STM JOURNALS

Recent Trends in
Parallel Computing

ISSN: 2393-8749
Volume 13, Issue 1, 2026
January–April
DOI (Journal): 10.37591/RTPC

https://journals.stmjournals.com/rtpc

Review RTPC

Design and Optimization of Domain-Specific Languages for High-Performance Computing Applications

Bhavisha Vishalbhai Parvadiya*

Abstract

The accelerating demand for computational power in scientific, engineering, and data-intensive domains has driven high-performance computing (HPC) systems toward unprecedented levels of parallelism and architectural complexity. Contemporary HPC platforms integrate multicore CPUs, many-core GPUs, accelerators, and deep memory hierarchies, creating significant challenges for software development and performance optimization. Traditional general-purpose programming languages and parallel programming frameworks provide low-level control over hardware resources but require extensive manual tuning, resulting in poor productivity and limited performance portability. Domain-specific languages (DSLs) have emerged as a transformative approach to HPC software development by offering high-level abstractions that encode domain knowledge directly into language constructs and compiler infrastructures. This enables aggressive, semantics-aware optimizations that are difficult or impossible to achieve using conventional programming paradigms. This paper presents a comprehensive and deeply analytical study of the design principles, classification, and optimization techniques of DSLs for HPC applications. It further examines prominent DSL frameworks through detailed case studies, analyzes their performance benefits, and discusses open challenges and future research directions in the context of exascale and energy-aware computing.

Keywords: Compiler optimization, domain-specific languages, heterogeneous systems, high-performance computing, parallel architecture, performance portability

INTRODUCTION

High-performance computing (HPC) has become a foundational technology underpinning modern scientific discovery, advanced engineering design, and large-scale data analytics. Applications such as global climate simulation, weather prediction, computational chemistry, nuclear fusion modeling, aerospace design, and artificial intelligence training require the processing of massive datasets and execution of complex numerical algorithms at extreme scales. The computational requirements of these applications continue to grow exponentially, placing increasing pressure on both hardware capabilities and software efficiency of the devices.

***Author For Correspondence**
Bhavisha Vishalbhai Parvadiya
E-mail: bhavishaparvadia@gmail.com

Assistant Professor, Department of Computer Science and Engineering, Sardar Patel College of Administration and Management, Gujarat, India

Received Date: January 28, 2026
Accepted Date: February 01, 2026
Published Date: March 20, 2026

Citation: Bhavisha Vishalbhai Parvadiya. Design and Optimization of Domain-Specific Languages for High-Performance Computing Applications. Recent Trends in Parallel Computing. 2026; 13(1): 17–22p.

The architectural evolution of HPC systems has shifted dramatically from homogeneous CPU-based platforms to heterogeneous systems composed of multicore CPUs, GPUs, vector processors, and specialized accelerators. Although these architectures offer exceptional peak performance, achieving sustained performance in practice is extremely challenging. Programmers must explicitly manage fine-grained parallelism, data locality, synchronization, and communication, often at multiple levels of the memory hierarchy [1].

Historically, HPC applications have been

implemented using general-purpose languages such as C, C++, and Fortran, combined with explicit parallel programming models such as MPI, OpenMP, CUDA, and OpenACC. Although these approaches provide direct control over the hardware, they significantly increase the code complexity and development time. Moreover, performance optimizations are often tightly coupled with specific hardware architectures, leading to poor portability and high maintenance costs as systems evolve [2].

Domain-specific languages (DSLs) address these challenges by elevating the level of abstraction. Instead of programming at the level of threads, memory buffers, and synchronization primitives, DSLs allow developers to express computations in terms of domain concepts such as tensors, grids, meshes, stencils, and operators. The responsibility of translating these high-level specifications into efficient low-level implementations is delegated to the DSL compiler and the runtime system. This paradigm shift has the potential to simultaneously improve productivity, performance, and portability, making DSLs a central research topic in modern HPC [3, 4].

BACKGROUND AND MOTIVATION

Complexity of HPC Architectures

Modern HPC architecture is characterized by extreme concurrency, hierarchical memory systems, and heterogeneous execution units. Exploiting these architectures efficiently requires a deep understanding of hardware characteristics, including cache behavior, memory bandwidth limitations, vectorization, and communication costs. Even minor inefficiencies in data movement or synchronization can lead to significant performance degradations [5].

Furthermore, the rapid pace of hardware innovation introduces critical challenges in software sustainability. Applications must be continuously adapted to new architectures, often requiring extensive refactoring and retuning of the models. This situation highlights the limitations of low-level hardware-centric programming approaches and underscores the need for higher-level abstractions.

Rationale for Domain-Specific Languages

The fundamental idea behind DSLs is to restrict the programming model to a specific application domain. This restriction enables the compiler to reason more precisely about the program behavior and apply optimizations that would be unsafe in a general-purpose context. By embedding domain knowledge directly into the language, DSLs can do the following:

- Expose inherent parallelism explicitly
- Optimize memory access patterns automatically
- Reduce data movement and synchronization overhead
- Improve performance portability across architecture

Empirical studies have demonstrated that DSL-based systems can achieve performance comparable to or even exceeding hand-optimized code while significantly reducing development effort and code complexity [6].

CLASSIFICATION OF DOMAIN-SPECIFIC LANGUAGES

Embedded Domain-Specific Languages

Embedded DSLs are implemented within a host language, which reuses its syntax, type system, and development tools. This approach facilitates rapid prototyping and seamless integration with the existing codebases. However, optimizations are constrained by the semantics and compilation model of the host language, limiting the potential for domain-specific performance tuning [4].

Standalone Domain-Specific Languages

Standalone DSLs define independent syntax, semantics, and compiler infrastructure. This design provides full control over code generation and optimization strategies, making it ideal for performance-critical HPC applications. The primary drawback is the significant effort required to develop and maintain custom compilers and tools.

Library-Based Domain-Specific Languages

Library-based DSLs appear as conventional software libraries but internally perform transformations such as source-to-source compilation, just-in-time compilation, or runtime optimization. They offer a balance between usability and high performance, making them popular for numerical computing, machine learning, and data-intensive HPC workloads.

DESIGN PRINCIPLES OF HPC-ORIENTED DSLs

Domain-Centric Abstraction

Effective DSLs align language constructs closely with the mathematical and conceptual structures of the target domain, such as grids, tensors, and differential operators. This alignment enhances expressiveness, reduces programming complexity, and provides compilers with rich semantic information that can be exploited for aggressive domain-specific optimization [7].

Separation of Algorithm and Execution Strategy

Separating the algorithm from its execution strategy is a key design principle in DSL development. This separation allows developers to experiment with multiple optimization techniques, such as loop tiling, vectorization, or parallel scheduling, without modifying the underlying algorithm, thereby improving code reuse, adaptability, and maintainability [8].

Declarative Programming Paradigm

Declarative DSLs focus on specifying what computation should be performed, rather than how it should be executed. This abstraction grants compilers flexibility to determine the optimal execution strategy, enabling automatic optimization and portability across heterogeneous hardware platforms.

Explicit Parallel Patterns

Many DSLs provide explicit representations of parallel patterns, such as maps, reductions, stencils, and pipeline operations. By exposing these patterns, DSLs enable automatic parallelization and efficient mapping to multicore CPUs, GPUs, and distributed-memory systems while reducing manual parallel programming effort.

OPTIMIZATION TECHNIQUES IN DSL COMPILERS

Compile-Time Optimizations

DSL compilers leverage extensive static analyses to perform transformations such as loop fusion, unrolling, tiling, constant propagation, and dead code elimination. These compile-time techniques reduce computational overhead, enhance instruction-level parallelism, and improve cache locality, leading to substantial performance gains in modern HPC architectures [9].

Memory-Centric Optimizations

Because memory access often dominates the execution time in HPC applications, DSLs emphasize memory-aware strategies. The techniques include data layout transformations, cache blocking, and minimizing data transfers between the host and accelerators. These optimizations improve memory bandwidth utilization and reduce latency, which are critical for large-scale simulations [10].

Parallel and Heterogeneous Code Generation

DSLs automatically generate optimized backend codes for diverse platforms, including multicore CPUs, SIMD vector units, GPUs, and distributed-memory systems. By abstracting hardware-specific details, DSLs enable developers to achieve high performance without manual tuning for each target architecture separately.

Auto-Tuning and Adaptive Optimization

Auto-tuning frameworks explore multiple parameter configurations, such as tile sizes, thread assignments, and memory layouts, to identify the optimal execution strategy for a given platform. Combined with adaptive runtime systems, this approach enhances performance portability and allows applications to dynamically adjust to hardware heterogeneity and workload variations [11].

REPRESENTATIVE CASE STUDIES

Halide

Halide is a domain-specific language for image and signal processing that introduces a clear separation between algorithm specifications and execution scheduling. This design allows developers to explore multiple optimization strategies, such as parallelization, vectorization, and memory optimization, without altering the core algorithm. Consequently, Halide achieves high performance and portability across diverse hardware architectures [8].

FEniCS

FEniCS is a DSL framework for scientific computing that enables users to express partial differential equations in a high-level mathematical form. These formulations are automatically translated into an optimized solver code, supporting scalable parallel execution. FEniCS significantly improves developer productivity while maintaining the performance of large-scale simulations [12].

ANTAREX DSL

The ANTAREX DSL targets performance and energy efficiency challenges in exascale systems through autotuning and runtime adaptation. Its non-intrusive design allows optimization strategies to be applied without modifying the application source code, thereby enabling dynamic performance and power-aware optimization in heterogeneous HPC environments [13].

CHALLENGES AND LIMITATIONS

Despite the substantial benefits offered by Domain-Specific Languages (DSLs) in improving productivity, performance portability, and optimization efficiency, several challenges and limitations hinder their widespread adoption in high-performance computing (HPC) environments. One major limitation is restricted domain applicability; DSLs are typically designed to target narrow problem domains such as linear algebra, stencil computations, or machine learning. While this specialization enables aggressive optimization, it reduces flexibility and makes DSLs unsuitable for applications that span multiple computational domains or require irregular flow control, thereby limiting their general-purpose usability [1, 2].

Another significant challenge is debugging and program comprehension. The high-level abstractions provided by DSLs often obscure the generated low-level code, making it difficult for developers to trace performance bottlenecks or correctness errors. Traditional debugging tools are generally designed for general-purpose languages and offer limited support for DSL-specific semantics, compiler transformations, and runtime behavior. Consequently, diagnosing errors or validating performance becomes more complex, particularly when aggressive compiler optimizations or code generation techniques are applied [3].

The immaturity of tooling ecosystems further constrains the adoption of DSLs. Compared to established HPC programming models such as MPI, OpenMP, or CUDA, many DSLs lack comprehensive development environments, including robust debuggers, profilers, static analysis tools, and integrated development environment (IDE) support. This lack of mature tooling increases the learning curve for developers and reduces confidence in deploying DSL-based solutions in production-level HPC systems [4].

In addition, integration with legacy HPC software presents a critical barrier. Many large-scale scientific applications represent decades of development and optimization in languages such as Fortran, C, and C++. Incorporating DSLs into existing codebases often requires substantial refactoring, interoperability layers, or hybrid programming models, which can introduce performance overheads and maintenance complexity. Ensuring seamless coexistence between DSL-generated codes and legacy components remains an open research challenge [5].

Addressing these challenges through improved compiler transparency, enhanced tooling support, better interoperability mechanisms, and more flexible DSL design methodologies remains an active and

important area of research. Overcoming these limitations is essential for enabling the broader adoption of DSLs as a foundational technology for future exascale and post-exascale HPC systems.

FUTURE RESEARCH DIRECTIONS

Future research on domain-specific languages (DSLs) for high-performance computing (HPC) is expected to advance along several complementary directions as computing systems move toward exascale and beyond. One of the most promising areas is machine-learning-guided compiler optimization, in which data-driven models are used to predict optimal transformation sequences, scheduling strategies, and hardware mappings. By learning from prior compilation and execution data, machine learning techniques can significantly reduce the search space of autotuning and enable adaptive optimization strategies that outperform traditional heuristic-based approaches [1, 2].

Another critical direction is the development of energy-aware and power-efficient DSLs. As energy consumption is a dominant constraint in large-scale HPC systems, future DSLs must incorporate energy as a first-class optimization objective alongside performance. This includes exposing energy-related parameters at the language level and enabling compilers and runtimes to dynamically trade off performance, power, and thermal constraints. Energy-aware DSLs are expected to play a key role in sustainable exascale computing [3].

Research is also moving toward unified multi-domain DSL frameworks that can support multiple scientific domains within a common infrastructure. Rather than developing isolated DSLs for individual applications, future systems aim to provide extensible compiler platforms that allow for reusable abstractions, shared optimization passes, and interoperable runtime systems. Such unified frameworks can significantly reduce development efforts and improve long-term maintainability while preserving domain-specific optimization capabilities [4].

Finally, dynamic runtime adaptation is expected to become increasingly important for exascale systems characterized by hardware heterogeneity, system variability, and fault-tolerance requirements. Future DSLs will be integrated closely with intelligent runtime systems capable of monitoring execution behavior and adapting scheduling, data placement, and parallelization strategies at runtime. This adaptive approach enables applications to respond to changing system conditions, thereby improving resilience, efficiency, and overall performance scalability [5, 6].

Together, these research directions highlight the evolving role of DSLs as intelligent, adaptive, and energy-conscious programming frameworks that are essential for next-generation HPC applications.

CONCLUSION

Domain-specific languages (DSLs) represent a fundamental paradigm shift in the development of high-performance computing (HPC) software by redefining the relationship between application logic and hardware execution. Unlike traditional general-purpose programming approaches that require developers to manually manage low-level architectural details, DSLs elevate programming to a higher level of abstraction rooted in domain concepts such as tensors, grids, stencils, and mathematical operators. This abstraction enables developers to express complex computational algorithms in a concise, readable, and maintainable manner, while delegating the responsibility of performance optimization to sophisticated compilers and runtime systems. By embedding domain knowledge directly into language semantics, DSLs enable aggressive domain-aware optimizations, such as automatic parallelization, memory layout transformation, and hardware-specific code generation, which are difficult or unsafe to implement manually in conventional languages. DSLs provide a sustainable and scalable solution for achieving high performance, performance portability across heterogeneous architectures, and improved programmer productivity. With the continued evolution of HPC systems toward exascale computing and increased architectural heterogeneity, DSLs are increasingly recognized as critical enabling technologies for future scientific and engineering applications.

REFERENCES

1. Kirk DB, Hwu WMW. Programming Massively Parallel Processors: A Hands-On Approach. 3rd ed. Burlington (MA): Morgan Kaufmann Publishers; 2016.
2. Dongarra J, Beckman P, Moore T, Aerts P, Aloisio G, Andre JC, et al. The International Exascale Software Project roadmap. Int J High Perform Comput Appl. 2011;25:3–60. doi:10.1177/1094342010391989.
3. Gibbons J, Wu N. Folding domain-specific languages: deep and shallow embeddings (functional pearl). ACM SIGPLAN International Conference on Functional Programming (ICFP '14), Gothenburg, Sweden. 2014 Sep 1–6. p. 339–347. doi:10.1145/2628136.2628138.
4. Sujeeth AK, Brown KJ, Lee H, Rompf T, Chafi H, Odersky M, Olukotun K. Delite: a compiler architecture for performance-oriented embedded domain-specific languages. ACM Trans Embed Comput Syst. 2014;13:1–25. doi:10.1145/2584665.
5. Hennessy JL, Patterson DA. Computer Architecture: A Quantitative Approach. 5th ed. Waltham (MA): Elsevier; 2011.
6. Mernik M, Heering J, Sloane AM. When and how to develop domain-specific languages. ACM Comput Surv. 2005;37:316–344. doi:10.1145/1118890.1118892.
7. DeVito Z, Hegarty J, Aiken A, Hanrahan P, Vitek J. Terra: a multi-stage language for high-performance computing. Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, Seattle, WA, USA. 2013. p. 105–116. doi:10.1145/2491956.2462166.
8. Ragan-Kelley J, Adams A, Sharlet D, Barnes C, Paris S, Levoy M, Amarasinghe S, Durand F. Halide: decoupling algorithms from schedules for high-performance image processing. Commun ACM. 2018;61(1):106–115. doi:10.1145/3150211.
9. Oancea CE, Rauchwerger L. Logical inference techniques for loop parallelization. Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, Beijing, China. 2012. p. 509–520. doi:10.1145/2254064.2254124.
10. Choi JW, Bedard D, Fowler R, Vuduc R. A roofline model of energy. 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS), Cambridge, MA, USA. 2013. p. 661–672. doi:10.1109/IPDPS.2013.77.
11. Püschel M, Moura JMF, Johnson JR, Padua D, Veloso MM, Singer BW, Xiong J, Franchetti F, Gacic A, Voronenko Y, Chen K, Johnson RW, Rizzolo N. SPIRAL: code generation for DSP transforms. Proc IEEE. 2005;93(2):232–275. doi:10.1109/JPROC.2004.840306.
12. Logg A, Mardal KA, Wells GN, editors. Automated Solution of Differential Equations by the Finite Element Method: The FEniCS book. Lecture Notes in Computational Science and Engineering. Berlin: Springer Science & Business Media; 2012. doi:10.1007/978-3-642-23099-8.
13. Silvano C, Agosta G, Bartolini A, Beccari AR, Benini L, Besnard L, et al. ANTAREX: A DSL-based approach to adaptively optimizing and enforcing extra-functional properties in high performance computing. 2018 21st Euromicro Conference on Digital System Design (DSD), Prague, Czech Republic. 2018. p. 600–607. doi:10.1109/DSD.2018.00105.

