A PARALLEL ARCHITECTURE AND ALGORITHMS FOR OPTICAL COMPUTING

Ahmed LOURI
Department of Electrical and Computer Engineering, University of Arizona, ECE Building Room 320J, Tucson, AZ 85721, USA

Received 28 November 1988

A new parallel optical architecture is introduced for computing massively data-parallel applications. The system processes two-dimensional binary images as basic computational entities. The processing is based on the optical symbolic substitution (SS) technique. New optical SS rules are introduced as well as a technique for designing and mapping data-parallel algorithms onto the proposed architecture. Implementation issues and performance analysis are also considered.

1. Introduction

Along with the tremendous progress in science and technology, processing of large amounts of data in real-time has been increasingly required in a wide variety of applications. Examples of these applications include signal and image processing. A common factor of these applications is the high degree of *data-parallelism* in which simple arithmetic and logic operations are simultaneously applied across all the data points. Current electronic computing systems are not capable of dealing with the computational requirements of massively data-parallel applications due primarily to the limited processor-memory bandwidth (the von Neumann bottleneck) and the lack of adequate interconnects for inter-processor communications.

The driving features of optical systems are the massive fine-grain parallelism and the high degree of communication flexibility. Large images of bright and dark spots can be moved around with a great ease. These attributes are well suited for applications that require processing large amounts of structured data such as multi-dimensional arrays and that favor SIMD (single instruction multiple data) mode of computations. In recent years, several SIMD optical architectures of varying degrees of flexibility and design complexity have been proposed [1–5]. Explored here, is a parallel architecture for massively parallel computing that is amenable to optical implementation and a technique for mapping parallel algorithms onto it.

2. The bit-plane architecture

Fig. 1 depicts a block diagram of the basic components of the *bit-plane* architecture. The architecture manipulates bit planes (or binary images) as basic computational entities. Each bit plane i corresponds to a weight factor 2^i in the binary representation as shown in fig. 2. Up to three bit planes can be processed simultaneously. For images of $n \times n$ elements, it follows that up to $3n^2$ operations are performed concurrently. The heart of the architecture is the processing unit. Locally, this unit can be viewed as a bit-serial or a bit-slice processor, since it performs one logical operation, on one, two or three single-bit operands. Globally, it can be viewed as a plane-parallel processor, since it performs the same operation on a large set of operands encoded as bit planes in parallel. This bit-serial processing allows flexible data formats and almost unlimited precision.

2.1. The processing unit

The processing unit operates in the SIMD mode of computation, where the same operation is applied to all the data entries. In the proposed system, processing is based on the optical symbolic substitution logic [1]. Information is coded as spatial symbols in

0 030-4018/89/$03.50

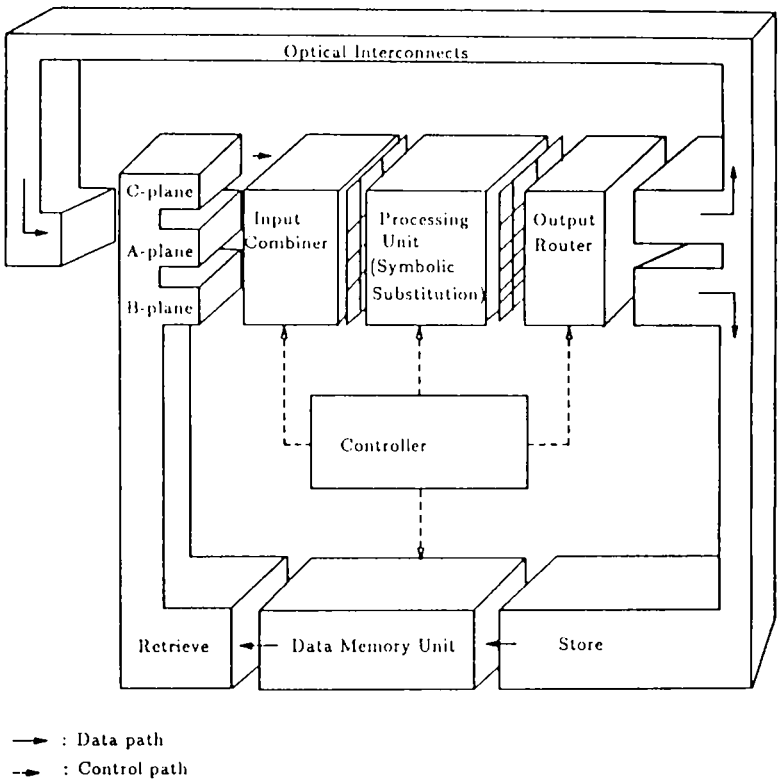


Fig. 1. The architecture of an optical bit-plane array processor.

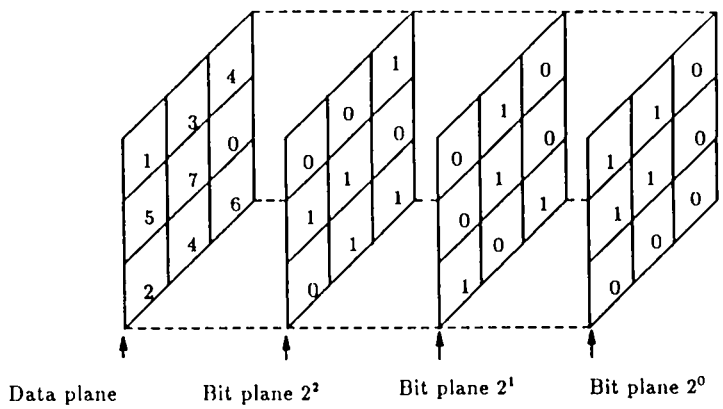


Fig. 2. Data representation as a stack of bit planes.

the input planes. Computation proceeds in transforming these symbols into other symbols according to a set of symbolic substitution rules specifying how to replace every input symbol. The processing unit is equipped with three fundamental operations: a *logical NOT* which inverts all the entries of an input plane, a *logical AND*, denoted by $\triangle$, that performs the logical and of the overlapping bits of two bit planes, and a *full ADD*, denoted by $\uplus$, that performs the full addition of the overlapping bits of the three input planes. By overlapping bits, it is meant bits with the same cartesian coordinates (i,j) in the input

planes. These operations constitute a complete arithmetic and logic set capable of computing any arithmetic or logic function.

2.2. Input/output data routing

The data represented as bit planes is fed to the system through three input planes, namely A-, B-, and C-plane as shown in fig. 1. Depending on the fundamental operation needed at a given computational step, the input combiner performs three data movement functions: for the logical NOT, it simply latches the relevant input plane to the processing unit. For the logical AND, the data movement required is called the *2-D perfect shuffle*. This function performs the shuffling of the row position, i, of the data such that the overlapping bits from the two planes become spatially adjacent. This function does not affect the column position, j. The data movement required for the full ADD operation is called the *2-D 3-shuffle*. This function is similar to the 2-D shuffle function except that it performs a 3-way shuffling of the rows of the three input planes [5].

The output router is responsible for directing the processed data to its appropriate destination. It also performs three data movement functions, namely, *feeding back* to the input combiner, a partial result such as a carry bit plane resulting from a full ADD operation, sending a final result to the data memory for storage, and *shifting* the output either in the X or Y direction by a variable number of pixels. This shift enables communication between pixels in the plane. By means of this spatial shifting, data can be moved among widely and at arbitrarily separated locations in the plane.

3. Optical implementation considerations

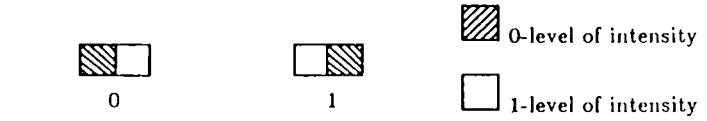
In order to process information optically, we use light intensity and positional coding for the data representation. A possible representation is to encode the logic value 0 by two pixels dark-bright, and the logic value 1 by the inverse pattern, bright-dark as shown in fig. 3a. In this coding scheme, a logic value is represented not only by the intensity of the bright pixel but also by its position, which has some implementation advantages [6].

3.1. Optical substitution rules for 2-D arithmetic and logic

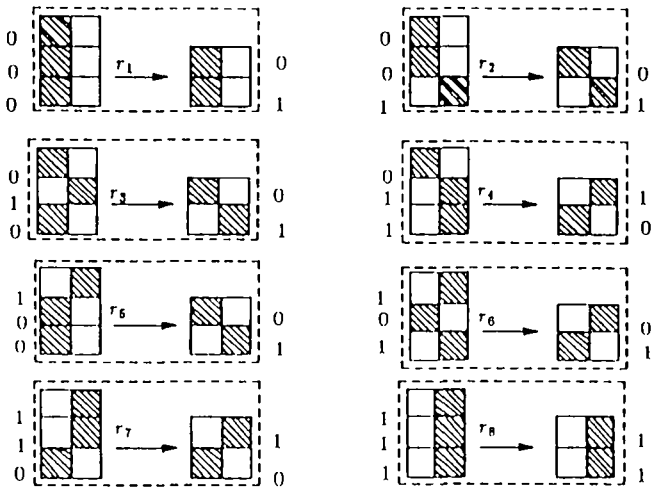
Figs. 3 (b–d) depicts the symbolic substitution rules required to optically implement the fundamental operations: logical NOT, logical AND, and full ADD. These SS rules are derived from the truth table specifications of these operations. The left-hand side patterns (or search patterns) of the SS rules represent the input combinations and the right-hand sides (or replacement patterns) represent the table entries. The full ADD operation manipulates three bits which gives rise to eight combinations. If we put the bit symbols on the top of each other, we produce eight SS rules for the full ADD. Similary, the logical NOT, and AND give rise to two and four SS rules respectively. Note that for the logical AND and the full ADD operations, each bit is provided by a separate bit plane. These bits have the same coordinates i, j in each plane. The grouping of bits into left-hand patterns is accomplished by the data movement functions described earlier. Optical implementation of the two processing steps involved in symbolic substitution (pattern recognition and pattern substitution) have been suggested by several researchers [6–9].

3.2. Implementation of the processing unit

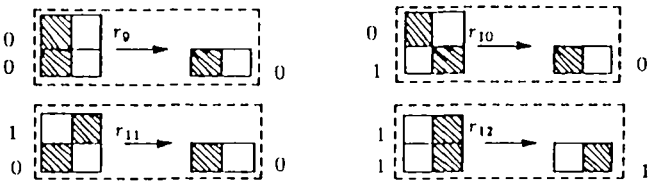
In order to process several SS rules simultaneously, the output of the input combiner is replicated a number of times equating the number of SS rules to be activated at a given stage of computation; for example, to perform the pairwise addition of three input planes, we need to replicate the formatted plane (output of the input combiner) eight times corresponding to the eight SS rules associated with it. Each copy is sent to one of the eight SS rules r_1 to r_8 in fig. 3. After the necessary substitutions, the outputs of all the active SS rules are optically superimposed to form the processed result. Thus the processing unit can be implemented with three modules, namely, an ADD module, an AND module and a NOT module as illustrated in fig. 4. Each module comprises the SS rules of the corresponding operation. A dynamic beam steering element (an acousto-optic or electro-optic deflector) is used to deflect the input plane to the desired module.



(a) Light intensity coding of the values 0 and 1.



(b) Optical SS rules for the full ADD operator



(c) Optical SS rules for the logical AND operator

(d) Optical SS rules for the logical NOT operator

Fig. 3. Optical SS rules for the fundamental operation: full ADD, logical AND and logical NOT.

3.3. Data routing and memory organization

The input and output router perform only data movement functions, no processing is required. A wide variety of optical methods can be imagined for realizing the data movement functions described earlier [10].

To maintain the 2-D processing throughout the system, the data memory must be bit-plane addressable. For single plane storage, such as the input and output planes, SLM technology and bistable optical latches [11,12] can be used. However, this would not be sufficient to build a data memory unit capable of holding a large number of bit planes. At present

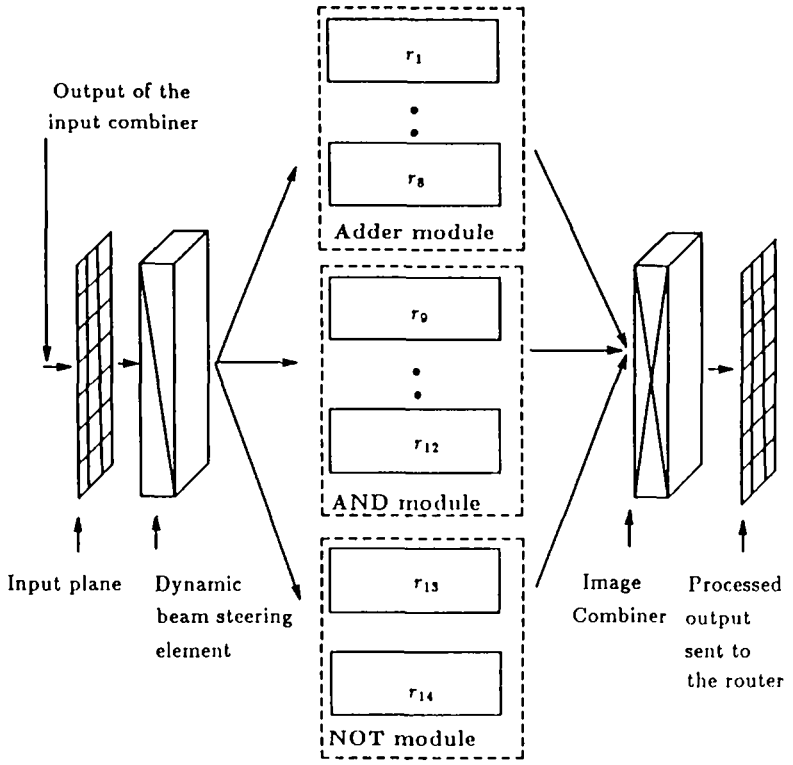


Fig. 4. The logical structure of the processing unit.

time, no real-time optical memory exists that can achieve read and write of optical data in 2-D format at high-speed. However, volume holograms, with their ability to store information in three dimensions show the potential for a dramatic increase in optical storage density [13]. High storage density can be achieved in random access optical memory by recording stacked holograms in photorefractive crystals. Another way of implementing the data memory would be the extension of the optical disk technology. Although optical disks are write-once today, their extremely large storage capacity may render their nonerasability relatively unimportant for the lifetime of the computation intended by the optical architecture. The execution sequence and the data flow may be controlled by a very fast microprocessor that executes an instruction program and generates control signals to the optical devices.

4. Mapping data-parallel algorithms

The bit-plane architecture exploits data parallelism at the hardware level, which enables it to process an entire data plane at once. To enforce this capability at the algorithm design level, we view the design and the mapping process as a hierarchical structure as shown in fig. 5. At the highest level of the hierarchy is the application we wish to solve, i.e. signal and image processing, vision, radar application, etc. The next level identifies the various algorithms that can be used to compute these applications; these include matrix algebra, numerical transforms, solutions of partial differential equation, etc. A further analysis of these algorithms reveals that they share a common set of high-level operations, which we call *computing substructures*. These substructures can in turn be decomposed into a set of fundamental operations such as the full ADD, the logical AND and NOT. The rationale behind this approach is that numerous data-parallel algorithms share common fea-

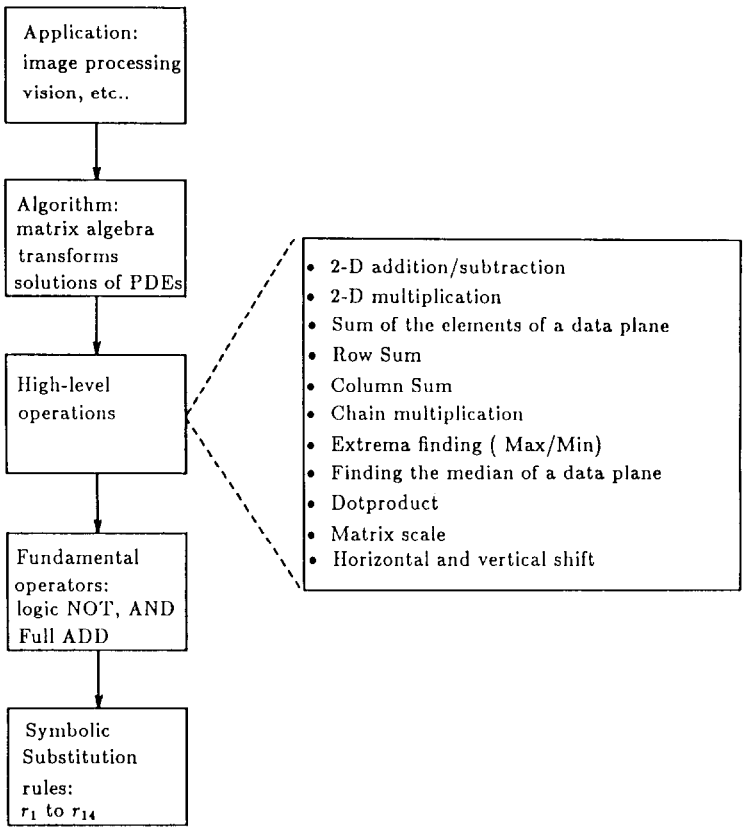


Fig. 5. A hierarchical top-down approach to the mapping of algorithms onto the bit-plane architecture.

tures such as localized operations, intensive computations, matrix operations and communication patterns. The high-level computing substructures are meant to capture these features. These substructures are directly mapped onto the hardware, and parallel algorithm are built upon these constructs so as to provide an efficient algorithm-architecture mapping. In this paper, we focus on the implementation of a sample of these computing substructures, and show how they can be used to efficiently map parallel algorithms onto the architecture.

In what follows, the boldface notation i.e. **X**, **Y** etc. denotes a data plane (or a stack of bit planes), and the italic notation (i.e. X, Y) designates a single bit plane. The notation $A(B \text{ or } C) \leftarrow X$ is interpreted as data transfer from memory location X to input plane $A(B$ or $C)$. The notation $X \leftarrow Y$ denotes data transfer from memory location Y to X. This involves loading Y, going through the processing unit without any effect, and storing it in X. The notation $C \leftarrow 0$ is interpreted as loading the C-plane with a zero bit plane (all entries are 0). Loop indices and parameter calculation such as "for $i=a$ to b" should be interpreted as control instructions that are executed by the control unit.

4.1. 2-D addition/subtraction

This substructure refers to the addition (subtraction) of corresponding elements of two $n \times n$ data planes **X** and **Y** of integers. The result is a data plane $\mathbf{S}=\{\mathbf{s}_{ij}\}$, where $\mathbf{s}_{ij}=\mathbf{x}_{ij} \pm \mathbf{y}_{ij}$ for $i, j=1, ..., n$. Let **X** be an $n \times n$ q-bit planes, $X_{q-1}, X_{q-2}, ..., X_0$ where q is the precision of the operands, X_0 being the least significant and X_{q-1} being the most significant bit planes respectively. Similar considerations take place for the data plane **Y**, the procedure is as follows

Procedure 2-D Addition(X,Y)
begin
$C \leftarrow 0$;
for $k:=0$ **to** $q-1$ **do**
$A \leftarrow X_k$;
$B \leftarrow Y_k$;
$S_k, C_{out} \leftarrow A \oplus B \oplus C_{in}$;
endfor
$S_q \leftarrow C$;
end 2-D Addition

The notation $S_k, C_{out} \leftarrow A \oplus B \oplus C_{in}$, in the above procedure, designates the addition of bit planes A and B together with the previous carry C_{in}; the sum bit plane is to be stored in S_k and the resulting carry bit plane, C_{out}, is routed back to input plane C (C_{in} and C_{out} represent the same physical location). The procedure starts by initializing the C-plane to zero, and loading bit planes X_0, Y_0 into A-plane and B-plane respectively. The processing unit applies the full ADD SS rules simultaneously to the 2-D 3- shuffled plane. The sum bits are extracted from the output plane through masking operations and stored in S_0, and the carry bits are extracted and fed back to the C-plane for the next iteration, while the memory unit loads the bit planes X_1 and Y_1 in the A-plane and B-plane respectively. The whole process continues until X_{1-1} and Y_{q-1} are added and the sum S_0, S_1, ..., S_q stored as a stack of bit planes in the memory. The addition of two q-bit planes is done in q iterations, regardless of the number of operands to be added.

Representation of numbers in two's complement form allows 2-D subtraction by adding few additional steps to the 2-D addition procedure. The pairwise subtraction of two data planes **X**, **Y** is done by first forming the two's complement of the subtrahend **Y**, then add it to **X**, using the 2-D addition substructure.

4.2. 2-D multiplication

This substructure refers to the multiplication of overlapping elements of two data planes. Let **X** and **Y** be as described previously, then the product **P** is a $2q$-bit planes $\mathbf{P}=P_{2q-1}P_{2q-2}...P_0$, where $\mathbf{p}_{ij}=\mathbf{x}_{ij}\times\mathbf{y}_{ij}$. This substructure uses the logical AND and the full ADD operations. The complete procedure is as follows

Procedure 2-D Multiplication(X,Y)
begin
for $k:=0$ **to** $2q-1$ **do** $P_k \leftarrow 0$;
for $l:=0$ **to** $q-1$ **do**
$C \leftarrow 0$;
for $m:=0$ **to** $q-1$ **do**
$A \leftarrow X_m$;
$B \leftarrow Y_l$;
$B \leftarrow A \triangle B$;
$A \leftarrow P_{m+l}$;
$P_{m+l}, C \leftarrow A \oplus B \oplus C$;
endfor
$P_{q+l} \leftarrow C$;
endfor
end 2-D Multiplication

The time complexity of the 2-D multiplication is $O(q^2)$, independent of the number of pairs to be multiplied. Note that, unlike the conventional shift and add multiplication algorithm, we did not need to shift the previous partial product to generate the current one. Instead, we start the addition at the bit plane corresponding to the amount of shift required.

4.3. 2-D shift

We define two substructures for shifting a data plane by a variable number of pixels. The shift considered here is the logical shift, where columns (or rows) of 0s enter the opposite direction of the shift. Given $\mathbf{P}=P_{q-1}P_{q-2}...P_0$, and $\mathbf{X}=X_{q-1}X_{q-2}...X_0$, we define a horizontal shift substructure, denoted by $\mathbf{H}_\alpha(\mathbf{P})$, to be the data plane **P** shifted in the X-axis by α columns ($+\alpha$ for positive shift, and $-\alpha$ for negative shift). The amount of shift α is applied to every bit plane P_i comprising the data plane **P**. The shifted plane can be either stored in itself or in a different memory location, therefore the notation $\mathbf{X} \leftarrow \mathbf{H}_\alpha(\mathbf{P})$ is interpreted as shifting the data plane **P** by α columns and storing it in **X**. Similarly, we define the vertical shifting operation, denoted by $\mathbf{V}_\alpha(\mathbf{P})$, to be the data plane **P** shifted along the Y-axis by α rows ($+\alpha$ for upward shift, and $-\alpha$ for downward shift).

4.4. Row accumulation

This refers to calculating the sum of all the elements of a data plane columnwise. The initial plane

S is split horizontally (using vertical shifting) into two planes X and Y, each with half the data entries of S; next these planes are added using the 2-D addition procedure. This split and add process is repeated for $\log_2 n$ iterations, after which, the first row of S holds the sums of all the rows of S. In other word, the elements of each column are accumulated and stored in the first entry of each column.

Procedure Row-Sum(S,X,Y)
begin
for $k=1$ **to** $\log_2 n$ **do**
$\alpha := n/2^k$;
$\beta := \Sigma_{i=1}^{i=k} (n/2^k)$;
$\mathbf{X} \leftarrow \mathbf{V}_{-\beta}(\mathbf{S})$;
$\mathbf{V}_{+\beta}(\mathbf{X})$;
$\mathbf{Y} \leftarrow \mathbf{V}_{+\alpha}(\mathbf{S})$;
S←2-D Addition (X,Y);
endfor
end Procedure Row-sum

4.5. Column accumulation

This substructure refers to the accumulation of the elements of a data plane rowwise. It is similar to the Row-Sum substructure, with the exception that the elements of the data plane are accumulated along the rows. Initially, the data plane S is split vertically (using horizontal shifting) into two data planes X and Y. These planes are then added using the 2-D addition substructure. The same steps are repeated for $\log_2 n$ iterations, after which the first column of S contains the accumulated columns.

Procedure Column-Sum(S,X,Y)
begin
for $k=1$ **to** $\log_2 n$ **do**
$\alpha := n/2^k$;
$\beta := \Sigma_{i=1}^{i=k} (n/2^k)$;
$\mathbf{X} \leftarrow \mathbf{H}_{-\beta}(\mathbf{S})$;
$\mathbf{H}_{+\beta}(\mathbf{X})$:
$\mathbf{Y} \leftarrow \mathbf{H}_{+\alpha}(\mathbf{S})$;
S←2-D Addition (X,Y);
endfor
end Procedure Column-Sum

4.6. Matrix multiplication

As an example of algorithm mapping, we present a parallel algorithm for matrix multiplication which is based on the use of the computing substructures introduced above. Let X and Y be $n \times n$ matrices (assuming same size for simplicity) then their product $\mathbf{X} \times \mathbf{Y} = \mathbf{Z}$ is an $n \times n$ matrix whose elements are given by

$$\mathbf{z}_{ij} = \sum_{k=1}^{k=n} \mathbf{x}_{ik}\mathbf{y}_{kj}, \quad i,j = 1, ..., n. \tag{1}$$

We assume that the matrix X is stored as n matrices of size $n \times n$: $\mathbf{X}^n$, $\mathbf{X}^{n-1}$, ..., $\mathbf{X}^1$, where each matrix $\mathbf{X}^i$ is formed by transposing the ith row of X and replicating it n times. Put differently, each column of $\mathbf{X}^i$ is equal to the ith row of X, for $i = 1, ..., n$. Let $\mathbf{T}_k$ be the matrix formed by the 2-D multiplication of matrices $\mathbf{X}^k$ and Y, then $\mathbf{T}_k = \{\mathbf{t}_{ij}\}$, where $\mathbf{t}_{ij} = \mathbf{x}_{kj}\mathbf{y}_{ij}$ for $i,j = 1, ..., n$. Thus summing the elements of each column of $\mathbf{T}_k$ using the Row-Sum procedure will produce a matrix say $\mathbf{Z}_k$ whose first row presents the kth row of matrix Z:

$$\mathbf{Z}_k = \sum_{i=1}^{i=n} t_{ij}, \quad j = 1, ..., n, \tag{2}$$

where the first row of $\mathbf{Z}_k$ represents the kth row of Z and all the other rows are 0s. By repeating these steps for all values of k ($k = 1, ..., n$), we produce n matrices $\mathbf{Z}_n$, $\mathbf{Z}_{n-1}$, ..., $\mathbf{Z}_1$. The first row of each matrix $\mathbf{Z}_i$ represents the ith row of the final product matrix Z. Each matrix Z_k is shifted by $1-k$ rows downward. All the shifted matrices are then added pairwise to produce the final matrix Z:

Procedure Matrix Multiply(Z,X,Y)
begin
for $k := 1$ **to** n **do**
$\mathbf{T}_k \leftarrow$ 2-D Multiplication($\mathbf{X}^k$, Y);
$\mathbf{Z}_k \leftarrow$ Row Sum($\mathbf{T}_k$);
endfor
for $k := 1$ **to** n **do** $\mathbf{V}_{(1-k)}(\mathbf{Z}_k)$;
for $k := 1$ **to** n **do** Z←2-D Addition (Z,$\mathbf{Z}_k$);
end Matrix Multiply

The time complexity of the algorithm is $O(n(q\log_2 n + q^2))$, where q is the operand length. It can be seen that the time complexity of the multi-

plication algorithm is logarithmic in $n(O(n \log n))$ as opposed to cubic in $n(O(n^3))$ for the conventional triple loop matrix multiplication.

5. Performance analysis

In this section, we estimate the theoretical performance of the optical architecture by evaluating several performance measures and compare them to the ones of existing SIMD array processors.

5.1. Asymptotic performance

Hockney and Jesshope [14] have introduced two parameters (r_∞, $n_{1/2}$) to give a first-order characterization for the *asymptotic performance* of a parallel computing system. The first parameter: r_∞ gives a quantitative measure of the maximum rate of computation in units of equivalent scalar operations performed per second. For an array processing system, r_∞ is evaluated as follows [14]

$$r_\infty = n^2 t_{\text{array}}, \tag{3}$$

where t_{array} is the time taken to execute one operation on all the PEs, this is usually taken as the processing rate, and n^2 is the total number of PEs. The *half-performance length*: $n_{1/2}$ characterizes the amount of hardware parallelism in a computer architecture. For a nonpipelined array processor, the factor $n_{1/2}$ is defined to be the vector length required to achieve half the maximum performance $(r_\infty/2)$ [14]. For an array processing system, $n_{1/2}$ is half the array size $n^2/2$ [14].

The MPP [15] with an array of 128×128 PEs and 10 MHz rate has achieved 6×10^9 8-bit operations/s. The CLIP [16] with a 96×96 array and a 25 μs cycle time has achieved 3.7×10^9 bit operations/s. The ICL DAP [17] with a 64×64 array and 0.2 μs cycle time, was described to achieve 10^8 32-bit operations/s. The Connection Machine [18] with 65536 PEs and a 0.5 μs machine cycle time can achieve 13×10^{10} bit operations/s (the CM-2 model). For the optical case, if we assume that the processing unit is formed with NOR-gate arrays [6] of size 1000×1000, and 100 MHz processing rate, $r_\infty = 10^{14}$ bit operations/s.

5.2. Communication and I/O capabilities

Communication plays a crucial part in determining the overall system performance. There are many communication metrics in the literature, we choose the most widely used for our purposes:

Communication bandwidth is the maximum number of messages that can be simultaneously exchanged in one time step. Hence the bandwidth of the optical system is $O(n^2)$, since up to n^2 PEs can send and receive data at a time. The data transmission in the MPP and the CLIP is one column at a time, therefore their bandwidth is $O(n)$, the DAP on the other hand, transmits data in a row-parallel fashion which amounts to the same bandwidth factor $O(n)$. The Connection Machine has a maximum sustained communication bandwidth of $O(n^2)$.

The *diameter* is the maximum number of communication cycles (or links) needed for any two PEs to communicate. For the optical case, this factor is 1, since we allow any number of shifts in either directions in one cycle time. The MPP and the DAP are mesh-connected and therefore have a diameter of $2(n-1)$. The CLIP has a hexagonal connectivity and therefore has a diameter of $n\sqrt{2}$. The Connection Machine, has a diameter of $O(\log_2 n)$.

Broadcasting is the ability to send the value in a certain PE to all the other PEs. The amount of communication cycles to achieve this is considered a measured of communication performance. This value is $O(n)$ for the DAP, MPP, and CLIP, and $O(\log_2 n)$ for the Connection Machine. As far as the optical system is concerned, broadcasting a value in one PE to all other $n^2 - 1$ PEs can be done in $O(\log_2 n)$ steps.

In current implementations of the MPP and the CLIP, I/O is handled in column-parallel fashion while the DAP is row-parallel (data is loaded into the processing array one column or one row at a time). By contrast with the optical system, I/O activities are handled in plane-parallel manner. This ability gives the optical system an I/O speedup of n, for an $n \times n$ input image, over the MPP, CLIP and the DAP which could be a tremendous speed advantage, considering the large potential value of n(eventually 1000). Table 1 summarizes the various performance measures considered above.

Table 1.
Performance comparison of the optical bit-plane architecture with electronic array processors

Computing System	Performance metrics						
	Maximum performance (r_∞)	Parallelism ($n_{1/2}$)	Diameter	Bandwidth	Broadcasting	I/O capability	Cycle time (μs)
Optical Architecture (1000×1000) PEs	10^{14} bit op/s	500000	1	n^2	$O(\log_2 n)$	n^2	10^{-2} (potentially)
MPP (128×128)	6×10^9 8-bit op/s	8192	$2n-2$	n	$O(n)$	n	10^{-1}
DAP (64×64)	3.7×10^8 32-bit op/s	2048	$2n-2$	n	$O(n)$	n	2×10^{-1}
CLIP (96×96)	3.7×10^9 bit op/s	4608	$\sqrt{2}n$	n	$O(n)$	n	25
Connection Machine (64K PEs)	13×10^{10} bit op/s	32768	$O(\log_2 n)$	n^2	$O(\log_2 n)$	n^2	5×10^{-1}

n^2 = the total number of PEs. (processing array size).
r_∞ = the asymptotic performance = processing array size/processing time (Hockney and Jesshope).
$n_{1/2}$ = amount of hardware parallelism. It is the vector length required to achieve half the maximum performance. In case of array processing, half the maximum performance is achieved with half the array size ($n^2/2$).

6. Conclusions

In this communication, a parallel optical computing model based on symbolic substitution is introduced as well as a hierarchical mapping technique for mapping parallel algorithms onto it. Several numerical algorithms were mapped onto the architecture. Initial theoretical performance analysis of the proposed system was conducted. The analysis has shown that the optical architecture has a great potential for outperforming existing array processors. Hence, it is an attractive alternative to current computing systems for applications that require processing large amounts of data at high-speed. Furthermore, the communication flexibility and parallel I/O of the optical system seems to be unmatchable by other electronic processors.

References

[1] A. Huang, in: Proc. IEEE Tenth Intern. Optical Computing Conf., Catalog 83CH1880-4, 1983, p. 13.
[2] A.A. Sawchuk and T.C. Stand, Proc. IEEE 72 (1984) 758.
[3] J. Tanida and Y. Ichioka, Appl. Optics 25 (1986) 1565.
[4] T.J. Drabik and S.L. Lee, Appl. Optics 25 (1986) 4053.
[5] A. Louri and K. Hwang, in: Proc. 15th Intern. Symp. on Computer Arch., (Honolulu, Hawaii), IEEE/ACM, 1988.
[6] K.H. Brenner, A. Huang and N. Streibl, Appl. Optics 25 (1986) 3054.
[7] E. Botha, D. Casasent and E. Barnard, Appl. Optics 27 (1988) 817.
[8] J.N. Mait and K.H. Brenner, Appl. Optics 27 (1988) 1692.
[9] K. Hwang and A. Louri, Optical multiplication and division using signed-digit symbolic substitution, Optical Engineering, special issue on Optical Computing, March 1989, to be published.
[10] A.W. Lohmann, Appl. Optics 25 (1986) 1543.
[11] A.D. Fisher and J.N. Lee, in: Proc. SPIE, Optical and Hybrid Computing 634 (1986) 352.
[12] S.D. Smith, Appl. Optics 25 (1986) 1550.
[13] D. Psaltis, J. Yu, X.G. Gu and H. Lee, Technical Digest, OSA Topical Meeting on Optical Computing, pp. TuA3.1-TuA3.4, 1987.
[14] R.W. Hockney and C.R. Jesshope, Parallel computers: architecture, programming and algorithms (Adam Hilger Ltd., Bristol, 1981).
[15] K.E. Batcher, IEEE Trans. on Computers, C-29 (1980) 83.
[16] M.J. Duff, in: CLIP 4: Special Computer Architecture for Pattern Recognition, eds. Fu and Ichkana (CRC Press, 1982).
[17] S.F. Reddaway, in: First Annual Symposium on Computer Architecture, (Florida) IEEE/ACM, 1973, p. 61.
[18] Thinking Machine Corporation, Connection machine model CM-2 technical summary, Tech. Rep. Series HA87-4, Thinking Machine Corporation, 1986.