**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.

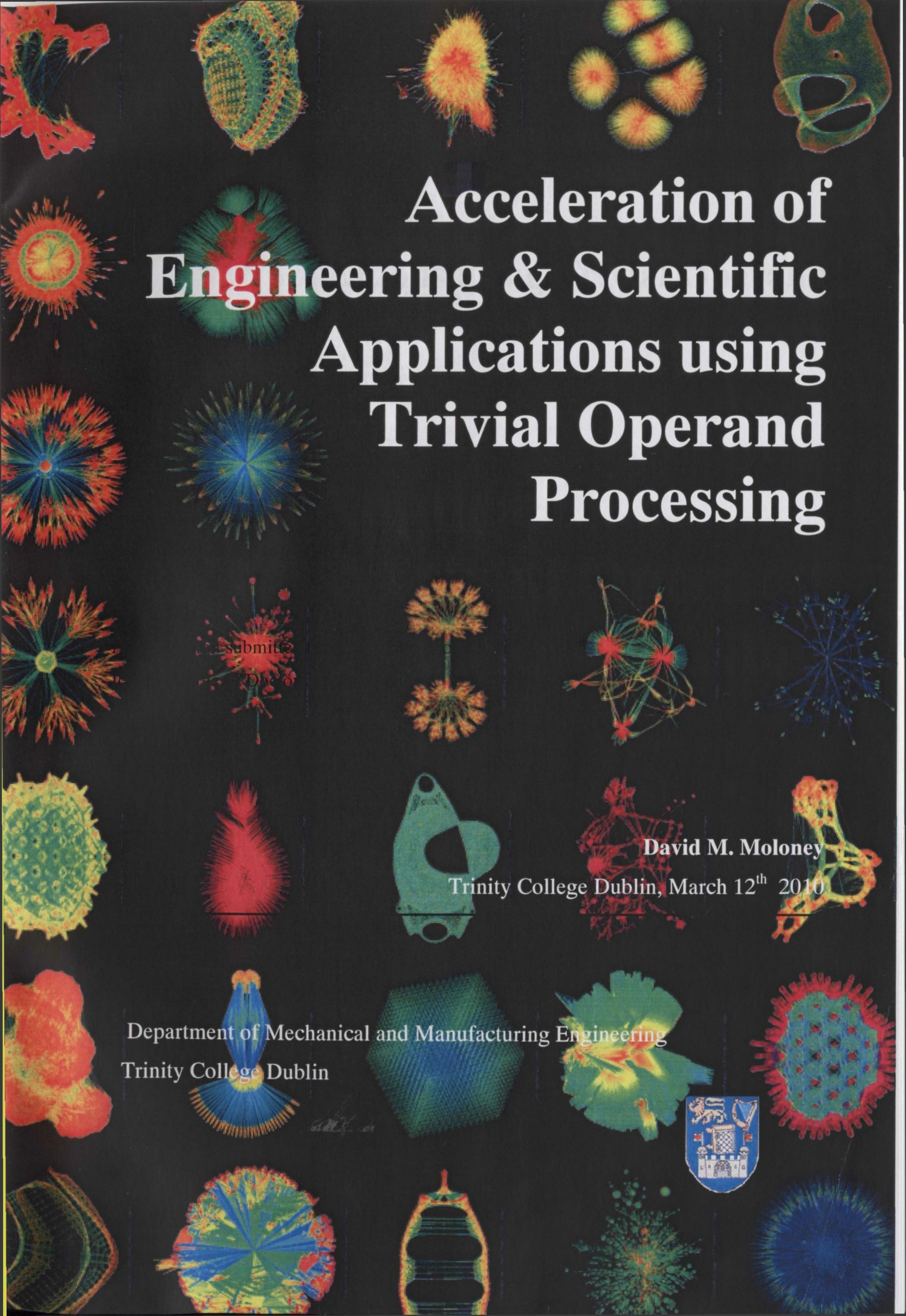
**Liability statement**

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.



# Acceleration of Engineering & Scientific Applications using Trivial Operand Processing


**David M. Moloney**

Trinity College Dublin, March 12th 2010

Department of Mechanical and Manufacturing Engineering

Trinity College Dublin

*Dedicated to Agnieszka, Paolo & Sean*

# Acceleration of Engineering & Scientific Applications using Trivial Operand Processing

A thesis submitted to the University of Dublin
for the degree of Doctor of Philosophy

**David M. Moloney**
Trinity College Dublin, March 12th 2010

---

Department of Mechanical and Manufacturing Engineering
Trinity College Dublin

# Abstract

"The fool who persists in his folly will become wise."

- William Blake

A vast array of scientific and engineering problems require the solution of linear systems of equations of the form $A * x = y$, where $A$ is the coefficient matrix of the system and $y$ is a vector of unknowns and $x$ is a vector of scalar known values. In practice the matrix $A$ is large and sparse for real-world problems. A range of iterative methods is used depending on the nature of the problem to be solved and multiple solvers are normally included in commercial and public domain applications based on these methods. Most implementations of iterative methods [3] and the linear algebraic operations on which they are based have to date been software implementations commonly implemented in the form of FORTRAN or C/C++ libraries. All mathematical libraries in common use make use of data-structures to store sparse matrices efficiently, and such methods can thus be regarded as a form of compression. General purpose computer architectures designed to provide a platform for the development and execution of applications perform very poorly on this class of operation often delivering only a few per cent of their peak processing capability to the user application. The design of such general-purpose machines is of necessity a compromise, and often the designer is faced with the problem that an improvement introduced to address performance issues for one application class has a detrimental effect on the performance of another. Like a balloon, which a child seeks to squeeze between his hands, the balloon when compressed in one direction, expands in the other 2 dimensions, conserving volume. In this thesis a variety of techniques for accelerating Sparse Matrix computations are proposed and evaluated experimentally. It will be shown that the performance of the kernel Sparse Matrix Vector Multiplication (SMVM) operation which dominates the execution time of iterative methods can be improved systematically when compared to General Purpose Processors and very significantly when compared to Special Purpose Computers, using streaming matrix compression and decompression to boost the sustainable Floating-Point performance compared with other architectures. Finally the case for low-cost hardware acceleration to further boost SMVM performance is outlined.

## Contributions of this work

Sparse linear algebra and in particular Sparse Matrix Vector Multiplication (SMVM) have been identified as one of the "7 Dwarves" [1][2] or unresolved key problems in the design of modern computer systems. In this thesis a variety of techniques for accelerating Sparse Matrix computations are proposed and evaluated experimentally. It will be shown that the performance of the kernel Sparse Matrix Vector Multiplication (SMVM) operation, which dominates the execution time of iterative methods, can be improved dramatically compared to General Purpose Processors (GPP) and significantly when compared to Special Purpose Computers (SPC).

The specific improvements over the state-of-the-art proposed, which boost performance, proposed in this thesis are:

- A first Bitmap Block Compressed Sparse Row (BBCSR) sparse matrix storage method is proposed which eliminates the zero fill associated with the BCSR (Block Compressed Sparse Row) sparse matrix format
- Benchmarking on a 50 matrix set of large sparse matrices demonstrates a significant speed-up in 7/50 cases using the proposed BBCSR format, using a standard gcc compiler and Intel Xeon processor, when compared with CSR and BCSR formats
- A second sparse matrix format Scheduled Block Compressed Sparse Row (SBCSR) format is proposed which addresses the need to perform up to r*c bitmap comparisons (where r and c are respectively the number of rows and columns in the dense block sub-matrix) and branches performed to implement the BBCSR Sparse Matrix Vector Multiplication (SMVM)

Again the SBCSR method is benchmarked against BBCSR, BCSR and BCSR methods for the same 50-matrix set, using the same configuration of gcc compiler, RHEL and Xeon processor

- A generic hardware accelerator is described which allows the SBCSR method to be utilised without penalty when compared with BCSR SMVM

- Integer sparse matrices such as the DCT coefficient matrices used in video applications are easily supported
- Finally the proposed hardware also allows compressed sparse data-structures to be random-accessed in situ without prior decompression, offering a major advantage over the state of the art

The work described carried out by the author at TCD and latterly at Movidius Ltd. has resulted in the following patent applications, the first of which has already been granted, and the remaining 3 of which are the subject of on-going patent applications:

- Geraghty D., Moloney D., "Data processing system and method", US2009030960 (A1), Priority Date 2005-05-13
- Moloney D., "A processor", WO2009101119 (A1) - 2009-08-20, Priority Date 2008-02-11
- Moloney D., "A processor exploiting trivial arithmetic operations", EP2137610 (A1) - 2009-12-30, Priority Date 2007-03-15
- Moloney D., "A circuit for compressing data and a processor employing same", EP2137821 (A1) - 2009-12-30, Priority Date 2007-03-15

The same work has also contributed so far to the following publications:

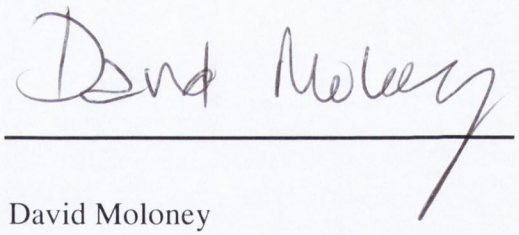
- D. Moloney, D. Geraghty, C. McSweeney and C. McElroy, "Streaming Sparse Matrix Compression/Decompression", in Lecture Notes in Computer Science 2005 (HiPEAC Conference), Springer-Verlag, No. 3793, pp. 116-129
- D. Moloney, C. McSweeney, C. McElroy and D. Geraghty, "Hardware accelerator for finite element iterative methods", IEE Irish Signals and Systems Conference 2005, pp.330–337
- D. Gregg, C. McSweeney, 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

## Declaration

I hereby declare that this thesis has not been submitted as an exercise for a degree at this or any other University and that it is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

I agree that the Library may lend or copy this thesis upon request.

Signed,



David Moloney

March 12th 2010

# Acknowledgements

The author gratefully acknowledges the assistance and guidance of his supervisor Mr Dermot Geraghty of the Department of Mechanical and Manufacturing Engineering, TCD. He also acknowledges the suggestions and guidance of Dr David Gregg of the Department of Computer Science, TCD.

The author thanks Enterprise-Ireland whose generous funding under the Research Innovation Fund RIF 2002/466 without which he would have been unable to start his research.

The generous donations of over €10,000 worth of equipment by Mr Gerry Maguire (formerly of Parthus-Ceva plc. now with Atlantic Bridge Ventures), Mr John Bourke and Mr Martin Farnan at Ceva plc. were most helpful to the research group, and ultimately to this work.

The later research work on bitmap SMVM, from October 2005 onwards, which generated the most interesting results, was conducted by the author at Movidius Ltd.

The author acknowledges the assistance of Mr Ivan Griffin, Mr Daire McNamara and Mr Alan Donnelly formerly of Frontier-Silicon (Ireland) Ltd. and the late Mr Martin Mellody of Movidius Ltd. in the form of discussions on tools and methodologies, and Bob Tait, whose advice on the nuances of MSWord was invaluable.

## Special Thanks

Si ringrazia il Dr Marco Scappecia per interventi di Radiologia e Risuonanza a cura di Fiammetta. Inoltre si ringrazia Dr Maurizio Salvati, Dr Cristian Brogna e l'equipe di neurochirurgia al Policlinico Umberto I di Roma per la loro sensibilita' e capacita' tecnica nei due interventi chirurgici di Fiammetta per tumore celebrale primario di tipo GBM.

Si ringrazia inoltre il Prof. de Paola, Dr di Palma per il corso post-operativo di radio e chemioterapia all'ospedale Fatebene Fratelli di Roma, ed il Professor Franca dell riparto oncologico del Policlinico Umberto Primo di Roma per la cura di Fiammetta.

Ringrazio tantissimo Dr Gloria Trocchi la sorella di Fiammetta per essere stata piu' di una sorella, anche per me e di aver dato sempre tutto per seguire Fiammetta dal punto di vista medico, navigando il sistema sanitario Italiana con grande abilita'.

In oltre ringrazio il padre di Fiammetta, Augusto e sua sorella Giulia Trocchi per l'aiuto costante durante la mallatia di Fiammetta e dopo.

I also wish to Thank Dr. Liam Grogan and his oncology team including Dr Con Murphy, Dr Lizzy Smyth and Nurses Carla and Aine for level of compassion and flexibility and the superlative level of care and the great sensitivity shown both to Fiammetta and myself during the many months we were weekly visitors to St. Clare's day-unit at Beaumont Hospital.

I would like to thank all of the staff of St. Francis Hospice in Raheny, especially Nurse Margaret, for their care and support given during the last weeks of Fiammetta's life.

I thank all of my family and friends both in Ireland and Italy and my colleagues including Sean Mitchell at Movidius Ltd. for their ongoing support and understanding in what were the most difficult years of my life.

# In Memoriam

**Fiammetta Trocchi (1959-2007)**

**"I wanted you to see what real courage is, instead of getting the idea that courage is a man with a gun in his hand. It's when you know you're licked before you begin but you begin anyway and you see it through no matter what."**

- ***Harper Lee, To Kill a Mockingbird, 1960***

**Martin Mellody (1975-2009)**

**"The light that burns twice as bright burns half as long …**
**and you have burned so very, very brightly []"**

**- Dr Eldon Tyrell, Bladerunner, 1982**

# Acronyms

| | |
|---|---|
| **SMVM** | Sparse Matrix Vector Multiplication |
| **SoC** | System on Chip |
| **FMA** | Fused Multiply Add |
| **EBE** | Element-by Element |
| **RAW** | Read After Write hazard |
| **CSR** | Compressed Sparse Row |
| **CSC** | Compressed Sparse Column |
| **BCSR** | Block Compressed Sparse Row |
| **BCSC** | Block Compressed Sparse Column |
| **BBCSR** | Bitmap Block Compressed Sparse Row |
| **BBCSC** | Bitmap Block Compressed Sparse Column |
| **SBCSR** | Scheduled Block Compressed Sparse Row |
| **SBCSC** | Scheduled Block Compressed Sparse Column |
| **FP** | Floating Point |
| **FPU** | Floating Point Unit |
| **GPP** | General Purpose Processor |
| **GPU** | Graphics Processing Unit |
| **CMP** | Chip Multi-Processor |
| **FEM** | Finite Element Method |
| **PDE** | Partial Differential Equations |
| **CFD** | Computational Fluid Dynamics |
| **FPGA** | Field Programmable Gate Array |
| **CAD** | Computer Aided Design |
| **CG** | Conjugate Gradient |
| **CGNE/CGNR** | Conjugate Gradient on Normal Equations |
| **GMR** | Generalised Minimal Residual |
| **BiCG** | Bi-Conjugate Gradient |
| **QMR** | Quasi-Minimal Residual |
| **CGS** | Conjugate Gradient Squared |
| **BiCGSTAB** | Bi-Conjugate Gradient Stabilised |
| **CHEB** | Chebyshev Iteration |
| **IDR** | Induced Dimension Reduction |

| | |
|---|---|
| **DDOT** | Double precision DOT product |
| **DDIV** | Double precision DIVision |
| **DAXPY** | Double precision Alpha.X Plus Y |
| **FLOPS** | Floating Point Operations Per Second |
| **IEEE** | Institute of Electrical and Electronics Engineers |
| **IEEE-754** | IEEE 754 Floating-Point Arithmetic Standard |
| **NZ** | Non-Zero |
| **ICSR** | Incremental Compressed Sparse Row |
| **ICCS** | Incremental Compressed Column Storage |
| **HiSM** | Hierarchical Sparse Matrix |
| **IPC** | Instructions Per Clock |
| **CPI** | Clocks Per Instruction |
| **FO4** | Fanout Of 4 |
| **WAW** | Write After Write hazard |
| **WAR** | Write After Read hazard |
| **PC** | Program Counter |
| **OOO** | Out-Of-Order execution |
| **SIMD** | Single Instruction Multiple Data |
| **VLSI** | Very Large Scale Integration |
| **DRAM** | Dynamic Random Access Memory |
| **LRU** | Least Recently Used |
| **ILP** | Instruction Level Parallelism |
| **TLP** | Thread Level Parallelism |
| **RF** | Register File |
| **MT** | Multi-Threading |
| **SMT** | Simultaneous Multi-Threading |
| **I/O** | Input Output |
| **ISA** | Instruction Set Architecture |
| **CMT** | Chip Multi-Threading |
| **TLB** | Translation Lookaside Buffer |
| **DP** | Double Precision floating point |
| **SP** | Single Precision floating point |
| **NOE** | Network Offload Engine |
| **XML** | eXtensible Markup Language |
| **OS** | Operating System |

| | |
|---|---|
| **API** | Application Programming Interface |
| **MAC** | Multiply ACcumulate |
| **FEA** | Finite Element Analysis |
| **LSI** | Latent Semantic Indexing |
| **LSA** | Latent Semantic Analysis |
| **RCM** | Reverse Cuthill-McKee |
| **HPC** | High Performance Computing |
| **UBCSR** | Unaligned Block Compressed Sparse Row |
| **MTL** | Matrix Template Library |
| **LU** | Lower/Upper Decomposition |
| **CF** | Cholesky Factorisation |
| **CPU** | Central Processing Unit |
| **ZZCSR** | Zig-Zag Compressed Sparse Row storage format |
| **ZZICSR** | Zig-Zag Incremental Compressed Sparse Row storage format |
| **ZZBCSR** | Zig-Zag Block Compressed Sparse Row storage format |
| **SBCSR_C** | Scheduled Block Compressed Sparse Row (column ordered) |
| **MPSoC** | Multi-Processor System On Chip |
| **SDRAM** | Synchronous Dynamic Random Access Memory |
| **LUT** | Look-Up Table |
| **MUX** | Multiplexer |
| **FA** | Full-Adder |
| **HBCSR** | Hybrid Block Compressed sparse Row |

# Contents

# 1

# Chapter 1

**"Anyone who has never made a mistake has never tried anything new."**

**- Albert Einstein**

## 1 Introduction

Many scientific and engineering problems require the solution of linear systems of equations of the form $A * x = y$, where $A$ is the coefficient matrix of the system and $x$ is a vector of unknowns and $y$ is a vector of scalar known values. In practice the matrix $A$ is large and sparse for real-world problems yet it must fit within the available system memory, hence efficient storage of the matrix data is a requirement to be able to solve problems of arbitrary size. Also when the matrix size is very large classical solution methods such as Gaussian Elimination are no longer efficient in terms of processing and memory requirements so iterative methods are typically applied to such problems. A range of iterative methods is used depending on the nature of the problem to be solved and multiple solvers are normally included in commercial and public domain applications based on these methods.

A major review of the key challenges in computing systems by Patterson et. al at UC Berkeley [1] concluded that Sparse Matrix by Vector multiplication (SMVM), was one of

the key challenges, or "7 dwarves", whose resolution was likely to significantly advance progress in computer architecture development.

Most implementations of iterative methods [3] and the linear algebraic operations on which they are based have to date been software implementations commonly implemented in the form of FORTRAN, C or C++ mathematical libraries. These libraries make use of the IEEE-754 [4] compliant floating-point units (FPU) included as part of high performance computing systems, microprocessors and workstations. All mathematical libraries in common use make use of data-structures to store sparse matrices efficiently, and such methods can thus be regarded as a form of compression.

Such libraries or associated tools also contain additional algorithms which are used to pre-process the matrix data in order to ensure Sparse Matrix Vector Multiplication (SMVM) makes the most efficient use possible of the available processing and memory resources on a given platform. The techniques applied typically involve graph-based re-ordering of the non-zero elements and their associated indices in order to maximise spatial and temporal locality which in turn tends to maximise the amount of processing which can be achieved for each read or write from or to system memory. Increased spatial or temporal locality tends to result in data required by the next step of the SMVM algorithm being read from an internal register or cache, rather than from external memory, resulting in a lower access time and higher processing speed.

The final software optimisation, which can be performed on a typical computing system, is to split a large sparse matrix and associated vectors for use in a matrix-vector product into a series of smaller problems, which can be run independently on a number of parallel processors. The partial results from the parallel processors can then be combined in a simple post-processing step (such as addition of the vector sub-segments) to complete the Sparse Matrix-Vector Multiplication.

The focus of this work is on scientific and engineering applications primarily because a large body of test data is available for such problems [6][11][12]. However, linear algebra is emerging as enabling functionality in a range of applications such as Spam-filtering [13] and face-recognition [14] which are embedded into infrastructure products today and may well find their way into more deeply embedded and even mobile devices in the future.

According to Kogge [15] there are three approaches to performing Sparse Matrix-Vector Multiplication (SMVM) on programmable computers:

- Inner Product where the result vector y is computed one inner-product at a time, using aij and xj values read from memory
- Sub-matrix where the matrix A is tiled up into sub-matrices which are read along with equally sized segments of the y and x vectors
- Column Scaling where multiple entries in the result vector y are read and updated as each column is scanned for aij values and multiplied by a single entry from the x-vector xi

The inner product method is commonly used and works well with the Compressed Sparse Row format detailed in section 3.1. The advantage of the inner-product method is that y values are held in a register rather than memory resulting in fast access. Ideally a software designer would like to be able to produce an updated y value each cycle as partial-products are accumulated, however the correct previous y-value is not immediately available due to the fact that the adder has multiple clock-cycles of latency, resulting in a circular dependency called a Read After Write (RAW) hazard. Care must be taken in sequencing operations to ensure that the hardware does not detect potential RAW hazards and stall the processor in order to eliminate the RAW hazards.

The sub-matrix approach is often used for dense-matrix multiplication as it is highly efficient in terms of the amount of memory bandwidth usage. The method is not compatible, however with the most widely used Sparse Matrix storage formats Compressed Sparse Row (CSR/CSR) and Compressed Sparse Column (CSC/CCS).

A variety of other formats have been developed over the years to better handle certain variants in terms of matrix structure, however the formats in most common use are CSR/CSC and their blocked variants BCSR/BCSC. The blocked formats are particularly useful where the underlying matrix has some structure that allow it to be partitioned, thus improving locality and hence performance. However blocking is complex and time-consuming and may disimprove system performance and power, key concerns in an increasingly energy-conscious world.

## 1.1 Thesis Organisation

The background to the broad use of Sparse Matrix Vector Multiplication in a myriad of applications from designing aircraft to performing a Google search is explained in Chapter 2. Equally it will be shown that SMVM is the dominant performance determinant in many of these applications and the iterative methods upon which they depend.

A key issue in the performance of these applications is the storage format used to represent the sparse matrix and a range of existing storage formats along with the arithmetic methods (SMVM) that operate on them are surveyed at length in Chapter 3 and returned to again in Chapters 4, 5 and 6.

Users and developers of these large-scale applications is system performance and have that computer systems often perform very poorly on this class of applications. The defining characteristics of modern processor architectures and their implementations, and more importantly the limitations that processor and hardware architectures place on applications performance are surveyed in depth in chapter 4.

The available software techniques for improving the performance of SMVM across the range of processor architectures is fully explored in Chapter 5 and the overhead of many of these methods is found to require tens to hundreds or even thousands of unoptimized SMVMs in order to return an overall improvement in performance.

In partial answer to some of the performance limitations of existing blocked sparse storage formats two new sparse-storage formats are introduced in Chapter 6. The formats are both based on trivial techniques first identified by Richardson [138] in 1992. These methods have remained largely ignored in the intervening period as the solution identified by Richardson didn't address the fundamental underlying problem of limited memory bandwidth, which was highlighted by McKee [33] as the "Memory Wall". Essentially the method outlined by Richardson adds to the difficulties the programmer and processor designer face by first fetching trivial data into the processor, consuming valuable bandwidth, before deciding that the data is not required and can be bypassed. Indeed having multiple parallel floating-point and trivial processing units makes the processor larger, slower and more difficult to program, negating many of the supposed benefits of trivial operand processing.

The key insight explored in this work is that by decoupling trivial operand detection (a trivial multiplication involves multiplication by a trivial operand i.e. +1, -1 or 0) from processing, a compression can be obtained. This compression increases effective memory bandwidth between processor and the entire memory hierarchy, and separately trivial operand processing can then be performed without the necessity for specialised hardware in the processor pipeline as proposed by Richardson. It is also shown that the overhead of trivial operand detection and tagging has negligible cost which is dwarfed by the cost of assembling the data-structures required for large scale numerical applications, thus the benefits of the proposed accrue almost entirely to the whole application.

The performance of these methods is explored in detail in terms first experimentally using a suite of 50 large sparse matrices as a benchmark suite running on a typical engineering workstation with modern multicore processor. Tuning of the type proposed by Vuduc is avoided in order not to introduce any bias at the expense of increased run-time for the benchmark suite and a standard gcc C-compiler with optimisations was used in all experiments. The Bitmap Block Compressed Sparse Row (BBCSR) format was shown to perform better than Block Compressed Sparse Row (BCSR) or Compressed Sparse Row (CSR) reference methods in 7 out of 50 cases, performing on average 7.85% better when compared with CSR, and 13.93% compared with BCSR. The reasons for this increase in performance are analysed by examining the assembly code in detail as well as the processor architecture and cache hit-rates, as well as the effect of fill on BCSR performance.

The observations made by analysing BBCSR performance lead to the insight that additional overhead in the form of additional comparisons could be avoided if a schedule could be generated from the non-zero pattern. This schedule would only compute the required non-trivial partial-products, eliminating the comparison overhead for trivial values and greatly simplifying the development of SMVM libraries. This observation and the discovery of an obscure feature of the gcc compiler enabled the Scheduled Block Compressed Sparse Row (SBCSR) format to be developed. This format was also duly benchmarked and analysed in both row-major and column-major scheduled variants.

The row-major SBCSR format was found to be up to 60% faster than BCSR or CSR in 2 out of 50 cases, and the column-major format was found to be as fast in the case of the

vibrobox and gyro_m matrices, but also between 3 and 16% faster than the other formats in a further 4 out of 50 cases.

In chapter 7 the shortcomings of the software BBCSR and SBCSR formats were highlighted and the case for hardware acceleration was outlined. The implementation and functional model for the accelerator were described and the resultant hardware cost estimated.

Finally in Chapter 8 the conclusions for the work as a whole are drawn and directions for future work are explored.

## 1.2 Contributions

In this thesis a variety of techniques for accelerating Sparse Matrix computations are proposed and evaluated experimentally. It will be shown that the performance of the kernel Sparse Matrix Vector Multiplication (SMVM) operation, which dominates the execution time of iterative methods, can be improved dramatically compared to General Purpose Processors (GPP) and significantly when compared to Special Purpose Computers (SPC).

The specific improvements over the state-of-the-art proposed, which boost performance, proposed in this thesis are:

- A first Bitmap Block Compressed Sparse Row (BBCSR) sparse matrix storage method is proposed which eliminates the zero fill associated with the BCSR (Block Compressed Sparse Row) sparse matrix format
- Benchmarking on a 50 matrix set of large sparse matrices demonstrates a significant speed-up in 7/50 cases using the proposed BBCSR format, using a standard gcc compiler and Intel Xeon processor, when compared with CSR and BCSR formats
- A second sparse matrix format Scheduled Block Compressed Sparse Row (SBCSR) format is proposed which addresses the need to perform up to r*c bitmap comparisons (where r and c are respectively the number of rows and columns in the dense block sub-matrix) and branches performed to implement the BBCSR Sparse Matrix Vector Multiplication (SMVM)

Again the SBCSR method is benchmarked against BBCSR, BCSR and BCSR methods for the same 50-matrix set, using the same configuration of gcc compiler, RHEL and Xeon processor

- A generic hardware accelerator is described which allows the SBCSR method to be utilised without penalty when compared with BCSR SMVM
- Integer sparse matrices such as the DCT coefficient matrices used in video applications are easily supported
- Finally the proposed hardware also allows compressed sparse data-structures to be random-accessed in situ without prior decompression, offering a major advantage over the state of the art

The work described carried out by the author at TCD and latterly at Movidius Ltd. has resulted in the following patent applications, the first of which has already been granted, and the remaining 3 of which are the subject of on-going patent applications:

- Geraghty D., Moloney D., "Data processing system and method", US2009030960 (A1), Priority Date 2005-05-13
- Moloney D., "A processor", WO2009101119 (A1) - 2009-08-20, Priority Date 2008-02-11
- Moloney D., "A processor exploiting trivial arithmetic operations", EP2137610 (A1) - 2009-12-30, Priority Date 2007-03-15
- Moloney D., "A circuit for compressing data and a processor employing same", EP2137821 (A1) - 2009-12-30, Priority Date 2007-03-15

The same work has also contributed so far to the following publications:

- D. Moloney, D. Geraghty, C. McSweeney and C. McElroy, "Streaming Sparse Matrix Compression/Decompression", in Lecture Notes in Computer Science 2005 (HiPEAC Conference), Springer-Verlag, No. 3793, pp. 116-129
- D. Moloney, C. McSweeney, C. McElroy and D. Geraghty, "Hardware accelerator for finite element iterative methods", IEE Irish Signals and Systems Conference 2005, pp.330–337
- D. Gregg, C. McSweeney, 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

# 2

# Chapter 2

**"I ran into Isosceles. He had a great idea for a new triangle!"**

**- Woody Allen**

## 2 Finite Element Method (FEM) Applications

An overview of Finite Element (FEM) Applications and the numerical methods that underpin them is presented to set the Sparse Matrix-Vector Multiplication (SMVM) computational kernel, which it is proposed to accelerate, in context. There is a large body of publications from a range of application areas on FEM and a large number of books such as the classic text by Zienkiewicz [16] with some available freely on the web [17][18].

### 2.1 Introduction

The Finite Element Method (FEM) is a numerical technique for finding approximate solutions to real-world scientific and engineering problems first proposed by Courant [19] in 1943, drawing on the earlier work of Rayleigh, Ritz and Galerkin on Partial Differential Equations (PDE), for structural engineering problems. Clough [20] eventually coined the term Finite Element to describe the method in 1960. FEM analysis was rapidly extended, and improved by mathematicians and engineers for use in aeronautics [21], large structural

engineering and military projects. It required the use of mainframe computers to perform even relatively limited analysis and in fact even mainframes were not optimised for this kind of workload. Over sixty years later Finite Element Analysis is now available to individual scientists and engineers as a method of solving real-world problems, using low-cost personal computers.

The finite-element method is used to solve a simplified mathematical model of the actual physical problem under consideration. The method allows approximate solutions to such problems with bounded error to be computed by reducing a continuum with an infinite number of degrees of freedom to a set of elements with a finite number of degrees of freedom; the effect is to reduce from a set of equations with an infinite number of unknowns to one with a finite number of unknowns. Elastic, thermal, fluid-flow and electrostatics problems, to name but a few, are representable in terms of a set of governing equations and associated boundary conditions as shown in Figure 2-1.

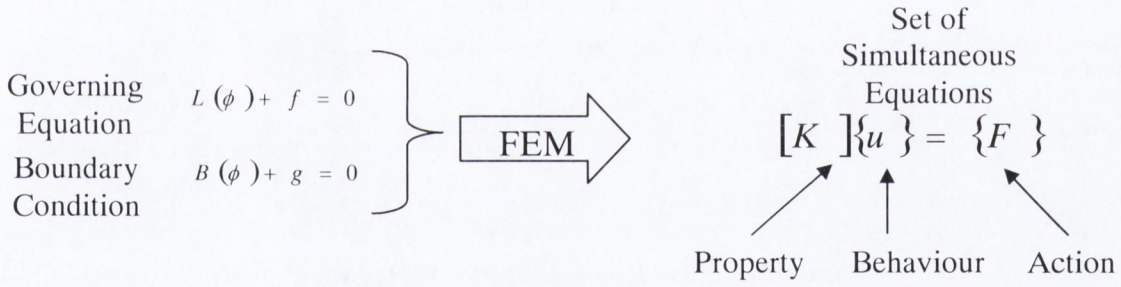



**Figure 2-1 FEM Governing Equations & Boundary Conditions (source: [22])**

In the case of each type of problem the Property [K] and the Action {F} elaborated by the Finite Element Method (FEM) allow the unknown Behaviour {u} to be solved for at each finite element in the structure. The unknown behaviour solved for in a variety of systems is shown in Table 2-1.

| Type of Problem | Property [K] | Behaviour {u} | Action {F} |
|---|---|---|---|
| Elastic | stiffness | displacement | force |
| Heat-flow | conductivity | temperature | heat source |
| Fluid | viscosity | velocity | body force |
| Electrostatic | dielectric permitivity | electric potential | charge |

**Table 2-1 Behaviours Solved for by FEM in different problems (source: [16])**

### 2.1.1 Finite Element (FEM) Analysis

In general the finite element method consists of 3 fundamental steps:

- Pre-processing
- Analysis
- Post-processing

The pre-processing step is the most tedious to perform for the designer and consists of the reduction of a complex structure into a collection of basic elements (triangles or other basic element shapes), connected by nodes. Two types of mesh can be used in Finite Element (FEM) problems, structured and unstructured. As can be seen in Figure 2-2 a structured mesh while simpler requires a much finer grid to provide the same level of detail as an unstructured mesh. The trade-off is one of more yet simpler calculations for a structured mesh as opposed to fewer, but more complex calculations for an unstructured mesh.

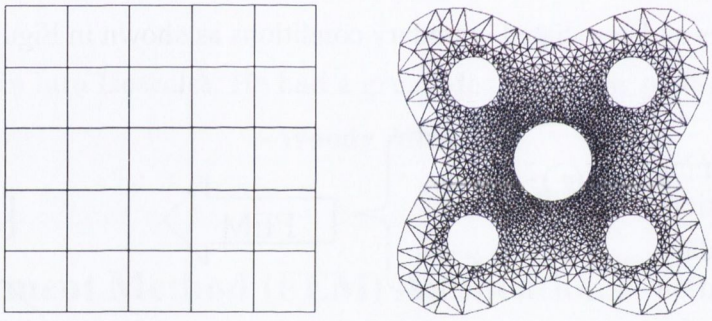


**Figure 2-2 Structured Versus Unstructured Meshes (source: [22])**

A user-directed mesh generator program using as input a drawing produced by a Computer Aided Design (CAD) package typically performs this step. An example of a finite-element mesh with triangular elements superimposed on a drawing of a truck axle is shown in Figure 2-3.

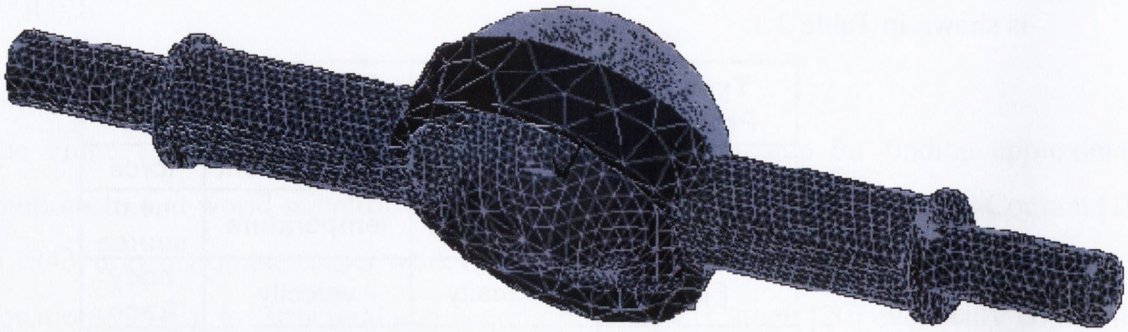


**Figure 2-3 Triangular Finite Element Mesh Superimposed on Truck Axle**

In the Analysis step the dataset produced by the mesh generator is imported. The mesh elements are reconnected at the nodes, which provide the connectivity generated by the mesh generator that holds the collection of elements together, thus approximating the original structure. This process results in a set of simultaneous algebraic equations, which can be solved by direct or iterative numerical methods. The set of simultaneous equations is represented as a matrix problem as shown in Figure 2-4 where the matrix K is known as the global stiffness matrix, the vector $\vec{u}$ represents the unknowns and the vector $\vec{f}$ represents the known perturbation applied to the system being modelled.

$$\begin{bmatrix} K_{11} & K_{12} & K_{13} & .. & K_{1n} \\ K_{21} & K_{22} & K_{23} & .. & K_{2n} \\ K_{31} & K_{32} & K_{33} & .. & K_{3n} \\ \vdots & \vdots & \vdots & . & \vdots \\ K_{n1} & K_{n2} & K_{n3} & .. & K_{nn} \end{bmatrix} \begin{Bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_n \end{Bmatrix} = \begin{Bmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_n \end{Bmatrix}$$

**Figure 2-4 Matrix Formulation of FEM Problem (source: [22])**

The stiffness matrix $K$ is assembled using element stiffness matrices $k$ specific to the problem area (elasticity, electrostatics etc.) and the element and connectivity information from the finite element mesh. The nature of the FEM matrix assembly process allows the same Finite Element solver to be applied to determine the aerodynamic properties of an aerofoil (fluid dynamics) and the stresses and strains which occur in the same aerofoil (elasticity) using the same CAD model of the aerofoil and even the same mesh, by changing the element matrix type used during matrix assembly. In the final post-processing step the unknown behaviour $u$ solved for by the FEM analysis is overlaid on top of the CAD drawing with coloured contours representing the field values.

### 2.1.2 Matrix Assembly

The Element-by Element (EBE) scheme was developed for heat conduction problems [6] and subsequently extended to structural and solid mechanics problems [7]. EBE has the benefit of avoiding matrix-assembly, however using these methods can result in up to 8 times more FLOPS being required to solve the same system in 3D applications [23] when compared with solution methods using the assembled matrix they will be not be considered here.

In fact EBE is only of benefit where the number of iterations required to achieve convergence using the assembled matrix is low enough to undercut the matrix assembly

time overhead on the method using the assembled matrix. The stiffness matrix K generated by the matrix assembly step in a Finite Element Analysis is always sparse but the exact pattern of non-zeroes is dependent on the row and column addresses, which in turn depend on the node numbering employed. In the example generated using [24] are shown in Figure 2-5 in all three cases the same 3x3 mesh was used and only the numbering of the nodes was varied.

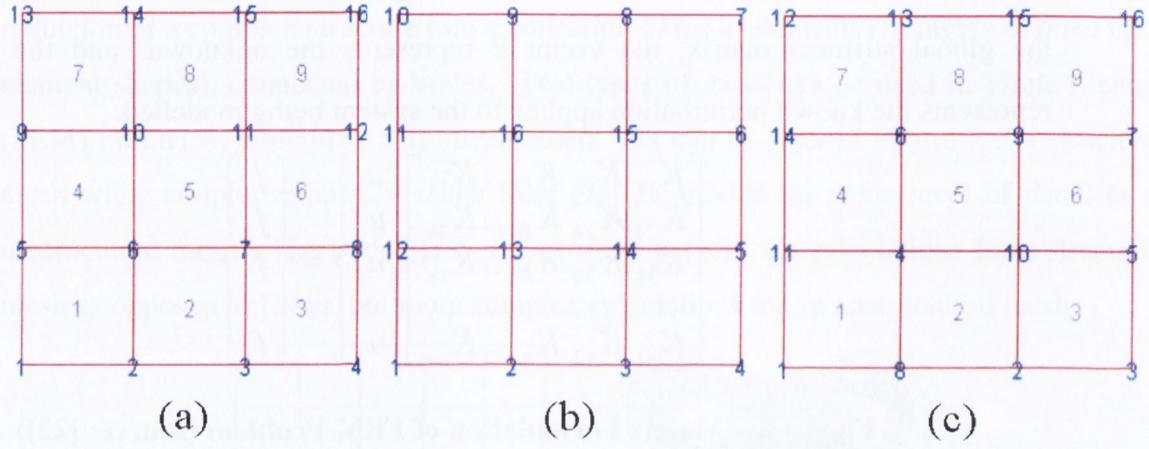



**Figure 2-5 Finite Element Mesh Node Numbering (source: [24])**

The effect of the node numbering employed in Figure 2-5 on the pattern of non-zeroes in the stiffness matrix is shown in Figure 2-6; (a) results from a node numbering, which progresses along rows or columns successively, the pattern in (b) results from a spiral ordering and (c) results from a random ordering.

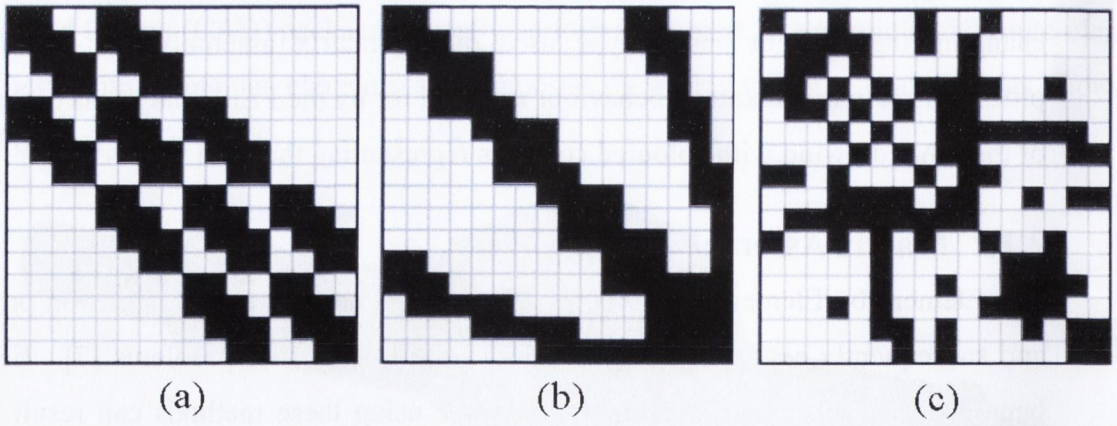



**Figure 2-6 Effect of Mesh Numbering Schemes on Stiffness Matrix (source: [24])**

A typical example of an assembled finite element stiffness matrix is the bcsstk32 matrix from the Harwell-Boeing collection published on MatrixMarket [11] which was generated from the static analysis of an automobile chassis. The plot in Figure 2-7 shows the sparsity

pattern of the assembled stiffness matrix. The matrix is symmetric, its order is 44609, and however, it only contains 2 M non-zeroes meaning that 99.95% of its elements are zero.

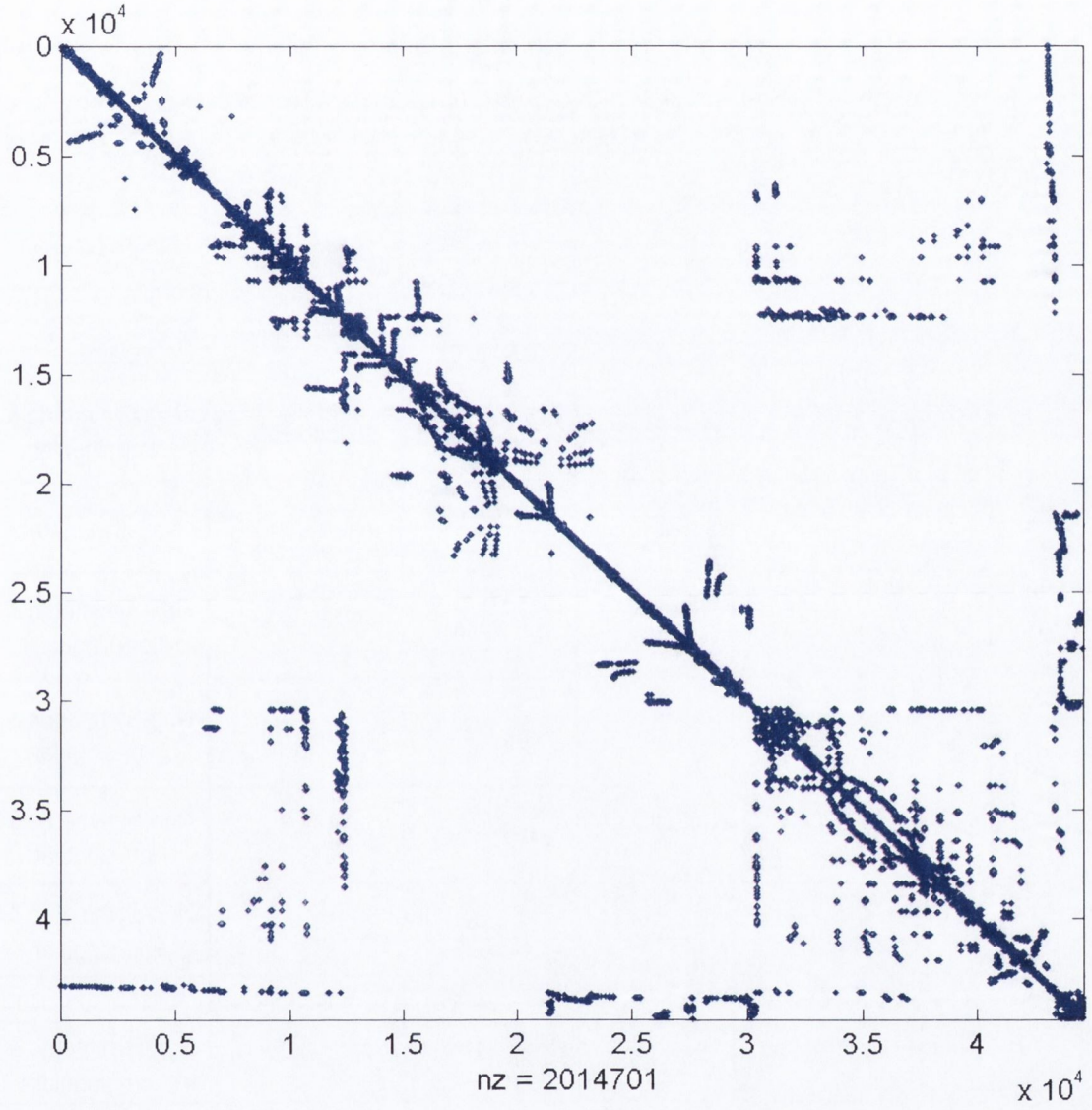



**Figure 2-7 Sparse Stiffness Matrix Plot (bcsstk32) from FEM Analysis (source: [11])**

## 2.2 Iterative Methods

If the stiffness matrix A is relatively small, Direct Methods can be applied independently of sparsity. However, if the matrix A is very large and sparse, Iterative Methods are more efficient. Direct Methods have been used historically and didactically but iterative methods are now the core of commercial solvers because of their superior computational efficiency. Iterative methods are so-called because an algorithm is applied to a linear system repeatedly until either the number of iterations exceeds a user-defined threshold or the solution converges to within the user-defined convergence constraint. Iterative

methods are characterised by the fact that they only access the coefficient matrix $A$ or its transpose $A^T$ (equivalent to stiffness matrix $K$ if the source of the system of equations is a Finite Element Analysis) via a matrix-vector product $y = A \cdot x$. This matrix-vector product is typically a Sparse Matrix-Vector Multiplication (SMVM) where the matrix-vector multiplication algorithm takes advantage of sparse matrix storage to reduce or eliminate entirely the number of trivial multiplications by zero, which would otherwise occur.

| Name | | Conv. | Storage | Matrix-Type | SMVM |
|---|---|---|---|---|---|
| Conjugate Gradient | CG | depends on matrix | matrix+6N | Symmetric Positive Definite | 1 |
| Conjugate Gradient on Normal Eqns | CGNE | slow | matrix+6N | Non-Symmetric & Non-singular | 1 |
| | CGNR | | | | |
| Generalised Minimal Residual | GMRES | depends on matrix | matrix+(i+5)*N | Non-Symmetric | 1 |
| BiConjugate Gradient | BiCG | irregular | matrix+10N | Non-Symmetric & Non-singular | 1 |
| Quasi-Minimal Residual | QMR | smooth | matrix+16Nc | Non-Symmetric & Non-singular | 1 |
| Conjugate Gradient Squared | CGS | irregular | matrix+11N | Non-Symmetric & Non-singular | 2 |
| Bi-Conjugate Gradient Stabilised | BiCGSTAB | smooth | matrix+10N | Non-Symmetric & Non-singular | 2 |
| Chebyshev Iteration | CHEB | | matrix+5N | Positive Definite | 1 |
| Induced Dimension Reduction | IDR | IDR(1) same as BiCGSTAB | | Non-Symmetric & Non-singular | 1 |

**Table 2-2 Characteristics of Iterative Methods (source: [5])**

The most commonly used iterative methods, their advantages and disadvantages, are described in detail by Dongarra et al in [5] and a summary of the characteristics of those methods is given in Table 2-2, where N is the order of the matrix. The matrix dominates the storage requirements for all methods, although the term dependent on the matrix order $n$ can be significant if GMRES or QMR are employed. The Sparse Matrix Vector Multiplication (SMVM) dominates the computational requirements. The higher computational cost of a single iteration may be offset by the fact that some methods require

fewer iterations to converge, e.g. if BiCGSTAB requires less than half the iterations of BiCG the overall computational cost of BiCGSTAB will be lower for a given problem.

In addition to these methods IDR (Induced Dimension Reduction), a forerunner to BiCGSTAB, introduced by Sonneveld [6] is highly efficient in solving non-symmetric systems. In fact IDR(s) with s>1 can be much faster than BiCGSTAB in many cases. Generally the number of matrix-vector products decreases as s is increased with IDR(4) and IDR(6) being close to the optimal convergence curve of full GMRES [9].

### 2.2.1 Conjugate Gradient Method

The Conjugate Gradient (CG) method is one of the oldest and best-known iterative methods and is commonly used to solve positive definite systems. The convergence of CG is good in general and can even be super-linear (rate of convergence increases from iteration to iteration) depending on the properties of the matrix [5].

L1. For i=1,2,...

L2. Solve $M * z^{(i-1)} = r^{(i-1)}$ // optional pre-conditioner

L3. $\rho_{i-1} = r^{(i-1)^T} * z^{(i-1)}$ // DDOT (1)

L4. If i=1 $p^{(1)} = z^{(0)}$ // MEM COPY

L5. Else

L6. $\beta_{i-1} = \rho_{i-1} / \rho_{i-2}$ // DDIV (2)

L7. $p^{(i)} = z^{(i-1)} + \beta_{i-1} * p^{(i-1)}$ // DAXPY (3)

L8. endif

L9. $q(i) = A * p(i)$ // SMVM (4)

L10. $\alpha_i = \rho^{i-1} / p^{(i)^T} * q^{(i)}$ // DDIV, DDOT (5)

L11. $x^{(i)} = x^{(i-1)} + \alpha_i * p^{(i)}$ // DAXPY (6)

L12. $r^{(i)} = r^{(i-1)} - \alpha_i * q^{(i)}$ // DAXPY (7)

L13. check for convergence; continue if necessary

L14. end

**Listing2-1 Conjugate Gradient (CG) Algorithm Pseudo-code (source: [5])**

It can be seen from the pseudo-code for the Conjugate Gradient algorithm (CG) in Listing2-1 that it consists of seven major operations, excluding the pre-conditioner.

Iterative methods such as Conjugate Gradient (CG) use a range of matrix, vector and scalar operations (BLAS [25]); and a full range of operations with the exception of matrix transpose used in other iterative methods can be seen in Table 2-3 and is taken as being representative of all other iterative methods for the remainder of this text.

| line | BLAS | Note | ADD SUB | DIV | MULT | FLOPS | Cycles |
|---|---|---|---|---|---|---|---|
| 3 | DDOT | dot-product | N | | N | 2N | N |
| 6 | DDIV | division | | 1 | | 1 | 1 |
| 7 | DAXPY | a*x+y | N | | N | 2N | N |
| 9 | SMVM | sparse matrix vector product | N | | NZ | N + NZ | NZ |
| 10 | DDOT DDIV | dot-product, division | N | 1 | N | 2N + 1 | N |
| 11 | DAXPY | a*x+y | N | | N | 2N | N |
| 12 | DAXPY | a*x+y | N (sub) | | N | 2N | N |
| 13 | | convergence check | | | | 2N | |
| total | | | 6N | 2 | NZ + 5N | NZ + 13N + 1 | NZ + 5N + 1 |

**Table 2-3 Conjugate Gradient Matrix Operations (source: [5])**

In Table 2-3 the symbol N denotes the matrix/vector order or number of rows or columns and NZ denotes the number of Non-Zero elements in the sparse matrix and the typical ratio of NZ to N for the large matrices considered in this work is on average greater than 20 non-zero elements per matrix row/column. FPU utilisation is the measure of how long the FPU is waiting for data rather than performing useful work. Thus it can be seen that the execution time of the entire CG (Conjugate-Gradient) method is dominated by the SMVM operation (and a solve step, if it is included) for large systems. In fact the table does not take into account that the Floating-Point Operations Per Second (FLOPS) rate achieved for the Sparse Matrix-Vector Multiplication (SMVM) depends heavily on access to external memory in a typical computer and hence the number of notional cycles and hence the real FLOPS rate may be heavily distorted by the overhead of accessing sparse matrix data. It is possible to transform the operations in the CG algorithm to run in parallel while maintaining the same behaviour [5] however this does not change and may even increase the number of operations or cycles depending on the target architecture.

Studies of the performance of Computational Fluid Dynamics (CFD) applications conducted by Anderson in [26] and Smith in [27] that show that a very low percentage (less than 5%) of the processing power in many computer systems is actually delivered to the application. Even vector supercomputers perform poorly on large Finite Element applications as shown by Taylor in [23].

## 2.3 Summary

It has been seen that the dominant element of all iterative solvers from the point of view of execution time and FLOPS performance is the Sparse Matrix by Vector (SMVM) operation, which used to solve a linear system of equations, using a large sparse matrix and a dense vector of known initial (boundary) conditions.

From the performance point of view the execution time of an iterative method such as Conjugate Gradient (CG) is dominated by the speed of the floating-point multiplication and addition operators, as well as the overhead of addressing Sparse Matrix data. The overhead of addressing Sparse Matrix data can be reduced in some cases by splitting it into dense sub-blocks but this may introduce some zero fill values where the block structure is not an exact match for the underlying sparse non-zero pattern.

This is particularly true on current general-purpose computers, where the poor performance of Sparse Matrix by Vector (SMVM) code leads to poor overall performance on Finite Element (FEM) and other applications, which depend on SMVM as their main processing step.

# 3

# Chapter 3

**"The Matrix is a system, Neo. That system is our enemy.**
**But when you're inside, you look around, what do you see?"**
**- Morpheus: The Matrix**

## 3 Sparse Matrix Storage Formats

A matrix can be described as sparse when the number of non-zero elements is very small compared to the matrix dimension. A typical sparse matrix (bcsstk13 from MatrixMarket [11] is a Fluid-flow generalized symmetric eigenvalue matrix, containing 83883 non-zero elements) structure plot in Figure 3-1 shows that storing all entries including zeroes can be highly inefficient for such matrices. In such matrices it is worthwhile storing only non-zero data elements for two reasons:

- Storing sparse matrices in dense format leads to unnecessary memory accesses and trivial floating-point operations. Using sparse data-structures and methods eliminates these trivial operations and memory accesses, while increasing performance.
- A dense storage structure for a matrix requires $N^2$ elements. Storing the matrix in sparse format minimises the memory requirements for storage of such matrices.

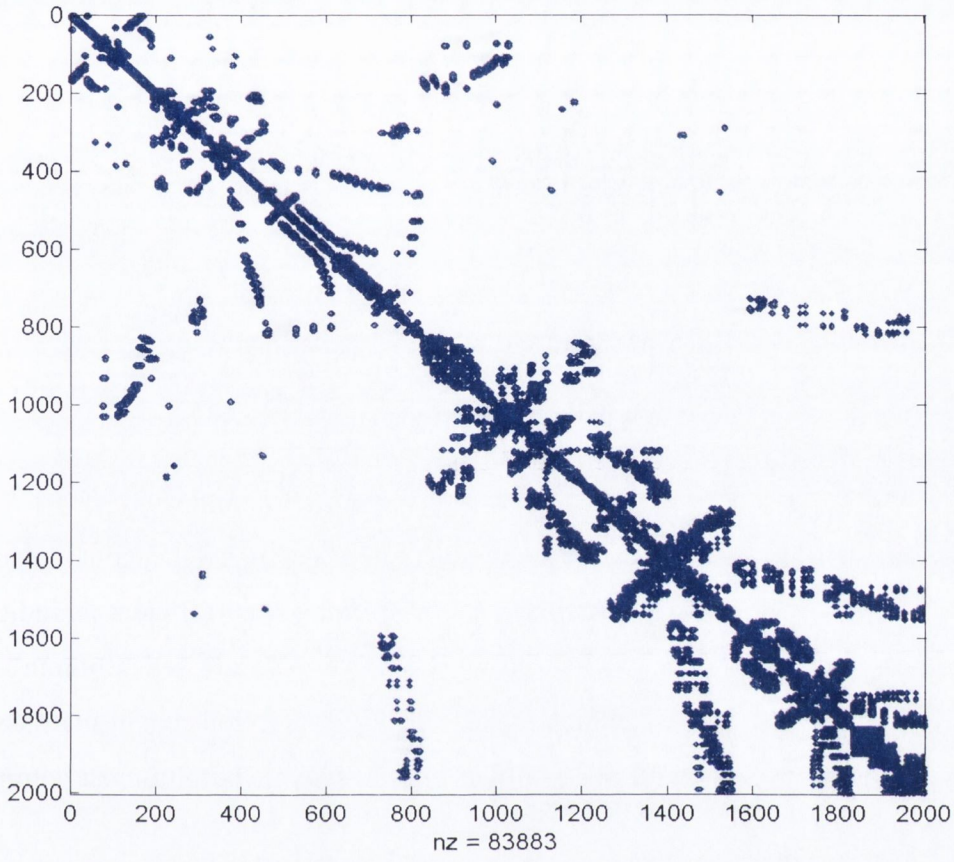



**Figure 3-1 Spy Plot of bcsstk13 Sparse Matrix (source: [11])**

A wide variety of formats for the storage of sparse matrices has been proposed. Typically such methods differ in terms of the relative amount of storage required, the amount of indirect addressing required for operations such as Sparse Matrix-Vector Multiplications (SMVM) and their suitability for execution on various kinds of single and multi-processor systems. The choice of storage method is of particular importance for iterative methods used in the solution of sparse linear systems and useful surveys of the full range of storage formats as well as the iterative methods that operate on them can be found in [1] by Barrett et al and [28] by Duff et al.

The most straightforward way of storing a sparse matrix is in coordinate format, i.e. the non-zero value is stored along with its row and column coordinates. This results in a storage requirement of 16 bytes per non-zero element (16 * NZ bytes), where row and column indices are represented by 32-bit (4-byte) integers, and the non-zero value is stored as a 64-bit (8-byte) IEEE-754 [4] double-precision floating-point number. A

typical example of a small sparse matrix stored in coordinate format is shown in Figure 3-2.

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | **1** | **2** | 0 | 0 | 0 |
| 1 | **3** | **4** | **5** | 0 | 0 |
| 2 | 0 | **6** | **7** | 0 | 0 |
| 3 | 0 | 0 | **8** | **9** | 0 |
| 4 | 0 | 0 | 0 | **10** | **11** |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| row | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 3 | 3 | 4 | 4 |
| col | 0 | 1 | 0 | 1 | 2 | 1 | 2 | 2 | 3 | 3 | 4 |
| value | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

**Figure 3-2 Matrix in Coordinate Format**

Many of the well-known sparse representations such as Jagged Diagonal require the matrix sparsity to conform to a particular pattern in order to be efficient as indicated in [1] therefore the Compressed Sparse Row (CSR) or Compressed Column Storage formats (CCS), or other formats, which make no such assumptions about matrix sparsity structure are the most general purpose formats and those which are in most common use in software libraries.

## 3.1 Compressed Sparse Row/Column (CSR/CCS) Storage

Compressed row and compressed column storage formats are the most general purpose formats as they make no assumptions about the matrix sparsity structure, yet are highly efficient as they do not store any unnecessary elements. Compressed Sparse Row (CSR) stores the matrix non-zero elements contiguously in an array in memory, along with two arrays of row and column indices related to those non-zero entries. The non-zero entries are typically represented as IEEE double-precision (64-bit) floating-point numbers whereas the row and column indices are typically represented as 32-bit integers. The non-zero entries are stored in the value array in the order they are traversed in a row-wise fashion. The column indices are stored in the col array in the same order, and the row array contains the starting indices of the rows and the non-zero entries are stored in the value array. The advantage of the CSR scheme is that 2*NZ + N + 1 elements storage are required, rather than the $N^2$ elements which would be required to store the entire matrix including zero entries, where NZ is the number of non-zero matrix entries and N

is the number of matrix rows (matrix order). The only disadvantage is that at least one index must be read from the row and/or column arrays in order to retrieve a non-zero entry and perform some operation upon it. The CSR format can support symmetric matrix storage where only the upper or lower triangular portion of the matrix above or below the matrix is stored reducing the storage requirement to approximately half of what would be required if the matrix were stored in non-symmetric format. The Matrix Market file format [11] is row-ordered and stores non-zeroes in coordinate format (row, column, value). It should be noted that Matrix Market files sometimes contain explicit zero values. A CSR representation of the sample matrix is shown in Figure 3-3.

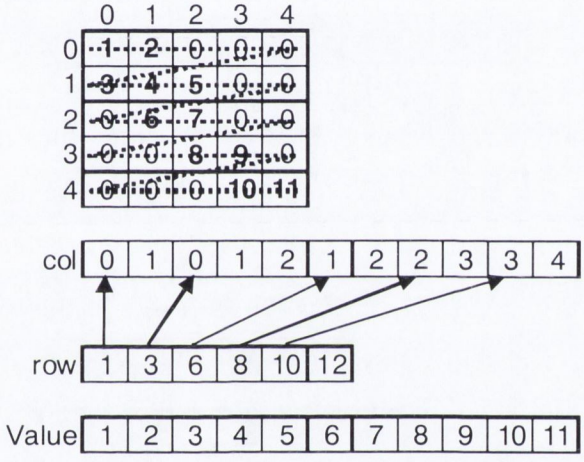



**Figure 3-3 Matrix in CSR Format**

The CCS representation of the same sample is shown in Figure 3-4.

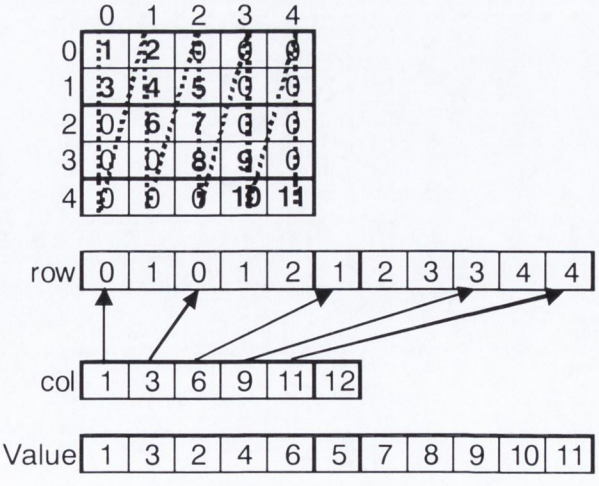



**Figure 3-4 Matrix in CCS Format**

The compressed column format is similar to CSR in that non-zeroes are stored in an array with two other arrays, which hold pointers to the beginning of matrix columns, and

indices for non-zero entries within those columns. Non-zero entries are stored in the order columns of the matrix are traversed rather than by row as in the case of CSR. The Harwell-Boeing file format [29] stores matrices in CCS format and is a good match for the FORTRAN programming language, which stores matrices in a column-wise order. As the number of non-zero elements per row or per column is typically quite low (average 20 non-zeroes per column in this work) the overhead of accessing the supporting row and column matrices represents a significant part of the cost of a matrix-vector operation. In terms of processing efficiency although in widespread use the CSR/CCS format could be improved in that it requires an indirect addressing step for each scalar operation in a matrix-vector product or pre-conditioner solve [1].

## 3.2 ICSR/ICCS Format

Incremental Compressed Sparse Row (ICSR) and Incremental Compressed Column Storage (ICCS) are variants of Compressed Row/Column storage proposed by Koster [30]. The proposed format works by observing that in Sparse Matrix-Vector Multiplication (SMVM), the inner loop performs an indirect addressing step for each iteration through the matrix data-structure, and seeks to reduce the penalty associated with it. In ICSR non-zero entries within a row (i) are stored in order of increasing column index j so the location (i,j) of a non-zero entry in the A matrix $a_{ij}$ can be stored as an offset to the previous index, and similarly for column addresses. The example matrix is shown in ICCS2 format below in Figure 3-5.

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | **1** | **2** | 0 | 0 | 0 |
| 1 | **3** | **4** | **5** | 0 | 0 |
| 2 | 0 | **6** | **7** | 0 | 0 |
| 3 | 0 | 0 | **8** | **9** | 0 |
| 4 | 0 | 0 | 0 | **10** | **11** |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| row | 0 | 1 | 0 | 1 | 2 | 1 | 2 | 2 | 3 | 3 | 4 |
| row_inc | 0 | 1 | -1 | 1 | 1 | -1 | 1 | 0 | 1 | 0 | 1 |
| offset | | | +5 | | | +5 | | | | | |
| row_inc' | 0 | 1 | 4 | 1 | 1 | 4 | 1 | 0 | 1 | 0 | 1 |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| col | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 4 |
| col_inc | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| col_inc' | 0 | 1 | 1 | 1 | 1 | | | | | | |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Value | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

**Figure 3-5 Matrix in ICCS2 Format**

The format exploits the fact that column incremental addresses do not change very often to store the increment only when it is non-zero and adds an increment n to the row-index to signal the change in column index at that point, this has the effect of reducing the memory required to store the matrix in ICCS format. The amount of memory required to store a matrix in ICCS2 format in the required 3 array structures for non-zeroes, column and row increments is $2 * nnz(A) + nnec(A) + 1$. Where nnec(A) is the number of non-zero column increments. The effect of using incremental addressing is to reduce the number of assembly language instructions in the critical loop from 26 to 15 instructions resulting in a performance gain for the SMVM operation of approximately 30%.

A second format called ICCS1 is also proposed which reduces the number of arrays required to store the sparse matrix to two, one to hold the non-zero elements and the other to hold both row and column addresses. This second format however packs row and column addresses into 32-bit integers. Deleting empty rows and columns using two permutation matrices facilitates the packing. The deletions help to ensure column increments are minimised so that row-increments can occupy the remainder of the 32-bit integer with minimal probability of exceeding the 32-bit range. The ICCS1 format thus requires $2 * nnz(A) + 1$ words. In terms of the SMVM operation additional overhead is added as the operand and result vectors must be reordered to compensate for the permutation of the source matrix. In practice the permutation is not required for iterative methods, as the non-singular matrices they generally require contain no empty columns.

## 3.3 SameType and StructType Formats

In the SameType format proposed in [31] the indices incur a performance penalty if they are not stored in the same format as the non-zeroes as they must be converted to the correct type before use by software. There is no added storage inefficiency in the case that the indices are 32-bit integers and non-zeroes are stored as 32-bit floats, however if non-zeroes must be stored as 64-bit doubles a 33% storage inefficiency results if indices are stored as doubles. If on the other hand indices are maintained as 32-bit quantities additional processing overhead is required to split 64-bit doubles into two 32-bit quantities, which are then appended, or pre-pended to the 32-bit indices. The second format proposed by Silva is called StructType and is essentially coordinate format where

each {row, col, non-zero} triplet is stored as a structure element in an array of such structures which represent the A matrix.

The advantage of StructType over SameType format is that no type conversions are required and storage is optimal as the structure can use the appropriate subtypes for each index and non-zero data element albeit at an increased cost as both row and column elements are stored for each non-zero. According to Silva combining indices and non-zeroes into a single contiguous data-structure reduces cache misses by improving spatial locality, and hence increases system performance. One way to get around alignment problems introduced by the miss-match between 32-bit integers and 64-bit double-precision numbers would be to store two 64-bit doubles and two 32-bit doubles in a single structure, the only disadvantage being that the final integer/double pair might require padding with zeroes in some cases.

## 3.4 Hierarchical Sparse Matrix (HiSM) Format

In [32] Vassiliadis et al proposed a storage format which reduces the storage required for row and column addresses by using a hierarchical storage scheme. The hierarchical storage and an underlying 8x8 vector machine architecture saves address overhead by factoring common terms out of addresses, reducing them to approximately 3-bits from an initial 32-bits. A matrix is converted to HiSM format by subdividing it into multiple s*s sub-matrices where s is the dimension of sub-matrix which can be handled by the associated vector architecture. A simple example of the HiSM format with s=4 is shown in Figure 3-6.

The large source matrix is tiled up into four 4x4 matrices as it will not fit in a single 4x4 matrix so one level of hierarchy is required. The top data-structure contains pointers to the 4x4 sub-matrices and a length (len) entry, which tells the HiSM hardware whether the sub-matrix is empty or not. Each one of the 3 sub-matrices which have non-zero lengths (s0, s2 and s3) are then represented using 2-bit row and column addresses, and corresponding non-zero value for each non-zero entry in the source matrix. The use of s=4 reduces addresses to 2-bits in this example however the value proposed by Vassiliadis in [32] for s is 64 resulting in 3-bit address references.

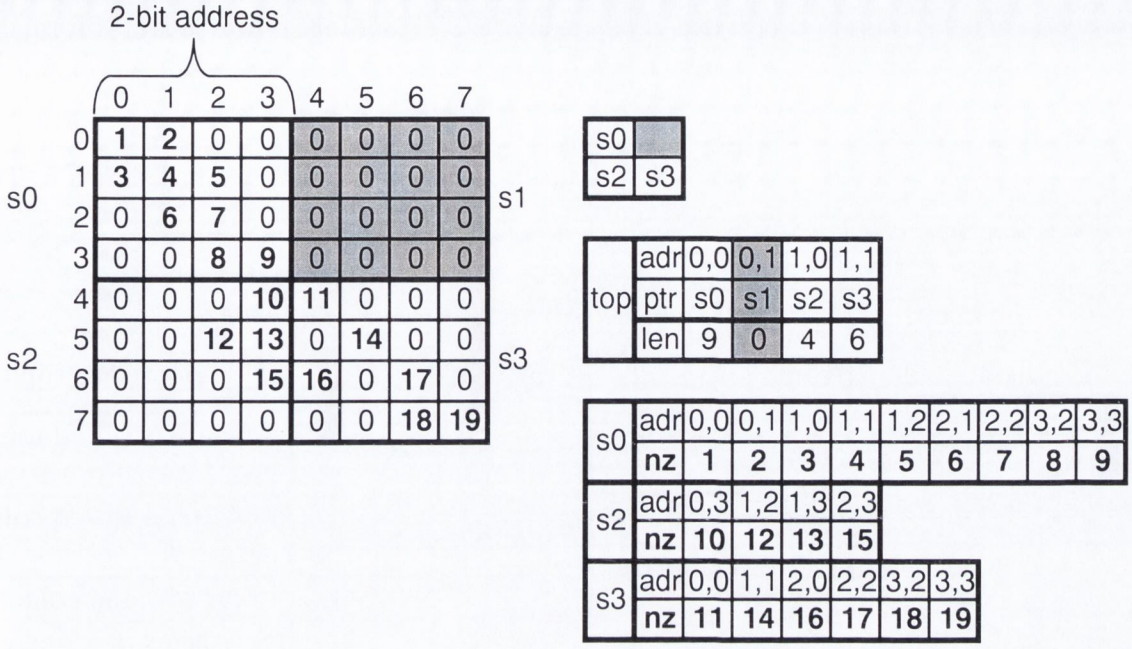



**Figure 3-6 HiSM Format Storage Example**

The main benefit of the scheme is a reduction of almost 28% in the overall storage requirement owing to the use of two 3-bit references rather than 32-bit values as in CSR. One disadvantage with the format is that it requires square matrices for which the number of rows/columns N is an integer multiple of s for optimum efficiency. Additionally multiple levels of hierarchical storage (L) and look-ups are required as the total number of non-zeroes NZ grows as shown in Figure 3-7, however most problems should require at most 5 levels of hierarchy in order to contain the complete matrix ($64^5$ = 1000M non-zero sparse matrix entries).

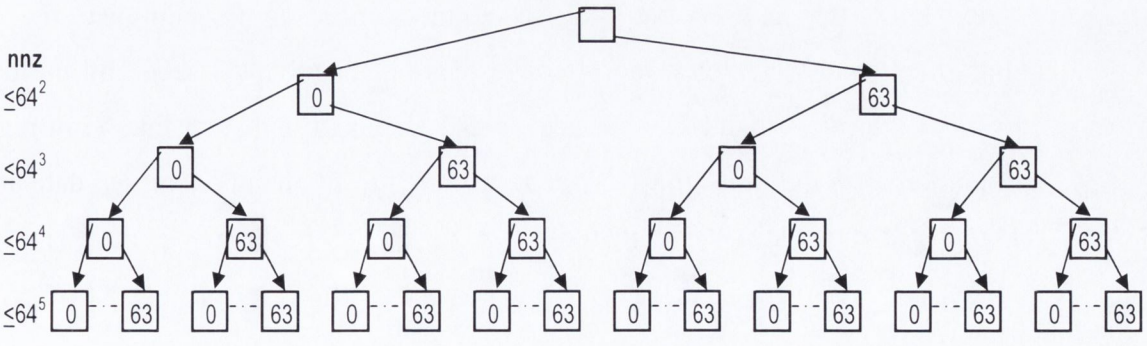



**Figure 3-7 Levels of HiSM Storage Hierarchy Required**

Typically however, as stated in [32], the HiSM format achieves between 72 and 76% of the CSR memory requirement for storage of the same matrix. It is also claimed to increase simulated Sparse Matrix-Vector Multiplication (SMVM) performance by up to

5.3 times when compared to CSR on a Generic Vector Processor (GVP). This is achieved using an architectural extension to GVP optimised for the HiSM format.

## 3.5 Summary

The sparse matrix storage formats surveyed have the characteristics shown in Table 3-1, where the A matrix is sparse and of dimension n rows by m columns.

| Format | Assumptions | Memory Words | Notes |
|---|---|---|---|
| Coordinate | None | 3*nnz(A) | entries stored as triplets |
| CRS | None | 2*nnz(A)+m+1 | non-zeroes stored row-wise |
| CCS | None | 2*nnz(A)+n+1 | non-zeroes stored column-wise |
| ICCS1 | Assumes column increment small | 2*nnz(A)+1 | Stores row and column increments in a single array |
| ICCS2 | None | 2*nnz(A)+nnec(A) | Only non-zero column increments nnec(A) are stored to reduce memory |
| SameType | None | 3*nnz(A) | less efficient for double-precision non-zeroes |
| StructType | None | 2*(n+nnz(A)) | Better performance than SameType at cost of increased storage |

**Table 3-1 Features of a Selection of Sparse Matrix Storage Formats**

A key issue for any Sparse Matrix storage format is to minimise the memory requirements and to increase the effective memory-bandwidth "seen" by the data-path, equally any format should seek to maximise data-locality whether that be in registers or cache in order to minimise the average access-time to frequently accessed data and hence maximise performance.

# 4

# Chapter 4

*"I know nothing by experience,*
*though I know something by observation"*
*- Lord Goring: "An ideal husband" (Oscar Wilde)*

## 4 Hardware Support for SMVM

According to Kogge [15] there are three approaches to performing Sparse Matrix-Vector Multiplication (SMVM) on programmable computers:

- Inner Product where the result vector y is computed one inner-product at a time, using $a_{ij}$ and $x_j$ values read from memory
- Sub-matrix where the matrix *A* is tiled up into sub-matrices which are read along with equally sized segments of the y and x vectors
- Column Scaling where multiple entries in the result vector y are read and updated as each column is scanned for $a_{ij}$ values and multiplied by a single entry from the *x*-vector $x_i$

In the following sections a survey of hardware performance enhancement techniques relevant to the proposed architecture is presented.

### 4.1 Hardware Performance Enhancement Techniques

There are two main methods of increasing single-core processor performance:

- Raising the processor clock-frequency through pipelining and process scaling
- Increasing the number of executed Instructions Per Clock (IPC)

These two techniques are highly interdependent and it is difficult if not impossible to increase IPC and processor clock frequency simultaneously. A review of the important features in terms of processor performance, which are relevant to all architectures covered in this work, is given in the following sections.

Of course multiple processors can be employed in parallel using one or both techniques in order to achieve greater speed-ups assuming enough of the code can be restructured to take advantage of parallelism in a multicore processor, and such multicore processors will be dealt with later in this section.

### 4.1.1 Processor Pipelining

According to Hennessy in [34] pipelining is an implementation whereby multiple instructions are overlapped in execution. The reason for pipelining is that the amount of work which can be achieved in a single clock cycle is limited by the number of logic levels required to implement a desired logical function (work) and the processing delay associated with each level of logic. The circuit shown in **Figure 4-1**, produces a complete result y = a+b*c each clock cycle i.e. the circuit has a clock latency of 1.

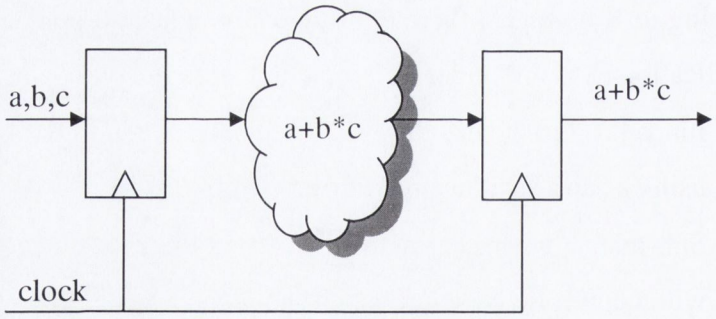



**Figure 4-1 Un-pipelined Circuit**

In a pipelined approach a large logic function consisting of many logic levels which evaluates in a single clock cycle is partitioned into multiple groups, each containing fewer levels of logic which can evaluate in parallel in a single shorter clock cycle. This divide and conquer approach ensures that the maximum operating frequency can be achieved with a given fabrication technology. The weakness of frequency scaling alone is that it does not take effects such as start-up delays and stalls into account and is thus a

poor measure of processor performance. A pipelined version of the previous circuit is shown in **Figure 4-2** and has a clock latency of two. In this example because the multiplier and adder have been separated into two pipeline stages the circuit can now run at a speed dictated by the propagation delay through the adder or the multiplier, whichever is slower, rather than the sum of the two delays. In practice in a real design every effort is made to balance the delays of all pipelining stages in order that the whole design is able to run at the maximum frequency possible rather than being limited arbitrarily by a single stage.

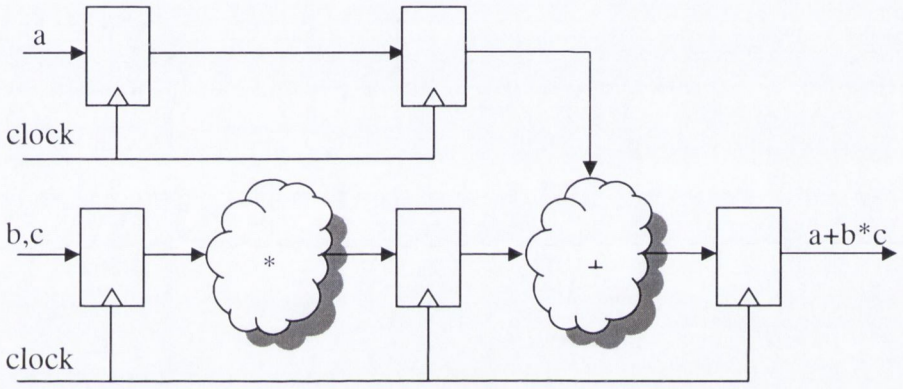



**Figure 4-2 Pipelined Circuit**

A disadvantage of this configuration is that it takes two clock cycles for the result to appear at the circuit rather than one as in the non-pipelined case. This increased latency is important as the pipeline has first to be filled before starting to produce results, and also has to be refilled following a pipeline hazard or stall. The pipeline stall penalty eats into the raw performance gains achieved by pipelining:

$$Performance - Gain = \frac{CPI_{unpipelined} * Clock_{unpipelined}}{CPI_{pipelined} * Clock_{pipelined}}$$

**Equation 4-1 Performance Improvement due to Pipelining**

CPI (Clocks per Instruction) is the average number of clock-cycles taken for a given instruction to execute. Computer architects and processor designers often use a technology independent metric based on a fanout-of-4 inverter (FO4), where an inverter drives 4 identical copies of itself, as a means of comparing processor implementations as shown in **Figure 4-3**.

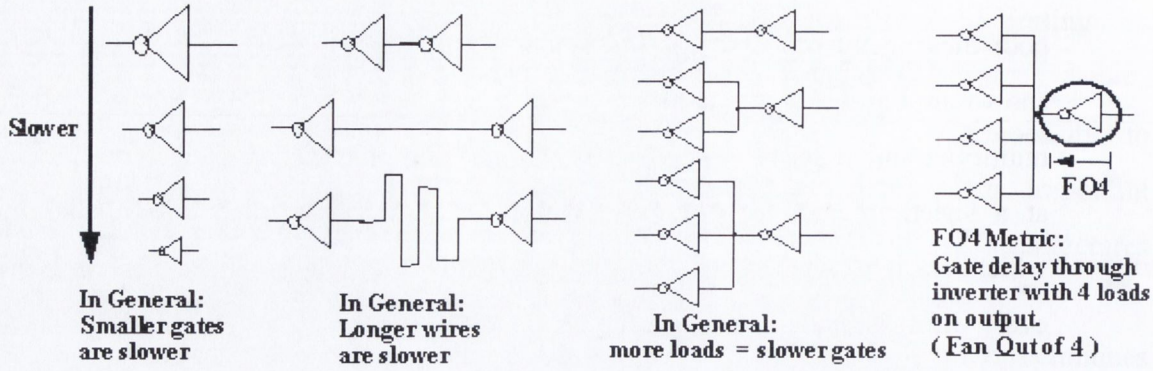



**Figure 4-3 Fanout-of 4 (FO4) (source: [36])**

From the pure performance point of view it has been shown by Hrishikesh in [37] that there is an optimal amount of logic which can occur between two clocked registers yielding maximum performance in terms of clock frequency, and that current commercial microprocessor design has almost reached this optimum point as shown in **Figure 4-4**.

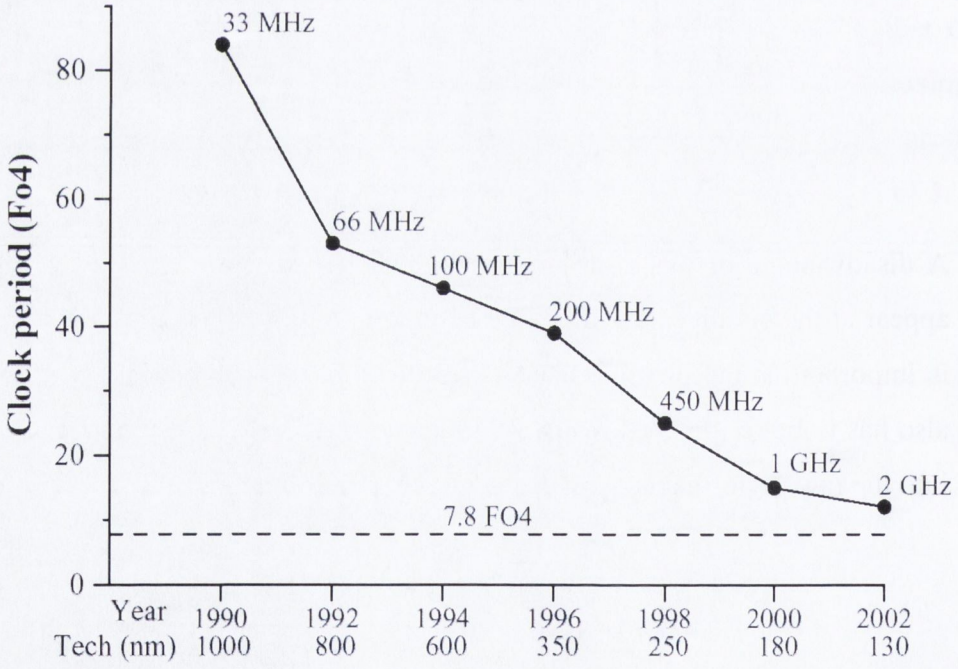



**Figure 4-4 Pipelining as a Function of FO4 Delay (source: [37])**

In fact without the use of specialised structures and process technology FO4 will actually start to increase from 2010 onwards according to Tanabe [39] as can be seen in **Figure 4-5** unless improvements to structures and process technology are made at the transistor level.

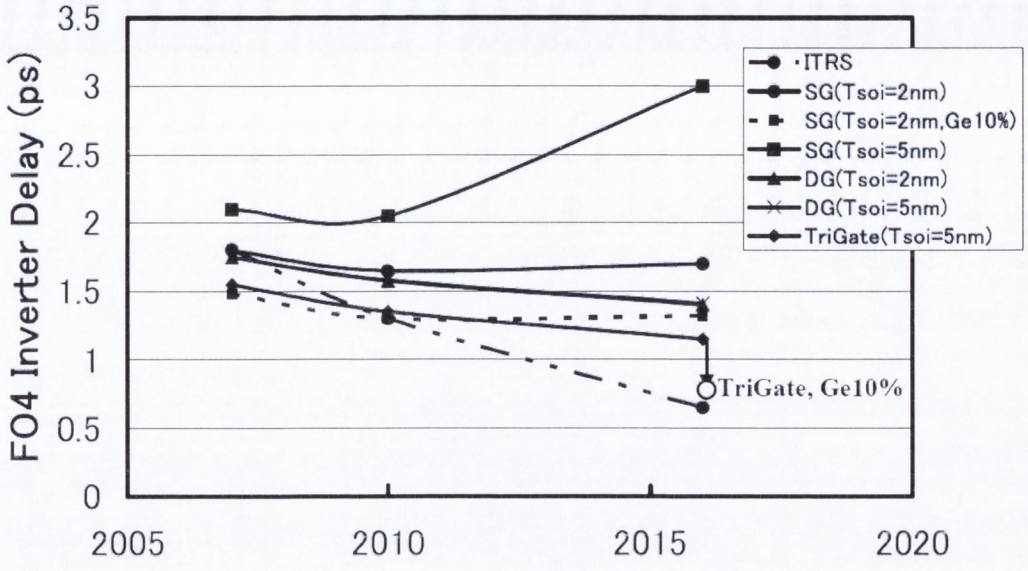



**Figure 4-5 FO4 scaling (source: [39])**

In **Figure 4-5** Tanabe simulates the behaviour of 3 different transistor structures under scaling to predict FO4 inverter circuit delay characteristics compared to the ITRS roadmap. The 3 devices are Single Gate (SG), Dual Gate (DG) and Trigate transistors which are being introduced to mitigate the Short Channel Effect (SCE) which is increasingly important as devices continue to scale below 65nm. These devices are compared to ITRS predictions of pure scaling of existing transistor structures. As can be seen the SG device only produces a delay improvement below 2nm body thickness. On the other hand the DG delay is superior at 5nm body thickness. This difference is mainly due to the controllability of short channel effect (SCE). Compared to the ITRS roadmap predictions for delay characteristics it was found that, at 65nm node only SG SOI structures and, at 45nm node, SG SOI + strained-Si channel device or TriGate device and, in 22nm node, TriGate + strained-Si channel device will meet the ITRS predictions. According to Sprangle [40] continuing to pipeline into the future will yield about 65% of the theoretical performance improvement predicted by scaling at a cost of increased cache bandwidth requirements. It is ultimately power, and the associated problem of cooling, which places a limit on the amount of pipelining which is employed in a given design. The other and highly important implication of pipelining is that it increases the number of clocked elements in the design and thus increases power:

$$P \cong 2 * f * C_l * Vdd^2$$

**Equation 4-2 CMOS Dynamic Power**

Where $f$ is the frequency of the clock applied to the pipeline stages, $C_l$ is the load capacitance driven by each pipeline stage and $V_{dd}$ is the power-supply voltage.

With the transition to ever deeper process geometries logic delay (FO4) has become increasing less important with respect to interconnect delay as a proportion of overall delay, as shown by Horowitz [38]. This means that any improvements to FO4 delay due to more complex and expensive process technology will at best cancel out the effects of increasing interconnect delay.

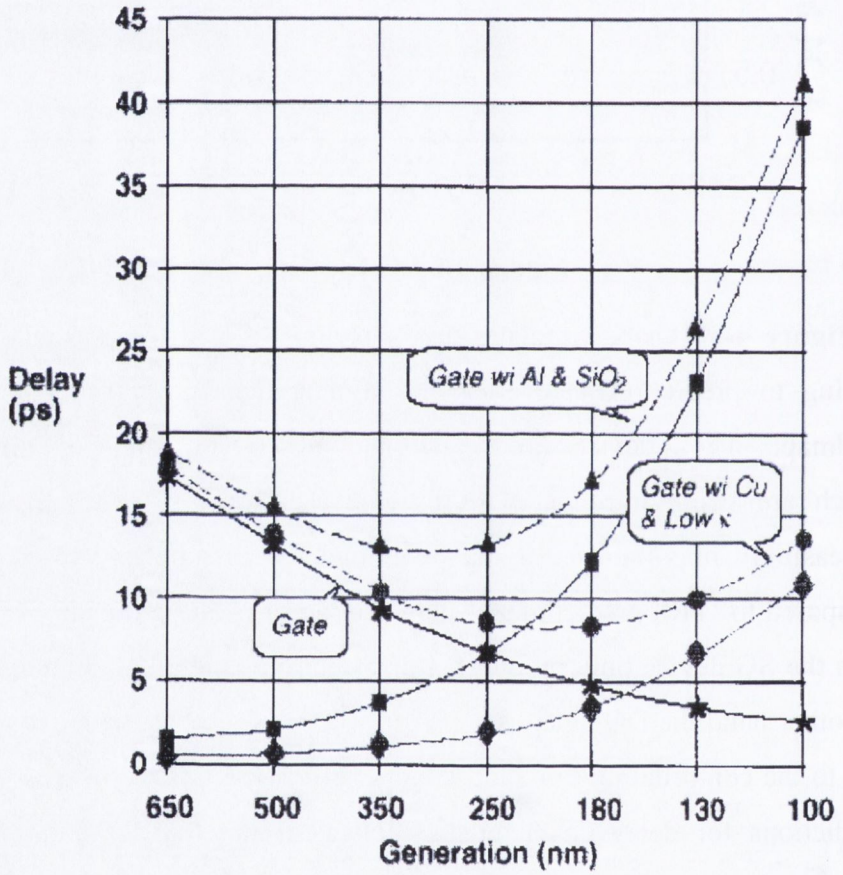



**Figure 4-6 Logic vs. Interconnect Delay Scaling (source: [38])**

It can be concluded that the increase in FO4 delay as well as interconnect delay, and the buffering to compensate for voltage drops (I*R) due to increasing wiring resistivity means that smaller, simpler multicore architectures will guarantee better performance as technologies continue to scale, while limiting power dissipation.

### 4.1.2 Pipeline Hazards

A pipeline hazard is a condition which prevents the next instruction in a computer program from executing in its designated clock cycle. Reasons for a hazard and consequent stall could include a cache miss, or a data-dependency where two instructions operate on the same register and the earlier issue instruction has not yet written back the register contents due to the pipeline latency. Such a data-dependency can occur where two instructions which operate on the same data in a register, memory or cache location are issued closer together in terms of their dispatch cycle than the pipeline is long in terms of stages, where each stage executes in a single cycle.

According to Hennessy [34] three types of pipeline hazard exist:

- Structural Hazards occur where resource limitations mean that not all combinations of instructions can be executed simultaneously in the pipeline.
- Data Hazards occur when an execution of an instruction depends on the results from a previous instruction in a way that is exposed by overlapping instructions in the pipeline.
- Control Hazards occur through the effects of pipelining instructions such as branches which modify the Program Counter (PC)
- Data hazards can be further subdivided [41] into the following sub-classes:
- Write After Write (WAW) hazard where one part of the processor pipeline incorrectly overwrites a location *i* resulting in an incorrect value being stored in *i*
- Read After Write (RAW) hazards where a data dependency exists in a pipeline such that an early stage attempts to read location *i* before it has been updated by a later stage, resulting in the old value of *i* being used incorrectly
- Write After Read (WAR) hazards where a pipeline stage writes to *i* before the old value contained in *i* is read, resulting in the new value being read incorrectly

In a typical processor with an In-Order pipeline, dependencies in the linear execution pattern can result in hazards which make it necessary to stall the pipeline, allowing instructions issued before the stall to complete, and the stall to be cleared, before refilling the pipeline and allowing those instructions issued after the stalled instruction to complete. The Out-Of-Order (OOO) technique allows execution to proceed in cases where a normal in-order pipeline would stall by using multiple functional units which can execute in parallel. OOO execution circumvents some of the problems associated with

In-Order pipelines by "looking" at a large window of instructions to be executed which are present in a large instruction pre-fetch buffer and executing those without dependencies in parallel. OOO designs have largely fallen out of favour due to their high implementation and have been replaced by multiple in-order pipelines in Chip Multiprocessors which will be discussed in more detail in the following sections.

RAW data-hazards occur frequently in SMVM codes. According to Taylor [23] over 20 instructions, and possibly many more, would be required in the instruction pipeline in order to avoid stalls due to RAW hazards in SMVM applications. However in such applications there is very little in the way of data reuse, therefore in many cases the data associated with those later instructions might not be in the cache resulting in a cache-miss which would negate any performance gain due to out-of-order execution. For the purposes of this work we will limit ourselves to simple in-order pipelines as Out-of-Order techniques do little to improve Sparse Matrix Vector Multiplication where the main limitation is memory bandwidth and not stalls.

### 4.1.3 Floating-Point Unit (FPU)

Direct SMVM operation on the CSR format entails that the solution-vector y is calculated one entry at a time, as a sum of the products of A-matrix entries and x-vector entries. The x-vector entries are obtained by indirect reference using the addresses stored as part of the sparse matrix A as shown in **Table 4-1**.

$$
\begin{aligned}
y_0 = {} & a_{00} * x_0 + \\
& a_{01} * x_1 + \\
& a_{02} * x_2 + \\
& a_{03} * x_3 + \\
& a_{04} * x_4
\end{aligned}
$$

| $y_0$ |
|---|
| |
| |
| |
| |

| $a_{00}$ | $a_{01}$ | $a_{02}$ | $a_{03}$ | $a_{04}$ |
|---|---|---|---|---|
| | | | | |
| | | | | |
| | | | | |
| | | | | |

| $x_0$ |
|---|
| $x_1$ |
| $x_2$ |
| $x_3$ |
| $x_4$ |

**Table 4-1 CSR Sparse Matrix Vector Multiplication Order**

As can be seen from **Figure 4-7** one of the difficulties with the CSR format is the dependency on the previous y-register value which introduces a RAW hazard if y is calculated incrementally using a single floating-point multiplier and adder pair.

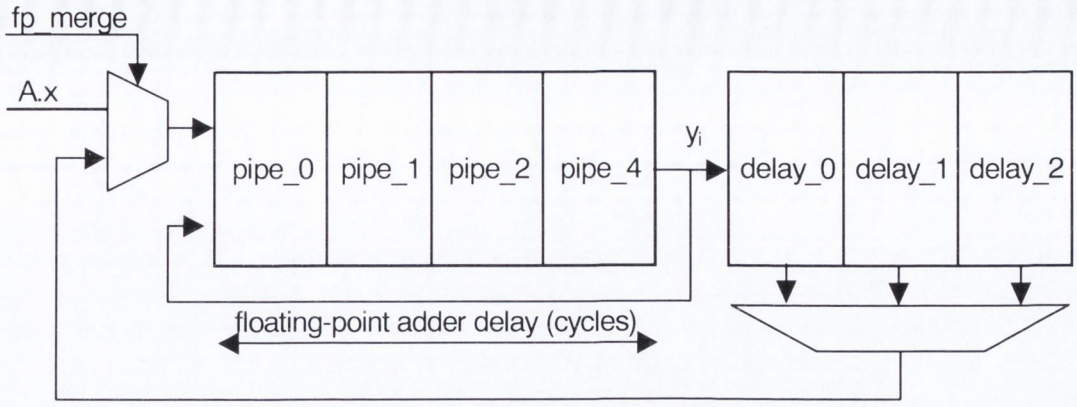



**Figure 4-7 FP Adder Configuration (CSR)**

One approach to resolving the dependency is to stall as each product reaches the floating-point adder input, however this approach results in low performance. Another approach is to allow multiple sums of products to be accumulated independently, one in each of the pipeline stages in the floating-point adder, however this requires additional cycles at the end of a sequence of computations to propagate out the results adding additional cycles of delay. At the end of the last multiplication the sums of products must be merged into a single y-vector entry by re-circulation through the adder until the terms {W,X,Y,Z} from each pipeline stage have been summed together to produce a single vector entry as shown in **Table 4-2**. The only disadvantage with this approach is the incremental delay required which adds to the start-up overhead for each matrix row. In general an n-stage floating-point adder pipeline will require approximately $n * (\log_s(n) + 1)$ cycles to merge through its own pipeline (2 operand adder) where n is a power of 2. This delay can be mitigated but not eliminated by overlapping the propagation delay with the next set of computations, and maintaining intermediate results in the pipeline adds to register pressure in the processor.

| cycle | pipe_0 | pipe_1 | pipe_2 | pipe_3 | delay_0 | delay_1 | delay_2 |
|---|---|---|---|---|---|---|---|
| | W | X | Y | Z | | | |
| | | W | X | Y | Z | | |
| | | | W | X | Y | Z | |
| 1 | Y+Z | | | W | X | Y | Z |
| 2 | | Y+Z | | | W | X | Y |
| 3 | W+X | | Y+Z | | | | |
| 4 | | W+X | | Y+Z | | | |
| 5 | | | W+X | | Y+Z | | |
| 6 | | | | W+X | | Y+Z | |
| 7 | | | | | W+X | | Y+Z |
| 8 | W+X+Y+Z | | | | | | |
| 9 | | W+X+Y+Z | | | | | |
| 10 | | | W+X+Y+Z | | | | |
| 11 | | | | W+X+Y+Z | | | |
| 12 | | | | | W+X+Y+Z | | |

**Table 4-2 CSR FP Adder Pipeline Merge Operation**

The SPAR architecture is proposed by Taylor [23] as a solution to some of these problems and consists of two linear arrays, one of non-zero values and the other of row or column addresses. Zero entries are selectively introduced into the value array to demarcate the end of columns in the row/column array. The SPAR representation of the sample matrix in Figure 3-2 is shown in **Table 4-3**.

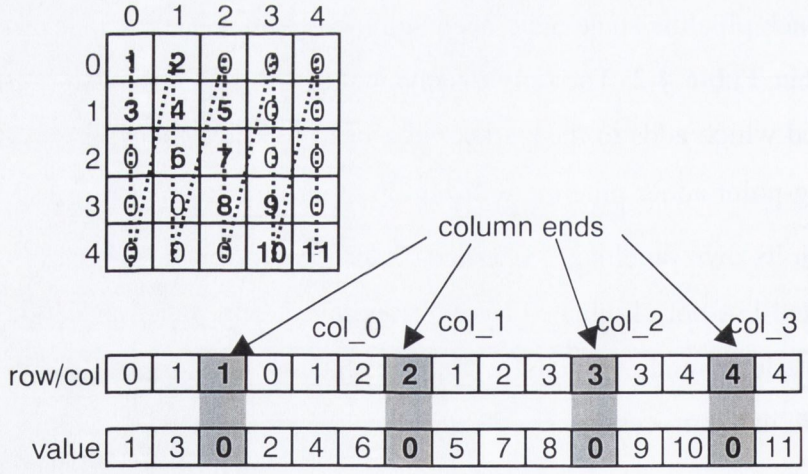



**Table 4-3 Matrix in SPAR Format**

As a result the SPAR data-structure consists of one very long vector instruction meaning that the vector start-up overhead cost is only incurred once per sparse matrix-vector multiplication. This however comes at an increased cost in terms of the additional storage required to store N*64-bit double-precision zero entries, where N is the number of columns in the sparse matrix. The increase in storage requirements for a matrix

containing 24 non-zero entries per column is 4 bytes per column (64-bit 0.0 value used to denote end of column instead of 32-bit integer) or approximately 2.7%.

In order to improve data-locality as in the StructType sparse matrix format detailed in Section 3.3 the Sparse Matrix is stored as an array of structures as shown in **Listing4-1**.

```
struct spar_entry {
  double k;
  int      r;
};
```

**Listing4-1 SPAR Data-Structure**

Corresponding to the SPAR data structure the code to implement an unsymmetric Sparse Matrix-Vector Multiplication is shown in **Listing4-2**.

```
while (i < A.max_entries) {
   if (A.k[i]==0.0) i_col = A.r[i]; // 0.0 marks end of column
   else {
     i_row = A.r[i];
     y[i_row] += A.k[i]* x[i_col];
   }
   i++;
}
```

**Listing4-2 Unsymmetric SPAR SMVM Code**

A block diagram of SPAR is shown in **Figure 4-8**, and one of the main features is the inclusion of an 8kB direct-mapped Y-cache, which was sufficient to achieve their target 95% cache hit-rate for the suite of test-matrices. In the diagram two separate SDRAMs interfaces allow the vector data to be accessed via the R-AGU (address generation unit) and 64-bit double precision A-matrix data via the K-AGU. When a zero-value is found in the A-matrix data (value array) by the ZDL (zero delay line) block it generates a signal which is used to load the x-vector value into the p[i] register from the same SDRAM connected to the R-AGU. Storing the column and row addresses in the same array in SDRAM thus allows a reduction in the complexity of the SPAR hardware compared with a hardware CSR multiplier. The A-matrix values in a column are multiplied successively by the x-vector value stored in the p[i] register by a floating-point multiplier and each partial product is added as a contribution to the y-values stored in the cache block. In this case the cache used was a 8kB direct-mapped cache. Delays are included in the circuit in order to compensate for the delay through the floating-point adder so that the

partial product from the multiplier is added to the correct y-value in the cache. The cache is a write-back cache meaning that 64-bit results from the cache are only written back to the third SDRAM when data is evicted from the cache owing to a cache miss or when the cache is explicitly flushed from the cache at the end of the SMVM operation. Conserving SDRAM bandwidth in this manner leads to higher performance and lower power as data is only written to or read from the external SDRAM when absolutely necessary.

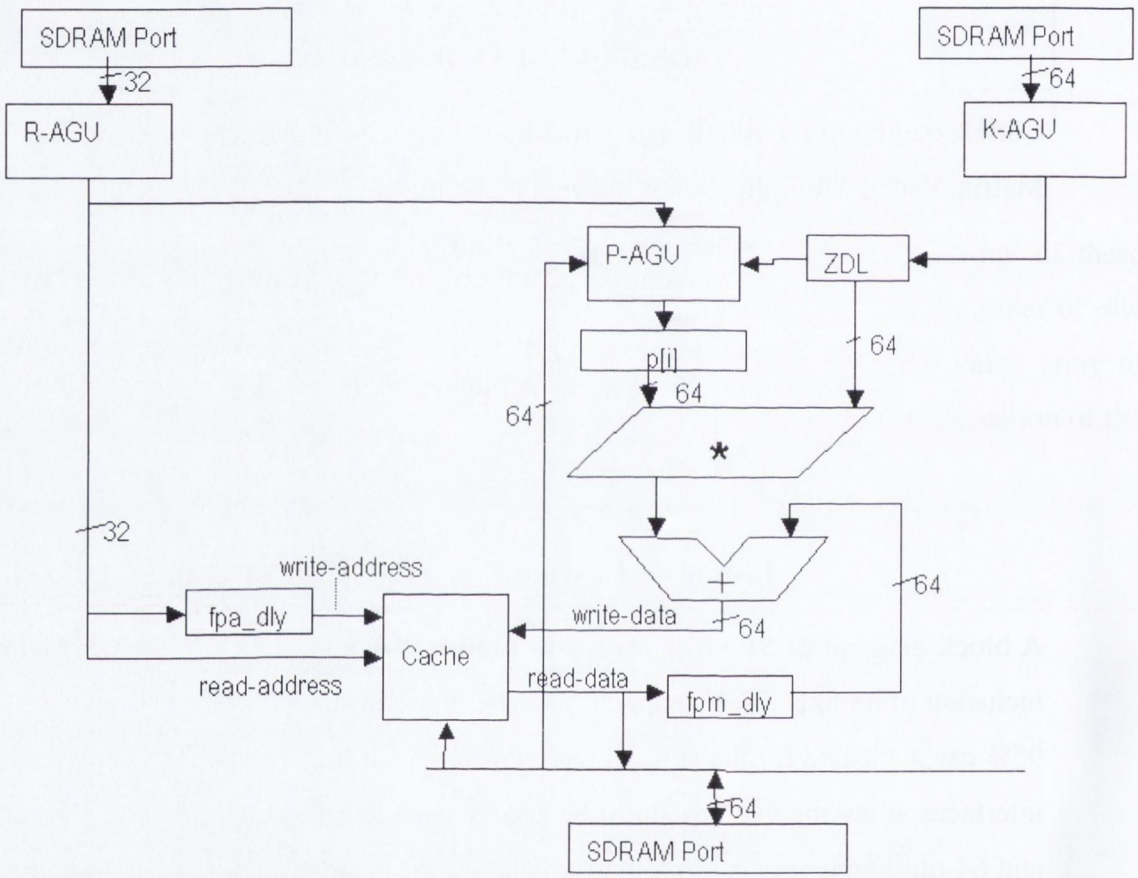



**Figure 4-8 SPAR Architecture (source: [23])**

The authors found that for even moderately sized FEM matrices the FPU utilisation for the SPAR architecture is approximately 96% irrespective of start-up delay, assuming that pipeline stalls can be eliminated, and that the effect of the start-up delay on FPU utilisation is negligible for large sparse matrices.

The SPAR simulation model used by Taylor is ideal, however, in that there is an assumption that all stalls due to RAW (read after write) hazards can be eliminated by reordering the non-zero entries in a software pre-processing step, prior to starting the

SPAR Sparse Matrix-Vector Multiplication (SMVM) operation. However the authors do not describe how the elimination of RAW hazards can be guaranteed or the computational overhead in pre-processing the sparse matrix to eliminate hazards.

In practice a single FPU even with the enhancements proposed by Taylor still has disadvantages in that instruction bandwidth is high and the number of floating-point operations that can be carried out per clock-cycle is low. A better solution is to compute the products corresponding to a partial row or column from the Sparse Matrix in a single cycle. Considering a 4-element Single Instruction Multiple Data (SIMD) datapath capable of processing a 4-entry segment of a sparse row or column the instruction bandwidth is reduced by a factor of 4 and the number of FLOPS per cycle is multiplied by 4 for the same clock frequency, thus taking better advantage of process-technology in line with Moore's law.

### 4.1.4 Memory

A second area of focus in this work is that of achieving high performance where external memory bandwidth is limited. It will be seen that this is especially important in the case of Sparse Matrix Vector Multiplication (SMVM). The reason for this is that in modern Very Large Scale of Integration (VLSI) processors internal memory and bus bandwidths and processing speed in Floating Point Operations per Second (MFLOPS) are very high and have continued to grow as VLSI manufacturing processes have improved, however pin bandwidths required to interface to external memory devices have not kept pace with internal processor bandwidths giving rise to a performance bottleneck. External memories are preferred over internal memories as they are low-cost commodity products. The cost per megabyte (MB) of external DRAM chips is at least one order of magnitude cheaper than to integrate the same DRAM into a processor die, and in the case that SRAM is integrated onto the processor the penalty is even worse. A second reason for using external memories is that they allow a machine to be easily upgraded to support problems of arbitrary size, which cannot be handled in internal memories.

McKee coined the term "Memory Wall" [42] to describe the situation where improvements in processor speed will eventually be masked out by slower improvements in commodity DRAM speed and read/write latency. Processor performance has followed Moore's law because it has harnessed improvements in both technology and in architectural design. On the other hand most bandwidth improvements in DRAM

technology were achieved by low-cost incremental improvements to existing DRAM architectures, relying mainly on process scaling for any improvements in speed, which also resulted in lower DRAM latency as a by-product.

It is expected that computing systems performance will soon be dominated by memory at a 7% performance increase per year (as opposed to processor speeds which have been growing conservatively at 50% per annum), leading to an increasing disparity between memory and processor bandwidth as shown in **Figure 4-9**. As can be seen the use of multiple processors exacerbates the problem as multiple processors contend for access to a shared memory subsystem.

According to Jacob [44] hundreds of papers in recent years have looked at the problem in terms of what can be done on the CPU side to tolerate or reduce memory latency. Memory latency is the time, typically measured in clock cycles taken to access a randomly addressed instruction or data in memory.

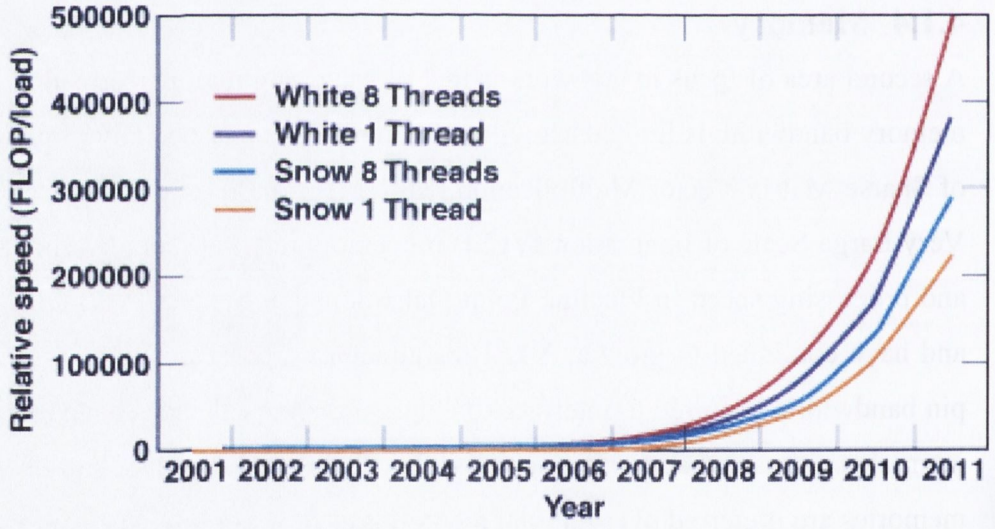



**Figure 4-9 Diverging Processor and Memory Speeds (source: [43])**

Memory bandwidth on the other hand is the rate at which those words can be delivered to a waiting processor. In the case of large data-structures such as sparse-matrices the memory latency penalty is incurred once when accessing the beginning of the structure in memory, however thereafter data/instructions are delivered to the processor at a rate determined by the available memory bandwidth. For large sparse matrices the effect of limited memory bandwidth far outweighs the initial penalty due to the latency in addressing the start of the sparse matrix. Unfortunately most of the improvements in general purpose programmable computers address latency reduction and tolerance, rather

than increasing memory bandwidth and hence do not directly address the requirements of this work.

The best predictor of sustained memory bandwidth in computer systems has been McCalpin's STREAM benchmark [45], which is based on the measured sustainable memory bandwidth available to ordinary user programs rather than on theoretical "peak bandwidth". As reported in [33] STREAM benchmark results illustrate the imbalance between peak MFLOPS and sustainable bandwidth, with peak MFLOPS increasing at 50% versus the 35% per annum for sustainable bandwidth for the same architectures.

Additional work by Asanovic [46] shows that the STREAM benchmark [45] results often fall far short of the available pin-bandwidth, with some machines achieving 1/6 of the available pin-bandwidth in terms of STREAM benchmark performance. This indicates an architectural imbalance between the processor core design and the I/O subsystem which does not succeed in matching the processing bandwidth with the available I/O bandwidth. Oliker et al [48] suggest that current processor architectures fail to offer a sustainable path for improvement as processor core speeds continue to outstrip reductions in memory subsystem latency, and that there is a critical need for future microprocessors to add architectural enhancements to address applications exhibiting significant levels of memory access irregularity.

### 4.1.5 Cache

Modern processors include at least some onboard memory as part of their hierarchical memory subsystem with the latency and speed of read/write operations increasing with the distance from the processor. Onboard memory is normally integrated in the form of instruction and/or data caches whose function is to improve processor average performance. A cache is a local fast memory, which contains copies of frequently accessed data or instructions, and works by reducing the average access time the processor incurs when accessing memory as shown in Equation 4-3.

$$t_{worstcase} = t_{external} \text{ to } t_{avg} = t_{cache} * hitrate + t_{external} * missrate$$

**Equation 4-3 Memory Access-Time**

Caches work by exploiting the temporal and spatial locality of addresses and their associated data or instructions:

- Temporal locality is the likelihood that a recently referenced address will be referenced again in the near future
- Spatial locality is the likelihood that a close neighbour of a recently referenced address will be referenced in the near future

The processor architecture is organized so that it looks in the fast local cache for data/instructions before looking in the next level of the hierarchy and so on. If the desired data/instruction is found in the cache a cache "hit" is said to have occurred, otherwise a cache "miss" occurs. Temam and Jalby [48] divide cache misses in 3 classes:

- Cross-interference (conflict) misses - line flushed by element from another array
- Self-interference (capacity) misses - line is flushed by another element of the same array
- Intrinsic (compulsory) misses - line is loaded for the first time

The cache hit-rate depends both on its type, its size, the number of cache-lines and the cache line-length.

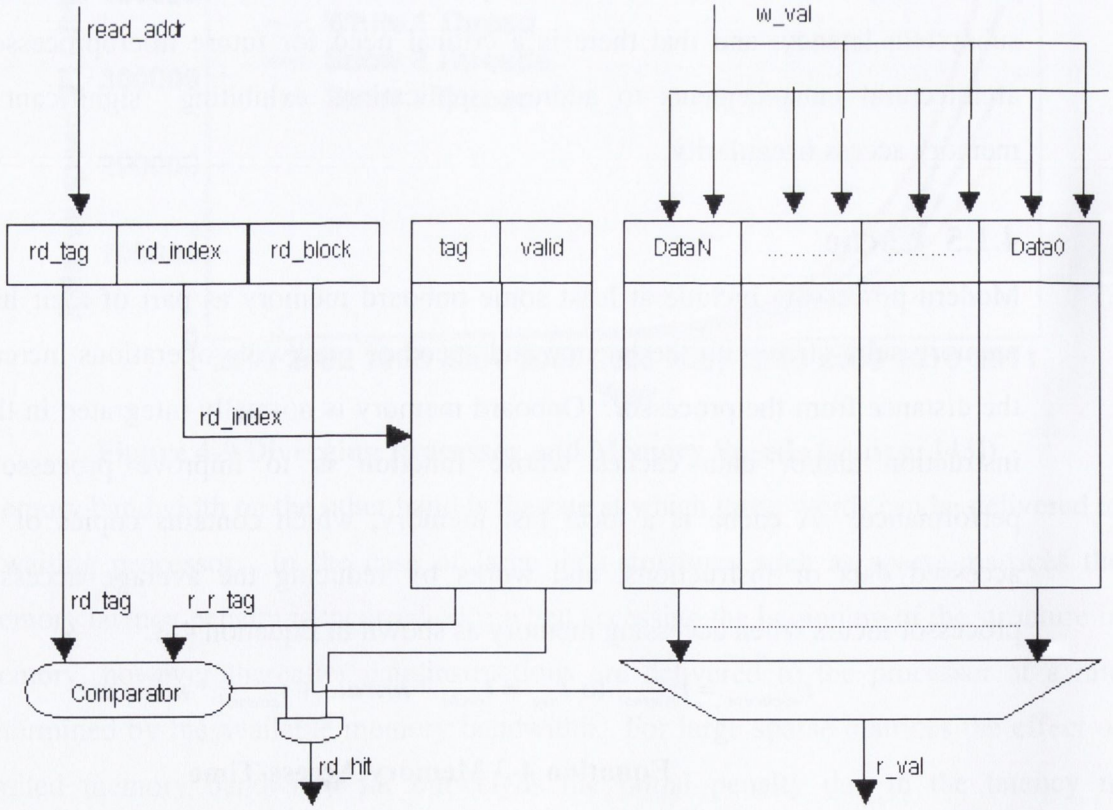



**Table 4-4 Direct-Mapped Cache Block diagram (source: [50])**

Data and Instruction Caches of the type used in General Purpose Processors are typically characterized by 3 major parameters:

- Capacity
- Cache line-length
- Associativity

In general, caches consist of several sets, each of which consists of n ways, where n is called the cache associativity, and is usually 1, 2, 4, or 8. Caches are called direct mapped for n=1, set associative otherwise. It is also possible albeit expensive to implement a fully associative cache. A direct-mapped cache such as that shown in **Table 4-4** has an associativity of 1. An associativity of 1 means each address maps to one, and only one location in the cache. The benefit of a direct-mapped cache design is that it is very simple and can make use of conventional RAM as building-blocks resulting in a fast low-cost implementation.

The disadvantage of direct-mapped caches is that the fact that addresses map to only a single location can result in "thrashing" where program code operating within a restricted address range causes cache-lines which are required in the near future to be ejected from the cache leading to a high percentage of cache misses and reduced performance. Such thrashing behaviour is especially likely in the case of smaller caches as it is more likely in such cases that multiple addresses will alias to the same cache line. Set-associative caches alleviate the shortcomings of direct-mapped caches at the expense of additional area and complexity. Cache associativity increases cache performance by reducing the number of cache conflicts ("thrashing") [51].

While associativity reduces the number of conflicts by allowing locations to map to multiple cache lines, it has the disadvantage of slowing down the cache because of its added complexity. On cache updates, the replacement strategy determines the way a new line is put into, evicting the previous contents of the line. Common replacement strategies are least recently used (LRU) and round robin.

Temam and Jalby [48] were the first to develop a detailed model of cache misses for Sparse Matrix Vector Codes. They concluded that cache line size has the greatest impact on cache misses, while associativity has the least. As part of his work Vuduc [52] extended the model proposed by Temam and Jalby, and concludes that it is important to

have strictly increasing line lengths through the memory hierarchy to achieve good performance using his execution model.

### 4.1.6 Pre-Fetching

As outlined previously the memory wall is increasingly the obstacle to achieving higher Instruction Level Parallelism (ILP) despite OOO and other techniques intended to mask increasing DRAM access latencies. According to Hennessy and Patterson [34] instructions and/or data can be pre-fetched by hardware either directly into the cache or into an external buffer which can be accessed more quickly than main memory. Pre-fetching relies on using memory bandwidth which would otherwise remain available but unused, as otherwise it could actually degrade performance by interfering with demand misses. Bobba et. al state in [41] that as DRAM access latencies continue to increase even such techniques can no longer fully hide the effect of increasing latency. In such circumstances data pre-fetching is a technique which attempts to minimise cache misses and hence the effect of increased latency on performance, by anticipating future data accesses and moving required data closer in the cache hierarchy to the processor.

Generally a good data-pre-fetch scheme should have the following properties:

- Pre-fetches must be useful (pre-fetched data must be used in the near future)
- Pre-fetches must be timely (pre-fetch shouldn't displace data required in the near future)
- Pre-fetches should not lead to cache pollution i.e. displacing data to be used in the near future with pre-fetched data which is not accessed

### 4.1.7 Data Compression

It has also been seen that I/O pins are expensive and scale at 1/5 of the rate that the underlying semiconductor process technology does. Thus if anything can be done to exploit the properties of the data set to increase the effective bandwidth of the I/O pins used to transfer the compressed data-set will increase the overall system throughput as the processor will not be as starved of data. Relying on I/O bandwidth alone to increase performance is at least 5 times more costly than improving FPU utilisation which is cheap as it is related to architecture and underlying process technology [53].

The potential for the compression of address information over I/O pins was first highlighted by Hammerstrom and Davidson [54]. This work was extended by Farrens

and Park [55] who showed that a simple base-register could be used to hold the upper address-bits while less significant bits are transmitted over a reduced width address bus can cut address bandwidth between a processor and memory by up to 60% without significant loss in performance. However a complex fully associative base-register cache with LRU was required for optimal performance. Employing a more practical set-associative or direct-mapped cache increased the number of address bits from 11 to 16-bits which still saves 50% compared with a full 32-bit bus with little in the way of performance degradation. A recent survey paper by Liu [56] compares the various address compression schemes in detail. Address compression schemes can also reduce I/O power-dissipation by from 13% [56] up to 84% [57] for some media applications

A recent reference to the exploitation of entropy in floating-point numbers was made by Citron [58].

Analysis showed significant potential for compression of addresses along with more limited potential for the compression of integer values and floating-point exponents. An implementation of a cache-based compression scheme showed that the hit rate for integer values was high at 99% but that overall the compaction achieved was dominated by floating-point values which achieved a much lower hit rate of 42%. Citron concluded that the main gain was achieved by compacting addresses from 64-bits to 24-bits although 32-bit addresses such as those used in this work are more appropriate for this class of application.

Finally a unified approach to compress all levels of the memory hierarchy presented by Hallnor and Reinhardt in [59] and derived from IBM's MXT scheme [60] is claimed to provide all of the advantages of the previous schemes without the expense of compression and decompression at each stage. The authors use a single compression algorithm and block size so that data can be transferred between main memory and a compressed L3 cache. They were able to achieve a performance increase equivalent to about 50% of what could be achieved by doubling the L3 cache size to 2MB for approx. 1/10 the area increase.

## 4.2 General Purpose Processors (GPP)

According to Hennessy [34] the limits on the increase in performance, achievable by pipelining architectures are that deep pipelining can lead to an eventual increase in CPI because it increases pipeline dependencies and associated penalties. Additionally Flynn

points out that ensuring high IPC (Instructions Per Cycle) from an architecture relies on increased Instruction-Level Parallelism (ILP). Techniques used to increase IPC such as Out-of-Order execution dramatically increase the implementation cost of a processor by virtue of the additional hardware necessary to support multiple partially executed instructions active simultaneously inside the processor.

The Intel Pentium4 [61] is a good example of the compromises inherent in processor design For marketing reasons Intel decided to focus on processor clock speed targeting 40% higher speed in the same process technology as the previous generation processor. This was achieved by increasing the pipeline depth from 3 stages in the Pentium II (P2), to 10 stages in the Pentium III (P3) to 20 stages in the Pentium 4 (P4) as shown in **Figure 4-10**. Note that the CoreDuo has a pipeline depth similar to the Pentium III.

Although this approach extracts the maximum possible in terms of processor clock rate from a given process technology this increase in clock rate comes at a cost in terms of the increased miss penalty which can occur for instance when a branch miss prediction occurs and the complete pipeline has to be refilled incurring a miss penalty of up to 20 cycles where the processor can do no useful work.

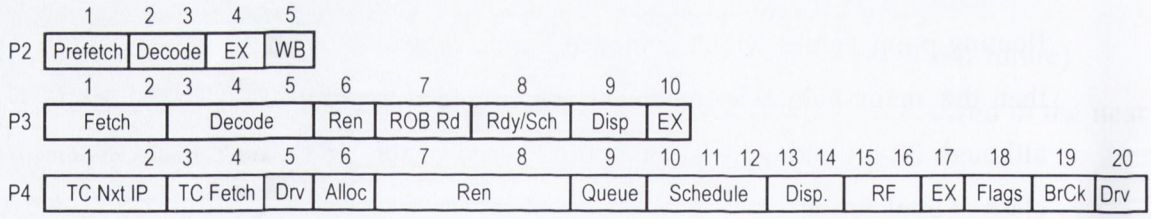



**Figure 4-10 Increasing Pipeline Depth of Intel Processors (source: [61])**

The memory hierarchy of the Pentium4 is shown in **Figure 4-11**. The Pentium 4 designers attempted to mitigate the miss penalty by increasing the size of the Branch Target Buffer by a factor of 8 times to 4KB, however the average IPC was still reduced by approximately 20% eating into the 40% gain in terms of clock frequency with respect to the Pentium III as reported by Intel in [61]. The other negative effect of increasing the processor clock frequency is that the L1 cache, which has to run at the same speed as the processor pipeline had to be physically small in order to minimise parasitic capacitance and meet the target speed requirements.

Due to these design constraints, the 8KB L1 cache of the Pentium4 can only hold 1024 double-precision (64-bit) vector entries; therefore the Pentium4 will spend most of its

time accessing the L2 cache. However, the 256KB L2 cache-size means that if we assume each double-precision vector entry to be multiplied by the Sparse Matrix in CSR/CCS format consumes 64 bits, the Pentium4 L2 cache can hold 32k vector entries. Thus under best-case conditions problems, which do not fit entirely in the cache, i.e. sparse matrices beyond 32k entries on a side, perform poorly on the P4 architecture.

The trend in processor micro-architectures has changed in the past 5 years since the design of the Pentium4, due to the difficulties in scaling deep sub-micron technologies, and currently all of the leading microprocessor manufacturers have moved away from technology scaling as the primary means of increasing performance, and towards new multicore architectures as a means of sustaining performance increases through increased parallelism.

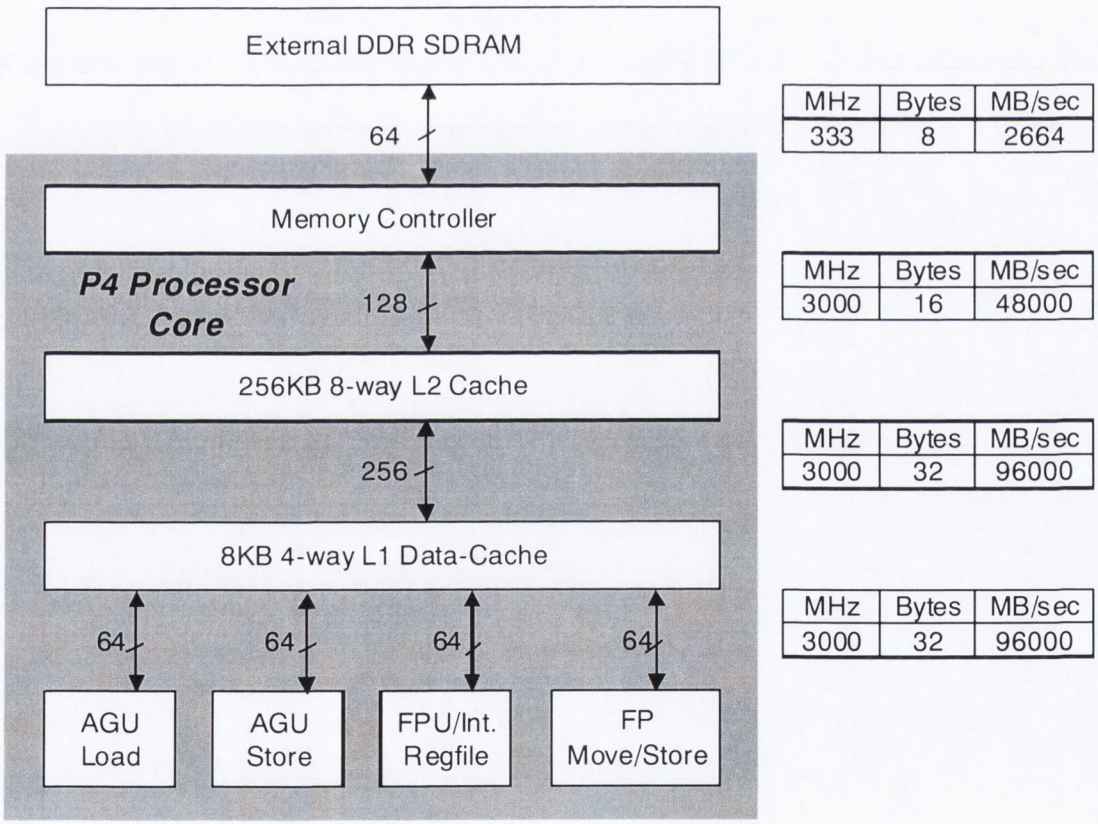



**Figure 4-11 Processor Cache Hierarchy and Bandwidths (source: [61])**

## 4.3 Chip Multiprocessor (CMP)

Over the past 2 decades processor designs have achieved dramatic gains in terms of single-threaded performance using a variety of micro-architectural techniques including: superscalar issue, Out-Of-Order (OOO) issue, on-chip caches, deep pipelines and

associated branch predictors. However these improvements have come at the cost of ever-decreasing efficiency in terms of performance achieved per additional transistor. According to [81] processor performance increases have slowed from 60% per year in the 1990s, to 40%, a performance increase of only 20% from 2000 to 2004. Processor designers rely on 2 forms of parallelism in order to achieve high performance: instruction-level parallelism (ILP) and thread-level parallelism (TLP). Processors designed to exploit ILP have special hardware that allows them to dynamically identify independent instructions that can be issued, in parallel, in the same cycle, by maintaining a pool of instructions in a large associative window, along with a register renaming mechanism that eliminates any false dependence between instructions.

The shortcomings of simply increasing ILP and the case for a single-chip multiprocessor were first highlighted by Olukotun et al. [82]. As indicated in [88] one of the major issues with Multicore architectures is programming them efficiently, more specifically enabling a single executable to be written for machines which share an Instruction Set Architecture (ISA). Similar techniques have also been proposed in [88] to distribute power dissipation "hot-spots" over a larger Multicore die, reducing associated cooling problems.

Assuming that the programming issues can eventually be resolved, the techniques proposed for Multicore processors are suitable for multi-tasking environments as they split program functionality across multiple cores and associated caches decoupling processes which would otherwise tend to compete for resources such as the external memory interface. Unfortunately in SMVM applications the main limitation is the low ratio of calculations per word fetched from external memory meaning that applying Multicore processors to such problems simply means that more processors will spend more time waiting for data to be fetched from external memory owing to the arbitration overhead for multiple cores sharing a single external memory bus unless something can be done to increase the effective memory pin bandwidth available.

Processor designers can exploit TLP by executing different threads in parallel, and processor architectures designed to exploit TLP often contain features which also allow them to take advantage of ILP within a thread. Multi-Threading (MT) is a technology where multiple hardware threads are integrated into the same processor core. Typically such threads appear to the programmer as logical processors with an independent

Register-File (RF) for each thread, where all threads share the same execution engine and L1/L2 caches. Simultaneous Multi-Threading (SMT) is a variety of MT where multiple threads execute simultaneously on the same core.

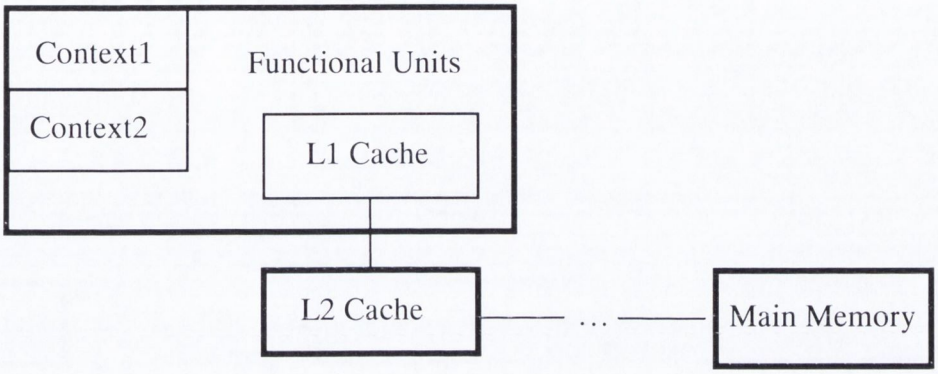



**Figure 4-12 SMT Processor**

A Chip Multi-Processor (CMP) consists of multiple, fully-featured processor cores on the same processor die. A typical CMP shown in **Figure 4-13** has separate L1 caches for each core, with both cores sharing a second-level L2 cache and external memory interface. CMPs generally have greatly reduced resource contention compared to SMTs owing to the higher degree of resource duplication.

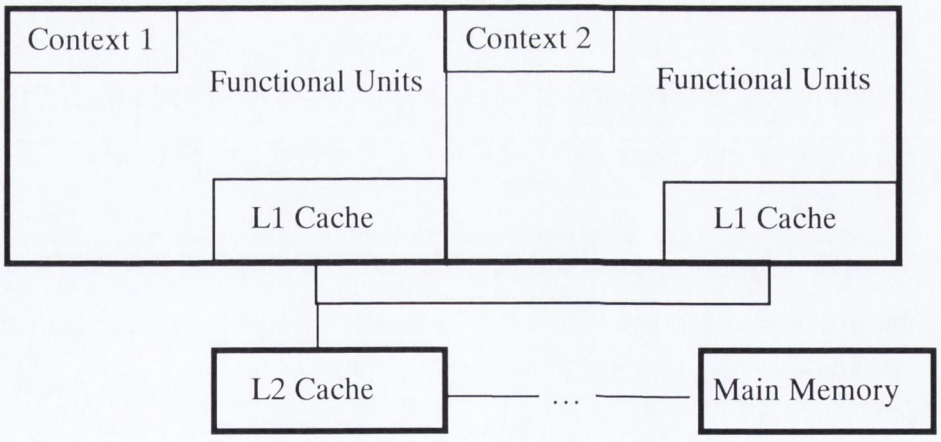



**Figure 4-13 CMP Processor**

Multi-threading may also be supported in CMPs [83] but generally the cores used in CMPs for desktop rather than server class computers, focus on simplicity, die-size and power consumption. However there are examples of Chip Multi-Threading (CMP/MT or

CMT), such as Sun's Niagara processor [90], in which 32 simultaneous execution threads are supported using an 8-core CMP which supports 4 MT threads on each core.

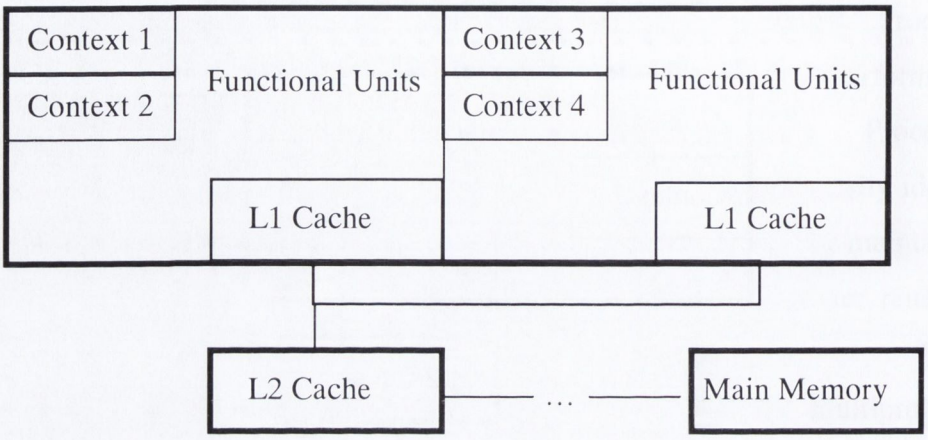



**Figure 4-14 CMT (CMP/MT) Processor**

In some applications multiple instruction sets are supported in heterogeneous CMPs, however we will concentrate on general-purpose CMPs which can share the same processor instruction-set (for instance x86) across all cores.

The main driver of the elements shown in the table is power dissipation, which led to the abandonment of the 4GHz Pentium4 development at Intel and its replacement by a simpler dual-core CMP derived from Intel's mobile Pentium-III development [91]. Workloads with high levels of TLP continue to present problems on such architectures as shown in [92] due to the limited size (64-entries) of the Pentium4 TLB resulting in poor coverage of the address-space. In general CMP/MT processors improve overall performance on certain applications by covering the latency of a stall on a given thread by allowing other pending threads to execute on the same pipeline, however this comes at the expense of increased resource contention [92]. This problem can lead to dramatic falls in performance when the different threads have conflicting datasets resulting in cache thrashing [92]. The major issues with single-core designs and how they are addressed by CMP/MT designs are detailed in **Table 4-5**.

| | Single-Core | | CMP/MT |
|---|---|---|---|
| | Problem | Solution | Solution |
| Memory Stall Latency | when CPU stalls 100s of useful instruction cycles are wasted | can use large caches and ILP to cover latency but effectiveness is limited | Latency covered by TLP. When a thread stalls another takes over the MT cores pipeline |
| Branch Prediction | branch miss-prediction causes v. long pipe to be flushed | there is a limit to the accuracy of branch predictions even if more HW used | Latency covered by TLP. When a thread stalls (miss-prediction) another takes over pipeline |
| Power Dissipation | scaling performance using clock rate and ILP leads to v. high power/heat dissipation | ultimately limits performance and reliability of CPU | yields better CPU power/performance and distributes hot spots better over die |
| Die Size | speed of light is fundamental limit on clock rate for single core | interconnect delays scale worse than gate delays so die size is a performance limit for large single cores | simpler CMP cores are smaller so performance can be increased compared to single complex core |
| Complexity | Very large and complex cores are difficult to debug and verify | larger and larger teams are required to verify and debug cores | using a single v. simplecore performance scales, not complexity |

**Table 4-5 Single-core versus CMP/MT Performance**

A good example of a CMP is IBM's Cell processor [83], co-developed with Sony and Toshiba containing a PowerPC core along with 8 specialised SPUs (Synergistic Processing Units) shown in Figure 4-15. Other examples include Intel's Larabee GPU [84] which is based on simplified in-order x86 cores augmented with vector units and Intel's SCC [85] which contains a cluster of twenty-four dual-core x86 processors, giving a multicore cluster of 48 cores on a single die. More recently low-power embedded CMPs which can rival the performance of ASICs in terms of performance/watt have begun to appear. A good example of this trend is the Stanford ELM processor [86], which dissipates between 1.5 and 3x the power of an ASIC implementation in the same process technology and clock rate. This efficiency is a factor of 23x better than a RISC processor implemented in the same technology and is achieved by careful segmentation of the register file hierarchy for data, and the introduction of an Instruction Register File into which blocks of instructions are loaded, rather than fetching a VLIW instruction from cache on each 200MHz clock cycle.

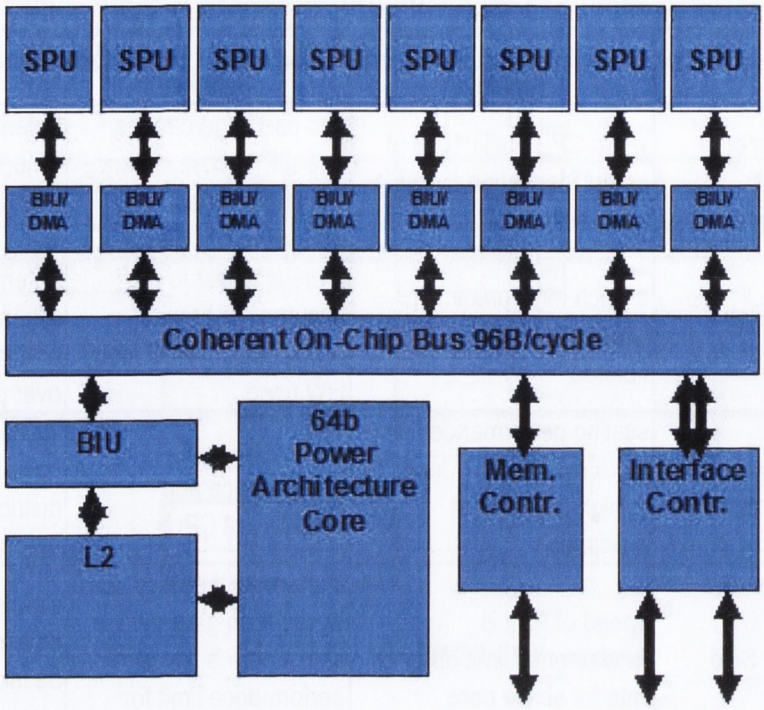



**Figure 4-15 IBM Cell Processor (source: [83])**

Interestingly it is pointed out in [83] that there are increasing opportunities for the inclusion of special-purpose accelerators in CMT/CMP designs as the increased performance (10x) and power-efficiency of accelerators becomes very attractive when the cost of the accelerator hardware can be amortised over many concurrently executing threads. Examples of such accelerators presented in [83] are Network Offload Engines (NOE), Cryptographic Accelerators, OS accelerators and XML parsing and presumably a similar case could be made for the acceleration of SMVM for Web-search acceleration. Indeed heterogeneous processing appears to be the path of choice within the CMP community.

## 4.4 Stream Processors

Stream processors have emerged from commercial work on Graphics Processing Units (GPUs) over the past two decades as well as academic work such as the Imagine stream processor co-developed at Stanford University and MIT between 1996 and 2001 as described by Rixner in [62]. Over this period Graphics Processing Units (GPUs) have moved from being fixed-function and largely fixed-point arithmetic pipelines, to being highly programmable floating-point pipelines over the past 5 years as shown in **Figure 4-16**. Data is input to a GPU pipeline in the form of quad-vectors $(x, y, z, w)$ which

allow graphical rotations to be performed by matrix multiplication. In more recent GPUs programmability has been greatly extended.

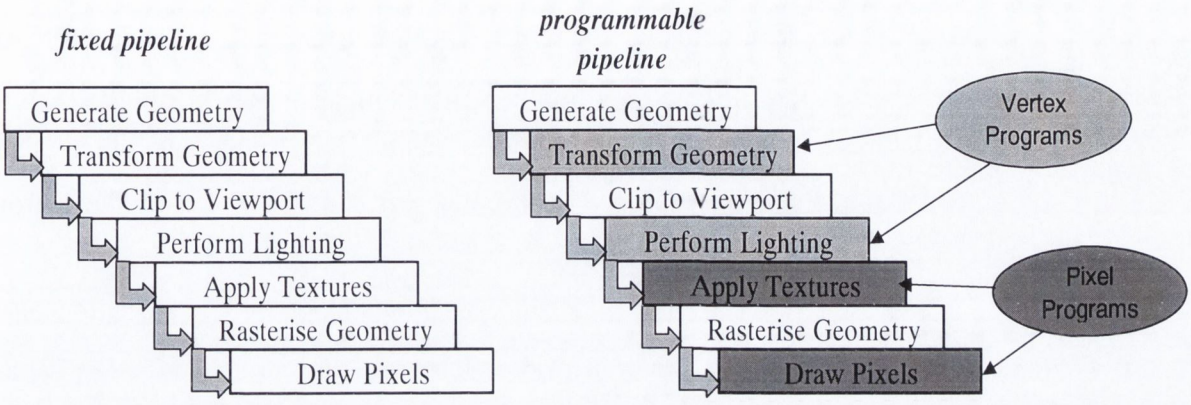



**Figure 4-16 Fixed vs. Programmable GPU Pipelines (source: [80])**

The programmable elements in GPU graphics pipelines consist of multiple floating-point units under the control of simple programs or shaders and the two major vendors of GPUs have converged on IEEE single-precision (32-bit) floating-point format for their GPU pipelines although they do not implement the IEEE754 standard exactly. A typical example of this is the IBM Cell processor [63] which only implements the truncation (round to zero) mode of the IEEE standard. The choice of this rounding mode leads to poor performance on non-graphics workloads like FFTW [64].

Comparisons [80] of stream and vector computing models on these architectures show that, for some applications at least, the streaming model is superior to the vector model. According to the authors, scientific applications performance is better by 34% on streaming versus vector machines when cache is not used and 58% if cache is used, although the gains in media applications are much smaller. While streaming architectures offer very high performance, programmability and legacy code remain huge issues to be resolved before such architectures can replace clusters of x86 processors. Initially, according to Ujaldon [66] using GPUs for general-purpose computations entailed disguising input data as vertex attributes, large data-structures as textures, instructions as kernels, and final results as portions of video memory. Essentially the applications such as SMVM had to be rewritten using APIs such as OpenGL [74] or DirectX [76] as graphics shaders. The academic community recognised the power of the GPU hardware as well as the issue of how to program it and responded with stream programming languages such as Brook [70] and StreamIt [73]. Commercial stream

programming languages and APIs were made available by PeakStream, and RapidMind which was acquired by Intel and introduced as its own data-parallel programming API called Ct [72] to address the software development challenge.

The most recent developments in terms of the programming model for streaming architectures are the proprietary Cuda [68] C-like language which supports Nvidia GPUs, ATI's Brook port [69] and the cross-platform OpenCL [67] language which is supported by both ATI and Nvidia developed by Apple Corp. and now part of the Khronos [75] family of standards for mobile devices.

The advances on the software side have been matched by similar advances on the hardware side and the two most recent products from Nvidia called Tesla [77] and more recently Fermi [77] fully support double-precision floating-point and even go beyond double precision internally where a compound MAC (Multiply Accumulate) has been used to increase numerical precision on operations such as dot products. The Nvidia Fermi contains 512 cores and supports the new IEEE 754-2008 standard [78] for floating-point arithmetic. Fermi contains 8 Streaming Multiprocessor (SM) blocks, each containing 32 cores as shown in **Figure 4-18** .

The key reason for the interest in streaming architectures is the very high levels of floating-point performance which can be achieved in comparison to microprocessors such as the Pentium4 as shown in **Figure 4-17**. This being said, the available memory bandwidth on GPUs continues to lag the amount of on-chip computational bandwidth.

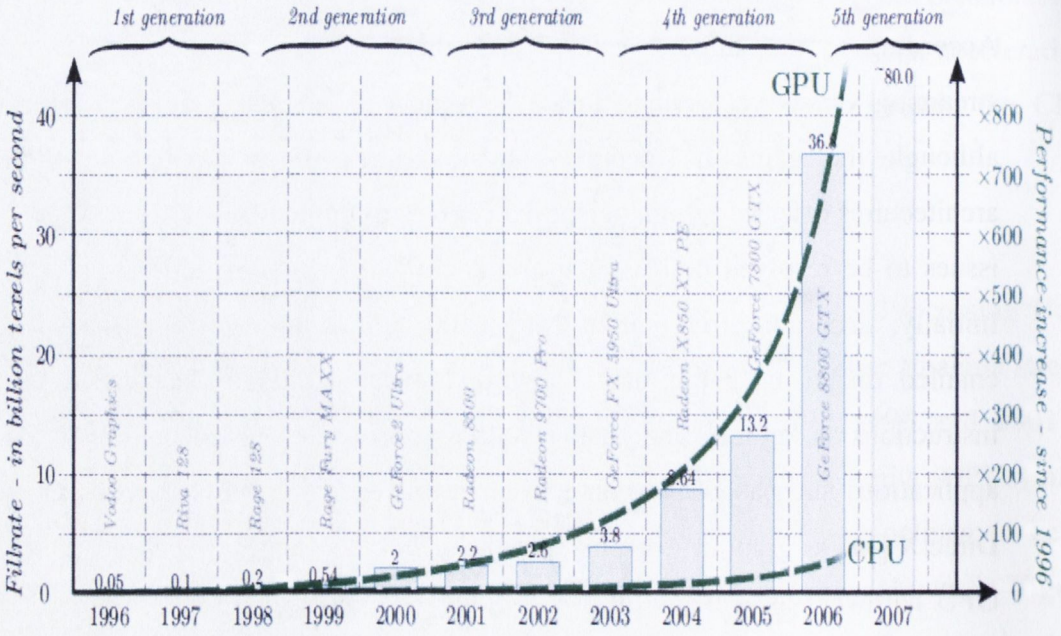



**Figure 4-17 GPU Performance Trend (source: [65])**

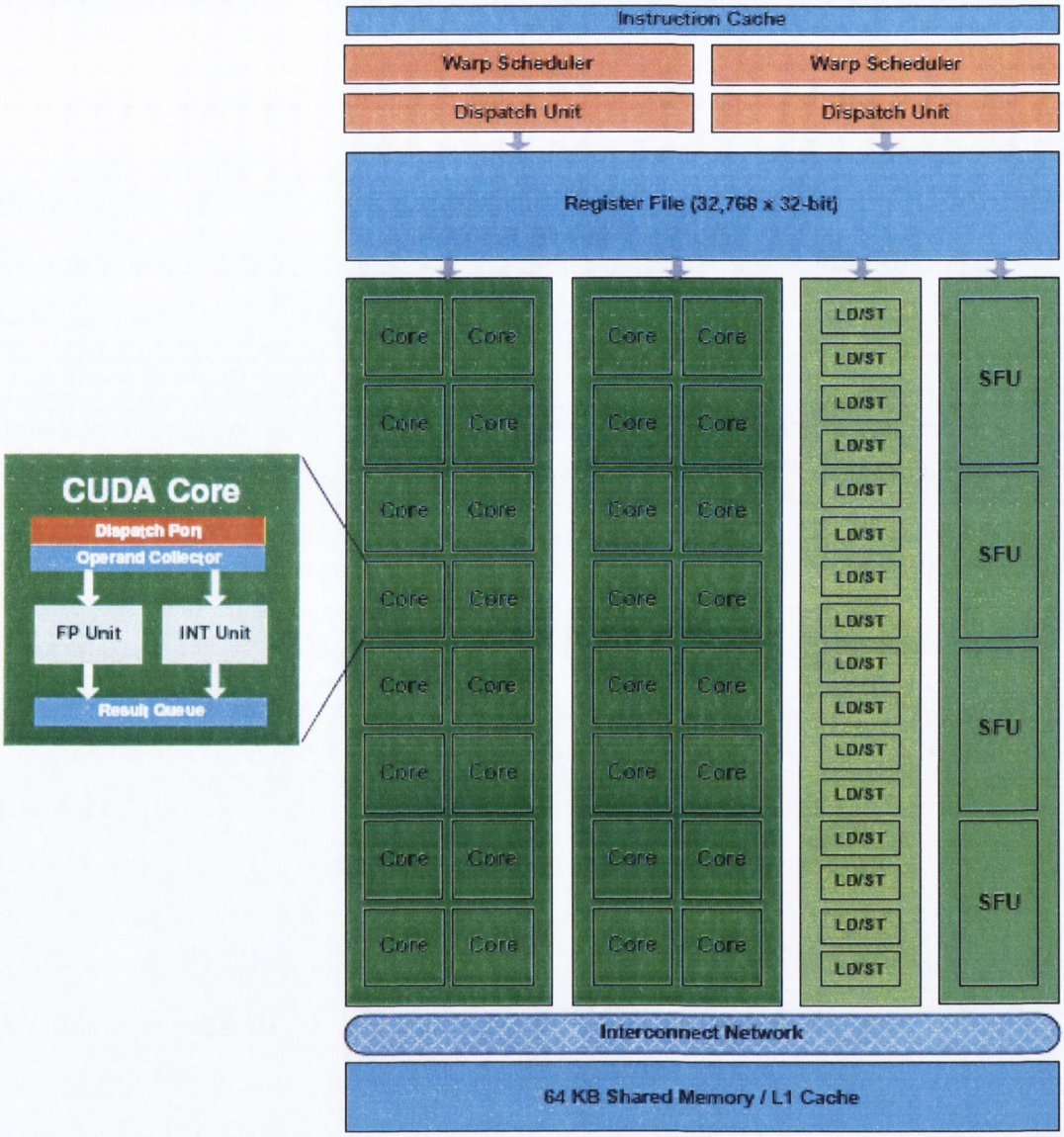



**Figure 4-18 Nvidia Fermi Streaming Multiprocessor (source: [77])**

## 4.5 Summary

Although General Purpose Processors such as the Intel Pentium4 and AMD Opteron dominate the workstation and cluster computing markets they perform poorly on Finite-Element and other Scientific and Numerical problems due to their internal architectures and compromises made during their design aimed at addressing a broad range of applications. One of the main reasons for the lack of performance of general-purpose architectures is that they rely on the spatial and temporal locality of their onboard caches and registers for high performance, and if the data set does not map well to the generic

onboard cache the architecture as a whole performs poorly. Benchmarks such as STREAM [45] and SpecFP [46] show the performance of general purpose architectures is poor for Sparse Matrix-Vector Multiplication (SMVM), and is not keeping pace with the speed of the processor core. A primary reason for this is a phenomenon known as the memory wall [33] and is a common issue for all architectures.

Another common problem is that in order to clock the processor at the highest possible rate it is often very deeply pipelined. While deep pipelining increases the operating frequency and hence MFLOPS in normal conditions it also negatively affects performance when a hazard occurs and the pipeline has to be flushed and refilled. Again with short row or column vectors as occur in SMVM operations the probability of stalls is increased leading to reduced FPU utilisation in such applications.

Vector-register supercomputers such as the NEC Earth Computer and Cray-X1 [93] solve some of the issues associated with general purpose architectures and continue to be the most powerful machines in terms of FLOPS performance, and in terms of FPU utilisation, however these machines are well beyond the budgets of individual engineers and scientists. Although vector machines perform better than general purpose processors they still suffer from performance limitations for short vectors. This is the case in SMVM operations where the start-up delay degrades the performance, as the delay in filling the pipeline is of the same order as the number of vector elements to be processed (on the order of 24 non-zeroes per matrix row/column).

Chip-level solutions for High-Performance Computing (HPC) have continued to progress along 2 parallel tracks. The first is the continued evolution of multicore processors in the form of traditional CMP (Chip Multi-Processors) based on existing architectures such as the x86 and SPARC. Good examples of this are Intel's Larabee and SCC as well as IBM's Cell processor which is a hybrid rather than homogeneous CMP.

The streaming (GPU) paradigm has also been adopted by the designers of a number of specialist architectures aimed at supercomputing including Merrimac [94] and Clearspeed [95], and specialised GPUs such as Nvidia's Tesla and Fermi have also emerged, with double-precision support, and general purpose programmability through C-like languages such as CUDA and OpenCL. This latter class of architecture has already led to the availability of desktop supercomputers capable of Teraflops performance levels [96][97].

# 5

# Chapter 5

*"I love talking about nothing. It is the only thing I know anything about"*

*-Lord Goring: "An Ideal Husband" (Oscar Wilde)*

## 5 Software SMVM

Achieving high performance Sparse Matrix-Vector Multiplication (SMVM) on modern microprocessors is a well-known problem. It has been widely reported that the raw performance of such microprocessors using un-optimised SMVM codes is poor and usually achieves less than 10% of the peak performance of such processors [26]. A variety of techniques have been proposed to deal with this problem and achieve a higher percentage of a processors FLOPS capability given that computational requirements in application areas such as Finite Element Analysis (FEA/FEM) and Latent Semantic Indexing/Analysis (LSI/LSA) used in data-mining and search-engines such as Google continue to outpace the rate at which new generations of processor can be deployed. The techniques which have been proposed to achieve optimal performance on uniprocessor nodes are matrix-reordering and automatic matrix library tuning. The former is used to improve locality in terms of the vector result from an SMVM operation, whereas the latter is used to transform the

source-matrix storage format and the SMVM code which operates on that format to the underlying processor architecture in such a way as to obtain optimal performance for that architecture. Matrix reordering is dealt with in section 5.5 along with analysis of the results presented in relevant publications and of the microprocessors used where necessary to illustrate the shortcomings of these techniques. Parallel computing has traditionally been the preserve of large organisations such as government research institutes, weather forecasters, the defence industry, aircraft and motor manufacturers and Universities, where scientists and engineers wait dutifully in line for access to such machines. Due to trends in microprocessor design which have emerged in the last few years, individual engineers and scientists are beginning to have access to personal computers based on Chip Multi-Processor (CMP) technology where multiple identical processors are integrated onto a single processor die along with cache memory and bus interface logic. The issue of how such processors are designed and are programmed and how SMVM problems and matrices can be optimally partitioned to run efficiently on such processors is dealt with in the following sections.

## 5.1 Sparse Matrix Vector Multiplication (SMVM)

A brief introduction to the problem of Sparse Matrix Vector Multiplication (SMVM) and the limitations imposed by processor architectures on SMVM performance follows. The rationale and means of storing and processing sparse matrices are discussed at length in Chapter 3 in particular Compressed Sparse Row (CSR) format is discussed in section 3.1.

### 5.1.1 SMVM Algorithm

A sparse matrix $A$ stored in CSR format can be multiplied by a dense vector $x$ to generate a dense result vector $y$ by the code shown in **Listing 5-1**.

```
L1. for (i=0; i<n; i++) // process all rows
L2. for ( j=ptr[i]; j<ptr[i+1]; j++) // row dot-product
L3. y[i] = y[i] + val[j] * x[col[j]]; // multiply-accumulate
```

**Listing 5-1CSR SMVM code**

In the SMVM code the first loop is executed n times, where n is the matrix dimension or number of matrix rows, so the *ptr* array is accessed 2n times as is the result vector *y*. The inner loop (dot-product for each row) is executed *nz* times where *nz* is the number of non-zeroes in the sparse matrix, meaning that the *val* array is accessed nz times as are *col* and *x* (indirectly). The total number of array references is therefore $3nz + 4n$ of which *nz* accesses are indirect accesses to the *x* vector. The total number of floating-point operations is *nz* multiplies and the same number of additions, or $2nz$ in total. In terms of integer operations the outer loop performs *n* additions and *n* comparisons and the inner loop performs $nz+n$ additions, and *nz* comparisons. The integer operations can typically be ignored as they are performed by dedicated looping and indexing hardware present in modern microprocessors.

### 5.1.2 Memory Bandwidth

An important factor in the performance of SMVM codes is the balance between program memory bandwidth requirements and computational requirements. As reported by Gropp [26] the sustainable memory bandwidth in SMVM dominated codes does not match the computational requirements making the peak FLOPS performance numbers quoted by microprocessor vendors meaningless for this class of application. In fact the authors report that McCalpin's STREAM benchmark [45] is a much better predictor of SMVM and application performance for this class of problem.

By way of example the data transfer requirements in bytes of the SMVM code presented in **Listing 5-1** are as follows assuming 4-byte (32-bit) addresses and 8-byte floating-point numbers and a square matrix, where n is the matrix order (#rows or columns) and nz is the number of non-zero elements to be stored explicitly in the sparse data-structure:

$$Bytes\_transferred = 12 * n + 20 * nz$$

**Equation 5-1 Data Transfer**

The amount of data that needs to be transferred for each FLOP is:

$$\frac{Bytes\_transferred}{FLOPS} = \frac{6 * n + 10 * nz}{nz}$$

**Equation 5-2 Data Bandwidth Requirement**

This means that the maximum possible SMVM FLOPS performance for a processor is given by:

$$M_{BW} = \frac{BW}{\left(6 * \frac{n}{nz}\right) + 10}$$

**Equation 5-3 SMVM Upper FLOPS bound**

$M_{BW}$ is the FLOPS rate and $BW$ is the peak memory bandwidth and as can be seen it is heavily dependent on how sparse the matrix is $nz/n$. This number however represents an upper bound on performance and in practice implementation details such as the cache architecture tend to reduce the actual SMVM performance achieved.

### 5.1.3 Cache Memory

In order to provide an optimal match for workload characteristics processors may opt for a range of cache designs from simple and fast direct mapped caches to highly flexible and expensive fully associative caches. Most modern microprocessors integrate set-associative caches, which fall in the middle ground between low-cost Direct-Mapped and high performance fully-associative caches, in order to provide high performance across a broad range of workloads.

A block can be placed in different positions in caches of the same size but with different levels of associativity as shown in **Figure 5-1**.

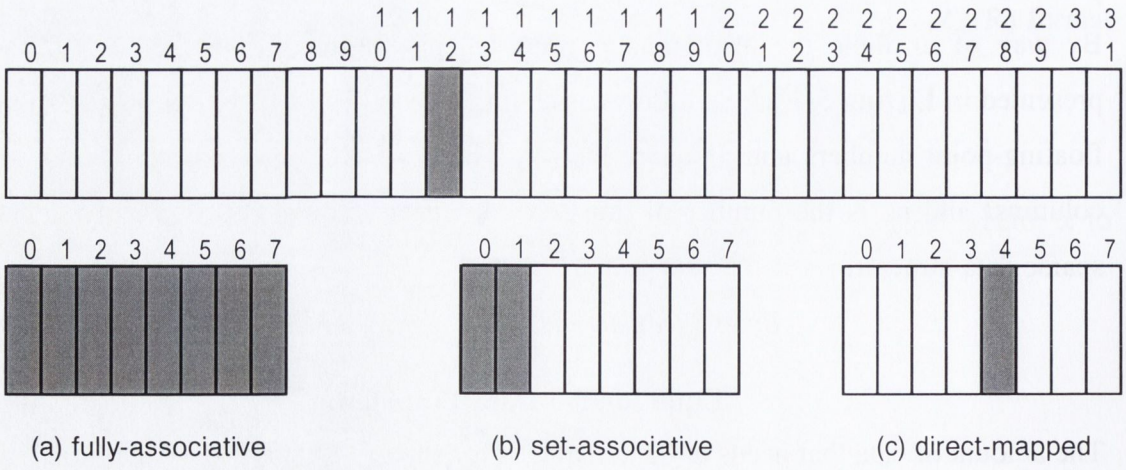



**Figure 5-1 Block-Placement vs. Cache-Type (source: [34])**

If the data can only be placed in one position in the cache it is said to be Direct-Mapped as in **Figure 5-1** (c) and the mapping is of the form:

(Block-Address) MOD (Number of blocks in cache)

**Equation 5-4 Block Placement**

If the block can be placed anywhere in the cache it is said to be fully-associative as shown in **Figure 5-1** (a). Otherwise if the block can be placed at a restricted number of locations in the cache it is said to be set-associative where a set is a group of blocks in the cache. A block is first mapped onto a set within the cache by bit selection and then the block is placed within that set. In the 2-way set associative cache in **Figure 5-1** (b)) 1 bit could choose between the 2 sets. A Direct-mapped cache is actually a set-associative cache with a set size of one and a fully associative cache is set associative cache where all words are stored in the same set.

### 5.1.4 Blocking

According to Hennessy and Patterson [34] blocking is the best known cache aware optimisation. Cache blocking works by reducing the number of cache misses by increasing temporal locality. Instead of operating on entire rows and/or columns of an array blocked algorithms operate on sub-matrices or blocks, the goal being to maximise the number of time sub-matrix data is reused before being replaced. By reducing the block size further to the point where it can be held in processor registers the number of program loads and stores can be minimised further increasing program execution speed. This optimisation also increases program speed in another way as registers can be accessed for reading and writing within the current processor cycle whereas most L1 caches typically require at least 2 cycles to access for reading and writing. Where associative caches are used, choosing a block size smaller than the available capacity can reduce conflict misses since blocking reduces the amount of active data in the cache at any given time.

### 5.1.5 Execution Models & Cache Behaviour on SMVM Codes

In reference to the cache behaviour associated with **Listing 5-1** the accesses to *y* and *ptr* have perfect spatial and temporal locality and the code can be modified to hold these values in registers for the duration of the inner loop iterations thus eliminating unnecessary memory references. The *val* array holding the *A* matrix non-zero values, along with the *ptr* array which holds the references to the beginning of each row (in

the *col* array) each have no temporal locality but perfect spatial locality. As each element of each array is only used once cache misses due directly to these arrays are all intrinsic, however they may cause conflict (cross-interference) misses on other arrays whose elements they may cause to be evicted from the cache.

Temam and Jalby [48] were the first to develop a detailed model of cache misses for Sparse Matrix Vector Multiplication given the assumptions that the target machine had a single level of cache hierarchy, and that the square banded matrices were stored in CSR format and had a random distribution of non-zeroes in the band. In their work they showed how cache misses vary with matrix structure (dimension, density, and bandwidth) and cache parameters (line size, associativity, and capacity and their model includes approximations of conflict misses, particularly self-interference misses. They concluded that cache line size has the greatest impact on cache misses, while associativity has the least impact. They also showed that self-interference misses can be minimized by reducing the matrix bandwidth (can be achieved by matrix re-ordering as shown in section 5.2) and maximizing cache capacity.

As will be seen later Vuduc [52] extended the model proposed by Temam and Jalby [48], and he points out that while it is common practice in processor designs to match cache line-lengths across different levels of the hierarchy this leads to poor performance on SMVM codes using his execution model for processor behaviour. He concludes that it is important to have strictly increasing line lengths through the memory hierarchy to achieve good performance using his execution model. He also points out that his model could still be improved to model conflict misses and matrix-dependent spatial locality more accurately, as well as processor instruction issue limitations.

## 5.2 Manual Performance Tuning

A variety of software techniques have been proposed to improve the performance on Sparse Matrix-Vector Multiplication (SMVM) on conventional General Purpose Processors (GPPs). According to Toledo [101] the four main performance bottlenecks in software implementations of SMVM are:

- Large number of cache misses due to poor data locality
- Tendency of multiple functional units to miss on the same cache-line
- Poor data locality (reuse) causes Floating-Point Units to be under utilised

- Array reference translation from integer to byte offset (extra integer operations)

Three techniques are outlined by Toledo [101] to improve the performance of SMVM codes on modern microprocessors by addressing one or more of these bottlenecks depending on whether the techniques are combined or used individually.

### 5.2.1 Reducing Cache Misses by Reordering

The first technique used by Toledo [101] is matrix reordering which is used to minimise the number of cache misses as accesses to the dense x vector and potential for data reuse depend on the sparsity structure of the A matrix. The bandwidth $b$ of a sparse matrix is the maximum distance between two elements in any row of the matrix. Reordering reduces the matrix bandwidth by permuting matrix rows and columns and their associated non-zero entries. Often, the elements in a sparse matrix can be reordered so that the bandwidth of the new matrix is smaller than the maximum possible bandwidth. Obviously in order to permute (reorder) the rows and columns of the matrix $A$ without altering the system of equations the indices of the unknowns in $x$ and the RHS vectors must undergo the same permutations as the $A$ matrix. This problem is known to be NP complete [102] so that brute-force methods requiring very long run times have to be used to produce an optimal solution, or else heuristic short-cuts are used in order to achieve acceptable results in reasonable time. Cuthill-McKee [102] and Reverse Cuthill-McKee (RCM) [104] are examples of such heuristics. Reducing the bandwidth of a matrix for instance reduces the time taken to perform Gaussian Elimination on a linear system from $O(n^3)$ to $O(nb^2)$ where $n$ is the order of an $n*n$ matrix. The benefits of reordering are large if $n >> b$. Reordering in the context of Toledo's work is performed so that the spatial locality of the $x$ vector is maximised, thus maximising data-reuse and minimising cache misses and the number of associated low-bandwidth accesses to external memory.

### 5.2.2 Pre-Fetching

As outlined previously the memory wall is increasingly the obstacle to achieving higher Instruction Level Parallelism (ILP) despite OOO (Out-of-Order) and other techniques intended to mask increasing DRAM access latencies. Bobba [41] states that as DRAM access latencies continue increasing, even such techniques can no longer fully hide the effect of increasing latency. In such circumstances data pre-

fetching is a technique which attempts to minimise cache misses and hence the effect of increased latency on performance, by anticipating future data accesses and moving required data closer in the cache hierarchy to the processor. According to Bobba et al. [41] a good data-pre-fetch scheme should have the following properties:

- Pre-fetches must be useful i.e. the pre-fetched data must be used by the processor in the near future
- Pre-fetches should be timely i.e. data should not be fetched too early lest it displace data which is required in the near future
- Pre-fetches should not lead to cache pollution i.e. displacing data to be used in the near future with pre-fetched data which is not accessed

According to Hennessy and Patterson [34] instructions and/or data can be pre-fetched by hardware either directly into the cache or into an external buffer which can be accessed more quickly than main memory. Pre-fetching relies on using memory bandwidth which would otherwise remain available but unused, as otherwise it could actually degrade performance by interfering with demand misses.

An example of such degradation occurs where hardware pre-fetch engines require a long stream of contiguous accesses in order to detect a viable pre-fetch stream [23]. While other architectures do not fare as badly performance still degrades significantly where large numbers of contiguous addresses are not present, as is the case of SMVM where large matrices have on the order of 20 non-zero entries per column [23], far fewer than the 128 or more contiguous addresses typically required for efficient hardware pre-fetching.

According to Toledo [101] pre-fetching works very well in the case of dense matrix-vector multiplication as the ratio of floating-point to load instructions is high (value reuse per load is high), however in SMVM this ratio is less than one and the bandwidth required to load data from memory is the performance bottleneck. Rather than using pre-fetching to hide memory latency Toledo advocates fetching both matrix data and column indices before they are to be used to minimise loss of memory bandwidth due to stalls.

According to Mowry [51] since prefetching *hides* rather than *reduces* latency, so if a program is already memory-bandwidth limited, it is impossible for prefetching to

increase performance. Locality optimizations such as cache blocking, however, actually *decrease* the number of accesses to main memory, thereby reducing both latency and required bandwidth. Therefore, the best approach for coping with memory latency is to first *reduce* it as much as possible, and then *hide* whatever latency remains. Weidendorfer and Trinitis [105] suggest a technique called Interleaved Block Pre-fetching to deal with this situation. The technique transforms a large block of data which does not fit into the cache into a succession of smaller blocks which do fit. The data in the smaller block is processed multiple times with only the first iteration subject to the external memory bandwidth limitation. Further iterations access the data in the cache, allowing the next data prefetch to be overlapped with execution of the current loop. Unfortunately this technique does not appear to be suitable for the vast bulk of SMVM arithmetic operations as data reuse is limited to the x and y vectors (y = A.x) unless the matrix is symmetric.

Compilers can override this behaviour by inserting explicit pre-fetch directives; however generally there is insufficient information at compile time to make this choice. The technique increases instruction bandwidth, which can exceed the benefits of data pre-fetching if care is not taken.

### 5.2.3 Register Blocking

Toledo reduces the number of loads required from external memory to internal registers or cache by restructuring the matrix, splitting it into multiple smaller matrices which have a denser, more regular structure. Such a locally dense but globally sparse structure is a feature of some application areas such as Finite element Analysis of engineering problems. The conversion of the source matrix in Compressed Sparse Row (CSR) format into the modified format was performed in a greedy (choosing locally optimal results in the hope of global optimality) fashion, by scanning the source matrix in pairs of rows and looking for locally dense 2x2 and 1x2 blocks and was optimal for 1x2 blocks. The effect of blocking was to reduce the number of loads with respect to unblocked.

### 5.2.4 Toledo's Results

Toledo's techniques in conjunction with optimal reordering boosted SMVM performance on 9 Boeing matrices by about 2.5x, but the 4 NasGraph matrices used had less than 20 non-zeroes per column and were actually slowed down by reordering

by between 10 and 20%, and also achieved little improvement benefit from pre-fetching and none at all from blocking. The cost of the blocking scheme was equivalent to 4-15 unblocked SMVMs using the original matrix; however 3 of the 13 matrices (23%) required the equivalent of 100 SMVMs to perform blocking due to the overhead of paging and memory management in RCM. In summary Toledo states that if the matrix is used more than 75 times blocking produces a reduction in execution time.

Despite claims of low overhead (1-3 SMVM operations) for re-ordering presented in [101] our experience using Matlab shows that these overheads are in practice an order of magnitude or more larger than those presented by Toledo when a database of very large matrices is used as opposed to the 13 matrices used by Toledo in his work.

According to Toledo a fringe benefit of RCM reordering reported by Duff and Meurant in [100] is that when a Conjugate Gradient (CG) iterative solver uses an incomplete Choelsky pre-conditioner the ordering of the matrix affects the convergence rate. Using RCM in this context reduces the number of cache misses, enables blocking and accelerates convergence. This if found to be true on further investigation might offset some of the overheads seen in our work.

Finally Toledo suggests that these techniques are best combined to produce optimal results however he suggest no approach for how this should be done either automatically or by a programmer. The issue of applying such techniques using an automated approach to achieve optimal performance for a given processor architecture without detailed knowledge of its internal structure has been the subject of research at UC Berkeley [137] and which will be presented in the next section.

## 5.3 Automatic Performance Tuning

A difficulty with software performance enhancement techniques used by Toledo and others is that it is difficult to generalise these techniques to the point that they can be reliably incorporated into compilers to produce both efficient and operationally correct code according to Yotov [106] and Demmel [107], resulting in great effort and expense to design, maintain and port tuned libraries to keep pace with technological evolution. Furthermore compilers cannot be relied upon to do the optimisation work as the choice of algorithm may depend on the input data as shown by Toledo [101]. Demmel identifies the 3 most important considerations in the design of a numerical software library as:

- Portability
- Performance
- Scalability

In recent years in order to address this challenge both model-based and experiment-based matrix kernel generators such as ATLAS [108], PhiPAC [109], Sparsity [110] and OSKI [111] make extensive use of these techniques to transform code in order to extract maximum performance from a given machine architecture.

The focus of Vuduc's work [52] is on register-blocking and to a lesser extent on multiplication by multiple vectors, and is supported by the development of models of the processor and its cache and memory sub-systems used to estimate the performance of register-blocked code on a particular matrix.

### 5.3.1 Register Blocking Revisited

Vuduc extends the work carried out by Im and Yellick on Sparsity [110] at UC Berkeley to search the solution space varying the number of rows and columns in the register blocked SMVM independently rather than together. Both Sparsity and OSKI rely on transforming the matrix from CSR to BCSR (Block Compressed Sparse Row) in order to reduce the overhead of addressing sparse matrix entries. An example of a matrix stored in 2x3 BCSR format is shown in **Figure 5-2**.

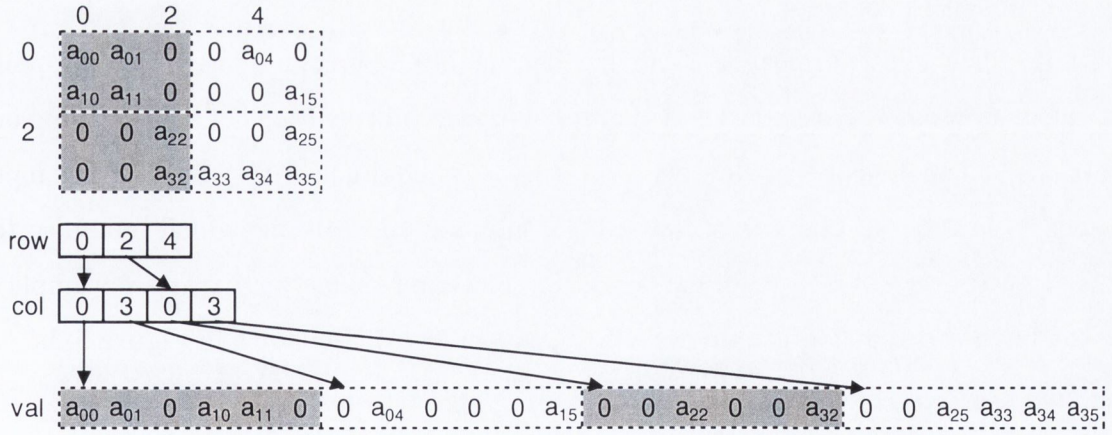



**Figure 5-2 2x3 BCSR Sparse Matrix Storage Format**

As can be seen the data-structure has 3 elements; an array of row pointers terminated by the number of dense r*c sub-matrices, an array of column indices and an array of non-zero values (plus zero fill if required).

```
void sparse_mvm_bcsr_2x3( int M, int n,
    const double* Aval, const int* Aind, const int* Aptr,
    const double* x, double* y )
{
    int I;
    for( I = 0; I < M; I++, y += 2 ) { // loop over block rows
        register double y0 = y[0], y1 = y[1];
        int jj;

        // loop over non-zero blocks
        for( jj = Aptr[I]; jj < Aptr[I+1]; jj++, Aval += 6 ) {
            int j = Aind[jj];
            register double x0 = x[j], x1 = x[j+1], x2 = x[j+2];

            y0 += Aval[0]*x0; y1 += Aval[3]*x0;
            y0 += Aval[1]*x1; y1 += Aval[4]*x1;
            y0 += Aval[2]*x2; y1 += Aval[5]*x2;
        }
        y[0] = y0;  y[1] = y1;
    }
}
```

**Listing 5-2 C code for 2x3 SMVM using BCSR format (source: [112])**

It can also be seen in this example that the dense sub-matrices contain a considerable number of zeroes required as fill-in (12/24 or 50% fill). The advantage is that if the underlying data maps well to the dense block size and does not contain a lot of fill less indirect addressing overhead is required as shown in **Listing 5-2**, thus speeding up the SMVM operation.

If the data does not map well to the chosen block structure many floating-point calculations will be trivial as they multiply the zero fill entries rather than carrying out useful calculations. For this reason a search is required on a sample of the input matrix in order to choose the best BCSR sub-block dimensions which minimise fill and maximise FLOPS performance. Using the same source matrix as an example it can be seen that a 2x2 blocking scheme results in less fill as shown in **Figure 5-3**.

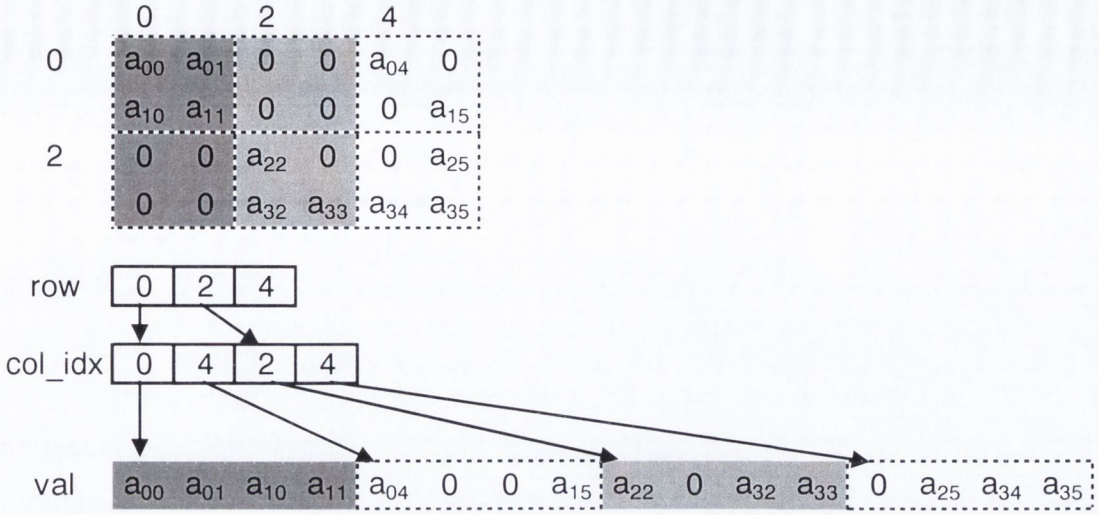



**Figure 5-3 Optimised BCSR 2x2 Blocking**

As can be seen the 2x2 sub-block storage format results in fewer fill-ins (4/16 or 25%) than the 2x3 case resulting in an increase in FLOPS performance. The SMVM code corresponding to the 2x2 BCSR matrix sub-block format is shown in **Listing 5-3**.

```
    void smvm_2x2( int bm, const int *b_row_start,
                   const int *b_col_idx, const double *b_value,
                   const double *x, double *y )
    {
        int i, jj;

        /* loop over block rows */
1       for( i = 0; i < bm; i++, y += 2 ) {
2           register double d0 = y[0];
3           register double d1 = y[1];
4           for( jj = b_row_start[i]; jj < b_row_start[i+1];
                   jj++, b_col_idx++, b_value += 2*2 ) {
5               d0 += b_value[0] * x[b_col_idx[0]+0];
6               d1 += b_value[2] * x[b_col_idx[0]+0];
7               d0 += b_value[1] * x[b_col_idx[0]+1];
8               d1 += b_value[3] * x[b_col_idx[0]+1];
            }
9           y[0] = d0;
10          y[1] = d1;
        }
    }
```

**Listing 5-3 BCSR 2x2 SMVM code (source: [112])**

### 5.3.2 Automatic SMVM Performance Tuning

The approach used by Vuduc in OSKI **[111]** builds on the Sparsity system for generating highly-tuned implementations of the SMVM kernel. OSKI given a matrix, kernel, and machine selects a fast SMVM implementation using a two step procedure:

- identify and generate a space of reasonable implementations
- search this space for the fastest one using a combination of heuristic models and actual experiments (i.e., running and timing the code)

The cost of using OSKI to tune a library breaks down into two cost elements:

- A static once-off element to characterise the underlying machine which is amortised over many uses of the library on the specified machine/library
- A run-time search to characterise each individual matrix with a view to choosing the optimum r*c block size

A block diagram of how the OSKI library tuning system works is shown in Figure 5-4.

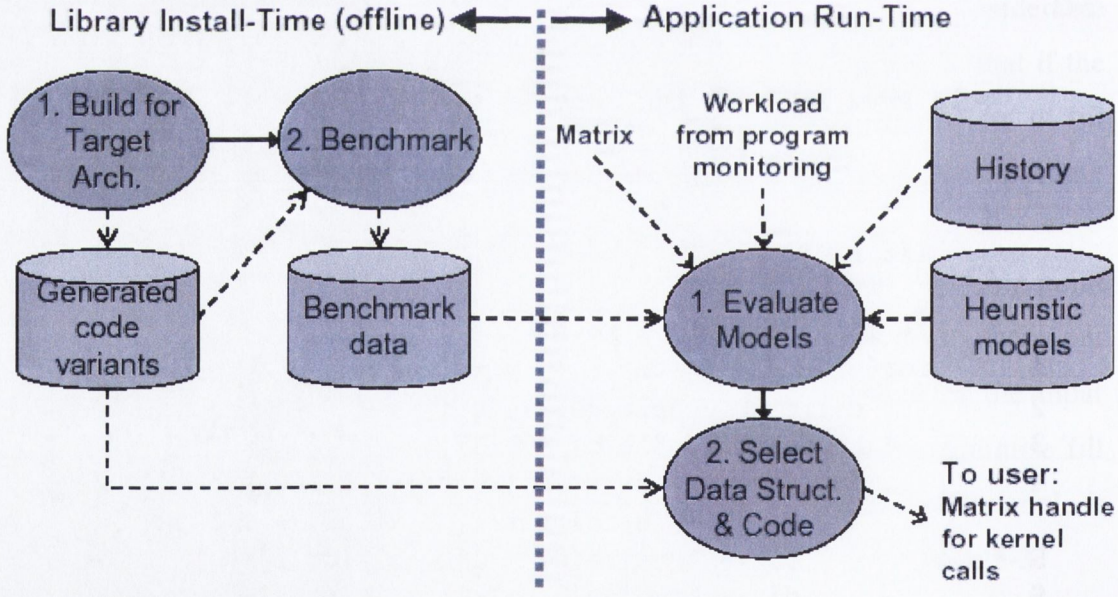



Figure 5-4 **OSKI Library Performance Tuning (source: [111])**

### 5.3.3 Vuduc's Results

By way of example of the BCSR format the FEM Matrix raefsky3, from the University of Florida Sparse Matrix collection is 0.33% sparse (1.5M non-zeroes) and consists entirely of uniformly aligned, dense 8x8 sub-blocks.

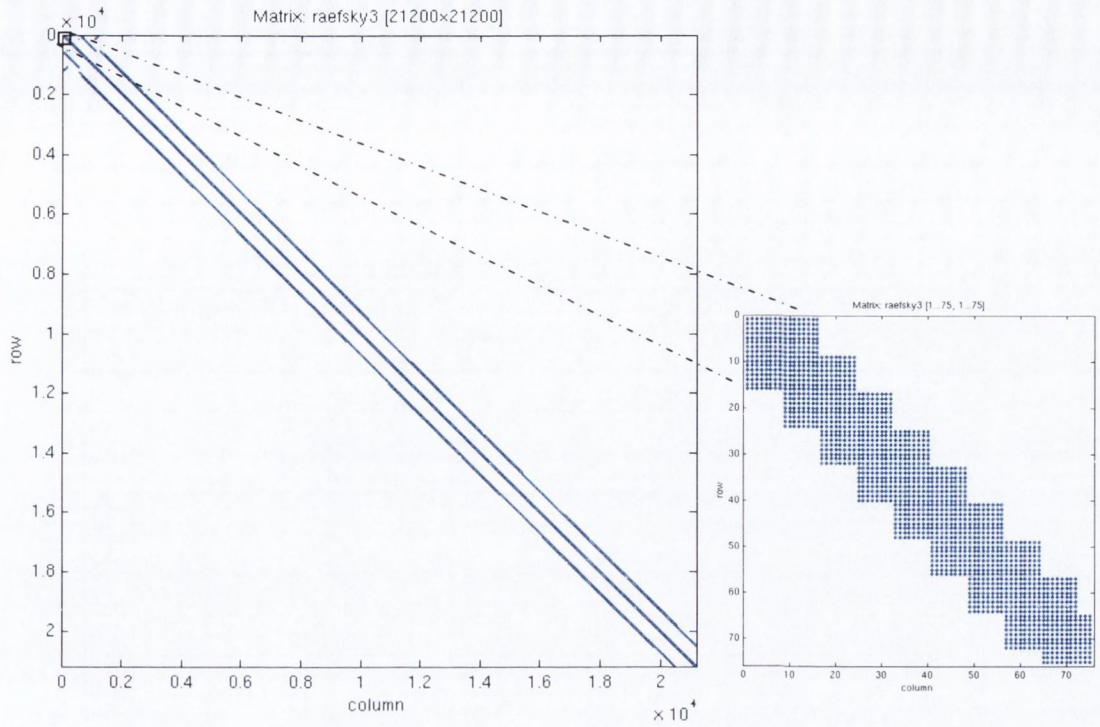



**Figure 5-5 raefsky3 spy plot (and detailed section)**

This matrix (**Figure 5-5**) should be a good target for SMVM implementations to fully unroll the multiplication by each block, exposing instruction-level parallelism (ILP) and opportunities for register level reuse. The OSKI tuning algorithm searched a space of sixteen possible r*c block organisations (1-4 rows * 1-4 columns) around the presumed 8*8 optimum, and surprisingly even on this well-behaved example performance is not well correlated with block size (r*c), and varies across platforms as shown in **Table 5-2**.

| Platform | Year | Peak MFLOPS | Best MFLOPS | % peak MFLOPS | Best org. r*c |
|---|---|---|---|---|---|
| Ultra 2i | 1998 | 667 | 63 | 9.4% | 8x8 |
| Pentium III-M | 1999 | 800 | 120 | 15.0% | 2x8 |
| Power 3 | 1998 | 1500 | 196 | 13.1% | 4x4 |
| Itanium | 2001 | 3200 | 229 | 7.2% | 4x1 |
| Power 4 | 2001 | 5200 | 703 | 13.5% | 4x1 |
| Itanium 2 | 2002 | 3600 | 1120 | 31.1% | 4x2 |
| Average | | | | 14.9% | |

**Table 5-1 Peak performance (%) of raefsky3 matrix SMVM (source: [52])**

Rather than perform an exhaustive search for the best SMVM execution-time both Sparsity and OSKI allow a user-defined portion of the source matrix to be used to derive a blocking scheme. This sub-optimal sampling is combined with performance estimation heuristic models to come up with a set of register blocking codes, with varying r*c, to run and search for the optimum performance.

| | | speed-up | | | |
|---|---|---|---|---|---|
| # | name | Itanium2 | Itanium1 | Pentium-III | Pentium-III M |
| 1 | dense2000 | 4.12 | 1.60 | 2.46 | 1.91 |
| 2 | raefsky3 | 4.07 | 1.61 | 2.38 | 1.70 |
| 5 | venkat01 | 4.03 | 1.59 | 2.24 | 1.66 |
| 3 | olafu | 3.44 | 1.49 | 2.15 | 1.53 |
| 4 | bcsstk35 | 3.25 | 1.51 | 2.15 | 1.74 |
| 8 | nasasrb | 3.15 | 1.48 | 2.05 | 1.53 |
| 12 | raefsky4 | 2.80 | 1.26 | 1.60 | 1.22 |
| 10 | ct20stif | 2.79 | 1.21 | 1.53 | 1.26 |
| 6 | cryst02 | 2.74 | 1.58 | 2.30 | 1.65 |
| 7 | cryst03 | 2.73 | 1.57 | 2.30 | 1.66 |
| 13 | ex11 | 2.70 | 1.24 | 1.48 | 1.19 |
| 9 | 3dtube | 2.69 | 1.55 | 2.23 | 1.65 |
| 11 | bai | 2.57 | 1.10 | 1.51 | 1.27 |
| 17 | rim | 1.97 | 1.05 | 1.23 | 1.00 |
| 36 | shyy161 | 1.97 | 1.02 | 1.04 | 1.00 |
| 21 | goodwin | 1.97 | 1.12 | 1.24 | 1.00 |
| 20 | lhr10 | 1.96 | | 1.31 | 1.00 |
| 27 | pwt | 1.94 | 1.06 | 1.06 | 1.00 |
| 15 | vavasis3 | 1.88 | 1.30 | 1.46 | 1.23 |
| 26 | onetone2 | 1.72 | | 1.04 | 1.00 |
| 25 | finan512 | 1.62 | 1.02 | 1.03 | 1.00 |
| 24 | coater2 | 1.55 | | 1.06 | 1.00 |
| 28 | vibrobox | 1.41 | 1.09 | 1.03 | 1.00 |
| 40 | gupta1 | 1.18 | 1.01 | 1.00 | 1.00 |
| 41 | lpcreb | 1.05 | | 1.06 | 1.00 |
| 42 | lpcred | 1.00 | | 1.06 | 1.00 |
| 44 | lpnug20 | 1.00 | 1.21 | 1.03 | 1.00 |
| **average speed-up** | | **2.35** | **1.30** | **1.56** | **1.27** |

**Table 5-2 Version2 Heuristic Speed-up by architecture (source: [52])**

The main difference between the register blocking heuristics used by Im in Sparsity and that proposed by Vuduc is that the original scheme proposed by Im assumes square r*c BCSR blocks while Vuduc's scheme allows rectangular blocks. The result is that Vuduc's scheme has a larger 2-dimensional space of possible solutions to

search rather than a one-dimensional space as in the case of the original heuristic. The proposed techniques are claimed to achieve 31% of peak and 4× speedups over CSR for a benchmark set of 44 matrices. However, the four processor architectures are benchmarked using a sub-set of only 27 of the initial 44 as shown in Table 5-2.

For these 27 matrices, the computationally more expensive version2 heuristic produces a speed-up on the Itanium2 architecture which is up to 2x the speed-up produced using the version1 heuristic, while closely matching what is achieved using a fully exhaustive search which is computationally very expensive.

Vuduc presents data in [52] in which BCSR reduces the execution time of SMVM to 2/3 that of CSR (1.5× speedup) but at a cost of requiring storage of 50% additional explicit zero entries. As can be seen the speed-up on the Itanium2 architecture is inversely correlated with the amount of sub-matrix fill as shown in Figure 5-6. The reason for this is obvious in that zero fill consumes FLOPS and memory bandwidth but does no useful work and hence does not speed up the calculations.

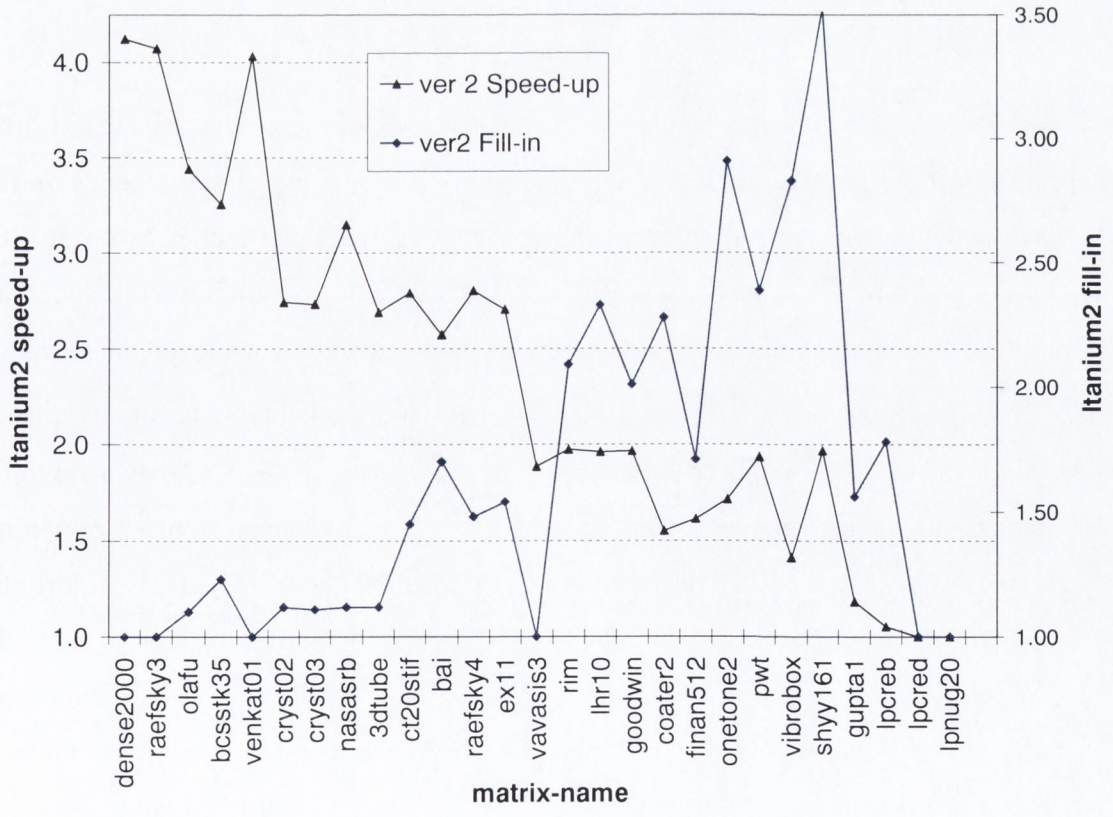



**Figure 5-6 Itanium2 Ver2 Heuristic Speed-up vs Fill-in (source: [52])**

In the results presented by Vuduc the Itanium2 distorts the performance gain which is averaged across all architectures and matrices in the 27 matrix benchmark as shown in Table 5-3. Firstly the benefits of the version2 heuristic in terms of speed-up only apply to the Itanium2 architecture whose peak performance increases by 38% and average performance increases by 27% using the proposed heuristic as opposed to the version1 heuristic from Sparsity/Im [112]. The speed-up due to the version2 heuristic is only between 3 and 10% in the case of the other architectures. The average speed-up for the Itanium2 is 2.35x and not 4.07x, and the average speed-up for all matrices and architectures using the version2 heuristic is 1.59x.

| | Pentium-III | | Pentium-III M | | Itanium 1 | | Itanium 2 | | group average |
|---|---|---|---|---|---|---|---|---|---|
| | speed up | % of exh. | speed up | % of exh. | speed up | % of exh. | speed up | % of exh. | |
| *avg. exh.* | 1.51 | 100% | 1.22 | 100% | 1.32 | 100% | 2.37 | 100% | 1.61 |
| *avg. v2 heur.* | 1.50 | 99% | 1.21 | 99% | 1.30 | 99% | 2.35 | 99% | 1.59 |
| *avg. v1 heur.* | 1.47 | 97% | 1.20 | 98% | 1.19 | 90% | 1.73 | 73% | 1.40 |
| *peak exh.* | 2.38 | 100% | 1.79 | 100% | 1.61 | 100% | 4.08 | 100% | 2.46 |
| *peak v2 heur.* | 2.38 | 100% | 1.74 | 97% | 1.61 | 100% | 4.07 | 100% | 2.45 |
| *peak v1 heur.* | 2.30 | 97% | 1.74 | 97% | 1.51 | 94% | 2.52 | 62% | 2.02 |

**Table 5-3 Average versus Peak Performance by Architecture**

Based on Vuduc's data the number of SMVM iterations to achieve an overall 20% speed-up for an iterative method using the proposed tuning algorithm depends on the matrix and the acceleration achieved however using the same model as before the speed-up could require an average of 67 iterations to achieve a 20% overall speed-up ($\beta = 1/1.2$) as shown in Table 5-4, where $\alpha$ is the speed-up achieved in terms of SMVM time alone, and $\beta$ is the overall speed-up required including tuning overhead. As can be seen the more speed-up achieved by tuning, the fewer SMVMs need to be evaluated in an iterative algorithm in order to achieve an overall speedup. Converting a matrix from CSR to BCSR format costs 5 to 40 unblocked SMVMs, therefore the upper bound on the overhead of using OSKI tuning is the equivalent of over 40 SMVM operations where translation of the matrix into BCSR format confers an advantage, and a baseline of 1-11 unblocked SMVMs even in cases where trans-coding offers no advantage thus decelerating performance in non-FEM applications.

| matrix | tuning overhead | # Accelerated SMVMs to gain 20% from tuning | | | | | |
|---|---|---|---|---|---|---|---|
| | | α=1.5 | α=2.0 | α=2.5 | α=3.0 | α=3.5 | α=4.0 |
| 2 | 22 | 132 | 66 | 51 | 44 | 41 | 38 |
| 3 | 22 | 132 | 66 | 51 | 44 | 41 | 38 |
| 4 | 36 | 216 | 108 | 84 | 72 | 66 | 62 |
| 5 | 18 | 108 | 54 | 42 | 36 | 33 | 31 |
| 6 | 40 | 240 | 120 | 93 | 80 | 74 | 69 |
| 7 | 40 | 240 | 120 | 93 | 80 | 74 | 69 |
| 8 | 38 | 228 | 114 | 88 | 76 | 70 | 66 |
| 9 | 40 | 240 | 120 | 93 | 80 | 74 | 69 |
| 10 | 38 | 228 | 114 | 88 | 76 | 70 | 66 |
| 11 | 18 | 108 | 54 | 42 | 36 | 33 | 31 |
| 12 | 23 | 138 | 69 | 54 | 46 | 42 | 40 |
| 13 | 24 | 144 | 72 | 56 | 48 | 44 | 42 |
| 15 | 21 | 126 | 63 | 49 | 42 | 39 | 36 |
| 17 | 27 | 162 | 81 | 63 | 54 | 50 | 47 |
| 21 | 26 | 156 | 78 | 60 | 52 | 48 | 45 |
| average # SMVMs | | 173 | 86 | 67 | 57 | 53 | 49 |

**Table 5-4 # tuned SMVMs required to achieve overall 20% speed-up**

This means that any iterative method using OSKI or a similar methodology would have to complete at least 40 SMVM operations before beginning to see any advantage from matrix kernel tuning in worst-case conditions. This would suggest that kernel tuning is not much of an improvement over RCM which was described in section 5.2. The other disadvantage of this methodology is that the user must call the "tune" routine explicitly however the tuning result can be saved to be used on future runs with a similar matrix. It is clearly indicated in [107] by Demmel et al that the overhead of searching can be much longer than traditional code compile times and that the overheads must be justified by actual application behaviour, which can be difficult to predict based on knowledge of the source-code alone.

Vuduc observes that there is a performance gap between matrices consisting primarily of dense blocks of a single size, uniformly aligned, and matrices whose structure consists of multiple block sizes with irregular alignment. For such matrices he recommends splitting the Sparse Matrix A into the sum $A = A1 + \cdots + As$, where each Ai may be stored with a different block size, and storing each Ai in a unaligned block compressed sparse row (UBCSR) [113] format that relaxes both row and column alignments of BCSR, at the cost of indirection to x and y instead of just x as

in BCSR and CSR. The main disadvantage with this technique is that it further expands the search range and hence the size of the overhead in terms of tuning that must be amortised before a benefit due to tuning can be realised.

Generally much of the performance advantage in Vuduc's experiments seems attributable to the features of the Itanium2 rather than anything the tuning is achieving. Those features include:

- Large (multi-Megabyte) L3 cache
- 2 Floating-point Multiply ACumulates (FMACs) per cycle
- 128-bit wide external memory bus

Strangely no results for performance tuning on the Pentium4 are presented despite the fact that it was available during the period the work was conducted.

This analysis of Vuduc's results including their specific suitability for the Itanium2 compared with other processors is confirmed by Buttari et al [114].

Furthermore Yotov et al. cast doubt on the utility of search to determine machine parameters in [106] and suggest that a static machine model can produce results on a par with search techniques, with the exception of the Itanium2 that is. Certainly if Yotov et al. are correct near optimal Register Blocking could be achieved at a fraction of the computational cost of searching making register-blocking useful even where the number of SMVM operations is very low.

In summary the approach of searching for a solution in the manner proposed by Vuduc only makes sense if:

- The user is knowledgeable enough to use the tool
- The user is using a computing platform that is a good match for the tool
- The matrix has underlying structure which can be tuned for

On the last point it is important to note that because of the use of a performance heuristic a fixed penalty of 1-11 SMVM operations is incurred where the method is used irrespective of whether the CSR to BCSR conversion is performed in the end leading to degradation in performance for non-FEM matrix applications.

## 5.4 Further Optimisations

A number of further optimisations occur in the literature and can be applied either manually or automatically. These optimisations include:

- Cache Blocking
- TLB Blocking

- Copy Optimisation
- Recursive Blocking
- Block Data-Layout

### 5.4.1 Cache Blocking

Cache blocking improves locality of accesses to the vectors x and y by dynamically inspecting the matrix data structure and changing it into a sequence of sparse sub-matrices so that the portions of the vectors x and y for each sub-matrix fit in the cache. Cache-blocking is especially useful when the source vector x is very large. In this case a search is performed to split the sparse matrix into $2^k * 2^l$ blocks so as to maximize SMVM performance. The fundamental trade-off that needs to be made is whether the benefits of the added locality outweigh the costs associated with the added accesses to the data structures.

As with the register blocking the issue is to how to automate the process of searching and cache blocking. To this end the work of Vuduc using automatically tuned register-blocking kernels to improve SMVM performance has been extended by Nishtala et al [115] to consider the problem of cache blocking of SMVM operations which is known to be important for some matrix and machine combinations.

The main difference between cache blocking and register blocking is that register blocking modifies the sparse matrix data structure (transforming from CSR to BCSR storage format) in order to decrease the overall memory traffic whereas cache blocking reorders memory accesses to increase temporal locality in a manner similar to that achieved by RCM reordering. The data-structure is a variant of BCSR with an optimisation to avoid iterating over rows which do not contain nonzero elements. Prior work on performance modelling of cache-blocking assumed that the matrices were small enough so that x and y fit in the cache which is rarely the case. To address this, the authors added a TLB buffer, ignored by previous models, to the model used by Vuduc [52] to predict optimum block sizes. Nishtala concludes [116] by saying that cache blocking appears to be most effective when all of the following are true:

- vector x does not fit in cache
- vector y fits in cache
- non zeros are distributed throughout the matrix and not in bands
- non zero density is sufficiently high

He also suggests a density of less than $10^{-3}$ (above this threshold register-blocking is better) and more than $10^{-5}$ (less than this threshold cache-blocking provides no speed-up) for cache-blocking to be useful. He also found that cache blocking does not help with band matrices no matter how large x and y are since the matrix structure already lends itself to the optimal access pattern. Unfortunately unlike the work of Vuduc no indication is given as to the search overhead required to perform cache-blocking.

Nishtala's most useful contributions are the suggestions for processor architects that:

- TLB misses reduced by cache blocking can also be avoided by creating large page sizes
- separate memory busses to the x vector and A matrix would improve performance as only x accesses are improved by caching due to reduced conflict misses

However, the problem of deciding when to apply cache-blocking, i.e. when it is likely to pay off is still open according to Nishtala [116].

### 5.4.2 TLB Blocking

A computer's memory is typically laid out in blocks of fixed size, called pages, a subset of which, called the working set of 64-128 translations, resides in the Translation Look-aside Buffer (TLB). The TLB is a small associative cache that stores the most recently used virtual-physical page translations corresponding to memory accesses. Each TLB entry points to a page which is typically between 6 and 64kB in size (controlled by the operating system). A TLB cache miss causes a TLB entry to be replaced with an entry from the page-table in memory. Additionally if data is spread too widely over the virtual address space it can result in an access to the page-table in main memory even though the required data is actually in the cache (L1/L2). Zhang and Zhang [117] present a variety of TLB blocking and padding strategies to be used depending on whether the TLB is fully-associative as in the case of the Sun UltraSparc-II or set-associative as in the case of the Pentium-II processor. In the case of set-associative padding causes block rows to be mapped to different TLB cache-lines, thus preventing conflict misses caused because multiple pages map to the same entry in the TLB. In the case of a fully-associative TLB the authors reported that the block size selected had to be less than or equal to the number of TLB entries otherwise array accesses would cause TLB thrashing reducing performance.

### 5.4.3 Copy Optimization

Interestingly some of the scientific libraries which library tuning was meant to make obsolete have incorporated internal code generation making use of the template programming paradigm available in the C++ language. The MTL library [118] for instance uses template programming techniques to allow register blocking and cache-miss reduction code to be automatically inferred by a C++ compiler at compile time. Another technique implemented in MTL is copy optimisation which was explored extensively by Lam et al [119]. The objective of this optimisation is that neither set-associativity nor multiple-word cache lines eliminate the large variance in the performance of blocked algorithms. A technique called copy optimisation is used to copy non-contiguous data into contiguous cache locations in such a way as to eliminate self-interference. This is achieved by mapping each word within a block to its own location making self-interference impossible. According to Lam et al applying this technique, while not always possible, allows cache misses to be bounded to within a factor of 2 from the ideal. Copying is to be avoided if the reuse factor is low which would seem to rule it out from the point-of-view of SMVM operations, the portion of the variable array to be used shifts over time or a large fraction of the data fits into the cache; in any of these cases the cost of copying may outweigh the benefits. Furthermore copying allows at least half of the cache to be used in each blocked loop nest making the penalty due to cache misalignment negligible. Finally in cases where the cache is set-associative copying eliminates not just self-interference but also cross-interference (conflict) misses. A detailed cost/benefit analysis of copying is presented by Temam et al in [120], and in [120] they report a 98% reduction in TLB misses due to copying in matrix-matrix multiplication on the Alpha processor.

### 5.4.4 Recursive Blocking

As was seen previously register, cache and TLB blocking require a detailed knowledge of the underlying machine architecture, and typically different blocking parameters for each level in the storage hierarchy, in order to achieve high performance. Recursive blocking [122] extracts improved performance from the memory hierarchy by repeatedly partitioning the problem into smaller and smaller sub-problems, so that data corresponding to different levels of the recursion tree fits into different levels of the memory hierarchy. This style of blocking improves

temporal locality and provides efficient cache and TLB blocking, while register blocking is used within sub-blocks.

Recent work by Wise et al [123][124] shows that Morton ordering of matrices offers a viable alternative to traditional row and column-major ordering. While this organisation is intuitive it does not take into account the fact that optimised codes no longer operate over rows and columns but rather over sub-blocks extracted from the matrix using register and cache-blocking. The underlying row or column-major storage means that any given sub-matrix will contain several disjoint portions from rows or columns which reside over several pages of memory, as referenced elements in the less favourable direction (row/column) tend to become farther away in memory [126]. This in turn leads to inefficient use of the memory hierarchy and performance which falls a long way short of the peak that hardware will support. An array in Morton order is decomposed as a $2^d$-ary tree whose sub-trees have contiguous addresses in memory. This improved locality minimizes page and cache misses, and thus performance. This ordering is used in conjunction with Ahnentafel (compact binary tree storage in level-by-level order) indexing on a block basis, with traditional Cartesian indexing within blocks.

One disadvantage of Morton-ordering is an expansion in terms of address space, however in contrast to other methods this extra data is stored in virtual memory tables rather than in cache. This expansion is a result of the Ahnentafel quad-tree organisation where 2 additional address-bits are used to distinguish between addresses in each of the 4 quad-tree branches {00, 01, 10, and 11}. Wise et al report a dramatic reduction in TLB as well as L1 and L2 cache misses using Morton ordering, and more importantly performance is almost flat as a function of problem size and type meaning that expensive searches on per matrix basis as advocated by Im [112] and Vuduc [52] are not required to obtain high performance across a large range of data sets and problem sizes. Other recursive orderings are evaluated extensively in [126] in the context of matrix multiplication, and advocates a modified and coarser-grained quad-tree layout in order to improve performance over that obtained by Wise et al.

### 5.4.5 Block Data Layout

Prasanna et al. [127] contend that most of the focus to date on bridging the gap between memory latency and processor speed has been expended on control transformations which change the loop iteration order and hence the data access-

pattern. As an alternative they propose a blocked data layout as opposed to the conventional row and column-major layouts shown in Figure 5-7. In a row-major layout of a large sparse matrix due to large stride (distance) between successive columns in a row can cause cache conflicts, and additionally if every row in a matrix is larger than the machines page size, column accesses can cause TLB thrashing, further degrading performance. In block data-layouts these problems are overcome by splitting large matrices into smaller B*B matrices in which all sub-matrices are mapped onto contiguous memory locations in row-major (or column-major layout if required) as shown in Figure 5-7(c). The only major assumptions made are that the cache is direct-mapped and the TLB is fully set-associative and a Least Recently Used (LRU) replacement policy is employed.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |
| 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 |
| 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 |

(a) row-major

| 0 | 8 | 16 | 24 | 32 | 40 | 48 | 56 |
|---|---|---|---|---|---|---|---|
| 1 | 9 | 17 | 25 | 33 | 41 | 49 | 57 |
| 2 | 10 | 18 | 26 | 34 | 42 | 50 | 58 |
| 3 | 11 | 19 | 27 | 35 | 43 | 51 | 59 |
| 4 | 12 | 20 | 28 | 36 | 44 | 52 | 60 |
| 5 | 13 | 21 | 29 | 37 | 45 | 53 | 61 |
| 6 | 14 | 22 | 30 | 38 | 46 | 54 | 62 |
| 7 | 15 | 23 | 31 | 39 | 47 | 55 | 63 |

(b) col-major

| 0 | 1 | 4 | 5 | 8 | 9 | 12 | 13 |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 7 | 10 | 11 | 14 | 15 |
| 16 | 17 | 20 | 21 | 24 | 25 | 28 | 29 |
| 18 | 19 | 22 | 23 | 26 | 27 | 30 | 31 |
| 32 | 33 | 36 | 37 | 40 | 41 | 44 | 45 |
| 34 | 35 | 38 | 39 | 42 | 43 | 46 | 47 |
| 48 | 49 | 52 | 53 | 56 | 57 | 60 | 61 |
| 50 | 51 | 54 | 55 | 58 | 59 | 62 | 63 |

(c) 2x2 block layout

**Figure 5-7 Different Data Layouts**

According to the authors block data-layout when associated with tiling, where block and tile sizes are the same, offers improved TLB performance when compared with alternatives such as copying (section 5.4.3) and padding. They go on to apply the same techniques to LU Decomposition and Cholesky Factorisation (CF), pointing out that the overhead of copying (section 5.4.3) cancels out any gains in these applications. The number of TLB misses using a block data layout coupled with tiling is 91-96% less than that with a canonical (row/column-major) layout when using either a generic access pattern or real applications such as LU or CF. They report that these results hold for a wide range of machines and problem sizes. A further variant in terms of block-layout is zigzag row-major or column-major ordering which is detailed in section 5.8.

## 5.5 RCM Reordering

The results of a Matlab elaboration using 40 large symmetric matrices are shown in Table 5-5. It can be seen that Reverse Cuthill-McKee (RCM) re-ordering [104] shows that the run-time for Sparse Matrix-Vector Multiplications (SMVM) can indeed be reduced. Toledo [101] claims RCM equates to only a few SMVM operations and claims excellent results across a range of matrices. However it can also be seen in Table 5-5 that the actual re-ordering is computationally expensive often equating to tens of SMVM operations. As a result such re-orderings are only of interest for iterative algorithms where a large number of iterations are required to achieve any performance increase using re-ordering.

As can be seen from Table 5-5 tens of iterations are required to justify the cost of RCM but the benefits of RCM scale relatively linearly in terms of the percentage iterations required beyond the breakeven point necessary to achieve a target performance increase of 20%.

A likely explanation for this improvement in performance is that re-ordering produces longer runs of contiguous addresses, which in turn allows the hardware pre-fetch logic in the Pentium4 to work more efficiently. It was found that the benefits of reordering in smaller matrices (less than 100k non-zeroes) were less obvious as the matrices are more likely to fit in the internal cache of the Pentium 4 completely, reducing the miss rate, hence a larger number of iterations is required to see a benefit from reordering, and hence they are not included in the table. These same benefits should accrue to all architectures, which make use of caches, although the exact breakeven points etc. will depend on the specifics of a particular cache implementation.

| matrix name | non zeroes | t_smvm | t_rcm | t_smvm' | N_rc | # SMVMs to break even with RCM | target reduction in exec time | # SMVMs to get +20% perf. using RCM | % extra SMVM iterations for 20% perf. (RCM) |
|---|---|---|---|---|---|---|---|---|---|
| obstclae | 197608 | 0.008709 | 2.456 | 0.00072 | 3415.3 | 307.3 | 0.2 | 393.0 | 27.88% |
| torsion1 | 197608 | 0.008739 | 2.448 | 0.00052 | 4681.2 | 298.0 | 0.2 | 378.5 | 27.02% |
| jnlbrng1 | 199200 | 0.016291 | 2.455 | 0.00055 | 4472.2 | 156.0 | 0.2 | 196.7 | 26.10% |
| minsurfo | 203622 | 0.008999 | 3.446 | 0.00054 | 6358.5 | 407.5 | 0.2 | 517.7 | 27.04% |
| bcsstk28 | 219024 | 0.011171 | 0.345 | 0.00010 | 3413.7 | 31.1 | 0.2 | 39.0 | 25.29% |
| bcsstk25 | 252241 | 0.011557 | 1.314 | 0.00022 | 6001.5 | 115.9 | 0.2 | 145.6 | 25.61% |
| bcsstk16 | 290378 | 0.014074 | 0.363 | 0.00011 | 3273.4 | 26.0 | 0.2 | 32.6 | 25.25% |
| vibrobox | 301700 | 0.014509 | 0.915 | 0.00017 | 5445.4 | 63.8 | 0.2 | 80.0 | 25.37% |
| crystm02 | 322905 | 0.017727 | 0.935 | 0.00018 | 5108.1 | 53.3 | 0.2 | 66.8 | 25.33% |
| gyro_m | 340431 | 0.016434 | 1.416 | 0.00055 | 2588.2 | 89.1 | 0.2 | 112.4 | 26.09% |
| cvxbqp1 | 349968 | 0.089424 | 3.358 | 0.00064 | 5222.0 | 37.8 | 0.2 | 47.4 | 25.23% |
| bcsstk38 | 355460 | 0.015299 | 0.573 | 0.00012 | 4856.4 | 37.7 | 0.2 | 47.3 | 25.24% |
| bcsstk17 | 428650 | 0.019049 | 0.757 | 0.00016 | 4670.4 | 40.1 | 0.2 | 50.2 | 25.27% |
| wathen100 | 471601 | 0.023111 | 2.547 | 0.00053 | 4823.1 | 112.8 | 0.2 | 141.8 | 25.73% |
| gridgena | 512084 | 0.025928 | 3.060 | 0.00073 | 4180.0 | 121.4 | 0.2 | 152.9 | 25.91% |
| wathen120 | 565761 | 0.02459 | 2.321 | 0.00073 | 3170.8 | 97.3 | 0.2 | 122.5 | 25.97% |
| crystm03 | 583770 | 0.028275 | 1.623 | 0.00034 | 4731.4 | 58.1 | 0.2 | 72.8 | 25.38% |
| finan512 | 596992 | 0.030109 | 4.656 | 0.00135 | 3438.5 | 161.9 | 0.2 | 204.8 | 26.49% |
| Pres_Poisson | 715804 | 0.031677 | 1.085 | 0.00023 | 4736.0 | 34.5 | 0.2 | 43.2 | 25.23% |
| gyro_k | 1021159 | 0.052122 | 1.341 | 0.00024 | 5543.3 | 25.9 | 0.2 | 32.4 | 25.15% |
| bcsstk36 | 1143140 | 0.049583 | 1.653 | 0.00031 | 5315.7 | 33.6 | 0.2 | 42.0 | 25.20% |
| bcsstk35 | 1450163 | 0.067312 | 2.174 | 0.00044 | 4997.7 | 32.5 | 0.2 | 40.7 | 25.20% |
| qa8fm | 1660579 | 0.124468 | 4.395 | 0.00121 | 3640.9 | 35.7 | 0.2 | 44.7 | 25.31% |
| qa8fk | 1660579 | 0.076426 | 4.380 | 0.00125 | 3495.6 | 58.3 | 0.2 | 73.1 | 25.52% |
| oilpan | 2148558 | 0.151974 | 4.976 | 0.00140 | 3554.3 | 33.0 | 0.2 | 41.4 | 25.29% |
| vanbody | 2329056 | 0.10066 | 3.415 | 0.00062 | 5472.4 | 34.1 | 0.2 | 42.7 | 25.20% |
| ct20stif | 2600295 | 0.114401 | 3.790 | 0.00067 | 5690.4 | 33.3 | 0.2 | 41.7 | 25.18% |
| nd3k | 3279690 | 0.145184 | 1.324 | 0.00013 | 10027.0 | 9.1 | 0.2 | 11.4 | 25.03% |
| t3dh_e | 4352105 | 0.191977 | 5.790 | 0.00149 | 3878.0 | 30.4 | 0.2 | 38.1 | 25.25% |
| nd6k | 6897316 | 0.293898 | 2.761 | 0.00024 | 11409.6 | 9.4 | 0.2 | 11.8 | 25.03% |
| bmw7st_1 | 7318399 | 0.403275 | 10.929 | 0.00243 | 4504.8 | 27.3 | 0.2 | 34.1 | 25.19% |
| hood | 9895422 | 0.450926 | 18.407 | 0.00389 | 4733.1 | 41.2 | 0.2 | 51.6 | 25.27% |
| crankseg_1 | 10614210 | 0.457083 | 5.950 | 0.00069 | 8573.2 | 13.0 | 0.2 | 16.3 | 25.05% |
| bmwcra_1 | 10641602 | 0.493584 | 13.599 | 0.00270 | 5038.4 | 27.7 | 0.2 | 34.7 | 25.17% |
| pwtk | 11524432 | 0.53264 | 16.751 | 0.00382 | 4382.7 | 31.7 | 0.2 | 39.7 | 25.23% |
| crankseg_2 | 14148858 | 0.607738 | 7.677 | 0.00081 | 9512.9 | 12.6 | 0.2 | 15.8 | 25.04% |
| nd12k | 14220946 | 0.633305 | 5.696 | 0.00048 | 11767.9 | 9.0 | 0.2 | 11.3 | 25.02% |
| af_shell4 | 17562051 | 1.072393 | 45.274 | 0.01101 | 4112.5 | 42.7 | 0.2 | 53.5 | 25.32% |
| af_shell8 | 17579155 | 1.141235 | 43.379 | 0.00917 | 4731.0 | 38.3 | 0.2 | 48.0 | 25.25% |
| af_shell7 | 17579155 | 1.237358 | 45.226 | 0.01415 | 3196.0 | 37.0 | 0.2 | 46.4 | 25.36% |

**Table 5-5 Effect of RCM Reordering on SMVM Performance1**

---

1 t_smvm time to perform SMVM without reordering
t_smvm' time to perform SMVM with reordering
t_rcm time to perform RCM reordering
N_rc t_rcm/t_smvm' (# smvm iterations to justify cost of RCM)

## 5.6 Exploiting Parallelism

Scientific and engineering applications such as SMVM are not necessarily either written in such a way as to take advantage of parallelism. Even "embarrassingly parallel" applications such as 3D graphics [92] may be unable to take advantage of all of the available parallelism owing to how the problem maps to the underlying hardware. Specifically with respect to SMVM kernels a number of approaches allow inherent parallelism to be exploited:

- Code partitioning through the restructuring of SMVM kernels using parallel programming constructs such as those embodied in OpenMP
- Sparse Matrix (data) partitioning using graph-based techniques into sub-matrices which can be multiplied separately on different cores

Sparse Matrix partitioning is by definition data-dependent and must be performed on a matrix-by-matrix basis whereas transforming SMVM kernels to exploit parallelism using OpenMP is generic and applicable to all matrices.

### 5.6.1 OpenMP

The OpenMP API (application program interface) [128] was introduced in 1997 in an effort to standardise programming models for shared memory systems, and is used to explicitly direct multithreaded, shared memory parallelism. The API consists of 3 components;

- Compiler Directives,
- Runtime Library Routine
- Environment Variables

OpenMP supports the C, C++ and FORTRAN programming languages as well as a range of operating systems and all OpenMP programs begin as a single sequential process or Master Thread as depicted in Figure 5-8.

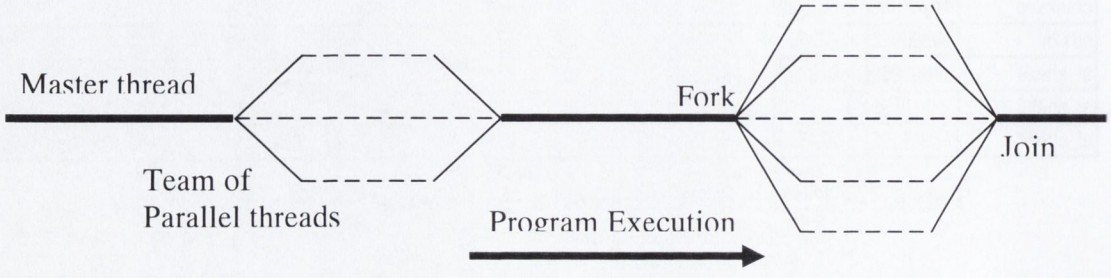



**Figure 5-8 OpenMP Program Execution**

The user also specifies necessary synchronization like locks, barriers, etc to ensure correct execution of the parallel region. At runtime, threads are forked for the parallel region and are typically executed in different processors sharing the same memory and address space. Statements enclosed by the parallel region construct are executed in parallel. A Join occurs at the end of parallel constructs, where the threads synchronize and terminate after completing the statements in the parallel construct. The structure of a typical OpenMP program is shown in Listing 5-4.

```
#include <omp.h>
main () {
  int var1, var2, var3;
  Serial code
  ...
  #pragma omp parallel private(var1, var2) shared(var3) {
    Parallel section executed by all threads
    ...
    All threads join master thread and disband
  }
  Resume serial code
}
```

**Listing 5-4 Structure of Typical OpenMP Program**

Relatively little exists in terms of published results for the performance of CMP and SMT systems using OpenMP. According to [129] Integrating both CMP and SMT into one processor means that threads interact in a more complex manner which is not addressed by the current version of OpenMP, especially for systems composed of multiple CMPs.

This view is shared by Curtis-Maury et al. [92] who evaluated the performance of OpenMP applications from the NAS Parallel Benchmarks suite on a real SMT system versus a simulated CMP system with similar hardware. They found that SMTs suffer from resource contention, whereas CMPs are more efficient (and cost effective) due to greater resource duplication. In terms of application performance 30% of the test-cases showed a marked reduction in performance when a second thread per processor was enabled in SMTs. On the CMP-based multiprocessor the activation of the second

core always resulted in performance improvements. However in a majority of cases (8/14) placing the required number of threads on as few processors as possible resulted in higher performance than spreading them across all of the available processors. They concluded that existing OpenMP code scales better on CMPs than SMTs, and that to maximize the efficiency of OpenMP on SMTs, new capabilities are required by the runtime environment and/or the programming interface.

Packirisamy and Barathvajasankar in [130] suggest that given the similarity with clusters, OpenMP can be extended to take advantage of the potential for both fine-grained and thread-level parallelism in CMPs. The authors propose a technique for multi-core environments, where say the outer loop is parallelized between processors and the inner loop is parallelized for the processing elements inside each processor.

Using OpenMP, Kotakemori et al. [131] have benchmarked the performance of 7 sparse storage formats, as well as conversions between these same formats using a 16-node Itanium2 based system capable of supporting up to 32 simultaneously executing threads. The performance figures obtained confirm Vuduc's assertion [52] that BCSR is the optimal sparse matrix storage format for the Itanium2, offering a significant performance advantage of between 1.13 and 2.59x over CSR (average 1.8x) depending on the source matrix used, even when a single thread is used. In most cases the speedup achieved by additional threads is linear up to 32 threads whereupon the speedup achieved by doubling the number of processors falls sharply due to resource-sharing when 2 threads run on each processor node. In all but one case (matrix f which is a diagonal matrix) BSR offers the highest performance. As with Vuduc's work the overhead in converting between CSR and BCSR format is non-trivial and means to 10s to 100s of SMVM operations would have to be performed in order to make the conversion worthwhile in terms of overall run-time.

## 5.7 Matrix Partitioning

Partitioning and load balancing are important issues in parallel scientific computing using multiprocessors or CMPs, the goal being to distribute the workload among the available processors in a way that minimises communication cost and maximises performance. The most common approach to distributing a Sparse Matrix multiplication across multiple processors is to divide the matrix by rows or columns so as to approximately divide the workload for instance where a CSR SMVM is distributed by rows as the compiler splits the outer loop across the available

processors. A more generic approach is to use graph partitioning to perform load balancing. In this case data are represented as vertices in a graph, and edges represent dependencies between data. Graph partitioning attempts to minimize the number of cross-edges in the graph between processors, as each such edge results in communication between processors.

According to Bisseling and Vastenhouw [132] the distribution of a Sparse Matrix multiplication and associated matrix and vectors data and results consists of four phases. A good distribution scheme should achieve the following objectives:

- Spread the matrix non-zeros evenly across the processors
- Minimize *communication volume*, i.e. total # data words communicated
- Spread communication (sending and receiving) evenly over the processors
- Partition matrix in both dimensions, e.g., by splitting it into rectangular blocks

Mondriaan [132] differs from the majority of partitioners now in use which perform a 1-D partitioning in that it partitions in 2 dimensions. According to the authors Finite-Element matrices are unlikely to benefit greatly from the two-dimensional partitioning approach taken in Mondriaan. The reason for this is that the square non-symmetric matrices required by iterative algorithms such as GMRES, QMR, BiCG, and Bi-CGSTAB impose an additional constraint on the input and output vector distribution which makes it more difficult to balance the communication, and may even lead to an increase in communication volume. In fact the original application of Mondriaan in fact is the design of a parallel web-search engine based on latent semantic indexing.

Zoltan is a hypergraph partitioner similar to Mondriaan. When compared to the other partitioners detailed here Zoltan [134] is run as a parallel task rather than sequentially on a single CPU. One of the challenges in the design of a good parallel partitioner is load-balancing within the partitioner itself! The speedup achieved by using 64 CPUs to perform the partitioning ranges from 0-25 times faster than using a single CPU.

The published results in terms of run-times for Zoltan clearly show the advantage of parallelising of the partitioning task. However, partitioning leads to overheads of hundreds or thousands of un-partitioned CSR SMVMs which must be amortised before any benefit will arise from the use of Sparse Matrix partitioning.

## 5.8 Cache Oblivious SMVM Partitioning

In [135] Yzelman and Bisseling introduce a cache-oblivious method for sparse matrix–vector multiplication based on the hypergraph-based sparse matrix

partitioning methods used in Mondriaan [132]. In their scheme partitioning is performed such as to induce cache-friendly behaviour during sparse matrix–vector multiplication. The authors also presented new variants of the CSR and ICSR where the zigzag format refers to the alternation of column elements from row to row, thus maximizing the likelihood of cache hits from row to row of the sparse matrix as the SMVM computation progresses.

The column-ordering in zigzag versus normal CSR format are shown in Figure 5-9. Experimental results demonstrate a saving in computation time up to 50% in one case. However that the method performs best on sparse matrices with relatively low numbers of non-zeroes per row or column and which are not already ordered favourably. They also note that as cache size increases the benefits of the reordering scheme decrease as might be expected. The ICSR format was also found to be noticeably faster than CSR on the architectures targeted, however overall the overhead of the proposed reordering and partitioning scheme ranges from hundreds to over one hundred thousand unoptimized SMVM products depending on the matrix used and the number of experimental reorderings performed. It is worth noting that the zigzag orderings proposed by the authors could also be applied to BCSR and other related sparse storage formats.

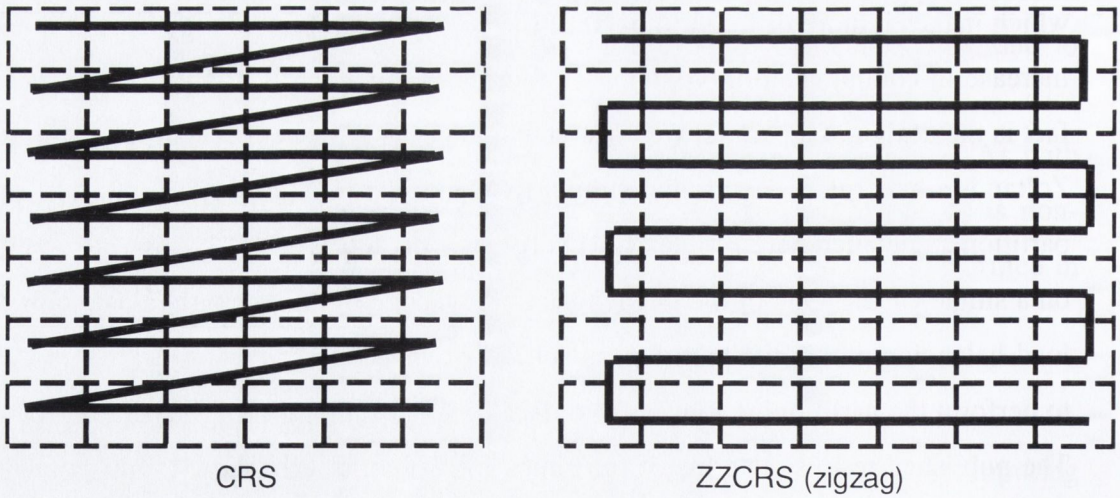



**Figure 5-9 ZZCSR Column-Ordering**

## 5.9 Summary

According to Gropp et al [26] applications which are dominated by sections of bandwidth-limited code such as SMVM are doomed to achieve an ever decreasing fraction of peak performance due to the widening processor-memory performance gap

or "Memory Wall". In brief, memory bandwidth presents an upper bound on the performance of such applications which compilers cannot improve requiring a new approach on the part of application developers to circumvent the bottleneck. One possibility suggested by Gropp et al is that SMVM operations using multiple rather than a single vector might be used by transforming the application code, thus improving overall performance. This optimization has been thoroughly investigated by both Im [112] and Vuduc [52] and has been shown to offer significant speed-ups, where applicable.

Within these bounds code and data can be tuned either manually or automatically to extract as much performance as possible from the processor hopefully getting close to the bandwidth bound. The tuning algorithms proposed to date target only SMVM performance and not pin bandwidth or power consumption which is regarded by the author as a severe limitation in an energy-conscious world where the electricity consumed by server farms is of real business concern in the siting and running of server farms by corporations such as Google [136].

As previously seen, caches, and local registers can be used to improve the performance of SMVM codes by improving average access times. Blocking and other transformations can be used to improve temporal and spatial locality allowing the highest performance possible to be achieved for a given processor architecture, although as will be seen later these transformations do impose a significant start-up cost which must be amortised by performing multiple SMVM operations on the same transformed data. There is also evidence to suggest that where multiple levels of cache are used cache-line lengths should increase rather than all having the same length for optimal performance.

An "elephant in the room" with regard to all of the SMVM published work is that the overhead in converting between CSR and BCSR format is non-trivial and means up to 10s to 100s of SMVM operations would have to be performed in order to make the conversion worthwhile in terms of overall run-time.

The important contributions by other researchers are noted in the following sections:

Prefetching

- According to Mowry [51] and Lam et al [119] locality optimizations (blocking) *reduce* latency, unlike prefetching which *tolerate* latency

- Locality optimizations reduce memory bandwidth requirements but are limited in their applicability because any associated code-transformations must be legal, while prefetching is not subject to these constraints and is more broadly applicable

Register-Blocking

- Register-Blocking improves performance through register reuse and lowering the indexing overhead at the expense of data-dependent zero-fill
- A matrix density $10^{-3}$ is required in the sparse source matrix to gain any benefit from register blocking (Nishtala et al [115])
- 10s to 100s of equivalent unoptimized SMVMs must be amortised to make the search and blocking effort worthwhile

Cache-Blocking

- Cache-blocking improves performance by improving temporal locality of source vector accesses, only makes sense when the source vector $x$ doesn't fit in cache
- The technique is of use in a restricted class of sparse matrices [115] which are less dense ($10^{-5}$) than those for which register-blocking is appropriate ($10^{-3}$).

Cache-Implementation

- Multi-word cache lines reduce the number of cache misses but increase the amount of memory traffic.
- Set-associativity improves the average cache miss-rate but does not address wide variations in miss-rate between problem sizes.

TLB-Blocking

- Small TLBs such as that used in the Pentium4 Xeon can result in poor performance as TLB misses cause valid data in the cache to be evicted
- Padding can improve TLB performance preventing conflict misses caused because multiple pages map to the same entry in direct-mapped TLBs
- In the case of fully-associative TLB the block size should be less than or equal to the number of TLB entries in order to avoid TLB thrashing

Copying

- copy optimisation is used to copy non-contiguous data into contiguous cache locations in such a way as to eliminate self-interference and can significantly

decrease the number of TLB misses, but only if the reuse factor is not low, as in the case of SMVM

Recursive Blocking

- Improves temporal locality and provides efficient cache and TLB blocking, while register blocking is used within sub-blocks.
- Performance is almost flat as a function of problem size and type eliminating the need for searches to obtain good performance however performance still falls short of hand-coded vendor libraries

Combining techniques

- According to Im [112] there is no benefit to combining register and cache-blocking
- Temam et al [120] conclude that the order in which blocking transformations should be applied is bottom-up, i.e. TLB blocking should follow cache blocking, which should follow register-blocking and show the interaction between the techniques step-by-step (in 10 discrete combinations) for matrix multiplication
- Lam et al [119] recommend combining cache-blocking with copying where possible as it achieves the lowest level of cache misses and consistent improvement independent of the data set
- Prasanna et al. report [127] that a combination of Block Data Layout and Tiling significantly improves performance of both caches and TLB for a large range of machines and problem sizes and is preferable to padding and copy optimisation

Search

- Yotov et al. [106] cast doubt on the generality of the work performed by Vuduc suggesting that the Itanium2 benefits disproportionately from the proposed technique. They suggest that for a range of other processors a simple mathematical model will suffice potentially making register blocking attractive even where the number of SMVM iterations is very low

RCM Reordering

- Experimentally it was found that the overhead of RCM reordering numbers 10s to 100s of equivalent un-reordered SMVMs, thus limiting the utility of

RCM to iterative algorithms where a large number of iterations are required to achieve any performance increase using re-ordering

OpenMP

- Given the similarity between clusters and CMPs it is suggested that OpenMP can be extended as shown by Packirisamy and Barathvajasankar in [130] to take advantage of the potential for both fine-grained and thread-level parallelism in CMPs. The authors propose a technique for multi-core environments, where the outer loop is parallelized between processors and the inner loop is parallelized for the processing elements inside each processor to optimise performance

Sparse Matrix Partitioning

- Experimentally is was found that partitioning using Mondriaan incurs overheads equivalent to 100s to 1000s of optimised CSR SMVMs of the un-partitioned source-matrix
- An interesting by-product of the work by Bisseling et al [135] is that if the rows in the matrix are scanned alternately from left to right ant then right to left, performance can be improved. The ZZCSR and ZZICSR formats proposed by the authors could be extended to blocked formats such as BCSR, to create zigzag formats such as ZZBCSR

# 6

# Chapter 6

*"A designer knows he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away."*

*- Antoine de Saint-Exupery*

## 6 Software SMVM Revisited

As was seen in chapter 5 the state of the art methods for computing Sparse Matrix Vector Products (SMVM) have improved little over the past few decades and performance improvements have been driven largely by advances in processor and semiconductor process technology. The focus has been rather on tuning existing methods such as the work by Im [137] and Vuduc [52] on tuning BCSR where the sparse matrix has some underlying structure, often in the case of non-structured matrices such as the Google-Matrix, BCSR and related methods offer no improvement and may in fact disimprove results if applied to such matrices. In general SMVM has had little if any influence on the design of mainstream microprocessors as outlined in chapter 4 despite the obvious problems in terms of scaling I/O bandwidth performance, particularly where Chip Multi-Processors (CMPs) exacerbate problems by contending for increasingly scarce I/O bandwidth. A key observation in the work of Vuduc et al is [137] [52] that a sizeable number of the entries in typical blocked sparse-matrices consist of zero fill. These values even if they do not contribute to the result of an SMVM are nonetheless fetched as 64-bit

double-precision values from memory and multiplied with all of the attendant problems in terms of power-dissipation and system throughput. An obvious improvement to BCSR and other blocked SMVM schemes would be to find some way of avoiding trivial operations due to zero-fill, whether storing or loading these values to memory, moving them via shared busses or indeed performing arithmetic operations using these zero fill-in values.

## 6.1 Trivial Arithmetic

According to Richardson [138] and Lilja [139] a significant number of trivial computations are performed during the execution of processor benchmarks and other numerically intensive applications. By trivial computations Richardson intended those that can be simplified or where the result is zero, one, or equal to one of the input operands. It was shown that for certain programs studied, up to 67% of operations were trivial and fast detection and evaluation of these trivial operations using dedicated hardware in the pipeline yielded significant speedup. However, to date, work on the exploitation of trivial operands appears to be focused on dynamically occurring trivial operands in the pipeline rather than static trivial operands occurring in the input data and no published work appears to detail the exploitation of trivial operands in Sparse Matrix compression.

| Operation | Normal | Bypassable |
|---|---|---|
| Add | X+Y | X=0<br>Y=0 |
| Subtract | X-Y | Y=0<br>X=Y |
| Multiply | X*Y | X=0,+1,-1<br>Y=0,+1,-1 |
| Divide | X/Y | X = {0, Y, -Y}<br>Y=1 |
| Absolute Value | \|X\| | X=0<br>X={positive} |
| Square Root | $(X)^{0.5}$ | X=0, 1 |

**Table 6-1 IEEE Arithmetic trivial operation table**

Furthermore as the detection of, and bypass of, dynamically occurring trivial operands as proposed by Richardson introduces problems of its own (e.g. Pipeline bubbles due

to the difference in FPU latency between trivial and non-trivial paths) such features have, to date, not appeared in commercial architectures. It has been shown [139] that approximately 30% of all arithmetic instructions (when averaged over a large number of integer and floating-point benchmarks), accounting for 12% of all dynamic instructions, are trivial computations. These trivial operations occur despite the use of compiler optimisation techniques, and are not heavily dependent on the program's specific input values. One of the reasons for this is that trivial values are created dynamically during program execution by mathematical operations e.g. cancellation $i - i = 0$ and by multiplication by zero. From Table 6-1 derived from [139] it can be seen that only a few of the proposed trivial operations suggested are relevant to iterative methods and matrix/vector operations.

This author notes that although not mentioned by Richardson or Lilja it is possible that trivial multiplication could be extended to the general case of multiplication by powers of 2 resulting in a small unit which adds exponents and leaves the mantissa of the multiplicand unmodified in a manner similar to that proposed in [139].

## 6.2 Computing with bitmaps

The main issue with the method proposed by Richardson is that triviality is determined by floating-point comparators which in turn choose to compute the trivial results either using a complete floating-point unit or alternately a lower-latency and lower-complexity unit in the case one or both of the inputs are trivial.

While offering some promise, in the form of reduced latency and potentially higher throughput, this approach has several disadvantages:

- Programs must take account of variable latency
- Additional hardware including comparators, multiplexers and special FP units
- Memory bandwidth and storage requirements are not reduced as zeroes (trivial values) are stored as full precision floating-point numbers before being fetched and acted upon by the processor

A much better approach would be to perform the floating-point (or integer) comparisons off-line, and store these decisions along with the data in such a way that only non-trivial computations are actually carried out on full-precision data. This approach has the advantages that:

- Minimal additional hardware is required (1-bit multiplier is an AND gate)
- Storage and bus bandwidth requirements are minimised

- Latency remains constant
- Power is minimised

### 6.2.1 Reference BCSR SMVM

The BCSR data-structure consists of 3 arrays as shown in Figure 6-1. The row (row_start) array holds the row entries containing non-zero tiles, a second col (col_idx) array containing the column addresses of the non-zero tiles and a val (value) array containing the actual non-zero entries (with fill) for all of the non-zeroes in the Sparse Matrix, arranged in tile-by-tile order.

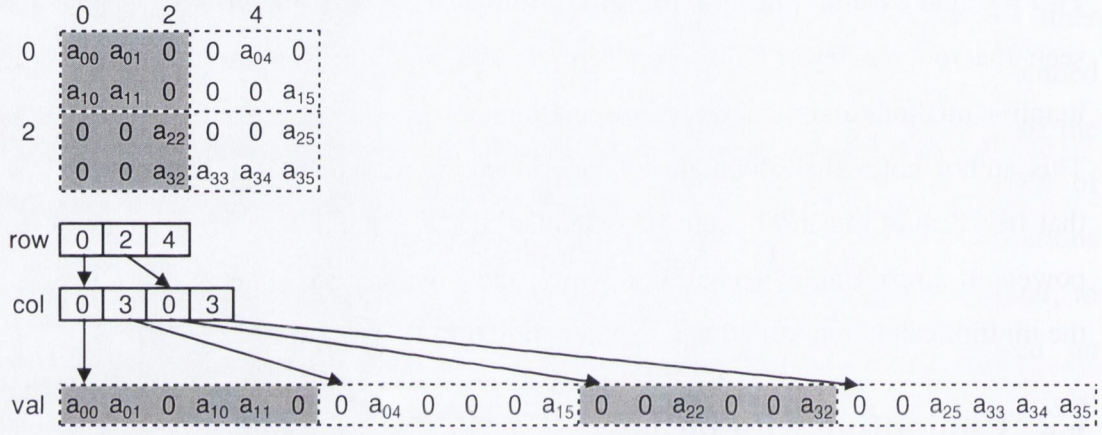



**Figure 6-1 2x3 BCSR Sparse Matrix Storage Format**

A generic BCSR Sparse Matrix Vector Product (Multiplication) code which operates on the BCSR data-structure is shown in Listing 6-1.

```
L1.  void bcsr_smvm(int bm, int r, int c, int *row_start,
   int *col_idx, double *value, double *src, double
   *dest) {

L2.        int i, j, ii, jj;

L3.        for (i=0; i<bm; i++, dest+=r) {
L4.          for (j=row_start[i]; j<row_start[i+1]; j++,
   col_idx++, value+=r*c) {
L5.            for (ii=0; ii<r; ii++) {
L6.              for (jj=0; jj<c; jj++) {
L7.                dest[ii] += value[ii*c + jj] *
   src[(*col_idx) + jj];
L8.              }
L9.            }
L10.       }
L11.     }
L12.  } // bcsr_smvm()
```

**Listing 6-1 Generic BCSR SMVM C-code**

While having the benefit of being easy to understand and generic (can handle arbitrary sized r*c BCSR tiles) this C code results in a highly inefficient

implementation in terms of performance as many address calculations have to be performed for each r*c products necessary to complete the SMVM operation.

In practice a library of C-functions is written, one for each of the r*c tile sizes required by the library designer. Each of these C-functions is either written by hand or generated by another program and each must be extensively verified to ensure correct operation. For reference purposes the code for a 4x4 BCSR Sparse Matrix Vector Multiplication (SMVM) is shown in **Listing 6-2**. As can be seen the matrix vector code is optimised for performance by unrolling which removes loop calculations and assigning values which are frequently reused to variables which the C compiler will assign to registers. Both techniques are commonly used in optimised SMVM codes.

```
L1.     void bcsr_smvm4x4(int bm, int *row_start, int
   *col_idx, double *value, double *src, double *dest)
   {

L2.       int i, j, r=4, c=4;
L3.       Type y0, y1, y2, y3, x0, x1, x2, x3;

L4.       for (i=0; i<bm; i++, dest+=4) { // r
L5.         y0 = dest[0];
L6.         y1 = dest[1];
L7.         y2 = dest[2];
L8.         y3 = dest[3];
L9.         for (j=row_start[i]; j<row_start[i+1]; j++,
   col_idx++, value+=16){ // r*c
L10.          x0 = src[(*col_idx)    ]; // unrolled
   loop
L11.          x1 = src[(*col_idx) + 1];
L12.          x2 = src[(*col_idx) + 2];
L13.          x3 = src[(*col_idx) + 3];
L14.          y0 += value[ 0] * x0; // row 0
L15.          y0 += value[ 1] * x1;
L16.          y0 += value[ 2] * x2;
L17.          y0 += value[ 3] * x3;
L18.          y1 += value[ 4] * x0; // row 1
L19.          y1 += value[ 5] * x1;
L20.          y1 += value[ 6] * x2;
L21.          y1 += value[ 7] * x3;
L22.          y2 += value[ 8] * x0; // row 2
L23.          y2 += value[ 9] * x1;
L24.          y2 += value[10] * x2;
L25.          y2 += value[11] * x3;
L26.          y3 += value[12] * x0; // row 3
L27.          y3 += value[13] * x1;
L28.          y3 += value[14] * x2;
L29.          y3 += value[15] * x3;
L30.        }
L31.        dest[0] = y0;
```

```
L32.          dest[1] = y1;
L33.          dest[2] = y2;
L34.          dest[3] = y3;
L35.        }
L36.    } // bcsr_smvm4x4()
```

**Listing 6-2 4x4 BCSR SMVM**

The following gcc command line options can be used to produce annotated assembly language output:

gcc -c -g -Wa,-ahl=test_jtsmvm.asm test_jtsmvm.cpp

Using this command, the x86 assembly language corresponding to the following line of C-code, has been produced as shown in Listing 6-3.

y0 += value[0]*x0;

```
65:test_jtsmvm.cpp ****       y0 += value[       0] * x0;
606 .stabn 68,0,65,LM40-__Z16bcsr_smvm4x4_iPiS_PdS0_S0_
607                   LM40:
608 02ce 8B4514             movl  20(%ebp), %eax
609 02d1 DD00               fldl  (%eax)
610 02d3 DC4DC8             fmull -56(%ebp)
611 02d6 DD45E8             fldl  -24(%ebp)
612 02d9 DEC1               faddp %st, %st(1)
613 02db DD5DE8             fstpl -24(%ebp)
```

**Listing 6-3 4x4 BCSR SMVM x86 Assembler**

As can be seen one line of C-code is translated by the C-compiler into a total of six x86 instructions on lines 608-613 in the listing but containing no branches or other control flow instructions. Obviously if the A matrix entry (value[0]) is zero then the six instructions have been executed needlessly, leading to unnecessary consumption of bandwidth and power.

### 6.2.2 Bitmap Block Compressed Sparse Row Format (BBCSR)

The BBCSR data-structure consists of 4 arrays as shown in Figure 6-2. The BBCSR structure augments the 3 BCSR arrays (row_start, col_idx and value) with a bitmap_idx array containing a bitmap, each entry (bit) of which denotes whether a non-zero value is present at that position in the tile or not. The value array contents differ from those in a BCSR data-structure in that only the actual non-zero entries are stored without any zero-fill (unless the 1-bit entries in the bitmap are counted) for all of the non-zeroes in the Sparse Matrix, arranged in tile-by-tile order.

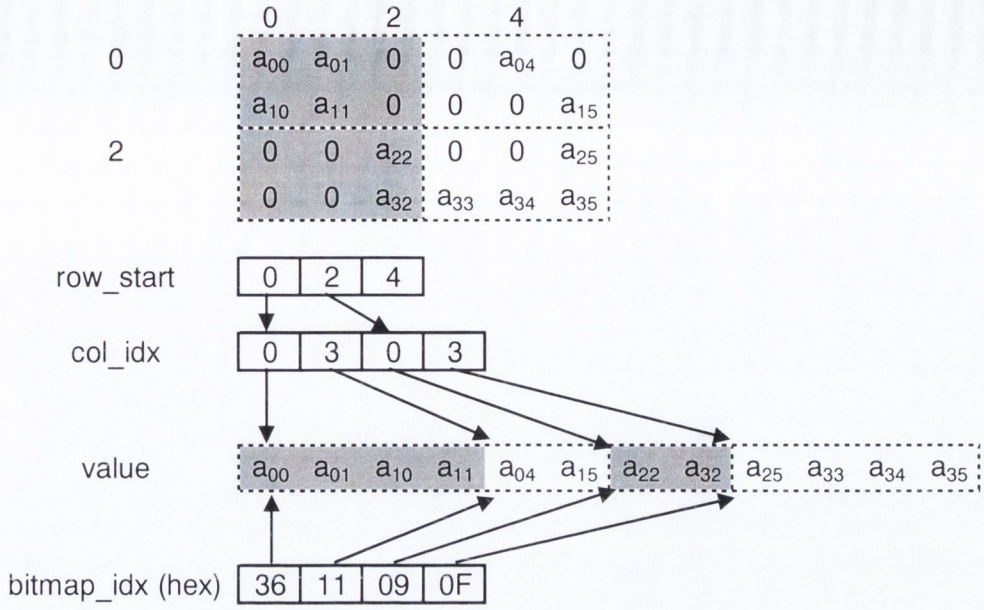



**Figure 6-2 2x3 BBCSR Sparse Matrix Storage Format**

The bitmap entries shown in Figure 6-2 are hexadecimal values, for instance the first entry 36 (hex) corresponds to the binary bitmap (00)11 0110. The C-code for a generic BBCSR SMVM which operates on the BBCSR data-structure is shown in Listing 6-4. One of the main differences with respect to the BCSR SMVM is evident on L10 of the code where a bitmap bit corresponding to a sub-tile entry is tested to see if a non-zero entry is present in the value array. The floating-point multiplication and addition and associated array look-ups and address arithmetic on L12-14 are only performed if the bitmap bit indicates a non-zero is present.

From this analysis it is obvious that if the dense sub-tile of the Sparse Matrix contains zero fill a long sequence of instructions including floating-point instructions with long latency will be replaced by a simple integer operation to test an array bit and skip these costly operations in the case of a zero fill entry. The converse is also true in that if the tile contains little or no fill additional bit-test instructions will be added as overhead to the required sequences of instructions for the evaluation of tile non-zero entries. Apart from the number of zero fill values there are also the effects of the higher instruction count on the cache hierarchy to be considered making a static analysis of the code of little quantitative use.

```
L1.   void bbcsr_smvm(int bm, int r, int c, int *row_start, int
   *col_idx, int *bitmap_idx, double *value, double *src,
   double *dest) {

L2.      int i, j, ii, jj, i_, _nz, bitmap, nz;
L3.      int test_bit;
```

```
L4.     for (i=0; i<bm; i++, dest+=r) {
L5.       for (j=row_start[i]; j<row_start[i+1]; j++,
   bitmap_idx++, col_idx++, value+=nz) {
L6.         bitmap = *bitmap_idx     & 0x0000FFFF;
L7.         nz     = *bitmap_idx>>16 & 0x0000FFFF;
L8.         _nz    = 0;
L9.         for (i_=0; i_<(r*c) && _nz<nz; i_++) {
L10.          test_bit = bitmap&(1<<((r*c)-i_-1));
L11.          if (test_bit) {
L12.            ii = (i_ & 0x0000000C) >> 2;
L13.            jj =  i_ & 0x00000003
L14.            dest[ii]+=value[_nz++]*src[(*col_idx)+jj];
L15.          }
L16.        }
L17.      }
L18.    }
        L19.      } // bbcsr_smvm()
```

**Listing 6-4 Generic BBCSR SMVM Code**

Similar to BCSR the BBCSR SMVM code can be unrolled for performance reasons as shown in **Listing 6-5**. The BBCSR data-structure contains 4 arrays, three of which are the same as BCSR (row_start, col_idx and value), augmented by an array of bitmaps describing the pattern of non-zeroes within an r*c BBCSR tile.

```
L1.     void bbcsr_smvm4x4_ur3(int bm, int r, int c, int
   *row_start, int *col_idx, int *bitmap_idx, double *value,
   double *src, double *dest) {

L2.       int i, j, _nz, bitmap, nz;
L3.       double y0, y1, y2, y3, x0, x1, x2, x3;

L4.       for (i=0; i<bm; i++, dest+=r) {
L5.         y0 = dest[0];
L6.         y1 = dest[1];
L7.         y2 = dest[2];
L8.         y3 = dest[3];
L9.         for (j=row_start[i]; j<row_start[i+1]; j++,
   bitmap_idx++, col_idx++, value+=nz) {
L10.          bitmap = *bitmap_idx     & 0x0000FFFF;
L11.          nz     = *bitmap_idx>>16 & 0x0000FFFF;
L12.          _nz    = 0;
L13.          x0 = src[(*col_idx)    ];
L14.          x1 = src[(*col_idx) + 1];
L15.          x2 = src[(*col_idx) + 2];
L16.          x3 = src[(*col_idx) + 3];
L17.          if (bitmap&32768) y0+= value[_nz++] * x0;
L18.          if (bitmap&16384) y0+= value[_nz++] * x1;
L19.          if (bitmap& 8192) y0+= value[_nz++] * x2;
L20.          if (bitmap& 4096) y0+= value[_nz++] * x3;
L21.          if (bitmap& 2048) y1+= value[_nz++] * x0;
L22.          if (bitmap& 1024) y1+= value[_nz++] * x1;
```

```
L23.            if (bitmap&  512) y1+= value[_nz++] * x2;
L24.            if (bitmap&  256) y1+= value[_nz++] * x3;
L25.            if (bitmap&  128) y2+= value[_nz++] * x0;
L26.            if (bitmap&   64) y2+= value[_nz++] * x1;
L27.            if (bitmap&   32) y2+= value[_nz++] * x2;
L28.            if (bitmap&   16) y2+= value[_nz++] * x3;
L29.            if (bitmap&    8) y3+= value[_nz++] * x0;
L30.            if (bitmap&    4) y3+= value[_nz++] * x1;
L31.            if (bitmap&    2) y3+= value[_nz++] * x2;
L32.            if (bitmap&    1) y3+= value[_nz++] * x3;
L33.          }
L34.          dest[0] = y0;
L35.          dest[1] = y1;
L36.          dest[2] = y2;
L37.          dest[3] = y3;
L38.        }
L39.      } // bbcsr_smvm4x4_ur3()
```

**Listing 6-5 Unrolled BBCSR 4x4 SMVM Code**

A 1-bit in a bitmap indicates that the corresponding value in the value array is non-zero, and a 0-bit in the bitmap indicates that no value is stored in the value array. Each non-zero is therefore represented by 65-bits between the value and the bitmap arrays. The array of bitmaps called bitmap_idx is an array of 32-bit integers, with the lower 16-bits representing the non-zero pattern in BBCSR tile up to a 4x4 and the upper 16-bits representing the non-zero count for the same tile.

The x86 assembler corresponding to one line of C-code :

```
if (bitmap&32768) y0 += value[_nz++] * x0;
```

is shown in **Listing 6-6**.

```
115:test_jtsmvm.cpp ****     if (bitmap&32768) y0 +=
value[_nz++] * x0;
967                     .stabn 68,0,115,LM77-
__Z17bbcsr_smvm4x4_ur3iiiPiS_S_PdS0_S0_
968                     LM77:
969 05a0 8B45F0             movl  -16(%ebp), %eax
970 05a3 C1E80F             shrl  $15, %eax
971 05a6 83E001             andl  $1, %eax
972 05a9 84C0               testb %al, %al
973 05ab 7420               je    L36
974 05ad 8B45F4             movl  -12(%ebp), %eax
975 05b0 8D14C500           leal  0(,%eax,8), %edx
975      000000
976 05b7 8B4520             movl  32(%ebp), %eax
977 05ba DD0402             fldl  (%edx,%eax)
978 05bd DC4DC0             fmull -64(%ebp)
979 05c0 DD45E0             fldl  -32(%ebp)
980 05c3 DEC1               faddp %st, %st(1)
981 05c5 8D45F4             leal  -12(%ebp), %eax
```

```
982 05c8 FF00                incl  (%eax)
983 05ca DD5DE0              fstpl -32(%ebp)
984                     L36:
```

**Listing 6-6 BBCSR x86 Assembler**

Note also that the BBCSR data-structure could be optimised by packing the bitmap as a 64 bit entry into the existing value array, removing the overhead required to address the bitmap_idx array. This optimisation would also allow larger bitmaps supporting tiles up to 8x8 in size rather than the current 32-bit entries in the bitmap_idx array.

### 6.2.3 Experimental Setup

The following section describes the benchmark sparse matrix suite, target machine parameters and C compiler used for SMVM performance measurements.

The target machine used for all experiments was based on an Intel CoreDuo E6600 processor [141] running at 2.4GHz as shown in **Figure 6-3**. This is a modern multicore CPU present in workstations and servers typically used for large scale engineering simulations and Computer Aided Design (CAD).

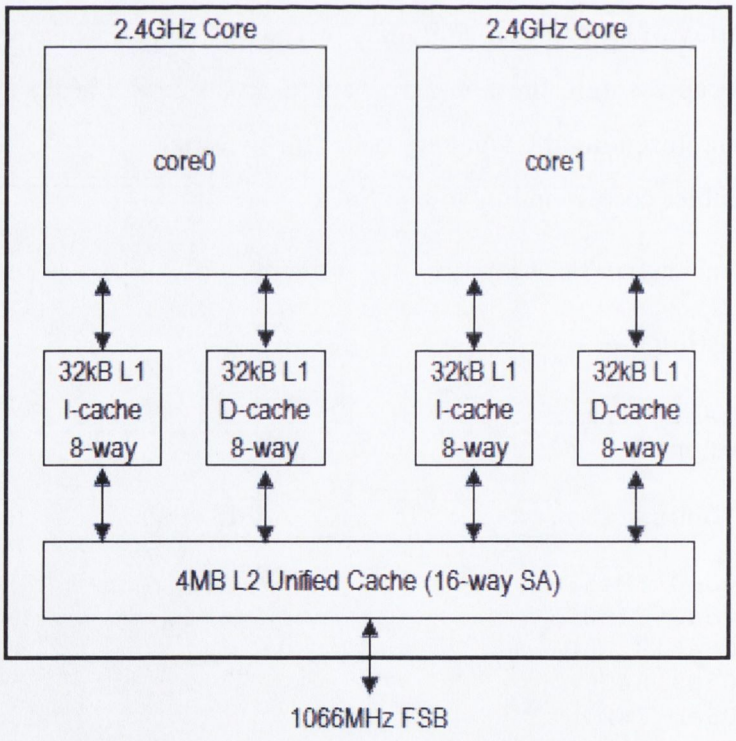



**Figure 6-3 Intel CoreDuo E6600**

When run on the target machine to gauge off-chip bandwidth from the processor, the Stream [45] benchmark (with parameters NTIMEs=250 & 16Mbyte array-size), predicts a system performance of around 120MFLOPS as shown in Table 6-2.

| Function | Rate (MB/s) | Bytes/iter | MFLOPS | Avg time | Min time | Max time |
|---|---|---|---|---|---|---|
| Copy | 2496.7 | 16 | 156.0 | 0.013 | 0.013 | 0.0247 |
| Scale | 2429.9 | 16 | 151.9 | 0.0133 | 0.013 | 0.0222 |
| Add | 2842.8 | 24 | 118.4 | 0.0171 | 0.017 | 0.0286 |
| Triad | 2896.6 | 24 | 120.7 | 0.0167 | 0.017 | 0.0229 |

**Table 6-2 CoreDuo Target System Stream [45] Benchmark Performance**

The benchmark matrix suite of 50 large matrices is drawn from the UF Sparse Matrix and Matrix Market collections as shown in Table 6-3. The matrix suite spans application areas from mechanical engineering structural problems and Finite-Element, to DNA electrophoresis, Computational Fluid Dynamics, Latent Semantic Analysis (web search other than Google search which is based on PageRank and other proprietary techniques), optimisation, materials science and graph theory. In terms of comparison with other works 11 of the 50 chosen matrices also appear in the suite of matrices used by Goumas et al [142]. Similarly to the work of Goumas the C-compiler used is gcc v4.1.2 running under Red Hat Enterprise Edition (RHEL) Linux with the `-O3` command-line options for the most aggressive level of optimisation.
Finally rather than using an approximation of the optimal tile size for the BCSR and BBCSR blocked SMVM each tiling was performed and the complete SMVM product was calculated for the whole matrix.
To give an idea of complexity the 50 sparse matrices contain a total of 120M non-zeroes, and understandably the runtime for the entire sparse matrix suite is very long. In particular as in the case of these experiments no search of the kind proposed by Vuduc is carried out. In fact, as was pointed out previously the overhead of reading in matrices and converting them from coordinate to CSR or blocked formats takes 10s to 100s of equivalent SMVM operations if search is not used to sample part of the matrix. In this work the run time is further compounded by the fact that each of the conversions and SMVM operations is performed for each of 16 possible block-sizes from 1x1 to 4x4.

| | name | M | N | nz | Application |
|---|---|---|---|---|---|
| ***m01*** | af_shell8 | 504855 | 504855 | 9046865 | structural |
| ***m02*** | cage13 | 445315 | 445315 | 7479343 | DNA |
| ***m03*** | nd12k | 36000 | 36000 | 7128473 | 2/3D |
| ***m04*** | crankseg_2 | 63838 | 63838 | 7106348 | structural |
| ***m05*** | pwtk | 217918 | 217918 | 5926171 | structural |
| ***m06*** | hood | 220542 | 220542 | 5494489 | structural |
| ***m07*** | bmwcra_1 | 148770 | 148770 | 5396386 | structural |
| ***m08*** | crankseg_1 | 52804 | 52804 | 5333507 | structural |
| ***m09*** | SHIPSEC5 | 179860 | 179860 | 5146478 | structural |
| ***m10*** | M_T1 | 97578 | 97578 | 4925574 | structural |
| ***m11*** | SHIP_003 | 121728 | 121728 | 4103881 | structural |
| ***m12*** | SHIPSEC1 | 140874 | 140874 | 3977139 | structural |
| ***m13*** | bmw7st_1 | 141347 | 141347 | 3740507 | structural |
| ***m14*** | nd6k | 18000 | 18000 | 3457658 | 2/3D |
| ***m15*** | SHIPSEC8 | 114919 | 114919 | 3384159 | structural |
| ***m16*** | s3dkq4m2 | 90449 | 90449 | 2455670 | structural |
| ***m17*** | THREAD | 29736 | 29736 | 2249892 | |
| ***m18*** | t3dh_e | 79171 | 79171 | 2215638 | moel reduction |
| ***m19*** | 18_tbdlinux | 112757 | 20167 | 2157675 | LSA |
| ***m20*** | gupta2 | 62064 | 62064 | 2155175 | optimization |
| ***m21*** | cage12 | 130228 | 130228 | 2032536 | DNA |
| ***m22*** | s3dkt3m2 | 90449 | 90449 | 1921955 | structural |
| ***m23*** | smt | 25710 | 25710 | 1889447 | structural |
| ***m24*** | oilpan | 73752 | 73752 | 1835470 | structural |
| ***m25*** | nd3k | 9000 | 9000 | 1644345 | ND set |

| | cbb1x1 | 1x1 | 1x2 | 1x3 | 0 |
|---|---|---|---|---|---|
| ***m26*** | 3dtube | 45330 | 45330 | 1629474 | CFD |
| ***m27*** | ct20stif | 52329 | 52329 | 1375396 | structural |
| ***m28*** | raefsky4 | 19779 | 19779 | 1328611 | structural |
| ***m29*** | vanbody | 47072 | 47072 | 1191985 | structural |
| ***m30*** | gupta1 | 31802 | 31802 | 1098006 | optimization |
| ***m31*** | fidap011 | 16614 | 16614 | 1091362 | CFD |
| ***m32*** | bcsstk32 | 44609 | 44609 | 1029655 | structural |
| ***m33*** | turon_m | 189924 | 189924 | 912345 | 2D/3D |
| ***m34*** | qa8fk | 66127 | 66127 | 863353 | FE matrix |
| ***m35*** | bcsstk35 | 30237 | 30237 | 740200 | structural |
| ***m36*** | fidapm11 | 22294 | 22294 | 623554 | CFD |
| ***m37*** | msc10848 | 10848 | 10848 | 620313 | structural |
| ***m38*** | msc23052 | 23052 | 23052 | 588933 | structural |
| ***m39*** | bcsstk37 | 25503 | 25503 | 583240 | structural |
| ***m40*** | bcsstk36 | 23052 | 23052 | 583096 | CFD |
| ***m41*** | cage11 | 39082 | 39082 | 559722 | DNA electrophoresis |
| ***m42*** | e40r0100 | 17281 | 17281 | 553956 | driven cavity |
| ***m43*** | crystk02 | 13965 | 13965 | 491274 | materials |
| ***m44*** | *af23560* | *23560* | *23560* | *484256* | *CFD* |
| ***m45*** | *wathen120* | *36441* | *36441* | *301101* | *Random 2/3D* |
| ***m46*** | *gridgena* | *48962* | *48962* | *280523* | *optimization* |
| ***m47*** | *fidap019* | *12005* | *12005* | *259863* | *FEM* |
| ***m48*** | *wathen100* | *30401* | *30401* | *251001* | *Random 2/3D* |
| ***m49*** | *gyro_m* | *17361* | *17361* | *178896* | *model reduction* |
| ***m50*** | *vibrobox* | *12328* | *12328* | *177578* | *vibroacoustic* |

**Table 6-3 Benchmark Matrix Suite**

### 6.2.4 Comparative BBCSR SMVM Performance

All timings in this work rely on cycle.h which is a platform-independent mechanism for measuring elapsed processor cycles between 2 events from the FFTW package [143]. Unfortunately as cycle.h returns only the number of cycles and not elapsed time it is not possible to accurately correlate the number of cycles with elapsed time and hence produce a MFLOPS figure for the target machine for all of the matrices in the test-suite to compare with the triad MFLOPS and bandwidth figures predicted by Stream [45]. However, a cycle-count is a more realistic measure of relative SMVM time, as only the number of cycles consumed by the SMVM method is tabulated, and not those consumed by the operating system etc.

The results shown in Table 6-4 were tabulated by performing 33 SMVM calculations for each sparse matrix of the 50 matrix benchmark suite. This figure of 33 SMVMs consists of 16 BBCSR tilings (1x1 to 4x4), 16 BCSR tilings and a reference CSR SMVM. A shell script ran each of the SMVMs serially and all results were tabulated in an Excel spreadsheet for analysis and comparison. This contrasts with the work of Vuduc et al who sampled a smaller section of the sparse matrix and inferred a CPU runtime using a parameterised machine model in order to save time in the search for optimal tile size for a given matrix. It was felt that optimal tilings requiring the processing of the entire matrix rather than just a subset would isolate the optimal SMVM method from the tuning effect in Vuduc's approach.

Reading the table from left to right the first column is the matrix name, followed by the number of rows (M), number of columns (N) and the number of explicit non-zeroes in the matrix (nz). In some cases nz may differ from the figure in the MatrixMarket format file as such files often contain at least some explicit zeroes. The next column details the fastest SMVM method of the particular matrix under consideration. In order to give more insight into the relative merits of each of the techniques and yet make the table readable a subset of the 33 SMVM results is shown and subdivided into 3 groups. The first group is the relative ranking for the BBCSR, BCSR and CSR SMVM techniques (1 is fastest and 3 is slowest). The next is the sub-block tiling (rows x columns) associated with the fastest BBCSR and BCSR SMVMs. Finally the last group shows the number of cycles measured using cycle.h for the fastest BBCSR tiling, the fastest BCSR tiling followed by the CSR SMVM cycle-count.

| name | M | N | nz | fastest SMVM | Method tSMVM rank | | | Block Tile Size | | tSMVM (sec) | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | BBCSR | BCSR | CSR | BBCSR | BCSR | BCSR | CSR |
| af_shell8 | 504855 | 504855 | 9046865 | *BCSR* | 2 | 1 | 3 | 3x2,3x3 | 2x2 | 132.00 | 156.47 |
| cage13 | 445315 | 445315 | 7479343 | *CSR* | 2 | 3 | 1 | 2x1 | 2x1 | 112.80 | 90.78 |
| nd12k | 36000 | 36000 | 7128473 | *BCSR* | 2 | 1 | 3 | 3x2,3x3 | 3x3 | 87.23 | 117.94 |
| crankseg_2 | 63838 | 63838 | 7106348 | *BCSR* | 3 | 1 | 2 | 2x3 | 2x1 | 97.45 | 116.83 |
| pwtk | 217918 | 217918 | 5926171 | *BCSR* | 2 | 1 | 3 | 3x2,3x3 | 1x3 | 81.18 | 99.18 |
| hood | 220542 | 220542 | 5494489 | *BCSR* | 2 | 1 | 3 | 4x1 | 2x1 | 81.85 | 100.97 |
| bmwcra_1 | 148770 | 148770 | 5396386 | *BCSR* | 2 | 1 | 3 | 3x2,3x3 | 3x3 | 63.50 | 92.77 |
| crankseg_1 | 52804 | 52804 | 5333507 | *BCSR* | 3 | 1 | 2 | 4x1 | 2x1 | 71.84 | 86.70 |
| SHIPSEC5 | 179860 | 179860 | 5146478 | *BCSR* | 2 | 1 | 3 | 2x1 | 2x1 | 49.29 | 85.04 |
| M_T1 | 97578 | 97578 | 4925574 | *BCSR* | 2 | 1 | 3 | 3x2,3x3 | 3x3 | 43.90 | 115.89 |
| SHIP_003 | 121728 | 121728 | 4103881 | *BCSR* | 2 | 1 | 3 | 3x2,3x3 | 2x1 | 32.06 | 96.27 |
| SHIPSEC1 | 140874 | 140874 | 3977139 | *BCSR* | 2 | 1 | 3 | 3x2,3x3 | 1x2 | 30.80 | 93.72 |
| bmw7st_1 | 141347 | 141347 | 3740507 | *BCSR* | 2 | 1 | 3 | 4x2 | 2x2 | 56.18 | 65.57 |
| nd6k | 18000 | 18000 | 3457658 | *BCSR* | 2 | 1 | 3 | 3x2,3x3 | 3x3 | 42.44 | 57.64 |
| SHIPSEC8 | 114919 | 114919 | 3384159 | *BCSR* | 2 | 1 | 3 | 3x2,3x3 | 1x2 | 27.56 | 79.23 |
| s3dkq4m2 | 90449 | 90449 | 2455670 | *BCSR* | 2 | 1 | 3 | 2x3 | 2x2 | 34.05 | 41.87 |
| THREAD | 29736 | 29736 | 2249892 | *BCSR* | 2 | 1 | 3 | 3x2,3x3 | 3x3 | 19.42 | 52.11 |
| t3dh_e | 79171 | 79171 | 2215638 | *CSR* | 3 | 2 | 1 | 2x1 | 2x1 | 44.50 | 39.81 |
| 18_tbdlinux | 112757 | 20167 | 2157675 | *CSR* | 2 | 3 | 1 | 2x1 | 1x2 | 28.14 | 23.69 |
| gupta2 | 62064 | 62064 | 2155175 | *CSR* | 3 | 2 | 1 | 2x1 | 2x1 | 43.08 | 37.69 |
| cage12 | 130228 | 130228 | 2032536 | *CSR* | 2 | 3 | 1 | 2x1 | 2x1 | 29.68 | 23.55 |
| s3dkt3m2 | 90449 | 90449 | 1921955 | *BCSR* | 2 | 1 | 3 | 2x4 | 2x2 | 28.45 | 33.14 |
| smt | 25710 | 25710 | 1889447 | *BCSR* | 3 | 1 | 2 | 4x1 | 1x2 | 29.08 | 32.51 |
| oilpan | 73752 | 73752 | 1835470 | *BCSR* | 2 | 1 | 3 | 4x1 | 2x1 | 20.62 | 31.27 |
| nd3k | 9000 | 9000 | 1644345 | *BCSR* | 2 | 1 | 3 | 3x2,3x3 | 3x3 | 19.91 | 26.50 |
| 3dtube | 45330 | 45330 | 1629474 | *BCSR* | 2 | 1 | 3 | 3x2,3x3 | 3x3 | 18.87 | 36.07 |
| ct20stif | 52329 | 52329 | 1375396 | *BCSR* | 2 | 1 | 3 | 4x1 | 1x2 | 20.08 | 24.37 |
| raefsky4 | 19779 | 19779 | 1328611 | *BCSR* | 2 | 1 | 3 | 3x2,3x3 | 1x2 | 10.59 | 12.09 |
| vanbody | 47072 | 47072 | 1191985 | *BCSR* | 3 | 1 | 2 | 4x1 | 2x1 | 18.32 | 21.00 |
| gupta1 | 31802 | 31802 | 1098006 | *CSR* | 3 | 2 | 1 | 2x1 | 2x1 | 21.23 | 18.52 |
| fidap011 | 16614 | 16614 | 1091362 | *BCSR* | 3 | 1 | 2 | 2x3 | 2x1 | 9.13 | 10.45 |
| bcsstk32 | 44609 | 44609 | 1029655 | *BCSR* | 3 | 1 | 2 | 4x1 | 3x1 | 16.13 | 17.80 |
| turon_m | 189924 | 189924 | 912345 | *CSR* | 2 | 3 | 1 | 2x1 | 1x2 | 26.63 | 21.95 |
| qa8fk | 66127 | 66127 | 863353 | *CSR* | 3 | 2 | 1 | 3x2,3x3 | 2x1 | 17.13 | 16.22 |
| bcsstk35 | 30237 | 30237 | 740200 | *BCSR* | 2 | 1 | 3 | 3x2,3x3 | 3x3 | 9.57 | 12.57 |
| fidapm11 | 22294 | 22294 | 623554 | *CSR* | 2 | 3 | 1 | 2x1 | 2x1 | 8.42 | 6.38 |
| msc10848 | 10848 | 10848 | 620313 | *BCSR* | 2 | 1 | 3 | 3x2,3x3 | 3x3 | 7.11 | 10.61 |
| msc23052 | 23052 | 23052 | 588933 | *BCSR* | 2 | 1 | 3 | 4x2 | 1x2 | 8.83 | 10.30 |
| bcsstk37 | 25503 | 25503 | 583240 | *BCSR* | 2 | 1 | 3 | 4x2 | 1x2 | 8.94 | 10.19 |
| bcsstk36 | 23052 | 23052 | 583096 | *BCSR* | 2 | 1 | 3 | 4x2 | 1x3 | 9.10 | 10.60 |
| cage11 | 39082 | 39082 | 559722 | *CSR* | 2 | 3 | 1 | 2x1 | 1x2 | 8.40 | 6.13 |
| e40r0100 | 17281 | 17281 | 553956 | *BCSR* | 2 | 1 | 3 | 4x2 | 1x2 | 5.39 | 5.69 |
| crystk02 | 13965 | 13965 | 491274 | *BCSR* | 2 | 1 | 3 | 3x2,3x3 | 3x3 | 5.89 | 8.57 |
| ***af23560*** | ***23560*** | ***23560*** | ***484256*** | ***BBCSR*** | ***1*** | ***2*** | ***3*** | ***4x4*** | ***2x2*** | ***4.65*** | ***4.89*** |
| ***wathen120*** | ***36441*** | ***36441*** | ***301101*** | ***BBCSR*** | ***1*** | ***2*** | ***3*** | ***3x2,3x3*** | ***2x1*** | ***6.19*** | ***6.49*** |
| ***gridgena*** | ***48962*** | ***48962*** | ***280523*** | ***BBCSR*** | ***1*** | ***2*** | ***3*** | ***2x4*** | ***2x2*** | ***6.12*** | ***6.14*** |
| ***fidap019*** | ***12005*** | ***12005*** | ***259863*** | ***BBCSR*** | ***1*** | ***2*** | ***3*** | ***4x4*** | ***2x2*** | ***2.51*** | ***2.57*** |
| ***wathen100*** | ***30401*** | ***30401*** | ***251001*** | ***BBCSR*** | ***1*** | ***2*** | ***3*** | ***2x3*** | ***2x1*** | ***5.41*** | ***5.45*** |
| ***gyro_m*** | ***17361*** | ***17361*** | ***178896*** | ***BBCSR*** | ***1*** | ***3*** | ***2*** | ***3x2,3x3*** | ***2x1*** | ***4.01*** | ***2.82*** |
| ***vibrobox*** | ***12328*** | ***12328*** | ***177578*** | ***BBCSR*** | ***1*** | ***3*** | ***2*** | ***2x1*** | ***2x1*** | ***3.08*** | ***2.76*** |

**Table 6-4 tSMVM BBCSR vs BCSR vs CSR**

As can be seen from the results in Table 6-4 no one SMVM method is optimal for all sparse matrices and there is considerable variability in the SMVM run-time depending on the tile size selected as well as whether the SMVM is computed using CSR, BCSR or BBCSR representations.

This being said the proposed BBCSR scheme is faster than both CSR and BCSR schemes in 7 out of the 50 cases, specifically the matrices af23560, wathen120, gridgena, fidap019, wathen100, gyro_m and vibrobox. The af23560 matrix is from a CFD (Computational Fluid Dynamics) problem, the wathen100 and wathen120 matrices are randomly generated 2/3D problems, vibrobox is a vibro-acoustics matrix, gyro_m is from a model reduction problem, gridgena from an optimisation problem and fidap019 from an FEM (Finite Element Method) application.

Of the remaining 43 cases from the 50 matrix set CSR is the fastest method in 10 of them specifically, cage13, t3dh_e, tbdlinux, gupta2, cage12, gupta1, turon_m, qa8fk, fidapm11 and cage11. In 6 out of these 10 cases (cage13, 18_tbdlinux, cage12, turon_m, fidapm11 and cage11) BBCSR outperforms BCSR by on average 7.3%, and in the remaining 4 cases underperforms BCSR by on average 4.2%.

In the remaining 33 cases BCSR is the fastest method and in all but 5 cases (crankseg_2, crankseg_1, smt, vanbody, fidap011 and bcsstk32) BBCSR outperforms CSR in second place by on average %.

As can be seen from Figure 6-4 BCSR is heavily influenced by BCSR tiling, and for the 50-matrix set 1x2 (row x column) or 2x1 tilings result in the best run-times in 60% of cases (30/50).

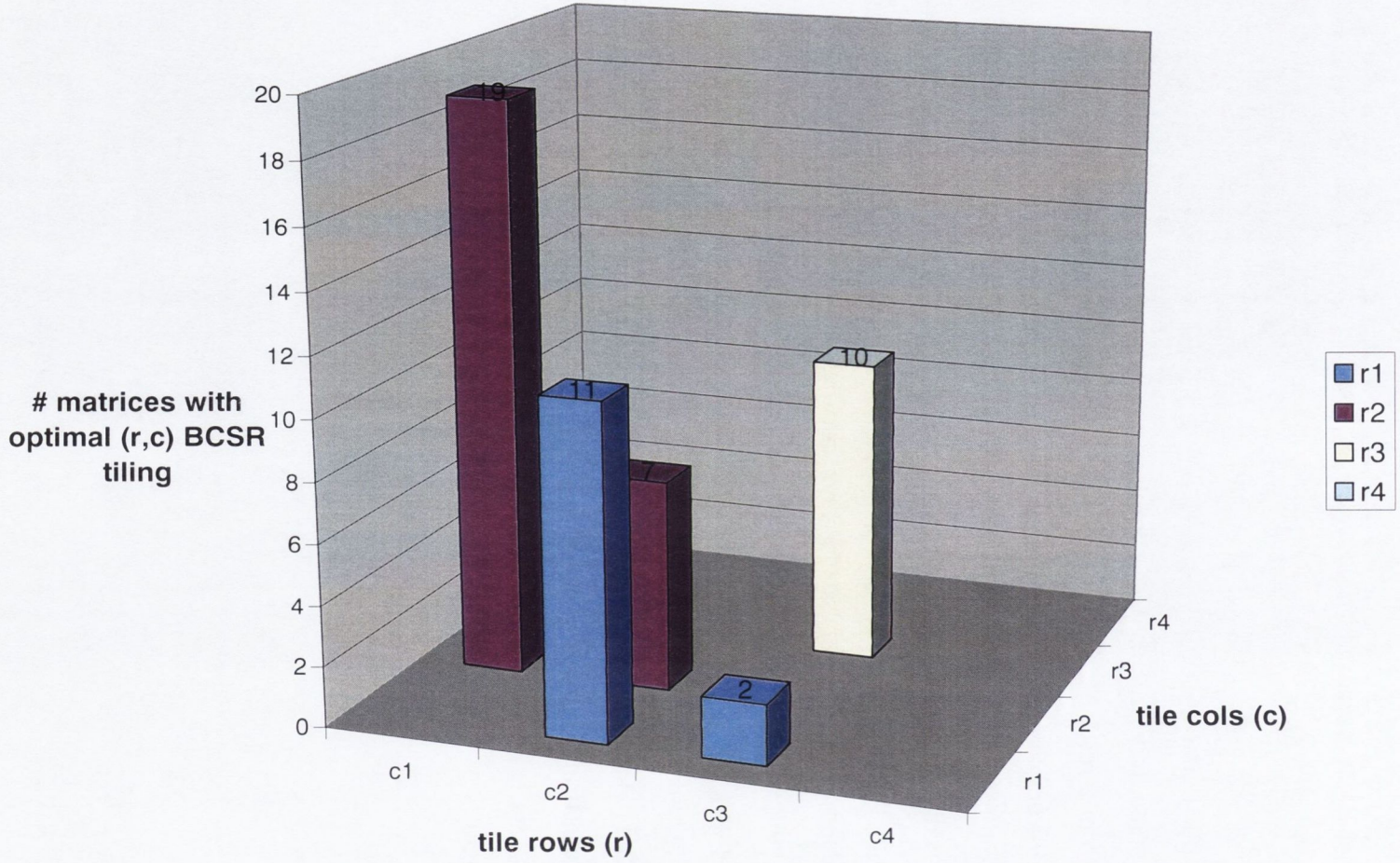



**Figure 6-4 Influence of tiling on BCSR SMVM**

In the case of BBCSR tiling still has a significant effect on SMVM execution-time, however the tile sizes are radically different as shown in Figure 6-5, in fact as can be seen from the figure in many cases there are equivalent run-times for multiple BBCSR tilings for example 3x2 and 2x3. In this respect BBCSR differs from BCSR where a single tiling produces the fastest SMVM run-time in all cases.

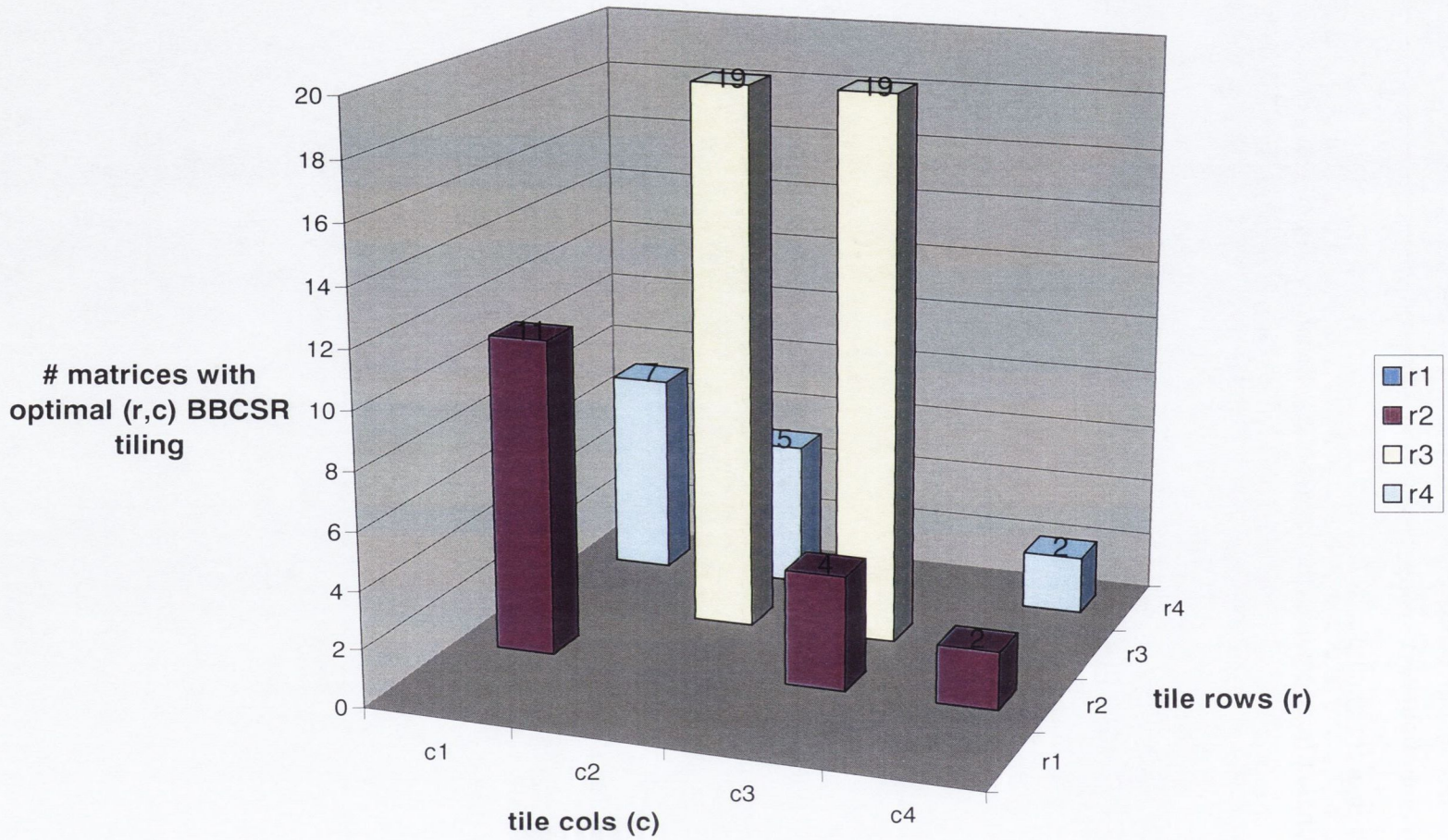



**Figure 6-5 Influence of tiling on BBCSR SMVM**

### 6.2.5 Factors Influencing BBCSR SMVM Execution

Examining the cases where the BBCSR scheme is fastest an approximate calculation of 12 Bytes per non-zero (an 8-byte double plus a 4-byte address) in terms of the size of the BBCSR data-structure would place each of the 7 matrices around the 4MByte L2 cache-size in terms of memory footprint, assuming the processor is not running other significantly-sized workloads at the same time.

Following this it was felt that further investigation might help to explain the mechanism behind the superior BBCSR performance in these cases. Fortunately the Valgrind tool [144] provides the necessary infrastructure (cachegrind) to simulate existing x86 binaries on a parameteriseable cache model containing I1, D1 and L2 caches without modification.

Each of the CSR, BCSR and BBCSR binaries were run for the 7 matrix subset using the following command-line parameters using Valgrind v3.2.1:

```
valgrind --tool=cachegrind --I1=32768,8,64 --
D1=32768,8,64 --L2=4194304,16,64 --log-file-
exactly=af23560.csr ./test_csr.exe af23560.mtx
```

The results from the valgrind cache simulations are shown in Table 6-5. As can be seen from the 3rd, 4th and 5th columns of the table the numbers of L1 instruction-cache misses are all very low and approximately equal for each of the matrices at around 2k misses. Almost all of these 2k misses also generated misses on the shared L2 cache. Bearing in mind that these misses are totals for the whole x86 binary and not just the SMVM portion of it and the programs also contain large amounts of code to read MatrixMarket [11] format matrices and convert them to CSR/BCSR/BBCSR data-structures, the numbers of instruction cache misses are exceedingly low suggesting that the 32kB L1 instruction cache is a very good match for the tight inner SMVM code loops. It can be reasonably concluded that differing code sizes and behaviour for the 3 SMVM methods does not explain the difference in SMVM execution-time.

The more likely explanation is therefore that differences in data-cache behaviour have a stronger correlation with SMVM execution time. The relationships between the various SMVM methods execution-times and cache misses are shown in Table **6-6**.

| Matrix | SMVM | I refs | I1 misses | L2i misses | D refs | D1 misses | L2D rd | L2 refs | L2 misses |
|---|---|---|---|---|---|---|---|---|---|
| *af23560* | csr | 2170740462 | 1726 | 1713 | 1012177973 | 3845023 | 870091 | 3846749 | 871804 |
| | bcsr | 28010482583 | 1803 | 1786 | 5185794505 | 281969372 | 990892 | 281971175 | 992678 |
| | bbcsr | 11915749464 | 1895 | 1881 | 3069084495 | 142836563 | 835303 | 142838458 | 837184 |
| *wathen120* | csr | 1735300057 | 1728 | 1715 | 873946727 | 4175429 | 1238335 | 4177157 | 1240050 |
| | bcsr | 88096133251 | 1811 | 1794 | 15495854531 | 669090537 | 1416946 | 669092348 | 1418740 |
| | bbcsr | 30331932249 | 1895 | 1880 | 6864102720 | 336580354 | 1230888 | 336582249 | 1232768 |
| *gridgena* | csr | 1665176525 | 1977 | 1730 | 818346838 | 3639304 | 935614 | 3641281 | 937344 |
| | bcsr | 113171441970 | 2055 | 1804 | 18809915076 | 1203244465 | 1093032 | 1203246520 | 1094836 |
| | bbcsr | 35392121249 | 1723 | 1475 | 7982189185 | 465581822 | 766145 | 465583545 | 767620 |
| *fidap019* | csr | 1154944991 | 1737 | 1711 | 524598402 | 1781335 | 335706 | 1783072 | 337417 |
| | bcsr | 7874150652 | 1806 | 1775 | 1611679623 | 74188753 | 414556 | 74190559 | 416331 |
| | bbcsr | 3704258484 | 1888 | 1865 | 1064017197 | 37968266 | 345980 | 37970154 | 347845 |
| *wathen100* | csr | 1441603423 | 1732 | 1719 | 725251385 | 3442421 | 842790 | 3444153 | 844509 |
| | bcsr | 61552896232 | 1809 | 1792 | 10903764948 | 466297433 | 991428 | 466299242 | 993220 |
| | bbcsr | 19658944340 | 1899 | 1884 | 4588618374 | 234919937 | 835799 | 234921836 | 837683 |
| *gyro_m* | csr | 1118556980 | 1752 | 1738 | 557112078 | 3085171 | 719589 | 3086923 | 721327 |
| | bcsr | 20743208258 | 1817 | 1800 | 3882885493 | 154547239 | 885846 | 154549056 | 887646 |
| | bbcsr | 7647972089 | 1893 | 1878 | 1927632363 | 78774189 | 744370 | 78776082 | 746248 |
| *vibrobox* | csr | 1062760100 | 2061 | 1735 | 556448212 | 3393039 | 1021139 | 3395100 | 1022874 |
| | bcsr | 10969571775 | 2125 | 1795 | 2237223890 | 79902247 | 1165486 | 79904372 | 1167281 |
| | bbcsr | 5808930629 | 2200 | 1873 | 1555102732 | 41669687 | 1054027 | 41671887 | 1055900 |

**Table 6-5 BBCSR fastest subset Cachegrind Simulation results**

| Matrix | SMVM | D1 misses | rank | L2 misses | ΔL2 % | rank | tSMVM | Δsmvm % | rank |
|---|---|---|---|---|---|---|---|---|---|
| *af23560* | csr | 3845023 | 1 | 871804 | 4.14% | 2 | 4894344 | 21.24% | 3 |
| | bcsr | 281969372 | 3 | 992678 | 18.57% | 3 | 4648653 | 15.15% | 2 |
| | bbcsr | 142836563 | 2 | 837184 | 0.00% | 1 | 4036905 | 0.00% | 1 |
| *wathen120* | csr | 4175429 | 1 | 1240050 | 0.59% | 2 | 6493068 | 7.12% | 3 |
| | bcsr | 669090537 | 3 | 1418740 | 15.09% | 3 | 6186879 | 2.06% | 2 |
| | bbcsr | 336580354 | 2 | 1232768 | 0.00% | 1 | 6061752 | 0.00% | 1 |
| *gridgena* | csr | 3639304 | 1 | 937344 | 22.11% | 2 | 6144336 | 3.28% | 3 |
| | bcsr | 1203244465 | 3 | 1094836 | 42.63% | 3 | 6121368 | 2.89% | 2 |
| | bbcsr | 465581822 | 2 | 767620 | 0.00% | 1 | 5949387 | 0.00% | 1 |
| *fidap019* | csr | 1781335 | 1 | 337417 | 0.00% | 1 | 2566593 | 9.28% | 3 |
| | bcsr | 74188753 | 3 | 416331 | 23.39% | 3 | 2514564 | 7.07% | 2 |
| | bbcsr | 37968266 | 2 | 347845 | 3.09% | 2 | 2348604 | 0.00% | 1 |
| *wathen100* | csr | 3442421 | 1 | 844509 | 0.81% | 2 | 5446944 | 4.76% | 3 |
| | bcsr | 466297433 | 3 | 993220 | 18.57% | 3 | 5407488 | 4.00% | 2 |
| | bbcsr | 234919937 | 2 | 837683 | 0.00% | 1 | 5199462 | 0.00% | 1 |
| *gyro_m* | csr | 3085171 | 1 | 721327 | 0.00% | 1 | 2820854 | 7.57% | 2 |
| | bcsr | 154547239 | 3 | 887646 | 23.06% | 3 | 4013139 | 53.03% | 3 |
| | bbcsr | 78774189 | 2 | 746248 | 3.45% | 2 | 2622463 | 0.00% | 1 |
| *vibrobox* | csr | 3393039 | 1 | 1022874 | 0.00% | 2 | 2764686 | 1.77% | 2 |
| | bcsr | 79902247 | 3 | 1167281 | 14.12% | 3 | 3077805 | 13.29% | 3 |
| | bbcsr | 41669687 | 2 | 1055900 | 3.23% | 1 | 2716727 | 0.00% | 1 |
| Matrix | SMVM | D1 misses | rank | L2 misses | ΔL2 % | rank | tSMVM | Δsmvm % | rank |
| *Average* | csr | | | | 3.95% | | | 7.86% | |
| | bcsr | | | | 22.20% | | | 13.93% | |
| | bbcsr | | | | 1.40% | | | 0.00% | |

**Table 6-6 tSMVM vs. Cache Misses**

As can be seen from **Table 6-6** CSR always generates the lowest number of L1 data-cache misses, followed by BBCSR and then BCSR. Typically BBCSR SMVM generates 50% or less D1 misses than BCSR for the 7 matrices in the table. The reason for this is most likely to be that BBCSR skips actual calculations based on some values based in bitwise comparisons as it was designed to do.

Looking at the number of L2 data-misses in the case of the af23580, wathen120, gridgena, wathen100 and vibrobox matrices BBCSR generates the fewest misses, followed by CSR and then BBCSR. In the case of the remaining fidap019 and gyro_m matrices CSR generates the fewest misses, followed by BBCSR and then BCSR. In all cases BCSR generates the highest number of L2 data-misses.

In terms of correlation between SMVM execution time and L2 cache the correlation is strong with the lowest number of L2 cache misses corresponding to the fastest $t_{SMVM}$ for all matrices with the exception of fidap019 and gyro_m. In these two cases CSR

actually has the lowest number of L2 cache misses but this does not correspond to the minimum SMVM execution time. This being said the L2 cache miss rate is strongly correlated to the SMVM execution time, with BBCSR minimising both in 5/7 cases.

In summary for the 7 cases where BBCSR minimises SMVM execution time it does so by an average of 7.85% compared with CSR, and 13.93% compared with BCSR. For the same 7 matrices the deviation of the L2 cache miss rate for BBCSR deviates by 1.4% from the minimum, CSR by 3.95% and BCSR by 22.2% from the minimum L2 miss rate for a particular matrix from the 7 matrix subset.

Some additional light is shed on the reason for poor BCSR performance by the data shown in Table 6-7. The table shows the minimum zero fill for BCSR tiled matrices across the range 1x2 to 4x4. It can be seen that for the relevant examples the fill rate is 50% on average meaning that large numbers of trivial data are being fetched needlessly, leading to lower than expected SMVM execution times. The corresponding BBCSR fill rates are very low as a single bit in a 32-bit integer is effectively used to code for a 64-bit, double-precision zero fill data-value.

It is probable that a method of determining if BBCSR storage will be advantageous during the sampling of matrix data in the tiling process used in packages such as OSKI [111] The performance overhead of such a modification would be negligible as the same arithmetic operations have to be performed for both tiling methods and only the final decision making to model the expected performance of each method and choose the best performer need be modified.

| filename | # | tilings | min% |
|---|---|---|---|
| af_shell8 | 1 | 1x2 | 9% |
| cage13 | 2 | 1x2,2x1 | 90% |
| nd12k | 2 | 1x3,3x1 | 12% |
| crankseg_2 | 2 | 1x2,2x1 | 13% |
| pwtk | 1 | 1x3 | 12% |
| hood | 2 | 1x2,2x1 | 13% |
| bmwcra_1 | 3 | 1x3,3x1,3x3 | 0% |
| crankseg_1 | 2 | 1x2,2x1 | 13% |
| SHIPSEC5 | 2 | 1x2,2x1 | 53% |
| M_T1 | 2 | 1x3 | 0% |
| SHIP_003 | 2 | 1x2,2x1 | 58% |
| SHIPSEC1 | 2 | 1x2,2x1 | 51% |
| bmw7st_1 | 1 | 1x2 | 11% |
| nd6k | 2 | 1x2,2x1 | 12% |
| SHIPSEC8 | 2 | 1x2,2x1 | 46% |
| s3dkq4m2 | 1 | 1x2 | 13% |
| THREAD | 2 | 1x3 | 1% |
| t3dh_e | 1 | 1x2 | 48% |
| 18_tbdlinux | 1 | 1x2 | 74% |
| gupta2 | 2 | 1x2,2x1 | 55% |
| cage12 | 2 | 1x2,2x1 | 88% |
| s3dkt3m2 | 1 | 1x2 | 9% |
| smt | 2 | 1x2,2x1 | 16% |
| oilpan | 2 | 1x2,2x1 | 32% |
| nd3k | 2 | 1x2,2x1 | 11% |
| 3dtube | 2 | 1x3 | 1% |
| ct20stif | 1 | 1x2 | 11% |
| raefsky4 | 2 | 1x2,2x1 | 13% |
| vanbody | 2 | 1x2,2x1 | 12% |
| gupta1 | 2 | 1x2,2x1 | 56% |
| fidap011 | 2 | 1x2,2x1 | 15% |
| bcsstk32 | 1 | 1x2 | 14% |
| turon_m | 2 | 1x2,2x1 | 67% |
| qa8fk | 1 | 1x2 | 34% |
| bcsstk35 | 1 | 1x2 | 3% |
| fidapm11 | 2 | 1x2,2x1 | 78% |
| msc10848 | 3 | 1x3 | 0% |
| msc23052 | 2 | 1x2,2x1 | 13% |
| bcsstk37 | 1 | 1x2 | 11% |
| bcsstk36 | 2 | 1x2,2x1 | 8% |
| cage11 | 2 | 1x2,2x1 | 78% |
| e40r0100 | 2 | 1x2,2x1 | 20% |
| crystk02 | 3 | 1x3 | 0% |
| **af23560** | **1** | **1x2,2x1** | **30%** |
| **wathen120** | **1** | **1x2** | **28%** |
| **gridgena** | **3** | **1x2,2x1** | **33%** |
| **fidap019** | **2** | **1x2,2x1** | **36%** |
| **wathen100** | **1** | **1x2** | **28%** |
| **gyro_m** | **1** | **1x2** | **100%** |
| **vibrobox** | **2** | **1x2,2x1** | **89%** |

**Table 6-7 Minimum BCSR Minimum Fill% vs. Tiling**

## 6.3 BBCSR Optimisation

Another factor influencing the execution-time of BBCSR SMVM codes is the distribution of non-zeroes in a tile. Specifically, on examination of lines L17-32 of Listing 6-5 it can be seen that each line of the SMVM product will execute subject to the bitmap being tested returning a one, i.e. the bitmap bit was set to one denoting a non-zero entry is present at that point in the bitmap. As can be seen all bitmap bits are tested regardless of the number of zeroes meaning that there is a fixed overhead.

The performance of the SMVM code could be improved by testing the number of non-zeroes tested (_nz) against the known number of non-zero bit in the bitmap leading to an early exit from the SMVM product code. The disadvantage of this technique however is that it may not produce a speed-up as it adds additional comparisons to the code and in any case the tile might contain a non-zero in the lower right-hand side of the tile, meaning that all of the tests for bitmap bits and non-zero count are incurred for all 16 elements of the 4x4 tile.

Another possible optimisation would be to reorder the sequence of code for the bitmap comparisons so that the test for non-zero count exits as early as possible. Indeed given that the bitmap itself holds the key to which comparisons are required it is possible to eliminate the comparisons altogether by generating a tile SMVM code tailored to each individual bitmap. The disadvantage of this approach is however that a function-call (the bitmap itself would be used to look-up a pointer to a specialised SMVM function from a table) is required for each tile SMVM which adds overhead and secondly in the case of a 4x4 tile size there are $2^{16}$ possible bitmap values, and hence up to 65k SMVM codes which would have to be written or generated automatically and then verified for correctness.

This approach was felt to be possible but impractical and unlikely to lead to a systematic advantage in terms of BBCSR so a new approach was adopted which eliminated comparisons and terminated early but without the need for per-bitmap SMVM codes and function-call tables.

## 6.4 Scheduled Bitmap SMVM

The disadvantage with the BBCSR code is that depending on the pattern of non-zeroes in the r*c sized tile the execution time can vary widely. If all of the non-zeros could be grouped together by sorting the execution time would be minimised but this

would require the writing of $2^{16}$ different orderings of the SMVM code for the 4x4 tile-size alone leading to an explosion in the amount of code to be written or generated and subsequently verified for correctness. On top of this the contents of each tile would have to be sorted leading to a significant overhead in setting up the sparse data-structures. Furthermore an additional overhead would be incurred in that the bitmap would have to be used as a pointer to a table of functions containing the correct version of the 4x4 SMVM to be run for a particular non-zero reordering. This lack of locality would lead to reduced instruction-cache performance which would most likely outweigh the benefits of any increase in SMVM speed.

A better alternative would be to vary the schedule through the 16 lines of the matrix-vector product depending on the bitmap. This would involve first generating a schedule from the bitmap and then executing that schedule. Normally this would require assembly language coding of the entire SMVM code as ANSI C/C++ does not support jumps (goto) to labels. However such a feature was recently introduced in gcc [145] enabling a jump-table with variable schedule order to be implemented in C-code. This approach is applied to efficient interpreters by Ertl and Gregg in [146].

### 6.4.1 Scheduled Block Compressed Sparse Row Format (SBCSR)

The SBCSR data-structure consists of 4 arrays as shown in Figure 6-2. The SBCSR structure augments the 3 BCSR arrays (row_start, col_idx and value) with schedule array containing the schedule of non-zero operations to be performed. As in the BBCSR case the value array contains only non-zero entries without any zero-fill. Schedules for matrices are constructed as follows:

- The matrix is tiled into four 2x3 sub-matrices
- The sub-matrices are processed left to right across the rows in successive rows
- For each row the data-structure row_start array is populated with a row entry if the row contains at least one sub-matrix with non-zero entries

For each 2x3 sub-matrix

- The col_idx array is populated with the column coordinate of the upper left-hand side of the sub-matrix
- The value array is filled with the non-zero values, scanning the sub-matrix row by row and each column within a row from left to right
- For each non-zero in the value array the element number of the non-zero is inserted into the schedule, allowing non-zeroes to be skipped by the scheduler

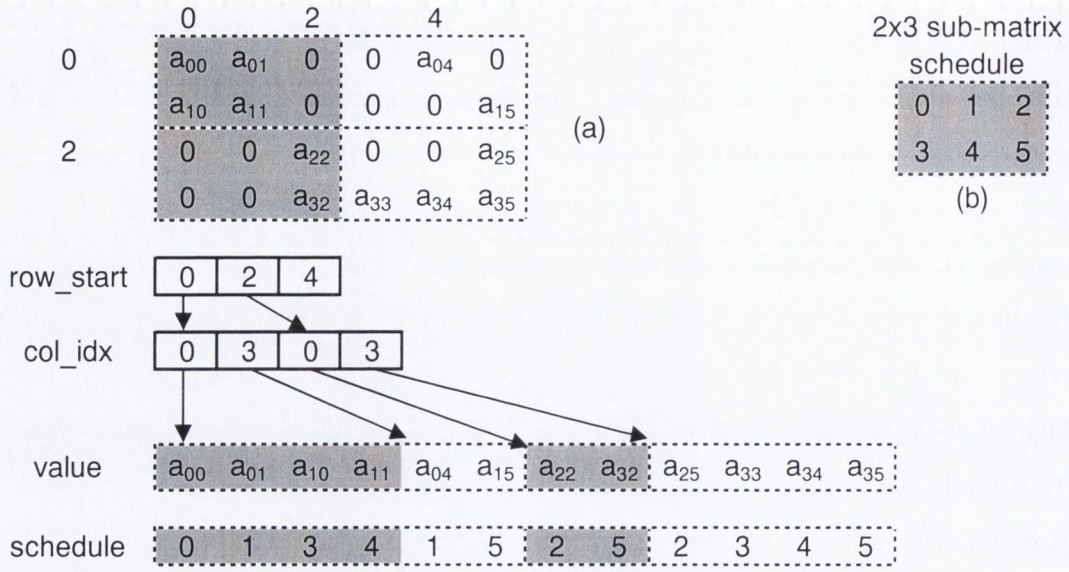



**Figure 6-6 2x3 SBCSR Sparse Matrix Storage Format**

In the specific example shown in **Figure 6-6 (a)** the row_start array contains both row indices 0 and 2 as both rows contain non-zero sub-matrices. The col_idx array contains the four starting column references [0,3,0,3] for each of the 4 sub-matrices. The value array contains the $a_{00}$, $a_{01}$, $a_{10}$ and $a_{11}$ non-zero entries from the first 2x3 matrix;, $a_{04}$ and $a_{15}$ from the second; $a_{22}$ and $a_{32}$ from the third; and finally $a_{25}$, $a_{33}$, $a_{34}$ and $a_{35}$ from the fourth sub-matrix. Correspondingly the schedule array contains $0^{th}$, $1^{st}$, $3^{rd}$ and $4^{th}$ references corresponding to the non-zero positions in the first sub-matrix and so on as shown in **Figure 6-6 (b)**.

The code in Listing 6-7 shows the SBCSR SMVM for a 4x4 tile size. In order to implement the desired functionality in the SBCSR format each line of the SMVM product is labelled (as shown in L21), each preceded by a jump (goto) a label (as shown in L20). As the schedules are relatively short 8-bit chars can be used for the schedule array, thus improving storage efficiency. A final label to terminate the jump sequence is included on L52 and a complete jump-table containing the addresses of all 17 labels is constructed by the compiler as shown in L4 of Listing 6-7.

```
L1.     void sbcsr_smvm4x4 (int bm, int r, int c,
  unsigned char *schedule, int *row_start, int
  *col_idx, int *bitmap_idx, Type *value, Type *src,
  Type *dest) {
L2.       int i, j, _nz, bitmap, nz;
L3.       Type y0, y1, y2, y3, x0, x1, x2, x3;
```

```
L4.       static void *jt[] = {&&j0,&&j1,&&j2,&&j3,&&j4,
   &&j5,&&j6,&&j7,&&j8,&&j9&&j10,&&j11,&&j12,&&j13,
   &&j14,&&j15,&&terminate};

L5.       for (i=0; i<bm; i++, dest+=r)
L6.         y0 = dest[0];
L7.         y1 = dest[1];
L8.         y2 = dest[2];
L9.         y3 = dest[3];
L10.        for (j=row_start[i]; j<row_start[i+1]; j++,
   bitmap_idx++, col_idx++, value+=nz) {
L11.          bitmap = *bitmap_idx     & 0x0000FFFF;
L12.          nz     = *bitmap_idx>>16 & 0x0000FFFF
L13.          _nz    = 0;
L14.          x0 = src[(*col_idx)    ];
L15.          x1 = src[(*col_idx) + 1];
L16.          x2 = src[(*col_idx) + 2];
L17.          x3 = src[(*col_idx) + 3];
L18.          goto *jt[*schedule++];
L19.          j0:  y0+= value[_nz++] * x0;
L20.          goto *jt[*schedule++];
L21.          j1:  y0+= value[_nz++] * x1;
L22.          goto *jt[*schedule++];
L23.          j2:  y0+= value[_nz++] * x2;
L24.          goto *jt[*schedule++];
L25.          j3:  y0+= value[_nz++] * x3;
L26.          goto *jt[*schedule++];
L27.          j4:  y1+= value[_nz++] * x0;
L28.          goto *jt[*schedule++];
L29.          j5:  y1+= value[_nz++] * x1;
L30.          goto *jt[*schedule++];
L31.          j6:  y1+= value[_nz++] * x2;
L32.          goto *jt[*schedule++];
L33.          j7:  y1+= value[_nz++] * x3;
L34.          goto *jt[*schedule++];
L35.          j8:  y2+= value[_nz++] * x0;
L36.          goto *jt[*schedule++];
L37.          j9:  y2+= value[_nz++] * x1;
L38.          goto *jt[*schedule++];
L39.          j10: y2+= value[_nz++] * x2;
L40.          goto *jt[*schedule++];
L41.          j11: y2+= value[_nz++] * x3;
L42.          goto *jt[*schedule++];
L43.          j12: y3+= value[_nz++] * x0;
L44.          goto *jt[*schedule++];
L45.          j13: y3+= value[_nz++] * x1;
L46.          goto *jt[*schedule++];
L47.          j14: y3+= value[_nz++] * x2;
L48.          goto *jt[*schedule++];
L49.          j15: y3+= value[_nz++] * x3;
L50.          terminate:;
```

```
L51.          }
L52.          dest[0] = y0;
L53.          dest[1] = y1;
L54.          dest[2] = y2;
L55.          dest[3] = y3;
L56.        }
L57.    } // sbcsr_smvm4x4()
```

**Listing 6-7 SBCSR 4x4 SMVM C-Code**

The x86 assembler corresponding to two lines of C-code from the SBCSR SMVM for a 4x4 tile size:

```
goto *jt[*schedule++];
j0:  y0 += value[_nz++] * x0;
```

is shown in Listing 6-8:

```
176:test_jtsmvm.cpp ****       goto *jt[*schedule++]; j0:
y0 += value[_nz++] * x0; // row 0
1462                  .stabn 68,0,176,LM114-
__Z16bbcsr_smvm4x4_jtiiiPhPiS0_S0_PdS1_S1_
1463                  LM114:
1464 09b0 8B4514          movl 20(%ebp), %eax
1465 09b3 0FB610          movzbl    (%eax), %edx
1466 09b6 8D4514          leal 20(%ebp), %eax
1467 09b9 FF00            incl (%eax)
1468 09bb 8B149500        movl
     _ZZ16bbcsr_smvm4x4_jtiiiPhPiS0_S0_PdS1_S1_E2jt(,%edx
,4), %edx
1468      000000
1469 09c2 8955A4          movl %edx, -92(%ebp)
1470                  L76:
1471 09c5 FF65A4          jmp  *-92(%ebp)
1472                  L53:
1473 09c8 8B45F4          movl -12(%ebp), %eax
1474 09cb 8D14C500        leal 0(,%eax,8), %edx
1474      000000
1475 09d2 8B4524          movl 36(%ebp), %eax
1476 09d5 DD0402          fldl (%edx,%eax)
1477 09d8 DC4DC0          fmull     -64(%ebp)
1478 09db DD45E0          fldl -32(%ebp)
1479 09de DEC1            faddp     %st, %st(1)
1480 09e0 8D45F4          leal -12(%ebp), %eax
1481 09e3 FF00            incl (%eax)
1482 09e5 DD5DE0          fstpl     -32(%ebp)
```

**Listing 6-8 x86 Assembler for SBCSR SMVM 4x4**

### 6.4.2 SBCSR Schedule Generation

As can be seen from the previous section L18-L50 will be executed differently on a tile-by-tile basis, depending on how the schedule for the tile has been generated. Generating a schedule for a tile consists of 2 steps:

- schedule pruning
- schedule optimisation (optional)

In the schedule pruning phase non-zeros leading to trivial operations are pruned from a full tile schedule in order to create a pruned schedule which computes only those products corresponding to tile non-zeroes. In the optional schedule optimisation phase pruned schedule is reordered to deal with data-dependencies such as RAW and maximise performance. It should be noted that whereas schedule pruning has no collateral effects, schedule optimisation leads to additional overhead in terms of reordering the tile non-zero values to reflect the optimised schedule.

A schedule can be generated directly by examining the tile non-zero distribution or alternately the bitmap as shown in Listing 6-9.

```
L1.     void sbcsr_smvm4x4 (int bm, int r, int c,
        unsigned char *schedule, int *row_start, int
        *col_idx, int *bitmap_idx, Type *value, Type *src,
        Type *dest) {
L2.     void bm2sch(unsigned int bitmap, unsigned int
        *sc, int r, int c) {
L3.       unsigned nz=0, *p, mask;
L4.       if (bitmap & 32768) sc[nz++] = 15; // row 0
L5.       if (bitmap & 16384) sc[nz++] = 14;
L6.       if (bitmap &  8192) sc[nz++] = 13;
L7.       if (bitmap &  4096) sc[nz++] = 12;
L8.       if (bitmap &  2048) sc[nz++] = 11;  // row 1
L9.       if (bitmap &  1024) sc[nz++] = 10;
L10.      if (bitmap &   512) sc[nz++] =  9;
L11.      if (bitmap &   256) sc[nz++] =  8;
L12.      if (bitmap &   128) sc[nz++] =  7;  // row 2
L13.      if (bitmap &    64) sc[nz++] =  6;
L14.      if (bitmap &    32) sc[nz++] =  5;
L15.      if (bitmap &    16) sc[nz++] =  4;
L16.      if (bitmap &     8) sc[nz++] =  3;  // row 3
L17.      if (bitmap &     4) sc[nz++] =  2;
L18.      if (bitmap &     2) sc[nz++] =  1;
L19.      if (bitmap &     1) sc[nz++] =  0;
L20.      sc[nz++] = r*c; // terminate
L21.    } // bm2sch()
```

**Listing 6-9 x86 Bitmap Schedule Generator**

The code shown in the scheduler only implements the first phase of the 2 phases outlined earlier and the correspondence between the schedule and the non-zero distribution for a sample 4x4 tile is shown in Figure 6-7.

| | c0 | c1 | c2 | c3 |
|---|---|---|---|---|
| r0 | 1.0 | 1.0 | 1.0 | 1.0 |
| r1 | | 1.0 | 1.0 | 1.0 |
| r2 | | | 1.0 | 1.0 |
| r3 | | | | 1.0 |

| schedule | 0 | 1 | 2 | 3 | 5 | 6 | 7 | 10 | 11 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| executed | 19 | 21 | 23 | 25 | 29 | 31 | 33 | 39 | 41 | 49 | |
| skipped | | | | | 27 | | | 35 | | 43 | |
| | | | | | | | | 37 | | 45 | |
| | | | | | | | | | | 47 | |

**Figure 6-7 Bitmap Schedule Generation Example**

As can be seen the generated schedule simply skips the non-zero elements of the product and thus would skip L27, L35, L37, L43, L45 and L47 of the SBCSR SMVM code in Listing 6-7.

### 6.4.3 SBCSR Schedule Optimisation

The optimisation of the SBCSR schedule depends heavily on the underlying processor data-path. As was pointed out in chapter 4 a key property of a typical processor datapath which leads to performance degradation for the execution of SMVM codes is the RAW hazard associated with the floating-point adder which sums partial-products from a matrix-row.

The standard method for implementing Out-Of-Order (OOO) processing is Tomasulo's algorithm [147]. Examples of more modern implementations of Tomasulo-like algorithms for hardware implementation in microprocessors are those by Sassone [148] and Farrell [149]. In an OOO processor hardware will eliminate, or at least mitigate these hazards by interleaving calculations from multiple rows in order to ensure the floating-point adder does not stall as processing continues along multiple rows in parallel. The limitation of course is the length of the reorder buffer that the processor can examine with a view to reordering dependencies. As mentioned previously OOO processors have fallen out of favour due to the complexity of the

resulting hardware and diminishing returns in terms of speedup using the OOO hardware, especially given the move to multicore processors has led to a drive for more, simpler processor cores to hit performance requirements for new products.

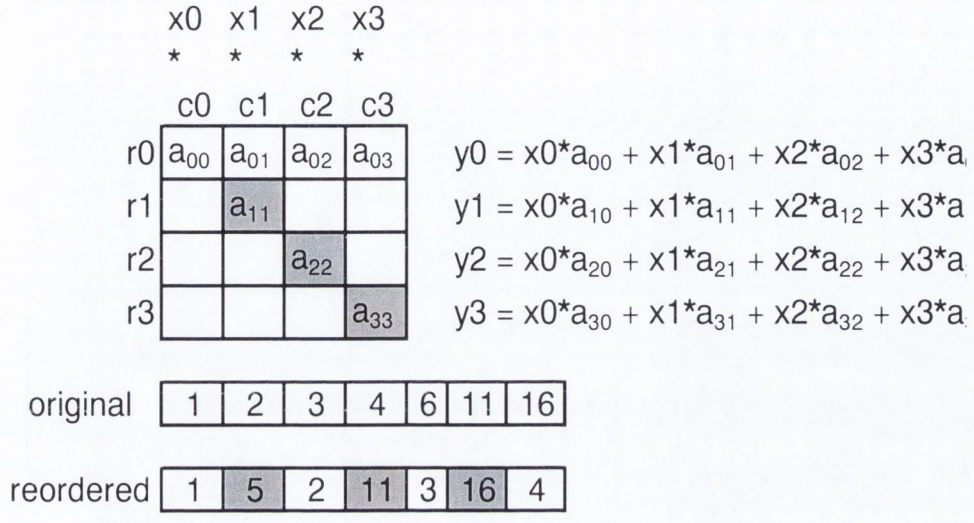



**Figure 6-8 SBCSR Column Reordering**

As the scheduling for the SBCSR method can be performed statically as part of the matrix format conversion process there is no dependency on the underlying hardware and hence any scheduling algorithm can be used. The simplest approach to minimising RAW hazards is to reorder the SBCSR tiles into column order in a manner similar to that used in SPAR [23]. An example of an SBCSR tile, original and reordered schedule are shown in **Figure 6-8**. As can be seen the column reordering inserts products from other columns to minimise the effects of RAW hazards by removing dependencies linked to the calculation of y0.

In practice the precise scheduling algorithm to be used will depend on the latencies of the floating-point adder as well as how many parallel adders the processor contains. The code to perform column-oriented reordering and non-zero pruning is shown in **Listing 6-10**, this reordering is preceded by reading-out the tile non-zeroes column-wise into the value array. Note: re-orderings of arbitrary sophistication are possible.

```
L1.     void bm2sch_c(unsigned int bitmap, unsigned int
  *sc, int r, int c) {
L2.       unsigned nz=0, unsigned int *p, mask;
L3.       if (bitmap & 32768) sc[nz++] = 15; // row 0
L4.       if (bitmap &  2048) sc[nz++] = 11; // row 1
L5.       if (bitmap &   128) sc[nz++] =  7; // row 2
```

```
L6.       if (bitmap &     8) sc[nz++] =  3; // row 3
L7.       if (bitmap & 16384) sc[nz++] = 14; // row 0
L8.       if (bitmap &  1024) sc[nz++] = 10; // row 1
L9.       if (bitmap &    64) sc[nz++] =  6; // row 2
L10.      if (bitmap &     4) sc[nz++] =  2; // row 3
L11.      if (bitmap &  8192) sc[nz++] = 13; // row 0
L12.      if (bitmap &   512) sc[nz++] =  9; // row 1
L13.      if (bitmap &    32) sc[nz++] =  5; // row 2
L14.      if (bitmap &     2) sc[nz++] =  1; // row 3
L15.      if (bitmap &  4096) sc[nz++] = 12; // row 0
L16.      if (bitmap &   256) sc[nz++] =  8; // row 1
L17.      if (bitmap &    16) sc[nz++] =  4; // row 2
L18.      if (bitmap &     1) sc[nz++] =  0; // row 3
L19.                         sc[nz++] =r*c; // end
L20.    } // bm2sch_c()
```

**Listing 6-10 Column Oriented Pruner & Scheduler**

### 6.4.4 SBCSR SMVM Performance

The performance of SBCSR relative to the other techniques is shown in Table 6-9. As can be seen from the table SBCSR only offers an advantage in the case of 2 of the matrices out of the 50 matrix set. Specifically in the case of the vibrobox and gyro_m matrices SBCSR is 57.5% and 60.6% faster than its nearest rival BBCSR.

This is most likely due to the nature of the sparsity pattern in the non-zero blocks which in the case of BBCSR consumes many cycles scanning the bitmap bit-by-bit, whereas the schedule in SBCSR allows the multiplication to exit early by jumping to the terminate: label on L50 of the listing, once all of the essential multiplications (non-zeroes) have been completed. In the other 48 cases SBCSR is significantly worse than all of the other methods owing to the large overhead in terms of instructions of implementing the proposed scheme.

As can be seen the column oriented tile reordering makes for a significant improvement in the case of the matrix set selected adding 6 additional matrices to the vibrobox and gyro_m matrices for which the SBCSR format was fastest of all the SMVM methods evaluated. The gains in performance are modest ranging from 3.84 to 16.23%, but are nevertheless significant. It is possible to order the SBCSR code first by rows and then by columns or the converse. This may improve the schedule execution in some cases by removing or minimising data-dependencies. The SBCSR_C variant of the format orders by columns first followed by rows, depending on the bitmap non-zero entries. The execution times for the SBCSR_C variant are shown in Table 6-8. As can be seen the column oriented tile reordering makes for a

significant improvement in the case of the matrix set selected adding 6 additional matrices to the vibrobox and gyro_m matrices for which the SBCSR format was fastest of all the SMVM methods evaluated. The gains in performance are modest ranging from 3.84 to 16.23%, but are nevertheless significant.

| matrix | SBCSRC % min | tSBCSRC | tBBCSR | tBCSR | tCSR |
|---|---|---|---|---|---|
| *vibrobox* | *-56.93%* | *1169991* | *2716727* | *3077805* | *2764686* |
| *gyro_m* | *-60.63%* | *1032544* | *2622463* | *4013139* | *2820854* |
| *wathen100* | *-16.23%* | *4355811* | *5199462* | *5407488* | *5446944* |
| cage11 | *-11.69%* | *5411304* | 7992450 | 8403597 | 6127776 |
| turon_m | *-6.55%* | *20510775* | 25533711 | 26630127 | 21948984 |
| 3dtube | *-3.84%* | *18144360* | 20071683 | 18869562 | 36070227 |
| bcsstk32 | *-4.15%* | *15465231* | 18762489 | 16134930 | 17797104 |
| vanbody | *-16.03%* | *15380298* | 21025314 | 18316431 | 21004245 |

**Table 6-8 SBCSR_C SMVM Execution Times**

As can be seen the column oriented tile reordering makes for a significant improvement in the case of the matrix set selected adding 6 additional matrices to the vibrobox and gyro_m matrices for which the SBCSR format was fastest of all the SMVM methods evaluated. The gains in performance are modest ranging from 3.84 to 16.23%, but are nevertheless significant.

| name | #1 | SBCSR %min | tSMVM (us) | | | |
|---|---|---|---|---|---|---|
| | | | tSBCSR | tBBCSR | tBCSR | tCSR |
| ***vibrobox*** | ***SBCSR*** | ***-57.54%*** | ***1153521*** | ***2716727*** | ***3077805*** | ***2764686*** |
| ***gyro_m*** | ***SBCSR*** | ***-60.57%*** | ***1033983*** | ***2622463*** | ***4013139*** | ***2820854*** |
| *wathen100* | *BBCSR* | *58.72%* | *8252451* | *5199462* | *5407488* | *5446944* |
| *fidap019* | *BBCSR* | *88.72%* | *4432311* | *2348604* | *2514564* | *2566593* |
| *gridgena* | *BBCSR* | *70.37%* | *10135782* | *5949387* | *6121368* | *6144336* |
| *wathen120* | *BBCSR* | *60.44%* | *9725445* | *6061752* | *6186879* | *6493068* |
| *af23560* | *BBCSR* | *33.38%* | *5384610* | *4036905* | *4648653* | *4894344* |
| crystk02 | *BCSR* | *72.31%* | *10155969* | 6530301 | 5893974 | 8567973 |
| e40r0100 | *BCSR* | *75.95%* | *9485541* | 5478273 | 5390892 | 5694435 |
| cage11 | *CSR* | *259.74%* | *22043817* | 7992450 | 8403597 | 6127776 |
| bcsstk36 | *BCSR* | *55.90%* | *14190093* | 9950949 | 9102132 | 10595205 |
| bcsstk37 | *BCSR* | *71.08%* | *15302556* | 9894474 | 8944515 | 10191267 |
| msc23052 | *BCSR* | *94.43%* | *17177535* | 10120770 | 8834742 | 10298376 |
| msc10848 | *BCSR* | *84.75%* | *13131270* | 7906545 | 7107579 | 10608309 |
| fidapm11 | *CSR* | *223.09%* | *20623824* | 7831233 | 8421228 | 6383268 |
| bcsstk35 | *BCSR* | *61.88%* | *15500259* | 10377585 | 9574920 | 12571740 |
| qa8fk | *CSR* | *125.20%* | *36524232* | 17419968 | 17133606 | 16218360 |
| turon_m | *CSR* | *220.80%* | *70412022* | 25533711 | 26630127 | 21948984 |
| bcsstk32 | *BCSR* | *80.35%* | *29099016* | 18762489 | 16134930 | 17797104 |
| fidap011 | *BCSR* | *71.88%* | *15691698* | 10481778 | 9129447 | 10446453 |
| gupta1 | *CSR* | *329.72%* | *79598124* | 21733227 | 21227724 | 18523107 |
| vanbody | *BCSR* | *86.53%* | *34164918* | 21025314 | 18316431 | 21004245 |
| raefsky4 | *BCSR* | *74.20%* | *18444420* | 11086551 | 10587942 | 12087468 |
| ct20stif | *BCSR* | *94.69%* | *39089808* | 24283386 | 20078325 | 24369651 |
| 3dtube | *BCSR* | *86.78%* | *35244630* | 20071683 | 18869562 | 36070227 |
| nd3k | *BCSR* | *86.50%* | *37123848* | 25648092 | 19905615 | 26500023 |
| oilpan | *BCSR* | *78.50%* | *36806445* | 20718324 | 20619486 | 31272381 |
| smt | *BCSR* | *115.53%* | *62678610* | 34052247 | 29081187 | 32514264 |
| s3dkt3m2 | *BCSR* | *71.89%* | *48902157* | 28907757 | 28448892 | 33137343 |
| cage12 | *CSR* | *238.84%* | *79789050* | 26838306 | 29675160 | 23548014 |
| gupta2 | *CSR* | *302.43%* | *151680645* | 44443008 | 43083603 | 37690929 |
| 18_tbdlinux | *CSR* | *556.21%* | *155477232* | 27062721 | 28142433 | 23693229 |
| t3dh_e | *CSR* | *280.86%* | *151635015* | 48766536 | 44502642 | 39813813 |
| THREAD | *BCSR* | *80.87%* | *35133183* | 21132468 | 19424835 | 52106409 |
| s3dkq4m2 | *BCSR* | *68.15%* | *57249162* | 34648911 | 34045659 | 41870007 |
| SHIPSEC8 | *BCSR* | *136.98%* | *65315583* | 29733813 | 27561240 | 79225065 |
| nd6k | *BCSR* | *87.23%* | *79460154* | 55056816 | 42440310 | 57644208 |
| bmw7st_1 | *BCSR* | *78.38%* | *100219338* | 63143208 | 56183454 | 65568159 |
| SHIPSEC1 | *BCSR* | *109.18%* | *64431495* | 32684904 | 30802077 | 93720321 |
| SHIP_003 | *BCSR* | *109.36%* | *67113801* | 33443694 | 32056434 | 96271227 |
| M_T1 | *BCSR* | *76.86%* | *77642190* | 47295747 | 43899444 | 115887132 |
| SHIPSEC5 | *BCSR* | *180.34%* | *138183939* | 52999650 | 49291101 | 85037760 |
| crankseg_1 | *BCSR* | *122.70%* | *159994710* | 88609779 | 71844768 | 86701950 |
| bmwcra_1 | *BCSR* | *78.66%* | *113444352* | 74747205 | 63497493 | 92765313 |
| hood | *BCSR* | *79.20%* | *146667483* | 92423358 | 81847575 | 100973970 |
| pwtk | *BCSR* | *68.36%* | *136682784* | 82593486 | 81182988 | 99184896 |
| crankseg_2 | *BCSR* | *116.80%* | *211270122* | 117895122 | 97449201 | 116827506 |
| nd12k | *BCSR* | *89.92%* | *165674853* | 115644636 | 87233697 | 117935532 |
| cage13 | *CSR* | *250.33%* | *318039381* | 100447812 | 112801320 | 90784071 |
| af_shell8 | *BCSR* | *70.33%* | *224840646* | 140009796 | 132003729 | 156470292 |

**Table 6-9 SBCSR SMVM Relative Performance**

## 6.5 Summary

In this section two completely new methods of blocked sparse matrix storage were proposed to address the issue of zero-fill which occurs frequently when blocking is applied to sparse-matrices with underlying structure in application areas such as Finite Element Method and Computational Fluid Dynamics commonly used to solve mechanical and aeronautical engineering problems.

A representative matrix suite with large matrices which cannot reside entirely in the internal caches of the processor was chosen and the performance of the methods was evaluated on a recent commercial engineering workstation containing an Intel CoreDuo processor. All 16 tile sizes from 1x1 to 4x4 were evaluated exhaustively for each sparse matrix and the SMVM execution-time measured using the cycle counter from TTFW. No attempt was made to use tuning to speed up the tile selection process along the lines proposed by Vuduc and used in OSKI so the results are independent of any bias that might have been introduced by the tuning method.

BBCSR was shown to offer the fastest SMVM execution time it in 7 out of 50 cases when compared against CSR and BCSR and does so by an average of 7.85% compared with CSR, and 13.93% compared with BCSR. For the same 7 matrices the deviation of the L2 cache miss rate for BBCSR deviates by 1.4% from the minimum, CSR by 3.95% and BCSR by 22.2% from the minimum L2 miss rate for a particular matrix from the 7 matrix subset. The relationship between the amount of zero fill and higher performance by BBCSR was also clearly highlighted, as were the reasons for relatively poorer BBCSR performance in the remaining 43 cases.

It was also shown that BBCSR SMVM code can be specialised on a per-bitmap basis, eliminating all bitmap comparison overhead, however this is at the cost of a function-call per non-zero tile and considerable effort in creating and generating the 65k possible SMVM codes to cover all permutations of a 4x4 tile bitmap.

Based on the shortcomings of the BBCSR method an alternate possibility based on bitmap scheduling was identified and the Scheduled Block Compressed Sparse Row storage scheme and relative SMVM was proposed. The code to generate schedules for the SBCSR SMVM was also discussed as was the possibility of combining pruning of zero-dependent partial-products and reordering those products so as to minimise dependencies and hence increase throughput by eliminating stalls due to

RAW-hazards. Considerable further work could be done on tuning SBCSR schedules for each bitmap pattern for all target processor architectures.

The implementation in the SBCSR SMVM C code was enabled by the use of a non-standard feature of the gcc compiler. Code analysis showed that the generated code contained many additional assembly language instructions with respect to BBCSR; however despite low expectations the SBCSR method actually outperformed BBCSR, BCSR and CSR in the cases of the vibrobox and gyro_m matrices. This was counterbalanced by significantly poorer performance for SBCSR for the remainder of the matrix suite. An interesting area for further work would be to combine BBCSR and SBCSR together with BCSR formats into a single data-structure with a single hybrid software SMVM method. In the proposed method and data-structure individual tiles would be stored as BBCSR or SBCSR tiles where BCSR fill is high and as native BCSR where the fill is low. The author believes such a hybrid format could offer optimal performance for a wide range of matrices and local fill patterns could be tuned for on a tile-by-tile basis rather than averaged basis used by Vuduc.

The main reason for the poor performance of SBCSR in all but 2 out of 50 cases is that the x86 pipeline and hardware are ill-suited to the kind of code used in SBCSR leading to poor performance. It was also shown that reordering tiles in column-order as part of the scheduling process improves the SBCSR performance significantly making it the fastest method in 8 of 50 cases, compared with the 2 cases in which the row-oriented tile format SBCSR is fastest.

The introduction of bitmap and scheduling hardware would doubtless drastically improve performance as all of the bitmap-testing, address-generation and scheduling could be performed in parallel with the floating-point operations making the execution-time for a dense tile the same as BCSR. Such a hardware-enhanced processor would have all of the advantages of SBCSR and BBCSR in terms power, storage and bandwidth efficiency and none of the disadvantages. A further advantage would be a reduction in terms of the number of instruction fetches and associated I-cache misses as a single hardware function-call with a bitmap parameter would replace a long sequence of instructions. And finally the use of accumulators to store y values as they are accumulated and x values to hold the input vector while A matrix values are streamed directly to functional units without first being loaded into registers and then retrieved would greatly reduce register pressure inside the processor as well as latency and power dissipation associated with register file access.

# 7

# Chapter 7

*"Pray be as trivial as you can"*

*- Mabel Chiltern: "An ideal husband" (Oscar Wilde)*

## 7 Hardware Support for Bitmap SMVM

As seen previously in Chapter 4 memory and I/O bandwidth are fundamental limitations on the design of computer systems as no matter what memory technology is used to support computation. Independently of what technology is used to connect that memory to the processor, a particular implementation of a memory subsystem using those technologies has an upper limit on how much information can be transferred between processor and memory in a given time, this is the available memory bandwidth and the limitation of compute power by available memory bandwidth is often referred to as the "memory-wall" [42].

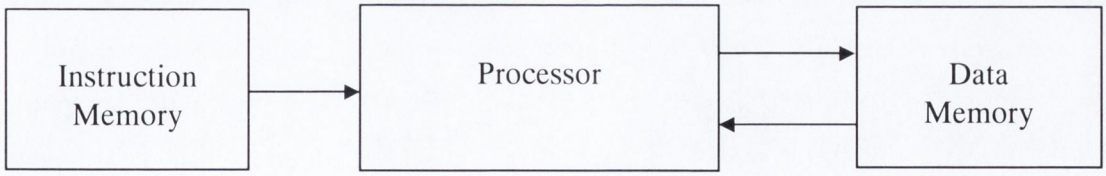



**Figure 7-1 Processor-Memory System**

Indeed in the case of multicore processors or MPSoCs (Multi Processor System on Chip) contention between multiple processors for shared resources lowers the achievable limit in terms of memory subsystem bandwidth. As can be seen these matrices are quite sparse, containing zeroes which require memory bandwidth to load/store while not contributing to the

result of the calculations performed using the whole matrix, i.e. they are trivial values from the point of view of arithmetic.

## 7.1 Observations on Software SMVM

SMVM performance depends on the interplay of many system parameters including:

- External Memory Bandwidth and fill demands due to the sparse matrix A
- Data cache misses due to locality issues to do with the x and y vectors
- Processor register spillage to the tile size, intermediate variables and control code
- Load/Store and Register file overheads
- RAW hazards due to dependencies in matrix vector product from column to column
- Branch penalties due to control code
- Instruction cache misses due to the chosen tile-size for register blocking

All of these issues combined mean that there is no single tile size which is optimal for each processor; even those sharing the same instruction-set, as different models of processor often have different-sized caches, differences in cache-hierarchy (whether an L3 cache is included for instance) and the width and clock frequency of the external memory bus.

As was seen in chapter 6 one of the big issues with BCSR is the amount of instruction code required as all loops are fully unrolled and many instructions are required to set up and hold variables required in the unrolling process. This problem was further exasperated by the additional bitmap testing and scheduling code in BBCSR and SBCSR respectively which generates further instructions. The number of instructions per SMVM partial-product for each of the 3 methods is shown in Listing 7-1. In the 3 cases the core arithmetic operations common to all SMVM methods are highlighted, whereas the overhead instructions to conditionally execute those core instructions depending either on the bitmap or schedule are not. As can be seen there are 6 core arithmetic instructions common to all 3 methods, while BBCSR and SBCSR have 9 and 11 additional overhead instructions respectively.

It has been seen that in both BBCSR and SBCSR cases much of the gain in data-path performance owing to trivial operand (fill) elimination which itself is data-dependent is offset by other factors to do with the increased code size of the proposed methods, including:

- Increased register pressure and spillage due to loop unrolling
- Increased instruction code bandwidth requirements
- Increased Instruction-cache misses due to the large size of unrolled code
- Overheads associated with bitmap or schedule processing
- Branch penalties

| | BCSR | BBCSR | SBCSR |
|---|---|---|---|
| C-Code | y0 += value[0] * x0; | if (bitmap&32768) y0 += value[_nz++] * x0; | goto *jt[*schedule++]; j0: y0 += value[_nz++] * x0; |
| Assembler | | | movl 20(%ebp), %eax<br>movzbl (%eax), %edx<br>leal 20(%ebp), %eax<br>incl (%eax)<br>movl _bbcsr_smvm4x4_jt(,%edx,4) , %edx<br>movl %edx, -92(%ebp)<br>L76:<br>jmp *-92(%ebp)<br>L53: |
| | | movl -16(%ebp), %eax<br>shrl $15, %eax<br>andl $1, %eax<br>testb %al, %al<br>je L36<br>movl -12(%ebp), %eax<br>leal 0(,%eax,8), %edx | movl -12(%ebp), %eax<br>leal 0(,%eax,8), %edx<br>000000 |
| | movl 20(%ebp), %eax<br>fldl (%eax)<br>fmull -56(%ebp)<br>fldl -24(%ebp)<br>faddp %st, %st(1) | movl 32(%ebp), %eax<br>fldl (%edx,%eax)<br>fmull -64(%ebp)<br>fldl -32(%ebp)<br>faddp %st, %st(1) | movl 36(%ebp), %eax<br>fldl (%edx,%eax)<br>fmull -64(%ebp)<br>fldl -32(%ebp)<br>faddp %st, %st(1) |
| | | leal -12(%ebp), %eax<br>incl (%eax) | leal -12(%ebp), %eax<br>incl (%eax) |
| | fstpl -24(%ebp) | fstpl -32(%ebp) | fstpl -32(%ebp) |
| | | L36: | |
| # Instructions | 6 | 15 | 17 |

**Listing 7-1 Register-Blocked SMVM Relative Code Sizes**

## 7.2 The Argument for Hardware Acceleration

Given the class of applications being considered are based largely on Sparse Matrix Vector Multiplication and the software methods outlined in Chapter 6 which allow sparsity to be addressed in a platform independent way in software, it makes sense to attempt to extend the performance of software by providing hardware support to accelerate data compression and decompression as well as computation using compressed sparse data-structures.

Given these requirements the author proposes that in order to maximise the benefits of register-blocking a hardware accelerator is necessary rather than relying entirely on software for SMVM performance in future processor architectures.

The introduction of bitmap and scheduling hardware would doubtless drastically improve performance as all of the bitmap-testing, address-generation and scheduling could be performed in parallel with the floating-point operations making the execution-time for a dense tile the same as BCSR. Such a hardware-enhanced processor would have all of the advantages of both SBCSR and BBCSR in terms power, storage and bandwidth efficiency and none of the disadvantages of either method. A further advantage would be a reduction in terms of the number of instruction fetches and associated I-cache misses as a single hardware function-call with a bitmap parameter would replace a long sequence of instructions. And finally the use of accumulators to store y values as they are accumulated and x values to hold the input vector while A matrix values are streamed directly to functional units without first being loaded into registers and then retrieved would greatly reduce register pressure inside the processor as well as latency and power dissipation associated with register file access.

The accelerator seeks to increase the effective memory bandwidth for sparse data-structures and minimise the limitation of the "memory-wall" on computation by storing data in a compressed format, and providing a means of compression and decompression which is suitable for block-structured data used in many applications such as computer graphics, rigid-body dynamics, finite-element analysis and other scientific and engineering applications, which operate on large data sets which must be stored in memory.

In order to mitigate the effect of the "memory-wall" the processor pipeline is also modified in such a way as to take advantage of compression, increasing the processing rate beyond what can be achieved by operating on compressed data alone. A typical example of the desirability of compression is the use of matrix representations and linear algebra operators to simulate reality on a 2-dimensional screen in computer graphics and related applications. In 3D graphics for instance, operations on a source data matrix often consist of rotations and other transformations, and often sequences of them, of the type shown in Figure 7-2.

| | | |
|---|---|---|
| $S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$<br>scaling matrix | $T(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$<br>translation matrix | |
| $R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$<br>$x$-axis rotation matrix | $R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$<br>$y$-axis rotation matrix | $R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$<br>$z$-axis rotation matrix |
| $SH_{xy}(sh_x, sh_y) = \begin{bmatrix} 1 & 0 & sh_x & 0 \\ 0 & 1 & sh_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$<br>$xy$ shear matrix | $SH_{xz}(sh_x, sh_z) = \begin{bmatrix} 1 & sh_x & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & sh_z & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$<br>$xz$ shear matrix | $SH_{yz}(sh_y, sh_z) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ sh_y & 1 & 0 & 0 \\ sh_z & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$<br>$yz$ shear matrix |

**Figure 7-2 Transformation Matrices used in 3D Graphics**

## 7.3 Prior Art

A review of memory compression and decompression approaches was provided in section 4.1.7. A key problem for programmers using compressed memory sub-systems is that data has to be decompressed before it can be operated upon as shown in Figure 7-3.

This usually involves reading the compressed data from one part of memory decompressing it and storing the decompressed data in another uncompressed portion of memory or to internal processor registers. In both cases valuable additional bandwidth and memory resources are consumed and the compression/decompression logic is typically an adjunct to cache memory in practical implementations. This solution has the disadvantage that additional memory bandwidth is required to read compressed data, store it in uncompressed form, and read it back into the processor to be operated upon. Additional memory capacity is also required to hold the uncompressed data and the decompression process will increase pressure on the processors register-files. In such schemes the utility of compression depends entirely on data-reuse amortising the costs of compression/decompression, relegating such schemes to situations where reuse is high. As has been pointed out previously, reuse in the case of

SMVM is negligible and limited to the x and y vectors only, while the A matrix values are used only once, unless multiple iterations of the SMVM operation are performed.

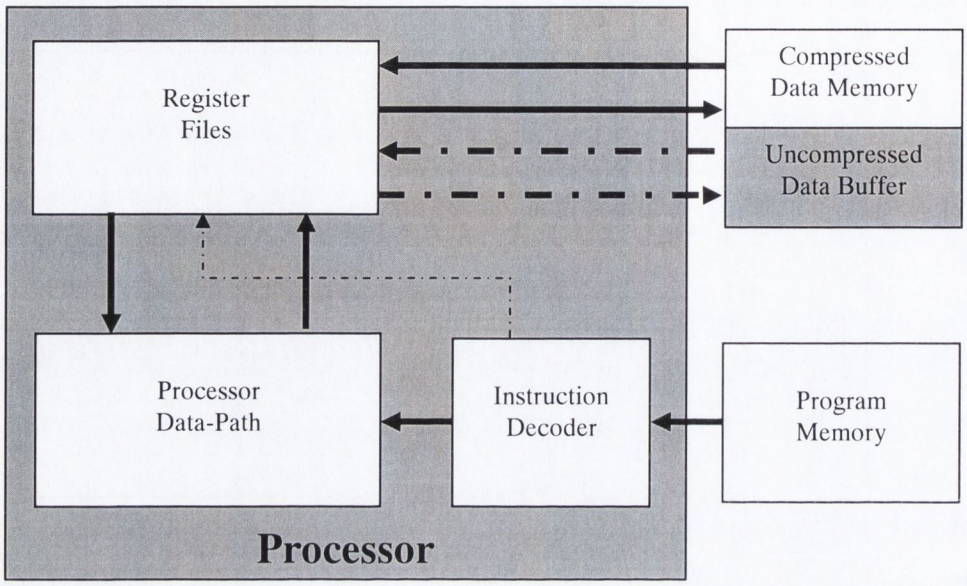



**Figure 7-3 Conventional Compressed Processor-Memory System**

Clearly this is a sub-optimal solution which explains why such compressed memory subsystems have remained an academic curiosity rather than entering the industry mainstream. A further issue with current compressed memory systems is the inability to randomly access compressed or hybrid data-structures while compressed in memory. Random access in such systems normally means uncompressing the data to an area of memory or registers in order to perform random access which is obviously inefficient from a memory, power and processor throughput point-of-view.

## 7.4 Proposed Solution

The proposed hardware accelerator is shown in Figure 7-4 and allows an appropriately modified processor to operate directly on compressed data in memory without the requirement for decompression, thus eliminating the requirement for an additional uncompressed data buffer and additional processor and memory bandwidth required to handle decompression of compressed data into a buffer area for further processing.

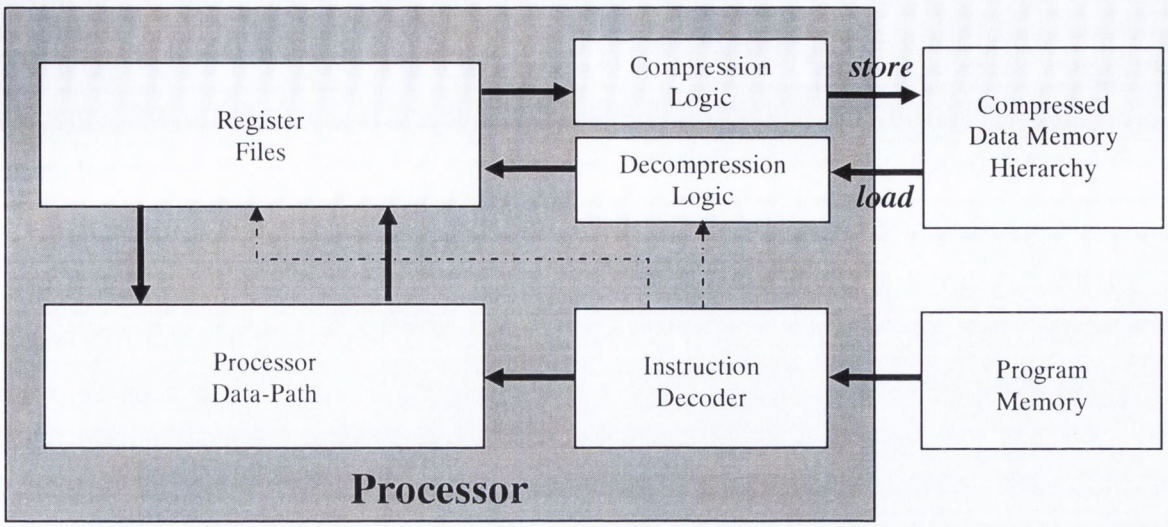



**Figure 7-4 Proposed Compressed Processor-Memory System**

The accelerator allows compressed and hybrid compressed/uncompressed structures of arbitrary size and complexity, consisting of arbitrary data, including the following:

- double/single/16-bit precision floating-point matrices, vectors and scalars
- signed/unsigned integer matrices, vectors and scalars
- signed/unsigned characters (8-bit numbers) matrices, vectors and scalars
- address-pointers

In the preferred embodiment of the proposed invention the compression and decompression logic is further hidden within the processor, freeing the software programmer from the low-level mechanics of reading or writing to the compressed memory subsystem. The proposed method is not limited as in previous cases to a particular level in the memory hierarchy, ex between L2 and L3 cache, but is fully transparent to all levels of the hierarchy maximising flexibility and applicability to a wide range of applications and data-reuse frequencies.

## 7.5 Basic Compression Method

As was seen in Chapter 6 and described in [52] Register-blocking is a useful technique for accelerating matrix algebra (particularly Finite-Element), however it has the disadvantage in that for many matrices (ex. Google) zero fill has to be added decreasing effective FLOPS, and increasing memory bandwidth requirements, both of which are commodities which are limited by current technology in modern computing systems.

In fact the growing gap between processing capabilities and memory bandwidth which are increasing at highly disparate rates of 50% and 7% per annum respectively is often referred to as the "Memory Wall". There have been many claims of "breaking" the memory wall and they usually consist of using a cache to reduce the probability of having to go off-chip, and/or

using multi-threading so that the latency and penalties associated with going off-chip can be mitigated. These approaches merely hide the problem of limited external memory bandwidth rather than solving it and for this reason it is not uncommon to see large engineering applications pulling down processor performance to 10% or less of the manufacturers quoted peak performance specification [26].

The proposed solution generally is to use a bitmap to code for zero and non-zero entries in dense sub-matrix in the manner shown in Figure 7-5. In the example shown a compression of 37.5% is achieved. The proposed solution has the advantage that each zero entry in the matrix to be stored is represented by a 1-bit rather than a 32-bit single-precision, 64-bit double-precision, etc. number. Overall compression achieved as long as 1 or more entries per dense-matrix are zero in a 32-bit memory system storing a sparse matrix with 32-bit single-precision entries. A similar argument holds for a 64-bit or other basic storage unit, whether integer or floating-point.

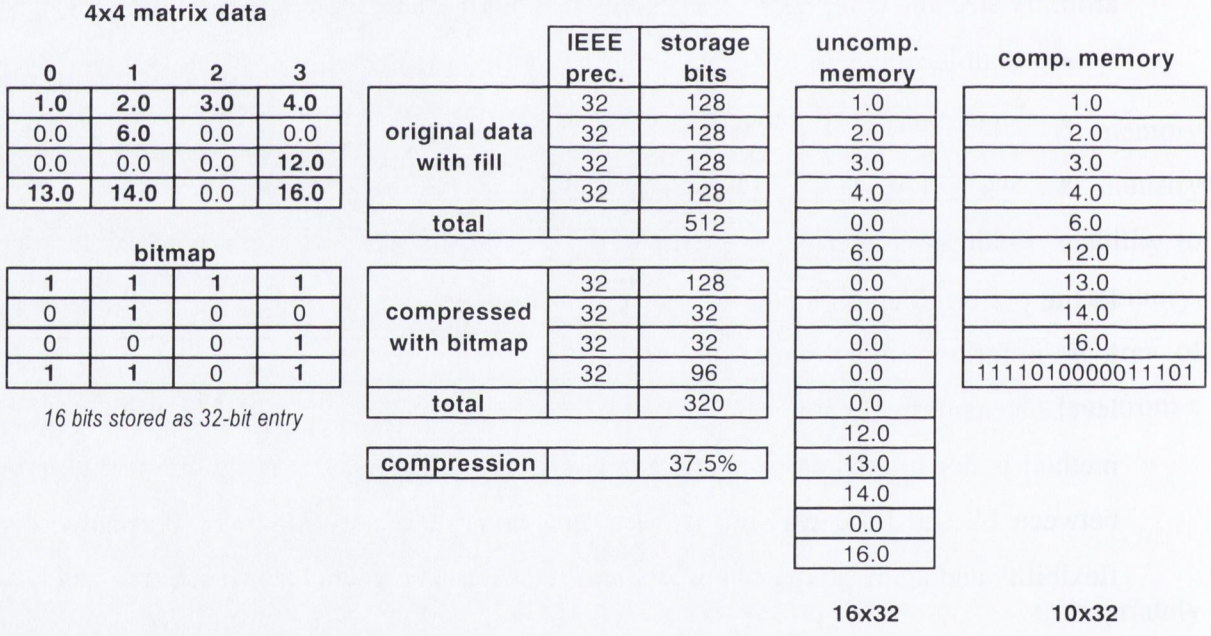



**Figure 7-5 Bitmap Compression Principle**

## 7.6 Conventional Sparse Matrix-Vector Multiplication

One of the key uses of sparse data-structures in computer science and engineering applications is the storage of sparse matrices and the key application of sparse matrices is in the solution of systems of simultaneous equations by either direct or iterative methods. The core operation at the heart of these direct and iterative methods is the multiplication of a sparse matrix by a dense vector to produce a dense result vector.

The calculation is of the form $y = Ax$, where A is a sparse matrix and y and x are dense vectors as shown in Equation 7-1.

$$y\begin{bmatrix}0\\1\\2\\3\end{bmatrix} = A\begin{bmatrix}00 & 01 & 02 & 03\\10 & 11 & 12 & 13\\20 & 21 & 22 & 23\\30 & 31 & 32 & 33\end{bmatrix} * x\begin{bmatrix}0\\1\\2\\3\end{bmatrix}$$

**Equation 7-1 Sparse Matrix-Vector Multiplication**

The detailed calculations for a 4x4 sparse matrix-vector multiplication performed row-wise are shown in Equation 7-2.

$$\begin{aligned}
y_0 &= a_{00} * x_0 + a_{01} * x_1 + a_{02} * x_2 + a_{03} * x_3\\
y_1 &= a_{10} * x_0 + a_{11} * x_1 + a_{12} * x_2 + a_{13} * x_3\\
y_2 &= a_{20} * x_0 + a_{21} * x_1 + a_{22} * x_2 + a_{23} * x_3\\
y_3 &= a_{30} * x_0 + a_{31} * x_1 + a_{32} * x_2 + a_{33} * x_3
\end{aligned}$$

**Equation 7-2 4x4 Sparse Matrix Vector Multiplication**

In a row-based formulation the elements in the y result vector are computed one row at a time from a row of the A matrix multiplied by the x vector. In general the form of the multiplication and summation is shown in Equation 7-3.

$$y[row] = a[row, col0] * x[col0] + a[row, col1] * x[col1] + a[row, col2] * x[col2] + a[row, col3] * x[col3]$$

**Equation 7-3 Vector Computation (y)**

The steps involved in dense matrix-vector calculations are:

- pre-load x vector into registers within the processor (reused for all y entries)
- initialise y vector
- read A matrix element-by-element or row-by-row into registers within the processor depending on the width of the data-bus
- multiply a[row, col] by x[col] and sum with y[row]
- repeat until all rows/columns have been processed

In the case of a sparse matrix many of the A.x terms in Equation 7-3 will obviously be zero as many of the columns within a row of the sparse A matrix will be zero. Conventional implementations of sparse matrix-vector multipliers have no means of knowing and/or avoiding trivial multiplications where an element of the A matrix is sparse, resulting in relatively longer run-times and power-dissipation for the overall matrix-vector multiplication.

## 7.7 Compressed Sparse Matrix-Vector Multiplication

If the Sparse Matrix has been compressed using the bitmap compression method outlined in section 7.5 the bitmap designates which matrix elements are zero allowing trivial multiplications to be eliminated and summations of y vector elements from constituent partial-products to be simplified. As the bitmap entries are 1-bit, the multiplication operation reduces to a logical AND or an if statement in C.

$$y_0 = bm_{00} * a_{00} * x_0 + bm_{01} * a_{01} * x_1 + bm_{02} * a_{02} * x_2 + bm_{03} * a_{03} * x_3$$
$$y_1 = bm_{04} * a_{10} * x_0 + bm_{05} * a_{11} * x_1 + bm_{06} * a_{12} * x_2 + bm_{07} * a_{13} * x_3$$
$$y_2 = bm_{08} * a_{20} * x_0 + bm_{09} * a_{21} * x_1 + bm_{10} * a_{22} * x_2 + bm_{11} * a_{23} * x_3$$
$$y_3 = bm_{12} * a_{30} * x_0 + bm_{13} * a_{31} * x_1 + bm_{14} * a_{32} * x_2 + bm_{15} * a_{33} * x_3$$
$$bm_n \in \{0,1\}$$

**Equation 7-4 Compressed Matrix-Vector Multiplication**

Based on the bitmap compression the sparse matrix-vector multiplication can be decomposed into the following steps:

- pre-load x vector into registers within the processor (reused for all y entries)
- initialise y vector and read in bitmap into internal register
- Expand bitmap into uncompressed schedule for SMVM and store in register
- Compress schedule to perform only multiplications corresponding bitmap non-zeroes
- Multiply a[row, col] by x[col] and sum with y[row] according to compressed schedule
- Repeat until all rows/columns have been processed

The transformation matrices used in 3D graphics and game physics (rigid-body dynamics) applications shown in Figure 7-2 are a good example. The 32-bit bitmaps corresponding to the 8 matrices are shown in detail in Table 7-1.

| *matrix* | *upper 16 bits are unused 0x000* | | | | | | | | | | | | | | | | *nz* | *comp %* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| scaling | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 4 | 68.8% |
| translation | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 7 | 50.0% |
| ***x*** -axis rot. | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 6 | 56.3% |
| ***y*** -axis rot. | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 6 | 56.3% |
| ***z*** -axis rot. | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 6 | 56.3% |
| ***xy*** shear | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 6 | 56.3% |
| ***xz*** shear | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 6 | 56.3% |
| ***yz*** shear | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 6 | 56.3% |

**Table 7-1 Compression Bitmaps of Graphics Transformation Matrices**

As can be seen Table 7-1 3D graphics transformation matrices contain a large percentage of trivial (zero values) allowing over 50% data-compression to be achieved.

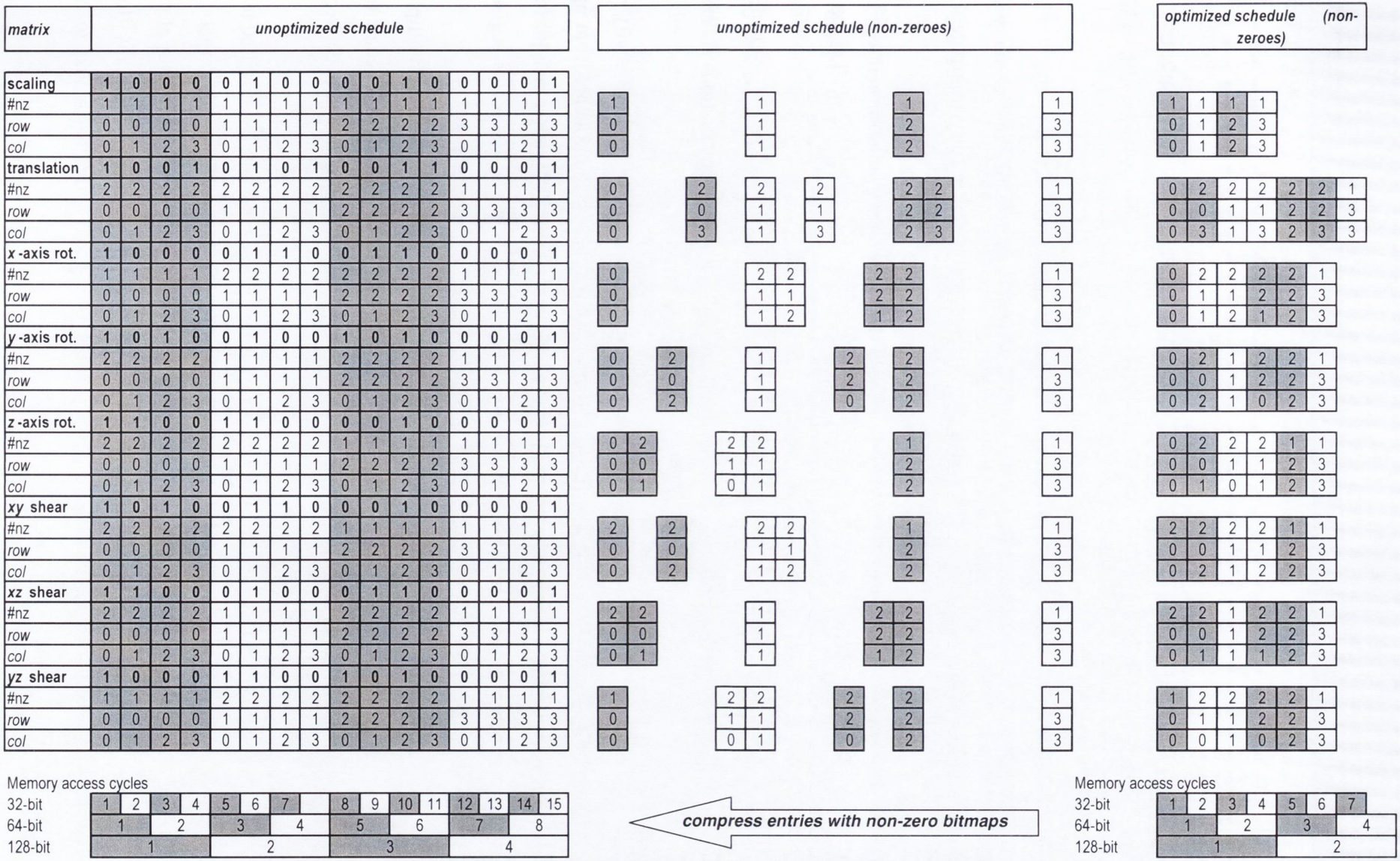



**Figure 7-6 3D Graphics SMVM Scheduling Examples**

## 7.8 Accelerator Overview

The goal is to implement a streaming SMVM coprocessor which can be integrated into a processor-based system as either a coprocessor or built into an existing processor pipeline as a specialised complex instruction or tightly coupled coprocessor. In order to maximise efficiency the A-matrix will be accommodated in either a very large on-chip memory or more likely in external commodity SDRAM allowing the solution to scale to arbitrary sized-problems. The main goal of building a coprocessor is to maximise bandwidth efficiency, power and latency.

This is achieved by building an accelerator with the following attributes:

- Hardware bitmap compression of the A-matrix, and possibly the x and y-vectors maximises bus, memory controller and I/O bandwidth and is fully transparent to the memory hierarchy unlike current schemes
- Single instruction for entire matrix or tile SMVM product evaluation reduces I-cache bandwidth and misses, represents an enormous saving in both I-RAM/cache bandwidth and power compared with current architectures
- Single instruction is expanded into a schedule of sub-instructions by the onboard scheduler and controller, eliminating all fill and associated instructions to calculate fill-related products that are required in conventional processors
- Grouping of products across multiple-rows of the A-matrix where the sparsity pattern allows means fewer register-file reads and fewer cycles as components from up to N tile rows can be dispatched in parallel to an N-wide SIMD FPU rather than performing N separate MAC (Multiply Accumulate) operations because the vector register-file does not allow independent access to vector elements
- Streaming A-matrix access means the A-matrix values go directly to be multiplied by the appropriate x-vector entries without first having to be written to a vector register file, resulting in a large saving in power and latency compared with conventional processors

In the following sections a 4-way SIMD FPU is considered but this is easily modified to allow 1, 2, 4, 8 etc. FPUs to be used in parallel, depending on the performance target for the accelerator. Similarly in the following examples a 128-bit SIMD FPU comprising of 4 IEEE single-precision MAC units is considered but this is readily changed to handle any floating-point format including double and extended precision, or equally for that matter to include sparse matrices and vectors comprised of integers of arbitrary precision.

## 7.9 Functional Model of Accelerator

The proposed hardware accelerator has a hardware function-call style interface where a single matrix-vector multiply instruction is passed in along with pointers to the A matrix, x and y vectors, tile row and column dimensions and a bitmap representing the sparsity pattern. Effectively a single instruction is issues to the accelerator which is expanded into a long sequence of instructions depending on the bitmap pattern.

The tile-element ordering and the Sparse Matrix value array ordering are assumed to have been chosen ahead of time by the software applications programmer in a way consistent with the hardware interface and also in order to maximise overall system-performance. For instance the layout of the data-structures by the software programmer could also include scanning the tiles in a matrix row (group of rows) alternately from left to right and right to left in zigzag order to minimise cache misses associated with the x-vector at the end of rows thus extending the work of Yzelman and Bisseling [135]. The hardware accelerator offers the software programmer to continue to make all of these system-level trade-offs without any loss of flexibility.

In the case of the following abstract model of the hardware accelerator it is assumed for simplicity that the tile elements are arranged in column-wise order in order to minimise dependencies and hence RAW hazards. Simplified C-code for the basic block-smvm is shown in Listing 7-2, where bmp is the sparsity bitmap, r and c are the tile dimensions, y and x are the output and input vectors respectively and the array a, is a linear array of A matrix values.

```
L1.             void bsmvm(int bmp,int r,int c,double
   *y,double *a,double *x) {
L2.     struct element sch[16];
L3.     unsigned       nz=0;
L4.     unsigned       i=0;
L5.     double         *y_reg;
L6.     double         *x_reg;
L7.     double         *a_reg;

L8.     // generate schedule containing row/column addrs
L9.     // for all bitmap non-zeroes in parallel
L10.    //
L11.    if (bmp&0x8000) {sch[nz].r=0; sch[nz].c=0; nz++;}
L12.    if (bmp&0x4000) {sch[nz].r=1; sch[nz].c=0; nz++;}
L13.    if (bmp&0x2000) {sch[nz].r=2; sch[nz].c=0; nz++;}
L14.    if (bmp&0x1000) {sch[nz].r=3; sch[nz].c=0; nz++;}
L15.    //
L16.    if (bmp&0x0800) {sch[nz].r=0; sch[nz].c=1; nz++;}
L17.    if (bmp&0x0400) {sch[nz].r=1; sch[nz].c=1; nz++;}
```

```
L18.    if (bmp&0x0200) {sch[nz].r=2; sch[nz].c=1; nz++;}
L19.    if (bmp&0x0100) {sch[nz].r=3; sch[nz].c=1; nz++;}
L20.    //
L21.    if (bmp&0x0080) {sch[nz].r=0; sch[nz].c=2; nz++;}
L22.    if (bmp&0x0040) {sch[nz].r=1; sch[nz].c=2; nz++;}
L23.    if (bmp&0x0020) {sch[nz].r=2; sch[nz].c=2; nz++;}
L24.    if (bmp&0x0010) {sch[nz].r=3; sch[nz].c=2; nz++;}
L25.    //
L26.    if (bmp&0x0008) {sch[nz].r=0; sch[nz].c=3; nz++;}
L27.    if (bmp&0x0004) {sch[nz].r=1; sch[nz].c=3; nz++;}
L28.    if (bmp&0x0002) {sch[nz].r=2; sch[nz].c=3; nz++;}
L29.    if (bmp&0x0001) {sch[nz].r=3; sch[nz].c=3; nz++;}
L30.    //
L31.    // perform schedule of MACs using single FPU
L32.    y_reg = y;
L33.    x_reg = x;
L34.    a_reg = a;
L35.    while (i<nz) { // process one non-zero at a time
L36.    y_reg[sch[i].r] += a[i] * x_reg[sch[i].c];
L37.    i++;
L38.    }
L39.    // assign output of smvm operation
L40.    y = y_reg;
L41.    }// bsmvm()
```

**Listing 7-2 C-Code Hardware Block SMVM**

It should be noted that while column-major ordering is assumed in the above code it is trivial, either in hardware or in software, to swap row and column addresses in the schedule to accommodate a row-major format for the basic tile. Similarly it is simple to perform loop-unrolling in a manner similar to that used for the BCSR code in section 6.2.1 for a 4-way SIMD floating-point MAC as shown in Listing 7-3.

```
L1.  while (i<nz) { // schedule MACs using 4x SIMD FPU
L2.    y_reg[sch[i  ].r] += a[i  ] * x_reg[sch[i  ].c];
L3.    y_reg[sch[i+1].r] += a[i+1] * x_reg[sch[i+1].c];
L4.    y_reg[sch[i+2].r] += a[i+2] * x_reg[sch[i+2].c];
L5.    y_reg[sch[i+3].r] += a[i+3] * x_reg[sch[i+3].c];
L6.    i+=4; // advance by 4 non-zeroes
L7.  }
```

**Listing 7-3 SIMD HW Block SMVM C-Code (bsmvmX4)**

Here the 4 new values of the elements of the y-register are calculated in parallel using fixed offsets to look-up the row and column addresses of scheduled calculations 4 at a time (in one cycle) rather than one at a time as in the previous example. In practice the adders implicit in the code above will not be required as the pointer to the schedule array can be advanced 4-elements at a time rather than as shown above (for simplicity).

## 7.10 Accelerator Hardware Implementation

The accelerator consists of the following major functional blocks:

- Software interface
- Bitmap scheduler
- SMVM SIMD Datapath
- Control Logic
- Memory interface

In the following sections the design of the elements of the hardware accelerator will be explained in detail. A top-level block-diagram of the accelerator is shown in Figure 7-7.

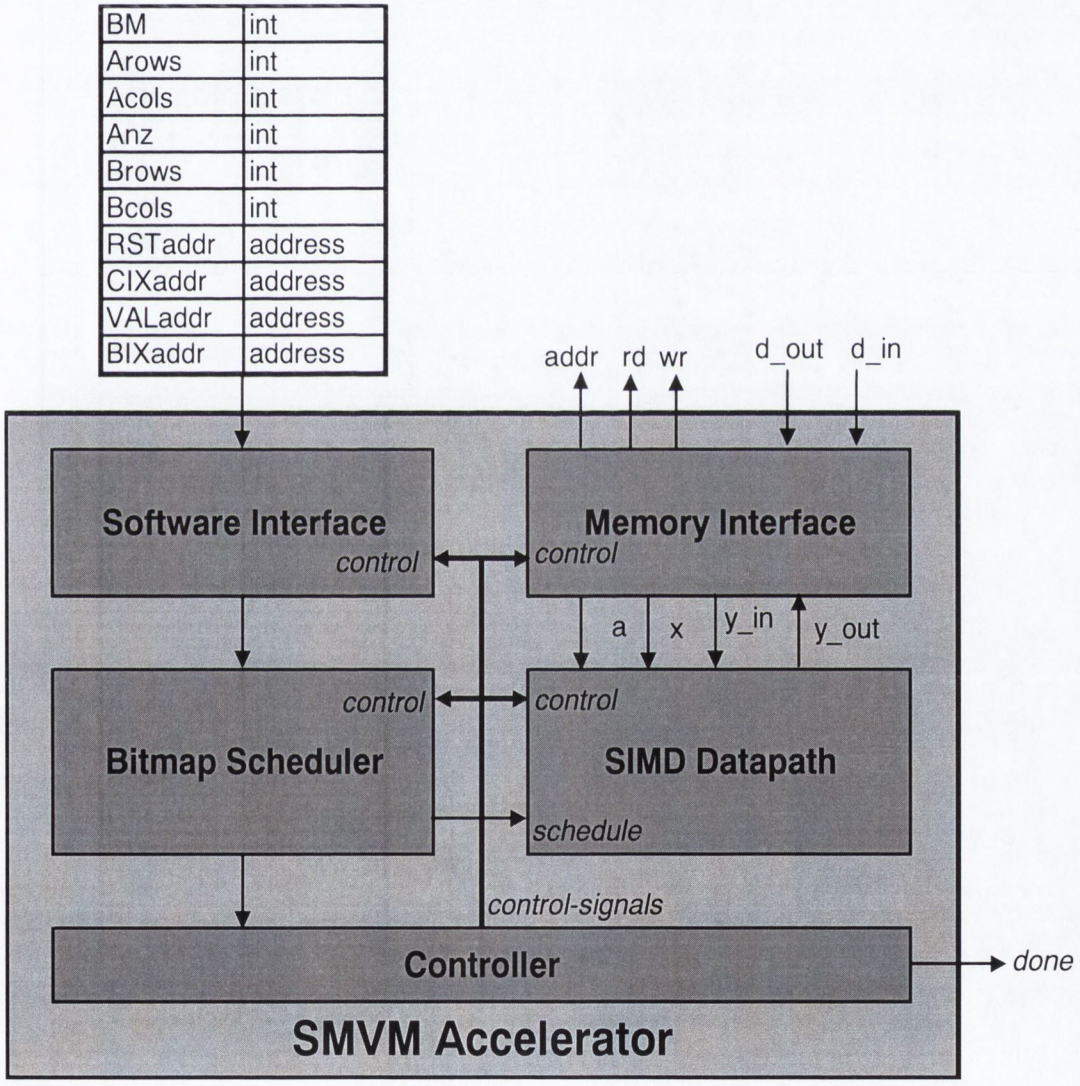



**Figure 7-7 Sparse SMVM Hardware Accelerator Block Diagram**

A system block-diagram showing how the accelerator is interfaced as a coprocessor to a host processor, caches, peripherals, memory controller and external SDRAM memory is shown in Figure 7-8.

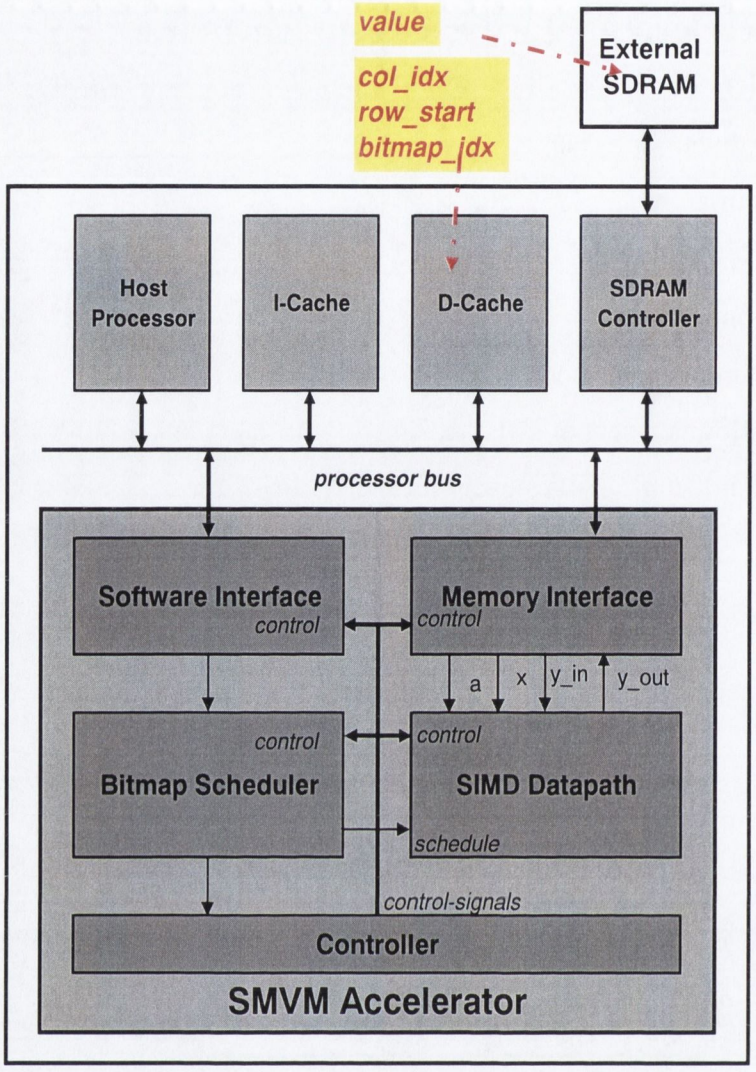



**Figure 7-8 Hardware SMVM Coprocessor**

The typical data layout for this kind of coprocessor will be to keep the values which exhibit most locality in the on-board caches while the A-matrix value array (and possibly the bitmap array) will typically be held in external SDRAM as it has no reuse unless the matrix is symmetric or if a matrix-matrix product is being computed.

A simplified but complete functional C-model for the accelerator is shown in Listing 7-4 and as can be seen replaces lines L17-L32 in Listing 6-5 with a single hardware function call (bsmvmX4 outlined in Listing 7-2 and Listing 7-3) which removes all of the conditional code which was shown to slow down bitmap SMVM execution. The main loop in the code processes the sparse matrix tile-by-tile until the entire y-vector result has been computed. This loop is the main outer element of the control logic. The inner loop then scans through the tiles in a row of matrix blocks, loading the bitmap for the tile, the segment of the x-vector and then generating the schedule from the bitmap, loading the A values and performing the

smvm multiplication for the tile according to the generated schedule. Additional logic is also required to generate the control signals necessary to load and store data held in caches or SDRAM.

```
L1.     void load_y(double *y, double *dest) {
L2.       y[0] = dest[0];
L3.       y[1] = dest[1];
L4.       y[2] = dest[2];
L5.       y[3] = dest[3];
L6.     } // load_y()

L7.     void store_y(double *y, double *dest) {
L8.       dest[0] = y[0];
L9.       dest[1] = y[1];
L10.      dest[2] = y[2];
L11.      dest[3] = y[3];
L12.    } // store_y()

L13.    void load_x(double *x, double *src, int *col_idx) {
L14.      x[0] = src[(*col_idx)    ];
L15.      x[1] = src[(*col_idx) + 1];
L16.      x[2] = src[(*col_idx) + 2];
L17.      x[3] = src[(*col_idx) + 3];
L18.    } // load_x()

L19.    void load_bmp(int bitmap, int *bitmap_idx) {
L20.      bitmap = *bitmap_idx & 0x0000FFFF;
L21.    } // load_bmp()

L22.    void hw_smvm4x4(int bm, int r, int c, int *row_start,
          int *col_idx, int *bitmap_idx, double *value, double *src,
          double *dest) {

L23.      int i, j, bitmap=0, nz=0;
L24.      double y[4], x[4];

L25.      for (i=0; i<bm; i++, dest+=r) {
L26.        load_y(y,dest); // load y-vector segment
L27.        for (j=row_start[i]; j<row_start[i+1]; j++,
          bitmap_idx++, col_idx++, value+=nz) {
L28.          load_bmp(bitmap,bitmap_idx); // load bitmap
L29.          load_x(x,src,col_idx); // load x-vec segment
L30.          // load A matrix values & do 4x4 bitmap smvm
L31.          bsmvmX4(bitmap,r,c,y,value,x);
L32.        }
L33.        load_y(y,dest); // store back
L34.      }
L35.    } // hw_smvm4x4()
```

**Listing 7-4 HW Accelerator (Abstract C-Model)**

In the case where the accelerator functionality is integrated into the processor pipeline rather than as a standalone accelerator only the bsmvmX4 functionality would be built into the hardware. In this way the inner and outer loops could be implemented in software for flexibility and the existing memory interface contained in the processor could be reused, thus maximising flexibility while reducing the amount of hardware required.

### 7.10.1 Software Interface

As can be seen in Figure 7-7 the software interface allows the following parameters for the matrix-vector product to be loaded into the accelerator:

- BM is the number of bitmap-blocked tiles in the A-matrix
- Arows the number of rows in the A-matrix
- Acols the number of columns in the A-matrix
- Anz the number of non-zero entries in the A-matrix
- Brows – number of rows in a block tile
- Bcols- number of columns in a block tile
- VALaddr – base address of the (A-matrix) value array
- RSTaddr – base address of the row-start array
- CIXaddr – base address of the col_idx array
- BIXaddr – base-address of the bitmap array

The data-structure that the accelerator accesses in memory is of the form shown in Figure 7-9. The figure is purely illustrative and the data-structures are fully independent and can occur in any order or position within the accelerator memory map.

These values are written into internal accelerator registers via a memory-mapped interface attached to the host processor memory bus, and as soon as they have been loaded computation can be enabled by writing the start code to the hardware-accelerator command register. The register map for the software interface including command register is shown in Table 7-2.

As can be seen the command register allows the accelerated smvm to be started, smvm calculations to be paused, a paused smvm to be resumed or the accelerator to be stopped and all registers reset. All registers are reset with the exception of the NZ-count which shows the number of A-matrix non-zeroes processed to date by the accelerator in the current smvm, and the cycle-count which shows the number of elapsed cycles in the current smvm operation.

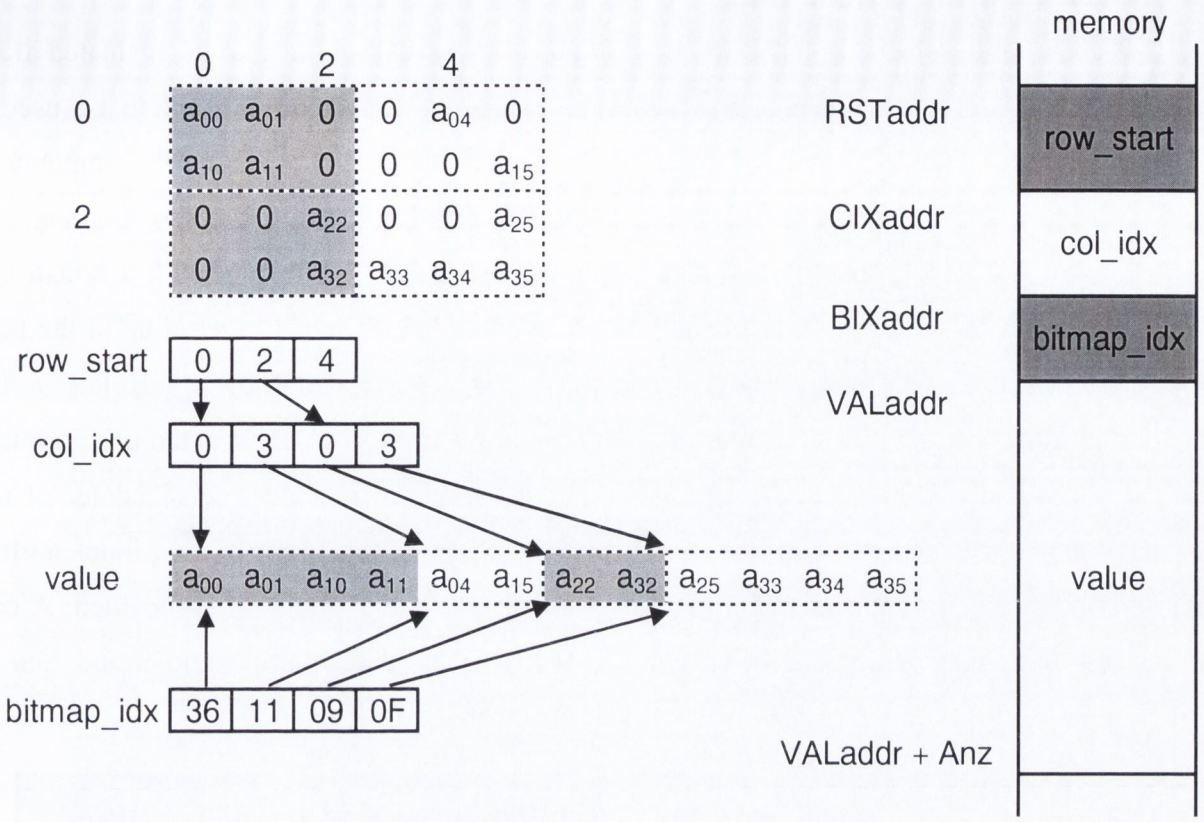



**Figure 7-9 HW Accelerator Data-Structure**

If required additional registers could easily be added to allow the programmer debug SMVM codes by interrogating the contents of the X and Y vector segment and other registers internal to the accelerator.

| Address | Name | Type | b[31:4] | b3 | b2 | b1 | b0 |
|---|---|---|---|---|---|---|---|
| 0x0000 | command | Read/Write | reserved | stop | resume | pause | start |
| 0x0001 | BM | Read/Write | 32-bit integer | | | | |
| 0x0002 | Arows | Read/Write | 32-bit integer | | | | |
| 0x0003 | Acols | Read/Write | 32-bit integer | | | | |
| 0x0004 | Anz | Read/Write | 32-bit integer | | | | |
| 0x0005 | Brows | Read/Write | 32-bit integer | | | | |
| 0x0006 | Bcols | Read/Write | 32-bit integer | | | | |
| 0x0007 | RSTaddr | Read/Write | 32-bit address | | | | |
| 0x0008 | CIXaddr | Read/Write | 32-bit address | | | | |
| 0x0009 | BIXaddr | Read/Write | 32-bit address | | | | |
| 0x000A | VALaddr | Read/Write | 32-bit address | | | | |
| 0x000B | NZcount | Read Only | 32-bit counter | | | | |
| 0x000C | CYCcount | Read Only | 32-bit counter | | | | |

**Table 7-2 HW Accelerator Control Registers**

### 7.10.2 Bitmap Scheduler

The bitmap scheduler generates a list of non-zero partial-products to be evaluated along with their relative column and row addresses along with a non-zero count to be used by the controller block. In this section the proposed scheduler hardware implements the functionality of lines L11 to L29 of Listing 7-2. The scheduler implementation and the list it produces is independent of whether a single FPU or SIMD FPU is used to evaluate and sum the SMVM partial-products. The bitmap schedule is compressed according to the bitmap as shown in Table 7-3. The rescheduling shown is achieved by controlling a bank of multiplexers and a re-scheduler of arbitrary complexity can be constructed using multiple bit-slices and multiplexers. In general using the proposed method a re-scheduler of arbitrary complexity can be constructed from an array of $(N^2+N)/2$, 4-bit by two-input multiplexers where N is the number of bitmap bits and corresponding slots to be scheduled. A complete 64-bit scheduler capable of scheduling 16 partial-product multiplications and four 4-input additions to sum the partial-products is shown in Figure 7-11.

| bitmap bit | note | schedule |
|---|---|---|
| 0 | unused slot (contains 0) | advance schedule of all slots one slot to the right of current position |
| 1 | required slot (contains non-zero) | do not advance schedule of slots to right of current position |

**Table 7-3 Bit-slice of Bitmap Scheduler**

As can be seen in the diagram the re-scheduler consists of 120 by 4-bit, 2:1 multiplexers with associated Look-Up Tables (LUTs). The majority of the logic however is comprised by the four hundred and eighty 2:1 multiplexers. If the re-scheduler is included as part of a programmable processor pipeline it can also function as a general purpose 64-bit shifter (in steps of 4 bits or multiples of 4 bits) if an additional 2:1 multiplexer is included at the input to select between LUT outputs and an input register or bus.

The C-code for the population-counter is shown in Listing 7-5.

```
L1.     int popcount(int bmp) {
L2.       int nzc = 0;

L3.       // count # non-zeroes in bitmap
L4.       if (bmp&0x8000) nzc++;
L5.       if (bmp&0x4000) nzc++;
L6.       if (bmp&0x2000) nzc++;
L7.       if (bmp&0x1000) nzc++;
L8.       if (bmp&0x0800) nzc++;
```

```
L9.       if (bmp&0x0400) nzc++;
L10.      if (bmp&0x0200) nzc++;
L11.      if (bmp&0x0100) nzc++;
L12.      if (bmp&0x0080) nzc++;
L13.      if (bmp&0x0040) nzc++;
L14.      if (bmp&0x0020) nzc++;
L15.      if (bmp&0x0010) nzc++;
L16.      if (bmp&0x0008) nzc++;
L17.      if (bmp&0x0004) nzc++;
L18.      if (bmp&0x0002) nzc++;
L19.      if (bmp&0x0001) nzc++;
L20.      return(nzc);
L21.  } // popcount()
```

**Listing 7-5 Population-Counter C-code**

The final element of the scheduler is an iteration-counter which determines the number of arithmetic iterations necessary to perform the SMVM calculations using an N-element wide SIMD FPU. The C-code for the iteration-counter is shown in Listing 7-6.

```
L1.   int itercount(int nzc) {
L2.     int iter;
L3.     int round;

L4.     // calculate # SIMD cycles to perform SMVM on tile
L5.     // divide nzc by 4 as SIMD does 4 MACs/cycle
L6.     // round up if 2 lsbs of NZ count are 11/10/01
L7.     //
L8.     round = (nzc&2 | nzc&1) ? 1 : 0;
L9.     iter  = (nzc/4) + round;

L10.    return(iter);
L11.  } // itercount()
```

**Listing 7-6 Iteration-Counter C-Code**

The block-diagram for the population and iteration counters, as well as the truth-table for the full adder is shown in Figure 7-12. The population/iteration counter consists of a tree of full-adders which computes the sum of the non-zero bitmap bits and returns it as a 5-bit binary number (required precision to represent 16 bitmap bits).

In general a modified structure can be constructed to incorporate the scheduler functionality within a general-purpose 64-bit shifter. The resultant shifter can function in one of 4 possible modes:

- Bitmap scheduler mode where LUT outputs are grouped according to the input bitmap to compose a minimum length multiplication/addition schedule (only non-zeroes)

- Shifter mode where the 64-bit input is shifted left according to the bitmap applied
- 64-bit right shifter by using an input and output stage to reverse bit order before performing a left shift
- Grouper/extractor mode where the bytes from the 64-bit input are grouped together according to the bitmap bits applied (in multiples of 4-bits)

The grouper mode could be useful in extracting bytes quickly from a complex data-structure. Another element of the scheduler is a population-counter which counts the number of ones in the bitmap. The scheduler works by only selecting row/column address pairs, for which there is a corresponding non-zero bitmap entry, and compacting them together into a single schedule of operations for which no trivial operations need be executed in order to compute the matrix-vector product. The scheduler is composed of byte-wide 2:1 muxes. For simplicity a 1-bit wide 2:1 mux is shown in Figure 7-10.

| EN | A | B | Z |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

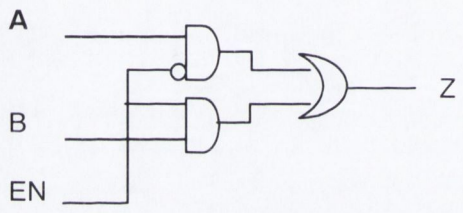



**Figure 7-10 Multiplexer (2:1) Truth-Table and Logic Diagram**

If a bitmap entry is non-zero the multiplexer output is the local row/col address pair. If on the other hand the bitmap entry is zero, the local row/col pair can be skipped and the row/col pair to the right of the current pair is selected as the multiplexer output. This selection procedure is continued from right to left for the 16 bitmap entries and corresponding row/col pairs. The result at the end of the selection procedure is that only those row/col pairs corresponding to bitmap non-zeroes are grouped and concatenated by row, and then by column from left to right, eliminating any trivial operations. The logic is repetitive and contains mainly local wiring, with the exception of the bitmap entries used to control each line of selection multiplexers. This regularity as well as the use of simple components results in a fast and area efficient design.

The hardware required for the entire scheduler consists of approximately 1440 gates to implement the scheduler from 480 2:1 multiplexers and to implement the population/iteration counter requires 19 FA (Full-Adder) cells and one OR gate so a total of around 115 gates. The total hardware requirement for the scheduler is therefore around 1.6k gates.

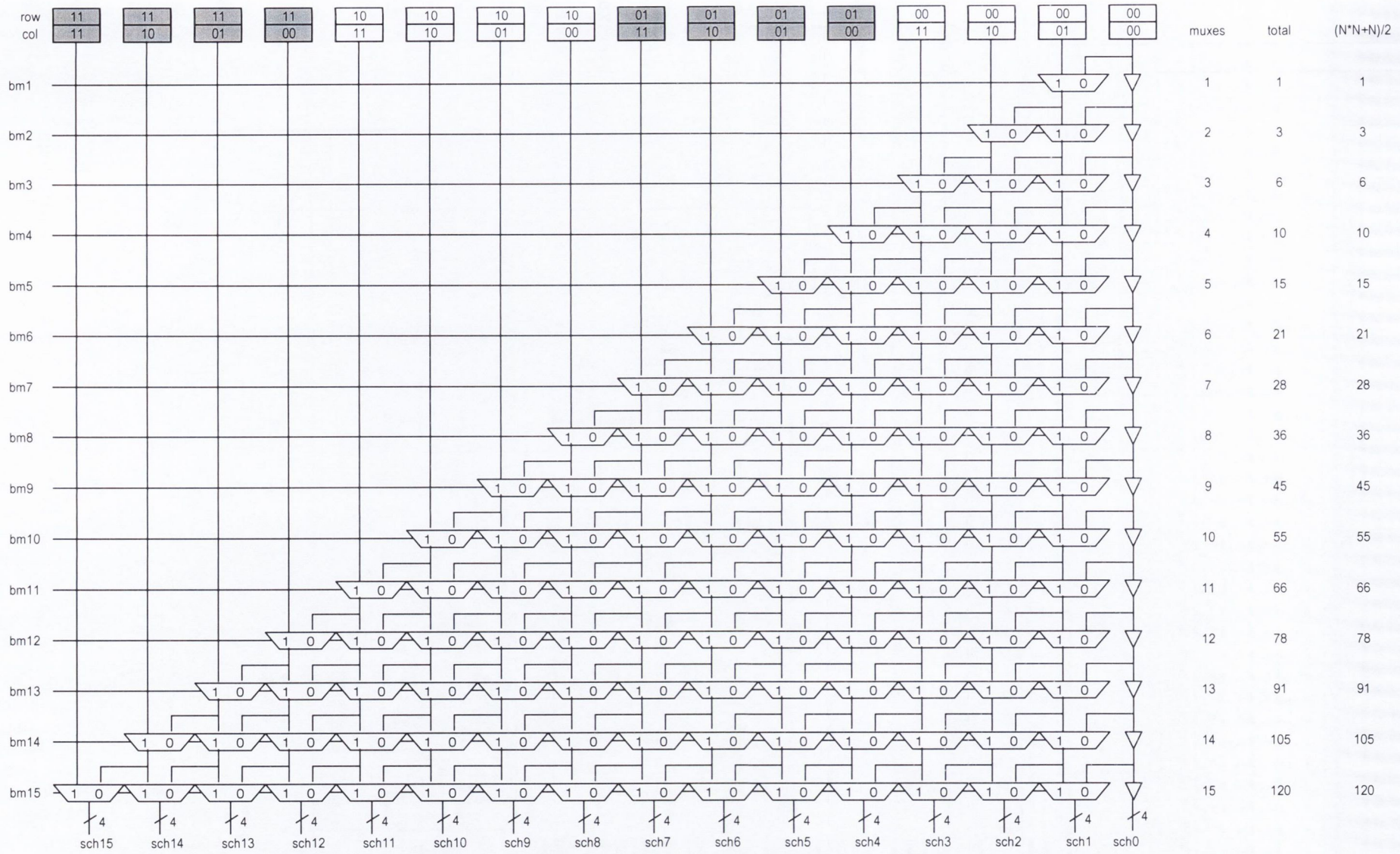



**Figure 7-11 Bitmap Scheduler Implementation (64-bit = 16 x 4-bit)**

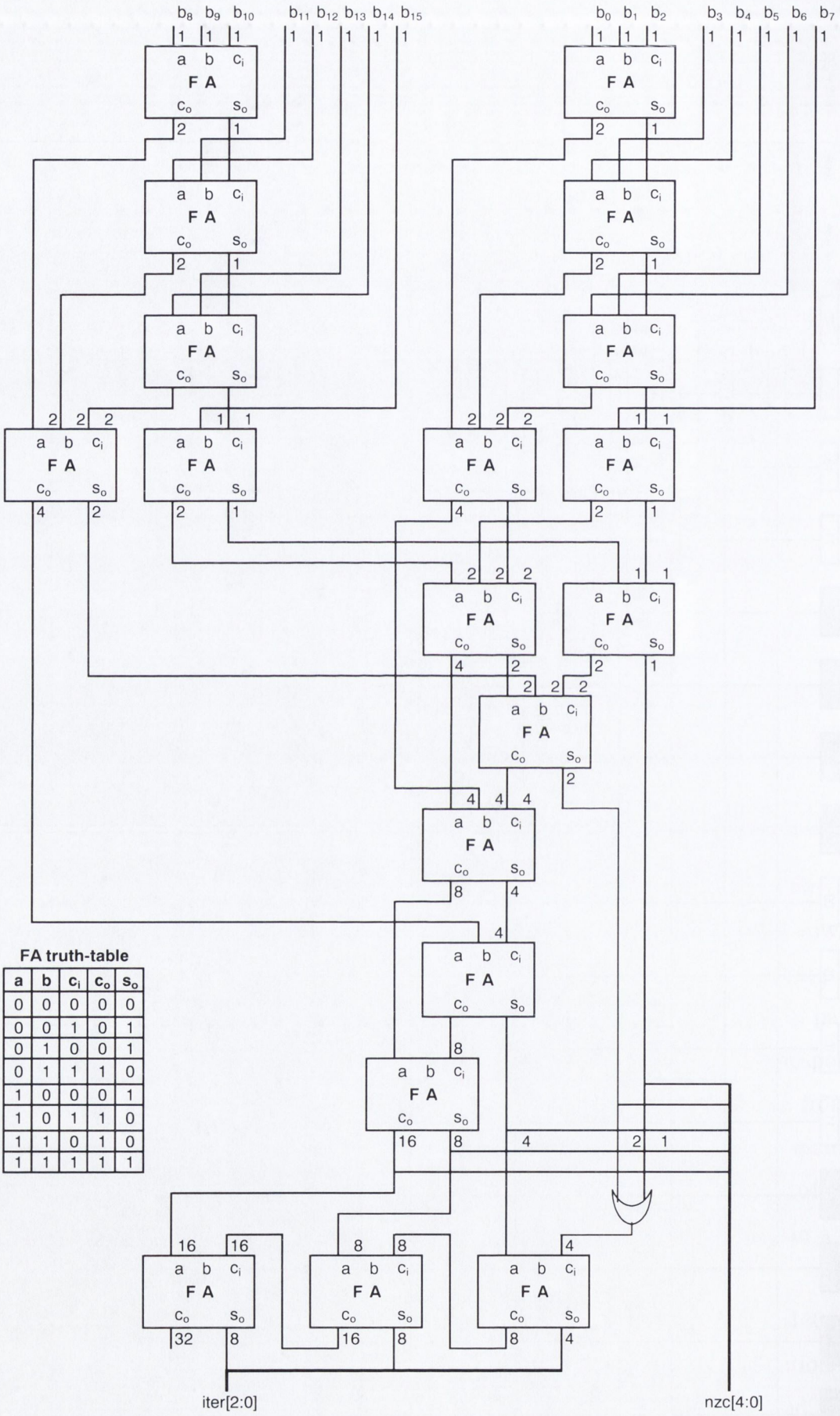



**Figure 7-12 Population & Iteration Counter Logic**

### 7.10.3 Control Logic

The control-logic for the accelerator applies all of the relevant control signals along with column and row addresses from the bitmap generated schedule to the internal blocks in order to ensure the correct products are calculated, summed and stored back to the correct y-registers. In the control logic signals are generated to:

- Load y-vector entries into internal registers corresponding to each row of tiles across the A-matrix (load_y control signal)
- Load bitmap for tile into register (load_bmp)
- Generate schedule from tile bitmap
- Load x-vector entries into internal registers corresponding to each A-matrix tile (load_x)
- Stream (Read) A entries from memory (load_a)
- Select the correct x vector entries to be multiplied by each A-matrix entry
- Evaluate each A.x partial product in sequence (amultx)
- Select the correct y value to be updated by adding the A.x partial-product in the FP adder
- Update the correct y-vector registers
- Write y-vector register contents back to memory at the end of an A-matrix row

The example matrix and data-structure are shown in **Figure 7-13** and the corresponding control signals are shown in Figure 7-14.

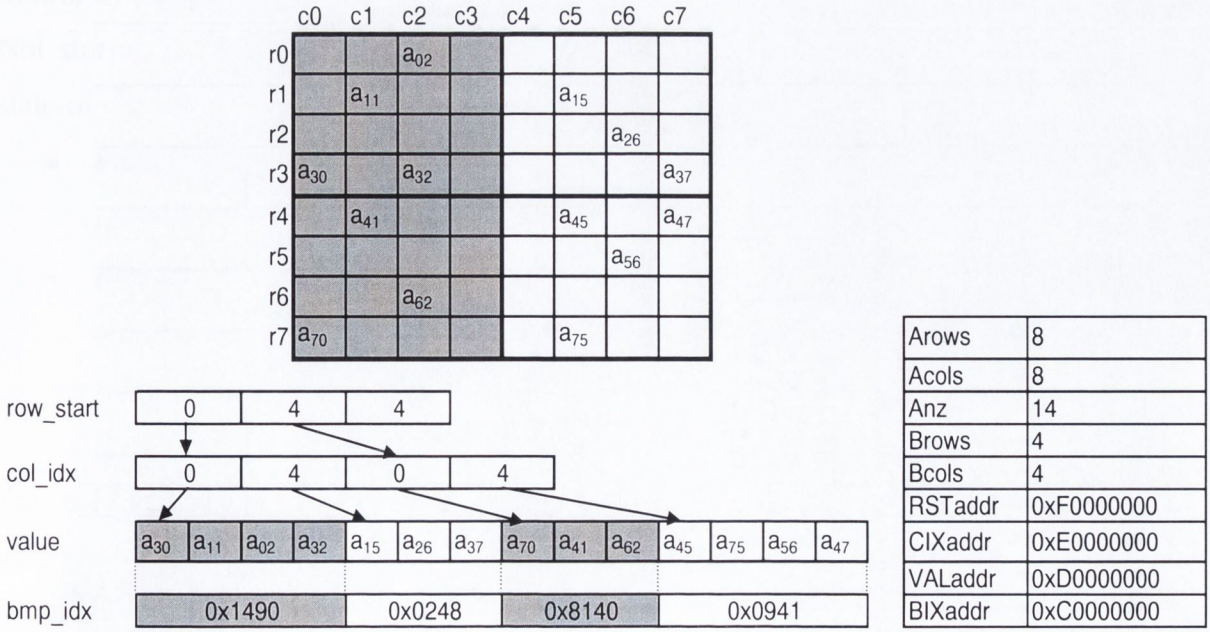



**Figure 7-13 Example Matrix & Data-Structures**

The segment of the timing diagram shown assumes a 4x4 block tile and single FP multiplier and FP adder, each with single clock-cycle latency, rather than a SIMD unit for simplicity and the period for which the control-signals are shown correspond to the first 2 tiles and relative bitmaps. Note that the timing-diagram is simplified and does not include the datapath source and destination multiplexer control-signals derived from the schedule.

The y-register is initially loaded with 4 values that hold for the first 2 matrix tiles. Once these values have been loaded the bitmap corresponding to the first matrix tile is fetched, and a schedule is generated. Next the first 4 x-register values are loaded in the next 4 clock-cycles. Following this, the first 4 non-zero A-matrix values are fetched from the value array in memory and multiplied by the x-register entries to produce 4 partial products. These partial-products are then summed with the 4 y-vector entries stored in the y-register over 4 cycles. Next the second tile and associated bitmap are processed updating the y-register values to complete the matrix-vector product. Finally the y-vector values corresponding to the first row of A-matrix tiles are written back to memory from the y-register and the computation of the smvm product corresponding to the next row of A-matrix tiles can be computed.

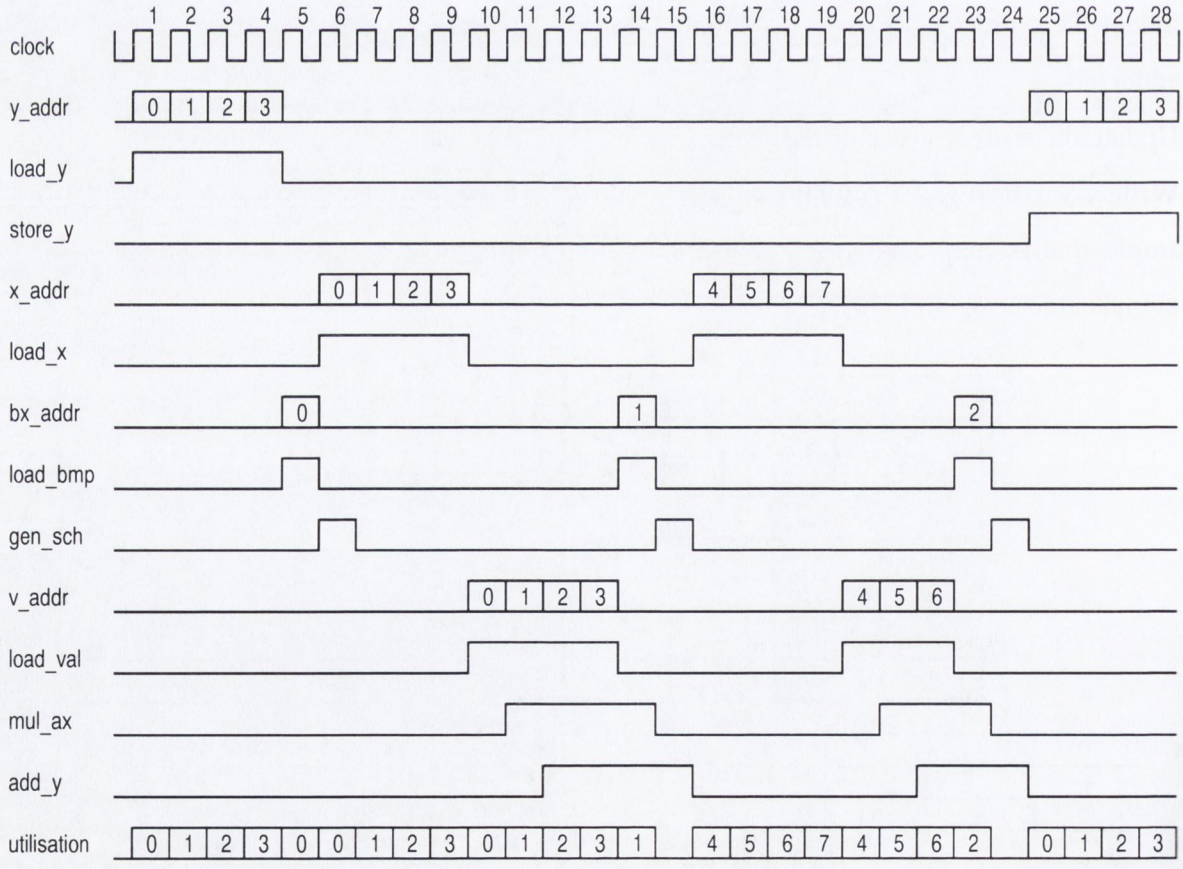



**Figure 7-14 Control Logic Timing-diagram**

The control-logic also contains logic to detect data-dependencies which can lead to RAW hazards and to stall the datapath until these dependencies have been resolved. Equally the

control-logic can halt (pause) the operation of the accelerator to wait for data from an external bus, data-cache or indeed external SDRAM.

All control-signals generated by the control-logic are designed to pipeline and to overlap operations which can be carried out simultaneously where possible, resulting in a high bus bandwidth utilisation of 26/28 cycles or 93% which is good. Realistically the bus utilisation which can be achieved will be lower than this once the long latencies of high-frequency floating-point units used in typical processor cores, are considered, and care will need to be taken in designing the pipelining scheme in order to maximise bus utilisation.

### 7.10.4 Memory Interface

The memory interface is controlled by the control-logic and increments the 4 address pointers and generates memory read and write signals in order to ensure all data required by the accelerator arrives in a timely manner from the appropriate addresses in memory or cache external to the accelerator and that the results generated by the accelerator are written back to the correct addresses in memory or cache external to the accelerator.

### 7.10.5 SMVM using Bitmap Schedule

The non-zero elements of A are multiplied by the corresponding elements of x which are looked up from a register using the column reference from the corresponding schedule entry. The elements of A are read from memory directly and multiplied as they enter the processor. There is no requirement to store the elements of the A sparse matrix in the case of Sparse Matrix by vector multiplication as the entries in A are only used once.

Not storing the elements of A in a register-file has several advantages compared with the state-of-the-art:

- Power and time (latency) associated with a write of a row of the A matrix to the register-file is saved
- Power and time (latency) associated with a read of a row of the A matrix from the register-file is saved
- Register-pressure associated with temporary storage of A matrix entries in the register-file is avoided

Storing the x-vector in a temporary register rather than a multi-ported register file has the advantage that the relatively higher power associated with a read of the x vector for each row of the A matrix to be multiplied is saved as a simple temporary register can be used to hold the entries of x.

The hardware required to perform the multiplication of the non-zero entries in A by the appropriate elements of the vector x stored in a local register is shown in **Figure 7-15**. In the figure A-dly denotes a delay to match the delay in clock-cycles through the floating-point adder, and M-dly denotes a delay to match that through the floating-point multiplier. These delays are required in order to line up the times at which the multiplexer selection signals arrive with the data arriving at the floating-point adder and multiplier.

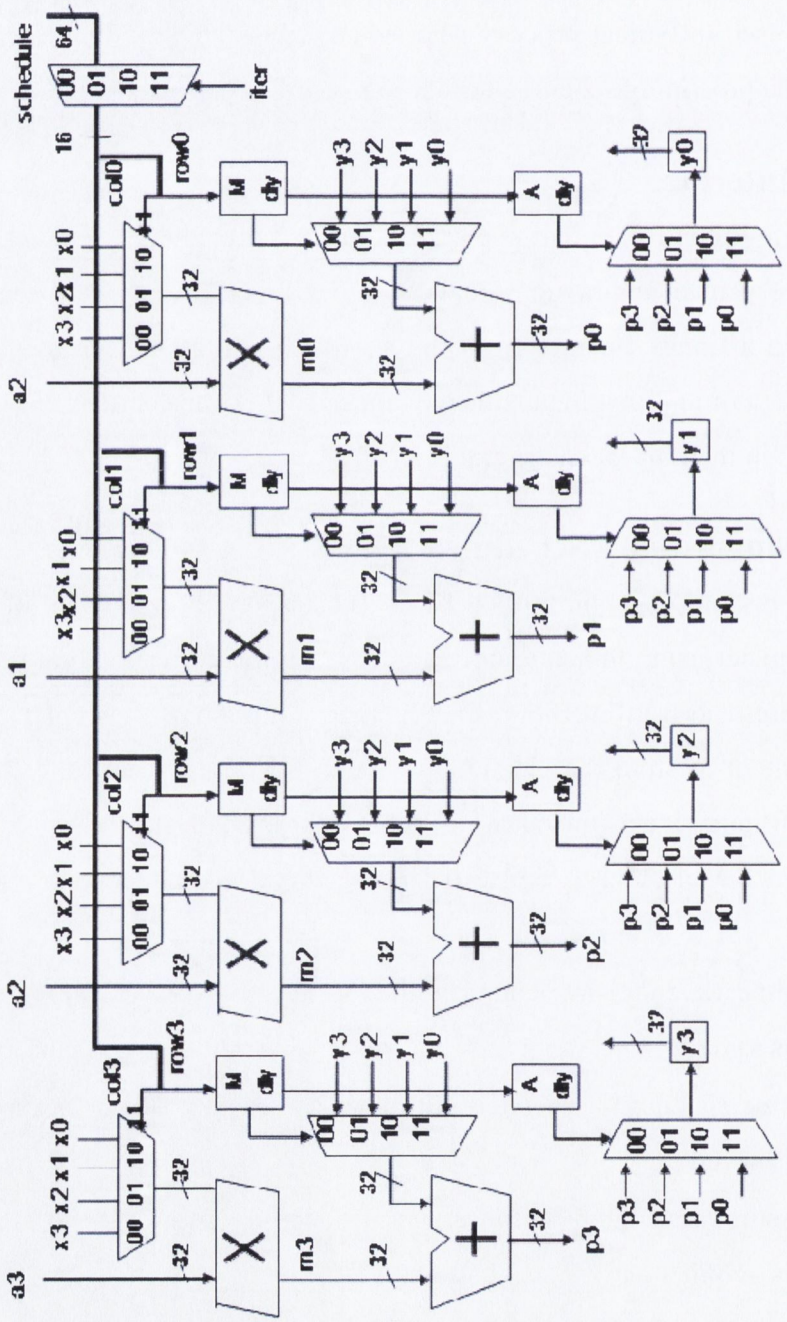



**Figure 7-15 Bitmap Controlled SMVM Datapath (128-bit SIMD)**

### 7.10.6 Hardware Requirements

The breakdown in terms of hardware gate complexity for the full hardware accelerator using a 128-bit SMID FPU consisting of four IEEE754 compliant FP adders and four multipliers is detailed in Table 7-4.

| Block | Gates | % total |
|---|---|---|
| Software Interface | 2688 | 3.35% |
| Scheduler | 1600 | 1.99% |
| Control-Logic | 3072 | 3.82% |
| Memory Interface | 1200 | 1.49% |
| Datapath | 71792 | 89.35% |
| Total | 80352 | 100.00% |

**Table 7-4 HW Accelerator Gate-Count**

The gate counts for the adders and multipliers are taken from a commercial data-sheet [150]. As can be seen the gate-count for the scheduler and all of the other logic accounts for around 10% of the total accelerator gate-count. This means that the 8.5k gates required for the accelerator logic outside the FPU accounts for only 12% additional logic when compared with what would be required to implement a SIMD FPU of the type implemented commonly in GPUs or the Intel SSE2 instruction-set [151].

## 7.11 Summary

In this section a hardware implementation for an accelerator to support bitmap blocked SMVM was outlined in detail, including functional description, logic implementation and gate-counts. As was demonstrated the overhead of implementing the proposed hardware is around 12% when compared with the SIMD FPU hardware already included in many x86 compatible processors. It was also shown that the impact can be further mitigated by incorporating the scheduling logic into the shifter unit already present in many commercial processors. Furthermore the possibility of just implementing the hardware matrix-vector multiply in an existing processor pipeline would leverage the existing memory interface, registers and allow the control logic to be implemented in software thus further reducing the hardware overhead while maximising flexibility.

# Chapter 8

*"Imagination is the beginning of creation.*
*You imagine what you desire,*
*you will what you imagine and at last you create what you will."*
*- George Bernard Shaw*

## 8 Conclusions

The reader has been guided through the world of complex applications [156] which depend on the solution of complex systems of equations represented by matrices for their answers. These applications are as diverse as can be imagined ranging from the design of aircraft, bridges and other structures to performing a Google search. In fact a key property of these problems in general and Google search, and of the latter in particular is the fact that the systems of equations and by extension the matrices that represent them contain far more zero coefficients than non-zeroes, i.e. they are sparse. In the case of Google the Google-Matrix has some 3 Billion entries on a side with an average of 6-7 non-zeroes per row/column.

Users and developers of large-scale applications based on sparse-matrices using SMVM as a principal kernel have found that the performance of computer systems is often extremely poor, often achieving less than 10% of manufacturers stated performance on this class of applications [26]. Unfortunately processor architectures and software methods which depend on them have advanced relatively little over recent decades and the major techniques have been in use for over 20 years with only iterative refinements, and often "the cure is worse than

the disease" with many apparent methods for speeding up calculations only offering a return if used for the equivalent of 10s or 100s of unoptimized SMVM operations.

Previous work by Vuduc [52] and others on Sparse Matrix-Vector Multiplication using the Block Compressed Sparse Row (BCSR) format has shown that many large matrices contain large amounts of zero-fill when tiled into locally dense tiles leading to lower than predicted performance and higher than predicted power dissipation due to the bandwidth necessary to fetch and process zeros in what the method assumes to be a dense sub-matrix (tile).

In partial answer to come of the performance limitations of existing blocked sparse storage formats two new sparse-storage formats are introduced in Chapter 6. The formats are both based on trivial techniques first identified by Richardson [138] in 1992. These methods have remained largely ignored in the intervening period as the solution identified by Richardson didn't address the fundamental underlying problem of limited memory bandwidth, which was highlighted by McKee [33] as the "Memory Wall". Essentially the method and hardware implementation outlined by Richardson adds to the difficulties the programmer and processor designer face by first fetching trivial data into the processor, consuming valuable bandwidth, before deciding that the data is not required and can by bypassed. Indeed having multiple parallel floating-point and trivial processing units makes the processor larger, slower and more difficult to program, negating many of the supposed benefits of trivial operand processing.

The key insight in this work is that by decoupling trivial operand detection from processing, a compression can be obtained. This compression increases effective memory bandwidth, and separately trivial operand processing can then be performed without the necessity for specialised hardware in the processor pipeline as proposed by Richardson [138]. It is also shown that the overhead of trivial operand detection and tagging has negligible cost which is dwarfed by the cost of assembling the data-structures required for large scale numerical applications, thus the benefits of the proposed accrue almost entirely to the whole application.

The performance of these methods is explored in detail in terms first experimentally using a suite of 50 large sparse matrices as a benchmark suite running on a typical engineering workstation with modern multicore processor, and the Bitmap Block Compressed Sparse Row (BBCSR) format was shown to perform better than BCSR or CSR reference methods in 7 out of 50 cases. Observing the shortcomings of BBCSR led to the development of the Scheduled Bitmap Block Compressed Sparse Row (SBCSR) format. This format was also duly benchmarked and analysed in both row-major and column-major scheduled variants.

The row-major SCBRS format was found to be faster than BCSR or CSR in 2 out of 50 cases, and the column-major format was found to be faster than either BCSR or CSR formats in 7 out of 50 cases. Finally the shortcomings of the software BBCSR and SBCSR formats were highlighted and the case for hardware acceleration was outlined. The implementation and functional model for the accelerator were described and the resultant hardware cost estimated.

## 8.1 Thesis Contributions

In this thesis a variety of techniques for accelerating Sparse Matrix computations are proposed and evaluated experimentally. It will be shown that the performance of the kernel Sparse Matrix Vector Multiplication (SMVM) operation, which dominates the execution time of iterative methods, can be improved dramatically compared to General Purpose Processors (GPP) and significantly when compared to Special Purpose Computers (SPC).

The specific improvements over the state-of-the-art proposed, which boost performance, proposed in this thesis are:

- A first Bitmap Block Compressed Sparse Row (BBCSR) sparse matrix storage method is proposed which eliminates the zero fill associated with the BCSR (Block Compressed Sparse Row) sparse matrix format
- Benchmarking on a 50 matrix set of large sparse matrices demonstrates a significant speed-up in 7/50 cases using the proposed BBCSR format, using a standard gcc compiler and Intel Xeon processor, when compared with CSR and BCSR formats
- A second sparse matrix format Scheduled Block Compressed Sparse Row (SBCSR) format is proposed which addresses the need to perform up to r*c bitmap comparisons (where r and c are respectively the number of rows and columns in the dense block sub-matrix) and branches performed to implement the BBCSR Sparse Matrix Vector Multiplication (SMVM)
- Again the SBCSR method is benchmarked against BBCSR, BCSR and BCSR methods for the same 50-matrix set, using the same configuration of gcc compiler, RHEL and Xeon processor
- A generic hardware accelerator is described which allows the SBCSR method to be utilised without penalty when compared with BCSR SMVM
- Integer sparse matrices such as the DCT coefficient matrices used in video applications are easily supported

- Finally the proposed hardware also allows compressed sparse data-structures to be random-accessed in situ without prior decompression, offering a major advantage over the state of the art

The work described carried out by the author at TCD and latterly at Movidius Ltd. has resulted in the following patent applications, the first of which has already been granted, and the remaining 3 of which are the subject of ongoing patent applications:

- Geraghty D., Moloney D., "Data processing system and method", US2009030960 (A1), Priority Date 2005-05-13
- Moloney D., "A processor", WO2009101119 (A1) - 2009-08-20, Priority Date 2008-02-11
- Moloney D., "A processor exploiting trivial arithmetic operations", EP2137610 (A1) - 2009-12-30, Priority Date 2007-03-15
- Moloney D., "A circuit for compressing data and a processor employing same", EP2137821 (A1) - 2009-12-30, Priority Date 2007-03-15

The same work has also contributed so far to the following publications:

- D. Moloney, D. Geraghty, C. McSweeney and C. McElroy, "Streaming Sparse Matrix Compression/Decompression", in Lecture Notes in Computer Science 2005 (HiPEAC Conference), Springer-Verlag, No. 3793, pp. 116-129
- D. Moloney, C. McSweeney, C. McElroy and D. Geraghty, "Hardware accelerator for finite element iterative methods", IEE Irish Signals and Systems Conference 2005, pp.330–337
- D. Gregg, C. McSweeney, 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.

## 8.2 Scope for Further Work

In the following sections a number of areas requiring further investigation following on from this work are outlined along with possible approaches where known.

### 8.2.1 Hardware Coprocessor Implementation

The hardware accelerator proposed in chapter 7 could be implemented on FPGA [152] and interfaced to external SDRAM in order to gauge the real speed-up that can be achieved by implementing the support for bitmap SMVM processing in hardware. In order to allow this to be achieved the C-code already outlined would have to be translated into a hardware

description language (HDL) such as Verilog [153], simulated and verified and then mapped to the target FPGA device, making judicious use of available components such as floating-point units and SDRAM controllers where available in order to minimise the effort and time involved.

### 8.2.2 Bitmap Hardware Integration in Existing Processor

As was indicated in section 7.11 the hardware accelerator proposed could be implemented as part of a processor pipeline in order to reduce the overhead. For instance in the x86 processor instruction set architecture (ISA) complex instructions are already translated into a series of simpler uOPs (micro-operations) as detailed in [152]. Thus the accelerator functionality could be implemented within the ptlsim simulator [155] as part of the x86 processor pipeline. This would allow the performance benefits of the SBCSR technique to be measured over a large number of existing applications. Furthermore the same instrumented version of ptlsim could also be enhanced to recognise sequences of binary code which could benefit from the proposed technique using dynamic binary instrumentation. The latter approach would allow existing x86 binaries to be evaluated for potential performance benefits.

### 8.2.3 SMVM Tuning

The BBCSR and SBCSR sparse storage formats described in Chapter 6 can be advantageous but are obviously not optimal in terms of performance for all matrices, i.e. their usefulness depends on the non-zero pattern in the underlying sparse matrix. The experiments conducted in the same chapter focussed on exploring a large range of possible tilings exhaustively in order to make a fair comparison between CSR, BCSR and the proposed methods. In practice this exhaustive approach is wasteful of compute resources and time. A better approach would be to expand on the approach used by Vuduc [52] where the amount of fill in the sparse matrix in target sparse storage formats such as CSR and BCSR is estimated by sampling a portion of the sparse matrix rather than the whole, to include the proposed storage schemes. According to Vuduc tilings yielded by the sampling approach yields results that are on average within 10% of the fastest matrix vector multiplication times obtained by exhaustive search. This being said even using the sampling approach is grossly inefficient with 20-40 SMVM times being expended to estimate fill meaning that the payback for tiled schemes is most likely to occur where the same matrix is used to compute a number of SMVM products greater than the time taken to estimate fill.

### 8.2.4 Hybrid Sparse Matrix Storage Formats

An interesting area for further work would be to combine BBCSR and SBCSR together with BCSR formats into a single data-structure with a single hybrid software SMVM method. In the proposed method and data-structure individual tiles would be stored as BBCSR or SBCSR tiles where BCSR fill is high and as native BCSR where the fill is low. The author believes such a hybrid format could offer optimal performance for a wide range of matrices and local fill patterns could be tuned for on a tile-by-tile basis rather than averaged basis used by Vuduc.

For example the code shown in Listing 8-1 shows how a Hybrid BCSR (HBCSR) could accommodate dense blocks where the BCSR format is an exact match for the underlying tile, i.e. contains no zero fill, while blocks containing fill could be multiplied using a bitmap as in BBCSR. Alternately a schedule could be generated and the blocks containing fill could be multiplied according to the schedule generated from the non-zero tile elements or the bitmap.

```
L1.   void hbcsr_smvm4x4(int bm, int r, int c, int *row_start, int
   *col_idx, Type *value, Type *src, Type *dest) {
L2.     int i, j, _nz, bitmap, nz;
L3.     Type y0, y1, y2, y3, x0, x1, x2, x3;
L4.     long long tag; // 64-bit integer
L5.
L6.     for (i=0; i<bm; i++, dest+=r) { // block-
L7.       y0 = dest[0];
L8.       y1 = dest[1];
L9.       y2 = dest[2];
L10.      y3 = dest[3];
L11.      for (j=row_start[i]; j<row_start[i+1]; j++, col_idx++,
   value+=nz) {
L12.        // first entry in value[] related to block is a tag
L13.        _nz = 0;
L14.        tag = memcpy(value,&tag,8); // copy to 64-bit int
L15.        value++; // advance pointer to skip bitmap entry
L16.        x0 = src[(*col_idx)    ];
L17.        x1 = src[(*col_idx) + 1];
L18.        x2 = src[(*col_idx) + 2];
L19.        x3 = src[(*col_idx) + 3];
L20.        if (tag==0.0) { // 0.0 means block is dense
L21.          y0 += value[ 0] * x0; // row 0
L22.          y0 += value[ 1] * x1;
L23.          y0 += value[ 2] * x2;
L24.          y0 += value[ 3] * x3;
L25.          y1 += value[ 4] * x0; // row 1
L26.          y1 += value[ 5] * x1;
L27.          y1 += value[ 6] * x2;
L28.          y1 += value[ 7] * x3;
L29.          y2 += value[ 8] * x0; // row 2
L30.          y2 += value[ 9] * x1;
L31.          y2 += value[10] * x2;
L32.          y2 += value[11] * x3;
L33.          y3 += value[12] * x0; // row 3
```

```
L34.          y3 += value[13] * x1;
L35.          y3 += value[14] * x2;
L36.          y3 += value[15] * x3;
L37.          nz=16; // advance value pointer by r*c
L38.        }
L39.        else { // block contains fill so bitmap SMVM
L40.          bitmap = tag     & 0x000000000000FFFF; // 4x4 bitmap
L41.          nz     = tag>>16 & 0x000000000000FFFF; // NZ for 4x4
L42.          if (bitmap & 32768) y0 += value[_nz++] * x0; //row 0
L43.          if (bitmap & 16384) y0 += value[_nz++] * x1;
L44.          if (bitmap &  8192) y0 += value[_nz++] * x2;
L45.          if (bitmap &  4096) y0 += value[_nz++] * x3;
L46.          if (bitmap &  2048) y1 += value[_nz++] * x0; //row 1
L47.          if (bitmap &  1024) y1 += value[_nz++] * x1;
L48.          if (bitmap &   512) y1 += value[_nz++] * x2;
L49.          if (bitmap &   256) y1 += value[_nz++] * x3;
L50.          if (bitmap &   128) y2 += value[_nz++] * x0; //row 2
L51.          if (bitmap &    64) y2 += value[_nz++] * x1;
L52.          if (bitmap &    32) y2 += value[_nz++] * x2;
L53.          if (bitmap &    16) y2 += value[_nz++] * x3;
L54.          if (bitmap &     8) y3 += value[_nz++] * x0; //row 3
L55.          if (bitmap &     4) y3 += value[_nz++] * x1;
L56.          if (bitmap &     2) y3 += value[_nz++] * x2;
L57.          if (bitmap &     1) y3 += value[_nz++] * x3;
L58.        }
L59.      }
L60.      dest[0] = y0;
L61.      dest[1] = y1;
L62.      dest[2] = y2;
L63.      dest[3] = y3;
L64.    }
L65.     } // hbcsr_smvm4x4()
```

**Listing 8-1 HBCSR 4x4 SMVM C-Code**

In SBCSR and hybrid formats containing SBCSR tiles considerable further work could be done on tuning SBCSR schedules, for each possible bitmap pattern, on each of the target processor architectures. Additionally the author believes there is considerable merit in exploring zigzag variants of BBCSR, SBCSR and hybrid formats, along the lines proposed in [135]. The bitmap SMVM formats could also be useful for managing variable block sizes and unaligned BCSR (UBCSR) [52] in a general hybrid format.

### 8.2.5 Extended Scope for Trivial Operand Processing

This author notes that although not mentioned by Richardson or Lilja it is possible that trivial multiplication could be extended to the general case of multiplication by powers of 2 resulting in a small unit which adds exponents and leaves the mantissa of the multiplicand unmodified in a manner similar to that proposed in [139]. It would be interesting to profile matrices to see what proportion of the data falls into this category and to see how different floating-point units with differing latencies can be accommodated as part of the bitmap scheduling process.

### 8.2.6 Quantifying Power dissipation

Although the suspicion is that BBCSR and SBCSR formats reduce power-dissipation because they eliminate both the needless fetching of zero-fill values from memory as well as trivial operations utilising them, no experimental evidence has been gathered in this work to prove it. It would therefore be interesting to investigate the power-dissipation of the proposed methods on power-dissipation and correlate this information with fill or some other metric so a tuning method along the lines of that proposed by Vuduc can tune for lowest power, or power-performance rather than just performance as at present.

### 8.2.7 Scalability

A key issue with engineering problems is that they scale to meet and exceed the available computing resources. This scaling has implications for both the numerical precision required and the address-space necessary to access the data. In the former case while double-precision is the standard for numerical computations today, 80-bit extended precision is already in common use and supported by a variety of computing platforms including the x86 (Intel and AMD) ISA. In fact the recently ratified IEEE754-2008 standard [78] for floating-point arithmetic provides for binary and decimal 128-bit formats, and 256-bit arithmetic is used in physics and computational chemistry applications [157]. Indeed a recent review by Bailey [158] identifies a range of numerical applications requiring 128-bit or higher precision. Bitmap compression would achieve 1:128 or better for zero-fill in these applications. Similarly while 32-bit is the current standard for addresses, investigation into 64-bit SMVM has already begun [142]. It would be interesting to investigate the benefits of bitmap compression for these applications.

### 8.2.8 Lookahead Bitmap Scheduling

A further refinement of the scheduler would be to perform look-ahead in order to resolve dependencies. In principle if a schedule can be generated in one clock-cycle and the resultant SMVM takes NZ cycles, the scheduler can look ahead at the next N bitmaps to evaluate whether data dependencies and associated RAW hazards can be eliminated.

As can be seen in the following example shown in Figure 8-1 if scheduling is performed independently on a bitmap at a time, dependencies and associated RAW hazards occur in association with the summation of y[1] as each element of row 1 in the tile is non-zero. If this schedule were processed a stall would occur in association with each addition of the partial products to y[1]. The solution depicted in the same figure is to compute the schedules for 2 bitmaps within the same matrix row dependently, looking ahead to see which slots in the

second bitmap schedule can be interleaved with those from the first bitmap schedule in order to remove dependencies. This Lookahead scheduling can be extended to further bitmaps on the same basis as shown in order to accommodate floating-point adders with progressively higher latencies which cause proportionately higher stall penalties if not resolved.

bitmap1 0x0F00

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |
| 2 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 |

bitmap2 0x5448

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 0 | 1 | |
| 0 | 1 | 0 | 0 | |
| 0 | 1 | 0 | 0 | |
| 1 | 0 | 0 | 0 | |

independent scheduling

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| sch (bm1) | col | 0 | 1 | 2 | 3 | | | | | |
| | row | 1 | 1 | 1 | 1 | | | | | |
| sch (bm2) | col | | | | | | 0 | 1 | 1 | 3 |
| | row | | | | | | 3 | 0 | 1 | 2 | 0 |

lookahead scheduling

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| sch (bm1) | col | 0 | | 1 | | 2 | | 3 | | |
| | row | 1 | | 1 | | 1 | | 1 | | |
| sch (bm2) | col | | 0 | | 1 | | 1 | | 3 | 1 |
| | row | | 3 | | 0 | | 2 | | 0 | 1 |

**Figure 8-1 Lookahead Bitmap Scheduling**

The only disadvantages with extended bitmap scheduling are:

- Initial start-up latency of N cycles where N stages of Lookahead are used in bitmap scheduling
- Additional complexity in terms of registers to hold partial products whose addition to the y-value is deferred because a RAW hazard would otherwise arise.

### 8.2.9 Further Experimental SMVM Benchmarking

It would make sense to extend this work to benchmark the proposed sparse matrix formats and SMVM methods across multiple processor architectures and variants of the same CPU containing different numbers of cores, FSB speeds, cache sizes etc. Furthermore on multicore processors it would potentially increase performance if decompression code were run as a thread on a separate core in a CMP. Thus one core would decode bitmaps and prepare schedules which the other core would elaborate. It would also be interesting to investigate the utility of bitmaps for other linear-algebra operations such as matrix-matrix products.

### 8.2.10 SMVM Benchmark suites

The lack of availability of libraries and lack of standard benchmark suites of sparse matrices means that researchers in the field are forced in the first case to recreate the results of others, and in the second can "cherry-pick" results to prove their point. The field in general would benefit from a repository for SMVM codes and standardised test-suites. Test-suites can be drawn from repositories such as MatrixMarket [11], UF Sparse Matrix Collection [6] or Parasol [12]. Reliance on these matrices alone as "canned problems" is felt to be insufficient by the author in that they may not contain sufficient data to gather statistically useful results. Furthermore reading in very large sparse matrices in text format and converting them to tiled data-structures is very time-consuming and can take tens to hundreds of times longer than SMVM products.

A more useful approach would be to synthesise sparse data-structures algorithmically which have the desired statistical properties in terms of:

- number of non-zeroes per row/column
- particular block-structure or tile size
- particular non-zero density or pattern within blocks/tiles

Such a library would allow whole populations of matrices with particular properties to be created and evaluated algorithmically in a platform independent and reproducible manner allowing Sparse Matrix storage schemes and associated Sparse Matrix vector multiplication to be more thoroughly explored than with a fixed Sparse Matrix suite. A useful feature in such a library would be to extract algorithmic descriptions of sparse matrices from existing repositories such as MatrixMarket allowing them to be described in a concise manner, and used as a basis for the generation of new matrices with similar properties.

### 8.2.11 Other Uses for Bitmaps

An interesting extension of the work presented here is that the bitmap hardware can easily be extended to allow random access to compressed structures or hybrid structures in memory, without first having to decompress them. A representative hybrid data-structure is shown in Figure 8-2 and consists of a 3D scaling matrix which is around 50% sparse along with three dense 4-element vectors, a pointer to the next structure in memory and a 32-bit bitmap associated with that next structure element. As can be seen in Figure 8-3, random access to any 4-element compressed or uncompressed element of the hybrid data-structure in memory can be readily achieved by employing a chain of simple adders and some additional logic to mask out the relevant sections. Assuming that each sub-element in a 4-element sub-structure

is the same length the bitmap bits can be summed directly to form an offset which is added to the base address of the hybrid data-structure, thus forming the address of the required sub-element of the structure. The proposed scheme is fully transparent to the memory hierarchy of the processor advancing beyond the compression schemes surveyed in Chapter 4.

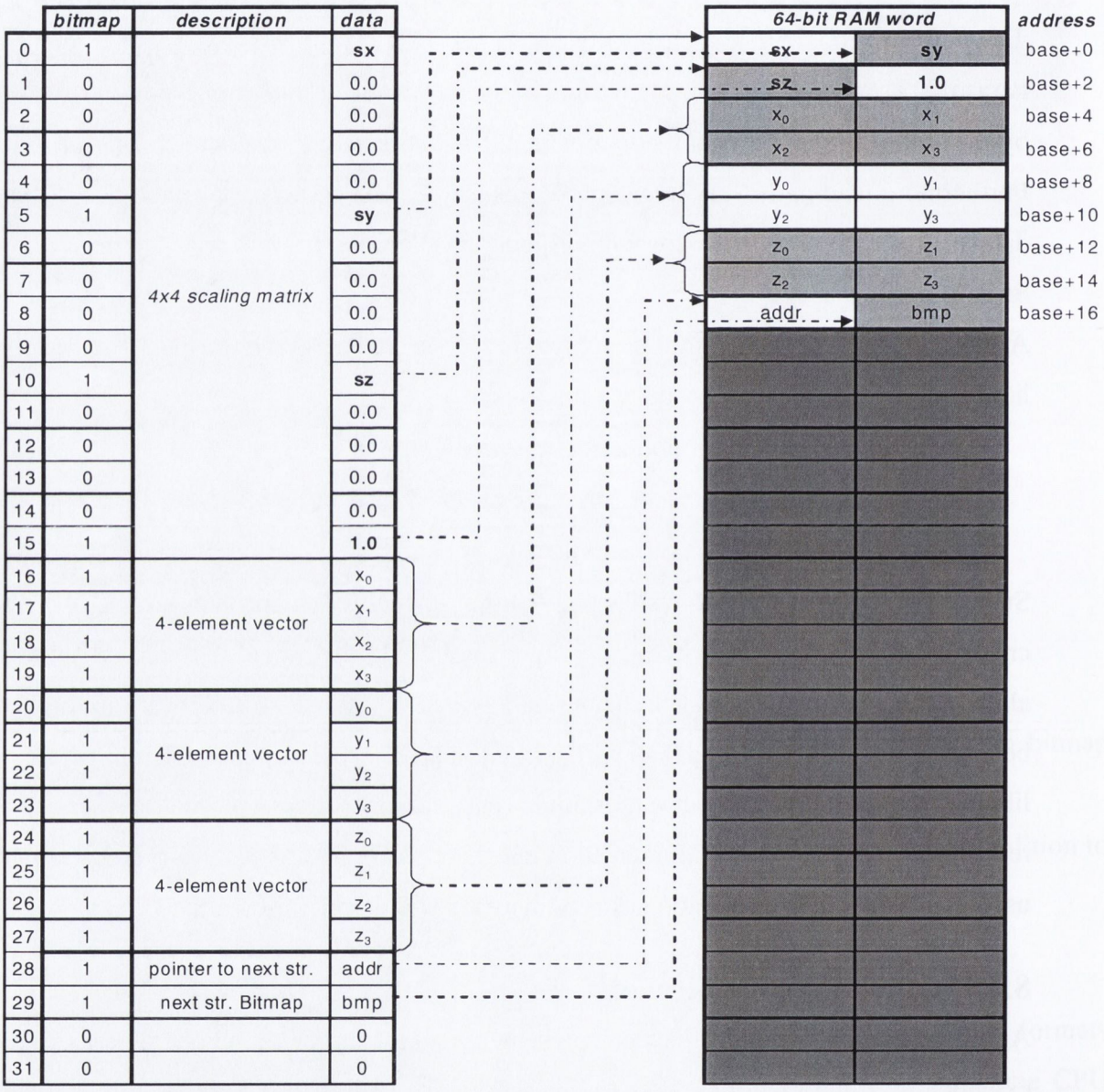



**Figure 8-2 Random Access to Sparse Data-Structures**

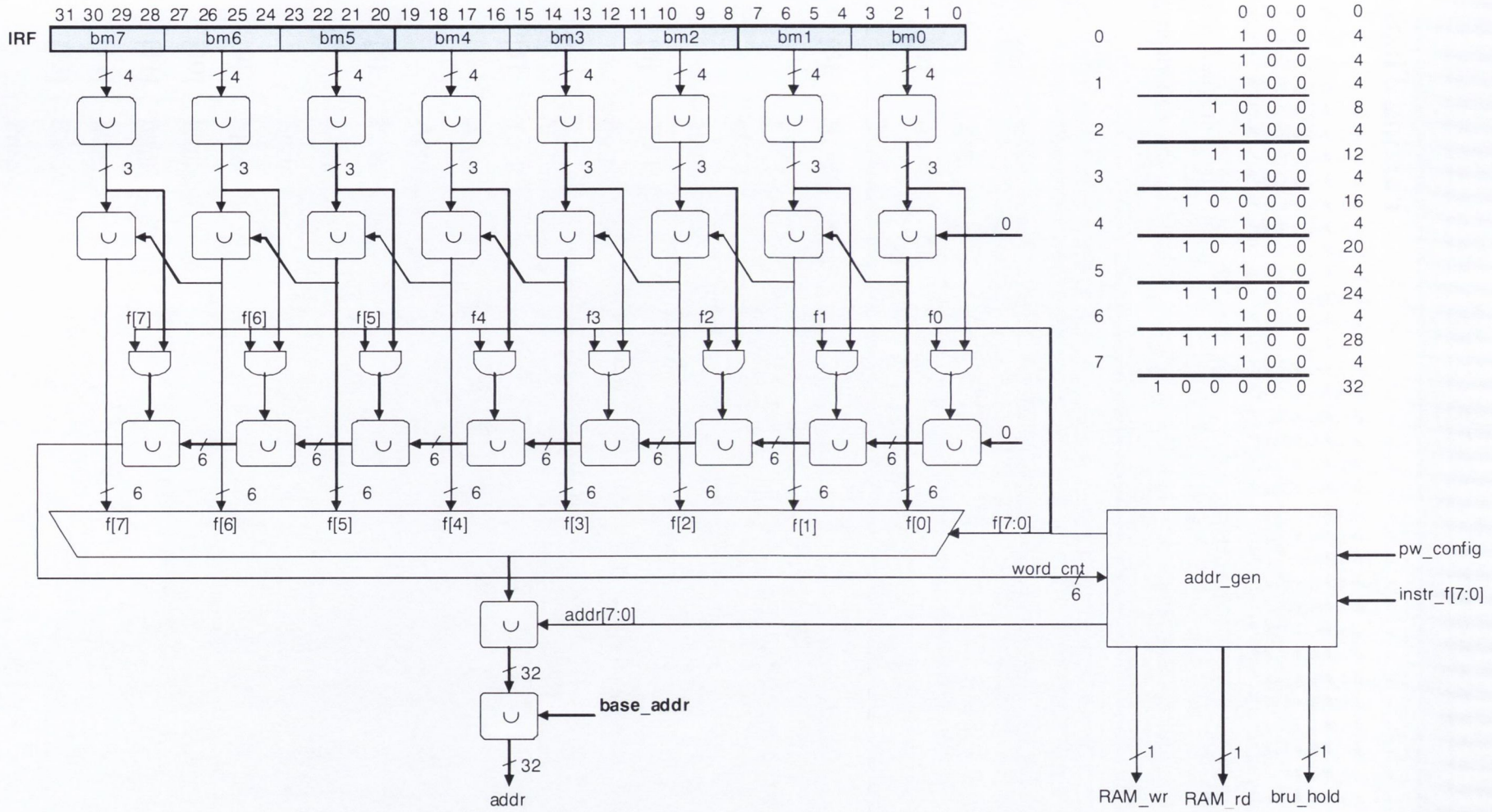



**Figure 8-3 Random-Access Bitmap Addressing**

## Parting Thought

The work described here is the result of 6 years of on and off effort, much of which was not reported here as it was not considered (by the author) to merit inclusion. It has been a voyage of personal and scientific discovery with many roads taken and subsequently back-tracked when a new approach was required and brings to mind "The Road not Taken" by the American poet Robert Frost from his "Mountain Interval" collection in 1916.

Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveller, long I stood
And looked down one as far as I could
To where it bent in the undergrowth;

Then took the other, as just as fair
And having perhaps the better claim,
Because it was grassy and wanted wear;
Though as for that, the passing there
Had worn them really about the same,

And both that morning equally lay
In leaves no step had trodden black.
Oh, I kept the first for another day!
Yet knowing how way leads on to way,
I doubted if I should ever come back.

I shall be telling this with a sigh
Somewhere ages and ages hence:
two roads diverged in a wood, and I --
I took the one less travelled by,
And that has made all the difference.

# Bibliography


[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams and K. A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley", EECS Department, UC Berkeley, Technical Report No. UCB/EECS-2006-183, 2006

[2] J. Shalf, "The new landscape of parallel computer architecture", Journal of Physics: Conference Series, 2007, Vol. 78, pp. 1-15

[3] Y. Saad, "Iterative Methods for Sparse Linear Systems - Second Edition", SIAM, 2003, ISBN 0-89871-534-2

[4] IEEE Standards Board, "IEEE Standard for Binary Floating-Point Arithmetic", Technical Report ANSI/IEEE Std. 754-1985, IEEE, New York, 1985

[5] 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, Philadelphia, PA

[6] T.J.R Hughes, I. Levit and J. Winget, "Element-by-Element Implicit Algorithms for Heat Conduction", Journal of the Engineering Mechanics Division, ASCE, 109 (1983), pp.576-585

[7] T.J.R Hughes, I. Levit and J. Winget, "An Element-by-Element Solution Algorithm for Problems of Structural and Solid Mechanics", Computer Methods in Applied Mechanics and Engineering, 36 (1983), pp.241-254

[8] P. Wesseling and P. Sonneveld, "Numerical Experiments with a Multiple Grid and a Preconditioned Lanczos Type Method", Lecture Notes in Mathematics 771, Springer-Verlag, Berlin, Heidelberg, New York, pp. 543–562, 1980.

[9] http://www2.cs.cas.cz/harrachov/slides/Gijzen.pdf (accessed 08/04/2010)

[10] http://www.cise.ufl.edu/research/sparse/ (accessed 08/04/2010)

[11] http://math.nist.gov/MatrixMarket/ (accessed 08/04/2010)

[12] http://www.parallab.uib.no/projects/parasol/data/ (accessed 08/04/2010)

[13] Kevin R. Gee, "Using latent semantic indexing to filter spam", SAC '03: Proceedings of the 2003 ACM symposium on Applied computing, pp.460-464

[14] N. Muller, L. Magaia, B. M. Herbst, "Singular Value Decomposition, Eigenfaces, and 3D Reconstructions", SIAM Review, Vol. 46, No. 3, pp. 518-545

[15] P. Kogge, "The Architecture of Pipelined Computers", McGraw-Hill, ISBN-0-07-035237-2, 1981

[16] O. C. Zienkiewicz and R L. Taylor, "The finite element method. Vol. I. Basic formulations and linear problems", London: McGraw-Hill, 1989.

[17] L. Oliker, X. Li, P. Husbands, R. Biswas, "Effects of Ordering Strategies and Programming Paradigms on Sparse Matrix Computations", SIAM Review, Vol. 44, No. 3, pp.373-393

[18] G.P. Nikishkov, "Introduction to the Finite Element Method", Lecture Notes, University of Aizu, Aizu-Wakamatsu 965-8580, Japan http://web-ext.u-aizu.ac.jp/~niki/feminstr/feminstr.html (accessed 08/04/2010)

[19] R. Courant, "Variational methods for the solution of problems of equilibrium and vibrations," Bull. Amer. Math. Soc., 49 (1943), pp. 1–23.

[20] R.W. Clough, "The finite element method in plane stress analysis", Proceedings of the Second ASCE Conference on Electronic Computation, 1960

[21] F.T. Johnson, E.N. Tinoco and N.J. Yu, "Thirty Years of Development and Application of CFD at Boeing Commercial Airplanes, Seattle", AIAA paper 2003-3439, 2003, pp.1-24

[22] http://web.mit.edu/course/16/16.810/www/16.810_L4_CAE.pdf (accessed 08/04/2010)

[23] Taylor V.E.; Ranade A.; Messerschmitt D.G, "SPAR: a new architecture for large finite element computations", IEEE Trans. on Computers, Vol. 44, No. 4, April 1995, pp.531–545

[24] http://web-ext.u-aizu.ac.jp/~niki/javaappl/jassem/jassem.html (accessed 08/04/2010)

[25] http://www.netlib.org/blas/ (accessed 08/04/2010)

[26] W. Anderson, W. Gropp, D. Kaushik, D. Keyes & B. Smith, "Achieving high sustained performance in an unstructured mesh CFD application", Conference on High Performance Networking and Computing, Portland, Oregon, USA, 1999, pp.1-11

[27] W.D. Smith and A.R. Schnore. "Towards an RCC-Based Accelerator for Computational Fluid Dynamics Applications", Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms, June 23 - 26, 2003, Las Vegas, Nevada, USA, pp.222-234

[28] I. S. Duff, A. M. Erisman and J. K. Reid, "Direct Methods for Sparse Matrices", Oxford University Press, London, 1986

[29] I. Duff, R. Grimes, J. Lewis, "Sparse Matrix Test Problems", ACM Transactions on Mathematical Software, Volume 15, March 1989, pp.1-14

[30] J. Koster, "Parallel templates for numerical linear algebra, a high-performance computation library", MSc. Thesis, Dept. of Mathematics, Utrecht University, July 2002

[31] M. Silva, R. Wait, "Sparse matrix storage revisited". In Proceedings of the 2nd Conference on Computing Frontiers (Ischia, Italy, May 04 - 06, 2005.), CF '05. ACM Press, New York, NY, pp.230-235

[32] P.T. Stathis, S. Vassiliadis, S. D. Cotofana, "A Hierarchical Sparse Matrix Storage Format for Vector Processors", Proceedings of IPDPS 2003, pp. 61a, Nice, France, April 2003

[33] McKee, S. A., "Reflections on the memory wall", In Proceedings of the 1st Conference on Computing Frontiers (Ischia, Italy, April 14 - 16, 2004). CF '04. ACM Press, New York, NY, pp.162-168

[34] J. Hennessy, D. Patterson, "Computer Architecture A Quantative Approach, 4th Edition", Morgan-Kaufmann, ISBN 978-0123704900, September 27, 2006

[35] Goddeke, D. and Strzodka, R. and Turek, S., "Accelerating Double Precision FEM Simulations with GPUs", Proceedings of ASIM 2005 - 18th Symposium on Simulation Technique, Erlangen, Germany, 2005, pp.139-144

[36] http://www.realworldtech.com/page.cfm?ArticleID=RWT081502231107&p=1 (accessed 08/04/2010)

[37] Hrishikesh, M. S., Burger, D., Jouppi, N. P., Keckler, S. W., Farkas, K. I., and Shivakumar, P. 2002. "The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays", Proceedings of the 29th Annual international Symposium on Computer Architecture (Anchorage, Alaska, May 25 - 29, 2002), pp.14-24

[38] M.A. Horowitz, R.Ho, K.W. Mai, "The Future of Wires", Proceedings of the IEEE, 2001, pp. 490-504

[39] R. Tanabe, Y. Ashizawa, H. Oka, "CMOS Scaling Analysis Based on ITRS Roadmap by Three-Dimensional Mixed-Mode Device Simulation", Proc. of SISPAD2004, pp.303-306

[40] Sprangle, E. and Carmean, "Increasing processor performance by implementing deeper pipelines", Proceedings of the 29th Annual international Symposium on Computer Architecture 2002, pp.25-34.

[41] J. Bobba, M. Moravan and U. Saeed, "TAP: Taxonomy for Adaptive Pre-fetching", University of Wisconsin, Madison http://pages.cs.wisc.edu/~moravan/tap.pdf (accessed 08/04/2010)

[42] Wulf W., and McKee S., "Hitting the memory wall: Implications of the obvious", Computer Architecture News, 23(1), pp.20-24, 1994

[43] https://computation.llnl.gov/casc/sc2001_fliers/MemWall/MemWall01.html (accessed 08/04/2010)

[44] B. Jacob, "A case for studying DRAM issues at the system level", IEEE Micro, Vol. 23 , No. 4, July-Aug. 2003, pp.44-56

[45] J. D. McCalpin, "Memory Bandwidth and Machine Balance in Current High Performance Computers", Technical Committee on Computer Architecture (TCCA) Newsletter, IEEE Computer Society, December 1995

[46] J. L. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millennium", Computer, v.33 n.7, July 2000, pp.28-35

[47] K. Asanovic, "*Vector Microprocessors*", PhD Thesis, UC Berkeley, 1987

[48] G. Griem, L. Oliker, J. Shalf, and K. Yelick, "Identifying Performance Bottlenecks on Modern Microarchitectures using an Adaptable Probe", IPDPS'04 - Workshop 14 , pp. 255-263

[49] O. Temam and W. Jalby. "Characterizing the behaviour of sparse algorithms on caches", Proceedings of Supercomputing '92, 1992, pp.578-587

[50] J. Feldman, C. Retter, "Computer Architecture: A Designer's Text Based on a Generic RISC", McGraw-Hill, ISBN 0-07-113318-6, 1994

[51] Todd C. Mowry, "Tolerating Latency Through Software-Controlled Data Prefetching", Ph.D. thesis, Stanford University, Computer Systems Laboratory, March 1994

[52] R. Vuduc, "Automatic performance tuning of sparse matrix kernels", PhD thesis, UC Berkeley, California, USA, December 2003

[53] Allan, A.; Edenfeld, D.; Joyner, W.H., Jr.; Kahng, A.B.; Rodgers, M.; Zorian, Y.; "2001 technology roadmap for semiconductors", IEEE Computer, Volume 35, Issue 1, Jan. 2002, pp.42–53

[54] Hammerstrom, D. W. and Davidson, E. S. 1977. "Information content of CPU memory referencing behaviour". In *Proceedings of the 4th Annual Symposium on Computer Architecture* (March 23 - 25, 1977). ISCA '77. ACM Press, New York, NY, pp.184-192

[55] Park, A. and Farrens, "Address compression through base register caching", In Proceedings of the 23rd Annual Workshop and Symposium on Microprogramming and Microarchitecture (Orlando, Florida, United States, November 27 - 29, 1990). International Symposium on Microarchitecture. IEEE Computer Society Press, Los Alamitos, CA, pp.193-199

[56] J. Liu, K. Sundaresan, N. R. Mahapatra. "Dynamic Address Compression Schemes: A Performance, Energy, and Cost Study", 2004 IEEE International Conference on Computer Design (ICCD'04), 2004, pp. 458-463

[57] Suresh, D. C., Agrawal, B., Yang, J., and Najjar, W. 2005. "A tunable bus encoder for off-chip data buses". In *Proceedings of the 2005 international Symposium on Low Power Electronics and Design* (San Diego, CA, USA, August 08 - 10, 2005). ISLPED '05. ACM Press, New York, NY, pp.319-322

[58] D. Citron. "Exploiting Low Entropy to Reduce Wire Delay.", Computer Architecture Letters, Volume 3, Jan. 2004, pp.1

[59] E. G. Hallnor and S. K. Reinhardt, "A Unified Compressed Memory Hierarchy", Proc. 11th International Symposium on High Performance Computer Architecture, HPCA-11, 2005, pp.201-212

[60] B. Abali, H. Franke, S. Xiaowei et. al, "Performance of Hardware Compressed Main Memory", Proc. 7th Int'l Symposium On High Performance Computer Architecture, 2001, pp.73-81

[61] "Desktop Performance and Optimization for Intel® Pentium® 4 Processor", Intel Corporation, Order number: 249438-01, Feb 2001

[62] S. Rixner, "Stream Processor Architecture", Kluwer Academic Publishers, 2002

[63] B. Flachs, S. Asano, S.H. Dhong, P. Hofstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H. Oh, S. M. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, N. Yano, "A Streaming Processing Unit for a CELL Processor", ISSCC 2005, pp.134-135

[64] http://www.fftw.org/cell/ (accessed 13 April 2010)

[65] http://www.jatit.org/volumes/research-papers/Vol7No2/6Vol7No2.pdf (accessed 08/04/2010)

[66] Manuel Ujaldon, Joel H. Saltz, "Exploiting parallelism on irregular applications using the GPU", Proceedings of the 2005 International Parallel Computing Conference *(*PARCO'05*)*, 2005, pp.639-646

[67] http://www.khronos.org/opencl/ (accessed 08/04/2010)

[68] http://www.nvidia.com/object/cuda_home_new.html (accessed 08/04/2010)

[69] http://www.amd.com/us/products/technologies/stream-technology/Pages/stream-technology.aspx (accessed 23 April 2010)

[70] http://graphics.stanford.edu/projects/brookgpu/ (accessed 08/04/2010)

[71] www.stanford.edu/class/ee380/Abstracts/070926-**PeakStream**.pdf (accessed 23 April 2010)

[72] http://software.intel.com/en-us/data-parallel/ (accessed 23 April 2010)

[73] http://groups.csail.mit.edu/cag/streamit/ (accessed 08/04/2010)

[74] http://www.opengl.org (accessed 08/04/2010)

[75] http://www.khronos.org/ (accessed 23 April 2010)

[76] http://msdn.microsoft.com/en-us/directx/default.aspx (accessed 08/04/2010)

[77] http://www.nvidia.com/object/personal_supercomputing.html (accessed 23 April 2010)

[78] IEEE, "754-2008 IEEE Standard for Floating-Point Arithmetic", Aug. 29 2008, ISBN 978-0-7381-5753-5

[79] http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf (accessed 23 April 2010)

[80] N. Jayasena, W. J. Dally, "Streams and Vectors: A Memory System Perspective", Proc. MSP-6 Workshop on Media and Streaming Processors and DSPs, 2004

[81] D. Geer, "Chip Makers Turn to Multicore Processors", IEEE Computer Magazine, May 2005, pp. 11-13

[82] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, K. Chung, "The Case for a Single-Chip Multiprocessor", Proc. 7th Int. Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII), 1996, pp.2-11

[83] M. Gschwind, P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, T. Yamazaki, "Synergistic processing in Cell's multicore architecture", IEEE Micro, March 2006, pp.10-24

[84] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan and P. Hanrahan, "Larrabee: A Many-Core x86 Architecture for Visual Computing", ACM Trans. On Graphics, Vol.25, No. 3, Aug. 2008, pp.2-15

[85] http://techresearch.intel.com/articles/Tera-Scale/1826.htm (accessed 23 April 2010)

[86] W. J. Dally, J. Balfour, D. Black-Shaffer, J. Chen, R. C. Harting, V. Parikh, J. Park, D. Sheffield, "Efficient Embedded Computing", IEEE Computer, July 2008, pp.27-33

[87] L. Spracklen & S. G. Abraham, "Chip Multithreading: Opportunities and Challenges", Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA-11), 2005, pp.248-252

[88] S. Heo, K. Barr, and K. Asanovic, "Reducing power density through activity migration", Proceedings of the International Symposium on Low Power Electronics and Design, 2003, pp.217-222

[89] P. Michaud, "Exploiting the Cache Capacity of a Single-Chip Multi-Core Processor with Execution Migration", HPCA-10,Proceedings, 2004, pp.186-195

[90] P. Kongetira, K. Aingaran, K. Olukotun, "Niagara: a 32-way Multithreaded SPARC processor", IEEE Micro, Mar-Apr 2005, pp.21-29

[91] S. Gochman et al., "Intel Pentium-M Processor: Microarchitecture and Performance", Intel Technology Journal, Vol. 7, No. 2, 2003, pp.22-36

[92] M. Curtis-Maury, X. Ding, C. D. Antonopoulos, D. S. Nikolopoulos , "An Evaluation of OpenMP on Current and Emerging Multithreaded/Multicore Processors", Proceedings of IWOMP05, pp.133-144

[93] Oliker, L.; Canning, A.; Carter, J.; Shalf, J.; Ethier, S.; "Scientific Computations on Modern Parallel Vector Systems", Proceedings SC2004, 2004, pp.10-20

[94] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonté, J.H. Ahn, N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, I. Buck, "Merrimac: Supercomputing with Streams", SC2003, November 2003, Phoenix, Arizona, pp.35-43

[95] http://www.clearspeed.com (accessed 08/04/2010)

[96] http://www.wired.com/gadgetlab/2009/08/personal-supercomputers/ (accessed 23 April 2010)

[97] http://www.geeks.co.uk/11345-university-of-antwerp-crams-12-teraflop-speeds-into-a-desktop (accessed 23 April 2010)

[98] P. Macioł and K. Banas, "Testing Tesla Architecture for Scientific Computing: the Performance of Matrix-Vector Product", Proceedings of the International Multiconference on Computer Science and Information Technology 2008, pp. 285–291

[99] J. Demmel, "The Future of Numerical Linear Algebra", BeBOP Group, UC Berkeley http://www.cs.berkeley.edu/~demmel/Utah_Apr05.ppt (accessed 08/04/2010)

[100] I. S. Duff and G. Meurant, "The Effect of Ordering on Preconditioned Conjugate Gradient," *BIT*29, 1989, pp.635-657

[101] Toledo S., "Improving the memory-system performance of Sparse Matrix vector multiplication", IBM Jnl. of Research and Development, Vol.41, no.6, pp. 711-725

[102] Garey, M.R.; Johnson, D.S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: W.H. Freeman. ISBN 0-7167-1045-5

[103] E. Cuthill and J. McKee. "Reducing the bandwidth of sparse symmetric matrices", In Proc. 24th Nat. Conf. ACM, pages 157-172, 1969

[104] W.-H. Liu and A. H. Sherman, "Comparative analysis of the Cuthill-McKee and the reverse Cuthill-McKee ordering algorithms for sparse matrices", SIAM Journal on Numerical Analysis, 13(2):198-213, April 1976

[105] J. Weidendorfer and C. Trinitis, "Cache Optimizations for Iterative Numerical Codes Aware of Hardware Prefetching", volume 3732 of Lecture Notes in Computer Science. Springer, 2006, pp. 921–927

[106] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali and P. Stodghill, "Is Search Really Necessary to Generate High-Performance BLAS?", Proceedings of the IEEE, Vol. 93, No. 2, Feb 2005, pp. 358- 386

[107] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. C. Whaley, K. Yelick, "Self Adapting Linear Algebra Algorithms and Software", Proceedings of the IEEE, Vol. 93, No. 2, Feb. 2005, pp. 293-312

[108] http://netlib.org/atlas (accessed 08/04/2010)

[109] http://www.icsi.berkeley.edu/~bilmes/phipac/ (accessed 08/04/2010)

[110] http://www.cs.berkeley.edu/~yelick/sparsity/ (accessed 08/04/2010)

[111] http://bebop.cs.berkeley.edu/oski (accessed 08/04/2010)

[112] E.-J. Im, "Optimizing the Performance of Sparse Matrix-Vector Multiplication", Ph.D. thesis, University of California, May 2000 (accessed 08/04/2010)

[113] R. W. Vuduc and H-J. Moon, "Fast sparse matrix-vector multiplication by exploiting variable block structure", Proc. International Conference on High-Performance Computing and Communications (HPCC), Sep 2005, pp.807-816

[114] A. Buttari, V. Eijkhout, J. Langou and S. Filippone, "Performance Optimization and Modelling of Blocked Sparse Kernels", ICL, Department of Computer Science, University of Tennessee, Report No. ICL-UT-04-05, 2004

[115] R. Nishtala, R. Vuduc, J. Demmel, K. Yelick, "When Cache Blocking Sparse Matrix Vector Multiply Works and Why", Applicable Algebra in Engineering, Communication, and Computing: Special Issue on Computational Linear Algebra and Sparse Matrix Computations, 2005, pp. 297-311

[116] R. Nishtala, R. W. Vuduc, J. W. Demmel and K. A. Yelick, "Performance Modelling and Analysis of Cache Blocking in Sparse Matrix Vector Multiply", EECS Department, University of California, Berkeley, Technical Report No. UCB/CSD-04-1335, 2004

[117] Z. Zhang and X. Zhang, "Fast Bit-Reversals on Uniprocessors and Shared-Memory Multiprocessors", *SIAM J. Sci. Comput.* 22, 6 (Jun. 2000), pp. 2113-2134

[118] J. Siek & A. Lumsdaine, "The Matrix Template Library: Generic Components for High-Performance Scientific Computing", IEEE Journal of Computing in Science & Engineering, Nov.-Dec. 1999, pp.70-78

[119] M. S. Lam, E. E. Rothberg and M. E. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms", ASPLOS-IV, Palo Alto, CA, Apr. 1991

[120] O. Temam, E. Granston, and W. Jalby, "To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts", In Proceedings of Supercomputing '93, Portland, OR, November 1993, pp. 410 - 419

[121] D. Parello, O. Temam, and J.-M. Verdun, "On increasing architecture awareness in program optimizations to bridge the gap between peak and sustained processor performance: matrix-multiply revisited", Proc. SC2002, pp.1-11

[122] E. Elmroth, F. Gustavson, I. Jonsson, and B. Kågström, "Recursive Blocked Algorithms and Hybrid Data Structures for Dense Matrix Library Software", SIAM Review, Vol. 46, No. 1, 2004, pp. 3-45

[123] S. T. Gabriel. and D. S. Wise, "The Opie Compiler: from Row-major Source to Morton-ordered Matrices", Proc. 3rd Workshop on Memory Performance Issues (WMPI-2004), New York: ACM Press, 2004 June, pp. 136-144

[124] D. S. Wise, C. L. Citro, J. J. Hursey, F. Liu, and M. A. Rainey, "A Paradigm for Parallel Matrix Algorithms: Scalable Cholesky", Proc. Euro-Par'05, Lecture Notes in Computer Science 3648, Berlin: Springer, August 2005, pp.687-698

[125] David S. Wise, "Ahnentafel indexing into Morton-ordered arrays, or matrix locality for free", Euro-Par 2000 – Parallel Processing, 2000, pp.774-784

[126] S. Chatterjee, A. R. Lebeck, Praveen K. Patnala, M. Thottethodi, "Recursive Array Layouts and Fast Parallel Matrix Multiplication", IEEE Transactions on Parallel and Distributed Systems (IEEE TPDS), 2002, pp. 1105 - 1123

[127] N. Park, B. Hong and V. K. Prasanna, "Analysis of Memory Hierarchy and Block Layout", Proc. Of the Intl. Conf. on Parallel Processing ICPP'02, 2002, pp.35-44

[128] OpenMP Application Program Interface, Version 2.5, public draft, November 2004

[129] C. Liao, Z. Liu, L. Huang, and B. Chapman, "Evaluating OpenMP on Chip MultiThreading Platforms", First International Workshop on OpenMP, IWOMP 2005. Eugene, Oregon USA. June 1-4, 2005 LNCS 4315, 2008, pp.178-190

[130] V. Packirisamy, H. Barathvajasankar, "OpenMP in Multicore Architectures" http://www-users.cs.umn.edu/~harish/reports/openMP.pdf (accessed 08/04/2010)

[131] H. Kotakemori, H. Hasegawa, T. Kajiyama, A. Nukada, R. Suda and A. Nishida, "Performance Evaluation of Parallel Sparse Matrix–Vector Products on SGI Altix3700" http://www.nic.uoregon.edu/iwomp2005/Papers/f27.pdf (accessed 08/04/2010)

[132] R. H. Bisseling and B. Vastenhouw, "A Two-Dimensional Data Distribution Method for Parallel Sparse Matrix-Vector Multiplication", SIAM REVIEW, Vol. 47, No. 1, 2005, pp. 67–95

[133] S. Riyavong, "Experiments on Sparse Matrix Partitioning", CERFACS Working Note WN/PA/03/32, CERFACS, 42 Avenue G. Coriolis, 31057 Toulouse Cedex, France

[134] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek, "Parallel Hypergraph Partitioning for Scientific Computing", http://www.math.uu.nl/people/bisseling/IPDPS06.pdf (accessed 08/04/2010)

[135] A. N. Yzelman and Rob H. Bisseling, "Cache-oblivious sparse matrix-vector multiplication by using sparse matrix partitioning methods", SIAM Journal on Scientific Computing, **31**, No. 4 (2009), pp. 3128-3154

[136] http://www.spectrum.ieee.org/energy/environment/the-greening-of-google (accessed 08/04/2010)

[137] E.-J. Im, K. A. Yelick, R. Vuduc, "SPARSITY: An Optimisation Framework for Sparse Matrix Kernels", International Journal of High Performance Computing Applications, 18 (1) , February 2004, pp. 135-158

[138] S. E. Richardson, "Exploiting Trivial and Redundant Computation", 11th IEEE Symposium on Computer Arithmetic, June 29 - July 2, 1993, Windsor, Ontario, pp. 220-227

[139] J.J. Yi and D.J. Lilja, "Improving Processor Performance by Simplifying and Bypassing Trivial Computations", International Conference on Computer Design, September 2002, pp.462-465

[140] Acken, K.P.; Irwin, M.J.; Owens, R.M.; Garga, A.K., "Architectural optimisations for a floating point multiply-accumulate unit in a graphics pipeline", Proceedings of ASAP 96 Conference, Aug. 1996, pp.65 – 71

[141] http://www.intel.com/products/processor/core2duo/specifications.htm (accessed 08/04/2010)

[142] Georgios I. Goumas, Kornilios Kourtis, Nikos Anastopoulos, Vasileios Karakasis, Nectarios Koziris: "Understanding the Performance of Sparse Matrix-Vector Multiplication". PDP 2008: pp.283-292

[143] http://www.fftw.org/ (accessed 08/04/2010)

[144] N. Nethercote, "Dynamic Binary Analysis and Instrumentation", PhD Dissertation, University of Cambridge, November 2004

[145] http://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html (accessed 08/04/2010)

[146] M. A. Ertl and D. Gregg, "The structure and performance of efficient interpreters", in Journal of Instruction-Level Parallelism, vol. 5, November 2003

[147] Tomasulo R.M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", IBM Journal of Research and Development, 11(1), January 1967, pp.25-33

[148] Sassone P.G., Rupley J., Brekelbaum E., Loh G. H., Black B., "Matrix Scheduler Reloaded", 34th International Symposium on Computer Architecture (ISCA 2007), June 9-13, 2007, San Diego, California, USA 2007, pp.335-346

[149] J. Farrell, T. Fischer, "Issue logic for a 600-Mhz out-of-order execution microprocessor", IEEE JSSC, Vol. 33, No. 5, May 1998, pp.702-712

[150] http://www.asics.ws/doc/efpu_brief.pdf (accessed 08/04/2010)

[151] http://download.intel.com/technology/architecture/new-instructions-paper.pdf (accessed 08/04/2010)

[152] http://www.xilinx.com (accessed 08/04/2010)

[153] Thomas D., Moorby P., "The Verilog Hardware Description Language", Kluwer Academic Publishers, Norwell, MA. ISBN 0-7923-8166-1

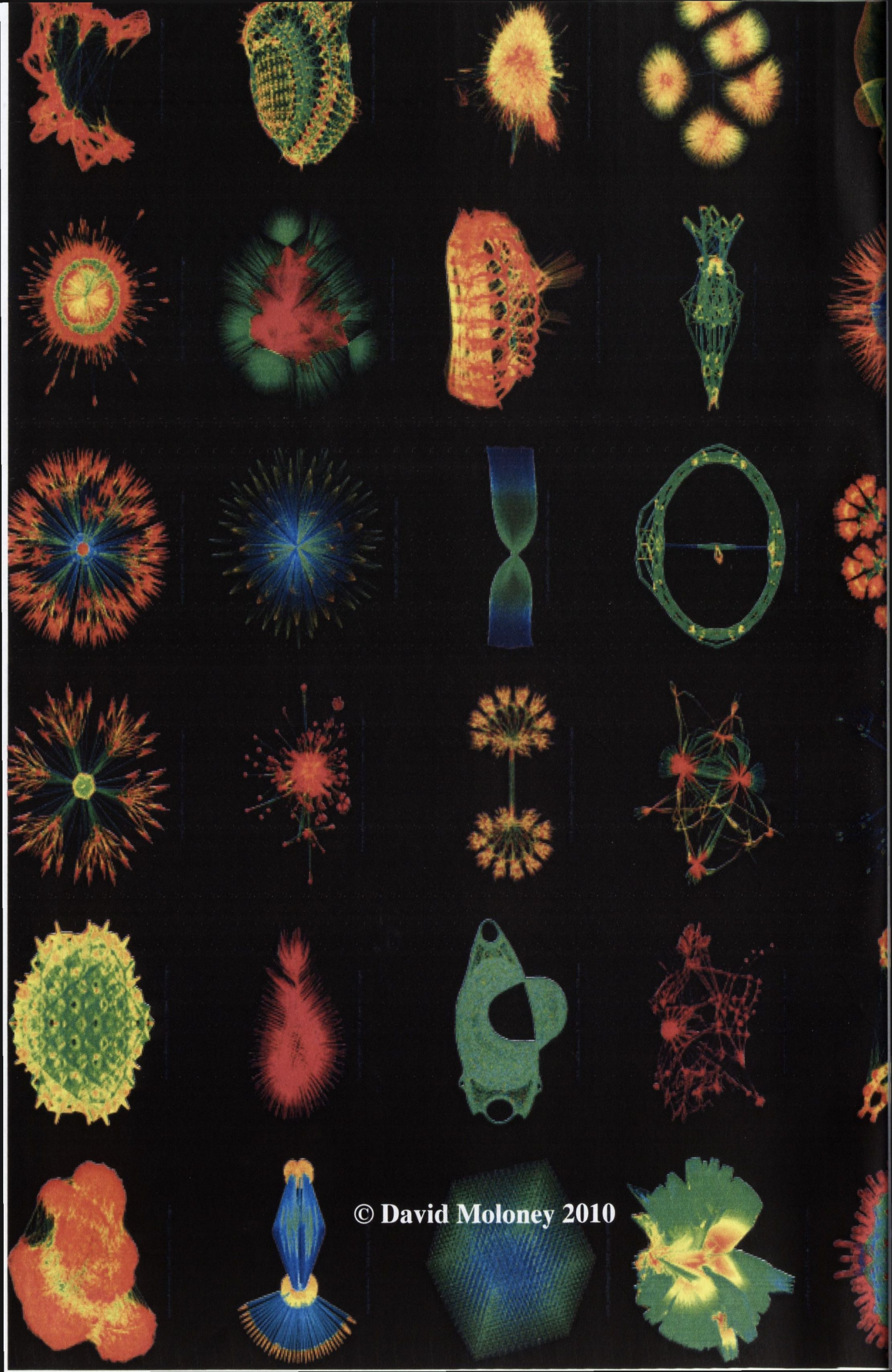
[154] Gwennap L., "Intel's P6 Uses Decoupled Superscalar Design", Microprocessor Report., 16 February 1995, pp.9-15

[155] Matt T. Yourst, "*PTLsim:* A Cycle Accurate Full System x86-64 Microarchitectural Simulator", Proc. ISPASS 2007, April 25-27, 2007, pp.23-34

[156] http://www.research.att.com/~yifanhu/GALLERY/GRAPHS/indexAll.html (accessed 08/04/2010)

[157] Yozo Hida, Xiaoye S. Li and David H. Bailey, "Algorithms for Quad-Double Precision Floating Point Arithmetic," *IEEE ARITH15*, 2001, pp.155-162

[158] D. Bailey, "High-Precision Arithmetic in Scientific Computation", *Computing in Science and Engineering*, May-Jun, 2005, pp.54-61


© David Moloney 2010