Exploring the Construction of a Domain-Aware Toolchain for High-Performance Computing

Patrick McCormick, Christine Sweeney, Nick Moss, Dean Prichard,
Samuel K. Gutierrez, Kei Davis, Jamaludin Mohd-Yusof
Los Alamos National Laboratory
Los Alamos, NM, USA

***Abstract*—The push towards exascale computing has sparked a new set of explorations for providing new productive programming environments. While many efforts are focusing on the design and development of domain-specific languages (DSLs), few have addressed the need for providing a fully domain-aware toolchain. Without such domain awareness critical features for achieving acceptance and adoption, such as debugger support, pose a long-term risk to the overall success of the DSL approach. In this paper we explore the use of language extensions to design and implement the *Scout* DSL and a supporting toolchain infrastructure. We highlight how language features and the software design methodologies used within the toolchain play a significant role in providing a suitable environment for DSL development.**

Keywords*—*Domain Specific Language, Compiler, Debugging, LLVM, GPU, High-Performance Computing

I. Introduction

The challenges of productively programming the exascale generation of high-performance computing systems has led to a groundswell of research activities in new programming models, languages, and supporting runtime systems. Successful solutions will be required to effectively exploit various hardware developments including increasing chip-level parallelism, higher levels of available concurrency, heterogeneous chip and architecture designs, and increasingly complex memory hierarchies. The taxonomy of solutions to these technical challenges is often presented as a dichotomy: namely, as *evolutionary* or *revolutionary* approaches [1]. The evolutionary path favors leaving today's applications relatively untouched while modifying the underlying infrastructure to address the challenges. In contrast, the revolutionary path assumes the freedom to develop new, potentially radically different approaches that primarily seek to improve a combination of developer productivity, application performance, and portability. These two categories represent diametrically opposing viewpoints that, not surprisingly, highlight choices that must be made and challenges that must be overcome to leverage existing infrastructure investments for high-performance software and forge techniques to develop future software.

In this paper we focus on the role that *domain-specific languages* (DSLs) and their supporting infrastructures play within this taxonomy. Specifically, we describe our approach to designing and creating a proof-of-concept implementation of the *Scout* DSL and a supporting domain-aware toolchain that seeks to reduce the risks associated with adopting a revolutionary approach. Our approach aims to build a DSL in such a way that we can maintain the benefits of a general-purpose toolchain but also maintain a domain awareness within the entire toolchain. Scout is a strict superset of the C and C++ languages and extends the general-purpose toolchain to maintain this domain context throughout the compilation process. This step is critical to enable support for a productive and *complete*, domain-aware environment for developing, debugging, and profiling applications.

A. Domain-Specific Languages

Although domain-specific languages have only recently become popular in the high-performance computing (HPC) research community, they have been a common part of computing for decades [2]. A DSL is typically thought of as *a programming language of reduced expressiveness that is targeted at developers in a specific, focused problem domain.* For the purposes of this paper we will use the common distinction between *stand-alone* and *embedded* (or *internal*) DSLs. A stand-alone DSL is implemented as an entirely new language with custom syntax and a supporting parser, interpreter or compiler, supporting runtime system, and ideally, a debugger tailored to the DSL. In contrast, an *embedded* DSL is directly implemented in the syntax of an existing language, in which case the mechanisms of the host language (e.g. functions, overloaded operators, metaprogramming, type system, macros, etc.) are used to provide domain-centric functionality, and to a limited extent special syntax. Outside of HPC this technique has become increasingly common with the design, implementation, and growing popularity of languages such as Ruby, Groovy, and Scala [3] and has had a longer history in languages such as Lisp [4], [5] and Haskell [6].

Why use a DSL rather than simply taking the approach of combining a general-purpose language with an application-centric library (e.g. API) to provide DSL-like functionalities? At a high level, the use of domain-specific classes, methods, and functions—in effect a library or libraries—would appear to meet the same goals of abstraction, productivity, and portability. This question has already been considered from seemingly all angles and is well addressed in the literature. To quote Mernik et al. [7],

1) Appropriate or established domain-specific notations are usually beyond the limited user-definable operator notation offered by general-purpose languages. A DSL offers appropriate domain-specific notations from the start.
2) Appropriate *domain-specific constructs and abstractions* cannot always be mapped in a straightforward way to functions or objects that can be put in a library. (Traversals

and error handling are cited as common examples [8], [9].)

3) Use of a DSL offers possibilities for analysis, verification, optimization, parallelization, and transformation in terms of DSL constructs that would be much harder or unfeasible if a general-purpose language had been used because the source code patterns involved are too complex or not well defined.

In short, DSLs *"offer domain-specificity in better ways"* [7]. We add a fourth point:

4) Use of a DSL offers the *possibility* of debugging with reference to the DSL syntax and constructs. If thoughtfully designed, the DSL constructs themselves can enhance, rather than complicate, the overall debugging process.

As will be discussed, the way in which a DSL is implemented critically affects the ability to debug programs written in it.

The many advantages of DSLs have led to increased attention to the concept in the HPC-focused research community. In particular, because they are specialized they can provide improved productivity, maintainability, and portability [10], [11], as well as supporting validation and domain-aware optimizations [12], [13], and thus improved reliability and performance. These attributes could provide significant advantages for addressing the challenges of programming the next several generations of high-performance computing systems. However, DSLs also have significant risks in terms of the costs associated with their design, implementation, maintenance/longevity, and developer education, all of which can have significant implications for their successful adoption.

We seek to reduce some of the costs and risks of a fully-custom stand-alone DSL by introducing domain-centric *conservative extensions* [14] to the C and C++ programming languages, thus allowing us to leverage an existing implementation infrastructure while retaining the benefits of a stand-alone DSL *and* of a general-purpose programming language. In particular, we focus on the capabilities this enables in terms of providing a *domain-aware* software development toolchain.

B. Toolchain Support

Given the challenges and complexity of language and compiler design and implementation, stand-alone DSL activities have rarely considered the challenges of supporting *domain awareness* throughout the full toolchain (i.e., compiler, linker, debugger, profiler, and interfaces to HPC-centric system software and operating system features). In the HPC community this is manifested by the common use of *source-to-source* code generation or transformation in which the primary focus is on reducing the complexity and overall costs of development by leveraging an existing, unmodified toolchain. We posit that this approach is valuable for exploration but in the long term is likely to result in a lack of adoption. Specifically, we claim that to maximize overall developer productivity the abstractions provided by a DSL must persist throughout the toolchain, particularly for optimization and debugging. Furthermore, and most importantly, the loss of domain-specific knowledge is inherent in source-to-source compilation techniques that (by design) split the toolchain into disjoint infrastructures. In addition to the loss of context, this disjoint nature can lead to unexpected/undesirable results when the target compiler, lacking knowledge of domain-specific semantics, cannot perform otherwise even straightforward optimizations.

The design and modularity across the tools that make up the toolchain have significant implications for both the flexibility and extensibility that are needed for developing and supporting a domain-specific infrastructure. These considerations played a central role in our choice to use extensions to existing languages. As such it has driven our exploration to adopt the LLVM Compiler Infrastructure [15], [16]. We further discuss the advantages and disadvantages of this choice in Section III-B2. In addition, the higher levels of abstraction in DSLs require considerations in terms of the toolchain's interface to the underlying system's features and software layers. In this paper these layers of the toolchain include the DSL-specific supporting runtime environment.

C. Supporting Software Layers

Programming language implementations generally have supporting software interposed between the nominal executable (the compiled program) and underlying system-level services (such as parallel I/O or communication facilities), the operating system, or bare hardware. These may be ordinary libraries, such as *libc* for the C language, or may be more directly involved with the execution of programs, such as an interpreter, garbage collector, or virtual machine. We refer to these supporting software infrastructure components, individually or collectively, as run-time support, or just *runtimes*.

The highest-level abstractions supported by a domain-specific language, combined with the complexities of a supporting toolchain and the nuances of achieving high performance on the underlying architecture, make the capabilities and features of the runtime infrastructure of significant importance. In general this layer (or layers) must provide an abstract model of computation that provides a performance portability layer that is also a suitable target for effective and efficient code generation. When combined together with the various challenges posed by future HPC systems, these software layers will play a critical role in the overall success of a DSL.

II. Scout Language Overview

Scout, the domain-specific language used in our studies, is based on a set of domain-centric extensions to the C/C++ family of languages. These extensions introduce higher-level (more abstract) data structures and parallel constructs that can be reasoned about during the compilation process (i.e., from the front-end to back-end stages) and within the overall toolchain. In particular, we focus on supporting a debugging infrastructure that both preserves the domain context used in the source program and supports the use of the language extensions to facilitate debugging. This section introduces the basic syntax and semantics introduced by these extensions.

A. Scout Data Types

Scout supports a common paradigm in computational science by implementing a class of *mesh* abstract data types as fundamental (first-class) concrete data types. These mesh types include the programmer-specified *fields* that are stored as data at various locations in the topology of the mesh. For example,

```
% uniform mesh MyUniformMesh {
%  // Define the fields stored on the mesh.
%   cells    : float  temperature;
%   vertices: float3 velocity;
%   edges    : float3 flux;
% };

uniform mesh MyUniformMesh {
 // Define the fields stored on the mesh.
 cells    : float  pressure, temperature;
 vertices: float3 vorticity;
 edges    : float3 velocity;
};
```

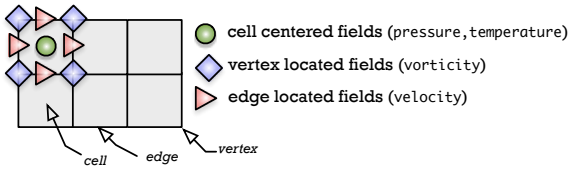


Fig. 1: ***The declaration of a mesh type and the locations of cell-, vertex-, and edge-located field data. The graphic shows the location of these fields in a two-dimensional instance of the mesh type.***

the code snippet shown in Figure 1 declares a uniform mesh with temperature and pressure stored at the cell centers, vorticity stored as a three-component floating point vector at the vertices, and velocity (also a three-component float vector) stored at the edges of the mesh. For clarity, Figure 1 also gives a pictorial representation of the field locations in a mesh instance. This generalizes to three-dimensional mesh constructs by introducing `faces` as a fourth field storage location in the mesh declaration.

It is important to note that this construct only declares the mesh *type*, including the fields of the mesh; it does not define an actual instance of the mesh. Unlike structures in C/C++ the programmer may make no assumptions about how the fields of a mesh will eventually be laid out in memory, or even what memories it might occupy (e.g., CPU, GPU, or distributed memory). To avoid overly complicated data layout decisions and code analysis issues related to aliasing, C-style pointers are not presently allowed within the fields of a mesh, or in members of any aggregate types and the types they contain. Once the mesh type and fields are defined, a mesh instance is declared by using the defined type and providing the dimensions (numbers of cells) of the mesh axes. Figure 2 shows the extended syntax used for these declarations.

```
// Define a two-dimensional uniform mesh with 3 cells
// along the x-axis and 2 cells along the y-axis.
MyUniformMesh umesh[3,2];
```

Fig. 2: ***Definition of a two-dimensional uniform mesh using mesh type and field information from Figure 1.***

To simplify program analysis for code generation and optimization several restrictions are placed on how meshes may be accessed and how assignments may be made to mesh fields. Following the abstract data type model, fields may only be accessed via mesh-centric operations and within those constructs field data is immutable, thus enforcing static single assignment (SSA) constraints within those code regions.

Scout includes other data types that are not in standard C/C++: 2, 3, 4, and 8-wide vector types (used above) for each of the built-in scalar data types (e.g. `short`, `int`, `double`, etc.). Each of these types supports member-wise subscripting and named accesses (like OpenCL, e.g., `vector.xyz`) and assignment, and unary, arithmetic, relational, and comparison operators.[1] Additional data types for representing both *on-* and *off-screen* image data are discussed in Section II-B3.

B. Scout Parallel Constructs

To address scaling and increasing core counts the language exposes two classes of parallelism, namely data- and task-parallelism. This section discusses the fundamental syntax and semantics, including restrictions, for both.

1) Data-Parallel Operations: Computation over mesh elements is implemented via a `forall` statement that processes a given set of locations (e.g. cells, vertices, faces). By definition, the programmer may make no assumptions regarding the order of execution across these locations—as implied by the `forall` keyword this is (potentially) a data-parallel operation. Each element that is processed is identified within the `forall` body by an explicitly named, locally-scoped variable as part of the `forall` syntax. The individual fields stored within the element may be accessed through this variable. For example, Figure 3 shows both the `forall` syntax and how it is possible to use these named variables to operate over connected mesh elements in a nested fashion. The language places a restriction on the operations allowed within nested `forall` constructs: the outermost construct can read and write mesh components while inner constructs have only read access.

```
// For all cells 'c' of the mesh 'umesh'
forall cells c in umesh { // 'c' -> active cell
  ...
  forall vertices v in c { // 'v' -> active vertex
    // vertex values are read-only, cell values
    // are read/write-able...
    ...
  }
}
```

Fig. 3: ***The `forall` construct provides a data-parallel model of execution over the given set of locations within the mesh topology.***

2) Task-Parallel Operations: Tasks in the language are represented by decorating a C-style function with the `task` type modifier. These *task functions* (or *tasks*) must be pure and side-effect free: they must always return the same result when provided the same input values, cannot modify or depend on hidden and/or global state, cannot contain statically declared variables, and cannot access input/output devices/streams. In addition, to avoid aliasing issues and to support the potential of distributed memories, any C/C++ types used as parameters to tasks must currently maintain purity. Specifically, the compiler presently enforces their being passed by value. Figure 4 shows the basic syntax for a task function and that individual tasks are launched using the standard C/C++ syntax for a function invocation.

The (partial) order of task execution is constrained only by the data dependencies between tasks as established by program

[1]These vector types and operations are easily supported and extended by the Clang and LLVM infrastructure.

```
// Define a task that operates over the given mesh.
// This function must be pure (side-effect free).
task void MyTask(MyMesh &m) {
  // body of task...
}
...
MyTask(m); // Invoke the task on the mesh.
```

Fig. 4: ***The `task` modifier specifies that a C-style function is pure and can therefore be executed in a task-parallel fashion with other task functions within the program.***

order of invocation. A hierarchical structure of tasks may be constructed by having task functions invoke further tasks. The most significant aspects of this capability are handled by the runtime as discussed in Section III.

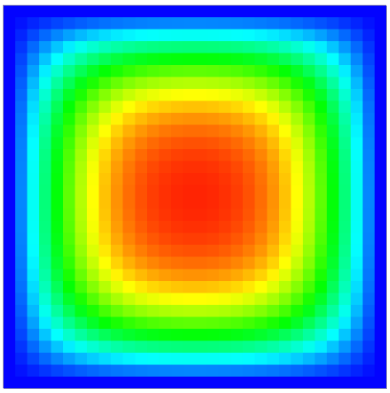
3) Visualization Constructs: Complementing the computational operations over meshes, the language also provides constructs for the *in-situ* visualization of the mesh elements. These operations are supported by the `renderall` statement that operates in the same fashion as the `forall` statement over the specified elements of a mesh. The `renderall` statement imposes the same order of execution restrictions as `forall`. In addition, all `renderall` statements must assign a value to the implicitly declared variable `color` that represents a four-channel floating point color value in the RGB (red, green, blue) color space with the fourth channel representing an opacity/transparency component. This color is assigned to the visual representation of the current mesh element that is being processed. Other than `color`, only data values declared within the scope of the `renderall` body may be assigned a value. As shown below, the `renderall` statement requires a *rendering target* that can be a type of either a `window` (*on-screen*) or `image` (*off-screen* file-based image). Each time the `renderall` statement is executed it will save the resulting image to the designated target. Because of the input/output nature of these targets they are only allowed to be used as local variables within the body of a task. The code snippet shown in Figure 5 presents an example image and the basic `renderall` syntax.

The mesh data types and language constructs described in this section all rely on layers of supporting software of varying degrees of complexity. The current runtime layers all represent works in progress and we are actively exploring the trade-offs and implications that the design of these layers have on supporting the overall language features, code generation, and the flexibility of the overall toolchain.

The remainder of this paper focuses on the supporting software layers for these extensions as well as the steps we take to retain and utilize their provenance throughout the compilation process. With this information propagated through the stack, it is then possible to harness a general-purpose debugging infrastructure to provide domain-specific functionality.

III. Runtime

The runtime layers associated with the Scout language provide a number of supporting features. In particular, they support visualization, graphics and windowing system operations, initializing and utilizing GPU resources, and managing



```
extern const float MAX_TEMPERATURE;
...
// Create a 512x512 window for displaying mesh
// elements.
window win[512,512];
...
// Render the cells to the window.  'color'
// must be assigned to within the loop body.
// This assigns a color to the 'active' cell.
renderall cells c in umesh to win {
  float norm_temp = c.temperature / MAX_TEMPERATURE;
  // Use the HSV (hue, saturation,value) colorspace
  // to assign a color from blue (cold) to red (hot)
  // for the cell.
  color = hsv(240.0 - 240.0 * norm_temp, 1.0, 1.0);
}
```

Fig. 5: ***The `renderall` construct supports a data-parallel model for producing* in-situ *visualizations of the given elements of the mesh topology. The rendered image shows the temperatures of cells from a simple heat transfer computation where the temperature increases from blue to red.***

the scheduling of tasks and data movement within the system. The vast majority of the runtime components are implemented in C++ but we have created a C API-based layer of abstraction on top of these C++ components. The design of this layer considers the language design with an overall goal of simplifying code generation. In this sense, our approach is more complex than source-to-source implementations. However, our path affords us the modularity (front-end independence) gained by using a common intermediate representation. We discuss this aspect in greater detail in Section III-B2. In this section we briefly discuss the runtime layer for supporting multiple GPU architectures and then in greater detail discuss the runtime support for task- and data-parallelism and execution in a distributed memory environment. Other runtime aspects, especially visualization and graphics support, are still in very early design stages and are beyond the scope of this paper.

A. GPU Architecture Support

One of the goals in the design of the toolchain is to support a single source to multiple architectures code generation path. In the current implementation we support the generation of data-parallel code for the `forall` and `renderall` statements on multiple GPU architectures. Specifically, we support NVIDIA and AMD GPUs via the CUDA and OpenCL APIs for moving data between the host and discrete GPU memories and for launching computations. The differences between these two APIs are abstracted by a common runtime interface to simplify code generation. The code generation for the `forall`

and `renderall` constructs occurs at the intermediate representation level in the compiler and is discussed further in Section III-B2.

B. Distributed Memory and Parallelism

To simplify the process of programming complex, large-scale distributed-memory architectures from any language or library-based interface it is clearly critical to have a powerful and well-designed supporting software layer. In addition to meeting key performance criteria, our interests in this layer are support for locality and data movement within the memory hierarchy and across a distributed memory system, data- and task-parallelism, and efficient and flexible scheduling of hardware resources. As ever, we seek to simplify the process of compiler-driven code generation. In short, the choice of avoiding a source-to-source path is important but the design of the runtime interface can significantly complicate the code generation stage of compilation. For example, dealing with C++ name mangling and/or a heavily template-based interface can be cumbersome and error-prone.

1) The Legion Runtime System: Recently the Legion runtime system [17], [18] was integrated into Scout. Legion is an open-source, data-centric programming model and runtime system for writing portable parallel and distributed programs. Legion was chosen as Scout's runtime for several reasons, including

- Ability to expose high levels of concurrency; targets distributed, heterogeneous hardware architectures; and makes effective use of complex memory hierarchies;
- Provides expressive abstractions that insulate users (i.e., application developers and automated code generators) from physical data layout considerations, data movement, task placement, and hardware characteristics (i.e., its interface is well-suited for automated code generation);
- Has demonstrated the ability to run effectively across a diverse set of large-scale HPC systems; and,
- Provides data and concurrency models that are consistent with the Scout programming model and semantics.

The design of Legion can be understood in terms of three concepts: *tasks* for both task-level and data-parallel concurrency, *logical data regions* (logical regions), and *physical data regions* (physical regions). Tasks are simply C++ functions that have been registered with the Legion runtime. Tasks must be pure (side-effect free) in the sense that they only read from or write to data regions passed to them by the runtime on their invocation. In addition to C++ support, tasks may also be wrappers around CUDA-based GPU kernels.

A logical region is a data region that is created by the runtime on behalf of the application. Structurally it is either a set, or a one-, two-, or three-dimensional array of data *points*, with each point in a region having a uniform set of *fields* with each field of specified size (in bytes). Conceptually it is isomorphic to a set or array of C structures, or colloquially of plain old data structures. Logical regions are a machine-independent abstraction for describing sets of data that can be used by tasks.

Task invocation entails specifying what parts (indexes, fields) of which logical regions the invoked task should have access, together with specific access privileges (read, write, read/write, etc.) and coherence requirements (for the case of data being shared among tasks), all on a per-region, per-field basis. The runtime then creates, for each logical region, a physical region—in effect a handle to the specified parts of the logical region—that is passed to the invoked task and remains valid only for the duration of its execution. Through these physical regions, or "views," the task may read or write data in the logical regions via runtime-provided accessors.

Because the Legion runtime knows the program order of task invocation and has precise information about what data, permissions, and coherence each task invocation requires, and tasks have no side effects unknown to the runtime, the runtime can infer the actual temporal data dependencies between tasks and relax serial program ordering to achieve task-level parallelism constrained only by those data dependencies and the availability of compute resources. On a distributed system this allows the runtime to make various scheduling and data-motion optimizations such as temporarily replicating read-only data on remote nodes.

Legion also provides a task launching mechanism to simplify task invocation for data parallelism. This requires a specification of a partitioning of one or more logical regions, and a task that will be multiply launched, once for each element of the element of the partition. When sufficient compute resources are available these tasks can be fully parallelized. Legion also provides a mechanism to specify that such a set of tasks *must* run in parallel to avoid deadlock because of inter-task synchronization, in which case the system will gracefully abort if resources cannot be made available.

2) Legion/Scout C Interface: The Legion/Scout C Interface (*lsci*) is the bridging interface between Scout's runtime library and Legion's C++ interface. It exposes a thin, specialized C interface that provides high-level abstractions to create, partition, update, and destroy Legion regions via Scout's domain abstractions (e.g., uniform meshes). The impetus behind providing such an interface is twofold. First, generating and interfacing to C code is more convenient than to C++ with Scout's compiler infrastructure. Second, and more importantly, the Scout runtime library is insulated from low-level Legion details. That is, the Scout runtime library never directly manipulates low-level Legion constructs (e.g., logical regions), but rather manipulates the higher-level, domain-centric constructs

```
lsci_unimesh_t umesh;
// Create a 3x2 2D uniform mesh.
lsci_unimesh_create(&umesh, 3, 2, 1, ctx, rt);
// Add a cell-centered temperature field to the mesh.
lsci_unimesh_add_field(&umesh, LSCI_FIELD_CELL,
                       LSCI_TYPE_FLOAT,
                       "temperature", ctx, rt);
// Evenly partition mesh into 2 sub-domains.
lsci_unimesh_partition(&umesh, 2, ctx, rt);
// Call generated routine responsible for
// setting umesh's initial conditions.
set_initial_conds(&umesh, ctx, rt);
```

Fig. 6: ***Example* lsci *runtime library calls similar to what would be generated by the Scout compiler to create and manipulate a uniform mesh.***

provided by *lsci* that match the Scout model. Figure 6 provides an example.

IV. Compiler

Our toolchain is based on the LLVM compiler infrastructure. Specifically this includes the Clang C/C++ front-end [19], the LLDB debugger [20], and the LLVM back-end [15]. Together these packages represent a well-designed, modular toolchain that has been widely adopted by industry and is also growing in popularity in the academic and research communities. This modularity, in concert with a per-component, library-based design, represents what we consider to be the fundamental strength of this infrastructure, especially in comparison to more monolithic, stand-alone compiler designs. This section provides brief overviews of the Clang and LLVM components and describes the overall approach taken in both employing and extending them to support Scout and to preserve domain-awareness throughout the toolchain. We provide details on supporting debugging with LLDB in Section V. For reference, Figure 7 provides a high-level diagram of the compiler stages discussed in the remainder of the paper.

A. Parsing and Semantic Analysis

One of our key design decisions was to express language features as *first-class* constructs rather than representing them using existing constructs. While this provides useful syntactic distinctions, it also plays a significant role in avoiding confusion, conflicts, and ambiguities with the semantics of C/C++. We have extended Clang to recognize our keywords, enforce our grammar rules and semantic restrictions, and store our own unique nodes in the abstract syntax tree (AST). Although this sounds daunting, we achieve this and reuse the vast majority of the Clang infrastructure for handling lexical analysis, parsing, and AST operations with little or no modification. Not surprisingly, this also enables us to maintain consistent information (e.g., line information, shared symbols) across both the base C/C++ languages and Scout's extensions.

After the parsing stage the semantic analysis of the constructs described in Section II follow in a very similar pattern in utilizing the Clang infrastructure. More specifically, this stage uses Clang's *visitor* pattern to add additional checks that validate the rules for each of these constructs. Once again, these operate in concert with those used for both the C and C++ languages. Our development environment automatically and regularly checks that our compiler passes Clang's own regression tests as well as Scout-specific ones. Once parsing and semantic analysis are completed the preservation of the domain context must be handled as the AST is lowered to the LLVM intermediate representation (IR) [21].

B. Lowering and Domain Preservation

The preservation of Scout's data types and constructs has two possible implementation paths. The first is to extend the LLVM IR language to support Scout's data types and statements. While this path would preserve the domain information, it suffers a significant disadvantage of wiring a DSL-centric intermediate representation into LLVM and forcing a significant number of changes throughout the infrastructure. In fact, the LLVM developers have provided a thoughtful warning against extending LLVM in this manner.[2] Overall, this path would not only introduce a significant amount of effort but also violate the important design decision of maintaining a significant level of both language (front-end) and architecture independence in the toolchain.

Fortunately, LLVM provides an alternative path in the implementation of the IR that helps maintain this independence and also reduces the impact on the overall code base. This path involves the use of LLVM's metadata capability to store information directly within the IR [22]. This information may then be used by metadata-centric analysis, optimization, and code generation passes. For example, the mesh data types represented within the AST are lowered to the IR by first converting them into LLVM's aggregate *structure* type and then providing a table of metadata information to capture the original properties of the mesh (e.g., fields and their locations within the mesh). Figure 8 shows the completed lowering of the mesh information given in Figures 1 and 2. Metadata entries, identified by an exclamation point followed by an integral value (e.g. `!0`), can be named, store typed LLVM values, and reference other metadata entries. This capability is flexible enough to build (simple) custom data structures within the IR. Importantly, this metadata is only visible to those parts of the infrastructure that are aware of its presence—in other words, the entire standard functionality of LLVM remains unaffected by the presence of Scout's metadata.

We leverage LLVM's metadata capabilities throughout the process of lowering the AST to the intermediate language as well as within the code analysis and generation stages. In particular, this allows us to annotate `task` functions and `forall` and `renderall` constructs with enough detail that we can process them using the LLVM Pass Framework [23]. These details play a central role in transforming DSL-level constructs into supporting runtime calls and finally into a debugger-friendly executable form.

C. Code Generation

Code generation tasks begin with the metadata-enhanced intermediate form that was lowered from the AST representation. The IR can simply pass through the default LLVM components to generate a sequential version of the source code that can be targeted to many of the back-end code generators available for LLVM (x86, ARM, PowerPC, etc.). For parallel execution of the input program the compiler currently supports two paths: (1) transformations of the `forall` and `renderall` constructs into GPU kernels, and (2) transformation of the code into a distributed memory version supported by the Legion runtime. At present we have yet to combine these two paths into a single code generation target. The remainder of this section provides an overview of the steps the compiler takes to generate code for each of these code paths.

1) GPU Kernel Generation: The basic data-parallel form supported by the `forall` construct maps easily to the execution model of GPUs. The details of GPU kernel generation show an additional approach for leveraging LLVM's metadata support, the pass infrastructure, and various other features. For both brevity and clarity we generalize our discussion as well

[2] See http://llvm.org/docs/ExtendingLLVM.html.

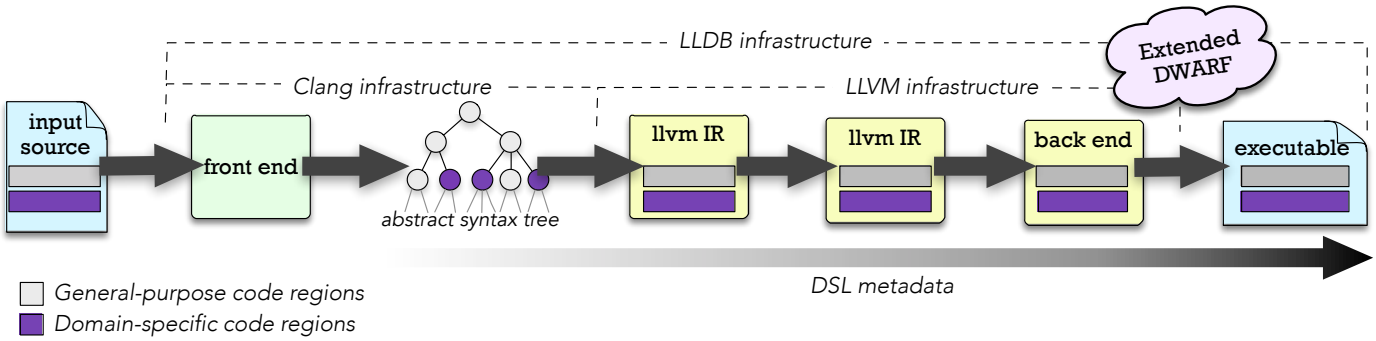


Fig. 7: ***The high-level stages of the compiler and the association with the various components within the LLVM infrastructure. The front end accepts mixed C/C++ and Scout language features and maintains the domain-centric metadata throughout the compilation process. This metadata is finally transformed into an extended DWARF format that is recognized by a modified version of the LLDB debugger.***

```
; Mesh lowered to aggregate structure type
%MyUniformMesh = type{float*, <3xfloat>*,<3xfloat>*, ... }
; Named metadata identifies each mesh type and the
; topology locations of fields (now members in the
; structure).
!0 = metadata !{metadata !"MyUniformMesh",
                metadata !"uniform", i32 2,
                metadata !"cells", metadata !1,
                metadata !"vertices", metadata !2,
                metadata !"edges", metadata !3}
!1 = metadata !{metadata !"float", metadata !"temperature"}
!2 = metadata !{metadata !"float3", metadata !"velocity"}
!3 = metadata !{metadata !"float3", metadata !"flux"}
```

Fig. 8: ***A sample of the LLVM intermediate language representation of a aggregate data type accompanied by metadata to maintain mesh type information. The arrows in the figure show the use of building a simple data structure to capture the topological location of fields within the mesh.***

as limit it to details associated with targeting NVIDIA's CUDA parallel computing platform.

In the current implementation GPU code generation is controlled solely by command line arguments that specify a target architecture. With this mode enabled, the compiler takes the following series of steps to transform a `forall` construct into a GPU-targeted kernel.

1) The `forall` body is lowered independently of the implied serial looping construct. In place of induction variables, a set of hardware-independent thread index variables are generated in the body. In a later stage of code generation these placeholders are replaced with CUDA-specific `threadIdx` values.
2) Next, the full set of instructions in the body of the `forall`, including the newly created thread variables, are extracted and inserted into a newly created function that represents the GPU kernel. This function is then registered using a metadata construct that marks it for further processing by a target-specific pass. In addition, a call to launch the kernel, as well as the data movement calls for moving data between host and GPU memory, are inserted at the location of the original `forall` statement.
3) The final transformation stage is handled by a target-specific *ForallPTX* pass that transforms the thread index values in the kernel and generates an in-lined character string version of the kernel in NVIDIA PTX form [24]. This can be accomplished using either the open-source NVPTX code generator in LLVM or the libNVVM API from NVIDIA [25]. The final form of the kernel and data movement calls are compatible with the CUDA Driver API [26].

The generation steps for AMD's processors (and the OpenCL runtime) are very similar in nature but somewhat more involved because of the need to create an Executable and Linking Format (ELF) version of the kernel. As mentioned in Section III, the runtime layer provides a set of common GPU calls for the initialization and launching of kernels, thereby simplifying code generation for the two supported GPU platforms.

2) Legion Code Generation: Given the high-level overview of Legion provided in Section III-B1, our mapping from Scout's data types are straightforward. Specifically, we transform a mesh variable into a logical region, and the fields of the mesh into fields of the logical region. These code generation steps are significantly simplified by the *lsci* runtime interface.

The more challenging aspect of the code generation process is related to the transformation of task functions into Legion tasks and the associated initialization and registration steps. The transformation of a task function requires the creation of an associated Legion entry point. This entry point is the actual Legion task and is responsible for unwrapping physical region data to recreate a mesh instance and also any non-Scout data types passed as function arguments. The final step is to then generate a call to the task function as defined by the programmer. This leaves the original task in a debugger-friendly form. This overall process is assisted by the presence of metadata in the IR that helps to distinguish task functions from non-task functions. In its current implementation the Legion runtime requires all tasks to be registered during initialization, that is, before the runtime has been started. To address this requirement we generate an associated initialization/registration function that is called at the start of program execution. It is also necessary for the code generator to replace all task function calls with Legion task launches. Readers interested in the details of the Legion interface are encouraged to visit the Legion tutorials page at http://legion.stanford.edu/tutorial.

V. Debugger Support

We regard debugging as having first-class importance, and the absence of a domain-aware infrastructure as a crippling shortcoming for the overall adoption of DSLs and in terms of delivering on the expectations of improved developer productivity. Much like developing a language and the supporting compiler infrastructure, writing a fully functional debugger from scratch is a daunting and expensive task. In source-to-source DSL implementations the transformation of the DSL to the target language strips all domain context and there are no direct methods to relate the executable back to the DSL source. While line number associations are possible, the debugger only operates in the context of the conventional language, leaving debugging details typically in the hands of the developer of the source-to-source translator and not the application programmer. Unfortunately the general trend in the community has been to leave debugging as a consideration of secondary importance.

By building on Clang and LLVM we are able to leverage the LLDB infrastructure to implement a debugger that does not suffer from the shortcomings of source-to-source approaches. The majority of this capability was afforded by the modular and library-based design of the Clang and LLVM components. In this case, the LLDB debugger directly builds on the capabilities provided by these libraries, and therefore also on the Scout DSL extensions. With this initial capability in place, a set of modifications and extensions is required to complete an initial domain-aware debugger. In the remainder of this section we briefly discuss these implementation details as well as the resulting capabilities they enable in the debugger.

A. Implementation

The implementation details of supporting the DSL within LLDB follow very similar paths to the steps taken to extend Clang and LLVM. As part of the extensions to Clang it was necessary to lower custom debugging information into the LLVM IR. In LLVM debugging information is actually represented as a special case of the metadata that was previously described. This metadata is lowered into the final executable using the DWARF format, a debugging file format for procedural languages [27]. In our case it was necessary to extend the back-end targets in LLVM to handle these steps. In particular, we took advantage of the extensibility of the DWARF format to capture the mesh and field type information.

With the DSL information embedded in the DWARF portion of the binary we then had to ensure that LLDB was able to process these sections and rebuild a Scout version of the Clang AST for use within the debugging session. The direct utilization of Clang as a collection of libraries provides a significant advantage for these steps. In particular, LLDB's ability to parse and *just-in-time* compile user-specified code during a debugging session provides tremendous savings in terms of reducing development time. In addition, it also enables DSL constructs to be directly utilized during a debugging session. In particular, the use of `renderall` statements can be directly used during debugging to build customized visualizations that aid in the debugging process. A example debugging session is shown in Figure 9, illustrating that not only is the debugger aware of the DSL constructs, but that these constructs can be entered and used directly for debugging purposes during the debugging session.

```
(lldb) expr {
  window win[512,512];
  renderall cells c in heat_mesh to win {
    float norm_h = h_now / MAX_TEMP;
    float hue = 240.0f - 240.0f * norm_h;
    color = hsv(hue, 1.0f, 1.0f);
  }
}
```

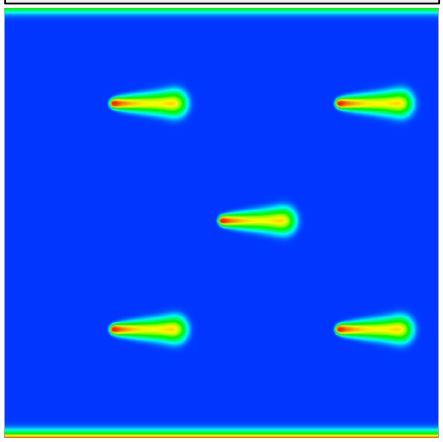


Fig. 9: ***Using DSL language constructs from directly within the debugger. Here the LLDB `expr` command is being used to evaluate a `renderall` expression to make a rendering of the cells of a flow-based heat transfer calculation.***

The steps we have taken to preserve domain-awareness in the debugging infrastructure are only the first of many. In particular the complexities involved in providing a scalable, distributed-memory-based, task- and data-parallel, and heterogeneous-architecture-aware debugger, that also maintains a fully domain-focused interface, will require a significant amount of additional effort.

VI. Related Work

While there is a significant amount of literature that covers a wide range of DSLs, we focus our attention on work that primarily focused on high-performance computing. While there are many emerging language activities, there are surprisingly few efforts in this space.

The most closely related language in concept to Scout is Liszt [28], [29], a DSL for programming unstructured mesh-based solvers. Liszt uses and manipulates the Scala language's abstract syntax tree to perform code transformations and optimizations and ultimately targets a custom runtime and source-to-source generated C++ and/or CUDA. This generated code is then handed off to a native system compiler to produce the final executable.

There are loose similarities with languages like Halide [30] that focus on the optimization of the image processing pipeline. The associated constructs share a similarity to the stencil computations used within scientific applications and therefore the language has potential outside of its original scope. In a similar

fashion to Scout, Halide utilizes the LLVM infrastructure for its final code generation stages but is based on a custom front-end and middle stages for the majority of analysis and optimization operations.

Finally, the Delite compiler framework and runtime is focused on enabling the rapid construction of high-performance DSLs [31], [32], [33]. While Delite focuses significantly on infrastructure, three languages have been designed that focus on machine learning (OptML) [34], data queries and transformations (OptiQL), and graph analysis (OptiGraph).

In comparison to DSLs, domain-specific libraries/frameworks for high-performance scientific computing are much more common. Although there are numerous efforts to consider, we cite OP2 and SIERRA as examples that are similar in spirit and use designs that motivate the use of mesh constructs and fields [35], [36]. In addition, OP2 also addresses portability of mesh-based applications on both CPU and GPU-accelerated architectures.

VII. Evaluation

In this section we briefly consider the challenges and benefits of our approach to designing and implementing the Scout toolchain. This is based entirely on our experiences and given that many significant items of work remain, it is not meant to be complete or conclusive in nature.

In terms of challenges, the development of a productive, full-featured, domain-focused programming environment for HPC applications will likely require a significant investment. Though this can vary based on approach and overall scope, costs are driven in large part by compiler development, details of the targeted architectures, complexities associated with the supporting runtime infrastructures, and the incorporation of developer-facing productivity tools (e.g. profilers and debuggers). Our adoption of the LLVM infrastructures provided a significant cost savings and a feature set that would have been impossible to reproduce in a reasonable amount of time. On the other hand, especially in comparison to source-to-source approaches, this approach carries with it the complexities of a full toolchain and the thus the associated learning curves. Finally, while DSLs provide some clear advantages for improving developer productivity and more effective code generation, there are many challenges to achieving both acceptance and adoption. The predominant concerns are the longevity of specialized toolchains and their interoperability with existing libraries and general-purpose programming languages such as Fortran.

While these are formidable challenges, our experience highlights some benefits of the approach. First, we have been able to write small mesh-based programs that require far fewer lines of code (hundreds as opposed to thousands) than would be needed otherwise. It is important to note that our efforts have been concentrated on enabling the toolchain and demonstrating feasibility as opposed to refining the choice of language features to address a specific set of needs for a particular problem domain. Furthermore, the foundation of a familiar programming language and a corresponding toolchain have been helpful for understanding the requirements of providing longevity within the very active LLVM community. Via the abstractions on top of the Legion runtime, we have been able to hide many of the complexities of data movement and provide initial support for task- and data-parallel forms of computation in distributed-memory environments. Although we do not have performance numbers ready for publication, gains seen in other Legion-based efforts show promise. As a first step our goals for providing a domain-aware toolchain have been realized and we are in a position to explore further capabilities based on the foundation discussed in this paper. While this initial step is critical to improving the acceptance of DSLs, the goal of adoption remains a challenge that will require further effort and strong collaboration with scientists from various application domains.

VIII. Conclusions

While we have not yet completed many significant steps in our exploration, our work to date has convinced us that the requirements of the entire toolchain should be considered when designing and developing a DSL. Care must be taken in the front-end design so that essential information is made available to the toolchain downstream. These requirements have many ramifications, including on language features, runtime capabilities supporting the programming model, exploiting hardware acceleration, and debugging. We have developed a solid and extensible basis for further exploration of a domain-aware toolchain, and can go forward with development with a deeper understanding of the benefits and challenges of this approach. We believe this approach can improve productivity on increasingly complex high-performance computing platforms by building in domain-specific language constructs and tools that simplify programming and by providing an underlying implementation that takes advantage of architectural features.

Acknowledgments

The majority of the work presented in this work is supported by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the United States Department of Energy, under the guidance of Dr. Lucy Nowell. Additional support was provided by the Department of Energy's National Security Administration, Advanced Simulation and Computing Program.

The authors would like to thank Mike Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken for their support and assistance with the Legion runtime. Additional thanks to colleagues at both AMD and NVIDIA for their support and efforts to make their architectures available via the LLVM infrastructure.

Los Alamos National Laboratory is operated by Los Alamos National Security LLC for the U.S. Department of Energy under contract DE-AC52-06NA25396.

References

[1] "DOE ASCR 2011 Exascale Programming Challenges Workshop Report," July 2011.

[2] J. Bentley, "Programming pearls: little languages," *Commun. ACM*, vol. 29, no. 8, pp. 711–721, Aug. 1986. [Online]. Available: http://doi.acm.org/10.1145/6424.315691

[3] D. Ghosh, *DSLs in Action*, 1st ed. Manning Publications, December 2010.

[4] L. Tratt, "Domain specific language implementation via compile-time meta-programming," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 6, pp. 31:1–31:40, Oct. 2008. [Online]. Available: http://doi.acm.org/10.1145/1391956.1391958

[5] D. Hoyte, *Let Over Lambda*, 1st ed. lulu.com, August 2010.

[6] P. Hudak, "Modular domain specific languages and tools," in *Software Reuse, 1998. Proceedings. Fifth International Conference on*, Jun 1998, pp. 134–142.

[7] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Comput. Surv.*, vol. 37, no. 4, pp. 316–344, Dec. 2005. [Online]. Available: http://doi.acm.org/10.1145/1118890.1118892

[8] D. Bonachea, K. Fisher, A. Rogers, and F. Smith, "Hancock: a language for processing very large-scale data," in *Proceedings of the 2nd conference on Domain-specific languages*, ser. DSL '99. New York, NY, USA: ACM, 1999, pp. 163–176. [Online]. Available: http://doi.acm.org/10.1145/331960.331981

[9] J. Gray and G. Karsai, "An examination of dsls for concisely representing model traversals and transformations," in *Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 9 - Volume 9*, ser. HICSS '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 325.1–. [Online]. Available: http://dl.acm.org/citation.cfm?id=820756.821831

[10] A. van Deursen and P. Klint, "Little languages: little maintenance," *Journal of Software Maintenance*, vol. 10, no. 2, pp. 75–92, Mar. 1998. [Online]. Available: http://dx.doi.org/10.1002/(SICI)1096-908X(199803/04)10:2⟨75::AID-SMR168⟩3.0.CO;2-5

[11] R. B. Kieburtz, L. McKinney, J. M. Bell, J. Hook, A. Kotov, J. Lewis, D. P. Oliva, T. Sheard, I. Smith, and L. Walton, "A software engineering experiment in software component generation," in *Proceedings of the 18th international conference on Software engineering*, ser. ICSE '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 542–552. [Online]. Available: http://dl.acm.org/citation.cfm?id=227726.227842

[12] D. Bruce, "What makes a good domain-specific language? APOSTLE, and its approach to parallel discrete event simulation," ser. DSL '97 – First ACM SIGPLAN Workshop on Domain-Specific Languages, in Association with POPL '97, 1997.

[13] V. Menon and K. Pingali, "A case for source-level transformations in matlab," in *Proceedings of the 2nd conference on Domain-specific languages*, ser. DSL '99. New York, NY, USA: ACM, 1999, pp. 53–65. [Online]. Available: http://doi.acm.org/10.1145/331960.331972

[14] M. Felleisen, "On the expressive power of programming languages," *Science of Computer Programming*, vol. 17, pp. 35–75, December 1991.

[15] "The LLVM Compiler Infrastructure Project," http://www.llvm.org, Aug. 2014.

[16] C. Lattner and V. Adve, "Llvm: a compilation framework for lifelong program analysis transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, March 2004, pp. 75–86.

[17] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 66:1–66:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389086

[18] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Structure slicing: Extending logical regions with fields," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Los Alamitos, CA, USA: IEEE Computer Society Press, 2014 – to appear.

[19] Clang, "Clang: A C Language Family Frontend for LLVM," http://clang.llvm.org, Aug. 2014.

[20] "The LLDB Debugger," http://lldb.llvm.org, Aug. 2014.

[21] LLVM, "LLVM Language Reference Manual," http://llvm.org/docs/LangRef.html, August 2014.

[22] D. Patel and C. Lattner, "Extensible Metadata in LLVM IR," http://blog.llvm.org/2010/04/extensible-metadata-in-llvm-ir.html, 4 2010.

[23] LLVM, "Writing an LLVM Pass," http://llvm.org/docs/WritingAnLLVMPass.html, August 2014.

[24] NVIDIA, "NVIDIA Parallel Thread Execution ISA Version 4.0," http://docs.nvidia.com/cuda/parallel-thread-execution/, August 2014.

[25] NVIDIA, "libNVVM API," http://docs.nvidia.com/cuda/libnvvm-api/index.html, August 2014.

[26] NVIDIA, "NVIDIA CUDA Driver API," http://docs.nvidia.com/cuda/cuda-driver-api/index.html, August 2014.

[27] M. J. Eager, "Introduction to the DWARF Debugging Format," *Group*, 2007.

[28] Z. DeVito and P. Hanrahan, "Designing the language liszt for building portable mesh-based pde solvers," in *SciDAC 2011 Conference*, July 2011.

[29] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. 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*, ser. SC '11, 2011, pp. 9:1–9:12.

[30] 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*, ser. PLDI '13. New York, NY, USA: ACM, 2013, pp. 519–530. [Online]. Available: http://doi.acm.org/10.1145/2491956.2462176

[31] K. Brown, A. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "A heterogeneous parallel framework for domain-specific languages," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, Oct 2011, pp. 89–100.

[32] T. Rompf, A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Odersky, and K. Olukotun, "Building-blocks for performance oriented dsls," in *DSL '11: IFIP Working Conference on Domain-Specific Languages*, 2011.

[33] A. Sujeeth, T. Rompf, K. Brown, H. Lee, H. Chafi, V. Popic, M. Wu, A. Prokopec, V. Jovanovic, M. Odersky, and K. Olukotun, "Composition and reuse with compiled domain-specific languages," in *ECOOP 2013 Object-Oriented Programming*, ser. Lecture Notes in Computer Science, G. Castagna, Ed. Springer Berlin Heidelberg, 2013, vol. 7920, pp. 52–78. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-39038-8_3

[34] A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Wu, A. R. Atreya, K. Olukotun, T. Rompf, and M. Odersky, "Optiml: an implicitly parallel domainspecific language for machine learning," in *in Proceedings of the 28th International Conference on Machine Learning, ser. ICML*, 2011.

[35] G. R. Mudalige, M. B. Giles, J. Thiyagalingam, I. Z. Reguly, C. Bertolli, P. H. J. Kelly, and A. E. Trefethen, "Design and initial performance of a high-level unstructured mesh framework on heterogeneous parallel systems," *Parallel Comput.*, vol. 39, no. 11, pp. 669–692, Nov. 2013. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2013.09.004

[36] J. R. Stewart and H. Edwards, "A framework approach for developing parallel adaptive multiphysics applications," *Finite Elements in Analysis and Design*, vol. 40, no. 12, pp. 1599 – 1617, 2004, the Fifteenth Annual Robert J. Melosh Competition. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0168874X04000186