Proceedings of the 20th Conference on Computer Science and Intelligence Systems (FedCSIS) pp. 753–758
DOI: 10.15439/2025F8280
ISSN 2300-5963 ACSIS, Vol. 43

NPDP programming for RISC multi-core processors

Mateusz Gruzewski, Marek Palkowski
West Pomeranian University of Technology in Szczecin
ul. Zolnierska 49, 71-210 Szczecin, Poland
e-mail: mpalkowski@zut.edu.pl

***Abstract*—In recent years, parallel architectures have become ubiquitous due to advancements in AI and cloud computing. However, parallel processing is not limited to x86 CISC CPUs and advanced graphics cards, GPUs; it also includes computations on ARM-based and RISC-V devices. In recent years, ARM processors have been adapted to incorporate an increasing number of cores. Today, mobile devices feature at least eight execution units, typically divided into energy-efficient and performance-oriented groups. RISC-based parallel processors are also integrated on development boards supported by the Linux kernel. In this article, we tested our NPDP Benchmark Suite for non-serial polyadic dynamic programming, primarily in the field of computer algorithms and bioinformatics, to evaluate the performance of the RISC processors under study, as well as code locality and cache efficiency. The benchmark consists of 10 kernels written in C++ and OpenMP. In the Android environment, we used the JAVA NDK (Native Development Kit) to port the application. For Apple machines, we used a port to OpenMP for parallelization. For the RISC-V native Linux environment, we applied the native Linux setup for efficient execution. Finally, we summarized the article and outlined future work.**

I. Introduction

RISC-based parallel processors, including ARM and RISC-V architectures, are becoming increasingly important in modern computing, particularly in mobile, embedded, and energy-efficient systems. Despite their growing adoption and multicore capabilities, their performance under complex, irregular workloads remains less explored compared to traditional x86-based platforms.

Nonserial Polyadic Dynamic Programming (NPDP) is an advanced class of dynamic programming techniques characterized by:

- Nonserial dependencies: subproblems do not follow a strict linear order;
- Polyadic recurrences: subproblems may depend on more than two others;
- Dynamic behavior: the dependency structure changes depending on input data or runtime conditions.

Such problems arise in domains like RNA folding, sequence alignment, and optimal polygon triangulation. Their nontrivial dependency patterns pose challenges for both parallel execution and efficient memory access.

Despite progress in compiler optimizations, limited attention has been paid to the performance of NPDP workloads on modern RISC-based architectures. In this work, we evaluate the execution-time performance of a 10-kernel NPDP benchmark suite compiled with OpenMP on several RISC platforms, including Apple ARM chips, Android smartphones, and a RISC-V development board. We apply three polyhedral compilers—Pluto, Traco, and Dapt—for automatic loop transformation and parallelization.

II. NPDP Benchmark Suite

The Non-serial Polyadic Dynamic Programming (NPDP) benchmark suite [1] consists of the following ten kernels:

1) Nussinov Algorithm - Predicts RNA secondary structures by identifying the optimal folding pattern.
2) Zuker Algorithm – An extension of RNA folding prediction that considers free energy minimization.
3) Smith–Waterman (SW) Algorithm – Performs local sequence alignment to identify similar regions between two nucleotide or protein sequences.
4) Needleman–Wunsch (NW) Algorithm – Conducts global sequence alignment for comparing entire sequences.
5) Smith–Waterman Algorithm for Three Sequences (SW3D) – Extends the original algorithm to align three sequences simultaneously.
6) Counting Algorithm – Used in RNA folding studies to count specific structural configurations.
7) McCaskill's Algorithm (Mcc) – Calculates the partition function and base-pairing probabilities in RNA secondary structures.
8) Maximum Expected Accuracy (MEA) Algorithm – Predicts RNA secondary structures based on maximizing expected accuracy.
9) Knuth's Algorithm – Determines the optimal binary search tree, minimizing the search time for a set of keys.
10) Optimal Polygon Triangulation – Divides a polygon into triangles to minimize the total edge length or another cost function.

III. Polyhedral optimization for NPDP loop nests

The polyhedral model represents loop nests as polyhedra with affine loop bounds and schedules. This model provides a foundation for advanced loop transformations and the analysis of data dependences. By harnessing the power of the polyhedral model, compilers can automatically optimize loops, enhance performance (especially in terms of locality with loop tiling), and exploit parallelism (particularly with loop skewing for NPDP codes) [2].

Nonserial Polyadic Dynamic Programming (NPDP) applications are generally easy to parallelize using a loop transformation technique known as skewing [3]. Skewing is a transformation applied to nested loops to restructure the iteration space,

ensuring that the dependent computations can be executed in parallel. It is particularly useful for NPDP problems where dependencies form a non-trivial pattern.

Consider a standard DP recurrence with dependencies on previous iterations:

$$DP(i,j) = f(DP(i-1,j), DP(i,j-1))$$

The corresponding loop structure might be:

```
for (i = 1; i <= N; i++) {
  for (j = 1; j <= N; j++) {
    DP[i][j] = f(DP[i-1][j], DP[i][j-1]);
  }
}
```

This loop cannot be directly parallelized because each iteration `(i, j)` depends on previous iterations. By applying the loop skewing transformation:

$$i' = i, \quad j' = i + j$$

we transform the loop structure into:

```
for(j' = 2; j' <= 2N; j'++) {
  lb = max(1, j'-N);
  ub = min(N, j'-1);
  for(i = lb; i <= ub; i++) {
    int j = j' - i;
    DP[i][j] = f(DP[i-1][j], DP[i][j-1]);
  }
}
```

Thus, NPDP problems, despite their complex dependencies, can be efficiently parallelized using loop skewing [3]. However, parallelization alone is not sufficient. Optimal cache utilization is also crucial, and scheduling blocks in NPDP remains challenging due to non-uniform dependencies [4], [5], [6].

NPDP problems, such as the Nussinov algorithm [7], can often be represented in the polyhedral model, enabling advanced loop transformations as performed by compilers like Pluto [8], Dapt [9], and Traco [5].

The polyhedral compilers Pluto [8], Dapt [9], and Traco [5] are based on affine transformation frameworks (ATF), space-time tiling, and tile correction, respectively. Pluto excels in generating well-balanced affine schedules, but for NPDP codes, the framework can address a set of affine equations to tile all loop nests [5]. ATF is unable to tile the Nussinov or SW kernels, and it cannot parallelize the tiled code of the Mcc kernel [1]. The Traco compiler generates 3D tiles using the transitive closure of the dependence graph of the union of loop dependences. For some tasks from the NPDP benchmark suite, it is not possible to compute the exact transitive closure of the dependency graph; in such cases, an over-approximation is applied [1]. Further transformations for 3D-tiling of NPDP codes were implemented in the Dapt compiler by dividing the iteration space into timed parallel spaces. Dapt addresses irregularities in code obtained in Traco, providing a comprehensive solution to optimize and refine the generated code [10].

Loop tiling, also referred to as loop blocking or loop partitioning, is a widely used optimization to improve cache efficiency [11]. It restructures loops into smaller blocks (tiles), reducing cache misses and improving memory access locality. In polyhedral compilation, tiling is often applied in combination with other transformations, such as skewing, to facilitate both locality and parallelism.

Ultimately, the compilers generate cache-efficient, parallelized code using the same underlying approaches—primarily the polyhedral model and its supporting library, ISL (Integer Set Library)—for both dependency analysis and code generation. This unified approach ensures that the code generation aligns with the dependency structure and optimizations applied.

Along with the original NPDP benchmark codes, the repository https://github.com/markpal/NPDP_Bench also includes OpenMP code generated by polyhedral compilers, which we adapted for execution on RISC-based devices.

IV. Experimental study

The experimental evaluation was conducted across three distinct RISC-based hardware platforms: Apple ARM-based laptops (M2 Pro, M3, and M4 Pro), a high performance RISC-V development board (Banana Pi BPI F3 SpacemiT K1), and an Android-based mobile device (Samsung Galaxy A55). All experiments focused on executing NPDP kernels compiled with OpenMP parallelisation support [12].

To ensure consistent and platform-specific testing, two separate benchmarking projects were developed. The first project, designed for macOS and the RISC-V platform, was responsible for executing the benchmarks and recording performance results to structured output files. The second project was implemented in Android Studio and dedicated to the Android operating system. It included a complete test execution scenario, with background task management and result storage integrated into the application logic.

Across all platforms, the NPDP computational kernels remained identical. Only the system-specific integration layer differed, adapting the same computational logic to the requirements of each operating system.

A. ARM-based Apple Platforms

The first experimental setup involved Apple machines equipped with ARM-based processors: M2 Pro, M3, and M4 Pro. These platforms feature heterogeneous multi-core architectures combining high-performance and energy-efficient cores. All experiments were conducted natively on macOS. Specifically, the M2 Pro system ran macOS Sequoia 15.1.1, the M3 system used macOS Sequoia 15.2, and the M4 Pro machine operated on macOS Sequoia 15.3. Since the default Apple Clang compiler does not provide support for OpenMP, the LLVM toolchain was installed via Homebrew, along with the necessary runtime library (libomp). The benchmarks were compiled using the following command:

```
/opt/homebrew/opt/llvm/bin/clang++
  -fopenmp \
  -I/opt/homebrew/opt/llvm/include \
  -L/opt/homebrew/opt/llvm/lib \
  -o test test.cpp
```

This configuration allowed for the successful compilation and execution of OpenMP-parallelized code on macOS. Tests were conducted in terminal shells with controlled background activity to minimize external interference. While we initially considered evaluating performance on iPhones, it was not possible to execute OpenMP code due to iOS restrictions. The operating system lacks support for dynamic linking against the OpenMP runtime, and instead promotes the use of proprietary concurrency models such as Grand Central Dispatch [13].

B. RISC-V Platform

The second experimental platform was based on the RISC-V architecture. We employed the Banana Pi BPI-F3 board [14], featuring an eight-core SpacemiT K1 [15], [16] processor with 16 GB LPDDR4 RAM and 128 GB of eMMC storage. The system ran Bianbu Linux 1.0rc1 (codename mantic) [17], a lightweight RISC-V–oriented distribution based on Ubuntu 23.10, designed for embedded and development use cases. The board was configured in a headless mode and accessed remotely via SSH over Ethernet.

Due to limited support for the RISC-V Vector Extension (RVV) in available prebuilt toolchains, we compiled a custom cross-compilation toolchain targeting rv64gcv on an external x86-64 machine. The NPDP benchmark suite was compiled with OpenMP and vectorization enabled, using the following command:

```
riscv64-unknown-linux-gnu-g++ \
  -march=rv64gcv -mabi=lp64d -O3 \
  -fopenmp -ftree-vectorize \
  -fopt-info-vec-optimized \
  -o test test.cpp
```

The compiled binaries were transferred to the Banana Pi board and executed under Bianbu Linux. This setup enabled the assessment of OpenMP-parallelized NPDP kernels on a native RISC-V platform with partial RVV support and a modern, minimal Linux runtime environment.

C. ARM-based Mobile Platform

The final test environment was a Samsung Galaxy A55 smartphone running Android 13. The device is powered by an ARM-based octa-core processor, comprising four Cortex-A78 cores (2.7 GHz) [18] and four Cortex-A55 cores (2.0 GHz) [19]. The NPDP benchmark suite was ported using the Android Native Development Kit (NDK) [20], allowing native C++ code to be executed within the Android runtime environment. The OpenMP-enabled kernels were compiled into shared libraries and invoked via a Java Native Interface (JNI) wrapper.

To minimize interference from the graphical interface and system events, the benchmarks were executed in the background using a dedicated background thread. This ensured isolated execution, independent of the UI thread. Results were written to files in the internal storage and subsequently retrieved for further analysis. This setup enabled reliable measurement of kernel performance under realistic conditions on mobile hardware, although variability due to thermal management and background activity could not be entirely eliminated. All measurements were repeated multiple times to ensure consistency.

D. Execution Time Constraints and Omitted Results

In all presented Tables 1-5, certain entries are marked with the symbol "–". These indicate cases in which a particular kernel execution was terminated or skipped due to exceeding predefined time limits. To maintain practical runtime and result comparability, thresholds were set at 20 minutes for input size 2200, and 40 minutes for larger sizes (5000 and 10000). If a kernel exceeded these limits, execution was interrupted and the result omitted from the final report.

This limitation was observed across all platforms and test configurations. Algorithms such as *Knuth*, *McCaskill*, *Triangulation*, and *Zuker* frequently exhibited prohibitive execution times, especially in the original and Traco-generated code variants. Additionally, computationally complex kernels such as *SW3D* and *MEA* consistently demonstrated exceedingly long runtimes and were therefore excluded from measurement.

As a result of preliminary testing, the input sizes used for RISC-V and Android platforms were deliberately limited to 500, 1000, and 2200. This decision was made to ensure feasibility on resource-constrained devices. Moreover, certain algorithms such as *Knuth* and *Zuker* were excluded entirely from tests on these two platforms due to previously observed stability and performance issues during execution, including excessive runtimes or failures. These exclusions were applied consistently across both platforms as a conservative measure to preserve experimental integrity.

The exclusion of the Knuth and Zuker algorithms was driven by consistent and reproducible performance issues. The Zuker kernel, which involves complex free-energy minimization and deeply nested loop structures, incurred high computational overhead. On several platforms, particularly those using Traco-generated variants, execution times exceeded predefined limits for larger input sizes. The Knuth algorithm, in turn, triggered instability on resource-constrained RISC-V systems, including segmentation faults and abnormal termination, likely due to its memory access patterns and recursive structure. These issues rendered reliable benchmarking infeasible and led to the omission of these kernels from the reported results.

It is important to note that all benchmarked algorithms were based on identical NPDP kernel implementations. The only platform-specific differences resided in the build system, integration method, and runtime environment.

TABLE I: Execution times of algorithms on M2 Pro (in seconds)

	Org			Traco			Pluto			Dapt		
Algorithm	**2200**	**5000**	**10000**	**2200**	**5000**	**10000**	**2200**	**5000**	**10000**	**2200**	**5000**	**10000**
Nussinov	1.10	35.61	362.59	0.53	4.72	36.28	0.24	5.83	56.14	0.31	3.24	23.89
Counting	4.38	85.47	1010.36	0.67	10.41	135.36	0.94	10.34	97.47	0.65	11.72	135.67
Knuth	1.65	45.62	-	2.50	41.46	-	0.33	8.50	-	0.26	7.64	-
Mcc	7.96	195.77	-	1.56	31.55	-	3.40	162.98	-	1.21	43.64	-
Triang	2.74	99.51	-	0.93	14.87	-	0.57	24.22	-	0.69	23.97	-
Zuker	392.47	-	-	43.91	-	-	15.17	-	-	16.47	-	-
NW	5.81	249.22	2299.86	0.96	62.34	500.14	2.04	71.42	569.67	1.09	69.79	576.01
SW	6.26	267.26	-	1.20	61.78	505.01	1.79	72.21	585.32	1.13	72.06	576.97

TABLE II: Execution times of algorithms on M3 (in seconds)

	Org			Traco			Pluto			Dapt		
Algorithm	**2200**	**5000**	**10000**	**2200**	**5000**	**10000**	**2200**	**5000**	**10000**	**2200**	**5000**	**10000**
Nussinov	1.13	34.40	336.34	0.60	6.41	52.73	0.29	6.43	67.48	0.58	5.54	40.64
Counting	4.22	57.26	739.27	1.12	14.10	198.04	1.21	12.08	135.17	0.97	13.75	197.44
Knuth	1.94	51.16	-	1.70	26.49	-	0.35	10.46	-	0.38	11.57	-
Mcc	1.14	8.48	-	0.25	1.63	-	0.64	2.76	-	0.29	1.91	-
Triang	3.37	178.10	-	0.92	25.25	-	0.72	40.42	-	1.02	50.82	-
Zuker	288.26	-	-	54.67	-	-	25.71	-	-	32.68	-	-
NW	6.81	351.05	2483.51	1.50	67.36	678.42	1.59	115.30	1192.05	2.06	96.08	857.60
SW	8.37	494.12	-	2.11	83.54	864.38	1.55	104.24	1087.03	1.87	79.62	830.81

TABLE III: Execution times of algorithms on M4 Pro (in seconds)

	Org			Traco			Pluto			Dapt		
Algorithm	**2200**	**5000**	**10000**	**2200**	**5000**	**10000**	**2200**	**5000**	**10000**	**2200**	**5000**	**10000**
Nussinov	0.92	20.32	244.97	0.49	3.28	21.11	0.20	2.75	28.44	0.29	2.56	17.85
Counting	4.54	61.70	718.30	0.62	7.09	84.69	1.07	12.08	112.15	0.54	6.38	80.93
Knuth	1.61	56.41	-	1.59	25.15	-	0.21	6.98	-	0.20	6.72	-
Mcc	7.63	360.85	-	1.21	56.16	-	4.57	276.38	-	0.95	52.85	-
Triang	4.74	126.24	-	0.48	14.32	-	0.93	17.60	-	0.64	15.90	-
Zuker	238.54	-	-	23.16	-	-	12.16	-	-	11.57	-	-
NW	11.06	606.88	-	1.12	87.72	490.37	2.04	98.99	516.79	1.21	131.91	768.41
SW	12.89	599.46	-	1.19	182.67	1195.24	2.06	97.16	594.67	1.31	176.46	981.69

TABLE IV: Execution times of algorithms on Risc-V (in seconds)

	Org			Traco			Pluto			Dapt		
Algorithm	**500**	**1000**	**2200**	**500**	**1000**	**2200**	**500**	**1000**	**2200**	**500**	**1000**	**2200**
Nussinov	0.28	5.26	98.36	0.19	1.34	9.35	0.07	0.74	20.63	0.11	0.74	8.89
Counting	0.64	4.77	68.33	0.15	0.70	11.9	0.30	1.44	11.35	0.14	0.72	10.89
Knuth	-	-	-	-	-	-	-	-	-	-	-	-
Mcc	9.42	17.52	272.59	1.20	3.33	51.96	1.69	8.54	100.66	1.78	4.04	100.99
Triang	5.12	42.38	490.66	1.12	7.71	71.36	1.17	8.65	105.52	1.02	7.60	98.06
Zuker	-	-	-	-	-	-	-	-	-	-	-	-
NW	1.99	26.77	382.56	0.37	5.47	106.47	0.91	9.58	159.87	0.37	6.30	131.24
SW	1.98	26.37	381.60	0.39	5.30	104.66	0.40	6.52	124.30	0.38	5.97	124.12

E. Result Discussion

Tables I–V present execution times for all kernels across four code variants (Org, Traco, Pluto, Dapt) and five hardware platforms. Overall, the results confirm that polyhedral compilation significantly improves performance, and that Dapt frequently produces the fastest code across most tested configurations.

On Apple ARM platforms (M2–M4), the Dapt-generated code consistently outperforms both the original and other compiler variants. For example, on the M4 Pro (Table III), Dapt executes the Nussinov kernel in 17.85s for input size 10000, while the original version requires 244.97s. This trend is visible across most kernels and input sizes. However, it is worth noting that Apple systems are based on a heterogeneous architecture not explicitly designed for HPC-like parallel workloads. Their strong performance stems primarily from advanced chip design, high memory bandwidth, and aggressive power management at the cost of being proprietary and expensive.

Although Dapt yields strong results overall, certain kernels like Counting show comparable or even better performance with Traco on selected platforms. This may be due to Traco's tile correction approach occasionally producing layouts with better cache fit for specific iteration spaces [10]. While Dapt generally provides more consistent tiling across irregular kernels, localized cache effects may favor Traco in select scenarios.

The RISC-V platform, represented by the low-cost Banana Pi BPI-F3 board, completed all tests but with significantly

TABLE V: Execution times of algorithms on Samsung Galaxy A55 (in seconds)

	Org			Traco			Pluto			Dapt		
Algorithm	**500**	**1000**	**2200**	**500**	**1000**	**2200**	**500**	**1000**	**2200**	**500**	**1000**	**2200**
Nussinov	0.14	1.17	13.11	0.15	2.18	17.16	0.19	0.87	4.13	0.20	0.83	7.27
Counting	0.38	2.83	36.07	0.22	1.76	10.15	0.33	2.83	20.45	0.16	1.18	8.63
Knuth	-	-	-	-	-	-	-	-	-	-	-	-
Mcc	0.42	3.59	72.35	0.31	2.28	22.37	0.28	2.37	29.69	0.23	1.50	16.76
Triang	1.05	8.04	127.95	0.80	5.80	43.78	0.89	3.74	37.60	0.82	3.42	35.75
Zuker	-	-	-	-	-	-	-	-	-	-	-	-
NW	0.88	7.16	111.91	0.49	2.75	27.49	0.67	2.99	30.64	0.82	2.91	28.98
SW	1.00	8.09	124.30	0.55	3.13	27.36	0.78	3.04	32.49	0.79	2.97	34.10

higher execution times. This reflects both early-stage toolchain maturity and the limited performance of the tested hardware. RISC-V remains promising due to its openness and Linux compatibility, and future work will explore more capable boards.

The Android platform (Samsung Galaxy A55) exhibits intermediate results. Despite using a modern ARM-based SoC, the performance is limited by the operating system, which does not prioritize background computation. For instance, thermal throttling and restricted OpenMP threading in the ART environment reduce the potential gains from polyhedral optimizations. Android also introduces unpredictable performance variability due to dynamic task scheduling and user-space constraints. As a result, OpenMP kernels run slower and less deterministically compared to Linux-based systems.

While the Dapt compiler provides robust performance across all platforms, the relative gains vary depending on the architecture. Moreover, some kernels (e.g., Knuth, Zuker, SW3D, MEA) were excluded due to excessive runtimes or instability, especially in Traco or original versions. This was consistent across all environments and input sizes.

In summary, the results highlight the benefits of polyhedral compilation techniques for NPDP workloads and expose the current disparity between platform classes. These results are in line with earlier work on multi-core systems, where OpenMP-based parallelization and loop restructuring also led to notable performance improvements in geospatial workloads [21]. Apple systems deliver the highest raw performance but are closed and costly. RISC-V is significantly slower on current boards but offers openness, modularity, and full Linux support — key advantages for long-term research. Android devices, while architecturally capable, suffer from system-level limitations that reduce usable compute throughput. These findings justify the need for continued testing on new RISC-V platforms and more refined kernel-level optimization.

V. Conclusion

When comparing research conducted on RISC processors, it is essential to consider factors such as cost, licensing model, and the intended application of the processing unit itself. In terms of raw performance, the fastest processors currently available are undoubtedly the high-end Apple M4 Pro units, although they also come at a significantly higher cost. However, Apple and Cortex processors are based on closed architectures, which limits their utility in development and research contexts. In contrast, RISC-V processors, due to their open-source architecture, offer far greater flexibility and potential for innovation.

Unfortunately, the studied Banana chip performance still lags behind ARM-based solutions. While Linux kernel support exists, substantial work remains to improve overall efficiency. Features such as vector processing are still in the developmental stage, and compilers require significant refinement to reach production-grade maturity. Nonetheless, the open-source nature of the platform and the dedicated efforts of volunteer contributors suggest a promising future—potentially more so than the long-standing but limited Android NDK.

In terms of performance at the same price point, it is currently easier to utilize older CISC CPUs or other types of RISC-based GPUs. While multi-board computers like Banana, Leeche, or other RISC-V developer boards are becoming more widespread, especially among emerging manufacturers from countries affected by U.S. sanctions, their performance capabilities are still maturing and not yet competitive with mainstream alternatives. However, the performance of RISC-V may improve in the future—the question is how rapidly this improvement will occur. We also plan to evaluate emerging high-performance RISC-V platforms, including 64-core development boards that are now becoming commercially available [22].

In future work, we will extend our research framework to include other polyhedral benchmark suites and a detailed analysis of the energy consumption on these devices.

References

[1] M. Palkowski and W. Bielecki, "NPDP benchmark suite for the evaluation of the effectiveness of automatic optimizing compilers," *Parallel Computing*, vol. 116, p. 103016, Jul. 2023. doi: 10.1016/j.parco.2023.103016. [Online]. Available: https://doi.org/10.1016/j.parco.2023.103016

[2] S. Verdoolaege, "Integer set library - manual," www.kotnet.org/~skimo/isl/manual.pdf, 2011, accessed on: 2024-01-11.

[3] L. Liu, M. Wang, J. Jiang, R. Li, and G. Yang, "Efficient nonserial polyadic dynamic programming on the cell processor." in *IPDPS Workshops*. Anchorage, Alaska: IEEE, 2011, pp. 460–471.

[4] R. T. Mullapudi and U. Bondhugula, "Tiling for dynamic scheduling," in *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*, S. Rajopadhye and S. Verdoolaege, Eds., Vienna, Austria, Jan. 2014.

[5] M. Palkowski and W. Bielecki, "Parallel tiled Nussinov RNA folding loop nest generated using both dependence graph transitive closure and loop skewing," *BMC Bioinformatics*, vol. 18, no. 1, p. 290, 2017. doi: 10.1186/s12859-017-1707-8

[6] V. K. Tchendji, F. I. K. Youmbi, C. T. Djamegni, and J. L. Zeutouo, "A parallel tiled and sparsified Four-Russians algorithm for Nussinov's RNA folding," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, pp. 1–12, 2022. doi: 10.1109/tcbb.2022.3216826
[7] R. Nussinov *et al.*, "Algorithms for loop matchings," *SIAM Journal on Applied mathematics*, vol. 35, no. 1, pp. 68–82, 1978.
[8] U. Bondhugula *et al.*, "A practical automatic polyhedral parallelizer and locality optimizer," *SIGPLAN Not.*, vol. 43, no. 6, pp. 101–113, Jun. 2008. doi: 10.1145/1379022.1375595
[9] W. Bielecki and M. Poliwoda, "Automatic parallel tiled code generation based on dependence approximation," in *Parallel Computing Technologies*, V. Malyshkin, Ed. Cham: Springer International Publishing, 2021, pp. 260–275.
[10] M. Palkowski and M. Gruzewski, "Time and energy benefits of using automatic optimization compilers for NPDP tasks," *Electronics*, vol. 12, no. 17, p. 3579, Aug. 2023. doi: 10.3390/electronics12173579. [Online]. Available: http://dx.doi.org/10.3390/electronics12173579
[11] J. Xue, *Loop Tiling for Parallelism*. Norwell, MA, USA: Kluwer Academic Publishers, 2000. ISBN 0-7923-7933-0
[12] OpenMP Architecture Review Board, "OpenMP application program interface version 5.2," https://www.openmp.org/specifications, 2021, accessed on: 2023-10-22.
[13] Apple Inc., "Grand central dispatch (gcd)," https://developer.apple.com/documentation/dispatch, 2009, accessed: 2025-05-05.
[14] Banana Pi Team, "Banana pi bpi-f3 documentation," https://docs.banana-pi.org/en/BPI-F3/BananaPi_BPI-F3, 2025, accessed: 2025-05-05.
[15] SpacemiT, "Spacemit key stone k1 – octa-core 64-bit risc-v ai cpu," https://www.spacemit.com/en/key-stone-k1/, 2024, accessed: 2025-05-05.
[16] Banana Pi Team, "Spacemit k1 8-core risc-v chip brief," https://docs.banana-pi.org/en/BPI-F3/SpacemiT_K1, 2024, accessed: 2025-05-05.
[17] SpacemiT, "Bianbu linux," https://bianbu.spacemit.com/en/, 2024, accessed: 2025-05-05.
[18] Arm Ltd., "Arm cortex-a78 processor," https://developer.arm.com/Processors/Cortex-A78, 2020, accessed: 2025-05-05.
[19] ——, "Arm cortex-a55 processor," https://developer.arm.com/Processors/Cortex-A55, 2017, accessed: 2025-05-05.
[20] Google LLC, "Android ndk – native development kit," https://developer.android.com/ndk, 2024, accessed: 2025-05-05.
[21] B. Bylina, J. Potiopa, M. Klisowski, and J. Bylina, "The impact of vectorization and parallelization of the slope algorithm on performance and energy efficiency on multi-core architecture," in *Proceedings of the 16th Conference on Computer Science and Intelligence Systems (FedCSIS)*, ser. Annals of Computer Science and Information Systems, vol. 25. PTI, 2021. doi: 10.15439/2021F68 pp. 283–290.
[22] Sophgo, "Sg2042 risc-v 64-core soc," 2025, accessed on: 2024-07-13. [Online]. Available: https://en.sophgo.com/sophon-u/product/introduce/sg2042.html