MATRIX MULTIPLICATION ON LUCAS

Lennart Ohlsson
Bertil Svensson

Department of Computer Engineering
University of Lund
P.O. Box 725
220 07 LUND, SWEDEN

Abstract

Multiplication of two N by N matrices involves N^3 multiplications of elements. The task allows a large amount of parallelism to be utilized, indicating that it can be efficiently executed on a parallel computer. This paper describes how matrix multiplication is performed on LUCAS, an SIMD type parallel processor with bit-serial processing elements. The interconnection network is of Perfect Shuffle/Exchange type. The case of study is when the number of processing elements is between N^2 and N^3. The algorithm presented can be applied to any computer with the same interconnection structure. Formulas showing how the execution time depends on data length and matrix size are presented together with measured values from execution on LUCAS.

1. Introduction

LUCAS (Lund University Content Addressable System) is a machine of SIMD type using bit-serial processing elements (PEs). It was designed and implemented at the Department of Computer Engineering at the University of Lund [Sve]. The main purpose of the project was to provide a research tool for investigation of applicability of this kind of computer organization. As part of the project a high level language, Pascal/L, has been developed [Fernl], and LUCAS is presently being evaluated in a number of application areas [Kru1], [Kru2], [Ohl], [Sve].

A critical detail in the design of any SIMD processor is the interconnection network between memory units and processing elements (PEs). Perfect Shuffle/Exchange is generally regarded as one of the most useful structures. For some algorithms it is the optimal one. Perfect Shuffle/Exchange has been chosen as general purpose interconnection network on LUCAS.

We have studied multiplication of N by N matrices on LUCAS for different values of N. Of special interest is the case when the number of PEs is large compared to the matrix size (between N^2 and N^3 PEs). In this case very large gains in execution time can be obtained.

In the next section we describe those properties of LUCAS that are relevant to the solution of the problem; i.e. the processing elements and the interconnection structure.

In section 3 we give a matrix multiplication algorithm for the case when the number of PEs is equal to N^2. This is then generalized in section 4 to $M*N^2$ PEs, $1 \leq M \leq N$ and M is a power of two.

2. The LUCAS Processor Array

Associative parallel processors have two characteristic features: (1) Memory cells are accessed according to their contents, rather than by fixed addresses, and (2) data is processed in parallel in a selected subset of memory words [Fos].

The architecture of LUCAS is shown in Figure 1. It consists of three parts: a Host system, a Control Unit and an Associative Array.

Data resides in the Associative Array, which consists of 128 identical processing elements. The working mode is bit-serial word-parallel, which means that one bit of each word is processed at the same time. A number of consecutive bitslices is called a <u>field</u>. A field contains a vector of data-items, and the location of a specific data-item is uniquely given by its field address and the memory word number, called <u>index</u>. E.g. A(i) refers to the data-item in field A, word number i.

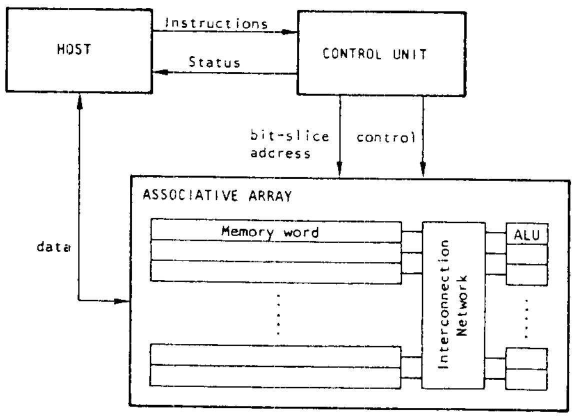


Figure 1. LUCAS

A PE is shown in Figure 2. It consists of an arithmetic logic unit (ALU), capable of performing 32 different functions on five one-bit inputs. The PE has two 1-bit registers, T(tag), R(result), C (carry) and X(auxilliary). When writing into the memory, data from the R register is used. The output of the Tag register is connected to the write control logic of the associatied memory word. It is used for activation control by inhibiting change in the memory of those PEs where T is zero.

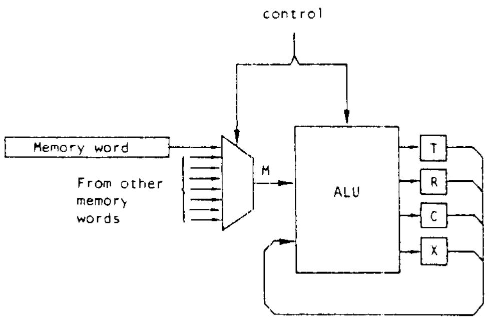


Figure 2. A processing element in LUCAS

All the one-bit outputs from the memory words are gathered on a 128-bit wide bus, accessible from all PEs over an interconnection network. Through a strapping area for each PE, eight of these are chosen. This choice defines the topology of the Associative Array. One of the inputs is fixed to be the output of the PEs own memory word.

An interconnection structure that provides generality, in the sense that arbitrary permutations of the 128-bit vector can be performed in a small number of steps, is the Perfect Shuffle/Exchange network [Ston], [Park]. It occupies two of the seven available inputs of each PE, one for the Perfect Shuffle connection and one for the Perfect Shuffle + Exchange connection. The effects of Perfect Shuffle (PS) and Perfect Shuffle + Exchange (XS) are shown in Figure 3.

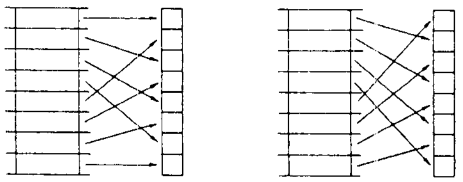


Figure 3. Perfect Shuffle (PS) and Perfect Shuffle + Exchange (XS)

The Host, which is interchangeable, is an ordinary mini- or microcomputer. At present a Z80 based system is used. During execution it sends instructions to the Control Unit and handles input and output of data to the Associative Array. The Control Unit executes the instructions in the form if microprograms, sending the same microinstructions to all the PEs in the Associative Array.

3. Matrix multiplication algorithm, N^2 PEs

We now present an algorithm for matrix multiplication on a LUCAS-type machine. This section is organized as follows: First we make some basic assumptions and definitions. Then we study the effects of transferring a matrix over the interconnection network of LUCAS. Next comes a detailed description of the algorithm, which can be divided into four distinct phases: Pre-alignment, Multiplication, Summation and Post-alignment. Finally we give a short summary of the algorithm in a form that is suitable for the coming generalization in section 4.

3.1 Basic assumptions and definitions

Throughout this section we assume that the matrices to be multiplied, A and B, both are of size N by N elements, where $N=2^n$ for an integer n. We further assume that the number of PEs is equal to N^2 (LUCAS, with its 128 PEs does not suit into this scheme).

When we illustrate different steps of the algorithm, we use small values of n - typically 2, 3 or 4 - without explicitly indicating each time that a generalization to any n can be made.

We assume that the matrices A and B are stored in two fields of the associative array in row-major order, and that the result matrix, C, also is to be stored in this way (Figure 4a). Element a_{ij} is then located in field A, memory word i*N+j. We will refer to this location as A(i,j) or explicitly $A<i_{n-1}i_{n-2}\ldots i_1 i_0 j_{n-1}j_{n-2}\ldots j_1 j_0>$, where $<i_{n-1}i_{n-2}\ldots i_1 i_0>$ and $<j_{n-1}j_{n-2}\ldots j_1 j_0>$ are the binary representations of i and j respectively.

We can now state the initial conditions as:

$$A(k,i)=a_{ki} \qquad B(i,j)=b_{ij}$$

The elements of C are computed from the elements of A and B according to the formula:

$$c_{kj} = \sum_{i=0}^{N-1} a_{ki} * b_{ij}$$

We want to obtain:

$$C(k,j)=c_{kj}$$

A	PS(A)	XS(A)
a_{00}	a_{00}	a_{20}
a_{01}	a_{20}	a_{00}
a_{02}	a_{01}	a_{21}
a_{03}	a_{21}	a_{01}
a_{10}	a_{02}	a_{22}
a_{11}	a_{22}	a_{02}
a_{12}	a_{03}	a_{23}
a_{13}	a_{23}	a_{03}
a_{20}	a_{10}	a_{30}
a_{21}	a_{30}	a_{10}
a_{22}	a_{11}	a_{31}
a_{23}	a_{31}	a_{11}
a_{30}	a_{12}	a_{32}
a_{31}	a_{32}	a_{12}
a_{32}	a_{13}	a_{33}
a_{33}	a_{33}	a_{13}
(a)	(b)	

Figure 4. A Matrix A stored in row-major order (a) and the effects of the PS and XS operations (b).

Requiring that the matrices are to be stored in row-major order is no limitation. This is because working with matrices stored in column-major means, if we "think" row-major, to work with transposed matrices. Since:

$$A*B=C \iff B^T*A^T=C^T$$

we only need to change order of A and B in the column-major case.

3.2 Shuffling a matrix

Applying the operators PS and XS to a matrix:

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix}$$

stored in row-major form, yields the following result matrices (see Figure 4b):

$$PS(A) = \begin{bmatrix} a_{00} & a_{20} & a_{01} & a_{21} \\ a_{02} & a_{22} & a_{03} & a_{23} \\ a_{10} & a_{30} & a_{11} & a_{31} \\ a_{12} & a_{32} & a_{13} & a_{33} \end{bmatrix}$$

$$XS(A) = \begin{bmatrix} a_{20} & a_{00} & a_{21} & a_{01} \\ a_{22} & a_{02} & a_{23} & a_{03} \\ a_{30} & a_{10} & a_{31} & a_{11} \\ a_{32} & a_{12} & a_{33} & a_{13} \end{bmatrix}$$

As shown by Stone, $\log_2 N$ perfect shuffles of a matrix stored in this form, results in the matrix being transposed [Ston]. (In the case illustrated: $PS^2(A)=A^T$.) This can be shown in the following way:

Consider an arbitrary element a_{ij}. It is initially stored in memory word $<i_{n-1}i_{n-2}\ldots i_1 i_0 j_{n-1} j_{n-2}\ldots j_1 j_0>$. The PS operator brings this element to word $<i_{n-2}\ldots i_1 i_0 j_{n-1}\ldots j_1 j_0 i_{n-1}>$. After n perfect shuffles, element a_{ij} will arrive at word $<j_{n-1}j_{n-2}\ldots j_1 j_0 i_{n-1} i_{n-2}\ldots i_1 i_0>$, i.e. the matrix A is transposed:

$$PS^n(A) = A^T$$

Similarly, the XS operator brings the element a_{ij} to word $<i_{n-2}\ldots i_1 i_0 j_{n-1}\ldots j_1 j_0 \bar{i}_{n-1}>$, where $\bar{i}_1$ denotes the binary complement of i_1.

Transposition of matrices is one operation that is required in order to perform efficient matrix multiplication on LUCAS. Other kinds of matrix transformations are also needed in order to align the elements for multiplication. We will next turn our attention to the nature of these alignments, which constitute the first phase of the algorithm.

3.3 Pre-alignment

With N^2 processors we have the potential to simultaneously perform all multiplications required to compute an entire row of the result matrix. To compute row number k of C, row number k of A has to be aligned with every column of B. Let $A^{(k)}$ denote the matrix that has as its columns, row number k of A. This can be expressed:

$$A^{(k)}(i,j) = a_{ki}$$

Formation of $A^{(k)}$ from A means broadcasting of N elements (row number k of A) to all N^2 memory words according to a certain pattern.

We define two additional operators that can be applied to a field of the memory (the result field need not be the same as the source field).

Broadcast Upper BU: Words with even indices are loaded using the PS input; words with odd indices are loaded using the XS input.

Broadcast Lower BL: Words with even indices are loaded using the XS input; words with odd indices are loaded using the PS input.

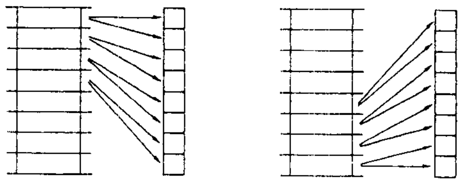


Figure 5. Broadcast Upper (BU) and Broadcast Lower (BL).

BU and BL are illustrated in Figure 5. The result of BU is that the contents of the upper half of the memory is spread out to all words. Looking at the binary representation of the index we see that BU transfers the contents of word number $<0k_{n-2}\ldots k_1 k_0 i_{n-1} i_{n-2}\ldots i_1 i_0>$ to words $<k_{n-2}\ldots k_1 k_0 i_{n-1} i_{n-2}\ldots i_1 i_0 x>$, where x takes the values 0 and 1. n successive applications of BU will spread the contents of word number $<00..00\, i_{n-1} i_{n-2}\ldots i_1 i_0>$ to words number $<i_{n-1}i_{n-2}\ldots i_1 i_0 xx..xx>$. That is, n BUs spread row 0 to all columns:

$$BU^n(A) = \begin{bmatrix} a_{00} & a_{00} & a_{00} & a_{00} \\ a_{01} & a_{01} & a_{01} & a_{01} \\ a_{02} & a_{02} & a_{02} & a_{02} \\ a_{03} & a_{03} & a_{03} & a_{03} \end{bmatrix} = A^{(0)}$$

Similarly, the result of BL is that the contents in the lower half of the memory is spread out to all words, and n applications of BL will spread row number N-1 to all columns, i.e. form $A^{(N-1)}$.

From this it should be clear that the following sequence of operations will spread any row $k = \langle k_{n-1}k_{n-2}\ldots k_1k_0\rangle$ of A to all columns, i.e. it forms $A^{(k)}$:

```
for l:=n-1 downto 0 do
    if $k_l=0$ then BU else BL;
```

Example: Row number <1011> is broadcast to all columns by the sequence BL,BU,BL,BL.

The formation of all $A^{(k)}$, k=0,1,...,N-1, from A can be described by the tree structure of Figure 6. The number of broadcasts necessary to form all $A^{(k)}$ is:

$$2+4+..+N/2+N = \sum_{k=0}^{n-1} N/2^k = 2(N-1)$$

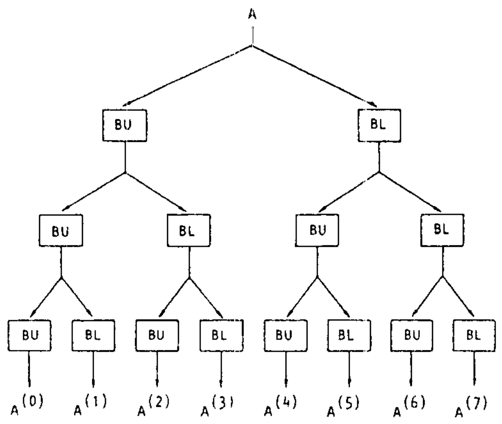


Figure 6. The pre-alignment tree.

3.4 Multiplication

The matrices $A^{(k)}$, formed by the above procedure, are now multiplied, element by element, with the matrix B. We call the resulting matrices $C_{(k)}$. Their respective elements are:

$$C_{(k)}(i,j) = A^{(k)}(i,j) * B(i,j) = a_{ki} * b_{ij}$$

3.5 Summation and Post-alignment

Element c_{kj} of the result matrix can be obtained from $C_{(k)}$ by summing all elements of column j. In other words, the columnsums of $C_{(k)}$ are the elements of row k of C. The computation of these forms the third step of the algorithm.

The interconnection network can be used to perform the summation efficiently. We introduce the operation ADDShuffle (ADDS) of a field F:

ADDS: PS(F) + XS(F)

The result of ADDS is that the contents of words with indices $\langle xi_{n-2}\ldots i_1i_0j_{n-1}j_{n-2}j_1j_0\rangle$ (x=0,1) are added and the results are put in words number $\langle i_{n-2}\ldots i_1i_0j_{n-1}j_{n-2}\ldots j_0x\rangle$. n successive applications of ADDS to a matrix stored in F yield all column-sums of the matrix, since the contents of all words $\langle xx..xxj_{n-1}j_{n-2}\ldots j_1j_0\rangle$ are added.

Thus, n successive ADDS operations on $C_{(k)}$ yield the desired result. The N sums are obtained at positions number $\langle i_{n-1}i_{n-2}\ldots i_1i_0xx..xx\rangle$, i.e. each column contains N copies of a sum. Column j of $ADDS^n(C_{(k)})$ contains N copies of element c_{kj} of the result matrix C:

$$ADDS^n(C_{(k)}) = \begin{matrix} c_{0k} & c_{1k} & c_{2k} & c_{3k} \\ c_{0k} & c_{1k} & c_{2k} & c_{3k} \\ c_{0k} & c_{1k} & c_{2k} & c_{3k} \\ c_{0k} & c_{1k} & c_{2k} & c_{3k} \end{matrix}$$

A result matrix, C, can be formed from all $ADDS^n(C_{(k)})$, k=0,1...,N-1, by taking as row k of C, row number k of $ADDS^n(C_{(k)})$:

$$C(k,j)=ADDS^n(C_{(k)})(k,j) = \sum_{i=0}^{N-1} C_{(k)}(i,j)=c_{jk}$$

It can be seen that the desired result matrix is the transpose of this matrix and can be obtained by n Perfect Shuffles. This is the post-alignment step.

The formation of C can be described as a merge of different rows of all $ADDS^n(C_{(k)})$. This process of merging parts from different results can be started earlier, since in the computation of each $ADDS^n(C_{(k)})$ there is redundancy present. Matrix $C_{(k)}$ contains the information needed to compute row number k of C. The sum of column $\langle j_{n-1}j_{n-2}\ldots j_1j_0\rangle$ needs to be preserved at word number $\langle j_{n-1}j_{n-2}\ldots j_1j_0k_{n-1}k_{n-2}\ldots k_1k_0\rangle$ only. This is achieved by, at step number l (l=n-1,...,0) of the addition, storing the result in the even word only if $k_l=0$, and in the odd word only if $k_l=1$. This gives room for two simultaneous ADDShuffles, working on different fields - i.e. on different $C_{(k)}$ - but using a common result field. We call this operation ADDMerge (ADDM(F0,F1)):

ADDMerge: in words with even indices do: ADDS(F0)
in words with odd indices do: ADDS(F1)

The process of forming C from all $C_{(k)}$ can be described by the tree shown in Figure 7. The number of ADDM-operations in this tree is N-1.

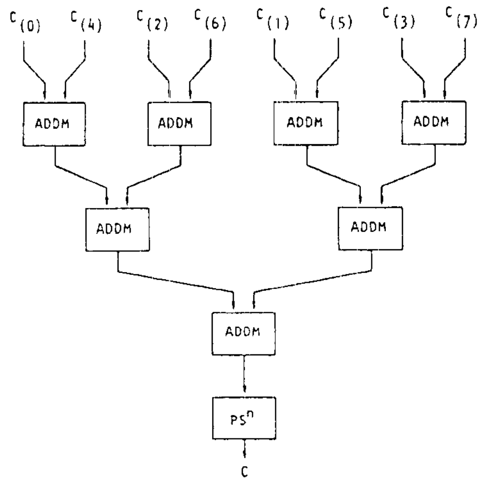


Figure 7. Summation and post-alignment

3.6 Review

The algorithm starts with matrices stored as follows:

$$A\langle k_{n-1}k_{n-2}\ldots k_1k_0i_{n-1}i_{n-2}\ldots i_1i_0\rangle = a_{ki}$$

$$B\langle i_{n-1}i_{n-2}\ldots i_1i_0j_{n-1}j_{n-2}\ldots j_1j_0\rangle = b_{ij}$$

It then produces N different matrices $A^{(k)}$ such that:

$$A^{(k)}\langle i_{n-1}i_{n-2}\ldots i_1i_0j_{n-1}j_{n-2}\ldots j_1j_0\rangle = a_{ki}$$

These are all multiplied by B to form $C_{(k)}$:

$$C_{(k)}\langle i_{n-1}i_{n-2}\ldots i_1i_0j_{n-1}j_{n-2}\ldots j_1j_0\rangle$$
$$= a_{ki} * b_{ij}$$

The different $C_{(k)}$ are then summed over i and at the same time merged to only one matrix:

$$C\langle j_{n-1}j_{n-2}\ldots j_1j_0k_{n-1}k_{n-2}\ldots k_1k_0\rangle$$
$$= \sum_{i=0}^{N-1} a_{ki} * b_{ij} = c_{kj}$$

After n perfect shuffles we finally obtain:

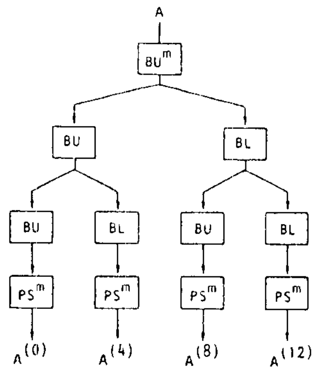
$$C\langle k_{n-1}k_{n-2}\ldots k_1k_0j_{n-1}j_{n-2}\ldots j_1j_0\rangle = c_{kj}$$

which is the desired result.

4. Matrix multiplication algorithm, $M*N^2$PEs

In this section we will show how the algorithm described above can be modified to reduce the number of multiplications by a factor of M, where $M=2^m$ and $0\le m\le n$, if we have $M*N^2$ PEs available.

The idea is that M different $A^{(k)}$ now can be stored in the same field. If the matrix B is broadcast and then properly aligned, we can produce M different $C_{(k)}$ simultaneously. The elements of C are then computed by a number of ADDMerge operations. The post-alignment, which in the N^2 case was a simple transposition, is here more complicated. However, it is possible to do the permutation with 2*(2n+m) passes through the Perfect Shuffle/Exchange network. The algorithm tree is shown in Figure 8.



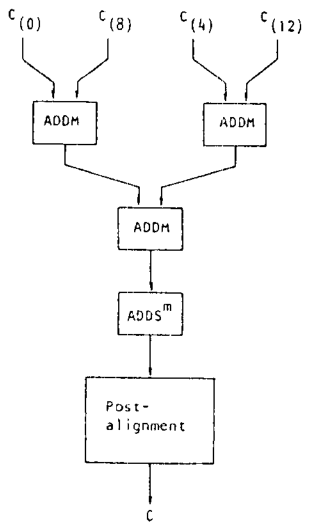


Figure 8. Pre-alignment, summation and post-alignment in the general case with $M*N^2$ PEs. Trees are drawn for m=2, i.e. M=4.

In this case the initial conditions are:

$$A\langle 0\ldots 0k_{n-1}k_{n-2}\ldots k_1k_0i_{n-1}i_{n-2}\ldots i_1i_0\rangle$$
$$= a_{ki}$$

$$B\langle 0\ldots 0i_{n-1}i_{n-2}\ldots i_1i_0j_{n-1}j_{n-2}\ldots j_1j_0\rangle$$
$$= b_{ij}$$

The number of initial zeros are m. As we want to work with full parallelity we start off by spreading A and B with m BU-operations each:

$$A<k_{n-1}k_{n-2}\cdots k_1k_0i_{n-1}i_{n-2}\cdots i_1i_0xx\ldots xx>$$
$$= a_{ki}$$

$$B<i_{n-1}i_{n-2}\cdots i_1i_0j_{n-1}j_{n-2}\cdots j_1j_0xx\ldots xx>$$
$$= b_{ij}$$

We now only need to produce N/M different matrices $A^{(k)}$ (k=0,M..N-M). This is done by n-m broadcasts and m Perfect Shuffles for each $A^{(k)}$. They satisfy:

$$A^{(k)}<i_{n-1}i_{n-2}\cdots i_1i_0j_{n-1}j_{n-2}\cdots j_1j_0$$
$$k_{n-m-1}\cdots k_0> = a_{ki}$$

They are then multiplied by $BU^m(B)$ to form $C_{(k)}$:

$$C_{(k)}<i_{n-1}i_{n-2}\cdots i_1i_0j_{n-1}j_{n-2}\cdots j_1j_0$$
$$k_{n-m-1}\cdots k_0> = a_{ki}*b_{ij}$$

Each $C_{(k)}$ contains information not only of row k of C, but also of rows k+1,k+2,...,k+M-1. After ADDMerge in n-m steps we only have one C matrix field. m additional ADDShuffles yield the elements of C, m times duplicated;

$$C<j_{n-1}j_{n-2}\cdots j_1j_0k_{n-m-1}\cdots k_0k_{n-1}\cdots$$
$$k_{n-m-0}xx\ldots xx> = c_{kj}$$

As can be seen, the final permutation of the C elements is not as straightforward as in the N^2 case. We want to reach a state where:

$$C<00\ldots 00k_{n-1}k_{n-2}\cdots k_1k_0j_{n-1}j_{n-2}\cdots j_1j_0>$$
$$= c_{kj}$$

This permutation can not be reached in $\log_2P$ passes through the Perfect Shuffle/Exchange Network, where P is the number of PEs. However, in $\log_2P$ passes it is possible to reach an intermediate state, from which the desired result can be obtained in another $\log_2P$ passes. For the sake of readability we only show it in the special case when n=4 and m=2:

$$C<j_3j_2j_1j_0k_1k_0k_3k_2xx> = c_{kj}$$

In the first 10 ($\log_2P$) passes C is permuted to:

$$C<00(k_3\oplus j_1)(k_2\oplus j_0)k_1k_0j_3j_2j_1j_0> = c_{kj}$$

where $\oplus$ is addition modulo 2.
From this it is possible to reach:

$$C<00k_3k_2k_1k_0j_3j_2j_1j_0> = c_{kj}$$

5. Timing

On a bit-serial machine like LUCAS the summation and alignment operations execute in a time that is proportional to the field size, i.e. the number of data-bits, b. Multiplication time is proportional to b^2. Lower bounds for execution times, in number of clockcycles, for the basic operations used for matrix multiplication are listed below. Measured execution times are somewhat larger, since initializing and controlling the operations take some time.

Shuffle:	2b
Broadcast:	3b
ADDShuffle:	3b
ADDMerge:	5b
Multiplication:	$3b^2$

This gives us the following execution time for matrix multiplication with N^2 PEs:

Pre-alignment:	2(N-1)*3b
Multiplication:	$N*3b^2$
Summation:	(N-1)*5b
Post-alignment:	n*2b

In the case with $M*N^2$ PEs we have:

Pre-alignment:	2*m*3b + 2(N/M-1)*3b + m*(N/M)*2b
Multiplication:	$(N/M)*3b^2$
Summation:	(N/M-1)*5b + m*3b
Post-alignment:	2*(2n+m)*2b

The presented algorithm has been programmed and tested on LUCAS for 8 by 8 matrices. As LUCAS has 128 PEs we have N=8 and M=2. LUCAS runs on a 5 MHz clock. Execution times in microseconds are listed below. Values in parentheses are obtained from the formulas above.

	b=8	b=12	b=16
Pre-alignment:	55 (51)	81 (77)	106 (102)
Multiplication:	255 (154)	489 (346)	799 (614)
Summation:	34 (29)	48 (43)	63 (58)
Post-alignment:	48 (45)	70 (67)	93 (90)
Total:	392 (278)	688 (533)	1061 (864)

For comparison the same task has been programmed in assembly language on a conventional VAX 11/780 computer. The execution time obtained for 8 by 8 matrices was approximately 3600 microseconds, regardless of the number of bits.

6. Conclusions

We have shown that matrix multiplication on a Perfect Shuffle-connected computer can be done efficiently for various matrix sizes. It is important that the most time consuming step - the multiplication - is performed with full parallelity. To accomplish this, some non-trivial pre-alignment operations are required. We have presented a simple control scheme that performs these with a minimal number of passes through the interconnection network. This makes pre-alignment time small compared to multiplication time. The summation of products is performed in a way that makes maximum use of the available parallelity. To obtain the result matrix in the correct order, a post-alignment step is required. We have shown that this can also be accomplished in a few passes over the interconnection network.

To give a sense of the amount of inevitable overhead time in an implementation, the algorithm has been programmed and tested on LUCAS. The measured execution times exceed the absolute lower bounds for this type of processor with typically 20-30%. The amount of pure data alignment compared to total computation time is between 18 and 26%, and the ratio decreases with increasing data length.

A characteristic property of LUCAS type processors is a highly repetitive structure. This makes them very well suited for VLSI implementation, which would yield a very cost effective alternative to a conventional computer. A commercial LUCAS with 1024 PEs is quite feasible. The lower bound for the matrix multiplication algorithm on such a machine, with 5 MHz clock, 16 bit data, is for 16 by 16 matrices 0.96 milliseconds and for 32 by 32 matrices 6.0 milliseconds. Corresponding execution times on a VAX 11/780 are 26 and 240 milliseconds respectively.

Acknowledgement

This research was in part supported by the National Swedish Board for Technical Development.

References

[Fern1] C. Fernström, "Programming Techniques on the LUCAS Associative Array Processor", Proceedings of the 1982 International Conference on Parallel Processing.

[Fern2] C. Fernström, "The LUCAS Processor Array and its programming environment", Lund University, Dept of Computer Engineering, Ph.D. dissertation, 1983.

[Kru1] I. Kruzela, B. Svensson, "The LUCAS Architecture and its Application to Relational Data Base Management", Proc of the 6th Workshop on Computer Architecture for Non Numeric Processing, 1981.

[Kru2] I. Kruzela, "An Associative Array, Processor Supporting a Relational Algebra", Lund University, Dept of Computer Engineering, Ph.D. dissertation, 1983.

[Ohl] L. Ohlsson, "Real time spectral analysis of speech on a small associative computer", Lund University, Dept of Computer Engineering, Technical Report, 1982.

[Park] D.S. Parker, "Notes on Shuffle/Exchange-Type Switching Networks", IEEE Trans on Computers, vol. C-29, pp. 213-222, 1980.

[Ston] H.S. Stone, "Parallel Processing with the Perfect Shuffle", IEEE Trans on Computers, vol. C-20, pp. 153-161, 1971.

[Sve] B. Svensson, "LUCAS Processor Array Design and Applications", Lund University, Dept of Computer Engineering, Ph.D. dissertation, 1983.