REALIZATION OF EXTENSIONS TO FADDEEV ALGORITHM ON ARRAY OF SIMD PROCESSORS.

Hai Van Dinh Le
Marek A. Perkowski

Department of Electrical Engineering,
Portland State University,
Portland, Oregon 97207,
tel. (503) 464-3806 x 23.

ABSTRACT

The paper presents three types of extensions: vertical, horizontal, and two-dimensional to the new, Faddeev algorithm based, systolic architecture for matrix computations, presented in: *H.V.D. Le, M.A. Perkowski, "A New General Purpose Systolic Architecture for Matrix Computations". Proc. Intern. Conf. on Computing and Information, ICCI'89, Ontario, Canada, 1989.* It has essential advantages over previous architectures of this type and finds various applications: extensions to Faddeev algorithm can be used in many problems, including Karmarkar algorithm. The extensions described in this paper not only increase a system throughput from two to four fold but also enhance the inherent programmability of Faddeev's algorithm. This allows our architecture to perform very complex matrix calculations.

1. INTRODUCTION.

It has been conservatively estimated that 75 percent of all scientific applications involve some form of **matrix computations.** In general, matrix computations are very expensive in term of processing time. For real time operation, required by such applications as robotics, signal processing, computer vision, computer graphics animation, simulation and modeling, Computer Aided design, and many other, the processing power of serial computers is simply inadequate. Other applications, that are of special interest to our research group are related to designing specialized co-processor architectures [22,8] for solving NP-complete, combinatorial and other problems related to high-level and logic level synthesis of digital circuits (also multi-valued). They include extremely fast solving of linear programming problems (also using the Karmarkar algorithm that is particularly well suited for architecture presented in this paper), as well as spectral transforms and other modern approaches to logic synthesis. All these problems require fast matrix computations composed of various sequences of inversion, transposition, multiplication, and addition.

In [18] a new systolic architecture for Faddev algorithm was proposed, that has essential advantages over the well known architectures of this type [2,20,21]. Because Faddeev's algorithm is *inherently general purpose,* our architecture is able to perform a wide class of matrix computations. And since the architecture is *systolic based,* it brings *massive parallelism* to all of its computations. As a result, many matrix operations including *addition, multiplication, inversion, LU-decomposition, transpose, solutions to linear systems of equations, convolution,* 2D-polynomial operations, and other, can be now performed extremely fast. This architecture has several other advantages as well: *better performance for smaller cost, easy reconfigurability, size independence - problem decomposition, processing of sparse arrays, and increased throughput* [15-19]. Our design can be *re-configured* during run time to perform different functions with the uses of various control signals that are propagated throughout the arrays. It allows also for *maximum overlaps* of processing between consecutive computations, thereby increasing system throughput.

In this paper only the new extensions are described, the reader is assumed to be familiar with [18].

2. FADDEEV'S ALGORITHM AND MATRIX TRIANGULARIZATION

Faddeev's algorithm [4,19,20,15] is a *general purpose algorithm,* useful for a wide class of matrix operations and especially suited for systolic implementation. It calculates $C A^{-1} B + D$ from

$$\begin{bmatrix} A & B \\ -C & D \end{bmatrix} \tag{2.1}$$

Since the underlying procedure to carry out Faddeev's algorithm is *matrix triangularization,* any systolic implementation of the algorithm should be based on a structure which can perform triangularization efficiently. The triangular systolic array developed by Gentleman and Kung as a common platform for two different triangularization methods which is shown in Fig. 2.1 can execute both *Gaussian elimination with neighbor pivoting* or *orthogonal triangularization* [6,11]. Triangular systolic array for matrix triangularization.

The array consists of two types of cells: the *boundary cells* (represented by circles) and the *internal cells* (represented by squares). These cells are locally interconnected into a triangular mesh. Each cell stores a *microprogram,* enabling it to interact with its neighbors in such a way that a triangularization procedure can be carried out. Changing the *microprograms of the cells* will allow the array to execute different procedures. In the following discussion, the term *data row* refers to a row of entries of matrix X where the term array row means a row of cells of the array.

The triangular array of Fig. 2.1 can perform Gaussian elimination with neighbor pivoting using the cells shown in Fig. 2.2. As its microcode reveals, the boundary cell generates two modification factors: a multiplier $\mathbf{M}_{out}$ as well as a Boolean variable $\mathbf{V}_{out}$ which signals a row interchange when it has a value of one. This occurs at every *array cycle,* the maximum length of time necessary for a cell to execute its microprogram once. The array can perform orthogonal triangularization using the cells specified in Fig. 2.3.

3. THE DUAL MODE ARCHITECTURE

The new systolic implementation of Faddeev's algorithm [18] is based on *dual mode array,* which, in its basic form, reduces the I/O bandwidth requirement by half and the number of cells needed by more than one third, comparing to the architectures from [2,20,21]. It consists of a **square array** in which the cells are orthogonally connected (Fig. 3.1). Data bus interconnections between cells are indicated by arrows. Functionally, there are two types of cells. The first type consists of all the *diagonal cells* (denoted by circles) of the array and the second type of all the *non-diagonal cells* (denoted by squares). Depending on the actual processing phase, the array functions in one of the two **modes** : the *T (triangular) mode* or the *S (square) mode.* Together, these two modes implement Faddeev's algorithm to compute $\mathbf{C}\ \mathbf{A}^{-1}\mathbf{B} + \mathbf{D}$ from (2.1). When the array is in T mode, cells of rows **i** where **i = 1, 2, .., w** and columns **j** where **j ≥ i**, form a triangular sub-array which is based on Gentlemen and Kung's array of Fig. 2.1. It performs *Gaussian elimination with neighbor pivoting on* **A**, and ordinary Gaussian elimination on **C**. During this mode of operation, the circular and square cells essentially carry out the same functions specified by Fig. 2.2 boundary and internal cells, respectively. When in S mode, the entire array is used to process **B** and **D**. In this mode, every cell of the array acts similarly to the internal cell of Fig. 2.2, i.e. circular cells functionally become square cells. In order to switch the array from one mode to another, it is only necessary to change the program of the diagonal cells. This is accomplished using the cell microprograms listed in Fig. 3.2.

The careful reader may observe [15] that by alternating between the two operational modes T and S, our array essentially simulates the one-T and one-S system of Chuang and He [2] to solve (2.1) with a *smaller number of cells and half the bandwidth requirement.* Naturally, the input data flow will have to be slightly modified because of the differences in the array's topology.

The circular and square cells, as shown in Fig. 3.2, have identical I/O and control bandwidth: two **n** -bit data input ports, two **n** -bit data output ports, four one-bit control input ports and four one-bit control output ports, for a total bandwidth of **4n+8.** In fact, this number is comparable to the actual pin count that the Chuang and He internal cell would need, since their cell does require extra control capabilities to work properly.

Control Signals Interconnections.

As shown in Fig. 3.1, the circular cell relies on three external control signals C1, C2, and C4 for internal computation and itself generates signal C3, all of which it broadcasts locally to its neighbors for correct operation of the entire array. The square cell uses only C3 and C4, and passes all control signals it receives to neighboring cells unchanged. C1, C2, C3, and C4 are all one-bit boolean values whose functions and interconnection patterns are described below. C1 controls the behavior of diagonal cells and consequently selects the *operation mode* of the array. When C1 is *true,* the diagonal cells execute the portion of their code that enables them to function like Kung's boundary cells from Fig. 2.1, thus changing the array into T mode. Otherwise, with C1 *false,* diagonal cells function like square cells, and the array is in S mode. Because of the strict timing required, mode switching should occur as entries of the first row of **B** reach each cell, i.e. the switching sweeps across the array in skewed waves as the transition between C and B flows through the cells. This can be accomplished without the need to address separate control signals to each individual diagonal cell. In fact, C1 needs to be fed only to the top left diagonal cell of the array and will be pipelined through the array to reach every diagonal cell. As the data flow changes from matrix A to matrix C, T mode processing in the array gradually switches from *Gaussian elimination with pivoting* to *non-pivoting Gaussian elimination.* This event is started with C2, whose value is *true* for **pivoting allowed** and *false* for **pivoting not allowed.** Again, C2 is fed only to the top left diagonal cell and propagated through the array. Generated internally by diagonal cells when they are in T mode, C3 is the functional equivalent of $\mathbf{M}_{out}$ of the boundary cell from Fig. 3.2. It is thus used to direct square cells on the same row to pivot incoming data when *true,* or not to pivot when *false.* When switching between the T and S modes of operation, it is essential that the **X** registers in each and every cell of the array are cleared to zero **before** the new data elements arrive. If C4 is *true,* a cell will clear its **X** register prior to receiving $\mathbf{X}_{in}$ from its northern neighbor. The **X** register remains unchanged if C4 is *false.* C4 is distributed throughout the array.

Data Flow Description

Again suppose that **A, B, C** and **D** of (2.1) are **n × n** matrices and the available bandwidth is **w = n.** The input data flow, of width **n** and length **4n,** will be continuous and consists of matrices **A, C, B** and **D,** in that order, skewed as shown in Fig. 3.1. Note that the control signals necessary for each step are displayed alongside the data flow.

Processing will be as follows. Initially, **A** enters the array followed by **C** ; because C4 is *true* (for the duration of one cycle), all cells will clear their **X** register of values left from any previous calculations. With C1 and C2 both *true,* cells of the upper triangle begin performing *Gaussian elimination* (with *neighbor pivoting*) to *triangularize* **A** as its data elements are upon them. As C1 reaches each diagonal cell, the array gradually switches to T mode.

When entry $\mathbf{c}_{11}$ of matrix **C** arrives at the top left cell, C2 becomes *false* which disables neighbor pivoting in the diagonal cells. Thus, only the ordinary Gaussian elimination is performed to annul **C.** Throughout this period, C1 remains true, hence the array remains in T mode.

Subsequently, as **B** reaches the array, C4 goes *true* again for the duration of one cycle (step), long enough for the top left cell to store this value; the signal is then propagated to all cells and clears their **X** registers. At the same time, C1 becomes *false* and remains so until the last row of **D** is in the array. As C1 reaches each diagonal cell, it turns it into a square cell and thus gradually changes the array to S mode as the data elements of **B** are pipelined through the array. The results, shown in Fig. 3.1, fully emerge from the bottom of the array after **6n - 1** steps for $\mathbf{C}\ \mathbf{A}^{-1}\mathbf{B} + \mathbf{D}$ and **5n** steps for the solution to a linear system.

Storage and Feedback of Modification Factors.

During the processing of matrices **A** and **C,** *modification factors* $\mathbf{M}_{out}$ and *pivoting control bits* C3 are generated by diagonal cells based on incoming values $\mathbf{X}_{in}$. They are then sent to the right to the square cells on the same row to modify adjacent $\mathbf{X}_{in}$ values. As it reaches the edge of the array, this data stream to the right is stored in $\mathbf{B}_q$, a FIFO queue of size w × w shown in Fig. 3.1. This queue acts as a delay mechanism that will recirculate its contents to the left side of the array for the processing of **B** and **D** as they arrive at the array.

To reduce demands on available bandwidth between the host and the array, B_q should not be implemented using the conventional memory of the host. Instead, the queue should be a *dedicated buffer* made up entirely of shift registers and run at the same clock rate as the array. This represents the most efficient way to implement the horizontal feedback path.

4. MULTIPLE ARRAY CONFIGURATION

By using multiple arrays, the system of Fig. 4.1 gives better throughput than the single array under the same I/O constraint. This is because each subarray effectively replaces one iteration, with partial results from one subarray immediately processed by the next, thereby maximizing concurrency while eliminating the corresponding iteration. Such a system will be called **L - tuple** arrays system (**L** = 2 in Fig. 4.1), or **L-subarrays system.** Detailed analysis of the Multiple Array Configuration and solving size independent problems is given in *[15]*.

5. EXTENSIONS TO FADDEEV'S ALGORITHM

In the previous sections, the reader has seen the ease with which the new systolic array uses massive parallelism to solve many types of matrix problems via Faddeev's algorithm. The actual size of the array, and therefore its throughput, is shown to be restricted only by the available bandwidth between the host and the array. Even this restriction is effectively circumvented when a number of such arrays are combined into a system to give a desired level of performance. Such a multiple arrays system reaches its maximum throughput rate when its pipeline is completely filled with data. By ensuring that the input data flow is continous, this maximum throughput rate is maintained at all times. It would seem then, algorithmically speaking, that nothing further can be done to induce more parallelism into matrix computations. However, that last observation is simply not true. We have found that, by extending Faddeev's algorithm, the maximum throughput rate of a system can be nearly quadrupled. Furthermore, such a tremendous improvement in system throughput requires absolutely no architectural modification to the system.

5.1. HORIZONTAL EXTENSION TO FADDEEV'S ALGORITHM

Before illustrating how we extend Faddeev's algorithm, let us introduce the concept of compatibility between matrix problems. Suppose we have matrices **A, B** and **D** of order *n*, upon which we wish to perform the operations A^{-1}, $A^{-1}B$, and $A^{-1}+D$. From basic properties of Faddeev algorithm, we can solve these matrix problems with this algorithm by formulating them as:

$$\underset{(1)}{\begin{bmatrix} A & I \\ -I & 0 \end{bmatrix} \Rightarrow A^{-1}} \qquad \underset{(2)}{\begin{bmatrix} A & B \\ -I & 0 \end{bmatrix} \Rightarrow A^{-1}B} \qquad \underset{(3)}{\begin{bmatrix} A & I \\ -I & D \end{bmatrix} \Rightarrow A^{-1}+D} \tag{5.1}$$

where **I** is the identity matrix. These constructs reveals that they all have identical left halves, i.e. they consist of the same matrix **A** in their top left quadrant and the same matrix **-I** in their bottom left quadrant. When this is the case, we say that the problems are *horizontally compatible*.

Obviously, solving **x** horizontally compatible problems involves repeating the calculations for the same left side **x** number of times. In the case of (5.1) where **x = 3**, solving (1), (2) and (3) requires repeating the process of triangularizing **A** and annulling **-I** three times. If by some means the redundant iterations of this process are eliminated, nearly half of the calculations necessary to solve (2) and (3) of (5.1) can be skipped. This would yield a large savings in the computing time. To accomplish this, we extend Faddeev's algorithm horizontally to the right so that (5.1) is reformulated as

$$\begin{bmatrix} A & I & B & I \\ -I & 0 & 0 & D \end{bmatrix} \atop \quad\;\;(1)\,(2)\,(3) \tag{5.2}$$

Grouping (1), (2) and (3) together as in (5.2) allows us to triangularize **A** and annul **-I** only once, and reuse the multipliers generated from that several times on the right. The results will appear as:

$$\begin{bmatrix} A^{(k)} & I^{(k)} & B^{(k)} & I^{(k)} \\ 0 & A^{-1} & A^{-1}B & A^{-1}+D \end{bmatrix} \atop \qquad (1) \quad (2) \quad (3)$$

It is easy to see that the horizontal extension to Faddeev's algorithm maps particularly well to a system using our systolic array design: *it requires absolutely no architectural nor algorithmic modification*, either at the system level, subarray level or cell level. When the available I/O bandwidth is *w*, (5.2) is parallely decomposed into (x + 1)m input strips, each **2mw** in length, as shown in Fig. 5.1.

As before, the L-subarrays system of Fig. 4.1 will process this input data flow in **k** iterations, where the value of **k** depends on **m** and **L.** When **m** is an exact multiple of **L,** we have **k = m/L** and the system will compute **x** horizontally compatible problems in

$$(L+1)w - 1 + \sum_{k=1}^{m/L} [(x+1)m - (k-1)L]\,[2m-(k-1)L]w \tag{5.3}$$

cycles. In the above equation, the first product term of the summation represents the number of input strips for each iteration, while the second term indicates the strips length. The solution to the first problem will come out after

$$(L+1)w - 1 + \sum_{k=1}^{(m/L)-1} [(x+1)m - (k-1)L]\,[2m-(k-1)L]w + (m+L)^2 w$$

cycles, with the second line of the equation indicating that only part of the **k** *th* iteration is needed. Afterward, solutions to subsequent **(x - 1)** problems are transmitted to the output one for every **(m + L)n** cycles. In the special case when **m = L,** we have **k = 1** and the system will solve the first problem in **(4m + 1)n + w - 1** cycles. As to subsequent problems, the system will complete one every **2mn** cycles. The difference between the two throughput equations of the first problem is due to the fact that the input data flow for **x** horizontally compatible problems consist of **(x - 1)m** more strips than that of a single problem. This means that during each iteration, the system has that many more strips to process. Thus when **k > 1,** the previous iterations will delay the output of results whereas with **k = 1,** those delays are non-existent.

When **m** is not an exact multiple of **L,** the number of iterations required for the system to process (5.2) is **k** = ⌈ **m/L** ⌉ , *with the k^{th} iteration involving only the first $m_{mod\,L}$ subarrays* of the system. The total throughput will be

$$(m_{mod\,L}+1)w - 1 + \sum_{k=1}^{\lceil m/L \rceil} [(x+1)m - (k-1)L]\,[2m-(k-1)L]w \tag{5.4}$$

with solution to the first problem coming out after

$$(m_{mod\,L}+1)w - 1 + \sum_{k=1}^{\lceil m/L \rceil} [(x+1)m - (k-1)L]\,[2m-(k-1)L]w + (m+m_{mod\,L})^2 w$$

cycles. Again, the second line of the above equation indicates that only part of the last iteration is needed by the system to compute the first problem. Afterward, solutions to subsequent x - 1 problems will emerge one for every $(\mathbf{m} + \mathbf{m}_{\mathbf{mod\ L}})\mathbf{n}$ cycles.

Since the input data flow of **x** horizontally compatible problems consists of only **(x + 1)m** strips, versus the **2xm** strips required if they are not compatible, large saving in storage space can be gained on the host side. On the other hand, the length of the FIFO buffer B_r should be **((x + 1)m - L) (2m - L)w - Lw** since the intermediate results after the first iteration have many more strips. Because the length of each strip is still **2mw**, the capacity of the buffers B_q should remain unchanged.

To get an idea of how much the system throughput can be improved when horizontal extension is applied, suppose that we have a system of **L** = 4 subarrays, with each array of size **w** = 32. On this system, we wish to perform **x** = 50 operations with matrices of order **n** = 128. If these operations are not compatible, solving them one at a time without processing overlaps will take a total of 110,350 steps. With processing overlaps, this number is reduced to 102,559. However, if the operations are horizontally compatible, they can be processed by the system in 52,383 steps. The improvement in throughput is

$$\frac{102{,}559}{52{,}383} = 1.96,$$

nearly by a factor of two. Of course, this number can vary depending on **x.** As **x** gets larger, the improvement factor gets closer to two.

5.2. VERTICAL EXTENSION TO FADDEEV'S ALGORITHM

Even when a group of matrix problems are not horizontally compatible, they may exhibit another type of compatibility which can also be exploited to give an equivalent speedup in system throughput. To expand on this, let's suppose that we have **y** = **3** matrix operations to perform, namely **CB, B + D** and **EB + D** where **B, C, D** and **E** are of order **n.** Like before, we can express these problems as

$$\underset{(1)}{\begin{bmatrix} I & B \\ -C & 0 \end{bmatrix} \Rightarrow CB} \qquad \underset{(2)}{\begin{bmatrix} I & B \\ -I & D \end{bmatrix} \Rightarrow B+D} \qquad \underset{(3)}{\begin{bmatrix} I & B \\ -E & D \end{bmatrix} \Rightarrow EB+D} \tag{5.5}$$

Because the left side of problems (1), (2) and (3) of (5.5) are not the same, they are not horizontally compatible. However, it can be observed that they all have the identity matrix **I** in their top left quadrant and matrix **B** in their top right quadrant. To put it differently, these problems all have identical top half. When this is the case, we say that the problems are *vertically compatible.*

To avoid repeating the same calculations on the identical top sides of vertically compatible problems, *we extend Faddeev's vertically such that (5.5) becomes*

$$\begin{array}{l} \begin{bmatrix} I & B \\ -C & 0 \\ -I & D \\ -E & D \end{bmatrix} \begin{array}{l} \\ (1) \\ (2) \\ (3) \end{array} \end{array} \tag{5.6}$$

When **y** vertically compatible problems are grouped together as in (5.6), the common top *side needs to be processed only once. This means that after the top left quadrant is triangularized* and the top right quadrant is modified with the generated multipliers, they can be used repeatedly to annul the left side of succeeding stages and transform their right side into solutions.

In the case of (5.6), solving it involves only the annulment **-C, -I** and **-E.** This is because the identity matrix **I** in the top left quadrant is, by its nature, already triangularized; as a consequence, matrix **B** in the top row will remain unmodified. Annulling **-C, -I** and **-E** while extending the operations to the right will give

$$\begin{bmatrix} I & B \\ 0 & CB \\ 0 & B+D \\ 0 & EB+D \end{bmatrix} \begin{array}{l} \\ (1) \\ (2) \\ (3) \end{array}$$

which shows the solutions to (1), (2) and (3) in the right quadrants.

As with horizontal extension, systems using our array design can handle vertical extension to Faddeev's algorithm without any modification.

Shown in Fig. 5.2, the input data flow of **y** vertically compatible problems consists of **2m** strips, where each strip is **(y + 1)m** blocks long. The L-subarrays system of Fig. 4.1 will process this data flow in **k** iterations. When **m** is an exact multiple of **L, k = m/L** and the process will be completed in

$$(L+1)w - 1 + \sum_{k=1}^{m/L} [2m-(k-1)L]\,[(y+1)m - (k-1)L]w \tag{5.7}$$

cycles. When **m** is not an exact multiple of **L, k** = ⌈ **m/L** ⌉ and the throughput is computed as

$$(m_{mod\,L}+1)w - 1 + \sum_{k=1}^{\lceil m/L \rceil} [2m-(k-1)L]\,[(y+1)m - (k-1)L]w \tag{5.8}$$

In throughput equations (5.7) and (5.8), the first product term within the summation represents the number of input strips for each iteration. The length of each strip, on the other hand, is indicated by the second product term. Even so, note that (5.7) and (5.8) are identical to (5.3) and (5.4), respectively, save for the variables **x** and **y.** After the **k** *th* iteration, the set of **y** solutions emerges in **m** output strips. As shown in Fig. 5.2, an output strip consists of **y** segments, each of width **w** and length **mw.** Each segment **i = 1, 2,..., y** is part of the solution to the i th problem. Because a solution is divided into **m** segments with each segment part of an output strip, the solutions will not be completely out until the last strip has emerged. Thus, the number of steps needed for the first solution to come out is computed by subtracting **(y - 1)mw** from (5.7) or (5.8). Each following solutions takes another **mw** steps.

Again, storage space needed on the host side is greatly reduced since the input data flow of **y** vertically compatible problems is only **2(y + 1)m^2w** long, as opposed to 4**y m ^{2}w** were they not compatible. However, the length of the FIFO buffer **B $_q$** should be ((**y** + 1)**m** - 1)**w** to accommodate longer strips of modification factors. In addition, the length of **B $_r$** should be **(2m - L)((x + 1)m - L)w - Lw** to adequately hold intermediate results with longer strips.

5.3. TWO-DIMENSIONAL EXTENSION TO FADDEEV'S ALGORITHM

While using either one of the previously described extensions yields substantial reduction in computing time, still greater improvement in throughput is possible when both techniques are combined into a two-dimensional extension to Faddeev's algorithm. To illustrate, consider the matrix operations **AB, AE + F, B + D and E + G.** As before, **A, B, D, E, F** and **G** are all matrices of order n. Formulating the operations as follow:

$$\underset{(1)}{\begin{bmatrix} I & B \\ -A & 0 \end{bmatrix} \Rightarrow AB} \quad \underset{(2)}{\begin{bmatrix} I & E \\ -A & F \end{bmatrix} \Rightarrow AE+F} \quad \underset{(3)}{\begin{bmatrix} I\,B \\ -I\,D \end{bmatrix} \Rightarrow B+D} \quad \underset{(4)}{\begin{bmatrix} I\,E \\ -I\,G \end{bmatrix} \Rightarrow E+G} \tag{5.9}$$

reveals that (1) and (2) are horizontally compatible, as with (3) ana (4). Furthermore, (5.9) also shows that (1) and (3) are vertically compatible, as with (2) and (4). Thus, using horizontal extension, (5.9) becomes

$$\underset{(1)\,(2)}{\begin{bmatrix} I & B & E \\ -A & 0 & F \end{bmatrix}} \qquad \underset{(3)\,(4)}{\begin{bmatrix} I & B & E \\ -I & D & G \end{bmatrix}} \tag{5.10}$$

Since both constructs of (5.10) have identical top halves, vertical extension can also be used to further obtain:

$$\begin{bmatrix} I & B & E \\ -A & 0 & F \\ -I & D & G \end{bmatrix} \begin{matrix} \\ (1)\ and\ (2) \\ (3)\ and\ (4) \end{matrix} \tag{5.11}$$

This results in a two-dimensional extension to Faddeev's algorithm. Annulling **-A** and **-I** in (5.11) and extending the operations to its right prompt the solutions to (1), (2), (3) and (4) to appear as

$$\begin{bmatrix} I & B & E \\ 0 & AB & AE+F \\ 0 & B+D & E+G \end{bmatrix} \begin{matrix} \\ (1)\ and\ (2) \\ (3)\ and\ (4) \end{matrix}$$

As (5.11) reveals, the two-dimensional extension to Faddeev's algorithm allows a compatible matrix problem to share three of its quadrants with others, instead of two. This translates into the elimination of a larger number of calculations per problem. The input data flow of (5.11) for the L-subarrays system is shown in Fig. 5.3. When the number of problems is **x** across by **y** long, the input data flow is decomposed into **(x + 1)m** parallel strips, each (**y** + 1)**mw** in length. If **m** is an exact multiple of **L,** the total number of steps for the L-subarrays system of Fig. 4.1 to process this data flow is

$$(L+1)w - 1 + \sum_{k=1}^{m/L} [(x+1)m-(k-1)L]\,[(y+1)m-(k-1)L]w \tag{5.12}$$

If m is not an exact multiple of **L,** then the number of steps needed is computed as

$$(m_{mod\,L}+1)w - 1 + \sum_{k=1}^{|m/L|} [(x+1)m-(k-1)L]\,[(y+1)m-(k-1)L]w \tag{5.13}$$

Subtracting **[(x - 1)(ym + L) + (y - 1)]mw** from (5.12) or **[(x - 1)(ym + m $_{mod\ L}$) + (y − 1)]mw** from (5.13) will, in both cases, give the number of steps elapsed before the solution to the first problem is completely out. The interval between solutions to problems on the same column is mw steps. Between problems on the same row, this interval is computed as **(ym + L)mw** when **m $_{mod\ L}$ = 0, or (ym + m $_{mod\ L}$)mw** when **m $_{mod\ L}$ ≠ 0.** Because of the increases in number of strips and in their length, the capacity of buffers **B $_q$** and **B $_r$** should be expanded as previously indicated. To see how much of an improvement over single dimension extensions this technique is capable of, let us again assume that we have a system of **L** = 4 subarrays, with each array of size **w** = 32. With this system, 10000 operations are to be performed on a number of matrices of order **n** = 128. Solving the problems one at a time without processing overlaps will take a total of 22,070,000 steps. Maximizing processing overlaps will reduce this number to 20,480,159. If single dimension extensions can be used, the problems can be solved in 10,241,183 steps. The improvement in throughput is

$$\frac{20{,}480{,}159}{10{,}241{,}183} = 2.0$$

However, if compatibilities between these problems are exploited such that the two-dimensional extension can be used with **x** = 100 and **y** = 100, the total throughput will be 5,223,071 steps. The improvement factor is thus

$$\frac{20{,}480{,}159}{5{,}223{,}071} = 3.92,$$

almost doubling the speedup figure achieved with single dimension extension. As was noted before, the improvement factor grows closer to four as **x** and **y** get larger. Another advantage of the two-dimensional extension is that it further enhances the inherent programmability of Faddeev's algorithm. For example, should it be necessary to compute U, where

$$\mathbf{U} = (\mathbf{A}\,\mathbf{E} + \mathbf{F})(\mathbf{E} + \mathbf{G})^{-1}(\mathbf{B} + \mathbf{D}) + \mathbf{A}\,\mathbf{B}, \tag{5.14}$$

then the matrix from (5.11) can be rearranged to become

$$\begin{bmatrix} I & E & B \\ -I & G & D \\ -A & F & 0 \end{bmatrix} \tag{5.15}$$

Solving (5.15), that is annulling **-I** and **-A** while extending the operations to the right will give

$$\begin{bmatrix} I & E & B \\ 0 & E+G & B+D \\ 0 & AE+F & AB \end{bmatrix} \tag{5.16}$$

Observe that within the box of (5.16), the necessary components of (5.14) are already correctly positioned such that repeating the Faddeev's procedure on them will produce the final result

$$\begin{bmatrix} I & E & B \\ 0 & (E+G)^{(k)} & (B+D)^{(k)} \\ 0 & 0 & U \end{bmatrix} \tag{5.17}$$

In short, to compute **U** from matrix (5.15), one only needs to triangularize the augmented matrix formed from **I, E, -I** and **G,** then annul the augmented matrix formed from **-A** and **F** while extending both operations to the rightmost column of (5.15). Using the L-subarrays system, **U** is computed from the input data flow of (5.15) in **2k** iterations. The first **k** iterations are needed to compute the matrices in the box of (5.16). This intermediate results is immediately fed back into the system for another **k** iterations, after which **U** is forwarded to the output.

6. CONCLUSION

By now, it is clearly obvious that the symbiosis of Faddeev's algorithm and the new systolic array system described here has given rise to a very powerful and versatile tool. The algorithm itself provides a considerable generality of operation which should allow the system to have a large range of application in the scientific and industrial fields. In return, the system has brought massive parallelism to the multitude of matrix operations capable by the algorithm. Furthermore, the system's enormous potential for parallelism can now be fully exploited to yield very high throughput with the Faddeev's algorithm extensions. As compared to other designs from the literature, this system does not suffer any of their drawbacks while providing many practical advantages, some of which can be summarized as follows:

- The system provides identical performance using a smaller number of cells or arrays. Indeed, given an equal number of arrays, its performance will be superior. When taken into account the fact that its design is ideally suited for the extensions made to Faddeev's algorithm, its throughput potential far outdistances any other systems previously considered.
- From a user point of view, operating the system is exceedingly simple: the input data flow is fed only to the top array and system controls consist of a few signals to each array top left cell.
- The design of the system is truly modular, with simple and regular interconnections between cells and between modules. Hence it is very amenable to expansion: adding extra blocks of shift registers will allow it to handle correspondingly larger problems, while increasing the number of arrays will yield higher throughput.
- Since all modules are square blocks w * w in size, it is topologically more economical and efficient in terms of PC board area. In conclusion, the system most important advantage is that while its design is simple enough for implementation to be an easy task, it is abundantly powerful and versatile to make that task worthwhile.

7. LITERATURE

[1] R. L. Burden et al, Numerical Analysis, *PWS Publishers,* Boston, MA, 1981, pp. 289-294. [2] H. Y. H. Chuang and G. He, "A Versatile Systolic Array For Matrix Computations," *The International Symposium on Computer Architecture,* 1985, pp. 315-322. [3] P. M. Dew, "VLSI Architectures for Problems in Numerical Computation," in *Supercomputers and Parallel Computation,* ed. by D. J. Paddon, Oxford University Press, New York, 1984, pp. 2-21. [4] D. K. Faddeev and V. N. Faddeeva, Computational Methods of Linear Algebra, *W. H. Freeman and Company,* 1963, pp. 150-158. [5] W. M. Gentleman, "Error Analysis of QR Decompositions by Givens Transformations," in "Linear Algebra and Its Applications", *American Elsevier Publishing Company,* New York, 1975, pp. 189-197 [6] W. W. Gentleman and H. T. Kung, "Matrix Triangularization by Systolic Arrays," *Proc. SPIE,* The International Society of Optical Engineering, Vol. 298, 1981, pp. 19-26. [7] W. Handler, "Innovative Computer Architecture - How to Increase Parallelism but Not Complexity," in *Parallel Processing Systems,* ed. by David J. Evans, Cambridge University Press, Cambridge, MA, 1982, pp.23-32. [8] P. M. Ho, and M. A. Perkowski, "Systolic Architecture for Solving NP-hard Combinatorial Problems of Logic Design and Related Areas", *Proc. of ISCAS'89, IEEE International Symposium on Circuits and Systems,* May 9-12, Portland, Oregon 1989. [9] K. Hwang and F. A. Briggs, Computer Architecture and Parallel Processing, *McGraw-Hill,* New York, 1984, pp. 768-774. [10] H. T. Kung and C. E. Leiserson, "Systolic Arrays (for VLSI)," *Sparse Matrix Proc.,* 1978, Society for Industrial and Applied Mathematics, 1979, pp. 256-282. [11] H. T. Kung, "Systolic Array for Orthogonal Triangularization," *Proc. SPIE,* San Diego, CA, 1981, pp. 19-26. [12] H. T. Kung, "Why Systolic Architectures?", *IEEE Computer Magazine,* Vol. 15, No. 1, January 1982, pp. 37-46. [13] H. T. Kung, "Notes on VLSI Computation," in *Parallel Processing Systems,* ed. by David J. Evans, Cambridge University Press, Cambridge, MA, 1982, pp.339-356. [14] S. Y. Kung, "VLSI Array Processors," *IEEE ASSP Magazine,* Vol. 2, No. 3, July 1985, pp. 4-22. [15] H. V. D. Le, A New General Purpose Systolic Array for Matrix Computations, *M. Sc. Thesis,* Department of Electrical Engineering, Portland State University, 1988. [16] H. V. D. Le, and M. A. Perkowski, "New General Purpose Systolic Array Architecture for Extended Faddeev Algorithm for Matrix Computations", *Submitted to IEEE Transactions on Computers.* [17] H. V. D. Le, and M. A. Perkowski, "Real Time Graphical Simulation of Systolic Arrays", *Proceedings of IEEE ISCAS'89, International Symposium on Circuits and Systems,* May 9-12, 1989, Portland, Oregon. [18] H. V. D. Le, and M. A. Perkowski, "A New General Purpose Systolic Architecture for Matrix Computations". *Proc. International Conference on Computing and Information,* ICCI'89, Ontario, Canada, 1989. [19] H. V. D. Le, and M. A. Perkowski, "Size Independent Implementation of Matrix Operations on TASA - A Two-Dimesional Array Matrix Architecture", *Proc. of Intern. Phoenix Conf. on Computers and* Comm., IPCCC-90, Phoenix, Arizona, March 1990. [20] J. G. Nash and S. Hansen, "Modified Faddeev Algorithm for Matrix Manipulation," *Proc. SPIE,* Vol. 495, August 1984, pp. 39-46. [21] J. G. Nash, "A Systolic/Cellular Computer Architecture for Linear Algebraic Operations," *Proc. 1985 IEEE International Conference on Robotics and Automation,* March 1985, pp. 779-784. [22] M. A. Perkowski, "Systolic Architecture for the Logic Design Machine", *Proc. 1985 IEEE International Conference on Computer Aided Design,* pp. 133-135.

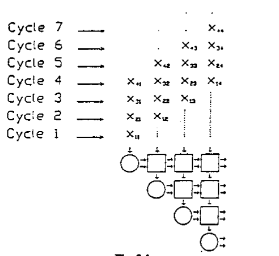
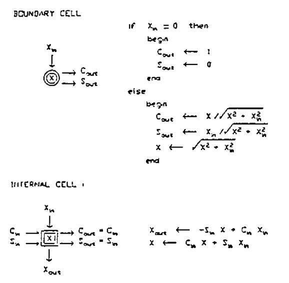


Fig. 2.1.

Triangular systolic array for matrix triangularization.



BOUNDARY CELL

```
if  $X_{in} = 0$  then
    begin
      $C_{out} \leftarrow 1$
      $S_{out} \leftarrow 0$
    end
else
    begin
      $C_{out} \leftarrow X / \sqrt{X^2 + X_{in}^2}$
      $S_{out} \leftarrow X_{in} / \sqrt{X^2 + X_{in}^2}$
      $X \leftarrow \sqrt{X^2 + X_{in}^2}$
    end
```

INTERNAL CELL :

```
$X_{out} \leftarrow -S_{in} X + C_{in} X_{in}$
$X \leftarrow C_{in} X + S_{in} X_{in}$
```

Fig. 2.3.

Microcode specifications of boundary cell and internal cell for orthogonal triangularization.

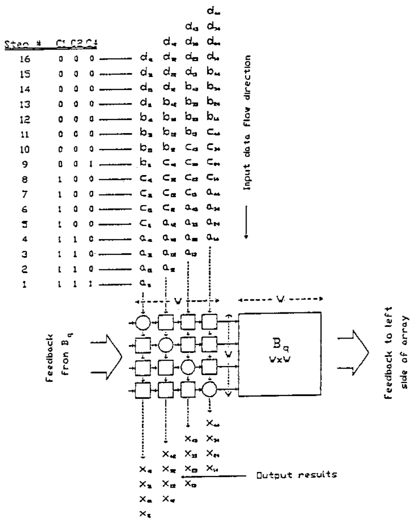
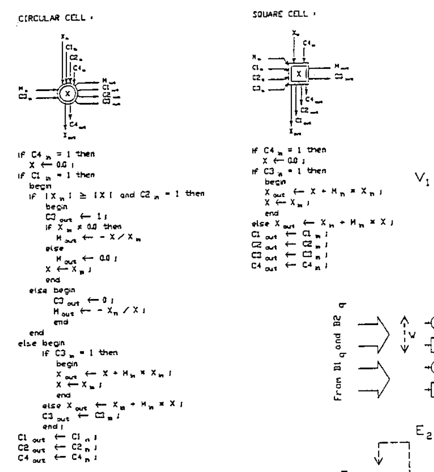


Fig. 3.1.

Dual mode systolic implementation of Faddeev's algorithm.

The number of cells needed is smaller and I/O bandwidth requirement is reduced.



CIRCULAR CELL :

```
if C4 $_{in}$ = 1 then
   X ← 0.0 ;
if C1 $_{in}$ = 1 then
   begin
   if | X $_{in}$ | ≥ | X | and C2 $_{in}$ = 1 then
       begin
       C3 $_{out}$ ← 1 ;
       if X $_{in}$ ≠ 0.0 then
          H $_{out}$ ← - X / X $_{in}$
       else
          H $_{out}$ ← 0.0 ;
       X ← X $_{in}$ ;
       end
   else begin
         C3 $_{out}$ ← 0 ;
         H $_{out}$ ← - X $_{in}$ / X ;
         end
   end
else begin
      if C3 $_{in}$ = 1 then
         begin
         X $_{out}$ ← X + H $_{in}$ * X $_{in}$ ;
         X ← X $_{in}$ ;
         end
      else X $_{out}$ ← X $_{in}$ + H $_{in}$ * X ;
      C3 $_{out}$ ← C3 $_{in}$ ;
      end ;
C1 $_{out}$ ← C1 $_{in}$ ;
C2 $_{out}$ ← C2 $_{in}$ ;
C4 $_{out}$ ← C4 $_{in}$ ;
```

SQUARE CELL :

```
if C4 $_{in}$ = 1 then
   X ← 0.0 ;
if C3 $_{in}$ = 1 then
   begin
   X $_{out}$ ← X + H $_{in}$ * X $_{in}$ ;
   X ← X $_{in}$ ;
   end
else X $_{out}$ ← X $_{in}$ + H $_{in}$ * X ;
C1 $_{out}$ ← C1 $_{in}$ ;
C2 $_{out}$ ← C2 $_{in}$ ;
C3 $_{out}$ ← C3 $_{in}$ ;
C4 $_{out}$ ← C4 $_{in}$ ;
```

Fig. 3.2.

Microprogram specifications of the circular and square cells

for the array's dual mode operation.

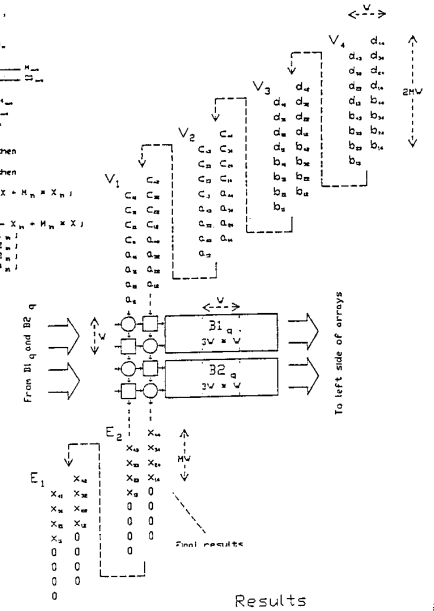
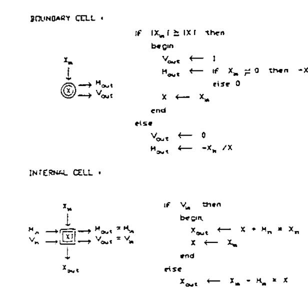


Fig. 4.1.

L-tuple arrays system processing a problem larger than the I/O bandwidth w.

Again w = 2, n = 4 and m = 2. With L = 2 arrays, the problem is solved in one iteration.



BOUNDARY CELL :

```
if  $|X_{in}| \geq |X|$  then
    begin
      $V_{out} \leftarrow 1$
      $H_{out} \leftarrow$ if $X_{in} \neq 0$ then $-X/X_{in}$
                  else 0
      $X \leftarrow X_{in}$
    end
else
    $V_{out} \leftarrow 0$
    $H_{out} \leftarrow -X_{in} / X$
```

INTERNAL CELL :

```
if  $V_{in}$  then
    begin
      $X_{out} \leftarrow X + H_{in} * X_{in}$
      $X \leftarrow X_{in}$
    end
else
    $X_{out} \leftarrow X_{in} - H_{in} * X$
```

Fig. 2.2.

Microcode specifications of boundary cell and internal cell

for Gaussian elimination with neighbor pivoting.

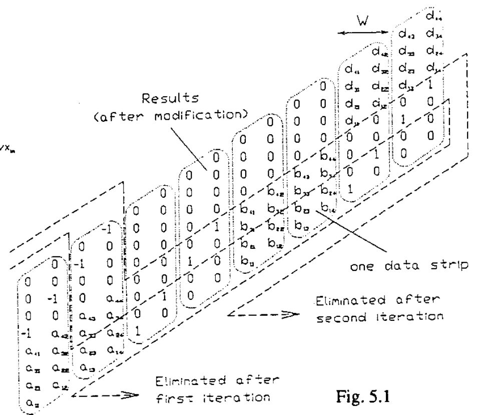


Fig. 5.1

Parallel decomposition of x = 3 horizontally compatible problems.

For this example, n = 4, w = 2, and m = 2.

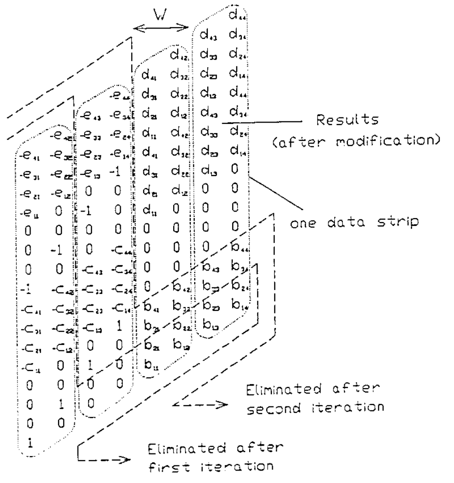


Fig. 5.2.

Parallel decomposition of y = 3 vertically compatible problems. Again n = 4, w = 2 and m = 2

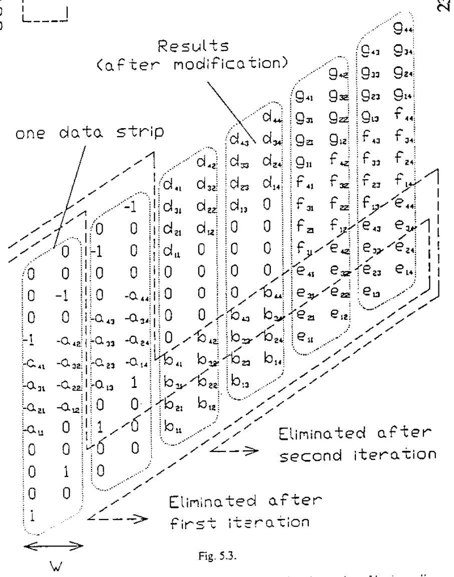


Fig. 5.3.

Parallel decomposition of x by y compatible problems. x = 2 is the number of horizontally

compatible problems, and y = 2 is the number of vertically compatible problems.

As before, n = 4, w = 2 and m = 2.