SIZE INDEPENDENT IMPLEMENTATION OF MATRIX OPERATIONS ON TASA - A TWO-DIMENSIONAL ARRAY MATRIX ARCHITECTURE

Hai Van Dinh Le, and Marek A. Perkowski

Department of Electrical Engineering, Portland State University,
Portland, OR 97207, tel. (503) 725-3806.

ABSTRACT

A Two-dimensional Array Systolic Architecture (TASA) is a general-purpose parallel computer for solving a wide class of computationally intensive problems. Application of TASA to implement Faddeev algorithm for classical matrix algebra operations was shown in [3]. It is presented below that this architecture is problem size independent for Faddev algorithm.

Nearly 75% of computer applications involve some kind of matrix manipulation. Faddev algorithm [1] is a base of several new and highly parallel architectures for general purpose matrix operations [2,4,5]. In [3] an implementation of Faddev algorithm with TASA has been introduced that has several advantages over the algorithms from [2,4,5]. They include: better performance for smaller cost, easy reconfigurability, maximum overlaps between consecutive computations, processing of sparse arrays, and increased throughput for several extensions to Faddeev algorithm. What is most important, TASA (Fig. 1) uses a *mesh structure* suitable for very many other CAD algorithms, in contrast to the well-known Faddev and other matrix architectures that require various and exotic structures. It will be shown that TASA is truly problem size independent, i.e. matrices which are arbitrarily large can be easily decomposed to be processed by a fixed size array.

Solving Size Independent Problems.

Faddeev's algorithm calculates $\mathbf{C\,A^{-1}\,B + D}$ from

$$\begin{matrix} \mathbf{A\ B} \\ \mathbf{-C\ D} \end{matrix} \qquad (2.1)$$

Since the underlying procedure to carry it out 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 can execute both *Gaussian elimination with neighbor pivoting* or *orthogonal triangularization*. The array consists of two types of cells: the *boundary cells* (represented by circles) and the *internal cells* (represented by squares). 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* (Fig. 2) allows TASA to execute different procedures. TASA reduces the I/O bandwidth requirement by half and the number of cells needed by more than one third, comparing to [2,4,5]. Functionally, there are two types of cells. The first type consists of all the *diagonal cells* (circles) of the array and the second type of all the *non-diagonal cells* (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*. 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. It performs *Gaussian elimination with neighbor pivoting on* **A**, and ordinary Gaussian elimination on **C**. 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 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.

The circular cell relies on three external *control signals C1*, C2, and C4 (binary) for internal computation. Itself, it generates signal C3. All those binary signals are broadcasted locally by a cell to its neighbors. The square cell passes received control signals C3, C4 to neighboring cells in unchanged form. 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 and the array is in **a T mode.** Otherwise, with C1 *false*, diagonal cells function like square cells, and the array is in the **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.

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.**

Generated internally by diagonal cells when they are in T mode, C3 is the functional equivalent of $\mathbf{M}_{out}$ of the boundary cell. 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*. A virtue of TASA is that it can readily handle problems of *arbitrary size* without requiring any architectural modification, while the throughput can be improved proportionally by adding any number of arrays to the existing system. This gives the array a degree of flexibility that makes it truly useful in real life implementation: performance is adjustable according to cost constraint while versatility is preserved regardless of expansion of any size. For problems larger than array size, the input data flow shown in Fig. 1 is decomposed into smaller strips which are processed continuously by the array, one after another. The intermediate results from each strip will then be fed back to the array for further processing. This vertical feedback and the horizontal feedback of the modification factors constitute two dimensional feedback paths for the array.

Input Decomposition and Vertical Feedback Path.

With matrices of size **n** *where* **n** *is* **m** *times the* available bandwidth **w** , (2.1) can be parallely decomposed into **2m** strips, each **w** in width and **2n** in length as in [2]. Each strip in turn consists of **2m** $\mathbf{w \times w}$ blocks which are of the same size as the array.

For **w = 2, n = 4** and **m = 2**, Fig. 1 shows an array with its input data flow decomposed parallely into four strips numbered from V_1 to V_4. These strips are processed by the array one after another continuously. Note that the strips of intermediate results all have leading blocks of zeros. The procedure begins with the array set to T mode as V_1 arrives. While V_1 is being processed, a horizontal data stream consisting of values M_{out} and signals C3 is generated and moved rightwards into B_q. Subsequently, the array is switched to S mode for the *computation of the remaining strips,* V_2 *to* V_4. *In this mode,* the contents of B_q is recirculated back to the array as vertical data of each strip arrive, thus ensuring proper processing. As shown in Fig. 1, each input strip V_2, V_3, V_4 generates an output strip $V_2^{(1)}$, $V_3^{(1)}$, $V_4^{(1)}$, of length $\mathbf{(2m - 1)w = 6}$ that is preceded by a block of zeroes as it emerges from the array. In Fig. 3, these intermediate results are stripped of their zero blocks and then fed back to the array where the above procedure is repeated. The final results, strips E_1 and E_2, come out from the bottom of the array, each $\mathbf{(2m - 2)w = 4}$ in length and likewise, is preceded by a zero block. are stripped of their leading blocks of zeroes before re-entering the array.

Fig. 4 shows a mapping of input and output data flow of each iteration to array execution steps. The dash/dotted lines represent input strips, while the dotted lines represent the output strips. Notice that input data flow of the second iteration is optimized, i.e. zero blocks that exist between output strips of the first iteration are eliminated. In general, a $\mathbf{w \times w}$ array will solve a problem which is decomposed into **2m** strips of length **2mw** and width **w**, in **m** iterations. During the $\mathbf{i^{th}}$ iteration, where **i = 1, 2, ... , m**, the array eliminates the strip V_i (in T mode) and reduces the length of each of the remaining strips by **w** (in S mode). This is because each remaining strip leaves behind one $\mathbf{w \times w}$ block of data in the **X** registers as it is being processed by the array, and subsequently emerges with a $\mathbf{w \times w}$ block of zeroes preceding it. These zero blocks can be skipped in the next iteration to shorten processing time without incurring any error. Final results after the $\mathbf{m^{th}}$ iteration consists of **m** strips, each **mw** in length and **w** in width.

The number of steps needed for the array of Fig. 1 to compute $\mathbf{C\,A^{-1}\,B + D}$ is:

$$(2w-1) + \sum_{k=1}^{m}(2m-k+1)^2\, w = 7/3(m^2n) + 3/2(mn) + 1/6(n) + 2w - 1 = O(m^2n)$$

Controls and Horizontal Feedback Path

In Fig. 4, values of C1, C2, and C4 necessary for the above example are illustrated at each step. C3 is not shown since it is dependent on input data and generated on the fly by the diagonal cells. For each control signal, a 1 represents the boolean value *true* and 0 represents *false;* when a signal remains unchanged from its previous value, a dash (-) entry is entered. The pattern is as follow: for each iteration, C1 is *true* during the first strip and *false* throughout the remaining strips. C2 is *true* only where pivoting is allowed, i.e. the portion of the first strip which contains data elements of matrix **A**, and *false* anywhere else. C4 clears the **X** registers of the array each time a new strip arrives, therefore it is true at the first step of each strip and false elsewhere.

In general, an input strip with **N** blocks of vertical data will generate a corresponding **N** blocks of horizontal modification factors pairs (M_{out} and C3); thus, the storage of the horizontal data stream should be **N** blocks long so that timings for horizontal feedback are accurate. Because the array itself acts as a $\mathbf{w \times w}$ block of storage, for each i^{th} iteration, the FIFO queue B_q should be $(2m - i)w$ long. With **m = 2** and **w = 2**, Figs. 1,3 show the corresponding length of B_q for each iteration.

The buffer B_q should have the addressing capability such that its length can vary in units of blocks. This permits the array to solve problems of arbitrary size, as long as B_q maximum length is adequate for the largest of them.

Multiple Arrays Configurations.

Even though both have throughput time $O(m^2n)$, the parallel decomposition system from [2] is slightly faster when compared to the array from Fig. 1. (Let us observe that the system from [2] cannot solve problems with **m > 4**). Given a problem, the former will solve it with steps less than the latter. This stems from its use of two subarrays, where some overlaps in processing are possible when the S array is working on a strip while the T array processes intermediate results from the previous strip. Likewise, by using multiple arrays, the system of Fig. 5 gives better throughput than the single array of Fig. 1 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. 5), or L-subarrays system. Again w = 2, n = 4 and m = 2. The problem is solved in one iteration. In Fig. 6, control and timing sequences of Fig. 5 subarrays are illustrated. Because the input strips $V_4^{(1)}$ of the second array are interspersed by blocks of zeroes which cannot be removed, buffer $B\,2_q$ is required to have the same length as $B\,1_q$, instead of being one block shorter. In general, a problem requiring m iterations on a single array will need only k = m/L iterations on a system of L-tuple arrays, assuming that m is an exact multiple of L. After each i^{th} iteration, the length of partial results will be $(2m - iL)^2w$. Hence, the system will compute $CA^{-1}B + D$ of such a problem in

$$(L+1)w - 1 + \sum_{k=1}^{m/L}(2m-(k-1)L)^2w = 7/3(kmn) + 3/2(mn) + 1/6(nL) + (L+1)w - 1$$

steps. The first part represents the number of steps taken for input data of the last iteration to traverse the system, and the summation term gives the number of steps to feed input data of all iterations into the system. Final results in this case always emerge from the bottom of the last array of the system. Thus, when m = L (as in the example used in Fig. 5), $CA^{-1}B + D$ is computed in a single pass with total processing time equal to $(4m+1)n + w - 1 = O(mn)$, which is identical to the performances of the decomposed systems from [2]. However, note that the system of Fig. 5 is totally independent of problem's size and the number of cells used is smaller since the T arrays are eliminated.

When m is not an exact multiple of L , that is when $m_{mod\,L} \neq 0$, the number of iterations required to complete the problem is $k = \lfloor m/L \rfloor$, with the k^{th} iteration employing only the first $m_{mod\,L}$ subarrays of the system. The total processing time will be

$$(m_{mod\,L}+1)w - 1 + \sum_{k=1}^{\lfloor m/L \rfloor}(2m-(k-1)L)^2w$$

Again, the summation term represents the time necessary to feed input data of k iterations into the system. However, since only the first $m_{mod\,L}$ subarrays of the system are used during the k^{th} iteration, final results will emerge from the bottom of the $m_{mod\,L}^{th}$ subarray, instead of the last subarray. Therefore, the first term of the throughput equation reflects the shorter path through which data has to traverse during the k^{th} iteration.

REFERENCES: [1] D. K. Faddeev and V. N. Faddeeva, Computational Methods of Linear Algebra, *W. H. Freeman and Company*, 1963, pp. 150-158. [2] H.Y.H. Chuang, G. He, "A Versatile Systolic Array for Matrix Computations", *Proc. Intern. Symp. Comp. Arch., 1985, pp. 315-322.* [3] H. V. D. Le, and M. A. Perkowski, "A New General Purpose Systolic Architecture for Matrix Computations". *Proc. Intern. Conf. Comput. Inform.*, Toronto, May 23-29, 1989, pp. 182-185. [4] J. G. Nash and S. Hansen, "Modified Faddeev Algorithm for Matrix Manipulation," *Proc. SPIE*, Vol. 495, August 1984, pp. 39-46. [5] J. G. Nash, "A Systolic/Cellular Computer Architecture for Linear Algebraic Operations," *Proc. Intern. Conf. Robot. Autom., March 1985*, pp. 779-784.

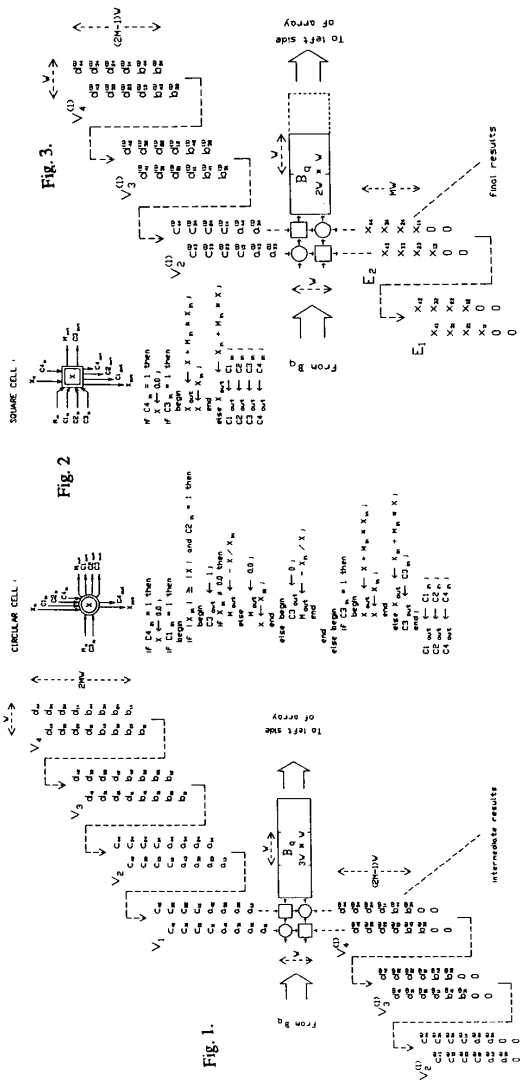

Fig. 1.

Fig. 2

Fig. 3.

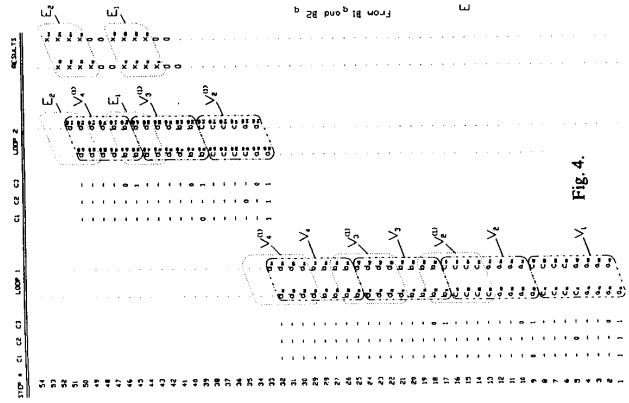

Fig. 4.

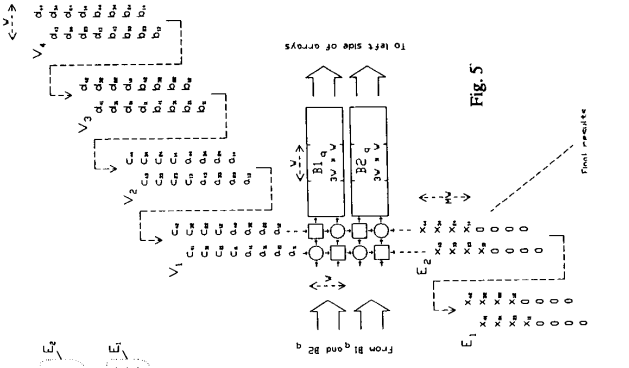

Fig. 5

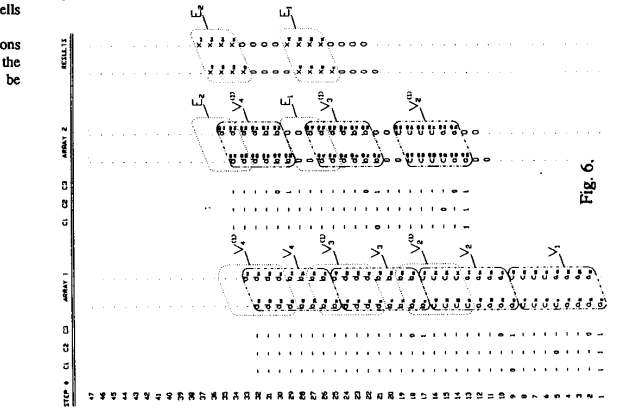

Fig. 6.