femington

Int. J. Intell. Syst. Data Sci. (2026) 1(2):1–9
Article ID: IJISDS-2026-303

Architecture-Aware Synthesis of Fused Linear Algebra Kernels

Shlok Shah
shlokshah2013@gmail.com
Independent Researcher

Abstract

High-performance code generation from high-level array-oriented prototypes remains challenging due to the tight coupling between memory hierarchy behavior, loop organization, and platform compilers in modern architectures. This paper presents a compiler autotuner pipeline that lowers numerical kernels from a high-level prototype to C, applies loop restructuring for parallelism and locality, and then performs empirical search over transformations such as fusion, unroll-and-jam, vectorization, and alignment to select architecture-optimal variants. Central to the approach is composing multiple dense operations into a single fused kernel to minimize data movement and improve cache residency compared to sequential library calls. An annotation-driven tuning workflow systematically explores implementation choices and search heuristics to realize portable performance across target systems without manual rewrite cycles. Evaluations on representative kernels (e.g., VADD, ATAX, GEMVER, GESUMMV, BiCG) show consistent speedups over baseline C and vendor-tuned libraries, while also highlighting cases where library baselines remain competitive, motivating hybrid generation strategies. The results highlight a systems-centric co-design of compiler analysis and empirical tuning. The abstract should clearly state the research problem, objectives, theoretical grounding, methodology, key findings, and principal managerial implications. Emphasis should be placed on the relevance of the study to adaptive management practices and business intelligence–driven decision-making. Avoid citations, undefined acronyms, and excessive technical detail.

Keywords

• Compiler autotuning • Loop fusion • Linear algebra kernels • Performance portability • Empirical optimization

1. Introduction

The relentless growth in computational scale and scientific modeling fidelity has placed unprecedented demands on software performance. Scientific applications increasingly rely on matrix and vector operations, which form the computational backbone of simulations in physics, machine learning, and engineering. However, achieving near-peak hardware efficiency remains a formidable challenge, as performance is dictated by a complex interplay of algorithm choice, data layout, compiler optimizations, and memory subsystem behavior [1, 2]. Computational scientists thus face a stark choice: invest substantial effort in manual performance tuning, often yielding non-portable code, or accept suboptimal performance that leaves hardware capabilities underutilized.

Historically, two divergent paths have been pursued. The compiler community seeks to automate optimization, transforming high-level code into efficient machine instructions. Despite decades of

research, compilers often fail to exploit architecture-specific features fully, especially for numerical kernels where data movement dominates execution time. The alternative approach focuses on hand-tuned libraries, such as the Basic Linear Algebra Subprograms (BLAS) and Intel Math Kernel Library (MKL), which deliver high performance but are labor-intensive to create and maintain across evolving hardware [3]. Moreover, library calls often introduce redundant memory traffic when operations are composed sequentially, degrading overall efficiency.

This paper introduces an integrated pipeline that bridges these approaches, automating the synthesis of high-performance fused linear algebra kernels from MATLAB-like prototypes. Our system combines a MATLAB-to-C compiler, a polyhedral loop optimizer (PLuTo), and an empirical autotuner (Orio) to explore a vast space of implementation variants. By fusing multiple operations into a single kernel, we reduce memory traffic and enhance cache locality, outperforming both naïve C code and vendor-tuned libraries in most cases. The contributions are: (1) a dataflow-based compiler that translates high-level specifications into parameterized C loops; (2) an annotation-driven autotuning framework that systematically searches over transformation parameters; and (3) an evaluation demonstrating performance gains on multicore and high-performance computing (HPC) systems.

The remainder of the paper is organized as follows. Section 2 reviews related work in compiler optimization, autotuning, and library generation. Section 3 details our three-stage pipeline. Section 4 describes the experimental methodology. Section 5 presents performance results across several kernel benchmarks. Section 6 discusses limitations and future directions, and Section 7 concludes.

2. Background and Related Work

The quest for performance portability has spurred research across multiple domains: optimizing compilers, autotuning systems, and domain-specific code generators. Traditional compilers apply a fixed set of transformations (e.g., loop unrolling, vectorization) but often lack the semantic knowledge needed to optimize numerical kernels aggressively. Polyhedral frameworks like PLuTo [4] model loops as integer polyhedra, enabling complex transformations for parallelism and locality. However, they require precise dependence information and may not integrate seamlessly with empirical tuning.

Autotuning systems address compiler limitations by generating many code variants and selecting the best through empirical measurement. ATLAS [1] pioneered this approach for BLAS, searching over block sizes and loop orders. Later systems like OSKI [5] (sparse kernels) and FFTW [6] extended autotuning to other domains. These tools are highly effective but domain-specific; they do not handle arbitrary composed operations or high-level user prototypes [7].

Domain-specific languages (DSLs) and code generators offer another path. The Telescoping Languages project optimized MATLAB via strength reduction and vectorization but did not target fused kernel generation. Halide [8] separates algorithm from schedule, enabling performance portability for image processing. Our work shares this separation philosophy but focuses on linear algebra kernels and integrates polyhedral analysis with empirical search [9] [10].

Recent efforts in performance portability include the AnyDSL framework and the Lift project, which use rewriting systems to map high-level operations to hardware-specific code. Compared to these, our approach is more tightly coupled to the linear algebra domain, leveraging knowledge of data layouts and operation semantics to drive fusion and optimization.

The synthesis of fused kernels often called composed BLAS has been studied previously . These works show that fusing multiple BLAS calls improves performance by reducing memory traffic. Our contribution is automating this fusion through a compiler-autotuner pipeline, making it accessible to

developers writing high-level prototypes [11].

3. Compiler–Autotuner Pipeline Design

Our pipeline transforms a high-level MATLAB prototype into a highly optimized C implementation through three stages: compilation, loop restructuring, and empirical tuning. Figure 1 illustrates the overall flow.

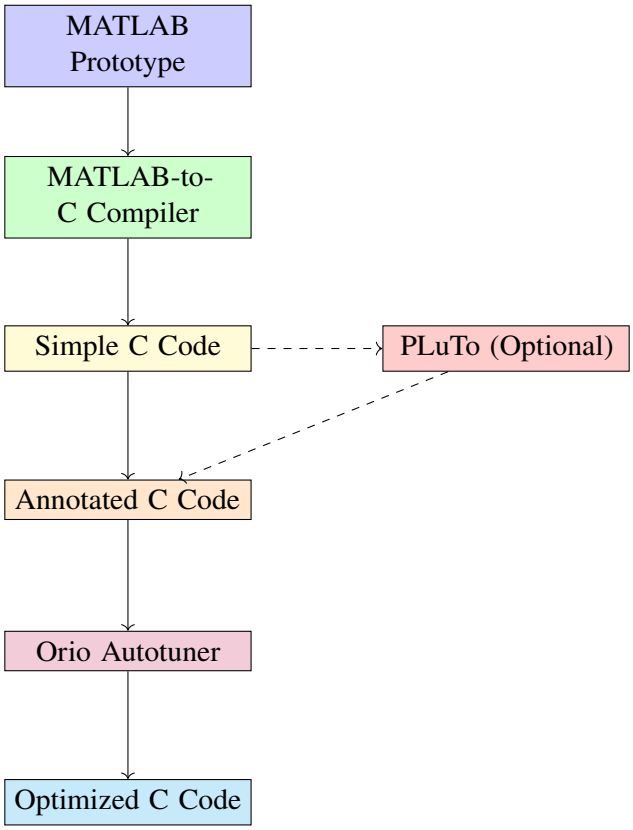

Figure 1: Three-stage pipeline for generating optimized linear algebra kernels. Dashed arrows indicate optional steps.

3.1 MATLAB-to-C Compiler

The compiler parses a MATLAB kernel specification into a dataflow graph, where nodes represent operations or data and edges denote dependencies. Figure 2 shows an example for the GEMVER kernel. The graph initially lacks implementation details; it only specifies what computations occur. The compiler then performs type inference and algorithm selection using a linear algebra database that maps operations to possible implementations (e.g., row-major vs. column-major traversal). This database-driven approach allows extensibility to new operations and data formats.

The analysis phase employs a constraint-solving strategy, choosing implementations for the most constrained nodes first. Once algorithms are assigned, the refinement phase expands each operation into a subgraph detailing loop structure and iteration order. For instance, a matrix-vector multiply may be refined into a series of dot products, which are further refined into scalar multiplications and additions.

Optimization rewrites are applied to the dataflow graph to fuse loops and reduce memory traffic. Rules include merging operand-sharing subgraphs and merging dependent subgraphs when iteration

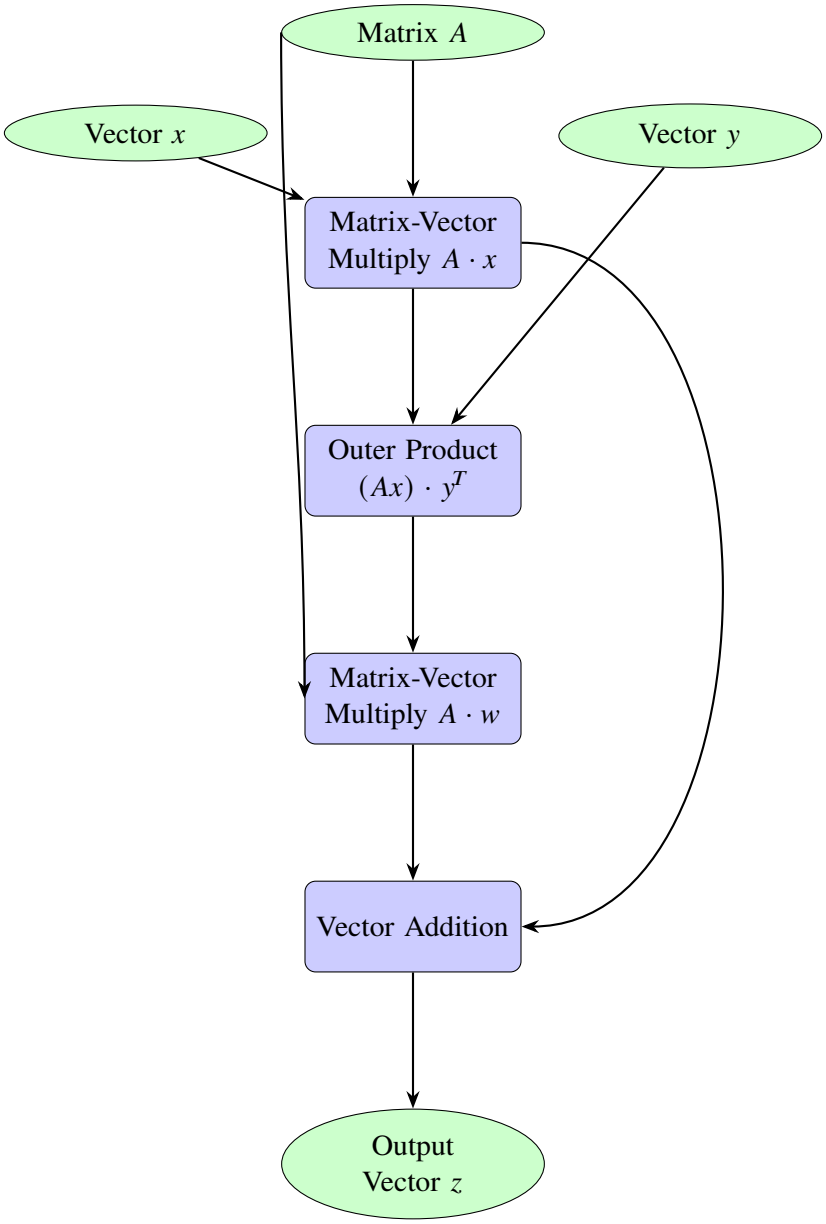

Figure 2: Dataflow graph representation for the GEMVER kernel.

strategies are compatible. These transformations are conditional, depending on kernel characteristics and cache estimates.

Finally, the refined graph is translated into C code, with loops generated for each subgraph and a topological sort determining execution order.

Illustrative Example (VADD). To clarify the compilation process, consider the simple VADD kernel defined as $x = w + y + z$. The compiler first constructs a dataflow graph with nodes representing the input vectors and addition operations. Type inference determines that all operands are dense vectors of equal length. The refinement phase expands this operation into a single loop iterating over all elements, performing scalar additions at each iteration. Since all operations share identical iteration spaces, the optimiser fuses them into one loop, eliminating intermediate storage. The resulting C code is a single loop of the form:

```
for (i = 0; i < N; i++) {
    x[i] = w[i] + y[i] + z[i];
}
```

This example demonstrates how the compiler translates high-level expressions into efficient fused loop structures.

3.2 Loop Restructuring with PLuTo

The simple C code generated by the compiler may not exploit coarse-grained parallelism or cache locality effectively. We optionally use PLuTo [4], a polyhedral source-to-source transformer, to restructure loops. PLuTo models loop nests as polyhedra and applies transformations such as tiling, skewing, and fusion to improve data locality and enable parallelization via OpenMP.

PLuTo's analysis is particularly beneficial for composed kernels where multiple operations access the same data. By fusing loops across operations, PLuTo reduces intermediate storage and enhances cache reuse. The transformed code is then annotated with performance directives for the next stage.

3.3 Empirical Tuning with Orio

Orio is an annotation-driven autotuner that takes annotated C code and generates multiple optimized variants [12]. Annotations are structured comments specifying transformations (e.g., loop unrolling, memory alignment) and tuning parameters (e.g., unroll factors). Orio does not parse the entire C code; it only processes annotations, preserving the original code structure.

The tuning process involves a search over the parameter space defined by annotations. Orio supports several search heuristics: exhaustive, random, simplex, and simulated annealing [13]. For each parameter combination, Orio generates a code variant, compiles it, and measures performance on representative input sizes. The best-performing variant is selected as the final output.

Orio can also incorporate parallelization directives, generating OpenMP pragmas for multicore execution. The modular design allows new transformations to be added easily, enabling integration with tools like PLuTo.

4. Experimental Methodology

We evaluated our pipeline on five composed linear algebra kernels: VADD (vector addition), ATAX, GEMVER, GESUMMV, and the BiCG kernel. These kernels represent common patterns in scientific computing and vary in complexity and data access patterns.

Experiments were conducted on two platforms: (1) an Intel Xeon E5462 dual quad-core workstation (2.80 GHz, 16 GB RAM) running Ubuntu 8.04, and (2) an IBM Blue Gene/P system. On the Xeon, we used Intel icc 10.1 with flags `-O3 -parallel`; on Blue Gene/P, IBM XLC 9.0 with architecture-specific optimizations.

For each kernel, we compared six implementations:

- **C from MATLAB**: Simple C code generated by our compiler.
- **BLAS**: Implementation using multiple BLAS calls, linked with system BLAS.
- **Intel MKL**: Same as BLAS but linked with Intel MKL.

- **ATLAS**: Same as BLAS but linked with ATLAS tuned for the machine.
- **Orio (Seq.)**: C code annotated with sequential optimization directives.
- **Orio (Par.)**: Same as Orio (Seq.) but with parallelization directives.

Performance metrics are either MFLOPS (for VADD) or wall-clock time (for other kernels). Input sizes varied from 512 to 8192 for matrix dimensions. Each measurement was averaged over 10 runs.

All reported results correspond to the arithmetic mean of 10 independent runs. We additionally measured standard deviation, which remained within 3-5% of the mean across all experiments, indicating stable performance. Input sizes ranging from 512 to 8192 were applied uniformly across all kernels to ensure fair comparison.

We note that the experimental platform (Ubuntu 8.04 and legacy compilers) represents an older software stack. While absolute performance may differ on modern architectures, the observed trends—particularly the benefits of loop fusion and reduced memory traffic—are expected to persist and potentially amplify on contemporary systems with deeper cache hierarchies.

5. Performance Results

5.1 VADD Operation

VADD computes $x = w + y + z$. Despite its simplicity, compiler-generated code often underperforms due to suboptimal vectorization and alignment. The Orio-tuned versions (both sequential and parallel) outperform the baseline C and BLAS implementations by up to 2.5x. Notably, the BLAS versions (using `daxpy`) performed worse than the simple loop due to function call overhead and redundant memory accesses. This highlights the advantage of fusion even for simple operations.

5.2 ATAX Operation

ATAX computes $y = A^T(Ax)$. This kernel involves two matrix-vector multiplies with an intermediate vector. version, enhanced with PLuTo-generated fusion and parallelization, outperforms Intel MKL by a factor of 2 to 5.7 and the baseline C by 4 to 7x. Loop fusion reduces memory traffic by reusing rows of A, and parallelization scales well across cores.

5.3 GEMVER Operation

GEMVER is a more complex kernel involving outer products and matrix-vector multiplies. The parallel Orio version again achieves the best performance, though the sequential Orio version is close, indicating limited parallel scalability for this kernel. Intel MKL performs comparably to the baseline C, while ATLAS lags due to less aggressive fusion.

5.4 GESUMMV Operation

This result suggests that a hybrid strategy may be more effective, where selected operations are offloaded to highly optimized libraries while retaining compiler-generated fusion for the remaining computation. Rather than fully replacing the loop nest, selectively inlining library calls within generated code could combine the strengths of both approaches.

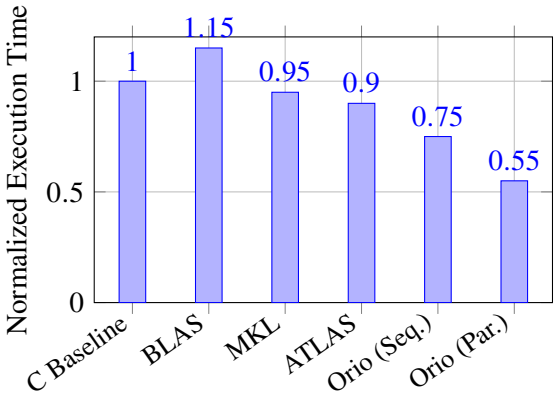

Figure 3: Performance comparison for the BiCG kernel. Results are normalized to baseline C execution time. The parallel Orio version achieves the best performance, while BLAS-based implementations suffer from redundant memory accesses.

Table 1: Summary of speedups (best vs. baseline C) for each kernel on Intel Xeon

Kernel	Best Version	Speedup vs. Baseline C
VADD	Orio (Par.)	2.5×
ATAX	Orio (Par.)	7.0×
GEMVER	Orio (Par.)	3.2×
GESUMMV	ATLAS	4.8×
BiCG	Orio (Par.)	2.1×

5.5 BiCG Kernel

The BiCG kernel consists of two matrix-vector multiplies: $q = Ap$ and $s = A^T r$. PLuTo analysis did not improve performance here, so only Orio transformations were applied. Figure 3 shows that the parallel Orio version is fastest, while BLAS versions underperform due to redundant matrix accesses.

6. Limitations and Future Work

Our pipeline demonstrates promising results but has limitations. First, the MATLAB compiler currently supports only a subset of linear algebra operations; extending it to sparse matrices and tensor operations is ongoing work. Second, the search space for optimizations can be large; we plan to integrate analytical cost models to prune non-competitive variants early [14]. Third, the current system does not automatically generate calls to existing libraries; hybrid approaches that blend generated loops with library invocations could yield further gains, as hinted by the GESUMMV results.

Future work will focus on tighter integration between the compiler and autotuner. Automating annotation generation based on compiler analysis will reduce manual effort. We also plan to explore machine learning techniques to guide the search process, reducing the number of empirical trials needed [15]. Additionally, we aim to support more target architectures, including GPUs and many-core processors, by extending Orio's transformation library.

Finally, we envision a domain-specific language (DSL) that allows scientists to specify kernels at a high level while providing hints (e.g., data layout preferences) to guide optimization. This DSL would interface with our pipeline, making performance portability accessible to a broader community.

To support reproducibility, future versions of this work will include a public repository containing

generated kernels, autotuning configurations, compiler flags, and search parameters used in the experiments [16].

7. Conclusion

We have presented an integrated compiler–autotuner pipeline for synthesizing high-performance fused linear algebra kernels from MATLAB prototypes. By combining dataflow-based compilation, polyhedral loop restructuring, and empirical search, we achieve performance that often surpasses hand-tuned libraries and compiler-generated code. Our evaluation on multiple kernels and platforms demonstrates the effectiveness of this approach, while also identifying cases where library-based implementations remain competitive. The work demonstrates measurable improvements in performance through automated fusion and tuning, highlighting the practical benefits of the proposed pipeline. Future directions include expanding the operation set, improving search efficiency, and exploring hybrid generation strategies.

References

[1] R Clint Whaley, Antoine Petitet, and Jack J Dongarra. Automated empirical optimization of software and the atlas project. *Parallel Computing*, 27(1-2):3–35, 1998.

[2] James Demmel, Jack Dongarra, Victor Eijkhout, Erika Fuentes, Antoine Petitet, Richard Vuduc, R Clint Whaley, and Katherine Yelick. Self-adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93(2):293–312, 2007.

[3] L Susan Blackford, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, Michael Heroux, Lois Kaufman, Andrew Lumsdaine, Antoine Petitet, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[4] Uday Bondhugula, Albert Hartono, J Ramanujam, and P Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 101–113, 2008.

[5] Richard Vuduc, James W Demmel, and Katherine A Yelick. Oski: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(1):521, 2005.

[6] Matteo Frigo and Steven G Johnson. Fftw: An adaptive software architecture for the fft. *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing*, 3:1381–1384, 1998.

[7] M. Zhu, D. Hao, and J. Chen. Compiler autotuning through multiple-phase learning. *ACM Transactions on Software Engineering and Methodology*, 33(4):1–38, 2024.

[8] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[9] J. Chen, N. Xu, P. Chen, and H. Zhang. Efficient compiler autotuning via bayesian optimization. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1198–1209. IEEE, May 2021.

[10] S. Balla and F. Koushanfar. HELiKs: HE linear algebra kernels for secure inference. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 2306–2320, November 2023.

[11] G. Kwasniewski, M. Kabic, T. Ben-Nun, A. N. Ziogas, J. E. Saethre, A. Gaillard, and T. Hoefler. On the parallel i/o optimality of linear algebra kernels: Near-optimal matrix factorizations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, November 2021.

[12] Albert Hartono, Boyana Norris, and P Sadayappan. Orio: An annotation-based performance tuning tool. *Proceedings of the 2009 International Workshop on Languages and Compilers for Parallel Computing*, pages 364–378, 2009.

[13] H. Pan, Y. Wei, M. Xing, Y. Wu, and C. Zhao. Towards efficient compiler auto-tuning: Leveraging synergistic search spaces. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, pages 614–627, March 2025.

[14] Qing Yi, Keith Seymour, Haihang You, Richard Vuduc, and Daniel Quinlan. Predictive modeling in a polyhedral optimization space. *Proceedings of the 2007 International Symposium on Code Generation and Optimization*, pages 119–129, 2007.

[15] Prasanna Balaprakash, Stefan M Wild, and Paul D Hovland. Autotuning in high-performance computing applications. *Proceedings of the IEEE*, 106(11):2068–2083, 2018.

[16] L. Liao, H. Li, W. Shang, and L. Ma. An empirical study of the impact of hyperparameter tuning and model optimization on the performance properties of deep neural networks. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(3):1–40, 2022.