THESIS

# RNA SECONDARY STRUCTURE PREDICTION USING ALPHAZ

Submitted by

Tanveer Pathan

Department of Electrical and Computer Engineering

In partial fulfillment of the requirements

for the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Fall 2010

COLORADO STATE UNIVERSITY

August 18, 2010

WE HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER OUR SUPERVISION BY TANVEER PATHAN ENTITLED RNA SECONDARY STRUCTURE PREDICTION USING ALPHAZ BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE.

Committee on Graduate Work

Committee Member : Sudeep Pasricha

Committee Member : A. P. Willem Bohm

Advisor : Sanjay Rajopadhye

Department Head : Anthony A. Maciejewski

ABSTRACT OF THESIS

RNA SECONDARY STRUCTURE PREDICTION USING ALPHAZ

Optimizing complex algorithms using conventional programming languages can be a difficult task. The performance aspect of such implementations relies on the programmer and the target architectures. Minor alterations to the algorithm can result in a considerable amount of reprogramming effort.

In our work, we experiment with equational programming using the AlphaZ tool. We illustrate our work using a fairly complex algorithm for RNA secondary structure prediction. The algorithm is extracted from the UNAFold software package, a huge C library, by identifying the time consuming parts. It is then represented as equations, which are transformed for optimizations, followed by code generation and finally validating the generated code by plugging it back into the original C program. The algorithm, in its basic form, has complexity $O(N^4)$. We show that the optimized $O(N^3)$ algorithm can be derived systematically using AlphaZ. We used the AlphaZ system to automatically generate a C program that implements this improved algorithm. Our work forms the basis for future optimizations and also acts as a case-study for polyhedral equational programming in the real world.

Tanveer Pathan
Department of Electrical and Computer Engineering
Colorado State University
Fort Collins, CO 80523
Fall 2010

# ACKNOWLEDGEMENTS

I sincerely thank my adviser Dr. Sanjay Rajopadhye for his continuous support and guidance throughout my stay at Colorado State University. I am very grateful to have had the opportunity to work with him in the field of High Performance Computing. He has also helped me experience the industrial aspects of High Performance Computing by providing me opportunities to intern at the Los Alamos National Laboratory and Apolent Corporation. He has been a mentor to me and has constantly guided me through the difficult times during this entire period.

I would also like to thank Dr. A. P. Willem Bohm and Dr. Sudeep Pasricha for being on my committee and for their valuable feedback with respect to my work. I have had the opportunity to learn from Dr. Bohm through his class and it was a delightful experience. I thank Dr. Pasricha for accepting my request for being on the committee.

I would like to thank Dr. Craig Rasmussen for being my mentor and providing both personal and professional support during my stay at the Los Alamos National Laboratory.

I would like to thank my lab mates for their support and feedback during the entire course of my research. Being a part of the MELANGE research group has helped me gain some wonderful experiences which have helped me and will continue to help me through the rest of my professional life. I would like to specially thank Dr. DaeGon Kim and Tomofumi Yuki for helping me by fixing the bugs in the AlphaZ tool as quickly as possible. I would also like to thank Dr. Gautam Gupta and Pradeep Srinivasa for their invaluable support and guidance, and for being such great friends.

I would like to thank Elisabeth Wadman, Karen Ungerer, Sharon Van Gorder and Kim Judith for helping me with all the administrative process.

I would like to express my gratitude towards all my friends for supporting me through the difficult times and always being there for me when I needed them. I specially thank my friends Guhan, Sridhar, Bala, Anand, Avantika and Karthik for their continuous support. I thank my parents for their blessings and guidance which have helped me get through difficult situations. I thank my brother for his support and wishes.

On an ending note, I extend my sincere regards to the almighty God, the most beneficent and the most merciful.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

# Chapter 1

# Introduction

Bioinformatics is the application of statistics and computer science to the field of molecular biology. The field of bioinformatics is advancing at a rapid rate. With these advances, various algorithms are being developed to simulate sub-fields like RNA folding, protein folding, sequence alignment and so on, because experimental methods to realize the results for these sub-fields are expensive and time consuming. Each of these sub-fields have various algorithms to achieve their end results, which are computationally intensive, often with correspondingly significant space requirements. Since processor technologies are no longer scaling with respect to Moore's law, we are now in the era of multi-core processors and parallel architectures. Programming for these architectures is a challenging and time consuming task for an algorithm developer. Hence, the task of implementing various bioinformatics algorithms, which are generally complex, turns out to be a strenuous one when considering these architectures. Fortunately, most of these algorithms involve mathematical operations over affine control loops (ACLs). This enables us to represent these algorithms in the polyhedral model, which is a mathematical framework for representing the iteration space of these loops along with the equational specification of the computations within the iteration space. The polyhedral representation enables us to apply transformations on the algorithm in a easy and precise manner. One such transformation is code generation, which produces a program to implement the algorithm on the specified target architecture. Since most bioinformatics algorithms are designed using mathematical equations, writing a specification for the polyhedral model can be

a relatively easy task for an algorithm developer, which can then be transformed into code for a specified architecture using code generators. Ideally, this would eliminate the need to write programs by hand for the new upcoming architectures including multi-core processor architectures.

## 1.1 The Polyhedral Model

The polyhedral model can be viewed as a mathematical framework where the algorithmic specification is represented in the form of equations. These equations are associated with domains which represent regions in the space where the equational specification holds. The domains for these equations are required to be strict, so that computations are defined for every point in the domain space.

The polyhedral model allows us to realize a general view of the algorithms, in the form of equational specification, without the necessity to know the target architecture on which the algorithms are to be implemented. These equational specifications can later be transformed to programs for one or many target architectures with the help of code generators developed for the polyhedral model.

At Colorado State University, our research group is developing a tool called AlphaZ to provide the functionality of the polyhedral equational model. It uses a basic transformation engine developed by COREquations [14]. The AlphaZ client and the COREquations engine together provide access to the various transformations available in the framework. This framework takes in an Alphabets program as input which is a list of equations specifying the computation along with their associated domains. The frameworks enables the user with the ability to analyze, transform and generate code for the input Alphabets program.

## 1.2 RNA Secondary Structure Prediction

In this thesis, we illustrate the advantages of using equational programming for realizing bioinformatics algorithms. Specifically, we concentrate on one of the bioinformatics algorithms, RNA Secondary Structure Prediction, also known as RNA

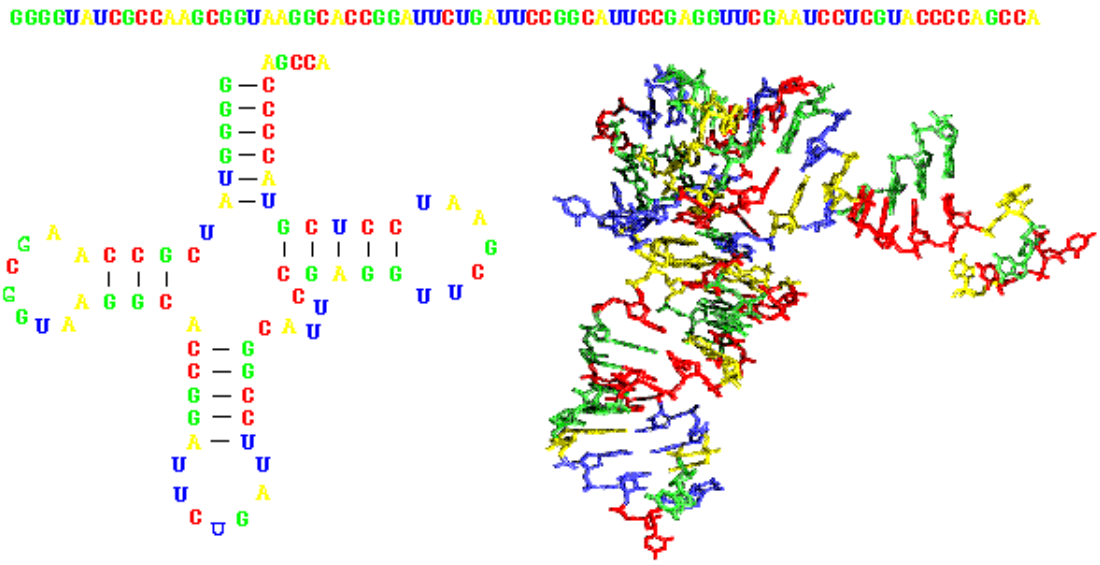

Figure 1.1: Primary, secondary and tertiary structures for tRNA. [2]

folding.

RNA molecules play a very important role in the process of protein synthesis which is essential to all living organisms on this planet. Also, viruses consist of RNA molecules and their biological role can be deduced from the functionality of the RNA molecules. This functionality of RNA molecules is determined by its secondary and tertiary structures shown in figure 1.1. However, as mentioned earlier, the experimental realization of these structures is a very expensive and time consuming process and therefore one has to resort to computational methods of determining secondary structures such as Comparative Sequence Analysis and Thermodynamics based structure prediction. In this thesis, we focus on one of the thermodynamics based structure prediction algorithm implemented in the UNAFold software package [22] developed by Markham and Zuker. The current implementation of UNAFold is composed of sequential C programs and Perl scripts. The UNAFold package addresses the folding of one single-stranded nucleotide sequence and the hybridization between two single-stranded nucleotide sequences.

In this thesis, we only focus on the single-stranded nucleotide sequence folding algorithm of the UNAFold package. We extract the equational specifications from

the UNAFold C program and present it in the form of an Alphabets program, which can then be analyzed and transformed to explore the scope of optimizations that can be performed within the polyhedral framework. Using the transformations provided in AlphaZ, we systematically derive the $O(N^3)$ algorithm, proposed by Lyngso et. al. [20], starting from the original $O(N^4)$ algorithm. We then use the demand-driven code generator in AlphaZ to generate the program for the derived $O(N^3)$ algorithm and plug it back into the original UNAFold C program. Our work justifies that AlphaZ can make a transition from handling toy problems like matrix-matrix multiply and LU decomposition to tackling real-life applications like RNA secondary structure prediction.

In the following chapter, we introduce AlphaZ, a system for programming using equations, and the algorithm for RNA secondary structure prediction used in UNAFold [22]. The next chapter will explain the details involved with respect to the implementation using the AlphaZ system, including a formal derivation for the $O(N^3)$ algorithm proposed by Lyngso et. al. [20]. This is followed by a chapter showing the results of our work. The final chapter concludes this document along with the prospective work that would follow.

# Chapter 2

# Background

In this chapter, we introduce the AlphaZ system which enables programming using equations. We explain the semantics of Alphabets, the input language for AlphaZ, followed by the various transformations available in the AlphaZ system. Later, we explain the algorithm for RNA secondary structure prediction. And, finally, we introduce the state of the art in RNA secondary structure prediction.

## 2.1 AlphaZ

AlphaZ is a system for polyhedral equation transformation and code generation. It makes extensive use of the COREquations transformation engine[14]. The COREquations transformation engine implements the polyhedral model framework based on the Parma Polyhedral Library [5]. It provides AlphaZ with a basic foundation by exporting functionality like Alphabets program parser, AST constructor, basic polyhedral operations, transformations, analyses and so on. AlphaZ communicates with the COREquations engine through a client-server setup. In addition to the functionality imported from COREquations engine, AlphaZ provides its own transformations and analyses. The interaction with the COREquations engine is completely invisible to the user, who simply sees an integrated system.

The block diagram in figure 2.1 shows the functionality aspect of the AlphaZ system framework from a user's perspective. The AlphaZ system takes in an Alphabets program as input. Within the AlphaZ system, the Alphabets program is parsed to create an Abstract Syntax Tree (AST) which acts as an intermediate representa-

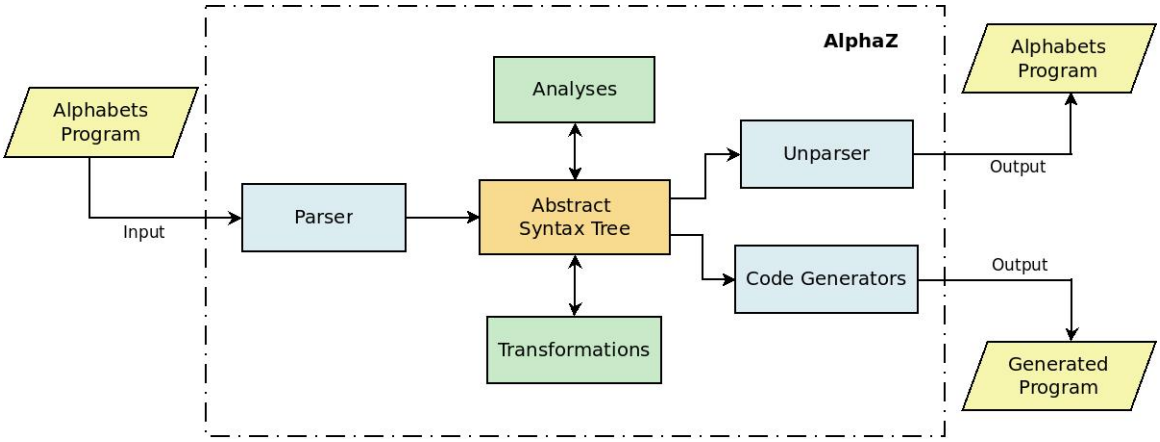


Figure 2.1: Block diagram showing the AlphaZ system framework from a user's perspective.

tion for the system. Transformations and analyses are applied on the AST to either modify or analyze the AST, respectively. The output from the AlphaZ system is obtained from the AST by either unparsing it to get another Alphabets program or by using a code generators to get programs that run on the specified architectures. The AlphaZ system is currently under active development and is used as a research tool. As a result, it has some limitations, which will be mentioned as needed.

### 2.1.1 Alphabets

Alphabets is an equational programming language which uses equations to specify the computations and domains to specify the computational space, and is based on the polyhedral model. It is an extension to the earlier Alpha [25, 38, 13] language and hence is very close in terms of syntax and semantics.

In this section, we introduce the key elements of the language necessary to understand this document. For a complete grammar of Alphabets, one may refer to [1].

#### 2.1.1.1 Structure of an Alphabets program

Figure 2.2 shows the syntactic structure of an Alphabets program. An Alphabets program begins with declarations for the external functions used in the program. This is followed by a number of Alphabets system declarations. The system declaration consists of a system name with the associated size parameters along with their

```
[External function declarations]
affine systemName {sizeParameters | sizeParameterDomains}
given
    [data type] inputVars {domain};
returns
    [data type] outputVars {domain};
using
    [data type] localVars {domain};
through
    [Equations defining outputVars and localVars]
.
```

Figure 2.2: Structure of an Alphabets program.

domains, a list of input, output and local variable declarations, and the equations defining the output and local variables. The keywords ***given, returns*** and ***using*** identify the input, output and local variable declarations respectively. Variable declaration consists of data type, variable name, and its domain, the set of index points where the variable is defined. The domain is a union of polyhedra, each one defined by a set of constraints. The ***through*** keyword identifies the functionality of the system which is specified using a list of equations. The period at the end marks the end of the system definition. An Alphabets program can contain more than one system in it.

#### 2.1.1.2 Domains in Alphabets

Domains in Alphabets take the form `{indexList | indexConstraints}`. As an example, consider a domain, $D_X = \{0 \leq i \leq j \leq N\}$, which in Alphabets syntax is written as `{i,j | 0<=i<=j<=N}`, where $i$ and $j$ are indices suggesting that the domain is two dimensional and $N$ is a parameter for the domain. Figure 2.3 shows the geometrical representation of the domain $D_X$ which illustrates that it is a triangle, due to the constraint $i \leq j$, on the two-dimensional $(i, j)$ plane.

#### 2.1.1.3 Equations in Alphabets

Equations in Alphabets are of the form `var = expr`; where `var` is the computed variable and `expr` is an Alphabets expression.

An Alphabets expression can be of the following forms,

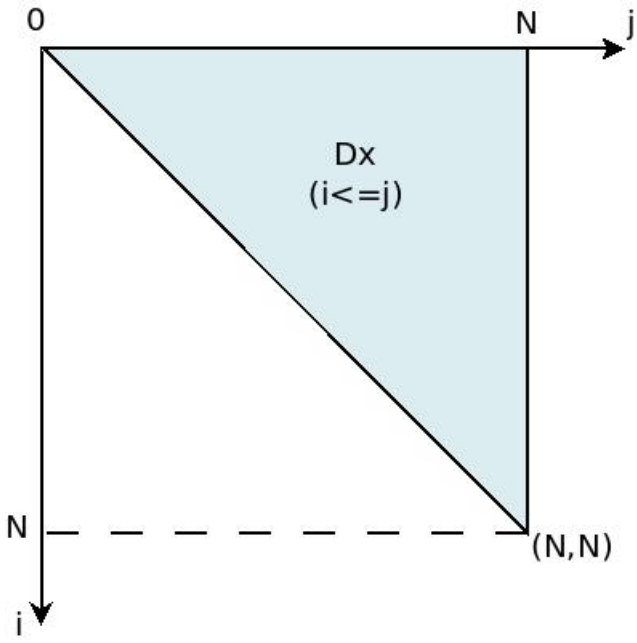


Figure 2.3: Geometrical representation of the domain $D_X$.

- **Atomic:** a variable or a constant.
- **Index:** *val*`(indexList->`*f*`(indexList))` or `[`*f*`(indexList)]` that is, the value of an affine function *f* applied to the list of indices.
- **Pointwise Operators:** There are three types of pointwise operators in Alphabets, as listed below.
  - **Operator:** `expr` *op* `expr`, where *op* is a binary operator.
  - **Conditional:** `if (condition) then` $\texttt{expr}_1$ `else` $\texttt{expr}_2$ where `condition` is an expression which evaluates to a `boolean` type. The conditional expression in Alphabets is strict, that is, both true and false cases should be defined.
  - **External function application:** `funcName(`$\texttt{expr}_1$`,` $\texttt{expr}_2$`, ... ,` $\texttt{expr}_N$`)` where `funcName` is the name of the external function to be called.
- **Case:** `case` $\texttt{expr}_1$`;` $\texttt{expr}_2$`; ...;` $\texttt{expr}_N$`; esac`, which is list of semi-colon separated expression enclosed by the keywords ***case*** and ***esac***.

- **Dependence:** `(indexList -> f(indexList))@expr` or `expr[f(indexList)]` where *f* is an affine dependence function.
- **Restriction:** `{domain}:expr` that is, `expr` is restricted to `domain`.
- **Reduction:** `reduce(op,(indexList->`$f_p$`(indexList),expr)` where *op* is a commutative and associative binary operator and $f_p$ is the projection function which specifies the projection along which the reduction is to applied. Alternate syntax for a reduction expression is *OP*`([`$f_p$`(indexList)], expr)` where *OP* is either *MAX*, *MIN* or *SUM*.

Also, `expr` can comprise of multiple Alphabets expressions nested together.

Equations in Alphabets are strict, which means `expr` should be defined at every point in the domain of `var`. The domain of the expression need not be specified unless its a restriction expression. In all other cases, AlphaZ evaluates their domains by using the rules of the polyhedral model.

### 2.1.2 Interactive Console

From an AlphaZ user perspective, the interactive console is an important part of the tool as it provides a means to interact with the underlying AlphaZ system. Currently, AlphaZ adopts BeanShell [28] console to export its transformations and analyses in the form of commands to the user. BeanShell also enables the user to apply a sequence of transformations on the input Alphabets program through scripts which call the required transformations in the form of commands. If a user does not prefer to use the interactive console, he/she can choose to adopt the command line interface by using the BeanShell Interpreter instead.

### 2.1.3 AlphaZ Utilities

The AlphaZ system provides many utilities to modify the input Alphabets program. These utilities are accessed though commands from the BeanShell console. AlphaZ broadly classifies these commands into six categories which are listed below.

1. Basic
2. Analysis
3. Transformation
4. Domain Calculator
5. Expression
6. TPM (Time, Processor and Memory) Specification

In this section, we just focus on the commands we use in our work. However, a complete guide on the commands can be accessed either through the `Help` utility or the AlphaZ website [27]. The following is the list of commands used in our work. More details on these commands are provided in the Appendix.

- `ConnectServer(String serverName)`
- `ReadProgram(String inputFilePath)`
- `Show()` and `AShow()`
- `Save(String outputFilePath)` and `ASave(String outputFilePath)`
- `RenameVariable(String sysName, String oldName, String newName)`
- `Normalize()`
- `FactorOutFromReduction(String sysName, String varName, int occr, int optr, int oprd)`
- `NormalizeReductions(boolean unique, String sysName, String varName, int occr)`
- `PermutationCaseReduce(String sysName, String varName)`
- `ReductionDecomposition(String sysName, String varName, int occr, String func1, String func2)`

- `RemoveUnusedVariables(String sysName, boolean removeInputs)`
- `SimplifyingReductions(String sysName, String varName, String reuse, int occr)`
- `SubstituteByDef(String sysName, String trgVar, String srcVar, int numOccr)`
- `WriteC(String sysName, String dirPath)`

## 2.2 RNA Secondary Structure Prediction

### 2.2.1 RNA Secondary Structure

RNA (Ribonucleic acid) molecule is made up of a single-stranded sequence of nucleotide units. These nucleotide units consist of a nitrogen base, a ribose sugar and a phosphate. The nitrogen bases in the nucleotide units have a tendency to chemically interact with one another to form covalent hydrogen bonds resulting in base pairs. This causes the nucleotide sequence to fold on itself. RNA nucleotide units typically constitute of four such nitrogen bases, namely, Adenine (A), Guanine (G), Cytosine (C) and Uracil (U). According to Watson-Crick base pairing rules, Adenine (A) can pair with Uracil (U) and Guanine (G) can pair with Cytosine (C). Under some special circumstances, Guanine (G) can pair with Uracil (U) which is called the wobble base pair. These base pairings among the bases in the sequence result in the secondary structure. In reality, knots and pseudoknots can occur, but in many algorithms that predict the secondary structure they are ignored for computational simplicity. A pseudoknot free folding is planar. In order to maintain the planar structure, pseudoknots are avoided by imposing a restriction on base pairing rules where a base pair cannot share bases with any other base pair in the structure and also, that a base can contribute to only one sub-structure.

Figure 2.4 shows a typical RNA secondary structure comprising of all the different sub-structures. The sub-structure with two consecutive bases pairs is called a *stacked pair* or a *stack*. The sub-structure with one closing base pair and no enclosed base

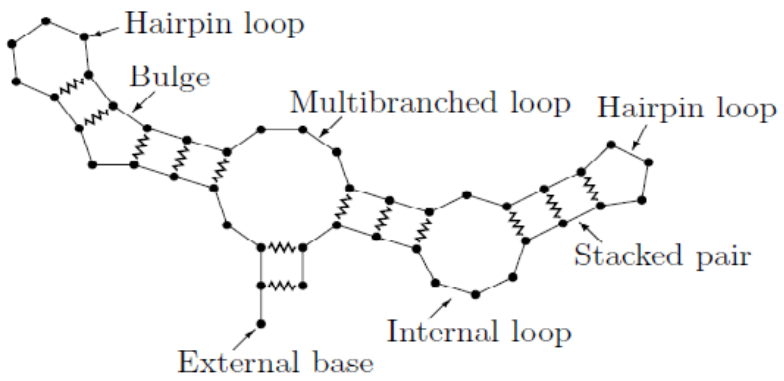


Figure 2.4: Sub-structures within the RNA secondary structure [20].

pairs is called a *hairpin loop*. The sub-structure with one closing base pair and exactly one enclosed base pair is called an *interior loop*. An interior loop having a length of zero on one of the sides is called a *bulge*. The sub-structure with one closing base pair and more than one enclosed base pairs is called a *multibranched loop* or a *multiloop*. Any unpaired base or base pair which does not belong to any of the sub-structures is called as *external* or *dangle*.

### 2.2.2 Prediction Algorithm based on Thermodynamic model

Thermodynamic based RNA secondary structure prediction algorithm finds the optimal structure for a given RNA sequence with minimum free energy. It is devised around dynamic programming [8] techniques to optimize the associated cost function, which is the free energy for a subsequence $(i, j)$ where $i$ and $j$ are the nucleotide (base) indices and $1 \leq i < j \leq N$ for a RNA sequence $s$ with $N$ nucleotides (bases). The free energy of subsequences for the different kinds of substructures are stored in three data arrays, namely, $Q$, $Q'$ and $QM$, which are then traced back to find the optimal structure.

The recurrence equations to fill these arrays, from [21], are presented below for convenience.

For $1 \leq i < j \leq N$, we have

$$Q(i,j) = \min \begin{cases} b + Q(i+1,j), \\ b + Q(i,j-1), \\ c + E_{ND}(i,j) + Q'(i,j), \\ b + c + E_{5'D}(i+1,j) + Q'(i+1,j), \\ b + c + E_{3'D}(i,j-1) + Q'(i,j-1), \\ 2b + c + E_{DD}(i+1,j-1) + Q'(i+1,j-1), \\ QM(i,j) \end{cases} \tag{2.1}$$

$$Q'(i,j) = \min \begin{cases} E_H(i,j) \\ E_S(i,j) + Q'(i+1,j-1) \\ \min\limits_{i<i'<j'<j} \{E_{BI}(i,j,i',j') + Q'(i',j')\} \\ a + c + E_{ND}(j,i) + QM(i+1,j-1) \\ a + b + c + E_{3'D}(j,i) + QM(i+2,j-1) \\ a + b + c + E_{5'D}(j,i) + QM(i+1,j-2) \\ a + 2b + c + E_{DD}(j,i) + QM(i+2,j-2) \end{cases} \tag{2.2}$$

$$QM(i,j) = \min_{i+1 \le k \le j-2} \{Q(i,k-1) + Q(k,j)\} \tag{2.3}$$

where $a$, $b$, $c$ are constants for calculating energy of a multibranched loop and $E_S$, $E_H$, $E_{BI}$, $E_{5'D}$, $E_{3'D}$, $E_{ND}$, $E_{DD}$ are energy functions contributing for different sub-structures. $E_S(i,j)$ is the energy of the stacked pair with $[i.j]$ as the closing base pair and $[i+1.j-1]$ as the enclosed base pair. $E_H(i,j)$ is the energy of the hairpin loop with $[i.j]$ as the closing base pair. $E_{BI}(i,j,i',j')$ is the energy of the internal loop (including bulges) with $[i.j]$ as the closing base pair and $[i'.j']$ as the enclosed base pair. $E_{5'D}(i,j)$ is the terminal stacking energy of the base pair $[i.j]$ with the unpaired base $i-1$ at $5'$ end. $E_{3'D}(i,j)$ is the terminal stacking energy of the base pair $[i.j]$ with the unpaired base $j+1$ at $3'$ end. $E_{ND}(i,j)$ is the terminal stacking energy of the base pair $[i.j]$ with the no dangling or unpaired bases at either $3'$ end or $5'$ end. $E_{DD}(i,j)$is the terminal stacking energy of the base pair $[i.j]$ with unpaired bases $i-1$ and $j+1$ at both $5'$ and $3'$ ends respectively.

When the third term in equation (2.2) is observed, we notice that its computational domain is four dimensional, as indicated by the presence of four free index variables, i, j, i' and j'. As a result, the computational complexity of this algorithm is $O(N^4)$. In order to avoid huge execution times, a common heuristic is adopted.

The heuristic is to restrict the size of internal loops to $k$ which is usually 30 [17]. With this heuristic, the complexity of evaluating internal loops reduces to $O(k^2N^2)$. However, in 1999, Lyngso et. al. [20] derived a new method of evaluating internal loops with $O(N^3)$ complexity which does not restrict the size of internal loops to $k$. With $O(N^3)$, the execution time is still large. But, when the heuristic is applied to $O(N^3)$ algorithm, the complexity reduces $O(kN^2)$, thereby reducing the execution time. However, the $O(N^3)$ algorithm has never been implemented in the public release of the software.

## 2.3 Related Work

Many algorithms exist for the RNA secondary structure prediction problem along with the sequential and parallel implementations. A web server based application Mfold [41], written in FORTRAN and C, implements the algorithm [43, 42] based on calculating minimum free energy of the structure using thermodynamic data from Mathews et. al. [23]. The UNAFold software [22] is an extension and update to Mfold written in C and Perl using the algorithms described in [21] which forms the basis for our work.

The Vienna RNA package [17] implements prediction of secondary structure using two approaches, minimum free energy [43, 42, 40] and partition functions [26], and targets both sequential and parallel architectures with distributed memory.

GTfold [24] implements the minimum free energy algorithm [43, 42, 40] along with the optimization proposed by Lyngso et. al. [20] on shared memory architectures using OpenMP [3] parallelization. The work of Jacob et. al. [18] implements the maximum base pairing algorithm by Nussinov [29] on Field Programmable Gate Arrays (FPGAs) using systolic array synthesis [31]. Their recent work [19] tackles the implementation of Zuker algorithm [40] on FPGAs. Rizk et. al. [37, 36] implemented the algorithms in UNAFold [22] on Graphic Processing Units (GPUs) exploiting parallelism at multiple levels using CUDA [30].

More recent efforts [32, 35, 34, 39, 4, 9, 10, 11, 12] involve the realization of al-

gorithms predicting secondary structures including pseudoknots which usually tend to have a computational complexity of $O(N^5)$. NUPACK [39, 10, 11, 12], pknotsRG [33, 34] and pknotsRE [35] are some of applications which implement these algorithms.

# Chapter 3

# Implementation

The RNA secondary structure prediction algorithm proposed in [21], is currently implemented using C in the UNAFold software package [22]. In this chapter, we will systematically implement the RNA secondary structure prediction algorithm in AlphaZ. The flowchart in figure 3.1 illustrates the procedure that we followed to realize this implementation. We start by profiling the C program which implements the RNA secondary structure prediction algorithm to identify the compute-intensive parts. The compute-intensive part is rewritten in Alphabets by using the equations extracted from the C program. The Alphabets program is then loaded into the AlphaZ system to carry out the necessary transformations for optimizing the algorithm. This is followed by code generation. The generated code is then plugged back into the UNAFold C program to replace the compute-intensive part. The following sections in this chapter explain these steps in detail.

## 3.1 Profiling the UNAFold program

For the purpose of time profiling, the UNAFold program for RNA secondary structure prediction consists of the four following modules as shown in figure 3.2 .

1. **Input module:** This module comprises of functions to parses command line inputs and read the input file containing the RNA sequence.

2. **Energy Table Initialization (ETI) module:** This module comprises of functions to initialize the required energy tables which assist in energy function

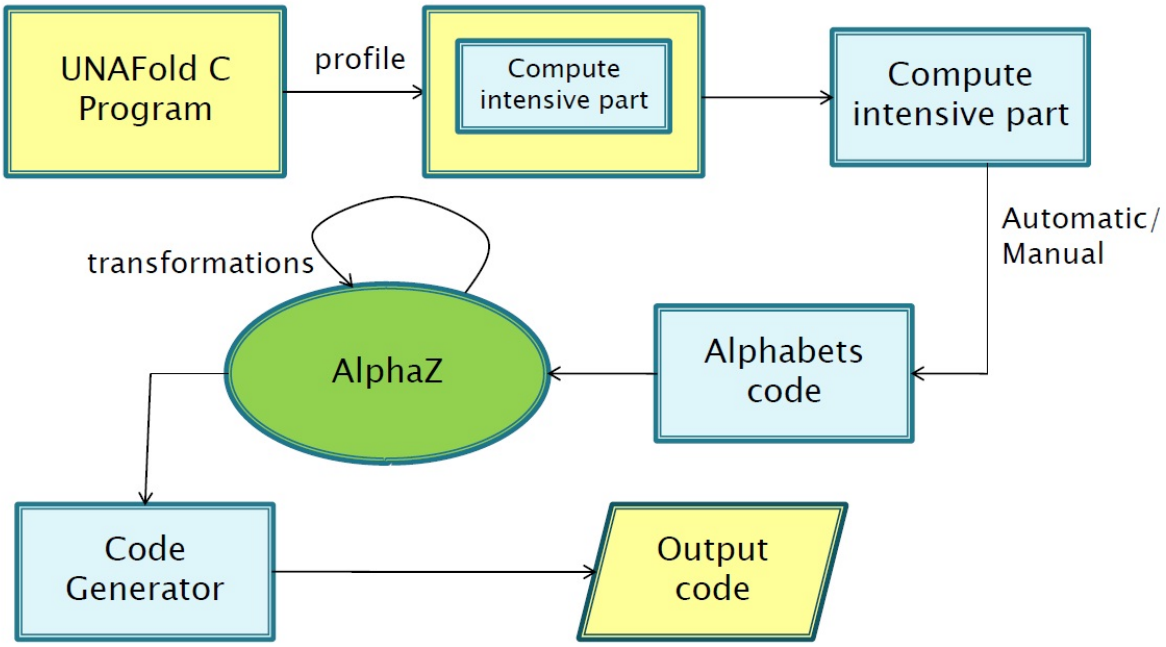


Figure 3.1: Flowchart showing the entire higher level process of using AlphaZ.

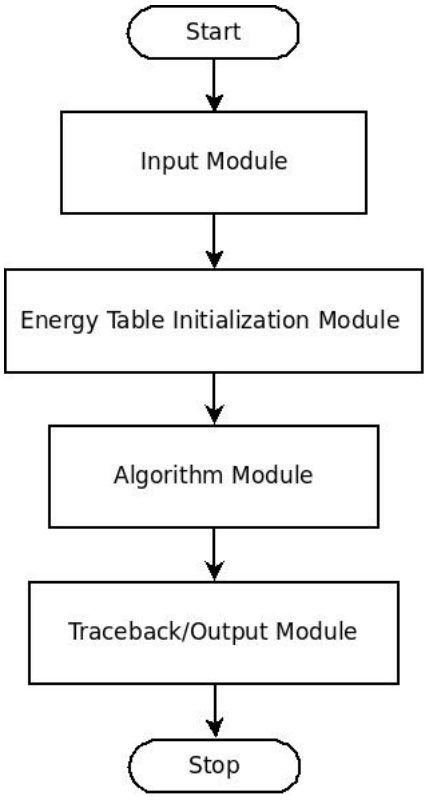


Figure 3.2: UNAFold code structure

| Sequence length | Input Module | ETI Module | Algorithm Module | T/O Module | Total | |
|---|---|---|---|---|---|---|
| 1000 | 1 | 55 | 1678 | 15 | 1749 | Time(ms) |
| | 0.057 | 3.145 | 95.940 | 0.858 | 100 | Percentage |
| 5000 | 1 | 631 | 225467 | 54 | 226153 | Time(ms) |
| | 4.4e-4 | 0.279 | 99.697 | 0.024 | 100 | Percentage |
| 10000 | 1 | 2401 | 2661763 | 242 | 2664407 | Time(ms) |
| | 4.0e-5 | 0.090 | 99.901 | 0.009 | 100 | Percentage |

Table 3.1: Time profile for UNAFold C program.

calculations.

3. **Algorithm module:** This module comprises of one function namely *fillMatrices1()* which implements the RNA secondary structure prediction algorithm and fills the three arrays $Q$, $Qprime$ and $QM$ to compute the minimum free energy and the RNA secondary structure.

4. **Traceback/Output (T/O) module:** This module comprises of functions for traceback to reconstruct the optimal structure from the three arrays $Q$, $Qprime$ and $QM$. It also writes the optimal secondary structure for the RNA sequence to an output file.

The UNAFold program for RNA secondary structure prediction is profiled to measure the time spent in each of these modules. The purpose of this time profile is to validate our hypothesis that the algorithm module accounts for majority of the execution time. This implies that the algorithm module is to be rewritten in Alphabets so that it can be optimized using AlphaZ. It also implies that the other modules can be reused in their original form without the necessity for being rewritten. Table 3.1 shows the time profile with varying sequence lengths. It can be seen that about 99 percent of the time is spent in the algorithm module which calculates the $Q$, $Qprime$ and $QM$ arrays. Hence, it can be deduced that the algorithm for filling these arrays is the compute intensive part of the UNAFold program for RNA secondary structure prediction.

| Module | Source Lines of Code |
|---|---|
| Input | 410 |
| Energy Table Initialization | 1070 |
| Algorithm | 120 |
| Traceback/Output | 1500 |
| **Total** | **3100** |

Table 3.2: SLOC for UNAFold C program

We use the Source Lines of Code (SLOC) metric to provide an estimate of the size of the UNAFold C program, namely, `hybrid-ss-min.c` as show in the table 3.2 . From this table, we see that a large part of the code implements the Input, Energy Table Initialization and Traceback/Output modules. The part of the code implementing the Algorithm module is just 120 lines. This implies that we can reuse 2980 lines of source code from the UNAFold C program with slight modifications which make it compatible with the code generated from AlphaZ.

## 3.2 Extracting Equations

The equations for RNA secondary structure prediction described in section 2.2.2 cannot be directly translated to an Alphabets program as they do not have the necessary and complete domain specification associated with them. Hence, there is a need to extract these equations from the C program, `hybrid-ss-min.c`, of the UNAFold software package. We can extract the complete and exact domains for these equations by inspecting the iteration spaces of these computations in the C program. These new equations with their domain specifications can now be written as an Alphabets program which forms the basis for the rest of our work.

The equational specification representing the RNA secondary structure prediction algorithm can be extracted from the C program in two ways namely, automatic and manual. The automatic extraction term is a bit misleading as the tools which implement this require the input C program to comply with specific formats. This is mainly because these tools are still under development and extracting equations from a generic C program is a difficult problem. One such tool is FADA (Fuzzy

```
void foo (int N, int *A, int *B, int *C)
{
    int i;
    for (i = 1; i < N; i++)
    {
        if (i < 10)
        {
            C[i] = A[i] * B[i];
        }
        else
        {
            C[i] = A[i] + B[i];
        }
    }
}
```

Figure 3.3: A Simple C function

Array Dependency Analysis) [6, 7], which performs a dataflow analysis to track the definitions and uses of variable values in order to compute an equivalent equational specification of the input C program. However, FADA tool has some strict constraints on the input C program. It requires that the C program be static, that is, the control flow in the C program should be known at compile time, and also that the array subscripts for the variables be affine functions of the surrounding loop indices.

In this thesis, we take the manual approach to extracting equations in order to avoid rewriting the C program to make it FADA tool friendly. The manual approach is also not simple as it involves extracting the exact domains for the computations by analyzing the loop bounds surrounding the computation in the C program. Care has to be taken to account for the conditional statements, if any, within the loop body which affect the iteration space of the computation. The following section explains how to deduce equations from a simple C function.

### 3.2.1 A Simple Example

Consider the C program snippet in figure 3.3. This C function has a one dimensional *for* loop construct with a *conditional if-else* construct within its body surrounding the statements specifying the computations. The Alphabets equivalent of this C function is shown in figure 3.4. The function name *foo* translates to the system name

```
affine foo {N | N > 0}
given
    int A {i | 0 < i < N};
    int B {i | 0 < i < N};

returns
    int C {i | 0 < i < N};

through
    C[i] = case
               {| i < 10}:  A[i] * B[i];
               {| i >= 10}:  A[i] + B[i];
           esac;
.
```

Figure 3.4: Alphabets equivalent of the C function in figure 3.3

in Alphabets. The arguments in the function definition translate to input/output variables and parameter specifications, that is, $A$ and $B$ are declared as inputs (specified by the keyword ***given***), $C$ is declared as an output (specified by the keyword ***returns***) and $N$ is declared as a parameter for the system *foo*. The domains for these input/output variables are deduced from the *for* loop construct in the C function, specifically, the range of the loop variable $i$. In this example, we see that the range of $i$ is from 1 to $N$, hence the domain corresponding to variables $A$, $B$ and $C$ is $1 \leq i < N$ which in Alphabets specification is written as $\{i|1 \leq i < N\}$. Note that in this example, the *conditional if-else* construct does not affect the domains of the input/output variables because computations are defined over the entire range of the *for* loop. However, it does affect which computational statement is contributing to the result $C$ for specific regions of the iteration space. By analyzing the conditional in the *if* statement, we see that for the region of the iteration space where $i < 10$, $C[i] = A[i] * B[i]$; is the computational statement, and for the remainder of the iteration space it is $C[i] = A[i] + B[i]$;. The *conditional if-else* construct in the C function translates to case statements (enclosed within the keywords ***case*** and ***esac***) with domain restrictions corresponding to the conditional in the *if* statement. However, in C, the *else* statement spans the remainder of the loop iterations where the *if* statement evaluates to a false. In Alphabets, this has to specified in the form of a domain which is the relative complement of domain corresponding to the

```
affine fillMatrices1 {N, MAXLOOP|N>4 && MAXLOOP > 0}
given
int valid_pairs {i,j | 1 <= i <= N && 2 <= j <= N};

returns
int Q {i,j | 1 <= i <= N && 2 <= j <= N};
int Qprime {i,j | 1 <= i <= N && 2 <= j <= N};
int QM {i,j | 1 <= i <= N && 2 <= j <= N};

using
int QBI {i,j | 1 <= i <= N && 2 <= j <= N};
```

Figure 3.5: Alphabets system declaration for fillMatrices1

*if* statement, represented by $D_{if}$, in the domain of $C$, represented by $D_C$, that is, $D_{else} = D_C - D_{if}$ where $D_{else}$ is the domain corresponding to the *else* statement. Alternatively, the C function represents the following equation,

$$C[i] = \begin{cases} i < 10 & A[i] * B[i] \\ i >= 10 & A[i] + B[i] \end{cases} \qquad for\ 1 \leq i < N \tag{3.1}$$

Through equation (3.1), we see that the Alphabets program is very identical to the actual equations implemented by the C program. Hence, the claim that it is easier to write algorithms in Alphabets directly though equations is solidified.

### 3.2.2 UNAFold Equations

We follow the same analysis, as the example in the previous section, on the *fillMatrices1()* function (refer to Appendix) of the UNAFold C program which implements the RNA secondary structure prediction algorithm. We deduce that the Alphabets program will comprise of affine system named *fillMatrices1* which takes one input *valid_pairs* and produces three outputs $Q$, $Qprime$ and $QM$ by using a local variable $QBI$. The input *valid_pairs* holds the information about which bases in the input RNA sequence can form base pairs $(i.j)$ according to the Watson-crick base pairing rules. All these variables share the same domain, that is, $(1 \leq i \leq N, 2 \leq j \leq N)$. The *fillMatrices1* system takes $N$ and $MAXLOOP$ as input parameters, where $N$ is the length of the RNA sequence and $MAXLOOP$ is the maximum size of the internal loop (previously referred as $k$). 3.5 shows the corresponding Alphabets code

fragment. The details of the computations involving each of these variables follow.

The values of the array $Q$ are computed according to equation (3.2) given below.

$$Q(i,j) = \begin{cases} i < j-3 & \min \begin{cases} b + Q(i+1,j) \\ b + Q(i,j-1) \\ c + E_{ND}(i,j) + Q'(i,j) \\ QM(i,j) \\ b + c + E_{5'D}(i+1,j) + Q'(i+1,j) \\ b + c + E_{3'D}(i,j-1) + Q'(i,j-1) \\ 2b + c + E_{DD}(i+1,j-1) + Q'(i+1,j-1) \end{cases} \\ i \geq j-3 & \infty \end{cases} \tag{3.2}$$

where $E_{DD}(i,j) = E_{ND}(i,j) + E_{tstackm}(j,i)$ and the last three cases of the *min* term are considered only if $dangle = 1$. Also, if $noisolate = 1$, the occurrence of $Q'(m,n)$ is replaced by $E_S(m,n) + Q'(m+1,n-1)$.

The Alphabets code fragment corresponding to equation (3.2) is shown in figure 3.6

The values of the array $Q'$ are computed according to equation (3.3) given below.

$$Q'(i,j) = \begin{cases} i < j-3 & \begin{cases} \min \begin{cases} E_H(i,j) \\ E_S(i,j) + Q'(i+1,j-1) \\ QBI(i,j) \\ a + c + E_{ND}(j,i) + QM(i+1,j-1) & if\ i.j\ pair \\ a + b + c + E_{3'D}(j,i) + QM(i+2,j-1) \\ a + b + c + E_{5'D}(j,i) + QM(i+1,j-2) \\ a + 2b + c + E_{DD}(j,i) + QM(i+2,j-2) \end{cases} \\ \infty & otherwise \end{cases} \\ i \geq j-3 & \infty \end{cases} \tag{3.3}$$

where the last three cases of the *min* term are considered only if $dangle = 1$ and

$$QBI(i,j) = \begin{cases} i < j-6 & \min\limits_{4 \leq d \leq j-i-3,\, i<i'<j-d} \{E_{BI}(i,j,i',i'+d) + Q'(i',i'+d)\} \\ i \geq j-6 & \infty \end{cases} \tag{3.4}$$

```
Q[i,j] = case {|i>=j-3}:  INFINITY_VAL(0);
              {|i<j-3}:  min((b(0) + Q[i+1,j]),
                             (b(0) + Q[i,j-1]),
                             (if (noisolate(0) == 1) then (c(0) + End([i],[j]) +
Es([i],[j]) + Qprime[i+1,j-1]) else (c(0) + End([i],[j]) + Qprime[i,j])),
                             QM[i,j],
                             (if (nodangle(0) == 0) then
                                 (if (noisolate(0) == 1) then
                                     (min((if ([i] < ([j]-6)) then (min((b(0) +
c(0) + End(([i]+1),[j]) + Ed5([j],([i]+1)) + Es(([i]+1),[j]) + Qprime[i+2,j-1]),
(b(0) + c(0) + End([i],([j]-1)) + Ed3(([j]-1),[i]) + Es([i],([j]-1)) +
Qprime[i+1,j-2]))) else INFINITY_VAL(0)),
                                          (if ([i] < ([j]-7)) then ((2*b(0)) +
c(0) + End(([i]+1),([j]-1)) + Etstackm(([j]-1),([i]+1)) + Es(([i]+1),([j]-1)) +
Qprime[i+2,j-2]) else INFINITY_VAL(0))))
                                     else (min((b(0) + c(0) + End(([i]+1),[j]) +
Ed5([j],([i]+1)) + Qprime[i+1,j]),
                                               (b(0) + c(0) + End([i],([j]-1)) +
Ed3(([j]-1),[i]) + Qprime[i,j-1]),
                                               (case {|i<j-3-2}:  ((2*b(0)) + c(0) +
End(([i]+1),([j]-1)) + Etstackm(([j]-1), ([i]+1)) + Qprime[i+1,j-1]);
                                                     {|i>=j-5}:  INFINITY_VAL(0);
                                                esac))))
                                 else INFINITY_VAL(0)));
         esac;
```

Figure 3.6: Alphabets equivalent of equation (3.2)

where the occurrence of $Q'(m,n)$ is replaced by $E_S(m,n) + Q'(m+1,n-1)$ if $noisolate = 1$.

The corresponding Alphabets code fragment for equations (3.3) and (3.4) are shown in figures 3.7 and 3.8 .

The values of the array $QM$ are computed according to the equation (3.5) given below.

$$QM(i,j) = \begin{cases} i < j-8 & \min\limits_{i+4 \le k \le j-5} \{Q(i,k-1) + Q(k,j)\} \\ i \ge j-8 & \infty \end{cases} \tag{3.5}$$

Figure 3.9 shows the equivalent Alphabets code fragment for equation (3.5).

In all the above equations, the energy functions $E_S$, $E_H$,$E_{BI}$,$E_{ND}$,$E_{3'D}$,$E_{5'D}$ and $E_{tstackm}$ are specified as external functions in the Alphabets program. These external functions refer to the respective functions defined in the UNAFold C program.

Before we move on to optimizations, we restrict the RNA secondary structure prediction algorithm by ignoring special cases like *dangle* and *noisolate* as it helps

```
Qprime[i,j] = case {|i>=j-3}:  INFINITY_VAL(0);
                   {|i<j-3}:  if (Eval_isFinite(valid_pairs[i,j]) > 0) then
                                  min(Eh([i],[j]),
                                      (Es([i],[j]) + Qprime[i+1,j-1]),
                                      QBI[i,j],
                                      (a(0) + c(0) + End([j],[i]) + QM[i+1, j-1]),
                                      if ((nodangle(0) == 0) && ([j] > 2))
then (a(0) + b(0) + c(0) + End([i],[j]) + Ed5([i],[j]) + QM[i+1,j-2]) else
INFINITY_VAL(0),
                                      if ((nodangle(0) == 0) && ([i] < ([N]-1)))
then (a(0) + b(0) + c(0) + End([i],[j]) + Ed3([i],[j]) + QM[i+2,j-1]) else
INFINITY_VAL(0),
                                      if ((nodangle(0) == 0) && ([j] > 2) && ([i]
< ([N]-1))) then (a(0) + (2 * b(0)) + c(0) + End([i],[j]) + Etstackm([i],[j])
+ QM[i+2,j-2]) else INFINITY_VAL(0) )                              else
INFINITY_VAL(0);
             esac;
```

Figure 3.7: Alphabets equivalent of equation (3.3).

```
QBI[i,j] = case {|i>=j-6}:  INFINITY_VAL(0);
                {|i<j-6} :  MIN([d,ip], {|d<=j-i-3 && d>=3+1 &&
d>=j-i-2-MAXLOOP && ip>i && ip<j-d && ip<=N}:  if (noisolate(0) == 1) then
(Ebi([i],[j],[ip],[ip+d]) + Es([ip],[jp]) + Qprime[ip+1,ip+d-1]) else
(Ebi([i],[j],[ip],[ip+d]) + Qprime[ip,ip+d]));
           esac;
```

Figure 3.8: Alphabets equivalent of equation (3.4).

```
QM[i,j] = case {|i>=j-8}:  INFINITY_VAL(0);
               {|i<j-8}:  MIN([k], {|i+3+1<=k<=j-3-2}:  (Q[i,k-1] + Q[k,j]));
          esac;
```

Figure 3.9: Alphabets equivalent for equation (3.5)

```
Q[i,j] = case {|i>=j-3}:  INFINITY_VAL(0);
              {|i<j-3}:  min((b(0) + Q[i+1,j]),
                             (b(0) + Q[i,j-1]),
                             (c(0) + End([i],[j]) + Qprime[i,j]),
                             QM[i,j]);
          esac;
```

Figure 3.10: Alphabets equivalent of equation (3.6)

```
Qprime[i,j] = case {|i>=j-3}:  INFINITY_VAL(0);
                   {|i<j-3}:  if (Eval_isFinite(valid_pairs[i,j]) > 0) then
                                 min(Eh([i],[j]),
                                     (Es([i],[j]) + Qprime[i+1,j-1]),
                                     QBI[i,j],
                                     (a(0) + c(0) + End([j],[i]) + QM[i+1, j-1]))
                              else INFINITY_VAL(0);
              esac;
```

Figure 3.11: Alphabets equivalent for equation (3.7)

to illustrate, with simplicity, the transformations applied to optimize the algorithm. The resulting equations (3.6), (3.7), (3.8), (3.9) for $Q$, $Q'$, $QBI$ and $QM$ are given below along with their Alphabets equivalents in figures 3.10 , 3.11 , 3.12 and 3.13 respectively. Also, note that the equation for $QM$ is unchanged since it is independent of *dangle* and *noisolate* cases.

$$Q(i,j) = \begin{cases} i < j-3 & \min \begin{cases} b + Q(i+1,j) \\ b + Q(i,j-1) \\ c + E_{ND}(i,j) + Q'(i,j) \\ QM(i,j) \end{cases} \\ i \geq j-3 & \infty \end{cases} \tag{3.6}$$

```
QBI[i,j] = case {|i>=j-6}:  INFINITY_VAL(0);
                {|i<j-6} :  MIN([d,ip], {|d<=j-i-3 && d>=3+1 && d>=j-i-2-MAXLOOP
&& ip>i && ip<j-d && ip<=N}:  (Ebi([i],[j],[ip],[ip+d]) + Qprime[ip,ip+d]));
           esac;
```

Figure 3.12: Alphabets equivalent for equation (3.8)

```
QM[i,j] = case {|i>=j-8}:  INFINITY_VAL(0);
               {|i<j-8}:  MIN([k], {|i+3+1<=k<=j-3-2}:  (Q[i,k-1] + Q[k,j]));
          esac;
```

Figure 3.13: Alphabets equivalent for equation (3.9)

$$Q'(i,j) = \begin{cases} i < j-3 & \begin{cases} \min \begin{cases} E_H(i,j) \\ E_S(i,j) + Q'(i+1,j-1) \\ QBI(i,j) \\ a + c + E_{ND}(j,i) + QM(i+1,j-1) \end{cases} & if\ i.j\ pair \\ \infty & otherwise \end{cases} \\ i \geq j-3 & \infty \end{cases} \tag{3.7}$$

$$QBI(i,j) = \begin{cases} i < j-6 & \min\limits_{4 \leq d \leq j-i-3, i<i'<j-d} \{E_{BI}(i,j,i',i'+d) + Q'(i',i'+d)\} \\ i \geq j-6 & \infty \end{cases} \tag{3.8}$$

$$QM(i,j) = \begin{cases} i < j-8 & \min\limits_{i+4 \leq k \leq j-5} \{Q(i,k-1) + Q(k,j)\} \\ i \geq j-8 & \infty \end{cases} \tag{3.9}$$

## 3.3 Optimizations

The following types of optimizations can be done using AlphaZ.

1. Exploiting re-use in reductions.
2. Specifying a memory mapping.
3. Specifying a processor allocation.
4. Specifying a schedule.
5. Tiling

Of all the above optimizations, we explore the first one, that is, to exploit the re-use in reductions. We do not explore the other optimizations since we are currently

limited to code generation using *WriteC* which is a demand-driven code generator. More details on this issue will be covered in section 3.4.

Referring to equation (3.8), we see that the domain of the reduction expression is four dimensional over $N$. Hence, it has a computational complexity of $O(N^4)$. As as result, the complexity of the entire algorithm is $O(N^4)$. Lyngso et. al. [20] has derived a $O(N^3)$algorithm by exploring reuse of partial results in the reduction term of *QBI*. This result is classically known as *fast internal loop evaluation* or *fast-i-loops*. We provide a systematic derivation and proof of this result using the transformations in AlphaZ, thereby arriving at the $O(N^3)$ algorithm. A similar derivation is presented in [15], but has never been implemented using AlphaZ.

### 3.3.1 Mathematical derivation of Fast-i-loops

The equation for evaluating internal loops is reproduced here for convenience.

$$QBI(i,j) = \begin{cases} i < j-6 & \min\limits_{4 \le d \le j-i-3, i<i'<j-d} \{E_{BI}(i,j,i',i'+d) + Q'(i',i'+d)\} \\ i \ge j-6 & \infty \end{cases} \tag{3.10}$$

To expose the reuse in the reduction expression of the equation (3.10),we need to split the $E_{BI}$ term into its constituents. The $E_{BI}$ term accounts for the energy values of the internal loops based on the asymmetry, size, closing base pair and enclosed base pair. Hence, $E_{BI}$ is split into the following constituents,

1. Asymmetry penalty, $Asym$, of the internal loop.
2. Size penalty, $Sp$, depending on the size of the internal loop.
3. Stacking energy, $E_S$, of the closing base pair $(i.j)$.
4. Stacking energy, $E_S$, of the enclosed base pair $(i'.j')$.

This is represented by the equation (3.11) given below,

$$E_{BI}(i,j,i',j') = Asym(i'-i-1, j-j'-1) + S_P(i'-i+j-j'-2) + E_S(i,j) + E_S(i',j') \tag{3.11}$$

Note that, special case of internal loops like bulges and internal loops of $size < 4$ are handled differently. Hence, we rewrite equation (3.11) as

$$E_{BI}(i,j,i',j') = \begin{cases} E_{BI_{special}}(i.,j,i',j') & for\, special\, cases \\ E_{BI_{generic}}(i,j,i',j') & for\, generic\, case \end{cases} \tag{3.12}$$

where

$$E_{BI_{generic}} = Asym(i'-i-1, j-j'-1)+S_P(i'-i+j-j'-2)+E_S(i,j)+E_S(i',j') \tag{3.13}$$

Reuse within the reduction expression is exploited in the generic case. Substituting equation (3.13) in equation(3.10), we get,

$$QBI(i,j) = \begin{cases} i < j-6 & \min\limits_{4\le d\le j-i-3, i<i'<j-d} \left\{ \begin{array}{l} Asym(i'-i-1, j-i'-d-1) \\ +S_P(j-i-d-2) \\ +E_S(i,j) \\ +E_S(i',i'+d) \\ +Q'(i',i'+d) \end{array} \right\} \\ i \ge j-6 & \infty \end{cases} \tag{3.14}$$

For the sake of simplicity, we extract the first case in equation (3.14) and present it below,

$$QBI(i,j) = \min_{4\le d\le j-i-3, i<i'<j-d} \left\{ \begin{array}{l} Asym(i'-i-1, j-i'-d-1) \\ +S_P(j-i-d-2) \\ +E_S(i,j) \\ +E_S(i',i'+d) \\ +Q'(i',i'+d) \end{array} \right\} \tag{3.15}$$

In equation (3.15), we see that the term $E_S(i,j)$ is independent of $d$ and $i'$. Also, since *addition* distributes over *min*, we can distribute out $E_S(i,j)$ from the reduction. As as result, we have,

$$QBI(i,j) = E_S(i,j) + \min_{4\le d\le j-i-3, i<i'<j-d} \left\{ \begin{array}{l} Asym(i'-i-1, j-i'-d-1) \\ +S_P(j-i-d-2) \\ +E_S(i',i'+d) \\ +Q'(i',i'+d) \end{array} \right\} \tag{3.16}$$

Since the reduction in equation (3.16) is a double reduction, we decompose it into two reductions to exploit further reuse. The resulting equation after decomposing reductions is given below,

$$QBI(i,j) = E_S(i,j) + \min_{4\le d\le j-i-3} \left\{ \min_{i<i'<j-d} \left\{ \begin{array}{l} Asym(i'-i-1, j-i'-d-1) \\ +S_P(j-i-d-2) \\ +E_S(i',i'+d) \\ +Q'(i',i'+d) \end{array} \right\} \right\} \tag{3.17}$$

Now, let us isolate the inner reduction expression from equation (3.17). We have,

$$QBI(i,j) = E_S(i,j) + \min_{4\le d\le j-i-3} \{X(i,j,d)\} \tag{3.18}$$

where

$$X(i,j,d) = \min_{i<i'<j-d} \left\{ \begin{array}{l} Asym(i'-i-1, j-i'-d-1) \\ +S_P(j-i-d-2) \\ +E_S(i',i'+d) \\ +Q'(i',i'+d) \end{array} \right\} \tag{3.19}$$

In equation (3.19), we see that $S_P(j-i-d-2)$ is independent of $i'$. Hence, $S_P(j-i-d-2)$ is distributed out of the reduction resulting in,

$$X(i,j,d) = S_P(j-i-d-2) + \min_{i<i'<j-d} \left\{ \begin{array}{l} Asym(i'-i-1, j-i'-d-1) \\ +E_S(i',i'+d) \\ +Q'(i',i'+d) \end{array} \right\} \tag{3.20}$$

Now, let us replace the reduction in equation (3.20) by $Y(i,j,d)$. Hence, we have,

$$X(i,j,d) = S_P(j-i-d-2) + Y(i,j,d) \tag{3.21}$$

where

$$Y(i,j,d) = \min_{i<i'<j-d} \left\{ \begin{array}{l} Asym(i'-i-1, j-i'-d-1) \\ +E_S(i',i'+d) \\ +Q'(i',i'+d) \end{array} \right\} \tag{3.22}$$

In equation (3.22), we detect a scan in the reduction along $i+j$. This scan reduces the complexity for evaluating internal loops by re-using previously calculated partial results. Hence, we can rewrite equation (3.22) as,

$$Y(i,j,d) = \min \left\{ \begin{array}{l} Y(i+1,j-1,d) \\ \\ \min\limits_{i+1<i'<j-d-1} \left\{ \begin{array}{l} Asym(i'-i-1, j-i'-d-1) \\ +E_S(i',i'+d) \\ +Q'(i',i'+d) \end{array} \right\} \end{array} \right. \tag{3.23}$$

```
affine fillMatrices1 {N, MAXLOOP|N>4 && MAXLOOP > 0}
given
int valid_pairs {i,j | 1 <= i <= N && 2 <= j <= N};

returns
int Q {i,j | 1 <= i <= N && 2 <= j <= N};
int Qprime {i,j | 1 <= i <= N && 2 <= j <= N};
int QM {i,j | 1 <= i <= N && 2 <= j <= N};

using
int QBI {i,j | 1 <= i <= N && 2 <= j <= N};
int Ebi {i,j,ip,jp | 1 <= i < ip < jp < j <= N && ip-i+j-jp-2 <= MAXLOOP};
```

Figure 3.14: *fillMatrices1* system with the new local variable *Ebi*.

The final simplified and complexity reduced equations are presented below,

$$QBI(i,j) = E_S(i,j) + \min_{4 \le d \le j-i-3} \{X(i,j,d)\} \tag{3.24}$$

where

$$X(i,j,d) = S_P(j-i-d-2) + Y(i,j,d) \tag{3.25}$$

and

$$Y(i,j,d) = \min \left\{ \begin{array}{l} Y(i+1,j-1,d) \\ \\ \min\limits_{i+1<i'<j-d-1} \left\{ \begin{array}{l} Asym(i'-i-1, j-i'-d-1) \\ +E_S(i', i'+d) \\ +Q'(i', i'+d) \end{array} \right\} \end{array} \right. \tag{3.26}$$

### 3.3.2 Fast-i-loops using AlphaZ

In order to circumvent the current limitations of AlphaZ, we need to slightly modify the derivation explained in the previous section. Particularly, the factoring out of the $E_S(i,j)$ term should be performed after decomposing the double reduction, in order to work around the current limitation of the `PermutationCaseReduce` transformation (see Appendix for details on the limitations of the transformation).

So far all the energy functions were implemented as external function calls in the Alphabets program. But, in order to split $E_{BI}$, we need to represent it as a local variable in the Alphabets system. We introduce a local variable *Ebi* in the *fillMatrices1* system as shown in figure 3.14 . The new Alphabets equation for *QBI* is shown in figure 3.15 where the functional call for *Ebi* (refer figure 3.12) is replaced

```
QBI[i,j] = case {|i>=j-6}:  INFINITY_VAL(0);
                {|i<j-6} :  MIN([d,ip], {|d<=j-i-3 && d>=3+1 && d>=j-i-2-MAXLOOP
&& ip>i && ip<j-d && ip<=N}:  (Ebi[i,j,ip,ip+d] + Qprime[ip,ip+d]));
           esac;
```

Figure 3.15: New Alphabets equations for *QBI*.

```
affine fillMatrices1 {N, MAXLOOP|N>4 && MAXLOOP > 0}
given
int valid_pairs {i,j | 1 <= i <= N && 2 <= j <= N};

returns
int Q {i,j | 1 <= i <= N && 2 <= j <= N};
int Qprime {i,j | 1 <= i <= N && 2 <= j <= N};
int QM {i,j | 1 <= i <= N && 2 <= j <= N};

using
int QBI {i,j | 1 <= i <= N && 2 <= j <= N};
int Ebi {i,j,ip,jp | 1 <= i < ip < jp < j <= N && ip-i+j-jp-2 <= MAXLOOP};
int Ebi_special {i,j,ip,jp |ip-i-1 == 0 && j-jp-1 == 1} ||
                {i,j,ip,jp |ip-i-1 == 1 && j-jp-1 == 0} ||
                {i,j,ip,jp |ip-i-1 == 0 && j-jp-1>1} ||
                {i,j,ip,jp |ip-i-1>1 && j-jp-1==0} ||
                {i,j,ip,jp |ip-i-1==1 && j-jp-1 == 1} ||
                {i,j,ip,jp |ip-i-1==1 && j-jp-1 == 2} ||
                {i,j,ip,jp |ip-i-1==2 && j-jp-1 == 1} ||
                {i,j,ip,jp |ip-i-1==2 && j-jp-1 == 2};
int Ebi_generic {i,j,ip,jp|ip-i>=2 && j-jp>=4 && jp-ip>=1} ||
                {i,j,ip,jp|ip-i>=4 && j-jp==3 && j-ip>=4} ||
                {i,j,ip,jp|ip-i==1 && j-jp==1 && j-i>=3} ||
                {i,j,ip,jp|ip-i>=4 && j-jp==2 && j-ip>=3};
```

Figure 3.16: Introducing *Ebi_special* and *Ebi_generic* in the *fillMatrices1* system.

by the local variable. But, as seen in equation (3.12), we need to handle special cases of internal loops separately. Hence, we introduce two new local variables, namely *Ebi_special* and *Ebi_generic*, in the *fillMatrices1* system as shown in the figure 3.16 and their definitions are given in figures 3.17 and 3.18 , respectively. Also, the Alphabets equation for *Ebi* is shown in figure 3.19 which complies with the equation (3.12). With these modifications to the existing Alphabets program, we are ready to apply transformations in AlphaZ according to the derivation in section 3.3.1 with slight changes to be in compliance with the current version of AlphaZ (refer section 2.1.3).

The first transformation is to substitute *Ebi* by its definition in *QBI*. This is done using the command `SubstituteByDef("fillMatrices1", "QBI", "Ebi");`. This

```
Ebi_special[i,j,ip,jp] =
          case {|ip-i-1 == 0 && j-jp-1 == 1} || {|ip-i-1 == 1 && j-jp-1 == 0} :
Ebi_Bulge1([i],[j],[ip],[jp]);
               {|ip-i-1 == 0 && j-jp-1>1}:  Ebi_Bulge([i],[j],[ip],[jp],[j-jp-1]);
               {|ip-i-1>1 && j-jp-1==0}:  Ebi_Bulge([i],[j],[ip],[jp],[ip-i-1]);
               {|ip-i-1==1 && j-jp-1 == 1}:  Ebi_iloop1x1([i],[j],[ip],[jp]);
               {|ip-i-1==1 && j-jp-1 == 2}:  Ebi_iloop1x2([i],[j],[ip],[jp]);
               {|ip-i-1==2 && j-jp-1 == 1}:  Ebi_iloop2x1([i],[j],[ip],[jp]);
               {|ip-i-1==2 && j-jp-1 == 2}:  Ebi_iloop2x2([i],[j],[ip],[jp]);
          esac;
```

Figure 3.17: Alphabets equation for *Ebi_ special*.

```
Ebi_A2[i,j,ip,jp] = Sp([ip-i-1+j-jp-1]) + Es([i],[j]) + Es([jp],[ip]) +
Asym([ip-i-1],[j-jp-1]);
```

Figure 3.18: Alphabets equation for *Ebi_ generic*.

is followed by the command `RemoveUnusedVariables();` to delete the definition and declaration of *Ebi* as it is no longer needed. The resulting equation for *QBI* is show in figure 3.20 .

The next transformation is to decompose the double reduction in *QBI*. This is done using the command `ReductionDecomposition("fillMatrices1", "QBI", "(i,j,d->i,j)", "(i,j,d,ip->i,j,d)");`. The resulting equation for *QBI* is shown in figure 3.21 .

We, then, apply the `NormalizeReductions(true, "fillMatrices1", "QBI", 1);` command to move the reductions into temporary local variables. The resulting equation for *QBI* is shown in the figure 3.22 . This is followed by two `RenameVariable` commands, `RenameVariable("fillMatrices1","AlphaZTempRed_1_QBI","QBI_W");` and `RenameVariable("fillMatrices1","AlphaZTempRed_2_QBI","QBI_X");` to provide sensible names to the temporary variables introduced by AlphaZ. The resulting equation for *QBI* is shown in figure 3.23 .

```
Ebi_A[i,j,ip,jp] = case
                      Ebi_A1[i,j,ip,jp];
                      Ebi_A2[i,j,ip,jp];
                   esac;
```

Figure 3.19: Alphabets equation for *Ebi*.

```
QBI[i,j] = case {|i-j>=-6} :  INFINITY_VAL(0);
                {|-i+j>=7} :  reduce(min, [k,l], case
                                                    {|-i+l>=1 &&
-i+j-k>=3 && k>=4 && j-k-l>=1 && MAXLOOP+i-j+k>=-2 && N-l>=0} :
(Ebi_special[i,j,l,k+l]+Qprime[l,k+l]);
                                                    {| -i+l>=1 &&
-i+j-k>=3 && k>=4 && j-k-l>=1 && MAXLOOP+i-j+k>=-2 && N-l>=0} :
(Ebi_generic[i,j,l,k+l]+Qprime[l,k+l]);
                                                esac);
           esac;
```

Figure 3.20: *QBI* after `SubstituteByDef` command.

```
QBI[i,j] = case {|i-j>=-6} :  INFINITY_VAL(0);
                {|-i+j>=7} :  reduce(min,[k],reduce(min,[l],case
                                                              {|-i+l>=1
&& -i+j-k>=3 && k>=4 && j-k-l>=1 && MAXLOOP+i-j+k>=-2 && N-l>=0} :
(Ebi_special[i,j,l,k+l]+Qprime[l,k+l]);
                                                              {|-i+l>=1
&& -i+j-k>=3 && k>=4 && j-k-l>=1 && MAXLOOP+i-j+k>=-2 && N-l>=0} :
(Ebi_generic[i,j,l,k+l]+Qprime[l,k+l]);
                                                                 esac));
           esac;
```

Figure 3.21: *QBI* after `ReductionDecomposition` command.

```
QBI[i,j] = case {|i-j>=-6} :  INFINITY_VAL(0);
                {|-i+j>=7} :  AlphaZTempRed_1_QBI[i,j];
           esac;

AlphaZTempRed_1_QBI[i,j] = reduce(min, [k], AlphaZTempRed_2_QBI[i,j,k] );

AlphaZTempRed_2_QBI[i,j,d] = reduce(min, [k], case
                                                    {|-i+k>=1 &&
-i+j-d>=3 && d>=4 && j-d-k>=1 && MAXLOOP+i-j+d>=-2 && N-k>=0} :
(Ebi_special[i,j,k,d+k]+Qprime[k,d+k]);
                                                    {|-i+k>=1
&& -i+j-d>=3 && d>=4 && j-d-k>=1 && MAXLOOP+i-j+d>=-2
&& N-k>=0} :  (Ebi_generic[i,j,k,d+k]+Qprime[k,d+k]);
esac);
```

Figure 3.22: *QBI* after `NormalizeReductions` command

```
QBI[i,j] = case {|i-j>=-6} :  INFINITY_VAL(0);
                {|-i+j>=7} :  QBI_W[i,j];
           esac;

QBI_W[i,j] = reduce(min, [k], QBI_X[i,j,k]);

QBI_X[i,j,d] = reduce(min, [k], case
                                   {|-i+k>=1 && -i+j-d>=3 && d>=4 && j-d-k>=1 &&
MAXLOOP+i-j+d>=-2 && N-k>=0} :  (Ebi_special[i,j,k,d+k]+Qprime[k,d+k]);
                                   {|-i+k>=1 && -i+j-d>=3 && d>=4 && j-d-k>=1 &&
MAXLOOP+i-j+d>=-2 && N-k>=0} :  (Ebi_generic[i,j,k,d+k]+Qprime[k,d+k]);
                                esac);
```

Figure 3.23: $QBI$ after `RenameVariable` command.

```
QBI_X[i,j,d] = case
                  {|i-j+d==-3 && -i+j>=7 && i>=1 && N-j>=0} ||
                  {|-i+j-d>=4 && d>=4 && i-j+d>=-5 && i>=1 && MAXLOOP+i-j+d>=-2
&& N-j>=0} :  reduce(min, [k], (Ebi_special[i,j,k,d+k]+Qprime[k,d+k]));
                  {|-i+j-d>=6 && d>=4 && i>=1 && MAXLOOP+i-j+d>=-2 && N-j>=0} :
(reduce(min, [k], (Ebi_special[i,j,k,d+k]+Qprime[k,d+k])) min reduce(min, [k],
(Ebi_generic[i,j,k,d+k])));
               esac;
```

Figure 3.24: $QBI\_X$ after `PermutationCaseReduce` command

Then, `PermutationCaseReduce("fillMatrices1", "QBI_X");` command is applied to pull the case statements out from the body of the reduction expression in $QBI\_X$. The resulting equation for $QBI\_X$ is shown in the figure 3.24 .

To make the Alphabets code more legible, we then apply the four commands show below in order,

1. `NormalizeReductions(true, "fillMatrices1", "QBI_X");`
2. `RenameVariable("fillMatrices1","AlphaZTempRed_1_QBI_X","QBI_X1");`
3. `RenameVariable("fillMatrices1","AlphaZTempRed_2_QBI_X","QBI_X2");`
4. `RenameVariable("fillMatrices1","AlphaZTempRed_3_QBI_X","X");`

The result of these four transformations is shown in figure 3.25 .

We now shift our focus to the equation for $X$ as it has $Ebi\_generic$ term to expose reuse. We substitute the definition of $Ebi\_generic$ in $X$ by using the command `SubstituteByDef("fillMatrices1", "X", "Ebi_generic");`. This is followed by

```
QBI_X[i,j,d] = case
                  {|i-j+d==-3 && -i+j>=7 && i>=1 && N-j>=0} ||
                  {|-i+j-d>=4 && d>=4 && i-j+d>=-5 && i>=1 && MAXLOOP+i-j+d>=-2
&& N-j>=0} :  QBI_X1[i,j,d];
                  {|-i+j-d>=6 && d>=4 && i>=1 && MAXLOOP+i-j+d>=-2 && N-j>=0} :
(QBI_X2[i,j,d] min X[i,j,d]);
               esac;

QBI_X1[i,j,d] = reduce(min, [k], (Ebi_special[i,j,k,d+k]+Qprime[k,d+k]));

QBI_X2[i,j,d] = reduce(min, [k], (Ebi_special[i,j,k,d+k]+Qprime[k,d+k]));

X[i,j,d] = reduce(min, [k], (Ebi_generic[i,j,k,d+k]+Qprime[k,d+k]));
```

Figure 3.25: *QBI_X* after `NormalizeReductions` and `RenameVariables` commands.

```
X[i,j,d] = reduce(min, [k], {|-i+k>=2 && d>=1 && j-d-k>=4} ||
                            {|j-d-k==3 && -i+j-d>=7 && d>=1} ||
                            {|i-j+d==-2 && i-k==-1 && -i+j>=3} ||
                            {|j-d-k==2 && -i+j-d>=6 && d>=1} :  (Sp([-i+j-d-2]) +
Es([i],[j]) + Es([d+k],[k]) + Asym([-i+k-1],[j-d-k-1]) + Qprime[k,d+k]));
```

Figure 3.26: $X$ after `SubstituteByDef` command.

the command `RemoveUnusedVariables();` to get rid of the declaration and definition of *Ebi_generic* as it is not used anymore. The resulting equation for $X$ is shown in the figure 3.26 .

Now, we can distribute out *Es([i],[j])* and *Sp([-i+j-d-2])* from the reduction expression in $X$ as they do not depend on the reduction index $k$. This can be ideally done using the transformation `FactorOutFromReduction`. But, due to the current limitations, this transformation does not handle index value expressions. So, we apply this transformation by hand after saving the intermediate Alphabets program using `Save("temp.ab");` command. The modified equation for $X$ is shown in figure 3.27 . This modified `temp.ab` file is then loaded back in to AlphaZ using the

```
X[i,j,d] = Ebi_stacking([i],[j]) + Ebi_sizePenalty([-i+j-d-2])
         + reduce(min, [k] ,{|-i+k>=2 && d>=1 && j-d-k>=4} ||
                            {|j-d-k==3 && -i+j-d>=7 && d>=1} ||
                            {|i-j+d==-2 && i-k==-1 && -i+j>=3} ||
                            {|j-d-k==2 && -i+j-d>=6 && d>=1}:
(Ebi_stacking([d+k],[k]) + Ebi_asymmetry([-i+k-1],[j-d-k-1]) + Qprime[k,d+k]));
```

Figure 3.27: $X$ after `FactorOutFromReduction` transformation.

```
X[i,j,d] = ((Ebi_stacking([i],[j]) + Ebi_sizePenalty([-i+j-d-2])) + Y[i,j,d]);

Y[i,j,d] = reduce(min, [k], {|-i+k>=2 && d>=1 && j-d-k>=4} ||
                           {|j-d-k==3 && -i+j-d>=7 && d>=1} ||
                           {|i-j+d==-2 && i-k==-1 && -i+j>=3} ||
                           {|j-d-k==2 && -i+j-d>=6 && d>=1} :
(Ebi_stacking([d+k],[k]) + Ebi_asymmetry([-i+k-1],[j-d-k-1]) + Qprime[k,d+k])
);
```

Figure 3.28: $X$ after `NormalizeReductions` and `RenameVariable` transformations.

```
Y[i,j,d] = case
              {|-i+j-d>=6 && d>=4 && i-j+d>=-7 && i>=1 && MAXLOOP+i-j+d>=-2
&& N-j>=0}:  reduce(min, [k], {|i-k==-2 && -i+j-d>=7 && d>=4 && i>=1 &&
MAXLOOP+i-j+d>=-2 && N-j>=0} || {|i-j+d==-6 && i-k==-2 && -i+j>=10 && i>=1
&& MAXLOOP>=4 && N-j>=0} || {|j-d-k==4 && -i+j-d>=7 && d>=4 && i-j+d>=-8
&& i>=1 && MAXLOOP+i-j+d>=-2 && N-j>=0} || {|i-j+d==-7 && i-k==-4 &&
-i+j>=11 && i>=1 && MAXLOOP>=5 && N-j>=0} || {|j-d-k==2 && -i+j-d>=6 &&
d>=4 && i>=1 && MAXLOOP+i-j+d>=-2 && N-j>=0} :  ((Ebi_stacking([d+k],[k]) +
Ebi_asymmetry([-i+k-1],[j-d-k-1])) + Qprime[k,d+k]));
              {|-i+j-d>=8 && d>=4 && i>=1 && MAXLOOP+i-j+d>=-2 && N-j>=0} :
(reduce(min, [k], {| i-k==-2 && -i+j-d>=7 && d>=4 && i>=1 && MAXLOOP+i-j+d>=-2
&& N-j>=0} || {|i-j+d==-6 && i-k==-2 && -i+j>=10 && i>=1 && MAXLOOP>=4 && N-j>=0}
|| {|j-d-k==4 && -i+j-d>=7 && d>=4 && i-j+d>=-8 && i>=1 && MAXLOOP+i-j+d>=-2 &&
N-j>=0} || {|i-j+d==-7 && i-k==-4 && -i+j>=11 && i>=1 && MAXLOOP>=5 && N-j>=0}
|| {|j-d-k==2 && -i+j-d>=6 && d>=4 && i>=1 && MAXLOOP+i-j+d>=-2 && N-j>=0} :
((Ebi_stacking([d+k],[k]) + Ebi_asymmetry([-i+k-1],[j-d-k-1])) + Qprime[k,d+k]) )
min Y[i+1,j-1,d]);
           esac;
```

Figure 3.29: $Y$ after `SimplifyingReductions` transformations.

command `ReadProgram(“temp.ab”);`.

In order to isolate the reduction expression from $X$, we now apply `NormalizeReductions(true, "fillMatrices1", "X");` command followed by `RenameVariable("fillMatrices1","AlphaZTempRed_1_X","Y");` command resulting in the Alphabets equation shown in figure 3.28 .

The final step is to apply the transformation to expose the scan in the equation for $Y$. This is done by using the `SimplifyingReductions("fillMatrices1", "Y", "1,-1,0,0");` command. The last parameter in the SimplifyingReductions transformation specifies the direction of sharing along each of the dimensions. The resulting equation for $Y$ is shown in figure 3.29 .

To make the Alphabets code more readable, we apply the following commands in the order shown below,

```
Y[i,j,d] = case {|-i+j-d>=6 && d>=4 && i-j+d>=-7 && i>=1 && MAXLOOP+i-j+d>=-2 &&
N-j>=0}:  Z1[i,j,d];
               {|-i+j-d>=8 && d>=4 && i>=1 && MAXLOOP+i-j+d>=-2 && N-j>=0}:
(Z2[i,j,d] min Y[i+1,j-1,d]);
          esac;

Z1[i,j,d] = reduce(min, [k], {|i-k==-2 && -i+j-d>=7 && d>=4 && i>=1 &&
MAXLOOP+i-j+d>=-2 && N-j>=0} || {|i-j+d==-6 && i-k==-2 && -i+j>=10 && i>=1
&& MAXLOOP>=4 && N-j>=0} || {|j-d-k==4 && -i+j-d>=7 && d>=4 && i-j+d>=-8
&& i>=1 && MAXLOOP+i-j+d>=-2 && N-j>=0} || {|i-j+d==-7 && i-k==-4 &&
-i+j>=11 && i>=1 && MAXLOOP>=5 && N-j>=0} || {|j-d-k==2 && -i+j-d>=6 &&
d>=4 && i>=1 && MAXLOOP+i-j+d>=-2 && N-j>=0} :  ((Ebi_stacking([d+k],[k]) +
Ebi_asymmetry([-i+k-1],[j-d-k-1])) + Qprime[k,d+k]));

Z2[i,j,d] = reduce(min, [k], {|i-k==-2 && -i+j-d>=7 && d>=4 && i>=1 &&
MAXLOOP+i-j+d>=-2 && N-j>=0} || {|i-j+d==-6 && i-k==-2 && -i+j>=10 && i>=1
&& MAXLOOP>=4 && N-j>=0} || {|j-d-k==4 && -i+j-d>=7 && d>=4 && i-j+d>=-8
&& i>=1 && MAXLOOP+i-j+d>=-2 && N-j>=0} || {|i-j+d==-7 && i-k==-4 &&
-i+j>=11 && i>=1 && MAXLOOP>=5 && N-j>=0} || {|j-d-k==2 && -i+j-d>=6 &&
d>=4 && i>=1 && MAXLOOP+i-j+d>=-2 && N-j>=0} :  ((Ebi_stacking([d+k],[k]) +
Ebi_asymmetry([-i+k-1],[j-d-k-1])) + Qprime[k,d+k]));
```

Figure 3.30: $Y$ after `NormalizeReductions` and `RenameVariable` transformations.

1. `NormalizeReductions(true, "fillMatrices1", "Y");`

2. `RenameVariable("fillMatrices1", "AlphaZTempRed_1_Y", "Z1");`

3. `RenameVariable("fillMatrices1", "AlphaZTempRed_2_Y", "Z2");`

The resulting equation for $Y$ is shown in the figure 3.30 . Observe that the domains in the reduction expressions in *Z1* and *Z2* have equalities between dimensions. This shows that the computational complexity of the equation has been reduced.

The complete script which applies the transformations mentioned above is shown in figure (3.31) . In the script, one might notice the Normalize(); command being called a couple of times. This helps in simplifying the AlphaZ AST and eases the job of the code generators. Also, notice the WriteC(); command called at the end of the script. This command generates Demand-driven C program for the Alphabets program.

```
ConnectServer("alphabets.corequations.com");
ReadProgram("unafold.ab");
SubstituteByDef("fillMatrices1", "QBI", "Ebi");
RemoveUnusedVariables();
Normalize();
ReductionDecomposition("fillMatrices1", "QBI", "(i,j,d->i,j)",
"(i,j,d,ip->i,j,d)");
Normalize();
NormalizeReductions(true, ‘‘fillMatrices1’’, ‘‘QBI’’, 1);
RenameVariable(‘‘fillMatrices1’’, ‘‘AlphaZTempRed_1_QBI’’, ‘‘QBI_W’’);
RenameVariable(‘‘fillMatrices1’’, ‘‘AlphaZTempRed_2_QBI’’, ‘‘QBI_X’’);
PermutationCaseReduce("fillMatrices1", ‘‘QBI_X’’);
Normalize();
NormalizeReductions(true, ‘‘fillMatrices1’’, ‘‘QBI_X’’);
RenameVariable(‘‘fillMatrices1’’, ‘‘AlphaZTempRed_1_QBI_X’’, ‘‘QBI_X1’’);
RenameVariable(‘‘fillMatrices1’’, ‘‘AlphaZTempRed_2_QBI_X’’, ‘‘QBI_X2’’);
RenameVariable(‘‘fillMatrices1’’, ‘‘AlphaZTempRed_3_QBI_X’’, ‘‘X’’);
Normalize();
SubstituteByDef("fillMatrices1", "X", "Ebi_generic");
RemoveUnusedVariables();
Normalize();
Save("temp.ab");
#Edit this file and do a FactorOutFromReduction by hand.
ReadProgram("temp.ab");
Normalize();
NormalizeReductions(true, ‘‘fillMatrices1’’, ‘‘X’’);
RenameVariable(‘‘fillMatrices1’’, ‘‘AlphaZTempRed_1_X’’, ‘‘Y’’);
Normalize();
SimplifyingReductions("fillMatrices1", "Y", "1,-1,0,0");
Normalize();
NormalizeReductions(true, "fillMatrices1", "Y");
RenameVariable("fillMatrices1", "AlphaZTempRed_1_Y", "Z1");
RenameVariable("fillMatrices1", "AlphaZTempRed_2_Y", "Z2");
Normalize();
WriteC();
```

Figure 3.31: Script for deriving Fast-i-loops algorithm using AlphaZ.

## 3.4 Code generation

After the transformations described in section 3.3.2 have been applied, we use the `WriteC();` command in AlphaZ to generate a demand-driven C program for the RNA secondary structure prediction algorithm. Since AlphaZ is a research tool which is still in development, we are limited to using the demand-driven code generator.

The demand-driven code generator has the following limitations.

1. It uses its own default memory map. Hence, we cannot explore optimizations based on memory usage.

2. As the name implies, a demand-driven code generator uses the demand-driven schedule. So, optimizations based on scheduling computations is ruled out.

3. Since a demand-driven schedule is inherently sequential, processor allocation does not come in to the picture.

4. Tiling of the computation space cannot be explored because of the demand-driven schedule.

Although, the demand-driven code generator, `WriteC`, is the most functional code generator in AlphaZ, it has some issues. The memory access function generated by WriteC for the input and output variables is $i * (N + 1) + j$ whereas the UNAFold C program uses $(i - 1) * (N - 1) + (j - 1)$ instead. Currently, this issue is resolved by editing the generated C program manually. Another issue involves the macros generated by `WriteC`. The UNAFold C program has a macro $MAX(a, b)$ which also exists in the C program generated by WriteC. This gives rise to compilation errors as there are multiple macro definitions with the same identifier. This issue is currently resolved by inspecting the `WriteC` generated code manually and editing out the duplicated macro if it is not used. If it is used, then the duplicate macro is renamed and all its occurrences in the `WriteC` generated program are replaced by the new name for the macro.

# Chapter 4

# Results

The focus of this thesis is not comparing performances against various implementations of RNA secondary structure prediction algorithms that exist. On the contrary, our objective is to compare the performance we have achieved between the original and the fast-i-loops versions of the algorithm implemented using AlphaZ. It is to show that we can benefit from equational programming by using AlphaZ to do the transformations needed to arrive at the fast-i-loops algorithm. Before we get to the comparison between original and fast-i-loops algorithm, we will see where the AlphaZ generated demand-driven C program (WriteC program) stands against the original UNAFold C program. Table 4.1 shows this comparison. We see that WriteC program is slower than the original UNAFold C program. This is because WriteC program is demand-driven which essentially means that when a computational statement, let's say, $S1$ needs the value produced by another computational statement $S2$, that is, $S1$ demands $S2$, it is then when $S2$ is computed and $S1$ waits for the result. This is inherently implemented by recursive calls to functions computing these statements. And, the UNAFold C program is implemented using a sequential schedule with some

| **Sequence length** | **UNAFold C program Time (ms)** | **AlphaZ WriteC program Time (ms)** |
|---|---|---|
| 100 | 3 | 7 |
| 200 | 18 | 47 |
| 300 | 50 | 137 |
| 400 | 107 | 313 |

Table 4.1: Execution time of UNAFold C program vs WriteC generated program.

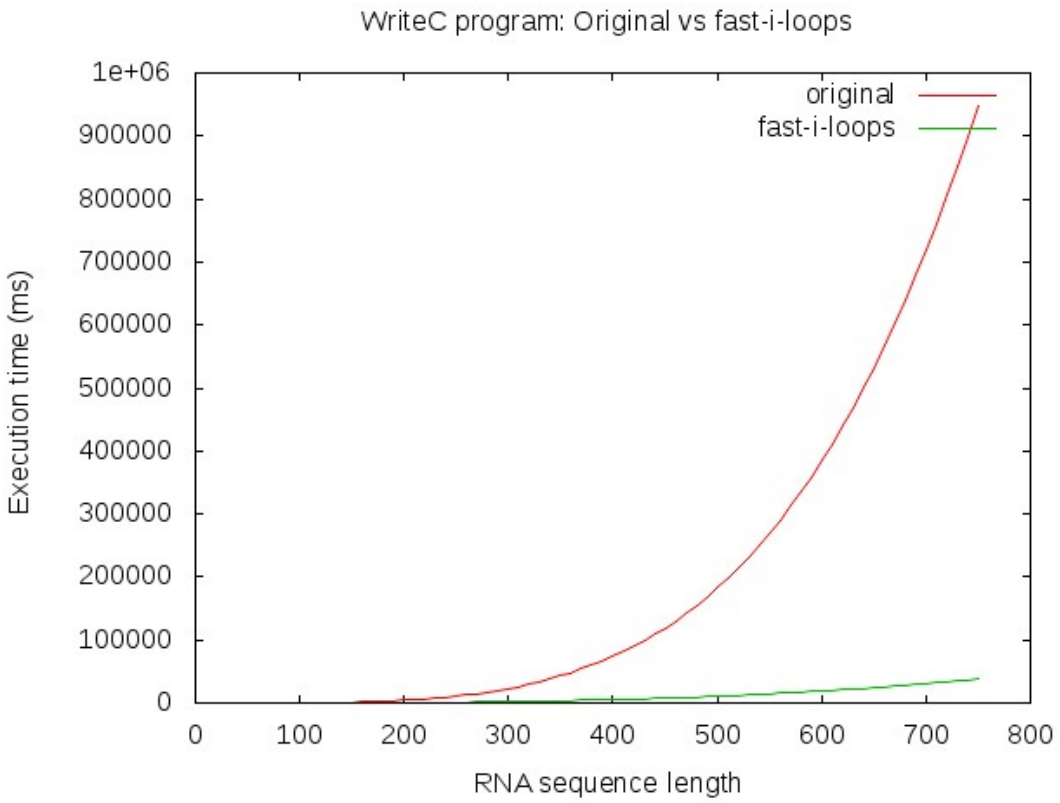


Figure 4.1: Execution time comparison between original and fast-i-loops version of RNA secondary structure prediction algorithm generated using AlphaZ on a 3.0 GHz processor with 16 GB of memory.

optimizations to improve cache behavior. However, as more efficient code generators are implemented in the AlphaZ framework, we can use them to generate more efficient programs using the same Alphabets code.

The graph in figure 4.1 shows a comparison of execution times between the WriteC generated programs for the original and fast-i-loops version of the RNA secondary structure prediction algorithm on a 3.0GHz processor with 16 GB of memory. The same graph is shown in figure 4.2 on a logarithmic scale. The relative speed up between the two versions is displayed in the graph shown in figure 4.3 . From these graphs, we can see that the WriteC generated program for the fast-i-loops version is significantly faster than the WriteC generated program for the original $O(N^4)$ algorithm. Note that, in order to ensure that the computations in the program remain dense, base pairing was forced for all the combinations of A, G, C, U bases, as opposed

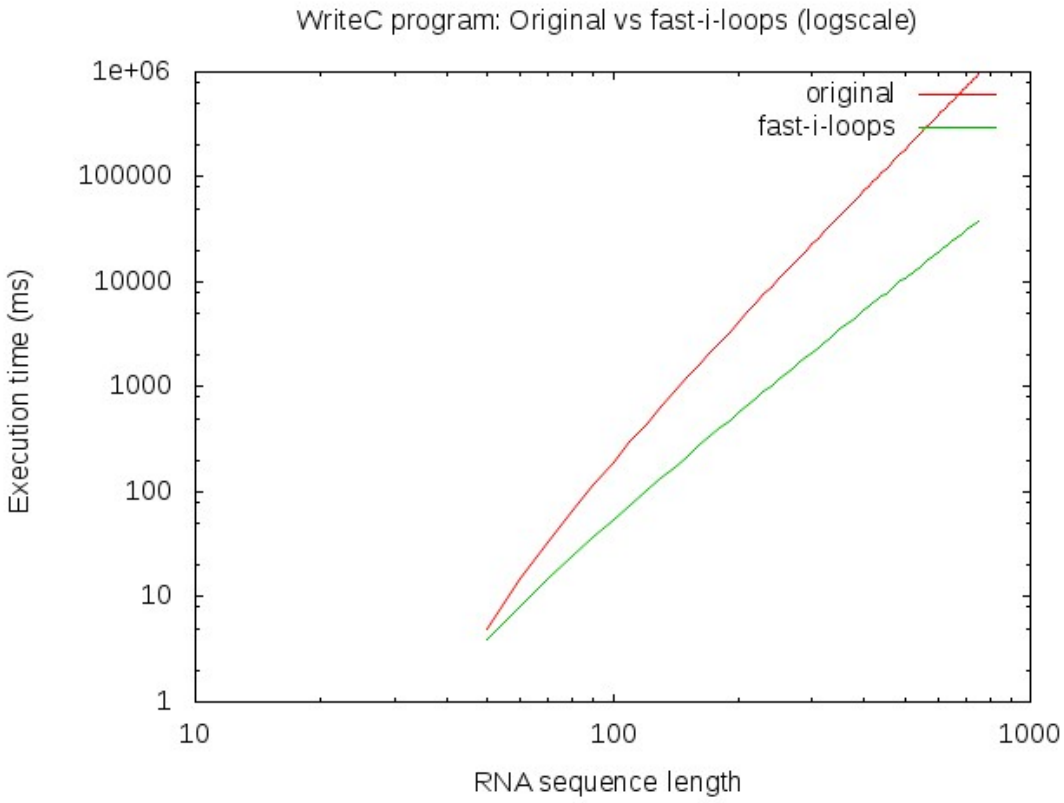


Figure 4.2: Execution time comparison using logarithmic scale between original and fast-i-loops version of RNA secondary structure prediction algorithm generated using AlphaZ on a 3.0 GHz processor with 16 GB of memory.

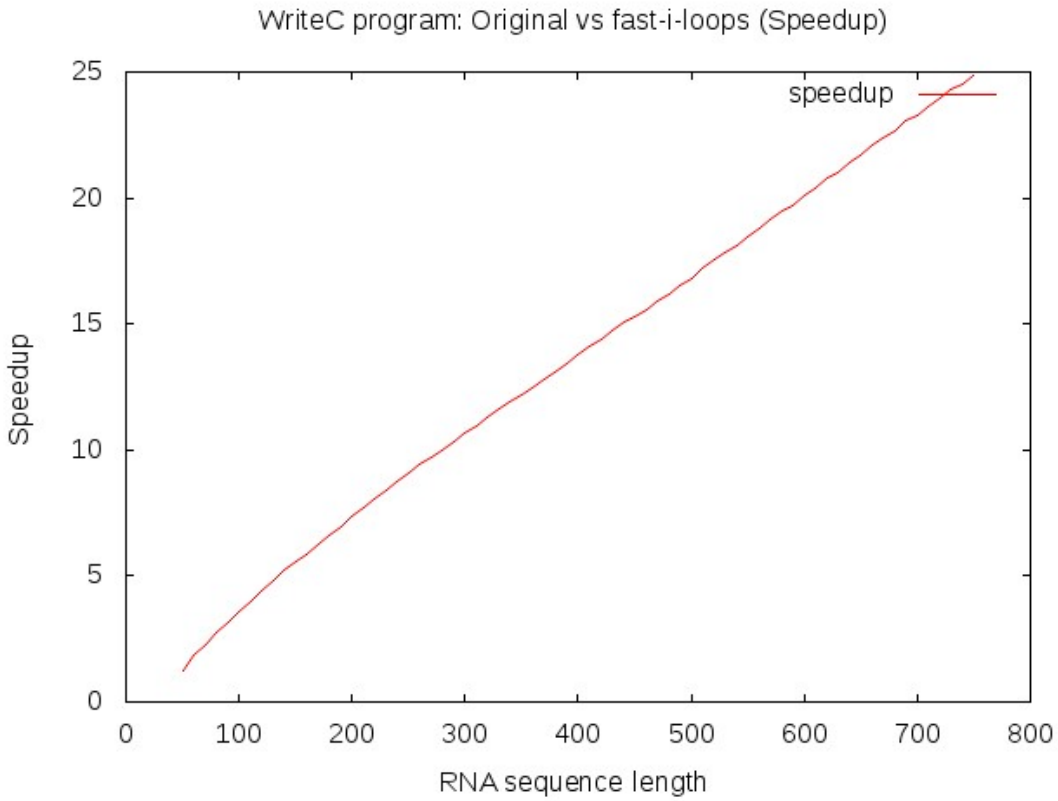


Figure 4.3: Speed Up of the fast-i-loops version against the original version of the RNA secondary structure prediction algorithm generated using AlphaZ on a 3.0 GHz processor with 16 GB of memory.

to following the Watson-Crick base pairing rules which result in sparse computations. This was done only to collect performance data for the WriteC generated programs.

# Chapter 5

# Conclusion And Future Work

In this thesis, we have shown that programming using equations is beneficial and the AlphaZ framework helps us to realize this. It is relatively easy to optimize algorithms when we think in terms of equations. This has been shown by deducing the fast-i-loops algorithm for evaluation of internal loops in the RNA secondary structure prediction algorithm. It would be a very ardent task to achieve this by writing programs in imperative languages manually. Also, implementing complex algorithms manually on new complex and parallel processor architectures would be quite challenging and time consuming. But, with the help of code generators in AlphaZ, this could be a relatively simple and time saving task.

Our case study of implementing RNA secondary structure prediction algorithm using AlphaZ proves that the AlphaZ framework is ready to tackle real-life applications and leave the toy problems like matrix-matrix multiply, LU decomposition and optimal string parenthesization behind. It also acts as a base for future optimizations of the algorithm using AlphaZ. This thesis also serves as a limited guide for new users to get started with AlphaZ and explore the advantages associated with the idea of programming using equations.

## 5.1 Future Work

As mentioned earlier, AlphaZ is a research tool which is still under development. Lot of optimizing transformations and code generators are still under development and are in their early stages. Code generators like Sequential C code generator,

Tiled C code generator and OpenMP code generator are in development. As these code generators develop, we will have more options to optimize and implement our algorithms. For example, we can explore different memory mappings in each of these code generators to reduce the memory space requirements of the RNA secondary structure prediction algorithm. We can also explore different schedules and processor allocations including the popular wavefront schedule for the algorithm using the OpenMP code generator. The availability of a Tiled C code generator allows us to optimize the algorithm by exploring different tile sizes and tile dimensions to improve data locality depending on the target architecture for implementation.

# REFERENCES


[1] Alphabets Grammar. http://www.cs.colostate.edu/ cs560/Lectures/GrammerAlphabets.pdf.

[2] max planck institut informatik. http://www.mpi-inf.mpg.de/.

[3] OpenMP: OpenMP API Specification for parallel programming. http://openmp.org/wp/.

[4] T. Akutsu. Recent advances in rna secondary structure prediction with pseudoknots. *Current Bioinformatics*, 1:115–129, MAY 2006.

[5] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.

[6] D. Barthou, J. F. Collard, and P. Feautrier. Fuzzy Array Dataflow Analysis. *Journal of Parallel and Distributed Computing*, 40(2), 1997.

[7] M. Belaoucha, D. Barthou, A. Eliche, and S. Touati. FADAlib: an open source C++library for fuzzy array dataflow analysis. *International Conference on Computational Science, ICCS*, 2010.

[8] R. E. Bellman. Dynamic Programming. *Princeton University Press, Princeton, NJ.*, 1957.

[9] X. Chen, S. He, D. Bu, F. Zhang, Z. Wang, R. Chen, and W. Gao. Flexstem: Improving predictions of rna secondary stuctures with pseudoknots by reducing the search space. *Bioinformatics*, 24:1994–2001, SEP 2008.

[10] R. M. Dirks, J. S. Bois, J. M. Schaeffer, E. Winfree, and N. A. Pierce. Thermodynamic analysis of interacting nucleic acid strands. *SIAM Rev*, 49:65–88, 2007.

[11] R. M. Dirks and N. A. Pierce. A partition function algorithm for nucleic acid secondary structure including pseudoknots. *J Comput Chem*, 24:1664–1677, 2003.

[12] R. M. Dirks and N. A. Pierce. An algorithm for computing nucleic acid base-pairing probabilities including pseudoknots. *J Comput Chem*, 25:1295–1304, 2004.

[13] F. Dupont de Dincehcin. *SystËmes structurÈs d'Èquations rÈcurrentes : mide en œuvre dans le langage Alpha et applications.* PhD thesis, UniversitÈ de Rennes, IRISA, Rennes, janvier 1997.

[14] G. Gupta. COREquations: Equational Programming., 2008. www.corequations.com.

[15] G. Gupta and S. Rajopadhye. Generation of the fast-i-loops algorithm.

[16] G. Gupta and S. Rajopadhye. Simplifying reductions. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 30–41, New York, NY, USA, 2006. ACM.

[17] I. L. Hofacker, W. Fontana, P. F. Stadler, L. S. Bonhoeffer, M. Tacker, and P. Schuster. Fast Folding and Comparison of RNA Secondary Structures. *Monatshefte fur Chemie*, 125:167–188, FEB 1994.

[18] A. C. Jacob, J. D. Buhler, and R. D. Chamberlain. Accelerating Nussinov RNA secondary structure prediction with systolic arrays on FPGAs. *IEEE International Conference on Application-specific Systems, Architectures and Processors*, 2008.

[19] A. C. Jacob, J. D. Buhler, and R. D. Chamberlain. Rapid RNA Folding: Analysis and Acceleration of the Zuker Recurrence. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2010.

[20] R. B. Lyngso, M. Zuker, and C. N. S. Pedersen. Fast evaluation of internal loops in rna secondary structure prediction. *Bioinformatics*, 15:440–445, JUN 1999.

[21] N. R. Markham. *Algorithms and Software for Nucleic Acid Sequences.* PhD thesis, Rensselaer Polytechnic Institute Library, May 2006.

[22] N. R. Markham and M. Zuker. UNAFold: software for nucleic acid folding and hybriziation. *Bioinformatics*, II. Structure, Functions and Applications(453):3–31, 2008.

[23] D. H. Mathews, J. Sabina, M. Zuker, and D. H. Turner. Expanded sequence dependence of thermodynamic parameters improves prediction of RNA secondary structure. *J Mol Biol*, 288(5):911–40, 21 May 1999.

[24] A. Mathuriya, D. A. Bader, C. E. Heitsch, and S. C. Harvey. GTfold: A Scalable Multicore Code for RNA Secondary Structure Prediction. *24th Annual ACM Symposium on Applied Computing (SAC)*, March 2009.

[25] C. Mauras. *Alpha, un langage equationnnel pour la conception et la programmation d'architectures paralleles synchrones.* PhD thesis, Universite de Rennes I, Rennes, France, December 1989.

[26] J. S. McCaskill. The equilibrium partition function and base pair binding probabilities for RNA secondary structure. *Biopolymers*, 29:1105–1119, 1990.

[27] MELANGE. AlphaZ. http://www.cs.colostate.edu/AlphaZ/.

[28] P. Niemeyer. BeanShell. http://www.beanshell.org/.

[29] R. Nussinov, G. Piecznik, J. R. Grigg, and D. J. Klietman. Algorithms for loop matchings. *SIAM Journal on Applied Mathematics*, 1978.

[30] NVIDIA. CUDA (Compute Unified Device Architecture). http://www.nvidia.com/object/cuda_home.html#.

[31] S. V. Rajopadhye. Synthesizing systolic arrays with control signals from recurrence equations. *Distributed Computing*, 3(2):88–105, 1989.

[32] J. Reeder and R. Giegerich. Design, implementation and evaluation of a practical pseudoknot folding algorithm based on thermodynamics. *BMC Bioinformatics*, 5(104), AUG 2004.

[33] J. Reeder and R. Giegerich. Design, implementation and evaluation of a practical pseudoknot folding algorithm based on thermodynamics. *BMC Bioinformatics*, 5(1):104, 2004.

[34] J. Reeder, P. Steffen, and R. Giegerich. pknotsRG: RNA pseudoknot folding including near-optimal structures and sliding windows. *Nucleic Acids Research*, 35:W320–W324, JUL 2007.

[35] E. Rivas and S. R. Eddy. A dynamic programming algorithm for RNA structure prediction including pseudoknots. *J Mol Biol*, 285(5):2053–68, 5 February 1999.

[36] G. Rizk and D. Lavenier. *GPU accelerated RNA folding algorithm.* Addison Wesley (in press). GPU Computing Gems 4.

[37] G. Rizk and D. Lavenier. GPU accelerated RNA folding algorithm. *Internation Conference on Computational Science*, pages 1004–1013, 2009.

[38] H. L. Verge. Reduction operators in alpha. In *PARLE '92: Proceedings of the 4th International PARLE Conference on Parallel Architectures and Languages Europe*, pages 397–411, London, UK, 1992. Springer-Verlag.

[39] J. N. Zadeh, C. D. Steenberg, J. S. Bois, B. R. Wolfe, M. B. Pierce, A. R. Khan, R. M. Dirks, and N. A. Pierce. NUPACK: analysis and design of nucleic systems. *J Comput Chem.*

[40] M. Zuker. Computer prediction of RNA structure. *Methods Enzymol.*, 180:262–88, 1989.

[41] M. Zuker. Mfold web server for nucleic acid folding and hybridization prediction. *Nucleic Acids Res.*, 31(13):3406–15, 1 July 2003.

[42] M. Zuker and D. Sankoff. RNA Secondary Structures and their Prediction. *Bull. Mathematical Biology*, 46:591–621, 1984.

[43] M. Zuker and P. Stiegler. Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. *Nucleic Acids Res*, 9(1):133–48, 10 January 1981.

# Appendix

## I. List of AlphaZ commands

- `ConnectServer(String serverName)`
    - This command connects the AlphaZ client to the COREquations server which is available at "*alphabets.corequations.com*".
- `ReadProgram(String inputFilePath)`
    - This command reads the Alphabets program from the file specified by the path `inputFilePath` and synchronizes it with the COREquations server.
- `Show() and AShow()`
    - These commands print the current program on to the BeanShell console in two different formats, pure and array respectively.
- `Save(String outputFilePath)` and `ASave(String outputFilePath)`
    - These commands save the current program to a file specified by the path `outputFilePath` in pure and array formats respectively.
- `RenameVariable(String sysName, String oldName, String newName)`
    - This command renames a variable from `oldName` to `newName` in the Alphabets system `sysName`.
- `Normalize()`

- This command normalizes the current program by simplifying nested expressions and domains.

- `FactorOutFromReduction(String sysName, String varName, int occr, int optr, int oprd)`

  - This command factors out the operand `oprd` of the operator `optr` from the reduction expression at occurrence `occr` in the variable `varName` of the system `sysName`. Currently, this command does not handle index expressions correctly and it also does not check for the legality of the resulting expression.

- `NormalizeReductions(boolean unique, String sysName, String varName, int occr)`

  - This command replaces reduction expression at occurrence `occr` in the variable `varName` of the system `sysName` with reference to a automatically generated local variable which contains the replaced reduction expression as its definition. All the arguments for this command are optional. If none of the arguments are specified, the command applies to all the reduction expressions in the current program. If the argument `unique` is set to true, the transformation replaces every reduction expression by a unique local variable. Otherwise, identical reduction expressions are replaced by the same local variable.

- `PermutationCaseReduce(String sysName, String varName)`

  - This commands moves out a case expression from within a reduction expression in the variable `varName` of the system `sysName`. `varName` is an optional argument. If it is not specified, the transformation is applied to all the reduction expressions in the system `sysName`. Currently, this command does not handle a case expression having more than two cases. It

also does not handle reduction expression with projections to more than one dimension.

- `ReductionDecomposition(String sysName, String varName, int occr, String func1, String func2)`
  - This command replaces a reduction expression at occurrence `occr` in the variable `varName` of the system `sysName` with a nested reduction expression having projection functions as `func1` and `func2`. The function composition of `func1` and `func2` should be equal to the projection function of the original reduction expression. The `occr` argument is optional which defaults to 0 if not specified.
- `RemoveUnusedVariables(String sysName, boolean removeInputs)`
  - This command removes all unused variables, that is, the variables which do not contribute to any equations that define output variables or other variables that are used by the output. Both the arguments to this command are optional. If `sysName` is not specified, the command is applied to all the systems in the current program. If `removeInputs` is not set to false, then unused input variables will also be removed by default.
- `SimplifyingReductions(String sysName, String varName, String reuse, int occr)`
  - This command transforms a reduction expression into multiple reduction expressions which enable the re-use of partial results along the direction specified by the re-use vector `reuse` when computing the result at a different location in the domain of the reduction expression. This transformation is applied to the reduction expression at occurrence `occr` in the variable `varName` of the system `sysName`. `occr` is an optional argument with the default value being 0. This command implements the simplifying reductions technique proposed in [16].

- `SubstituteByDef(String sysName, String trgVar, String srcVar, int numOccr)`
    - This command substitutes the references to the source variable `srcVar`, by its definition, in the equation for the target variable `trgVar` of the system sysName. `numOccr` is an optional argument. By default, all the occurrences of `srcVar` are substituted unless limited to `numOccr`.
- `WriteC(String sysName, String dirPath)`
    - This command generates a demand driven C function for the system `sysName` and writes the generated code to the directory specified by `dirPath`. Both its arguments are optional. If `sysName` is not specified, then the command generates a C function corresponding to each Alphabets system in the current program. If `dirPath` is not specified, the generated code is written to the current working directory.

# II. *fillMatrices1()* function from `hybrid-ss-min.c`

```
void fillMatrices1()
{
  int i, j, k;

  for (j = 2; j <= g_len; ++j)
    for (i = j - TURN - 1; i >= 1; --i)
    {
      ENERGY au;
      au = auPenalty(i, j);

      if (isFinite(Qprime(i, j)))
      {
         Qprime(i, j) = min4(Eh(i, j),
                             Es(i, j) + Qprime(i + 1, j - 1),
                             QBI(i, j),
                             g_multi[0] + g_multi[2] + au + QM(i + 1, j - 1));
if (!g_nodangle)
         {
           if (j > 2)
             Qprime(i, j) = min2(Qprime(i, j), g_multi[0] + g_multi[1] +
g_multi[2] + au + Ed5(i, j) + QM(i + 1, j - 2));
```

```
          if (i < g_len - 1)
            Qprime(i, j) = min2(Qprime(i, j), g_multi[0] + g_multi[1] +
g_multi[2] + au + Ed3(i, j) + QM(i + 2, j - 1));
          if (j > 2 && i < g_len - 1)
            Qprime(i, j) = min2(Qprime(i, j), g_multi[0] + 2.0 * g_multi[1] +
g_multi[2] + au + Etstackm(i, j) + QM(i + 2, j - 2));
        }
      }

      QM(i, j) = INFINITY;
      for (k = i + TURN + 1; k <= j - TURN - 2; ++k)
        QM(i, j) = min2(QM(i, j), Q(i, k) + Q(k + 1, j));

      if (g_noisolate)
        Q(i, j) = min4(ssOK(i, i) ?  g_multi[1] + Q(i + 1, j) :  INFINITY,
                       ssOK(j, j) ?  g_multi[1] + Q(i, j - 1) :  INFINITY,
                       i <= j - 2 ?  g_multi[2] + au + Es(i, j) + Qprime(i + 1,
j - 1) :  INFINITY,
                       QM(i, j));
      else
        Q(i, j) = min4(ssOK(i, i) ?  g_multi[1] + Q(i + 1, j) :  INFINITY,
ssOK(j, j) ?  g_multi[1] + Q(i, j - 1) :  INFINITY, g_multi[2] + au + Qprime(i,
j), QM(i, j));

      if (!g_nodangle)
      {
        if (g_noisolate)
        {
          if (i < j - TURN - 3)
          {
            Q(i, j) = min2(Q(i, j), g_multi[1] + g_multi[2] + auPenalty(i + 1,
j) + Ed5(j, i + 1) + Es(i + 1, j) + Qprime(i + 2, j - 1));
            Q(i, j) = min2(Q(i, j), g_multi[1] + g_multi[2] + auPenalty(i, j -
1) + Ed3(j - 1, i) + Es(i, j - 1) + Qprime(i + 1, j - 2));
          }
          if (i < j - TURN - 4)
            Q(i, j) = min2(Q(i, j), 2.0 * g_multi[1] + g_multi[2] + auPenalty(i
+ 1, j - 1) + Etstackm(j - 1, i + 1) + Es(i + 1, j - 1) + Qprime(i + 2, j - 2));
        }
        else
        {
          Q(i, j) = min2(Q(i, j), g_multi[1] + g_multi[2] + auPenalty(i + 1, j)
+ Ed5(j, i + 1) + Qprime(i + 1, j));
          Q(i, j) = min2(Q(i, j), g_multi[1] + g_multi[2] + auPenalty(i, j - 1)
+ Ed3(j - 1, i) + Qprime(i, j - 1));
          if (i < j - TURN - 2)
            Q(i, j) = min2(Q(i, j), 2.0 * g_multi[1] + g_multi[2] + auPenalty(i
+ 1, j - 1) + Etstackm(j - 1, i + 1) + Qprime(i + 1, j - 1));
        }
      }
    }
  }
}
```