View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Washington University St. Louis: Open Scholarship

Washington University in St. Louis

Washington University Open Scholarship

All Computer Science and Engineering Research

Computer Science and Engineering

Report Number: wucse-2009-39

2009

Throughput-optimal systolic arrays from recurrence equations

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

Many compute-bound software kernels have seen order-of-magnitude speedups on special-purpose accelerators built on specialized architectures such as field-programmable gate arrays (FPGAs). These architectures are particularly good at implementing dynamic programming algorithms that can be expressed as systems of recurrence equations, which in turn can be realized as systolic array designs. To efficiently find good realizations of an algorithm for a given hardware platform, we pursue software tools that can search the space of possible parallel array designs to optimize various design criteria. Most existing design tools in this area produce a design that is latency-space optimal. However, we instead... **Read complete abstract on page 2.**

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation

Jacob, Arpith C.; Buhler, Jeremy D.; and Chamberlain, Roger D., "Throughput-optimal systolic arrays from recurrence equations" Report Number: wucse-2009-39 (2009). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/17

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/17

Throughput-optimal systolic arrays from recurrence equations

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

Complete Abstract:

Many compute-bound software kernels have seen order-of-magnitude speedups on special-purpose accelerators built on specialized architectures such as field-programmable gate arrays (FPGAs). These architectures are particularly good at implementing dynamic programming algorithms that can be expressed as systems of recurrence equations, which in turn can be realized as systolic array designs. To efficiently find good realizations of an algorithm for a given hardware platform, we pursue software tools that can search the space of possible parallel array designs to optimize various design criteria. Most existing design tools in this area produce a design that is latency-space optimal. However, we instead wish to target applications that operate on a large collection of small inputs, e.g. a database of biological sequences. For such applications, overall throughput rather than latency per input is the most important measure of performance. In this work, we introduce a new procedure to optimize throughput of a systolic array subject to resource constraints, in this case the area and bandwidth constraints of an FPGA device. We show that the throughput of an array is dependent on the maximum number of lattice points executed by any processor in the array, which to a close approximation is determined solely by the array's projection vector. We describe a bounded search process to find throughput-optimal projection vectors and a tool to perform automated design space exploration, discovering a range of array designs that are optimal for inputs of different sizes. We apply our techniques to the Nussinov RNA folding algorithm to generate multiple mappings of this algorithm into systolic arrays. By combining our library of designs with run-time reconfiguration of an FPGA device to dynamically switch among them, we predict significant speedup over a single, latency-space optimal array.

Department of Computer Science & Engineering

Washington University in St.Louis
School of Engineering & Applied Science

2009-39

Throughput-optimal systolic arrays from recurrence equations

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

Corresponding Author: jarpith@cse.wustl.edu

Web Page: http://hpcb.wustl.edu

Abstract: Many compute-bound software kernels have seen order-of-magnitude speedups on special-purpose accelerators built on specialized architectures such as field-programmable gate arrays (FPGAs). These architectures are particularly good at implementing dynamic programming algorithms that can be expressed as systems of recurrence equations, which in turn can be realized as systolic array designs. To efficiently find good realizations of an algorithm for a given hardware platform, we pursue software tools that can search the space of possible parallel array designs to optimize various design criteria. Most existing design tools in this area produce a design that is latency-space optimal. However, we instead wish to target applications that operate on a large collection of small inputs, e.g. a database of biological sequences. For such applications, overall throughput rather than latency per input is the most important measure of performance.

In this work, we introduce a new procedure to optimize throughput of a systolic array subject to resource constraints, in this case the area and bandwidth constraints of an FPGA device. We show that the throughput of an array is dependent on the maximum number of lattice points executed by any processor in the array, which to a close approximation is determined solely by the array's projection vector. We describe a bounded search

Type of Report: Other

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160

Throughput-optimal systolic arrays from recurrence equations

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

Department of Computer Science and Engineering
Washington University in St. Louis
St. Louis, Missouri. 63130 USA
{jarpith,jbuhler,roger}cse.wustl.edu

Abstract

Many compute-bound software kernels have seen order-of-magnitude speedups on special-purpose accelerators built on specialized architectures such as field-programmable gate arrays (FPGAs). These architectures are particularly good at implementing dynamic programming algorithms that can be expressed as systems of recurrence equations, which in turn can be realized as systolic array designs. To efficiently find good realizations of an algorithm for a given hardware platform, we pursue software tools that can search the space of possible parallel array designs to optimize various design criteria. Most existing design tools in this area produce a design that is latency-space optimal. However, we instead wish to target applications that operate on a large collection of small inputs, e.g. a database of biological sequences. For such applications, overall throughput rather than latency per input is the most important measure of performance.

In this work, we introduce a new procedure to optimize throughput of a systolic array subject to resource constraints, in this case the area and bandwidth constraints of an FPGA device. We show that the throughput of an array is dependent on the maximum number of lattice points executed by any processor in the array, which to a close approximation is determined solely by the array's projection vector. We describe a bounded search process to find throughput-optimal projection vectors and a tool to perform automated design space exploration, discovering a range of array designs that are optimal for inputs of different sizes.

We apply our techniques to the Nussinov RNA folding algorithm to generate multiple mappings of this algorithm into systolic arrays. By combining our library of designs with run-time reconfiguration of an FPGA device to dynamically switch among them, we predict significant speedup over a single, latency-space optimal array.

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

General Terms Algorithms, Performance, Design

Keywords FPGA, Dynamic Programming, Throughput Optimization, Runtime Reconfiguration, Systolic Array

1. 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. One response to this challenge is to map analytical algorithms for these data sets into nontraditional computing architectures, such as chip multiprocessors, graphics processors, or field-programmable gate arrays (FPGAs), that can exploit their structure to execute them orders of magnitude faster than general-purpose microprocessors. This approach is especially fruitful when the algorithms can be expressed as uniform recurrences, whose regular structure lends itself to the fine-grained parallelization offered by these architectures.

To map a recurrence onto a fine-grained parallel architecture, we can realize it as a *systolic array* [1], a collection of simple parallel processing elements with regular interconnections. Systolic arrays are particularly well-suited to FPGAs, both because these devices can implement arbitrary circuits to realize processing elements and because their structure rewards simple, regular, locally-connected arrays of such elements. Recent advances in FPGA technology, including increased logic and memory resources and high-bandwidth, low-latency communication infrastructure [2, 3, 4, 5], have enabled the implementation of many resource- and data-intensive algorithms as systolic arrays [6, 7].

Fine-grained parallel architectures are difficult to program at a low level, often requiring a specialized implementation language and careful attention to resource usage and communication timing. To remove these burdens from the programmer, we turn to the well-studied area of automatic derivation of systolic arrays from recurrences [8, 9]. Previous work in this area provides powerful tools to compute mappings from high-level algorithms to low-level implementations, but our attempts to apply these tools to our target application domains have exposed two key limitations, which we seek to remedy in this work.

First, most techniques to realize systolic arrays from a recurrence seek an array that is latency-space optimal [12, 21, 27]. Such an array computes a single instance of the recurrence in the shortest time possible (minimum latency); among all arrays that achieve this shortest time, a latency-space optimal array requires the fewest processing elements. However, in our application domains, 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. The literature of automated systolic array design places little emphasis on optimizing arrays for throughput. We therefore address the problem of throughput-optimal array design here.

Second, we seek not just a single optimal array design but a library of designs optimized for different-sized inputs. A large data set may contain inputs of many different sizes, but a given systolic array has one characteristic input size, requiring different-sized inputs to be split or padded. A single array made sense when the target platform was a fixed-function integrated circuit, but modern devices can be rapidly reconfigured with designs optimized for different input sizes. For example, FPGAs from Xilinx and Altera support the loading of a new hardware circuit in tens to hundreds of milliseconds [13]. It is therefore feasible for one computation to use a range of array designs, each optimized to maximize throughput on inputs of a different size using all available computational resources.

The authors of a recent study [14] used a collection of throughput-optimal arrays on FPGAs to speedup the acceleration of RNA folding of a large database of sequences. When the input size distribution is known a priori, they assign small inputs to high-throughput, highly resource-intensive arrays and large inputs to lower-throughput, lower-resource arrays. This results in a net speedup over using a single latency-space optimal array. That work, however, does not show how to generate a collection of throughput-optimal arrays for a given recurrence. The procedures introduced here can automatically design a range of such arrays.

In this work, we first give a mathematical definition for the throughput of a systolic array. We show that a useful bound on throughput can be computed solely from the array's *projection vector*, i.e. its mapping of steps in the recurrence onto compute elements, independent of the *schedule* of times at which these steps are executed. This observation leads to a straightforward search strategy for finding throughput-optimal arrays. We then describe a software tool we have written to accept recurrence descriptions of programs and perform the aforementioned search. The schedule and allocation functions generated by our tool can be fed into array synthesis software such as MMAlpha [15], PARO [16], or PICO-NPA [17] to synthesize low-level HDL descriptions of systolic arrays. Finally, we apply our methods to find novel high-throughput systolic array designs for the Nussinov RNA folding algorithm.

2. Background: Parallelization of Recurrences

We seek to accelerate regular loop programs expressed as a set of parametrized uniform recurrences [8, 9]. A parametrized recurrence defines the computation of an n-dimensional data variable $X(z)$ over a domain $\mathcal{D}$, representing the individual steps or subproblems of the recurrence. The domain may have one or more *size parameters*, e.g. the length of an input sequence. The data dependencies between points in $\mathcal{D}$ are assumed to be uniform for efficient systolic array implementation; however, our design techniques also work for recurrences with more general affine dependencies.

A system of recurrences can be realized as a systolic array by finding two functions on the domain $\mathcal{D}$:

1. A *scheduling function*, which gives the time at which $X(z)$ is computed for each point $z \in \mathcal{D}$.
2. An *allocation function*, which gives the physical processing element of the array that computes each $X(z)$.

These two functions are sufficient to specify the shape and connections of the array, which can then be fully realized in the language of the target platform, e.g. Verilog or VHDL for FPGAs.

2.1 Scheduling

The scheduling function $\tau : \mathbb{Z}^n \to \mathbb{Z}$ maps each point $z \in \mathcal{D}$ to a computation time $\tau(z)$ such that:

1. $\tau(z) \geq 0, \ \forall\, z$ (positivity condition);
2. $X(z) \leftarrow X(y) \Rightarrow \tau(z) > \tau(y)$ (causality condition).

We are interested in schedules given by an affine function

$$\tau(z) = \lambda z + \alpha,$$

where λ is a n-vector and α is a constant.

2.2 Allocation

Once a schedule has been determined, we need to assign the computation of every instance $X(z)$ to a processing element. We define a linear allocation function $\pi(z) : \mathbb{Z}^n \to \mathbb{Z}^{n-1}$ such that:

$$\tau(z) = \tau(z') \Rightarrow \pi(z) \neq \pi(z') \text{ (injectivity condition).}$$

The allocation function can be fully specified by a direction vector u along which the computation domain is to be projected. Given a schedule, the injectivity constraint can be specified as $\lambda u \neq 0$. Once a suitable projection vector has been selected, the allocation function can be constructed as an $n \times n - 1$ matrix π such that $\pi u = 0$.

2.3 Array Utilization

A systolic array constructed as described above has a *utilization*, or *efficiency*, given by the reciprocal of $\gamma = |\lambda u|$. If this product is not equal to one, then each processing element in the array is active in only one of every γ clock cycles. Idle processors are not a good use of compute resources, so we wish to utilize each processing element as much as possible.

There are two approaches to increasing the efficiency of under-utilized arrays. If throughput is most important, γ instances of the input problem can be interleaved to execute simultaneously on the array. We call this strategy *processor pipelining*. Alternatively, if resource usage is a constraint, γ abstract processing elements may be "clustered" to run on a single processor with multiplexed inputs that is active every cycle.

2.4 Examples

Throughout this work we will reference a number of example recurrences to better illustrate the concepts introduced. Here we show only the domain of lattice points for each computation; the interested reader will find the entire algorithm including dependencies and the recurrence body in the referenced works.

Sorting. We use a recurrence similar to the one introduced by Rao [18] to sort a set of N integers. The output is the set sorted in descending order; an array implementing sorting must transfer the sorted numbers back to the host. The domain of computation is $\mathcal{D} = \{\, i, j \mid 1 \leq i \leq N + 1;\ 1 \leq j \leq i + 1 \,\}$.

Banded Smith-Waterman. This is a version of the banded Smith-Waterman algorithm described by Chao [19] for DNA and protein sequence alignment. The dynamic programming recurrence aligns two sequences of lengths M and N along a band of width w centered on its middle diagonal. The output is a scalar representing the score of an alignment, or correspondence, between the two sequences that minimizes the (weighted) edit distance between them. The domain of computation is $\mathcal{D} = \{\, i, j \mid 1 \leq i \leq M;\ \max(1, i - \frac{w}{2} + 1) \leq j \leq \min(N, i + \frac{w}{2}) \,\}$.

Nussinov. The Nussinov algorithm is used to compute the score of the optimal folded substructure of an RNA sequence of length N. This recurrence is the most complex of all the examples we consider; the reader is referred to existing work [26] for details. The domain of computation is $\mathcal{D} = \{\, i, j, k \mid 1 \leq i \leq N;\ i \leq j \leq N;\ 1 \leq k \leq \frac{j-i}{2} \,\}$.

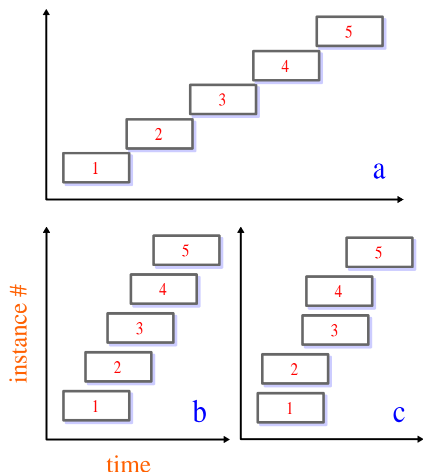


Figure 1. (a) The latency-optimal array executes five input instances in sequence and has an execution time of $5\mathcal{L}_{opt}$ clock cycles. **(b)** The array optimized for throughput pipelines a new input every $\frac{1}{3}\mathcal{L}_{opt}$ clock cycles. **(c)** When the array is not fully efficient (here 50%), we simultaneously pipeline two input instances every $\frac{2}{3}\mathcal{L}_{opt}$ clock cycles. Both throughput-optimized arrays are $2.3\times$ faster than the latency-optimal array.

3. Characterizing Throughput-Optimal Arrays

We say that a systolic array executes an *instance* $\mathcal{I}$ when the array evaluates the system of recurrence equations for a particular input. This input defines integer values for any size parameters of the parametrized recurrences. The computation time of $\mathcal{I}$, i.e. the time to evaluate all points in the domain $\mathcal{D}(\mathcal{I})$, is given by the latency $\mathcal{L}$ of the array's schedule. $\mathcal{L}$ is defined as the difference in execution times of its first and last scheduled points:

$$\mathcal{L} = \max\ \{\ \tau(z) \mid z \in \mathcal{D}(\mathcal{I})\ \} - \min\ \{\ \tau(z) \mid z \in \mathcal{D}(\mathcal{I})\ \}.$$

Existing work in array design has focused on finding a latency-optimal schedule $\mathcal{L}_{opt}$ [21, 22, 23, 24] to build high-performance parallel arrays. The total execution time for m equal-sized inputs on a latency-optimal array is $m\mathcal{L}_{opt}$.

We are more interested in the *throughput*, rather than the latency, of a systolic array. Throughput is the deciding factor when minimizing the *total* execution time of a large number of input instances. Define β, the *block pipelining period* of an array, as the earliest time after starting to compute on an input, at which computation on a second input on an array can be started safely, i.e. without the same processing element trying to compute on both inputs at the same time. The reciprocal of β is the throughput of the array measured in input instances processed per clock.

Consider the recurrence with computation domain $\mathcal{D}(\mathcal{I})$ on input instances $\mathcal{I}$ of some fixed size in all dimensions. Let $\tau(z) = \lambda z + \alpha$, $z \in \mathcal{D}(\mathcal{I})$ be some schedule for computing the recurrence for $\mathcal{I}$. Given a series of instances $\mathcal{I}_2, \mathcal{I}_3, \ldots$ of the same size as $\mathcal{I}$, we would like a *pipelined schedule* to compute them all efficiently. Since we know that successive instances of the recurrence can start at intervals of β without processor contention, we define a pipelined schedule of the kth instance as $f(k, z) = \tau(z) + (k-1)\beta$. A throughput-optimal array executes m instances of the input in time $(m-1)\beta + \mathcal{L}$, which is faster than the latency-optimal array if $\beta < \mathcal{L}_{opt}$ and $\mathcal{L}$ is not much larger than $\mathcal{L}_{opt}$. In optimizing for throughput, we may have to sacrifice latency, but this sacrifice can be amortized over a large number of inputs. The advantage of throughput-optimized arrays is illustrated in Figure 1.

Before we proceed further, we must clarify our definition of throughput for FPGAs. There are two ways to optimize the throughput of a systolic array on an FPGA. As described above, we may minimize the period, measured in clock cycles, between the pipelining of two instances of the problem on an array; alternatively, we may minimize the clock period itself. The frequency at which an array can be clocked depends on the length of the critical path in the design, which is almost always in the processing element. The critical path depends on the computations in the body of the recurrence and may be improved by judicious ordering of these operations and their optimal scheduling on any shared resources. Methods to minimize critical path length are well-known in the field of high-level hardware synthesis. For example, past work by Rosseel [25] builds "throughput-optimal" arrays from recurrences primarily by optimizing the scheduling of operations in each processing element, while minimizing the total resources consumed. The block pipelining period is treated as only a secondary consideration.

In our work, we assume that the design and clock period of a processing element are fixed and seek to minimize the block pipelining period between instances on the array. Maximizing throughput may increase the number of processing elements used, but this is acceptable provided the target device can accommodate them. Both our approach and critical-path optimization could be used together, with some compromise between the area of each processing element (more for faster processors) and the total number of elements on the array (more for lower values of β).

3.1 A Design Criterion for Throughput Optimality

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., their computation domains contain the same number of points such that any point $z \in \mathcal{D}(\mathcal{I}) \Leftrightarrow z \in \mathcal{D}(\mathcal{I}')$. We would like to derive a schedule to execute both instances one after the other in a pipelined fashion. The first instance $\mathcal{I}$ executes on the schedule $\tau(z)$ derived as in Section 2. We use a pipelined linear schedule f for $\mathcal{I}'$ given by $f(z') = \tau(z') + \beta$. To be feasible, β must be large enough to satisfy the constraint $f(z') > \tau(z)\ \forall\ z \in \mathcal{D}(\mathcal{I}), z' \in \mathcal{D}(\mathcal{I}')$.

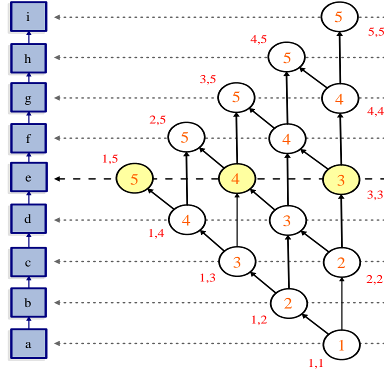
We clarify this concept using the example in Figure 2. Figure 2a shows a triangular lattice with its integer points represented by circles. The index vector of the two-dimensional points at the boundary is shown for clarity. Figure 2b shows two instances $\mathcal{I}$ and $\mathcal{I}'$ being pipelined through the array. The time dimension is along the x-axis.

Because the two instances are of the same size, they have identical lattices, and so we can express the feasibility constraint for a pipelined schedule as

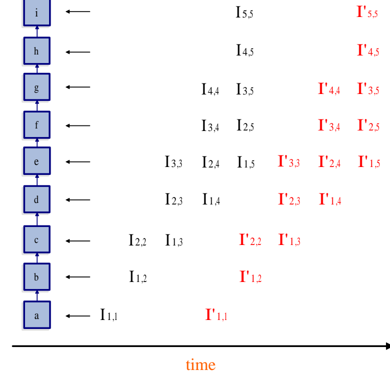
$$\forall\ z_1, z_2 \in \mathcal{D}(I) \text{ with } \pi(z_1) = \pi(z_2),\ f(z_1) > \tau(z_2). \quad (1)$$

Intuitively, f is a feasible schedule only if no two points from the two instances are assigned for execution on the same processing element at the same time.

The key question is, how small can we make β while satisfying the above constraint, i.e. how quickly can we start the next computation after the previous one has started? The answer clearly depends on the array design, and one would therefore expect the minimal β to be a function of both its projection vector u, which determines π, and its schedule λ. However, we now prove that the throughput of the array can be usefully lower-bounded *independent of the schedule* λ. This important observation allows us to effectively optimize for throughput in the next section by searching only the space of projection vector candidates, rather than having to jointly consider projections and schedules.



(a) The triangular lattice is projected horizontally along the dashed lines onto a linear array of processors. The execution time of every lattice point is shown in the circle. Processor e executes the maximum number of points: $k_{max} = 3$.



(b) We can pipeline two instances of the same size $\mathcal{I}$ and $\mathcal{I}'$ with a pipelining period $\beta = 3$. We are guaranteed that there will be no processor contention.

Figure 2. Pipelining the execution of two input instances on an array to improve throughput.

Preliminaries. Any two lattice points z_1 and z_2 in $\mathcal{D}(\mathcal{I})$ are executed by the same processing element if and only if

$$z_1 - z_2 = ku \; , \; k \in \mathbb{Z},$$

where u is the projection vector of the array. We denote the maximum number of points executed by any processing element as $k_{max} = 1 + k_{ilp}$ where

$$k_{ilp} = \max \; \{ \; k \mid z_1 - z_2 = ku, \; z_1, z_2 \in \mathcal{D}(\mathcal{I}) \; \}$$

Since the domain of the system of recurrence equations can be defined as a convex polyhedron $Cz \leq d$, we can find k_{ilp} by solving the following integer linear program:

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

The following theorem shows that $f(z)$ is a feasible pipelined schedule if and only if $\beta > k_{ilp}|\lambda u|$.

Theorem 1. $f(z) > \tau(z') \;\; \forall \; z, z' \in \mathcal{D}(\mathcal{I})$ with $\pi(z) = \pi(z')$ 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. Similarly, the point z_2 of the first instance must execute before z_1 of the second.

We therefore have

$$\begin{array}{rcl} f(z = z_2) & > & \tau(z' = z_1) \\ f(z_2) & > & \tau(z_2 + k_{ilp}u) \\ \lambda z_2 + \alpha + \beta & > & \lambda(z_2 + k_{ilp}u) + \alpha \\ \beta & > & k_{ilp}\lambda u. \end{array}$$

Similarly, 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 otherwise that $f(z) \leq \tau(z')$ for some $z, z' \in \mathcal{D}(\mathcal{I})$. Since $\pi(z) = \pi(z')$, we have $z' - z = ku$. WLOG, we assume that k is a positive integer. Then we have

$$\begin{array}{rcl} f(z) & \leq & \tau(z + ku) \\ \lambda z + \alpha + \beta & \leq & \lambda(z + ku) + \alpha \\ \lambda z + \beta & \leq & \lambda z + k\lambda u \\ \beta & \leq & k\lambda u \end{array}$$

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

Corollary. Given multiple instances of the recurrence, all of the same size, a valid schedule for the mth instance is given by $f(m, z) = \tau(z) + (m-1)\beta$. Indeed, we can show that the schedules for the mth and $m-1^{th}$ instances do not overlap using a proof similar to that of Theorem 1. By the transitive nature of a pipelined schedule, it follows that none of the schedules overlap. □

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

3.2 Example: the Nussinov Algorithm

Existing work on the Nussinov algorithm [26] shows two arrays derived from a three-dimensional uniform recurrence for it: the GKT and GJQ arrays with projection directions $[0, 0, -1]$ and $[-1, 0, 0]$ respectively. Both have a latency-optimal schedule $\tau(i, j, k) = -2i + 2j - k - 1$ and an execution time of $2N - 6$ clock cycles for a problem of size N. We can use our theorem above to show that these arrays permit pipelining to realize throughput greater than the trivial bound implied by their latency.

We retain the latency-optimal schedule and determine the throughput of the two arrays using Theorem 1. The throughput of the GKT array with projection direction $[0, 0, -1]$ is the reciprocal of $k_{max} = \frac{N-1}{2}$ and the array efficiency is 100%. By pipelining a new instance of the input every $\frac{N-1}{2}$ clock cycles, we obtain an implementation that is $4\times$ faster than the latency-optimal one.

Similarly, we derive the throughput of the GJQ array to be the reciprocal of $k_{max} = N - 2$, which is $2\times$ faster than the latency-optimal array. In this case, however, $|\lambda u| = 2$ and so we must simultaneously compute two instances of the input on the array.

3.3 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 2.3); confronted with an array that is not fully efficient, we can increase the throughput by simply 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 upper bounded by $\frac{1}{k_{max}}$ input instances per clock cycle. We conclude that *for any schedule, regardless of the utilization, the throughput is always at least as good as* $\frac{1}{k_{max}}$.

Knowing that throughput of a systolic array is effectively independent of its schedule greatly simplifies our exploration of array design space. We can search over possible projection vectors to find one with near-optimal throughput and only then derive its schedule, rather than having to consider both allocation and schedule simultaneously. More concretely, suppose we choose some projection vector u with throughput $\frac{1}{k_{max}}$ that satisfies the input bandwidth constraint. The system can transfer data words for one input into the array every k_{max} clock cycles. Select any schedule such that the utilization of the array is the reciprocal of $\gamma = |\lambda u|$. The system must now transfer data words for γ input instances in time $(k_{max} - 1)\gamma + 1$, which leaves the throughput and input bandwidth requirement nearly unchanged. Hence, we can be confident that the estimate $\frac{1}{k_{max}}$ is a good guide to the actual throughput of our chosen array, and that the bandwidth requirement remains feasible no matter the actual array utilization.

4. Finding Throughput-Optimal Projection Vectors

We now describe a procedure for searching the space of projection vectors u for a given set of recurrences to discover high-throughput array designs. We do not limit ourselves to a single array design; rather, we seek a collection of throughput-optimal arrays with a variety of different area requirements. Because the number of processing elements, and hence the area, in an array scales with its input size, a very high-throughput but area-intensive array may be feasible for small input sizes, while only lower-throughput arrays may be feasible for larger sizes. We can switch among these arrays by reconfiguration to process a collection of different-sized inputs as efficiently as possible, achieving better overall throughput than any single array. We refer the reader to recent work [14] for a description of one practical reconfiguration strategy.

It is instructive to study how variation in the projection vector affects the characteristics of the array. In general, increasing the magnitude of u increases throughput but may also increase the input bandwidth consumed, resource requirements, or the lengths of the interconnection network.

Increasing the magnitude of the projection vector decreases the number of points executed by each processing element in the array. This frees up processors sooner for the next input but comes at the cost of increased resource usage — more processing elements are required to execute all points in the domain for any one input. In the extreme case, we can increase the magnitude of u so much that only one point is executed on each processing element. The resulting array has the best possible block pipelining period of 1. In practice, however, we are limited by the on-chip resources and cannot use this extreme solution, so we assume in what follows that each processing element executes at least two lattice points in the domain. A more precise bound on throughput as a result of FPGA resource constraints is given later in this section.

Another constraint on our search is that keeping the array fully fed with input must not require more than the available input bandwidth into the FPGA. To enforce this constraint, we now show an bandwidth-derived upper bound on the magnitude of u, using a technique similar to that of Wong and Delosme [27] for finding a space-optimal allocation.

4.1 A search procedure for projection vectors

Our search for feasible projection vectors will be an enumerative approach using upper bounds on the magnitude of the vector. We will base these bounds on two important constraints: the input system bandwidth, and the available FPGA resources. The basic mathematical tools were first used by Wong and Delosme [27] to develop bounds for space-optimal projection vectors.

4.1.1 Basic Definitions

We describe fundamental definitions for convex bodies in this section. A detailed introduction is available in standard texts [28]. For a closed convex body $\mathcal{D}$, such as that described by a system of recurrence equations, the *support function* h is defined as

$$h(s) = \max \{ zs \mid z \in \mathcal{D} \},$$

where s is a unit vector of dimension n. Intuitively, the support function gives the distance along direction s from the origin to the support plane of $\mathcal{D}$ that has exterior normal vector s (see Figure 3). We define the *width* of $\mathcal{D}$ in the direction s as the distance between

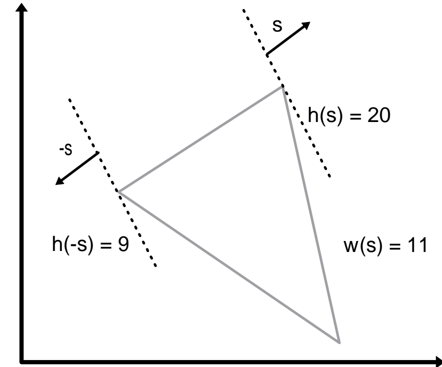


Figure 3. Width of a convex body in the direction s. The number of lattice points along direction s is $1 + w(s)$.

the two support planes of $\mathcal{D}$ orthogonal to s and $-s$:

$$w(s) = h(s) + h(-s).$$

We now present three facts we will use in deriving our bounds. We refer the reader to Wong and Delosme [27] for their proofs.

Fact 1. Given a projection direction u (not necessarily a unit vector) the maximum number of lattice points executed on any processing element, k_{max}, is at most

$$1 + \left\lfloor \frac{w(u)}{|u|^2} \right\rfloor.$$

Fact 2. An upper bound on $w(u)$ can be shown to be

$$w(u) \leq |u|\sqrt{w(e_1)^2 + w(e_2)^2 + \ldots + w(e_n)^2},$$

where e_i is the ith canonical row vector whose i^{th} element is one and all others are zero; $w(e_i)$ is the width of the lattice along the ith dimension.

Fact 3. If the projection of $\mathcal{D}$ by a vector u is $\mathcal{D}_u$, the number of lattice points in $\mathcal{D}_u$ is bounded from below by

$$|\mathcal{D}_u| \geq \frac{|\mathcal{D}|}{1 + \left\lfloor \frac{w(u)}{|u|^2} \right\rfloor}.$$

4.1.2 Input bandwidth bound

We will now develop a bound on the magnitude of the projection vector using the input bandwidth constraint. The bandwidth requirement of the array is determined by the data requirement per input instance and the throughput of the array. We will relate the throughput, given by k_{max}, to the width function and use it to derive a bound on $|u|$.

We assume that the final clock period of the array is known. For a systolic array this will likely be determined by the logic in a processing element. Routing delays on long wires and controller logic may also affect the clock period but there are techniques to estimate this period for systolic arrays [25]. We can use one of many heuristic techniques [30] or simply synthesize a single processing element to estimate its period. We then compute the system input bandwidth: m data bits per clock period of the array. We are given the data requirements of the array, say b data bits per input instance. In general this is a function of the input size (recurrence size parameters). In what follows, we assume the size parameters have been fixed.

For the input bandwidth constraint to hold, the array must satisfy the following equation

$$\frac{b}{k_{max}} \quad \leq \quad m.$$

Rearranging terms, we have

$$k_{max} \quad \geq \quad \frac{b}{m}.$$

We assume that $b > m$, since otherwise the recurrence is not bandwidth limited and we can select any u such that only one lattice point is executed on every processing element; there is no need of a search.

We can use Fact 1 to inject the projection vector into the bandwidth constraint:

$$1 + \left\lfloor \frac{w(u)}{|u|^2} \right\rfloor \quad \geq \quad \frac{b}{m}.$$

Since we assume (though it is not essential to our derivation) that each processor executes at least two lattice points, $k_{max} \geq 2$, and so

$$\left\lfloor \frac{w(u)}{|u|^2} \right\rfloor \quad \geq \quad 1.$$

Simplifying, we have

$$\frac{2w(u)}{|u|^2} \quad \geq \quad \frac{b}{m}.$$

Using Fact 2 we substitute for $w(u)$ to get

$$\frac{2|u|}{|u|^2}\sqrt{w(e_1)^2 + w(e_2)^2 + \ldots + w(e_n)^2} \quad \geq \quad \frac{b}{m},$$

which simplifies to

$$|u| \quad \leq \quad \frac{2m}{b}\sqrt{w(e_1)^2 + w(e_2)^2 + \ldots + w(e_n)^2}. \qquad (3)$$

We now compute the bandwidth bound for the sorting, Smith-Waterman and Nussinov recurrences. Our FPGA system supports an input bandwidth of 64 bits per clock at 133 MHz; we assume the generated hardware arrays will clock at least as fast. These numbers are representative of modern FPGA systems.

For the sorting application, we assume it operates on N 32-bit integers. We have $w(e_1) = N$ and $w(e_2) = N + 1$, and the bound on the magnitude of the projection vector for $N = 10, 100,$ or 1000 is 6.

Take an instance of the banded Smith-Waterman algorithm that aligns a query to a target protein sequence, each of length N. The characters of a protein sequence may be represented using 5 bits. We assume the query is fixed and each input instance is a new target sequence. The width of the lattice along the canonical row vectors is $N - 1$; for $N = 300$ and 500, the bound is 37. If instead the array was used to align 16-bit unicode text documents we get a more reasonable bound of 12 for documents of length 300.

The Nussinov algorithm folds an RNA of length N, where each character of the sequence can be represented in 3 bits. The width of the lattice along the three canonical row vectors are $N - 3$, $N - 3$, and $\frac{N-3}{2}$. Bounds for $N = 25$ and 50 are 57 and 61, too large to be useful. This is due to the fact that with just three bits per character, the algorithm is far from being bandwidth-constrained.

We conclude that the bandwidth bound is useful in restricting the search space, but only if the recurrence has high data requirements, or the bandwidth into the FPGA system is limited.

4.1.3 FPGA resource bound

In the case of an FPGA target we are uniquely able to exploit the resource constraint, i.e., the number of processing elements that can be instantiated on the device. Unlike an ASIC, an FPGA has a fixed number of logic gates that may be used to instantiate processing elements—there is no penalty in using as many of them as required.

The regular nature of systolic arrays with recurring processing elements enables us to predict the resource usage with relative confidence. The primary contribution to area is the logic within the processing element which can be estimated from the body of a recurrence. Handling logic predicated by conditional statements in a recurrence is challenging since depending on the projection direction some processing elements may not need to instantiate the logic and will use fewer resources. We may overcome this problem by either taking a conservative approach and overestimating the number of processing elements that can be instantiated, or by computing an average processing element based on the size of conditional subdomains.

Existing work [30] estimates the number of logic gates used on a specific FPGA device for any hardware circuit in a high level description. The authors are able to estimate resource usage for large designs to within 5% of the true value in less than a second. Unfortunately, we did not have access to similar tools so we predict the maximum number of processing elements that can be placed on an FPGA device using actual synthesis results. This is done by building a single processing element using the Synplify synthesis tool (execution time is in the order of minutes).

Let the predicted maximum number of processing elements that can be instantiated on the FPGA device be p. Projection of the domain $\mathcal{D}$ by the vector u induces the processor space $\mathcal{D}_u$. To satisfy the area constraint, we must have

$$|\mathcal{D}_u| \quad \leq \quad p.$$

Substituting the lower bound in Fact 3 we get

$$\frac{|\mathcal{D}|}{1 + \left\lfloor \frac{w(u)}{|u|^2} \right\rfloor} \quad \leq \quad p.$$

Using the two-points-per-processor constraint as in the previous section, we obtain

$$\frac{|\mathcal{D}|}{\frac{2w(u)}{|u|^2}} \quad \leq \quad p.$$

Substituting Fact 2 for $w(u)$ and simplifying, we get the bound

$$|u| \quad \leq \quad \frac{2p}{|\mathcal{D}|}\sqrt{w(e_1)^2 + w(e_2)^2 + \ldots + w(e_n)^2}. \qquad (4)$$

Note the similarity between the bandwidth and area bounds.

Returning to the Nussinov algorithm, we generated a processing element with a 16-bit datapath using all logic resources on a Xilinx Virtex-4 LX100-12 FPGA. From this we predict (optimistically) that at most 700 processing elements can be synthesized. For $N = 25$ and 50 the area bound is 38 and 10 respectively. The latter is a workable bound for an enumerative search process. With a 32-bit datapath we get a prediction of 327 processing elements and a bound of 18 and 5 for the two lengths.

For the banded Smith-Waterman algorithm, we are able to support up to 480 9-bit processing elements on the FPGA device; here the limiting factor is the number of available block ram memories. We used a band length of 66 and sequences of 300 and 500 characters, which are reasonable for typical protein sequences. In each case the bound derived is 22, slightly lower than the one previously derived. In a practical implementation banded Smith-Waterman is likely to be one of many hardware stages on the same FPGA and will have fewer available resources leading to a better bound.

5. Finding Compatible Schedules

Given a projection vector, we use standard techniques [21, 22, 23, 24] to derive a linear schedule satisfying the constraints in Section 2.1. The schedule must also not conflict with the selected projection vector.

We formulate an integer linear program using the Vertex method to find the lowest latency schedule. As noted in Section 3, however, the effect of latency on the total execution time of a large number of inputs is negligible. We have another criterion that may be optimized: the reciprocal of array utilization γ. An array that is not fully efficient requires pipelining of multiple inputs simultaneously. As a practical matter, the hardware implementation requires γ independent buffers at the input and output side (to interleave and deinterleave the input/output). We may choose to minimize γ when selecting the schedule or limit it to a user-defined cut-off. In this work we use a weighted objective function that minimizes both γ and the latency, with the utilization being given a higher priority.

6. 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, so that we can reuse such a package's 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 [31] for polyhedral manipulations, the PIP library [32] for solving integer linear programs, and the Barvinok library [33] for counting the number of integral points in a polyhedron.

The main loop shown in Algorithm 1 iterates over the space of candidate projection vectors analyzing their throughput and finding compatible schedules. The output is a collection of arrays, one for every distinct value of throughput. The selected schedule and allocation functions may be passed on to MMAlpha for array generation. Finally, the program detailed in [14] selects a sequence of one or more of these arrays depending on the input distribution for use on FPGA devices that support run-time reconfiguration.

7. Results

Tables 1-3 show some of the projection vectors suggested by our software on the three example recurrences. In each case, the latency-space optimal array is shown within the first three rows. For every projection vector, we are able to find a schedule that maximizes utilization with only a minor increase in latency that can easily be amortized over hundreds of inputs. The average length of the links in the array's interconnection network increases for high throughput arrays. Banded Smith-Waterman shows a number of attractive arrays with throughputs equal to multiple fractions of the width of the band. Protein sequences have an average length of 300 but may be several times longer, allowing us to use a combination of these arrays on a reconfigurable target.

The results for Nussinov show the two standard arrays: -1 0 0 and 0 0 -1. As mentioned in Section 3.2, in the case of the former, the array utilization is 50%, and we have to simultaneously pipeline two RNA sequences through the array. Our tool suggests an alternate projection 1 1 0 with the same throughput and area requirement that is fully efficient. Latency increases $1.5\times$ but is eas-

Algorithm 1 Explore allocation/schedule space

```
 1: procedure EXPLORE(Recurrence, Parameter Instantiations)
 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:         ll ← AverageLinkLength(π)
 9:                 ▷ Links in the induced interconnection network
10:         #PE ← NoPEs(π, τ)
11:                 ▷ Count the number of PEs in the processor space
12:     end for
13:
14:     Sort the solutions by T, #PE, γ and ll
15:     Select one array for every distinct T
16: end procedure
```

Table 1. Some arrays showing the throughput-area tradeoff for the sorting recurrence. The example assumes 100 32-bit integers and a bound of 6.

u	k_{max}	$\#PEs$	γ	**Latency**	**Avg.** ll
0 1	$N+2$	101	1	201	0.5
1 0	$N+1$	102	1	201	0.5
1 1	$N+1$	102	2	201	1.0
1 -1	$\frac{N+3}{2}$	202	1	301	1.0
2 -1	$\frac{N+4}{3}$	302	1	201	1.5
3 -1	$\frac{N+5}{4}$	401	1	302	2.0
3 -2	$\frac{N+6}{5}$	499	1	201	2.5
5 -1	$\frac{N+7}{6}$	596	1	504	3.0
4 -3	$\frac{N+8}{7}$	692	1	201	3.5

Table 2. Throughput-area tradeoff for banded Smith-Waterman for sequences of length 300 and a band width of 66. A bound of 22 was used.

u	k_{max}	$\#PEs$	γ	**Latency**	**Avg.** ll
1 1	N	66	2	598	0.7
1 0	66	300	1	598	0.7
0 1	66	300	1	598	0.7
1 -1	33	599	1	897	1.3
2 -1	22	898	1	598	2.0
3 -1	17	1197	1	897	2.7
3 -2	14	1494	1	598	3.3
4 -3	10	2088	1	598	4.7
3 -5	9	2385	1	897	5.3

Table 3. Throughput-area tradeoff for the Nussinov recurrence with $N = 51$ and a bound of 10.

u	k_{max}	$\#PEs$	γ	**Latency**	**Avg.** ll
1 1 0	$N-2$	625	1	144	1.6
-1 0 0	$N-2$	625	2	96	0.9
0 1 0	$N-2$	625	2	96	1.1
0 0 -1	$\frac{N-1}{2}$	1225	1	96	1.1
1 1 -1	$\frac{N}{3}$	1801	1	96	2.0
0 1 2	$\frac{N+1}{4}$	2353	1	144	1.7
2 1 -2	$\frac{N+2}{5}$	2882	1	144	2.9
0 1 3	$\frac{N+3}{6}$	3388	1	192	2.1
3 3 2	$\frac{N+4}{7}$	3872	1	144	2.7

ily amortized. Many other projections with lower block pipelining periods are presented.

As expected, an increase in throughput causes a proportional increase in the number of processing elements. One might be tempted to simply use multiple instantiations of the latency-space optimal array on the FPGA device. This approach, however, is not scalable with the number of parallel instantiations. Having y parallel units increases the number of processing elements performing I/O by a factor of y. For example, in the Nussinov example each array has N processing elements that read the input sequence. Furthermore, each unit requires independent I/O buffers or a bus with potentially long delays. In the Smith-Waterman example, we need five parallel units of array 1 1 to achieve the same throughput as array 1 0. Using a single high throughput array does not cause a similar increase in I/O to processing elements and each new problem instance can be loaded sequentially in time.

The bandwidth and area bounds we have developed are quite loose, as can be seen from the results; however, the execution time of the design space exploration tool is not a concern. The sorting and Smith-Waterman recurrences run in under 5 seconds and Nussinov in 35 seconds on an Intel Core 2 Duo workstation. In these three cases, 36, 464 and 1729 candidate projection vectors were explored. These running times compare quite favorably to the hardware synthesis time, which amounts to several hours per array design.

Next we used our software to predict the performance of a Nussinov accelerator on a Xilinx Virtex-5 LX330-2 FPGA. We do not have physical access to the device but would like to estimate the speedup possible by moving to the newer generation of FPGA devices.

We synthesized a processor with a datawidth of 5 and estimated that at most 2590 of them may be placed on the target. We generated the list of arrays for $N = 50$ (runtime 5.5 minutes) using our tool and fed the algorithm outlined in [14] to estimate the speedup on the reconfigurable target. In the referenced work the authors synthesize each array to determine the largest array size that can be placed on the FPGA. Here we approximate it using the resource requirements of a processor from array 0 0 -1 (and is optimistic). The task is to fold a dataset of 22.6 million pyrosequencing reads that is sorted by sequence length.

The reconfiguration strategy predicted is shown in Figure 4. Without reconfiguration the single latency-space optimal array 1 1 0 is used (pipelining a new sequence every $N-2$ clock cycles). Allowing for reconfiguration, the algorithm predicts a 37% speedup (2.7$\times$ speedup if the single array optimizes for latency of an individual sequence instead of overall execution time). This is despite most of the input being biased toward the lowest throughput array.

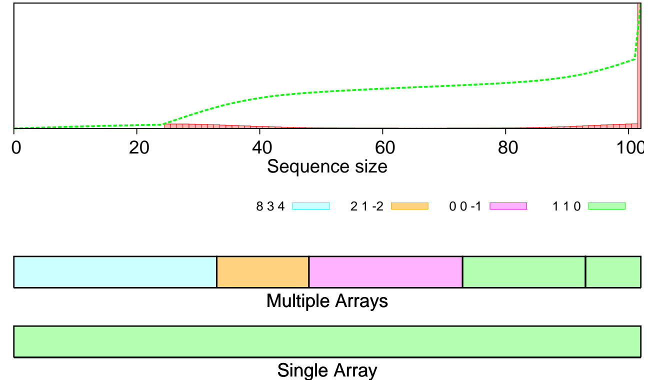


Figure 4. Optimal selection of Nussinov arrays to fold pyrosequencing reads. Top: histogram and cumulative frequency of sequence lengths. Middle: design with reconfigurations. Bottom: best single array supporting all input lengths (up to 102 bases). Size of each array instantiation is given by length of longest sequence it processes.

8. Related Work

Decades of research have gone into the automatic synthesis of systolic arrays from systems of recurrence equations using the polyhedral model [9]. As we have mentioned, the practice has been to find a single array that has a latency-optimal schedule and a corresponding allocation that minimizes the number of processors generated [12]. All these methods generate an $n-1$ dimensional processor space with unidimensional time from a recurrence of n dimensions.

The latency-optimal schedule is found using integer linear programming methods by inspecting the execution times at the vertices of the domain [21, 23, 24] or by application of Farkas' Lemma on the polyhedron's faces [22]. We use the Vertex method to find suitable schedules, though minimizing its latency is not as important as increasing the array's utilization.

Wong [27] details an enumerative search technique to minimize the number of processors in an allocation which is used by PARO [16]. They derive bounds using tools from the geometry of numbers. We have reused these ideas to derive input bandwidth and area bounds applied to maximizing throughput.

Another option is to limit the search to allocation functions that instantiate arrays that have a local interconnection network [29]. This approach does an exhaustive search over all allocation functions that are guaranteed to generate interconnections that are nearest-neighbor. It works well if we are certain of the interconnection network desired, e.g. the four nearest neighbors. In many cases, however, we wish to allow a small number of long-range links on our FPGA device. Making such a compromise greatly increases the search space.

Tools implementing these techniques to synthesize hardware arrays include MMAlpha [15], PARO [16] and PICO-NPA [17]. Our design space exploration tool may be integrated into these packages.

There are two major differences between our work and related work: we optimize for throughput, and we identify multiple arrays that tradeoff throughput for FPGA resources.

The only work that we are aware of to optimize the throughput using space-time methods is by Rosseel [25]. The authors target real-time signal processing applications in video, speech and image processing that require high throughput rather than low latency. The goal is to find a single array that matches the requested throughput (for some fixed recurrence parameters) and minimizes

the area requirement. They also use an enumerated search procedure. The authors recognize that throughput is dependent on the block pipelining period but do not directly minimize k_{ilp}; rather they attempt to estimate the product $k_{ilp}|\lambda u|$ through a multi-phase interleaved search through the allocation and schedule spaces. In our procedure, we estimate the block pipelining period using only the projection vector, so we can search the two spaces independently. Priority is given to optimizing the processing element by minimizing the area-operation interval product. This step can be integrated into our method to optimize the processing element design. Low utilization processing elements are not preferred because they must be clustered and increase area requirements; we simply pipeline multiple input instances. Of the allocation candidates that match the requested throughput, they select ones that increase average usage of the processors. Finally, the allocation matrices they consider have elements typically restricted to $0, \pm 1$ to reduce execution time, so they may not find many interesting space mappings.

Recent work [14] exploits run-time reconfiguration on FPGAs using multiple arrays that tradeoff throughput for area rather than using the single latency-space optimal array. The authors give a dynamic programming algorithm to select a sequence of one or more of these arrays that executes small inputs on high throughput and large inputs on low throughput arrays. Our work can be used in a prior step to produce the candidate arrays from which the set is selected based on the input distribution.

We do not deal with the important problem of partitioning to process inputs that are too large to fit on an FPGA device or recurrences of more than three dimensions. Recent work by Bondhugula et al. [34] generates a partitioned array on an FPGA (this reference also details related work). This is complementary to our work; a partitioned array for large inputs may be used as one of the reconfiguration candidates.

9. Conclusion

In this work we have introduced a procedure to systematically find throughput-optimal systolic arrays from uniform recurrence equations. We are motivated by throughput rather than latency as a performance metric and the use of a reconfigurable target that can select from multiple arrays which tradeoff throughput for area. We describe how to optimize for throughput by only inspecting the projection vector space and derive bounds based on bandwidth and area constraints for an enumerative search. We have shown results for a number of algorithms including the Nussinov RNA folding recurrence.

In the future we plan to research extensions of this approach to locally parallel globally sequential partitioned arrays. We may be able to use similar ideas to pipeline multiple iterations on the partitioned array to improve throughput.

Acknowledgments

This work was supported by NIH award R42 HG003225 and NSF awards DBI-0237902 and ITR-427794. R.D. Chamberlain is a principal in BECS Technology, Inc.

References

[1] H.T. Kung. Why systolic architectures? *Computer*, 15(1):37–46, Jan 1982.

[2] SRC Computers, Inc. *Series H and I MAP processors*, http://www.srccomp.com/.

[3] XtremeData, Inc. *XD2000F development system*, http://www.xtremedatainc.com/.

[4] DRC Computer Corporation. *DS2000 development system*, http://www.drccomputer.com/.

[5] Nallatech. *Intel Xeon FSB FPGA Accelerator Module*, http://www.nallatech.com/.

[6] Tarek El-Ghazawi, Esam El-Araby, Miaoqing Huang, Kris Gaj, Volodymyr Kindratenko, and Duncan Buell. The promise of high-performance reconfigurable computing. *Computer*, 41(2):69–76, 2008.

[7] S. Derrien and P. Quinton. Parallelizing HMMER for hardware acceleration on FPGAs. *International Conference on Application-specific Systems, Architectures and Processors*, pages 10–17, July 2007.

[8] Dominique Lavenier, Patrice Quinton, and Sanjay Rajopadhye. Advanced Systolic Design, in Digital Signal Processing for Multimedia Systems, Chapter 23, Parhi and Nishitani eds, March 1999.

[9] Patrice Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proceedings of the 11th annual International Symposium on Computer Architecture*, pages 208–214, 1984.

[10] Yoshiki Yamaguchi, Tsutomu Maruyama, and Akihiko Konagaya. High speed homology search with FPGAs. In *Proceedings of Pacific Symposium on Biocomputing*, pages 271–282, 2002.

[11] R. P. Jacobi, M. Ayala-Rincon, L. G. A. Carvalho, C. Llanos, and R. Hartenstein. Reconfigurable Systems for Sequence Alignment and for General Dynamic Programming. *Genetics and Mol. Research*, 4(3):543–552, 2005.

[12] Frank Hannig and Jürgen Teich. Design space exploration for massively parallel processor arrays. In *PaCT '01: Proceedings of the 6th International Conference on Parallel Computing Technologies*, pages 51–65, 2001.

[13] B. C. Brodie, R. D. Chamberlain, B. Shands, and J. White. Dynamic reconfigurable computing. In *Military and Aerospace Programmable Logic Devices*, 2003.

[14] Reference withheld, September 2009.

[15] A.C. Guillou, P Quinton, T. Risset, C. Wagner, and D Massicotte. High level design of digital filters in mobile communications. In *DATE Design Contest*, March 2001.

[16] Frank Hannig, Holger Ruckdeschel, Hritam Dutta, and Jürgen Teich. PARO: Synthesis of Hardware Accelerators for Multi-Dimensional Dataflow-Intensive Applications. In *Proceedings of the Fourth International Workshop on Applied Reconfigurable Computing (ARC)*, Lecture Notes in Computer Science (LNCS), London, United Kingdom, March 2008. Springer.

[17] Robert Schreiber, Shail Aditya, Scott Mahlke, Vinod Kathail, B. Ramakrishna Rau, Darren Cronquist, and Mukund Sivaraman. PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators. *Journal of VLSI Signal Processing Systems*, 31(2):127–142, 2002.

[18] S.K. Rao and T. Kailath. Regular iterative algorithms and their implementation on processor arrays. *Proceedings of the IEEE*, 76(3):259–269, Mar 1988.

[19] Kun-Mao Chao, William R. Pearson, and Webb Miller. Aligning two sequences within a specified diagonal band. *Comput. Appl. Biosci.*, 8(5):481–487, 1992.

[20] S. Y. Kung. *VLSI array processors*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[21] S. Balev, P. Quinton, S. Rajopadhye, and T. Risset. Linear programming models for scheduling systems of affine recurrence equations—a comparative study. In *SPAA '98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 250–258, New York, NY, USA, 1998. ACM.

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

[23] B. Lisper. Linear programming methods for minimizing execution time of indexed computations. In *In P. Feautrier and F. Irigoin, editors, International Workshop on Compilers for Parallel Computers*, pages 131–142, 1990.

[24] C. Mauras, P. Quinton, S. Rajopadhye, and Y. Saouter. Scheduling affine parameterized recurrences by means of variable dependent timing functions. *Proceedings of the International Conference on Application Specific Array Processors*, pages 100–110, Sep 1990.

[25] J. Rosseel, F. Catthoor, and H. De Man. An optimisation methodology for array mapping of affine recurrence equations in video and image processing. In *Proceedings of the International Conference on Application-specific Systems, Architectures and Processors*, pages 415–426, August 1994.

[26] A. Jacob, J. Buhler, and R. Chamberlain. Accelerating Nussinov RNA secondary structure prediction with systolic arrays on FPGAs. In *Proceedings of the International Conference on Application Specific Array Processors*, pages 191–196. IEEE Computer Society, 2008.

[27] Y. Wong and J.-M. Delosme. Space-optimal linear processor allocation for systolic arrays synthesis. In *Proceedings of the 6th International Parallel Processing Symposium*, pages 275–282, 1992.

[28] R. Schneider. *Convex bodies : the Brunn-Minkowski theory*. Cambridge University Press, Cambridge, 1993.

[29] X. Zhong, S. Rajopadhye, and I. Wong. Systematic generation of linear allocation functions in systolic array design. *J. VLSI Signal Process. Syst.*, 4(4):279–293, 1992.

[30] Dhananjay Kulkarni, Walid A. Najjar, Robert Rinker, and Fadi J. Kurdahi. Compile-time area estimation for LUT-based FPGAs. *ACM Transactions on Design Automation of Electronic Systems*, 11(1):104–122, 2006.

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

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

[33] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica*, 48(1):37–66, June 2007. `http://www.kotnet.org/~skimo/barvinok/`.

[34] Uday Bondhugula, J. Ramanujam, and P. Sadayappan. Automatic mapping of nested loops to FPGAS. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 101–111, New York, NY, USA, 2007. ACM.