JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **13**, 118–122 (1991)

A Fast Algorithm for Gaussian Elimination over *GF*(2) and Its Implementation on the GAPP*

Çetin K. Koç and Sarath N. Arachchige

Department of Electrical Engineering, University of Houston, Houston, Texas 77204

A fast algorithm for Gaussian elimination over $GF(2)$ is proposed. The proposed algorithm employs binary search technique to locate 1s along the columns of the large binary matrix being triangularized. The algorithm requires $2m^2 + m \log_2 n - 2m$ bit operations to triangularize an $n \times m$ matrix on a bit-array with n processors and $m + 2 + \lceil \log_2 n \rceil$ bits of vertical memory per processor. Details of an implementation on the Geometric Arithmetic Parallel Processor are also presented. © 1991 Academic Press, Inc.

1. INTRODUCTION

The implementation of the continued fraction method for factorization of large integers requires triangularization of a large Boolean matrix. The operations are performed in the Galois field with two elements, i.e., $GF(2)$. Let Q be the set $\{Q_1, Q_2, \ldots, Q_n\}$ where each Q_i is a positive integer. The continued fraction method requires the determination of a subset $R \subset Q$ such that the product of all $Q_i \in R$ is a perfect square. This is achieved by first factoring each Q_i over the set of predetermined prime numbers $P = \{p_1, p_2, \ldots, p_m\}$, i.e.,

$$Q_i = p_1^{a_{i1}} p_2^{a_{i2}} \cdots p_m^{a_{im}}$$

for all $i = 1, 2, \ldots, n$. Now consider the $n \times m$ matrix

$$A = [A_{ij}] = [a_{ij} \pmod 2)]$$

with 0–1 entries for $1 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant m$. We apply *elementary column operations* to this matrix over the field $GF(2)$, and mark each row in which a pivot element is located. An unmarked nonzero row at the end of the elimination process implies the existence of a subset R satisfying the above property [10].

The continued fraction method for factorization of large integers was proposed by Lehmer and Powers [9]. This algorithm was first implemented on a computer by Morrison and Brillhart [10] to factor $F_7 = 2^{2^7} + 1$. The availability of massively parallel processors made this algorithm very attractive. Parkinson and Wunderlich [13] implemented this algorithm on the ICL Distributed Array Processor (DAP). Wunderlich [17] has also implemented the continued fraction method on the NASA-Goodyear Massively Parallel Processor (MPP). The details of the parallel algorithm can be found in [18].

Implementation of the Gaussian elimination algorithm over $GF(2)$ (more generally, over $GF(p)$) on systolic computing systems and in VLSI has been considered by several researchers. Takefuji *et al.* have proposed a matrix solver using iterative logic circuits suitable for VLSI implementation [16]. Systolic algorithms for Gaussian elimination of dense matrices without pivoting have been known [7, 1]. A systolic array for triangularizing real matrices using partial pivoting was given by Barada and El-Amawy [2]. This array has $O(n^2)$ processors, however, it requires $O(n^2)$ time due to the complications in pivot searching for matrices with real number entries.

Partial pivoting cannot be avoided for matrices with entries over $GF(2)$ since every other element may be zero. Hochet *et al.* [8] and Cosnard *et al.* [4] have proposed systolic arrays for Gaussian and Gauss–Jordan elimination and triangularization algorithms over $GF(p)$. These arrays have $O(n^2)$ processors and accomplish their task in $O(n)$ time. These algorithms triangularize the matrix as the elements of the matrix enter the two-dimensional systolic array from the north end. The pivot searching strategy is very simple. During the jth stage of the algorithm the jth column of the matrix enters the jth processor on the diagonal. As the elements of the jth column enter, we check if $A_{ij} = 0$ for $j \leqslant i \leqslant n$. If this element is zero, it continues to flow; it is not chosen to be a pivot, and it does not need to be updated. We pick the first nonzero element as the pivot. Once the pivot is selected, we compute the coefficients necessary to update every row which contains a nonzero element in its jth position.

In this paper, we assume that matrix A is already in the array's memory. In the implementation of the continued fraction algorithm, matrix A may be produced in the processor array. Also, most massively parallel array processors have fast means of I/O from and to the host computer. For

* This research was partially supported by the NCR Corporation, Micro Electronics Division, Fort Collins, CO.

example, the Geometric Arithmetic Parallel Processor (GAPP) has a Data Communication line (CM) which is a separate I/O bus that accepts data from the south end of the array, and shifts it to the north without interfering with ongoing computations [12]. Thus, the matrix can be entered into the array and saved in the memory very quickly.

We present a fast algorithm which uses binary search technique to find the pivot. We first explain the Gaussian elimination algorithm over $GF(2)$ and give an example in Section 2. The parallel algorithm and its mapping have been described in Section 3. The details of the implementation on the GAPP using the GAPP PC Development System are given in Section 4. In the last section, we summarize the results and discuss future work.

2. THE ALGORITHM

The Gaussian elimination over $GF(2)$ on the matrix A requires elementary column operations rather than elementary row operations. Here we give an example which will be useful in understanding the steps of the algorithm. Let $Q = \{6, 42, 105, 20, 63\}$, and $P = \{2, 3, 5, 7\}$. Since we have

$$6 = 2^1 3^1 5^0 7^0, \quad 42 = 2^1 3^1 5^0 7^1, \quad 105 = 2^0 3^1 5^1 7^1,$$
$$20 = 2^2 3^0 5^1 7^0, \quad 63 = 2^0 3^2 5^0 7^1,$$

we obtain A as

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 2 & 0 & 1 & 0 \\ 0 & 2 & 0 & 1 \end{bmatrix} \pmod 2 = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

On this matrix we perform elementary column operations and mark each row on which we find a pivot. We start with the first column, and search for a pivot in this column. Since $A_{11} = 1$, A_{11} is selected as the pivot element and column 1 as the pivot column. We then perform a search for 1s in the row in which the pivot element is found. If element $A_{1k} = 1$ for $k \neq 1$ is nonzero, we add column 1 to column k (addition is done in $GF(2)$). Since $A_{12} = 1$, we add column 1 to column 2 to obtain the updated column 2. This will produce

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \star \\ 1 & 0 & 0 & 1 & \\ 0 & 1 & 1 & 1 & \\ 0 & 0 & 1 & 0 & \\ 0 & 0 & 0 & 1 & \end{bmatrix},$$

where $\star$ denotes the marking. We then proceed to pick column 2 and search for a pivot in this column. Since $A_{32} = 1$, we mark row 3, and search for 1s in row 3. We find $A_{33} = A_{34} = 1$, thus add column 2 to columns 3 and 4:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \star \\ 1 & 0 & 0 & 1 & \\ 0 & 1 & 0 & 0 & \star \\ 0 & 0 & 1 & 0 & \\ 0 & 0 & 0 & 1 & \end{bmatrix}.$$

The next pivot is A_{43}; however, there are no other 1s in row 4. We only mark row 4 and obtain

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \star \\ 1 & 0 & 0 & 1 & \\ 0 & 1 & 0 & 0 & \star \\ 0 & 0 & 1 & 0 & \star \\ 0 & 0 & 0 & 1 & \end{bmatrix}.$$

The last pivot found is A_{24} and since $A_{21} = 1$ we add column 4 to column 1, and obtain

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \star \\ 0 & 0 & 0 & 1 & \star \\ 0 & 1 & 0 & 0 & \star \\ 0 & 0 & 1 & 0 & \star \\ 1 & 0 & 0 & 1 & \end{bmatrix}.$$

Note that row 5 has not been used. Since $A_{51} = A_{54} = 1$, row 5 and all rows i for which $A_{i1} = 1$ or $A_{i4} = 1$ are dependent. From the above matrix we see that rows 1, 2, and 5 are dependent. If we sum row 1, row 2, and row 5 in $GF(2)$, we obtain a zero row:

$$\begin{array}{rl} 1000 & \text{Row 1 } (Q_1 = 6) \\ 0001 & \text{Row 2 } (Q_2 = 42) \\ \oplus\ 1001 & \text{Row 5 } (Q_5 = 63) \\ \hline 0000 & \end{array}$$

This implies that $R = \{Q_1, Q_2, Q_5\}$ and product $Q_1Q_2Q_5$ forms a perfect square:

$$Q_1Q_2Q_5 = 6 \cdot 42 \cdot 63 = (2 \cdot 3) \cdot (2 \cdot 3 \cdot 7) \cdot (3^2 \cdot 7)$$
$$= (2 \cdot 3^2 \cdot 7)^2 = 126^2$$

We summarize the algorithm below.

```
FOR $j = 1, 2, \ldots, m$ DO
    BEGIN
    Search for $A_{ij} = 1$ in column $j$
    IF found THEN
        BEGIN
        Mark row $i$
        FOR $k = 1, 2, \ldots, j-1, j+1, \ldots, m$ DO
            $A_{ik} = 1$ THEN add column $j$ to column $k$
        END
    END
```

In the following proposition, we count the number of bit operations required by the Gaussian elimination algorithm over $GF(2)$. A *bit operation* is defined as a test operation on a single-bit variable (e.g., $A_{ij} = 1$) or a binary operation in $GF(2)$.

PROPOSITION 1. *The sequential Gaussian elimination algorithm requires $m^2n + m^2 - m$ bit operations to triangularize an $n \times m$ matrix.*

Proof. The search for the first 1 on the jth column requires n bit operations in the worst case. For all m columns, we perform mn operations. Once a 1 is located in row i, we check if $A_{ik} = 1$ for $k = 1, 2, \ldots, j-1, j+1, \ldots, m$, and add column j to column k if this is true. This step requires a single test operation $A_{ik} = 1$, and n bit operations for the addition of column j to column k. Since these operations have to be performed for $j = 1, 2, \ldots, j-1, j+1, \ldots, m$ and $i = 1, 2, \ldots, n$, a total of $m(m-1)(1+n)$ steps is needed. Thus, the total number of sequential bit operations required by the Gaussian elimination algorithm is $mn + m(m-1)(n+1) = m^2n + m^2 - m$. ■

3. THE PARALLEL ALGORITHM

The $n \times m$ matrix can be mapped on a $p \times q$ rectangular array of pq processors in several ways. Our implementation views the $p \times q$ rectangular array of processors as a linear array with pq processors. Suppose that each one of the pq processors has a vertical memory of L bits. A pivot column in the Gaussian elimination algorithm over $GF(2)$ is simply a column that has a 1. If we add the pivot column to another column having a 1 in the same position, then the resulting element will be equal to 0, since $1 + 1 = 0 \pmod 2$ or $1 \oplus 1 = 0$. The addition operation is performed for all elements of these two columns. Thus, we map the $n \times m$ matrix in such a way that each row is in one of the single-bit processors' memory, requiring $n = pq$ and $m = L$. For example, the DAP has 64×64 processors where each processor has 4096 bits of vertical memory [15], i.e., $p = q = 64$ and $L = 4096$. The largest matrix that can be triangularized on the DAP has dimensions 4096×4096. The GAPP chip consists of 6×12 single-bit processors with 128 bits of memory per processor [5, 12], i.e., $n = 72$ and $m = 128$.

This mapping, however, does not consider the amount of memory needed for bookkeeping operations, for example, the marking of the used rows. This is, however, not significant since one bit per processor is sufficient. This single-bit variable is named U throughout. Furthermore, in each processor, we reserve another single-bit variable W and a d-bit mask variable named M where $d = \lceil \log_2 n \rceil$. These variables are to be used by the binary search algorithm to determine the position of the first 1 in a column. Thus, an $n \times m$ matrix can be triangularized on a $p \times q$ bit-array in which each processor has L bits of vertical memory, where $n = pq$ and $m = L - 2 - d = L - 2 - \lceil \log_2 n \rceil$.

Since the result of a search operation has to be broadcast to all processors, the processor array needs to be equipped with a global output line. The Global Output Line (GO) provided on the GAPP chip has the following propety: it is pulled low whenever any of the single-bit registers NS contains 1. This is accomplished by tying together the outputs of the open-drain gates for each one of the NS registers to form wired-or logic. The Global Output line makes the GAPP chip very powerful since it provides global information about the local data.

Initially we save the matrix A in the array processor's memory, and assign $U = W = 0$. Let M_d be the most significant bit of variable M. We assign M as follows: dth bit is 1 for the first half of the processors, and is 0 for the second half of processors, $(d-1)$st bit is 1 for the first and the third quarters of the processors, is 0 for the second and fourth quarters of the processors, and so on. Thus, processor i has the n-bit binary number $(M_d M_{d-1} \cdots M_1) = 2^d - i$. An example of matrix A and the distribution of variables U, W, and M is shown in Fig. 1. The following operations are executed for $j = 1, 2, 3, \ldots, m$.

Searching for the First 1 *in Column j.* This is the first step of the algorithm. Let M_k be the kth bit of variable M in a processor, and A_{ij} be the jth bit in the ith row of A, which is located in processor i. We perform binary search (see, e.g., [6]) by executing the following code in all processors for $i = 1, 2, 3, \ldots, n$:

```
NS := $A_{ij}$
IF GO = 0 THEN
    BEGIN
    W := NS
    FOR $k = d, d-1, d-2, \ldots, 1$ DO
        BEGIN
        NS := $W \wedge M_k$
        IF GO = 0 THEN $W$ := NS
        END
    END
```

The search is started if the jth column of A is nonzero. First, this column is placed in register W. At step k we perform the AND operation of $W \wedge M_k$ for $k = d, d-1, \ldots, 1$. The result, if nonzero, is put in W; otherwise the content of W remains unchanged. Thus, at the end of this procedure, $W = 1$ in at least one of the processors. To prove that W contains at most one nonzero bit, suppose that $W_{i_1} = W_{i_2} = 1$ for $1 \leqslant i_1 < i_2 \leqslant n$. Mask variables in processor i_1 and i_2 have values $2^d - i_1 = (M_d M_{d-1} \cdots M_1)$ and $2^d - i_2 = (N_d N_{d-1} \cdots N_1)$, respectively. Since $2^d - i_1 > 2^d - i_2$, we must have $M_k = N_k$ for $k = d, d-1, \ldots, l-1$ and

Processor	A	U	W	M
1	0110001000	0	0	1111
2	0001110011	0	0	1110
3	0100110100	0	0	1101
4	0111001000	0	0	1100
5	1101011001	0	0	1011
6	0010011000	0	0	1010
7	1000001011	0	0	1001
8	1101001001	0	0	1000
9	0011010100	0	0	0111
10	0101000100	0	0	0110
11	1000010011	0	0	0101
12	0111101001	0	0	0100
13	0010100000	0	0	0011
14	0100101100	0	0	0010
15	0101010011	0	0	0001
16	0011010101	0	0	0000

FIG. 1. Matrix A and variables U, W, and M for $n = pq = 16$, $m = 10$, $d = 4$, and $L = 16$.

$M_l > N_l$ for some $d \leqslant l \leqslant 1$. For example, for $i_1 = 6$ and $i_2 = 7$ we have $2^4 - 6 = (1010)$ and $2^4 - 7 = (1001)$, i.e., $l = 2$ since $M_4 = N_4 = 1$, $M_3 = N_3 = 0$, and $M_2 > N_2$. The configuration of the masks forces W_{i_2} to be zeroed before W_{i_1} if $i_1 < i_2$. Therefore, when the search procedure ends, $W_i = 1$ in processor i which has the first 1 on the jth column of matrix A. In Fig. 2, values of W are shown at each step of the search algorithm.

Marking of Row i. Once the search procedure is finished we mark the row in which this 1 is located. This is simply accomplished by performing an OR operation.

$$U := U \vee W$$

Addition of Column j to Column k. The ith processor has immediate access to the ith row of matrix A. After the row is marked, we add column j to column k for all $k \neq j$ and $A_{ik} = 1$:

```
FOR $k = 1, 2, \ldots, j-1, j+1, \ldots, m$ DO
    BEGIN
    NS := $W \wedge A_{ik}$
    IF GO = 0 THEN $A_{ik} := A_{ik} \oplus A_{ij}$
    END
```

PROPOSITION 2. *The parallel Gaussian elimination algorithm requires $2m^2 + m\log_2 n - 2m$ bit operations to triangularize an $n \times m$ matrix on a bit-array with n processors and $m + 2 + \lceil \log_2 n \rceil$ bits of vertical memory per processor.*

Proof. Matrix A, single-bit variables W and U, and d-bit variables can be fed to the $p \times q$ bit-array from one end using $L = m + 2 + \lceil \log_2 n \rceil$ steps. We then start executing the parallel algorithm. In the following, we assume that the matrix and the bookkeeping variables are already in the array's memory, and count only the total number of bit operations required for the execution of the algorithm.

Once a particular column is selected, the algorithm proceeds to search for the first 1 in this column. This requires

	$k = 4$			$k = 3$			$k = 2$			$k = 1$			
Processor	W	M_4	NS	W	M_3	NS	W	M_2	NS	W	M_1	NS	W
1	0	1	0	0	1	0	0	1	0	0	1	0	0
2	0	1	0	0	1	0	0	1	0	0	0	0	0
3	0	1	0	0	1	0	0	0	0	0	1	0	0
4	0	1	0	0	1	0	0	0	0	0	0	0	0
5	1	1	1	1	0	0	1	1	1	1	1	1	1
6	0	1	0	0	0	0	0	1	0	0	0	0	0
7	1	1	1	1	0	0	1	0	0	0	1	0	0
8	1	1	1	1	0	0	1	0	0	0	0	0	0
9	0	0	0	0	1	0	0	1	0	0	1	0	0
10	0	0	0	0	1	0	0	1	0	0	0	0	0
11	1	0	0	0	1	0	0	0	0	0	1	0	0
12	0	0	0	0	1	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	1	0	0	1	0	0
14	1	0	0	0	0	0	0	1	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	1	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0
GO			0			1			0			0	

FIG. 2. Values of W at each step of the binary search algorithm.

d operations for each column, and thus md operations for all columns. We then add column j to column k for all $k \neq j$ and $j = 1, 2, \ldots, m$, where at each step a single test and a single addition is performed by each processor. This requires $m(m-1)2$ bit operations in the worst case. Thus, a total of $md + 2m(m-1) = 2m^2 + md - 2m$ bit operations is required. ■

4. IMPLEMENTATION ON THE GAPP

The parallel Gaussian elimination algorithm described above has been implemented on the GAPP. This is achieved by using the GAPP PC Development System [11]. It is composed of two parts: the hardware board which is compatible with the IBM-PC I/O bus and contains a 12×12 array of processor elements implemented with 2 GAPP chips, and the software package which allows the user to program the GAPP array using the high level programming language GAL (GAPP Algorithm Language) which is a subset of C. The GAPP PC Development System also has a facility for interactive debugging of the programs.

The 12×12 array allows the user to work with matrices as large as 144×128. Considering the memory space required for variables U, W, and the d-bit mask variable where $d = \lceil \log_2 144 \rceil = 8$, we see that the largest matrix size is limited to 144×118. We have implemented the continued fraction algorithm where the Gaussian elimination algorithm runs on the GAPP board while the other parts of the algorithm run on the PC. However, the interface between the PC and the GAPP array is menu-driven and does not perform real-time data transfer. The GAPP PC Development System has been designed for developing and debugging parallel programs rather than for number-crunching purposes.

The actual code is approximately 50 lines written in GAL. Triangularization of a matrix of size 56×39 requires 3211 GAPP instructions, where 1 GAPP instruction takes 100 ns (the clock rate is 10 MHz). Since the data communication between the PC and the GAPP array is *menu-driven,* the total user-time for factoring an integer with 10 digits approaches 7 s.

5. CONCLUSIONS

We have presented a fast algorithm for the implementation of the Gaussian elimination over $GF(2)$ on massively parallel bit-arrays, and described a particular implementation on the GAPP. Since the current GAPP chip has 72 processors per chip and only 128 bits of memory per processor, the size of the problems to be solved on the GAPP is limited. Several GAPP chips can be put together to form a larger array in order to work with larger matrices. Construction of a special-purpose computing device for factorization of large integers was suggested by Poet [14]. A special-purpose massively parallel bit-array designed using GAPP-like chips, e.g., BLITZEN [3], coupled with a high-speed scientific workstation and high-throughput I/O channels could provide a cost-effective alternative to using the vector processors or the general-purpose massively parallel processors for the implementation of the continued fraction method to factor large integers.

ACKNOWLEDGMENTS

The authors thank the anonymous referees for pointing out Refs. [4, 6] and making suggestions which improved the presentation.

REFERENCES

1. Ahmed, H. M., Delome, J. M., and Morf, M. Highly concurrent computing structures for matrix arithmetic and signal processing. *IEEE Comput.* **15** (1982), 65–82.
2. Barada, H., and El-Amawy, A. Systolic architecture for matrix triangularisation with partial pivoting. *IEE Proc. Part E: Comput. Digital Techniques* **135,** 4 (July 1988), 208–213.
3. Blevins, D. W., Davis, E. W., Heaton, R. A., and Reif, J. H. BLITZEN: A highly integrated massively parallel machine. *J. Parallel Distrib. Comput.* **8** (1990), 150–160.
4. Cosnard, M., Duprat, J., and Robert, Y. Systolic triangularization over finite fields. *J. Parallel Distrib. Comput.* **9** (1990), 252–260.
5. Davis, R., and Thomas, D. Systolic array chip matches the pace of high-speed processing. *Electronic Design* **32,** 19 (Sept. 1984), 207–218.
6. Falkoff, A. D. Algorithms for parallel-search memories. *J. ACM* **9,** 4 (1962), 488–511.
7. Gentleman, W. M., and Kung, H. T. Matrix triangularisation by systolic arrays. *Proc. SPIE 298, Real-Time Signal Processing IV,* San Diego, CA, 1981, pp. 19–26.
8. Hochet, B., Quinton, P., and Robert, Y. Systolic Gaussian elimination over $GF(p)$ with partial pivoting. *IEEE Trans. Comput* **38,** 9 (Sept 1989), 1321–1324.
9. Lehmer, D. H., and Powers, R. E. On factoring large numbers. *Bull. Amer. Math. Soc.* **37** (1931), 770–776.
10. Morrison, M. A., and Brillhart, J. A method of factoring and the factorization of F_7. *Math. of Computat.* **29** (1975), 183–205.
11. NCR Corporation, Dayton, OH. GAPP PC development system, NCR45GDS1, 1985.
12. NCR Corporation, Dayton, OH. GAPP: Geometric arithmetic parallel processor, NCR45CG72, 1987.
13. Parkinson, D., and Wunderlich, W. A compact algorithm for Gaussian elimination over $GF(2)$ implemented on highly parallel computers. *Parallel Comput.* **1,** 1 (Aug. 1984), 65–73.
14. Poet, P. The design of special purpose hardware to factor large integers. *Comput. Phys. Commun.* **37,** 1–3 (July 1985), 337–341.
15. Reddaway, S. F. DAP—A distributed array processor. *Proc. 1st Symposium on Computer Architecture,* 1973, pp. 61–65.
16. Takefuji, Y., Kurokawa, T., and Aiso, H. Fast matrix solver in $GF(2)$. *Proc. 6th Symposium on Computer Arithmetic.* IEEE Computer Society Press, Aarhus, Denmark, June 20–22, 1983, pp. 138–143.
17. Wunderlich, M. C. Factoring numers on the massively parallel processor. In Chaum, D. (Ed.). *Advances in Cryptology, Proceedings of Crypto 83.* Plenum, New York, 1984, pp. 87–102.
18. Wunderlich, M. C. Implementing the continued fraction factoring algorithms on parallel machines. *Math. Computat.* **44,** 169 (Jan. 1985), 251–260.

Received March 29, 1990; accepted March 12, 1991