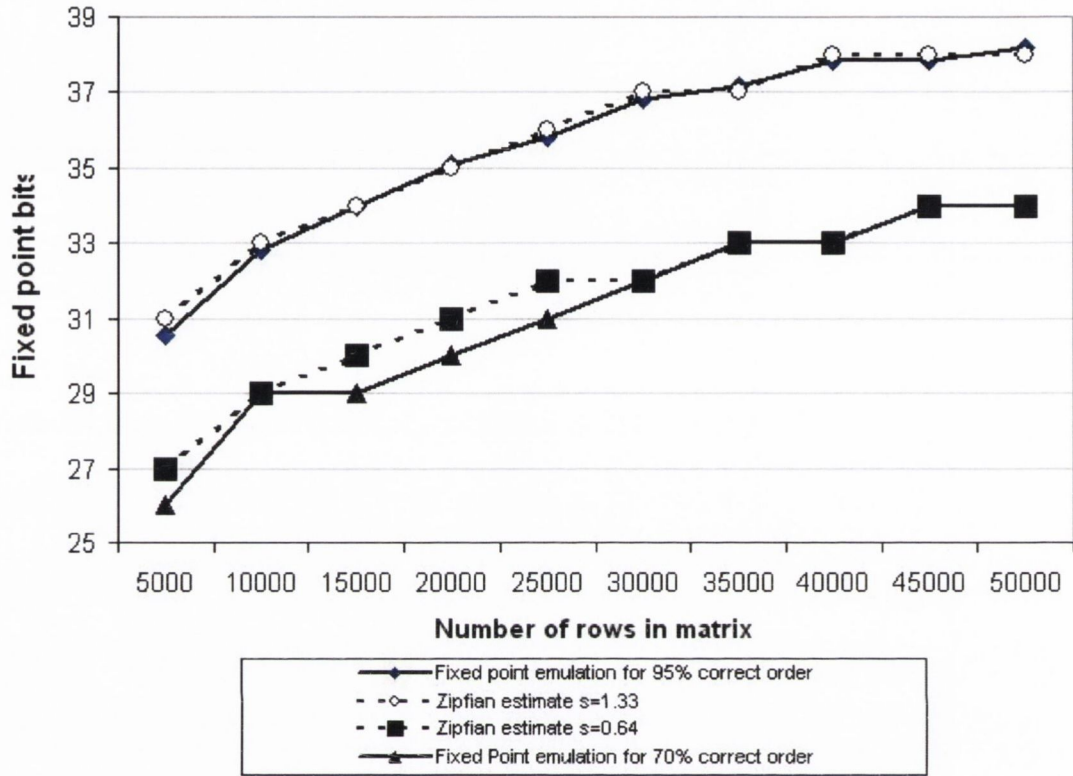


Figure 6.23 Fitting Zipfian distributions to emulation graphs

The Zipfian distribution can be used to estimate the precision needed for a PageRank vector of any size, once an appropriate s value has been found to match the Zipfian estimate to the fixed point emulation precision estimate. Setting $s=1.33$ fits the Zipfian curve to the 95% correctly ordered fixed point emulation graph. Thus, in Figure 6.24, the Zipfian estimate of fixed point precision needed is extrapolated for systems between 5 billion and 1 trillion pages.

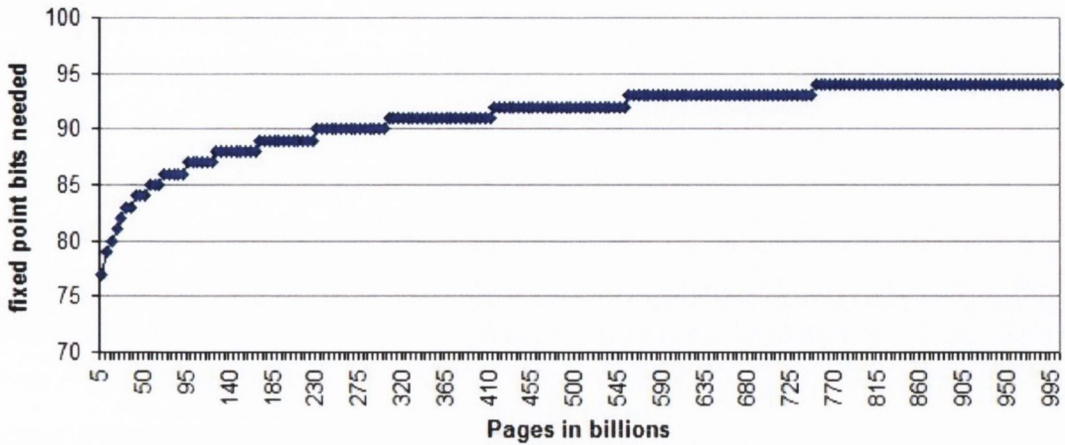


Figure 6.24 Estimating fixed point precision needed for larger systems with Zipfian distribution

The Internet currently crawled by the Google spiders contains over 1 trillion pages [6]. From Figure 6.24, 94 bit fixed point precision is required to represent this collection accurately. It is possible that lower precision would suffice as only a tiny fraction of these pages will be returned at any one time and so differentiating between small PageRank scores may not be necessary. In this section it has been shown that fixed point arithmetic can be used to calculate the PageRank algorithm. In the next section the performance benefits of migrating to fixed point arithmetic are explored.

6.7 Fixed Point Performance on Virtex-II

In the preceding section the possibility of reducing arithmetic precision as a means to increasing performance was discussed. Lower precision arithmetic units can be implemented with shorter pipelines than the floating point equivalent. A short pipeline adder should increase performance by removing the NOPs associated with RAW hazard avoidance. The Zipfian distribution data in Figure 6.24 shows that it is not practical to use fixed point arithmetic for an ever increasing size of matrix. Using fixed point arithmetic in large system would increase the memory storage requirements and memory bandwidth requirements as a system of 1 trillion pages needs 94 bits of precision. This is 30 bits more than is required by floating point arithmetic. If the Internet size increase further larger number systems would need to be used. However, for smaller subsets of the IA matrix it might be a viable alternative to floating point arithmetic.

In this section a number of experiments on the FPGA architecture's performance are presented assuming that a 64-bit fixed point number can be used to represent the data. The 64-bit fixed point word was chosen for a number of reasons. Firstly, and most importantly it is more than large enough to represent the test matrices adequately. Tests in section 6.6.1 indicate that any fixed point number with more than 38-bits of data should be large enough to represent this data set. Secondly, the 64-bit fixed point arithmetic unit can be easily substituted for the double precision units in the modular FPGA architecture design. This saves a complex and time consuming redesign of the system. Finally, since it is not practical to implement the 94 fixed point arithmetic unit needed to represent the one trillion page IA matrix, the fixed point solution can only be aimed at smaller collections of data like that in use at a business or home setting. The 64-bit fixed point adder used in these experiments is a single cycle adder, which will give the maximum possible performance boost to the FPGA based PageRank systems.

The fixed point results are presented in three sections. Firstly, in section 6.7.1 a fixed point version of the SCAR system is presented. Secondly, the fixed point version of the PageRank HW is discussed in section 6.7.2. Both of these architectures will be compared with their floating point equivalent. Finally, this section will conclude with a comparison of the performance of fixed point architectures and the GPP.

6.7.1 Fixed Point SCAR implementation

The dual MAC SCAR system performed about 20% better than the single MAC SCAR unit when using floating point arithmetic to calculate the PageRank vector. The dual MAC SCAR's stream also contains about 5% more NOPs due to its longer adder latency than the single MAC SCAR. Reducing the adder latency to a single cycle should significantly decrease the number of NOPs in the data stream. Thus, the stream with the most NOPs should benefit most from the migration to the single cycle fixed point adder. For this reason the dual MAC SCAR was used in these tests instead of the single MAC SCAR unit. The floating point arithmetic units in the dual SCAR system were replaced with single cycle fixed point adders and multipliers. The delay registers on the address signals to the Y-buffer were also removed to ensure data coherency. The net result of this change was a reduction in the number of slices occupied by the dual MAC SCAR unit; see Table 6.11. The fixed point

arithmetic units are a good deal smaller and less complicated than the floating point units they replaced.

Table 6.11 FPGA utilisation for fixed point Dual MAC SCAR on Virtex-II

Logic name.	Utilisation (Available)	% used
Slice Flip-Flops	42280 (67584)	62%
4 input LUTs	40,119 (67584)	59%
Multipliers	94 (144)	65%
Block RAM	134 (144)	93%
Gclk	8 (16)	50%
Occupied slices	33179 (33392)	93%

The fixed point arithmetic units reduced the NOPs in the dual MAC SCAR stream to an average of less than 2%. That is a reduction of over 10% from the average of the floating point version. The remaining NOPs are caused by the reordering scheme. The current version of the reordering scheme requires two NOPs to be placed after a block update because two cycles are required to stop the data streaming in from memory. Having two NOPs directly after the block update ensures that no data is received before the new X-buffer vector fragment is read in from memory.

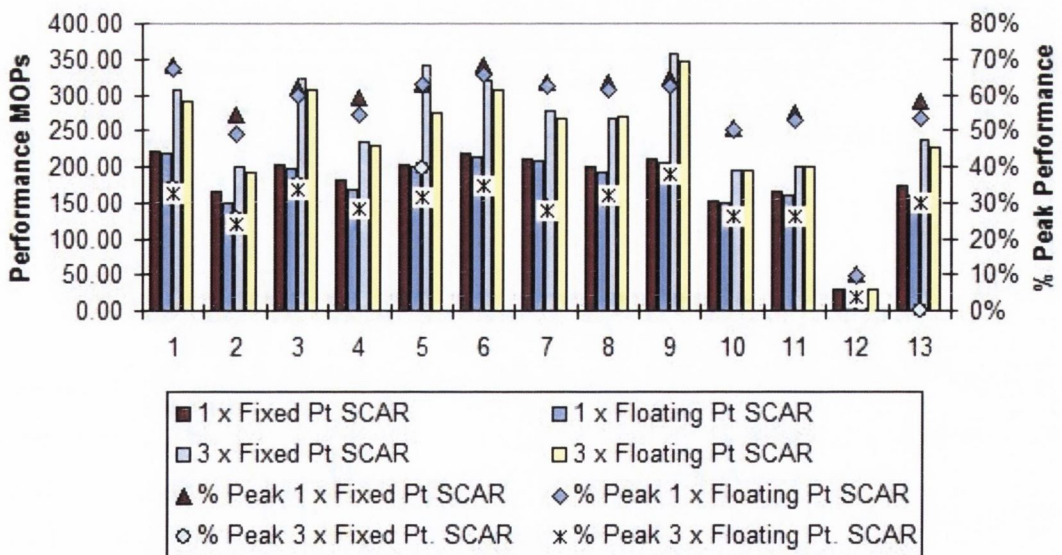


Figure 6.25 Performance results for the fixed point dual MAC SCAR system on Virtex-II.

The performance results of the fixed-point dual MAC SCAR are in Figure 6.25, together with the performance results of the floating point dual MAC SCAR. Figure 6.25 also shows the % peak performance of each of the systems achieved. It is clear

from Figure 6.25 that removing the NOPs in the stream has indeed increased performance on every single matrix. Unfortunately, though, the increase is very moderate. The average increase is a mere 3%.

The 3% increase is for the overall PageRank calculation and not just the SMVM. The system still must contend with many of the issues discussed earlier, namely, bus contention on the shared vector bus and the vector-unit bottle neck, which will be discussed later in section 7.3.

6.7.2 Fixed Point PageRank HW implementation

The second fixed point system implemented was the dual PageRank HW system with the larger 2048 line X-buffers. Once again, as was the case in the floating point implementation, there is a trade off between the larger X-buffers and having the third pattern adder unit on the FPGA. The floating point units are replaced with single cycle 64 bit fixed point arithmetic units and details of the FPGA-utilisation for the system is given in Table 6.12.

Table 6.12 FPGA utilisation for fixed point Dual Path PageRank HW with 2048 line X-buffers on Virtex-II

Logic name.	Utilisation (Available)	% used
Slice Flip-Flops	32474 (67584)	48%
4 input LUTs	33,954 (67584)	50%
Multipliers	3 (144)	2%
Block RAM	134 (144)	93%
Gclk	8 (16)	50%
Occupied slices	27,705 (33392)	81%

The migration from floating to fixed point arithmetic has significantly decreased the number of occupied slices. In Table 6.12 there is a 10% reduction in the number of occupied slices when compared with the equivalent floating point system in Table 6.7. The Block RAM usage remains unchanged and is the limiting factor in this design. If more Block RAM were available a third Pattern adder could be implemented on the chip. The number of multipliers is greatly reduced as the system only contains a single 64 bit fixed point multiplier. The system can still be clocked at a maximum clock rate of 97 MHz. The clock rate did not increase because the longest path is in the memory controller and so remained unchanged by migrating to

fixed point. The results of the benchmarks on this system are shown in Figure 6.26 together with the corresponding results of the equivalent floating point system.

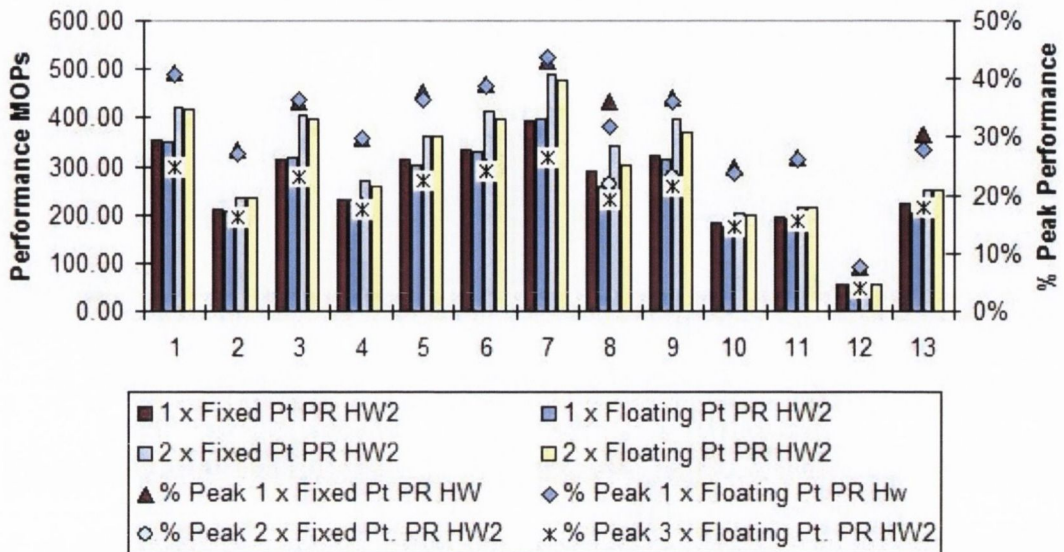


Figure 6.26 Performance results for fixed point dual path PageRank HW with 2048 line X-buffer on Virtex-II

The fixed point system performs on average about 3% better than the equivalent floating point system. The number of NOPs in the system is greatly reduced to a mere 1.5% of the overall stream length. As was the case with the SCAR system the overall speedup is very modest and this can be attributed to the Vector Unit bottle neck, bus contention and the very sparse and scattered structure of the IA matrices.

6.7.3 Fixed Point Architecture compared

The fixed point implementation of the dual MAC SCAR and the dual Path PageRank HW system did increase the performance of the system when compared with their floating point equivalents. Using fixed point arithmetic decreased the number of slices needed to implement both systems, due to their reduced complexity. However, this reduction in complexity did not lead to any significant increase in achievable clock rate or significant increase in performance. The fixed point versions of the dual MAC SCAR and PageRank HW decreased the number of NOPs in the system by 10% and 15% respectively. The increase in performance of both architectures was about 3% over that of the floating point equivalent. This disparity between the number of NOPs removed and the increase in performance can be attributed to a

number of issues that have been discussed earlier, namely, the shared bus contention, the very sparse and scattered structure of the IA matrices and the single path vector unit. Figure 6.27 shows the performance of both the fixed and floating point versions of dual MAC SCAR and the Dual Path PageRank HW as a percentage of GPP performance.

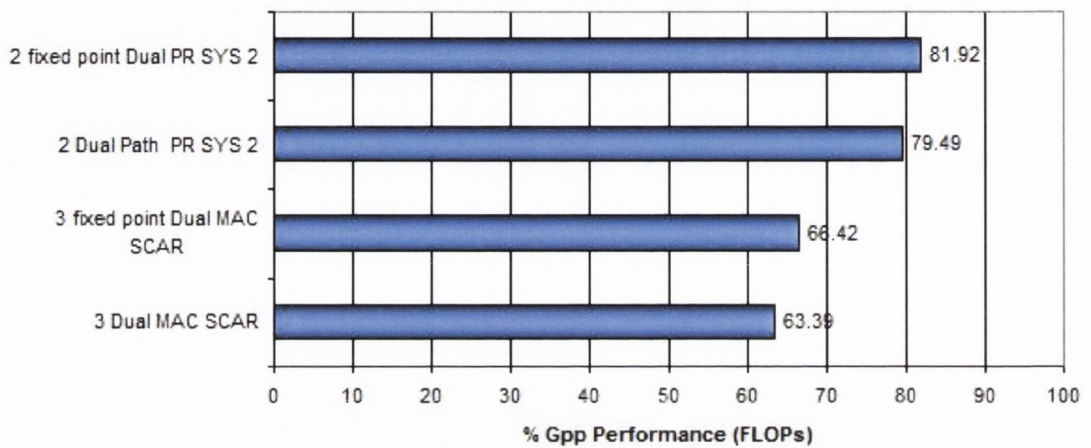


Figure 6.27 Fixed and Floating point performance as a percentage of GPP performance

Decreasing the precision in this case has done little to increase the performance of the overall PageRank algorithm. This poor increase in performance together with the impracticality of using fixed point arithmetic in a full scale system indicates that fixed point arithmetic is of little use for increasing the performance of the PageRank algorithm. Significant improvement can be made with this system only if the Vector Unit clock speed is increased and memory bandwidth is increased. Migrating the design to a newer FPGA family would increase the achievable clock rate and allow the design to connect to newer versions of SDRAM. In the next section the experiments with the Virtex-5 FPGA are discussed.

6.8 Floating Point Performance on Virtex-5

The Virtex-5 FPGA is the newest member of the Xilinx Virtex family available⁹. It differs from previous generations in a number of ways which are discussed in section 2.3.2. One of the major differences between the Virtex-5 FPGA and the Virtex-II FPGA is that it can support the higher speed DDR-2 and DDR-3 SDRAM modules. The Virtex-5 can also achieve a much higher clock rate than that of the Virtex-II. As

⁹ Virtex-6 was introduced at time of writing

discussed earlier the Virtex-5 FPGA was not used in these benchmarks because a suitable development board could not be sourced. Instead the design was implemented on the Virtex-II FPGA and a clock cycle counter was used to measure the time taken to calculate the PageRank algorithm. The system design was then reconfigured and targeted at the Virtex-5 device and using the post map clocking information of the Virtex-5 system, the results obtained from the Virtex-II hardware experiments were extrapolated to give the projected performance of the Virtex-5 device.

The system was targeted at Virtex-5 –xcv5lx155 device with a speed grade of -3. This was the smallest device that the whole system could fit on. The Virtex-5 devices smaller than xcv5lx155, did not have enough DSP48E units in particular. Table 6.13 compares the Virtex-II v6000 device to the xcv5lx155 FPGA.

Table 6.13 Resources of the Virtex-II v6000 FPGA and Virtex-5 lx155 compared

Part Name.	Slices	Mult/DSP	BRAM	User I/O
Virtex-II v6000	33,792	144	1,056Kb	1,104
Virtex-5 lx155	24,320 ¹⁰	128	6,912Kb	800

The Virtex-5 device has over six times more BRAM than the Virtex-II device. Ideally this BRAM could be used to further increase the X and Y buffers and thus reduce the overhead associated with reading and writing the X and Y buffer respectively to/from memory.

Three versions of the hardware were ported to the Virtex-5 device. These were the single MAC SCAR unit, the Dual MAC SCAR unit and the Dual Path PageRank HW system with the larger cache. All three systems achieved a maximum clock rate of about 250MHz. Since the system is using DDR-2 667, a clock rate of 222MHz is required to fully utilise the memory bandwidth when 192 bits (two 96 bit words) are used in every clock cycle; see section 2.4. The single MAC SCAR system does not use the full memory bandwidth and so it can be clocked at 250MHz. The dual MAC SCAR and dual path PageRank HW do utilise the full memory bandwidth when running at 222MHz and so this clock speed was used when projecting results for these architectures. Changing the clock speed of the architectures will not increase the efficiency or adder utilisation of the architecture and so they remain the same as

¹⁰ Virtex-5 slices differ from Virtex-II slices, see section 2.3.2

indicated for the Virtex-II system. Figure 6.28 shows the performance data of the single MAC SCAR system on the Virtex-5 FPGA with one, two and three SMVM units respectively. The GPP results are also given for comparison. As discussed earlier the single MAC SCAR unit does not fully utilise the available memory bandwidth of the DDR2-667 and so it is clocked at the maximum speed achievable for the architecture of 250 MHz.

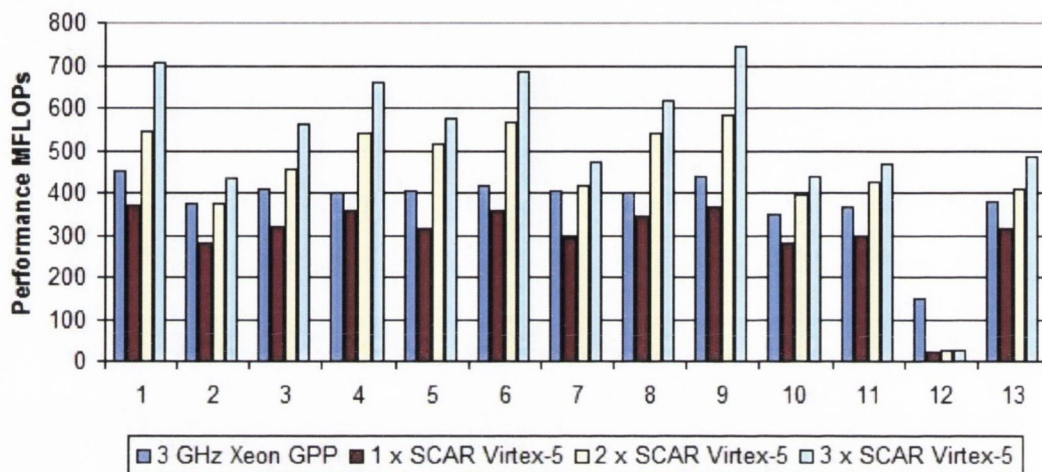


Figure 6.28 Performance results for single MAC SCAR system on Virtex-5 with 250MHz clock

The single MAC SCAR architecture has an average performance of 300 MFLOPs, 445 MFLOPs and 528 MFLOPs for the PageRank algorithm being calculated on one, two and three SMVM unit respectively. The system with two and three SMVM units outperforms the GPP by an average of 17% and 40% respectively. As discussed earlier the single MAC SCAR does not utilise the full memory bandwidth available from the SDRAM. The dual MAC SCAR was designed to fully utilise this bandwidth and so it too was implemented on the Virtex-5 platform. The clock rate of the dual MAC SCAR system was set at 222 MHz as that is the clock speed needed to match the computational and memory bandwidths. Figure 6.29 shows the performance figures of one, two and three dual MAC SCAR units running on the Virtex-5 FPGA with DDR-667 SDRAM.

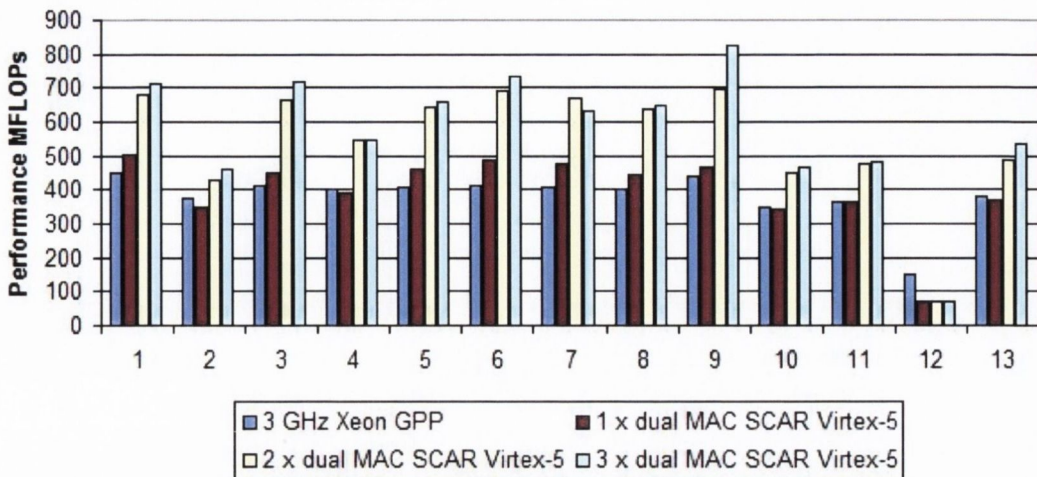


Figure 6.29 Performance results for dual MAC SCAR system on Virtex-5 with 222MHz clock

The dual MAC SCAR system on Virtex-5 increases the average performance figures to 400 MFLOPs, 550 MFLOPs and 575 MFLOPs for one, two and three SMVM units respectively. As was the case with the Virtex-II performance results, the dual MAC SCAR performs about 10% - 15% better than the single MAC SCAR system. A single dual MAC SCAR performs on average 5% better than the Intel Xeon Woodcrest processor. Three dual MAC SCAR units can perform about 1.5 times the speed of the Xeon Woodcrest. This is despite having a clock rate about 13 times slower than that of the Xeon Woodcrest.

The third architecture to show promise in the Virtex-II results was the specialised dual path PageRank HW. Two versions of this HW architecture, with different sized caches, were developed on the Virtex-II platform. The version with the larger cache was implemented on Virtex-5. Increasing the cache size on the Virtex-II device meant that only two SMVM units in the form of pattern adders could be implemented. This was due to a shortage of BRAM on the device. The Virtex-5 FPGA contains enough BRAM to implement all three SMVM units. However, since these Virtex-5 results are extrapolated from the Virtex-2 results it is not possible to include the third unit in these results. The performance results for one and two dual PageRank HW units on Virtex-5 are presented in Figure 6.30, together with the performance of the Intel Woodcrest processor for comparison.

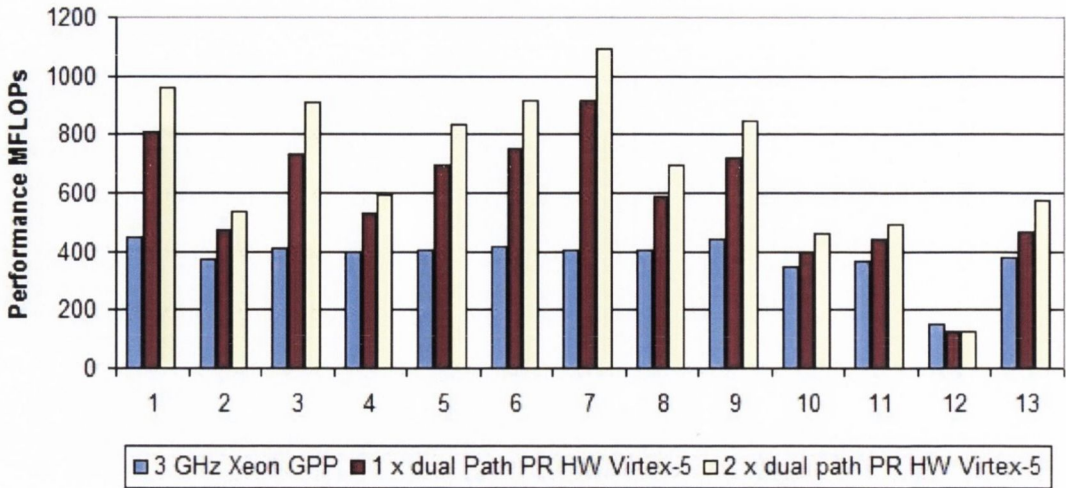


Figure 6.30 Performance results for dual path PageRank HW system on Virtex-5 with 2048 line cache and 222MHz clock

The dual path PageRank HW unit outperforms the Intel Woodcrest on every matrix with the exception of matrix 12 which has caused problems for all the FPGA based solvers and has a much lower performance level than the other matrices on the GPP. Matrix 12 as discussed earlier has only 4 NZE per column which is well below the average of 10 expected in an IA matrix. Matrix 12 also appears to show no form of grouping which is uncommon in IA matrices. The average performance of the dual path PageRank HW on Virtex-5 is 588 MFLOPs and 695 MFLOPs for a system containing one and two dual path pattern adder units respectively. The average performance is about 20% better than that of the dual MAC SCAR system on Virtex-5 and 80% better than the Intel Woodcrest computing the PageRank algorithm on the test matrices. Figure 6.31 is a summary of the performance achieved by the three FPGA based architectures implemented on Virtex-5 as a percentage of the Intel Woodcrest performance.

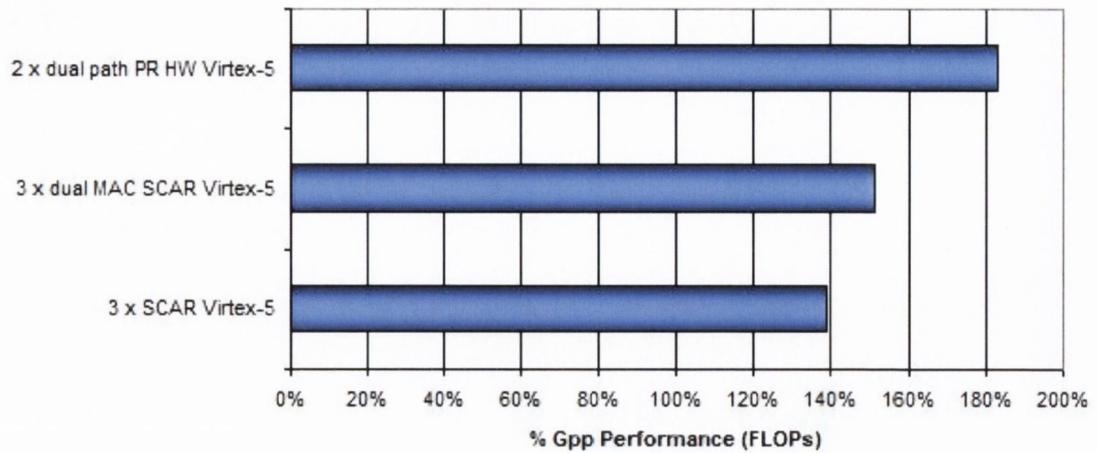


Figure 6.31 Summary of performance of FPGA architectures on Virtex-5 as a % of GPP performance.

It is clear from Figure 6.31 that all three of the FPGA based architectures outperforms the GPP when implemented on Virtex-5. This is due to the increased memory bandwidth achieved by upgrading from DDR-266 to DDR2-667. The clock rate of the FPGA based system has also more than doubled which means that the Vector Unit calculates its answer quicker and the system can utilise the extra memory bandwidth. The result is that the FPGA, which now uses the same memory as the GPP, can perform the PageRank algorithm about 1.8 times quicker than the Intel Woodcrest, despite having a clock rate 13 times slower. This is achieved by parallelising the PageRank algorithm and increasing the achievable memory bandwidth by implementing multiple independent banks of memory.

6.9 Summary

This chapter started with a look at the test matrices being used to benchmark the test architectures. They are real IA matrices obtained from crawls by the webgraph project. These matrices are compared with the more familiar FE matrices. IA matrices are often sparser and less organised than FE matrices. Unlike FE matrices, IA matrices are never symmetric. The base line performance for these tests was calculated by running the PageRank algorithm on a high end general purpose processor, namely the Intel Xeon Woodcrest. The Intel Xeon Woodcrest achieved a sustained performance of about 400 MFLOPs which is only about 4% of its theoretical computational bandwidth. The calculation is limited by the memory

bandwidth. Even with its sophisticated cache structure, the Intel Xeon Woodcrest could only utilise about 22% of its FSB memory bandwidth.

With the discussion of the Intel Xeon Woodcrest results complete, a number of tests were run on two SMVM architectures designed for FE matrices, called SPAR and SCAR. The SPAR architecture is a column based approach to SMVM. It achieved a sustained performance of about 110 MFLOP with three SMVM units in parallel. The SPAR had an adder utilisation of about 35% which is much greater than that of the Woodcrest which had 4% adder utilisation. The poor FLOPS performance was caused by a low clock rate, RAW hazard avoidance in the adder pipeline and slow cache line replacement on misses. The SCAR aimed to alleviate the problems inherent in the SPAR architecture by tiling the matrix and interleaving rows to avoid RAW hazards. This works very well on FE matrices but is of limited use in IA matrices. The SCAR achieved an average of 208 MFLOPs for the test matrices which is about 41% adder utilisation. The SPAR and SCAR do not utilise the full memory bandwidth available to the system and so a third system was implemented to match the computational bandwidth with the memory bandwidth. This was done by adding an extra MAC unit into the SCAR system to create the dual MAC SCAR; see Section 5.5.3.

The dual MAC SCAR was a large system occupying 98% of the slices on the Virtex-II 6000 FPGA. The dual MAC SCAR increased the performance of the three SCAR system to 256 MFLOPs. However, there was a drop in adder utilisation to 28% because the dual MAC system effectively increased the adder latency and thus the possibility of RAW hazards occurring in the adder pipeline.

The SPAR and SCAR architecture are generic SMVM hardware units. The third and final hardware architecture discussed is an architecture designed especially for the PageRank algorithm and thus is referred to as the PageRank HW system. This system takes advantage of the structure of the IA matrix, where all entries in a column are the same. This fact is used to break the SMVM operation into two steps: a dense element by element multiplication and a pattern addition. This allows the multipliers to be removed from the SMVM unit as well as further compression of the NZE datawords. The PageRank HW architecture consists of two buffers and an adder tree which sums up to 6 NZE per clock cycle. Two flavours of this architecture were implemented. The first had three dual path pattern adder units and 1024 entries in its

X-buffer. The second had two dual path pattern adder units and 2048 entries in its X-buffer. The second system with the larger X-buffer performed better with an average performance of 255 MFLOPS and 302 MFLOPs for one and two pattern adder units respectively. The adder utilisation did fall in this system to 19% but it still performed better than the other FPGA based architectures due to its high level of parallelisation. The dual path PageRank HW system achieved about 80% the performance sustainable on the Intel Xeon Woodcrest. This was despite having a clock 30 times slower and SDRAM that had a memory bandwidth of less than half of that of the DDR-2 667 used by the Woodcrest. The dual MAC SCAR, single MAC SCAR and SPAR achieved approximately 64%, 55% and 29% of the performance achieved by the Woodcrest respectively.

The Chapter continued with a discussion of the performance limitations of the system. These are shared bus contention, load balancing and NOP frequency. Firstly, in a test designed to show the effects of shared bus contention the performance of two dual path PageRank HW units calculated using a shared bus and then using independent buses. This test showed that if bus contention was removed two dual path PageRank HW units would perform 70% quicker than a single unit. This is in comparison to the 30% increase in performance achieved with the shared bus system. However, when the memory copy needed to duplicate the X-vector is taken into account, the performance of the overall system drops to only 10% faster than the dual path PageRank HW system with the shared bus.

Secondly, the load balancing of the data stream to the SMVM units was investigated. The stream lengths for each of the SMVM units were compared. In FE matrices it was found that the average difference in stream length was 3% with a maximum difference in stream length of 20%. However, the IA matrix streams differed by an average of 10% with a maximum difference of over 30% in one of the matrices. A modification to the load balancing algorithm decreased this to less than 1% difference in stream lengths between the parallel SMVM units.

Thirdly, the effect of NOPS being introduced into the stream as a method of RAW hazard prevention was discussed. The SCAR was designed to work with a simple round robin reordering scheme. This proved unsatisfactory when used with IA matrices as the stream often contained many NOPS. Another scheme called opportunistic reordering was created which greatly reduced the number of NOPS. In

an attempt to reduce the NOP count further the RCM matrix reordering scheme was applied to the test matrices. Although using RCM did increase performance slightly it did not have enough of an effect to justify the time take to reorder the matrix. A direct correlation between NOP count and adder latency was established. Since floating-point adders are quite complicated it is very hard to reduce their pipeline depth. One way of reducing pipeline depth is to reduce precision and so an investigation into fixed point arithmetic was carried out.

This investigation into fixed point arithmetic started with fixed point emulation of the PageRank algorithm. The number of bits needed to calculate the PageRank vector for the test matrices was determined to be 38 bits. However, a full scale system would take many more. Since it would take too long to emulate fixed point arithmetic on a full scale system, a Zipfian distribution curve was used to estimate the number of bits needed to represent a full scale system of 1 trillion pages in fixed point. It was estimated that 94 bits of fixed point precision would be needed. A 94 bit number is quite impractical and would increase memory storage requirements. However, fixed point arithmetic versions of the SCAR and PageRank HW units were produced using a 64 bit fixed point number to investigate the usefulness of fixed point in smaller systems.

The fixed-point versions of the dual MAC SCAR and the dual path PageRank HW systems show a huge reduction in NOPs from 12% and 16% NOPs to 1% and 1.5% respectively. However, the performance increase is very modest at a mere 3% since the SMVM operation is only part of the calculation. The slow clock rate and single path Vector Unit are now one of the critical paths in the hardware.

The final section of this chapter is a look at how these architectures would perform if implemented on newer generations of FPGA. Implementing the system on a Virtex-5 FPGA allows the faster DDR-2 667 SDRAM to replace the DDR-266 SDRAM used on the development platform. The Virtex-5 also increases the clock rate to a maximum of 250 MHz. The single MAC SCAR, dual MAC SCAR and dual path PageRank HW systems were implemented and results extrapolated from the Virtex-II data and the new post MAP¹¹ clock rate. All three architectures perform better than the Intel Woodcrest. The single MAC SCAR and dual MAC SCAR perform about

¹¹ Process used to map the circuit to FPGA architecture.

40% and 50% faster than the Intel Woodcrest. The dual path PageRank HW system is the fastest of all the tested architectures and it can calculate the PageRank vector for the test matrices at almost twice the speed of the Intel Woodcrest, despite having a clock rate thirteen times slower.

The results presented in this chapter are the first attempts at speeding up the PageRank algorithm using an FPGA to the best of the author's knowledge. It is clear that it is possible with modern FPGAs and SDRAM. The viability of the FPGA approach will be discussed in the next section with details of where the author thinks more work could be done to improve the result further.

Chapter Seven

7 Discussion and Conclusions

7.1 Introduction

The modern Internet, with over 1 trillion pages [6], has become the global network imagined by Licklider in 1962 [1]. One of the major issues faced by Internet engineers is the ranking of pages returned to user queries. This thesis attempts to investigate FPGA based hardware for these ranking algorithms. In Chapter 3 this ranking was shown to be simply a linear algebra problem. Two ranking algorithms were discussed in detail. Firstly, the HITS algorithm, used by the Ask search engine was discussed. Secondly, Google's PageRank algorithm was presented. PageRank is the most popular search algorithm used on the Internet today. The PageRank algorithm is an iterative algorithm which consists of a single SMVM and nine vector operations per iteration. Brin and Page, the creators of PageRank, claimed that about 50 iterations of the PageRank algorithm are needed to achieve the correct PageRank order [49]. Brin and Page used commodity PCs to calculate the PageRank algorithm. Research presented in Chapter 4 showed that the GPP performs quite poorly when calculating SMVM. It often achieves only about 3-4% of its peak computational bandwidth on IA matrices. In Chapter 5 and Chapter 6 a number of FPGA solutions for the PageRank algorithm were proposed and discussed. In the remainder of this chapter these results will be discussed. The Chapter concludes with a look at possible future work that might be beneficial to this project.

7.2 Contribution of this Thesis

In this thesis the viability of using an FPGA based system to solve the PageRank Internet ranking algorithm was investigated. To the best of the author's knowledge this work is the first time this possibility has been investigated. The major findings of this thesis can be broken down into four broad categories:

- Precision considerations for the PageRank algorithm
- Implementation of FPGA hardware suitable for execution the PageRank algorithm
- Implementation and benchmarking of two existing SMVM architectures
- Design, Implementation and benchmarking of new FPGA hardware specially for the PageRank algorithm

7.2.1 Precision of the PageRank calculation

In Section 6.6, a system for estimating the fixed point precision needed for the PageRank algorithm was proposed, through using a fixed point emulator for small IA matrices. A Zipfian distribution was fitted to the fixed-point emulation results and the results were extrapolated to full size IA matrices. The use of fixed point arithmetic was investigated in an attempt to remove NOPs associated with RAW hazards in the adder pipeline. Fixed-point adders often have significantly shortened adder pipelines and thus should reduce the number of NOPs. Consequently the performance of the PageRank calculation should be increased by this process.

Use of the emulation-extrapolation method showed that approximately 95 bits of fixed point precision would be needed to ensure that 95% of the pages in the PageRank vector for a system of 1 trillion pages were in the correct order. The fixed point solution, therefore, would increase the storage requirement of the PageRank algorithm as each NZE in the fixed point system would require 31 bits more memory than the NZE in the 64 bit floating point system. This would be an increase in memory requirements of approximately 50%. Larger data buses would be required by the FPGA, which could increase the FPGA resource usage significantly. It was therefore decided that a fixed-point arithmetic system for a full scale PageRank calculation was impractical. Instead, a fixed point solution may be viable for smaller

corporate search facilities and so a 64 bit fixed point system was implemented. The results of this system were presented in Section 6.7 and they will be discussed in Section 7.2.3.

7.2.2 PageRank in Hardware

In Chapter 5, the design and implementation of a hardware platform capable of computing the PageRank algorithm was described. The hardware needed to be capable of dealing with the very large and sparse matrices associated with PageRank algorithm. It was decided that the architecture designed for the FIAMMA project [19] was a suitable platform as the only limitation on matrix size was the size of the SDRAM used by the system.

The first step in implementing the PageRank algorithm in hardware was to identify the operations carried out by the PageRank algorithm. Once identified, a number of hardware units were developed to carry out these operations. Software libraries were also written to control the hardware units. These libraries allowed the MicroBlaze to communicate with hardware units and thus control their operation. In this way, the PageRank algorithm written in C code and executed on the MicroBlaze was computed by the custom hardware.

A number of variants of the hardware PageRank solver were implemented. The first hardware system implemented was a double precision floating point system running on a Virtex-II V6000 FPGA. This system contained three SMVM units, an all-purpose vector unit, Microblaze controller, PC interface and four WB buses and memory controllers for connection to four independent banks of SDRAM. The clock rate of the implemented design was approximately 100 MHz. The clock rate varied with the SMVM unit implemented. Three different SMVM units were implemented and tested on this system. Two of these SMVM units were pre-existing, state of the art SMVM units. The third was a special SMVM unit designed for use with the PageRank algorithm and was designed as part of this work. These units will be discussed further in the following two sections.

The second hardware variant for the PageRank algorithm was a 64-bit fixed point version on the Virtex-II V6000 FPGA. In this system, all the floating point arithmetic units were replaced with fixed point arithmetic units. This reduced the size of the

system but did not increase the clock rate in any of the systems implemented because the longest path was in the memory controller which remained unchanged by the migration to fixed point arithmetic. Two SMVM units were implemented using fixed point arithmetic. These were the specialised PageRank HW and the generic SMVM unit (SCAR) which had a performed better than the SPAR unit in the double precision tests.

The final variant implemented was a double precision floating point system on the Virtex-5 lx155 FPGA. Once again, the SCAR and PageRank HW were implemented on this platform. The clock rate of the Virtex-5 systems increased to a maximum of 250 MHz. This last system also used DDR2-667 memory which is over two times faster than the DDR-266 used in the Virtex-II designs.

7.2.3 Implementation and Benchmarking of two Generic Linear Algebra Systems

The two generic SMVM systems were the SPAR and SCAR system [17, 115]. The SPAR architecture processes the matrix data in a column major format. The SPAR was first proposed in 1995. However, to the best of the author's knowledge, it was never built. This implementation of SPAR, therefore is the first full scale system to contain a SPAR unit and the first implementation of the SPAR architecture on FPGA. The hardware solver system with three SPAR units occupied 84% of the 33392 slices available on the Virtex-II FPGA. Initial tests showed that the long columns of the IA matrices caused a huge number of Y-cache misses and so a modification was made to the SPAR architecture and data ordering. An X-buffer was implemented and added to the design to save on costly reads to SDRAM. The matrix was divided into bands before encoding using the SPAR data structure. Thus, the number of Y-cache misses was greatly reduced. Y-cache misses were very costly, as they required the data in the cache line be written out to SDRAM before the new cache line could be read in. The hardware system with one and three SPAR SMVM units achieved an average performance of 70 MFLOPS and 110 MFLOPs respectively. This was approximately 33% of the peak computational performance and only 30% of the performance achieved by the GPP against which it was benchmarked. It was clear from these results that the SPAR architecture would be of no use with IA matrices and so no further tests were run on the SPAR architecture.

The second SMVM architecture implemented and benchmarked was the SCAR architecture which was designed and built as part of the FIAMMA project for FE stiffness matrices. The SCAR architecture is block-row based. The hardware system for solving the PageRank algorithm with three SCAR units occupied 79% of the Virtex-II V6000 slices. It was, therefore, slightly smaller than the SPAR. The state machine in the SCAR was greatly simplified over the SPAR's state machine, because it moved the issue of dealing with RAW hazard detection to the software reordering system. The architecture calculating the PageRank vector with one, two and three SCARs achieved a performance of 126 MFLOPS, 180 MFLOPS and 208 MFLOPS respectively. This performance represented a 1.8x performance improvement over that of the SPAR. The SCAR achieved approximately 55% of the GPP performance with a clock rate of 96MHz which is over 30X slower than the GPP clock rate. The SCAR achieved approximately 42% of its theoretical computational bandwidth when using IA matrices. These utilisation figures of computational bandwidth were good enough to enable the SCAR to easily outperform the GPP when the SCAR architecture was migrated to the newer Virtex-5 FPGA with DDR-2 667. The SCAR on Virtex-5 actually performed approximately 40% better than the GPP despite having a much lower clock rate (250 MHz vs. 3 GHz).

The architecture with SCAR SMVM units performed at 40% of its peak computational peak bandwidth when calculating the PageRank algorithm. However, due to the low clock rate of the FPGA system, the computational bandwidth didn't match the available memory bandwidth. In order to eliminate this mismatch and to fully utilise the memory bandwidth, a second version of the SCAR system was implemented. It was referred to as the dual MAC SCAR as it has two MAC units internally. The WB buses in the system were doubled in size to carry two NZEs at a time to utilise the two MAC units in the dual MAC SCARs. The memory controller was adapted to include a two time domain. In previous versions of SCAR the memory was clocked at the same rate as the rest of the system. This was changed to allow the memory to be clocked at a different rate to the rest of the system. This meant that the memory could be clocked at full speed to fill a FIFO and the larger WB buses could utilise the data at the same rate. Since the dual MAC SCAR used two NZE on every clock cycle the system clock rate could be lower than the memory clock rate and still fully utilise the memory bandwidth. The results of the Virtex-II

system were an average performance of 176 MFLOPs, 247 MFLOPS and 259 MFLOPS for a hardware system containing one, two and three dual SCAR systems respectively. This system occupied 98% of the available FPGA slices making it the largest of the systems implemented. The dual SCAR system achieved approximately 62% of the GPP performance. However, the utilisation of the computational/memory bandwidth dropped to 28%. This was due to increased adder latency, bus contention and a calculation bottleneck in the Vector Unit (all of which will be discussed in Section 7.3). The dual SCAR system was also used to match computational bandwidth to memory bandwidth in the Virtex-5 system with the result that the dual SCAR system achieved a 1.5 times improvement on the GPP performance.

As discussed in 7.2.2, an investigation into fixed point performance was carried out using the dual MAC SCAR. The fixed point system reduced the size of the system by about 5%. The clock rate remained unaffected as the longest path was in the memory controller. Changing the design to fixed point should remove the 12% of the data stream that are NOPs, which were inserted to deal with RAW hazards. However, a speed-up of only 3% was achieved. This was due to the fact that RAW hazards only affected the SMVM part of the calculation and so the vector operations did not benefit from the removal of NOPs from the stream. The issue of shared bus contention was also a problem which will be discussed in Section 7.3.

7.2.4 Implementation and Benchmarking of specialised PageRank Hardware

The SCAR architecture did outperform the GPP with the IA matrices in the PageRank algorithm when implemented on Virtex-5. However, it was the third SMVM architecture that performed best. The third architecture was designed to take advantage of characteristics of the IA matrices. The PageRank algorithm requires that all values in the column of an IA matrix must be equal and sum to one. Thus there is no need to save the value of every NZE. Instead the matrix can be represented with a dense vector containing the NZE value, if any, for any given row and a pattern matrix. Using this format the SMVM can be achieved by performing a dense element by element vector multiplication and then by using the pattern matrix to add the various elements associated with a given column. This process was given

the name “pattern addition” as it removed the need for multipliers in the SMVM units.

Removing the need to store the value of the NZE reduced the number of bits needed to represent the NZE. The NZE could now be represented using two 16 bit addresses which gave its position in the matrix block. This reduction in memory requirements effectively tripled the memory bandwidth. In order to match computational bandwidth with available memory bandwidth each of the PageRank HW units must process six NZEs per clock cycle. This was done using an adder tree and the resulting architecture utilised 93% of the available slices on the Virtex-II V6000. The number of hardware multipliers used dropped from 65% to 8% when compared with the SCAR architecture. In order to fit three PageRank units on the FPGA the X and Y buffer size had to be reduced to 1024 vector entries, which was half the size of the buffers used by the SCAR architecture. Using adder trees allowed multiple NZE to be processed per clock cycle but this came at a cost of increasing the overall adder latency and thus the number of NOPS in the stream. Increasing the number NZE processed per clock cycle also increased the shared bus contention as the units processed blocks faster but reading in new X vector fragments and writing out Y vector fragments still took the same time as before thus leaving units waiting for use of the vector bus.

The hardware architecture with one, two and three PageRank HW units for pattern addition performed at 228 MFLOPS, 270 MFLOPs and 281 MFLOPS respectively. The poor performance increase when multiple PageRank HW units were added was due to the shared vector bus which was now saturated. In an attempt to alleviate this issue, one of the PageRank HW units was removed and the size of the X and Y buffers were increased to 2048. The results of this version of the architecture were better and the new PageRank HW system achieved 255 MFLOPS and 302 MFLOPS for the system with one and two PageRank HW units respectively. Although this was still well below the peak computational bandwidth, it meant that the FPGA system was performing at 80% that of the GPP despite having a clock rate over 30 times slower and memory that is clocked at less than half the speed of the DDR2-667 used by the GPP. This second version of the PageRank HW was then ported to the Virtex-5 FPGA using DDR2-667 and achieved a speed up of over 1.8 times that of the GPP. This result was the best result obtained from any of the architectures and it proved

that in order to achieve sizable speed up hardware designed specially for the PageRank algorithm and IA matrices was needed.

The PageRank HW was also implemented in a fixed point version, but like the SCAR fixed point version only very moderate performance increases were achievable.

7.3 Limitations and Future Work

Throughout the benchmarking work, discussed in the previous section, a number of limitations on the performance of the architecture were repeatedly mentioned. These were NOPs in the data stream, shared bus contention and algorithm bottle neck in the vector unit.

Firstly, the NOPs in the data stream were inserted to avoid RAW hazards and their frequency was dependent on matrix structure and the adder latency. Changing the structure of the matrix could be achieved using reordering algorithms but these can be very costly. In Section 6.5.3, RCM was used in an attempt to decrease the number of NOPs and thus increase performance, but it proved to be of little use with the test matrices used. If a suitable reordering scheme was found, an increase in performance of up to 12% and 16% could be achieved for the SCAR and PageRank HW systems respectively. The number of NOPs in the stream could also be reduced by the implementation of short pipeline adders. More research into this area, especially on the Virtex-5 and Virtex-6 platforms could help with optimising this architecture.

Secondly, shared bus contention proved to be a serious issue in this architecture when dealing with IA matrices. This was caused by sparse matrix blocks where almost 90% of the matrix blocks contained less than one element per row. In section 6.5.1 an experiment was carried out to estimate the performance increase that could be achieved by removing bus contention. This was done by making a copy of the X vector at the end of every calculation and allowing the SMVM units to access their own copy of the X vector. The second PageRank HW architecture with two SMVM units was used for this, as it had a free memory channel. When two independent copies of the X vector and Y vector were used, the system with two SMVM units performed 70% quicker than the single SMVM unit architecture. This increase, however, was reduced to 30% when the time taken to do the memory copy was taken

into account, showing that bus contention was a real issue in this system. The project could benefit from a more in-depth look into ways of reducing this issue.

Improvements may be achieved by increasing the efficiency of the vector bus. It may be possible to implement the memory copy required to have two separate vector buses as part of one of the other vector operations carried out in the Vector unit.

Currently a number of vector operations are carried out on the current estimate of the PageRank vector on each iteration. If a feature was added to the Vector unit to allow multiple output destinations to be requested for one of these operations the vector copy could be done as part of one of these other vector operations. This, however, would require a major change to the Vector Unit hardware. Currently, the shared vector bus is only 128 bits wide. This is significantly smaller than the matrix data bus which is 192 bits wide. This change alone would increase the buses transfer speed by 50%. However it would do nothing for the latencies in starting and ending a memory read or write to SDRAM. It is possible that some improvements could also be made in regard to this memory operation start-up latency if the SDRAM controllers and WB bus were redesigned to minimise start-up latencies or if pre-fetching memory data was implemented,

Finally, the Vector Unit has become a performance bottle neck in this system and any future version of the hardware should look into implementing a parallelised version of the Vector Unit. Currently, the Vector unit only has a single MAC pipeline and does not utilise the full memory bandwidth available. When the PageRank algorithm was profiled, the Vector unit was found to perform 24% of the operations needed to complete the algorithm. The remaining 76% of the operations were carried out by the SMVM unit. For this reason, this work focused on accelerating the SMVM operator. Table 7.1 shows the percentage of PageRank calculation time spent carrying out operations in the Vector Unit and SMVM unit for the different versions of the FPGA hardware.

Table 7.1 Time spend on operations carried out by the Vector unit and SMVM unit when calculating the PageRank algorithm. (figures in bold indicate vector unit limited architectures)

Hardware Version.	% time in Vector unit	% time in SMVM unit
Work Load	24%	76%
3x single MAC SCAR	27%	73%
3x dual MAC SCAR	33%	77%
3x PageRank HW 1	44%	56%
2x PageRank HW 2	47%	53%

Table 7.1 shows that as improvements are made to the SMVM performance, the percentage of the calculation spent calculating the vector operations increases. For optimal performance vector operation time and SMVM operation time should remain proportional to the work load being carried out by each unit. It is clear therefore that as the SMVM performance increases the Vector unit becomes a bottleneck.

Throughout all these tests, the vector operations took the same amount of time to complete. The increase in percentage of the calculation spent in the Vector unit was caused by an overall reduction in the time taken to complete the algorithm due to the increase in SMVM performance.

One method of increasing the performance of the Vector unit would be to parallelise the unit's operations. This could be done by adding a number of parallel MAC units with an adder tree to add the results together at the end. The adder tree may or may not be used depending on the operation. It would be needed in the dot product operation but would not be needed in the element-by-element multiplication operation, for example.

Implementing a parallelised Vector unit on the Virtex-II system would not be practical in its current state as the system utilises up to 98% of the device. The large Virtex-5 FPGA should easily have enough logic to accommodate a parallelised version of the Vector unit. The system could be parallelised to fully use the available memory bandwidth. The Virtex-5 system described in Section 6.8 includes 4 banks of DDR2-667 and has maximum clock rate of 222 MHz. Since the Vector unit uses 64 bits per clock cycle, three parallel MAC units would be needed to fully utilise the memory bandwidth. If this system was implemented the Vector unit performance would increase almost 3 times its current performance. This would give the whole PageRank algorithm an increase in performance of 18-30% depending on which

system the FPGA was running. This performance increase is sizable and would be well worth investigating in future versions of this hardware.

Another issue that should be addressed is the data word size of the PageRank hardware. Currently the system uses a 96 bit data word which consists of three 32 NZE. This could be reduced to a 64 bit data word if the extra bandwidth was needed. This is due to the fact that the system was designed to utilise three NZE from the same row of the matrix. Currently, the row value for each of the NZE was encoded in the PageRank hardware dataword. Since they all came from the same row the row address only needed to be included once. This was not implemented as part of this architecture as the clock rate of the system was unable to increase further to utilise the extra memory bandwidth that would be made available and so no performance increase would have been apparent. However, in future implementations, if memory bandwidth rather than clock rate was to become the major bottleneck this could increase the memory bandwidth by 33%.

A final point that is worth mentioning is the power considerations of the project. Currently, the issue of power consumption of the FPGA versus the GPP has not been considered. The GPP has a power rating of 80Watts. Since the GPP can run at about 400 MFLOPS for the PR algorithm the performance per watt rating of the GPP is approximately 5 MFLOPS/W. Calculating a similar performance per watt figure is a non-trivial task for the FPGA as it is affected by clock rate, logic utilisation and temperature. The FPGA however is counted a low-power device and so should have a much smaller power requirement than the GPP and thus make it more cost-effective to use the FPGA for the PageRank calculation. A studio of the power usage of this architecture would need to be carried out before the extent of this reduction in cost could be estimated.

This system is a first attempt at calculating the PageRank algorithm on FPGA and as such there are many areas where work could be carried out in the future, especially as faster FPGAs and memory modules become available. In the preceding paragraphs a number of these areas have been discussed with a view to aiding any future development of this project. In the author's opinion, these areas are, in the author's opinion, where the biggest performance increases could be achieved.

7.4 Final Thoughts

PageRank is a very important algorithm relied on by billions of people to retrieve useful information from the vast Internet. This algorithm is simply a linear algebra problem. It is widely known that large matrix calculations perform poorly on GPP. Thus, in this thesis benchmarks for the PageRank algorithm running on FPGA were presented for the first time. A hardware system for solving the PageRank eigenvector was described and a number of versions were benchmarked. These included fixed and floating point versions as well as versions running on different generations of FPGAs. The results were mixed, but promising. The floating point solution proved too slow on the Virtex-II FPGA, but when it was targeted at Virtex-5, a number of versions of the solver could outperform the GPP by almost 2X. The specialised PageRank hardware proved to be the best performing architecture. The SPAR, which is a generic SMVM unit, had disappointing results when using column major operations. The SCAR which is a block row solver for SMVM fared much better.

The fixed point versions of the architecture failed to increase the performance significantly and required a huge number of bits to represent large matrices. For full scale matrices, fixed point solutions are probably not practical but they may be of some use on smaller corporate matrices.

Overall, the FPGA increased the performance of the algorithm. However, this increase in performance was not large enough to justify a change from GPP to FPGA systems for solving PageRank. As FPGA prices drop more and their performance increases in future generations, it might yet be viable to accelerate the PageRank algorithm with an FPGA.

8 References

- [1] J. C. R. Licklider and E. C. Welden, "On-line man-computer communication," in *Proceedings of the May 1-3, 1962, spring joint computer conference* San Francisco, California: ACM, 1962.
- [2] J. Markoff, "An Internet Pioneer Ponders the Next Revolution," in *New York Times*, 1999.
- [3] L. Kleinrock, *Communication nets; stochastic message flow and delay*: Dover Publications, Incorporated, 1972.
- [4] V. G. Cerf and R. E. Kahn, "A protocol for packet network intercommunication," *SIGCOMM Comput. Commun. Rev.*, vol. 35, pp. 71-82, 2005.
- [5] T. Berners-Lee, "WWW: past, present, and future," *Computer*, vol. 29, pp. 69-77, 1996.
- [6] Google, "We Knew the Web was big ..." in *The official Google Blog*.available at <http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html> accessed on may 5th,2009
- [7] Google, "Technical Information."available at <http://www.google.com/corporate/tech.html> accessed on November 18th,2008
- [8] C. Moler, "The World's Largest Matrix Computation," in *Matlab News & Notes*.available at http://www.mathworks.com/company/newsletters/news_notes/clevescorner/oct02_cleve.html accessed on Feb. 12th,2009
- [9] L. A. Barroso, J. Dean, and U. Holzle, "Web search for a planet: The Google cluster architecture," *Micro, IEEE*, vol. 23, pp. 22-28, 2003.
- [10] Xilinx, "Our History."available at <http://www.xilinx.com/company/history.htm> accessed on November 18th,2008
- [11] D. Moloney, D. Geraghty, and F. Connor, "The performance of IEEE floating-point operators on FPGAs," in *IEE ISSC 2004*. vol. CP506: IEE Conf. Pub. 2004, 2004, p. 601.

- [12] K. Underwood, "FPGAs vs. CPUs: trends in peak floating-point performance," in *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays* Monterey, California, USA: ACM, 2004.
- [13] L. Zhuo and V. Prasanna, K., "High Performance Linear Algebra Operations on Reconfigurable Systems," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*: IEEE Computer Society, 2005.
- [14] M. deLorimier and A. DeHon, "Floating-point sparse matrix-vector multiply for FPGAs," in *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays* Monterey, California, USA: ACM, 2005.
- [15] Xilinx, "Virtex-II Complete Data sheet."available at http://www.xilinx.com/support/documentation/data_sheet/ds031.pdf accessed on March 10th,2009
- [16] W. D. Peterson, "Appendix A - WISHBONE Tutorial," WISHBONE SoC Architecture Specification, Revision B.3.
- [17] V. E. Taylor, A. Ranade, and D. G. Messerschmitt, "SPAR: a new architecture for large finite element computations," *Computers, IEEE Transactions on*, vol. 44, pp. 531-545, 1995.
- [18] S. McGettrick, D. Geraghty, and C. McElroy, "Searching the Web with an FPGA Based Search Engine," in *Reconfigurable Computing: Architectures, Tools and Applications*, Brazil, 2007, pp. 350-357.
- [19] D. Gregg, C. Mc Sweeney, C. McElroy, F. Connor, S. McGettrick, D. Moloney, and D. Geraghty, "FPGA Based Sparse Matrix Vector Multiplication using Commodity DRAM Memory," in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, 2007, pp. 786-791.
- [20] S. McGettrick, D. Geraghty, and C. McElroy, "Towards an FPGA solver for the PageRank EigenVector Problem," in *Parallel Computing: Architectures, Algorithms and Applications, ParCo 2007*, Julich and RWTH Aachen, Germany, 2007, pp. 793-800.
- [21] S. McGettrick, D. Geraghty, and C. McElroy, "An FPGA architecture for the Pagerank eigenvector problem," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, 2008, pp. 523-526.
- [22] Xilinx, "FPGA Design Flow Overview."available at http://www.xilinx.com/itp/xilinx7/help/iseguide/html/ise_fpga_design_flow_overview.htm accessed on January 14th,2009

-
- [23] "System C."available at <http://www.systemc.org> accessed on February 16th,2009
- [24] sc2v, "System-C to Verilog."available at <http://www.opencores.org/projects.cgi/web/sc2v/overview> accessed on February 16th,2009
- [25] Synplicity, "Synplify Pro - Product overview."available at <http://www.synplicity.com/products/synplifypro/> accessed on May 5th,2009
- [26] Xilinx, "Xilinx ISE 8 Software Manuals and Help - PDF Collection."available at <http://www.xilinx.com/itp/xilinx8/books/manuals.pdf> accessed on October 23rd,2009
- [27] Xilinx, "Getting Started with EDK."available at http://www.xilinx.com/support/documentation/sw_manuals/edk8li_getstarted.pdf accessed on October 23rd,2009
- [28] Alpha-Data, "ADP-DRC-II development platform datasheet."available at <http://www.alpha-data.com/pdf/ADP-DRC-II.pdf> accessed on Dec. 12th,2008
- [29] Xilinx, "Virtex-II Pro / Virtex-II Pro X Complete Data Sheet."available at http://www.xilinx.com/support/documentation/data_sheets/ds083.pdf accessed on May 10th,2009
- [30] Xilinx, "Virtex-4 Family Overview."available at http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf accessed on May 10th,2009
- [31] Xilinx, "Virtex-5 Family Overview."available at http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf accessed on May 10th,2009
- [32] Xilinx, "Virtex-5 FPGA User Guide."available at http://www.xilinx.com/support/documentation/user_guides/ug190.pdf accessed on May 10th,2009
- [33] Xilinx, "Virtex-5 Family Brochure."available at http://www.xilinx.com/publications/prod_mktg/Virtex_family_brochure.pdf accessed on May 10th,2009
- [34] Xilinx, "Xilinx Memory Interface Generator (MIG) User guide."available at http://www.xilinx.com/support/documentation/ip_documentation/ug086.pdf accessed on May 10th,2009
- [35] MemoryC.com, "Computer Memory suppliers."available at <http://www.memoryc.com> accessed on Jan. 15th,2009

- [36] Intel, "Dual-Core Intel Xeon Processor 5100 series Data sheet."available at <http://download.intel.com/design/Xeon/datashts/31335503.pdf> accessed on February 12th,2009
- [37] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, A., and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing* Reno, Nevada: ACM, 2007.
- [38] N. Brookwood, "The Role of Intelligent Design in the Evolution of Multi-Core Processors," *Insight* 64, 2006.
- [39] SiSoftware, "System ANalyser, Diagnostic and Reporting Assistant (SANDRA)."available at <http://www.sisoftware.co.uk> accessed on November 22nd,2008
- [40] "IEEE standard for binary floating-point arithmetic," *ANSI/IEEE Std 754-1985*, 1985.
- [41] A. N. Langville and C. D. Meyer, *Google's PageRank and Beyond: The Science of Search Engine Rankings*: Princeton University Press, 2006.
- [42] D. Austin, "How Google Finds Your Needle in the Web's Haystack," in *AMS Feature Column*.available at <http://www.ams.org/featurecolumn/archive/pagerank.html> accessed on February 12th,2009
- [43] J. Cho and H. Garcia-Molina, "The Evolution of the Web and Implications for an Incremental Crawler," Stanford InfoLab, Technical Report 1999.
- [44] D. Fetterly, M. Manasse, M. Najork, and J. L. Wiener, "A large-scale study of the evolution of Web pages," *Software: Practice and Experience*, vol. 34, pp. 213-237, 2004.
- [45] Google, "WebMaster tools."available at <http://www.google.com/webmasters/> accessed on Dec. 17th.,2008
- [46] F. Schneider, N. Blachman, and E. Fredricksen, *How to Do Everything with Google*: McGraw-Hill Osborne Media, 2003.
- [47] R. Lempel and S. Moran, "SALSA: the stochastic approach for link-structure analysis," *ACM Trans. Inf. Syst.*, vol. 19, pp. 131-160, 2001.
- [48] J. M. Kleinberg, "Authoritative sources in a hyperlinked environment," *J. ACM*, vol. 46, pp. 604-632, 1999.

- [49] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," Stanford InfoLab, Technical Report 1999.
- [50] S. Brin and L. Page, "The anatomy of a large-scale hypertextual Web search engine," *Comput. Netw. ISDN Syst.*, vol. 30, pp. 107-117, 1998.
- [51] R. A. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*: Addison-Wesley Longman Publishing Co., Inc., 1999.
- [52] C. D. Meyer, *Matrix analysis and applied linear algebra*: Society for Industrial and Applied Mathematics, 2000.
- [53] M. W. Berry, S. T. Dumais, and G. W. O'Brien, "Using linear algebra for intelligent information retrieval," *SIAM Rev.*, vol. 37, pp. 573-595, 1995.
- [54] Ask.com, "About: Web Search." available at <http://help.ask.com/en/docs/about/webmasters.shtml> accessed on April 20th, 2009
- [55] seobook.com, "How Search Engines Work: Search Engine Relevancy Reviewed." available at <http://www.seobook.com/relevancy/#msn> accessed on May 8th, 2009
- [56] C. Ding, X. He, P. Husbands, H. Zha, and H. Simon, "Link Analysis: Hub and Authorities on the World Wide Web," Lawrence Berkeley National Laboratory May 2001.
- [57] K. Bharat and M. R. Henzinger, "Improved algorithms for topic distillation in a hyperlinked environment," in *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval* Melbourne, Australia: ACM, 1998.
- [58] L. Ellis, "Google Marks Ten Years of Growth Since Founding." available at <http://www.searchengineworld.com/google/3458482.htm> accessed on December 6th, 2008
- [59] A. Langville and C. Meyer, "Deeper Inside PageRank," *Internet Mathematics*, vol. 1, pp. 335-380, 2003.
- [60] V. E. Taylor, "Application-Specific Architecture for Large Finite-Element Applications," in *Electrical Engineering*. vol. Doctor of Philosophy: University of California at Berkeley, 1991.
- [61] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct methods for sparse matrices*: Oxford University Press, Inc., 1986.
- [62] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, *Templates for the Solution*

- of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*: SIAM, 1994.
- [63] E.-J. Im and K. A. Yelick, "Optimizing Sparse Matrix Computations for Register Reuse in SPARSITY," in *Proceedings of the International Conference on Computational Sciences-Part I*: Springer-Verlag, 2001.
- [64] S. Toledo, "Improving the memory-system performance of sparse-matrix vector multiplication," in *IBM Journal of Research and Development*, 1997.
- [65] C. Mc Sweeney, "An FPGA accelerator for the iterative solution of sparse linear systems," in *School of Engineering, Dept. of Mechanical and Manufacturing Engineering*. vol. Masters in Science Dublin: University of Dublin, Trinity College, 2006.
- [66] A. Arasu, J. Novak, A. Tomkins, and J. Tomlin, "PageRank computation and the structure of the web," in *11th International WWW Conference Hawaii, USA*: ACM Press, 2002.
- [67] G. H. Golub and C. Greif, "An Arnoldi-type algorithm for computing page rank," *BIT Numerical Mathematics*, vol. 46, pp. 759-771, 2006.
- [68] G. M. Del Corso, A. Gulli, and F. Romani, "Exploiting Web Matrix Permutations to Speedup PageRank Computation."
- [69] D. Gleich, L. Zhukov, and P. Berkhin, "Fast Parallel PageRank: A Linear System Approach," in *WWW2005 Chiba, Japan, 2005*.
- [70] S. Kamvar, T. Haveliwala, and G. Golub, "Adaptive methods for the computation of PageRank," *Linear Algebra and its Applications*, vol. 386, pp. 51-65, 2004.
- [71] S. D. Kamvar, T. H. Haveliwala, C. D. Manning, and G. H. Golub, "Extrapolation methods for accelerating PageRank computations," in *Proceedings of the 12th international conference on World Wide Web Budapest, Hungary*: ACM, 2003.
- [72] S. D. Kamvar, T. H. Haveliwala, C. D. Manning, and G. H. Golub, "Exploiting the Block Structure of the Web for Computing," 2003.
- [73] Y. Lu, B. Zhang, W. Xi, Z. Chen, Y. Liu, M. R. Lyu, and W.-y. Ma, "The powerrank web link analysis algorithm," in *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters* New York, NY, USA: ACM, 2004.

- [74] C. Lee, G. Golub, and S. Zenios, "A Two-Stage Algorithm for Computing PageRank and Multistage Generalizations," *Internet Mathematics*, vol. 4, pp. 299-327, 2007.
- [75] A. N. Langville and C. D. Meyer, "A reordering for the PageRank problem," *SIAM J. Sci. Comput*, vol. 27, pp. 2112--2120, 2004.
- [76] Y. Zhu, S. Ye, and X. Li, "Distributed PageRank computation based on iterative aggregation-disaggregation methods," in *Proceedings of the 14th ACM international conference on Information and knowledge management* Bremen, Germany: ACM, 2005.
- [77] J. T. Bradley, D. V. Jager, W. J. Knottenbelt, and A. Trifunovic, "Hypergraph partitioning for faster parallel PageRank computation," in *Lecture Notes in Computer Science 3670*, 2005, pp. 155--171.
- [78] A. Cevahir, C. Aykanat, A. Turk, and B. Cambazoglu, "A Web-Site-Based Partitioning Technique for Reducing Preprocessing Overhead of Parallel PageRank Computation," in *Applied Parallel Computing. State of the Art in Scientific Computing*, 2008, pp. 908-918.
- [79] N.-Y. Xu, X.-F. Cai, R. Gao, L. Zhang, and F.-H. Hsu, "FPGA-based Accelerator Design for RankBoost in Web Search Engines," in *Field-Programmable Technology, 2007. ICFPT 2007. International Conference on*, 2007, pp. 33-40.
- [80] Y. Freund, R. Iyer, R. E. Schapire, and Y. Singer, "An efficient boosting algorithm for combining preferences," *J. Mach. Learn. Res.*, vol. 4, pp. 933-969, 2003.
- [81] W. Gropp, D. Kaushik, D. Keyes, and B. Smith, "Toward realistic performance bounds for implicit CFD codes," in *Proceedings of Parallel CFD'99*, 1999.
- [82] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *Proceedings of the 1969 24th national conference*: ACM, 1969.
- [83] A. George, "An efficient band-oriented scheme for solving n by n grid problems," in *Proceedings of the December 5-7, 1972, fall joint computer conference, part II* Anaheim, California: ACM, 1972.
- [84] E.-J. Im, "Optimizing the Performance of Sparse Matrix-Vector Multiplication," University of California at Berkeley 2000.

- [85] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee, "Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply," in *Supercomputing, ACM/IEEE 2002 Conference*, 2002, pp. 26-26.
- [86] S. Williams, R. Vuduc, L. Oliker, J. Shalf, K. A. Yelick, and J. Demmel, "Tuning Sparse Matrix Vector Multiplication for multi-core SMPs," Office of Science.
- [87] J. A. Swanson, G. R. Cameron, and J. C. Haberland, "Adapting the Ansys Finite-Element Analysis Program to an Attached Processor," *Computer*, vol. 16, pp. 85-91, 1983.
- [88] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *Micro, IEEE*, vol. 28, pp. 39-55, 2008.
- [89] M. Garland, "Sparse matrix computations on manycore GPU's," in *Proceedings of the 45th annual conference on Design automation Anaheim, California*: ACM, 2008.
- [90] J. Bolz, I. Farmer, E. Grinspun, and P. Schroder, "Sparse matrix solvers on the GPU: conjugate gradients and multigrid," *ACM Trans. Graph.*, vol. 22, pp. 917-924, 2003.
- [91] D. Goddeke, R. Strzodka, and S. Turek, "Accelerating Double Precision FEM Simulations with GPUs," in *Proceedings of ASIM 2005 - 18th Symposium on Simulation Technique*, 2005.
- [92] A. Wolfe, M. Breternitz, Jr., C. Stephens, A. L. Ting, D. B. Kirk, R. P. Bianchini, Jr., and J. P. Shen, "The white dwarf: a high-performance application-specific processor," in *Proceedings of the 15th Annual International Symposium on Computer architecture* Honolulu, Hawaii, United States: IEEE Computer Society Press, 1988.
- [93] S. Craven and P. Athanas, "Examining the viability of FPGA supercomputing," *EURASIP J. Embedded Syst.*, vol. 2007, pp. 13-13, 2007.
- [94] M. deLorimier, "Floating-Point Sparse Matrix-Vector Multiply for FPGAs," California Institute of Technology, 2005.
- [95] Cray-Supercomputers, "Cray XD1 Supercomputer," in *Legacy Products*. available at <http://www.cray.com/products/Legacy.aspx> accessed on January 13th, 2009

- [96] L. Zhuo and V. K. Prasanna, "High-performance and area-efficient reduction circuits on FPGAs," in *Computer Architecture and High Performance Computing, 2005. SBAC-PAD 2005. 17th International Symposium on*, 2005, pp. 52-59.
- [97] V. K. Prasanna and G. R. Morris, "Sparse Matrix Computations on Reconfigurable Hardware," *Computer*, vol. 40, pp. 58-64, 2007.
- [98] Y. El-Kurdi, W. J. Gross, and D. Giannacopoulos, "Sparse Matrix-Vector Multiplication for Finite Element Method Matrices on FPGAs," in *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*: IEEE Computer Society, 2006.
- [99] Y. El-kurdi, D. Fernandez, E. Souleimanov, D. Giannacopoulos, and W. J. Gross, "FPGA architecture and implementation of sparse matrix vector multiplication for the finite element method," *Computer Physics Communications*, vol. 178, pp. 558-570, April 2008.
- [100] J. Fender, "An FPGA-based hardware development system with multi-gigabyte memory capacity and high bandwidth." vol. Masters in Science Toronto: University of Toronto, 2005.
- [101] M. Becvar and P. Stukjunger, "Fixed-Point Arithmetic on FPGA," *Acta Polytechnica*, vol. 45, pp. 67-72, 2005.
- [102] Xilinx, "Core Generator."available at <http://www.xilinx.com/tools/logic.htm> accessed on May 10th,2009
- [103] Xilinx, "Multiplier v11.0," Logicore 2009.
- [104] Xilinx, "Adder/Subtractor V11."available at http://www.xilinx.com/support/documentation/ip_documentation/c_addsub_ds214.pdf accessed on May 12th 2009
- [105] B. Fagin and C. Renard, "Field programmable gate arrays and floating point arithmetic," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 2, pp. 365-367, 1994.
- [106] N. Shirazi, A. Walters, and P. Athanas, "Quantitative analysis of floating point arithmetic on FPGA based custom computing machines," in *Proceedings of the IEEE Symposium on FPGA's for Custom Computing Machines*: IEEE Computer Society, 1995.
- [107] L. Louca, T. A. Cook, and W. H. Johnson, "Implementation of IEEE single precision floating point addition and multiplication on FPGAs," in *FPGAs for*

- Custom Computing Machines, 1996. Proceedings. IEEE Symposium on*, 1996, pp. 107-116.
- [108] W. B. Ligon, III, S. McMillan, G. Monn, K. Schoonover, F. Stivers, and K. D. Underwood, "A re-evaluation of the practicality of floating-point operations on FPGAs," in *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, 1998, pp. 206-215.
- [109] E. Roesler and B. E. Nelson, "Novel Optimizations for Hardware Floating-Point Units in a Modern FPGA Architecture," in *Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*: Springer-Verlag, 2002.
- [110] Xilinx, "LogiCore Floating-Point Operator V4.0."available at http://www.xilinx.com/support/documentation/ip_documentation/floating_point_ds335.pdf accessed on November 10th,2008
- [111] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev, "64-bit floating-point FPGA matrix multiplication," in *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays* Monterey, California, USA: ACM, 2005.
- [112] G. Govindu, L. Zhuo, S. Choi, P. Gundala, and V. K. Prasanna, "Area, and Power Performance Analysis of a Floating-Point Based Application on FPGAs," in *Seventh Annual Workshop on High Performance Embedded Computing HPEC*, 2002.
- [113] P. Karlstrom, A. Ehliar, and D. Liu, "High-performance, low-latency field-programmable gate array-based floating-point adder and multiplier units in a Virtex 4," *Computers & Digital Techniques, IET*, vol. 2, pp. 305-313, 2008.
- [114] Xilinx, "Virtex-II pro complete data sheet."available at http://www.xilinx.com/support/documentation/data_sheets/ds083.pdf accessed on March 10th,2009
- [115] D. Gregg, C. McElroy, F. Connor, B. McElroy, and D. Geraghty, "A Method and Hardware Architecture for Sparse Matrix by Vector Multiplication," Patent document, 2007.
- [116] Stanford, "The Stanford WebBase Project."available at <http://diglib.stanford.edu:8091/~testbed/doc2/WebBase> accessed on May 13th,2009

- [117] P. Boldi and S. Vigna, "WebGraph: things you thought you could not do with Java\&trade," in *Proceedings of the 3rd international symposium on Principles and practice of programming in Java* Las Vegas, Nevada: Trinity College Dublin, 2004.
- [118] P. C. Roth and J. S. Vetter, "Intel Woodcrest: An Evaluation for Scientific Computing," in *8th LCI International Conference on High-Performance Clustered Computing*, United States, 2007.
- [119] G. Palla, I. Farkas, I. Derényi, A.-L. Barabási, and T. Vicsek, "Reverse engineering of linking preferences from network restructuring," *Physical Review E*, vol. 70, p. 046115, 2004.

Appendix One

9 Appendix 1: FE Matrices

Table 9.1 Summary of Finite Element Matrices (Bus Utilisation and FPU Utilisation quoted as a single SMVM SCAR unit).

Name	Rows	NNZ	Ave. Row Length	Sparsity	Bus utilisation	FPU utilisation
av41092	41092	1683902	41	0.10%	14.76%	58.41%
axle_covered	177330	12641162	71	0.04%	12.29%	79.09%
bm_slsb	71505	2090374	29	0.04%	9.25%	87.88%
bm-ttnb	52061	571967	11	0.02%	16.28%	81.16%
carrier	12174+	9157985	75	0.06%	8.20%	83.22%
ex35	19716	227872	12	0.06%	15.54%	82.19%
exp_bm-slsb	71505	4109243	57	0.08%	6.40%	90.46%
exp_bm-ttnb	52061	1091873	21	0.04%	9.74%	86.33%
front_axle_5KN	42291	2998741	71	0.17%	3.57%	88.39%
li	22695	1215181	54	0.24%	5.10%	93.05%
lung2	109460	492564	4	0.00%	31.34%	49.62%
ns3da	20414	1679599	82	0.40%	11.45%	84.26%
pkustk01	22044	979380	44	0.20%	8.82%	85.81%
pkustk03	63336	3130416	49	0.08%	8.41%	87.88%
pkustk04	55590	4218660	76	0.14%	6.29%	81.16%
pkustk05	37164	2205144	59	0.16%	8.06%	87.36%
pkustk06	43164	2571768	60	0.14%	8.10%	86.84%
pkustk08	22209	3226671	145	0.65%	3.41%	89.43%
pkustk09	33960	1583640	47	0.14%	9.10%	86.84%
pkustk10	80676	4308984	53	0.07%	9.02%	86.84%
pkustk11	87804	5217912	59	0.07%	8.97%	85.29%
pkustk12	94653	7512317	79	0.08%	6.72%	78.57%
pkustk13	94893	6616827	70	0.07%	7.26%	83.22%
pkustk14	151926	14836504	98	0.06%	6.28%	86.33%
pli	22695	1350309	59	0.26%	4.61%	93.56%
poisson3Db	85623	2374949	28	0.03%	50.97%	40.84%

qa8fk	66127	1660579	25	0.04%	10.59%	87.36%
qa8fm	66127	1660579	25	0.04%	10.60%	87.36%
sme3Dc	42930	3148656	73	0.17%	22.08%	72.37%
solid186	268692	18212700	68	0.03%	9.13%	83.22%
solid187	268692	18212700	68	0.03%	9.13%	83.22%
solid191	268692	18285672	68	0.03%	9.10%	83.22%
solid45	61812	2054614	33	0.05%	9.09%	86.84%
solid92	268692	18061226	67	0.03%	9.21%	83.22%
stomach	213360	3021648	14	0.01%	28.41%	56.86%
thermal_69k	68019	1337939	20	0.03%	12.81%	85.81%
torso1	116158	8516500	73	0.06%	6.64%	86.84%
torso2	115967	1033473	9	0.01%	33.36%	63.06%
torso3	259156	4429042	17	0.01%	29.30%	63.06%
turb1	246517	17824567	72	0.03%	10.70%	79.09%

Appendix Two

10 Appendix 2: Source Code

Due to the length of the source code of this work it was decided not to include it as an appendix. However, a copy of the code may be obtained from the author under normal college license agreements by emailing him at mcgettrs@tcd.ie.