Efficient Matrix Multiplication on SIMD Computers*

P. Bjørstad F. Manne T. Sørevik M. Vajteršic†

Dedicated to Gene H. Golub on the occasion of his 60th. birthday

Abstract

We describe efficient algorithms for matrix multiplication on SIMD computers. We consider SIMD implementations of Winograd's algorithm in the case where additions are faster than multiplications, as well as classical kernels and the use of Strassen's algorithm. Actual performance figures using the MasPar family of SIMD computers are presented and discussed.

*Institutt for Informatikk, University of Bergen, Thormøhlensgate 55 N-5008 Bergen, Norway. This work was supported in part, by NAVF, grant 413.89/024 and in part by NTNF, contract IT0228.20484

†On leave from the Slovak Academy of Sciences, Bratislava, supported by NTNF.

1 Introduction

One of the basic computational kernels in many linear algebra codes is the multiplication of two matrices. It has been realized that most problems in computational linear algebra can be expressed in block algorithms and that matrix multiplication is the most important kernel in such a framework. This approach is essential in order to achieve good performance on computer systems having a hierarchical memory organization. Currently, computer technology is strongly directed towards this design, due to the imbalance between the rapid advancement in processor speed relative to the much slower progress towards large, inexpensive, fast memories. This algorithmic development is highlighted in the second edition of Matrix Computations by Gene H. Golub and Charles F. Van Loan [13], where the formulation of block algorithms is an integrated part of the text. The rapid development within the field of parallel computing and the increased need for cache and multilevel memory systems are indeed well reflected in the changes from the first to the second edition of this excellent text and reference book. Their notation and analysis of the 'level-3 fraction' of a given matrix algorithm, emphasizes the importance of efficient computational kernels for BLAS-3 [11] type operations.

Today, hierarchical memories provide the same motivation as small memories and secondary storage did in the sixties and seventies. The organization of linear algebra computations in terms of a complete library of block algorithms with a corresponding set of routines operating on individual blocks, is more than twenty years old, see for example [2].

Matrix multiplication is a very compute intensive task, but also rich in computational parallelism, and hence well suited for parallel computation. The problem has a simple structure and well understood mathematical properties. It is therefore often used as a benchmark for parallel computers. Despite this, the task of writing an efficient implementation of the BLAS-3 kernels for any particular advanced architecture machine is often nontrivial [3]. We will in this paper, do a careful study of the matrix multiplication problem on SIMD computers. In order to appreciate the often subtle architectural differences between different computers, one must relate this type of work to a particular computer. We will use the MasPar MP-1 computer for our actual implementations, a brief description of this computer can be found in section 2. Most of the discussion is relevant for other data parallel machines (like the AMT DAP or the CM-2), but actual parameters will of course be different.

In particular, we will consider the relative speed of addition and multiplication as well as the relative speed of arithmetic and communication, in order to find efficient algorithms. We show that nonstandard algorithms like the one proposed by Winograd [25] and the fast method of Strassen [23] can be efficiently implemented on SIMD computers. Winograd's algorithm is attractive in the case where additions are faster than multiplications.

As was observed by Brent [7], the exchange of multiplications with additions, can give significant speedup, provided that floating point addition is executed faster than floating point multiplication. This was indeed the case in the late sixties and early seventies, but the difference decreased in the following years. Quite recently, this trend has been partially changed resulting in new computer systems where again additions are less expensive than multiplications.

In the MP-1 computer, each processor is only 4 bits wide. Arithmetic must then be implemented using 4 bit 'nibbles' and while addition is linear in the number of nibbles, multiplication is quadratic in the number of nibbles of the mantissa. Similarly, the AMT DAP is based on single bit processors while the CM-2 has special hardware for floating point arithmetic. It may be expected that also individual SIMD processors will become more complex in future generations. This will increase the floating point speed and tend to reduce the time difference between addition and multiplication. On the other hand, this difference may be present also in modern high performance microprocessors. An example is the Intel i860 chip [16], where 64 bit additions can be executed at a peak rate of 40 Mflops, while 64 bit multiplications can be performed in parallel, but at a maximum rate of 20 Mflops.

On any distributed memory computer performing matrix computations, important questions are how to map the matrices onto the computer and how to design an efficient data flow between processors. On massively parallel systems this issue is critical. The problem has attracted much interest and a number of systolic arrays have been proposed for the problem, see [17, 19] and their references. Some systolic algorithms impose a very specific design on the hardware that can be used, we focus on algorithms that can be implemented efficiently on general purpose SIMD computers.

After a brief description of the MasPar MP-1 computer, we focus, in Section 3, on the case with N^2 processors where all matrices are $N \times N$. These algorithms are the computational kernels for various block algorithms needed to handle the multiplication of arbitrary sized matrices. In Section 4, we discuss the case where each matrix dimension is of the form kN, for $k = 2, 3, \ldots$. In Section 5, we discuss briefly some of the questions related to the case where the dimensions are arbitrary.

2 Some Basic Features of the MasPar MP-1 Computer

The MasPar MP-1 system is a massively parallel SIMD computer system. The system consists of a high performance UNIX workstation (FE) and a data parallel unit (DPU). The DPU consist of at least 1024 processor elements (PEs) each with 16Kb of memory and 192 bytes of register space. All processors execute instructions broadcast by an array control unit (ACU) in lockstep, but each processor can disable itself based on logical expressions for conditional execution. It should be noted that the individual processors may operate not only on different data, but also in different memory locations, thus supporting an indirect addressing mode.

There are three different communication mechanisms available, the Xnet, the router and the global or-tree.

The PEs are interconnected by a 2 dimensional toroidal mesh which also allows for diagonal communication. In the MasPar terminology this is called the Xnet. The Xnet operates in three modes:

-Xnet:	time is: startup + #bits $*$ distance
-XnetP(ipe) :	time is: startup + #bits + distance
-XnetC(opy) :	time is: startup + #bits + distance

The two last modes are useful for regular, non-local communication, but require that the processors between any pair of communicating processors be inactive. Thus for sending over longer distance XnetP is much faster than basic Xnet. XnetC is similar to XnetP, but it leaves a copy of the transmitted variable on all the intermediate processors. The notation Xnet[k] means that the communication distance is k with respect to the processor mesh.

MasPar also supports arbitrary node to node communication through a three-stage switch called the router. For our purpose and for the current range of available models, the router communication cost is constant, independent of the size of the machine. This means that the router despite its much higher startup time, becomes more competitive compared with Xnet as the machine scales up in size, for all data movements where the communication distance scales with the size of the machine.

The global or-tree can move data from the individual processors to the ACU. If many processors send data at the same time a global reduction results. We take advantage of this mechanism to find the maximum data value of an array at a very small cost in time.

MasPar currently supports Fortran, including a substantial part of the new F90 standard [21] and C based on Kernighan and Ritchie [18], extended to take advantage of the MasPar architecture.

Floating point is implemented in software. We define the average time of a floating point instruction, $\alpha = \frac{1}{2}(Mult+Add)$. Measured in units of α, the floating point performance of the MP-1, corresponds to a peak speed of 0.0355 Mflops in 64 bit arithmetic per processor, or 290 Mflops for a machine having 8192 processors. The processors can access memory in parallel with the execution of arithmetic or communication instructions. We define the ratio

$$\delta = \frac{Load}{\alpha}, \tag{1}$$

where 'Load' is the communication time between local memory and registers to load or store one (64 bit) word. Expressing the relative speed of memory access and floating point arithmetic, we expect $\delta \leq 1$ on balanced systems. Due to the asynchronous nature of this operation on the MP-1, δ varies in the interval $(0.05 - 0.5)$ depending on the algorithm. Also define the ratio

$$\gamma = \frac{Xnet[1]}{\alpha}, \tag{2}$$

expressing the time of nearest neighbor communication relative to the time of an average floating point operation. On the MasPar MP-1, $\gamma \approx 0.2$ and a floating point multiplication takes approximately three times the time for a corresponding floating point addition, all in 64 bit precision.

A more detailed general description of the MasPar MP-1 computer can be found in [4, 9, 22].

3 Multiplying $N \times N$ matrices on an $N \times N$ Processor array

To emphasis the algorithmic structure, we first describe the basic algorithms for the special case of square $N \times N$ matrices that fit exactly on a N^2 processor machine. We assume (as is the case on current machines) that N is a power of two. Later, we discuss the modifications necessary to obtain fast algorithms for larger matrix problems.

3.1 Cannon's data flow for the standard algorithm

The standard definition of matrix multiplication, $C = AB$, as

$$c_{i,j} = \sum_k a_{i,k} b_{k,j} \quad \forall\, i, j \tag{3}$$

provides an obvious method for the computation. Evaluating each of the N^2 elements requires exactly N multiplications and $N-1$ additions, a sequential complexity of $2N^3 - N^2$. If N^3 processors are available, the N^3 multiplications may be done in one step and the N^2 sums of N terms in $\log N$ steps. On a local memory machine one must, however, take communication costs into account. On a 2-D mesh of processors with nearest neighbor communication only, Gentleman [12] has proved that there does not exist any parallel algorithm with communication complexity of order less than $O(N)$. This result is independent of the number of processors available. Thus unless we use the router communication, the largest number of processors for which we can hope to achieve optimal efficiency is $O(N^2)$.[1]

A data flow scheme for the evaluation of (3) on a 2-D mesh of processors where the matrices fit exactly on the processor grid, was designed by Cannon [8]. The algorithm is well described in [13], but since it has similarities with the alternative algorithms to be described and since it serves to introduce our notation, we briefly describe it in the next paragraph.

Only one element from each matrix is stored on each processor. In order to keep all the processors busy, we need to assure that each processor has elements from A and B that form a product term (i.e. $a_{i,k}$ and $b_{k,j}$ for some k). In Cannon's scheme this is done by an initial preskewing of the matrices. The A matrix is preskewed by rows, while the B matrix is preskewed by columns, as in the 4×4 example shown in Figure 1.

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

Figure 1, Standard preskewing.

With this preskewing a very simple data flow scheme guarantees that each processor

[1] Note that the use of pipelined communication on the mesh, like XnetP, has a cost proportional to the distance, but such a small constant (one clock cycle) that it can be used efficiently to simulate models of communication that violates this assumption for all computers in the MP-1 family.

gets appropriate pairs of elements in each step. The entire multiplication is described by the algorithm below.

In all our algorithms we use, $\leftarrow$, to denote assignment, while $\Leftarrow$ denotes data transmission. All operations are performed on matrix elements. Subscripts which occur in algorithms, do not represent the indices of the matrix elements, but a processor address. All processor addresses are modulo N. We assume the processors as well as the matrices to have indices running from $0, ..., N-1$.

Standard Matrix Multiplication
on all processors:
Preskew A; Preskew B;
on all processors:
    $c \leftarrow ab$;
for $l = 1, N-1$
    on all processors (i, j):
        $a_{i,j} \Leftarrow a_{i,j+1}$;
        $b_{i,j} \Leftarrow b_{i+1,j}$;
        $c \leftarrow c + ab$;

This algorithm performs all the multiplications needed for (3) and accumulates them in c. The difference from the standard outer product update of C, is that the index k in the term $a_{i,k}b_{k,j}$, takes different values (in fact $k = (i+j+l) \pmod N$) on different processors in each step. Consequently, the updates take place in different order on the elements of C. In this algorithm we keep all the processors busy using only nearest neighbor communication (Xnet[1]).

In the preskewing all elements of A move along rows on the processor grid while B's elements are moving along columns. This is a very regular communication and consequently well suited for the Xnet. We tried two different implementations.

Linear Preskewing
/* A and B are initialized with
element (i, j) on processor (i, j) */
for $k = 1, N-1$
    on processors (i, j) where $i \geq k$:
        $a_{i,j} \Leftarrow a_{i,j+1}$;
    on processors (i, j) where $j \geq k$:
        $b_{i,j} \Leftarrow b_{i+1,j}$;

or alternatively,

Logarithmic Preskewing
/* A and B are initialized with
element (i, j) on processor (i, j) */
for $k = 0,\ \log N\ - 1$
  on all processors (i, j) where $i(mod\ 2^k)$ is odd:
    $a_{i,j} \Leftarrow a_{i,j+2^k}$;
  on all processors (i, j) where $j(mod\ 2^k)$ is odd:
    $b_{i,j} \Leftarrow b_{i+2^k,j}$; .

The total data transmission (words times distance) resulting from these two algorithms, is in both cases $N^2(N-1)$, but their execution times on the MasPar MP-1 are different. With fewer iterations we reduce loop overhead and logical tests (with resulting changes in the active processor set), as well as the accumulated startup time for Xnet. Consequently, the logarithmic preskewing should perform better.

The router may also be used to preskew the matrices. The router views the processors as a linear array and each processor must compute the destination address for its variable. The actual communication can then be viewed as taking place in parallel. We have a total of $2N(N-1)$ 64 bit words that must be moved. The communication rates in Table 1 refers to this and does not consider the distance of communication. If (i, j) is the coordinate of a processor then $p = N * i + j$ is the router address. The individual i, j and p are all predefined and available on each processor.

The router preskew then takes the simple form:

Router Preskewing
/* A and B are initialized with
element (i, j) on processor $p = N * i + j$ */
on all processors p:
  $q \leftarrow p - i$;
on processors where $(j < i)$:
  $q \leftarrow q + N$;
$a_q \Leftarrow a_p$;
$q \leftarrow p - N * j$;
on processors where $(i < j)$:
  $q \leftarrow q + N^2$;
$b_q \Leftarrow b_p$;

While the speed of a preskew based on Xnet depends on the size of the computer, a router[2] preskew does not. Thus increasing the size of the machine makes the router more competitive relative to the Xnet.

Table 1, Mwords/s in preskewing

Machine size	1024	2048	4096	8192
Matrix size N	32	64	64	128
Linear preskew	3.0	4.0	6.0	8.1
Log preskew	5.1	7.2	10.7	15.0
Router preskew	5.0	10.2	20.5	41.1

We present preskewing data for both square and rectangular machines in Table 1. The matrix size N will always be taken equal to the larger of the two sides if the processor mesh is non-square. In this case, the matrix is mapped to the processor array by having each processor store two matrix elements. We note that the router bandwidth increases proportionally with the machine size, resulting in a constant time for the preskewing, while the two algorithms using Xnet have a bandwidth increase proportionally to the square root of the machine size. This reflects that the average communication distance grows as the square root of the number of processors. Also note how much faster the logarithmic preskew is compared with the linear, in fact for the 1024 processor machine this is the method of choice.

3.2 Data flow for Winograd's algorithm

Winograd [25] proposed the following method for matrix multiplication:
Let

$$d_{i,j} = \sum_{k=0}^{N/2-1} (a_{i,2k} + b_{2k+1,j})(a_{i,2k+1} + b_{2k,j}) \tag{4}$$

and

$$a_i^* = \sum_{k=0}^{N/2-1} a_{i,2k} a_{i,2k+1} \tag{5}$$

[2] Clearly this is only true for the current range of machines. In general one would expect the time to grow logarithmically with the number of processors since the number of stages in such a switch will increase with the number of processors.

$$b_j^* = \sum_{k=0}^{N/2-1} b_{2k+1,j} b_{2k,j} , \tag{6}$$

then the elements of C can be computed as

$$c_{i,j} = d_{i,j} - a_i^* - b_j^* . \tag{7}$$

The exact flop count for this algorithm is $2N^3 + 3N^2 - 2N$, which is slightly more than the standard product (3). However, the number of multiplications is one half at the expense of additions. Consequently on the MP-1, there is a potential maximum speed up of 25% using Winograd's algorithm, if we are able to construct an efficient data flow scheme for the algorithm.

The numerical stability of this algorithm was analyzed by Brent [7]. He shows that scaling of the matrices A and B is essential. Define the norm $\|A\| = \max_{i,j} \mid a_{i,j} \mid$. If the crude, but easy to implement scaling

$$A \leftarrow 2^p A, \quad B \leftarrow 2^{-p} B, \tag{8}$$

where p is an integer such that

$$\frac{1}{2} \leq 2^{2p} \frac{\|A\|}{\|B\|} < 2 , \tag{9}$$

then (7) will compute $AB + E$ with $\|E\|$ bounded by

$$\|E\| \leq \frac{9}{8}(n^2 + 16n)u\|A\| \, \|B\|, \tag{10}$$

with u being the unit roundoff of the machine. This compares well with the corresponding bound for the standard algorithm

$$\|E\| \leq n^2 u \|A\| \, \|B\| + O(u^2) , \tag{11}$$

although a generally stronger, componentwise bound exists for this algorithm [13]. In Table 2, we compare two different scaling algorithms. Both algorithms first find $\|A\|$ and $\|B\|$ in the two matrices. The scaling is then performed as outlined above. We scale by a power of two, implemented either as a shift of the exponent or by a straightforward multiplication. We report the performance in millions of 64 bit words scaled per second. This scaling takes advantage of the global or-tree for finding the maximum elements.

We note that exponent shifting is much faster and also avoids extra rounding errors. The drawback is a more machine dependent implementation.

Table 2, Mwords scaled/s in Winograd's algorithm

Machine size	1024	2048	4096	8192
Matrix size	32	64	64	128
Multiplication	3	9	12	36
Exponent shift	9	22	35	86

The correction terms (5) and (6) are easily computed by a standard log-sum in parallel for all rows and columns. The choice of communication for this operation is XnetP. When found, the correction terms are broadcasted along rows or columns using XnetC. The parallel arithmetic complexity of (5) and (6) is $\log N$. In computing the log-sum there will be $\log N$ startups for the XnetP and a total transmission cost proportional to N. (Remember that on the MP-1, the transmission cost will be dominated by the $\log N$ term for all existing values of N.)

Keeping one element from each matrix on each processor, we need $(a_{i,2k}, b_{2k+1,j})$ on processor (i,j). Next, we need $(a_{i,2k+1}, b_{2k,j})$ followed by $(a_{i,2k+2}, b_{2k+3,j})$ and $(a_{i,2k+3}, b_{2k+2,j})$. This is the same regular data flow as in Cannon's algorithm, except that the elements of B are pairwise interchanged. The corresponding preskewing is shown in Figure 2.

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

Figure 2, Preskewing for Winograd I

With this initialization, we are able to obtain all the sums $a_{i,2k+1} + b_{2k,j}$ and $a_{i,2k} + b_{2k,j+1}$ on all processors. The difficulty is, however, that the two sums in (4) do not turn up at the same time on every processor. The processors are divided into two groups in a checkerboard pattern. On the 'black' processors the second sum turns up one step later than on the 'white'. In order to do the multiplication and the update of $d_{i,j}$ simultaneously on all processors, we need to store the first sum in a register on the 'white' processors until the second sum has been computed. This results in an extra interchange of the values of two variables. The algorithm can be described as follows:

Winograd I (Single Correction)

```
on all processors:
    Scale A; Scale B;
/* compute the correction terms */
on all processors:
    a*_i ← logsum(a_{i,1} a_{i,0} + a_{i,3} a_{i,2} + ··· + a_{i,N-1} a_{i,N-2});
    b*_j ← logsum(b_{0,j} b_{1,j} + b_{2,j} b_{3,j} + ··· + b_{N-2,j} b_{N-1,j});
    c ← -a*_i - b*_j;
/* preskew according to figure 2 */
on all processors:
    Preskew A; Preskew B;
/*multiplication phase*/
on all processors:
    s0 ← a + b;
for l = 1, N/2
    on all processors:
        a_{i,j} ⇐ a_{i,j+1};
        b_{i,j} ⇐ b_{i+1,j};
        s1 ← a + b;
        a_{i,j} ⇐ a_{i,j+1};
        b_{i,j} ⇐ b_{i+1,j};
        s2 ← a + b;
    on processors (i, j) where (i + j) is even:
        tmp ← s0;  s0 ← s2;  s2 ← tmp;
    on all processors:
        c ← c + s1 * s2;
```

By unrolling the loop one level, the three assignments needed for swapping, can be replaced by one.

Another strategy which allows us to group together pairs of elements of A and B where the index k differ by 1, is based on making copies of A and B which are shifted one column or row, respectively. If we keep the same simple data flow, but cyclically send one copy into the other, always sending the one which is a step ahead, we manage to move both copies two positions in only two steps, instead of four. However, we are now computing

$$d_{i,j} = \sum_{k=1}^{N/2} (a_{i,2k} + b_{2k-1,j})(a_{i,2k-1} + b_{2k,j}) \tag{12}$$

on half the processors. Note that the indices in (12) must be taken modulo N. Being different from (4), we need different correction terms on these processors. Using the two versions simultaneously, we now compute two sets of correction terms. For each processor we use the term that corresponds to the $d_{i,j}$ that is computed. The algorithm can be stated:

Winograd II, Double Correction

/* Scale the matrices as in (8) and (9) */
on all processors:
  Scale A; Scale B;
/* compute the correction terms */
on all processors:
  $a_i^* \leftarrow logsum(a_{i,1}a_{i,0} + a_{i,3}a_{i,2} + \cdots + a_{i,N-1}a_{i,N-2});$
  $aa_i^* \leftarrow logsum(a_{i,0}a_{i,N-1} + a_{i,2}a_{i,1} + \cdots + a_{i,N-2}a_{i,N-3});$
  $b_j^* \leftarrow logsum(b_{0,j}b_{1,j} + b_{2,j}b_{3,j} + \cdots + b_{N-2,j}b_{N-1,j});$
  $bb_j^* \leftarrow logsum(b_{N-1,j}b_{0,j} + b_{1,j}b_{2,j} + \cdots + b_{N-3,j}b_{N-2,j});$
on all processors where $(i + j)$ even:
  $c \leftarrow -a_i^* - b_j^*;$
on all processors where $(i + j)$ odd:
  $c \leftarrow -aa_i^* - bb_j^*;$
/* preskew as for Cannon's algorithm */
on all processors:
  Preskew A; Preskew B;
/* multiplication phase */
on all processors:
  $\tilde{a}_{i,j} \Leftarrow a_{i,j+1};$
  $\tilde{b}_{i,j} \Leftarrow b_{i+1,j};$
  $c \leftarrow c + (a + \tilde{b}) * (\tilde{a} + b);$
for $k = 1, N/2 - 1$
  on all processors:
    $a_{i,j} \Leftarrow \tilde{a}_{i,j+1};$
    $b_{i,j} \Leftarrow \tilde{b}_{i+1,j};$
    $\tilde{a}_{i,j} \Leftarrow a_{i,j+1};$
    $\tilde{b}_{i,j} \Leftarrow b_{i+1,j};$
    $c \leftarrow c + (a + \tilde{b}) * (\tilde{a} + b);$

While computing the double set of log-sums we always have enough processors to do the arithmetic in parallel for the sums. However, when using XnetP for the communication all intermediate processors must be idle. The communication time for computing the correction terms is doubled, while the arithmetic has the same time complexity as in the single correction case.

Table 3, SIMD matrix multiplication kernels.

Machine size	1024		2048		4096		8192	
Matrix size	32		64		64		128	
	Mflops	OH	Mflops	OH	Mflops	OH	Mflops	OH
Cannon	24	16%	54	8%	103	8%	226	4%
Winograd I	21	38%	47	24%	102	25%	220	13%
Winograd II	20	43%	57	27%	100	29%	258	16%

3.3 Timing results

We have carefully timed the different routines. The results are presented in Tables (3) and (4). The Mflops[3] are based on flop counts for the standard method (3). We compare the three algorithms in Table 3, where all calculations are performed in 64 bit precision. We present Mflops figures and the percent of the total time spent in 'OverHead'. The column labeled 'OH' covers preskewing and in the case of Winograd, scaling and computation of the correction terms.

The results require a few comments. When we compare the two variants of Winograd as stated in this paper, it seems that Winograd I should be slightly superior in terms of complexity. This advantage can be seen on the square machines (1024,4096). On the non-square machines, we need to store two matrix elements on each processor. This leads to a reduction from 4 to 3 in the nearest neighbor communication in the inner loop, but doubles the register requirements. In Winograd I there is the additional need to unroll the inner loop one level. The resulting code requires more registers than currently available. Winograd II is therefore considerably faster on the non-square machines. As predicted by the analysis, the overhead of all three algorithms is reduced as N increases. Cannon's algorithm is competitive on the smaller machines due to its lower overhead, but on the 8192 processor machine (and on larger machines) we notice how the 25 percent saving in arithmetic puts Winograd ahead in performance.

Since the relative speed of multiplication and addition depends on the length of the mantissa, we give results for both 32 bit and 64 bit precision floating point formats in Table 4. These formats have 23 and 52 bits in the mantissa respectively. We observe

[3] Since Winograd's algorithm needs some additional operations for doing the correction terms ($O(n^2)$), the correct flop counts are actually somewhat higher here. But a fair comparison from a practical point of view requires the same flop counts for both algorithms. All Mflops figures in this paper refer to the standard method (3).

Table 4, Dependence on floating point format.

Precision	64 bit		32 bit	
Algorithm	Mflops	OH	Mflops	OH
Cannon	226	4%	461	7%
Winograd I	220	13%	384	16%
Winograd II	258	16%	452	21%

that Cannon more than doubles in performance, while the speedup is about 75 percent for Winograd, consistent with the relative importance of multiplications in the two algorithms.

4 Block algorithms

In this section we discuss how to multiply matrices having more elements than the number of processors available. Again, assuming N^2 processors, we first deal with square matrices of size $n = kN$, where $k = 2, 3, \ldots$.

There are two common ways to partition the matrix. One can either divide it into k^2 blocks, each of size $N \times N$ and distribute one element of each submatrix to the corresponding processor. Alternatively, one can split the matrix into N^2 $k \times k$ blocks and distribute each block to an individual processor.

In the first case, one can simply do the matrix multiplication by a block version of the standard algorithm. This requires k^3 calls to a routine for doing the matrix multiplication of $N \times N$ matrices. The preskewing can be done once for each block, a total of k^2 calls to the preskewing routine. Similarly, for the Winograd kernel, both the scaling and the correction terms can be computed directly on the global matrix. This improves the parallel complexity to $O(k^2+k \log N)$ for the correction terms and to $O(k^2)$ for the scaling. Thus asymptotically the arithmetic of the kernel loop will dominate the entire computation. This approach gives the same ratio between communication and arithmetic as for the $N \times N$ case considered in Section 3. In Table 5, we present data for this scheme with $N = 128$ and 8192 processors.

In the second case, it is straightforward to do a block version of Cannon's algorithm. In this case we get a block preskewing. In N steps each processor will do a matrix multiplication of $k \times k$ blocks and send the two blocks to its neighbors. Now we have $O(k^3 N)$ arithmetic operations, but only $O(k^2 N)$ communication. This advantage may, however, be offset by the more frequent access to memory, of order $O(k^3N)$. Using the relations (1) and (2) we obtain the inequality

$$(\gamma - \delta)k \geq \gamma + \hat{\delta}, \tag{13}$$

that must hold if this algorithm shall be faster than the first one considered. Here $\hat{\delta}$ corresponds to the memory access speed for the first blocking strategy. The relation shows that local memory access must be faster than nearest neighbor communication, for the second blocking strategy to give a faster method. This is only true on the MP-1 if overlap between register loads and arithmetic can be achieved. The global $N \times N$ memory access (fetching one number to each processor) cannot easily be overlapped and $\hat{\delta} \approx .5$, while the reading of local $k \times k$ blocks facilitates a δ of approximately .08 in our case. We conclude that for sufficiently large matrices, the second strategy will be most efficient. We employ Cannon's data flow, but have a choice between standard matrix multiplication at the block level or the use of Winograd's method.[4] We note that preskewing, scaling and correction terms can be performed in the same way as above.

Finally, note that Winograd's algorithm cannot be applied to matrix blocks since (4)-(7) depends on commutativity. In addition to the two alternatives already mentioned, there is obviously a 'virtual processor' Winograd method, where the data for each virtual processor is grouped locally and assigned to physical processors. The complexity of this method is similar to the first block method considered in this section, but the programming is more complex. In addition, this approach suffers from a more expensive and complicated preskewing.

4.1 Strassen's algorithm on an SIMD machine

Strassen first presented his algorithm for matrix multiplication in [23]. It is based on a recursive divide and conquer scheme. The algorithm is clearly presented in Chapter 1 of Matrix Computations [13]. It is well known that the algorithm has a sequential complexity of $O(n^{2.807})$ as compared to $O(n^3)$ for ordinary matrix multiplication. Because of lower order terms it is advisable to employ an ordinary matrix multiplication routine when the dimension of the blocks reaches some preset cutoff point $n_0 \geq 8$ [14]. Due to algorithm overhead that grows with the number of recursions, as well as efficient use of the hardware at hand, we chose to take $n_0 = 128$, for our 8192 processor machine. We

[4]In this case k should be even, or the code must simulate the algorithm for $k + 1$.

Table 5, Performance of block algorithms with $N \times N$ kernel blocks based on kernels from Table 3

	64 bit precision				32 bit precision			
	Block Cannon		Block Winograd		Block Cannon		Block Winograd	
n	Time	Mflops	Time	Mflops	Time	Mflops	Time	Mflops
256	0.15	230	0.12	272	0.07	484	0.07	490
512	1.15	233	0.92	291	0.54	495	0.50	531
1024	9.14	235	7.14	301	4.30	500	3.89	551
2048	72.82	236	56.50	304	34.12	503	30.47	564

can then use one of the computational kernels described in the previous section with $N = 128$.

Lately, there has been a renewed interest in Strassen's algorithm. Bailey [1] implemented it on a CRAY-2 and reports of speedups up to 2.01 for $n = 2048$. The numerical properties of this algorithm are analyzed in [6] and more recently in [14], see also Golub and Van Loan [13] for a discussion of problems where Strassen's algorithm should not be used. The algorithm satisfies the following error bound:

$$\|E\| \leq ((\frac{n}{n_0})^{\log_2 12}(n_0^2 + 5n_0) - 5n)u\|A\|\,\|B\| + O(u^2)\ , \tag{14}$$

where $n_0 \leq n$ is the cutoff point mentioned above ($\log_2 12 \approx 3.6$). This should be compared with standard multiplication (11) and with Winograd's algorithm (10). The error bound is somewhat weaker, but may still be regarded as acceptable unless small, componentwise relative errors are required. Empirical results from both Bailey [1] and Higham [15] show that the error in Strassen is small enough to justify its use in applications where speed is crucial. Also note that our choice of $n_0 = 128$ improves the bound for realistic values of n, compared to having a very small value. Both IBM and CRAY support routines for fast matrix multiplications using Strassen's algorithm.

In this section we restrict k to be a power of 2 (ie. $k = 2^l\ \ l = 1, 2, ..$) and we partition the matrix into k^2 blocks of size $N \times N$. With this layout of the matrix all additions and subtractions can be performed in parallel without any communication between the PEs. At each step of the algorithm, each processor views its data as being a local $n \times n$ matrix on which it is performing Strassen's algorithm. Once the cutoff point is reached, each processor will have one element that fits into one of the standard kernel matrix multiplication algorithms described earlier. Both Cannon and Winograd algorithms were tried as the kernel to perform the matrix multiplications. Note that in both cases we can perform the preskewing of the k^2 blocks in a preprocessing step. Also, the scaling step in Winograd's algorithm can be performed as part of the preprocessing. This reduces the 'Overhead' in Table 3, significantly. The use of Winograd as a computational kernel in Strassen's algorithm, also slightly changes the error bound (14) to

$$\|E\| \leq ((\frac{n}{n_0})^{\log_2 12}(\frac{9}{8}n_0^2 + 23n_0) - 5n)u\|A\|\,\|B\| + O(u^2)\,, \tag{15}$$

Strassen's algorithm will have $l = log(k)$ levels of recursion and require approximately $k^{2.8}$ (kernel) matrix multiplications each of size $N \times N$. Note that each processor therefore will do $2k^{2.8}N$ nearest neighbor communications compared with only $2k^2N$ for the block methods. Asymptotically, Strassen will always win due to the lower exponent in arithmetic complexity, but for practical problems we obtain the inequality

$$(1 + \gamma + \hat{\delta}/N)k^{\log_2 7} \leq (1 + \delta)k^3 + (\gamma + \delta + \hat{\delta})k^2 \tag{16}$$

for values of k where Strassen's method will outperform the asymptotically best block algorithm. Here $\hat{\delta}$ again refers to memory access that cannot easily be overlapped with arithmetic. The last term on the right hand side comes from the memory access when sending the blocks to neighbor processors. On the MP-1, this inequality is always satisfied. If we neglect the $\hat{\delta}/N$ term and the k^2 terms then (16) simplifies to

$$k \geq \left(\frac{1+\gamma}{1+\delta}\right)^5 \tag{17}$$

The value of k is therefore quite sensitive to an increase in γ. For example, if we assume that $\delta \ll 1$ and take the quite reasonable value of $\gamma = 1$, then $k \geq 32$, corresponding to 5 levels of recursion in Strassen. This implies that the matrix must be at least of dimension 4096 requiring more than 400 Mbyte of memory, perhaps exceeding the size of the machine.

The algorithm was tried on matrices of size kN, $N = 128$, $k = 2^l$, $l = 1, 2, 3, 4$. Tables 6 and 7 gives the timings of the different cases. Comparing Table 5 with the left parts of Tables 6 and 7 we find in agreement with the discussion, that the partitioning into $N \times N$ blocks is best for smaller matrices. The crossover point is around $n = 2048$, slightly higher than predicted. In 32 bit precision δ increases and the same effect is even more pronounced. As predicted by (16) Strassen's algorithm is faster than the block methods for any levels of recursion on the MP-1. We note that our block Winograd

Table 6, Performance of block algorithms in 64 bit precision.

	Block Cannon		Block Winograd		Strassen Cannon		Strassen Winograd	
n	Time	Mflops	Time	Mflops	Time	Mflops	Time	Mflops
256	0.21	163	0.20	170	0.13	256	0.11	294
512	1.35	199	1.17	230	0.91	295	0.79	341
1024	9.62	223	7.82	275	6.34	339	5.47	392
2048	72.44	237	56.87	302	44.42	387	38.14	450

Table 7, Performance of block algorithms in 32 bit precision.

	Block Cannon		Block Winograd		Strassen-Cannon		Strassen-Winograd	
n	Time	Mflops	Time	Mflops	Time	Mflops	Time	Mflops
256	0.11	315	0.11	302	0.06	538	0.06	530
512	0.68	394	0.66	409	0.43	624	0.43	619
1024	4.80	447	4.40	488	3.00	716	3.01	714
2048	36.00	477	31.85	539	21.02	817	21.01	817

code is faster than the one processor CRAY-2 figures using the CRAY MXM library, reported by Bailey [1]. Our results for Strassen's method are also quite comparable with his.

Another similar algorithm, due to Winograd [5], which uses only 15 additions and subtractions (as compared to 18 by Strassen) was also implemented. There was no significant improvement in execution time, since the block multiplication completely dominates the small saving in arithmetic.

We note that Manber [20] claims that Strassen's algorithm cannot be easily parallelized. Our results show a practical, parallel version, but depends on a very favorable, low value of the parameter γ.

5 Matrices of arbitrary size

Suppose we have two $n \times n$ matrices and N^2 processors. If n/N is an integer we can use any of the algorithms defined in section 3 or 4. If n/N is not an integer we may divide the matrices into $k \times k$ blocks, $k = \lceil n/N \rceil$ and place the K^2 blocks, $K = \lceil n/k \rceil$, in the upper left $K \times K$ part of processor array. With this mapping of the data there is at least two simple modifications of the standard algorithms from Section 4.

We may extend the matrix with zero blocks and run the algorithm as before. Considering only the multiplication part, the parallel complexity of this algorithm will be

$$2k^2 N\alpha(k+\gamma) \tag{18}$$

where α and γ are defined in (2). Alternatively, we only use the $K \times K$ processors. In this case the boundaries must be handled using two extra XnetP$[N-K]$ in the inner loop. For comparison we get:

$$2k^2 K\alpha(k+\gamma+\hat{\gamma}) \tag{19}$$

where $\hat{\gamma}$ is defined like γ, but using the time of XnetP$[N-K]$ instead of Xnet[1]. For the MP-1 one can assume that $\hat{\gamma} < 2\gamma$ for interesting values of K. This shows that the last approach should be used if

$$K < \frac{k+\gamma}{k+2\gamma} N \; . \tag{20}$$

With $\gamma \approx 1/5$, this will almost always be the best choice.

Consider now the case where the matrix blocks in our partition are non-square. This is necessary when the matrices (or the processor array) are non-square. In Cannon's scheme the elements of two matrices move in every step. Cannon chose A and B to move, while the elements of C remain in place. We may as well move B and C or A and C. For the previously described preskewing these alternative data flows force one of the matrices to move along diagonals. While the arithmetic work is independent of the data flow, the communication time is not. Assuming that the matrices are of different shape and partitioned as above, we will minimize the communication effort by always sending the two matrices with the smallest block sizes. This possibility is available when the interconnection network supports diagonal communication like on the MP-1.

Finally, let us consider the case where the number of matrix elements is less then the number of processors. Alternative algorithms based on making copies of A and B to all processors exist. If this is done properly, up to n^3 processors can participate in the multiplication phase. Finally, the summation of all n^2 innerproducts must take at least $\log n$ steps. However, as proved by Gentleman [12], the communication complexity is still $O(n)$ for a 2-D mesh with nearest neighbor communication only. Typically, we want a binary tree network to support this kind of algorithm [10]. On the MP-1, one can do a rather efficient simulation of a binary trees using XnetP. In particular, one can design efficient algorithms for matrices of dimension $n = 2^l$, $n < N$. M. Vajteršic has described such algorithms for the MP-1 in [24].

6 Conclusions

We have developed and analyzed data flow algorithms for Winograd's and Strassen's matrix multiplication algorithms and shown that they can be efficiently implemented on a state of art massively parallel SIMD computer. The algorithms perform close to the theoretical maximum of the machine and provide a very cost effective way of doing large scale matrix computations. Our algorithms can also be implemented on alternative SIMD machines like the AMT DAP and the CM-2. In order to predict the performance on these machines the parameters α, δ and γ must be determined and major architectural differences (e.g. router performance, XnetP type communication) must be taken into account. We note in particular, that Strassen's algorithm depends on a very favorable communication speed γ. There will be a considerable challenge to maintain this property in future data parallel computing systems.

Acknowledgement

We thank Dr. Robert Schreiber for suggesting the data flow scheme employed in the second Winograd algorithm and for stimulating discussions. Also, thanks to Ken Jacobsen at MasPar for providing us with technical data and Erik Boman for coding the fast scaling algorithm used to stabilize the Winograd algorithm.

References

[1] D. H. Bailey, *Extra high speed matrix multiplication on the cray-2*, SIAM Journal of Science and Statistical Computing, 9 (1988), pp. 603–607.

[2] K. Bell, B. Hatlestad, O. E. Hansteen, and P. O. Araldsen, *NORSAM, a programming system for the finite element method. Users manual, Part 1, General description.*, NTH, Trondheim, 1973.

[3] C. H. Bishof, *Fundamental linear algebra computations on high-performance computers*, tech. report, Argonne National Laboratory, August 1990.

[4] T. Blank, *The MasPar MP-1 architecture*, in Proceedings of IEEE Compcon Spring 1990, IEEE, February 1990.

[5] G. Brassard and P. Bratley, *Algorithms: Theory and Practice*, Prentice Hall, 1988.

[6] R. P. Brent, *Algorithms for matrix multiplication*, Tech. Report Technical Report CS 157, Computer Science Department, Stanford University, 1970.

[7] R. P. Brent, *Error analysis of algorithms for matrix multiplication and triangular decomposition using Winograd's identity*, Numerische Mathematik, 16 (1970), pp. 145–156.

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

[9] P. Christy, *Software to support massively parallel computing on the MasPar MP-1*, in Proceedings of IEEE Compcon Spring 1990, IEEE, February 1990.

[10] E. Dekel, D. Nassimi, and S. Sahni, *Parallel matrix and graphs algorithms*, SIAM Journal on Computing, 10 (1981), pp. 657–675.

[11] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling, *A set of level 3 basic linear algebra subprograms*, ACM Trans. Math. Soft., 16 (1990).

[12] W. M. Gentleman, *Some complexity results for matrix computations on parallel processors*, Journal of the Association for Computing Machinery, 25 (1978), pp. 112–115.

[13] G. H. Golub and C. F.Van Loan, *Matrix Computations*, Johns Hopkins Univ. Press, 1989. Second Edition.

[14] N. J. Higham, *Exploiting fast matrix multiplication within the Level 3 BLAS*, ACM Trans. Math. Software, 16 (1990), pp. 352–368.

[15] ——, *Stability of a method for multiplying complex matrices with three real matrix multiplications*, Tech. Report Num.Anal.Report no.181, Dept. of Mathematics, University of Manchester, January 1990.

[16] Intel Computer Corporation, *i860 64-bit Microprocessor Programmer's Reference Manual*, 1990.

[17] H. J. Jagadish and T. Kailath, *A family of new efficient arrays for matrix multiplication*, IEEE Transactions on Computers, 38 (1989), pp. 149–155.

[18] B. W. Kernighan and D. M. Ritchie, *The C programming language*, Prentice Hall, 1978.

[19] H. T. Kung, *Why systolic architecture?*, Computer, 15 (1982), pp. 37–46.

[20] U. Manber, *Introduction to Algorithms*, Adison-Wesley, 1989.

[21] M. Metcalf and J. Reid, *Fortran 90 Explained*, Oxford Science Publications, 1990.

[22] J. Nickolls, *The design of the MasPar MP-1, a cost effective massively parallel computer*, in Proceedings of IEEE Compcon Spring 1990, IEEE, February 1990.

[23] V. Strassen, *Gaussian elimination is not optimal*, Numerische Mathematik, 13 (1969), pp. 354–356.

[24] M. Vajteršic, *Matrix multiplication algorithms for matrices of the size $n \leq 128$ on the MasPar parallel computer*, tech. report, Institutt for Informatikk, University of Bergen, Thormøhlens gate 55, N-5008 Bergen, NORWAY, August 1990.

[25] S. Winograd, *A new algorithm for inner product*, IEEE Transactions on Computing, C-18 (1968), pp. 693–694.