Optimizing General Design Objectives in Processor-Array Design *

Kumar N. Ganapathy and Benjamin W. Wah
Coordinated Science Laboratory
University of Illinois, Urbana-Champaign
Urbana, IL 61801.
e-mail: {kumar,wah}@manip.crhc.uiuc.edu

Abstract

In this paper we present an improved search procedure for the General Parameter Method (GPM) [1]. Our procedure maps uniform dependence algorithms to application-specific processor arrays (PAs). It can optimize general design objectives with certain non-monotonicity properties, *i.e.*, those that do not increase monotonically with the parameters. An example of such an objective is the minimization of the total completion time, including load and drain times. In contrast, earlier design methods can only deal with monotonic objectives. We present results for the matrix-product problem using this search technique; application of the technique on the transitive-closure problem is presented elsewhere [2]. We also show that the parameters in GPM can be expressed in terms of the schedule vector $\vec{\Pi}$ and allocation matrix **S** in the popular dependence-based method (DM), thereby allowing GPM to be used in DM for finding optimal designs for uniform dependence algorithms.

1 Introduction

Many applications of digital signal processing, medical imaging, digital communications and control are characterized by repeated execution of a small number of computationally intensive operations. To meet performance requirements of these applications, it is often necessary to dedicate hardware with parallel processing capabilities to these specialized operations. Processor arrays, due their structural regularity and consequent suitability for VLSI implementation, are frequently used for this purpose.

*Research Supported by Joint Services Electronics Program contract N00014-90-J-1270, National Science Foundation grant MIP 92-18715, and an IBM graduate fellowship grant.

Proc. IEEE International Parallel Processing Symposium, April 1994.

This paper presents new results for the systematic mapping of uniform dependence algorithms into *application-specific processor arrays* (PAs). It extends our previous approach, called the General Parameter Method (GPM), for synthesizing PAs [3, 1]. Our main contribution here is an improved search method for optimizing general objectives that may vary non-monotonically with the parameters. In contrast, earlier search methods can only deal with objective functions that increase monotonically with the parameters. Hence, our proposed framework degenerates to our earlier search method [1] in case of monotonic objectives, but can also optimize more general non-monotonic design objectives. We present new designs in Section 5 that optimize computation time as well as completion times (the latter include load and drain times). These designs have not been found before.

There has been a lot of research in developing a design method to map uniform dependence algorithms to PAs. Most of these methods are based on the dependence method (DM) [4]. In DM, the mapping of an algorithm to a PA is characterized by a *linear mapping matrix* $\mathbf{T} = \begin{bmatrix} \vec{\Pi} \\ \mathbf{S} \end{bmatrix}$, where $\vec{\Pi}$ is the schedule vector and **S** is the allocation matrix. The design of the PA is then equivalent to determining the elements of **T**. This general representation of a feasible design as a particular mapping matrix allows DM to be applied to uniform as well as non-uniform recurrences. However, in DM, the generality in representation leads to large search spaces for optimal designs, as the optimization problem is posed as an integer programming problem [4, 5]. To overcome this complexity, feasible designs in DM are generally heuristic and are found by first specifying a "good" allocation matrix, and then determining the schedule vector that minimizes the computation time. In contrast, GPM is restricted to uniform recurrences, but is able to find optimal designs for general user-specified (possibly non-monotonic) objectives using efficient search techniques.

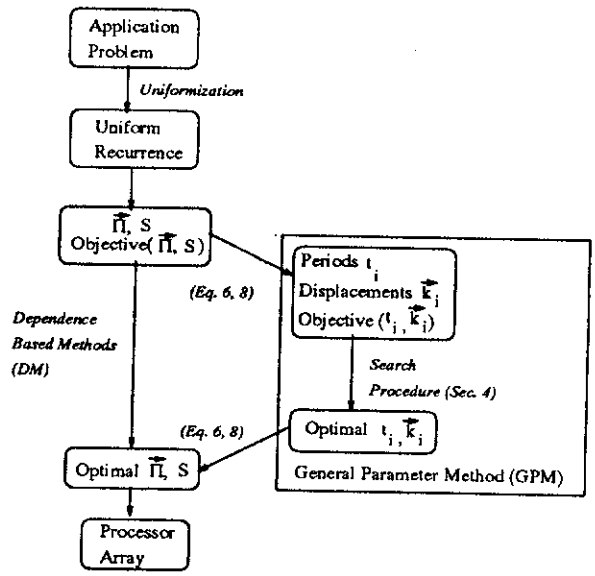

Figure 1: Using GPM to find optimal designs in DM.

GPM can be used as an intermediate step or as a black box to find optimal designs in DMs. Designers of PAs familiar with DM can utilize GPM to obtain better designs in terms of $\vec{\Pi}$ and $\mathbf{S}$ as depicted in Figure 1. After defining the objective (possibly non-linear and non-monotonic) in terms of $\vec{\Pi}$ and $\mathbf{S}$, the designer converts the objective in terms of the GPM parameters using (6) and (8). GPM is then used to generate optimal PAs efficiently for the given objective. The solutions obtained by GPM are then converted to $\vec{\Pi}$ and $\mathbf{S}$ in DM using (6) and (8) again. This step involves solving two sets of simultaneous equations for $\vec{\Pi}$ and $\mathbf{S}$ from the periods and displacements in GPM.

2 Algorithm Model

Affine dependence algorithms can be used to model a large number of applications in digital signal/image processing and other scientific applications. Affine dependence algorithms can be represented by the following Fortran-like nested loops, which are equivalent to systems of affine recurrence equations (AREs).

$$\begin{aligned}&\text{DO } (j_1 = l_1, u_1 \ ; \ j_2 = l_2, u_2 \ ; \ \cdots \ ; \ j_n = l_n, u_n)\\ &\quad S_1(\vec{J}) \ ;\\ &\quad \vdots\\ &\quad S_t(\vec{J}) \ ;\\ &\text{END}\end{aligned}$$

The column vector $\vec{J} = [j_1, j_2, \cdots, j_n]^T$ is the index vector (or index point). $S_i(\vec{J})$, $i = 1, \cdots, t$, are t assignment statements in iteration $\vec{J}$ having the form

$$Z_i(y(\vec{J})) = \phi\left[Z_1(x_1(\vec{J})), \cdots, Z_r(x_r(\vec{J}))\right], \qquad (1)$$

where $1 \le i \le r$. In (1), all loop bounds l_i and u_i are affine functions of loop variables $j_1, \ldots, j_{i-1}$, and indexing functions $y()$ and $x_k()$, $k = 1, \cdots, r$, are affine functions of the form $\mathbf{A}\vec{J} + \vec{d}$. If iteration $\vec{J}$ depends on iteration $\vec{J'}$, then this dependence can be described by a dependence vector $\vec{d} = \vec{J} - \vec{J'}$. The dependencies in the algorithm can be shown by a dependence graph (DG) over an n-dimensional polyhedral domain, where nodes are labeled by index vectors that correspond to the operations inside the nested loops, and arcs correspond to the loop-carried dependencies.

Uniform dependence algorithms or uniform recurrence equations (URE) form a sub-class of AREs, where the indexing functions $y()$ and $x_k()$ are of the form $\vec{J} - \vec{d}$ (matrix $\mathbf{A}$ is the identity matrix now), and $\vec{d}$ is a constant vector of n elements.

There exist many techniques to transform AREs to UREs where the affine dependencies are "uniformized." The basic idea is to select a few basic uniform dependence vectors such that all affine dependencies of the ARE can be expressed as non-negative integer linear combinations of the basis vectors.

In this paper, we focus on algorithms that can be modeled as UREs as well as AREs that can be uniformized. Hence, the starting point of our mapping assumes a convex polyhedral domain and a set of constant dependence vectors collected into a matrix called the dependence matrix D.

Example 1. The following are the uniformized recurrences for the product of two $N \times N$ matrices $\mathbf{A}$ and $\mathbf{B}$ to yield $\mathbf{C}$. In this example, $n = r = 3$.

$$\begin{aligned} C(i,j,k) &= C(i,j,k-1) + A(i,j,k) \times B(i,j,k) && (2)\\ A(i,j,k) &= A(i,j-1,k) && (3)\\ B(i,j,k) &= B(i-1,j,k) && (4)\end{aligned}$$

The boundary conditions are $A(i,0,k) = \mathbf{A}[i,k]$, $B(0,j,k) = \mathbf{B}[j,k]$, $C(i,j,0) = \mathbf{C}[i,j]$. The three dependencies are $(0,0,1)^t$, $(0,1,0)^t$, and $(1,0,0)^t$.

3 General Parameter Method

In GPM, the characterization of the behavior, correctness, and performance of a PA is defined in terms of a set of scalar and vector parameters. The crux of

GPM is the characterization of the behavior, correctness, and performance of a PA by a set of vector and scalar parameters. The details of GPM can be found in the references [3, 1]. We summarize the parameters and constraints in GPM below.

Parameter 1: Periods. The periods capture the time between execution of the source and sink index points of a dependence vector. Suppose the time at which an index point $\vec{I}$ (defined for the uniform recurrence equation) is executed is given by function $\tau_c(\vec{I})$. The period of computation t_j along dependence direction $\vec{d}_j$ is defined as

$$t_j = \tau_c(\vec{I} + \vec{d}_j) - \tau_c(\vec{I}), \quad j = 1, 2, \cdots, r. \quad (5)$$

The number of periods defined is equal to r, the number of dependencies in the algorithm. In terms of DM, t_j satisfies the following equation.

$$t_j = \vec{\Pi}\,\vec{d}_j\ , \quad (6)$$

where $\vec{\Pi}$ is the schedule vector in DM.

Parameter 2: Velocities. $\vec{V}_j$, the velocity of a datum, is defined as the directional distance passed during a clock cycle. Since *processing elements* (PEs) are at unit distance from their neighbors, and buffers (if present) must be equally spaced between PEs, the magnitude of velocity must be a rational number of the form i/j where i, j are integers and $i \leq j$ (to prevent broadcasting). This implies that in j clock cycles, x propagates through i PEs and $j - i$ buffers. All tokens of the same variable have the same velocity (both speed and direction) which is constant during the execution in the PA. The total number of velocity parameters is r (one for each variable), and each velocity is an m-element vector, where m is the dimension of the PA. Hence, $\vec{V}_j$ is given by

$$\vec{V}_j = \frac{\vec{k}_j}{t_j}, \quad j = 1, 2, \cdots, r, \quad (7)$$

where $\vec{k}_j$ is the (vector) distance between the execution locations of the source and sink index points of dependence vector $\vec{d}_j$. In the notation of DM, $\mathbf{S}$, the allocation matrix, is related to $\vec{k}_j$ and $\vec{d}_j$ as follows.

$$\vec{k}_j = \mathbf{S}\,\vec{d}_j\ . \quad (8)$$

Parameter 3: Spacings. Consider variable Ω_i pipelined along dependence vector $\vec{d}_i$, $1 \leq i \leq r$. The directional distance in the processor space from token $\Omega_i(\vec{I} - \vec{d}_j)$ to $\Omega_i(\vec{I} - \vec{d}_i)$ is defined as spacing parameter $\vec{S}_{i,j}$. Since there are r variables Ω_i, $1 \leq i \leq r$, each associated with dependency $\vec{d}_i$, $1 \leq i \leq r$, there are $r - 1$ non-trivial spacing parameters for each variable and one trivial spacing parameter, $\vec{S}_{i,i} = \vec{0}$. Each spacing parameter $\vec{S}_{i,j}$ is an m-dimensional vector, where m is the dimension of the PA. A total of $r(r-1)$ non-trivial spacing parameters are defined. In the notation of DM (based on (7), (8), and (12)),

$$\vec{S}_{i,j} = \mathbf{S}\,\vec{d}_j - \frac{\vec{\Pi}\,\vec{d}_j}{\vec{\Pi}\,\vec{d}_i}\mathbf{S}\,\vec{d}_i. \quad (9)$$

The total number of parameters defined is $r \times (r+2)$, of which r of them are periods (scalars); the remaining $r^2 + r$ are m-dimensional vectors, of which r of them are velocities and r^2 are spacings (r of these spacings are trivially zero). Assignment of values to these parameters defines a specific PA with a particular number of processors, buffers, and data-input pattern.

Example 2. For (2), there are 3 periods t_1, t_2, t_3 and 3 velocities $\vec{V}_1$, $\vec{V}_2$, $\vec{V}_3$. There are 9 spacing parameters $\vec{S}_{i,j}, i, j = 1, 2, 3$, of which $\vec{S}_{i,i} = \vec{0}$. For instance, for variable C, $\vec{S}_{1,2}$, $\vec{S}_{1,3}$ define distances between $(\mathbf{C}(i,j) \rightarrow \mathbf{C}(i,j+1))$, $(\mathbf{C}(i,j) \rightarrow \mathbf{C}(i+1,j))$, respectively.

4 Design Method

The design of a feasible PA is equivalent to choosing an appropriate set of parameters that satisfy the constraints imposed by dependency and application requirements. The search for the "best" design can be represented by the following optimization problem.

$$\textbf{Minimize} \quad b(N, t_1, \ldots, t_r, \vec{k}_1, \ldots, \vec{k}_r) \quad (10)$$

$$\textbf{Subject To:} \quad \begin{cases} 1 \leq t_i, i = 1, \ldots, r, \\ 0 \leq \left|\vec{k}_i\right| \leq t_i, i = 1, \ldots, r \\ \text{constraints in (12), (13), (14),} \\ \#PE \leq \#PE^{UB} \text{ and } T_c \leq T_c^{UB}. \end{cases} \quad (11)$$

The constraints on the parameters for any feasible lower-dimensional PA are listed below.

1. All data tokens used in computing the result at an index point must arrive at a PE simultaneously.

$$\vec{V}_i\, t_i = \vec{V}_j\, t_i + \vec{S}_{j,i}, \quad i, j = 1, 2, \cdots, r. \quad (12)$$

2. Linearly dependent dependence vectors must satisfy the following constraint:

$$\vec{T} \cdot \mathbf{N} = 0, \text{ and } \mathbf{K}\,\mathbf{N} = 0\,, \tag{13}$$

where g is the rank of dependency matrix $\mathbf{D}$, and $\mathbf{N}$ is the Null Space of $\mathbf{D}$.

3. Data-input conflicts (two tokens sent into a boundary PE simultaneously) must be avoided. Let $\mathbf{S}_i' = \left[\vec{S}_{i,1}, \vec{S}_{i,2}, \ldots, \vec{S}_{i,g-1}\right]$ be a matrix of $g-1$ independent spacings. Further, let $L_k, U_k, k = 1, 2, \cdots, g-1$, be integers such that the positions of all tokens of variable Ω_i, relative to a common origin, are defined as $\sum_{k=1}^{g-1} \vec{S}_{i,k}\beta_k$, $L_k \leq \beta_k \leq U_k$. To avoid data-input conflicts, we need the following constraint.

$$\mathbf{S}_i'\,\vec{\alpha} \neq \vec{0}, \text{ where } \vec{\alpha} \neq \vec{0}, \text{ and } \alpha_i \in [(L_i - U_i), \cdots, (L_i + U_i)], 1 \leq i \leq g-1. \tag{14}$$

The first constraint in (11) ensures that all precedence constraints are satisfied. Since the target PA is systolic, $\left|\vec{k}_i\right|$ should not exceed t_i in order to prevent data broadcasting. Hence, the second constraint in (11) is introduced. The third constraint indicates that the recurrence is evaluated correctly by the PA satisfying dependency requirements, and be free of data-link and computational conflicts. The fourth constraint indicates user-defined bounds on T_c and $\#PE$ to be observed.

The optimal design for the formulation given by (10) and (11) is found by a search algorithm. Since, in general, the objective function is nonlinear, involving functions such as ceiling, floor, and maximum or minimum of a set of terms, it is difficult to describe a comprehensive algorithm that covers all possible cases. In the rest of this section, we first describe a pruning strategy used in our search algorithm. This is followed by a discussion on searches with objectives that are functions of T_c, T_{comp}, T_{drain}, and $\#PE$. We then present the search algorithm and show its application for special cases of optimizing T_c and $\#PE$.

4.1 Pruning Strategy

The search space defined by the constraints in (11) results in a worst-case complexity of

$$O\left(\sum_{i=1}^{g}(t_i^{max})^2\right) = O\left((T_{comp}^{seq})^{2g}\right), \tag{15}$$

where T_{comp}^{seq} is the time needed to process the recurrence sequentially, and t_i^{max} is the maximum value of period t_i such that the computation time $T_{comp} \leq T_{comp}^{seq}$. Since T_{comp} is the sum of periods t_i (to be shown in (24)), $t_i^{max} \leq T_{comp}^{seq}$. (15) is true because we iterate in the worst case all combinations of t_i and $\left|\vec{k}_i\right| \leq t_i$, $i = 1, \ldots, r$. Note that this search space in polynomial in terms of the parameters in GPM and the size of the URE to be evaluated.

To reduce this search space, we need to develop effective pruning strategies so that suboptimal designs do not have to be evaluated. In this section, we present one such strategy that prunes based on incumbent designs obtained in the search. Our pruning strategy takes the objective function b (assuming to be minimized) and decomposes it as follows.

$$b(N, t_1, \ldots, t_r, \vec{k}_1, \ldots, \vec{k}_r) = f\left(t_1, \ldots, t_r, \vec{k}_1, \ldots, \vec{k}_r, e(t_1, \ldots, t_r, \vec{k}_1, \ldots, \vec{k}_r)\right) \tag{16}$$

where N is not represented explicitly since it is a constant in the optimization. The decomposition is done in such a way that $e()$ [1] is a monotonic function of its variables, which may be a subset of $t_1, \ldots, t_r, \vec{k}_1, \ldots, \vec{k}_r$. The intuition behind this decomposition is as follows.

If the objective function $b(t_1, \ldots, t_r, \vec{k}_1, \ldots, \vec{k}_r)$ is a monotonic function of its variables, then the optimal value of the parameters can be found by enumerating combinations of values of variables from their smallest permissible values (given by (11)) until a feasible design that satisfies the constraints is found. Since $b()$ is monotonic, the first feasible design obtained is also the optimal design.

The above idea of enumerating values of a monotonic function can be extended to the general case of non-monotonic objective functions. This is done by first identifying $e()$, a monotonic component of the objective that can be enumerated efficiently. The search proceeds by enumerating designs so that values of $e()$ grow monotonically. (The combination of parameter values used in $e()$ are substituted into (11), and the constraint equations are solved to see if there exists a feasible design.) Whenever a feasible design is obtained, an upper bound on $e()$ is computed by setting variables in $b()$ that are not included in $e()$ to their extremum values. (This upper bound means that no optimal design will have an objective value whose monotonic component $e()$ is larger than the upper bound.) The search is then repeated, refining the upper bound each time a feasible design is found. It stops when the upper bound on $e()$ is smaller than or equal to $e()$ of the best feasible design.

[1] For notational ease, we denote functions without their arguments.

From the above description, it is clear that the search strategy is an adaptation of a general branch-and-bound search. The monotonic component $e()$ corresponds to the variables searched in the branch-and-bound process, and bounding is performed by evaluating the lower bound every time a feasible solution is found.

The above works well when the objective is relatively simple. For problems with complex objectives, the search may be inefficient because the lower bound may be loose or may be difficult to find. For such problems, rewriting the objective in terms of composite variables (expressed in terms of the primary variables $t_1, \ldots, t_r, \vec{k}_1, \ldots, \vec{k}_r$) can simplify the choice of the extremum values for variables other than those in $e()$. This is illustrated as follows.

Consider an objective expressed as a function of composite variables T_{comp}, T_{load}, T_{drain}, and $\#PE$ as follows.

$$B = b\left(T_{comp}, T_{load}, T_{drain}, \#PE\right). \tag{17}$$

It is easy to see that $T_{comp} = T_{comp}(t_1, \ldots, t_r)$ is monotonic with respect to the g periods $t_1, \ldots, t_r$. (An exact characterization is shown in (24).) Hence, we choose T_{comp} as the monotonic component of objective function $b()$ and enumerate the periods $t_1, \ldots, t_r$ in an increasing order from their smallest permissible values (unity).

T^{UB}_{comp} can be refined if $b()$ is monotonically increasing with T_{comp}, T_{load}, T_{drain} and $\#PE$. In this case, T^{UB}_{comp} can be obtained by setting $T_{load} = T_{drain} = 0$, $T_{comp} = T^{min}_{comp}$, and $\#PE = \#PE^{min}$ and by solving

$$\begin{aligned} B^{inc} &= b\left(T^{UB}_{comp}, T^{min}_{load}, T^{min}_{drain}, \#PE^{min}\right) && (18)\\ &= b\left(T^{UB}_{comp}, 0, 0, \#PE^{min}\right), && (19)\end{aligned}$$

where B^{inc} is the objective value of the current incumbent design. Hence,

$$T^{UB}_{comp} = b^{-1}\left(B^{inc}, T^{min}_{load}, T^{min}_{drain}, \#PE^{min}\right), \tag{20}$$

where $b^{-1}()$ is the inverse function of $b()$ that rearranges (19) to compute T^{UB}_{comp} in terms of known constants.

T^{UB}_{comp} can further be refined if $\#PE$ can be expressed as a function of $\left|\vec{k}_1\right|, \ldots, \left|\vec{k}_r\right|$. In this case, $\#PE$ is minimum when exactly one of the $\left|\vec{k}_i\right|$s is 1, and the rest of the $\left|\vec{k}_j\right|$s, $j \neq i$, are 0. (An exact characterization is shown in (25).)

For instance, let the objective function be

$$B = \left(T_{comp} + T_{load} + T_{drain}\right)^2 \times \#PE\ . \tag{21}$$

According to (19), we have

$$\begin{aligned} B^{inc} &= \left(T_{comp} + 0 + 0\right)^2 \times \#PE \\ \Longrightarrow\ T^{UB}_{comp} &= \sqrt{B^{inc}/\#PE^{min}} && (22)\end{aligned}$$

Similarly, if the objective function to minimize completion time T_c,

$$\begin{aligned} B &= T_c = T_{comp} + T_{load} + T_{drain} \\ \Longrightarrow\ T^{UB}_{comp} &= B^{inc} - \left(T^{min}_{load} + T^{min}_{drain}\right) \\ &= B^{inc} - (0 + 0) = B^{inc} = T^{inc}_c && (23)\end{aligned}$$

T^{UB}_{comp} is refined continuously as new incumbent designs are found in the search. The search stops when there is no combination of t_i, $i = 1, \ldots, r$, that satisfies $T_{comp} \leq T^{UB}_{comp}$.

A special case of the optimization is to find a design with the minimum computation time T_{comp} (not including load and drain times). This was done in our earlier work [1, 3]. Here, $T^{UB}_{comp} = B^{inc} = T^{inc}_{comp}$, and the first feasible design is the optimal design that minimizes T_{comp}.

4.2 Search Procedure

In this section, we present our search procedure for minimizing $b(\#PE, T_c) = b(T_{comp}, T_{load}, T_{drain}, \#PE)$ where T_{comp} is a function of $t_1, \ldots, t_r$; T_{load} and T_{drain} are functions of $t_1, \ldots, t_r, \left|\vec{k}_1\right|, \ldots, \left|\vec{k}_r\right|$; and $\#PE$ is a function of $\left|\vec{k}_1\right|, \ldots, \left|\vec{k}_r\right|$.

1. Choose g periods and g displacements to be unconstrained parameters. Without loss of generality, let these periods and displacements be t_i and $\vec{k}_i$, $1 \leq i \leq g$, respectively.
2. Initialize T^{UB}_{comp} to be T^{seq}_{comp}, the computation time required to evaluate the recurrence sequentially.
3. Set the values of all the g unconstrained periods t_i, $i = 1, \ldots, g$, to be unity.
4. Choose the magnitude of the g unconstrained displacements $\left|\vec{k}_i\right|$, $i = 1, \ldots, g$, to be zero.
5. Compute the values of the other dependent $r - g$ periods and $r - g$ displacements using the conditions in (13).
6. Compute T^{cur}_{comp} using the periods and displacements found, where T^{cur}_{comp} is the computation time (without load and drain times) required

for processing the recurrence. T_{comp}^{cur} is found by substituting the current values of t_i, $i = 1, \ldots, r$, in (10). (Note that the design may not be feasible at this time.) If $T_{comp}^{cur} > T_{comp}^{UB}$, then exit with the incumbent design.

7. Solve for the spacing parameters from (12).

8. Check for data-input conflicts using (12) and (14) on the spacing parameters; also, check whether the constraints on T_c and $\#PE$ are violated (Constraint 4 in (11)).

9. If the solution is not feasible, then increment one of the $\left|\vec{k}_i\right|$s and repeat Steps 5, 6, 7 and 8 until all $\left|\vec{k}_i\right|$s equal t_i, $i = 1, \ldots, r$. If all $\left|\vec{k}_i\right|$s equal t_i and no feasible design is found, then go to Step 10. If a feasible design is found, then go to Step 11.

10. Increment one of the periods such that T_{comp}^{cur} increases by the lowest possible value. Go to Step 4.

11. Compute B^{cur}, the objective value achieved by the current design found. If $B^{cur} < B^{inc}$, then set $B^{inc} = B^{cur}$, and compute T_{comp}^{UB} for the current design using (20). Increment one of the $\left|\vec{k}_i\right|$s and go to Step 5.

For a design that minimizes $\#PE$ alone, the search procedure described above needs to be changed. In this case, $e()$ should be defined as a function of $\left|\vec{k}_1\right|$, $\ldots$, $\left|\vec{k}_r\right|$, and the search should start iterating with the smallest combinations of $\left|\vec{k}_1\right|, \ldots, \left|\vec{k}_g\right|$.

5 Applications: 3-Dimensional Cube Graph Algorithms

3-dimensional Cube graph algorithms form an important fundamental class of problems in signal and image processing. In fact, most commercially available DSP microprocessors are evaluated on the number of multiply-and-accumulate operations they can perform per second. In this section we synthesize a linear PA for computing the product of two $N \times N$ matrices.

The recurrences describing the matrix product and parameter definitions are indicated in Examples 1 and 2. A total of 8 relevant parameters are defined: 3 periods t_1 ($\mathcal{C}$), t_2 ($\mathcal{A}$), t_3 ($\mathcal{B}$), 3 displacements $\vec{k}_1, \vec{k}_2, \vec{k}_3$, and 6 spacings. For a linear PA all the parameters are scalars.

We illustrate in this section five formulations of the optimization of PAs: a) T_{comp}-optimal designs without bound on $\#PE$, b) T_c-optimal designs without bound on $\#PE$, c) $\#PE$-optimal designs without bound on T_c or T_{comp}, d) optimal designs with specific bounds on T_{comp} or $\#PE$, and e) optimal designs with specific bounds on T_c or $\#PE$.

5.1 Performance Attributes

The attributes we are interested are T_{comp} (computation time), T_{load} (load time), T_{drain} (drain time), $\#PE$ (processor count), and T_c (completion time), where $T_c = T_{load} + T_{comp} + T_{drain}$. For the case of computing the matrix product in a linear PA, T_{comp}, $\#PE$, T_{load} and T_{drain} are stated below without proof due to space limitations.

$$T_{comp} = (N-1)(t_1 + t_2 + t_3) + 1, \quad (24)$$

$$\#PE = (N-1)\left(\left|\vec{k}_1\right| + \left|\vec{k}_2\right| + \left|\vec{k}_3\right|\right) + 1, \quad (25)$$

$$T_{load} = T_{load}(\mathcal{A}) + T_{load}(\mathcal{B}), \quad (26)$$

$$T_{drain} = T_{drain}(\mathcal{C}) \quad (27)$$

where $T_{load}(A)$ for non-stationary input **A** (*i.e.*, $\vec{k}_2 \neq 0$) is given by

$$T_{load}(\mathcal{A}) = 1 + (N-1)\frac{\left(\mathcal{G}(\vec{S}_{2,1}, -\vec{v}_2) + \mathcal{G}(\vec{S}_{2,3}, -\vec{v}_2)\right)}{|\vec{v}_2|} \quad (28)$$
$$+(N-1)\left\lceil \frac{t_3\left\{\mathcal{G}(\vec{k}_1, \vec{k}_2) + \mathcal{G}(\vec{k}_2, \vec{k}_2) + \mathcal{G}(\vec{k}_3, \vec{k}_2)\right\}}{\left|\vec{k}_2\right|} \right\rceil$$

where

$$\mathcal{G}(\vec{x}, \vec{y}) = \begin{cases} |\vec{x}| & \text{if } \vec{x}, \vec{y} \text{ are in opposite directions} \\ 0 & \text{otherwise} \end{cases} \quad (29)$$

$T_{load}(\mathcal{B})$ and $T_{drain}(\mathcal{C})$ for non-stationary variables **B** and **C** are similar to (29). If some $\vec{k}_i = 0$, for $i = 1, 2, 3$, then that input (or output) is stationary and preloaded into (or post-drained from) the PA.

Tables 1 and 2 show the optimal linear designs found by the search procedure of GPM. The objectives used are to minimize T_{comp} and T_c, respectively. In finding these designs, t_3 is incremented before t_1 or t_2 in Step 10 of the search procedure presented in Section 4 (refer to (24)). This is done as it increases T_{comp} by the smallest amount. Note that the designs were developed without bounds on $\#PE$s. In these tables, we list T_{load}, T_{comp}, T_{drain}, $\#PE$s needed, the equivalent $\vec{\Pi}$ and **S** in DM, and the CPU time used by the search procedure running on a Sun Sparcstation 10/30. Thus, by establishing the equivalence between DM and GPM, GPM serves as a powerful tool to find optimal designs in DM (Figure 1).

Table 1: T_{comp} or $\#PE$ optimal linear arrays for product of two $N \times N$ matrices ($T_c = (T_{load}, T_{comp}, T_{drain})$).

N	GPM: T_{comp}-Optimal Linear PA Designs							GPM: $\#PE$-optimal Designs)	
	Periods	Distances	Schedule	Allocation	Min T_{comp} Designs		SS10/30		
	(t_1, t_2, t_3)	$(\vec{k}_1, \vec{k}_2, \vec{k}_3)$	$\vec{\Pi}$	S	T_c	#PE	(sec.)	T_c	#PE
3	(1,1,2)	(0,−1,1)	(2,1,1)	(1,−1,0)	(5,9,4)	5	-	(4,11,3)	3
4	(1,1,3)	(0,−1,1)	(3,1,1)	(1,−1,0)	(10,16,6)	7	-	(7,19,6)	4
8	(1,3,3)	(0,−1,2)	(3,3,1)	(2,−1,0)	(43,50,23)	22	-	(23,71,22)	8
16	(1,3,4)	(0,−2,3)	(4,3,1)	(3,−2,0)	(69,121,90)	76	-	(87,271,86)	16
32	(1,5,5)	(0,−3,4)	(5,5,1)	(4,−3,0)	(208,342,351)	218	-	(343,1055,342)	32
64	(1,6,7)	(0,−5,6)	(7,6,1)	(6,−5,0)	(455,883,1391)	694	1	(1367,4159,1366)	64
100	(1,7,9)	(0,−6,7)	(9,7,1)	(7,−6,0)	(810,1684,3370)	1288	6	(3335,10099,3334)	100
200	(1,11,11)	(0,−9,10)	(11,11,1)	(10,−9,0)	(2434,4578,13415)	3782	99	(13335,40199,13334)	200
300	(1,12,14)	(0,−11,13)	(14,12,1)	(13,−11,0)	(4242,8074,30132)	7177	101	(30001,90299,30000)	300

Table 2: T_c-optimal linear PAs for product of two $N \times N$ matrices.

N	Periods	Distances	Schedule	Allocation	Min T_c Designs		SS10/30
	(t_1, t_2, t_3)	$(\vec{k}_1, \vec{k}_2, \vec{k}_3)$	$\vec{\Pi}$	S	$(T_{load}, T_{comp}, T_{drain})$	#PE	(sec.)
4	(1,2,2)	(0,−1,1)	(2,2,1)	(1,−1,0)	(7,16,6)	7	-
8	(1,1,5)	(0,−1,3)	(5,1,1)	(3,−1,0)	(22,50,23)	29	1
16	(1,3,4)	(0,−2,3)	(4,3,1)	(3,−2,0)	(69,121,90)	76	18
32	(1,4,7)	(0,−3,4)	(7,4,1)	(4,−3,0)	(167,373,351)	218	1161
64	(6,5,8)	(5,4,−7)	(8,5,6)	(−7,4,5)	(649,1198,531)	1009	176
100	(7,7,9)	(6,5,−8)	(9,7,7)	(−8,5,6)	(1249,2278,925)	1882	483
200	(9,8,14)	(8,7,−13)	(14,8,9)	(−13,7,8)	(3216,6170,2912)	5573	3400
300	(11,11,16)	(10,9,−15)	(16,11,11)	(−15,9,10)	(6061,11363,4935)	10167	10903

Note that the designs in Table 1 requires **C** to be stationary (as $\vec{k}_1 = 0$), and **A** and **B** moving. These design are found by optimizing T_{comp} and are not affected by the values of T_{drain}. We further note that the designs in Table 2 are based on optimizing T_c. As a result, they have less total completion time and more $\#PE$s than the corresponding designs in Table 1.

Table 2 shows that for N up to 35, the optimal designs for T_c have stationary output matrix **C**. For $N > 35$, the optimal designs to minimize T_c have moving input and output matrices. To arrive at this conclusion, we need to compute the lower and upper bounds of T_{drain} (T^{lb}_{drain} and T^{ub}_{drain}) when **C** is stationary. Given N^2 elements in the PA to be drained, we assume that each PE in the PA has 3 input and 3 output ports (one each for **A**, **B**, **C**). Hence, T^{lb}_{drain} can be estimated as $\lceil \frac{N^2}{3} \rceil$. To compute T^{ub}_{drain} we develop a heuristic drain schedule for elements of **C** when computation in the PA is completed. (It is possible to start draining elements of **C** even before computations in the PA are completed; however, the resulting lower and upper bounds will be much more complex and will be case dependent.) For $N \leq 35$, we found that designs with stationary **C** have smaller T_c even when T^{ub}_{drain} is used as compared to the best designs when **C** is moving. On the other hand, for $N > 35$, designs with moving **C** have smaller T_c as compared to designs with stationary **C** even when T^{lb}_{drain} is used. These lead us to conclude that **C** should be stationary for $N \leq 35$ and moving for $N > 35$.

It is important to point out that the objective used (whether to minimize T_{comp} or T_c) depends on the application. If the linear PA is used to evaluate a single matrix product, then minimizing T_c will be important. On the other hand, if the PA is used for pipelined evaluation of matrix products, then minimizing T_{comp} may be important.

If the objective is to minimize $\#PE$, then Theorem 1 characterizes the $\#PE$-optimal design.

Theorem 1. *The parameters* $(t_1, t_2, t_3) = (N, 1, 1)$ *and* $(\vec{k}_1, \vec{k}_2, \vec{k}_3) = (0, 0, 1)$ *result in a linear PA with a primary objective of minimizing the number of PEs, and a secondary objective of minimizing the completion time.*

Note that in the optimal-$\#PE$ designs, input **A** and output **C** are stationary (preloaded) with each PE containing one row of **A** and **C**. Examples of these minimum-$\#PE$ designs are shown in Table 1. From (6) and (8), the corresponding DM parameter for the minimum-$\#PE$ designs are $\vec{\Pi} = [1, 1, N]^t$ and $\mathbf{S} = [\mathbf{1}, \mathbf{0}, \mathbf{0}]^{\mathbf{t}}$.

As a comparison, Table 3 shows the values of $T_{load}, T_{comp}, T_{drain}$, and $\#PE$ for designs obtained by Lee and Kedem (LK) [5] and Shang and Fortes (SF) [4]. These designs are feasible ones that do not optimize T_c, T_{comp}, or $\#PE$. The following table presents the parameters of these designs as well as the corresponding GPM parameters.

Method	$\vec{\Pi}$	S	(t_1, t_2, t_3)	$(\vec{k}_1, \vec{k}_2, \vec{k}_3)$
LK	$[1, 2, N-1]^t$	$[1, 1, -1]^t$	$(N-1, 2, 1)$	$(-1, 1, 1)$
SF	$[1, N-1, 1]^t$	$[1, 1, -1]^t$	$(1, N-1, 1)$	$(-1, 1, 1)$

Table 3: Feasible designs found by DM ($T_c = (T_{load}, T_{comp}, T_{drain})$).

N	Designs by LK [5]		Designs by SF [4]	
	T_c	#PE	T_c	#PE
3	(7,11,9)	7	(7,9,5)	7
7	(19,55,73)	19	(67,49,13)	19
15	(43,239,393)	43	(379,225,29)	43
31	(91,991,1801)	91	(1771,961,61)	91
63	(187,4031,7689)	187	(7627,3969,125)	187
101	(301,10301,20001)	301	(19901,10201,201)	301
201	(601,40601,80001)	601	(79801,40401,401)	601
301	(901,90901,180001)	901	(179701,90601,601)	901

Comparing the results in Tables 1 and 2, we found, for instance, that for a problem of size 200, the T_c-optimal design is 5.44 times faster than the $\#PE$-optimal design in terms of completion time, and uses 27.87 times more PEs than the $\#PE$-optimal design. (The T_c-optimal design for $N = 200$ requires 12,298 time units and 5573 PEs, whereas the $\#PE$-optimal design requires 66,868 time units and 200 PEs.) A designer might be unwilling to settle for either the large number of PEs required in the minimum-time design or the long completion time of the minimum-processor design. In practice, there may be bounds on the number of processors or the completion time or both. In the following, we present optimal designs with bounds on T_c and $\#PE$.

Figure 2 shows how $\#PE$ varies with T_c for 2 different problem sizes: $N = 50$, and 100. The y-axis $\#PE$ is normalized by $\#PE^{max}$ (processor count of the minimum-T_c design), and the x-axis T_c is normalized by T_c^{max} (completion time of the minimum-processor design). This lets us compare different problem sizes uniformly on the same scale. It is also clear from Figure 2 that there are better alternatives than the time-optimal or $\#PE$-optimal designs.

Given the bound T_c^{ub} (*resp.* $\#PE^{ub}$) the designer can use Figure 2 to find the minimum $\#PE$ (*resp.* T_c) required and decide (possibly from a cost perspective) if it is acceptable. The designer can exploit the initial steep decline in the plots to choose an alternative design that trades performance for cost. For instance, the minimum $\#PE$ required for $N = 50$ drops by 43% for only a 15% increase in completion time.

If both T_c and $\#PE$ are bounded from above, then we draw a horizontal line across the graph for the desired bound on $\#PE$. The intersection between this line and the stepped curve represents the minimum T_c needed for any feasible design. If this minimum T_c is less than the desired T_c, then a feasible design can be obtained that represents the best design under both time and processor constraints.

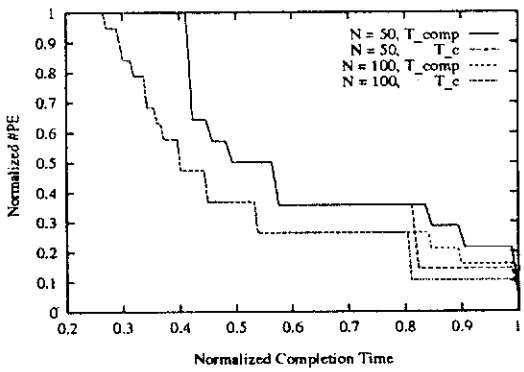

Figure 2: Performance trade-offs: Variation in $\#PE$ with T_c or T_{comp}. The plots are given for two problem sizes $N = 50$ and 100.

Figure 2 also shows the trade-offs when T_{comp} is bounded instead of T_c. These plots (denoted by T_{comp} in Figure 2) are obtained by computing the completion times T_c of designs that optimize T_{comp}. Given a bound T_c^{ub}, we can see that the number of processors obtained by minimizing T_c is less than or equal to the number of processors obtained by minimizing T_{comp}.

References

[1] K. N. Ganapathy and B. W. Wah, "Synthesizing optimal lower dimensional processor arrays," *Proceedings of International Conference on Parallel Processing*, Pennsylvania State University Press, Aug. 1992.

[2] J. A. B. Fortes, B. W. Wah, W. Shang, and K. N. Ganapathy, "Algorithm-specific parallel processing with linear processor arrays," in *Advances in Computers* (M. Yovits, ed.), Academic Press, 1994.

[3] K. N. Ganapathy and B. W. Wah, "Optimal design of lower dimensional processor arrays for uniform recurrences," *Proc. Application Specific Array Processors*, pp. 636–648, IEEE Computer Society Press, Aug. 1992.

[4] W. Shang and J. A. B. Fortes, "On mapping of uniform dependence algorithms into lower dimensional processor arrays," *IEEE Trans. on Parallel and Distributed Systems*, vol. 3, pp. 350–363, May 1992.

[5] P.-Z. Lee and Z. M. Kedem, "Mapping nested loop algorithms into multidimensional systolic arrays," *IEEE Trans. on Parallel and Distributed Systems*, vol. 1, pp. 64–76, Jan. 1990.