January 2010

Parallelization of dynamic programming recurrences in computational biology

Arpith Jacob
Washington University in St. Louis

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

Recommended Citation

Jacob, Arpith, "Parallelization of dynamic programming recurrences in computational biology" (2010). *All Theses and Dissertations (ETDs)*. 169.
https://openscholarship.wustl.edu/etd/169

WASHINGTON UNIVERSITY IN ST. LOUIS

School of Engineering and Applied Science

Department of Computer Science and Engineering

Thesis Examination Committee:
Jeremy Buhler, Chair
Michael Brent
Ron Cytron
Mark Franklin
Robert Morley
Bill Smart
David Taylor

PARALLELIZATION OF DYNAMIC PROGRAMMING RECURRENCES IN

COMPUTATIONAL BIOLOGY

by

Arpith Chacko Jacob

A dissertation presented to the Graduate School of Arts and Sciences
of Washington University in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

December 2010
Saint Louis, Missouri

ABSTRACT OF THE THESIS

Parallelization of dynamic programming recurrences in computational biology

by

Arpith Chacko Jacob

Doctor of Philosophy in Computer Science

Washington University in St. Louis, 2010

Research Advisor: Professor Jeremy Buhler

The rapid growth of biosequence databases over the last decade has led to a performance bottleneck in the applications analyzing them. In particular, over the last five years DNA sequencing capacity of next-generation sequencers has been doubling every six months as costs have plummeted. The data produced by these sequencers is overwhelming traditional compute systems. We believe that in the future compute performance, not sequencing, will become the bottleneck in advancing genome science.

In this work, we investigate novel computing platforms to accelerate dynamic programming algorithms, which are popular in bioinformatics workloads. We study algorithm-specific hardware architectures that exploit fine-grained parallelism in dynamic programming kernels using field-programmable gate arrays (FPGAs). We advocate a high-level synthesis approach, using the recurrence equation abstraction to represent dynamic programming and polyhedral analysis to exploit parallelism.

We suggest a novel technique within the polyhedral model to optimize for throughput by pipelining independent computations on an array. This design technique improves on the state of the art, which builds latency-optimal arrays. We also suggest a method to dynamically switch between a family of designs using FPGA reconfiguration to achieve a significant performance boost. We have used polyhedral methods to parallelize the Nussinov RNA folding algorithm to build a family of accelerators that can trade resources for parallelism and are between 15-130x faster than a modern dual core CPU implementation. A Zuker RNA folding accelerator we built on a single workstation with four Xilinx Virtex 4 FPGAs outperforms 198 3 GHz Intel Core 2 Duo processors. Furthermore, our design running on a single FPGA is an order of magnitude faster than competing implementations on similar-generation FPGAs and graphics processors.

Our work is a step toward the goal of automated synthesis of hardware accelerators for dynamic programming algorithms.

Acknowledgments

I am deeply indebted to my advisor Dr. Jeremy Buhler, who has patiently guided me through my entire graduate career. I am thankful for the hours of discussions, all the comments and suggestions, and for helping me successfully reach this stage in my career. I would like to thank Dr. Buhler for taking a chance on me all these years ago, and for his continued support through the "known unknowns" and the "unknown unknowns." Dr. Buhler has spent countless hours reviewing and perfecting my work. I am eternally grateful for your support.

I would like to thank Dr. Roger Chamberlain with whom I have also worked closely during my entire graduate career. I am extremely grateful for his confidence in me and the many hours spent helping me improve my work. Dr. Buhler and Dr. Chamberlain have always afforded me great independence, and even played along with some of my crazier ideas. You have taught me how to do research and I have thoroughly enjoyed working with both of you.

I would like to thank members of my committee: Dr. Mark Franklin, Dr. Ron Cytron, Dr. Michael Brent, Dr. Bill Smart, Dr. Robert Morley, and Dr. David Taylor. You have asked the hard questions and forced me to consider my research in the larger context. In particular, I would like to thank Dr. Franklin, who showed me the importance of communicating research results through the (always merciless, sometimes frustrating) culling of technical detail in presentations. Thank you all for sacrificing your valuable time for me.

I would like to thank members past and present of the High Performance Computational Biology group at Washington University. Special thanks to Joseph Lancaster with whom I have worked for the last six years. I am especially grateful for your patience in the early years as I came to grips with the world of hardware programming—I can only imagine how frustrating I might have been! I would also like to thank Praveen Krishnamurthy, Eric Tyson, Brandon Harris, Peng Li, Lin Ma, Rahav Dor,

<div align="right">Arpith Chacko Jacob</div>

Washington University in Saint Louis
December 2010

Unless the LORD builds the house,

  its builders labor in vain.

Unless the LORD watches over the city,

  the watchmen stand guard in vain.

In vain you rise early

  and stay up late,

toiling for food to eat—

  for He grants sleep to those He loves.

                    Psalm 127:1-2

He named it Ebenezer, saying,

"Thus far has the LORD helped us."

                    1 Samuel 7:12b

Contents

List of Tables

List of Figures

xiii

Chapter 1

Introduction

Dynamic programming (DP) is a computational technique based on the principle of optimality [15], which states that the optimal solution to a subproblem is part of the optimal solution to a larger problem. DP has been used to efficiently solve many combinatorial optimization problems in polynomial space and time. This technique has become widely used in fields such as control theory, operations research, finance, computer vision, speech recognition, and computational biology.

Our domain of interest is computational biology, where DP is widely used to process biosequence information. The recent arrival of high-throughput *next-generation* sequencing machines [134] allows researchers to generate millions of short sequence reads in parallel at low cost. Next-generation sequencers have enabled researchers [106] to conduct large-scale studies to detect mutations in genomes that determine phenotypes and predict risk factors of an individual, detect antibiotic resistant pathogens [70] and identify their integration sites in a host genome [159], and discover novel noncoding RNAs [85]. Another major initiative aims to study the complex microbial community that inhabit the human body [150] to understand how these organisms may regulate the intake of nutrients by individuals (and hence their susceptibility to malnourishment and obesity), drug metabolism, and immune response, and how they can even cause behavioral and psychiatric disorders.

Fast computational analyses will play a critical, and we argue, a central role, in enabling researchers realize this vision. In this dissertation we explore novel compute architectures, high-level abstractions and efficient parallelization strategies to accelerate DP algorithms used for bioinformatics analyses by one to two orders of magnitude.

1.1 Dynamic Programming in Computational Biology

The sequencing of the human genome has ushered in an era where computational methods play a central role in understanding the molecular processes sustaining life. The DP technique has seen use in computational biology since the early 1970s for pairwise sequence comparison, multiple alignment and secondary structure prediction [131]. In Fall 2010, a search for the phrase "dynamic programming" in the journal of *Bioinformatics*, *BMC Bioinformatics*, and *IEEE/ACM Transactions on Computational Biology and Bioinformatics* returned over 1400 articles.

Table 1.1 lists DP algorithms from the computational biology literature. It includes 25 algorithms classified under eight categories. Alignment using probabilistic [43] and non-probabilistic [57] models is most popular and has been applied to numerous biological problems. Hidden Markov models [73] have been applied to motif identification [43] and gene prediction [24]. Parsing of hidden Markov models can be done efficiently using DP. Stochastic Context-Free Grammars [129] offer a richer framework to fold RNA and protein sequences to determine their most likely secondary structure. Conditional Random Fields [92] are a new modeling technique introduced within the last several years and look set to produce more accurate results on a number of these biological problems. Most of the algorithms listed in the table represent a family of solutions that have subtle but important variations depending on the biological problem being solved.
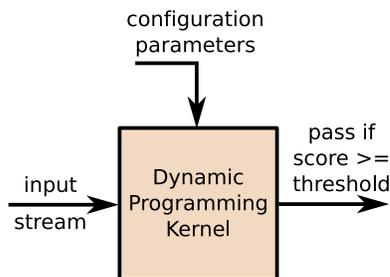
Figure 1.1: A *filter* computes the score of inputs using a DP kernel and discards those that do not cross a threshold. As a result, finding the score of the optimal solution using DP is more important than the state path that leads to the solution.

Table 1.1: A compilation of dynamic programming algorithms from computational biology. A detailed description of these algorithms, including the time and space growth functions, are in Appendix A. Legend for Accelerators: F - FPGA, G - GPU, CB - IBM Cell broadband, S - SIMD extensions on a workstation, CL - Cluster of workstations

Algorithm	Time Complexity	Space Complexity	Accelerator	Comments
Alignment				
Global/Local	$O(n^2)$	$O(n^2)$	F, G, CB, S, CL	n - sequence length.
Arbitrary Gaps	$O(n^3)$	$O(n^2)$	CL	
Profile HMMs				
Full Viterbi	$O(nl^2)$	$O(nl)$		n - sequence length.
Full Forward	$O(nl^2)$	$O(nl)$		l - model length.
Plan7 Viterbi	$O(nl)$	$O(nl)$	F, G, CB, S, CL	
Plan7 Forward	$O(nl)$	$O(nl)$		
Posterior-Viterbi	$O(nl)$	$O(nl)$		
HMM-HMM Alignment	$O(l^2)$	$O(l^2)$		
MSA-HMM Alignment	$O(nl)$	$O(l)$		
Gene Prediction				
GHMM Viterbi	$O(n^3l^2)$	$O(nl)$		m,n - sequence length.
GHMM Forward	$O(n^3l^2)$	$O(nl)$		l - model length.
GPHMM Viterbi	$O(mnl^2d^4)$	$O(mnl)$		d - maximum length of
GPHMM Forward	$O(mnl^2d^4)$	$O(mnl)$		observations in each state.
GeneWise	$O(mn)$	$O(mn)$		
Secondary Structure				
Nussinov	$O(n^3)$	$O(n^2)$	F, CL	n - sequence length.
Zuker	$O(n^4)$	$O(n^2)$	CL	w - window length.
Zuker and Lyngsø	$O(n^3)$	$O(n^2)$	CL	
PKNOTS	$O(n^6)$	$O(n^4)$		
Rnall	$O(w^3n)$	$O(w^3)$		
Simultaneous Sequence and Structural Alignment				
Sankoff	$O(n^6)$	$O(n^4)$		n - sequence length.
FOLDALIGN	$O(n^4)$	$O(n^4)$		
Stochastic Context-Free Grammars				
Inside/Outside	$O(l^2n^3)$	$O(ln^2)$	F	m,n - sequence length.
CYK	$O(l^2n^3)$	$O(ln^2)$	F	l - number of states in model.
Inside for Pair SCFGs	$O(n^3m^3)$	$O(n^2m^2)$		
Conditional Random Fields				
CRF-Forward	$O(nl^2)$	$O(nl)$		n - sequence length.
CRF-Viterbi	$O(nl^2)$	$O(nl)$		l - number of states in model.
Haplotyping Problem				
MSR	$O(mn^2)$	$O(n)$		m - number of fragments.
				n - number of SNP sites.

DP kernels are often employed as *filters*, as illustrated in Figure 1.1. The filter may be configured using one or more *query* parameters, such as a sequence or a probabilistic model. A *database* of discrete inputs, such as sequences, are then streamed through the kernel. The DP computation is applied to each input, and a score is computed; those inputs that score above a threshold are reported for inspection by the user. Filters often discard a majority of inputs in the database stream, which are too low-scoring to cross the threshold of interest. As a consequence, it is more important to be able to quickly compute the score of an optimal solution rather than the sequence of decisions that lead to the optimal solution.

Some filters have no configuration parameters. For example, to identify novel microR-NAs, one simply folds a stream of input sequences. In contrast, pairwise alignment configures the filter with a query sequence and then streams a database of sequences through. Furthermore, the filter may be repeatedly reconfigured with a new query, and the computation repeated on the entire database. Here, the goal is to identify query-database pairs that score above a threshold and may be of interest to the user. In this work, we assume that the query set is always large enough so that the time and space complexity of filters depend on the size of both the query and the database. We will henceforth refer to all filters as *search* applications.

Search applications are useful for quickly identifying a small subset of sequences that can then be studied using time-consuming, more manual analyses. For example, we used NCBI BLAST [7] to compare a virus sequence against the NR nucleotide database containing 14,332,765 sequences. The search completed in tens of seconds and returned under 1934 matching sequences ($< 0.14\%$ of the database) sorted by the statistical significance of each match.

One way of accelerating search applications is to use heuristics to approximate the DP computation. Examples include NCBI BLAST [7] and HMMERHEAD [120] to accelerate pairwise sequence alignment and motif finding respectively. These applications use a variety of computationally inexpensive kernels to discard a majority of inputs before applying expensive DP. The design of heuristics is beyond the scope of this work—we look to accelerate the full DP computation. Nevertheless, applications that use heuristics often employ a filtering pipeline consisting of progressively

more computationally expensive DP kernels, which account for a significant fraction of program execution time.

1.2 A Bottleneck in Computational Biology Analysis
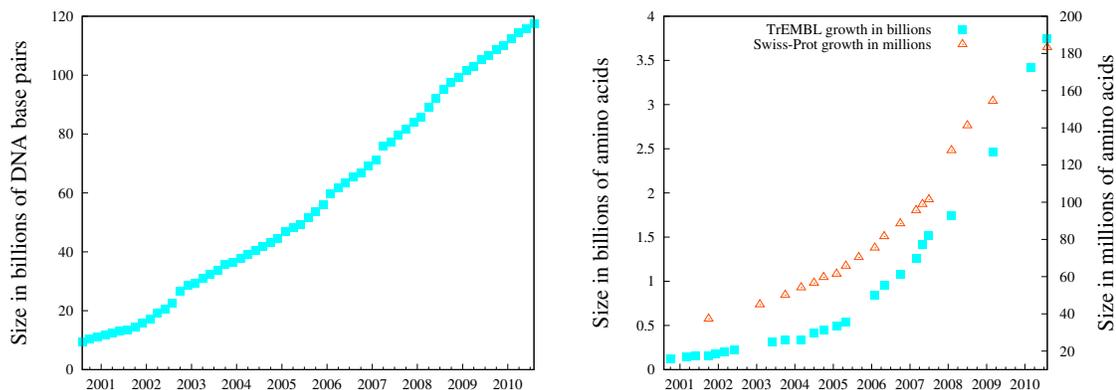
(a) Growth of GenBank [1] DNA database.

(b) Growth of UniProtKB [6] TrEMBL (left y axis) and Swiss-Prot (right y axis) protein databases.

Figure 1.2: The growth of biosequence databases is straining computational analysis pipelines.

An explosion of sequence information over the last decades has led to a computational bottleneck in the DP-based applications analyzing them. DNA and protein databases are growing in size as new organisms and environmental samples are sequenced. Figure 1.2 shows the sustained growth of popular DNA and protein databases over the last decade. Moreover, as listed in Table 1.1, the DP algorithms that operate on this data are compute-bound and frequently run in time quadratic or worse in the input size.

Biologists have relied on advances in integrated circuit technology to keep pace with the increasing computational demands of their work. Over the past four decades the number of transistors per unit area has doubled every two years, a phenomenon referred to as Moore's Law. Microprocessor vendors have in turn been able to achieve a twofold increase in CPU performance every 18 months (we will henceforth refer to the

doubling of CPU performance as Moore's Law). Underwood [151] notes that between 1998 and 2003 clock speed improvement yielded a $12\times$ increase in CPU performance compared to only a $4\times$ improvement due to architectural enhancements. Clock speed increase has been the primary contributor to an essentially "free" performance boost for computational biology applications.

Today, however, the increase in the performance of modern processors has slowed down due to three reasons. First, performance may be improved by pipelining functional units in the processor core. Unfortunately, beyond a certain pipeline depth—already exceeded on modern processors—the pipeline control overhead negates any performance benefit, and circuit design becomes significantly more complex. Second, typical sequential instruction streams have limited usable parallelism, restricting the ability of superscalar processors to fully exploit concurrency. The third and most important reason, though, is that with increasing transistor density and higher clock speeds, cooling the processor without exotic and expensive mechanisms becomes increasingly difficult. There has been a trend away from high clock speeds in recent years, which is forcing scientists to reconsider their reliance on sequential general-purpose processors for high-speed sequence analysis.

To illustrate this problem with a concrete example, we analyzed the results of running the SPEC CPU2006 benchmark [5] on general-purpose workstations. The benchmark suite is designed to measure compute-intensive integer and floating-point performance on a range of publicly available systems to gauge the capabilities of the processor, memory, and compiler. Tests are performed by hardware vendors and reported on the SPEC website for analysis by researchers.

For the purposes of our discussion, we analyzed the performance of one of the programs in the integer benchmarks, HMMER2 [41], a popular profile-HMM bioinformatics search tool. A run of the benchmark compares a hidden Markov model representing a protein motif against the Swiss-Prot protein database version 41, released in February 2003. From other work [35] we know that HMMER2's runtime is dominated by the Viterbi DP algorithm; HMMER2 therefore closely resembles the kinds of applications that are of interest in this work.

The results we have accumulated are from tests performed by vendors on state-of-the-art systems. The performance we report for each system has been normalized to

that of a 296 MHz UltraSPARC II reference CPU. We limit our analysis to single-chip systems, where a single chip is placed in one socket of a motherboard. A chip in turn may have multiple cores, and each core may have one or more hardware threads.
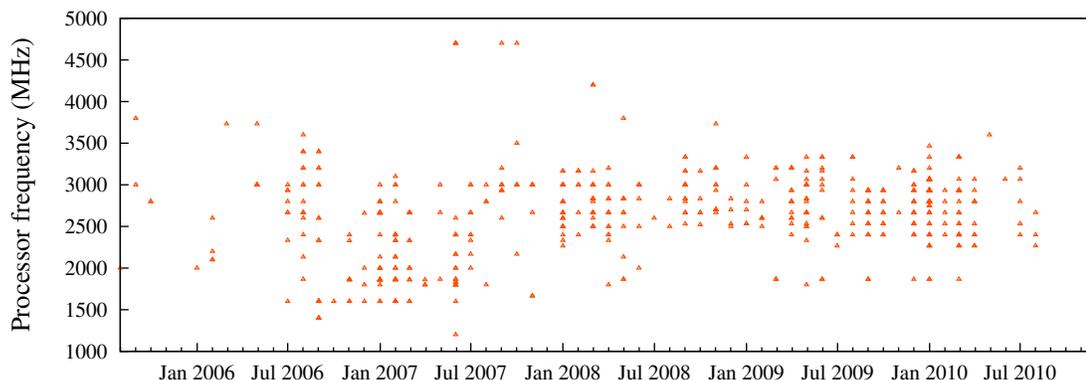
(a) Plot of processor frequency (MHz) of systems running the SPEC CPU2006 benchmark as a function of hardware availability dates.
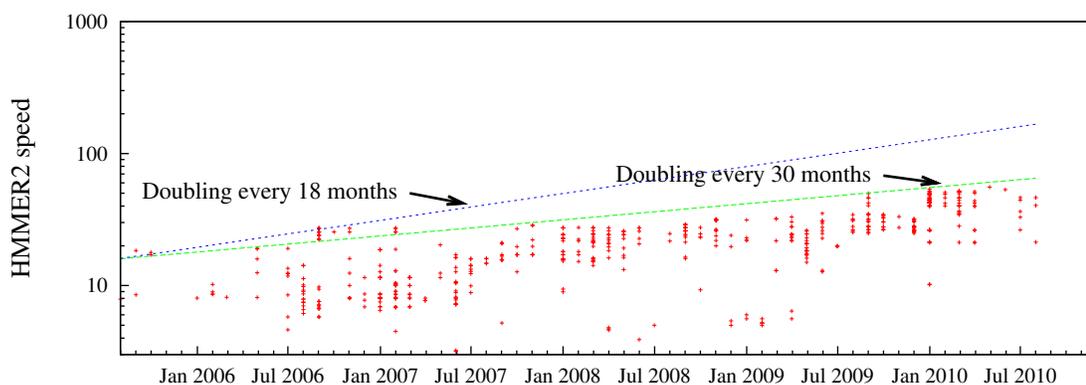
(b) Plot of speed of a comparison of a single query HMM against the Swiss-Prot database, normalized to an UltraSPARC CPU, as a function of hardware availability dates. The y axis is in log scale.

Figure 1.3: The two graphs plot historical trends of processor frequency and HMMER2 speed improvement. Only single chip (socket) systems from SPEC CPU2006 are included; however, each chip may have multiple cores.

Figure 1.3a plots processor frequency in MHz as a function of the hardware availability date. Over the last five years, processor frequency has remained steady, and even dropped marginally, to settle between 2 and 3 GHz. In Figure 1.3b we plot the speed of a HMMER2 comparison of the query HMM against Swiss-Prot version 41. The speed is the inverse of the time to sequentially compare a single query against every protein sequence in the Swiss-Prot database and is a measure of single core performance.

After initially tracking Moore's law until 2006, speed is now only doubling every 30 months. If these trends hold, clearly the time to compare a query to a database on one core will decrease rather slowly in the future.
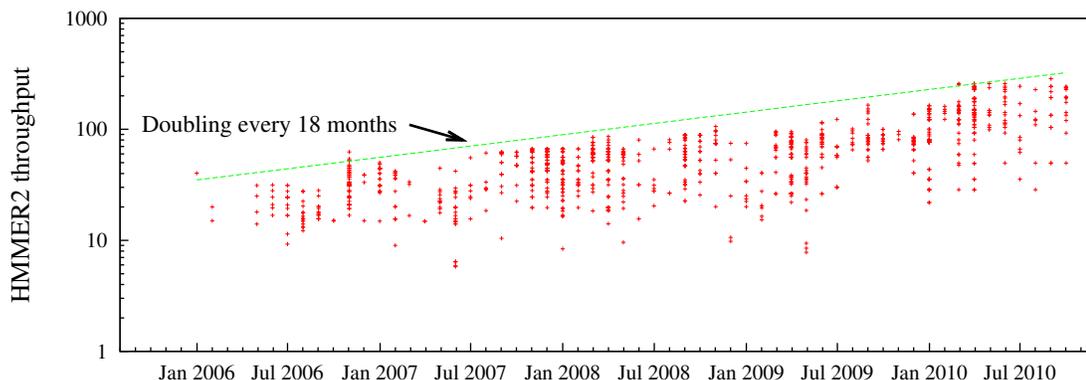
Figure 1.4: HMMER2 throughput measured on various multicore workstations using Swiss-Prot version 41 as the database. This is a measure of performance assuming that the sequence database does not grow in size.

Fortunately, many search applications in computational biology, including HMMER2, are *embarrassingly parallel*; we can easily parallelize across all cores on a single chip by simultaneously executing multiple query-database comparisons, which are completely independent of each other. Thus, in this environment *throughput* (rate) becomes a more important metric than speed. Figure 1.4 plots this throughput metric measured in jobs per second, normalized to the reference machine, as a function of hardware availability dates. As more transistors are packed into the same area, and processor manufacturers place multiple cores in a single chip, HMMER2 throughput is seen doubling every 18 months over the last five years.

This trend at first glance seems heartening—assuming we can continue to double the number of cores per chip every 18 months, and provide sufficient memory bandwidth to each core. However, an important consideration not yet taken into account is the growth of biosequence databases. The HMMER2 speed and throughput measurements in Figures 1.3 and 1.4 assume the Swiss-Prot database size remains fixed—we have already shown this to not be the case in Figure 1.2b. Figure 1.5 shows the throughput of HMMER2 adjusted for growth in the Swiss-Prot database, which is now only doubling every 30 months. Another way to interpret the same data is as

8
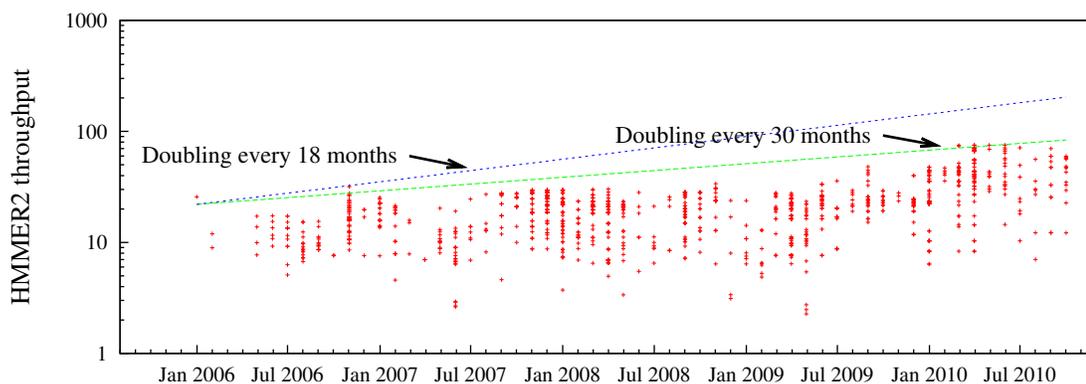
Figure 1.5: HMMER2 throughput adjusted for growth of the Swiss-Prot database. Throughput is now seen to be doubling only every 30 months.

follows. Suppose an institution buys a state-of-the-art multicore workstation (or cluster) in January 2006 that simply satisfies their compute requirements. To keep up with the growth of Swiss-Prot, they would have had to upgrade thrice in the span of five years simply to maintain throughput.
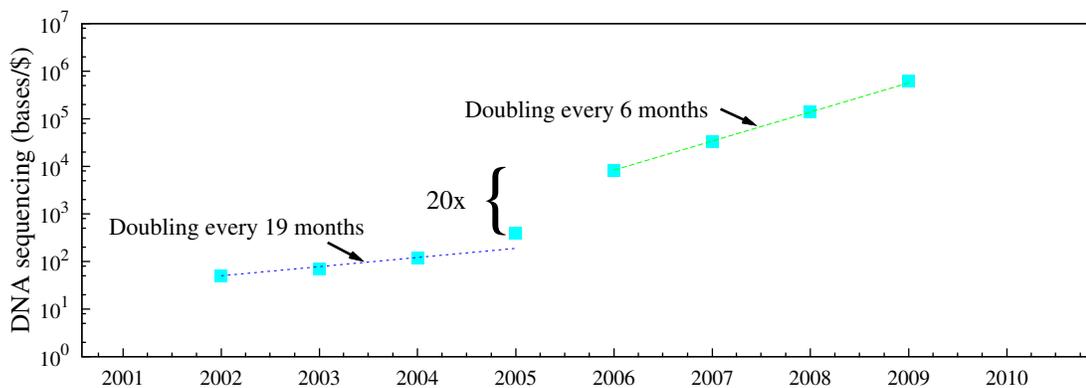
Figure 1.6: The graph plots bases sequenced per unit cost as a function of time. We draw two important conclusions: a) sequencing costs are falling rapidly, while compute performance is only tracking Moore's law, and b) a 20× improvement in bases sequenced per unit cost formed a chasm between the pre- and post-NGS eras that is yet to be bridged by advances in compute performance. Cost for sequencing was made available by Stein [140].

One of the determining factors in the growth of these databases is the cost of DNA sequencing. Figure 1.6 plots this cost over the last decade, expressed as the number

of bases sequenced per dollar. These costs are taken from Stein [140], who collated them from vendor press releases, catalogs, and websites. They are not adjusted for inflation, nor are overhead, depreciation, or personnel costs included.

Before 2005, advances in Sanger sequencing technology, which was used to sequence entire genomes and was primarily responsible for the growth of Swiss-Prot, doubled the number of bases sequenced per dollar every 19 months. In other words, sequencing costs halved in price every 19 months. More strikingly, starting in 2005 there was a marked expansion in DNA sequencing with the arrival of next-generation sequencers (NGS) [134]. NGS allows high-throughput sequencing at a fraction of the cost of traditional Sanger sequencing and enabled applications such as resequencing of genomes and *metagenomics*, which studies the genomes of mixtures of microorganisms in environments such as deep mines, ice cores, sea water, or the human gut. Since the emergence of NGS, the number of bases that can be sequenced per dollar has been doubling every six months as sequencing costs have plummeted (see Figure 1.6). We expect this growth to continue in the near future (2-4 years) as companies push toward the goal of a $1,000$ sequencing of a single human genome.

The drop in sequencing cost is reflected in the amount of DNA sequence available in public collections post-NGS, for example in NCBI's Sequence Read Archive (SRA) [4]. This database stores reads from next-generation sequencers made by companies such as Roche, Illumina, and Applied Biosystems. It contained over 8.5 terabases at the end of 2009 and was growing at the rate of 1 terabase per month[1]. Contrast this with the size of the GenBank database, which was less than 50 gigabases pre-NGS.

Anecdotal evidence suggests that NGS has already made an impact on computational pipelines of molecular biologists. One of the goals of the Human Microbiome Project is to study microbial communities in various sites of adult humans. Samples are taken at six body sites from 16 individuals, generating about 33 gigabases per sample per person. Mitreva [112][2] reports that the greatest challenge is computational—it takes over two months to process data that was sequenced in only 0.3 months!

We acknowledge that not all sequencing projects will result in an equal increase in computational demand. While sequencing new organisms will enlarge databanks,

---

[1]http://www.bio-itworld.com/BioIT_Article.aspx?id=94069

[2]Available online at http://hmpdacc.org/outreach_conf.php

resequencing projects need only store reads with unique variants, which is usually a fraction of the size of the original genome. However, in metagenomics, where entire microbial communities are sequenced, it is likely that only partially sequenced genomic reads will be available, and we expect this to be a major consumer of compute resources.

To understand how these trends may impact future sequencing projects we performed a simple economic analysis. In addition to the DNA bases sequenced per dollar curve in Figure 1.6, we calculated the HMMER2 throughput per dollar growth curve. Unfortunately the total costs of the various systems used to run the SPEC workload are not reported, even as many of the individual components are listed. This makes it a difficult task to establish historical system costs.
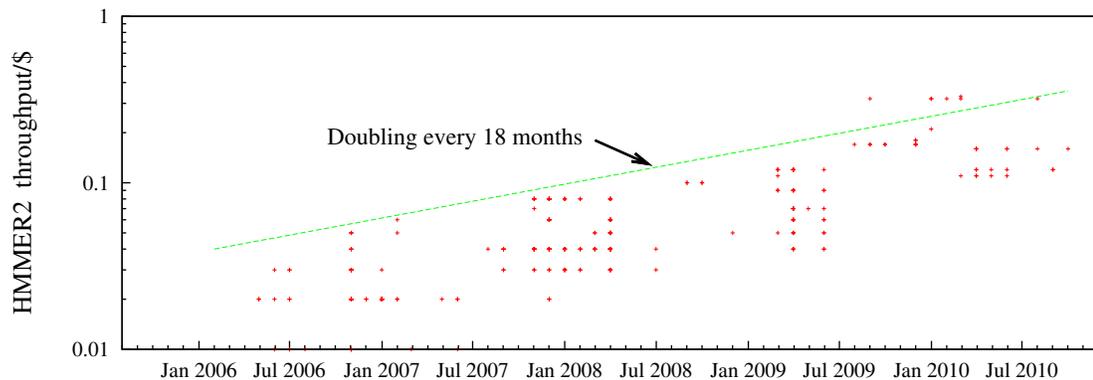
Figure 1.7: Historical growth of HMMER2 throughput per dollar on state-of-the-art general-purpose processors.

As an alternative, we turned to the TPC benchmark[3] to find historical pricing. The mission of the TPC group is to evaluate the performance of database transaction processing on vendor systems. Many of the group's benchmarks measure performance per dollar, and so include a detailed break down of the price of each machine. We found that many of the systems used to run the TPC benchmark were similar to those in the SPEC reports. In this analysis we decided to only include the cost of the processor, which is the price a member of the public can expect to pay to procure the CPU at a certified date. We were able to price sufficient processors in the

---

[3]http://www.tpc.org/

SPEC results in this manner to discover the trend shown in Figure 1.7. HMMER2 throughput per dollar tracks Moore's law and doubles every 18 months.

The two graphs in Figures 1.6 and 1.7 allow us to predict the scale of the computational bottleneck in the near future. *Assuming the trends shown continue, there will have to be an exponential increase in spending on compute resources to handle the increased sequencing workload after an upgrade to simply maintain the runtime before the upgrade.* Indeed, computing expense doubles every 9 months even as sequencing cost remains fixed. Note that this pattern holds even if the compute resource in question is a cluster of workstations or a computing cloud such as Amazon's EC2.

Our analysis in this section has shown that the speed of an individual DP-based comparison is advancing at a sluggish pace, primarily because processor frequency has stagnated. It has become necessary to parallelize search applications on multi-cores to track the Moore's law curve. However, low-cost sequencing is increasing the amount of biosequence data available and is stressing computational pipelines. The advent of NGS in 2005 created an inflection point where sequencing capacity increased by an order of magnitude without a corresponding improvement in computational power. Subsequently, DNA sequencing capabilities are increasing at a far higher pace than compute performance. We assert that as costs for sequencing continue to fall, compute performance, not sequencing, will become the bottleneck in advancing genome science.

1.3 Rise of Non-traditional Computing Architectures

Even before next-generation sequencing (NGS), biologists had turned to clusters of commodity workstations to process their massive databases. In the post-NGS era it is imperative that we investigate novel compute platforms that can keep up with low-cost sequencing. In this section we evaluate clusters along with alternate computing platforms, and advance the case for building customized hardware accelerators.

Clusters. Workstation clusters make use of the embarrassingly parallel nature of search applications to achieve acceleration with good scalability and relatively limited development effort. As one example, NCBI reported using a Linux cluster for running BLAST as early as 2002 [110]. Parallel versions of popular tools include mpi-BLAST [51], a grid implementation of NCBI BLAST, and mpiHMMER [157], used to accelerate protein motif search on a cluster.

Cluster implementations, however, have challenges and important limitations. Lin et al. [98] state that a key challenge for massively parallel sequence analysis tools on clusters is in handling their irregular computation and I/O patterns. Splitting workload among individual nodes in a cluster when performing large genome-to-genome scale comparisons is not completely straightforward. One approach is to segment a query set into subsets that are each compared to the entire database on individual nodes. However, this approach does not scale well if the database cannot fit in local memory. Conversely, segmentation partitions the database into chunks that can fit in local memory; however, it introduces communication overhead when results must be merged from disparate compute nodes. Moreover, it is often difficult to predict the runtime of search tools based on their sequence sizes alone, particularly for those that employ heuristics. Dynamic load balancing schemes must be employed to achieve scalability on large clusters.

In addition to implementation challenges, clusters have many practical limitations: they have a significant initial investment, are expensive to maintain, occupy large amounts of floor space, and need effective powering and cooling solutions. Nevertheless, clusters are often preferred due to their relative ease of use compared to other parallel systems.

Cloud Computing. This is a computation-for-hire paradigm, touted as an alternative to clusters that could potentially ameliorate some of their limitations. Amazon's Elastic Computing Cloud (EC2) is one such platform where biologists can rent compute resources as required. Consequently, upfront capital expenditure, maintenance costs, and floor space are no longer an issue, and users will be able to take advantage of economies of scale. Renting resources is particularly attractive when demand fluctuates, e.g., if peak performance is required only after a new release of a

sequence database. The alternative of maintaining an idle cluster that has been over-provisioned 2–10× of its average load can be expensive. However, a significant issue with cloud computing is how to handle security for private databases [140]. Institutional users may be unwilling to transfer sequence data outside their organization, both to protect their intellectual property and due to legal concerns.

Notwithstanding the flexibility of the cloud, we wanted to investigate if Amazon's EC2 is economical for sustained compute-intensive workloads. We suggest that this is not the case by using an analytical model from the literature to compare the performance per dollar of Amazon's EC2 to that of a cluster.

A report from U.C. Berkeley attempts to compare the economics of using Amazon's EC2 service versus a user-owned cluster [9]. Assuming a CPU-bound application, which is reasonable for the domain under consideration, the authors calculate that in 2008, $1 bought 128 hours on a user-owned 2 GHz CPU. The cost to rent the same compute resource on EC2 was $2.56. Even assuming that power, cooling, and amortized cost of floor space doubles this cost of computation on a cluster [9], Amazon's cloud computing service was still 28% more expensive.

To illustrate the price difference with a concrete example, we refer to a recent study by Wall et al. [156] that documents the migration of a comparative genomics algorithm, reciprocal smallest distance, to the cloud. The application executes NCBI BLAST and ortholog estimation, with the former taking 58% of total runtime in their experiments. The authors rented 136 hours on 50 High-CPU Extra Large instances (approximately ten 2 GHz CPUs each) on Amazon's EC2, for a total cost of $6,256. This price includes the cost of the compute resource and the cost of using the Elastic MapReduce framework, but does not include cost of storage or data transfer into or out of the cloud. Assuming a user cluster of the sort discussed in the prior paragraph that has an amortized cost of $2 for 128 hours on a 2 GHz CPU, renting was over 5× more expensive.

Nonetheless, companies such as DnaNexus[4] and GenomeQuest[5] hope to leverage the flexibility and low investment advantages of cloud computing to enable analysis of NGS data loads.

---

[4]http://www.dnanexus.com/
[5]http://www.genomequest.com/

Specialized Architectures. Special-purpose hardware is becoming increasingly popular to accelerate algorithms in domains such as computational biology, signal processing, and computational science. Their specialized architectural features enable exploitation of parallelism available in algorithms to realize several fold speedup over general-purpose processors. Accelerators for applications in computational biology have been built on network processors [162], graphics processors (GPUs) [71], the IBM cell [48, 101], and field programmable gate arrays (FPGAs) [35, 66, 95].

FPGAs are an attractive target architecture because they are a relatively cost-effective medium to deploy customized accelerators and can exploit the fine-grained parallelism available in many DP algorithms. We have previously used FPGAs to accelerate sequence analysis algorithms such as Smith-Waterman for local similarity search ($100\times$) [65], NCBI BLAST ($30\times$) both for DNA [86] and protein [80] comparison, and HMMER ($190\times$) [79]. El Ghazawi et al. [46] compare the performance of DNA Smith-Waterman alignment on an SRC-6 system comprising four FPGAs and an Opteron processor. They report that the accelerator is $1138\times$ faster, i.e., it is equivalent in performance to a 1138-node Opteron cluster. Compared to this cluster, the FPGA solution is $6\times$ cheaper, occupies $34\times$ less floor space, and consumes $313\times$ less power.

GPUs are inexpensive, commodity co-processors that exploit the single instruction, multiple data stream (SIMD) paradigm. Although primarily intended for accelerating graphics rendering, they are being successfully applied to general purpose computing. A GPU's main advantages are its high memory bandwidth and a large number of parallel processors that efficiently implement floating point arithmetic. GPUs also run at higher clock speeds than FPGAs; for example, benchmark applications in a recent study [30] are clocked up to $6.5\times$ faster on a GPU than on an FPGA (for comparable 90 nm technology devices).

It is instructive to study the advantages and disadvantages of FPGAs and GPUs. Our analysis uses two recent comparative studies, one by Che et al. [29], and the other by Cope et al. [30]. The major advantage of an FPGA is that we can build an accelerator that has been customized to an application. A GPU on the other hand exploits SIMD parallelism, but performance is adversely affected in the presence of irregular computations such as conditional branching. An advantage of GPUs is the availability of high-bandwidth memories, but modern FPGAs provide a large number

of distributed on-chip memory blocks that can be better utilized to accelerate DP kernels. We believe that the advantages of modern FPGA devices make them more suitable for implementing DP accelerators.

The primary motivation for using an FPGA is to build circuits customized for the specific datapath of a DP kernel. In effect, we can build an application-specific instruction set, whereas on a GPU, the kernel must be mapped onto a fixed instruction set architecture. Because we can customize the hardware, FPGAs are able to better exploit parallelism and pipelining than GPUs. The physical architecture of FPGAs is well suited for DP accelerators, which have a regular structure with a near-neighbor communication network. While floating-point operations on FPGAs are expensive, fixed-point arithmetic, which is most common in DP kernels, can be implemented efficiently; moreover, bitwidths can be customized according to the desired precision to improve resource utilization and application performance.

Che states that a GPU is ideal for exploiting algorithms that do not have a large number of data dependencies; unfortunately, we expect many DP kernels to include such dependencies. On an FPGA we have more freedom to explore various parallelization strategies to hide the effect of dependencies. Another major disadvantage of the SIMD paradigm of GPUs is its inability to efficiently implement conditional branching. Such control code can be found in DP kernels, which serializes computation in a SIMD execution unit. In contrast, on an FPGA we can expend redundant computational units to handle branches more efficiently.

A major benefit of GPUs is their access to an off-chip high-bandwidth, high-capacity memory store. To use this resource efficiently, computation must dominate in order to hide the high latency for off-chip memory access. FPGAs, on the other hand, make available a large number of distributed on-chip memory blocks that in aggregate provide higher bandwidth than the low-latency memories in GPUs (for example, texture memory). Cope quotes a peak bandwidth on a Virtex-4 XC4VSX25 FPGA of 288 GB/sec, five times higher than a GeForce 7900 GTX (both 90 nm technology). Moreover, the distributed memory blocks can be integrated into the datapath of the application unlike on the GPU. In cases where off-chip memory access is required, FPGAs can more effectively exploit data reuse to mitigate the effect of low bandwidth.

A comparison by Che of the Needleman-Wunsch DP kernel shows that an FPGA outperforms a GPU for pairwise sequence comparisons of lengths of up to 2048 characters each. This is true despite the fact that their hardware implementation uses floating point operations and does not use the entire FPGA die area.

All of the factors we have highlighted suggest that FPGAs will outperform GPUs from a purely throughput perspective when accelerating DP kernels. Because GPUs are commodity devices capital expenditure is significantly less than for FPGA systems. However, FPGAs typically have one to two orders of magnitude higher energy efficiency [84, 145]. This is an important factor when considering the lifetime cost of a large cluster of compute nodes, which is essential for high-throughput bioinformatics.

1.4 Challenges of Development in Hardware Languages

While the performance boost of FPGA accelerators over a general-purpose CPU is significant, a major impediment to their widespread use is development cost. Custom circuits on FPGAs are typically described using hardware description languages (HDLs), which require a paradigm shift from sequential languages. HDLs are used to program at the logic gate level and provide little abstraction. Another frustrating aspect is code reusability. Often a hardware module is tailored to one algorithm, and the implementation is dependent on a specific set of parameters. For example, though the underlying computations in a Smith-Waterman and a HMMER accelerator are comparable, their hardware implementations share little similarity. HDL designs are difficult and time-consuming to develop, are hard to debug, and scale poorly to large designs.

In response, researchers have developed high-level hardware languages that aim to provide a conventional software-like programming language for parallel hardware. Ideally, a supporting compiler should automatically extract parallelism from the high-level language and generate hardware arrays with performance close to that of hand-optimized HDL designs. Examples of such languages include Streams-C [55], Impulse-C, ASC [111], Sea Cucumber [148], and Trident [147, 149]. Recent comparisons

17

of a high-level language to an HDL show a reduction in development time. For example, researchers [45, 163] demonstrate an up to 2.5× reduction in development time using Impulse-C, but performance is only about half that of manually optimized HDL designs.

While high-level hardware languages abstract the underlying architecture with varying degrees of success, we raise a new issue that is just as important: existing high-level languages provide no freedom for the compiler to explore the large design space of arrays optimized for throughput, latency, or area. This is important for a DP kernel, which may have a large number of non-obvious parallel arrays in the accelerator design space.

```
                                              for (j = 1; j <= M+N-1; j++) {
                                                i = max {1, j-M+1};
    for (i = 1; i <= M; i++) {                  for (d = min {j, M}; d >= 1 && i <= M; d--) {
      for (j = 1; j <= N; j++) {                  X[i][d] = min { ... };
        X[i][j] = min { ... };                    i++;
      }                                         }
    }                                         }
```

(a) Standard sequential implementation.　　　　(b) Antidiagonal parallel implementation.
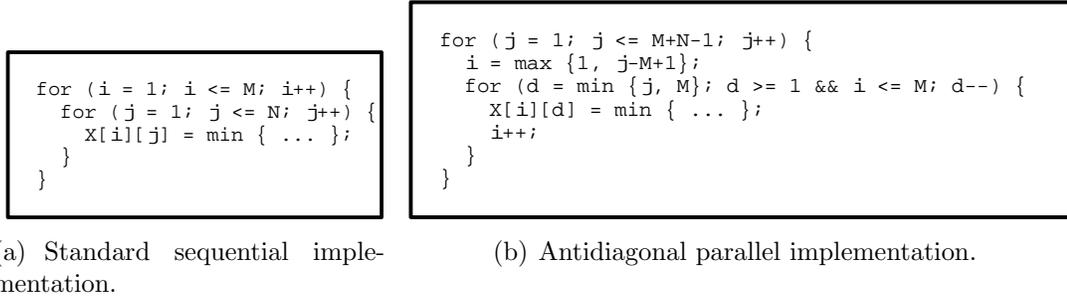
Figure 1.8: Two implementations of the Smith-Waterman algorithm in a C-like high-level hardware description language.

As an example, Figure 1.8 shows two implementations of the Smith-Waterman DP recurrence in a C-like high-level language. They perform identical subproblem computations but in different order: the first is a typical implementation used on sequential processors, while the second has parallelism in the inner loop. Most existing hardware compilers do not generate the parallel implementation given the sequential one, or do so only for limited cases. The onus is on the user to carefully select the appropriate subproblem ordering. Moreover, exploring various parallelization strategies to generate an array that is guaranteed to have optimal throughput, latency, or area is beyond the capabilities of most high-level compilers.

A central goal of this thesis is to use an automatic procedure to explore the design space of possible subproblem orderings, with a guarantee on optimal performance. The techniques we will propose and demonstrate in this dissertation are a first step toward a compiler that can automatically explore and synthesize accelerators from

high-level representations of DP kernels and thus significantly reduce the costs associated with FPGA implementations.

1.5   A New Approach to an Old Problem

In this work, we represent DP kernels as systems of recurrence equations [83] and use polyhedral theory [96] to generate application-specific *systolic arrays* [90]. A systolic array is a regular, synchronized grid of locally-connected processors well-suited to exploit the fine-grained parallelism available in DP. The polyhedral model is a powerful theoretical framework that can represent and analyze regular loop programs (or DP) with static dependencies and can generate efficient systolic arrays. Loop transformations such as interchange, skewing, unrolling, and tiling can be investigated systematically to expose far more parallelism than *ad hoc* methods. Our aim is to use and extend these techniques to explore the design space of possible systolic arrays optimized for throughput or latency subject to resource constraints.

Before we proceed further, it is worthwhile questioning why we see promise in application-specific systolic arrays. Since their introduction in the 1980s to build efficient, verifiable VLSI designs, systolic arrays have had a long history and yet were replaced by general-purpose superscalar processors with principles quite contrary to the systolic philosophy of simplicity, locality, and regularity. Are we revisiting ideas that have been previously tested and found wanting?

We believe that a number of recent trends merit reconsideration of the systolic philosophy in a new light [96].

1. General-purpose computing was so successful largely due to an ability to consistently boost the clock frequency of microprocessors. As has been noted, however, this trend has ceased, and focus has shifted to parallel programming.

2. High-level synthesis of recurrence equations offers an excellent abstraction and a well-studied formal framework to automatically exploit parallelism in the kinds of applications we are interested in.

3. FPGAs have increased in popularity as a relatively cheap method to prototype and deploy hardware circuits. The underlying fabric of FPGAs rewards simple, locally connected, and regular circuits—an ideal match for systolic arrays. Advances in FPGA technology, such as increased logic and memory resources and low-latency high-bandwidth coupling with CPUs are a very recent phenomenon.

4. Most importantly, we have in computational biology a domain with abundant algorithms that can be efficiently parallelized. These DP kernels comprise a significant fraction of execution time of popular applications and are performance-limited by rapidly expanding datasets.

1.6 Dissertation Overview

The overarching question we ask in this dissertation is:

How do we best exploit parallelism in dynamic programming kernels, and how can we build efficient hardware accelerators that are guaranteed to be throughput-, latency-, or area-optimal?

Currently, researchers choose from a number of standard accelerator configurations using ad-hoc design selection criteria. Choices include cached shared versus distributed memory designs, and instruction-based execution units versus processors with a fixed datapath. Often, there is no way to determine if the selected design with its parameter choices has optimal latency, throughput, or area usage. Our goal is to assemble a formal framework to analyze and exploit parallelism in DP kernels. Accordingly, we ask, what are good high level abstractions for representing DP, and how do we synthesize efficient accelerators while optimizing performance and resource usage?

Rather than target *every* parallel program (or conversely, just one specific application) as is often done, we choose to focus on a *class of algorithms*, namely dynamic programming, which is specific in its scope, yet encompasses a large set of problems, even outside the confines of computational biology. Indeed, researchers from U.C.

Berkeley [10] distinguish DP as one of 13 important classes of problems for paralleliza-
tion. By narrowing our focus to DP, we are able to apply specialized parallelization
techniques that would be less beneficial for general programs.

1.6.1 Brief Background

To help illuminate our research objectives, we use the Nussinov RNA folding DP al-
gorithm [117], which is used to find an optimal structural folding of an RNA molecule
S. Nussinov defines a two-dimensional variable $X(i, j)$ that holds the score of the op-
timal folded structure formed by the subsequence $S_{i...j}$; hence, this optimal structure
for the entire RNA of length N is at $X(1, N)$. Chapter 4 introduces RNA folding in
more detail and the reasons for accelerating it.

The Nussinov DP algorithm can be succinctly described by the recurrence

$$X(i, j) = \max \begin{cases} X(i+1, j) \\ X(i, j-1) \\ X(i+1, j-1) + \delta(S_i, S_j) \\ \max_{i < q < j} [X(i, q) + X(q+1, j)]. \end{cases} \tag{1.1}$$

Note how the score $X(i, j)$ of the optimal folding of subsequence $S_{i...j}$ is computed
using four cases that cover all possible ways of folding the subsequence. These are
in turn computed using smaller subproblems $X(i+1, j)$, $X(i, j-1)$, $X(i+1, j-1)$,
$X(i, q)$, and $X(q+1, j)$. If we build from smaller to larger subproblems and cache
solutions in a table, DP can solve the problem in $O(N^3)$.

Building from smaller to larger subproblems in DP is analogous to the concept of
dependencies in a recurrence. Consider the score $X(i+1, j)$ of the optimal folding of
subsequence $S_{i+1...j}$. Because we need the score at $X(i+1, j)$ before we can compute
$X(i, j)$, we say the latter *depends* on the former. This dependency can be represented
by a constant vector $[1, 0]$, which is computed by subtracting the two sets of indices.
Note that this dependency holds when computing the optimal folding of any subse-
quence of the RNA. Such constant vector dependencies are termed *uniform*. Other

uniform dependencies in the algorithm are $[0, -1]$ and $[1, -1]$. Uniform dependencies are preferred for efficient systolic array synthesis because they translate to data movement between adjacent processors.

In contrast, the fourth term in the recurrence gives rise to *affine* dependencies, for example, $X(i, j)$ depends on cell $X(q + 1, j)$ such that the difference $q + 1 - i$ is not constant. In this case the length of the dependencies vary according to the subsequence of the RNA being considered. Affine dependencies translate to inefficient arrays in hardware. Fortunately, many affine dependencies can be made uniform by a process called localization [123].

1.6.2 Summary of our Approach

We systematize the acceleration of dynamic programming by following the workflow illustrated in Figure 1.9. While we have focused on FPGAs in this work, many problems we pose and their solutions are also applicable to ASICs, and partially, to other implementation targets.

We start by representing any DP algorithm as a recurrence and apply existing localization techniques to transform it to a purely uniform recurrence. Note that a larger class of loop programs can also be represented as uniform recurrences and accelerated with the techniques we develop. However for the purposes of this dissertation we will focus only on dynamic programming.

In the course of building special-purpose accelerators for a uniform recurrence, we find one or more *array mappings*. An array mapping fully describes the properties of a systolic array to be built, including the processor configuration, the interconnection network, and the synchronized movement of data.

We consider two categories of array mappings. An *unconstrained mapping* assumes a large enough FPGA target is available and builds an array that exploits all available parallelism. In contrast, *resource-constrained* mappings restrict parallelism in exchange for using fewer processors. Such mappings are useful if the target device has limited logic resources, or if the DP algorithm is resource-intensive.
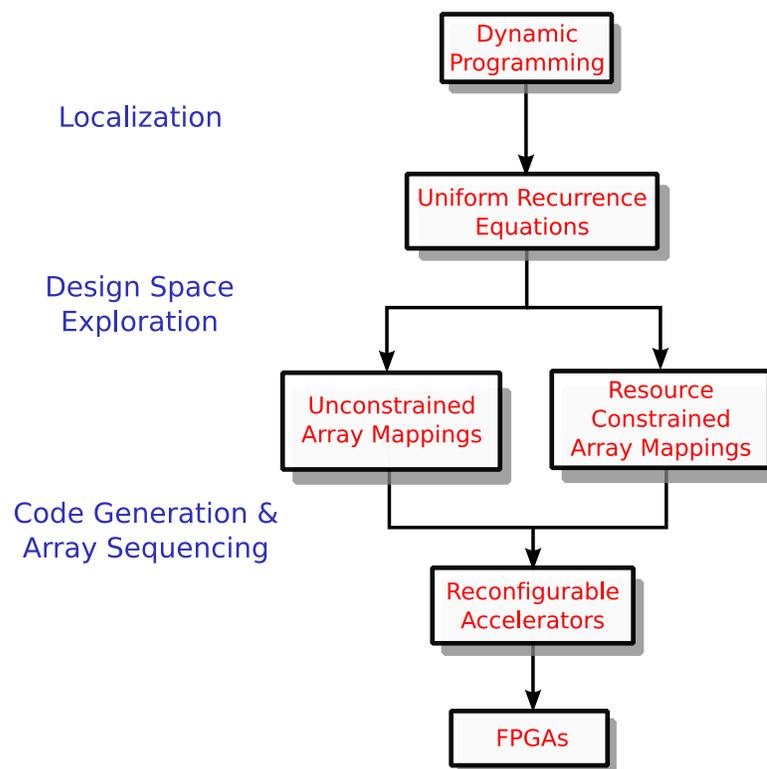
Figure 1.9: Overview of the research conducted in this dissertation, which transforms dynamic programming algorithms into special-purpose accelerators.

A natural question that arises is *"what constitutes a good array mapping?"* We could seek mappings that are optimized for latency. The state of the art is to identify a single latency-space optimal array. Such an array computes a single instance of the recurrence with the lowest latency, or in our example, folds a single RNA in the shortest time possible. Among all such arrays, the one that requires the fewest processors is then selected. Latency-optimal arrays are useful in cases where a DP computation must be accelerated for realtime applications.

Throughput, however, is often the most important performance metric for search applications in computational biology. In our example, the time to fold any single RNA is less important than the time to fold an entire database of them. We propose to build arrays optimized for throughput, even at the expense of latency, by pipelining the computation of multiple input instances.

Next, we show that *"good"* array mappings are not necessarily intuitive and that finding them is not straightforward. The space of possible mappings is large, and we need efficient algorithms to search and evaluate the design space. We propose an enumerative procedure to find optimal high throughput mappings for any input recurrence. Finally, we suggest the use of multiple throughput-optimized accelerators for a single recurrence. Our search algorithm suggests arrays that can tradeoff area on the FPGA for higher throughput.

We then turn to the problem of resource-constrained mappings and apply an existing technique to discover such arrays for dynamic programming recurrences. Once array mappings are identified for a DP recurrence, we can synthesize them on the target FPGA. While we do not tackle the problem of code generation in this dissertation, we give a number of general techniques for efficient implementation of our arrays on FPGA fabrics.

Figure 1.10 describes the optimization problem for DP recurrences. Given a DP kernel we may build throughput-, latency-space-, or area-optimal arrays depending on the implementation constraints. Throughput-optimized arrays, which we introduce in this work, exploit maximum parallelism but also use the most logic and memory resources. Resource-constrained arrays severely limit area usage in exchange for exploiting limited parallelism; latency-space optimal arrays, on the other hand, are a compromise between the two. Note that while by definition there is only one latency-space optimal array of interest, there may be several useful throughput-optimized and resource-constrained arrays that tradeoff area for increased parallelism. The figure shows examples of the three types of arrays we will derive in subsequent chapters for the three-dimensional Nussinov RNA folding algorithm (after localization).

Modern FPGA platforms allow the programming and re-programming of hardware circuits in hundreds of milliseconds through a process called reconfiguration. We investigate its use to further speed up search applications by quickly switching between a library of array mappings for a recurrence based on input size. The goal is to use high throughput, resource-intensive mappings for inputs (e.g. RNAs) of small size while switching to lower-throughput mappings when area becomes an issue. We give an algorithm to select appropriate mappings and their sequence of execution given this objective and demonstrate the effectiveness of reconfiguration.
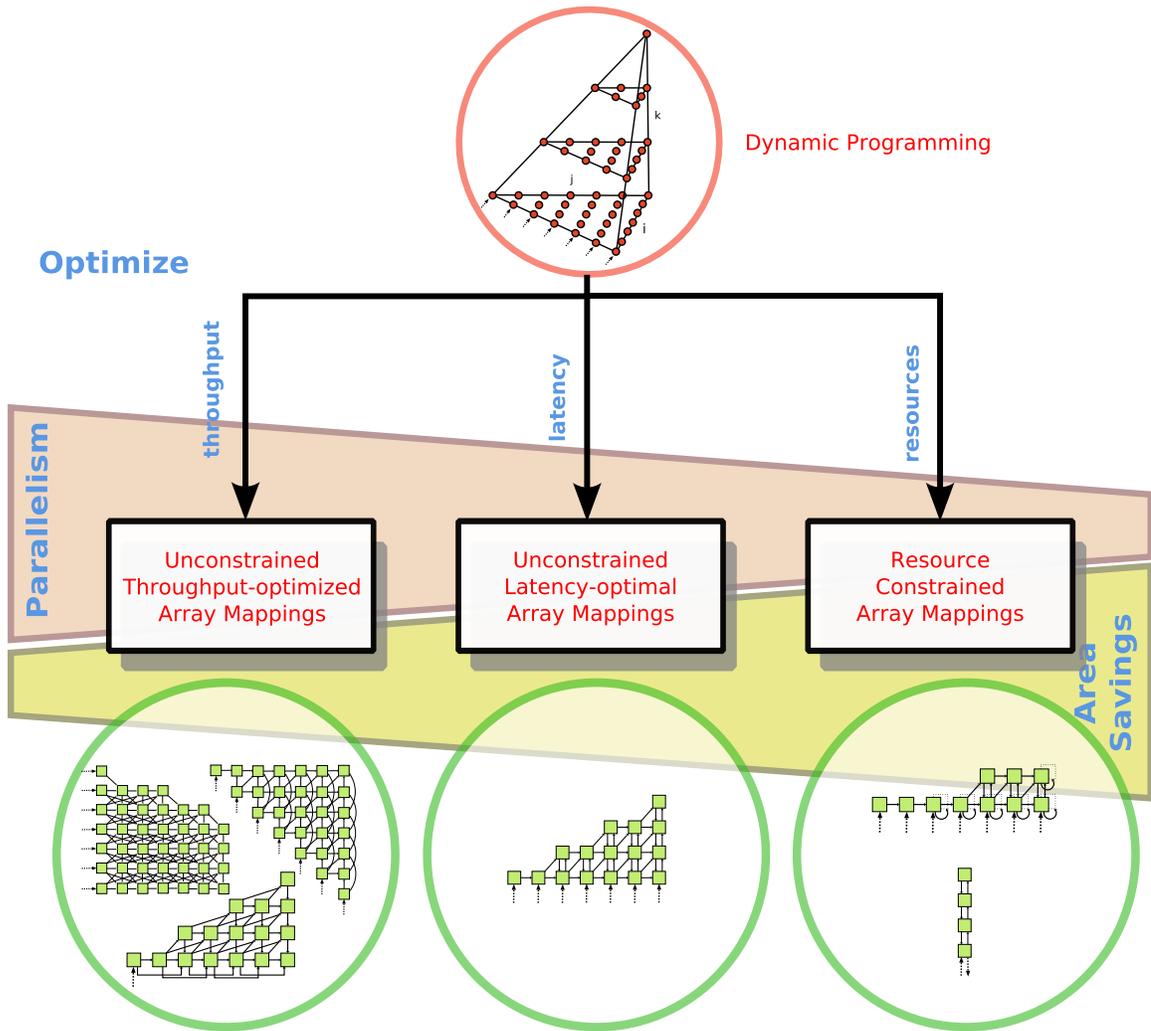
Figure 1.10: Three performance criteria that may be optimized when building customized hardware accelerators for a DP kernel. Selecting one of the three branches affects the amount of parallelism exploited and the resources consumed.

Finally, we apply these techniques to accelerate the Nussinov and Zuker RNA folding algorithms. We implement and test a large number of newly discovered array mappings and experimentally verify a one-to-two order of magnitude speedup over software implementations. We show that these novel arrays outperform even competing accelerator platforms.

1.6.3   Research Questions

The specific research questions we address in this dissertation are as follows:

1. Thus far the state of the art has been to build a single latency-space optimal array using polyhedral theory for recurrence equations. This has also tended to be the practice when building DP accelerators in computational biology. However, we ask:

   Can we exploit pipeline parallelism to build throughput-optimized arrays? Can we model throughput analytically and design an efficient, automated search procedure to identify throughput-optimized arrays?

2. In studying the problem of resource-constrained arrays, we apply existing procedures to accelerate DP. We investigate:

   How well do these techniques apply to finding high-performance array mappings for DP? How do we implement these arrays to effectively use resources on modern FPGA fabrics?

   We also compare the systolic philosophy, in particular the use of distributed memory and pipelined data flow, to processor architectures with shared caches. The choice of architecture determines how data flows to processors in the array and the mechanism for synchronization. We would like to identify the advantages and disadvantages of each and ascertain the right architecture for our purpose.

   How well do systolic arrays perform compared to shared-memory architectures?

3. We know that many search applications operate on inputs of varying sizes, be it biosequences, profile hidden Markov models, or stochastic context-free grammars. We are able to design a library of accelerators that are optimal for distinct ranges of input sizes.

> *Can we use FPGA reconfiguration to use this library of accelerators to accelerate DP computation of an entire database of inputs?*

4. Finally, we would like to demonstrate these techniques on DP algorithms in computational biology.

> *Can we improve the performance of the Nussinov and Zuker RNA folding algorithms on FPGAs using the techniques proposed in this dissertation? How do our FPGA accelerators compare to multi-core and GPU implementations?*

1.6.4 Contributions

A detailed list of our contributions follows.

1. We extend polyhedral theory to build throughput-optimized systolic arrays for any recurrence equation [76], and our technique is able to exploit more parallelism than current methods.

   - We are able to exploit pipeline parallelism on systolic arrays by simultaneously operating on multiple inputs. We demonstrate that these arrays process a database of inputs faster than a latency-optimal array.

   - We develop a novel integer-linear-programming formulation to analytically model throughput of systolic arrays.

   - We prove that the throughput is dependent on the processor configuration and is independent of the timing of data flow. This surprising but important discovery simplifies design space exploration and allows us to search the two components of an array mapping independently.

- We give an efficient, automatic search procedure to find throughput-optimized array mappings. This enumerative procedure optimizes throughput and resource usage using system input bandwidth and resource bounds.

- We further boost performance of arrays by improving the clock frequency of these designs. We do so by pipelining iterations on individual processors using the retiming circuit optimization. We develop an integer linear program to select appropriate array mappings according to the desired number of pipeline stages in a processor. This allows hardware synthesis tools to automatically pipeline processors without any additional programmer effort.

- Our design procedure has implications on I/O properties. Because we demonstrate that latency can be sacrificed as long as throughput is optimized, we are able to design arrays with lower instantaneous I/O.

- We have written a computer program intended to be a plugin to a compiler backend that accepts descriptions of any input recurrence and automatically suggests throughput-optimized arrays using the ideas we have developed.

Our emphasis on throughput is relevant for a wider domain of problems beyond dynamic programming and computational biology. We believe optimizing for throughput rather than latency is more appropriate for domains such as audio, video, and signal processing.

2. We analyze the Nussinov RNA folding algorithm and apply transformations such as localization to make it more amenable to parallelization [74]. We then design and implement novel systolic arrays accelerating RNA folding on a Xilinx Virtex 4 FPGA.

   - We design two latency optimal arrays, FS-A and FS-B, that are similar to those previously proposed for string parenthesization.

   - We increase performance of array FS-A four-fold by pipelining multiple inputs on the array as suggested by our previous contribution. The same technique can be used to improve performance of the decades-old array [60] for string parenthesization.

- We propose a novel throughput-optimal array, FS-C, that is $2\times$ faster than the latency-space optimal array while using the same number of processors. This array also has better instantaneous I/O properties than the latency-space optimal array.

- Our software tool also suggests a number of other arrays for Nussinov RNA folding that trade off area for increased throughput. We implement and test one such array.

- We apply the technique to automatically pipeline iterations on processors by building a parametrized implementation of array FS-C. By varying the number of pipeline stages, the synthesis tools is able to clock the design $2\times$ faster than the unpipelined design with only a 37% increase in area usage.

3. We apply resource-constrained array mapping to design and implement a number of arrays for Nussinov that can be used to fold long sequences. We also suggest and verify the efficiency of a number of implementation techniques to improve clock frequency and area usage of systolic arrays on modern FPGAs.

   - We propose an unpartitioned 1-D array for Nussinov that can be used for RNAs of length $N \leq 545$ bases with execution time $\Theta(N^2)$.

   - We propose a partitioned 1-D array for Nussinov that can be used for RNAs of length $N \leq 597$ bases with execution time $\Theta(N^2W)$.

   - We propose a partitioned 2-D array for Nussinov that can be used for RNAs of length $N \leq 305$ bases with execution time $\Theta(NW)$.

   Here W is the partition width, which controls how much parallelism is sacrificed to save area. By modifying this parameter, we are able to design a family of arrays suitable for various sequence lengths.

4. We demonstrate FPGA reconfiguration on our platform and the performance boost it brings to a DP application [75].

   - Given a family of parametrized designs, we suggest a novel dynamic programming algorithm to select an optimal set of arrays, their sizes, and the sequence of configuration on an FPGA. A second algorithm is also proposed to select an optimal set of at most k designs. This is useful

in situations where the circuit configurations must be stored in limited on-board memory.

- We implement these algorithms in a computer program to operate on any family of accelerators and input distribution.

- We test these ideas by folding a database of pyrosequencing reads of various lengths using the Nussinov algorithm and reconfiguring among multiple arrays.

5. We then turn our attention to the Zuker RNA folding algorithm. This algorithm is more challenging to accelerate because it consists of a larger number of more complex dependencies, and uses resource-intensive tables in the DP computation. We are able to apply a number of novel transformations to reduce its time complexity and simplify the dependencies, making it more amenable to parallelization [77].

- We build a highly efficient 1-D array for Zuker that is able to fold RNAs of length $N \leq 273$ bases in $\Theta(N^2)$ time.

- We qualitatively compare our architecture to shared memory implementations of the algorithm on FPGAs and show better scalability. We quantify the performance of our array and show that it outperforms other accelerators on FPGAs, GPUs, and multicores.

The software tools and hardware arrays developed during the course of this research are made available at `http://www.cse.wustl.edu/~jarpith/`.

1.6.5 Outline

We now introduce the various chapters of the dissertation. We have organized our exposition to build up our work according to the workflow in Figure 1.9.

We gather related work in Chapter 2 that is relevant to our high-level goal. We first describe high-level code generation approaches for DP-based computational biology applications. We also give a brief overview of a number of research compilers for

synthesizing hardware from recurrence equations into which the software tools we have developed may be integrated.

Chapter 3 introduces systolic arrays, recurrence equations, and the basics of polyhedral theory. We describe the localization transformation and summarize state-of-the-art methods to build latency-space optimal arrays.

We parallelize the Nussinov RNA folding algorithm in Chapter 4 by applying the general techniques introduced in the previous chapter. We go through the various steps of localization and derive two latency optimal arrays.

In Chapter 5, we answer the question of how to exploit pipelining to improve throughput of systolic arrays. We show how to pipeline computation on independent inputs in the array as well as pipeline iterations in each processor.

We use existing techniques to build resource-constrained array mappings for the Nussinov DP kernel in Chapter 6. We suggest implementation techniques to improve clock frequency and area usage on modern FPGA fabrics and experimentally verify results on our platform.

Chapter 7 introduces the potential of FPGA reconfiguration to speed up search applications. We use the various arrays developed for the Nussinov computation in the previous chapters to demonstrate the capabilities of reconfiguration.

We analyze and parallelize the Zuker RNA folding algorithm in Chapter 8. We design and implement a 1-D array and compare its performance to related work.

Finally, in Chapter 9 we discuss the conclusions we have arrived at after answering the research questions raised in this dissertation. We discuss directions for exploration in the future and speculate on how our work may help designers of large compute systems for computational biology and other workloads.

Chapter 2

Related Work

In this chapter we survey high-level code generation tools relevant to accelerating dynamic programming in computational biology. Related work that is pertinent to individual aims in our dissertation is included in the corresponding chapters; for example, Zuker accelerators from the literature are surveyed in Chapter 8.

The purpose of this review is three-fold. First, it serves to show the interest of the computational biology community in source-to-source transformation tools that ease the development of both sequential and parallel DP-like programs. In particular, we survey several tools that are specific to dynamic programming. High-level tools for computational biology have an important role in providing abstractions suitable for the domain expert. For example, they may provide user-friendly constructs for pairwise sequence alignment or for representing Hidden Markov Models. We will see that these tools use DP to implement their search algorithms but are limited in their scope; they only accept a subset of possible DP algorithms. We also discuss a number of unique high-level abstractions for DP that complement the recurrence abstraction we have selected in this work.

Second, we point out that most related work aims to generate correct code from a high-level abstraction, not necessarily high performance programs. Moreover, most tools generate sequential code rather than parallel programs; very few target accelerator platforms. This is not necessarily due to a lack of interest but is more likely due to a lack of know-how within the community that has developed these solutions. In contrast, this dissertation motivates the need to parallelize dynamic programming and suggests a methodology to do so. An important distinction in our work is that

we seek a way to optimize performance of generated code. In our work we develop design-space exploration algorithms that can optimize various performance criteria.

Finally, the review of related work shows that we can significantly enhance these tools by incorporating the methods developed in this dissertation. We can extend the tools by transforming the parsed input program descriptions of these high-level languages to recurrence equations and then following the workflow we introduced in the previous chapter. We can use these domain-specific tools as a vehicle to further our work and achieve acceptance by the wider community.
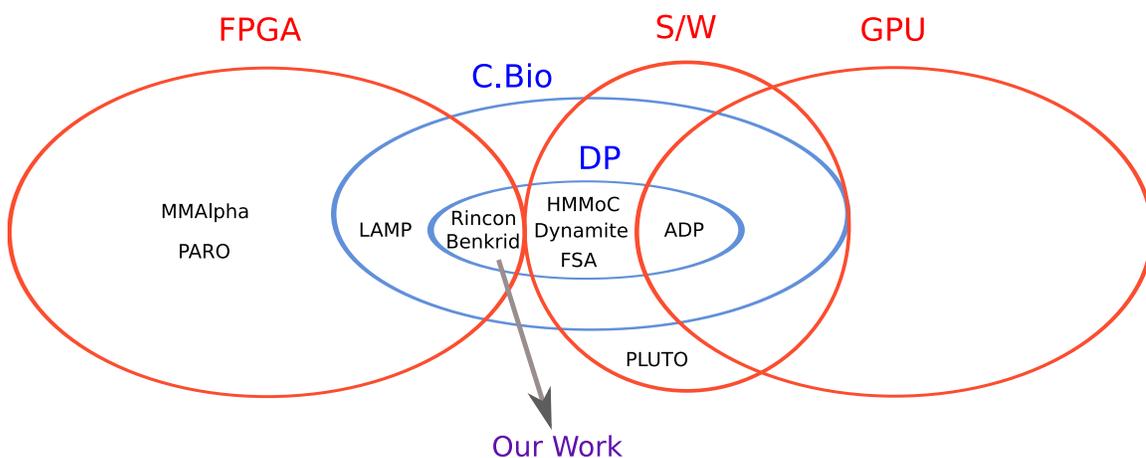
Figure 2.1: A Venn diagram documenting the relationships among related projects surveyed in this chapter. We are interested in tools that generate serial or parallel code for dynamic programming algorithms in computational biology. Here we consider FPGA, GPU, and multi-core accelerators. Three works–MMAlpha, PARO, and PLUTO–target general recurrence equations, rather than computational biology specifically, but are included for their significance.

Figure 2.1 shows the relationship between the various projects within the purview of our dissertation. In this work, we consider tools for accelerating computational biology applications using parallel systems. In particular, we are interested in a subset of these applications that use the DP paradigm. We have considered FPGA, GPU, and multi-core accelerator platforms.

In the next section, we survey high-level code-generation tools for computational biology algorithms. Most of these tools generate only sequential program code. Next,

we look at work done on high-level hardware synthesis tools for computational bio-
logy. Finally, we survey high-level hardware and software synthesis tools for general
recurrence equations, which provide a strong basis for our work.

2.1 High-Level Code-Generation Tools for Computational Biology

There have been several attempts in the research community to ease the development
of DP algorithms in computational biology by providing high-level abstractions.
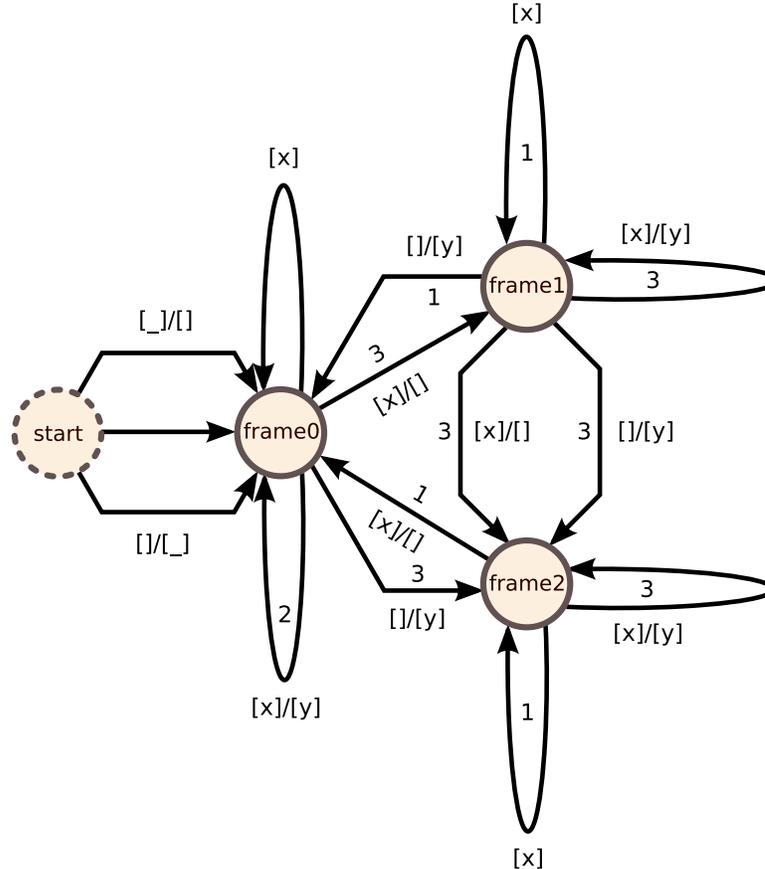
Figure 2.2: A frame maintenance aligner represented by a weighted finite-state ma-
chine is entered graphically into the visual programming environment developed by
Searls and Murphy [135]. The figure is taken from that paper.

34

Finite-state automata. Finite-state automata are a natural and elegant abstraction that has been shown to model alignment algorithms. Karp and Held [82] established their use to model DP algorithms. Searls and Murphy [135] develop the use of weighted finite-state automata to represent mechanisms of biological mutation. One such example, a frame maintenance aligner, is shown in Figure 2.2. This aligner is able to model positions of nucleotide triplets (which code for amino acids) in DNA sequences to improve alignment accuracy. The authors augment each state in these machines with a recursive score function of inputs, transitions, and edge weights, allowing the machine's use for pairwise alignment algorithms.

A visual programming environment generates sequential Prolog and C++ code from these representations. The Prolog code performs DP top-down with the memoization optimization (i.e., caching results of computations) to conserve runtime; nevertheless, the recursive overhead results in two orders of magnitude slowdown compared to the C++ code. No details are given on the C++ backend, which performs DP bottom-up. This is a non-trivial task, as at a minimum, an analysis of the dependencies is required to generate correct code. The authors mention no effort to explore the design space of possible orderings of the computation of subproblems in order to improve runtime. Although the authors do not compare the machine-generated code to hand-optimized implementations, they acknowledge that the latter will always be faster. The authors believe their tool is useful for quickly prototyping new alignment algorithms even though the generated code is not optimal.

Searls and Murphy note that the automata abstraction enables optimizations such as reduction of the number of states (recurrence variables) using standard techniques for finite-state automata, reduction in the dimensionality of recurrence variables, and even the use of sparse matrices to save space. While this is not the problem we are tackling in this dissertation, it is possible to convert DP recurrences into finite-state automata and perform the same optimizations. We believe the finite-state automata abstraction brings ease of use and will help domain experts understand and develop complex DP kernels. Integrating our work into the backend of this visual programming environment will enable a direct path to FPGA implementations.

```
matrix ProteinSW                                state INSERT offi="0" offj="1"
query  type="PROTEIN"    name="query"             source MATCH
target type="PROTEIN"    name="target"              calc="gap"
resource type="COMPMAT" name="comp"                 endsource
resource type="int"      name="gap"              source INSERT
resource type="int"      name="ext"                 calc="ext"
state MATCH offi="1" offj="1"                        endsource
  calc="AAMATCH( comp,                            query_label  INSERT
                 AMINOACID(query,i),              target_label SEQUENCE
                 AMINOACID(target,j)            endstate
                )"                              state DELETE offi="1" offj="0"
  source MATCH                                     source MATCH
    calc="0"                                         calc="gap"
    endsource                                       endsource
  source INSERT                                   source DELETE
    calc="0"                                         calc="ext"
    endsource                                       endsource
  source DELETE                                   query_label  SEQUENCE
    calc="0"                                     target_label INSERT
    endsource                                   endstate
  source START                                  endmatrix
    calc="0"
    endsource
  query_label  SEQUENCE
  target_label SEQUENCE
endstate
```

Figure 2.3: Code for the Smith-Waterman alignment algorithm in the Dynamite language. Note the use of the keyword *state* to represent recurrence variables, *source* to indicate dependencies, and keywords *offi* and *offj* to specify dependency offsets. Only two-dimensional recurrences can be expressed in Dynamite. The example is taken from [20].

Dynamite [18, 20]. This is a code-generation tool that produces C code from an abstract definition of a sequence analysis DP recurrence. Finite-State Machines, though entered in a textual format, are again used to represent the recurrence. One of the innovations of Dynamite is the use of domain-specific constructs to ease development of sequence search tools. Dynamite is designed explicitly for searching a query against a target sequence—and therefore supports only two-dimensional recurrences—with specific syntax provided to represent states (data variables), transitions (dependencies), and computations for each variable. Common domain-specific types to represent protein and DNA sequences are provided, as well as efficient database search routines. Special states are used to represent complex initialization and boundary conditions and to allow multiple matches of the query sequence to the target. Figure 2.3 shows a code sample of the popular Smith-Waterman alignment algorithm represented in the Dynamite language.

Dynamite is capable of generating dynamic programming C code to find the optimal solution to sequence alignment problems in linear space and to recover the best alignment by traceback. The program was used extensively in generating models for the gene-structure prediction programs GeneWise and Genomewise [19]. In addition, the authors report its use for other biological sequence applications, musical copyright databases, and econometrics[6].

Dynamite is limited to the small subset of pairwise DP alignment algorithms. The RNA folding algorithms we accelerate in this dissertation are beyond Dynamite's scope. Dynamite always selects a fixed ordering when computing a two-dimensional DP algorithm and assumes the programmer has coded the input description to satisfy dependencies. No dependence analysis is performed to select a feasible ordering, nor is any design space exploration done to choose an optimal one. Furthermore, Dynamite only generates sequential C code.

Slater and Birney [136] extended Dynamite to automatically generate heuristics for biological sequence comparison. Heuristics are algorithmic approximations to a DP kernel and are used when the original DP-based application is prohibitively expensive. Heuristics can significantly speed up runtime but sacrifice the accuracy of results. While the use of approximations is a legitimate approach to accelerating DP, we are interested here in a faithful acceleration that maintains accuracy of results.

Algebraic Dynamic Programming (ADP).   Another recent effort uses an algebra to describe DP in a high-level language [53]. ADP imposes a structure to DP that firstly describes subproblem decomposition and secondly provides a mechanism to score decompositions. A regular tree grammar, which applies to trees rather than strings, is used to define problem decomposition. The grammar implicitly defines all possible solutions to a DP problem, i.e., the grammar constructs the search space. An accompanying algebra provides a mechanism to score every possible solution of the tree grammar. An ADP example of the Smith-Waterman alignment algorithm is shown in Figure 2.4.

Converting an ADP to recurrence equations is straightforward, but generating imperative code is more challenging. The authors use dependence analysis to select a

---

[6]http://www.ebi.ac.uk/Dynamite/

```
GRAMMAR:

swaterman_alignments alg inpX inpY = axiom ALIGNMENT where
  (nil, DEL, INS, MAT, optimize) = alg

  ALIGNMENT   = tabulated (
                  nil   ><< empty                           |||
                  MAT   <<< xbase -~~ ALIGNMENT ~~- ybase |||
                  DEL   <<< xbase -~~ ALIGNMENT           |||
                  INS   <<<             ALIGNMENT ~~- ybase ... optimize
                )

ALGEBRA:

unit :: swaterman_algebra char int
unit = (nil, DEL, INS, MAT, optimize) where
  nil        = 0
  DEL x s    = s - 1
  INS s y    = s - 1
  MAT a s b  = if a==b then s+1 else s-1
  optimize [] = []
  optimize xs = [maximum xs]
```

Figure 2.4: Sample code for the Smith-Waterman algorithm in the ADP language, which includes a grammar and an evaluation algebra. The code is taken from [53] and scores a match with +1 and a mismatch or a gap with −1.

feasible computation ordering, though no effort is made to optimize for runtime. An extension of the compiler [139] generates parallel GPU code. However, the authors assume a limited set of possible dependencies and use a predetermined parallelization strategy. For example, the strategy the authors use works well for the Nussinov RNA folding algorithm but is infeasible for the Smith-Waterman algorithm.

HMMoC. Lunter [102] wrote a code generator for first- and higher-order hidden Markov models (HMMs) specified in XML format. The input is in a domain-specific language used to describe HMMs. HMMoC automatically generates efficient C++ implementations for DP algorithms on HMMs such as Forward, Backward, and Viterbi. The generated code includes commonly used optimizations such as log-space computation and banding to restrict DP to a certain region.

Summary. Tools such as Dynamite, ADP, and HMMoC play an important role in easing development of DP algorithms for domain experts and show interest of the computational biology community in domain-specific abstractions and compilation.

Most of these tools can only generate sequential code, or exploit limited forms of parallelism, whereas our contribution is in the use of polyhedral theory to parallelize any DP kernel. Our work is complementary to these tools; they can be augmented to generate recurrence equation descriptions, which can be analyzed and parallelized using the methods we introduce in this dissertation.

2.2   High-Level Hardware Synthesis Tools for Computational Biology

Researchers have recently begun to tackle the problem of easing the development of hardware accelerators for sequence analysis applications.

LAMP.   Van Court and Herbordt describe their approach in several papers [32, 152] and a Ph.D. thesis [31]. They state that one of the reasons for the lack of special-purpose accelerators in wide use in this field is the brittleness of solutions, i.e., their inflexibility in handling the customizations required by the biologist. Instead of focusing on point solutions, they stress the importance of developing a family of implementations.

Van Court [32] describes an architecture for general sequence search composed of modules that are independent and customizable. The Logic Architecture Model Parametrization (LAMP) tool suite [31, 152] provides mechanisms to automate and ease the development of an accelerator from a model describing a family of algorithms. Designing an accelerator in LAMP is based on roles assigned to the hardware designer and a domain expert.

The LAMP model describes the control, dataflow, and overall organization of an architecture. It is specified by a hardware designer in a markup language embedded in a standard HDL. Significant manual effort must be expended to keep the architecture as general as possible. The domain expert implements the functionality for each module in the model, customized for a point solution. A stated aim is to provide

a set of parametrized modules to the domain expert, though new modules are still written in an HDL.

Our research differs significantly from that of Van Court in its high-level approach. Van Court acknowledges a semantic gap between application authors and the FPGA fabric. He sees little hope for this gap to be bridged in the general case. His solution is to clearly delineate the work of the hardware designer and the domain expert. Although the LAMP tool suite enables easier development of families of any application (such as approximate string matching or rigid molecule interactions) and is not limited to DP, the burden on the hardware designer has not been decreased. Additionally, since a specific architecture is selected by the designer, it provides little freedom for design space exploration by the compiler.

Other Tools. Benkrid et al. [16] describe a parametrized skeleton for sequence alignment that is written in the high-level HDL Handel-C. Ayala-Rincón et al. [11] use rewriting logic to model systolic architectures for families of DP algorithms.

Summary. All of the work we have described relies on the designer to explore the design space and decide on a specific array configuration. As we will demonstrate in subsequent chapters, this requires considerable skill for all but the simplest of DP algorithms. Providing algorithms for automated analysis greatly eases the burden on the developer.

2.3   High-Level Synthesis Tools for Recurrence Equations

High-level synthesis tools for recurrence equations are the most general class of compilers in our survey. These tools attempt to automatically parallelize high-level code for execution on FPGA systems or multicore platforms. They are limited to compiling recurrence equations specified in a custom language or sequential loop programs.

PLUTO [22]. This is a recently developed open-source compiler to synthesize parallel openMP code for multicore systems. The compiler accepts subsets of C loops and automatically generates tiled, parallel C code. The compiler performs design space exploration to improve cache locality and exploit parallelism in tiled code. Because PLUTO targets multicores, not custom hardware, the optimization criteria and techniques used are not an ideal match for our use. For example, we optimize for throughput and have the freedom to build custom hardware.

MMAlpha. The MMAlpha tool [61, 154] is a semi-automatic compiler that generates synthesizable VHDL code describing systolic arrays. The compiler accepts input descriptions in a custom, single assignment language that represents recurrence equations. Localization to remove broadcast dependencies is supported in a semi-automatic fashion. MMAlpha can synthesize a single latency-space optimal array. Design space exploration is limited to a single component of the array mapping, namely, to finding a latency optimal schedule—space optimization is not performed. Resource-constrained array mappings are also beyond the capability of the compiler.

An advantage of MMAlpha is that it is freely available as an open-source download.

PARO [63]. Over the last decade, a more complete systolic array compiler for VLSI systems has been developed. Resource-constrained array mapping is supported, as is partitioning to process large input problems. The approach in PARO is to build a single optimal latency-space or resource-constrained array. While the compiler is not publicly available, the authors state that it performs design space exploration to optimize latency and area usage.

Summary. While PLUTO does not target hardware accelerators, MMAlpha and PARO are multi-decade research efforts that come closest to our goal of synthesizing hardware accelerators that exploit parallelism using polyhedral analysis. While these tools target loop programs, we have in turn applied polyhedral analysis to the class of dynamic programming algorithms.

The ideas developed in this dissertation differs significantly from the existing compiler projects. First, neither project considers throughput-optimized arrays. This is a novel paradigm within the polyhedral framework that we have proposed, which is able to exploit far more parallelism than existing techniques. None of these compilers is able to pipeline independent instances on the array, nor pipeline iterations in a processor to improve clock frequency. Hence, these tools will be less suitable for domains such as computational biology and media processing, where throughput is the primary concern; certainly, the novel arrays we have developed for Nussinov RNA folding would not have been discovered using the techniques implemented in the surveyed compilers.

Furthermore, all these approaches maintain that a single optimal array, whether latency-space, or resource-constrained, is sufficient as a hardware accelerator for any computation kernel. While appropriate for ASIC design, we will show in this dissertation that when applied to FPGAs, we can exploit reconfiguration to select from a range of arrays that tradeoff resources for increase parallelism, and so are optimal for various input size ranges. Selecting from our throughput-optimized arrays, as well as various resource-constrained arrays, we are able to show a several-fold speedup versus using a single resource-constrained or latency-space optimal array.

We note that the extensions that have been proposed in this work can augment MMAlpha and PARO. The software tools we have developed can be integrated into the optimization phase of these compilers to optimize for throughput, and support FPGA reconfiguration.

Chapter 3

Background: Parallelization of Dynamic Programming

In this chapter we introduce the background necessary to follow the parallelization techniques employed in this dissertation. We start by characterizing the class of dynamic programming algorithms targeted in this work. Since we are primarily interested in FPGAs as our implementation target, we focus our attention on an efficient hardware array architecture, the systolic array.

Next, we define the powerful systems of recurrence equations (SRE) abstraction and show how they may be utilized to represent both DP kernels and systolic arrays. The problem of identifying an appropriate systolic array to accelerate a DP kernel is translated to the problem of identifying the right recurrence-to-recurrence transformation.

We give a brief overview of polyhedral theory and show its use to transform SREs representing DP into those representing a systolic array. Finally we summarize state-of-the-art methods to build unconstrained latency-space optimal array mappings from DP recurrences. We have adapted much of the introductory material in this chapter from texts by Lavenier et al. [96] and others [144].

3.1 Dynamic Programming

In this dissertation we will tackle only discrete-input dynamic programming algorithms. By discrete we mean that the underlying decision process operates on an

ordered list of distinct input data items such as biosequences, alignments of multiple sequences, motifs represented by a hidden Markov model, or an ordered list of graph vertices.

In Section 3.4, we describe a popular classification for DP algorithms with the help of the recurrence equations abstraction. In general, DP algorithms from all classes can be parallelized using the techniques we employ. However, we briefly mention a subset of DP algorithms that cannot be efficiently accelerated using the parallelization technique and the accelerator architecture we employ.

Finally, we note that our parallelization strategy holds when the underlying decision process is either deterministic or stochastic. Supporting stochastic processes is important because they abound in computational biology, for example, probabilistic hidden Markov models and stochastic grammars.

3.2 Systolic Arrays

A *systolic array* [90] is a synchronous, data-driven computing network with simple, modular, *processing elements* (PE) and a regular interconnecting network. Data is passed through the network in a pipelined fashion synchronized by a global clock. The systolic paradigm is data-driven, unlike Von Neumann architectures, and is well suited for accelerating parallel, compute-bound applications. An important feature of the systolic approach is that a data item, once retrieved from memory, is processed through the entire array before being retired.
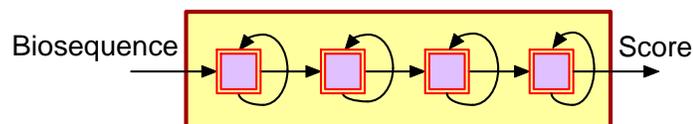
Figure 3.1: A linear, four-processing-element systolic array. In this example pipeline based parallelism occurs and, for long sequences, a speedup of four over a single processor is possible.

Figure 3.1 shows a four-PE linear systolic array used for a sequence alignment algorithm. The input sequence is fed into the leftmost PE, processed internally, and

transferred to its right neighbor. The final score is emitted by the rightmost PE in the array.

Systolic arrays can exploit the high degree of parallelism typically available in DP algorithms. The local and regular communication network simplifies routing costs and enables implementation of highly scalable designs in hardware.

3.2.1   History of Systolic Arrays

Systolic arrays were first introduced by Kung [90] in the early 1980s to exploit massive parallelism. Systolic arrays have been architected to accelerate image processing, text search, neural networks, finite element methods, and many other applications. We refer the interested reader to an overview by Lavenier et al. [96] for more details. In this section, we give a brief history of systolic arrays that have been used for computational biology analysis.

An early example of a systolic array was the Splash 2 machine [67], which was used for sequence matching with the Needleman-Wunsch dynamic programming algorithm. The machine consisted of 16 boards, each with 16 Xilinx 4010 FPGAs connected in a linear array. The authors implemented a linear systolic array for the algorithm in VHDL, with each of the 16 FPGAs implementing 14 PEs. A single board of the Splash 2 machine achieved three orders of magnitude speedup over a SPARC 10/30GX workstation.

SAMBA [59] was a 128-PE fully custom systolic array specifically designed to implement the Smith-Waterman sequence matching algorithm. The Kestrel processor [72] was a linear VLSI array consisting of 512 PEs. Although originally designed with sequence alignment in mind, the array is general-purpose and was used to accelerate graph problems, machine learning, and a computational chemistry application. Programming systolic arrays has always been a challenge that has never been satisfactorily addressed. The authors of the Kestrel processor developed a compiler to generate code for it but never used it to program the array, preferring to write hand-optimized assembly language programs.

3.3 Systems of Recurrence Equations

A *recurrence equation* is a variable that is defined at all integral points $\mathbf{z}$ in an n-dimensional domain D by an equation of the form:

$$X(\mathbf{z}) = \begin{cases} \quad \vdots \\ e_i(\ldots X(f(\mathbf{z}))\ldots) \qquad \text{if } \mathbf{z} \subseteq \mathcal{D}_i. \\ \quad \vdots \end{cases} \tag{3.1}$$

Here:

- $\mathbf{z} = \begin{bmatrix} i_1 \\ \vdots \\ i_n \end{bmatrix}$ is an integral *iteration* vector of n dimensions. It can be thought of as the indices of an n-dimensional nested *for* loop implementing the recurrence computation. We will alternatively refer to $\mathbf{z}$ as a *computation point*, or simply a point.

- X is a *data variable* indexed by $\mathbf{z}$. It may be thought of as an n-dimensional array. $X(\mathbf{z})$ denotes the value of the variable at a specific point.

- A variable may have multiple disjoint definitions (cases), also known as *domains*, represented here by $\mathcal{D}_i$. A domain is a set of points in $\mathbb{Z}^n$ where the data variable is defined. The domain of variable X is the union of the domains of the individual definitions. In this work, we consider only domains that can be represented as convex integer polytopes (see Section 3.6).

  A recurrence domain may be parametrized, i.e., it may have one or more *size parameters*, such as the length of an input sequence.

- The value of X at a point $\mathbf{z} \in \mathcal{D}_i$ is given by evaluating the expression e_i. The expression is a strict (always evaluates its arguments), single-valued function of its arguments and elementary operators. Each expression is evaluated only when its guarding predicate $\mathbf{z} \subseteq D_i$ is satisfied.

  We will refer to the collective list of expressions as the *body* of the recurrence, which is analogous to the body of a loop implementing this recurrence.

- $f(\mathbf{z})$ is a *dependency function*, $f : \mathbb{Z}^n \rightarrow \mathbb{Z}^n$. Given variable values at two computation points $X(\mathbf{z_1})$ and $X(\mathbf{z_2})$, we say that $X(\mathbf{z_1})$ *depends* on $X(\mathbf{z_2})$

if the latter is required for the computation of the former. We denote this dependency by $X(\mathbf{z_1}) \leftarrow X(\mathbf{z_2})$, and there exists a dependency function f such that $\mathbf{z_2} = f(\mathbf{z_1})$.

There may be multiple references to the data variable X in the right-hand-side definitions, each with a possibly distinct dependency function f.

An *affine recurrence equation* (ARE) has all its dependency functions in the form $f(\mathbf{z}) = A\mathbf{z} + \mathbf{b}$ where A is constant $n \times n$ matrix and $\mathbf{b}$ is a constant n-vector. A *uniform recurrence equation* (URE) has all dependency functions in the form $f(\mathbf{z}) = \mathbf{z} + \mathbf{b}$; i.e, a URE is an ARE where A is the identity matrix.

A *system of recurrence equations* (SRE) is a set of recurrence equations of the form in Equation 3.1 defining data variables $X_1 \cdots X_m$. These definitions may be mutually recursive. *Systems of affine recurrence equations* (SAREs) have affine dependencies and permit each data variable to be of varying dimension. *Systems of uniform recurrence equations* (SUREs) require that dependencies be uniform and force all data variables to be of the same dimension.

Recurrence equations with all uniform dependencies can be implemented on a systolic array with a resource efficient near-neighbor interconnection network. However, affine dependencies induce long-range communication and require broadcasts of data variables. In this work we only target SUREs for acceleration; we employ a process called *localization*, described in Section 3.6.4, to make recurrences with affine dependencies uniform.

Example. We illustrate these concepts with the following simple system of uniform recurrence equations.

$$
C(i,j) = \begin{cases} 0 & \text{if } j = 0 \text{ and } 1 \le i \le M \\ C(i, j-1) + a_{j,i} \cdot B(i,j) & \text{if } 1 \le j \le M \text{ and } 1 \le i \le M \end{cases} \tag{3.2}
$$

$$
B(i,j) = \begin{cases} b_j & \text{if } i = 1 \text{ and } 1 \le j \le M \\ B(i-1,j) & \text{if } 2 \le i \le M \text{ and } 1 \le j \le M . \end{cases} \tag{3.3}
$$

Here:

- $\mathbf{z} = \begin{bmatrix} i \\ j \end{bmatrix}$ is the iteration vector of the two-dimensional recurrences.

- There are two data variables, B, and C, both indexed by $\begin{bmatrix} i \\ j \end{bmatrix}$. The labels a and b represent input data items made available to the recurrence and are not data variables.

- Variable C has two disjoint domains:

  1. $\mathcal{D}_1 = \{\, i, j \mid 1 \le i \le M; j = 0 \,\}$.
  2. $\mathcal{D}_2 = \{\, i, j \mid 1 \le i \le M; 1 \le j \le M \,\}$.

  The complete domain of variable C is $\mathcal{D}_1 \cup \mathcal{D}_2 = \{\, i, j \mid 1 \le i \le M; 0 \le j \le M \,\}$. In fact, both variables share the same computation domain.

  The example is a parametrized system of recurrence equations with one parameter, M.

- The first of two dependencies in this example is $C(i, j) \leftarrow C(i, j-1)$; we say that $C(i, j)$ depends on $C(i, j-1)$ because the latter is required for the computation of the former. The second dependency is $B(i, j) \leftarrow B(i-1, j)$.

We can represent the dependency in Recurrence 3.2 by the dependency function $f_1(\begin{bmatrix} i \\ j \end{bmatrix}) = A_1 \begin{bmatrix} i \\ j \end{bmatrix} + \mathbf{b_1}$, where $A_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ and $\mathbf{b_1} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$. Since A_1 is the identity matrix, the dependency is uniform. Similarly, the dependency in Recurrence 3.3 is given by the dependency function $f_2(\begin{bmatrix} i \\ j \end{bmatrix}) = A_2 \begin{bmatrix} i \\ j \end{bmatrix} + \mathbf{b_2}$, where $A_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ and $\mathbf{b_2} = \begin{bmatrix} -1 \\ 0 \end{bmatrix}$. Because both dependencies are uniform, the example is a system of uniform recurrence equations.

3.4 Dynamic Programming as Systems of Affine Recurrence Equations

We know that all DP algorithms satisfy the *principle of optimality* [15]. Bellman gave a mathematical equation for this principle which is in a functional form. It is straightforward to see that this representation is equivalent to a system of recurrence equations.

Indeed, various models that are used for DP easily translate to systems of recurrence equations. Karp and Held [82] use a finite-state-automata-based model for DP, and Giegerich and coworkers [53] use a tree grammar, which is analogous to context-free grammars. Both works establish that a recurrence can always be derived from the respective models if a certain monotonicity property on the cost function, akin to the principle of optimality, is satisfied.

3.4.1 Classification of Dynamic Programming

We now describe a classification scheme for DP due to Wah et al. [155] that is useful for the informed reader to understand the kinds of DP algorithms accepted by our work. We will use the recurrence abstraction to illustrate the distinguishing features of each class.

DP formulations may be divided according to the form of the recurrence expression and the nature of the dependency. A DP formulation is termed *monadic* if its expression uses only a single recursive dependency, otherwise it is *polyadic*. An example of a monadic formulation is $X(i) = \min_{j<i}\{\ X(j) + c_{i,j}\ \}$. The single dependency is the data variable $X(j)$. Note, that $c_{i,j}$ is not a data variable but rather a precomputed table of cost values. An example of a polyadic formulation is $X(i,j,k) = \min\{\ X(i,k,k-1) + X(k,j,k-1)\ \}$, which uses two recursive X terms.

Serial DP formulations have expressions that depend on recursive data variables from an immediate predecessor. *Non-serial* DP formulations have arbitrary dependencies. An example of a serial formulation is $X(i,j) = \min\{\ X(i-1,j),\ X(i,j-1)\ \}$, and an example of a non-serial formulation is $X(i,j) = \min_{i\leq k\leq j}\{\ X(i,k) + X(k,j)\ \}$.

Dynamic programming problems are therefore classified as *monadic serial, monadic non-serial, polyadic serial*, and *polyadic non-serial*.

Monadic and polyadic formulations pose no problem to efficient hardware acceleration. Moreover, serial formulations can use a simple nearest-neighbor network when implemented on a systolic array. Non-serial formulations are more problematic because an accelerator that implements such algorithms must broadcast data values,

resulting in limited scalability and expensive interconnection networks. Fortunately many non-serial formulations in important DP algorithms do not have arbitrary dependencies but rather follow specific patterns that can be recognized and converted to a serial formulation. Our parallelization technique can handle non-serial DP kernels whose dependencies are defined as affine functions. We can then use the localization transformation to make such dependencies uniform.

A set of non-serial DP formulations that pose a serious challenge to acceleration using the techniques we employ are those that have recursive dependencies decided by input values. Consider a decision process over an input graph $G = (V, E)$ where V is the graph's vertices and E its edges. Let there be some DP algorithm for this process to find the minimum cost X at some vertex v_i with the formulation: $X(v_i) = \min_{(v_i, v_j) \in E} \{ X(v_j) + c(v_i) \}$. Here $X(v_i)$ depends on $X(v_j)$, where $j \neq i$, only if $(v_i, v_j) \in E$. Our parallelization technique requires that all recurrence dependencies be known at compile time and cannot easily handle non-serial formulations that are dependent on the input (in this case, the edge set). We must rewrite such recurrences to remove the dependency on the input. In this example, we force an ordering on the vertices and rewrite the recurrence as: $X(v_i) = \min_{1 \leq v_j \leq |V|} \{ X(v_j) + c(v_i) + exclude(v_i, v_j) \}$, where the function $exclude$ returns ∞ if $(v_i, v_j) \notin E$. This does, however, result in worst-case runtime behavior.

Summarizing this discussion more formally, our work is suitable to accelerate any DP algorithm that can be represented as an affine system of recurrence equations over an integral domain. Even with these limitations, we are able to accelerate a large number of DP kernels used in computational biology and other domains.

3.5   Systolic Arrays as Systems of Uniform Recurrence Equations

Every systolic array can be described by a SURE [96], where:

- All data variables of the recurrences have the same number of dimensions;
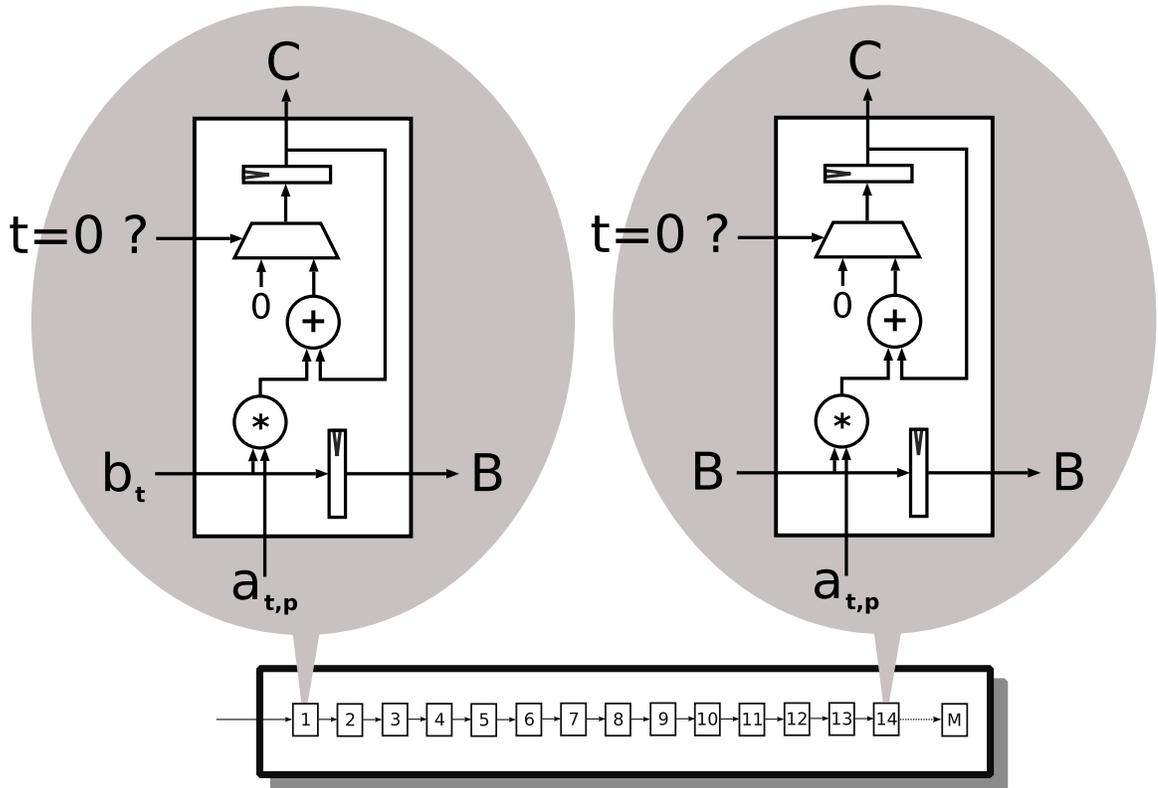
(a) A linear, M-PE systolic array that uses two distinct PE types. Registers are clocked by a global clock (not shown).

$$C(p, t) = \begin{cases} 0 & \text{if } t = 0 \text{ and } 1 \le p \le M \\ C(p, t-1) + a_{t,p} \cdot B(p, t) & \text{if } 1 \le t \le M \text{ and } 1 \le p \le M \end{cases}$$

$$B(p, t) = \begin{cases} b_t & \text{if } p = 1 \text{ and } 1 \le t \le M \\ B(p-1, t) & \text{if } 2 \le p \le M \text{ and } 1 \le t \le M \end{cases}$$

(b) A system of uniform recurrences corresponding to the systolic array with PE index p and time index t. This is similar to the example recurrence above, but with the indices reinterpreted.

Figure 3.2: Every systolic array can be described by a system of uniform recurrence equations.

- The time dimension is represented by a single element of the iteration vector, and the remaining elements represent the processing element coordinates;

- The computation pipeline in a processor (PE) is equivalent to the computation in the body (expressions) of the recurrence;

- Each link in the regular interconnection network of the systolic array has an equivalent uniform dependency in the recurrence;

- External input into the array is represented by input variables in the SURE and is used exactly once.

Figure 3.2 illustrates a systolic array and its equivalent system of uniform recurrence equations (which are the same as the example in Section 3.3). The recurrences' iteration vector can be reinterpreted as having one element representing the PE index in the systolic array, and the other element as time. The array uses two types of PEs: one at $p = 1$, and the second when $p > 1$. Each PE implements a custom datapath described by the recurrence body. A recurrence data variable is implemented in a PE by a register—in this example, two registers correspond to variables B and C.

There are two uniform dependencies in the SUREs, $B(p, t) \leftarrow B(p - 1, t)$, and $C(p, t) \leftarrow C(p, t - 1)$. As shown in the figure, the former corresponds to a spatial dependency that results in inter-PE, left-to-right communication, while the latter corresponds to a temporal dependency within a PE.

A multiplexer is used to initialize variable C. The multiplexer is controlled by a one-bit signal that is high when the predicate $t = 0$ is satisfied. The labels a and b correspond to input data values fed into the array by a controller.

3.6 Parallelization of Uniform Recurrences

We now describe how any DP algorithm that is represented by a uniform recurrence can be parallelized to generate a systolic array. We use polyhedral theory [122] to analyze and exploit the parallelism in recurrence equations.

3.6.1 Preliminaries of the Polyhedral Representation

In this dissertation, we target recurrence equations whose computation domain $\mathcal{D}$ can be described by a convex integer polytope. The set of iteration vectors $\mathbf{z}$ of these recurrences is defined by a set of inequalities representing an n-dimensional convex space:

$$\mathcal{D} = \{\, \mathbf{z} \in \mathbb{Z}^n \mid Q\mathbf{z} \le \mathbf{q} \,\}. \tag{3.4}$$

Here Q is a $d \times n$ matrix and $\mathbf{q}$ is a d-vector, where d is the number of inequalities describing the convex space. A data variable X in this recurrence is defined at every point in the domain, and $X(\mathbf{z})$ denotes the value of the variable at point $\mathbf{z}$.

The convex domain has an equivalent dual representation called its *generating system*. The generating system is given as a linear combination of its vertices $\mathcal{V} = \{\, v_1, \ldots, v_w \,\}$

$$\mathcal{D} = \{\, \mathbf{z} \in \mathbb{Z}^n \mid \mathbf{z} = \sum_{i=1}^{w} a_i v_i \,\} \tag{3.5}$$

where a_i are rationals and $\sum_{i=1}^{w} a_i = 1$.

The two representations of the computation domain are a succinct representation of a computation kernel and are useful for analysis and exploitation of the kernel's parallelism. Traditional high-level hardware synthesis tools unroll the loop representation of these kernels and must analyze every statement for parallelism (equivalent to inspecting every point in the computation domain). In contrast, polyhedral analysis operates on the kernel's polytope and is able to make broad generalizations on the availability of parallelism by only considering a few points in the polytope (such as its vertices).

In the next section, we show how the polyhedral representation may be used to discover and exploit parallelism in recurrence equations. We do so by finding a linear function termed an *array mapping* that transforms recurrence equations representing dynamic programming to equations that can be used to generate a parallel array.

3.6.2 Array Mappings for Recurrence Equations

A uniform recurrence is converted into a systolic array by finding a space-time array mapping. For all iteration vectors z computed by the recurrence, this mapping specifies:

- A *schedule* or a *timing function*, which gives the execution time of each point in the computation domain. This function is specified by an affine transformation τ.

- An *allocation function*, which determines the physical PE executing each point. The allocation is specified by a linear transformation π.

- The systolic array is then generated, executing computations in parallel and respecting the schedule.

Scheduling

Given a SURE, the scheduling function maps each data variable instance $X(\mathbf{z})$ to a computation time $\tau(\mathbf{z}) : \mathbb{Z}^n \to \mathbb{Z}$ such that:

- $\tau(\mathbf{z}) \geq 0$, $\forall\, \mathbf{z} \in \mathbb{Z}^n$ (positivity condition)

- $X(\mathbf{z_1}) \leftarrow X(\mathbf{z_2}) \Rightarrow \tau(\mathbf{z_1}) > \tau(\mathbf{z_2})$, $\forall\, \mathbf{z_1}, \mathbf{z_2} \in \mathbb{Z}^n$ (causality condition).

The positivity condition is for convention and ensures that all points are computed at non-negative time. The causality condition ensures that all dependencies are satisfied, i.e., before scheduling point $\mathbf{z_1}$, we must schedule its dependent value at $\mathbf{z_2}$. Note that the scheduling function assumes that the entire body of the recurrence is executed in a single clock cycle.

We may derive a distinct schedule for every data variable in the SURE, which allows fine-grained control and pipelining of operations in the body of the recurrence; however, in this work, we will assume a global schedule for all variables in the system of recurrences.

A scheduling function need not exist, for example, when there are cyclic dependencies in the recurrences. Computability of the scheduling function is undecidable in the general case but is known to be decidable for uniform recurrences, when all data variables have the same computation domain, and when the recurrence body is a strict (always evaluates its arguments), single-valued function [96].

An important point to note is that the schedule must be a closed-form function and cannot be specified by enumerating the execution times of every point in the domain. We are able to derive a compact representation for the schedule because the computation domain of the recurrence is a convex polytope.

We specify closed-form schedules as an affine function given by:

$$\tau(\mathbf{z}) = \lambda \cdot \mathbf{z} + \alpha, \tag{3.6}$$

where λ is a $n-$vector and α is a constant. Finding a good schedule for the recurrence involves finding suitable λ and α values.

Allocation

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

- $\tau(\mathbf{z_1}) = \tau(\mathbf{z_2}) \Rightarrow \pi(\mathbf{z_1}) \neq \pi(\mathbf{z_2}), \ \forall \ \mathbf{z_1}, \mathbf{z_2} \in \mathbb{Z}^n$ (injectivity condition).

The allocation function is a linear transformation specified by the matrix π of dimension $n\text{-}1 \times n$. It can be equivalently specified by a direction vector $\mathbf{u}$ such that $\pi\mathbf{u} = 0$. This vector is the direction along which the n-dimensional computation domain is projected, for execution, on an $(n\text{-}1)$-dimensional PE array. Given only a projection vector, the allocation function π can be constructed from it.

The injectivity condition ensures that if two points are executed at the same time instant, they will be computed on different PEs. This condition is given by the constraint $\lambda \cdot \mathbf{u} \neq 0$, for a schedule $\tau(\mathbf{z}) = \lambda \cdot \mathbf{z} + \alpha$.

The space-time array mapping is given by the linear transformation $\left[\begin{smallmatrix} \pi \\ \lambda \end{smallmatrix} \right]$ and trans-forms a DP recurrence with iteration vectors $\begin{bmatrix} i_1 \\ \vdots \\ i_n \end{bmatrix}$ into a systolic array recurrence with iteration vectors $\begin{bmatrix} p_1 \\ \vdots \\ p_{n-1} \\ t \end{bmatrix}$. Note the single time index and the n-1 PE indices.

Systolic Array Generation

The allocation transformation maps the computation domain onto the processor space, such that each integer point in the processor space instantiates a PE. Enumerating these integer points in a space is a classic problem called *scanning* of a polyhedron and is implemented using an algorithm such as Fourier-Motzkin elimination [8].

The datapath in each PE is implemented by a direct translation of the recurrence expressions. If a data variable has multiple disjoint definitions (cases), a multiplexer must be generated to select the appropriate expression value when its guarding predicate is satisfied.

Each uniform dependency $f(\mathbf{z}) = \mathbf{z} + \mathbf{b}$ in the recurrence is converted to a link in the direction $\pi(\mathbf{b})$ with a latency of $\lambda \cdot \mathbf{b}$ clocks. If $\pi(\mathbf{b})$ is the zero vector, the dependency is a temporal link within a PE; otherwise, it is a spatial link resulting in inter-PE communication. Because all dependencies are uniform, not affine, the inter-PE communication network is regular and efficient to synthesize in hardware.

3.6.3 Latency-Space Optimal Array Mappings

For any DP algorithm, there are a large number of possible accelerator configurations that implement it. Most projects that build application-specific accelerators for DP algorithms use an ad-hoc design process, often with no guarantee on the optimality of the design choices. Fortunately, the polyhedral method provides a framework to characterize important properties of the recurrence and build an optimized accelerator. In our study, this problem can be framed as finding the right space-time array mapping.

Since an array mapping embodies the processor configuration and schedule of execution on an accelerator, a natural idea is to find a mapping for a latency-space optimal array [96]. Such an array can compute the recurrence using the lowest-latency schedule; among all possible arrays with latency-optimal schedules, we select the one that requires the fewest processors.

Typically, there are a large number of possible schedules and allocations for an input recurrence, and it is expensive to enumerate every single one for evaluation. In this section, we summarize work done by researchers to efficiently explore this design space. The approach is to first, find a latency-optimal schedule using an integer linear program formulation [13, 99, 108], and second, find a compatible space-optimal allocation using a bounded search [161].

Latency-Optimal Schedules: Vertex Method

Recall that since a schedule is given by the affine function $\tau(\mathbf{z}) = \lambda \cdot \mathbf{z} + \alpha$, finding a good schedule involves finding suitable λ and α values with the constraint that the positivity and the causality conditions are met.

We describe the *vertex method* [13, 99, 108], which uses the polytope representation of recurrences to efficiently evaluate schedule candidates. It depends on the property that linear constraints are satisfied at all points in a convex polytope if and only if they are satisfied at its extremal vertices. This property allows us to derive a compact set of constraints to efficiently search for good schedules.

We may find a minimum-latency schedule by solving the following optimization problem.

$$
\begin{aligned}
\min \quad & (\max_{\mathbf{z_1}, \mathbf{z_2} \in \mathcal{D}} \tau(\mathbf{z_1}) - \tau(\mathbf{z_2})) \quad \text{subject to} \\
\tau(\mathbf{z_1}) \;\geq\; & 0 \\
\tau(\mathbf{z_1}) \;>\; & \tau(\mathbf{z_2}) \text{ if } X(\mathbf{z_1}) \leftarrow X(\mathbf{z_2})
\end{aligned}
\tag{3.7}
$$

for every point $\mathbf{z_1}$ and $\mathbf{z_2}$ in the computation domain. The objective function aims to minimize the maximum difference in execution times of any two points in the computation domain. The two constraints are added to satisfy the positivity and causality conditions respectively of Section 3.6.2. Unfortunately, the objective and

constraints are non-linear and can be made linear only by enumerating all points in the domain. However, this is impractical as it leads to an explosion in the number of constraints.

Fortunately, we can use a compact representation of a polytope to derive an integer linear program and efficiently solve the above optimization problem. We know that the positivity and causality constraints are satisfied at all points in the computation domain if and only if they are satisfied at the domain's vertices. We therefore use the vertex set $\mathcal{V}$ from the generating system representation of the recurrence's computation domain (see Section 3.6.1) to derive the following integer linear optimization problem to solve for λ and α.

$$
\begin{aligned}
\min \quad & (\max_{\mathbf{d} \in \mathbb{C}} \lambda \cdot \mathbf{d}) \quad \text{subject to} \\
\lambda \cdot v + \alpha \geq{} & 0 \ \ \forall \, v \in \mathcal{V} \\
\lambda \cdot \mathbf{b_i} <{} & 0 \ \ \forall \, f_i(\mathbf{z}) = \mathbf{z} + \mathbf{b_i}
\end{aligned}
\tag{3.8}
$$

where $\mathbb{C} = \{ \, v_1 - v_2 \mid v_1, v_2 \in \mathcal{V} \, \}$. The first constraint, applied at every vertex of the polytope, is a result of the positivity requirement. The second constraint is applied for all distinct uniform dependencies of the recurrence and is a result of the causality requirement.

The size of the integer linear program specification depends on a small number of vertices and uniform dependencies in the recurrence. In practice, we are able to solve this optimization problem in fractions of seconds using modern integer linear program solvers.

Optimizing Allocation

Once a schedule has been selected, the next task is to find a compatible optimal allocation transformation. There are a number of criteria that may be optimized, including the number of PEs instantiated in the array, the length and number of inter-PE communication links, and the number of PEs performing I/O from the array.

Most important, however, is the area cost, which may be reduced by finding a minimum-PE allocation. Unfortunately, there is no known technique to formulate a linear optimization problem for finding a space-optimal allocation because the number of PEs instantiated by any allocation on a n-dimensional polytope cannot be computed by a linear objective [33]. Instead, we use an approach by Wong and Delosme [161] that enumerates all candidate allocations for inspection. The authors derive bounds that can be used to limit the size of the search space and find a space-optimal allocation in reasonable time.

3.6.4 Localization

Before we end this chapter, we give an overview of how to handle recurrences with affine dependencies. Non-serial DP kernels with affine dependencies require broadcasting of variable dependencies and cannot be implemented on a regular inter-PE communication network. Fortunately we can make these dependencies uniform through a procedure called *localization*.
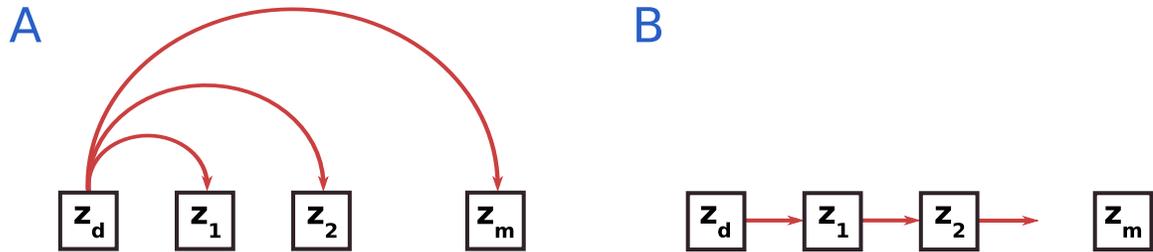
Figure 3.3: (a) Affine dependencies result in data broadcasts. (b) We can derive an equivalent uniform recurrence by pipelining broadcasts.

Localization is done by pipelining data broadcasts, which is explained with an example in Figure 3.3. Suppose a variable value $X(\mathbf{z_d})$ is an affine dependency for $X(\mathbf{z_1}), \ldots, X(\mathbf{z_m})$. Rather than broadcast $X(\mathbf{z_d})$ to all dependents as shown in Figure 3.3a, we may first deliver it to $\mathbf{z_1}$, then transfer it from $\mathbf{z_1}$ to $\mathbf{z_2}$, from $\mathbf{z_2}$ to $\mathbf{z_3}$, and so forth up to $\mathbf{z_m}$. If each of these transfers is uniform, we achieve a regular interconnection network.

In this section, we describe two situations where the technique of localization is applied. *Data pipelining* is used to make affine dependencies uniform. *Control pipelining* is used to efficiently synthesize predicates that guard expressions in the body of the recurrence, without using expensive comparator circuits.

Data Pipelining

Localization is carried out through a procedure termed *nullspace pipelining* [123]. Let $X(\mathbf{z_1}) \leftarrow X(\mathbf{z_d})$ be an affine dependency represented by the function $f(\mathbf{z}) = A\mathbf{z} + \mathbf{b}$. Recall that A is not the identity matrix. Since by definition $\mathbf{z_d} = f(\mathbf{z_1})$, we have $X(\mathbf{z_1}) \leftarrow X(f(\mathbf{z_1}))$.

Nullspace pipelining aims to replace $X(f(\mathbf{z_1}))$ with a new pipeline variable $X_{pipe}(f_u(\mathbf{z_1}))$, such that $f_u(\mathbf{z}) = \mathbf{z} + \mathbf{u}$ is a uniform dependency. The recurrence definition for the new pipeline variable is of the form:

$$
X_{pipe}(\mathbf{z}) = \begin{cases} X(\mathbf{z} + \mathbf{u'}) & \text{if } \mathbf{z} = \bar{\mathbf{z}} \\ X_{pipe}(\mathbf{z} + \mathbf{u}) & \text{otherwise.} \end{cases} \tag{3.9}
$$

The recurrence simply transfers its input to the output without performing any computation.
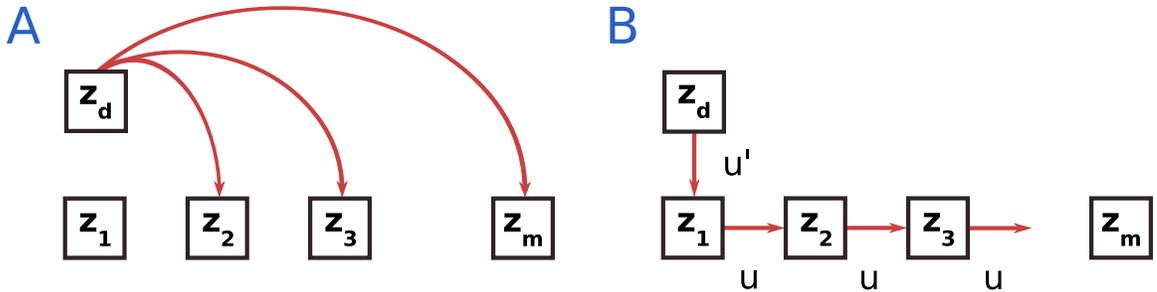
Figure 3.4: An affine dependence on point $\mathbf{z_d}$ is made uniform using a pipeline variable of the form in Recurrence 3.9.

Figure 3.4 illustrates a uniform pipeline for an affine dependency. Pipelining a broadcast involves two steps. First, we identify a vector $\mathbf{u}$ such that adding multiples of

it to $\mathbf{z_1}$ generates $\mathbf{z_2}, \mathbf{z_3}, \ldots$, i.e., all points that depend on $\mathbf{z_d} = f(\mathbf{z_1})$. Second, the pipeline must be initialized by a uniform dependence at its first point $\mathbf{z_1}$.

The key question is, how do we find such a pipeline for any affine dependency? We must be able to both generate a pipeline by finding a suitable constant vector $\mathbf{u}$, and initialize the pipeline by another constant vector $\mathbf{u'}$.

Lemma 3.6.1. *(Rajopadhye [123]) Two points $\mathbf{z_1}$ and $\mathbf{z_2}$ have an affine dependency $f(\mathbf{z}) = A\mathbf{z} + \mathbf{b}$ on the same point $\mathbf{z_d}$ if and only if they are separated by a constant vector $\mathbf{u}$ in the nullspace of A.*

Proof. Since $\mathbf{z_1}$ and $\mathbf{z_2}$ depend on the same point $\mathbf{z_d}$, $A\mathbf{z_1} + \mathbf{b} = \mathbf{z_d} = A\mathbf{z_2} + \mathbf{b}$. It follows that $A(\mathbf{z_1} - \mathbf{z_2}) = 0$ and hence $\mathbf{u} = \mathbf{z_1} - \mathbf{z_2}$ is in the nullspace of A.

Similarly, given a vector $\mathbf{u}$ in the nullspace of A, a point $\mathbf{z_1} = \mathbf{z_2} + \mathbf{u}$ has its affine dependency on the point $A\mathbf{z_1} + \mathbf{b}$, which is $A\mathbf{z_2} + A\mathbf{u} + \mathbf{b}$. Since $\mathbf{u}$ is in the nullspace of A, $A\mathbf{u} = 0$. Conclude that $A\mathbf{z_2} + \mathbf{b} = A\mathbf{z_1} + \mathbf{b}$. $\square$

We can use Lemma 3.6.1 to show that instead of broadcasting an affine data dependence to point $\mathbf{z_m}$, we can instead read the same data value from a pipeline variable through a uniform data transfer from point $\mathbf{z_m} + \mathbf{u}$.

Pipeline Generation. We can find a suitable constant vector $\mathbf{u}$ by finding a basis vector for the nullspace of the $n \times n$ matrix A (here we assume that the rank of the matrix is $n - 1$). The basis is not unique; one must be selected from a large space of possibilities. Furthermore, the selected basis vector must satisfy the causality constraint for the schedule, i.e., $\lambda \cdot \mathbf{u} < 0$.

Pipeline Initialization. In order to initialize the pipeline, the dependent value $\mathbf{z_d}$ must be made available to the first stage of the pipeline, i.e., the earliest scheduled point, identified by the predicate $\mathbf{z} = \bar{\mathbf{z}}$. It has been shown that this initialization is possible with a uniform dependency if $A^2 = A$; otherwise, we may be able to use multistage pipelining [123].

61

Control Pipelining

Recurrences with multiple disjoint definitions must be handled with care in order to synthesize efficient hardware. Consider Equation 3.9, which initializes the pipeline when the predicate $\mathbf{z} = \bar{\mathbf{z}}$ is satisfied. Recall from the example in Section 3.5 that such disjoint definitions are implemented in a PE with a multiplexer. But how do we evaluate a predicate and generate a signal to control the multiplexer?
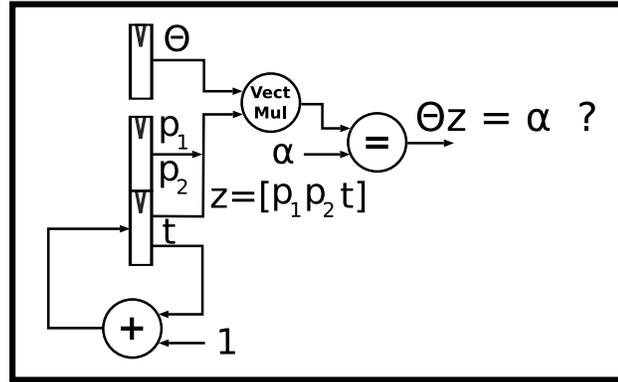
Figure 3.5: Evaluating a predicate requires maintaining the iteration vector $\mathbf{z}$ of the point being computed, and the use of arithmetic operations in every PE.

We can express such predicates as a conjunction of expressions of the form $\theta \mathbf{z} = \alpha$, where θ is a constant n-vector, $\mathbf{z} = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_{n-1} \\ t \end{bmatrix}$ is the iteration vector, and α is an integer. A naive implementation of this predicate in hardware is shown in Figure 3.5, with the output of the circuit controlling a multiplexer. However, implementing this logic in every PE is costly, wasting area resources and increasing complexity of the datapath.

Comparator hardware may be altogether eliminated from PEs by sharing control information, through pipelining, among all cells that check the same predicate. This pipeline simply transmits a single bit that is true whenever the predicate is satisfied. To find such a pipeline, consider all computation points satisfying $\theta \cdot \mathbf{z} = \alpha$. We first generate the pipeline by finding a basis $\mathbf{u}$ for the nullspace of θ that satisfies the causality constraint of the schedule. This pipeline is initialized at the boundary of the systolic array and is set to true whenever the predicate is satisfied. In this way, the expensive implementation of Figure 3.5 is required only at the boundary PEs.

3.7   Summary

In this chapter, we have described how any DP recurrence can be represented as a system of uniform recurrence equations and analyzed using the polyhedral model to build latency-space optimal arrays. In the next chapter, we show how to apply these methods to parallelize the Nussinov RNA folding algorithm.

Chapter 4

Accelerating the Nussinov RNA Folding Recurrence

In this chapter, we are interested in the problem of folding an RNA molecule to determine their secondary structure. RNA molecules carry out diverse functions in living cells, including catalysis of reactions [40], binding of small molecules [14], and targeted suppression of transcription and translation [64]. Although an RNA may be viewed as a linear sequence of characters, or *bases*, from the alphabet $\{A, C, G, U\}$, its function is actually determined by its *secondary structure*—the folded shape that results from pairing of complementary bases (mainly A-U and C-G) within one sequence. Determining this structure is key to analyses that identify and assign functions to RNAs.

Since experimental determination of an RNA's secondary structure is time-consuming, biologists use computer programs to predict the structure of RNA molecules. Efficient DP algorithms predict RNA structure using empirical models to estimate the free energies of folded structures. The simplest folding algorithm, due to Nussinov [117], uses the number of base pairs in the structure as a proxy for free energy, preferring structures with the most base pairs. This method was refined by Zuker [166] to predict a minimum free-energy structure using detailed and more accurate energy models. Figure 8.1 shows an RNA molecule folded into its two-dimensional structure.

RNA folding by any of the above methods requires time cubic in the length of the RNA. Although folding computations are generally restricted to RNAs of fewer than than 200 bases, important biological applications, including RNA motif finding [68,

Figure 4.1: An RNA sequence and its folded secondary structure with base pairs highlighted in blue.

119] and search for functional RNAs in genomic DNA [93, 114], may require folding millions of such short RNAs, which is computationally demanding.

We now introduce the simpler Nussinov RNA folding algorithm and develop parallel accelerators for this recurrence. We will tackle the more biologically accurate and challenging Zuker algorithm in Chapter 8. Accelerating Nussinov is important for two reasons. First, Nussinov has the same shape and dependency structure as Zuker. Accelerators for Nussinov represent the best-case scenario for RNA folding, allowing the exploration of a large design space that can inform the building of an efficient Zuker accelerator. Second, simplified versions of the Zuker algorithm that resemble Nussinov are used in applications that require interactive response [100, 105] or that use heuristics to predict secondary structure [12].

In the next section we introduce the Nussinov DP recurrence and describe the challenges to successful acceleration. We show how the recurrence may be finessed into a

form more suitable for efficient acceleration. The application of the localization transformation, which converts affine dependencies to uniform ones, is described in detail. Armed with the polyhedral analysis techniques introduced in the previous chapter, we develop two full-size, latency-optimal arrays. We have coded these arrays on an FPGA system and conclude with an experimental evaluation of the accelerators.

4.1   The Nussinov Algorithm

Given an RNA sequence S of length N, the Nussinov algorithm computes the largest number of base pairs $X(i,j)$ in any folded structure of subsequence $S_{i...j}$ as follows:

$$X(i,j) = \max \begin{cases} X(i+1,j) \\ X(i,j-1) \\ X(i+1,j-1) + \delta(S_i, S_j) \\ \max_{i<q<j} \left[X(i,q) + X(q+1,j) \right]. \end{cases} \tag{4.1}$$

The DP algorithm models four cases that build from smaller subproblems to compute the best structure of subsequence $S_{i...j}$. The cases, illustrated in Figure 4.2, cover all possible ways the RNA subsequence may fold. The DP algorithm selects the case with the maximum number of base pairs as the optimal folding of the subsequence.

As shown in the figure, we may either add an unpaired base S_i onto the best structure already computed for subsequence $S_{i+1...j}$ or add unpaired base S_j onto the best structure computed for subsequence $S_{i...j-1}$. Alternatively, in the third case, bases S_i and S_j are paired. The function δ evaluates to 1 if two bases are complementary (may be paired) or 0 otherwise. Finally, we may combine two optimal substructures of subsequences $S_{i...q}$ and $S_{q+1...j}$. The algorithm considers every way of forming the two substructures by varying the base position given by index q. By evaluating all possible ways of folding the subsequence $S_{i...j}$, the Nussinov algorithm is able to select the one with the maximum number of base pairs. We refer the interested reader to an article by Eddy [42] for more details.

The data variable X is defined over the domain $\mathcal{D} = \{\, i,j \mid 1 \leq i < j \leq N \,\}$, with the score of the best structure for the entire sequence at $X(1,N)$. In this work, we are
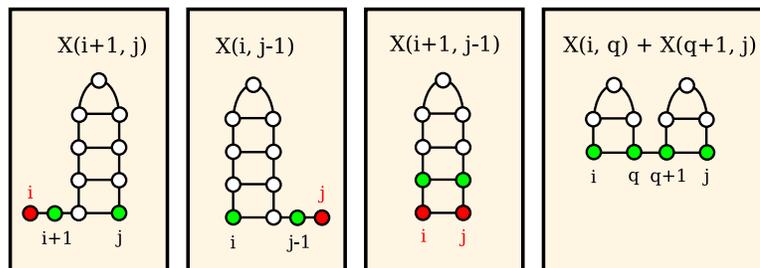
Figure 4.2: The Nussinov DP algorithm builds the score of the optimal structure for subsequence $S_{i\ldots j}$ by considering all four ways of breaking up the problem. The figure is taken from [42].

interested in accelerating only the computation of the value $X(i, j)$; the actual folded structures for sufficiently high-scoring RNAs may be computed later in software.
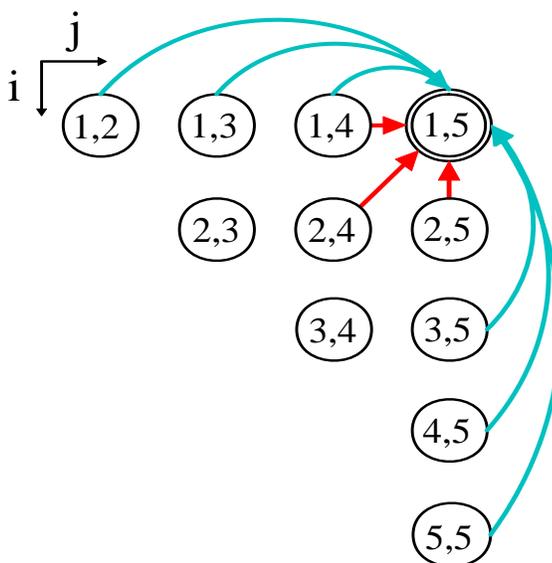
Figure 4.3: Nussinov data dependencies. $X(1, 5)$ depends on three adjacent cells (red) plus five non-local cells (blue).

The Nussinov recurrence is challenging to accelerate because of its non-local dependency structure, shown in Figure 4.3. Although the first three dependencies of $X(i, j)$ reference only adjacent cells of the DP matrix, the fourth term references widely separated cells. For example, $X(1, 5)$ depends on $X(1, 4)$, $X(2, 4)$, and $X(2, 5)$ but also on $X(1, 2)$, $X(1, 3)$, $X(3, 5)$, $X(4, 5)$, and $X(5, 5)$. On an FPGA, non-local cell dependencies result in routing delays that dominate the critical path of the computation. Worse yet, these non-local dependencies are *affine*; that is, $X(i, j)$ depends on other

cells $X(r, s)$ such that the differences $i - r$ or $j - s$ are not constant but rather depend on i and j. Affine dependencies result in a nonuniform structure that cannot easily be mapped onto an array of identically connected processing elements.

4.2   Parallelizing Nussinov

We now describe the use of polyhedral analysis to parallelize the Nussinov RNA folding algorithm. We first transform the recurrence described in Equation 4.1 to make it more amenable to parallelization. We then design full-size 2-D arrays using the parallelization techniques described in the previous chapter.

We start with Recurrence 4.1 and convert the $O(N)$-ary max operator to a binary operation by aggregating the operation along a new dimension k. We then use middle serialization [50] to derive the equivalent recurrence shown below. The transformations involved in serializing reductions are described elsewhere [62].

$$
X(i, j, k) = \max \begin{cases}
\delta(S_i, S_j) & \text{if } j - i = 1 \\
\\
X(i+1, j, k) \\
X(i, j-1, k) \\
X(i+1, j-1, k) + \delta(S_i, S_j) & \text{if } k = 1 \\
X(i, j, k+1) \\
X(i, i+k, 1) + X(i+k+1, j, 1) \\
X(i, j-k, 1) + X(j-k+1, j, 1) \\
\\
X(i, j, k+1) \\
X(i, i+k, 1) + X(i+k+1, j, 1) & \text{otherwise} \\
X(i, j-k, 1) + X(j-k+1, j, 1).
\end{cases}
\tag{4.2}
$$

Note how the fourth term in Recurrence 4.1 is aggregated along the newly introduced dimension k, where $1 \leq k \leq \lfloor (j - i)/2 \rfloor$, and the result of the max is available at $k = 1$. For an RNA of size N bases, the score of the best folded structure is now stored at $X(1, N, 1)$.

The terms $X(i, i+k, 1)$, $X(i+k+1, j, 1)$, $X(i, j-k, 1)$ and $X(j-k+1, j, 1)$ induce four affine dependencies of the form $f_i(\mathbf{z}) = A_i \mathbf{z} + \mathbf{b_i}$ as seen below, which are an obstacle for efficient array synthesis.

$$
A_1 = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \mathbf{b_1} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} ; A_2 = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \mathbf{b_2} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}
$$

$$
A_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 0 \end{bmatrix} \mathbf{b_3} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} ; A_4 = \begin{bmatrix} 0 & 1 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \mathbf{b_4} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}
$$

In the next section we make affine dependencies f_{1-4} uniform by introducing corresponding pipeline variables X_{1-4}. We also pipeline the input RNA sequence using variables P and Q.

4.2.1 Pipelining Affine Dependencies

We apply the localization technique introduced in the previous chapter by searching for basis vectors $\mathbf{u_{1-4}}$ for the nullspace of A_{1-4}. We first note that the rank of all four matrices is two; therefore, the nullspace in each case is one-dimensional and so is specified by a single basis vector. We derive the following basis vectors for the four pipelines: $\mathbf{u_1} = \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix}$, $\mathbf{u_2} = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}$, $\mathbf{u_3} = \begin{bmatrix} 0 \\ -1 \\ -1 \end{bmatrix}$, and $\mathbf{u_4} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$.

Next, we must initialize these pipelines. To see if initialization is possible with a uniform dependency, we check for the condition $A_i^2 = A_i$ ($i = 1 \ldots 4$). Indeed, this condition is satisfied for A_2 and A_3. Pipelines for these two dependencies may be initialized at the domain boundary $k = 1$ using constant initialization vectors $\mathbf{u_2'} = \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix}$ and $\mathbf{u_3'} = \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix}$. We can now construct two new uniform pipelines X_2 and X_3 that replace the affine dependency terms $X(i+k+1, j, 1)$ and $X(i, j-k, 1)$ respectively.

$$
X_2(i, j, k) = \begin{cases} X(i+2, j, k) & \text{if } k = 1 \\ X_2(i+1, j, k-1) & \text{otherwise} \end{cases} \tag{4.3}
$$

$$
X_3(i, j, k) = \begin{cases} X(i, j-1, k) & \text{if } k = 1 \\ X_3(i, j-1, k-1) & \text{otherwise} \end{cases} \tag{4.4}
$$
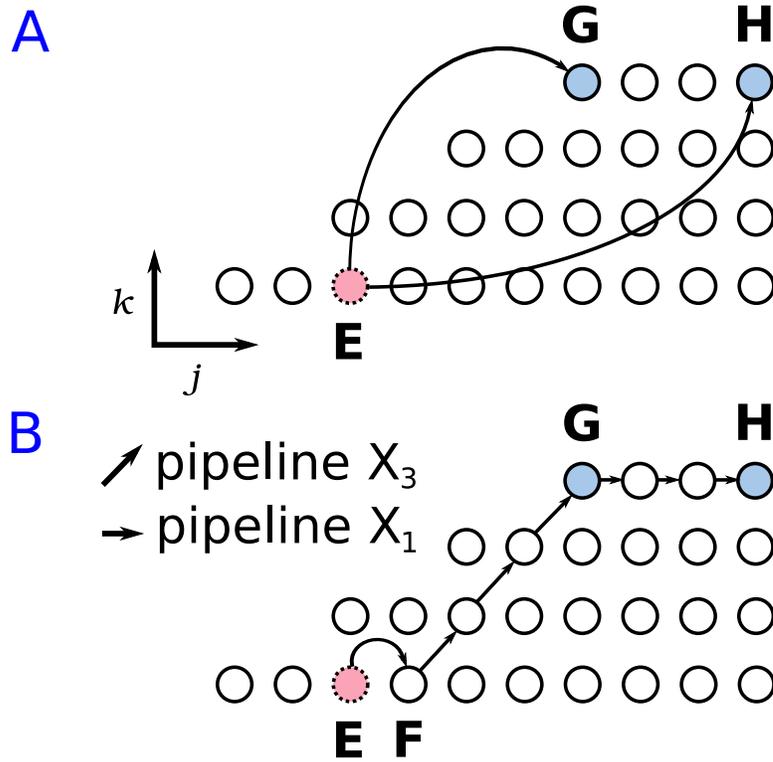
Figure 4.4: **(a)** Two affine dependencies $f_3 : \mathbf{G} \leftarrow \mathbf{E}$ and $f_1 : \mathbf{H} \leftarrow \mathbf{E}$ are to be pipelined. **(b)** Simple pipeline X_3 for dependency f_3 transfers $\mathbf{E}$ to all cells $\mathbf{F}$ through $\mathbf{G}$ that require it using uniform transfers. Multistage pipeline X_1 for dependency f_1 transfers $\mathbf{E}$ to all cells $\mathbf{G}$ through $\mathbf{H}$ and is initialized by X_3.

Broadcast dependencies f_1 and f_4 generate pipelines that must be initialized at the domain boundary $k = (j - i)/2$. However, neither pipeline can be initialized by a constant vector; initialization requires another affine dependence. Fortunately, we can pipeline this dependency too, using multistage pipelining. The pipeline for f_1 can be initialized using X_3, and the pipeline for f_4 using X_2. We can now construct two new uniform pipelines X_1 and X_4 that replace the affine dependency terms $X(i, i + k, 1)$ and $X(j - k + 1, j, 1)$ respectively.

Figure 4.4a shows an example of the two affine dependencies: $\mathbf{G} \leftarrow \mathbf{E}$ and $\mathbf{H} \leftarrow \mathbf{E}$, represented by functions f_3 and f_1 respectively. The dependencies are shown on the $j - k$ plane for a fixed value of i. Figure 4.4b shows their respective pipelines. The dependent cell $\mathbf{E}$ is pipelined through the set of cells that depend on it in a linear chain (shown by the straight line). Pipeline X_3 runs from $\mathbf{F}$ to $\mathbf{G}$ and is initialized with the value at cell $\mathbf{E}$ by a uniform dependency (shown by the arc). Pipeline X_1

running horizontally from **G** to **H** is a multistage pipeline and uses the last stage of X_3 for initialization.

Pipelining the Input Sequence

We now deal with the input sequence S used in the score computation of the recurrence. The input variables S_i and S_j are defined over one-dimensional domains but are used in computing the three-dimensional variable X. We therefore align S_i along the $j = 0 \wedge k = 1$ line and S_j along the $i = 0 \wedge k = 1$ line. The affine dependencies introduced are made uniform using nullspace pipelining. Their pipelines P and Q propagate horizontally and vertically across the array and are initialized by the input sequence at the domain boundary $j - i = 1$.

The final, transformed system of uniform recurrence equations is shown below:

$$X(i,j,k) = \max \begin{cases} \delta(P(i,j,k), Q(i,j,k)) & \text{if } j - i = 1 \\[2mm] X(i+1,j,k) \\ X(i,j-1,k) \\ X(i+1,j-1,k)+ \\ \quad \delta(P(i,j,k), Q(i,j,k)) & \text{if } k = 1 \\ X(i,j,k+1) \\ X_1(i,j,k) + X_2(i,j,k) \\ X_3(i,j,k) + X_4(i,j,k) \\[2mm] X(i,j,k+1), \\ X_1(i,j,k) + X_2(i,j,k) & \text{otherwise} \\ X_3(i,j,k) + X_4(i,j,k) \end{cases} \tag{4.5}$$

$$X_1(i,j,k) = \begin{cases} X_3(i,j,k) & \text{if } j - i = 2k \\ X_1(i,j-1,k) & \text{otherwise} \end{cases}$$

$$X_2(i,j,k) = \begin{cases} X(i+2,j,k) & \text{if } k = 1 \\ X_2(i+1,j,k-1) & \text{otherwise} \end{cases}$$

71

$$
X_3(i,j,k) = \begin{cases} X(i,j-1,k) & \text{if } k = 1 \\ X_3(i,j-1,k-1) & \text{otherwise} \end{cases}
$$

$$
X_4(i,j,k) = \begin{cases} X_2(i,j,k) & \text{if } j - i = 2k \\ X_4(i+1,j,k) & \text{otherwise} \end{cases}
$$

$$
P(i,j,k) = \begin{cases} S_i & \text{if } j - i = 1 \\ P(i,j-1,k) & \text{if } k = 1 \end{cases}
$$

$$
Q(i,j,k) = \begin{cases} S_j & \text{if } j - i = 1 \\ Q(i+1,j,k) & \text{if } k = 1. \end{cases}
$$

Unique dependencies of this SURE are as follows (each A_i is the identity matrix):

$$
\mathbf{b_1} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} ; \mathbf{b_2} = \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix} ; \mathbf{b_3} = \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix} ; \mathbf{b_4} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}
$$

$$
\mathbf{b_5} = \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix} ; \mathbf{b_6} = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} ; \mathbf{b_7} = \begin{bmatrix} 0 \\ -1 \\ -1 \end{bmatrix} .
$$

Control Pipelining

The two predicates in the Nussinov recurrence that must be pipelined are $j - i = 2k$ and $k = 1$. The predicate $j - i = 2k$ may use either $\begin{bmatrix} 0 \\ -2 \\ -1 \end{bmatrix}$ or $\begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}$ as valid basis vectors for its pipeline. We use the latter, which guarantees nearest-neighbor communication. Note that the predicate is always false at odd diagonals. At the even diagonal border $j - i = 2$, the control pipeline is initialized to 1 and then pipelined to subsequent even diagonals. The control pipeline is optimized out from odd diagonals in the array.

When aggregating along the k-axis, we need to substitute the term $X(i,j,k+1)$ by zero at $k = \lfloor (j-i)/2 \rfloor$. We use a pipeline similar to the one above, except that both odd and even diagonal pipelines are initialized to 1. For the predicate $k = 1$, we use a control pipeline with basis vector $\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$.
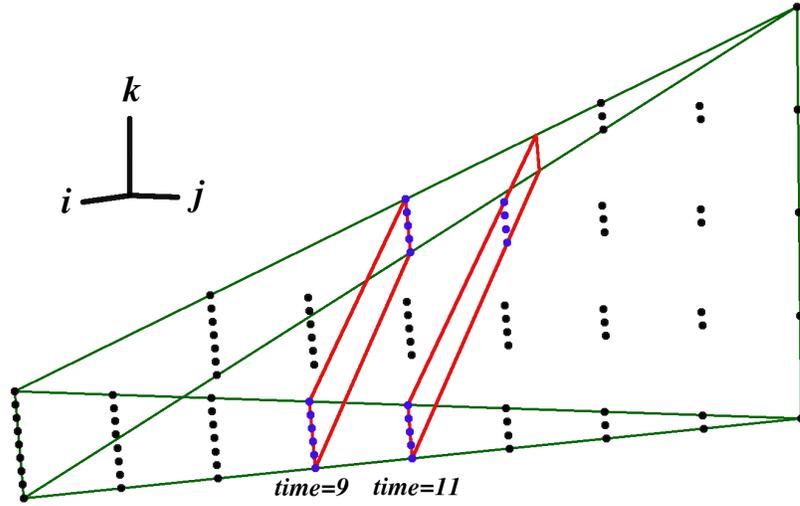
Figure 4.5: Points of the domain computed at time 9 and 11 are shown on hyperplanes (2-D planes).

4.2.2 Deriving Full-Size Arrays

A latency-optimal schedule for the Nussinov recurrence is given by $\tau(i, j, k) = -2i + 2j - k - 1$ where $\lambda = \begin{bmatrix} -2 \\ 2 \\ -1 \end{bmatrix}$ and $\alpha = -1$. A *time hyperplane*, in this case a 2-D plane, describes all points in the domain that are executed simultaneously on independent processors. Figure 4.5 shows points on two time hyperplanes at time 9 and 11. The total computation time for an RNA of length N is given by $\tau(1, N, 1) - \tau(1, 3, 1) + 1 = 2N - 5$ clock cycles.

Once a schedule has been found, the allocation function must be computed. Recall that an allocation can be described by an equivalent projection vector. The projection vector projects the domain of the recurrence onto processors. In the case of Nussinov, we project its 3-D domain along two directions as illustrated in Figure 4.6 to generate 2-D processor arrays FS-A and FS-B. The first array projection is along the direction $\begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}$ with allocation $\pi(i, j, k) = [i, j]$. The second, a space-optimal projection, is along the direction $\begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix}$ with allocation $\pi(i, j, k) = [j, k]$. Both arrays use the same latency-optimal schedule and have an execution time of $2N - 5$ clock cycles.

Array FS-A

A PE at location (i, j) in the array FS-A has an interconnection network computed by substituting the constant vectors, $\mathbf{b_i}$, of dependencies from Recurrence 4.5 in the processor allocation function $\pi(i, j, k) = [i, j]$. The latency of each link in the network is computed from the schedule as $\lambda \cdot \mathbf{b_i}$. The network derived from the seven uniform dependencies is shown below.

1.	$[i + 1, j]$	Latency 2
2.	$[i, j - 1]$	Latency 2
3.	$[i + 1, j - 1]$	Latency 4
4.	$[i, j]$	Latency 1
5.	$[i + 2, j]$	Latency 4
6.	$[i + 1, j]$	Latency 1
7.	$[i, j - 1]$	Latency 1

The control pipeline for predicate $j - i = 2k$ maps to the interconnection $\begin{bmatrix} i+1 \\ i-1 \end{bmatrix}$, which moves data diagonally across the array with latency 3. The pipeline for predicate $k = 1$ maps to $\begin{bmatrix} i+1 \\ j \end{bmatrix}$, which moves data vertically with latency 2.

To detect the final result $X(1, N, 1)$, we may either use a counter to count up to the execution time of the entire array or use control pipelines to detect the predicates $i = 1 \wedge j = N \wedge k = 1$. The first two predicates are always satisfied at PE $[1, N]$; for the predicate $k = 1$, we use the existing control pipeline.

The FS-A array requires a total of $\frac{N^2 - 3N + 2}{2}$ PEs to fold an RNA of length N.

Array FS-B

The structure of this array, shown in Figure 4.6b, derives from the allocation function $\pi(i, j, k) = [j, k]$. The control pipeline for predicate $j - i = 2k$ maps to the diagonal interconnection $\begin{bmatrix} i-1 \\ j-1 \end{bmatrix}$ and has a latency of 3. The predicate $k = 1$ is always true for all PEs at the bottom row $\begin{bmatrix} j \\ 1 \end{bmatrix}$.

The network derived from the seven uniform dependencies is as follows.

74

1.	$[i-1, j]$	Latency 2
2.	$[i-1, j]$	Latency 2
3.	$[i-2, j]$	Latency 4
4.	$[i, j+1]$	Latency 1
5.	$[i-2, j]$	Latency 4
6.	$[i-1, j-1]$	Latency 1
7.	$[i-1, j-1]$	Latency 1

In the FS-B array, the target value is computed by PE $[N, 1]$. We introduce a new control signal for the predicate $i = 1$, which is initialized at PE $[2, 1]$ and propagates
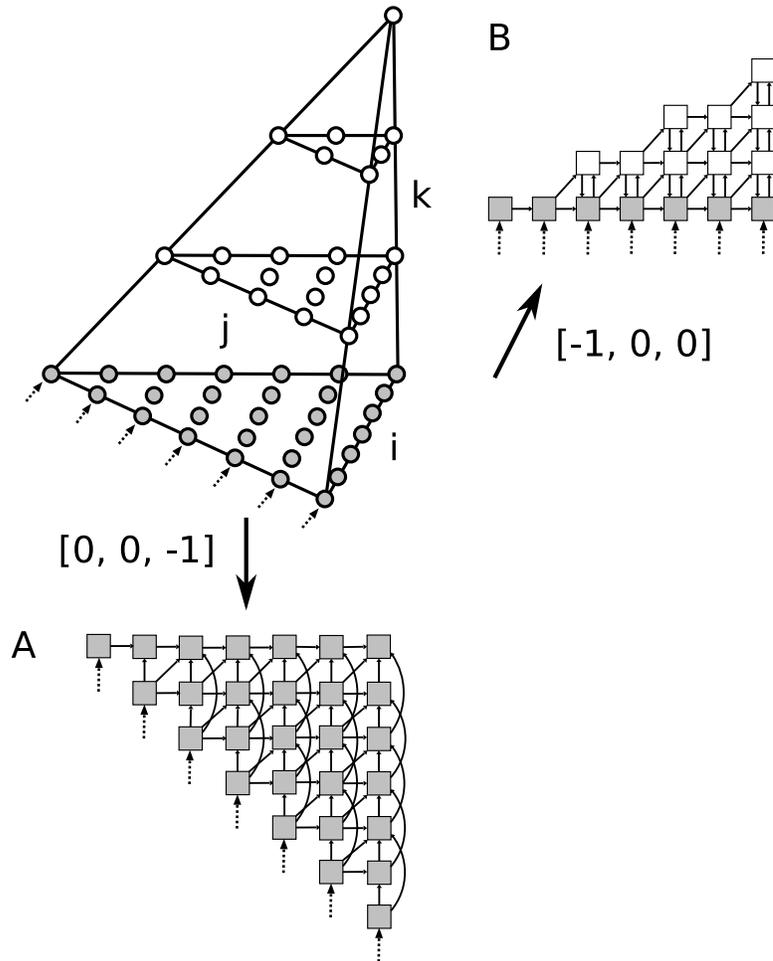
Figure 4.6: **(a)** Array FS-A projects the Nussinov computation domain along the k axis. **(b)** Projecting along the i axis yields the FS-B array. The RNA sequence is fed into PEs indicated by dashed arrows.

left to right with a latency of 2 through the bottom row of PEs. This signal is used to detect the result at PE $[N, 1]$.

The number of PEs required in the array to fold a sequence of length N is $\frac{N^2-N}{4}$, half the number required by the FS-A projection. Furthermore, just $N-1$ PEs in the FS-B array (shown shaded in Figure 4.6) implement the comparatively expensive computations at $k = 1$ of Equation 4.5, unlike array FS-A, which must implement this computation on every PE. Finally, if an implementation requires the cell values of the entire dynamic programming matrix (for example, to traceback the optimal secondary structure), the FS-A array requires I/O links at every PE. In contrast, the FS-B array generates these values at the bottom row of PEs.

4.3   PE Precision and Smaller Inputs

The width of datapaths in the arrays depends on the precision of the scores required in the RNA folding computations. Increased precision requires larger bit widths, which use more hardware resources and increase circuit complexity (i.e., critical path delay). We would like to use the lowest precision sufficient to produce correct results for the Nussinov algorithm.

We can use the fact that the Nussinov recurrence uses a simple scoring scheme of $+1$ for each base pair. For a sequence of length N, the maximum number of base pairs possible is $\frac{N}{2}$. We therefore limit the bit width of datapaths in PEs to $\log_2(\frac{N}{2} + 1)$.

All our arrays are designed for input sequences of length exactly N, determined at compile time. Given this length, the target score is always computed by the same PE. Input sequences of length $< N$ are handled as a special case. Suppose we have a sequence of length $m < N$ whose final score is computed at PE $\pi(1, m, 1)$. When calculating the score of a sequence of length $m + 1$, by the principle of optimality there exists at least one case in the dynamic programming recurrence that has a direct or indirect dependency on $X(1, m, 1)$. We can ensure that this value is always the optimal case by evaluating δ to zero whenever $i, j > m$. To achieve this, we pad the input sequence by $N - m$ special "unpairable" characters that yield a zero

score when paired with any other character. We can then show inductively that cell $X(1, N, 1)$'s value equals that of $X(1, m, 1)$.

4.4 Evaluation

In this section we evaluate the performance of our custom Nussinov arrays. We manually coded the two designs in VHDL, verified functional correctness in simulation, and then synthesized and verified correctness on our FPGA system. In this section we outline our results and compare performance of our accelerators to a software baseline. We start with a description of the software and hardware platforms used to compile these results. These two systems are employed throughout this dissertation.

4.4.1 Software Baseline

Our software baseline system is a 3 GHz Intel Core 2 Duo E6850 CPU with 4 MB L2 Cache and 4 GB system memory. The CPU was released in July 2007 and manufactured on a 65 nm process technology. We used Centos Linux version 2.6.23.1-42 as our operating system and compiled the baseline software with gcc version 4.4.0.

All runtimes are the wall clock time as reported by the unix *time* utility. We report both single core as well as dual core performance. To run our software on two cores, we split the input into two equal halves in an offline transaction and processed them using two independent instances of the software.

4.4.2 Hardware Baseline

To measure hardware performance we used an FPGA system provided by Exegy Inc[7]. Our designs were built on a Xilinx Virtex 4 LX100-12 FPGA, which has $49,152$ slices and 240 on-chip 18 kbit block RAM memories. Note that our FPGA was released earlier than our baseline system and uses an older process technology. The Virtex 4

---

[7]http://www.exegy.com/

family was released in June 2004 and manufactured on a 90 nm process technology. It has since been superseded by Virtex 5, 6 and more recently 7 FPGA families from Xilinx. We used the Virtex 4 because of easy access to this FPGA, however, we expect accelerator performance to improve substantially on newer generations of FPGAs.

4.4.3 Results

Table 4.1: Comparison of the two full-size Nussinov systolic arrays.

Design	# PEs	N	Total slices	Slice memories	Frequency (MHz)
FS-A	$\frac{N^2-3N+2}{2}$	41	42,137 (85%)	606 (2%)	130
FS-B	$\frac{N^2-N}{4}$	83	48,884 (99%)	706 (2%)	160

We synthesized our arrays post-place-and-route for the Xilinx Virtex 4 LX100-12 FPGA. Arrays FS-A and FS-B were built with a PE data width of 5 and 6 bits and clocked at 130 and 160 MHz respectively. We expected the FS-A array to clock faster due to its smaller data width but this advantage is negated by the expensive recurrence body computation that must be implemented in every PE of the array.

We built the two Nussinov arrays to process as large an RNA as possible. We report the FPGA resources used by each array in Table 4.4.3. The FS-A array can process a sequence of length at most 41, using 85% of slices on the FPGA. In contrast, the FS-B array can process sequences of length up to 83, 2× longer.

Table 4.2: Variation of array FS-B's resource usage with PE precision.

Precision	N	Total slices	Slice memories	Frequency (MHz)
6	83	48,884 (99%)	706 (2%)	160
8	65	45,923 (93%)	597 (2%)	140
16	47	46,158 (93%)	552 (2%)	130

We next built the FS-B array with various cell precisions as reported in Table 4.2. Increasing the data width noticeably increases area and so reduces the number of PEs that fit on the FPGA. Our design's critical path is also sensitive to the logic delays caused by the longer combinational path of the larger bit widths and so decreases the array clock frequency. The more complex Zuker RNA folding algorithm will need a precision of at least 16 bits, so similar full-size arrays are unlikely to fit on our FPGA.

To provide a baseline to compute the speedup of our accelerator, we wrote an optimized version of the Nussinov algorithm (Recurrence 4.1) in C. The software was compiled using gcc 4.4.0 with flags *-O3 -march=nocona -fomit-frame-pointer*. We created two synthetic RNA databases of 10 million randomly generated sequences of length 41 and 83 bases. The databases were of size 790 and $1,191$ MB respectively, and easily fit in system memory. We measured the time to compute the Nussinov scores of the two RNA databases on the dual core. We report the best of five runs with runtime including I/O operations such as the reading of sequences from disk into memory, although the operating system reads the database from cache after the first run.

To measure hardware performance, we folded the same two RNA databases on our FPGA system. We first verified correctness by matching the scores of each accelerator to that of the software implementation. The hardware runtime measures the time to read the database into system memory, transfer it onto the FPGA, perform the computation, and report the results to the user. Again, we report the best of five runs.

Table 4.3: Speedup of Nussinov arrays vs. modern x86 CPU. Runtimes are measured in seconds.

Design	N	Single core runtime	Dual core runtime	H/W runtime	Single core speedup	Dual core speedup
FS-A, FS-B	41	210.06	105.57	6.66	31.54×	15.85×
FS-B	83	1462.35	732.70	11.32	129.18×	64.73×

Table 4.3 shows the speedups of the Nussinov arrays for various sequence lengths. Both arrays have the same execution time of $2N - 5$ clocks, but FS-B is clocked 23% faster. FS-A can process a sequence of at most 41 bases on the Virtex 4 LX100-12 FPGA and achieves a speedup of 15× over the dual core. Our larger FS-B implementation is still faster than the dual core when processing 83 base RNAs.

Speedups increase significantly for arrays handling longer sequences because the array runs in time $\Theta(N)$ rather than the software's $\Theta(N^3)$. Larger families and newer FPGAs will support even longer sequences; for example, we built a FS-B array handling RNAs of length 100 on a Virtex-4 LX160 FPGA.

Our study demonstrates encouraging speedups on our accelerator system over general-purpose processors and provides a foundation for improvement in the rest of this dissertation. Note the importance of design space exploration. Clearly, we favor array FS-B for Nussinov RNA folding, which is more resource efficient and folds RNAs faster than FS-A. In the next chapter we explore how to find such favorable arrays.

4.5   Related Work

The FS-A and FS-B Nussinov arrays are similar to two classic 2-D systolic arrays for the well-known *optimal string parenthesization problem*. Given a string S of length N and a cost $W(i,j)$ for adding parentheses around substring $S_{i...j}$, this problem seeks the minimum cost $X(i,j)$ of any properly nested set of parentheses for $S_{i...j}$. Its recurrence mirrors the fourth case of the Nussinov algorithm: $X(i,j) = W(i,j) + \min_{i<q<j}[X(i,q) + X(q,j)]$.

A classic result of Guibas et al. [60] shows how to parallelize string parenthesization using a 2-D systolic array. Gachet et al. [50] later proposed an alternative space-efficient array. Arrays FS-A and FS-B are similar to these two well known designs. To our knowledge, Guibas' and Gachet's designs have neither been implemented in hardware nor adapted to perform RNA folding.

4.6   Conclusions

We have analyzed and accelerated Nussinov RNA folding by building two systolic array designs. While the core designs draw on classic results for string parenthesization, we have elaborated these designs into fully realized FPGA implementations and analyzed their performance. Our designs achieve up to a $64\times$ speedup over a modern x86-family dual core CPU when implemented on a Virtex 4 LX100-12 FPGA. Our realization of the Nussinov algorithm is an important starting point for designs that more closely approximate the detailed energy models used by the Zuker algorithm.

Chapter 5

Design Space Exploration of Throughput-optimized Arrays

In the previous chapter, we successfully parallelized the Nussinov RNA folding algorithm, building efficient hardware arrays using polyhedral analysis. However, our attempt to apply state-of-the-art techniques to our target application domain exposed two key limitations, which we seek to remedy in this chapter.

First, most techniques to realize systolic arrays from a recurrence seek an array that is latency-space optimal. Such an array computes a single instance of the recurrence with minimum latency; for example, our Nussinov arrays from the previous chapter can fold a single RNA in the shortest time possible. In many 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 its *throughput*, or equivalently the total execution time on the entire data set. As far as we are aware, the literature of automated systolic array design places little emphasis on optimizing arrays for throughput. We will address the problem of throughput-optimized array design in this chapter.

Second, we seek not just a single optimal array but a *library* of designs optimized for different-sized inputs. A large data set may contain inputs of many different sizes, but a systolic array implementation is of one fixed size, requiring different-sized inputs to be split or padded. A single array is reasonable when the target platform is a

fixed-function integrated circuit, but modern devices can be rapidly reconfigured with designs optimized for different input sizes. FPGAs from Xilinx and Altera support the loading of a new hardware circuit in tens to hundreds of milliseconds [23]. It is therefore feasible for one computation to use a range of array mappings, each optimized to maximize throughput on inputs of a different size, using all available computational resources. For example, we may use a collection of throughput-optimized arrays to speed up the acceleration of RNA folding of a large database of sequences. When the input size distribution is known *a priori*, we 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 just a single latency-space optimal array supporting the largest possible size. The effectiveness of this approach will depend on the number of arrays used, the overhead of loading each array, and the processing time on each array; we explore this optimization problem in detail in Chapter 7. The software tool we introduce here can automatically design a range of arrays that trade off area for throughput.

In this chapter, 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 *allocation function*, independent of the *schedule* of times at which these steps are executed. This observation leads to a search strategy for finding arrays with optimal throughput. We then describe a software tool we have written to accept recurrence descriptions of programs and perform the aforementioned search. The schedules and allocations generated by our tool can be fed into array synthesis software such as MMAlpha [61], PARO [63], or PICO-NPA [133] to synthesize low-level HDL descriptions of systolic arrays.

Finally, we present novel throughput-optimized arrays that have a 4-13× and 2-5× speedup over the latency-optimized array for the banded Smith-Waterman and Nussinov RNA folding algorithms respectively, when processing a large database of inputs.

5.1 Recapitulation of Background

In this section, we briefly recapitulate the background introduced in Section 3.3. Recall that we seek to accelerate a system of parametrized uniform recurrence equations. A parametrized recurrence defines the computation of an n-dimensional data variable $X(\mathbf{z})$ over a domain $\mathcal{D} \subset \mathbb{Z}^n$, representing the individual steps of the recurrence. The recurrence defines a domain of points at which computation is performed. We identify a point in the domain by its iteration vector $\mathbf{z}$, whose elements are loop index values. In what follows, we will use points and iteration vectors interchangeably.

A recurrence may be parametrized, i.e., it may have one or more *size parameters*, such as the length of an input sequence. The data dependencies between iteration vectors 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*, $\tau(\mathbf{z}) = \lambda \cdot \mathbf{z} + \alpha$, which gives the time when $X(\mathbf{z})$ is computed for each iteration vector $\mathbf{z} \in \mathcal{D}$.

2. An *allocation function*, $\pi(\mathbf{z})$, which gives the physical PE of the array that computes each $X(\mathbf{z})$. The allocation function can be fully specified by a direction vector $\mathbf{u}$ along which the computation domain is to be projected.

A systolic array has a *utilization*, or *efficiency*, given by $\frac{1}{|\lambda \cdot \mathbf{u}|}$ [96]. Given the product $\gamma = |\lambda \cdot \mathbf{u}|$, which is the initiation interval of computation points in the PE, each PE in the array is active one of every γ clock cycles. Idle PEs are a poor use of compute resources, so we wish to utilize each PE as much as possible.

There are two approaches to increasing the efficiency of underutilized arrays. If throughput is most important, γ instances of the input problem can be interleaved to execute simultaneously on the array. Alternatively, if resource usage is a constraint, the workload of γ virtual PEs may be run sequentially on a single physically realized PE—this is the approach taken in building a space-optimal array.

5.2 Motivating Examples

Throughout this chapter we will reference two recurrences to illustrate the concepts and methods introduced. Here we show only the domains of the recurrences; the interested reader will find the entire algorithms in the referenced works.

Although the computations in these recurrences' bodies are fairly simple integer operations, our technique applies independent of the operation complexity. Using floating point operators, multi-cycle operations, and on-chip memories does not invalidate the throughput optimization procedure we will describe (though it will reduce the size of the array that can be instantiated). Another important advantage to our method is that it is indifferent to dependencies in the recurrence; the throughput optimization procedure we suggest in this chapter depends solely on the shape and size of the computation domain.

Banded Smith-Waterman. This version of the Smith-Waterman algorithm for DNA and protein sequence alignment was described by Chao [28]. 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 a correspondence between the two sequences that minimizes the (weighted) edit distance between them. The computation domain 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. We have already introduced the Nussinov algorithm in Chapter 4. Its computation domain is $\mathcal{D} = \{\ i, j, k \mid 1 \leq i \leq N;\ i \leq j \leq N;\ 1 \leq k \leq \frac{j-i}{2}\ \}$.

We say that a systolic array executes a (domain) *instance* $\mathcal{I}$ when the array evaluates the system of recurrence equations for a particular input. This input could be, e.g., a certain DNA sequence. The input defines integer values for any size parameters of the recurrence.

We illustrate this concept with the Nussinov example. Let q_1 and q_2 be sequences of length $N = 100$ and $N = 200$ respectively. The Nussinov computation on the two sequences defines distinct instances $\mathcal{I}_1$ and $\mathcal{I}_2$ respectively. The computation domain

for q_1 is $\mathcal{D}(\mathcal{I}_1) = \{ \ i, j, k \ | \ 1 \leq i \leq 100; \ i \leq j \leq 100; \ 1 \leq k \leq \frac{j-i}{2} \ \}$, whereas the domain for q_2, $\mathcal{D}(\mathcal{I}_2)$, has an upper bound of 200 on i and j. Every input sequence defines a new instance of the recurrence that is to be executed on a systolic array; however, the domains for all sequences of the same length N have the same shape and contain the same number of iteration vectors.

5.3 Characterizing Throughput-optimized Arrays

The computation time for an instance $\mathcal{I}$, i.e. the time to evaluate all iteration vectors 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 the first and last scheduled iteration vectors:

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

As we have expounded in Chapter 3, existing work in array design has focused on finding a latency-optimal schedule $\mathcal{L}_{opt}$ 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 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 an array starts to compute on an input at which computation on a second input can be started safely, i.e., without any one PE trying to compute on both inputs at the same time. The reciprocal of β is the throughput of the array, measured in input instances completed per clock.

Consider the recurrence with computation domain $\mathcal{D}(\mathcal{I})$ on input instance $\mathcal{I}$ of some fixed size. Let $\tau(\mathbf{z}) = \lambda \cdot \mathbf{z} + \alpha$, $\mathbf{z} \in \mathcal{D}(\mathcal{I})$ 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 PE contention, we define a pipelined schedule of the kth instance as $f_k(\mathbf{z}) = \tau(\mathbf{z}) + (k-1)\beta$. A throughput-optimized array executes m instances of the input in time $(m-1)\beta + \mathcal{L}$, which is
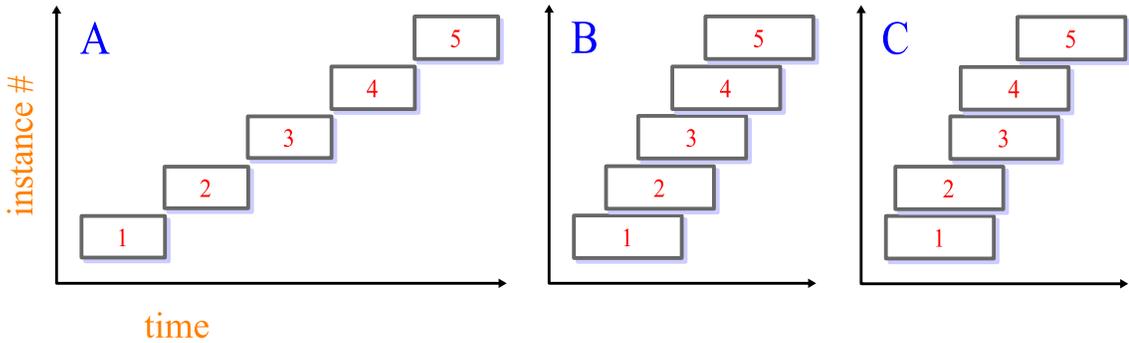
Figure 5.1: **(a)** A latency-optimal array executes five input instances in sequence and has an execution time of $5\mathcal{L}_{opt}$ clock cycles. **(b)** The array optimized for throughput pipelines a new input every $\frac{1}{3}\mathcal{L}$ clock cycles. **(c)** When the array is not fully efficient (here 50%), we simultaneously pipeline two input instances every $\frac{2}{3}\mathcal{L}$ clock cycles. The total execution time for arrays **b** and **c** are $\frac{7}{3}\mathcal{L}$ clocks. Though the throughput-optimized arrays have higher latency, they have an overall lower execution time than the latency-optimal array.

faster than the latency-optimal array if $\beta < \mathcal{L}_{opt}$ and the latency $\mathcal{L}$ of the throughput-optimized array is not much larger than $\mathcal{L}_{opt}$. In optimizing for throughput, we may have to sacrifice latency, but the higher latency can be amortized over a large number of inputs. The advantage of throughput-optimized arrays is illustrated in Figure 5.1.

Factors contributing to throughput. Before we proceed further, we clarify our notion of throughput as it applies to fixed-size arrays. There are several ways to increase the throughput of a systolic array on an FPGA. When selecting a schedule, one may minimize the intercommunication delay, which acts as a proxy for throughput. The schedule of (recurrence body) operations in a PE can be optimized to improve the clock frequency of the implementation [128]. High-level transformations on uniform recurrences can also pipeline the computation in a PE [37].

In contrast, our method minimizes the block pipelining period between input instances executed on the array. We are able to aggressively trade off area for increased throughput and to suggest far more solutions than the other techniques. Furthermore, our design procedure does not preclude optimization of the schedule in each

PE, which can be performed in a secondary step. See Section 5.8 for more information on related throughput optimization techniques.

5.3.1   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., $\mathcal{D}(\mathcal{I}) = \mathcal{D}(\mathcal{I}')$, so their domains contain the same number of iteration vectors: $\mathbf{z} \in \mathcal{D}(\mathcal{I}) \Leftrightarrow \mathbf{z} \in \mathcal{D}(\mathcal{I}')$.

We would like to derive a schedule to execute both instances one after the other in a pipelined fashion. Let the first instance $\mathcal{I}$ execute on the schedule $\tau(\mathbf{z})$, $\mathbf{z} \in \mathcal{D}(I)$, derived as in Section 5.1. We use a pipelined linear schedule f for $\mathcal{I}'$ given by $f(\mathbf{z}') = \tau(\mathbf{z}') + \beta$, where $\mathbf{z}' \in \mathcal{D}(\mathcal{I}')$. If the computation on the two instances is not to overlap execution on the array, the two schedules must satisfy the following feasibility constraint:

$$\forall \mathbf{z} \in \mathcal{D}(I) \text{ and } \mathbf{z}' \in \mathcal{D}(\mathcal{I}'), \ f(\mathbf{z}') > \tau(\mathbf{z}). \tag{5.1}$$

Because $\mathcal{D}(\mathcal{I}) = \mathcal{D}(\mathcal{I}')$, we can express the same constraint as

$$\forall \mathbf{z_1}, \mathbf{z_2} \in \mathcal{D}(I) \text{ with } \pi(\mathbf{z_1}) = \pi(\mathbf{z_2}), \ f(\mathbf{z_1}) > \tau(\mathbf{z_2}). \tag{5.2}$$

Intuitively, f is a feasible schedule only if no two iteration vectors from the two instances are assigned for execution on the same PE at the same time. Importantly, this constraint allows for simultaneous execution of two iteration vectors $\mathbf{z_1}, \mathbf{z_2}$ from two independent instances respectively, if they execute on different PEs, i.e., $\pi(\mathbf{z_1}) \neq \pi(\mathbf{z_2})$.

We clarify this concept using the example in Figure 5.2. Figure 5.2a shows a triangular domain with its points represented by circles. The indices of the domain points at the boundary are shown for clarity. Figure 5.2b shows two instances $\mathcal{I}$ and $\mathcal{I}'$ of this domain being pipelined through the array (the time dimension is along the x-axis). We can pipeline a new instance every $\beta = 3$ clock cycles.
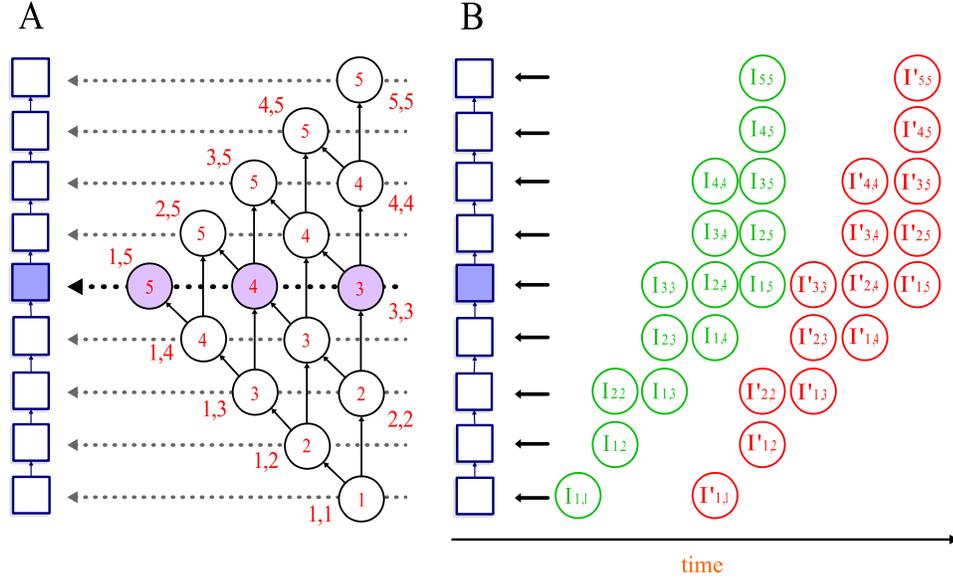
Figure 5.2: Pipelining the execution of two input instances on an array to improve throughput. **(a)** The triangular domain is projected horizontally along the dashed lines onto a linear array of PEs. The execution time of every domain point is shown in the circle. The shaded PE executes the maximum of $k_{max} = 3$ domain points. **(b)** We can therefore pipeline two instances $\mathcal{I}$ and $\mathcal{I}'$ of the same size with a pipelining period $\beta = 3$ and we are guaranteed that there will be no PE contention.

The key question is, how small can we make β while satisfying constraint 5.2, i.e., how quickly can we start the next computation after the previous one has started? The answer clearly depends on the array mapping, and one would expect the minimal β to be a function of both the array's projection vector $\mathbf{u}$ and its schedule λ. However, we now prove that the throughput of the array can be determined *independent of the schedule* λ. 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. Any two iteration vectors $\mathbf{z_1}$ and $\mathbf{z_2}$ in $\mathcal{D}(\mathcal{I})$ are executed by the same PE if and only if

$$\mathbf{z_1} - \mathbf{z_2} = k\mathbf{u}; \; k \in \mathbb{Z}.$$

88

We denote the maximum number of iteration vectors executed by any PE in one domain instance $\mathcal{I}$ as $k_{max} = 1 + k_{ilp}$, where

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

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

$$
\begin{aligned}
\text{Maximize} \qquad & k & \text{(5.3)}\\
\mathbf{z_1} - \mathbf{z_2} \ &=\ k\mathbf{u}\\
C\mathbf{z_1} \ &\leq\ \mathbf{d}\\
C\mathbf{z_2} \ &\leq\ \mathbf{d}.
\end{aligned}
$$

The following theorem bounds the value of β for which $f(\mathbf{z})$ forms a feasible pipelined schedule.

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

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

We therefore have

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

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

89

For the if part, assume to the contrary that $f(\mathbf{z_1}) \leq \tau(\mathbf{z_2})$ for some $\mathbf{z_1}, \mathbf{z_2} \in \mathcal{D}(\mathcal{I})$. Since $\pi(\mathbf{z_1}) = \pi(\mathbf{z_2})$, we have $\mathbf{z_2} - \mathbf{z_1} = k\mathbf{u}$. WLOG, we assume that k is a positive integer. Then we have

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

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

Corollary 5.3.1. *Given multiple instances of the recurrence, all of the same size, a valid schedule for the mth instance is given by $f_m(\mathbf{z}) = \tau(\mathbf{z}) + (m-1)\beta$.*

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 5.2, the shaded PE computes the largest number of domain points $k_{max} = 3$. The array being fully efficient, $|\lambda \cdot \mathbf{u}| = 1$, and we can pipeline a new input instance every three clock cycles.

5.3.2 Implications for Design-Space Exploration

We have shown in Theorem 1 that the block pipelining period must be greater than $k_{ilp}|\lambda \cdot \mathbf{u}|$. The product $|\lambda \cdot \mathbf{u}|$ is the reciprocal of array utilization (see Section 5.1); confronted with an array that is not fully efficient, we can increase its throughput by computing $|\lambda \cdot \mathbf{u}|$ input instances simultaneously (assuming I/O bandwidth is not a limitation). The throughput achievable on such an array is therefore $\frac{|\lambda \cdot \mathbf{u}|}{k_{ilp}|\lambda \cdot \mathbf{u}|+1}$, which is one input instance every $k_{ilp} + \left\lceil \frac{1}{|\lambda \cdot \mathbf{u}|} \right\rceil \, (= k_{max})$ clock cycles. Thus the throughput is exactly $\frac{1}{k_{max}}$ input instances per clock cycle. We conclude that *for any schedule, regardless of the utilization, the throughput achievable on an array with projection vector u is always one instance per k_{max} clock cycles.*

Knowing that the throughput of a systolic array is independent of its schedule greatly simplifies our exploration of array design space. We can search over possible projection vectors to find one with optimized throughput and only then derive its schedule, rather than having to consider both allocation and schedule simultaneously.

5.4   Finding Throughput-Optimized Projection Vectors

We now describe a procedure for searching the space of projection vectors $\mathbf{u}$ for a given set of recurrences to discover a collection of feasible, high-throughput array mappings. 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 $\mathbf{u}$ decreases the number of domain points executed by each PE in the array. This frees up PEs sooner for the next input and so improves throughput. In theory, for any input size, we can increase the magnitude of $\mathbf{u}$ without bound until each PE handles only one point of the domain, resulting in the best possible throughput of one new instance every cycle. In practice, however, such a design is likely infeasible—it will require more PEs than fit on the FPGA, and the high rate of new instances may require more input bandwidth than is available. We must search for smaller-magnitude projection vectors that are feasible on the target FPGA device.

It has been shown that there is no closed form function that can represent the number of processors instantiated by a projection vector [160] (for 3-D and higher dimension recurrences); hence, an enumerative search for suitable projection vectors is required. The space of 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. Because it is computationally infeasible to enumerate and test every projection vector, we need a strategy to limit the number of vectors to be tested in order to find the best choices.

Fortunately, to limit the number of distinct vectors $\mathbf{u}$ that must be tested, we can derive and apply *a priori* bounds on the magnitude $|\mathbf{u}|$ based on our target platform's resource constraints. These bounds come from two sources: FPGA area constraints and total system input bandwidth. Next, we derive these bounds using mathematical tools first used by Wong and Delosme [161] for space optimization.

5.4.1   A Search Procedure for Projection Vectors

For convenience, suppose that the input domain's 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 an array is *feasible* if it satisfies bandwidth and area constraints on the target platform. We first compute a bound B such that no array mapping with $|\mathbf{u}| > B$ is feasible for inputs of size $\geq N_0$ (or possibly $\leq N_0$ for bandwidth-limited recurrences). We then compute throughputs for all $\mathbf{u}$ with $|\mathbf{u}| \leq B$; in general, these throughputs are functions of N. The steps of this bounded search are outlined in Algorithm 1.

When many different vectors yield the same throughput, we compute array mappings (schedules and allocations) for each and keep only the array with the smallest area usage as estimated by the number of PEs and then the highest utilization. The algorithm finally returns a set $\mathcal{C}$ of array mappings, one per distinct throughput. All solutions are parametrized by N; we only use a specific value N_0 in the final sorting step.

The algorithm guarantees that every array that satisfies the area and bandwidth constraints is explored, though it may explore more mappings than is strictly necessary. The collection $\mathcal{C}$ of array mappings obtained from Algorithm 1 (using the area constraint) has the property that, for any input size $N \geq N_0$, one of the arrays in $\mathcal{C}$ that is feasible for input size N has the highest throughput among all feasible arrays of input size N. For $N' < N_0$, the array mappings in $\mathcal{C}$ are feasible for input size N' but are not necessarily throughput-optimal. N_0 is therefore the lower bound for the input size. Decreasing it proves throughput optimality for a larger number of parameter values but also increases computational expense; hence, a tradeoff has to be made.

Algorithm 1 Explore allocation/schedule space
---
  procedure EXPLORE(Recurrence, Parameter Values)
    $bound \leftarrow min(BandwidthBound, ResourceBound)$

    for each projection vector $\mathbf{u}$ within bounds **do**
      $T \leftarrow$ Throughput($\mathbf{u}$)
      $\pi \leftarrow$ Allocation($\mathbf{u}$)   ▷ *π is nullspace basis of* $\mathbf{u}$
      $\tau \leftarrow$ Schedule($\mathbf{u}$)
      $\gamma \leftarrow$ Utilization(τ, $\mathbf{u}$)
      $\#PE \leftarrow$ NumPEs(π, τ)   ▷ *Count number of PEs*
    end for

    Sort the solutions by T, $\#PE$, γ
    Select the best array for every distinct T for $\mathcal{C}$
  end procedure

Given inputs of fixed size N, we may simply choose the highest-throughput array mapping for size N from $\mathcal{C}$. If, however, the input stream contains inputs of varying sizes, we can switch among these arrays by reconfiguration to process the collection as efficiently as possible, achieving better overall throughput than any single array. Chapter 7 illustrates one practical reconfiguration strategy and its application to Nussinov RNA folding.

5.4.2   Basic Definitions for Bounds

We now derive the bounds used by Algorithm 1. We start by introducing some fundamental definitions for convex bodies. A detailed explanation is available in standard texts [132].

Every system of recurrence equations defines a closed convex body $\mathcal{D}$. The *width* of $\mathcal{D}$ in the direction $\hat{\mathbf{s}}$ is given by $w(\hat{\mathbf{s}}) = \{\ \mathbf{z_1} \cdot \hat{\mathbf{s}} - \mathbf{z_2} \cdot \hat{\mathbf{s}}\ |\ \mathbf{z_1}, \mathbf{z_2} \in \mathcal{D}\ \}$, where $\hat{\mathbf{s}}$ is a unit vector of dimension n. Intuitively, the width is the distance between the two support planes of $\mathcal{D}$ orthogonal to $\hat{\mathbf{s}}$ and $-\hat{\mathbf{s}}$ (see Figure 5.3).

We now present three propositions that we will use in deriving our bounds. We refer the reader to Wong and Delosme [161] for their proofs.
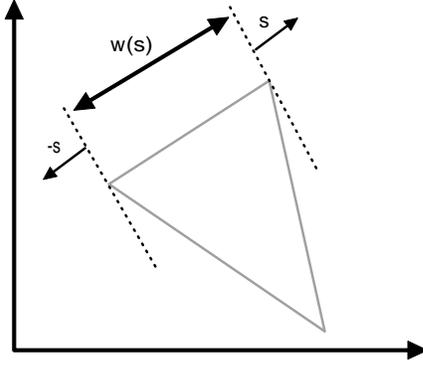
Figure 5.3: Width of a convex body along the direction $\hat{\mathbf{s}}$. The number of domain points along the unit vector $\hat{\mathbf{s}}$ is $1 + w(\hat{\mathbf{s}})$.

Proposition 1. *Given a projection direction $\mathbf{u}$ (not necessarily a unit vector), the maximum number of iteration vectors executed on any PE, k_{max}, is at most*

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

Proposition 2. *An upper bound on $w(\mathbf{u})$ can be shown to be*

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

where $\mathbf{e_i}$ is the ith canonical row vector whose i^{th} element is one and where other elements are zero; $w(\mathbf{e_i})$ is the width of the domain along the ith dimension.

Proposition 3. *If the projection of $\mathcal{D}$ by a vector $\mathbf{u}$ is $\mathcal{D}_\mathbf{u}$, the number of domain points in $\mathcal{D}_\mathbf{u}$ is bounded from below by*

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

Finally, we note an assumption used for convenience in the next two sections. Because array mappings that assign only one iteration vector per PE are not generally feasible for realistic input sizes, we will assume that each PE must execute at least two such iteration vectors. This assumption is not critical to the derivations but simplifies the forms of the final bounds.

94

Input Bandwidth Bound

We first 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 $|\mathbf{u}|$.

We assume that the final clock period of the array is known. For a systolic array this is determined by the logic in a PE. We can use one of many heuristic techniques [89] or simply synthesize a single PE to estimate its period. We then compute the normalized 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, these requirements are 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

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

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 $\mathbf{u}$ such that only one iteration vector is executed on every PE—there is no need for a search.

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

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

From our assumption that each processor executes at least two iteration vectors, $k_{max} \geq 2$, and so

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

Simplifying, we have

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

Using Proposition 2, we substitute for $w(\mathbf{u})$ to get

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

which simplifies to

$$|\mathbf{u}| \leq \frac{2m}{b} \sqrt{w(\mathbf{e_1})^2 + w(\mathbf{e_2})^2 + \ldots + w(\mathbf{e_n})^2}. \tag{5.4}$$

FPGA Resource Bound

In the case of an FPGA target, we are uniquely able to exploit the resource constraint, i.e., the number of PEs 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 PEs—there is no penalty for using as many of them as required.

The regular nature of systolic arrays enables us to predict their resource usage accurately. The primary contribution to area is the logic and on-chip memories used within a PE, which can be estimated from the body of a recurrence. Handling expressions predicated by conditional statements in a recurrence is challenging, since, depending on the projection direction, some PEs may not need to instantiate the logic and so will use fewer resources. We may overcome this problem either by taking a conservative approach, overestimating the number of PEs that can be instantiated, or by computing the size of an "average" PE based on the sizes of disjoint subdomains.

Existing work [89] can estimate 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 such estimation tools, so we predict the maximum number of PEs that can be placed on an FPGA device using actual synthesis results. This is done by building a single PE using the Synplify synthesis tool (execution time is on the order of minutes).

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

$$|\mathcal{D}_{\mathbf{u}}| \leq p.$$

Substituting the lower bound in Proposition 3, we get

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

Using the two-iteration-vectors-per-PE constraint as in the previous section, we obtain

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

Substituting Proposition 2 for $w(\mathbf{u})$ and simplifying, we get the bound

$$|\mathbf{u}| \leq \frac{2p}{|\mathcal{D}|}\sqrt{w(\mathbf{e_1})^2 + w(\mathbf{e_2})^2 + \ldots + w(\mathbf{e_n})^2}. \tag{5.5}$$

Note the similarity between the bandwidth and area bounds.

5.4.3   Search Complexity and Application to Examples

Our search considers every projection vector $\mathbf{u}$ 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 $O(2^n|\mathbf{u}|^n)$. 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 $\mathbf{u}$ would be roughly 40 for a two-dimensional recurrence, or 15 for a three-dimensional recurrence.

Input Bandwidth Bound.   We first compute the bandwidth bound for the 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.

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 that the query is fixed, and that each input instance is a new target sequence. The width of the domain along the canonical row vectors is $N - 1$; for $N = 300$ and 500, the bound is $|\mathbf{u}| \leq 37$. If instead the array is used to compare Unicode text, which requires 16 bits per character, the greater input bandwidth imposes a smaller bound of $|\mathbf{u}| \leq 12$ for strings 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 widths of the domain along the three canonical row vectors are $N - 3$, $N - 3$, and $\frac{N-3}{2}$. Bounds for $N = 25$ and 60 are $|\mathbf{u}| \leq 57$ and $|\mathbf{u}| \leq 61$, resulting in a far more expensive search than the previous case. This is due to the fact that with just three bits per input character, the algorithm is far from being bandwidth-constrained on our hardware platform. As expected, the bandwidth bound is useful for restricting the search space principally when the recurrence has high data requirements, or the bandwidth into the FPGA system is limited.

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 $N = 25$ and 60, the area bounds are $|\mathbf{u}| \leq 90$ and $|\mathbf{u}| \leq 16$, respectively. The latter constrains the search to the range of a few minutes that we stipulated earlier. With a 32-bit datapath, which requires substantially more resources per PE, we get a prediction of 327 PEs and bounds of $|\mathbf{u}| \leq 18$ and $|\mathbf{u}| \leq 5$ for the two lengths.

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 and 500 characters, which are reasonable for typical protein sequences. In each case, the bound derived is $|\mathbf{u}| \leq 22$. In a practical implementation, banded Smith-Waterman is likely to be just one of many hardware stages on the same FPGA and will have even fewer available resources, leading to an even better bound.

5.5 Selecting Schedules 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* [97]. As shown in Figure 5.4, 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.

As mentioned in Section 3.6.3, we use the Vertex method to find feasible schedules, which uses an integer linear program with constraints to satisfy causality. To enable retiming, we modify the schedule selection procedure to provide "new" delay registers (shown shaded in Figure 5.4) for the retimer to move into each PE. Specifically, we relax the schedule generated for an array to introduce longer delays on dependency links into a PE.

Recall from Section 3.6.3 that we would like to solve for the schedule $\tau(\mathbf{z}) = \lambda \cdot \mathbf{z} + \alpha$ such that for every uniform dependency $f_i(\mathbf{z}) = \mathbf{z} + \mathbf{b_i}$ we have a linear (causality)
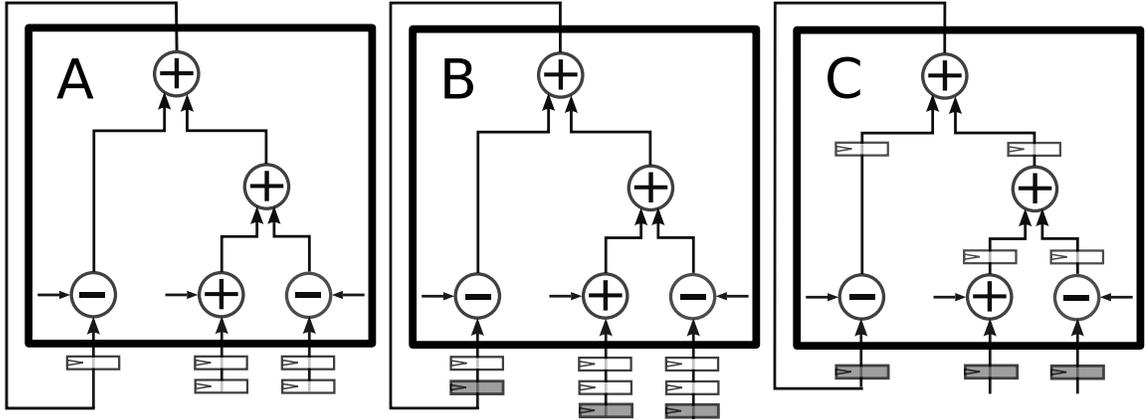
Figure 5.4: **(a)** The unpipelined PE has three operations in its critical path. **(b)** When the schedule is relaxed, new delay registers are generated on dependency links, which are moved by the circuit retimer as shown in **(c)** to reduce the length of the critical path to one operation.

constraint $\lambda \cdot \mathbf{b_i} < 0$ in the integer linear program. Suppose we would like to pipeline the PE by s stages. We modify the causality constraint to be $\lambda \cdot \mathbf{b_i} \leq -s$. Any schedule that satisfies this constraint is still feasible, for it satisfies the dependencies, but is also 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. where $\gamma = |\lambda \cdot \mathbf{u}| > 1$), 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.

For completeness, we describe the complete integer linear program we have used. The objective is to minimize the function $W\gamma + \mathcal{L}$, where γ and $\mathcal{L}$ are the efficiency and latency of the array, and W is an appropriately selected integer (we use 2048) that prioritizes utilization. We have also used the vertices $v \in \mathcal{V}$ of the computation

domain to formulate this linear optimization problem.

$$\min \ (W\gamma + \mathcal{L}) \tag{5.6}$$

$$
\begin{aligned}
\max_{v',v \ \in \ \mathcal{V}} \ \lambda(v - v') &\leq \ \mathcal{L} \\
|\lambda \cdot \mathbf{u}| &\leq \ \gamma \\
\lambda \cdot \mathbf{u} &\neq \ 0 \\
\lambda \cdot \mathbf{b_j} &\leq \ -s
\end{aligned}
$$

5.6 Software Tool

We have written a design space exploration tool in C++ implementing the ideas described in this chapter. This tool is intended to be a plugin to an automatic parallelization package such as MMAlpha [61], 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 in its dependencies, vertices, and the polyhedral domain of computation from text files.

We used the Polyhedral library [3] for polyhedral manipulations, the PIP library [2] for solving integer linear programs, and the Barvinok library [153] for counting the number of integral points in a polyhedron.

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

5.7 Results

Tables 5.1 and 5.2 show some of the projection vectors suggested by our software on the two example recurrences. The six columns from left to right are the projection vector, maximum number of points executed by any PE, number of PEs 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 the target FPGA.

The pipelining period can be computed as one plus the product of $k_{max} - 1$ and γ. In both examples, the latency-space optimal array is shown in the first row.

Novel throughput-optimized arrays. For every projection vector we are able to find a schedule that maximizes utilization, with at most 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, but so too do the links' delays, which automatically pipelines communication.

Banded Smith-Waterman shows a number of attractive arrays with throughputs equal to fractions of the width of the band. There have been numerous hardware arrays suggested to accelerate the Smith-Waterman computation, all of which use the latency-space optimal array with projection vector $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$. Protein sequences have lengths of 300-500 characters but can be several times longer, allowing us to use a combination of the suggested throughput-optimized arrays for banded Smith-Waterman on a reconfigurable target. Our alternate designs are predicted to execute 4-13$\times$ faster than the latency-optimized array.

Table 5.1: Throughput-area tradeoff for banded Smith-Waterman. Bounding was based on sequences of length $N_0 = 300$ and a band width of $w = 66$, forcing $|\mathbf{u}| \leq 22$. The throughput of an array is given by $\frac{1}{\beta}$ where $\beta = 1 + (k_{max} - 1)\gamma$.

u	k_{max}	#PEs	γ	**Latency**	**Max.** N
1 1	N_0	66	2	600	∞
1 0	66	300	1	600	480
1 -1	33	599	1	899	240
2 -1	22	898	1	600	160

Table 5.2: Throughput-area tradeoff for the Nussinov recurrence. Bounding was based on $N_0 = 61$, forcing $|\mathbf{u}| \leq 16$.

	u	k_{max}	#PEs	γ	$\mathcal{L}$	**Max.** N	**Speedup**
A	-1 0 0	$N_0 - 2$	900	2	117	82	1.0
B	1 1 0	$N_0 - 2$	900	1	175	82	2.0
C	0 0 -1	$\frac{N_0-1}{2}$	1770	1	117	59	3.7
D	1 2 0	$\frac{N_0-1}{2}$	1770	1	172	-	-
E	1 1 -1	$\frac{N_0}{3}$	2611	1	117	49	4.9
F	2 2 -1	$\frac{N_0+1}{4}$	3423	1	117	-	-

The latency-space optimal array FS-B derived in Chapter 4 for the Nussinov algorithm has projection direction $\begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix}$ and is shown in row A of Table 5.2. The array has a utilization of only 50%, so standard practice is to cluster two adjacent PEs, decreasing the space cost to 450 in the example. Unfortunately, this clustered array has a block pipelining period equal to its latency ($2N - 5$ clocks) and does not optimize throughput.

The table also includes array FS-A with projection direction $\begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}$ in row C, which is the second latency-optimal array derived in Chapter 4. State-of-the-art techniques allow us to pipeline a new RNA only every $2N - 5$ clocks. However, we see by Theorem 1 that a new RNA can be pipelined every $\frac{N-1}{2}$ clocks on the same array without processor contention, providing $4\times$ speedup. In fact, our tool discovers another array in row D that has the same throughput, but fewer PEs implementing resource-intensive calculations at domain points with $k = 1$.

Our search tool discovers novel arrays that realize still greater throughput and have better I/O properties. An alternate array, which we call FS-C, with projection direction $\begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$ has the same area requirement of FS-B but is fully efficient. Latency of this array increases $1.5\times$ but it is easily amortized. Furthermore, as shown in Figure 5.5, sequence data in array FS-C 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. Our tool finds many other projections that trade area for higher throughput.

Figure 5.5 shows how the original computation domain of Nussinov can be projected in three different directions to derive three full-size arrays. As a postscript to this discussion, we add that all these array mappings are also applicable to the string parenthesization problem, because of the similarity of its computation domain to that of Nussinov.

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$\times$ 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.

For comparison, we built FS-C, our newly discovered throughput optimized array,
to fold RNAs of 83 bases. We were able to route the design at 166 MHz. We then
folded 10 million randomly generated sequences and measured runtime. Recall from
Section 4.4.3 that the latency-space optimal array FS-B is 64.73× faster than our
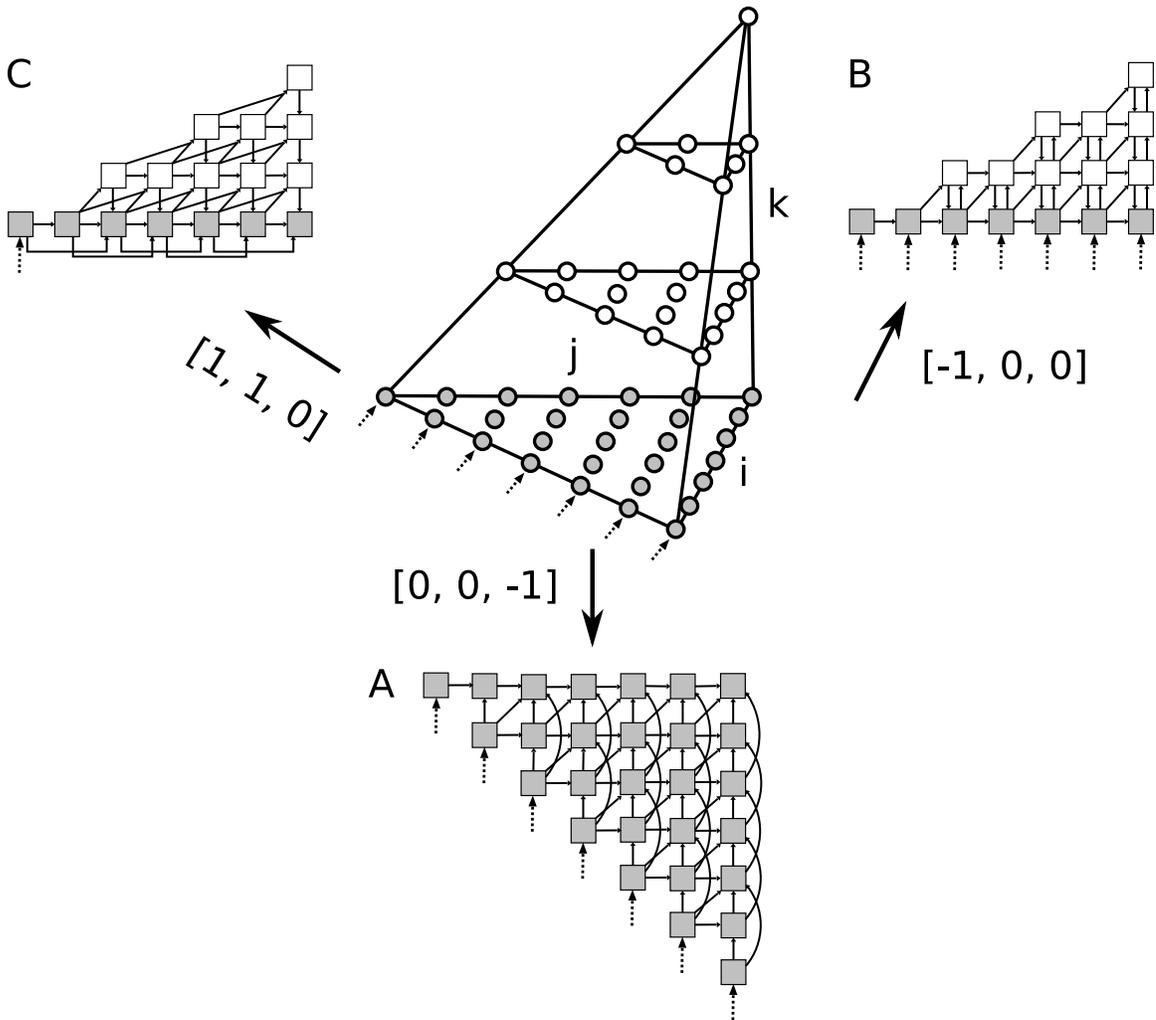
Figure 5.5: **(a)** Array FS-A projects the Nussinov computation domain along direc-
tion [0, 0, -1]. **(b)** Projecting along direction [-1, 0, 0] yields the FS-B array. **(c)**
Projecting along the diagonal direction [1, 1, 0] yields the FS-C array. The RNA
sequence is fed into PEs indicated by dashed arrows.

dual core software baseline. Array FS-C folds all 10 million RNAs in 5.60 seconds and is 130.84× faster than two cores.

Our bounds guarantee that all projections that satisfy the bandwidth and resource constraints are explored. As can be seen from Tables 5.1 and 5.2, the number of PEs required by some of the projections are larger than that supported by the FPGA, so we do explore a larger space than is strictly necessary for an input size of $N_0 = 61$. These alternate projections, however, can still be used for smaller input sequences; the predicted maximum RNA size that can be executed on each array is shown in Table 5.2 (this is an optimistic number). In any case, the execution time of the design space exploration tool was within the stipulated range of seconds to minutes. Searches for the Smith-Waterman recurrence ran in under 5 seconds and Nussinov in under 4 minutes on an Intel Core 2 Duo workstation. In these two cases, 36 and 7117 candidate projection vectors were explored.

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.

As expected, an increase in throughput causes a proportional increase in the number of PEs. One might be tempted simply to use multiple instantiations of the latency-space optimal array on the FPGA device. In the Smith-Waterman example, we may use five parallel units of array $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$ to achieve the same throughput as array $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$. However, this approach does not scale with the number of parallel instantiations. Having y parallel units increases the number of PEs while simultaneously increasing I/O by a factor of y. In the Nussinov example, each array has N PEs that read the input sequence. Furthermore, each unit requires independent I/O buffers or a bus with potentially long delays. Using a single high-throughput array does not cause a similar increase in I/O to PEs, and each new problem instance can be loaded sequentially in time on the same data path.
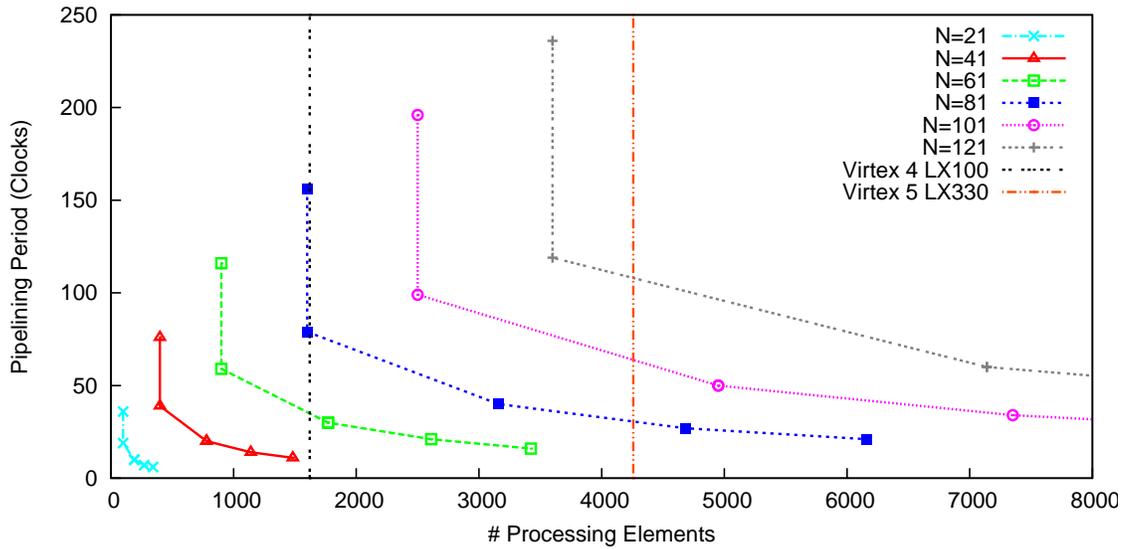
Figure 5.6: The graph compares the pipelining period and the number of PEs required for arrays A-E from Table 5.2 for various values of N.

Area versus throughput tradeoff. To understand how a user might select an appropriate array mapping for a particular FPGA device, in Figure 5.6 we graph the pipelining period and the number of PEs required for five mappings from Table 5.2. We draw multiple curves, each for a distinct value of N, the maximum length of an RNA that may be folded on the array. Also shown are two vertical lines representing two modern Xilinx FPGA devices. All points on the graph to the left of one of these two lines represents an array mapping (of a fixed N) that is predicted to fit on the particular FPGA device.

For a given N, the user identifies the corresponding curve and selects the array with the lowest pipelining period that is predicted to fit on the target FPGA device. As N varies (specifically for small values of N), it may be possible to use high throughput arrays that consume a large number of resources. We will explore this idea in more detail in Chapter 7.

High clock speed designs. Our tool ensures that the novel arrays (B-F in Table 5.2) 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

106
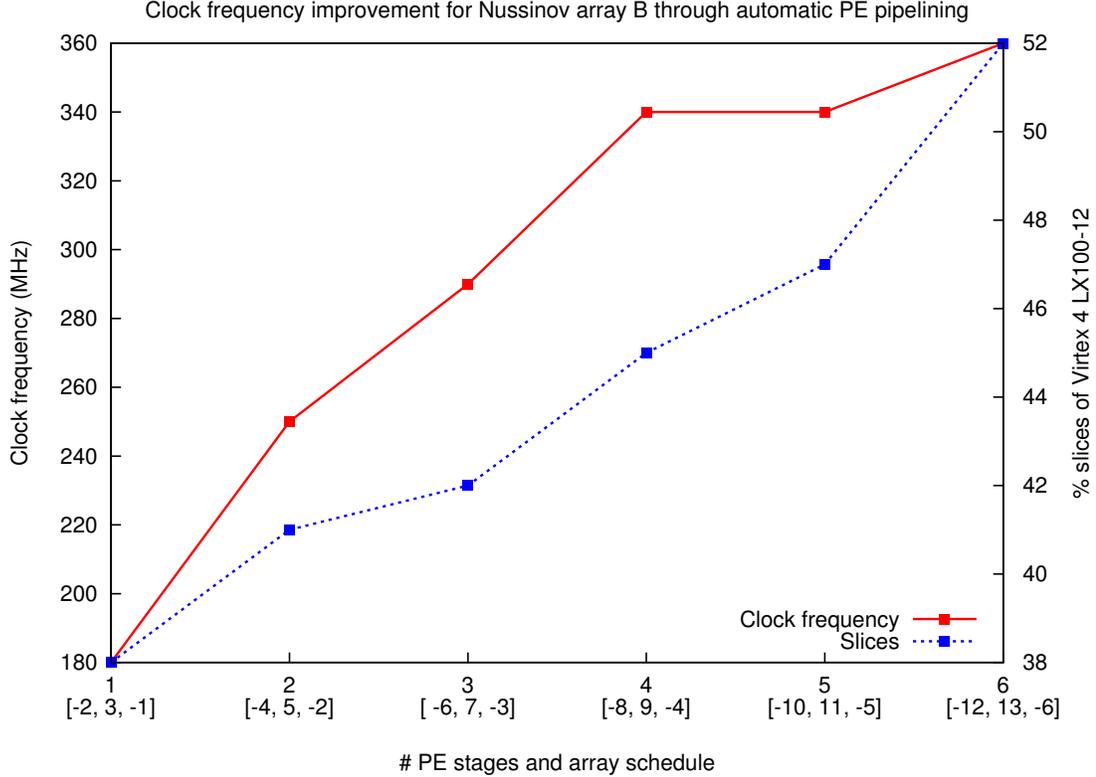
Figure 5.7: Improvement of array clock frequency as a function of pipelined stages in a Nussinov PE.

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 5.7 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, 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.

5.8   Related Work

Decades of research have gone into the automatic synthesis of systolic arrays from systems of recurrence equations using the polyhedral model. As we have mentioned, the practice has been to find a single latency-space optimal array. All these methods generate an $(n-1)$-dimensional array with unidimensional time from a recurrence of n dimensions.

Wong et al. [161] detail an enumerative search technique to minimize the number of PEs in an allocation, which is employed by PARO [63]. 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 with a local interconnection network [165]. 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. However, for many recurrences, including Nussinov, we wish to allow a small number of long-range links on our FPGA device. Making such a compromise greatly increases the search space.

There are two major differences between our work and related work: we optimize for throughput, and we identify multiple arrays that trade throughput for FPGA resources. The only work we are aware of that optimizes throughput using space-time methods is by Rosseel et al. [128]. The authors target real-time signal processing

applications in video, speech, and image processing that must optimize throughput rather than latency. The goal is to find a single array that matches the requested throughput (for some fixed recurrence parameter values) and minimizes the area requirement. They perform this search using simple enumeration.

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 $k_{ilp}|\lambda \cdot \mathbf{u}|$ through a multi-phase interleaved search through the allocation and schedule spaces. Using the newly introduced techniques in this chapter, we are able to directly estimate the block pipelining period using only the projection vector, allowing us to search the two spaces independently. Of the allocation candidates that match the requested throughput, Rosseel et al. select ones that increase average usage of the PEs. 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 not find many interesting space mappings.

Rosseel's approach gives priority to optimizing the schedule of operations in a PE by minimizing the area-operation interval product. This step can be integrated into our method to optimize the PE design. Rosseel's method attempts to avoid generating low utilization PEs since they must be clustered and could increase area requirements; we can instead simply pipeline multiple input instances.

A technique analogous to ours is software pipelining, used to simultaneously execute successive iterations of an inner loop on a VLIW machine. The goal is to fully utilize all the functional units available on the VLIW machine by exploiting parallelism in loop iterations. Lam [94] 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 resources (availability of functional units) and the inter-iteration dependencies. Lam suggests an iterative heuristic that searches multiple schedules, while optimizing the initiation interval between a lower and an upper bound.

In contrast, we pipeline iterations of multiple loops, which execute on independent input instances. As a result, an advantage of our approach is that we do not have to deal with dependence constraints, which requires the iterative search used by Lam. Iterations of a single loop instance are scheduled by the linear function τ, which

accounts for inter-iteration dependencies. An advantage of using custom hardware is that we can simply replicate functional units (as PEs) to avoid the resource constraint. Lam's technique executes only iterations of the innermost loop in parallel, while we can handle nested loops. Overall, we have more freedom to explore a larger space of parallel solutions.

5.9   Conclusion

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

Chapter 6

Resource-limited Array Mappings

Thus far we have explored the design of latency- and throughput-optimized arrays from recurrence equations. Both these approaches attempt to exploit as much parallelism as is available in a dynamic programming algorithm. In this chapter we deal with the case where parallelism is sacrificed in order to meet resource constraints.

There are two situations where exploiting all available parallelism is not the best approach. Latency- and throughput-optimized full-size array mappings generate an $(n-1)$-dimensional array from an n-dimensional recurrence. When a recurrence has a high dimension ($n > 3$), full-size arrays cannot be realized on modern FPGAs, since arrays of dimension three and higher cannot easily be synthesized on a 2-D substrate. Moreover, even for three-dimensional recurrences, such as Nussinov, we may prefer a 1-D rather than a 2-D array in order to conserve resources. In this section, we describe an existing technique that produces 1-D or 2-D mappings from recurrences of arbitrary dimension.

Even with a one- or two-dimensional mapping, a generated array may require more PEs than can fit on a given target FPGA. This may happen for a large computation domain or for recurrences that use resource-intensive operations and lookup tables and hence consume increased area per PE. For such arrays, we would like to trade parallelism for decreased resource usage by *partitioning* the 1-D or 2-D array onto a smaller subset of PEs.

In what follows, we apply existing techniques for resource-constrained array synthesis within the polyhedral framework to parallelize dynamic programming recurrences. We first describe a technique to generate 1-D and 2-D arrays. Next, we show how to

build partitioned 1-D and 2-D arrays. In this chapter, we use these techniques to build several novel resource-constrained arrays for the Nussinov RNA folding recurrence.

6.1   Generating 1-D and 2-D Arrays

We now outline the formal technique used to find a resource-constrained array mapping for a recurrence equation. We use an existing technique due to Darte et al. [34]. In this section we only state their results, giving an intuition for how the method works. Our application to the Nussinov RNA folding problem will help the reader understand how this method works. For complete details on the theory of this technique, we refer the interested reader to the paper.

Suppose we have a recurrence that defines the rectangular computation domain $\mathcal{D} = \{ i_1, \ldots, i_n \mid 1 \leq i_1 \leq S_1; \ldots ; 1 \leq i_n \leq S_n \}$. When creating a 1-D array, we select any one of the dimensions, say i_1, as the processor index to get the allocation function $\pi(i_1, \ldots, i_n) = [i_1]$. We place a PE at every point $1 \leq i_1 \leq S_1$ to derive a S_1-PE array. We assign a distinct rectangular parallelipiped of points for serial execution by every PE in the array. For example, a PE at $i_1 = c$ serially executes the rectangular parallelepiped of iteration vectors in the domain $\mathcal{D} = \{ i_1, \ldots, i_n \mid i_1 = c; 1 \leq i_2 \leq S_2; \ldots ; 1 \leq i_n \leq S_n \}$.

The challenge is to find a schedule that satisfies the causality constraint on the dependencies and also serially executes all points in a PE's parallelepiped without conflict, i.e., no two points must be scheduled on the PE at the same time. Darte et al. [34] gave a constructive procedure to create such schedules. For the 1-D array placed along dimension i_1, the schedule is of the form (up to a permutation of the iteration vector indices)

$$\tau(i_1, \ldots, i_n) = a_1 i_1 + a_2 i_2 + a_3 S_2 i_3 + \ldots + a_n S_2 \cdots S_{n-1} i_n, \tag{6.1}$$

where $a_n = \pm 1$ and the greatest common divisor of a_k and S_k is 1. The schedule is essentially enumerating all points in the rectangular parallelepiped corresponding to a PE: first, along index i_2, then i_3, and so on.

Similarly, for a 2-D array we select any two dimensions, say i_1 and i_2, as the processor indices and use the schedule (up to a permutation of the iteration vector indices)

$$\tau(i_1, \ldots, i_n) = a_1 i_1 + a_2 i_2 + a_3 i_3 + a_4 S_3 i_4 + \ldots + a_n S_3 \cdots S_{n-1} i_n, \qquad (6.2)$$

where $a_n = \pm 1$ and the greatest common divisor of a_k and S_k is 1. Values for a_i are selected to satisfy the causality constraint and minimize the latency for computing a recurrence.

The procedure we have outlined assumes that the DP recurrence has a computation domain that is a rectangular parallelepiped, i.e., each dimension of the domain has constant upper and lower bounds. For non-rectangular domains, such as the triangular Nussinov RNA folding domain, we expand them by creating a rectangular bounding box and assigning ∞ to data variables when computation is outside the triangle. The new domain for Nussinov RNA folding is therefore $\mathcal{D} = \{\, i_1, \ldots, i_n \mid 1 \leq i \leq N; 1 \leq j \leq N; 1 \leq k \leq \frac{N}{2} \,\}$. Figure 6.1 illustrates the original Nussinov domain in Figure 5.5 transformed to a rectangular parallelepiped, with newly introduced computation points shown as dashed circles. This transformation does not change the computed scores, but our array mappings now perform more work than is strictly necessary.

6.1.1   A 1-D Array for Nussinov

We now build a 1-D array for the Nussinov computation that may be used to fold long RNA sequences. We place $\frac{N}{2}$ PEs on the k axis, one on each point of the dimension $1 \leq k \leq \frac{N}{2}$. The advantages of this 1-D array mapping are two-fold: first, the compute-intensive operations on the plane $k = 1$ are restricted to a single PE; placing the array on any other dimension requires replication of these operations on every PE. Equally important, the selected allocation allows us to fold an RNA of length N using just $\frac{N}{2}$ PEs.

The 1-D array mapping for the system of recurrence equations 4.5 is given by:

$$\begin{aligned}
\tau(i, j, k) &= -k - 2i + Nj \\
\pi(i, j, k) &= [k].
\end{aligned} \qquad (6.3)$$

113
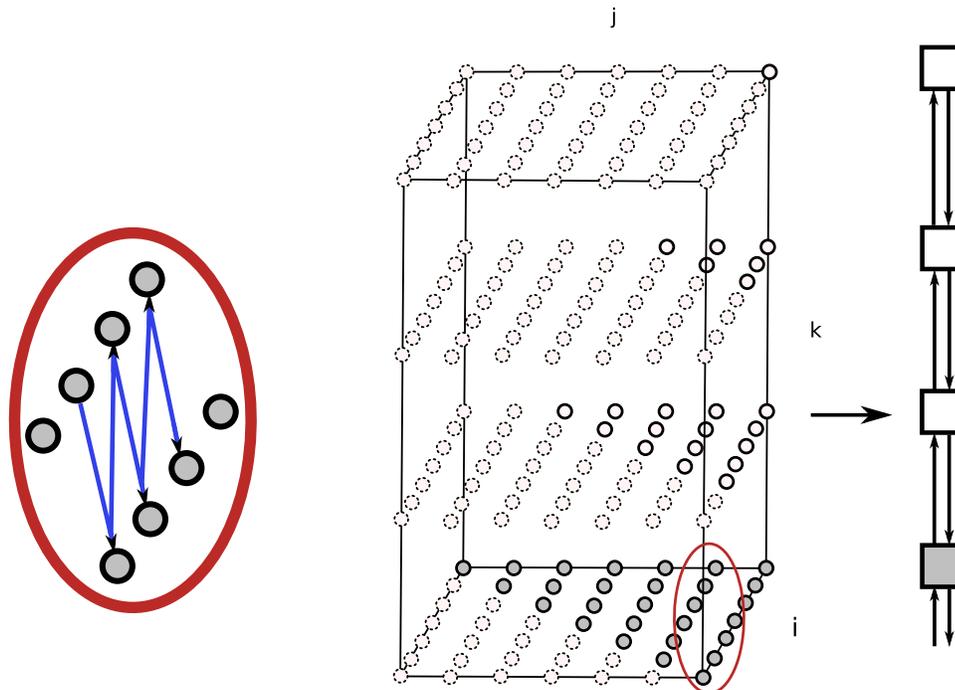
Figure 6.1: The 1-D array for the transformed Nussinov recurrence is generated by placing a linear array of PEs along the k axis. Computation points outside the original Nussinov domain are shown as dashed circles. Each PE in the array sequentially executes a rectangular plane of points as shown in the inset.

This array is illustrated in Figure 6.1. The schedule conforms to the format described in Equation 6.1, with $a_1 = -1, a_2 = -2$, and $a_3 = 1$. The schedule satisfies all the dependence constraints of the recurrence and is conflict-free if the greatest common divisor of N and 2 is 1, i.e., N must be odd. The latency of execution of an RNA of length N is $\tau(1, N, 1) - \tau(N, 1, 1) = N^2 + N - 2$ clocks.

6.2 Partitioned Arrays

In the previous section, we described a procedure to map recurrences of high dimension onto one- or two-dimensional arrays. We assumed that one PE is instantiated for every integral value of the processor dimension. This is a major limitation on resource-constrained FPGAs. In this section we describe the Locally Sequential, Globally Parallel (LSGP) method [36] to partition the processor index, which allows
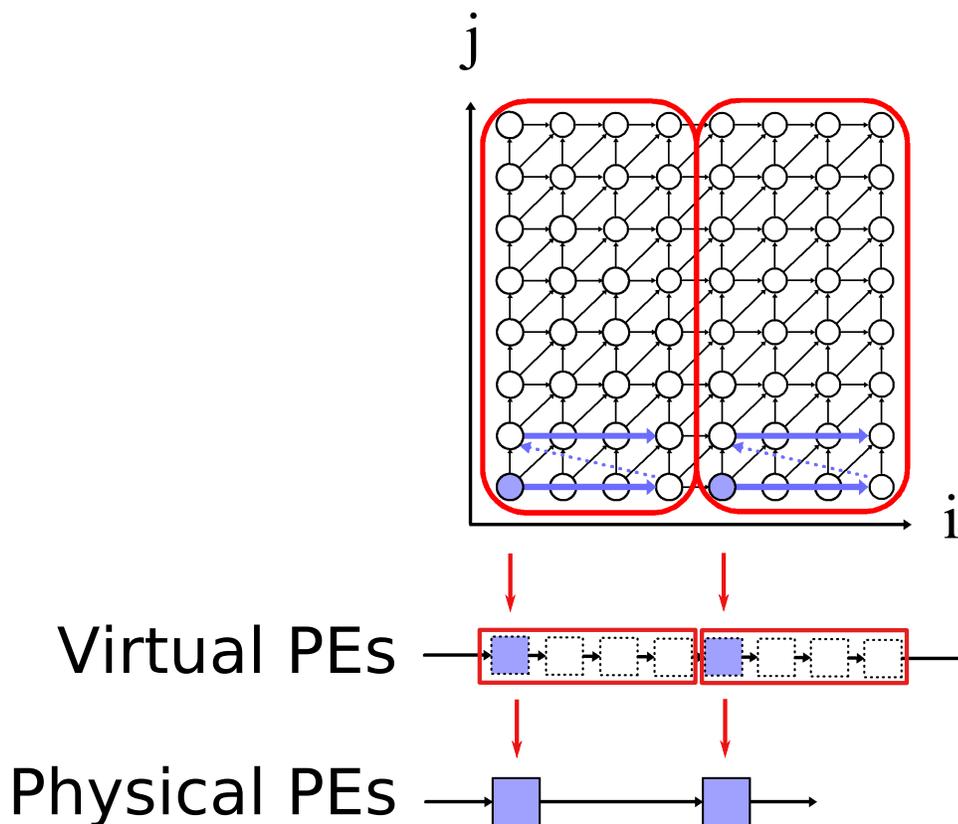
Figure 6.2: A 2-D recurrence domain that is computed by a linear array. The figure illustrates how the domain is projected onto virtual PEs and then partitioned onto physical PEs. array.

the use of a smaller subset of PEs to serially compute a larger workload. We describe how to construct a partitioned 1-D array and show examples of partitioned 1-D and 2-D arrays for the Nussinov algorithm.

Suppose we are given a rectangular computation domain $\mathcal{D} = \{ i_1, \ldots, i_n \mid 1 \leq i_1 \leq S_1; \ \ldots ; 1 \leq i_n \leq S_n \}$ with processor index i_1. Instantiating a PE for every integral value of i_1 requires S_1 PEs. We may instead choose to place PEs at regular intervals of W_1 on the processor index; we now need only $\frac{S_1}{W_1}$ PEs. We denote W_1 as the partition size; a larger partition saves computational resources but also reduces parallelism exploited by the array due to forced serialization. We term the original S_1 unpartitioned processors "virtual" PEs and the $\frac{S_1}{W_1}$ processors realized on the target "physical" PEs. It is the role of a physical PE to serially compute the workload of its partition's W_1 virtual PEs.

Figure 6.2 shows an example of partitioning on a two-dimensional recurrence. The i axis is selected as the processor index, with one physical PE assigned to four virtual PEs. Each physical PE computes its rectangular set of computational points sequentially, though two PEs can operate in parallel.

To perform LSGP partitioning, we must transform the recurrence description and its domain. We replace the single processor index i_1 by the tuple (p_1, v_1), which is related by the equality $i_1 = W_1(p_1-1)+v_1$. The transformed domain of the recurrence is now $\mathcal{D} = \{ p_1, v_1, i_2, \ldots, i_n \mid 1 \leq p_1 \leq \frac{S_1}{W_1};\ 1 \leq v_1 \leq W_1;\ 1 \leq i_2 \leq S_2;\ \ldots\ ; 1 \leq i_n \leq S_n \}$. We now place $\frac{S_1}{W_1}$ physical PEs, one for each integral value of p_1, and find a schedule as described in the previous section.

Partitioning affects the dependencies of the recurrence that span the originally selected processor index. Consider a uniform dependency $X(i_1, \ldots, i_n) \leftarrow X(i_1 + 1, \ldots, i_n)$, which spans processor index i_1. After partitioning, the dependency may refer to a point on the same partition of a physical PE, or to one on a neighboring PE. For example, given the recurrence

$$X(i_1, i_2, \ldots, i_n) = \max \begin{cases} X(i_1 - 1, i_2, \ldots, i_n) \\ X(i_1 + 1, i_2, \ldots, i_n), \end{cases} \tag{6.4}$$

partitioning transforms the two dependencies across i_1 to

$$X(p_1, v_1, i_2, \ldots, i_n) = \max \begin{cases} X(p_1, v_1 - 1, i_2, \ldots, i_n) & \text{if } v_1 > 1 \\ X(p_1 - 1, W_1, i_2, \ldots, i_n) & \text{if } v_1 = 1 \\ \\ X(p_1, v_1 + 1, i_2, \ldots, i_n) & \text{if } v_1 < W_1 \\ X(p_1 + 1, 1, i_2, \ldots, i_n) & \text{if } v_1 = W_1. \end{cases} \tag{6.5}$$

Similar transformations can be applied for longer dependencies on 1-D arrays, as well as dependencies that span both processor axes on 2-D arrays.

6.2.1　A Partitioned 1-D Array for Nussinov

We now partition the 1-D Nussinov array described in Section 6.1.1 along processor dimension k. We replace index k with two new ones (v_k, k'), where $1 \leq v_k \leq W_k$ and $1 \leq k' \leq \lceil \frac{N}{W_k} \rceil$. Here W_k is the width of the partition, and v_k serially enumerates the computation points in a partition that are to be executed by a single PE. The newly introduced indices are related by the equation $k = W_k(k'-1) + v_k$.

We rewrite the Nussinov system of recurrences 4.5 to replace indices (i, j, k) by (i, j, v_k, k'). This affects dependencies that cross the original k dimension. As an example, a dependence of (i, j, k) on $(i, j, k-1)$ is transformed into

$$
\begin{aligned}
&(i, j, v_k, k') \text{ depends on } (i, j, v_k - 1, k') \text{ if } v_k > 1\\
&(i, j, v_k, k') \text{ depends on } (i, j, W_k, k' - 1) \text{ if } v_k = 1.
\end{aligned}
\tag{6.6}
$$

Similarly, the dependence of (i, j, k) on $(i, j, k+1)$ is transformed into

$$
\begin{aligned}
&(i, j, v_k, k') \text{ depends on } (i, j, v_k + 1, k') \text{ if } v_k < W_k\\
&(i, j, v_k, k') \text{ depends on } (i, j, 1, k' + 1) \text{ if } v_k = W_k.
\end{aligned}
\tag{6.7}
$$

The newly introduced dependencies in both cases remain uniform. Applying the partitioning transformation on the Nussinov system of recurrence equations, we get

$$
X(i, j, v_k, k') = \max \begin{cases}
\delta(P(i, j, v_k, k'), Q(i, j, v_k, k')) & \text{if } j - i = 1 \\
\\
\begin{aligned}
&X(i+1, j, v_k, k') \\
&X(i, j-1, v_k, k') \\
&X(i+1, j-1, v_k, k')+ \\
&\quad \delta(P(i, j, v_k, k'), Q(i, j, v_k, k')) \\
&X(i, j, v_k + 1, k') \\
&X_1(i, j, v_k, k') + X_2(i, j, v_k, k') \\
&X_3(i, j, v_k, k') + X_4(i, j, v_k, k')
\end{aligned} & \text{if } k' = 1 \text{ and } v_k = 1 \\
\\
\begin{aligned}
&X(i, j, v_k + 1, k'), \\
&X_1(i, j, v_k, k') + X_2(i, j, v_k, k') \\
&X_3(i, j, v_k, k') + X_4(i, j, v_k, k')
\end{aligned} & \text{if } v_k < W_k \\
\\
\begin{aligned}
&X(i, j, 1, k' + 1), \\
&X_1(i, j, v_k, k') + X_2(i, j, v_k, k') \\
&X_3(i, j, v_k, k') + X_4(i, j, v_k, k')
\end{aligned} & \text{if } v_k = W_k
\end{cases}
$$

$$(6.8)$$

$$
X_1(i, j, v_k, k') = \begin{cases}
X_3(i, j, v_k, k') & \text{if } j - i = 2W_k(k' - 1) + 2v_k \\
X_1(i, j-1, v_k, k') & \text{otherwise}
\end{cases}
$$

$$
X_2(i, j, v_k, k') = \begin{cases}
X(i+2, j, v_k, k') & \text{if } k' = 1 \text{ and } v_k = 1 \\
\\
X_2(i+1, j, v_k - 1, k') & \text{if } v_k > 1 \\
X_2(i+1, j, W_k, k' - 1) & \text{if } k' > 1 \text{ and } v_k = 1
\end{cases}
$$

$$
X_3(i, j, v_k, k') = \begin{cases}
X(i, j-1, v_k, k') & \text{if } k' = 1 \text{ and } v_k = 1 \\
\\
X_3(i, j-1, v_k - 1, k') & \text{if } v_k > 1 \\
X_3(i, j-1, W_k, k' - 1) & \text{if } k' > 1 \text{ and } v_k = 1
\end{cases}
$$

$$
X_4(i, j, v_k, k') = \begin{cases}
X_2(i, j, v_k, k') & \text{if } j - i = 2W_k(k' - 1) + 2v_k \\
X_4(i+1, j, v_k, k') & \text{otherwise}
\end{cases}
$$

$$P(i, j, v_k, k') = \begin{cases} S_i & \text{if } j - i = 1 \\ P(i, j-1, v_k, k') & \text{if } k' = 1 \text{ and } v_k = 1 \end{cases}$$

$$Q(i, j, v_k, k') = \begin{cases} S_j & \text{if } j - i = 1 \\ Q(i+1, j, v_k, k') & \text{if } k' = 1 \text{ and } v_k = 1. \end{cases}$$

A schedule and allocation for the partitioned array is given by

$$\begin{aligned} \tau(i, j, v_k, k') &= -W_k k' - v_k - W_k i + N W_k j \\ \pi(i, j, v_k, k') &= [k'], \end{aligned} \tag{6.9}$$

where according to Equation 6.1, $a_1 = -W_k, a_2 = -1, a_3 = -1$, and $a_4 = 1$. To be conflict-free, the greatest common divisor of W_k and $\lceil \frac{N}{W_k} \rceil$ should be 1. We are now able to control the number of physical PEs required to accelerate the Nussinov computation by changing W_k.
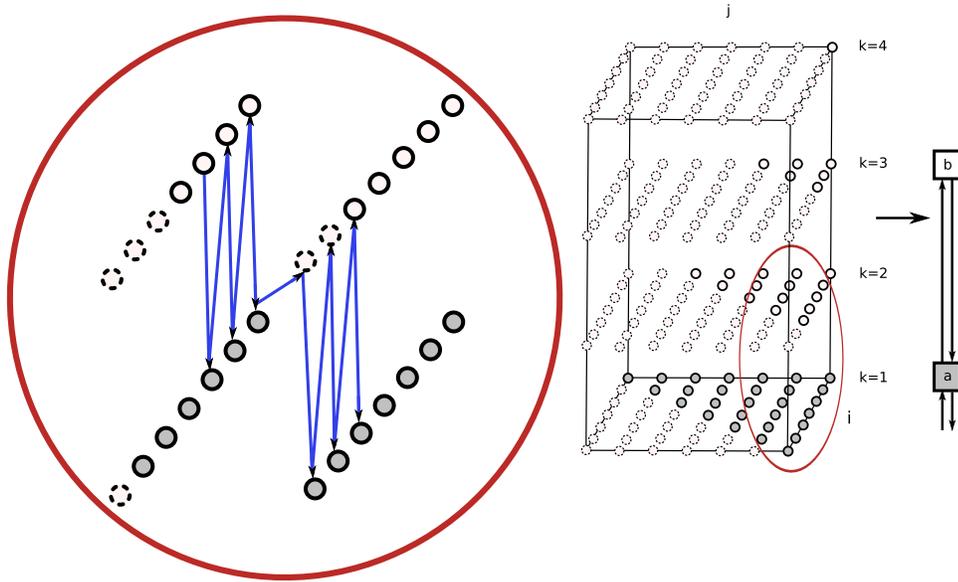
Figure 6.3: The 1-D partitioned array for the transformed Nussinov recurrence with $W_k = 2$. Each PE sequentially computes two planes of points according to the schedule illustrated in the inset.

Figure 6.3 shows the 1-D partitioned array for the Nussinov computation. We have allocated one physical PE for every two virtual processors, so for example, PE a executes the two rectangular planes at $k = 1$ and $k = 2$.

The time to fold an RNA of length N using this array is $\tau(1, N, 1, 1) - \tau(N, 1, W_k, 1) = N^2 W_k - 1$ clocks.

6.2.2 A Partitioned 2-D Array for Nussinov

Our final array is a partitioned 2-D array that is based on the FS-B mapping of Section 4.2.2. In order to reduce complexity, we will partition only a single dimension. Since the most resource-intensive operations are along the j dimension, we select the j axis for partitioning, which allows us to control the number of resource-intensive PEs generated.

Partitioning along dimension j is done by replacing its index by two new indices (v_j, j') with $1 \leq v_j \leq W_j$ and $1 \leq j' \leq \lceil \frac{N}{W_j} \rceil$. The three indices are related by the equation $j = W_j(j' - 1) + v_j$.

We rewrite the Nussinov system of recurrence equations to introduce the two new indices as described in Section 6.2. For example, the dependence of (i, j, k) on $(i, j - 1, k)$ is transformed into

$$
\begin{aligned}
&(i, v_j, j', k) \text{ depends on } (i, v_j - 1, j', k) \text{ if } v_j > 1 \\
&(i, v_j, j', k) \text{ depends on } (i, W_j, j' - 1, k) \text{ if } v_j = 1
\end{aligned}
\tag{6.10}
$$

The transformed system of recurrence equations is:

$$
X(i, v_j, j', k) = \max \begin{cases}
\delta(P(i, v_j, j', k), Q(i, v_j, j', k)) & \text{if } W_j(j'-1) + v_j - i = 1 \\[2mm]
\begin{aligned}
&X(i+1, v_j, j', k) \\
&X(i, v_j - 1, j', k) \\
&X(i+1, v_j - 1, j', k)+ \\
&\quad \delta(P(i, v_j, j', k), Q(i, v_j, j', k)) \\
&X(i, v_j, j', k+1) \\
&X_1(i, v_j, j', k) + X_2(i, v_j, j', k) \\
&X_3(i, v_j, j', k) + X_4(i, v_j, j', k)
\end{aligned} & \text{if } k = 1 \text{ and } v_j > 1 \\[2mm]
\begin{aligned}
&X(i+1, v_j, j', k) \\
&X(i, W_j, j' - 1, k) \\
&X(i+1, W_j, j' - 1, k)+ \\
&\quad \delta(P(i, v_j, j', k), Q(i, v_j, j', k)) \\
&X(i, v_j, j', k+1) \\
&X_1(i, v_j, j', k) + X_2(i, v_j, j', k) \\
&X_3(i, v_j, j', k) + X_4(i, v_j, j', k)
\end{aligned} & \text{if } k = 1 \text{ and } v_j = 1 \\[2mm]
\begin{aligned}
&X(i, v_j, j', k+1), \\
&X_1(i, v_j, j', k) + X_2(i, v_j, j', k) \\
&X_3(i, v_j, j', k) + X_4(i, v_j, j', k)
\end{aligned} & \text{otherwise}
\end{cases}
$$

$$(6.11)$$

$$
X_1(i, v_j, j', k) = \begin{cases}
X_3(i, v_j, j', k) & \text{if } W_j(j'-1) + v_j - i = 2k \\
X_1(i, v_j - 1, j', k) & \text{if } v_j > 1 \\
X_1(i, W_j, j' - 1, k) & \text{if } v_j = 1
\end{cases}
$$

$$
X_2(i, v_j, j', k) = \begin{cases}
X(i+2, v_j, j', k) & \text{if } k = 1 \\
X_2(i+1, v_j, j', k-1) & \text{otherwise}
\end{cases}
$$

$$
X_3(i, v_j, j', k) = \begin{cases}
X(i, v_j - 1, j', k) & \text{if } k = 1 \text{ and } v_j > 1 \\
X(i, W_j, j' - 1, k) & \text{if } k = 1 \text{ and } v_j = 1 \\[2mm]
X_3(i, v_j - 1, j', k - 1) & \text{if } k > 1 \text{ and } v_j > 1 \\
X_3(i, W_j, j' - 1, k - 1) & \text{if } k > 1 \text{ and } v_j = 1
\end{cases}
$$

$$X_4(i, v_j, j', k) = \begin{cases} X_2(i, v_j, j', k) & \text{if } W_j(j'-1) + v_j - i = 2k \\ X_4(i+1, v_j, j', k) & \text{otherwise} \end{cases}$$

$$P(i, v_j, j', k) = \begin{cases} S_i & \text{if } W_j(j'-1) + v_j - i = 1 \\ \\ P(i, v_j - 1, j', k) & \text{if } k = 1 \text{ and } v_j > 1 \\ P(i, W_j, j'-1, k) & \text{if } k = 1 \text{ and } v_j = 1 \end{cases}$$

$$Q(i, v_j, j', k) = \begin{cases} S_j & \text{if } W_j(j'-1) + v_j - i = 1 \\ Q(i+1, v_j, j', k) & \text{if } k = 1. \end{cases}$$

We chose to place PEs in a 2-D arrangement using the following array mapping

$$\pi(i, v_j, j', k) = [j', k].$$
$$\tau(i, v_j, j', k) = 2W_j j' - k + 2v_j - W_j i. \tag{6.12}$$

The schedule conforms to the format described in Equation 6.2, with $a_1 = 2W_j, a_2 = -1, a_3 = 2$, and $a_4 = -1$. The schedule satisfies all the dependence constraints of the recurrence and is conflict-free if the greatest common divisor of W_j and 2 is 1, i.e., W_j must be odd. Similarly, the greatest common divisor of a_1 and $\lceil \frac{N}{W_j} \rceil$ should be 1. The latency of execution for an RNA of length N is $\tau(1, W_j, \frac{N}{W_j}, 1) - \tau(N-1, W_j, \frac{N}{W_j}, 1) = W_j(N-2)$ clocks.

6.3 Evaluation

In this section we compare the various arrays suggested in this dissertation to accelerate the Nussinov algorithm. We have coded, synthesized and executed the three resource-constrained arrays derived in this chapter on our FPGA system. We have verified correctness by folding thousands of RNAs in hardware and comparing the result to the software baseline. We have used the same software and hardware systems described in Section 4.4 for our performance comparison.

Table 6.1 summarizes the six main arrays, which are parametrized by the maximum RNA size that can be folded on each one. The partitioned arrays are further parametrized by the tile widths W_j and W_k, which control the number of processing elements instantiated. The table is sorted by the number of processing elements

Table 6.1: Summary of three full-size and resource-constrained arrays for the Nussinov DP recurrence. The table lists the number of processing elements instantiated by each array to fold an RNA of length N, the amount of memory words required in each PE, and the pipelining period. A lower pipelining period equates to a faster array. The parameters W_j and W_k are tile widths as described in the previous sections.

Design	# PEs	Memory / PE	Pipelining Period
FS-A	$\frac{N^2-3N+2}{2}$	$\Theta(1)$	$\frac{N-1}{2}$
FS-B	$\frac{N^2-N}{4}$	$\Theta(1)$	$2N-5$
FS-C	$\frac{N^2-N}{4}$	$\Theta(1)$	$N-2$
Partitioned 2-D	$\frac{N^2+NW_j-2N}{4W_j}$	$\Theta(W_j)$	NW_j-2W_j
Unpartitioned 1-D	$\frac{N-1}{2}$	$\Theta(N)$	N^2+N-2
Partitioned 1-D	$\frac{N-1}{2W_k}$	$\Theta(NW_k+W_k^2)$	N^2W_k-1

required by each array, illustrating that the fewest PEs are required by the resource-constrained arrays. While these arrays use the least logic for computation in PEs, the number of memory words required per PE is high. Memory storage, however, can be efficiently implemented on modern FPGAs; in our work we have used on-chip block RAM memories and distributed memories in lookup tables (LUTs). Unsurprisingly, the full-size arrays have the lowest pipelining periods and exhibit the best speedups because they exploit the most parallelism. With partitioned arrays, we have the freedom to trade area for increased parallelism by varying the tile width.

Next, we studied the performance of the three resource-constrained arrays by synthesizing various instantiations on our target FPGA device. We report speedup over our dual core software baseline, as well as FPGA slice and block RAM usage. The speedup numbers are summarized in Figure 6.4, and the FPGA resource usage is graphed in Figures 6.5 and 6.6. For reference, we also list the same data values in Tables 6.2-6.4.
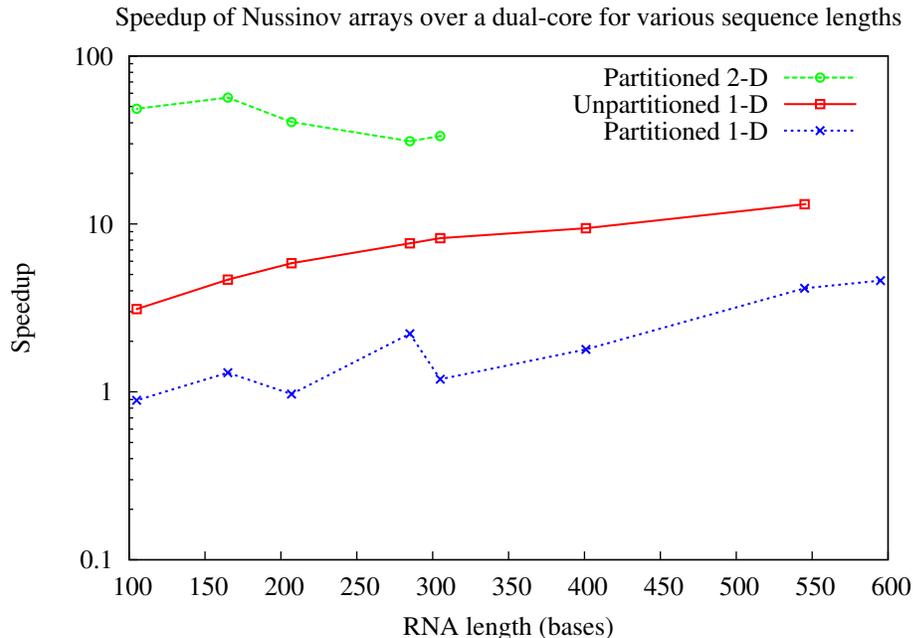
Figure 6.4: Speedup of three resource-constrained Nussinov arrays. The y-axis is in log scale. The three array families have distinct order of magnitude speedups.

Partitioned 2-D array. The performance of the one 2-D resource-constrained array we have suggested in this chapter is shown in Table 6.2. We wrote a script that uses heuristics to determine the fastest instantiation of this array post place-and-route for a large number of sequence lengths. For each sequence length, the script determines the lowest feasible tile width (hence, exploiting the most parallelism) and the highest possible frequency. We measured the speedup of five of these instantiations to determine array performance. In addition to the tile widths selected, we have listed the FPGA resources consumed, clock frequency of the designs, and the single core and dual core speedups compared to our baseline system. Performance was computed based on the time taken to fold 1 million randomly generated RNA sequences.

Recall that with our full-size arrays we were able to fold RNAs of length up to 83 bases. Using our partitioned 2-D array we can fold RNAs 3.6× longer, while still achieving order-of-magnitude speedups over the dual core baseline. For this implementation we did not use the on-chip block RAM memories; instead, storage in each PE was implemented using distributed memories in slices. These distributed memories limit the maximum length of RNAs that can be folded on this array to 305 bases.

FPGA slice usage of Nussinov arrays for various sequence lengths
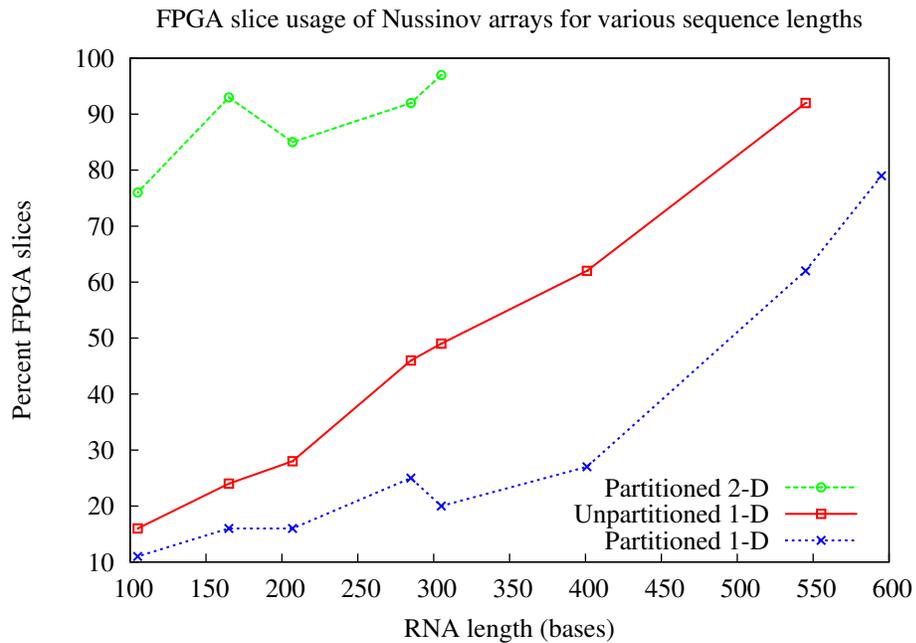
Figure 6.5: Percent of the target FPGA slices used by instantiations of three array types for various sequence lengths. Depending on the requirements, a designer can select the most suitable array type.

FPGA block RAM usage of Nussinov arrays for various sequence lengths
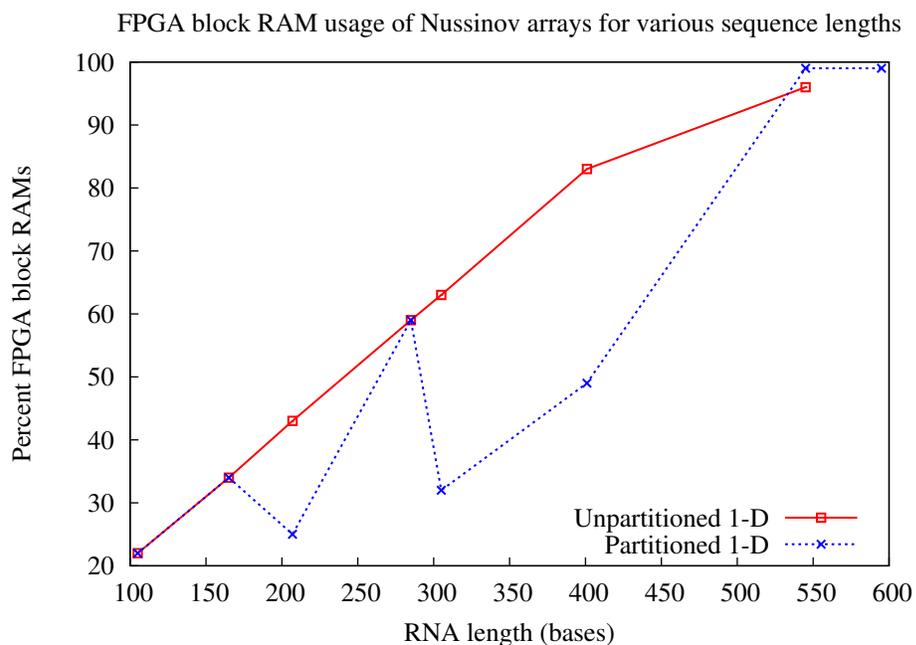
Figure 6.6: Percent of the target FPGA block RAMs used by instantiations of two array types. The partitioned 2-D array does not use block RAM memories.

Table 6.2: Speedups of the Nussinov partitioned 2-D array for various RNA lengths.

N	Precision	W_j	# PEs	Total slices	Slice memories	Frequency (MHz)	Single core speedup	Dual core speedup
305	8	61	455	48,099 (97%)	19,170 (78%)	180	66.62×	33.33×
295	8	59	440	47,376 (96%)	18,542 (75%)	180	-	-
285	8	57	425	45,661 (92%)	17,912 (72%)	180	62.19×	31.10×
275	8	55	410	44,907 (91%)	17,286 (70%)	180	-	-
265	8	53	395	43,251 (87%)	16,662 (67%)	180	-	-
255	8	51	380	40,848 (83%)	16,034 (65%)	180	-	-
245	8	35	487	43,386 (88%)	16,321 (66%)	180	-	-
225	8	45	335	34,149 (69%)	12,166 (49%)	180	-	-
215	8	43	320	31,410 (63%)	11,636 (47%)	180	-	-
207	7	23	513	41,788 (85%)	9,802 (39%)	180	80.96×	40.46×
201	7	67	200	24,135 (49%)	8,241 (33%)	180	-	-
189	7	27	375	33,009 (67%)	7,334 (29%)	180	-	-
185	7	37	275	28,246 (57%)	9,838 (40%)	180	-	-
177	7	59	176	20,910 (42%)	7,256 (29%)	180	-	-
169	7	13	585	42,288 (86%)	7,731 (31%)	180	-	-
165	7	11	653	46,137 (93%)	8,464 (34%)	190	112.92×	56.47×
161	7	23	319	28,153 (57%)	6,302 (25%)	180	-	-
153	7	17	378	30,217 (61%)	5,322 (21%)	180	-	-
145	7	29	215	20,780 (42%)	4,480 (18%)	180	-	-
141	7	47	140	16,371 (33%)	5,538 (22%)	180	-	-
133	7	19	263	23,595 (48%)	5,222 (21%)	180	-	-
129	7	43	128	14,971 (30%)	5,084 (20%)	180	-	-
125	6	5	800	47,546 (96%)	8,257 (33%)	190	-	-
121	6	11	358	26,189 (53%)	4,402 (17%)	190	-	-
117	6	9	403	28,760 (58%)	4,675 (19%)	190	-	-
105	6	5	567	37,456 (76%)	6,380 (25%)	190	96.92×	48.49×
93	6	3	729	41,557 (84%)	4,446 (18%)	180	-	-
85	6	5	374	26,766 (54%)	3,953 (16%)	190	-	-

Table 6.3: Speedups of the Nussinov unpartitioned 1-D array for various RNA lengths.

N	Precision	# PEs	Total slices	Slice memories	BRAMs	Frequency (MHz)	Single core speedup	Dual core speedup
545	9	272	45,692 (92%)	22,637 (92%)	232 (96%)	190	25.80×	13.10×
401	8	200	30,618 (62%)	6,099 (24%)	201 (83%)	190	18.71×	9.43×
305	8	152	24,545 (49%)	3,722 (15%)	153 (63%)	220	16.36×	8.22×
285	8	142	22,651 (46%)	3,220 (13%)	143 (59%)	220	15.27×	7.66×
207	7	103	14,181 (28%)	2,049 (8%)	104 (43%)	230	11.62×	5.83×
165	7	82	11,936 (24%)	1,572 (6%)	83 (34%)	230	9.41×	4.66×
105	6	52	8,099 (16%)	973 (3%)	53 (22%)	240	6.19×	3.11×

Table 6.4: Speedups of the Nussinov partitioned 1-D array for various RNA lengths.

N	Precision	W_k	# PEs	Total slices	Slice memories	BRAMs	Frequency (MHz)	Single core speedup	Dual core speedup
595	9	3	99	38,899 (79%)	22,619 (92%)	239 (99%)	180	8.97×	4.60×
545	9	3	91	30,732 (62%)	13,668 (55%)	239 (99%)	180	8.11×	4.14×
401	9	5	40	13,493 (27%)	3,315 (14%)	119 (49%)	180	3.55×	1.79×
305	8	6	26	10,036 (20%)	2,614 (10%)	77 (32%)	190	2.36×	1.19×
285	8	3	48	12,380 (25%)	1,756 (7%)	143 (59%)	190	4.41×	2.22×
207	7	5	21	7,877 (16%)	1,683 (6%)	62 (25%)	190	1.94×	0.97×
165	7	3	28	8,285 (16%)	1,164 (4%)	83 (34%)	190	2.63×	1.30×
105	6	3	18	5,692 (11%)	869 (3%)	53 (22%)	200	1.77×	0.89×

Unpartitioned 1-D array. Table 6.3 lists various instantiations of the unpartitioned linear array we built and tested on our FPGA platform. Performance was computed by folding 100,000 randomly generated RNAs. In this implementation we have used on-chip block RAMs for storage in each PE. This unpartitioned 1-D array is able to fold RNAs still larger than the previous arrays, up to 545 bases, using over 90% of the FPGA's LUT and block RAM resources. In this case we have sacrificed parallelism in order to be able to fold large RNA sequences on the target device. While the previous arrays fold an RNA in $\Theta(N)$ clocks, the linear array takes $\Theta(N^2)$ time, which reduces dual core speedup to $13\times$.

Table 6.3 also illustrates the tradeoff a user is able to make using our resource-constrained array. Given an FPGA device with fewer logic resources, the unpartitioned 1-D array may be used instead of the partitioned 2-D array in exchange for exploiting less parallelism. For example, the former array can fold RNAs of length between 105 and 305 bases using 22-51% of the logic resources as the latter array. The dual core speedups are, however, reduced to between 3-9$\times$.

Partitioned 1-D array. Our final array is the most resource-constrained array we consider in this dissertation. As shown in Table 6.4, we can fold RNAs of length up to 595 bases, which is sufficient for most biologists' needs, with a tile width of 3. Note that we are limited by on-chip storage rather than computational logic resources, having saturated the block RAM and distributed memories available on our FPGA device. The dual core speedup folding 595 base RNAs is a modest $4\times$.

We have also shown the partitioned 1-D array synthesized for sequence lengths between 105-545 to compare against the two prior arrays. The partitioned 1-D array uses the fewest logic and memory resources and can be used in situations where performance close to a dual core CPU is required. Such a situation may arise when, for example, an entire sequence analysis pipeline, of which RNA folding is one stage, is to be implemented in hardware.

Resource and parallelism tradeoff. To get a better understanding of the tradeoff between area and parallelism, we synthesized the partitioned 2-D array using various tile widths to fold sequences of length 105 bases. Recall from Section 6.2.2 that we

Table 6.5: Speedups of the Nussinov partitioned 2-D array showing the tradeoff between area and parallelism controlled by tile width W_j. All arrays were synthesized at a clock frequency of 180 MHz.

N	W_j	# PEs	Total slices	Slice memories	Single core speedup	Dual core speedup
105	5	567	37,718 (76%)	6,534 (26%)	92.29×	46.18×
105	7	413	28,184 (57%)	4,191 (17%)	67.87×	33.96×
105	15	207	16,568 (33%)	2,477 (10%)	32.60×	16.31×
105	21	155	13,765 (28%)	3,039 (12%)	23.45×	11.73×
105	35	104	11,420 (23%)	3,603 (14%)	14.18×	7.09×

may select any value for the tile width W_j such that W_j is odd, and the greatest common divisor of $2W_j$ and $\lceil \frac{N}{W_j} \rceil$ is 1.

Table 6.5 charts the results for tile widths between 5 and 35. We see a clear reduction in FPGA LUT resources for increased serialization of computation for large tile widths. Although serialization increases the amount of memory storage required per PE, our results show that they can be very efficiently implemented on modern FPGA fabrics.

6.4   Conclusions

While the primary goal of our dissertation is to exploit the maximum available parallelism in dynamic programming algorithms, providing the ability to limit parallelism exploited in exchange for resource savings has important consequences. The flexiblity allows a designer to accelerate resource-heavy algorithms, process large inputs, make use of low-cost, less capable FPGA devices, or implement dynamic programming accelerators as part of a larger hardware pipeline of accelerators. The designer may select the least resource-heavy array that matches the performance required.

We have used an existing technique to build partitioned and unpartitioned 1-D and 2-D arrays and demonstrated the resource versus parallelism tradeoff on an FPGA system. Our resource-constrained arrays allow us to fold RNAs of up to 595 bases,

satisfying the majority of biologists' requirements. A designer may select array instantiations that consume between 16-90% of FPGA resources for speedups of up to $56\times$ over a dual core implementation.

Chapter 7

Optimal Runtime Reconfiguration Strategies for Systolic Arrays

The philosophy of classic systolic array design is to build a single array that accelerates a system of recurrence equations [96]. The standard approach is to build a single latency-space optimal array. If resources are a constraint, a partitioned 1-D or 2-D array of fixed size is generated instead. In this dissertation, we have also suggested a novel technique to trade off resources and exploit increased parallelism. Nevertheless, just as in related work, we have also thus far only generated a single array accelerator for every recurrence.

Using a single array mapping can minimize total computation time when the input is a continuous stream of data, as in many signal-processing applications. Oftentimes, one may instead seek to process a large collection of *discrete* inputs. In such a case, array design can explicitly minimize total execution time by maximizing computational throughput, processing inputs in a pipelined fashion. Streams of discrete inputs arise naturally in the domain of bioinformatics, speech recognition, and media processing, where the input may consist of many short inputs such as high-throughput sequencing reads or of probabilistic sequence models such as hidden Markov models. In this chapter, we seek to deal efficiently with such streams of discrete inputs.

Apart from the choice of the appropriate design family—whether latency- or throughput-optimized, or resource-constrained—once an array design has been selected, it must be instantiated on the target device with a fixed array size and hence a fixed input size. If all inputs to the array are the same size, there is a naturally most efficient array size; otherwise, for any fixed array size, smaller inputs must be padded out

to the array's input size, while larger ones (if supported at all) must be split and processed in multiple passes. Existing work has focused largely on selecting a single array design that yields good performance over a range of input sizes [96]. This approach is natural for VLSI synthesis, in which the design cannot change to respond to variations in input size.

Other target platforms such as FPGAs, however, have the flexibility to handle a range of input sizes without sacrificing efficiency. Because these platforms can be reconfigured quickly (less than a second) with a new array design, one can efficiently alter the accelerator to accommodate runs of larger and smaller inputs. For smaller inputs, a smaller array instantiation eliminates the need for input padding and the associated useless computation. Moreover, because higher-throughput arrays often require substantially more computational resources, it may be possible to exploit pipelining at smaller sizes while reverting to more resource-efficient but slower arrays at larger sizes.

In this work, we present an algorithmic framework for deciding which of a large set of possible systolic array designs to use so as to minimize total computation time on a known distribution of input sizes. Our algorithms select a set of designs (either bounded or unbounded) and indicate when to switch among them, assuming that the inputs are sorted monotonically by size. We demonstrate the utility of our approach for the bioinformatics domain by accelerating an FPGA implementation of the Nussinov RNA folding algorithm. We apply our algorithm to select arrays from these families for both real and synthetic size distributions and estimate the resulting speedups.

Finally, we demonstrate a realization of runtime array reconfiguration on a Xilinx Virtex-4 LX100-12 FPGA device and measure the performance impact of reconfiguration on a database containing tens of millions of short RNA sequences to be folded. We demonstrate that our runtime reconfiguration scheme confers substantial efficiency benefits even when the input length distribution is biased toward low-throughput arrays, when reconfiguration time is close to one second, and when only a small number of distinct arrays may be used.

7.1 Selecting an Optimal Set of Arrays

Suppose we are given a large collection of input items with lengths in the range $1 \ldots M$. We assume that the collection has been analyzed offline to determine the number $C(i)$ of inputs of each size i, and that it has been sorted in nondecreasing order of input size. These assumptions are reasonable in, for example, bioinformatics data sets, which are typically generated and formatted offline and then stored in a database for analysis. In the case of an online search, sequences may be binned by size and buffered upon receipt from clients.

Let A be a set of candidate systolic array designs. Each design $a \in A$ is actually a family of instantiations $a(i)$ parametrized by input size i. The largest feasible input size $S(a)$ for a is determined by the resource limits of the target device. Array $a(i)$ has a *block pipelining period* $\beta_{a(i)}$, which is the required delay in cycles between successive inputs; the reciprocal of $\beta_{a(i)}$ is the array's throughput.

Let $E(i)$ be the minimum time required to process all inputs of size 1 to i inclusive, using some combination of designs from A. Our goal is to compute $E(M)$. A dynamic programming recurrence for $E(i)$ is given by

$$E(i) = \min_{a \in A} \ \min_{1 \le j < i} \left\{ \ E(j-1) + \rho + \delta_a(j, i) \ \right\}$$

$$\delta_a(j, i) = \begin{cases} L_a(i) + \displaystyle\sum_{j \le k \le i} C(k) \times \beta_{a(i)} & \text{if } i \le S(a) \\[2em] \infty & \text{otherwise,} \end{cases} \qquad (7.1)$$

where ρ is the reconfiguration time needed to load a new array on the target device, and $\delta_a(j, i)$ is the time to execute all inputs of length $j \ldots i$ on array $a(i)$. $\delta_a(j, i)$ is computed as the sum of the pipelining periods of the inputs and the latency of execution, $L_a(i)$, of the last input on the array. This latency, however, is negligible compared to the rest of the sum and the reconfiguration time, so we ignore it hereafter. All times are expressed in cycles but could be converted to seconds to combine designs with different clock speeds.

To compute the optimal execution time for inputs of length $1 \ldots i$, the recurrence considers all possible reconfiguration locations $1 \leq j < i$. Sequences of length $j \ldots i$ are executed on array $a(i)$. This is followed by reconfiguration and the optimal execution of the remaining inputs. We must consider all possible locations for reconfiguration and every candidate family of arrays in A. Correctness follows from the optimal substructure of the optimization problem.

The initialization condition is $E(0) = -\rho$, and the optimal execution time $E(M)$ over all inputs can be computed bottom-up in i. We can retrieve the optimal sequence of array instantiations using a standard traceback procedure.

The full algorithm solves M subproblems, each requiring maximization over $O(M|A|)$ cases. One may precompute and store all required δ values in time $O(M^2|A|)$, making each case constant-time and yielding total computation time of $O(M^2|A|)$. This running time is practical provided that the size range M is restricted or that the actual set of input sizes is sparse. If it is large, we could compute faster at some cost to result quality by quantizing the set of sizes considered.

7.2   Application to RNA Folding

We have applied our reconfiguration algorithm to the Nussinov RNA folding problem. In this section we first summarize the array families that will be considered.

We consider the two full-size arrays, FS-A and FS-C, described in Section 5.7. While both arrays have the same latency of $2N - 5$ clocks, their throughput differs. The block pipelining period β (reciprocal of throughput) of the two arrays is respectively $4\times$ and $2\times$ lower than the latency. Unfortunately, increased throughput also results in a larger array; the FS-A array uses twice as many processing elements as the FS-C array for the same input size.

We have also included a clustered version of array FS-B that we call FS-B-clustered. The processing elements of the FS-B array are only 50% efficient, i.e., they are active in only one of every two clock cycles. We can increase the array's efficiency [91] by clustering two processing elements into one. The throughput of the clustered array

Table 7.1: Full-size arrays for Nussinov recurrence.

Array	Pipelining Period (β)	# Processing Elements	Maximum N
FS-A	$\frac{N-1}{2}$	$\frac{N(N+1)}{2}$	49
FS-C	$N-2$	$\frac{N(\frac{N}{2}+1)}{2}$	81
FS-B-clustered	$2N-5$	$\frac{N(\frac{N}{2}+1)}{4}$	97

is $2N - 5$ clocks, the lowest among the three full-size arrays. However, clustering improves space efficiency and so increases the size of the largest RNA molecule that can be folded. Table 7.1 summarizes these results and shows the maximum size of RNA sequences that can be folded on arrays synthesized on a Xilinx Virtex-4 LX100-12 FPGA device.

Note that because we used an early version of our designs, which did not support the retiming optimization, the arrays are clocked at a low frequency of 80 MHz. However, this does not alter the performance boost achieved through reconfiguration, although, it does affect the speedup compared to the software baseline.

Another source of parallelism for array design comes from the discrete nature of the input stream. Because each input sequence can be processed independently, we can instantiate multiple copies of a small array to increase throughput for small input sizes. For example, the FS-C array can be instantiated on the target FPGA to fold sequences of length up to 81 and has $\beta = N - 2$. Using the formula in Table ??, we calculate that 1680 processing elements for the FS-C array can be synthesized on the target FPGA. If we equally divide these processing elements among two FS-C arrays of the same size, we can effectively reduce the block pipelining period to $\frac{N-2}{2}$. We can estimate the maximum size of k identical FS-C arrays running in parallel by finding the positive solution to the equation $\frac{N}{2}(\frac{N}{2} + 1) = \frac{1680}{k}$ (in general this equation is not quadratic). For $k = 2$, we can instantiate two copies of the FS-C array of size $N = 56$. In our optimization, we consider parallel instantiations of each array type as another family of arrays available for use.

7.3   Results

To gauge the performance impact of reconfiguration, we first tested our algorithm on synthetic data. We generated sequences whose length was distributed according to geometric ($p = 0.05$), normal ($\mu = 48$, $\sigma = 25$), and Pareto (order 0.6) distributions. We generated 1 billion bases in each case, and all sequences were at most 97 bases in length. The synthetic data represents ideal inputs with lengths biased toward higher-throughput array sizes. We used the three arrays described previously. We permitted a maximum of three parallel instantiations of each array running simultaneously on the target FPGA. This was done to reduce array synthesis time. We may achieve slightly better performance with four to five instantiations, but it is likely to provide diminishing returns. We assumed a reconfiguration time of 20 ms, as reported in [23].
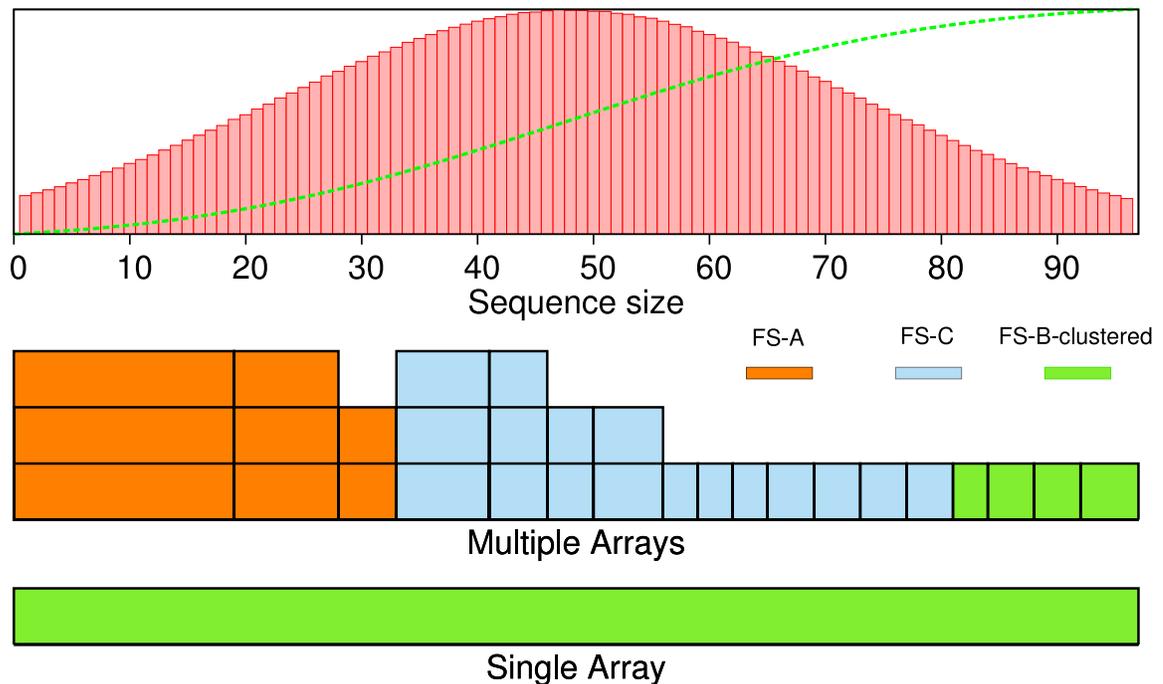
Figure 7.1: Optimal selection of Nussinov arrays to fold synthetic sequences with normally distributed lengths. Top: histogram and cumulative frequency of sequence lengths. Middle: design with reconfigurations produced by our algorithm. Bottom: best single array supporting all input lengths (up to 97 bases). Size of each array instantiation is given by length of longest sequence it processes.

The set of optimal arrays selected by our dynamic programming algorithm for the normally distributed sequences is shown in Figure 7.1 using three graphs. The top graph shows both a histogram and the cumulative frequency of all sequences. The traditional method selects a single array, in this case the slower FS-B-clustered array, at a fixed size large enough to fold all input sequences, as shown in the bottom graph. FS-B-clustered was selected because it was the only array type that fit on our FPGA given the sequence length requirement of 97. The reconfigured solution is shown in the middle graph. Our algorithm selects various instantiations of the FS-A, FS-C, and FS-B-clustered arrays to process subsets of the sequences. The size of an array instantiation is the length of the longest sequence it executes. The number of stacked boxes indicate the number of parallel instantiations of the array synthesized on the FPGA. The number of columns denote the number of array instantiations selected by the algorithm, in this case eighteen. Our algorithm predicts a speedup of 4× over the non-reconfigured solution. We obtained similarly encouraging results for the geometric (20×) and Pareto (2×) distributions.

7.3.1   Folding Pyrosequencing Reads

In the past decade, short 20-30 base noncoding RNAs have been discovered to be important regulators of eukaryotic genes [26]. One class of such RNAs is the microRNAs (miRNAs). An important biological problem is how to detect miRNA sequences in the genomic DNA from which they are copied. Currently, new sequences are scanned computationally to detect candidate miRNA precursor sequences, which are then experimentally validated [17]. An important feature of miRNA precursors is their distinctive secondary structure, which can be detected in part by looking for an unusually high number of paired bases when a short piece of DNA is treated as RNA and folded. We use the Nussinov algorithm to scan for high numbers of paired bases.

Large-scale genomic DNA sequencing today is done via *pyrosequencing*, a massively parallel sequencing technique. Pyrosequencing of a DNA sample can obtain short sequence fragments (100 bases or less) at a rate of tens of millions of bases per hour. While some pyrosequencing datasets are assembled to produce one long genomic sequence, others, especially environmental sequencing [124], sequence DNA from many organisms at once and so remain highly fragmented. We therefore investigated the

impact of our methods on the rate at which the Nussinov algorithm could be applied for miRNA detection within the fragmentary DNA produced by pyrosequencing.

We folded pyrosequencing reads from 130 environmental samples[8]. The dataset contained 22.6 million reads totaling 2.7 billion bases, with an average read length of 121. We sorted the sequences in the dataset by length and split them into two equal halves in an offline process. Two instances of the software was run independently on both cores of our 3 GHz Intel Core 2 Duo CPU baseline. The time to fold all reads in the dataset using the two cores was $8,975$ seconds.

We used the hardware system described in Section 4.4. For reconfiguration, our current system must load the FPGA configuration file from the host CPU's memory rather than on-board flash, and so has a reconfiguration time of 400 ms—dramatically longer than that reported in [23].

Sorting and formatting the dataset for use by the hardware takes significant execution time (about 70 seconds) using a naive strategy; we may be able to accelerate this step to achieve an efficient implementation, for example, by using in-memory sort. We have not included this cost in the execution time of the software or hardware runs and assume the database is preprocessed during data generation time.

To process all the collected reads, we split reads greater than 97 bases into smaller chunks with an overlap of 25 bases. This increases the workload but enables us to fold all sequences while still being long enough to predict important features of the structure. Figure 7.2 shows the results of the experiment. Sequence lengths were heavily biased toward the largest size supported by the hardware, which requires our slowest array (FS-B-clustered); hence, we do not expect a large speedup. Moreover, our algorithm selected fewer instantiations of the arrays due to our hardware's comparatively longer reconfiguration time. As expected, the FS-B-clustered array was used for the longest reads, followed by the FS-C array. Though the FS-A array can be selected for reads smaller than 50 bases, the algorithm preferred multiple units of FS-C, which has higher throughput. The FS-A array was used only for sequences smaller than 34 bases. We predicted a speedup of 51% for the reconfigured over the single-array solution.

---

[8]http://scums.sdsu.edu/meta_overview.php
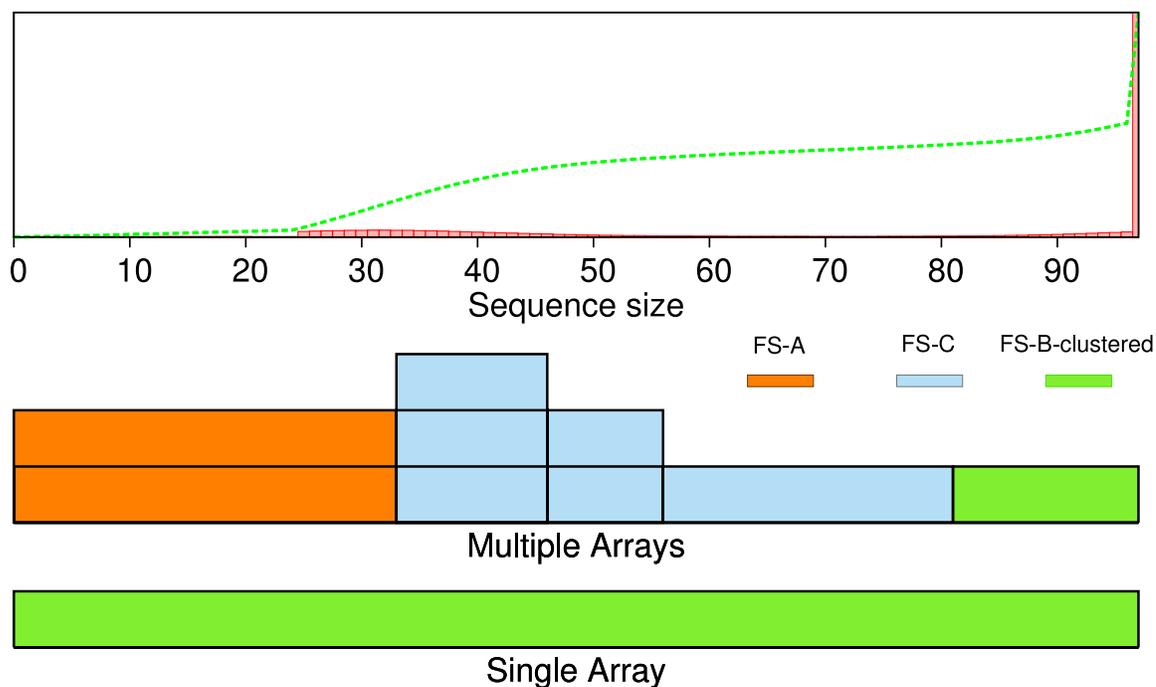
Figure 7.2: Optimal selection of Nussinov arrays to fold pyrosequencing reads. Estimated speedup of the reconfigured solution over the single-array approach was 51%; direct measurement yielded 48%.

Using the single-array solution, we achieved a runtime of 107 seconds and a dual core speedup of 84× over the software. Runtime reconfiguration using our set of five arrays resulted in an execution time of 72 seconds and an improved dual core speedup of 125× over the software. Indeed, runtime reconfiguration resulted in 48% faster execution than the single latency-space optimal array, closely matching our prediction. We achieved significant speedups through runtime reconfiguration despite the input's bias toward long sequences requiring the low-throughput FS-B-clustered array (accounting for 64 of the 72 seconds of execution). Finally, we note that input bandwidth was not a limiting factor for performance.

For comparison, we computed the best possible speedup on a hypothetical FPGA system that can load a new design without cost, i.e., for every sequence we always use the best array, assuming zero cost for reconfiguration. Our algorithm predicts a speedup of 59%. Clearly, improving FPGA reconfiguration time beyond the current 400 ms will result in diminishing returns in performance.
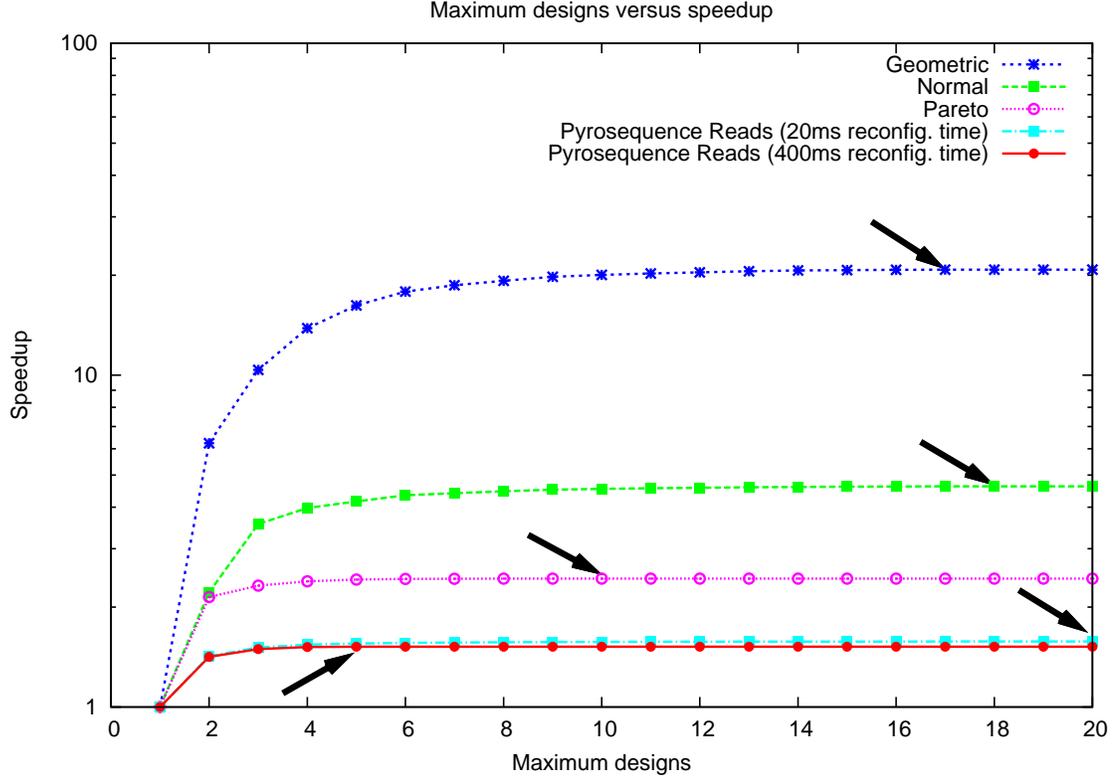
Figure 7.3: Speedup of a reconfigured solution as a function of the maximum number of instantiations allowed. We may choose to limit the number of instantiations to the knee of the curve in order to reduce synthesis time. The y axis is shown in log scale.

7.3.2 Restricting the Number of Arrays

We continued our inquiry by investigating the performance of reconfiguration when restricted to a limited number of arrays. The number of arrays may be limited for two reasons. First, if optimization selects arrays that have not yet been synthesized, the cost of processing a data stream is increased by the substantial costs of synthesis for each new design used. Second, even if synthesis cost is discounted (say, because a library of designs will be reused over many datasets), the target platform may have limited storage to hold alternative array designs (e.g. bitfiles for FPGAs) and so may not support rapid reconfiguration among many designs. For example, the system described in [23] can reconfigure quickly from a set of configuration files kept in on-board non-volatile storage, but this storage holds fewer than six designs.

The algorithm of Equation 7.1 can easily be applied to a restricted set of pre-synthesized array designs, but this does not address the problem of selecting too many designs for the target's storage. To restrict the number of designs actually used in our reconfiguring solution, we modify Equation 7.1 to compute $E(i, n)$, the minimum time required to execute all inputs of length 1 to i using at most n array instantiations. The revised recurrence adds a factor of n to the running time.

$$E(i, n) = \min \begin{cases} E(i, n-1) \\ \min_{1 \leq a \leq |A|} \quad \min_{1 \leq j < i} \left\{ \; E(j-1, n-1) + \rho + \delta_a(j, i) \; \right\} \end{cases} \tag{7.2}$$

Figure 7.3 shows how the best estimated speedup for our real and synthetic datasets, compared to the single-array baseline, varies with the number of designs allowed for reconfiguration. The smallest number of designs that minimizes execution time (computed using Equation 7.1) for the experiments is shown by the arrows. We can achieve 90% of the full speedup available from reconfiguration using just two designs for the pyrosequencing reads and 3-8 designs for the synthetic data.

7.3.3 Folding Long Reads

SRP002017. Next, we folded 454 pyrosequencing reads from a metagenomics sample of a mature Mediterranean deep chlorophyll maximum community[9]. The dataset contains 1.2 million reads with a total of 772 million bases. For this experiment we used the full-size arrays described in Table 7.1 as well as all of the partitioned 2-D arrays shown in Table 6.2 of Chapter 6. With these arrays we can now fold sequences up to 305 bases long. We pre-processed the SRP002017 dataset to split sequences longer than 305 bases to smaller chunks with an overlap of 25 bases.

Since the sequences in this database are long, our algorithm always chooses various instantiations of the partitioned 2-D array. Figure 7.4 shows the expected and measured speedups through reconfiguration using between one and six hardware designs. Our algorithm predicts an optimal speedup of 92% using six designs. On our FPGA platform we used the six suggested array designs to achieve an 88% speedup over the

---

[9]Sequence data is available under the SRA accession SRP002017 at `http://trace.ncbi.nlm.nih.gov/Traces/home/`
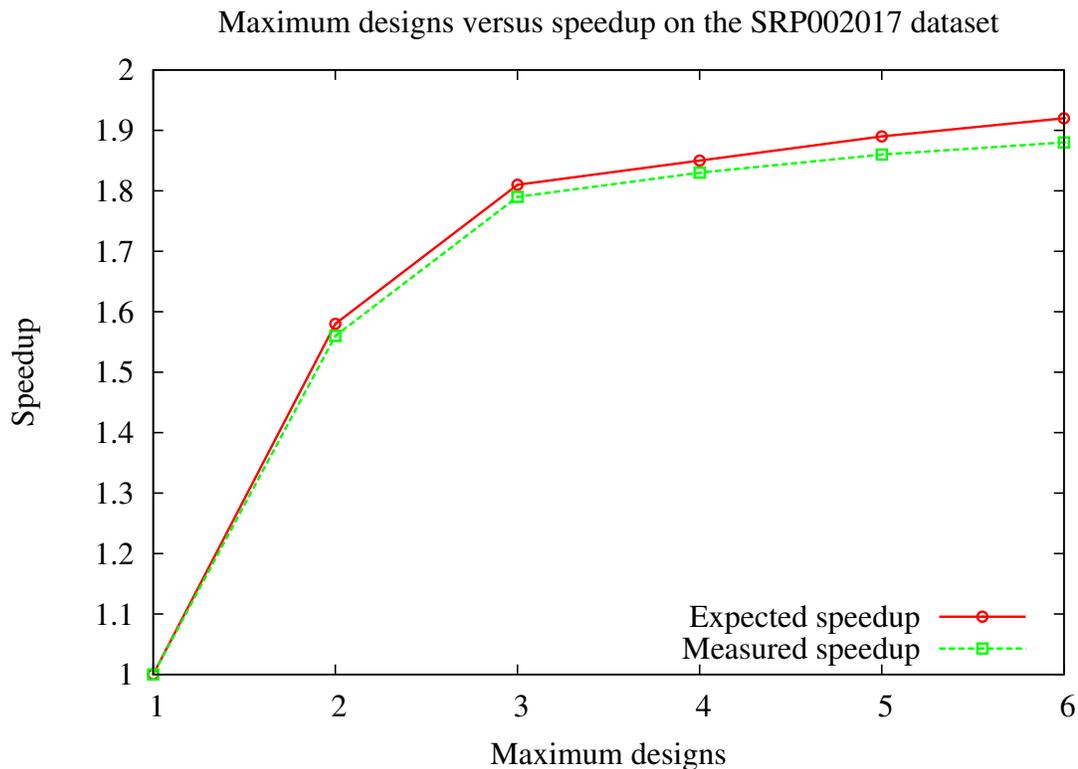
Figure 7.4: Reconfiguring the FPGA using up to six Nussinov array instantiations, we are able to achieve a speedup between 56 and 88% over a single array when folding the SRP002017 dataset.

single partitioned 2-D instantiation of size 305. Furthermore, as seen in the graph, we are able to achieve a significant 79% speedup using just three partitioned 2-D arrays. We folded the SRP002017 dataset using the software implementation on the dual core baseline, which ran in $4,203.00$ seconds—our reconfigured solution is $45.42\times$ faster.

SRP000960. For our final experiment we folded 485,079 reads totaling 143 million bases from the Human Microbiome Project next generation technology pilot[10]. For this experiment we used all of the Nussinov arrays we have built in this dissertation, allowing us to fold the entire dataset without having to split reads.

---

[10]Sequence data is available under the SRA accession SRP000960 at `http://trace.ncbi.nlm.nih.gov/Traces/home/`

The largest read in the dataset is 507 bases long, which can be processed by the unpartitioned 1-D array derived in Chapter 6. However, this array is also far slower than the partitioned 2-D array, which we would like to use to fold a significant fraction of smaller reads. Our algorithm predicts an optimal speedup of 6.72× using four instantiations each of the partitioned 2-D and unpartitioned 1-D arrays.
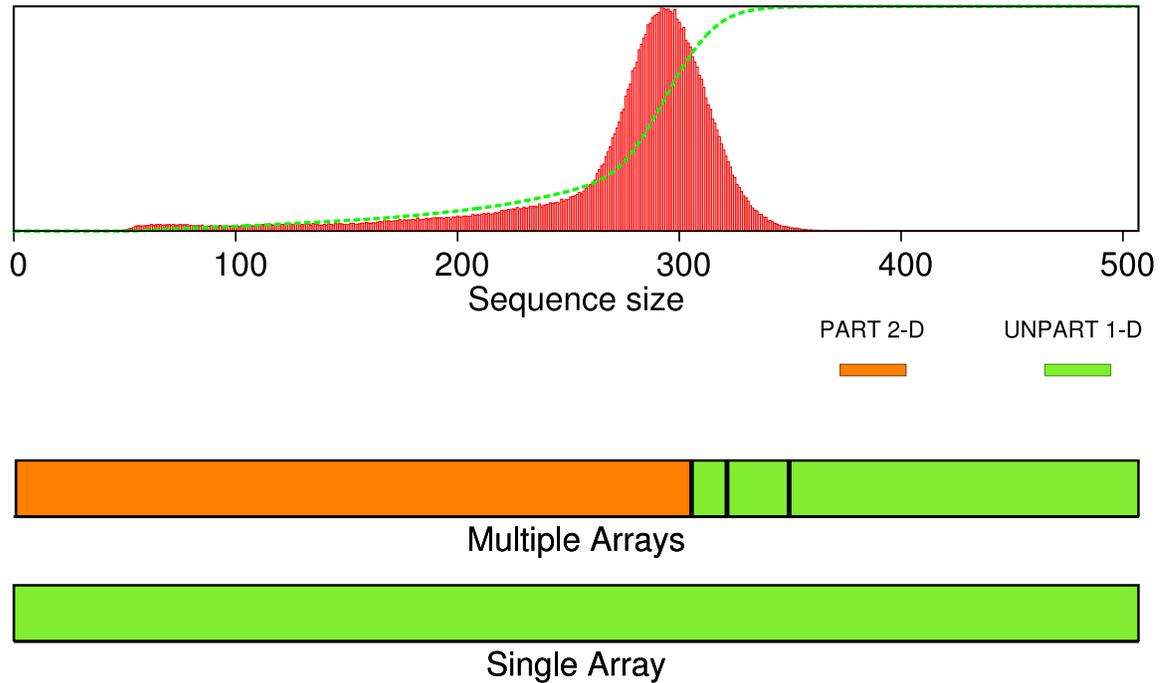
Figure 7.5: On the SRP000960 dataset we are able to achieve close to optimal speedup by reconfiguring between half the number of designs as the optimal solution. Our algorithm suggests the use of three unpartitioned 1-D arrays, and a single partitioned 2-D array.

Again, we are able to achieve substantial speedup using fewer designs. We used our program to specify the best solution when limited to selecting four designs. Figure 7.5 illustrates the distribution of read lengths and the four arrays used in our experiment. Notably, three of the selected arrays are instantiations of the unpartitioned 1-D array, which together process just 22% of the reads in the dataset. Our algorithm justifiably focuses on the small number of long reads as this is the bottleneck in the entire computation.

Our program predicts a 6.37× speedup using the four designs. The single unpartitioned 1-D array of size 507 bases takes 653.60 seconds on our FPGA system to fold all reads. Reconfiguring between the four designs suggested by our program, we performed the same computation in 103.93 seconds, achieving a 6.29× speedup. The same dataset could be folded in $1,331.83$ seconds on our dual core baseline. Our reconfigured solution is 12.81× faster.

7.4   Related Work

As far as we are aware, past work on space-time analysis only finds a single optimal array; our's is the first to take advantage of runtime reconfiguration to select multiple arrays optimal for different input subsets.

Runtime reconfiguration has been successfully used on FPGAs to improve the execution time of applications. Runtime customization [146] responds to input to produce optimized designs, for example using constant propagation, precision variation [21], and branch optimization [141]. In the case of constant propagation and precision variation, the circuit can be simplified when the input data item is known, improving performance and possibly reducing area requirements. In the case of branch optimization, a frequently executed branch case is optimized based on typical execution profiles. Specific application accelerators that use these techniques include SAT solvers [164], sequence alignment [121] and Viterbi decoding [143]. Our work is intended to be generally applicable to any application specified as a recurrence. Runtime customization techniques can be used to build systolic arrays for the candidate list used by our dynamic programming algorithm.

Our array selection algorithm can be applied to improve the performance of existing accelerators through runtime reconfiguration. For example, a recently published accelerator for the Viterbi recurrence used in motif finding [35] expressly notes that runtime reconfiguration based on the input model length is required for acceptable performance, though the authors do not give an algorithm. In addition, it may be possible to select alternate, less resource-intensive designs for smaller input sequences when it can be guaranteed that the sequence contains only a single copy of the input model.

7.5 Conclusion

Exploiting the power of reconfiguration can result in significant performance improvements for systolic array implementation of recurrences. We have described systematic algorithms to select array designs and reconfiguration points so as to realize maximum performance. We have validated our approach empirically and have obtained significant speedups over non-reconfiguring hardware for application of the Nussinov RNA folding algorithm to short nucleic acid sequences.

Two hurdles to realizing performance improvement using FPGA reconfiguration are the overhead of sequence sorting, and the runtime to compute the optimal array designs. As mentioned previously, sorting may be done during the data generation stage. Dealing with a database of sequences of varying length is also a challenge in a multi-core software implementation as the workload must be equally distributed among the available compute nodes. One solution is to have a runtime master feed compute nodes RNA sequences and equally distribute the workload. An alternative solution is to sort the sequences by length and build smaller, equal sized sub-databases that can be processed independently.

Our program to select the optimal array designs runs in 30-50 seconds on our software baseline. We believe this runtime can be reduced significantly using a number of enhancements. Currently, we use the python interpreter to calculate the runtime cost of each array. This was done to increase flexibility and ease of use, but also results in a significantly poorer performance than a C-only implementation. We will also be able to reduce runtime by limiting the choice of available array instantiations.

One important direction for further study is whether one can systematically exploit reconfiguration to improve performance based on criteria other than input size. For example, DNA sequence comparison algorithms in bioinformatics exhibit performance that is sensitive to the percentages of different DNA bases in each input [103]. A second direction for improvement would couple our algorithms to a tool that automates formal synthesis and exploration of the space of possible array designs, so that alternative families like those we built for Nussinov can rapidly be generated for new computational problems of interest.

Chapter 8

Analysis and Acceleration of the Zuker RNA Folding Recurrence

Given our success in applying polyhedral analysis to Nussinov, we now turn to the Zuker RNA folding algorithm. The Zuker dynamic programming algorithm [166] computes the minimum free-energy secondary structure of an RNA. Unlike the Nussinov recurrence, which considers only base pairing, Zuker uses more accurate, experimentally derived thermodynamic models for structural features such as loops and stacks. The Zuker recurrence also makes use of experimentally derived energy parameters to more accurately compute the contributions of special-case loop structures.

The Zuker algorithm is more challenging to accelerate than Nussinov because of its large number of dependencies and their complexity in addition to its resource-intensive parameters. Fortunately, our application of polyhedral analysis to Nussinov suggests ways to tackle these challenges.

In Section 8.1, we commence our study by introducing the Zuker recurrence and summarizing the difficulty in building an efficient special-purpose accelerator. We then simplify the recurrence in Section 8.2 by applying polyhedral transformations used on Nussinov. Once we have a form more conducive to efficient array synthesis, we build and evaluate a special-purpose array in Section 8.3. Due to the resource-intensive nature of this algorithm, we only build a 1-D array in this work. Finally, in Section 8.4 we compare our approach to related work from literature and show the superiority of polyhedral analysis.

8.1 The Zuker Recurrence

To be relevant to biologists, we target the specific Zuker implementation in the popu-
lar RNAfold program of the Vienna RNA package [69]. The program implements the
recurrence first described by Zuker [166] but also includes a number of enhancements
such as the modeling of special-case loops and dangling ends. We have omitted these
details from our discussion for brevity[11]; however, all these computations have been
implemented in our accelerator, which yields results identical to the `fill_arrays`
function of Vienna RNA version 1.8.4.

For the RNA sequence S of length N, the Zuker algorithm recursively computes three
data variables W, V, and VBI defined over the domain $\mathcal{D} = \{\, i, j \mid 1 \leq i < j \leq N \,\}$.
Supporting energy functions computed via table lookup are shown in lower case.
Note that δ is a lookup table, as opposed to being a simple function for the Nussinov
algorithm.

$$W(i, j) \;\; = \;\; \min \begin{cases} W(i + 1, j) + b \\ W(i, j - 1) + b \\ V(i, j) + \delta(S_i, S_j) \\ \min_{i < k < j} \{W(i, k) + W(k + 1, j)\} \end{cases} \tag{8.1}$$

$$V(i, j) \;\; = \;\; \min \begin{cases} \infty & \textit{if } (S_i, S_j) \textit{ is not a base pair} \\ \\ eh(i, j) & \textit{otherwise} \\ V(i + 1, j - 1) + es(i, j) \\ VBI(i, j) \\ \min_{i < k < j - 1} \{W(i + 1, k) + W(k + 1, j - 1)\} + c \end{cases} \tag{8.2}$$

$$VBI(i, j) \;\; = \;\; \min_{i < i' < j' < j} \{\, V(i', j') + ebi(i, j, i', j') \,\}. \tag{8.3}$$

$V(i, j)$ is the energy of the optimal structure formed by subsequence $S_{i...j}$, with bases
S_i and S_j forming a base pair. This structure is formed by a decomposition into its
constituent loops. The recurrence exhaustively considers all loop structures closed

---

[11]We plan to publicly distribute the complete source code for our Zuker array in the near future.

147
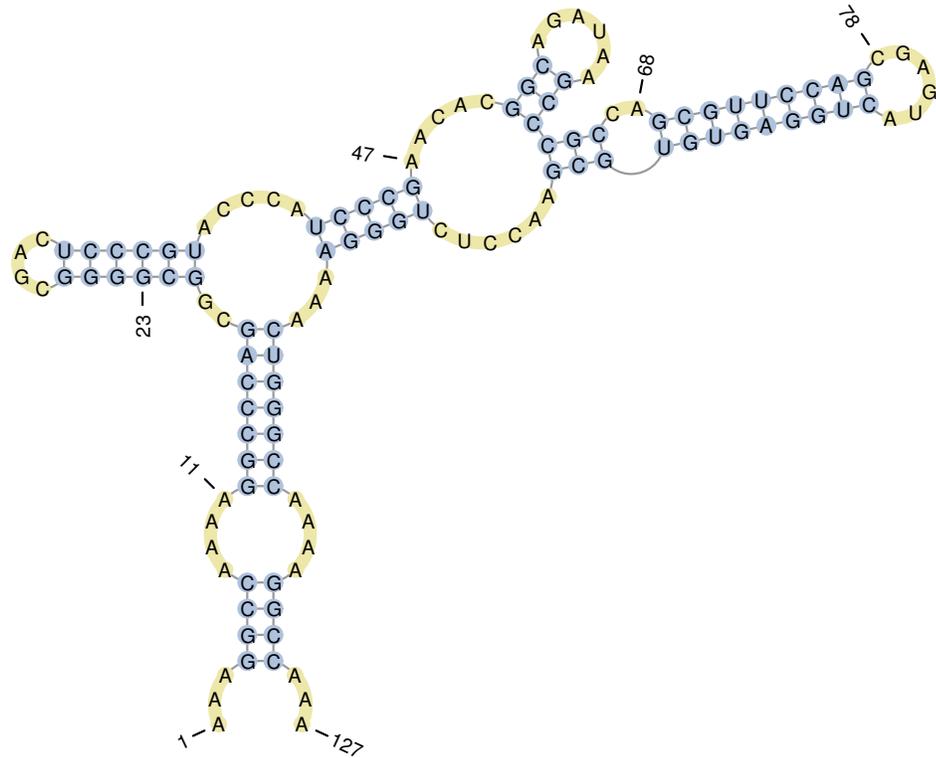
Figure 8.1: An example of an RNA folded into a secondary structure with free energy -53.90 kcal/mol. Types of structural features modeled by the Zuker folding algorithm include: dangling ends (1), internal loop (11), stack (23), multi-loop (47), bulge (68) and hairpin loop (78).

by every complementary base pair. The two bases at i and j may close a *hairpin loop* (*eh*), form part of a *stack* (*es*), close an *internal loop* or *bulge* (*VBI*), or be part of a *multi-loop* (*W*). See Figure 8.1 for examples of these features. The most likely scenario is selected using the minimization operation in Equation 8.2. If these two bases are not complementary, the energy score is set to infinity to signify an impossible structure. Note that Equation 8.1, used to compute the energy of multi-loops, has a structure identical to that of the Nussinov recurrence of Equation 1.1.

Equation 8.3 calculates the scores of internal loops and bulges using the scoring table *ebi* and is subject to the constraint $i' - i + j - j' > 2$. This equation also determines the time complexity of the algorithm, which is quartic in the length of the sequence. However, large internal loops are uncommon in nature, so implementations of Zuker
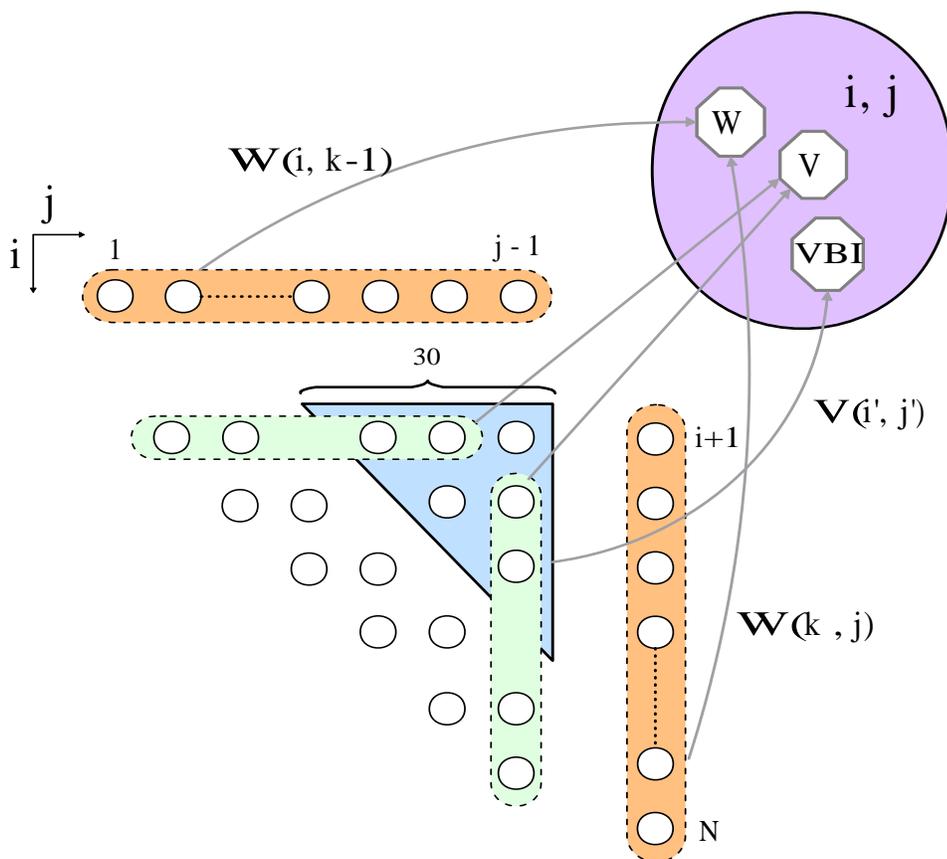
Figure 8.2: Long-range dependencies for the cell (i, j).

typically sacrifice accuracy for speed by limiting the size of internal loops to at most 30, reducing the overall time complexity to $\Theta(N^3)$, albeit with a large constant factor.

The computation domain for the recurrences is a triangle, as shown in Figure 8.2. Long-range dependencies of the enlarged cell (i, j) are depicted in dashed boxes. Dependencies for *VBI* are enclosed in the triangle, whose base and height are each at most 30 cells. Note that all cells (i, j) on a given anti-diagonal line $j - i = constant$ can be computed in parallel.

Finally, the minimum free energy $F(j)$ of the folded RNA subsequence $1 \ldots j$ is given by

$$F(j) = \min \begin{cases} F(j-1) \\ \min_{1 \leq i < j} \{V(i,j) + F(i-1)\}. \end{cases} \tag{8.4}$$

149

$F(N)$ is then the free-energy score of the entire molecule. As was the case for Nussinov, we do not accelerate the computation of the optimal structure itself, only its energy.

8.2 Parallelizing Zuker

Our first task is to simplify the Zuker recurrence to make it more amenable to parallelization. Notice that the final terms in Equations 8.1 and 8.2 are similar. Let $T(i,j)$ denote the quantity computed by the final term of Equation 8.1, i.e.,

$$T(i,j) = \min_{i<k<j} \left\{ W(i,k) + W(k+1,j) \right\} .$$ (8.5)

We can rewrite the final term in Equation 8.2 in terms of T:

$$\min \begin{cases} W(i+1,i+1) + W(i+2,j-1) + c \\ T(i+1,j-1) + c . \end{cases}$$ (8.6)

$W(i+1,i+1)$ is always ∞, so we can ignore the first case, allowing us to replace the reduction in Equation 8.2 with $T(i+1,j-1) + c$.

8.2.1 Handling Free Energy Scores

The energy tables in the Zuker algorithm are critical for accurately modeling the structural features of an RNA secondary structure. Unfortunately, they are also challenging to implement on an accelerator. We will now summarize the components of the energy tables *eh*, *es*, *c*, and *ebi* from recurrence 8.1-8.3; they are described in detail elsewhere [167][12].

The free energy for hairpin loops $eh(i,j)$ consists of contributions from three loop features:

---

[12]A succinct guide to the energy functions is also available online at `http://www.bioinfo.rpi.edu/zukerm/seqanal/`.

- The first energy component is dependent on the size of the loop, $j - i - 1$. Loops of size ≤ 30 have precomputed energies stored in a table. The energy of larger loops is computed using a mathematical model. We use a single table of precomputed energies of 256 entries, each with a precision of 10 bits.

- The effect of terminal mismatched pairs (a single energy value) closing a hairpin loop is considered.

- Finally, some hairpin loops are treated as special cases. Free energies of some hairpin loops of size 4 have been experimentally derived and are made available in a lookup table. This is stored as a sparse table with 30 9-bit entries.

The free energy contribution $es(i, j)$ is defined as a table $stacking(S_i, S_j, S_{i+1}, S_{j-1})$, with 49 10-bit entries. Multi-loops are scored using a constant c. The energy term $ebi(i, j, i', j')$ is a function used to score both internal loops and bulges. We will describe it in detail in the next section.

Simplifying the Internal Loop Computation

The internal loop and bulge structure computation in variable *VBI* presents two challenges. First, every cell (i, j) depends on a large triangular section of cells. Second, because the four indices (i, i', j, j') do not define a rectangular domain, an array generated using the linear polyhedral framework will have a suboptimal schedule.

Fortunately there exists an algorithmic technique to reduce the complexity of computing *VBI without* affecting its value. Lyngsø observed that the energy function *ebi* is not arbitrary but depends on the size of a loop [104], enabling him to simplify the internal loop computation to cubic time complexity. Lyngsø's optimization did not yield a speedup in software because standard implementations had already limited the size of internal loops to 30; however, in this work, we use the transformation to reduce the complexity of the recurrences to make them more amenable to hardware acceleration.

We now give a brief overview of the bulge and internal loop energies. We define two distinct energy functions: *ebb* for bulges and *ebi* for internal loops. Let $k =$

$i' - i + j - j' - 2$ be the size of a bulge or an internal loop. The energy function ebb can be split into contributions from three features:

- The size k of the loop, which is stored in a table with 30 10-bit entries.

- Stacking energies of the interior (S_i, S_j) and exterior $(S_{i'}, S_{j'})$ base pairs with the nearest unpaired bases. These energies are stored in a table with 49 10-bit entries. The stacking energy term is added only if the bulge loop is of size 1.

- If the loop size is greater than one, a constant penalty term is added if the interior or exterior base pairs are $\{A, U\}$ or $\{G, U\}$.

The bulge energy function is thus of the form:

$$ebb(i, j, i', j') = stacking(S_i, S_j, S_{i'}, S_{j'}) + terminalAUGU(S_i, S_j) + \\ terminalAUGU(S_{i'}, S_{j'}) + ebbsize(k) \ . \tag{8.7}$$

The energy function ebi can be similarly split with an additional contribution from an asymmetry component of the loop.

$$ebi(i, j, i', j') = ebistacking(S_i, S_j) + ebistacking(S_{i'}, S_{j'}) + \\ ebisize(k) + ebiasymmetry(|(i' - i - 1) - (j - j' - 1)|) \ . \tag{8.8}$$

Consider the internal loop closed by base pairs $(S_{i'}, S_{j'})$ and (S_{i+1}, S_{j-1}) in Figure 8.3. We say the loop is *lopsided* because the unpaired region on the left has two more bases than the region on the right. The asymmetry energy component is a penalty dependent on the loop's lopsidedness. Because *ebisize* is constant for a fixed loop size, we can reformulate the internal loop calculation to aggregate all dependencies $V(i', j')$ that have $i' - i + j - j' = constant$. We can build from smaller to larger subproblems that have identical lopsidedness; i.e., whenever i is increased by one base, j is also decreased by one. This idea is illustrated in Figure 8.3 and is the key to decreasing the time complexity of the internal loop computation.
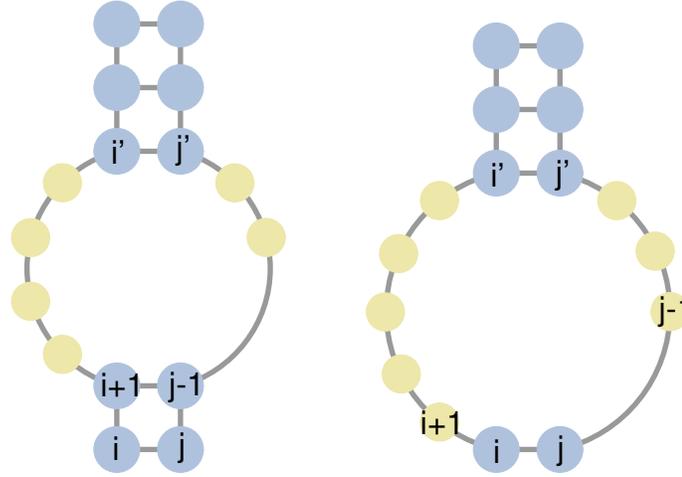
Figure 8.3: Difference in internal loop energy as the exterior base pair changes from $(i+1, j-1)$ to (i, j). Using Equation 8.8, we see that the energy for the second loop is $ebi(i+1, j-1, i', j') - ebistacking(S_{i+1}, S_{j-1}) + ebistacking(S_i, S_j) - ebisize(l-2) + ebisize(l)$. Here $l = i' - i + j - j' - 4$ is the loop size.

We break up the computation of VBI in recurrence 8.2 to use distinct terms for bulges and internal loops, leading to the following modification.

$$V(i,j) = \min \begin{cases} \vdots \\ \min_{i < k < j-1} \{ V(i+1, k) + ebb(i, j, i+1, k) \} \\ \min_{i+1 < k < j} \{ V(k, j-1) + ebb(i, j, k, j-1) \} \\ \\ VBI(i, j, 5) \\ \vdots \end{cases} \qquad (8.9)$$

The first two min reduction terms in the new recurrence compute the score of the best left and right bulges, while the final term computes the score of the best internal loop of size 5 or greater. Current implementations of the Zuker recurrence use experimentally computed score tables for internal loops of size 4 or less; we use six lookup tables provided by Vienna RNA to handle these special cases (not shown).

Let $VBI(i, j, k)$ be the score of the best internal loop of size $\geq k$ that has an exterior base pair (S_i, S_j). We compute it using the helper $VBI'(i, j, k)$, which is the score of the best internal loop of size k. Note that, strictly speaking, $VBI'(i, j, k)$ includes

loops that are left $(i + 1)$ and right $(j - 1)$ bulges of size k. However, these bulges become part of internal loops at larger values of k. We use the following recurrence to compute internal loop energies of all loops of size ≥ 5.

$$
VBI(i, j, k) = \min \begin{cases}
VBI(i, j, k + 1) & \text{if } k \geq 5 \\
VBI'(i + 1, j - 1, k - 2) + \\
\quad ebistacking(S_i, S_j) - \\
\quad ebistacking(S_{i+1}, S_{j-1}) + \\
\quad ebisize(k) - ebisize(k - 2)
\end{cases}
$$

$$
VBI'(i, j, k) = \min \begin{cases}
V(i + 1, j - k - 1) + & \text{if } k = 1 \\
\quad ebi(i, j, i + 1, j - k - 1) \\
V(i + k + 1, j - 1) + \\
\quad ebi(i, j, i + k + 1, j - 1) \\
\\
V(i + 1, j - k - 1) + & \text{if } k = 2 \\
\quad ebi(i, j, i + 1, j - k - 1) \\
V(i + k + 1, j - 1) + \\
\quad ebi(i, j, i + k + 1, j - 1) \\
V(i + 2, j - 2) + \\
\quad ebi(i, j, i + 2, j - 2) \\
\\
VBI'(i + 1, j - 1, k - 2) + & \text{if } k \geq 3 \\
\quad ebistacking(S_i, S_j) - \\
\quad ebistacking(S_{i+1}, S_{j-1}) + \\
\quad ebisize(k) - ebisize(k - 2) \\
V(i + 1, j - k - 1) + \\
\quad ebi(i, j, i + 1, j - k - 1) \\
V(i + k + 1, j - 1) + \\
\quad ebi(i, j, i + k + 1, j - 1).
\end{cases}
$$

Note how, when aggregating at $VBI'(i, j, k)$ from $VBI'(i+1, j-1, k-2)$, we subtract the energy contribution of $ebisize(k - 2)$ and add the contribution of $ebisize(k)$. This is possible because the energy components are the same for all interior pairs $V(i', j')$ such that $k = i' - i + j - j' - 2$, forming loops of the same size. Importantly, as we

154

increase the loop size by two, we ensure that the lopsidedness remains unchanged, since an extra base is added to both sides of the loop. We can further simplify this recurrence by expanding *ebi* and gathering common terms.

$$
VBI(i, j, k) = \min
\begin{cases}
VBI(i, j, k+1) + ebistacking(S_i, S_j) & \text{if } k = 5 \\
VBI'(i+1, j-1, k-2)+ \\
\quad ebistacking(S_i, S_j) + ebisize(k) \\
\\
VBI(i, j, k+1) & \text{if } k \geq 6 \\
VBI'(i+1, j-1, k-2) + ebisize(k)
\end{cases}
\tag{8.10}
$$

$$
VBI'(i, j, k) = \min
\begin{cases}
V(i+1, j-k-1)+ & \text{if } k = 1 \\
\quad ebistacking(S_i, S_{j-k})+ \\
\quad ebiasymmetry(1) \\
V(i+k+1, j-1)+ \\
\quad ebistacking(S_{i+k}, S_j)+ \\
\quad ebiasymmetry(1) \\
\\
V(i+1, j-k-1)+ & \text{if } k = 2 \\
\quad ebistacking(S_i, S_{j-k})+ \\
\quad ebiasymmetry(2) \\
V(i+k+1, j-1)+ \\
\quad ebistacking(S_{i+k}, S_j)+ \\
\quad ebiasymmetry(2) \\
V(i+2, j-2)+ \\
\quad ebistacking(S_i, S_j)+ \\
\quad ebiasymmetry(0) \\
\\
VBI'(i+1, j-1, k-2) & \text{if } k \geq 3 \\
V(i+1, j-k-1)+ \\
\quad ebistacking(S_i, S_{j-k})+ \\
\quad ebiasymmetry(k) \\
V(i+k+1, j-1)+ \\
\quad ebistacking(S_{i+k}, S_j)+ \\
\quad ebiasymmetry(k)
\end{cases}
\tag{8.11}
$$

155

The terms $V(i+1, j-k-1)$ and $V(i+k+1, j-1)$ are affine dependencies that must be pipelined.

In order to facilitate the uniformization of affine dependencies, we perform middle serialization as described in Section 4.2 for the Nussinov algorithm. We pipeline long-range dependencies including the RNA sequence, the W terms in recurrence 8.5, and the V terms in recurrence 8.11.

We must also handle the two bulge terms in recurrence 8.9. To serialize the two terms, we define $VBB(i, j, k)$ to be the score of the best bulge of size $\geq k$. We can rewrite the bulge computation as

$$VBB(i, j, k) = \min \begin{cases} VBB(i, j, k+1) & \text{if } k \leq j - i - 2 \\ V(i+1, j-k-1, 1) + \\ \quad ebb(i, j, i+1, j-k-1) \\ V(i+k+1, j-1, 1) + \\ \quad ebb(i, j, i+k+1, j-1) \end{cases} \tag{8.12}$$

Now we can build pipelines for the terms $V(i+1, j-k-1)$ and $V(i+k+1, j-1)$. This can be done by pipelining the first term along the basis vector $\begin{bmatrix} 0 \\ -1 \\ -1 \end{bmatrix}$ with a constant initialization vector $\begin{bmatrix} 1 \\ -2 \\ 0 \end{bmatrix}$ and the second term along $\begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}$ with the initialization vector $\begin{bmatrix} 2 \\ -1 \\ 0 \end{bmatrix}$.

The transformed Zuker recurrence equations are defined over the domain $\mathcal{D} = \{\, i, j, k \mid 1 \leq i \leq N; i \leq j \leq N; 1 \leq k \leq \max\{\ \min\{30, j - i - 2\}, \lceil \frac{j-i}{2} \rceil \,\} \,\}$. The final uniform system of recurrences for Zuker is given below.

$$W(i, j, k) = \min \begin{cases} W(i+1, j, k) + b & \text{if } k = 1 \\ W(i, j-1, k) + b \\ V(i, j, k) + \delta(PA(i, j, k), PB(i, j, k)) \\ T(i, j, k) \end{cases} \tag{8.13}$$

156

$$V(i,j,k) = \min \begin{cases} eh(& \text{if } k = 1 \text{ and} \\ \quad j - i - 1, & \text{(PA(i, j, k),} \\ \quad PA(i,j,k), PB(i,j,k), & \text{PB(i, j, k))} \\ \quad PA(i+1,j,k), PB(i,j-1,k), & \text{is a base pair} \\ \quad PA(i+1,j,k), PA(i+2,j,k), \\ \quad PA(i+3,j,k), PA(i+4,j,k), \\ \quad PA(i+5,j,k), PA(i+6,j,k) \\ \quad) \\ V(i+1,j-1,k)+ \\ \quad stacking(\\ \quad\quad PA(i,j,k), PB(i,j,k), \\ \quad\quad PA(i+1,j,k), PB(i,j-1,k), \\ \quad\quad) \\ VBB(i,j,k) \\ VBI(i,j,k) \\ T(i+1,j-1,k) + c \end{cases} \tag{8.14}$$

$$PA(i,j,k) = \begin{cases} S_i & \text{if } j - i = 1 \\ PA(i,j-1,k) & \text{if } k = 1 \end{cases} \tag{8.15}$$

$$PB(i,j,k) = \begin{cases} S_j & \text{if } j - i = 1 \\ PB(i+1,j,k) & \text{if } k = 1 \end{cases} \tag{8.16}$$

$$T(i,j,k) = \min \begin{cases} T(i,j,k+1) & \text{if } 2k \le j - i \\ PW_1(i,j,k) + PW_2(i,j,k) \\ PW_3(i,j,k) + PW_4(i,j,k) \end{cases} \tag{8.17}$$

$$PW_1(i,j,k) = \begin{cases} PW_3(i,j,k) & \text{if } 2k = j - i \\ PW_1(i,j-1,k) & \text{if } 2k < j - i \end{cases} \tag{8.18}$$

$$PW_2(i,j,k) = \begin{cases} W(i+2,j,k) & \text{if } k = 1 \\ PW_2(i+1,j,k-1) & \text{if } 2k \le j - i \end{cases} \tag{8.19}$$

$$PW_3(i,j,k) = \begin{cases} W(i,j-1,k) & \text{if } k = 1 \\ PW_3(i,j-1,k-1) & \text{if } 2k \le j - i \end{cases} \tag{8.20}$$

$$PW_4(i,j,k) = \begin{cases} PW_2(i,j,k) & \text{if } 2k = j - i \\ PW_4(i+1,j,k) & \text{if } 2k < j - i \end{cases} \tag{8.21}$$

$$
VBI(i,j,k) \;=\; \min
\begin{cases}
\begin{aligned}
&VBI(i,j,k+1)+ \\
&\quad ebistacking(\\
&\qquad PA(i,j,k), PB(i,j,k), \\
&\qquad PA(i+1,j,k), PB(i,j-1,k) \\
&\qquad\qquad)
\end{aligned} & \text{if } k=1 \\[2mm]
VBI(i,j,k+1) & \text{if } 2 \le k \le 4 \\[2mm]
\begin{aligned}
&VBI(i,j,k+1) \\
&VBI'(i+1,j-1,k-2)+ \\
&\quad ebisize(k)
\end{aligned} & \begin{aligned}&\text{if } k \ge 5 \text{ and} \\ &k \le \min\{30, \\ &\qquad j-i-2\}\end{aligned}
\end{cases}
\tag{8.22}
$$

$$
VBI'(i,j,k) \;=\; \min
\begin{cases}
\begin{aligned}
&PVI_1(i,j,k)+ \\
&\quad ebiasymmetry(1) \\
&PVI_2(i,j,k)+ \\
&\quad ebiasymmetry(1)
\end{aligned} & \text{if } k=1 \\[3mm]
\begin{aligned}
&PVI_1(i,j,k)+ \\
&\quad ebiasymmetry(2) \\
&PVI_2(i,j,k)+ \\
&\quad ebiasymmetry(2) \\
&V(i+2,j-2,k-1)+ \\
&\quad ebistacking(\\
&\qquad PA(i+2,j,k-1), PB(i,j-2,k-1) \\
&\qquad PA(i+1,j,k-1), PB(i,j-1,k-1), \\
&\qquad\qquad)+ \\
&\quad ebiasymmetry(0)
\end{aligned} & \text{if } k=2 \\[3mm]
\begin{aligned}
&VBI'(i+1,j-1,k-2) \\
&PVI_1(i,j,k)+ \\
&\quad ebiasymmetry(k) \\
&PVI_2(i,j,k)+ \\
&\quad ebiasymmetry(k)
\end{aligned} & \begin{aligned}&\text{if } k \ge 3 \text{ and} \\ &k \le \min\{30, \\ &\qquad j-i-2\}\end{aligned}
\end{cases}
\tag{8.23}
$$

158

$$
PVI_1(i,j,k) = \begin{cases} V(i+1,j-2,k)+ & \text{if } k=1 \\ \quad ebistacking(\\ \qquad PA(i+1,j,k), PB(i,j-2,k), \\ \qquad PB(i,j-1,k), PA(i,j,k) \\ \qquad) \\ PVI_1(i,j-1,k-1) & \text{if } k \le \min\{30, j-i-2\} \end{cases} \tag{8.24}
$$

$$
PVI_2(i,j,k) = \begin{cases} V(i+2,j-1,k)+ & \text{if } k=1 \\ \quad ebistacking(\\ \qquad PA(i+2,j,k), PB(i,j-1,k), \\ \qquad PB(i,j,k), PA(i+1,j,k) \\ \qquad) \\ PVI_2(i+1,j,k-1) & \text{if } k \le \min\{30, j-i-2\} \ . \end{cases} \tag{8.25}
$$

$$
VBB(i,j,k) = \min \begin{cases} VBB(i,j,k+1)+ & \text{if } k=1 \\ \quad terminalAUGU(\\ \qquad\qquad\quad PA(i,j,k), \\ \qquad\qquad\quad PB(i,j,k) \\ \qquad\qquad) \\ V(i+1,j-2,k) + ebbsize(k)+ \\ \quad stacking(\\ \qquad PA(i,j,k), PB(i,j,k), \\ \qquad PA(i+1,j,k), PB(i,j-2,k), \\ \qquad) \\ V(i+2,j-1,k) + ebbsize(k)+ \\ \quad stacking(\\ \qquad PA(i,j,k), PB(i,j,k), \\ \qquad PA(i+2,j,k), PB(i,j-1,k), \\ \qquad) \\ \\ VBB(i,j,k+1) & \text{if } k \le \min\{30, j-i-2\} \\ PVB_1(i,j,k) + ebbsize(k) \\ PVB_2(i,j,k) + ebbsize(k) \end{cases} \tag{8.26}
$$

$$PVB_1(i,j,k) = \begin{cases} V(i+1,j-2,k)+ & \text{if } k=1 \\ \quad terminalAUGU(& \\ \qquad\qquad PA(i+1,j,k), & \\ \qquad\qquad PB(i,j-2,k) & \\ \qquad) & \\ PVB_1(i,j-1,k-1) & \text{if } k \le \min\{30, j-i-2\} \end{cases} \qquad (8.27)$$

$$PVB_2(i,j,k) = \begin{cases} V(i+2,j-1,k)+ & \text{if } k=1 \\ \quad terminalAUGU(& \\ \qquad\qquad PA(i+2,j,k), & \\ \qquad\qquad PB(i,j-1,k) & \\ \qquad) & \\ PVB_2(i+1,j,k-1) & \text{if } k \le \min\{30, j-i-2\} \end{cases} \qquad (8.28)$$

8.2.2 1-D Zuker Array

Due to the resource-intensive nature of the system of recurrences in the previous section, we decided not to build full-size arrays for Zuker. Instead we use the procedure from Chapter 6 to map our array on to a linear array of physical processing elements. We allocate a 1-D set of processors along the recurrence's k dimension—one at each integral point on the k axis. The schedule and allocation of this array are given by

$$\tau(i,j,k) = -k - 2i + Nj$$
$$\pi(i,j,k) = [k] \ .$$

The schedule satisfies all the dependence constraints of the recurrence. To be conflict-free according to Equation 6.1, the greatest common divisor of N and 2 must be one; i.e., we can only build arrays for sequences of odd length. Even-length sequences are padded using a special character.

Each processor in the array executes a rectangular domain of points $\{\ i,j \mid 1 \le i \le N; 1 \le j \le N \ \}$. Our array assigns ∞ to the variables when computing outside the domain of the Zuker recurrence. An RNA of length N can be folded in $\tau(1,N,1) - \tau(3,3,1) = N^2 - 3N + 4$ clocks, though useful work is done on only 50% of the clock cycles.
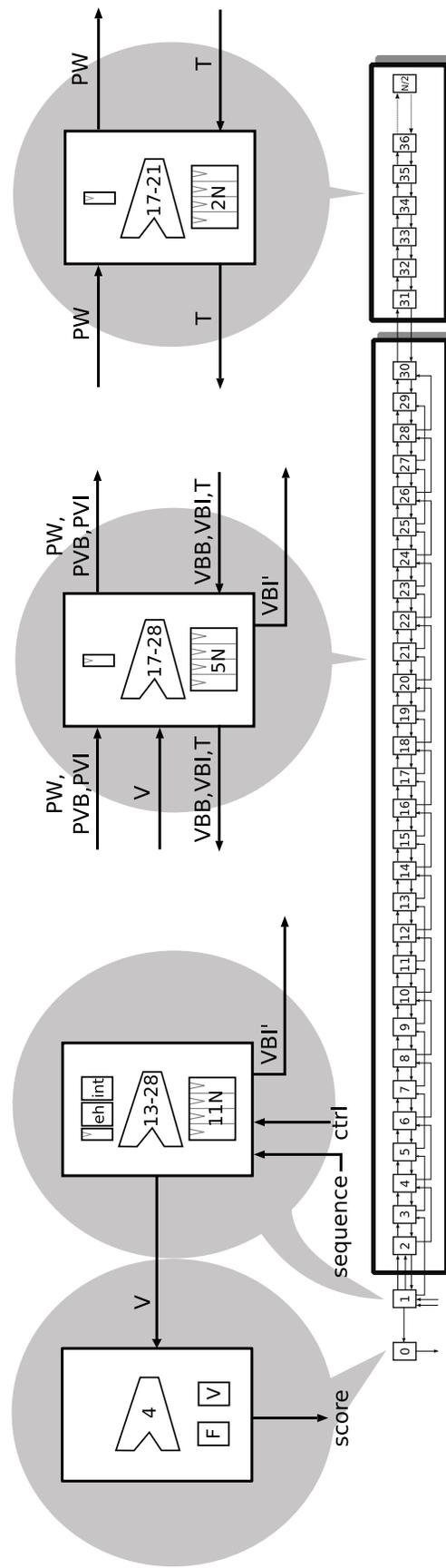
Figure 8.4: Overview of the 1-D Zuker array, which uses four main PE types. Equation numbers (from the previous section) computed by a PE are shown in the ALU block; energy functions stored in block RAMs and registers are shown at the top of each PE; and the growth of delay registers as a function of RNA length is shown in the bottom block.

A high-level overview of our array is shown in Figure 8.4. It is instructive to study the characteristics of the array using the system of recurrences in Equations 8.13-8.28. First, all communication between PEs is limited to the three adjacent neighbors. The PE placed at $k = 1$ is the most resource-intensive in the array, since it is the only one to instantiate the compute-intensive variables V and W defined at $k = 1$. Note that only this PE holds a copy of the memory-intensive hairpin (eh) and internal loop energy (ebi) tables used to compute V and W. Similarly, the stacking energy tables ($stacking$ and $ebistacking$) are required only at PE 1. Because a PE k always processes internal loops of a fixed size k, we can distribute one copy of the size ($ebbsize$ and $ebisize$) and asymmetry ($ebiasymmetry$) energy tables across the entire array. In contrast, existing work (see Section 8.4) requires a copy of all energy functions at each PE and must implement all computation in Equations 8.13-8.28 in every PE in the array, greatly limiting the number of PEs that can be instantiated on modern FPGAs. This limitation is a result of placing PEs along the j dimension of the recurrence. Polyhedral analysis allowed us to identify this issue and place PEs on the k dimension for a more resource-efficient implementation. This is also the reason we decided not to build full-size arrays for the Zuker algorithm.

Since internal loops are limited in size to 30, data variables VBB, PVB_1, PVB_2, VBI, VBI', PVI_1, and PVI_2 from Equations 8.22-8.28 need be calculated only on PEs 1-30. All subsequent PEs in the array implement only the T and PW_1-PW_4 computations, the latter four being simple data pipelines. As a result, there are fewer dependencies in these PEs, so local memory to store intermediate values is greatly reduced.

We use a single processor (PE 0 in Figure 8.4) to execute Equation 8.4 sequentially after every column j of $V(i, j)$ has been buffered. This adds a latency of N clocks to the folding computation.

Input sequence data is sent to PE 1 at regular intervals determined by the schedule. The V table values, which are required for the minimum free-energy computation, are always available at PE 1 (arrays instantiated along any other axis would have multiple sources). The minimum free-energy score is available at PE 0.

Finally, as N increases beyond 60, the size of the array tends to $\frac{N}{2}$. We can fold an RNA of length N with just half the processors needed by arrays placed along the i or the j axes; moreover, only the cheapest PE needs to be replicated.

8.3 Evaluation

We have implemented our Zuker accelerator in VHDL and built them for the Xilinx Virtex 4 LX100-12 FPGA. The architecture of an RNA folding engine contains a sequence module that buffers input sequences and feeds the systolic array. Purpose-built controllers feed the sequence and initialize the array at well-defined intervals as determined by the schedule. The array and its controller are in an isolated clock domain that runs faster than the sequence module and FPGA I/O blocks.

Although the transformations we have applied to the RNA folding algorithms are provably correct, verifying the accuracy of an implementation is a challenging task. We first wrote a C loop implementation of each transformed system of recurrences and confirmed identical output to the RNAfold program. The VHDL implementation of every energy function was first testbenched to ease debugging. All arrays were simulated in ModelSim on hundreds of randomly generated RNAs and verified against software output for an exact match. Finally, we validated correctness by computing scores of tens of thousands of RNAs using our FPGA platform and checking for an exact match to unmodified software.

8.3.1 Techniques for Synthesis After Polyhedral Analysis

We now give a few techniques that can be generally applied to synthesize optimized arrays on FPGAs after polyhedral analysis. Any dependence in a recurrence has to be sent from the source to the sink PE after a fixed delay that is computed using the schedule. These delays are usually programmed as shift registers with a global reset. Using a reset, however, can adversely impact the resource usage and speed of the implementation on a Xilinx FPGA. If these registers are coded with a reset, synthesis tools use an entire FPGA LUT and its associated flip flop to realize a

single-bit shift; without reset, a 16-bit shift register can be realized with these same resources. We can always remove the reset signal on dependency delay registers so long as necessary initialization conditions are programmed at the boundaries of the computation domain. This optimization is extremely useful, allowing us to fold a sequence 45 bases longer than is otherwise possible on the Zuker array.

Since our design uses very little on-chip memory for tables to support computation, a large fraction of the block RAM memories are unused. We can use these memories as FIFOs to implement the dependency delays. Our PE implementation is parametrized first to use all the on-chip block RAMs and only then to use LUTs to implement delays. With this optimization, we are able to increase by 75 bases the size of the largest RNA folded on the Zuker array.

To achieve an acceptable clock frequency for our design, the computation in a PE must be pipelined. Unfortunately, the techniques we have used assume that an entire iteration is executed in a single clock cycle. The schedule we have selected, however, does instantiate a large number of delays on the majority of dependencies. In Section 5.5 we suggested the use of retiming to automatically pipeline computation in a PE. We demonstrated a significant performance boost using this technique for the several Nussinov arrays we derived in Chapters 4 and 6. We therefore coded our Zuker array to allow the synthesis tools to take advantage of retiming. For dependency delays realized using block RAM memories, we always ensure that a small fraction of the delay is realized using registers, to allow retiming. Without retiming, our Zuker array synthesized to between 60-70 MHz; with retiming, the tools achieve a clock frequency of 130-166 MHz. This clock rate compares well with the manually optimized Zuker array mentioned in Section 8.4 that runs at 135 MHz on the same FPGA family.

8.3.2 Results for the Zuker Array

To ensure matching results to the software baseline, we have implemented additional data variables to calculate dangling energies as detailed in the Vienna RNA package. We use ten block RAM memories for the empirical loop energy scores in PE 1; all other energies are implemented in distributed memory. We have parametrized the

Zuker 1-D array to either output V values for traceback in software or send just the minimum free-energy score. The latency of our implementation is

$$9 + (N^2 - 3N + 4) + N = N^2 - 2N + 13 \qquad (8.29)$$

clock cycles. Using just 3 bits per base, the input data rate is very low, approximately $3N$ bits per N^2 clocks.

We built our array for the Xilinx Virtex 4 LX100-12 FPGA using SmartXplorer to explore different build strategies. The results in this section are all from experiments run in real hardware on our FPGA system. To confirm accuracy of our implementation, we built arrays to fold sequences of length 121, 251, 261, and 273. We folded 10,000 randomly generated RNAs of each size and compared their minimum free-energy scores to those of RNAfold. The two matched exactly. We also folded 1090 sequences of length 99 bases from the Rfam database on our array and confirmed matching scores to those of RNAfold.

For our software baseline, we ran RNAfold on the dual core 3 GHz Intel Core 2 Duo processor with 4 MB cache. We used gcc 4.4.0 with compilation flags *-O3 -march=nocona -fomit-frame-pointer* and measured only the time spent in computing the Zuker recurrence; traceback and I/O time were excluded. Section 4.4 describes both the software and hardware systems used in this performance comparison in detail.

We built an array with 136 PEs clocked at 130 MHz to fold RNAs of length 273 bases. This is the largest RNA that can be folded by our array on the given FPGA device; it subsumes most of the range of RNA sizes typically folded by biologists. Table 8.3.2 lists the resource usage of each PE type as reported by the synthesis tool. PE 1 is the most expensive processor in the array. We implemented all of its delays using logic (SRL16); block RAMs are used to implement the energy functions. The empirical loop energy calculation in this PE is the critical path of the design; without it, the array clocks at 170 MHz.

PEs 2 to 30 are less resource-intensive but still use 5 block RAMs to implement the delay registers. We use two implementations for PEs 31 to $\frac{N}{2}$. Version A implements all delay registers using block RAMs. Once we have run out of these memories, version

Table 8.1: Area report of processors in our array. The three rows represent number of LUTs used as shift registers for delays, number of LUTs for arithmetic, and number of block RAMs.

	Controller	PE0	PE1	PE2-30	PE31-N/2 Ver. A	PE31-N/2 Ver. B
SRL16s	20	17	1842	0	0	544
LUTs	1127	273	2426	762	276	136
BRAMs	0	3	10	5	2	0

B, the processor that is used the most in our design, implements delay registers using logic. This PE uses just 16% of the logic resources consumed by PE 1 and requires no block RAMs. This strong contrast demonstrates why we are able to fit far more processors on an FPGA device than Dou et al.; it is the main reason for our array's superior performance. After place and route, 99% of the block RAMs and 92% of the slices are used.

To compare array performance to the software baseline, we generated 100,000 random RNAs of length 273 and folded them both in hardware and in software. Our array, running in hardware, took 57.14 seconds, which almost exactly equals the runtime predicted by Equation 8.29. The baseline system performed the same computation in $5,894.44$ seconds on a single core and $2,950.07$ seconds on two cores; our array is $103.2\times$ faster than the single core and $51.6\times$ faster than the dual core CPU.

Table 8.2: Performance of the Zuker design folding 1 million randomly generated RNAs of length 267 on a multi-FPGA system. The hardware arrays were synthesized at 130 MHz.

Design	Runtime (secs)	Dual core speedup
Dual core software	$28,246.00$	$1\times$
1 card 2 FPGA	283.35	$99.69\times$
2 card 4 FPGA	142.11	$198.76\times$
3 card 6 FPGA	94.88	$297.70\times$
4 card 8 FPGA	72.10	$391.76\times$
5 card 10 FPGA	57.24	$493.47\times$
6 card 12 FPGA	47.85	$590.30\times$

For our final demonstration we did a workstation-to-workstation performance comparison. Our FPGA system is a general-purpose workstation with up to two PCI-X cards. Each card holds two Xilinx Virtex 4 LX100-12 FPGAs. Thus far, we have compared performance of a single FPGA against a dual core processor to do a sockets-to-sockets

study. It is also instructive to measure performance of the Zuker accelerator on all four FPGAs to get an idea of the performance of a single workstation accelerator.

We built our Zuker accelerator on both FPGAs of a single card with the necessary interfacing logic provided by Exegy Inc. The dual FPGA solution is able to fold RNAs up to 267 bases. The RNA database is streamed to both FPGAs on a card, with the first FPGA folding odd numbered RNAs, and the latter FPGA folding even numbered RNAs. The scores are merge in order on the first FPGA before returning the results back to software.

For our two card setup, we split the database into two halves and ran independent copies of the interfacing code. For the multi-workstation experiment we again split the database and ran independent copies of the software over *ssh* using a shared file system. The time to preprocess the database is not included in the runtimes.

As shown in Table 8.2, using both FPGAs on a single card results in a close to 100-fold speedup over the dual core software baseline. A single workstation accelerator with two cards shows a 198× speedup over our dual core workstation. Indeed, our single workstation, which is a single rack server, is equivalent in performance to 198 workstations and represents a tremendous savings in floor space. On three work-station accelerators we demonstrate a 590× speedup over a single general-purpose workstation.

8.4 Related Work

Recently, significant effort has been spent in accelerating the Zuker algorithm on multi-core processors, graphics accelerators, and FPGAs.

8.4.1 Multi-core Implementations

GTfold [107] is an OpenMP shared-memory implementation of Zuker on an IBM P5-570 server with 16 dual core 1.9 GHz CPUs. GTfold is able to fold an RNA of length

9781 bases 19× faster than a sequential implementation. The time to fold 11 RNAs, each over 7000 bases, was reduced from about 3 months to under 9 minutes.

The weakness of this approach is that the sequences must be large enough so that the communication time between cores is minimal compared to the computation time. Modern RNA-folding algorithms have poor accuracy when the sequence is larger than about 200 bases [38] and are considered unreliable for sequences thousands of bases in length. For example, the referenced study shows a median accuracy of 80% for sequences of average length 120, but accuracy falls to 41% for sequences of length 1000-3000. For short sequences of a few hundred bases, which encompass the majority of biologists' workload, there is too little work to distribute among the 32 cores.
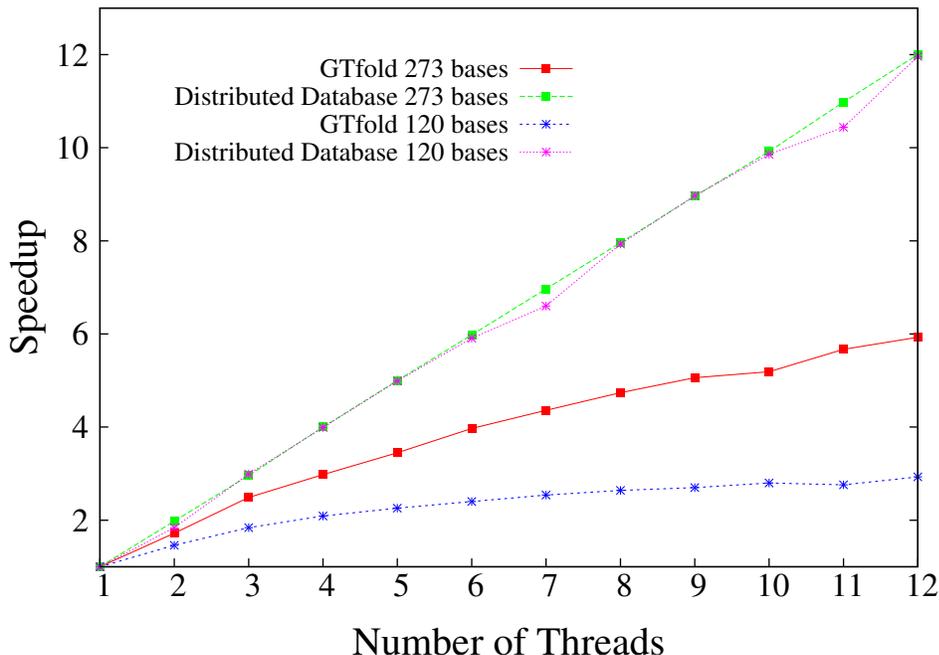
Figure 8.5: Speedup of GTfold and a split database approach on a twelve-core workstation.

To illustrate this point, we ran GTfold on two six-core 2.4 GHz AMD Opteron CPUs to fold databases of 120- and 273- base RNAs, each with 100,000 sequences. Figure 8.5 shows poor scaling for the OpenMP approach; on all 12 cores, speedup is only 6× for 273- base RNAs, and falls to 3× for 120- base RNAs. In contrast to the GTfold approach, one may accelerate the folding of a database of short sequences by distributing the workload among independent threads. Our experiments show near

perfect scaling as the database is split into equal chunks and distributed among up to 12 cores (we did not consider the time required to split the database).

8.4.2 RNA Folding on GPUs

Rizk and Lavenier [126] used an NVIDIA GTX280 GPU to search for microRNAs of length 120 bases, accelerating Zuker 17-fold versus one core of a Xeon 2.66 GHz workstation. This GPU has 30 multiprocessors, each a SIMD unit of eight 32-bit processors. The authors use multiple levels of parallelism: across several sequences, across cells on a diagonal, and across independent threads for the computation of each variable in the algorithm. Memory access is the bottleneck in their implementation—there is too little low-latency memory for fast access to the data variables and energy parameters.

Rizk and Lavenier used their GPU implementation to fold 40,000 randomly generated RNAs of length 120 bases on an NVIDIA Tesla C870 and GTX280. The execution times of the two GPUs were 32.8 and 18.9 seconds respectively. Accordingly, we built a new bitfile to fold 121-base RNAs. We were able to fit two arrays on our FPGA and clock the design at 110 MHz. To amortize I/O, we folded 80,000 randomly generated sequences in hardware and halved the runtime to derive the execution time for 40,000 sequences—2.72 seconds. Our design can fold 120-base RNAs 12.1× and 6.9× faster than the Tesla C870 and GTX280 respectively. Note that both GPUs we have compared against are from a newer hardware generation compared to the Virtex 4 family. Our hardware is also 119× faster than Rizk and Lavenier's baseline, a single core of a Xeon 2.66 GHz with 6 MB cache.

8.4.3 FPGA Implementations

The special-purpose FPGA accelerator sketched in Figure 8.6 was suggested by Dou et al. [39]. It uses a linear array of processing elements (PEs) connected to SDRAM memories through an on-chip cache. Every PE requires ten block RAM memories to store a copy of the energy parameters, despite extensive optimizations done by the authors. Due to this heavy memory usage, V and W elements that are computed in
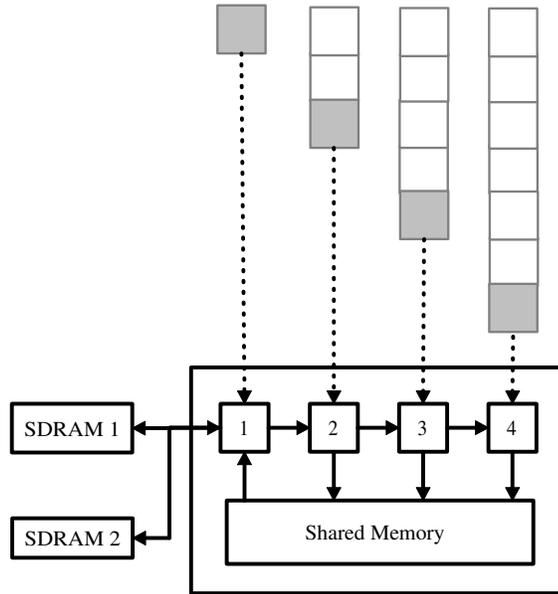
Figure 8.6: An FPGA array by Dou et al. [39] to accelerate Zuker.

the array must be stored in two SDRAM memories with a cache to support efficient reuse. The first PE requires direct access to SDRAM, but all other processors in the array need only left-to-right communication. Each PE is assigned one column of the computation domain, and the entire array computes cells along the same anti-diagonal simultaneously. Computation proceeds from bottom to top of a column, after which there is a synchronization phase when the results are written back to memory; then, the next block of columns is processed.

The authors' implementation on a Xilinx XC4VLX200-11 FPGA is memory-limited to 16 PEs clocking at 135 MHz. They achieve a $9.8\times$ speedup for 145-base sequences compared to a Pentium 4 2.6 GHz CPU. The implementation is able to process large RNAs and achieves a speedup of $14.3\times$ folding an RNA of length 2981 bases. We expect these speedups to halve if a modern processor core were used as the baseline.

Dou et al. mention a number of challenges faced in accelerating Zuker, including the number of dependence terms in computing the internal loop energy and the large number of energy parameters required. In particular, storage requirements for empirically computed energy tables of all possible hairpin and internal loops of certain sizes present a challenge to any meaningful acceleration. Variable dependence distances make it difficult to find task assignments that are balanced among PEs, and

long-range dependencies make it difficult to efficiently schedule the movement of V and W variables to the required PEs. One solution to the latter problem is to store these variables in a central cache until such time as they are required. However, this strategy may affect scalability if there are a large number of PEs in the array. The large number of small-granularity access operations also make it difficult to optimize scheduling for efficient external memory access. Furthermore, the long latency for off-chip memory access degrades performance.

Dou et al. implemented their array on the same FPGA family used in our experiments, a Xilinx Virtex 4 LX200-11. Their array can fold an RNA of length N using p processors in $\frac{N^3}{6p} + \frac{125p+12}{8p}N^2 + \frac{p+183}{12}N - \frac{3}{8}N^2$ clock cycles. They were able to fit 16 PEs on the device, clocking the array at 135 MHz. The estimated runtime on the 100,000 RNAs of length 273 is $1,007.4$ seconds. Our array performs the same computation $17.6\times$ faster.

Our Zuker array is faster than that of Dou et al. because we are able to fit a larger number of PEs on the FPGA. In fact, every PE in their array is roughly equivalent to the most resource-intensive PE in our array (PE 1, which contains the energy tables). Using polyhedral analysis we were able to identify the k axis for placement of PEs, whereas Dou's implementation places PEs along the j axis, necessitating duplication of the expensive energy tables in every PE.

Another interesting approach to special-purpose acceleration is the use of application-specific instruction-set processors. Moscola [113] uses this technique for a related problem, stochastic context-free grammar (SCFG) alignment to RNA sequences, which requires the computation of a dynamic programming matrix of cells, similar to RNA folding. While this recurrence is distinct from Zuker RNA folding, we may compare our systolic array architecture to a hypothetical application-specific instruction-set processor for the Zuker recurrence that is based on Moscola's work.

The architecture described by Moscola uses an array of PEs, each executing instructions from local memory. The matrix of cells, or computation points, are represented by nodes and dependencies as directed edges in a task graph. A single goal node, the final result of the computation, is identified, and all other nodes are labeled by their distance in number of edges to the goal. An execution time is assigned to each node starting from the ones with the greatest distance, gradually including those whose

predecessors have been scheduled. Mapping these instructions to processors is done in a way that optimizes resource usage.

Explicitly enumerating the schedule for every cell in the DP matrix results in a very large instruction stream. A 21-PE array running at 250 MHz requires an instruction streaming rate of 95 GBps, or alternately, several megabytes of storage in each PE. Rather than explicitly scheduling each cell, we turn to the polyhedral model. Here a single affine function is an implicit schedule for every cell in the matrix. This is possible because dependencies at every cell are identical, and known during compile time. The polyhedral model allows us to synthesize an efficient synchronous architecture that does not require an instruction stream; data flowing through the processor array activates the computation of cells in the matrix.

Similar to RNA folding, the SCFG alignment algorithm has a large number of non-local dependencies. Movement of these data dependencies in Moscola's architecture is done using a multi-port shared memory structure. To support efficient writes, memory is divided into several banks, each associated with a distinct PE. Every instruction executed in a PE may read up to 6 memory locations from any bank in shared memory, so a switching fabric is used for efficient routing.

The disadvantage of using shared memory to implement non-local communication is its poor scalability. Moscola reports that the logic for the switching fabric dwarfs other components of the architecture, growing to 90% of the resources when using only 21 processors. This limits the implementation to 20 processors on a Xilinx Virtex-5 FPGA. As demonstrated by the study of the Dou and Moscola architectures, communication via a shared memory structure leads to serious limitations in exploiting parallelism. Our approach is to instead use an interconnection network that pipelines data dependencies through neighboring PEs, leading to an efficient, low-cost, and highly scalable design. Since data dependencies are known a priori, we can use data pipelining techniques within the polyhedral framework to localize long range dependencies.

8.5 Conclusions

In this work, we have demonstrated the use of polyhedral analysis to build an array for the Zuker RNA folding algorithm. Our use of the polyhedral framework allowed systematic application of transformations and exploration of the design space, which is not easily achieved with *ad hoc* methods. We plan to investigate techniques that remove the limitation of a linear schedule, which in our case results in degraded performance. If we used FIFOs instead of fixed-delay registers for each dependency, we could double the speedup of our array by "skipping" iteration points outside the triangular computation domain.

Our array may be modified to implement closely related algorithms from the Vienna RNA package, including RNAalifold to fold a set of aligned RNAs and RNALfold to compute locally stable RNA secondary structures in entire genomes. In particular, an interesting exercise is to extend our Zuker accelerator to compute the partition function of the RNA structure ensemble using McCaskills algorithm [109]. While the dependencies of this algorithm are similar to the Zuker recurrence, computing the partition function requires multiply instead of addition operations, and summation instead of minimization of terms, which will increase resource requirements. It is likely that a newer generation of FPGAs with increased resources will be required to accelerate McCaskill's algorithm. Furthermore, we will have to dynamically scale the partition function values to avoid overflow/underflow.

Chapter 9

Conclusions & Future Directions

In this dissertation, we have advanced the idea of domain-specific acceleration, targeting the class of dynamic programming algorithms, where we identified a suitable high-level abstraction to represent these kernels and used parallelization techniques to build high-performance computing architectures.

We are motivated by the recent availability of low-cost, high-throughput next-generation sequencers that are producing vast quantities of DNA sequence that threaten to overwhelm traditional compute systems. In Chapter 1, we quantified this problem by showing that the historical growth of DNA bases per dollar of sequencing machines far outpaces performance per dollar of general-purpose workstations. While much effort and resources are being expended on low-cost sequencing with the aim of affordable human genome sequencing, considerably less attention is being paid to how these riches may be analyzed. We believe that unless this problem is addressed, compute performance, not low-cost sequencing, will become the bottleneck in advancing genome science.

We have suggested the targeted acceleration of dynamic programming algorithms, which find wide use in computational biology. We used the recurrence equation abstraction to represent these kernels and compiler techniques within the so-called polyhedral model to build systolic array accelerators. We showed how to parallelize dynamic programming kernels to build latency-, throughput-, and resource-optimized arrays depending on the user's requirements. By capitalizing on recent advances in FPGA systems, we were able to demonstrate significant speedups compared to modern multicore processors.

A major contribution in this work was our decision to optimize for throughput rather than take the prevailing approach in literature, which is to build latency-optimal arrays. We introduced a novel method to build throughput-optimized systolic arrays from recurrence abstractions within the polyhedral framework. Our key observation was that throughput is independent of the schedule of operations and depends solely on the processor allocation. Using this insight, we designed techniques to efficiently pipeline independent input instances on an array as well as multiple iterations in a single processor. The relative unimportance of latency allows designers to manipulate the schedule of operations to manage I/O into the array.

We parallelized the Nussinov RNA folding algorithm, designing and demonstrating two latency-optimal arrays and several throughput-optimized arrays on a Xilinx Virtex 4 FPGA platform. We experimentally verified a two-fold speedup of our throughput-optimized array over the latency-optimal one despite using fewer resources. Furthermore, similar throughput-optimized arrays for the string parenthesization problem improve on a decades-old latency-space optimal accelerator. We have also demonstrated the use of an existing partitioning technique to build three families of resource-constrained arrays for the Nussinov algorithm and have suggested techniques for efficient FPGA implementation.

We advanced the idea that for inputs that vary according to some attribute, such as sequence length, there may not be one optimal array for all inputs but rather multiple arrays optimal for ranges of the input attribute. We suggested the use of FPGA reconfiguration to switch between multiple arrays in response to varying input. We introduced a novel algorithm to select an optimal ordering of arrays, selected from a family of accelerators, that produces significant speedup over using a single array. We showed interesting speedup despite a reconfiguration time of hundreds of milliseconds and limiting the algorithm to use only a small number of arrays.

Finally, we analyzed and accelerated the challenging Zuker RNA folding algorithm, demonstrating close to $200\times$ speedup on a four-FPGA workstation over a dual core system. The flexibility of polyhedral techniques and the capabilities of modern FPGA devices allowed us build accelerators that outperform competing hardware accelerators designed using ad-hoc approaches, as well as GPU solutions.

9.1 Observations & Comments

We summarize a few key lessons learned during the course of this dissertation that can be generally applied by researchers.

- **Modern FPGA platforms are highly capable, high-performance compute solutions.** They are well suited for data parallel integer applications like those addressed in this dissertation. FPGA accelerators can achieve speedups on codes with some control dependency by replicating compute logic, and does not degrade performance like current GPU accelerators. Although our FPGA accelerators were clocked 15-20× slower than a general-purpose processor, we were still able to achieve significant speedups. The key was to exploit parallelism at all levels, including at the instruction- and loop-level, and to aggressively pipeline computations. We expect this efficiency advantage to remain as FPGAs grow in size and more cores are added to general-purpose processors. The regularity and locality of systolic array architectures allow them to scale on larger FPGA devices.

  We have used the relatively older-generation Virtex 4 FPGA family in our study, which has been superseded by three newer generations. The large amounts of logic and distributed memory elements on these newer devices allow designers to realize massively parallel accelerators, even for resource-hungry algorithms.

  Modern FPGA platforms support high-bandwidth coupling with the host CPU and disk subsystem allowing high-throughput data processing. Currently available systems, such as the one designed by Exegy Inc., which we used in this dissertation, can incorporate four FPGAs in a single host and offer significant performance improvement over GPU or CPU compute solutions.

- **It is worthwhile investigating if FPGA reconfiguration can enhance performance of other algorithm accelerators.** We have successfully improved performance through FPGA reconfiguration (even without partial reconfiguration). Furthermore, substantial benefits can be had with reconfiguration times as high as 500 ms. Our experiments suggest that further reducing this programming time will likely result in diminishing returns; it is more important

for FPGA vendors to support runtime reconfiguration across all FPGA families, even if at comparatively slow speeds.

- **The systolic array paradigm is well worth considering for use in accelerating loop programs.** The simplicity, regularity, and locality of interconnections between processors are an excellent match for FPGA fabrics. The regularity and locality allow systolic array designs to route at relatively high clock speeds.

  Importantly, there is a well-understood systematic design procedure to convert high-level loop programs (or recurrence equations) to systolic arrays. Polyhedral analysis allows the designer to optimize for multiple performance objectives, while accounting for resource-constrained targets. Synchronization of control and data, which can be difficult for programmers to manage, is mechanically specified through the use of a scheduling function. While hardware programming is challenging, the regularity of systolic arrays and their formal specification by an array mapping, makes programming them, even in VHDL, a far more manageable task. For example, each of the Nussinov arrays were programmed and tested in hardware in about 1-2 weeks. The high-performance Zuker array was implemented and verified in hardware in five weeks.

- **A domain-specific approach to parallelization that simultaneously considers the programming model, compilation techniques, and the hardware architecture can yield significant dividends.** We believe a domain-specific approach can produce just as much, if not more, of an impact as a general approach that aims to parallelize all sequential codes. The keys to our success using the domain-specific strategy included:

  - The domain of computational biology, which required acceleration of a large set of problems that could be mapped well onto the fabric of our accelerator.

  - The simple to analyze and optimize systolic array hardware architecture.

  - A high-level programming model that naturally describes a large class of problems in the domain and a suitable parallelization method. Focusing on a class of similar programs allowed us to apply specialized parallelization techniques that exploit the specific characteristics of the problem domain.

177

The parallelization techniques and the hardware architecture must be able to compete against hand-optimized designs, however, improvements in programmer productivity is of paramount importance. The long-term goal should be to work toward a parallelizing compiler that seamlessly ties the three components together.

We see success in applying this domain-specific parallelization strategy to other important algorithm classes.

9.2 Future Work

There are a number of short-term and longer-term challenges that remain to be addressed.

9.2.1 Short-term goals

Code generation. The important problem of code generation for systolic arrays remains a challenging problem. Specifically, generating efficient hardware code for latency- and throughput-optimized arrays, as well as automatically synthesizing control pipelines for algorithms such as Nussinov RNA folding is challenging. In addition, we have not fully explored the problem of designing the control architecture for throughput-optimized arrays. Additional work must be done to enable the automatic generation of correct control pipelines in the face of input pipelining. Furthermore, the problem of optimizing I/O into the systolic array, both reducing the number of processors performing I/O and the amount of buffering required, by manipulating the array schedule is unsolved.

Design space exploration of partitioned arrays. Although we have demonstrated the application of partitioning to build several novel resource-constrained arrays, we did not address the problem of design space exploration for partitioning. This is a challenging problem because of the large design space and a costly evaluation process due to the absence of linear formulations for the resource cost and

performance of a design—the only accurate way to measure array performance and resource usage is to implement the design on an FPGA, which is a time-consuming process. For this dissertation, we wrote a script to search the design space, performing a binary search over the tile parameters and automatically implementing the designs on an FPGA to determine resource cost. Additional investigation is required to understand how to more efficiently search this space and provide guarantees on optimality.

Efficient acceleration of non-rectangular domains. The partitioning technique we have used in this work assumes the input recurrence is always rectangular. This is a simplifying assumption that greatly simplifies synthesis of accelerators. We have had to modify the triangular RNA folding domains to use this parallelization technique, in the process, achieving only 50% utilization of processors. It will be worthwhile investigating hardware architectures that can overcome this limitation using some of the ideas presented at the end of Chapter 8.

Fully exploiting FPGA reconfiguration. We have demonstrated the utility of FPGA reconfiguration to switch between arrays to improve performance, responding to changes in input size distribution. A future goal is to generalize this algorithm to respond to other input properties. One avenue under investigation by members of our research group is to switch arrays according to the distribution of bases $\{G, C\}$ in a DNA sequence. A further challenge is to attain some of the benefits of improved performance through reconfiguration for online inputs, i.e., where the properties of the input are not known *a priori.*

9.2.2 Long-term goals

Automatic parallelization of dynamic programming. Our work is a step toward the longer-term goal of building a complete, automated system for accelerating dynamic programming algorithms. The entire workflow for such a compiler is illustrated in Figure 9.1. We envision a compiler that accepts domain-friendly input descriptions (called biological models), such as the weighted finite-state automata
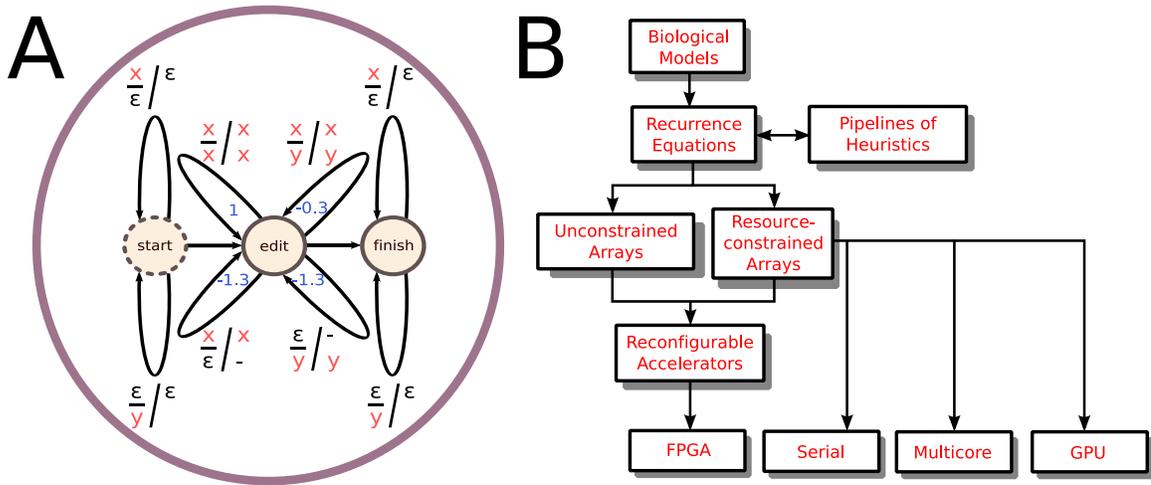
Figure 9.1: **(a)** A finite-state automata representation of the Smith-Waterman dynamic programming recurrence may be entered through a visual programming environment. **(b)** Future work describing a compiler for automated synthesis of dynamic programming accelerators.

from the visual programming environment of Searls and Murphy [135], and automatically generates optimized accelerator code. We have been exploring the use of an existing open-source compiler, MMAlpha, which may be extended for this purpose.

There are three main areas that have not been completely automated. First, programs written in domain-specific abstractions like those described in Chapter 2 must be automatically converted into recurrence equations. A basic translation procedure should be straightforward, however, additional effort will be required to exploit significant opportunities in optimizing time and space complexities of the input programs.

Second, non-uniform recurrences must be automatically transformed into uniform ones. In this work we have spent significant manual effort performing such analyses. Some recent results show promise in performing this transformation automatically [52].

Finally, existing code generators in compilers such as MMAlpha must be improved. In particular, they struggle with recurrence equations that use complex control predicates and fail to produce control pipelines to implement these codes. We were unsuccessful in generating VHDL code for our Nussinov accelerators using MMAlpha.

180

Generation of heuristics. An interesting avenue for investigation is the automated generation of heuristics for dynamic programming through the novel use of polyhedral analysis. In computational biology, heuristics are often used to manage the large execution time of dynamic programming kernels. However, as far as we are aware, they have always been designed in an ad-hoc manner. An excellent example is NCBI BLAST for the Smith-Waterman recurrence, or HMMERHEAD for the Viterbi recurrence. We may be able to use polyhedral analysis to identify heuristics that reduce runtime, for example, by simplifying reductions in recurrences or removing low-probability dependencies (determined through limited experimentation) that restrict parallelism.

Supporting heterogeneous processors. We believe that future high-performance computing platforms will be a heterogeneous blend of multiple compute resources. An interesting example is the tight coupling of reconfigurable logic with microprocessors. Recently, Xilinx announced that they would embed 800 MHz dual core ARM CPUs on an FPGA chip. Intel plans to ship the Stellarton processor in 2011, which lays an Atom processor in an Altera FPGA. Our work may be profitably extended to exploit such heterogeneous resources and expand the scope of algorithms that may be accelerated to more control-dependent codes.

Appendix A

A Survey of Dynamic Programming in Computational Biology

Alignment: The most common sequence analysis task is to compare two sequences for similarity. The alignment algorithm due to Needleman and Wunsch [116], and a later more efficient one by Gotoh [57], obtains the optimal global alignment between two DNA or protein sequences. Smith and Waterman [137] gave a recurrence to find the best local alignment— the best alignment between subsequences of two sequences. These recurrences have a relatively low time and space complexity of $O(n^2)$ but are still not practical for use with genome-size datasets.

Enhancement to these models is possible by modifying the basic recurrence; for example, finding multiple copies of a repeated motif in a sequence, or overlapping matches with overhanging ends. The fidelity of alignments produced are improved by changing the scoring functions used in the recurrence. Scalar values are sufficient for match and mismatch scores in DNA alignment but protein alignment requires a table of integer scores. Gap scores that are linear functions are the most efficient to implement, while affine functions lead to an increase in the space cost by a constant factor. Fully arbitrary gap functions increase the time complexity of the recurrence to $O(n^3)$ which is prohibitive in many cases.

In multiple sequence alignment more than two sequences are aligned to find areas of conservation in a family of related sequences. An optimal dynamic programming algorithm to align m sequences of average length n takes time exponential in the

input, $O(n^m)$. Due to the high computational cost, the algorithm is not widely used. Most multiple sequence alignment algorithms use a progressive strategy where pairwise sequence alignment is repeatedly applied. The pairwise dynamic programming algorithm is similar to the Smith-Waterman algorithm but operates on sequence profiles (see below) and requires traceback of the alignment. An example is ClustalW, which has been shown to spend more than 90% of its execution time in the pairwise alignment phase.

Probabilistic Modeling with Profile HMMs: Related sequences often exhibit shared domains that display varying levels of conservation. An alignment of multiple homologous sequences reveal collections of residues that are conserved and other regions with many mismatches and gaps. Searching a database using the conserved features of a family of sequences leads to higher quality matches than simple pairwise alignment, and is useful for example, to find distantly related sequences.

A Profile Hidden Markov Model (HMM) is a probabilistic model used to represent the salient features of a multiple alignment. The profile HMM is described by a set of states that model the probability of a match, insert or delete event at each position in the multiple alignment. HMM architectures have been devised to find multiple copies of a domain, fragments of domains, and local alignments. The most popular architecture is the Plan7 model used in the HMMER software [43] followed by SAM [73]. META-MEME [58] uses motifs (contiguous set of match states) separated by an insert state to build HMM architectures.

Given an HMM architecture, standard dynamic programming algorithms are available to locate matches to database sequences. While these recurrences share the same basic shape across different HMM architectures, they may have a varying number of variables, dependencies, initialization conditions and scoring functions. The Viterbi algorithm is used to find the most probable alignment of a sequence to an HMM. The Forward algorithm computes the probability that an HMM generates a sequence. Both recurrences have time complexity $O(nl^2)$ and space complexity $O(nl)$ where n is the length of the sequence and l, the length of the model. In practice, profile HMMs have a constant number of state transitions at each position of the model thus reducing time complexity to $O(nl)$. The Viterbi and Forward algorithms are important recurrences that have seen point accelerators.

183

Prediction of protein features with HMMs: Bystroff and Krogh survey the use of HMMs to detect various features in protein sequences [25] including signal peptides and their cleavage sites, protein secondary structure and membrane helices.

An early approach to predicting transmembrane helices in membrane proteins uses a non-probabilistic dynamic programming recurrence [81]. It finds the position and length of helices in a sequence but is of cubic time complexity. HMMs are better suited for this kind of prediction. It has been observed that there is an abundance of positively charged hydrophobic amino acids in selected regions of these helices; a fact easily incorporated into an HMM. TMHMM [88] uses an HMM architecture with seven labels (variables in the DP recurrence) that can detect helices of length between 15 and 35. The most probable topology of the transmembrane protein is computed using the Viterbi algorithm. A point to note, however, is that these HMM architectures contain loops and long-range state transitions and thus require the Viterbi and Forward algorithms that have time complexity $O(nl^2)$.

For applications that detect features in sequences, predicting the most probable labeling is more important that the most likely path. For example, when predicting transmembrane helices, certain groups of states represent the helix core, others loops or helix caps. The labeling of a sequence has a biological meaning not necessarily present in a state path. The 1-best [87] and the posterior-viterbi decoding [47] algorithms have been suggested for this purpose. They can be a more accurate predictor of the topology than Viterbi if each label consists of multiple states, as is true in this case.

HMM-HMM Alignment: Distantly related proteins with common structure and function are harder to detect by simple alignment techniques because they share very little primary sequence. Multiple sequences from a related family have more information—such as the position and frequency of conserved regions—than a simple sequence. Therefore it is beneficial to align families of multiple sequence alignments to yield a more sensitive search.

Profile-sequence methods align a query to a profile; profile-profile methods align two profiles. Both use algorithms similar to Smith-Waterman, but with non-trivial modifications to the similarity scoring scheme. HHsearch [138] performs a pairwise alignment of two HMMs, each of which summarizes a multiple sequence alignment. It

uses a simple $O(l^2)$ dynamic programming algorithm with five variables, and a log-odds metric to compare two columns of the HMMs. COACH [44] aligns a multiple sequence alignment to an HMM (that summarizes another multiple sequence alignment). It uses a recurrence similar to Viterbi but with several more variables and complex scoring functions.

Gene Prediction: Gene prediction programs predict the exon-intron structure of genes in genome sequences. Ab initio methods use just the sequence without any supporting evidence. GENSCAN [24] uses a probabilistic model called the Generalized Hidden Markov Model (GHMM) for prediction. It differs from the simple HMM in that each state in the model can emit more than one sequence residue. GHMMs are preferred for gene prediction because they can model the biologically observed length distributions of various gene components. Viterbi and Forward algorithms for GHMMs have a worst case time complexity of $O(n^3 l^2)$, nevertheless they are standard for modern gene prediction tools. Most of these tools limit the number of residues that can be emitted in each state, but this is often ad-hoc and difficult to parallelize. In speech recognition, the maximum length is limited to a constant D, making the decoding algorithms linear in the input sequence.

Evidence based predictors such as GeneWise and GenomeWise [19] use experimental evidence such as cDNA or homologous sequences to improve gene prediction. Conceptually, two first-order pair-HMMs are used: one is a gene prediction model, and the other a protein homology model. The gene prediction model converts a given DNA sequence to a protein sequence, and the protein homology model compares the predicted protein to a homologous sequence. GeneWise merges the two distinct models to create a single pair-HMM that directly compares an input genome to the homologous sequence, while considering all possible predictions of the protein. The authors report the use of several gene prediction models that compromise either speed or sensitivity. Each GeneWise pair-HMM can be solved using a distinct dynamic programming algorithm that varies according to the underlying model— nevertheless, all have a time complexity proportional to the product of the length of the two sequences. Dynamite, a code generation tool for dynamic programming, was used extensively to experiment with a large number of models.

Pachter at al. [118] tackle the problem of simultaneous alignment and gene prediction. The advantage of this method is that the alignment can be used to inform the prediction of genes, and the exon-intron structure helps generate a better alignment. The Generalized Pair HMM (GPHMM) is used with corresponding Forward and Viterbi algorithms. They have time complexity $O(mnl^2d^4)$, where m and n are the length of the two input sequences respectively, l the number of states, and d the maximum length of observations in each state. The space complexity is $O(mnl)$.

Secondary Structure: Many RNA sequences fold into a base-paired secondary structure that determines biological functions in the cell. It is therefore important to predict the structure of a sequence. Nussinov [117] gave the first such algorithm, which maximizes the number of base pairs in the optimal structure. It runs in time $O(n^3)$ and space $O(n^2)$ on an RNA of length n. A more sophisticated algorithm due to Zuker [166] attempts to find the structure with the lowest equilibrium free energy. It uses experimentally observed energy parameters and detects multi-branch loops, which increases its time complexity to $O(n^4)$. The time complexity can be reduced to be cubic by restricting the size of the multi-branch loop, or with a simplifying assumption on the energy function of multi-branch loops. The Zuker algorithm is widely used for RNA secondary structure prediction. Rivas and Eddy [125] improve these algorithms further by considering pseudoknots in their dynamic programming formulation. They are, however, infrequently used because of a high time complexity of $O(n^6)$ and space complexity of $O(n^4)$.

Rnall [158] finds local secondary structures by searching in a sliding window of a genome length sequence. The sequence window is folded inside out with a dynamic programming algorithm that forces the middle nucleotide to be in a hairpin loop. The time complexity of Rnall is $O(w^3n)$ for a genome of size n and a window of length w.

Simultaneous Sequence and Structural Alignment: Biologists are interested in locating both the sequence and structural alignment of related RNA sequences. Simultaneous alignment and folding on RNA sequences is used to iteratively guide each process, thus obtaining a better alignment. The Sankoff algorithm [130] integrates the global alignment and folding of two RNA sequences of length n to derive an algorithm that runs in time $O(n^6)$ and space $O(n^4)$. Sankoff also derives a dynamic programming algorithm to optimally align k RNA sequences that is too expensive

to run on modern workstations for realistic datasets. Therefore, similar to multiple sequence alignment, pairwise progressive alignment methods are used.

The Sankoff algorithm on two RNA sequences has been simplified further to make it more tractable. Gorodkin et al. [56] suggest an $O(n^4)$ running time simplification, FOLDALIGN, which ignores branches in the folded structures. Dynalign is another simplification that allow branches of only a fixed-size window, making the runtime cubic in the length of the sequence.

Stochastic Context-Free Grammars: Many related RNA sequences have a consensus secondary structure but share very little primary sequence. The information encoded in this structure must be taken into account when searching a database of RNAs if homologs are to be found. Simple formalisms like Hidden Markov Models, in contrast to the more powerful stochastic context-free grammars, cannot efficiently capture the base-paired secondary structure of RNAs. The consensus secondary structure of a multiple alignment of related RNAs along with sequence information can be coded in a probabilistic framework known as covariance models (CMs).

Given a CM representation and an observed sequence, the Inside dynamic programming algorithm computes the probability that the model generates the sequence. The algorithm has time complexity $O(l^2 n^3)$ where the CM has l states and the observed sequence is of length n. The CYK algorithm computes the maximum likelihood parse of the sequence by the CM, and has the same space and time complexity. Variations of these algorithms limit the search space to reduce the high time complexity, making searching of a large database feasible. The length of the longest aligned subsequence can be limited to a constant (rather than the length of the observed sequence), or DP restricted to a fixed-size band [115], making the algorithm linear in the sequence.

The Inside and CYK algorithms are non-trivial to parallelize because certain dependencies are dynamic, depending on the input model. Specifically, the dependence of an input state on its children is both dynamic, and requires a graph-like data structure which is harder to model than sequential datastreams. A possible solution is to assume a worst-case dependency structure and force an ordering on the CM states.

The Inside algorithm on pair-SCFGs performs simultaneous sequence alignment and secondary structure prediction of two sequences.

Conditional Random Fields: Hidden Markov Models have been successfully applied to a large number of problems in computational biology. Recent advances in machine learning have introduced Conditional Random Fields (CRFs) [92], which has been shown to perform better on many classification tasks. HMMs are generative models, in that they model the joint probability of a state and an observation sequence. CRFs are discriminative, meaning they model a conditional distribution of the state sequence on the observation sequence. The disadvantage of generative models is that incorporating sophisticated dependencies in the observation sequence can be intractable for inference. Since CRFs are conditional models, they do not need to model dependencies on observations, making inference easier. This allows the incorporation of arbitrary "features"— indicator variables that may be dependent on arbitrary positions in the observation sequence. CRFs are being used to advance the state-of-the-art on many of the classification tasks in computational biology, including the alignment of low similarity sequences, RNA folding and gene finding.

Like HMMs, CRFs use the Forward/Backward and Viterbi algorithms for inference. On a CRF model with l states and an observation sequence of length n, they both run in time $O(nl^2)$ and have a space complexity of $O(nl)$. The potentially challenging and novel aspect is the use of features. Efficient parallel arrays will have to be designed based on the specific dependencies of the features of each application. Features dependent on a small window of the observation sequence may be parallelizable with additional local memory and score tables. On the other hand, it might be altogether impossible to build efficient arrays for features with long dependencies (due to long communication links on the array) or features that are dependent on the state sequence (due to lack of parallelism). We are not aware of FPGA accelerators for CRFs and see it as an ideal example to demonstrate the use of our approach to ease the development of a family of accelerators.

Haplotyping Problem: The genomes of any two humans are identical except for single nucleotide variations called SNPs that occur approximately once every thousand bases. The string of nucleotides that occur at SNP locations of an individual is defined as its haplotype. The single individual haplotyping problem is to determine an individual's haplotype in the presence of incomplete and imperfect fragments of sequencing data. This is an area that has used dynamic programming to perform efficient combinatorial optimization. Rizzi et al. [127] give two algorithms to tackle

restricted versions of this problem that run in time $O(mn^2)$ and $O(m^2n)$ time where n is the number of SNP sites, and m, the number of fragments. An interesting aspect of these algorithms are their use of a graph data structure.

References

[1] GenBank database. `http://www.ncbi.nlm.nih.gov/genbank/`.

[2] Pip/piplib, a parametric integer linear programming solver, 2006. `http://www.piplib.org/`.

[3] Polylib - a library of polyhedral functions, 2007. `http://icps.u-strasbg.fr/polylib/`.

[4] Sequence read archive. `http://www.ncbi.nlm.nih.gov/sra`.

[5] SPEC CPU2006. `http://www.spec.org/cpu2006/`.

[6] UniProtKB TrEMBL and Swiss-Prot databases. `http://www.ebi.ac.uk/uniprot/`.

[7] Stephen F. Altschul, Thomas L. Madden, Alejandro A. Schffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman. Gapped BLAST and PSI-BLAST: A new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402, Sep 1997.

[8] Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. In *Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 39–50, 1991.

[9] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing, Feb 2009.

[10] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[11] Mauricio Ayala-Rincón, Ricardo P. Jacobi, Luis G. A. Carvalho, Carlos H. Llanos, and Reiner W. Hartenstein. Modeling and prototyping dynamically reconfigurable systems for efficient computation of dynamic programming methods by rewriting-logic. In *Proceedings of the 17th symposium on Integrated circuits and system design*, pages 248–253, 2004.

[12] Vineet Bafna, Haixu Tang, and Shaojie Zhang. Consensus folding of unaligned RNA sequences revisited. *Journal of Computational Biology*, 13(2):283–295, 2006.

[13] Stephan Balev, Patrice Quinton, Sanjay Rajopadhye, and Tanguy Risset. Linear programming models for scheduling systems of affine recurrence equations—a comparative study. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 250–258, 1998.

[14] R. T. Batey. Structures of regulatory elements in mRNAs. *Current Opinion in Structural Biology*, 16:299–306, 2006.

[15] Richard Ernest Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[16] Khaled Benkrid, Ying Liu, and Abdsamad Benkrid. Design and implementation of a highly parameterised FPGA-based skeleton for pairwise biological sequence alignment. In *Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 275–278, Washington, DC, USA, 2007. IEEE Computer Society.

[17] E. Berezikov, Cuppen E, and Plasterk RH. Approaches to microRNA discovery. *Nature Genetics*, 38 Suppl 1, June 2006.

[18] Ewan Birney. *Sequence alignment in bioinformatics*. PhD thesis, Sanger Centre, Wellcome Trust Genome Campus, 2000.

[19] Ewan Birney, Michele Clamp, and Richard Durbin. Genewise and genomewise. *Genome Research*, 14(5):988–995, May 2004.

[20] Ewan Birney and Richard Durbin. Dynamite: A flexible code generating language for dynamic programming methods used in sequence comparison. In *Proceedings of the 5th International Conference on Intelligent Systems for Molecular Biology*, pages 56–64. AAAI Press, 1997.

[21] Kiran Bondalapati and Viktor K. Prasanna. Dynamic precision management for loop computations on reconfigurable architectures. In *Field-Programmable Custom Computing Machines*, pages 249–258, 1999.

[22] Uday Bondhugula. *Effective Automatic Parallelization and Locality Optimization using the Polyhedral Model*. PhD thesis, The Ohio State University, 2008.

[23] Benjamin C. Brodie, Roger D. Chamberlain, Berkley Shands, and Jason White. Dynamic reconfigurable computing. In *Military and Aerospace Programmable Logic Devices*, 2003.

[24] Chris Burge and Samuel Karlin. Prediction of complete gene structures in human genomic dna. *Journal of Molecular Biology*, 268:78–94, 1997.

[25] Christopher Bystroff and Anders Krogh. Hidden markov models for prediction of protein features. 413, September 2007.

[26] R. W. Carthew and E. J. Sontheimer. Origins and mechanisms of miRNAs and siRNAs. *Cell*, 136(4):642–655, February 2009.

[27] Roger D. Chamberlain, Ron K. Cytron, Mark A. Franklin, and Ronald S. Indeck. The Mercury System: Exploiting truly fast hardware for data search. In *Proc. Int'l Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, pages 65–72, September 2003.

[28] 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.

[29] Shuai Che, Jie Li, J.W. Sheaffer, K. Skadron, and J. Lach. Accelerating compute-intensive applications with GPUs and FPGAs. pages 101 –107, jun. 2008.

[30] Ben Cope, Peter Y.K. Cheung, Wayne Luk, and Lee Howes. Performance comparison of graphics processors to reconfigurable logic: A case study. *IEEE Transactions on Computers*, 59:433–448, 2010.

[31] Tom Van Court. *LAMP - Tools for creating application-specific FPGA coprocessors*. PhD thesis, Boston University, 2006.

[32] Tom Van Court and Martin C. Herbordt. Families of FPGA-based accelerators for approximate string matching. *Microprocessors and Microsystems*, 31(2):135–145, 2007.

[33] Fimmel D. and Merker R. Determination of processor allocation in the design of processor arrays. *Microprocessors and Microsystems*, 22:149–155(7), 28 August 1998.

[34] Alain Darte, Robert Schreiber, B. Ramakrishna Rau, and Frédéric Vivien. Constructing and exploiting linear schedules with prescribed parallelism. *ACM Trans. Des. Autom. Electron. Syst.*, 7(1):159–172, 2002.

[35] 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.

[36] S. Derrien, S. Rajopadhye, and S. Sur-Kolay. Optimal partitioning for FPGA based regular array implementations. *International Conference on Parallel Computing in Electrical Engineering*, pages 155–159, 2000.

[37] S. Derrien, S. Rajopadhye, and S. Sur-Kolay. Combining instruction and loop level parallelism for FPGAs. In *Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 273–282, 2001.

[38] Kishore Doshi, Jamie Cannone, Christian Cobaugh, and Robin Gutell. Evaluation of the suitability of free-energy minimization using nearest-neighbor energy parameters for RNA secondary structure prediction. *BMC Bioinformatics*, 5(1):105, 2004.

[39] Yong Dou, Fei Xia, Xingming Zhou, and Xuejun Yang. Fine-grained parallel application specific computing for RNA secondary structure prediction on FPGA. In *Intl. Conf. on Computer Design*, pages 240–247, October 2008.

[40] E. A. Dougherty and J. A. Doudna. Ribozyme structures and mechanisms. *Annual Review of Biophysics and Biomolecular Structure*, 30:457–75, 2001.

[41] Sean Eddy. HMMER: Profile HMMs for protein sequence analysis. `http://hmmer.janelia.org/release-archive.html`.

[42] Sean R. Eddy. How do rna folding algorithms work? *Nature Biotechnology*, 22(11):1457–1458, November 2004.

[43] SR Eddy. Profile hidden Markov models. *Bioinformatics*, 14(9):755–763, 1998.

[44] Robert C. Edgar and Kimmen Sjolander. COACH: profile-profile alignment of protein families using hidden Markov models. *Bioinformatics*, 20(8):1309–1318, 2004.

[45] Esam El-Araby, Saumil G. Merchant, and Tarek El-Ghazawi. A framework for evaluating high-level design methodologies for high-performance reconfigurable computers. *IEEE Transactions on Parallel and Distributed Systems*, 99(PrePrints), 2010.

[46] 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.

[47] Piero Fariselli, Pier Martelli, and Rita Casadio. A new decoding algorithm for hidden markov models improves the prediction of the topology of all-beta membrane proteins. *BMC Bioinformatics*, 6(Suppl 4):S12, 2005.

[48] Kayvon Fatahalian, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

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

[50] Pierrick Gachet, Brigitte Joinnault, and Patrice Quinton. Synthesizing systolic arrays using DIASTOL. In *International Workshop on Systolic Arrays*, pages 25–36, 1986.

[51] Mark K. Gardner, Wu chun Feng, Jeremy Archuleta, Heshan Lin, and Xiaosong Mal. Parallel genomic sequence-searching on an ad-hoc grid: experiences, lessons learned, and implications. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 104, 2006.

[52] G Gautam and S. Rajopadhye. Simplifying reductions. In *Symposium on Principles of programming languages*, POPL '06, pages 30–41, 2006.

[53] Robert Giegerich, Carsten Meyer, and Peter Steffen. A discipline of dynamic programming over sequence data. *Science of Computer Programming*, 51(3):215–263, 2004.

[54] Maya Gokhale, Jonathan Cohen, Andy Yoo, W. Marcus Miller, Arpith Jacob, Craig Ulmer, and Roger Pearce. Hardware technologies for high-performance data-intensive computing. *Computer*, 41(4):60–68, 2008.

[55] Maya B. Gokhale, Janice M. Stone, Jeff Arnold, and Mirek Kalinowski. Stream-oriented FPGA computing in the streams-C high level language. In *Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 49–56, 2000.

[56] J Gorodkin, LJ Heyer, and GD Stormo. Finding the most significant common sequence and structure motifs in a set of RNA sequences. *Nucleic Acids Research*, 25(18):3724–3732, 1997.

[57] Osamu Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162(3):705–708, 1982.

[58] William N. Grundy, Timothy L. Bailey, Charles P. Elkan, and Michael E. Baker. meta-MEME: Motif-based hidden Markov models of protein families. *Computer Applications in the Biosciences (CABIOS)*, 13(4):397–406, 1997.

[59] P. Guerdoux-Jamet and D. Lavenier. SAMBA: hardware accelerator for biological sequence comparison. *Computer applications in the biosciences : CABIOS*, 13(6):609–615, 1997.

[60] L. T. Guibas, H. T. Kung, and C. D. Thompson. Direct VLSI implementation of combinatorial algorithms. In *Proceedings of the Caltech. Conference on VLSI*, pages 509–525, 1979.

[61] 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.

[62] Gautam Gupta, Sanjay Rajopadhye, and Patrice Quinton. Scheduling reductions on realistic machines. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 117–126, 2002.

[63] 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.

[64] G. J. Hannon. RNA interference. *Nature*, 418:244–51, 2002.

[65] Brandon Harris, Arpith Jacob, Joseph Lancaster, Jeremy Buhler, and Roger Chamberlain. A banded smith-waterman FPGA accelerator for *Mercury* BLASTP. In *IEEE Conference on Field Programmable Logic and Applications*, 2007.

[66] Martin C. Herbordt, Josh Model, Bharat Sukhwani, Yongfeng Gu, and Tom VanCourt. Single pass streaming BLAST on FPGAs. *Parallel Computing*, 33(10-11):741–756, 2007.

[67] D.T. Hoang. Searching genetic databases on Splash 2. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 185 –191, 1993.

[68] M. Höchsmann, T. Töller, R. Giegerich, and S. Kurtz. Local similarity in RNA secondary structures. In *Proc. of the IEEE Bioinformatics Conference*, pages 159–68, 2003.

[69] Ivo L. Hofacker, Walter Fontana, Peter F. Stadler, L. Sebastian Bonhoeffer, Manfred Tacker, and Peter Schuster. Fast folding and comparison of RNA secondary structures. *Chemical Monthly*, 125:167–188, February 1994.

[70] Christian Hoffmann, Nana Minkah, Jeremy Leipzig, Gary Wang, Max Q. Arens, Pablo Tebas, and Frederic D. Bushman. DNA bar coding and pyrosequencing to identify rare HIV drug resistance mutations. *Nucleic Acids Research*, 35(13):e91, 2007.

[71] Daniel Reiter Horn, Mike Houston, and Pat Hanrahan. ClawHMMER: A streaming HMMer-search implementation. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 11, 2005.

[72] Richard Hughey and Andrea Di Blas. The UCSC kestrel application-unspecific processor. In *Proceedings of the IEEE 17th International Conference on Application-specific Systems, Architectures and Processors*, pages 163–168, 2006.

[73] Richard Hughey and Anders Krogh. Hidden markov models for sequence analysis: Extension and analysis of the basic method. *Computer Applications in the Biosciences (CABIOS)*, 12:95–107, 1996.

[74] Arpith Jacob, Jeremy Buhler, and Roger Chamberlain. Accelerating Nussinov RNA secondary structure prediction with systolic arrays on FPGAs. In *Application-specific Systems, Architectures and Processors*, pages 191–196, 2008.

[75] Arpith Jacob, Jeremy Buhler, and Roger Chamberlain. Optimal runtime reconfiguration strategies for systolic arrays. In *International Conference on Field-Programmable Logic and Applications*, pages 162–167, 2009.

[76] Arpith Jacob, Jeremy Buhler, and Roger Chamberlain. Design of throughput-optimized arrays from recurrence abstractions. In *Application-specific Systems, Architectures and Processors*, pages 133–140, 2010.

[77] Arpith Jacob, Jeremy Buhler, and Roger Chamberlain. Rapid RNA folding: Analysis and acceleration of the zuker recurrence. In *Proceedings of the 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 87–94, 2010.

[78] Arpith Jacob and Maya Gokhale. Language classification using n-grams accelerated by FPGA-based Bloom filters. In *Proceedings of the 1st international workshop on High-performance reconfigurable computing technology and applications*, pages 31–37, 2007.

[79] Arpith Jacob, Joseph Lancaster, Jeremy Buhler, and Roger Chamberlain. Preliminary results in accelerating profile HMM search on FPGAs. In *IEEE International Workshop on High Performance Computational Biology*, 2007.

[80] Arpith Jacob, Joseph Lancaster, Brandon Harris, Jeremy Buhler, and Roger Chamberlain. *Mercury* BLASTP: Accelerating protein sequence alignment. *ACM Transactions on Reconfigurable Technology and Systems*, 2008.

[81] D. T. Jones, W. R. Taylor, and J. M. Thornton. A model recognition approach to the prediction of all-helical membrane protein structure and topology. *Biochemistry*, 33(10):3038–3049, 1994.

[82] Richard M. Karp and Michael Held. Finite-state processes and dynamic programming. *SIAM Journal on Applied Mathematics*, 15(3):pp. 693–718, 1967.

[83] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, 1967.

[84] Srinidhi Kestur, John D. Davis, and Oliver Williams. BLAS Comparison on FPGA, CPU and GPU. In *Proceedings of the 2010 IEEE Annual Symposium on VLSI*, pages 288–293, 2010.

[85] Marianthi Kiriakidou, Peter T. Nelson, Andrei Kouranov, Petko Fitziev, Costas Bouyioukos, Zissimos Mourelatos, and Artemis Hatzigeorgiou. A combined computational-experimental approach predicts human microRNA targets. *Genes & Development*, 18(10):1165–1178, 2004.

[86] P. Krishnamurthy, J. Buhler, R. Chamberlain, M. Franklin, K. Gyang, A. Jacob, and J. Lancaster. Biosequence similarity search on the *Mercury* system. *Journal of VLSI Signal Processing*, 49:101–121, 2007.

[87] Anders Krogh. Two methods for improving performance of a HMM and their application for gene finding. In *Proceedings of the 5th International Conference on Intelligent Systems for Molecular Biology*, pages 179–186, 1997.

[88] Anders Krogh, Bjorn Larsson, Gunnar Von Heijne, and Erik L. L. Sonnhammer. Predicting transmembrane protein topology with a hidden markov model: application to complete genomes. *Journal of Molecular Biology*, 305:567–580, 2001.

[89] Dhananjay Kulkarni, Walid A. Najjar, Robert Rinker, and Fadi J. Kurdahi. Compile-time area estimation for LUT-based FPGAs. *ACM Trans. Des. Autom. Electron. Syst.*, 11(1):104–122, 2006.

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

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

[92] John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the Eighteenth International Conference on Machine Learning*, pages 282–289, 2001.

[93] E. C. Lai, P. Tomancak, R. W. Williams, and G. M. Rubin. Computational identification of Drosophila microRNA genes. *Genome Biology*, 4:R42, 2003.

[94] Monica Lam. Software pipelining: an effective scheduling technique for VLIW machines. *ACM SIGPLAN Notices*, 23(7):318–328, 1988.

[95] D. Lavenier, G. Georges, and X. Liu. A reconfigurable index FLASH memory tailored to seed-based genomic sequence comparison algorithms. *Journal of VLSI Signal Processing Systems*, 48(3):255–269, 2007.

[96] 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.

[97] Charles E. Leiserson and James B. Saxe. Optimizing synchronous systems. In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*, pages 23–36, 1981.

[98] Heshan Lin, Xiaosong Ma, Wuchun Feng, and Nagiza F. Samatova. Coordinating computation and I/O in massively parallel sequence search. *IEEE Transactions on Parallel and Distributed Systems*, 99, 2010.

[99] B Lisper. Linear programming methods for minimizing execution time of indexed computations. In *International Workshop on Compilers for Parallel Computers*, pages 131–142, 1990.

[100] Daniel Lopresti. Three computationally demanding problems in search of ASAP solutions. In *Proc. of the IEEE 17th Int'l Conf. on Application-specific Systems, Architectures and Processors*, pages 214–222, 2006.

[101] Jizhu Lu, Michael Perrone, Kursad Albayraktaroglu, and Manoj Franklin. HMMer-cell: High performance protein profile searching on the Cell/B.E. processor. *IEEE International Symposium on Performance Analysis of Systems and software*, pages 223–232, April 2008.

[102] Gerton Lunter. HMMoC—a compiler for hidden markov models. *Bioinformatics*, 23(18):2485–2487, 2007.

[103] Gerton Lunter, Andrea Rocco, Naila Mimouni, Andreas Heger, Alexandre Caldeira, and Jotun Hein. Uncertainty in homology inferences: assessing and improving genomic sequence alignment. *Genome research*, 18(2):298–309, February 2008.

[104] Rune B. Lyngsø, Michael Zuker, and Christian N. S. Pedersen. Fast evaluation of internal loops in RNA secondary structure prediction. *Bioinformatics*, 15(6):440–445, June 1999.

[105] S. Maas, R. Kaushal, D. Lopresti, D. Drake, S. Hookway, W. Scheirer, M. Strohmaier, and C. Wojciechowski. A bioinformatics approach to identify recoding events of a-to-i RNA editing. In *Proceedings of the Computational Systems Bioinformatics Conference*, 2006.

[106] Elaine R. Mardis. The impact of next-generation sequencing technology on genetics. *Trends in Genetics*, 24(3):133 – 141, 2008.

[107] Amrita Mathuriya, David A. Bader, Christine E. Heitsch, and Stephen C. Harvey. GTfold: a scalable multicore code for RNA secondary structure prediction. In *Proc. ACM Symposium on Applied Computing*, pages 981–988, 2009.

[108] 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.

[109] J. S. McCaskill. The equilibrium partition function and base pair binding probabilities for RNA secondary structure. *Biopolymers*, 29:1105–1119, 1990.

[110] Scott McGinnis and Thomas L Madden. BLAST: at the core of a powerful and diverse set of sequence analysis tools. *Nucleic Acids Research*, 32(Web Server issue):W20–5, 2004.

[111] Oskar Mencer. ASC: A stream compiler for computing with FPGAs. *IEEE Transactions on Computer-Aided Design*, 2006.

[112] Makedonka Mitreva. Shotgun metagenomic sequencing and analysis at the washington university genome center. In *Human Microbiome Research Conference*, 2010.

[113] James M. Moscola. *Techniques for hardware-accelerated parsing for network and bioinformatic applications*. PhD thesis, Washington University in St. Louis, St. Louis, MO. USA. 63130, May 2008.

[114] T. Mourier et al. Genome-wide discovery and verification of novel structured RNAs in Plasmodium falciparum. *Genome Research*, 18:281–92, 2008.

[115] Eric P Nawrocki and Sean R Eddy. Query-dependent banding (QDB) for faster RNA similarity searches. *PLoS Computational Biology*, 3(3):e56, March 2007.

[116] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two sequences. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[117] Ruth Nussinov, George Pieczenik, Jerrold R. Griggs, and Daniel J. Kleitman. Algorithms for loop matchings. *SIAM Journal on Applied Mathematics*, 35(1):68–82, July, 1978.

[118] Lior Pachter, Marina Alexandersson, and Simon Cawley. Applications of generalized pair hidden markov models to alignment and gene finding problems. In *Proceedings of the fifth annual international conference on Computational biology*, pages 241–248, 2001.

[119] G. Pavesi, G. Mauri, and G. Pesole. An algorithm for finding conserved secondary structure motifs in unaligned RNA sequences. *Journal of Computer Science and Technology*, 19:2–12, 2004.

[120] Elon Portugaly and Matan Ninio. HMMERHEAD - accelerating HMM searches on large databases. In *Currents in Computational Molecular Biology - Poster Abstracts from RECOMB*, pages 250–251, 2004.

[121] Kiran Puttegowda, William Worek, Nicholas Pappas, Anusha Dandapani, Peter Athanas, and Allan Dickerman. A run-time reconfigurable system for gene-sequence searching. In *International Conference on VLSI Design*, pages 561–566, 2003.

[122] Patrice Quinton. The systematic design of systolic arrays. In *Automata Networks in Computer Science: Theory and Applications*, pages 229–260. Princeton University Press, 1987.

[123] Sanjay V. Rajopadhye. Synthesizing systolic arrays with control signals from recurrence equations. *Distributed Computing*, 3(2):88–105, 1989.

[124] C. S. Riesenfeld, P. D. Schloss, and J. Handelsman. Metagenomics: genomic analysis of microbial communities. *Annual Review of Genetics*, 38:525–552, 2004.

[125] E. Rivas and S. R. Eddy. A dynamic programming algorithm for RNA structure prediction including pseudoknots. *Journal of Molecular Biology*, 285(5):2053–2068, February 1999.

[126] Guillaume Rizk and Dominique Lavenier. GPU accelerated RNA folding algorithm. In *Proceedings of the 9th International Conference on Computational Science*, pages 1004–1013, 2009.

[127] Romeo Rizzi, Vineet Bafna, Sorin Istrail, and Giuseppe Lancia. Practical algorithms and fixed-parameter tractability for the single individual SNP haplotyping problem. In *WABI*, pages 29–43, 2002.

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

[129] Yasubumi Sakakibara, Michael Brown, Richard Hughey, I. Saira Mian, Kimmen Sjlander, Rebecca C. Underwood, and David Haussler. Stochastic context-free grammers for tRNA modeling. *Nucleic Acids Research*, 22(23):5112–5120, 1994.

[130] David Sankoff. Simultaneous solution of the RNA folding, alignment and protosequence problems. *SIAM Journal on Applied Mathematics*, 45(5):810–825, 1985.

[131] David Sankoff. The early introduction of dynamic programming into computational biology . *Bioinformatics*, 16(1):41–47, 2000.

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

[133] 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. *J. VLSI Signal Process. Syst.*, 31(2):127–142, 2002.

[134] Stephan C Schuster. Next-generation sequencing transforms today's biology. *Nature Methods*, 5(1):16–8, 2008.

[135] D. B. Searls and K. P. Murphy. Automata theoretic models of mutation and alignment. In *Proceedings of the Third International Symposium on Intelligent Systems for Molecular Biology*, pages 341–349, 1995.

[136] Guy St. C. Slater and Ewan Birney. Automated generation of heuristics for biological sequence comparison. *BMC Bioinformatics*, 6:31, 2005.

[137] Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[138] Johannes Söding. Protein homology detection by HMM–HMM comparison. *Bioinformatics*, 21(7):951–960, 2005.

[139] Peter Steffen, Robert Giegerich, and Mathieu Giraud. GPU parallelization of algebraic dynamic programming. *Workshop on Parallel Computational Biology*, pages 290–299, 2009.

[140] Lincoln Stein. The case for cloud computing in genome informatics. *Genome Biology*, 11(5):207, 2010.

[141] Henry Styles and Wayne Luk. Branch optimisation techniques for hardware compilation. In *Field-Programmable Logic and Applications*, pages 324–333, 2003.

[142] Ramanjulu Sunkar, Xuefeng Zhou, Yun Zheng, Weixiong Zhang, and Jian-Kang Zhu. Identification of novel and candidate mirnas in rice by high throughput sequencing. *BMC Plant Biology*, 8(1):25, 2008.

[143] Sriram Swaminathan, Russell Tessier, Dennis Goeckel, and Wayne Burleson. A dynamically reconfigurable adaptive viterbi decoder. In *Symposium on Field-programmable gate arrays*, pages 227–236, 2002.

[144] William Thies, Frédéric Vivien, and Saman Amarasinghe. A step towards unifying schedule and storage optimization. *ACM Transactions on Programming Languages and Systems*, 29(6):34, 2007.

[145] David Barrie Thomas, Lee Howes, and Wayne Luk. A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation. In *Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 63–72, 2009.

[146] T. J. Todman, G. A. Constantinides, S. J. E. Wilton, O. Mencer, W. Luk, and P. Y. K. Cheung. Reconfigurable computing: architectures and design methods. *Computers and Digital Techniques*, 152(2):193–207, 2005.

[147] Justin L. Tripp, Maya B. Gokhale, and Kristopher D. Peterson. Trident: From high-level language to hardware circuitry. *Computer*, 40(3):28–37, 2007.

[148] Justin L. Tripp, Preston A. Jackson, and Brad Hutchings. Sea cucumber: A synthesizing compiler for FPGAs. In *Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, pages 875–885, 2002.

[149] Justin L. Tripp, Kristopher D. Peterson, Christine Ahrens, Jeffrey D. Poznanovic, and Maya Gokhale. Trident: An FPGA compiler framework for floating-point algorithms. In *Proceedings of the 15th International Conference on Field-Programmable Logic and Applications*, pages 317–322, 2005.

[150] Peter J. Turnbaugh, Ruth E. Ley, Micah Hamady, Claire M. Fraser-Liggett, Rob Knight, and Jeffrey I. Gordon. The human microbiome project. *Nature*, 449(7164):804–810, October 2007.

[151] Keith Underwood. FPGAs vs. CPUs: trends in peak floating-point performance. In *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 171–180, 2004.

[152] T. Van Court and M.C. Herbordt. LAMP: a tool suite for families of FPGA-based computational accelerators. *International Conference on Field Programmable Logic and Applications*, pages 612–617, Aug. 2005.

[153] 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, 2007.

[154] Hervé Le Verge, Christophe Mauras, and Patrice Quinton. The ALPHA language and its use for the design of systolic arrays. *Journal of VLSI Signal Processing Systems*, 3(3):173–182, 1991.

[155] Benjamin Wah and Guo jie Li. Systolic processing for dynamic programming problems. *Circuits, Systems, and Signal Processing*, 7:119–149, 1985.

[156] Dennis Wall, Parul Kudtarkar, Vincent Fusaro, Rimma Pivovarov, Prasad Patil, and Peter Tonellato. Cloud computing for comparative genomics. *BMC Bioinformatics*, 11(1):259, 2010.

[157] John Paul Walters, Bashar Qudah, and Vipin Chaudhary. Accelerating the HMMER sequence analysis suite using conventional processors. *International Conference on Advanced Information Networking and Applications*, pages 289–294, April 2006.

[158] Xiu-Feng Wan, Guohui Lin, and Dong Xu. Rnall: an efficient algorithm for predicting RNA local secondary structural landscape in genomes. *Journal of Bioinformatics and Computational Biology*, 4(5):1015–1032, 2006.

[159] Gary P. Wang, Angela Ciuffi, Jeremy Leipzig, Charles C. Berry, and Frederic D. Bushman. HIV integration site selection: Analysis by massively parallel pyrosequencing reveals association with epigenetic modifications. *Genome Research*, 17(8):1186–1194, 2007.

[160] Yiwan Wong. *Algorithms for systolic array synthesis*. PhD thesis, New Haven, CT, USA, 1989. AAI9015858.

[161] Yiwan Wong and Jean-Marc Delosme. Space-optimal linear processor allocation for systolic arrays synthesis. In *Proceedings of the 6th International Parallel Processing Symposium*, pages 275–282, Washington, DC, USA, 1992. IEEE Computer Society.

[162] Ben Wun, Jeremy Buhler, and Patrick Crowley. Exploiting coarse-grained parallelism to accelerate protein motif finding with a network processor. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 173–184, Washington, DC, USA, 2005. IEEE Computer Society.

[163] Jimmy Xu, Nikhil Subramanian, Adam Alessio, and Scott Hauck. Impulse C vs. VHDL for accelerating tomographic reconstruction. *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, pages 171–174, 2010.

[164] Peixin Zhong, Margaret Martonosi, Pranav Ashar, and Sharad Malik. Solving boolean satisfiability with dynamic hardware configurations. In *Field-Programmable Logic and Applications*, pages 326–335, 1998.

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

[166] M. Zuker. Computer prediction of RNA structure. *Methods in Enzymology*, 180:262–88, 1989.

[167] M. Zuker, D Mathews, and D Turner. Algorithms and thermodynamics for RNA secondary structure prediction: A practical guide. *RNA Biochemistry and Biotechnology, NATO ASI Series*, pages 11–43, 1999.

Dynamic programming parallelization, Jacob, Ph.D. 2010