Accelerating the BPMax Algorithm for RNA-RNA Interaction

Chiranjeb Mondal
Department of Computer Science
Colorado State University
Fort Collins, CO, USA
chiranjeb.mondal@colostate.edu

Sanjay Rajopadhye
Department of Computer Science
Colorado State University
Fort Collins, CO, USA
sanjay.rajopadhye@colostate.edu

***Abstract*—RNA-RNA interactions (RRI) play an important role in various biological processes such as gene transcription, and are known to play a critical role in diseases such as cancer and Alzheimer's, necessitating efficient computational tools. To date, RRI programs like BPMax were developed and optimized by hand, and this is prone to human error, and costly to develop and maintain. Its high complexity ($\Theta(N^3M^3)$ in time and $\Theta(N^2M^2)$ in space) make it both essential and a challenge to parallelize it. In this paper, we present a parallelization of 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 employing a standard "diagonal-by-diagonal" execution order. We achieve 76 GFLOPS, which is about a fifth 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 117 GFPLOS. We do this with a polyhedral code generation tool, AlphaZ, that 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, and parallelization approaches, as well as tiling of the most dominant part of the computation.**

***Index Terms*—RRI, BPMax, polyhedral model, tiling**

I. 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 can provide valuable information about a biological function.

Researchers have long been studying these interactions and proposed different models. In 1978, Nussinov presented a model [1] that predicts secondary structure from single RNA folding. RNA-RNA interactions have been moved to the spotlight in biology since the mid-1990s with significant RNA interference discovery. Chitsaz et al. [2] 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, for sequences of length M and N. It uses 96 dynamic tables. However, running this compute and memory intensive program is extremely challenging. It takes days and months to get experimental results. So, Ebrahimpour-Boroojeny et al. [3] retreated from the slow comprehensive model and developed the BPPart for base-pair partition and BPMax for base-pair maximization. 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 performance as the input size grows.

Ebrahimpour-Boroojeny et al. conclude that BPMax [3] 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 [2], IRIS [4], RIP [5] follow similar recurrence patterns. So, besides the practical usefulness of BPMax, the learning and insights gleaned from this 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 highly optimized programs get generated from a simple correct program together with a set of performance tuning hints or directives.

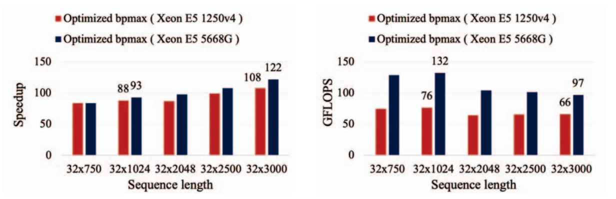
Fortunately, RRI algorithms fit the requirements of the *polyhedral model* [6]–[10] - a mathematical formalism that allows for just such program transformations. The polyhedral compilation has been the subject of intense research for 35 years, and yet, even a state-of-the-art polyhedral tool like PLUTO [11], [12] does not yield satisfactory performance [13]. Many of the performance optimization strategies need some careful thought by an expert. This gap can be bridged by a tool that allows semi-automatic transformation like Chill [14]. At CSU, we are

978-1-6654-3577-2/21/$31.00 ©2021 IEEE
DOI 10.1109/IPDPSW52791.2021.00042

developing and working with a similar tool, ALPHAZ [15], that operates at a higher level of abstraction.

The paper makes the following contributions:

- This 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 using polyhedral transformations that achieves over $100\times$ speedup over the original program shown in Figure 1. The most compute-intensive part of the BPMax achieves a $170\times$ speedup over the original implementation, a $1.5\times$ - $2\times$ improvement over a similar kernel optimized previously.
- The compilation scripts and optimized version of the BPMax program are available in GitHub public repository [16].



(a) Speedup over base program (b) Single-precision performance

Fig. 1: Summary of the optimization results

The paper proceeds as follows. We first set up the context and background of our work to highlight the BPMax algorithm, discuss the polyhedral model's role, and application of ALPHAZ . Then, we discuss multiple phases of the optimization process, the rationale behind different schedules, processor allocations, and memory mappings. Finally, we go over our performance results and conclude the paper with challenges and future directions.

II. RELATED WORK

There was no previous example of significant success of RRI optimization using polyhedral compilation to the best of our knowledge. Palkowski et al. [17] have used the polyhedral model to optimize Nussinov's algorithm [1]. However, it is related to single RNA strand folding only.

Varadrajan [13], [18] 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 $100\times$ speedup.

Glidemaster [19] 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 speedup 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.

III. BACKGROUND

This section highlights the BPMax algorithm, summarizes the polyhedral model, and then describes the code-generation technique using ALPHAZ .

A. *BPMAX ALGORITHM*

BPMax uses weighted base-pair counting for base-pair maximization. It considers both intermolecular and intramolecular

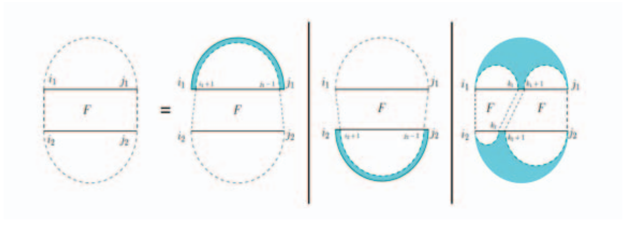


Fig. 2: The four cases defining table F

$$F_{i_1,j_1,i_2,j_2} = max \begin{cases} S^{(2)}_{i_2,j_2} & j_1 \le i_1 \\ S^{(1)}_{i_1,j_1} & j_2 \le i_2 \\ \text{iscore}(i_1,i_2) & i_1 = j_1 \text{ and } i_2 = j_2 \\ \max[\boxed{F_{i_1+1,j_1-1,i_2,j_2}} + \text{score}(i_1,j_1), & \\ \boxed{F_{i_1,j_1,i_2+1,j_2-1}} + \text{score}(i_2,j_2), & \\ H_{i_1,j_1,i_2,j_2}] & otherwise \end{cases} \tag{1}$$

$$H_{i_1,j_1,i_2,j_2} = \max \begin{cases} S^{(1)}(i_1,j_1) + S^{(2)}(i_2,j_2), \\ D_{i_1,j_1,i_2,j_2} \\ \boxed{\max_{k_2=i_2}^{j_2-1} S^{(2)}(i_2,k_2) + F_{i_1,j_1,k_2+1,j_2}} \\ \boxed{\max_{k_2=i_2}^{j_2-1} F_{i_1,j_1,i_2,k_2} + S^{(2)}(k_2+1,j_2)} \\ \boxed{\max_{k_1=i_1}^{j_1-1} S^{(1)}(i_1,k_1) + F_{k_1+1,j_1,i_2,j_2}} \\ \boxed{\max_{k_1=i_1}^{j_1-1} F_{i_1,k_1,i_2,j_2} + S^{(1)}(k_1+1,j1)} \end{cases} \tag{2}$$

$$D_{i_1,j_1,i_2,j_2} = \boxed{\max_{k_1=i_1}^{j_1-1} \max_{k_2=i_2}^{j_2-1} F_{i_1,k_1,i_2,k_2} + F_{k_1+1,j_1,k_2+1,j_2}} \tag{3}$$

base pairings. It does not allow pseudo-knots or crossings. Mathematically, it produces a four-dimensional triangular table - F-table (a triangular collection of triangles) based on two input sequences. Figure 2 shows the main cases for the F-table using Eddy-Rivas diagram. Equation 1, Equation 2, and Equation 3 show the complete recurrence equation of BPMax. Equation 3 highlighted in the blue color represents the double max-plus operation. It is the most compute-intensive portion of the algorithm. We use the same colors to highlight the dependence pattern in section IV.

B. Polyhedral Model

The Polyhedral model [6]–[10] 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. Static control parts like variables, iteration space (loop nests), and dependencies can be represented using polyhedra.

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

Listing 1: Prefix sum

Let us consider the prefix sum code highlighted in Listing 1 that 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

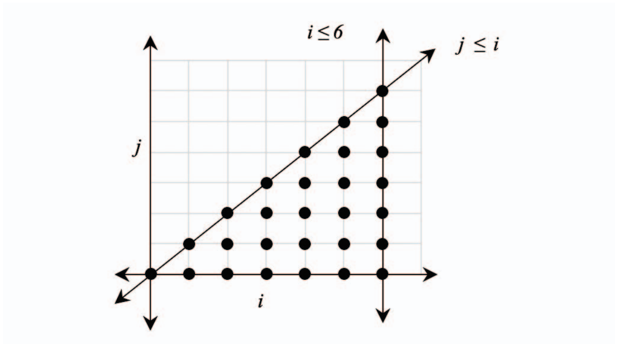


Fig. 3: Polyhedral iteration space for prefix-sum

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. However, they are affine, leading to the model's clean closure properties under 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 users to express one or more system of affine recurrence equations as a program, transform them using the polyhedral transformations that reduce the complexity of the program, use a better processor and memory allocation and then produce code for a language of interest.

C. ALPHAZ

ALPHA is a strongly typed functional language that works on a system of affine recurrence equations defined over polyhedral domains. Maurus [20] proposed this equational programming language as part of his doctoral dissertation. Subsequently, it has been extended to include subsystems and reductions [21]–[25]. Feautrier [8] showed that a polyhedral segment of code shown in Listing 1 can be translated to an ALPHA program. ALPHA is richer, mathematically cleaner, and more natural, especially, but not exclusively due to reductions. ALPHAZ is the tool that allows program transformations and user-directed compilation of ALPHA programs. It provides a general framework for analysis, transformation, and code generation in polyhedral equational model. ALPHAZ is similar to an earlier tool - MMALPHA , which targets field-programmable gate array based hardware design [26]. On the other hand, ALPHAZ targets code generation for multiprocessor shared-memory programs and focuses on programs with reduction operations.

Every ALPHA program has two pieces – system definition and compilation script. System definition of an ALPHA program allows users to express input and output of the program using polyhedral domains. It also allows a programmer to express the computation in terms of equations. The program containing the system definition is called alphabets. The second piece is the compilation script that contains commands to parse the input specification, transform the program based on the user input and finally generate the code. Algorithm 1 highlights the alphabets program for matrix multiplication. Algorithm 2 presents a compiling script for matrix multiplication.

Algorithm 1 Matrix Multiplication in Alphabets

```
1: affine MM $\{N, K, M \mid (M, N, K) > 0\}$
2: input
3:     float A $\{i, j \mid 0 \leq i < M \ \&\& \ 0 \leq j < K\}$ ;
4:     float B $\{i, j \mid 0 \leq i < K \ \&\& \ 0 \leq j < N\}$ ;
5: output
6:     float C $\{i, j \mid 0 \leq i < M \ \&\& \ 0 \leq j < N\}$;
7: local
8:     //local variables
9: output
10:    C[$i, j$] = reduce(+,   [$k$],   A[$i, k$] * B[$k, j$]);
```

1) Program Parsing: The first step is to read the alphabets program, which decomposes the equations into abstract syntax tree notation internally and sets up the stage for various program transformations.

2) Program Transformation: All transformations in ALPHAZ are semantic preserving. However, it is the responsibility of the user to ensure the transformations are valid. *Normalize* is the most basic transformation. It normalizes expression into normal form as per definition and makes the program easier to read and understand. *NormalizeReduction* transforms unnormalized form to normalize form. A normal form of a reduction transforms the reduce-expression to be a direct child of an equation. *setSpaceTimeMap* allows the user to specify schedule and processor allocation to specify the order in which one or more processors visit the iteration space. A system with multiple variables requires the dimension of all the spacetime maps to be equal. *NormalizeReduction* generates additional variables associated with the reductions. Specifying the schedule for such variable requires two different schedules to be provided – the first one specifies the order in which the iteration space will be visited. The other specifies the time at which

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",
9:                     "$(i, j, k \mapsto i, k, j)$",
10:                    "$(i, j \mapsto i, -1, j)$");
11: setParallel(prog, system, "", "0" );
12:
13: // $Step - 3 : Generate\ code$
14: generateWriteC(prog, system, outDir);
15: generateScheduleC(prog, system, outDir);
```

the initialization should occur before starting the reduction. It also has a dependency on the schedule. *setMemoryMap* - Memory map is a mapping between iteration points in the domain to the memory location. By default, ALPHAZ uses identity function as a memory map for each variable and bounding box of the polyhedral representation of the variable for storage. It allows multiple variables with different dimensions to share the same memory map based on affine function. *setMemorySpace* is similar to memory map. Except that multiple variables with the same dimension can share memory space. *setParallel* allows the user to specify one or more dimensions of the schedule to be executed in parallel by different threads. Users can also specify predicate as an ordering dimension to define the parallel loop dimension. Tiling transformation allows the user to chop the iteration space to improve data locality and adjust parallelization granularity.

3) Code Generation: This set of commands produce target code (e.g., c) based on the program transformation. ALPHAZ has various code generation options like – *generateWriteC* which is sequential in nature and useful to check the correctness of the program, schedule code generation – *generateScheduleC*. Sequential code generation hardly requires any

```
#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));
             }
         }
    }
}
```

Listing 2: Generated code - Matrix multiplication

program transformation. However, efficient schedule code generation depends on the choice of various transformations, mainly the target mapping-related transformations. The tiling transformation is not applied upfront rather invoked as part of the post processing phase of the schedule code generation.

IV. METHODS

We have staged our optimization process into three distinct phases. The following subsections describe these phases.

A. Phase-1:

This phase's primary goal is to express the BPMax equation in ALPHAZ , perform the first-level optimization on the most compute-intensive portion of the task and get a baseline estimation.

a) Multi-dimensional Affine Schedule for Double Max-plus Operation: We simplify the BPMax computation based on the following recurrence equation as the very first step.

$$F_{i_1,j_1,i_2,j_2} = \max_{k_1=i_1}^{j_1-1} \max_{k_2=i_2}^{j_2-1} F_{i_1,k_1,i_2,k_2} + F_{k_1+1,j_1,k_2+1,j_2} \quad (4)$$

Let us call this double max-plus computation R_0. Our goal is to find out the optimum multi-dimensional affine schedule [9], [10] for this equation. A schedule is only valid if it preserves the program semantic, which requires dependency analysis. We observe that each inner triangle of F-table is dependent on the triangles to the west and south. E.g., triangle C is dependent on all the Ax triangles towards the west and Bx triangles towards the south illustrated in Figure 4. There is no dependency from the triangle within. So, each inner triangle can be filled diagonally or bottom-up and then left to right. The first two dimensions of our multi-dimensional schedule can be either $(j_1 - i_1, i_1)$ or $(M - i_1, j_1)$ or $(-i_1, j_1)$ for F-table and R_0. There are many ways to formulate the next dimension for R_0. One such choice would be to use k_1. Figure 5 highlights the accumulation sequence based on this choice. It has the effect of performing multiple max-plus operations on a series of matrices (i_1, k_1) and $(k_1 + 1, j_1)$. It requires the third dimension of the F-table to be j_1, meaning we must finish computing all the max-plus operations for the current triangle before updating it. The inner three dimensions of the R_0 can be in any order since they do not have any dependencies. Thus, it can be $(j_2 - i_2, i_2, k_2)$ or

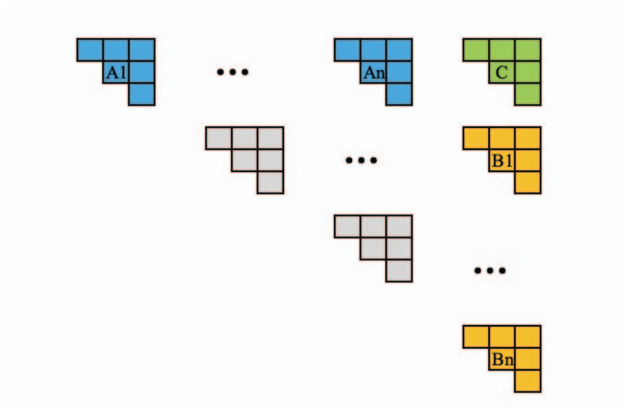


Fig. 4: Double max-plus dependency

(i_2, j_2, k_2) or $(M - i_2, j_2, k_2)$ or (i_2, j_2, k_2) or $(j_2 - i_2, k_2, i_2)$ or $(-i_2,\ k_2,\ j_2)$ etc. However, auto-vectorization is prohibited if k_2 is the innermost loop iteration. Other choices can be

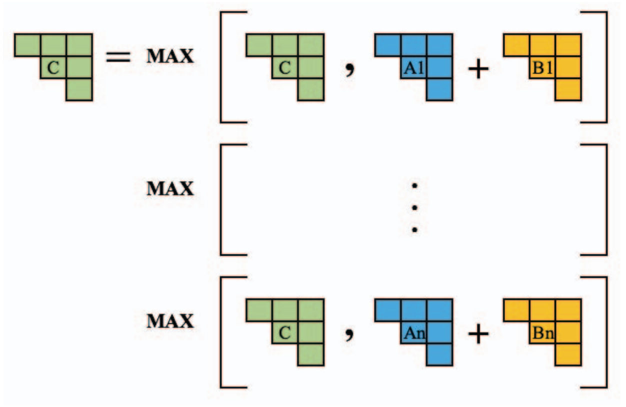


Fig. 5: Double max-plus accumulation sequence

viewed as loop permutations that allow auto-vectorization. For updating the final F-table entries, we can copy the data in any order. The original BPMax implementation uses $i_1, j_1, i_2, j_2 \mapsto j_1 - i_1, j_2 - i_2, i_1, i_2, k_1, k_2$ schedule for double max-plus computation. Table I shows various multi-dimensional affine schedules similar to Varadrajan's. We pick similar schedules to establish the baseline since her kernel was based on multiply-add and ours is max-plus. Also, we use single-precision storage to reduce the memory footprint of BPMax.

b) Machine Peak Analysis and Micro-benchmark: Next, we develop the roofline model for the target machine. Our schedule tries to exploit auto-vectorization that loads one scalar and vector of 8 elements from L1 to compute 8 max-plus operations, and the data access pattern is $Y = \max(a+X, Y)$. We create a micro-benchmark to estimate the attainable L1 bandwidth for such access pattern. It allocates two large one-dimensional arrays for each thread, initializes them with random numbers, and then invokes the kernel (Algorithm 3)

Algorithm 3 Max-plus streaming benchmark

```
1: for $iteration \leftarrow 0, MAX\_ITERATION$ do
2:     for $index \leftarrow 0, CHUNK\_SIZE$ do
3:         $Y[index] = \max(alpha + X[index], Y[index])$
4:     end for
5: end for
```

that computes the max-plus operation on the array. The micro-benchmark data is presented in the result section.

TABLE I: DOUBLE MAX-PLUS SCHEDULE

	Variable	*Schedule*
a	F	$(i_1, j_1, i_2, j_2 \mapsto j_1 - i_1, i_1, j_1, i_2, j_2, j_2)$
	R_0	$(i_1, j_1, i_2, j_2, k_1, k_2 \mapsto j_1 - i_1, i_1, k_1, i_2, k_2, j_2)$, $(i_1, j_1, i_2, j_2 \mapsto j_1 - i_1, i_1, i_1 - 1, i_2, i_2 - 1, j_2)$
b	F	$(i_1, j_1, i_2, j_2 \mapsto -i_1, j_1, j_1, -i_2, j_2, j_2)$
	R_0	$(i_1, j_1, i_2, j_2, k_1, k_2 \mapsto -i_1, j_1, k_1, -i_2, k_2, j_2)$, $(i_1, j_1, i_2, j_2 \mapsto -i_1, j_1, i_1 - 1, -i_2, i_2 - 1, j_2)$
c	F	$(i_1, j_1, i_2, j_2 \mapsto j_1 - i_1, i_1, j_1, i_2, j_2, j_2)$
	R_0	$(i_1, j_1, i_2, j_2, k_1, k_2 \mapsto j_1 - i_1, i_1, k_1, i_2, k_2, j_2)$, $(i_1, j_1, i_2, j_2 \mapsto j_1 - i_1, i_1, i_1 - 1, i_2, i_2 - 1, j_2)$

[a]Fine-grain schedule(diagonal), parallel dimension 3
[b]Fine-grain schedule(bottom-up), parallel dimension 3
[c]Coarse-grain schedule, parallel dimension 1

c) Insights from Phase-I: This phase highlights the possibility of further improvements of R_0 beyond loop permutation. Double max-plus performance attains just above 20% of our theoretical max-plus machine peak. We notice a significant collapse in performance when the input sequences are longer.

B. Phase-II

We have two objectives in this phase. First, find a complete schedule for BPMax that enables automatic vectorization for all the variables and estimate the other reduction term's overhead. Second, explore optimization opportunities for the double max-plus operation.

a) Multi-Dimensional Affine Schedule for BPMax: Figure 6 shows the complete BPMax dependencies for a point. The point colored in red depends on all the other colored points. BPMax has a total of five reductions. There are four additional reductions R_1 (green), R_2 (orange), R_3 (purple), R_4 (yellow) beside R_0 (blue). So, there are new dependencies in addition to the similar dependencies highlighted in the double max-plus computation. R_1 and R_2 have internal dependencies, but the other two have external dependencies. One common theme across various schedules is that $S^{(1)}$ and $S^{(2)}$ can be scheduled before scheduling any other variables. Also, order of filling up the inner triangle does not change with the introduction of the other terms. Thus, the first two dimensions of our multi-dimensional schedule can be either $(j_1 - i_1, i_1)$ or $(M - i_1, j_1)$ or $(-i_1, j_1)$ for F-table, R_0, R_1, R_2, R_3, R_4. R_3

and R_4 is over k_1 like R_0. So, the third dimension can be the same for all of them. Next, the inner three dimensions of the R_0 can also be the same as discussed earlier. R_3 and R_4 can use (i_2, j_2). However, we introduce additional terms to make the schedule dimension equal for all the variables. F-table, R_1, and R_2 must wait until k_1 reaches j_1 like double max-plus. Thus, their third schedule dimension becomes j_1. Final

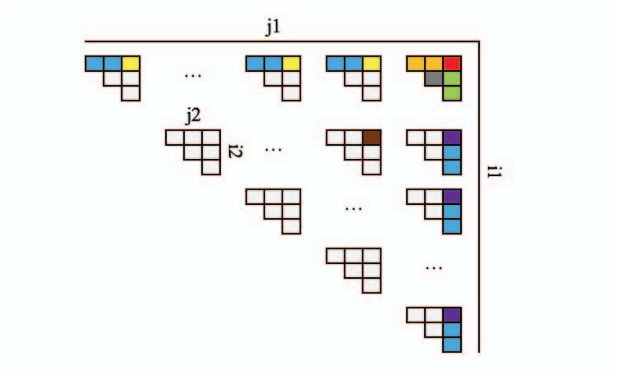


Fig. 6: BPMax dependency overview

F-table entries require intra-triangular dependencies to be evaluated which are similar to inter-triangular dependencies. So, it can be filled up diagonally or bottom-up and left to right. Thus, the inner three dimensions of the schedule for R_1 and

TABLE II: 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	$(i_1, j_1, i_2, j_2, k_2 \mapsto 1, -i_1, j_1, j_1, -i_2, 0, k_2, j_2)$, $(i_1, j_1, i_2, j_2 \mapsto 1, -i_1, j_1, j_1, -i_2, 0, i_2 - 1, j_2)$
R_0	$(i_1, j_1, i_2, j_2, k_1, k_2 \mapsto 1, -i_1, j_1, k_1, -1, -i_2, k_2, j_2)$, $(i_1, j_1, i_2, j_2 \mapsto 1, -i_1, j_1, i_1 - 1, -1, -i_2, i_2 - 1, j_2)$
R_3, R_4	$(i_1, j_1, i_2, j_2, k_1 \mapsto 1, -i_1, j_1, k_1, -1, -i_2, i2, j_2)$, $(i_1, j_1, i_2, j_2 \mapsto 1, -i_1, j_1, i_1 - 1, -1, -i_2, i_2, j_2)$

[a]Parallel dimension 5

TABLE III: 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	$(i_1, j_1, i_2, j_2, k_2 \mapsto 1, j_1 - i_1, i_1, j_1, -i_2, k_2, j_2)$, $(i_1, j_1, i_2, j_2 \mapsto 1, j_1 - i_1, i_1, j1, -i_2, i_2 - 1, j_2)$
R_0	$(i_1, j_1, i_2, j_2, k_1, k_2 \mapsto 1, j_1 - i_1, i_1, k_1, i_2, k_2, j_2)$, $(i_1, j_1, i_2, j_2 \mapsto 1, j_1 - i_1, i_1, i_1 - 1, i_2, i_2 - 1, j_2)$
R_3, R_4	$(i_1, j_1, i_2, j_2, k_1 \mapsto 1, j_1 - i_1, i_1, k_1, i_2, i_2, j_2)$, $(i_1, j_1, i_2, j_2 \mapsto 1, j_1 - i_1, i_1, i_1 - 1, i_2, i_2, j_2)$

[a]Parallel Dimension 2

R_2 can be $(j_2 - i_2, i_2, k_2)$ or $(N - i_2, j_2, k_2)$ or $(-i_2, j_2, k_2)$ or $(j_2 - i_2, k_2, j_2)$ or $(N - i_2, k_2, j_2)$ or $(-i_2, k_2, j_2)$ etc. We carefully chose the schedule for R_1 and R_2 since the innermost k_2 prevents automatic vectorization. We ensure that F-table gets updated when k_2 reaches j_2. Table II and III shows various multi-dimensional affine schedules which provide better results than the base schedule.

b) Parallelization Approach: We take two different types of parallelization approach in this phase - coarse and fine-grain. For coarse-grain parallelization, threads work on distinct inner triangles simultaneously. It is valid for R_0, R_1, R_2, R_3, and R_4. On the other hand, threads work on individual rows of an inner triangle simultaneously for fine-grain parallelization. It is only valid for R_0, R_3, and R_4.

c) Memory Optimization: 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 memory hierarchies. Reduction variables also take up memory space by default, which is wasteful. In this phase, coarse-grain parallelization still requires P (number of threads) instances of a $2 - D$ array for each reduction variables to be active in memory except R_0. R_0 shares memory with F. Fine-

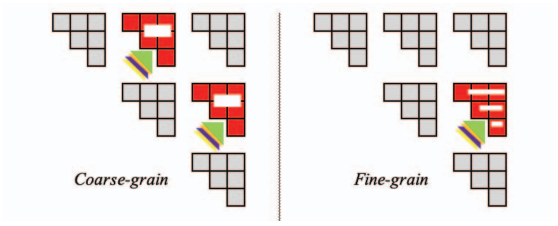


Fig. 7: BPMax phase-II memory map

grain requires only a $2 - D$ array for each of these variables illustrated in Figure 7.

d) 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

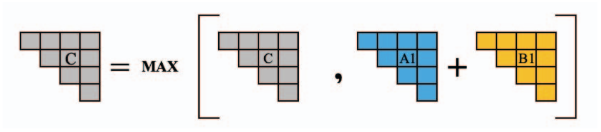


Fig. 8: A matrix instance of max-plus operation

of one matrix instance of max-plus operation. It is matrix multiplication like computation, except only a fraction of work is being done here, and the access pattern is imbalanced. We tile three inner dimensions with k_2 loop still in the middle and j_2 loop inside. So, this chops (i_2, k_2, j_2) iteration space, and we parallelize the outer i_2 dimension.

e) Insights from Phase-II: Loop permutation and automatic vectorization provide a significant speedup for the entire BPMax program. However, the program suffers from imbalanced parallelization. The tiled version of the R_0 attains 33% of our theoretical max-plus machine peak.

C. Phase-III

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

a) Parallelization Approach: Earlier, we have observed that the program quickly becomes DRAM-bound for the coarse-grain schedule since each thread computes an inner triangle. But, it allows us to parallelize R_1 and R_2. On the other hand, R_0, R_3, and R_4 can be computed using the fine-grain schedule, which reduces data movement between DRAM and LLCs. However, R_1 and R_2 are not easy to parallelize. These are optimum string parenthesization (OSP)-like computations that require further transformation like middle serialization. If we use the fine-grain parallelism without such transformation, only one thread stays active, 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 IV. It improves CPU utilization and limits the data movement between DRAM and LLCs. However, there are some limitations discussed in the result section.

TABLE IV: 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	$(i_1, j_1, i_2, j_2, k_2 \mapsto 1, j_1 - i_1, M, 0, i_1, -i_2, k_2, j_2)$, $(i_1, j_1, i_2, j_2 \mapsto 1, j_1 - i_1, M, 0, i_1, -i_2, i_2 - 1, j_2)$
R_0	$(i_1, j_1, i_2, j_2, k_1, k_2 \mapsto 1, j_1 - i_1, i_1, k_1, i_2, k_2, j_2, 0)$, $(i_1, j_1, i_2, j_2 \mapsto 0, j_1 - i_1, i_1, 0, i_2, 0, j_2, 0)$
R_3, R_4	$(i_1, j_1, i_2, j_2, k_1 \mapsto 1, j_1 - i_1, i_1, k_1, i_2, i_2, j_2, 0)$, $(i_1, j_1, i_2, j_2 \mapsto 0, j_1 - i_1, i_1, 0, i_2, 0, j_2, 0)$

[a]Parallel Dimension 4

b) Tiling Integration and Subsystem Scheduling: ALPHAZ produces inferior code when the tiling is applied to a subset of reduction operations. 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, R_4 and the primary system produces R_1 and R_2 along with final F-table output and consolidates the results from the subsystem. 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. Two lines of source code changes are made to achieve this. Table V summarizes the complete schedule for the two systems.

TABLE V: BPMAX HYBRID SCHEDULE WITH TILING

	Variable	***Schedule***
a	c	$(i_1, j_1 \mapsto M, i_1, j_1, 0)$
	R_0	$(i_1, j_1, k_1, k_2 \mapsto k_1, i_1, k_2, j_1)$, $(i_1, j_1 \mapsto -1, i_1, 0, j_1)$
	R_3, R_4	$(i_1, j_1, k_1 \mapsto k_1, i_1, i_1, j_1)$, $(i_1, j_1 \mapsto -1, i_1, 0, j_1)$
b	$S^{(1)}, S^{(2)}$	$(i_1, j_1 \mapsto 0, 0, j_1 - i_1, i_1, 0, 0, 0)$
	d	$(i_1, j_1 \mapsto 1, j_1 - i_1, i_1, j_1 - 4, 0, 0, 0)$
	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	$(i_1, j_1, i_2, j_2, k_2 \mapsto 1, j_1 - i_1, M, i_1, -i_2, k_2, j_2)$, $(i_1, j_1, i_2, j_2 \mapsto 1, j_1 - i_1, M, i_1, -i_2, i_2 - 1, j_2)$

a - Subsystem schedule(parallel dimension 1)
b - Root system schedule(parallel dimension 3)
c - Subsystem output
d - Subsystem call

c) Memory Optimization: We further optimize memory utilization in this phase. R_0, R_3 and R_4 are always computed before final F-table update. So, they share the memory with F-table. Also, only one row of an inner triangle is required for R_1 and R_2 to keep up with the F-table update. This has been highlighted in Figure 9. We also optimize redundant data copies during subsystem call using *setMemorySpaceForUseEquationOptimization* transformation.

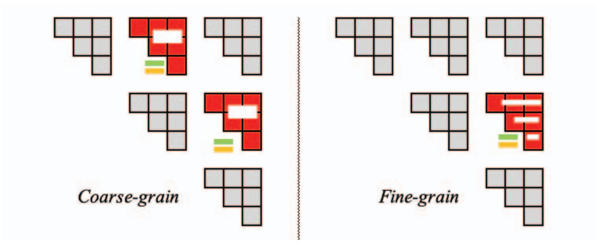


Fig. 9: BPMax phase-III memory map

d) Performance Tuning: To find an optimum tile shape, we start with cubic tiles and then adjust one or more dimensions to find a better tile shape that works moderately well across various inputs. However, we notice 10% performance differences between the best and generic tile sizes. The OMP dynamic-schedule works better than the static and guided-schedule due to an imbalanced workload. Next, we perform some manual memory optimization. The schedule initializes memory for each reduction body. Still, when one variable

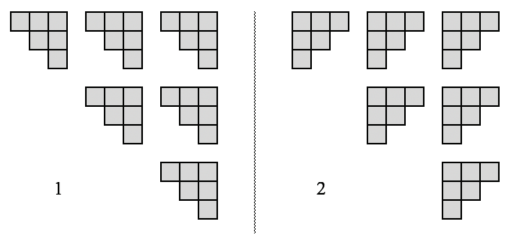


Fig. 10: Memory mapping schemes

shares memory space with multiple variables, memory initialization becomes redundant, and the current code generator does not optimize it. We comment out these macros, which attempt duplicate initializations to eliminate redundancies. We have tried two different memory transformations for the inner triangle highlighted in Figure 10 - 1 : $(i_2, j_2 \mapsto i_2, j_2)$ and 2 : $(i_2, j_2 \mapsto i_2, j_2 - i_2)$. Option-1 always performs better.

V. Results

We use Xeon E5-1650v4 to present the results of our optimization approach. Xeon E5-1650v4 has six cores where each core has 32 KB 8-way set associative L1 and 256 KB 8 way-set associative cache. They share a 15 MB 20-way set-associative cache.

A. Machine Peak Overview

Intel's micro-architecture specification indicates that the sustained L1 and L2 data cache bandwidth are 93 bytes and 25 bytes/cycle, respectively, whereas L3 bandwidth and DRAM bandwidth are 14 bytes/cycle and 76.8 GB/second,

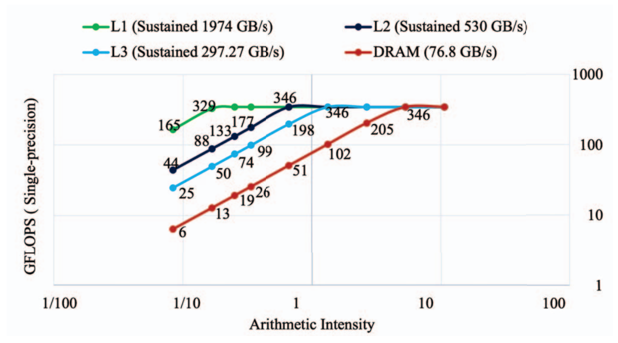


Fig. 11: Xeon E5 1650v4 roofline based on micro-architecture

respectively. Based on this data, we have come up with the roofline model shown in Figure 11. The theoretical max-plus machine peak is about 346 GFLOPS for single-precision. We will implicitly use GFLOPS to indicate the single-precision performance for the rest of the section. BPMax performs 2-arithmetic operations for 3-single-precision memory operations. So, its arithmetic intensity is $\frac{2}{(3\times4)}$ or $\frac{1}{6}$ (second data points on each series shown in Figure 11). Based on the roofline model, we expect to achieve around 329 GFLOPS based on L1 bandwidth. Our micro-benchmark data (Figure 12) shows that we achieve up to 120 GFLOPS with 6 threads and 240 GFLOPS with 12 threads.

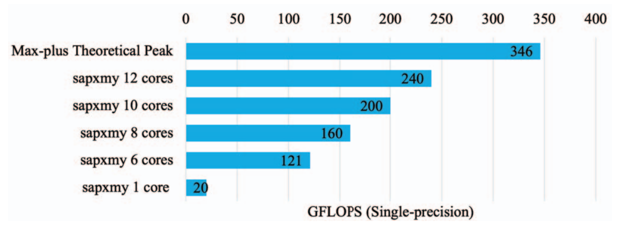


Fig. 12: Micro-benchmark for $Y = \max(a + X, Y)$

B. Performance Analysis of Double Max-plus Computation

In this section, we go over the results of our optimization approach for double max-plus computation. Figure 13 and Figure 14 show the performance and speedup comparison of double max-plus between different schedules using 6 threads. We see that the coarse-grain parallelization performs very poorly since it generates a lot of DRAM traffic and makes 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. Our tiling approach improves locality and maintains automatic vectorization. It enables us to get close to the 97% of our micro-benchmark target. We attain 117 GFLOPS with the tiling transformation. Tile dimensions of $(32 \times 4 \times N)$ and $(64 \times 16 \times N)$ are used for presenting the performance and speedup comparison. $(32 \times 4 \times N)$ is restricted for sequence length up to 2048.

We achieve around 178× improvement over the base implementation taken from the BPMax program. It is a sequential improvement of $40 - 200\%$ with 6 threads over Varadrajan's fine-grain schedule. However, her results show that hyper-threading helped improve performance by over 10% in some cases. We see minimal $(3 - 5\%)$ improvement with hyper-threading over six threads shown in Figure 18. We have done experiments with different tile sizes $(i_2 \times k_2 \times j_2)$ and found that the cubic tiles perform poorly. We observe the best result when j_2 is not tiled due to the streaming effect.

C. BPMax Performance Improvement

Figure 15 and 16 show the performance improvements and speedup of different versions of the BPMax program using 6 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 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 hybrid parallelization approach highlighted in green 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× speedup for longer sequence lengths with 6 threads. The improvement for the tiled version mainly comes from the optimization of R_0, R_3, R_4. The tiled version of the program reaches around 76 GFLOPS for moderate-size sequences. It is almost 60% lower than the best double max-plus version of the same sequence. Our analysis shows that R_3 and R_4 are almost free since those get computed along with the R_0. The other 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

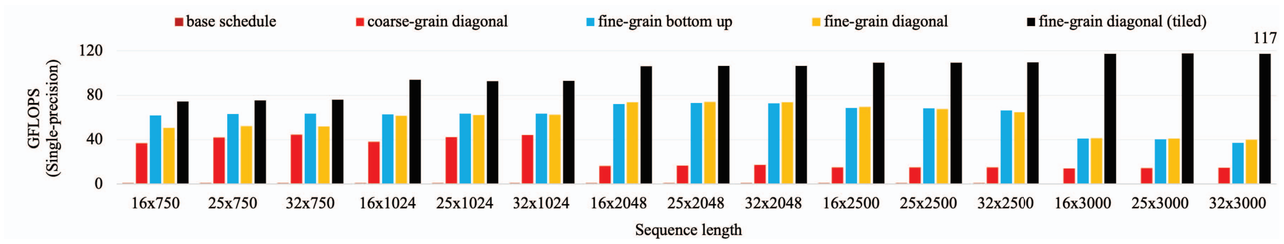


Fig. 13: Double max-plus performance comparison

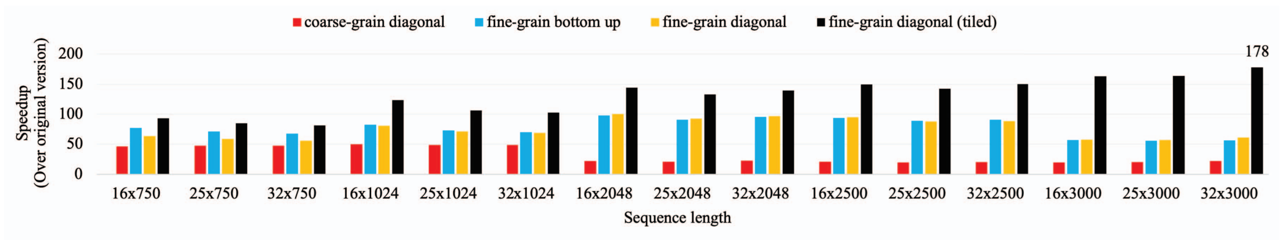


Fig. 14: Double max-plus speedup comparison

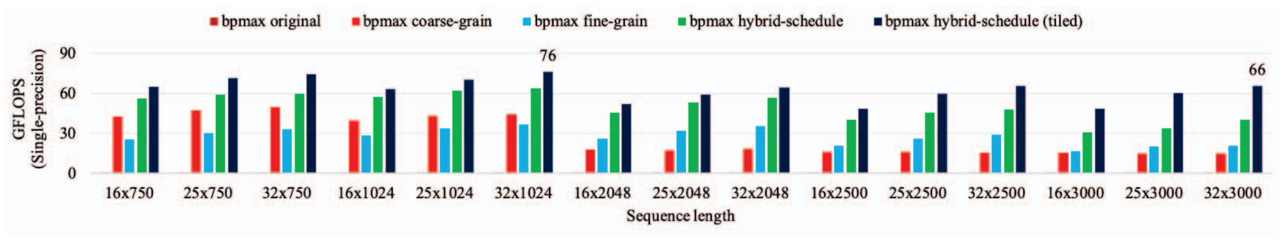


Fig. 15: BPMax performance comparison

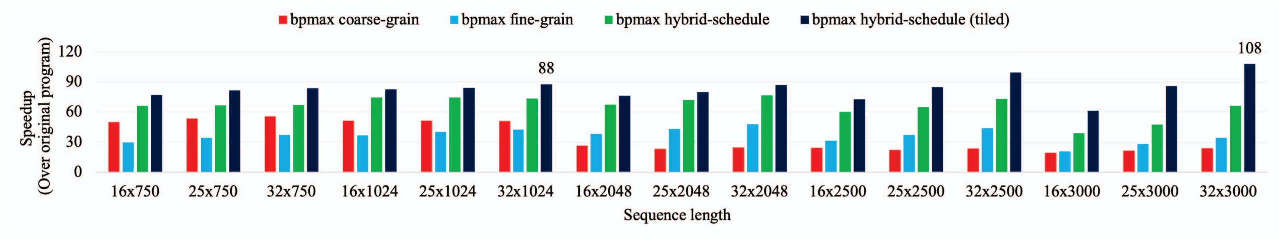


Fig. 16: BPMax speedup comparison

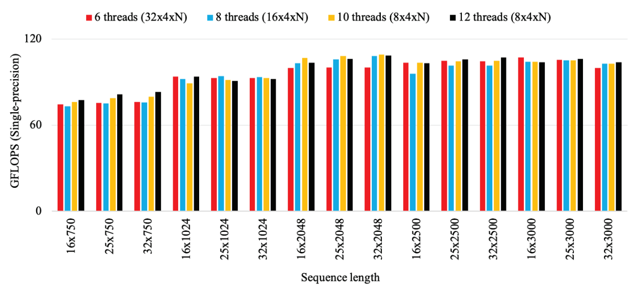


Fig. 17: Effect of hyper-threading on tiled double max-plus performance

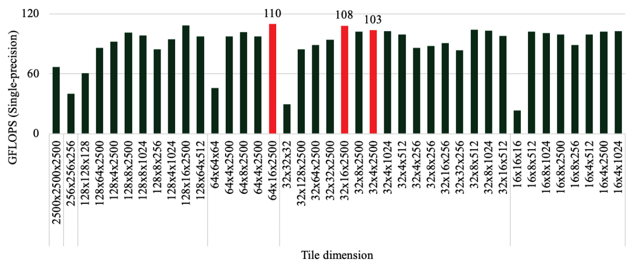


Fig. 18: Effect of tiling parameters ($i_2 \times k_2 \times j_2$) on double max-plus performance (sequence length - 16 x 2500)

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 6 threads for E5-1650v4).

Besides Xeon E5-1650v4, we have verified the scalability of our optimized program on Intel Xeon E-2278G, which has eight cores and runs almost at the same speed as E5-1650v4. The optimized BPMax performs the same or better on E-2278G compare to E5-1650v4, reaching close to one-fourth of the theoretical single-precision machine peak. Figure 1 shows an overview of the performance and speedup comparison on Xeon E-2278G and Xeon E5-1650v4.

D. Code Generation Metric

Table VI shows the line of code (*LOC*) generated by ALPHAZ for different BPMax program versions. We see an increase in *LOC* when the program is optimized. Also, the table highlights the complexities (based on *LOC*) between the double max-plus computation and the BPMax program.

TABLE VI: 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

VI. CONCLUSION

This work demonstrates the optimization process of a complete RRI program using polyhedral transformations. Our result shows that the tiling improves the performance of the most dominant part of the computation. However, the inner reductions are still inefficient, which limits the overall performance improvement. Also, the double max-plus operation remains bandwidth-bound even after tiling transformation. This indicates that an additional level of tiling at the register level is required to make the program compute-bound and improve performance. We also plan to apply tiling on R_1 and R_2 and distribute the computation over a cluster using MPI. All these transformations remain a challenge for ALPHAZ , and we hope to overcome them in the future.

REFERENCES

[1] R. Nussinov, G. Pieczenik, J. R. Griggs, and D. J. Kleitman, “Algorithms for loop matchings,” *SIAM Journal on Applied Mathematics*, vol. 35, no. 1, pp. 68–82, jul 1978.

[2] H. Chitsaz, R. Salari, S. C. Sahinalp, and R. Backofen, “A partition function algorithm for interacting nucleic acid strands,” *Bioinformatics*, vol. 25, no. 12, pp. i365–i373, may 2009.

[3] A. Ebrahimpour-Boroojeny, S. Rajopadhye, and H. Chitsaz, “Bppart and bpmax: Rna-rna interaction partition function and structure prediction for the base pair counting model,” apr 2019.

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

[5] F. W. D. Huang, J. Qin, C. M. Reidys, and P. F. Stadler, “Partition function and base pairing probabilities for RNA–RNA interaction prediction,” *Bioinformatics*, vol. 25, no. 20, pp. 2646–2654, aug 2009.

[6] 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*. Springer Verlag, LNCS 241, December 1986, pp. 488–503.

[7] P. Quinton and V. Van Dongen, “The mapping of linear recurrence equations on regular arrays,” *Journal of VLSI Signal Processing*, vol. 1, no. 2, pp. 95–113, 1989.

[8] P. Feautrier, “Dataflow analysis of array and scalar references,” *International Journal of Parallel Programming*, vol. 20, no. 1, pp. 23–53, Feb 1991.

[9] ——, “Some efficient solutions to the affine scheduling problem. Part I. one-dimensional time,” *International Journal of Parallel Programming*, vol. 21, no. 5, pp. 313–347, 1992.

[10] ——, “Some efficient solutions to the affine scheduling problem. Part II. multidimensional time,” *International Journal of Parallel Programming*, vol. 21, no. 6, pp. 389–420, 1992.

[11] U. Bondhugula, J. Ramanujam, and P. Sadayappan, “Pluto: A practical and fully automatic polyhedral program optimization system,” 2015.

[12] U. Bondhugula, A. 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.

[13] S. Varadarajan, “Polyhedral optimizations of RNA-RNA interaction computations,” Master’s thesis, Colorado State University, 2016.

[14] C. Chen, J. Chame, and M. Hall, “Chill: A framework for composing high-level loop transformations,” Tech. Rep., 2008.

[15] T. Yuki, G. Gupta, D. 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.

[16] C. Mondal and S. Rajopadhye, “Bpmax acceleration on cpu,” https://github.com/chiranjeb/BPMaxCPU, 2020.

[17] M. Palkowski and W. Bielecki, “Tiling nussinov’s RNA folding loop nest with a space-time approach,” *BMC Bioinformatics*, vol. 20, no. 1, apr 2019.

[18] S. 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.

[19] B. Gildemaster, P. Ghalsasi, and S. 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.

[20] C. Mauras, “Alpha : un langage equationnel pour la conception et la programmation d’architectures paralleles synchrones,” Ph.D. dissertation, Rennes 1, 1989.

[21] H. Le Verge, “Un environnement de transformations de programmmes pour la synthèse d’architectures régulières,” Ph.D. dissertation, L’Université de Rennes I, Oct 1992.

[22] ——, “Reduction operators in alpha,” in *Parallel Algorithms and Architectures, Europe*, ser. LNCS, D. Etiemble and J.-C. Syre, Eds. Springer Verlag, June 1992, pp. 397–411, see also, Le Verge Thesis (in French).

[23] d. F. and S. Robert, “Hierarchical static analysis of structured systems of affine recurrence equations,” in *International Conference on Application Specific Systems Architectures and Processors (ASAP 96)*, J. Fortes, C. Mongenet, K. Parhi, and V. Taylor, Eds. IEEE, August 1996, pp. 381–390.

[24] F. Dupont de Dincehcin, “Systèmes structurés d’équations récurrentes : mise en œuvre dans le langage Alpha et applications,” Ph.D. dissertation, Université de Rennes, janvier 1997.

[25] F. Dupont de Dinechin, P. Quinton, and T. Risset, “Structuration of the alpha language,” in *Massively Parallel Programming Models*, W. Giloi, S. Jahnichen, and B. Shriver, Eds. IEEE Conmputer Society Press, 1995, pp. 18–24.

[26] A.-C. Guillou, F. Quilleré, P. Quinton, S. Rajopadhye, and T. Risset, “Hardware design methodology with the Alpha language,” in *Forum on Design Languages*, Sept 2001.