Automatic Mapping of Nested Loops to FPGAs

Uday Bondhugula
Dept. of Computer Science & Engg.
The Ohio State University
2015 Neil Ave. 395DL
Columbus OH 43210 USA
bondhugu@cse.ohio-state.edu

J. Ramanujam
Dept. of Electrical & Computer
Engg. and Center for Computation &
Technology
Louisiana State University
jxr@ece.lsu.edu

P. Sadayappan
Dept. of Computer Science & Engg.
The Ohio State University
2015 Neil Ave. 395DL
Columbus OH 43210 USA
saday@cse.ohio-state.edu

Abstract

This paper present a framework for automatic mapping of perfectly nested loops with constant dependences onto regular processor arrays, suitable for direct implementation on Field Programmable Gate Arrays (FPGAs). The problem is modeled as that of finding a suitable completion procedure for a full-rank linear transformation on the iteration space. The approach enables extraction of necessary degrees of communication-free and pipelined parallelism to optimize performance under the resource constraints of limited logic resources and I/O bandwidth available on an FPGA. The generation of control signals for the custom processing elements is also addressed. Examples of automatic derivation of parallel designs for some common nested loops are provided. Experimental results on the Cray XD1 show that an FPGA-based matrix-multiplication design obtained using the framework attains significant speedup on the XD1's attached FPGA, when compared to execution on the XD1 CPU.

Categories and Subject Descriptors C.1.3 [*Processor Architectures*]: Other Architecture Styles—Adaptable architectures; D.3.4 [*Programming Languages*]: Processors—Compilers, Optimization

General Terms Algorithms, Design, Performance

Keywords FPGA, resource constraints, regular processor arrays, FPGA compilation, nested loops, linear transformation, scheduling, control signals

1. Introduction

Field Programmable Gate Arrays (FPGAs) are user-programmable VLSI devices comprised of configurable logic blocks and a configurable interconnect, allowing the user to "field program" the device any number of times. Modern FPGAs have a large amount of configurable resources that enable highly parallel designs. These designs can be adapted well to specific problems, leading to very efficient use of on-chip resources.

FPGAs have long been used in embedded image and signal processing applications. However, densities of FPGAs have currently reached a point where they are competing in performance with modern general-purpose microprocessors. For many applications, custom-built parallel FPGA designs are a number of times faster. Moreover, trends in peak performance of FPGAs [27] show this gap widening. All of these have led to vendors offering high performance systems that couple general-purpose microprocessors with reconfigurable application accelerators. Cray XD1 [8], SRC Mapstation, and SGI RASC are such systems.

Today, the most significant impediment to more widespread use of FPGA acceleration for high-performance computing is the effort required by the application designer to generate an FPGA-based implementation. The productivity of developing applications for FPGAs currently is extremely low compared to that for microprocessors. This is mainly due to the complexity involved in hardware design. Although a number of applications have demonstrated high speedups with FPGAs for high-performance computing applications [28, 11, 31, 6], all designs were developed manually.

In this paper, we address the problem of automatically mapping perfectly nested loop computations onto FPGAs. The main contributions of this paper are as follows. We provide an algorithm to automatically map perfectly nested loops with constant dependences and arbitrary polyhedral bounds to linear or 2-D processor arrays on FPGAs. We also address the generation of control signals for the designs synthesized by the algorithm. Experimental results on the Cray XD1 show that these designs sustain a high speedup over execution on the CPU. Our framework is also applicable to other parallel co-processor architectures such as the ClearSpeed CSX600 [7] and MPSoCs.

PPoPP'07 March 14–17, 2007, San Jose, California, USA.
Copyright © 2007 ACM 978-1-59593-602-8/07/0003…$5.00

The rest of this paper is organized as follows: In Section 2, we provide a brief overview of FPGAs and the Cray XD1. In Section 3, we describe the problem addressed. We outline the solution approach in Section 4. Section 5 develops the algorithm for mapping. Generation of control signals and other aspects of the target processor array are discussed in Section 6. This is followed by experimental results on the Cray XD1 in Section 7.

2. Background

In this section, we provide a brief overview of modern FPGAs and the Cray XD1 system.

An FPGAs comprises various configurable elements and embedded blocks. At the lowest hardware level, an FPGA is made up of multiple configurable blocks connected to a configurable interconnect (Fig. 1(b). Configurable Logic Blocks (CLBs) provide functional elements for combinatorial and synchronous logic, including basic storage elements. Apart from CLBs, modern FPGAs comprise I/O blocks, Block RAM, embedded dedicated multipliers, and sufficient resources for routing and global clocking.

In the Virtex-II Pro series of FPGAs [29], each CLB includes four slices, and dedicated horizontal routing resources. Each slice is equivalent, and comprises two function generators, two storage elements, large multiplexers, arithmetic logic gates, a fast look-ahead carry chain, and a few other resources. Each function generator is configurable as a 4-input look-up table (LUT), as a 16-bit shift register, or as 16-bit distributed RAM. Each CLB has an internal fast interconnect and connects to a switch matrix to access general routing resources. Apart from CLBs, the Virtex family provides a number of dual-ported RAM blocks, allowing the user to store data on-chip, and create deeper and wider pipelines. Each block RAM resource is an 18Kb dual-ported RAM, programmable in various depth and width configurations.

Each compute blade on the Cray XD1 system (Fig.1(a)) contains two 64-bit AMD Opteron processors, an interconnect processor which provides two 2 GB/s RapidArray links to the switch fabric, and an application acceleration module [8]. The application acceleration module provides an FPGA, a programmable clock source, and four banks of Quad Data Rate II SRAM. Communication with the system is done from pinned buffers in the system memory's from and to which the FPGA can access data via DMA transfer. A peak bandwidth of 1.6 GB/s is available each way over the interconnect. The FPGA on the XD1 is a Xilinx Virtex-II Pro XC2VP50 that has about 23,000 slices, and can be clocked at a maximum of 200 MHz.

3. Problem

We consider perfectly nested loops with constant dependences and arbitrary polyhedral bounds. These need to be mapped to processor arrays on FPGAs in a way that parallelism is maximized under the resource constraints of fixed I/O bandwidth and logic resources available to the FPGA. An efficient way of providing control signals to the processor array should also be derived.

Interconnection. Arbitrary interconnection of PEs increases routing complexity, thereby affecting target clock rate directly. A linear array has the least connectivity while being regularly and locally connected, followed by a 2-D mesh. Also, loop bounds are sufficiently large compared to the number of processors that can be put on a chip that a smaller dimensional processor space is preferable. The target architecture we map to is a linear or a 2-d processor array with links in the forward direction or (0,1) and (1,0) directions, respectively. We show that our approach can map all nested loops with constant dependences to these without the need for backward communication. Modern FPGAs use an island-style hierarchical routing fabric that fits well with a 1-D or a 2-D processor array. Since the maximum space dimensionality we have is two, our framework accommodates multi-dimensional time.

There is no abstraction of a global shared memory on the FPGA. All movement has to be explicitly managed and data access conflicts need to be taken care of. Hardware elements have limited fan-out. To capture this, we also take into account read dependences in the framework we propose.

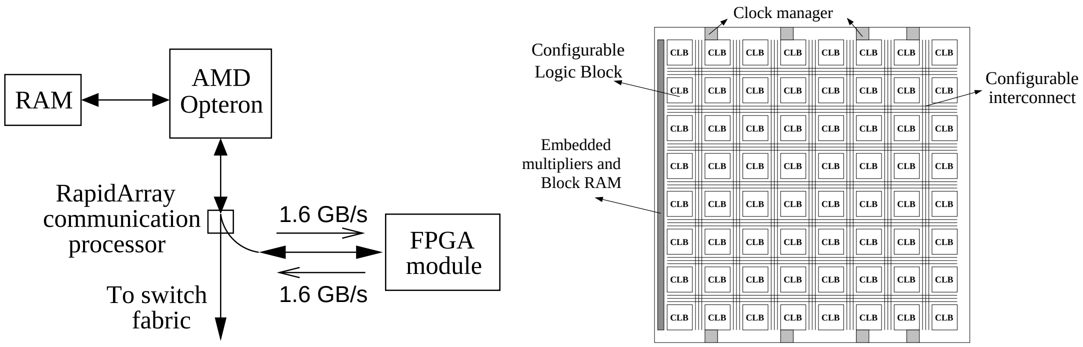
Near-neighbor communication on chip is inexpensive. Communication costs arising out of pipelined or fine-grained parallelism are thus not a concern. For FPGA-based designs, coarser granularity leads to more control and state ending up on the FPGA. A fine-grained parallel design shifts this overhead to software on the system making the design simpler to build, and is thus preferred.

LPGS tiling. Since the iteration space may be large along with the data set being processed, we tile the nested loops and accelerate computation for the inner set of loops on FPGAs keeping the outer tile-space loops in software. This technique is the well-known Locally Parallel Globally Sequential (LPGS) scheme used in the synthesis of fixed-size systolic arrays [21]. Each tile is executed in parallel on the FPGA and tiles are processed in sequence.

3.1 Control signals

The array of processors we map to are custom processors capable of performing computation specified in the loop body. These processors are not instruction-based and so cannot process a sequence of them; hence, the need to provide control signals to precisely inform when an iteration needs to be executed, along with information necessary to compute access functions to execute the iteration.

The cost of putting n processors along one degree of parallelism may not be the same as that of putting the same number along another degree. This is mainly due to overhead involved in providing control signals. Pipelined parallelism always incurs more control overhead than communication-



(a) The Cray XD1 system

(b) An FPGA

Figure 1. The Cray XD1 and an FPGA

free parallelism, as all processors along a dimension that has no communication often share control signals. This is explained more rigorously in Section 6.

4. Overview of solution approach

We find a non-singular transformation of the iteration space to a basis that can be readily inferred as processor dimensions (space) and time dimensions. The space loops are to be parallel, with at most near-neighbor communication. Multi-dimensional time schedules are allowed. To find the transformation, we present an algorithm that is a particular way of completing a non-singular transformation matrix, a general framework for which was proposed by Li and Pingali [18]. The algorithm systematically finds rows of the transformation matrix in a step-by-step manner, inferring necessary degrees of parallelism.

Consider a perfectly-nested loop nest with n levels of nesting. The *iteration space polytope* defines an n-dimensional set of points, characterized by a set of bounding hyperplanes. Each point in the iteration space is represented by a vector comprising indices of all loops from outermost to innermost. The dependences in the computation are represented by a matrix D, where each column defines a dependence vector. Let $D = \left(\vec{d}_1\ \vec{d}_2\ \ldots,\ \vec{d}_m\right)$. D includes flow, anti, and input/read dependences. Let R be the set of read dependences.

In the rest of this section, for convenience, we also treat D, R as sets of vectors instead of matrices wherever necessary to apply operations – set union ($\cup$) and set minus ($\setminus$) that convey dependence vectors being added to or removed from them. Let $D' = D \setminus R$.

Let T be a non-singular nxn square matrix. T is of rank n and can map the source iteration space one-to-one to the target iteration space. The goal is to find suitable coefficients, c_{ij}, for T.

$$
\begin{aligned}
T &= \begin{pmatrix} c_{11} & c_{12} & \ldots & c_{1n} \\ c_{21} & c_{22} & \ldots & c_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ c_{n1} & c_{n2} & \ldots & c_{nn} \end{pmatrix} \\
\vec{t} &= T\vec{s}
\end{aligned}
$$

Let row vector $r_i = (c_{i1} \quad c_{i2} \quad \ldots \quad c_{in})$

4.1 Communication-free parallelism

T extracts communication free parallelism through row r_i iff $r_i\vec{d}_j = 0$, for each $\vec{d}_j \in D$, i.e., r_i is orthogonal to all dependence vectors. The corresponding target loop in the transformed iteration space is thus an outer parallel loop and can be distributed across a dimension of processors without any communication along that dimension.

There exists communication-free parallelism iff the rank of the dependence matrix, D, is less than the dimensionality of the iteration space. Let c be the number of degrees of communication-free parallelism.

$$c = n - \mathrm{rank}(D)$$

Processors along a degree of communication-free parallelism can all start in parallel. Hence, the I/O bandwidth required is proportionate to the number of processors. All processors on the FPGA share a fixed I/O bandwidth available from the system. The aggregate bandwidth is the same even if we use more processors, unlike a general-purpose shared-memory or distributed memory parallel system. Extracting more than one degree of communication-free parallelism is thus not useful. Hence, placing enough processors along one degree of communication-free parallelism will be sufficient to utilize the entire I/O bandwidth available to the FPGA. Also, the number of processors that can be practically put along this dimension, being constrained by bandwidth, is generally small and not comparable to the loop bounds. Read dependences can thus be relaxed while finding communication-free parallelism: the resulting fan-out is small.

4.2 Pipelined parallelism

Any nested loop code with constant dependences has $n-c-1$ degrees of pipelined parallelism. Apart from the c loops that have communication-free parallelism, the rest of the $n-c$ loops can be made fully-permutable (by skewing), and have $n-c$ independent time partitioning solutions [20, 19]; the code can thus be mapped to $n-c-1$ dimensions of space with near-neighbor communication, with a one dimensional time schedule. Hence, a linear schedule with $n-c-1$ degrees of pipelined parallelism with near-neighbor communication can always be found. Alternatively, when $c \geq 1$, $n-c$ degrees of pipelined parallel space can be found with one of the dimensions of comm-free parallelism being used as time (serialized). Hence, as long as $n > c$, pipelined parallelism can be found. Pipelined parallelism does not use as much off-chip bandwidth as comm-free parallelism, and is thus preferable after one degree of comm-free parallelism has been found. The algorithm we propose in the next section extracts suitable number and kinds of these two forms of parallelism.

Interconnect communication. We use systolic techniques developed by Moldovan [21] in providing constraints concerning interconnection while completing the transformation matrix. Let L be an sxl matrix with its columns representing interconnection links of the processor array we are mapping to. As stated in Sec. 3, $s = 1$ or $s = 2$. The unit vectors represent communication links in the corresponding directions. For example, for a 2-D processor array:

$$L = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

5. Algorithm

From the discussion in the last section, we find that one and only one degree of communication free parallelism is preferable over pipelined parallelism. Hence, the order in which we extract these degrees is as follows: First, we search for a degree of communication-free parallelism, followed by a degree of pipelined parallelism, and the rest of the rows represent a multi-dimensional time schedule. If there is no comm-free parallelism, we extract two degrees of pipelined parallelism. In all cases, we extract two degrees of parallelism from loop nests with depth three or more, and one degree of parallelism from a 2-d loop nest.

1. **Communication-free parallelism**. If $c > 0$, there exists at least one degree of communication-free parallelism. We need to find a row vector that is orthogonal to all dependence vectors. As mentioned in Sec. 4.1, we suppress read dependences while finding comm-free parallelism. We thus have the following integer linear programming problem:

$$D' \leftarrow D \setminus R \tag{1}$$
$$r_1.\vec{d}_j = 0 \text{ for each } \vec{d}_j \in D' \tag{2}$$
$$\text{minimize } \sum_{i=1}^{n} c_{1i} \tag{3}$$

D' is used so that read dependences are ignored. To avoid the zero vector which is a trivial solution to the above, a polyhedral difference to remove the space where all vector elements are simultaneously zero can be used. This can be done by polyhedral libraries like PolyLib [1] or PPL (Parma PolyLib) [3]. Doing so leads to a union of many polyhedra and (3) is computed for each element of the union and the best is kept.

Once we find r_1, any components of read dependences along r_1 are to be ignored for communication or scheduling purposes. We project all read dependences in the space orthogonal to r_1, say U. We replace the read dependences in D by their projections in U. Any dependences that are parallel to r_1 are completely removed.

$$\begin{aligned} U &= I - r_1^T \left(r_1 r_1^T\right)^{-1} r_1 \\ R' &= U.R \\ D'' &= D \setminus R \cup R' = D' \cup R' \end{aligned}$$

Note that $r_1.R' = (r_1.U).R = \vec{0}^T$. Therefore, we have

$$r_1.D'' = \vec{0}^T \tag{4}$$

2. **Pipelined parallelism.** If $c = n$, there exist no degrees of pipelined parallelism: hence, after extracting one degree of communication-free parallelism, skip this step and go to Step 3 to find the rest of the rows that would be time dimensions.

If $c = 0$, we extract two degrees of pipelined parallelism (one in case of a 2-d loop nest). If $1 \leq c < n$, we extract exactly one degree of pipelined parallelism.

We ensure the following in the formulation: (1) all communication is done by links in the forward direction, (2) no more communication-free parallelism is found, and (3) the sum of the length of the communication paths that move data corresponding to dependences is minimized.

$$r_2 D \geq \vec{0}^T \tag{5}$$
$$\sum_{\vec{d}_j \in D''} r_2.\vec{d}_j > 0 \tag{6}$$
$$\text{minimize } \sum_{\vec{d}_j \in D''} r_2.\vec{d}_j \tag{7}$$

$r_2 D \geq \vec{0}$ makes sure that communication occurs only in the forward or upward direction. For r_2 to be independent of the first row, at least one of the dependences in

D'' should have a component along r_2. Since the first condition already enforces $r_2 D \geq \vec{0}^T$, condition (6) is sufficient. This condition automatically also avoids the zero vector. This formulation holds even if no degrees of comm-free parallelism were found, and r_2 was the first degree of pipelined parallelism. Once a time dimension is found in the next step, it would be evident that r_2 represents pipelined parallel space.

After steps 1 and 2 above, all space partitions have been found. Let these rows be Π. Let m'' be the number of columns (dependences) in D''. Let K be an lxm'' matrix such that its element $k_{ij} (\geq 0)$ is the number of times link i is used to move data corresponding to dependence $\vec{d_j}$. To compute K, all dependences in D'' need to be made use of. The non-negative constants in K can be obtained by solving:

$$\Pi D'' = LK \qquad (8)$$

$$\text{minimize} \sum_{i=1}^{l} k_{ij}, \;\; 1 \leq j \leq m'' \qquad (9)$$

Bound on the number of links required between neighboring PEs along dimension r: $\sum_{j=1}^{m''} k_{rj}$

3. **Finding time dimensions.** We need to find $n-2$ dimensions of time, one represented by each of the rows yet to be completed ($n-1$ in case of a 2-d loop nest). This step is repeated till all of those dimensions are found. Let this be the i^{th} one, say t_i.

   Let P be the matrix comprising rows of T found so far. Let E be the set of dependences that have not yet been carried by the time mappings so far. Initially, $E = D''$, since a valid multidimensional time schedule needs to carry all dependences in D''.

   For the source values due to the dependences to arrive from other PEs:

$$t_i.\vec{d_j} \geq \sum_{i=1}^{l} k_{ij}, \text{ for each } \vec{d_j} \in E$$

   To find the rest of the time dimensions that are independent of rows of P, we find vectors that have a strictly positive component in the sub-space orthogonal to the rows in P. Also, all dependences not carried so far should be lexicographically positive in the time dimensions being found henceforth. We solve the following:

$$Q \;\leftarrow\; I - P^T(PP^T)^{-1}P$$

$$t_i.\vec{d_j} \geq \sum_{i=1}^{l} k_{ij} \text{ for each } \vec{d_j} \in E \qquad (10)$$

$$t_i.\vec{d_j} \geq 0 \text{ for each } \vec{d_j} \in D'' \qquad (11)$$

$$Q.t_i^T > \vec{0} \qquad (12)$$

$$\text{minimize} \sum_{j=1}^{n} c_{ij} \qquad (13)$$

   Q represents the sub-space orthogonal to the rows of P. Note that $QP^T = \mathbf{0}$. In (11), for a valid time schedule, E on the R.H.S would have sufficed; however, to achieve full permutability, we use D'': this would allow us to readily tile and execute the transformed code under the LPGS scheme as explained later.

   Find all dependences carried by the time dimension just found, and remove them from E.

$$\begin{aligned} C &= \{\vec{d_j} \mid \vec{d_j} \in E,\; t_i.\vec{d_j} > 0\} && (14) \\ E &\leftarrow E \setminus C && (15) \end{aligned}$$

   Repeat step 3 until all dimensions of time are found.

Application of this algorithm stepwise to 1-D Jacobi and matrix-matrix multiplication can be found in Appendix A.1 and A.2 respectively.

5.1 Correctness of multi-dimensional schedule

We now prove that the time schedule obtained by the algorithm above is valid. Let $t_1, t_2, \ldots, t_k$ be the time transformation rows obtained, in that order. Let $\tau^T = \begin{pmatrix} t_1^T & t_2^T & \ldots & t_k^T \end{pmatrix}$ so that

$$T = \begin{pmatrix} \Pi \\ \tau \end{pmatrix}$$

We prove that $\tau.\vec{d_i} \succ \vec{0}$ for each $\vec{d_i} \in D''$.

Proof. At each of the repetitions of step 3, the algorithm finds a time row that maps all dependences to non-negative values (from (11)). Therefore, $\tau.\vec{d_i} \geq \vec{0}$ for each $\vec{d_i} \in D''$. We now need to show that $\tau.\vec{d_i} \succ \vec{0}$ strictly, i.e., all dependences in D'' have been carried by the schedule.

Let us assume to the contrary that there exists a $\vec{d_j}$ such that $\tau.\vec{d_j} = \vec{0}$. Since T is of full rank, $T.\vec{d_j} \neq \vec{0}$. This implies that $\Pi.\vec{d_j} \neq \vec{0}$, and since $\Pi.D'' \geq \mathbf{0}$ (from (4), (5)), there exists a row r_i in Π such that $r_i.\vec{d_j} > 0$.

$$\begin{aligned} r_i.\vec{d_j} > \vec{0} \;\; &\Rightarrow \;\; \exists\, k_{qj} \text{ such that } k_{qj} \geq 1 \quad \text{(from (8))} \\ &\Rightarrow \;\; t_1\vec{d_j} \geq 1 \quad \text{(from (10) since any } k_{ij} \geq 0) \\ &\Rightarrow \;\; \tau.\vec{d_j} \succ \vec{0} \end{aligned}$$

This is a contradiction to our earlier assumption. □

5.2 Summary

The non-singular transformation captures loop permutation, reversal, skewing, and scaling. Loop skewing captured in inequalities (5) and (11) makes sure that all components of the dependence vectors are positive in the transformed iteration space. Communication only along (0,1) and (1,0) is sufficient, and no backward links are required. The new iteration space is *fully permutable* and allows straightforward tiling under the LPGS scheme. For perfectly nested loops with constant dependences, such a transformation is always possible.

6. Control signals and optimizations

In this section, we discuss generation of control signals for processor arrays obtained from the algorithm described in the previous section.

If the space rows of T are Π, and the time rows τ, iteration $\vec{i}$ is to be executed at processor $\Pi.\vec{i}$ at time $\tau.\vec{i}$. Li and Pingali's code generation technique [18] generates code for nested loops under a non-singular transformation. Applying this to the transformation obtained by our algorithm would lead to code similar to shown in Fig. 6. The space loops (p_1, p_2) are either parallel or pipelined parallel with near-neighbor communication. The vector $(t_1, t_2, \ldots, t_k)$ represents the global time, and (p_1, p_2) is the processor's coordinate in the array. All access functions are in terms of $t_i's$ and $p_i's$. When T is unimodular, the transformed iteration space is dense. When T in not unimodular, loop bounds may involve floor or ceil operations, and points in the iteration space need to be skipped. However, to generate HDL code and control for the processor array, we view code generation in a different way.

```
for $t_1 = 1, N, 1$ do
    for $t_2 = 1, N, 1$ do
        ⋮
        for $t_k = 1, N, 1$ do
            for $p_1 = 1, N, 1$ do
                for $p_2 = t_1, t_1 + N, 1$ do
                    ...
```

Figure 2. Transformed code with space and time loops

Each processing element (PE) in the array has to be provided a signal that it uses to decide whether an iteration has to be executed. The corresponding iteration vector to compute access functions is also required. Since T is full-ranked, its inverse exists. From the transformation found in Sec. 4, a processor $\vec{p}$ executes an iteration at time $\vec{t}$ iff $T^{-1}[\ \vec{p}^T \quad \vec{t}^T\ ]^T$ is a valid iteration vector in the original iteration space polytope. Let $A\vec{i} \le \vec{b}$ describe the polytope of the original iteration space. Then, for processor $\vec{p}$ to be active at time $\vec{t}$:

$$A\left(T^{-1}\left(\vec{p}^T \quad \vec{t}^T\right)^T\right) \le \vec{b} \qquad (16)$$

Standard code generation techniques [2, 18] employing Fourier-Motzkin elimination [24] on the above set of inequalities, in which the time loops are eliminated first, would give bounds for each time component t_i as a function of $p_i's$ and t_j's where $1 \le j \le i-1$. A very naive approach would be for each PE to have a multi-dimensional time counter and logic to check these inequalities to determine whether it has to be active. The overhead is very high in this case, and infeasible if the core computation involves simple operations (for eg. non- floating-point). We provide a much more efficient scheme to distribute control signals to the PEs in an optimized fashion.

6.1 Control distribution

Let the k time loops of the transformed iteration space be t_1, t_2, $\ldots$, t_k. After Fourier elimination, the bounds for the i^{th} time dimension, t_i, can be written as:

$$\max\left(\ldots,\ \alpha_i p_1 + \beta_i p_2 + \sum_{j=1}^{i-1}\gamma_j t_j + c_i,\ \ldots\right) \le t_i \le \min\left(\ldots,\ \alpha'_i p_1 + \beta'_i p_2 + \sum_{j=1}^{i-1}\gamma'_j t_j + c'_i,\ \ldots\right) \qquad (17)$$

α's, β's, γ's and c's are constants. Let us assume for now that there is a single expression instead of a *max* of many expressions for the lower bound, and likewise for the upper bound. We remove this restriction later.

We associate each of the k time dimensions with two global PE controllers, one for the lower bound and the other for the upper bound, so that there are $2k$ controllers. Each of the controllers maintains a k-dimensional time counter and streams signals from a particular corner of the processor array. Since the lower and upper bounds of the time components are linear in p_i, control signals can be propagated from a PE with a specific delay to its neighbors without each PE having to test for time vectors it has to be active at. Note that since we can tile all loops (including time loops) in the transformed iteration space, each time dimension has to step through the same number of time points irrespective of arbitrary polyhedral bounds (the boundary tiles can be executed on the host). The signals for this tile are propagated to neighbors in the following way: The propagation delay any PE adds for the lower (or upper) bound signal of t_i is α_i (or α'_i) in the p_1 dimension, and β_i (or β'_i) in the p_2 dimension. The coefficients of p_i's in Eqn. 17 are thus the propagation delay factors in the corresponding dimensions. The signs of the co-efficients give the direction of propagation. The corner of the 2-D array where propagation starts (or the location of t_i's controller) also depends on the signs of α_i and β_i. For example, if both α_i and β_i are positive, the lower bound signal for t_i is propagated from the lower right corner, i.e., from PE(0,0). If $\alpha_i > 0$ and $\beta_i < 0$, it is propagated from the top left corner, i.e., PE(0,B_2-1), and so on. Note that higher dimensions of time have corresponding higher delay factors based on the product of the tile sizes of the lower dimensions.

Each PE has a $2k$-bit wide signal coming in from its neighbors that it propagates to neighbors on the opposite side of it in the corresponding dimension. The signs and magnitudes of α_i's and β_i's specify the direction and delay of propagation. The lower and upper bound signal registers at each PE are initialized to low and high respectively. A PE executes an iteration if and only if each of the $2k$ signals from its neighbors is high. Let (p_1^0, p_2^0) be the coordinates of the corner from where t_i's lower bound controller streams its signal, and (p_1^0', p_2^0') for the upper bound controller. Let L_i and U_i represent the lower and upper bounds for t_i in Eqn. 17.

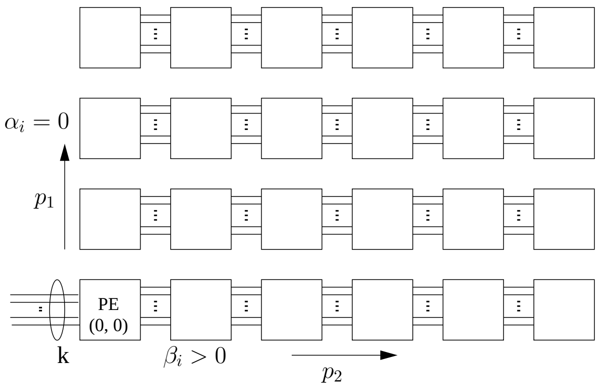


Figure 3. Control propagation

The signal streamed to the corner PEs by t_i's lower and upper bound controllers are respectively given by the booleans:

$$\begin{aligned} S_i &= \left(L_i\left(p_1 = p_1^0, p_2 = p_2^0\right) \ \le \ t_i\right) \\ S_i' &= \left(t_i \ \le \ U_i\left(p_1 = p_1^{0}\text{'}, p_2 = p_2^{0}\text{'}\right)\right) \end{aligned}$$

The propagation delays capture the redundant computation that would have been done at each PE to test for valid time vectors had the naive approach been followed. In the general case, the lower bound would be a *max* of several expressions, and the upper bound a *min*. Hence, multiple propagation delays are selected from, depending on the expression that is the maximum or minimum at that instant.

6.2 Common cases

A very common case is when $\alpha_i = \alpha_i'$, and $\beta_i = \beta_i'$. This implies that both the lower and upper bound controllers for t_i can be replaced by a single controller that propagates $S_i \wedge S_i'$.

If α_i or β_i is zero, all processors along the corresponding dimension need to be simultaneously given the same control signal, equivalent to a propagation delay of zero. Since p_2 is a degree of pipelined parallelism, $\beta_i \neq 0$. If p_1 is a degree of communication-free parallelism, α_1 is always zero, as all processors start concurrently along that dimension. Hence, the fan-out in sending control signals is no more than the number of processing elements along this dimension, which is very small (Sec. 4.1). As a result, processors along this dimension often end up sharing control that is automatically identified by synthesis tools (as resource sharing) and optimized. This justifies our initial explanation of communication-free parallelism providing more parallelism per slice utilized. Therefore, while determining tile sizes, we first determine the tile size for the degree of communication-free parallelism using the off-chip bandwidth constraint. Next, the number of processors that can fit based on slice or block RAM constraints determine the tile size associated with pipelined parallelism. Fig. 3 shows an example for a 2-D processor array with one degree of communication-free parallelism along (1,0), and a degree of pipelined parallelism along (0,1).

When T is non-unimodular, there is more overhead in generating control signals as the loop bounds may involve floor and ceiling operations, and processor activity may be periodic.

The proposed scheme moves as much control as possible from the PEs to the global controllers. All tile sizes along the time dimensions are set only to powers of two so that the global controllers can efficiently step through the entire time space. The computation of access functions required to execute an iteration can also be optimized in the same manner as described for control signals above. We skip details on this due to space constraints.

6.3 Example

Consider the following code as an example:

```
for $i = 1, N, 1$ do
    for $j = 1, N, 1$ do
        $a[i, j] = a[i, j-1] + a[i-1, j];$
```

One degree of pipelined parallelism is found with a linear schedule.

$$T = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \qquad T^{-1} = \begin{pmatrix} -1 & 1 \\ 1 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 1 \\ 1 \end{pmatrix} \ \le \ \begin{pmatrix} -1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} p \\ t \end{pmatrix} \ \le \ \begin{pmatrix} N \\ N \end{pmatrix}$$

After Fourier-Motzkin elimination, we have:

$$p + 1 \le t \le p + N$$

The access functions are: $(t-p, p)$, $(t-p, p-1)$, and $(t-p-1, p)$. Note that $\beta_1 = \beta_1' = 1$ here. Hence, we have a single controller located at the leftmost PE that streams a high signal during $1 \le t_1 \le N$ that is propagated across the linear array with a single cycle delay.

7. Experiments

We conduct all experiments on the Cray XD1. The FPGA on the XD1 is a Xilinx Virtex-II Pro XC2VP50 [29]. Current peak floating-point performance of FPGAs is only comparable to that of general-purpose microprocessors. Using limited precision exposes the importance of other aspects like overhead of control. We use limited precision (16-bit) matrix-matrix multiplication to measure and compare the performance of the FPGA design with that of the CPU.

Measurements for the general-purpose processor case were taken on a 2.2 GHz 64-bit AMD Opteron (as found on the XD1) with a 64 KB L1 data cache and a 1 MB L2 cache with a cache block size of 64 bytes. GCC 3.3 with "-O3" was used for compilation. The Cray User FPGA API provides functions to program the FPGA, write values and addresses to registers on the FPGA, taking care of virtual to

Module	Slices	
	B=16	B=32
Operator	8	
Control	104	105
PE	112	113
Global control	57	60
All PEs	7168	14464
Total (available)	23,316	

Table 1. Slice utilization: FPGA-MM

Size	CPU-MM	FPGA-MM	Speedup	FPGA-MM	Speedup
		B=16		B=32	
256	32.8ms	7.77ms	4.22	4.35ms	7.54
512	253ms	58.9ms	4.3	29.8ms	8.5
1024	2.01s	465ms	4.32	226ms	8.9
2048	16.1s	3.74s	4.3	1.8s	9.0
4096	128.6s	29.63s	4.34	14.2s	9.1
8192	1035.7s	237.33s	4.36	116.2s	8.91

Table 2. Measured performance comparison: CPU-MM vs FPGA-MM

physical address translation in the latter case. The general-purpose processor implementation for matrix-multiplication was tiled for the L2 cache. Copying to a contiguous buffer is done to avoid conflict misses. In the rest of this section, the CPU and FPGA implementations are referred to as CPU-MM and FPGA-MM respectively.

Applying the algorithm in Sec. 5, we find one degree of communication-free parallelism (p_1), and one degree of pipelined parallelism (p_2). Detailed derivation can be found in Appendix A.2.

```
for $t_1T = 1, N, B$ do
    for $p_2T = 1, N, B$ do
        for $p_1T = 1, N, 4$ do
            for $t_1 = t_1T, t_1T + B, 1$ do
                for $p_2 = p_2T, p_2T + B, 1$ do
                    for $p_1 = p_1T, p_1T + 4, 1$ do
                        $\langle$control + memory access$\rangle$
                        c[$p_1$, $t_1$-$p_2$]
                                += a[$p_1$, $p_2$]*b[$p_2$, $t_1 - p_2$]
                        $\langle$control + memory access$\rangle$
```

Figure 4. Parallel matrix multiplication for FPGA

Fig. 5 depicts the FPGA design. The processor array is similar to that in Fig. 3. Elements of matrix c are updated B times in the pipeline, once by each of the B PEs. The i^{th} column of PEs in Fig. 3 has the i^{th} column of a and the i^{th} row of b in local storage. The control signals in this case are simple: $\alpha_1 = 0$, $\beta_1 = 1$, corresponding to a pipelined start-up and stop in the p_2 dimension. The pipeline filling and draining was overlapped for successive tiles (Fig. 5).

Tile size determination. Four 16-bit elements can be read at a theoretical peak rate of 1.6 GB/s when the FPGA design is clocked at 200 MHz. We thus first set the tile size (or the number of processors) along p_1 to four. Then, based on the number of PEs that can fit on the FPGA, the tile size (or the number of processors) along the degree of pipelined parallelism is set. Note that a synthesis estimate for a single PE is required before the tile sizes are fixed. In our case, we find that the 4x32 processor array was the largest that could fit on the XC2VP50. The design thus uses the entire I/O bandwidth and slices even as larger amounts of both are available (with larger FPGAs).

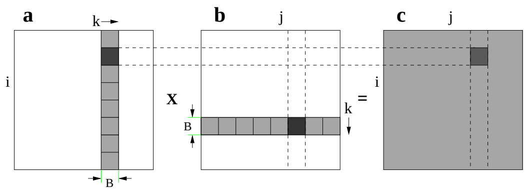


Figure 5. FPGA-MM

Resource utilization. Table 1 shows the slice utilization of the designs. Note that the multiply operation is mapped to the embedded dedicated multipliers available on the XC2VP50, and hence the low slice count of 8 for the operator. The scheme proposed in Sec. 6 moves as much control as possible from the PEs to the global controller. However, other housekeeping computations for scheduling local memory access (read and write) in PEs still incur a significant number of slices.

Results with the 4x16 processor array are also presented to observe scalability. The 4x16 processor array could be clocked at the full 200 MHz, while the 4x32 at 190 MHz. Layout transformation and compute-copy overlap, described in [5], are performed while integrating FPGA-MM with its software counterpart on the system. The outer loops t_1T, p_2T, p_1T are run in software. The FPGA design corresponds to loops t_1, p_2, and p_1 that can perform $4 * B$ operations in parallel.

As shown in Table 2, a speedup of 4.3 is measured using the 4x16 processor array, and speedup by a factor of 9 is obtained with the 4x32 array. Fig. 6 shows the GigaOps performance of the implementations. Though the CPU implementation we are comparing against incorporates blocking and copying, it is still not fully optimized. With a range of other optimizations [30] like register tiling, pipeline scheduling, and controlled unrolling, we estimate that a performance of up to 4.0 GigaOps can be obtained from the Opteron. This would still be less than the 9 GigaOps of the 4x32 array on the FPGA. Note that the largest FPGAs available currently are more than twice as large as the XC2VP50.

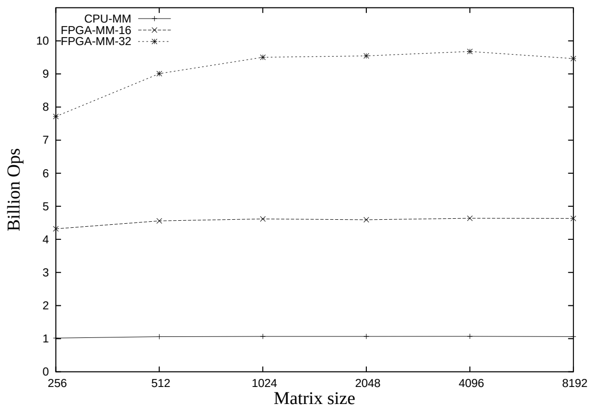


Figure 6. Performance: 16-bit matrix multiplication

8. Related work

A large body of literature exists on systolic synthesis from uniform recurrence equations [22, 21, 16]. Many studies also deal with mapping nested loops to multi-dimensional systolic arrays. Our work differs from them in that we develop space-time mappings considering resource constraints and control costs when implementing them on high-performance systems with FPGA accelerators.

Darte et al. [10, 9] address the problem of finding a tight linear schedule after tiling and clustering (LSGP) the virtual processor space. However, the problem of finding a good cluster shape is not addressed, i.e., the number of dimensions to cluster, and the choice of those dimensions. Clustering allows processing of larger tiles with the same number of processors, or perhaps fewer processors because of the added control complexity due to the juggling problem [10]. Our solution does not cluster the iteration space after tiling; instead, the tile size is determined by the number of processors that can fit on the FPGA.

Several projects address regular processor array synthesis within the polytope model. Among these are PICO-NPA [23], PARO [26, 4], MMAlpha [16] and Compaan [17]. At a high-level, many of these approaches (including ours) involve finding linear space and time mappings onto regular processor arrays. However, our approach to finding these mappings differs significantly from these works - both in how we find them, and the solution obtained. We use a step-by-step completion procedure using an LP formulation, and the mappings found are such that we have the right number and degrees of processor space, and a multi-dimensional time schedule, after which tile sizes are selected based on resource constraints. PICO [23] uses Darte's approach [10] for scheduling (compared earlier), after an orthogonal projection of the iteration space to the processor space. MMAlpha [16] takes a system of affine recurrence equations as input and derives systolic arrays for it. Both PARO [4] and MMAlpha use a more traditional approach of mapping to an $n-1$ dimensional processor space with a one-dimensional linear schedule when an n-dimensional iteration space is given. This has drawbacks mentioned in Sec. 3. PICO and PARO deal with VHDL code generation and hardware synthesis in much more detail than we do, and several of those ideas are applicable to our work. Finally, to the best of our knowledge, we are not aware of any of the above studies providing experimental results for an FPGA-accelerated system or resource utilization numbers on an FPGA that we can compare with. We are also not aware of any work that considers read dependences to fully address data access conflicts.

Lim and Lam [20] perform affine space-partitioning to find communication-free parallelism. If no degrees of communication-free parallelism are found, time-partitioning is done to find pipelined-parallelism. Our approach integrates space and time partitioning. Consider the following code. Lim's space partitioning would find two degrees of communication-free parallelism and terminate, which would lead to a design that can compute only at a rate that data can be fetched at. However, our approach would find one degree of communication-free parallelism, and one degree of pipelined parallelism which does not use any additional bandwidth but allows full use of resources on an FPGA ($p_1 = i$, $t_1 = j+k$).

```
for $i = 1, N, 1$ do
    for $j = 1, N, 1$ do
        for $k = 1, N, 1$ do
            $a[i, j] = a[i, j] + w[k] * b[i, j+k];$
```

Li and Pingali present a general completion procedure for non-singular transformations on iteration spaces[18]. We have adapted the procedure for use in the context of mapping loops onto FPGAs.

Feautrier[12] derives multi-dimensional time schedules for problems that do not admit an affine schedule. The code we consider always admits linear schedules, but multi-dimensional schedules are preferred in practice. Guillou et al. [14] address the control signal problem for multi-dimensional time schedules. However, there is significant cost per processing element associated with the solution. Our approach was motivated by the solution we manually developed in our earlier work [6].

So et al. [25] propose an automated hardware design space exploration approach (DEFACTO) that evaluates the effect of applying several loop transformations using synthesis estimation techniques. Our algorithm captures the effect different transformations would have on performance and is thus near-optimal without the need for further design space exploration. In situations where our algorithm generates multiple possible solutions, a design space exploration approach can be used to choose the best among these candidate designs.

Many projects have focused on translating high-level languages to hardware using various approaches. These include

SA-C, ROCCC [15], Streams-C [13], Impulse-C, Mitrion-C, Handel-C among others. Many of these tools either use a "soft instruction processor" approach or a "sea of gates" approach or employ intermediate formats for hardware compilation. Some of these tools do perform a limited set of loop transformations that are subsumed by our approach.

9. Conclusions and Future work

This paper has presented a framework to map nested loops with constant dependences to linear or 2-D processor arrays on FPGAs. The algorithm finds suitable degrees of communication-free and pipelined parallelism with a multi-dimensional time schedule. Parallelism is maximized under resource constraints by proper selection of tile sizes under the LPGS scheme. We also addressed the problem of efficiently providing control signals to the processor array. Experimental results with limited precision matrix multiplication on the XD1's FPGA show a speedup of 9x over execution on XD1's Opteron CPU.

We plan to extend this work to multiple statement iterations spaces including imperfectly nested loops and affine dependences. Existing work on affine per-statement mapping and scheduling can be used to obtain the affine counterpart of the procedure described here to solve a more general set of problems. We also plan to address hardware code (HDL) generation in detail.

Acknowledgments

We thank the Ohio Supercomputer Center, Springfield for use of their computing facilities. We would also like to thank the reviewers of the paper for their useful comments. This work is supported in part by the US Department of Energy's ASC program.

References

[1] PolyLib - A library of polyhedral functions. http://icps.u-strasbg.fr/polylib/.

[2] C. Ancourt and F. Irigoin. Scanning polyhedra with do loops. In *PPoPP'91*, pages 39–50, 1991.

[3] R. Bagnara, P. Hill, and E. Zaffanella. PPL: The Parma Polyhedra Library. http://www.cs.unipr.it/ppl/.

[4] M. Bednara and J. Teich. Automatic synthesis of FPGA processor arrays from loop algorithms. *Journal of Supercomputing*, 26(2):149–165, Sept. 2003.

[5] U. Bondhugula, A. Devulapalli, J. Dinan, J. Fernando, P. Wyckoff, E. Stahlberg, and P. Sadayappan. Hardware/software integration for FPGA-based all-pairs shortest-paths. In *IEEE FCCM'06*, Apr. 2006.

[6] U. Bondhugula, A. Devulapalli, J. Fernando, P. Wyckoff, and P. Sadayappan. Parallel FPGA-based all-pairs shortest-paths in a directed graph. In *IPDPS'06*, Apr. 2006.

[7] ClearSpeed Inc. ClearSpeed CSX 600, 2004. http://www.clearspeed.com/.

[8] Cray Inc. Cray XD1 whitepaper, 2005. http://www.cray.com/products/xd1/.

[9] A. Darte, S. Derrien, and T. Risset. Hardware/Software Interface for Multi-Dimensional Processor Arrays. In *IEEE ASAP*, pages 28–35, 2005.

[10] A. Darte, R. Schreiber, B. R. Rau, and F. Vivien. A Constructive Solution to the Juggling Problem in Processor Array Synthesis. In *IPDPS'00*, pages 815–822, 2000.

[11] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev. 64-bit floating-point FPGA matrix multiplication. In *ACM FPGA'05*, pages 86–95, 2005.

[12] P. Feautrier. Some efficient solutions to the affine scheduling problem. part II. multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, 1992.

[13] M. B. Gokhale, J. M. Stone, J. Arnold, and M. Kalinowski. Stream-Oriented FPGA computing in the Streams-C high level language. In *FCCM '00*, pages 49–56, 2000.

[14] A.-C. Guillou, P. Quinton, and T. Risset. Hardware synthesis for multi-dimensional time. *IEEE ASAP'03*, 00:40, 2003.

[15] Z. Guo, B. Buyukkurt, and W. Najjar. Input data reuse in compiling window operations onto reconfigurable hardware. In *LCTES '04*, pages 249–256, 2004.

[16] IRISA COSI. The MMAlpha environment. http://www.irisa.fr/cosi/ALPHA/mmalpha_english.html.

[17] B. Kienhuis, E. Rijpkema, and E. Deprettere. Compaan: Deriving process networks from Matlab for embedded signal processing architectures. In *8th International workshop on Hardware/software codesign*, pages 13–17, 2000.

[18] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. *International Journal of Parallel Programming*, 22(2):183–205, 1994.

[19] A. W. Lim, G. I. Cheong, and M. S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *ICS'99*, pages 228–237, 1999.

[20] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing*, 24(3-4):445–475, 1998.

[21] D. I. Moldovan and J. A. B. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Transactions on Computers*, 35(1):1–12, 1986.

[22] P. Quinton. Automatic synthesis of systolic Arrays from uniform recurrent equations. In *ISCA*, pages 208–214, 1984.

[23] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B. R. Rau, D. Cronquist, and M. Sivaraman. PICO-NPA: High-Level synthesis of non-programmable hardware accelerators. *J. VLSI Signal Process. Syst.*, 31(2):127–142, 2002.

[24] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1987. SchRI a 87:1 1.Ex.

[25] B. So, M. W. Hall, and P. C. Diniz. A compiler approach to fast hardware design space exploration in FPGA-based systems. In *PLDI'02*, pages 165–176, 2002.

[26] J. Teich, L. Thiele, and L. Zhang. Partitioning processor arrays under resource constraints. *Int. Journal on VLSI and Signal Processing Systems*, 17(1):5–20, 1997.

[27] K. Underwood. FPGAs vs. CPUs: Trends in peak floating-point performance. In *ACM FPGA'04*, 2004.

[28] K. D. Underwood and K. S. Hemmert. Closing the Gap: CPU and FPGA Trends in Sustainable Floating-Point BLAS Performance. In *IEEE FCCM'04*, Apr. 2004.

[29] Xilinx Inc. *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: complete data sheet*. Xilinx Inc., 2005.

[30] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. A. Padua, K. Pingali, P. Stodghill, and

P. Wu. A comparison of empirical and model-driven optimization. In *PLDI'03*, pages 63–76, 2003.

[31] L. Zhuo and V. K. Prasanna. High performance linear algebra on a reconfigurable supercomputer. In *Supercomputing*, Nov. 2005.

A. Appendix

A.1 1-D Jacobi

```
for $t = 1, N, 1$ do
    for $i = 2, N-1, 1$ do
        $a(t,i) = \frac{a(t-1,i-1)+a(t-1,i)+a(t-1,i+1)}{3}$;
```

$$D = \begin{pmatrix} 1 & 1 & 1 \\ 1 & -1 & 0 \end{pmatrix}$$

Step 1 We have no communication-free parallelism here. The algorithm extracts one degree of pipelined parallelism with a linear schedule.

Step 2 Pipelined parallelism

$$c_{11} + c_{12} \geq 0;\ c_{11} - c_{12} \geq 0;\ c_{11} \geq 0;\ 2c_{11} > 0$$

$$\text{minimize } c_{11}$$

$$c_{11} = 1,\ c_{12} = 0$$

$$L = (1),\ \ \Pi = (1\ \ 0)\ \Rightarrow\ K = (1\ 1\ 1)$$

Number of links required between adjacent PEs:

$$\sum_{j=1}^{3} k_{1j} = 3$$

$c_{12} = 1$ or $c_{12} = -1$ are also valid solutions.

Step 3 Time dimension(s)

$$Q = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$$

$$c_{22} > 0;\ c_{21} + c_{22} \geq 1;\ c_{21} - c_{22} \geq 1;\ c_{21} \geq 1$$

$$\text{minimize } \sum c_{2i}$$

$$c_{21} = 2,\ c_{22} = 1$$

$$\mathbf{T} = \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix} \qquad p_1 = t;\ \ t_1 = 2t + i$$

All iterations along $2t + i$ are executed in parallel.

$$\mathbf{T^{-1}} = \begin{pmatrix} 1 & 0 \\ -2 & 1 \end{pmatrix} \quad \Rightarrow \quad 2p_1 + 1 \leq t_1 \leq 2p_1 + B$$

A controller located near the leftmost PE streams a signal during $1 \leq t_1 \leq 3B$ that is shifted across the linear array with a propagation delay of 2 clock cycles.

A.2 Matrix-matrix multiplication

```
for $i = 1, N, 1$ do
    for $j = 1, N, 1$ do
        for $k = 1, N, 1$ do
            c[$i$, $j$] += a[$i$,$k$]*b[$k$,$j$];
```

Applying algorithm in Sec. 5, we find one degree of comm-free and one degree of pipelined parallelism.

$$D = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \quad R = \begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 0 \end{pmatrix} \quad D' = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

Step 1 Communication-free parallelism

$$r_1 D' = \vec{0}^T \quad \Rightarrow c_{13} = 0$$

$$\text{minimize } c_{11} + c_{12} + c_{13}$$

$$\mathbf{r_1} : c_{11} = 1,\ c_{12} = 0,\ c_{13} = 0$$

$$U = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad R' = UR = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \end{pmatrix}$$

$$D'' = D \setminus R \cup R' = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Step 2 Pipelined parallelism

$$c_{21} \geq 0;\ c_{22} \geq 0;\ c_{23} \geq 0;\ c_{23} + c_{22} > 0$$

$$\text{minimize } c_{21} + c_{22} + c_{23}$$

$$\mathbf{r_2} : c_{21} = c_{22} = 0,\ c_{23} = 1$$

$$\Pi = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad L = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

$$\therefore\ K = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$$

Step 3 Time dimension(s)

$$P = \Pi; \quad Q = I - P^T(PP^T)^{-1}P = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

$$c_{32} > 0;\ c_{32} \geq 1;\ c_{33} \geq 1$$

$$\text{minimize } c_{31} + c_{32} + c_{33}$$

$$c_{31} = 0,\ c_{32} = c_{33} = 1$$

$$\mathbf{T} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \qquad \Rightarrow \begin{pmatrix} p_1 \\ p_2 \\ t_1 \end{pmatrix} = \begin{pmatrix} i \\ k \\ j+k \end{pmatrix}$$

$$\mathbf{T^{-1}} \begin{pmatrix} \vec{p} \\ \vec{t} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} \vec{p} \\ \vec{t} \end{pmatrix} = \begin{pmatrix} p_1 \\ t - p_2 \\ p_2 \end{pmatrix}$$

$$\Rightarrow p_2 + 1 \leq t \leq p_2 + B$$

A single controller located at the left edge streams a signal during $1 \leq t \leq N$ that is propagated along (0,1) with a single cycle delay. All processors along (1,0) share the same control signal.