Wavefront Array Processor: Language, Architecture, and Applications

SUN-YUAN KUNG, MEMBER, IEEE, K. S. ARUN, STUDENT MEMBER, IEEE, RON J. GAL-EZER, STUDENT MEMBER, IEEE, AND D. V. BHASKAR RAO, STUDENT MEMBER, IEEE

***Abstract*—This paper describes the development of a wavefront-based language and architecture for a programmable special-purpose multiprocessor array. Based on the notion of computational wavefront, the hardware of the processor array is designed to provide a computing medium that preserves the key properties of the wavefront. In conjunction, a wavefront language (MDFL) is introduced that drastically reduces the complexity of the description of parallel algorithms and simulates the wavefront propagation across the computing network. Together, the hardware and the language lead to a programmable wavefront array processor (WAP). The WAP blends the advantages of the dedicated systolic array and the general-purpose data-flow machine, and provides a powerful tool for the high-speed execution of a large class of matrix operations and related algorithms which have widespread applications.**

***Index Terms*—Asynchrony, computational wavefront, concurrency, data-flow computing, matrix data-flow language, signal processing, systolic array, VLSI array processor, wavefront architecture.**

I. Introduction

A. VLSI Signal Processing

With the rapidly growing microelectronics technology leading the way, modern signal processing is undergoing a major revolution. The past two decades have witnessed a steep increase in the complexity of computations, processing speed requirements, and the volume of data handled in various signal processing applications. The availability of low cost, high density, fast VLSI devices has opened a new avenue for implementing these increasingly sophisticated algorithms and systems [1], [2]. While a major improvement in device speed is foreseen, it is in no way comparable to the rate of increase of throughput required by modern real-time signal processing. In order to achieve such increases in throughput rate, the only effective solution appears to be highly concurrent processing. Consequently, it has become a major challenge to update the current signal processing techniques so as to maximally exploit their potential for concurrent execution.

In a broad sense, the answer to this challenge lies in a cross-disciplinary research encompassing the areas of algorithm analysis, parallel computer design, and system applications. In this paper, we therefore introduce an integrated research approach aimed at incorporating the vast VLSI computational capability into modern signal processing applications.

The traditional design of parallel computers and languages is deemed unsuitable for our purposes. It usually suffers from heavy supervisory overhead incurred by synchronization, communication, and scheduling tasks, which severely hamper the throughput rate which is critical to real-time signal processing. In fact, these are the key barriers inherent in very large scale computing structure design. Moreover, although VLSI provides the capability of implementing a large array of processors on one chip, it imposes its own constraints on the system. Large design and layout costs [2] suggest the utilization of a repetitive modular structure. In addition, communication, which costs the most in VLSI chips in terms of area, time, and energy, has to be restricted (to *localized communication*) [1]. In general, highly concurrent systems require this locality property in order to reduce interdependence and ensuing waiting delays that result from excessive communication [1]. This locality constraint prevents the utilization of centralized control and global synchronization. The resulting use of *asynchronous distributed control* and *localized data flow* is an effective approach to the design of very large scale, highly concurrent computing structures.

B. A Special-Purpose VLSI Array Processor

The above restrictions imposed by VLSI will render the general-purpose array processor rather inefficient. We therefore restrict ourselves to a special class of applications, i.e., recursive[1] and local data-dependent algorithms, to conform with the constraints imposed by VLSI. This restriction, however, incurs little loss of generality, as a great majority of signal processing algorithms possess these properties. One typical example is a class of matrix algorithms. It has recently been indicated that a major portion of the computational needs for signal processing and applied mathematical problems can, in fact, be reduced to a basic set of matrix operations and other related algorithms [3], [4]. Therefore, a special-purpose parallel machine for processing these typical computational algorithms will be cost effective and attractive in VLSI system design.

Manuscript received January 11, 1982; revised June 18, 1982. This work was supported in part by the Office of Naval Research under Contract N00014-80-C-0457, N00014-81-K-0191, by the National Science Foundation under Grant ECS-80-16581, and by the Defense Advanced Research Projects Agency under Contract MDA903-79-C-0680.

The authors are with the Department of Electrical Engineering–Systems, University of Southern California, Los Angeles, CA 90089.

[1] In a recursive algorithm, all processors do nearly identical tasks, and each processor repeats a fixed set of tasks on sequentially available data.

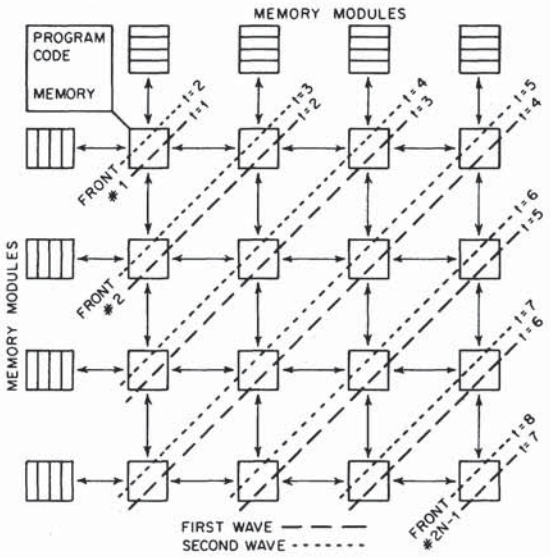

Fig. 1. The WAP configuration.

Very significantly, these algorithms involve repeated application of relatively simple operations with regular localized data flow in a homogeneous computing network. This leads to an important notion of *computational wavefront*, which portrays the computation activities in a manner resembling a wave propagation phenomenon. More precisely, the recursive nature of the algorithm, in conjunction with the localized data dependency, points to a continuously advancing wave of data and computational activity. The computational sequence starts with one element and propagates through the processor array, closely resembling a physical wave phenomenon (cf. Fig. 1). In fact, all algorithms that have *locality* and *recursivity* (and are thus implementable on our VLSI array processor) will exhibit this wave phenomenon. Therefore, the notion of a computational wavefront has attracted the attention of many researchers [2], [5]–[13].

The wavefront concept provides a firm theoretical foundation for the design of highly parallel array processors and concurrent languages. In addition, this concept appears to have some distinct advantages.

First, the wavefront notion drastically reduces the complexity in the description of parallel algorithms. The mechanism provided for this description is a special-purpose, wavefront-oriented language [7], [9], [10]. Rather than requiring a program for each processor in the array, this language allows the programmer to *address an entire front of processors*.

Second, the wavefront notion leads to a wavefront-based architecture which preserves Huygen's principle [14], and ensures that wavefronts never intersect. Therefore, a wavefront architecture can provide *asynchronous waiting* capability, and consequently, can cope with timing uncertainties, such as local clocking, random delay in communications, and fluctuations of computing times. In short, the notion lends itself to a (asynchronous) data-flow computing structure that conforms well with the constraints of VLSI.

The integration of the wavefront concept, the wavefront grammable computing network, which we will call the wavefront array processor (WAP). The WAP is, in a sense, an *optimal tradeoff* between the *globally synchronized* and *dedicated systolic array* [1], [15], [16] (that works on a similar set of algorithms), and the *general-purpose data-flow multiprocessors* [17]–[23]. It provides a powerful tool for the high-speed execution of a large class of algorithms which have widespread applications.

C. Organization

The organization of the rest of the paper is as follows. Section II elaborates on the computational wavefront. Section III proposes a wavefront-oriented language (MDFL) for programming the WAP, and provides a programming methodology. Section IV explains a possible hardware organization and architecture for the WAP. Section V illustrates some applications of the WAP by means of MDFL programs. Section VI compares the WAP to other array processors, such as the systolic array, data-flow multiprocessors, and conventional SIMD arrays like the Illiac IV.

II. Concept of Computational Wavefront

The wavefront array processor is configured in a square array of $N \times N$ processing elements with regular and local interconnections (cf. Fig. 1).

The computing network serves as a (data) wave-propagating medium. The notion of computational wavefront can be best explained by a simple example. To this end, we shall consider matrix multiplication as being representative. Let $A = \{a_{ij}\}$, $B = \{b_{ij}\}$, and $C = A \times B = \{c_{ij}\}$ all be $N \times N$ matrices. The matrix A can be decomposed into columns A_i and matrix B into rows B_j, and therefore,

$$C = A_1 * B_1 + A_2 * B_2 + \cdots + A_N * B_N. \tag{1}$$

The matrix multiplication can then be carried out in N recursions, executing

$$C^{(k)} = C^{(k-1)} + A_k * B_k \tag{2}$$

recursively for $k = 1, 2, \cdots, N$.

The exploitation of parallelism is now evident with the availability of $N \times N$ processors. A parallel algorithm for matrix multiplication is fairly simple. However, most existing parallel programs would need global interconnections in the computing network, while localized interconnections and data flow are much more desirable in VLSI systems.

The topology of the matrix multiplication algorithm can be mapped naturally onto the square, orthogonal $N \times N$ matrix array of the WAP (cf. Fig. 1). To create a smooth data movement in a localized communication network, we make use of the computational wavefront concept. For the purpose of this example, a wavefront in the processing array will correspond to a mathematical recursion in the algorithm. Successive pipelining of the wavefronts will accomplish the computation of all recursions.

As an example, the computational wavefront for the first

Suppose that the registers of all the processing elements (PE's) are initially set to zero:

$$C_{ij}^{(0)} = 0 \qquad \text{for all } (i, j);$$

the entries of A are stored in the memory modules to the left (in columns), and those of B in the memory modules on the top (in rows). The process starts with PE (1, 1), where

$$C_{11}^{(1)} = C_{11}^{(0)} + a_{11} * b_{11} \tag{3}$$

is computed. The computational activity then propagates to the neighboring PE's (1, 2) and (2, 1), which will execute

$$C_{12}^{(1)} = C_{12}^{(0)} + a_{11} * b_{12} \tag{4}$$

and

$$C_{21}^{(1)} = C_{21}^{(0)} + a_{21} * b_{11}. \tag{5}$$

The next front of activity will be at PE's (3, 1), (2, 2), and (1, 3), thus creating a computation wavefront traveling down the processor array. This computational wavefront is similar to optical wavefronts (they both obey Huygen's principle) since each processor acts as a secondary source and is responsible for the propagation of the wavefront. It may be noted that wave propagation implies localized data flow. Once the wavefront sweeps through all the cells, the first recursion is over. As the first wave propagates, we can execute an *identical* second recursion in parallel by *pipelining* a second wavefront immediately after the first one. For example, the (1, 1) processor will execute

$$C_{11}^{(2)} = C_{11}^{(1)} + a_{12} * b_{21} \tag{6}$$

and so on. In general, the (i, j)th processor will execute the kth recursion,

$$C_{ij}^{(k)} = a_{i1} * b_{1j} + a_{i2} * b_{2j} + \cdots + a_{ik} * b_{kj}. \tag{7}$$

The pipelining is feasible because the wavefronts of two successive recursions will never intersect (Huygen's wavefront principle), as the processors executing the recursions at any given instant will be different, thus avoiding any contention problems.

Locality, *regularity*, *recursivity*, and *concurrency* lead to the wavefront phenomenon. Thus, all algorithms which possess these properties will exhibit computational wavefronts. The notion of computational wavefronts leads to a wavefront-based language (a modified data-flow language) to program the processor array. This language (called the matrix data-flow language) is especially powerful for matrix and other related algorithms. It will be developed in the next section.

III. A Wavefront Language: Matrix Data-Flow Language

In contrast to the heavy burden of scheduling, resource sharing, as well as the control of processor interactions encountered in programming a general-purpose multiprocessor, the computational wavefront notion can facilitate the description of parallel algorithms and drastically reduce the complexity of parallel programming. In this section, we inscription of computational wavefronts and the corresponding data flow in a large class of algorithms (which exhibit the recursivity and locality mentioned earlier).

Since matrix algorithms are typical of this class, we call the language the matrix data-flow language (MDFL). This wavefront language permits the array processor to be programmable, broadens the range of applications, and also makes possible the simulation and verification of parallel algorithms.

A. Matrix Data-Flow Language

There exist two approaches to programming the WAP: a local approach describing the actions of each processing element, and a global approach describing the actions of each wavefront. To allow the user to program the WAP in both of these fashions, two versions of MDFL are proposed: global and local MDFL.

A global MDFL program describes the algorithm from the viewpoint of a wavefront, while a local MDFL program describes the operations of an individual processor. More precisely, the perspective of a global MDFL programmer is of *one wavefront passing across all the processors*, while the perspective of a local MDFL programmer is that of *one processor encountering a series of wavefronts*.

From the macroscopic point of view, a higher level language, closer to the algorithm, is desired for reducing the heavy burden on the programmer. Global MDFL provides such a tool, as it is easier to view the algorithm as a series of wavefronts. At the microscopic level, each PE executes its own set of instructions and performs localized interprocessor transactions. Implementing a program on such a system requires transforming the high level description (in global MDFL) into a set of lower level programs (in local MDFL) for the individual processors.[2] We have developed a compiler to do such a mapping of global wavefront-oriented MDFL programs into local programs for individual processors.

B. MDFL Instruction Set

At each front of activity, the computational wavefront performs similar tasks in all the processors involved (cf. Section II, matrix multiplication example and Section III-D on programming methodology). Hence, global (wavefront) instructions and local (processor) instructions are nearly identical. Most of the MDFL instruction set is, therefore, common to both global and local MDFL.

Table I is a complete list of the MDFL instruction repertoire. For the complete semantics and more detailed syntax, the reader is referred to a recent publication [10]. Unless otherwise specified, the following instructions belong to both global and local MDFL. The proposed instruction set is functionally complete, but improvements and modifications are still underway to incorporate additional features such as double precision, etc. There also appears to be room for the

[2] It suffices, in general, to describe a wavefront algorithm at four locations of the array: corner, FirstRow, FirstColumn, and Interior PE's. Consequently, every global MDFL program gets compiled into four slightly different local

TABLE I
MDFL INSTRUCTION SET

```
Data Transfer Instructions

FLOW <SOURCE REGISTER>, <DIRECTION>;
FETCH <DESTINATION REGISTER>, <DIRECTION>;
READ;


Recursion Oriented Instructions

REPEAT ... UNTIL TERMINATED;
WHILE WAVEFRONT IN ARRAY DO
        BEGIN ... END;
SET COUNT <NUMBER OF WAVEFRONTS>;
DECREMENT COUNT;
ENDPROGRAM.


Conditional Instructions

IF EQUAL THEN <STATEMENT>;
IF NOT-EQUAL THEN <STATEMENT>;
IF GREATER THEN <STATEMENT>;
IF LESS-THAN THEN <STATEMENT>;
IF <DIRECTION> DISABLED THEN <STATEMENT>;
CASE KIND =
        (1,1) : <STATEMENT>;
        (1,*) : <STATEMENT>;
        (*,1) : <STATEMENT>;
        INT   : <STATEMEMT>;
   ENDCASE;


Internal Processor Instructions

TSR     <SOURCE>, <DESTINATION>;
ADD     <SOURCE #1>, <SOURCE #2>, <DESTINATION>;
SUB     <SOURCE #1>, <SOURCE #2>, <DESTINATION>;
MULT    <SOURCE #1>, <SOURCE #2>, <DESTINATION>;
DIV     <SOURCE #1>, <SOURCE #2>, <DESTINATION>;
SQRT    <SOURCE>, <DESTINATION>;
CMP     <SOURCE #1>, <SOURCE #2>;
TST     <SOURCE>;
STORE;
NOP;
RESET;
BEGIN ... END;
DISABLE-SELF;
```

development of a higher level language suitable for the non-specialist programmer.

C. Programming Methodology

In this subsection, we provide guidelines for programming in global MDFL. The most straightforward method of programming the WAP would be to explicitly spell out the actions of each wavefront at each of its $(2n-1)$ positions (fronts) (cf. Fig. 1). Nevertheless, the regularity and recursivity in almost all matrix algorithms allows us to assume the following.

1) Space Invariance: The tasks performed by a wavefront in a particular kind of processor must be identical at all $(2n-1)$ fronts.

2) Time Invariance: Recursions are identical.

Accordingly, global MDFL provides two repetitive constructs, the space repetitive construct

WHILE WAVEFRONT IN ARRAY DO
BEGIN ⟨TASK T⟩ END

(so that T is repeated at all fronts), and the time repetitive construct

REPEAT ⟨ONE RECURSION⟩ UNTIL TERMINATED

(so that the same recursion is repeated).

successive wavefronts are pipelined through the array. As soon as the kth wavefront is propagated, the (1, 1) processor initiates the $(k+1)$st wavefront.

To allow for more than one wavefront per recursion, the complete global MDFL program will have the syntax

```
BEGIN
  SET COUNT ( );
  REPEAT
      ⟨TASKS A⟩;
      WHILE WAVEFRONT IN ARRAY DO
        BEGIN
          ⟨TASKS B⟩;
        END;
      WHILE WAVEFRONT IN ARRAY DO
        BEGIN
          ⟨TASKS C⟩;
        END;
      ⟨TASKS D⟩;
      DECREMENT COUNT;
  UNTIL TERMINATED;
ENDPROGRAM.
```

Each recursion will execute the instructions within the REPEAT ··· UNTIL construct. The number of recursions is set by SET COUNT. In this example, a recursion consists of two wavefronts. At the start, tasks A are performed only at the (1, 1) processor. The first wavefront of each recursion will perform tasks B at each of its $(2n-1)$ fronts. The second wave will execute tasks C in each of these fronts immediately after tasks B have been concluded. It should be noted that the number of different wavefronts within a recursion may vary from one application to another. At the end of each recursion, COUNT is decremented. When it becomes zero, TERMINATED is set and a "phase" of identical recursions is over.

The corresponding local MDFL program for interior processors (cf. Section III-A) will be

```
REPEAT
    ⟨TASKS B′⟩
    ⟨TASKS C′⟩
UNTIL TERMINATED;
```

where B' and C' are the compiled versions of B and C, with only relevant portions of the CASE statement extracted. The conversion of a global program into its local versions is thus fairly straightforward.

Certain syntax rules [10] are needed to ensure that there is no circular waiting for data between adjacent PE's. They also ensure that neighbors will not contend for the interprocessor bus at the same time. Concisely, the rules dictate that FETCHes in one direction precede (and equal in number) the FLOWs in the opposite direction, and that the data FETCHing precede any computation (cf. data-driven computation in Section IV-A). The complete proof of the claim that these rules will prevent deadlock and bus contention is omitted here [24].

Based on the above guidelines, a complete global MDFL program for matrix multiplication follows.

More programming examples will be provided in Section

Program 1. Matrix multiplication.

```
Array Size:     N x N

Computation:    C = A x B
                 th                (k)     (k-1)
                k   wavefront:  c    =  c      + a  b
                                 ij      ij       ik kj

                                k = 1, 2, ... , N

Initial:   Matrix A is stored in the Memory Module  (MM)

        on the left (stored row by row).  Matrix B is in

        MM on  the top and is  stored  column by column.

Final:     The result will be in the C registers.


1:      BEGIN
        SET COUNT 3;
        REPEAT;
           WHILE WAVEFRONT IN ARRAY DO
5:           BEGIN
                FETCH B, UP;
                FETCH A, LEFT;
                FLOW A, RIGHT;
                FLOW B, DOWN;
                ! C <- C + AB; *
10:             MULT A, B, D;
                ADD C, D, C;
             END;
           DECREMENT COUNT;
        UNTIL TERMINATED;
15:     ENDPROGRAM.
```

* Comments are enclosed between "!" and ";"

D. Summary of MDFL Features

Future VLSI multiprocessors must support massive concurrency to achieve a significant increase in performance; consequently, a base language for parallel computers must allow expression of concurrency of program execution on a large scale [20]. However, few languages support the idea of large scale concurrency, and only weak notions of locality exist in most of them [25].

It must be noted that a parallel program is not just an ensemble of separate programs for individual processors. More importantly, it should also define coordination between PE's including their interdependence for data and the sequencing of their tasks. As such, it is a tall order.

At present, data-flow languages appear to be the best candidate as the base language of parallel computers. Data-flow languages are asynchronous, data driven, and algorithmic. They avoid centralized control and shared memory to achieve *asynchrony* and *maximal concurrency* [20], [26]–[29].

MDFL basically possesses all these properties of data-flow languages. It shares the principle of *data-activated computation* and its consequent advantages. In addition, MDFL has a rather distinctive feature of regularity, and is built around the notion of locality. It is very close to the algorithms; hence, MDFL programs are modular and easy to understand. MDFL permits the programmer to address a front of processors at the same time, instead of programming individual processors separately. Being wavefront oriented, it permits viewing the algorithm as the repetition of a computational sequence (i.e., the wavefront) progressing through the array. In short, the wavefront notion makes it possible to program an array of asynchronous processors in a simplistic fashion, and leads to

IV. Wavefront Architecture

The hardware of the processing array is designed to support MDFL. The main architectural considerations include four general aspects: 1) interprocessor communications, based on wavefront requirements; 2) the basic PE configuration; 3) interfacing with the host computer; and finally, 4) the extendability and reconfigurability issues.

A. Interprocessor Communication

The configuration of the processor array is as shown in the schematic diagram of Fig. 1. It provides for data and control communications between orthogonally adjacent processors and links to memory modules through the first row and first column of processors.

To simulate the phenomenon of wavefront propagation, the processors in the array must *wait* for a primary wavefront (of data), then perform its computation and, finally act as a *secondary source* of new wavefronts. To implement this wait, processors are provided with data transfer buffers. Hence, a FETCHing of data involves an inherent WAITing for the buffer to be filled (DATA READY) by the adjacent data sourcing processor. Thus, if the software ensures that the processor always performs data FETCH before the computation (cf. syntax rules of Section III-C), *the processing will not be initiated until the arrival of the data wavefront* (this is similar to the concept of data flow machines [17]–[23], [26]–[29]). Each processor can FLOW data to the input buffers of the neighboring PE's, thus acting as a secondary source of data wavefronts (Huygen's principle). To avoid the overrunning of data wavefronts (in conformation with Huygen's principle), the processor hardware ensures that a processor cannot send new data to the buffer unless the old data have been used by the neighbor. Thus, the wavefront concept suggests that interprocessor communication employ buffers and "DATA READY/DATA USED" flags between adjacent processors.

The WAITs for wavefronts of data allow for *globally asynchronous* operation of processors, i.e., there is no need for global synchronization. Synchronization is a very critical issue in parallel processing, especially when one considers large scale systems. Two opposite timing schemes come to mind, namely, the synchronous and the asynchronous timing approaches. In the synchronous scheme, there is a global clock network which distributes the clocking signals over the entire chip. The global clock beats out the rhythm to which all the PE's in the array execute their tasks. In the basic synchronous configuration, all the PE's operate in unison, all performing the same identical operation. In contrast, the asynchronous scheme involves no global clock, and information transfer is by mutual convenience between each PE and its immediate neighbors. Whenever the data are available, the transmitting PE informs the receiver of the fact, and the receiver accepts the data when it needs them. It then conveys to the sender the information that the data have been used. This scheme can be implemented by means of a simple handshaking protocol [10], [30] (cf. Fig. 2).

B. PE Configuration

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.