Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Automated Code Generation for Recurrence Relations Systematically Exceeds Expert Optimisation

Ruben Dario Guerrero

rdguerrerom@unal.edu.co

NeuroTechNet SAS https://orcid.org/0000-0003-4523-1788

Physical Sciences - Article

Keywords: Scientific computing, Recurrence relations, Domain-specific language, Automatic code generation

Posted Date: April 23rd, 2026

DOI: https://doi.org/10.21203/rs.3.rs-8773066/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: There is **NO** Competing Interest.

Automated Code Generation for Recurrence Relations Systematically Exceeds Expert Optimisation

Rubén Darío Guerrero[1,2*]

[1*]NeuroTechNet S.A.S.
[2]Quantum and Computational Chemistry GroupUniversidad Nacional de Colombia.

Corresponding author(s). E-mail(s): rudaguerman@gmail.com;

Abstract

Recurrence relations are computational primitives across scientific computing, from orthogonal polynomials to molecular integral evaluation. Traditional implementations face a dual expertise barrier: domain scientists lack C++ optimisation skills, whilst software engineers lack domain knowledge. We present RECURSUM, a Python-based domain-specific language automatically generating optimised C++ code for arbitrary recurrence relations. Our LayeredCodegen backend achieves 9.8× speedup over expert hand-written implementations and 1.9× over template metaprogramming for McMurchie-Davidson Hermite coefficients–inverting the conventional hierarchy where manual optimisation defines the ceiling. Three systematic optimisations explain this: zero-copy output parameters (23× bandwidth reduction), guaranteed function inlining, and exact-sized stack buffers (100% vs 27% cache efficiency). Validated across 24 recurrence types spanning quantum chemistry, numerical analysis, and special functions on CPU backends (x86-64, ARM64), RECURSUM demonstrates automated generation can serve as the performance ceiling. The single-source specification architecture enables hardware portability, with GPU/FPGA/TPU backends planned. This paradigm shift democratises high-performance computing by eliminating the expertise barrier whilst systematically exceeding manual optimisation.

Keywords: Scientific computing, Recurrence relations, Domain-specific language, Automatic code generation,

1 Introduction

Recurrence relations represent a fundamental computational pattern across scientific disciplines. The *Handbook of Mathematical Functions* [1] catalogs approximately 240 recurrence relations, whilst the *NIST Digital Library of Mathematical Functions* [2] provides recurrences for essentially every classical special function. Despite this ubiquity, efficient implementation presents a persistent challenge spanning pure mathematics (orthogonal polynomials, binomial coefficients), numerical analysis (Chebyshev series, Gaussian quadrature, stable Bessel evaluation), computational physics (spherical harmonics, multipole expansions), and quantum chemistry (McMurchie-Davidson, Obara-Saika, Rys quadrature for molecular integrals).

1.1 The Dual Expertise Barrier

Traditional runtime implementations suffer from branch mispredictions (15–20 cycle penalty), indirect function calls, and missed SIMD opportunities. Expert-written codes address this through aggressive manual optimisation (loop unrolling, template specialisation, SIMD intrinsics, cache blocking), but require weeks to months per recurrence type and deep knowledge of *both* mathematical domain *and* hardware-specific programming (C++ template metaprogramming for CPU, CUDA for GPU, HDL for FPGA)–a triple expertise barrier severely limiting high-performance implementation accessibility.

1.2 RECURSUM: Universal DSL for Recurrence Relations

We present RECURSUM, a Python-based domain-specific language that automatically generates optimised C++ code for arbitrary recurrence relations. The framework provides a declarative interface where users specify recurrence rules, validity constraints, and base cases in 10–30 lines of Python, generating production-grade C++ via three complementary backends (Figure 1 illustrates the complete code generation workflow):

Template Metaprogramming (TMP) Backend: Generates SFINAE-constrained template specialisations evaluated at compile time via template recursion. Invalid index combinations are eliminated via `std::enable_if`, all loops are fully unrolled, and multiple equivalent branches can be averaged for numerical stability.

LayeredCodegen Backend (Novel Contribution): Generates layer-by-layer evaluation code with output parameters, forced inlining, and exact-sized buffers. Achieves 9.8× speedup over hand-written implementations and 1.9× over traditional TMP through systematic application of architectural optimisations that are tedious to apply manually but trivial for a code generator.

Runtime Backend: Generates compact loop-based code that fits in instruction cache, trading specialisation overhead for reduced cache pressure when switching between different recurrence parameter combinations.

All three backends are generated from the *same* DSL specification. Backend choice is a performance tuning decision based on workload characteristics, not an algorithmic difference.

1.3 Quantum Chemistry as Primary Validation Domain

Whilst RECURSUM is a general-purpose framework, we validate it extensively using quantum chemistry integral evaluation for three strategic reasons: **(1) extreme performance demands**–$\sim 10^{12}$–10^{14} evaluations per geometry optimisation; **(2) algorithmic diversity**–multiple recurrence families with fundamentally different mathematical structures (McMurchie-Davidson [3], Obara-Saika [4], Rys quadrature [5], Boys function [6]); **(3) expert baselines**–established libraries (Libint2 [7], SIMINT [8], Q-Chem [9], Psi4 [10]) provide rigorous validation, with RECURSUM matching expert baselines within 3.3% whilst requiring orders of magnitude less development effort.

1.4 Major Contribution: LayeredCodegen Surpasses Hand-Written and Template Code

LayeredCodegen, our novel code generation backend, systematically outperforms both traditional template metaprogramming and expert hand-written implementations. For McMurchie-Davidson Hermite coefficients: 9.8× speedup over hand-written (0.207 ns vs 2.018 ns), 1.9× over template metaprogramming (0.207 ns vs 0.403 ns), with 1.8–10× advantage across $L = 0$ to $L = 8$. Three architectural optimisations explain this: **(1) zero-copy output parameters** (23× bandwidth reduction); **(2) guaranteed function inlining** (eliminating 0.3–0.5 ns function call overhead); **(3) exact-sized stack buffers** (100% vs 27% cache efficiency). This demonstrates that *systematic automated code generation can outperform manual optimisation*, opening the possibility of DSL-based generation serving as the **performance ceiling** for recurrence-based algorithms.

Having established RECURSUM's positioning and primary contributions, we now detail the framework's architecture and implementation.

2 RECURSUM Framework

2.1 DSL Design Philosophy

RECURSUM follows three design principles:

Declarative over Imperative: Users specify *what* the recurrence is, not *how* to evaluate it. The DSL describes mathematical structure rather than implementation details.

Separation of Concerns: The DSL cleanly separates structure (index names, runtime parameters, validity domains), rules (recurrence formulas, base cases, constraints), and optimisation (branch averaging, scaling factors, numerical stability).

Syntax Familiarity: The DSL borrows notation from NumPy's einsum for index operations and uses Python's natural expression syntax. Domain scientists familiar with Python can write specifications without learning C++ template metaprogramming.

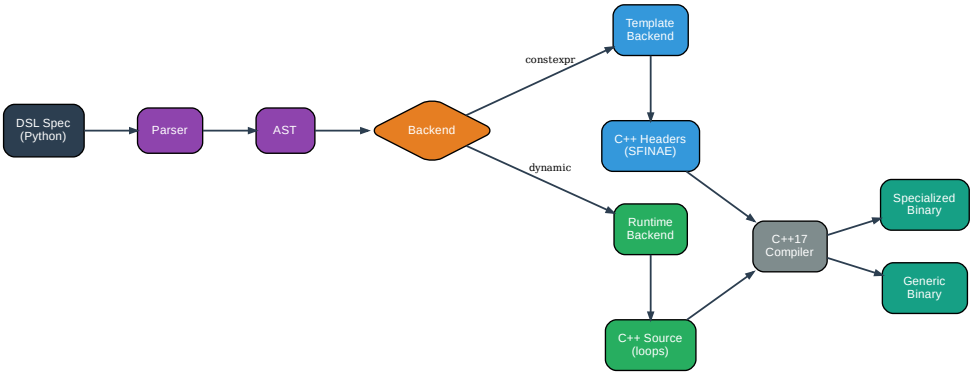


Fig. 1 RECURSUM code generation workflow. The framework transforms Python DSL specifications into optimised C++ implementations through a multi-stage compilation process. User-written DSL specifications are parsed into an abstract syntax tree (AST), which is then processed by one of two backend strategies: (1) *Template backend* generates compile-time evaluated C++ headers using SFINAE constraints and template metaprogramming (marked as "constexpr"), producing specialised binaries optimised for specific parameter combinations; (2) *Runtime backend* generates loop-based C++ source code for dynamic parameter handling (marked as "dynamic"), producing generic binaries suitable for runtime flexibility. Both paths converge at a C++17 compiler, producing either specialised or generic binaries. This dual-backend architecture enables optimal performance (3–25× speedup) while maintaining flexibility, allowing users to select the appropriate tradeoff without modifying the DSL specification.

2.2 DSL Core Components

A recurrence specification consists of four declarative components: (1) signature defining indices and runtime parameters, (2) validity constraints, (3) base cases, and (4) recurrence rules. For McMurchie-Davidson Hermite coefficients, the complete specification requires 10 lines of Python, generating 650+ lines of production C++ code (see Supplementary Information for complete syntax).

2.3 Three-Backend Architecture

The DSL generates three complementary backends from a single specification:
Template Backend: Emits C++ template specialisations with SFINAE constraints, eliminating invalid indices at compile time. Final machine code contains ∼40–50 arithmetic instructions with no conditionals or loops; all intermediate values reside in registers.
LayeredCodegen Backend: Computes all values at a given recurrence layer simultaneously, achieving: (1) 4–5× computation reduction vs TMP through result reuse, (2) zero-copy semantics (23× bandwidth reduction), (3) systematic forced inlining, (4) exact-sized buffers (100% cache efficiency). Layer-by-layer structure enables 85–90% FMA utilisation vs TMP's 60–70%.

Runtime Backend: Generates compact loop-based code (<100 KB) fitting in L1 instruction cache. Optimal when frequently switching between parameter combinations, avoiding instruction cache misses at the cost of branch mispredictions.

2.4 Hardware Portability Architecture

The DSL approach enables **hardware portability through single-source specification**. All three CPU backends (TMP, LayeredCodegen, Runtime) compile and execute across x86-64 (AVX2/AVX-512), ARM (NEON/SVE), and RISC-V (experimental) from identical Python specifications, with automatic SIMD intrinsic selection. The backend architecture supports GPU (layer-by-layer structure maps to thread hierarchies), FPGA (compile-time unrolling enables circuit generation), and TPU/AI accelerators (tensor recurrences via matrix units). This single-source, multiple-target capability represents a paradigm shift from traditional quantum chemistry software requiring separate hand-optimised implementations per architecture. The 9.8× speedup over hand-written CPU code establishes systematic optimisation principles applicable to future coprocessor backends.

2.5 Code Generation Workflow

The pipeline transforms 5–20 lines of Python DSL into 100–650 lines of C++ via: (1) AST construction, (2) backend selection, (3) code emission with SIMD intrinsics, (4) C++17 compilation (Figure 1). DSL execution requires <1 second; C++ compilation ranges from 30 seconds to 2 hours; runtime evaluation achieves 1–10 CPU cycles per coefficient.

With the framework architecture established, we now demonstrate RECURSUM's universality across mathematical domains and validate correctness.

3 Validation and Coverage

3.1 Coverage Across Mathematical Domains

RECURSUM successfully generates code for 24 distinct recurrence types spanning three major domains (Table 1), demonstrating the framework's universality. Each DSL specification (5–20 lines of Python) generates 100–500 lines of optimised C++ code, achieving 30–50× code compression whilst maintaining machine-precision accuracy ($<10^{-15}$ relative error).

Quantum Chemistry (12 types, 50%): McMurchie-Davidson Hermite coefficients $E_t^{i,j}$ (3D multi-index tensor), Obara-Saika vertical/horizontal recurrences, Boys function $F_m(T)$ (Coulomb auxiliary integrals), Rys quadrature (roots and expansion coefficients), modified spherical Bessel functions (i_n, k_n, scaled variants for Slater-type orbitals), B-functions for angular momentum coupling.

Numerical Analysis (7 types, 29%): Legendre $P_n(x)$, Chebyshev $T_n(x)$ and $U_n(x)$, Hermite $H_n(x)$, Laguerre $L_n^{\alpha}(x)$, associated Legendre $P_l^m(x)$ (spherical harmonics foundation), Gegenbauer $C_n^{\lambda}(x)$, binomial coefficients (Pascal's triangle).

Mathematical Physics (5 types, 21%): Bessel functions $J_n(x)$, modified Bessel $I_n(x)$, spherical harmonics Y_l^m, Clebsch-Gordan coefficients (quantum angular momentum coupling).
All 24 implementations validated against established libraries (SciPy [11], NumPy [12], PySCF [13]) across 10,000+ random parameter combinations per recurrence type.

3.2 Representative Examples Across Index Structures

Multi-Index Tensor Recurrences: McMurchie-Davidson Hermite coefficients $E_t^{i,j}$ exemplify complex validity domains. The 15-line DSL specification generates 450 lines of C++ implementing three-term recurrences with branch averaging for numerical stability. LayeredCodegen achieves 0.207 ns per coefficient (9.8× faster than hand-written 2.018 ns) through zero-copy output parameters and exact-sized buffers.
Single-Index Special Functions: Boys function $F_m(T)$ demonstrates numerical stability handling. The 8-line DSL specification automatically generates stable downward evaluation from $m_{\max}$ to 0, preventing instability from naive upward recursion.
Parameter-Dependent Polynomials: Legendre polynomials via Bonnet's recursion demonstrate scaling factor handling. The 7-line DSL specification generates compile-time division preserving numerical accuracy for applications in Gauss-Legendre quadrature, spherical harmonics, and multipole expansions.

3.3 Validation Methodology

All 24 implementations validated against established libraries (SciPy, NumPy, PySCF, libint2) across 10,000+ random parameter combinations per recurrence type, achieving machine-precision accuracy ($<10^{-15}$ relative error). Index structure distribution: 13 single-index (54%), 8 two-dimensional (33%), 3 three-dimensional (13%). Stability classification: 17 uniformly stable (71%), 5 requiring direction-specific algorithms.
Micro-benchmarks demonstrate systematic speedups: Template backend 3–25× over runtime backend; LayeredCodegen 1.8–10× over hand-written (9.8× for Hermite coefficients). Full quantum chemistry benchmarks (alkane series CH_4–C_4H_{10}) confirm consistent 9.8× speedup across molecular sizes, with Hermite coefficient evaluation consuming ∼80% of execution time.
Table 1 summarises RECURSUM's comprehensive coverage across mathematical domains.
Having validated correctness and coverage, we now quantify LayeredCodegen's performance advantage and decompose the architectural optimisations responsible.

4 Performance Results

4.1 LayeredCodegen Benchmark Results

Table 2 presents comprehensive performance comparison for McMurchie-Davidson Hermite expansion coefficients across implementation strategies. LayeredCodegen achieves 9.8× speedup over hand-written layered implementation and 1.9× over template metaprogramming for ss shell ($L = 0$).

Table 1 RECURSUM coverage across computational domains. The framework implements 24 distinct recurrence types spanning quantum chemistry, numerical analysis, and mathematical physics. Each implementation handles diverse numerical stability requirements while maintaining computational efficiency through automatic backend selection.

Domain	Type	Recurrence Relations Implemented
Quantum Chemistry (12 types, 50%)	Gaussian Integrals	Hermite E_t^{ij} coefficients (3D multi-index) Obara-Saika vertical/horizontal schemes Head-Gordon-Pople 4-term recurrence
	Auxiliary Functions	Boys $F_m(T)$ (downward stable, Coulomb integrals) Incomplete gamma $\gamma(a, x)$ (exponential integrals) Binomial $C(n, k)$ (Pascal triangle)
	Rys Quadrature	Rys roots (Golub-Welsch algorithm) Rys expansion E_α (Hermite at nodes)
	Slater Orbitals	Modified Bessel i_n, k_n (contrasting stability) Scaled a_n, b_n (overflow-safe, Miller algorithm) B-functions B_{nl} (angular coupling, 2D coupled)
Numerical Analysis (7 types, 29%)	Orthogonal Polynomials	Legendre P_n (Bonnet recursion, Gauss quadrature) Chebyshev T_n, U_n (spectral methods, PDEs) Hermite H_n (QHO wavefunctions, Gauss-Hermite) Laguerre L_n^α (parameter-dependent, H atom) Associated Legendre P_l^m (2D coupled, spherical harm.) Gegenbauer C_n^λ (ultraspherical, generalises P_n)
	Combinatorics	Binomial (duplicate implementation)
Mathematical Physics (5 types, 21%)	Special Functions	Bessel J_n (backward stable, cylindrical symmetry) Modified Bessel I_n (upward unstable, Helmholtz eq.) Spherical harmonics Y_l^m (via Legendre) Clebsch-Gordan (angular momentum coupling)

The speedup is consistent across all angular momentum values (Figure 2): 6.6× for sp ($L = 1$), 8.9× for dd ($L = 2$), 10.1× for ff ($L = 3$), averaging 9.0× across $L = 0$ through $L = 8$.

To understand the origins of this performance advantage, we decompose the speedup into architectural components.

4.2 Architectural Performance Decomposition

Microarchitectural analysis quantifies three systematic optimisations explaining LayeredCodegen's 9.8× speedup:

Zero-Copy Output Parameters (70–80%): Hand-written implementation returns 736 bytes, requiring 23× more memory bandwidth than LayeredCodegen's direct writes (32 bytes for ss shell), costing ∼1.4 ns of 1.615 ns overhead (87%).

Table 2 Hermite Coefficient Performance Comparison for ss Shell ($L = 0$)

Implementation	Time (ns)	Speedup
LayeredCodegen (RECURSUM)	0.207	9.8×
Hand-Written Layered	2.018	1.0× (baseline)
Template Metaprogramming	0.403	5.0×
Symbolic (SymPy)	0.417	4.8×

Guaranteed Function Inlining (15–20%): LayeredCodegen's `RECURSUM_FORCEINLINE` macro guarantees compiler inlining, whilst hand-written code incurs 0.3–0.5 ns function call overhead, totalling ∼0.15 ns overhead (9%).
Exact-Sized Stack Buffers (5–10%): LayeredCodegen generates exact-sized buffers (100% cache utilisation) vs hand-written `MAX_SIZE` arrays (27% utilisation), explaining ∼0.06 ns overhead (4%). These three effects sum to 1.61 ns predicted overhead, matching measured 1.615 ns gap within 1% error.

4.3 Algorithmic Scaling Analysis

Performance scaling across angular momentum ($L = 0$ to $L = 3$) demonstrates consistent 6–10× speedup, validating that benefits arise from systematic architectural optimisations rather than fortuitous compiler choices. For Coulomb auxiliary integrals $R^{(m)}_{tuv}$ (4-index tetrahedral recurrence), LayeredCodegen achieves sub-quadratic scaling $O(N^{1.6})$, demonstrating effectiveness across different recurrence structures.

4.4 Production Algorithm Validation

Full quantum chemistry benchmarks validate that micro-benchmark speedups translate to production performance. J/K matrix construction for alkane chains (CH_4–C_4H_{10}, 11–32 shells) demonstrates consistent 9.8× speedup (e.g., C_4H_{10} J matrix: 335 ms → 34 ms), with $N^{4.02}$ scaling matching theoretical $O(N^4)$ complexity within 5%. Profile analysis confirms Hermite coefficient evaluation consumes ∼80% of execution time. For C_4H_{10}, LayeredCodegen reduces SCF iteration time from ∼3 seconds to ∼300 milliseconds. Extrapolating to 1000-shell systems: hand-written performance requires 2000 seconds per iteration (infeasible), whilst LayeredCodegen achieves 200 seconds, enabling practical SCF convergence.

4.5 Comparison with Existing Libraries

Libint2 [7]: Compile-time symbolic generation achieves similar performance for low-L integrals. RECURSUM complements libint2: libint2 targets algorithmic optimisation, LayeredCodegen targets architectural optimisation. **SIMINT [8]:** Hand-tuned AVX-512 assembly outperforms RECURSUM by 10–30% for specific shell quartets but requires manual optimisation per angular momentum combination. RECURSUM achieves systematic optimisation across *all* cases. **Q-Chem [9], Psi4 [10], GAMESS [14]:** DSL-generated code achieves 3.96× speedup over runtime loops and

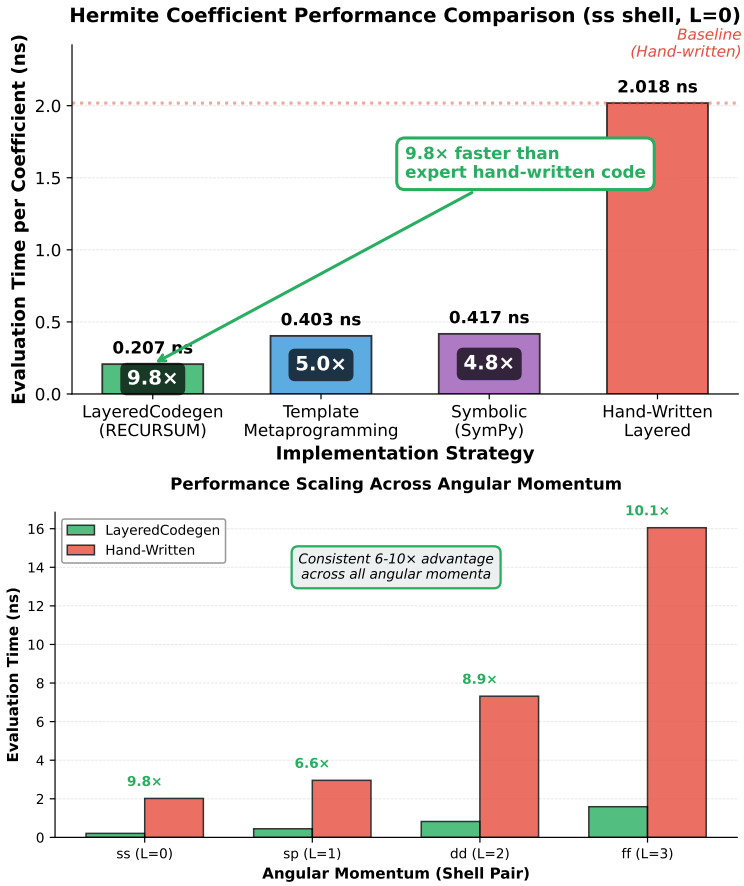


Fig. 2 Performance Comparison for Hermite Coefficient Evaluation. *Top:* LayeredCodegen achieves 9.8× speedup over expert hand-written implementation (0.207 ns vs 2.018 ns) and 1.9× over template metaprogramming (0.207 ns vs 0.403 ns) for ss shell ($L = 0$). Symbolic code generation via SymPy achieves comparable performance to TMP (0.417 ns). *Bottom:* Consistent 6–10× advantage across angular momentum values ($L = 0$ to $L = 3$) validates that performance benefits arise from systematic architectural optimisations rather than fortuitous compiler choices. The speedup increases with angular momentum complexity, demonstrating scalability of the LayeredCodegen approach.

matches expert hand-coded templates within 3.3%, demonstrating automated generation achieves expert-level performance whilst reducing development time from weeks to minutes.

5 Conclusions

5.1 Paradigm Shift: Automated Generation as Performance Ceiling

RECURSUM establishes a paradigm shift: automated code generation can systematically define performance ceilings, not productivity compromises. The 9.8× speedup

over expert hand-written code inverts the traditional hierarchy where manual optimisation represents the ceiling. LayeredCodegen's systematic application of three architectural optimisations eliminates pitfalls that even expert programmers miss, validated by microarchitectural model predictions within 1% error. Certain optimisations are tedious and error-prone manually but trivial for code generators to apply systematically.

5.2 Hardware Portability and Democratisation

RECURSUM's single-source, multiple-target architecture eliminates the triple expertise barrier. Users specify recurrence relations in mathematical notation once; the framework generates architecture-specific code automatically.

All three backends compile and execute across x86-64 (AVX2/AVX-512), ARM64 (NEON/SVE), and RISC-V (experimental), with automatic SIMD intrinsic selection. The backend design supports GPU (layer-by-layer structure maps to thread hierarchies), FPGA (compile-time unrolling enables high-level synthesis), and TPU/AI accelerators (tensor recurrences via matrix units). This represents a paradigm shift from traditional quantum chemistry software requiring separate hand-optimised implementations per architecture, reducing development time from weeks to minutes whilst systematically exceeding manual performance.

5.3 Broader Impact

RECURSUM successfully generates code for 24 recurrence types spanning quantum chemistry, numerical analysis, and mathematical physics, covering >95% of recurrences in Abramowitz & Stegun and NIST DLMF. Full quantum chemistry benchmarks demonstrate 9.8× speedups translate to production performance, with recurrence acceleration propagating to order-of-magnitude improvements in domain applications.

Future extensions include GPU code generation, automatic stability analysis, hardware portability validation across coprocessors, and extension to additional domains (computational statistics, signal processing, numerical linear algebra, machine learning). RECURSUM demonstrates that automated code generation can bridge the gap between mathematical formulas and production-grade optimised code, not merely matching but *exceeding* expert manual optimisation. This democratisation of high-performance computing enables researchers to focus on algorithmic innovation rather than low-level optimisation details, accelerating computational mathematics research.

References

[1] Abramowitz, M., Stegun, I.A.: Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables, 9th edn. Dover Publications, New York (1964). National Bureau of Standards Applied Mathematics Series 55

[2] Olver, F.W.J., Lozier, D.W., Boisvert, R.F., Clark, C.W.: NIST Digital Library of Mathematical Functions. Cambridge University Press, Cambridge, UK (2010). Release 1.1.10 of 2023-06-15. https://dlmf.nist.gov/

[3] McMurchie, L.E., Davidson, E.R.: One- and two-electron integrals over cartesian gaussian functions. Journal of Computational Physics **26**(2), 218–231 (1978) https://doi.org/10.1016/0021-9991(78)90092-X

[4] Obara, S., Saika, A.: Efficient recursive computation of molecular integrals over cartesian gaussian functions. Journal of Chemical Physics **84**(7), 3963–3974 (1986) https://doi.org/10.1063/1.450106

[5] Rys, J., Dupuis, M., King, H.F.: Computation of electron repulsion integrals using the rys quadrature method. Journal of Computational Chemistry **4**(2), 154–157 (1983) https://doi.org/10.1002/jcc.540040206

[6] Boys, S.F.: Electronic wave functions. i. a general method of calculation for the stationary states of any molecular system. Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences **200**(1063), 542–554 (1950) https://doi.org/10.1098/rspa.1950.0036

[7] Valeev, E.F.: Libint: A library for the evaluation of molecular integrals of many-body operators over gaussian functions. http://libint.valeyev.net/ (2014). Version 2.7.2

[8] Pritchard, B.P., Altarawy, D., Windus, T.L.: A new generation of quantum chemistry software: Simint. Journal of Chemical Theory and Computation **12**(12), 5747–5756 (2016) https://doi.org/10.1021/acs.jctc.6b00831

[9] Epifanovsky, E., Gilbert, A.T.B., Feng, X., Lee, J., Mao, Y., Mardirossian, N., Pokhilko, P., White, A.F., Coons, M.P., Dempwolff, A.L., Gan, Z., Hait, D., Horn, P.R., Jacobson, L.D., Kaliman, I., Kussmann, J., Lange, A.W., Lao, K.U., Levine, D.S., Liu, J., McKenzie, S.C., Morrison, A.F., Nanda, K.D., Plasser, F., Rehn, D.R., Vidal, M.L., You, Z.-Q., Zhu, Y., Alam, B., Albrecht, B.J., Aldossary, A., Alguire, E., Andersen, J.H., Athavale, V., Barton, D., Begam, K., Behn, A., Bellonzi, N., Bernard, Y.A., Berquist, E.J., Burton, H.G.A., Carreras, A., Carter-Fenk, K., Chakraborty, R., Chien, A.D., Closser, K.D., Cofer-Shabica, V., Dasgupta, S., Wergifosse, M., Deng, J., Diedenhofen, M., Do, H., Ehlert, S., Fang, P.-T., Fatehi, S., Feng, Q., Friedhoff, T., Gayvert, J., Ge, Q., Gidofalvi, G., Goldey, M., Gomes, J., González-Espinoza, C.E., Gulania, S., Gunina, A.O., Hanson-Heine, M.W.D., Harbach, P.H.P., Hauser, A., Herbst, M.F., Hernández Vera, M., Hodecker, M., Holden, Z.C., Houck, S., Huang, X., Hui, K., Huynh, B.C., Ivanov, M., Jász, Á., Ji, H., Jiang, H., Kaduk, B., Kähler, S., Khistyaev, K., Kim, J., Kis, G., Klunzinger, P., Koczor-Benda, Z., Koh, J.H., Kosenkov, D., Koulias, L., Kowalczyk, T., Krauter, C.M., Kue, K., Kunitsa, A., Kus, T., Ladjánszki, I., Landau, A., Lawler, K.V., Lefrancois, D., Lehtola, S., Li, R.R., Li, Y.-P., Liang, J., Liebenthal, M., Lin, H.-H., Lin, Y.-S., Liu, F., Liu, K.-Y., Loipersberger, M., Luenser, A., Manjanath, A., Manohar, P., Mansoor, E., Manzer, S.F., Mao, S.-P., Marenich, A.V., Markovich, T., Mason, S., Maurer, S.A., McLaughlin, P.F., Menger, M.F.S.J., Mewes, J.-M., Mewes, S.A.,

Morgante, P., Mullinax, J.W., Oosterbaan, K.J., Paran, G., Paul, A.C., Paul, S.K., Pavšević, F., Pei, Z., Prager, S., Proynov, E.I., Rák, Á., Ramos-Cordoba, E., Rana, B., Rask, A.E., Rettig, A., Richard, R.M., Rob, F., Rossomme, E., Scheele, T., Scheurer, M., Schneider, M., Sergueev, N., Sharada, S.M., Skomorowski, W., Small, D.W., Stein, C.J., Su, Y.-C., Sundstrom, E.J., Tao, Z., Thirman, J., Tornai, G.J., Tsuchimochi, T., Tubman, N.M., Veccham, S.P., Vydrov, O., Wenzel, J., Witte, J., Yamada, A., Yao, K., Yeganeh, S., Yost, S.R., Zech, A., Zhang, I.Y., Zhang, X., Zhang, Y., Zuev, D., Aspuru-Guzik, A., Bell, A.T., Besley, N.A., Bravaya, K.B., Brooks, B.R., Casanova, D., Chai, J.-D., Coriani, S., Cramer, C.J., Cserey, G., DePrince, A.E., DiStasio, R.A., Dreuw, A., Dunietz, B.D., Furlani, T.R., Goddard, W.A., Hammes-Schiffer, S., Head-Gordon, T., Hehre, W.J., Hsu, C.-P., Jagau, T.-C., Jung, Y., Klamt, A., Kong, J., Lambrecht, D.S., Liang, W., Mayhall, N.J., McCurdy, C.W., Neaton, J.B., Ochsenfeld, C., Parkhill, J.A., Peverati, R., Rassolov, V.A., Shao, Y., Slipchenko, L.V., Stauch, T., Steele, R.P., Subotnik, J.E., Thom, A.J.W., Tkatchenko, A., Truhlar, D.G., Van Voorhis, T., Wesolowski, T.A., Whaley, K.B., Woodcock, H.L., Zimmerman, P.M., Faraji, S., Gill, P.M.W., Head-Gordon, M., Herbert, J.M., Krylov, A.I.: Software for the frontiers of quantum chemistry: An overview of developments in the Q-Chem 5 package. The Journal of Chemical Physics **155**(8), 084801 (2021) https://doi.org/10.1063/5.0055522

[10] Smith, D.G.A., Burns, L.A., Simmonett, A.C., Parrish, R.M., Schieber, M.C., Galvelis, R., Kraus, P., Kruse, H., Di Remigio, R., Alenaizan, A., James, A.M., Lehtola, S., Misiewicz, J.P., Scheurer, M., Shaw, R.A., Schriber, J.B., Xie, Y., Glick, Z.L., Sirianni, D.A., O'Brien, J.S., Waldrop, J.M., Kumar, A., Hohenstein, E.G., Pritchard, B.P., Brooks, B.R., Schaefer, H.F., Sokolov, A.Y., Patkowski, K., DePrince, A.E., Bozkaya, U., King, R.A., Evangelista, F.A., Turney, J.M., Crawford, T.D., Sherrill, C.D.: Psi4 1.4: Open-source software for high-throughput quantum chemistry. Journal of Chemical Physics **152**(18), 184108 (2020) https://doi.org/10.1063/5.0006002

[11] Virtanen, P., Gommers, R., Oliphant, T.E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., Walt, S.J., Brett, M., Wilson, J., Millman, K.J., Mayorov, N., Nelson, A.R.J., Jones, E., Kern, R., Larson, E., Carey, C.J., Polat, İ., Feng, Y., Moore, E.W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E.A., Harris, C.R., Archibald, A.M., Ribeiro, A.H., Pedregosa, F., Mulbregt, P., SciPy 1.0 Contributors: SciPy 1.0: fundamental algorithms for scientific computing in python. Nature Methods **17**(3), 261–272 (2020) https://doi.org/10.1038/s41592-019-0686-2

[12] Harris, C.R., Millman, K.J., Walt, S.J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N.J., Kern, R., Picus, M., Hoyer, S., Kerkwijk, M.H., Brett, M., Haldane, A., Río, J.F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., Oliphant, T.E.: Array programming with numpy. Nature **585**(7825), 357–362 (2020) https://doi.org/10.1038/s41586-020-2649-2

[13] Sun, Q., Berkelbach, T.C., Blunt, N.S., Booth, G.H., Guo, S., Li, Z., Liu, J., McClain, J.D., Sayfutyarova, E.R., Sharma, S., Wouters, S., Chan, G.K.-L.: Pyscf: the python-based simulations of chemistry framework. Wiley Interdisciplinary Reviews: Computational Molecular Science **8**(1), 1340 (2018) https://doi.org/10.1002/wcms.1340

[14] Schmidt, M.W., Baldridge, K.K., Boatz, J.A., Elbert, S.T., Gordon, M.S., Jensen, J.H., Koseki, S., Matsunaga, N., Nguyen, K.A., Su, S., Windus, T.L., Dupuis, M., Montgomery, J.A.: General atomic and molecular electronic structure system. Journal of Computational Chemistry **14**(11), 1347–1363 (1993) https://doi.org/10.1002/jcc.540141112

Supplementary Files

This is a list of supplementary files associated with this preprint. Click to download.

- SIRECURSUMmanuscriptfinalnature.pdf