Accelerating RNA-RNA Interaction for Machine Peak

Chiranjeb Mondal, *Colorado State University* and Sanjay Rajopadhye, *Colorado State University*

Abstract—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. Algorithms to model RRI typically use dynamic programming, and have complexity $\Theta(N^3M^3)$ in time and $\Theta(N^2M^2)$ in space. This makes it both essential and challenging to parallelize them. Previous efforts to do so have been hand-optimized, which is prone to human error, costly to develop, and maintain. This paper presents a multi-core CPU parallelization of BPMax, one of the simpler RRI algorithms, generated by a user-guided polyhedral code generation tool, **AlphaZ**. The user starts with a mathematical specification of the dynamic programming algorithm, and provides the choice of polyhedral program transformations such as schedules, memory-maps, and multi-level tiling. **AlphaZ** automatically generates highly optimized code. At the lowest level, we implemented a small hand-optimized register-tiled "matrix max-plus" kernel, and integrated it with our tool-generated optimized code. Our final optimized program version is about $400\times$ faster than the base program, translating to around 312 GFLOPS, more than half of our platform's roofline machine peak (RMP) performance. On a single core, we attain 80% of RMP. The main kernel in the algorithm, whose complexity is $\Theta(N^3M^3)$, attains 58 GFLOPS on a single-core and 344 GFLOPS on multi-core (90% and 58% of RMP, respectively).

Index Terms—RRI, BPMax, Polyhedral Compilation

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 nucleotides—Adenine (A), Uracil (U), Guanine (G), and Cytosine (C). 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 combined secondary structure, which provides valuable information about biological function. Mortimer et al. highlight the emerging relationships between such RNA structure and the regulation of gene expression [1].

RNA-RNA interactions have 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. [2] developed **piRNA**, one of the most comprehensive thermodynamic RRI models. Running this compute and the memory-intensive program is exceptionally challenging. Boroojeny et al. [3] retreated from the comprehensive model and developed BPPart [3], which reduces the complexity by a constant factor of 10 and BPMax which maintains only one table. Still, these implementations suffer from poor performance as the input sequence size grows.

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 nor easily maintainable. The challenge grows as the complexity of the program increases. Ideally, the optimized programs should be generated from a simple correct input specification, together with a set of performance tuning hints or directives.

Fortunately, RRI algorithms fit the requirements of the *polyhedral model* [4], [5], [6], [7], [8], [9], [10], [11], [12], a mathematical formalism that allows for just such program transformations. The polyhedral compilation has been the subject of intense research for over 35 years. Yet, even state-of-the-art polyhedral tools like PLUTO [13], [14] fail to yield satisfactory performance. Specifically, Varadarajan [15] evaluated its performance on a simple program whose structure is close to the core computation of RRI algorithms. The performance was significantly lower than the hand-written baseline implementation. Many of the optimization strategies need insights from an expert. This gap can be bridged by tools like Chill [16], Hailde [17] and MMAlpha [18] that allow semi-automatic transformation. At CSU, we are developing and working with **AlphaZ** [19], a similar tool for generating optimized code that raises the level of abstraction. Specifically, our paper makes the following contributions:

- We show efficient tiling and scheduling of a RRI program - BPMax on a single CPU machine using a polyhedral code generation tool, **AlphaZ**.
- We implement a highly optimized max-plus register kernel, which achieves a performance closer to machine peak and integrate it with our auto-generated code.
- We generate highly optimized code for BPMax that achieves about $400\times$ speedup over the original program. It is 80% and 53% of our max-plus roofline [20] peak on single-core and multi-core, respectively.
- The most compute-intensive part of BPMax achieves about $400\times$ speedup over the original implementation, and about $4\times$ improvement over a previous optimization approach [21]. It is 90% and 58% of our platform's max-plus roofline peak on single-core and multi-core, respectively.

2 BACKGROUND

THIS section highlights the related work, namely the BPMax algorithm, and 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 [22] 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 [23] proposed 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. It has also been formulated as a Bayesian inference problem [24].

There were prior works on the optimization and parallelization of these algorithms on the CPU platform. Li et al. [25] worked on the CPU and GPU versions of the Nussinov [22] RNA folding. Swenson et al. [26] worked on a parallel secondary structure prediction program for multicore desktop. Wonnacott et al. [27] proposed automatic tilling of "mostly-tileable" loop nests and applied their technique on Nussinov's algorithm. However, their implementation is significantly slower than the hand-written C codes. Palkowski et al. [28] used the polyhedral model to optimize Nussinov's [22] algorithm and able to generate optimized program. Rizk et al. [29] presented a GPU implementation of Zuker's algorithm [23]. However, most of the optimization efforts were related to single RNA strand folding.

Varadarajan [15], [30] 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 [31] 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 The BPMax Algorithm

BPMax [3] uses weighted base-pair counting. 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.

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

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

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

Equations 1 and 2 completely specify the BPMax algorithm. There are five reductions each of which is highlighted in a different color. We also use the same colors to highlight the dependence pattern in Section 3. The blue reduction (R^0) represents the double max-plus operation. It is the most compute-intensive portion of the algorithm. The other reductions are R^1 (green), R^2 (orange), R^3 (purple), and R^4 (yellow).

2.3 Polyhedral Model

The Polyhedral model [4], [5], [6], [7], [8], [9], [10], [11], [12] is a mathematical framework for automatic optimization and parallelization of affine programs. The model provides an abstraction to represent static control parts like variables, iteration space (loop nests), and dependencies using integer points in polyhedra.

3 BPMAX OPTIMIZATION

IN this section, we first go over the BPMax dependencies, describe the limitations of current BPMax implementation, discuss the different optimization strategies, and some of the key elements of our implementation approach.

3.1 Dependency Analysis

BPMax algorithm computes a four-dimensional sparse table $F(i_1, j_1, i_2, j_2)$ shown in Figure 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 also called 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 1 shows the complete BPMax dependencies for an 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 a complexity of $\Theta(M^2N^3)$ and 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.

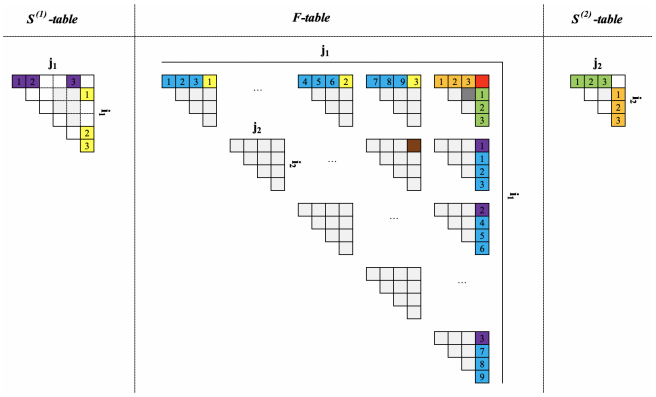


Fig. 1: BPMax dependency overview

3.2 Original Implementation

In the original BPMax implementation, all the diagonal elements are computed simultaneously, exposing the maximum level of parallelism. It accesses all the inner triangles (highlighted in light red) towards the left of the diagonal points (red points Figure 2). So, the total amount of the active data footprint 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,

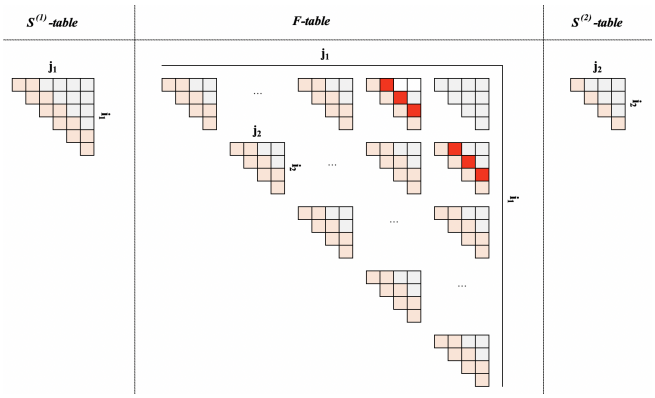
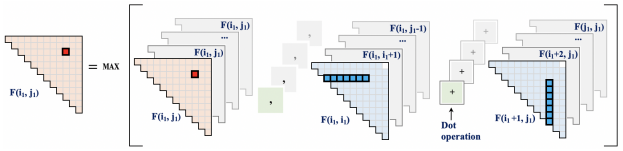


Fig. 2: BPMax Original Schedule

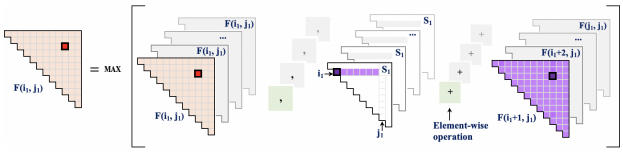
each of these reductions in the original implementation has loop carried dependencies, which prevents vectorization.

3.3 Previous Optimization Approach

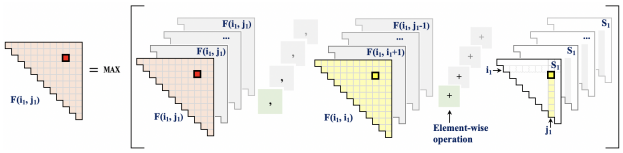
Our previous paper [21] introduced two levels of tiling to the BPMax - the first-level tilling to calculate each inner triangle at a time to improve the data locality. We also decomposed the double max-plus similar to [15] into a sequence of multiple matrix max-plus instances and applied loop tiling as the second-level tiling on each of the instances highlighted in Figure 3a. We were able to tile each instance since there is no dependency between the input and output matrices. However, this approach had a few issues when we attempted to take advantage of auto-vectorization. It performed better only when the innermost dimension (vector dimension) was longer, which effectively prohibited the smaller tile dimension reducing better data locality. We noticed 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



(a) Decomposition of R^0



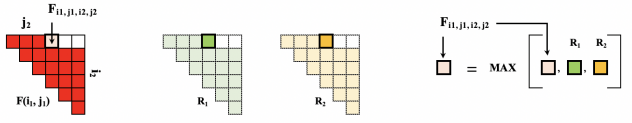
(b) Decomposition of R^3



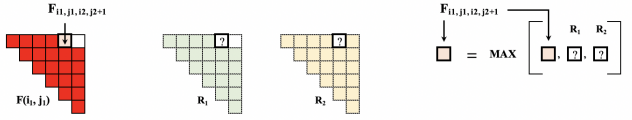
(c) Decomposition of R^4

Fig. 3: R^0, R^3, and R^4 Accumulation

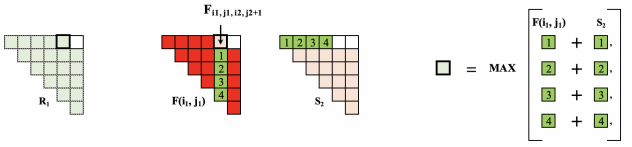
west of $F(i_1, j_1)$ and $S^{(1)}$ highlighted in Figure 1. To take advantage of the re-use, we decomposed the R^3 similar to R^0 as a set of max-plus operations between F-table entries ($\{F(k_1 + 1, j_1) \mid i_1 <= k_1 < j_1\}$) and $S^{(1)}$ highlighted in Figure 3b. However, R^3 is an element-wise operation



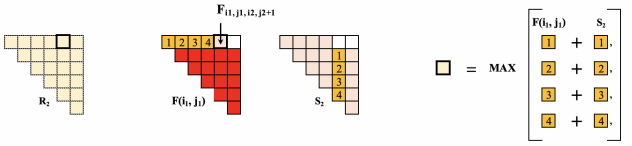
(a) Step 1: Let us assume that we are about to update the next element of $F(i_1, j_1)$: F_{i_1,j_1,i_2,j_2}. Results of R^0, R^3, and R^4 corresponding to all the elements of $F(i_1, j_1)$ are already accumulated in $F(i_1, j_1)$. Let us also assume that R^1 and R^2 are also computed for F_{i_1,j_1,i_2,j_2} element. Thus, it gets updated with the maximum of the F_{i_1,j_1,i_2,j_2}, R^1, and R^2.



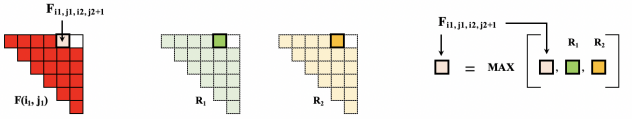
(b) Step 2: Next, we attempt to update F_{i_1,j_1,i_2,j_2+1} highlighted in thick bordered light red box. R^1, and R^2 are not computed yet for F_{i_1,j_1,i_2,j_2+1}. Thus we need to compute these two reduction results before updating this point.



(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.

Fig. 4: Illustration of F-table entry update with R^1 and R^2

instead of the matrix product-like operations done in R^0. Similarly, R^4 can also be expressed as a set of element-wise max-plus operations between $S^{(1)}$ and F-table entries highlighted in Figure 3c. After accumulating all the results from R^0, R^3, and R^4 into $F(i_1, j_1)$, we updated it using R^1 and R^2. R^1 and R^2 have dependencies with the inner triangle that is being computed and $S^{(2)}$. Thus, these three updates must happen in a specific order demonstrated in Figure 4. Unlike $R^{0,3-4}$, we could not to tile these two reductions for each $F(i_1, j_1)$. These are optimum string parenthesization (OSP)-like computations that require further transformation like middle serialization which were not trivial for our code generator. E.g., Simply pulling the reduction iteration(k_2) from the innermost loop nest prohibits loop-tiling of the iteration space.

3.4 New Optimization Strategy

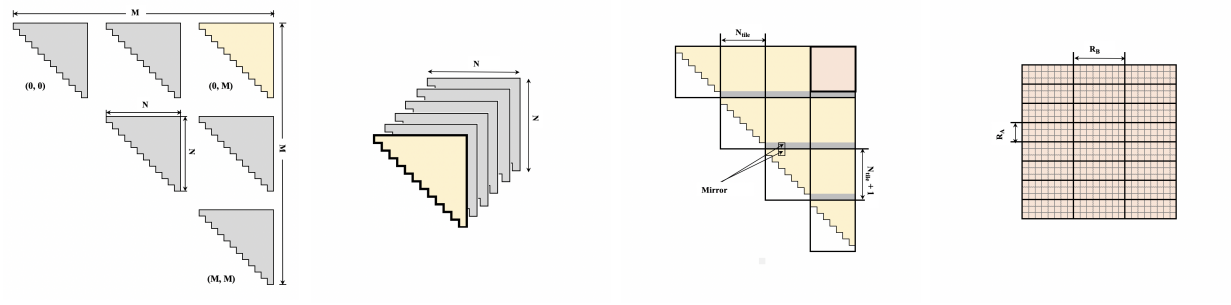
We introduce three levels of tiling in our new optimization strategy. The first-level tiling approach remains the same as our previous optimization approach, where we process each inner triangle as a tile. However, we take a different approach for the second-level tilling and introduce a third-level tiling highlighted in Figure 5. The main objective behind the second and third-level tiles is to express most of the computation using small matrix-plus instances and then compute them in the most optimized way.

3.4.1 Second-level Tiling

The second-level tile divides the computation of each inner triangle based on mono-parametric tile size. It takes an input parameter N_{tile} as a program input and partitions each inner triangle into multiple sections/tiles (N_{sec}) where, $N_{sec} = (N + N_{tile} - 1) \div N_{tile}$, N = length of the second sequence (inner triangle). In other words, it introduces two new dimensions ($0 \leq i_2 \leq j_2 \leq N_{sec} - 1$) for each $F_{i_1,j_1,i_{22},j_{22}}$ and transforms it to $F_{i_1,j_1,i_2,j_2,i_3,j_3}(i_{22}, j_{22} \mapsto i_2, j_2, i_3, j_3)$. This transformation allows us to decompose the computations easily and schedule them efficiently using our code generation tool. Now, all the second-level diagonal tiles are triangular since they represent the edge of the inner triangle. We add additional elements to these and make them rectangular. These elements are initialized to the max identity value (MIN_FLOAT). Each non-diagonal tiles are two dimensional (i_3, j_3) rectangular matrices. We add a row to each one of these tiles (i_2, j_2) to copy the first row of the tile (i_{2+1}, j_2). Thus the effective dimension of each one of second-level tile is $(N_{tile} + 1) \times N_{tile}$. It allows us to express all the BPMax reductions (R^0, R^1, R^2, R^3, and R^4) into many small matrix max-plus instances. Elements of each second-level tile are stored in row-major order, and the tiles themselves are stored in row-major order.

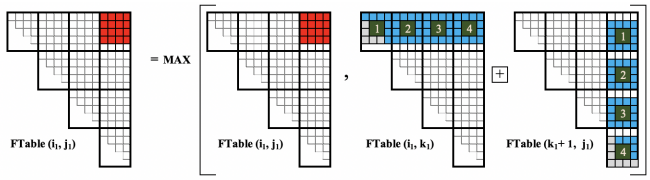
Mono-parametric Tiling of R^0: Mono-parametric tiling at the second-level transforms the original double max-plus into multiple matrix-plus operations. Figure 6a shows a triangular matrix-max plus instance corresponding to a first-level tile, and Figure 6b highlights the accumulation of a second-level tile (highlighted in red) using multiple small matrix max plus instances. The input and output tiles are distinct for each one of the second-level tile. Thus, the processing order could be either any row-major or column-major, or even reverse. It is possible to either accumulate the results for each output tile or process one tile at a time. If we compute each tile at a time, all the input tiles are used only once for computing the output, resulting in poor data locality. We avoid this by partially accumulating results for a tile by reusing one of the matrices, which is implemented using a second-level tile schedule.

Mono-parametric Tiling of R^3 and R^4: Mono-parametric tiling at the second-level for R^3 and R^4 is

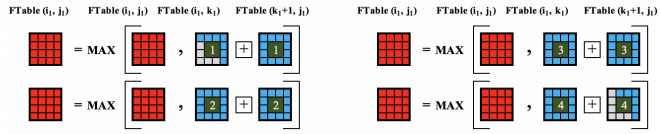


(a) FTable (b) Tile Level-I (c) Tile Level-II (d) Tile Level-III

Fig. 5: Complete Tiling Overview: Computation granularity of the first-level tile ($N \times N$) is the inner triangle. The computation granularity of the second-level tile ($N_{tile+1} \times N_{tile}$) is the mono-parametric section of each inner triangle. Finally, the computation granularity of the third tiling level ($R_A \times R_B$) is the inner triangle subsection using registers.

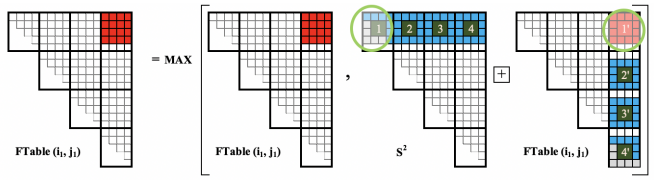


(a) Second-level tiling of R^0

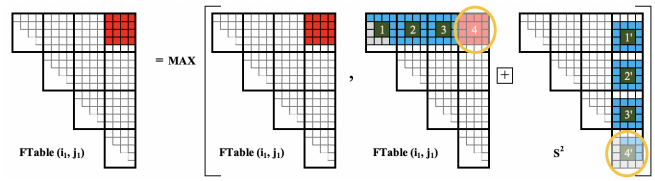


(b) Accumulation of R^0 using matrix max-plus

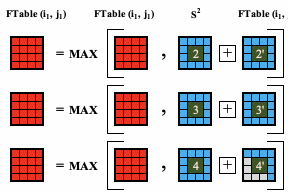
Fig. 6: Multi-level Decomposition of R^0



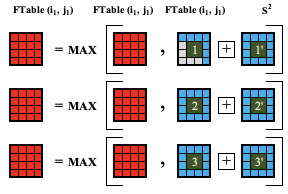
(a) Second-level Tiling of R^1



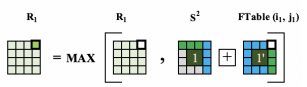
(b) Second-level Tiling of R^2



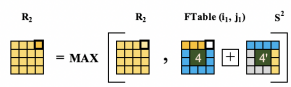
(c) $R^1 \rightarrow$ Matrix max-plus



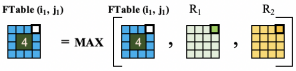
(d) $R^2 \rightarrow$ Matrix max-plus



(e) Residual R^1



(f) Residual R^2



(g) Final Update

Fig. 7: Multi-level Decomposition of R^1 and R^2

important since they use the same inner triangles used in R^0. These are element-wise operations between R^0 input tiles and S_1. R^0, R^3, and R^4 reductions share some input tiles for a given output tile. Equation 4 and 5 highlights recurrence for a second-level tile corresponding to R^3 and R^4. Tiling these computations at the second-level allows us to schedule the second-level tiles such that they share the same input tiles between R^0, R^3, and R^4 and improve data locality.

$$R^{(3)}_{i_1,j_1,i_2,j_2,i_3,j_3} = \max_{k_2=i_1}^{j_1-1} S_{i_1,k_1} + F_{k_1+1,j_1,i_2,j_2,i_3,j_3} \quad (4)$$

$$R^{(4)}_{i_1,j_1,i_2,j_2,i_3,j_3} = \max_{k_2=i_1}^{j_1-1} F_{i_1,k_1,i_2,j_2,i_3,j_3} + S_{k_1+1,j_1} \quad (5)$$

Mono-parametric Tiling of R^1 and R^2: One of the significant advantages of the new second-level tiling is that it enables the transformation of the two inner reductions R^1, and R^2 corresponding to $F(i_1, j_1)$ that significantly reduces complex atomic updates highlighted in Figure 4. After applying the second-level tile, we observe that each output tile has inter-tile and intra-tile dependencies. The inter-tile dependencies can be resolved by processing the tiles diagonally or bottom-up, and the intra-tile dependencies can be resolved using R^1, and R^2. Notice that majority of the R^1 and R^2 computations for each output tile can be transformed into several small matrix max-plus instances followed by a finalize phase that performs a small amount of R^1, and R^2. Like R^0 optimization, we can use a second-level tile schedule for these multiple matrices max-plus operations to reuse one of the input and partially accumulates results. Besides finalize phase, the diagonal tiles also require the R^1 and R^2 recurrences and use the same steps highlighted in Figure 4. Figure 7 show how a tile of $F(i_1, j_1)$ is computed by transforming the computations

into many small matrix max-plus problem instances and residual computations before making the final update.

3.4.2 Third-level Tiling

Second-level tiling allows us to improve data locality significantly. Now, we can rely on the vectorization process to improve CPU resource utilization and reduce L1 bandwidth by a factor of SIMD width. However, further optimization is required to reduce the L1 bandwidth and achieve maximum CPU utilization. So, we apply the register-blocking/tiling, where we compute a patch (third-level tile) of the second-level tile that fits in the vector register. Elements from the input matrices are loaded into the memory and used multiple times to update the patch, effectively reducing the memory accesses for the input and output.

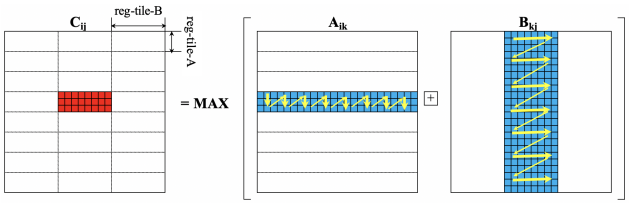


Fig. 8: Register Tiling

Sequential memory access is a well-known property of any modern CPU-specific register-tiled kernel. Its performance depends on how the scalar and vector inputs are read from the memory and their alignments. So, we transform the first input matrix (let us call this A) and the second input matrix (let us call this B) such that the memory access pattern from the register-tile kernel is sequential. Figure 8 shows the memory access pattern of our register tile. Previously, similar techniques have also been implemented for a register tile that performed FMA operations by Huang et al. [32]. We observe that an inner-F-table triangle or S_2 can be used several times as a A or B operand. Thus, it is possible to transform each triangle with two different memory layouts once and avoid transforming the same tile multiple times. We have explored three buffer transformation techniques for accessing data sequentially within the register-tiled code. They are based on when we transform **register-tile-operand-A** and **register-tile-operand-format-B**. The first one **[MPT+RT]:v1** transforms each inner F-table and S_2 to **register-tile-operand-A** and **register-tile-operand-format-B** exactly once but introduces four new F-table variables - $F(A)$, $F(B)$, $S_2(A)$, $S_2(B)$ in the system. The inner reductions R_1 and R_2 also use S_2 as the other operand for the max-plus operation. **[MPT+RT]:v2** uses on the fly transformation for both of the operands, and **[MPT+RT]:v3** uses pre-transformed S_2 but transforms the inner F-table on the fly.

Parallelization Strategy: We process the first-level tiles diagonally and assign all the cores to a first-level tile to accumulate the results from each instance of outer reductions - R^0, R^3, and R^4. Each core is responsible for processing all the second-level tiles of a particular row. It helps the cores share the input and out inner triangles in the L3 cache. After all the first-level tiles in a diagonal is accumulated from outer reductions, we assign each core to update the first-level tile with the inner reductions - R^1, R^2.

4 Implementation

In In this section, we present the code generation process using AlphaZ and key insights into some of our implementation techniques. We discuss formulation of different schedules, implementation of different optimization strategies with the AlphaZ, and optimization of matrix max-plus handwritten kernel.

4.1 Alphaz

Alpha [33] is a strongly typed functional language based on systems of affine recurrence equations defined over polyhedral domains. It was developed by Mauras [33] in 1989. Subsequently, it was extended to include subsystems and reductions [34], [35], [36], [37], [38]. 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 [18] , 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 [16], Hailde [17], and AlphaZ [19] 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.

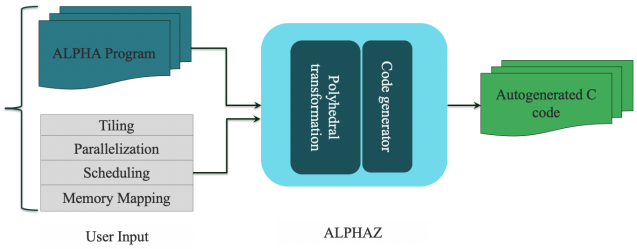


Fig. 9: Code generation methodology

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 9 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 1.

Algorithm 1 Matrix Multiplication in Alphabets

```
1: affine MM {N, K, M | (M, N, K) > 0}
2: input
3:     float A {i, j | 0 ≤ i < M && 0 ≤ j < K} ;
4:     float B {i, j | 0 ≤ i < K && 0 ≤ j < N} ;
5: output
6:     float C {i, j | 0 ≤ i < M && 0 ≤ j < N};
7: local
8:     //local variables
9: output
10:    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",
9: . "(i, j, k ↦ i, k, j)", "(i, j ↦ i, −1, j)");
10: setParallel(prog, system, "", "0" );
11:
12: // Step − 3 : Generate code
13: generateWriteC(prog, system, outDir);
14: generateScheduleC(prog, system, outDir);
```

Listing 1: 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));
        }
      }
    }
}
```

Subsystems: One of the primary challenges of using a polyhedral code generator is to produce readable, modular code. ALPHAZ subsystem construct is handy for addressing this. It helps organize a complex ALPHA program into different parts capable of taking one or more ALPHA variables as input to produce an output. ALPHAZ treats the subsystem itself like a variable to allow the user to specify a schedule for controlling the invocation and a memory map for optimizing variable passing between subsystem caller and callee. The subsystem invocations can be precisely controlled for any point in the iteration space.

4.2 Previous BPMax Schedules

Original BPMax Schedule: Let us recall that the original BPMax computed a four-dimensional table F-table based on five reductions (R^0, R^1, R^2, R^3, R^4) and two two-dimensional tables - $S^{(1)}$ and $S^{(2)}$. ALPHAZ treats each of these entities as a unique variable and requires the user to specify a schedule and a memory map. We have observed previously that $S^{(1)}$ and $S^{(2)}$ require only the input sequences. Thus, they can be scheduled before any other reductions. The schedule of the remaining variables are formulated based on the wavefront parallelization of the 6-D schedule space. We highlight the original program schedule in Table 1.

TABLE 1: BPMAX ORIGINAL PARALLELIZATION

Reduction	*Schedules*
$S^{(1)}$	$(i_1, j_1, k_1 \mapsto 0, j_1 - i_1, i_1, k_1, 0, 0, 0)$
$S^{(2)}$	$(i_2, j_2, k_2 \mapsto 0, j_2 - i_2, i_2, k_2, 1, 1, 1)$
R^0	$(i_1, j_1, i_2, j_2, k_1, k_2 \mapsto 1, j_1 - i_1, j_2 - i_2, i_1, i_2, k_1, k_2)^{a}$
R^1, R^2	$(i_1, j_1, i_2, j_2, k_2 \mapsto 1, j_1 - i_1, j_2 - i_2, i_1, i_2, k_2, 0)^{a}$
R^3, R^4	$(i_1, j_1, i_2, j_2, k_1 \mapsto 1, j_1 - i_1, j_2 - i_2, i_1, i_2, k_1, 0)^{a}$

[a]Parallel Dimension 3 (1-based)

We also recall the most optimized schedule from our prior work [21] shown in Table 2.

4.3 New Optimization Technique

We used the BPMax equation as specified in the original paper [3] in our previous optimization work [21]. In our current optimization work, we manually transform the original

TABLE 2: BPMAX HYBRID SCHEDULE WITH TILING

	Variable	*Schedule*
a	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	a	$(i_1, j_1 \mapsto 1, j_1 - i_1, i_1, j_1 - 4, 0, 0, 0)$
	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)

BPMax equations to apply the second and third-level tiling based on a mono-parametric tile parameter and develop a modified ALPHAprogram. It allows precise scheduling, memory mapping, and transformations of the second-level tiles. We express various computations using subsystems and apply optimization on each of them independently to produce modular code.

Similar to the prior optimization work, we compute the $S^{(1)}$ and $S^{(2)}$ table first. However, we apply mono-parametric tiling on $S^{(2)}$ so that we can use a second-level tile as one of the matrix max-plus operands for computing R^1 and R^2. Let us now discuss the F-table schedules at the different tilling levels.

First-level Tile Schedule: Each first-level tile (an inner triangle) goes through different computation phases (point-wise operations and reductions) associated with a unique subsystem. The point-wise operations initialize (subsystem - $Initilization$) the F-table and update (subsystem - $Pointwise^{sw}$) using the inner triangle to its south-west. Subsystem - $Reductions^{outer}$ accumulates the result from the outer reductions - R^0, R^3, R^4 and subsystem - $Reductions^{inner}$ makes the final update to each tile using the inner reductions - R^1, R^2. We process diagonal tiles one at a time for invoking point-wise operations ($Initilization$ and $Pointwise^{sw}$) and outer reductions $Reductions^{outer}$. $Reductions^{outer}$ accumulates the results from a set of $F(i_1, k_1)$ and $F(k_1 + 1, j_1)$ triangles where $i_1 \leq k_1 < j_1$. 2^{nd} and 3^{rd} dimensions of these subsystem's schedules control the diagonal processing order. Notice that initialization of the entire F-table is costly due to its footprint. So, we schedule $Initilization$ and $Pointwise^{sw}$ together (5^{th} and 6^{th} dimension of the schedule) before scheduling the $Reductions^{outer}$ (ordering is controlled by the 4^{th} dimension of the schedule). After completing all the outer reductions, we schedule $Reductions^{inner}$ (3^{rd} dimension greater than $M-1$) for multiple tiles simultaneously. Table 3 outlines the schedule for each one of these subsystems along with $S^{(1)}$ and $S^{(2)}$. Note that the parallel dimension 5 indicates that multiple threads are assigned to do point-wise operations on a particular tile, whereas each tile is finalized ($Reductions^{inner}$) by one thread.

TABLE 3: FIRST LEVEL TILE SCHEDULE

Subsystem	*Schedule*[a]
$S^{(1)}, S^{(2)}$	$(i_1 \mapsto 0, 0, i_1, 0, 0, 0, 0)$
$Initialization$	$(i_1, j_1, i_2, j_2 \mapsto 2, j_1 - i_1, i_1, -1, i_2, j_2, 0)$
$Pointwise^{sw}$	$(i_1, j_1, i_2, j_2 \mapsto 2, j_1 - i_1, i_1, -1, i_2, j_2, 1)$
$Reductions^{outer}$	$(i_1, j_1, k_1 \mapsto 2, j_1 - i_1, i_1, k_1, 0, 0, 0)$
$Reductions^{inner}$	$(i_1, j_1, k_1 \mapsto 2, j_1 - i_1, M, 0, i_1, 0, 0)$

[a]Parallel dimension 5

Second-level Tile Schedule: Second-level tiles are processed by $Reductions^{outer}$ and $Reductions^{inner}$ subsystems. $Reductions^{outer}$ accumulates partial results for all the second-level tiles $\{i_2, j_2 \mid 0 \leq i_2 \leq j_2 \leq N_{sec} - 1\}$ for a given first-level tile $F(i_1, j_1)$. It is responsible for scheduling R^3 (subsystem - R^3_t), R^4 (subsystem - R^4_t), multiple matrix max-plus input transformations (subsystem - $MT(F)^A$ and $MT(F)^B$), and multiple matrix max-plus computations (subsystem - MMP). For matrix max-plus operation, each second-level tile $F(i_1, j_1, i_2, j_2)$ is updated using a set of input tiles $F(i_1, k_1, i_2, k_2)$ and $F(k_1 + 1, j_1, k_2, j_2)$ where $i_2 \leq k_2 \leq j_2$. As noted earlier, instead of evaluating one output tile at a time, we use a schedule that accumulates results in the output tile. Since R^0, R^3, R^4 share the input tiles, we first schedule the R^3_t and R^4_t and reuse the input tiles in MMP. We schedule $MT(F)^A$ to transform an input tile and use it multiple times as the first operand in a MMP invocation. Before each MMP invocation, we schedule $MT(F)^B$ to transform the second operand of the matrix max-plus operation. Table 4 highlights the schedule for the different subsystems which are invoked from $Reductions^{outer}$.

TABLE 4: SECOND LEVEL TILE, OUTER REDUCTIONS

	Subsystem	*Schedule*[b]
	R^3_t, R^4_t	$(i_2, j_2 \mapsto 0, i_2, j_2, 0, 0, 0)$
a	$MT(F)^A$	$(i_2, j_2 \mapsto 0, i_2, j_2, 0, 0, 1)$
	$MT(F)^B$	$(i_2, j_2, k \mapsto 0, i_2, k_2, 1, j_2, 0)$
	MMP (R^0)	$(i_2, j_2, k_2 \mapsto 0, i_2, k_2, 1, j_2, 1)$

a - Optimized R_0
b - Parallel dimension 5

TABLE 5: SECOND LEVEL TILE, INNER REDUCTIONS

	Subsystem	*Schedule*
	$DiagonalTile$	$(i_2, j_2 \mapsto -i_2, j_2, 0, j_2, 0)$
a	$MT(S^2)^A$	$(i_2, j_2 \mapsto -i_2, j_2, 0, j_2, 1)$
	$MT(F)^B$	$(i_2, j_2, k_2 \mapsto -i_2, k_2, 3, j_2, 0)$
	MMP (R_1)	$(i_2, j_2, k_2 \mapsto -i_2, k_2, 3, j_2, 1)$
b	$MT(F)^A$	$(i_2, j_2 \mapsto -i_2, j_2, 4, j_2, 0)$
	$MT(S^2)^B$	$(i_2, j_2, k_2 \mapsto -i_2, k_2, 5, j_2, 0)$
	MMP (R_2)	$(i_2, j_2, k_2 \mapsto -i_2, k_2, 5, j_2, 1)$
	$Finalize$	$(i_2, j_2 \mapsto -i_2, j_2, 0, j_2, 0)$

a - Optimized R_1
b - Optimized R_2

$Reductions^{inner}$ subsystem takes a first-level tile $F(i_1, j_1)$ and S_2 as input and makes the final update to the $F(i_1, j_1)$. It is responsible for scheduling a diagonal tile (subsystem - $DiagonalTile$), optimized R_1 (subsystem - $MT(S^2)^A$, $MT(F)^B$, MMP), optimized R_1 ($MT(F)^A$, $MT(S^2)^B$, MMP) and residual patch up computation ($Finalize$). Table 4 highlights a bottom-up and left to right schedule for these subsystems. So, we first schedule the $DiagonalTile$ tile corresponding to each tile row. For each non-diagonal tile, we schedule all the subsystems that optimize the R^1, followed by R^2. Scheduling these subsystems is similar to optimizing R^0. Finally, we schedule $Finalize$ for each non-diagonal tile to resolve the intra-tile dependencies.

TABLE 6: REGISTER ALLOCATION STRATEGY

Number of A	Number of B	Number of YMMs for A	Number of YMMs for B	Number of YMMs for accumulations	Total YMMs Usage	Number of Memory access	Number of Max-plus Operations(v)
2	24	2	3	6	11	5	6
2	32	2	4	8	14	6	8
3	24	3	3	9	15	6	9
3	16	3	2	6	11	5	6
4	16	4	2	8	14	6	8

Third-level Tile Schedule: We implement the third-level tile using an optimized hand-written register-tiled kernel that performs matrix max-plus. We process the third-level register tile in a column-major order ($i_3, j_3 \mapsto j_3, i_3$). The design of the register-tiled kernel is target-dependent. We use intel intrinsic APIs to compute multiple max-plus operations using Intel Advanced Vector Extensions (AVX-256) registers.

Due to many architectural similarities, we use a common register-tiling implementation for our target architectures - Broadwell and Coffee Lake architecture. One of the main differences between these two architectures is the number of available floating-point addition units (FPA) per core. Even though both 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 and the number of instructions per cycle for vaddps and vmaxps are twice smaller for Broadwell than Coffee Lake, Broadwell architecture is significantly bottle-necked for the max-plus computation. The objective of the register tiling is to load the data into the registers and perform as many operations as we can without accessing memory. AVX-256 has sixteen 32-bit registers YMM0-YMM15, which perform a single instruction on multiple data elements. Each YMM register can hold eight single-precision floating points and be used to store operands or results to perform eight single-precision operations. The execution latency of vaddps and vmaxps operation on Coffee Lake is four cycles (3 for the Broadwell). Thus, we need to have 8 (6 for the Broadwell) independent chains of computations to fully-utilize both FPA execution ports for Coffee Lake.

We are interested in a data access pattern of $C = (a + B) \max C$, where a is a scalar and B, C are vectors. So, we load eight consecutive elements (vector) of B and C into the YMM registers (B, C) but load a single element (scalar) of a and broadcast it to a YMM register(A). The goal is to find the combination of A and B that maximizes CPU-resource utilization. Table 6 shows the different register tiles ($A \times B$) that maximize the resource utilization but minimize the AVX register allocation. We notice that 3×24 maximizes the resource utilization with the best memory access to compute ratio.

Memory Access: We implement several techniques to optimize memory access. Each core is responsible for executing a complete matrix max-plus operation that requires data transformation. We use dedicated buffer for these transformations and select them using omp_get_thread_num(). We ensure the processor-to-memory affinity by setting OMP_PLACES to 'cores' and set OMP_PROC_BIND to true to bind the OMP threads to physical core to improve data locality during the on the fly memory transformation. We use intel intrinsic for allocating these transformation buffer so that they are aligned to SIMD width (8×4 bytes).

5 RESULT

We present our results with two intel target architectures - Xeon E-2278G and Xeon E5-1650v4. Both architectures run AlmaLinux 8.5, where we have performed our experiments. Table 4.1 highlights the CPU properties. We use Intel compiler ICC 19.1.3.304 with -O3 -qopenmp -xCORE-AVX2 -ipo options as a compiler flag.

TABLE 7: 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 GFLOPS is typically used to highlight double-precision performance, we will use this terminology to denote the single-precision performance in the rest of our discussion. We first present the theoretical machine peak, then the roofline machine peak measured by the Intel Advisor before presenting the performance of the Double max-plus and BPMax.

5.1 Max-plus Machine Peak Analysis

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

$$\begin{aligned}\text{Single-precision Machine Peak (GFLOPS)} = \\ \text{Number of vector operations} \times \text{Instructions per cycle} \times \\ \text{Number of Cores} \times \text{Core Frequency (GHz)}\end{aligned} \tag{6}$$

Table 8 highlights the theoretical machine peak of our target architectures. 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.

Arithmetic Intensity of BPMax: BPMax computation can be summarized as $Y = max(a + X, Y)$. It uses three

TABLE 8: MAX-PLUS THEORETICAL MACHINE PEAK

Processor	Number of Cores	Frequency (GHz) [Base, Turbo]	Instructions per cycle [add, max]	Machine Peak Single Core (GFLOPS) (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]

single-precision memory accesses to perform two arithmetic operations (max and plus). So, its arithmetic intensity (AI) is $\frac{2}{(3\times4)}$or $\frac{1}{6}$ FLOPS/byte.

Coffee Lake Roofline: Figure 10 shows the max-plus roofline collected on the Coffee Lake machine using the Intel Advisor tool. We observe that the attainable max-plus machine peak of Coffee Lake in our environment is around 585 GFLOPS, which is significantly lower than the theoretical max-plus machine peak of 640 GLOPS with the processor running at maximum frequency.

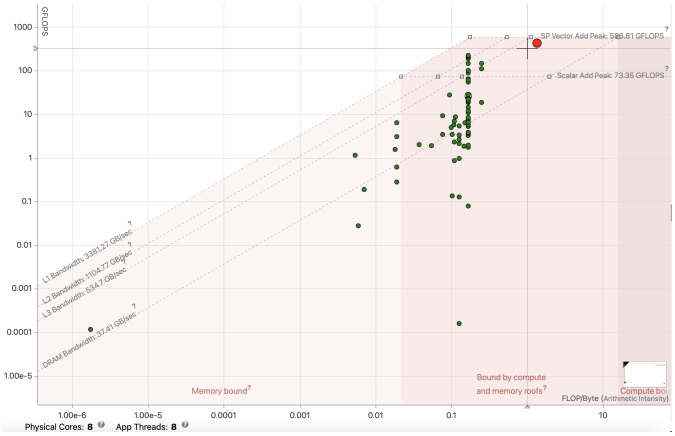


Fig. 10: Xeon E-2278G (Coffee Lake) roofline for max-plus

Broadwell Roofline: Figure 11 presents the Broadwell roofline model. From this roof line model, we observe that the attainable max-plus machine peak of Broadwell is around 165 GFLOPS, which is 85% of the theoretical machine peak (192 GFLOPS) with the processor running at maximum frequency.

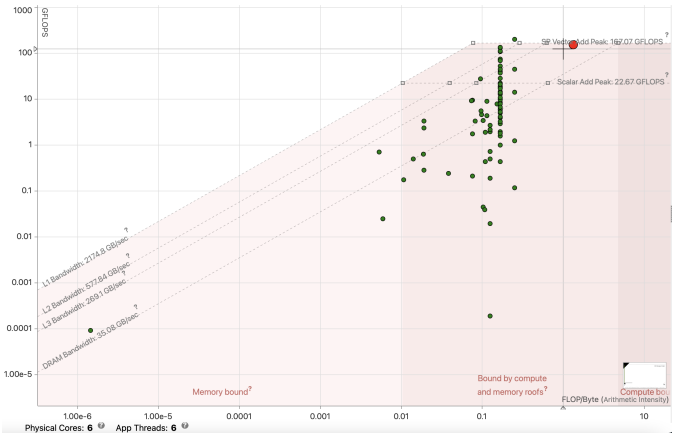


Fig. 11: Xeon E5 1650v4 (Broadwell) roofline for max-plus

5.1.1 *Tile Parameter Exploration*

We use three levels of tiling in our optimization. The size of the first-level tile depends on the second input sequence, which is a program parameter. That leaves us to explore second and third-level tiling parameters. We choose matrix-max plus operation on two matrices of size $M \times K$ and $K \times N$ to explore the best register tile and mono-parametric tile parameter. We choose square matrices $(M = N = K = 2800)$ such that the footprint exceeds the L3 cache. Figure 12 compares performance between different register tiles on Coffee Lake. We tile the three outer loops M, K, and N with a mono-parametric tile size of 192. Then, we register-tile each patch. The register-tile kernel assumes that the data is accessed sequentially. All the results shown here include the packing operation of these patches. We notice that the

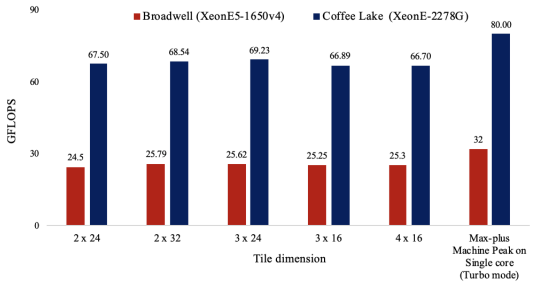


Fig. 12: Register Tiled Matrix max-plus Performance Comparisons. $M = K = N = 2880$, $N_{tile} = 192$

register-tile 3×24 performs the best which matches our theoretical register allocation strategy. To explore second-level tile size, we fixed the register tile parameters and vary the second-level tile size [48, 72, 120, 192]. We find that the second-level tile size of 48 performs better than the others for double max-plus and BPMax when the register-tile size is 3×24. When $N_{tile} = 48$, all the inputs and outputs of the register-tiled kernel fit in the L_1. So, we use $N_{tile} = 48$ and register-tile dimension of 3×24 for the rest of our experiments.

5.2 Double Max-plus Performance Improvement

5.2.1 *Single Core Performance*

Figure 13 presents the performance of the double max-plus on single thread of Xeon E5-1650v4 (Broadwell) and Xeon E-2278G (Coffee Lake). We compare the performance between the best optimized previous version [21] and the current version with the diagonal schedule. Figure 13 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

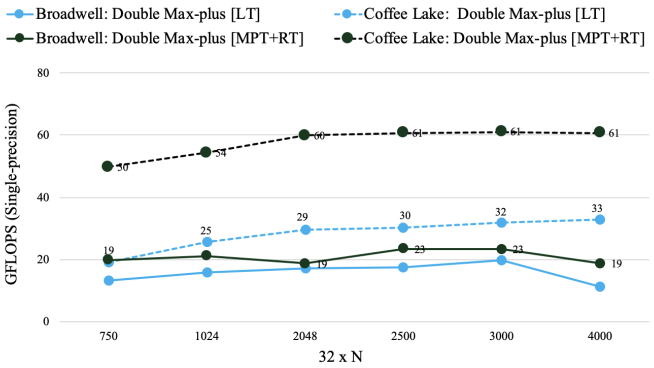


Fig. 13: Double Max-plus Single Core Performance

performance. The previous best-optimized version of the program highlighted in sky-blue reaches about 50% of the roofline machine peak on both Broadwell (dotted blue line) and Coffee Lake (dotted dark-green line), whereas the current best-optimized version attains over 90% and 80% of the max-plus roofline machine peak on Broadwell (dark-green continuous line) and Coffee Lake (dark-green dotted line), respectively. Both versions of the program performed poorly on Broadwell when N was larger than 3000. It is due to the F-Table memory footprint becoming close to Broadwell's DRAM capacity (16 GB) when $M = 32$, $N = 4000$, triggering swapping (disk-access) that reduces CPU resource utilization. We do not see this behavior on Coffee Lake.

5.2.2 *Multi-Core Performance Comparison*

For our experiments on multi-core, we choose three different values of M (16, 25, 32) and five different values of N (750, 1024, 2048, 2500, 3000) to measure the double max-plus performance for each combination of M and N. Figure 14 and Figure 15 show the performance and speedup comparisons of double max-plus between the base schedule, previously optimized best version ([LT]), and current best-optimized version ([MPT+RT]) with eight threads on Coffee Lake. The performance of the original code is about 1 GFLOPS, highlighted in dark red. The yellow color represents the

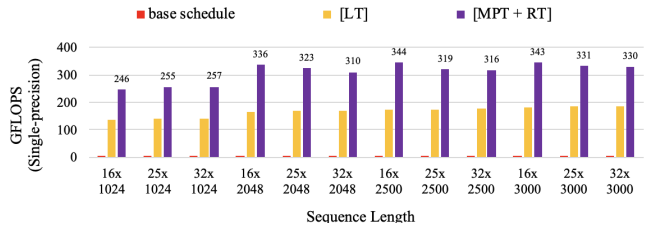


Fig. 14: Double Max-plus Performance, Coffee Lake

performance corresponding to the [LT]. It attains a maximum performance of 187 GFLOPS (32% of the roofline machine peak) on Coffee Lake. The best [MPT+RT] version, represented by the purple color, reaches a peak performance of 344 GFLOPS which is 58% of the roofline machine peak. These correspond to a speed up of 223× and 394× over the implementation available in the original BPMax program.

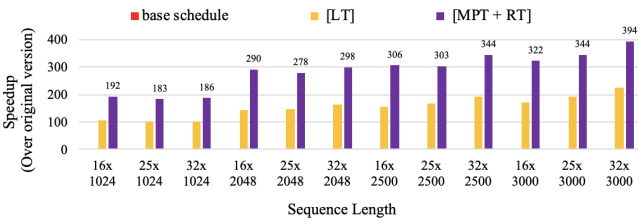


Fig. 15: Double Max-plus Speedup, Coffee Lake

5.3 BPMax Performance Improvement

We have chosen input parameters similar to Double max-plus to measure the BPMax performance.

5.3.1 *Single-Core Performance*

Figure 16 shows the single core performance of the best BPMax version, **[MPT+RT]:v3** with $M = 32$ and N varying between (750, 1024, 2048, 2500, 3000, 4000). We attain about 80% and 85% of the roofline max-plus machine peak on Coffee Lake and Broadwell, respectively.

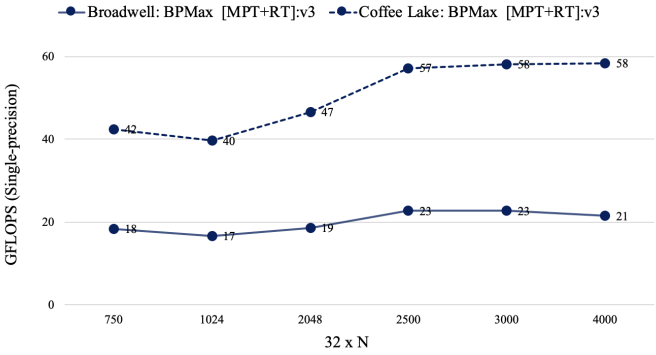


Fig. 16: BPMax Single Core Performance

5.3.2 *Multi-Core Performance*

Figure 17 and 18 show the performance improvements and speedup of various versions of BPMax program on Coffee Lake with 8 threads. We use the original implementation (BPMax original) as the reference highlighted in red. The performance and speedup with the best optimized previous version [21] ([LT]) is highlighted in brown. [LT] earlier achieved 100× speedup for longer sequence lengths. The last four data points use different versions of the current work.

The first data point [MPT] with our new implementation is highlighted in purple, which does not employ the register tiling but uses mono-parametric tiles for the second-level tile. It employs similar program transformations like all the versions of the [MPT] with register tiling to improve data locality for each tile. We observe that a mono-parametric tile size of 192 works better across all the inputs. [MPT] shows no performance improvement over [LT] because the best tile size was not long enough for effective vectorization.

We have experimented with three types of data-transformation techniques with our register-tiled code.

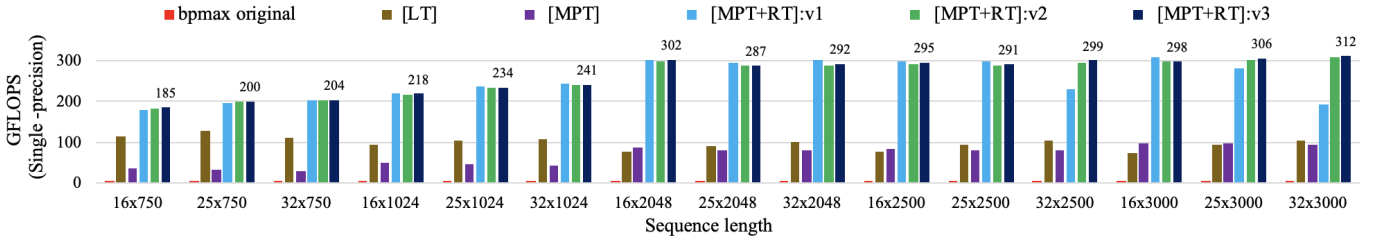


Fig. 17: BPMax performance comparison on Coffee Lake

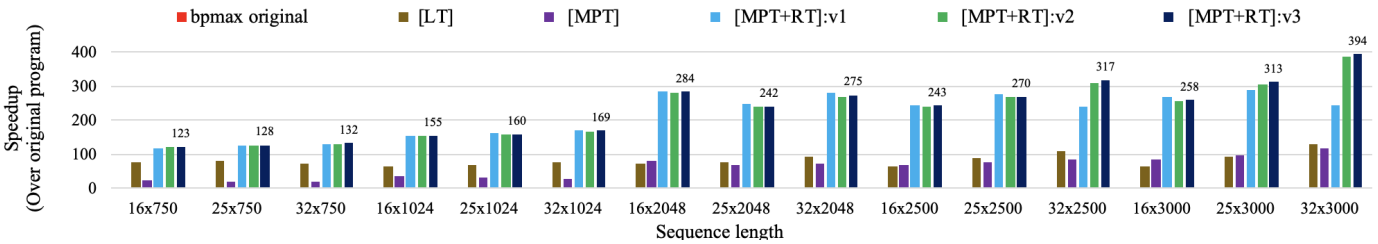


Fig. 18: BPMax speedup comparison on Coffee Lake

[MPT+RT]:v1 performs best when the memory footprint is low. As the memory footprint increases, it performs poorly. There is no significant differences between the **[MPT+RT]:v2** and **[MPT+RT]:v3**. However, the **[MPT+RT]:v3** performs consistently well across all the inputs. Figure 17 and 18 show the performance and speed up achieved by the **[MPT+RT]:v3**. It attains a peak performance of 53% of the roofline machine peak on Coffee Lake, about $400\times$ faster than the base program. Figure 10 shows the CPU usage of different parts of the BPmax program. We observe that 72% of the program's execution time (139s/187s) is spent in the register-tiled kernel, which achieved 82% of the roofline peak. The remaining 30% of the program spent significant time in vectorized code and memory initialization. However, it is important to note that additional work is done when a mono-parametric tile is filled with max-plus identity values corresponding to the triangular tiles from the edge of the inner triangle. These additional computations over the identity elements are excluded from the GFLOPS computation reported in the figures. Our polyhedral compilation scripts and source codes are available in the GitHub repository[1].

6 Conclusion

In this work, we have demonstrated the optimization process of a complete RRI program using polyhedral code generation tool. We have explored different schedules, memory maps, and tiling transformations for our optimization work using the polyhedral code generator - AlphaZ .

We have explored multi-level tiling in our optimization work. We observe that a register-tiled kernel was easier to integrate when the program was transformed using mono-parametric tiling. Also, mono-parametric tiling enabled us to tile the nearly tile-able OSP-like inner-reductions (R^1 and R^2). We achieved more than 50% of the roofline machine peak and improved the performance of the entire BPMax program by $400\times$. 70% of this work got done by the register-tiled loop. Analysis from Intel Advisor shows that this loop attained 80% of the roofline machine peak. In the future, it will be interesting to understand if there are further optimization opportunities like additional memory transformation to get close to the roofline machine peak.

We observed that double max-plus operation on single-core attained 80% of the roofline machine peak. But, performance dropped to 53% of the roofline machine peak with eight threads. So, finding opportunities to mitigate the scheduling and communication latency between the threads will be interesting. Our optimization work focused on the Intel platform. However, a future direction will be implementing the register-tiled kernel for a different processor architecture like AMD and comparing the performance improvement. 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.

1. https://github.com/chiranjeb/BPMaxCPU

References

[1] S. A. Mortimer, M. A. Kidwell, and J. A. Doudna, "Insights into RNA structure and function from genome-wide studies," *Nature Reviews Genetics*, vol. 15, no. 7, pp. 469–479, may 2014.

[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: RNA-RNA Interaction Partition Function in the Absence of Entropy," in *21st International Workshop on Algorithms in Bioinformatics (WABI 2021)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), A. Carbone and M. El-Kebir, Eds., vol. 201. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, pp. 14:1–14:24. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2021/14367

[4] 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.

[5] S. V. Rajopadhye, "Synthesis, optimization and verification of systolic architectures," Ph.D. dissertation, University of Utah, Salt Lake City, Utah 84112, December 1986.

[6] ——, "Synthesizing systolic arrays with control signals from recurrence equations," *Distributed Computing*, vol. 3, pp. 88–105, May 1989.

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

[8] ——, "The systematic design of systolic arrays," in *Automata Networks in Computer Science*, F. Fogelman Soulie, Y. Robert, and M. Tchuente, Eds. Princeton University Press, 1987, ch. 9, pp. 229–260, preliminary versions appear as IRISA Tech Reports 193 and 216, 1983, and in the proceedings of the IEEE Symposium on Computer Architecture, 1984.

[9] 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.

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

[11] ——, "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.

[12] ——, "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.

[13] U. Bondhugula, J. Ramanujam, and P. Sadayappan, "Pluto: A practical and fully automatic polyhedral program optimization system," The Ohio State University, Tech. Rep., 2015.

[14] 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.

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

[16] C. Chen, J. Chame, and M. Hall, "Chill: A framework for composing high-level loop transformations," University of Southern California, Tech. Rep., jun 2008.

[17] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation - PLDI '13*. ACM Press, 2013.

[18] 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.

[19] 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.

[20] S. Williams, D. Patterson, L. Oliker, J. Shalf, and K. Yelick, "The roofline model: A pedagogical tool for program analysis and optimization," in *2008 IEEE Hot Chips 20 Symposium (HCS)*. IEEE, aug 2008.

[21] C. Mondal and S. Rajopadhye, "Accelerating the BPMax algorithm for RNA-RNA interaction," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, jun 2021.

[22] 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.

[23] M. Zuker and P. Stiegler, "Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information," *Nucleic Acids Research*, vol. 9, no. 1, pp. 133–148, 1981.

[24] Y. Ding and C. E. Lawrence, "A bayesian statistical algorithm for RNA secondary structure prediction," *Computers & Chemistry*, vol. 23, no. 3-4, pp. 387–400, jun 1999.

[25] J. Li, S. Ranka, and S. 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.

[26] M. S. Swenson, J. Anderson, A. Ash, P. Gaurav, Z. Sükösd, D. A. Bader, S. C. Harvey, and C. E. Heitsch, "GTfold: Enabling parallel RNA secondary structure prediction on multi-core desktops," *BMC Research Notes*, vol. 5, no. 1, p. 341, 2012.

[27] D. Wonnacott, T. Jin, and A. Lake, "Automatic tiling of "mostly-tileable" loop nests," in *Fifth International Workshop on Polyhedral Compilation Techniques in conjunction with HiPEAC*, jan 2015.

[28] 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.

[29] G. Rizk, D. Lavenier, and S. Rajopadhye, "GPU accelerated RNA folding algorithm," in *GPU Computing Gems Emerald Edition*. Elsevier, 2011, pp. 199–210.

[30] 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.

[31] 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.

[32] J. Huang and R. A. van de Geijn, "BLISlab: A sandbox for optimizing GEMM," The University of Texas at Austin, Department of Computer Science, FLAME Working Note #80, TR-16-13, 2016. [Online]. Available: http://arxiv.org/pdf/1609.00076v1.pdf

[33] C. Mauras, "Alpha : un langage equationnel pour la conception et la programmation d'architectures paralleles synchrones," Ph.D. dissertation, Rennes 1, 1989.

[34] H. Le Verge, "Un environnement de transformations de programmes pour la synthèse d'architectures régulières," Ph.D. dissertation, L'Université de Rennes I, Oct 1992.

[35] ——, "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).

[36] 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.

[37] 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.

[38] 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.

Chiranjeb Mondal received his Bachelor of Engineering in Computer Science and Engineering from the National Institute of Technology, Durgapur, India, in 2004 and his Master of Science in Computer Science from Colorado State University, USA, in 2020. Currently, he is a Ph.D. student working on performance improvement using polyhedral compilation.

Sanjay Rajopadhye received his Bachelor of Technology (honours) in Electrical Engineering from the Indian Institute of Technology, Kharagpur, in 1980, and the Ph.D. in Computer Science from the University of Utah in 1986. He held academic positions at the University of Oregon, Oregon State University, and IRISA, Rennes. He is currently a professor in the Computer Science (CS) and in the Electrical and Computer Engineering (ECE) departments at Colorado State University. He is one of the inventors of the polyhedral model. His Ph.D. dissertation made three key contributions to the foundations of the polyhedral model: scheduling, locality, and closure.