Matrix Multiplication on the Connection Machine

S. Lennart Johnsson, Tim Harris
and Kapil K. Mathur

YALEU/DCS/TR-736
September 1989

To appear in the Proceedings of Supercomputing 1989

Matrix Multiplication on the Connection Machine

S. Lennart Johnsson*, Tim Harris and Kapil K. Mathur
Thinking Machines Corp.
245 First Street,
Cambridge, MA 02142

Abstract

A data parallel implementation of the multiplication of matrices of arbitrary shapes and sizes is presented. A systolic algorithm based on a rectangular processor layout is used by the implementation. All processors contain submatrices of the same size for a given operand. Matrix–vector multiplication is used as a primitive for local matrix–matrix multiplication in the Connection Machine system CM–2 implementation. The peak performance of the local matrix–matrix multiplication is in excess of 20 Gflops s^{-1}. The overall algorithm including all required data motion has a peak performance of 5.8 Gflops s^{-1}.

Keywords: data parallel algorithms, systolic algorithms, matrix multiplication.

1 Introduction

The multiplication of two matrices is one of the most common operations in scientific computing. On a data parallel architecture where each processor has its own local storage the matrices to be multiplied are, in general, distributed across several, but not necessarily all processors. Several different matrix products may be formed concurrently in disjoint sets of processors. Within each set data motion is required to compute a matrix product. In general, each processor will have more than a single matrix element of each operand assigned to it. A suitable local matrix multiplication algorithm is required in addition to a global algorithm that implements the appropriate data motion given the allocation of the operands. Matrix multiplication by the standard algorithms require $pr(2q-1)$ floating-point operations for the multiplication of a $p \times q$ matrix by a $q \times r$ matrix. This is independent of the order of traversal of the index space. Different order of traversals of the index space effect the efficiency of pipelines, the memory access times and the extent to which registers, or caches, can be used to reduce the need for memory bandwidth. Scientific software libraries such as the Basic Linear Algebra Subroutines (BLAS) [17,5] encapsulate architectural dependence of commonly used matrix operations. However, currently available libraries are designed for single processor systems, with the exception of LAPACK [2,1], a library design for shared memory multi-processor systems.

A library for a data parallel architecture must be designed for concurrent execution of different tasks as well as concurrent execution of each individual task. Critical issues are the specification of which tasks can be executed independently and concurrently, and the control and data motion implicit in the algorithms for each task. With respect to performance, data motion is the most critical issue in a data parallel network architecture. Load balance is another important issue in any parallel architecture. For the systolic matrix multiplication algorithm described here only communication is of significance.

The data motion aspect in the design of distributed algorithms is most prominent in the design of systolic algorithms. A variety of such algorithms have been proposed for the multiplication of dense and banded matrices [3,4,16,19,11,14,13,7,15]. Compared to a rank–1 update algorithm, the systolic algorithms do not require broadcasting. Systolic algorithms by Dekel et. al. [4] and Cannon [3] make use of nearest neighbor communication only. The Dekel algorithm assumes a Boolean cube topology and the Cannon algorithm assumes a two–dimensional mesh topology.

*Also affiliated with Department of Computer Science, Yale University, New Haven CT 06520

Systolic algorithms assume constant storage. A reduction in the communication time is possible, if data aggregation is allowed [9]. Minimizing communications time requires additional communication buffers and temporary storage. On the Connection Machine system [8] the communications overhead is small and consequently, constant storage algorithms are often preferable.

Most systolic algorithms for dense matrices are described assuming that the matrices are square, and that there is one element per processor. Generalization to a submatrix of each operand per processor is made in [9,12,10,6]. Algorithms have also been proposed for parallelizing all three loops present in a conventional algorithm for matrix multiplication [9,12].

The matrix multiplication algorithm for dense matrices of arbitrary shape described here involves data motion on a two dimensional mesh with wrap-around. Matrix–vector multiplication is the basic local memory to memory primitive for local matrix–matrix multiplication. First, the essential characteristics of the Connection Machine architecture are described briefly. Finally, the systolic algorithm for the global multiplication of the two matrices is presented. Key implementation issues are discussed in some detail, and performance results both for the local kernels and the overall matrix multiplication algorithm are reported.

2 The Connection Machine Architecture

The Connection Machine [8] is a data parallel architecture. It has a total primary storage expandable up to 2 Gbytes. At a clock rate of 7 MHz the data transfer rate to storage is approximately 45 Gbytes s^{-1}. The primary storage has 64K ports, and a simple bit–serial processor for each port. The Connection Machine model CM–2 can be equipped with hardware for floating–point arithmetic. With this option, 32 bit–serial processors share a floating–point unit, which is an industry standard, single chip floating–point multiplier and adder with a few registers. There is a 32–bit wide data path between each floating–point chip and the memory. A total of 2048 32–bit operands can therefore be accessed in a single cycle on a fully configured Connection Machine system with 64K ports.

The Connection Machine needs a host computer. The Connection Machine is mapped into the address space of the host. The program code resides in the storage of the host. It fetches the instructions, does the complete decoding of scalar instructions, and executes them. Instructions to be applied to variables on the Connection Machine are sent to a microcontroller,

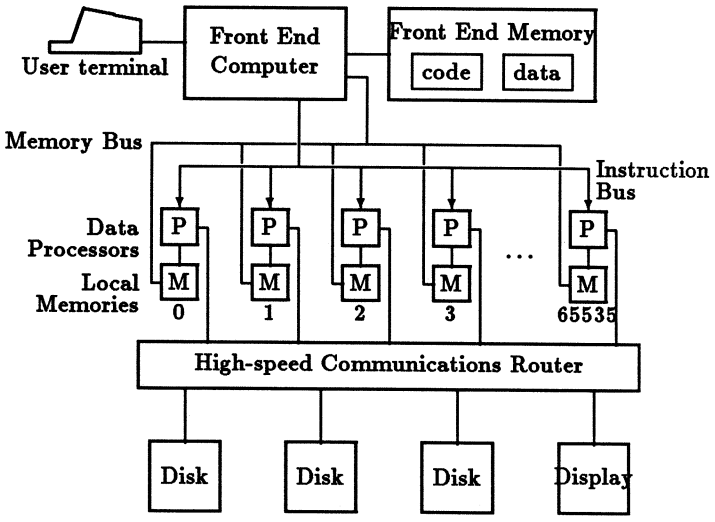

Figure 1: The Connection Machine System, CM–2 Architecture

which decodes and executes these instructions. The Connection Machine architecture is shown in Figure 1. The Connection Machine secondary storage system is known as the Data Vault. There are eight I/O channels, connecting the Data Vault to the Connection Machine. Each channel has a block transfer rate of up to approximately 30 Mbytes s^{-1}. The size of the secondary storage system is expandable up to 640 Gbytes. The Connection Machine can also be equipped with a frame buffer for fast high resolution graphics.

The Connection Machine bit–serial processors are organized such that there are sixteen such processors to a chip. These "processor" chips are interconnected as a 12–dimensional boolean cube. The communication is bit–serial and pipelined. The hardware supports concurrent communication on all ports. Through the bit–serial pipelined operation of the communication system, remote processor references require no more time than nearest neighbor references, if there is no contention for the communication channels. For communication in arbitrary patterns the Connection Machine is equipped with a router, which selects the shortest available path between the source and the destination.

The Connection Machine system supports two different address maps. In *cube address* mode the conventional translation between array indices and binary addresses are performed. In the *lattice address* mode array indices are instead encoded in a binary reflected Gray code [18]. Each axis is encoded separately. The advantage of this mode is that adjacent lattice points are placed in memory locations that differ in precisely one bit. For a memory with uniform access time there is no particular performance advantage to this encoding,

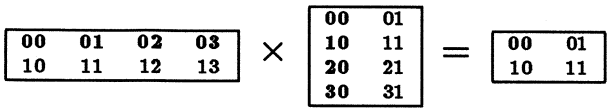

Figure 2: Local matrix multiplication operation when the sub–matrices residing on each processor are "conforming". A local matrix multiplication is performed on the entire block using the memory–to–memory primitive.

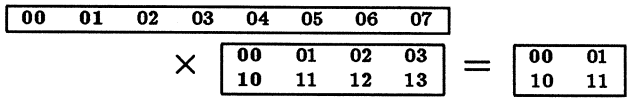

Figure 3: Local matrix multiplication operation when the sub–matrices residing on each processor are "non–conforming". In this case, the local matrix multiplication operation is performed only on the conforming part of the sub–matrices. The sub–matrices are then rotated unequally to allow use of all data.

but for a paged memory or a distributed memory, there is often a performance advantage of having adjacent lattice points assigned to memory locations that are close, or on adjacent processors. Local communication in the lattice also implies local communication between processors. The matrix multiplication algorithm described here makes use of this lattice emulation feature.

3 Data Allocation

The set of processors are assumed to be configured with N_0 processors along axis zero and N_1 processors along axis one. The processor address spans the space (k, ℓ), where $0 \leq k < N_0$ and $0 \leq l < N_1$. The matrix elements are assigned to processors by consecutive assignment [10]. Processor (k, ℓ) holds data elements $\tilde{A} = \{\alpha_0 k + \beta_0, \alpha_1 \ell + \beta_1 \mid 0 \leq \beta_0 < \alpha_0, 0 \leq \beta_1 < \alpha_1\}$ of matrix A, where $\alpha_0 = \lceil \frac{p}{N_0} \rceil$ and $\alpha_1 = \lceil \frac{q}{N_1} \rceil$. The data elements of matrix B assigned to the same processor are $\tilde{B} = \{\gamma_0 k + \delta_0, \gamma_1 \ell + \delta_1 \mid 0 \leq \delta_0 < \gamma_0, 0 \leq \delta_1 < \gamma_1\}$, where $\gamma_0 = \lceil \frac{q}{N_0} \rceil$ and $\gamma_1 = \lceil \frac{r}{N_1} \rceil$. The data elements of the product matrix C assigned to processor k, ℓ are $\{\alpha_0 k + \beta_0, \gamma_1 \ell + \delta_1 \mid 0 \leq \beta_0 < \alpha_0, 0 \leq \delta_1 < \gamma_1\}$. When $\alpha_0, \alpha_1, \gamma_0$, and γ_1 are equal to some power of two the lower order bits of the matrix element index encode the local matrix elements. For example, the lowest order $\log_2 \alpha_0$ bits of the row index and $\log_2 \alpha_1$ bits of the column index encode the local matrix elements of the matrix A.

If $N_0 \neq N_1$, then the range of the inner index for the two operands is clearly different in every processor. The alignment must assure that the inner index range for one operand is a subset of the inner index range of the other operand. This property must be maintained during the multiplication phase. We accomplish this task by emulating a square array with $\max(N_0, N_1)$ *virtual* processors assigned to each axis. Each physical processor emulates $\frac{\max(N_0, N_1)}{\min(N_0, N_1)}$ virtual processors. Figure 2 shows the local multiplication operation on the sub–matrices when the range of the inner index for the two operands is the same. In Figure 3 the range of the inner index is eight for operand $\tilde{A}$ and only two for operand $\tilde{B}$. Axis zero for operand matrix A emulates four virtual processors.

The current implementation of the Connection Machine system software requires that the axes length of any array corresponds to some power of two. For axis length p, $\lceil log_2 p \rceil$ address bits are assigned to the encoding of the elements along that axis. These address bits cannot be used for any other axis. The address field of the Connection Machine is divided into three parts (off–chip|on–chip|memory). The off–chip field consists of twelve bits that encode the 4096 Connection Machine processor chips, the on–chip field encodes the sixteen processors on each Connection Machine processor chip, and the lower order bits encode the memory addresses local to a processor. The default allocation scheme attempts to configure each part of the address space (off–chip, on–chip, and memory) to conform with the shape of an array. To the extent possible, all axes have a segment of each address field, and the ratio of the lengths of segments for different axes is approximately the same as the ratio of the length of the axes.

The number of physical processors in the Connection Machine system may be as large as 64K. When the number of physical processors exceeds the number of matrix elements the matrix is only allocated to a fraction of the total number of physical processors. For these cases, the array in which the matrix is stored must be extended beyond the size of the matrix. For example, CM–Fortran makes this extension by adding one more axis to the array. The length of this axis is equal to the the number of instances of the array that exactly fit in the physical machine.

The basic software on the Connection Machine system allocates the data serially to a processor. The bits of a word are allocated such that they have successive memory addresses. However, 32 Connection Machine processors share a floating–point unit that can access the memory of the 32 processors in bit–slices. By "transposing" the data of the 32 processors from *field–wise* to *slice–wise* each word is allocated across the memory of groups of 32 processors, each sharing a floating–point unit. Using this facility, a fully configured Connection Machine can be viewed as having 2048 32–bit wide processors, each with up to 1 Mbyte of memory. The kernels of the matrix multiplication routine are based on this slice–wise view of the ma-

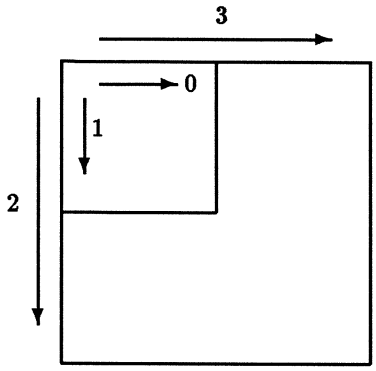

Figure 4: Loop ordering for local matrix–vector multiplication kernels.

chine.

The data transposition interchanges the lowest order off–chip bit and the processor bits (four bits) with the memory address field. A five–shuffle is performed. For a one–dimensional array the memory stride for bit k, with the lowest order bit being bit zero, is 2^s where $s = (k+5) \bmod (1 + \text{chip} + \text{memory bits})$. The stride for successive array elements is 32. The stride is increasing for elements at increasing distance up to a point. The stride for the fifth highest order bit is one. In the case of multi–dimensional arrays the stride of the different axes becomes fairly complex. Therefore, a memory reordering is performed such that the stride of the first array axis is one, the stride for the second equal to the length of the first axis, and so on.

4 Local Matrix Multiplication

In the global operation $C \leftarrow A \times B + C$, the matrix A is a $p \times q$ matrix, B a $q \times r$ matrix and C a $p \times r$ matrix. The matrices are distributed over a set of processors. Each processor in the set holds a sub–matrix $\tilde{A}$ of size $\tilde{p} \times \tilde{q}$, and a sub–matrix $\tilde{B}$ of size $\tilde{r} \times \tilde{r}$.

The local matrix multiplication makes use of matrix–vector multiplication, or vector–matrix multiplication depending on the shapes of the operand matrices $\tilde{A}$ and $\tilde{B}$. Those routines are based on SAXPY operations. Although the matrix–vector, or vector–matrix kernels are memory–to–memory operations, however in the SAXPY operation $y \leftarrow y + \alpha x$ the vector x is taken from memory and the vector y and the constant α are taken from and written to the registers of the floating–point unit. The vector length is determined by the architecture of the floating–point unit, and the way it is integrated into the Connection Machine. The architectural restrictions on the 32–bit and 64–bit floating–point units are different. The loop ordering of the matrix–vector kernel for both the 32–bit is shown in Figure 4. The loop labeled 0 is the inner–most loop. Loops 0 and 1 together define the SAXPY operation.

$\tilde{p} \times \tilde{q} \times \tilde{r}$	Gflops s^{-1}
$64 \times 4 \times 1$	7.84
$64 \times 8 \times 1$	11.98
$64 \times 16 \times 1$	16.35
$64 \times 32 \times 1$	17.35
$64 \times 64 \times 1$	19.08
$64 \times 256 \times 1$	20.38
$64 \times 4 \times 8$	9.20
$64 \times 8 \times 8$	13.53
$64 \times 16 \times 8$	17.74
$64 \times 32 \times 8$	18.02
$64 \times 64 \times 8$	19.55
$64 \times 256 \times 8$	20.51
$64 \times 4 \times 32$	9.38
$64 \times 8 \times 32$	13.71
$64 \times 16 \times 32$	17.88
$64 \times 32 \times 32$	18.11
$64 \times 64 \times 32$	19.61
$64 \times 256 \times 32$	20.52

Table 1: Performance data for the local matrix–multiplication kernels.

The time for data motion with this loop ordering is $\left\{\tilde{q}(\tilde{p}+1) + (\frac{\tilde{q}}{k_c} - 1)\tilde{p}\right\} t_\ell + \tilde{p}\frac{\tilde{q}}{k_c} t_s$, where k_c is the number of columns of C in the block matrix–vector product defined by loops labeled 0 and 1, t_ℓ is the time required for loading one data item, and t_s is the time required for storing one data item. When the loop traversal order is 0,1,3,2 the data motion time becomes $\tilde{p}\tilde{q}(1+\frac{1}{k_r})t_\ell + \tilde{p}t_s$, k_r is the number of rows in the block matrix–vector product defined by loops labeled 0 and 1. On the 32–bit floating–point unit, k_c may be several times as large as k_r. The loop ordering 0,1,2,3 minimizes the number of storage accesses.

The performance for a few matrix shapes are given in Table 1, and shown in Figure 5. The single–precision floating point rate shown in Figure 5 corresponds to a fully configured Connection Machine system with 2048 floating point units. For the example shapes of the local sub–matrices, the measured performance of the kernel is in excess of 20 Gflops s^{-1}.

5 Global Matrix Multiplication

The matrix multiplication algorithm for the Connection Machine described here is based on mesh emulation. The algorithm has two distinct phases – Alignment and Systolic multiplication. The alignment phase reallocates matrices A and B such that for a square array of processors, all processors have the same "inner" index of both operands. For a rectangular array of processors the range of inner indices of one of the

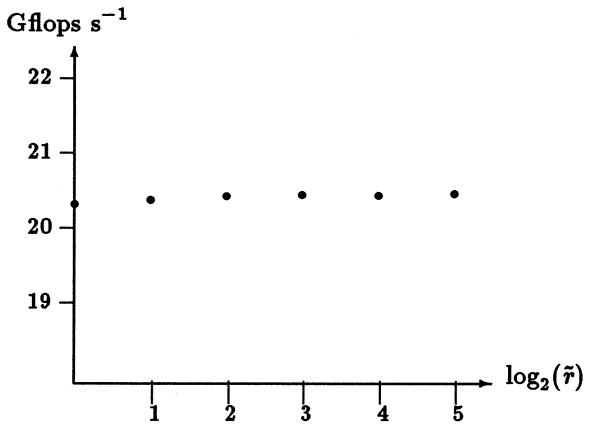

Figure 5: Performance of the local matrix–vector multiplication kernels during the multiplication of local sub–matrices $\tilde{A}$ and $\tilde{B}$ of shape 64×256 and $256 \times \tilde{r}$ respectively. The performance reported here is projected to a fully configured Connection Machine system with 2048 floating point units.

forall $i, j \in \{0, 1, \ldots, P-1\} \times \{0, 1, \ldots, P-1\}$ **do**

/* Alignment Phase*/

$a(i, j) \leftarrow a(i, (i+j) \bmod P)$
$b(i, j) \leftarrow b((i+j) \bmod P, j)$

/* Multiplication Phase */

$c(i, j) \leftarrow c(i, j) + a(i, j) \times b(i, j)$
for k:= 1 **to** $P-1$
  $a(i, j) \leftarrow a(i, (j+1) \bmod P)^{\dagger}$
  $b(i, j) \leftarrow b((i+1) \bmod P, j)^{\dagger}$
  $c(i, j) \leftarrow c(i, j) + a(i, j) \times b(i, j)$
endfor k

/* Re–alignment Phase*/

$a(i, j) \leftarrow a(i, (i-j) \bmod P)$
$b(i, j) \leftarrow b((i-j) \bmod P, j)$

endforall i, j

Table 2: Pseudo–code for the Cannon algorithm.

operands is a subset of the range of inner indices of the other operand. All processors then perform a multiplication and an addition concurrently using the local matrix–matrix kernel described in the previous section. In the case of a single matrix element per processor, and a square array of processors ($P \times P$), the algorithm can be described as in Table 2 [3]. The speed–up of the arithmetic part is proportional to the number of processors. The data motion[1] in the systolic multiplication phase of the algorithm requires nearest neighbor communication only. The multiplication of matrices of arbitrary shapes on Boolean cubes of any size involves generalizing the above algorithm for the following cases:

- The set of processors are configured with N_0 processors along axis zero, and N_1 processors along axis one, $N_0 \neq N_1$.
- A sub–matrix per processor instead of a single element.
- Arbitrary values for p, q, and r.
- Parallelization of the loop corresponding to the "inner" index.

The first generalization is necessary for several reasons. The primary reason is that the total number of processors $N = N_0 \times N_1$ to which the matrices are allocated may not be a square. In addition, for small matrices and a large number of processors, the operands for a matrix product may only extend over a subset of processors, even if only a single element of each operand is assigned to each processor. In this case, as well as in the case where each operand has several elements assigned to each processor, the optimum configuration for the processors is an array of the same shape as that of the product matrix C [9]. The need for the second and third generalization is apparent. The last generalization is motivated by the fact that the total number of processors N may be significantly greater than the number of elements $p \times r$ of the product matrix C. For $N \geq p \times r$ all three loops in a Fortran 77 code may be parallelized. The axis corresponding to the "inner index" can be instantiated partially, or totally, in space. With all three axes instantiated in space, the operands are assigned to orthogonal planes. For example, the matrix A can be assigned to the plane defined by axes zero and two, the matrix B to the plane defined by axes one and two, and the matrix C to the plane defined by axes zero and one. The matrix A is copied along axis one, and the matrix B along axis zero. The product matrix C is obtained by reduction along axis two.

The standard allocation scheme for arrays implies that, in general, the configuration of the physical machine is different for arrays of different shapes. For the algorithm described above the operands need to be allocated assuming the same physical machine. The reconfiguration of the set of physical processors to a common shape is performed as part of the alignment phase. The alignment is performed by the Connection Machine router.

[1] Corresponding to statements marked †.

$p \times q \times r$	8K	16K	32K	64K
$128 \times 1024 \times 1024$	117	196	—	—
$256 \times 1024 \times 1024$	190	321	582	968
$512 \times 1024 \times 1024$	281	496	825	1468
$256 \times 256 \times 256$	107	210	316	488
$512 \times 512 \times 512$	199	362	558	1062
$1024 \times 1024 \times 1024$	357	664	1045	1936
$2048 \times 2048 \times 2048$	—	—	1829	3463
$4096 \times 4096 \times 4096$	—	—	—	5814

Table 3: Performance in Mflops s^{-1} of the overall matrix multiplication algorithm.

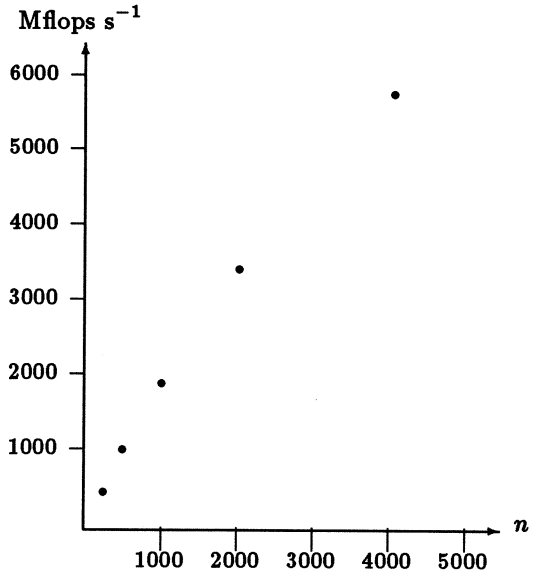

Figure 6: Performance of the matrix multiplication algorithm on a fully configured Connection Machine system CM–2 with 64K physical processors and 2048 floating–point units; n is the size of the square matrices.

The reconfiguration and alignment is made on data in field–wise format. The systolic multiplication phase is performed on data in the slice–wise format. The communication is made using the lattice emulation primitives, which can be applied directly to data stored slice–wise. The computations are performed in–place. The matrices A and B are subject to a complete revolution, and restored by an "un–alignment phase" after the systolic multiplication phase is complete. For the systolic multiplication phase, a processor in the algorithm description is equivalent to a floating–point processor.

The systolic algorithm described here requires no temporary storage. Some overall performance data for the Connection Machine implementation of the algorithm is reported in Table 3, and Figure 6. Although the systolic multiplication phase has a peak floating–point rate of approximately 8 Gflops s^{-1}, the overall algorithm peaks at 5.8 Gflops s^{-1} because the two alignment phases represent a significant portion of the total time.

6 Summary

The matrix multiplication algorithm presented in this article applies to matrices of arbitrary shapes and sizes. The processor layout is assumed to be rectangular. The size is arbitrary. The algorithm has two distinct phases: Alignment and Systolic multiplication. In the Connection Machine implementation the alignment is performed using the router. The systolic multiplication phase makes use of the mesh emulation capability. Each of the two input operands is subject to a complete revolution. One operand is subject to circular shifts along rows, the other along columns. For each cyclic shift, a multiplication of sub–matrices local to a processor is performed. Depending on the shape of the local sub–matrices, this matrix multiplication is based on either a matrix–vector multiplication, or a vector–matrix multiplication.

The measured peak performance of the local matrix multiplication kernel is in excess of 20 Gflops s^{-1}. The systolic matrix multiplication algorithm peaks at approximately 8 Gflops s^{-1}. After accounting for the initial alignment and the final un–alignment of the operand matrices, the total peak performance of the matrix multiplication implementation described here is 5.8 Gflops s^{-1}.

Acknowledgements

The timings on the fully configured Connection Machine system with 64K physical processors was made possible through the courtesy of the Advanced Computing Laboratory of the Los Alamos National Laboratories. These measurements were performed by Ralph Brickner at Los Alamos.

References

[1] Edward Anderson and Jack Dongarra. *Installing and Testing the Initial Release of LAPACK.* Technical Report MCS-TM-130, Argonne National Laboratories, May 1989.

[2] Christian Bischof, James Demmel, Jack J. Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, and Danny Sorensen. *LAPACK*

Working Note # 5, Provisional Contents. Technical Report ANL-88-38, Argonne National Laboratories, Mathematics and Computer Science Division, September 1988.

[3] L.E. Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm.* PhD thesis, Montana State Univ., 1969.

[4] Eliezer Dekel, David Nassimi, and Sartaj Sahni. Parallel matrix and graph algorithms. *SIAM J. Computing*, 10:657–673, 1981.

[5] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. *An Extended Set of Fortran Basic Linear Algebra Subprograms.* Technical Report Technical Memorandum 41, Argonne National Laboratories, Mathematics and Computer Science Division, November 1986.

[6] Geoffrey C. Fox, S.W. Otto, and A.J.G. Hey. *Matrix Algorithms on a Hypercube I: Matrix Multiplication.* Technical Report Caltech Concurrent Computation Project Memo 206, California Institute of Technology, dept. of Theoretical Physics, October 1985.

[7] Donald E. Heller. *Partitioning Big Matrices for Small Systolic Arrays*, pages 185–199. Prentice-Hall, Englewood Cliffs. NJ, 1985.

[8] W. Daniel Hillis. *The Connection Machine.* MIT Press, Cambridge, MA, 1985.

[9] Ching-Tien Ho and S. Lennart Johnsson. *Matrix Multiplication on Boolean Cubes Using Shared Memory Primitives.* Technical Report YALEU/DCS/RR-636, Department of Computer Science, Yale University, July 1988.

[10] S. Lennart Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. *J. Parallel Distributed Comput.*, 4(2):133–172, April 1987.

[11] S. Lennart Johnsson. *Computational Arrays for Band Matrix Equations.* Technical Report 4287:TR:81, Computer Science, California Institute of Technology, May 1981.

[12] S. Lennart Johnsson. Data parallel programming and basic linear algebra subroutines. In John R. Rice, editor, *Proceedings of the IMA Workshop on Mathematical Aspects of Scientific Software*, pages 183–196, Springer Verlag, 1987. YALE/DCS/RR-584, September 1987.

[13] S. Lennart Johnsson. Highly concurrent algorithms for solving linear systems of equations. In *Elliptic Problem Solving II*, Academic Press, 1983.

[14] S. Lennart Johnsson. VLSI algorithms for Doolittle's, Crout's and Cholesky's methods. In *International Conference on Circuits and Computers 1982, ICCC82*, pages 372–377, IEEE, Computer Society, September 1982.

[15] S. Lennart Johnsson and Ching-Tien Ho. *Multiplication of arbitrarily shaped matrices using the full communications bandwidth on Boolean cubes.* Technical Report YALEU/DCS/RR-721, Department of Computer Science, Yale University, July 1989.

[16] H.T. Kung and Charles E. Leiserson. *Algorithms for VLSI Processor Arrays*, pages 271–292. Addison-Wesley, 1980.

[17] C. L. Lawson, R.J. Hanson, D.R. Kincaid, and F.T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM TOMS*, 5(3):308–323, September 1979.

[18] E M. Reingold, J Nievergelt, and N Deo. *Combinatorial Algorithms.* Prentice-Hall, Englewood Cliffs. NJ, 1977.

[19] Uri Weiser and Al Davis. *Mathematical Representation for VLSI Arrays.* Technical Report UUCS-80-111, University of Utah, Dept. of Computer Science, September 1980.