Polyhedral Compiler Technology in Collaboration with Autotuning Important to Domain-Specific Frameworks for HPC

Mary Hall and Protonu Basu

University of Utah, School of Computing, Salt Lake City, UT 84103
Lawrence Berkeley National Laboratory, Berkeley CA 94721

Abstract. Domain-specific frameworks – including embedded domain-specific languages and libraries – increase programmer productivity by encapsulating proven manual optimization strategies into software modules or (semi-)automated tools. In such frameworks, optimizations and optimization strategies capitalize on knowledge of the requirements of a particular application domain to achieve high performance and architecture portability. While many strategies have been used to develop domain-specific frameworks, this position paper argues the importance of polyhedral compiler technology and autotuning for important classes of high-performance computing domains. Such an approach has the following advantages over other strategies: (1) composability; (2) software reuse; and, (3) facilitates performance portability.

Keywords: domain-specific frameworks, autotuning, polyhedral compiler technology

1 Introduction

The President's National Strategic Computing Initiative of July 2015 established as its first objective to accelerate the "...delivery of a *capable* exascale computing system that integrates hardware and software capability..." If we look at the architectural diversity among current supercomputers and also look forward a few years, it is clear that a variety of specialized accelerators (e.g., Nvidia Pascal GPUs vs. Intel Knights Landing many-cores) and memory systems (e.g., NVRAM and Near-Data Processing) will be developed, and different vendors will provide dramatically different hardware solutions. Consequently, attaining high performance of applications across different exascale platforms may require fundamentally different implementations of software: different algorithms, strategies for parallelization, loop order, data layout and mapping, and exploiting SIMD/SIMT. This need for different implementations is at odds with the goal of *performance portability*, whereby the same application performs well across platforms without significant rewriting. A key concern of the organizations targeting future exascale platforms is the high cost of developing and maintaining performance-portable applications for diverse exascale architectures, including

many-core CPUs and GPUs. Thus, by achieving performance portability, we will also dramatically increase programmer productivity.

Over the last several years, many researchers have addressed performance portability using two key approaches. First, *domain-specific frameworks* – including embedded domain-specific languages and libraries – encapsulate proven manual optimization strategies into software modules and (semi-)automated tools that can produce a collection of architecture-specific implementations. Such frameworks achieve high performance because the optimizations employed and the optimization strategy are specialized to the application domain. Second, *autotuning* involves empirically exploring a search space of possible implementations to identify the best implementation for a particular execution context (e.g., architecture and input data set). By automating the process of evaluating alternatives, autotuning mitigates the need for extensive manual tuning.

While both concepts are well established in the research community, they are nevertheless not widely deployed in the development of HPC applications. As the HPC community prepares for exascale, we must begin now to develop and harden the underlying software capability to provide performance portability and increase programmer productivity; this technology must be ready in a few years to be deployed in exascale applications.

In this position paper, we propose an approach that combines both concepts and, like several research compilers for HPC, relies on *polyhedral transformation and code generation*, which represents loop nest computations mathematically as integer sets, composes sequences of transformations, and generates code using polyhedra scanning. Polyhedral compiler technology and autotuning are well suited to work in collaboration with each other. The mathematical representation of polyhedral frameworks allows the compiler to try a variety of optimization strategies and adjust optimization parameters and still count on being able to generate correct code. Conversely, autotuning frees the compiler developer from having to encode the optimization decisions using a one-size-fits-all algorithm buried inside the compiler implementation. Instead, a variety of optimization strategies can be explored, permitting more aggressive exploration of which transformations to apply.

Our approach separates a high-level C/C++/FORTRAN implementation from architecture-specific implementation (OpenMP, CUDA, etc.), optimization, and tuning. Such an approach would enable exascale application developers to express and maintain a single, portable implementation of their computation, legal code that can be compiled and run using standard tools. An autotuning compiler and search framework, in conjunction with expert programmers and other tools, transforms the baseline code into a collection or search space of highly-optimized implementations. Then autotuning is used to explore this search space and derive final implementations that are best-suited for a specific execution context. *We believe such an approach is reaching a level of maturity that it could realistically be deployed in the early 2020s timeframe for exascale, but it will require institutional support and organization of the parallelizing compiler community to achieve this goal.*

The remainder of this position paper illustrates this approach to productivity and performance portability and its advantages over other approaches to domain-specific frameworks. It concludes by describing the challenges in deploying such an approach in HPC exascale applications.

2 Overview of Approach

Although most of the domain-specific framework literature is not examining HPC applications, the use of domain-specific frameworks in HPC dates back multiple decades, including the Tensor Contraction Engine (a domain-specific compiler), Chombo (a domain-specific C++ library), and high-performance libraries for dense linear algebra (BLAS) and sparse solvers (PETSc).

Recent years have seen polyhedral compiler technology maturing and being applied to code beyond kernels, and deployment in widely-used open source compilers such as LLVM and gcc. Nevertheless, it is broadly considered by potential HPC users to be a technology that is too limited in applicability and too hard to understand. Thus, other "simpler" approaches have gained traction in the HPC application community: (1) specialized manually-written libraries; (2) automatically-generated libraries like ATLAS, SPIRAL and FFTW; (3) specialization through C++ template expansion; (4) single-purpose custom DSLs; and, (5) eDSL frameworks that rely on rewriting rules. While all of these approaches have proven useful, they lack the composability and ability to optimize within context that is afforded from polyhedral frameworks. Therefore, we argue that polyhedral frameworks (in conjunction with autotuning) should be a building block for constructing domain-specific optimization frameworks for HPC.

We draw from our experience in working with application developers and applying the CHiLL autotuning compiler framework to HPC applications across a variety of application domains over the last several years. When used for HPC application code, we argue that the following features are valuable.

- *Composable transformation and code generation:* The importance of having a general and robust transformation framework, where different collections of transformations can be optionally used, is that the same tool can be applied to multiple different application domains. For example, in the last three years, CHiLL has targeted stencils and geometric multigrid, tensor contraction, spectral element methods and sparse linear algebra.
- *Extensible to new domain-specific transformations:* New optimizations that can be represented as transformations on loop nest iteration spaces can be added to such a framework and composed with existing transformations. For example, domain-specific transformations for geometric multigrid including expanding ghost zones and partial sums for higher-order stencils have been composed with existing communication-avoiding optimizations such as fusion and parallel wavefront. For sparse matrices, inspector/executor code generation and support for non-affine transformations are composed with existing tiling, skew, permute, shift and alignment operations. The tensor contraction support does not require new transformations, but only a new tensor-specific decision algorithm.

- *Optimization strategies and parameters exposed to autotuning:* Another requirement is the ability to generate a variety of optimized code that can be explored for different execution contexts. By exposing high-level expression of the autotuning search space as transformation recipes, the compiler writer, an expert programmer or embedded DSL designer can directly express how to compose transformations that lead to different implementations.
- *Search space navigation:* The compiler framework described above provides a way of expressing a search space of different implementations of a computation to target different execution contexts, including architectures, input data sets and phases of a computation. Typically, this search space is prohibitively large to explore in a brute force manner. Thus, autotuning incorporates sophisticated external search space navigation tools that use heuristics and machine learning to accelerate search space exploration and make it feasible. Examples of search space navigation tools used in the HPC community include Orio, Active Harmony and OpenTuner.

3 Deployment Challenges and Research Opportunities

There is a long history of parallelizing compiler technology in the HPC community, and many promising ideas that never made it into practice. Yet combining polyhedral frameworks and autotuning technology is well suited for code generation and optimization required for exascale. There are challenges to make this vision of practical use to HPC application developers; first consider polyhedral frameworks:

- The technology must be robust, widely available and with a long-term maintenance plan. Thus, incorporation into open source compilers with large development teams is needed. There must be a migration path for research advances to move into practice.
- To extend existing open source polyhedral frameworks to support domain-specific systems and autotuning, optimization strategies need to be exposed to the expert programmer and/or domain-specific tool developer.
- The technology must be more broadly applicable. Restricting to loop nest computations is appropriate for HPC, but we must go beyond affine array-based codes; e.g., indirection used in sparse, adaptive and unstructured algorithms, C++ iterators, parallel constructs must be supported.

For autotuning, a number of practical barriers remain:

- Search space navigation must be practical, which becomes more complex as autotuning goals expand.
- Autotuning needs to be part of an application's build process to truly offer performance portability and a path forward. By integrating into Makefiles, autotuning can be repeated after changes to the code or retargeting the application to new platforms or input data sets.
- Co-tuning of multiple related computations is needed to evaluate global optimizations such as data layout.

Acknowledgments. This work has been supported in part by DOE award DE-SC0008682 and NSF award CCF-1564074.