THESIS

# ACCELERATING THE BPMAX ALGORITHM FOR RNA-RNA INTERACTION

Submitted by

Chiranjeb Mondal

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Summer 2021

Master's Committee:

Advisor: Sanjay Rajopadhye

Louis-Noel Pouchet
Anton Betten

# ABSTRACT

## ACCELERATING THE BPMAX ALGORITHM FOR RNA-RNA INTERACTION

RNA-RNA interactions (RRIs) are essential in many biological processes, including gene transcription, translation, and localization. They play a critical role in diseases such as cancer and Alzheimer's. An RNA-RNA interaction algorithm uses a dynamic programming algorithm to predict the secondary structure and suffers very high computational time. Its high complexity ($\Theta(N^3M^3)$ in time and $\Theta(N^2M^2)$ in space) makes it both essential and a challenge to parallelize. RRI programs are developed and optimized by hand most of the time, which is prone to human error and costly to develop and maintain.

This thesis presents the parallelization of an RRI program - BPMax on a single shared memory CPU platform. From a mathematical specification of the dynamic programming algorithm, we generate highly optimized code that achieves over $100\times$ speedup over the baseline program that uses a standard "diagonal-by-diagonal" execution order. We achieve 100 GFLOPS, which is about a fourth of our platform's peak theoretical single-precision performance for max-plus computation. The main kernel in the algorithm, whose complexity is $\Theta(N^3M^3)$ attains 186 GFLOPS. We do this with a polyhedral code generation tool, AlphaZ, which takes user-specified mapping directives and automatically generates optimized C code that enhances parallelism and locality. AlphaZ allows the user to explore various schedules, memory maps, parallelization approaches, and tiling of the most dominant part of the computation.

# ACKNOWLEDGEMENTS

I want to thank my advisor Dr. Sanjay Rajopadhye for his guidance and encouragement in developing this thesis. I am also grateful to him for accepting me as a student, spending his precious time with me, and guiding me through the basics of high-performance computing and polyhedral compilation. I would also like to thank Dr. Louis-Noel Pouchet and Anton Betten for their valuable inputs.

I would also like to thank the Melange team at CSU. I want to thank Louis Narmour and Tomofumi Yuki for their support of ALPHAZ.

I must acknowledge the continuous encouragement from my parents Anurupa Mondal and Tarak Chandra Mondal, my sisters Anindita and Aparajita. Finally, I would like to mention the tremendous support from my wife Priyanka throughout the master's program, without whom this journey would not have been possible at all.

# DEDICATION

*I would like to dedicate this thesis to my parents, wife, and children (Prachi and Pragyan).*

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

# Chapter 1

# Introduction

Ribonucleic acid (RNA) is the origin of life. It plays an essential role in the coding, decoding, regulation, and expression of genes. RNA is a single strand formed by a sequence of four different types of nucleotides – Adenine (A), Uracil (U), Guanine (G), and Cytosine (C), which form a repeating structure. Different nucleotides may form bonds of varying strength. A single RNA strand folds into itself. Also, two different RNA strands can interact with each other, resulting in the secondary structure, which may provide valuable information about a biological function. Guanine forms the strongest bond with Cytosine, Adenine forms the next strongest bond with Uracil, and Uracil forms the weakest bond with Guanine. A single RNA strand folds into itself. Also, two different RNA strands can interact with each other, resulting in the secondary structure. Knowledge of such structure can provide useful information about a biological function and may be used in experiments. Mortimer et al. [23] highlights the emerging relationships between such RNA structure and the regulation of gene expression.

RNA-RNA interactions have been moved to the spotlight in biology since the mid-1990s with significant RNA interference discovery. Researchers have long been studying these interactions and proposed different models. Chitsaz et al. [4] developed piRNA - one of the most comprehensive thermodynamic models for RRI. It has an $\Theta(N^4M^2 + N^2M^4)$ time and $\Theta(N^4 + M^4)$ space complexity, where $M$ and $N$ are the numbers of nucleotides present in each RNA sequence. It uses $96$ dynamic tables. However, running this compute and the memory-intensive program is exceptionally challenging. It takes days and months to get experimental results. So, Ebrahimpour-Boroojeny et al. [8] retreated from the slow comprehensive model and developed the BPPart [9] to obtain base-pair partition function and BPMax to maximize the base-pair. They use a simplified energy model and consider only base-pair counting. Both of them have similar asymptotic time and space complexity of $\Theta(M^3N^3)$ and $\Theta(M^2N^2)$. BPPart uses $11$ tables, and BPMax uses a

single one. Nevertheless, the original implementation of even BPMax suffers from poor memory locality, lacks auto-vectorization, and runs extremely slow for longer input sequences.

Ebrahimpour-Boroojeny et al. [8] conclude that BPMax captures a significant portion of the thermodynamic information. The Pearson and Spearman's rank correlation between piRNA and BPMax is $0.904$ at $-180°C$ and $0.836$ at $37°C$ highlighting its importance. BPMax and other RRI algorithms such as piRNA [4], IRIS [26], and RIP [16] follow similar recurrence patterns. So, besides the practical usefulness of BPMax, the learning and insights gleaned from the BPMax optimization approach can be applied to the other RRI interaction algorithms with similar recurrence patterns.

Performance optimization requires exploiting parallelism and locality at multiple levels. It is a difficult task and often leads to hand-crafted code. Manual optimization is neither easily portable (e.g., to different platforms where different kinds of optimization are needed) nor easily maintainable (e.g., changing the optimization strategy may require changes to many parts of the code). This challenge grows as the complexity of the program increases. It is highly desirable that the optimized programs get generated from a simple correct program together with a set of performance tuning hints or directives.

Fortunately, RRI algorithms fit the requirements of the *polyhedral model* [11–13, 27–29, 31–33], a mathematical formalism that allows for just such program transformations. The polyhedral compilation has been the subject of intense research for 35 years. Yet, even a state-of-the-art polyhedral tool like PLUTO [1, 2] does not yield satisfactory performance. Specifically, Varadarajan [36] evaluated the performance of PLUTO on a simple program that implements the equation: $X_{i_1,j_1,i_2,j_2} = \sum_{k_1=i_1}^{j_1-1} \sum_{k_2=i_2}^{j_2-1} X_{i_1,k_1,i_2,k_2} \times X_{k_1+1,j_1,k_2+1,j_2}$, which is a core computation of RRI algorithms, and found it to be significantly lower than the hand-written baseline implementation. Many of the performance optimization strategies need some careful thoughts by an expert. This gap can be bridged by a tool that allows semi-automatic transformation like Chill [3]. At CSU, we are developing and working with a similar tool, ALPHAZ [38], that operates at a higher level of abstraction for generating optimized code.

## 1.1 Contribution

- We optimize the RRI program - BPMax on a single CPU machine using a polyhedral code generation tool - AlphaZ .
- It is the first time a complex RRI program like BPMax has been optimized using AlphaZ in its entirety. Previous attempts were limited to a micro-kernel only. It is also the first attempt to optimize BPMax on the CPU.
- We generate highly optimized code for BPMax that achieves more than $100\times$ speedup over the original program. It is close to one-fourth of our platform's theoretical machine peak.
- The most compute-intensive part of BPMax achieves a $223\times$ speedup over the original implementation, and a $2\times$ improvement over a similar kernel optimized by Varadarajan [36]. It is close to $40\%$ of our platform's machine peak.

## 1.2 Thesis Structure

Chapter 2 sets up the context with related work and background to highlight the BPMax algorithm, recurrence equations involved in the algorithm, discuss the polyhedral model's role, and application of AlphaZ in code optimization using a trivial example. Chapter 3 discusses multiple phases of the optimization process, the rationale behind different schedules, processor allocations, memory mappings, and tiling. We go over our performance results in Chapter 4. Finally, Chapter 5 presents our conclusion and future directions for BPMax and other RRI applications.

# Chapter 2

# Background

This chapter highlights the related work, BPMax algorithm, summarizes the polyhedral model, and then provides a brief background of our code generation tool - AlphaZ .

## 2.1 Related Work

One of the early studies on interactions between nucleotides of single RNA was proposed by Nussinov [24] in 1978 that predicts secondary structure based on the probability that maximizes the number of base pairs in it. Nussinov's algorithm has a complexity of $\Theta(N^3)$ time and $\Theta(N^2)$ space. In 1981, Zuker and Stiegler [39] came up with a more sophisticated algorithm to predict an optimal secondary structure through free energy minimization (FEM). An energy minimization algorithm assumes that the correct structure has the lowest amount of free energy. Most of the RNA secondary structure prediction uses a dynamic programming algorithm. Although, it has also been formulated as a Bayesian inference problem [5].

There were prior works on the optimization and parallelization of these algorithms on the CPU platform. Li et al. [21] worked on the CPU and GPU versions of the Nussinov [24] RNA folding. Swenson et al. [35] worked on a parallel secondary structure prediction program for multi-core desktop. Palkowski et al. [25] used the polyhedral model to optimize Nussinov's [24] algorithm. Rizk et al. [34] presented a GPU implementation of Zuker's algorithm [39]. However, most of the optimization efforts were related to single RNA strand folding.

Varadarajan [36, 37] applied semi-automatic transformation using AlphaZ for a simplified surrogate mini-app that mimicked the dependence pattern to focus only on the most compute-intensive portion of the original piRNA. The original shared-memory OpenMP programs related to BPMax, BPPart, and piRNA try to achieve maximum parallelization without auto-vectorization and suffer very poor locality. She exploited locality using both coarse and fine-grain parallelism and achieved around $31\times$ speedup.

Glidemaster [14] achieved significant speedup on a windowed version of the BPMax on GPU. However, only up to a limited number of nucleotide sequences or a window of nucleotide sequences can be processed on GPU due to memory constraints. Also, the cost of moving data out of the GPU memory negatively impacts the overall performance. So, it is crucial to speed up the algorithm on the CPU to avoid these constraints. It can also further open up the possibility of a higher degree of parallelism over multiple machines.

## 2.2 BPMax Algorithm

BPMax uses weighted base-pair counting for base-pair maximization. It considers both intermolecular and intramolecular base-pairings but disallows pseudo-knots or crossings. Mathematically, it produces a four-dimensional triangular table - $F$-table (a triangular collection of triangles) based on two RNA sequences. Figure 2.1 shows the main cases defined by Ebrahimpour-Boroojeny et al. [8] for the $F$-table using the Eddy-Rivas diagrams, used in describing bioinformatics algorithms on sequences. Recurrence equations 2.1 and 2.2 show the complete specification of BPMax [8]. There are five reductions, $R_0$, $R_1$, $R_2$, $R_3$, and $R_4$, each of which is highlighted in a different color. We use the same colors to highlight the dependence pattern in Chapter 3. The blue reduction ($R_0$) represents the double max-plus operation. It is the most compute-intensive portion of the algorithm.

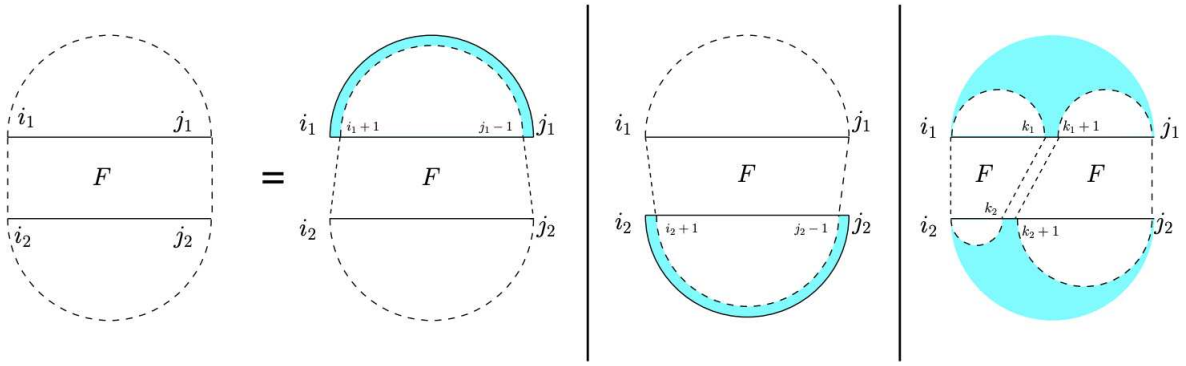



**Figure 2.1:** The four cases defining table $F$

$$
F_{i_1,j_1,i_2,j_2} = max \begin{cases} S^{(2)}_{i_2,j_2} & j_1 \le i_1 \\ S^{(1)}_{i_1,j_1} & j_2 \le i_2 \\ \text{iscore}(i_1,i_2) & i_1 = j_1 \text{ and } i_2 = j_2 \\ \max \begin{pmatrix} \boxed{F_{i_1+1,j_1-1,i_2,j_2}} + \text{score}(i_1,j_1), \\ \boxed{F_{i_1,j_1,i_2+1,j_2-1}} + \text{score}(i_2,j_2), \\ S^{(1)}_{i_1,j_1} + S^{(2)}_{i_2,j_2}, \\ \max\limits_{k_1=i_1}^{j_1-1} \max\limits_{k_2=i_2}^{j_2-1} F_{i_1,k_1,i_2,k_2} + F_{k_1+1,j_1,k_2+1,j_2} \quad \rightarrow \mathbf{R_0} \\ \max\limits_{k_2=i_2}^{j_2-1} S^{(2)}(i_2,k_2) + F_{i_1,j_1,k_2+1,j_2} \quad \rightarrow \mathbf{R_1} \\ \max\limits_{k_2=i_2}^{j_2-1} F_{i_1,j_1,i_2,k_2} + S^{(2)}(k_2+1,j_2) \quad \rightarrow \mathbf{R_2} \\ \max\limits_{k_1=i_1}^{j_1-1} S^{(1)}(i_1,k_1) + F_{k_1+1,j_1,i_2,j_2} \quad \rightarrow \mathbf{R_3} \\ \max\limits_{k_1=i_1}^{j_1-1} F_{i_1,k_1,i_2,j_2} + S^{(1)}(k_1+1,j1) \quad \rightarrow \mathbf{R_4} \end{pmatrix} & otherwise \end{cases} \tag{2.1}
$$

$$
S_{i,j} = max \begin{cases} 0 & j-i \le 4 \\ \max[S_{i+1,j-1} + \text{score}(i,j), \max\limits_{k=i}^{j-1} S_{i,k} + S_{k+1,j}] & otherwise \end{cases} \tag{2.2}
$$

## 2.3 Polyhedral Model

*The Polyhedral model* [11–13, 27–29, 31–33] is a mathematical framework for automatic optimization and parallelization of affine programs. A polyhedron is the intersection of finitely many half-spaces. It can be bounded (polytope) or unbounded. The model provides an abstraction to represent static control parts like variables, iteration space (loop nests), and dependencies using integer points in polyhedra.

**Listing 2.1:** Prefix sum

```
for(int i=0;i<7;i++ ) {
    sum[i]=0;
    for (int j=0 ; j<=i;  j++)
        sum[i] += array[j];
}
```

Let us consider the prefix sum code highlighted in Listing 2.1 which computes the prefix sum of an array of size 7. The iteration space for this computation can be represented using the intersection of the finite half-spaces or set of inequalities such as $j \leq i$, $i \leq 6$ and $j = 0$. The points in the iteration spaces are marked with the dots represented by the polyhedron with vertices $(0, 0), (0, 6)$, and $(6, 0)$. The data space is usually one or more dimensions less than the iteration space. As a result, the data access functions are many-to-one mappings from iteration to data space. Polyhedral transformation functions are affine. A function $f : \mathbb{R}^m \mapsto \mathbb{R}^n$ is affine if there is a matrix $A \in \mathbb{R}^{m \times n}$ and a vector $\vec{b} \in \mathbb{R}^n$ such that $\forall \vec{x} \in \mathbb{R}^m$, $f(\vec{x}) = A\vec{x} + \vec{b}$. An affine transformation preserves the collinearity and convexity of points while transforming a polyhedron into another polyhedron, leading to the model's clean closure properties under the program transformation.

Going back to the original equation, a concise way to look at this computation would be to view it as an equation: $sum[i] = \sum_{j=0}^{i} array[j]$. The idea behind a polyhedral tool like AlphaZ is exactly the reverse. It allows the user to express one or more system of affine recurrence equations as a program, transform them using the polyhedral transformations that reduce the complexity of

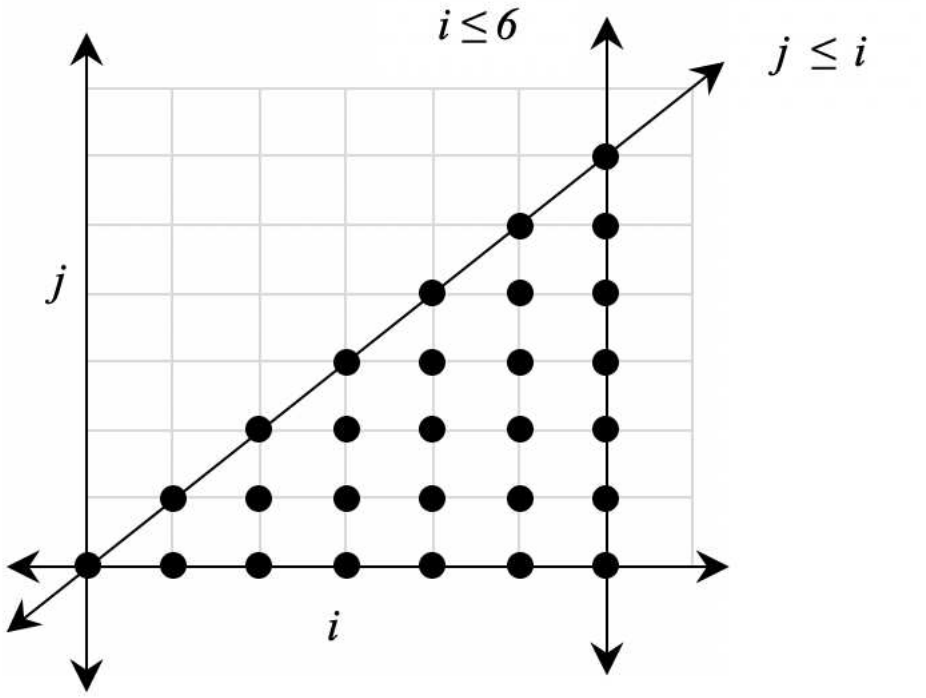



**Figure 2.2:** Polyhedral iteration space for prefix-sum

the program, use a better processor and memory allocation, and then produce optimized code for a language of interest.

## 2.4 ALPHAZ

ALPHA [22] is a strongly typed functional language based on systems of affine recurrence equations defined over polyhedral domains. It was developed by Mauras [22] in 1989. Subsequently, it was extended to include subsystems and reductions [6, 7, 10, 19, 20]. ALPHAZ is a tool that allows program transformations and user-directed compilation of ALPHA programs. It provides a general framework for analysis, transformation, and code generation in the polyhedral equational model. ALPHAZ is similar to an earlier tool - MMALPHA [15] , which targets field-programmable gate array-based hardware design. On the other hand, ALPHAZ targets code generation for multiprocessor shared-memory programs and focuses on programs with reduction operations.

Most of the polyhedral code optimization tools use hard-coded transformation strategies and generate code automatically. But the performance of such code often falls short of a hand-written

optimized version. To avoid fixed transformation strategies, tools like Chill [3], Hailde [30], and ALPHAZ [38] implement various code transformation APIs and present them to the users. It allows users to choose different transformations for a specific problem, enabling a large exploration space for the optimization process.

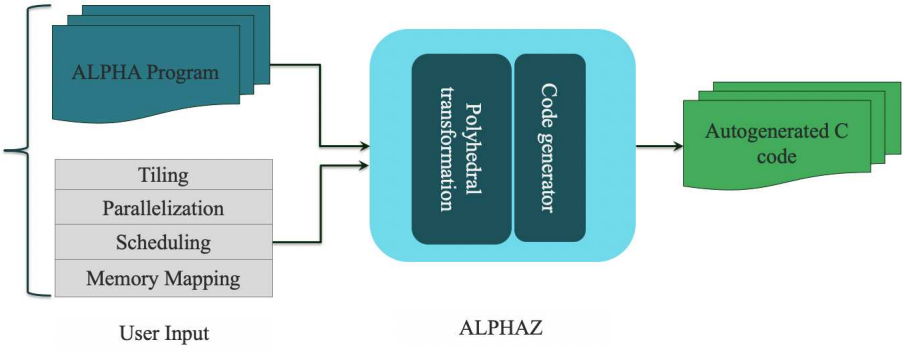



**Figure 2.3:** Code generation methodology

ALPHAZ code optimization process has two parts – input specification and compilation script. Input specification allows a user to express the computation using mathematical equations. The compilation script takes inputs (e.g., scheduling, parallelization, memory-mapping, and tiling transformations) from the user to generate optimized C code corresponding to the input specification. Figure 2.3 highlights the code optimization methodology using ALPHAZ .

All the transformations in ALPHAZ are semantics preserving. However, it is the responsibility of the user to ensure the transformations are valid. We use three important classes of transformations - target mappings, memory mappings, and tiling-related transformations. Target mappings-related transformations determine the execution order of the program. It allows the user to specify schedule and processor allocation for each variable present in the system. It also allows the user to specify one or more dimensions of the schedule to be executed in parallel by different threads. Memory mappings-related transformations allow multiple variables with different dimensions to share the same memory map based on affine function. It also allows multiple variables with the same dimension to share memory space. Tiling transformations chop the iteration space to improve

data locality and adjust parallelization granularity. Target and Memory mappings-related transformations require the user to specify affine functions to indicate the schedule or memory map. The affine functions are expressed as $(ListOfIndices \mapsto ListOfIndexExpressions)$.

- A schedule $(i, j \mapsto j, i)$ tells ALPHAZ that the iteration domain is 2-dimensional represented by $i$, $j$ as the $ListOfIndices$, and the points in this iteration domain should be visited in the order given by the $ListOfIndexExpression$ $j$ and $i$.

- A memory map $(i, j \mapsto j, i)$ tells ALPHAZ that the mapping is associated with a 2-D variable whose $(i, j)$-th element is stored at a location specified by $ListOfIndexExpressions$ - $(j, i)$. It also allows the user to save memory if there is an opportunity for many-to-one mapping. E.g., $(i, j, k \mapsto i, j)$.

Efficient scheduled code generation depends on the choice of target and memory mappings-related transformations. Algorithm 1 highlights the ALPHA program for matrix multiplication. Algorithm 2 presents a compiling script for matrix multiplication that produces the C code highlighted in Listing 2.2.

**Algorithm 1** Matrix Multiplication in Alphabets

```
affine MM {N, K, M | (M, N, K) > 0}
input
    float A {i, j | 0 ≤ i < M && 0 ≤ j < K} ;
    float B {i, j | 0 ≤ i < K && 0 ≤ j < N} ;
output
    float C {i, j | 0 ≤ i < M && 0 ≤ j < N};
local
    //local variables
output
    C[i, j] = reduce(+,  [k],  A[i, k] * B[k, j]);
```

**Algorithm 2** Matrix Multiplication Command Script

```
1: // Step − 1 : Parse Alphabet
2: prog=ReadAlphabets("MM.ab");
3: system = “MM”;
4: outDir="./src";
5:
6: // Step − 2 : Perform polyhedral transformation
7: Normalize(prog);
8: setSpaceTimeMap(prog, system, “C”, "(i, j, k ↦ i, k, j)", “(i, j ↦ i, −1, j)”);
9: setParallel(prog, system, “”, "0" );
10:
11: // Step − 3 : Generate code
12: generateWriteC(prog, system, outDir);
13: generateScheduleC(prog, system, outDir);
```

**Listing 2.2:** Generated code - Matrix multiplication

```
#define S1(i,j,i2) C(i,i2) = 0.0
#define S0(i0,i1,i2) C(i0,i2) = (C(i0,i2))+((A(i0,i1))*(B(i1,i2)))
    {
        int c1,c2,c3;
        #pragma omp parallel for private(c2,c3)
        for(c1=0;c1 <= M-1;c1+=1){
            for(c3=0;c3 <= N-1;c3+=1){
                S1((c1),(-1),(c3));
            }
          for(c2=0;c2 <= K-1;c2+=1){
            for(c3=0;c3 <= N-1;c3+=1){
                S0((c1),(c2),(c3));
            }
          }
        }
    }
```

# Chapter 3

# Method

In this section, we first go over the BPMax dependencies, describe the limitations of current BPMax implementation, and then go over the different phases of the optimization process.

BPMax algorithm computes a four-dimensional sparse table $F(i_1, j_1, i_2, j_2)$ shown in Figure 3.1. The sparsity can be viewed as a triangle of triangular collections, where $(i_1, j_1)$-th triangle is denoted by $F(i_1, j_1)$ and often referred to as an inner triangle. The $(i_2, j_2)$-th element of $F(i_1, j_1)$ is denoted as $F_{i_1,j_1,i_2,j_2}$. Figure 3.1 shows the complete BPMax dependencies for an

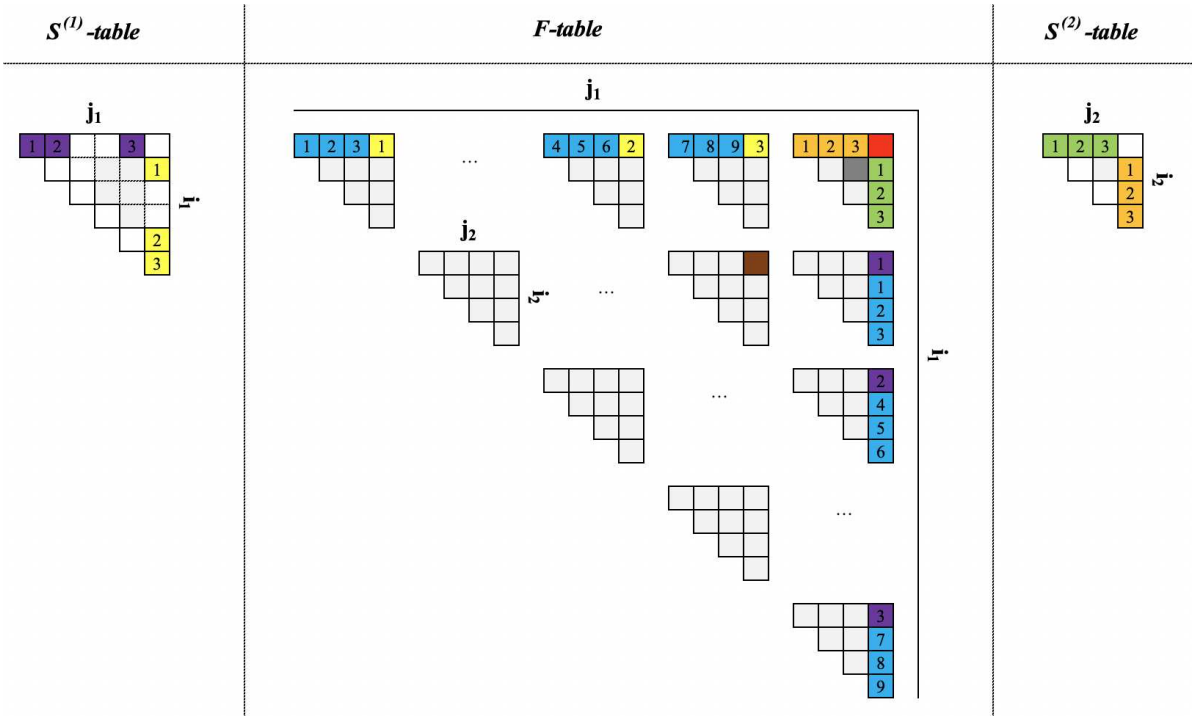



**Figure 3.1:** BPMax dependency overview

$F$-table element highlighted in red, which is dependent on all the blue, yellow, purple, orange, and green points of the $F$-table, $S^{(1)}$-table, and $S^{(2)}$-table. Each color represents the computation of a particular reduction operation ($R_0 - R_4$). All these reduction operations need to be completed to update the point highlighted in red. $R_0$ is the most compute-intensive ($\Theta(M^3N^3)$) reduction that

uses the points outside the current triangle. E.g., to compute the $R_0$ for the red point, the numbered blue points towards the left are added with the corresponding blue points towards the south, and then the max of all these values are computed. Now, $R_3$ and $R_4$ also use the elements from the external triangles as one of the operands and $S^{(1)}$ as the other operand. E.g., the numbered-yellow and purple points from the $F$-table are added with the corresponding yellow and purple points from the $S^{(1)}$-table, and then the max of yellow and purple results are computed to produce the $R_3$ and $R_4$, respectively. These two reductions have a complexity of $\Theta(M^3N^2)$. The remaining two reductions, $R_1$ and $R_2$, have intra-triangular dependencies. E.g., the numbered orange and green points from the $F$-table are added with the corresponding orange and green points from the $S^{(2)}$-table, and then the max of orange and green results are computed to produce the $R_1$ and $R_2$, respectively.

Table 3.1 highlights the original 6-D schedule of each one of these reductions. Figure 3.2 demonstrates the key issue with the original schedule, which fills up the $F$-table diagonal by diagonal. It is a wavefront-like parallelization of the 6-D schedule space. All the diagonal points

**Table 3.1:** BPMAX ORIGINAL SCHEDULE

| ***Reductions*** | ***Schedules*** |
|---|---|
| $S^{(1)}$ | $(i_1, j_1, k_1 \mapsto j_1 - i_1, i_1, k_1)$ |
| $S^{(2)}$ | $(i_2, j_2, k_2 \mapsto j_2 - i_2, i_2, k_2)$ |
| $R_0$ | $(i_1, j_1, i_2, j_2, k_1, k_2 \mapsto j_1 - i_1, j_2 - i_2, i_1, i_2, k_1, k_2)^{\text{a}}$ |
| $R_1, R_2$ | $(i_1, j_1, i_2, j_2, k_2 \mapsto j_1 - i_1, j_2 - i_2, i_1, i_2, k_2)^{\text{a}}$ |
| $R_3, R_4$ | $(i_1, j_1, i_2, j_2, k_1 \mapsto j_1 - i_1, j_2 - i_2, i_1, i_2, k_1)^{\text{a}}$ |

[a]Parallel Dimension 2 (0 based)

are computed simultaneously, exposing maximum level of parallelism. It accesses all the inner triangles (highlighted in light red) towards the left of the diagonal points (red points Figure 3.2). Thus, the total amount of the active data elements required to compute all these points simultaneously can exceed the last-level cache for a larger input size, triggering a lot of data movement between different levels of caches and main memory. Memory reuse is almost impossible as we

move to the next diagonal. Thus, the original program suffers from poor data locality. Also, each of these reductions in the original implementation has loop carried dependencies, preventing auto-vectorization. Keeping these challenges in mind, we have staged our optimization process into

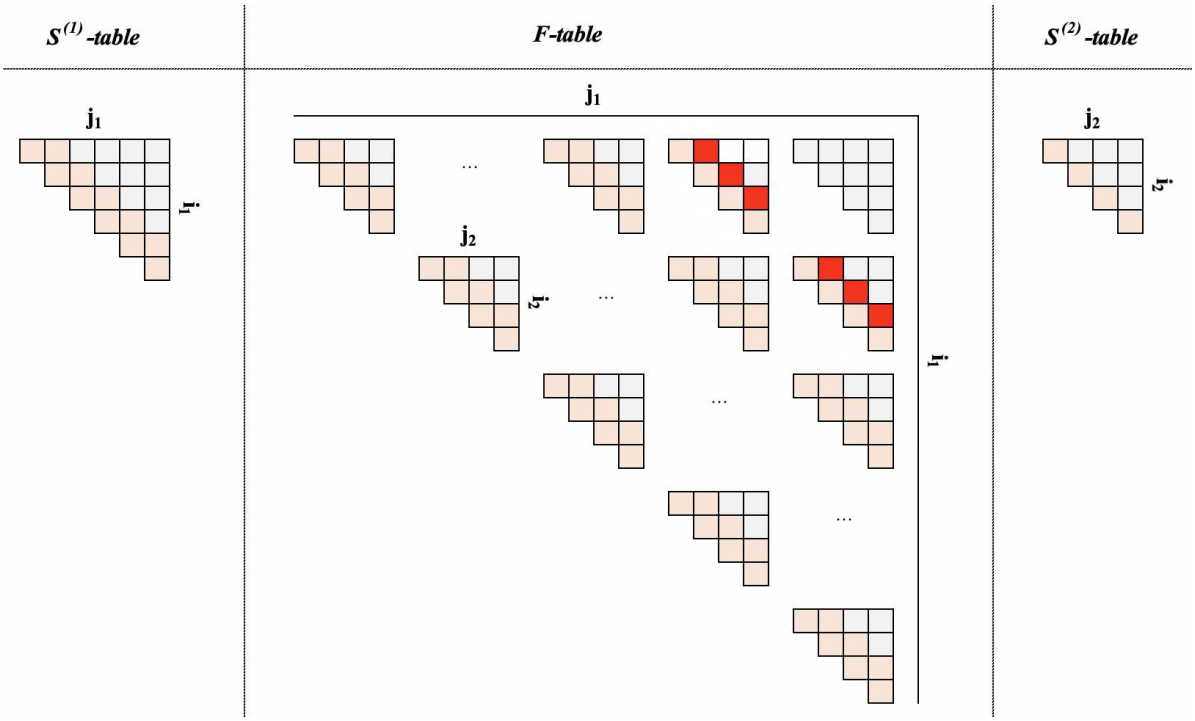



**Figure 3.2:** BPMax Original Schedule

three different phases. In the first phase, we optimize the double max-plus operation. Then, we attempt first level optimization of the entire BPMax program. We explore various schedules and parallelization approaches to enable auto-vectorization and improve the data locality. We also tile the most compute-intensive portion in the second phase. However, we observe that tiling the entire program with a single AlphaZ system generates very inefficient code. Thus, we express the BPMax using AlphaZ subsystem in the third phase. We also explore different memory optimizations, parallelization approaches to maximize resource utilizations.

## 3.1 Phase I

Previously, Varadarajan [36] optimized a double reduction kernel that uses double-precision multiply and add operation. But, it had the same data access pattern. She achieved $31\times$ performance improvement by using loop permutations to take advantage of the auto-vectorization. This phase's primary goal is to use the same schedule and replace it with the max-plus operation and get a baseline performance estimation for the $R_0$.

### 3.1.1 Optimum Schedule for Double max-plus Operation

Let us recall the double max-plus reduction:

$$F_{i_1,j_1,i_2,j_2} = \max_{k_1=i_1}^{j_1-1} \max_{k_2=i_2}^{j_2-1} F_{i_1,k_1,i_2,k_2} + F_{k_1+1,j_1,k_2+1,j_2} \tag{3.1}$$

In the previous section, we concluded the limitations of the base version of the BPMax due to the poor reuse across the diagonal elements of the multiple inner triangles for producing $R_0$, $R_3$, $R_4$. Instead of using this wavefront-like parallelization of the six-dimensional schedule space for $R_0$, Varadarajan's [36] schedule filled up one inner triangle at a time to reduce the required amount of active data and increases the chances of reuse. When we fill up one inner triangle at a time, the first two schedule dimensions iterate over the triangles and the last four dimensions of the schedule compute the reduction result for each element. The outer triangle can only be filled diagonally or "bottom-up left to right" to honor the double max-plus dependence highlighted in Figure 3.3. Thus, the first two schedule dimensions can be $(j_1 - i_1, i_1)$ or $(M - i_1, j_1)$ or $(-i_1, j_1)$. Now, the computation of $F(i_1, j_1)$ needs all the $F(i_1, k_1)$ triangles towards the west and $F(k_1 + 1, j_1)$ triangles ($i_1 \leq k_1 < j_1$) towards the south illustrated in Figure 3.3. More precisely, the computation of each element of $F(i_1, j_1)$ requires a set of $F(i_1, k_1)$ and $F(k_1 + 1, j_1)$ triangles. Now, two column-adjacent and row-adjacent elements of $F(i_1, j_1)$ have significant re-use from the $F(i_1, k_1)$ and $F(k_1 + 1, j_1)$, respectively.

The third dimension of the schedule iterates over these triangles to accumulate the max-plus result of the inner triangle. It is the same as the outer reduction index $k_1$. Figure 3.4a demonstrates

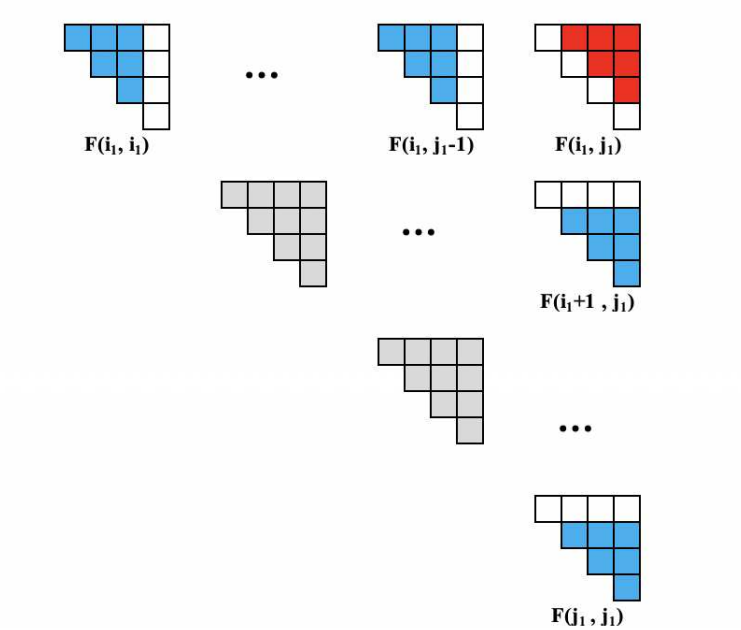
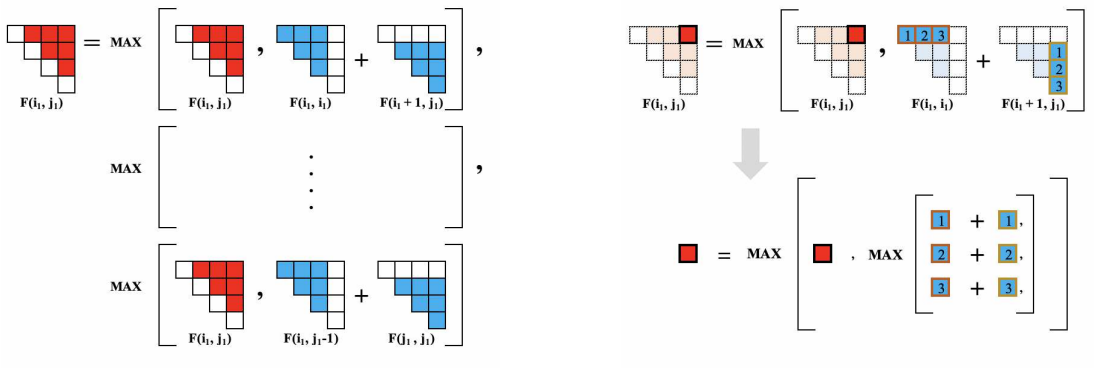



**Figure 3.3:** Double max-plus dependency

this accumulation sequence over $k_1$. The inner three dimensions of the schedule iterate over each set of these triangles and produce the final result using matrix-product like operation highlighted in Figure 3.4b. Except, it only performs a fraction of cubic max-plus operations. The performance of this computation is dependent on certain loop permutations. E.g., permutations with loop carried dependencies prevent auto vectorizations. Because of this, we choose $j_2$ as the innermost loop. Table 3.2 highlights different schedules used for the $R_0$.




**(a)** First level Decomposition **(b)** Second level Decomposition

**Figure 3.4:** Decomposition of double max-plus computation for an inner triangle

### 3.1.2 Parallelization Approach

We apply similar parallelization approaches used by Varadarajan [36] - coarse and fine-grain. Typically, the terms - fine and coarse-grain are used to highlight the thread and vector level parallelization. But, our definitions of fine and coarse-grain are based on how we process an inner triangle. Each thread computes one inner triangle in coarse-grain parallelization. In fine-grain parallelization, multiple threads work together to compute one inner triangle where each thread processes one or more rows of an inner triangle.

**Table 3.2:** DOUBLE MAX-PLUS SCHEDULE

| ***Schedule*** | **Parallelization Approach** | ***Parallel Dimension***[b] |
|---|---|---|
| $(i_1, j_1, i_2, j_2, k_1, k_2 \mapsto j_1 - i_1, i_1, k_1, i_2, k_2, j_2)$ | Fine-grain (diagonal[a]) | 3 |
| $(i_1, j_1, i_2, j_2, k_1, k_2 \mapsto -i_1, j_1, k_1, -i_2, k_2, j_2)$ | Fine-grain (bottom-up[a]) | 3 |
| $(i_1, j_1, i_2, j_2, k_1, k_2 \mapsto j_1 - i_1, i_1, k_1, i_2, k_2, j_2)$ | Coarse-grain | 1 |

[a]diagonal and bottom-up refer to how the inner triangles are filled up
[b]Parallel dimension is 0 based

### 3.1.3 Insights from Phase I

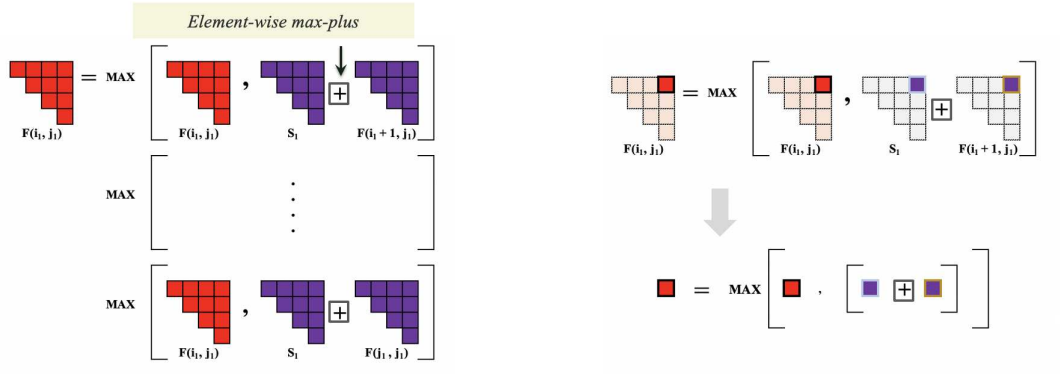
We are able to achieve significant performance improvement for $R_0$ from Phase I. However, we notice a significant collapse in performance when the input sequences are longer. It highlights the possibility of further improvements of $R_0$ beyond loop permutations.

## 3.2 Phase II

In this phase, we find a complete schedule for BPMax that enables automatic vectorization for all the variables and estimate the other reduction term's overhead. Then, we optimize the other reduction terms $R_{1-4}$, parallelize BPMax using coarse and fine-grain schedule, and also explore tiling of the double max-plus operation.

Previously, we noticed that the double-max plus reduction needs to access the $F$-table elements only. However, the other reductions have additional dependencies on either $S^{(1)}$ or $S^{(2)}$. $R_1$, $R_2$ depend on $S^{(2)}$ and $R_3$, $R_4$ depend on $S^{(1)}$. $S^{(1)}$ and $S^{(2)}$ have smaller complexities ($\Theta(M^3)$ and $\Theta(N^3)$, where $M$ and $N$ are the lengths of the two sequences) compare to the other reduction terms. We generate these two tables before filling up the $F$-table.

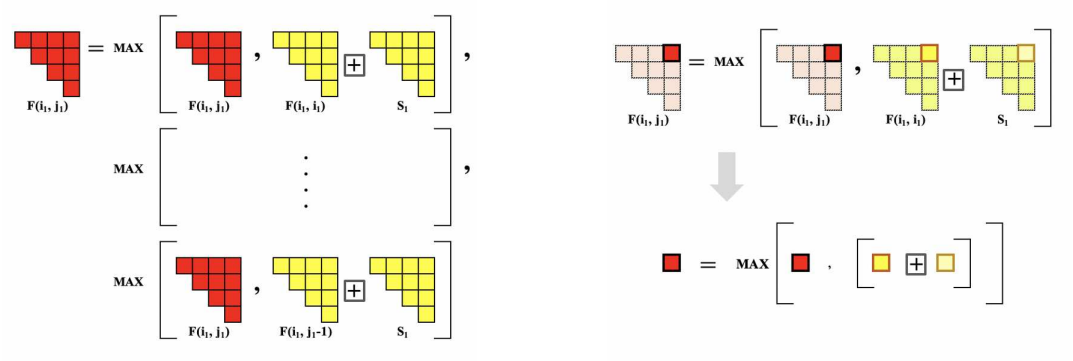
For BPMax optimization, we apply the same schedules for $R_0$ used in the previous phase. Next, we choose the $R_3$ and $R_4$. These two reductions have complexities of $\Theta(M^3N^2)$, which is significantly lower than the complexity of $R_0$ ($\Theta(M^3N^3)$). However, $R_3$ and $R_4$ use the same amount of data as $R_0$ but perform fewer operations. So, they need to be optimized. Figure 3.1 highlights that $R_3$ requires the same inner triangles towards the south of $F(i_1, j_1)$ and $S^{(1)}$, whereas $R_4$ requires the same inner triangles towards the west of $F(i_1, j_1)$ and $S^{(1)}$. To take advantage of the re-use, we can decompose the $R_3$ similar to $R_0$ as a set of max-plus operations between $F(k_1 + 1, j_1)$ and $S^{(1)}$ highlighted in Figure 3.5. However, it is an element-wise operation instead of the matrix product-like operations observed in $R_0$. Similarly, $R_4$ can also be expressed as a set of element-wise max-plus operations between $S^{(1)}$ and $F(i_1, k_1)$ highlighted in Figure 3.6. With this approach, $R_3$ and $R_4$ use the same first three dimensions of the schedule as $R_0$. The inner two dimensions of the schedule determine how the element-wise operations are computed.




**(a)** First level Decomposition

**(b)** Second level Decomposition

**Figure 3.5:** Decomposition of $R_3$ computation for an inner triangle




**(a)** First level Decomposition

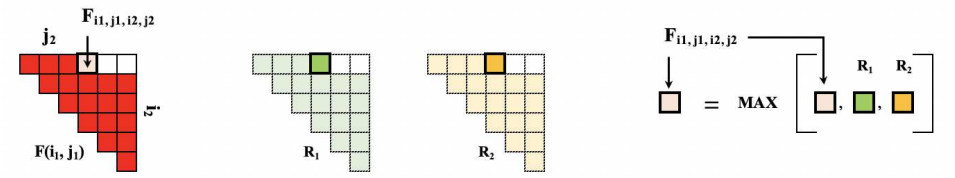
**(b)** Second level Decomposition

**Figure 3.6:** Decomposition of $R_4$ computation for an inner triangle

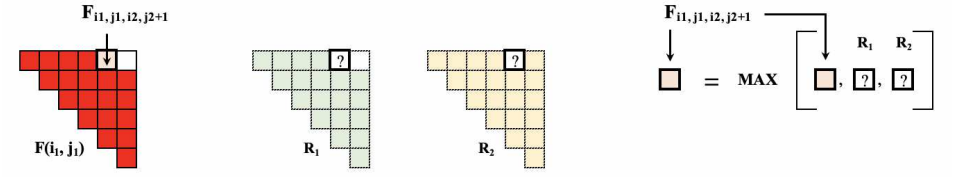
After accumulating all the results from $R_0$, $R_3$, and $R_4$ into $F(i_1, j_1)$, we update the elements of $F(i_1, j_1)$ using $R_1$ and $R_2$. The dependencies are now restricted to the current triangle. All the elements of the inner triangle except the first diagonal ($i_2 == j_2$) elements must evaluate $R_1$ and $R_2$. Then onwards, $R_1$ and $R_2$ are needed to be evaluated before updating $F_{i_1,j_1,i_2,j_2}$. These two reductions are dependent on the elements towards the south and west of $F_{i_1,j_1,i_2,j_2}$ present within $F(i_1, j_1)$. So, $F(i_1, j_1)$ can only be filled diagonally or bottom-up and then left to right. $F_{i_1,j_1,i_2,j_2}$ is updated after the $R_1$ and $R_2$ results are available. This process continues until all the elements of the inner triangle are updated. The inner three dimensions of the schedule for $R_1$ and $R_2$ can be $(j_2 - i_2, i_2, k_2)$, $(N - i_2, j_2, k_2)$, $(-i_2, j_2, k_2)$, $(j_2 - i_2, k_2, j_2)$, $(N - i_2, k_2, j_2)$, or $(-i_2, k_2, j_2)$ etc. We choose $j_2$ as the innermost loop nest to take advantage of the automatic vectorization. We ensure that $F$-table gets updated after $k_2$ reaches $j_2$. Figure 3.7 briefly highlights the final update sequence of an inner triangle.

### 3.2.1 Parallelization Approach

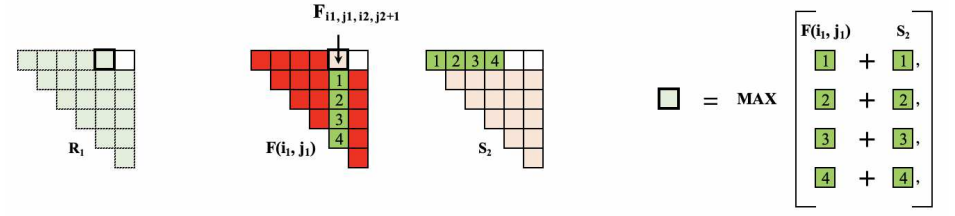
We observe that the coarse-grain parallelism works on all the BPMax reductions. But the fine-grain parallelism violates the dependency of the final $F$-table update, $R_1$, and $R_2$. Table 3.3 and 3.4 show various multi-dimensional optimized schedules.



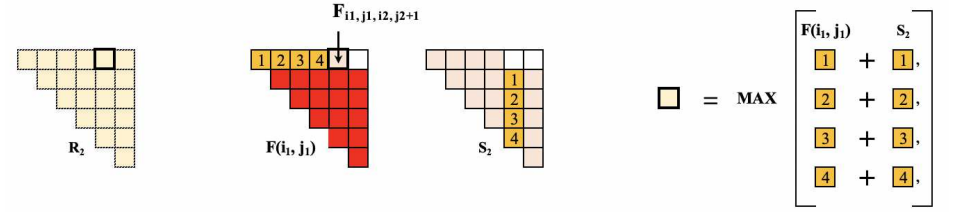

**(a)** Step 1: Let us assume that we are about to update the next element of $F(i_1, j_1)$: $F_{i_1,j_1,i_2,j_2}$. Results of $R_0$, $R_3$, and $R_4$ corresponding to all the elements of $F(i_1, j_1)$ are already accumulated in $F(i_1, j_1)$. Let us also assume that $R_1$ and $R_2$ are also computed for $F_{i_1,j_1,i_2,j_2}$ element. Thus, it gets updated with the maximum of the $F_{i_1,j_1,i_2,j_2}$, $R_1$, and $R_2$.



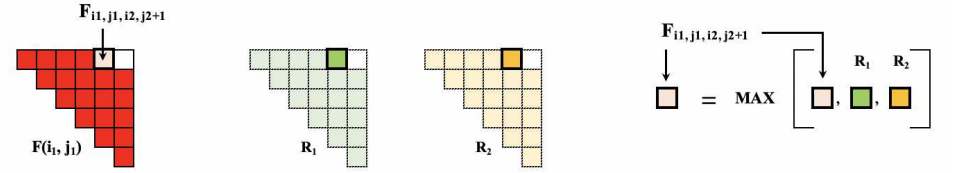

**(b)** Step 2: Next, we attempt to update $F_{i_1,j_1,i_2,j_2+1}$ highlighted in thick bordered light red box. $R_1$, and $R_2$ are not computed yet for $F_{i_1,j_1,i_2,j_2+1}$. Thus we need to compute these two reduction results before updating this point.




**(c)** Step 3: In this step, we compute $R_1$ for $F_{i_1,j_1,i_2,j_2+1}$. It requires all the $F(i_1, j_1)$-table elements towards the south of $F_{i_1,j_1,i_2,j_2+1}$ and all the $S^{(2)}$-table elements towards the west of the corresponding $S^{(2)}$-table element.




**(d)** Step 4: Now, we compute $R_2$ for $F_{i_1,j_1,i_2,j_2+1}$. It requires all the $F(i_1, j_1)$-table elements towards the west of $F_{i_1,j_1,i_2,j_2+1}$ and all the $S^{(2)}$-table elements towards the south of the corresponding $S^{(2)}$-table element.




**(e)** Step 5: We have all the reduction results available at this point to update $F_{i_1,j_1,i_2,j_2+1}$. Next, we compute the $R_1$ and $R_2$ for updating the next $F$-table entry. This process continues until all the elements are updated.

**Figure 3.7:** Illustration of $F$-table entry update with $R_1$ and $R_2$

**Table 3.3:** BPMAX FINE-GRAIN SCHEDULE

| ***Variable*** | ***Schedule***[a] |
|---|---|
| $S^{(1)}, S^{(2)}$ | $(i_1, j_1 \mapsto 0, 0, 0, 0, j_1 - i_1, i_1, 0, 0)$ |
| $F$ | $(i_1, j_1, i_2, j_2 \mapsto 1, -i_1, j_1, j_1, -i_2, 0, j_2, 0)$ |
| $R_1, R_2$ | $(i_1, j_1, i_2, j_2, k_2 \mapsto 1, -i_1, j_1, j_1, -i_2, 0, k_2, j_2)$, |
| $R_0$ | $(i_1, j_1, i_2, j_2, k_1, k_2 \mapsto 1, -i_1, j_1, k_1, -1, -i_2, k_2, j_2)$ |
| $R_3, R_4$ | $(i_1, j_1, i_2, j_2, k_1 \mapsto 1, -i_1, j_1, k_1, -1, -i_2, i_2, j_2)$ |

[a]Parallel dimension 5

**Table 3.4:** BPMAX COARSE-GRAIN SCHEDULE

| ***Variable*** | ***Schedule***[a] |
|---|---|
| $S^{(1)}, S^{(2)}$ | $(i_1, j_1 \mapsto 0, j_1 - i_1, i_1, 0, 0, 0, 0)$ |
| $F$ | $(i_1, j_1, i_2, j_2 \mapsto 1, j_1 - i_1, i_1, j_1, -i_2, j_2, j_2$ |
| $R_1, R_2$ | $(i_1, j_1, i_2, j_2, k_2 \mapsto 1, j_1 - i_1, i_1, j_1, -i_2, k_2, j_2)$ |
| $R_0$ | $(i_1, j_1, i_2, j_2, k_1, k_2 \mapsto 1, j_1 - i_1, i_1, k_1, i_2, k_2, j_2)$ |
| $R_3, R_4$ | $(i_1, j_1, i_2, j_2, k_1 \mapsto 1, j_1 - i_1, i_1, k_1, i_2, i_2, j_2)$, |

[a]Parallel Dimension 2

### 3.2.2 Tiling $R_0$

The fine-grain parallelism for the $R_0$ assigns one or more rows to each thread. Processing each row needs to access one complete inner triangle below that row before moving to the next. It motivates us to tile computations of one matrix instance of max-plus operation. It is a matrix

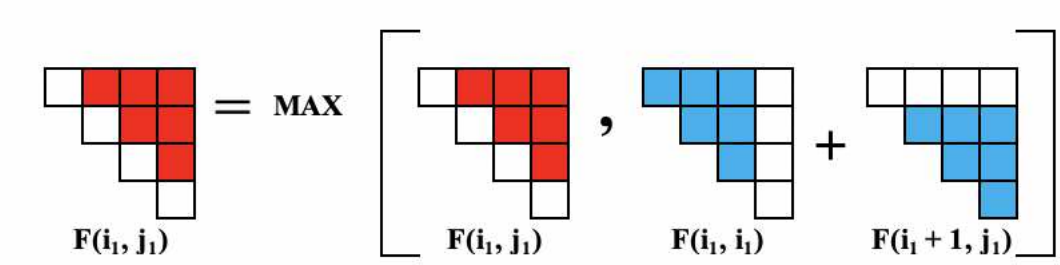



**Figure 3.8:** A matrix instance of max-plus operation

product-like computation, except only a fraction of work is being done here, and the access pattern

is imbalanced. We tile the three inner dimensions with $k_2$ loop still in the middle and $j_2$ loop as the innermost. So, this chops $(i_2, k_2, j_2)$ iteration space, and we parallelize the outer $i_2$ dimension.

### 3.2.3 Insights from Phase II

Loop permutations with automatic vectorization provide a significant speedup for the entire BPMax program. However, we are not able to apply fine-grain parallelization to all the reduction variables. We are successfully able to tile the double max-plus reduction. But, ALPHAZ produces in-efficient code when the entire BPMax program is tiled for the double max-plus computation.

## 3.3 Phase III

In this phase, we handle the load imbalance between the threads and partially ($R_0$, $R_3$, $R_4$) apply tiling to BPMax.

### 3.3.1 Parallelization Approach

From the dependence analysis, it can be seen that the BPMax will quickly become DRAM-bound for the coarse-grain schedule since each thread computes an inner triangle. But, it allows us to parallelize all the reduction operations. On the other hand, fine-grain parallelization can be applied to $R_0$, $R_3$, and $R_4$, reducing the data movement between DRAM and last level caches (LLCs). However, $R_1$ and $R_2$ are not easy to parallelize using fine-grain parallelization. These

**Table 3.5:** BPMAX HYBRID SCHEDULE

| ***Variable*** | ***Schedule***[a] |
|---|---|
| $S^{(1)}, S^{(2)}$ | $(i_1, j_1 \mapsto 0, 0, 0, j_1 - i_1, i_1, 0, 0, 0)$ |
| $F$ | $(i_1, j_1, i_2, j_2 \mapsto 1, j_1 - i_1, M, 0, i_1, -i_2, j_2, 0$ |
| $R_1, R_2$ | $(i_1, j_1, i_2, j_2, k_2 \mapsto 1, j_1 - i_1, M, 0, i_1, -i_2, k_2, j_2)$ , |
| $R_0$ | $(i_1, j_1, i_2, j_2, k_1, k_2 \mapsto 1, j_1 - i_1, i_1, k_1, i_2, k_2, j_2, 0)$, |
| $R_3, R_4$ | $(i_1, j_1, i_2, j_2, k_1 \mapsto 1, j_1 - i_1, i_1, k_1, i_2, i_2, j_2, 0)$, |

[a]Parallel Dimension 4

are optimum string parenthesization (OSP)-like computations that require further transformation like middle serialization. If we use the fine-grain schedule without such transformation, only one thread stays active during the processing of the two inner-reductions - $(R_1, R_2)$, leading to lower CPU resource utilization. We take advantage of the best of both worlds. We use the fine-grain parallelism for $R_0$, $R_3$, $R_4$ and the coarse-grain parallelism for $F$-table, $R_1$, $R_2$. We call this hybrid schedule shown in Table 3.5. We expect this to improve the CPU utilization and limit the data movement between DRAM and LLCs. However, there are limitations of this approach discussed in the result section.

### 3.3.2 Tiling Integration and Subsystem Scheduling

ALPHAZ produces inferior code when the tiling is applied to a subset of reduction operations.

**Table 3.6:** BPMAX HYBRID SCHEDULE WITH TILING

| | ***Variable*** | ***Schedule*** |
|---|---|---|
| a | Subsystem output | $(i_1, j_1 \mapsto M, i_1, j_1, 0)$ |
| | $R_0$ | $(i_1, j_1, k_1, k_2 \mapsto k_1, i_1, k_2, j_1)$, |
| | $R_3, R_4$ | $(i_1, j_1, k_1 \mapsto k_1, i_1, i_1, j_1)$, |
| b | Subsystem Call | $(i_1, j_1 \mapsto 1, j_1 - i_1, i_1, j_1 - 4, 0, 0, 0)$ |
| | $F$ | $(i_1, j_1, i_2, j_2 \mapsto 1, j_1 - i_1, M, i_1, -i_2, j_2, 0)$ |
| | $R_1, R_2$ | $(i_1, j_1, i_2, j_2, k_2 \mapsto 1, j_1 - i_1, M, i_1, -i_2, k_2, j_2)$ , |

a - Subsystem schedule(parallel dimension 1)
b - Root system schedule(parallel dimension 3)

It is due to the insertion of additional schedule dimensions needed to isolate the tiling band. So, we use ALPHA subsystem, which partitions BPMax computation into two systems. The subsystem produces an inner triangle using $R_0$, $R_3$, and $R_4$. Now, the primary system produces $R_1$, $R_2$, consolidates the results from the subsystem, and computes the final $F$-table output. It allows us to modularize the program and apply tiling transformation on $R_0$, $R_3$, and $R_4$ efficiently. The subsystem gets called for each instance of an inner $F$-table update. Finally, *use equation* construct

integrates these two systems. We invoke the subsystem call for each instance of the iteration space defined by the schedule's first two dimensions. Now, this requires us to specify the schedule for the subsystem invocation. Both systems are integrated manually. We perform minimal preprocessing since our code generator can not produce tiled code for the subsystem automatically. Few lines of source code changes are made to achieve this. Table 3.6 summarizes the complete schedule for the two systems.

## 3.4 Memory Optimization

The memory-overhead of our ALPHAZ generated code is $M^2 \times N^2$. However, we only need one-fourth of that memory. Even though it seems inefficient, the unused elements are never moved between the memory hierarchies. Reduction variables also take up memory space by default, which is wasteful. Without any memory optimization, coarse-grain parallelization requires $P$ (number of threads) instances of a $2 - D$ array for each reduction variables to be active in memory. Fine-

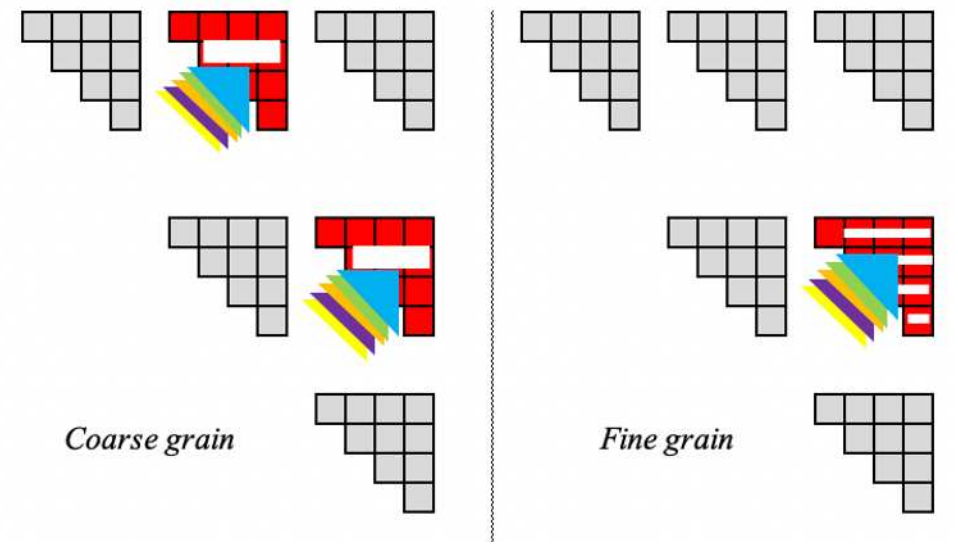



**Figure 3.9:** BPMax memory map without optimization

grain requires only a $2 - D$ array for each of the reduction variables illustrated in Figure 3.9. We almost eliminate the need for extra memory usage for the reduction variables using memory map

transformations. $R_0$, $R_3$, and $R_4$ are always computed before the final $F$-table update in all of our schedules, So, we use memory map transformations for $R_0$, $R_3$ and $R_4$ to share the memory

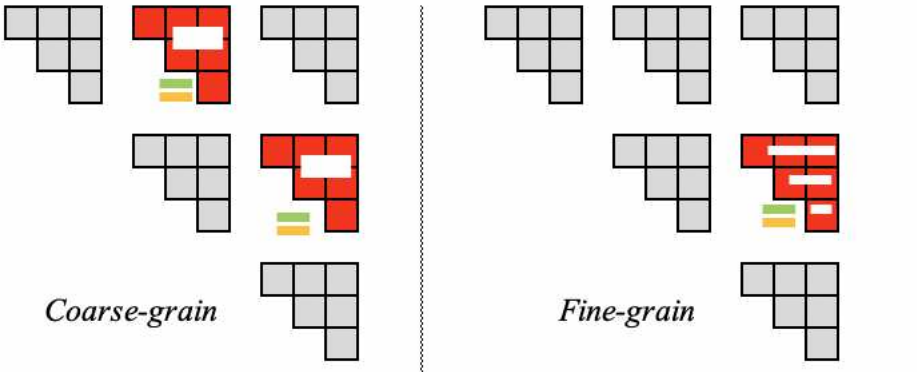



**Figure 3.10:** BPMax optimized memory map

with $F$-table. For the inner reductions - $R_1$ and $R_2$, our schedules update the final $F$-table entry bottom-up and then left to right where $j_2$ is the innermost loop. So, it accumulates intermediate results along a row. Thus, only one row of an inner triangle is required for $R_1$ and $R_2$ to keep up with the $F$-table update highlighted in Figure 3.10. Also, invocation of the subsystem calls creates new variables and copies data around by default. We optimize these redundant data copies using subsystem-related memory transformation.

## 3.5 Performance Tuning

We have attempted tuning various parameters such as tile size, OMP schedule, and memory maps to improve the performance. To find an optimum tile size, we started with cubic tiles and then adjusted the size of one or more dimensions to find a tile size that works moderately well across various inputs. However, we noticed that rectangular tiles work better than cubic tiles. We experimented the effect of OMP static, dynamic, and guided schedule and found that the OMP dynamic-schedule works better than the static and guided-schedule due to an imbalanced workload. We also performed some manual memory optimizations. Scheduling transformation initializes memory for each reduction body (corresponding to a variable), but when one variable shares

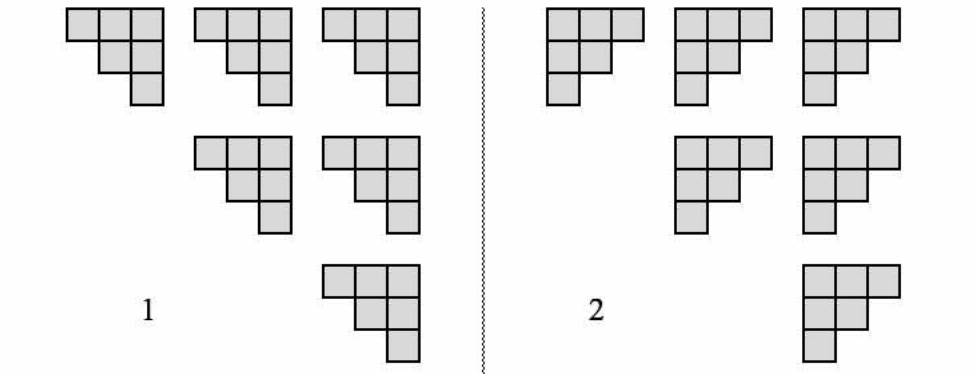



**Figure 3.11:** Memory mapping schemes

memory space with multiple variables, memory initialization becomes redundant. The current code generator does not optimize it. We comment out these macros, which attempt duplicate initialization to eliminate redundancies. We have tried two different memory transformations for the inner triangles highlighted in Figure 3.11 - 1 : $(i_2, j_2 \mapsto i_2, j_2)$ and 2 : $(i_2, j_2 \mapsto i_2, j_2 - i_2)$ and found that the option-1 always performs better.

## 3.6 Validation of Program Correctness

In addition to checking the final scores between reference and optimized version, we use AlphaZ toolset to verify the correctness of the optimized program using a verifier that compares the outputs of a schedule code with the sequential code. However, this is not sufficient due to the max-plus operation. We have developed a parallel version of the BPMax program to replace the max-plus with the plus-plus operation and apply the same sequence of transformations and run it through the verifier to ensure that our transformations and schedules do not change the semantics of the original program. One major challenge was running the validation on the longer sequences since the base, and sequential implementations are extremely slow.

# Chapter 4

# Results

We use Xeon E-2278G and Xeon E5-1650v4 to present the results of our approach. The CPU properties are highlighted in Table 4.1. One of the main differences between these two architectures is the number of available floating-point addition units (FPA) per core. Even though both of these architectures have two floating-point multiply-add (FMA) units, Broadwell has only one floating-point add unit, whereas Coffee Lake has two floating-point add units. Since the max operation is also executed using the FPA unit, Broadwell architecture is significantly bottle-necked for the max-plus computation.

**Table 4.1:** CPU PARAMETERS OVERVIEW

| Parameters | Xeon E5-1650v4 | Xeon E-2278G |
|---|---|---|
| Micro-Architecture | Broadwell | Coffee Lake |
| Number of cores | 6 | 8 |
| Number of threads | 12 | 16 |
| Base Frequency (GHz) | 3.6 | 3.4 |
| Turbo Frequency (GHz) | 4.0 | 5.0 |
| L1 Cache (KB) - Per Core | 32 | 32 |
| L2 Cache (KB) - Per Core | 256 | 256 |
| L3 Cache (MB) - Shared | 15 | 16 |
| DRAM (GB) | 16 | 32 |

Our optimized BPMax uses single-precision floating point. We measure the performance by calculating the number of single-precision floating-point operations executed per second (FLOPS). Now, GFLOPS indicates $10^9$ floating-point operations per second. Although it is typically used to highlight the double-precision performance, we implicitly use it in the BPMax optimization discussion to highlight the single-precision performance.

## 4.1 Max-plus Machine Peak Analysis

We calculate the theoretical CPU machine peak using the following equation:

$$\begin{aligned}\text{Single-precision Machine Peak (GFLOPS)} = \text{Number of Cores} \times \text{Core Frequency (GHz)} \times \\ \text{Number of vector operations} \times \\ \text{Instructions per cycle}\end{aligned} \tag{4.1}$$

Max-plus computation involves an addition and a max operation. Based on the Intel Intrinsics Guide [18], the number of instructions per cycle for both addition and max operation is 1 for Broadwell and 2 for Coffee Lake. These machines have AVX-256, so each core can use 8 AVX (Advanced vector extensions) registers simultaneously to perform eight single-precision operations. Table 4.2 highlights the theoretical machine peak for both of these platforms. The theoretical max-plus machine peak of Coffee Lake and Broadwell are $435.2$ and $172$ GFLOPS, respectively, when the cores run at the base frequency. We have observed that the processors run close to the base frequency during our experiments. Thus, we use these numbers as the theoretical machine peak.

**Table 4.2:** MAX-PLUS THEORETICAL MACHINE PEAK

| Processor | Number of Cores | Frequency (GHz) [Base, Turbo] | Instructions per cycle [add, max] | Machine Peak Single Core (GFLOPS) [Base, Turbo] | Machine Peak Total (GFLOPS) [Base, Turbo] |
|---|---|---|---|---|---|
| Xeon E5-1650v4 | 6 | [3.6, 4.0] | [1, 1] | [28.8, 32] | [172.8, 192] |
| Xeon E-2278G | 8 | [3.4, 5.0] | [2, 2] | [54.4, 80] | [435.2, 640] |

**Arithmetic Intensity of BPMax:** BPMax computation can be summarized as $Y = max(a + X, Y)$. It uses three single-precision memory accesses to perform two arithmetic operations (max and plus). So, its arithmetic intensity (AI) is $\frac{2}{(3\times4)}$or $\frac{1}{6}$. We use AI to determine the peak perfor-

mance based on the maximum L1 and L2 bandwidths using the roof-line analysis. These bandwidths are typically the average throughput of a long sequence of load/store operations from one level of cache to the other. We compute the memory bandwidth of any two levels of cache - $L_x$ and $L_y$ using Equation 4.2 for all the cores, where $L_y$ is closer to the core. It refers to $L_x$ bandwidth to $L_y$. With this notion, registers are considered as the last level of cache. We use maximum bandwidth specification to compute the peak performance in our roof-line model.

$$\begin{aligned}\text{Total } L_x \text{ bandwidth to } L_y = {} & \text{Number of Cores} \times \text{Core Frequency (GHz)} \\ & \times \text{Latency to move the data from } L_x \text{ to } L_y \text{ in bytes/cycle} \\ & \times \frac{10^9}{2^{30}} \text{GB/second}\end{aligned} \tag{4.2}$$

**Coffee Lake roofline:** Intel® micro-architecture specification from wiki-chip [17] indicates that the maximum L1 and L2 data cache bandwidths of Coffee Lake are $96$ bytes/cycle and $64$ bytes/-cycle, respectively, whereas L3 bandwidth and DRAM bandwidths are $32$ bytes/cycle and $39.74$ GB/second, respectively. Table 4.3 highlights the total memory bandwidth of different levels of caches for Coffee Lake using Equation 4.2. Based on this data, we have plotted the Coffee Lake roofline model shown in Figure 4.1. BPMax's arithmetic intensity of $\frac{1}{6}$ corresponds to the second

**Table 4.3:** Xeon E-2278G (Coffee Lake) Peak Memory Bandwidth

| Memory | Peak Bandwidth (per core) (bytes/cycle) | Max Memory Bandwidth (1 core) Max (GB/s) | Max Memory Bandwidth (8 cores) (GB/s) |
|---|---|---|---|
| L1 Bandwidth to Register | 96 | 303.98 | 2431.8 |
| L2 Bandwidth to L1 | 64 | 202.65 | 1621.2 |
| L3 Bandwidth L2 | 32 | 101.32 | 810.6 |
| DRAM Bandwidth to L3 | - | 39.74 | 39.74 |

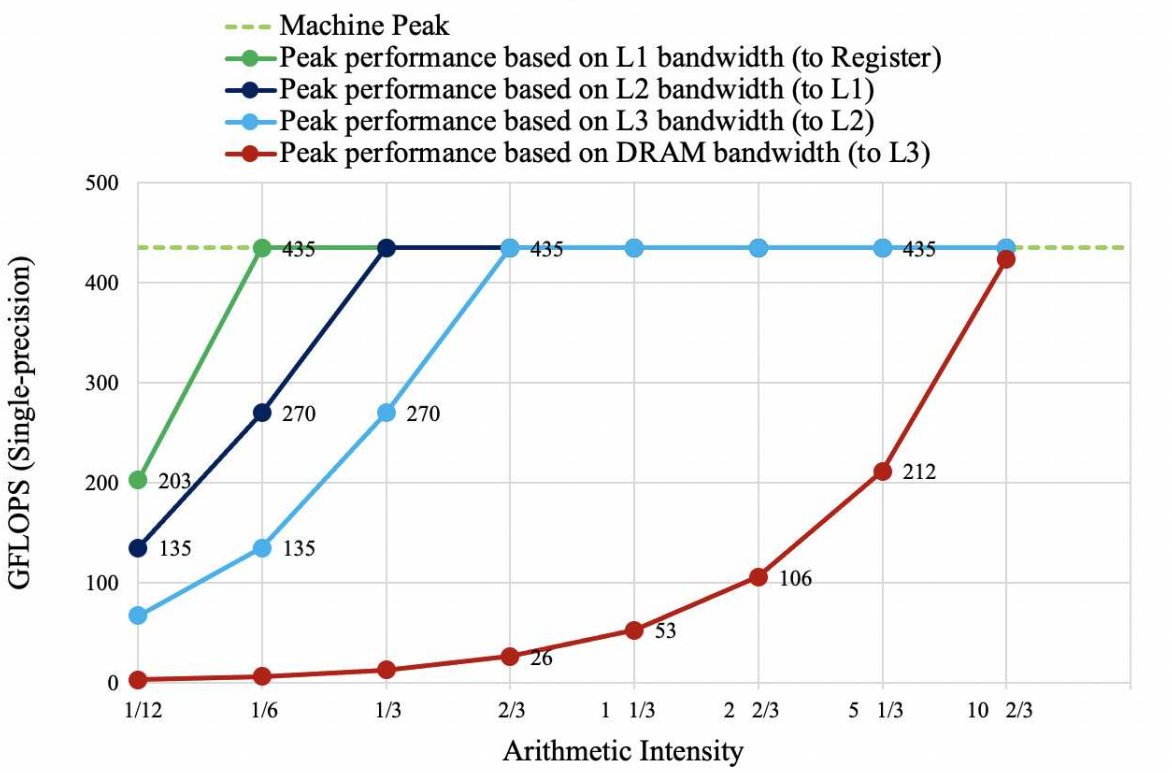



**Figure 4.1:** Xeon E-2278G (Coffee Lake) roofline for max-plus

data point on each series shown in Figure 4.1. Max-plus performance based on the maximum L1 and L2 bandwidths are 435 GFLOPS, 270 GFLOPS, respectively.

**Broadwell roofline:** Figure 4.2 presents the Broadwell roofline model using a similar analysis. Table 4.4 highlights the total memory bandwidth of different levels of caches for Broadwell using Equation 4.2. The roofline shows that the max-plus performance based on the maximum L1 and L2 bandwidths are 172 GFLOPS.

**Table 4.4:** Xeon E5 1650v4 (Broadwell) Peak Memory Bandwidth

| Memory | Peak Bandwidth (per core) (bytes/cycle) | Max Memory Bandwidth (1 core) Max (GB/s) | Max Memory Bandwidth (6 cores) (GB/s) |
|---|---|---|---|
| L1 Bandwidth to Register | 96 | 321.86 | 1931.1 |
| L2 Bandwidth to L1 | 64 | 214.57 | 1287.4 |
| L3 Bandwidth L2 | 32 | 107.28 | 643.7 |
| DRAM Bandwidth to L3 | - | 76.8 | 76.8 |

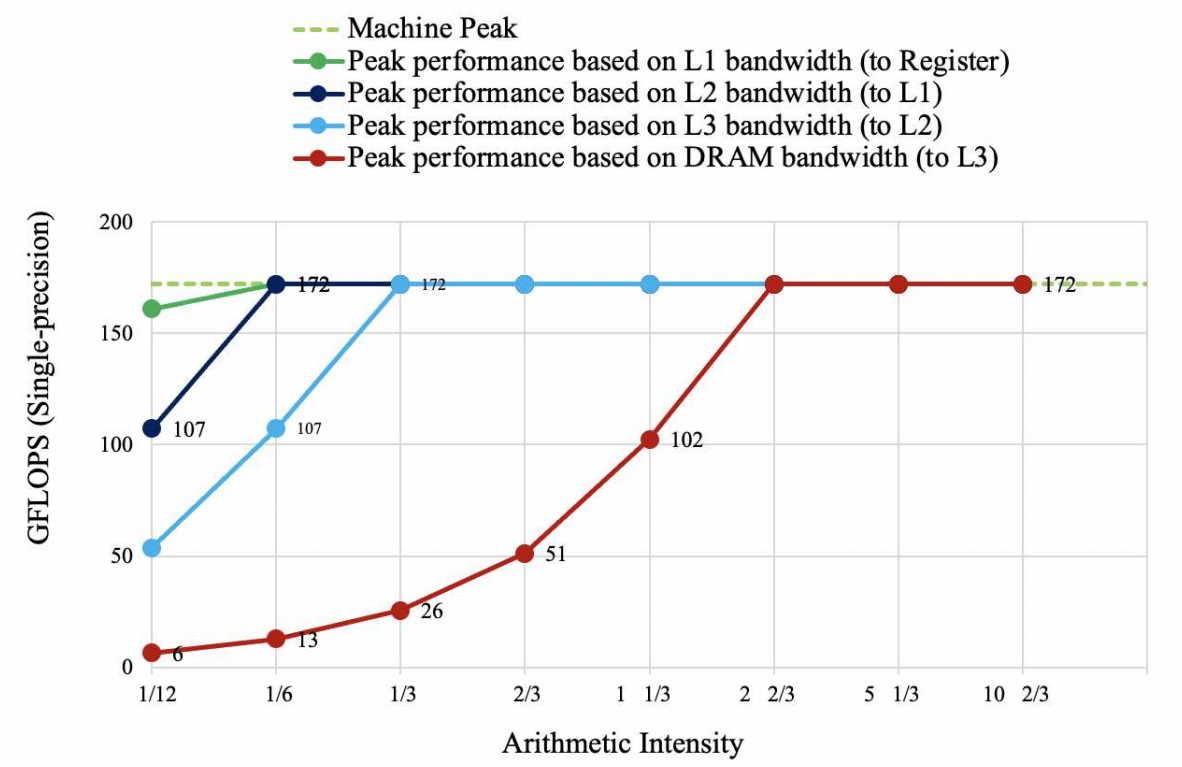



**Figure 4.2:** Xeon E5 1650v4 (Broadwell) roofline for max-plus

## 4.2 Performance Analysis of Double Max-plus Computation

In this subsection, we go over the results of our optimization approach for double max-plus computation. First, we discuss the double max-plus performance on a single core of Xeon E5-1650v4 (Broadwell) and Xeon E-2278G (Coffee Lake) highlighted in Figure 4.3. We use the fine-grain bottom-up schedule that uses loop permutation and the tiled version of the fine-grain diagonal schedule to compare the single-core performances. Figure 4.3 shows the performance of the double max-plus computation with these two versions of the code, when $M$ is fixed to $32$ and $N$ is varied between $750$ to $4000$. We have not presented the base version since it only attains a tiny fractional GFLOPS performance. The tiled version of the program highlighted by the dotted line reaches more than $50\%$ of the machine peak on both Broadwell (dotted blue line) and Coffee Lake (dotted dark-green line). But the loop permuted version that uses the fine-grain schedule attains only about $45\%$ and $30\%$ of the max-plus machine peak on Broadwell (blue line) and Coffee Lake (dark-green line). We observe the performance drop on both of the machines when $N$ exceeds $2500$. For this schedule, the maximum amount of memory required for each $k_2$ loop is

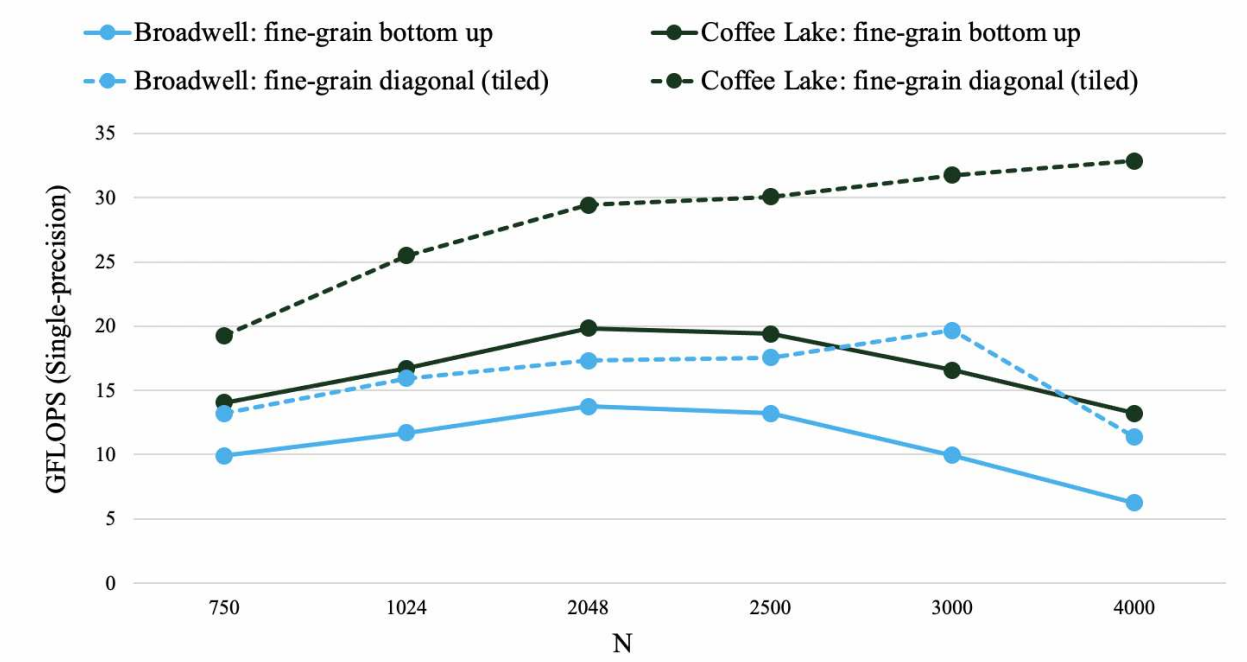



**Figure 4.3:** Double max-plus single core performance comparison on Coffee Lake and Broadwell

approximately one row of the inner-triangle of size $N$ for the result, one row of the inner-triangle of size $N$ for one of the operands, and an entire inner triangle of size $\frac{N^2}{2}$ for the other operand. So, the total size is $2 \times N + \frac{N^2}{2}$. For a 16 MB L3 cache, the value of N is approximately 2895. Therefore this schedule works better up to a sequence length of 2500. However, we observe the performance drops afterward. The tiling transformation improves locality and maintains the performance as $N$ increases in size. It also uses automatic vectorization. However, the tiled version of the program performs poorly on Broadwell when $N$ was larger than $3000$. The $F$-table footprint is very close to the DRAM capacity (16 GB) of Broadwell when $M = 32$, $N = 4000$, triggering swapping (disk-access) that reduces CPU utilization. Our tiling approach does not consider the tiling at the disk level. Thus, we restrict $N$ to $3000$ for the rest of our experiments.

Figure 4.4 and Figure 4.5 show the performance and speedup comparison of double max-plus between different schedules using eight threads on Coffee Lake. Figure 4.6 and Figure 4.7 present the same comparison using six threads on Broadwell. Performance of the code version with the original code achieves only about 1 GFLOPS highlighted in the dark red. We notice that the coarse-grain parallelization highlighted in the light red performs very poorly since it generates a lot of DRAM traffic, making the program slower. There is a minor difference between computing the inner triangles of $F$-table diagonally vs. bottom-up and left to right highlighted in orange and blue. In both cases, all the threads work on one inner triangle before moving to the next. The black color represents the performance corresponding to the tiled version. The tiled version of the code performs better than the non-tiled fine-grain version of the program and maintains the same performance with longer sequences. It attains a maximum performance of $187$ and $117$ GFLOPS on Coffee Lake and Broadwell, respectively. These correspond to a $223\times$ and $178\times$ improvement over the base implementation taken from the BPMax program. On Coffee Lake, it is a speedup improvement of more than $100\%$ over the basic loop-permuted version of the fine-grain schedule. We observe almost $200\%$ speedup improvement on Broadwell.

Figure 4.8 shows the effect of different tile sizes on double max-plus performance. We have chosen a fixed sequence length of $M = 16$ and $N = 2500$ to perform this experiment. We have

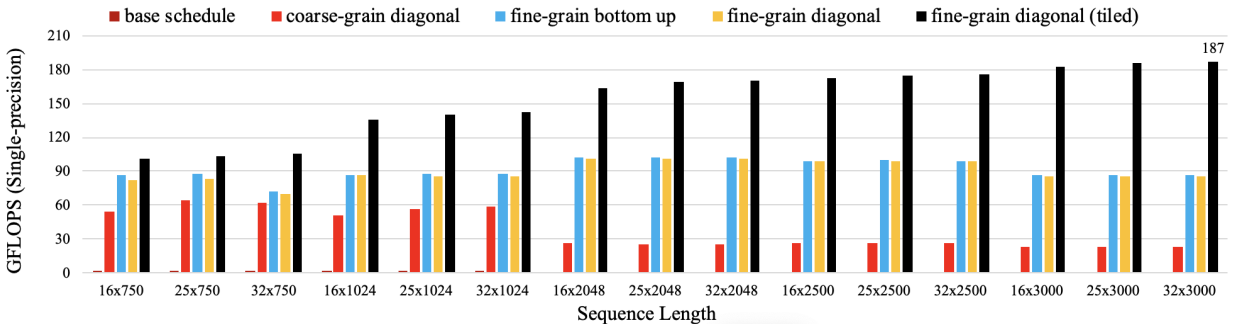



**Figure 4.4:** Double max-plus performance comparison on Coffee Lake

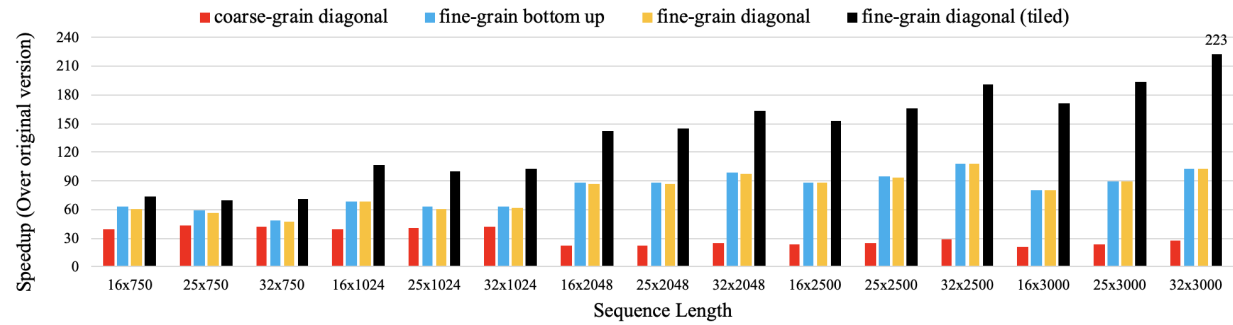



**Figure 4.5:** Double max-plus speedup comparison on Coffee Lake

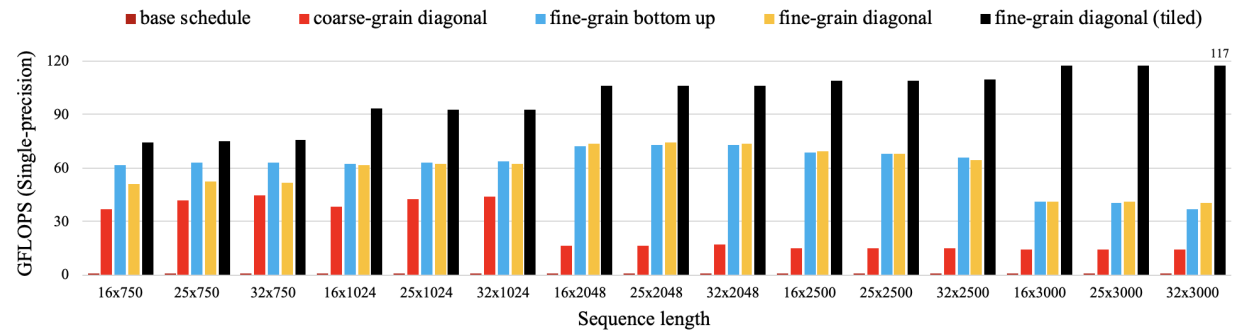



**Figure 4.6:** Double max-plus performance comparison on Broadwell

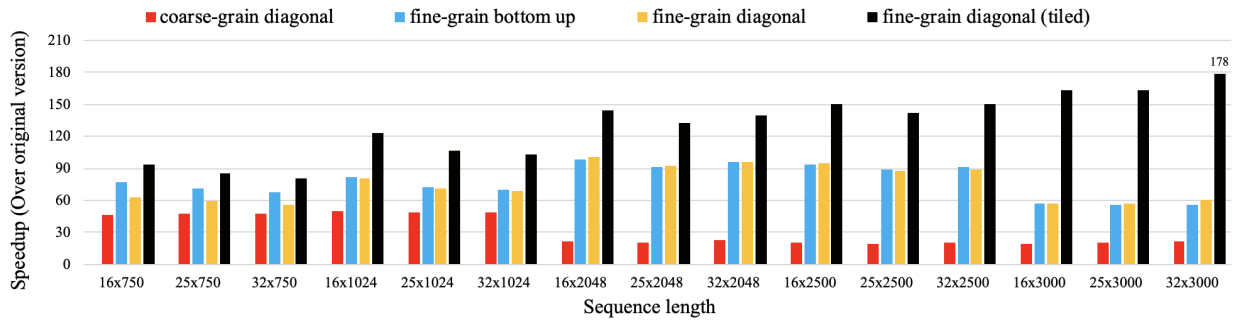



**Figure 4.7:** Double max-plus speedup comparison on Broadwell

explored different tile sizes ($i_2 \times k_2 \times j_2$) and found that the cubic tiles perform poorly. We get the best result when $j_2$ is not tiled, and the tile sizes of $i_2$ and $k_2$ are such that $tile\_size_{i_2} \times tile\_size_{k_2}$ fits in the L1 cache and $tile\_size_{k_2} \times j_2$ fits in the L2 cache. $tile\_size_{i2}$ is also dependent on the length of the sequence length $N$ since this determines the total workload for each core. Thus, the results shown in Figure 4.4, Figure 4.5, Figure 4.6, and Figure 4.7 use problem-specific best tile dimensions. Tile dimensions of $(32 \times 32 \times N)$ are used when $N \leq 2048$, $(32 \times 16 \times N)$ for $N = 2500$, and $(64 \times 16 \times N)$ when $N > 2500$. We have also experimented with hyper-threading and tiling on Coffee Lake and found that hyper-threading hurts the performance when tiling is applied. We observe a $(3 - 5\%)$ degradation with hyper-threading over eight threads highlighted in Figure 4.9.

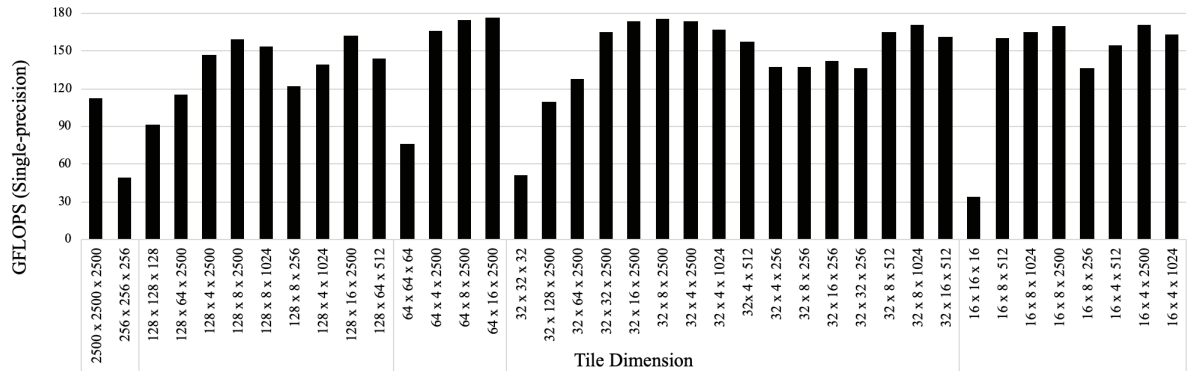



**Figure 4.8:** Effect of tiling parameters ($i_2 \times k_2 \times j_2$) on double max-plus performance (sequence length - 16 x 2500) on Coffee Lake

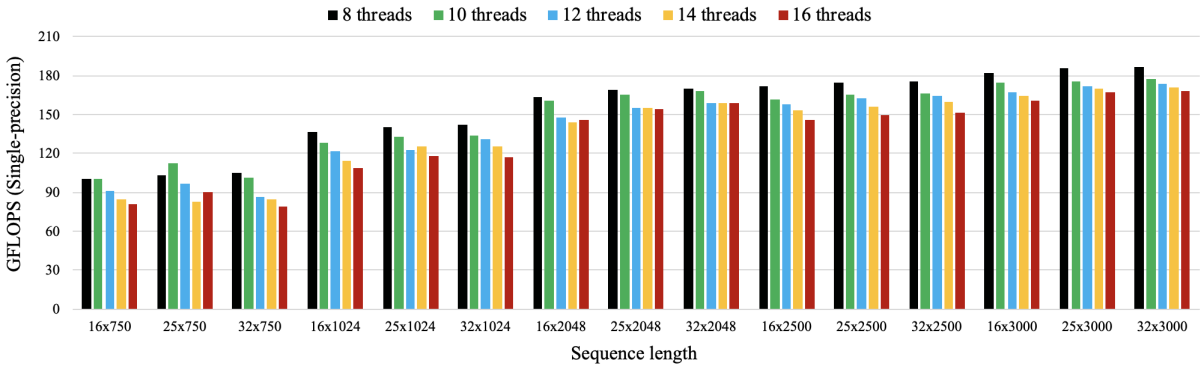



**Figure 4.9:** Effect of hyper-threading on tiled double max-plus performance on Coffee Lake

## 4.3 BPMax Performance Improvement

We have chosen three different values of $M$ (16, 25, 32) and five different values of $N$ (750, 1024, 2048, 2500, 3000) to measure the BPMax performance for each combination of $M$ and $N$. Figure 4.10 and 4.11 show the performance improvements and speedup of various versions of the BPMax program on Coffee Lake using $8$ threads. We use the original BPMax program as the reference since no better CPU version of the BPMax program is available. The coarse and fine-grain version of the program performs the worst, highlighted in light red and blue. As seen in the previous section, the coarse-grain schedule severely impacts double max-plus computation, affecting the overall performance. Fine-grain parallelism works better for $R_0$, $R_3$, $R_4$, but we cannot parallelize the $R_1$, and $R_2$ computations. The performance and speedup with the hybrid schedule are highlighted in green. It performs better than the coarse and fine-grain schedule. The tiled version of the hybrid schedule highlighted in dark blue performs best. It achieves $100\times$ speedup for longer sequence lengths.

The improvement for the tiled version mainly comes from the optimization of $R_0$, $R_3$, and $R_4$. On Coffee Lake, the tiled version of the program reaches around $127$ GFLOPS for small sequences and over $100$ GFLOPS for moderate to large-size sequences. It is about $25\%$ of the max-plus machine peak. But it is only $54\%$ of the double max-plus performance with the same input size. Our analysis shows that $R_3$ and $R_4$ are almost free since those get computed along with the $R_0$. However, the two $\Theta(M^2N^3)$ computations - $R_1$ and $R_2$ severely affect the overall performance. It is the effect of our schedule choice. Each thread is responsible for producing the final version of one inner triangle of $F$-table along with the $R_1$ and $R_2$. Both of these computations require most of the elements of one inner triangle of $F$-table and the $S^{(2)}$-table to compute one row for the worst case. So, the total amount of data required to process a row reaches about $\Theta(N^2)$, which is $16$ MB for an inner sequence of length $2048$. This issue gets amplified when we attempt hyper-threading (beyond $8$ threads).

Figures 4.12 and 4.13 show the BPMax performance and speedup improvement on Xeon E5-1650v4. We have used the same tile sizes as the Coffee Lake. We observe similar performance

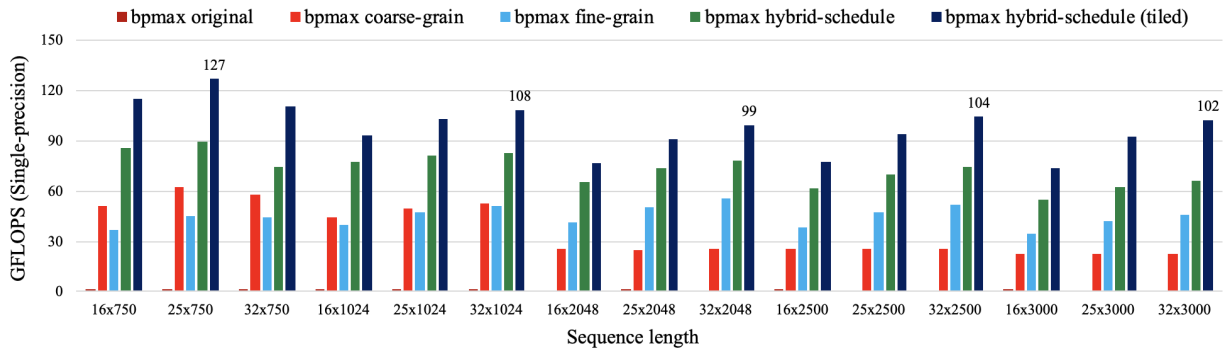



**Figure 4.10:** BPMax performance comparison on Coffee Lake

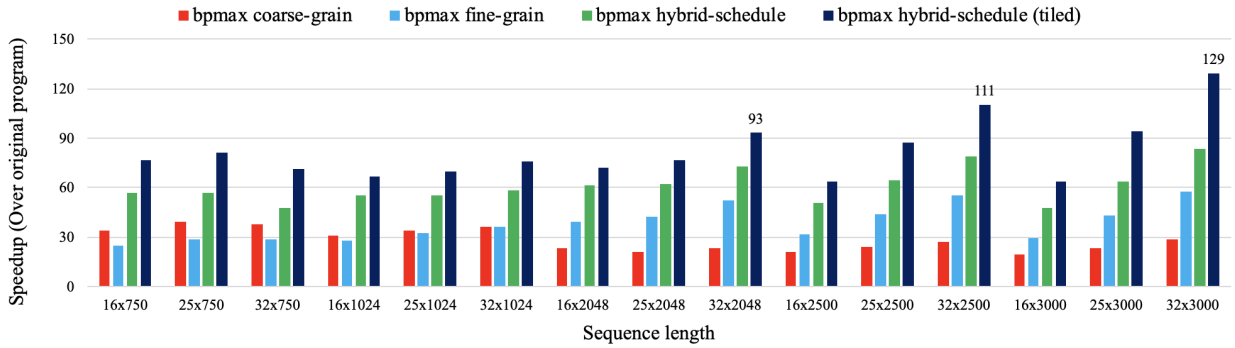



**Figure 4.11:** BPMax speedup comparison on Coffee Lake

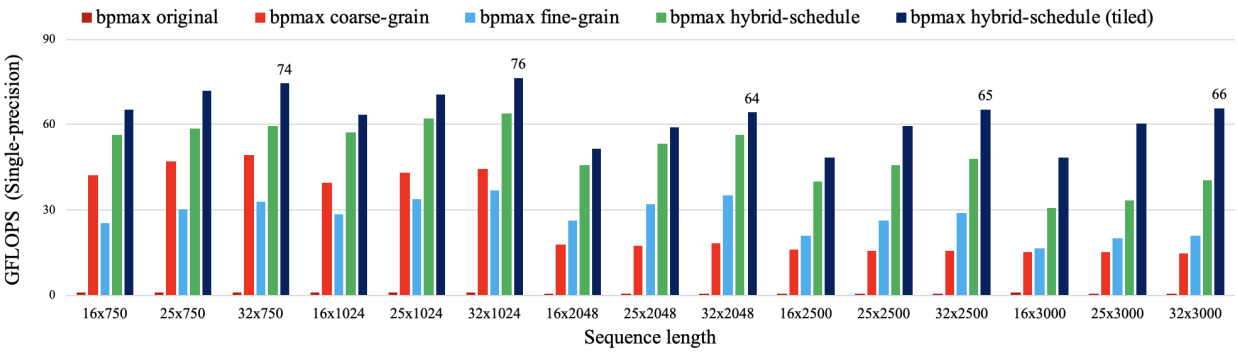



**Figure 4.12:** BPMax performance comparison on Broadwell

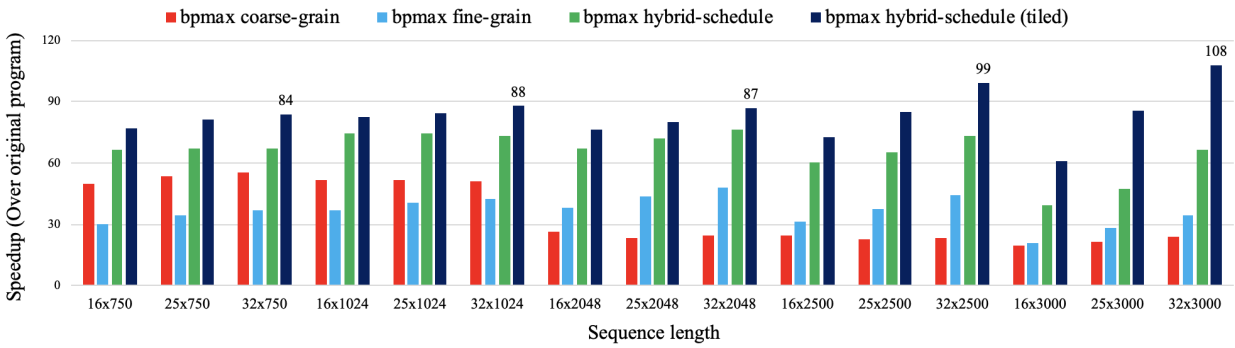



**Figure 4.13:** BPMax speedup comparison on Broadwell

characteristics as Coffee Lake for the different schedules. On this platform, the tiled version of the program reaches around $76$ GFLOPS for moderate-size sequences, which is about $45\%$ of the max-plus machine peak. It is only $60\%$ of the double max-plus version. However, it still achieves $100\times$ speedup.

## 4.4 Code Generation Metric

Table 4.5 shows the line of code (LOC) generated by ALPHAZ for different optimized programs. The original version of the BPMax program is hand-written and has $140$ lines of code. With the ALPHAZ optimization process, we observe an increase in LOC for all the different versions. The table also highlights the complexities (based on LOC) between the double max-plus computation and the BPMax program.

**Table 4.5:** AUTO-GENERATED CODE STATISTICS

| ***Implementation*** | LOC | a | b |
|---|---|---|---|
| BPMax base | 140 | 140 | NA |
| Double max-plus(coarse/fine) | 150 | None | 3 |
| BPMax coarse/fine/ hybrid | 1200 | None | 30 |
| BPMax hybrid with tiled | 1400 | <5 | 7 |

a - Hand written code
b - Macro replacement/Macro comment out

# Chapter 5

# Future Directions

In this work, we have demonstrated the optimization process of a complete RRI program using polyhedral transformations. We have explored different schedules, memory maps, and tiling transformation for our optimization work using the polyhedral code generator - AlphaZ . Our result shows that the tiling improves the performance of the most dominant part of the computation by two folds over a simple loop permutation that exploits auto-vectorization. We have achieved over $100\times$ improvements on CPU for the complete BPMax program with the tiling transformation applied to the outer reductions, including the most dominant part, and using different types of parallelizations on the reduction operations. However, the inner reductions are still inefficient, which limits the overall performance improvement. Also, the double max-plus operation remains bandwidth-bound even after the tiling transformation. It indicates that an additional tiling level at the register level is required to make the program compute-bound and improve performance further.

We envision that a register-tiled kernel and an extra level of tiling on the double max-plus computation will significantly improve the double max-plus performance. Besides optimizing the double max-plus operation, tiling of $R_1$ and $R_2$ are required to improve the overall BPMax performance. In the long term, it can also be beneficial to distribute the computation over a cluster using MPI (Message Passing Interface) program to take advantage of another level of parallelism. All these transformations remain a challenge for AlphaZ today. So, we also envision future work on AlphaZ to allow these advanced transformations.

Finally, we expect that similar polyhedral transformations can be easily applied to more complex RRI algorithms - like BPPart and piRNA using AlphaZ to generate optimized code and achieve significant speedup for them.

# Bibliography


[1] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation - PLDI '08*. ACM Press, 2008.

[2] Uday Bondhugula, J. Ramanujam, and P. Sadayappan. Pluto: A practical and fully automatic polyhedral program optimization system. 2015.

[3] Chun Chen, Jacqueline Chame, and Mary Hall. Chill: A framework for composing high-level loop transformations. Technical report, 2008.

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

[5] Ye Ding and Charles E Lawrence. A bayesian statistical algorithm for RNA secondary structure prediction. *Computers & Chemistry*, 23(3-4):387–400, jun 1999.

[6] F. Dupont de Dincehcin. *Systèmes structurés d'équations récurrentes : mise en œuvre dans le langage Alpha et applications*. PhD thesis, Université de Rennes, janvier 1997.

[7] F. Dupont de Dinechin, P. Quinton, and T. Risset. Structuration of the alpha language. In W.K Giloi, S. Jahnichen, and B.D. Shriver, editors, *Massively Parallel Programming Models*, pages 18–24. IEEE Conmputer Society Press, 1995.

[8] Ali Ebrahimpour-Boroojeny, Sanjay Rajopadhye, and Hamidreza Chitsaz. Bppart and bpmax: Rna-rna interaction partition function and structure prediction for the base pair counting model. apr 2019.

[9] Ali Ebrahimpour-Boroojeny, Sanjay Rajopadhye, and Hamidreza Chitsaz. Bppart: Rna-rna interaction partition function in the absence of entropy. WABI, 2021.

[10] de Dinechin F. and S. Robert. Hierarchical static analysis of structured systems of affine recurrence equations. In J. Fortes, C. Mongenet, K. Parhi, and V. Taylor, editors, *International Conference on Application Specific Systems Architectures and Processors (ASAP 96)*, pages 381–390. IEEE, August 1996.

[11] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, Feb 1991.

[12] Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part I. one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–347, 1992.

[13] Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part II. multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, 1992.

[14] Brandon Gildemaster, Prerana Ghalsasi, and Sanjay Rajopadhye. A tropical semiring multiple matrix-product library on GPUs: (not just) a step towards RNA-RNA interaction computations. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, may 2020.

[15] AC. Guillou, F. Quilleré, P. Quinton, S. Rajopadhye, and T. Risset. Hardware design methodology with the Alpha language. In *Forum on Design Languages*, Sept 2001.

[16] Fenix W. D. Huang, Jing Qin, Christian M. Reidys, and Peter F. Stadler. Partition function and base pairing probabilities for RNA–RNA interaction prediction. *Bioinformatics*, 25(20):2646–2654, aug 2009.

[17] Intel®. Intel microarchitectures. https://en.wikichip.org/wiki/intel/microarchitectures/.

[18] Intel®. Intel® intrinsics guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

[19] H. Le Verge. Reduction operators in alpha. In D. Etiemble and J-C. Syre, editors, *Parallel Algorithms and Architectures, Europe*, LNCS, pages 397–411. Springer Verlag, June 1992. See also, Le Verge Thesis (in French).

[20] H. Le Verge. *Un environnement de transformations de programmmes pour la synthèse d'architectures régulières*. PhD thesis, L'Université de Rennes I, Oct 1992.

[21] Junjie Li, Sanjay Ranka, and Sartaj Sahni. Multicore and GPU algorithms for nussinov RNA folding. In *2013 IEEE 3rd International Conference on Computational Advances in Bio and medical Sciences (ICCABS)*. IEEE, jun 2013.

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

[23] Stefanie A. Mortimer, Mary Anne Kidwell, and Jennifer A. Doudna. Insights into RNA structure and function from genome-wide studies. *Nature Reviews Genetics*, 15(7):469–479, may 2014.

[24] Ruth Nussinov, George Pieczenik, Jerrold R. Griggs, and Daniel J. Kleitman. Algorithms for loop matchings. *SIAM Journal on Applied Mathematics*, 35(1):68–82, jul 1978.

[25] Marek Palkowski and Wlodzimierz Bielecki. Tiling nussinov's RNA folding loop nest with a space-time approach. *BMC Bioinformatics*, 20(1), apr 2019.

[26] D. D. Pervouchine. Iris: intermolecular rna interaction search. *Genome informatics. International Conference on Genome Informatics*, 15 2:92–101, 2004.

[27] P. Quinton. Automatic synthesis of systolic arrays from recurrent uniform equations. In *11th Annual International Symposium on Computer Architecture, Ann Arbor*, pages 208–214, June 1984.

[28] P. Quinton. The systematic design of systolic arrays. In F. Fogelman Soulie, Y. Robert, and M. Tchuente, editors, *Automata Networks in Computer Science*, chapter 9, pages 229–260. Princeton University Press, 1987. Preliminary versions appear as IRISA Tech Reports 193 and 216, 1983, and in the proceedings of the IEEE Symposium on Computer Architecture, 1984.

[29] P. Quinton and V. Van Dongen. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI Signal Processing*, 1(2):95–113, 1989.

[30] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation - PLDI '13*. ACM Press, 2013.

[31] S. V. Rajopadhye. *Synthesis, Optimization and Verification of Systolic Architectures*. PhD thesis, University of Utah, Salt Lake City, Utah 84112, December 1986.

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

[33] S. V. Rajopadhye, S. Purushothaman, and R. M. Fujimoto. On synthesizing systolic arrays from recurrence equations with linear dependencies. In *Proceedings, Sixth Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 488–503. Springer Verlag, LNCS 241, December 1986.

[34] Guillaume Rizk, Dominique Lavenier, and Sanjay Rajopadhye. GPU accelerated RNA folding algorithm. In *GPU Computing Gems Emerald Edition*, pages 199–210. Elsevier, 2011.

[35] M Shel Swenson, Joshua Anderson, Andrew Ash, Prashant Gaurav, Zsuzsanna Sükösd, David A Bader, Stephen C Harvey, and Christine E Heitsch. GTfold: Enabling parallel RNA secondary structure prediction on multi-core desktops. *BMC Research Notes*, 5(1):341, 2012.

[36] Swetha Varadarajan. Polyhedral optimizations of RNA-RNA interaction computations. Master's thesis, Colorado State University, 2016.

[37] Swetha Varadarajan. A case study on RNA-RNA interaction application implementation using AlphaZ. In *Proceedings of the 4th ACM International Workshop on Real World Domain Specific Languages*. ACM, feb 2019.

[38] T. Yuki, G. Gupta, DG. Kim, T. Pathan, and S. Rajopadhye. AlphaZ: A system for design space exploration in the polyhedral model. In *Proceedings of the 25th International Workshop on Languages and Compilers for Parallel Computing*, September 2012.

[39] Michael Zuker and Patrick Stiegler. Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. *Nucleic Acids Research*, 9(1):133–148, 1981.