Programming Models for Extreme-Scale Visualization & Data Analysis

Introduction

Simulation and modeling play a fundamental role in the mission of the Department of Energy's Office of Science. In support of these efforts, the DOE has made significant investments over the past two decades in large-scale, high-performance computer systems. During this time, the scientific community has relied heavily on the performance growth of microprocessors within these systems to provide the building blocks for innovation and discovery across a range of disciplines. As energy constraints drive a revolutionary shift towards multi-core and heterogeneous, *throughput-oriented*, processor designs, the community faces a significant challenge in maintaining our rate of productivity across these critical scientific endeavors. In particular, we face a dilemma in finding productive techniques for successfully programming these new architectures. Tightly coupled with the past growth in computational capabilities has been an ever-growing wealth of data with increasingly complex levels of detail. An often under-appreciated aspect of these ongoing architecture changes is the impact they will have on our ability to gather observable, empirical, and measurable data essential to the formulation and testing of hypotheses. A fundamental challenge in guaranteeing the future success of scientific computing within DOE is to provide scientists with an effective, yet flexible way to leverage the power of new architectures to explore and analyze the results of computational experiments. This will require an *end-to-end* solution that begins with the details of the underlying hardware architectures and ends with a set of powerful abstractions that reduce the complexity of programming.

Science-Centric Strategy

The overarching goal of our effort is to provide computational scientists with an approach to software development that promotes the full integration of computation, data analysis, and visualization as *first-class* constructs within a programming language. This is achieved by designing and implementing a *domain-specific language* (DSL [4]) called Scout that incorporates these features into a supporting set of high-level abstractions. Figure 1 presents an example of an image produced using this approach. In this case, particular elements from a materials data set are selected, used to derive associated features and values of interest, and then mapped to a given visual representation. In this way, the approach to programming becomes much more representative and descriptive of the overall workflow. This also enables the DSL compiler to better understand the full context of the supporting computations and leverage that information to improve both optimization and code generation. This is in direct comparison to the common approach of using disparate software components that are typically integrated at a coarse-grained level. As is typical with most DSLs, our supporting set of abstractions, syntax, and semantics are designed to provide application scientists with a more expressive and natural mapping for computations while shielding them from the nuances of the underlying low-level software and hardware architectures.

Language Design

The fundamental abstraction within Scout is that of a computational mesh while the language introduces explicit mesh declarations, instantiations, and parallel computation over the various components of the mesh (e.g. cells, vertices, edges). This abstraction and a restricted set of operations are also used within the Liszt DSL developed at Stanford University [3]. In addition Scout adds features allowing the programmer to control details of how mesh components are visualized based on field values stored within the mesh, and/or derived values computed during the visualization process. We are actively exploring how best to provide programmatic interfaces for supporting multiple visual mappings (e.g. graphs, glyphs, etc.). Unlike C and C++, the underlying data layout details for mesh constructs are not available to the developer and are instead determined by the compiler. The source listing in Figure 2 provides a brief overview of the mesh constructs from the Scout language.

Compiler Toolchain

Our strategy for implementing Scout is to extend C/C++ to support the new set of abstractions and domain-specific constructs. This approach reduces the learning curve of an entirely new language, supports a smoother transition to the adoption of new methods of programming, and decreases some of the complexity of interoperating with legacy codes. We have based our work on the open-source Clang and LLVM Compiler Infrastructure [1, 2]. Furthermore, this avoids the necessity of implementing an entire compiler toolchain – instead focusing on the details of supporting the domain-specific features and the associated runtime components. Finally, the adoption of a full

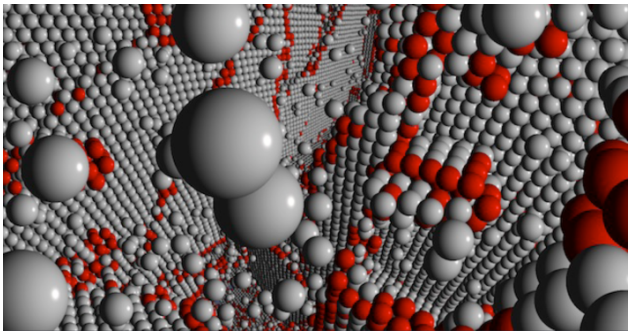

Figure 1: Programmatically selected atoms in a copper block that has been subjected to a shock. The gray atoms show stacking faults and red represents non-close-packed atoms. Our approach allows scientists to program accelerator-based architectures at a high-level of abstraction where these images can be generated at 30-60 frames per second for tens to hundreds of millions of atoms.

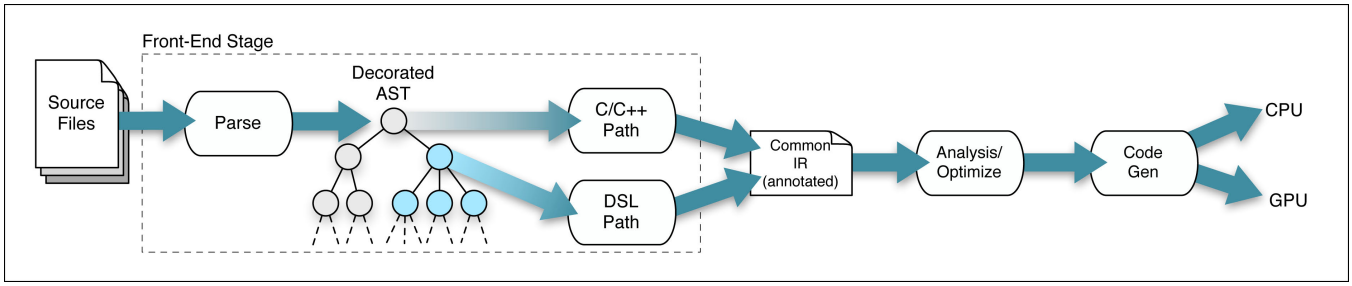

Figure 3: The Scout compiler toolchain leverages an extended C/C++ infrastructure allowing DSL and general-purpose constructs to co-exist within a single program. This approach allows the majority of the traditional compiler toolchain to remain intact, leveraging the broader efforts taking place within the Clang and LLVM communities. This has noticeably increased our flexibility to explore the power of a full compiler toolchain and has reduced the overall costs of development activities.

compiler infrastructure provides the flexibility to maintain and leverage domain-centric knowledge throughout the entire compilation process – something difficult to achieve when using a *source-to-source* approach. Figure 3 presents a high-level view of the modified compiler toolchain. Note that the DSL portions of the code are recognized and processed separately based on the abstract syntax tree (AST) representation of the program. Next both the C/C++ and DSL code are lowered to LLVM's common intermediate representation (IR); a key difference being that the IR from the DSL side still retains domain-specific annotations. At this stage, the IR can pass through both a common set of analysis and optimization passes as well as tailored, domain-centric passes. Using this approach we can take a single Scout program and target sections of a single program to a combination of sequential, multi-core CPU, and GPU code generation targets.

Initial results of this approach have shown distinct advantages to programmatically expressing the mappings from data to visual results. In particular, the expressiveness of a programming language in comparison to the traditional approach of user interface-driven tools, has shown promise. Along these lines, we are actively exploring the use of Scout toolchain within a mixed-language environment for supporting the *in situ* visualization of results within modern application codes. These experiments are designed to help us not only evaluate the overall approach, but to also better understand how the requirements and needs across different scientific disciplines impact the design and underlying implementation of the domain-specific language.

```
uniform mesh UniMesh {
 cells: // fields stored at cell centers
  float   presure, temperature;

 vertex: // fields stored at mesh vertices
  float3  velocity;
};
...
// Construct a two-dimensional instance of the mesh.
UniMesh my_mesh[512, 512];
...
// Compute over all cells of the mesh.
forall cells c of my_mesh {
  ...
}
// Render each cell of the mesh giving it a
// user-specified mapping to colors...
renderall cells c of my_mesh {
  ...
  color = ...;
}
```

Figure 2: An example of Scout's mesh types and supporting language constructs.

References

[1] Clang: A C Language Family Frontend for LLVM. http://clang.llvm.org, January 2012.

[2] The LLVM Compiler Infrastructure Project. http://www.llvm.org, January 2012.

[3] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. Liszt: a domain specific language for building portable mesh-based PDE solvers. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9:1–9:12, 2011.

[4] Martin Fowler. *Domain-Specific Languages (Addison-Wesley Signature Series)*. Addison-Wesley Professional, 1st edition, October 2010.

For more information contact:

Principal Investigator:
Patrick McCormick
Los Alamos National Laboratory
Phone: 505-665-0201
Email: pat@lanl.gov

Program Manager:
Dr. Lucy Nowell
Advanced Scientific Computing Research Office
Phone: 301-903-3191
Email: lucy.nowell@science.doe.gov