Design of throughput-optimized arrays from recurrence abstractions

Arpith C. Jacob
Jeremy D. Buhler
Roger D. Chamberlain

Arpith C. Jacob, Jeremy D. Buhler, and Roger D. Chamberlain, “Design of throughput-optimized arrays from recurrence abstractions,” in *Proc. of 21st IEEE International Conference on Application-specific Systems, Architectures and Processors*, July 2010, pp. 133-140.

Dept. of Computer Science and Engineering
Washington University in St. Louis

Design of throughput-optimized arrays from recurrence abstractions

Arpith C. Jacob*, Jeremy D. Buhler*, and Roger D. Chamberlain*†
*Department of Computer Science and Engineering, Washington University in St. Louis
†BECS Technology, Inc., St. Louis, Missouri
Email: {jarpith,jbuhler,roger}@wustl.edu

***Abstract*—Many compute-bound applications have seen order-of-magnitude speedups using special-purpose accelerators. FPGAs in particular are good at implementing recurrence equations realized as arrays. Existing high-level synthesis approaches for recurrence equations produce an array that is latency-space optimal. We target applications that operate on a large collection of small inputs, e.g. a database of biological sequences, where overall throughput is the most important measure of performance.**

In this work, we introduce a new design-space exploration procedure within the polyhedral framework to optimize throughput of a systolic array subject to area and bandwidth constraints of an FPGA device. Our approach is to exploit additional parallelism by pipelining multiple inputs on an array and multiple iteration vectors in a processing element. We prove that the throughput of an array is given by the inverse of the maximum number of iteration vectors executed by any processor in the array, which is determined solely by the array's projection vector. We have applied this observation to discover novel arrays for Nussinov RNA folding. Our throughput-optimized array is $2\times$ faster than the standard latency-space optimal array, yet it uses 15% fewer LUT resources. We achieve a further $2\times$ speedup by processor pipelining, with only a 37% increase in resources. Our tool suggests additional arrays that trade area for throughput and are 4-$5\times$ faster than the currently used latency-optimized array. These novel arrays are 70-$172\times$ faster than a software baseline.

***Keywords*-Systolic Array, Throughput Optimization, Recurrences, FPGA, Dynamic Programming.**

I. Introduction

Massive and growing data sets are a challenge common to many areas of computation today, including text processing, image and signal processing, and computational biology. Nontraditional computing architectures such as graphics processors or field-programmable gate arrays (FPGAs) can frequently execute these computations orders of magnitude faster than general-purpose microprocessors. We are interested in algorithms represented as systems of recurrence equations (regular loops with uniform dependencies) that can be realized as a *systolic array*, a collection of simple, parallel processing elements (PEs) with regular interconnections. Systolic arrays are well-suited to FPGAs, both because these devices can implement arbitrary circuits to realize customized PEs and because their structure rewards simple, regular, locally connected arrays.

Fine-grained parallel architectures are difficult to program, often requiring a specialized hardware description language (HDL) and careful attention to resource usage and communication timing. To remove this burden from the programmer, we turn to the well-studied field of automatic synthesis of systolic arrays from recurrences [1]. Previous work in this area provides powerful tools to compute mappings from high-level algorithms to low-level implementations. However, our attempts to apply these tools to our target application domains have exposed a key limitation, which we seek to remedy in this work.

Most techniques to realize systolic arrays from a recurrence seek an array that is latency-space optimal [2], [3]. Such an array computes a single instance of the recurrence in the shortest time possible, i.e., with minimum latency; among all such arrays, a latency-space optimal array requires the fewest PEs. In our application domains, however, we seek to accelerate computations over large collections of small, discrete inputs. For example, computational biology algorithms often work on large databases of short DNA or protein sequences, while video processing may require analysis of a stream of individual image frames. For such applications, the latency of computation on an individual input is less important than the *throughput*, or equivalently, the total execution time on the entire data set.

In this work we address the problem of throughput-optimized array design. We exploit parallelism found in pipelining computation of multiple inputs on an array. We first give a mathematical definition for the throughput of a systolic array. We show that the throughput can be computed solely from the array's *allocation function*, i.e., its mapping of iterations in the recurrence onto compute elements, independent of the *schedule* of times at which these steps are executed. This observation leads to an efficient search strategy for finding arrays with optimal throughput. Second, we pipeline multiple iterations of a recurrence on a PE to further improve performance by increasing clock frequency. We achieve this by automatically pipelining the PE datapath using the retiming circuit optimization, which requires no additional programmer effort.

We describe a software tool to accept recurrence descriptions of programs and perform the aforementioned search for array mappings. The schedules and allocations generated by our tool can be fed into array synthesis software such as MMAlpha [4] and PARO [5] to automatically synthesize HDL descriptions of systolic arrays. We have applied this

observation to discover novel arrays for Nussinov RNA folding. Our throughput-optimized array is $2\times$ faster than the standard latency-space optimal array, using fewer resources. We achieve a further $2\times$ speedup by processor pipelining, with only a 37% increase in resources. Our tool suggests additional arrays that trade area for throughput and are 4-$5\times$ faster than the currently used latency-optimized array. These novel arrays are 70-$172\times$ faster than a software baseline.

II. Background: Parallelization of Recurrences

We seek to accelerate regular loop programs expressed as a set of parametrized uniform recurrences [1]. We are interested in fixed-size arrays; the problem of partitioning large arrays is beyond the scope of this work.

A recurrence describes how to compute a data variable $X(z)$ over a convex domain $\mathcal{D} \subset \mathbb{Z}^n$. Points in the domain are identified by n-dimensional *index vectors* $z = [i_1, \ldots, i_n]$. The domain may be parametrized, i.e., it may have one or more *size parameters*, such as the length of an input sequence. A recurrence induces data dependencies between index vectors; for example, if $X(z)$ needs the value at $X(z+d_j)$ for its computation, we say d_j is a dependency vector. We consider uniform dependencies here, though our design techniques also work in the presence of more general affine dependencies.

A system of recurrences can be computed by a *systolic array*. Such an array can be built given a *schedule* and *allocation* on domain $\mathcal{D}$.

Schedule: A schedule $\tau : \mathbb{Z}^n \rightarrow \mathbb{Z}$ gives the time $\tau(z)$ at which $X(z)$ is computed for each point $z \in \mathcal{D}$. We consider only schedules described by an affine function $\tau(z) = \lambda z + \alpha$, where λ is an n-vector and α is a constant. A schedule must respect a recurrence's dependencies; for each dependency vector d_j, we impose the causality constraint $\tau(z + d_j) < \tau(z)$, which simplifies to $\lambda d_j < 0$.

Allocation: The allocation $\pi : \mathbb{Z}^n \rightarrow \mathbb{Z}^{n-1}$ maps each point $z \in \mathcal{D}$ to a PE $\pi(z)$ that computes $X(z)$. The allocation must be compatible with the schedule, in the sense that no two points may be computed on the same PE at the same time (injectivity constraint). The allocation can be specified by a *projection vector* u along which the domain $\mathcal{D}$ is projected; injectivity then requires that $\lambda u \neq 0$, where λ is the vector in the scheduling function. When the recurrence is realized as an array, each dependency d_i induces an inter-PE link in direction $\pi(d_i)$ with a delay of λd_i clocks.

Array Utilization: The *utilization* of an array is $1/\gamma$, where $\gamma = |\lambda u|$. If $\gamma > 1$, then each PE in the array is active in only one of every γ clock cycles, and the array is said to be *underutilized*. For such arrays, we can reduce the number of idle PEs and increase array throughput by interleaving computation for γ instances of the input problem. Alternatively, if resource usage is the principal constraint, γ virtual PEs in an array may be "clustered" to run serially on a single physical processor.

A. Examples

In this work, we use two recurrences to illustrate our methods. Below, we describe only the domains of the recurrences; the referenced works give their full equations. Although these recurrences' bodies consist of relatively simple integer operations, our techniques apply independent of the operations' complexity, since our method depends solely on the shape and size of the computation domain.

Banded Smith-Waterman: Chao et al. [6] introduced a recurrence that aligns two sequences of lengths N along a band of width w, centered on its middle diagonal. The output is a scalar representing the score of a correspondence between the two sequences that minimizes the weighted edit distance between them. The computation's domain is $\mathcal{D} = \{\ i, j \mid 1 \leq i \leq N;\ \max(1, i - \frac{w}{2} + 1) \leq j \leq \min(N, i + \frac{w}{2})\ \}$.

Nussinov: The Nussinov algorithm [7] computes the score of the optimal folded substructure of an RNA sequence of length N. Its domain is $\mathcal{D} = \{\ i, j, k \mid 1 \leq i \leq N;\ i \leq j \leq N;\ 1 \leq k \leq \frac{j-i}{2}\ \}$. This domain is similar in shape to those of other problems such as optimal string parenthesization.

A *domain instance* $\mathcal{I}$ describes the set of computations required to evaluate a recurrence for one particular input. The input defines the size parameters of the instance. For example, consider the Nussinov algorithm applied to two input sequences of length $N = 100$ and $N = 200$, which respectively define instances $\mathcal{I}_1$ and $\mathcal{I}_2$. The first instance includes points $\{\ i, j, k \mid 1 \leq i \leq 100;\ i \leq j \leq 100;\ 1 \leq k \leq \frac{j-i}{2}\ \}$, while the second induces an upper bound of 200 on i and j. Every sequence defines a distinct instance to be computed, but sequences of the same length induce domain instances with the same number of points to be computed.

III. Throughput-optimized Arrays

In this section, we introduce throughput-optimized arrays and describe a strategy for designing them. Define the *computation time* or *latency* $\mathcal{L}$ of an instance as the time to evaluate a recurrence at all points in its domain. Latency equals the difference in execution times between the first and last scheduled points. Previous work has focused on finding arrays whose schedules are latency-optimal [2], [8].

We instead seek to optimize the *throughput* of a systolic array, which is key to minimizing the *total* execution time over a stream of instances. Define β, the *block pipelining period* of an array, to be the earliest time (in clock cycles) after an array starts to compute on an input instance, at which computation on a second instance can be started without processor contention. For safe execution, the block pipelining period must be large enough that no single PE in the array is scheduled to compute points from both instances' domains at the same time.

Consider a recurrence to be evaluated on m input instances of the same size. Suppose we have a *pipelined array*

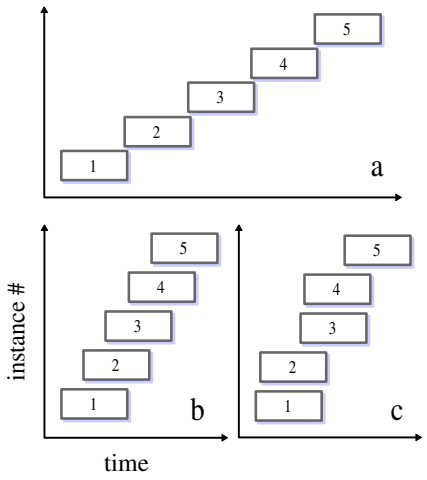

Figure 1. **(a)** The latency-optimal array sequentially executes five input instances in total time $5\mathcal{L}_{opt}$ clock cycles. **(b)** A throughput-optimized array that pipelines a new input every $\frac{1}{3}\mathcal{L}_{opt}$ clock cycles. **(c)** When the array is not fully efficient (here 50%), we simultaneously pipeline two input instances every $\frac{2}{3}\mathcal{L}_{opt}$ clock cycles. Both throughput-optimized arrays have execution time $2.33\mathcal{L}_{opt}$ and are $2.14\times$ faster than the latency-optimal array.

with schedule $\tau(z)$ for one instance and block pipelining period β. We can start computing on successive instances at intervals of β with the schedule $f(z') = \tau(z') + \beta$. The pipelined array completes m instances in time $(m-1)\beta+\mathcal{L}$, where $\mathcal{L}$ is the latency for one instance. By comparison, a latency-optimal array requires $m\mathcal{L}_{opt}$ cycles to complete the same instances. As shown in Figure 1, if $\beta < \mathcal{L}_{opt}$ (and the latency $\mathcal{L}$, if larger than $\mathcal{L}_{opt}$, is amortized over a large number of inputs), then the block-pipelined array has higher throughput than the latency-optimal array.

Based on the above discussion, we may improve throughput by finding arrays with small β. However, this block pipelining period depends on both an array's schedule and allocation. Jointly exploring schedule and allocation space to minimize β could greatly increase search complexity, making this form of throughput optimization computationally infeasible. We would like to formulate a precise optimization criterion related to β that allows us to rapidly find arrays of maximum throughput without an expensive joint search over schedules and allocations.

A. Design Criterion for Throughput Optimality

We will prove that the throughput of an array can be determined from its projection vector, *independently of its schedule*. This important observation allows us to efficiently optimize for throughput by searching only the space of projection vector candidates, rather than having to jointly consider projections and schedules.

Preliminaries: Assume a system of parametrized uniform recurrence equations and an associated schedule and allocation for a systolic array. We are given two instances $\mathcal{I}$ and $\mathcal{I}'$ of the same size, i.e., $\mathcal{D}(\mathcal{I}) = \mathcal{D}(\mathcal{I}')$, so their domains contain the same number of points: $z \in \mathcal{D}(\mathcal{I}) \Leftrightarrow z \in \mathcal{D}(\mathcal{I}')$.

We wish to derive a schedule to execute the two instances in pipelined fashion, one after the other. Let the first instance $\mathcal{I}$ execute on the schedule $\tau(z)$, $z \in \mathcal{D}(I)$, derived as in Section II. We seek a pipelined linear schedule f for $\mathcal{I}'$ given by $f(z') = \tau(z') + \beta$, where $z' \in \mathcal{D}(\mathcal{I}')$. To ensure that no PE is required to compute on both instances at the same time, the schedules τ and f must obey the following constraint:

$$\forall\ z_1, z_2 \in \mathcal{D}(I) \text{ with } \pi(z_1) = \pi(z_2),\ f(z_1) > \tau(z_2). \quad (1)$$

This feasibility constraint still allows simultaneous execution for two domain points z_1, z_2 from two independent instances *if* they execute on different PEs, i.e., $\pi(z_1) \neq \pi(z_2)$.

Any two domain points z_1 and z_2 in $\mathcal{D}(\mathcal{I})$ are executed by the same PE if and only if $z_1 - z_2 = ku\ ,\ k \in \mathbb{Z}$. We denote the maximum number of domain points executed by any one PE as $k_{max} = 1 + k_{ilp}$, where k_{ilp} is determined by solving the following integer linear program:

$$\begin{aligned} \text{Maximize} \quad & k \qquad (2)\\ z_1 - z_2 &= ku \\ Cz_1 &\leq d \\ Cz_2 &\leq d. \end{aligned}$$

We have used the convex polyhedron representation $Cz \leq d$ of the domain of the system of recurrences in this formulation.

Our key observation is that feasible values of the block pipelining period β are constrained by how many domain points are executed on any one PE. Figure 2 illustrates this idea for one particular array. As a general result, the following theorem bounds the value of β for which $f(z)$ forms a feasible pipelined schedule.

Theorem 1: The feasibility constraint $f(z_1) > \tau(z_2)$ $\forall\ z_1, z_2 \in \mathcal{D}(\mathcal{I})$ with $\pi(z_1) = \pi(z_2)$ is satisfied if and only if $\beta > k_{ilp}|\lambda u|$.

Proof: For the only-if part, we know that there exist $z_1, z_2 \in \mathcal{D}(\mathcal{I})$ such that $z_1 - z_2 = k_{ilp}u$. For conflict-free pipelining, the point z_1 of the first instance must execute before z_2 of the second. We therefore have

$$\begin{aligned} f(z_2) &> \tau(z_1) \\ f(z_2) &> \tau(z_2 + k_{ilp}u) \\ \lambda z_2 + \alpha + \beta &> \lambda(z_2 + k_{ilp}u) + \alpha \\ \beta &> k_{ilp}\lambda u. \end{aligned}$$

Similarly, the point z_2 of the first instance must execute before z_1 of the second. We have $f(z_2 + k_{ilp}u) > \tau(z_2)$, which simplifies to $\beta > k_{ilp}(-\lambda u)$, and so the proof follows.

For the if part, assume to the contrary that $f(z_1) \leq \tau(z_2)$

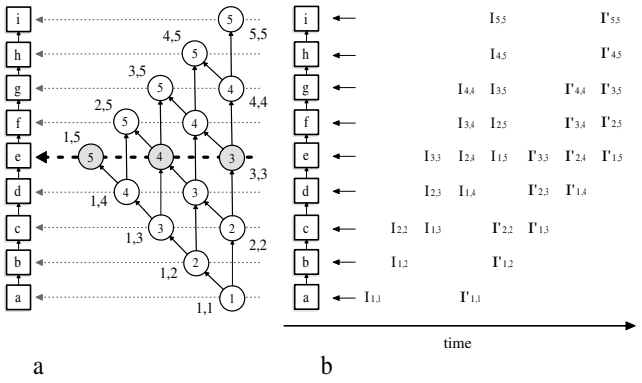

Figure 2. Pipelining the execution of two input instances on an array to improve throughput. **(a)** The triangular domain is projected horizontally along the dashed lines onto a linear array of processors. The execution time of every iteration point is shown in the circle. Processor e executes the maximum of $k_{max} = 3$ iteration points. **(b)** We can therefore pipeline two instances $\mathcal{I}$ and $\mathcal{I}'$ of the same size with a pipelining period $\beta = 3$ without risk of processor contention.

for some $z_1, z_2 \in \mathcal{D}(\mathcal{I})$. Since $\pi(z_1) = \pi(z_2)$, we have $z_2 - z_1 = ku$. WLOG, we assume that k is a positive integer. Then we have

$$\begin{aligned} f(z_1) &\leq \tau(z_1 + ku) \\ \lambda z_1 + \alpha + \beta &\leq \lambda(z_1 + ku) + \alpha \\ \lambda z_1 + \beta &\leq \lambda z_1 + k\lambda u \\ \beta &\leq k\lambda u. \end{aligned}$$

We know that $k \leq k_{ilp}$; since by definition, $\beta > k_{ilp}|\lambda u|$, we have a contradiction. ■

As a corollary, we can show through induction that multiple input instances can be safely pipelined on the array with a block pipelining period as calculated above.

For the example in Figure 2, processor e computes the largest number of iteration points $k_{max} = 3$. The array being fully efficient, $|\lambda u| = 1$, and we can pipeline a new input instance every three clock cycles.

B. Implications for Design-Space Exploration

We have shown in Theorem 1 that the block pipelining period must be greater than $k_{ilp}|\lambda u|$. The product $|\lambda u|$ is the reciprocal of array utilization (see Section II); confronted with an array that is not fully efficient, we can increase its throughput by computing $|\lambda u|$ input instances simultaneously (assuming the I/O bandwidth is not a limitation). The throughput achievable on such an array is therefore $\frac{|\lambda u|}{k_{ilp}|\lambda u|+1}$, which is one input instance every $k_{ilp} + \left\lceil \frac{1}{|\lambda u|} \right\rceil = k_{max}$ clock cycles. We conclude that *for an array with projection vector u, regardless of its schedule, the achievable throughput is always one instance per k_{max} clock cycles.*

IV. Finding Throughput-Optimized Projection Vectors

We now describe a procedure for searching the space of projection vectors u for a given set of recurrences to find array designs with high throughput. We do not limit ourselves to a single design; rather, we seek a collection of arrays with a variety of throughput/area tradeoffs. Increasing the magnitude of the vector u decreases the number of iteration points executed by each PE in the array and so improves throughput. However, there is a limit to this increase because extremely high-throughput arrays may require more PEs than fit on the FPGA, or the rate of instance processing may require more input bandwidth than is available. We seek high-throughput projection vectors that satisfy these system constraints.

In general, the space of possible projection vectors grows exponentially with the size of the input instance. For even modest-sized inputs, this space is too large to enumerate completely in reasonable time, especially if the recurrence involves multiple size parameters. Fortunately, we can derive and apply *a priori* bounds on the magnitude $|u|$ based on our target platform's resource constraints. These bounds come from two sources: FPGA logic and block RAM constraints, and total system input bandwidth.

A. Search Procedure for Projection Vectors

For convenience, suppose that the input domain size of a given recurrence is defined by a single integer parameter N, and let a fixed size N_0 for this parameter be given. We say that an array is *feasible* if it satisfies bandwidth and area constraints on the target platform. We first compute a resource bound B such that no array design with $|u| > B$ is feasible for inputs of size $\geq N_0$. We then compute throughputs for all u with $|u| \leq B$; in general, these throughputs are functions of N. The bounded search is outlined in Algorithm 1.

When many different vectors yield the same throughput, we compute schedules for each and keep only the array with the most desirable properties, in this case the smallest area (as estimated by the number of PEs) and high utilization. We use the vertex method [2], which solves an integer linear program, to derive a linear schedule satisfying the constraints of Section II. Among all feasible schedules, the scheduling ILP can select one that minimizes latency or has other desirable properties as described in Section V.

Our search algorithm finally returns a set $\mathcal{C}$ of parametrized array designs, one per distinct throughput. The algorithm guarantees that every array that satisfies the area and bandwidth constraints is explored, though it may explore more designs than is strictly necessary.

Given inputs of fixed size N, we may simply choose the highest-throughput array design for size N from $\mathcal{C}$. If, however, the input stream contains inputs of varying sizes,

Algorithm 1 Explore allocation/schedule space

```
1: procedure EXPLORE(Recurrence, Parameter Values)
2:     bound ← min(BandwidthBound, ResourceBound)
3:
4:     for each projection vector u within bounds do
5:         T ← Throughput(u)
6:         π ← Allocation(u)                    ▷ π is nullspace basis of u
7:         τ ← Schedule(u)
8:         #PE ← NumPEs(π, τ)                   ▷ Count number of processors
9:     end for
10:
11:    Sort the solutions by T, #PE, and γ
12:    Select one array for every distinct T for C
13: end procedure
```

we can switch among arrays in $\mathcal{C}$ by reconfiguration, achieving better overall throughput than any single array. Our previous work [11] describes one practical reconfiguration strategy and its application to a real-world example. For any input size $N \geq N_0$, one of the arrays in $\mathcal{C}$ that is feasible is guaranteed to be the throughput-optimal array. For $N < N_0$, the array designs in $\mathcal{C}$ are not necessarily throughput-optimal.

1) Input Bandwidth and FPGA Resource Bounds: Here, we only state the bounds we have derived for the resource constraints; the full derivations are available in an extended version of the paper [16]. The input bandwidth bound on the magnitude of the projection vector is given by

$$|u| \quad \leq \quad \frac{2m}{b}\omega(\mathcal{D}). \tag{3}$$

Here m is the system input bandwidth expressed in data bits per clock period of the array, b is the number of data bits required per input instance, and ω is a function that depends on the shape of the computational domain.

Similarly, we can express the FPGA resource bound as

$$|u| \quad \leq \quad \frac{2p}{|\mathcal{D}|}\omega(\mathcal{D}). \tag{4}$$

The bound is a function of p, an estimate of the number of PEs that can fit on the target FPGA device, and $|\mathcal{D}|$, the number of iteration points in the computational domain.

2) Search Complexity and Application to Examples: Our search considers every projection vector u with integral coordinates that has magnitude at most some bound derived from the above considerations. For a recurrence whose domain is in $\mathbb{Z}^n$, the number of vectors to consider grows as $O(2^n|u|^n)$. The search cost per vector is independent of the magnitude but depends on the domain shape and the dependencies; in practice, we can process 40-120 vectors/second for the recurrences considered here. To achieve overall search times of seconds to a few minutes, a reasonable bound on u would be roughly 40 for a two-dimensional recurrence, or 15 for a three-dimensional recurrence.

Input Bandwidth Bound: Our FPGA system supports an input bandwidth of 64 bits per clock at 133 MHz; we assume generated hardware arrays clock at least this fast.

Take an instance of the banded Smith-Waterman algorithm that aligns a query to a series of target protein sequences of length 300. The characters of a protein sequence may be represented using 5 bits. We assume that the query is fixed, and that each input instance is a new target sequence. Then the bandwidth bound is $|u| \leq 37$.

Assume the Nussinov algorithm folds RNAs of length 60, where each character of the sequence can be represented in 3 bits. The bound in this case is $|u| \leq 61$, resulting in a far more expensive search than in the previous example.

FPGA Resource Bound: Returning to the Nussinov algorithm, we generated a PE with a 6-bit datapath on a Xilinx Virtex 4 LX100-12 FPGA. From this synthesis, we predict that at most 1680 PEs can fit on the FPGA. For $N = 60$, the area bound is $|u| \leq 16$, which constrains the search to the range of a few minutes.

For the banded Smith-Waterman algorithm, we are able to support up to 480 9-bit PEs on the FPGA device; here, the limiting factor is the number of available on-chip block RAM memories. We used a band length of 66 and sequences of 300 characters, to derive a bound of $|u| \leq 22$.

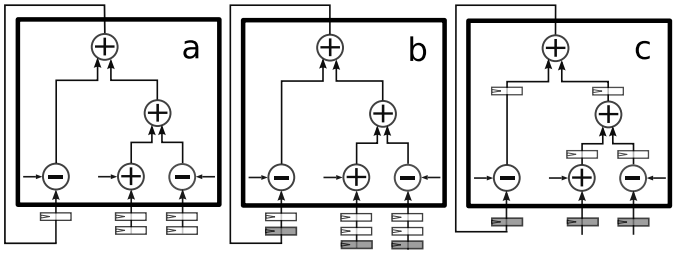

Figure 3. **(a)** The unpipelined PE has three operations in its critical path. **(b)** When the schedule is relaxed, new delay registers are generated on dependency links, which are moved by the circuit retimer as shown in **(c)** to reduce the length of the critical path to one operation.

V. Selecting Arrays to Support Retiming

Thus far, we have focused on improving throughput by pipelining input instances on an array. This form of throughput optimization is distinct from, and orthogonal to any effort to improve the array's clock period by internally pipelining each PE. We now show that with a little additional effort, we can augment our search procedure to select array designs that are amenable to PE pipelining. We can perform this pipelining *automatically* using synthesis tools, giving us the benefits of both a high rate of instances completed per cycle and a high clock frequency.

Manual pipelining of circuits is cumbersome and difficult to do efficiently without knowledge of the underlying implementation fabric. Fortunately, modern synthesis tools include a synchronous circuit optimization called *retiming* [9]. As shown in Figure 3, a retimer can automatically move registers into a combinational logic path to reduce the length of the critical path without affecting correctness, resulting in designs with higher clock frequencies.

To enable effective retiming, we modify our array design procedure to provide "new" delay registers (shown shaded in Figure 3) for the retimer to move into each PE. Specifically, we relax the schedule generated for an array to introduce larger delays on dependency links into a PE.

Suppose we would like to pipeline the PE by s stages. We modify the causality constraint of Section II to be $\tau(d_j) \leq -s$. Any schedule that satisfies this constraint is still feasible (satisfies dependencies) and is guaranteed to include at least s delay registers on every dependency link into a PE.

Although introducing artificial delays increases latency, the projection vector remains unchanged, and so the throughput of the array remains equal to the bound implied by Theorem 1. However, this bound assumes that for arrays that are not fully efficient (i.e. have $|\lambda u| > 1$), we interleave computation on multiple input instances. While technically feasible, interleaving is often undesirable because it increases complexity and requires multiple independent buffers at the input and output sides of the array to interleave and deinterleave the input/output. We therefore weight the objective function of our scheduling integer linear program to prioritize higher utilization over lower latency and discard schedules that result in less than fully efficient arrays.

VI. Software Tool

We have written a design-space exploration tool in C++ implementing the ideas described in this paper. This tool is intended to be a plugin to an automatic parallelization package such as MMAlpha [4], so that we can reuse its front-end and code-generation phases. We assume the input recurrence has been parsed and directly read its dependencies, vertices, and the polyhedral domain of computation from text files.

We used the Polyhedral library [12] for polyhedral manipulations, the PIP library [13] for solving integer linear programs, and the Barvinok library [14] for counting the number of integral points in a polyhedron.

Our tool computes the collection $\mathcal{C}$ of array designs using Algorithm 1. Any selected schedule and allocation pairs from $\mathcal{C}$ may be passed on to MMAlpha for array generation.

VII. Results

Tables I and II show some of the projection vectors suggested by our software on the two example recurrences. The latency-space optimal array is shown in the first row. Our design-space exploration required less than five seconds for the Smith-Waterman recurrence and less than four minutes for Nussinov on an Intel Core 2 Duo workstation. In these two cases, 36 and 7117 projection vectors were explored.

The six columns in Table I from left to right are the projection vector, the maximum number of domain points executed by any processor, the number of processors instantiated for the size N_0, the reciprocal of the array's utilization, the array's latency for a single instance, and the predicted maximum array size that will fit on a Xilinx Virtex 4 LX100-12 FPGA. The block pipelining period can be computed as one plus the product of $(k_{max} - 1)$ and γ.

Table I
Throughput-area tradeoff for banded Smith-Waterman. Bounding was based on sequences of length $N_0 = 300$ and a bandwidth of $w = 66$, forcing $|u| \leq 22$.

u	k_{max}	$\#PEs$	γ	$\mathcal{L}$	**Max.** N
1 1	N_0	66	2	598	∞
1 0	66	300	1	598	480
1 -1	33	599	1	897	240
2 -1	22	898	1	598	160

Table II
Throughput-area tradeoff for the Nussinov recurrence. Bounding was based on $N_0 = 61$, forcing $|u| \leq 16$.

	u	k_{max}	$\#PEs$	γ	$\mathcal{L}$	**Max.** N	**Speedup**
A	-1 0 0	$N_0 - 2$	900	2	116	82	1.0
B	1 1 0	$N_0 - 2$	900	1	174	82	2.0
C	0 0 -1	$\frac{N_0-1}{2}$	1770	1	116	59	3.7
D	1 2 0	$\frac{N_0-1}{2}$	1770	1	171	-	-
E	1 1 -1	$\frac{N_0}{3}$	2611	1	116	49	4.9
F	2 2 -1	$\frac{N_0+1}{4}$	3423	1	116	-	-

Novel throughput-optimized arrays: Banded Smith-Waterman shows a number of attractive arrays with block pipelining periods equal to fractions of the latency. Existing hardware arrays to accelerate the banded Smith-Waterman computation use the latency-space optimal array with projection vector $[1, 1]$. Our alternate designs execute 4-13$\times$ faster than the latency-optimized array.

The latency-space optimal array for the Nussinov algorithm (array A in Table II) derived in literature [15] has a utilization of 50%. We may cluster two adjacent PEs, decreasing the space cost to 450 in the example. This array has a block pipelining period equal to its latency, $2N_0 - 6$ clock cycles, and does not optimize throughput.

Our search tool discovers novel arrays that realize greater throughput and have better I/O properties. Array B has a pipelining period that is half that of the previous array. Its utilization is 100%, though with a longer latency schedule. Furthermore, as shown in Figure 4, sequence data in the new array is loaded serially into just one PE rather than in parallel across $N-1$ PEs, saving 45% of LUTs and 61% of registers in the controller. This property can be discovered automatically by studying the subset of iteration vectors that perform I/O. Overall, the throughput-optimized array uses 15% fewer LUTs than the latency-optimized array when folding a sequence of length 81.

An alternate array suggested in literature is array C [15], which projects points in the computation domain onto the i-j plane. Our tool suggests array D, which has the same performance as C but has fewer PEs implementing resource-intensive calculations at domain points with $k = 1$. Our tool finds many other projections that trade

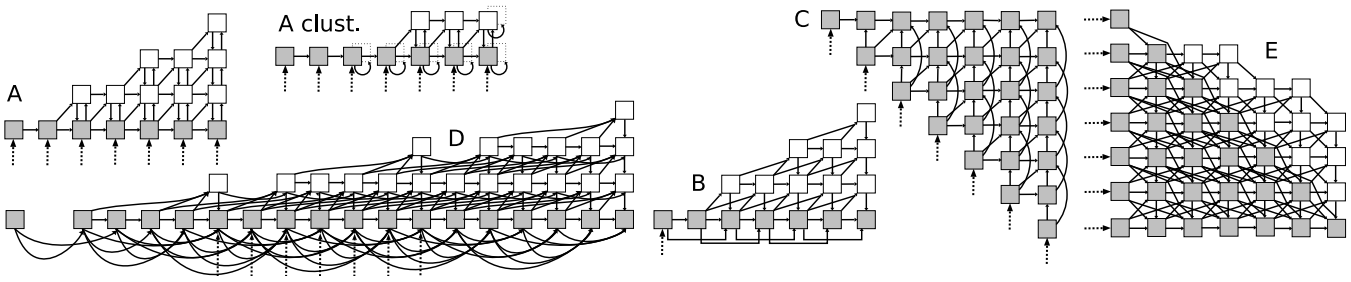

Figure 4. Some of the arrays suggested by our technique for the Nussinov algorithm.

area for higher throughput. We implemented some of these arrays on our target FPGA running at 150 MHz, folded 40 million RNAs of length 41, and report speedup normalized to array A; our arrays are 2-5× faster. A reference software implementation compiled using gcc 4.4.0 with flags *-O3 -march=nocona -fomit-frame-pointer* ran in 816 seconds on a single core of a Core II Duo 3 GHz CPU with 4 MB cache; arrays A-E are 34.8-172.3× faster.

These results show that the block pipelining period is indeed significantly lower than the latency for important recurrences. Our technique finds arrays with the same space cost as the latency-space optimal array but with a higher throughput. This allows us to pipeline multiple inputs, leading to a two-fold increase in speedup. Furthermore, for most distinct k_{max}, we are able to find an array that is fully efficient, with at most a minor increase in latency. These results validate our decision to optimize throughput by minimizing the block pipelining period rather than latency.

High clock speed designs: Our tool ensures that the novel arrays (B, D-F in Table II) have schedules that may be relaxed to allow retiming while maintaining full utilization. We implemented array B with an 8-bit datapath (sufficient precision for this application) parametrized by RNA length and the schedule function. We also used shift registers rather than counters (as is traditionally used) to implement the array controller, enabling the design to be clocked at high speeds. The retiming optimization was turned on in the synthesis tool, but no additional programmer effort was spent in realizing the high-speed arrays.

Figure 5 shows the increase in clock frequency as a function of the number of pipelined stages in a PE after place and route on a Xilinx Virtex 4 LX100-12 FPGA. The array derived using existing techniques clocks at 180 MHz. Arrays with pipelined PEs are up to 2× faster with only a 37% increase in slice usage. With a 16-bit datapath we can increase the clock frequency to 280 MHz before a single operation becomes the critical path. These operators will have to be manually pipelined to see further speed improvement. The area increase in both cases is small because, while relaxing the schedule increases the number of registers instantiated on the communication links, registers not used for retiming can be implemented as shift registers,

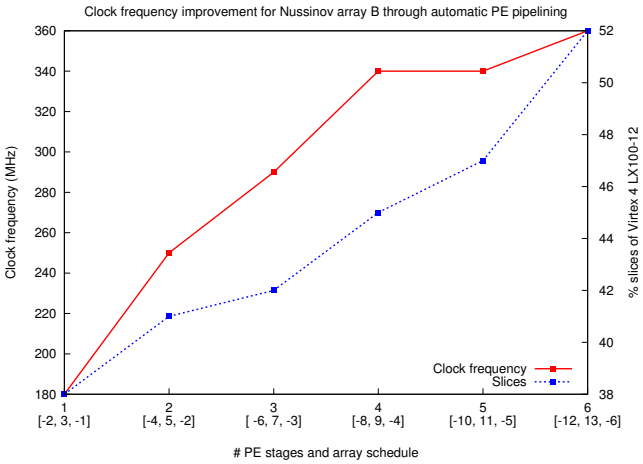

Figure 5. Improvement of array clock frequency as a function of pipelined stages in a Nussinov PE.

which are efficiently synthesized on modern FPGA fabrics. Note that the block pipelining period of $N_0 - 2$ clock cycles and the number of clocks to load sequence data are constant for all relaxed schedules. On the other hand, the latency-space optimal array A cannot be similarly pipelined to improve clock frequency without a drop in PE utilization.

VIII. Related Work

As mentioned previously, standard practice within the polyhedral framework has been to find a single latency-space optimal array. Wong et al. [3] describe an enumerative search to minimize the number of processors in an allocation, which is employed by PARO [5]. We have reused these ideas to derive input bandwidth and area bounds in our search for maximum-throughput arrays.

The only work we are aware of that optimizes throughput using space-time methods is by Rosseel et al. [17]. Their goal is to find a single array that matches the requested throughput and minimizes the area requirement. They do not pipeline multiple inputs as we do, but focus on improving the clock frequency of a PE. The authors recognize that throughput is dependent on the block pipelining period,

but they do not directly minimize k_{ilp}; rather, they attempt to estimate the product $k_{ilp}|\lambda u|$ through a multi-phase interleaved enumerative search through the allocation and schedule spaces. Because we are able to directly calculate the block pipelining period using only the projection vector, we can search the two spaces independently. The allocation matrices they consider have elements restricted to $0, \pm 1$, which reduces search time but does not optimize block pipelining period; hence, they may miss many interesting space mappings.

A technique analogous to ours is software pipelining, used to simultaneously execute successive iterations of an inner loop on a VLIW machine. Lam [18] outlines a procedure to determine a modulo schedule that permits continuous initiation of loop iterations at constant intervals. Scheduling of operations in the loop body is constrained by the availability of functional units and the inter-iteration dependencies. In contrast, we pipeline iterations of multiple loop instances. As a result, an advantage of our approach is that we do not have to deal with dependence constraints, which requires an iterative search for Lam's procedure. Overall, we have more freedom to explore a larger space of higher-throughput solutions using polyhedral theory.

Derrien et al. [10] first suggested using retiming to improve the clock frequency of arrays generated within the polyhedral framework. However, their focus is not on throughput-optimized arrays. Prior to scheduling and allocation, they use two new transformations of the computation domain — skewing and serialization — to allow retiming; the latter increases latency and k_{max}. This method decreases throughput of full-size arrays but works well for partitioned arrays where throughput is sacrificed to save logic resources.

IX. Conclusion

In this work, we have introduced a procedure to systematically find throughput-optimized systolic arrays from uniform recurrence equations. Our method exploits additional parallelism through the pipelining of multiple instances on an array as well as multiple iterations in a PE. Pipelining multiple instances necessitates changes to the control architecture and care in selecting control pipelines. However, preliminary analysis suggests this can be done in an automated fashion.

Newer generations of FPGAs have ever increasing numbers of gates and on-chip memory available for architects. For the Nussinov example, we were able to synthesize array B to process an RNA of several hundred bases on the latest Virtex 6. However, since biologists rarely fold RNAs longer than 200 bases, designers are faced with a question of how best to exploit these rich resources. One might simply use multiple instantiations of a single array. In the Smith-Waterman example, five parallel units of array $[1, 1]$ achieves the same throughput as array $[1, 0]$. However, this approach does not scale with the number of parallel instantiations. Alternately, we may use one of the novel arrays discovered by our tool that trades off area for increased throughput.

Acknowledgment

This work was supported by NIH award R42 HG003225. R.D. Chamberlain is a principal in BECS Technology, Inc.

References

[1] Dominique Lavenier et al. Advanced Systolic Design, in Digital Signal Processing for Multimedia Systems, Parhi and Nishitani eds, March 1999.

[2] S. Balev et al. Linear programming models for scheduling systems of affine recurrence equations—a comparative study. *Symp. Parallel Algorithms and Architectures*, 250–258, 1998.

[3] Y. Wong and J.-M. Delosme. Space-optimal linear processor allocation for systolic arrays synthesis. *International Parallel Proc. Symposium*, 275–282, 1992.

[4] A.C. Guillou et al. High level design of digital filters in mobile communications. In *DATE Design Contest*, March 2001.

[5] Frank Hannig et al. PARO: Synthesis of Hardware Accelerators for Multi-Dimensional Dataflow-Intensive Applications. *Workshop on Applied Reconfigurable Computing*, March 2008.

[6] K.M. Chao et al. Aligning two sequences within a specified diagonal band. *Comp. Appl. Biosci.*, 8(5):481–487, 1992.

[7] R. Nussinov et al. Algorithms for loop matchings. *SIAM Journal on Appl. Math.*, 35(1):68–82, July, 1978.

[8] P. Feautrier. Some efficient solutions to the affine scheduling problem: I. one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, 1992.

[9] Charles E. Leiserson, and James B.Saxe. Optimizing synchronous systems. *Foundations of Comp. Sc.*, 23–36, 1981.

[10] S Derrien et al. Combining Instruction and Loop Level Parallelism for FPGAs. *Field-Programmable Custom Computing Machines*, 273–282, 2001.

[11] A. Jacob, J. Buhler, and R. Chamberlain. Optimal runtime reconfiguration strategies for systolic arrays. *Field Programmable Logic and Application*, September 2009.

[12] PolyLib - A library of polyhedral functions, 2007. http://icps.u-strasbg.fr/polylib/.

[13] Feautrier, P. *PIP/Piplib, a parametric integer linear programming solver*, 2006. http://www.piplib.org/.

[14] Sven Verdoolaege et al. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica*, 48(1):37–66, June 2007.

[15] A. Jacob et al. Accelerating Nussinov RNA secondary structure prediction with systolic arrays on FPGAs. *Application-specific Systems, Arch. and Proc.*, pages 191–196, 2008.

[16] A. Jacob et al. Throughput-optimal systolic arrays from recurrence equations. *Washington University in St. Louis - Computer Science and Engineering Department, Tech. Rep. WUCSE-2009-39*, September 2009.

[17] J. Rosseel et al. An optimisation methodology for array mapping of affine recurrence equations in video and image processing. *Application-specific Systems, Architectures and Processors*, 415–426, August 1994.

[18] Monica Lam. Software pipelining: an effective scheduling technique for VLIW machines. *Conf. Prog. Lang. Design and Impl.*, 318–328, 1988.