A Parallel and Vectorized Implementation of the McCaskill Algorithm for x86-64 and RISC-V Architectures

Marek Palkowski[1 [0000−0002−5932−4523]], Sergio Iserte[2 [0000−0003−3654−7924]], Tomasz Olas[3 [0000−0002−7286−8023]], Mateusz Gruzewski[1 [0000−0002−9419−2749]], and Roman Wyrzykowski[3 [0000−0003−0328−2034]]

[1] West Pomeranian University of Technology in Szczecin, Zolnierska 49, 70721 Szczecin, Poland, mpalkowski@zut.edu.pl
[2] Barcelona Supercomputing Center, Plaça Eusebi Güell 1-3, 08034 Barcelona, Spain, sergio.iserte@bsc.es
[3] Czestochowa University of Technology, Dabrowskiego 69, 42201 Czestochowa Poland, olas@icis.pcz.pl

Abstract. In this paper, we study cache-efficient optimization and vectorization for the dynamic programming McCaskill algorithm, which computes the RNA partition function and base-pairing probabilities under a thermodynamic model. The McCaskill algorithm operates on the full ensemble of possible RNA secondary structures without pseudoknots, weighting them by their free energies to derive both the partition function and base-pairing probabilities. This task belongs to classical bioinformatics applications commonly benchmarked on multicore HPC systems. Well-known manual and automatic polyhedral code optimizations dedicated to dynamic programming tasks often avoid vectorization and focus solely on efficient loop tiling. As a result, such codes fail to exploit the vector computational capabilities of x86 processors and those available in newer architectures such as RISC-V. In this work, we demonstrate the generation of a readable, efficient OpenMP implementation of the McCaskill algorithm that leverages vector-level and cache-aware optimizations, outperforming related approaches. We analyze vectorization limitations on platforms lacking auto-vectorizing compilers, such as oneAPI, and propose an efficient implementation based on RISC-V vector intrinsics, demonstrating that hardware capabilities significantly exceed what current compilers can utilize.

Keywords: Vectorization · RNA folding · McCaskill's algorithm · Code optimization · RISC-V · Bioinformatics.

1 Introduction

Compute-intensive applications must be rigorously optimized — through the exploitation of advanced SIMD/vector instruction sets, careful management of cache locality, and scalable multithreading — to realize the performance potential of modern multicore processors. A particularly noteworthy application

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_48

domain of high-performance computing (HPC) is dynamic programming tasks in bioinformatics, which significantly hinder both manual optimization and automated compiler-based techniques [7, 9]. Considerable attention has been devoted to optimizing dynamic programming for RNA folding [13], as it represents a fundamental case study in computational biology [6, 9, 17].

RNA folding is the process by which a single-stranded RNA molecule folds into a stable secondary structure through intramolecular base pairing, which determines its biological function. However, even basic algorithmic code, such as the Nussinov algorithm from 1978, already challenges existing automated optimization techniques on multi-core CPUs, particularly on emerging modern x86-64 and RISC-V platforms [7, 12, 16]. Modern CPU architectures require software to exploit not only parallelism but also memory locality and data-level parallelism via vectorization. This applies to both high-end x86 platforms with large caches and wide AVX-512 vectors, as well as emerging energy-efficient RISC-V architectures, where enhanced memory hierarchies and the RISC-V Vector Extension (RVV) play a similar role [14].

Recent studies, including our previous work [7] on optimizing the Nussinov algorithm across diverse multicore architectures, indicate that dynamic programming kernels are not fully memory-bound and exhibit sufficient computational intensity to benefit from SIMD execution. In this paper, we build on the insights gained from that study and extend them to a more complex computational scheme, namely the McCaskill algorithm [11]. We propose a computation scheme that enables cache-efficient and vectorization, building upon the authors' prior experience and insights from related work [9, 12, 16, 18]. We demonstrate that none of the previously proposed solutions reported in the literature exploit vectorization for this algorithm and that this benchmark has not been evaluated on the increasingly popular RISC-V platform. Furthermore, we show the substantial optimization capabilities of modern x86-64 compilers, with particular emphasis on the oneAPI framework, and overcome the current limitations of available toolchains for RISC-V architectures [14]. In the experimental evaluation, we analyze computational speedups across varying problem sizes and assess scalability on modern 32-core Intel Xeon CPU Max 9462 and 96-core AMD EPYC 9654 architectures, as well as on two distinct RISC-V systems, 8-core Banana Pi BPI-F3 and 64-core Milk-V SG2042. We also point out the limitations of existing RISC-V compilation tools and propose an implementation of the McCaskill algorithm using RVV intrinsics in order to achieve the performance offered by the hardware.

2 Related Work

The McCaskill algorithm follows a Nussinov-like scoring scheme, which in turn enables the application of similar manual [9, 10, 17, 18] and automatic (polyhedral) transformations [2, 15] to those proposed for related dynamic programming kernels. These tasks share common structural properties: they typically construct a dynamic programming (DP) table over sequence indices (i, j), where entries

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_48

correspond to subsequences and only the upper triangular region ($i \leq j$) is evaluated, and their computations can be organized into block-level wavefronts.

Fekete et al. [5] were the first to parallelize the McCaskill algorithm on a computer cluster. Li et al. [9] showed that transposing the Nussinov dynamic programming matrix improves data locality on CPUs (referred to as *Transpose*) and revealed that GPU-oriented optimizations can be interpreted analogously to matrix multiplication. This approach was later extended by Zhao and Sahni [17] through the *ByBox* and *ByRow* schemes, with particular emphasis on cache optimization, and was then applied to the McCaskill problem [18]. *ByBox* is a cache-aware blocking strategy that groups and merges multiple dynamic-programming tiles into larger boxes, enabling wavefront execution with improved cache reuse and reduced memory traffic. To the authors' knowledge, this is currently the most efficient OpenMP implementation of the McCaskill algorithm.

In parallel, polyhedral compiler techniques were also advancing, aiming to optimize cache efficiency through loop tiling. This automatic loop transformation technique partitions iteration spaces into smaller blocks (tiles) to improve data locality. The McCaskill algorithm exhibits non-uniform dependencies, which classifies it as a non-serial polyadic dynamic programming (NPDP) problem. Researchers working on polyhedral approaches have highlighted the limitations of state-of-the-art optimizers based on the affine transformation framework (ATF) in inner-tiling, such as Pluto [3], when applied to NPDP kernels. To address these challenges, manual polyhedral transformations have been proposed, including Wonnacott's separation of the iteration space into problematic and non-problematic ("mostly tileable") regions [16], albeit limited to serial implementations, as well as the work of Mullapudi and Bondhugula [12], who identified opportunities to introduce reductions when parallelizing the NPDP recurrences. The limitations of ATF also motivated the development of source-to-source compilers based on the transitive reduction of dependence graphs and time–space tiling, as implemented in the *Dapt* compiler [2].

Unfortunately, these methods were not designed to utilize CPU vector instructions, and widely used implementations of the McCaskill algorithm, such as ViennaRNA Package [10], remain non-vectorized in their current form, relying on pointer-based data structures, complex control flow, and irregular memory access patterns that hinder effective SIMD vectorization. In our previous work, we proposed an optimization strategy for the Nussinov kernel that enables vector-level execution within a thread [7] on x86 and GPU platforms, and we subsequently investigated the resulting performance gains on the RISC-V architecture [14]. The experience from this work is applied to the McCaskill algorithm, which follows a more complex computational scheme.

3 Computation of RNA Partition Functions Using the McCaskill Algorithm

The McCaskill algorithm computes the partition function $Z = \sum_P \exp\left(-\frac{E(P)}{RT}\right)$, over all possible nested secondary structures P that can be formed by a given

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_48

```
for (i = N - 1; i >= 0; i--)
 for (j = i + 1; j < N; j++) {
  Q[i][j + 1] = Q[i][j];
   for (k = 0; k < j - i; k++) {
    Qbp1[k+i][j+1]=Q[k+i+1][j]*ERT*paired(rna[k+i], rna[j]);
    Q[i][j + 1] += Q[i][k + i] * Qbp1[k + i][j + 1];
}}
```

Listing 1: McCaskill algorithm

RNA sequence S. Here, $E(P)$ denotes the free energy of structure P, R is the universal gas constant, and T is the absolute temperature [11].

In this work, we consider a simplified formulation based on a Nussinov-like energy scoring scheme, in which each base pair contributes a fixed energy term E_{bp}, independent of its structural context. Under this assumption, two dynamic programming tables, Q and Q^{bp}, are constructed.

The entry $Q_{i,j}$ represents the partition function of the subsequence spanning nucleotide indices i through j, while $Q^{bp}_{i,j}$ denotes the contribution of structures in which nucleotides at positions i and j form a base pair, and is zero when such pairing is not allowed.

The McCaskill recurrences used to compute the tables Q and Q^{bp} are given below, where l denotes the minimal loop length:

$$Q_{i,j} = Q_{i,j-1} + \sum_{i \le k < j-l} Q_{i,k-1} \cdot Q^{bp}_{k,j}, \tag{1}$$

$$Q^{bp}_{i,j} = \begin{cases} Q_{i+1,j-1} \cdot \exp\left(-\dfrac{E_{bp}}{RT}\right), & \text{if nucleotides } S_i \text{ and } S_j \text{ can form a base pair,} \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

Given these partition function terms, base-pair probabilities, as well as probabilities that a given subsequence is unpaired, can be computed according to the Boltzmann distribution [11].

As shown in Listing 1, the McCaskill kernel is implemented as a triple-nested loop with an inner accumulation over k.

4 Optimization strategy for the McCaskill Algorithm

The McCaskill algorithm is characterized by the fact that, when iterating over the k loop for a given (i, j) element, the computations can be grouped and executed in parallel within a single thread. This enables SIMD vectorization, either automatically via compiler directives or explicitly via vector instructions such as AVX on x86 architectures and RVV on energy-efficient RISC processors [14]. Hence, our objective is to achieve massive vector-level parallelization of the operations in the McCaskill algorithm.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_48

Despite structural similarities in the code of the Nussinov and McCaskill algorithms, including the ability to be blocked and parallelized along diagonals, they differ substantially in their computational characteristics. McCaskill's code operates on floating-point matrices rather than integer-valued tables and requires the coordinated traversal of multiple dynamic programming matrices, notably the Q and Q_{bp} tables. All recurrences depend on nucleotide pairing functions over the alphabet $\{A, C, G, U\}$, and the resulting dynamic programming tables are typically sparse.

Zhao and Sahni [18] observed that the computation of the element $Q[i][j+1]$ depends on the values $Q[i][j]$, $Q[k+i+1][j]$, and $Q[i][k+i]$, which are independent and can therefore be computed in parallel, referring to it as the *OneArray* technique. At this point, one can observe a clear similarity to the Nussinov algorithm: the evaluation of the target element relies on the pair $Q[k+i+1][j]$ and $Q[i][k+i]$, while the element $Q[i][j]$ is precomputed before entering the innermost loop. This observation motivates the introduction of the notion of *problematic* pairing target blocks with the current tile, as well as *non-problematic* (or *almost-tileable*) instructions, as defined in the work of Wonnacott [16].

To effectively exploit cache locality, it is necessary to introduce block-based execution. We define this polyhedral approach as follows and generate a loop using Integer Set Library (ISL) tools [7].

$$TILE_{MCC}(II, JJ) := \left\{ \begin{array}{l} [ii, jj, i, j, k] -> [ii - jj, jj, i - j, j, k] : \\ 0 \le jj \le \frac{N}{bb} \ \wedge \ \ 0 \le ii \le \frac{N}{bb} \ \wedge \\ 0 \le i < N \ \wedge \ \ 0 \le j < N \ \wedge \ \ 0 \le k < j - i \ \wedge \\ ii \cdot bb \le i < (ii + 1) \cdot bb \ \wedge jj \cdot bb \le j < (jj + 1) \cdot bb \ \wedge \\ (ii = jj \wedge j > i) \ \vee \ (jj > ii) \end{array} \right\} \tag{3}$$

The set $TILE_{MCC}(II, JJ)$ defines a two-dimensional tiling of the McCaskill dynamic programming domain, where (II, JJ) denote tile coordinates and bb is the tile (block) size. The problem size N does not need to be divisible by `bb`, as boundary tiles are handled separately.

The indices i and j range over the original $N \times N$ dynamic programming matrix, while k corresponds to the innermost summation dimension constrained by $0 \le k < j - i$. The tiling constraints $II \cdot bb \le i < (II + 1) \cdot bb$ and $JJ \cdot bb \le j < (JJ + 1) \cdot bb$ assign each (i, j) pair to a unique tile. The additional condition $((II = JJ \wedge j > i) \vee JJ > II)$ enforces execution only over the upper triangular region of the dynamic programming matrix, including the diagonal tiles with the restriction $j > i$. This formulation preserves the data dependencies of the McCaskill algorithm while enabling block-based execution and diagonal wavefront parallelism.

We use parallelization along diagonals, which corresponds to loop skewing, a typical strategy for NPDP-type problems [7]. The algorithm traversal is depicted in Fig. 1.

Each computed block (II, JJ) depends on the diagonal blocks (II, II) and (JJ, JJ), as well as on pairs of blocks (II, m) and (m, JJ), where $II < m < JJ$

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_48

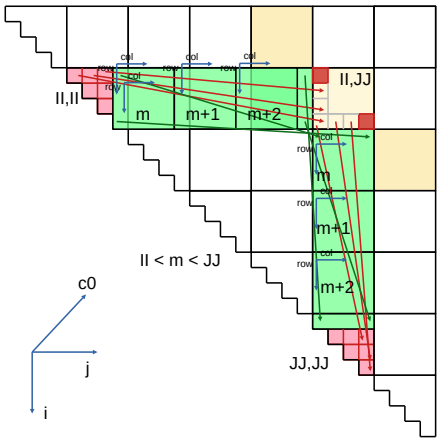

Fig. 1. Block-based (tiled) computation of the McCaskill dynamic programming tables with diagonal ordering. Indexing: (c0) denotes diagonals, (i, j) tile coordinates, while (row, col) represent indices within a tile.

is called the left and bottom blocks [18]. The block pairs $(II, JJ) \rightarrow (JJ, JJ)$ and $(II, II) \rightarrow (JJ, JJ)$ are classified as *problematic*, as they are not profitable targets for parallelization. We therefore define problematic operations as those block pairs whose operands access values from the cells of the currently computed block (II, JJ). It is worth noting that such pairs are associated with triangular blocks located on the first diagonal. In Fig. 1, these are highlighted in red, the block under computation is shown in yellow, and the non-problematic block pairs are marked in green. Consequently, the largest portion of the computation - namely the pairs of blocks (II, m) and (m, JJ) - is identified as *non-problematic* and constitutes the primary candidate for parallel execution. To generate tiled code, we apply tiled wavefront scheduling using the transformation $[II, JJ] \rightarrow [II - JJ, JJ]$, where *II-JJ* indicates the number of diagonals.

At this point, it is useful to identify several aspects essential to characterizing the parallelization strategy. Blocks located on the first and second diagonals do not contain any non-problematic block pairs; such pairs appear only starting from the third and subsequent diagonals. The largest number of non-problematic block pairs is associated with the last block on the final diagonal. Conversely, the first diagonal contains the largest number of blocks, and the degree of parallelism (i.e., the number of active threads) decreases progressively toward the last diagonal, where it is reduced to a single block.

Blocks lying on the same diagonal have the same number of non-problematic block pairs. The order in which these pairs are evaluated can be rearranged by exploiting the fact that the update of $Q[i][j + 1]$ is an addition operation and is therefore commutative and associative. Additional parallelism within non-problematic blocks can be exploited on GPUs or through vectorization on CPUs.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_48

```
// The serial diagonal loop
for (int c0 = 0; c0 <= (N-1)/bb; c0++)
  // parallel blocks on the diagonals
 #pragma omp parallel for
 for (int c1 = c0; c1 <= min((N -1)/bb, floord(N+c0-2, bb));
     c1++) {
  int jj = c1-c0;
  int ii = c1;
  double A[bb][bb], B[bb][bb], C[bb][bb], rna1[bb], rna2[bb];

  for(int row=0; row<bb; row++)
    for(int col=0; col<bb; col++)
       C[row][col] = 0;

  for (int m = jj+1; m < ii; m++) {
    // data copying
    for(int row=0; row<bb; row++){
       for(int col=0; col<bb; col++){
          A[row][col] = Q[bb * jj + row][bb*m  + col-1];
          B[row][col] = Q[bb * m + row][bb * ii + col ];
       }
       rna1[row] = RNA[bb * m -1 + row];
       rna2[row] = RNA[bb * ii + row];
    }
    //non-problematic statements
    for (int row = 0; row < bb; row++) {
       #pragma omp simd  // icpx  x86-64
       for (int col = 0; col < bb; col++) {
         double Cvalue = 0;
         #pragma omp simd reduction(+:Cval) // clang++ risc-v
           for (int e = 0; e < bb; e++){
           double v=B[e][col]*ERT*paired(rna1[e], rna2[col]);
            if(v) {
               Qbp[bb * m - 1 + e ][bb * _si + col] = v;
               Cvalue += A[row][e] * v;
            }
          }
          C[row][col] +=  Cvalue;
       }
    }
  }

// For all cells Q(i,j+1)=Q(i,j)  in the target block (II,JJ)
// add the value ( C(i%b, j%b) and the rest of paired vals in
// problem. blocks {(II, II)->(II,JJ); (II,JJ)->(JJ, JJ)}.
```

Listing 2: Parallel and vectorized McCaskill kernel (in OpenMP)

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_48

It is worth noting that when the nucleotides are unpaired, updating Q_{bp} is unnecessary, and updating Q can be omitted, as it does not contribute to the summation of the corresponding Q element. In practice, the value of Qbp is primarily determined by the outcome of RNA nucleotide pairing: it becomes nonzero only when both the Q term and the energy-related factor ERT are nonzero.

The computation of non-problematic blocks is illustrated in Listing 2. Each thread maintains a private array C, which must be initialized to zero before use. The operand values are loaded from the Q table into the auxiliary arrays A and B. Subsequently, each element of the block (II, JJ) accumulates a partial sum. At the same time, the corresponding value is written directly to the Qbp table. The number of paired non-problematic blocks can be determined from the diagonal index c_0 and is equal to $c_0 - 2$, which defines the number of iterations in the loop indexed by m.

In Listing 2, the loops in lines 25, 27, and 30 traverse the left and bottom blocks scanning indexes `{row, col, e}`, and the results are accumulated in the array C. These loops can be parallelized using a sum reduction on Q. The main difference between the proposed implementation and the *ByBox* approach of Zhao and Sahni [18] is that the target tile is not updated at every step. Instead, partial results are accumulated in local arrays; consequently, loops scanning non-problematic statements can be parallelized and preceded by appropriate pragmas, enabling SIMD vectorization of the multi-threaded code using AVX on x86 architectures and RVV on RISC-V processors.

5 Experimental Study: x86 Architecture

5.1 Methodology of Experiments

For the experimental study on x86 multicore processors, we compared our approach with the following OpenMP implementations:

- *ByBox* [18] combines pairs of left and bottom tiles to update the target tile at each step, while the base-pair table Q_{bp} is computed separately at the end.
- *Transpose* reuses the lower-left part of the array to support column-wise access, reducing memory usage and improving data locality [9].
- *OneArray* uses the rewritten recurrence relations from [18]; this scheme allows Q to be computed using a single array and enables diagonal parallelism.
- *Dapt* [2] is a polyhedral source-to-source compiler that applies 3D space–time tiling.

Other approaches, including Pluto-generated tiled code (serial-only) [3] and the *ByRow* scheme [18], were not considered further due to their extremely poor performance in practice.

On studied machines, we explicitly indicated the `#pragma omp simd` directive to the C++ compiler in order to enable vectorization of the loop iterating over the index variable *col*. For this purpose, we used the oneAPI compiler [8].

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_48

We analyze the possibilities for vectorizing the code. First, on x86 architectures, the oneAPI `icpx` compiler can vectorize the `col` loop. The variables `v` and `Cval` are private, the arrays `A` and `B`, as well as the function `paired`, are read-only, and the update of `C[row][col]` does not introduce any data dependencies. Particular attention must be paid to accesses to the `Qbp` array. Assuming that `bb`, `m`, and `_si` are constant, the write access has the form `Qbp[const + e][const + col]`, which is suitable for vectorization; therefore, these values do not overlap across threads. The value of `bb` is empirically set to 32.

5.2 Target Computing Platforms

We conducted the experiments with two multi-core x86-64 machines. The first one is equipped with an Intel Xeon CPU Max 9462 (the Sapphire Rapids HBM architecture), featuring 32 cores and 64 hardware threads, up to 3.5 GHz turbo, 150 MB L3 cache, 128 MB L2 cache, 3 MB L1d cache, and 2 MB L1i cache, as well as 128 GB of on-package HBM2e and 631 GB of DRAM. The second machine is equipped with an AMD Epyc 9654 CPU, featuring 96 cores and 192 hardware threads, operating at a base frequency of 2.4 GHz with a boost frequency of up to 3.7 GHz, and providing a cache hierarchy consisting of 64 KB L1 cache, 1 MB L2 cache per core, and 384 MB of shared L3 cache.

The code was compiled with the Intel oneAPI C++ Compiler 2025.0.4, `icpx` [8] using the flags `-O3`, `-qopt-report=max`, `-qopt-report-phase=vec`, `-march=native`, `-mavx512f` or `-mavx2`, and `-qopenmp`.

5.3 Experimental Results

Tables 1 and 2 present the measured execution times of the evaluated methods for Intel and AMD machines, respectively. Our approach, which uses vectorization only and no hyper-threading, outperforms the most efficient competing method, namely *ByBox*. Enabling AVX-512 provides a greater advantage over AVX2's 256-bit registers, demonstrating that our method benefits more from wider SIMD vectorization. For the Intel platform, the number of threads was chosen to match a configuration with hyper-threading enabled. For the AMD processor, the number of threads was set equal to the number of physical cores for related approaches. In all cases, the thread count was selected individually for each method to achieve the best possible execution time, ensuring a fair comparison that reflects the maximum performance potential of each approach. We observe that increasing the number of threads beyond the number of physical cores does not provide additional speedup for the SIMD version. The compiler report shows that the loops were successfully vectorized using SIMD, utilizing AVX-512 (8 double elements) and strip-mining for short loops. However, unaligned memory accesses and masked stores introduce overhead, reducing the achieved SIMD speedup due to memory inefficiencies and data dependencies. We also noted that the performance gap grows with problem size due to better parallelism, SIMD utilization, and cache blocking, while these effects are limited for small inputs.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_48

Table 1. Execution time (seconds) for different implementations and input sizes on Intel Xeon CPU Max 9462.

Size	Ours AVX-512 32 threads	Ours AVX2 32 threads	Ours No vec. 64 threads	ByBox 64 threads	Transpose 64 threads	OneArray 64 threads	Dapt 32 threads
1000	0.03	0.04	0.09	0.04	0.03	0.03	0.12
2500	0.13	0.17	0.63	0.24	0.19	0.21	0.63
5000	0.86	1.02	2.19	1.21	1.71	2.22	5.48
7500	2.91	3.06	5.28	3.71	6.09	9.39	20.02
10000	6.27	7.57	10.94	8.41	15.89	24.25	56.41
12500	12.83	13.76	21.01	15.02	33.25	50.99	136.38
15000	17.52	22.81	33.61	26.56	59.87	94.49	268.86

Table 2. Execution time (seconds) for different implementations and input sizes on AMD Epyc 9654.

Size	Ours AVX-512 96 threads	Ours AVX2 96 threads	Ours No vec. 96 threads	ByBox 96 threads	Transpose 96 threads	OneArray 96 threads	Dapt 96 threads
1000	0.06	0.09	0.13	0.11	0.41	0.35	0.32
2500	0.26	0.33	0.41	0.38	1.18	0.47	1.17
5000	0.79	0.99	1.33	1.22	3.37	3.84	7.34
7500	1.97	2.44	3.43	2.75	9.71	15.27	29.11
10000	3.40	4.53	7.18	5.71	22.42	45.98	80.36
12500	6.01	8.36	12.71	10.65	43.75	98.21	198.04
15000	9.25	12.89	20.69	16.86	74.62	183.35	330.21

The code generated by *Dapt*, as well as the *Transpose* and *OneArray* variants, is significantly slower than our approach. Compared to the CPU Max platform, the AMD platform offers a larger L3 cache and a higher thread count, resulting in better performance for both our approach and *ByBox*, indicating more effective processor utilization. For further investigations on the open and promising RISC-V architecture, we compared the proposed solution only with the best related method, *ByBox*.

6 Experimental Study: RISC-V Architecture

RISC-V is an open-standard Instruction Set Architecture (ISA) that enables royalty-free CPU development and a common software stack. Although current RISC-V processors offer relatively modest performance, the architecture is rapidly evolving as a general-purpose platform, increasingly targeting HPC workloads. We conducted an experimental study on two different machines: the Banana Pi BPI-F3 and the Milk-V Pioneer.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_48

```
for (int row = 0; row < bb; ++row) {
  int col = 0;
  while (col < bb) {
    size_t vl = __riscv_vsetvl_e64m8((size_t)(bb - col));
    vuint8m1_t vnt2 =
     __riscv_vle8_v_u8m1((const uint8_t*)&rna2[id][col], vl);
    vfloat64m8_t vacc  = __riscv_vfmv_v_f_f64m8(0.0, vl);
    vfloat64m8_t vzero = __riscv_vfmv_v_f_f64m8(0.0, vl);

    for (int e = 0; e < bb; ++e) {
      vfloat64m8_t vB =
       __riscv_vle64_v_f64m8(&B_elements[id][e][col], vl);
      vfloat64m8_t vtmp =
       __riscv_vfmul_vf_f64m8(vB, (double)ERT, vl);
      uint8_t nt1 = rna1[id][e];
      vbool8_t mp = paired_bits(vnt2, nt1, vl);

      vtmp = __riscv_vmerge_vvm_f64m8(vzero, vtmp, mp, vl);
      vbool8_t mBnz =  __riscv_vmfne_vf_f64m8_b8(vB, 0.0,vl);

      vbool8_t mstore = __riscv_vmand_mm_b8(mp, mBnz, vl);
      __riscv_vse64_v_f64m8_m(mstore,
            &Qbp[bb * m - 1 + e][bb * _si + col], vtmp, vl);

      double a = (double)A_elements[id][row][e];
      vacc = __riscv_vfmacc_vf_f64m8(vacc, a, vtmp, vl);
   }
vfloat64m8_t vC = __riscv_vle64_v_f64m8(&C[id][row][col],vl);
   vC = __riscv_vfadd_vv_f64m8(vC, vacc, vl);
   __riscv_vse64_v_f64m8(&C[id][row][col], vC, vl);
   col += (int)vl;
  }
}
```

Listing 3: Execution of non-problematic statements using RVV intrinsics.

6.1 Target Computing Platforms

Banana Pi BPI-F3 is an industrial RISC-V development board based on the SpacemiT K1 processor [1], featuring eight 64-bit cores clocked at 1.05 GHz with an eight-stage, in-order, dual-issue pipeline. Introduced in late 2023, the CPU complies with the RISC-V 64GCVB architecture and the RVA22 standard and is the first commodity processor to support the RVV 1.0 vector extension, providing a 256-bit VLEN with a 128-bit×2 execution width. The platform employs a simplified memory hierarchy with private 32 KB L1 instruction and data caches per core and a shared 1 MB L2 cache (two 512 KB banks), backed by 4 GB of LPDDR4-2666 memory (up to 16 GB) connected via a single memory controller

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_48

delivering up to 10.6 GB/s bandwidth. We upgraded the Bianbu OS to its third version (based on Ubuntu 25.04) and the `clang` compiler to version 20.

Milk-V Pioneer is a microATX developer motherboard based on the 64-core Sophon SG2042 RISC-V processor [4], which targets HPC workloads. The SG2042 operates at 2 GHz and is organized into 16 clusters of four XuanTie C920 cores connected via a 2D mesh network-on-chip (NoC). Each 64-bit C920 core implements a 12-stage out-of-order superscalar pipeline and supports the RV64GCV instruction set. Each core provides a private 64 KB L1 instruction and data cache, while a 1 MB L2 cache is shared within each cluster. All cores share 64 MB system-level L3 cache composed of 16 slices interconnected through the NoC. The processor integrates four memory controllers for 128GB of DDR4-3200, delivering up to 102.4 GB/s of bandwidth. The platform supports only the RVV 0.7.1 vector extension with a vector width of 128 bits.

6.2 Methods of Vectorization

With recent versions of the `clang` and `gcc` compilers supporting the RVV 1.0 vector extension, it becomes possible to rely on automatic compiler vectorization. However, on the Banana BPI-F3 platform, vectorization of the outer loop is inhibited by conservative safety checks and cost-model constraints, whereas it can be successfully vectorized on x86-64 systems. As a result, we enforce only vectorization at the innermost loop level using `#pragma omp simd reduction(+:Cval)`, explicitly declaring a reduction on the accumulation variable `Cval` (see Listing 2, line 29). In contrast, the `icpx` compiler adopts a more permissive heuristic, enabling more aggressive optimizations and relying on the programmer's assurance that the `#pragma omp simd` directive is safe for the outer loop.

The outer loop can nevertheless be vectorized using RVV intrinsics for Banana BPI-F3. The Listing 3 shows a modified loop from Listing 2 (Line 25) that loads data into vector registers corresponding to successive iterations of the `col` index variable. The computation is carried out in vector-length chunks determined dynamically by `vsetvl`. For each block of columns, nucleotide data in bit form are loaded into vector registers, and the accumulation vector `vacc` is initialized. The inner loop performs vectorized loads of the `Q` matrix, scales them by the constant `ERT`, applies a pairing mask based on the scalar nucleotide, and masks out invalid entries. Valid, nonzero results are selectively written to `Qbp` using masked stores, while fused multiply–accumulate operations update `vacc`. After the reduction over `e`, the accumulated vector is added to the corresponding entries of the `C` matrix and stored back to memory, and the loop proceeds to the next vector chunk with eight contiguous vector registers (*LMUL = 8, m8*). As a result, the kernel leverages both hardware register grouping and masked predication to coherently and efficiently combine operations on data of different widths within a single, integrated vector execution flow.

On the Milk-V SG2042 processor (RVV 0.7.1), where intrinsics are not supported by stable compilers (available only from RVV 1.0), a manually optimized assembly version of the kernel was developed. In practice, the assembly variant follows the same principles as the intrinsics-based implementation: dynamic

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_48

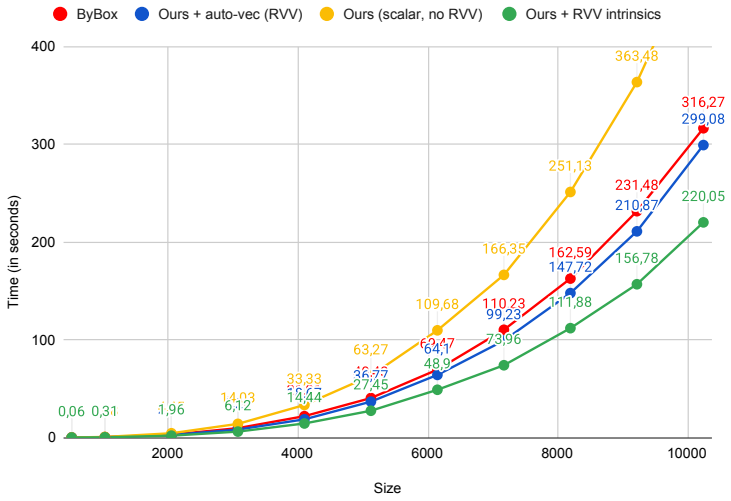

Fig. 2. Execution time (seconds) on Banana BPI-F3 using 8 threads.

vector length, *LMUL*=8 to maximize throughput for float64 operations, and masked predication for selective stores and accumulation. The key difference is that, at the assembly level, precise manual control of `vsetvli` is required (with frequent switching between `e8,m1` and `e64,m8`), along with explicit maintenance of the mask in register `v0` at critical points of the computation. This ensures pipeline consistency and provides full control over the vector configuration and predication semantics in RVV 0.7.1.

6.3 Experimental Results

On the Banana BPI-F3 platform, we achieved automatic vectorization; however, `clang` only enabled vectorization after applying an additional reduction in the innermost loop. Fig. 2 illustrates the cost of this weaker form of reduction, which provides only marginal performance improvements over the related *ByBox* method. In contrast, by employing explicit RVV intrinsics, we were able to better exploit the potential of the eight-core system and significantly reduce the overall execution time by vectorizing the outer loop through chunk-based processing, achieving an improvement of nearly 80 seconds versus almost 300 seconds with the auto-vectorization for the largest problem size considered ($N = 10240$).

In turn, on the SG2042 platform, where only RVV 0.7.1 is available, and compiler auto-vectorization is not supported, we implemented hand-written vector code in assembly. This implementation outperforms the *ByBox* approach for all studied problem sizes, ranging from 512 to 10240, as shown in Fig. 3(a). None of the evaluated implementations scale efficiently to 64 threads, since the McCaskill algorithm is predominantly memory-bound and the SG2042 architecture relies

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_48

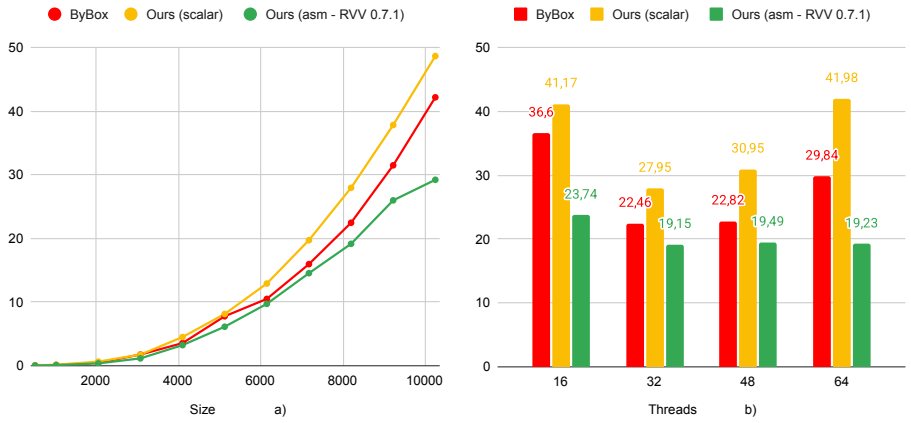

Fig. 3. Performance results on the SG2042 processor: (a) best execution time (in seconds) for different versions and problem sizes; (b) execution time (in seconds) for a problem size of 8192 using different numbers of threads.

on a mesh-based interconnect. Consequently, the best performance is achieved with 32-48 threads. Nevertheless, the proposed RVV-based implementation exhibits smaller performance degradations at higher thread counts compared to both the *ByBox* and scalar implementations, as illustrated in Fig. 3(b).

7 Conclusions

In this paper, we present an efficient OpenMP implementation of the McCaskill algorithm for computing RNA folding probabilities. The proposed implementation supports both AVX and RVV instructions on two x86-64 and two RISC-V machines, respectively. Experimental results show that the achieved speedups significantly outperform the *ByBox* approach proposed by Zhao and Sahni. For RISC-V platforms, we demonstrate that improved vectorization is achievable despite limited auto-vectorization by employing manually vectorized implementations based on RVV intrinsics and, when necessary, hand-written assembly.

As future work, we plan to implement the maximum expected accuracy (MEA) for a given RNA sequence and exploit the potential of vector instructions more extensively. Furthermore, the proposed implementation can be ported to other multi-core platforms, such as GPUs and FPGAs, since loops containing non-problematic operations can be permuted and parallelized in two dimensions (loops over *row* and *col* variables). Additionally, we intend to investigate the energy efficiency of optimized NPDP codes on modern RISC-V and ARM platforms using the strategy proposed in this paper, as scientific computing workloads constitute a representative and demanding benchmark for evaluating the maturity and performance/energy efficiency of emerging processors.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_48

References

1. Banana Pi BPI-F3. https://wiki.banana-pi.org/Banana_Pi_BPI-F3 (2024)
2. Bielecki, W., Poliwoda, M.: Automatic parallel tiled code generation based on dependence approximation. In: Malyshkin, V. (ed.) Parallel Computing Technologies. pp. 260–275. Springer International Publishing, Cham (2021)
3. Bondhugula, U., et al.: A practical automatic polyhedral parallelizer and locality optimizer. SIGPLAN Not. **43**(6), 101–113 (Jun 2008). https://doi.org/10.1145/1379022.1375595
4. Brown, N., Jamieson, M.: Performance characterisation of the 64-core SG2042 RISC-V CPU for HPC. In: ISC High Performance 2024 Int. Workshops. vol. 15058, pp. 354–377. Lect. Notes Comp. Sci. (2024)
5. Fekete, M., Hofacker, I.L., Stadler, P.F.: Prediction of RNA base pairing probabilities on massively parallel computers. Journal of Computational Biology **7**(1–2), 171–182 (Feb 2000). https://doi.org/10.1089/10665270050081441
6. Frid, Y., Gusfield, D.: A simple, practical and complete o -time algorithm for rna folding using the four-russians speedup. Algorithms for Molecular Biology **5**(1) (Jan 2010). https://doi.org/10.1186/1748-7188-5-13
7. Gruzewski, M., Palkowski, M.: Cross-platform and polyhedral programming for Nussinov RNA folding. Future Generation Computer Systems **169**, 107786 (2025)
8. Intel Corporation: Intel oneAPI DPC++/C++ Compiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html (2024), accessed: 2026-01-15
9. Li, J., Ranka, S., Sahni, S.: Multicore and GPU algorithms for Nussinov RNA folding. BMC Bioinformatics **15**(8), S1 (2014)
10. Lorenz, R., Bernhart, S.H., Höner zu Siederdissen, C., Tafer, H., Flamm, C., Stadler, P.F., Hofacker, I.L.: Viennarna package 2.0. Algorithms for Molecular Biology **6**(1) (Nov 2011)
11. McCaskill, J.S.: The equilibrium partition function and base pair binding probabilities for RNA secondary structure. Biopolymers **29**(6-7), 1105–1119 (may 1990)
12. Mullapudi, R.T., Bondhugula, U.: Tiling for dynamic scheduling. In: Rajopadhye, S., Verdoolaege, S. (eds.) Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques. Vienna, Austria (Jan 2014)
13. Nussinov, R., et al.: Algorithms for loop matchings. SIAM Journal on Applied mathematics **35**(1), 68–82 (1978)
14. Olas, T., Wyrzykowski, R., Olas, M., Pałkowski, M., Grużewski, M.: Towards the efficient use of RISC-V architecture in scientific computation (2026), manuscript submitted to Journal of Computational Science, No. JOCSCI-D-25-02945
15. Pałkowski, M., Bielecki, W.: Parallel cache-efficient code for computing the McCaskill partition functions. In: Proceedings of the Federated Conference on Computer Science and Information Systems (FedCSIS). vol. 18, pp. 207–210. IEEE / ACSIS (2019). https://doi.org/10.15439/2019F8, aCSIS, Vol. 18
16. Wonnacott, D., Jin, T., Lake, A.: Automatic tiling of "mostly-tileable" loop nests. In: IMPACT 2015: 5th International Workshop on Polyhedral Compilation Techniques, At Amsterdam, The Netherlands (2015)
17. Zhao, C., Sahni, S.: Cache and energy efficient algorithms for Nussinov's RNA folding. BMC Bioinformatics **18**(S15) (dec 2017)
18. Zhao, C., Sahni, S.: Efficient computation of RNA partition functions using McCaskill's algorithm. In: Proceedings of the Federated Conference on Computer Science and Information Systems (FedCSIS). vol. 21, pp. 449–452. IEEE / ACSIS (2020)

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_48