EURASIP Journal on Applied Signal Processing 2001:3, 129–146

Efficient Implementation of Nested-Loop Multimedia Algorithms

Surin Kittitornkun
Department of Electrical and Computer Engineering, University of Wisconsin, Madison, WI 53706, USA
Email: kittitor@cae.wisc.edu

Yu Hen Hu
Department of Electrical and Computer Engineering, University of Wisconsin, Madison, WI 53706, USA
Email: hu@engr.wisc.edu

Received 22 June 2001 and in revised form 22 August 2001

A novel dependence graph representation called the *multiple-order dependence graph* for nested-loop formulated multimedia signal processing algorithms is proposed. It allows a concise representation of an entire family of dependence graphs. This powerful representation facilitates the development of innovative implementation approach for nested-loop formulated multimedia algorithms such as motion estimation, matrix-matrix product, 2D linear transform, and others. In particular, algebraic linear mapping (assignment and scheduling) methodology can be applied to implement such algorithms on an array of simple-processing elements. The feasibility of this new approach is demonstrated in three major target architectures: application-specific integrated circuit (ASIC), field programmable gate array (FPGA), and a programmable clustered VLIW processor.

Keywords and phrases: dependence graph, systolic array, multiple-order, FPGA, VLIW.

1. INTRODUCTION

Many data-intensive multimedia algorithms such as motion estimation, 2D DCT/IDCT, matrix multiplication, and others consist of mainly deep nested-loops with relatively simple-loop body. They have been painstakingly implemented manually as hardwired ASICs (application specific integrated circuits) [1, 2] in order to meet the extremely high throughput rate demand of video signal processing algorithms. In view of the repetitive nature of the nested-loop algorithm formulation, the most effective strategy to speed up computation is to realize such an algorithm using an array of processing elements (PEs) to facilitate parallel processing.

In 1967, Karp et al. [3] introduced the notion of *uniform recurrence equation* as a powerful abstraction to describe nested-loop algorithm formulations. Lamport [4] considered the parallel execution of Do loops and proposed to use the algebraic construct of a *hyperplane* in the iteration index space to characterize the iterations that can be executed simultaneously. In 1982, Kung [5] and Leiserson [6] used the term *systolic array* to describe the pipelined execution of nested-loop formulated algorithms in a regular-structured, locally connected array of identical PEs. It has been argued that a systolic array structure is amenable for very large scale integration circuit (VLSI) implementation as it exploits maximum parallelism while requiring only local communication among processors. In the 1980s, Kung [7] and others [8, 9] have developed systematic design methodologies for systolic arrays. These methods facilitate algebraic mapping of the iteration indices of a nested-loop onto a systolic array with a linear task assignment and a linear schedule implementation.

With the rapidly increasing demands for real-time multimedia signal processing applications, it becomes ever important to pursue efficient implementation of data intensive multimedia processing algorithms. As pointed out earlier, many of them are suitable for parallel implementation using systolic array of PEs. One notable example is the numerous 1D and 2D array structures of block-based motion estimation algorithm [10, 11, 12, 13, 14].

In a recent study [15], we observed that the state-of-art systolic array design methodology does not always yield an optimal implementation. This is mainly due to a restrictive representation of a *single-assignment, localized* dependence graph (DG) [7]. More specifically, there are a number of situations where the execution order within a loop-nest may be reversed or even permuted without affecting the result. Summation of more than two numbers is a good example. On the other hand, data delivery within the array in such data-intensive algorithms can be organized such that redundant memory fetches can be eliminated to preserve memory

bandwidth. In a conventional dependence graph, every execution (data delivery) order must be explicitly specified. When there are multiple-choices of loop execution (data delivery) orders, one must select a specific execution order (data delivery) using heuristic. These premature commitments of particular orders often prevent an optimal implementation to be found.

In this paper, we propose a novel dependence graph representation, called *multiple-order dependence graph* (MODG). In a MODG, all the permissible execution (data delivery) orders will be represented explicitly in a concise format. As such, a single MODG representation can represent many of equivalent DGs. Since each of these equivalent DGs may lead to a locally optimal design, searching all of them combined is more likely to yield a globally optimal design.

As noted above, systolic array architecture style so far has been applied mainly for hardwired ASIC modules. With the advancement of deep submicron system-on-chip (SOC) manufacturing technology, it becomes clear that the cost of interconnect in terms of propagation delay, power consumption, and chip real estate has increased so much compared to that of logics. Furthermore, the speed gap between logic and dynamic memory gets widened.

As such, the architecture style of a regular array of processing elements with mostly local interconnects can be found in several different architecture styles. For example, modern field programmable gate array (FPGA) architectures such as the Xilinx Vertex II and the Altera APEX II are composed of a regular array of LUTs (look-up tables) and static RAM blocks as local storage. Even modern programmable digital signal processors also adopted a *clustered VLIW* (very long instruction word) architecture where the collection of functional units and a local register file within a cluster can be regarded as a powerful processing elements (PEs). These clusters can be programmed as a multiprocessor system to achieve parallel processing [16, 17].

In the remaining of this paper, we will first introduce basic definitions and the novel representation of MODG in Section 2. Then the design methodology based on MODG will be presented in Section 3. Next, we illustrate the advantages of MODG using three design examples: a hardwired ASIC implementation of block-based motion estimation, an FPGA implementation of matrix-matrix multiplication, and the implementation of separable 2D transform on a 4-cluster VLIW processor in Sections 4, 5, and 6, respectively. Lastly, Section 7 concludes this paper.

2. MULTIPLE-ORDER DEPENDENCE GRAPH

In this section, a novel representation of nested-loop, called *multiple-(execution) order dependence graph* as well as the basic definitions of dependence analysis will be introduced. The motivation for a multiple-order dependence graph representation will then be presented prior to its formal definition.

2.1. Nested-loop formulated algorithms

Many digital signal processing, image and video processing algorithms can be formulated compactly with a nested-loop formulation. Consider the following example of a matrix-matrix multiplication algorithm.

Example 1. Matrix-matrix multiplication $C = A \times B$

$$c_{i,j} = \sum_{k=1}^{K} a_{i,k} b_{k,j}, \tag{1}$$

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 multiple times

```
Do i = 1 to 3
    Do j = 1 to 3
        c[i, j] = 0
        Do k = 1 to 3
            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) corresponds to a single execution of the *loop body*.

In general, let i_m be the mth level loop index of loop-nest, and $\vec{i} = (i_1, i_2, \ldots, i_n)^t \in \mathbf{Z}$ be an n-dimensional 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 $\vec{a}$. Hence, the n-D *index space* $\mathbf{J}^n$ can be expressed as:

$$\mathbf{J}^n = \{\vec{i} = (i_1, i_2, \ldots, i_n)^t \mid i_1, i_2, \ldots, i_n \in \mathbf{Z}\}. \tag{2}$$

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

$$c[i, j] = c[i, j] + a[i, k] \times b[k, j], \tag{3}$$

where $a[i, k]$ and $b[k, j]$ are *input variables* and their values are needed to execute this loop, $c[i, j]$'s are *output variables* whose values will be computed by executing the loop-nest.

In Listing 1, the innermost k-loop is used to realize the summation of K product terms $a[i, k] \times b[k, j]$, $1 \le k \le K$. 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-times during the execution of the algorithm. In a *single-assignment* formulation [7], 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 $\{(i, j, k)^t; 1 \le j \le N\}$, and $b[k, j]$ will be made available to iteration indices $\{(i, j, k)^t; 1 \le i \le M\}$ 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 represented using intermediate variables to ensure that the single-assignment constraint is satisfied.

With the introduction of the 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

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

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

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

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

    EndDo k
  EndDo j
EndDo i

In the listing above, 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 also 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 recurrence equations). Therefore, 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.

A loop-nest is called a set of *uniform recurrence equations* (URE) [3] if its loop bounds are not functions of any output variables in the loop body and its dependence vectors are independent of the loop index $\vec{i}$. In other words, a URE's loop bounds are known constants before the execution of the algorithm. Almost all the data intensive nested-loop formulated multimedia algorithms are uniform recurrence equations. In general, a URE algorithm satisfying the single-assignment constraint can be described in a representation as shown in Figure 1 where l_m and u_m are i_m's lower and upper bounds and $m = 1, 2, \ldots, n$.

Three types of statement are included in the innermost loop body: the *Input/propagation 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, $\mathbf{I}_{v_j}^P \cap \mathbf{I}_{v_j}^I = \varnothing$ (empty set).

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 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

Figure 1: n-level nested Do-loop algorithm.

Compared to Listing 2, we have $a_3[\vec{i}]$ and $b_3[\vec{i}]$ as transmittal variables where $\vec{i} = (i,j,k)^t$. The initialization spaces of a_3 and b_3 are $\mathbf{I}_{a_3}^I = \{(i,j,k)^t \mid 1 \le i \le M, 1 \le k \le K, j = 0\}$ and $\mathbf{I}_{b_3}^I = \{(i,j,k)^t \mid 1 \le j \le N, 1 \le k \le K, i = 0\}$, respectively.

• *Computation variable.* The computation variable $v_k^n[\vec{i}]$ will be initialized to some constants at index set (iterations) $\vec{i} \in \mathbf{I}_{v_k}^N$, and then will be assigned to output values at other iterations $\vec{i} \in \mathbf{I}_{v_k}^C$ according to the recurrence function $\mathcal{F}_{v_k}$.

Use Listing 2 as an example, we have $\mathbf{I}_c^N = \{(i,j,0)^t \mid 1 \le i \le M, 1 \le j \le N\}$ and $\mathbf{I}_c^C = \{(i,j,k)^t \mid 1 \le i \le M, 1 \le j \le N, 1 \le k \le K\}$. The operator $\mathcal{F}_{v_k}$ is the summation operation.

• *Output variable.* These are results that will be stored back to memories outside the processor array. 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 $k = K$.

Based on the URE algorithm formulation, a set of data *dependence vectors* can be identified.

• *Propagation* dependence vector: $\vec{d}_{v_j}$. In Example 1,

$$\vec{d}_a = (0,1,0)^t, \qquad \vec{d}_b = (1,0,0)^t. \tag{4}$$

• *Computation* dependence vector: $\vec{d}_{v_k}$. Likewise,

$$\vec{d}_c = (0,1,0)^t. \tag{5}$$

These dependence vectors, together with the indices in the index space form a *dependence graph* (DG) as shown in Figure 2. In a dependence graph, each node represents the execution of an iteration of the loop-nest. An edge from one node to the other indicates that some data must be made available from the source iteration to the destination iteration.

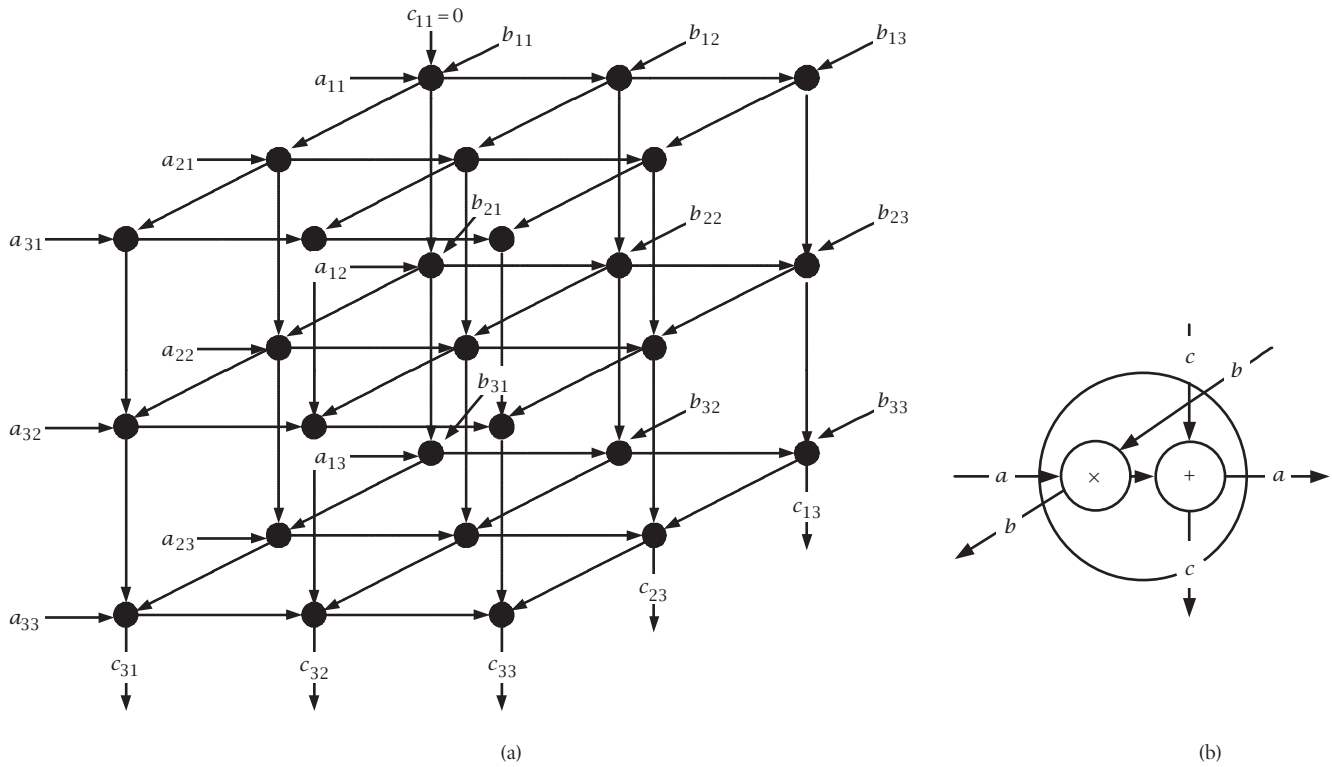

Figure 2: 3×3 matrix-matrix multiplication in a 3D dependence graph (a) and its node (loop body)(b).

The dependence vectors constrain the execution order that must be followed to ensure correctness of the algorithm. However, we note that in this example, alternate execution orders are possible:

(a) Since the operation of summation is invariant to any permutation of its operands, we may obtain the same result by reversing the direction of the dependence vector $(0, 0, 1)^t$ to $(0, 0, -1)^t$. This leads to a different execution order

$$c[i, j, k] = c[i, j, k + 1] + a[i, j, k] \times b[i, j, k]. \quad (6)$$

(b) Since the transmittal variables remain unchanged throughout the execution, the actual transmission direction will not be critical. Hence, instead of using current dependence vectors $(1, 0, 0)^t$, $(0, 1, 0)^t$, one may also use alternate dependence vectors $(-1, 0, 0)^t$, $(0, -1, 0)^t$.

Clearly, with the set of alternate dependence vectors, different, yet functionally equivalent dependence graph may be constructed. Motivated by this observation, we set out to propose a powerful way to represent a dependence graph and all those dependence graphs with alternative execution orders in a unified manner. We call it *multiple-(execution) order dependence graph* (MODG).

2.2. An n-dimensional multiple-order dependence graph (n-D MODG)

The dependence graph [7] or the iteration space dependence graph [18] is a graphical representation of data dependencies among loop iterations of a nested Do-loop. A directed dependence graph 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. It can be observed from Figure 2 that the 3-level nested Do-loop results in a 3D dependence graph. In contrast, 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 2 (n-dimensional multiple-order dependence 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 can be written as

$$k_n \triangleq (\vec{i}, D_V, I_V, O_V, F_V), \quad (7)$$

where V is a set of variables. Similar to C++ object oriented programming language, we use "." to access a particular field. 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,

$$k_n.D_V = \{k_n.D_v \mid \forall v \in V\}, \tag{8}$$

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 categories of variables.

2.2.1 Transmittal variable

In traditional systolic design methodology, each instance of input variable $v[\vec{g}]$ is reused and propagated according to a single dependence vector $\vec{d}_v$ as shown in Figure 1, that is,

$$k_n.D_v = \vec{d}_v, \quad \forall k_n.\vec{i} \in \mathbf{I}_v^p \cup \mathbf{I}_v^I. \tag{9}$$

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 dependence graph becomes localized. Once it is input inside the processor array, it is called the *transmittal* variable instead.

Actually, each instance $v[\vec{g}]$ of the variable v can be reused several times during the course of execution. Ordering should not be imposed as long as it is delivered correctly. 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 instance of variable $v[\vec{g}]$,

$$\mathbf{I}_v^{\vec{g}} = \{k_n.\vec{i} \mid \mathcal{H}_v(k_n.\vec{i}) = \vec{g},\ \forall k_n \in K_n\}, \tag{10}$$

where

$$\mathcal{H}_v(\vec{i}) : \vec{i} \longrightarrow \vec{g}, \quad \exists \vec{i} \in \mathbf{I}_v^p \cup \mathbf{I}_v^I. \tag{11}$$

The indexing function $\mathcal{H}_v$ is similar to $\mathcal{G}_v(\vec{i})$. Therefore, at any index point, each variable is associated with a set of multiple-order dependence vectors

$$k_n.D_v = \{\vec{i}_1 - k_n.\vec{i} \mid k_n.\vec{i} \neq \vec{i}_1,\ \forall \vec{i}_1 \in \mathbf{I}_v^{\vec{g}}\} \tag{12}$$

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

2.2.2 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

$$\begin{aligned} \mathcal{F}_v(a,b) &= \mathcal{F}_v(b,a), \\ \mathcal{F}_v(\mathcal{F}_v(a,b),c) &= \mathcal{F}_v(a,\mathcal{F}_v(b,c)) \\ &= \mathcal{F}_v(b,\mathcal{F}_v(a,c)), \end{aligned} \tag{13}$$

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 $\vec{d}_v$, that is,

$$k_n.D_v = \vec{d}_v, \quad \forall k_n.\vec{i} \in \mathbf{I}_v^N \cup \mathbf{I}_v^C, \tag{14}$$

as if it were an in-order operation.

Given a multiple-order operation, the instance $v[\vec{g}]$ 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 *a multiple-order index set* as a set of MODG nodes of a variable $v[\vec{g}]$ to be

$$\mathbf{I}_v^{\vec{g}} = \{k_n.\vec{i} \mid \mathcal{H}_v(k_n.\vec{i}) = \vec{g},\ \forall k_n \in K_n\}, \tag{15}$$

where

$$\mathcal{H}_v(\vec{i}) : \vec{i} \longrightarrow \vec{g}, \quad \exists \vec{i} \in \mathbf{I}_v^N \cup \mathbf{I}_v^C. \tag{16}$$

Therefore, a set of multiple-order dependence vectors of variable v can be obtained as

$$k_n.D_v = \{\vec{i}_1 - k_n.\vec{i} \mid k_n.\vec{i} \neq \vec{i}_1,\ \forall \vec{i}_1 \in \mathbf{I}_v^{\vec{g}}\}. \tag{17}$$

Back to the matrix product example, we can identify the indexing functions $\mathcal{G}$ 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 $k = K$. 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 $\mathcal{H}_c(i,j,k) = (i,j)$.

2.3. Summary

We call the dependence graph with multiple-order dependence vectors for both propagation and multiple-order operator a multiple-order dependence graph (MODG). Although MODG contains cycles due to those multiple-order dependence vectors, no self loops are introduced. MODG is still computable because it is formulated based on the computable dependence graph [7]. After the mapping is applied, a few of dependence vectors will become feasible and the final one will be selected appropriately. The mapping methodology will be introduced in the following section.

3. MAPPING METHODOLOGY OF MODG

Due to the regular structure of n-D MODG, the task of scheduling and assignment of each index (loop body) to execute on a number of processing elements (PEs) at a certain clock cycle becomes an algebraic projection. This projection is called *systolic* or *space-time* mapping. As a consequence, the index space $\mathbf{J}^n$ 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.

3.1. 1D space-time mapping

We propose the 1D space-time mapping of n-D MODG to a 1D array of PEs. The advantages of 1D array are three folds. First, the final 1D array can be adjusted to fit the chip area easily. Second, the input/output port is already on the array boundary. Third, the array is easy to rearrange to a 2D array. The 1D mapping matrix is defined below.

Definition 3 (1D space-time mapping matrix, T_1). The 1D space-time mapping matrix T_1 consists of a scheduling vector $\vec{s}$ and an allocation vector $\vec{P}$ as,

$$T_1 = \begin{bmatrix} \vec{s}^t \\ \vec{P}^t \end{bmatrix} = \begin{bmatrix} s_1 & s_2 & \cdots & s_n \\ p_1 & p_2 & \cdots & p_n \end{bmatrix}, \tag{18}$$

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 $\text{Rank}(T_1) = 2$.

Prior to applying T_1 to n-D MODG, we define the 1D PE array representation to accommodate the mapping. In a big picture, the mapping process can be visualized as shown in Figure 3.

Definition 4 (1D processing element array, K_1). A 1D 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 $k_1.t(\vec{i})$. Each PE is equivalent to a set of n-D MODG nodes,

$$k_1 \triangleq \{k_n \mid k_1.\mathbf{J}^1 = \vec{P}^t(k_n.\vec{i}),\ \forall k_n \in K_n\} \tag{19}$$

assigned (allocated) to the PE number $k_1.\mathbf{J}^1$, where $\text{PE}_{\min} \le k_1.\mathbf{J}^1 \le \text{PE}_{\max}$ and

$$\begin{aligned} \text{PE}_{\max} &= \max(\vec{P}^t\vec{q} \mid \vec{q} \in \mathbf{J}^n), \\ \text{PE}_{\min} &= \min(\vec{P}^t\vec{q} \mid \vec{q} \in \mathbf{J}^n). \end{aligned} \tag{20}$$

The clock schedule associated with each $k_1 \in K_1$ is obtained from

$$k_1.t(\vec{i}) = \{\vec{s}^t(k_n.\vec{i}) \mid k_1.\mathbf{J}^1 = \vec{P}^t(k_n.\vec{i}),\ \forall k_n \in K_n\}. \tag{21}$$

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 $k_1.E_v[\tau] - k_1.R_v[\tau]$ pair results from the space-time mapping of the multiple-order dependence vectors of an MODG node, such that

$$\begin{bmatrix} k_1.R_v[\tau] \\ k_1.E_v[\tau] \end{bmatrix} = T_1(k_n.D_v) = \begin{bmatrix} \vec{s}^t \\ \vec{P}^t \end{bmatrix} k_n.D_v. \tag{22}$$

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 the next two subsections.

Although this 1D mapping matrix T_1 was originally proposed by Lee and Kedem [19], ours 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.
- Ours can be applied to several different target architectures such as ASIC, FPGA and clustered VLIW which will be illustrated later.

3.2. Mapping objective functions: performance

The 1D 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 optimization as well.

3.2.1 Number of cycles (N_{cycle})

For ASIC implementation, the total parallel execution time, t_{total}, can be computed by

$$t_{\text{total}} = t_{\text{cycle}} \times N_{\text{cycle}}, \tag{23}$$

where t_{cycle} is the PE's longest critical propagation delay, which depends on the PE architecture and the implementation technology, and N_{cycle} is the number of clock cycles [7] given by

$$N_{\text{cycle}} = \max_{\vec{p},\vec{q}\in\mathbf{J}^n} \{\vec{s}^t(\vec{p} - \vec{q})\} + 1. \tag{24}$$

In other words, it is the number of cuts by the equitemporal hyperplane [4] 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.

3.2.2 Number of PEs (N_{PE})

In the 1D or linear array mapping, the number of PEs, N_{PE} can be expressed as

$$N_{\text{PE}} = \text{PE}_{\max} - \text{PE}_{\min} + 1. \tag{25}$$

In other words, N_{PE} is the number of distinct projections of the index space $\mathbf{J}^n$ on the vector $\vec{P}$.

3.2.3 Utilization (U)

The maximum PE array utilization is the maximum ratio of active PEs and the number of PEs, N_{PE}. It is obtained

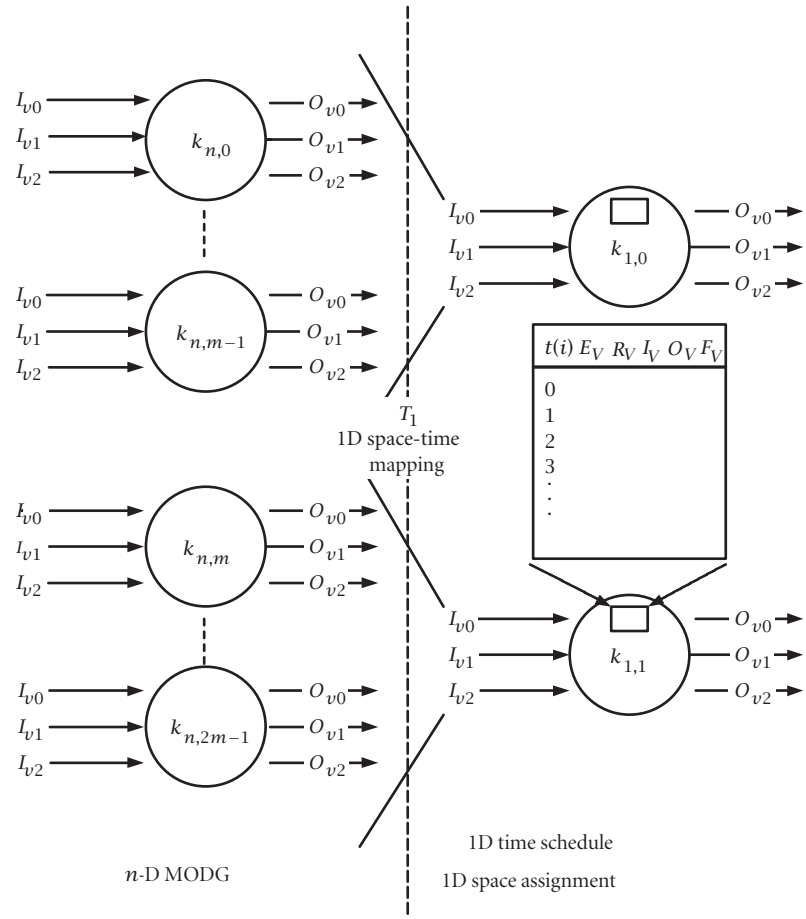

FIGURE 3: Space-time mapping of n-D multiple order dependence graph to a 1D processor array.

by

$$U_{\max} = \frac{\max_{\tau} |\{k_1 \mid \tau \in k_1.t(\vec{i})\}|}{N_{\mathrm{PE}}}, \tag{26}$$

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 $N_{\mathrm{cycle}} - N_{\mathrm{PE}}$ product,

$$U_{\mathrm{avg}} = \frac{|K_n|}{N_{\mathrm{PE}} \times N_{\mathrm{cycle}}}. \tag{27}$$

3.2.4 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

$$\mathrm{IO}_v[\tau] = \{k_1 \mid k_1.F_v[\tau] = 1,\ \tau \in k_1.t(\vec{i}),\ \forall k_1 \in K_1\}. \tag{28}$$

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

$$\#\mathrm{IO}_v = \max_{\tau} |\mathrm{IO}_v[\tau]|, \tag{29}$$

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

3.2.5 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 (28), B_v is equivalent to the total number of input/output occurrences averaged over N_{cycle},

$$B_v = \frac{\sum_{\tau \in k_1.t(\vec{i}), \forall k_1 \in K_1} |\mathrm{IO}_v[\tau]|}{N_{\mathrm{cycle}}}. \tag{30}$$

3.3. Design constraints

The 1D 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.

3.3.1 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, that is, $T_1\vec{p} = T_1\vec{q}$, $\vec{p} \neq \vec{q}$, $\vec{p}, \vec{q} \in \mathbf{J}^n$. Unlike [19, 20], no *communication conflict* is introduced by our method. In order to efficiently detect the conflicts, a 2D integer array A of size $N_{\text{PE}} \times N_{\text{cycle}}$ is used,

$$\text{Conflict} = \text{No} \xleftrightarrow{\text{iff}} A[T_1\vec{q}] \leq 1, \quad \forall \vec{q} \in \mathbf{J}^n, \tag{31}$$

where

$$A[T_1\vec{q}] = \begin{cases} 0 & \text{initialization,} \\ A[T_1\vec{q}] + 1 & \text{otherwise.} \end{cases} \tag{32}$$

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 [19] is $O(N^{2n})$.

3.3.2 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 $\mathbf{I}_v^{\vec{g}}$ of each input instance $v[\vec{g}]$. Thus, the terminal flag is assigned by

$$k_1.F_v[\tau] = \begin{cases} 1, & \tau = \min\left(k_1.t(\vec{i})\right) \\ 0, & \tau > \min\left(k_1.t(\vec{i})\right) \end{cases} \quad \forall k_n.\vec{i} \in \mathbf{I}_v^{\vec{g}}. \tag{33}$$

3.3.3 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 (15). Thus, the terminal flag is assigned by

$$k_1.F_v[\tau] = \begin{cases} 1, & \tau = \max\left(k_1.t(\vec{i})\right) \\ 0, & \tau < \max\left(k_1.t(\vec{i})\right) \end{cases} \quad \forall k_n.\vec{i} \in \mathbf{I}_v^{\vec{g}}. \tag{34}$$

3.3.4 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.

$$\max\left(\vec{s}^t\vec{p} \mid \forall \vec{p} \in \mathbf{I}_{v_p}^{\vec{f}}\right) < \min\left(\vec{s}^t\vec{q} \mid \forall \vec{q} \in \mathbf{I}_{v_c}^{\vec{g}}\right). \tag{35}$$

3.4. Heuristic search

Due to the exponential growth in both area and time complexity of MODG, its mapping methodology must be limited to some heuristic search. To be competitive in the market, some strategies are necessary to cut the design time and cost. From the definition of T_1 in (18), its elements are limited to $s_i \in \{0, \pm 1, \pm l_i \pm 1, \pm u_i \pm 1\}$ and $p_i \in \{0, \pm 1, \pm l_i \pm 1, \pm u_i \pm 1\}$. As a matter of fact that the search is independent to one another, a number of different iterations can be distributed in a network of workstation [21] and compared to the performance evaluations at the end. Finally, the problem size should be scaled down to reduce the computational complexity.

3.5. Summary

In summary, the mapping can be cast as an optimization problem with the objective functions in Section 3.2 subject to the design constraints in Section 3.3 with some search strategies in Section 3.4. Furthermore, the search of T_1 must be subject to the architecture mapping constraints of hardwired ASIC, FPGA, and clustered VLIW processor in the subsequent sections, respectively.

4. HARDWIRED ASIC MAPPING

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 [1]. As a major step towards saving memory bandwidth, Yeo and Hu [13] 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 of block 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 $(2p + 1)^2$ blocks in the search area, can be obtained by

$$MV = \arg\left\{\min \text{MAD}(m, n)\right\} - (p, p), \quad 0 \leq m, n \leq 2p, \tag{36}$$

where 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

$$\text{MAD}(m, n) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \left| x(i, j) - y(i + m - p, j + n - p) \right|, \tag{37}$$

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 Figure 4 where $D_{\min}$ is the *minimum distortion* measured using MAD. It can be noticed that MAD(m,n) and min are of multiple-order operators. In addition, it has been observed in [14] that there are many possible data propagation patterns. Therefore, the proposed MODG is applicable.

Do $v = 0$ to $N_v - 1$
Do $h = 0$ to $N_h - 1$
  $MV(h,v) = (0,0)$;
  $D_{\min}(h,v) = \infty$;
  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)
      $D_{\min}(h,v) =$ MAD(m,n);
      $MV(h,v) = (m - p, n - p)$;
    endif
EndDo n, m, h, v

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

4.1. Result

Due to the limited presentation space, we can scale the problem size down to $N_v = 3$, $N_h = 3$, $N = 4$, and $p = N/2 = 2$. As indicated earlier in Section 3, T_1 must be searched to obtain the final 1D array that satisfies numerous constraints. The heuristic search is limited to the surrounding regions to the previous result from [14, 15]. Our objective is to equalize the bandwidth of current frame pixel x and previous frame pixel y while minimizing N_{cycle}.

The optimal solution is

$$T_1 = \begin{bmatrix} N^2 & N_h N^2 & 2p+1 & 2 & N & 1 \\ 0 & 0 & 2p+1 & 1 & 0 & 0 \end{bmatrix}. \tag{38}$$

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 number of registers (delay elements).

Depicted in Figure 5, the 2D array is obtained semi-automatically where the □ indicates the input ports to the array. On the other hand, the schedules of current frame pixel $x(i,j)$, previous frame pixel $y(i,j)$ at $\tau = 35, 36$, and 37 are listed in Figure 6(a), (b), and (c), respectively. Each $x(i,j)$ propagates from left PE to right PE every two cycles in parallel and to the consecutive row every five cycles only in the first column. The propagation of $D_{\min}$ is of the similar pattern while the final result is the minimum one among all rows. The previous frame pixels $y(i,j)$ propagate downward every cycle and to the right PE in the first row only. Although the schedules show that two $y(i,j)$ pixels are needed per cycle, every $y(i,j)$ is fetched once and reused throughout the entire execution.

4.2. Discussions

Table 1 quantitatively compares different aspects among 2D arrays in terms of search range, N_{PE}, throughput (cycles per block), memory bandwidth ratio B_y/B_x, and fan-out. We were able to achieve unity B_y/B_x bandwidth ratio. It can be noticed that ours B_y/B_x ratio is superior to others with a few more registers to compensate with the huge demand of y's memory bandwidth. The less bandwidth demand, the less pressure on the on-and off-chip memory becomes. The array can eliminate redundant memory fetches thus preserve memory (I/O) transactions as well as power consumption at the same time.

We define *fan-out* as the total number of loads (sinks) whose source has more than two loads. It essentially represents the undesirable broadcasting mechanism. Greater fan-out implies bigger effective capacitance C_{eff} which the propagation delay as well as the power consumption according to the very well-known equation,

$$\text{Power} \propto C_{\text{eff}} V_{\text{sup}}^2 f_{\text{CLK}}, \tag{39}$$

where V_{sup} and f_{CLK} are the supply voltage and operating clock frequency, respectively.

It can be observed at this point that the MODG and its mapping methodology can reduce redundant memory fetches of previous frame pixels by at least 50% while achieving frame-level pipelining [14], and no broadcast. The bottom line is that this 2D array is *not achievable* from the traditional mapping methodology [7]. That is because every variable instance must propagate/compute in only one direction by enforcing a uniform edge-delay pair. The MODG can represent the motion estimation loop-nest as if it were represented in human thinking which results in a more flexible array structure and more efficient implementation.

5. FPGA ARCHITECTURE MAPPING

FPGAs can serve not only as a vehicle for rapid prototyping, but also as a deployment of special-purpose reconfigurable embedded architecture. In recent million-gate SRAM-based FPGA architectures, the on-chip programmable modules are organized in a regular fashion with pipelined local as well as long interconnects. Thus, it provides excellent implementation of systolic array structure. For example, the recent implementation of 512-tap systolic FIR filter on Xilinx Virtex can achieve maximum clock frequency of 117 MHz [23]. They pointed out that the current FPGA mapping tools are still inefficient for mapping DSP algorithm to its regular structure.

Based on traditional design methodology [7], only shift registers (delay elements) are required. This is mainly due to the hardwired architecture. In this section, our target

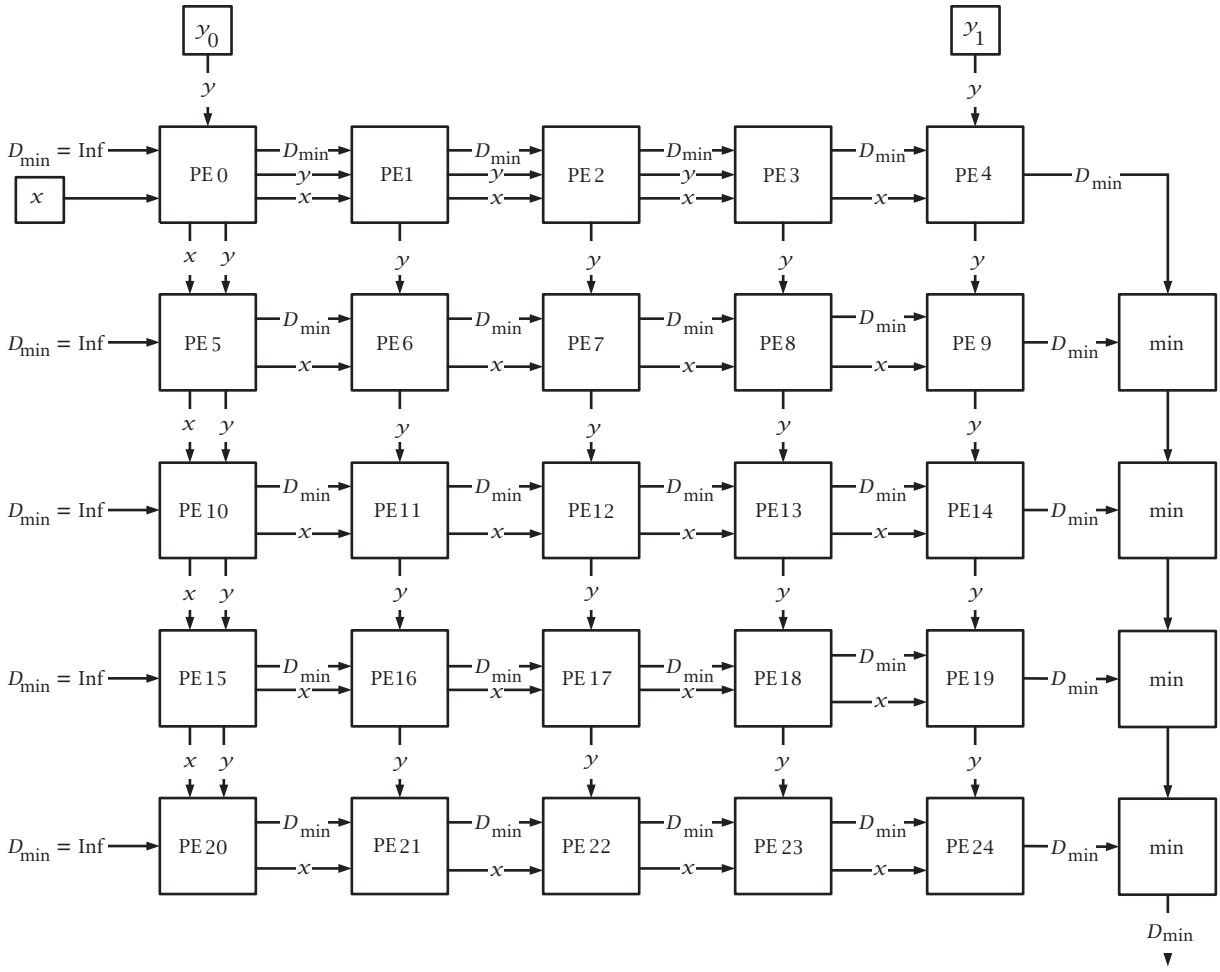

Figure 5: 2D full search block matching motion estimation array ($N = 4$, $p = N/2 = 2$).

architecture is the SRAM-based FPGAs such as those of Xilinx Virtex family.

5.1. Programmable-length shift register

It is a magnificent feature of the look-up table (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 of length $N \le 16$. Hence, an 8-bit N-cycle delay consumes only 8 LUTs if $N \le 16$ rather than $8N$ flip/flops or equivalently $8N$ LUTs in FPGA because one flip/flop is provided per LUT [24]. 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 equations.

$$\begin{aligned} k_1.r_v[\tau] &> 0, \\ k_1.e_v[\tau] &\neq 0, \\ |k_1.e_v[\tau]|\,(k_1.r_v[\tau]) &= \min\left(k_1.R_V[\tau]\,|\,k_1.E_v[\tau]\,|\right). \end{aligned} \tag{40}$$

5.2. Distributed 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 [24], two LUTs in the same slice can be configured as a single-port 32×1-bit ROM/RAM or a dual-port 16×1-bit ROM/RAM . Unlike 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:

$$\begin{aligned} k_1.R_v[\tau] &= \{mN \mid N > 0,\ m \neq 0,\ m \in \mathbf{Z}\}, \\ k_1.E_v[\tau] &= \{0\}. \end{aligned} \tag{41}$$

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

5.3. 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

Table 1: Performance comparison for 2D FSBM ME arrays, $N = 4$, $p = N/2 = 2$.

Type	Search Range	N_{PE}	Throughput (Cycles/Block)	Registers (bytes)	B_y/B_x	Fan-Out
[10] AB2	$-2/+2$	16	40	94	10	0
[11] Type 1	$-1/+1$	16	40	180	2	100
[22]	$-2/+2$	16	65	94	4	0
[13]	$-2/+1$	16	16	64	2	16
[14]	$-2/+2$	25	16	152	2	8
Ours	$-2/+2$	25	16	164	1	0

$x(i,j)$ $y(i,j)$

11,6	11,4	10,6	10,4	9,6		9,4	9,3	8,6	8,5	7,8
10,5	10,3	9,5	9,3	8,5		9,3	9,2	8,5	8,4	7,7
9,4	8,6	8,4	7,6	7,4		9,2	8,5	8,4	7,7	7,6
8,3	7,5	7,3	6,5	6,3		9,1	8,4	8,3	7,6	7,5
6,6	6,4	5,6	5,4	4,6		8,4	8,3	7,6	7,5	6,8

(a)

12,3	11,5	11,3	10,5	10,3		9,5	9,4	9,3	8,6	8,5
10,6	10,4	9,6	9,4	8,6		9,4	9,3	8,6	8,5	7,8
9,5	9,3	8,5	8,3	7,5		9,3	9,2	8,5	8,4	7,7
8,4	7,6	7,4	6,6	6,4		9,2	8,5	8,4	7,7	7,6
7,3	6,5	6,3	5,5	5,3		9,1	8,4	8,3	7,6	7,5

(b)

12,4	11,6	11,4	10,6	10,4		9,6	9,5	9,4	8,7	8,6
11,3	11,5	10,3	9,5	9,3		9,5	9,4	9,3	8,6	8,5
9,6	9,4	8,6	8,4	7,6		9,4	9,3	8,6	8,5	7,8
8,5	8,3	7,5	7,3	6,5		9,3	9,2	8,5	8,4	7,7
7,4	6,6	6,4	5,6	5,4		9,2	8,5	8,4	7,7	7,6

(c)

Figure 6: Schedule of full search block matching motion estimation: (a) at cycle $\tau = 35$, (b) $\tau = 36$, and (c) $\tau = 37$ ($N_h = 3$, $N_v = 3$, $N = 4$, $p = N/2 = 2$).

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:

$$\begin{aligned} &\#\mathrm{IO}_v[\tau] = 1, \quad \forall \tau \in k_1.t(\vec{i}),\ \forall k_1 \in K_1 \\ &\#\mathrm{IO}_v > 1. \end{aligned} \tag{42}$$

5.4. Other constructs

Algorithms that involve inner dot products such as FIR filters can exploit the LUTs to store filter coefficients and fast-carry adders using distributed arithmetic [25]. In modern the 10-million-gate FPGA architecture like Xilinx Virtex II, a number of highly optimized 18-bit ×18-bit multipliers are provided and distributed throughout the chip [26].

5.5. Example: matrix-matrix multiplication

Matrix-vector and -matrix multiplications are the fundamental operation of 1D orthogonal transform, 2D/3D graphics, and many others. A number of 1D processor arrays have been proposed in [19, 20, 27, 28]. However, their results suffer from low processor utilization due to propagation constraints of unknown input data. Normally, the coefficient matrix is known beforehand. We can exploit this fact using MODG to FPGA architecture mapping.

We let x, c, and y be the $N \times N$ input, coefficients, and output matrices, respectively, where

$$y = cx. \tag{43}$$

The algorithm can be formulated as 3-level nested Do-loop as shown earlier. Our goal is to minimize N_{PE} and N_{cycle} while achieving 100% maximum utilization, single-port pipelined input data x, stored coefficients c, and single-port output data y. In a bounded solution space, the heuristic search pruned out more than half of the invalid solutions according to the mapping constraint in (31). Then, only the valid solutions were evaluated subject to the following constraints: the propagation input x in (40), the distributed ROM for coefficients c in (41), and the bus-based output constraint in (42). This transformation matrix T_1 is one of the solutions

$$T_1 = \begin{bmatrix} -1 & -4 & 1 \\ 1 & 0 & 0 \end{bmatrix}. \tag{44}$$

From (24) and (25) we achieve $N_{PE} = 4$ and $N_{cycle} = 19$ from the PE array and its clock schedule as shown in Figures 7 and 8, respectively. The propagation of input x is pipelined along the array to multiply with the stored coefficients c in each PE. The output y is accumulated every cycle and placed on the bus every N cycles. The output bus interface exploits abundant three-state buffers of FPGA. The schedule is equivalent to cutting the original 3D MODG of a matrix-matrix product into 2D slices, tiling each slice of MODG to a 2D MODG, and projecting the tiled 2D MODG to a 1D PE array.

Table 2 compares our 1D array with the previous works. All of them are targeted at 1D PE array. Assuming the same critical path delay, ours outperforms theirs in terms of N_{PE}, #IO, utilization, and latency. The smaller number of PEs,

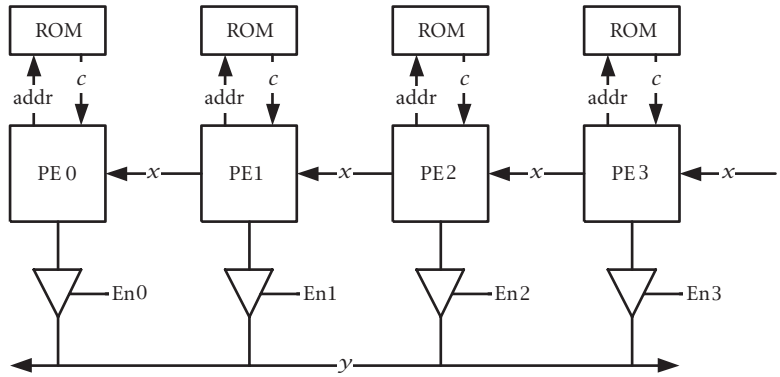

Figure 7: Architecture of pipelined bus $N \times N$ matrix multiplication $N = 4$.

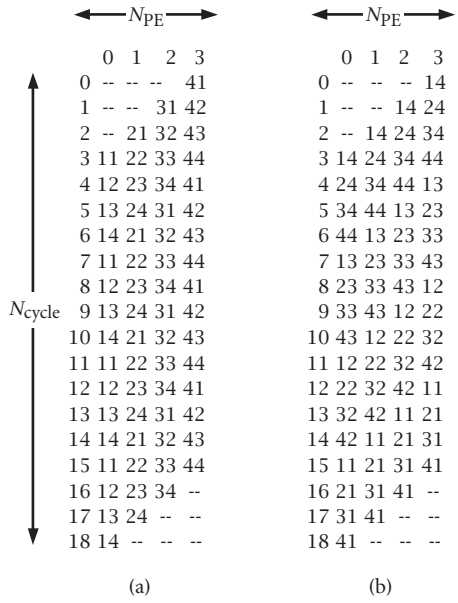

Figure 8: Schedule of pipelined bus $N \times N$ matrix multiplication $N = 4$: (a) coefficient $c[i, j]$ and (b) input $x[i, j]$.

the fewer resources it utilizes. Ours is even better since its target architecture is the FPGA. Due to the known coefficient matrix c, we save $M = 8$ pins where M is c's and x's bit precision and y is of $3M$-bit precision. The utilization figures, $U_{\max}$ and U_{avg}, are of importance when the power supply is limited. Unlike ours, other PE arrays spend most of the time propagating data. Finally, latency is more concerned as the delay from applying input to getting the first output increases which results in less hardware utilization. Each single-port N-entry M-bit ROM consumes $\lceil N/16 \rceil M$ LUTs [24] where $\lceil x \rceil$ denotes the smallest integer greater than or equal to x. The LUT consumption ratio of [27] to ours is over 3.85 with the multiplier built from LUTs. The ratio would increase up to 5.25 if the Virtex II architecture were assumed because each built-in multiplier does not consume any LUT [26].

Table 2: Matrix-matrix product, $N = 4$, $M = 8$ is the precision (bits) (Virtex FPGA is assumed).

Performance	[27]	[19]	[20]	[28]	Ours
PE Array	1D	1D	1D	1D	1D
N_{cycle}	31	19	19	19	19
N_{PE}	8	10	10	10	4
$U_{\max}$	50%	60%	70%	60%	100%
U_{avg}	25.8%	33.7%	33.7%	33.7%	84.2%
Latency(cycles)	8	18	18	25	4
#IO(pins)	$6M$	$5M$	$5M$	$5M$	$4M$
LUTs/PE	216	168	144	136	112
LUTs Ratio	3.85	3.75	3.21	3.04	1.00

Table 3: Number of LUTs/PE, $N = 4$, $M = 8$ is the precision (bits) (Virtex FPGA is assumed).

Performance	[27]	[19]	[20]	[28]	Ours
(24 + 16)-b Adder	24	24	24	24	24
(8-bit×8=16)-b Multiplier	48	48	48	48	48
24-b y's Registers	72	72	48	24	24
8-b c's Registers	56	8	16	24	8
8-b x's Registers	16	16	8	16	8
Total	216	168	144	136	112

Table 3 enumerates the number of LUTs estimated for datapath of each PE. The Xilinx core generator system provides not only the netlists of those components in PE datapath, that is, bit-parallel adder, multiplier, and others but also the accurate estimation of LUTs consumption. This allows the designer to predict the complexity of each PE in advance. On the other hand, the control circuitry of each PE was difficult to estimate because it is mechanism dependent and varies from architecture to architecture. Note that the mapping can be applied to matrix products of any size.

In the next section, we will apply the MODG and its space-time mapping to pipeline nested Do-loop algorithms in lustered VLIW video signal processors.

6. CLUSTERED VLIW MAPPING

After the extensive study of multimedia workload [16] especially video encoding/decoding, it has been analyzed that a video signal processor should be equipped with a large number of ALUs (arithmetic and logic units), shifters, multipliers, and multiported register file to gain the available data parallelism. Current examples of clustered VLIW processor are the TI's C6X Velociti and the analog device's TigerSharc families. The Princeton video signal processor [17, 29, 30] has been recently proposed and targeted at 1 GHz operating clock frequency. In order to achieve the goal, the so-called *clustered* VLIW has been adopted with up to eight homogeneous clusters of functional units. Some time-critical video signal processing algorithms involve deep nested Do-loop such as 2D DCT/IDCT (discrete cosine transform/inverse discrete cosine transform), block-based motion estimation, and others.

However, the current instruction scheduling of clustered VLIW is limited to basic block level [31] and tries to perform both partitioning and instruction scheduling to all clusters [32]. Particularly multimedia workload, inter-procedure analysis, and loop transformation are needed to discover coarse-grain parallelism that is not visible at basic block level [17]. Hence, there is not enough instruction-level parallelism (ILP) to fully utilize the available hardware parallelism.

As a coarse-grain loop pipelining methodology, the MODG and its space-time mapping can be applied to fully exploit loop-level parallelism. We assume the clustered VLIW architecture just like the Princeton video signal processor with predicate execution capability currently adopted by the Intel IA64 architecture. The algorithm to its architecture mapping is subject to the following constraints. However, these constraints can be used as objective functions as well.

6.1. Number of clusters (C)

Since N_{PE} is a function of problem size, it can be greater than the number of clusters C after the mapping. The solutions are constrained such that the number of PEs is an integer multiple- of the number of clusters.

$$N_{\mathrm{PE}} = mC, \tag{45}$$

where $m > 0$, $m \in \mathbf{Z}$. Therefore, m PEs will be mapped to the same cluster. This is beneficial to instruction scheduler because basic block will be expanded so that intracluster parallelism can be exploited.

6.2. Intercluster communication channels

The number of intercluster transactions should be minimized to avoid being communication bottleneck. The links can be implemented as crossbar switches [30] or busses [31]. In each PE or cluster k_1, the number of occurrences of nonzero-length edge and nonnegative delay pair of variable v can be expressed as $\{\tau \mid \exists k_1.E_v[\tau]_i > 0, \exists k_1.R_v[\tau]_i \geq 0\}$ where $k_1.E_v[\tau]_i$ and $k_1.R_v[\tau]_i$ are the ith element of $k_1.E_v[\tau]$ and $k_1.R_v[\tau]$, respectively. Hence, its cardinality yields the number of intercluster transactions of PE $k_1 \in K_1$.

6.3. Local register file size

If the loop is unrolled according to the obtainable solution, the basic block will be enlarged and instruction scheduler will be able to exploit ILP more. However, the number of registers needed may grow. Since each cluster has its own local register file, register allocation can be performed independently. To aid the register allocation using graph coloring such as the one proposed by Chaitin et al. [33], live range of each variable instance can be obtained to formulate register interference graph. For each processor number $k_1.\mathbf{J}^1$ the LiveRange of variable instance $v[\vec{g}]$ can be written as

$$\begin{aligned}&\mathrm{LiveRange}_v^{\vec{g}}\\ &\quad = \max\{\vec{s}^t(\vec{p}-\vec{q}) \mid \forall \vec{p}, \forall \vec{q} \in \mathbf{I}_v^{\vec{g}},\ k_1.\mathbf{J}^1 = \vec{P}^t\vec{p} = \vec{P}^t\vec{q}\}.\end{aligned} \tag{46}$$

6.4. Local memory space

The number of variable v's instances in the PE of processor number $k_1.\mathbf{J}^1$ is the cardinality of the following set, $\{\vec{g} \mid \vec{P}^t(k_n.\vec{i}) = k_1.\mathbf{J}^1, \exists k_n.\vec{i} \in \mathbf{I}_v^{\vec{g}}\}$. The total space of local memory required in each cluster is the summation over all variables, $\forall v \in V$. The size of local memory should be large enough to hold all necessary variable instances. For example, local memory of 2 Kbytes per cluster is projected by [16].

6.5. Example: 2D separable transformation

The 2D DCT/IDCT is a common 2D separable linear transform in image and video processing. Its matrix-matrix product is formulated as

$$X = c(cx)^t, \tag{47}$$

where c, x, and X are the $N \times N$ transform coefficient, input and output matrices, respectively. We can rewrite it in a two-pass matrix-matrix multiplication

$$y = cx, \qquad Y = y^t, \qquad X = cY. \tag{48}$$

Its single loop-nest algorithm was derived in the appendix for more details. It can be noticed that the innermost loop body is invariant. Therefore, loop collapsing [18] was applied. Eventually, the single-assignment loop and its dependence graph can be illustrated in Figures 9 and 10, respectively. As a result, we could formulate its MODG. The mapping is constrained to the following conditions. First, since practical 2D DCT/IDCT are of $N \times N$ elements where $N = 8$, the mapping is constrained to $N_{\mathrm{PE}} = C = N = 4$ in this case. Second, one input port per input variable, c and x. Finally, intercluster transaction should be kept as low as possible. One of the solutions is this T_1,

$$T_1 = \begin{bmatrix} -1 & -8 & 1 \\ -1 & 0 & 0 \end{bmatrix}. \tag{49}$$

The assignments and schedules of x, Y, and c are illustrated in Figure 11. The $c[m,n]$-$x[m,n]$ pair or the $c[m,n]$-$Y[m,n]$ pair represents the iteration in the first and second

passes, respectively. The iterations are assigned to clusters 0–3 and skewed/ordered relative to one another to pipeline fetching data from data cache. The overall schedule shows that no intercluster transactions are required. The following is a simple-pseudo assembly code for matrix-matrix product of $y = cx$.

```
ldi R12,0          ;i = 0
L0:                ;label L0
ldi R11,0          ;j = 0
L1:                ;label L1
ldi R11,0          ;k = 0
L2:                ;label L2
ldi R3,0           ;y[i,j] = 0
ldi R10,0          ;k = 0
ld R0,c[R12,R10]   ;R0← c[i,k]
ld R1,x[R10,R11]   ;R1← x[k,j]
mul R2,R0,R1       ;R2← c[i,k] × x[k,j]
add R3,R3,R2       ;R3← R3+c[i,k] × x[k,j]
addi R10,R10,1     ;k = k + 1
ble L3,R10,4       ;branch to L1 if R10 ≤ 4
st y[R12,R11],R3   ;y[i,j] ←R3
addi R11,R11,1     ;i = i + 1
ble L2,R11,4       ;branch to L2 if R11 ≤ 4
addi R12,R12,1     ;j = j + 1
ble L1,R12,4       ;branch to L3 if R12 ≤ 4
```

According to the loop schedule in Figure 11, the assembly code above is replicated in all four clusters to form a bigger loop body. They are pipelined/skewed by one cycle as shown in Figure 12. We assume that each cluster has one memory load/store unit that takes one cycle latency if data cache hits. Although the first load may stall the following instructions for a certain number of clock cycles, the subsequent loads will take only one cycle due to cache hit. This corresponds to the constraint of having single input port for c and x. The four loop bodies are almost identical and can be scheduled with any existing instruction scheduling algorithm.

Unlike software pipelining that schedules instructions from adjacent loop iterations to the available functional units in any cluster [31], our scheme is different in such a way that each cluster is assigned to execute instructions solely belonging to an independent loop body. In addition, software pipelining using modulo scheduling [34] yields each result every initiation interval to increase throughput rate which can be limited by the available ILP. Ours instead enhances the throughput by a larger factor of the number of clusters C executing in parallel. Furthermore, the subsequent independent iterations can be executed in parallel if the loop is unfolded and more functional units such as multiple-ALUs and multipliers are available.

7. CONCLUSION

Multimedia signal processing is subject to real-time constraint, for example, audio/video coding/decoding, 2D/3D graphics. To meet such high throughput demand, either application-specific integrated circuit (ASIC) or application-specific instruction set processor (ASIP) is often utilized. ASIC can be of the form hardwired ASIC or programmable hardware like FPGA. On the other hand, clustered VLIW architecture is adopted as the main architecture of ASIP to exploit static instruction level parallelism due to predictable behavior of multimedia algorithms.

As a matter of fact that most of multimedia algorithms are characterized as data-intensive and data-parallel algorithms. Furthermore, a huge portion of total execution time is consumed by elementary operations such as motion estimation, matrix multiplication, 1D/2D linear transform, and others that can be described as nested-loop with simple-innermost loop body. The novel multiple-order dependence graph (MODG) is proposed to represent all possible execution and data delivery orders. Along with its architecture mapping methodology, it is targeted as a computer aided design tool.

Depending upon the target architecture, our design methodology can assign and schedule parallel/pipelined execution of nested Do-loop algorithms subject to architecture-specific constraints. We have demonstrated the applications of MODG in three different implementation styles, hardwired ASIC, FPGA, and clustered VLIW processor using block matching motion estimation, matrix-matrix product, and 2D separable transformation, respectively.

Do $m = 1$ to N
  Do $n = 1$ to N
    Do $k = 1$ to N

$$c_3[m,n,k] = \begin{cases} c[m,k], & n = -1 \\ c_3[m,n-1,k], & n \geq 0 \end{cases}$$

$$x_3[m,n,k] = \begin{cases} x[k,n], & m = -1 \\ x_3[m-1,n,k], & m \geq 0 \end{cases}$$

$$y_3[m,n,k] = \begin{cases} 0, & k = -1 \\ y_3[m,n,k-1] + c_3[m,n,k] \\ \qquad \times x_3[m,n,k], & k \geq 0 \end{cases}$$

    EndDo k
    Do $k = N + 1$ to $2N$

$$c_3[m,n,k] = \begin{cases} c[m,k-N], & m = -1 \\ c_3[m-1,n,k], & m \geq 0 \end{cases}$$

$$Y_3[m,n,k] = \begin{cases} y_3[n,m,k], & k = N \\ Y_3[m,n,k-1], & k > N \end{cases}$$

$$X_3[m,n,k] = \begin{cases} 0, & n = -1 \\ X_3[m,n-1,k] + c_3[m,n,k] \\ \qquad \times Y_3[m,n,k], & n \geq 0 \end{cases}$$

    EndDo p
  EndDo n
EndDo m

Figure 9: Single-assignment of two-dimensional separable transform algorithm.

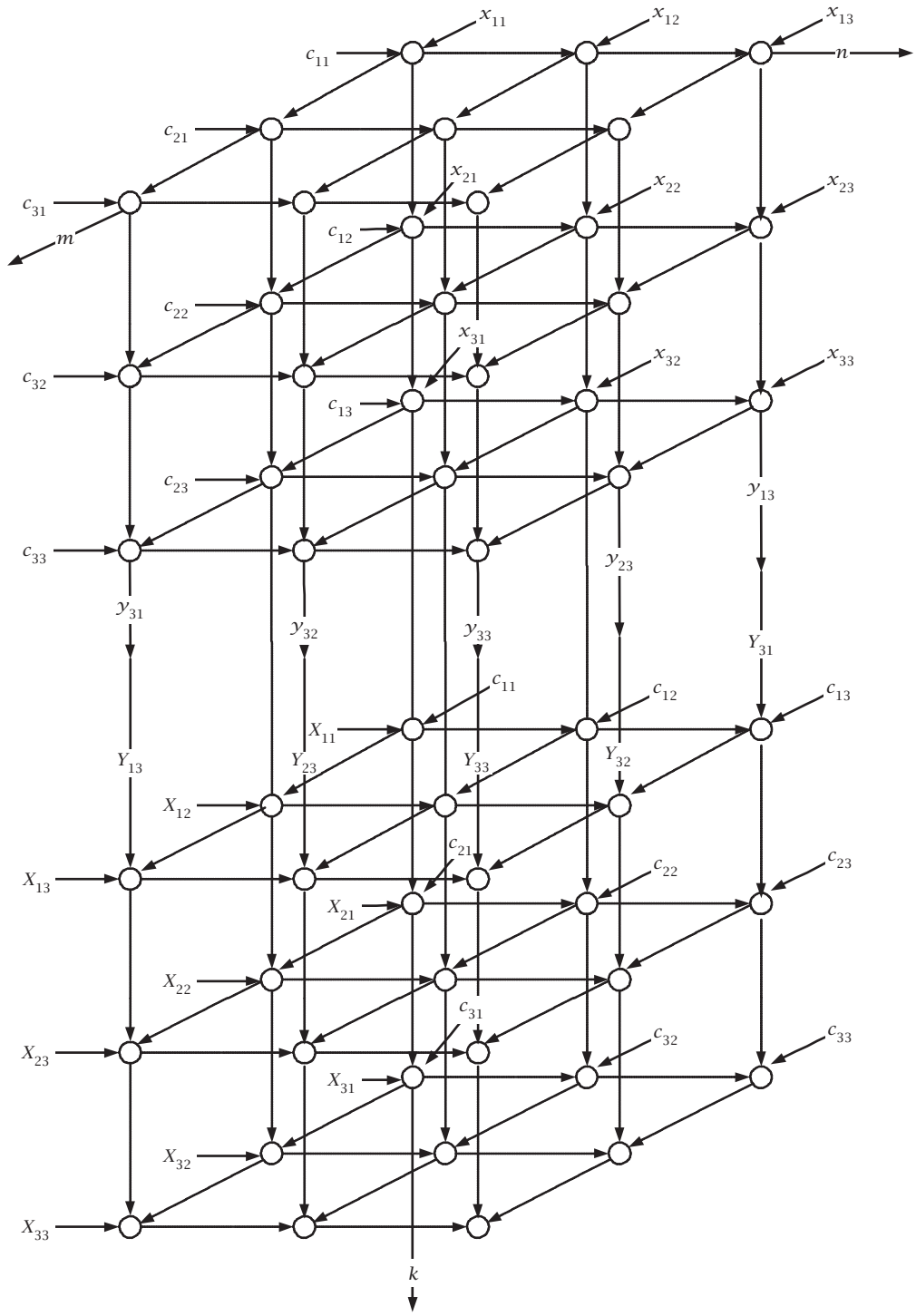

FIGURE 10: Dependence graph of two-dimensional transformation, $Y = (cx)^t$, $X = cY$, $N = 3$.

Firstly, the best-ever 2D systolic motion estimation array was obtained in hardwired architecture. It outperforms others by demanding the least memory bandwidth with mostly local interconnects, small amount of fan-out, and reasonable number of registers. Secondly, we were able to obtain systolic array structures that match well to the modern FPGA architecture with optimal execution time, less LUT consumption, low I/O pin count, high processor utilization, and consequently low power consumption. Finally, the achievable loop-level parallel execution schedule for clustered VLIW processor was able

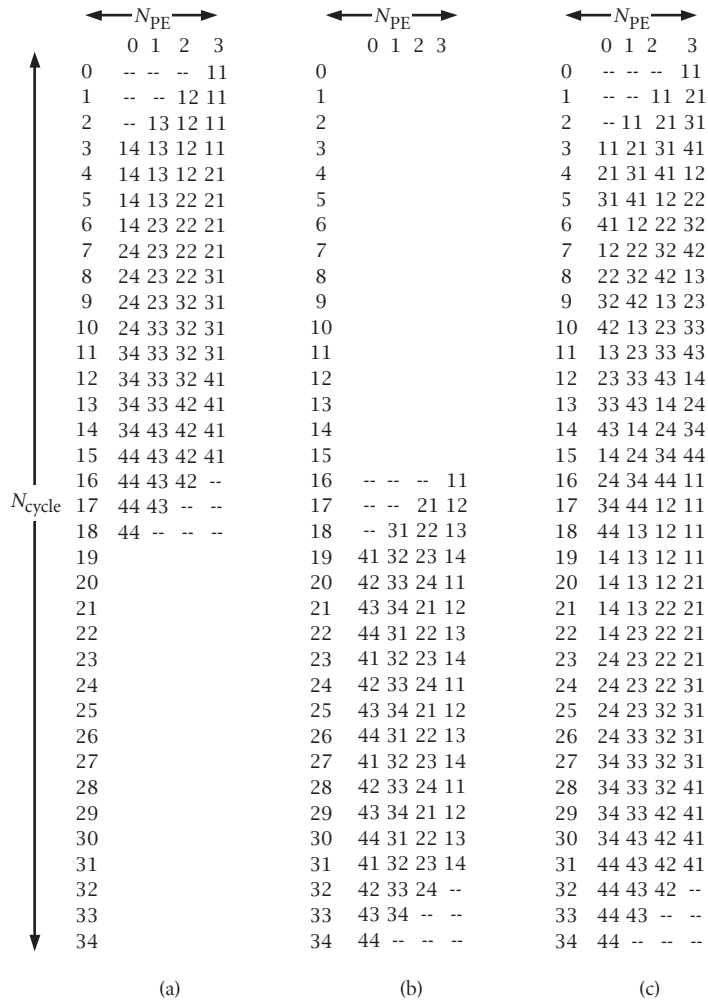

FIGURE 11: Schedule of 2D separable transformation: (a) input $x[i, j]$, (b) intermediate data $Y[i, j]$, and (c) coefficients $c[i, j]$.

Cluster 0	Cluster 1	Cluster 2	Cluster 3
			ldi r12,0
		ldi r12,0	ldi r11,0
	ldi r12,0	ldi r11,1	ldi r10,0
ldi r12,0	ldi r11,2	ldi r10,0	ldi r3,0
ldi r11,3	ldi r10,0	ldi r3,0	ldi r10,0
ldi r10,0	ldi r3,0	···	ld r0,c[r12,r10]
ldi r3,0	···		ld r1,x[r10,r11]
···			mul r2,r0,r1
			add r3,r3,r2
			addi r10,r10,1
			st y[r12,r11],r3
			addi r11,r11,1
			addi r12,r12,1
			...

FIGURE 12: Instruction scheduling of cluster-parallel pipelined matrix-matrix product, $C = 4$.

to exploit both cluster level and instruction level parallelism. The methodology acts like a coarse-grain loop pipelining scheduler to parallel available clusters while minimizing intercluster communication.

Although the time and area complexities of the MODG grow exponentially, it is the only concise representation that expresses parallelism explicitly. Moreover, its 1D array target and design constraints as well as heuristic search strategy can counterbalance for its complexity on a modern network of workstations.

APPENDIX

SINGLE-PASS SEPARABLE 2D TRANSFORMATION

Do $m = 1$ to N
  Do $n = 1$ to N
    Do $i = 1$ to N
      Do $j = 1$ to N
        $y[i, j] = 0$

```
            Do k = 1 to N
                y[i, j] = y[i, j] + c[i, k] × x[k, j]
            EndDo k
        EndDo j
    EndDo i
    X[m, n] = 0
    Do p = 1 to N
        Y[p, n] = y[n, p]
        X[m, n] = X[m, n] + c[m, p] × Y[p, n]
    EndDo p
  EndDo n
EndDo m
```

After loop collapsing [18], the nested Do-loop above becomes

```
Do m = 1 to N
  Do n = 1 to N
    y[m, n] = 0
    Do k = 1 to N
        y[m, n] = y[m, n] + c[m, k] × x[k, n]
    EndDo k
    X[m, n] = 0
    Do p = 1 to N
        Y[p, n] = y[n, p]
        X[m, n] = X[m, n] + c[m, p] × Y[p, n]
    EndDo p
  EndDo n
EndDo m
```

REFERENCES

[1] S. Microelectronics, "Sti3220 motion estimation processor," Jan. 1994.

[2] T. Xanthopoulos and A. P. Chandrakasan, "A low-power dct core using adaptive bitwidth and arithmetic activity exploiting signal correlations and quantization," *IEEE J. Solid-State Circ.*, vol. 35, no. 5, pp. 740–750, 2000.

[3] 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, 1967.

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

[5] H. T. Kung, "Why systolic architectures?," *IEEE Comput.*, vol. 15, no. 1, pp. 37–46, 1982.

[6] C. E. Leiserson, *Area-Efficient VLSI Computation*, MIT Press, Cambridge, Massachusetts, 1983.

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

[8] S. K. Rao and T. Kailath, "Regular iterative algorithms and their implementation on processor arrays," *Proc. IEEE*, vol. 76, no. 4, pp. 259–282, 1988.

[9] D. I. Moldovan, *Parallel Processing: From Applications to Systems*, Morgan Kaufmann, San Mateo, CA, 1993.

[10] T. Komarek and P. Pirsch, "Array architectures for block matching algorithms," *IEEE Trans. Circ. Syst.*, vol. 36, no. 10, pp. 1301–1308, 1989.

[11] L. D. Vos and M. Stegherr, "Parameterizable vlsi architectures for the full-search block-matching algorithm," *IEEE Trans. Circ. Syst.*, vol. 36, no. 10, pp. 1309–1316, 1989.

[12] K.-M. Yang, M.-T. Sun, and L. Wu, "Vlsi designs for block-matching algorithms," *IEEE Trans. Circ. Syst.*, vol. 36, no. 10, pp. 1317–1325, 1989.

[13] 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, no. 5, pp. 407–416, 1995.

[14] S. Kittitornkun and Y. H. Hu, "Frame-level pipelined motion estimation array processor," *IEEE Trans. Circuit Syst. Video Technol.*, vol. 11, no. 2, pp. 248–251, 2001.

[15] S. Kittitornkun and Y. H. Hu, "Reconfigurable processor array synthesis," in *Int. Conf. Parallel and Distributed Computing, Applications, and Technologies*, 2001.

[16] J. Fritts, W. Wolf, and B. Liu, "Understanding multimedia application characteristics for designing programmable media processors," vol. 3655, pp. 2–13. The International Society for Optical Engineering, 1998.

[17] Z. Wu and W. Wolf, "Design study of shared memory in vliw video signal processor," in *Proc. Int. Conf. Parallel Architectures and Compilation Techniques*, pp. 52–59. 1998.

[18] M. J. Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley, Redwood City, CA, 1996.

[19] P. Z. Lee and Z. M. Kedem, "Synthesizing linear array algorithms from nested for loop algorithms," *IEEE Trans. Comput.*, vol. 37, no. 12, pp. 1578–1598, 1988.

[20] W. Shang and J. A. B. Fortes, "Time optimal linear schedules for algorithms with uniform dependencies," *IEEE Trans. Comput.*, vol. 40, no. 6, pp. 723–742, 1991.

[21] *Condor high throughput computing*, http://www.cs.wisc.edu/condor/.

[22] C. H. Hsieh and T. P. Lin, "Vlsi architecture for block-matching motion estimation algorithm," *IEEE Trans. Circuit Syst. Video Technol.*, vol. 2, no. 2, pp. 169–175, 1992.

[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, 2001.

[24] Xilinx, "Virtex 2.5v field programmable gate arrays," May 2000.

[25] R. D. Turney, C. Dick, D. B. Parlour, and J. Hwang, *Modeling and implementation of dsp fpga solutions*, Xilinx.

[26] Xilinx, "Virtex-ii 1.5v field programmable gate arrays," Jan. 2000.

[27] V. K. P. Kumar and Y.-C. Tsai, "Synthesizing optimal family of linear systolic arrays for matrix computations," in *Proc. Int. Conf. Systolic Arrays*, pp. 51–60. 1988.

[28] K. G. Ganapathy and B. Wah, "Optimal design of lower dimensional processor arrays for uniform recurrences," in *Int. Conf. Application Specific Array Processors*, pp. 636–648. 1992.

[29] S. Dutta, K. J. O'Connor, W. Wolf, and A. Wolfe, "A design study of a 0.25-μm video signal processor," *IEEE Trans. Circuit Syst. Video Technol.*, vol. 8, no. 4, pp. 501–519, 1998.

[30] W. Wolf, "Alternative architectures for video signal processor," in *Proc. IEEE CS Workshop VLSI*, pp. 5–8. 2000.

[31] J. Sanchez and A. Gonzalez, "Modulo scheduling for a fully-distributed clustered vliw architecture," in *Proc. 33rd ann. IEEE/ACM Int. Symp. Microarchitecture*, pp. 124–133. 2000.

[32] R. Leupers, "Instruction scheduling for clustered vliw dsps," in *Proc. Int. Conf. Parallel Architectures and Compilation Techniques*, pp. 291–300. 2000.

[33] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Understanding multimedia application characteristics for designing programmable media processors," *J. Comput. Lang.*, vol. 3655, pp. 2–13, 1981.

[34] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proc. 27th Ann. Workshop on Microprogramming*, pp. 63–74. 1994.

Surin Kittitornkun received his B. Eng (2nd Honor) from King Mongkut's Institute of Technology Ladkrabang (KMITL), Bangkok, Thailand in 1991. He received M. Eng and Telecom Finland Prize for academic excellence from Asian Institute of Technology (AIT), Bangkok, Thailand in 1995. He is currently a Ph.D. Candidate at Department of Electrical and Computer Engineering, University of Wisconsin, Madison. His research interests include VLSI architecture for digital signal/image processing, and FPGA/reconfigurable computing. He was a summer intern with Broadband Wireless System Department, Motorola, Inc., Schaumberg, IL in 1998.

Yu Hen Hu received a BSEE degree from National Taiwan University, Taipei, Taiwan, ROC in 1976. He received MSEE and Ph.D. degrees in Electrical Engineering from University of Southern California, Los Angeles, California in 1980 and 1982, respectively. From 1983 to 1987, he was an assistant professor of the Electrical Engineering Department of Southern Methodist University, Dallas, Texas. He joined the Department of Electrical and Computer Engineering, University of Wisconsin, Madison, Wisconsin in 1987 as an assistant professor (1987–1989), and is currently an associate professor. His research interests include multimedia signal processing, artificial neural networks, fast algorithms and design methodology for application specific microarchitectures, as well as computer aided design tools for VLSI using artificial intelligence. He has published more than 150 journal and conference papers in these areas. He is a former associate editor (1988–1990) for the IEEE Transaction of Acoustic, Speech, and Signal Processing in the areas of system identification and fast algorithms. He is currently associate editor of Journal of VLSI Signal Processing. He is a founding member of the neural network signal processing technical committee of IEEE signal processing society and served as chair from 1993-1996. He is a former member of VLSI signal processing technical committee of the signal processing society. Currently he serves as the secretary of the IEEE signal processing society (1996–1998). Dr Hu is a fellow of the IEEE.