OpenK: An Open Infrastructure for the Accumulation, Sharing and Reuse of High Performance Computing Knowledge

Yue Zhao[⋆], Xipeng Shen[⋆], Chunhua Liao[+]
[⋆] Computer Science Department, North Carolina State University
[+] Lawrence Livermore National Laboratory

ABSTRACT

In high performance computing (HPC), often a wide range of knowledge about the particular domains, software applications, and hardware architectures are necessary for applying the right optimizations in the appropriate contexts. In today's practice, however, such knowledge has been expressed and managed in an ad-hoc manner by each individual. Systematic accumulation, sharing and reuse of the knowledge across a broad community has been difficult. This paper introduces OpenK, an ontology-based open infrastructure for solving the problem. By centering around ontology – a generic knowledge representation – and extensible designs, it offers a promising support for lowering the barriers.

1 INTRODUCTION

Software optimization on a program often requires a wide range of knowledge, from the program code, to the algorithm it implements, the domain, the underlying hardware, properties of various optimizations, their interactions, and so on. The problem is especially important for High Performance Computing (HPC), where, some performance-critical applications run on some large, complex supercomputers or clusters. Optimizing these applications is especially difficult, due to the high requirement of the desired performance and energy efficiency, as well as the growing complexity in HPC software and hardware.

Currently, HPC knowledge has been scattered everywhere in various formats. Examples include implicit assumptions about the problem domains, semantics and usage constraints of library functions, analysis and optimization guidelines that are hard-coded inside compilers, hardware architecture attributes and settings written in whitepapers or complicated system configuration files, and findings written in research papers, reports, and other documentations. The adhoc knowledge representation and organization have created tremendous difficulties for the HPC community to effectively accumulate, share, and reuse HPC knowledge.

Take optimizations of stencil computation as an example. Stencil codes are a class of iterative kernels which update array elements according to some fixed patterns, called stencils. As the core computing pattern in many scientific simulations, high-performance stencil computing has drawn many studies in the last decades. Just an incomplete counting on Google Scholar shows that there are 1,640 research documents published on the topic "stencil computations" in the last five years, some on GPU memory performance, some on non-uniform memory access (NUMA), and some on thread scheduling or other optimizations. In addition, our interactions with U.S. DOE National Labs show that even more empirical studies on the topic have left undocumented.

A survey of the practices in U.S. National Labs shows that despite the large volume of prior work, there are still many explorations going on the topic everyday. Many of them are simple repetitions of some others' explorations due to the unawareness of the prior work or the unavailability of their results. For example, a number of groups have spent a lot of man power in finding out the influence of different loop tilings on some stencil computations on NUMA systems. Many of the experiments are repetitions of the (some formally documented, some not) explorations already done by some other groups on identical or very similar platforms and have ended up with the same conclusions or insights.

A fundamental reason for the redundant efforts is the difficulty in systematically collecting and leveraging prior experiences. That takes time and is sometimes even infeasible. For instance, collecting some major works on optimizing stencil computations and summarizing the main contributions of each took a substantial part of a thesis work [12]. Even today, with the help of modern search engines, it still takes the lead author of this paper about two months to find a modest set of relevant papers, read and digest them.

The observations on stencil computations represent a fundamental issue facing HPC—the lack of a flexible, open infrastructure for the HPC community to effectively share, accumulate, and reuse HPC knowledge.

Developing such an infrastructure is the goal of this work. There have been some scholarly research databases [22, 33, 38], digital libraries, or technical forums (e.g., stackoverflow.com). However, they cannot meet the needs. For instance, none of them allow automatic inference tools to be built upon the knowledge base, none of them offer a structural representation of the deep knowledge such that automatic software optimization agents can easily leverage, and none of them allow the representation of computing programs or extensions to their infrastructures. There are some knowledge managing systems in other domains, such as Google Knowledge Graph [2] and GoKB [1]. These systems are either for general knowledge or a domain (e.g., e-resource library) without the special properties and requirements of HPC (e.g., program code representations, systematic support for program optimizations, etc.).

The OpenK system we propose in this work is the first infrastructure that provides a comprehensive way to help the HPC community systematically accumulate, share, and reuse HPC knowledge. OpenK's design centers around *ontology* [37], a primary way to standardize the concept definitions and knowledge representations for a domain. Such a choice is crucial for removing inter-user collaboration barriers as it standardizes terminology and concepts, and allows many sources of knowledge to be represented in a way immediately ready for automatic inferences and analysis.

OpenK is based on the modern development of knowledge engineering. Some special properties of HPC, however, creates a range of open challenges. Examples include what techniques from the broad field of knowledge engineering will best fit for this work, how to put these techniques together to meet the specific needs of HPC, how to leverage them effectively, and what potential benefits such an infrastructure can bring.

This paper describes the results of our efforts in answering these questions. It presents the structure of OpenK, and the rationale for the major design choices. It demonstrates the usage and benefits of OpenK through three case studies: an optimization advisor for stencil computations, a GPU data placement optimizer, and a collaborative data flow analysis. In them, OpenK substantially improves the productivity of code analysis and optimizations and the extensibility of HPC optimization tools.

The results show that OpenK is a promising infrastructure for HPC knowledge management and for promoting the HPC community synergy. At a higher level, OpenK is a step to bridge the gap between HPC and knowledge engineering fields. Recent years have witnessed exiting advancements in cognitive computing, knowledge engineering, and other related fields. By allowing systematic representations of HPC knowledge and connecting them with program optimizations, this work may open many opportunities for HPC to benefit from advancements in those fields.

Overall, this work makes the following contributions:

- After exploring a broad set of techniques on knowledge engineering, it identifies the techniques that suite HPC needs and puts them together into the first infrastructure for supporting HPC knowledge management, accumulation, sharing, and reuse.
- It reveals a range of design considerations, insights, and principles, especially on overcoming the special complexities in HPC. Some of the insights include the use of ontology as the representation to accommodate the large variety of (and often unstructured) knowledge in HPC, the extensive and modular design to support the many different domains in HPC, the versatile interfaces to enable the needed human and software interactions in optimizing HPC programs, and the leverage of the ontology ecosystem that has been rapidly built in the past decade.
- It reports the usefulness of OpenK in improving the productivity and tool extensibility of HPC in two use case studies, demonstrating its promise in removing the long-standing barriers for HPC knowledge management.

2 BACKGROUND

For the central role that ontology plays in OpenK, we first give a brief introduction of ontology. Ontology [37] is a concept originating in Philosophy, referring to the study of the nature of being, as well as the basic categories of them and their relations [28]. In recent decades, it has become a formal way to explicitly represent the knowledge in a domain. The ontology of a domain consists of the concepts (or called classes) in the domain, their instances (or called individuals), and their relations (or called properties).

Ontology is embodied with a defined vocabulary, which users can take as the standard terminology to formally express the knowledge in that domain. The expression of a piece of knowledge can state explicitly the relations between these individuals and the classes defined in the ontology. An ontology or ontology-based expression can be visualized as Figure 1 illustrates. Each node in the graph represents a concept or instance, and each edge carries a property indicating the relations between the two nodes it connects. In Figure 1, the edge between "Robot" and "Arm" indicates that the latter is a part of the former.

Textually, a common language for expressing ontologies is the Web Ontology Language (OWL) [16]. OWL is based on the description logic (DL) and is expressive enough for building sophisticated knowledge bases while still supporting efficient inference. Specifically, Resource Description Framework (RDF) is a commonly used simple format of OWL for knowledge representation. Each piece of knowledge is represented as a triple, (subject, property, object). For instance, ("Tatooine", instanceOf, "Planet"), states that "Tatooine" is

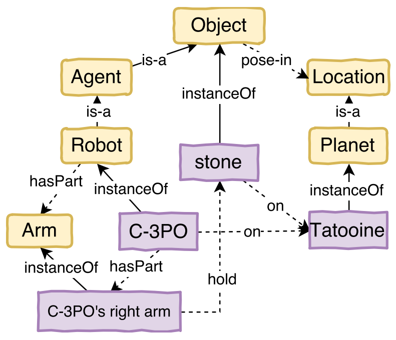


Figure 1: An example ontology on Robotics.

an instance of "Planet". Another popular variant is in the form of property(subject, object). So the previous example can also be written as instanceOf ("Tatooine","Planet").

Ontology has been used in the Semantic Web, software meta data management, Robotics, and some other fields outside HPC. The last several decades have witnessed some rapid development of ontology-related techniques. A large body of tools (e.g., Stanford Protégé [14], SWI-Prolog Semantic Web Library [40], and HermiT [15]) have been developed for creating ontologies and automatic reasoning upon an ontology-based knowledge base, which enables automatic questions-and-answers, consistency check of the knowledge base, derivation of new knowledge, and so on.

3 SCOPE AND DESIGN CONSIDERATIONS

We first note that fully automating HPC knowledge aquisition from data and reports is not the goal of this work. It is a task requiring continuous research in Machine Learning, NLP, and Information Retrieval. The goal of this work is to provide an infrastructure, through which the HPC community and automatic tools can put HPC knowledge together in a way easy for other people or tools to reuse. The main consideratons for design are in seven dimensions.

(1) *Generality.* The knowledge base must be able to accommodate a wide range of knowledge of various levels (e.g., software and hardware), sources (e.g., domain experts, code), and forms (e.g., code structures, optimization rules), thanks to the broad range of knowledge that HPC needs.

(2) *Deep Inference.* The representation of knowledge should be amenable for automatic logic inferences (e.g., finding out the best way to optimize a given stencil computation on a specific hardware) to make it possible to do sophisticated knowledge exploitation rather than simple search as what common search engines do. Such inferences are important for tapping into the full potential of the accumulated knowledge for HPC.

(3) *Consistent Conceptualization.* For the same concept, different groups may use different names (e.g., *worker groups* in OpenCL [34] versus *thread blocks* in CUDA [29] for GPU). Inconsistency in the definition of concepts and terminology may cause lots of difficulties for search and many confusions in knowledge representations. The knowledge management infrastructure should ideally prevent them.

(4) *Usability.* With the infrastructure, it must be easy for both the community and automatic software agents to add, update, retrieve, and visualize the HPC knowledge. And furthermore, its knowledge representation should be standard, intuitive, and efficient in order to simplify the collaborations among different users and software agents, helping promote synergies.

(5) *Extensibility.* As an infrastructure for maximizing the synergy of the HPC community, it should be inherently open and extensible. That refers to the infrastructure itself, which should be easy to extend with new or improved functionalities (e.g., deriving new knowledge through Machine Learning modules). That also refers to the knowledge base built in the infrastructure in the sense that the community should be able to add and access it flexibly.

(6) *Quality.* Quality control is essential for an open knowledge base. The infrastructure should have a way to identify issues (e.g., inconsistency) in the knowledge base, mechanisms for assessing the quality of added knowledge, and methods for helping users avoid the negative influence of errors in the added knowledge.

(7) *Ecosystem.* It would be ideal if there is already a software ecosystem built around the knowledge representation. The existing tools in the ecosystem will help simplify the various tasks (knowledge base construction, consistency check, inferences) related with the usage of the knowledge infrastructure.

4 OVERALL STRUCTURE OF OPENK

Designed to meet those considerations, OpenK offers a list of features. Through its web or programming interface, programmers or software tools (e.g., compilers or program optimizers) can add a broad range of knowledge into a centralized public knowledge base on a cloud, and can retrieve the knowledge, visualize it, or conduct automatic inferences (through logic programming) upon it for enhancing the quality of HPC. It allows easy extensions with additional modules and functionalities. It uses automatic consistency checks, crowdsourcing and some other ways to help control and assess the quality of knowledge. It facilitates the construction of a common conceptual framework to help avoid terminological or conceptual ambiguities. The rich existing ecosystem of ontology simplifies both the construction and the usage of the infrastructure.

Figure 2 depicts the overall structure of the OpenK framework, which consists of a front end, a middle end, and a back

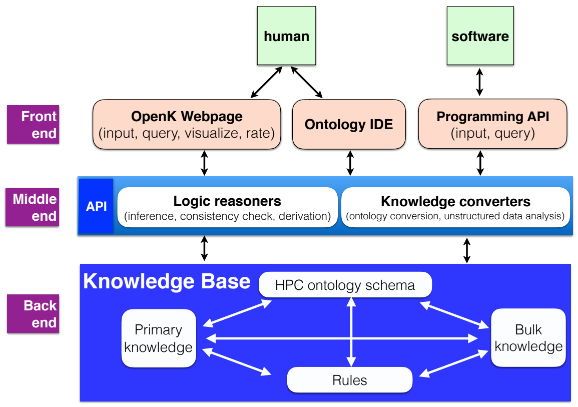

human
software
Front end
OpenK Webpage (input, query, visualize, rate)
Ontology IDE
Programming API (input, query)
Middle end
API
Logic reasoners (inference, consistency check, derivation)
Knowledge converters (ontology conversion, unstructured data analysis)
Back end
Knowledge Base
HPC ontology schema
Primary knowledge
Bulk knowledge
Rules

Figure 2: Overall structure of OpenK.

end. This section provides a brief overview of each part; next section elaborates the internal design.

Front End. The front end is the interface to the users (both humans and software agents). It includes three components. The first is a webpage–based interface, through which, humans can do the following:

- Submit queries related with the OpenK knowledge base; the results will be displayed on the webpage for browsing or downloading.
- Visualize—in a RDF graph manner—the knowledge base or the query results.
- Submit ratings on a piece of knowledge to reflect the degree of approval to the knowledge (crowdsourcing).
- Suggest extensions to the HPC ontology (e.g., adding new concepts or relations to a sub-domain of the ontology.)
- Submit new modules for logic reasoning of the knowledge base or for converting raw information into the ontology format.
- Upload new knowledge into the knowledge base. The uploaded knowledge can be in various formats, including domain-specific forms, RDF, unstructured documents (e.g., manuals on a library like MPI [18]), or source code of a computer program.

The second interface is a locally installed integrated development environment (IDE) based on Protégé [14]. It offers similar functionalities as the web interface offers, and additionally, provides further conveniences to the user through various plugins for refactoring, querying and reasoning.

The third interface is a programming interface, which is intended to be used in other software that needs to access the knowledge base. What it currently supports are to add new knowledge and to submit queries for knowledge retrieval. The API is based on SWI-Prolog Semantic Web Library [40] and the Jena Ontology Framework [8].

Middle End. The middle end of OpenK consists of two main components. The first is a set of logic reasoners including SWI-Prolog [7] and HermiT [15] , which conduct automatic logic inferences on the OpenK knowledge base to find answers to queries (e.g., finding the type of memory that best fits an array's access pattern as Section 6 will present), or check consistency of the knowledge base. This component is designed to be extensible, allowing easy incorporations of modules for other types of analysis of the knowledge base (e.g., statistical analysis through Machine Learning).

The second component is a set of knowledge converters, which take the inputs from various sources and/or in various formats and generates knowledge represented in RDF triples and sends it to the OpenK knowledge base. One of the implemented converters in the current OpenK is a C/C++ program converter, which converts an input C/C++ program into RDF that captures the main language concepts, code structures, data structures, and control flows in the input program [41]. Another converter in OpenK converts a hardware specification written in a language called MSL [10] into RDF. In addition, OpenK is equipped with Pellet [32], an efficient reasoner for querying and processing OWL-2 ontologies and Semantic Web Rule Language (SWRL) rules [17]. For an unstructured document uploaded by a user, if there is no existing module for converting its content into RDF, OpenK would treat the entire document as a single instance and put it, along with any available meta data related with the document, into the OpenK knowledge base. New converters can be continuously added through some APIs.

All the middle-end modules themselves are documented in the OpenK knowledge base such that users can easily find out the right modules to use if they would like to specify the reasoners or converters to use in their queries (such specifications are optional; without them, OpenK would select some default modules that are compatible to the given query.)

Back End. The back end of OpenK is the actual knowledge base. Based on their nature, the knowledge in OpenK can be classified into four kinds.

The first is an HPC ontology schema, which is made up of the concepts and their relations in the various HPC domains. It defines the vocabulary for the domains (e.g., constructs and terms of C language or GPU architectures). To facilitate a continuous growth of the ontology schema, OpenK offers a high-level taxonomy of the domains of HPC whereby each sub-domain can be separately populated by users in the sub-domain through a set of API.

The second kind is the knowledge on some concrete entities (e.g., a program or a hardware). We call them *primary* *knowledge*. Each element in it represents a concrete object or an attribute of a concrete object (e.g., a memory system and its size).

The third kind is *logic rules*, such as what optimizations should be applied under what conditions. These rules often embody the optimization insights some explorations have obtained (e.g. if a read-only array on GPU fits in constant memory, it should be put into constant memory for improved performance.)

The final kind is *bulk knowledge*, each entity of which is a collection of information (e.g., the manual of a library), the content of which is not yet parsable or representable in ontology. Some unstructured documentations written for humans to read belong to this category. Even though their contents are not yet amenable for automatic logic inferences, keeping them in the knowledge base, along with their meta data, can help provide references to users. Moreover, as more advanced tools are added into OpenK, these documents could later become automatically exploitable.

Despite their different natures, the four kinds of knowledge are represented in some coherent formats in OpenK, forming a seamlessly integrated knowledge base for the reasoners in the Middle End to use. In the next section, we give a more detailed description of OpenK, and explain how they make OpenK able to meet the seven requirements listed earlier.

5 INTERNALS OF OPENK

The main techniques in OpenK are in the following five aspects:

- *HPC knowledge representation:* How to represent the various HPC knowledge internally to offer both high efficiency and good flexibility.
- *HPC knowledge submission:* How to ease knowledge submission.
- *HPC knowledge derivation:* How to support the derivation of new (or generalized) knowledge from what OpenK already contains.
- *HPC knowledge quality control:* How to assess and control the quality of the knowledge in OpenK.
- *HPC knowledge capitalization:* How to effectively support the capitalizations of the knowledge in OpenK.

We next describe them each, along with the important design principles and insights.

5.1 HPC Knowledge Representation

Driven by the various sources and forms of HPC knowledge, we equip OpenK with an ontology-based knowledge representation. As a formal way to provide a consistent conceptualization of a domain, an ontology of a domain defines the set of concepts, their relations, and corresponding terms

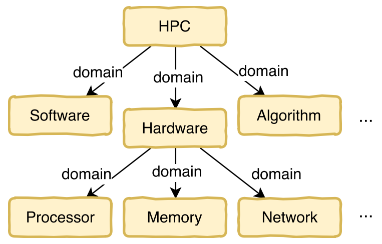


Figure 3: Part of the HPC knowledge taxonomy.

(and aliases) specific to the domain. Having such an ontology can effectively avoid ambiguities in conceptualization and terminology usage of a domain, removing barriers for collaborations among a community in both growing and leveraging the knowledge base.

It is not necessary to build a complete HPC ontology for OpenK knowledge base to be useful. As the ontology for some HPC sub-domain is constructed, problems in the sub-domain can start benefiting from the infrastructure and knowledge base. Currently, OpenK provides the ontologies defined for several specific areas of HPC, including an ontology of the C programming language (based on C99), an ontology for GPU hardware, and an ontology for stencil computations. These are useful for our case studies shown later.

On the other hand, OpenK provides a framework with a two-dimensional strategy to help with the creation of the ontology and the organization of the knowledge of more HPC sub-domains.

On one dimension (horizontal dimension), we classify the set of HPC knowledge into a hierarchy of domains and create a high-level HPC knowledge taxonomy as illustrated in Figure 3. This taxonomy is represented in RDF inside the OpenK knowledge base and can be queried and visualized by users through the OpenK interface. The taxonomy has two main purposes. (1) By organizing the knowledge base into domains, it makes the knowledge base easy to browse. A user can focus on the domains interesting to her. (2) The isolations provided by the taxonomy among different domains help simplify the expansion of the HPC ontology. When a special interest group design and create the ontology (concepts and vocabularies) specific to N-body simulations, for instance, their changes to the HPC ontology will be constrained to a local scope of the taxonomy in OpenK, casting little perturbation to the ontologies (and their usage) of other domains. Meanwhile, despite the separate development of the domains, the knowledge base in one domain can connect with the knowledge base in another domain through RDF edges that carry the corresponding relations between them.

The second dimension (vertical dimension) is a classification of the knowledge within each domain and some standardized representations. As mentioned in the previous section, the knowledge is categorized into primary knowledge, bulk knowledge, and optimization rules.

The first two kinds are both represented in RDF, respectively for detailed knowledge with clear semantic defined in the ontology and for linked resources the content of which is not yet parsed and represented in ontology. The generic format of RDF allows various knowledge to be expressed in a single coherent format, making automatic inferences across the different types and levels of knowledge possible. Objects in the knowledge base are named in *internationalized resource identifiers (IRIs)* [16] to ensure the uniqueness of resource names.

The third kind—rules—is worth further explanations. In HPC, through the decades of research efforts, the community have attained many insights on how to maximize the execution speed, power efficiency, and reliability of certain applications running on certain systems. These insights, involving logic relations, are often cumbersome to be expressed in RDF. But seamlessly integrating these insights into the OpenK knowledge base is essential for capitalizing such exiting valuable knowledge. And the integration should allow automatic inferences with these insights upon the other parts of the knowledge base. Our solution is to use Semantic Web Rule Language (SWRL) [17] to represent rules. SWRL is a widely supported language extending OWL with logic rules [17]. It helps extend the OpenK RDF-based knowledge base with optimization rules; these rules may reference the instances and concepts in the RDF triples. Modern rule engines (e.g., Drools [30], the default rule engine integrated into Protégé 5.0) conduct inferences on SWRL rules and its related RDF-based ontologies seamlessly. A special component of the OpenK knowledge base is the representation of OpenK itself. It describes the structure of the knowledge base in RDF and gets automatically updated when the knowledge base changes (e.g., a sub-domain is added), making it easy for users to learn about OpenK through its own interface.

5.2 HPC Knowledge Submission

As an infrastructure supporting HPC knowledge accumulation, OpenK allows the submission of a wide range of knowledge. If the external source is already in RDF or SWRL rules, they can be directly added into the knowledge base. For other kinds of sources, the knowledge conversion tools in OpenK offer the help. OpenK is currently equipped with a set of conversion tools: a ROSE-compiler–based tool for converting C/C++ programs into RDF [41], a tool for converting hardware specifications (in MSL [10]) into RDF, and a tool [39] for conversions between SWRL rules and Prolog rules to support using Prolog as a logic reasoner. As an extensible infrastructure, it allows new conversion tools for other types of knowledge to be added continuously. With these tools, raw materials can be automatically converted into knowledge stored into the knowledge base of OpenK.

For public HPC knowledge sources (e.g., public benchmarks, documentations of some common libraries), OpenK can automatically retrieve those materials and invoke the conversion tools to turn them into the representations in its knowledge base; it can keep the knowledge up to date by repeating the process periodically. For a non-public knowledge source that a user would like to share (e.g., some experimental results, some API documentations), the user can provide the URL of the source for OpenK to retrieve, or directly upload the materials through the OpenK interface.

5.3 HPC Knowledge Derivation

Knowledge derivation is to derive new knowledge from the knowledge already in the knowledge base. This feature is essential for generalizing HPC knowledge. For instance, when GPU started to get adopted in HPC, a number of HPC application groups in some US DOE national labs put in lots of man power trying to port their applications onto GPU and optimize them. Many of these efforts could have been saved if the experiences from some other groups could be effectively translated into general insights and hence benefit the porting of some other applications. If OpenK was available, one group may submit their observations of the effects of different optimizations and porting strategies of their applications, along with the applications themselves, into the knowledge base of OpenK. Although each of these observations may be attained only on some specific application, as more observations are submitted, some automatic machine learning tools could try to find correlations between the effective optimizations with certain properties of the applications and hence crystallize the observations into some general insights (e.g., for programs with many irregular memory accesses, try to coalesce the memory references among threads in a warp.) Such general knowledge may then be put into the knowledge base of OpenK, and benefit new applications.

Through OpenK APIs, users can upload tools for deriving knowledge from its RDF knowledge base. For proof of concept, we have uploaded a correlation-based rule derivation engine into OpenK. In addition to machine learning-based tools, some logic inference tools can also provide some useful new knowledge that could be added into the knowledge base. For instance, OpenK already contains several program analysis tools, including canonical loop analysis for finding canonical loops in a program, data access pattern analysis for deriving the patterns of memory references from a program, point-to alias analysis, and so on. Each of these analysis tools may derive some useful attributes about a program.

When these tools are invoked by a user, the results could be also stored in the knowledge base of OpenK to save the time needed for re-deriving those attributes. Upon changes to the program, the save relevant attributes are invalided.

As more derived knowledge is added into the knowledge base, there could be space concerns. OpenK split the knowledge base into a core knowledge base and multiple loadable supplemental knowledge bases. The core knowledge base is always loaded and others are loaded as needed. In addition, OpenK is equipped with a cache-like management mechanism. It maintains a buffer to store derived supplemental knowledge. When the upper limit of the buffer gets reached, OpenK starts to evict some of the stored knowledge (which would need to be re-derived when needed). For the eviction policy, there can be multiple choices: least-recently-used (LRU), least-frequently-used (LFU), and their variants.

5.4 Knowledge Quality Control

Knowledge systems are frequently subject to inconsistency and errors. A common difficulty for knowledge system development is to maintain a high quality of the knowledge. As an open platform that intends to accommodate knowledge from many users in the HPC community, the quality control of the knowledge base in OpenK is especially important. Errors could be in various forms: factual errors in the submitted primary knowledge, missing conditions in a submitted optimization insight, misleading insights obtained through some biased experiments, and so on.

OpenK has three quality control mechanisms.

The first leverages the properties of ontology (RDF) and SWRL-rules. Thanks to the simple yet generic format of RDF for knowledge representation and SWRL for logic rules, it is easy to automatically detect inconsistencies in a RDF-based knowledge base. For instance, assume we have a *hasMemory* relation with a constraint that its object field must be an instance of a Memory type. If a piece of knowledge like *(machine1, hasMemory, disk1)* is put into the knowledge base, where *disk1* is not a memory type, consistency checking by reasoners can easily discover such an error. There have been a number of reasoners built for that purpose. We choose HermiT [15], which is an ontology reasoner based on hypertableau calculus that run more efficiently than other tools.

The second mechanism is to leverage a community synergy for quality control through crowdsourcing. Upon the return of a query on a piece of knowledge, the webpage also provides a field allowing the user to indicate whether she finds the knowledge useful or erroneous and for the user to put down her comments. At the same time, the current rating of the piece of knowledge is also shown along with the returned knowledge on a query (for queries submitted both by users directly or by software agents), based on which, the user or software agent can get certain idea of the quality of the knowledge and choose to use the knowledge or not. Domain administrators may periodically review the pieces of knowledge marked as erroneous.

Finally, OpenK has a built-in approval scheme. Some major changes (e.g., adding the ontology of a new domain) to the knowledge base or tool-sets are subject to the approval of the administrator. Tool changes (e.g., adding a new knowledge derivation tool) are put into a beta branch when they are submitted by the users initially. Only after sufficient validations by the community (through crowdsourcing), they are committed into the master branch.

5.5 HPC Knowledge Capitalization

There are many ways to use the OpenK knowledge base. A user may directly query it to find out some properties about a hardware or a program. She may also use OpenK as a tool for code refactoring or manual optimizations by submitting a program and the target system models and querying OpenK for advices to optimize the program. OpenK can also be used to assist some third-party software analysis or optimization tools. The objectives that OpenK may help achieve can be of a broad range, from execution speed to energy efficiency, performance debugging, and so on. For instance, a compiler may query it to leverage some domain knowledge in its compilation and optimization of a program; a program autotuner may query it such that the expert's insights or domain knowledge stored in OpenK can help the autotuner effectively prune the configuration space and quickly find the best parameters.

The usage of ontology in OpenK for knowledge representation simplifies the development of the software agents for capitalizing the knowledge base. The tools can be written quickly through descriptive programming, and can seamlessly reference all kinds of relevant knowledge (programs, hardware, domain-specific insights, etc.) in a single format. We will demonstrate some of these benefits through the case studies described in the next section.

To support these uses, OpenK provides a set of interfaces, for human users and software agents to submit queries and receive the answers. The interface to humans include a search box on the webpage, which accepts both keywords and queries written in SPARQL and Prolog (two common languages for RDF-based logic querying). The keyword-based search is similar to those in common search engines based on the Apache Solr framework [13]. The support to SPARQL and Prolog queries makes it possible for users to express some more complex queries. SPARQL is a standard RDF query language. It allows for a query to consist of triple patterns, conjunctions, disjunctions, and optional patterns. Many OWL reasoners, including HermiT, already support

SPARQL queries. Prolog is a general-purpose logic programming language. It is declarative: The program logic is expressed in terms of relations, represented as facts and rules. We use SWI-Prolog as our Prolog engine.

OpenK reports the search results with hyperlinks on the keywords, through which, users can learn more about the relevant knowledge. It also provides a visualization tool built on the D3 library [6]. to display the results in a relational graph, allowing flexible zooming functionalities.

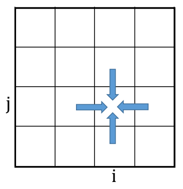
OpenK provides a set of programming API for software agents to submit queries as well. The results returned to software agents can be in several common formats (RDF, JSON, CVS, XML), depending on the options used in the submitted query.

Efficiency. Much of the inference upon ontology is through logic reasoning. Recent years have seen some substantial improvements of the performance of logic reasoners [5, 36]. For example, in SWI-Prolog, the ontology is stored as relation triples of (subject property object) with C extensions and some indexes are built for each element in the triples. So a search of a particular element can be done in constant time. Additional optimizations can be applied to queries. The *cut* operator (i.e., the ! symbol) in Prolog, for instance, can help avoid unwanted backtracking in search. It often helps to quickly narrow down the search space if one follows some existing guidelines when writing queries [7]. Meanwhile, in cases where the relevant knowledge base is simple and consists of straightforward facts in triples (e.g., some memory configurations), one may construct a customized lightweight parser in high performance languages to achieve better performance than using a heavy-weight logic reasoner. Our case studies shown next indicate that the efficiency of ontology-based program analysis has an efficiency level comparable to traditional imperative implementations.

6 USE CASE STUDIES

In this section, we describe three use case studies of OpenK. The first is to assist programmers in manual transformations of some stencil applications to achieve a high performance, the second is to enable collaborative operations of two different compilers to enhance the quality of their program analyses, and the third is to help an automatic GPU program optimizer determine the best ways to place data on GPU memory.

Through them, we examine the conveniences that OpenK brings to the organization of various HPC-relevant knowledge, to the accumulation of knowledge, and to the exploitation of the combination of these knowledge for program analysis and optimizations. We will also report our observations on the efficiency of ontology-based analysis, and the experience we had as users of OpenK in inputting and retrieving the knowledge.



$$A[i,j] = 0.25 * (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1])$$

Figure 4: An example of stencil computations.

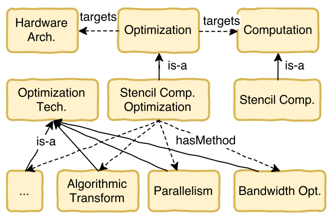


Figure 5: Part of the ontology of stencil computation optimizations.

6.1 Case I: Stencil Computation Optimizations

The first case study uses stencil computation to demonstrate how OpenK supports knowledge representation, accumulation, reuse, and capitalizations.

Stencil computation refers to a computation pattern, in which, the value of a point in a multi-dimensional spatial grid at time t is updated iteratively using the values of its neighboring points at times before t. Figure 4 gives a simple example, in which, the value of point $A[i,j]$ is set to the average of its four immediate neighbors. Different stencil computations differ in the expressions and neighbor sets, but are all about computations local to a grid.

Stencil computation is an extensively studied computation pattern. However, for the lack of a systematic way to organize the knowledge, the prior findings are scattered in thousands of papers and reports, difficult for the public to leverage. Our survey in DOE National Labs show that, for a given stencil computation and a target system, frequently, people just start a whole process of manual exploration and performance tuning to reach some satisfying performance. Duplicated experiments are common.

In this case study, we examine how the situation could be different if OpenK is made available to the community. To do that, we build the ontology schema for the stencil computation sub-domain in OpenK. Figure 5 gives a glimpse of it. We collected 30 research papers on optimizations of stencil computations, and then ask 10 computer science graduate students to play the role of the different research groups that

1. On single-core cache-based arch, tiling is effective if problem size is large than cache capacity.
2. For stencils with higher arithmetic intensity (>27-point), register blocking and reordering is useful.
3. On x86 and Blue Gene/P arch, explicit SIMD is effective.
4. On x86 arch and AMD Barcelona, icc compiler (w/ automatic SIMD) is better than gcc.
5. SIMD requires data alignment (16-byte boundary) and array padding.
6. For memory bound computation, bandwidth optimization is effective, and SIMD has limited benefits.
7. 7-point stencil is easy to be memory bound.
8. Xeon E5355, 3D 7-point stencil, problem size 512^3, cache blocking is ineffective but CSE (common subexpression elimination) is useful.

Figure 6: Some insights on optimizing stencil computations.

have done those prior studies. For that purpose, they are then asked to read three of the papers beforehand to get familiar with the works. Before the experiments, they learned OpenK through its documents. Among these students, four have some general understandings of stencil computations, the others do not. In the experiment, they are asked to put down the key insights or knowledge from the papers they read in the format required by OpenK, and upload them into OpenK.

Knowledge Representation and Accumulation. The experimental results show that in most cases, the time taken to represent and upload the knowledge in one paper is 10–20min (without counting the time in reading the papers). The longest time is 45min, which is for a theory-intensive paper with knowledge involving some mathematical derivations and model analysis. It is also the first paper the student worked on. According to the feedback from the students, the process needs only a short learning period, and the framework is overall convenient to use for representing and uploading knowledge from the papers.

The knowledge uploaded by the students are automatically integrated into a knowledge base by OpenK. Along with some previously uploaded optimization insights that were derived from a thesis on high performance stencil computations [12], it forms an ontology-based knowledge base for optimizations of stencil computations. Figure 6 shows some of the insights (in English for readability).

Knowledge Capitalization. To test the usefulness of the knowledge base, a student who has only some general understanding of stencil computations uses OpenK as an "optimization advisor" to optimize the performance of four stencils: a 3-D 7-point stencil on a 512^3-point grid and on a $1024^2 \times 256$ grid, and a 3-D 27-point stencil on the same two grids. The 7-point stencils are from a heating-process simulation, and the 27-point stencils are from Stencil Probe [20]. The target architecture is Intel Xeon E3-1200 (4-core 3.2GHz, 8G RAM, 32K L1, 256K L2 and 6MB L3).

Table 1: The speedup of different optimizations. The stars (*) mark the ones suggested by the OpenK (size-1: 512^3, size-2: $1024^2 \times 512$, CSE: common subexpression elimination).

Grid size	neighbor size	Cache blocking	SIMD, Array padding	Unrolling, CSE
size-1	7-p	10% *	≈ 0	≈ 0
size-1	27-p	≈ 0	13% *	30% *
size-2	7-p	22% *	≈ 0	≈ 0
size-2	27-p	8%	14% *	42.5% *

OpenK proves quite useful. Even though the particular architecture was not used in the knowledge base in OpenK, after the student inputs the computation patterns and the architecture model into OpenK, OpenK still provides several suggested optimizations likely suiting the target stencil computations on that architecture. Some of the suggested optimizations are those that have shown to hold across architectures for a certain kind of stencils (e.g., the second insight in Figure 6), some are inferred by the inference tool in OpenK across multiple insights based on the properties of the input stencil and the underlying architecture (e.g., the 6th and 7th rules in Figure 6 lead to the suggestion on optimizing for memory bandwidth if the input is a 7-point stencil to run on a system with less memory bandwidth than a similar system already tested and recorded in the knowledge base).

Table 1 reports the main set of optimizations the student has implemented. The stars ($*$) indicate the optimizations suggested by OpenK to the stencils. For comparison, the student implemented all three optimizations on all of the stencils. The results show that OpenK suggests the optimizations suitable to each stencil. The entire optimization process took less than 4 hours. In comparison, without OpenK, a user that has no clear directions to explore would need to investigate a wide range of possible code transformations, which, according to our interactions with the groups in DOE national labs, often takes at least several days to achieve a speedup comparable to what the student has achieved.

6.2 Case II: Enabling Cooperations Among Software Tools

Our second case study demonstrates the benefits of OpenK in promoting a synergy between different software tools. We use Liveness analysis in two compilers as the example. Liveness analysis is one of the primary data flow analyses in

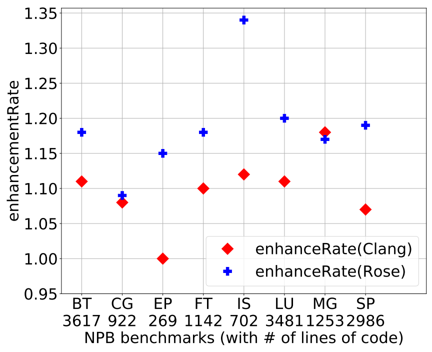


Figure 7: Improvement of the precision of Liveness analysis by cooperations enabled by OpenK between LLVM-Clang and ROSE compilers.

compilers. A variable is *live* at a point if there is at least one path leaving that point, along which, the variable is used before being redefined. Such information drives many program optimizations (e.g., dead store elimination). As a *may-type* data flow analysis, Liveness analysis is conservative—that is, if a variable belongs to the Live-out set of a basic block, it means that the compiler is uncertain whether the variable is dead at the end of the basic block. So, for two different Liveness analyses (both are sound and conservative), if a variable belongs to the result from one analysis but not the other, we can conclude that that variable is not live at the end of the said basic block. In another word, the intersection of the results of the two Liveness analyses gives a more precise result than either of them.

Our study uses two open-source compilers, LLVM-Clang [23] and ROSE [31]. They both have their own Liveness analyses developed before. However, they differ in the reported Live-out sets, and also the variable names and program locations used in their representations. Because OpenK can already represent an arbitrary C/C++ program in ontology, we write converters to map the the variable names and program locations used in the Live-out sets of both compilers to our ontology representation. With that, through Prolog, we develop a 65-line Prolog code that works on the ontology representation to easily extract out the intersection of the sets of Live-out variables reported by the two compilers for each basic block of a given program.

As the output (denoted as C) is the intersection of the outputs of two analyses, we define the precision enhancement over the results of one of the analyses A as

$$enhancement = \frac{|A|}{|C|}.$$

On the NAS Parallel Benchmarks (NPB) [4], the approach improves the average precision of Liveness analysis by over 10% and 20% for LLVM-Clang and ROSE respectively.

This case study demonstrates the benefits of OpenK in bridging the gap between separately developed software tools. Although it is possible that one may be able to combine the results of the two compilers through the development of an ad-hoc common representation, the representation needs to be designed and more over, code needs to be developed from scratch to map the results of two compilers to the common representation, to manipulate the representation, and to deal with their different code representations. In comparison, the ontology-based knowledge representation plus the existing facilities OpenK offers (e.g., converters of C/C++ programs to ontology, seamless integration with the Prolog engine) help ease the process. Furthermore, as the results are represented in ontology as well, they can be easily used with other ontology-represented knowledge (e.g., hardware or library APIs) for other uses.

6.3 Case III: GPU Data Placement

Our third case study demonstrates the benefits of OpenK in helping make software optimizers more extensible. The problem is GPU data placement. GPU has complex memory systems [10]. On NVIDIA Kepler GPUs, for instance, there are more than eight types of memory and cache with different performance characteristics. Determining which type of memory should hold which data object requires the knowledge about the program (such as data access patterns), hardware memory specifications and optimization rules.

Previous work uses customized memory specifications to express the needed architecture knowledge. For example, the PORPLE framework [10] uses a memory specification language (MSL) for the description of a memory system. The disadvantage is that it is only extensible for a new GPU with a similar architecture. For hardware with new attributes beyond the MSL syntax, the MSL syntax and its parser would need some non-trivial modifications.

In OpenK, the ontology representation is more generic and extensible. We don't need to design any specific syntax for hardware knowledge. Only new concepts and relations (or properties) in the hardware domain need to be added. Figure 8 illustrates part of the GPU ontology. Listing 1 shows the description of the Global Memory on the NVIDIA K20c GPU card. The ontology captures relations between the global memory and other components of the GPU and also the attributes of the memory. It is expressive and easy to extend. With the ontology-based representation of both program and hardware knowledge, it is easy to develop analyses to guide data placement optimizations on GPU.

OpenK also simplifies the combination of various methods for data placement. Data placement guidance can rely on heuristic rules, mathematical modeling, or their combinations. For instance, a previous study [19] makes the decisions based on some empirical rules on data access patterns and

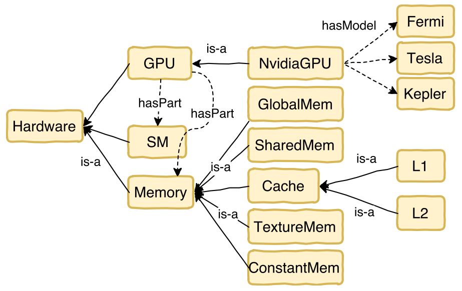


Figure 8: Part of the GPU hardware ontology.

Listing 1: ontology of Tesla K20c global memory

```
globalMem_k20c type GlobalMemory;
globalMem_k20c hasUpperLevel L2;
globalMem_k20c shareScope die;
globalMem_k20c pieces 1;
globalMem_k20c software_manageable true;
globalMem_k20c accessible "rw";
globalMem_k20c has_size 5032706048;
... % omitted
```

memory properties. An example rule is that constant memory is suitable for read-only data which is small enough and satisfies the *same address read* condition (i.e., memory accesses to the array are the same across all threads in a warp). PORPLE [10], on the other hand, uses analytical cost models for finding good placements of data. It enumerates possible placement plans and estimates the cost of each plan in terms of memory latency. It coordinates the placements of all arrays rather than considering them separately as the rule-based method does.

With OpenK, it is easy to combine the heuristic rules and the analytical models into a single analysis for guiding data placement. For example, the aforementioned rule for constant memory can be expressed in SWRL shown in Listing 2 in OpenK. The algorithm used in the previous data placement optimizer (PORPLE) can be written concisely as Listing 3 in the logic programming language Prolog to work on the OpenK knowledge base. It shows three Prolog rules. The ":" separates the *head* (left) and the *body* (right) of the rule; names in italic font are variables. The meaning of a rule is that the head is true if the all the phrases in the body are true.

The top-level algorithm in Prolog optimalPlace finds all the possible placement plans and estimates the cost of each plan. The possiblePlacement enumerates all possible plans. With the help of the Thea library [39], the SWRl rule for placing data into constant memory (suitableMem) is integrated into the Prolog algorithm to help narrow down the search space. The estimation of the cost of each placement is based on cache behaviors and is computing-intensive. OpenK implements the computing through a Prolog-based extension called *computable* [35].

Listing 2: SWRL rule for using constant memory (italic for variables)

```
isArray(x)∧readOnly(x)∧ConstantMem(y)
∧sizeFit(x,y)∧sameAccessWithinWarp(x)
-> suitableMem(x,y)
```

Listing 3: Algorithm of data placement written in Prolog

```
optimalPlace(BestPlacement) :-
findall(Placement,
(possiblePlacement(Placement),
hasCost(Placement, Cost)),
Placements),
minimalCost(Placements, BestPlacement).

possiblePlacement(Placement) :-
findall(Place,
(isArray(Array),
suitableMem(Array, Mem),
Place = place(Array, Mem)),
Placement).

hasCost(Placement, Cost) :-
hasCommand(hasCost, Cmd),
call(Cmd, Placement, Cost).
```

We measure the performance improvements that the suggested data placements by OpenK bring to a set of GPU programs. These programs have been used in prior studies [9, 10], including some from the RODINIA suite [9] and two kernels *glassForce, glassControl* from the DOE application LULESH [21]. The speedups over the original benchmarks are shown in Figure 9, agreeing with the speedups achieved using the previous imperative implementation of the placement analysis. The analysis time taken by OpenK is on average 5% longer than the imperative implementations do.

Compared to the previous MSL-based method [10], the OpenK-based analyses are more extensible and sharable due to their ontology representations and declarative algorithms. For example, in PORPLE, to add knowledge of some new GPUs memory features, the MSL format must be modified to add the corresponding fields; the code for parsing MSL and querying the placement engine should be revised as well. In OpenK, the changes are simpler: Through a graphic interface, users can easily add or remove a concept, and add queries on the new features into the analysis rules.

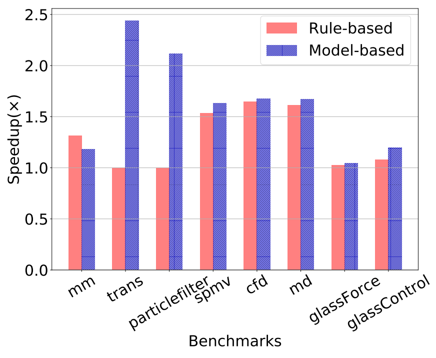


Figure 9: Speedup of benchmarks on Tesla K20c.

7 RELATED WORK

For its expressiveness, ontology has been adopted in many fields. In software engineering, software ontology (SWO) [26] is proposed for the meta information of software (e.g., licenses, publishing processes, data formats). A recent study uses ontology to simplify the development of program analysis tools [41]. Ontology has also been shown to be useful for domain-specific language development [24]. None of them is about building up an entire knowledge management infrastructure.

There have been some infrastructures developed to help manage the knowledge in some domains, including GoKB for the meta data of electronic library resources [1], clinical research [11], some for enabling scholarly discovery through interlinked profiles of people and other research-related information (e.g., VIVO [22]), some for scientific literature search (e.g., Semantic Scholar [38]), and some for identifying useful hypotheses and experiments in a scientific literature [33]. Although these search tools could help reduce the time a researcher needs to find literatures, they do not support systematic representation, accumulation, sharing and capitalization of the key knowledge contained in the literatures. Google Knowledge Graph [2] is a framework for representing some general knowledge (along with certain medical-domain knowledge). It is not an open infrastructure and lacks the necessary features (e.g., code representation, support to program optimizations) to fit the needs of HPC. OpenHPC.community [3] is a website dedicated to promote collaborations among the HPC community. But it is for aggregating common tools and libraries (e.g., provisioning tools, scientific libraries) for deploying and managing HPC Linux clusters, rather than for offering a general platform for HPC knowledge sharing.

There are also some related work on specific task. For example, the Program Database Toolkit (PDT) [25] builds a database of high level program information from source code to assist program analysis in IDEs. Instead, we build knowledge graph from source code which enables interaction with other knowledge like hardware and also enables logic inference. Another relevant work is the LighthouseHPC project [27], which is designed for the matrix algebra computation problem in HPC. It uses software ontology to guide optimization and also contains a systematic taxonomy for the matrix algebra problem.

To the best of our knowledge, OpenK is the first infrastructure designed to help the HPC community to systematically accumulate, share, and reuse HPC knowledge. It builds upon the recent advancements in knowledge engineering, while featuring a unique design for overcoming the special complexities in HPC, including the use of ontology as the representation to accommodate the large variety of knowledge in HPC, the extensive and modular design to support the many different domains in HPC, the versatile interfaces to enable the needed complex human and software interactions in optimizing HPC programs, and the leverage of the ontology ecosystem that has been rapidly built in the past decade.

8 CONCLUSION

In this paper, we have introduced OpenK, the first infrastructure that offers some systematic support to help the HPC community accumulate, share, and reuse various HPC knowledge. Centered around ontology, its design makes it able to represent different sources and types of HPC knowledge in a coherent format, amenable for logic reasoners and other inference tools to effectively guide software optimizations on complex, and continuously changing HPC architectures. Experiments demonstrate the promise of OpenK for facilitating HPC in improving the productivity in code optimizations and enhancing the extensibility of software optimizers. We plan to release the infrastructure to the public to help promote the synergy among the HPC community.

9 ACKNOWLEDGEMENT

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and was partially supported by the LLNL-LDRD Program under Project No. 18-ERD-006. IM release number: LLNL-TR-809546.

REFERENCES

[1] Global open knowledgebase. http://gokb.org/.
[2] Google knowledge graph. https://www.google.com/intl/es419/insidesearch/features/search/knowledge.html.
[3] Openhpc.community. http://www.openhpc.community/.
[4] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. Technical report, The International Journal of Supercomputer Applications, 1991.

[5] N. Bassiliades, G. Antoniou, and I. Vlahavas. A defeasible logic reasoner for the semantic web. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 2(1):1–41, 2006.

[6] M. Bostock, V. Ogievetsky, and J. Heer. D^3 data-driven documents. *Visualization and Computer Graphics, IEEE Transactions on*, 17(12):2301–2309, 2011.

[7] I. Bratko. *Prolog (3rd Ed.): Programming for Artificial Intelligence.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[8] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: Implementing the semantic web recommendations. In *Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers &Amp; Posters*, WWW Alt. '04, pages 74–83, New York, NY, USA, 2004. ACM.

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.

[10] 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*, Washington, DC, USA, 2014.

[11] C. D. Pierce, D. Booth, C. Ogbuji, C. Deaton, E. Blackstone, and D. Lenat. SemanticDB: A Semantic Web Infrastructure for Clinical Research and Quality Reporting. *Current Bioinformatics*, 7(3):267–277, 2012.

[12] K. Datta. *Auto-tuning Stencil Codes for Cache-Based Multicore Platforms.* PhD thesis, EECS Department, University of California, Berkeley, Dec 2009.

[13] T. A. Foundation. Apache solr. http://lucene.apache.org/solr/, 2016.

[14] 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.

[15] B. Glimm, I. Horrocks, B. Motik, G. Stoilos, and Z. Wang. Hermit: an owl 2 reasoner. *Journal of Automated Reasoning*, 53(3):245–269, 2014.

[16] P. Hitzler, M. Krötzsch, B. Parsia, P. F. Patel-Schneider, and S. Rudolph. OWL 2 web ontology language primer. *W3C recommendation*, 27(1):123, 2009.

[17] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, M. Dean, et al. Swrl: A semantic web rule language combining owl and ruleml. *W3C Member submission*, 21:79, 2004.

[18] S. Huss-Lederman, B. Gropp, A. Skjellum, A. Lumsdaine, B. Saphir, J. Squyres, et al. Mpi-2: Extensions to the message passing interface. *University of Tennessee*, 1997.

[19] B. Jang, D. Schaa, P. Mistry, and D. Kaeli. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Trans. Parallel Distrib. Syst.*, 22(1):105–118, Jan. 2011.

[20] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In *Proceedings of the 2005 Workshop on Memory System Performance*, MSP '05, pages 36–43, New York, NY, USA, 2005. ACM.

[21] I. Karlin, A. Bhatele, and B. L. LULESH programming model and performance ports overview. Technical Report LLNL-TR-608824, Dec. 2012.

[22] D. B. Krafft, N. a. Cappadona, B. M. Devare, B. J. Lowe, and J. Corson-rikert. VIVO : Enabling National Networking of Scientists. *Technology*, 2010.

[23] C. Lattner. LLVM and Clang: Next generation compiler technology. In *The BSD Conference*, pages 1–2, 2008.

[24] C. Liao, P.-H. Lin, D. J. Quinlan, Y. Zhao, and X. Shen. Enhancing domain specific language implementations through ontology. In *Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, WOLFHPC '15, pages 3:1–3:9, New York, NY, USA, 2015. ACM.

[25] K. A. Lindlan, J. Cuny, A. D. Malony, S. Shende, F. Juelich, R. Rivenburgh, C. Rasmussen, and B. Mohr. A tool framework for static and dynamic analysis of object-oriented software with templates. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, 2000.

[26] J. Malone, A. Brown, A. L. Lister, J. Ison, D. Hull, H. Parkinson, and R. Stevens. The software ontology (SWO): A resource for reproducibility in biomedical data analysis, curation and digital preservation. *Journal of Biomedical Semantics*, 5(1):25, 2014.

[27] B. Norris, S.-L. Bernstein, R. Nair, and E. R. Jessup. Lighthouse: A user-centered web service for linear algebra software. *CoRR*, abs/1408.1363, 2014.

[28] N. F. Noy and D. L. McGuinness. Ontology development 101: A guide to creating your first ontology, 2001.

[29] C. Nvidia. Nvidia CDUA C programming guide. *NVIDIA Corporation*, 120:18, 2011.

[30] M. Proctor. Drools: a rule engine for complex event processing. In *Applications of Graph Transformations with Industrial Relevance*, pages 2–2. Springer, 2011.

[31] D. Quinlan and C. Liao. The ROSE source-to-source compiler infrastructure. In *Cetus Users and Compiler Infrastructure Workshop, in conjunction with PACT 2011*, Galveston Island, Texas, USA, Oct. 2011.

[32] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical owl-dl reasoner. *Web Semantics: science, services and agents on the World Wide Web*, 5(2):51–53, 2007.

[33] S. Spangler, J. N. Myers, I. Stanoi, L. Kato, A. Lelescu, J. J. Labrie, N. Parikh, A. M. Lisewski, L. Donehower, Y. Chen, O. Lichtarge, A. D. Wilkins, B. J. Bachman, M. Nagarajan, T. Dayaram, P. Haas, S. Regenbogen, C. R. Pickering, and A. Comer. Automated hypothesis generation based on mining scientific literature. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '14*, pages 1877–1886, New York, USA, aug 2014.

[34] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.

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

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

[37] M. Uschold, M. Gruninger, et al. Ontologies: Principles, methods and applications. *Knowledge engineering review*, 11(2):93–136, 1996.

[38] M. Valenzuela, V. Ha, and O. Etzioni. Identifying meaningful citations. 2014.

[39] V. Vassiliadis, J. Wielemaker, and C. Mungall. Processing OWL2 ontologies using Thea: An application of logic programming. In *PROCEEDINGS OF THE 5TH INTERNATIONAL WORKSHOP ON OWL: EXPERIENCES AND DIRECTIONS*, 2009.

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

[41] Y. Zhao, G. Chen, C. Liao, and X. Shen. Towards ontology-based program analysis. In *ECOOP 2016–Object-Oriented Programming*. 2016.