UNIVERSITY OF ADELAIDE
14.11.00
LIBRARY

Design and Evaluation of a Memory Architecture for a Parallel Matrix Processor Array

Nicholas M. Betts B.E.(Hons), B.Sc.

Department of Electrical and Electronic Engineering,
University of Adelaide.

A thesis presented in fulfilment of the requirement
for the degree of Doctor of Philosophy.

October, 2000

Table of Contents

List of Figures

Abstract

Although the computational power of general purpose computer architectures continues to grow, it is still possible to achieve far greater performance for a limited problem domain using a specialised computer architecture. In this thesis, a specialised matrix processor architecture is proposed that targets numerically intensive algorithms that can be cast in matrix terms. Such algorithms cover a very wide range of applications, from fields such as signal processing, control, and numerical simulations.

The work presented in this thesis builds on earlier work on matrix processors, where a mesh connected array of processing elements was used to perform matrix-matrix multiplication and the elementwise matrix operations of addition, multiplication, division, square root and comparison. A different memory architecture using large programmer-controlled registers and a load/store methodology is proposed, as an alternative to the previously described memory/memory architecture with caches. The major contributions lie in the following areas:

- the design of a novel specialised matrix processor architecture that utilises available memory technology in a fundamentally new way to achieve high performance computing;
- theoretical analysis of the architecture to explain the basis of its performance;
- extensive simulation of the architecture to evaluate performance, scalability and programmability.

Analysis of the load/store memory architecture shows how performance of the system depends on fundamental parameters of the system, such as the relative speeds of computation and memory, and ratio of problem and memory sizes. It is concluded that the proposed architecture provides a realistic and efficient way of achieving high performance computation for a wide range of problems.

Statement of Originality

This work contains no material which has been accepted for the award of any other degree or diploma in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text.

I give consent to this copy of my thesis, when deposited in the University Library, being made available for loan and photocopying.

Nicholas M. Betts
20th October, 2000.

Acknowledgements

Firstly, thanks to my supervisor Mike Liebelt for his seemingly endless patience with my mid-stream changes of topic, insane globe-trotting and the never-ending sequence of thesis drafts.

To the other members of the MATRISC project team, Andrew Beaumont-Smith, Kiet To, Cheng Chew Lim, for numerous discussions about the MATRISC architecture and especially to Warren Marwood, who got the whole thing rolling.

To the guys in the HiPCAT (aka. Digital) Lab all the way from the beginning to now, AJ, Ben, Shannon, Sam, Tim, Braden, Nasser and Lama, for making the times between ground breaking research much more fun.

Finally, to Nat for everything. But especially for the superhuman effort of proof-reading the final draft of this thesis.

UNIVERSITY OF ADELAIDE LIBRARY

Chapter 1

Introduction

The computational power of general purpose computer architectures continues to grow, seemingly without limit[Yu 96]. It is still possible, and necessary, to achieve far greater performance for a limited problem domain using a specialised computer architecture.

This thesis describes the development of a specialised computer architecture for dense matrix algorithms. Matrix operations provide particularly rich opportunities for acceleration by specialised hardware, because of their regular computational pattern and high compute to memory bandwidth ratio. By exploiting the computationally intensive matrix-matrix multiplication operation, this architecture achieves a better price/performance ratio than other architectures.

Because specialised architectures are only useful for a limited range of problems, they are rarely commercially successful due to the high cost of development. Successful specialised architectures have relied as much on supporting a wide range of important applications, as they have on performance improvement over general purpose machines for any one application. Hence there is considerable potential in an architecture that targets fundamental operations that are used in a number of different applications. One such example is dense matrix operations. There are a wide range of engineering, scientific and business applications that are based on algorithms expressed in matrix terms, and whose performance is strongly determined by the speed of a small set of matrix primitives. Examples of such are computational fluid dynamics, petroleum reservoir simulation and weather forecasting.

This chapter introduces a range of background information beginning with a discussion of key concepts from the field of computer architecture, followed by an overview of memory architectures in particular. The final section introduces matrix arithmetic and

algorithms, discusses some applications of dense matrix calculations, and justifies the focus on dense matrix calculations.

1.1 Key Concepts from Computer Architecture

This section describes some fundamental concepts and terminology from computer architecture that will be central to the work in later chapters.

1.1.1 Amdahl's Law

Amdahl's Law[Amdahl 67] describes the improvement in speed of some operation when an enhancement is made that speeds up the execution of only part of the operation. If a fraction, F, of the overall operation is sped up by a factor, S_F, then the overall operation is sped up by a factor, S, given by,

$$S = \frac{1}{(1-F) + \frac{F}{S_F}}$$

Although Amdahl's law is very simple, it is of fundamental importance when evaluating the performance of any computer architecture. There are a number of ways of expressing the underlying principle of Amdahl's Law, the most important of which include;

- **Make the common case fast. -** Speed improvements have more effect on the overall performance when they apply to an operation that is executed often, that is when F is large.
- **Balance the improvements over all operations. -** Large improvements to only part of a problem give only moderate overall benefit; effort is better spent trying to improve all aspects equally. For example, improving 50% of a problem by 10 times produces a speed up of 1.81, improving 50% of a problem by 100 times produces a speed up of 1.98, only 9% more. One particularly important application of this idea is to the implementation of algorithms on parallel computers. Because, in general, any algorithm contains a certain amount of code that cannot be parallelised and must therefore operate at scalar speed, there is a limit to the improvement that parallel machines can bring. However in many numerical algorithms, the parallel fraction increases with the size of the data set

and so parallel machines continue to provide speed up, but only for large problems. This phenomenon is known as the Gustafson-Baris law[Gustafson 88].

- **Restructure problems so that fast operation occur most often. -** When examining the impact of a proposed enhancement, the method used to solve a particular problem should be examined to ensure that the enhancement is being used as often as possible, provided that this doesn't incur prohibitive penalties elsewhere. The best way to solve a given problem with the enhanced system may be different from the best way to solve the problem without the enhancement. This observation leads to the following important principle: when measuring the performance of a specialised computer architecture, an algorithm which is optimised for the enhanced architecture should be used. This may often involve a completely new approach to the problem.

Although these qualitative statements provide a good guide when designing computer architectures, the most important application of Amdahl's Law is still to perform a quantitative analysis to evaluate the performance of proposed enhancements.

1.1.2 Reduced Instruction Set Computers

To increase the performance of a scalar processor, it is necessary to either increase the work done by each instruction or decrease the time taken to perform each instruction. Before the 1980s, most work on computer architectures had focused on the first alternative by evolving processors with increasingly complex instructions. General improvements in the implementation technology were then relied on for decreasing execution time of each instruction. It was believed that software cost could be reduced by reducing the *semantic gap*[Wulf 81] between the machines instruction set and high level languages. However this approach often led to powerful but slow instructions that compiler writers chose not to use.

A series of projects started in the late 1970s and early 1980s challenged this philosophy, concentrating instead on machines that executed very simple instructions at the maximum possible rate[Patterson and Ditzel 80]. These machines, dubbed *Reduced Instruction Set Computers* (RISC), have a number of important characteristics.

RISC machines have a large number of general purpose registers. By providing 32 or more registers, compilers could much more easily arrange the code so that most accesses occurred to registers rather than to memory.

An arithmetic instruction may have to specify up to three addresses, two input operands and a result. In general these addresses may be in either the main memory or the processor's register file. RISC processors only allow accesses to the registers, which classifies them as *register-register* architectures. In a register-register architecture, data is only moved between the registers and main memory using explicit load and store instructions, and for this reason these architectures are also known as *load/store* architectures.

These architectural features simplify pipelining, where the execution of consecutive instructions is partially overlapped. Pipelining greatly improves performance by significantly reducing the effective number of clock cycles taken per instruction.

1.1.3 Parallelism

Because the performance of a single processor is limited by the available technology, designers have long sought to build machines where a number of processors act in parallel. Parallel architectures may be classified according to a number of different characteristics. One of the most important classifies machines according to whether they have single or multiple instruction streams and whether they have single or multiple data streams, thus resulting in four categories[Flynn 66].

- **Single Instruction Single Data (SISD) -** This class corresponds to scalar processors.
- **Single Instruction Multiple Data (SIMD) -** Machines in this class consist of a number of processors that perform the same operation on different pieces of data.
- **Multiple Instruction Single Data (MISD) -** It is generally held that there is no architecture that falls into this category, although some suggest that systolic architectures are appropriate examples[Kung 88].
- **Multiple Instruction Multiple Data (MIMD) -** Processors in machines of this class each perform a different set of instructions. This approach is more flexible and powerful than SIMD but requires more programmer effort to synchronise the operation of the different processors.

A second classification is based on how memory is organised. Hennessy and Patterson[Hennessy and Patterson 96] proposes two orthogonal classifications: first, the location of physical memory, being either *centralised* or *distributed* with the processors; and second, the type of addressing used, being either a single *shared address space* for all processors or a set of *multiple private address spaces*.

For a system with a small number of processors, a centralised memory may be the most efficient solution. Centralised memory is both simpler to build and easier to program than distributed memory. However, as the number of processors increases, the contention for a central memory must eventually force the use of distributed memory. The use of caches at each processing node can increase the number of processors that can be supported by a central memory, but this introduces the problem of keeping the caches consistent.

Using multiple private address spaces increases the difficulty of writing software because the programmer must use explicit message passing to transfer data between different processors. When a shared address space is used with distributed memory, some areas of memory operate at different speeds, depending on their distance from the processor, and for this reason this type of machine is also called a *non-uniform memory access* (NUMA) machine.

Parallel architectures can also be classified according to the way in which the processors are connected together. Some common connection topologies are shown in Figure 1.1.

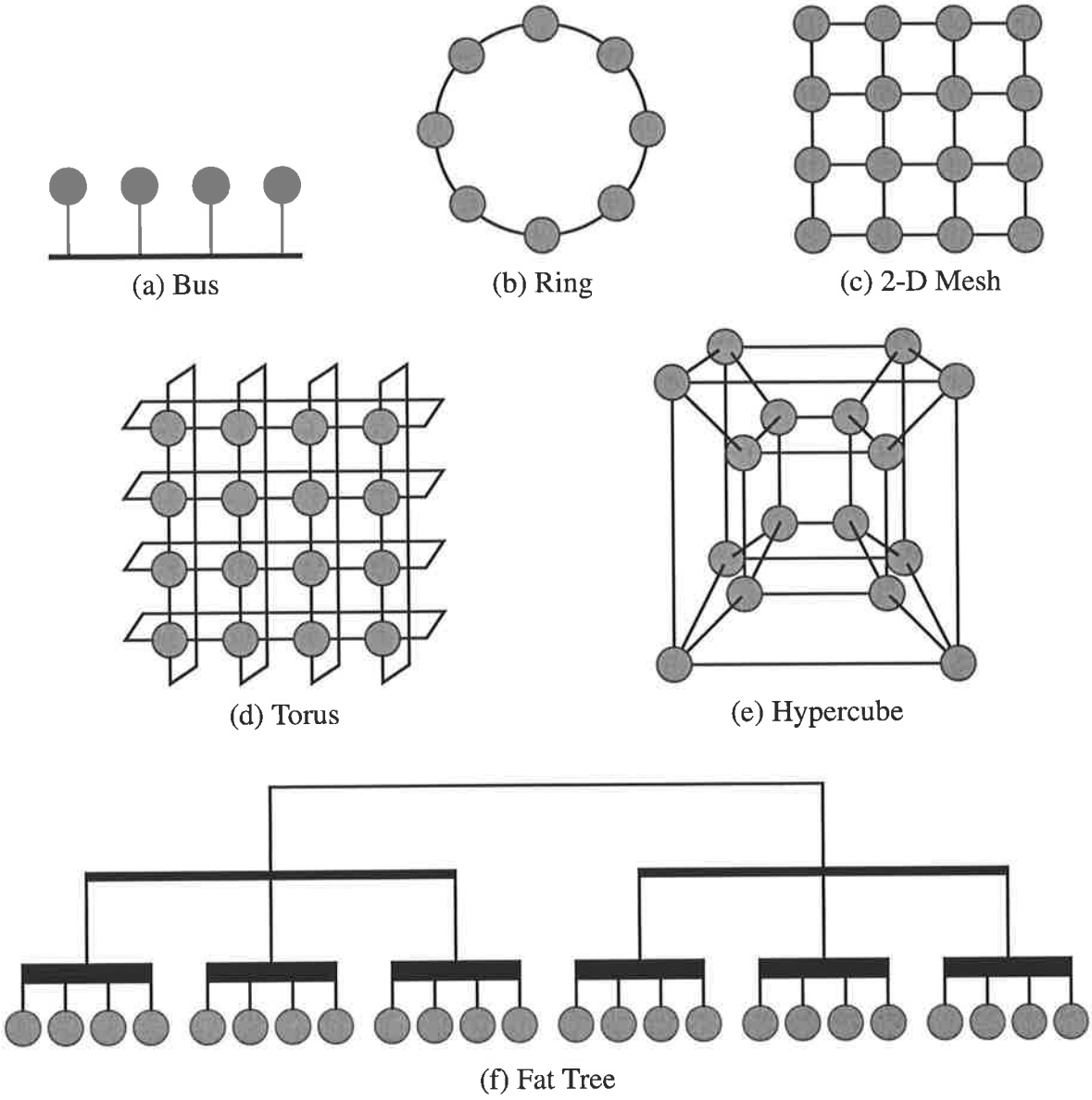


Figure 1.1: Multiprocessor Connection Topologies

Bus connections are a simple way to connect a small number of processors, particularly if a centralised memory is used. Buses do not scale well to large numbers because contention for access to the shared bus eventually forms a bottleneck.

A ring interconnect is quite simple and many algorithms can be mapped onto it successfully. Although the total inter-processor bandwidth scales with the number of processors, the useful size of the ring is limited by the longest path between processors, which increases the average latency between the processors.

In contrast, a 2-D mesh provides much more inter-processor bandwidth and the longest path between processors grows only with the square root of the number of processors. The

torus, which is simply a wrapped around 2-D mesh, is similar but the longest path is halved. This connection maybe extended in the third dimension to form either a 3-D mesh or 3-D torus.

A hypercube network consists of nodes connected in the arrangements of higher dimensional analogues of the cube. The network contains $N = 2^n$ processors, with each connected to n neighbours. Figure 1.1(e) shows a four dimensional hypercube. The inter-processor bandwidth is very high, but the size of the network must grow by factors of two and the number of connections per node can become large. However, the longest path is only $n = \log_2 N$ connections, and many elegant and simple algorithms map well onto the structure.

The fat tree groups nodes into a hierarchy. Using a binary tree with higher bandwidth, higher in the tree it is possible to show that for a fixed amount of hardware a fat tree is nearly the best routing network of that size[Leiserson 85]. It is also possible to use higher bandwidth lower in the tree to take advantage of algorithms that can be structured to use much more local than global communication. This is illustrated in Figure 1.19(f) where the thickness of the lines indicates the relative bandwidth of the connections. Such a system is generally easier to build but does not achieve good performance in all situations.

1.1.4 Compute-Bound and Memory-Bound Systems

In many situations it is convenient to view a computer architecture as consisting of a set of computational elements, operating at some rate, on data supplied by an memory system with some given bandwidth. Considering a system where a specific algorithm is performed on a specific data set, the following definitions are made: the system is *compute-bound* if an increase in the memory bandwidth would not significantly reduce the total execution time; and, the system is *memory-bound* if an increase in the computational execution rate would not significantly reduce the total execution time.

If an algorithm is such that it is likely to be compute-bound or memory-bound on most architectures, then the algorithm itself may be described as compute-bound or memory-bound. Similarly, an architecture may be described as compute-bound or memory-bound if it behaves in that fashion for most algorithms. These definitions provide a terminology for describing broad architectural features in terms of memory and compute power.

In parallel machines, the same terms may be used but substituting memory bandwidth with inter-processor communication bandwidth.

1.1.5 Measuring Performance

The question of how to measure the performance of a computer architecture is very complex. To the end user, the figure of merit is usually the time taken to run the user's applications, but other metrics such as reliability, accuracy, and physical configuration may be important in some situations.

In numerical computing it is common practice to measure the performance of algorithms using the number of floating point operations, or flop, executed per second (flop/s). The term flop, when used as a unit of measure, will not have a plural 's' added. Note that the abbreviation flops, or FLOPS, may be used for FLoating point Operations Per Second, particularly in the context of phases like, "500 MFLOPS processor". This usage will be avoided here. The precision of numbers used is important and should be stated; throughout this thesis double precision IEEE floating point will be used.

The floating point operations may be addition, subtraction, multiplication and division. If flop/s figures are calculated using required floating point operations and elapsed time, and comparisons are only made between the same operation on the same data set, then they are exactly equivalent to running time. By using the required number of flop for an *operation* (the task) rather than the actual number of flop performed by the *algorithm* (the method used to complete the task), fair comparisons can be made between situations where different algorithms are used. In particular, if an algorithm uses more than the minimum number of floating point operations, it is not fair to include the extra operations when calculating the algorithm's flop/s rate.

In general, the number of floating point operations per second varies wildly between different operations and data sets. However for dense matrix algorithms, flop/s rate is relatively constant. Thus floating point operations are often a good indicator of running time on a scalar processor, because floating point operations dominate the algorithms.

One way of attempting to gauge the performance of different machines is by using standard benchmarks. The LINPACK benchmark, which involves solving a dense system of linear equations, is widely used for measuring dense matrix performance[Dongarra 98]. There are three different versions of this benchmark, each of which uses a different set of conditions and caters to a different size of machine.

The first version involves solving a linear system of order n=100 using a given Fortran program, which may not be altered in any way. The resultant performance figure, simply

called the "LINPACK Benchmark", is calculated in Mflop/s from the actual running time and a theoretical required operation count of $2n^3/3 + 2n^2$. This version of the benchmark has for some time been too small to adequately measure performance and all current systems achieve a small percentage of their peak performance for it.

The second version involves solving a system of order n=1000 using any algorithm, provided that it produces accurate results. In particular, if the Gaussian Elimination algorithm is used, it must employ partial pivoting. This performance figure, also in Mflop/s, is called "LINPACK TPP", which is an abbreviation for 'towards peak performance'.

For the largest and most powerful of the currently available parallel machines, linear systems of order 1000 are not large enough for the architecture to reach maximum performance. To compare these machines, a third version is used with the same accuracy and flop counting rules as before, but no limit on the problem size. The following performance figures are reported:

R_{max}, the performance in Gflop/s for the largest problem run on the machine;

N_{max}, the size of the largest problem run on the machine;

$N_{1/2}$, the size where half the R_{max} execution rate is achieved; and,

R_{peak}, the theoretical peak performance in Gflop/s for the machine.

This last benchmark is called "LINPACK MPP", which is an abbreviation for 'massively parallel processor'.

Comparing machines using LINPACK is potentially misleading because it involves only one algorithm. Many applications exist whose running time is dominated by solving dense systems of equations, but Amdahl's Law shows that LINPACK may not predict the performance of such applications. Use of LINPACK as a broad indication of a computer system's performance in this thesis is justified by its wide availability and because dense matrix algorithms are precisely what is of interest here.

1.2 Memory Architectures

As the number and speed of computational elements in most computer architectures continues to increase rapidly, the task of supplying operands to these elements becomes increasingly difficult. One method that has been used to quantify this problem is the STREAM benchmark[McCalpin 95]. STREAM measures sustained memory bandwidth

using four long vector operations, and defines *machine balance* as,

$$\text{machine balance} = \frac{\text{peak floating ops/cycle}}{\text{sustained memory ops/cycle}}$$

A machine with a balance greater than one will be memory-bound for long vector operations. Conversely machine with a balance less than one will be compute-bound for long vector operations. An extensive survey of recent machines has shown that the majority have a high, that is poor, machine balance. Examination of historical trends has shown an increase in balance with time for most classes of machine, the notable exception to which is vector computers, because they use interleaved memory systems that provide very high sustainable bandwidth. The underlying reason for poor machine balance is that microprocessor speed is increasing at 80% per year, whereas memory speed is increasing at only 7% per year. The result is a tendency towards exponential growth in compute to memory ratio. Of course memory that tracks the processor speed, such as on-chip cache and fast SRAM on an MCM with the processor, is available but its cost per byte is far too high for it to be used for main memory.

To satisfy this rapidly increasing need for memory bandwidth, ever more sophisticated memory architectures are being used, and are becoming increasingly important in determining overall performance. The primary reason that complex memory architectures have become necessary is that the design of memory systems is drawn in two mutually incompatible directions. Firstly, memory speed must improve so that it can supply operands to computational units whose speed is constantly improving. Secondly, memory size must increase so that larger data sets and more complex algorithms may be used.

1.2.1 Memory Technologies

In attempting to produce memory that is both fast and large, a number of distinct memory technologies have emerged as the most cost effective. Here these memories will be classified into the following broad categories: register memory, SRAM and DRAM.

1.2.1.1 Register Memory

Register memory is made using the fastest memory technology available in order to match the speed of the computational units with which it is associated. Typically, registers support multiple reads and writes per clock cycle. This memory also requires the largest circuit area per bit and has the highest power dissipation, so its size is severely limited by

cost.

1.2.1.2 SRAM

Static Random Access Memory (SRAM) is built from an array of latches which hold their value indefinitely in the presence of power. With increasing die size and decreasing line size, it has become possible to include quite large SRAM memories on the same chip as a processor. These *on-chip SRAMs* require fewer transistors per bit than register memories, and can therefore be much larger. On-chip SRAM is now used on almost all new microprocessors as a first level cache and, in the case of some such as the DEC Alpha[DEC 97], as a second level cache as well.

SRAM implemented a separate chip (*off-chip SRAM*) suffers from the delays associated with off-chip communication. However, this is somewhat offset by the fact that a semiconductor process specifically tailored to SRAM devices may be used, which allow them to consume less power and die area per bit.

1.2.1.3 DRAM

Dynamic Random Access Memory (DRAM) stores each bit as a charge on a small capacitor. The memory is dynamic because its contents must be periodically read and rewritten, an operation know as *refreshing*, so that the data is not lost as the charge on the capacitor leaks away. DRAM chips are fabricated using specialised semiconductor processes to achieve the highest possible densities. The size of DRAMs doubles approximately every 3 years. Currently 64Mbit chips are available commercially, and 1Gbit chips have been demonstrated[Yoo et al. 96]. In contrast, the speed of DRAM chips has been relatively static for some time, prompting development of a series of new DRAM architectures to improve transfer rates[Jones et al. 92].

The internal architecture of a DRAM chip consists of an array of bit cells of which one row can be read out at a time and latched by a row of sense amplifiers. Accesses to different columns within a row already sensed are considerably faster than accesses that require the sensing of a new row. Traditional DRAM interfaces require the row address and then the column address to be sent to the chip via an asynchronous interface.

Synchronous DRAMs (SDRAMs) use a synchronous interface where a known number of clock cycles occurs between supplying the address and the data being available. This allows the device accessing the memory to do useful work in the interval, including making

further requests to the memory, which allows SDRAM to achieve much higher throughput than an asynchronous interface. SDRAMs are currently being used in most new PC systems.

RAMBUS DRAMs[Crisp 97][Rambus 97] use a fast narrow synchronous bus to transfer data to and from the memory. The current generation, called Direct RAMBUS, uses a 16bit bus operating at 800MHz to provide a peak bandwidth of 1.6GByte/s. The advantage of the RAMBUS is that it achieves a high speed memory system with few components, which allows it to achieve high performance in low end consumer products, and also allows the possibility of using more than one channel to gain very high performance in high end systems. RAMBUS also allows large block transfers and random accesses within the row of memory currently cached on the sense amplifiers, both of which can considerably reduce the time wasted on control overhead when accessing the memory.

1.2.2 Memory Hierarchies

An ideal memory architecture would involve using a single memory that was both fast enough to keep up with the computational units, and large enough to satisfy the requirement of the most memory-hungry algorithm to be run on it. Unfortunately, the fastest memory technologies are very expensive and consume a great deal of power, and so only a small amount can reasonably be used in a memory system. Instead, a memory system that uses a range of different memory elements, which utilise different memory technologies, must be used.

The different memory technologies can be arranged into memory hierarchies to give the illusion of a single fast large memory. A typical memory hierarchy is shown in Figure 1.2. It is composed of a central processing unit (CPU) linked to registers, two caches and a main memory. The size of each memory represents the amount of storage it provides, which increases further from the CPU. The width of the interconnections is representative of their speed, which is higher closer to the CPU. Note that the figure is not to scale.

In general, memory accesses pass through each level of the memory hierarchy. In load/ store machines, accesses to all lower levels of memory must occur through the registers, but in other types of machines, memory is often accessed directly. Special mechanisms may also be used to bypass one, or more, levels of cache in situations where the cache would hinder performance.

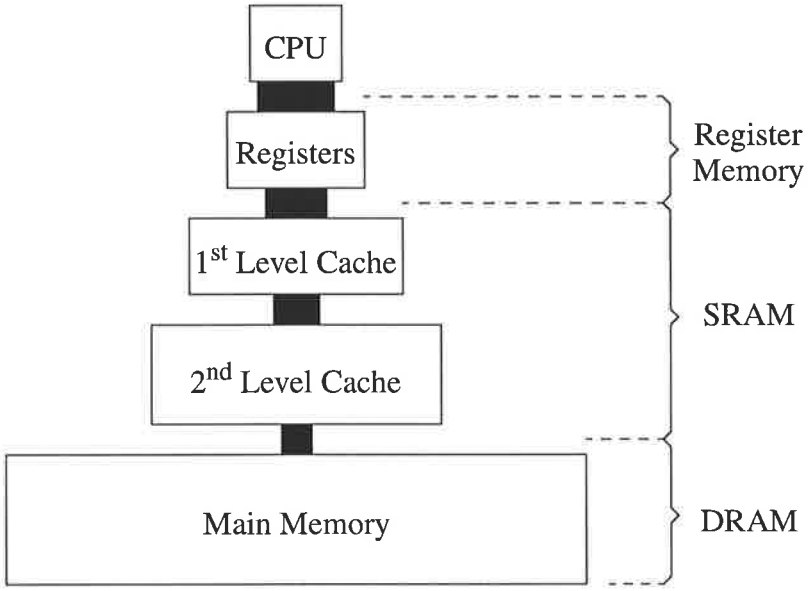


Figure 1.2: Typical Memory Hierarchy

Usually the memory technologies used to construct each level of the hierarchy are as shown in the right of Figure 1.2. However there are also functional characteristics of each level that are important regardless of the technology. Here we discuss three of the most important levels, registers, caches and main memory, because they will be of particular interest in the architecture of the matrix processor.

1.2.2.1 Registers

Registers are storage locations that are directly coupled to the computational units with which they are associated. They are usually located on the same chip as the computation units and built using the fastest possible techniques. The functional characteristic of registers is that they are under explicit user control. In load/store architectures, all data used must pass through the registers. Determining the best way to use the register may often complicate programming, but the explicit control provides the ability to extract maximum performance.

An example of registers that are not on the same chip as the processor are the E-registers of the Cray T3E multiprocessor[Anderson et al. 97]. The Cray T3E is a NUMA MIMD machine, where each node has 512 user, and 128 system, 64bit E-registers that can be used to transfer data to and from global (remote or local) memory. The processor issues a series of Get or Put operations to the E-registers, which then perform the transfers without further processor intervention. Although access to remote memory is very slow, the large number of

E-registers allows a high degree of pipelining. E-registers can also be used simply to bypass the data cache for local memory access, which is faster for certain access patterns where the cache behaviour impedes performance.

1.2.2.2 Caches

Caches store copies of sections of main memory in relatively fast memory to reduce the latency seen by the processor when accessing memory. Whenever the processor reads a particular address, the cache checks if it has that piece of data and if it does, a *cache hit* is said to have occurred, and the cache passes the data to the processor. If the cache doesn't have the data, a *cache miss* is said to have occurred, and the cache loads that data from main memory and passes it to the processor. The cache loads data from main memory only in certain sized blocks, called *cache lines*, each of which is large enough to hold some small number of adjacent words.

A cache relies on the properties of memory access patterns to reduce the average time required to access memory: *spacial locality*, which is the tendency of locations near one another to be accessed one after the other; and *temporal locality*, which is the tendency for the same address to be accessed repeatedly in a short space of time.

In contrast to registers, caches are not explicitly controlled by the user. This tends to make caches less of a programming burden, but to achieve optimum performance, the effects of cache behaviour must often be taken into account.

The operation of caches can be improved using a number of techniques, such as prefetching[Baer and Chen 94]. One particular example is the intelligent prefetching performed by '*streams*' in the Cray T3E[Anderson et al. 97]. When a large amount of memory is accessed sequentially, a cache improves performance significantly because each cache line provides a number of cache hits. However, a cache miss occurs as each new line is brought into the cache. Streams detect the occurrence of sequential access and begin speculatively prefetching the required cache lines before they are needed. This mechanism can improve loading bandwidth by 179% and storing bandwidth by 71%.

1.2.2.3 Main Memory

Main memory is the bottom of the semiconductor memory hierarchy, with the largest capacity and the slowest access time. In particular, the latency of main memory is usually very high even though the bandwidth may be quite large. In a multiprocessor system, main

memory may be shared amongst the processors, in which case it may be able to support multiple transactions.

Various methods can be used to increase the bandwidth of main memory. The simplest method is to use a very wide memory, for example 128 or 256 bits. This allows whole cache lines to be loaded in few memory transactions, possibly only one, but does not reduce latency. It also increases system cost by requiring a large number of memory chips and wide buses.

Another method to achieve a high main memory bandwidth is to use a narrow, but short cycle time, bus interface such as that employed by RAMBUS memory. This can reduce the number of memory chips required for the memory system, thereby reducing system cost, and allows the use of multiple channels to increase the bandwidth.

Alternatively, multiple interleaved memory banks may be used with adjacent memory words in different banks. This allows accesses that occur to different banks to be pipelined and thus a high transfer rate can be achieved. This type of system is weakest when memory is accessed with a stride equal to the number of banks, as in this case all the accesses occur to one bank and the transfer rate is drastically reduced. Interleaved memory is used in vector computers where memory is often accessed sequentially. However, they have limited temporal locality and so most do not uses caches as they are not very effective. As these machines rely on a fast main memory rather than caches to supply a high memory bandwidth to the CPU, they have a very high sustainable memory bandwidth and thus achieve the best 'machine balance' as measured by the STREAM benchmark[McCalpin 95].

The design of memory hierarchies is thus based on complex trade-offs between the size and speed of the different levels of the hierarchy in determining the overall performance of the memory architecture. For example, a larger but slower cache may give a better average access time because its larger size reduces the number of expensive misses. Memory architectures are invariably customised for the processor implementation.

1.3 Matrix Arithmetic and Algorithms

There are a large number of algorithms in the field of numerical computation that are readily expressed in matrix terms, which results in a concise description of the task to be

performed and makes plain the data parallelism. This section defines a notation for describing matrices and matrix operations, and introduces a number of fundamental matrix concepts. The issues involved in efficient implementation of matrix operations and how these affect algorithm design are discussed. In addition, some examples of real applications are given to justify the focus of the MATRISC architecture on matrix computations.

1.3.1 Matrix Notation

Matrix operations will be described using the notation of Golub and Van Loan[Golub and Van Loan 96]. This notation is convenient because it aligns closely with the Matlab matrix language, which will be used extensively in later chapters.

Unless stated otherwise, matrix elements will be assumed to belong to the set of real numbers, denoted $\Re$. The set of all $n \times m$ matrices will be denoted $\Re^{n \times m}$ and defined by,

$$A \in \Re^{n \times m} \quad \Leftrightarrow \quad A = (a_{ij}) = \begin{bmatrix} a_{11} & \cdots & a_{1m} \\ \cdots & & \cdots \\ a_{n1} & \cdots & a_{nm} \end{bmatrix} \qquad a_{ij} \in \Re$$

When a capital letter is used to denote a matrix, the corresponding lower case letter with subscript ij is used to refer to the j^{th} element of the i^{th} row. When describing matrix algorithms, the same element will be denoted in Matlab style as A(i,j).

The set of n dimensional vectors is denoted $\Re^n$ and is defined by,

$$x \in \Re^n \quad \Leftrightarrow \quad x = \begin{bmatrix} x_1 \\ \cdots \\ x_n \end{bmatrix} \qquad x_i \in \Re$$

Note that $\Re^n$ is identified with $\Re^{n \times 1}$, so the members of $\Re^n$ are column vectors.

Often a rectangular adjacent subset of the elements of a matrix, called a *submatrix*, will be of interest. This will be denoted using Matlab style colon notation, defined by,

$$A(r_1{:}r_2, c_1{:}c_2) = \begin{bmatrix} a_{r_1c_1} & \cdots & a_{r_1c_2} \\ \cdots & & \cdots \\ a_{r_2c_1} & \cdots & a_{r_2c_2} \end{bmatrix}$$

When the index on either side of the colon is omitted, the value is assumed to be the extreme value for that particular matrix dimension. A single index value with no colon indicates that just one row or column in that direction is desired. For example, a single row

of a matrix is denoted $A(r, :)$ and a single column $A(:, c)$.

1.3.2 Matrix Operations

The fundamental matrix operations are addition, matrix multiplication, scalar multiplication, elementwise (or Hadamard) multiplication and transposition. These operations are, respectively, denoted and defined as,

$$C = A + B \Rightarrow c_{ij} = a_{ij} + b_{ij} \qquad A, B, C \in \Re^{n \times m}$$

$$C = AB \Rightarrow c_{ij} = \sum_{k=1}^{r} a_{ik} b_{kj} \qquad A \in \Re^{n \times r}, B \in \Re^{r \times m}, C \in \Re^{n \times m}$$

$$C = \alpha A \Rightarrow c_{ij} = \alpha a_{ij} \qquad A, C \in \Re^{n \times m}, \alpha \in \Re$$

$$C = A \otimes B \Rightarrow c_{ij} = a_{ij} b_{ij} \qquad A, B, C \in \Re^{n \times m}$$

$$C = A^T \Rightarrow c_{ij} = a_{ji} \qquad A \in \Re^{n \times m}, C \in \Re^{m \times n}$$

There is a requirement for matrix multiplication that the number of columns in the first matrix equal the number of rows in the second. Two matrices that have this size relationship are called *conformal*. A matrix product where n=m=1 is called an *inner product*. A matrix product where r=1 is called an *outer product*.

A matrix which equals its transpose is *symmetric*.

The additive identity matrix is called the *zero matrix* and it has all zero elements. The zero matrix is denoted 0; it may be any size. The multiplicative identity is a square matrix, simply called the *identity matrix*, denoted I_n and defined by,

$$AI_n = I_n A = A \qquad \forall A \in \Re^{n \times n}$$

The identity matrix has elements equal to one on the leading diagonal and zero elsewhere.

The inverse of a matrix is denoted by A^{-1} and defined by,

$$C = A^{-1} \Rightarrow AC = CA = I$$

A matrix cannot have an inverse unless it is square. This fact can be derived very simply by considering the conformality requirements implied in the definition. It is possible for a square matrix not to have an inverse, in which case it is called *singular*. A matrix whose transpose is its inverse is *orthogonal*.

During the development of matrix algorithms, several special matrix forms are used. An *upper triangular* matrix contains only zero elements below the leading diagonal, a *lower*

triangular matrix contains only zero elements above the diagonal. *Unit upper triangular* and *unit lower triangular* matrices are defined similarly but also have all elements on the diagonal equal to one.

It is often useful in matrix algorithms to have a method of measuring the size of a vector. To do this, a class of functions called the *p-norms* is often used. The double bar symbol, $\| \ \|$, is used to denote the norms and they are defined by,

$$\|x\|_p = (|x_1|^p + \ldots + |x_n|^p)^{\frac{1}{p}}$$

The most useful p-norms are the 1-norm, 2-norm and ∞-norm.

$$\|x\|_1 = (|x_1| + \ldots + |x_n|)$$

$$\|x\|_2 = (|x_1|^2 + \ldots + |x_n|^2)^{\frac{1}{2}} = (xx^T)^{\frac{1}{2}}$$

$$\|x\|_\infty = \max_{1 \le i \le n} |x_n|$$

The 2-norm is equal to the *length* of the vector in Euclidean space.

1.3.3 Block Matrices

A block matrix is one in which the rows and columns have been partitioned, dividing the matrix into a number of submatrices, or blocks. In general, an $n \times m$ matrix A may be blocked as

$$A = \begin{bmatrix} A_{11} & \ldots & A_{1r} \\ \ldots & & \ldots \\ A_{q1} & \ldots & A_{qr} \end{bmatrix} \begin{matrix} n_1 \\ \\ n_q \end{matrix}$$
$$\begin{matrix} m_1 & & m_r \end{matrix}$$

where $n_1 + \ldots + n_r = n$, $m_1 + \ldots + m_q = m$, and $A_{\alpha\beta}$ designates the (α,β) block or submatrix. Block $A_{\alpha\beta}$ has dimension $n_\alpha \times m_\beta$. Row and column blocking are special cases of general matrix blocking, with the conditions r=1 and q=1 respectively. In most applications, a particular block size will be chosen and all the blocks will be the same width and/or height, except for possibly the last one, which may be smaller if the matrix size is not a multiple of the block size.

Block matrices are important because they allow a clear description of block algorithms. The term *block algorithm* is used to imply an algorithm rich in matrix-matrix multiplication.

On most general purpose machines, increasing the amount of matrix-matrix multiplication tends to increase performance. On a special purpose architecture optimised for matrix multiplication increasing the amount of matrix-matrix multiplication is absolutely critical to achieving optimum performance.

Operations on blocked matrices can be performed just as on matrices with scalar entries, provided that certain dimension requirements are met.

For addition, the matrices must both be partitioned in the same way. If B is partitioned as,

$$B = \begin{matrix} \begin{bmatrix} B_{11} & \cdots & B_{1r} \\ \cdots & & \cdots \\ B_{q1} & \cdots & B_{qr} \end{bmatrix} \begin{matrix} m_1 \\ \\ m_q \end{matrix} \\ \begin{matrix} n_1 & & n_r \end{matrix} \end{matrix}$$

then the sum with matrix A above, $C = A + B$, can be regarded as yet another matrix partitioned in the same way and given by,

$$C = \begin{bmatrix} C_{11} & \cdots & C_{1r} \\ \cdots & & \cdots \\ C_{q1} & \cdots & C_{qr} \end{bmatrix} = \begin{bmatrix} A_{11}+B_{11} & \cdots & A_{1r}+B_{1r} \\ \cdots & & \cdots \\ A_{q1}+B_{q1} & \cdots & A_{qr}+B_{qr} \end{bmatrix}$$

For matrix multiplication the operands must be partitioned thus,

$$A = \begin{matrix} \begin{bmatrix} A_{11} & \cdots & A_{1r} \\ \cdots & & \cdots \\ A_{q1} & \cdots & A_{qr} \end{bmatrix} \begin{matrix} m_1 \\ \\ m_q \end{matrix} \\ \begin{matrix} p_1 & & p_s \end{matrix} \end{matrix}, \qquad B = \begin{matrix} \begin{bmatrix} B_{11} & \cdots & B_{1r} \\ \cdots & & \cdots \\ B_{q1} & \cdots & B_{qr} \end{bmatrix} \begin{matrix} p_1 \\ \\ p_s \end{matrix} \\ \begin{matrix} n_1 & & n_r \end{matrix} \end{matrix} \tag{1.1}$$

The product, $C = AB$, is then partitioned and calculated as follows,

$$C = \begin{matrix} \begin{bmatrix} C_{11} & \cdots & C_{1r} \\ \cdots & & \cdots \\ C_{q1} & \cdots & C_{qr} \end{bmatrix} \begin{matrix} m_1 \\ \\ m_q \end{matrix} \\ \begin{matrix} n_1 & & n_r \end{matrix} \end{matrix} \tag{1.2}$$

$$C_{\alpha\beta} = \sum_{\gamma=1}^{s} A_{\alpha\gamma} B_{\gamma\beta} \qquad \alpha = 1{:}q \qquad \beta = 1{:}r$$

One special case that deserves mention, because it is fundamental to the operation of the MATRISC processor described in later chapters, is forming a product AB between a row

partitioned A and a column partitioned B. This situation is equivalent to $s = 1$.

$$A = \begin{bmatrix} A_1 \\ \dots \\ A_q \end{bmatrix} \begin{matrix} m_1 \\ \\ m_q \end{matrix}, \qquad B = \begin{bmatrix} B_1 & \cdots & B_r \end{bmatrix}, \qquad C = \begin{bmatrix} C_{11} & \dots & C_{1r} \\ \dots & & \dots \\ C_{q1} & \dots & C_{qr} \end{bmatrix} \begin{matrix} m_1 \\ \\ m_q \end{matrix}$$

$$\begin{matrix} n_1 & & n_r \end{matrix} \qquad \begin{matrix} n_1 & & n_r \end{matrix}$$

$$C_{\alpha\beta} = A_\alpha B_\beta$$

Note that there are no dimensionality requirements on the partitions, so they can all be any size. The result, C, consists of blocks that are the product of a row partition from A and a column partition from B; there is no summation required.

1.3.4 Matrix Primitives

When implementing matrix algorithms there are several combinations of the basic matrix and vector operations that occur so frequently that they have been given special names. The *saxpy* operation, which is a mnemonic for 'scalar alpha x plus y', is defined by,

$$z = \alpha x + y$$

An *outer product update* occurs when a matrix is altered by the addition of the result of an outer product, such as,

$$A \leftarrow A + xy^T$$

A *gaxpy* operation, which is a mnemonic for 'general A x plus y' is defined by,

$$z = Ax + y$$

When performing matrix operations, the amount of data required compared with the number of floating point operations is very important. The number of floating point operations gives an indication of the amount of computation required, while the amount of data gives an indication of the memory transfers required. The ratio between the two indicates whether the operation will be compute-bound or memory-bound.

Three levels are defined to describe the effort required by different operations[Golub and Van Loan 96]. Level-1 operations involve an amount of data and flop that are linear in the dimension of the operands, for examples dot product and saxpy. Level-2 operations require a quadratic amount of data and flop, for examples outer product, outer product updates and gaxpy. Level-3 operations involve a quadratic amount of data and a cubic amount of flops, for example matrix multiplication. Since the compute to data ratio grows with problem size for level-3 operations, they are always compute-bound for sufficiently large problems, and

so level-3 operations are often themselves described as 'compute-bound'. Because of the large number of floating point operations required by level-3 operations, they tend to dominate the running time of many algorithms. This is fortunate as these operations also provide the best opportunity for acceleration by specialised hardware.

1.3.5 Sparse Matrices

It often happens in practical matrix computation that problems arise where a matrix has a large number of zero elements; such matrices are called *sparse*. Matrices which are not sparse are called *dense*. If the location of the non-zero elements are known or can be calculated, it is often possible to derive faster algorithms for operating on the matrix by avoiding computations on the zero elements. Some types of sparse matrices are shown in Figure 1.3.

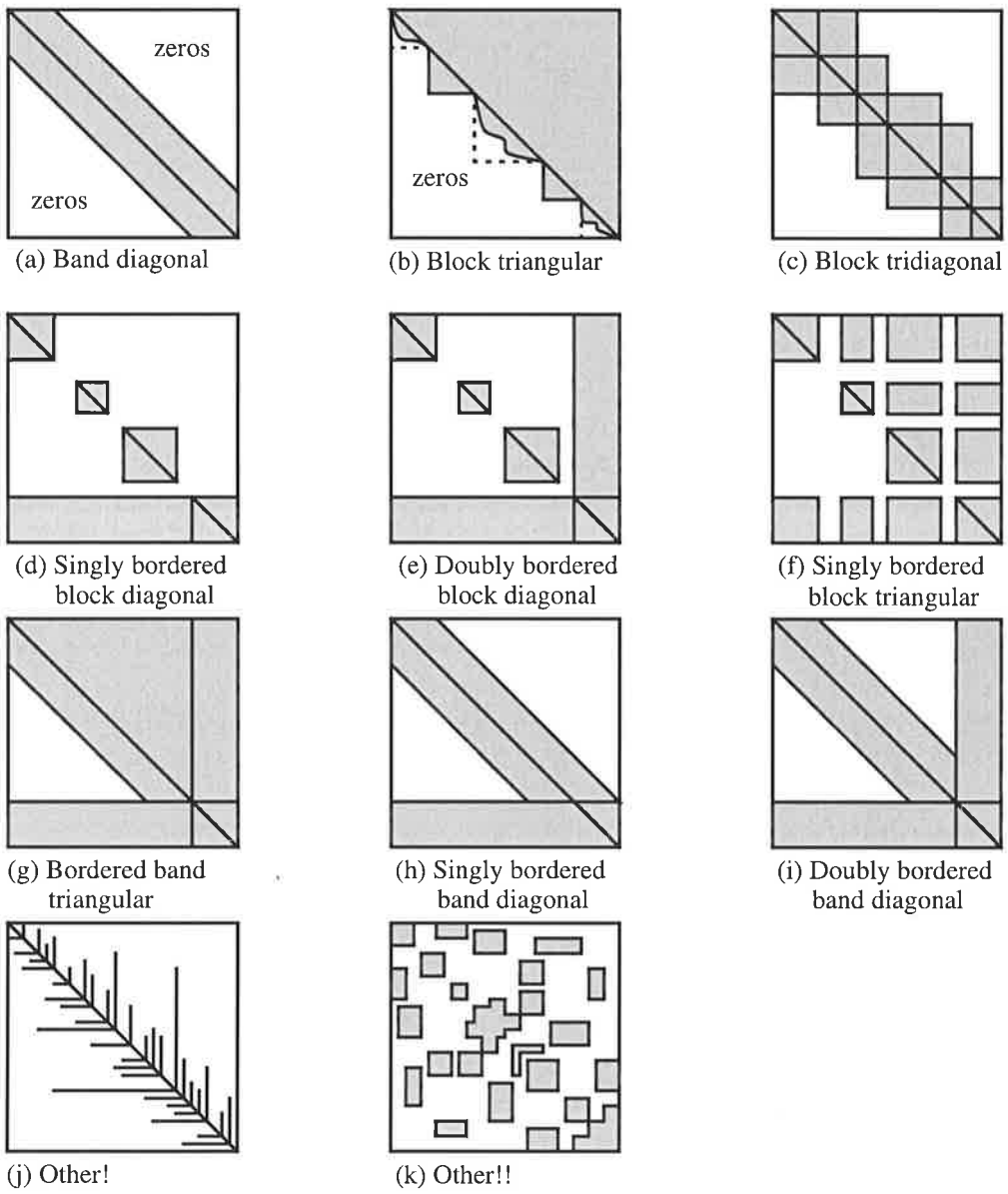


Figure 1.3: Sparse Matrix Types[Press et al. 92]

When implemented, sparse matrix algorithms often do not have the regular computational patterns that make dense matrix operations so attractive for hardware acceleration. For this reason, sparse matrices are problematic for the architecture described in this thesis.

1.3.6 Programming Matrix Algorithms

Programming of matrix algorithms can be problematic for three principal reasons.

- Numerical programming is difficult in general as subtle variation in the way in which calculations are performed can significantly effect the accuracy of the results.

- Speed is of critical importance. Optimisation is often very effective as programs tend to spend a very large amount of time in a very small amount of code. However, the optimisations are not always straight-forward.
- A wide variety of often fundamentally different architectures are used, which requires a lot of code rewriting.

To combat these difficulties, numerical applications are written using a number of numerical library kernels, which implement a range of common matrix operations[Dongarra and Walker 95]. Most fundamental of these kernels is the Basis Linear Algebra Subprograms (BLAS)[Dongarra et al. 88], which provide level 1, 2 and 3 matrix operations such as dot product, gaxpy and matrix-matrix product. Algorithms for solving linear systems are provided by LINPACK[Dongarra et al. 79] and its successor LAPACK[Anderson et al. 92].

Originally, these kernels were designed to operate on dense matrices running on scalar or vector machines, but more recently versions for parallel machines and for sparse matrices have been developed. Parallel BLAS (PBLAS) provides the standard BLAS operations for a wide range of parallel multiprocessors[Choi et al. 94b] [Choi et al. 96b][Chtchelkanova et al. 97]. ScaLAPACK[Choi et al. 96a] provides LAPACK on parallel machines. Versions of BLAS for sparse matrices are also being developed[Duff et al. 97].

Using these kernels not only provides high performance[Elman and Lee 95][Rajagopalan 98], but also aids good software engineering practice by promoting code reuse. For new architectures, it offers the opportunity to port a large amount of existing code much more easily and to achieve high performance by optimising only a small amount of standard kernel code.

Modern compiler technology is becoming increasingly sufficient to achieve optimum performance without using hand-coded assembler or machine-specific tweaks[Carr and Lehoucq 97][Bilmes et al. 97]. However, such developed compilers are unlikely to be available for radically new architectures.

1.3.7 Example Applications

Matrix algorithms form the core of a vast amount of engineering, scientific and business applications. In this section, some of these applications are highlighted and some of the underlying algorithms used are explained briefly.

The first major group applications that use matrix algorithms involve numerical

simulations of various systems. Some examples are:

- Weather forecasting
- Climate modelling
- Computational Fluid Dynamics
- Petroleum Reservoir Simulation
- Structural Analysis
- Electromagnetic Modelling

Common to all these applications is the fact that they can use any increases in computation power to produce more accurate results, consider more design scenarios, consider larger systems, and to produce the results more quickly. Thus advances in the computing power available for performing simulations are especially significant because they allow advances to be made in a range of disciplines.

These applications generally operate by expressing the behaviour of the system in terms of a matrix, or matrices, and then performing some calculation on the matrix to extract desired information about the system. Common calculations performed on matrices include, solving a system of linear equations, finding a least squares solution to a system of equations, finding eigenvalues and eigenvectors and computing matrix decompositions. A wide variety of algorithms are used to perform these calculation.

A specific example of such an application is solving linear equations arising out of integral formulations of Maxwell's equations[Forsman et al. 95]. The matrices involved are dense and large, having orders in the thousands, but they tend to have a large number of elements with relatively small absolute values. However, the large and small elements are scattered quite randomly throughout the matrix.

A second major source of applications that use matrix algorithms is signal processing. Signal processing algorithms tend to be characterised by using smaller matrices, but performing more iterations on them than algorithms used in numerical simulations. Some example are:

- Finite Impulse Response (FIR) filter
- Discrete Fourier transform
- Hartley transform
- Kalman filtering

The architecture described in this thesis is intended for operation on dense matrices. This focus is justified by two facts. Firstly, there are a wide variety of applications which can

benefit from the acceleration of matrix operations. Secondly, the regular computation and data flow inherent in dense matrix calculations, in particular the compute-bound matrix-matrix multiplication operation, are ideal for acceleration using a specialised computer architecture. Regular computation and data flow are important because they offer the possibility of improving performance by using simple parallel structures. Compute-bound operations are generally an easier target for acceleration because, as the need for complex memory hierarchies shows, memory is more difficult to accelerate than computational units. However, the importance of the memory architecture is still paramount, as will be shown in the following chapters. Thus the architecture is targeted at an area where acceleration is both useful and achievable.

Chapter 2

Performance of Matrix Computations on Different Architectures

In order to gauge the relative performance of matrix algorithms running on a specialised architecture, an understanding of the performance on existing machines is required. Accordingly, this chapter begins by discussing three classes of computer architecture: scalar architectures, vector architectures, and massively parallel processor architectures. Examination of the strengths and weakness of these machines in dealing with dense matrix problems gives an insight into the means by which a specialised matrix processing architecture may exploit the nature of the problem to extract improved performance. The chapter closes with a discussion of specialised architectures for performing matrix calculations including a brief discussion of the architecture with which this thesis is concerned, the Load/Store MATRISC architecture.

2.1 Scalar Architectures

Scalar architectures account for the vast majority of all computers in use today. Substantially all new scalar machines are built around a superscalar RISC microprocessor such as the DEC Alpha[DEC 97] or PowerPC[IBM 97], or an Intel processor such as the PentiumII[Intel 97]. The memory architecture consists of one or two levels of on-chip cache, backed by an off-chip cache and a main memory consisting of DRAM.

These microprocessors can complete one, two or four floating point operations per clock

cycle, and have clock frequencies up to 600MHz. The highest peak performance available is the DEC Alpha with 1.2Gflop/s. The achieved performance is usually much less; some examples are shown in Table 2.1.

Machine and Configuration	LINPACK Benchmark (100 × 100)		LINPACK TPP (1000 × 1000)		Peak
	Mflop/s	% Peak	Mflop/s	% Peak	Mflop/s
IBM RS6000/397 (160MHz ThinNode)	315	49%	532	83%	640
DEC 500/500 (1 processor 500MHz)	235	24%	590	59%	1000
DEC AlphaStation 600 5/333MHz	153	23%			666
Gateway 2000 G6-200 Pentium Pro	62	31%			200

Table 2.1: LINPACK Performance for Scalar Machines[Dongarra 98]

The principal advantage of scalar machines is their applicability to a wide range of problems, which in turn leads to rapid increases in performance brought about by the huge research and development budgets available. In terms of matrix algorithms, scalar machines achieve their best performance with regular dense problems, but the performance on sparse and irregular problems is still good.

Some important applications for scalar machines that use matrix algorithms are CAD, visualisation and small-scale numerical simulations. These applications are characterised by using relatively small matrices and by often being interactive.

There is a wide range of powerful tools available to facilitate programming of scalar architectures. Optimisation of numerical codes involves avoiding the pipelining hazards of specific processors, and maximising cache reuse by using block algorithms to increase the temporal locality of memory references [Agarwal et al. 94].

2.2 Vector Architectures

Vector computers, such as the original Cray machines, were the first to be dubbed supercomputers. Recent vector machines, such as the Cray T90 series, the NEC SX-4 and the Hitachi S-3800, are all multiprocessor systems containing up to 32 vector processors. Each vector processor contains vector registers and deeply pipelined functional units that

operate on them. In addition, the processor also has scalar registers and supports scalar operations. All recent vector architectures use a register/register mode of computation, with explicit load and store operations. No caching is used between the processor and main memory; instead an interleaved main memory is used that can support a very high data rate, provided that consecutive accesses occur to different banks. The reason for this arrangement is that data reuse in numeric codes, although often high, does not have strong temporal locality. However because data is accessed as vectors in a large number of cases, the interleaved memory banks can expect to be contention-free.

The peak performance for a single vector processor ranges from just under 2Gflop/s up to 8Gflop/s. The maximum number of processors per machine ranges from 4 to 32, but the fastest processors are only available in the smallest number so the total peak performance of all machines is well below 100Gflop/s.

The LINPACK results for a number of vector machines is shown in Table 2.2. The percentage of peak performance achieved is good, with over 80% of peak achieved for large problems on a single processor. As expected, scaling to a large number of processors significantly reduces the percentage of the peak speed achieved.

Machine and Configuration	LINPACK Benchmark (100×100)		LINPACK TPP (1000×1000)		Peak
	Mflop/s	% Peak	Mflop/s	% Peak	Mflop/s
Cray T94 (1 proc. 2.2ns)	705	39%	1603	89%	1800
Cray T932 (32 proc. 2.2ns)	N/A		29360	51%	57600
NEC SX-4/1 (1 proc. 8ns)	578	29%	1944	97%	2000
NEC SX-4/32 (32 proc. 8ns)	N/A		31060	49%	64000
Hitachi S-3800/180 (1 proc. 2ns)	408	5%	6431	80%	8000
Hitachi S-3800/180 (4 proc. 2ns)	N/A		20640	65%	32000

Table 2.2: LINPACK Performance for Vector Supercomputers[Dongarra 98]

Vector supercomputers achieve high performance by providing the ability to work with vectors as well as scalar quantities, resulting in performance gains in two principal ways. Firstly, very fast, deeply pipelined functional units are used to operate on the vectors, which is successful because the high overhead for the deep pipeline can be recovered over the length of the vector and because there are no data hazards present between different

elements of the vector. Secondly, each vector instruction performs a large amount of work, which means that instruction issue is less likely to become a bottleneck on performance.

Vector machines are well suited to a wide range of numerical problems because of the ease with which many problems can be expressed in vector terms. Typical applications are large numerical simulations such as weather prediction, computational fluid dynamics, seismology and structural analysis.

In general, programming a vector computer is more difficult than programming a scalar machine. The principal task is to express algorithms in terms of vectors, with vector lengths that are as long as possible. Compilers often do a good job of producing code for a vector machine[Tanaka et al. 90][Levine et al. 91][Luecke et al. 91]. When an algorithm is easily and naturally expressed in vector terms, the programming task is relatively straight forward. In this case, the semantic gap between the high level of the algorithm and the low level of the instruction set is relatively narrow. The discussion of RISC architectures in §1.1.2, described the problems of trying to improve performance by closing the semantic gap; in this case, however, it works well because vector architecture provides simple operations that are useful in a very wide range of algorithms.

Vector computers were once the most powerful machines available. The design was principally driven by the very high cost of hardware when the architecture was introduced in the 1970s, which dictated that a small number of very fast computational units was more economical than to a larger number of slower ones. Recently vector supercomputer designs have changed because of the much lower cost of hardware. For example, the Hitachi 2800 series uses four physical pipelines to implement one logical pipeline[Wong et al. 95]. Over the last decade, the use of large vector machines has been on the decline. A survey of the 500 most powerful computers installed worldwide showed that vector machines comprised only 14% of the total in June 1998[Dongarra et al. 98], compared with 69% in June 1993[Dongarra et al. 93]. The main reason for this decline was competition from massively parallel processors built around commodity microprocessors. These machines achieve higher performance and better price/performance ratios than vector architectures because they use little, or none, of the expensive custom technology used by vector machines.

2.3 Massively Parallel Processor Architectures

Massively Parallel Processors (MPPs) are parallel machines with a large number of processors. The current generation of MPPs, such as the Intel ASCI Red[Mattson et al. 96], the Cray T3E[Anderson et al. 97] and the Hitachi SR2201[Fujii et al. 97], are MIMD machines using 100 to 10000 processor nodes, built using commodity microprocessors. Memory is distributed, with each node having 64MB to 2GB. The interconnection topology is a two or three dimensional grid or torus, and the communication uses custom logic or, increasingly, commodity logic. I/O is usually carried out using special nodes connected to RAID disk arrays or high bandwidth networks.

Peak performance of MPPs is approximately the peak performance of a scalar machine based on the same microprocessor multiplied by the number of processing nodes, giving performances of 100 - 2000Gflop/s. On dense problems, such as LINPACK, these architectures achieve a large proportion of their peak performance, for example 73% for the Intel ASCI Red. LINPACK MPP results for a number of machines are shown in Table 2.3.

Machine and Configuration	R_{max}		N_{max}	$N_{1/2}$	R_{peak}
	Gflop/s	% Peak			Gflop/s
Intel ASCI Red (200MHz Pentium Pro proc. 9152)	1338	73%	235000	63000	1830
Cray T3E-900 (450MHz proc. 1320)	670	56%	128832	23184	1188
Intel Paragon XP/S MP (50MHz proc. 6768))	281	83%	128600	25700	338
Hitachi SR2201/1024 (150MHz proc. 1024)	232	76%	155520	34560	307
Thinking Machines CM-5 (proc. 1024)	60	46%	52224	24064	131

Table 2.3: LINPACK MPP Performance for Massively Parallel Processor Machines[Dongarra 98]

The problem orders required to reach peak performance for these machine are huge. A 235000×235000 matrix in double precision requires 411GB, well beyond the size of the 32- bit address space that is still used by most lower end machines. The performance of these machines is also enormous, with the Intel ASCI Red being the first machine to surpass

1Tflop/s, a figure that was once described as the 'Holy Grail' of numerical computing.

In recent years, the development of MPPs has been boosted in the USA by the Advanced Strategic Computing Initiative (ASCI) of the Department of Energy. This program is part of an effort by the US government to replace nuclear weapons testing with simulations. The goal is to build a 100Tflop/s computer by 2003-2004[Mattson et al. 96]. A significant emphasis of this program is on using "commercial-off-the-shelf" (COTS) hardware to tap into its enormous rate of performance improvement and to reduce costs.

MPPs are typically used for very large numerical simulations in all areas, such as those previously described for vector machines, but are able to handle even larger data sets.

Dividing any computation among a number of processors is difficult, because of the need to minimise communication overheads. However, the use of standard numerical kernels, as described in §1.3.6, allows software to be written for MPPs and still achieve good performance. There is a large body of work that has been carried out on the basic algorithms upon which the numerical kernels are based, for example [Huss-Lederman et al. 94][Choi et al. 96a], which has resulted in optimised kernels.

2.4 Special Purpose Architectures

Exactly what constitutes a special purpose architecture for matrix operations is difficult to define. To some extent, vector computers and MPPs have special features that can be exploited when operating on matrices. However, these architectures are not focused on matrices, and in particular they do not treat matrices as a fundamental data type in hardware.

Only architectures that provide matrix operations in hardware will be considered here. The focus will be on dense matrices and architectures that attempt to support a wide range of algorithms, but sparse matrices and architectures for specific matrix algorithms, will also considered. The bulk of the work in this area is theoretical, with very few implementations and no commercial product, to the author's knowledge.

2.4.1 Systolic Arrays

The term *systolic array* describes a class of specialised computer architectures consisting of a number of relatively simple processing elements connected in an array[Kung and

Leiserson 79][Kung 82][Kung 88]. Systolic arrays are characterised by the regular flow of data through the array: in medicine, the term systolic describes the regular contraction of the heart pumping blood through the body.

There are many advantages to systolic arrays. Communication is local, so interconnection paths are short and therefore faster and easier to build. Processing elements are simple and so can be replicated multiple times within one chip using VLSI techniques. The array provides parallel speed up by pipelining the calculation.

The first systolic architecture proposed for matrix multiplication used a hexagonal array of inner step processors, which performed a multiply-accumulate operation each time step. Such an array performing the calculation $C = AB$ is shown in Figure 2.1. Between every pair of matrix elements shown, for instance a_{21} and a_{32}, there are two zeroes inserted into the data to obtain the correct timing. For clarity these are omitted from the figure. This architecture suffers from two problems: processors are only used 33% of the time because of the zeroes inserted into the input; and the size of the input and result matrices are limited by the physical array size. The first problem can also be solved, by re-organising the array to increase utilisation of processing elements to 100%[Kung and Leiserson 79]. The second problem can be overcome by partitioning large matrices into blocks that are as large as processor array will allow. However, addition of partial results would be required to form the final result.

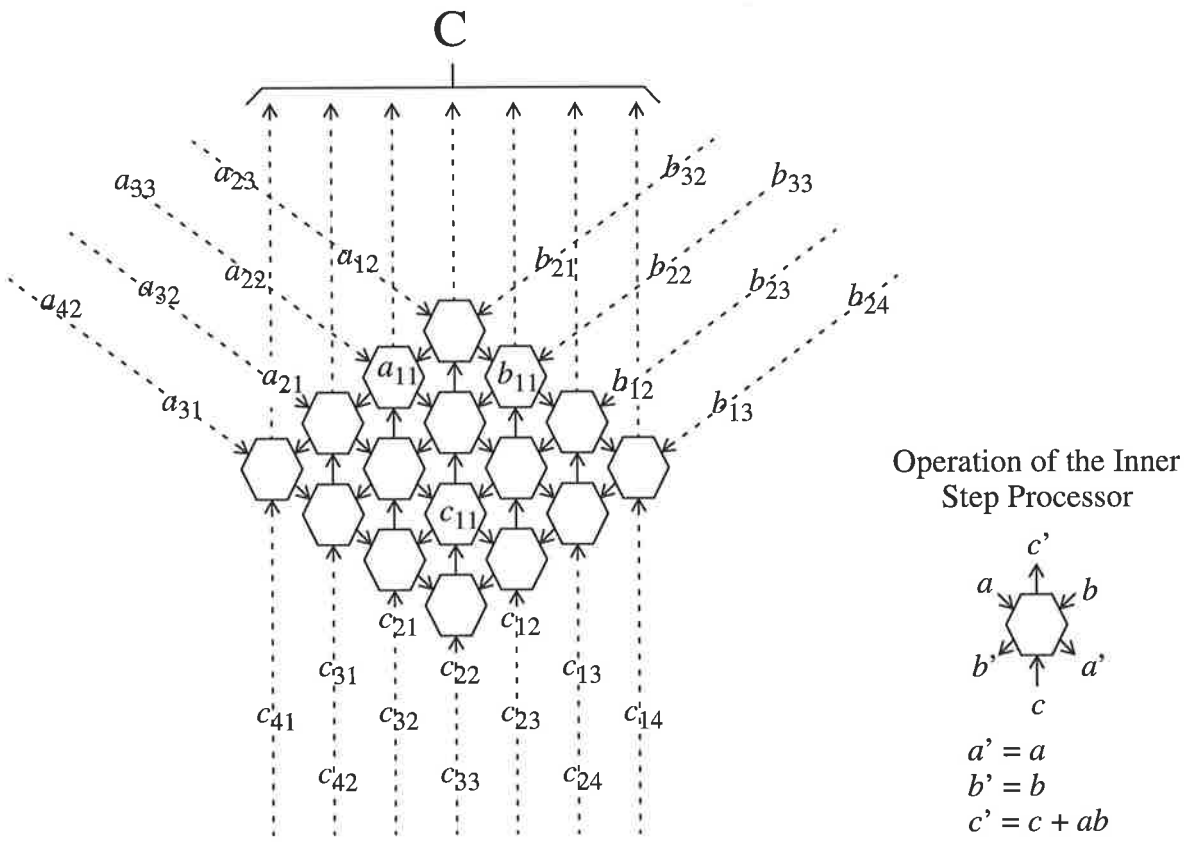


Figure 2.1: Hexagonal Matrix Multiplication Array, $C = AB$

In 1981, Whitehouse and Spieser [Whitehouse and Speiser 81] proposed a different systolic architecture to implement matrix multiplication. Called the engagement processor, the architecture was based around a mesh-connected array of inner product accumulation cells. A diagram of the cell and array operation is shown in Figure 2.2.

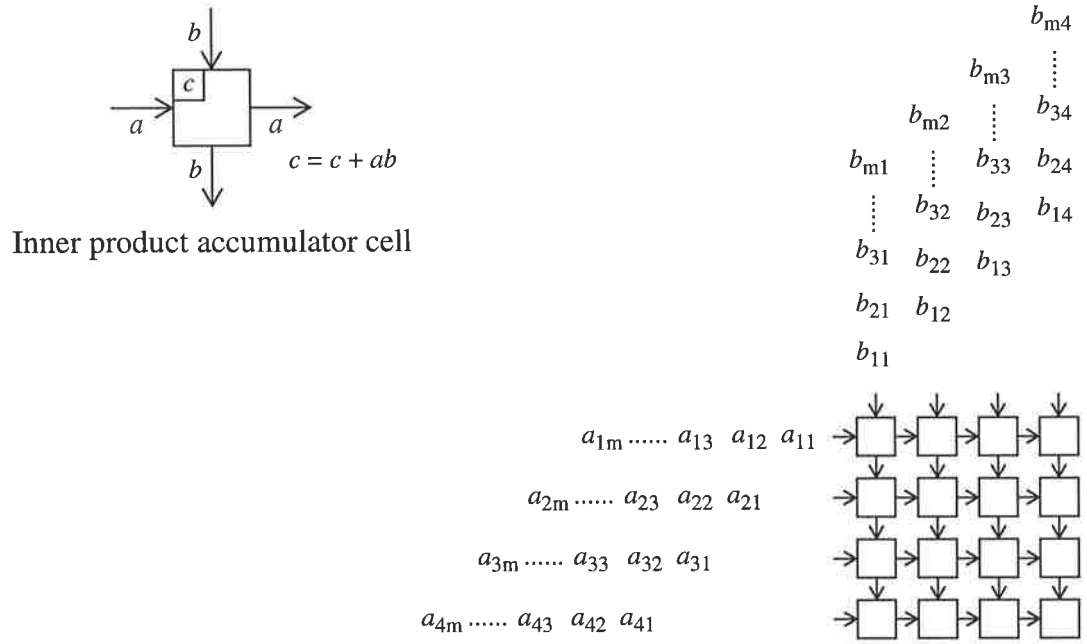


Figure 2.2: The Engagement Processor

The engagement processor accumulates the product in the array so that, when the operation is complete, the result must be extracted by some method. The size of the result is limited by the size of the physical processor array, and hence so is the size of the input matrices in one dimension, because of the conformality requirement. However the size of the input matrices in the other dimension is not limited. An $N \times N$ array of processors can be used to compute the product of an $N \times L$ and $L \times N$ matrix, where L may be arbitrarily large. By partitioning large matrices into strips as wide as the array, they maybe multiplied together with no addition of partial results being necessary, because each multiplication forms part of the final result.

In the engagement processor, the processing elements are utilised 100% of the time, except for the required skewing of the input data. However, appropriate pipelining of consecutive operations can result in full utilisation at all times.

It is possible to decrease the latency of the calculation by making alterations to the array, these alterations require data to be fed into the middle of the array and thereby destroy the regularity of the structure[Tsay and Chang 95]. It is also possible to implement this array using a bus interconnection rather than a systolic one. In this case, skewing of the input data is no longer required and the latency of the calculation is minimised. However, implementing a bus interconnection may be more difficult and does not scale well to a large

number of processors, in some circumstances.

2.4.2 SPAR

SParse matrix Architecture and Representation (SPAR)[Taylor 95] is a scheme for calculating matrix-vector products on large sparse matrices, particularly those arising in finite element method (FEM) calculations. This architecture is based on that of a vector processor with some additional hardware, and the representation used is a modification of a conventional sparse matrix data structure called Column-Major Nonzero Storage(CMNS). The SPAR architecture reflects the fact that sparse matrix calculations are always memory bound, rather that being compute bound like dense matrix-matrix multiplication.

The SPAR data structure, for a sparse matrix K, is demonstrated below. The matrix is represented as two vectors. $\vec{K}_V$ holds the non-zero elements of K in column-major order, with zero elements separating the elements of one column from the next. $\vec{R}$ has one element for each element in $\vec{K}_V$, which is the row number of that corresponding element. The elements of $\vec{R}$ which correspond to zeros in $\vec{K}_V$, shown underlined in (2.1), store the column number of the column of data that follows.

$$K = \begin{bmatrix} k_{11} & 0.0 & k_{13} & 0.0 & 0.0 \\ 0.0 & k_{22} & 0.0 & 0.0 & k_{25} \\ k_{31} & 0.0 & k_{33} & 0.0 & k_{35} \\ 0.0 & 0.0 & 0.0 & k_{44} & 0.0 \\ 0.0 & k_{52} & k_{53} & 0.0 & k_{55} \end{bmatrix} \tag{2.1}$$

$$\vec{K}_V^T = [k_{11}, k_{31}, 0.0, k_{22}, k_{52}, 0.0, k_{13}, k_{33}, k_{53}, 0.0, k_{44}, 0.0, k_{25}, k_{35}, k_{55}]$$

$$\vec{R}^T = [1, 3, \underline{2}, 2, 5, \underline{3}, 1, 3, 5, \underline{4}, 4, \underline{5}, 2, 3, 5]$$

To compute the matrix vector product, $\vec{v} = K\vec{p}$, the following algorithm is used.

```
column = 1
for index = 1 : Nnz + N - 1
        if (KV[index] == 0.0) then
                column = R[index]
        else
                v[R[index]] = v[R[index]] + KV[index]*p[column]
        end if
end for
```

This algorithm operates over a single long vector and so, in a vector architecture, it has the potential to operate at high speed because the startup overhead can be amortised over the

long vector length. The problem for a normal vector architecture is the conditional branch within the loop and the indirect reference to $\vec{v}$. To solve these problems, the SPAR architecture uses the data path shown in Figure 2.3.

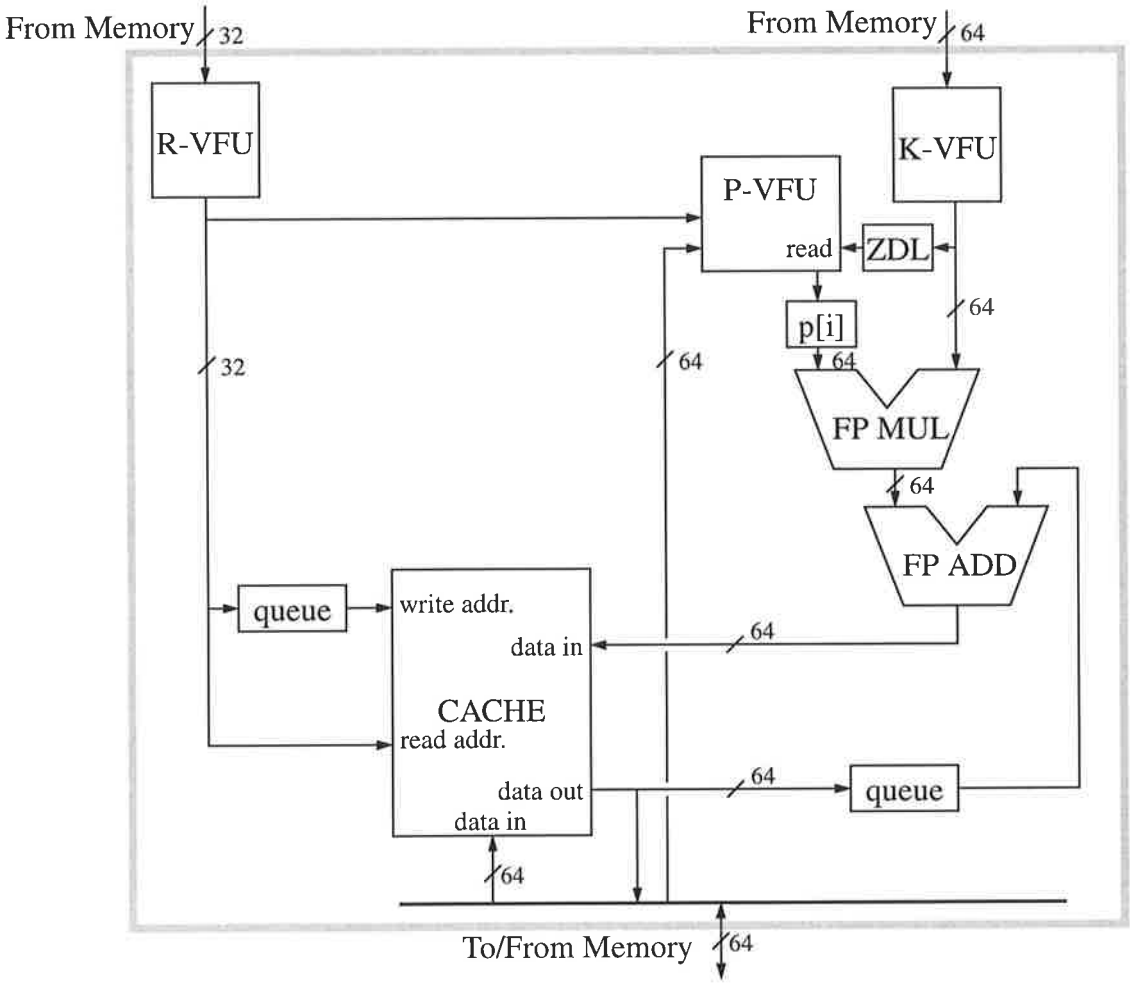


Figure 2.3: The SPAR Data Path

The computation is performed by a vector floating point multiplier and vector floating point adder. The vectors $\vec{K}_V$, $\vec{R}$ and $\vec{p}$, which are accessed sequentially, are read using vector-fetch-units (VFU) that access data directly from an interleaved main memory. Zero Detection Logic (ZDL) is used to latch in the next element of $\vec{p}$ based on the column address in $\vec{R}$. The elements of $\vec{v}$, which are accessed somewhat at random, are stored in a cache that has the addresses supplied by the R-VFU. Because the sparse matrices involved in FEM are banded, that is, they have all their elements relatively close to the diagonal, accesses to the cache have strong locality and so it can achieve very high hit rates with only a modest cache size. In addition, by fetching the addresses in $\vec{R}$ in advance of the data in

$\vec{K}_V$, the latency of a cache miss can be hidden, giving an effective hit rate of 100%. A potential read after write (RAW) data hazard exists when accessing $\vec{v}$, but this can be avoided by suitable preprocessing of the SPAR representation.

Simulation of the architecture showed that it achieved 96% utilisation of the floating point units for matrices arising out of 3-D FEM calculations. The architecture can also be used for forward and backward substitution by making only minor changes.

2.4.3 MATRISC and SCAP

The MATRISC (MATrix RISC) processor was developed by Marwood[Marwood 94] as a complete architecture for accelerating matrix algorithms. The architecture was particularly intended to act as a matrix coprocessor for a scalar RISC processor, possibly as a node of a larger parallel machine. The fundamental philosophy behind the design was to provide a small set of matrix primitives to operate directly on matrix data in hardware, which would provide a general purpose machine for accelerating a wide range of matrix algorithms. To achieve this aim, MATRISC provided both computational units **and a memory architecture** that directly support matrices as a data type. This architecture, shown in Figure 2.4, will be referred to as the Memory/Memory MATRISC architecture, to distinguish it from the architecture presented in this thesis.

Computation in MATRISC is performed by an engagement processor with enhancements to allow elementwise matrix operations such as addition. The operation to be carried out on a particular piece of data is encoded with that data and travels with it across the array. Elementwise operations are carried out using only the processing elements on the leading diagonal of the array.

Data is transferred between the array and memory using three address generators, two for the input operands and one for the result. The address generators can produce a range of address patterns for accessing different types of matrices. To increase available memory bandwidth, there are two data caches, one for the X input and one for the Y input to the array.

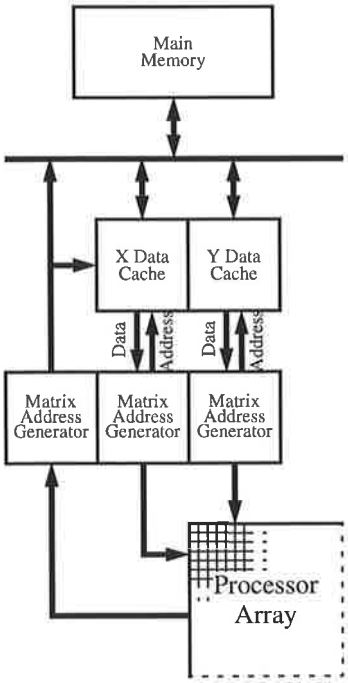


Figure 2.4: The MATRISC Memory/Memory Architecture

The matrix address generator uses an address calculation of the form

$$n = base\_address + \langle n_1\Delta_1 + n_2\Delta_2 + n_3\Delta_3 + n_4\Delta_4 \rangle_q \quad \text{where} \quad n_i \in 0 \ldots N_i - 1 \tag{2.2}$$

To sequentially address all elements of the matrix in some order determined by the constants Δ_1, Δ_2, Δ_3, Δ_4 and q, the expression for the address n is evaluated for all values of n_1, n_2, n_3 and n_4 in order. Note that each address can be generated from the previous one using only addition. The modulo operation, denoted $\langle\ \rangle_q$, is performed using a conditional subtraction of q at each calculation, rather than using division as would be required in general. N_1 and N_2 represent the number of columns and rows in the matrix, N_3 and N_4 are higher dimensional analogues.

This address generator can provide access to the following types of matrices [Marwood 94]:

- Normal and transpose matrices
- Diagonal, constant and circulant matrices
- Prime factor mapped matrices
- Sub-matrices and partitioned matrices.

For simple matrix mappings, the address generator need only be two dimensional. Extra dimensions are used for higher dimensional prime factor mappings and for partitioned

matrices, which are extremely importance since a matrix must be partitioned if it is larger then the size of the processor array.

Some example address sequences are shown in Table 2.4 with the appropriate address generator co-efficients. The examples are two dimensional for simplicity.

Matrix Type	base	Δ_1	Δ_2	N_1	N_2	q	Address Sequence
Normal	0	1	5	5	3	15	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
Transposed	0	5	1	5	3	15	0 5 10 1 6 11 2 7 12 3 8 13 4 9 14
Circulant	0	1	4	5	3	5	0 1 2 3 4 4 0 1 2 3 3 4 0 1 2
Constant	0	0	0	5	3	1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Prime Factor	0	3	5	5	3	15	0 3 6 9 12 5 8 11 14 2 10 13 1 4 7
Submatrix	6	1	5	2	2	15	6 7 11 12

Table 2.4: Examples of Two Dimensional Address Generation

2.4.3.1 The SCalable Array Processor (SCAP)

The SCalable Array Processor (SCAP) was built under a contract with the Australian Department of Defence to demonstrate the feasibility of the MATRISC concept[Clarke et al. 92]. The processor contained a 20×20 array of processing elements and was configured as a coprocessor attached to the SBus of a SUN SPARCStation 1 workstation.

The processing elements were fabricated in 1.2μm silicon technology, with an array of 5 $\times$ 4 elements requiring 265000 transistors, on each chip. Single precision IEEE floating point number format was used and each processing element was capable of 1Mflop/s. Communication used bit serial links, both between processing elements and between chips. The chips were first packaged individually, and in later versions on a multichip module. The address generators and data formatters were implemented as a separate chip. A 128kB cache was used.

Measured performance of the SCAP was around 115Mflop/s for a 500×500 matrix multiplication, falling to approximately 78Mflop/s for 100×100 product. The performance fell well below the theoretical peak for the architecture of 400Mflop/s because of overheads involved in the SBus used to connect it to the host computer and by the interface to the operating system.

2.5 Evolution of MATRISC

Since the original conception of the MATRISC architecture[Marwood 94], there has been a considerable amount of work done to improve the architecture. Most of this effort has concentrated on the memory architecture although modifications to the processor array have also been suggested. This emphasis on improving the memory architecture reflects that fact that although the architecture focuses on the compute-bound operation of matrix multiplication, it is memory-bound operations, such as matrix addition, that limit the overall performance.

To examine the reasons for various memory architecture design choices, the compute to memory bandwidth ratio of the MATRISC processor will now be discussed. Consider a $p \times p$ array of processing elements performing a matrix multiplication, which operates with a cycle time of τ seconds. As each processing element performs a multiplication and an addition every cycle, this results in a computational rate R of

$$R = \frac{2p^2}{\tau} \tag{2.3}$$

At each cycle, $2p$ operands are required at the edge of the array. Thus the *bandwidth*, that is, the number of operands that must be supplied by the memory architecture per second, denoted B, is given by

$$B = \frac{2p}{\tau} \tag{2.4}$$

Substituting (2.4) into (2.3) leads to

$$R = Bp$$

This result shows that the computational rate is the product of the bandwidth and the number of processors in each array dimension. Thus if the bandwidth is kept constant, the computation rate can be increased by increasing the number of processors. It was one of the expected benefits of the MATRISC processor that high performance could be achieved by simply scaling the size of the processor array. For instance, consider doubling p, which gives four times as many processors. If a constant bandwidth is maintained, the cycle time must be increased by a factor of two, yielding a doubling of the total computational rate.

Thus it would seem that the performance of the MATRISC architecture could be increased to very high levels simply by using a large array of slow processing elements.

Unfortunately, this is only the case for algorithms that spend 100% of their time multiplying large matrices. Algorithms that use matrix addition or use matrices that are smaller than the processor array will run at much less than peak speed.

The speed of matrix addition on MATRISC is limited only by the bandwidth; the size of the processor array makes no difference. On a $p \times p$ array, matrix multiplication uses all of the processing elements, where each performs a multiply-accumulate operation every cycle. Matrix addition uses just the p processing elements on the diagonal, where each performs only an addition operation every cycle. This means that, in terms of floating point operations per second, addition is $2p$ times slower than multiplication on the MATRISC architecture.

Matrix multiplication on matrices smaller than the array is also a problem. Consider the formation of a $p/2 \times p/2$ product on a $p \times p$ array. If the data moves through the array at the same speed as for a full sized product, then only a quarter of the available computational cycles and half the available bandwidth will be used. If all the bandwidth could be harnessed, then the performance would double. To allow this, Marwood proposed the concept of a *constant bandwidth array*, which contains a large number of slow processing elements with a smaller number of progressively faster ones, shown diagrammatically in Figure 2.5 [Marwood 94]. Large matrices would use all the processing elements operating at a slow speed, while smaller matrices would use a small number of fast processing elements, resulting in the memory bandwidth always being fully utilised. Even with the constant bandwidth array, however, performance of a half size multiplication is still only half that of a full sized product. This means that, along with addition, multiplication of small matrices will limit performance of the MATRISC architecture, particularly of larger arrays.

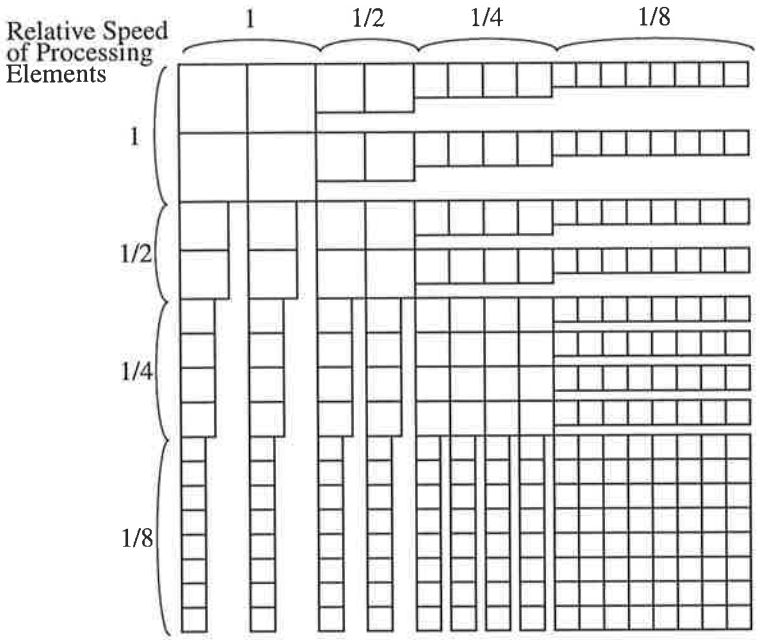


Figure 2.5: The Constant Bandwidth Array

A further problem encountered with the constant bandwidth array is that fabricating processing elements in a range of speeds is difficult to do efficiently in terms of chip area. For example, a multiplier that is eight times slower than the fastest multiplier is very much larger than one eighth of the area required by the fastest. Marwood proposed a systolic ring architecture for the multiplier accumulator, which allowed speed to be traded for area by inserting more, or fewer, cells in the ring. However, implementation studies showed that the area required for control overhead resulted in the area-time product being better for the fastest and largest processing elements[Beaumont-Smith 95][Beaumont-Smith et al. 96].

One important reason that justifies the focus of the MATRISC architecture on matrix multiplication is that high memory bandwidth is expensive and difficult to achieve, while computational units are relatively cheap. Because of the mismatch between the memory bandwidth and computational requirement of matrix multiplication, and because it varies with matrix size, the memory bandwidth and computational power cannot both be fully utilised at the same time for all operations. The operation for which compute/memory bandwidth balance occurs is multiplying matrices that are the same size as the processor array. For addition and for smaller matrix multiplication, much of the computational power of the processor will lie idle, effectively being wasted for those operations. Given that computational power is in general cheaper that memory bandwidth, the option of "wasting" computational rather than memory capability should achieve a better cost/performance

ratio. Conversely, note that some bandwidth is wasted by multiplication of matrices larger than the processor array size, because less writing of results occurs.

2.5.1 Faster Address Generation and Interleaved Memory

In 1994-5, further work on the MATRISC concept was carried out by Tim Shaw at the University of Adelaide[Shaw 95]. This work focused on two problems; increasing the available memory bandwidth to the array, and connecting multiple MATRISC processors to form a very powerful matrix supercomputer.

The basic design of the architecture was not altered significantly, but a new design for the address generator using carry free arithmetic was presented, and a method of interleaving main memory, similar to that used in vector architectures, was proposed. Simulations predicted that these changes would allow the fetching of one operand every 10ns on both the X and Y inputs to the array, giving a peak computational rate of 4Gflop/s for a 20×20 array and 8Gflop/s for a 40×40 array.

Shaw also proposed a multi-processor based on a hypercube connected array of MATRISC processors and suggested algorithms that could be used for matrix multiplication and Gauss-Jordan elimination.

2.5.2 Parallel Address Generators with Multiple Memory Banks

In late 1995 work on the MATRISC project was renewed. A number of new ideas were proposed to increase the performance of the memory architecture.

One approach that was investigated was to modify the architecture to use a large number of address generators operating in parallel, connected to a large number of main memory banks via a switching network[To 96]. Each memory bank consisted of a RAMBUS channel and an associated controller, which attempted to group accesses into single RAMBUS transactions to maximise throughput. The design was abandoned because of the difficulty of designing an appropriate switching network and the latency of memory accesses.

2.5.3 Multi-Dimensional Memory Architecture

An alternative method for increasing the performance of the MATRISC architecture, named the Multi-Dimensional Memory Array (MDMA), is being developed at the University of Adelaide[Beaumont-Smith et al. 97a][Beaumont-Smith et al. 97b]. This

architecture is a radical departure from previous ones, as it increases performance by providing each processing element with it own memory and allowing more flexible communication between processing elements. This arrangement is in contrast to the original *edge-fed* architecture where all data reached the processing elements from the edge of the array.

The MDMA has a significant advantage over edge-fed array architectures because elementwise operations such as addition are as fast as matrix multiplication. This is due to the fact that during matrix addition all the processing elements can perform calculations using their local data, rather that just those on the leading diagonal. However, the amount of memory that can be integrated with each processing element is limited and can significantly reduce the maximum problem size. The added complexity of the architecture is also likely to increase the difficulty of programming.

This architecture represents the decision to waste memory bandwidth rather than compute cycles, and thus will often have a low utilisation of memory bandwidth even though the utilisation of computational power is high. Therefore it is reasonable to expect that the MDMA will achieve higher performance than any edge-fed array, but at higher cost. However, because of the differences in control complexity, a VLSI implementation of both types would be needed to quantify the relative performances.

2.5.4 Load/Store Architecture

The Load/Store MATRISC architecture, the topic of this thesis, uses an edge-fed array with a fundamentally new memory architecture in an effort to significantly improve memory bandwidth. The architecture, shown abstractly in Figure 2.6, uses two large matrix registers, one on each side of the array, which can supply data in parallel to the edge of the array. The registers are constructed of SRAMs with a set of interface chips. Each SRAM supplies data to one row or column of the processor array. Use of SRAM allows the registers to be large enough to hold, and operate on, large matrices (up to the order of several hundred).

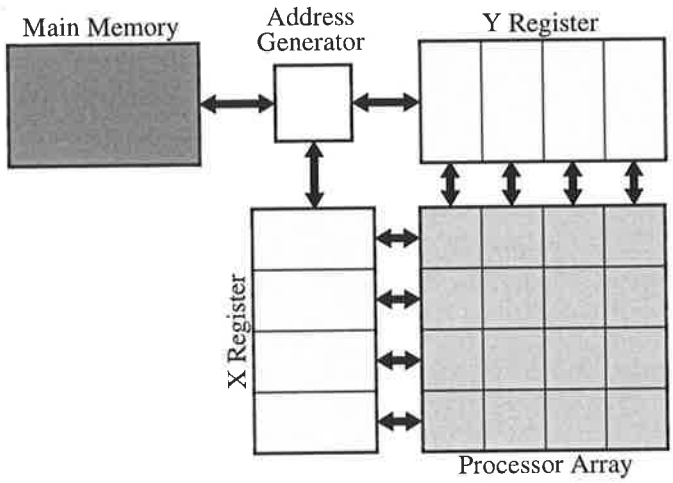


Figure 2.6: The Load/Store MATRISC Architecture

Data is transferred between main memory and the registers using a modified address generator under explicit programmer control, hence the name of the architecture. The address generator allows the main memory DRAMs to be accessed in large sequential blocks, regardless of the mapping being applied, which gives the best performance from the main memory.

This architecture completes the RISC nature of MATRISC by providing large general purpose registers and a register-register mode of computation with explicit load/store instructions. Significant advantages of this architecture are the high bandwidth provided to the processor array and the fact that it scales with the size of the array, assuming that the bus can maintain a constant cycle time. The grounds for this assumption are discussed in §3.4. However, each of the registers is divided so that, during any one compute operation, not all the data can reach all the processors. This restriction is in sharp contrast to the original architecture, where a flexible mapping could be applied to each matrix on every operation.

The Load/Store architecture is the first design for an edge-fed MATRISC system that is both realisable and provides sufficient bandwidth to support the speed of current floating point computational units. Compared to the MDMA, the Load/Store architecture is considerably lower in cost, can more easily operate on large matrices and is simpler to program.

Chapter 3

The Load/Store Architecture

The proposed Load/Store MATRISC architecture is based on a mesh connected array of inner product cells, configured to perform outer product accumulation. Such an array provides a solid basis for a specialised matrix processor. It is derived from Marwood's Memory/Memory MATRISC architecture but differs in a number of aspects, most significantly in its memory architecture.

The fundamental basis of the MATRISC philosophy can be expressed thus, "*Providing matrix primitives in hardware is an effective means of accelerating many numerical applications because,*

- *the algorithms used in many numerical applications are naturally expressed in matrix terms, and,*
- *matrix primitives implemented in hardware, particularly the compute bound operations of matrix-matrix multiplication, can achieve very high speeds because of the inherent parallelism*".

Furthermore, the processor is regarded as operating on matrix data. This distinction is important because it justifies classification of the architecture as RISC, because although at the scalar level the operations the architecture performs are very complex, when regarded as matrix operations they are very simple. There is also a logical progression from scalar architectures, to vector architecture, to matrix architectures. This suggests that MATRISC will have many of the benefits of vector architecture, such as lessened chance of instruction issue bottleneck, and applicability to a wide range of algorithms by supporting a fundamental numerical data structure in hardware.

The Load/Store architecture expands on this basic philosophy by introducing the major RISC features missing from the Memory/Memory architecture, which are that all operations

are performed out of a large set of general purpose registers, and that all transfers between the registers and main memory are under the direct control of the programmer.

It is important to understand that the classification of the registers is based on their function rather than the type of memory technology used to construct them.

3.1 Outline of the Load/Store Architecture

The original MATRISC architecture as described by Marwood (see §2.4.3) and implemented in the SCAP project was a memory/memory architecture. Following the work of Beaumont-Smith, Shaw and To (see §2.5.1 and §2.5.2), it appeared that major changes to the architecture were needed to achieve optimum performance with current technologies. Specifically, a memory architecture that could provide a much greater bandwidth was required.

3.1.1 Impetus behind the Load/Store Architecture

In the Memory/Memory architecture, the address generator has to generate an address and fetch one operand for each row, or column, of the processor array in the same amount of time taken by the processing elements to perform one multiply-accumulate operation. Hence for a $p \times p$ array, the memory system has to produce operands p times faster than the time taken for the processing elements to perform one multiply-accumulate operation. This was a feasible situation when the processing elements were using relatively slow computational units and narrow communication channels were used to carry the data across the array. In fact, for matrix multiplication the performance of the system increases, for a constant memory bandwidth, when using a larger array of slower processing elements. However, as was described in §2.5, slow processing elements do not make the best use of silicon area. The best area-time products are achieved for large fast computational units, and in this case, the Memory/Memory architecture does not provide sufficient bandwidth. For example, assuming a pipelined processing element operating with a 5ns cycle time and a small 5×5 processor array, the Memory/Memory architecture would need to produce one operand every 1ns, which is simply not possible with current technology.

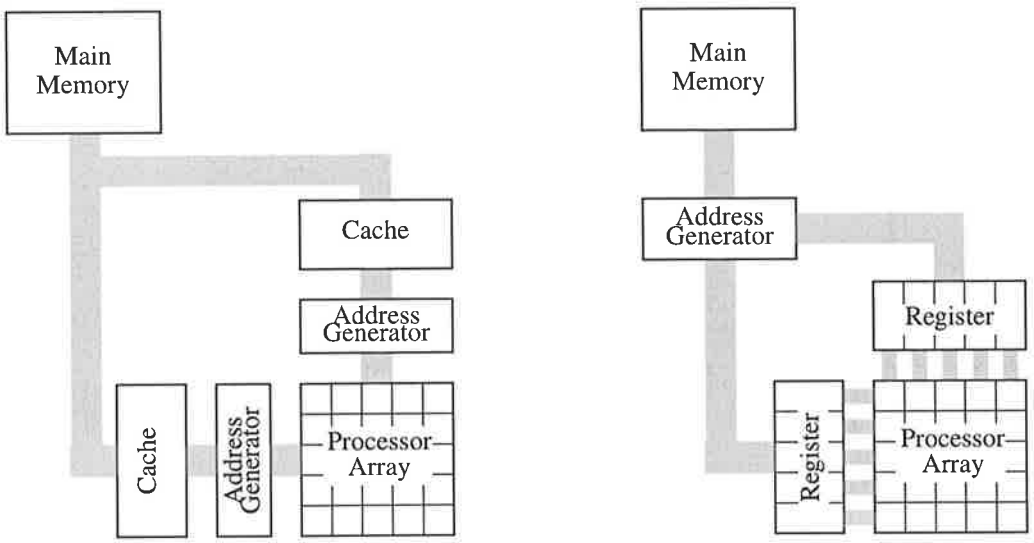
A second important reason for increasing the memory bandwidth is that very few algorithms consist entirely of matrix multiplication. Most also use bandwidth-limited

operations such as matrix addition. Increasing the array size has no effect on the performance of these operations, so only a faster memory system can accelerate them.

The fundamental cause of the lack of the memory bandwidth, and hence performance, in the Memory/Memory architecture is the serial nature of the address generator. To achieve a significant increase in performance, a parallel memory architecture is required at the edge of the array. However, due to the memory/memory mode of access the address generators make this very difficult. The situation is equivalent to that of requiring parallel accesses to a central memory, as occurs in centralised memory MIMD machines. Some attempts have been made at designing such a system for MATRISC (see §2.5.2), but they were not successful and the studies concluded that this approach was unlikely to succeed.

The Load/Store architecture proposes to solve this difficulty by making two changes. The first change is to use a bank of parallel caches at the edge of the array and to move the address generator from the cache/array boundary to the main memory/cache boundary. Each row and column of the array will then have a direct connection, via a bus, to one slice of the parallel cache. This greatly increases the data bandwidth available at the side of the array, and does so in a way that scales with the size of the array. The second change is to adopt a load/store method of computation and to regard the caches as matrix registers. All computations are then carried out from explicit locations in the registers, and results written back to the registers. In addition, all movement of data between the main memory and the registers will be performed explicitly by the programmer using load and store operations. Although the registers use the same SRAM memory technology as the caches in the Memory/Memory architecture, they can properly be regarded as registers because of the way they function.

The difference between the Memory/Memory and Load/Store approaches is shown diagrammatically in Figure 3.1. Note that Figure 3.1(a) is an abstraction of the Memory/Memory architecture shown in Figure 2.4. Figure 3.1(b) is an abstraction of Figure 2.6, which is in turn a simplification of Figure 3.2 which follows later. Note that only one address generator is required in the Load/Store architecture between the registers and main memory.



(a) Memory/Memory (b) Load/Store

Figure 3.1: Comparison of Memory/Memory and Load/Store Architectures

The most significant advantage of the Load/Store architecture is that it greatly increases the data bandwidth to the processor array and hence the possible rate of computation. However the approach had several disadvantages, which will now be outlined.

The principal disadvantage is that the flexibility of the system is reduced because it is not possible to apply a flexible address mapping to every matrix operand. However, it is possible to reorganise most algorithms so that they only need to perform a mapping when the data is moved between main memory and the registers. Detailed simulations to show that such re-organisation can be done successfully for a range of applications are described in Chapter 5.

Furthermore, the Load/Store architecture is considerably more complex than the Memory/Memory architecture, requiring many more components. In particular, some form of address generation is required between the registers and processor array. This is necessary because each register will generally hold more than one matrix, and because some operations will be applied to only part of a matrix held in the registers. In both of these cases an address generator is required to extract only that part of the register that is required for the operation. This address generation will be much simpler than that required between main memory and registers, because once loaded into the register the matrix will have already had a mapping applied to it and because each address generator will be dealing with only a slice of the matrix.

The last major disadvantage is that the address generator is potentially a scalar

bottleneck, just as it was in the Memory/Memory architecture. Whether it effects the performance of the array depends on how much data must be transferred to and from main memory. Analysis of this problem is one of the important aspects of the simulations performed in the following chapters.

In line with the MATRISC philosophy the architecture is usually regarded as manipulating matrix data types. However, the architecture can be alternatively viewed as a parallel SIMD machine operating on scalar data. In this case, the change from the Memory/ Memory architecture to the Load/Store architecture is a move from shared memory to distributed memory, with the concomitant advantages, such as improved performance and scalability, and disadvantages, such as increased complexity and difficulty of programming.

3.1.2 Load/Store Architecture Outline

A functional block diagram of the complete Load/Store architecture is shown in Figure 3.2, with a 5×5 processor array. This figure is a more detailed version of Figure 2.6 and Figure 3.1(b).

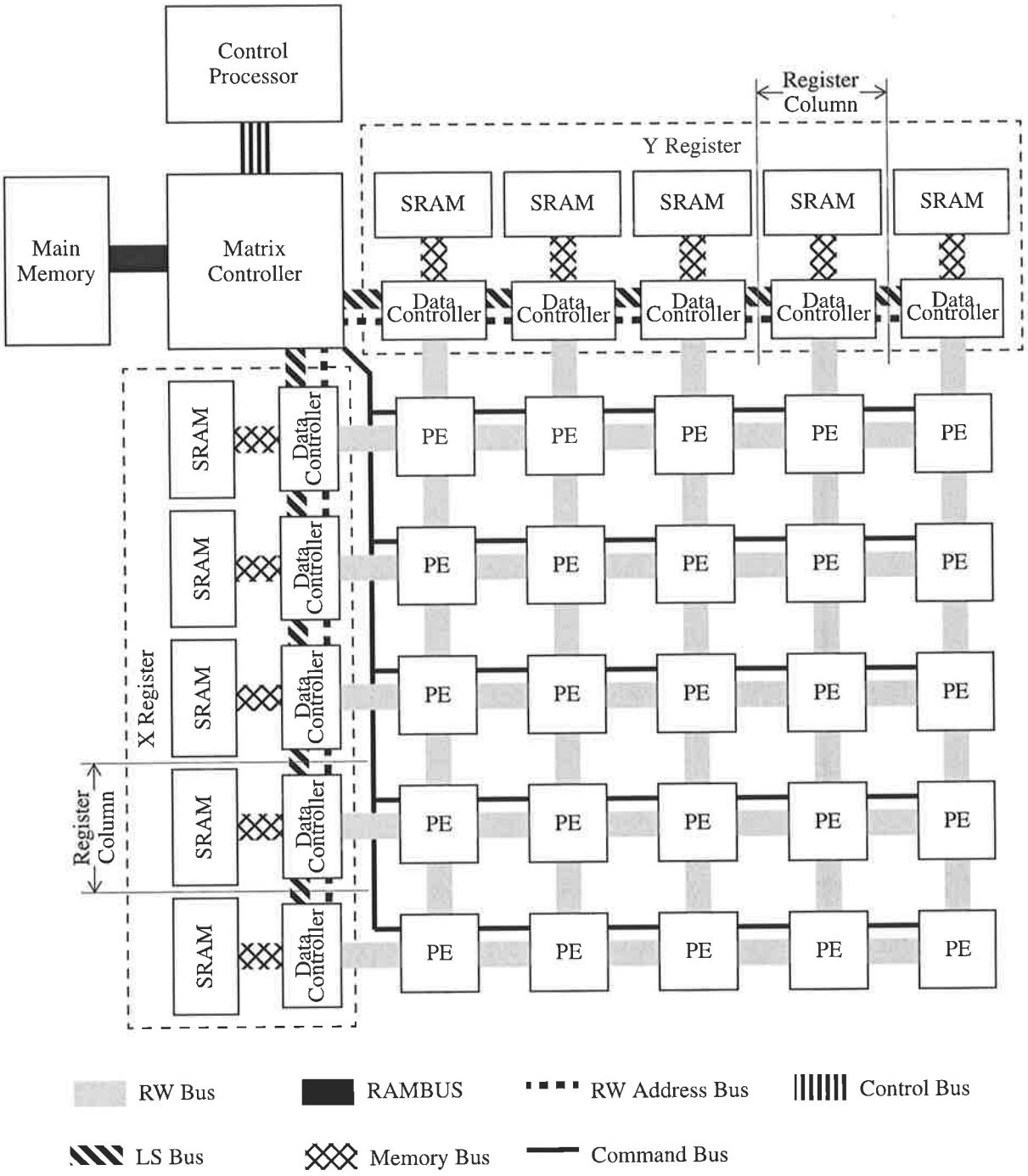


Figure 3.2: The Load/Store MATRISC Architecture

The architecture consists of six blocks: a processor array, two registers, a central matrix controller, a supervising control processor and large main memory.

The processor array is a mesh of processing elements (labelled PE) connected via buses. Each processing element can perform double precision IEEE multiply-accumulate, multiply, add, divide and square root operations. Each processing elements contains an accumulator, which is used to store the partial sum during matrix multiplication, and a result

register, which holds a calculated result while it is waiting to be written back to the registers. The result register allows another calculation to begin during the writeback phase of the preceding instruction. The operation performed by all the processing elements in the array is the same, except that the processing elements are aware of their position in the array and some commands apply only to some processing elements.

The registers each consist of a number of *register columns*; one register column for each row of the processor array in the case of the X register, and one register column for each column of the processor array, in the case of the Y register. Each register column consists of one, or more, SRAM chips connected to one row or column of the processor array through a data controller. This partitioned arrangement restricts the flexibility of the architecture, as not all the data in the registers is directly available to all processing elements, but it provides very high data bandwidth in a manner which scales with array size. Data can still be passed to any processing element, but this requires the overhead of an otherwise unnecessary copy operation between the registers. All data transfers between the processor array and the registers involve the passing of an entire *register row*, consisting of one operand from the same address within each register column. It is important to note that a matrix stored in a register may have either its rows or columns aligned to the register rows; this will depend on the algorithm and the mapping applied by the address generator. To avoid ambiguity, the terms *register row*, *register column*, *matrix row* and *matrix column* will be used where necessary.

In addition to the parallel reading and writing of register rows, the registers also support a single port for load/store accesses to main memory. Load/store transfers are performed by the matrix controller, which contains two address generators; one to generate the main memory addresses and one to generate the register addresses.

The matrix controller coordinates the operation of computation within the processor array, and the load/store transfers between main memory and the registers. Computations are controlled by generating an instruction that is sent on the command bus to all processing elements within the array, and by generating the addresses of the register rows that must be transferred to and from the array. The instructions specify what command is to be performed and the size of the operands. The command bus also communicates status information back to the matrix controller from the processing elements over a wire-OR bus line.

Load/store transfers are controlled by generating addresses in main memory and in the

registers, and by buffering the data moving between them.

The control processor is a tightly integrated scalar processor, which controls the overall operation of the architecture. It has its own program and data memory, and operates by accessing a number of control registers in the matrix controller.

The main memory is a large bank of DRAM storage space for data that can not be held in the registers. RAMBUS DRAM memory is preferred because of its generally superior abilities.

3.1.3 System parameters

The following symbols are used to denote architectural parameters, some of which have yet to be defined.

- p - physical array size (grid of $p \times p$ processing elements)
- V - maximum virtual factor
- v - virtual factor
- P - virtual array size ($=vp$)
- s - register column size in operands
- S - size of each register ($=sp$)
- B_{LS} - load/store bandwidth in operands per second
- B_{DC} - data controller bandwidth for reads in operands per second
- B_{CP} - control processor execution bandwidth in instructions per second
- F - load/store bandwidth factor ($=B_{DC}/B_{LS}$)
- G - control bandwidth factor ($=B_{DC}/B_{CP}$)

The remainder of this chapter describes the various aspects of the Load/Store architecture in more detail.

3.2 Processor Array Operations

This section is concerned with single array operations. The ways in which multiple array operations can be combined to perform a matrix operation on operands that exceed the processor array size is covered in Chapter 4.

In line with the reduced instruction set aspect of the MATRISC philosophy, there are only a small number of operations that the processor array can perform. These can be

divided in three groups: conformal multiply operations, elementwise operations and test operations.

The computational operations performed by the processor array are controlled by the matrix controller, which generates addresses and command signals that are sent to the registers to read rows of data and pass them to the array, and to write back the rows of result data from the array. Simultaneously it sends instructions on the command bus to all the processing elements in the array instructing them what to do with the data.

The most fundamental timing interval for the architecture is the *array cycle*. During each array cycle one row of data is transfer to the processor array from each register bank, one row of data is transferred from the processor array to each register bank, and one computation is performed in the processing elements. The nature of the computation for the different types of array operation are explained in the following sections.

3.2.1 Conformal Multiply Operations

Multiply operations are performed using outer product accumulation, as shown in Figure 2.2. First, read address generators for the registers are set up to transfer the correct register rows into the array. The processing elements then perform a multiply-accumulate operation in every array cycle, and when the input is finished, the data in the accumulator is transferred to the result register. Finally, the results are written back to the registers one row at a time. The writeback phase occurs in parallel with the reading and computation of the next operation. Note that this description omits any mention of the virtual processing capability, which is described in detail in §3.3.

There are two versions of the multiply operation: ordinary multiply and chain multiply. The chain operation is exactly the same as the normal operation except that the accumulators in the processing elements are not reset to zero when the operation begins, which means that the second product is added to the first. Chain operations are particularly useful when performing matrix multiplication with complex valued matrices.

Figure 3.3 shows the timing of data transfers to and from each processing element during a single multiplication operation.

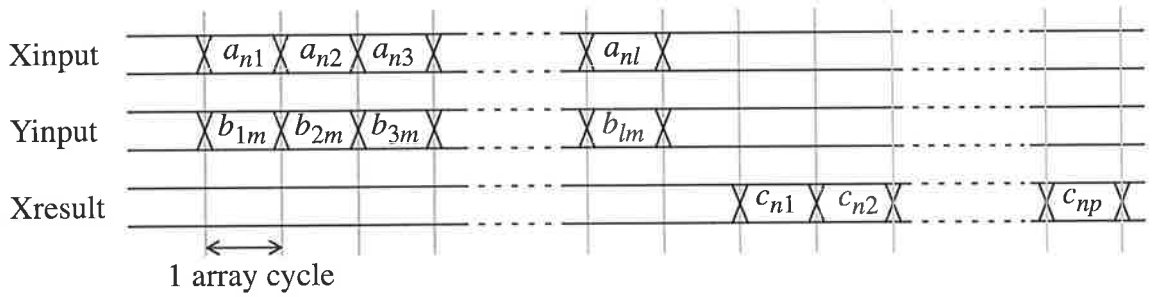


Figure 3.3: Multiply Operation Timing

Note that each processing element produces a single result value and so only one of the result elements shown on the result bus is actually driven by this element. The other result elements are produced by the other processing elements in row n of the processor array.

A latency of one array cycle is shown between the final input value arriving and the first result being returned to the registers. A single cycle of latency is reasonable because an array cycle is relatively long time; for the simulations in Chapter 5 an array cycle is 20ns. However if this latency were increased to 2 or 3 array cycles it would still not represent a significant proportion of the overall latency of the operation.

3.2.2 Elementwise Operations

The elementwise operations that can be performed on the Load/Store architecture are addition, Hadamard multiplication, elementwise division and elementwise square root. The method used to perform elementwise operations on the Load/Store architecture is similar to that chosen on the Memory/Memory architecture. The data is transferred to the array just as for a multiply operation, but with one of the operands being transposed. Each processing element computes the operation on its two inputs and moves the result directly into the result register. The elements of the matrix sum are formed on the leading diagonal elements of the processor array; only these result are written back to the registers while all others are discarded.

In a simple extension of this method of writeback, the Load/Store architecture can also write back the results from a single row or column of processing elements, rather than those on the diagonal. This modification allows the computation of a matrix-scalar sum, product or quotient. In the Memory/Memory architecture, such matrix-scalar operations were performed by using the address generators to generate a constant matrix.

Figure 3.4 shows the timing of data transfers to and from each processing element during a single elementwise operation.

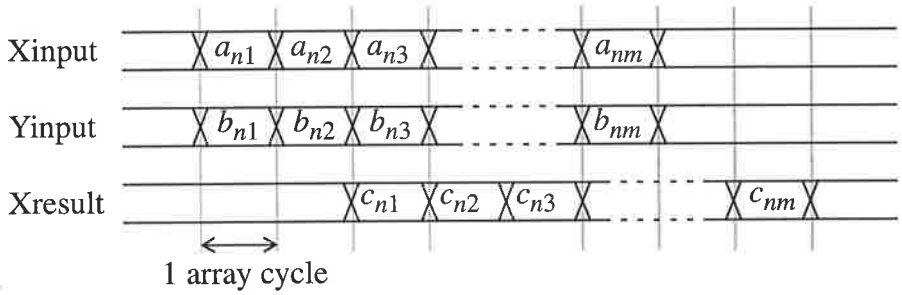


Figure 3.4: Elementwise Operation Timing

Here the writeback of results occurs in parallel with the operation. Again one cycle of latency is assumed.

During elementwise operations, reading of the inputs from, and writing of the results back to, the registers occurs simultaneously except for a short delay due to pipelining within the processing element. This is in contrast to multiply operations, where the result is written back only after all inputs have been read and the computation is finished. Thus, elementwise operations following a multiply operation must stall so that the writeback circuits are available, as can be seen in Figure 3.5.

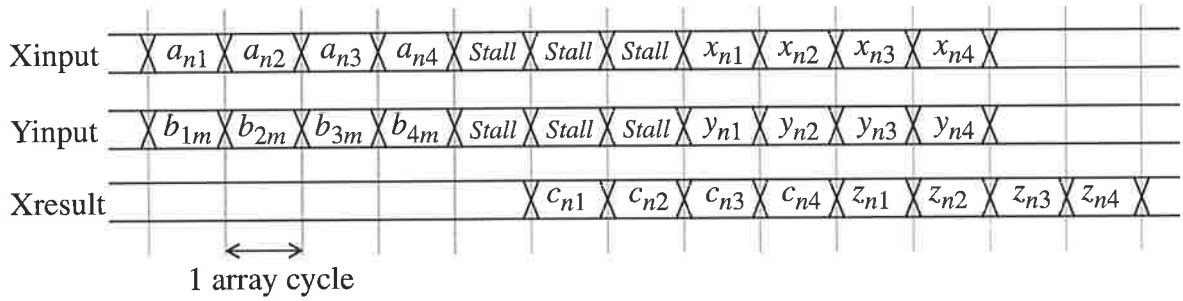


Figure 3.5: Interleaving Addition and Multiply Operations

Performing addition on the diagonal elements requires one of the input matrices to be transposed. This requirement in the Memory/Memory architecture is trivial, as the address generators can apply an arbitrary mapping to every operand and result. In the Load/Store architecture, this requirement may cause performance problems in some algorithms, and determining whether it introduces unacceptable overheads is one of the reasons for detailed simulations. An advantage of this requirement is that it allows the explicit formation of the transpose, which is necessary in some calculations.

The processing elements can also be commanded to alter the sign of their input operand. The input may either be left unchanged, negated, the absolute value taken or the sign function applied, as defined below. This feature can be used to perform subtraction.

$$\text{sign}(x) = \begin{cases} 1 & x > 0 \\ 0 & x = 0 \\ -1 & x < 0 \end{cases}$$

3.2.3 Test Operations

Test operations allow a comparison to be made between the current value of the accumulators in the processing elements and zero; no data is transferred between the array and the registers. The test may be applied to all processing elements, or to a single row or column, or to a single processing element. These options allow testing of specific elements and fast testing in parallel without a large overhead. The overall result is the boolean-OR of all the results for the processing elements taking part in the test. The test applied to the accumulator may be either for zero, not zero, positive, not positive, negative or not negative. The result of the test is communicated from the processing elements to the matrix controller as part of the command bus. To simplify testing of matrix data it is possible to disable the writeback of results from the array.

3.3 Virtual Processing Elements

One of the most difficult design trade-offs to make with the MATRISC architecture is balancing the performance for both large and small matrices. The highest computational performance, for multiply operations, is achieved by using a very large array that takes advantage of the compute-bound nature of the matrix product. However, the computational rate for matrices smaller than the array size is drastically less. The constant bandwidth array, see §2.5, was proposed to combat this difficulty, but studies have shown that the requirement to implement processing elements in a range of speeds and sizes is impractical[Beaumont-Smith 95][Beaumont-Smith et al. 96]. Instead, the studies demonstrated that the most efficient use of hardware occurs when the processing elements are made as fast as possible within a given technology. However, it is difficult to make a memory architecture that can meet the bandwidth requirements of the fastest possible processors.

Rather than using a large array of slow processing elements, here it is proposed to use a small array of fast elements that acts as though it were a large array of slow elements by

using time multiplexing. Thus a large *virtual array* is formed from a small number of *multiplexing processing elements*. A single multiplexing processing element can act as a $v \times v$ array of elements up to some size $V \times V$. The value v is called the *virtual factor*, and may vary between different calculations from 1 up to V, the *maximum virtual factor*. Restriction of v to binary powers is assumed, which greatly simplifies address generation and processing element control logic.

The extra hardware required to implement a multiplexing processing element consists of two input buffers of length V and an accumulator and result register of length V^2. All of these buffers can be implemented with rotating shift registers, rather than requiring random access registers. A block diagram of the construction of a multiplexing processing element is shown in Figure 3.6. The small labels identifying signals are used in the timing diagram in Figure 3.9.

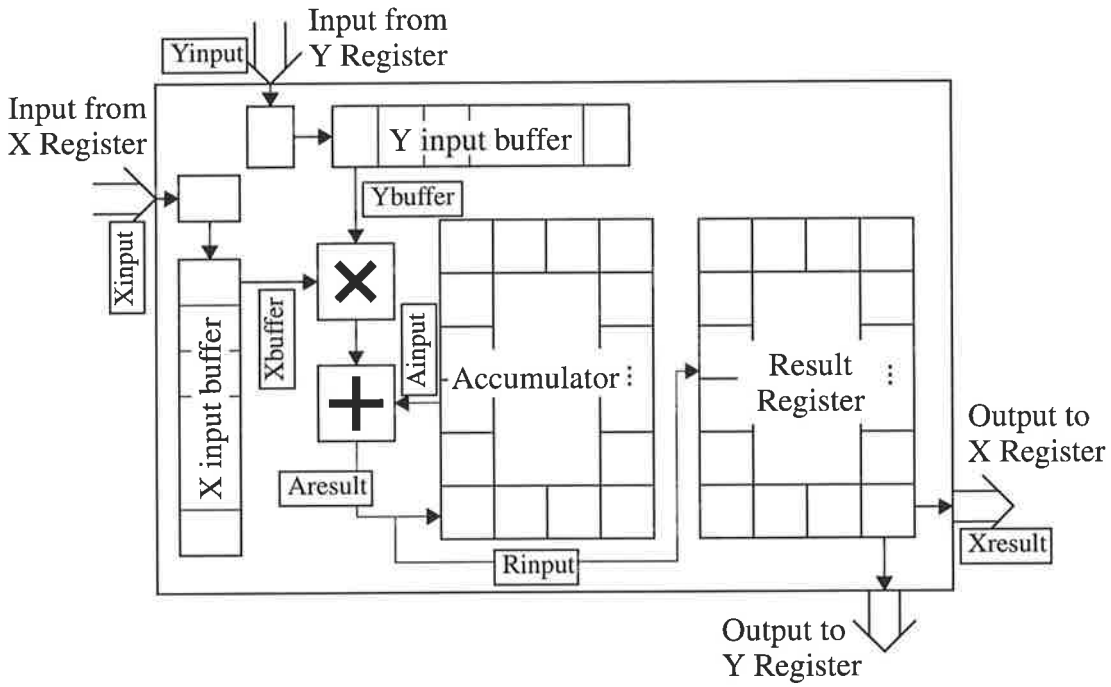
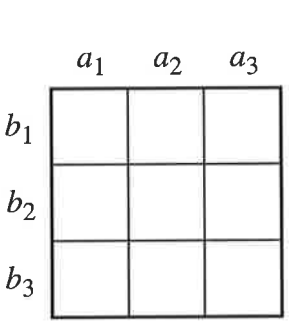


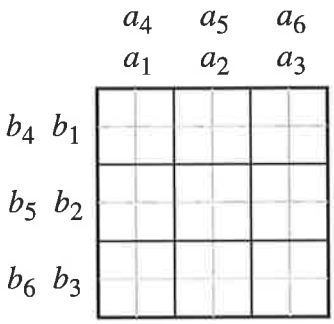
Figure 3.6: Internal Construction of a Multiplexing Processing Element

To understand how a multiplexing processing elements works, first consider the operation of the processor array without virtual factors. In every array cycle, two vectors of matrix elements are read into the array, and the outer product of those two vectors is formed and added to the values already in the accumulators of the processing elements. If a virtual factor of, for example, two is used, the array will act as though it is has twice as many processing elements along each side. The two vectors read into the array will be twice as

long, but because the bandwidth remains the same, transfer of the full vector to the array now takes two array cycles. The calculation of the outer product of vectors twice as long involve four times as much work, but because there is twice as long to complete the operation the processor array need only operate at twice the computational rate. This point is shown diagrammatically in Figure 3.7, which shows a single outer product step for a 3×3 array both with and without a virtual factor. In general, a virtual factor of v requires each processing element to perform v multiply-accumulate operations per array cycle.



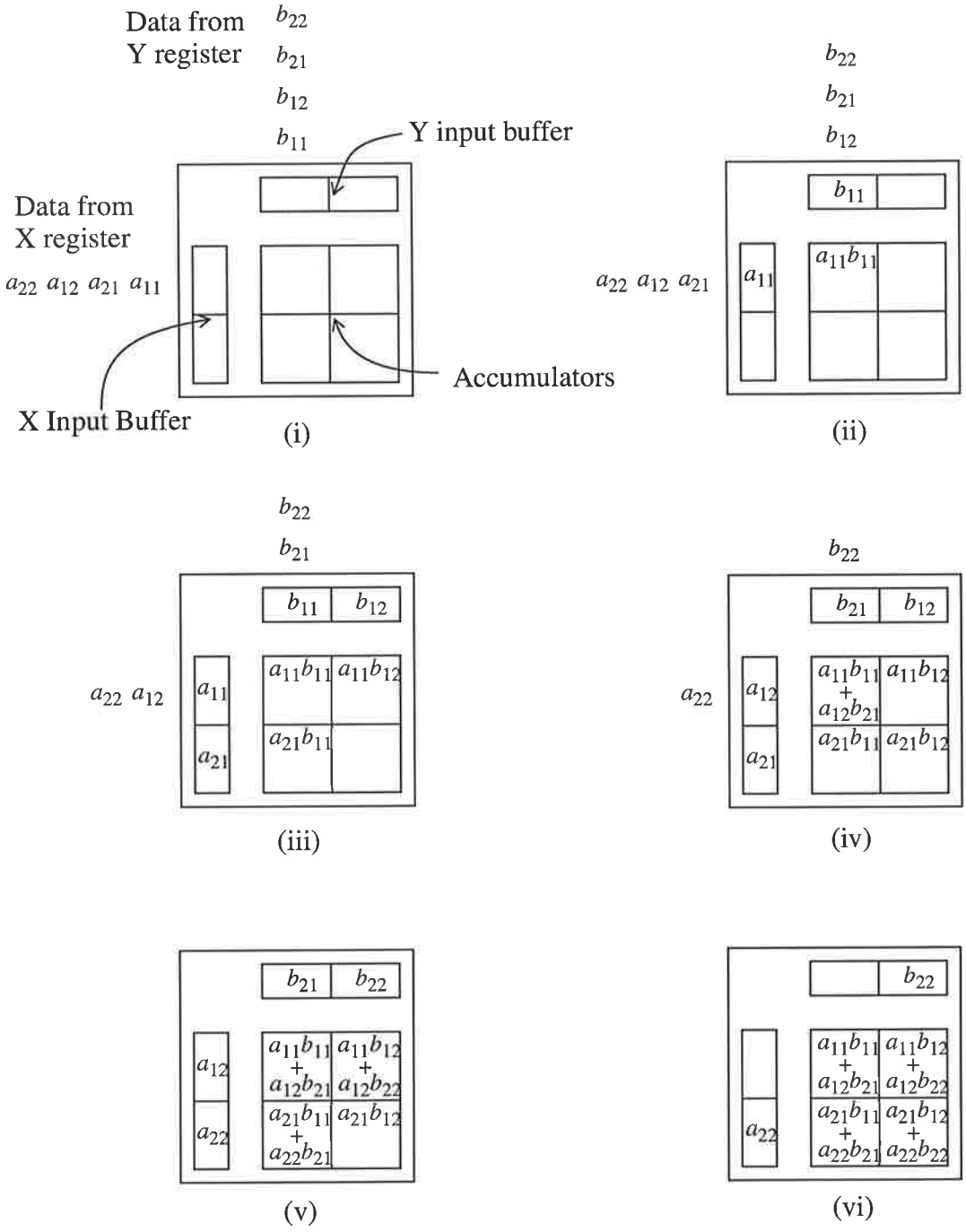
a) No Virtual Factor, v=1
Outer product of two vectors of length 3, requiring 9 multiply-accumulates in 1 array cycle



b) Virtual Factor, v= 2
Outer product of two vectors of length 6, requiring 36 multiply-accumulates in 2 array cycles

Figure 3.7: Single Outer Product Step from Non-Virtual and Virtual Multiplication

Consider Figure 3.8 which illustrates the product between two 2×2 matrices on a single multiplexing processing element with a virtual factor of two. The accumulator registers are shown as a square array for clarity. The formation of this matrix product requires two outer product accumulations. The exact order of computation within a processing element is determined by data availability, which can be illustrated by examining the order of computation for the first outer product. In the first cycle (Figure 3.8(ii)) only two operands have arrived so only one of the four multiply-accumulates required for the first outer product can be performed. In the second cycle (Figure 3.8(iii)) all the data is present, but the processing element only has time to do two of the remaining three calculations. The final multiply-accumulation is performed in the following cycle (Figure 3.8(iv)) and overlaps the first calculation for the next row/column. Figure 3.8(v) and Figure 3.8(vi) show the completion of the calculation for the next row/column.



$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

Figure 3.8: Operation of a Multiplexing Processing Element, $v = 2$

The timing of the data transfer within a multiplexing processing element for the same product is shown in Figure 3.9. The definition of the signals are shown in Figure 3.6.

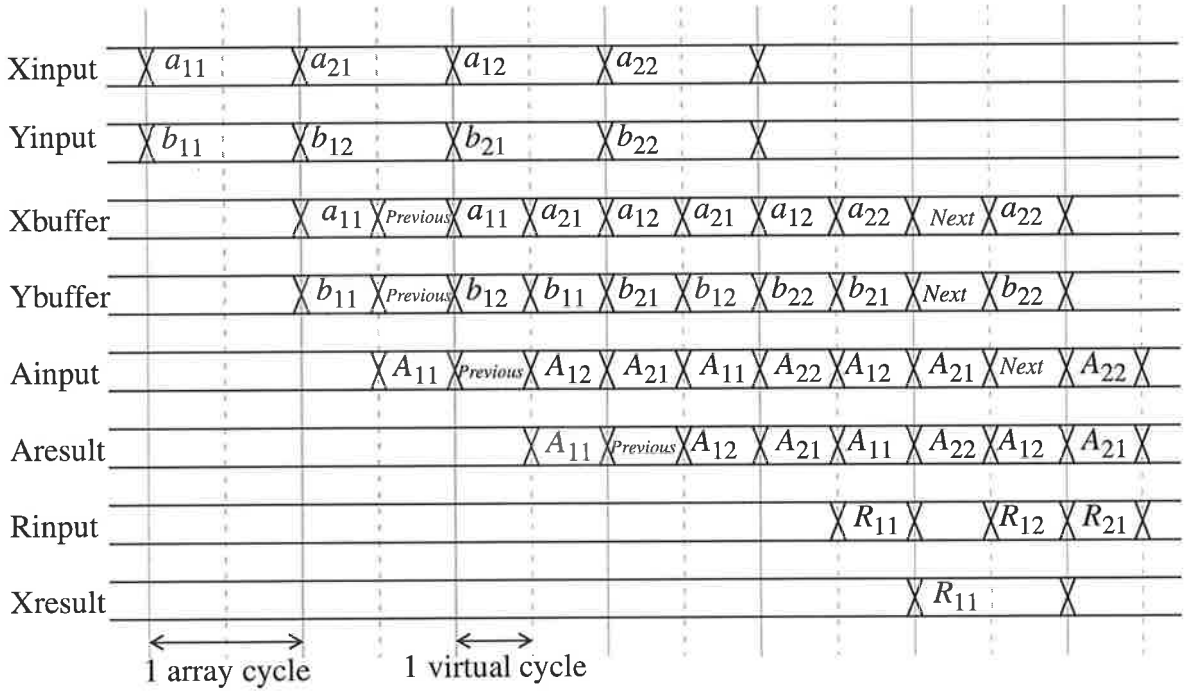


Figure 3.9: Virtual Processing Timing

The timing of the multiply-accumulate ($A = xy + A$) is assumed to require x and y in the first cycle, A in the second cycle and to return the result after 3 cycles. So the four shaded values are the inputs and result of one multiply-accumulate operation. The only timing dependency is the access to a single accumulator value (such as A_{11}). The values labelled '*previous*' and '*next*' are quantities from the preceding and following multiply operations respectively.

The operation of a multiplexing processing element is described by the following pseudo-code. The outer loop iterates through the v array cycles taken to deliver the v words of data to each array edge. The inner loop iterates through the v computation cycles that take place within each array cycle. `X_input` and `Y_input` are the X and Y inputs to the processing element, which have new data in every iteration of the outer loop.

```
for i in 0 to v-1 loop
     X_buffer[i] := X_input
     Y_buffer[i] := Y_input
     for j in 0 to v-1 loop
          A[i*v+j] := A[i*v+j] + X_buffer[j]*Y_buffer[(i-j) mod v]
     end loop
end loop
```

The indices of the `X_buffer` and `Y_buffer` entries that are multiplied together are shown in the table below. The shaded entries are those which use data loaded in this period of v

array cycles, rather than the previous one.

i - array cycles

j - virtual computation cycles	0		1		2		...	v-1	
0	0	0	0	1	0	2	...	0	v-1
1	1	v-1	1	0	1	1	...	1	v-2
2	2	v-2	2	v-1	2	0	...	2	v-3
...	...	...	...	...	...	...	...	...	...
v-1	v-1	1	v-1	2	v-1	3	...	v-1	0

Note that every possible pair from the buffers are multiplied together, and so the outer product of the two inputs is formed and accumulated with the product from the previous stage.

The code above can be reorganised in the following form, where the `rotate` procedure simply rotates a vector by the given amount.

```
for i in 0 to v-1 loop
     Y_buffer[0] := Y_input
     for j in 0 to v-1 loop
          if i = j then X_buffer[0] := X_input
          A[0] := A[0] + X_buffer[0]*Y_buffer[0]
          rotate(A, 1)
          rotate(X_buffer, 1)
          if j /= v-1 then rotate(Y_buffer, -1)
          end loop
end loop
```

Expressing the computation in this form illustrates that the storage for `A`, `X_buffer` and `Y_buffer` need only be shift registers rather than the more general random access registers. The only difference is that here the values in the accumulator vector are in a different order.

3.4 Bus Array Interconnection

As discussed in §2.4.1, it is possible to use a bus interconnection within the processor array rather than a systolic one. A bus interconnection decreases the latency of each operation and may reduce the complexity of the interconnection because skewing of the input data across the array is not required. Operands are simultaneously transferred from the

registers to all the processing elements in one row or column of the array, and the result from any processing element in one row or column can be directly written back to the register. In addition, the operation that each processing element performs on the data is sent to all processing elements on a global command bus, rather than accompanying the data as it passes through the array, as is required in the systolic case.

Another advantage of a bus connection is that the number of communication lines per processing element is reduced, so fewer I/O pads are required per chip. When bit serial or narrow connections are being used, as in the SCAP implementation of MATRISC, this consideration is less important. However, when the buses must be made wide to achieve the required data rates, it may be difficult to find room for all the connections. Considering that the Load/Store architecture has result writeback buses in both directions, a bus interconnection still requires only two thirds of the connections for a systolic interconnection scheme.

In general, the serious limitation with the bus approach is that buses do not scale. For long bus lengths, driving the bus is slower and requires more power, which imposes some upper limit on the size of the processor array. Since one of the precepts of the original MATRISC work was that arbitrarily high performance could be achieved by scaling the size of the array, any limitation on size was avoided. Results described in the following chapters show that there are limits to the size of an array that would be useful in practice, suggesting that the most useful configurations of the architecture would have relatively few processors, of order $p \leq 10$. This being the case, the scalability advantage of the systolic array is of less importance.

Limitations on bus length could be alleviated by partitioning the bus hierarchically. Imagine a 16×16 processor array formed by a 4×4 array of chips, with each chip containing 4×4 processing elements. Each bus would drive 4 chips, and on those chips the bus would drive 4 processing elements.

Whether buses are the best connection method in a particular implementation technology depends on complex hardware considerations involving balance between fan out and pin count, which requires in-depth implementation studies that are beyond the scope of this work. However, for the reasons presented above, it likely that a bus system would provide better performance than a systolic system; for this reason, and for simplicity, all analysis assumes a bus system. Furthermore, in an architectural sense, the significant differences between bus and systolic interconnects is latency. As will be evident in later chapters, the

important performance parameters of the Load/Store architecture do not include latency, which is therefore not modelled in most of the simulations.

As Figure 2.4 showed, the results of a computation can be written back to either the caches or main memory in the Memory/Memory architecture. Under a load/store paradigm, any writeback must occur to the registers as data may only be moved to main memory during a store operation. To achieve writeback to both registers, two writeback paths are used, one in the X direction and one in the Y direction. More hardware is required than for only one writeback path, but the two paths ease programming and possibly the task of physically laying out the design by keeping the architecture symmetric in the X and Y directions. The alternative of one writeback path would require that the result be transposed as it was written to one register but not transposed when written to the other register. The difference between the two approaches is shown in Figure 3.10.

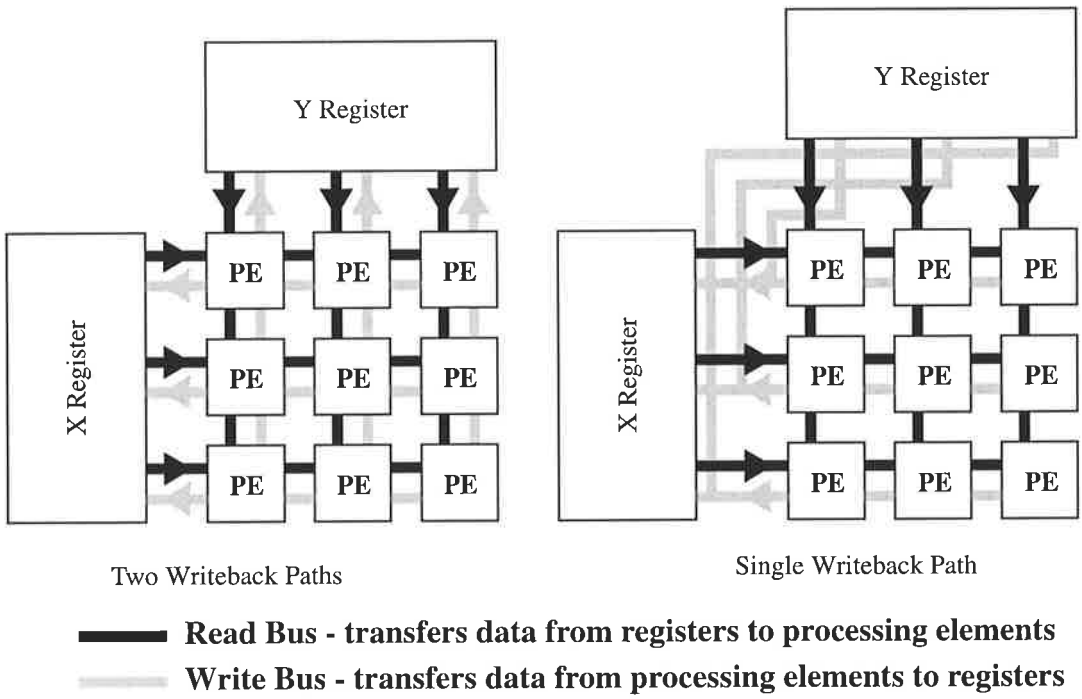


Figure 3.10: Register Writeback Alternatives

3.5 Matrix Storage in Registers and Read/Write Address Generation

In a system with a $p \times p$ physical array using register columns that are s deep, the register

can be regarded as an $s \times p$ array. Loading a matrix into a register implies mapping it on to an $s \times p$ array. There are a number of possible register mappings that may be used, and these influence the type of read/write address generation required.

3.5.1 Register Addressing Schemes

Registers require two different types of addresses. *Row addresses* refer to whole register rows, and are used by read/write accesses to the registers, since these accesses only operate on complete register rows. Row addresses run from 0 to s-1. *Element addresses* refer to individual elements within the registers and are used by load/store accesses to registers. There are three likely choices for the element address scheme, shown in Figure 3.11 for the case where $p = 3$.

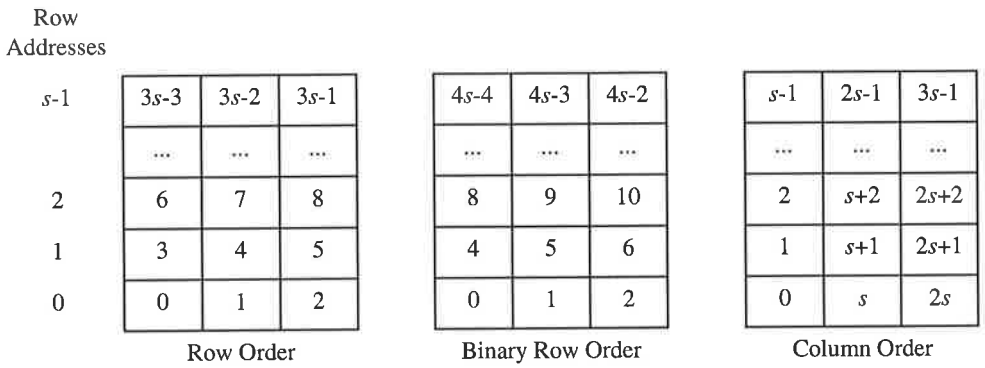


Figure 3.11: Element Addressing Schemes

The most obvious choice is row order element addressing, where addresses increase sequentially across each register row. The potential problem with this system is that when the hardware performs a load or store data transfer, this address must be decomposed into the number of a register column and an address within that column. This calculation will involve division by p, which would be costly. The situation could be rectified by forcing p to be a power of two, but this is quite restrictive. The next two address schemes avoid this problem.

The second addressing scheme, binary row order, uses addresses that advance by a power of two when moving from one row to the next. The problem with this method is that it leaves holes in the address space, which means that calculating the register addresses for the load and store operations is more difficult. The difficulties can be avoided by modifying the

address generator so that, for instance in the $p = 3$ case, the bottom 2 bits of the adders in the address generator clock over to 00 when they reach 10, and not when they reach 11. However in this case, using the same matrix controller, and hence address generators, with different sized arrays would be more difficult.

The final form of addressing is column ordering, which avoids all these problems entirely. Because s is the number of words in an SRAM it will always be a power of two, so decomposing the address involves only splitting the address in two at some bit position. Enabling the matrix controller to cope with the likely range of values for s is straightforward. However, because of its intuitive appeal the row order element addressing is used in the following simulations. This is not important at the current level of abstraction because it only influences the amount of hardware that would be required to perform the address calculation.

3.5.2 Read/Write Address Generation

The read and write address generators calculate the row addresses that are sent to the data controllers. Because the matrices in the registers have already passed through a flexible mapping, these address generators can be less flexible. The most basic form of address generation is to allow a base address and a variable stride; anything less than this is insufficient. For example, with a base of 2 and a stride of 3, the addresses produced would be 2, 5, 8, 11, etc. This scheme will be referred to as *simple addressing*.

$$\text{address}(n) = \text{base} + n \times \text{stride}$$

A much more flexible form of addressing would be to use a two dimensional difference engine, such as cut-down version of the four dimensional address generator used in the Memory/Memory architecture, see Equation (2.2). A generator as complex as this is not necessary, but certain restricted forms of it may be useful. If the modulo operation by q is ignored, Δ_2 set equal to 1 and N_2 set equal to v, then the *smart virtual addressing* scheme is obtained. For example, with a base of 2, a stride (Δ_1) of 3 and a virtual factor of 2, the addresses produced would be 2, 3, 5, 6, 8, 9, 11, 12, etc.

$$\text{address}(n) = \text{base} + \left\lfloor \frac{n}{v} \right\rfloor \times \text{stride} + n \mod v$$

As v is a small power of two, implementing this address generator is straightforward. The choice of address generator is mutually dependent on the choice of matrix storage method, so these are discussed in the following section.

3.5.3 Matrix Storage Methods

Perhaps the most natural way to store a matrix in a register is to load it one row at a time. The first element of the first row would go into location 0, the second into location 1, and so on. The first element of the second row would be placed in a new row of the register, even if that left a gap. This method is called *simple matrix storage* and is shown in Figure 3.12(i), for a 3×8 matrix A on a 3×3 processor array.

There are two possible variations on this pattern. First a *limited block size* could be used, meaning only part of the first row is loaded before the second is begun. This effectively divides the matrix into a number of partitions of a given width. The width of the block is chosen to be equal to the virtual array size; in this example, a virtual factor of two has been assumed, so on a 3×3 processor array the partition width is 6, see Figure 3.12(ii). It will be seen later that this choice of width leads to the simplest addressing.

The second variation is that elements may be *interleaved*, where elements in adjacent columns of one row are not in adjacent columns of a register row. Figure 3.12(iii) demonstrates interleaving by a factor of two, which leads to very simple addressing when using a virtual factor of two.

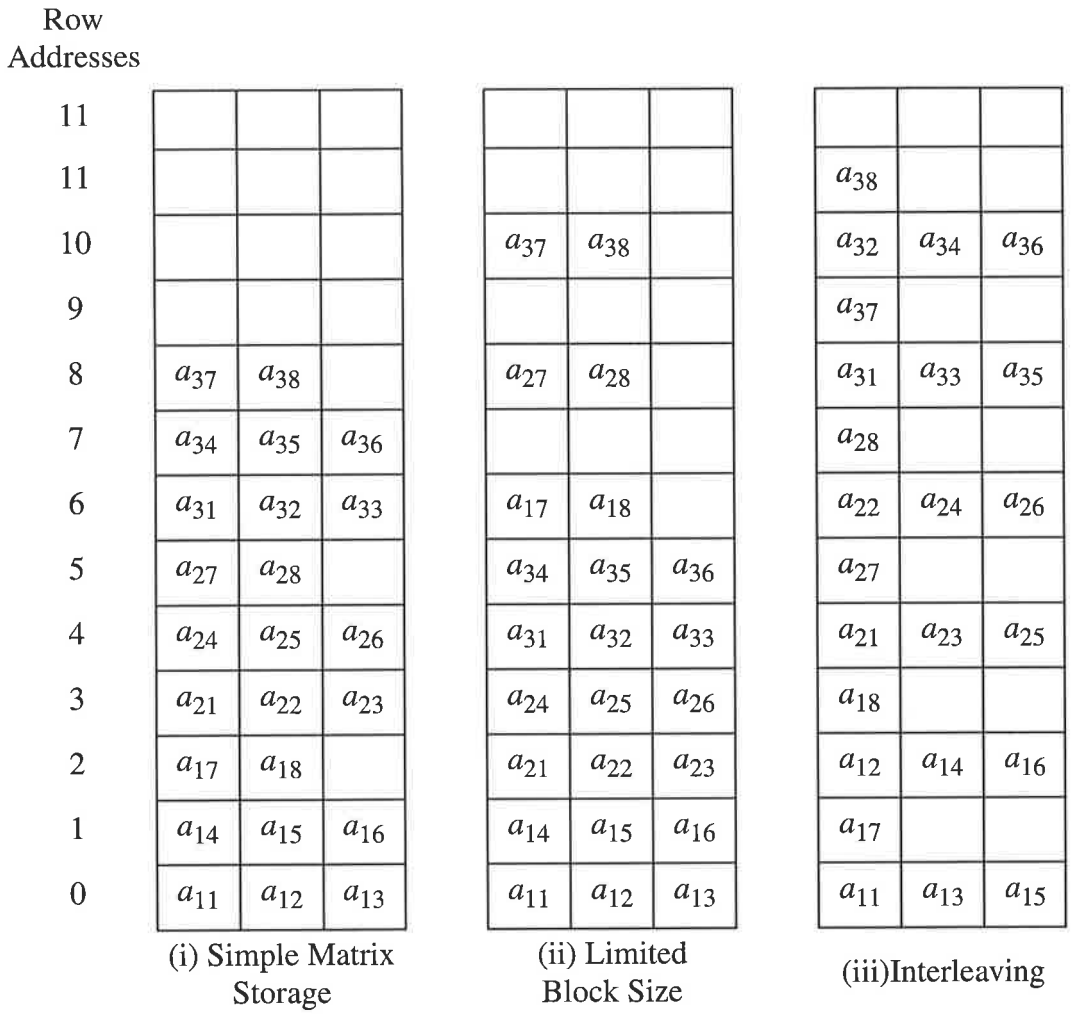


Figure 3.12: Matrix Storage Methods

Note that in all the storage schemes, elements may have to store extra zeros to pad out the register rows that do not contain a whole matrix row. Also, the limited block size and interleaved schemes tend to waste more space than simple matrix storage, although this has a relatively smaller impact with large matrices.

To appreciate why any of these storage methods are better than the others, it is necessary to examine what happens when matrix multiplication is performed. In the following example, a 3×3 physical array with a virtual factor of 2 will be used to examine the product $C = AB$. A is stored in the X register and B and C in the Y register. A and B are both 12×12, which means that this product must be partitioned. Each multiplication operation being discussed here calculates only a quarter of the result.

The first case to consider is the simple matrix storage with simple addressing only,

shown in Figure 3.13. The processor array is represented by the dark boxes in the bottom right corner. Each processing element contains four elements of the result matrix (there are four values as each processing elements acts as a virtual 2×2 array). The data in the X and Y registers is shown to the left and top of the processor array respectively, with each register row labelled by its row address. Note that not all the data for the two matrices is shown, just enough to demonstrate the pattern of storage. The dark boxes in the top left corner represent a column of processing elements. Their contents are the row addresses needed to write back to the Y register the elements calculated at those positions. For example the three shaded entries in the result are written back as one register row to the shaded row address. Of course using the exact row addresses shown would cause C to overwrite B so these addresses should actually be regarded as relative to some higher row address where we want to store C.

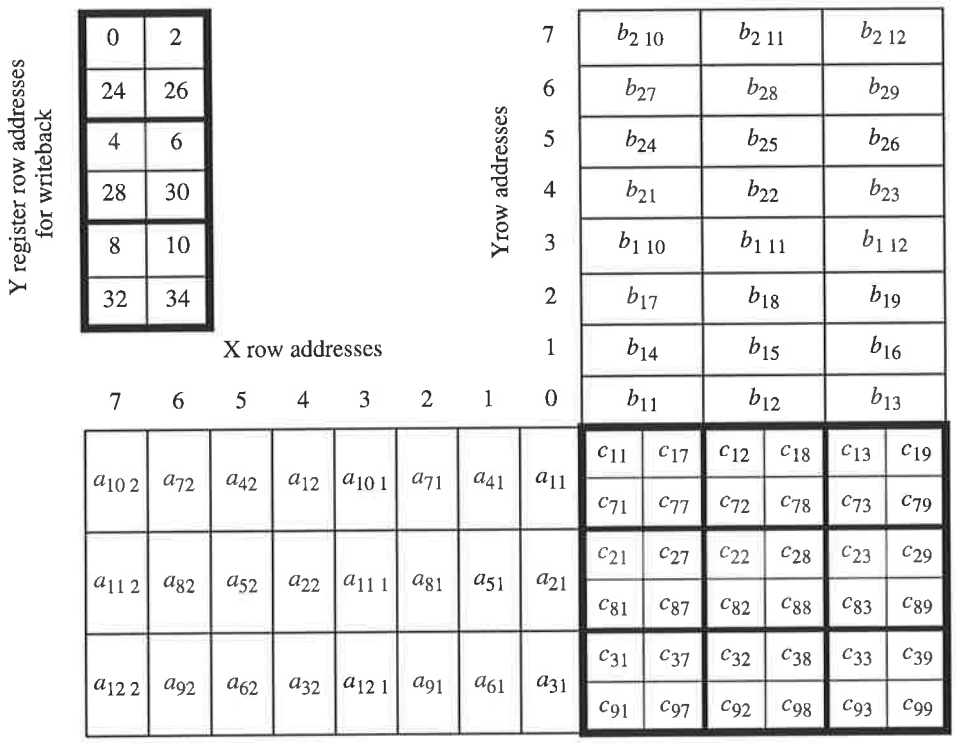


Figure 3.13: Multiplication with Simple Matrix Storage and Simple Addressing

The read stride that must be used is 2, which means that the first two register rows read into the array (row addresses 0 & 2) are from the first matrix row and the next two (row addresses 4 & 6) are from the second matrix row, and so on. The array of writeback addresses shows the row addresses to which one double column of results must be written

back. Note that the results can be extracted from the array in any order; what is important is that there be some order in which the results may be extracted that corresponds to a writeback address sequence that can be produced by the address generator. The address generator can only produce strictly increasing sequences; the writeback addresses in order are;

0 2 4 6 8 10 24 26 28 30 32 34

This sequence cannot be produced by simple addressing because of the jump in the middle. Therefore, simple matrix storage with simple addressing will not work.

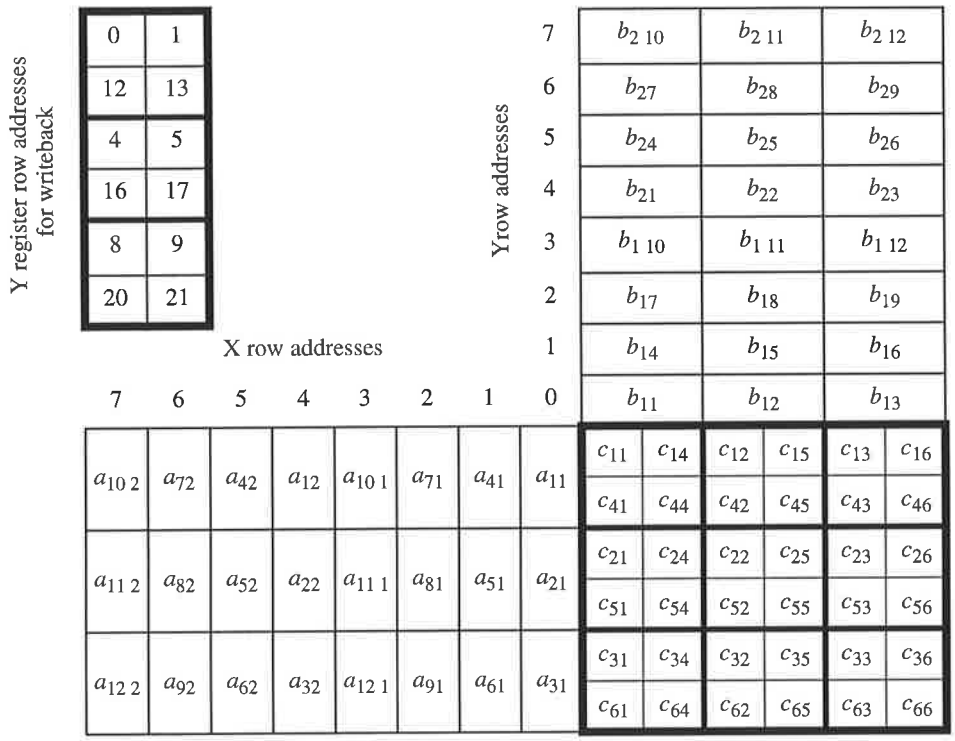


Figure 3.14: Multiplication with Simple Matrix Storage and Smart Virtual Addressing

The next case to consider is simple matrix storage with smart virtual addressing, shown in Figure 3.14. The read stride in this example is four. The sequence of writeback addresses required in this instance is

0 1 4 5 8 9 12 13 16 17 20 21

These addresses can be generated with smart virtual addressing using a stride of four. Thus simple matrix storage is only possible using smart virtual addressing or a more flexible scheme.

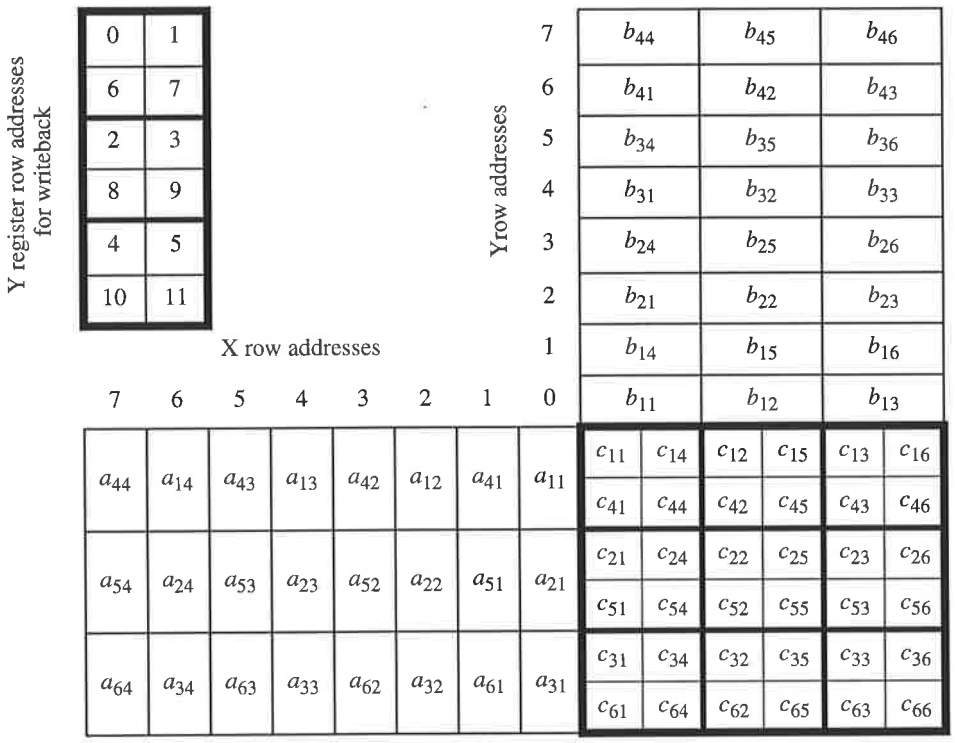


Figure 3.15: Multiplication with Block Size = Virtual Size and Simple Addressing

The next storage scheme to consider is limited block size, where there is potentially a range of choices for what size to use. Fortunately the only useful choice is to make the block size equal to the virtual size being used. The result of using this approach with simple addressing is shown in Figure 3.15.

The read stride is 1. The write generator also needs a stride of 1, as can be seen from the simple nature of address sequence required;

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

This very simple address pattern results from setting the block size equal to the virtual array size.

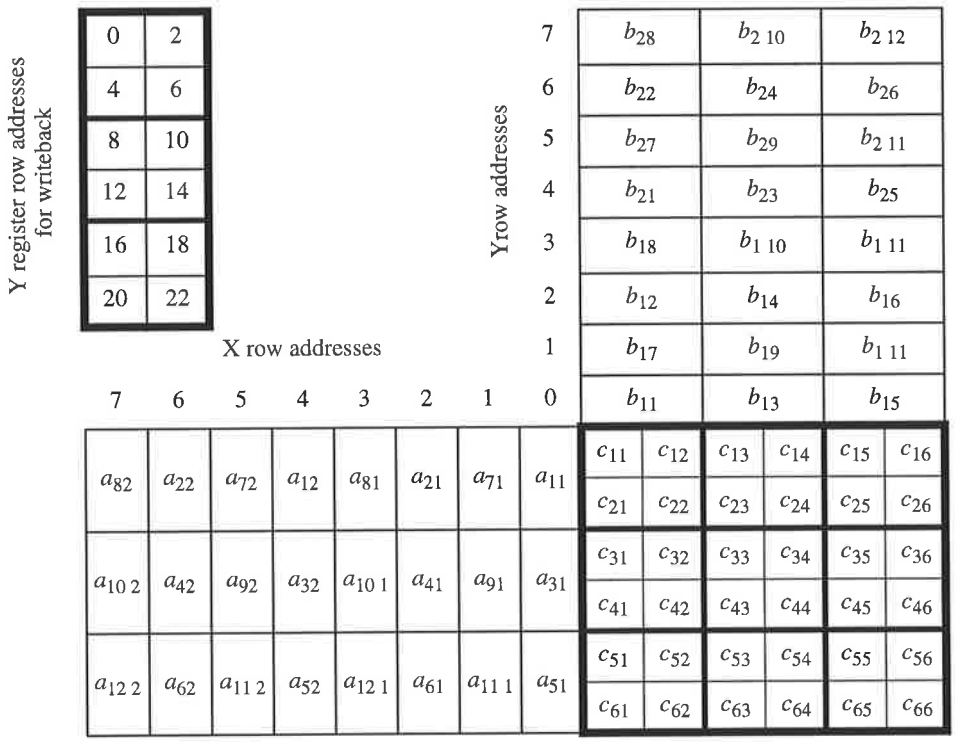


Figure 3.16: Multiplication with Interleaving and Simple Addressing

The final variation is to interleave the storage of rows as shown in Figure 3.16. This method works with a read and write stride both equal to 2, using only simple addressing. It also has the property that the writeback will occur in an ordered fashion across the array, which is not necessary, but would simplify some parts of the matrix controller.

Although both limited block size and interleaved storage work with the simple address generator, they are not used for the following reasons. Because these storage schemes are constrained to work with a particular virtual size, a matrix stored this way cannot be operated on using different virtual factors. These schemes divide the matrix so that adjacent elements are not together, which makes it much more difficult to perform operations on submatrices. Lastly, the inverted address generator described in §3.6, which is used for load/store transfers, cannot transfer one of these matrices as a single operation unless the matrix dimensions are a multiple of the virtual array size. More complex load/store address generators are possible, but in light of the other difficulties and the relatively simple nature of smart virtual addressing, they have not be pursued.

For these reasons simple matrix storage with smart virtual addressing is used in the Load/Store MATRISC architecture.

3.5.4 Load/Store Access to Registers

The justification for inverted address generation (introduced in §3.6), which is used to improve data transfer rates between registers and main memory, is that the SRAMs used to construct the register can support random accesses. However what is required is random access to all the elements of the register, which are spread across a number of register columns, each of which consists of one, or more, SRAMs. This access must be accomplished while making the minimum impact on the bandwidth available for reads and writes. Load and store access to the registers is achieved via the LS bus, which runs between the matrix controller and all the data controllers for a particular register (see Figure 3.2).

Read or write accesses operate on register rows, and thus occur in parallel across all the columns of a particular register. If one column is transferring data as part of a load operation, then all the other register columns must either also be doing a load operation or sitting idle. For this reason, it is desirable to perform a load or store to all register columns at once, which is achieved by having small buffers in the data controllers. Data is transferred across the LS bus to and from these buffers. When necessary, read and write accesses are stalled and one or more register accesses are *stolen* to allow the contents of the buffers to be transferred to or from the register columns.

Read/write access to the registers alternates between read and write. To maintain the balance between read and write accesses, it is desirable to steal accesses to the registers in pairs, which is relatively simple to organise. The principal requirement being buffer space for two elements in each data controller.

The discussion so far has implicitly assumed that a normal or in-order mapping has been applied during load and store operations. If this is the case, consecutive pieces of data will be sent to adjacent register columns so that when the steal is performed, all the buffers in the data controllers will be full. This will also usually occur with a prime factor mapping for most sized matrices. However when a transpose mapping is used, consecutive elements are all transferred to one register column, and so after two elements have been transferred, the data controller's buffer will be full and a pair of register accesses must be stolen. The result is a severe reduction in available compute bandwidth, the exact amount depending on the speeds of different memories. However, a reasonable estimate is that it would be halved (see Figure 5.11). Methods of avoiding this problem are discussed in §5.3.6.4.

One possibility to enhance load/store performance is to have the data controller

automatically steal idle read and write cycles. Idle cycles occur during matrix multiplication whenever the product being performed is not square and equal to the array size, because then the time taken to read the inputs into the array is not equal to the time taken to write the result out of the array. Thus, either read or write cycles must be idle at some stage. For example, when doing a square 100×100 matrix multiplication on a 10×10 processor array, 100 products between a 100×10 and 10×100 matrix are actually performed. For each step, the input read takes 100 cycles and the result writeback takes 10 cycles, leaving 90% of the write cycles idle. A further source of idle cycles is due to the fact that the result is only written back to one register, leaving that the other register with idle write cycles.

Another possible use for the idle cycles is for more reads or writes. In the 100×100 example, almost double the normal read bandwidth could be obtained. This possibility has not been pursued for two reasons. Firstly, for the increase in bandwidth to be usable the processor array must have extra computational capability that is not used at other times. Secondly, it provides no way to speed up elementwise operations such as addition, which are the operations most limited by bandwidth constraints.

3.6 Inverted Address Generation

The address generator used in the Memory/Memory architecture is very flexible and powerful, but is not well suited to use in the Load/Store architecture. A new method of address generation will be proposed here, which is better suited to the Load/Store architecture where address generation takes place between main memory and the registers, rather than between the caches and the processor array.

Firstly, in the Load/Store architecture, two address generators are necessary; one to generate the addresses in main memory and one to generate the addresses in the registers. If the address generator from the Memory/Memory architecture, hereafter called the *normal address generator*, is used to generate the main memory addresses, then the generator for the register addresses will simply have to produce a sequential series of addresses.

For some mappings, the normal address generator produces addresses that are far from sequential. For example, applying a transpose to a matrix produces addresses with a stride equal to the matrix dimension. This is not a problem in the Memory/Memory architecture where the addresses are sent to the cache, which, being built of SRAMs, handles all address

sequences equally well as long as data is present in the cache. However in the Load/Store architecture, the addresses are sent to main memory, which, being built of DRAMs, performs very poorly unless it is accessed sequentially. In both architectures, accesses to the DRAMs are unlikely to use the same row of memory more than once for real-sized problems, which implies that the DRAMs will be transferring data at less than their maximum rate. This is particularly important with the latest high performances DRAMs, such as those using RAMBUS technology, which have many features to take advantage of regular access patterns.

Inverted address generation employs the mathematical inverse of normal address generation, which results in sequential addresses being sent to the main memory and non-sequential addresses to the registers. The concept is illustrated in Figure 3.17, which shows both normal and inverted address generation for a simple two dimensional prime factor mapping.

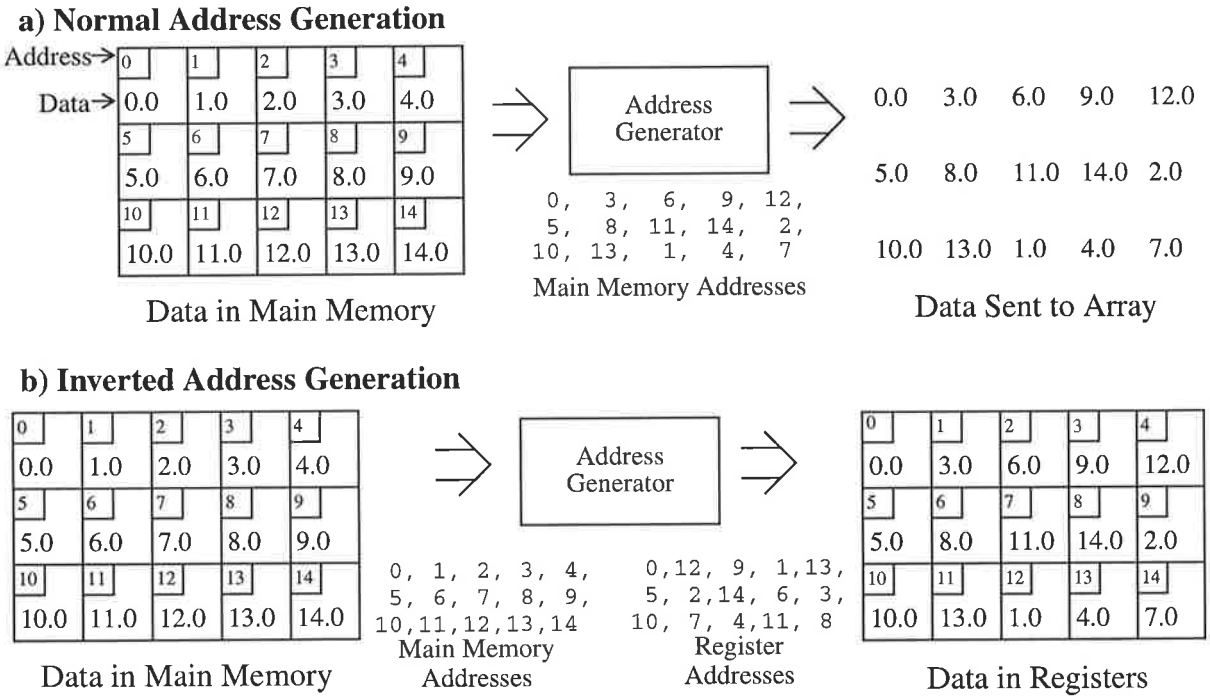


Figure 3.17: Example of Normal versus Inverted Address Generation

Inverted address generation allows the full bandwidth of main memory to be harnessed, but suffers from two principal drawbacks. The first is that because the data does not arrive in order in the registers, it cannot be used until a whole load operation is complete, which adds considerable latency to the overall operation. To alleviate this effect, large problems can be loaded in stages so that calculation may begin before loading is fully complete. The second

difficulty is that, because all data must be loaded before computation can begin, problems greater than the register size must be explicitly partitioned. These difficulties are not insurmountable, but do mean that the architecture is more complex to program. Some explicit partitioning algorithms are examined in Chapter 5.

The alternative to using inverted address generation is to use the normal address generator, and to increase the speed of main memory in some way. The access patterns that are generated have non-zero stride with tens to hundreds of elements. As such, these patterns are very similar to those often found on vector supercomputers, suggesting that an interleaved memory, as found on vector machines, would be able to achieve high performance. However, such a system requires the use of considerably more memory chips, and would still have poor performance if the stride is equal to the number of interleaved memory banks.

3.6.1 Inverted Address Generator

In order to produce the inverted address sequence for the registers, a new address generator is required because some sequences, such as the one in Figure 3.17, cannot be generated with the normal address generator. For constant and circulant matrices, the mappings are not one-to-one and thus are not invertible; but these cases can be dealt with in other ways. For other mappings, a suitable address generator uses the following form in four dimensions;

$$\text{register\_address}(n) = \text{s\_base} + \langle \delta_1 \left\lfloor \frac{n}{\sigma_1} \right\rfloor \rangle_{q_1} + \langle \delta_2 \left\lfloor \frac{n}{\sigma_2} \right\rfloor \rangle_{q_2} + \langle \delta_3 \left\lfloor \frac{n}{\sigma_3} \right\rfloor \rangle_{q_3} + \langle \delta_4 \left\lfloor \frac{n}{\sigma_4} \right\rfloor \rangle_{q_4} \quad (3.1)$$

This form was obtained by considering useful invertible mappings (normal, transpose, prime factor and Chinese remainder theorem), and finding a simple generator that could handle the inverse of all of them. One of the major strengths of the normal address generator is that it requires relatively little hardware.

Although the inverted address generator equation looks complicated, it will be shown below that it can be computed with relatively simple hardware, comparable with the hardware requirements of the normal address generator. The following sections show how this new generator can be used to generate mappings of interest. In doing so it is useful to decompose the inverted address generator equation in the following way;

$$\begin{aligned} \text{register\_address}(n) &= \text{s\_base} + \sum_{i=1}^{4} M_i n_i \\ n_i &= \langle \delta'_i \left\lfloor \frac{n}{\sigma_i} \right\rfloor \rangle_{q'_i} \\ \delta_i &= M_i \delta'_i \\ q_i &= M_i q'_i \end{aligned} \tag{3.2}$$

From Equation(2.2) it can be seen that the normal address generator maps from (n_1, n_2, n_3, n_4) to n. The goal of inverted address generation is to map from n to (n_1, n_2, n_3, n_4), as can be seen from the second line of (3.2). This part of address generation is controlled by the values n_i, δ'_i, q'_i, which are known as the *modified address generator parameters*. To form the final register address, (n_1, n_2, n_3, n_4) must be mapped back to a single number as seen in the first line of (3.2). This part of the address generation is controlled by the M_i parameters, which determine how the matrix is stored in the registers. In the following discussions, as with the normal address generator, n is assumed to range from 0 to N-1, where $N = N_1N_2N_3N_4$, and the n_i range from 0 to N_i-1.

3.6.1.1 Normal and Transpose Mappings

Using the normal address generator, the normal (in order) mapping is given in two dimensions by,

$$n = \langle n_1 + N_1 n_2 \rangle_N$$

where the operation modulo N is superfluous because the content of the angle brackets is at most N-1.

The inverse of this mapping, and the corresponding modified address generator parameters, can be expressed as follows;

$$n_1 = \langle n \rangle_{N_1} \qquad n_2 = \langle \left\lfloor \frac{n}{N_1} \right\rfloor \rangle_{\text{max\_int}}$$

that is,

$$\begin{aligned} \delta'_1 &= 1 & \delta'_2 &= 1 \\ \sigma_1 &= 1 & \sigma_2 &= N_1 \\ q'_1 &= N_1 & q'_2 &= \text{max\_int} \end{aligned}$$

where max_int is the largest value possible in the number scheme used, and effectively prevents the modulo operation in that dimension. Note that where only two dimensions of the address generator are required, the other dimensions can be disabled by setting the corresponding δ_i to zero.

For the normal address generator, the transpose mapping is very similar to the normal mapping,

$$n = \langle N_2 n_1 + n_2 \rangle_N$$

The inverse mapping is also similar,

$$n_1 = \left\langle \left\lfloor \frac{n}{N_2} \right\rfloor \right\rangle_{\text{max\_int}} \qquad n_2 = \langle n \rangle_{N_2}$$

that is

$$\delta'_1 = 1 \qquad \delta'_2 = 1$$

$$\sigma_1 = N_2 \qquad \sigma_2 = 1$$

$$q'_1 = \text{max\_int} \qquad q'_2 = N_2$$

These equations can easily be evaluated using the proposed inverted address generator.

3.6.1.2 Prime Factor Mapping

In the four-dimensional prime factor case, the mapping is of the form,

$$n = \left\langle \sum_{i=1}^{4} \frac{N}{N_i} n_i \right\rangle_N$$

where N_i are relatively prime. The inverse address mapping of this is given by,

$$n_i = \left\langle \left\langle \left\lceil \frac{N}{N_i} \right\rceil^{-1} \right\rangle_{N_i} n \right\rangle_{N_i} \qquad i = 1 \ldots 4$$

that is

$$\delta'_i = \left\langle \left\lceil \frac{N}{N_i} \right\rceil^{-1} \right\rangle_{N_i}$$

$$\sigma_i = 1$$

$$q'_i = N_i$$

proof of which is given in Appendix B.

3.6.1.3 Chinese Remainder Theorem Mapping

The Chinese remainder theorem mapping is of the form,

$$n = \left\langle \sum_{i=1}^{4} \frac{N}{N_i} \left\langle \left\lceil \frac{N}{N_i} \right\rceil^{-1} \right\rangle_{N_i} n_i \right\rangle_N$$

where N_i are relatively prime. The inversion is very simple:

$$n_i = \langle n \rangle_{N_i} \qquad i = 1 \ldots 4$$

that is $\quad \delta'_i = 1$

$$\sigma_i = 1$$

$$q'_i = N_i$$

The proof is also given in Appendix B.

3.6.2 Main Memory Address Generation

So far, the main memory address generator has been assumed to produce only sequential addresses. Being able to achieve this simple access pattern is the underlying motivation for inverting the address generation. However, the system can be made more flexible and useful by using a slightly more complex address generator. This generator will allow the loading of a single partition of very large problems, while still being able to support a high bandwidth from at least some types of DRAM. The equation for this main memory address generator is a much simplified two dimensional version of the normal address generator (Equation (2.2)).

$$\text{main\_memory\_address} = \text{d\_base} + sd_1 + td_2$$

By using a simple form of two-dimensional mapping to generate the main memory addresses, the loading of large transpose matrices can be handled, where s is the stride from above, d_1 is the low order counter ranging from 0 to D_1-1, d_2 is the higher order counter ranging from 0 to D_2-1, and t is another stride. For example, consider loading a 100 word wide strip of a 10000 × 10000 matrix. Using s=1, D_1=100, t=10000 and D_2=10000 this generator would produce the addresses 0, 1, 2, 3, ... 99, 10000, 10001, 10002, ... 10099, 20000, 20001, ... etc. The SRAM side of the address generator would then pack these in the correct locations within the registers.

This main memory address generator duplicates much of the functionality of the register address generator but without it, loading parts of large problems would be very difficult. However because the register address generator is very flexible, the main memory address generator can be inflexible and still allow fully flexible mappings. For example, s and D_1 can be limited to small values to fit in with the capabilities of available DRAMs.

3.6.3 Address Generator Implementation

The four dimensional inverted address generator equation, (3.1), can be evaluated for a

sequence of N addresses by the following pseudo-code. This code shows that the sequence can be calculated with only the use of counters and adders; no multipliers are required.

```
c[1 to 4] = 0
s[1 to 4] = 0
for n in 1 to N loop
   for i in 1 to 4 loop
      c[i] := c[i] + 1
      if c[i] = sigma[i] then
         c[i] := 0
         s[i] := s[i] + delta[i]
         if s[i] > q[i] then s[i] := s[i] - q[i] end if
      end if
   end loop
   address = s_base + s[1] + s[2] + s[3] + s[4]
end loop
```

There are a number of choices when trading off the size and speed of this generator. To achieve maximum speed, it can be heavily pipelined as the data dependencies are localised. It should be faster than the normal address generator used in the Memory/Memory architecture as there is no equivalent to the step of deciding which Δ_i to add in. On the other hand, a pipelined version could use up to 12 adders and thus would be larger than the original normal address generator.

3.6.4 Testing with RAMBUS

RAMBUS DRAMs[Rambus 97] use a high performance bus standard for connecting main memory (DRAMs) to processors or caches. They are technically superior to other DRAMs, as well as cost effective. Some of the more important features are;

- byte wide transfers every 2ns giving a peak transfer rate of 500MB per second (~470MB/s maximum usable)
- internal caching of rows giving short access times
- intelligent address mapping to maximise row reuse
- random access transfers (within one row) using serial address updates.

3.6.4.1 RAMBUS Speed

The amount of time required for a RAMBUS transaction depends on two factors; the size of the transfer and whether it is a hit or a miss on the rows of memory currently cached within the DRAMs. The size of the transfer may be any multiple of eight bytes up to 256 bytes. The transaction has a fixed overhead plus a variable length component proportional to the transaction time. For a miss, there is an additional length of time, called the retry delay,

while the DRAM accesses the required row.

To set a benchmark for the results of the following simulation, Table 3.1 shows transfer rates for loads that always hit and loads that always miss for various transaction sizes and retry delays.

Transaction Size (Bytes)	100% Hits	100% Misses with differing DRAM Retry Delay			
		80ns	100ns	120ns	140ns
8	182	64.5	55.6	48.8	43.5
16	267	114	100	88.9	80
32	348	186	167	151	138
64	410	271	250	232	216
128	451	352	333	317	302
256	470	413	400	388	376

Table 3.1: RAMBUS Transfer Rates (MB/s)

3.6.4.2 Address Generation Comparison

Table 3.2 contains a summary of the simulated results for loading a matrix using the normal and inverted address generators. The simulation assumed eight 64Mbit RAMBUS DRAMs as described in [Rambus 95], which implies 32 rows of 2K each are cached within the DRAMs. A retry delay of 100ns was used for all tests. Both generators were configured to take advantage of the RAMBUS ability to randomly access addresses within one row in a single transaction by grouping consecutive non-unit stride accesses wherever possible.

Matrix Size	In Order		Transpose		Prime Factor	
	Normal	Inverted	Normal	Inverted	Normal	Inverted
10×10	404	442	404	442	406	446
20×20	445	463	415	460	396	460
50×50	455	463	364	463	357	463
100×100	457	463	193	463	183	463
200×200	460	463	69	463	69	463
500×500	462	463	56	463	56	463
1000×1000	463	463	56	463	56	463

Table 3.2: Loading Rates for Different Mappings (MBytes/s)

These figures show that the inverted address generator performs close to the maximum speed possible for the bus for all matrix sizes and mapping. The normal address generator performs equally well for in order access to the matrix, but its performance drops significantly when applying transpose or prime factor mappings, especially to larger matrices. In fact, for matrices over 500×500, the normal address generator is forced to use 8 byte transfers that always miss. This is the worst possible case and results in transfer rates that are only 12% of the inverted address generator's performance.

From these results, it is possible to see that inverted address generation can achieve very high main memory transfer rates while still allowing flexible matrix mappings.

3.7 Control Processor

The SCAP implementation of the Memory/Memory architecture operated by receiving a series of commands from the host computer that set up the address registers, followed by a command for the particular operation to be performed. For short duration operations, the slow communication with the host was found to be a considerable overhead. Given the much higher computation speeds possible with more recent implementation technologies, it is essential that any new architecture include a more efficient control mechanism.

The proposed solution uses a general purpose RISC CPU closely coupled to the control logic in the matrix controller. Such an arrangement would allow a program to be downloaded from the host and left to run without host intervention. The precise method by

which this control would be accomplished is still open, but the simple method detailed below is used for the simulations discussed in later chapters.

The MATRISC architecture from the programmer's point of view consists of two major parts; a matrix data path and a tightly coupled general purpose control processor. The control processor is directly programmable and controls the data path as a co-processor. The control processor described here is a simple, general purpose integer RISC processor with a small set of co-processor control instructions. The principal short term goal in defining a behaviour for this processor was to provide a target for a compiler for the MATRISC architecture. The precise form the control processor would taken in an actual implementation is an open question. An existing commercial RISC CPU would be ideal because of its high performance and low cost. However such processors use intelligent bus controllers that are optimised for caches, and thus introduce considerable latency in accessing the bus in order to achieve maximum throughput. What is required here is a fast simple device that can directly access its bus with low latency. A semi-custom or programmable logic implementation of a simple processor, possibly a synthesized core, would be a better solution.

The communication model between the data path and the control processor is illustrated in Figure 3.18, which is an abstraction of the complete architecture diagram in Figure 3.2.

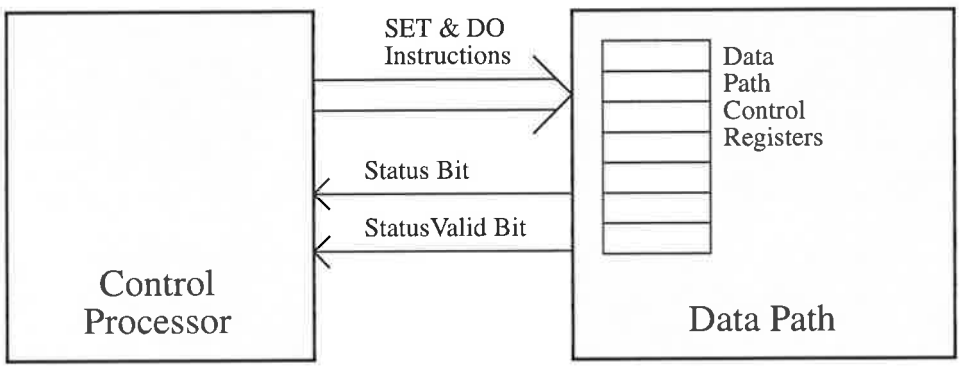


Figure 3.18: Communication Model between the Control Processor and the Data Path

The data path contains a number of control registers that hold the parameters of the operations it performs, for example, the address generator parameters for load and store operations. These control registers reside in the matrix controller and are set by the control processor when it executes a SET instruction. The data path is commanded to perform an operation when the control processor executes a DO instruction. Communication from the

data path to the control processor is achieved via the use of a status bit, which indicates results of test operations.

Timing of SET and DO instructions is simple. The data path has a short buffer for storing instructions and when it is full, the controlling processor must stall if it wishes to issue another. The timing of the communications from the data path to the control processor is complicated by the fact that data path operations generally take long and unpredictable amounts of time. A major difficulty lies in ensuring that the value of the status bit tested by the control processor was set by the desired data path test operation. This problem may be solved by sending a second signal from the data path to the control processor, a StatusValidBit, that indicates when the status line is valid. Any test of the status bit inside the control processor is stalled until a valid result has been returned by the data path.

3.7.1 Control Processor Hardware

The control processor has the following registers;

- general purpose 32 bit integer registers named R0, R1, R2, etc.
- a program counter (PC)
- a status register with zero (Z), negative (N) and data path (D) flags.

The zero and negative flags are set by all arithmetic instructions. The data path flag is set only when a data path test instruction is performed.

3.7.2 Instruction Set

The control processor instructions can have 0, 1, 2 or 3 arguments depending on the instruction. The addressing modes used, and an example of each, are shown in Table 3.3. Constants are limited to 16 bits.

Addressing Mode	Example
Constant	#3
Register	R2
Absolute	1000
Indirect	(R2)
Indexed	4(R2)
PCrelative	10(PC)

Table 3.3: Addressing Modes

The simple RISC instruction set used by the control processor is shown in Table 3.4. A complete general purpose instruction set is required because the control processor may have to implement an arbitrary algorithm in order to control the matrix data path.

Instruction Type	Instruction Template	Meaning
3 Address Compute	`ADD s1, s2, d`	`d := s1 + s2`
	`SUB s1, s2, d`	`d := s1 - s2`
	`MUL s1, s2, d`	`d := s1 * s2`
	`DIV s1, s2, d`	`d := s1 / s2`
	`OR  s1, s2, d`	`d := s1 | s2`
	`AND s1, s2, d`	`d := s1 & s2`
	`LSL s1, s2, d`	`d := s1 << s2`
	`LSR s1, s2, d`	`d := s1 >> s2`
	`ASR s1, s2, d`	`d := s1 >>s2`
2 Address Compute	`NOT s1, d`	`d := ~s1`
	`LDI #s1, d`	`d := s1`
	`LDH #s1, d`	`d := s1 << 16`
Compare	`CMP s1, s2`	`s1 - s2`
Memory Transfer	`LD ad, r`	`r := mem[ad]`
	`ST r, ad`	`mem[ad] := r`
Branch	`JMP ad`	`PC := ad`
	`BEQ ad`	`if (Z=1) PC :=ad`
	`BNE ad`	`if (Z=0) PC :=ad`
	`BPL ad`	`if (N=0) PC :=ad`
	`BMI ad`	`if (N=1) PC :=ad`
	`BDC ad`	`if (D=0) PC :=ad`
	`BDS ad`	`if (D=1) PC :=ad`
Data Path Control	`SET s, register`	`register := s`
	`DO operation`	`begin operation`
Miscellaneous	`NOP`	`nop operation`
	`END`	

Table 3.4: Control Processor Instruction Set

The source operands `s1`, `s2` and `s` may use either constant or register address modes, with the restriction that an instruction may not have two constant operands. Operands

shown as `r` or `d` may only use register address modes. Address operands, shown as `ad`, may use Absolute, Indirect, Indexed or PCrelative address modes. The Z and N condition codes are set only by compute and compare instructions.

The names and a brief description of the data path control registers are shown in Table 3.5

Control Register Name	Description
Load/Store Address Generator Registers	
Saddress, Delta1, Sigma1, Q1, Delta2, Sigma2, Q2, Delta3, Sigma3, Q3, Delta4, Sigma4, Q4	Register address generator parameters
Daddress, Stride, Runlength, Increment	Main memory address generator parameters
LSLength	Length of load/store transfer
Computation Operation Registers	
Xaddress, Xstep, Yaddress, Ystep, Raddress, Rstep, Length	X register input, Y register input and result address generators
Virtual	Virtual factor for multiplication
Mode, WBMode, XSignMode, YSignMode	Mode flag (see below)
Row, Column	Row and column selector for testing and scalar operations

Table 3.5: Data Path Control Registers

The values in all of the data path control registers are integers except the Mode registers, which hold a mode value (encoded as a integer). The sign mode values are Plus, Minus, Abs and Sign, while the writeback mode values are of LinearX, LinearY, LinearBoth, DiagonalX, DiagonalY, DiagonalBoth and NoWB. Note that WBMode, XSignMode and YSignMode are actually stored in one register referred to as the Mode, which allows the modes to be set individually or together.

The data path operations are shown in Table 3.6. A complete detailed description of the

precise operation of these data path instructions can be found in Appendix A.

Multiply	DivideXY	TestZ	LoadX	PrintM
Chain	DivideYX	TestNZ	LoadY	PrintX
Addition	SqrtX	TestP	StoreX	PrintY
Hadamard	SqrtY	TestN	StoreY	

Table 3.6: Data Path Operations

3.8 Summary

This chapter has described the proposed Load/Store MATRISC architecture in detail. The most important aspect is the novel memory architecture, which is proposed as a replacement for the existing bandwidth-deficient Memory/Memory system.

The Load/Store memory architecture replaces the caches of the Memory/Memory system with a parallel memory that operates as registers under programmer control. This arrangement provides high bandwidth at the edge of the processor array that scales with the size of the array. A new inverted address generator allows load and stores, that is transfers between main memory and the registers, to operate at close to the peak speed of the main memory for a wide range of useful mappings.

The processor array has also been modified. The processing elements are capable of performing multiply-accumulate, multiplication, addition, division and square root operations. They contain extra registers and control logic, which allows each processing element to act as a small array of virtual processing elements. Virtual processing allows the array as a whole to create a better balance between computation and memory bandwidth for a range of matrix sizes; this is similar to the constant bandwidth array proposed for the Memory/Memory system, but without the difficulty of fabricating processing elements in a range of speeds.

Buses have been used to connect the processing elements within the array, as this simplifies the operation of the array, reduces the latency of operations and reduces the processing element pin count. With the smaller array sizes expected for the Load/Store architecture, the limited scalability of the bus system is unlikely to be a problem.

The coordination of the registers and the processor array is performed by a central matrix

controller, and the overall operation of the architecture is controlled by a simple RISC control processor.

Chapter 4

MATRISC Architecture Programming

In order to implement the relatively complex algorithms needed to examine the performance of the Load/Store MATRISC architecture, the programming methodology outlined in this chapter was developed. This work also allowed the difficulties of programming the architecture to be investigated as a performance metric for the architecture.

The first section introduces a nomenclature to describe precisely how a matrix is stored in the registers and how arbitrarily sized matrix operations can be performed on the fixed sized processor array.

A simple language, called Masm, for expressing algorithms on the architecture, is then described. Masm handles the partitioning of large matrix calculations and the generation of the control processor code for matrix operations. A programming methodology, which shows how the Masm language can be used to implement algorithms, is then proposed.

Finally, a simulation in VHDL, including a refinement of the design of the architecture, is discussed.

4.1 Matrix Storage and Matrix Operations

In order to program the matrix processor, it is necessary to determine exactly how operations on matrices larger than the processor array are performed, which in turn depends on how the matrices are stored in the registers. By developing a nomenclature for these aspects of programming, it becomes possible to exactly describe the situations where the limited flexibility of the Load/Store architecture causes an overhead. It will then be

demonstrated that this overhead can be avoided in most, but not all, situations by reorganising the computation.

4.1.1 Register Storage Methods

The address schemes used for register rows and individual register elements was described in §3.5, as was the simple matrix storage method for storing matrix elements. Building on this, a nomenclature for completely describing how a particular matrix is stored will now be outlined.

The most basic classifications of the way in which a matrix is stored are which register the matrix is in, and which direction its rows and columns lie within the register. A matrix may be in either the X or Y registers, and it may or may not be transposed. The four possibilities resulting from these distinctions are shown in Figure 4.1 for an 8×4 matrix A. Each diagram shows a section of either the X or Y register for a system with physical array size $p = 5$, and illustrates where the various elements of A are located within it.

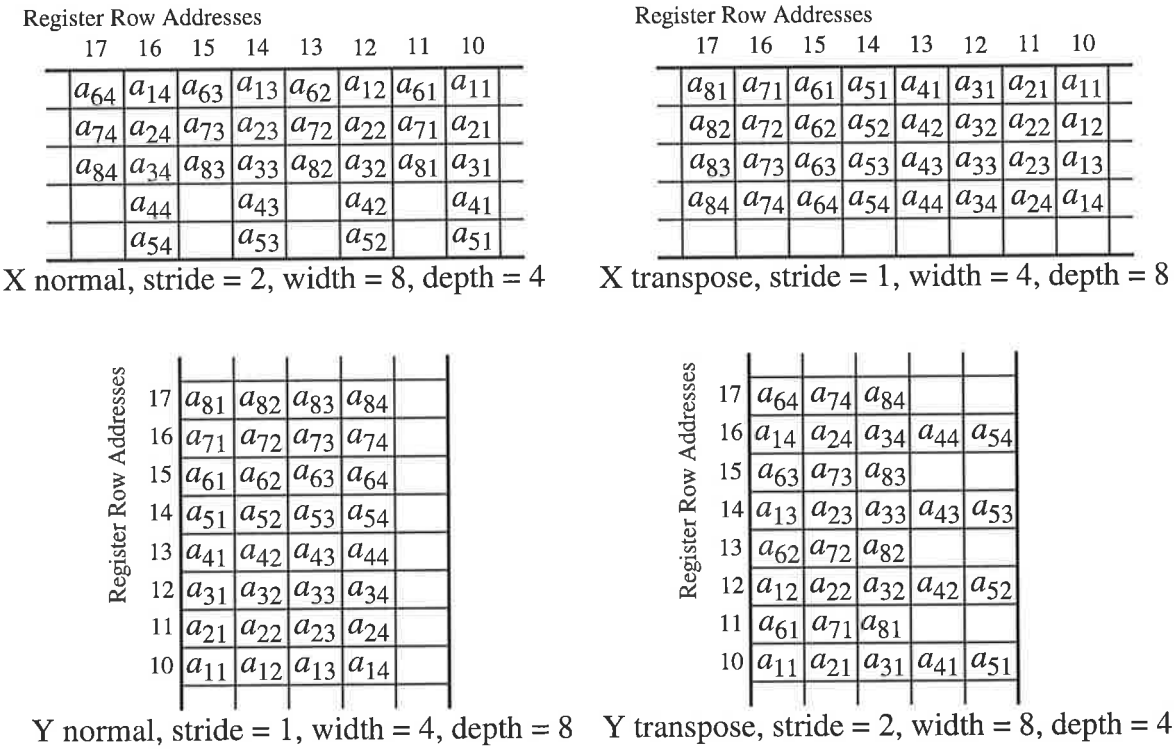


Figure 4.1: The Four Basic Matrix Storage Methods, base = 10, rows = 8, columns = 4, offset = 0

The type of storage used for a matrix determines whether the matrix rows or columns can be accessed directly, since the rows of a matrix do not necessarily align with the register

rows. Using the new terminology, it can be seen that in Y normal and X transpose forms the rows of the matrix lie along the register rows, and thus matrix rows, or sections of rows, are transferred to the array during computation. In X normal or Y transpose forms the columns of the matrix are transferred to the array during computation. This distinction is very important. For example, the matrix product AB is the sum of the outer products of the columns of A and the rows of B. When performing a matrix multiplication, the processor array forms the outer product of two register rows every array cycle, and adds them to the totals in the accumulators. Thus to form the product AB, A must be stored so its columns are aligned with the register rows, that is, in either X normal or Y transpose forms. Similarly, B must be in either Y normal or X transposed forms. Furthermore, the two matrices must be in different registers. It will be seen that ensuring that matrices are in the correct storage form is the most important and difficult task in programming the Load/Store architecture, and that it can result in the need to copy data from one register to another. This is the first example of an overhead of the Load/Store memory architecture.

Given the storage type, the exact location of a matrix within the register can be described using five numerical parameters. The first parameter, called the *base*, is the row address where the top left element, a_{11}, is located.

The second parameter is the *stride,* which is the difference between the row addresses of the rows containing any two matrix elements that are adjacent in the matrix and stored in the same register column. For example, in X normal storage, the matrix rows lie the direction of the register columns, and so the stride can be found by examining any two adjacent elements in the same matrix row, such as a_{11} and a_{12}. In the X normal example in Figure 4.1, the stride is 2 because the register row addresses of the rows that contain a_{11} and a_{12}, differ by two.

The third, and fourth parameters are the number of rows and columns in the matrix, or alternatively, the width and depth. The *width* is the size of the matrix in the direction that lies along register rows and is thus equal to the number of rows in X normal and Y transpose storage and the number of columns in X transpose or Y normal storage. The *depth* is the size of the matrix in the direction that lies along the register columns and is thus equal to the number of columns in X normal and Y transpose storage and the number of rows in X transpose or Y normal storage. Using the width and depth was found to provide a more concise description of most operations than using the row and column.

The fifth parameter required is the *offset*, which is the register column of the top left

matrix element, a_{11}, assuming that the columns are numbered from 0 to p-1. A matrix with zero offset is called *aligned*, and one with non-zero offset is *unaligned.* All the matrices in Figure 4.1 are aligned; an example of unaligned storage using X normal form is shown for the same example matrix A in Figure 4.2.

Register Row Addresses

21	20	19	18	17	16	15	14	13	12	11	10
a_{84}	a_{34}		a_{38}	a_{33}		a_{82}	a_{32}		a_{81}	a_{31}	
	a_{44}			a_{34}			a_{42}			a_{41}	
	a_{54}			a_{35}			a_{52}			a_{51}	
	a_{64}	a_{14}		a_{36}	a_{13}		a_{62}	a_{12}		a_{61}	a_{11}
	a_{74}	a_{24}		a_{37}	a_{23}		a_{72}	a_{22}		a_{71}	a_{21}

X normal

Figure 4.2: Unaligned Matrix Storage, base = 10, stride = 3, rows = 8, columns = 4, offset = 3

For the matrices shown thus far the parameters are redundant because the stride can be calculated from the width using,

$$\text{stride} = \left\lceil \frac{\text{width} + \text{offset}}{p} \right\rceil$$

A matrix for which the above equation is true is said to have *full stride*. However, this relationship does not always hold, especially when dealing with submatrices.

The storage of vectors and scalars is described in the same way as storage of matrices, but the number of either rows or columns, or both, is one. Treating these cases separately can result in fewer control processor instructions but not fewer array cycles, so this optimisation is not considered here.

4.1.2 Register Operations

There are four different classes of operations that may be performed on operands in the registers. These are;

- multiplication operations
- addition operations - this class also includes elementwise multiplication, division and square root
- scalar operations - both between scalars and between scalars and matrices
- submatrix operations.

In the following sections, the procedures by which these operations can be performed on the matrix processor will be explained.

4.1.2.1 Multiplication Operations

Formation of the product AB involves calculation of the sum of the outer products of the columns of A and the rows of B. A must be stored so that its columns are aligned with the register rows and B must be stored so that its rows are aligned with the register rows. Additionally, A and B must be in different registers. Thus either A must be stored in X normal form and B in Y normal form, or A must be stored in Y transpose form and B in X transpose form. The result of the calculation will be in the same format as one of the operands.

The way in which multiplication operates for aligned matrices can be seen in the abstract diagram in Figure 4.3, for a product $C = AB$ on a 5×5 processor array. The A and B matrix elements are shown as they are positioned within the registers, with the actual matrix elements shaded and the empty part of each register row blank. The B matrix in the Y register, for instance, is 10×11. The registers are shown 'unrolled' by the factor equal to the stride of the matrices being operated on, in this case three, so that register rows 0, 1, 2 are shown side by side, with register rows 3, 4, 5 below them. Thus, in the diagram adjacent matrix elements appear next to one another.

As the matrices are wider than the physical array, the product must be formed by partitioning the matrices and using a number of separate array sized operations. The theory of block matrix operations was described in §1.3.3, and in particular the case of a row-partitioned A multiplied by a column-partitioned B gives the result $C_{\alpha\beta} = A_{\alpha}B_{\beta}$. Referring to Figure 4.3, it can be seen that this result implies that if a partition from A is multiplied by a partition from B, the result will be the block of C that lies at the intersection of the two partitions in the diagram. In fact, each partition of C represents the result of a single multiply operation. The shaded entries in C show the processing elements within the array that produce a useful result.

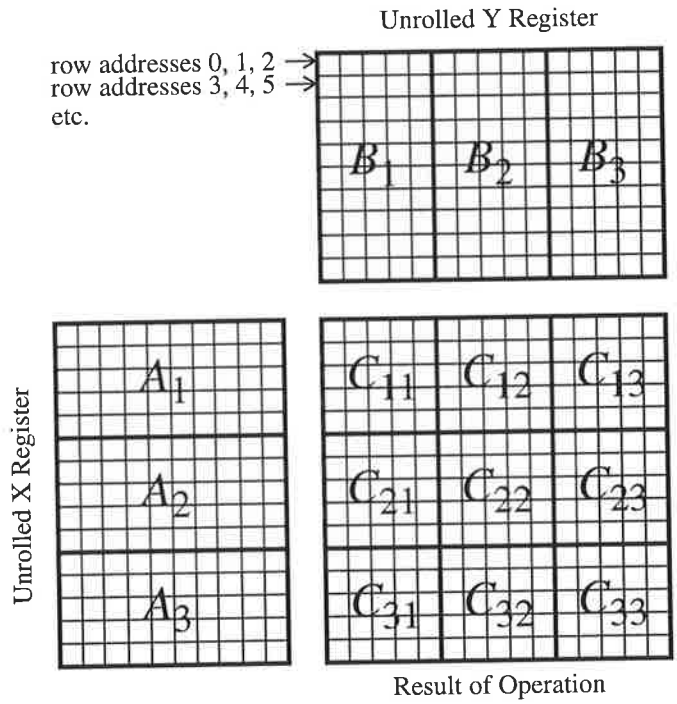


Figure 4.3: Aligned Multiplication

Multiplication with unaligned matrices is quite similar and is shown in Figure 4.4.

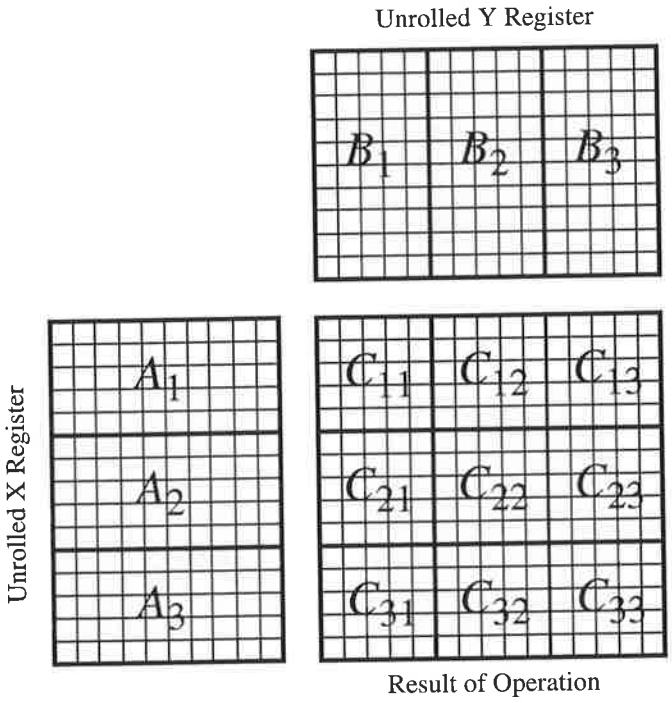


Figure 4.4: Unaligned Multiplication

It can been seen that unaligned multiplication can be performed in the same manner as aligned multiplication. The difference between aligned and unaligned multiplication occurs

when the blocks of C matrix are written back to the registers. The offset of the result will be equal to the offset of matrix A if it is written back to the X register, and equal to the offset of matrix B if written back to the Y register. It is important to note that two matrices can be multiplied regardless of their alignment.

By performing a multiplication by the identity matrix, assignment between a given matrix and another matrix in the other register with the same transposed-ness can be performed. In addition, if there is an unaligned matrix in a register that is multiplied by an aligned identity matrix in the other register and the result is written back to the second register, then a copy from an unaligned to an aligned matrix is performed. As the identity matrix is largely zeroes, the copy operation is best performed by dividing the matrix into array size blocks, which results in each element being transferred once.

The MATRISC architecture always writes back the results from all processing elements when performing a multiplication. This means that if two 2×2 matrices are multiplied on a 5×5 array then a 5×5 result will be written back to the registers. If care is not taken, then part of the register being used to store other information may be over written. When programming the architecture, this effect must be taken into account to ensure correct operation. This problem could be avoided by using a masking register that would enable or disable writeback to different register columns. Such a solution has been widely used in vector computers[Agerwala and Karp 84].

4.1.2.2 Odd Sized Multiplication

Multiplication that produces a result matrix whose dimensions are not a multiple of the virtual array size are termed *odd sized*. Each individual multiplication operation on the MATRISC architecture can be performed with a range of virtual factors. When a number of multiply operations are used to form a complete matrix product between two larger than array-sized matrices, it is possible to use a different virtual factor for each operation. Determination of the optimum virtual factors to use will now be considered.

For simplicity, a product with a square, $n \times n$, result and an architecture where a virtual factor of 1, 2 or 4 may be used, will be assumed. A higher virtual factor is faster when all the processing elements are performing useful calculations. Any multiplication result will have one *regular* portion that is calculated using the maximum virtual factor of 4. The rest of the matrix, forming a strip along two sides, could use a smaller virtual factor. The four possible cases are shown in Figure 4.5. The shaded area represents the result matrix

elements, the lightly printed grids are $p \times p$ sections of the result and the dark grids are the partitions into which the result is divided.

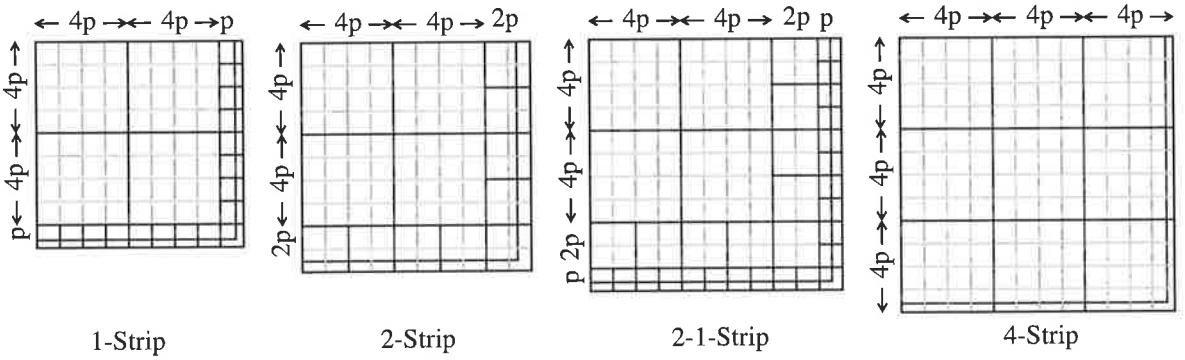


Figure 4.5: Methods for Calculating the Irregular Portion of an Odd Sized Multiplication

The time taken for each of these cases will now be calculated. The size of the array is n and the size of the regular portion is n_R. Note that calculating a $p \times p$ section of result, that is virtual factor 1, takes n cycles, virtual factor 2 takes $2n$ cycles and virtual factor 4 takes $4n$ cycles. The times taken by each method for the non-regular portion are;

$$\text{1-Strip\_time} = \text{time\_per\_1-block} \times \text{number\_of\_1-blocks} = n \times \left[2\left(\frac{n_R}{p}\right) + 1\right]$$

$$= \frac{2nn_R}{p} + n$$

$$\text{2-Strip\_time} = \text{time\_per\_2-block} \times \text{number\_of\_2-blocks} = 2n \times \left[2\left(\frac{n_R}{2p}\right) + 1\right]$$

$$= \frac{2nn_R}{p} + 2n$$

$$\text{2-1-Strip\_time} = \text{time\_per\_2-block} \times \text{number\_of\_2-blocks} + \text{time\_per\_1-block} \times \text{number\_of\_1-blocks}$$

$$= 2n \times \left[2\left(\frac{n_R}{2p}\right) + 1\right] + n \times \left[2\left(\frac{n_R + 2p}{p}\right) + 1\right]$$

$$= \frac{4nn_R}{p} + 7n$$

$$\text{4-Strip\_time} = \text{time\_per\_4-block} \times \text{number\_of\_4-blocks} = 4n \times \left[2\left(\frac{n_R}{4p}\right) + 1\right]$$

$$= \frac{2nn_R}{p} + 4n$$

These result shown that it is fastest to used the narrowest strip possible, except for the 2-

1-Strip, which take longer than the 4-Strip. Note that a wider strip than necessary may always be used but will require the use of more storage in the registers.

4.1.2.3 Addition Operations

Formation of the sum $A+B$ can be regarded as the summing the rows of A with the rows of B, or the columns of A with the columns of B. In the MATRISC architecture, matrix addition is performed using the leading diagonal processing elements, which implies that either the rows of both A and B or the columns of both A and B must run into the registers. Thus the allowable storage forms are A in X normal and B in Y transpose or vice versa, or A in X transpose and B in Y normal or vice versa. The result of the calculation will be in the same format as one of the inputs. The methodology described in this section also applies to elementwise multiplication and division.

Addition between aligned matrices is shown in Figure 4.6, which is similar to the figures for multiplication but with two important differences. First, as stated above, one of the matrices must be transposed, therefore the matrices are either both row partitioned or both column partitioned. Thus, referring again to §1.3.3, the sum must be formed by adding corresponding partitions, that is, adding A_1 to B_1, A_2 to B_2 and A_3 to B_3. The second difference is that the section of the diagram labelled 'Result of Operation' does not represent the actual result matrix, but instead represents the processing elements where the sum is formed. For example, assume that the A matrix is in X transpose form and hence that the partitions are column partitions. When the partitions A_1 and B_1 are added, the first column of each matrix intersect at the top left processing element. Similarly, the other corresponding columns in the partitions intersect at the other processing elements on the leading diagonal of the processor array. This illustrates that although all the processing elements perform the same operation of summing their two inputs every array cycle, it is only the elements on the leading diagonal that form the actual matrix sum.

Note that if the matrices A and B have full stride or have a depth of one, then the register rows that the matrices occupy are consecutive. It this case, the whole sum can be formed with a single addition operation rather than being partitioned. Performing the operation in this way requires the same number of array cycles but many fewer control processor instructions.

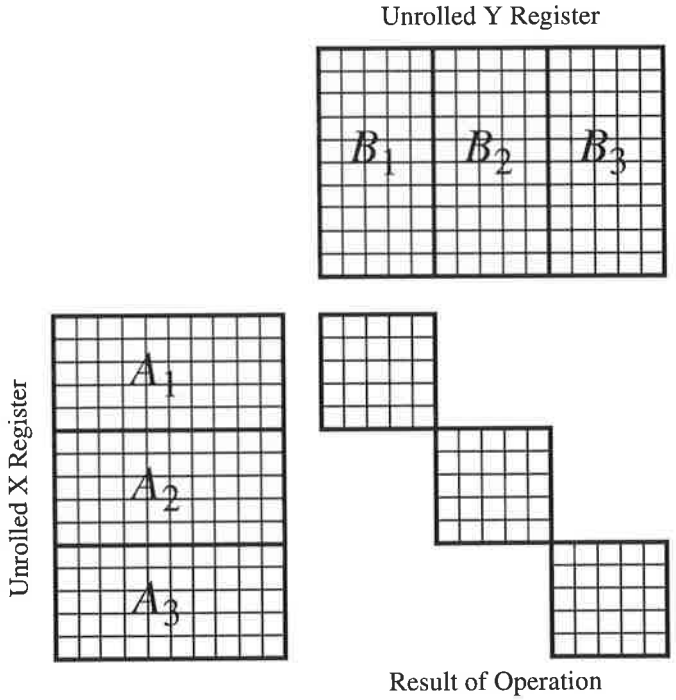


Figure 4.6: Aligned Addition

If the matrices being added are unaligned but the offsets are equal then the result still falls on the diagonal and so the addition is computed as described for aligned addition. The offset of the result is the same as the offset of the two inputs.

Unaligned addition with where the offset are not equal is shown in Figure 4.7. The result elements do not fall on the leading diagonal of the array because the offset of the two inputs are different and so this operation cannot be performed directly on the MATRISC architecture, because writeback from non-leading diagonals is not possible. An architectural enhancement allowing a writeback mode that would write back the results from the correct processing elements is possible. If this were done, a second problem arises. Consider the addition shown in Figure 4.7 and assume the writeback will occur to the Y register. The first two columns of the result are formed by doing an addition operation between A_1 and B_1. The third column of the result can be performed by an addition operation between A_2 and B_1. However, because results must be written back in whole register rows, this second addition will overwrite the first operation result with garbage. This problem can be solved by introducing a mask that selects only portions of a register row to be written back. Note that although these two modifications would make unaligned addition possible, it would still take approximately twice as long as aligned addition. So the operation could be performed just as quickly by using a multiply copy operation to change the alignment of

one of the matrices and then performing the addition; for this reason, the potential architectural modifications described here have not been used.

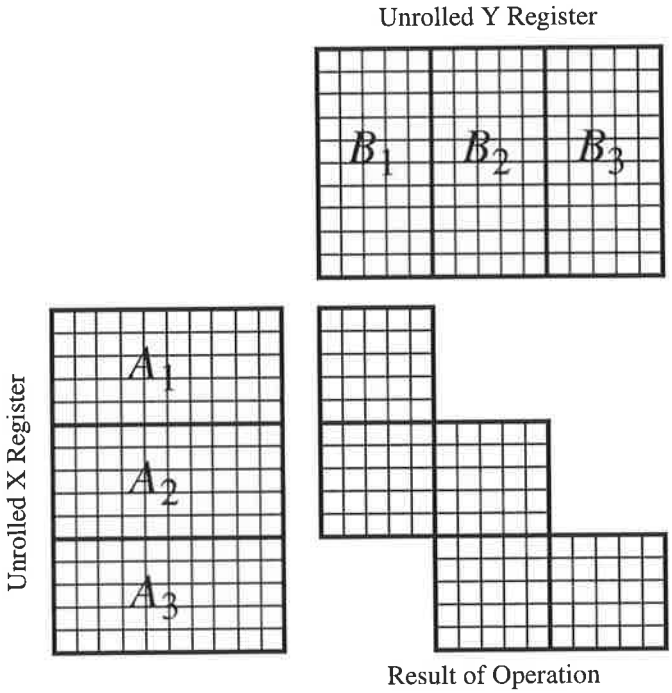


Figure 4.7: Unaligned Addition

By performing an addition with a zero matrix, assignment between a given matrix and another matrix in the other register in the relatively transposed storage form can be performed. Assignment between a given matrix and another matrix in the same form in the same register can also be performed. There is no way that any single operation can perform an assignment from one matrix to another relatively transposed in the same register.

4.1.2.4 Scalar Operations

Scalar operations exploit the ability of the MATRISC processor to perform a writeback from a row or column of processing elements, rather than from the leading diagonal, when performing an elementwise operation. The method is illustrated in Figure 4.8, which shows a matrix operand in the X register and a scalar operand in the Y register. Again, the result part shows the processing elements where the result is formed rather than the whole result matrix.

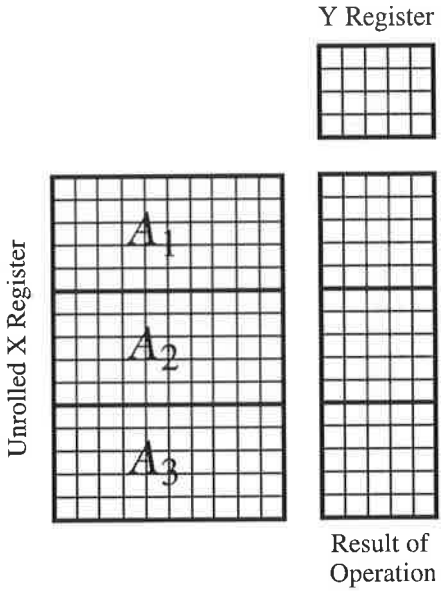


Figure 4.8: Scalar Operations

The storage type of the scalar is irrelevant, save that it must be in the opposite register to the matrix operand. The result matrix must be written back to the same register in which the matrix operand was stored, and it must be stored in the same format. The matrix operand may be unaligned, in which case the offset of the result will be the same as the matrix operand's offset.

4.1.2.5 Submatrix Operations

Submatrix operations are handled using a matrix description which refers to, or aliases, the desired submatrix. Two examples are shown in Figure 4.9, where the original matrix is shown light-shaded and the submatrix dark-shaded.

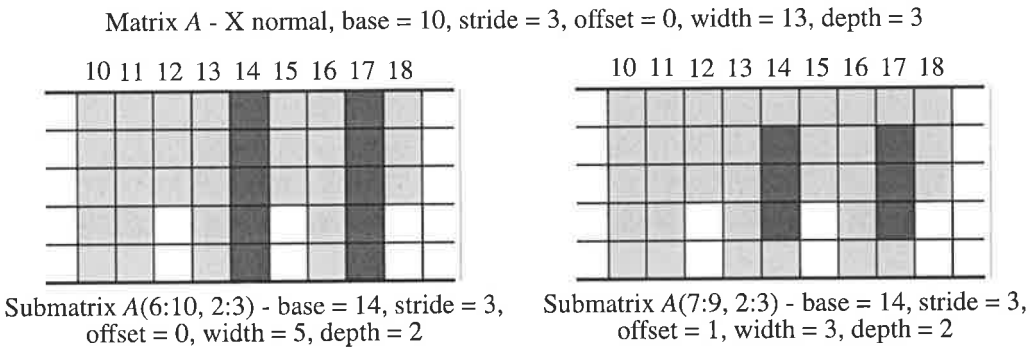


Figure 4.9: Submatrix Operations by Reference

The submatrix can then be used as the input or result of any matrix operation. However, care must be taken when assigning to a submatrix to avoid overwriting data outside the

UNIVERSITY OF ADELAIDE LIBRARY

submatrix. When results of a computation are written back to the registers, a whole register row is always written, so assigning a result to a matrix that does not fill a whole number of register rows must write some elements outside the matrix. In Figure 4.9, for instance, the submatrix on the left fills two whole register rows so assignment to it will not affect the elements of the matrix that lie outside the described submatrix. The submatrix on the right does not fill a whole matrix row so assignment cannot occur to this matrix without affecting other elements of A. Note that when matrix multiplication is performed, a $pv \times pv$ result is always written back regardless of the size of the inputs.

In practice, it often turns out that overwriting the outside elements is not important, for example, in Gaussian Elimination where the structure of the algorithm means that zero elements are simply assigned to zero again. However, this limitation of the architecture may degrade performance in some circumstances and can add considerably to the complexity of programming. Architectural modifications to remove this limitation, such as adding another control register to set the size of writeback, have not been explored because initial results suggested that data overwriting occurred rarely and resulted in only a small time penalty.

4.1.3 Main Memory Storage Methods

As main memory is standard, linearly addressed memory, all the normal scalar processing methods of dense matrix storage apply. The most obvious storage method is continuous storage in either row-major or column-major order. The inverted address generation scheme used in the architecture was specifically designed to allow matrices stored in main memory to use these storage methods and still allow flexible mapping into the registers. Whether row- or column-major order is more appropriate will depend on the mapping being applied as the matrix is loaded into the registers. If a choice is possible, it should be made to avoid applying a transpose mapping to a matrix, as these can slow the execution of computations. For simplicity, the current simulation tools always use row-major order.

4.2 Masm: MATRISC Assembly-like Language

Programming even basic algorithms at the level of the control processor instruction set is very tedious, so a number of tools have been developed to aid the process. These tools

include a compiler for implementing algorithms and a simulator to run the code and produce the result of the calculation and a number of performance metrics.

The principal programming tool is the *MATRISC assembly-like language* (Masm) compiler, which converts a very simple language with many assembly-like features into code for the *MATRISC simulator.* The simulator executes the control processor code defined in §3.7, and simulates the matrix data path as described in Appendix A. No allowance is made for stalls required to prevent data and structural hazards. Three performance metrics are produced. The *array cycle count* measures the number of array cycles taken to transfer data between the registers and the processor array. This figure directly measures the time required to complete the computational part of the algorithm. The *load/store transfer count* measures the number of words transferred between main memory and the registers. The *control processor operation count* measures the number of control processor instructions executed.

The Masm language is not a true assembly language in that it has very little to do with the code executed on the control processor. However, it provides very few high level language constructs except symbolic names. Its principal feature is to produce the control processor code for an entire matrix operation from a symbolic representation of the operation. For example, it produces the control processor code to perform a partition matrix product from the source code 'A = B * C'.

The language syntax is based on the three address code described in [Aho et al. 86]. The only flow control statements are conditional and unconditional goto. All computation statements have one or two input operands and a result, which makes the compiler very simple.

The following sections introduce various parts of the language, and describe their syntax using a series of grammar productions, see [Aho et al. 86]. The conventions for grammar rules are as follows: nonterminal symbols are shown in italics; terminal symbols are shown in bold, and are literally the text as written except for the symbols, **Identifier**, **Number** and **String**, which are parsed by a lower level lexical analyser; and the vertical bar represents alternative productions for a nonterminal symbol.

4.2.1 Top Level Syntax

At the top level, a Masm program consists of a sequence of declarations and statements terminated by semicolons. There are no functions or procedures, although their

functionality is supported somewhat by templates (see §4.2.3.3).

program → *list*
list → *list list_item* ; |
list_item ;
list_item → *statement* |
declaration

4.2.2 Definitions and Data Types

Masm definitions introduce variables of four types: integer variables, matrix register variables, matrix main memory variables, and matrix aliases.

declaration → **integer Identifier** |
matrix *storage* **Identifier (Number , Number)** |
matrix main Identifier
alias *storage* **Identifier**
storage → **Xstore | Xtstore | Ystore | Ytstore**

Some examples are,

```
integer i;
matrix Xstore A(100, 100);
alias Xstore B;
```

Integer variables are the most basic type, and correspond directly to integer values on the control processor. Masm statically allocates space for each of the integer variables defined. There is no need for a stack because no function calls are used. Typical uses for integer variables are as loop counters and for performing index calculations when operating on sub-arrays. Data path control registers appear in Masm as integer variables; assignment to which generates the appropriate SET instruction in the generated code for the control processor. The variable names are formed by the name of the register prefixed by a full stop (for example `.Xaddress`). This arrangement gives the programmer complete control when necessary, but is usually only required when debugging. Other special variables that are available are shown in Table 4.1 .

Special variable	Meaning
`.p`	Physical array size
`.v`	Maximum virtual factor

Table 4.1: Special Integer Variables

Matrix register variables effectively assign a name to part of one of the two matrix registers, which results in a static allocation of register space (discussed in §4.3). Note that the size specified for the variable determines how much register space is set aside for that variable, and thus defines the maximum size of the matrix data that can be stored in the variable when the program runs. At runtime, a smaller matrix may be stored in the variable. Additionally, the size of the stored matrix may change.

Each matrix register variable has five integer variables associated with it that hold the numerical parameters that describe the matrix, namely, the base, stride, width, depth and offset. The names of these integer variables are formed by the name of the matrix followed by a full stop, followed by the name of the parameter (for example `A.base`). The base and stride are constants since they describe the physical layout of the matrix register variable within the register, which does not change during execution. The other parameters can change since they are used to describe the size of the current contents of the matrix register variable at any time, which varies as the program runs and is rarely the same size as the static size declared in the register definition.

Matrix aliases are use to perform submatrix operations by reference. They have the same integer variables as matrix register variables, but all of them are variable. Matrix aliases allow submatrices to be used, but force the difficulties of submatrix operations onto the programmer. Matrix aliases together with matrix register variables, are collectively referred to as matrix variables.

Matrix main memory variables associate a name with an area of main memory. These variables are declared without a size because they are associated with a matrix data file, using the Masm load statement (see §4.2.3.4). Their size is generated from the matrix in the file.

4.2.3 Statements

Masm statements can be divided into four groups: arithmetic, flow control, template calls and miscellaneous.

4.2.3.1 Arithmetic Statements

Arithmetic statements on integer variables may perform the operations of addition, subtraction, multiplication, division, unary minus, and simple assignment. Arithmetic statements on matrix variables may perform the operations of addition, subtraction,

multiplication, division, unary minus, transposition and simple assignment. Separate symbols are use for the scalar and elementwise versions of the matrix operators.

statement → **Identifier** = *value* |
Identifier = *value operator value* |
Identifier = **sqrt** *value* |
Identifier = *value* ‘
value → **Identifier** | **Number**
operator → + | - | * | / | .* | ./ | ++ | -- | ** | //

Some examples are;

```
C = A * B;
A = A // b;
row = row + 1;
```

The meanings of the binary operators are shown in Table 4.2. Note that some operators are restricted to either integer or matrix values, and that there are separate operators for scalar operation on matrices.

Operation	Meaning for integers	Meaning for matrices	Matrix operation class
+	Addition	Matrix addition	Elementwise
-	Subtraction	Matrix subtraction	Elementwise
*	Multiplication	Matrix multiplication	Conformal
/	Division	Not allowed	
.*	Not allowed	Elementwise multiplication	Elementwise
./	Not allowed	Elementwise division	Elementwise
++	Not allowed	Scalar addition	Scalar
--	Not allowed	Scalar subtraction	Scalar
**	Not allowed	Scalar multiplication	Scalar
//	Not allowed	Scalar division	Scalar

Table 4.2: Meaning of Masm Binary Operators

When using the Masm language, the programmer is responsible for ensuring that certain conditions are met so that matrix operation can be performed correctly. These restrictions ensure that any Masm matrix arithmetic statement maps directly to an operation that the MATRISC processor can perform without the use of any temporary storage. The requirements are shown in Table 4.3 for the three classes of matrix operations, and are

expressed for an imaginary operation R = A op B. The storage forms are abbreviated, with X being X normal form, Xt being X transpose form, and similarly for the Y register forms.

Operation class	Allowable storage (A-B-R)	Dimension restrictions
Elementwise (+, -, .*, ./)	X-Yt-Yt X-Yt-X Yt-X-Yt Yt-X-X Xt-Y-Y Xt-Y-Xt Y-Xt-Y Y-Xt-Xt	A.width = B.width = R.width A.depth = B.depth = R.depth A.offset = B.offset = R.offset
Conformal (*)	X-Y-X X-Y-Y Yt-X-Yt Yt-X-X	A.depth = B.depth When A and R in same register A.width = R.width B.width = R.depth otherwise vise-versa
Scalar (++, --, **, //)	A scalar X-Y-Y Y-X-X X-Yt-Yt Y-Xt-Xt Xt-Y-Y Yt-X-X Xt-Yt-Yt Yt-Xt-Xt	A scalar B.width = R.width B.depth = R.depth

Table 4.3: Storage and Dimension Restrictions for Binary Matrix Operations

In addition to these restrictions, the programmer must be aware that all operations write back to the registers in blocks. Any elementwise or scalar operation must write back a whole p element row of the register, and conformal operations (multiply) must write back a $pv \times pv$ block of the register. If the result of an operation does not have a zero offset or is smaller that the size of the writeback the operation will therefore alter part of the register that is outside the result. In practice it turns out that this is rarely a problem because the structure of algorithms means that the overwritten element is no longer needed, or is overwritten with the same value that it already contains.

Matrix assignment operations are actually performed using addition to zero or a modified multiplication by identity. Templates handle the details of the operation so the programmer does not need to explicitly use the zero and identity matrices, but the restrictions for addition and multiplications still apply to the source and destination. In particular, the source and destination must be the same size.

Whenever a matrix is the result of an operation, its width, depth and offset are set appropriately given the size of the inputs so that they describe the new contents of the matrix. When performing multiplication with inputs that are not aligned, the base of the result matrix may also be altered.

4.2.3.2 Flow Control Statements

The flow control statements used in Masm are conditional goto, unconditional goto and labels. The conditional goto can test any integer variable or the data path control flag by using the special name `.dflag`.

statement → **if** *value relop value* **goto Identifier** |
goto Identifier |
Identifier :
relop → == | ~= | < | <= | > | >=

Some examples are;

```
Loop:;
if i > 10 goto End;
goto Loop;
End:;
```

4.2.3.3 Template Call Statements

Templates are used by the compiler to replace matrix operations with equivalent integer commands for the control processor. Templates are effectively functions that are always expanded in-line. They have matrix and integer parameters, which are always passed by reference, and local integer variables. Most of the templates have restrictions on the storage forms of their matrix parameters, so there are several versions to cover all allowable possibilities. The compiler chooses the appropriate template version. Templates are compiled separately, then hardwired into the compiler.

Templates can also be called directly by Masm code using a sequence of param statements followed by a call statement.

statement → **param Identifier** |
call Identifier |
Identifier = alias *submatrix* |
print Identifier
submatrix → **Identifier (** *range* **,** *range* **)**
range → *value* **:** *value* | *value* | **:**

Some examples are;

```
row = alias A(i, 0:lastcol);
param A;
param B;
call .T_operation;
```

Two particular template calls are so important that they have their own syntax. The alias

statement makes the result, which must be an alias type variable, refer to the submatrix described. The print statement prints the given matrix on the simulator's standard output, and is present for debugging purposes only.

4.2.3.4 Miscellaneous Statements

There is only one miscellaneous statement in the Masm language. The load statement is used to link a matrix main memory variable to an operating system file.

statement → **load Identifier String**

An example is;

```
load D 'gauss.mat'
```

4.2.4 A Simple Example

The simple program below shows how the Masm language can be used. In this example, two matrices are loaded and multiplied together, and then the top left 3×3 section is printed out.

```
matrix Xstore A(20, 20);
matrix Ystore B(20, 20);
matrix Ystore C(20, 20);
matrix main Amain;
matrix main Bmain;
alias Xstore P;

load Amain 'A.mat';
load Bmain 'B.mat';
A = Amain;
B = Bmain;
C = A * B;

P = alias C(0:2, 0:2);
print P;
```

More complex examples will be described when larger algorithms are developed.

4.3 Programming Methodology

Programming the MATRISC architecture using the Masm language is not straightforward, because the programmer has to make many decisions about how to use the matrix register space, and must ensure that all matrix operations meet certain storage

requirements. In Masm terms, the programmer has to declare a number of matrix register variables, which can be used to implement the desired algorithm. Each matrix arithmetic statement must then be written to obey the storage type and dimension restrictions in Table 4.3. To achieve this, the following four-step programming methodology was developed.

- **Express algorithm in Matlab Code** - This gives a definite starting point for deriving the Masm code.
- **Determine a set of matrix register variables** - No formal method has been developed for this stage. Instead, a simple heuristic procedure is used as follows. First, a matrix register is assumed for the matrix variables in the Matlab code, possibly adding in extra register variables where a temporary variable will obviously be needed. Second, the storage requirements for the matrix operations are listed. Third, a search is made for a set of matrix storage types that satisfy all the requirements. This step could use an exhaustive search of all permutations but a logical argument can often be used instead. If there is no solution, extra matrix register variables and/or extra operations are added until a solution is possible. The final step is to determine a size for the variables, which must be based on knowledge of the algorithm.
- **Express algorithm using the matrix register variables** - Once the register variables have been determined, the original Matlab code can be re-expressed in terms of the variables, and any extra operations required to satisfy the matrix storage requirement can be inserted. The resultant code is called *MATRISC compliant* Matlab code, and is distinguished by the fact that all the matrix operations have a one to one correspondence with operations performed by the architecture when the final Masm code is executed. Careful examination of this code can be used to determine the array cycle count for the algorithm.
- **Translate into Masm code** - Finally, the Masm code can be written, which involves implementing loops and if-then-else statements using goto's, and translating integer calculations to be free of the need for intermediate variables. When algorithms are presented in the following chapter, the results of this step are omitted as there is no useful information that can be obtained from it. It would be straightforward, although time consuming, to develop a compiler for MATRISC compliant Matlab code.

This methodology is by no means the kind of logical procedure that can be easily implemented in a compiler. However, it suffices to allow a human programmer to implement relatively complex algorithms, certainly to the level of linear equation solvers for instance, which can then be used as part of a larger application. Improved methods of programming the architecture and incorporating them in a compiler is beyond the scope of this thesis, but would form an important part of future work on the MATRISC Load/Store architecture.

4.3.1 Programming Example

To illustrate the programming process, a small example will now be worked through in detail.

4.3.1.1 Express Algorithm in Matlab Code

The algorithm to be implemented is the block outer product update from the block Gaussian Elimination algorithm (see §5.4.3). Although the code below appears simple, this example demonstrates the subtleties of programming the MATRISC architecture. The matrix A is the matrix of equations being solved by the algorithm. The variables i and be are the row indices of the beginning and end of the block respectively. Due to the structure of the rest of the algorithm, all the matrices in this code fragment are aligned.

```
Z = A(i:be, be+1:)
W = A(be+1:, i:be)
A(be+1:, be+1:) = A(be+1:, be+1:) - WZ
```

4.3.1.2 Determine a Set of Matrix Register Variables

To assist in describing the process whereby the register matrix variables are derived, the code is re-expressed below. This step is not usually required, but is shown here for clarity.

```
Z = A(i:be, be+1:)
W = A(be+1:, i:be)
C = WZ
A(be+1:, be+1:) = A(be+1:, be+1:) - C
```

The storage restrictions implied by these lines of code are as follows.

- W and Z can be in any form except the transpose of the form A is in, because no single operation can copy from one storage form to the transpose of that form.
- Either W must be in X normal form and Z in Y normal form, or W must be in Y

transpose form and Z must be in X transpose form. C must be in the same form as either W or Z.

- C must be in the opposite register and have the relatively transposed storage form to A.

This last restriction sets the storage form for C, when the storage form of A is known. In turn, the second restriction implies storage forms for W and Z. The storage forms of C, W, and Z, given the four possibilities for A are shown in Table 4.4.

A	C	W	Z
X normal	Y transpose	Y transpose	X transpose
X transpose	Y normal	X normal	Y normal
Y normal	X transpose	Y transpose	X transpose
Y transpose	X normal	X normal	Y normal

Table 4.4: Storage Combinations for Block Gaussian Outer Product Step

Note that in every case, either W or Z is in the transposed form of A thus violating the first storage restriction. Therefore, an extra copy operation will be required at some point. For a Gaussian Elimination on a square matrix A, Z and W are almost exactly the same size and both are smaller than C or A, so it is most efficient to copy one of these two matrices. Because of the mirror symmetry between the registers there will always be at least two solutions, if there are any, to problem of choosing storage forms, and sometimes four solutions.

As the implementation of the Gaussian Elimination algorithm in the following chapter chooses A to be in Y normal form, that form will be used here also. Now that the storage form has been fixed, the size of the matrix registers must be determined. Rather than picking definite sizes, shapes, such as square, single row or single column, are chosen instead. Exact sizes for the different shapes are determined when producing the final Masm code. The chosen set of register variables is displayed diagrammatically in Figure 4.10.

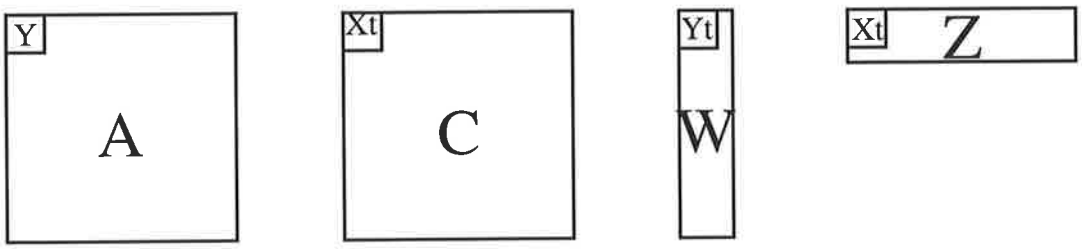


Figure 4.10: Matrix Register Variables for Block Gaussian Outer Product Step

Matrix register variable A is chosen to be square, because it holds the matrix being reduced which is usually square. C is also square because it should be the same size matrix as A. W has as many rows as A but is only as wide as the block size. Similarly, Z has as many columns as A but is only as high as the block size.

4.3.1.3 Express Algorithm Using the Matrix Register Variables

Now the algorithm can be expressed in terms of these matrix register variables, as follows;

```
C = A(i:be, be+1:)
Z = C
W = A(be+1:, i:be)
C = WZ
A(be+1:, be+1:) = A(be+1:, be+1:) - C
```

Note that each line can be achieved by a single matrix calculation.

4.3.1.4 Translate into Masm Code

Finally the algorithm can be expressed in Masm code;

```
matrix Ystore A(200, 200);
matrix Xtstore C(200, 200);
matrix Ytstore W(200, 20);
matrix Xtstore Z(20, 200);
alias Ystore Ablock;

integer i;
integer be;
integer last_column;
integer last_row;
integer be_plus_1;

% In the full Gaussian Elimination code here loads the matrix A
% performs initial calculations

be_plus_1 = be + 1;
Ablock = alias A(i:be, be_plus_1:last_column);
C = Ablock;
Z = c;

Ablock = alias A(be_plus_1:last_row, i:be);
W = Ablock;

C = W * Z;

Ablock = alias A(be_plus_1:last_row, be_plus_1:last_column);
Ablock = Ablock - c;
```

The Masm code is verbose due to the limitations of the compiler. Fortunately there is

nothing of interest to be discovered from this code that cannot also be found in the preceding, more concise, MATRISC compliant Matlab code.

4.4 VHDL Simulation

In order to examine some of the issues of timing within the Load/Store architecture, a relatively detailed simulation in VHDL was written. The simulation was structural at the level of Figure 3.2 and behavioural below that, except that it did not include the control processor. Instead, the matrix controller was driven by a series of instructions derived by hand. The principal area of interest was in the operation of the array itself, particularly the interleaving of read/write and load/store accesses to the registers.

In order to implement the simulation, the detail of the design of the architecture, as expressed in Chapter 3, had to be expanded. The following sections describe the internal operation of the MATRISC controller and the data controller that were used in the VHDL simulation. The simulation assumed SRAMs without delayed write, so read/write accesses used a technique called framing, as described in the following section.

4.4.1 SRAM Bus Turnaround

Because of the large register size required to implement many of the target applications, commercial SRAM chips are the only feasible choice for building the registers.

However, the registers are expected to perform multiple read and write operations concurrently, so an extra layer of control and buffering is required on top of the SRAM in the form of the data controller. The data controller buffers the data between the RW bus attached to its row or column of processing elements and the SRAM, thus alternate read and write operations are performed by the SRAM.

At the time when the VHDL simulation was being performed, the SRAM that was a likely candidate for building the registers were 5ns access synchronous flow-through SRAMs[IBM 95]. These chips' read cycle operates by sampling the address on one rising clock edge and presenting the data by the next rising clock edge. This behaviour results in a single read operation taking two clock cycles. Fortunately, due to pipelining, two reads only takes three cycles, and in general n reads take n+1 cycles, so large blocks of read operations are most efficient. Unfortunately, because the SRAM is being used as a register, accesses to

it alternate between reads and writes.

If the reads and writes were strictly alternated, each pair would take 3 cycles, which results in only 66% of the SRAM bandwidth being used. To improve the utilisation of the SRAM, a number of reads can be performed followed by the same number of writes; such a grouping will be called a *frame*. Using frames with two to eight reads together results in 80%, 85.7%, 88.9%, 90.9%, 92.3%, 93.3% and 94.1% utilisation of bandwidth respectively. A substantial improvement is observed by grouping of just two reads. Conversely, for a groupings of above four reads, the law of diminishing returns quickly limits any further improvement. Thus in a binary world, grouping either two or four reads together would be most appropriate. As the complexity increases greatly with larger groups, a size of two was chosen for the simulations.

There is an alternative available in the form of new SRAM chips with a delayed write capability. Delayed write means that the address of a write operation can be supplied in one cycle and the data to be written in the next cycle, which allows reads and writes to be interleaved in alternate cycles. Possible data hazards are handled internally by the SRAM itself. Using such chips, framing is no longer necessary, significantly reducing the complexity of the data controllers. At the level of abstraction used in the functional MATRISC simulator, the distinction between these two types of SRAM is not important, but it is only relevant to the more detailed VHDL simulation. The VHDL simulation uses framing because that was required at the time is was developed.

Multi-ported memory devices were not considered principally because of their size, which lags significantly behind the largest fast memories available. Large registers are critical to the viability of the Load/Store MATRISC architecture. The data controllers effectively make each register column an multi-ported device that is specifically tailored to the rest of the memory architecture.

4.4.2 Matrix Controller

The most complex part of the architecture is the matrix controller, which is not surprising as it must control the operation of a large parallel system. Internally, the matrix controller can be divided into five sections as shown in Figure 4.11. Note that the simulator currently uses a behavioural model of the MATRISC controller, with the internal sections shown in the diagram implemented as separate processes.

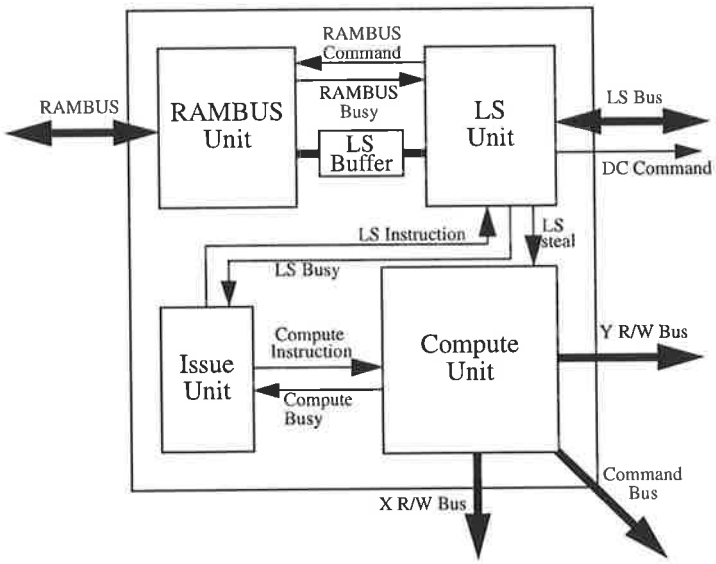


Figure 4.11: Internal Structure of the Matrix Controller

4.4.2.1 Issue Unit

The issue unit loads instructions from a file and passes them to either the LS unit or the compute unit depending on the type of the instruction. If the required unit is busy, the issue unit waits until it is ready. Synchronising instructions are held in the issue unit until the appropriate conditions are met, and then discarded. A more advanced method would be to allow out-of-order issue, which, for example, would allow a compute instruction to "leap-frog" over a load instruction that was stalled because the LS unit was still busy with a previous load. The method was not used because the simpler approach was adequate for the VHDL simulation to fulfil its aim of allowing investigation of the timing of data transfers within the array.

4.4.2.2 RAMBUS Unit

The RAMBUS unit transfers data between the RAMBUS DRAM and the internal LS buffer. It is clocked by the RAMBUS clock and is slaved to the LS unit, in that it performs commands specified by the LS unit. The most substantial part of the unit is the DRAM address generator and the logic required to divide a large operation into legally sized RAMBUS transactions.

4.4.2.3 LS Buffer

The LS buffer is required because the RAMBUS unit and the LS unit operate at different clock frequencies and because both are subject to unpredictable stalls. In the case of the

RAMBUS unit, the stalls are due to misses when accessing the RAMBUS DRAMs. In the LS unit, stalls are due to waiting for the beginning of a frame to perform a steal. The buffer thus acts as an ideal source or sink of data for each unit, the only limit being that the RAMBUS unit cannot begin a read transaction if the data may overflow the buffer, nor may it begin a write transaction until all the required data is in the buffer. This introduces some possibly unnecessary latency, but simplifies the control by guaranteeing predictable behaviour. The RAMBUS and controller clocks are in a 3:2 ratio of frequency and are synchronised.

4.4.2.4 LS Unit

The LS unit transfers data between the LS bus and the LS buffer. During a load operation, each data element from the LS buffer is transmitted along the LS bus in parallel with an address produced by the register address generator. The number of elements sent to each data controller is recorded, which is necessary because the buffer in each data controller can only hold two elements. When the next address would result in a third element being sent to the same data controller, the operation is stalled until the beginning of a frame so that cycles may be stolen to move the data from the buffers in the data controllers into the SRAMs. During a store, the addresses are generated and sent to the data controllers via the LS bus. A record is kept of the order in which the different data controllers are accessed. When a third address is about to be sent to the same data controller, the operation is again stalled and cycles stolen to move data from the SRAMs into the buffers in the data controllers. This data is then extracted from the data controller buffers in order the addresses were sent, and pushed into the LS buffer. At the same time, the next set of addresses are sent to the data controllers. There are two subtleties: firstly, a transfer from SRAMs to data controller buffers must not occur until the previous contents of the buffers have been extracted; and secondly, the new record of the data controllers accessed must not overwrite the old one until the corresponding elements have been extracted.

4.4.2.5 Compute Unit

The compute unit can be divided into sections as shown in Figure 4.12, which also shows the major communication paths but not all control lines.

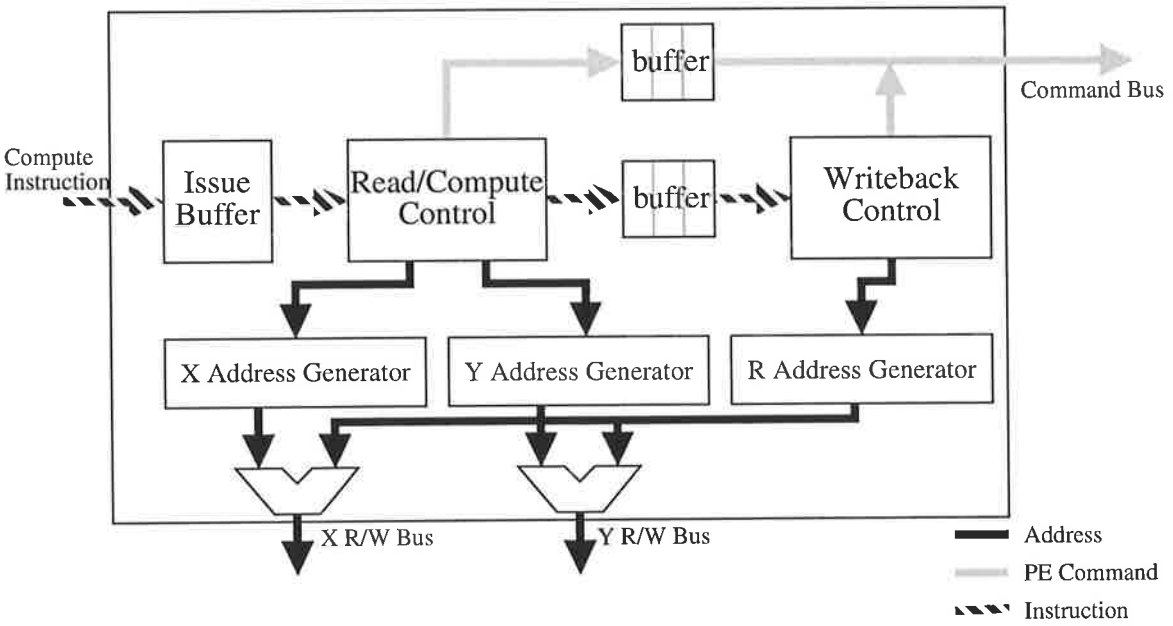


Figure 4.12: Compute Unit

The read/compute control initialises the read (X and Y) address generators, generates the compute part of the PE command and moves the instruction on to the writeback control when necessary. The writeback control initialises the write (R) address generator and generates the writeback part of the PE command. Buffers are needed at a number of points to compensate for latencies within the processor array, and to align read and write addresses correctly with respect to the frame clock.

4.4.3 Data Controller

The purpose of the data controller is to regulate the flow of data into and out of the register SRAMs. In particular, it must allow access for loads and stores, and for reads and writes. The data controller acts as specialised "front end" for the SRAM, and is on a separate chip only because the custom fabrication of large SRAMs is impractical. The diagram of the internal structure of the data controller is shown in Figure 4.13. Note that this diagram excludes internal control lines from the control logic to the various buffers and also the control lines to the SRAM.

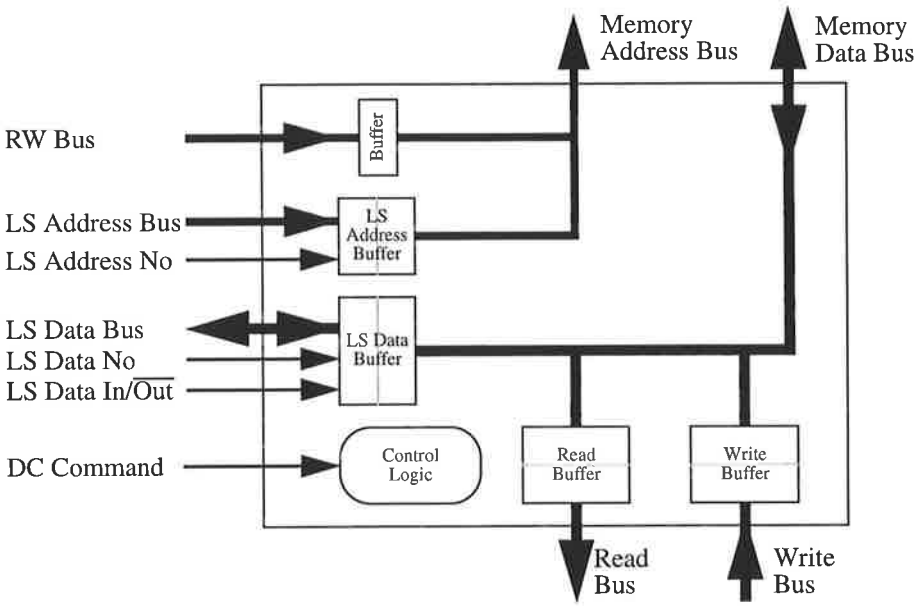


Figure 4.13: Internal Structure of the Data Controller

The LS Address Buffer stores the addresses to be used in subsequent load or store operations. It latches the value from the LS Address Bus whenever the LS Address No matches the number of the particular data controller. The buffer has space for two addresses, and the one which is latched depends on the LSB of the LS Address No. The LS Data Buffer performs similarly except that its contents may be transferred to or from the bus depending on the value of the LS Data In/$\overline{\text{Out}}$ line. The read and write buffers provide temporary storage for data travelling into and out of the processor array. They are necessary because of the different clocks used by the memory bus and the read/write buses.

The data controller is quite a simple piece of hardware, as it consists only of a small number of fast buffers and basic control hardware. However, the requirement for low latency and a large number of pins may present difficulties in implementation.The latency problem can be reduced by reorganising the timing to allow more time for the data to pass through the data controller. The pin count problem can be reduced by using narrower buses that are clocked faster, for example, clocking at 5ns would give an estimated pin count of 180, which is not large by current standards.

4.5 Summary

This chapter introduced a number of methods and tools that are used to implement

complex algorithms on the Load/Store MATRISC architecture, and described how the algorithms in Chapter 5 can be performed on the architecture.

An exact method for describing how a matrix is stored in the registers was defined. Each matrix is described using a storage form, which can be either X normal, X transpose, Y normal, or Y transpose, and 5 integer parameters called base, stride, width, depth and offset. It was demonstrated how operations on arbitrarily sized matrices could be performed on a fixed size array by partitioning the matrices. The design of the matrix registers was seen to lead to restrictions on the storage forms and offsets of the inputs and results of different operations.

A simple language, called Masm, has been developed for implementing algorithms on the architecture. A Masm compiler has been written that converts Masm into control processor code, as defined in §3.7, for execution on a high-level simulator of the architecture. The language is very limited, and in particular it requires the programmer to handle all the decisions about how to used the available register space, which is achieved by using matrix register variables that statically allocate an area of one of the registers. The programmer must also ensure that all matrix operations performed adhere to the storage form and offset requirements.

A methodology for using Masm to implement complex algorithms has been designed; it consists of four steps. First, the algorithm is expressed in Matlab code. From this description, a set of matrix register variables is then derived. The algorithm is re-expressed using Matlab in terms of the matrix register variables in such a way that all the operations obey the storage form and offset restrictions; this form is called MATRISC compliant Matlab code. Finally, the MATRISC compliant Matlab is converted into Masm code. No way to automate this procedure has yet been developed; the principal difficulty lies in the fact that to production of efficient final code must rely on specific properties of the algorithm that cannot be automatically deduced, in any simple way, from the description of the algorithm.

The final part of this chapter detailed a VHDL model of the Load/Store MATRISC architecture that contains all the major components of the architecture, as shown in Figure 3.2, with the exception of the control processor. This model that will be used to perform detailed simulations, which focus on verifying the timing of control signals.

Chapter 5

Simulation

This chapter describes the simulation of the Load/Store MATRISC architecture for a number of different operations and algorithms. The goal of this simulation is to come to a better understanding of the architecture, which will be achieved by;

- Expressing the performance of the architecture using appropriate metrics in terms of a compact set of architectural parameters.
- Isolating cases where specific features of the architecture particularly degrade or improve performance, with an eye to improving the architecture.
- Gauging the performance of the architecture on different algorithms and problem sizes in order to identify its strengths and weaknesses.
- Estimating real world performance of an implementation of the architecture using currently available technology.

5.1 Performance Evaluation

When estimating the performance of any architecture it is important to choose a set of appropriate metrics. The principal metrics for the Load/Store architecture are array cycle count, load/store transfer count and control processor operation count. These metrics measure respectively the time required to perform the calculation on the processor array, the time required to transfer data between main memory and the registers, and the time required for algorithm control.

Generally, the data transfer and the control processor operations occur in parallel with the computations in the array. However, when one of these activities depends on another

then it may have to stall. The MATRISC simulator described in Chapter 4 is not sufficiently detailed to predict the extra amount of time required by a particular algorithm due to stalls. However, the possible impact of stalls on performance can be assumed to be negligible for the following reasons.

Most algorithms proceed by loading some data into registers, operating on it, and finally storing the result. When this is the case, the MATRISC architecture can overlap the computational and data transfer sections of a series of calculations so that the rate at which results are produced depends only on the greater of the time taken for computation and the time taken for data transfer. This approach implicitly assumes a concern with throughput, rather than latency, when determining performance.

In the absence of data-dependent branching, the control processor produces a series of SET and DO instructions for the matrix controller that are in no way dependent on the operations of the matrix data path. Given a large enough buffer to hold these instructions, the time taken will be the greater of the time required for computation and the time required for control. In the presence of data-dependent branching, the assumption of no stalls is much less accurate.

If stalls are ignored, the time for the operation to be completed can be taken as the longest of the time required for computation, the time required for data transfer and the time required for control. The desired situation is that the computation should require the most time; when this is the case, the processor is compute-bound, which implies that the memory architecture is supplying data sufficiently quickly that the highest computational rate is achieved. In general for any algorithm, there will be a range of problem sizes that are compute-bound.

5.1.1 Performance Metrics

In order to examine the behaviour of the simulated algorithms, up to four values will be calculated. These are:

- the number of array cycles required to compute the algorithm result using the Load/Store architecture. This figure measures the expected performance of the architecture when data transfer and control overheads are ignored. By assuming an appropriate cycle time and a required floating point operation count for the algorithm being performed, a performance figure in floating point operation per second can be found.

  It is critical to note that all calculations of floating point performance reported in this

thesis are required flop/s. This means that they are calculated using the theoretical minimum required floating point operation count for the operation, rather than the actual number of floating point operations used by the algorithm used to perform the operation. Using this method implies that comparing the flop/s rate for two different algorithms, or sets of architectural parameters, that perform the same operation on the same data is equivalent to comparing running times. For different operations, or data, the flop/s rate provides a measure of the speed that the architecture performs the operation, verses an intrinsic measure of the amount of work that the operation requires.

- the number of array cycles required to compute the algorithm result using a hypothetical *perfect memory architecture*. It is possible to identify situations where extra array cycles are used because of the limitations of the Load/Store memory architecture, such as when data is copied between registers simply to convert it into a required storage form, or when a matrix is unaligned. By calculating a separate array cycle count without these overheads, the penalty of using the Load/Store memory architecture can be measured. A perfect memory architecture would correspond to a Memory/Memory architecture that never stalled for cache misses. Such a system would be very difficult to build if it were to run at the same speed as the Load/Store architecture, and would certainly be more expensive in terms of hardware resources.

  By comparing this value to the array cycles for the Load/Store architecture an overhead for the Load/Store architecture compared to the perfect memory architecture can be found.
- the time required to perform the load/store transfers between the registers and main memory, expressed as a percentage of the time required for computation. This figure is found by calculating the number of load/store transfers and converting it into an equivalent number array cycles by multiplying by F, the ratio of data controller to load/store bandwidth, which allows direct comparison between the computation and data transfer time. The time required to load or store an $n \times m$ matrix, expressed as array cycles is given by,

  $$\text{LoadStoreArrayCycles}(n, m) = Fnm$$

  In turn this figure is then converted to a percentage of the computational time by dividing by the number of array cycles required to compute the algorithm result.
- the amount of register memory required by the algorithm. Register space is limited and

when a given algorithm requires more space than is available, a new algorithm that partitions the data into appropriately sized blocks must be used. Partitioned algorithms require more load/store cycles and often more array cycles than an unpartitioned algorithm, which results in a complex space/time trade-off.

In order to simplify the presentation of results, a particular set of architectural and implementation parameters (shown in Table 5.1) is used as a standard. These parameters represent a system that would be feasible in late 1990s technology.

Array Size, p	5
Maximum Virtual Factor, V	4
Array cycle time, T_A	20ns
Load/store Cycle Time, T_{LS}	15ns
Controller Cycle Time, T_C	10ns
Load/store bandwidth factor, F	0.75
Control bandwidth factor, G	0.5

Table 5.1: Standard Simulation Parameters

The array size of 5 was chosen after initial results indicated that a smaller array would be appropriate. The array cycle time is based on 10ns SRAM for the registers, since the registers must be able to do one load and one store in each array cycle. The load/store cycle time is based on the original RAMBUS standard, which operates byte-wide at 533MHz, thus taking 15ns to transfer a 64bit double precision floating point number. The maximum virtual factor is assumes a 5ns cycle time for the adder-multiplier in the processing element. The virtual factor is the ratio between the speed of the computational elements and the array cycle time. The controller cycle time is based on a 100MIPS processor.

5.1.2 Performance Evaluation Method

To obtain values for each of the performance metrics, a detailed implementation of the algorithm is first derived in the Masm language for the Load/Store MATRISC architecture. Expression in Masm allows identification of overheads, with respect to the perfect memory architecture, that may not be evident from an implementation of the algorithm on a scalar processor. The possible types of overhead are:

- copy operations required because a matrix is in the wrong storage format, or to avoid writing back over useful data,

- any operation that is unaligned,
- multiply operations that write back more register rows than necessary for a particular product, known as *oversized writeback*.

From this implementation, mathematical expressions for the number of array cycles and the number of load/store cycles are produced. A second expression for the number for array cycles that ignores overheads attributable to the Load/Store memory architecture is also derived; this expression represents the perfect memory architecture. Both the array cycle count expressions are usually quite complex and cannot be simplified because of the presence of the ceiling function $\lceil\ \rceil$. To produce a more compact expression of the results, some details such as offsets and the ceiling function are ignored. These approximate equations make plain the way the metric depends on the architectural parameters.

The amount of register memory used is calculated by examining the algorithm. For some cases, a partitioned algorithm is derived for problems too large to fit into a given register size. Partitioned algorithms require more load/store transfers, and may require more array cycles.

The algorithm is then simulated on the MATRISC simulator, which provides confirmation of the array cycle and load/store transfer counts already derived as well as confirmation of the correctness of the implementation. The simulator also gives the control processor instruction count, which cannot be calculated by just examining the algorithm.

When implementing the algorithms, every reasonable effort will be made to optimise them fully for the MATRISC architecture. However, developing a fully optimal algorithm can be extremely time consuming and proving that it is optimal, even more so. For the purposes of these simulations, it is sufficient that a serious effort is made towards optimisation, focusing particularly on the characteristics of the architecture.

The simulation methodology described in this section will now be applied to a number of algorithms, beginning with the basic operations of matrix addition and multiplication, and continuing with more complex algorithms for linear systems solution, matrix orthogonalisation and the Fourier transform.

5.2 Addition

Addition is a very simple operation that provides important results used by more

complex algorithms. Note that everything in this section applies to other elementwise operations, and to copying by adding to zero.

With reference to §4.1.2.3, it can been seen that addition can be performed by adding together pairs of p element wide partitions of the matrix, which requires multiple addition operations. Alternatively, if the matrix has full stride or has a depth of one, the whole addition algorithm can be performed using one operation. Either way, the number of array cycles required is equal to the number of register rows that each operand occupies, and is given by the equation,

$$\text{AdditionCycles}(w, d, o, p) = d\left\lceil \frac{w+o}{p} \right\rceil \tag{5.1}$$

where w is the width of the matrices, d is the depth, and o is the offset. For two matrices to be added on the Load/Store architecture, they must have the same width, depth and offset (See Table 4.3). If a perfect memory architecture were used, there would be no offset so the cycle requirement would be,

$$\text{AdditionCyclesP}(w, d, p) = d\left\lceil \frac{w}{p} \right\rceil \tag{5.2}$$

Ignoring the ceiling function gives the continuous approximate expression as,

$$\text{AdditionCyclesA}(w, d, p) = \frac{dw}{p} \tag{5.3}$$

The time required to perform matrix addition is proportional to the number of elements in the matrix, as is the time required to load the matrices into registers. Thus, as the bandwidth available for computation is much greater than the bandwidth for load and store transfers, performing an addition from operands in main memory will always be memory-bound. This can be expressed mathematically by calculating the ratio of compute time to load/store time,

$$\frac{\text{AdditionCyclesA}(w, d, p)}{3\text{LoadStoreArrayCycles}(d, w)} = \frac{\frac{dw}{p}}{3dwF} = \frac{1}{3pF} \tag{5.4}$$

which is 0.09 for the standard parameters, and would be much less than one for any conceivable set of system parameters. Note that the factor of 3 in (5.4), is because two inputs must be loaded and one result stored.

As addition only uses the p processing elements on the leading diagonal of the array, only p useful floating point operations are performed per array cycle. For a given array cycle time, this translates into a rate in Mflop/s that will be referred to as the *addition speed* of the

architecture. The addition speed with the standard parameters is 250Mflop/s. Of course, any particular addition operation will operate at less than the addition speed unless the operands completely fill their register rows.

Operations between a scalar and a matrix require the same number of cycles as addition, so (5.1), (5.2) and (5.3) apply and the processor array runs at addition speed. In the case where an operation is performed between two scalars, the same equations apply as a scalar is simply a 1×1 matrix, but only one useful floating point operation is performed per array cycle. For a given array cycle time, this translates into a rate in Mflop/s which will be referred to as the *scalar speed* of the architecture. The scalar speed with the standard parameters is 50Mflop/s.

5.3 Multiplication

With reference to §4.1.2.1, it can be seen that multiplying two matrices requires that each partition of one matrix be multiplied by each partition of the other. For multiplication with a virtual factor v, each partition is vp elements wide. Thus, if a matrix has width, w, and offset, o, it must be divided into $\lceil (w+o)/vp \rceil$ strips. Recall that for matrix multiplication on the Load/Store architecture, the conformality requirement equates to the fact that the depth of the two matrices being multiplied must be equal (see Table 4.3). The number of cycles required to transfer each pair of partitions into the array is dv, where d is the depth of the matrix. However, v^2p cycles are required to transfer the result out of the array, which means that the number of cycles required to deal with each pair of partitions is the maximum of these two.

The overall expression for the number of array cycles required for multiplying two matrices with widths w_1 and w_2, offsets o_1 and o_2, and depth d, is

$$\text{MultiplicationCycles}(w_1, d, w_2, o_1, o_2, p, v) = \max(d, pv)v\left\lceil \frac{w_1 + o_1}{vp} \right\rceil \left\lceil \frac{w_2 + o_2}{vp} \right\rceil \quad (5.5)$$

No offset would exist in the hypothetical perfect memory architecture. In addition, oversized writeback would not exist, which introduces a dependence on which register the result is written to. Assuming that the result is written back to the register that matrix 1 comes from, the result is

$$\text{MultiplicationCyclesP}(w_1, d, w_2, p, v) = \max(d, \min(pv, w_2))v\left\lceil \frac{w_1}{vp} \right\rceil \left\lceil \frac{w_2}{vp} \right\rceil \quad (5.6)$$

The approximate result is,

$$\text{MultiplicationCyclesA}(w_1, d, w_2, p, v) = \frac{dw_1w_2}{vp^2} \quad (5.7)$$

Multiplication can perform up to $2vp^2$ useful floating point operations per array cycle. Using a particular array cycle time to convert this to Mflop/s gives the *multiplication speed* of the architecture. The multiplication speed with the standard parameters is 2500Mflop/s, 5000Mflop/s and 10000Mflop/s for virtual factors 1, 2, and 4 respectively. To run at the full multiplication speed, the operand matrices must be at least as large as the virtual array size in each dimension. Three important situations where this is not the case deserve special mention; matrix-vector products, outer products and dot products.

Matrix-vector products use only one column or row of the processor array, and so only perform $2p$ flop per array cycle, resulting in a *matrix-vector speed* that is twice the addition speed. The matrix-vector speed with the standard parameters is 500Mflop/s.

Outer products use all the processing elements in the array but writing back of the results forms a bottleneck. Consider a $p \times 1$ by $1 \times p$ outer product on a $p \times p$ array. Only one array cycle is required to transfer the input into the array and the computation requires only one array cycle. However, writing back the $p \times p$ result to registers takes p array cycles. Averaging the p^2 floating point operations that occur over the p array cycles the whole operation requires, gives p floating point operations per array cycle. Thus, outer products operate at addition speed.

Dot products, also called inner products, use only one processing element and so perform only 2 useful flop per array cycle. This determines a *dot product speed* for the architecture that is twice the scalar speed. The dot product speed with the standard parameters is 100Mflop/s.

In some situations, multiplication by the identity matrix is used to copy a matrix from one register to the other. As the identity is largely zeros, this is best achieved by dividing the matrix into array-sized blocks and copying each one by multiplying by an array-sized identity. Copying each block requires p array cycles. The number of cycles required to copy a matrix of width, w, depth, d, and offset, o, is

$$\text{MultiplyCopyCycles}(w, d, o, p) = p\left\lceil \frac{d}{p} \right\rceil \left\lceil \frac{w + o}{p} \right\rceil \quad (5.8)$$

Note that because copying is a bandwidth-limited operation, there is no advantage in using a virtual factor greater than one. With the perfect architecture, there are no offsets so the result becomes

$$\text{MultiplyCopyCyclesP}(w, d, p) = p\left\lceil \frac{d}{p} \right\rceil \left\lceil \frac{w}{p} \right\rceil \tag{5.9}$$

The approximate result is,

$$\text{MultiplyCopyCyclesA}(w, d, p) = \frac{dw}{p} \tag{5.10}$$

By comparing this result to (5.3), it can clearly be seen that copying using multiplication occurs at the same speed as copying using addition, to the level of approximation in the two equations.

For a matrix of size $n \times n$, the time required to load it into the registers from main memory is proportional to its size, n^2. The time taken to perform a matrix multiplication on two such matrices is proportional to n^3. In both cases, the constants of proportionality depend on the particular configuration of the architecture. Thus for a particular configuration there is always a size of matrix, which will be referred to as the *critical size*, above which the time to multiply is greater than the time to load. For matrices as large, or larger, than the critical size, the architecture will be compute-bound, for matrix multiplication from main memory, if enough register space is available.

The balance between compute and load/store times for matrix multiplication will now be examined in more detail for a number of different cases.

5.3.1 Unlimited Register Size

The first case is that of multiplying two $n \times n$ matrices stored in main memory, in a system that has an unlimited amount of register memory. The operation proceeds in three parts; the two matrices are loaded into the registers, multiplied together, and then the result is stored in main memory. In practice, these parts would be overlapped to some degree to reduce the latency of the operation. In the following calculations, the approximate result for the number of array cycles required to perform operations will be used, where necessary, to allow the resulting expression to be simplified into meaningful results.

The time taken to load the operands and store the result, measured in array cycles, is given by,

$$\text{load\_store\_cycles} = \text{LoadStoreArrayCycles}(n, n) = 3Fn^2$$

The number of array cycles required to perform the computation can be found using (5.7), and is given approximately by,

$$\text{compute\_cycles} = \text{MultiplicationCyclesA}(n, n, n, p, v) = \frac{n^3}{vp^2} \tag{5.11}$$

It desirable that the processor be compute-bound so that the greatest possible amount of work is achieved. For this to occur, the computation time must be greater than the time taken to load the data.

$$\begin{aligned} &\text{compute\_cycles} > \text{load\_cycles} \\ &\frac{n^3}{vp^2} > 3Fn^2 \\ &n > 3vp^2F \end{aligned} \tag{5.12}$$

This inequality shows that the critical size is $3vp^2F$, which, assuming $v = V = 4$, evaluates to 225 with the standard parameters.

For the compute-bound situation to occur, the registers must each be large enough to store a matrix of the critical size. Only a small amount of space is required for the result, as it can be stored to main memory immediately. Given this requirement, s, the amount of SRAM memory needed for each register column, can be calculated as follows,

$$\begin{aligned} S &> n^2 \\ sp &> (3vp^2F)^2 \\ s &> 9v^2p^3F^2 \end{aligned} \tag{5.13}$$

For the standard parameters, this result implies that each register column must contain at least 10125 elements. Using 64bit numbers, this equates to 633k bits and so could be achieved with a single 1Mbit SRAM for each register column. Conversely, if a 4Mbit SRAM were used, it would have enough room for a critical size matrix with $p = 9$ but not for $p = 10$. Equation (5.13) places a stringent limit on the size of the array, because of the cubic dependence on the physical array size. A larger array could be achieved by using multiple SRAMs or larger SRAMs, or by reducing F or v, but the fact remains that very large arrays will be impractical for performing matrix multiplication out of main memory with the Load/Store memory architecture.

5.3.2 Limited Register Size

Now assume that the matrices being multiplied are larger than the registers, that is,

$n^2 > S$. In this case only some parts of the two matrices can be in the registers at any one time, so there is a choice about the order in which to load and compute them. The algorithm presented here divides the matrices into partitions for loading and computation. As in all multiplication algorithms so far, the size of partitions is n, the full size of the matrix, in one direction, and vp, the virtual array size, in the other direction.

The limited register algorithm proceeds by initially loading both registers full of partitions. One register becomes the *fixed register*; each piece of data is loaded into it only once. The other register is the *variable register,* which will need to load each partition many times. The computation proceeds by multiplying the partitions in the fixed register by all partitions in the variable register. As soon as any partition in the variable register has been used, that part of the register is reloaded with the next needed partition. When all of the variable partitions have been loaded, a new set of partitions is loaded into the fixed register. At this point, there are two choices for how to proceed.

Method 1, the simpler choice, involves loading the first partitions from the variable matrix again and proceeding though the variable matrix in the same order as before, multiplying all the partitions with the new partitions in the fixed register. This situation is shown in Figure 5.1(a), where the arrows on the result matrix show the order in which it is calculated.

Method 2 takes advantage of the fact that new partitions need not be loaded into the variable register when new partitions are loaded into the fixed register. Instead, the variable partitions already loaded are used, and the evaluation proceeds through the variable register matrix again but in the opposite direction. In this case, the computation snakes up and down the result matrix as shown in Figure 5.1(b).

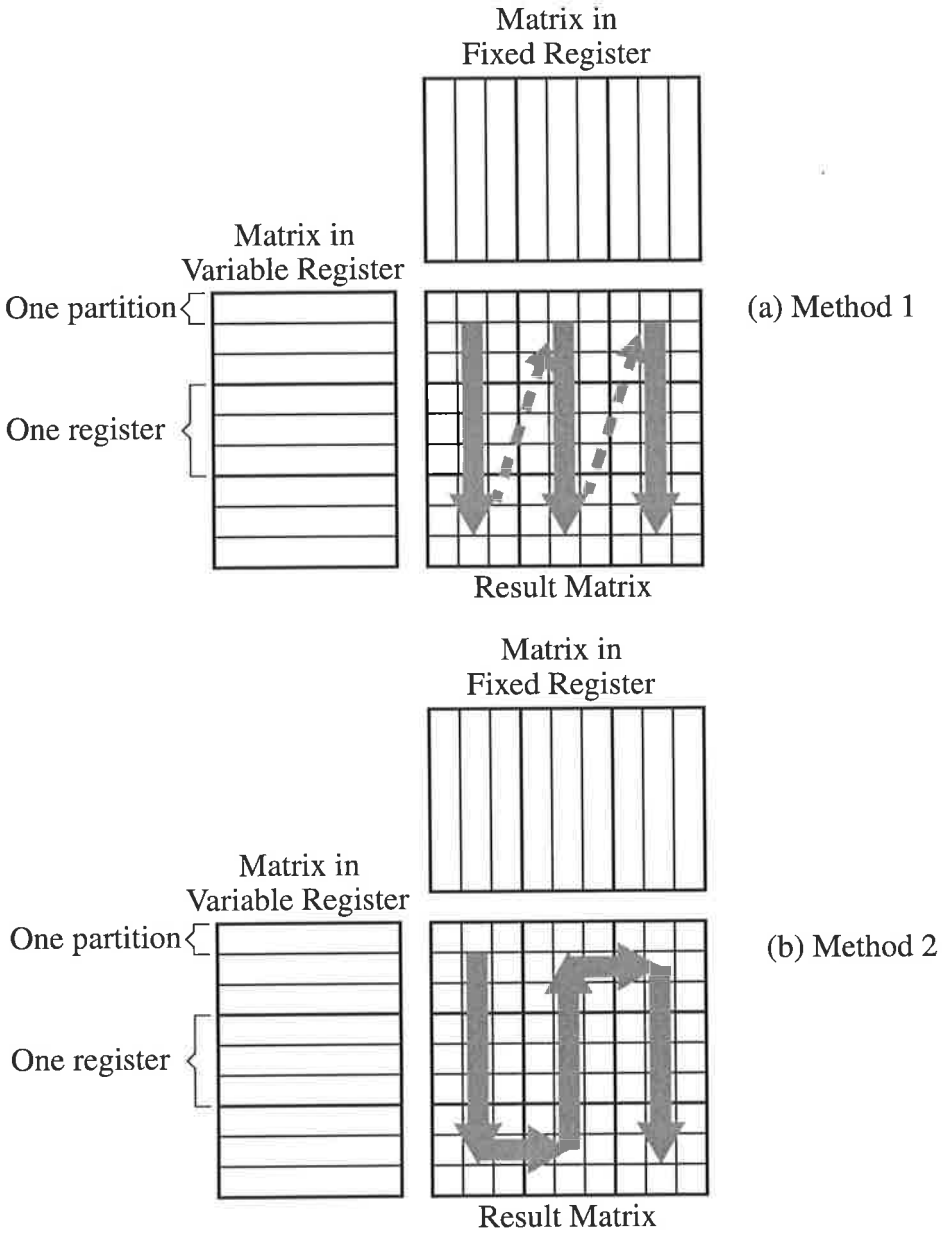


Figure 5.1: Computation Pattern for the Limited Register Algorithm

For Method 1, the number of times the matrix must be loaded into the variable register is computed by dividing the matrix size by the register size. The total load/store time is taken as the time to load the whole matrix that many times plus one for loading of the fixed register, plus another one for storing of the final result.

$$\text{load\_store\_cycles\_method\_1} = \left(\frac{\text{matrix\_size}}{\text{register\_size}} + 2\right) \times \text{matrix\_load\_time}$$

$$= \left(\frac{n^2}{S} + 2\right) n^2 F$$

The required register size, to ensure compute-bound operation, is found by,

compute_time > load_store_cycles_method_1

$$\frac{n^3}{vp^2} > \left(\frac{n^2}{S} + 2\right) n^2 F$$

$$S > \frac{n^2}{n/vp^2F - 2}$$

$$S > \left(\frac{1}{3\bar{n} - 2}\right) n^2 \qquad \text{where} \qquad \bar{n} = \frac{n}{3Vp^2F}$$

The quantity $\bar{n}$ measures the size of the matrices relative to the critical size. Note that if $\bar{n} = 1$, the register size must be as large as the matrix, as would be expected.

For Method 2, the fact that the partitions in the variable register need not be reloaded when the fixed register is reloaded is taken into account. The time taken to load the fixed register is absorbed by the loading of the variable register, apart from the first register full. Thus the load/store time is,

$$\begin{aligned} \text{load\_store\_cycles\_method\_2} &= \left(\frac{\text{matrix\_size}}{\text{register\_size}} + 1\right) \times \text{matrix\_load\_time} \\ &\quad + \text{register\_load\_time} \\ &= \left(\frac{n^2}{S} + 1\right) n^2 F + SF \end{aligned}$$

The required register size is now found as,

compute_time > load_store_time_method_2

$$\frac{n^3}{vp^2} > \left(\frac{n^2}{S} + 1\right) n^2 F + SF$$

$$0 > S^2 + \left(n^2 - \frac{n^3 F}{vp^2 F}\right) S + n^4$$

$$S > \frac{\left(\frac{n^3}{vp^2F} - n^2\right) \pm \sqrt{\left(n^2 - \frac{n^3F}{vp^2F}\right)^2 - 4n^4}}{2}$$

$$S > \left(\frac{3\bar{n} - 1 - \sqrt{9\bar{n}^2 - 6\bar{n} - 3}}{2}\right) n^2$$

This bound on the register size is smaller than in Method 1; at best, it is 25% smaller, occurring for $\bar{n} \approx 1.15$.

5.3.3 Block Multiplication

Until now, only algorithms that use the minimum time to compute the result have been considered. Every time two partitions have been multiplied, they produced a $vp \times vp$ section of the final result. In this section, an algorithm that computes intermediate results and then combines them is described. It is significant as it can operate on unlimitedly large matrices with a finite amount of register space, with the only penalty being a small compute overhead.

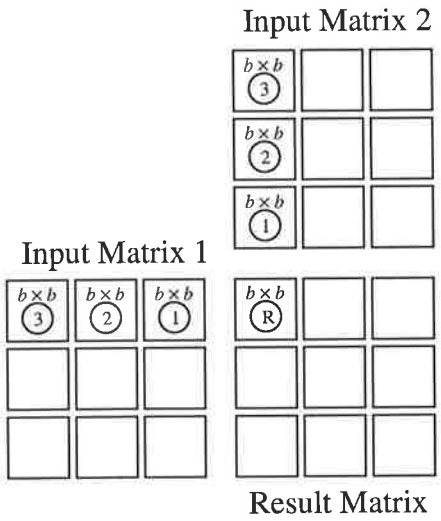


Figure 5.2: Block Multiplication

The method uses the general block multiplication of (1.1) and (1.2) with $n \times n$ matrices partitioned into $b \times b$ sized blocks as shown in Figure 5.2. To compute the result block R, the pairs of blocks labelled 1, 2 and 3 are multiplied and the results added together. By using (5.7) to calculate the number of array cycles required to form the product between each pair of blocks, and multiplying by the number of blocks, the total number of cycles required to perform the multiplication part of calculating R is given by,

$$\begin{aligned} \text{multiply\_cycles} &= \text{number\_of\_blocks} \times \text{cycles\_to\_multiply\_block} \\ &= \frac{n}{b} \times \frac{b^3}{vp^2} \\ &= \frac{nb^2}{vp^2} \end{aligned}$$

This result represents the same amount of time it would take to produce a $b \times b$ block of the result using the other methods discussed so far. The number of array cycles required to add the partial results together equals the number of blocks minus one, multiplied by the

number of cycles required to add two blocks, which can be found using (5.3).

$$\begin{aligned} \text{add\_cycles} &= (\text{number\_of\_blocks} - 1) \times \text{time\_to\_add\_block} \\ &= \left(\frac{n}{b} - 1\right) \times \frac{b^2}{p} \\ &= \frac{nb}{p} - \frac{b^2}{p} \end{aligned}$$

The overhead that addition incurs over the minimum computation time is,

$$\text{overhead} = \frac{\text{add\_cycles}}{\text{multiply\_cycles}} = \frac{\frac{nb}{p} - \frac{b^2}{p}}{\frac{nb^2}{vp^2}} = \frac{vp}{b} - \frac{vp}{n}$$

As the matrix size increase, this overhead increases to a maximum of vp/b.

A simple choice for b is the critical size ($3vp^2F$). This choice ensures overall compute-boundedness, because each multiplication will be compute-bound and the addition adds only more compute time as it is performed on data in the registers. In this case, the maximum overhead will be approximately 8.9% for the standard parameters.

The register size requirements for this algorithm are that each register should be able to hold two $b \times b$ matrices. The calculation proceeds by loading the two $b \times b$ input matrix blocks and multiplying them. This intermediate result is stored in the X register (for example) and then accumulated with the previous result stored in the Y register. Note that the size requirement is fixed no matter how large the operand matrices are. If a 4Mbit SRAM were used for each register column, two critical sized matrices could be held in each register for an array up to 7×7, assuming a virtual factor of 4.

The fixed register size requirement and a small computational overhead makes this method appealing for very large matrices. The overhead of the addition could be reduced by using larger blocks, which would be possible in the case of the standard parameters.

5.3.4 Non-Square Products

So far, it has been assumed that two square matrices a are being multiplied to produce a square result. This assumption simplifies the analysis but neglects a number of important cases. In general, a matrix product can be formed between an $n \times l$ matrix and an $l \times m$ matrix, producing an $n \times m$ result. The possible cases when $n = m = l$ is not true are shown below. Note that in the non-square case, it has been assumed that $n \leq m$ without loss of

generality.

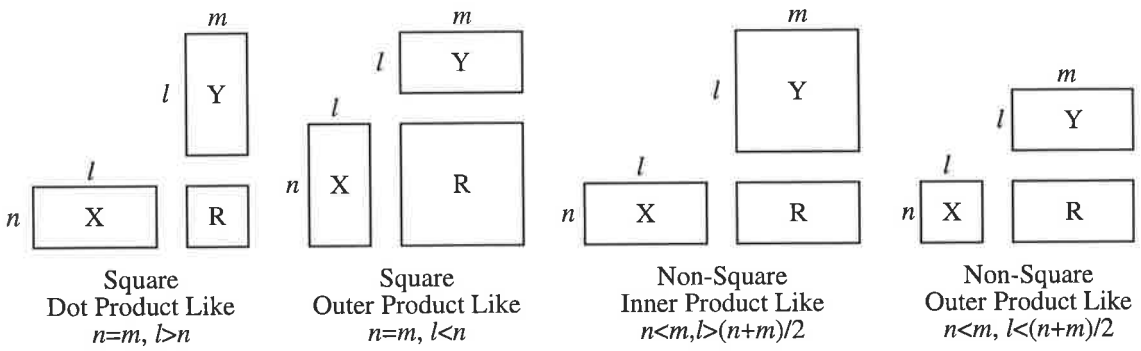


Figure 5.3: Classification of Non-Square Matrix Multiplication

This classification provides a terminology for discussing differently sized products, but it should be used with caution as it results from a somewhat arbitrary division of a continuous range of possibilities.

5.3.4.1 Unlimited Register Size

Generalising the condition for compute-boundedness from the unlimited register result, (5.12), for the non-square case gives,

$$\text{compute\_time} > \text{load\_time}$$

$$\frac{nml}{vp^2} > (nl + lm + nm)F$$

$$n > Vp^2F\left(1 + \frac{n}{l} + \frac{n}{m}\right)$$

$$m > Vp^2F\left(1 + \frac{m}{n} + \frac{m}{l}\right)$$

$$l > Vp^2F\left(1 + \frac{l}{m} + \frac{l}{n}\right)$$

This result can be interpreted as follows. Suppose n is the largest of n, m and l. Therefore n/l and n/m must be greater than one, and hence n must be greater than the critical size of $3Vp^2F$. As the three conditions are symmetrical, this reasoning will apply regardless of which dimension is the largest and hence at least one of n, m, or l must be greater than the critical size.

The smallest value for any of the dimensions is one third of the critical size, that is, Vp^2F. This value would only occur in the limiting case; for example, n could approach Vp^2F if both m and l were very large.

5.3.4.2 Limited Register Size

The case with limited register size is a little more complicated. As it is assumed that $n \leq m$, the smaller matrix ($n \times l$) is the one which must be stored in the variable register to minimise load time. For loading method 1, from §5.3.2, the required register size is,

$$\text{compute\_time} > \text{load\_time\_method\_1}$$

$$\frac{nml}{vp^2} > \frac{nl}{S} \times nlF + lmF + mnF$$

$$S > \frac{n^2 l^2}{\frac{nml}{Fvp^2} - ml - mn}$$

$$S > \frac{nl}{3\bar{m} - \frac{m}{n} - \frac{m}{l}} \qquad \text{where} \qquad \bar{m} = \frac{m}{3Vp^2F}$$

For loading method 2, the required register size is,

$$\text{compute\_time} > \text{load\_time\_method\_2}$$

$$\frac{nml}{vp^2} > \frac{nl}{S} \times nlF + SF + nmF$$

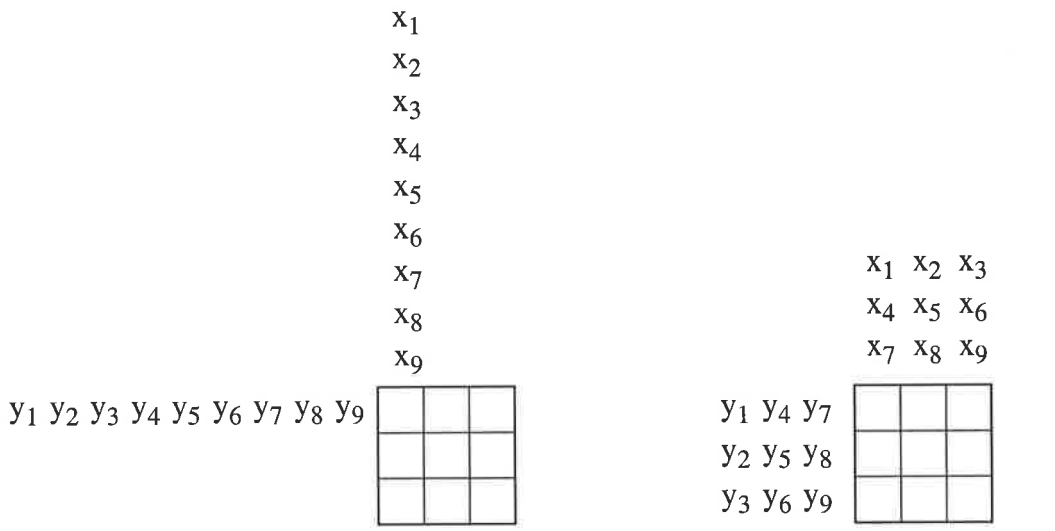
$$S > \left(3\bar{m} - \frac{m}{l} - \sqrt{9\bar{m}^2 - 6\frac{m}{l}\bar{m} + \frac{m^2}{l^2} - 4} \right) \frac{nl}{2}$$

Both of these results are encouraging in that the register size required is determined primarily by the size of the smaller matrix. In addition, as the larger matrix grows, the register size requirement decreases.

Note that the block matrix multiplication of §5.3.3 can also be applied to non-square products.

5.3.5 Broadside Dot Product

The speed of dot products on the MATRISC architecture is very poor when they are performed as matrix products. However, dot products can be calculated in a different fashion that increases the speed of calculation by almost a factor of p for sufficiently long vectors. The key to this approach is to feed the two vectors into the array in transpose form, hence the name broadside dot product. The method is demonstrated in Figure 5.4, for two 9 element vectors on a 3×3 array.



(a) Normal Dot Product (b) Broadside Dot Product

Figure 5.4: Comparison of Normal and Broadside Dot Product Computation

Each of the processing elements performs a multiply-accumulate as usual for a matrix product, but the writing back of the results occurs from the leading diagonal elements, as for addition. This procedure results in p partial results that must be summed together to produce the final dot product. This addition is accomplished by transposing the vector of partial results to the other register, and then doing a normal dot product with a vector containing ones. If either the start or the end of the input vectors do not fill an entire register row, the incomplete rows must be multiplied and summed separately.

Theoretical analysis of this operation is complex, and does not lead to an insightful description of its performance. Instead this method was implemented as a Masm template, and a number of simulations carried out using the standard parameters to test the speed of the operation. As the broadside dot product uses p processing elements, each performing 2 floating point operations per array cycle, for the bulk of the computation when operating on long vectors, the peak possible speed is the matrix-vector speed, which is 500Mflop/s for the standard parameters. The results for matrices with an offset of 0 and 4 are shown in Figure 5.5. The results for offsets 1, 2, 3 are approximately linearly distributed between the results for offsets 0 and 4.

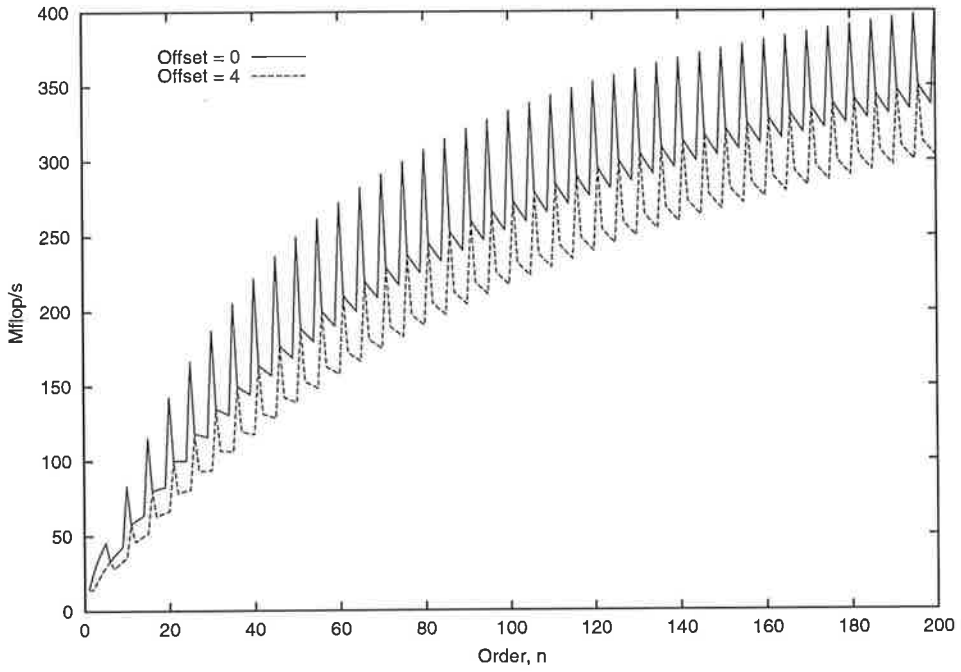


Figure 5.5: Performance of Broadside Dot Product

The graph above shows that the broadside method is faster than the dot product speed of 100Mflop/s for all vector lengths above 33. Vectors that precisely fill register rows run much more quickly than vectors that partially fill register rows. For vectors of length 200, 60% to 80% of the matrix-vector speed is achieved, depending on the offset, giving a speed up of 3 to 4 times over the normal dot product.

Broadside dot product could potentially accelerate any algorithm that heavily uses the dot product operation. However, there are two interrelated drawbacks: firstly, the speed is lower than ordinary dot product for short vector lengths; and secondly, the vectors are required in the transpose of the storage form required for normal dot products. In some situations, this requirement may be advantageous and result in avoiding superfluous copying, but often it will require extra copy operations. In particular, the storage form requirement means that short vector lengths cannot be handled by the normal dot product method to improve performance.

A significant improvement to the speed of the broadside dot-product could be made by improving the method used to add the partial results. It is possible to sum p numbers in $\lceil \log_2 p \rceil$ steps. This could be achieved on the MATRISC processor array, without adding extra interconnection paths using the method shown in Figure 5.6.

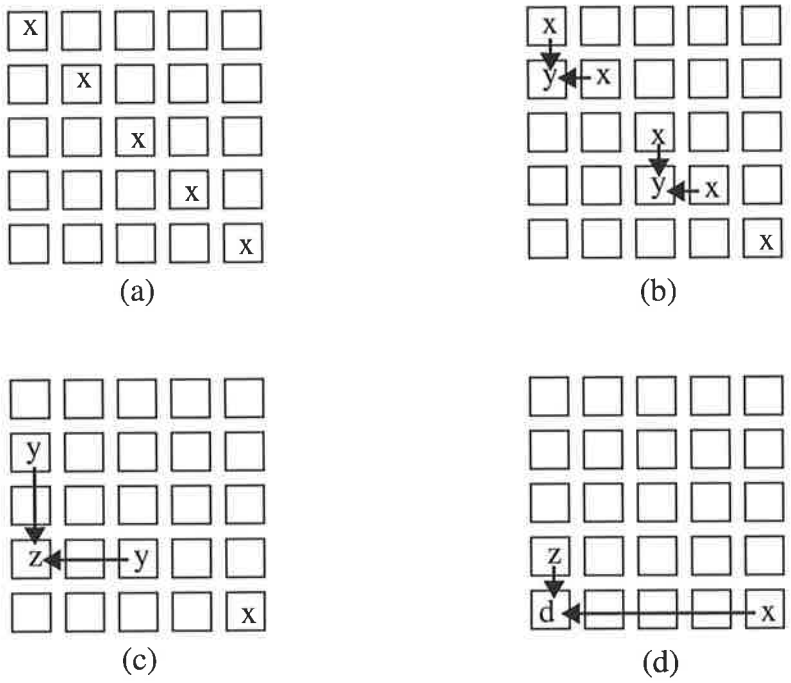


Figure 5.6: Partial Result Addition Using Binary Tree Method

Figure 5.6(a) shows the partial results in the leading diagonal elements of the processor array. The rest of the figure shows the partial results being added using a binary tree pattern. Note how it is only necessary to transfer data along the existing buses within the array; no new paths are needed. This would entail altering the buses to allow such a transfer, but this would not be difficult because it would not require more bandwidth just additional control mechanisms to allow the processing element to read from the write bus at the correct time.

5.3.6 VHDL Simulation Results

To assess the basic design of the Load/Store architecture, a number of basic operations were simulated in the VHDL model described in §4.4. However, the detail of the simulation and the insufficient power of the available machines meant that it was impossible to simulate the architecture with the standard parameters with critical sized matrices, as the run times would have been impractical (the simulations performed took several hours per run). In order to test the theoretical results, a much smaller architecture was used with a 3×3 array of processors and a maximum virtual factor of 2. For such a system, the critical size is 40.5, which can be simulated in a reasonable length of time. The tests involved first calculating load, store and compute times with relatively large registers, followed by implementation of a multiplication algorithm that operates with limited register space.

The first tests performed simply measured the speed of a single load, store, addition and multiplication with a large amount of register space. The principal reason was to gauge whether the theoretical calculations agreed with the more detailed VHDL simulations, which take into account more complex phenomena such as contention for access to the registers between load/store transfers and computation.

5.3.6.1 Load

The first test simulated the loading of a square matrix, of sizes ranging from 1×1 to 50×50. When a matrix is loaded, it has a mapping applied to it that can affect the rate at which it is loaded. However, the design of the inverted address generator is intended to make loading operate at a constant speed, which will be close to the maximum for the main memory, for all mappings. These results test how effectively the inverted address generator achieves this goal.

Three different mappings were used in simulations: normal (in-order), transpose and prime factor. As it is possible to apply a prime factor mapping only to a matrix whose dimensions are relatively prime, the prime factor test were performed with matrix sizes 2×1 up to 50×49. The results for these cases are shown in Figure 5.7, along with the theoretical calculation based on loading one matrix element every load/store cycle time (15ns). The importance of this graph is that it shows that the theoretical results correspond well to the behaviour observed on the MATRISC simulator.

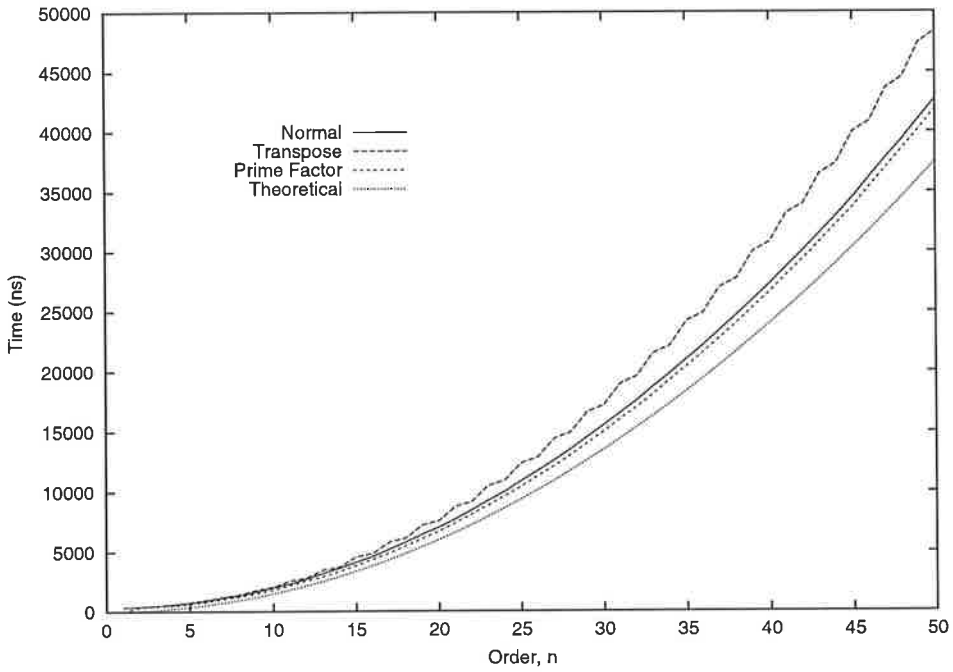


Figure 5.7: VHDL Simulation of Matrix Loading

The results show that the normal and prime factor mappings took 10% to 15% longer than the theory predicted. This discrepancy is quite reasonable because the theoretical model assumes that the whole RAMBUS bandwidth will be used for data transfer, which is never the case in practice due to control overheads. Note that prime factor mapping appeared slightly faster than normal mapping only because it was forced to use a slightly smaller matrix.

The transpose mapped case was approximately 30% slower than predicted. The fundamental reason for this is that during a transpose mapping, consecutive matrix elements are usually being loaded into the same register column. Conflicts for access to this register column mean that the load operation must stall occasionally. As will be seen in §5.3.6.4, transpose mapping should be avoided for other reasons.

5.3.6.2 Store

The same range of tests was also performed for the store operation. The results are shown in Figure 5.8.

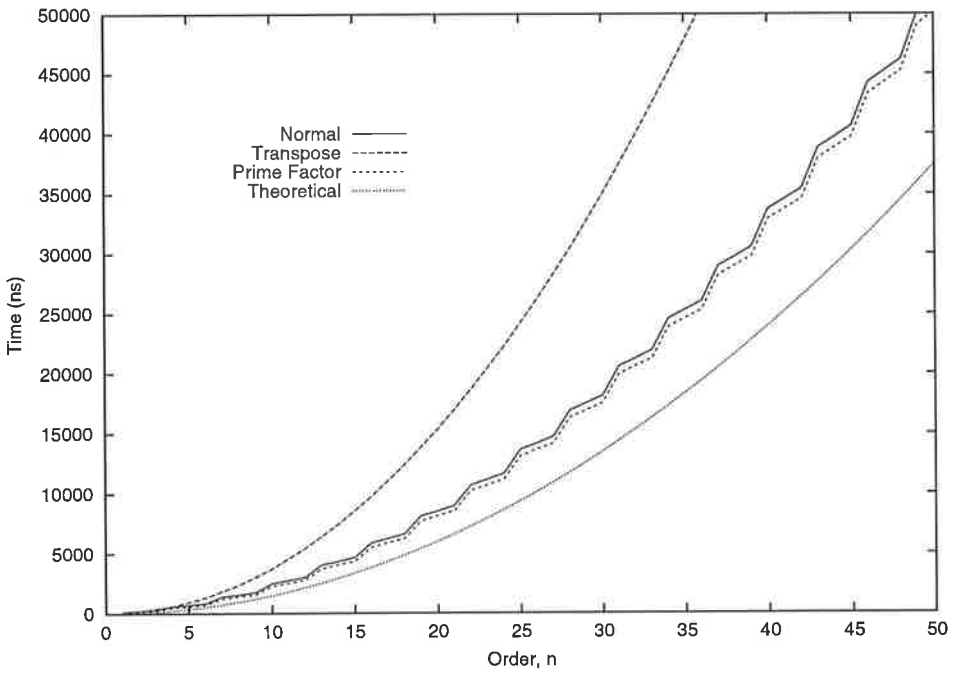


Figure 5.8: VHDL Simulation of Matrix Storing

The major difference between these and the load results was the drastic slowing down of the transpose mapping. Again this was due to conflicts when repeatedly accessing a single register column, but was more dramatic as these conflicts affect stores more than loads.

5.3.6.3 Matrix Addition

The speed of matrix addition was tested for the same range of matrix sizes. Results are shown in Figure 5.9. The graph shows a comparison of the simulated results with the theoretical results, which were calculated in array cycles by using (5.1), and converted to a time by multiplying by the array cycle time of 20ns.

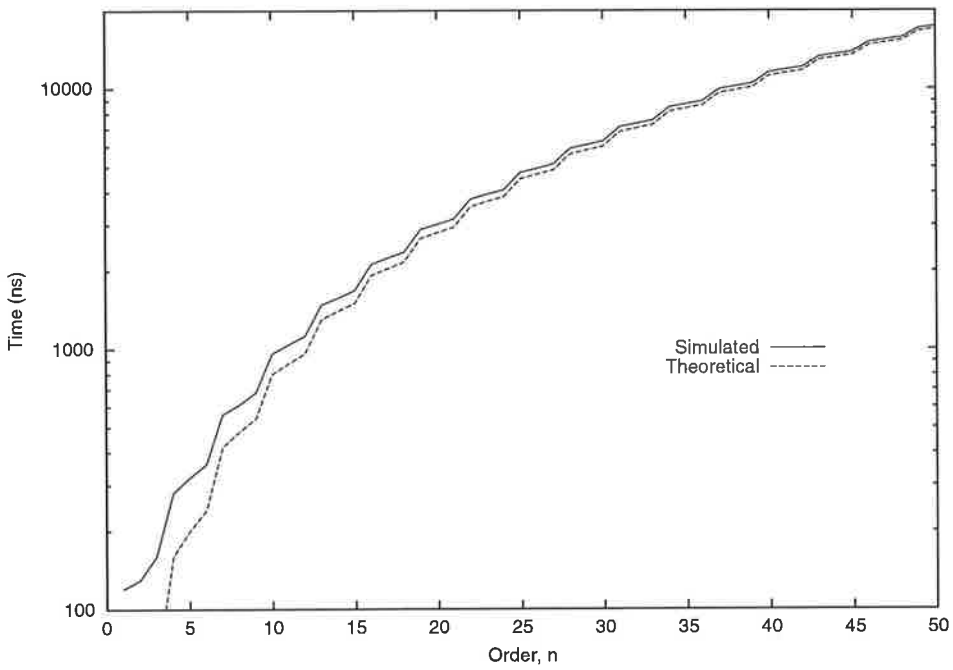


Figure 5.9: VHDL Simulation of Matrix Addition

The results show that the theoretical calculations agrees well with the more detailed VHDL simulation, except for small matrix orders, where the overheads of starting and finishing the simulation run dominate.

5.3.6.4 Matrix Multiplication and Critical Size

The speed of matrix multiplication was tested for the same range of matrix sizes and the results are shown in Figure 5.10. The graph shows a comparison of the simulated results with the theoretical results, which were calculated in array cycles by using (5.5) and converted to a time by multiplying by the array cycle time of 20ns. The graph also shows the load/store times that would be required if the multiplication were performed out of main memory. These figures were calculated as twice the measured load time plus the measured store time, both times assuming a normal address mapping.

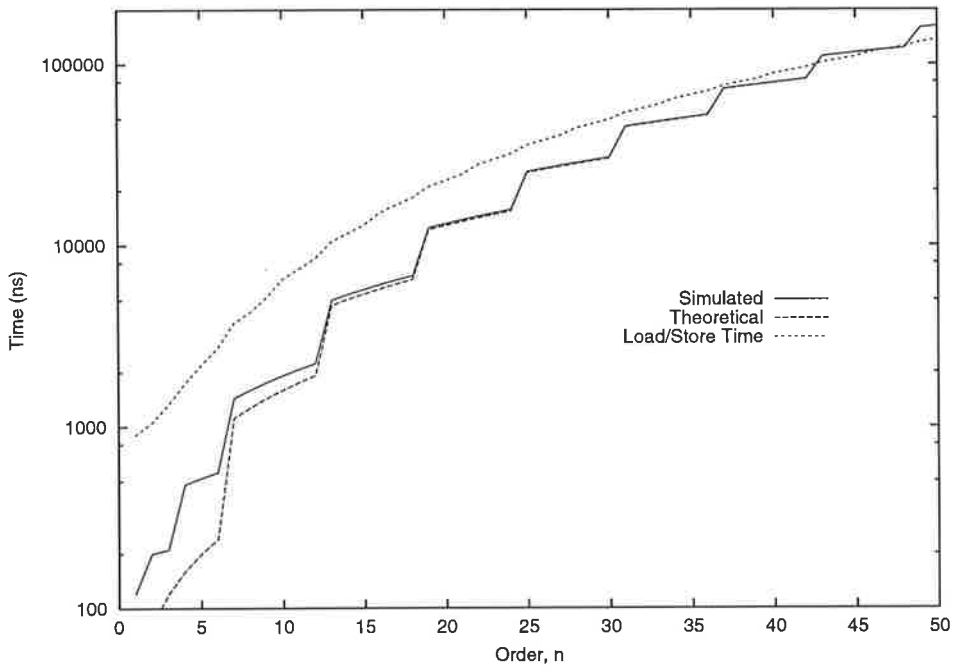


Figure 5.10: VHDL Simulation of Matrix Multiply Times

The simulated performance matched the theoretical calculation very well, except for small matrix sizes, where the overheads of starting and finishing a series of calculations dominate.

The critical size for this architecture is 40.5, which should correspond on the graph to the matrix order for which the matrix multiply and load/store times are equal. In fact, this occurs at an order of 43. A higher value than predicted by theory was to be expected, because the simulation showed that while the theory predicts matrix multiplication time very well, it under estimates the time required for data transfer.

One aspect of the Load/Store architecture that has been ignored until now is that, when a load or store operation accesses the registers, it must interrupt any computation being performed at that time. Further simulations were performed with a load operation occurring at the same time as the multiplication. The increase in the amount of time required for computation is plotted in Figure 5.11. Note that if a prime factor mapping were applied during the load, the results were identical to the normal mapping so this case is not shown.

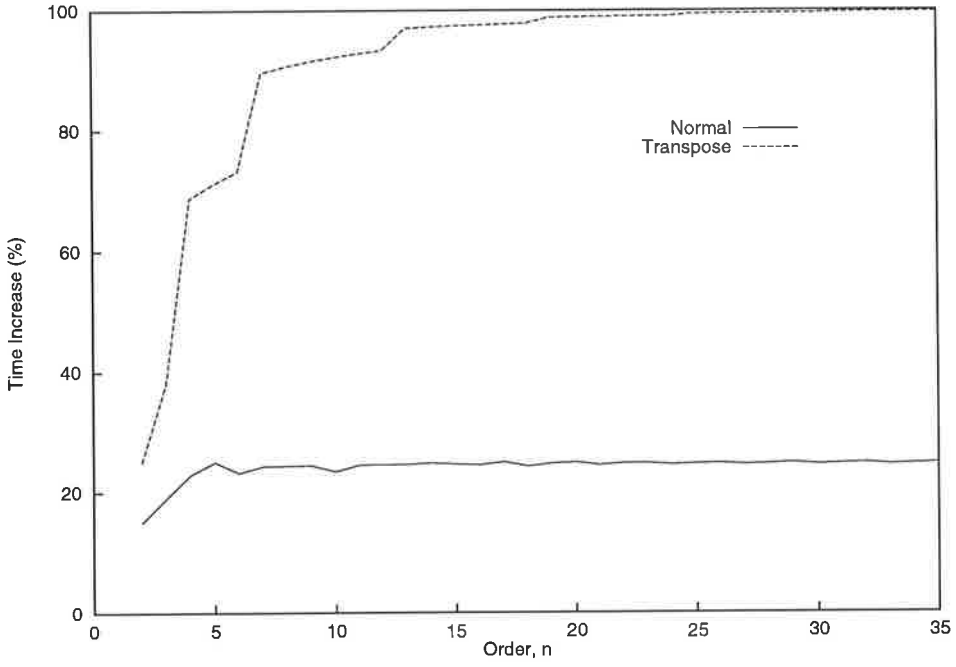


Figure 5.11: Increase in Matrix Multiply Time due to Concurrent Load Operation

The maximum possible increase in time required was 100%, which corresponded to the load operation stealing the maximum amount of register accesses allowed. This maximum occurred when a transpose load was being performed for larger matrices; this result was expected and would occur regardless of the size of the array. The root cause of the poor performance of transpose loading was that the registers are divided into partitions, which will always slow down the transpose case. It is also a reminder that the fundamental limitation on the architecture is accesses to the registers.

The increase for a normal load was much smaller, about 25%. Calculating the expected value theoretically in the normal case was not possible, but it would be inversely proportional to the physical array size; thus the 25% penalty can be expected to be lower for large arrays. This test indicated that reorganising computations so that matrices are loaded in normal order will result in faster operation. Such organisation may require loading into one register and then copying the transpose into the other register.

5.3.6.5 Multiplication from Main Memory

Tests of operations on matrices larger than the critical size are problematic in that they are complex and time consuming. For these reasons, only a single simulation of multiplying

two 72×72 matrices out of main memory was performed. The operation used method 1 loading and, without writing the result back to main memory, took 562370ns. This time included the latency of initially loading the caches full of data, which required 69120ns, leaving 493250ns. The theory, which has been shown to agree with the measured value almost exactly for multiples of the virtual size of the array, implied that the compute time should be 414720ns. Thus the actual time taken was 18% longer than expected. This effect was due to the fact that a load was often being performed while a computation was taking place and because the current compiler does not produce code that maximises overlap of load/store and compute instructions. Better overlap is possible, but will require a better compiler and probably a modification to the architecture to allow out-of-order issue of instructions.

5.3.7 Conclusions

Matrix multiplication is the most fundamental task that can be performed on the matrix processor. It has been shown that a single matrix multiplication performed out of main memory will only be compute-bound if its order is larger than a certain critical size, $3Vp^2F$. For the standard architectural parameters, this size is quite large at 225×225.

Reducing critical size is desirable for two reasons. The first is that many problems are not of that high an order. The second is that a large critical size implies that a large amount of register memory will be needed, which is both expensive and limited by technological restrictions.

The total computational speed of the architecture is $2Vp^2B_{DC}$ flop/s ($F = B_{DC}/B_{LS}$). The only way to reduce the critical size without also reducing the computational speed is to increase the load/store bandwidth, which would be possible by using more than one RAMBUS channel or by switching to a faster main memory bus.

For matrices larger than the critical size, the fastest possible multiplication algorithms require an amount of register memory that grows linearly with the matrix order. Alternatively, a slightly slower block algorithm can be used that requires a fixed amount of memory for matrices of any size.

5.4 Gaussian Elimination

Despite its relative simplicity, Gaussian elimination is one of the most important and fundamental matrix algorithms. This section describes a number of different algorithms for Gaussian elimination in order of increasing speed and complexity, followed by an investigation of various overheads of the Load/Store architecture for these algorithms.

Here, Gaussian elimination is examined from the standpoint of its most common and straight-forward application, that of solving linear systems. Consider the following system of linear equations.

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 \end{aligned}$$

These equations can be expressed in matrix terms as follows.

$$\begin{aligned} Ax &= b \\ \text{where} \quad A &= [a_{ij}] \\ b &= [b_i] \\ x &= [x_i] \end{aligned}$$

Such a system is often expressed as an augmented matrix, consisting of the side by side concatenation of A and b.

$$[A|\ b] = \left[\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{array}\right]$$

The situation often arises that a linear system must be solved for a number of different right hand sides. In this case, an augmented matrix with a number of 'b' columns can be used.

All the methods of solution used here operate by applying transformations to the augmented matrix that do not alter the solution to the set of equations it represents, but which eventually result in a simpler matrix where the solution can more easily be determined.

From the above equations, it is clear that if the A part of the augmented matrix were the

identity matrix, the solution to the system would be $x_1 = b_1$, $x_2 = b_2$, $x_3 = b_3$. Performing transformations until A is the identity matrix results in an algorithm called Gauss-Jordan reduction.

An alternative method is to reduce the augmented matrix to unit upper triangular form, which leads to a matrix with the form shown below. The tildes indicate that the elements are a transformed version of the original element at that position.

$$\left[\begin{array}{ccc|c} 1 & \tilde{a}_{12} & \tilde{a}_{13} & \tilde{b}_1 \\ 0 & 1 & \tilde{a}_{23} & \tilde{b}_2 \\ 0 & 0 & 1 & \tilde{b}_3 \end{array}\right]$$

The solution can be found using backsubstitution. The solution for x_3 can be read off as $x_3 = \tilde{b}_3$. By substituting this value into the second equation, the solution for x_2 can be determined from $x_2 = \tilde{b}_2 - \tilde{a}_{23}x_3$. Finally, by substituting both known values into the first equation, x_1 can be determined from $x_1 = \tilde{b}_1 - \tilde{a}_{12}x_2 - \tilde{a}_{13}x_3$.

There are a number of methods for performing both the reduction and the substitution step. Here, a number of different approaches will be described, starting with the simplest and progressing towards faster, more complicated versions, which are better optimised for the architecture. Initially, the load/store and control processor overheads and the finite size of registers will be largely ignored, and instead the focus will be on developing a fast algorithm in terms of array cycles. Finite registers will be considered in §5.4.8.1. The question of accuracy of the solution will be deferred until the discussion of partial pivoting in §5.4.6. Situations in which the Load/Store memory architecture is slower than the hypothetical perfect memory architecture will be noted as they occur, but quantitative simulation results will only be presented in §5.4.7.

5.4.1 Gauss-Jordan Elimination

There are a number of transformations that can be applied to the matrix of a linear system without changing the solution. Here, one of the simplest set of transformations, called row operations, will be used. In order to solve a linear system, it is sufficient to use just three basic row operations; swapping two rows, multiplying a row by a non-zero scalar, and subtracting one row from another. In practice, a combination of the last two, subtracting a scalar multiple of one row from another, is often used.

Gauss-Jordan elimination proceeds by moving along the leading diagonal elements of the matrix, and at each step using row operations to zero the elements above and below the diagonal. The algorithm for an $n \times m$ augmented matrix A, expressed in Matlab code, is

```
for i = 1 : n
        A(i, i:) = A(i, i:)/A(i,i)
        for j = 1 : n
                if j ~= i
                        A(j, i:) = A(j, i:) - A(j, i) .* A(i, i:)
                end
        end
end
```

This algorithm requires

$$(2m-n)n^2-(m+n/2)n+n/2 \quad \text{flop} \tag{5.14}$$

A derivation of this result is included in Appendix C.

5.4.1.1 Implementation

Implementing Gauss-Jordan elimination on the Load/Store MATRISC architecture is straightforward. The matrix register variables used for the row operation version are shown in Figure 5.12. A large square variable A, in Y normal form, is used to hold the array as it is transformed. The Y normal form is chosen so that individual rows can be operated on easily. Two variables, B and C, in X transpose form are used to hold a single row during intermediate calculations. X transpose form is chosen to allow addition-type operations with the rows of A. Given a specific amount of register memory, these variables would be made as large as possible, placing a restriction on the largest possible problem size. This restriction is not important given the current assumption that the problem fits into register memory.

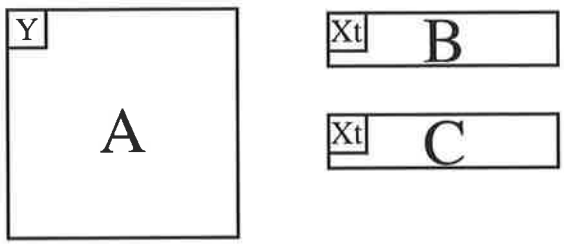


Figure 5.12: Matrix Register Variable Assignment for Row Operation Gaussian Elimination

The zeroing of each column is performed in two steps. The first step, row normalisation, requires three array operations. The row is copied from the A register variable to the B

register variable, divided by the pivot element, and the result stored back in A. Finally the row is copied into B for later use. In the perfect memory architecture, only the actual division operation would be required, as the two copy operations are only necessary to get the data into the correct registers.

The second step is zeroing the elements above and below the pivot, which requires two operations per element being zeroed. The first operation multiplies the copy of the normalised row in B by the element to be zeroed, and stores the result in C. This result is then subtracted from the row in A and the result written back to A.

In terms of the matrix register variables, the algorithm expressed in MATRISC Compliant Matlab code is

```
for i = 1 : n
        B = A(i, i:)
        A(i, i:) = B./A(i,i)
        B = A(i,i:)
        for j = 1 : n
                if j~= i
                        C := B .* A(j,i)
                        A(j, i:) = A(j,i:) - C
                end
        end
end
```

5.4.1.2 Theoretical Results

Examining the algorithm above, the required number of array cycles can be expressed as,

$$\text{GaussJordanRowCycles}(n, m, p) = \sum_{i=0}^{n-1} (3 + 2(n-1))\text{AC}(m-i, 1, i \bmod p, p) \quad (5.15)$$

In this and all further equations for array cycle counts, the abbreviations shown in Table 5.2 are used for the cycle counts of the basic operations. The abbreviation represents, either the full expression, the perfect architecture expression or the approximate expression, as appropriate for the expression it appears in.

Full Function Name	Abbreviation	Full Equation	Perfect Arch. Equation	Approximate Equation
AdditionCycles	AC	(5.1)	(5.2)	(5.3)
MultiplicationCycles	MC	(5.5)	(5.6)	(5.7)
MultiplyCopyCycles	MCC	(5.8)	(5.9)	(5.10)

Table 5.2: Abbreviations of Basic Operation Function Names

The names of all other array cycle count functions will be similarly abbreviated by taking the capital letters, for example GaussJordanRowCycles becomes GJRC.

By substituting the exact expression for the number of addition cycles, given in (5.1), into equation (5.15), the number of cycles required by Gauss-Jordan elimination can be expressed as follows.

$$\begin{aligned}\text{GaussJordanRowCycles}(n, m, p) &= (2n+1)\left\lceil n\left\lceil \frac{m}{p} \right\rceil - \right. \\ &\left. \frac{1}{p}\left\lfloor \frac{n(n-1)}{2} - \left\lfloor \frac{n-1}{p} \right\rfloor \frac{p(p-1)}{2} - \frac{((n-1) \bmod p)((n-1) \bmod p + 1)}{2} \right\rfloor \right\rceil\end{aligned} \tag{5.16}$$

The details of the derivation are straightforward and can be found in Appendix C. This expression sheds little light on the performance of the algorithm because of the complexity of the various floor, ceiling and modulus operations. A more usable expression results from substituting the approximate expression for the number of array cycles required by addition given in (5.3), into equation (5.15), giving the following result. Again, the full derivation can be found in Appendix C.

$$\text{GaussJordanRowCyclesA}(n, m, p) = \frac{(2m-n)n^2}{p} + \frac{(m+n/2)n}{p} + \frac{n/2}{p} \tag{5.17}$$

Comparison of this expression to the number of floating point operations required by the algorithm, given in (5.14), shows that the cubic terms are consistent with p floating point operations being performed per array cycle. Thus, the algorithm will effectively perform at the addition speed of the architecture if the approximate form is valid, which it will be for sufficiently large matrices.

It is important to realise that the number of cycles required by Gauss-Jordan Elimination for the perfect memory architecture cannot be obtained from (5.15), by substituting the expression for the number of array cycles required by addition with the perfect memory architecture. The reason for this is that the implementation expressed in MATRISC compliant code shown above, and hence (5.15) which is derived from it, includes operations that would not be necessary with the perfect memory architecture. Only the division operation is required to normalise the row in the perfect case, resulting in the following expression for the perfect architecture:

$$\text{GaussJordanRowCyclesP}(n, m, p) = \sum_{i=0}^{n-1} (1 + 2(n-1))\text{AC}(m-i, 1, p)$$

Simulation results for Gauss-Jordan elimination are given in §5.4.2.3 along with the results for Gaussian Elimination with backsubstitution.

5.4.2 Gaussian Elimination with Backsubstitution

A more efficient algorithm than Gauss-Jordan Elimination for solving a linear system is to use row operations to reduce the matrix to unit upper triangular form and then to find the solution using backsubstitution.

The reduction of the matrix to triangular form uses row operations as did the previous algorithm, but only the elements below the diagonal are zeroed. The backsubstitution phase of the algorithm works backwards through the array and, at each step, the unknown that has been determined is substituted into all preceding equations.

The resultant algorithm shown below is the classic statement of Gaussian elimination.

```
for i = 1 : n
        A(i, i:) = A(i, i:)/A(i,i)
        for j = i+1 : n
                A(j, i:) = A(j, i:) - A(j, i) .* A(i, i:)
        end
end

for i = n : -1: 1
        for j = i - 1 : -1: 1
                A(j, n:) = A(j, n:) - A(j, i) .* A(i, n:)
        end
end
```

This algorithm requires

$$(2m - 4n/3)n^2 - (m - n/2)n - n/6 \quad \text{flop} \tag{5.18}$$

which is about two thirds of the operations required for the Gauss-Jordan algorithm when only one right hand side is involved.

All calculations of floating point performance reported in this chapter for Gaussian Elimination assume that $(2m - 4n/3)n^2 - (m - n/2)n - n/6$ floating point operations are used regardless of whether or not a particular approach actually uses more operations. For systems with one right hand side, this value corresponds to the $2n^3/3 + 2n^2$ flops assumed when specifying performance for the LINPACK benchmark[Dongarra 98]. The exact equation to calculate the floating point rate is,

$$\text{floating point rate} = \frac{(2m - 4n/3)n^2 - (m - n/2)n - n/6}{C_A T_A}$$

where C_A is the number of array cycles required and T_A is the array cycle time, which is

20ns for all the simulations.

5.4.2.1 Implementation

The matrix register variables required for Gaussian Elimination with backsubstitution are the same as those for Gauss-Jordan Elimination. The algorithm expressed in terms of the matrix register variables is,

```
for i = 1 : n
        B = A(i, i:)
        A(i, i:) = B ./ A(i,i)
        B = A(i, i:)
        for j = i+1 : n
                C = A(j, i) .* B
                A(j, i:) = A(j, i:) - C
        end
end

for i = n : -1: 1
        for j = i - 1 : -1: 1
                C = A(j, i) .* A(i, n:)
                A(j, n:) = A(j, n:) - C
        end
end
```

For a linear system with only one right hand side, that is, one where A has $n + 1$ columns, the backsubstitution part of this algorithm performs only scalar operations and as such, it is an extremely poor choice for the MATRISC architecture.

5.4.2.2 Theoretical Results

Examining the algorithm above, the number of array cycles needed to perform this algorithm can be expressed as,

$$\begin{aligned}\text{GaussRowCycles}(n, m, p) = &\sum_{i=0}^{n-1} (3 + 2(n-i-1))\text{AC}(m-i, 1, i \bmod p, p) \\ &+ \sum_{i=0}^{n-2} (1 + 2(n-i-1))\text{AC}(m-n, 1, n \bmod p, p)\end{aligned} \tag{5.19}$$

Even the slight increase in complexity of the backsubstitution algorithm over the Gauss-Jordan case means that, when the exact expression for AC() is substituted into (5.19), the resulting exact numerical expression for the number of array cycles required is unusably complex. This result is omitted here and will not be attempted for the more complex algorithms that follow. Equation (5.19) can still be evaluated directly to calculate the exact number of array cycles required and was used to generate some of the simulation graphs, as

it produces exactly the same array cycle counts as the full simulation in Masm, but much more quickly.

An approximate expression for the required array cycles can be obtained by substituting the approximate expression for the array cycles required by addition, given in (5.3), into (5.19).

$$\text{GaussRowCyclesA}(n, m, p) = \frac{(2m - 4n/3)n^2}{p} + \frac{(2m - n/2)n}{p} - \frac{m}{p} + \frac{11n/6}{p}$$

The details of the derivation of this equation are given in Appendix C. Comparison of this result with the number of floating point operations required by the scalar algorithm, given in (5.18), shows that the cubic terms are again in agreement, indicating that the algorithm is performing at addition speed.

The expression for the required array cycles with the perfect memory architecture is given below. The only difference to the result for the Load/Store memory architecture is that the normalisation of each row requires only one, rather than three, operations.

$$\begin{aligned}\text{GaussRowCyclesP}(n, m, p) = & \sum_{i=0}^{n-1} (1 + 2(n - i - 1))\text{AC}(m - i, 1, p) \\ & + \sum_{i=0}^{n-2} (1 + 2(n - i - 1))\text{AC}(m - n, 1, p)\end{aligned}$$

5.4.2.3 Simulation Results

Both the Gauss-Jordan and Gaussian Elimination with backsubstitution algorithms were simulated with the MATRISC simulator. The results for a range of matrix orders with the standard parameters are shown in Figure 5.13 expressed in Mflop/s. The number of array cycles required in all cases was exactly given by equations (5.15) and (5.16) for Gauss-Jordan elimination, and equation (5.19) for Gaussian Elimination with backsubstitution.

The output of the calculations were all checked for correctness by calculating the residue of the calculated solution, $\hat{x}$, which is defined as $\|b - A\hat{x}\|_\infty$. A perfectly accurate solution has a residue of zero. In the inexact setting of floating point calculations, a small residual indicates an accurate solution in most cases[Golub and Van Loan 96]. Interestingly, all the residues were small (~10^{-9}-10^{-11}) even though no special measures, such as pivoting, were taken to ensure an accurate solution (see §5.4.6). This result was probably due to the matrices being used, which contained random elements in the set -10.000, -9.999, -9.998,...

9.999, 10.000. However, these results suggest that Gaussian elimination without pivoting, followed by a residual check, may be an efficient method of producing results in some circumstances.

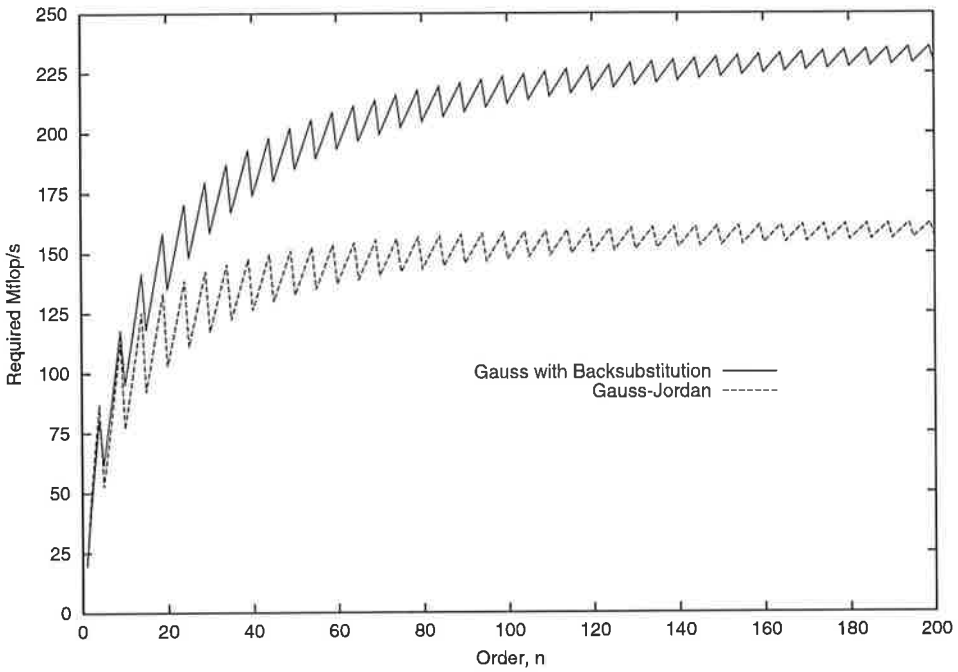


Figure 5.13: Performance of Gauss-Jordan Reduction and Gaussian Elimination with Backsubstitution

The results show a number of interesting features. The characteristic sawtooth appearance with a period equal to the matrix size clearly demonstrates the superior performance of matrices that are exact multiples of the processor array size. Given that the algorithms perform only addition-type operations, the best performance possible with the standard architectural parameters is 250Mflop/s. As the Gauss-Jordan algorithm uses 50% more floating point operations than the Gaussian Elimination method, which uses the minimum possible number of floating point operations, the highest performance it could reach is two-thirds of 250Mflop/s, or 167Mflop/s. The performance of both algorithms appears to asymptotically approach these maximum figures, which is encouraging as it indicates that numerical calculations of the architecture's performance can be quite accurate, at least for large matrices. The performance for small matrices is much less because a much larger proportion of time is spent operating on vectors that are not exact multiples of the array size. However, the algorithms reach half their peak performance for quite small matrices; order 10 for Gauss-Jordan and order 12 for Gaussian Elimination with

backsubstitution.

As neither algorithm uses any matrix multiplication, upon which the potential high performance of the MATRISC architecture depends, they achieve only a small fraction of the peak Mflop/s possible. However, these algorithms are essential building blocks for the algorithms described in the following sections.

5.4.3 Gaussian Elimination using Block Multiplication

It is possible to formulate Gaussian Elimination as a block algorithm so that the majority of floating point operations occur in matrix multiplication. To understand this block algorithm, first note that the reduction to upper triangular form can be expressed as an outer product update.

```
for i = 1 : n
        A(i, i:) = A(i, i:)/A(i,i)
        A(i+1:, i:) = A(i+1:, i:) - A(i+1:, i)*A(i, i:)
end
```

As the depth of the outer product is one, the above algorithm performs at addition speed and so provides no advantage, in terms of reduced array cycles for the MATRISC architecture, over using row operations. Importantly however, this algorithm provides the basic idea for the block multiplication algorithm that does significantly improve performance.

The block multiplication algorithm uses an outer-product-like block update that has a depth greater than one. The algorithm to reduce the matrix to triangular form is expressed in Matlab code as,

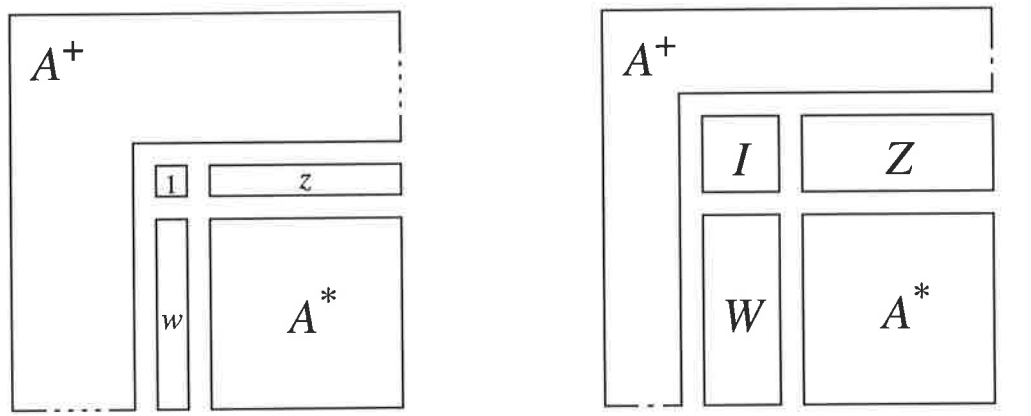
```
for i = 1 : b: n
        be = min(i+b-1,n)
        for bi = i : be
                A(bi, bi:) = A(bi, bi:)/A(bi,bi)
                for j = i : be
                        if j ~= bi
                                A(j, bi:) = A(j, bi:) - A(j, bi) .* A(bi, bi:)
                        end
                end
        end

        Z = A(i:be, be+1:)
        W = A(be+1:, i:be)
        A(be+1:, be+1:) = A(be+1:, be+1:) - W*Z
end
```

The computation proceeds using a block size, b, which would be chosen as a multiple of the processor array size for the MATRISC architecture. One of the row-operation based

algorithms discussed above is used on the first b rows to reduce the upper left $b \times b$ submatrix to the identity. Here, the row operation version of Gauss-Jordan Elimination is used for this initial reduction step.

The next step is the block matrix update, which involves multiplication with a depth of b. As b is at least as large as the processor array size, the multiplication proceeds at full speed. Derivation of this algorithm can be found in [Golub and Van Loan 96].

The outer product update step is illustrated in Figure 5.14, which shows the case for both the simple outer product algorithms and the block algorithm at an arbitrary point in the computation. A^* represents the section of the matrix yet to be reduced, while A^+ represents the reduced part of the matrix.



(a) Simple outer product update step
$A^* = A^* - wz$

(b) Block outer product update step
$A^* = A^* - WZ$

Figure 5.14: Comparison of Simple Row and Block Outer Product Updates

The speed of the backsubstitution phase can also be improved using a better algorithm. The following algorithm uses vector operations and relies on the structure of the reduced matrix when a block algorithm is used. This structure is shown in Figure 5.15.

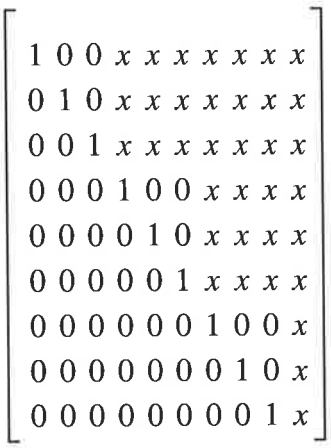


Figure 5.15: Structure of a Matrix after Reduction to Upper Triangular form with the Block Algorithm (using a block size of 3)

The algorithm to perform the backsubstitution is,

```
for i = b*floor((n - 1) / b): -b : 0
        be = min(i + b - 1, n)
        A(1:i-1, n+1:) = A(1:i-1, n+1:) - A(1:i-1, i:be) * A(i:be, n+1:)
end
```

Although this algorithm uses matrix multiplication, it only runs at matrix-vector speed for a single right hand side problem because the product is between a matrix and a column vector.

5.4.3.1 Implementation

The matrix register variables required for the block multiplication algorithm are driven by the block update step,

```
A* = A* - WZ
```

This situation was explained in detail in §4.3.1. The matrix register variables used are illustrated in Figure 5.16, which uses some different names to the discussion in Chapter 4 so as to fit in with the implementation of previous versions of Gaussian Elimination. The C variable must be as large as A to hold the result of the multiplication part of the block update, which can then be subtracted from part of A to finish the update step. As C is stored in X transpose form, the matrices W and Z must be in Y transpose and X transpose forms respectively. As W and Z are submatrices of A, it would be preferable to copy one to the X register and to form the product directly with the other submatrix. However, this is impossible as A is stored in Y normal form and neither of the two halves of the product can be in this form.

The solution is to have another variable E in Y transpose form, and to copy the matrix W

to the C variable and then into E. This double copy allows the transpose of W to be formed in the original register. That multiple copies must be formed in order to get the matrices into the correct storage form for the computation to be possible is due to the limitations of the Load/Store architecture.

Note that the storage requirements could have been met by performing the transpose on the result of the matrix product, rather than on the W matrix. However, this alternative would have been slower because the result matrix is always larger than the factors being multiplied.

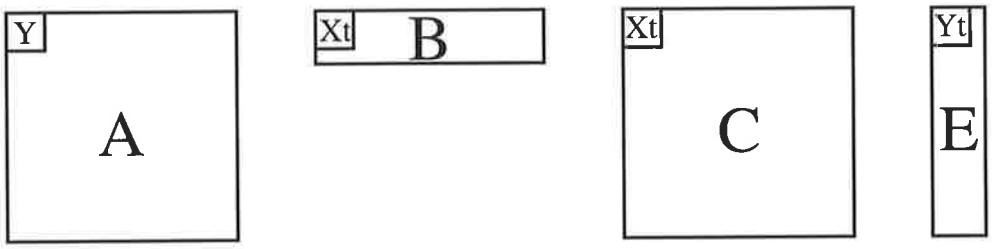


Figure 5.16: Matrix Register Variable Assignment for Block Multiplication Gaussian Elimination

The full algorithm in terms of the matrix register variables, including the vector backsubstitution, is,

```
for i = 1 : b : n
        be = min(i + b -1, n)
        for bi = i : be
                B = A(bi, bi:)
                A(bi, bi:) = B/A(bi, bi)
                B := A(bi,bi:)
                for j = i to be
                        if j ~= bi
                                C := B .* A(j,bi)
                                A(j, bi:) = A(j,bi:) - C
                        end if
                end
        end
        B = A(i:be, be+1:)
        C = A(be+1:, i:be)
        E = C
        C = E * B
        A(be+1:, be+1:) = A(be+1: be+1:) - C
end

for i = b*floor((n - 1) / b): -b : 0
        be = min(i + b - 1, n)
        B = A(1:i-1,i:be)
        E = B
        B = A(i:be, n+1:)
        C = E * B
        A(1:i-1, n+1:) = A(1:i-1, n+1:) - C
end
```

5.4.3.2 Theoretical Results

The first important result is to calculate the performance of the block outer product update step. Assuming that the dimensions of A^* of Figure 5.14(b) are $n \times m$, the average number of floating point operations per array cycle is found as,

$$\begin{aligned} \text{flop\_per\_array\_cycle} &= \frac{\text{multiplication\_flop} + \text{addition\_flop}}{\text{MC}(n, b, m, p, v) + \text{AC}(n, m, p)} \\ &= \frac{2nbm + nm}{\frac{nmb}{vp^2} + \frac{nm}{p}} \\ &= vp^2 \frac{2b+1}{b+vp} \end{aligned} \tag{5.20}$$

If the block size, b, is set equal to the virtual array size, vp, then approximately vp^2 flop are performed per array cycle, which is half the multiplication speed of the architecture. For the standard parameters, the speeds for v = 1, 2 and 4 are 1375Mflop/s, 2625Mflop/s and 5125Mflop/s respectively.

Examining the algorithm in the above section, the required number of array cycles can be expressed as,

$$\begin{aligned} \text{GaussBlockCycles}(n, m, p, b, v) = &\sum_{i=0}^{\frac{n}{b}-1} \begin{array}{l} \text{GJRC}(b, m-ib, p) + \text{AC}(m-bi-b, b, 0, p) \\ + \text{AC}(b, n-bi-b, 0, p) \\ + \text{MCC}(n-bi-b, b, 0, p) \\ + \text{AC}(m-bi-b, n-bi-b, 0, p) \\ + \text{MC}(m-bi-b, b, n-bi-b, 0, 0, p, v) \end{array} \\ &+ \sum_{i=1}^{\frac{n}{b}-1} \begin{array}{l} \text{AC}(b, bi, 0, p) + \text{MCC}(b, bi, 0, p) + \text{AC}(m-n, b, n \bmod p, p) \\ + \text{MC}(m-n, b, bi, n \bmod p, 0, p, v) + \text{AC}(m-n, bi, n \bmod p, p) \end{array} \end{aligned}$$

By substituting the appropriate approximate expressions for array cycles from (5.3), (5.7), (5.10), and (5.17), the following approximate expression for the required number of array cycles for the block algorithm can be derived. Details are shown in Appendix C.

$$\begin{aligned} \text{GaussBlockCycles}(n, m, p, b, v) = &\frac{n}{2pb}\left[2nm - \frac{4n^2}{3} - 2nb^2 + nb + 4mb^2 + 4mb - \frac{2b^2}{3} + b\right] \\ &+ \frac{n}{2p^2v}\left[2nm - \frac{4n^2}{3} - 2mb + nb + \frac{b^2}{3}\right] - \frac{mb}{p} \end{aligned} \tag{5.21}$$

Some insight can be gained into this result by examining it in the context of some broad assumptions. To further simplify the result, it is also converted to a figure in flop/s, which is

expressed relative to the peak flop/s speed for the architecture, denoted M_{peak}.

By assuming that the block size is much smaller that the size of the matrix, that is $b<<n,m$, equation (5.21) simplifies to,

$$\text{GaussBlockMFlop} = \frac{M_{\text{peak}}}{1 + \frac{vp}{b}}$$

Given the approximations made, this result is equivalent to the flop/s rate achieved by the block outer product update step, as shown in (5.20). Thus, for a fixed block size, the performance should approach the limit imposed by the block outer product update step for large matrices.

Alternatively, assuming that $b = n/2$, the approximate performance achieved is,

$$\text{GaussBlockMFlop} = \frac{2M_{\text{peak}}}{3vp}$$

which is much less than the peak speed. This result clearly shows that the block factor must be significantly less than the size of the matrix being reduced for the peak speed to be approached.

The expression for the number of cycles on the perfect memory architecture is,

$$\begin{aligned}\text{GaussBlockCyclesP}(n, m, p, b, v) = & \sum_{i=0}^{\frac{n}{b}-1} \text{GJRC}(b, m-ib, p) \\ & + \text{AC}(m-bi-b, n-bi-b, p) \\ & + \text{MC}(m-bi-b, b, n-bi-b, p, v) \\ & + \sum_{i=1}^{\frac{n}{b}-1} \text{MC}(m-n, b, bi, p, v) + \text{AC}(m-n, bi, p)\end{aligned} \tag{5.22}$$

5.4.3.3 Simulation Results

The block algorithm was simulated on the MATRISC simulator with $b = pv$. Given the limitation of the block outer product update step the highest possible performance for the standard parameters for the three cases $v = 1$, 2 and 4, are given by (5.20) as 1365Mflop/s, 2625Mflop/s and 5125Mflop/s respectively. The simulated results for $v = 1$, 2 and 4 are shown in Figure 5.17.

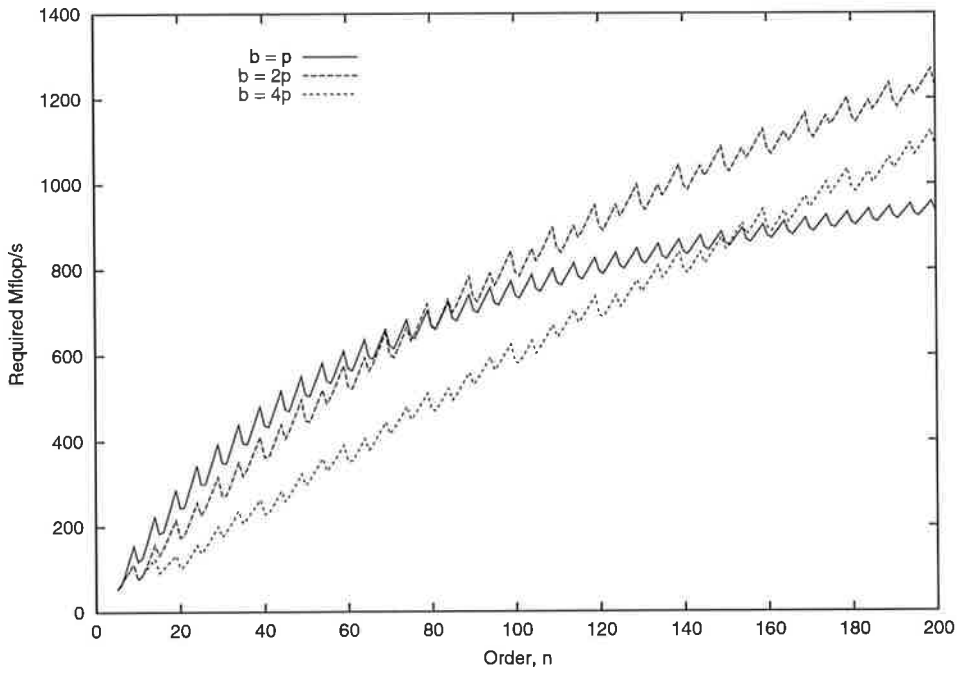


Figure 5.17: Performance of Gaussian Elimination using Block Multiplication

The $v = 1$ curve reaches a level of performance of approximately 1000Mflop/s, or about 73% of the possible peak. The $v = 2$ and $v = 4$ curves show that these cases require much larger matrix sizes before they will reach their best performance, as the curves are obviously not levelling off at $n = 200$.

5.4.4 Gaussian Elimination using Multiplication by the Inverse Matrix

Another method for solving the linear system $Ax = b$ is to multiply by the inverse of A, that is, by evaluating $x = A^{-1}b$. This method is almost never used in practice as it is more expensive, in terms of floating point operations, than solving the system using other methods because of the need to explicitly calculate the inverse. However, in the situation with many right hand sides, the computation is dominated by matrix multiplication and so the algorithm can be very efficient on the MATRISC architecture.

The inverse is calculated by solving the multiple right hand side problem below, using one of the methods previously discussed.

$$AX = I$$

This equation yields the solution $X = A^{-1}$, by the definition of matrix inverse.

5.4.4.1 Implementation

The most useful way to implement this algorithm is to use a matrix register variable in Y normal storage form that has sufficient space to insert the identity matrix to the left of matrix A. Gauss-Jordan elimination can then be applied to the top left $n \times 2n$ submatrix of the register as described previously, except that the left and right hand sides of the equation are reversed. This reduction requires two single row register variables in X transpose form. When this initial reduction is complete, the top left $n \times n$ submatrix is the required inverse, which can then be copied to another register variable H, stored in X normal form. The algorithm is completed by multiplying the remainder of the matrix by the inverse.

Expressed in MATRISC compliant Matlab code, the algorithm is,

```
A(1:n, n+1:n+m) = matrix to be reduced
A(1:n, 1:n) = I(n)
for i = i : n
        B = A(i, n+i)
        A(i, 1:2n) = A(i,1:2n) ./ B
        B = A(i, 1:2n)
        for j = 1 : n
                if j ~= i
                        C = B .* A(j, n+i)
                        A(j, 1:2n) = A(j, 1:2n) - C
                end
        end
end
H = A(1:n, 1:n);
A(1:n, 2n+1:n+m) = H*A(1:n, 2n+1:n+m)
```

The algorithm is formulated in this way so it can be applied to part of a larger matrix. For example, it can be used to replace the Gauss-Jordan elimination step in the block multiplication algorithm of the previous section. The space to the left of matrix A in this case has already been reduced to zeroes, and so the identity can be copied there without causing errors.

This algorithm will only work correctly only if n is a multiple of p; otherwise, some of the right hand sides will be solved along with the inversion step.

5.4.4.2 Theoretical Results

Examining the algorithm above, the required number of array cycles can be expressed as,

$$\begin{aligned}\text{GaussInvertCycles}(n, m, p, v) = {} & \text{GJRC}(n, 2n, p) + \text{AC}(n, n, 0, p) + \text{MCC}(n, n, 0, p) \\ & + \text{MC}(n, n, m-n, 0, 0, p, v)\end{aligned} \tag{5.23}$$

By substituting the appropriate approximate expressions from equations (5.3), (5.7), (5.10) and (5.17), the number of array cycles required for the multiplication by the inverse

method can be expressed as,

$$\text{GaussInvertCyclesA}(n, m, p, v) = \frac{mn^2 - n^3 + 3pvn^3}{p^2 v} + \frac{9n^2}{2p} + \frac{n}{p}$$

Comparison of this result with the number of cycles required for Gauss-Jordan elimination, given in (5.17), shows that multiplication by the inverse is faster when

$$m > 2n + \frac{n^2 + 2vpn}{2vpn + vp - n}$$

Details of the derivation are given in Appendix C. The second term is much smaller than the first, and for $n = vp$, which is the most interesting case, it is 1.5. Thus, the multiplication by the inverse method should be faster for systems where m is just a little greater than $2n$. As forming the inverse requires solving an $n \times 2n$ problem using Gauss-Jordan elimination, an obvious lower limit of $m = 2n$ exists for the size of matrix that will benefit from this method. That the method is faster at such a small value above this lower limit shows the importance of using multiplication as much as possible on the MATRISC architecture.

The expression for the number of cycles on the perfect memory architecture is,

$$\text{GaussInvertCyclesP}(n, m, p, v) = \begin{array}{c}\text{GJRC}(n, 2n, p) + \text{AC}(n, n, p) \\ + \text{MC}(n, n, m - n, p, v)\end{array} \tag{5.24}$$

The difference from the Load/Store memory architecture case is that the calculated inverse does not need to be copied.

5.4.4.3 Simulation Results

As performance of the explicit inverse method can only be faster than other methods when $m > 2n$, the situations of interest are for wide matrices; simulations were therefore performed for $n = 5$, 10 and 20, with $m = n+1$ to 200. The results of this inverse method in comparison to the two row operation methods are shown in Figure 5.18.

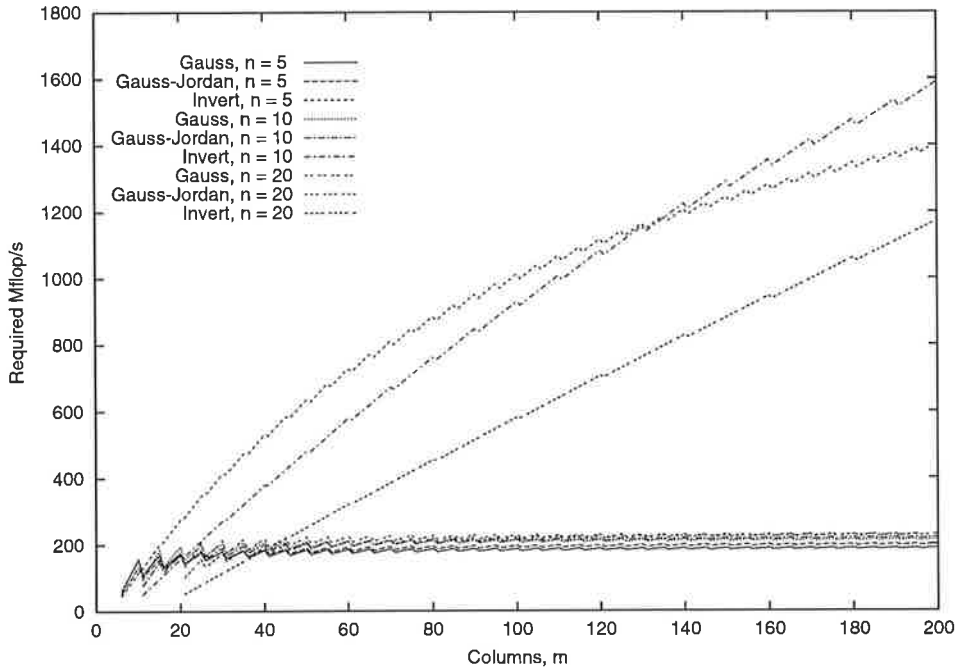


Figure 5.18: Performance of Gaussian Elimination using Multiplication by the Inverse Matrix

These results show that multiplying by the inverse is faster than row operations for a $n \times m$ matrix if m just a little greater than $2n$, in agreement with the numerical results above. As this algorithm will eventually be dominated by matrix multiplication for a sufficiently wide matrix, the limiting peak speeds for the cases $v = 1$, 2 and 4 are 2500Mflop/s, 5000Mflop/s and 10000Mflop/s respectively. None of the cases come close to these speeds for matrix orders below 200, although they achieve a large speed-up over the row operation algorithms.

5.4.5 Fast Gaussian Elimination

A fast algorithm that combines the best aspects of all the previous algorithms is now described.

The first step in defining the Fast algorithm is to realise that all algorithms discussed thus far have particular matrix sizes for which they are the most suitable. For small matrices, Gaussian Elimination with backsubstitution is best, unless they are very wide, in which case the explicit inverse algorithm is faster. Larger matrices are best handled using the block algorithm, with increasing block size as the matrix size increases. The Fast algorithm selects the fastest method for the particular matrix problem size.

The second step is to realise that many of the reduction algorithms discussed so far have

recursive structure. For example, the block algorithm uses another reduction algorithm to reduce each block, and the inverse algorithm must use a different algorithm to calculate the inverse. Whenever the Fast algorithm must solve a smaller subcomponent, it calls itself recursively to make sure that the subcomponent is solved as quickly as possible. When using block multiplication, the fastest block sizes is chosen from a the following set; all the virtual sizes and all multiples of the maximum virtual size, up to the size of the system being solved.

5.4.5.1 Implementation

As this algorithm is relatively complex, and particularly because of its recursive nature, it has not been implemented in Masm for the simulator. However, it is possible to envisage how it would be implemented, and to assess its performance theoretically. The matrix register variables used for the block multiplication algorithm, plus a small square matrix variable H in X normal form for the explicit inverse method, would be sufficient for implementing the fast algorithm. The variable H need only be small, as the inverse method is never the fastest for matrices with a large number of rows.

Three limited versions of the Fast algorithm have been implemented in Masm. Both are variations of the block multiplication algorithm, where the block reduction step is performed with a faster algorithm.

In the first two versions, the block algorithm with a virtual factor of 2 or 4 is used, except that the step of reducing the pivot block to the identity is performed using block algorithm with a virtual factor of 1. The two resulting algorithms are called Block 12 and Block 14.

The third version uses the block algorithm with a virtual factor 4, except that step of reducing the pivot block is performed using explicit pivot block inversion. This algorithm is labelled Invert in the following results.

5.4.5.2 Theoretical and Simulated Results

To calculate the performance of the Fast algorithm the following set of mutually recursive functions were used.

$$\text{GaussFastCycles}(n, m, p) = \max\begin{cases} \text{GRC}(n, m, p) \\ \text{GJRC}(n, m, p) \\ \text{GIFC}(n, m, p) \\ \text{GBFC}(n, m, p) \end{cases}$$

$$\text{GIFC}(n, m, p) = \begin{array}{l}\text{GFC}(n, 2n, p) + \text{AC}(n, n, p) + \text{MCC}(n, n, p) \\ \quad + \text{MC}(n, n, m-n, p, v)\end{array}$$

$$\text{GBFC}(n, m, p) = \max\begin{cases}\text{GBFCB}(n, m, p, p, 1) \\ \text{GBFCB}(n, m, p, 2p, 2) \\ \text{GBFCB}(n, m, b, 4p, 4) \qquad b = 1 \ldots \dfrac{n}{4p}\end{cases}$$

$$\text{GBFCB}(n, m, p, b, v) = \begin{array}{l}\text{GFC}(b, m-ib, p) + \text{AC}(m-bi+1, b, p) \\ \quad + \text{AC}(b, n-bi+1, p) \\ \quad + \text{MCC}(n-bi+1, b, p) \\ \quad + \text{AC}(m-bi+1, n-bi+1, p, v) \\ \quad + \text{MC}(m-bi+1, b, n-bi+1, p, v)\end{array}$$
$$+ \sum_{i=0}^{\frac{n}{b}-1} \begin{array}{l}\text{AC}(b, bi, p) + \text{MCC}(b, bi, p) + \text{AC}(m-n, b, p) \\ \quad + \text{MC}(m-n, b, bi, p, 1) + \text{AC}(m-n, bi, p)\end{array}$$

Note that GFC is an abbreviation for GaussFastCycles.

The equations for the behaviour of the perfect memory architecture are similar, but with the equation for GIFC modified as in (5.24) and the equation for GBFCB modified as in (5.22).

A comparison of the Fast algorithm with the block algorithm using virtual factor 4 and the three limited fast algorithms that were implemented in Masm is shown in Figure 5.19.

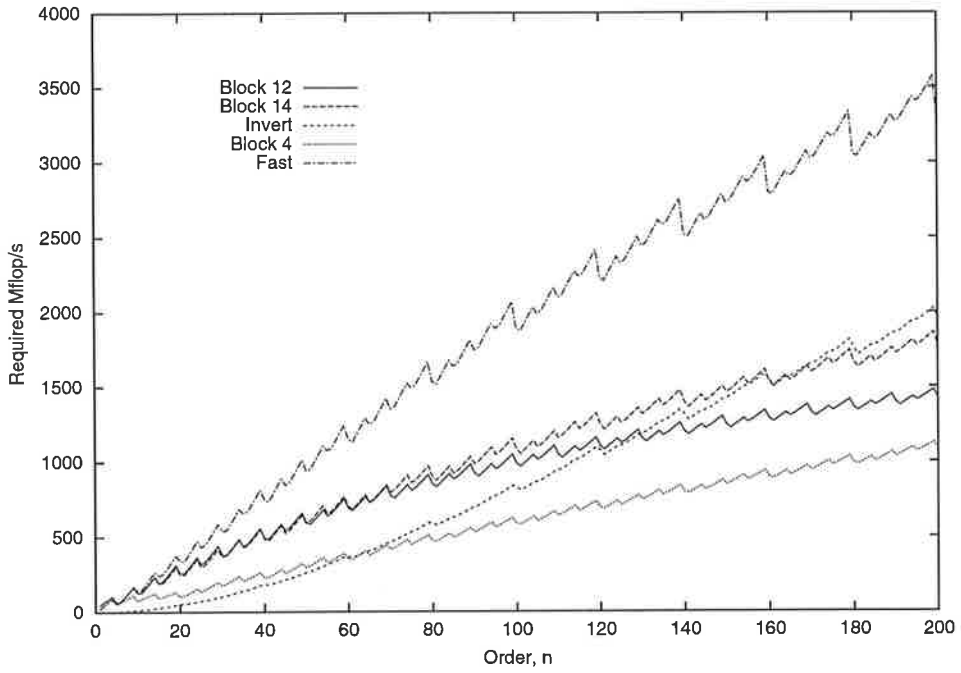


Figure 5.19: Performance of Theoretical Fast Algorithm Compared with Simulated

The Fast algorithm clearly performs much better than any of the other algorithms. Also, the Block 12, Block 14 and Invert algorithms all do much better than the simple block multiplication algorithm (labelled Block 4). None of these algorithms achieves close to its peak performance for matrix sizes below 200. The results up to order 1000 are shown in Figure 5.20. Note that these curves, and all that follow for order up to order 1000, are plotted from the results for n = 100, 200, ... 1000, so they don't have the characteristic sawtooth appearance (simulation of every order up to 1000 would have required a prohibitively large amount of computation time).

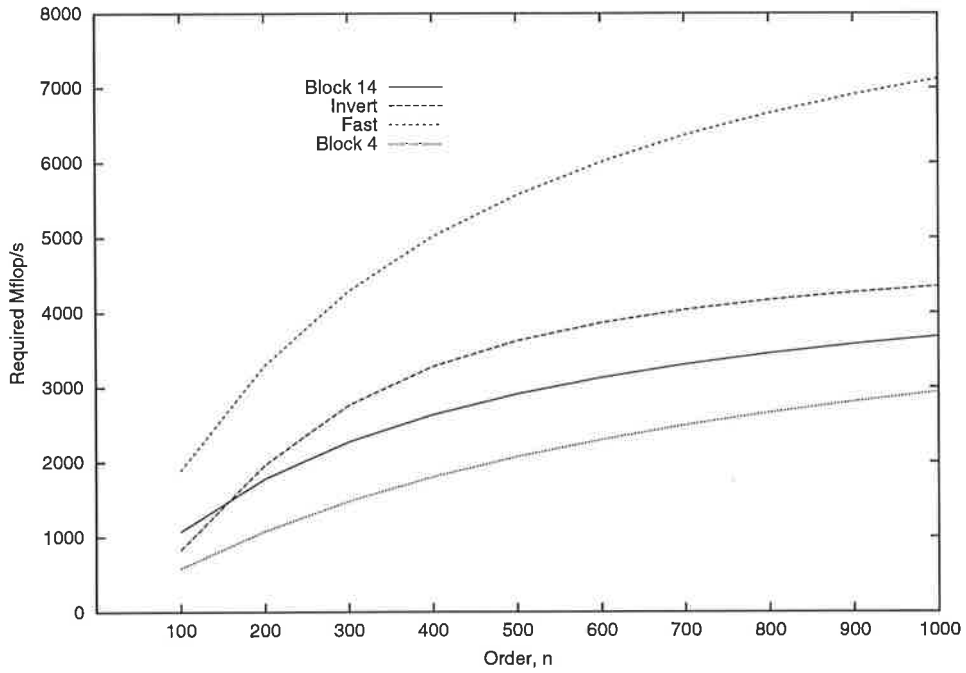


Figure 5.20: Performance of Theoretical Fast Algorithm Compared with Simulated - Large Matrix Sizes

Recall that because of their use of a maximum block size of $4p$, the performance of Block4, Block 14 and Invert methods is limited to 5000Mflop/s. The Fast method, which is not restrained in this way, achieves an impressive 7200Mflop/s, which represents 72% of peak performance. Importantly, achieves half of this speed for matrices of order ~250.

Although there are undoubtedly many small improvements that could be made to the Fast algorithm, it would seem that this algorithm must be relatively close to optimum for the MATRISC architecture, at least in terms of the number of array cycles required. This algorithm harnesses the best aspects of a wide variety of techniques. The experience gained in developing this algorithm suggests that further improvements would only come with the investment of an inordinate amount of effort. In addition, the fact that the Fast algorithm achieves 72% of peak performance shows that there are not huge improvements in speed still to be made.

Now that an optimised algorithm has been devised, it is appropriate to examine its performance for a range of architectural parameters. Since only array cycles are being considered at present, only varying the array size, p, and the maximum virtual factor, V, will be of interest. The performance of the Fast Algorithm for array sizes 5 and 10, with maximum virtual factors 1, 2 and 4 is shown in Figure 5.21 and Figure 5.22.

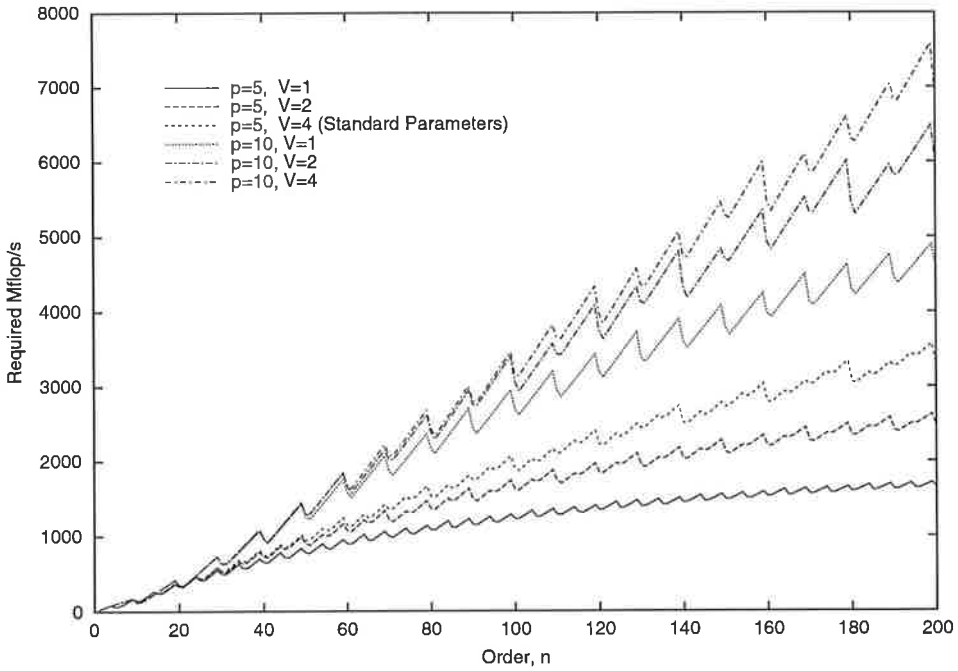


Figure 5.21: Performance of Fast Gaussian Elimination for Different Array Configurations

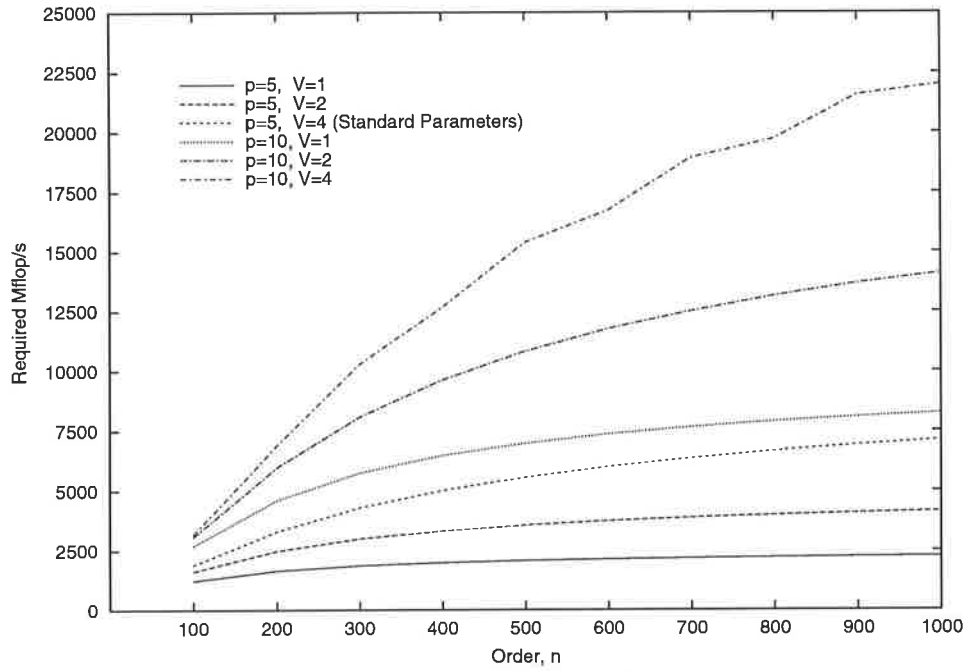


Figure 5.22: Performance of Fast Gaussian Elimination for Different Array Configurations - Large Matrix Sizes

As the performance of multiplication is proportional to vp^2, doubling v will double peak performance and doubling p will quadruple peak performance. However, the increase in the

performance of any algorithm is less, particularly for small matrix sizes, due to increased non-multiplication overheads. Increasing v only increases the speed of multiplication and so, by Amdahl's Law, will not increase overall performance by the same factor. Doubling p has a more significant effect on the overall speed as it also increases the addition speed by a factor of two. Comparison of the two cases $p = 5$, $v = 4$ and $p = 10$, $v = 1$, which have the same peak performance reveals that larger p is faster than larger v, and again, the difference is largest for small matrices. In terms of hardware, the $p = 10$, $v = 1$ case requires four times as many processors and twice as much register space. Thus the fact that the $p = 5$, $v = 4$ case achieves 86% of the performance of the $p = 10$, $v = 1$ case for large matrices demonstrates that using the virtual array concept can be very effective in increasing performance without increasing memory bandwidth.

An alternative way to examine these performance figures is to plot each as a percentage of the peak performance for that particular configuration. The same data as Figure 5.22 is replotted in this fashion in Figure 5.23.

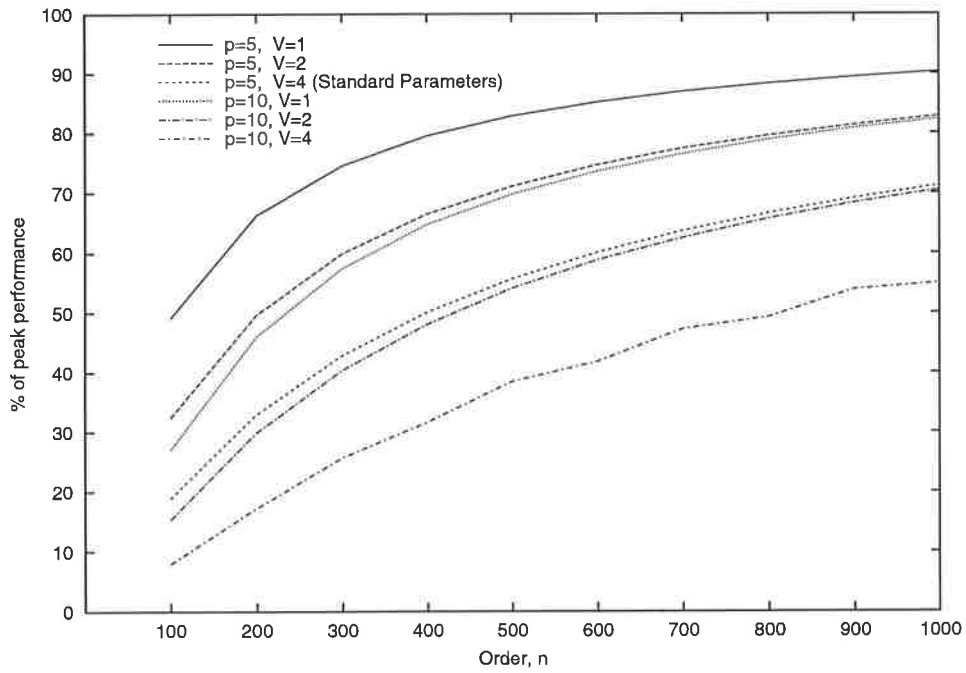


Figure 5.23: Performance of Fast Gaussian Elimination for Different Array Configurations - Large Matrix Sizes as a Percentage of Peak Performance

This graph clearly shows that performance as a percentage of peak is a function of the maximum virtual array size, pV. The reason for this phenomenon is that the individual operations performed by the algorithm operated at a percentage of peak speed that is a

function of the virtual array size. Multiplication operates at 100% of peak when using the maximum virtual factor. Addition-type operations perform p floating point operations per cycle out of a possible peak of $2Vp^2$, and so operate at $100/(2pV)\%$ of peak. The fact that the percentage of peak performance achieved by different two systems with the same virtual size is not exactly the same comes about because addition-type operations must be divided into array-sized blocks, which introduces a dependence on p, and because some multiplication uses less than the maximum virtual factor.

5.4.6 Pivoting Algorithms

All algorithms discussed so far can fail even when the system of equations actually has a solution if at any time the pivot element is zero. The augmented matrix of a simple example of a system that causes such failure is shown below.

$$\begin{bmatrix} 0 & 1 & 2 \\ 1 & 1 & 4 \end{bmatrix}$$

Furthermore, numerical instability, causing inaccurate results, can occur whenever the pivot element is sufficiently small. The solution to these difficulties is to make the pivot element as large as possible by swapping rows and/or columns. In the example above, interchanging the two rows produces a system solvable by Gaussian elimination. It is also possible to swap columns, which requires keeping track of the swaps because the corresponding elements of the result are also swapped. Swapping columns and rows is known as *complete pivoting*, while swapping only rows is known as *partial pivoting*. Except in cases where numerical rank deficiency is an issue, partial pivoting is considered sufficient to ensure accuracy (see [Golub and Van Loan 96]). For that reason, and because it is also much simpler, it is the method chosen here.

The modifications to the row operation algorithm to perform pivoting are quite straightforward. Before each pivot is used, the column below the pivot is searched for the largest element and the row containing it is swapped into the pivot position. As the searching part of the pivoting operation uses only scalars, it is very inefficient, not just in its use of array cycles, but also in terms of the control processor instruction overhead. To help reduce the control processor overhead, a new template, represented in the Matlab code as the function MaxElement(), was added to the Masm compiler to find the maximum element in the column of a matrix. This operation could be have been written directly in Masm but

comparing the individual elements of the column would involve applying matrix operations to scalars. Although such matrix operations require the same number of array cycles no matter how they are performed they require more control processor cycles than they would if they were expanded using a template that assumed only scalar quantities. As the operations in the MaxElement() template assume scalars it uses significantly fewer control processor instructions. The algorithm for the reduction to upper triangular form is,

```
for i = 1 : n
        max_index = MaxElement(A(i:,i),B)
        B = A(max_index, i:)
        A(max_index, i:) = A(i, i:)
        A(i, i:) = B                    -- Can remove this line

        B = A(i, i:)                    -- And this line
        A(i, i:) = B./A(i,i)
        B = A(i,i:)
        for j = i+1 : n
                C = B .* A(j,i)
                A(j, i:) = A(j,i:) - C
        end
end
```

The second argument to the template is a temporary variable, which is necessary because the template cannot allocate register memory for its temporary storage requirements.

Pivoting in block algorithms is much more complex and makes considerably more of an impact on performance. During the scalar portion of the algorithm, the whole pivot column must be zeroed so that the correct choice of pivot can be made for the next column. If the row operations are applied to the whole row then we have nothing more than the row operation algorithm. If the row operations are applied to whole rows only in the pivot block, then swapping a row is complicated by the need to un-transform the row going out of the pivot block, and to transform the row coming into the pivot block. Thus pivoting is best performed using an explicit pivot block inversion algorithm, such as the Invert algorithm, because such algorithms only operates on the columns under the pivot block. In addition, when the block update step is applied the original values from the columns under the pivot block are needed so row operations that zero these elements must operate on a copy.

The pivoting algorithm implemented for simulation uses a copy of the columns under and in the pivot block and performs Gauss-Jordan elimination on them, using row operations with pivoting. Whenever a swap is made, the corresponding rows in the original matrix are also swapped. When swapping is complete, the original matrix is unchanged except that rows have been swapped into positions so that the maximum possible pivots will

occur. The normal explicit pivot block inversion is then applied. This algorithm may waste time by doing some work twice but it is much simpler than trying to calculate the inverse of the pivot block while the pivoting is being performed. The amount of extra work is not likely to be large because the probability is that the maximum pivot will be outside the pivot block and thus swapping it in will require all the work done on the right hand side of that row to be repeated anyway.

5.4.6.1 Implementation

The pivoting algorithms require another matrix register variable called PC that uses Y normal storage, and has as many rows as A and columns equal to the block size. The code for just the pivoting part of the algorithm is,

```
PC = A(i,: i:be)
for bi = 1 : b
        max_index = MaxElement(PC(bi:,bi),B)
        B = A(i + max_index -1, i:)
        A(i + max_index -1, i:) = A(i + bi -1, i:)
        A(i + bi -1, i:) = B
        B = PC(max_index, bi:)
        PC(max_index, bi:) = PC(bi, bi:)

        PC(i, i:) = B./PC(bi,bi)
        B = PC(bi,bi:)
        for j = bi+1 : n-i
                C = B .* PC(j,i)
                PC(j, i:) = PC(j,i:) - C
        end
end
```

Note that the rows in the original matrix A are swapped at the same time as those in the temporary PC matrix.

5.4.6.2 Simulation Results

The performance of the pivoting and non-pivoting versions of Gaussian elimination using row operations and using explicit pivot block inversion are shown in Figure 5.24 and Figure 5.25.

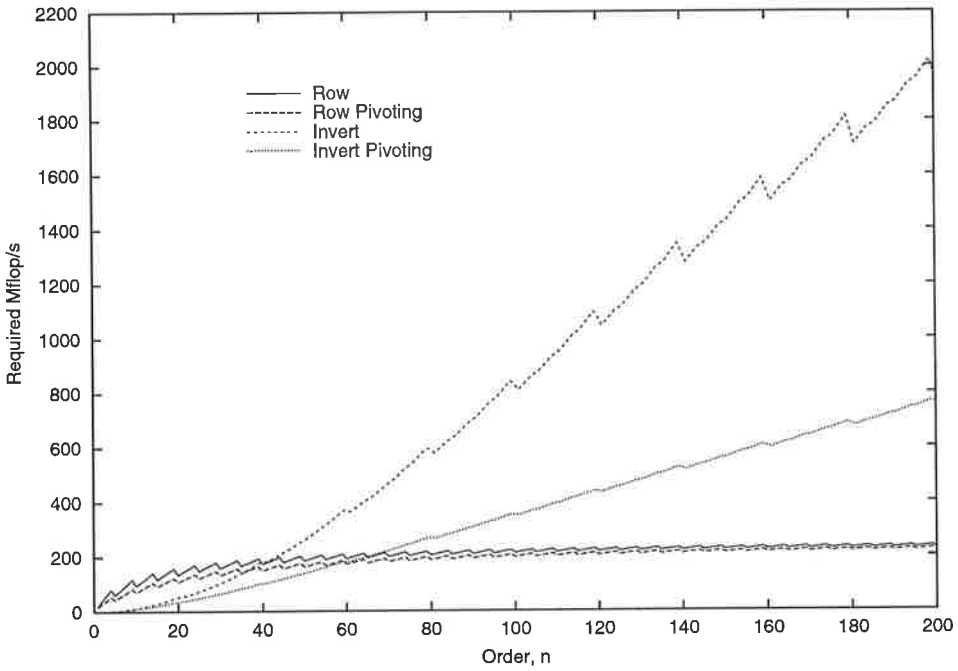


Figure 5.24: Performance of Pivoting Algorithms

The results show that pivoting has little effect on the performance of row operation algorithm, because this algorithm operates at addition speed and so the overhead of the $O(n^2)$ pivoting operation proceeding at scalar speed is not severe. The performance of the explicit block inversion algorithm, however, is more than halved by the inclusion of pivoting. This is due to the fact that the difference between multiplication speed and scalar speed is much larger and because of the extra copying required to make the block algorithm work. So pivoting should be avoided if possible because it severely reduces performance.

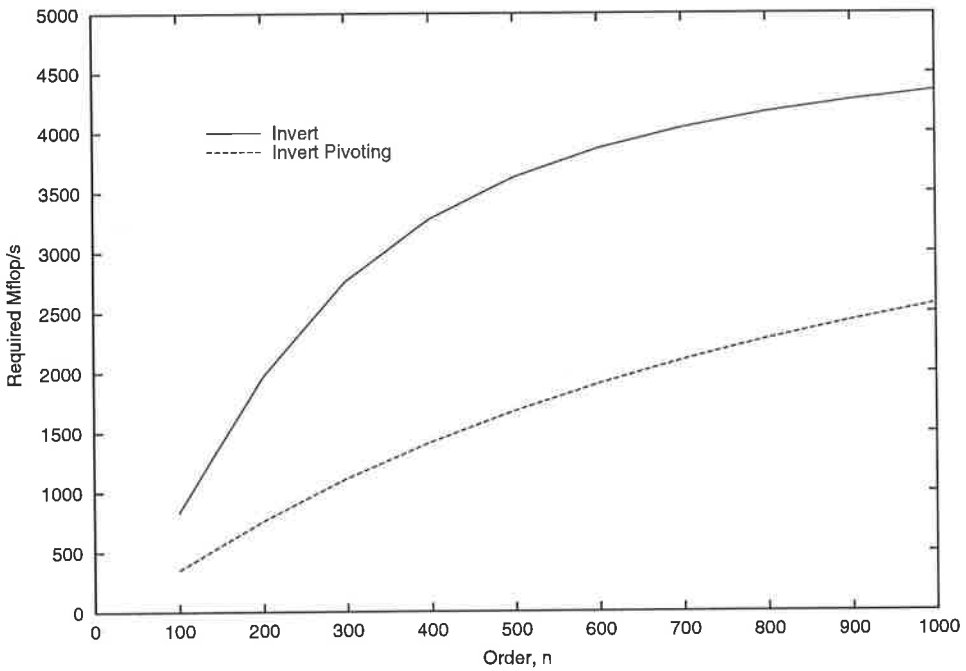


Figure 5.25: Performance of Pivoting Algorithms - Large Matrix Sizes

For larger matrix sizes, the relative performance of the pivoting algorithm improves a little due to the $O(n^2)$ complexity of the pivoting step.

The accuracy of the performance figures produced by the MATRISC simulator in predicting the performance of a real architecture are based on assumptions, one of which is that there are no dependencies between two successive operations that may cause the architecture to stall. This assumption breaks down when pivoting is used due to the large amount of data-dependent branching that occurs during the search for the pivot element. The performance of this architecture in a real system would therefore be less than that predicted by these results.

Many of the potential stalls could be avoided by using appropriate branch prediction, because most of the time the largest element is not being found. However this would entail much more complex control logic.

5.4.7 Load/Store Memory Architecture Overhead

As discussed in §5.1.1, the Load/Store memory architecture requires more operations compared with a hypothetical perfect memory architecture. This does not include data transfer between main memory and registers, that is covered in the next section. By evaluating the expressions derived for array cycle counts for the Fast algorithm with the

perfect memory architecture, and comparing them with results for the Load/Store memory architecture, the overhead of the Load/Store architecture can be defined as,

$$\text{Load/Store memory architecture overhead} = \frac{C_{A(\text{Load/Store})} - C_{A(\text{perfect})}}{C_{A(\text{perfect})}} \times 100\%$$

where $C_{A(\text{Load/Store})}$ is the number of array cycles required when using the Load/Store memory architecture and $C_{A(\text{perfect})}$ is the number of array cycles required when using the perfect memory architecture.

The graphs in Figure 5.26 and Figure 5.27 show the Load/Store memory architecture overhead for a range of array sizes and virtual factors.

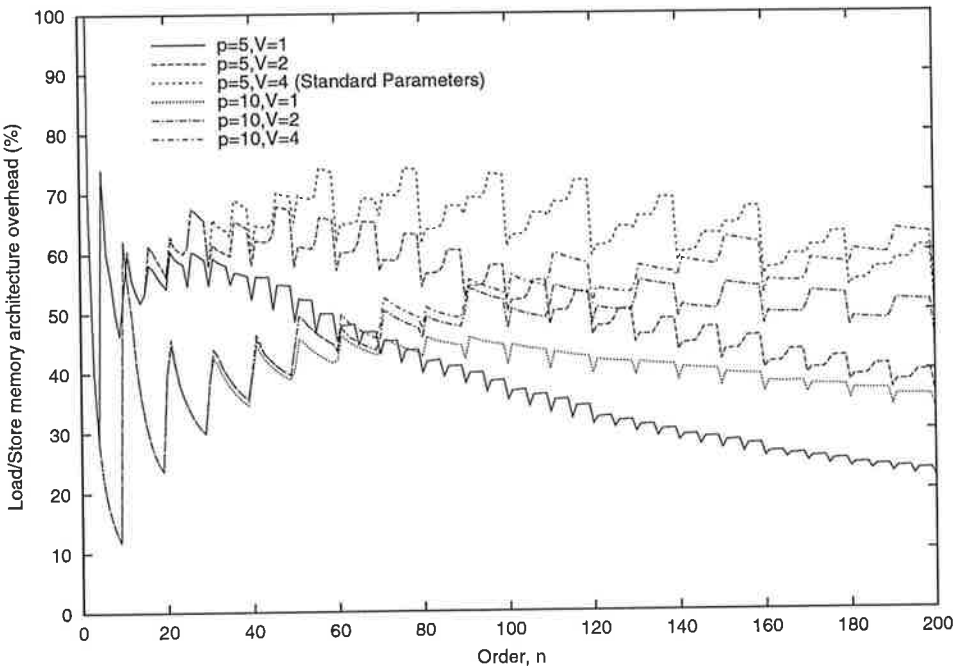


Figure 5.26: Overhead of the Load/Store Architecture versus the Perfect Architecture for Fast Gaussian Elimination

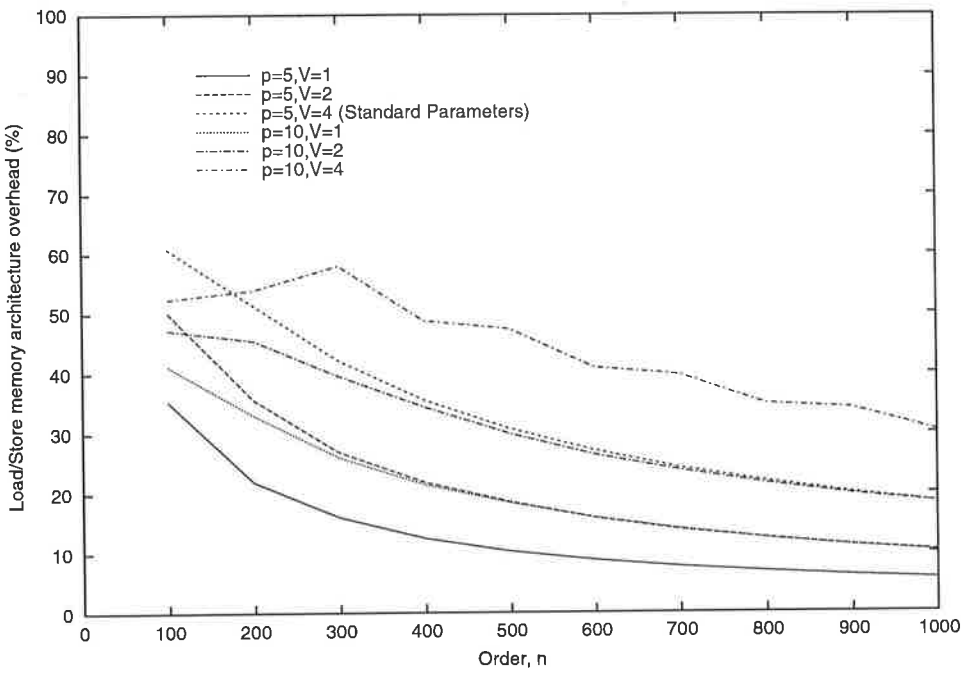


Figure 5.27: Overhead of the Load/Store Architecture versus the Perfect Architecture for Fast Gaussian Elimination - Large Matrix Sizes

The figures show that the overhead is large for matrix orders less than 200, being between 50% and 75% for the standard parameters. The overhead decreases for large matrix orders, falling to under 20% for order 1000. There is again a very strong correlation between the results for different systems with the same virtual array size, with the overhead increasing for larger sizes.

These results indicate that an improved memory architecture could bring about significant improvements in performance for small matrices, but much more limited improvements for large matrices.

5.4.8 Load/Store Transfer Time and Register Size Limitations

Until this point, the time taken to move the data between main memory and the matrix registers and the limitation of finite matrix register space have been ignored. In the sense of throughput, the preceding calculations are a valid prediction of performance, provided that the data fits in the available register space and the time required for load/store data transfer is less than that required for the computation. This assertion holds because the architecture allows data transfer to occur in parallel with computation and because data transfer only occurs at the beginning and the end of the algorithm. The relative time required for data transfer will now be examined, followed by an algorithm to cover the case when the data is

larger than the matrix registers.

The time required for the transfer of data between the registers and main memory as a proportion of the time required for computation is shown for a range of array sizes and virtual factors in Figure 5.28 and for larger matrix sizes in Figure 5.29. The vertical scale represent represents the load/store transfer time measured as a percentage of the computation time, calculated as,

$$\text{load/store transfer time} = \frac{FC_{LS}}{C_A} \times 100\%$$

where C_{LS} is the required number of load/store cycles, C_A is the required number of array cycles, and F is the load/store bandwidth factor.

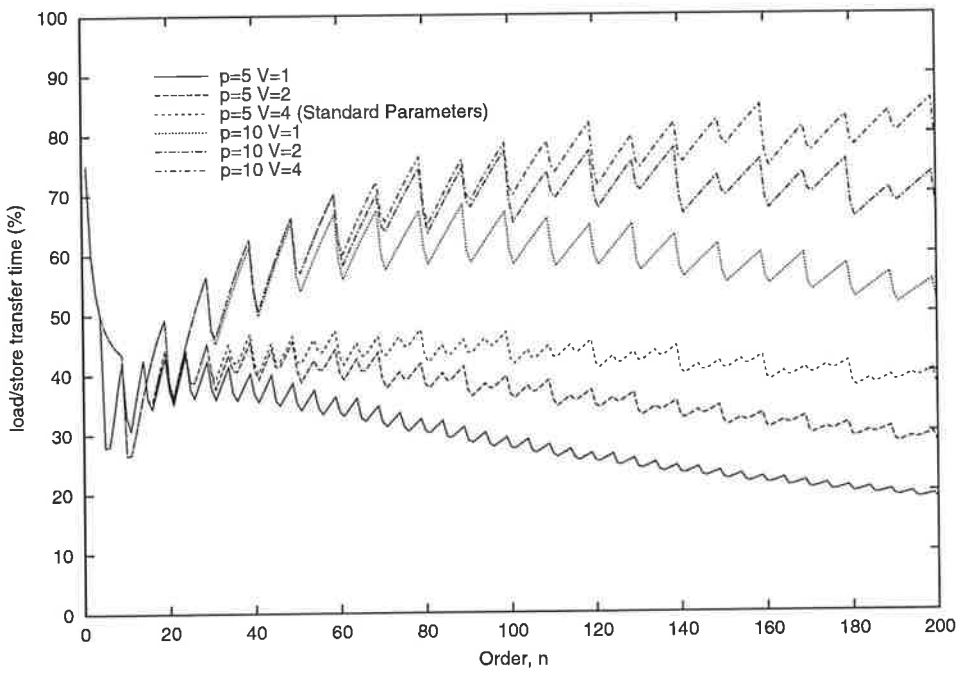


Figure 5.28: Load/Store Transfer Time for Fast Gaussian Elimination

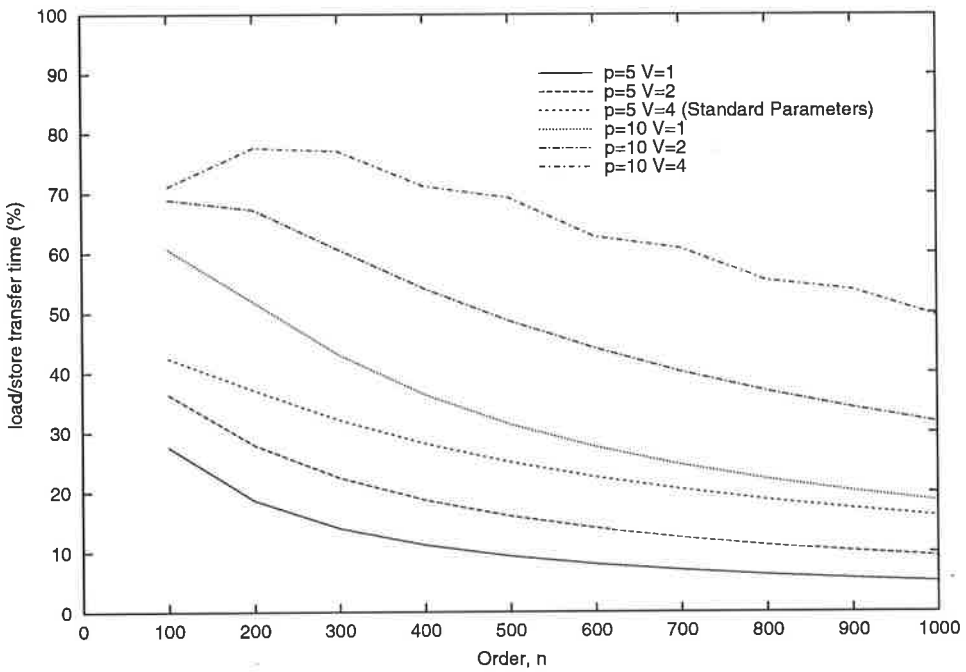


Figure 5.29: Load/Store Transfer Time for Fast Gaussian Elimination - Large Matrix Sizes

In all cases shown, the percentage values are less than 100 indicating that load/store transfers require less time than computation. These results indicate that, given sufficient register space, the architecture is compute-bound and that the performance calculations in the previous sections provide a valid prediction of performance.

The percentage transfer time drops significantly for large matrices. For the standard parameters, it falls from between 35% to 50% for orders up to 200, to less than 17% for order 1000. This result is expected because Gaussian Elimination is a level-3 algorithm, that is, it has $O(n^3)$ computational complexity and $O(n^2)$ data transfer complexity.

The graphs show that the load/store transfer percentage increases with array size and with maximum virtual factor, suggesting that a large enough array would be memory-bound for small matrix sizes. However, the level-3 nature of the algorithm means that it must be compute bound for sufficiently large matrices regardless of array size.

5.4.8.1 Main Memory Gaussian Elimination

The preceding results show that Gaussian Elimination is compute-bound when an unlimited amount of register space is available. When the problem to be solved is larger than the registers, a different algorithm that can operate out of main memory and still be compute-bound, must be used. As with the multiplication algorithms, which operate out of

main memory, this new algorithm have certain restrictions on the size of matrix for which it is applicable. The algorithm is simply the block algorithm, using multiplication by the inverse to reduce the block. This method is chosen because it allows the calculation to be broken into steps such that a fixed register size can be used to operate on matrices of unlimited size. Each iteration of the algorithm proceeds in three stages, each of which is independently compute-bound. The size of the block, b, is chosen to be at least the critical size, but small enough that two $b \times b$ matrices will fit into one register. The three stages are;

- **Inverting the pivot block -** This stage will be compute-bound because it simply involves performing Gaussian Elimination on a matrix that fits into register memory.
- **Multiplying the rest of the row by the inverse -** This stage will be compute-bound because it involves performing a multiplication operation on matrices larger than critical size.
- **Performing the block outer product update step -** This stage involves the step shown in Figure 5.14(b) and will compute-bound providing certain conditions are met.

To perform the block outer product step, a limited register size multiplication algorithm of the style of §5.3.4.2 will be used. For simplicity, loading will be performed using method 1. Additional computation time will be required to perform the addition part of the block update and additional load/store time will be required to load the section of the matrix to be updated. The dimensions of the various matrices are shown in Figure 5.30, which reproduces part of Figure 5.14(b).

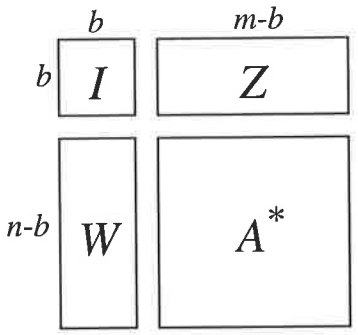


Figure 5.30: Dimensions of Matrices in the Main Memory Block Outer Product Update

The condition for the update to be compute-bound is,

$$\text{compute\_time} > \text{load\_store\_time}$$

$$\frac{(n-b)b(m-b)}{vp^2} + \frac{(n-b)(m-b)}{p} > F\left(\begin{array}{c}(n-b)(m-b) + b(n-b)\dfrac{b(n-b)}{S} \\ +(m-b)b + (n-b)(m-b)\end{array}\right)$$

$$(b+vp)(n-b)(m-b) > Fvp^2\left((2n-n)(m-b) + \frac{b^2(n-b)^2}{S}\right)$$

Assuming that $n = m$ and $b = \alpha Fvp^2$, this result reduces to,

$$\left(\alpha + \frac{1}{Fp}\right)(n-b) > 2n - b + \frac{b^2(n-b)}{S}$$

$$n > b\left(1 + \frac{1}{\alpha + \dfrac{1}{Fp} - 2 - \dfrac{b^2}{S}}\right)$$

Evaluating this inequality for the case where b is equal to the critical size ($\alpha = 3$) and the registers each have enough space to store two critical-sized matrices ($S = 2b^2$), the result is $n > 2.30b$: the algorithm can continue until this condition fails. The part of the matrix remaining to be reduced at this point is of order approximately $1.3b$, which can be loaded entirely into to the registers and reduced.

5.4.9 Control Computation Time

The operation of the matrix data path is controlled by the matrix controller. The integer scalar operations it performs occur in parallel with the operation of the processor array, and the time required by these operations can be measured relative to the matrix computation time in the same manner as the load/store time.

The time required for the control processor instructions as a proportion of the time required for computation, for a number of algorithms, is shown in Figure 5.31 and for larger matrix sizes in Figure 5.32. The vertical scale represents the amount of time required to execute the control processor instructions, expressed as a percentage of the time required by the computation performed on the processor array. It is calculated as,

$$\text{control instruction time} = \frac{GC_{CP}}{C_A} \times 100\%$$

where C_{CP} is the number of control processor computation cycles, C_A is the number of array cycles, and G is the ratio of data controller to control processor execution bandwidth. The control is taking more time than the computation for percentages above 100. Note the

there are no figures for the Fast algorithm as the number of control processor instructions can only be measured by simulating the algorithm on the MATRISC simulator.

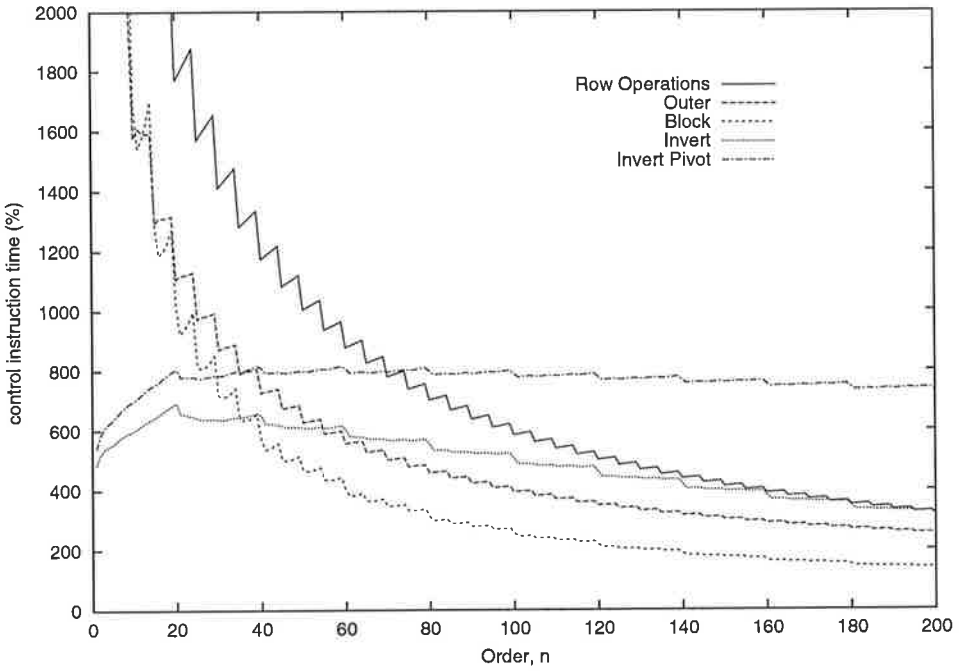


Figure 5.31: Control Instruction Time for Gaussian Elimination

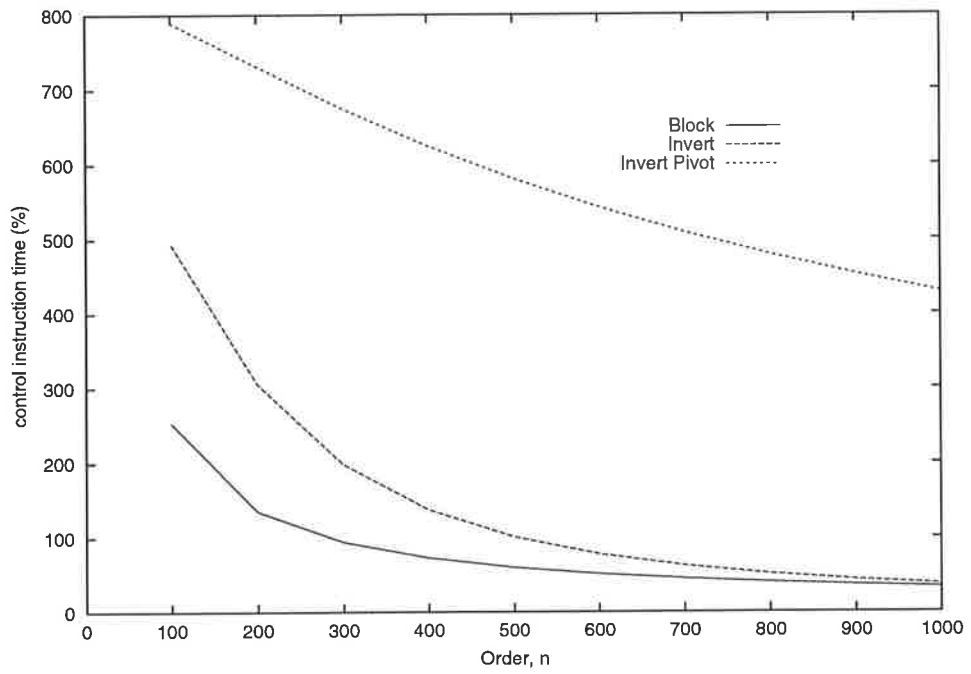


Figure 5.32: Control Instruction Time for Gaussian Elimination - Large Matrix Sizes

These figures show that the time required to execute the control processor instructions far

exceeds the time spent performing the calculation. One major reason for this poor performance is that the Masm compiler does not optimise the code for the controller in any way. Writing an optimising compiler would be straightforward but time consuming; fortunately, there is a better way to estimate the performance in the best case. The absolute minimum number of control processor instructions that must be executed are all the SET instructions that change the data path registers rather than simply set them to the same value again, and all the DO instructions. By having the MATRISC simulator count only this subset of the control processor instructions, the results in Figure 5.33 and Figure 5.34 are obtained.

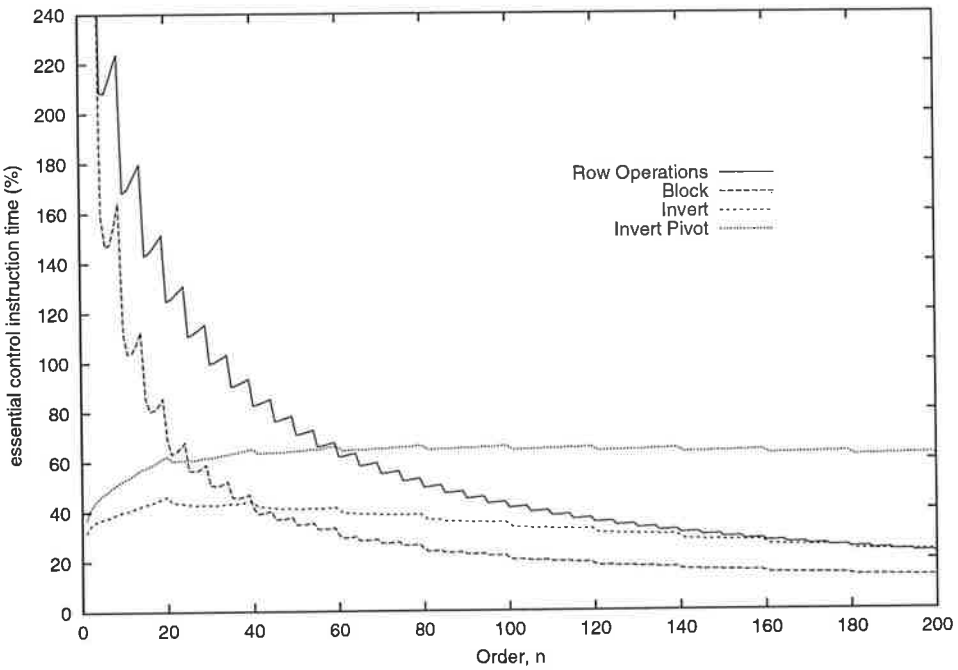


Figure 5.33: Essential Control Instruction Time for Gaussian Elimination

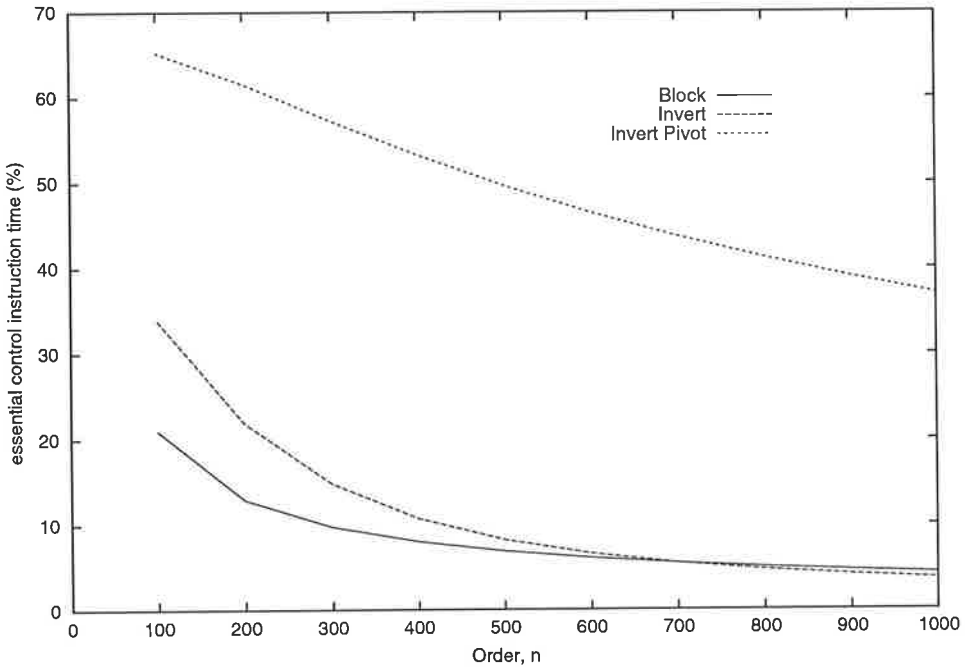


Figure 5.34: Essential Control Instruction Time for Gaussian Elimination - Large Matrix Sizes

The time taken by the essential controller instructions is less than the time required for the calculation for all algorithms for matrix orders above 30. By using a control processor program consisting of only the required SET and DO instructions, these minimal operation counts can be achieved; this is effectively an unrolling of every loop in the algorithm. However, the program memory required to store these instructions could become very large. Some actual operation counts are shown in Table 5.3. These figures suggest that this method may be acceptable for the order 100 case, where the Block 14 algorithm only uses 10428 essential control processor operations. However, the number of operations required for the order 1000 case are clearly too large.

Algorithm	Order, n = 100		Order, n = 1000	
	Total	Essential	Total	Essential
Row	1875051	111880	N/A	N/A
Invert	406300	23109	5737605	421425
Invert Pivot	1540080	99836	111683895	7381693
Block 14	161071	10428	6063646	552873

Table 5.3: Control Processor Operation Counts

Using only SET and DO instructions for whole algorithms would be impractical as it would require a different program for each size of input, and would be impossible in the case of data-dependent branches such as those used in pivoting algorithms. This technique could, however, be used to optimise relatively large subsections of an algorithm if necessary.

It is clear that an optimising compiler is required, but improved methods of control and other optimisations of the architecture may also be necessary.

To further reduce the time required by control processor instructions, the architecture could be modified in various ways. Firstly, the speed of the control processor could be improved relative to the array cycle time; a doubling of control processor performance should be achievable with currently available microprocessors if integration with the matrix controller were efficient enough. Secondly, a four dimensional read/write address generator would allow a single DO instruction to perform an entire matrix multiplication between two matrices of arbitrarily large size; this would eliminate the looping required to divide matrices into array-sized strips, but would provide no benefit for matrices smaller than the array. Thirdly, smaller improvements may be possible by optimising the way in which the SET and DO instructions control the data path; for example, DO instructions which launch scalar data path instructions only, so that fewer SET instructions would be required to set up the operation.

5.4.10 Conclusions

Gaussian Elimination on the MATRISC Load/Store architecture achieves good performance for medium to large matrices. An algorithm that predominantly uses matrix multiplication is critical to good performance. The optimised Fast algorithm uses a range of different techniques to achieve the best performance across all matrix sizes.

Performance increases with matrix size, with the standard parameters achieving 7200Mflop/s, which is 72% of peak performance for n = 1000, and reaching half of that performance for n = 250. Smaller matrices achieve lower performance; approximately 2000Mflop/s for n = 100 and approximately 1000Mflop/s for n = 50. If partial pivoting is used, the simulated performance of these algorithms is approximately halved. Furthermore, because of the data-dependent branching involved in searching for the pivot element, the simulator is most likely to be over estimating the performance of the pivoting algorithm in practice.

The overhead of the Load/Store memory architecture compared to the hypothetical perfect memory architecture peaks at around 70% for matrices of order 100, but quickly falls to under 20% for order 1000.

Gaussian Elimination is compute-bound for any matrix small enough to be loaded into one of the registers. Matrices that are too large to fit in the registers can be handled if each register is large enough to hold two critical sized matrices.

The time taken to perform the matrix controller instructions was found to be significantly longer than the time taken to perform the computation on the processor array. By counting only those instructions essential to the calculation, it was determined that this problem was largely due to the very poor code produced by the Masm compiler.

5.5 Householder QR Factorisation

QR factorisation of a matrix A involves finding an orthogonal matrix Q and an upper triangular matrix R, such that $A = QR$. This operation has a number of applications, in particular, solving the least squares problem, where the aim is to find a vector x, which minimises $\|Ax - b\|_2$, where $A \in \Re^{m \times n}$ with $m \geq n$ and $b \in \Re^m$. In this section, factorisation is achieved by applying a series of Householder Reflections to A to reduce it to upper triangular form[Golub and Van Loan 96].

As with Gaussian Elimination, there are a number of different ways in which Householder QR Factorization can be implemented. A vector-based solution is described first, followed by a more efficient block version that utilises matrix multiplication.

5.5.1 Vector Householder QR

The algorithm for Householder QR, expressed in Matlab code, is,

```
for i = 1:n
        v = A(i:, i)
        u = v(1) + sign(v(1))*norm(v)
        v(1) = 1
        v(2:) = v(2:)/u
        B = 2/(v' * v)
        w = B * A(i:, i+1:)' * v
        A(i:, i+1:) = A(i:, i+1:) + v * w'
end
```

This algorithm requires approximately $2n^2(m - n/3)$ flop. When it terminates, A

contains the computed R matrix. The matrix Q can be implicitly calculated by storing the vector, v, calculated in each iteration of the loop. The explicit Q matrix can be accumulated in $4(m^2n - mn^2 + n^3/3)$ flop from these vectors, if required[Golub and Van Loan 96].

If backsubstitution is applied to the top left n-1 $\times n$ submatrix of R, the resulting solution vector is the least squares solution of the system of linear equations represented by A. In particular, if $m = n$-1, A is not overdetermined and represents a normal linear system. This is a very numerically stable method for solving a linear system, better that Gaussian Elimination with partial pivoting, but it involves twice as many floating point operations[Golub and Van Loan 96].

5.5.1.1 Implementation

The Householder algorithm is considerably more complicated than Gaussian Elimination, which makes the implementation quite tedious given the limitations of the Masm language. The set of matrix register variables used is shown below in Figure 5.35. The variables R and VW are square, V, V2, V3, W1, W2 are single rows or columns and U, B and B2 are scalars. While it may be possible to use fewer vector and/or scalar variables, the size of these variables is insignificant in comparison to the size of the two square matrix variables, which are necessary.

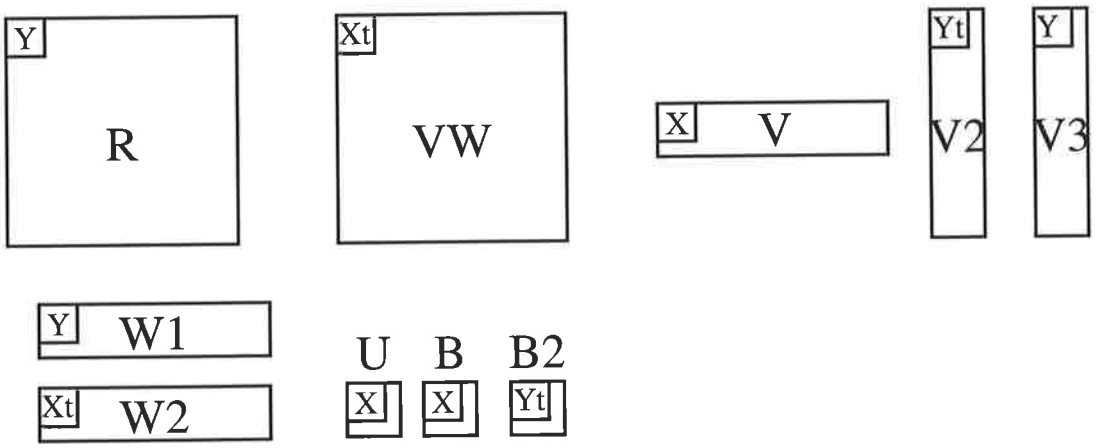


Figure 5.35: Matrix Register Variables for Vector Householder QR Factorisation

The algorithm, expressed in MATRISC compliant Matlab code, is,

```
for i = 1:n
        V = R(i:, i)'
        U = V * R(i:,i)
        U = sqrt U
        B = U * sign(R(i,i))
        B2 = B + R(i,i)
```

```
        V(1, 2:) = V(1,2:)//B2
        V(0,0) = 1
        V2 = V'
        V3 = V'
        B = V * V3
        B2 = 2 // B
        V = V ** B2
        W1 = V * R
        W2 = W1
        VW = V2 * W2
        R(i:, i:) = R(i:, i:) + VW
end
```

As with the block outer product update step in Gaussian Elimination, there are a number of extra copy operations required to get values into the right storage form for the calculation to proceed.

To test the correctness of the Masm implementation, a backsubstitution step was added and the algorithm used to solve a linear system. The solutions were found to be correct in all cases shown later in performance graphs.

5.5.1.2 Theoretical Results

The following numerical sum formula for the number of array cycles required was derived from the algorithm above, but it is too complex to permit any meaningful simplification.

$$\text{HouseRowCycles}(n, m, p) = \sum_{i=0}^{\min(n, m)} \left\lceil \begin{array}{l} 3\text{AC}(1, n-i, i \bmod p, p) + \\ 2\text{MC}(1, n-i, 1, i \bmod p, i \bmod p, p, 1) + \\ \text{AC}(1, n-i-1, i \bmod p, p) + \\ \text{MCC}(1, n-i, i \bmod p, p) + \\ \text{MC}(1, n-i, m-i, i \bmod p, i \bmod p, p, 1) + \\ \text{AC}(m-i, 1, i \bmod p, p) + \\ \text{MC}(n-i, 1, m-i, 0, i \bmod p, p, 1) + \\ \text{AC}(m-i, n-i, i \bmod p, p) + 5 \end{array} \right\rceil$$

The formula for the perfect memory architecture is,

$$\text{HouseRowCyclesP}(n, m, p) = \sum_{i=0}^{\min(n, m)} \left\lceil \begin{array}{l} 2\text{AC}(1, n-i, i \bmod p, p) + \\ 2\text{MC}(1, n-i, 1, i \bmod p, i \bmod p, p, 1) + \\ \text{AC}(1, n-i-1, i \bmod p, p) + \\ \text{MC}(1, n-i, m-i, i \bmod p, i \bmod p, p, 1) + \\ \text{MC}(n-i, 1, m-i, 0, i \bmod p, p, 1) + \\ \text{AC}(m-i, n-i, i \bmod p, p) + 5 \end{array} \right\rceil$$

5.5.1.3 Simulation Results

The algorithm was simulated for a range of square matrices with the standard parameters and the results are shown in Figure 5.36. A required flop count of $2n^2(m - n/3)$ was used.

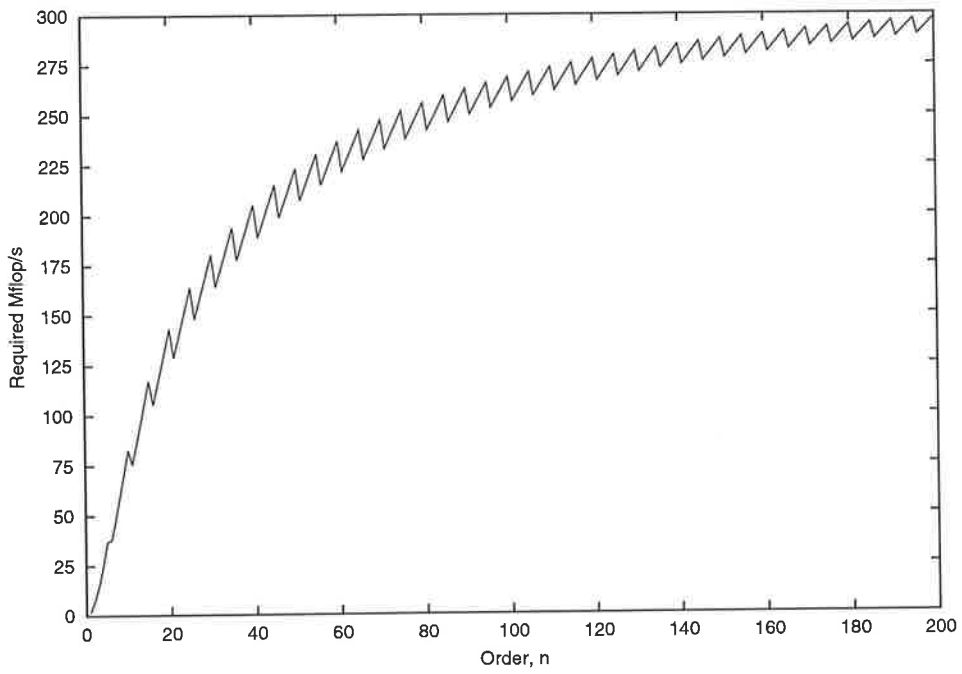


Figure 5.36: Performance of Vector Householder QR

As this algorithm uses a significant amount of matrix-vector multiplication, which performs at twice the addition speed of the architecture, the overall performance rises above the addition speed of 250Mflop/s. The bulk of the floating point operations occur in two places; the matrix-vector multiplication of $w = Av$, and outer product update of $A = A + vw^T$. Both of these operations require the same number of flop, but because the first runs twice as quickly as the second, the maximum performance possible is 333Mflop/s. However, the algorithm also uses a significant amount of scalar and dot product operations that reduce performance, particularly for small matrix orders. The algorithm reaches half its peak performance for a matrix of order approximately 28, compared to order 12 for Gaussian Elimination using row operations.

5.5.2 Block Householder QR

As with Gaussian Elimination, it is possible to develop a block version of Householder QR that is rich in matrix multiplication. The block algorithm proceeds by applying the vector algorithm to a narrow block of columns, while building up a block representation of

the transformations applied in two matrices, *W* and *Y*. When the block is complete the block transformation is applied to the rest of the matrix and the algorithm proceeds onto the next block. The algorithm, expressed in Matlab code, is,

```
for i = 1:b:n
        be = min(i + b - 1, n)
        for bi = i : be
                v = A(bi:, bi)
                u = v(1) + sign(v(1))*norm(v)
                v(1) = 1
                v(2:) = v(2:)/u
                B = v' * v
                w = B * A(bi:, bi+1:be) * v
                A(bi:, bi+1:be) = A(bi:, bi+1:be) + v * w'
                z = -B * (v + W * Y' * v)
                W = [W z]
                Y = [Y v]
        end
        A(i:, be+1:) = A(i:, be+1:) + Y * (W' * A(i:, be+1:))
end
```

5.5.2.1 Implementation

The block algorithm uses the same matrix register variables as the vector version, plus those shown in Figure 5.37. The size of these variables is the block size in one direction and the size of the matrix being reduced in the other.

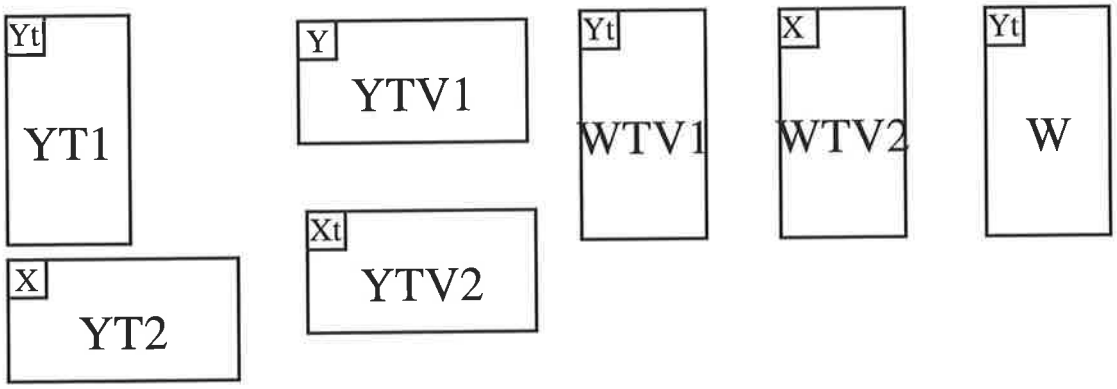


Figure 5.37: Extra Matrix Register Variables for Block Householder QR Factorisation

The algorithm, expressed in MATRISC compliant Matlab code, is,

```
for i = 1:n
        be = i + b -1
        for bi = i : be
                V = R(i:, i)'
                U = V * R(i:,i)
                U = sqrt (U)
                B = U * sign(R(i,i))
                B2 = B + R(i,i)
```

```
            V(1, 2:) = V(1,2:)//B2
            V(0,0) = 1
            V2 = V'
            V3 = V'
            B = V * V3
            B2 = 2 // B
            V = V ** B2
            W1 = V * R
            W2 = W1
            VW = V2 * W2
            R(i:, i:be) = R(i:, i:be) + VW
            YT2 = YT1'
            YTV1 = YT2 * V3
            YTV2 = YTV1
            WYTV1 = W * YTV2
            WYTV2 = WYTV1
            WYTV2 = V2 + WYTV2
            WYTV2 = WYTV2 ** B2
            W = [W WYTV2]
            YT1 = [YT1 V2]
        end
        YT2 = W
        YTV1 = YT2 * R(i:, be+1:)
        YTV2 = YTV1
        VW = W * YTV2
        R(i:, be+1:) = R(I:, be+1:) + VW
end
```

5.5.2.2 Theoretical Results

The following numerical sum formula for the number of array cycles required was developed for this algorithm. Again, it is too complex to permit meaningful simplification. The equation is divided into three parts for simplicity.

$$\text{HouseBlockCycles}(n, m, p, v, b) = \sum_{i=0}^{\left\lceil \frac{\min(n, m)}{b} \right\rceil} \begin{bmatrix} \text{BlockReduce}(n, bi, \min(bi + b, n, m) - 1, \\ \min(bi + b, m), p) + \\ \text{MCC}(n - bi, b, 0, p) + \\ \text{MC}(b, n - bi, m - bi + b, 0, 0, p, 1) + \\ \text{AC}(m - bi + b, b, 0, p) + \\ \text{MC}(m - bi + b, b, n - bi, 0, 0, p, 1) + \\ \text{AC}(m - bi + b, n - bi, 0, p) \end{bmatrix}$$

$$\text{BlockReduce}(n, i_1, i_2, m', p) = \sum_{i=i_1}^{i_2} \left\lceil \begin{array}{l} 3\text{AC}(1, n-i, i \bmod p, p) + \\ 2\text{MC}(1, n-i, 1, i \bmod p, i \bmod p, p, 1) + \\ \text{AC}(1, n-i-1, i \bmod p, p) + \\ \text{MCC}(1, n-i+i \bmod p, i \bmod p, p) + \\ \text{MC}(1, n-i, m'-i, i \bmod p, i \bmod p, p, 1) + \\ \text{AC}(m'-i, 1, i \bmod p, p) + \\ \text{MC}(n-i, 1, m'-i, 0, i \bmod p, p, 1) + \\ \text{AC}(m'-i, n-i, i \bmod p, p) + 5 + \\ \text{UpdateRep}(n, i, i_1, p) + \\ \text{AC}(n-i_1, 1, 1, p) + \\ \text{AC}(n-i, 1, i \bmod p, p) \end{array} \right\rceil$$

$$\text{UpdateRep}(n, i, i_1, p) = \begin{cases} 2\text{AC}(1, n-i, i \bmod p, p) & \text{when } i = i_1 \\ \text{MCC}(n-i, i-i_1, i \bmod p, p) + & \text{otherwise} \\ \text{MC}(i-i_1, n-i, 1, 0, i \bmod p, p, 1) + \\ \text{AC}(1, i-i_1, i \bmod p, p) + \\ \text{MC}(n-i, i-i_1, 1, 0, i \bmod p, p, 1) + \\ 2\text{AC}(n-i_1, 1, 0, p) + \\ \text{AC}(n-i, 1, i \bmod p, p) \end{cases}$$

The formula for the perfect memory architecture is given below. It was used to produce the perfect architecture results in §5.5.3.

$$\text{HouseBlockCyclesP}(n, m, p, v, b) = \sum_{i=0}^{\left\lceil \frac{\min(n,m)}{b} \right\rceil} \left\lceil \begin{array}{l} \text{BlockReduceP}(n, bi, \min(bi+b, n, m) - 1, \\ \min(bi+b, m), p) + \\ \text{MC}(b, n-bi, m-bi+b, 0, 0, p, 1) + \\ \text{MC}(m-bi+b, b, n-bi, 0, 0, p, 1) + \\ \text{AC}(m-bi+b, n-bi, 0, p) \end{array} \right\rceil$$

$$\text{BlockReduceP}(n, i_1, i_2, m', p) = \sum_{i=i_1}^{i_2} \left\lceil \begin{array}{l} 2\text{AC}(1, n-i, i \bmod p, p) + \\ 2\text{MC}(1, n-i, 1, i \bmod p, i \bmod p, p, 1) + \\ \text{AC}(1, n-i-1, i \bmod p, p) + \\ \text{MC}(1, n-i, m'-i, i \bmod p, i \bmod p, p, 1) + \\ \text{MC}(n-i, 1, m'-i, 0, i \bmod p, p, 1) + \\ \text{AC}(m'-i, n-i, i \bmod p, p) + 5 + \\ \text{UpdateRepP}(n, i, i_1, p) + \\ \text{AC}(n-i, 1, i \bmod p, p) \end{array} \right\rceil$$

$$\text{UpdateRepP}(n, i, i_1, p) = \begin{cases} \text{AC}(1, n-i, i \bmod p, p) & \text{when } i = i_1 \\ \text{MC}(i-i_1, n-i, 1, 0, i \bmod p, p, 1) + & \text{otherwise} \\ \text{MC}(n-i, i-i_1, 1, 0, i \bmod p, p, 1) + \\ \text{AC}(n-i_1, 1, 0, p) + \\ \text{AC}(n-i, 1, i \bmod p, p) \end{cases}$$

5.5.2.3 Simulation Results

The results for the block algorithm are shown in Figure 5.38 and Figure 5.39, for reduction of a square matrix for the standard parameters. The block size used is equal to the virtual array size, that is, $b = vp$.

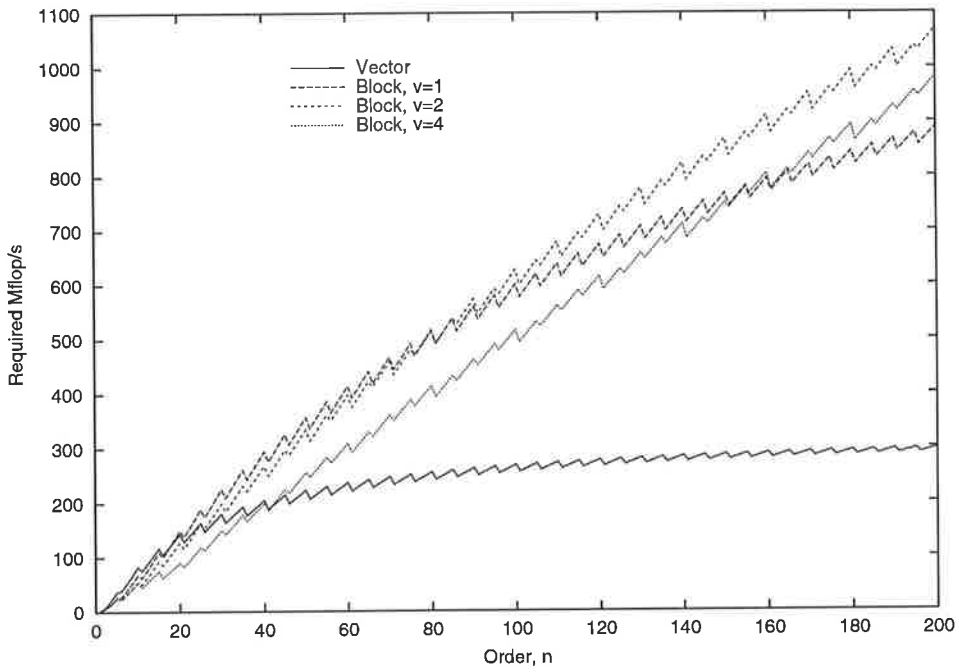


Figure 5.38: Performance of Block Householder QR

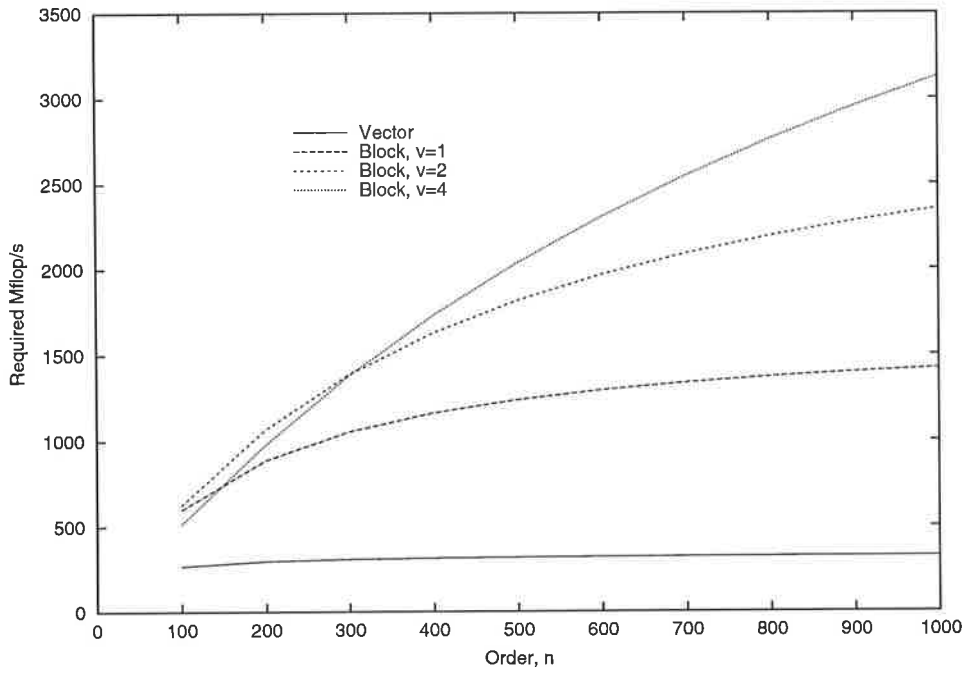


Figure 5.39: Performance of Block Householder QR - Large Matrix Sizes

The results show that the block algorithm achieves considerable performance gains over the vector algorithm. However, the shape of the graphs shows that the algorithm does not approach its peak performance, even matrices of order 1000.

A theoretical estimate of the maximum performance can be obtained by noting that the bulk of the computation is performed in the block update step. This step consists of two parts, a matrix multiplication and a block outer product update, both of which require the same number of floating point operations. As was discussed for Gaussian Elimination in §5.4.3.2, the speed of the block outer product update is limited to slightly more than half the architecture's multiplication speed, because the block size is equal to the virtual array size. The limiting speeds for Householder QR for the standard parameters with $v = 1$, 2 and 4 are 1750Mflop/s, 3417Mflop/s and 6750Mflop/s respectively.

In the case of Gaussian Elimination, the limitation of the block outer product step was overcome by the Fast algorithm, which used a larger block size where appropriate. However, the success of using a larger block size relied on recursively using a smaller block size to process each block. Otherwise, it would have been necessary to reduce the block with vector and scalar operations, the time for which would have eroded any gain from using a larger block size. The results of using a number of different block sizes for the Householder algorithm are shown in Figure 5.40.

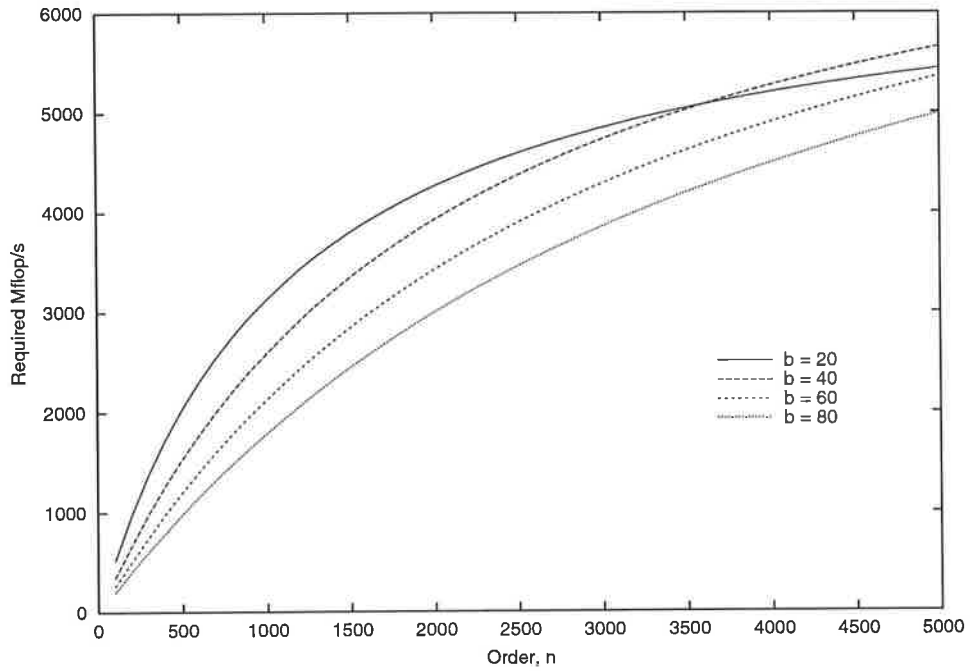


Figure 5.40: Performance of Block Householder QR with Larger Block Sizes

These results show that larger block sizes only outperform smaller ones for very large matrix sizes. Unfortunately, there is no known way to make a recursive algorithm with the Householder QR Factorisation.

The performance of the block Householder algorithm for a number of different array sizes and maximum virtual factors is shown in Figure 5.41.

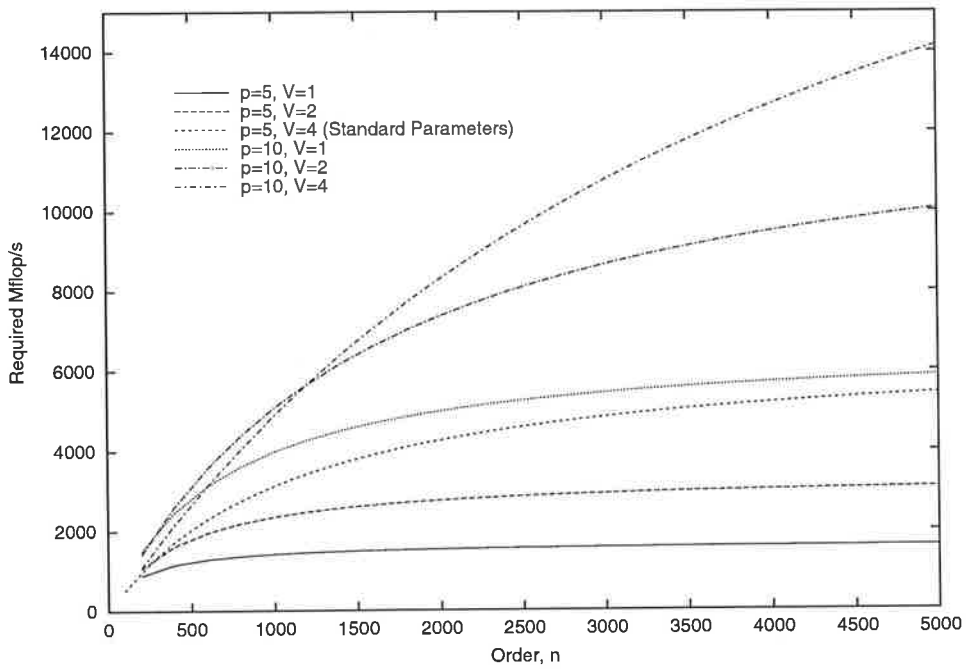


Figure 5.41: Performance of Householder QR for Different Array Configurations

The same results expressed as percentage of peak performance for the particular architectural configuration is shown in Figure 5.42.

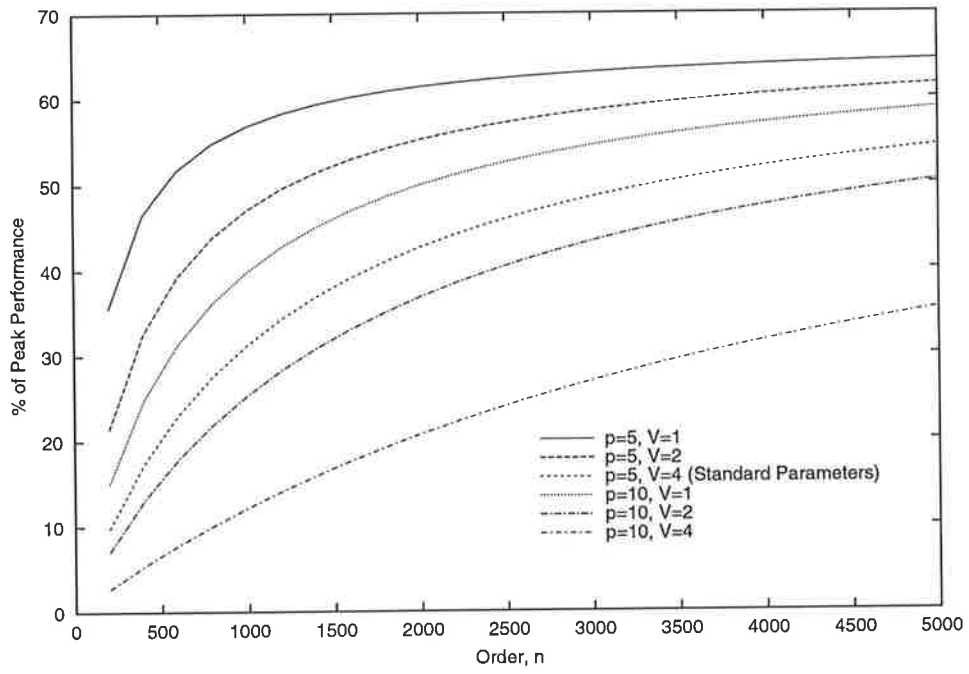


Figure 5.42: Performance of Householder QR for Different Array Configurations as a Percentage of Peak Performance

The best performances are significantly less than for Gaussian Elimination, and require

much larger matrices to achieve them. For Gaussian Elimination, the Fast algorithm achieved 72% of peak performance for the standard parameters, and achieved half of this performance for matrix order ~250. Householder QR factorisation achieves only 54% of peak performance, and reaches half of this performance for matrix order 700.

Note that there is no significant correlation between the results for systems with the same virtual array size, as there was for Gaussian Elimination, because the bulk of the non-multiplication overhead in the Householder QR algorithm is due to dot products. As dot products only perform 2 floating point operations per array cycle, they achieve only $1/pv^2 \times 100\%$ of peak, which depends on virtual factor as well as virtual array size.

5.5.3 Load/Store Memory Architecture Overhead

The overhead of the Load/Store memory architecture compared with the hypothetical perfect architecture for Householder QR factorisation for various architectural parameters is shown in Figure 5.43 and Figure 5.44.

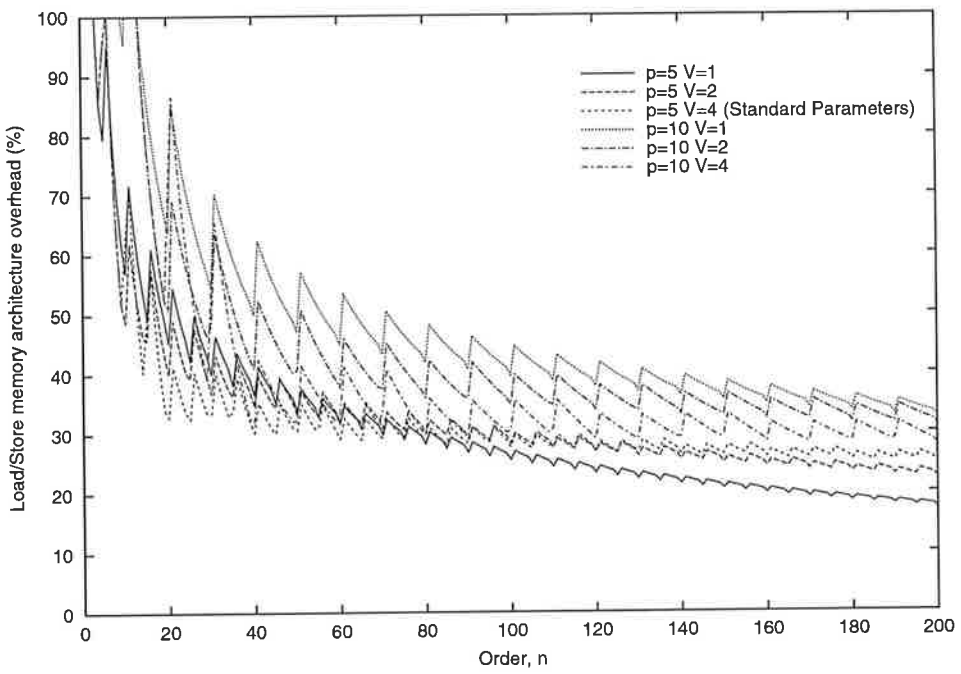


Figure 5.43: Overhead of the Load/Store Architecture versus the Perfect Memory Architecture for Householder QR

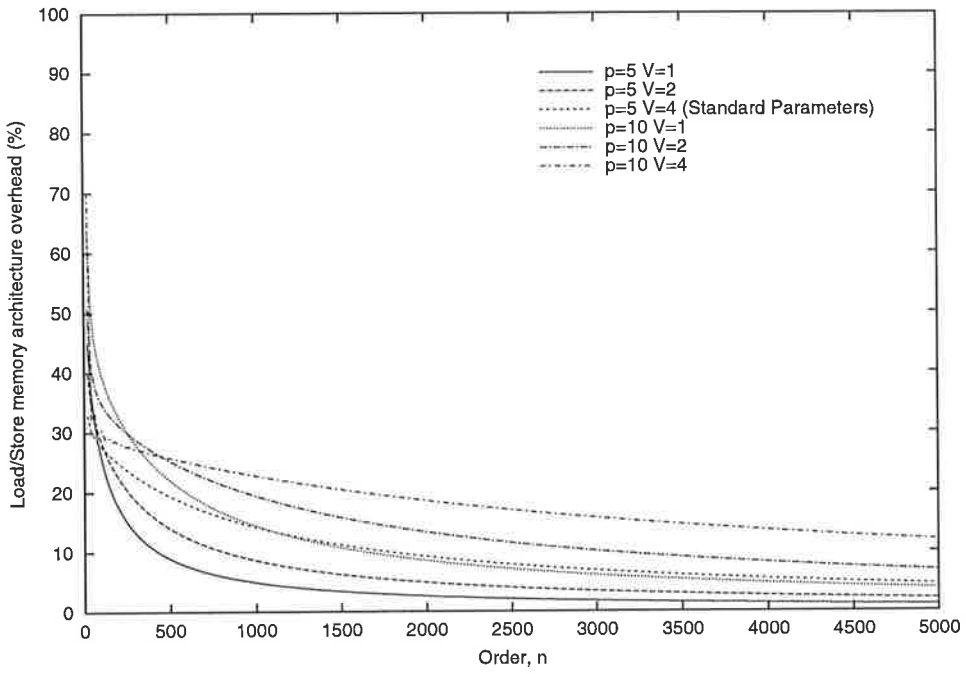


Figure 5.44: Overhead of the Load/Store Architecture versus the Perfect Memory Architecture for Householder QR - Large Matrix Sizes

The overheads of the Load/Store architecture are less than for Gaussian Elimination, due in part to the use of dot products in the Householder algorithm, which perform equally well on both memory architectures. The overhead for the standard parameters for Gaussian Elimination was 70% for small matrices, falling to under 20% for order 1000, but for Householder QR factorisation, the overhead is 50%, falling to under 15% for order 1000.

5.5.4 Load/Store Transfer Time

The time required to perform load/store transfers as a percentage of the time required for computation is shown in Figure 5.45 and Figure 5.46. These figures assume that the algorithm is being used to solve a linear system, and thus that the load/store transfer consists of loading the matrix to be factored and storing the vector of the result. If the Q and R factor matrices are required, approximately three times as much load/store transfer time will be necessary; however, this is countered by the fact that explicit formation of the Q matrix requires approximately twice as much computation, so the load/store time as a percentage will be approximately 50% more.

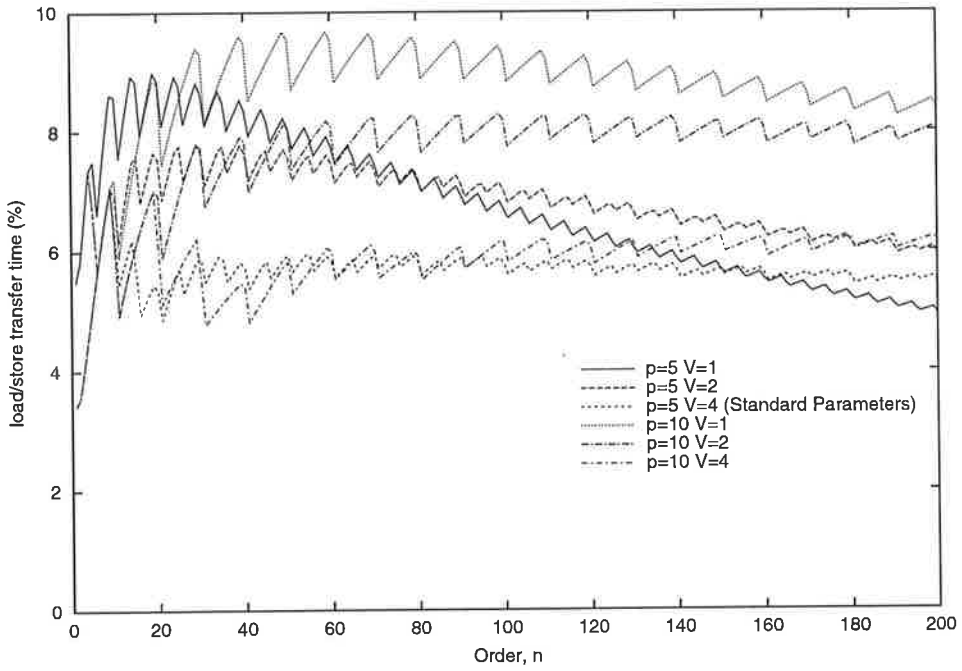


Figure 5.45: Load/Store Transfer Time for Householder QR

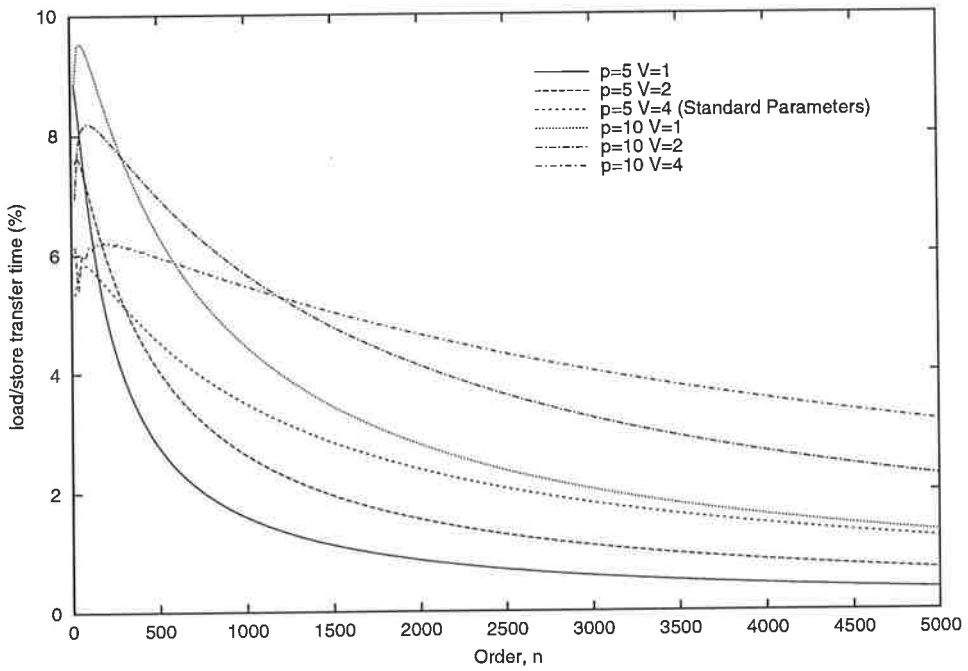


Figure 5.46: Load/Store Transfer Time for Householder QR - Large Matrix Sizes

The relative load/store transfer time is much lower than for Gaussian Elimination, because QR factorisation requires more floating point operations for every data value loaded, and because the performance of this algorithm is not as high as for Gaussian Elimination.

5.5.5 Conclusions

Householder QR Factorisation achieves good performance on the Load/Store MATRISC architecture. As with Gaussian Elimination, the key to an efficient implementation is using a block matrix algorithm that is rich in matrix multiplication. However, the Householder algorithm is more complex than Gaussian Elimination, and there are significantly more overheads in the form of sections of the algorithms that do not convert to matrix multiplication. The result of these overheads is that the algorithm requires larger matrix sizes to achieve its best performance. When both algorithms use a fixed block size, the Householder algorithm achieves higher performance than Gaussian Elimination because the main block update step uses more matrix multiplication. However there is no Householder QR counterpart to the Fast algorithm for Gaussian Elimination, which intelligently uses a range of block sizes, so the best performance achieved by Householder QR is less than that of Gaussian Elimination. For the standard parameters, the best performance is 5400Mflop/s or 54% of peak performance, with half of this performance being achieved for matrix order 700.

The performance penalty of the Load/Store memory architecture compared to the hypothetical perfect memory architecture is less than 50% for matrices above order 50. The time required for load/store transfers is 10%, or less, of that required for computation, which implies that the algorithm is compute-bound for all cases.

Thus Householder QR Factorisation is a feasible algorithm for the Load/Store MATRISC architecture. Although it is slower than Gaussian Elimination, its use for solution of linear systems would be appropriate when stability is an issue, because it is a more stable algorithm than Gaussian Elimination with pivoting. As the Householder QR algorithm uses twice as many floating point operations as are required for linear system solution its performance, measured in required floating point operations, would be halved. So it would achieve a maximum of 2700Mflop/s. This means that, given the problems with the data dependent branching in the pivoting algorithm, Householder QR achieves a comparable speed to Gaussian Elimination with pivoting.

5.6 Fourier Transform

The Fourier Transform is a fundamental operation in signal processing. The discrete

fourier transform (DFT) takes as a real or complex vector of length N, and transforms it into a complex vector of length N. Here, the most general case of a complex input and complex result will be considered. The DFT of an input vector $x(n)$ to a result vector $X(k)$ is defined as,

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn} \qquad \text{for} \qquad k = 0 \ldots N-1$$
$$\text{where} \qquad W_N^{kn} = e^{-2\pi kn/N} \tag{5.25}$$

This equation can also be expressed as a matrix-vector product as follows,

$$X = Wx \qquad \text{where} \qquad W = [w_{ij}] \;\; w_{ij} = W_N^{ij} = e^{-2\pi ij/N}$$

Using this definition to compute the DFT requires $6N^2$ real floating point operations. There are many ways to express the transform that can reduce the amount of computation required. The most common is a family of algorithms loosely known as the Fast Fourier Transform, FFT, which require $5N\log_2 N$ real floating point operations but are only applicable to the case where the vector length is a binary power.

As the FFT does not uses matrix multiplication, different algorithms are used by the MATRISC processor to perform the Fourier transform. Two algorithms, the prime factor mapped Fourier transform and the common factor mapped Fourier transform, which are both rich in matrix multiplication, are described in the following sections.

5.6.1 Prime Factor Mapped Fourier Transform

The prime factor mapped Fourier Transform algorithm is particularly well suited to the MATRISC architecture because of its extensive use of matrix multiplication. The algorithm involves mapping the linear DFT to a higher order structure where calculation of the transform becomes a series of matrix-matrix multiplications. This derivation of the algorithm principally follows Marwood[Marwood 94], but introduces a new notation to more compactly express the transform with higher order mapping.

Firstly, an arbitrary multidimensional mapping is described, beginning with the discrete Fourier transform of a vector $x(n)$ to a vector $X(k)$ as shown in (5.25). For a q-dimensional mapping, the vector length N is divided into q factors, N_i, such that $N = N_1 N_2 \ldots N_q$. These factors are used to define two mappings, one for the input vector and one for the result.

$$n = \langle \sum_{i=1}^{q} M_i n_i \rangle_N \qquad \text{where} \qquad n_i = 0 \ldots N_i - 1$$

$$k = \langle \sum_{j=1}^{q} L_j k_j \rangle_N \qquad \text{where} \qquad k_j = 0 \ldots N_j - 1$$

M_i and L_j are constants that must be chosen so as to make the mappings unique. Applying these mappings to (5.25) gives,

$$Y(k_1, k_2, \ldots, k_q) = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} \cdots \sum_{n_q=0}^{N_q-1} y(n_1, n_2, \ldots, n_q) \prod_{i=1}^{q} \prod_{j=1}^{q} W_N^{M_i L_j n_i k_j}$$

$$\text{where} \qquad y(n_1, n_2, \ldots, n_q) = x(\langle \sum_{i=1}^{q} M_i n_i \rangle_N) \qquad (5.26)$$

$$Y(k_1, k_2, \ldots, k_q) = X(\langle \sum_{j=1}^{q} L_j k_j \rangle_N)$$

Y and y are q-dimensional tensors used to express the algorithm more succinctly. Note that if the product $M_i L_j$ contains a factor of N, the corresponding W factor is unity and can be ignored. By choosing the M_i and L_j mapping coefficients appropriately, many of the W factors can be removed and the whole equation simplified greatly.

The prime factor algorithm applies in the situation where N_1 to N_q are all relatively prime. In this case, the following mapping coefficients may be used.

$$M_i = \frac{N}{N_i}$$

$$L_j = \frac{N}{N_j}$$

Using these mapping coefficients results in the cancellation of all W factors except for the case $i = j$. The transformation thus reduces to,

$$Y(k_1, k_2, \ldots, k_q) = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} \cdots \sum_{n_q=0}^{N_q-1} y(n_1, n_2, \ldots, n_q) W_N^{M_1 L_1 n_1 k_1} W_N^{M_2 L_2 n_2 k_2} \ldots W_N^{M_q L_q n_q k_q} \qquad (5.27)$$

To express this equation in matrix terms, the matrices W_i with dimensions $N_i \times N_i$ are defined as,

$$W_i = [w_{jk}] \qquad w_{jk} = e^{\frac{-2\pi ijkM_iL_i}{N}} \tag{5.28}$$

A family of products between a matrix and a q-dimensional tensor are defined below, where functional notation is use to refer to tensor and matrix elements.

$$C = (W \times Y)_i \Rightarrow C(n_1, n_2, \ldots, n_i, \ldots, n_q) = \sum_{i=0}^{N_i} W(n_i, i)Y(n_1, n_2, \ldots, i, \ldots, n_q) \tag{5.29}$$

$$\text{where} \qquad C, Y \in \Re^{N_1 \times N_2 \times \ldots \times N_i \times \ldots \times N_q} \qquad W \in \Re^{N_i \times N_i}$$

These products can be expressed as a sequence of matrix operations quite easily, by extending the Matlab colon to tensors.

$$C = (W \times Y)_i \Rightarrow C(n_1, n_2, \ldots, \underbrace{:}_{\text{index } i}, \underbrace{:}_{\text{index } \langle i \rangle_N+1}, \ldots, n_q) = WY(n_1, n_2, \ldots, :, :, \ldots, n_q) \quad \text{for} \quad n_j = 1 \ldots N_j \quad j = 1 \ldots q,\ j \neq i,\ j \neq \langle i \rangle_N + 1 \tag{5.30}$$

Equation (5.30) states that the product can be formed by repeated matrix multiplication operations on two dimensional 'slices' of the tensor. Note that the second colon index does not need to be $\langle i \rangle_N + 1$. It can be any other value between 1 and N except i. When implementing this equation on the MATRISC architecture, there is some advantage to be gained by choosing it to correspond to an N_x factor that is large and close to a multiple of the virtual array size.

Using this notation, and the W_i defined in (5.28), equation (5.27) for the prime factor mapped Fourier transform can be compactly expressed as,

$$Y = (W_1 \times (W_2 \times (\ldots \times (W_q \times y)_q \cdots)_{\ldots})_2)_1$$

5.6.1.1 Theoretical Results

The expression for the number of array cycles taken by the prime factor algorithm is,

$$\text{PMFTCycles}(N_1, N_2, \ldots, N_q, p, v) = \sum_{i=1}^{q} 4\text{MC}(N_i, N_i, N_{\langle i \rangle_N+1}, 0, 0, p, v)\frac{N}{N_i N_{\langle i \rangle_N+1}} \tag{5.31}$$

The factor of 4 is due to the fact that complex multiplication must be used. Equation (5.31) is used to calculate precise performance figures in §5.6.1.2. Using the continuous approximation for the number of array cycles required for the multiplication gives,

$$\text{PMFTCyclesA}(N_1, N_2, \ldots, N_q, p, v) = \sum_{i=1}^{q} 4\frac{N_i N_i N_{\langle i \rangle_N + 1}}{vp^2}\frac{N}{N_i N_{\langle i \rangle_N + 1}} = \frac{4N}{vp^2}\sum_{i=1}^{q} N_i$$

If the N_i could be chosen arbitrarily from the real numbers, which they most certainly cannot, the number of cycles required would be minimised by using $N_1 = N_2 = \ldots = N_q = \sqrt[q]{N}$. Although this simplification is impossible, it provides a very simple expression that sets an upper bound on performance.

$$\text{PMFTCyclesAUB}(N, q, p, v) = \frac{4qN\sqrt[q]{N}}{vp^2}$$

One factor that this equation makes clear is that the prime factor algorithm is $O(N^{1+1/q})$, which implies that asymptotically it cannot compete with the Fast Fourier Transform for any fixed value of q.

5.6.1.2 Simulation Results

The prime factor algorithm was implemented in Masm and run on the MATRISC simulator for several different set of factors. For all test cases performed, the results produced were correct and the number of array cycles required agreed with the theoretical above.

The actual implementation in Masm was complicated by 2 factors. First, complex arithmetic was required, which necessitated the addition of another template to the Masm compiler to perform complex multiplication. Second, the Masm language has no facility for manipulating 3- and 4-dimensional tensor structures. This difficulty was solved by directly manipulating the matrix description variables, that is, the base, stride, offset, width and depth. This was relatively straightforward and the complete Masm code for the three dimensional case is included in Appendix D.

The matrix register variables required for the algorithm are shown in Figure 5.47.

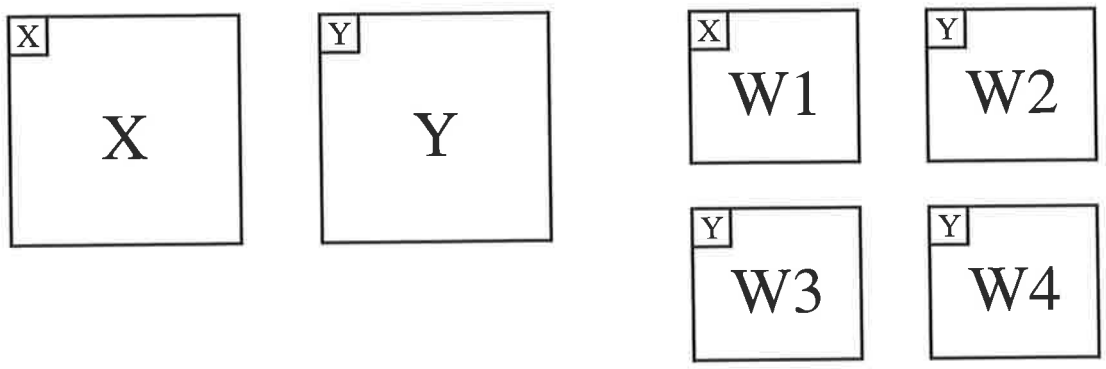


Figure 5.47: Matrix Register Variables for Prime Factor Mapped Fourier Transform

Two large matrices, one in X storage and one in Y storage, are required to hold the mapped vector as it is transformed. For three- and four-dimensional mappings this register space actually holds a tensor structure. Depending on the order of the mapping used space is also required for up to four W matrices. The size of these weight matrices depends on the vector length and how it is factorised, but they generally require less space than the vector being transformed for three- and four-dimensional mappings. Recall that all these matrices are complex and so require twice the space of a real matrix.

The graphs of the simulation results appear below with those for the common factor algorithm in §5.6.2.2.

5.6.2 Common Factor Fourier Transform

Equation (5.26) for a multidimensional Fourier Transform can also be expressed in matrix terms for the case where the N_i have a common factor. In this case, the mapping used in the prime factor case is no longer unique, so the mapping below is used instead.

$$M_i = \prod_{k=1}^{i-1} N_k \qquad \text{for } i = 2 \ldots q \qquad \text{and} \qquad M_1 = 1$$

$$L_j = \prod_{k=1}^{q-i} N_{q-k+1} \qquad \text{for } j = 1 \ldots q-1 \qquad \text{and} \qquad L_q = 1$$

These mapping coefficients cancel all W terms for which $j > i$. To express the transformation in matrix terms, a further set of W matrices, with dimensions $N_i \times N_j$ are defined as,

$$W_{ij} = [w_{kl}] \qquad w_{kl} = e^{\frac{-2\pi i k l M_i L_j}{N}}$$

A family of elementwise products between a matrix and a q-dimensional tensor is defined as,

$$C = (W \otimes Y)_{ij} \Rightarrow$$

$$C(n_1, n_2, \ldots, n_i, \ldots, n_j, \ldots, n_q) = W(n_i, n_j)Y(n_1, n_2, \ldots, n_i, \ldots, n_j, \ldots, n_q)$$

$$\text{where} \qquad C, Y \in \Re^{N_1 \times N_2 \times \ldots \times N_i \times \ldots \times N_q} \qquad W \in \Re^{N_i \times N_j}$$

Again these operations can be described simply as a sequence of matrix operations.

$$C = (W \otimes Y)_{ij} \Rightarrow C(n_1, n_2, \ldots, \underbrace{:}_{\text{index } i}, \ldots, \underbrace{:}_{\text{index } j}, \ldots, n_q) = W \otimes Y(n_1, n_2, \ldots, :, \ldots, :, \ldots, n_q)$$

$$\text{for} \qquad n_k = 1 \ldots N_k \qquad k = 1 \ldots q, k \neq i, k \neq j$$

The common factor algorithm can now be expressed using the new W matrices and elementwise products as,.

$$\begin{aligned}
T_1 &= (W_1 \times y)_1 \\
T_2 &= (W_2 \times (W_{21} \otimes T_1)_{21})_2 \\
T_3 &= (W_3 \times (W_{31} \otimes (W_{32} \otimes T_2)_{32})_{31})_3 \\
&\ldots \\
Y &= (W_q \times (W_{q1} \otimes (W_{q2} \otimes (\ldots \otimes (W_{q\,q-1} \otimes T_{q-1})_{q\,q-1})_{\ldots})_{q2})_{q1})_q
\end{aligned}$$

Note that the intermediate T_i tensors are only used to simplify the presentation of the algorithm.

5.6.2.1 Theoretical Results

The expression for the number of array cycles required by the common factor algorithm is,

$$\text{CMFTCycles}(N_1, N_2, \ldots, N_q, p, v) = \sum_{i=1}^{q} 4\text{MC}(N_i, N_i, N_{\langle i \rangle_N + 1}, 0, 0, p, v)\frac{N}{N_i N_{\langle i \rangle_N + 1}} + \sum_{i=1}^{q}\sum_{j=1}^{i-1} 4\text{AC}(N_i, N_j, 0, p)\frac{N}{N_i N_j}$$

This equation is used to calculate precise performance figures in §5.6.2.2. Using the continuous approximation for the number of array cycles required for the multiplication gives,

$$\text{CMFTCyclesA}(N_1, \ldots, N_q, p, v) = \sum_{i=1}^{q} \sum_{j=1}^{i-1} 4\frac{N_i N_j}{p}\frac{N}{N_i N_j} + \sum_{i=1}^{q} 4\frac{N_i N_i N_{\langle i \rangle_N + 1}}{vp^2}\frac{N}{N_i N_{\langle i \rangle_N + 1}}$$

$$= \frac{4q(q-1)N}{p} + \frac{4N}{vp^2}\sum_{i=1}^{q} N_i$$

A simple upper bound on performance can again be found by setting $N_1 = N_2 = \ldots = N_q = \sqrt[q]{N}$.

$$\text{CMFTCyclesAUB}(N, q, p, v) = \frac{[\sqrt[q]{N} + vp(q-1)]4qN}{vp^2}$$

The common factor mapped algorithm is $O(N^{1+1/q})$, the same as the prime factor algorithm, but it involves a considerable overhead, in the form of extra elementwise operations, with respect to the prime factor algorithm. In practice, the performance of both algorithms for a particular vector length, N, is largely determined by choosing N_i factors which are both, as close to equal as possible, and close to multiples of the architecture's virtual size. For this reason, it is possible that the common factor algorithm can be faster than the prime factor algorithm for a particular vector length despite the overhead of the extra computation, because of the greater flexibility it allows when choosing the N_i factors.

5.6.2.2 Simulation Results

In all the simulation results presented below were produced as follows. First, all possible factorisations, up to dimension 4, for a particular vector length were generated. The performance of each different factorisation was then calculated using the prime factor or common factor algorithm as required by the set of factors. For each matrix-tensor product, each possible choice for the second dimension of the 'slice' was tested and the best selected. The performance for that vector length was chosen to be the best performance of any factorisation.

The required flop count for a vector of length N was set as $5N\log_2 N$, which is the number of flop required by the Fast Fourier Transform. This is possibly an unfair way to measure the performance because the FFT only applies to vector lengths of the form 2^N; however, it provides a consistent method to judge performance without wading into the complexities of Fourier Transforms for arbitrary vector lengths. This method is reasonable in the situation where there is considerable flexibility in choosing the vector length.

The performance with the standard parameters for vector lengths up to 100 is shown in

Figure 5.48. These vector lengths are too short to have many applications, but the graph clearly shows the random dependence of the performance on vector length. This random nature is due to the fact that the performance of each vector length depends strongly on how that length divides into factors, which is essentially random.

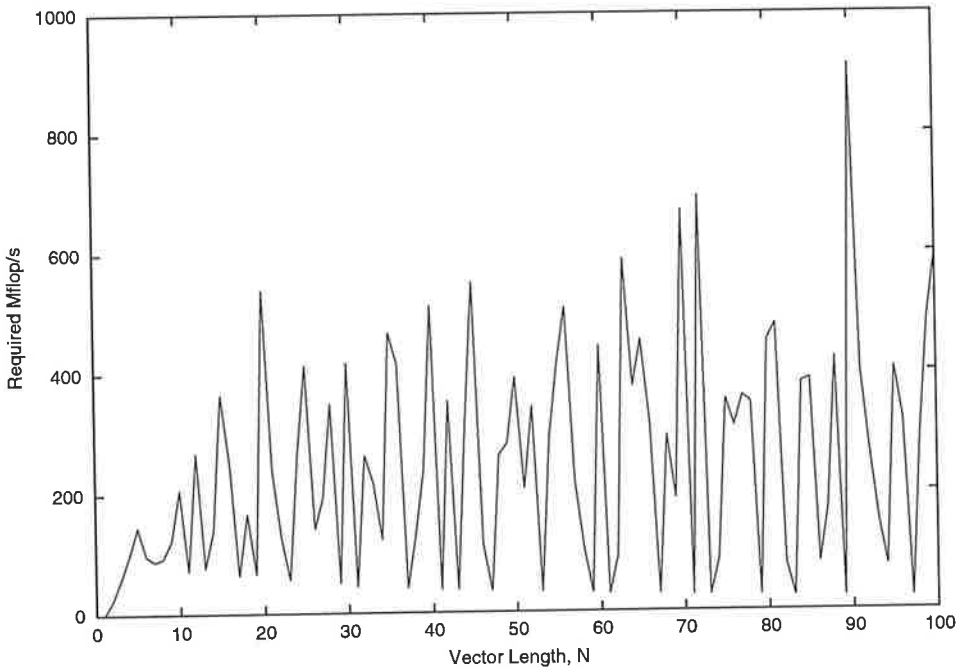


Figure 5.48: Performance of the Fourier Transform

Performance for vector lengths up to 10000 is shown in Figure 5.49. This graph was computed by plotting the best performance between 100 and 199, then between 200 and 299, etc., and so the graph shows the best performance that can be obtained by a single vector within a group of 100 vector lengths.

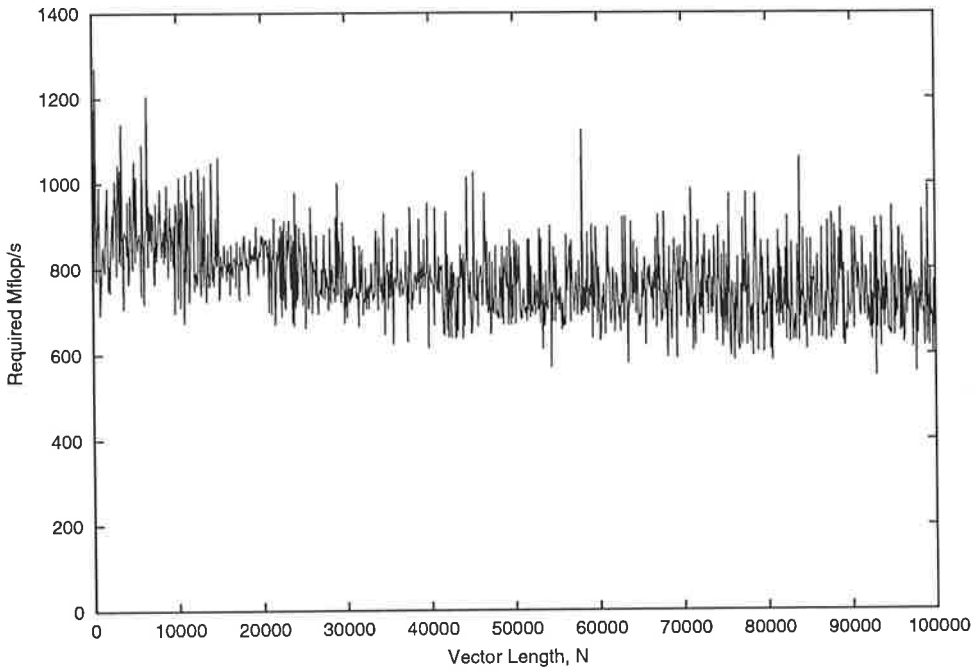


Figure 5.49: Performance of the Fourier Transform - Long Vectors

The performance graph is quite different to those for the previous algorithms. The performance for short vectors is best, around 900Mflop/s, but falls away gently for longer length, down to around 700Mflop/s by length 100000. The good performance for short vector lengths is due to the fact that the prime factor algorithm uses only matrix multiplication. The reason for the decline in performance is that, as the vector length grows, the number of floating point operations required by the prime factor algorithm grows faster than the number required by the FFT, which is used to set the required floating point count for the calculation of performance.

The order of mapping for the best performance for each range of vector lengths is shown in Figure 5.50. Note that each point plotted again represents the best performance from a range of 100 vector lengths, and that given this method assessing performance, the prime factor mapping outperformed the common factor mapping in every case.

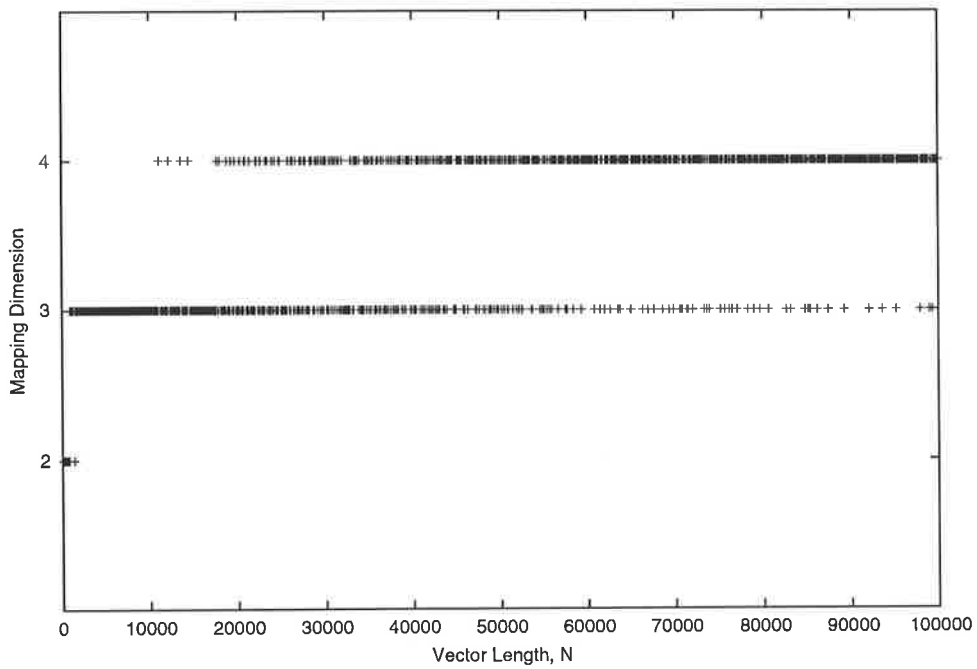


Figure 5.50: Mapping Dimension Used to Achieve Best Performance of Fourier Transform

From the graph it can be seen that the fourth order mapping only becomes useful for vector lengths above 10000, and that the third order mapping continues to be better for a small proportion of vectors up to length 100000.

That the prime factor mapped algorithm outperformed the common factor mapped algorithm in these results depended on the size of the range of vector lengths for each plotted point. The best mapping for single vector lengths between 50000 and 50100 are each plotted separately in Figure 5.51, which clearly shows that the common factor algorithm is occasionally the best for a specific vector length. Note that lower order mappings q=2 and q=1 (equivalent to directly using the definition of the DFT) are often the best for a specific vector length.

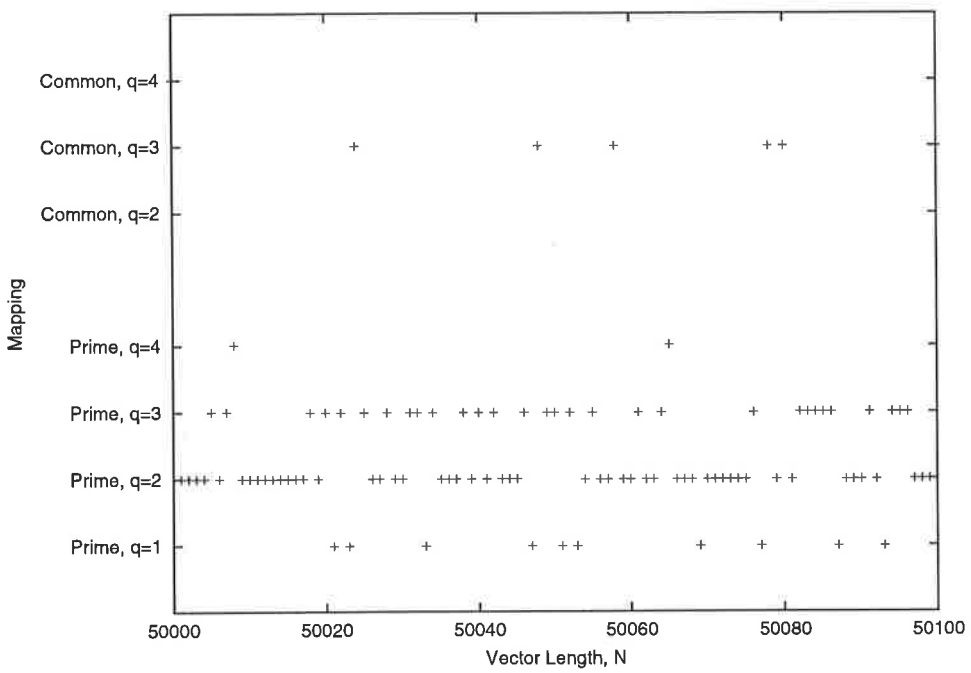


Figure 5.51: Mapping Dimension and Type Used to Achieve Best Performance of Fourier Transform

The results for different array sizes and maximum virtual factors are shown in Figure 5.52. This graph, and all those that follow, are plotted by taking the best performance over a range of a 1000 different vector lengths, for example, 1000 to 1999, 2000 to 2999. Plotting the performance in this way gives a smoother curve that more easily allows comparison between the different architectural configurations.

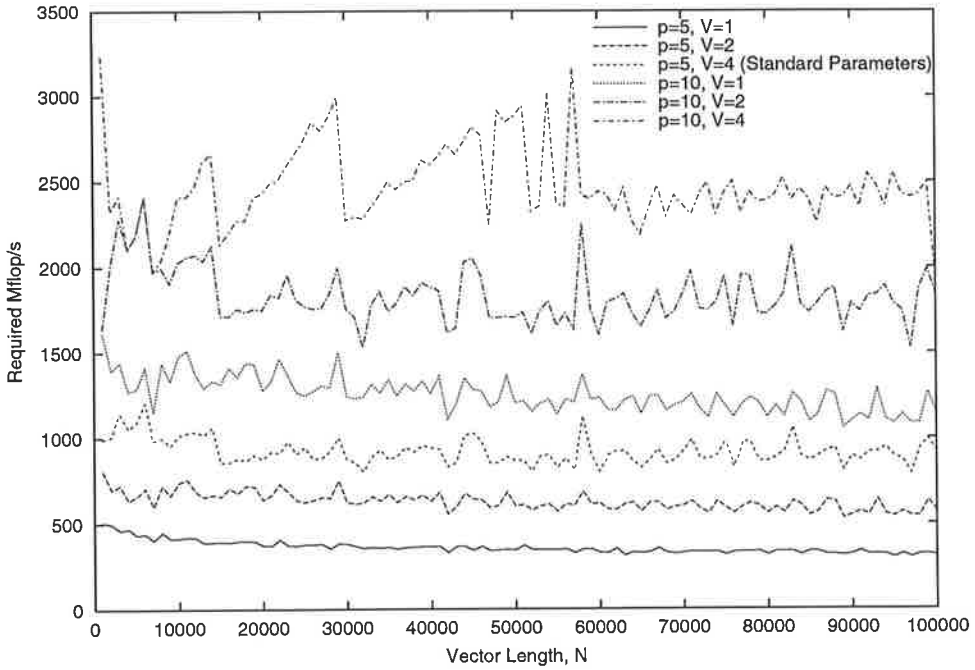


Figure 5.52: Performance of Fourier Transform for Different Array Configurations

The performance increases for larger arrays and higher virtual factors, but not in proportion to their peak performance. This phenomenon can be more clearly observed in Figure 5.53, which shows the performance plotted as a percentage of the peak performance for each architectural configuration.

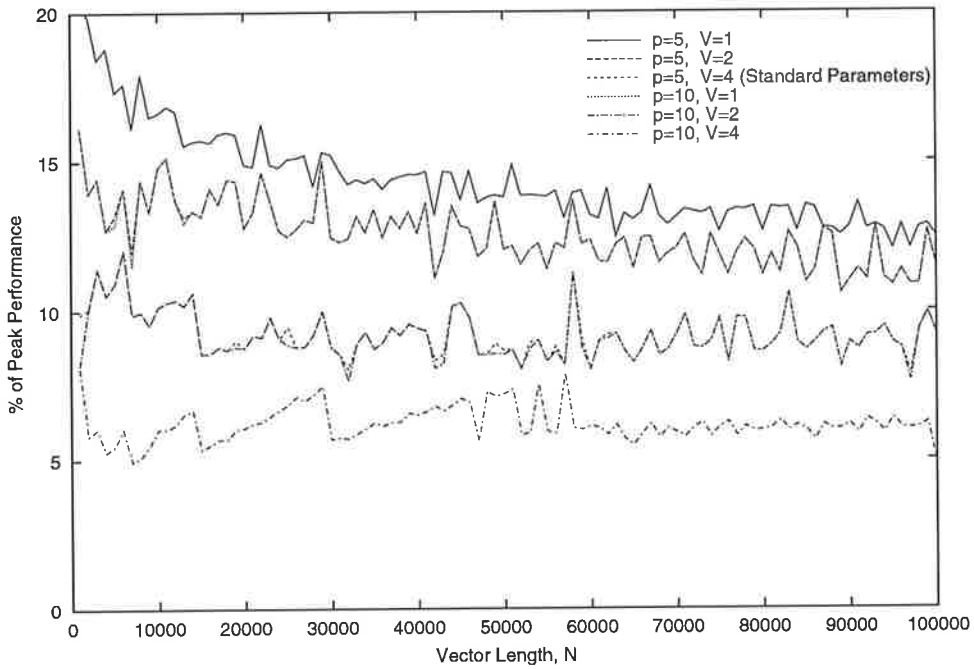


Figure 5.53: Performance of Fourier Transform for Different Array Configurations as a Percentage of Peak Performance

The percentage of peak performance obtained much lower than for Gaussian Elimination (Figure 5.23) and Householder QR Factorisation (Figure 5.42). The results for configurations with the same virtual array size, *pv*, are identical for almost all vector lengths.

The principal reason for the performance being so low is that the prime factor algorithm must perform many more floating point operations than the FFT for the same vector length. Performance plotted as a percentage of peak performance using the number of floating point operations that the prime factor algorithm actually uses, is shown in Figure 5.54.

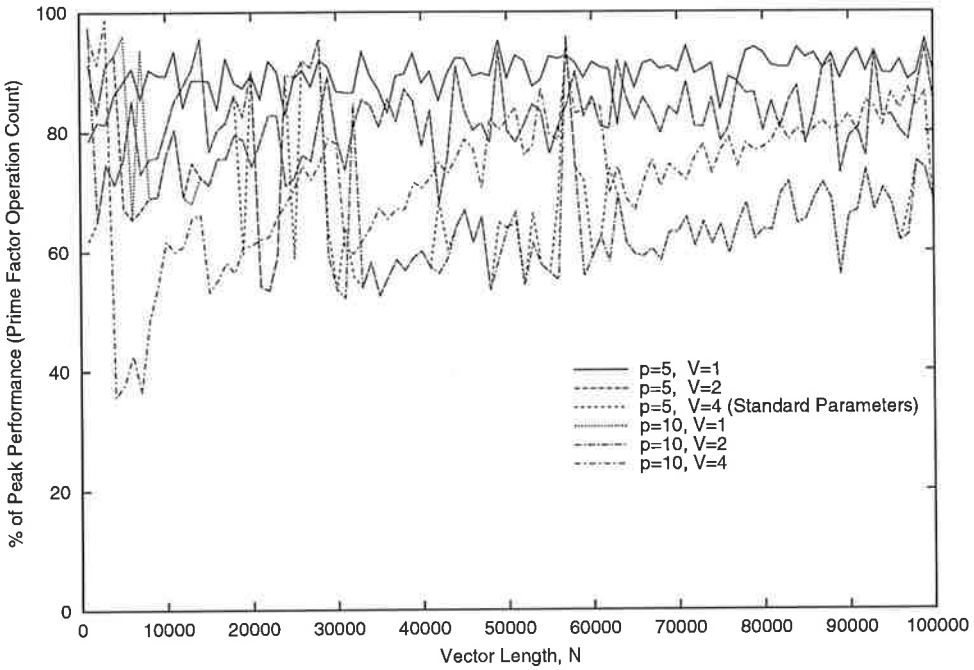


Figure 5.54: Performance of Fourier Transform for Different Array Configurations as a Percentage of Peak Performance using Prime Factor Mapped Operation Count

It can be seen that the MATRISC architecture achieves a very high percentage of peak performance when the results are calculated in this way. The reasons why performance falls short of the peak is that the size of matrix products being formed are not exact multiples of the virtual array size, and in some cases, oversized writeback requires extra cycles. These overheads are generally larger for larger virtual array sizes.

The results show that MATRISC architecture implements the prime factor mapped Fourier transform quite efficiently, achieving a high percentage of its peak performance. However, if there is enough flexibility in the choice of vector length, then the prime factor mapped algorithm is not as efficient, in terms of floating point operation count, as the Fast

Fourier Transform.

5.6.3 Load/Store Memory Architecture Overhead

The overhead of the Load/Store memory architecture compared with the hypothetical perfect memory architecture for the Fourier transform for various architectural parameters is shown in Figure 5.55. The performance of each memory architecture was taken as the best flop/s rate achieved in a range of a thousand elements of vector length. The comparison was made in this way because the architectures achieved their best performance for different vector lengths.

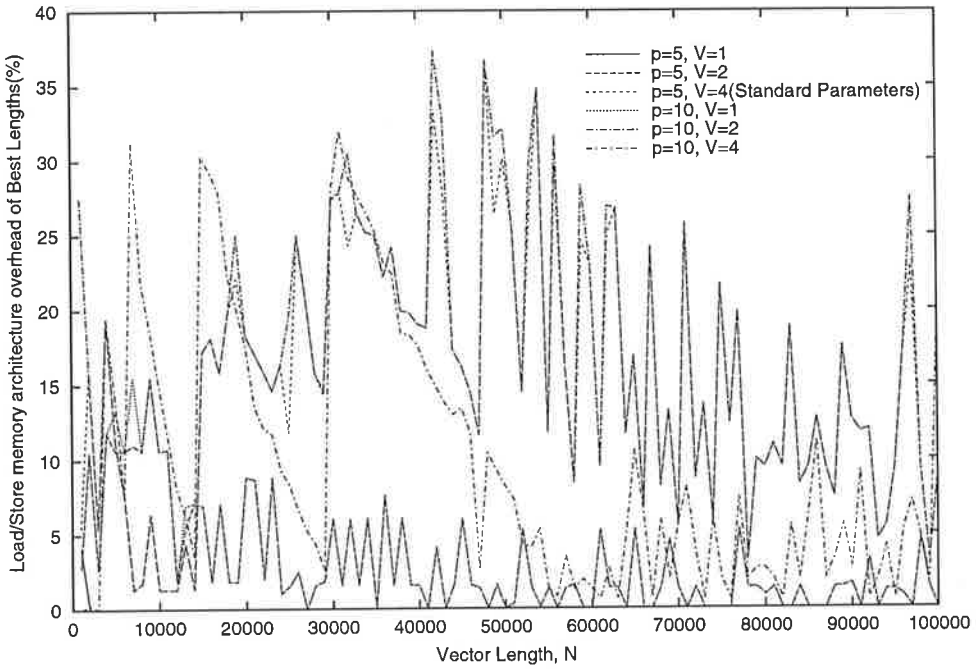


Figure 5.55: Overhead of the Load/Store Architecture versus the Perfect Memory Architecture for Fourier Transform

Again, for cases where the virtual array size is the same, the results are identical for most vector lengths. Note that the overhead for the case p=5,v=1 is zero except for N=1000.

As the implementation of the Fourier transform requires no extra operations to put the data into the correct storage form, and because no unaligned matrices are used, the only situation where the perfect architecture differs from the Load/Store architecture is when it does not perform oversized writeback from small multiplication operations. This accounts for the fact that the overhead is much smaller for the Fourier transform than for previous algorithms. All overhead could thus be eliminated for this algorithm by changing the

architecture to have a further control register that sets the size of writeback from multiplication operations.

5.6.4 Load/Store Transfer Time and Register Size Limitations

The time required to perform load/store transfers as a percentage of the time required for computation is shown in Figure 5.56 for a number of different array configurations. The load/store transfers include loading a complex input vector and storing a complex result, but do not include loading the W matrices, which would probably be done only once for a large number of transforms.

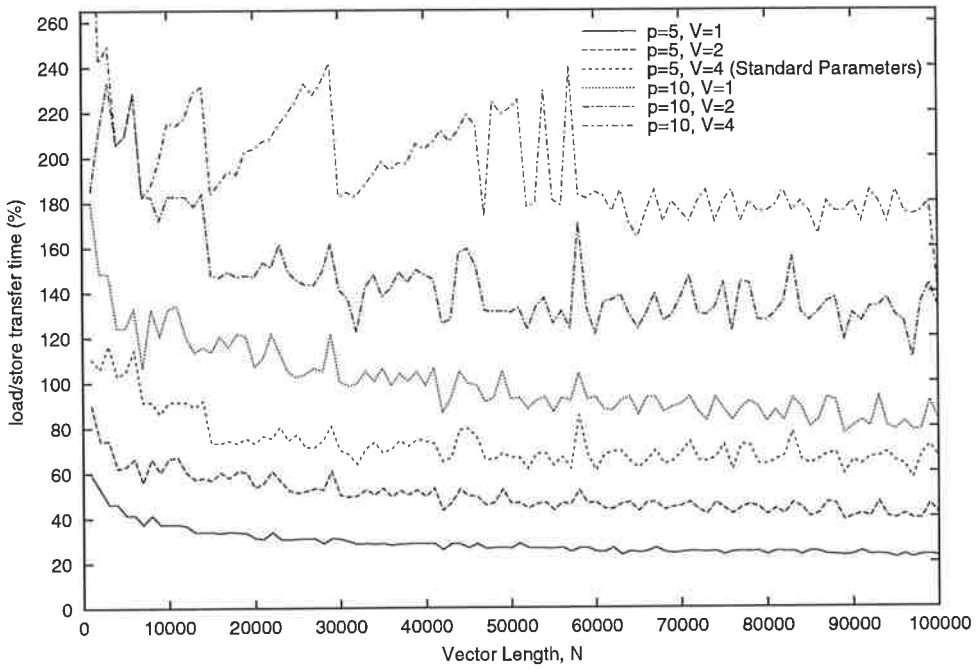


Figure 5.56: Load/Store Transfer Time for the Fourier Transform

For the calculation to be compute-bound, the relative transfer time must be less than 100%. When the array size is 5, the architecture can be expected to be compute-bound for a maximum virtual factor of 1, 2 or 4. However, an array size of 10 would only be compute-bound for a maximum virtual factor of 1, and then only for long vector lengths, roughly those above 50000 elements.

This situation arises because the load/store bandwidth does not scale with the size of the array, and so it forms an ever increasing bottleneck as the array size increases. It is particularly bad in the case of the prime factor mapped Fourier transform because the asymptotic computational requirement, $O(N^{1+1/q})$, only slightly exceeds the load/store

requirement, O(N), and so the load/store bottleneck improves only very slowly with longer vector lengths. This situation contrasts sharply with Gaussian Elimination and Householder QR reduction, where the O(N^3) computational requirement outstripped the O(N^2) load/store requirement, resulting in a small relative load/store transfer time for large problem sizes.

A further complication of the large load/store transfer time with the Fourier transform occurs when the data size is larger than the registers, and so must be partitioned. In this situation, each vector element must be transferred between main memory and the register more than once, which means that the relative transfer time must be less than 50% for the architecture to be compute bound. This implies that it is highly likely that the architecture will be memory-bound even for small array sizes. Thus the size of vector with which the architecture can efficiently deal is limited to the available register size.

5.6.5 Conclusions

At first glance, the prime factor mapped Fourier transform algorithm seems to be perfect for the MATRISC architecture because it exclusively uses matrix multiplication. However, there are two significant limitations.

Firstly, the prime factor mapped Fourier transform requires considerably more floating point operations than the Fast Fourier Transform for a given vector length. Thus if the Fast Fourier Transform is used to set the required number of floating point operations, the performance of the prime factor mapped transform reaches only a small percentage of peak performance. In an application, this method of performance evaluation is only fair when there is a large freedom in the choice of vector length; if this is the case, a binary power vector length can be chosen to allow use of the FFT. If the choice of vector length is more restricted, the prime factor mapped transformation may become the best method available. When the prime factor algorithm itself is used to set the required flop count, the performance is near 100% of peak.

Secondly, because of the low computational complexity compared with the data size of the Fourier transform, it is only compute-bound for small processor arrays. However, even for small arrays, the size of the largest problem that can be handled is limited by the size of the registers because partitioning the problem and performing the computation from main memory is memory-bound.

5.7 Summary

This chapter has presented a range of results, both theoretical and simulated, for five different operations and algorithms; addition, multiplication, Gaussian Elimination, Householder QR Reduction and Fourier Transform.

5.7.1 Basic Operations

Matrix multiplication is the most important operation on the MATRISC architecture, as the processor array is specifically designed to exploit the dense regular computation of matrix multiplication. Multiplication can achieve the peak speed for the architecture of $2Vp^2$ floating point operations per array cycle, but achieving this speed requires that the depth of the operands be greater than the size of the virtual array being used.

Matrix multiplication is a level-3 operation, which means that the computational requirement is $O(n^3)$ and the memory bandwidth requirement is $O(n^2)$ for multiplying two square matrices of order n. In general, these requirements imply that matrix multiplication will be compute-bound for sufficiently large matrices. The matrix size above which the architecture is compute-bound, called the critical size, was found to be $3Vp^2F$, which evaluates to 225×225 for the standard parameters. Matrices of unlimited size may be multiplied efficiently using a compute-bound blocked matrix approach, provided that there is sufficient space to store two matrices of critical size in each register. The fundamental point is that, with sufficient register space, level-3 operations will be compute-bound even though the computational bandwidth is much greater than the load/store bandwidth. This was demonstrated for matrix multiplication and for the level-3 algorithms, Gaussian Elimination and Householder QR Factorisation.

For each register to hold two matrices of critical size, each register column must have space for $18V^2p^3F^2$ elements. As the amount of memory per column will be limited by technological restraints, the size of the processor array that will be compute-bound will also be limited. For example, if one 4Mbit SRAM is used per register column, the maximum array size will be 7×7 if a maximum virtual factor of 4 is used.

Elementwise computation operations, such as addition, are quite different to matrix multiplication because they are memory bandwidth limited. These operations perform only p floating point operations per array cycle, so operate at only $100/(2Vp)\%$ of the

processor array's peak performance. For this reason, the disparity between the performance of matrix multiplication and matrix addition grows larger for larger arrays.

5.7.2 Computational Performance

The computational speed of the architecture, as measured by the number of array cycles required to perform the algorithm, was found by either simulating the given algorithm on the MATRISC simulator, or evaluating a theoretically derived expression. In some cases, values were calculated using both methods and were found to be the same, thus validating the models used. Performance was then expressed as a rate of floating point operations per second using the required number of floating point operations for the operation, and an array cycle time of 20ns. This floating point speed was also converted to a percentage of peak performance for the particular set of architectural parameters under consideration.

The performance of level-3 algorithms was low for small matrix sizes, where non-multiplication overheads dominated the calculation, but increased for larger matrices where matrix multiplication began to predominate. For Gaussian Elimination, a Fast algorithm was developed that achieved 72% of peak performance for the standard parameters, and reached half this performance for a matrix of order 250. Pivoting algorithms reduced performance by at least a factor of two. The block algorithm for Householder QR factorisation achieved 54% of peak performance, and reached half this for a matrix of order 700.

The performance of the Fourier Transform, which is not a level-3 algorithm, was between 60% and 90% of peak performance relative to the actual number of floating point operations required by the algorithm, but only about 10% of peak performance relative to the number of operations required for the FFT.

For all algorithms, absolute performance was greater for larger arrays and larger maximum virtual factors, but the percentage of peak performance with these configurations was less. This is due to increased overheads from matrices that do not exactly fill a register row and from the increase in the ratio between the multiplication and addition speeds. The result is that larger systems will have a poorer price/performance ratio that will, at some point, place a limit on the size of system that is viable.

5.7.3 Memory Architecture Performance

The performance of the Load/Store memory architecture was assessed using two

different measures.

The first was to assume a hypothetical 'perfect' memory architecture and to calculate the number of array cycles required for each algorithm on this architecture. Comparison of this new figure with that for the Load/Store memory architecture allowed calculation of the overhead caused by the partitioned nature of the matrix registers in Load/Store architecture. Note that there is no way that the perfect memory architecture could be constructed to run at the same array cycle time as the Load/Store memory architecture, but nonetheless, it provides a useful theoretical benchmark.

For the level-3 algorithms, the overhead was high for small matrices but fell away with larger matrices as matrix multiplication in the block update step became dominant. For example, the overhead for Gaussian Elimination with the standard parameters was 70% for small matrices, falling to under 20% for matrix order 1000, and was 50% for Householder QR Reduction, falling to under 15% for matrix order 1000. The overhead increased with virtual array size, but less than linearly. For the prime factor mapped Fourier transform, the overhead was a maximum of 37% for the standard parameters.

The second measure of the Load/Store memory architecture was the length of time taken to transfer data between main memory and the registers, expressed as a percentage of the time taken for computation. If this figure is less than 100%, the architecture is compute-bound given sufficient register memory, and so the architecture will run at the floating point rates calculated from the array cycle counts.

For the level-3 algorithms, the relative proportion of load/store transfers compared with computation is inversely proportional to the matrix order, which means that these algorithms must be compute-bound for sufficiently large matrices. However, because of their relatively poor computational performance for small matrices, both level-3 algorithms were compute-bound for all matrix orders. In addition, a compute-bound algorithm for Gaussian Elimination operating from main memory was presented, which required only that each register have space for two matrices of critical size.

On the other hand, the proportion of time spent on data transfer for the q-dimensional prime factor mapped Fourier Transform decreased only very slowly with vector length, because it has $O(N^{1+1/q})$ computational complexity and $O(N)$ data transfer for a vector length N. The relative load/store transfer time also increased with virtual array size, which meant that the algorithm was memory-bound for large array sizes, and was compute-bound for small arrays only when the data fits into the registers.

5.7.4 Conclusion

The Load/Store architecture performed well on algorithms that are rich in matrix multiplication. The best performance was generally achieved for large matrices, where the overhead of non-multiplication sections of the computation was relatively small. For operations such as the Fourier Transform, where the most efficient algorithms cannot be expressed as matrix multiplication, the performance was somewhat lessened.

These results demonstrate that an architecture that performs matrix operations directly in hardware can effectively accelerate important algorithms. Furthermore, the Load/Store memory architecture can successfully provide a very high memory bandwidth, thus overcoming the bandwidth limitation of the earlier Memory/Memory MATRISC architecture.

Chapter 6

Conclusion

This thesis has proposed a new specialised architecture for accelerating the performance of applications based on matrix algorithms. The significance of this architecture lies in its use of two important computer architecture design principles: the load/store paradigm, and the support of dense matrix data types in hardware.

The Load/Store MATRISC architecture is derived from Marwood's MATRISC architecture, which successfully demonstrated the concept of a specialised architecture for matrix operations using a memory/memory architecture. The most significant limitation in the original architecture was a lack of memory bandwidth. Accordingly, the most important new development described in this thesis was a high bandwidth memory architecture that supports a load/store paradigm, and which brings the architecture fully into line with the RISC philosophy by providing general purpose registers, and by restricting data flow between the registers and memory to explicit load and store instructions. It is significant that the original design goal for the Load/Store architecture was simply to increase performance by improving the memory architecture: that a load/store memory architecture emerged as the most effective solution is evidence of the applicability of the RISC principle to a wide range of computing tasks.

The success of hardware support for dense matrix operations is based on the two principal tenets of the MATRISC philosophy: firstly, that many useful algorithms can be expressed in matrix terms; and secondly, that because of their inherent parallelism, matrix primitives, especially matrix-matrix multiplication, can be implemented in hardware to achieve high computational rates. The regularity of matrix operations means that parallelism can be extracted by replicating functional units, leading to high performance with relatively little control overhead. In addition, the MATRISC processor array takes

advantage of the level-3 nature of the matrix product by having p^2 processors while requiring the bandwidth of only $2p$ register columns. A higher computational rate than would otherwise be possible is thus achieved for the given memory bandwidth, which improves the cost effectiveness of the architecture because, in general, memory bandwidth is more expensive to provide than computational bandwidth.

The processor array was further enhanced by a new architectural feature proposed in this thesis called the virtual array. By using processing elements that can act as if they were a small array of virtual elements, a better balance between computational and memory bandwidth can be obtained over a wide range of input sizes.

The crucial advantage of the Load/Store memory architecture is the increased bandwidth provided to the processor array by the parallel registers. Importantly, this bandwidth scales with the size of the processor array. The bandwidth between main memory and the registers is much lower and does not scale with the size of the array. However, compute-bound operation is still possible for level-3 algorithms if the problem size is large enough and there is sufficient register memory available. For matrix-matrix multiplication, the simplest level-3 operation, there is a critical size above which the architecture will be compute-bound given enough register space. This critical size is equal to $3Vp^2F$, which evaluates to 225×225 for the standard parameters. If each register is large enough to hold two matrices of critical size, which requires $18V^2p^3F^2$ elements per register column, then a compute-bound block multiplication algorithm can be used for arbitrarily large matrices above critical size. The fact that multiplication between matrices of unlimited size can be compute-bound with registers of a fixed size is fundamental to the success of this new architecture, and indicates that the disparity between load/store and read/write bandwidths does not restrict the processing speed when a sufficiently large amount of register memory is available.

The Load/Store memory architecture achieves a high memory bandwidth by using two parallel memories connected to the processor array. These memories are built from SRAMs, but are controlled as registers rather than as caches. The registers consist of a number of register columns that are each connected to one row, or column, of the processor array. This design limits the flexibility of the system because not all the data in the registers is directly available to all of the processing elements, which implies that the data in the registers must be correctly stored for the computation to be possible, and that superfluous copy operations are sometimes required. The penalty, relative to a hypothetical 'perfect' memory

architecture, is 50% or over for some algorithms, but falls to under 20% for calculations on large matrices. However, it is important to note that this 'perfect' architecture against which the comparison was made is not realisable, and so these figures do not represent a deficiency with respect to a realistic alternative.

In the Memory/Memory MATRISC architecture, the address generators lay between the processor array and the two caches, backed by main memory. Conversely, with the Load/Store memory architecture, the address generators for the loading and storing of matrices must transfer matrix data between a main memory composed of DRAMs and parallel registers composed of SRAMs. If the address generator from the Memory/Memory architecture were used in the Load/Store architecture, the non-sequential addresses it produces for some mappings would result in a low data transfer from the main memory DRAMs. However, by using the novel inverted address generator proposed in this thesis, which generates the inverse mapping to the normal address generator, fast loading and storing can be achieved because the inverse address generator sequentially accesses main memory, thus getting the best possible transfer rate from the DRAMs.

Through simulation and analysis, the Load/Store MATRISC architecture has been shown to be a viable system for the implementation of dense matrix algorithms. Although the architecture has not been built, it presupposes nothing not currently available, and it achieves over 50% of peak performance for a number of significant algorithms. Absolute performance will improve as technology advances but the principles and trade-offs remain.

The most significant limitation of the architecture discovered was that the size of the processor array that can be used is limited. This is due to two fundamental architectural factors.

Firstly, the ratio between the speeds of matrix multiplication, the architecture's fastest operation, and the elementwise operations, such as matrix addition, grows larger with an increase in array size. Thus as the array size increases, the performance of any algorithm that does not solely use matrix multiplication will be increasingly affected by the overhead of the non-matrix multiplication part of the algorithm. A larger array will achieve a better absolute performance than a smaller array, but a poorer percentage of peak performance thereby reducing cost/performance ratio. For example, the Fast Gaussian Elimination algorithm achieves 7200Mflop/s for a 5×5 array, which is 72% of peak performance, and 22000Mflop/s for a 10×10 array, which is only 55% of peak performance, when both have a maximum virtual factor of 4. This limitation is inherent in the edge-fed MATRISC

processor array, and is not due to the Load/Store memory architecture.

Secondly, the amount of memory required for each register column to ensure compute-bound operation grows with the array size cubed. This relationship implies a limit on the size of the array because technology will limit the amount of memory that can be used for each register column. For example, one 4Mbit SRAM per register column would provide enough register memory to ensure compute-bound operation for a 7×7 array (see (5.13) and §5.3.3). For algorithms that are less computationally intensive than a level-3 algorithm, such as the Fourier Transform, the lack of load/store bandwidth causes a bottleneck for sufficiently large arrays. For example, a 10×10 array is memory-bound for the Fourier Transform. This limitation is due to the Load/Store memory architecture.

With the current architecture, the control processor also presented some limitations. The time taken to execute control processor instructions was found to be much greater than the time required to perform the computation on the processor array. For Gaussian Elimination using the relatively optimised Invert algorithm, control instructions took 15 times longer than computation, mainly due to very poor code generation in the Masm compiler. Analysis revealed that essential control processor instructions required less time than computation for matrices above order 100. Use of an optimising compiler and enhancements to the control of the architecture would solve this problem.

The process of implementing these different algorithms on the MATRISC processor was relatively straightforward, with almost all of the difficulties being due to the primitive programming environment. In practice, the problems associated with ensuring that matrices are in the correct storage forms, and correctly aligned to make the computation possible, were relatively few. However, it is still unclear how to automate the process so that the architecture can be programmed from a high level language.

The results in this thesis are based on three assumptions about the behaviour of different aspects of the MATRISC Load/Store architecture. Firstly the buses used to connect the processor array were assumed to scale while maintaining a constant array cycle time. This assumption must certainly break down beyond a certain size that will depend on the technology used to build the buses. However, as stated above, the array size is limited by other factors and the assumption that the bus will scale to around a 10×10 array is reasonable and verified by other research by the MATRISC group. Secondly, a control processor with low latency was assumed. Such a processor is not commercially available, but is certainly feasible in a custom or semi-custom design. Finally, it was assumed that

stalling will have a minimal impact on the performance of the architecture. This was based on the small amount of data dependent branching in most dense matrix algorithms and on the relatively small pipelining latency compared to the overall latency of each operation. All three assumptions would require additional investigation before the architecture could be implemented.

However, despite these limitations, the MATRISC Load/Store architecture is still a technically viable system for achieving high computational performance for a number of important algorithms. These results have further demonstrated the potential of the MATRISC philosophy and shown how limitations in the original Memory/Memory architecture can be overcome. Furthermore, areas in which the performance of the new architecture is poor have been identified and investigated. These limitations have been shown to be either fundamental to the edge-fed MATRISC processor array or caused by the Load/Store memory architecture.

This work has provided an understanding of the basic properties of the MATRISC architecture that is relevant not only to attempts to further improve the MATRISC architecture, but also to the design of matrix-based architectures in general.

There is a further question as to whether the MATRISC architecture, in any form, or the Load/Store MATRISC architecture in particular, is economically viable. The answer to this question at the present time is almost certainly no. The applications that the architecture targets are important but not dominant. The cost of developing the architecture to the point that it could compete with commercial microprocessors is likely to be too large.

However it is possible that the architecture will become economically viable in the future for two reasons. Firstly, the dense matrix algorithms will become more widely used, for example, in more sophisticated real time control and in simulating more realistic virtual environments. Secondly, the bulk of the improvement in the performance of scalar microprocessors has come through developing ever more advanced architectures. For example both the principal high volume producers have integrated simple multiprocessor features into their products. At some point it will become technically possible to integrate a small MATRISC array as part of a 'scalar' processor, at which point the high design cost will be recoverable from volume sales.

6.1 Further Work

This work has raised a number of interesting questions about specialised matrix-based architectures that would benefit from further research. The following sections briefly discuss three of these areas.

6.1.1 Multiprocessor Node Integration

A relatively open architectural issue is the way in which a matrix processor should be integrated into an existing computing environment. There have been two principal suggestions. The first is as an accelerator card on a traditional uniprocessor machine, such as a workstation. The SCAP implementation operated in this way. The second is as a node of a massively parallel machine, which was investigated by Shaw. Both of these approaches seem promising, but they require investigation and careful implementation for the architecture to achieve optimum performance.

A third option, which amalgamates these two ideas, is to use such a processor to accelerate the nodes of a cluster of workstations. In this case, the architecture would physically be implemented as an accelerator card, but would function as part of a multiprocessor. A potential problem with such a system is that a dramatic increase in the computational power of the nodes would lead to the interconnection bandwidth becoming insufficient, thus causing a bottleneck. However, there is great potential to achieve massive computational performance at a relatively low cost.

6.1.2 Matrix Controller Integration

Simulations have demonstrated that the matrix controller and control processor used to perform the overall control of the architecture were too slow. Although investigation revealed that this situation was largely caused by the poor code generation of the Masm compiler, this is still an area where significant architectural advances are possible.

One exciting possibility would be to integrate a small array onto a single chip as part of a powerful microprocessor. The recent Pentium III processor includes a feature called 'Streaming SIMD Extensions' that allows parallel elementwise operations on a four element vector. Expanding this facility to use a MATRISC outer product array to perform matrix multiplication would result in an extremely powerful numerical processor. The

difficulty would be having sufficient on-chip storage to maintain the computation, but at some point in the future, such an architecture will be realisable.

6.1.3 MDMA Hybrid Approach

As discussed in §2.5.3, a new approach to the MATRISC architecture called the Multi-Dimensional Memory Architecture (MDMA) has also been proposed. MDMA involves integrating some memory onto every processing element and using a more flexible interconnection network within the processor array. However the size of the memory that each processing element can have is quite limited when compared with the size of the matrix registers in the Load/Store architecture.

One possible scenario would be to combine the two architectures so that an MDMA array was connected to the Load/Store memory architecture. This would allow the flexibility of the MDMA array, while providing a large amount of storage close to the array for algorithms that require it, such as linear system solution.

There are a number of issues with how such a system would work, particularly with respect to control and programming.

Appendix A

Matrix Data Path

The programmer's model of the matrix data path is shown in Figure A.1. It consists of the processor array, two registers, a main memory and the matrix controller. The controller orders the flow of data throughout the data path according to the SET and DO instructions sent by the control processor. The operations performed by the data path can be divided into two groups, load-store operations and compute operations. The groups are controlled by different functional units within the matrix controller and so can be performed in parallel.

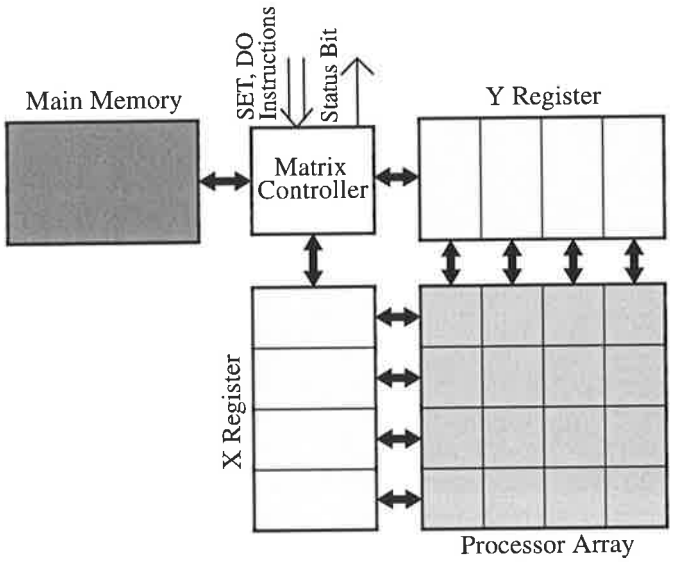


Figure A.1: Matrix Data Path - Programmer's Model

The processor array consists of a $p \times p$ grid of processing elements. Each processing element contains a $V \times V$ array of accumulator and result registers, where V is the maximum virtual factor.

The registers store rows of data that each have p elements. The data in the registers is

either accessed row at a time by computational operations or individual elements are accessed by load or store operations. The addressing scheme used to refer to locations within the registers is shown in Figure A.2.

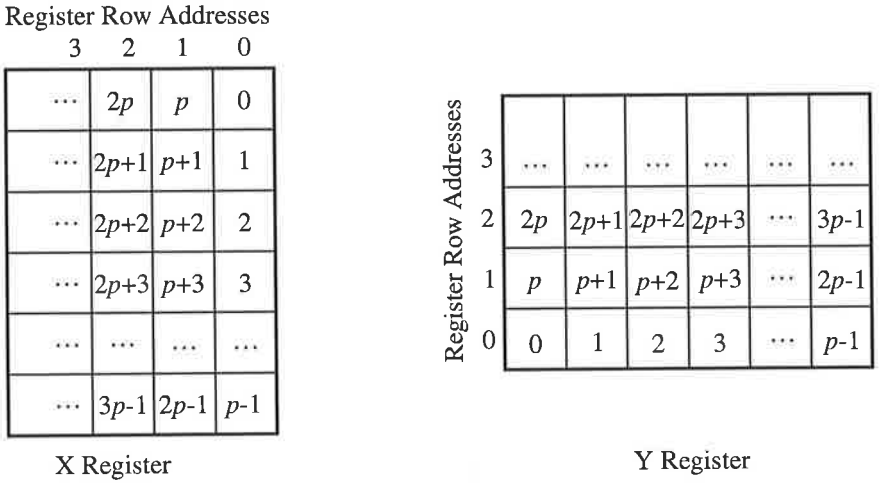


Figure A.2: Register Addressing Scheme

The numbers shown in the individual element locations within the registers are the element addresses used when individual elements are referred to during load and store operations. The register row addresses are used to refer to whole register rows during compute operations.

To fully describe the different operations, pseudo-code descriptions will be used. The code is based on Ada with Matlab colon notation to indicate subarrays. The following variable names will be used to indicate data path storage locations in the code,

M() - main memory

X() - the X register. The index is assumed to be an element address when used in load and store operations and the expression is then a single number. In compute operations, the address is assumed to be a register row address and the expression is a vector of length p.

Y() - the Y register. The index is treated as for X().

A()() -the processing element accumulators. This is a $p \times p$ array of $V \times V$ arrays. A(i, j) is the accumulator array of the processing element in row i, column j.

R()() -the processing element result registers. Its form mirrors that of A().

Data path control registers are referred to by their names and temporary variables are used as needed. It is important to realise that the pseudo-code indicates what operation is performed rather than how it is actually performed.

A.1 Load-Store Operations

Load and store operations involve transferring data to the registers from main memory, and from the registers to main memory, respectively. Each transfer occurs between a sequence of SRAM addresses in register memory and a sequence of DRAM addresses in main memory. The two address sequences are produced by two address generators according to the equations,

$$\text{SRAM\_address}(n) = \text{Saddress} + \langle \delta_1 \left\lfloor \frac{n}{\sigma_1} \right\rfloor \rangle_{q_1} + \langle \delta_2 \left\lfloor \frac{n}{\sigma_2} \right\rfloor \rangle_{q_2} + \langle \delta_3 \left\lfloor \frac{n}{\sigma_3} \right\rfloor \rangle_{q_3} + \langle \delta_4 \left\lfloor \frac{n}{\sigma_4} \right\rfloor \rangle_{q_4}$$

$$\text{DRAM\_address}(n) = \text{Daddress} + \text{Increment} \left\lfloor \frac{n}{\text{Runlength}} \right\rfloor + \text{Stride}(n \mod \text{Runlength})$$

where n is an index number that ranges from 0 to LSLength-1, which implies that LSLength values are transferred. δ_i, σ_i and q_i are the values of the data path control registers Delta_i, Sigma_i and Q_i respectively.

The pseudo-code of the X register operations is as follows: the Y register operations are analogous.

```
procedure LoadX is
        for i in 0 .. LSLength-1 loop
                X(SRAM_address(i)) := M(DRAM_address(i));
        end loop;
end LoadX;

procedure StoreX is
        for i in 0 .. LSLength-1 loop
                M(DRAM_address(i)) := X(SRAM_address(i));
        end loop;
end StoreX;
```

A.2 Compute Operations

Compute operations can be further subdivided into three groups: multiply operations, addition-like operations and test operations.

Any data loaded from the registers into the processor array is affected by the setting of the two sign mode control registers. The operation performed is defined by the function,

$$\text{SignX}(\underset{\sim}{a}) = \begin{cases} \underset{\sim}{a} & \text{XSignMode} = \text{Plus} \\ -\underset{\sim}{a} & \text{XSignMode} = \text{Minus} \\ |\underset{\sim}{a}| & \text{XSignMode} = \text{Abs} \\ \underset{\sim}{a}/|\underset{\sim}{a}| & \text{XSignMode} = \text{Sign} \end{cases}$$

for the X operand and a similar function for the Y operand. Here the modulus sign | | indicates an elementwise modulus operation rather than the length of the vector. Also 0/0 = 0.

Multiply and addition-like operations produce results that are stored in registers depending on the setting of WBMode. The writeback is classed as either linear or diagonal and can occur to the X register or the Y register, or no writeback maybe performed at all. The meaning of the linear and diagonal classes is dependent on the type of operation being performed as is explained below.

A.2.1 Multiply Operations

There are two multiply operations: Multiply and Chain. Multiply begins by clearing the processing element's accumulators whereas Chain does not, so a chain operation effectively adds the new product being formed to the previous contents of the array.

Multiply operations can take advantage of the *virtual array* concept, where, by using time multiplexing of the processing elements, the physical processor array can be used as if it where a larger virtual array. For example, if a virtual factor of two is used the array is effectively $2p \times 2p$.

During a multiply operation, the source operands are first transferred into the processor array from the registers and the appropriate multiply-accumulate operations are performed within the processing elements. This is followed by a writeback phase during which the results are transferred from the accumulators to the result registers and then written back to the registers. If a linear writeback mode is set, then the whole result is written back one row or column at a time. If a diagonal writeback is set, only the leading diagonal elements are written back as one register row. The reading of operands and the writing back of results is controlled by three simple address generators that generate register row addresses according to the equations,

$$\text{X\_source\_address} = \text{Xaddress} + \text{Xstep}\left\lfloor \frac{n}{\text{Virtual}} \right\rfloor + n \bmod \text{Virtual}$$

$$\text{Y\_source\_address} = \text{Yaddress} + \text{Ystep}\left\lfloor \frac{n}{\text{Virtual}} \right\rfloor + n \bmod \text{Virtual}$$

$$\text{result\_address} = \text{Raddress} + \text{Rstep}\left\lfloor \frac{m}{\text{Virtual}} \right\rfloor + m \bmod \text{Virtual}$$

Here n is a sequence number, which ranges from 0 to Length-1, and m is a sequence number, which ranges from 0 to vp-1 for linear writeback. The pseudo-code description of the multiply and chain operations is as follows. Note that to save space, the Y register writeback modes are not shown; they are analogous to the X register modes.

```
procedure Multiply is
        A(:,:)(:,:) := 0.0;
        Chain;
end Multiply

procedure Chain is
        for i in 0 .. Length-1 step Virtual loop
                for j in 0 .. Virtual-1 loop
                        X_input(j:p*Virtual-Virtual+j:Virtual) := Xsign(X(X_source_address(i+j)));
                        Y_input(j:p*Virtual-Virtual +j:Virtual) := Ysign(Y(Y_source_address(i+j)));
                end loop;
                for j in 0 .. Virtual*p - 1 loop
                        for k in 0 .. Virtual*p -1 loop
                                A(j/Virtual,k/Virtual)(j mod Virtual, k mod Virtual) :=
                                        A(j/Virtual, k/Virtual)(j mod Virtual, k mod Virtual)
                                                                        + X_input(j)*Y_input(kl);
                        end loop;
                end loop;
        end loop;
        R := A;
        case WBMode is
                when LinearX =>
                        for i in 0 .. Virtual*Virtual*p - 1 step Virtual loop
                                i1 := i mod Virtual;
                                i2 := (i/Virtual) mod Virtual;
                                i3 := i/(Virtual*Virtual);
                                X(result_address(i)) := R(:, i3)(i1, i2);
                        end loop;
                when Diagonal X =>
                        for i in 0 .. p-1 loop
                                X_result(i) := R(i,i);
                        end loop;
                        X(result_address(0)) := X_result;
                when NoWB =>
                        null;
        end case;
end Chain;
```

A.2.2 Addition-like Operations

The addition-like operations are addition, Hadamard multiplication, division and square

root, which all operate in the same way except for the operation performed within the processing element. All these operations use a virtual factor of 1 regardless of the setting of the Virtual control register.

During addition-like operations, the operands are transferred into the array in the same fashion as for multiply operations. However, after every pair of data elements that each processing element receives is operated on, the result is immediately moved to the processing element's result register and written back. Diagonal writeback modes write back only the results from the leading diagonal processing elements. Linear writeback modes write back a the results from a column of processing element, if the destination is the X register, or results from a row of processing elements, if the destination is the Y register. The row or column used is controlled by the Row and Column control registers.

The addition-like operation are described by the following pseudo-code that again omits the Y register writeback modes.

```
procedure Addition_Like_Operation (operation) is
    for i in 0 .. Length-1 loop
        X_input(0:p-1) := Xsign(X(X_source_address(i)));
        Y_input(0:p-1) := Ysign(Y(Y_source_address(i)));
        for j in 0 .. p - 1 loop
            for k in 0 .. p - 1 loop
                case operation is
                    when Addition =>
                        A(j,k)(0,0) := X_input(j) + Y_input(k);
                    when Hadamard =>
                        A(j,k)(0,0) := X_input(j) * Y_input(k);
                    when DivideXY =>
                        A(j,k)(0,0) := X_input(j) / Y_input(k);
                    when DivideYX =>
                        A(j,k)(0,0) := Y_input(k) / X_input(j);
                    when SqrtX =>
                        A(j,k)(0,0) := sqrt(X_input(j));
                    when SqrtY =>
                        A(j,k)(0,0) := sqrt(Y_input(k));
                end case;
            end loop;
        end loop;
        R := A;
        case WBMode is
            when LinearX =>
                X(result_address(i)) := R(:,Column)(0,0);
            when Diagonal X =>
                for i in 0 .. p-1 loop
                    X_result(i) := R(i,i)(0,0);
                end loop;
                X(result_address(i)) := X_result;
            when NoWB =>
                null;
        end case;
    end loop;
end Addition_Like_Operation;
```

A.2.3 Test Operations

The test operation are TestZ, TestNZ, TestP and TestN, which are abbreviations for test for zero, test for not zero, test for positive and test for negative, respectively. The result of the test is sent to the control processor's D flag. These operations test the current contents of the processing elements accumulators. They do not take any source operands from the registers or write back any results. The actual processing elements tested are controlled by the Row and Column control registers, with -1 being interpreted as an 'all' value. Thus if both Row and Column are -1, then the whole array is tested; if just Column is -1, then Row selects a row to be tested; if just Row is -1, then Column selects a column to be tested; and if neither is -1, then Row and Column select a single element to be tested. When multiple elements are tested, the overall test result is the logical OR of the results for each element. This mean that TestZ, for example, on a vector tests for any zero element.

The pseudo-code for test operations is as follows.

```
procedure Test_Operation (operation) is
        if Row = -1 then
                row_start := 0;
                row_stop := p-1;
        else
                row_start := Row;
                row_stop := Row;
        end if;
        if Column = -1 then
                col_start := 0;
                col_stop := p-1;
        else
                col_start := Column;
                col_stop := Column;
        end if;
        D := false;
        for i in row_start .. row_stop loop
                for j in col_start .. col_stop loop
                        case operation is
                                when TestZ =>
                                        D := D or (A(i,j)(0,0) = 0.0);
                                when TestNZ =>
                                        D := D or (A(i,j)(0,0) /= 0.0);
                                when TestP =>
                                        D := D or (A(i,j)(0,0) > 0.0);
                                when TestN =>
                                        D := D or (A(i,j)(0,0) < 0.0);
                        end case;
                end loop;
        end loop;
end Test_Operation;
```

A.3 Instruction Pipelining

There are two types of pipelining, where the execution of two operations overlap in time, present in the MATRISC processor data path. One is overlap between load-store and compute operations, and the other is overlap between different compute operations. The second form occurs because the writeback phase of each compute operation occurs in parallel with the operand reading of the next operation, particularly for multiply operations. The presence of pipelining is important as it introduces the possibility of data hazards, where incorrect operation results because accesses to data do not happen in the expected order. Interaction between load-store and compute operations can result in all of the main types of data hazard: read after write (RAW), write after read (WAR) and write after write (WAW). However, it is guaranteed that the reads from registers and writes to registers of different compute instructions occur in the order of instruction execution. This fact implies that only write after read hazards are possible between compute instructions.

Prevention of data hazards is usually performed in hardware by stalling one or more instructions. This approach is not feasible in the MATRISC architecture because of the difficulty in detecting when hazards occur. The data flow analysis techniques used to avoid hazards in scalar processors are not appropriate in MATRISC because there are hundreds of thousands of register elements and each operation can take an arbitrarily long time. It is proposed that special instructions be introduced to prevent data hazards under the direction of software.

Two new data path operations are proposed: SyncAll and SyncCompute. SyncAll will stop the execution of any load-store or compute instructions that follow it until all previous operations have completed. SyncCompute will stop the execution of any compute operation that follows it until all previous compute operations have been completed. Exactly when these instructions will be needed in the code will depend in a complex way on the actual implementation of the architecture. However by using dependency information from a high level description of the algorithm to be performed a compiler could prevent data hazards with little overhead in terms of unnecessary Sync operations.

Appendix B

Inverted Address Generator Proofs

This appendix contains the proofs of the inverses used for the prime factor and Chinese remainder theorem mappings.

The notation $\langle a \rangle_N$ is use to denote a modulo N, that is, the remainder when a is divided by N. The following simple modulo properties are stated without proof.

$$\langle ab \rangle_N = \langle \langle a \rangle_N \langle b \rangle_N \rangle_N$$
$$\langle a + b \rangle_N = \langle \langle a \rangle_N + \langle b \rangle_N \rangle_N$$
$$M \langle a \rangle_N = \langle Ma \rangle_{MN}$$

The inverse of a modulo N is denoted $\langle a^{-1} \rangle_N$ and defined by,

$$\langle a \langle a^{-1} \rangle_N \rangle_N = 1$$

and exists if and only if a and N are coprime.

B.1 Lemma 1

Consider p mutually prime numbers, N_1 to N_p, then

$$\langle a \rangle_{N_i} = 1 \qquad \forall i \in 1 \ldots p \qquad \Rightarrow \qquad \langle a \rangle_N = 1 \qquad \text{where} \qquad N = \prod_{i=1}^{p} N_i$$

The proof is as follows. First, using the division process, there must exist integers m_1 to m_p, such that a can be expressed as,

$$a = m_1 N_1 + 1 = m_2 N_2 + 1 = \ldots = m_p N_p + 1$$
$$\Rightarrow m_1 N_1 = m_2 N_2 = \ldots = m_p N_p$$

Now as N_1 and N_2 are coprime, m_1 must be divisible be N_2 and similarly by N_3 to N_p. Hence m_1 may be expressed as,

$$m_1 = \alpha N_2 N_3 \ldots N_p = \frac{\alpha N}{N_1}$$

for some integer α.

Now by substituting this expression for m_1 into the equation for expressing a, the result follows.

$$\begin{aligned} a &= m_1 N_1 + 1 \\ &= \frac{\alpha N}{N_1} N_1 + 1 \\ &= \alpha N + 1 \\ &\Rightarrow \langle a \rangle_N = 1 \end{aligned}$$

B.2 Theorem 1

Consider p mutually prime numbers, N_1 to N_p, then

$$\langle \sum_{i=1}^{p} \overline{N}_i \langle \overline{N}_i^{-1} \rangle_{N_i} \rangle_N = 1$$

where $N = \prod_{i=1}^{p} N_i$ and $\overline{N}_i = \frac{N}{N_i}$

The proof is as follows. Consider the sum in brackets modulo N_j. All the terms in the sum except the j^{th} term have a factor of N_j and so contribute nothing.

$$\sum_{i=1}^{p} \overline{N}_i \langle \overline{N}_i^{-1} \rangle_{N_i} \rangle_{N_j} = \langle \overline{N}_j \langle \overline{N}_j^{-1} \rangle_{N_j} \rangle_{N_j} \equiv 1$$

Now the result follows by applying Lemma 1 from above.

B.3 Prime Factor Mapped Proof

In the four-dimensional prime factor case the mapping is of the form

$$n = \langle \sum_{i=1}^{4} \overline{N}_i n_i \rangle_N$$

and N_i are relatively prime. The inverse address mapping of this is given by

$$n_i = \langle \langle \overline{N}_i^{-1} \rangle_{N_i} n \rangle_{N_i} \qquad i = 1 \ldots 4$$

The proof proceeds by substituting the inverse into the original expression, giving

$$\begin{aligned}
n &= \langle \sum_{i=1}^{p} \overline{N}_i \langle \langle \overline{N}_i^{-1} \rangle_{N_i} n \rangle_{N_i} \rangle_N \\
&= \langle \sum_{i=1}^{p} \langle \langle \overline{N}_i^{-1} \rangle_{N_i} n \overline{N}_i \rangle_N \rangle_N \\
&= \langle n \sum_{i=1}^{p} \langle \overline{N}_i^{-1} \rangle_{N_i} \overline{N}_i \rangle_N \\
&= \langle \langle n \rangle_N \langle \sum_{i=1}^{p} \langle \overline{N}_i^{-1} \rangle_{N_i} \overline{N}_i \rangle \rangle_N \\
&= \langle \langle n \rangle_N \rangle_N \\
&= \langle n \rangle_N \\
&= n
\end{aligned}$$

B.4 Chinese Remainder Theorem Proof

The Chinese remainder theorem mapping is of the form

$$n = \langle \sum_{i=1}^{4} \overline{N}_i \langle \overline{N}_i^{-1} \rangle_{N_i} n_i \rangle_N$$

where N_i are relatively prime. The inversion is very simple.

$$n_i = \langle n \rangle_{N_i} \qquad i = 1 \ldots 4$$

The proof proceeds by substituting the inverse into the original expression, giving

$$
\begin{aligned}
n &= \Big\langle \sum_{i=1}^{p} \overline{N}_i \langle \overline{N}_i^{-1} \rangle_{N_i} \langle n \rangle_{N_i} \Big\rangle_N \\
&= \Big\langle \sum_{i=1}^{p} \langle \overline{N}_i^{-1} \rangle_{N_i} \langle \overline{N}_i n \rangle_N \Big\rangle_N \\
&= \Big\langle \sum_{i=1}^{p} \langle \langle \overline{N}_i^{-1} \rangle_{N_i} \langle \overline{N}_i n \rangle_N \rangle_N \Big\rangle_N \\
&= \Big\langle \sum_{i=1}^{p} \langle \overline{N}_i^{-1} \rangle_{N_i} \overline{N}_i n \Big\rangle_N \\
&= \Big\langle n \sum_{i=1}^{p} \langle \overline{N}_i^{-1} \rangle_{N_i} \overline{N}_i \Big\rangle_N \\
&= \langle n \rangle_N \\
&= n
\end{aligned}
$$

Appendix C

Operation Counts

C.1 Fundamental Sums

Many of the operation counts below are derived by using the following fundamental summations and theorems. The proofs of these results are straightforward.

$$\sum_{i=1}^{n} 1 = n$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n} i^2 = \frac{n(2n+1)(n+1)}{6}$$

$$\sum_{i=1}^{n} \left\lfloor \frac{i}{p} \right\rfloor = \frac{1}{p}\left[\frac{n(n+1)}{2} - \left\lfloor \frac{n}{p} \right\rfloor \frac{p(p-1)}{2} - \frac{(n \bmod p)(n \bmod p+1)}{2}\right]$$

$$\left\lfloor \frac{i}{p} \right\rfloor = \frac{i - i \bmod p}{p}$$

$$\left\lceil \frac{i}{p} \right\rceil = \frac{i+p-1-(i+p-1) \bmod p}{p}$$

C.2 Gauss-Jordan Elimination

C.2.1 Floating Point Operation Count

The number of floating point operations required for Gauss-Jordan elimination is based on counting the operations required by the following pseudo code.

```
for i = 1:n
        A(i, i:) = A(i, i:)/A(i,i)
        for j = 1:n
                if j ~= i
                        A(j,i:) = A(j,i:) - A(j,i)*A(i,i:)
                end
        end
end
```

Let the matrix A being reduced be $n \times m$. The length of the vector being manipulated at each point is $m - i + 1$. However the floating point operation occurring in the i^{th} column at each point is trivial; either dividing a number by itself, multiplying by one, or subtracting a number from itself. Thus counting only the truly necessary operations the vector length is $m - i$. The total number of flops is given by,

$$\begin{aligned}
&\sum_{i=1}^{n} (m-i) + 2(n-1)(m-i) \\
&= (2n-1)\sum_{i=1}^{n} m-i \\
&= (2n-1)\left(mn - \frac{n(n+1)}{2}\right) \\
&= 2mn^2 - mn - n^3 + \frac{n^2}{2} - n^2 + \frac{n}{2} \\
&= (2m-n)n^2 - (m+n/2)n + n/2
\end{aligned}$$

C.2.2 Exact Array Cycle Count

The result for the exact number of array cycles required by Gauss-Jordan elimination is obtained by beginning with equation (5.15) and substituting in the expression for the number of cycles required by addition from equation (5.1).

$$\text{GaussJordanRowCycles}(n, m, p) = \sum_{i=0}^{n-1} (3 + 2(n-1))\text{AC}(m-i, 1, i \bmod p, p)$$

$$= \sum_{i=0}^{n-1} (2n+1)\left\lceil \frac{m-i+i \bmod p}{p} \right\rceil$$

$$= (2n+1)\sum_{i=0}^{n-1} \frac{m-i+i \bmod p+p-1-(m-i+i \bmod p+p-1) \bmod p}{p}$$

$$= (2n+1)\sum_{i=0}^{n-1} \frac{m+p-1-(m+p-1) \bmod p}{p} - \frac{i-i \bmod p}{p}$$

$$= (2n+1)\sum_{i=0}^{n-1} \left\lceil \frac{m}{p} \right\rceil - \left\lfloor \frac{i}{p} \right\rfloor$$

$$= (2n+1)\left[n\left\lceil \frac{m}{p} \right\rceil - \right.$$

$$\left. \frac{1}{p}\left[\frac{n(n-1)}{2} - \left\lfloor \frac{n-1}{p} \right\rfloor \frac{p(p-1)}{2} - \frac{((n-1) \bmod p)((n-1) \bmod p+1)}{2} \right] \right]$$

C.2.3 Approximate Array Cycle Count

The result for the exact number of array cycles required by Gauss-Jordan elimination is obtained by beginning with equation (5.15) and substituting in the approximate expression for the number of cycles required by addition from equation (5.3).

$$\text{GaussJordanRowCyclesA}(n, m, p) = \sum_{i=0}^{n-1} (3 + 2(n-1))\text{AC}(m-i, 1, i \bmod p, p)$$

$$= \sum_{i=0}^{n-1} (2n+1)\frac{m-i}{p}$$

$$= \frac{1}{p}\sum_{i=0}^{n-1} (2mn+m) - (2n+1)i$$

$$= \frac{1}{p}\left[(2mn+m)n - (2n+1)\frac{n(n-1)}{2} \right]$$

$$= \frac{(2m-n)n^2}{p} + \frac{(m+n/2)n}{p} + \frac{n/2}{p}$$

C.3 Gaussian Elimination with Back Substitution

C.3.1 Floating Point Operation Count

The number of floating point operations required for Gauss-Jordan elimination with backsubstitution is based on counting the operations required by the following pseudo code.

```
for i = 1:n
        A(i, i:) = A(i, i:)/A(i,i)
        for j = i+1:n
                A(j,i:) = A(j,i:) - A(j,i)*A(i,i:)
        end
end
for i = n:-1:1
        for j = i:-1:1
                A(j,n+1:) = A(j, n+1:) - A(j,i)*A(i,n+1:)
        end
end
```

If the matrix A being reduced is $n \times m$ then the number of flops required by the reduction step is given by,

$$\sum_{i=1}^{n} (m-i+1) + 2(n-i)(m-i+1)$$

$$= \sum_{i=1}^{n} (2nm + 2n + m + 1) - (2n + 2m + 3)i + 2i^2$$

$$= (2nm + 2n + m + 1)n - (2n + 2m + 3)\frac{n(n+1)}{2} + \frac{2n(2n+1)(n+1)}{6}$$

$$= \left(m - \frac{n}{3}\right)n^2 + \frac{n^2}{2} - \frac{7n}{6}$$

The number of flops required by the backsubstitution step is given by,

$$\sum_{i=1}^{n} 2(n-i)(m-n)$$

$$= 2(m-n)\sum_{i=1}^{n} n - i$$

$$= 2(m-n)\left(n^2 - \frac{n(n+1)}{2}\right)$$

$$= (m-n)n^2 - (m-n)n$$

Adding these two results gives the total flops as,

$$(2m - 4n/3)n^2 - (m - n/2)n - n/6$$

C.3.2 Approximate Array Cycle Count

The approximate expression for the array cycles required by Gaussian Elimination with backsubstitution can be obtained by substituting the approximate expression for the array cycles required by addition, given in (5.3), into (5.19).

$$\begin{aligned}
\text{GaussRowCyclesA}(n, m, p) &= \sum_{i=0}^{n-1} (3 + 2(n - i - 1))\text{AC}(m - i, 1, i \bmod p, p) \\
&\quad + \sum_{i=0}^{n-2} (1 + 2(n - i - 1))\text{AC}(m - n, 1, n \bmod p, p) \\
&= \sum_{i=0}^{n-1} (2n + 1 - 2i)\frac{m - i}{p} + \sum_{i=0}^{n-2} (2n - 1 - 2i)\frac{m - n}{p} \\
&= \frac{1}{p}\sum_{i=0}^{n-1} (2mn + m) - (2m + 2n + 1)i + 2i^2 + \frac{1}{p}\sum_{i=0}^{n-2} (2mn - 2n^2 - m + n) - (2m - 2n)i \\
&= \frac{1}{p}\Big[(2mn + m)n - (2m + 2n + 1)\frac{n(n-1)}{2} + 2\frac{n(2n-1)(n-1)}{6} \\
&\qquad + (2mn - 2n^2 - m + n)(n - 1) - (2m - 2n)\frac{(n-1)(n-2)}{2}\Big] \\
&= \frac{(2m - 4n/3)n^2}{p} + \frac{(2m - n/2)n}{p} - \frac{m}{p} + \frac{11n/6}{p}
\end{aligned}$$

C.4 Gaussian Elimination Using Block Multiplication

The approximate expression for the array cycles required by Gaussian Elimination using block multiplication can be obtained by substituting the approximate expression for the array cycles required by addition(5.3), multiplication(5.7), copying by multiplication(5.10) and Gauss-Jordan Elimination(5.17), into (5.19).

To simplify the presentation, the reduction and backsubstitution parts will be presented separately. First the reduction step.

$$\text{GaussBlockCycles}(n, m, p, b, v) = \sum_{i=0}^{\frac{n}{b}-1} \begin{array}{l} \text{GJRC}(b, m-ib, p) + \text{AC}(m-bi-b, b, 0, p) \\ + \text{AC}(b, n-bi-b, 0, p) \\ + \text{MCC}(n-bi-b, b, 0, p) \\ + \text{AC}(m-bi-b, n-bi-b, 0, p) \\ + \text{MC}(m-bi-b, b, n-bi-b, 0, 0, p, v) \end{array}$$

$$= \frac{1}{p}\sum_{i=0}^{\frac{n}{b}-1} \begin{array}{l} (2(m-bi)-b)b^2 + (m-bi+b/2)b + \frac{b}{2} + (m-bi-b)b + b(n-bi-b) \\ + (n-bi-b)b + (m-bi-b)(n-bi-b) + \frac{(m-bi-b)(n-bi-b)}{pv} \end{array}$$

$$= \frac{1}{p}\sum_{i=0}^{\frac{n}{b}-1} \begin{array}{l} 2mb^2 + 2mb + 2nb - b^3 - \frac{5b^2}{2} + b - (4b^2 + 2b^3)i \\ + \left(1 + \frac{b}{pv}\right)(mn - mb - nb + b^2 - (mb + nb - 2b^2)i + b^2 i^2) \end{array}$$

$$= \frac{n}{2pb}\left[\begin{array}{l} 4mb^2 + 4mb + 4nb - 2b^3 - 5b^2 + b - (4b^2 + 2b^3)\frac{(n-b)}{b} \\ + 2mn - 2mb - 2nb + 2b - (mb + nb - 2b)\frac{(n-b)}{b} + b^2\frac{(n-b)}{3b}\frac{(2n-1)}{b} \end{array}\right]$$

$$\frac{n}{2p^2v}\left[2mn - 2mb - 2nb + 2b - (mb + nb - 2b)\frac{(n-b)}{b} + b^2\frac{(n-b)}{3b}\frac{(2n-1)}{b}\right]$$

$$= \frac{n}{2pb}\left[nm - \frac{n^2}{3} + 4mb^2 + 3mb - 2nb^2 - \frac{2b^2}{3} + b\right] + \frac{n}{2p^2v}\left[nm - \frac{n^2}{3} - mb + \frac{b^2}{3}\right]$$

Now the backsubstitution step.

$$\text{GaussBlockCycles}(n, m, p, b, v)$$

$$= \sum_{i=1}^{\frac{n}{b}-1} \begin{array}{l} \text{AC}(b, bi, 0, p) + \text{MCC}(b, bi, 0, p) + \text{AC}(m-n, b, n \bmod p, p) \\ + \text{MC}(m-n, b, bi, n \bmod p, 0, p, v) + \text{AC}(m-n, bi, n \bmod p, p) \end{array}$$

$$= \frac{1}{p}\sum_{i=1}^{\frac{n}{b}-1} b^2 i + b^2 i + (m-n)b + \frac{(m-n)b^2 i}{vp} + (m-n)bi$$

$$= \frac{1}{p}\sum_{i=1}^{\frac{n}{b}-1} (mb - nb) + (mb - nb + 2b^2)i + \frac{(mb^2 - nb^2)}{vp} i$$

$$= \frac{n}{2bp}\left[(2mb - 2nb)\left(1 - \frac{b}{n}\right) + (mb - nb + 2b^2)\frac{(n-b)}{b} + \frac{(mb^2 - nb^2)}{vp}\frac{(n-b)}{b}\right]$$

$$= \frac{n}{2bp}[mn - n^2 + mb + nb] + \frac{n}{2vp^2}[mn - n^2 - mb + nb] - \frac{mb}{p}$$

Adding these two results gives the total number of cycles required.

$$\frac{n}{2pb}\left[2nm - \frac{4n^2}{3} - 2nb^2 + nb + 4mb^2 + 4mb - \frac{2b^2}{3} + b\right]$$
$$+ \frac{n}{2p^2v}\left[2nm - \frac{4n^2}{3} - 2mb + nb + \frac{b^2}{3}\right] - \frac{mb}{p}$$

C.5 Gaussian Elimination Using Multiplication by the Inverse

The approximate expression for the array cycles required by Gaussian Elimination using multiplication by the inverse can be obtained by substituting the approximate expression for the array cycles required by addition(5.3), multiplication(5.7), copying by multiplication(5.10) and Gauss-Jordan Elimination(5.17), into (5.23).

$$\text{GaussInvertCycles}(n, m, p, v) = \begin{array}{l}\text{GJRC}(n, 2n, p) + \text{AC}(n, n, 0, p) + \text{MCC}(n, n, 0, p) \\ \qquad + \text{MC}(n, n, m-n, 0, 0, p, v)\end{array}$$

$$= \frac{(2(2n)-n)n^2}{p} + \frac{(2(2n)+n)n}{p} + \frac{n}{2p} + \frac{n}{p} + \frac{n}{p} + \frac{n(m-n)}{p^2v}$$

$$= \frac{mn^2 - n^3 + 3vpn^3}{p^2v} + \frac{9n^2}{2p} + \frac{n}{2p}$$

Appendix D

Masm Code for Fourier Transform

The following is the Masm code for the three dimensional prime factor mapped Fourier Transform.

```
%
%  PFM3.masm
%

integer virtual; virtual = 1;

% Temporary integer variable
integer temp;
alias Xstore Xview;
alias Ystore Yview;

% W1r, W1i are the real and imaginary parts of the first weight matrix
matrix main W1r_main;
matrix main W1i_main;
load W1r_main 'W1r.mat';
load W1i_main 'W1i.mat';
matrix Xstore W1r(100, 100);
matrix Xstore W1i(100, 100);
W1r = W1r_main;
W1i = W1i_main;

% W2r, W2i are the real and imaginary parts of the second weight matrix
matrix main W2r_main;
matrix main W2i_main;
load W2r_main 'W2r.mat';
load W2i_main 'W2i.mat';
matrix Ystore W2r(100, 100);
matrix Ystore W2i(100, 100);
W2r = W2r_main;
W2i = W2i_main;

% W3r, W3i are the real and imaginary parts of the second weight matrix
matrix main W3r_main;
matrix main W3i_main;
load W3r_main 'W3r.mat';
load W3i_main 'W3i.mat';
```

```
matrix Ystore W3r(100, 100);
matrix Ystore W3i(100, 100);
W3r = W3r_main;
W3i = W3i_main;

% Calculate the factors and strides for the mapped matrix
% in the X and Y registers
integer N1; N1 = W1r.width;
integer N2; N2 = W2r.width;
integer N3; N3 = W3r.width;
integer N;  N  = N1 * N2; N = N * N3;

integer pv; pv = .p * virtual;
integer X_stride;
X_stride = N1 + pv;
X_stride = X_stride - 1;
X_stride = X_stride/pv;
X_stride = X_stride * virtual;

integer Y_stride;
Y_stride = N2 + pv;
Y_stride = Y_stride - 1;
Y_stride = Y_stride/pv;
Y_stride = Y_stride * virtual;

integer X_N3_stride;
X_N3_stride = N2 + pv;
% X_N3_stride = X_N3_stride - 1; Allow extra room for result overwrite
X_N3_stride = X_N3_stride/pv;
X_N3_stride = X_N3_stride * pv;
X_N3_stride = X_N3_stride * X_stride;

integer Y_N3_stride;
Y_N3_stride = N1 + pv;
Y_N3_stride = Y_N3_stride - 1;
Y_N3_stride = Y_N3_stride/pv;
Y_N3_stride = Y_N3_stride * pv;
Y_N3_stride = Y_N3_stride * Y_stride;


% Calculate the inverse of N2*N3 modulo N1 (delta1)
integer test;
integer delta1; delta1 = 1;
Delta1Loop:;
test = delta1 * N2;
test = test * N3;
temp = test / N1;
temp = temp * N1;
test = test - temp;
if test == 1 goto Delta1found;
delta1 = delta1 + 1;
if delta1 >= N1 goto Error;
goto Delta1Loop;
Delta1found:;

% Calculate the inverse of N3*N1 modulo N2 (delta2)
integer delta2; delta2 = 1;
```

```
Delta2Loop:;
test = delta2 * N3;
test = test * N1;
temp = test / N2;
temp = temp * N2;
test = test - temp;
if test == 1 goto Delta2found;
delta2 = delta2 + 1;
if delta2 >= N2 goto Error;
goto Delta2Loop;
Delta2found:;

% Calculate the inverse of N1*N2 modulo N3 (delta3)
integer delta3; delta3 = 1;
Delta3Loop:;
test = delta3 * N1;
test = test * N2;
temp = test / N3;
temp = temp * N3;
test = test - temp;
if test == 1 goto Delta3found;
delta3 = delta3 + 1;
if delta3 >= N3 goto Error;
goto Delta3Loop;
Delta3found:;

% X is the data input matrix it is loaded from the file pfm.mat
matrix main X_main;
load X_main 'pfm.mat';
temp = X_main.width * X_main.depth;
if N ~= temp goto Error;

% Load X using a prime factor map
matrix Xstore Xr(100, 100);
matrix Xstore Xi(100, 100);
.LSLength = X_main.width * X_main.depth;
.Delta1 = delta1;
.Sigma1 = 1;
.Q1     = N1;
temp = X_stride * .p;
.Delta2 = delta2 * temp;
.Sigma2 = 1;
.Q2     = N2 * temp;
temp = X_N3_stride * .p;
.Delta3 = delta3 * temp;
.Sigma3 = 1;
.Q3     = N3 * temp;
.Delta4 = 0; .Sigma4 = 1; .Q4 = 1;
.Saddress = Xr.base * .p;
.Stride    = 1;
.Runlength = X_main.width;
.Increment = X_main.width;
.Daddress  = X_main.base;
call .LoadX;

% View the mapped matrix
Xview.base = Xr.base;
```

```
Xview.stride = 1;
Xview.offset = 0;
Xview.width = 5;
Xview.depth = 12;
%print Xview;


% The Y matrix is the place where the data goes in the Y register
matrix Ystore Yr(100, 100);
matrix Ystore Yi(100, 100);

% Setup the aliases to the X amd Y mapped bits

alias Xstore Xralias;
alias Xstore Xialias;
alias Ystore Yralias;
alias Ystore Yialias;

Xralias.base   = Xr.base;
Xralias.stride = X_stride;
Xralias.offset = 0;
Xralias.width  = N1;
Xralias.depth  = N2;
Xialias.base   = Xi.base;
Xialias.stride = X_stride;
Xialias.offset = 0;
Xialias.width  = N1;
Xialias.depth  = N2;

Yralias.base   = Yr.base;
Yralias.stride = Y_stride;
Yialias.base   = Yi.base;
Yialias.stride = Y_stride;

integer zero; zero = 0;

%
% Multiply by the W2 weight matrix
%

integer i;

i = 0;
W2Loop:;
    param Xralias;
    param Xialias;
    param W2r;
    param W2i;
    param Yralias;
    param Yialias;
    param zero;
    param virtual;
    call  .T_complex_X_Y_Y_virtual;

    Xralias.base = Xralias.base + X_N3_stride;
    Xialias.base = Xialias.base + X_N3_stride;
    Yralias.base = Yralias.base + Y_N3_stride;
```

```
    Yialias.base = Yialias.base + Y_N3_stride;

    i = i + 1;
if i < N3 goto W2Loop;

%
%  Multiply by the W1 matrix
%

Xralias.base = Xr.base;
Xialias.base = Xi.base;
Yralias.base = Yr.base;
Yialias.base = Yi.base;

i = 0;
W1Loop:;
    param W1r;
    param W1i;
    param Yralias;
    param Yialias;
    param Xralias;
    param Xialias;
    param zero;
    param virtual;
    call  .T_complex_X_Y_X_virtual;

    Xralias.base = Xralias.base + X_N3_stride;
    Xialias.base = Xialias.base + X_N3_stride;
    Yralias.base = Yralias.base + Y_N3_stride;
    Yialias.base = Yialias.base + Y_N3_stride;

    i = i + 1;
if i < N3 goto W1Loop;

%print Xview;
Xview.base = Xi.base;
%print Xview;

%
%  Multiply by the W3 matrix
%

temp = N2 - 1;
temp = temp * X_stride;
Xralias.base = Xr.base + temp;
Xialias.base = Xi.base + temp;

Xralias.stride = X_N3_stride;
Xralias.depth  = N3;
Xialias.stride = X_N3_stride;
Xialias.depth  = N3;

alias Xstore Rralias;
alias Xstore Rialias;
Rralias.base   = Xralias.base + X_stride;
Rralias.stride = Xralias.stride;
Rialias.base   = Xialias.base + X_stride;
```

```
Rialias.stride = Xialias.stride;

i = 0;
W3Loop:;
    param Xralias;
    param Xialias;
    param W3r;
    param W3i;
    param Rralias;
    param Rialias;
    param zero;
    param virtual;
    call  .T_complex_X_Y_X_virtual;

    Xralias.base = Xralias.base - X_stride;
    Xialias.base = Xialias.base - X_stride;
    Rralias.base = Rralias.base - X_stride;
    Rialias.base = Rialias.base - X_stride;

    i = i + 1;
if i < N2 goto W3Loop;

Xview.base = Xr.base;
%print Xview;
Xview.base = Xi.base;
%print Xview;


%
% Print the final result
%
.LSLength = N * 2;
.Delta1 = delta1;
.Sigma1 = 1;
.Q1     = N1;
temp = X_stride * .p;
.Delta2 = delta2 * temp;
.Sigma2 = 1;
.Q2     = N2 * temp;
temp = X_N3_stride * .p;
.Delta3 = delta3 * temp;
.Sigma3 = 1;
.Q3     = N3 * temp;
temp = Xi.base - Xr.base;
temp = temp * .p;
.Delta4 = temp; .Sigma4 = N; .Q4 = 1000000;
temp = Xr.base + X_stride;
.Saddress = temp * .p;
call .PrintX;

goto End;
Error:;

End:;
```

Bibliography

[Agarwal *et al.* 94]R.C. Agarwal, F.G. Gustavson and M. Zubair, "Improving performance of linear algebra algorithms for dense matrices, using algorithmic prefetch", IBM Journal of Research and Development, Vol. 38, No. 3, May 1994, pp. 265-275.

[Agerwala and Karp 84]T. K. M. Agerwala and A. H. Karp, "Architecture for Vector Mask Registers", IBM Technical Disclosure Bulletin, Vol. 27, No. 7A, December 1984, pp. 4061-4063.

[Amdahl 67] Gene H. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities", Proceedings AFIPS Spring Joint Computer Conference 30, April 1967, pp. 483-485.

[Anderson *et al.* 92]E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov and D. Sorensen, "LAPACK Users Guide", SIAM, May 1992.

[Anderson *et al.* 97]Ed Anderson, Jeff Brooks, Charles Grassel and Steve Scott, "Performance of the CRAY T3E Multiprocessor", http://www.cray.com/products/systems/crayt3e/1200/performance.html, to appear in Procedings of Supercomputing 97.

[Aho *et al.* 86]Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman, "Compilers - Principles, Techniques, and Tools", Addison-Wesley, 1986.

[Baer and Chen 94]Jean-Loup Baer and Tien-Fu Chen, "Evaluation of Hardware and Software Prefetching", IFIP Transactions A: Computer Science and Technology, No A-44, April 1994, pp. 257-266.

[Beaumont-Smith 95]Andrew Beaumont-Smith, "Gallium Arsenide Design Methodology and Testing of a Systolic Floating Point Processing Element", Masters Thesis, University of Adelaide, 1995.

[Beaumont-Smith *et al.* 96]A. Beaumont-Smith, W. Marwood, C.C. Lim and K. Eshraghian, "Design and Implementation of a GaAs Systolic Floating Point Processing Element", IEE Proceedings - Computers and Digital Techniques, Vol. 143, No.5, September 1996.

[Beaumont-Smith *et al.* 97a]A. Beaumont-Smith, J. Tsimbinos, W. Marwood, C.C. Lim, M. Liebelt and K. To, "A Matrix Processor Implementation of the Voltera Model", Proceedings of the 2nd Australian Workshop on Signal Processing Applications, Brisbane, Australia, December 1997, pp. 195-198.

[Beaumont-Smith *et al.* 97b]A. Beaumont-Smith, M. Leibelt, C.C. Lim, K.To and W.Marwood, "A Digital Signal Multi-Processor for Matrix Applications", Proceedings of the 14th Microelectronics Conference, Melbourne, Australia,

September 1997, pp. 245-250.

[Bilmes *et al.* 97]Jeff Bilmes, Krste Asanovic, Chee-Whye Chin and Jim Demmel, "Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology", Proceedings of the 1997 International Conference on Supercomputing July 7-11 1997.

[Carr and Lehoucq 97]Steve Carr, R.B. Lehoucq, "Compiler blockability of dense matrix factorizations", ACM Transactions on Mathematical Software, Vol. 23, No. 3, September 1997.

[Choi *et al.* 94a]Jaeyoung Choi, Jack J. Dongarra and David W. Walker, "Pumma: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers", Concurrency Practice and Experience, Vol. 6, No. 7, October 1994.

[Choi *et al.* 94b]Jaeyoung Choi, Jack J. Dongarra and David W. Walker, "PB-BLAS: A set of parallel block basic linear algebra subprograms", Proceedings of the Scalable High-Performance Computing Conference May 23-25 1994.

[Choi *et al.* 96a]Jaeyoung Choi, Jack J. Dongarra, L. Susan Ostrouchov, Antoine P. Petitet, David W. Walker and R. Clint Whaley, "Design and Implementation of the ScaLAPACK LU, QR and Cholesky Factorisation Routines", Scientific Programming, Vol. 5, 1996, pp.173-184.

[Choi *et al.* 96b]Jaeyoung Choi, Jack J. Dongarra and David W. Walker, "PB-BLAS: a set of parallel block basic linear algebra subprograms", Concurrency Practice and Experience, Vol. 8, No. 7, September 1996.

[Chtchelkanova *et al.* 97]Almadena Chtchelkanova, John Gunnels, Greg Morrow, JamesOverfelt and Robert A. Van De Geijn, "Parallel implementation of BLAS: general techniques for level 3 BLAS", Concurrency Practice and Experience, Vol. 9, No. 9, September 1997.

[Clarke *et al.* 92]R.J. Clarke, I.A.Curtis, W. Marwoord and A.P. Clarke, "A Floating Point Matrix Arithmetic Processor: A Implementation of the SCAP Concept", Asia-Pasicific Conference on Circuits and Systems, December 1992, pp. 519-524.

[Crisp 97] Richard Crisp, "Direct Rambus Technology: The New Main Memory Standard", IEEE Micro, Vol. 17, No. 6, November/December 1997.

[DEC 97] "Digital Semiconductor 21164 Alpha Microprocessor Hardware Reference Manual", Digital Equipment Corporation, EC-QP99B-TE, February 1997.

[Dongarra *et al.* 79]J.J. Dongarra, J.R. Bunch, C.B.Moler and G.W. Stewart, "LINPACK Users' Guide", SIAM, 1979.

[Dongarra *et al.* 88]J.J Dongarra, J, Du Croz, S. Hammarling and I. Duff, "A set of level 3 basic linear algebra subprograms", ACM Transactions on Mathematical Software, Vol. 18, 1990, pp. 1-17.

[Dongarra *et al.* 93]Jack J. Dongarra, Hans W. Meuer and Erich Strohmaier, "Top 500 Report", June 1993, available at http://www.top500.org/reports/1993/report93.ps.

[Dongarra and Walker 95]Jack J. Dongarra and David W. Walker, "Software libraries for linear algebra computations on high performance computers", SIAM Review, Vol. 37, No. 2, June 1995.

[Dongarra *et al.* 98]Jack J. Dongarra, Hans W. Meuer and Erich Strohmaier, "Top 500 Supercomputer Sites", 11th edition, June 1998.

[Dongarra 98] Jack J. Dongarra, "Performance of Various Computers Using Standard Linear Equations Software", http://www.netlib.org/benchmark/performance.ps, August 12, 1998.

[Duff *et al.* 97]IIain S. Duff, Michele Marrone, Giuseppe Radicati, Carlo Vittoli, "Level 3 basic linear algebra subprograms for sparse matrices: A user-level interface", ACM Transactions on Mathematical Software, Vol. 23, No. 3, September 1997.

[Elman and Lee 95]Howard C. Elman and Dennis K.-Y. Lee, "Use of linear algebra kernels to build an efficient finite element solver", Parallel Computing, Vol. 21, No. 1, January 1995.

[Forsman *et al.* 95]Kimmo Forsman, William Gropp, Lauri Kettunen, David Levine and Jukka Salonen, "Solution o Dense System of Linear Equations Arising from Integral-Equation Formulation", IEEE Antennas and Propagation Magazine, Vol. 37, No. 6, December 1995, pp. 96-100.

[Flynn 66] Flynn, M.J., "Very High Speed Computing Systems", Proceedings IEEE, Vol. 54, No. 12, December 1966, pp.1901-1909.

[Fujii *et al.* 97]Hiroaki Fujii, Yoshiko Yasuda, Hideya Akashi, Yasuhiro Inagami, Makoto Koga, Osamu Ishihara, Masamori Kashiyama, Hideo Wada and Tsutomu Sumimoto, "Architecture and performance of the Hitachi SR2201 Massively Parallel Processor System", Proceedings of the 1997 11th International Parallel Processing Symposium, IPPS 97, April 1-5 1997, pp. 233-241.

[Golub and Van Loan 96]Gene H. Golub and Charles F. Van Loan, "Matrix Computations", John Hopkins, 1996.

[Gustafson 88]John L. Gustafson, "Reevaluating Amdahl's Law", Communications of the ACM, Vol. 31, No. 5, May 1988, pp. 532-533.

[Hennessy and Patterson 96]Hennessy, John L. and Patterson, David A., "Computer

Architecture: A Quantitative Appraoch", Morgan Kauffman, 1996.

[Huss-Lederman *et al.* 94]Huss-Lederman, S., Jacobson, E.M., Tsao, A., and Zhang, G., "Matrix multiplication on the Intel Touchstone Delta", Concurrency: Practice and Experience, Vol. 6, No. 7, October 1994, pp. 571-594.

[IBM 95] IBM Preliminary Data Sheet for IBM43610QLA, IBM Corporation, 1995.

[IBM 97] "PowerPC 750 RISC Microprocessor Technical Summary", IBM, August 1997.

[Intel 97] "Pentium II Processor Developer's Manual", Intel Corporation, 243502-001, October 1997.

[Jones *et al.* 92]Fred Jones, Betty Prince, Roger Norwood, Joe Hartigan, Wilbur C. Vogley, Charles A. Hart and David Bondurant, "Memory--A new era of fast dynamic RAMs", IEEE Spectrum, Vol. 29, No. 10, October 1992.

[Kung 88] Kung, S.Y., "VLSI Array Processors", 1988.

[Kung and Leiserson 79]Kung, H.T. & Leiserson, C. E., "Systolic Arrays (for VLSI)", Sparse Matrix Proceedings 1978, SIAM, 1979, pp. 256-282.

[Kung 82] Kung, H.T., "Why Systolic Architectures?", Computer, Vol. 15, No. 1, January 1982, pp. 37-46.

[Leiserson 85]Charles E. Leiserson, "Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing", IEEE Transactions on Computers, Vol. C-34, No. 10, October 1985, pp. 892-901.

[Levine *et al.* 91]David Levine, David Callahan and Jack Dongarra, "Comparative study of automatic vectorizing compilers", Parallel Computing, Vol. 17, No. 10-11, December 1991, pp. 1223-1244.

[Luecke *et al.* 91]Glenn Luecke, Waqar Haque, James Hoekstra, Howard Jespersen and James Coyle, "Evaluation of fortran vector compilers and preprocessors", Software - Practice and Experience, Vol. 21, No. 9, September 1991, pp. 891-905.

[Marwood 94]Marwood, Warren, "An Integrated Multiprocessor for Matrix Algorithms", PhD Thesis, University of Adelaide, 1994.

[Mattson *et al.* 96]Timothy G. Mattson, David Scott and Stephen Wheat, "TeraFLOP supercomputer in 1996: the ASCI TFLOP system", Proceedings of the 1996 10th International Parallel Processing Symposium, April 15-19 1996, pp. 84-93.

[McCalpin 95]John D. McCalpin, "Memory Bandwidth and Machine Balance in Current High Performance Computers", IEEE Computer Society Technical Committee

on Computer Architecture (TCCA) Newsletter, December 1995.

[Patterson and Ditzel 80]D. A. Patterson and D. R. Ditzel, "The case for the reduced instruction set computer", Computer Architecture News, Vol. 8, No. 6, October 1990, pp. 25-30.

[Press *et al.* 92]William H. Press, Saul A. Teukolsky, William T. Vetterling and Brian P. Flannery, "Numerical Recipes in C: The Art of Scientific Computing", 2nd Edition, 1992.

[Rajagopalan 98]Dilip Rajagopalan, "Use of high level basic linear algebra subprograms (BLAS) in viscolelastic flow simulations", Journal of Non-Newtonian Fluid Mechanics, Vol. 74, No. 1-3, January 1998.

[Rambus 95] 64-Megabit DRAM Press Release, Rambus Inc., Mountain View, California.

[Rambus 97] "Direct Rambus Technology Disclosure", Rambus Inc, available at http://www.rambus.com/html/drtechov.pdf.

[Shaw 95] Shaw, Tim, "No Excess Babbage - Design Considerations for the Interface to a Systolic Matrix Processor", Masters Thesis, Univeristy of Adelaide, 1995.

[Tanaka *et al.* 90]Yoshikazu Tanaka, Kyouko Iwasawa and Yukio Umetani, "Compiling techniques for first-order linear recurrences on a vector computer ", Journal of Supercomputing, Vol. 4, No. 1, March 1990, pp. 63-82.

[Taylor 95] Valerie E. Taylor, Abhiram Ranade and David G. Messerschmitt, "SPAR: A New Architecture for Large Finite Element Computations", IEEE Transactions on Computers, Vol. 44, No. 4, April 1995, pp. 531-545.

[To 96] Kiet To, Personal Communication.

[Tsay and Chang 95]Jong-Chuang Tsay and Pen-Yuang Chang, "Some New Designs of 2-D Array for Matrix Multiplication and Transitive Closure", IEEE Transactions on Parallel and Distributed Systems, Vol. 6, No. 4, April 1995, pp. 351-362.

[Whitehouse and Speiser 81]H.J. Whitehouse and J.M. Speiser, "SONAR Applications of Systolic Array Technology", Conference Record, IEEE EASTCON, Washington DC, November 17-19, 1981.

[Wong *et al.* 95]W.F. Wong, Yoshio Oyanagi and Eiichi Goto, "Evaluation of the Hitachi S-3800 Supercomputer Using Six Benchmarks", The International Journal of Supercomputer Applications, Vol. 9, No. 1, Spring 1995, pp. 58-70.

[Wulf 81] Wulf, William A., "Compilers and Computer Architecture", Computer, Vol. 14, No. 7, July 1981.

[Yoo *et al.* 96] Jei-Hwna Yoo, Chang-Hyun Kim, Kyu-Chan Lee, Kye-Hyun Kyung, Seung-Moon Yoo, Jung-Hwa Lee, Moon-Hae Son, Jin-Man Han, Bok-Moon Kang, Ejaz Haq, Sang-Bo Lee, Jai-Hoon Kim, Byung-Sik Moon, Keum-Yong Kim, Jae-Gwan Park, Ku-Phil Lee, Kang-Yoon Lee, Ki-Nam Kim, Soo-In Cho, Jong-Woo Park and Hyung-Kyu Lim, "A 32-Bank 1Gb DRAM with 1GB/s Bandwidth", pp.378-379, 1996 IEEE International Solid-State Circuit Conference Digest of Technical Papers - Volumn(39) XXXIX

[Yu 96] Albert Y.C. Yu, "Future of microprocessors", IEEE Micro, Vol. 16, No. 6, December 1996, pp. 46-53.