THESIS

VIENNARNA - OPTIMIZING A REAL-WORLD RNA FOLDING PROGRAM

Submitted by

Vidit V. Save

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Fall 2023

Master's Committee:

Advisor: Sanjay Rajopadhye

Shrideep Pallickara
Taiowa Montgomery

ABSTRACT

VIENNARNA - OPTIMIZING A REAL-WORLD RNA FOLDING PROGRAM

RNA folding is the dynamic process of intra-molecular interactions that makes a linear RNA molecule acquire a secondary structure. Predicting the acquired secondary structure is critical for gene regulation, disease characterization, and improving drug design. ViennaRNA is a highly utilized tool in the synthetic biology community to predict RNA secondary structures. This package is constantly updated to add new features and uses techniques like vectorization to boost its single-core performance. However, reviewing the package revealed that adopting known HPC optimizations to the code base could significantly improve the current performance.

Optimizing a program with over 10k lines of code creates several software engineering challenges. Hence, toy kernels that mimic the code's behavior were initially used to explore possible optimizations. These kernels helped save compilation time and boil down the optimization process for the multi-branch loop prediction, a part of RNAfold, to 5 simple steps.

On applying the optimizations described in this thesis, a $2\times$ speedup can be observed for the entire program with a $4.2\times$ speedup for the optimized part of the code. Using Intel's Roofline toolkit shows that applying these optimizations helped achieve cache utilization close to the theoretical L1 bandwidth of the machine. As a part of this thesis, incremental patches were created to integrate optimizations without disrupting the code base while ensuring the program's correctness.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to Dr. Sanjay Rajopadhye for his insights and guidance, which were indispensable to this Master's thesis. I genuinely appreciate my work and research group colleagues who provided encouragement and a supportive academic environment. I am grateful to the Computer Science Department of Colorado State University for providing access to servers, which were critical to benchmark this thesis. Finally, I would like to thank my family and friends for their encouragement.

DEDICATION

This thesis would not have been possible without my family's love, support, and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Ribonucleic Acid (RNA) is a type of nucleic acid molecule that plays a vital role in protein synthesis, gene expression, and other biological processes. It consists of four nitrogenous bases, namely Adenine (A), Guanine (G), Cytosine (C), and Uracil (U). [6] The single-strand nature of RNA facilitates base-pairing interactions to create stable secondary structures. This process is called RNA folding and the acquired structure is known as the RNA secondary or tertiary structure. The secondary structure of an RNA molecule is determined by the sequence of its nucleotides, which can fold back on themselves and form complementary base pairs, creating a variety of structural motifs.

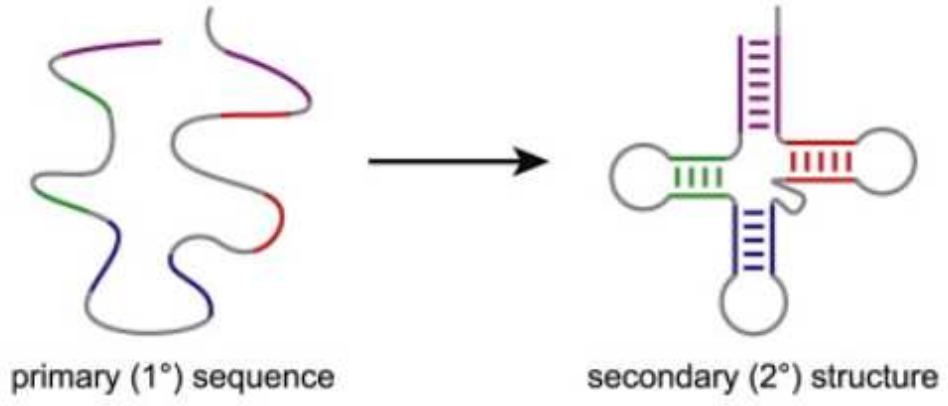


Figure 1.1: RNA secondary structure [19]

Predicting the RNA secondary structure is crucial to identify functional domains within the molecule as well as for other drug and therapeutic applications. Hence, a significant investment in developing computational algorithms to predict these secondary structures has been seen over the years. There have been a wide range of approaches, from simple and early dynamic programming approaches like Nussinov's algorithm to thermodynamically based packages like ViennaRNA, which use the thermodynamic properties of an RNA molecule for prediction.

ViennaRNA [11] is a software package widely used to analyze RNA molecules in bioinformatics, molecular biology, and related research fields. The package, developed by researchers at the University of Vienna, offers algorithms for predicting RNA secondary structure, minimum free energy, RNA-RNA interactions, and various other applications. According to their website, the ViennaRNA package 2.0+ has been downloaded over 87,600 times.

The authors of the ViennaRNA package have invested in manually vectorizing the code to obtain better performance on modern systems. However, their approach could be significantly improved by using optimizations like tiling and loop permutation to bring the code performance for RNAfold close to the L1 cache bandwidth. This highlights the disconnect between the High-Performance Computing (HPC) and Bioinformatics communities concerning program optimizations. Hence, the main objective of this thesis is to try and bridge this gap by developing incremental patches to add known optimizations while maintaining the existing code structure.

1.1 Contribution

This thesis focuses on optimizing and documenting the incremental changes made to the RNAfold program. This creates a platform for others to replicate findings, explore optimization choices, and further optimize other parts of this package. Throughout the modifications, it was ensured that the original code structure would be retained, leading to some software engineering challenges.

The patch developed as a part of this thesis uses the ViennaRNA package 2.5.1 to apply, compile, test, and benchmark optimizations automatically. The applied optimizations for the multi-branch computation in RNAfold can help us obtain a $2\times$ speedup for sufficiently long RNA strand sizes.

1.2 Thesis structure

- Chapter 2 presents the background and an overview of the tools and techniques used. This chapter dives into the background necessary to understand the basics of the RNAfold program in ViennaRNA. Further, SIMD vectorization and Loop tiling are described. Finally, the tools used for this thesis are described briefly, namely the Intel Advisor's Roofline Toolkit and AlphaZ.

- Chapter 3 explores the optimization approaches that could be implemented using some reference kernels. This chapter helps the reader understand the significance of using these reference kernels. Further, based on use cases, three different toy kernels are explored to understand parallelism drawbacks, tiling strategies, and middle-phase separation. Each toy kernel includes a brief insight into how these findings can be translated into the RNAfold program.

- Chapter 4 describes the process of integrating optimizations into the RNAfold program. The chapter starts by describing the software engineering challenges and then moves to the actual steps followed for optimization. A glimpse of the correctness testing framework follows the five optimization steps.

- Chapter 5 provides a dedicated analysis of the performance and ways to replicate the results. The results section is broadly split into the actual code and the code without the largest interior loops. This section also describes the machine setup used and details of the custom script to benchmark the codes.

- Chapter 6 summarizes findings and outlines potential areas for future research like parallelism, interior loop optimization, and multi-strand computation.

Chapter 2

Background

2.1 Vienna RNA

ViennaRNA [11] consists of a C code library and 31 stand-alone programs to predict and compare RNA secondary structures. The package is available in many forms like precompiled binaries, source code, bindings for Python/Perl, and a web server. ViennaRNA uses thermodynamic models, such as Turner's nearest model [21], in which a secondary structure is decomposed into several characteristic substructures. [17] The figure below shows some of these substructures namely hairpin loops, internal loops, bulge loops, multi-branch loops, and external loops as shown in the figure:

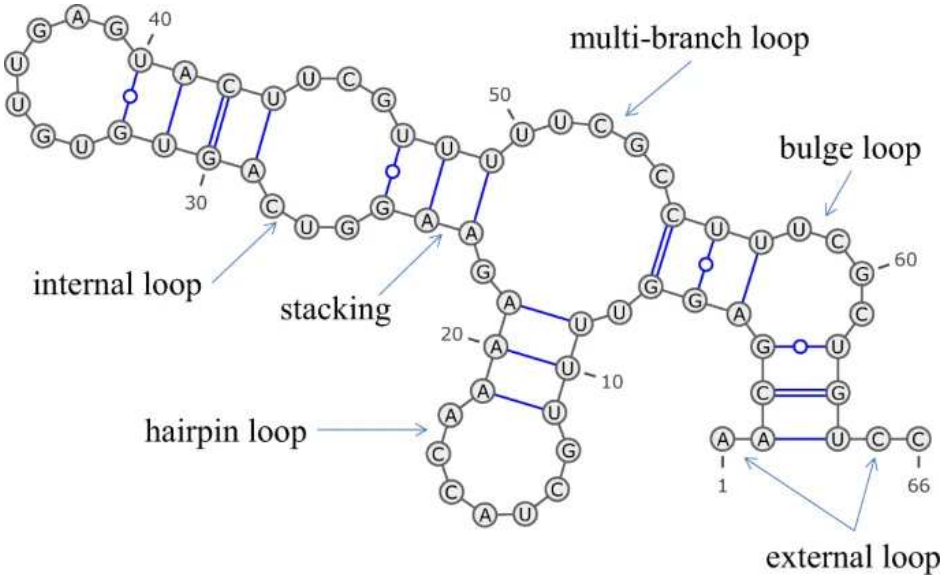


Figure 2.1: Different bond types calculated in ViennaRNA [17]

The source code for ViennaRNA contains 360,772 lines of code. Hence, this thesis's optimization scope will be limited to the multi-branch computation of RNAfold. This loop structure was selected because it accounts for over 60% of the program execution time for realistic problem sizes.

A simplified code structure of the minimum free energy function of the RNAfold program is given below, along with a table providing the lines of code present in the files this thesis modifies:

Pseudo Code for single stranded mfe.c

```
for (i = length - 1; i >= 1; i--) {
   for (j = i + 1; j <= length; j++) {
      decompose_pair();
      vrna_E_ml_stems_fast();
      E_ml_rightmost_stem();
   }
   rotate_aux_arrays();
}
```

Table 2.1: Lines of code in the RNAfold program (Path is relative to src/ViennaRNA)

Program Description	Relative File Path	Lines
Minimum free energy	./mfe.c	3,377
Multibranch Loops	./loops/multibranch.c	2,037
Higher Order SIMD	./utils/higher_order_functions_<instruction_set>.c	718

2.2 Optimization techniques used

2.2.1 SIMD Vectorization

Vectorization is the process of transforming a scalar operation, acting on individual data elements (SISD), to a single instruction that concurrently operates on multiple data elements (SIMD). [16]

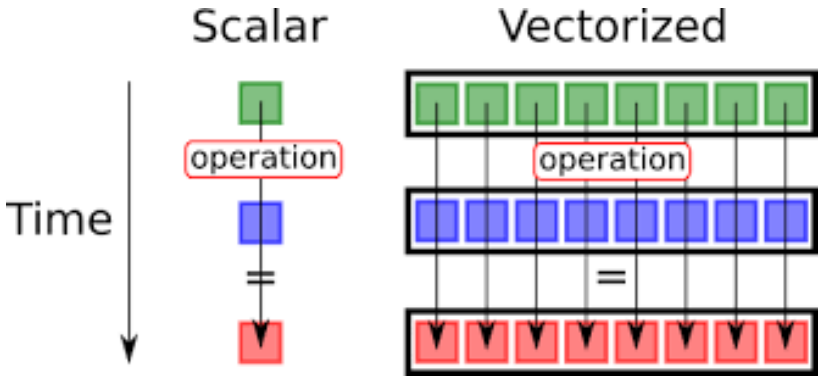


Figure 2.2: Scalar vs AVX2 Vectorized operations

There are two ways to introduce vector intrinsics to a codebase. The first is auto-vectorization, in which the compiler adds vector intrinsics to a code deemed suitable for vectorization. GCC's optimization flag "`-O3`" is an example of enabling automatic vectorization. Adding the "`-fopt-info-vec-optimized`" flag generates an optimization report with details of the vectorized and non-vectorized loops. The second is manual vectorization, in which vector intrinsics must be manually implemented in the codebase based on the selected CPU architecture. General-purpose compilers often cannot modify schedules to exploit vectorization, thus creating a need to implement architecture-specific manual vectorization. Intel's Intrinsics Guide website [8] provides resources for programmers to use intrinsics and understand the latency / throughput values. The "`lscpu`" command in Linux can be used to identify intrinsics flags available in the selected architecture. The table below describes some of the major instruction sets used in Intel CPUs:

Table 2.2: Description of Basic Instruction Sets

Instruction Set	Vector Length	FP32 Operations per vector
SSE	128-bits	4 FP32 ops per vector
AVX/AVX2	256-bits	8 FP32 ops per vector
AVX512	512-bits	16 FP32 ops per vector

The RNAfold program in ViennaRNA uses manually optimized SSE and AVX512 intrinsics in the codebase. However, their approach to vectorization is not efficient as the schedule requires the results of the vectorized operation to be accumulated into a single point. Not only does this decrease performance, but it also introduces the need for additional computations in the form of infinity checks.

2.2.2 Loop Tiling

Tiling [23] is an optimization technique, usually applied to loops, that orders the application data accesses to maximize the number of cache hits [5]. This transformation improves the data and spatial locality by avoiding unnecessary and expensive memory access from higher levels of the memory hierarchy.

Tile size exploration is an essential part of tiling. A too-large tile size could create cache misses between subsequent accesses, whereas a really small tile size might not saturate the available compute units. Hence, several tile sizes must be explored to determine the optimal one based on the selected application.

Loop permutation and Vectorization can be combined with tiling to improve the code's performance further. The goal is to determine legal permutations to enable vectorization of the innermost tile. A "legal" schedule is a schedule that takes into account the dependencies between tasks and ensures that tasks are executed in the correct order. The tasks mentioned above are the operations performed inside the innermost loop body.

The examples below use the blue arrows to help understand the time at which a point is computed in the base and tiled codes.

Base Code

```
t = 0;
for i from (0, N):
   for j from (0, N):
      c[i][j] = t++;
```

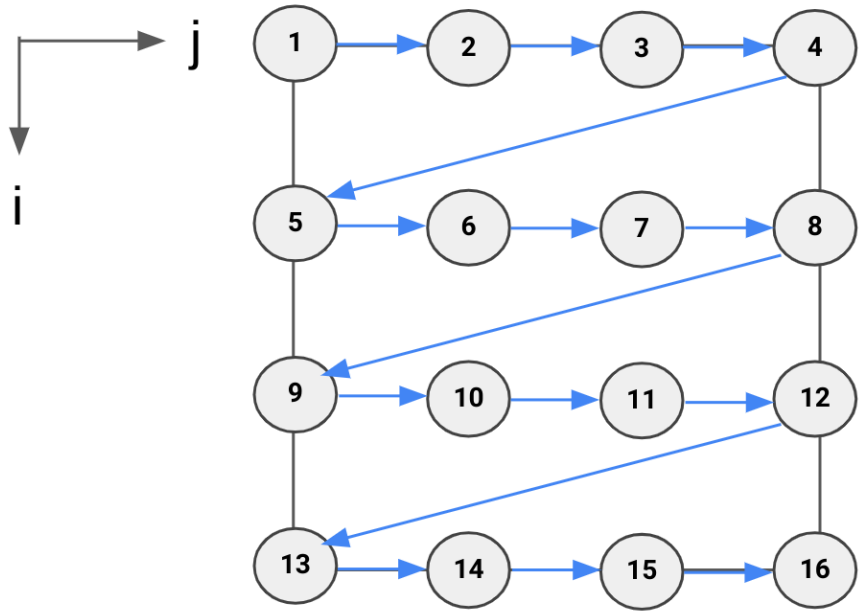


Figure 2.3: Base Code Execution Order

Tiled Code (ready to be vectorized)

```
t = 0;
for ii from (0, N, tile):
  for jj from (0, N, tile):
    for i from (ii, MIN(N,ii+tile),1):
      for j from (jj, MIN(N,jj+tile),1):
        c[i][j] = t++;
```

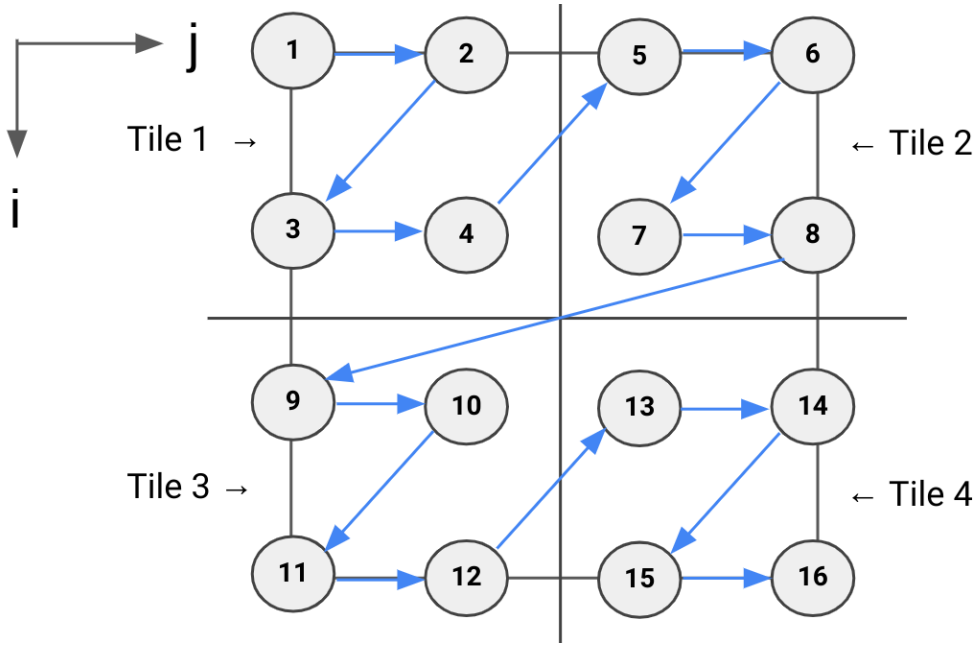


Figure 2.4: Tiled Code Execution Order

2.3 Tools used

2.3.1 Roofline Model / Intel Advisor's Roofline Toolkit

The roofline model [22] is a performance modeling technique used to identify performance bottlenecks and analyze performance on a given hardware platform. Intel's Roofline Toolkit [18] provides an intuitive visual representation of application performance in relation to hardware limitations like memory bandwidth and computational peaks. [2] It also helps to identify the time spent in each function/loop executed, which can be helpful to trace function calls and manual GFLOP calculation. A simplified example of the GFLOPs vs. Arithmetic Intensity plot produced by the tool is shown below.

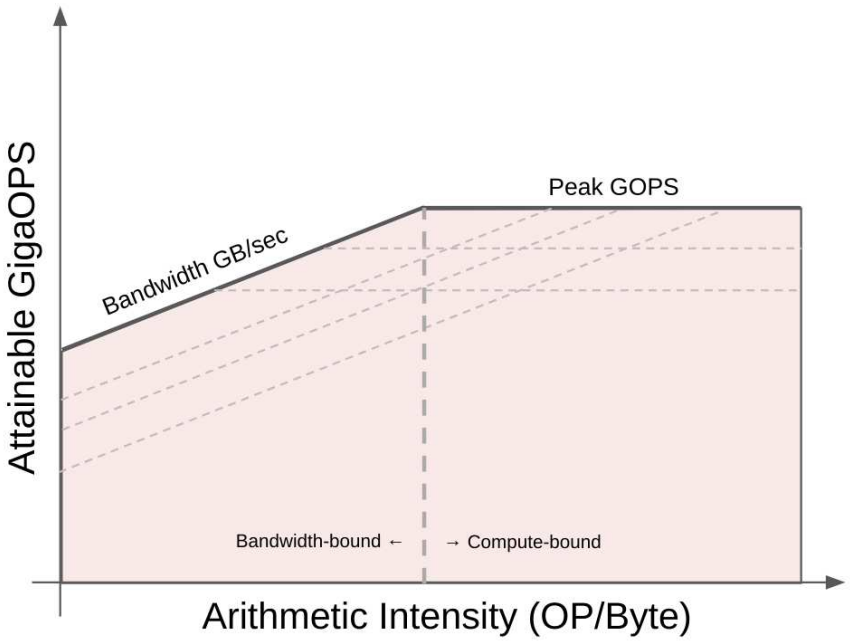


Figure 2.5: Basics of Roofline Model highlighting the Memory/ Compute bound sections

2.3.2 ALPHA/ AlphaZ

ALPHA [13] is a strongly typed functional language based on systems of affine recurrence equations defined over polyhedral domains. AlphaZ [1] is a tool that allows program transformations and user-directed compilation of ALPHA programs. [14] The code optimization process in AlphaZ starts with the input specification in which mathematical equations are used to express computations. After this, the compilation script takes inputs, like the schedule, memory mapping, transformations, etc., from the user to generate optimized C code.

As a part of this thesis, AlphaZ was used to generate different schedules to enable parallelism and tiling for a toy kernel. The produced C codes were benchmarked to decide the optimal strategy to optimize the ViennaRNA code.

Chapter 3

Reference Kernels / Toy Programs

3.1 Significance of exploring toy kernels

This section will discuss the motivation for using toy kernels to optimize the ViennaRNA package. The toy kernels selected closely mimic the behavior of the actual RNA folding program. Hence, optimizations and explorations performed on these reference programs can be directly translated to the ViennaRNA package. Here are some more benefits of using toy kernels:

3.1.1 Savings in Compile time

As mentioned in the background section, the ViennaRNA package consists of 31 kernels. The compilation process recursively iterates through each directory to compile various kernels and then creates the Python and Perl (Swig) bindings. This process requires 25-30 minutes. However, reducing it to 5-7 minutes is possible by disabling the scripting interfaces and using multiple cores for the make command. From an experimentation standpoint, waiting 5 minutes for compilation to test every minor change seems like a long time.

3.1.2 Faster Program Exploration

The most essential part of program transformation is validation of correctness. Since the package has a lot of variables whose dependence needs to be respected, this process could take a long time. Using a toy kernel can reduce the dependencies to explore the problem space more thoroughly and quickly.

Loop tiling and permutation cause drastic changes to the code base. But, one of the goals of this thesis was to maintain the package's code structure while improving its performance. Using toy kernels helped to explore possibilities to bypass this and revisit some known facts about the code behavior with transformations.

3.2 Nussinov: Toy kernel to explore RNAfold

3.2.1 Kernel Overview

Nussinov's algorithm uses a dynamic programming approach to predict the RNA secondary structure. The algorithm was named after Ruth Nussinov, who introduced it in 1978 in a paper titled "Fast algorithm for predicting the secondary structure of single-stranded RNA". [15] Nussinov's algorithm serves as a foundation for more advanced algorithms that handle pseudoknots in the secondary structure prediction.

$$X[i,j] = \begin{cases} i = j: & W_{\mathrm{i,j}} \\ i < j: & W_{\mathrm{i,j}} + \max\limits_{i=[k,j)} \left(X[i,k] + X[k+1,j]\right) \end{cases} \tag{3.1}$$

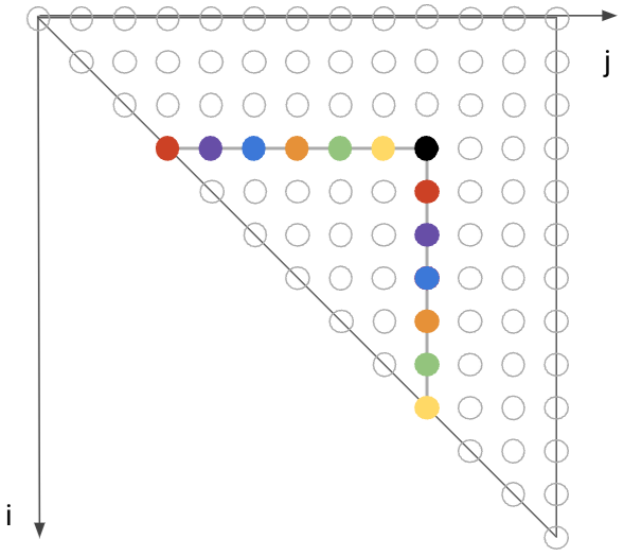


Figure 3.1: Memory space accessed to compute one point in the Nussionv-like kernel

The recurrence equation (3.1) shows that the weight values ($\mathtt{W_{ij}}$) for each element in the triangular space are provided, i.e., `i == j`. To compute each inner point ($\mathtt{X_{ij}}$), all the south and west points in the iteration space need to be read. As shown in the figure above, the values in similar colored points are added, and the maximum value of the results is selected. The obtained

value is added to the point's initial weight to obtain the value of the current point denoted by the black point.

The multi-branch loop computation in the RNAfold program of ViennaRNA operates similarly to this Nussinov-like kernel. Instead of a max-plus operation, the RNAfold program performs a min-plus operation. In terms of the data required, the Nussinov-like kernel uses two triangular matrices, "`X`" and "`W`", however, the RNAfold program uses the following three matrices:

- `fML[ij]` : A triangular matrix to store intermediate results and read all points south of the current point
- `Fmi[j]` : A $2 \times N$ array list to store the current and previous rows. It is used to read all the points to the west of the current point
- `DMLi[j]` : A $3 \times N$ array list to store the current and two previous rows. `DMLi[j]` is used to read and store the computed results.

3.2.2 Exploring Various Schedules

AlphaZ automatically generated parallel and tiled kernel implementations using the recurrence equations. The table below describes the generated schedules:

Table 3.1: Describe Schedules for Nussinov-like toy kernel

Schedule Name	Description
<-i,k,j> (Not Vectorized)	Bottom to top left to right without vectorization
<-i,k,j> (Vectorized)	Enables vectorization for -IKJ Schedule
<d,j,k> (Tiled)	Enables 8x8 tiling
<d,j,k> (Tiled Mutlicore)	Use 8x8 tiles with 6 threads

On benchmarking these codes, it was surprisingly observed that the <-i,k,j> vectorized schedule using single core outperformed parallel implementations. For the problem sizes 10K to 15K, it was observed that increasing the number of cores decreased performance. The performance charts comparing these schedules are shown below:

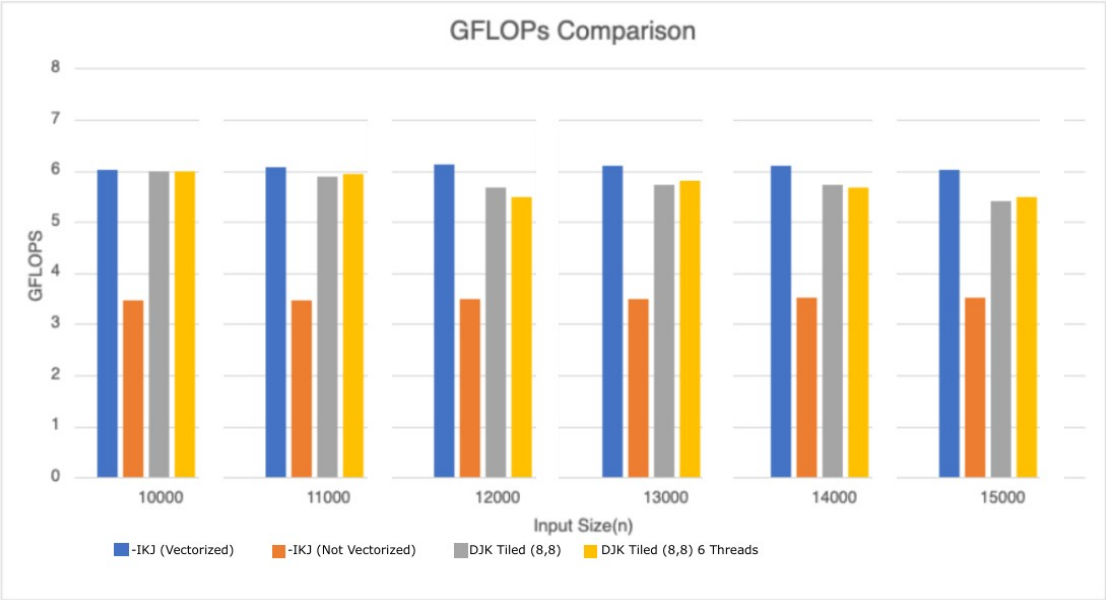


Figure 3.2: GFLOPs comparison for different Nussinov-like kernel schedules

3.2.3 Insights from the Kernel

At a glance, it seems counterintuitive that parallelism decreased performance. The roofline analysis of the program shows that the most expensive compute was close to the DRAM bandwidth for the problem size of 16,000. However, some manual computation reveals that it is possible to fit the entire data needed in the L3 cache for floating-point values.

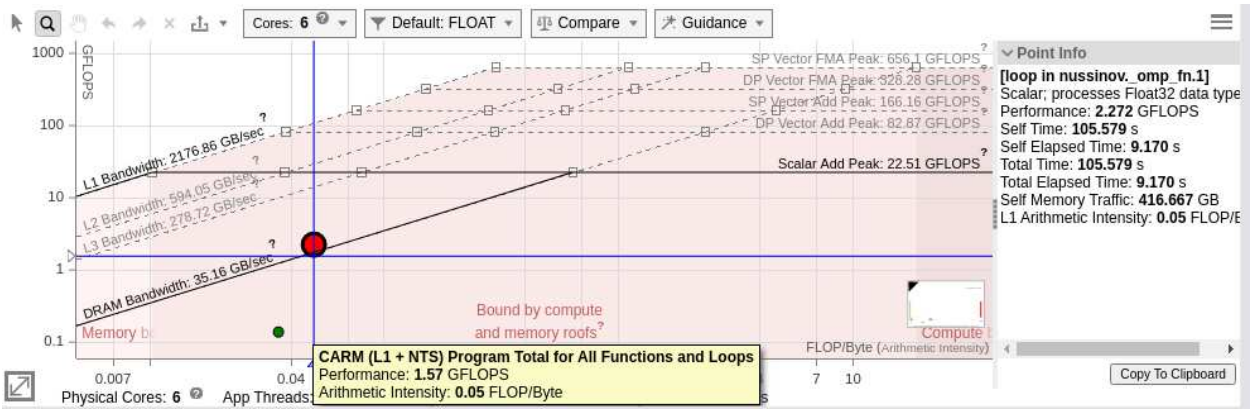


Figure 3.3: Roofline chart for <d,j,k> shows point close to DRAM

CPUs have private L1 and L2 caches but share the L3 level cache. Since the L3 storage is shared, using 6 cores causes L3 memory eviction, leading to increased DRAM accesses. This experiment verified the known finding that using multiple cores on RNA folding packages often hurt the performance of the code due to memory bandwidth-related issues. The figure below demonstrates what amount of the iteration space needs elements from the DRAM (Dark Blue region) when all 6 cores run in parallel.

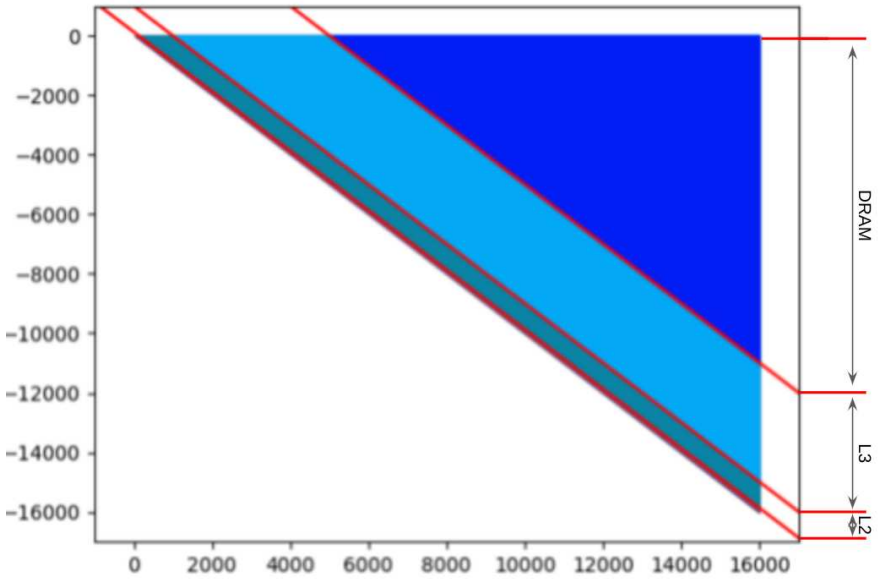


Figure 3.4: Iteration space segmentation based on accesses from different levels of memory

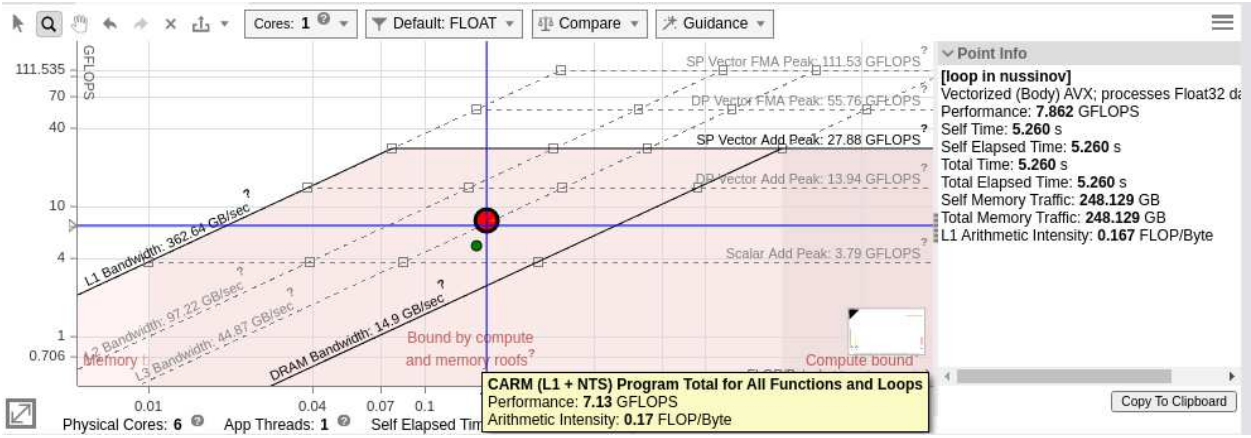


Figure 3.5: Roofline chart for <-i,k,j> shows point close to L3

Figure 3.5 shows that using a single core with the <-i,k,j> vectorized schedule puts the expensive compute above the L3 bandwidth. Since the roofline chart reveals the code to be L3 memory-bound, the program's cache locality must be improved using multi-level tiling to put us in the L1 cache bandwidth. This observation aligns with the results by Langdon et al. [9] and Li et al. [10] and highlights the need for using tiling transformation to improve performance for single as well as multicore CPU computation.

3.3 SmithWaterman: Toy Kernel to explore Tiling

3.3.1 Kernel Overview

The Smithwaterman [20] kernel is a computational algorithm used in bioinformatics to perform sequence alignment between two nucleotides or protein sequences. The algorithm is based on dynamic programming and identifies the optimal local alignment between two sequences by scoring the matches, mismatches, and gaps between them.

This SmithWaterman-like kernel, mentioned below, was used to optimize locality using tiling and permuting to enable vectorization while respecting the memory dependencies. The pseudo-code for the kernel is shown using the equation (3.2):

$$X[i\%2][j] = \begin{cases} i = j: & W_{\text{i,j}} \\ i < j: & \max(2 * X[i\%2][j-1], X[(i+1)\%2][j] + X[(i+1)\%2][j-1]) \end{cases} \tag{3.2}$$

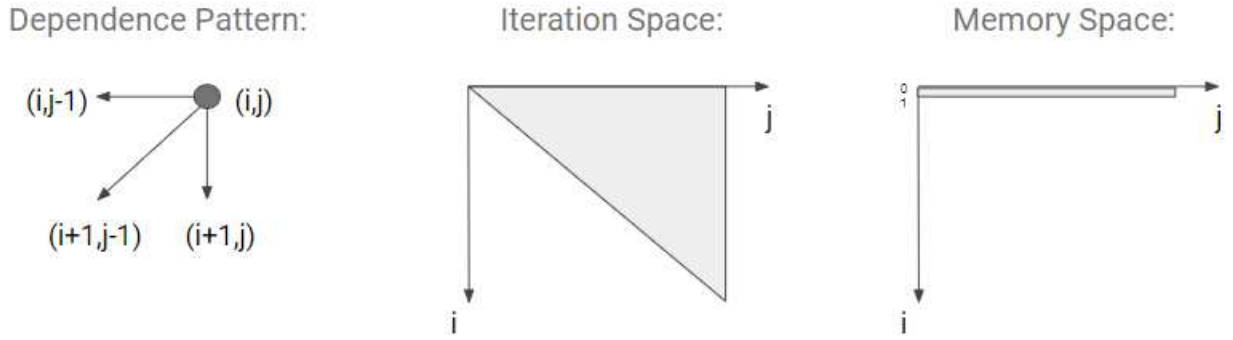


Figure 3.6: Dependence pattern, Iteration and Memory space of SmithWaterman like kernel

The dependence pattern describes the values necessary to compute any given point. For computing the point `(i,j)`, the south `(i+1,j)`, west `(i,j-1)`, and southwest `(i+1,j-1)` points are needed. The second and third diagrams show that the iteration space is triangular, but the memory space is linear with two rows. Hence, it can be inferred that the output value is

overwritten every second iteration, and loop tiling will require careful planning with additional memory allocations.

In the base ViennaRNA code, a <-i,j,k> schedule is followed. This test kernel aims to find a way to convert the schedule to a tiled <-i,j,k>. Tiling would allow us to improve data locality and permute loops for vectorizing the code while respecting the memory dependencies.

3.3.2 Exploring Optimizations

The memory allocation used in this kernel is one of the main reasons why conventional tiling approaches cannot be used. Since the output is overwritten every second iteration, having a tile height greater than two is only possible by allocating additional memory. The brute force solution would be to allocate an additional triangular matrix to store intermediate results. However, as previously noted, RNA kernels are often memory-bound, and adding quadratic memory is not an optimal strategy.

The amount of additional memory required can be lowered by taking advantage of the recurrence relation. This approach can be explained by using the "tiled" iteration space diagram below:

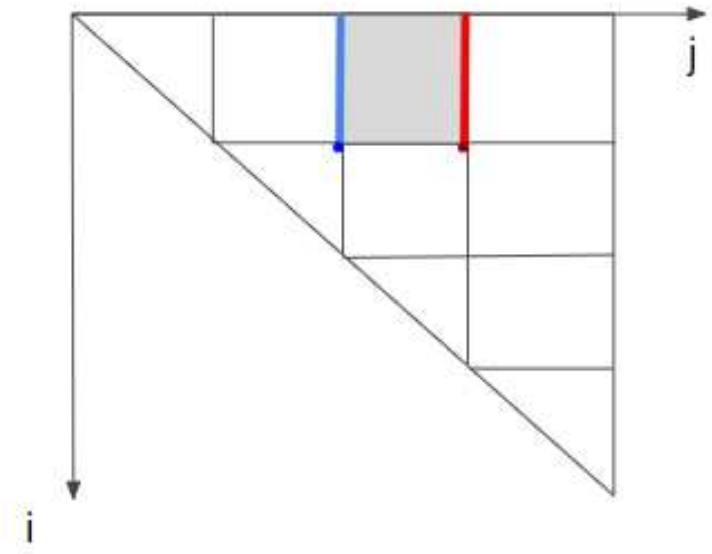


Figure 3.7: Additional I/O memory allocated for Smithwaterman like kernel

Each point in the gray box represents points that do not need elements outside the given tile. An exception to this is the lower side of the box. However, this result is already present in the memory and is not initially overwritten.

The iteration points shown by the blue line are the points that need two elements `(i,j-1)` and `(i+1,j-1)` outside the tile to be computed. These points are overwritten with every second iteration of `i`, causing tiling problems. A similar thing can be observed for the red points, except that the output points are overwritten. Based on the recurrence equation, the iterations of `j` are expected to move from left to right and never in the other direction. Hence, the input points are no longer needed to proceed in the right direction and can be overwritten by output points.

By the above logic, `(ti + 2)` additional elements are needed to compute the results for each tile correctly, where `ti` is the tile height in the `i` direction. Assuming that the single core <-i,j> schedule is followed, as found to be optimal with the Nussinov experiment, no two inputs are required to exist at any given time. This reduces the required additional memory allocation to `(ti + 2)` for the entire computation.

Further, the iteration space for "`j`" into three parts to reduce the memory strain:

- Diagonal Tiles: (`ii == jj`): Diagonal inputs are predefined. They do not need additional memory for inputs but need to store output points
- Middle Tiles: Need `(ti + 2)` additional memory elements as explained for reading inputs and writing to output spaces
- End Tiles: `(jj + tj == N)`: Do not need output writes as these values will not be read by any other elements

3.3.3 Insights from Kernel

The key takeaway from this toy kernel was that the code can be legally tiled without drastically altering the code structure. An in-depth performance analysis of this section was not produced because adding more control flow with tiling would slightly decrease performance. However, in the long run, tiling is beneficial as it provides better cache reuse, which is crucial for RNA folding programs.

The "`mfe.c`" file in the RNAfold program contains a similar structure to this observed toy kernel. However, instead of 2 matrices in the toy kernel, the program use these three variables, which were updated using the findings of this kernel:

- `DML[i][j]` : A `(tile_size+2)` $\times$ `(N+1)` matrix to read/store the produced outputs
- `Fmi[i][j]` : A `(tile_size)` $\times$ `(N+1)` matrix to access the elements below the point to be computed
- `fML[i][j]` : A triangular matrix to access the elements to the right in the current row

3.4 Nussinov: Toy Kernel to explore Middle-Phase separation

3.4.1 Exploring Optimization

After the code is legally tiled, the next step would be to understand dependencies in the code to simplify them, as described in the 2015 paper by Wonnacott et al. [24] Considering a Nussinov-like kernel, each element needs all values to the left of it and all the values below it. Figure(i), shown below, is obtained by extending this idea for an entire tile. Further, Let us consider a point in the tile and create three sections of the iteration space, as shown in Figure(ii).

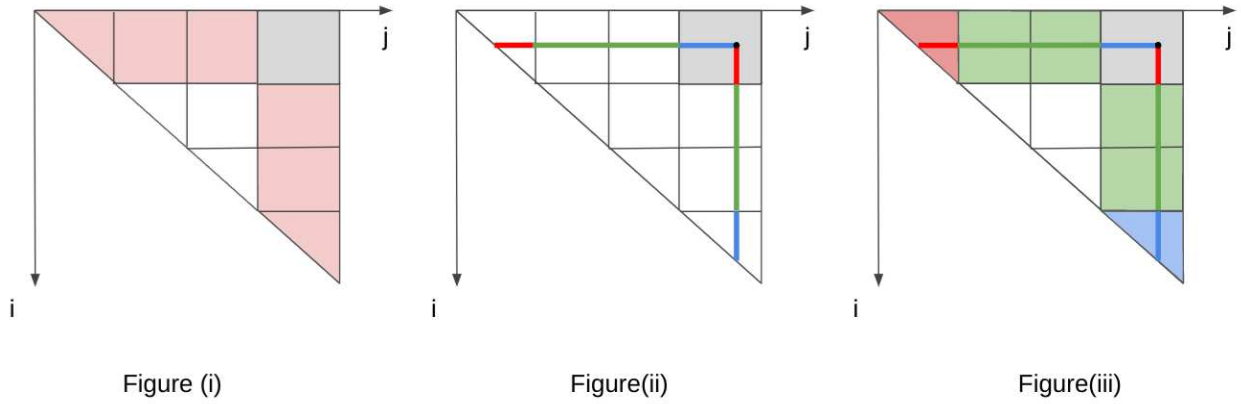


Figure 3.8: Process of identifying and extracting middle iterations for the Nussinov-like kernel

The left triangle, described by the points shown in red, needs values computed inside the gray square to progress. Following a tiled approach, it can be said that the elements needed for this computation have not been finally computed yet. Thus, proceeding without modifications would produce an incorrect solution. A similar thing can be observed for the blue points in the lower triangle. The points mentioned in green, surrounded by square tiles, create a section that does not require any points inside the current tile to proceed. It can be confidently said that the points necessary for this computation have already been processed on strictly following the bottom-to-top and left-to-right schedule.

In Figure (iii), the iterations identified by the green rectangles are extracted for performing out-of-order optimizations. The two triangular sections must be processed in the bottom-to-top and left-to-right order after the middle section to maintain program correctness.

Now, performance optimizations can be easily performed on the extracted middle section. As seen from the findings of the first toy kernel, the <-i,j,k> loop order is not vectorizable. Hence, it should be permuted to the <-i,k,j> loop order. Using the `-O3` optimization flag with `-fdump-tree-optimized` in GCC displays the optimization records, showing that the code was vectorized. The GCC compiler was chosen in this case to align with the compiler used in the make files of the ViennaRNA package. Further, OpenMP can be used to collapse the inner two loops to parallelize the computations of the middle section.

The pseudo-code transformation from the base code to the middle section separated code is shown below:

Base Code

```
for (i = N-1; i > 0; i--) {
   for (j = i; j < N; j++) {
      min = INF;
      for (k = i; k <= j-1; k++)
         min = MIN(min, C[i][k] + C[k+1][j]);
      C[i][j] = W[i][j] + min;
   }
}
```

Optimized Code with separated middle section

```
for (ii = N-1; ii > 0; ii-=T){
   for (jj = ii; jj <= N-1; jj+=T){
      // Middle Section
      for (i=MAX(ii,0); i > MAX(ii-T,0); i-=1){
         #pragma omp parallel for collapse(2)
         for (k = ii; k <= MAX(jj-T,0); k++)
            for (j=MAX(jj-T+1,i+1); j <= MIN(jj+T,jj); j+=1)
               C[i][j] = MIN(C[i][j], C[i][k] + C[k+1][j]);
      }
      // Patch Section
      for (i=MAX(ii,0); i > MAX(ii-T,0); i-=1){
         for (j=MAX(jj-T+1,i+1); j <= MIN(jj+T,jj); j+=1){
            // Left Triangle Computation
            for (k = i; k < ii; k++)
               C[i][j] = MIN(C[i][j], C[i][k] + C[k+1][j]);
            // Lower Triangle Computation
            for (k = MAX(jj-T,0)+1; k <= j-1; k++)
               C[i][j] = MIN(C[i][j], C[i][k] + C[k+1][j]);
         }
      }
   }
}
```

3.4.2 Insights from Kernel

The performance comparisons for the optimized and unoptimized codes are shown in the table below. This shows the huge gains achievable from combining vectorization and parallelism for the desired problem sizes. The Problem size in the table describes the nucleotide length, and the execution times for codes are measured in seconds.

Table 3.2: Nussinov-like Toy Kernel Execution times (in seconds)

Problem Size (kilo nt)	Original Code	Vectorized (-O3)	Parallelized (OpenMP)
10	607.710	121.8568	137.525
11	780.138	134.2988	154.540
12	1007.554	163.4512	192.020
13	1275.400	203.8966	211.922
14	1674.162	259.5046	241.494
15	2040.856	331.5308	270.699
16	2406.877	420.0044	342.915

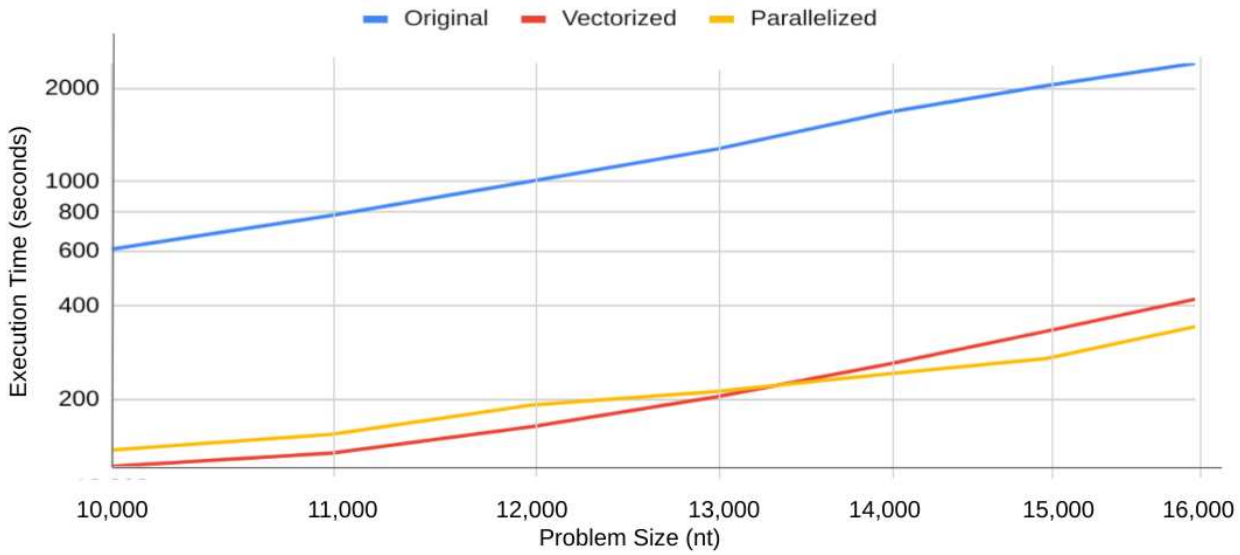


Figure 3.9: Benchmarking execution times for Nussinov-like toy kernel on a log-log scale

Unfortunately, the findings from this toy kernel cannot be directly used for the ViennaRNA package, as auto-vectorization actually lowers the code performance. Parallelization is complicated as dependencies and function calls must be modified to allow OpenMP to parallelize the for loops automatically. Switching to manual vector intrinsics fixes the vectorization issue. Manually parallelizing the kernel is outside the scope of this thesis but is briefly explored in the results and future work section.

Chapter 4

Implementation

4.1 Software engineering challenges

One of the main goals of the modifications presented in this thesis was to optimize the code while respecting the existing code structure of the package. This led to some software engineering challenges and design choices which are documented as part of the GitHub repository mentioned in Appendix A. Some of these challenges and their solutions are mentioned below:

4.1.1 Obscured Loop Bodies

Generally, kernels to be optimized by HPC engineers have well-defined loops. However, ViennaRNA uses a modular approach to programming, which means that the part of code to be optimized is buried under 3-5 function calls with several dependencies. This poses a software engineering challenge for optimization without modifying the code structure drastically. To solve this issue, an approach used in the third toy kernel was implemented to separate the independent computation parts as the middle section. General optimization principles could now be used to modify the "middle" section while the code structure is preserved for the patch section.

4.1.2 Additional memory allocation conforming to existing structure

As seen in the Smithwaterman-like kernel, additional memory spaces need to be allocated to ensure the correctness of results while respecting the dependencies of the kernel. For the RNAfold program, this modification must be implemented in all the inner function calls where the memory elements were previously accessed. Macros were defined to structure function calls using a part of the pointers to avoid significant code modifications. This modification ensures that knowledgeable programmers of this package can understand the inner workings of these functions without relearning them.

4.1.3 Validation of program Correctness

The simplest form of correctness verification consists of feeding various inputs to the program and verifying the correctness of the outputs. [7] Instead of verifying just the resulting minimum free energy value, a Python script was implemented to verify the three intermediate matrices as well. This same approach can be extended to verify the speedup on removing the time-consuming elements of the code, the $O(N^4)$ section of the interior loop.

4.2 Adding optimizations to code

As mentioned in the Introduction, the multiloop computation in the RNAfold program begins in the `mfe.c` file and performs the final action in the `higher_order_functions*.c` file. It is essential to disable the previous instances of manual vectorization by the ViennaRNA authors since the optimization mentioned in this thesis changes the schedule of operations. The configuration flag `-disable-simd` can be used for this purpose. Now, the dispatcher selects the default option present as `fun_zip_add_min_default` in the `higher_order_functions.c` file. After this initial setup, the results from the toy kernel explorations can be used to simplify the optimization process into the five steps mentioned below:

4.2.1 Allocating Memory and Using Macros

As seen in the SmithWaterman-like kernel, tilling the code without modifying the code structure can be done by allocating additional memory. This modification must be made for the helper/auxiliary arrays in the `mfe.c` file. First, the "`get_aux_arrays`" function must be modified to allocate additional memory for the `Fmi` and `DMLi` matrices. Since the computation for both needs all the values from the previous iteration, quadratic memory with tile height must be allocated. This matrix allocation scheme must also be used in the functions "`clear_aux_arrays`" and "`free_aux_arrays`". The dimensions for the new matrices are given below:

- `Fmi` is a matrix with dimensions `(tile_size) * (length_of_strand+1)`
- `DMLi` is a matrix with dimensions `(tile_size+2) * (length_of_strand+1)`

This modification creates a two-dimensional indexing pattern for `Fmi` and `DMLi`, whereas the actual script uses one-dimensional indexes. Memory macros can help directly use this new index pattern instead of manually updating all the inner function calls. The file `mfe.c` is responsible for passing particular rows of these helper/auxiliary arrays to the "`vrna_E_ml_stems_fast`" function. These can be rewritten as `Fmi[i%tile]` and `DMLi[i%(tile+2)]` to ensure that the correct values are passed post the tiling phase.

4.2.2 Implementing Loop Tiling

Splitting the tiling transformation into loop blocking and permutation steps is an efficient way to tackle a large codebase. This helps to ensure correctness by detecting code-breaking dependencies caused by function calls.

Loop blocking is the process of splitting the iteration space into blocks without altering the loop structure. This transformation should always produce the correct code and is a great step to verify the edge cases for the tiles. The SmithWaterman-like kernel very closely mimics the step for creating the loop blocks. Hence, the findings from this toy kernel directly translate to the changes needed in the `mfe.c` code.

The next step was permuting the loops to modify the code so that each tile was fully processed before moving to the next one. The outer two loops (`ii` and `jj`) are responsible for the inter-tile schedule, and the inner two loops (`i` and `j`) are responsible for the intra-tile schedule. The intra-tile schedule indices are set to `i` and `j` to ensure that the inner content of the computation performed remains unmodified. After this step, a thorough correctness validation is performed to ensure the legality of transformation for all the cases.

4.2.3 Middle/ Patch Phase Separation

As explained in section 3.4, splitting the middle iterations is vital to optimize a large section of the computation for the new schedule. The multi-branch computation consists of a triply nested loop to compute the min-plus operation. However, this is not directly visible and is hidden among function calls. The functions called for the multi-branch computation are shown below:

- `mfe.c : "fill_arrays"`
- `multibranch.c : "vrna_E_ml_stems_fast"`
- `multibranch.c : "E_ml_stems_fast"`
- `higher_order_functions.c : "vrna_fun_zip_add_min"`
- `higher_order_functions.c : "vrna_fun_zip_add_min_<sse/avx>"`

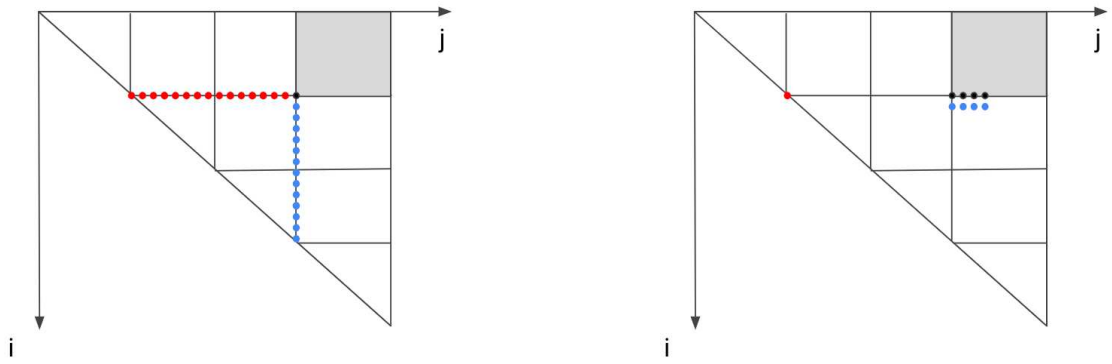
Exploring the function call stack shows that the loop of interest is a part of the "modular decomposition" section in `multibranch.c`. As mentioned in the third toy kernel exploration, this section must be split into three parts. To retain a structure similar to the base code, the middle section was named "`vrna_E_ml_stems_fast_middle`" and the two triangular section computations along with the rest of the code were named "`vrna_E_ml_stems_fast_patch`".

This transformation relies on vectorizing the middle section and performing the triangular sections linearly. The performance can be further improved by removing some fixed control flows. The middle section does not exist for the first two tiles of every new iteration of ii. Hence, these two iterations can be removed outside and performed directly with the non-vectorized setup, and the rest of the iterations, with the middle sections, are vectorized. Now, the patch section can be simplified by collapsing the function calls to expose the triply nested loops to be vectorized.

4.2.4 Loop Permutation and SIMD Vectorization

Vectorization is an important optimization to exploit the best performance on modern processors. The higher-order functions of the RNAfold program are responsible for code vectorization in the ViennaRNA package. The authors of ViennaRNA got a 29.34% speedup by using vectorization over non-vectorized code. The figures below show the contrast between the approach used by the authors of ViennaRNA, shown in Figure 4.1 (i), and our optimization approach, shown in Figure 4.1 (ii):

The original approach to vectorization seemed efficient mainly due to contiguity in memory blocks for both the input array (`Fmi`) and the matrix (`DMLi`). This approach puts all the elements to the left and bottom of a point into vectors to compute a given point. However, one major drawback of this approach is that all the resulting computation needs to be combined into a single point after each computation. This requires additional operations and infinity checks, which cause the program to inflate the number of actual operations performed. As a result, the performance of the program is significantly lowered.



(i) Original SIMD Approach (ii) Optimized SIMD Approach

Figure 4.1: Original vs Optimized Approach for vectorization

As a part of the approach proposed in this thesis, the loops were permuted such that "`j`" is the innermost loop instead of "`k`". Instead of computing a single point, an entire line of points can be computed using one point on the left and all the points below the current line. This approach can avoid infinity checks as the output of the sum operation is always guaranteed to be less than the maximum value for 32-bit vectors. However, there is a minor drawback with this approach. The points needed to compute `DMLi` are not contiguous in nature, and manual vector loading (`_mm_set_epi32`) is not efficient. A way to fix this would be to read four contiguous vectors (`_mm_load_si128`) and transpose them to get four of the required vectors. This results in a 2D memory layout for computation, shown by the figure 4.2:

The approach is described for SSE vectorization but can be extended to AVX2 using 8x8 tiles. The most efficient way to perform an 8x8 SIMD transpose operation using AVX2 has been described by zBoston. [4] This approach would create the lowest latency while not adding any additional operations to deviate from the actual roofline chart. AVX512 was not used in this thesis as Intel has dropped the support for the AVX512 instruction sets in the latest two generations of consumer chips. Manually vectorized codes have been developed for SSE, to have a fair comparison against the actual Vienna package, and AVX2, to demonstrate the best performance achievable for the selected architecture.

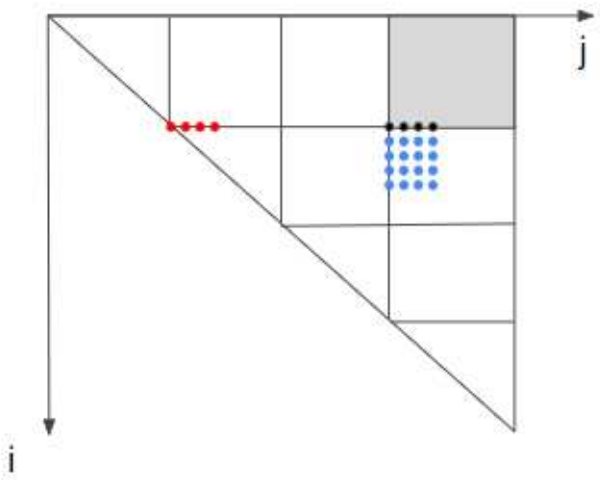


Figure 4.2: Memory accessed after unrolling loop to use transpose for SIMD

4.2.5 Simplifying Expensive Modulo Operations

Due to its cyclic nature, the modulo operator simplifies the addresses needed to access data for tiled codes. However, the modulo operator cannot be processed as a single operation and is often required to perform multiply, subtract, and divide operations in the ALU. This operator, although expensive, is still less expensive than the Transpose operation described in the previous section. Thus, the <k,j,i> loop permutation provides us with the best performance. Being in the innermost loop, eliminating this operation would significantly enhance the performance. A way to remove the modulo "%" operator can be described using the pseudo-code below:

Pseudo code of the innermost loop "i"

```
for (i = i_start; i >= i_last; i -= 1) {
   READ(fmi[i%tile])
}
```

Here, it can be seen that the bounds of the `i` loop go from `i_start` to `i_last`, and the data accessed is stored in the index `( i % tile )`. This means that the length of the actual data has cyclic indexes going from 0 to tile. It can be derived that the index starts at `( i_start % tile )`, eventually reaching zero, and starts again from `(tile - 1)` till it reaches `(( i_start % tile ) - 1)`. Hence, the existing loop can undergo fission to form two loops, as shown in the snippet below:

Pseudo code of the split innermost loop "i"

```
int i2end = i_start % tile;
for (i = i2end; i >= 0; i -= 1) {
   READ(fmi[i])
}
for (i=tile-1 ; i > i2end; i -= 1) {
   READ(fmi[i])
}
```

4.3 Overview of the Testing Framework

Code correctness validation is an integral part of testing. Hence, each successful compile is followed by a small code validation script implemented in Python. This script validates the output of the compiled code to the results obtained by the original code for the same input.

Inputs to the code validation script are generated during runtime by selecting a random combination of the characters "A", "C", "G", and "U". A snippet of the actual code to generate random inputs is given below:

Input generation for Testing Framework

```
input_list = ['A', 'C', 'G', 'U']
input_str = "".join([random.choice(input_list) for i in range(N)])
```

These generated inputs are stored in a temporary file and are provided to the original binary, followed by the newly compiled binary file. The outputs from "stdout" for each of these cases are stored to be compared. Along with the outputs, the execution time for each case is also stored in a variable.

Performance and Correctness comparisons should be repeatable. Hence, the testing script contains a variable "`N_repeat`", which repeats the above experiment N times. It is recommended to run the test at least 10 times so the script can remove values above and below two times the standard deviation before taking the average to produce reproducible results. The script also provides the speedup ratio as an output to see the achieved speedup at a glance.

Chapter 5

Results

5.1 Performance analysis

The testing framework was executed on Intel's Coffee Lake and Alder Lake processors for a thorough performance analysis across device architecture and cache sizes. The first CPU used was a 3rd generation Intel Xeon 2278G server processor clocked at a 3.4GHz base frequency with 32 GB of RAM. Another test was performed on the 12th generation Intel i7-12700K consumer processor with base performance and efficiency frequencies of 3.6GHz and 2.7GHz, respectively.

5.1.1 Initial Performance Analysis

The initial performance analysis was performed on the RNAfold program provided in the ViennaRNA package with no modifications. Here are the four programs that were compared:

- No SIMD: RNAfold with Vectorization disabled
- Original Code: RNAfold with no modifications. Defaults SSE vectorization (Selected processor does not support the AVX512 instruction set)
- Our Optimized SSE Code: Applies all the optimizations mentioned in the implementation section to the RNAfold program. Uses SSE vectorization for a fair comparison
- Our Optimized AVX2 Code: Uses AVX2 vectorization to show the best possible single-core performance for the selected CPU

Given below is the table that shows the runtime of the codes mentioned above. An average of $1.8\times$ and $2.2\times$ speedup was obtained over the original RNAfold program using the SSE and AVX2 intrinsics, respectively.

Table 5.1: Initial Execution times for Optimizations

Problem Size	No SIMD	Original	Optimized SSE	Optimized AVX2
10,000	171.638	125.088	75.634	64.202
11,000	221.680	159.739	92.744	78.053
12,000	280.914	200.028	114.782	94.760
13,000	350.075	247.375	137.351	112.137
14,000	429.169	300.551	162.694	131.870
15,000	520.731	362.687	193.460	149.995
16,000	620.195	428.097	209.331	171.459

The Roofline charts for the original case is provided below for a more in-depth analysis. The problem size for these figures, generated by Intel Advisor's Roofline toolkit, was set to nt=15,000. The experiments were performed with the highest accuracy settings to create repeatable results. However, it must be noted that this process may take over to an hour for generating each figure.

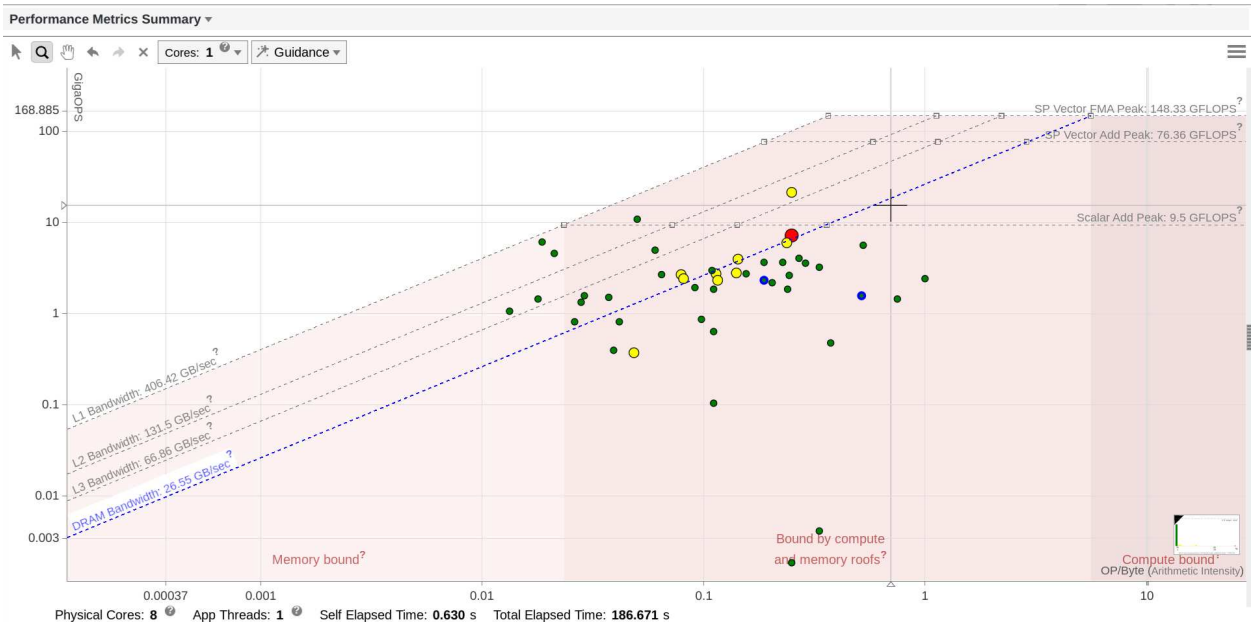


Figure 5.1: Roofline chart for Original Code shows computation in DRAM bandwidth

The original plot shows that the innermost loop of the multibranch computation lies way under the DRAM bandwidth level. The roofline model shows that this section computes 1.59 and 2.35

GFLOPS for the innermost and second loops respectively. However, as previously discussed, this value is an over approximation of the actual operations needed due to additional conditional flows present in the existing approach.

5.1.2 Performance Analysis with no internal loops

As seen by the red dots in the previous figure, the code spends a large proportion of the execution time on the internal loop computation. Since optimizing the internal loops is outside the scope of this thesis, the performance comparisons should be rerun by removing the interior loops to understand the speedup obtained by the optimizations described in this thesis.

Completely removing the internal loops from the program provides no meaningful results. Hence, only the most time-consuming section of this computation was removed. This was the quadruply nested loop with MAX_LOOPS set to 30. It must be noted that this change alters the expected output and is strictly for the purpose of performance measurement. The quadruply nested loops for both the Original and Modified code were removed for correctness validation.

The roofline charts for the SSE and AVX2 optimized multibranch code are shown below:

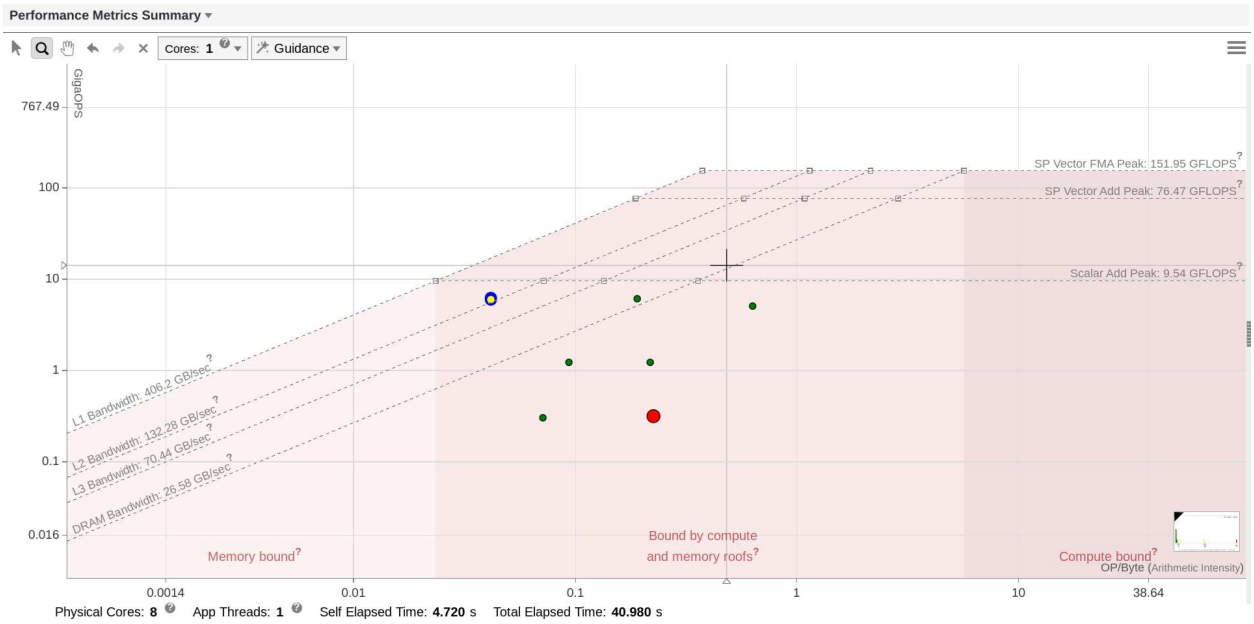


Figure 5.2: Roofline chart for Optimized SSE code (no interior) shows performance in L1 B/W

The roofline charts show that the innermost computation for the SSE and AVX2 code lies close to the L1 bandwidth. This means that the code exhibits excellent cache reuse. When compared to the original SSE code, using 2-level tiling brings the computations for the points of interest in the L1 bandwidth from the earlier DRAM bandwidth. However, the points of interest are stuck under the oblique part of the roofline. This means that even with the best optimizations, it is unlikely to reach the theoretical compute peak performance deliverable by the system without improving the arithmetic intensity.

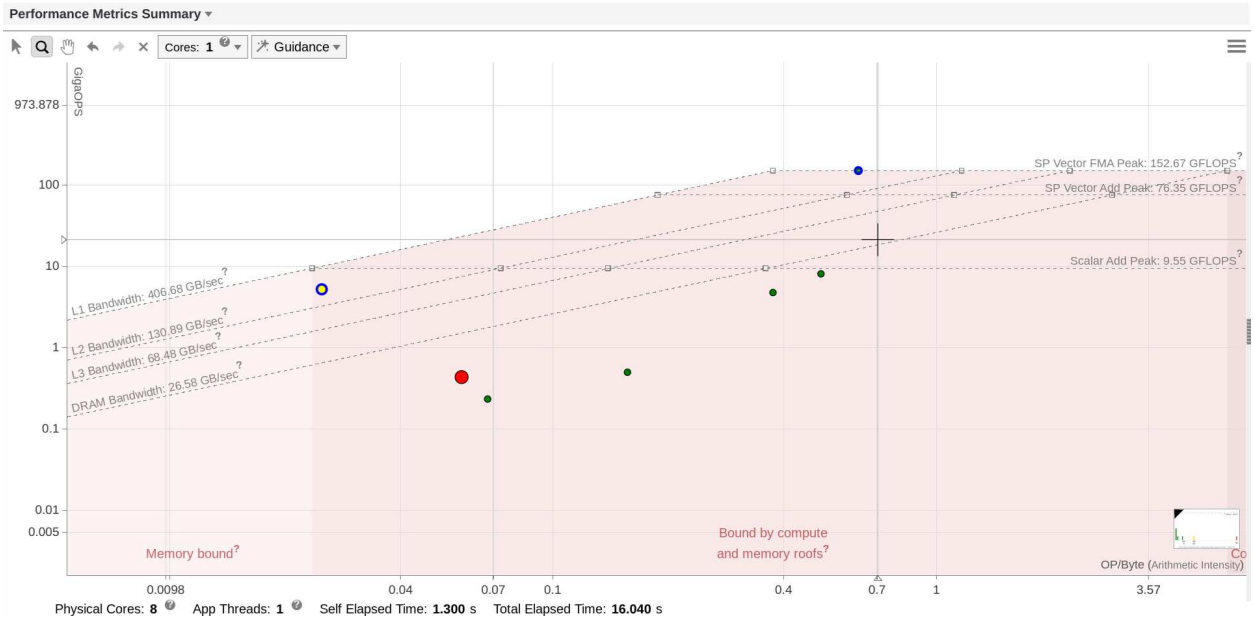


Figure 5.3: Roofline chart for Optimized SSE code (no interior) shows performance in L1 B/W

AVX2 is the best intrinsic available for the selected processor configuration and has a wider bit vector length than SSE. Thus, a slightly higher arithmetic intensity is expected for the code. Since the L1 cache is private for each core, a future task could look at utilizing multiple cores to improve the code performance further.

Given below are the performance comparisons between the Original and Modified code without internal loops.

Table 5.2: Execution times with Interior Loops removed

Problem Size	No SIMD	Original	Optimized SSE	Optimized AVX2
10,000	129.555	82.528	33.385	21.885
11,000	170.750	106.191	42.181	27.193
12,000	219.656	136.529	53.723	33.764
13,000	277.473	170.675	66.366	40.910
14,000	344.375	210.848	80.015	48.545
15,000	422.026	258.228	97.750	58.095
16,000	510.005	310.404	112.093	65.257

5.1.3 Comparing the execution times for large inputs

The tables below describes the key points to be inferred from the experiments performed:

Table 5.3: Speedup values compared to baselines with internal loops

Instruction Set	Speedup - No SIMD	Speedup - Original
SSE4	$2.56\times$	$1.81\times$
AVX2	$3.13\times$	$2.21\times$

Table 5.4: Speedup values compared to baselines without internal loops

Instruction Set	Speedup - No SIMD	Speedup - Original
SSE4	$4.19\times$	$2.59\times$
AVX2	$6.81\times$	$4.21\times$

A head-to-head comparison of the 8 cases is shown below using a log-log plot. Using a log-log plot helps us better understand the performance comparisons, as the series' slopes are almost constant. This constant is approximately 3, which is the complexity of the most expensive section (cubic). According to the graph below, our optimized script performs better than the Original RNAfold with interior loops disabled.

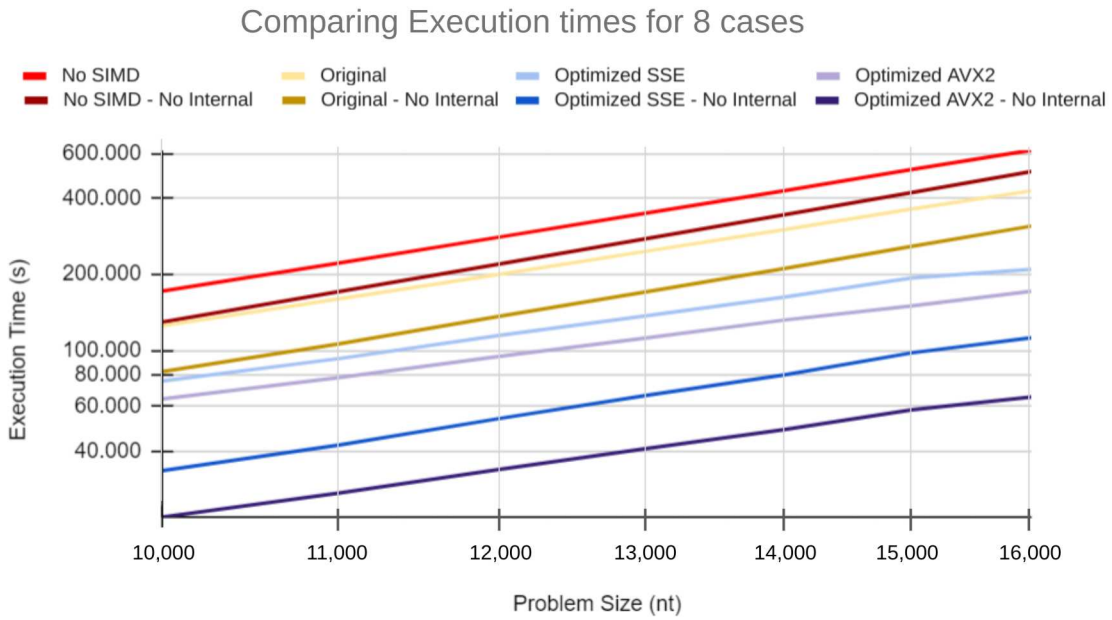


Figure 5.4: Compare execution times for all the tested cases on a log-log plot

5.1.4 Comparing the execution times for smaller inputs

This section describes the code behavior for smaller input sizes. The sizes compared here range from 1,000 to 10,000 in increments of 1,000. According to the graph shown below, it can be derived that the optimized code always performs better than the original code provided in the package. It is expected to observe a lower speedup for smaller problem sizes as the interior loop takes up a more significant section of the code's execution time. A chart with the execution times observed for smaller inputs is shown below. The lines in Figure 5.5 seem to converge close to problem size 1,000. This is observed because the `MAX_LOOPS=30` value makes the outer two loops have a size of 900, which dominates the execution time for smaller sizes.

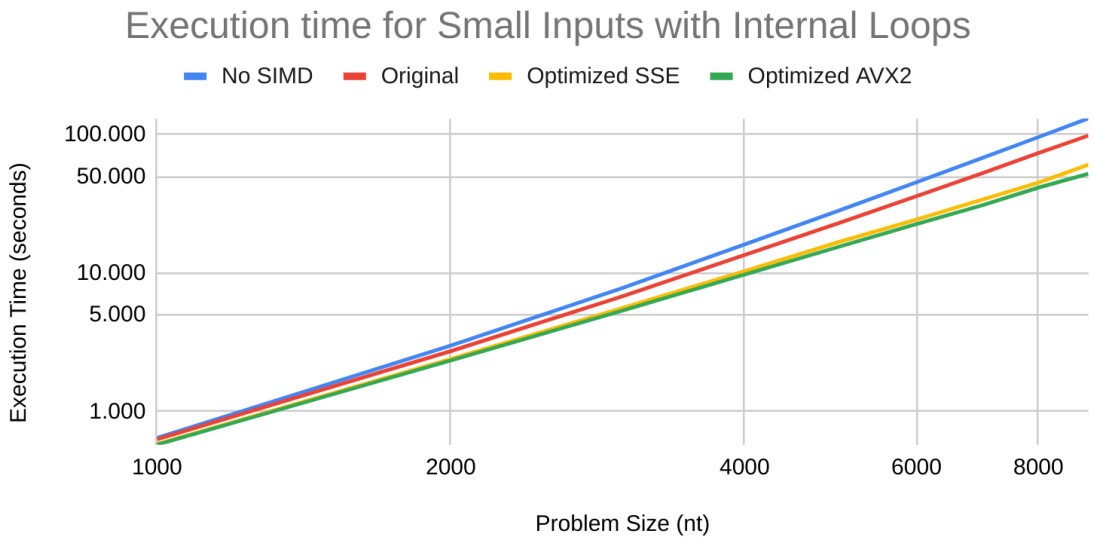


Figure 5.5: Execution times for small input sizes with internal loops

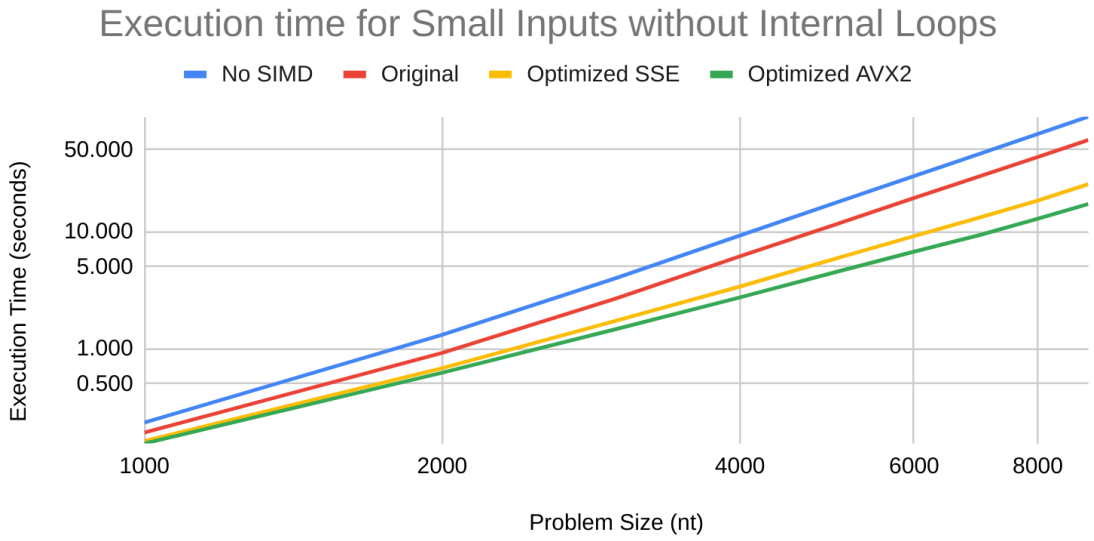


Figure 5.6: Execution times for small input sizes without internal loops

5.2 Replicating Results

One of the goals of this thesis was to provide a platform for others to make optimizations as well. One of the best ways for other developers to understand the performance implications of the applied changes would be to provide them with programs for comparison. However, doing so is not feasible, both in terms of time and disk space needed, for a package as big as ViennaRNA. Hence, an automated Python script was created to generate the modified source code, compile the package, and compare the runtimes against other binaries using simple bash commands. Using this, developers can execute the generated binaries and see the script changes made by individual modifications. A high-level overview of the binary generation and benchmarking process is mentioned below:

5.2.1 Creating Binaries

The starting point of the code is the downloaded source file for ViennaRNA 2.5.1. It has a relatively small size of 55MB compared to the actual source and compiled binaries of 1.2GB, making the script much more concise. The first input to the script is used to decide whether or not internal loops must be implemented. As previously mentioned, using internal loops gives the same results as the actual ViennaRNA package. In contrast, internal loops can be removed to isolate the speedup observed by the optimizations performed through this thesis. The second and third inputs to the script describe the points to be compared. It is always expected to compare two points with the same interior loop implementation to avoid correctness test errors.

Further, the implemented optimization scripts are copied to the appropriate folders before compiling to create binaries for the selected points. The script also has the option to reduce disk usage by deleting source codes on compiling. This can be helpful for cases in which sources are not needed but binaries are required for comparison.

5.2.2 Benchmarking / Testing

Two points of comparison are provided to the script as a part of the command line arguments. Based on the inputs, the script detects if the compiled binaries are already present in the expected folders. This is helpful to save time as the compilation process may take 5-7 minutes.

The testing script contains a small and a large test. Small tests are targeted to validate correctness during continued development. The larger tests are more suitable to ensure correctness after the expected optimizations are performed. The experiments are repeated atleast 10 times, and the values above and below two times the standard deviation are removed before averaging the result to ensure the reproducibility of the result. The ratio of the two computed results is also provided to measure the improvements at a glance.

Chapter 6

Future Work

This section describes the possible future optimizations to the RNAfold program. As previously stated, an important objective of this thesis was to transform the codebase to make it suitable for future code optimizations while retaining the package's original structure. Detailed reasoning for compromises made and possible future optimization exploration are mentioned below:

6.1 Explore Parallelism

As seen in the third toy kernel, parallelism significantly improves code performance for realistic sizes. However, during implementation in RNAfold, it was noticed that the OpenMP commands could not provide a significant improvement. Using the Intel Advisor Toolkit revealed that the overhead of critical sections was slightly lower than the savings achieved by using multiple cores. This indicates a similar result as observed for the first Nussinov-like toy kernel. A proposed way to benefit from parallelism would be to explore efficient register tile sizes to overcome the cache limitations.

Another way to parallelize the code could be to opt for a completely different schedule. The latter option was not explored in the context of the Vienna Package. However, information about possible schedules could be derived from the first toy kernel and the use of the AlphaZ tool. Some of the implementations of the <d,j,k> tiled multicore schedules have been described in the papers by Gildemaster et al. [3]

6.2 Interior loops optimization

Interior loop computation is one of the most time-consuming functions in the RNAfold program. The current implementation of the internal loops performs an $O(N^4)$ operation. However, it is a common practice to limit the upper bounds for the outer two loops to

MAX_LOOPS=30. This effectively meant that a similar complexity was observed for problem sizes close to nt=900 for Interior loops and Multibranch loops. However, the Multibranch computation was considered for optimization as the package would be used for realistic applications and concatenated RNA strands.

The interior loops can be optimized by switching to an $O(N^3)$ algorithm described in the paper titled "Fast interior loop optimization" by Lyngso and Zuker. [12] It describes the assumption that the destabilizing energy depends on the base pairs' size, asymmetry, and stacking energies with respect to the nearest unpaired bases. The paper describes an example that lets the algorithm create one internal loop instead of two bulges, thereby increasing the stability in a significant way.

6.3 Multi-Strand computation

To limit the scope of this thesis, it was decided to focus on single-strand RNAfold optimizations. Limiting the number of strands to 1 eliminates two control flows for updating fms3 and fms5 matrices. This limit simplifies the process followed to perform the tiling transformation legally. Adding multi-strand computations to the codebase will not require additional memory as fms3 and fms5 are declared matrices.

The computation for fms5 involves obtaining all the points in the previous iteration of `i` once the iteration of `j` is completed. Since tiling requires us to start the computation of `i` before `j` is completed, an updated code for `update_fms5_arrays` is needed after the completion of `j` in each tile boundary. A similar dependency is seen in the case of fms3 arrays.

Chapter 7

Conclusion

Packages like ViennaRNA, widely used by synthetic biologists, have invested in manual vectorization to improve single-core performance. However, the gains observed are significantly lower than expected, highlighting the disconnect in optimization techniques used by the HPC and Synthetic Biology community. This thesis attempts to bridge this gap by producing incremental changes to the RNAfold program of ViennaRNA. Using the patch produced by this thesis, a $2\times$ speedup can be observed with just under 1200 lines of optimized code.

Optimizing RNAfold produced many software engineering challenges, which were simplified using toy kernels. These kernels closely mimicked the operations performed by the program and played a critical role in optimization exploration to create valuable insights for optimizations. Further, the optimization steps could be condensed to 5 transformations, some of which include multi-level loop tiling, permutation, unrolling, and vectorization. These optimizations helped achieve the performance in the L1 cache bandwidth for the computation to be optimized.

Other goals achieved through this thesis were to retain the existing code structure and make code changes accessible to other developers. A git repository was created for documentation, and an automated Python script was used to apply modifications automatically to the entire package, making the code more distributable.

Bibliography

[1] AlphaZ Wiki. https://www.cs.colostate.edu/AlphaZ/wiki/doku.php/#alphaz_wiki, October 2018.

[2] Cedric Andreolli. Find CPU & GPU performance headroom using Roofline Analysis – Intel. www.intel.com/content/www/us/en/developer/videos/cpu-gpu-performance-headroom-roofline-analysis.html.

[3] Gildemaster B., Rajopadhye S., Chitsaz H., and Abdo Z. A GPU accelerated RNA-RNA interaction program. *Masters thesis, Colorado State University*, 2021.

[4] Z Boston. How to transpose a 16x16 matrix using SIMD instructions? stackoverflow.com/questions/29519222/how-to-transpose-a-16x16-matrix-using-simd-instructions, 2023.

[5] Bevin Brett. Efficient use of Tiling – Intel. https://www.intel.com/content/www/us/en/developer/articles/technical/efficient-use-of-tiling.html, June 2016.

[6] Suzanne Clancy. Chemical structure of RNA – nature publishing group. https://www.nature.com/scitable/topicpage/chemical-structure-of-rna-348/, 2008. 7(1):60.

[7] Verifying the Correctness of Programs. www.cs.cornell.edu/courses/cs312/2006sp/lectures/lec10.html.

[8] Intel® Intrinsics Guide – Version 3.6.6 Intel. www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html, May 2023.

[9] W. B. Langdon and Ronny Lorenz. CUDA RNAfold. *Cold Spring Harbor Laboratory*, April 2018.

[10] Junjie Li, Sanjay Ranka, and Sartaj Sahni. Multicore and GPU algorithms for nussinov RNA folding. *BMC Bioinformatics*, 15(S8), July 2014.

[11] Ronny Lorenz, Stephan H Bernhart, Christian Höner zu Siederdissen, Hakim Tafer, Christoph Flamm, Peter F Stadler, and Ivo L Hofacker. ViennaRNA package 2.0. *Algorithms for Molecular Biology*, 6(1), November 2011.

[12] R B Lyngsø, M Zuker, and C N Pedersen. Fast evaluation of internal loops in RNA secondary structure prediction. *Bioinformatics*, 15(6):440–445, June 1999.

[13] Christophe Mauras. Alpha : un langage equationnel pour la conception et la programmation d'architectures paralleles synchrones. *PhD thesis, Rennes 1*, 1989.

[14] Chiranjeb Mondal and Sanjay Rajopadhye. Accelerating the BPMax algorithm for RNA-RNA interaction. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, June 2021.

[15] R Nussinov and A B Jacobson. Fast algorithm for predicting the secondary structure of single-stranded RNA. *Proceedings of the National Academy of Sciences*, 77(11):6309–6313, November 1980.

[16] G.M. Raskulinec and E Fiksman. SIMD Functions via OpenMP. *High Performance Parallelism Pearls*, 2015.

[17] Kengo Sato, Manato Akiyama, and Yasubumi Sakakibara. RNA secondary structure prediction using deep learning with thermodynamic integration. *Nature Communications*, 12(1), February 2021.

[18] Alexandra Marie Shinsel. Intel® Advisor Roofline – Intel. https://www.intel.com/content/www/us/en/developer/articles/guide/intel-advisor-roofline.html, April 2018.

[19] Scott K. Silverman. A forced march across an RNA folding landscape. *Chemistry & Biology*, 15(3):211–213, March 2008.

[20] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, March 1981.

[21] Douglas H. Turner and David H. Mathews. NNDB: the nearest neighbor parameter database for predicting stability of nucleic acid secondary structure. *Nucleic Acids Research*, 38(suppl_1):D280–D282, October 2009.

[22] Samuel Williams, Andrew Waterman, and David Patterson. Roofline. *Communications of the ACM*, 52(4):65–76, April 2009.

[23] M. Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing - Supercomputing '89*. ACM Press, 1989.

[24] David Wonnacott, Tian Jin, and Allison Lake. Automatic tiling of "mostly-tileable" loop nests. In *Fifth International Workshop on Polyhedral Compilation Techniques*, January 2015.

Appendix A

Github Repository

This section refers to the GitHub repository associated with the project. Please visit the repository URL below to access the latest version of the source code, documentation, and other related resources.

Repository Name: ViennaRNA_Thesis

GitHub URL: https://github.com/ViditSave/ViennaRNA_Thesis

Owner: ViditSave