Mapping Deep Nested Do-Loop DSP Algorithms to Large Scale FPGA Array Structures

Surin Kittitornkun, *Member, IEEE,* and Yu Hen Hu, *Fellow, IEEE*

***Abstract*—Recently, FPGAs (field programmable gate arrays) technology have made significant advances in both speed and capacity. Millions of logic gates are now available for reconfiguration programming. To fully exploit the potential of so many programmable devices, powerful design methodology must be developed. In this paper, we propose a novel systematic computer-aided design methodology that can efficiently implement deeply nested do-loop algorithms on a FPGA. Specifically, our design methodology maps the loop dependence graph onto a linear array of locally connected processing elements to exploit parallelism. Due to the regular structure of this linear array of processors, it can be easily implemented on a FPGA. While this method is based on conventional systolic array design methodology, our proposed approach exhibits two distinct features that contribute to its superior performance: 1) We developed a novel multiple-order dependence graph representation that is able to efficiently represent distinct, yet correct algorithm execution orders. 2) We developed new FPGA-specific architectural constraints during the mapping process. As such, FPGA implementations based on our approach will utilize much fewer lookup tables while achieving superior performance.**

***Index Terms*—Dependence graph, field programmable gate arrays (FPGAs), high-level synthesis, parallel computing, systolic array.**

I. Introduction

RECENTLY, field programmable gate arrays (FPGAs) have made significant advances in both speed and capacity. Current state-of-the-art FPGAs can contain up to ten million logic gates on a single chip and can run at up to 420 MHz [1]. Next generation FPGAs, with even faster clock rate and more logic gates, are just around the corner. During the past decade, FPGAs have out-grown their traditional role as a rapid prototyping tool for application specific integrated circuit (ASIC) development. Instead, many FPGAs have been deployed in consumer products facilitating hardware-assisted embedded processing [2]. Moreover, with on-line reconfiguration capability, many FPGAs now support run-time reconfigurable computing [3].

Such a powerful reconfigurable architecture offers tremendous opportunity to implement computation intensive signal and video processing algorithms at low power and low cost. The main challenge faced by design engineers is how to fully exploit the potentials of FPGA to realize real time, high throughput parallel multimedia signal processing algorithms. To achieve this goal, it calls for effective and efficient computer-aided design methodology that can assist a human designer to quickly explore the design space and identify the optimal design option.

In this paper, we consider the problem of implementing deeply nested Do-loop algorithms on a large scale FPGA chip. We focus on nested Do-loop algorithms because they appear frequently in computation-intensive signal and video processing algorithm formulations, such as motion estimation, two-dimensional (2-D) linear transformations, matrix/vector multiplication, and so on.

Unlike existing FPGA design tools, our approach assumes that the FPGA can be used to realize a linear array of processors (processing elements) with localized interprocessor communication. In other words, we assume a linear systolic array can be realized on a single FPGA chip. As such, our design methodology will be based on the conventional systolic array design methods [4] that map a nested DO-loop algorithm onto systolic array structures. Our approach is motivated by the following observations.

- The increase in the number of configurable logic gates per FPGA chip demands a systematic, structured architecture such as a systolic array.
- Nested loop algorithm formulation appears in numerous computation-intensive multimedia computing algorithms. Efficient implementation of deep nested loops can dramatically improve the performance.
- Systolic array design style can effectively exploit parallelism inherent in the nested loop algorithm and, therefore, reduce processing time. Moreover, such high computing throughput rates of systolic arrays can be achieved from FPGA due to its spatial parallelism and inherent temporal pipelining.
- In recent SRAM-based FPGA architecture [1], [5], [6], the on-chip programmable modules are organized in a regular fashion with pipelined local as well as long interconnects. Thus, it should facilitate an excellent implementation of systolic array structure.
- For mobile multimedia communication, low-power design is an essential design consideration. Shorter computing time implies less power consumption.

Existing signal and image processing algorithms to FPGA mapping tools/methodologies include [7]–[9] and [10]. They all focus on the previous generation of FPGAs. In [7], a behavioral description partitioning tool of algorithms is reported for multi-FPGA systems. The behavioral specification is represented by a data-flow graph at the coarse-grained level and a behavioral graph at a fine-grained level. During partitioning, embedded memory is utilized to save the interfacing pins among

Manuscript received September 22, 2001; revised March 31, 2002.

The authors are with the Department of Electrical and Computer Engineering, University of Wisconsin, Madison, WI 53706 USA (e-mail: kittitor@cae.wisc.edu; hu@engr.wisc.edu).

Digital Object Identifier 10.1109/TVLSI.2002.801622

1063-8210/03$17.00 © 2003 IEEE

FPGAs. The MATCH project [8] maps Matlab nested loop algorithms for FPGA implementation. The loop body is pipelined to a number of stages and scheduled subject to memory constraints so that each loop iteration is started every initiation period. Hence, the computation throughput is increased. In addition to Matlab, Einhardt, and Luk [9] target at loops written in C programming language. Finally, DG2VHDL [10] is a VHDL generator of the hardware counterpart of a nested loop algorithm. Due to identical functionality of each loop body, the mapping results in an array of small processing elements (PEs).

However, current FPGA mapping tools are inefficient for mapping nested loop algorithm to its regular structure since no built-in FPGA constructs have been integrated in the mapping. Our approach is distinct from these existing approaches in two major aspects.

1) We propose a novel representation of loop dependence graphs so that different execution orders of a loop can be represented in a concise format. By mapping this *multiple execution order dependence graph* (MODG) into a processor array, more feasible solutions may be uncovered and the solution will be closer to the globally optimal solution.
2) We formulate and impose additional architectural constraints during the mapping process based on the unique characteristics of a modern FPGA architecture. By doing so, the mapping can be performed to closely match the underlying FPGA architecture. As such, more efficient implementation can be accomplished. Specifically, we use the Xilinx Virtex architecture [5] as an example and deduce a set of architecture dependent constraints to achieve this goal.

The proposed methodology is aimed at mapping (scheduling and assignment) a nested loop algorithm to an array of PEs. Each PE is composed of datapath and control logics. The datapath can be pipelined to meet a high computing throughput rate while exploiting built-in constructs such as lookup table (LUT), local storage BlockRAM, and others. To illustrate this novel design methodology, we use block-matching mothion estimation as an example. Compared to the existing implementations, our results achieve compact FPGA layout and high hardware utilization with fewer I/Os.

We begin by introducing the notion of systolic array in Section II. The MODG formulation, a novel representation of nested loop, and its design methodology are elaborated next in Section III. An example of motion estimation is illustrated and discussed in Section IV. At the end, Section V concludes this paper along with its limitations.

II. Systolic Array and FPGA

In this section, we will describe how our research is motivated. Inspired by the classical systolic array, this paper is intended as an algorithm to FPGA mapping methodology. It overcomes the limitation of systolic mapping while exploiting the built-in constructs of FPGA.

A. Systolic Array Mapping or Space–Time Mapping

We will use the example of a matrix–matrix multiplication algorithm to illustrate the basic notation and formulation of systolic array space–time mapping.

Example 1: Matrix–matrix multiplication $C = A \times B c_{i,j} = \sum_{k=1}^{K} a_{i,k} b_{k,j}$ where $A = [a_{i,j}]$, $B = [b_{i,j}]$, and $C = [c_{i,j}]$ are matrices of appropriate dimensions. The corresponding nested loop algorithm formulation can be expressed as:

```
Listing 1: Matrix-Matrix Multiplication
Do i = 1 to M
 Do j = 1 to N
  c[i,j] = 0
  Do k = 1 to K
   c[i,j] = c[i,j] + a[i,k] × b[k,j]
  EndDo k
 EndDo j
EndDo i
```

where i, j, and k are loop indices. Together, they form an (iteration) *index space* where each point (i,j,k) within the loop bounds corresponds to a single execution of the *loop body*.

Let i_m be the mth level loop index of the of a loop nest. We denote $\vec{i} = (i_1, i_2, \ldots, i_n)^t \in \mathbf{Z}^n$ to be an n-Dimensional (n-D) column index vector of an n-level nested Do loop where $\mathbf{Z}$ is the space of integer numbers and $\vec{a}^t$ denotes the transposition of $\vec{a}$. Then, the n-D *index space* $\mathbf{J}^n$ can be expressed as

$$\mathbf{J}^n = \left\{ \vec{i} = (i_1, i_2, \ldots, i_n)^t \mid i_1, i_2, \ldots, i_n \in \mathbf{Z} \right\}. \quad (1)$$

In this example, the loop body consists of a single *recurrence equation*

$$c[i,j] = c[i,j] + a[i,k] \times b[k,j]$$

where $a[i,k]$ and $b[k,j]$ are *input variables* and their values are needed to execute this loop. $c[i,j]$ is an *output variable* whose value will be computed.

In Listing 1, the innermost k-loop is used to realize the summation of K product terms $a[i,k] \times b[k,j]$, $1 \leq k \leq K$. While $c[i,j]$ is the final result, it is also used to store intermediate results at kth iteration. In other words, the same memory address designated to $c[i,j]$ is assigned to new values multiple K times during the execution of the algorithm. In a *single-assignment* formulation [4], we introduce a set of new intermediate variables to store the intermediate results. As such, every variable will be assigned to a new value at most once during the execution of the algorithm.

In this example, the input variable $a[i,k]$ will be used in each of the j loops, and $b[k,j]$ will be used in each of the i loops. In particular, $a[i,k]$ will be made available to iterations with indices $\{(i,j,k)^t; 1 \leq j \leq N\}$, and $b[k,j]$ will be made available to iteration indices $\{(i,j,k)^t; 1 \leq i \leq M\}$ in the index space $\mathbf{J}^3$. In a parallel computing platform, if different iterations are executed at different processors, these input variables must be propagated or broadcast to different processors to facilitate the computation. This routine of input variables can be repre-

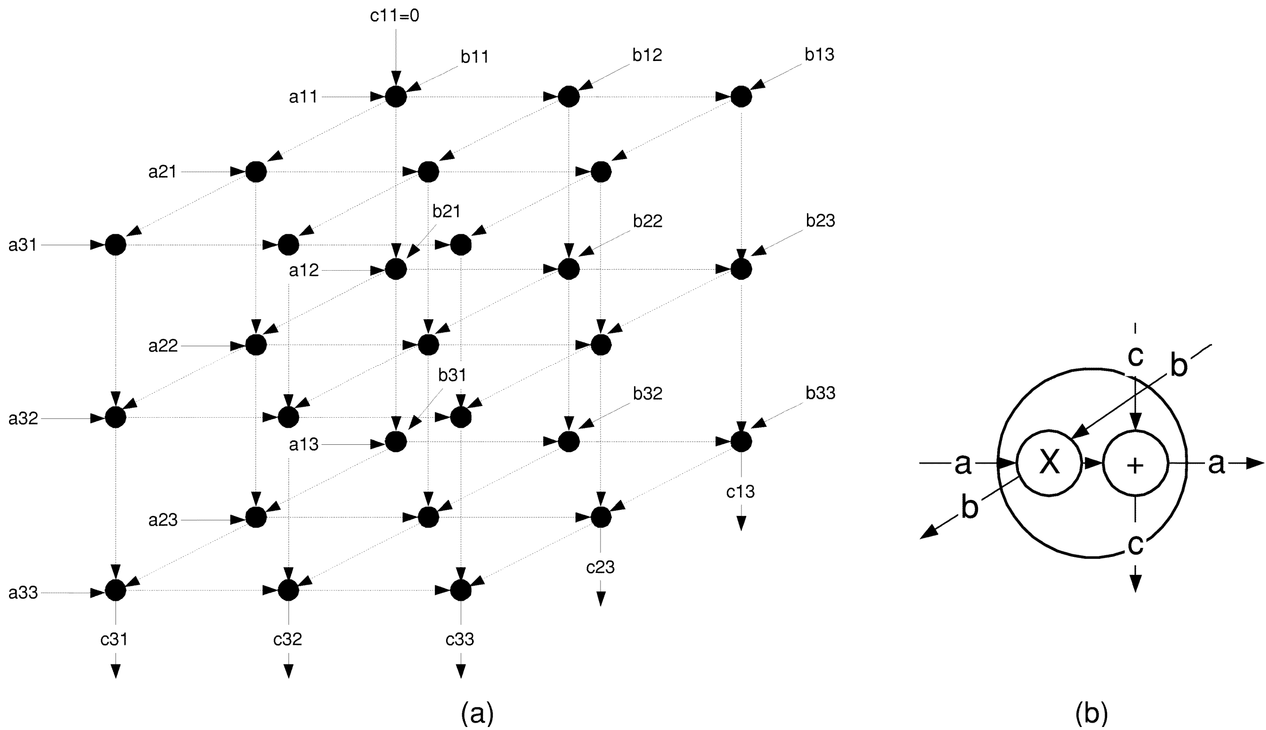

Fig. 1. The 3×3 matrix–matrix multiplication in a 3-D dependence graph (a) and its node (loop body) (b).

sented using intermediate variables to ensure that the single-assignment constraint is satisfied.

With the introduction of these intermediate variables, every variable associated with a particular iteration will have the full set of indices. For example, the matrix–matrix multiplication loop body can now be rewritten as follows.

Listing 2: Single-Assignment Matrix-Matrix Multiplication

Do $i = 1$ to M

Do $j = 1$ to N

Do $k = 1$ to k

$$a_3[i,j,k] = \begin{cases} a_3[i,j-1,k], & j > 0 \\ a[i,k], & j = 0 \end{cases}$$

$$b_3[i,j,k] = \begin{cases} b_3[i-1,j,k], & i > 0 \\ b[k,j], & i = 0 \end{cases}$$

$$c_3[i,j,k] = \begin{cases} c_3[i,j,k-1] + a_3[i,j,k] \times b_3[i,j,k], & k > 0 \\ 0, & k = 0 \end{cases}$$

$$c[i,j] = c_3[i,j,k], \quad k = K$$

EndDo k

EndDo j

EndDo i

In the above listing, a_3 and b_3 are the *transmittal variables* of a and b, respectively, and c_3 is the *computation variable* of c. We may now define an inter-iteration *dependence vector* as the set of index differences between the output of each iteration (on the left-hand side of each equation) and the input (on the right-hand side of the equations). For example, the dependence vector of variable a_3, b_3, and c_3 are $\vec{d}_{a_3} = (0,1,0)^t$, $\vec{d}_{b_3} = (1,0,0)^t$, and $\vec{d}_{c_3} = (0,0,1)^t$, respectively.

The plot of a graph representing index points corresponding to the algorithm in the index space $\mathbf{J}^3$ and the dependence vectors is called the *dependence graph* [4] (DG). The DG of Listing 2 is plotted in Fig. 1. In other words, DG (also known as the iteration space DG [11]) is a graphical representation of data dependencies among loop iterations of a nested Do loop. It consists of a set of nodes (vertices) and a set of edges. Each node corresponds to a loop index, $\vec{i} \in \mathbf{J}^n$, or the innermost loop body regardless of its complexity. Each directional edge represents either a *propagation* or a *computation* dependence vector.

A loop nest is called a set of *uniform recurrence equations* (URE) [12] if its dependence vectors are independent of the loop index $\vec{i}$. In other words, a UREs loop bounds are known constants before the execution of the algorithm. Almost all the data intensive nested loop formulated multimedia algorithms can be formulated as uniform recurrence equations.

In a URE formulated algorithm, each loop has a set of uniform dependence vectors, that can be described by a *dependence matrix* D_V

$$D_V = \begin{bmatrix} \vec{d}_b & \vec{d}_a & \vec{d}_c \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where $V = \{a, b, c\}$ is a set of variables. Due to the regular structure of n-D DG, the task of scheduling and assignment of individual index (loop body) to be executed on a particular processor at a particular clock cycle can be solved using algebraic projection. This projection is called *systolic* or *space–time* mapping. The systolic mapping is limited to nested loop algorithm with uniform recurrence equation. The *mapping matrix* T[4] consists of a *scheduling vector* $\vec{s}$ and a *processor allocation matrix* P. In this example, $\vec{s} = (i,j,k)^t = (0,0,1)^t$ and

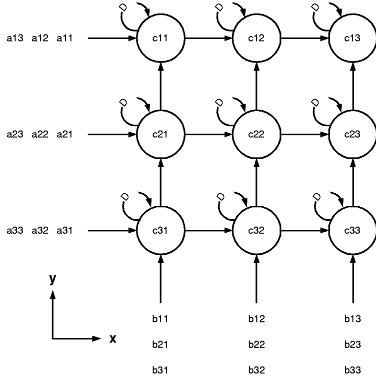

Fig. 2. Two-dimensional processor array for $N \times N$ matrix multiplication, $N = 3$ after the mapping (D indicates a register).

$P = [\vec{p}_1 \quad \vec{p}_2 \quad \cdots \quad \vec{p}_{n-1}]$ where $\vec{p}_m$ is an n-element column vector, $1 \leq m \leq n-1$. T must be a nonsingular matrix so that the mapping has no conflict such that $T\vec{p} \neq T\vec{q}$, where $\vec{p} \neq \vec{q}$, $\forall\, \vec{p}$, $\forall\, \vec{q} \in \mathbf{J}^n$

$$T = \begin{bmatrix} \vec{s}^t \\ P^t \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}.$$

The mapping of dependence matrix D_V results in a *delay–edge* matrix

$$TD_V = \begin{bmatrix} \vec{s}^t \\ P^t \end{bmatrix} D_V = \begin{bmatrix} r_b & r_a & r_c \\ \vec{e}_b & \vec{e}_a & \vec{e}_c \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}.$$

The scalar–valued delay or pipeline register r_v is associated with an edge (inteprocessor link) $\vec{e_v} = (x, y)^t$. For example, the vector $\vec{e}_c = (0, 0)^t$ indicates a self loop where $r_c = 1$ indicates one register element to store the intermediate results as shown in Fig. 2. It can be observed that the three-dimensional (3-D) DG has been projected along the k axis to a 2-D processor array and a 1-D execution schedule.

The existing systolic mapping methodology has a number of limitations due to an unreasonable assumption that data are always available. For instances, it makes the number I/O ports to the array grow as a function of problem size. A large number of I/O ports puts more pressure on the memory bandwidth to sustain such a high computation rate. Secondly, the notion of *memory hierarchy* has not been used. Memory hierarchy, consisting of register file, cache, and main memory, is very common in both embedded and general-purpose architecture. Specifically register file can store intermediate values for later usage. The fact in the past that transistors (logics) were relatively more expensive than the interconnects is opposite to the current situation particularly in FPGA. Based on these limitations, we will describe the solution in the context of multimillion gate FPGA architecture.

B. LUT-Clustered FPGA Architecture

In this paper, we assume that a modern multimillion gate FPGA is used as a reconfigurable coprocessor tailored to computation and data-intensive algorithms. The algorithm is described in the form of inter-iteration dependence graph [4]. The resulting processor array can exploit both loop–level or inter-iteration pipelining and parallelism in a multi-million gate FPGA architecture such as the Altera Apex II [6] and the Xilinx Virtex families [1], [5]. These FPGA architectures are LUT-clustered [13] in which each logic cluster is composed of a number of LUT's where intracluster interconnect is faster than intercluster (general-purpose) routing. Hence, local/pielined communication is preferred. After the mapping, each processor will consist of word-level cores provided by FPGA manufacturers such as Xilinx's Core Generator system. These cores are relationally placed and routed to exploit the faster local or intracluster interconnects.

Do $i_1 = l_1$ to u_1
....
  Do $i_n = l_n$ to u_n
    Input/Propagation Statements

$$v_j^n[\vec{i}] = \begin{cases} v_j[\mathcal{G}_{v_j}(\vec{i})], & \vec{i} \in \mathbf{I}_{v_j}^I \\ v_j^n[\vec{i} - \vec{d}_{v_j}], & \vec{i} \in \mathbf{I}_{v_j}^P \end{cases}$$

    ....
    Computation/Initialization and Statements

$$v_k^n[\vec{i}] = \begin{cases} v_k^I, & \vec{i} \in \mathbf{I}_{v_k}^N \\ \mathcal{F}_{v_k}(v_k^n[\vec{i} - \vec{d}_{v_k}], v_j^n[\vec{i}], \ldots), & \vec{i} \in \mathbf{I}_{v_k}^C \end{cases}$$

    ....
    Output Statements

$$v_k[\mathcal{G}_{v_k}(\vec{i})] = v_k^n[\vec{i}], \quad \vec{i} \in \mathbf{I}_{v_k}^O$$

    ....
  EndDo i_n
....
EndDo i_1

Fig. 3. An n-level single-assignment nested Do-loop algorithm.

Particularly, each cluster or configurable logic block is composed of four set of LUT's in the Virtex architecture [5]. Each LUT, associated with one flip-flop (FF) and carry logic, can function as a 4-input/1-output Boolean function generator, a synchronous programmable-length up to 16×1-bit shift register, first-in–first-out (FIFO) buffer, or a synchronous 16-entry $\times$ 1-bit ROM/RAM. Altogether, two LUT's can be programmed as either a 32-entry $\times$ 1-bit single-port ROM/RAM or a 16-entry $\times$ 1-bit dual-port ROM/RAM. Due to insufficient capacity and efficiency of an LUT, a number of 4096-bit BlockRAM's are integrated on chip. Each BlockRAM can be programmed to be of particular width and depth such as an 8-bit $\times$ 512-entry, 16-bit $\times$ 256-entry, single-/dual-port RAM. In addition, the Virtex II [1] features programmable 18-bit by 18-bit multiplies for DSP applications.

III. MODG and Its Design Methodology

This section discusses the notions of multiple-order dependence graph and its space–time mapping methodology. The mapping can be cast as an optimization searching for the optimal solution evaluated by a number of objective functions (performances) subject to the design and architectural constraints (costs). Because the complexity of MODG is of high-order polynomial, heuristic search strategies are necessary to seek the near optimal solution.

A. n-Dimensional Multiple-Order Dependence Graph (n-D MODG)

A generalized model of single-assignment nested Do loop is shown in Fig. 3 where l_m and u_m are index variable i_m's lower and upper bounds and $m = 1, 2, \ldots, n$. Three types of statements are included in the innermost loop body: the *Input/propa-*

gation statement, the *Computation/initialization statement*, and the *Output statement*. Each type of statement corresponds to a particular type of the variables.

- *Transmittal variable*—An input variable v_j is first assigned to the transmittal variable $v_j^n[\vec{i}]$ at iteration indices $\vec{i} \in \mathbf{I}_{v_j}^I$, and then propagated along a *propagation* dependence vector $\vec{d}_{v_j}$ for $\vec{i} \in \mathbf{I}_{v_j}^P$. Specifically, $\mathbf{I}_{v_j}^P \cap \mathbf{I}_{v_j}^I = \phi$ (empty set).

  Referring to Listing 2, we have $a_3[\vec{i}]$ and $b_3[\vec{i}]$ as transmittal variables. The initialization spaces of a_3 and b_3 are $\mathbf{I}_a^I = \{(i,j,k)^t \mid 1 \leq i \leq M,\ 1 \leq k \leq K,\ j = 0\}$ and $\mathbf{I}_b^I = \{(i,j,k)^t \mid \leq j \leq N,\ 1 \leq k \leq K,\ i = 0\}$, respectively.
- *Computation variable*—The computation variable $v_k^n[\vec{i}]$ will be initialized to some constants at an index $\vec{i} \in \mathbf{I}_{v_k}^N$, and then will be assigned to intermediate values at other iterations $\vec{i} \in \mathbf{I}_{v_k}^C$ according to the recurrence function $\mathcal{F}_{v_k}$. In Listing 2, we have $\mathbf{I}_c^N = \{(i,j,0)^t \mid 1 \leq i \leq M,\ 1 \leq j \leq N\}$ and $\mathbf{I}_c^C = \{(i,j,k)^t \mid 1 \leq i \leq M,\ 1 \leq j \leq N,\ 1 \leq k \leq K\}$ where $\mathcal{F}_c$ is the summation operation.
- *Output variable*—These are results that will be stored back to memories outside the array of processors. Their values will be assigned at a set of output index points $\mathbf{I}_{v_k}^O$. From Listing 2, $c[i,j]$ is the output variable and $\mathbf{I}_c^O = \{(i,j,k)^t \mid k = K\}$. The indexing function of output variable c is $\mathcal{G}_c(i,j,k) = (i,j)$ if $k = K$.

We define an MODG as a set of MODG nodes where each node is associated with a number of information fields including edges as dependence vectors as follows.

Definition 1: n-Dimensional Multiple-Order Depenence Graph, K_n.

An n-D MODG, K_n, is a set of MODG nodes. Each node, $k_n \in K_n$, is a collection of the following fields of information, n-D index, multiple-order dependence vectors set, input data set, output data set, and terminal flag set. □

Each n-D MODG node, $k_n \in K_n$, containing a tuple-of information fields is equivalent to

$$k_n \triangleq (\vec{i}, D_V, I_V, O_V, F_V)$$

where V is a set of variables. Similar to C++, we use "." to access a particular field of information. For example, $k_n.\vec{i}$ denotes an n-D index field, $k_n.\vec{i} = (i_1, i_2, \ldots, i_n)^t \in \mathbf{J}^n$. $k_n.I_V$ is a set of *input data*, where $k_n.I_V = \{k_n.I_v \mid \forall\, v \in V\}$. Likewise, $k_n.O_V$ is a set of *output data*, $k_n.O_V = \{k_n.O_v \mid \forall\, v \in V\}$. Next, a set of Boolean *terminal flag* specifies whether the node is either an input or an output node for each variable, $v \in V$, $k_n.F_V = \{k_n.F_v \mid \forall\, v \in V\}$ where $F_v \in \{0,1\}$ and $F_v = 1$ denotes the true logic value. Finally, $k_n.D_V$ is a set of dependence vectors associated with variable set V,

$$k_n.D_V = \{k_n.D_v \mid \forall\, v \in V\}$$

where $k_n.D_v$ denotes a set of all feasible dependence vectors of variable v. $k_n.D_v$ can be obtained depending upon the following variable categories.

1) Transmittal Variable: An input variable $v[\vec{g}]$ is reused, propagated and called the *transmittal* variable according to a single dependence vector $\vec{d}_v$ as shown in Fig. 3, i.e., $k_n.D_v = \vec{d}_v,\ \forall\, k_n.\vec{i} \in \mathbf{I}_v^P \cup \mathbf{I}_v^I$. This dependence vector is determined during the algorithm formulation and restricted to an adjacent or neighboring index. Hence, the propagation along a particular direction based on heuristic is imposed such that the DG becomes localized.

Actually, each $v[\vec{g}]$ can be reused several times during the course of execution. Ordering should not be imposed as long as it is correctly delivered. To represent all the possible dependence vectors among all reusing MODG nodes, we define a *broadcast index set* as a set of n-D indices associated with each variable instance $v[\vec{g}]$:

$$\mathbf{I}_v^{\vec{g}} = \left\{k_n.\vec{i} \mid \mathcal{H}_v(k_n.\vec{i}) = \vec{g},\ \forall\, k_n \in K_n\right\} \tag{2}$$

where $\mathcal{H}_v(\vec{i}) : \vec{i} \longrightarrow \vec{g},\ \exists\, \vec{i} \in \mathbf{I}_v^P \cup \mathbf{I}_v^I$. The indexing function $\mathcal{H}_v$ is similar to $\mathcal{G}_v(\vec{i})$ shown in Fig. 3. Therefore, at any index point, each variable is associated with a set of multiple-order dependence vectors

$$k_n.D_v = \left\{\vec{i}_1 - k_n.\vec{i} \mid k_n.\vec{i} \neq \vec{i}_1,\ \forall\, \vec{i}_1 \in \mathbf{I}_v^{\vec{g}}\right\}$$

including the localized dependence vector $\vec{d}_v$.

2) Computation Variable: In a nested Do loop algorithm, the computation variable v is an output of a particular recurrence function $\mathcal{F}_v$. This function can be as simple as add, multiply, minimum, maximum, and the like. If these operators follow both commutative and associative laws such that

$$\mathcal{F}_v(a,b) = \mathcal{F}_v(b,a)$$

and

$$\begin{aligned}\mathcal{F}_v(\mathcal{F}_v(a,b),c) &= \mathcal{F}_v(a,\mathcal{F}_v(b,c))\\ &= \mathcal{F}_v(b,\mathcal{F}_v(a,c))\end{aligned}$$

respectively, they are called *multiple-order* operators. Otherwise, functions or operators that do not follow both laws are considered *in-order*. Traditionally, each instance of computation variable v is computed and propagated along a local dependence vector $\vec{d}_v$, i.e.,

$$k_n.D_v = \vec{d}_v, \quad \forall\, k_n.\vec{i} \in \mathbf{I}_v^N \cup \mathbf{I}_v^C$$

as if it were an in-order operation.

Given a multiple-order operation, $v[\vec{g}]$ can be computed in many different orders of execution. Ordering should be relaxed provided that it is semantically correct and its numerical condition is satisfied. To represent all the possible dependence vectors among all computing MODG nodes, we define an *multiple-order index set* as a set of MODG nodes of a variable $v[\vec{g}]$ to be

$$\mathbf{I}_v^{\vec{g}} = \left\{k_n.\vec{i} \mid \mathcal{H}_v(k_n.\vec{i}) = \vec{g},\ \forall\, k_n \in K_n\right\} \tag{3}$$

where $\mathcal{H}_v(\vec{i}) : \vec{i} \longrightarrow \vec{g},\ \exists\, \vec{i} \in \mathbf{I}_v^N \cup \mathbf{I}_v^C$. Therefore, a set of multiple-order dependence vectors of variable v can be obtained as

$$k_n.D_v = \left\{\vec{i}_1 - k_n.\vec{i} \mid k_n.\vec{i} \neq \vec{i}_1,\ \forall\, \vec{i}_1 \in \mathbf{I}_v^{\vec{g}}\right\}.$$

Referring to the matrix product example, we can identify the indexing functions $\mathcal{G}$ of a, b, and c as $\mathcal{G}_a(i,j,k) = (i,k)$ if $j = 0$, $\mathcal{G}_b(i,j,k) = (k,j)$ if $i = 0$, and $\mathcal{G}_c(i,j,k) = (i,j)$ if $k = K$, respectively. On the other hand, the indexing function $\mathcal{H}$ of both input variables, a and b are $\mathcal{H}_a(i,j,k) = (i,k)$ and $\mathcal{H}_b(i,j,k) = (k,j)$, respectively. Since the summation follows both associative and commutative laws, it is a multiple-order operation and its indexing function $\mathcal{H}_c(i,j,k) = (i,j)$.

B. One-Dimensional Space–Time Mapping

We employ the 1-D space–time mapping of n-D MODG to a 1-D array of PEs. Although, it was originally proposed by Lee and Kedem [14], it is different from theirs in the following aspects:

- There is no restriction on the ratio of delay and edge length.
- Input and output ports are not necessary at either end of the array.

The index space $\mathbf{J}^n$ defined in (1) is mapped (both assigned and scheduled) subject to the dependencies described by dependence vectors. The obtained schedule assumes synchronous operations in which the whole loop body is executed in one clock cycle latency. The advantages of 1-D array are threefold. First, the final 1-D array can be adjusted to fit the FPGA easily. Second, the input–output port is already on the array boundary. Third, the array is easy to rearrange to a 2-D array. The 1-D mapping matrix is defined below.

Definition 2: 1-D space–time mapping matrix, T_1

The 1-D space–time mapping matrix T_1 consists of a scheduling vector $\vec{s}$ and an allocation vector $\vec{P}$ as

$$T_1 = \begin{bmatrix} \vec{s}^t \\ \vec{P}^t \end{bmatrix} = \begin{bmatrix} s_1 & s_2 & \dots & s_n \\ p_1 & p_2 & \dots & p_n \end{bmatrix} \tag{4}$$

where $T_1 \in \mathbf{Z}^{2\times n}$ and $s_i, p_i \in \mathbf{Z}$. Furthermore, $\vec{s}$ and $\vec{P}$ must be linearly independent so that $\mathrm{Rank}(T_1) = 2$. □

Prior to applying T_1 to n-D MODG, let us define the 1-D PE array representation to accommodate the mapping.

Definition 3: One-D Processing Element Array, K_1

A 1-D PE array is a set of nodes or PEs in which each node is resulting from a projection of a number of n-D MODG nodes. Each PE encapsulates the PE number and a clock schedule of feasible delay–edge pairs, input–output data, and terminal flags. □

In other words, each PE is a tuple of $(\mathbf{J}^1, t(\vec{i}), R_V, E_V, r_V, \ e_V, I_V, O_V, F_V)$. Input, output, and terminal sets are assigned to a PE and ordered according to the synchronous schedule $k_1.t(\vec{i})$. Hence, it is equivalent to a set of n-D MODG nodes:

$$k_1 \triangleq \left\{ k_n \mid k_1.\mathbf{J}^1 = \vec{P}^t(k_n.\vec{i}), \ \forall\, k_n \in K_n \right\} \tag{5}$$

assigned (allocated) to the PE number $k_1.\mathbf{J}^1$, $PE_{\min} \leq k_1.\mathbf{J}^1 \leq PE_{\max}$, where $PE_{\min} = \min(\vec{P}^t\vec{q} \mid \vec{q} \in \mathbf{J}^n)$ and $PE_{\max} = \max(\vec{P}^t\vec{q} \mid \vec{q} \in \mathbf{J}^n)$. The clock schedule associated with each $k_1 \in K_1$ is obtained from

$$k_1.t(\vec{i}) = \left\{ \vec{s}^t(k_n.\vec{i}) \mid k_1.\mathbf{J}^1 = \vec{P}^t(k_n.\vec{i}), \quad \forall\, k_n \in K_n \right\}.$$

At a particular cycle $\tau \in k_1.t(\vec{i})$ of PE number $k_1.\mathbf{J}^1 = \vec{P}^t(k_n.\vec{i})$, its feasible edges and delay sets are $k_1.E_V[\tau] = \{k_1.E_v[\tau] \mid \forall\, v \in V\}$ and $k_1.R_V[\tau] = \{k_1.R_v[\tau] \mid \forall\, v \in V\}$. Each variable v's $k_1.E_v[\tau]$–$_1.R_v[\tau]$ pair results from the space–time mapping of the multiple-order dependence vectors of an MODG node, such that

$$\begin{bmatrix} k_1.R_v[\tau] \\ k_1.E_v[\tau] \end{bmatrix} = T_1(k_n.D_v) = \begin{bmatrix} \vec{s}^t \\ \vec{P}^t \end{bmatrix} k_n.D_v.$$

In addition, the terminal flag set at this cycle τ becomes $k_1.F_V[\tau] = \{k_1.F_v[\tau] \mid \forall\, v \in V\}$ where $k_1.F_v[\tau] = k_n.F_v$. Finally, the final edge–delay pair, $k_1.e_v[\tau]$–$k_1.r_v[\tau]$, $k_1.e_v[\tau] \in k_1.E_v[\tau]$, $k_1.r_v[\tau] \in k_1.R_v[\tau]$, are chosen appropriately subject to the design objectives (performance) and design constraints (cost) in Section III-B1 and B2.

1) Mapping Objective Functions: Performance: The 1-D PE array can be evaluated based on the following performance characteristics. As objective functions, the number of cycles, the number of PEs, and the utilization are analogous to the execution time, the area, and the efficiency in hardware, respectively. Besides, physical input–output pins and memory bandwidth are of increased importance as the speed gap between logic and memory, especially dynamic RAM gets wider. Additionally, these objective functions can be used to constrain the search as well.

- Number of cycles, N_{cycle}: for FPGA implementation, the total parallel execution time, t_{total}, can be computed by

  $$t_{\mathrm{total}} = t_{\mathrm{cycle}} \times N_{\mathrm{cycle}}$$

  where t_{cycle} is the PEs longest critical propagation delay, which depends on the PE architecture and the implementation technology, and N_{cycle} is the number of clock cycles [4] given by

  $$N_{\mathrm{cycle}} = \max_{\vec{p},\vec{q} \in \mathbf{J}^n} \left\{ \vec{s}^t(\vec{p} - \vec{q}) \right\} + 1. \tag{6}$$

  In other words, it is the number of cuts by the equitemporal hyperplane [15] perpendicular to the scheduling vector $\vec{s}$. Although, N_{cycle} seems to depend on the scheduling vector $\vec{s}$ only, the question on how many PEs are utilized and how the data are delivered to the right PE still remains.
- Number of PEs, N_{PE}: in the 1-D or linear array mapping, the number of PEs, N_{PE} can be expressed as

  $$N_{\mathrm{PE}} = \mathrm{PE}_{\max} - \mathrm{PE}_{\min} + 1. \tag{7}$$

  In other words, N_{PE} is the number of distinct projections of the index space $\mathbf{J}^n$ on the vector $\vec{P}$.
- Utilization, U: the maximum PE array utilization is the maximum ratio of active PEs and the number of PEs, N_{PE}. It is obtained by

  $$U_{\max} = \frac{\max_\tau \left| \{k_1 \mid \tau \in k_1.t(\vec{i})\} \right|}{N_{\mathrm{PE}}} \tag{8}$$

where $|a|$ denotes the cardinality or size of set a. On the other hand, the average utilization is the ratio of the number of MODG nodes and the $N_{\rm cycle} - N_{\rm PE}$ product

$$U_{\rm avg} = \frac{|K_n|}{N_{\rm PE} \times N_{\rm cycle}}. \tag{9}$$

- Number of input–output ports, *#IO*: due to limited number of physical I/O pins of a given target architecture, it is important to minimize the number of I/O ports. Based on our formulation, the set of PEs performing I/O of either input or output variable v at cycle τ is

$$\mathrm{IO}_v[\tau] = \left\{ k_1 \mid k_1.F_v[\tau] = 1, \quad \tau \in k_1.t(\vec{i}), \quad \forall\, k_1 \in K_1 \right\}. \tag{10}$$

Therefore, the number of variable v's I/O ports is given by

$$\#\mathrm{IO}_v = \max_{\tau} |\mathrm{IO}_v[\tau]| \tag{11}$$

where $|\mathrm{IO}_v[\tau]|$ denotes the size of $\mathrm{IO}_v[\tau]$.

- Memory bandwidth: the memory bandwidth associated with variable v, B_v, is the number of input–output instances via a particular input–output port per unit time. In this case, the unit time is a clock cycle. From (10), B_v is equivalent to the total number of input–output occurrences averaged over $N_{\rm cycle}$,

$$B_v = \frac{\sum\limits_{\tau \in k_1.t(\vec{i}),\ \forall\, k_1 \in K_1} |\mathrm{IO}_v[\tau]|}{N_{\rm cycle}}. \tag{12}$$

2) Design Constraints: The 1-D space–time mapping is basically a search of scheduling and allocation vectors yielding the best PE arrays according to certain objective functions and subject to the following design constraints within bounded search space. The mapping process is targeted as a computer-aided design tool. On the contrary to traditional mapping, it lets constraints decide the feasibility, and connectivity, as well as the final architecture. In general, the mapping conflicts prune out the infeasible solutions. The propagation and multiple-order constraints are responsible for assigning input and output ports. In addition, the causality constraint will validate the solution whether the producer–consumer concept is violated.

- Mapping conflict: due to insufficient rank of T_1, a mapping conflict occurs when two MODG nodes are assigned to the same PE and scheduled at the same cycle, i.e., $T_1\vec{p} = T_1\vec{q}$, $\vec{p} \neq \vec{q}$, $\vec{p}, \vec{q} \in \mathbf{J}^n$. Unlike [14] and [16], no *communication conflict* is introduced by our method. In order to efficiently detect the conflicts, a 2-D integer array A of size $N_{\rm PE} \times N_{\rm cycle}$ is used such that

$$\text{No conflicts} \overset{\text{iff}}{\longleftrightarrow} A[T_1\vec{q}] \leq 1, \quad \forall\, \vec{q} \in \mathbf{J}^n \tag{13}$$

where

$$A[T_1\vec{q}] = \begin{cases} 0, & \text{initialization} \\ A[T_1\vec{q}] + 1, & \text{otherwise.} \end{cases}$$

Each array element $A[T_1\vec{q}]$ is increased by one if the previous value is zero from the initialization phase. Otherwise, the computation conflict is detected. The worst-case time complexity of this method is $O(N^n)$ where $N = \max\{u_i - l_i + 1,\ i = 1, 2, \ldots, n\}$ while that of [14] is $O(N^{2n})$.

- Propagation constraint: despite the fact that a systolic array can achieve high computation throughput, one of the reasons that it has not been successful is partly due to its high demand of memory bandwidth. As a result, every input variable should be traced and reused as many times as possible to eliminate redundant memory fetches. Thus, we can eventually save the bandwidth and the number of I/O ports. Hence, the input port should be determined after the mapping process rather than from a predetermined terminal flag assigned by a human designer. The input port is assigned to a k_1 PE where the first instance of data appears and propagated to the next instance. Permissible edge-delay pairs are obtained by eliminating edges that point backward in the time domain.

The mapping process starts by initializing the terminal vectors $k_n.F_v = 0$, $\forall\, k_n \in K_n$. After space–time mapping, the terminal flag will be true at the first appearance in the broadcast index set $\mathbf{I}_v^{\vec{g}}$ in (2) of each input instance $v[\vec{g}]$. Thus, the terminal flag is assigned by

$$k_1.F_v[\tau] = \begin{cases} 1, & \tau = \min(k_1.t(\vec{i})) \\ 0, & \tau > \min(k_1.t(\vec{i})) \end{cases} \quad \forall\, k_n.\vec{i} \in \mathbf{I}_v^{\vec{g}}$$

- Multiple-order operation constraint: likewise, given an output variable v after mapping, the terminal flag will be true at the last appearance in the multiple-order index set of each instance $v[\vec{g}]$, $\mathbf{I}_v^{\vec{g}}$ in (3). Thus, the terminal flag is assigned by

$$k_1.F_v[\tau] = \begin{cases} 1, & \tau = \max(k_1.t(\vec{i})) \\ 0, & \tau < \max(k_1.t(\vec{i})) \end{cases} \quad \forall\, k_n.\vec{i} \in \mathbf{I}_v^{\vec{g}}$$

- Causality constraint: the purpose of this causality constraint is to prevent consuming intermediate data before it is fully produced. For every instance of intermediate producer variable $v_p[\vec{f}]$ and instance of consumer variable $v_c[\vec{g}]$, the following inequality must be satisfied

$$\max\left(\vec{s}^{\,t}\vec{p} \mid \forall\, \vec{p} \in \mathbf{I}_{v_p}^{\vec{f}}\right) < \min\left(\vec{s}^{\,t}\vec{q} \mid \forall\, \vec{q} \in \mathbf{I}_{v_c}^{\vec{g}}\right). \tag{14}$$

C. FPGA Architectural Constraints

In addition to the application of MODG to very large scale integration (VLSI) implementation, our current target architecture is the SRAM-based FPGAs such as those of Xilinx Virtex family [1], [5]. Furthermore, our methodology can be slightly modified to match other targets such as Altera APEX II and Lucent Orca FPGAs because they have similar (almost the same) features as those of Xilinx Virtex family.

1) Programmable-Length Shift Register or FIFOs: It is a magnificent feature of the LUT of an SRAM-based FPGA and can be exploited by our design methodology to propagate input data that are unknown at design time. Each LUT can be programmed as 1-bit shift register of $N \leq 16$ length. Hence, an 8-bit N-cycle delay consumes only eight LUTs if $N \leq 16$ rather than $8N$ flip-flops (FFs) or equivalently $8N$ LUTs in FPGA because one FF is provided per LUT [5]. For an input variable v

at PE $k_1 \in K_1$, the final edge–delay pair at cycle $\tau \in k_1.t(\vec{i})$ is subject to the following

$$\begin{aligned} k_1.r_v[\tau] &> 0 \\ k_1.e_v[\tau] &\neq 0 \\ |k_1.e_v[\tau]|(k_1.r_v[\tau]) &= \min(k_1.R_V[\tau]|k_1.E_v[\tau]|). \end{aligned} \quad (15)$$

2) ROM/RAM: Besides working as the 4-bit input/1-bit output Boolean function generator, an LUT can be utilized as a single- or dual-port ROM/RAM. According to [5], two LUTs in the same slice can be configured as a single-port 32-entry $\times$1-bit ROM/RAM or a dual-port 16-entry $\times$1-bit ROM/RAM. Unlike a hardwired ASIC design, FPGA can be reconfigured with any initialization values embedded in the configuration bit stream. ROM and RAM are suitable for input and output variables, respectively. This is due to the fact that RAM can be updated while ROM cannot. However, the control mechanism is a little more complex.

For a constant and known input variable v at PE $k_1 \in K_1$, the delay and edge sets at cycle $\tau \in k_1.t(\vec{i})$ are subject to the following equations to utilize the distributed ROM

$$\begin{aligned} k_1.R_v[\tau] &= \{mN \mid N > 0,\ m \neq 0,\ m \in \mathbf{Z}\} \\ k_1.E_v[\tau] &= \{0\}. \end{aligned} \quad (16)$$

The constraints are similar for an output variable v to use distributed RAM.

3) Three-State Buffer: To save the multiplexer which actually consumes LUTs, an output bus with three-state buffers is an efficient alternative to a high fan-in multiplexer. The output data is placed on the bus at different cycles without bus collision. This design strategy consumes virtually zero LUT provided that built-in three-state buffers are inherently available from the occupied logic LUTs. A bus-based design for an output variable v is subject to the following constraints

$$\begin{aligned} \#\mathrm{IO}_v[\tau] &= 1, \quad \forall\, \tau \in k_1.t(\vec{i}),\ \forall\, k_1 \in K_1 \\ \#\mathrm{IO}_v &> 1. \end{aligned} \quad (17)$$

D. Heuristic Search

We denote $N = \max(u_i - l_i,\ 1 \leq i \leq n)$ in order to determine the space–time complexity of an MODG. Hence, in a particular n-level nested Do loop algorithm, the number of MODG nodes is $O(N^n)$. In addition, the variable *reuse factor* is a polynomial function of N, e.g., $O(N^m),\ 1 \leq m \leq n$. Therefore, the worst-case number of dependence vectors is $O(N^n \times |V|N^m) = O(|V|N^{n+m})$ where $|V|$ is the number of variables. Regarding the matrix multiplication example, the maximum data reuse factor can be express as $O(N)$ and $|V| = 3$. Consequently, the total number of dependence vectors is $O(3N^4)$.

With an exhaustive search for T_1 in (4), the number of search iterations to be conducted is $O(M^{2n})$, where $u_i \leq M \leq \prod_{i=1}^{n}(u_i)$ and $1 \leq i \leq n$. Each search iteration has to manipulate an MODG of $O(|V|N^{n+m})$ complexity. As a result, the complexity of an exhaustive search can reach $O(|V|N^{n+m} \times M^{2n}) = O(|V|N^{n+m}M^{2n})$. Due to the insufficient rank of T_1, the solutions are sparsely scattered which makes the exhaustive search inefficient and time-consuming.

```
Do v = 0 to N_v − 1
Do h = 0 to N_h − 1
    MV[h, v] = (0, 0);
    D_min[h, v] = ∞;
    Do m = 0 to 2p
    Do n = 0 to 2p
        MAD[m, n] = 0;
        Do i = 0 to N − 1
        Do j = 0 to N − 1
            MAD[m, n] = MAD[m, n] + |x[hN + i, vN + j]
            −y[hN + i + m − p, vN + j + n − p]|;
        EndDo j, i
        If D_min[h, v] > MAD[m, n] Then
            D_min[h, v] = MAD[m, n];
            MV[h, v] = [m − p, n − p];
        EndIf
EndDo n, m, h, v
```

Fig. 4. Six-level nested Do loop FSBM motion estimation algorithm.

In order to speedup the searching process, a heuristic search strategy is necessary. A much smaller search set is defined as

$$S = \left\{0, \pm 1, \prod (u_i u_j), \prod (u_i u_j u_k),\ 1 \leq i, j, k \leq n\right\}.$$

From the definition of T_1 in (4), the complexity of heuristic search is reduced to $O(|V|N^{n+m}|S|^{2n})$, where $\lceil |S| \rceil = n^3 + n^2 + 3 \ll M$. As a matter of fact, each search iteration is independent of one another, a number of different iterations can be distributed in a high throughput network of workstations, e.g., Condor [17] and compared the performance evaluations at the end. Finally, the problem size should be scaled down to reduce the computational complexity.

IV. Motion Estimation: An Example

Due to limited space, we can only demonstrate the use of our methodology using, block matching motion estimation.

It has been widely accepted that the full search block matching (FSBM) motion estimation is one of the most time-consuming task for digital video encoding. Several hardwired ASICs have been manufactured including the STi3220 motion estimation processor [18]. As a major step toward saving memory bandwidth, Yeo and Hu [19] proposed the formulation of a six-level nested Do-loop algorithm to represent a single-frame, multiple-block motion estimation. A typical video frame consists of $N_h \times N_v$ blocks of pixels where N_h is the number of blocks in each row and N_v is the number of rows in each frame. A motion vector, $[m, n]$ of an $N \times N$ block of pixels, which yields the minimum *mean-absolute distortion* (MAD) between current block and the $(2p+1)^2$ candidates in the search area, can be obtained from

$$\mathrm{MV} = \arg\{\min \mathrm{MAD}[m, n]\} - [p, p]$$

where $0 \leq m, n \leq 2p$, p is the *search range* in number of pixels and usually less than or equal to block size N. The corresponding MAD of a vector $[m, n]$ is obtained by

$$\mathrm{MAD}[m, n] = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} |x[i, j] - y[i + m - p, j + n - p]|$$

where $x[i, j]$ and $y[i, j]$ are the luminance pixels of the current and previous frames, respectively. A six-level nested Do loop MAD-based FSBM algorithm is shown in Fig. 4, where $D_{\min}$ is

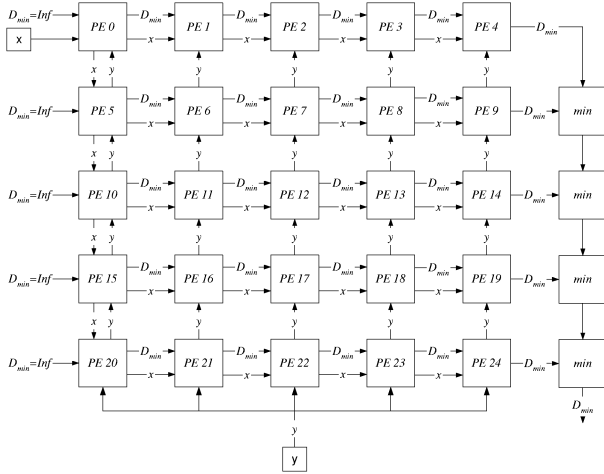

Fig. 5. Fully pipelined array of full search block matching motion estimation ($N_h = Nv = 4,\ N = 4,\ p = N/2 = 2$).

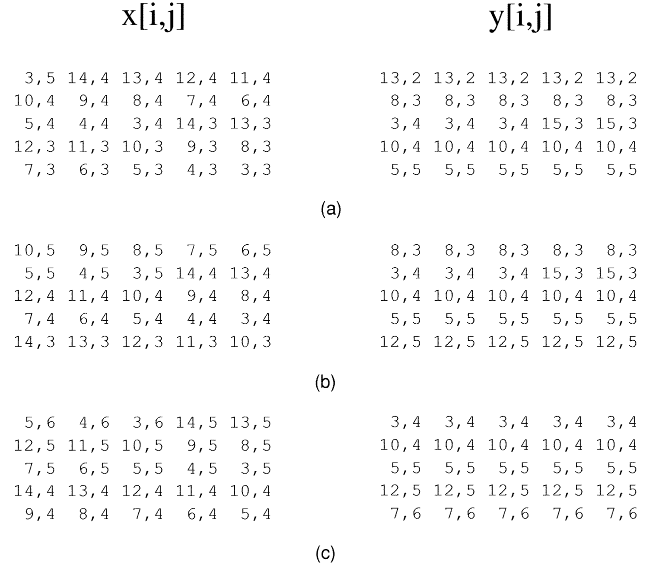

Fig. 6. A fully pipelined schedule of full search block matching motion estimation: (a) at cycle $\tau = 35$, (b) $\tau = 36$, and (c) $\tau = 37$ ($N_h = Nv = 4,\ N = 4,\ p = N/2 = 2$).

the *minimum distortion* measured using MAD. It can be noticed that MAD and min are of multiple-order operators. In addition, it has been observed in [20] that there are many possible data propagation patterns. Therefore, the proposed MODG is applicable.

To make the problem tractable for analysis, we scale the problem size down to $N_v = N_h = 4$, $N = 4$, and $p = N/2 = 2$. As indicated earlier in Section III-D, T_1 must be searched to obtain the final one-dimensional (1-D) array that satisfies numerous constraints. In this case, the search was limited to the line-by-line pixel scanning order. This pixel scan reduces the buffer size to just one line of pixels rather than the whole frame in block scanning mode as previously used. Our objective is to equalize the bandwidth of current frame pixel x and previous frame pixel y while minimizing N_{cycle}

$$T_1 = \begin{bmatrix} N & N_h N^2 & 1 & 2p+1 & 1 & N^2 \\ 0 & 0 & 1 & 2p+1 & 0 & 0 \end{bmatrix}.$$

The permissible edges-delay pair of each PE, $k_1.e_v[\tau] - k_1.r_v[\tau]$, $v \in \{x, y, D_{\min}\}$, was chosen in such a way to minimize the total LUT consumption.

Each PE in Fig. 5 corresponds to a motion vector $[m, n]$ with the relationship to PE number $k_1.\mathbf{J}^1 = nN + m$ Hence, the array must be composed of $(2p+1)^2$ PEs. The candidate MAD is accumulated from for consecutive N cycles and stored for $N_h N$ cycles while waiting for its next period of computation. N_h M-bit MADs are stored in a distributed dual-port RAM occupying $2M$ LUTs for smaller video frame size ($N_h \leq 16$). In a bigger frame size ($16 \ll N_h \leq \lfloor 4096/M \rfloor$), the dual-port BlockRAM [5] should be used instead. The $x[i,j]$ pixels, as shown in Fig. 6, are fetched in line scanning mode. This pattern can efficiently utilize data cache's both spatial and temporal locality. The $y[i,j]$ pixels are reused to the minimum memory bandwidth required. The array exploits the use of LUTs as delay elements or first-in first-out (FIFOs). This leads to a more compact layout where a 16-bit FIFOs occupies the space of only one LUT instead of 16 LUTs if it is designed using 16 FFs. Tables I and II summarize and compare the attained motion estimation processor array with other FPGA implementations.

TABLE I
BLOCK MATCHING MOTION ESTIMATION PE'S DATAPATH AREA COMPLEXITY (LUTS) ESTIMATED FROM THE XILINX CORE GENERATOR, ($N_h = Nv = 3,\ N = 4,\ p = N/2 = 2$)

Performance	Using LUT	Using FF
8-bit $\|x - y\|$	24	24
16-bit Adder	16	16
16-bit Comparison	17	17
8-bit x's Reg.	8	16
8-bit y's Reg.	8	56
16-bit D_{min}'s RAM	32	48
Total Area (LUTs)	105	177

TABLE II
PERFORMANCE COMPARISON FOR 2-D FSBM ME ARRAYS, $N = 4$, $p = N/2 = 2$

References	[21] AB2	[22] Type 1	[19]	[20]	*Ours*
Scan Mode	Block	Block	Block	Block	*Line*
Range	-2/2	-1/1	-2/1	-2/2	*-2/2*
N_{PE}	16	16	16	25	*25*
Cycles/Frame	656	656	272	256	*256*
LUTs	1,440	1,904	1,024	2,289	2,625
LUTs/PE	90	119	64	92	*105*
B_y/B_x	10	2	2	2	*1*
Fan-Out	0	100	16	8	*5*
#IO (pins)	80	40	40	40	*32*

V. CONCLUSION

A novel multiple (execution)-order dependence graph (MODG) and its FPGA mapping constraints are proposed for run-time configurable systolic arrays. As concise representations of several fundamental nested loop algorithms, MODG can be exploited to obtain array structures that match well to the modern SRAM-based FPGA architectures such as the Xilinx Virtex II and so on. Therefore, the obtained array is close to the globally optimal solution in many aspects: minimal execution time, reasonable LUT consumption, low I/O pins, high-hardware utilization, and eventually low-power consumption. In conjunction with either the Xilinx Floorplaner or the custom-made tool, the implementation of the obtainable

systolic arrays can achieve high operating clock frequency and efficient LUT utilization. As it was reported in the development of a 117-MHz systolic FIR filter on FPGA [23], a custom-made tool [24] was used to aid the placement and routing process.

Despite the high-order polynomial complexity of the MODG as a result of maximum parallelism, it is a very concise representation that explicitly expresses parallelism. Nevertheless, its design constraints and targeted 1-D and 2-D processor arrays can counterbalance for its complexity on modern computers/workstations and network of workstations.

REFERENCES

[1] *Virtex-II 1.5 v Field Programmable Gate Arrays*, Xilinx, Inc., San Jose, CA, 2000.

[2] C. Dick and F. J. Harris, "Configurable logic for digital communications: Some signal processing perspectives," *IEEE Commun. Mag.*, vol. 37, pp. 107–111, Aug. 1999.

[3] P. Sundaranjan and S. A. Guccione, "XVPI: A protable hardware/software interface for virtex," *Proc. SPIE*, vol. 4212, pp. 90–95, Nov. 2000.

[4] S. Y. Kung, *VLSI Array Processors*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

[5] *Virtex 2.5 v Field Programmable Gate Arrays*, Xilinx, Inc., San Jose, CA, 2000.

[6] *Apex II Programmable Logic Device Family, Ver. 1.1*, Altera Corporation, San Jose, CA, 2001.

[7] V. Srinivasan, S. Govindarajan, and R. Vemuri, "Fine-grained and coarse-grained behavioral partitioning with effective utilization of memory and design space exploration for multi-FPGA architectures," *IEEE Trans. VLSI Syst.*, vol. 9, pp. 140–158, Feb. 2001.

[8] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden, and D. Zaretsky, "A matlab compiler for distributed, heterogeneous, reconfigurable computing systems," in *Proc. IEEE Symp. Field-Programmable Custom Computing Mach.*, Napa Valley, CA, Apr. 2000, pp. 39–48.

[9] M. Einhardt and W. Luk, "Pipeline vectorization," *IEEE Trans. Computer-Aided Design*, vol. 20, pp. 234–248, Feb. 2000.

[10] A. Stone and E. S. Manalokos, "DG2VHDL: To facilitate the high level synthesize of parallel processing array architectures," *J. VLSI Signal Process.*, vol. 24, no. 1, pp. 99–120, Feb. 2000.

[11] M. J. Wolfe, *High Performance Compilers for Parallel Computing*. Reading, MA: Addison-Wesley, 1996.

[12] R. M. Karp, R. E. Miller, and S. Winograd, "The organization of computations for uniform recurrence equations," *J. ACM*, vol. 14, no. 3, pp. 563–590, Jul. 1967.

[13] A. Marquardt, V. Betz, and J. Rose, "Speed and area tradeoffs in cluster-based FPGA architectures," *IEEE Trans. VLSI Syst.*, vol. 8, pp. 84–93, Feb. 2000.

[14] P. Lee and Z. M. Kedem, "Synthesizing linear array algorithms from nested for loop algorithms," *IEEE Trans. Comput.*, vol. 37, pp. 1578–1598, Dec. 1988.

[15] L. Lamport, "The parallel execution of Do loops," *Commun. ACM*, vol. 17, no. 2, pp. 83–93, Feb. 1974.

[16] W. Shang and J. A. B. Fortes, "Time optimal linear schedules for algorithms with uniform dependencies," *IEEE Trans. Comput.*, vol. 40, pp. 723–742, June 1991.

[17] Condor High Throughput Computing. Univ. Wisconsin Madison. [Online]. Available: http://www.cs.wisc.edu/condor/

[18] *Sti3220 Motion Estimation Processor*, STMicroelectronics, Germany, 1994.

[19] H. Yeo and Y. H. Hu, "A novel modular systolic array architecture for full-search block matching motion estimation," *IEEE Trans. Circuit Syst. Video Technol.*, vol. 5, pp. 407–416, Oct. 1995.

[20] S. Kittitornkun and Y. H. Hu, "Frame-level pipelined motion estimation array processor," *IEEE Trans. Circuit Syst. Video Technol.*, vol. 11, pp. 248–251, Feb. 2001.

[21] T. Komarek and P. Pirsch, "Array architectures for block matching algorithms," *IEEE Trans. Circuit Syst.*, vol. 36, pp. 1301–1308, Oct. 1989.

[22] L. D. Vos and M. Stegherr, "Parameterizable VLSI architectures for the full-search block-matching algorithm," *IEEE Trans. Circuit Syst.*, vol. 36, pp. 1309–1316, Oct. 1989.

[23] D. R. Martinez, T. J. Moeller, and K. Teitelbaum, "Application of reconfigurable computing to a high performance front-end radar signal processor," *J. VLSI Signal Processing*, vol. 28, no. 1–2, pp. 65–83, May.–Jun. 2001.

[24] T. J. Moeller, "Field Programmable Gate Arrays for Radar Front-End Digital Signal Processing," M.S. dissertation, Mass. Inst. Technol., Cambridge, 1999.

Surin Kittitornkun (SM'96–M'02) received the B.Eng. (2nd Hons.) degree in engineering from the King Mongkut's Institute of Technology Ladkrabang (KMITL), Bangkok, Thailand, in 1992 and the M.Eng. degree from the Asian Institute of Technology (AIT), Bangkok, in 1995. He received the M.S.E.E. and Ph.D. degrees in electrical and computer engineering, from the University of Wisconsin-Madison (UWM), in 1997 and 2002, respectively.

Currently, he is a Lecturer in the Department of Computer Engineering, KMITL. In 1998, he joined the Broadband Wireless System Department, Motorola, Inc., Schaumberg, IL, as an intern. From 2000 to 2002, he was a Teaching Assistant at UWM. His research interests include FPGA/VLSI architecture for digital signal/image processing, and reconfigurable computing.

Dr. Kittitornkun received the Telecom Finland Prize for academic excellence from the Asian Institute of Technology (AIT), Bangkok, Thailand, on 1995.

Yu Hen Hu (F'00) received the B.S.E.E. degree from National Taiwan University, Taiwan, R.O.C., in 1976, and the M.S. and Ph.D. degrees, both in electrical engineering, from the University of Southern California, Los Angeles, in 1980 and 1982 respectively.

Currently, he is a Professor in the Department of Electrical and Computer Engineering, University of Wisconsin–Madison. Previously, he had been with the Department of Electrical Engineering, Southern Methodist University, Dallas, TX. His research interests include multimedia signal processing, design methodology, and implementation of signal processing algorithms and systems, sensor network and distributive signal processing algorithms, and neural network signal processing. He has served as associate editor for *Journal of VLSI Signal Processing* and *European Journal of Applied Signal Processing*, and has published more than 200 papers in technical journals and conferences. He is the editor of *Programmable Digital Signal Processors* (New York: Marcel Dekker, 2002) and *Handbook of Neural Network Signal Processing* (Boca Raton, FL: CRC, 2002).

Dr. Hu has served as associate editor for IEEE TRANSACTIONS ON SIGNAL PROCESSING and IEEE SIGNAL PROCESSING LETTERS. He has served as secretary of IEEE Signal Processing Society, on the Board of Governors of IEEE Neural Network Council, as Chair of IEEE Signal Processing Society, and on the Neural Network Signal Processing Technical Committee.