Enhancing Domain Specific Language Implementations Through Ontology

Chunhua Liao[‡*], Pei-Hung Lin[‡], Daniel J. Quinlan[‡]
Yue Zhao [§], Xipeng Shen [§],
[‡]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory
[§]Computer Science Department, North Carolina State University
Email: [‡]{liao6, lin32, dquinlan}@llnl.gov, [§]{zhao30, xshen5}@ncsu.edu

ABSTRACT

Domain specific languages (DSLs) offer an attractive path to program large-scale, heterogeneous parallel computers since application developers can leverage high-level annotations defined by DSLs to efficiently express algorithms without being distracted by low-level hardware details. However, performance of DSL programs heavily relies on how well a DSL implementation, including compilers and runtime systems, can exploit knowledge across multiple layers of software/hardware environments for optimizations. The knowledge ranges from domain assumptions, high-level DSL semantics, to low-level hardware features. Traditionally, such knowledge is either implicitly assumed or represented using ad-hoc approaches, including narrative text, source-level annotations, or customized software and hardware specifications in high performance computing (HPC). The lack of a formal, uniform, extensible, reusable and scalable knowledge management approach is becoming a major obstacle to efficient DSLs implementations targeting fast-changing parallel architectures.

In this paper, we present a novel DSL implementation paradigm using an ontology-based knowledge base to formally and uniformly exploit the knowledge needed for optimizations. An ontology is a formal and explicit knowledge representation to describe concepts, properties, and individuals in a domain. During the past decades, a wide range of ontology standards and tools have been developed to help users capture, share, utilize and reason domain knowledge. Using modern ontology techniques, we design a knowledge base capturing concepts and properties of a problem domain, DSL programs, and hardware architectures. Compiler interfaces are also defined to allow interactions with the knowledge base to assist program analysis, optimization and code generation. Our preliminary evaluation using stencil computation shows the feasibility and benefits of our approach.

*Corresponding author

WOLFHPC2015, November 15-20 2015, Austin, TX, USA

ACM 978-1-4503-4016-8/15/11 ...$15.00
DOI: http://dx.doi.org/10.1145/2830018.2830022.

Categories and Subject Descriptors

D.3.m [**Software**]: Programming Languages—*Miscellaneous*; M.1 [**Knowledge Management**]: Knowledge Engineering Methodologies

Keywords

Ontology, Knowledge base, Domain-Specific Language, Compiler, High-Performance Computing

1. INTRODUCTION

Domain-specific languages (DSLs) [30] offer an attractive path to program large-scale, heterogeneous parallel computers since application developers can leverage a set of high-level notations and abstractions defined by DSLs to efficiently express high-level algorithms, without being distracted by low-level, fast-changing hardware details. The performance of DSL programs heavily relies on how well a DSL implementation can leverage a wide range of knowledge from multiple layers of the DSL software/hardware stack to enable optimizations. Example knowledge includes domain assumptions, high-level DSL semantics, low-level hardware features, and so on. High-level DSL semantics are often lost during DSL lowering (a compiler process of converting a DSL program to lower level representations) and extremely difficult or even impossible for a compiler to recover. For example, a DSL may define a high-level array abstraction which has the *non-aliasing* and *non-overlapping* semantics. Its implementation may choose to lower the array abstraction to a low-level C pointer type. It is very difficult for classic compiler analysis to figure out a pointer's aliasing and overlapping properties, which prevents a wide range of optimizations. Many other semantics are even more intractable for compilers to discover. For instance, a DSL data container storing objects may have the semantics of *unique* and *sorted*. No compiler analysis can detect such properties if the stored objects are not known at compile time.

Currently, a range of informal, ad-hoc methods are used to communicate semantics of an application domain and details of a machine architecture with compilers and runtime systems, including customized semantics specification files [14], memory specification language [6], and so on. These methods are not uniform, reusable and scalable. Formally, some studies used ontology [26], a formal domain knowledge specification based on description logic [3], to conduct domain analysis and language grammar design [27, 5]. However, these studies did not target high performance computing.

They also did not leverage ontology in DSL compiler implementations. As a result, there is still an urgent need for using a formal and holistic knowledge management approach for enhancing DSL implementations in high performance computing (HPC).

In this paper, we present a novel ontology-based approach to enhance DSL implementations tailored for HPC. We use modern ontology-based knowledge engineering techniques to formally and systematically capture, store, and utilize multiple layers of knowledge which is essential for effective DSL implementations. Both domain-specific semantics and HPC hardware features are explicitly represented by using the standard Web Ontology Language (OWL)[18], one of the most popular ontology languages. We also define a compilation framework interacting with an ontology-based knowledge base. Preliminary results using stencil computation show that our new ontology-driven DSL implementation paradigm can dramatically improve reusable domain knowledge accumulation and effectively guide code generation and optimizations.

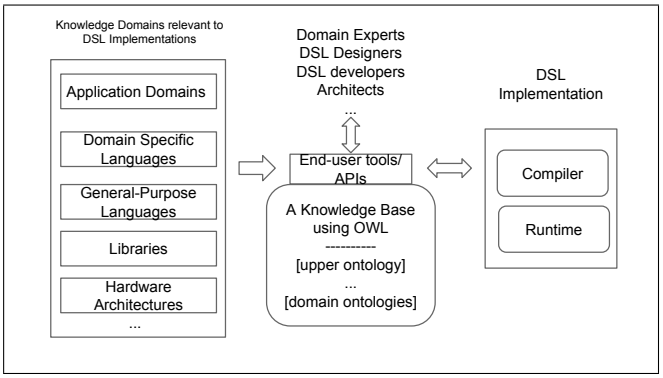

Figure 1: A new Ontology-Driven DSL Implementation Paradigm

2. METHODOLOGY

In this section, we give an overview of our ontology-driven DSL implementation paradigm. We then give detailed descriptions of key techniques, challenges and solutions of our approach.

2.1 Overview

As shown in Figure 1, our approach focuses on how to formally and systematically represent and utilize a range of domain knowledge which is relevant to enhance HPC DSL implementations. Our fundamental motivation is to apply modern ontology-based knowledge engineering techniques in the field of high performance computing in order to enable more optimizations.

The central component of our approach is an ontology-based knowledge base storing information representing common sense concepts (referred to as upper ontology), application domains (e.g. stencil computation), programs (e.g. DSL programs), libraries, hardware architectures, and so on. Through end-user tools and programmable APIs, the knowledge base can interact with human users (e.g. domain experts, DSL developers, programmers, architects) and software agents (e.g. compilers and runtime systems) to acquire knowledge and answer queries. Storing information across different domains of a DSL enables opportunities of bridging semantics gaps, i.e. communicating high-level domain semantics with low-level implementations so the compiler and runtime can better analyze and optimize DSL programs for a target hardware platform.

2.2 Ontology-Based Knowledge Base

It is a challenging task to manage the diverse knowledge needed for supporting DSL implementations targeting fast changing high performance computing architectures. The techniques used must be intuitive, flexible, scalable, user-friendly, efficient and mature. We choose an ontology-based knowledge base to store and utilize knowledge required for HPC DSLs.

Ontology-based knowledge bases have been gaining increasing popularity in multiple fields, including biology [2], ambient intelligence [24], and robotics [28], partially driven by the maturing ecosystem of semantic web movement led by the World Wide Web Consortium. An ontology [26] is a formal specification for explicitly representing knowledge about types, properties, and interrelationships of the entities in a domain. It provides a common vocabulary to represent and share domain concepts. The theory foundation of modern ontology techniques is Description Logics (DLs) [3], a family of logical languages for knowledge representation. DLs have several dialects with different expressiveness and different efficiencies for reasoning. Web Ontology Language (or OWL) [18, 19] is one of the most popular ontology languages.

As shown in Figure 2, OWL contains classes, properties, and individuals. *Classes* (or concepts) denote sets of individuals. Classes may be organized into a hierarchy via inheritance relations. Classes are denoted as nodes in an ontology's visulization graph. Person, Man, and Women are three classes in the example ontology modeling families. Two special classes are provided by default, owl:Thing and owl:Nothing. owl:Thing is the most general class and the superclass of any other class in the ontology. Class owl:Nothing is empty and is the subclass of every included class.

Properties (or relations, roles) specify binary relations between objects. Properties are shown as edges in ontology graphs. OWL defines two main kinds of properties: object properties and datatype properties. Object properties connect objects to other objects. For example, the hasWife property connects John and Mary. Datatype properties relate an object to datatype values. One example datatype property is hasAge, which connects a person to an integer value indicating the person's age. A property may have some restrictions (constraints), which describe existential or universal restrictions, a value type, allowed range of values, the number of the values (cardinality), and so on. It may also have characteristics such as transitive, symmetric, reflexive, etc. Similiar to classes, a property may have subproperties to form a property hierarchy (hasSpouse has sub properties: hasWife and hasHusband). An OWL query on a generic property will return all subproperties asserted.

Individuals (or instances) denote single individuals in the domain. In the example, John is an individual of the class Man.

OWL has different kinds of syntax for different purposes. We use a concise form, functional syntax, in this paper. As shown in Table 1, OWL, as a description logic language, is equipped with a formal (i.e. machine readable) semantics: a precise specification of the meaning of OWL ontologies. The

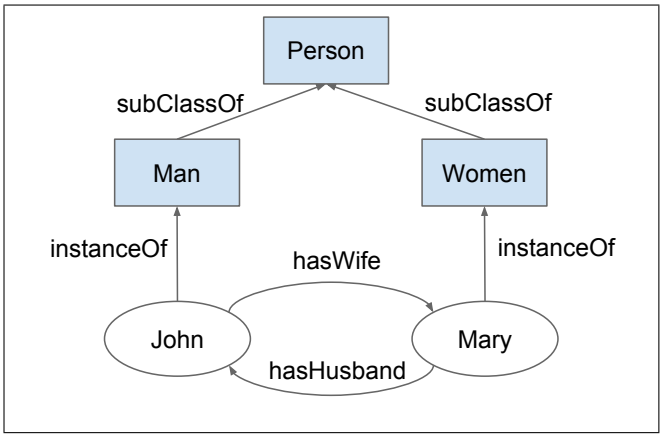

Figure 2: Example Ontology for the family domain

```
Prefix(:=<http://example.com/owl/families/>)
Ontology(<http://example.com/owl/families>
 Declaration( NamedIndividual( :John ) )
 Declaration( NamedIndividual( :Mary ) )
 Declaration( Class( :Person ) )
 Declaration( Class( :Woman ) )
 Declaration( ObjectProperty( :hasWife ) )
 Declaration( ObjectProperty( :hasSpouse) )
 SubClassOf( :Woman :Person )
 SubClassOf( :Man :Person )
 SubObjectPropertyOf( :hasWife :hasSpouse )
 ObjectPropertyAssertion( :hasWife :John :Mary )
 ... )
```

Figure 3: Family ontology in functional syntax

semantics of OWL are defined by interpreting concepts as sets of individuals and properties as sets of ordered pair of individuals. These individuals are typicallyed assumed from a given domain. Non-atomic concepts and properties have semantics which are defined based on semantics of atomic ones. Figure 3 shows a concrete example of using OWL 2's functional syntax to describe a family ontology. OWL requires that each entity in the ontology must have a unique string-based id, or internationalized resource identifier (IRI), in order to unambiguously refer to concepts, relations, and individuals. IRIs are defined in different namespaces to avoid name collision. A namespace can have a short alias called prefix.

Our choice of using OWL for representing DSL knowledge has several prominent benefits, including 1) leveraging existing knowledge engineering methodologies [8] and tools [11] to allow people of different backgrounds to collaboratively make domain knowledge explicit, 2) providing a common taxonomy and vocabulary to enable knowledge interoperability among multiple sources including human users and software agents, 3) providing knowledge reuse since the ontology provides a persistent knowledge base with standard formats with query and update interfaces [33, 12], 4) facilitating knowledge validation and generation using logic programming connected with reasoning/inference engines such as FaCT++ [29] and SWI-Prolog [34].

A knowledge base to support HPC DSL implementations must be comprehensive enough to cover sufficient information while efficient enough to handle updates and queries. We use a modular design to organize knowledge into different modules based on their layers in the DSL software/hardware stack. This design allows on-demand loading of relevant knowledge modules to effeciently serve a purpose. Stale information about hardware can also easily be discarded.

Generating high quality knowledge is another challenge considering the large scope to be covered in our paradigm. We use a hybrid approach combining both manual and automated processes to generate the contents of the knowledge base. The manual process invovles a team of people from different disciplines, including domain experts, DSL designers, programmers, and architects. Fundamental concepts and relations for different domains are established by the team. Instances are mostly automatically generated by tools (e.g. a compiler-based tool converting an input program into OWL instances). In addition, some existing ontologies [17, 28] already model some portion of the knowledge we are interested in. We simply directly import them into our knowledge base or cherry-pick relevant entities.

2.3 Compiler and Runtime Interface

The interface of an ontology-based knowledge base is essential for allowing interactions with DSL compilers, runtime systems, and tools, in order to bridge the semantics gap in DSL implementations. We discuss requirements, challenges, and solutions to enable productive, flexible and portable interactions.

The main requirement for the knowledge base's interface is that it must support bidirectional interaction, i.e. software agents not only passively query the knowledge base for information, but also actively update the knowledge base with new knowledge. The reason is that significant pieces of HPC knowledge are indeed not prior knowledge, but highly dependent on a particular execution instance of a program running on a given environment. Therefore, it is impossible to prepare a static, prebuilt, all-you-need knowledge base for HPC. In addition, the interface should also be fast and easy to deploy in HPC environments.

To meet the main requirement, we use SWI-Prolog[34] as the main interface between the knowledge base and DSL implementations. SWI-Prolog provides a semantic web library which can load OWL ontologies into memory and represent them as logic terms (predicates). Standard Prolog queries can be written to query and update the knowledge base. Moreover, SWI-Prolog has excellent language interoperability. It can work as a host language loading foreign C/C++ libraries. It can also be embedded in existing C and C++ programs to serve as a logical engine. These features allow the knowledge base to be dynamically connected to compilers and runtime systems written in general-purpose languages such as C and C++.

While Prolog provides powerful querying and reasoning capabilities to interact with an ontology knowledge base, some HPC environments may not able to provide Prolog for various reasons. Even if Prolog is provided, some simple and frequent queries do not need logic programming with unnecessary overhead caused by language interoperability. To alleviate these problems, we developed a light weight ontology parser and query library written in C++ to allow software agents to parse and query OWL files using familiar and convenient C++ function calls. Example C++ function calls include those loading OWL files, retrieving the number of CPUs for a machine, obtaining memory features of a GPUs and so on.

Additional support in the compilers and runtime systems is needed to help bridge the semantics gap in DSL implementations. Proper connections must be established between domain concepts, DSLs, and low-level host languages so knowl-

Functional Syntax	Formal Semantics	Natual Language Semantics
$Declaration(Class(CE))$	$(CE)^C \subseteq \Delta_I$	CE is a class within an object domain
$Declaration(NamedIndividual(a))$	$(a)^I \in \Delta_I$	a is an individual within an object domain
$Declaration(ObjectProperty(OPE))$	$(OPE)^{OP} \subseteq \Delta_I \times \Delta_I$	OPE is an object property connecting two objects
$subClassOf(CE_1\ CE_2)$	$(CE1)^C \subseteq (CE2)^C$	class CE_1 is a subclass of class CE_2
$ClassAssertion(CE\ a)$	$(a)^I \in (CE)^C$	individual a is an instance of class CE
$ObjectPropertyAssertion(OPE\ a_1\ a_2)$	$((a_1)^I, (a_2)^I) \in (OPE)^{OP}$	a_1 is related to a_2 via ObjectProperty OPE
$ObjectIntersectionOf(CE1\ ...\ CEn)$	$(CE_1)^C \cap ... \cap (CE_n)^C$	a class resulting from intersecting class CE_1 to CE_n

Table 1: OWL version 2.0: example functional syntax and semantics

edge from higher level entities can be associated with their corresponding low-level entities, and vice versa when needed. Some connections at higher levels (e.g. application domain and DSL programs) can be manually established by domain experts and DSL designers. For example, a stencil domain may have a concept of *stencil*. A concrete stencil DSL may have a construct named *MyStencil*. Then the *subclassOf* relation can be manually added by a DSL designer into the knowledge base. A later query of the DSL construct can automatically obtain the semantics defined at the domain level. Many connections also exist inside an implementation when it translates (or lowers) high-level constructs into lower level ones. One example is some high-level DSL array abstractions are translated into low-level primitive pointer types available in the host languages. Our solution is to extend a compiler to explicitly keep track of essential transformation steps and update the knowledge base with the correlation of associated program constructs.

```
1  #define DIM 2
2  int main(int argc, char* argv[])
3     {
4       Point lo, hi;
5    // Space discretization
6       Box bxdest(lo,hi);
7       Box bxsrc=bxdest.grow(1);
8  ...
9    // Source and destination data containers
10      RectMDArray<double,1> Asrc(bxsrc);
11      RectMDArray<double,1> Adest(bxdest);
12 ...
13      double ident, C0;
14   // Shift and Stencil declarations
15      array<Shift,DIM> shft_vec = getShiftVec();
16      Stencil<double> laplace = C0*(shft_vec^zero);
17   // Stencil formation using Shift
18      for (int dir=0;dir<DIM;dir++)
19         {
20           Point thishft = getUnitv(dir);
21           laplace = laplace + ident*(shft_vec^thishft);
22           laplace = laplace + ident*(shft_vec^(thishft*(-1)));
23         }
24    // Apply stencil computation using data containers in space
25      Stencil<double>::apply(laplace, Asrc, Adest, bxdest);
26    }
```

Figure 4: Laplacian example with Shift Calculus DSL

3. EVALUATION

In this section, we use a stencil DSL as an example to evaluate our ontology-based knowledge base used to enhance DSL implementations.

Stencil computation is widely used in many scientific applications for partial differential equations, finite element and finite difference methods. The computation often follows a regular pattern using nested loops to update points in a discretized space. A common stencil pattern references surrounding points in a 2D or 3D grid to update a center point.

3.1 Stencil DSL: Shift Calculus

AMR Shift Calculus [9] is a light-weight embedded DSL that provides a generalized abstraction to express stencil computation. This DSL relies on C++ as its host language and additionally leverages the Chombo library [7], a library suite for partial differential equations, to describe the spacial discretization. We use an example (shown in Fig. 4) for a two-dimensional 5-point stencil (a center point with four neighbors) to describe the DSL's syntax and semantics. The description about the AMR Shift Calculus is split into two parts: one presents the description for the spacial discretization and the other introduces the formation of the stencil.

The discretization of space is given as $(i_0, ..., i_D) = i \in \mathbb{Z}^D$. The Shift Calculus DSL uses a C++ class *Point* to represent the points in the rectangular lattice $\mathbb{Z}^D$. The Shift Calculus DSL uses the C++ class *Box* from the Chombo library to represent a rectangular region in $\mathbb{Z}^D$. A *Box* is defined by specifying the Points defining its low and high corners. An example is at line 6 in Fig. 4 where the *lo* and *hi* points define the *bxdest* Box. A Box can grow in all directions by a given size (shown in line 7). For example, B.grow(s) grows the Box B in all directions by a size s (s can be negative corresponding to shrinking). A C++ class $RectMDArray$ introduced by the DSL is a *real* type container. The memory size for a $RectMDArray$ is determined by a Box associated with it. If the *Adest* in Fig. 4 contains data for a N^2 two dimensional Box, the *Asrc* will contain a larger memory size for a $(N+2)^2$ Box after its growth.

The Shift Calculus DSL also describes the formation of a stencil. Two C++ template classes, $Shift$ and $Stencil$, are provided to describe the stencil used in the computation. The $Shift$ class has a data member with type of class $Point$. A default constructor sets a $Shift$ object that has an origin point in the multidimensional coordinates (represented as (0,0) in a 2D coordinate, or (0,0,0) in 3D.). In Fig. 4, an array of $Shift$ at line 15 stores the unit vectors in all directions in space. We refer to this array as $ShiftVector$ in this paper. The unit vectors are (1,0) and (0,1) in 2-dimensional coordinates. A special operator $\wedge$ is designed as a arithmetic operator for the $Shift$ class. The coordinate values of a $Point$ in all directions will be multiplied to the $Shift$ objects stored in the $ShiftVector$. Its defintion in C++ is in the following code snippet:

```
1 inline Shift operator^(array<Shift,DIM>, a_shiftvec, Point a_exp)
2 {
3   Shift ret;
4   for(int dir=0; dir < DIM; dir++)
5      ret = ret * Shift(a_shiftvec[dir].m_shift)*a_exp[dir]);
6   return ret;
7 }
```

A *Stencil* class contains essential information for a stencil with arbitrary shape and size. The information includes

```
1  int main(int argc,char *argv[])
2  {
3  ...
4    const class Point lo(zero);
5    const class Point hi = getOnes() * adjustedBlockSize;
6    const class Box bxdest(lo,hi);
7    const class Box bxsrc = bxdest .  grow (1);
8    class RectMDArray< double  , 1 , 1 , 1 > Asrc(bxsrc);
9    class RectMDArray< double  , 1 , 1 , 1 > Adest(bxdest);
10   const double ident = 1.0;
11   const double C0 = -6.00000;
12 ...
13   double *sourceDataPointer = Asrc . getPointer();
14   double *destinationDataPointer = Adest . getPointer();
15   for (k = lb2; k < ub2; ++k) {
16     for (j = lb1; j < ub1; ++j) {
17       for (i = lb0; i < ub0; ++i) {
18         destinationDataPointer[arraySize_X * (arraySize_Y * k + j) + i] =
                 sourceDataPointer[arraySize_X * (arraySize_Y * (k + -1) + j)
                 + i] + sourceDataPointer[arraySize_X * (arraySize_Y * (k + 1)
                  + j) + i] + sourceDataPointer[arraySize_X * (arraySize_Y * k
                  + (j + -1)) + i] + sourceDataPointer[arraySize_X * (
                 arraySize_Y * k + (j + 1)) + i] + sourceDataPointer[
                 arraySize_X * (arraySize_Y * k + j) + (i + -1)] +
                 sourceDataPointer[arraySize_X * (arraySize_Y * k + j) + (i +
                 1)] + sourceDataPointer[arraySize_X * (arraySize_Y * k + j) +
                  i] * -6.00000;
19       }
20     }
21   }
22 }
```

Figure 5: generated sequential C++ output for Laplacian example

all the coefficients and neighboring points (represented by the coordinates of a point using the class *Point* discussed earlier), and other parameters for adaptive mesh refinement (AMR). Arithmetic operators, such as +, ∗, +=, are applicable to form the stencil shape in a *Stencil* object. A *Stencil* object, *laplace*, at line 16 in Fig. 4 is initialized to contain the origin point with a coefficient *CO* associated with it. Line 18 to line 23 shows a loop that adds the adjacent neighboring points in every direction to the origin point into the *laplace* object. A fixed coefficient, *ident*, is applied to all these neighboring points. The final step (shown at line 25) is to apply the defined stencil to the source and target data containers to execute the computation.

We built a Shift Calculus DSL compiler based on the ROSE source-to-source compiler framework [22]. The compiler takes the source code written in the DSL and generates the source codes in C++. A vendor compiler, such as GCC or the Intel compiler, is then used to generate the final executable for a platform. Fig. 5 shows the sequential C++ output code generated from the example in Fig. 4. The generated code is not friendly to further optimizations. The reason is that the semantics of member function *RectMDArray<>::getPointer()* are not known to a compiler so the returned pointer (subsequently the assigned *sourceDataPointer* and *destinationDataPointer*) will be assumed to be able to point to any memory location. For example, dependency analysis will conservatively report that the two pointers may alias to each other and parallelization of the loop nests will not be activated.

3.2 Ontology for Stencil Computation

To represent and utilize semantics in DSL implementations, we use ontology to model key concepts and relations related to stencil computation. As a preliminary evaluation, we focus on a limited set of domains, including the Shift Calculus DSL, host language programs, and libraries. Hardware is also modeled to facilitate optimizations requiring hardware details. Figure 6 shows high-level concepts of the stencil computation ontology.

The DSL domain captures concepts and relations implicitly and explicitly included in the Shift Calculus DSL. Example concepts include *Stencil*, *Point*, *Grid*, *Coordinate*, *RectMDArray* and so on. As an embedded DSL built on top of a C++ library. We also model concepts and properties of the library types and interface functions.

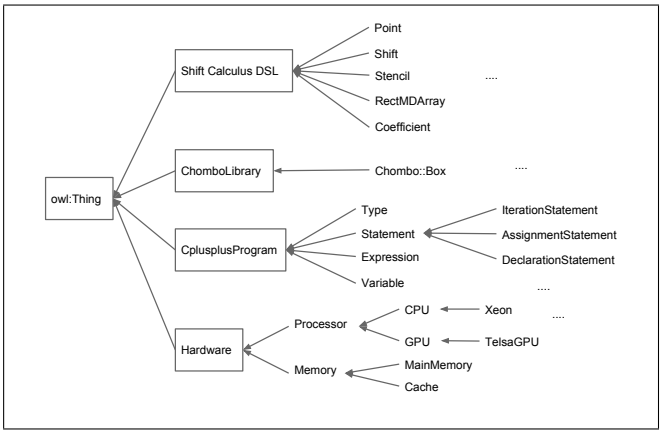

Figure 6: Partial top level concepts of the stencil computation ontology

The host program domain contains C++ program concepts and relations. Some top level concepts include *Expression*, *Statement*, *Variable*, *Type* and so on. Relations include direct source level connections among language constructs, such as *hasType*, *hasBody*, *hasScope*, *hasName*, *hasValue*, and so on. Many more interesting program construct relations, including *call* and *calledBy* for function call relation, *alias* for variable aliasing, *overlap* for variables with overlapped memory storage, *access* (along with its subproperties *read* and *write*) for side effects of language constructs, and *dependence* (with sub-properties for true, anti, and output dependencies) for dependence relations. For a function with a returned type, we introduce the concept of *returnedVariable*, which is unique for each function and related to the function via a *returnedBy* relation. Semantics (such as aliasing) of the returned variable can then be conveniently described.

One implementation detail is each entity in the OWL ontology must have a unique id, or internationalized resource identifier (IRI) to allow unambiguous references. IRIs are defined in different namespaces to avoid name collision. We use the following choices when creating IRIs for the entities.

- All entities are organized under a namespace *http://www.semanticweb.org/stencilComputation#* .
- Fundamental classes and properties are denoted by their standard or most commonly used names. For example, *ForStatement* is used to indicate the for loops. Alternative popular names are also added.
- Individuals representing source code constructs use their source code location information to form IRIs. For example, a for loop located between a start position (line 27 column 1) and an end position (line 39 column 50) in a source file is denoted as */path/file.c:27-1:39-50*.
- Named language entities use qualified names as their IRS. For example, a class defined in a library is specified as *MyLibrary::FirstClass*.

With all these concepts and relations defined in multiple layers, the stencil computation ontology can act as a knowledge base storing a rich set of semantics which are essential to DSL optimization. For example, two aliasing variables can be expressed as *ObjectPropertyAssertion (:alias :var1 :var2)*.

3.3 Compiler Implementation With Ontology

We enhanced the original Shift Calculus DSL compiler to interact with the ontology-based knowledge base to store and retrieve software and hardware information relevant to optimizations.

Figure 7 shows the internals of the enhanced DSL compiler and some supporting components. One obvious addition is a knowledge generator, which traverses the abstract syntax tree (AST) generated from an input DSL program and generates instances of classes and relations. The generated knowledge is stored in the knowledge base, through SWI-Prolog's semantic web library interface. The generator may be invoked multiple times as needed during the DSL lowering process to generate knowledge tied to different levels of the AST. The generator also helps propagate some semantics in the AST. For example, the *returnedVariable* instance of a function call will be related to a left hand variable via a *SameIndividual* relation in a statement like *a = function();*. As a result, the semantics associated with the returned value of a function are propagated to the left hand object accepting the returned object in a Prolog query.

Other components are also free to update the knowledge base when necessary. For example, we have improved ROSE to provide a set of API functions to support transformation tracking. The DSL transformation (or lowering) phase calls these API functions to explicitly store mapping information between input and output program constructs of essential transformation steps. The transformation tracking API automatically updates the knowledge base with *subClassOf(:low-level-entity :high-level-entity)* to connect these entities so queries on low-level entities can return semantics associated with high-level entities.

An integrated development environment for OWL, Protege, is included in order to enable interactions between the knowledge base and human users. The knowledge base can be manually updated for additional domain knowledge.

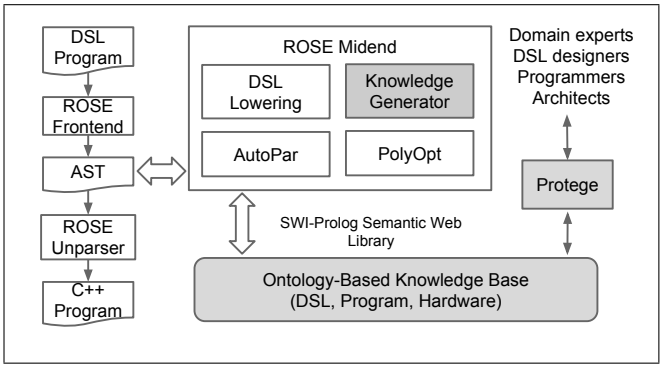

Figure 7: Compiler Implementation with ontology

We use two optimization components in ROSE, AutoPar and PolyOpt, as examples to demonstrate the utilization of the knowledge base.

ROSE has a module conducting automatic parallelization (referred to as AutoPar [14]) by inserting OpenMP directives into sequential codes. AutoPar is a semantic-aware parallelizer since it can leverage ROSE's high-level AST to recognize high-level abstractions and exploit their semantics for automatic parallelization. Previously, a customized semantics-specification file was designed to store the list of abstractions and their semantics. The compiler has a special parser to read the file and later use the information to help parallelization. In this paper, we extended AutoPar to additionally query the ontology-based knowledge base via Prolog Semantic Web library. Using liveness analysis, AutoPar was also extended to insert accelerator directives (*omp target device(..) map(..)*) introduced in OpenMP 4.0. For example, if an array typed variable which is only live-in at the entrance (and not live-out at the exit) of a parallel loop offloaded to an accelerators should have a map type of *to*.

Internally, AutoPar uses a dependence elimination algorithm to tell if a loop can be parallelized or not. A conservative dependency analysis first generates all potential dependence relations associated with a loop. A set of rules are then used to eliminate these dependencies. One example rule is that if a dependence is caused by a reduction variable (obtained by a separate reduction recognition analysis), the dependence can be eliminated. Another example is that, if a dependency reported by the dependence analysis is caused by two pointers and later the two pointers are found to be not aliasing or overlapping each other (obtained via high-level semantics stored in the knowledge base), it can also be eliminated. The loop can be parallelized if there is no dependencies left in the end.

ROSE also has a polyhedral optimizer, named PolyOpt, to perform sophisticated loop transformation and nested parallelization. Previously, PolyOpt took optimization parameters from user command lines to check the eligibility to perform transformations and then execute the transformation. Many of the optimization parameters are related to hardware-specific information, such as cache line size and cache memory hierarchy. PolyOpt is extended to query the ontology-based knowledge base for hardware features of a target platform.

3.4 Preliminary Results

We present a preliminary study to show the effectiveness of our work. The study takes an input code written in Shift Calculus DSL that applies a Laplacian operator and performs computation on a 7-point stencil (shown in Figure 4). The size of the source *Box* in this example is set to 512. Our DSL compiler generates the following four output variants:

- a sequential C++ output code without any optimization,
- a parallel C++ output code with classic OpenMP parallel loop directives.
- a tiled and parallelized output code generated from the polyhedral transformation with OpenMP directives inserted.
- a parallel output code with OpenMP 4.0 accelerator directives inserted. The code is further translated into CUDA code by ROSE's OpenMP accelerator implementation [15].

Table 2: Performance

Performance Table (in sec.)	
Output Variant	exec. time
C++ serial	1.51482
C++ OpenMP Parallel	0.380562
C++ Polyhedral Tiled+Parallel	0.503307
CUDA w/ data transfer	9.29446
CUDA w/o data transfer	3.14713e-05

All generated C++ code variants are further compiled by GCC version 4.8.3. The generated CUDA code is compiled by NVCC compiler version 7.0.

We run the tests on a 24-core workstation with Intel Xeon CPU E5-2620 V.3 and 64 GB memory. Four Nvidia Tesla K40c GPUs are also available on the same system. The performance results are listed in Table 2. It is clear that with additional software and hardware information, the ShiftCalculus DSL compiler can enable more optimizations which often leads to better performance. The only exception is the CUDA version when data transferring overhead is counted.

4. RELATED WORK

Ontology techniques have been used to accumulate and share knowledge in different domains. Prominent manually created ontologies include Cyc [17] and SUMO [20], which are aimed at specifying general-purpose concepts as upper ontologies. Many more domain-specific ontologies exist. One of the most successful ontolgoies is the gene ontology [2], which addresses the need for consistent descriptions of gene products and their relationships across all species in bioinformatics. In ambient intelligence [10], a research field of studying digital and proactive environment sensing to assist users in their daily lives, ontologies are used to model both environment context [21] and human behaviors [24]. In Robotics, KnowRob [28] is an influential ontology-based knowledge base for describing perception and actions of service robots. To the best of our knowledge, our work is the first attempt to apply ontology to the multiple domains related to DSL targeting HPC.

Some previous studies [32, 16, 5, 31, 27, 4] have explored using ontology to help develop domain specific languages for programming or modeling purposes. Most of these studies [27, 5] focus on domain analysis and/or language design, without discussion connections with implementation and optimizations aimed for performance. A notable study [5] compared ontology-based domain analysis with classic domain analysis using Feature-Oriented Domain Analysis (FODA). The authors also showed how ontology can be translated to DSL grammars. Others studied domain-specific modeling languages [31, 4], not for programming languages. For example, Walter et. al. [32] relies on expressiveness of OWL2 and its reasoning facilities to check concept satisfiability and consistency of domain-specific modeling . Lortal et. al [16] propose to reuse the knowledge of a robotic ontology to develop robotics modeling languages. In contrast, our work focuses on using ontology to enchance DSL implementations for programming parallel computers. We use ontology to capture not only software domain semantics (knowledge), but also crucial hardware details. Our approach also defines how compilers and runtime systems can interact with an ontology-based knowledge base to facilitate DSL code generation and optimizations.

Numerous DSLs, including stencil DSLs, have been developed for HPC. We only name a few examples for brevity. The ExaSlang[25] is one of the DSLs in ExaStencil project [13] that focuses on highly scalable multigrid solvers. It uses high-level syntax to describe the algorithmic information in a multigrid computation. The optimizations for a ExaSlang program are part of the code generation pipeline and will not be directed by the high-level language. A customized Target Platform Description Language (TPDL) is used by ExaStencil to describe machine information. Halide [23] is a DSL designed to describe image processing pipelines. Users explicitly specify the pipelines using chains of functions. The Halide compiler uses an auto-tuning approach to retrieve an optimal scheduling and performs required optimizations. STELLA (STEncil Loop Language) [1] is a C++ domain specific embedded language designed to implement the different stencil motifs for structured grids used in the Consortium for Small-Scale Modeling (COSMO). STELLA abstracts the stencil formulation to allow users to write codes with good portability. Our work is unique in that we enhance DSL implementations by adding a formal and dedicated knowledge base to explicitly store multiple layers of information related to domains, programs, and hardware.

5. CONCLUSIONS

In this paper, we have presented a novel ontology-based knowledge representation and utilization approach to capture, share and use both software and hardware information needed to enable efficient domain-specific language implementations targeting high performance computing. Compared to traditional ad-hoc approaches using scattered, customized annotations and specifications, our approach is formal, uniform, standardized, reusable, extensible and scalable. The chosen modern ontology language, OWL, enables us to leverage a wide range of knowledge engineering, validating, and reasoning tools developed for OWL. Using a stencil DSL as an example, we have demonstrated how our approach can be used to model essential concepts and properties in multiple software and hardware domains. We also have shown how the resulting knowledge base can easily interact with human users and software agents (e.g. compilers) to acquire new knowledge and retrieve existing knowledge to facilitate DSL implementations.

6. REFERENCES

[1] A. Arteaga, D. Ruprecht, and R. Krause. A stencil-based implementation of parareal in the c++ domain specific embedded language stella. *Applied Mathematics and Computation*, 2015.

[2] M. Ashburner, C. A. Ball, J. A. Blake, D. Botstein, H. Butler, J. M. Cherry, A. P. Davis, K. Dolinski, S. S. Dwight, J. T. Eppig, et al. Gene ontology: tool for the unification of biology. *Nature genetics*, 25(1):25–29, 2000.

[3] F. Baader. *The description logic handbook: theory, implementation, and applications*. Cambridge university press, 2003.

[4] M. Bräuer and H. Lochmann. *An ontology for software models and its practical implications for semantic web reasoning*. Springer, 2008.

[5] I. Ceh, M. Crepinšek, T. Kosar, and M. Mernik. Ontology driven development of domain-specific languages. *Computer Science and Information Systems*, 8(2):317–342, 2011.

[6] G. Chen, B. Wu, D. Li, and X. Shen. PORPLE: An Extensible Optimizer for Portable Data Placement on GPU. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 88–100, Washington, DC, USA, 2014. IEEE Computer Society.

[7] P. Colella, D. T. Graves, N. Keen, T. J. Ligocki, D. F. Martin, P. McCorquodale, D. Modiano, P. Schwartz, T. Sternberg, and B. V. Straalen. *Chombo Software Package for AMR Applications - Design Document*. Lawrence Berkeley National Laboratory, 2009. `https://seesar.lbl.gov/anag/chombo/ChomboDesign-3.0.pdf`.

[8] O. Corcho, M. Fernández-López, and A. Gómez-Pérez. Methodologies, tools and languages for building ontologies. where is their meeting point? *Data & knowledge engineering*, 46(1):41–64, 2003.

[9] A. Dubey. Stencils in scientific computations. In *Proceedings of the Second Workshop on Optimizing Stencil Computations*, pages 57–57. ACM, 2014.

[10] K. Ducatel, M. Bogdanowicz, F. Scapolo, J. Leijten, and J.-C. Burgelman. *Scenarios for ambient intelligence in 2010*. Office for official publications of the European Communities, 2001.

[11] J. H. Gennari, M. A. Musen, R. W. Fergerson, W. E. Grosso, M. Crubézy, H. Eriksson, N. F. Noy, and S. W. Tu. The evolution of protégé: an environment for knowledge-based systems development. *International Journal of Human-computer studies*, 58(1):89–123, 2003.

[12] S. Harris, A. Seaborne, and E. PrudâĂŹhommeaux. Sparql 1.1 query language. *W3C Recommendation*, 21, 2013.

[13] C. Lengauer, S. Apel, M. Bolten, A. Größlinger, F. Hannig, H. Köstler, U. Rüde, J. Teich, A. Grebhahn, S. Kronawitter, et al. Exastencils: Advanced stencil-code engineering. In *Euro-Par 2014: Parallel Processing Workshops*, pages 553–564. Springer, 2014.

[14] C. Liao, D. J. Quinlan, J. Willcock, and T. Panas. Semantic-aware automatic parallelization of modern applications using high-level abstractions. *International Journal of Parallel Programming*, 38(5-6):361–378, 2010.

[15] C. Liao, Y. Yan, B. R. de Supinski, D. J. Quinlan, and B. Chapman. Early experiences with the openmp accelerator model. In *OpenMP in the Era of Low Power Devices and Accelerators*, pages 84–98. Springer, 2013.

[16] G. Lortal, S. Dhouib, and S. Gérard. Integrating ontological domain knowledge into a robotic dsl. In *Models in Software Engineering*, pages 401–414. Springer, 2011.

[17] C. Matuszek, J. Cabral, M. J. Witbrock, and J. DeOliveira. An introduction to the syntax and content of cyc. In *AAAI Spring Symposium: Formalizing and Compiling Background Knowledge and Its Applications to Knowledge Representation and Question Answering*, pages 44–49. Citeseer, 2006.

[18] D. L. McGuinness and F. Van Harmelen. Owl web ontology language overview. *W3C recommendation*, 10(10):2004, 2004.

[19] B. Motik, P. F. Patel-Schneider, B. Parsia, C. Bock, A. Fokoue, P. Haase, R. Hoekstra, I. Horrocks, A. Ruttenberg, U. Sattler, et al. Owl 2 web ontology language: Structural specification and functional-style syntax. *W3C recommendation*, 27(65):159, 2009.

[20] A. Pease, I. Niles, and J. Li. The suggested upper merged ontology: A large ontology for the semantic web and its applications. In *Working notes of the AAAI-2002 workshop on ontologies and the semantic web*, volume 28, 2002.

[21] D. Preuveneers, J. Van den Bergh, D. Wagelaar, A. Georges, P. Rigole, T. Clerckx, Y. Berbers, K. Coninx, V. Jonckers, and K. De Bosschere. Towards an extensible context ontology for ambient intelligence. In *Ambient intelligence*, pages 148–159. Springer, 2004.

[22] D. Quinlan and C. Liao. The ROSE source-to-source compiler infrastructure. In *Cetus Users and Compiler Infrastructure Workshop*, Galveston Island, TX, USA, October 2011.

[23] 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. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[24] N. D. Rodríguez, M. P. Cuéllar, J. Lilius, and M. D. Calvo-Flores. A survey on ontologies for human behavior recognition. *ACM Computing Surveys (CSUR)*, 46(4):43, 2014.

[25] C. Schmitt, S. Kuckuk, F. Hannig, H. Köstler, and J. Teich. Exaslang: a domain-specific language for highly scalable multigrid solvers. In *Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, pages 42–51. IEEE Press, 2014.

[26] S. Staab and R. Studer. *Handbook on ontologies*. Springer Science & Business Media, 2013.

[27] R. Tairas, M. Mernik, and J. Gray. *Using ontologies in the domain analysis of domain-specific languages*. Springer, 2009.

[28] M. Tenorth and M. Beetz. KnowrobâĂŤknowledge processing for autonomous personal robots. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, pages 4261–4266. IEEE, 2009.

[29] D. Tsarkov and I. Horrocks. Fact++ description logic reasoner: System description. In *Automated reasoning*, pages 292–297. Springer, 2006.

[30] A. Van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *Sigplan Notices*, 35(6):26–36, 2000.

[31] T. Walter, F. S. Parreiras, and S. Staab. Ontodsl: An ontology-based framework for domain-specific languages. In *Model Driven Engineering Languages and Systems*, pages 408–422. Springer, 2009.

[32] T. Walter, F. S. Parreiras, and S. Staab. An ontology-based framework for domain-specific

modeling. *Software & Systems Modeling*, 13(1):83–108, 2014.

[33] J. Wielemaker. SWI-Prolog Semantic Web Library 3.0.

[34] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.