THESIS

POLYHEDRAL OPTIMIZATIONS OF RNA-RNA INTERACTION COMPUTATIONS

Submitted by

Swetha Varadarajan

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 2017

Master's Committee:

Advisor: Sanjay Rajopadhye

Wim Bohm
Jesse Wilson

ABSTRACT

POLYHEDRAL OPTIMIZATIONS OF
RNA-RNA INTERACTION COMPUTATIONS

Studying RNA-RNA interaction has led to major successes in the treatment of some cancers, including colon, breast and pancreatic cancer by suppressing the gene expression involved in the development of these diseases. The problem with such programs is that they are computationally and memory intensive: $O(N^4)$ space and $O(N^6)$ time complexity. Moreover, the entire application is complicated and involves many mutually recursive data variables. We address the problem of speeding up a surrogate kernel (named OSPSQ) that captures the main dependence pattern found in two widely used RNA-RNA interaction applications- IRIS and piRNA.

The structure of the OSPSQ kernel perfectly fits the constraints of the polyhedral model, a well-developed technology for optimizing codes that belong to many specialized domains. However, the current state-of-the-art automatic polyhedral tools do not significantly improve the performance of the baseline implementation of OSPSQ.

With simple techniques like loop permutation and skewing, we achieve an average of 17x sequential and 31x parallel speedup on a standard modern multi-core platform (Intel Broadwell, E5-1650v4). This performance represents 75% and 88% of attainable single-core and multi-core L1 bandwidth. For further performance improvement, we describe how to tile all six dimensions and also formulate the associated memory trade-off.

In the future, we plan to implement these tiling strategies, explore the performance of the code for various tile sizes and optimize the whole piRNA application.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Ribonucleic acid (RNA) is a single stranded nucleic acid. In cancerous cells, micro RNA molecules (miRNA) often bind to messenger RNA (mRNA) molecules and change their functions. Knowledge of this RNA-RNA interaction has led to major successes in the treatment of some cancers, including colon, breast and pancreatic by suppressing the gene expression involved in the development of these diseases [1–3]. We consider optimizing two applications that predict RNA-RNA interaction structure and associated properties- IRIS [4] and piRNA [5]. Both these applications involve calculations on single RNA folding as well as the interaction between RNA molecules.

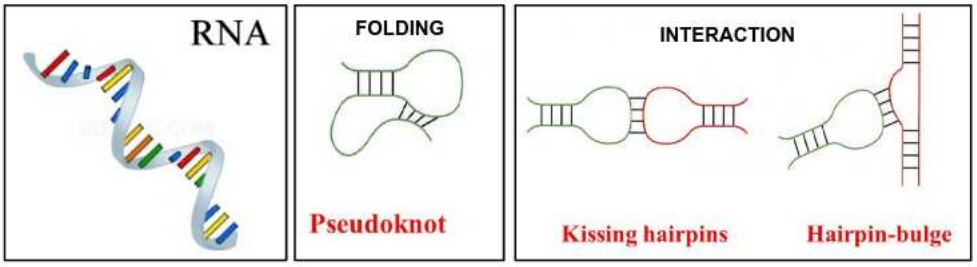


Figure 1.1: Example of a single RNA molecule, its folding and interaction loops formed between two RNA molecules

The IRIS package uses dynamic programming to find secondary structures that have the maximum number of base pairs when two RNA sequences interact. piRNA is a bio-informatics application that computes the partition function of two interacting RNA sequences using a similar algorithm. The partition function is a weighted sum over the set of all possible secondary structures (S) given by the equation 1.1 where R is the universal gas constant and T is the temperature.

$$Q(T) = \sum_{s \in S} e^{-G_s/RT} \tag{1.1}$$

This partition function is a scalar value and can be used to compute several other properties like equilibrium concentration and melting temperature of interacting nucleic acids. Both the applications have the time complexity of $O(N^3M^3)$ and space complexity of $O(N^2M^2)$ where N and M are lengths of the input sequences.

On examining these two applications, we find that the dominant computations fit the polyhedral model [6–8]. This model is widely accepted as a powerful tool for optimizing a well defined, limited, yet widely prevalent class of computational patterns that occur in many applications in the form of nested loops. There are several frameworks, some integrated into compilers, and some stand-alone user-guided tools that utilize these techniques for (semi) automatic compilation of codes to benefit from locality and parallelization.

piRNA is written in C++. Partition function computations contribute to 96% of the execution time and fit the polyhedral model. Although the authors implemented a simple parallelization strategy, we will see that it can be significantly improved. The parallelization of piRNA is not well suited for modern computers that have sophisticated memory hierarchy and SIMD features. The entire application is complicated and involves many mutually recursive data variables. It is difficult to apply automatic polyhedral tools on this whole application due to these complicated code structure. It is also known that polyhedral techniques are successful in optimizing computations found in small to medium benchmark like Polybench [9], relatively few attempts tackle large applications [10]. So, we decided to analyze a surrogate kernel named Optimal String Parenthesization SQuared (OSPSQ) that captures the main dependence pattern. Chapter 2 explains the complicated structure of piRNA application and details of the gap between piRNA and OSPSQ. The IRIS application also has OSPSQ-like dependence structure. In OSPSQ, the operation is multiply-add whereas, with IRIS, it is max-plus. Although the operations are different, the dependence pattern is same, and optimizations for OSPSQ can be applied to IRIS without any modifications.

In this work, we focus on optimizing OSPSQ computation. Although this kernel is small when compared to the entire piRNA application, it is much more complex than kernels typically handled by most polyhedral tools, since it has $O(N^4)$ space and $O(N^6)$ time complexity. Most of the

Polybench kernels has $O(N^2)$ memory and $O(N^3)$ time complexity. It has only two kernels that have $O(N^3)$ space and $O(N^4)$ time. Even the highly optimized BLAS library has routines only for $O(N^3)$ time and $O(N^2)$ space computations.

1.1 CONTRIBUTIONS

- After identifying that the control and dependence patterns of the critical part of both the applications can be succinctly described by a simple surrogate kernel that we call OSPSQ, we achieved 17× sequential and 31× parallel speed-up on Intel Broadwell.
- This performance corresponds to 75% and 88% of attainable single-core and multi-core L1 bandwidth.
- For further improving performance, we show how to tile all six dimensions and also formulate the associated memory trade-off.

1.2 RELATED WORK

Although we use important principles of the polyhedral model in this thesis, we believe that the readers can understand the concepts without the mathematical formulations of the polyhedral model. We provide the necessary background on polyhedral tools and concepts as and when required throughout the thesis.

Different algorithms have been proposed to predict RNA-RNA interaction structure. However, most of them suffer from high computational time. There are several algorithms and techniques to mitigate this problem but to our knowledge, none of these aims at improving the performance based on the dependence structure between the variables in the code. For interested readers, following is a review of optimization techniques that are entirely orthogonal to our strategy.

Syed Ali Ahmed and Saad Mneimneh introduce an approximation algorithm [11], where an RNA-RNA interaction graph is created in which every edge represents a possible bond in or between two RNA sequences. A set of edges is found to maximize the number of bonds. Fenix et

al. [12] introduce a statistical sampling algorithm based on some modifications to the grammars. It calculates the interaction probabilities for any given single region on RNA. Alkan et al. [13] develop fast algorithms to minimize the free-energy between RNA molecules.

RactIP [14] predicts RNA-RNA interaction using integer programming. It uses the approximate information of the internal and external base pairing probabilities of joint structures as an objective function of integer programming. The authors claim that "RactIP is the fastest method for predicting both internal structures and binding sites simultaneously on condition of the comprehensive class of interactions." But, their computational complexity remains the same.

Salari et al. [15] propose a fast interaction prediction broken down into two steps. In step 1, they focus on calculating the probability of at most two regions remaining unpaired within a given RNA sequence of length n; their method calculates the probability of any pair of regions of length $\leq$ w each in $O(n^4w)$ time and $O(n^2)$ space. In step 2, for two RNA sequences, their method computes the most probable non-conflicting matching of accessible regions in $O(n^2.w^4 + n^3/w^3)$ time and $O(w^4 + n^2/w^4)$ space complexity. They achieve this based on the observation that independent secondary structure is preserved even after the joint structure is formed with another RNA. In a follow-up work [16], they apply sparsification techniques to reduce time and space complexity to $O(n^4\Psi(n))$ and $O(n^2\Psi(n) + n^3)$ respectively for some function $\Psi(n)$. This function turns out to have small values for the range of n encountered in practice. Using the polymer-zeta property, they prove that $\Psi(n) = O(n)$. So, they achieve linear time and space improvement.

Hamidreza et al. [17] introduce a fast algorithm, based on sparse folding, to calculate an upper bound on the partition function. The space complexity is $O(m^2 + n^2 + MFES(m, n))$, and the time complexity of $O(MFE(m, n)l)$ for RNA-RNA interaction. Here, MFES and MFE are the memory space and the running time of minimum free energy prediction and $l \leq n(ln + m)$ is a sequence-dependent parameter.

1.3 THESIS STRUCTURE

Chapter 2 explains the structure of piRNA application and details of the gap between piRNA and OSPSQ. It also motivates the reason for why we chose to manually optimize OSPSQ computation amidst the existence of powerful automatic tools. Chapter 3 explains the implemented optimization strategies. In Chapter 4 we present the experimental evaluation. In Chapter 5 we introduce the six-dimensional tiling technique and the associated memory trade-off derivation. Chapter 6 presents conclusion and pointers to future work.

Chapter 2

OSPSQ CREATION AND MOTIVATION

piRNA is written in C++. The entire application is divided into several files of about 8000 lines of code. Out of this, partition function calculations are about 1000 lines and contribute to 96% of execution time. It includes code for calculating partition function on a single sequence as well on the interaction of two RNA sequences. The time and space complexity to calculate single sequence partition function tables are $O(N^4)$ and $O(N^2)$ respectively. There are techniques to optimize these calculations [18–22]. The asymptotic time complexity of interaction (joint) partition function $O(N^6)$ is greater than this So, we are focused only on the optimization of interaction partition function computations.

There are about 96 four-dimensional tables used to calculate the joint partition function. These tables depend on the single partition function results as well as among itself. The listing 2.1 shows the code structure for the calculation of all the 96 tables.

Listing 2.1: Code structure of piRNA joint partition function calculations

```
1 for (d1 = 1; d1 <= N − 1; d1++) {
2       for (d2 = 1; d2 <= M − 1 ; d2++){
3             for (i = 0; i <= N− d1 − 1; i++){
4                   for (k = 0; k <= M− d2 − 1 ; k++){
5                         j=i+d1; l=k+d2;
6                         c[i][j][k][l] = quadratic_accumulation_function_call(i,j,k,l);
7 }}}}
```

Here, i, j, k and l are the four data dimensions. To calculate one single variable (c[i][j][k][l]), a quadratic (double-loop) accumulation of values are involved. These quadratic accumulations are structured into four different function calls depending on the type of computation. Since the asymptotic complexity of all the function calls is same, we can deduce the dominant computation in the code just by having the count of these function calls.

From table 2.1, we can see that “product" function call is the dominant computation. We further explain the computation of each of these four calls next. In the equations below, the parameter

Table 2.1: Profiling of quadratic accumulation function calls in piRNA

S.No.	Function call	Percentage of total calls	Percentage of execution time
1	product	67.9%	68%
2	lsemiproduct	9.5%	1.8%
3	rsemiproduct	10.7%	20.8%
4	capproduct	11.9%	2%

names in function calls and the variable on the left hand side are place-holders, and come from the set of 96 tables (variable names) in the program. d and e denote the accumulation dimensions.

1. **product**: The input to this function call is a and b which are four dimensional tables that belong to the same 96 set of tables. The output of this function call is one scalar variable. Equation 2.1 represents the computation.

$$c[i,j,k,l] = product(a,b) = \sum_{d=i}^{j-1} \sum_{e=k}^{l-1} a[i,d,j,e] * b[d+1,j,e+1,l] \tag{2.1}$$

2. **rsemiproduct**: The input to this function call is a1, a2 and b. b is one of the 96 four dimensional tables. a1 and a2 are two dimensional tables from single partition function calculations. The output of rsemiproduct function call is one scalar variable. This function call can be made to have a structure similar to product by performing outer product of a1 and a2 tables. Equation 2.2 gives the equation of this computation.

$$c[i,j,k,l] = rsemiproduct(a1,a2,b) = product(outerproduct(a1,a2),b) \tag{2.2}$$

3. **lsemiproduct**: This computation is similar to rsemiproduct. The input to this function call is a, b1 and b2. a is one of the 96 four dimensional tables. b1 and b2 are two dimensional tables from single partition function calculations. The output of lsemiproduct function call is one scalar variable. This function call can be made to have a structure similar to product by performing outer product of b1 and b2 tables.

$$c[i,j,k,l] = lsemiproduct(a, b1, b2) = product(a, outerproduct(b1, b2)) \tag{2.3}$$

4. **capproduct**: The input to this function call is a and b. Here, b is the only one which belongs to the 96 four dimensional table set. The table $a\prime$ is also four dimensional but has different data layout and is calculated from the results of single partition function calculations. There are two variations of this function depending on the limits of the accumulation loop dimensions d and e. The output of this function call is one scalar variable. Equation 2.4 gives the equation of the computation.

$$\begin{aligned} c[i,j,k,l] &= capproduct1(a\prime, b) = \sum_{d=i}^{j-2} \sum_{e=d+2}^{j} a\prime[d+1, e-1, k, l] * b[i,j,d,e] \\ c[i,j,k,l] &= capproduct2(a\prime, b) = \sum_{d=k}^{l-2} \sum_{e=d+2}^{l} a\prime[k, l, d+1, e-1] * b[i,j,d,e] \end{aligned} \tag{2.4}$$

From the above discussion, we see that product kind of dependencies contribute to 90.6% (combining product, lsemiproduct and rsemiproduct) of the execution time and 88.1% of total accumulation loop function calls. Since the inter-dependence of these 96 variables are complicated, in this work, we address the problem of speeding up a surrogate kernel (named OSPSQ) that captures the “product" dependence pattern.

2.1 PLUTO PERFORMANCE ON OSPSQ

We used PLUTO [23], one of the current state-of-the-art automatic polyhedral tools to optimize OSPSQ kernel. This tool transforms C programs from source to source for coarse-grained parallelism and data locality simultaneously. We compared the performance of PLUTO on baseline code. The details of the baseline code are provided in Chapter 3. The target mapping that PLUTO produces for OSPSQ-base is $(i,j,k,l,d,e \rightarrow l-k, l, j-i, j, d, e)$ and it tiles the outer four loops with default tile size (32). Please refer section 2.2 for the contextual meaning of target mapping and the notation.

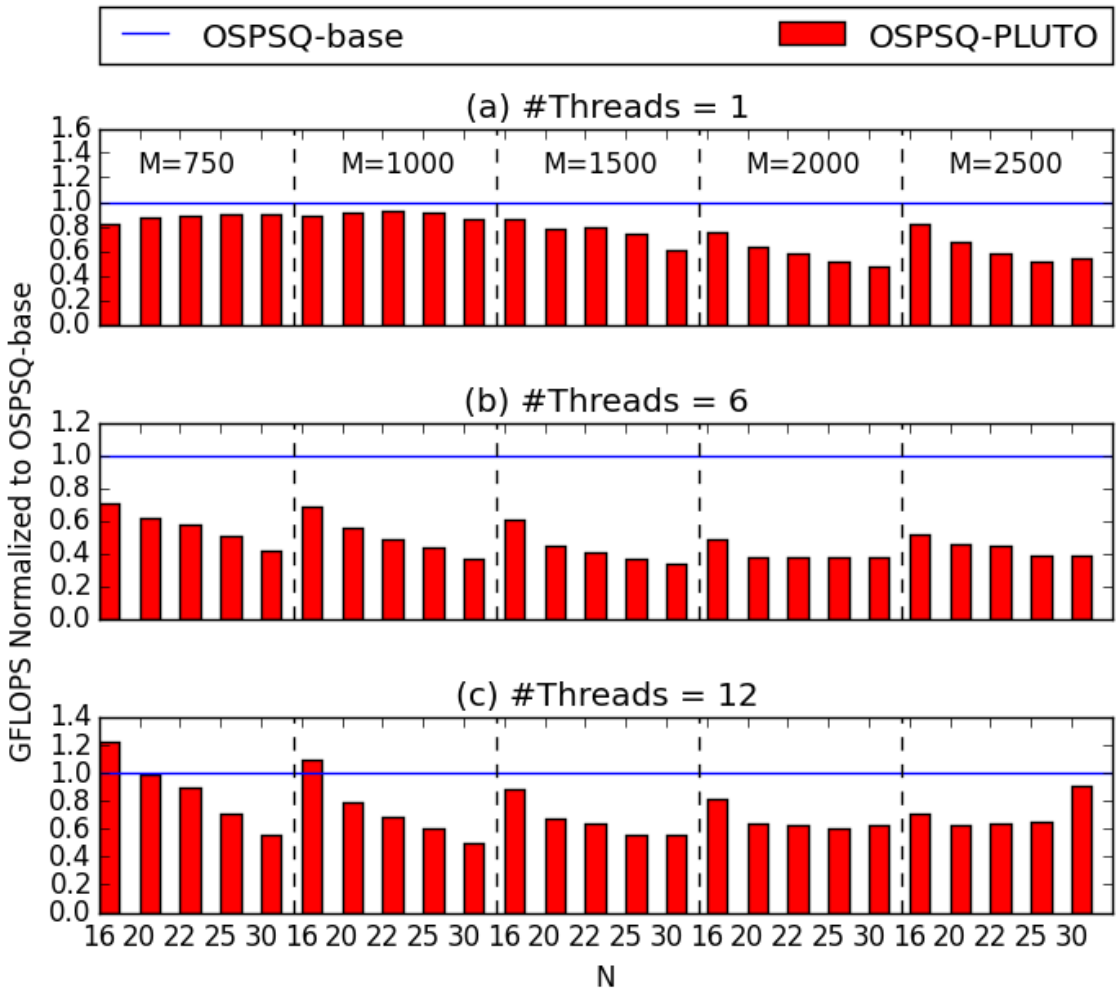


Figure 2.1: Pluto versus OSPSQ-base: Performance Comparison on single core, six core and twelve threads

As seen in figure 2.1, PLUTO's performance on average is 18% poor on single-core, 51% poor on six-core, 26% poor on twelve-core when compared to the baseline code. It has improved performance on twelve-core for two problem instances. Here, N is the length of miRNA and M is the length of mRNA.

- 22% improvement when N is sixteen and M is 750
- 9% improvement when N is sixteen and M is 1000

But these are 20% and 25% less than the best performance that the baseline code achieves respectively. It can be seen that although the structure of the OSPSQ kernel perfectly fits the

constraints of the polyhedral model, the current state-of-the-art automatic polyhedral tool PLUTO does not significantly improve the performance of the baseline implementation of OSPSQ.

2.2 ALPHAZ : BACKGROUND

Since the automatic polyhedral tool didn't improve OSPSQ performance, we use the manual polyhedral toolset- AlphaZ [24] to generate the optimized C code. This section provides a brief background on this tool-set. There are two important parts in generating this optimized code.

1. Input specification: The input âĂIJprogramâĂİ consists of one or more mathematical equations (using the Alpha/Alphabets language) that specify just what needs to be computed. It is also called as an Alpha specification.

2. Command script: To produce a (conventional/imperative) program that implements the input specification, one needs to specify a schedule (when), a processor allocation (who), and a memory allocation (where to store). These are specified as commands using a script which has three main parts.

   (a) *Transformations:* These commands are used to perform transformations. They produce a different Alpha specification. It is the responsibility of the user to make sure that the transformations are valid and produces the required results.

   (b) *Target Mapping:* These commands are used to specify the execution order of the program. This also needs the user's effort to specify a valid map. Memory mapping allows the variables to use the same memory space. Space-Time map specifies the order in which the iteration points are executed for each variable. An alternate way to call the space-time map is scheduled as it schedules the execution order of the iteration points.

   (c) *Code-generation:* These commands are used to transform alphabets to other high-level languages.

We specify the transformations and mappings using a notation:
$(ListOfIndices \rightarrow ListOfIndexExpressions)$. Consider the following two examples:

1. A target mapping $(i, j \rightarrow j - i, i)$. This map says that the initial iteration domain of the variable is two dimensional represented by indices i and j. The points in this iteration space are visited in the order given by the right-hand side of the arrow.

2. A memory map $(i, j, k \rightarrow i)$. This map says that the initial memory allocation (in the alpha specification) of the variable is three dimensional. But, in the generated code it is one dimensional. So, there is a quadratic memory saving.

Chapter 3

OSPSQ DEPENDENCE STRUCTURE AND OPTIMIZATION

Now that we have identified OSPSQ as the surrogate kernel to optimize, in this chapter we discuss the OSPSQ dependence structure and the applied optimizations.

3.1 OSPSQ DEPENDENCE STRUCTURE

The recursion equation of OSPSQ is given by equation 3.1.

$$T[i,j,k,l] = \sum_{d=i,e=k}^{d=j-1,e=l-1} T[i,d,k,e] * T[d+1,j,e+1,l] \tag{3.1}$$

Each value T[i,j,k,l] is the double summation of a quadratic number of values. Figure 3.1 shows the four dimensional variable T that is defined, as well as the dependences in the computations.

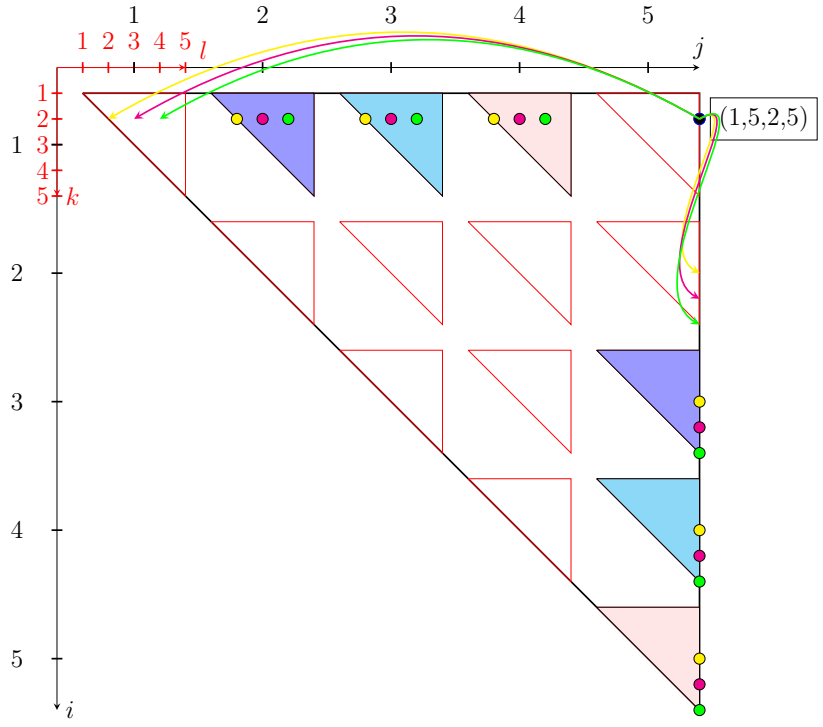


Figure 3.1: OSPSQ Dependence pattern

Black axes represent the outer two dimension and each point in it is a 2D triangle. The black axes range over the outer two indices, i, and j. The red axes rage over the inner dimension k and l. The bounds $i < j < k < l$ imply that the data is an upper triangular array of smaller triangles.

For example, to calculate $[1, 5, 2, 5]$, the magenta, yellow and green points in the west white triangle is multiplied to the corresponding color-coded locations in the south white triangle. This pattern repeats for blue, cyan and pink triangles. These multiplied values are reduced using addition operator.

This dependence pattern can be imagined as a multi-dimensional extrapolation of Optimal String Parenthesization (OSP) [25]. But, OSP has the two-dimensional data structure. So, the bounds are just $i < j$ implying that the data is an upper triangular array of points. In Chapter 5, we give a background on OSP dependence structure and tiling problem to explain the tiling technique of OSPSQ.

3.2 OSPSQ BASELINE CODE

The starting point of all optimization is an Alpha specification. We chose a target mapping $(i, j, k, l, d, e \rightarrow j - i, l - k, i, k, d, e)$ that matches piRNA code. We call this version as *OSPSQ-base*.

From the OSPSQ equation, for every iteration of the inner-most loop *e*, there are three reads and one write of data. Since e is the innermost loop (i.e., rightmost index in the target mapping) in the generated code, the read of $T[i, j, k, l]$ is independent of *e*, and this variable can be assumed to be in the register. So, we can say that for every two reads of data, two operations are performed. The two operations (one multiplication and one addition) are on double precision floating point numbers (8 bytes each). So, the Arithmetic Intensity (AI) would be,

$$AI_{OSPSQ} = 2/(3 * 8) = 1/12 \tag{3.2}$$

For Intel Broadwell machine e5-1650 v4, the stream triad [26] benchmark on single core gives the peak DRAM bandwidth as 17.8 GB/s. The peak single-core performance is 57.6 GFLOPS

since each core on the 3.6GHz processor can issue fused multiply-add (two ops) per clock cycle on two sets of four-way double precision vector units (i.e., 16 ops per cycle). The inflection point of the roofline model graph [27], is at 3.24 so the baseline code is memory bound.

3.3 OSPSQ OPTIMIZATION

From the ICC vectorization guide, the necessary and sufficient conditions for an affine program to be vector friendly are,

1. No dependence in the inner-most loop nest.

2. Constant stride access (preferably ± 1) in the inner-most loop nest.

The innermost loop in OSPSQ baseline code is,

Listing 3.1: Innermost loop of OSPSQ: baseline

```
1  for(e=k;e<l;e++)
2  T[i,j,k,l] += T[i,d,k,e] * T[d+1,j,e+1,l]
```

One iteration (read of T[i,j,k,l]) of the inner-most loop depends on its previous iteration (write of T[i,j,k,l]) which creates a bubble in the pipeline and thereby decreases performance. Loop permutation is one way to work around this issue. Loops have to be permuted such that there is no forward dependence (read-after-write) present in the innermost loop and resulting transformation is legal. In our code, making either of the accumulation loop dimensions, *d,e* as the inner-most loop will create a forward dependence. So, we have a choice of having one of *j-i, l-k, i, k* as the inner-most loop.

In table 3.1, stride length refers to the difference in length between the reads of two successive memory address if that particular dimension is made as the inner-most loop. It can be seen that when $l - k$ is the innermost loop, we achieve unit stride and hence we have an improved target mapping:$(i, j, k, l, d, e \rightarrow j - i, i, k, d, e, l - k)$. We call this version as *OSPSQ-vect*.

The baseline schedule traverses along diagonals. In other words, it is a column-major access. In order to benefit from cache behaviour, we move to row-major sequential schedule $(i, j, k, l, d, e \rightarrow$

Table 3.1: Stride length for the choice of inner-most loop dimension

Inner-most loop dimension	Stride length
j-i	$O(M^2)$
l-k	1
i	$O(M^2 * N)$
k	$O(M)$

$-i, j, d, -k, e, l)$. We call this version as *OSPSQ-best-row*. There are several points to note in the schedule that we chose.

- Memory is accessed row-wise but in reversed order for the outer-most loop (i) to maintain legality.

- From figure 3.1, we call the triangle formed by outer two dimensions as an outer triangle and the triangle formed by inner two dimensions as smaller triangles. Memory accessed within the small triangle can be in forward k or reverse $-k$ row-direction. Since the outer triangle is accessed in reverse order, it makes sense to access the lower triangle also in reverse order so that the successive memory accessed is contiguous.

- We have l as the inner-most loop to benefit from vectorization.

- The accumulation loop d is exchanged with the k loop. This is also because of the stride length argument. The stride length for d loop can be either $O(M^2)$ or $O(M^2N)$ depending on which data is accessed. But, the stride length of k is $O(M)$ which is at least an order of magnitude less than d loop. So, this loop exchange ensures better utilization of bandwidth.

- The target size of the application is such that outer dimension is small (20-30) and inner dimension is around 1000. For instance, if we chose the outer dimension range as 30 and inner dimension range as 1000, from figure 3.1, we can see that there will be $30*30/2 = 450$ smaller triangles of dimension $1000*1000$. So having the d loop on top of k, l, e loops, makes us view the computation as a series of 450 two-dimensional OSP problems. The data foot-

print of the entire problem would be $O(N^2 * M^2)$. But, for one iteration of the k loop, the amount of memory needed is $2 * M + M^2/2$. We can conveniently fit the data in a 15MB LLC until when $M = 1936$.

As a fair comparison, we experiment on a code with $(i, j, k, l, d, e \rightarrow j - i, i, d, k, e, l - k)$ schedule that uses the baseline diagonal-traversal, improved vectorization and also the benefits of interchanging d and k loops. We call this version as *OSPSQ-best-col*.

We chose two different parallelization strategies. *Fine-grain parallelism:* Here, outer loops traversing the collection of small triangles are executed sequentially, and rows of the inner triangles are executed in parallel. *Coarse-grain parallelism:* Here, all small triangles along an outer triangle's diagonal are computed in parallel by different threads.

Chapter 4

EXPERIMENTAL EVALUATION AND RESULTS

The machine used for the experiments is Intel Broad-well: e5-1650v4 and its specifications are shown in Table 4.1 along with chosen compiler flags.

Table 4.1: Machine configuration

Architectural parameters	Broadwell
Processor	E5-1650v4
Number of cores	6
Base freq	3.60 GHz
Turbo boost freq	4 GHz
Instruction Set Features	AVX 2.0, FMA3
Max. DP performance	384 GFLOPS/s
L1 cache	32KB per core
L2 cache	256KB per core
L3 cache	15MB shared
RAM	2 * 8GB
Memory support	DDR4-2400
Max. Memory bandwidth	38.4 GB/s
Compiler	ICC 16.0.2 20160204
Compiler Flags	O3, xhost, ipo, fma, qopenmp

Table 4.2: Test cases: Sequential OSPSQ

S.No.	Test Case	Target Mapping
1	OSPSQ-base	$(i, j, k, l, d, e \rightarrow j - i, l - k, i, k, d, e)$
2	OSPSQ-vect	$(i, j, k, l, d, e \rightarrow j - i, i, k, d, e, l - k)$
3	OSPSQ-best-col	$(i, j, k, l, d, e \rightarrow j - i, i, d, k, e, l - k)$
4	OSPSQ-best-row	$(i, j, k, l, d, e \rightarrow -i, j, d, -k, e, l)$

Table 4.3: Test cases: Parallel OSPSQ

S.No.	Test Case	Parallel loop dimension
1	OSPSQ-base-par	III in OSPSQ-base
2	OSPSQ-best-row-par	IV in OSPSQ-best-row
3	OSPSQ-best-col-coarse	II in OSPSQ-best-col
4	OSPSQ-best-col-fine	IV in OSPSQ-best-col

Based on different Target Mapping, we have four test cases for sequential and four parallel version. Target mapping of these versions are provided in Table 4.2 and 4.3. The memory mapping for these test cases are $(i, j, k, l, d, e \rightarrow i, j, k, l)$.

piRNA is often used to investigate interactions between miRNA and mRNA, and the typical lengths of these sequences are 20-25 nucleotides and 1000-300. There are 25 data-points corresponding to all possible combinations of five values of M (750, 1000, 1500, 2000, 2500) and N (16, 20, 22, 25, 30) each.

4.1 SINGLE-CORE PERFORMANCE

Figure 4.1 illustrates the sequential results. The baseline for all speed-up data is OSPSQ-base version. In the graph, all the versions are normalized to the base version. We use the arithmetic mean for summarizing the costs, harmonic mean for summarizing the rates and geometric mean for summarizing the ratios [28].

The average speed-up of OSPSQ-vect version is 7.4×. This speed-up is because the vector length is four for doubles and with two FMA units, we can achieve up to 8× speed-up.

Average speed-up of OSPSQ-best-col and OSPSQ-best-row across 25 data-points are 16.01× and 16.09× respectively. The maximum speed-up we obtain is 21.52× for OSPSQ-best-row version when N=30 and M=1500. There is not much performance difference between OSPSQ-best-row and OSPSQ-best-col version which is not surprising as row and column-major access for outer triangular matrix do not significantly affect performance. In either schedule, the traversal within a single inner triangle is the same and having an efficient schedule for this part of the kernel is the

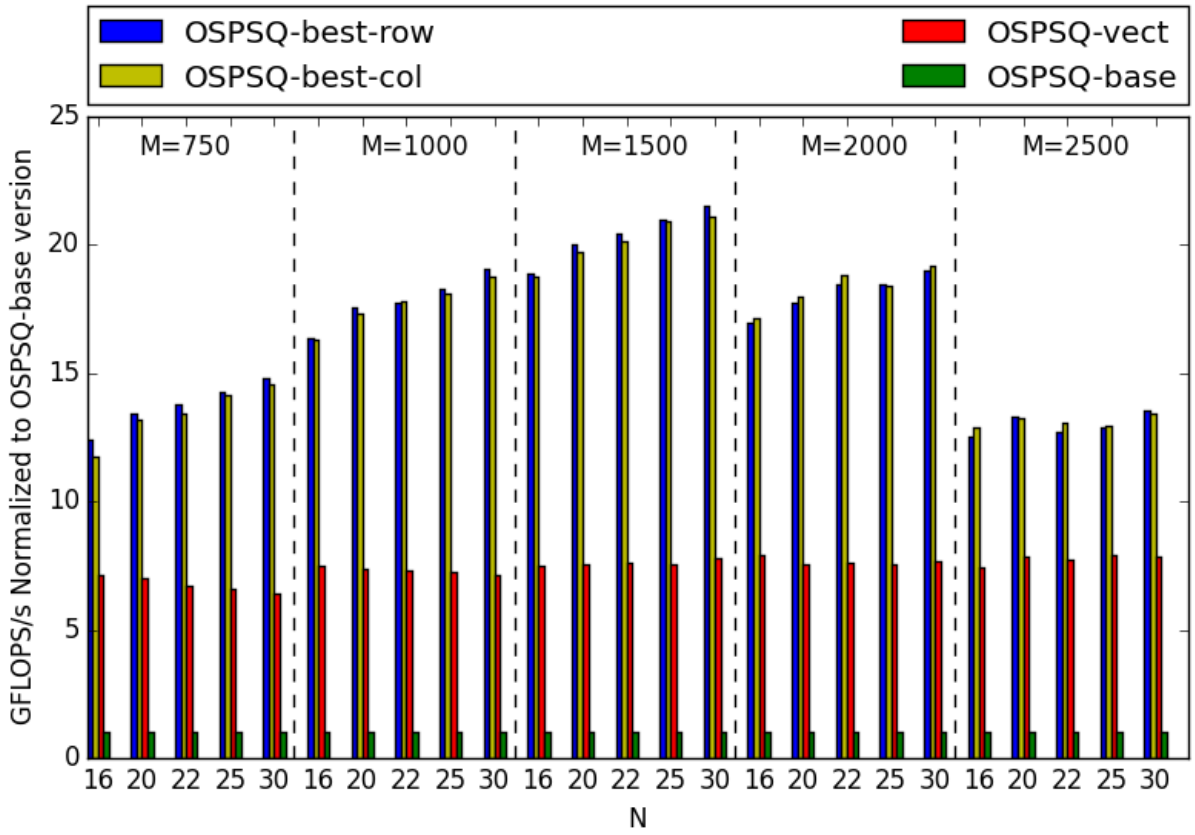


Figure 4.1: Performance comparison of OSPSQ sequential test cases normalized to the baseline on e5-1650v4. Vectorization test case shows an average of 7 fold improvement and the two locality test cases shows added two-three fold improvement.

key determinant of the overall performance. In addition to the 8× speed-up from the vectorization benefits, these two versions get an extra two to three-fold speed-up because of locality.

For a given N, when we increase M, the performance increases up to M=1500 and drops after that for OSPSQ-best-row and OSPSQ-best-col versions. This variation is because when M is around 1930, the $2 * M + M^2/2$ capacity exceeds 15MB LLC and we get the performance loss due to DRAM traffic. With OSPSQ-base version, an average performance loss of the geometric mean of the flops/s rate observed is 9% with increasing value of M for a given N. Whereas with OSPSQ-vect version, performance decrease of 9.5% is observed when N is 16 and 6.8% when N is 22. When N is 25, performance degradation drops to 5.2%. On average, we observe a 5.5% performance loss with increasing value of M for a given N.

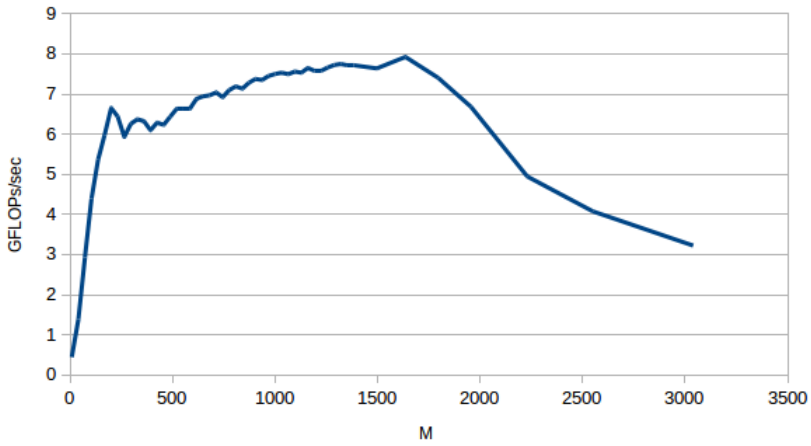


Figure 4.2: GFLOPS versus M when N=30 on OSPSQ-best-col version: Performance drops when M is around 1650 as the memory foot-print exceeds LLC

Across the 25 data-points for OSPSQ-best-row version, we obtained maximum speed-up when N is 30. So keeping N as 30, we vary M between two and 3200 (maximum malloc capacity) to observe the maximum performance that we can obtain on a single-core. The total number of data-points sampled was 1610. The interval between successive data points was not constant. From figure 4.2, it can be seen that maximum performance (7.92 GFLOPS) is achieved when M=1636 which is less than 1930 as per the previous discussion. This performance makes sense because the AlphaZ generated code has a rectangular domain and so, having a strict $M^2/2$ for memory calculation need not be true.

For an AI of 1/12, the corresponding bandwidth would be 95.04 GBytes/sec. This bandwidth is 75% of DAXPY L1 bandwidth measured using LMBench suite [29]. We compare the performance to DAXPY since the innermost loop of OSPSQ is similar to DAXPY kernel. So, we conclude that we have achieved the best performance for the given bandwidth and there is a need to tile for further improvements.

4.2 MULTI-CORE PERFORMANCE

We have four test cases for comparing parallel performance. OSPSQ-base-par is the parallelization found in piRNA application. We apply the two parallelization strategy to OSPSQ-best-col

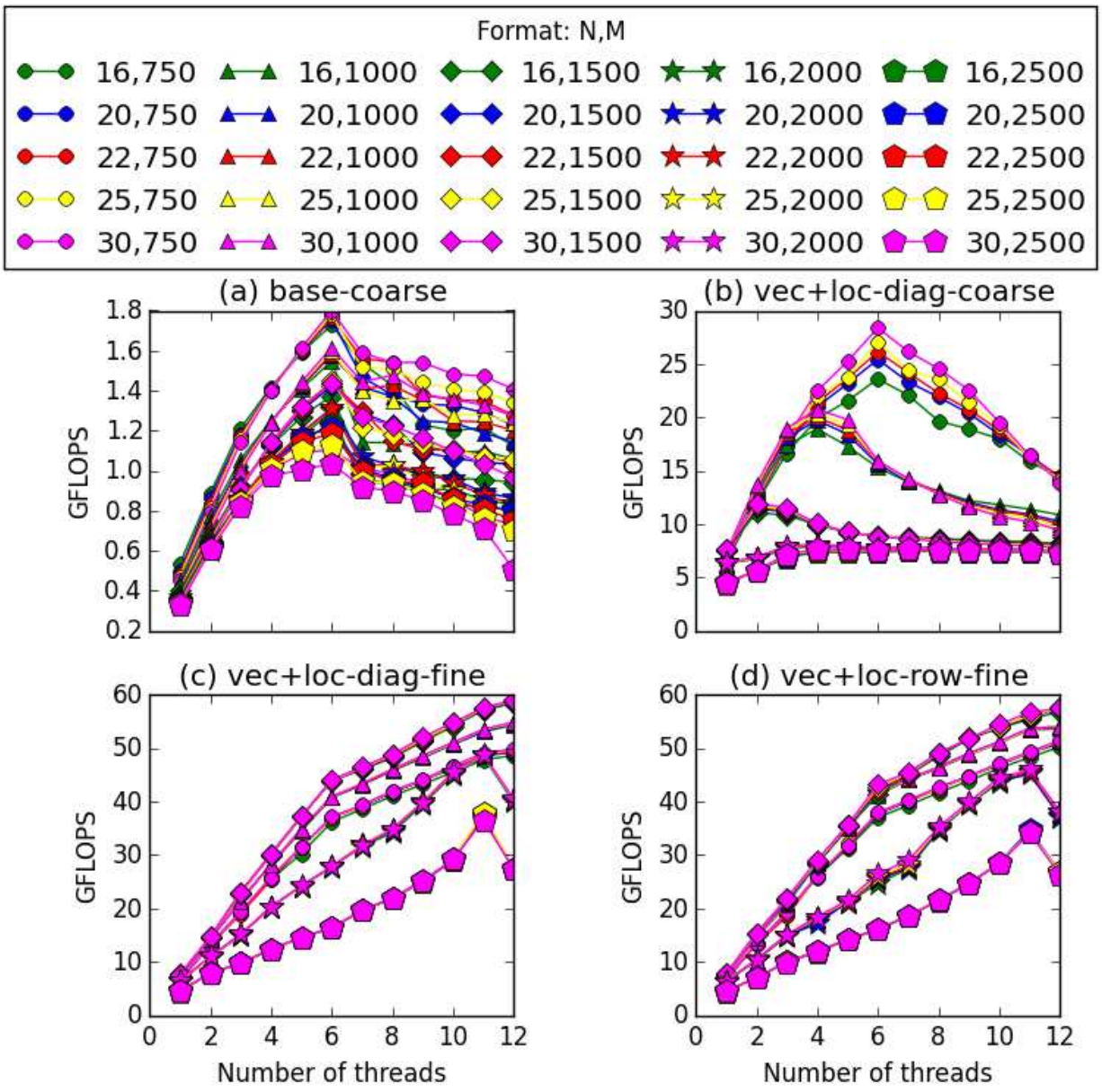


Figure 4.3: Performance comparison of OSPSQ parallel test cases on e5-1650v4 showing effects of hyper-threading, memory requirement and scaling, performance variation with N and M

version whereas for OSPSQ-best-row version we could use only the fine-grain parallelization. For the fine-grain parallelization versions, we observe that dynamic scheduling improves performance on average by 2.5$\times$ whereas it does not improve for coarse-grain version. We do not parallelize the vectorization only version as the OSPSQ-best-col, and OSPSQ-best-row versions carry the benefits this version.

Figure 4.3, shows the parallel performance of the four test cases across twelve threads. Although there are 25$\times$12 values plotted for each version, we can see that there is an overlap of the plots giving an illusion that there are only 5$times$12 data points. In other words, we can see five band formation corresponding to five different values of M. Following the color and symbol coded legends in the figure helps in better understanding of this band formation. So, parallel performance is not very sensitive to the variation of N. But, the bands and scaling across threads is not uniform across the versions.

1. OSPSQ-best-row-par and OSPSQ-best-col-fine: The top-most band is when M is 1500 followed by 1000, 750, 2000 and 2500. This is because of LLC as discussed for sequential performance. It can also be seen that a perfect scaling across threads is obtained even with hyper-threading. There is a performance drop at twelve threads when M=2k and 2.5k. On average, hyper-threading on six cores (with twelve threads) seems to improve performance by 34% when M $\leq$ 1500. When M is 2k, it is 45% and 67% for 2.5k. Since every row is executed in parallel, they share common read accesses from the south triangles. Hyper-threading can overlap the read traffic of multiple threads and thereby improve performance. The best performance that we obtain on six cores without hyper-threading is about 44 GFLOPS. For an AI of 1/12, this corresponds to 528 GBytes/sec which is about 66% of DAXPY L1 bandwidth measured using LMBench suite. With hyper-threading, the best performance is about 59 GFLOPS, and the corresponding bandwidth is 88% of DAXPY L1 bandwidth.

2. OSPSQ-best-col-coarse: Here, the band formation is in the increasing order of M. Top-most is 750, and the bottommost is 2.5k. The scaling drops when the LLC is not big enough to fit the data required for all the cores. So, the capacity limit would be $P * (2M + M^2/2)$

where P is some cores. It is also interesting to note that hyper-threading severely degrades performance for this version. This performance degradation is because LLC is not able to handle several DRAM traffic demands and results in unnecessary stalls.

3. OSPSQ-base-par: The band formation is similar to OSPSQ-best-col-coarse. But, scaling is not dependent on the LLC. It scales on average up to 3.6x on 6-cores. Hyper-threading slightly degrades performance beyond that.

Chapter 5

OSPSQ : TILING

In the previous chapter, we saw that to improve performance; there is a need to tile the computations. In this chapter, we discuss the tiling challenges and its solutions.

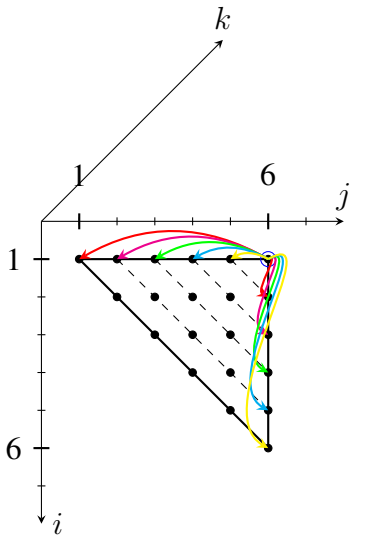
5.1 MIDDLE SERIALIZATION : BACKGROUND

In chapter 3, we mentioned that OSPSQ is a multi-dimensional extrapolation of OSP computation. To explain the problem associated with tiling all six dimensions in OSPQSQ, we present the recurrence equation, dependence pattern and the tiling problem of 2D-OSP here. The recurrence equation for OSP is,

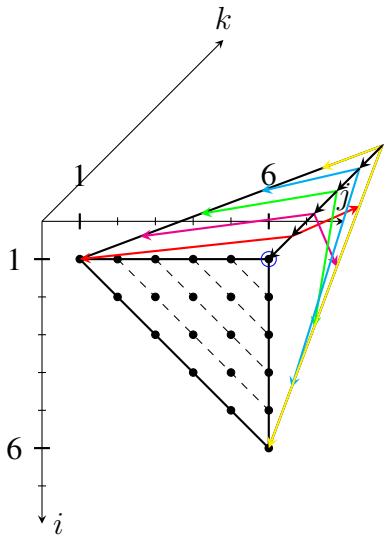
$$T[i,j] = \begin{cases} 0, & \text{if } i = j. \\ \min\limits_{k=i}^{j-1} T[i,k] + T[k+1,j] + (P[i-1] * P[k] * P[j]), & \text{if } j > i. \end{cases} \tag{5.1}$$

where P is the input. Considering only the table (T) dependencies, we see that an iteration point (1,6) in figure 5.1a depends on all its west and south neighbors. The arrowhead says that the particular point must be computed before the computation of the point at which the tail of the arrow is located. So, to compute (1,6), a total of 10 points must be computed before. The color-coded edges represent the pair of data points that (1,6) depends to compute one instance of the accumulation loop.

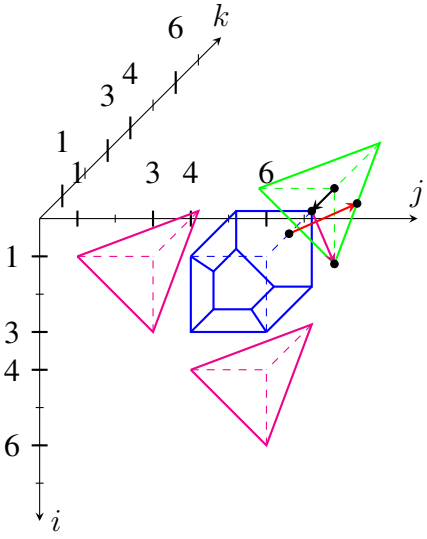
Figure 5.1b shows the dependence pattern in three dimensions. The points (1,6,1), (1,6,2) until (1,6,5) all contribute to one single point (1,6). It can be seen that the farthest west point((1,1) in red) is paired with the south point((2,6) in red). These two points are added to get (1,6,1). To calculate (1,6,2), the two magenta points and (1,6,1) (represented by the black arrow) are required. This pattern continues until (1,6,5) which is the required result. So, in general, the results of accumulation are located on the $16(\vec{k} - \vec{j})$ plane formed by points (1,2,1), (1,6,5) and (5,6,5). We



(a)



(b)



(c)

Consumer	Producer
(6,1,1) blue tile	(6,2,4) green tile
(6,1,2) blue tile	(6,3,3) green tile
(6,1,3) green tile	(6,1,2) blue tile

(d)

Figure 5.1: (a) Dependence graph of a OSP computation in 2D view (b)dependence graph of an OSP computation in 3D view (c) Inter-tile dependencies (d) Table to show the iteration points that create inter-tile dependency

can say that any point in the accumulation dimension will depend on the top-most plane and the immediate bottom neighbor.

When we generalize this dependence pattern, it can be realized that points in a diagonal depend on the south and west points of diagonals below it. So, to produce a functionally correct code, one of the many ways the program can traverse is along the diagonals starting from the inner-most where $j = i$. This scheduling is accurate with three-dimensional reasoning as well.

The problem arises when we try to tile all three dimensions. Suppose for a tile size three; we get tiles as shown in figure 5.1c. Now, the dependencies are not among points instead tiles. The magenta tiles do not depend on any other tiles. The blue tile depends on magenta tiles and green tile. The green tile depends on magenta tiles and blue tile. The legality of tiling is that there should not be any cyclic dependencies between the tiles. Because of this reason, it is not possible to tile all three dimensions in OSP. The inter-tile dependency between blue and green tile is due to the points illustrated in figure 5.1d. The producer is one whose results must be computed before the consumer can use it. This conflict occurs in the two accumulation dimensions of OSPSQ.

GKT serialization or the middle-serialization was introduced by Gubias et al. [30] for a 2D systolic array. This solves the locality problem found in 2D OSP problem and enables tiling in all three dimensions. It has been established that the automation of these techniques is an open challenge and no state-of-the-art automatic polyhedral tools can apply these transformations. Therefore, there is a need to apply them through manual or semi-automatic polyhedral tools.

The idea behind middle-serialization is to iterate the accumulation loop starting from the middle. When going from the middle, the results are accumulated on the i-j plane. This ensures that all tiles depend on bottom tiles alone and avoids inter-tile dependencies. We call this as OSPGKT version, and the corresponding recurrence equation is given by equation 5.2.

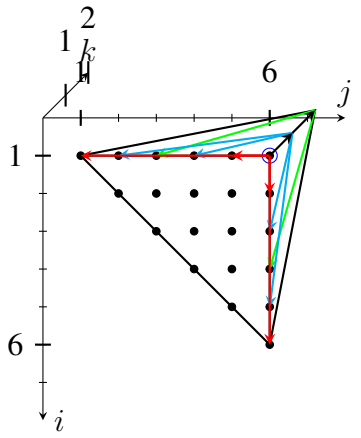


Figure 5.2: Dependence pattern in OSPGKT

$$T[i,j] = \begin{cases} 0, & \text{if } i == j. \\ \min_{k=0}^{(j-i)/2} min((T[i,i+k] + T[i+k+1,j] + (P[i-1] * P[i+k] * P[j])), \\ \qquad (T[i,j-k-1] + T[j-k,j] + (P[i-1] * P[j-k-1] * P[j]))), & \\ & \text{if } j > i. \end{cases} \tag{5.2}$$

Here, k has to be executed in decreasing order. In equation 5.1, the accumulation loop was from i to j. Now, it is from 0 to j-i/2. For example shown in figure 5.2, j-i=5. Since there is an odd number of points, in the first accumulation iteration, we have only one set of data points represented by green arrows. In the next iteration, the four blue points are touched. Notice that instead of accessing the west most point, the data points are accessed from middle and goes towards west. Similarly, instead of accessing the near south, the data points are touched from the middle and goes towards the nearest point. This order of accumulation ensures that we do not wait for the closest data point to start our computation. Once the middle points are available, the calculation can begin. This method not only helps to tile all three dimensions but also help to improve locality and performance.

5.2 OSPSQGKT : SIX DIMENSIONAL TILING

Here, we apply GKT method to tile all six dimensions of OSPSQ. In the original reduction body, we have two dependencies for each iteration. When we do a middle serialization, the accumulation loop reduces by half and number of dependencies per iteration doubles and so, for double accumulation it increases by a factor of 4. Equation 5.3 reflect this phenomena. We append "GKT" and call it as OSPSQGKT. The accumulation loops d, and e are executed in decreasing order.

$$\begin{aligned} T[i1,j1,i2,j2] = \sum_{d=0}^{(j1-i1)/2} \sum_{e=0}^{(j2-i2)/2} ((T[i1,i1+d,i2,i2+e]* \\ T[i1+d+1,j1,i2+e+1,j2]) + (T[i1,j1-d-1,i2,j2-e-1]* \\ T[j1-d,j1,j2-e,j2]) + (T[i1,i1+d,i2,j2-e-1]* \\ T[i1+d+1,j1,j2-e,j2]) + (T[i1,j1-d-1,i2,i2+e]* \\ T[j1-d,j1,i2+e+1,j2]))/func(([j1]-[i1]),[d],([j2]-[i2]),[e]) \end{aligned} \tag{5.3}$$

Notice the "$func()$" term in equation 5.3. This additional term is because, when we have an odd number of terms to accumulate, we end up repeating the computations. This phenomenon happens for every odd diagonal in the outer as well as inner accumulation loop. In the OSPGKT example to calculate (1,6), we saw that there was an odd number of points to accumulate. There was no need for the division because the operation performed is an idempotent operator, min. Whereas with OSPSQ, the accumulation operation is multiply, and so it either doubles or quadruples the result calculated.

Consider diagrams shown in figure 5.3. It shows two different execution order in the double accumulation iteration. $(6,6)$ gives the accumulated result. When we tile the iteration space according to the execution order in figure 5.3a, we get tiles with cycles as the first point in each row depends on the last point of the previous column. Tiling the iteration space with the execution order in figure 5.3b yields cycle-free legal tiles. This order of scheduling ensures tiling in all

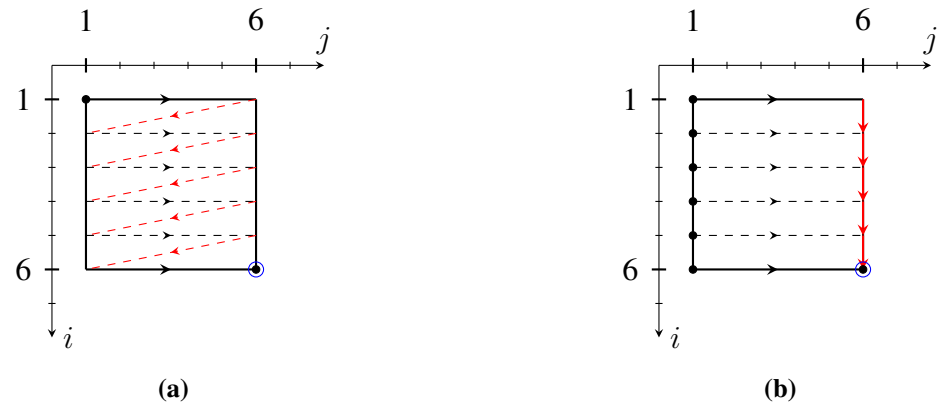


Figure 5.3: Execution order of double reduction in (a) yields tiles with cycles (b) yields cycle free tiles

six dimensions but needs an additional memory. This additional memory varies under different circumstances. We describe the steps to derive the memory formula.

1. Sequential tiled version: There is a change in the memory requirements depending on the loop permutations.

    - As long as the accumulation loops are the innermost, there is no additional memory. Since each tile will be executed in sequential order, the partial result from each tile can be accumulated as a scalar to the actual output data point. So, the total memory required is $O(N^4)$ which the initial data-space.

    - We denote the number of non-accumulation loops inner to accumulation loop by k. If any of the non-accumulation loops is made inner to the accumulation loop, there is a necessity to save the partial results. This is because, the accumulation loops are in the time dimension and therefore, many-to-one memory mapping occurs which results in overriding of data if additional memory is not allocated. For example, let us have a space-time map $(i, j, k, l, d, e \rightarrow i, j, k, d, e, l)$. Here, l is the non-accumulation loop inner to the accumulation loops d and e. To compute a point (i,j,k,l), we revisit that same point for every iteration of d and e loop. There are N such points for a given (i,j,k) plane. The memory required to store partial results would be N for every (i,j,k) plane. Assuming N is the range of i,j and k dimensions, the additional memory required

would be $O(N^4)$. Since we are dealing with sequential tiles of identical size say s, we can save the memory by re-using the tile space of (i,j,k) plane. So, the final additional memory required is $(s^3 * N)$.

2. Sequential non-tiled version: The additional memory required will increase by order of N. This is because in figure 5.3a there is only one starting point $(1, 1)$ whereas in figure 5.3b there are six starting points (represented by black dots) corresponding to the number of rows. In other words, there exist a parallel dimension. The memory space cannot be shared with the four-dimensional table as it will be over-written when each row is executed. Therefore an additional $O(N)$ memory is required to store the accumulations of each row. This is the minimum additional memory required when the accumulation loops are the inner-most. There is an increase in the order by N for each non-accumulation loop that is inserted inner to the accumulation loops as discussed for tiled version. So, it is better not to use this kind of schedule when there is no tiling involved.

3. Parallel versions: With each parallel dimension, there is an increase in the memory required by order of N. This is because, with each parallel dimension, there must be unique locations to store the partial tile accumulations without which there will be overriding of data. We denote the number of parallel dimensions outside the accumulation as p. This does not include the loops inner to the accumulation dimensions. The same argument on tiled version's tile space re-use holds here too.

$$Memory\_required = s^{t*(d-k-p)} * N^{k+p} + O(N^4) \tag{5.4}$$

where,

- N : Problem size
- s : Tile size
- d : no. of data dimension, here it is 4

- p : no. of parallel dimension outside the accumulation loop
- k : no. of non-accumulation loops inner to accumulation loop
- t : if the code is tiled or not (takes 0 or 1)

Equation 5.4 gives the formula for additional memory required for various scenario. The first term with the involving s gives the amount of additional tile space memory required. If the code is not tiled, it is zero. The second term is one if the code is sequential and accumulation loop are the inner-most. If not, it increases as per the previous discussion. The third term is the default data space memory of the program. For example, the sequential tiled schedule $(i, j, k, l, d, e \rightarrow -i, d, j, -k, e, l)$ would require $O(s * N^3)$ additional memory. In future, we plan to implement these tiling strategies and explore the performance of the code for various tile sizes.

Chapter 6

CONCLUSION & FUTURE WORK

In this work, we were able to identify and optimize OSPSQ kernel that captures the main dependence pattern found in RNA-RNA interaction applications - IRIS and piRNA. With simple techniques like loop permutation and skewing, we achieve an average of 17x sequential and 31x parallel speedup on a standard modern multi-core platform (Intel Broadwell, E5-1650v4). This performance represents 75% and 88% of attainable single-core and multi-core L1 bandwidth. For further performance improvement, we describe how to tile all six dimensions and also formulate the associated memory trade-off. In the future, we plan to implement these tiling strategies, explore the performance of the code for various tile sizes and optimize the whole piRNA application.

Bibliography

[1] Maitri Y. Shah and Alessandra Ferrajoli and Anil K Sood and Gabriel Lopez-Berestein and George A. Calin. MicroRNA Therapeutics in Cancer âĂŤ An Emerging Concept. *EBioMedicine*, 12(Supplement C):34 – 42, 2016.

[2] Behzad Mansoori, Siamak Sandoghchian Shotorbani, and Behzad Baradaran. RNA Interference and its Role in Cancer Therapy. *Adv Pharm Bull*, 4(4):313–321, Dec 2014. 25436185[pmid].

[3] T. F. Duchaine and F. J. Slack. RNA interference and micro RNA -oriented therapy in Cancer: rationales, promises, and challenges. *Curr Oncol*, 16(4):61–66, Aug 2009.

[4] Dmitri D Pervouchine. IRIS: intermolecular RNA interaction search. *Genome informatics. International Conference on Genome Informatics*, 15 2:92–101, 2004.

[5] Hamidreza Chitsaz, Raheleh Salari, S. Cenk Sahinalp, and Rolf Backofen. A partition function algorithm for interacting nucleic acid strands. *Bioinformatics*, 25(12):i365–i373, 2009.

[6] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The Polyhedral Model is More Widely Applicable Than You Think. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 283–303, Berlin, Heidelberg, 2010. Springer-Verlag.

[7] Gautam Gupta and Sanjay Rajopadhye. The Z-polyhedral Model. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 237–248, New York, NY, USA, 2007. ACM.

[8] Cedric Bastoul. Code Generation in the Polyhedral Model Is Easier Than You Think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society.

[9] Louis-Noël Pouchet. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 2012.

[10] Tomofumi Yuki, Gautam Gupta, Tanveer Pathan, and Sanjay Rajopadhye. Systematic Implementation of fast-i-loop in UNAfold using AlphaZ. Technical report, CS-12-102, Colorado State University, 2012.

[11] Ahmed, Syed Ali and Mneimneh, Saad. *Multiple RNA Interaction with Sub-optimal Solutions*, pages 149–162. Springer International Publishing, Cham, 2014.

[12] Fenix W. D. Huang, Jing Qin, Christian M. Reidys, and Peter F. Stadler. Target prediction and a statistical sampling algorithm for RNAâĂŞRNA interaction. *Bioinformatics*, 26(2):175–181, 2010.

[13] Alkan, Can and Karakoç, Emre and Nadeau, Joseph H. and Şahinalp, S. Cenk and Zhang, Kaizhong. *RNA-RNA Interaction Prediction and Antisense RNA Target Search*, pages 152–171. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[14] Yuki Kato, Kengo Sato, Michiaki Hamada, Yoshihide Watanabe, Kiyoshi Asai, and Tatsuya Akutsu. RactIP: fast and accurate prediction of RNA-RNA interaction using integer programming. *Bioinformatics*, 26(18):i460–i466, 2010.

[15] Salari, Raheleh and Backofen, Rolf and Sahinalp, S. Cenk. Fast prediction of RNA-RNA interaction. *Algorithms for Molecular Biology*, 5(1):5, Jan 2010.

[16] Salari, Raheleh and Möhl, Mathias and Will, Sebastian and Sahinalp, S. Cenk and Backofen, Rolf. *Time and Space Efficient RNA-RNA Interaction Prediction via Sparse Folding*, pages 473–490. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[17] Hamidreza Chitsaz, Elmirasadat Forouzmand, and Gholamreza Haffari. An Efficient Algorithm for Upper Bound on the Partition Function of Nucleic Acids. *Journal of Computational Biology*, 20(7):486–494, 2013.

[18] Rune B Lyngso, Michael Zuker, and CN Pedersen. Fast evaluation of internal loops in RNA secondary structure prediction. *Bioinformatics (Oxford, England)*, 15(6):440–445, 1999.

[19] David Wonnacott, Tian Jin, and Allison Lake. Automatic tiling of âĂIJmostly-tileableâĂİ loop nests. In *5th International Workshop on Polyhedral Compilation Techniques, Amsterdam, The Netherlands*, 2015.

[20] Francisco Almeida, Rumen Andonov, Daniel Gonzalez, Luz M. Moreno, Vincent Poirriez, and Casiano Rodriguez. Optimal Tiling for the RNA Base Pairing Problem. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, pages 173–182, New York, NY, USA, 2002. ACM.

[21] Guangming Tan, Shengzhong Feng, and Ninghui Sun. Locality and Parallelism Optimization for Dynamic Programming Algorithm in Bioinformatics. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[22] Ron Sass and Russell Tessier, editors. *18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2010, Charlotte, North Carolina, USA, 2-4 May 2010*. IEEE Computer Society, 2010.

[23] Uday Bondhugula, J. Ramanujam, and P. Sadayappan. PLuTo: A Practical and Fully Automatic Polyhedral Parallelizer and Locality Optimizer. Technical Report OSU-CISRC-10/07-TR70, The Ohio State University, October 2007.

[24] Hironori Kasahara and Keiji Kimura, editors. *Languages and Compilers for Parallel Computing, 25th International Workshop, LCPC 2012, Tokyo, Japan, September 11-13, 2012, Revised Selected Papers*, volume 7760 of *Lecture Notes in Computer Science*. Springer, 2013.

[25] Thomas H Cormen. *Introduction to algorithms*. 2009.

[26] John D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[27] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[28] Torsten Hoefler and Roberto Belli. Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses when Reporting Performance Results. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 73:1–73:12, New York, NY, USA, 2015. ACM.

[29] Larry McVoy and Carl Staelin. Lmbench: Portable Tools for Performance Analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association.

[30] L. J. Guibas, H. T. Kung, and C. D Thompson. Direct VLSI Implementation of Combinatorial Algorithms. In *In: Proceedings of the Caltech Conference On Very Large Scale Integration*, pages 509–525. California Institute of Technology , Pasadena, CA, 1979.