D-TEC – Domain Specific Language Support for Exascale

Progress Report: June 2018

MIT Team
Saman Amarasinghe (Team Coordinator)
Adam Chlipala
Armando Solar-Lezama
Youssef Marzouk
Michael Carbin
Una May O'Reilly

1 Introduction

Exascale Challenges Parallel computers today are mainly programmed using message passing and coarse-grained threads (e.g. MPI [14] and OpenMP [11]). These models provide high performance, but are tedious to use and error prone. Looking ahead to exascale, there will be a dramatic increase in hardware complexity, which will lead to intolerable software complexity using current programming models.

DSLs, Promise and Obstacles Domain specific languages (DSLs) ore a path to Exascale software based on using automated code transformation to shift the burden of portable performance from the application programmer to the compiler.

The promise of DSLs has been amply demonstrated by the success of at least a dozen DSLs for scientific computing. Consequently, interest in DSLs for HPC has increased in recent years as evidenced by publications, workshops, and conferences. Indeed, the X-Stack FOA for this proposal expects that scientists will use DSLs "to focus on their science rather than the fine details of a complex Exascale system". However, widespread adoption of the DSL approach is hindered by major implementation challenges. First, it takes a large effort to implement even a small programming language. Most commonly used implementation techniques support only part of the effort, such as parsing or interpretation, while leaving difficult tasks like semantic analysis, optimization, or code generation to be implemented manually. Second, most techniques support only restricted forms of DSLs, omitting such desirable characteristics as custom concrete syntax, custom control structures, the ability to interleave DSL and conventional code or multiple DSLs in the same file, or the ability to call conventional libraries in DSL code. Third, most techniques produce implementations which are markedly inferior to conventional compilers, omitting important features

such as thorough semantic checking, global optimization of DSL code, optimization across boundaries between DSL and conventional code or between multiple DSLs, and debugging and profiling compiled code at the DSL source level. Recently a few authors have shown how to avoid most of these limitations using functional host languages or novel research languages, but that work is not applicable to common high performance computing languages (e.g. Fortran, C++, Python).

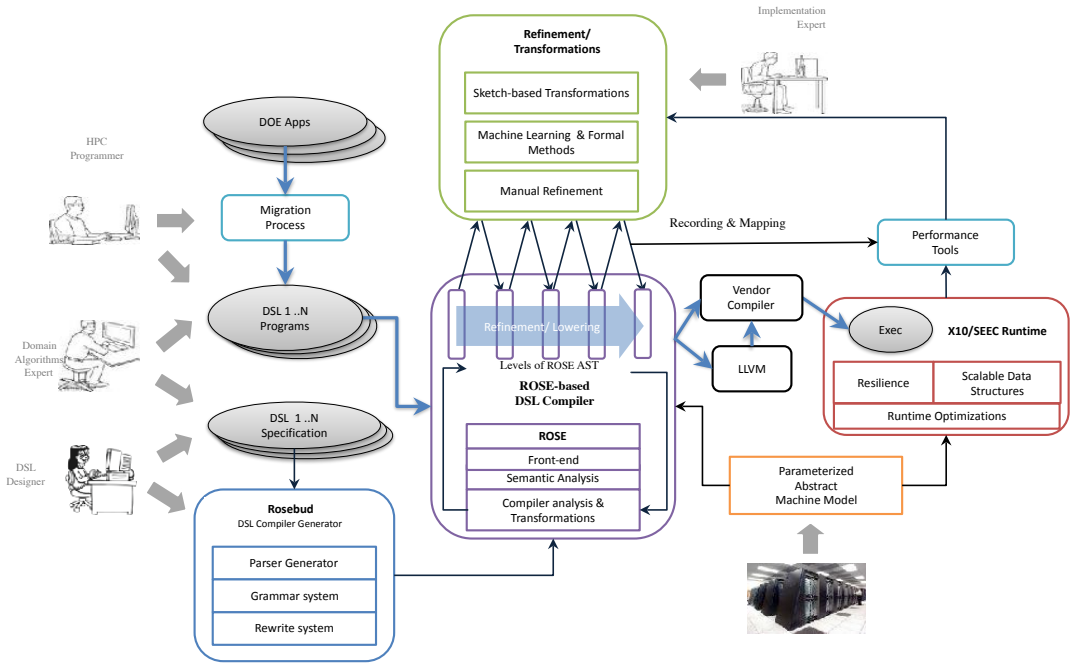


Figure 1: Overview of our DSL development approach, defining the support to make it economical to build DSLs for narrowly defined domains.

Summary of Original Proposal The initial project (referred to as D-TEC) was for three years and started fall of 2012; its focus has been on the development of the infrastructure supporting DSLs. The goal is that DSLs can be built easily and economically to address the requirements of future HPC software development on Exascale architectures. As shown in Figure 1, we proposed to use a novel integrated approach to support the lifecycle of DSL development and involve all major players including HPC programmers, domain algorithm experts, DSL designers, implementation experts, and so on. The resulting software artifacts and processes will include components supporting application migration, DSL specifications, DSL language parsing, compiler construction, refinement and transformation, performance analysis, runtime support, as well as parameterizing abstract extreme-scale machines.

Proposed Work For this one-year renewal proposal, the plan was to continue the research of DSL techniques to support Exascale computing across multiple layers of the software stack and directly address critical Exascale research issues such as performance, locality, resiliency and uncertainty quantification. The plan focusses on integrating many of the technologies that we have

built as well as integrate D-TEC artifacts with technologies developed by other teams. Within our own project, the goal is to continue to refine compiler and runtime optimizations and address interoperability. Working with application groups, the focus is on demonstrating the benefits of the developed DSL techniques across multiple dierent platforms. Collaborate with other X-stack projects to leverage external research and development results and benefit broader user communities is also a part of the plan.

2 Accomplishments

This year, we have three new publications, and two new publications under submission. For this one-year renewal proposal, the MIT team accomplished the following tasks.

- Demonstrate the productivity obtained when using DSLs (i.e., whether they provide good productivity compared to classical approaches).
- Demonstrate the effectivness of DSLs (i.e., whether they can provide best class performance on current hardware and whether they can provide portability).
- Demonstrate frameworks for productive development of highly-expressive DSLs.
- Demonstrate the ability of DSLs to overcome exascale challenges including scalability, resiliency, portability of performance, verification, etc.
- Demonstrate paths from legacy applications to new DSLs (i.e., whether it is possible to translate legacy code into the new DSLs).
- Demonstrate variety of DSLs.

Accomplishments by the other teams are not reported in this progress report. In the remaining of this section we provide details about each one of these accomplishments.

2.1 Demonstrate Productivity to Hero Class Performance

Halide [12] is a programming language and a compiler designed to make it easier to write high-performance stencil computations (structured grids). It targets multiple hardware architectures ranging from multi-core systems and GPUs to supercomputers. After its first appearance in 2012, the Halide language has attracted a high academic and industrial interest. A relatively large community is now participating in the development of the Halide compiler. The language is now being used in production by Google, Adobe, Facebook and Qualcomm and the number of contributors to the Halide compiler reached 60 contributors [1] from different institutions and industrial partners.

Halide has the advantage of separating the algorithm from the optimizations (a.k.a., the scheduling commands). The user writes an architecture-independent algorithm and then provides a set of scheduling commands to indicate how the algorithm should be optimized for a given architecture. To adapt a given code to a new hardware, the user needs only to modify the schedule without the

[1] https://github.com/halide/Halide

need to modify the algorithm. Furthermore, adapting a given piece of code to a new architecture can be automated with the use of an auto-tuner which simplifies the whole process.

The separation between algorithms and scheduling commands also simplifies the exploration of optimizations: one can quickly try different optimizations simply by writing different scheduling commands. The compiler would generate the optimized code automatically. This simplifies greatly the exploration of optimization trade-offs.

The Halide DSL expresses Local Laplacian Filtering (one of the important algorithms used in Adobe Photoshop) in a mere 60 lines of code within a few hours, compared to the actual implementation in Photoshop, which is 1500 lines of code written over 3 programmer-months and a reference (unoptimized) C implementation that uses 300 lines of code. The Photoshop version is threaded and vectorized, resulting in mixing of optimizations with algorithm, while the Halide version separates out the domain-specific code from the schedule that describes how to optimize it. This enables easy and quick exploration of the space of optimizations and enables high productivity in general.

2.2 Demonstrate the Effectiveness of DSLs

2.2.1 Demonstrate Best of Class Performance on Current Hardware

We ported the HPCG benchmark (`http://hpcg-benchmark.org`) to the Tiramisu compiler.

Tiramisu is a compiler for expressing fast and portable data parallel computations. It provides a simple C++ API for expressing algorithms (Tiramisu expressions) and how these algorithms should be optimized by the compiler. Tiramisu can be used in areas such as linear and tensor algebra, deep learning, image processing, stencil computations and machine learning. The Tiramisu compiler is based on the polyhedral model thus it can express a large set of loop optimizations and data layout transformations. Currently it targets (1) multicore X86 CPUs, (2) Nvidia GPUs, (3) Xilinx FPGAs (Vivado HLS) and (4) distributed machines (using MPI). It is designed to enable easy integration of code generators for new architectures.

The Tiramisu implementation was $1.2\times$ faster than the original HPCG implementation (C+OpenMP).

We also implemented a set of dense linear algebra kernels in TIRAMISU and compared the performance of generated code with that of highly optimized libraries such as Intel MKL Blas. The Tiramisu implementation matched the performance of the most challenging Intel MKL BLAS kernels such as gemm (generalized matrix multiplication). Tiramisu is the first compiler to achieve such performance.

2.2.2 Demonstrate Performance Portability on Existing Hardware

We ported the HPGMG miniapp to Halide. The original miniapp was written in C with OpenMP for shared memory parallelism.

Porting the original HPGMG miniapp (C+OpenMP) to Halide was straightforward and a schedule for CPU (i.e., optimizations for execution on CPU) were generated automatically using OpenTuner (our auto-tuning tool).

The Halide implementation was $1.3\times$ faster than the original implementation (C+OpenMP).

To demonstrate the performance portability on existing hardware, we ported the HPGMG Halide implementation to GPU. To do that, we changed the original CPU schedule to a new GPU

schedule. This meant changing few scheduling commands (i.e., few lines of code). No modification on the algorithm was needed and only the schedule part was modified.

The Halide code ported to GPU was $1.77\times$ faster than the original HPGMG implementation (C+OpenMP).

2.3 Demonstrate more Productive Development of Highly-expressive DSLs

2.3.1 TIRAMISU

Building an effective mid-level compiler requires ensuring enough information is passed from higher levels to enable optimizations. Furthermore, if the representation used in the mid-level compiler is too low-level, transformations and optimizations may require undoing work from the higher level layer. For example, LLVM and other low-level compiler frameworks use compact three-address instruction sets with a single monolithic memory abstraction to optimize for single-threaded performance. However, this abstraction is insufficient for the middle-end. First, information about coarse-grain parallelism is missing, making it difficult to identify parallelization opportunities. Second, memory layout decisions are already made, making it difficult to apply optimizations such as loop fusion, parallelization, and specialization for NUMA or GPUs, since such transformations often require drastic rearrangement of memory layouts for intermediates. Thus, building a new mid-level compiler requires introducing a new representation.

TIRAMISU introduces a novel three-layered representation that is ideal for mapping between the architecture-independent front-end to a single-thread-optimizing back-end, while taking advantage of all the high-level architectural features. The top layer, the *Abstract Computation Layer*, describes only the computations to perform. At this level, memory locations are not considered and all dependences are represented using producer-consumer relationships. The second layer, the *Computation Placement Layer*, maps statements onto an explicit processor and virtual timeline of execution. The final layer, *Concrete Computation Layer*, specifies where to store produced data until they are consumed. We use a mathematical representation based on the polyhedral model [8, 9, 1, 4, 6] with extensions to handle irregular programs [5, 3] (programs with non-affine control flow and array accesses). TIRAMISU is not an automatic parallelizing compiler. Transformation decisions are left to the front-end compiler; TIRAMISU only provides mechanisms to implement these decisions.

TIRAMISU is designed mainly for programs that operate over dense data using loop nests. It allows compilers to transform their architecture-independent intermediate representations to a single-thread-optimizing back-end intermediate representation while taking advantage of modern architectural features such as multicore parallelism, non-uniform memory (NUMA) hierarchies, clusters, and accelerators like graphics processing units (GPUs).

The design of the IRs enables the framework to apply advanced transformations on arbitrary loop nests. A typical workflow for using TIRAMISU is as follows: High level language compilers parse input programs and perform domain specific optimizations before translating the program into Layer I of the TIRAMISU intermediate representation. The first layer of the IR is then transformed to lower layers (Layer II and Layer III), and finally LLVM IR is generated.

TIRAMISU provides multiple advantages

- The use of TIRAMISU reduces the effort needed for developing a high level language compiler since it provides a rich, generic and reusable infrastructure for code optimization and

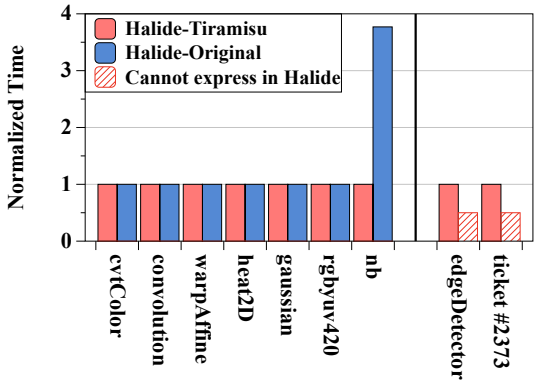


Figure 2: CPU execution time comparison between Halide-Original and Halide-TIRAMISU

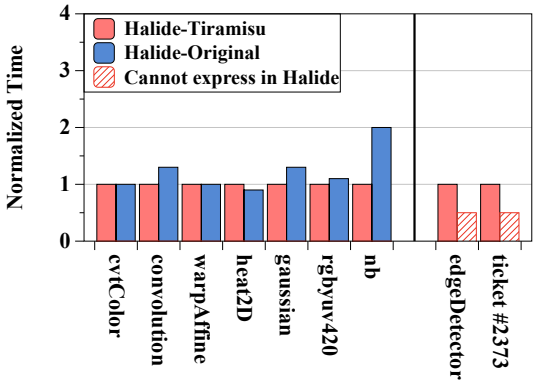


Figure 3: GPU kernel execution times for Halide-Original and Halide-TIRAMISU

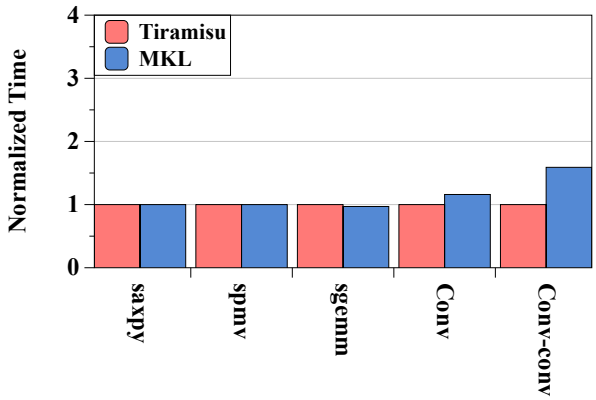


Figure 4: Comparing CPU code generated from TIRAMISU with Intel MKL

code generation;

- The novel three-level IR of TIRAMISU allows full architecture independence, thus high level languages that target TIRAMISU can be fully architecture-independent which leads to code portability;
- TIRAMISU uses state-of-the-art polyhedral techniques with many pragmatic extensions and thus it can perform a large set of complex code transformations and can generate highly very efficient code.

We have implemented a prototype version of TIRAMISU and integrated this prototype version into the Halide compiler. The preliminary results are very encouraging. Figure 2 shows that, in six benchmarks, the performance of code generated from TIRAMISU (called Halide-TIRAMISU in the Figure) matches the performance of code generated from Halide (called Halide-original in the figure). In one of the benchmarks, the performance of code generated from TIRAMISU outperforms that of Halide. This is because TIRAMISU could perform optimizations that are difficult to do in Halide. In two other benchmarks, TIRAMISU could express computational patterns that the

current Halide compiler cannot express, therefore TIRAMISU complements Halide with many new capabilities without degrading its current performance. Figure 3 shows a comparison between GPU code generated by TIRAMISU and GPU code generated by Halide.

We also evaluated TIRAMISU by implementing a set of linear algebra and neural network kernels, including `saxpy` ($Y = \alpha X + Y$), `spmv` ($y = Ax$ with A sparse), `sgemm` ($C = \alpha AB + \beta C$), `conv` (a neural network convolution layer), and `conv-conv` (two `conv` layers fused together). Figure 4 shows a comparison between the performance of CPU code generated by TIRAMISU and the Intel MKL library. For linear algebra we use matrices of size 1060×1060 and vectors of size 1060 while for DNN we use 512×512 as the data input size, 16 as the number of input/output features and a batch size of 32.

For `saxpy`, `spmv` and `sgemm`, TIRAMISU matches the performance of Intel MKL. The comparison between the TIRAMISU implementation of `sgemm` and Intel MKL is interesting in particular because the Intel MKL implementation of this kernel is well-known for its hand-optimized performance. We used a large set of optimizations to match Intel MKL. These optimizations include two-level blocking of the three-dimensional `sgemm` loop, fusing the computation of $T = \alpha AB$ and $C = T + \beta C$ into a single loop, vectorization, unrolling, array packing, register blocking, and separation of full and partial tiles (which is crucial to enable vectorization, unrolling, and reduce control overhead). We also used auto-tuning to find the best tile size, unrolling factor and vector length for the machine on which we run our experiments. For the `conv` kernel, TIRAMISU outperforms the Intel MKL implementation due to the tuning of vector size, unrolling factor and tile size. In `conv-conv`, TIRAMISU fuses the two convolution loops which improves data locality.

2.3.2 TACO: a Compiler for Dense and Sparse Linear Algebra

Sparse tensor operations (i.e., n-dimensional matrice operations) are fundamental operations in many types of computations. There are infinite linear and tensor algebra expressions, making it impossible to write code for all of them. We can restrict ourselves to low-order tensors and binary expressions, but then we lose performance because we have to materialize large and often sparse temporaries. This problem cries out for a compiler approach.

Consider the expression $y = \alpha Ax + \beta y$, where A is a matrix, x, y are vectors, and α, β are scalars. The expression comes from the OSKI sparse linear algebra library, and was likely chosen for performance in common cases. Good libraries choose a few variants and provide routines for converting to and from the required formats. However, expressions come in many different shapes and data formats; thus, limiting to a few variants is inefficent at best and inadequate at worst.

The Tensor Algebra Compiler (TACO) is the first system to compile any compound basic linear or tensor algebra expression to code that evaluates the expression through a single pass over the combined (sparse) iteration space of the operands. Large sparse temporaries and format conversions are therefore avoided. The operands can be dense, sparse or mixed, which makes it a powerful mechanism for emitting fused sparse computation from Simit programs.

Research Involved

- Develop a compiler theory for dense/sparse linear and tensor algebra,
- Develop a formalism for describing tensor storage formats,

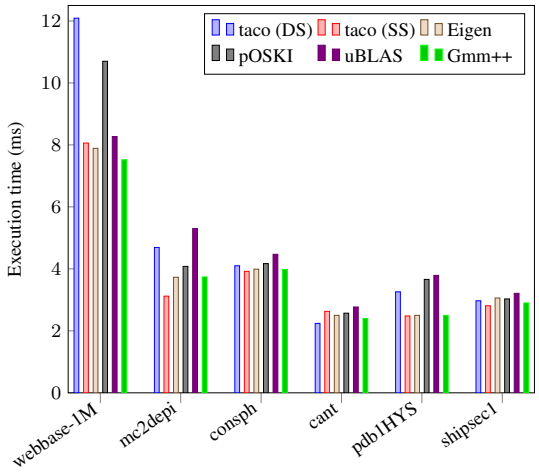


Figure 5: SpMV performance on matrices from real-world applications using taco and various other existing libraries. Results shown here for pOSKI are untuned. We evaluated taco with matrices stored using the dense-sparse (DS) and sparse-sparse (SS) formats, which are equivalent to the compressed sparse row format and the doubly compressed sparse row format respectively.

- Optimizations to exploit tensor structure (symmetry, sparsity, precision),
- Develop a portable library that generates code for different platforms (CPU/GPU), and
- Distributed parallel execution on super-computers.

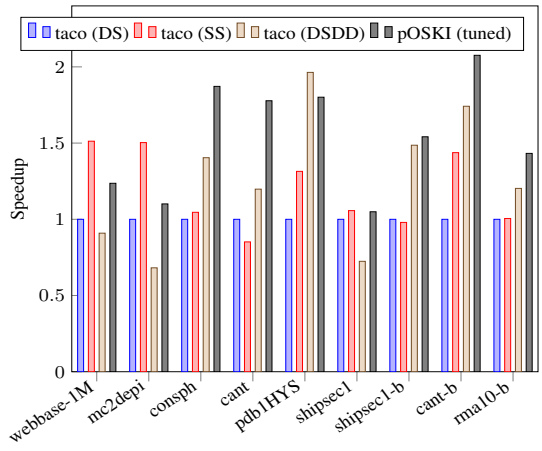
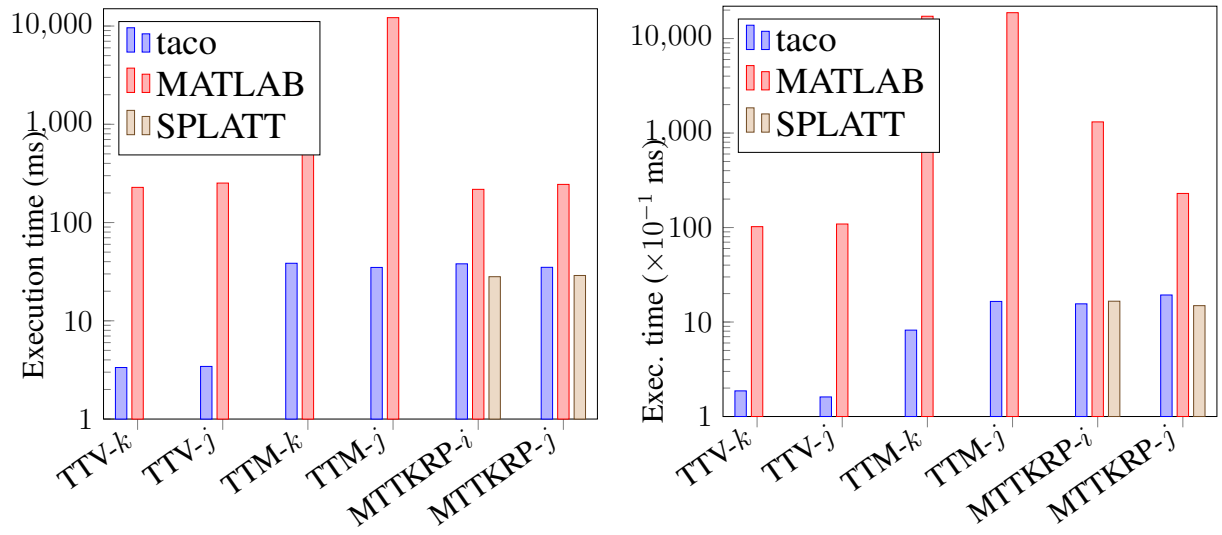


Figure 6: Performance of blocked SpMV on various matrices using taco, compared with tuned pOSKI. We evaluated taco with matrices stored using the dense-sparse, sparse-sparse, and dense-sparse-dense-dense (DSDD) formats, the last of which is equivalent to the blocked compressed sparse row format.

Preliminary experiments have shown that taco is able to synthesize efficient sparse linear and tensor algebra kernels that are competitive with - and even superior to - state-of-the-art, manually-implemented kernels in terms of performance, even as taco supports a much wider set of operations than traditional libraries. As shown in Figures 5 and 6, taco-generated SpMV and blocked SpMV kernels can match the performance of hand-tuned equivalents that are found in high-performance sparse linear algebra libraries like Eigen and even pOSKI, which uses auto-tuning techniques. In

contrast to the aforementioned libraries though, as Figure 7 shows, taco can also generate a wide range of sparse tensor algebra kernels that outperform MATLAB implementations by orders of magnitude and that are even competitive with hand-optimized C equivalents provided by SPLATT.



(a) Facebook dataset

(b) Enron dataset

Figure 7: Performance of sparse tensor algebra kernels generated with taco and implemented in other existing libraries (i.e. MATLAB Tensor Toolbox and SPLATT).

2.3.3 Mostly Automated Formal Verification of Loop Dependencies with Applications to Distributed Stencil Algorithms

The class of stencil programs involves repeatedly updating elements of arrays according to fixed patterns, referred to as stencils. Stencil problems are ubiquitous in scientific computing and are used as an ingredient to solve more involved problems. Their high regularity allows massive parallelization. Two important challenges in designing such algorithms are cache efficiency and minimizing the number of communication steps between nodes. In this work, we introduce a mathematical framework for a crucial aspect of formal verification of both sequential and distributed stencil algorithms, and we describe its Coq implementation. We present a domain-specific embedded programming language with support for automating the most tedious steps of proofs that nested loops respect dependencies, applicable to sequential and distributed examples. Finally, we evaluate the robustness of our library by proving the dependency-correctness of some real-world stencil algorithms, including a state-of-the-art cache-oblivious sequential algorithm, as well as two optimized distributed kernels.

2.4 Demonstrate How to Overcome Exascale Challenges

Exascale challenges include scalability, resiliency, portability of performance, verification, etc. Using DSLs permit the use of high-level abstractions to capture program semantics and automate the generation of low-level code that avoids library overhead and takes into account the particular challenges of operating at exascale.

MSL is a synthesis-enabled DSL for distributed implementations. It makes writing complicated distributed kernels easier by using synthesis. Programmers provide serial spec, parallel skeleton and description of data distribution and the compiler completes implementation using synthesis.

Code generated using *MSL* scales to 16K+ cores and matches the performance of hand-optimized Fortran+MPI (Figure 8)

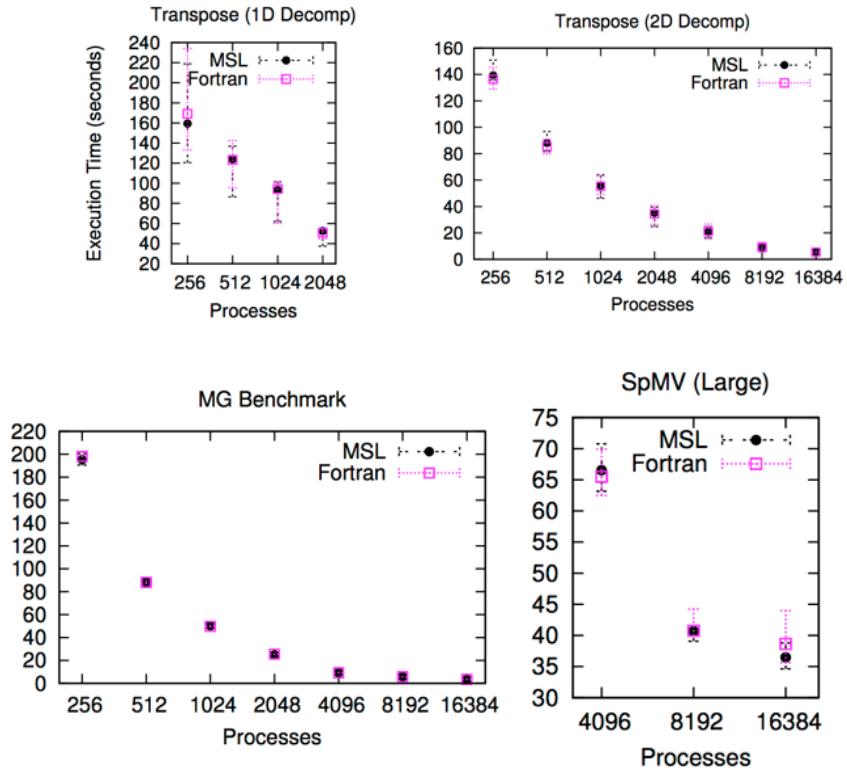


Figure 8: Performance of code generated using *MSL* compared to hand-optimized Fortran+MPI code.

2.5 Demonstrate a Path from Legacy Applications to New DSLs

2.5.1 STNG: Synthesis Tool to Migrate Legacy Fortran Kernels to Halide

We developed STNG, a system to automatically lift the level of abstraction of a low-level Fortran implementation to a high-level stencil DSL (Halide) in a provably correct way.

Lifting low-level Fortran code to a high-level DSL makes it possible to exploit the performance and performance portability benefits of the DSL in the context of legacy code. By lifting the code, we are able to exploit the optimization power of the DSL compiler and achieve order-of magnitude performance improvements on existing code.

The key idea is to leverage our Sketch synthesis infrastructure to search for an expression in the stencil DSL that can be proven to be equivalent to the original Fortran code. We rely on an SMT solver to perform the final verification step in order to guarantee correctness of the transformation. The main advantage of the tool is that it is based on sound and static methods for the verification of equivalence.

Our experiments show that lifting legacy code to a DSL and applying DSL optimizations is better than optimizing original code. The results in Figure 9 are early promising results.

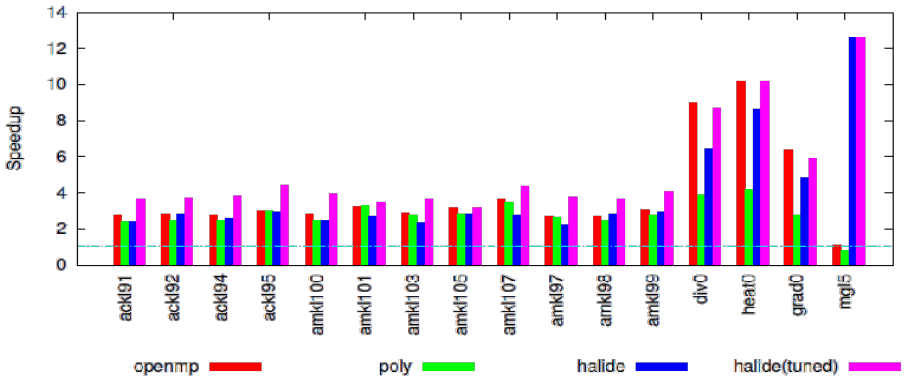


Figure 9: Speedup of automatically synthesised Halide relative the original serial code (the synthesised Halide is further tuned).

2.5.2 Helium: Lifting Legacy Stencil Kernels to Halide DSL

We also extended our Helium system to lift more kernels from DOE relevant applications to the Halide DSL. Helium is a system that uses unsound simple dynamic program analysis to identify stencil kernels and lift them to Halide. We used Helium to lift the miniGMG app from binary code to Halide. We then optimized the obtained Halide. The result was a $4.25\times$ speedup over the original binary.

2.5.3 Sketch-based Synthesis for Lulesh

We created a complete implementation of Lulesh in Sketch. This is the first miniApp fully implemented in our Sketch system. It is allowing us to explore the use of synthesis in the context of a complete miniApp for both shared memory and distributed memory parallelism. It is also allowing us to explore the use of synthesis to exploit thread-level parallelism.

Research Involved We have extended our Sketch system to allow us to leverage synthesis in the context of hybrid shared-memory + message-passing implementations. Sketch now includes a model for deterministic shared memory parallelism that allows the programmer to leverage multi-cores while avoiding the problems with non-determinism that can arise with such programs. Sketch achieves this by automatically inserting checks that prevent data-races that may lead to non-determinism in a kernel.

2.6 Demonstrate a Variety of Domain Specific Languages

2.6.1 Halide DSL for Stencil Computations

Code generated by the Halide compiler [12] is shown to outperform hand-written code. For example, from simple Halide programs written in a few hours, Halide demonstrated performance up

to $5\times$ faster than hand-tuned C, intrinsics, and CUDA implementations optimized by experts over weeks or months, such results are beyond the reach of past automatic compilers.

Our experiments demonstrated that code generated from Halide can run on a cluster and achieve good scalability. Halide is now a lot more relevant for the DoE community. Our experiments also demonstrated that distributed partitioning across NUMA nodes on a SMP provides higher performance than flat partitioning. Figure 10 shows the scalability of a stencil computation generated using Halide and run on the CORI supercomputer (Cray XC40 supercomputer).

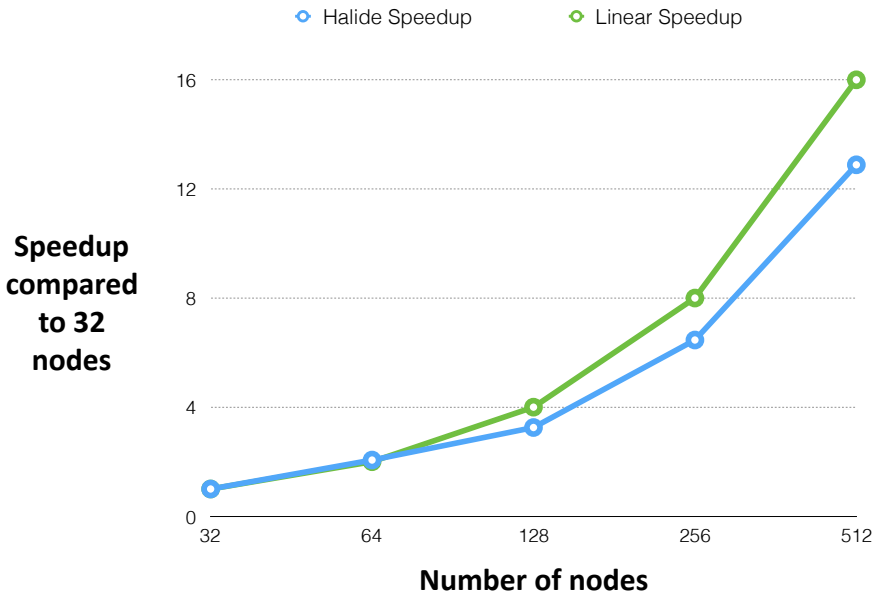


Figure 10: Scalability of code generated from Halide on CORI (Cray XC40)

Research Involved

- Extensions to handle distributed memory and communication.
- Introduced new tiling schemes such as duplication of work to reduce computation across nodes while reusing the data within the node.

2.6.2 Simit DSL

Simulations of detailed physical systems, such as the stresses on airplane wings or LQCD, are computationally intensive, requiring both efficient algorithms and painstakingly-optimized implementation. Many simulation codes are characterized by their *local-global structure*: the heart of simulation deals with solving global problems over large, sparse linear algebraic systems, but the elements in these systems are connected to a tiny subset of the entire system (sites in a lattice are only connected to a few neighbors). Solving the overall problem requires implicit construction of

Table 1: Comparison of two applications implemented with Matlab, Vectorized Matlab, Eigen [2], hand-optimized C++ (SOFA [7] and Vega [13]) and Simit, showing the productivity and performance of Simit. (Intel Xeon E5-2695 v2)

		ms per frame		Source lines		Memory (MB)	
Implicit Springs	Matlab	13,280	23.7×	142	1.5×	1,059	6.1×
	Matlab Vec	2,115	3.8×	230	2.5×	1,297	7.5×
	Eigen	883	1.6×	314	3.4×	339	2.0×
	SOFA	588	1.1×	1,404	15.1×	94	0.5×
	Simit	559	1.0×	**93**	1.0×	173	1.0×
Neo-Hookean FEM	Matlab	207,538	181.3×	234	1.3×	1,564	12.8×
	Matlab Vec	16,584	14.5×	293	1.6×	53,949	442.2×
	Eigen	1,891	1.7×	363	2.0×	626	5.1×
	SOFA	1,512	1.3×	1,541	8.6×	324	1.8×
	Vega	1,182	1.0×	1,080	6.0×	614	5.0×
	Simit	1,145	1.0×	**180**	1.0×	173	1.0×

a large global matrix before solving the resulting system of linear equations. Both the operator construction and the solve are difficult to implement and optimize.

The typical approach taken by practical implementations is to separate the assembly and global stages of the computation. Generically, this leads to C++ code to assemble matrices from a mesh data structure, which are then passed to optimized linear algebra libraries in order to perform the solve in a scalable manner. However, this approach incurs performance penalties due to the need to translate between data structures, as well as the inability of these libraries to take advantage of underlying matrix structure that is apparent at the mesh level. To mitigate this, high-performance implementations integrate assembly and linear algebra using a single set of data structures; however, optimizing such implementations is very challenging, rarely resulting in a reusable or performance portable code.

Simit [10] is a new language and compiler infrastructure for expressing computations characterized by this local-global structure that seeks to enable high-performance simulation code by separating the structure of a system from its behavior. As such, it is well suited for LQCD. The underlying programming model does this by providing two key abstractions as well as the ability to map between them. The first is a hypergraph with hierarchical edges, which represents the geometric structure of a physical system, such as a lattice. The second key abstraction is blocked tensors, a generalization of matrices and vectors, operations on which describe computation on the whole system. The *assembly map* maps between these two abstractions, giving both a local view of assembly on the graph and a global view of system-wide computation as linear algebra.

The assembly map also conveys the relationship between local and global operations to the compilation framework, enabling it to compile the whole program into a globally-optimized representation targeted to a specific architecture. This is combined with a physical representation of the data that is similar to blocked compressed sparse row matrix format. Combining local-global mapping awareness, sparsity-aware code generation and a blocked sparse matrix format results in code that is fully optimized for the particular problem at hand, instead of being cobbled together from libraries which cannot compose optimizations across function call boundaries.

Performance Results Table 1 shows the performance of Simit on two problems, compared with current state of the art implementations using a variety of optimized frameworks. Simit obtains

performance equivalent to hand-optimized code automatically, while allowing the programmer to express algorithms in a high level way with many fewer lines of code. Simit's expressibility is similar to Matlab, but the performance is far greater.

3 Recent Publications

We have published the following papers as part of DTEC funded research (submitted papers currently under review are marked with '*'):

Submitted* Tiramisu: A Code Optimization Framework for High Performance Systems. Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Patricia Suriana, Shoaib Kamil, Saman Amarasinghe. ArXiv e-prints. February, 2018.

Submitted* Sparse Tensor Algebra Optimizations with Workspaces. Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, and Saman Amarasinghe. ArXiv e-prints. April, 2018.

OOPSLA'18 Unified Sparse Formats for Tensor Algebra Compilers. Chou Stephen, Kjolstad Fredrik and Amarasinghe Saman. ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications.

ASE'17 TACO: A Tool to Generate Tensor Algebra Kernels. Kjolstad Fredrik, Chou Stephen, Lugato David, Kamil Shoaib and Amarasinghe Saman. 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE).

OOPSLA'17 The Tensor Algebra Compiler. David Lugato Fredrik Kjolstad, Shoaib Kamil Stephen Chou, Saman Amarasinghe. ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications.

ACM T. Graph. Simit: A Language for Physical Simulation. Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David I. W. Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M. Kaufman, Gurtej Kanwar, Wojciech Matusik, Saman Amarasinghe. ACM Trans. Graph..

PPoPP'16 Distributed Halide. Shoaib Kamil Tyler Denniston, Saman Amarasinghe. Symposium on Principles and Practice of Parallel Programming.

PLDI'16 Verified lifting of stencil computations. Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, Armando Solar-Lezama. Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation.

ITP'16 Mostly Automated Formal Verification of Loop Dependencies with Applications to Distributed Stencil Algorithms. Thomas Gregoire, Adam Chlipala. Proceedings of the Interactive Theorem Proving - Seventh International Conference (ITP'16). August 2016.

PLDI'15 Autotuning Algorithmic Choice for Input Sensitivity. Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O'Reilly, Saman Amarasinghe. ACM SIGPLAN Conference on Programming Language Design and Implementation.

PLDI'15 Helium: Lifting High-Performance Stencil Kernels from Stripped x86 Binaries to Halide DSL Code. Charith Mendis, Jeffrey Bosboom, Kevin Wu, Shoaib Kamil, Jonathan Ragan-Kelley, Sylvain Paris, Qin Zhao, Saman Amarasinghe. ACM SIGPLAN Conference on Programming Language Design and Implementation.

PACT'14 OpenTuner: An Extensible Framework for Program Autotuning. Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, Saman Amarasinghe. International Conference on Parallel Architectures and Compilation Techniques.

4 Software released

We publicly released the following artifacts. We hope the DOE application community will help evaluate them and provide constructive feedback so that these artifacts will become the basis for an operational exascale tool software stack. Software will also be made available through the D-TEC web site [60] and the proposed X-Stack website, if appropriate.

- Halide is released and available at `http://halide-lang.org/`. We will continue to update the release to add features developed as part of this project.
- OpenTuner is available at `http://opentuner.org/` and will continue to be updated as it evolves during the course of this project.
- Tiramisu is released publicly at `https://tiramisu-compiler.org/`.
- Simit is released publicly at `http://simit-lang.org/`.
- TACO is released publicly at `http://tensor-compiler.org/`.
- The ROSE connection to OpenTuner are all released through ROSE compiler available at `http://www.rosecompiler.org/`.
- The new version of Sketch (1.7.2) is released as open source and is available at `http://people.csail.mit.edu/asolar`. We will continue to update the release to add features developed as part of this project.
- MSL, the distributed synthesis language, will be released at the end of the project at `http://people.csail.mit.edu/asolar`.
- A Coq library for verifying dependencies of stencil implementations `https://github.com/mit-plv/stencils`.

References

[1] C. Ancourt and F. Irigoin. Scanning polyhedra with do loops. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '91, pages 39–50, New York, NY, USA, 1991. ACM.

[2] P. Avery, A. Bachrach, S. Barthelemy, C. Becker, D. Benjamin, C. Berger, A. Berres, J. Blanco, M. Borgerding, R. Bossart, K. Brix, G. Brun, P. Buttgenbach, T. Capricelli, N. Carre, J. Ceccato, V. Chalupecky, B. Chretien, A. Coles, M. Danoczy, J. Dean, G. Drenkhahn, C. Ehrlicher, M. Fernandes, D. Ferro, R. Garg, M. Gautier, A. Gladky, S. Glaser, M. Glisse, F. Gosselin, G. Guennebaud, P. Hamelin, M. Hanwell, D. Harmon, C.-P. He, H. Heibel, C. Hertzberg, P. Holoborodko, T. Holy, T. Irons, B. Jacob, B. de Jong, K. Kim, M. Klammler, C. Kahler, A. Korepanov, I. Krivenko, M. Kruisselbrink, A. Kundu, M. Lenz, B. Li, S. Lipponer, D. Lowenberg, D. Luitz, N. Maks, A. Mantzaflaris, D. Marcin, K. Margaritis, R. Martin, R. Marxer, V. Massa, C. Mayer, F. Meier-Dornberg, K. Mierle, L. Montel, E. Nerbonne, A. Neundorf, J. Newton, J. Niesen, D. Nuentsa, J. Oberlander, J. van den Oever, M. Olbrich, S. Pilgrim, B. Piltz, B. Piwowarski, Z. Ploskey, G. Po, S. Popov, M. Rajagopalan, S. Rajko, J. Repinc, K. Riddile, R. Roberts, A. Rodriguez, P. Romn, O. Ruepp, R. Rusu, G. Saupin, O. Saut, B. Schindler, M. Schmidt, D. Schridde, J. Schwendner, C. Seiler, M. Senst, S. Sheorey, A. Somerville, A. Stapleton, S. Strothoff, L. Swirski, A. Szalkowski, S. Traversaro, P. Trojanek, A. Truchet, A. Tsourouksdissian, J. Tyrer, R. Ulerich, H. de Valence, I. Vanhassel, M. van Dyck, S. Wheeler, F. Witherden, U. Wolfer, M. Yguel, and P. Zoppitelli. Eigen software package, v3. `http://eigen.tuxfamily.org`, 2010.

[3] R. Baghdadi, A. Cohen, T. Grosser, S. Verdoolaege, A. Lokhmotov, J. Absar, S. van Haastregt, A. Kravets, and A. F. Donaldson. PENCIL language specification. Research Rep. RR-8706, INRIA, 2015.

[4] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT–13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France, September 2004. Classement CORE : A, nombre de papiers acceptés : 23, soumis : 122, student award.

[5] M. Benabderrahmane, L. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *Proceedings of the International Conference on Compiler Construction (ETAPS CC'10)*, LNCS, Paphos, Cyprus, Mar. 2010. Springer-Verlag.

[6] U. K. Bondhugula. *Effective automatic parallelization and locality optimization using the polyhedral model*. PhD thesis, Citeseer, 2010.

[7] F. Faure, J. Allard, S. Cotin, P. Neumann, P.-J. Bensoussan, C. Duriez, H. Delingette, and L. Grisoni. SOFA: A modular yet efficient simulation framework. In P. Merloz and J. Troccaz, editors, *Surgetica 2007 - Computer-Aided Medical Interventions: tools and applications*, Surgetica 2007, Gestes médicaux chirurgicaux assistés par ordinateur, pages 101–108, Chambéry, France, Sept. 2007.

[8] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, Feb. 1991.

[9] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, July 1967.

[10] F. Kjolstad, S. Kamil, J. Ragan-Kelley, D. I. W. Levin, S. Sueda, D. Chen, E. Vouga, D. M. Kaufman, G. Kanwar, W. Matusik, and S. Amarasinghe. Simit: A language for physical simulation. *ACM Trans. Graph.*, 35(2):20:1–20:21, May 2016.

[11] OpenMP Architecture Review Board. OpenMP application program interface, v3.0, 2008.

[12] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[13] F. S. Sin, D. Schroeder, and J. Barbič. Vega: Non-linear fem deformable object simulator. In *Computer Graphics Forum*, volume 32, pages 36–48, 2013.

[14] D. W. Walker and J. J. Dongarra. Mpi: a standard message passing interface. *Supercomputer*, 12:56–68, 1996.