ABSTRACT

ZHAO, YUE. Lowering Knowledge Barriers for HPC: From Representations to Infrastructures and Acquisitions. (Under the direction of Xipeng Shen.)

With software becoming more and more complex and hardware evolving rapidly, tasks in the high performance computing (HPC) become increasingly complex. As a result, modern HPC development requires multiple levels of knowledge, ranging from application domain expertise to algorithms, numerical analysis, software systems, and computer architecture.

This thesis work examines HPC development from a knowledge engineering perspective, pointing out that the essential barrier for modern HPC development is on how to effectively acquire, represent, accumulate, share and exploit HPC knowledge.

To lower the barrier, this work explores the novel techniques in three aspects. First, it investigates the use of Ontology for HPC knowledge representation in the task of program analysis and optimizations. Second, it discusses the construction of an infrastructure named OpenK for HPC knowledge management and exploitation. Finally, it explores a set of techniques to bridge the gap between machine learning and HPC for knowledge acquisition and capitalization.
Lowering Knowledge Barriers for HPC: From Representations to Infrastructures and Acquisitions

by

Yue Zhao

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2018

APPROVED BY:

Min Chi

Guoliang Jin

Chunhua Liao
External Member

Huiyang Zhou

Xipeng Shen
Chair of Advisory Committee
DEDICATION

To my wife.
BIOGRAPHY

Yue Zhao was born in ShanXi, China. He earned his Bachelor’s degree in Information Engineering from the Shanghai Jiao Tong University in 2009 and his M.Phil. (Master of Philosophy) degree from the Hong Kong Polytechnic University in 2012.

Since the Fall of 2013, he began to work with Dr. Xipeng Shen as a Ph.D. student. His research interest mainly resides in the high performance computing domain, with a focus on leveraging knowledge engineering and machine learning techniques to assist the program analysis and optimization.
ACKNOWLEDGEMENTS

First, I want to thank my advisor, Dr. Xipeng Shen, for his guidance and support throughout my Ph.D. studies. His academic advice and patience is essential to the completion of my Ph.D. degree. Moreover, his enthusiasm and attitude to the research sets a positive example beyond the completion of my Ph.D studies.

Next, I would like to thank the rest of my dissertation committees – Dr. Min Chi, Dr. Guoliang Jin, Dr. Chunhua Liao, Dr. Frank Mueller and Dr. Huiyang Zhou – for their time and effort to attend my exams. Their insightful comments and suggestions are important to this thesis.

My sincere thanks also go to Dr. Chunhua Liao, for his mentoring during my internship in the Lawrence Livermore National Laboratory and our collaboration.

I will not forget to extend my gratitude to Dr. George Rouskas, Ms. Kathy Luca and all others in the graduate school for their helps during my Ph.D study.

In addition, I would like to thank all other friends including but not limited to my group mates and colleagues for their help.

Last but not the least, I would like to thank my family – my parents and especially my wife Weijie Zhou – for their love, support and encouragement.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF TABLES</th>
<th>viii</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>ix</td>
</tr>
</tbody>
</table>

## I Prologue

### Chapter 1 Introduction

1.1 Motivation .................................................. 2
1.2 Overview of the Proposed Solutions ........................ 3

## II Ontology-based Knowledge Engineering for High Performance Computing

### Chapter 2 Introduction to Ontology and Logic Programming

2.1 Ontology .................................................. 6
2.2 Logic Programming ........................................ 7

### Chapter 3 PATO - Program Analysis through Ontology

3.1 Problem Statement .......................................... 10
3.2 Overview of the PATO System .............................. 13
3.3 Challenges and Solutions .................................. 16
3.3.1 Ontology Design ........................................ 17
3.3.2 Knowledge Generation ................................... 19
3.3.3 Knowledge Utilization ................................ 21
3.3.4 Framework Design and PATO ............................ 22
3.4 Experience with PATO ...................................... 24
3.4.1 Canonical Loop Analysis ............................... 24
3.4.2 Pointer Analysis ........................................ 27
3.4.3 Other Experience ....................................... 29
3.5 Related Work ............................................... 30

### Chapter 4 OpenK - An Open Infrastructure for the Accumulation, Sharing and Reuse of HPC Knowledge

4.1 Problem Statement .......................................... 32
4.2 Scope and Design Considerations .......................... 35
4.3 Overall of the OpenK System .............................. 36
4.4 Internals of OpenK ......................................... 39
4.4.1 HPC Knowledge Representation ........................ 39
4.4.2 HPC Knowledge Submission ............................. 41
4.4.3 HPC Knowledge Derivation .............................. 42
4.4.4 Knowledge Quality Control ............................. 43
4.4.5 HPC Knowledge Capitalization ........................ 43
4.5 Experience with OpenK ................................................................. 45
4.5.1 Case I: Stencil Computation Optimizations ............................. 45
4.5.2 Case II: Enabling Cooperations Among Software Tools .......... 48
4.5.3 Case III: GPU Data Placement ............................................... 50
4.6 Related Work ............................................................................ 53

III Knowledge Acquisition and Capitalization through Machine Learning 56

Chapter 5 Sparse Matrix Format Selection For SpMV .......................... 57
5.1 Problem Statement ................................................................. 57
5.2 Background ............................................................................ 58
5.2.1 Sparse Matrix Storage Format ............................................. 58
5.2.2 Convolutional Neural Networks (CNN) .................................. 59

Chapter 6 Bridging the Gap between Deep Learning and Sparse Matrix Format Selection 62
6.1 Motivation .............................................................................. 62
6.2 Overview of the CNN based Solution ...................................... 64
6.3 Input Representations ............................................................ 65
6.3.1 Representation from Traditional Methods ............................. 65
6.3.2 Augmentation with Density Representation ......................... 66
6.3.3 Distance Histogram Representation .................................... 67
6.4 CNN Structure Designs ........................................................ 68
6.4.1 Insufficiency of Common Structures ................................... 68
6.4.2 Late-merging Structure ....................................................... 69
6.5 Cross-Architecture Adaptations .............................................. 70
6.5.1 Concept and Two-fold Contributions .................................. 70
6.5.2 Options Explored ............................................................... 71
6.6 Evaluations ............................................................................ 72
6.6.1 Methodology ..................................................................... 72
6.6.2 Speedup ............................................................................ 75
6.6.3 Model Migrations ............................................................... 76
6.6.4 Impact of CNN Structures .................................................. 77
6.6.5 Discussion ........................................................................ 78
6.7 Insights .................................................................................. 79
6.8 Related Work ......................................................................... 79

Chapter 7 Overhead-Conscious Sparse Matrix Format Selection .......... 81
7.1 Motivation .............................................................................. 81
7.2 Design Choices and Challenges .............................................. 83
7.2.1 Implicit Versus Explicit Treatment ..................................... 83
7.2.2 Challenges ........................................................................ 85
7.3 Overhead-Conscious Predictor ................................................ 86
7.3.1 Learning Method ............................................................... 86
7.3.2 Two-Stage Lazy-and-Light Scheme ................................... 87
7.3.3 The Second-Stage Prediction and Feature Selection 89
7.4 Evaluations 91
   7.4.1 Methodology 91
   7.4.2 Impact of the Overhead on Format Selection 92
   7.4.3 Performance Comparison of Primary Predictors 93
   7.4.4 Performance on Entire Applications 95
7.5 Related work 99

IV Epilogue 101

Chapter 8 Conclusions and Future Directions 102
   8.1 Conclusions 102
   8.2 Future Studies 103

BIBLIOGRAPHY 105
### LIST OF TABLES

| Table 3.1 | Constraints in Andersen’s pointer analysis. | 28 |
| Table 4.1 | The speedup of different optimizations. The stars (*) mark the ones suggested by the OpenK (size-1: 512², size-2: 1024² × 512, CSE: common subexpression elimination). | 48 |
| Table 6.1 | Hardware platforms used in the experiments. | 72 |
| Table 6.2 | Prediction quality on Intel® CPU (bottom decimals show overall accuracies). | 74 |
| Table 6.3 | Prediction results on the GPU platform. | 75 |
| Table 7.1 | The full set of feature candidates of a matrix. | 90 |
| Table 7.2 | Hardware platforms used in the experiments. | 91 |
| Table 7.3 | The conversion (CSR to other formats) time normalized by a single SpMV on CSR. | 93 |
| Table 7.4 | The number of matrices that favor each of the formats in the conversion overhead-oblivious (OO) and overhead-conscious cases (when the loop has 100 or 1000 iterations). | 93 |
| Table 7.5 | Prediction errors of normalized format conversion time and SpMV time | 94 |
| Table 7.6 | Speedup of applications. | 98 |
| Table 7.7 | The numbers of matrices that favor each of the formats in the four applications. | 98 |
| Table 7.8 | Speedup comparison for selected matrices. (PR: PageRank, Sp: Speedup) | 98 |
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 2.1</td>
<td>An example ontology on Robotics.</td>
<td>7</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>Overview of using ontology for program analysis.</td>
<td>14</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Specification of a canonical loop (derived from OpenMP manual [Ope13]).</td>
<td>15</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Top-level ontology for C program analysis.</td>
<td>18</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>Structure of PATO.</td>
<td>22</td>
</tr>
<tr>
<td>Figure 3.5</td>
<td>Time comparison of the canonical loop analysis. The X-axis shows the benchmark names and their numbers of source-code lines.</td>
<td>27</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Overall structure of OpenK.</td>
<td>36</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>Part of the HPC knowledge taxonomy.</td>
<td>40</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>An example of stencil computations.</td>
<td>45</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>Part of the ontology of stencil computation optimizations.</td>
<td>46</td>
</tr>
<tr>
<td>Figure 4.5</td>
<td>Some insights on optimizing stencil computations.</td>
<td>47</td>
</tr>
<tr>
<td>Figure 4.6</td>
<td>Enhancement of Liveness analysis by leveraging ontology to combine the results from the Liveness analysis in LLVM Clang and ROSE. (Dots show outliers). Left bars: the enhancement over LLVM Clang results; right bars: the enhancement over ROSE results.</td>
<td>50</td>
</tr>
<tr>
<td>Figure 4.7</td>
<td>Part of the GPU hardware ontology.</td>
<td>51</td>
</tr>
<tr>
<td>Figure 4.8</td>
<td>Speedup of benchmarks on Tesla K20c.</td>
<td>54</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>Sparse matrix storage formats and their corresponding SpMV pseudo-code (adapted from [Li13]).</td>
<td>59</td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>Illustration of CNN.</td>
<td>61</td>
</tr>
<tr>
<td>Figure 6.1</td>
<td>Overview of the model construction process.</td>
<td>64</td>
</tr>
<tr>
<td>Figure 6.2</td>
<td>Image scaling loses some subtle but critical information for matrices.</td>
<td>66</td>
</tr>
<tr>
<td>Figure 6.3</td>
<td>The density representation and histogram representation of the matrix in Figure 6.2(a).</td>
<td>66</td>
</tr>
<tr>
<td>Figure 6.4</td>
<td>Traditional CNN structure that merges the info from different channels at the early stage of the networks.</td>
<td>68</td>
</tr>
<tr>
<td>Figure 6.5</td>
<td>Proposed late-merging CNN structure.</td>
<td>69</td>
</tr>
<tr>
<td>Figure 6.6</td>
<td>Speedups from CNN over DT-based predictions.</td>
<td>76</td>
</tr>
<tr>
<td>Figure 6.7</td>
<td>Prediction accuracies of different retraining methods on a new platform (AMD® platform). (Accuracies at x=0 are without retraining.)</td>
<td>77</td>
</tr>
<tr>
<td>Figure 6.8</td>
<td>CNN late-merging structure (layer shapes in parentheses).</td>
<td>77</td>
</tr>
<tr>
<td>Figure 6.9</td>
<td>Comparison of the loss function convergence for two different structural designs.</td>
<td>78</td>
</tr>
<tr>
<td>Figure 7.1</td>
<td>The typical workflow for an application to benefit from improved matrix formats, and the objectives targeted by the previous studies and by this work.</td>
<td>82</td>
</tr>
</tbody>
</table>
Figure 7.2  The histogram of the overall speedups of program PageRank when it uses a previous decision tree-based predictor with an idealized 100% accuracy (for minimizing $T_e$ in Figure 7.1). The annotated numbers indicate percentage of samples with speedup $<1$, $=1$, $>1$ respectively; $<1$ means slowdown. CSR is the default format. ................................................................. 83

Figure 7.3  The two-stage prediction system. ................................................................. 87

Figure 7.4  The comparison of speedups based on $SpMV_frame$. The baseline is the performance on the default CSR format. ................................................................. 95

Figure 7.5  Feature ranking for $T_{spmv}$ of DIA format. ...................................................... 96

Figure 7.6  Feature ranking for $T_{spmv}$ of BSR format. ...................................................... 96

Figure 7.7  The histogram of the overall speedups of program PageRank when it uses our OC format selector predictor. CSR is the default format. ................................. 99
Part I

Prologue
CHAPTER

1

INTRODUCTION

1.1 Motivation

With software and hardware systems being more and more complex, modern HPC development is increasingly challenging. Tasks in HPC such as program analysis and optimization usually require multiple levels of knowledge ranging across software, hardware and domain expertise. However, there exists a barrier that hinder the full utilization of the wide range of HPC knowledge. This thesis points out that the essential barrier is on how to effectively acquire, represent, accumulate, share and exploit HPC knowledge. Therefore, the major focus of this thesis is on how to lower this barrier.

To address the problem, this thesis explores novel techniques in three aspects. First, this thesis proposes to leverage the knowledge engineering techniques to standardize and represent the wealth of information in the HPC community. Second, this thesis discusses the experience to construct the infrastructure to facilitate the knowledge management in the HPC domain. Last, this thesis shows that machine learning is a viable complementary approach for acquisition and capitalization of the HPC knowledge. It can be used to synthesize raw or semantically-enriched content to yield insights. With the sparse matrix format selection problem as a case study, this thesis inspects the special challenges and explores the essential techniques to bridge the gap between the machine learning and HPC tasks.

The following section highlights the scope and major topics covered in this thesis.
1.2 Overview of the Proposed Solutions

Knowledge engineering through ontology

In high performance computing (HPC), program analysis is fundamental for program optimization, debugging, and many other tasks. Developing program analyses is challenging and error-prone. To ease the program analysis, a promising methodology is the declarative program analysis. However, current declarative program analysis is subject to some critical limitations in supporting cooperation among analysis tools and guiding program optimizations. It also often requires much effort for repeated program preprocessing.

To break through these limitations, we leverage the ontology-centered knowledge engineering to standardize the definitions of concepts and the representation of the knowledge in the domain. As a result, we developed a prototype called PATO for conducting program analysis based on the ontology-based program representation.

Furthermore, for HPC applications, applying the right optimizations in the appropriate context often requires a wide range of knowledge about the domain, the software application and the hardware architecture. However, in today’s practice, such knowledge has been expressed and managed in an ad-hoc manner by each individual. With software and hardware evolving rapidly, such ad-hoc representation of the HPC knowledge becomes a barrier to accumulate and reuse the HPC knowledge.

As an attempt to lower the barrier, we propose the OpenK system as an open infrastructure for the systematic accumulation, sharing and reuse of the knowledge across the HPC domain. The foundation of the OpenK centers around the ontology-based knowledge engineering.

In Chapter 3 and Chapter 4, we discuss the challenges and solutions of designing the PATO system and OpenK system, respectively. Our experience with the two systems confirm the potential of ontology for complementing existing declarative program analysis, promoting cooperative analysis between compilers, and effectively guiding optimizations.

Knowledge engineering through machine learning – the case study of the matrix format selection for SpMV

Sparse matrix vector multiplication (SpMV) is an important kernel in many applications and is often the major performance bottleneck. Among many optimizations, the storage format of sparse matrices critically affects the performance of the SpMV. The sparse matrix format selection problem is to determine the best storage format for a matrix to maximize the performance of the SpMV. However, due to the complex interactions between the SpMV application, the runtime system and other factors, this task is challenging for programmers.

To assistant programmers in making decisions, we exploit machine learning techniques to
empirically understand the complex systems. From the viewpoint of the knowledge engineering, however, there exists gap between the machine learning techniques and the effective knowledge acquisition and capitalization for the problem.

In Chapter 6, we present a systematic exploration on the promise and special challenges of deep learning for the SpMV format selection problem. We describe how to effectively bridge the gap between deep learning and the special needs of the HPC problem through a set of techniques on matrix representations, deep learning structures, and cross-architecture model migrations. The proposed solution cuts format selection errors by two thirds, and improves SpMV performance by 1.73× on average over the state-of-the-art work.

Previous studies on selecting the appropriate format for a given matrix have ignored the influence of runtime prediction overhead and format conversion overhead. For many common uses of SpMV, such overhead is part of the execution time and may outweigh the benefits of new formats. Ignoring them frequently makes the predictions sub-optimal and sometimes inferior. On the other hand, the overhead is difficult to consider, as it, along with the benefits of having a new format, varies from matrix to matrix, and from application to application.

To address the overhead issue, Chapter 7 explores the pros and cons of various possible treatments to the overhead and presents an explicit approach to solve it. The core of the solution is a set of regression models for learning the influence of the overhead and the benefit of new format on the overall program performance. The proposed two-stage scheme can help control the risks in the format predictions and at the same time maximize the overall format conversion benefits. Experiments show that the technique outperforms previous techniques significantly. It improves the overall performance of applications by 1.14X to 1.43X, significantly larger than the 0.82X to 1.24X upperbound speedups overhead-oblivious methods could give.
Part II

Ontology-based Knowledge Engineering for High Performance Computing
2.1 Ontology

Ontology is a concept originating in Philosophy, referring to the study of the nature of being, as well as the basic categories of being and their relations [NM01]. In recent decades, it has become a branch in Information Science for representing the knowledge in a particular domain. In this context, an ontology is a formal explicit description of a domain's knowledge, including concepts (or classes), properties of each concept (or relations) and individuals (or instances of classes) [SS13].

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 2.1 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 2.1, 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) [Hit09]. 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 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").

Developing an ontology for a domain has many benefits, including 1) making domain knowledge explicit to expose what is known and what is unknown, 2) enabling knowledge interoperability by providing a common taxonomy and vocabulary, 3) providing knowledge reuse since the ontology is a persistent knowledge base, and 4) facilitating knowledge validation and reasoning using existing inference engines (or reasoners).

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 [Gen03], SWI-Prolog Semantic Web Library [Wie12], and HermiT [Gli14]) 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.

2.2 Logic Programming

The theory foundation of ontology is Description Logic (DL) [Krö12], a family of formal knowledge representation languages for formal reasoning on the concepts of a domain. DL is expressive enough to build sophisticated knowledge bases while still supporting efficient inference. DL allows the use
of axioms to describe a knowledge base. For instance, an axiom Person(\textit{Alex}) describes that “Alex” is an instance of class “Person”, and Person $\sqsubseteq$ Human describes the subsumption relationship between concept Person and concept Human. DL languages could use different grammars. Conventionally, a simple yet uniform format to represent axioms is (subject, property, object) triple. Axioms in DL languages can all be mapped to such a format. For instance, in Person(\textit{Alex}), “Alex” is the subject, “instance of” is the property, and “Person” is the object. Visually, it is an “instance of” edge flowing from the “Alex” node to the “Person” node in an ontology graph.

Through decades of development, a large body of tools (e.g., Stanford Protégé [Gen03], SWI-Prolog Semantic Web Library [Wie12], etc.) 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.

Like declarative program analysis, most of these tools leverage logic programming languages (e.g., Prolog) for inferences.

In Prolog, the program logic is expressed in terms of relations, represented as facts and rules, and a computation is a query over these relations. To begin, the basic concepts in Prolog are the \textit{term} and \textit{atom}. Term is the single data type in Prolog. It can be a constant or a variable \footnote{By conversion, terms beginning with lowercase are constants and terms beginning with uppercase are variables.}. An \textit{atom} is a predicate with a list of terms as the arguments. It has the form of $p(X_1, X_2, \ldots, X_n)$, where $p$ is a predicate symbol that represents some statement and $X_i$ are terms. A \textit{ground atom} or \textit{fact} is a predicate with all its arguments constant. In Prolog, atoms are a way to describe relations and a set of facts form an instance of database.

In addition, Prolog rules can express logical inferences. A rule is of the form

$$A : - B_1, B_2, \ldots, B_n.$$ which reads “$B_1$ and $B_2$ and ... and $B_n$ implies $A$”. The form also suggests how computation should be carried out as “in order to compute $A$, we first need to compute $B_1, \ldots$”. The Prolog rule can describe recursive relations. As an example, List 2.1 shows the rule to compute the \textit{reachability} relation between all pairs of nodes in a graph. The link defines a set of facts somewhere else.

A Prolog program is a collection of rules, which can be applied to an instance of the database

\begin{verbatim}
Listing 2.1 Example of Prolog rules
1 reachable(X, Y) :- link(X, Y).
2 reachable(X, Y) :- link(X, Y), reachable(Z, Y).
3 % the following is a query
4 reachable(X, Y)?
\end{verbatim}
(i.e., a set of facts). The results are the set of facts inferred by applying the rules.

The features that the Prolog can define recursive relations and recursive queries naturally make it well-suited for querying graphs and relational structures. Besides, the high level abstraction capability and the simple syntax help rapid prototyping the complex analysis system. In our problem, we can describe the logical semantics of the operations in the dataflow model as facts or rules, and the analysis algorithm as the logic inference rules.

The Prolog works in a symbolic way. The predicates and terms alone are not meaningful to the system. To encode semantics in Prolog, we need to define a glossary of symbols to use (referred to as the “dictionary”). The semantics of the Prolog program are reflected by both the relations described by the predicates and the rules constraining the relations.
CHAPTER 3

PATO - PROGRAM ANALYSIS THROUGH ONTOLOGY

3.1 Problem Statement

Program analysis [Nie04] is a common way for deriving various properties of a program from its code. It is fundamental for many aspects of modern computing, including program optimizations, vectorization and parallelization, performance or correctness bug identification, task scheduling, and so on.

There are mainly two ways to implement a program analysis. A traditional way is imperative, in which, thousands of lines of code (often in some imperative programming languages) is developed based on some compiler framework for analyzing program constructs, types, control or data flows to infer certain properties of the target program. Programmers typically need to go through some steep learning curve about the structure and internal details of a complex compiler, while the results are often unsatisfactory: The code is often difficult to maintain, and bugs are common [Yan11]. The analysis, being specific to a particular compiler, is hard to extend, to compose, or to reuse for other compilers.

The second approach, *declarative program analysis*, has been proposed to overcome the productivity issues [Ull88; Hor95; Daw96]. With it, the developers just need to define some abstract domains and then use some logic programming language (e.g., Datalog [WL04]) to describe the analysis rules
that govern the relations or properties of interest. Some automatic tools can then automatically do
the inferences over a certain representation of some relations in the target program to find out the
wanted relations or properties of the program. Experiments have shown that, with this approach, the
code size of a program analysis often reduces by orders of magnitude compared to the imperative
approach, and the analyses become easier to maintain and extend [BS09]. Moreover, with the sub-
stantial improvement in optimizations of the logic processing engines (e.g., bddbddd [Wha05] and
numerous optimizations to inference engines [Wie12]), the performance and scalability concerns of
declarative program analyses have been largely resolved.

To further improve this promising paradigm of program analysis, particularly, we need to inves-
tigate solutions to three most important limitations of the current declarative program analysis:

• **Cooperations.** By expressing the analysis at a high level, different analysis tools could po-
tentially reuse an analysis, and different analyses could get composed together into a more
sophisticated analysis. However, in practice, these benefits have been difficult to achieve
in general, due to the differences in the analysis-specific representations of programs and
relations. In one analysis, the domain may be variable names and heap addresses, and the
relation may be “assigning one address to a variable”; in another analysis, the domain may be
expressions and the relation may be “calculated before a program point”. To compose the two
analyses, the variable names and heap addresses in the first analysis may have to be mapped
to the expressions in the second domain, which would require much code development (likely
entwined with the code in the compilers), especially if the two analyses were developed by
different users based on different compilers that use different intermediate representations
(IRs).

• **Optimizations.** So far, explorations of declarative program analysis have been focused on
understanding program behaviors (largely for the purpose of debugging), for which, program-
level knowledge has been enough. It is, however, insufficient for another important purpose of
program analysis, guiding program optimizations. For the multi-facet dependence of perfor-
mance, program optimizations often need knowledge from various sources: the program itself,
the hardware, the algorithms, the program input datasets, and various domain-specific or
problem-specific knowledge. Consequently, to provide useful optimization guidance, declar-
ative program analysis must support the representations of the various kinds of knowledge,
and allow easy linkage among them, even if the various kinds of knowledge may come from
different sources. The analysis-specific nature of the current declarative analysis designs offers
poor support to these needs.

• **Preprocessing.** A declarative program analysis typically requires some preprocessing to extract
useful relations from the target programs to build up a relational database. This step hurts
the productivity benefits of this approach: As it is usually tightly coupled with some compiler framework, it is tedious and error-prone to develop. What makes this especially problematic is that different program analyses often use different relations or ways to define the same or similar relations. As a result, preprocessing needs to be developed for almost every newly developed program analysis, seriously throttling the productivity benefits of declarative program analysis.

In our work, we advocate using Ontology-based knowledge engineering to address the three limitations all together. Our proposal comes from the observation that all the three major limitations essentially stem from a single fundamental shortcoming in current declarative program analysis: the lack of a systematic conceptual framework to govern the definition, representation, and organization of the various kinds of knowledge (relations in a program, rules, domains, hardware configurations, etc.) related with program analysis. The ad-hoc analysis-specific approach used in today's designs of declarative program analysis is the fundamental reason for the much effort required for preprocessing, and the barriers for supporting cooperations and optimizations.

The key idea in our proposal is to leverage ontology to help standardize the definitions of domains, relations, and other concepts in program analysis, and to establish a single flexible representation of program constructs as well as other kinds of knowledge related with program analysis and its usage. With that, knowledge from various sources may be linked seamlessly as long as they follow the standardized representation. Sharing the same conceptual framework and set of terminology, different program analyses will be easy to compose and inter-operate together. The standardization will also make it possible to develop a single comprehensive database of the relations of a program to serve for various program analyses, removing the needs for the separate development of preprocessing for each analysis.

Ultimately, it would be desirable to establish a standard Program Analysis ontology to describe programs, analysis, and related concepts—liken how the Semantic Sensor Network ontology by W3C [Com12] facilitates the work in the sensor network domains. Reaching that goal would require the coordinated effort from the community and goes beyond the scope of this paper.

This work first focuses on the following four-fold objectives:

- To introduce the idea of integrating ontology into program analysis, and explain the concept of ontology-based program analysis and its potential benefits.
- To investigate the feasibility of having a single representation of a program in ontology to facilitate various program analyses and hence reduce the much effort for developing program preprocessing as required in current declarative program analysis.
- To validate the promise of ontology as the representation of various kinds of knowledge related to program optimizations, and hence extend existing declarative program analysis to guide
program optimizations.

- To confirm the benefits of ontology for facilitating easy cooperations of different analysis tools.

To reach the four objectives, we have developed a prototype framework named PATO (which stands for Program Analysis Through ontology). In this prototype, we explore the use of several principles to define a concept-proof ontology for C program representations. Based on it, we have developed five program analyses: canonical loop analysis, pointer analysis, control flow graph construction, data access pattern analysis, and GPU data placement guidance. These analyses differ in domains, relations, scopes, and intended usage. Our experiments show that a single ontology-based representation can successfully support all these analyses (without separate preprocessing per analysis). The analyses inherit the productivity benefits of declarative program analysis, reducing the lines of code by tens of times compared to imperative implementations. Using the liveness analysis on two compilers (ROSE [Ros] and LLVM [LA04]), we confirm the benefits of ontology for promoting cooperations among different analysis tools. And using GPU data placement optimization, we demonstrate the seamless linkage of various sources of knowledge (programs, domain experts, and hardware) enabled by ontology, and reveal the potential of ontology-based program analysis for guiding program optimizations.

3.2 Overview of the PATO System

By offering a generic way to represent the knowledge in a domain, ontology simplifies the accumulation and share of various kinds of knowledge among people or software agents. At the same time, it allows automatic analysis and utilization of knowledge through high-level declarative logic programming, thanks to its description logic foundation. These properties make it potentially valuable for facilitating program analysis which is essentially about reasoning about the knowledge related to a program.

Figure 3.1 illustrates the basic idea of ontology-based program analysis. It centers around a knowledge base built upon ontology. The knowledge base may consist of the basic knowledge about the code of the target program, as well as other knowledge (e.g., architecture attributes) relevant to the program analysis. An ontology converter, equipped with a parser, derives the basic program knowledge from the code of the program, expresses the knowledge in a standard format, and puts it into the knowledge base. This basic knowledge may include the structures and components (control blocks, data structures, etc.) of the program. In addition, the knowledge base may include some knowledge that could be imported about some libraries, or directly input by a domain expert about some properties of the program or the hardware it executes on (for optimizations).
Built upon description logic, ontology-based program analysis keeps the conveniences of declarative program analysis. Rather than writing thousands of lines of code inside a complex compiler, users can simply write some logic queries about the kind of properties (e.g., which loops are canonical loops) of the program that they want to know. These queries should follow some ontology query APIs. The APIs will then return the answers that are automatically obtained from the ontology-based knowledge base. For complex queries, it can leverage many existing ontology reasoning tools [Wie12; TH06]. Users of the ontology-based knowledge base can be humans or software agents (e.g., tools for program optimizations or testing.)

We use canonical loop analysis to illustrate how the idea of ontology-based program analysis works. A canonical loop is a type of well-structured loop conforming to specifications as shown in Figure 3.2. Because of its regular structure, it has been the focus of many studies on parallelization and loop optimizations [Kan99; Lia10; DM13].

The goal of canonical loop analysis is to recognize whether a loop is in a canonical form. Traditional implementations of the analysis (e.g., the implementation in the ROSE compiler [Ros]) contains hundreds of lines of code for examining the IR of a loop. The code is tied to a particular internal data structure of the chosen compiler, hard to port to another compiler or maintain.

In a traditional program analysis development, the task requires an insertion of a separate pass
over some intermediate representation of the whole program, which may need the development of thousands of lines of code. For example, in the ROSE compiler [Ros] (a source-to-source compiler broadly used in High Performance Computing), the pass works on an Abstract Syntax Tree (AST).

To analyze the code at that level, a programmer needs to implement many lines of code written in procedural languages (C/C++). The canonical loop analysis in the ROSE compiler consists of 380 lines of source code for examining the representations of the structure of each loop and check them against the conditions in Figure 3.2.

In ontology-based program analysis, the process is simpler. The programmer needs to invoke some provided ontology converter on the code of the target program. An ontology-based knowledge base is then produced to capture the program constructs, components, and their relations. The programmer then just needs to use a declarative logic programming language to describe rules governing the forms that a canonical loop should conform. Treating those rules as queries on the ontology of the program, existing logic reasoners can then automatically find all the canonical loops in the target program. For example, a fragment of C code is shown in Listing 3.1. The code’s corresponding ontology representation is shown in Listing 3.2. Each line is a triple: (subject, predicate, object). The numbers in the subjects or objects are the line and column numbers of the beginning
Listings 3.2 Sample ontology for C snippet (“x rdf:type y” indicates that y is the type of the code segment in the range x)

1 (‘3:1,5:1’, rdf:type, ‘ForStatement’)
3 (‘3:6,3:10’, rdf:type, Variable’)
4 (‘3:1,5:1’, ‘hasForInit’, ‘3:6,3:14’)
5 (‘3:1,5:1’, ‘hasForTest’, ‘3:17,3:22’)
6 (‘3:1,5:1’, ‘hasForIncr’, ‘3:25,3:28’)
7 (‘3:1,5:1’, ‘hasBody’, ‘3:30,5:1’)

and ending positions of a language construct in the source code. (Next section explains the triples in the example in details.) The different program constructs can be easily extracted by Prolog queries like those in Listing 3.3.

Listing 3.3 Sample analysis rules

1 isForStatement(Loop) :-
2 \rdfs(Loop, rdf:type, c:ForStatement).
3 hasForInit(Loop, InitExpr) :-
4 \rdfs(Loop, c:hasForInit, InitExpr).

Allowing the use of logic programming, ontology-based program analysis inherits the productivity benefits of declarative program analysis. More importantly, it overcomes the three aforementioned shortcomings of existing declarative program analysis by leveraging ontology for standardizing the concept definitions in a domain and the flexible representation of various sources of knowledge.

3.3 Challenges and Solutions

The challenges for integrating ontology into program analysis exist in each of the four main steps: the design of an ontology for the domain, the generation of the knowledge, the utilization of the knowledge base, and the design of the entire framework. In this section, we discuss each of the challenges and present some principles we use to address them. At the end, we describe PATO, the prototype framework we have developed to do program analysis upon ontology-based program representations.
3.3.1 Ontology Design

Challenges

To create an ontology for any domain, the first primary task is to define the vocabulary to be used in the domain. That includes the definition of the concepts, properties, and restrictions in the domain. These definitions establish the conceptual terms and their relations of the ontology-based knowledge base for the domain. Even though there are some de facto procedures on designing an ontology [NM01], program analysis has some special challenges. Program analysis has a large variety of tasks (e.g., loop analysis, data access pattern analysis, alias analysis, dependence analysis, liveness analysis, busy expression analysis, etc.) involving a huge set of diverse concepts and relations. Even a larger variety exists in the input programs. So the first question to use ontology for program analysis is how to design an intuitive, efficient and flexible ontology that can facilitate the various program analyses and input programs.

Solutions

In this work, we focus on the design of ontology for representing programs of a particular programming language (C). The design of the complete ontology of program analysis is left to future work.

Through our explorations, we have found the following three principles helpful.

- **Language standard-oriented design.** When designing the vocabulary of an ontology, it helps if one starts with reusing language constructs and their categorizations defined in the standard of the programming language of the target programs. Despite the variety of the input programs, they are all artifacts following the particular programming language. With the constructs and their categorizations of the programming language covered, we can easily express programs using the language in the ontology-based knowledge base. This approach helps achieve a good coverage of the input programs with a vocabulary familiar to users. For example, to model C programs in an ontology, we followed the standard of C99 and enumerate all program constructs and concepts in a top-down fashion. Figure 3.3 shows a fraction of the top-level ontology for C programs. The main concepts in the domain include variables, expressions, statements, and so on. The “construct” on the edges indicate these vocabularies are the basic constructs in the C program domain.

- **Being generic in property designs.** Besides classes of concepts, an ontology also contains a vocabulary for properties (or called relations). For example, an Expression instance may have some Type, an Identifier may refer to some Definition. Because there may be many properties to express, in our practice, we follow a principle trying to define properties in a generic way,
and encode semantic meanings into concepts whenever possible. That allows the possible use of a small set of properties and their combinations to express a large number of possible properties in a program. For example, when describing an identifier has static storage class, one approach is to define a property hasStaticStorage and use it like (someVar hasStaticStorage true). Alternatively, we may define the concept Storage as a class and use a simpler property hasStorage as (someVar hasStorage static), where static is a member of Storage. There are several benefits for this second choice. First, using generic properties makes the set of property vocabularies small and thus easy to manage. For example, hasStorage is used to describe all storage classes instead of creating specific properties for each storage class. Second, stripping semantics from property make writing logic rules more flexible. For example, the knowledge of (someVar hasStorage static) can be queried by the keyword hasStorage. Otherwise, users may need to use many specific keywords as hasStaticStorage, hasExternStorage, and so on.

• Continuous enrichment. In our exploration, we find that continuous enrichment of the ontology vocabulary can be helpful. For instance, in a canonical loop analysis, a user gives the description of the concept of a canonical loop. If that concept turns out to be needed frequently (by many users), the concept could then be integrated into the ontology framework to save the need for repeated descriptions. Given that the program analysis ontology is intended to be used by a community, a complexity is that different users may use different names for a single user-defined relation (e.g., canonical loop), making the detection of the repeated use of a user-level concept difficult. Ontology-based logic reasoners come in handy. As the descriptions of user-defined relations all use the vocabulary in the same ontology, the reasoners can easily do a logic reasoning on the descriptions to decide whether two user-defined relations are equivalent. After recognizing the frequently needed user-level concepts, such concepts can be added into the standard ontology for future reuse.

Based on the three principles, we came up with an ontology for C program representations to facilitate C program analysis. It contains 178 concepts and 68 properties. It is not intended to be complete (e.g., many program analysis concepts are not yet defined), but is sufficient for examining the support of a single ontology to multiple different program analyses as we will show later in this paper.
3.3.2 Knowledge Generation

Challenges

Building up a knowledge base is essential for any application of ontology. For program analysis, the knowledge base shall include the important knowledge related with the to-be-analyzed program. There are several main questions to answer.

- **Ontology converter.** How to construct an ontology converter that can automatically convert a given program into the ontology representation needed by many common program analyses.

- **Naming.** A program may contain functions, statements, expressions, variables, and so on. Any of them may appear many times in different locations. A challenge is what naming scheme the ontology should use to reference each reference to avoid ambiguity.

- **Mapping.** One of the objectives of ontology-based program analysis is to facilitate the cooperations among different compilers and other program analysis tools. A difficulty is that they may have different internal representations of a program. To make them able to interact based on the program ontology, the instances of program constructs in the knowledge base should be possible to map to some common ground meaningful to the different tools. One intuitive choice of such common ground would be the source code of the program. That will also make it easy for human to collaborate with program analysis tools. A complexity in using source code for references is how to make the naming robust to code changes.

- **Space.** The knowledge about a program can be tremendous. Besides the basic knowledge directly driven from the program, there could be many other kinds of higher-level knowledge such as canonical loops, data dependences among statements, and so on. The higher-level knowledge is derivable from the lower-level knowledge. The derivation through reasoning may take non-trivial time. But if all this knowledge is saved in the knowledge base, the space cost could be large. How to strike a good balance is important for practical usage of this new program analysis paradigm.

Solutions

We come up with the following solutions to these challenges.

- **Ontology converter:** Our experience shows that an ontology converter can be easily created through a translator built on top of a source-to-source compiler such as ROSE [Ros]. A source-to-source compiler usually produces an abstract syntax tree (AST) representation which is close to the input code. The translator traverses the AST of the program to get structural and semantic information, which is then stored into the knowledge base as ontology. The program
constructs are represented as individuals (i.e., instances) of some of the classes defined in the language ontology. Relations between them are represented by properties.

- **Naming and Mapping.** In our naming scheme, we borrow the *internationalized resource identifiers (IRIs)* [Hit09] that OWL uses. It helps avoid name conflicts. For the ontology of C program language, names of concepts are built directly from the corresponding terms used in the C language standard\(^1\). For example, the concept type is referenced as `c:Type`. An example IRI for the concept of types in a C program domain can look like “http://example.com/owl/CProgram:Type”. Some aliases can be defined as short names for the prefix strings of a domain.

More care needs to be taken for designing the naming scheme for representing the instances of a program construct. Named constructs such as types, variables, functions can use the C++ qualified name concept to uniquely identify them. For an unnamed construct (e.g., an assignment statement) or a reference to named construct (e.g., variable reference), a common intuitive approach is to use its location in the source code of the program, such as file url, start location, end location where the location is a pair of (line number, column number). For instance, “http://my.com/file1.c, 3:1, 5:1” could refer to a loop that spans from the beginning of the third line to the beginning of the fifth line of file1.c. The problem with this scheme is that some minor changes to the original program may invalidate all the names of the constructs after the modification point.

We find *scoped IRI* useful to restrict the impact of a code change to the names. In *scoped IRI*, a name is composed of some qualified names and some relative locations in the source code. For constructs like functions, structures, global variables, we add their scopes before their names. Other constructs within these constructs are named by their locations, while the line numbers are relative to the start line number of their surrounding constructs rather than the beginning of the source-code file. Using this method, a global variable declared in the first line of a file “s.c” (from column 5 to 6) can be named as `s.c::1:5,1:6`, while a variable declared on the first line inside a function foo in file “s.c” (from column 6 to 10) can be named as `s.c::foo()::1:6,1:10`. Thus, if there is some change of the code, only the names in the same scope as the changing point is need to be updated.

- **Space.** To address the space challenge of putting everything into a knowledge base, we 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. We also design a cache-like management mechanism to alleviate the problem. It maintains a buffer to store derived supplemental knowledge. When the upper limit of the buffer gets reached, it starts to evict some of the stored knowledge (which would need to be rederived when

\(^1\)We follow the naming convention of UpperCamelCase for classes and lowerCamelCase for properties.
needed). For the eviction policy, there can be multiple choices: least-recently-used (LRU), least-frequently-used (LFU), and their variants.

### 3.3.3 Knowledge Utilization

#### Challenges

Some challenges also exist in the utilization of the knowledge base for program analysis. **(1) Efficiency.** In many cases (e.g., analyzing a large program), the runtime efficiency of conducting a program analysis could be important. A question to answer is whether the improved productivity of the new paradigm hurts the runtime efficiency and if so, how to improve the efficiency. **(2) Generality.** Using declarative programming languages could be awkward for some usage cases, especially when they involve some mathematical computations. Such cases however do exist in some program analysis and optimizations, for instance, when they relate with some performance models (an example is the data placement optimizations Section 3.4 will describe). Effectively overcoming such limitations is important for the general applicability of ontology-based program analysis.

#### Solutions

We address these issues by both creating some shortcuts and leveraging the features of existing ontology tools.

- **Efficiency.** Recent years have seen some significant improvement of the performance of logic reasoners [TH06; Bas06]. Many optimizations have been developed. For instance, in SWI-Prolog, the ontology is stored as relation triples of (subject property object) with C extensions and some indices 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. For instance, Prolog provides the cut operator (i.e., the ! symbol) to avoid unwanted backtracking in search. We find that following some existing guidelines when writing queries [Bra01] can be quite helpful for quickly narrowing down the reasoner’s search space.

  In scenarios 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, which can further help achieve good performance than going through a heavy-weight logic reasoner.

- **Generality.** Ontology-based logic reasoning meets the needs of many typical program analyses, but is not quite suitable for expressing analyses that involve a lot of mathematical computations (e.g., a regression-based performance modeling). We find that a mechanism called computable [TB09] originating in Robotics can help resolve the issue. Computable is some
special ontology entity that can attach procedures to some classes or properties. When the deduction rule queries the individuals of one of the classes or the properties, the associated procedures are invoked to compute individuals for the target class or property. The procedure can be written in a wide range of programming languages (e.g., C, C++, Python).

Besides allowing the direct input of queries by users written in logic programming languages, ontology-based program analysis also allows queries coming from a third-party software (which could be written in even imperative languages like C, C++ or Java.) For such cases, the ontology can be processed with libraries for those languages (e.g., the OWL API [HB11] or Prolog interface to foreign languages[BC02]). That offers the conveniences for existing software tools (e.g., a compiler) to easily leverage ontology for program analysis.

### 3.3.4 Framework Design and PATO

The final challenge is how to organize the various components together into a unified framework for program analysis. It includes choosing the DL language and the reasoner that suit the needs of program analysis, designing and implementing the APIs for both knowledge base construction and queries, and integrating all the components together into a complete cohesive framework that offers effective support to various program analyses.

![Figure 3.4](image-url) Structure of PATO.
To answer these questions, we have developed a prototype framework named PATO, which integrates all the aforementioned solutions to the various challenges, and leverages the power of existing ontology tools.

Figure 3.4 outlines the main components of PATO. The knowledge base in PATO can be in two forms, represented by the two round-corner boxes on the right part of the figure. On the disk, the knowledge is stored as OWL files. Each entry in the files is in an OWL triple (subject, property, object). For instance, (var1, hasValue, 0) means a variable has a constant value of 0. The reason for selecting this form is that OWL triple is one of the standard (and space-efficient) formats for ontology representations and is accepted by many ontology tools. Another part of the knowledge base is the computables, which are attached to the concepts and properties in the triple collection. The cache-like buffering mechanism is used for the space efficiency of the knowledge base.

Two of the primary ways to add knowledge into the knowledge base are shown at the bottom of Figure 3.4. In the first way, there is a parser (based on ROSE [Ros]) for converting an input program code into an AST preserving source level information. We have also developed a translator that translates the basic program knowledge on the AST into OWL triples and stores them into the knowledge base. During the translation, the translator uses the scoped IRI as the naming scheme. In the second way, we adopt Stanford ProtÃŠžl'gÃŠl' [Gen03], an interactive tool for ontology creation and manipulation. Through its GUI, a user can intuitively add entries into the knowledge base. The plugins of ProtÃŠžl'gÃŠl' also provide many other features to the users, such as visualization, validation, and querying.

There are two ways to use the knowledge base. The first is shown at the top of Figure 3.4. It is through the SWI-Prolog reasoning engine. To use the knowledge base in this way, at the beginning, the OWL files are loaded into memory; through it, the OWL files are converted into an internal knowledge base through the existing Semantic Web Library [Wie12]. The in-memory organization of the knowledge base features some efficient indexing schemes. When seeing a query from a user or software agent, the Prolog engine would start working on the internal knowledge base to provide the answers. Prolog provides a general interface for input logic rules and queries. We have further developed a higher-level set of API tailored to represent some common terms used in program analysis tasks (e.g., loops, functions, etc.), through them, the users may write even more concise descriptions.

The second way to use the knowledge base is shown as the shortcut (the diagonal path) in Figure 3.4. Through a direct query interface we have developed in C++, the user or software agent may directly work on the OWL file collection (and computables) without going through Prolog. This shortcut is useful when Prolog is not available to a user or too costly to use in some special scenarios. For tasks which only need some simple fact search (without the need for much reasoning), this approach is more efficient than the Prolog-based approach because it avoids the overhead in Prolog interpretation and other associated cost. Both the Prolog and the shortcut approach can insert new
knowledge (e.g., derived in a previous program analysis) into the knowledge base.

It is worth noting that the rich set of available tools on ontology proves helpful in our development and usage of PATO. Besides the aforementioned usage of ProtÃél'gÃľ [Gen03] for ontology development, we find the FaCT++ reasoner [TH06] helpful for checking the consistency of the ontologies, the Prolog semweb library [Wie] useful for loading, parsing, and manipulating ontology-based knowledge bases, and the Prolog engine [Wie12] a convenient tool for inferences upon the knowledge bases.

3.4 Experience with PATO

This section describes our experience of using PATO for program analysis. To examine the feasibility of using a single ontology to support multiple different program analyses efficiently, we implement five types of program analysis on PATO: canonical loop analysis, pointer analysis, control flow graph construction and data access pattern analysis. They differ in domains, relations, scopes, and intended uses.

For the interest of space, our description concentrates on the canonical loop analysis, the pointer analysis. We briefly cover the other analyses and the GPU data placement experiment at the end. Without noting otherwise, each reported timing result is the average of 10 times of repeated measurements collected on a machine equipped with Intel Core-i5 CPU of 3.2GHz (8GB DRAM, 500Gb HDD hard drive) running Ubuntu 14.04. For the results that show large variances, we also report the statistics on the variances. The Prolog used is SWI Prolog, and the primary compiler is the ROSE compiler (EDG 4x-Based version) [Ros].

3.4.1 Canonical Loop Analysis

We first explain the Prolog code for canonical loop analysis. As mentioned earlier, canonical loop analysis (CLA) checks whether a loop is in a predefined canonical form. Our experiment uses the canonical loop form defined in the OpenMP specification [Ope13], shown in Figure 3.2. The specification of an OpenMP canonical loop can be written as declarative Prolog rules as shown in Listing 3.4. We use italic font to distinguish variable from normal symbols.

In the Prolog specification, the head cannonicalLoop(Loop) asks for individuals that satisfy all clauses in the body. The (,) plays the role of logic conjunction (AND operation). Every clause in the body is deducted by its own rule. In the end, the deduction is backed by queries on the existing knowledge base. The isForStatement(Loop) is a more readable wrapper of the ontology query Loop is-a ForStatement, where the variable Loop binds to individuals if ontology triples (some-loop is-a ForStatement) exist in the knowledge base. Once the Loop is bound to some individual, clauses like hasForInit(Loop, InitExpr) search the knowledge base for (some-loop hasForInit some-init-expr)
Listing 3.4 Prolog specification of an OpenMP canonical loop (italic upper-case for variables, lower-case for properties)

% top level rule to find canonical loop
canonicalLoop(Loop) :-
isForStatement(Loop), !, %'!' prevents backtracking
hasForInit(Loop, InitExpr), %',' means logic AND
canonicalInit(InitExpr, LoopVar),
hasForTest(Loop, TestExpr),
canonicalTest(TestExpr, LoopVar),
hasForIncr(Loop, IncrExpr),
canonicalIncr(IncrExpr, LoopVar),

(hasType(LoopVar, 'IntType'); %';' means logic OR
hasType(LoopVar, 'PointerType'))

hasBody(Loop, ForBody),

% supportive rules to find canonical init-exp
canonicalInit(Init, LoopVar) :-
hasOperator(Init, AssignOperator), !,
hasLeftOperand(AssignOperator, VarRef),
referTo(VarRef, LoopVar),
hasRightOperand(AssignOperator, LB).

% rules with same heading: combined using logic OR
canonicalInit(Init, LoopVar) :-
hasVarDecl(Init, LoopVar),
hasInitializer(Init, Initializer),
hasValue(Initializer, LB),
% the rest is omitted ...
triples. Then canonicalInit(InitExpr, LoopVar) checks if the found initial expression individuals conform to the language specification. The query also returns the loop variable LoopVar if it can find it.

The analysis further checks whether the loop’s init construct conforms to the specification. The first rule handles the var = lb style while the second rule deals with other styles. Different rules with the same head name form the logic disjunction (OR operation). The cut operator (i.e., the ! symbol) is used to prevent unwanted backtracking. It means that, as long as the first rule matches the form hasOperator(Init, AssignOperator), there’s no need to check the second rule of the variable declaration form. Line 11 to 12 are the rules to do the type checking of loop variables. The (;) means logic disjunction (equivalent to two separated rules).

The ontology in PATO successfully supports the analysis. We compare it with the imperative implementation in the ROSE compiler. The algorithm in ROSE traverses the AST tree of a program to find for statement nodes and check whether their sub-tree are in the canonical form. The code is written in C++ and is specific to the ROSE AST’s internal data structures. The time complexity is $O(n)$, where $n$ is the number of AST nodes.

The code length of PATO-based analysis is about half of the imperative implementation in the ROSE compiler (190 versus 380 lines). We use the NAS Parallel Benchmarks (NPB) [Bai91] for the measurement.

The results are shown in Figure 3.5. The frontend time corresponds to the parsing time of the parser in Figure 3.4. PATO uses the same frontend as the ROSE implementation. The time of program knowledge generation (KB gen) represents the time taken by the ontology translator in the PATO framework. Finally, the other two bars are the time of the canonical loop analysis with PATO and ROSE respectively.

For the canonical loop analysis alone, we can see that the PATO declarative approach even beats the native implementation for most cases. It is slower only when the line number is small. One reason is that the ROSE’s imperative implementation traverses the whole AST tree to find all for loops. For PATO’s declarative implementation, the isForStatement relation information is stored explicitly and can be efficiently accessed by the hash-indexed keyword implementation of Prolog.

The ontology-based declarative analysis does have an obvious overhead: It needs an extra step to traverse the AST tree and build the initial knowledge base. However, as shown in the figure, the overhead is smaller than the frontend time. Besides, for the PATO system, the generated knowledge base can be reused for different program analyses (e.g., control flow graph analysis).

An appealing property of using ontology is the good extensibility. The previous discussion only covers canonical loops in C programs that use primitive loop variable types, such as pointer and integer types. OpenMP allows C++ canonical loops that use complex iterators as long as the iterator supports random access to the data elements. Examples of random access iterators include std::vector<T>::iterator and std::deque<T>::iterator. Also, programmers often define their
own random access iterators. A conventional solution to extend an imperative CLA implementation is to store the known random access iterators, including the custom-defined ones, into a container for the compiler implementation to look up. The ad-hoc solution imposes barriers for exchanging the knowledge with other tools or developers. In PATO, adding such a support is simpler. One can easily add the concept RandomAccessIterator in the ontology and define it as is-a(Iterator)∧has(RandomAccess). The knowledge of is-a(Iterator) property can be gained by the knowledge builder while the knowledge has(RandomAccess) can be inserted into the knowledge base using the standard OWL API, either automatically by tools or manually by developers. The analysis rules can then reason about random access iterators. The resulting ontology can be then shared and reused by different analyses.

3.4.2 Pointer Analysis

This part describes our experience in implementing Andersen's pointer analysis in PATO. Andersen's pointer analysis is a well-known inclusion based analysis [And94]. The analysis result is typically represented as the points-to set $pts(x)$ for each pointer variable $x$. It is flow-insensitive and does not distinguish different program execution points but computes what the pointers may refer to at any time of program execution.

The analysis is commonly regarded as solving a set constraint problem. It classifies assignments involving pointers into several kinds: taking the address of a stack variable or a heap allocated space, copying a pointer from one variable to another, and assignments through dereferences or references to a multilevel pointer. It defines a propagation rule (or called constraint) for each of the cases as
illustrated in Table 3.1 (for C programs).

The analysis consists of two steps. The first step preprocesses the target program to produce a simplified intermediate representation, which keeps only the statements that involve pointer manipulations. Each kind of the assignments involving pointers is represented with a special notation. For example, \( p = \& b \) is represented as \( \text{stackLoc}(p, b) \), \( p = \text{malloc}(...) \) is represented as \( \text{heapLoc}(p,...) \), \( p = *pp \) becomes \( \text{load}(p, pp) \), and \( *pp = q \) turns into \( \text{load}(pp, q) \).

The second step propagates the points-to relations based on those constraints and solves the problem by computing graph transitive closures through an iterative worklist algorithm [And94].

<table>
<thead>
<tr>
<th>Constraint type</th>
<th>Statement</th>
<th>Propagation rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>( p = &amp; b )</td>
<td>( \text{loc}(b) \in \text{pts}(p) )</td>
</tr>
<tr>
<td>Simple</td>
<td>( p = q )</td>
<td>( \text{pts}(p) \supseteq \text{pts}(q) )</td>
</tr>
<tr>
<td>Complex</td>
<td>( p = *pp )</td>
<td>( \forall v \in \text{pts}(pp) \cdot \text{pts}(p) \supseteq \text{pts}(v) )</td>
</tr>
<tr>
<td>Complex</td>
<td>( *pp = q )</td>
<td>( \forall v \in \text{pts}(pp) \cdot \text{pts}(v) \supseteq \text{pts}(q) )</td>
</tr>
</tbody>
</table>

In our implementation on PATO, the preprocessing step is done on the program ontology representation through some simple pattern matching rules. For example, the address taken pattern \( (p = \& b) \) is matched by the following:

**Listing 3.5** The Prolog rule to match the address-taken instruction

```prolog
matchAddressTaken(RHS) :- hasOperator(RHS, AddressOp), hasOperand(RHS, LocRef), referTo(LocRef, Var).
```

The implementation breaks complex statements into simple ones by introducing temporary variables. For example, \( *p = *q \) becomes \( tmp = *q; *p = tmp \). Structures are analyzed in a fieldsensitive manner, while an array is regarded a single data object as done in most previous studies [And94; HT01; BS09].

For the analysis step, the propagation rules in Table 3.1 can be directly mapped into logic rules in Prolog as shown in Listing 3.6.

However, our experiments show that Prolog inferences based on these rules are not efficient. It evaluates the rules in a top-down manner with deep recursions and costly search through a large space. Inspired by previous work [BS09; Wha05], we instead implement the classic worklist algorithm in Prolog as illustrated as follows:

**Listing 3.7** The worklist algorithm for pointer analysis in Prolog

```prolog
andersonPtr :-
```
**Listing 3.6** The Prolog rules for the points-to computation

```prolog
% p = &a; p = malloc()
pointsTo(P, Loc) :- stackLoc(P, Loc); heapLoc(P, Loc).
% p = q
pointsTo(P, X) :- copy(P, Q), pointsTo(Q, X).
% p = *pp
pointsTo(P, X) :- load(P, PP), pointsTo(PP, V), pointsTo(V, X).
% *pp = q
pointsTo(V, X) :- pointsTo(PP, V), store(PP, Q), pointsTo(Q, X).
```

2 select(WorkList, V),
3 propLoad(V); propStore(V);
4 propFieldLoad(V); propFieldStore(V);
5 propEdge(V),
6 andersenPtr. % iteratively execute the analysis
7
8 propLoad(V) :-
9 load(P, V),
10 pts(V, A), % for each A in pts(V)
11 (\+ edge(A, P) -> assertz(edge(A, P)), add(WorkList, A)).
12 % others are omitted

Line 11 in the listing means that as soon as Prolog finds out that there is no edge from A to P it stops search and instead add A into the worklist and continue with next A’ in pts(V, A’). The use of \(\rightarrow\) enables an early stop of useless searches, bringing large performance benefits.

The entire implementation takes less than 500 lines of code, about 300 lines of which are for the preprocessing step. When being applied to the NPB benchmarks, the analysis takes no more than 2 seconds on a program. We have also applied the analysis to three programs [Mor] with more pointer operations: bzip2 (7K lines of source code), gzip (8.6K lines), and oggenc (58K lines). The analysis times are 2.1 sec, 3 sec, and 36 sec respectively.

### 3.4.3 Other Experience

For the interest of space, we briefly describe two other experiments.

One of them is an analysis to find out a program’s array access patterns, including the access expressions to each array, lower and upper bounds of the loops surrounding an array reference, numbers of reads and writes to an array, and the ranges of its elements that are accessed. With PATO, each of the types of information can be easily extracted from the code through just a few lines of Prolog statements. A previous imperative data access pattern analysis [Che14] has more than 2000
lines of source code. PATO, on the other hand, only needs 180 declarative rules to implement that analysis.

The second is to write code for control flow graph construction. It requires the examination of control flows of the whole program. On PATO, it is done through a set of simple rules based on the algorithm of inductive graph constructions [Fis09]. We compare our PATO implementation with the implementations of ROSE and Clang [Lat08], which also use the inductive construction algorithm but differ in the internal data structures and implementations. The PATO version uses only 400 lines of code, up to 8.75X shorter than the traditional compiler implementations (1200 and 3500 lines for ROSE and Clang respectively). The speed of the PATO analysis is similar to that of the Clang, and is 10-40% faster than that of the ROSE thanks to its efficient storage and query of the knowledge base.

3.5 Related Work

Ontology has been used to build various knowledge bases in different domains, including Biology [Ash00], Ambient Intelligence [Duc01; Pre04; Rod14], Robotics [TB09], and others [Mat06; NP01; Pea02]. This work was enlightened by these studies, but concentrates on the special challenges facing program analysis.

In the software domain, ontology has been introduced, but mainly for software management and teaching of programming concepts, rather than program analysis. Specifically, Software ontology (SWO) [Mal14] in the domain of software engineering focuses on the meta information of software (e.g., licenses, publishing processes, data formats). COPS [Lan07] offers a sub-ontology for managing the knowledge related with image processing. Eden and others [ET07] have provide some theoretical discussions on the unique aspects in designing an ontology for programs, but without exploring the use of ontology for program analysis. There are several ontology designs for teaching some programming languages [SG06; Gan11]. This current work, to our best knowledge, is the first proposal on a systematic integration of ontology into program analysis.

There are some prior efforts trying to ease the difficulties in the development of program analysis. We discuss them in two aspects.

The first aspect is in the construction of a program analysis. Some prior studies have offered some interfaces for simplifying the construction of a program analysis. OpenAnalysis [Opea], for instance, introduces a set of analysis-specific interfaces (e.g., traverse all statements) as the building blocks for constructing a specific analysis. Other efforts in the same direction include GENOA [Dev92] and StarTool [JA88]. These tools focus on imperative program analysis. There are some efforts that employ declarative program analysis to improve the productivity of program analysis development. JTransformer [Kni07], for instance, is a tool integrated into Eclipes that allows the use of Prolog for analyzing and transforming Java source code. JunGL [Ver06] introduces a scripting language for writing program analyses. It combines ML and Datalog. SemmleCode [Ver07] is a tool that stores
program-related data into a knowledge base, and allows the use of Datalog to write a program analysis that analyzes the program by querying the knowledge base. There are some other work [Haj06; BS09; Wha05] falling into the same category.

The second aspect is in the representation of a program. Some prior work has tried to develop a common software exchange format (SEF) for representing a program to facilitate the interoperation of software analysis and refactoring tools. Such a format needs both a schema (or called metamodel) that describes the objects and relationships, and a syntax that describes how model elements are to be stored and transmitted. The Dagstuhl Middle Metamodel (DMM) [Let04] is a representative of the former. DMM consists of a set of models that capture program elements and their relations. It follows some prior efforts such as Columbus [Fer02] for C++ and the UML metamodel [ Boo05]. There are some other metamodels developed, such as Program Element Fact (PEF) developed in JTransformer [Kni07], and DIMPLE [BF07]. Graphs are the most popular format for storing program elements in memory. TA [Hol97] and TGraphs [Ebe08] are two examples, which are both based on typed graphs (i.e., directed graphs with attributes on both nodes and edges). GXL [Hol00] was introduced as a generic way to use XML to represent such graphs. JunGL [Ver06] uses some special graphs along with abstract syntax trees.

This current work shares some similarities with these prior studies, such as the use of logical programming to simplify the development of program analysis, and the creation of a common program representation. However, this work differs from the prior studies in several major aspects. First, it is the first work that points out the potential of ontology for the program analysis community to standardize the conceptualization for program analysis, and to promote the reuse and interoperations of analysis tools. Second, this work is the first that points out the benefits of ontology as a unified representation for not only program elements and relations but also knowledge from other sources (e.g., hardware knowledge) that are essential for program optimizations. It demonstrates the promise through GPU data placement optimizations. Third, unlike many of the prior studies that attempt to create a standalone tool for program analysis (e.g., JTransformer [Kni07], Semmlecode [Ver07]), this work aims to proposing an approach or a paradigm, which could potentially be employed by many program analysis tools and compilers, as demonstrated by the experiments described in the previous section.
4.1 Problem Statement

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 ad-hoc 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 [Dat09]. 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 [Kra10; Val14; Spa14], 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 [Goo] and GoKB [Gok]. 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 comprehen-
sive way to help the HPC community systematically accumulate, share, and reuse HPC knowledge. OpenK’s design centers around ontology [UG96], 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 chapter 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.
4.2 Scope and Design Considerations

We first note that fully automating HPC knowledge acquisition 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 considerations 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 [Sto10] versus `thread blocks` in CUDA [Nvi11] 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.

### 4.3 Overall of the OpenK System

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 4.1 depicts the overall structure of the OpenK framework, which consists of a front end, a middle end, and a back 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 [HL97]), or source code of a computer program.

The second interface is a locally installed integrated development environment (IDE) based on Protégé [Gen03]. 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 [Wie12] and the Jena Ontology Framework [Car04].

**Middle End**

The middle end of OpenK consists of two main components. The first is a set of logic reasoners including SWI-Prolog [Bra01] and Hermit [Gli14], 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 4.5 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 [Zha16]. Another converter in OpenK converts a hardware specification written in a language called MSL [Che14] into RDF. In addition, OpenK is equipped with Pellet [Sir07], an efficient reasoner for querying and processing OWL-2 ontologies and Semantic Web Rule Language (SWRL) rules [Hor04]. 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.

### 4.4 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.

#### 4.4.1 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 (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.
Figure 4.2 Part of the HPC knowledge taxonomy.

(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 4.2. 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) [Hit09] 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) [Hor04] to represent rules. SWRL is a widely supported language extending OWL with logic rules [Hor04]. 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 [Pro11], 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.

4.4.2 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 [Zha16], a tool for converting hardware specifications (in MSL [Che14]) into RDF; and a tool [Vas09] 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.
4.4.3 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 invalidated.

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.
4.4.4 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 [Gli14], 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.

4.4.5 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 [Fou16]. 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 [Bos11], 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 [TH06; Bas06]. 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 [Bra01]. 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.

4.5 Experience with OpenK

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.

4.5.1 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 4.3 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 4.4 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 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.

### 4.5.1.0.1 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.
1. On single-core cache-based arch, tiling is effective if problem size is larger 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 512³, cache blocking is ineffective but CSE (common subexpression elimination) is useful.

Figure 4.5 Some insights on optimizing stencil computations.

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 [Dat09], it forms an ontology-based knowledge base for optimizations of stencil computations. Figure 4.5 shows some of the insights (in English for readability).

Meanwhile, we ourselves played the role as the related community and created the metaknowledge (i.e., vocabularies and concepts) for the stencil computation optimization domain and upload it into OpenK. Figure 4.4 gives a glimpse of it. Besides, the ontology about the related architectures are represented in ontology and stored in the knowledge base. It provide an encyclopedic reference for information query.

4.5.1.0.2 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 512³-point grid and on a 1024² × 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 [Kam05]. The target architecture is Intel Xeon E3-1200 (4-core 3.2GHz, 8G RAM, 32K L1, 256K L2 and 6MB L3).

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
Table 4.1 The speedup of different optimizations. The stars (*) mark the ones suggested by the OpenK (size-1: 512\(^2\), size-2: 1024\(^2\) × 512, CSE: common subexpression elimination).

<table>
<thead>
<tr>
<th>Grid size</th>
<th>neighbor size</th>
<th>Cache blocking</th>
<th>SIMD, padding</th>
<th>Array</th>
<th>Unrolling, CSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>size-1</td>
<td>7-p</td>
<td>10% *</td>
<td>≈ 0</td>
<td>≈ 0</td>
<td></td>
</tr>
<tr>
<td>size-1</td>
<td>27-p</td>
<td>≈ 0</td>
<td>13% *</td>
<td>30% *</td>
<td></td>
</tr>
<tr>
<td>size-2</td>
<td>7-p</td>
<td>22% *</td>
<td>≈ 0</td>
<td>≈ 0</td>
<td></td>
</tr>
<tr>
<td>size-2</td>
<td>27-p</td>
<td>8%</td>
<td>14% *</td>
<td>42.5% *</td>
<td></td>
</tr>
</tbody>
</table>

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 4.5), 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 4.5 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 4.1 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.

4.5.2 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 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 [Lat08] and ROSE [QL11]. 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

\[
enhancement = \frac{|A|}{|C|}.
\]

The results are shown in Figure 4.6. Box plots are used to show the distribution of the enhancement rates over all the basic blocks in the program. In the plot, the dots are outliers, and the intervals show the range of the lower 75 percentile of the enhancement in the observed result. The plot considers only the basic blocks whose Liveout sets are not empty in the result from at least one of the two compilers. The results indicate that the synergy improves the average precision of Liveness analysis by over 10% and 20% for LLVM Clang and ROSE respectively.

It is worth noting that the two compilers use different internal representations for programs and the Liveness analysis results. It is possible to write some special code to map their results to enable such a combination without using ontology. However, using the ontology designed in this work, the benefits come as simple side products of the ontology-based program analysis (by leveraging the converters and the standardized representation developed for many other program analyses). The productivity benefits would become even more prominent when many types of analyses cooperate across compilers.

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.,
Figure 4.6 Enhancement of Liveness analysis by leveraging ontology to combine the results from the Live-
ess analysis in LLVM Clang and ROSE. (Dots show outliers). Left bars: the enhancement over LLVM Clang results; right bars: the enhancement over ROSE results.

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 [Che14]. 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 [Che14] 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. Listing 4.1 shows part of the MSL syntax and the specification of the constant memory of a GPU (Tesla M2075).

The specification states the type of memory (constMem), its unique ID (in numbers), whether it is software manageable (Y or N), and whether a GPU kernel can read or write the memory (the "rw" field). The other fields indicate its dimensionality, memory size, number of banks, memory latency, and so on.

In OpenK, the ontology representation is more generic and extensible. We don’t need to design...
\textbf{Listing 4.1} PORPLE Memory Specification Language.

\begin{verbatim}
1  \% syntax
2  memSpec ::= name id swmng rw dim size blockSize
3     banks latency upperLevels lowerLevels shareScope
4     concurrencyFactor serialCondition ; end-of-line
5  \% an example, ? represents unknown attributes
6  constantMem 1 Y r na 64K ? ? 360clk <cL2 cL1> <>
7  die ? ...
\end{verbatim}

\textbf{Figure 4.7} Part of the GPU hardware ontology.

any specific syntax for hardware knowledge. Only new concepts and relations (or properties) in the hardware domain need to be added. Figure 4.7 illustrates part of the GPU ontology. Listing 4.2 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

\textbf{Listing 4.2} Ontology of Tesla K20c global memory

\begin{verbatim}
1  globalMem_k20c type GlobalMemory;
2  globalMem_k20c hasUpperLevel L2;
3  globalMem_k20c shareScope die;
4  globalMem_k20c pieces 1;
5  globalMem_k20c software_manageable true;
6  globalMem_k20c accessible "rw";
7  globalMem_k20c has_size 5032706048;
8  ... \% omitted
\end{verbatim}
guidance can rely on heuristic rules, mathematical modeling, or their combinations. For instance, a previous study [Jan11] makes the decisions based on some empirical rules on data access patterns and 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 [Che14], 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 4.3 in OpenK. The algorithm used in the previous data placement optimizer (PORPLE) can be written concisely as Listing 4.4 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 [Vas09], 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* [TB09].

Listing 4.3 SWRL rule for using constant memory (italic for variables)

1. `isArray(x)∧readOnly(x)∧ConstantMem(y)`
2. `sizeFit(x,y)∧sameAccessWithinWarp(x)`
3. `→ suitableMem(x,y)`

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 [Che09; Che14], including some from the RODINIA suite [Che09] and two kernels `glassForce`, `glassControl` from the DOE application LULESH [Kar12]. The speedups over the original benchmarks are shown in Figure 4.8, 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 [Che14], 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
Listing 4.4 Algorithm of data placement written in Prolog

```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).
```

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.

4.6 Related Work

For its expressiveness, ontology has been adopted in many fields. In software engineering, software ontology (SWO) [Mal14] 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 [Zha16]. Ontology has also been shown to be useful for domain-specific language development [Lia15]. 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 [Gok], clinical research [D. 12], some for enabling scholarly discovery through interlinked profiles of people and other research-related information (e.g., VIVO [Kra10]), some for scientific literature search (e.g., Semantic Scholar [Val14]), and some for identifying useful hypotheses and experiments in a scientific literature [Spa14]. 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 [Goo] 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 [Opeb] 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) [Lin00] 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 [Nor14], 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.

Figure 4.8 Speedup of benchmarks on Tesla K20c.
This current work employs some ideas in the prior studies in converting program code into ontology, and its use of descriptive program analysis also shows the productivity benefits. However, the main focus of this work is on the design of the overall OpenK infrastructure. As a general HPC knowledge sharing infrastructure, its benefits go much beyond supporting descriptive program analysis, extending to the support of more effective program optimizations, promoting the synergy among HPC community, and enabling easy cooperation among different compilers and program analysis and optimization tools.
Part III

Knowledge Acquisition and Capitalization through Machine Learning
5.1 Problem Statement

In High Performance Computing (HPC), there are many cases calling for decision making that resemble some problems in other domains on which machine learning algorithms have shown effectiveness. However, HPC problems also feature some distinctive attributes that pose some special challenges to these algorithms. In this thesis, we take one of the fundamental HPC problems, sparse matrix storage format selection, as the focus to explore this direction.

Sparse matrix vector multiplication (SpMV) is one of the most important, widely used kernels in many scientific applications (e.g., linear equation system solvers) [KR01; BP98]. It is also often the performance bottleneck of their executions [Fal06; Bol03]. Optimizing its performance is difficult because of the irregular indirect data access patterns.

One of the most important factors people have observed for the SpMV performance is the selection of the proper format to represent sparse matrices in memory. Various storage formats have been proposed for diverse application scenarios and computer architectures [LT16; SK12; Kou11; LV15a; Vud05; Cho10; BG09; Yan14; Xie18; LV15b]. As observed in numerous studies [Li13; Sed15; Cho10; BG09; Vud05; MG16; Liu17; Wan18], the different formats may substantially affect the data locality, cache performance, and ultimately the end-to-end performance of SpMV (for as much as
several folds [Li13]). Meanwhile, there is no single format that has been found optimal for all sparse matrices. The proper format of a matrix depends on many factors, including the characteristics of the sparse matrix, the hardware architecture, the implementation of the SpMV library, the application, and so on.

Based on the previous studies [Li13; Sed15; Ben16], we exploit different machine learning techniques to solve the sparse matrix selection problem.

## 5.2 Background

This section briefly introduces the relevant background before we present our proposed solutions to the SpMV format selection problem.

### 5.2.1 Sparse Matrix Storage Format

To efficiently store and process a sparse matrix, compressed data structures (also called storage formats) are used by storing only its nonzero entries. More than 50 different storage formats have been proposed [LT16; SK12; Kou11; LV15a; Vud05; Cho10; BG09], most of which are customized for certain special scenarios. Four basic formats CSR, COO, DIA, ELL are extensively used in numerous applications [Li13; BG09]. Most of the other formats are mostly derived from these four basic formats to better fit computer architectures or capture the features of a particular type of matrices. For example, block formats (e.g. BCSR, BELL [Vud05; Cho10]) benefit sparse matrices with dense sub-blocks to get better cache performance, while hybrid formats (e.g. HYB, Cocktail [BG09; SK12]) take basic formats as building blocks. In this proof-of-concept work, we start with the four basic formats; the techniques can be easily extended to include other formats.

Figure 5.1 shows an example sparse matrix represented in CSR, COO, DIA, ELL formats respectively and their corresponding SpMV algorithms. It use *nrows, ncols*, and *nnzs* to represent the number of rows, columns, and nonzero entries of the sparse matrix respectively. Compressed Sparse Row (CSR) format uses three arrays, *ptr, indices*, and *data*, to represent a sparse matrix; *data* stores the values of nonzero entries, *indices* stores their corresponding column indices, and the beginning positions of all rows are stored in *ptr*. COOrdinate (COO) format explicitly stores the row and column indices and the values of all nonzero entries in *rows, cols*, and *data* arrays separately.

Diagonal (DIA) format stores non-zeros along the diagonal direction (from top left to bottom right). In the DIA example in Figure 5.1, the first column of *data* contains the two elements on the left bottom diagonal of matrix A, the second column for the principle diagonal of matrix A, and the third column for the diagonal on the top right of the matrix. The array *offsets* records the offsets of each diagonal from the the principle diagonal. The idea of the ELLPACK (ELL) format is packing all non-zeros to *compact ncols* columns, then storing the packed dense matrix by padding zeros
in a two-dimensional \( n \times m \) array; \( data \) stores this dense matrix and \( indices \) stores their corresponding column indices. Please refer to prior document [Saa94] for more details.

\[
A = \begin{bmatrix}
1 & 5 & 0 & 0 \\
0 & 2 & 6 & 0 \\
8 & 0 & 3 & 7 \\
0 & 9 & 0 & 4
\end{bmatrix}
\]

**Figure 5.1** Sparse matrix storage formats and their corresponding SpMV pseudo-code (adapted from [Li13]).

### 5.2.2 Convolutional Neural Networks (CNN)

CNN is a class of networks that finds non-linear models of patterns in inputs and makes data classifications. The most popular type of CNNs used in image recognition are convolutional neural networks (CNNs) [Kri12; He15].

A CNN usually consists of a stack of layers of nodes. In Figure 5.2 (a), the leftmost layer is the input level, with each node representing one element in the input vector (e.g., the gray value of a pixel in an input image), the rightmost is the output level, with each node representing the predicted...
probability for the input to belong to one of two classes.

The output layer of a CNN gives the final prediction, while the other layers gradually extract out the critical features from the input. A CNN may consist of mixed types of layers, some for subsampling results (pooling layers), some for non-linear transformations. Convolution layers are of the most importance, in which, convolution shifts a small window (called a filter) across the input, and at each position, it computes the dot product between the filter and the input elements covered by the filter, as Figure 5.2 (b) shows in a 2-D case. In Figure 5.2 (a), the weights of every three edges connecting three input nodes with one layer-2 node form the filter $< w_1, w_2, w_3 >$ at that level. The result of a convolution layer is called an activation map. Multiple filters can be used in one convolution layer, which will then produce multiple activation maps. The last layer (i.e., the output layer) usually has a full connection with the previous layer.

Part of the CNN training process is to determine the proper values of the parameters in the filters (i.e., weights on the edges of the networks). In training, all the parameters in the network are initialized with some random values, which are refined iteratively by learning from training inputs. Each training input has a label (e.g., the ground truth of its class). The forward propagation on an input through the network gives a prediction; its difference from its label gives the prediction error. The training process (via back propagation) revises the network parameters iteratively to minimize the overall error on the training inputs. In using CNN, only forward propagation is needed to get the prediction.

Note that the size of the inputs to a CNN is typically fixed, equaling to the number of nodes in its input layer. If the raw inputs are of different sizes, they have to be normalized to the unified size.
Figure 5.2 Illustration of CNN.
6.1 Motivation

Deep Neural Networks (DNN) has already shown high impact on various applications via its several distinctive appealing properties, such as much less demand for feature extraction and highly accurate prediction. Its special efficacy is demonstrated in assisting perceptions and decision makings.

The matrix format selection problem is a good candidate for DNN, especially Convolutional Neural Networks (CNN), to solve, for several reasons. First, selecting the proper format is a challenging task for programmers. The proper format of a sparse matrix depends on its matrix size, nonzero distribution, architecture characteristics, and so on. Second, it has been also difficult for traditional machine learning techniques to solve. Due to the difficulties in coming up with the right features of matrices for learning and the complex relations between SpMV performance and the proper format of a sparse matrix, so far, traditional machine learning techniques have achieved an average of 85% [Li13] and 78% [Sed15] prediction accuracy \(^1\). Considering the misprediction

---

\(^1\)The accuracy number of [Li13] is the average of different platforms and precisions, and that of [Sed15] is the average of different platforms using their "Advanced2" feature set.
may lead to several-fold performance loss, a more accurate prediction model is needed. Third, the problem resembles some other tasks that CNN has proved effective. Particularly, it is akin to image classification—such as, to tell whether an image contains a dog or a cat; in both problems, the right decisions are primarily determined by the spatial patterns of the elements in an input. For image classification, the patterns are of pixels, and for sparse matrix format selection, they are of non-zero elements. As CNN has shown good efficacy on image classifications, the similarity of the two problems suggests some promise for it to work for sparse matrix format selection.

On the other hand, the problem poses some special challenges to CNN. Three of them are especially prominent.

The first is input representation. CNN typically requires all input data to be of the same size (as it usually has a fixed number of visible nodes.) For a dataset that contains data of different sizes, some data normalization methods are usually applied to transform them into a fixed size. For images, the transformation could be cropping, scaling or sampling. That approach works in general for image processing because it keeps the major patterns of the objects in the image. But sparse matrix format selection is sensitive to some subtle features of a matrix, which can get lost by those traditional transformations. For instance, as Section 6.3 will elaborate, scaling creates some diagonals for a non-diagonal matrix; as having diagonals critically affects format selection for SpMV, the scaled images mislead CNN learning and prediction. Therefore, new research explorations are necessary for understanding the effects of the various transformations for keeping the important features of sparse matrices, and for finding the fixed-size representations that properly suit the need of CNN for sparse matrix format selection.

The second special challenge is the design of the suitable CNN structures. CNN structure refers to the number of network layers, the type of network of each layer, and the number of nodes on each layer. Differences in CNN structures affect the quality of the learning results significantly. For image processing, researchers and practitioners have used different CNN structures for some public image sets. However, because of the different representations of the input data, the prior explored structures for image processing may not work effectively for sparse matrix format selection. New research is hence needed for identifying the structures of CNN that fit the needs of sparse matrix format selection.

The third challenge is the architectural dependence of sparse matrix format selection. As prior studies have shown [Vud05; Wil09; Cho10], many factors of a machine (e.g., memory bandwidth, cache size, number of cores) could affect the performance of SpMV on a particular matrix format and the best format for a given matrix. A prediction model built for one machine rarely works well for another. Re-training CNN is time-consuming. How to efficiently migrating a model across systems is a problem specifically important for HPC problems.

This chapter crystallizes all the explorations on these problems into a set of novel findings on the applications of CNN to sparse matrix format selection, which could shed insights for bridging
the gap between CNN and other HPC problems.

### 6.2 Overview of the CNN based Solution

In this work, by overcoming some major difficulties, we successfully construct a CNN-based sparse matrix format selector for SpMV. This section gives a high-level overview of the construction process, and lists the major challenges which will be discussed in detail in later sections.

#### Figure 6.1 Overview of the model construction process.

As the left half of Figure 6.1 shows, the construction process consists of four steps. It assumes that there are already a large set of sparse matrices $S$ that the construction can use as its training inputs, and a target computing platform $P$ (where the executions of SpMV happen.)

1. The first step collects training labels. It runs SpMVs for all matrices on $P$ multiple times, using a different matrix format each time. By measuring the execution times, for each matrix, it finds out the format that SpMV runs fastest and labels that matrix with the ID of that format. (2) The second step normalizes each of the matrices into a fixed size such that they can be fed into the input layers of CNN. (3) The third step designs the structure of CNN. The parameters of the designed CNN are initialized to some random values. The output layer is composed of $K$ output nodes with each corresponding to one of the $K$ matrix formats to choose from. (4) The fourth step runs the standard CNN training algorithm on the collected labels and the normalized matrices to finally determine the value of each parameter of the CNN, and concludes the construction process.

Inference with the trained CNN model is easy. For a given matrix, it is first normalized to the fixed size required by the CNN. The normalized representation is then fed into the trained CNN, the output nodes of which give the probabilities for each of the formats to be the best choice for that input matrix to use.

Steps one and four are easy to realize. Steps two and three face some research challenges special
to the sparse matrix format selection problem. Additionally, the resulting CNN is specific to the training platform $P$ as it uses the labels collected on $P$. How to quickly migrate the learned CNN to another platform is another research challenge. The next three sections separately describe our solutions to these three major research issues.

### 6.3 Input Representations

Matrices are of various sizes. For them to work with CNN, they have to be represented in a single size as the input layer of a CNN requires. This process is called matrix normalization. It is important that the normalization keeps the features of the original matrix that are crucial for determining the appropriate format for the matrix to use. In this section, we first describe the common normalization method used in image processing, and then present two novel methods we propose to overcome the limitations of the traditional methods in the context of sparse matrix format selection.

#### 6.3.1 Representation from Traditional Methods

In CNN-based image processing, input normalization is through image scaling, which down-samples large images or interpolates small images. We tried to apply the same method to matrices. In down sampling, for instance, a block in the original matrix maps to one element in the new matrix. Because the spatial patterns of non-zero elements rather than their exact values are relevant to SpMV performance, the values of the elements of the normalized matrix are set to binary. The new element is set to zero if the original block contains all zeros, and 1 otherwise. It results in a binary matrix.

The scaling method’s results are not satisfying, reflected by low prediction accuracies (88%) by its constructed CNN (detailed in Section 6.6). The main reason is that although scaling keeps the coarse-grained patterns of objects in an image, it loses some subtle info that is critical for matrix format selection.

Figure 6.2 shows such an example. The original matrix contains irregular diagonals, but after down-sampling, the normalized matrix becomes a perfect regular diagonal matrix. Being a diagonal matrix is an important property for matrix format selection: Some matrix storage formats (e.g., DIA) are designed particularly for efficiently storing diagonal matrices; the normalization result hence causes confusions to the CNN construction as well as the format prediction. Such cases happen frequently on large sparse matrices.

We explored a number of representations to help address the limitations, two of which showed good promise, described next.
6.3.2 Augmentation with Density Representation

In density representation, instead of producing zero or one for each block of the original matrix, it produces a decimal value between 0 and 1, equaling the number of nonzero entries in a block divided by the block size, as Figure 6.3 (a) illustrates for the original matrix in Figure 6.2 (a).

Compared to the binary representation, the density representation captures more detailed variations among the different regions of the original matrix. Binary representation is still useful for capturing the overall spatial patterns of the non-zero elements in the original matrix. Using both of them in one CNN could possibly get the best of both worlds. We will return to this point when discussing the structure design of CNN in Section 6.4. The density and binary representations can be produced by one traversal of the original matrix.
Algorithm 1 Normalized through Histogram Sampling.

1: procedure \textsc{HistNorm}(A, r, BINS)
2: /* create a row histogram for an input matrix A */
3: /* the target representation is a $r \times \text{BINS}$ matrix $R$*/
4: initialize a $r \times \text{BINS}$ empty matrix $R$
5: ScaleRatio = A.height/r
6: MaxDim = max(A.height, A.width)
7: for each non-zero entry $e$ in $A$ do
8: \hspace{1em} int row = $e$.row / ScaleRatio
9: \hspace{1em} bin = BINS * $|e$.row $- e$.col$ / MaxDim
10: \hspace{1em} $R[$row$][bin]$++
11: return $R$

6.3.3 Distance Histogram Representation

Our second proposal is distance histogram or called histogram representations, which stores the spatial distribution of non-zero elements in a matrix through histograms. It consists of two matrices, with one storing the histograms for the rows of the original matrix, and the other for the columns. The histogram is based on the distance between an element and the principal diagonal of the original matrix.

Algorithm 1 outlines the algorithm for constructing the row histogram from a given matrix. We take the original matrix in Figure 6.2 (a) as an example for explanation. In this example, we try to construct a 4-row histogram matrix $R$ for the matrix. Every two consecutive rows in the original matrix produce one row in the histogram matrix. Suppose that we want the histogram to have 4 bins regarding the distances from the elements to the principal diagonal of the original matrix. Now consider the bottom two rows (rows 6 and 7 with 0-base index) in Figure 6.2 (a). Row 6 contains only one non-zero element (with value 23), whose distance from the principal diagonal is 1; the histogram bin number for that distance is $\lfloor 1/2 \rfloor = 0$. It hence causes $R[3][0]$ to increase by 1 (lines 9 and 10 in Algorithm 1). Row 7 contains two non-zero elements (with values 17 and 11), and their distances from the principal diagonal are 4 and 1 respectively. Their histogram bins are 2 and 0 respectively (calculated by line 9 in Algorithm 1), causing $R[3][2]$ and $R[3][0]$ each to increase by one. Hence, the result of the bottom row of $R$ is $[2, 0, 1, 0]$.

In the same vein, one can construct a histogram for the columns of the original matrix. Together they form the histogram representation for the matrix. The values in both matrices are then normalized to the range of $[0,1]$ by dividing the largest value in each.

Compared to the binary and density representations, the histograms—using numerical values and leveraging distances rather than direct spatial locations—tend to capture richer information about the distribution of non-zero elements in the matrix. Meanwhile, their sizes are more flexible to
adjust. For binary and density representations, because matrices can be larger in either dimension, the representations are typically made square to strike a tradeoff. For histograms, there is no such a need; the number of histograms can differ from the number of rows or columns. A benefit of the relaxed constraint is that the size of the histograms could be smaller. For instance, in our experiments reported later, $128 \times 128$ is the size that the binary and density representations should take to get a good prediction accuracy, while $128 \times 50$ already works well for histograms.

6.4 CNN Structure Designs

A CNN network may take various structures with different depths or widths or types of layers. Different structures have distinct modeling strengths and generalities. For instance, a more complex network with deeper and wider layers can typically capture more complex relations between the inputs and outputs, but at the same time, it would require more data to train than a simpler network needs.

6.4.1 Insufficiency of Common Structures

Despite the large varieties, in image processing, the usual structure of a CNN is as Figure 6.4 illustrates. The values in different channels of an image (e.g., Red, Green, Blue values of pixels) together form a single input layer, and the other layers all work on the values combined from all these input nodes. The differences among different CNNs for image processing have been mostly on the configurations of each of the layers after the input layer.

![Figure 6.4](image)

**Figure 6.4** Traditional CNN structure that merges the info from different channels at the early stage of the networks.

In our exploration, we start with a set of CNNs of the similar type of structure as used in image processing. However, after training each of the CNNs, we find the resulting prediction accuracies unsatisfying, regardless of which input representation is used (detailed in Section 6.6.)

Through examinations of the intermediate computation results from the CNNs, we found that the fundamental reason was the combinations of the info from different sources in the early stages
of the networks. In image processing, the different sources of information are just different channels of the input image, and have a homogeneous semantic. For instance, for an RGB image, the \( j^{th} \) element in each of the three channels all corresponds to the \( j^{th} \) pixel in the image, and the values are all about the numerical values of the pixel's color, just in different color channels.

However, in our problem, the different sources of the info do not have such homogeneous semantic or one-to-one matching relations. Consider the binary representation and the density representation. Even though each element is derived from one region in the original matrix, they have different types of values. In the histogram case, even though the elements in the two histogram representations have the same value type, they do not have a one-to-one matching relation: one for rows, the other for columns. As a result, the combinations of different sources of info at the early stage of CNN work well for image processing, but not for our problem.

6.4.2 Late-merging Structure

To address the problem, we employ an alternative late-merging structure. As Figure 6.5 illustrates, the structure consists of two separate convolutional networks with each processing the info from one source, and only at the very last stage, the outputs of the two networks are put together as joint features, fed to the fully connected layer for the final output. The two convolution networks can be regarded as processes to extract the critical features from each of the two sources of input information. The final layer combines these features for the final prediction.

![Figure 6.5 Proposed late-merging CNN structure.](image)

Such a structure avoids the early mixing of the influence from the different sources in the early-merging structure, and gives two-fold benefits. First, it leads to a simpler problem. In the early-merging case, the convolution network has to simultaneously consider the influence from both sources of info and extract out their combined features. In comparison, each of the convolution networks in the late-merging structure only needs to extract the features from one source of info. The simplicity entails a simpler structure needed for the convolutional networks, and hence the
reduced demands for the amount of training data and time. Second, the late-merging structure avoids the complexities from the different value types and semantics of the different sources of info.

The way that the late-emerging structure combines features from different sources is similar to feature concatenations in other models (e.g., Inception [Sze15]). It fits the special properties of sparse matrices in the context of format selection.

For either the early-merging or late-merging structure, there are numerous possible designs of the CNN with different numbers of layers and other configurations. Following the common practice, we choose the configuration through the trial-and-error method by training and testing many CNNs with different configurations and select the best one.

### 6.5 Cross-Architecture Adaptations

The need for cross-architecture migrations is another special aspect of sparse matrix format selection compared to CNN-based image processing. In image processing, data labels are oblivious to computing systems: A dog in a picture is a dog on whatever machines. But it is not the case for sparse matrix format selection. The labels are the best formats for the input matrices, which could differ significantly from one machine to another. As a result, the CNN trained on one machine cannot predict the formats for another machine well (as Section 6.6.3 shows).

The implication of the architecture-dependence is that a new CNN has to be built on each different machine, which includes the collection of labels by rerunning SpMV on each matrix on the new machine and rerunning the CNN training algorithm to determine the appropriate parameters. Both parts take a lot of time. Such a process takes about 75 hours in our experiments for about 9200 matrices.

#### 6.5.1 Concept and Two-fold Contributions

To alleviate this problem, we explore the use of transfer learning. Transfer learning is an idea existing in the deep learning field [Raz14; Yos14]. In image recognition, people can train a CNN on a large dataset, and then use the pre-trained network as the base in training a CNN for a new dataset. In the previous studies, the idea is only explored for speeding up the training of CNN across datasets.

Our contributions are two-fold. First, we introduce this idea to HPC to support cross-architecture portability of CNNs, and point out its benefits in saving both data collection time and CNN training time. Second, we empirically examined the options for materializing the idea in this new context and identified the suitable method. We next describe the explored options and the rationale of each; Section 6.6 will provide the empirical comparisons.
6.5.2 Options Explored

We have primarily considered two ways to materialize the transfer learning idea for sparse matrix format selection.

**Continuous evolvement.**

This method treats the existing CNN as an intermediate state of the new CNN, based on which, it feeds the new set of training data collected on the new machine to the CNN to continue training the CNN until a convergence. This method reuses the structure and parameters of the previous CNN. Because these parameters have reflected certain important features of input matrices, reusing them could provide a better starting point than random values for the CNN construction for the new dataset. Meanwhile, even though the labels on two machines could differ, they usually do not differ completely. By inheriting the parameters of the previous CNN, the training of the new CNN could benefit from certain relations that those parameters have already captured between the features of matrices and the suitable formats.

**Top evolvement.**

This second method inherits all parts of the previous CNN and keeps both the structure and the parameters unchanged, except the parameters of the top fully connected layer. We call the output of the reused part of the CNN the *CNN codes*. Because of the reuse, the CNN codes of a matrix stay the same across machines. Therefore, in the construction of the new CNN, this method just feeds these codes as inputs to the top layer of the CNN and uses the standard back propagation (on the labels collected on the new machine) to learn the suitable parameters of this top layer. After that layer is trained, we get the updated CNN by putting that layer on top of the reused part of the previous CNN. The rationale of the *top evolvement* method is that the major part of a CNN can be considered as the extraction of critical features of inputs. Because the tasks on the two machines share a similar nature, the set of critical features of the inputs could be similar—hence, the reuse. The top layer combines these features together and makes the final decision on the output and is hence closely related to the labels of data—hence, the update.

6.5.2.1 Qualitative Comparisons

Qualitatively, the *top evolvement* involves a much smaller set of parameter changes than the first option does. As a result, it needs fewer training data, which entails a shorter time needed for collecting new labels and also a shorter time for the CNN training algorithm to converge. On the other hand, the *continuous evolvement* allows a much larger freedom for adjustment; as a result, theoretically speaking, the best CNN it can provide (infinite training data are allowed) shall be no worse than the
Table 6.1 Hardware platforms used in the experiments.

<table>
<thead>
<tr>
<th></th>
<th>Intel® CPU</th>
<th>AMD® CPU</th>
<th>NVIDIA® GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>Xeon CPU E5-4603</td>
<td>A8-7600 Radeon R7</td>
<td>GeForce GTX TITAN X</td>
</tr>
<tr>
<td>Freq.</td>
<td>2.4 GHz</td>
<td>3.1 GHz</td>
<td>1.08 GHz</td>
</tr>
<tr>
<td>Cores</td>
<td>24</td>
<td>4</td>
<td>3072</td>
</tr>
<tr>
<td>Memory</td>
<td>64GB DDR3 1.9 GHz</td>
<td>12GB DDR3 2.1 GHz</td>
<td>12GB GDDR5 3.5 GHz</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>103 GB/s</td>
<td>25.6 GB/s</td>
<td>168 GB/s</td>
</tr>
<tr>
<td>OS/Driver</td>
<td>SUSE Linux Server 11</td>
<td>Ubuntu 16.04</td>
<td>CUDA 8.0</td>
</tr>
<tr>
<td>Compiler</td>
<td>ICC (17.01)</td>
<td>GCC (6.2)</td>
<td>NVCC (8.0)</td>
</tr>
</tbody>
</table>

one provided by the top evolvement method. However, if the objective is to achieve a high-quality CNN at a minimum training cost, the answer may not be that clear. Section 6.6 will offer some quantitative comparison results.

6.6 Evaluations

To evaluate the efficacy of the technique, we run a series of experiments and compare the CNN-based method with the state-of-the-art method in both prediction accuracies and resulting SpMV speeds. In addition, we report the influence of the three kinds of input representations, the benefits of the late-merging CNN structure, the impact of the two methods of transfer learning, and the sensitivity of the learning results to the granularity of the input representations.

6.6.1 Methodology

Baseline for Comparison

The state of the art in sparse matrix format prediction is a traditional machine learning model produced by Li et al. [Li13] and Sedaghati et al. [Sed15]. They manually came up with a set of features of matrices, and built up a decision tree based on the features and labels of a set of training matrices. They showed that the decision tree model outperforms previous methods in prediction accuracies and the resulting SpMV performance.

Hardware

We construct format prediction models for two CPU systems and one GPU platform, detailed in Table 6.1. The trainings of CNNs all happen on the GPU.
Software Platforms and Formats to Select

As Figure 5.1 has shown, different formats require SpMV to be coded differently. To evaluate the speedups of SpMV brought by format predictions, we need to use a SpMV library that can work with multiple matrix formats. A SpMV library contains only a set of procedures with each working with one of a small set of matrix formats, which determine the set of formats we can use in our experiments.

On CPU, we experiment with two SpMV libraries. One is Intel® MKL [Wan14], which supports COO, CSR, DIA, and several other formats. The other is a multi-purposed SpMV benchmarking program (called SMATLib in this paper) from the previous work [Li13], which supports COO, CSR, DIA, ELL formats. Both libraries are multithreaded and run in parallel.

On Intel® MKL [Wan14], the speedups from our prediction over the use of the default format (CSR) is up to 53.7× (1.46× on average). We concentrate our CPU result discussions on the SMATLib platform rather than MKL because SMATLib was used in the state-of-the-art study on matrix format prediction [Li13]. Reusing it allows for a direct head-to-head comparison with the previous work. The framework repeats a performance measurement for 50 times. The average is taken as the performance. We observe some small variances across the repeated trials; they are negligible compared to the significant performance differences between the different formats.

On GPU, we experiment with the NVIDIA® cuSPARSE library [Nau10], which supports COO, CSR, ELL, HYB and BSR formats. A previous work reports some promising performance of a new format CSR5 over some alternative formats and publishes the implementation [Liu16]. We append to cuSPARSE with the CUDA implementation from that work to make it also work with CSR5 format.

The set of formats covered in this study are restricted by the formats supported by these existing libraries. The support of these covered formats by these (commercial) libraries indicates their competitiveness and general applicability. The set unavoidably leaves some formats uncovered. With the idea verified, the approach can be easily extended to the selection of other formats.

Dataset

Our experiments use a set of 9200 matrices (total size around 400GB). These matrices include the 2757 real-world matrices from the SuiteSparse matrix collection [DH11] (which was also used in the previous studies [Li13; Sed15]), and some extra matrices derived from them. The derivation attempts to create some variations of the existing matrices, and at the same time, do not deviate too much from the real-world matrices. To do this, we use some simple heuristics like cropping, transforming and randomized combinations of the original matrices.
Table 6.2 Prediction quality on Intel® CPU (bottom decimals show overall accuracies)

<table>
<thead>
<tr>
<th>Format</th>
<th>Ground Truth</th>
<th>CNN/Bin.</th>
<th>CNN/Bin./Density</th>
<th>CNN/Hist.</th>
<th>DT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Recall</td>
<td>Precise</td>
<td>Recall</td>
<td>Precise</td>
<td>Recall</td>
</tr>
<tr>
<td>COO</td>
<td>0.53</td>
<td>0.70</td>
<td>0.71</td>
<td>0.74</td>
<td>0.71</td>
</tr>
<tr>
<td>CSR</td>
<td>0.94</td>
<td>0.92</td>
<td>0.94</td>
<td>0.94</td>
<td>0.97</td>
</tr>
<tr>
<td>DIA</td>
<td>0.82</td>
<td>0.83</td>
<td>0.82</td>
<td>0.85</td>
<td>0.93</td>
</tr>
<tr>
<td>ELL</td>
<td>0.79</td>
<td>0.78</td>
<td>0.82</td>
<td>0.80</td>
<td>0.90</td>
</tr>
<tr>
<td>Overall</td>
<td>0.88</td>
<td>0.90</td>
<td>0.93</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cross Validation

For the evaluations on the SpMV format prediction, we separate testing data from training data through 5-fold cross validations. This is a method commonly used in statistical learning for evaluations. It takes 20% of valid matrices out to form a test set and uses the remaining valid matrices for training. It repeats the process for 5 times with a different subset of the dataset taken out as the test set.

In the rest of this section, Sections 6.6.1 to 6.6.4 first report the prediction accuracies of the machine learning models and the speedups they brought to SpMV. Section 6.6.5 then discusses the runtime overhead and the related practical usage issues.

Prediction Accuracy

This part compares the CNN models (with late merging) and the previous Decision Tree (DT) model [Li13; Sed15] in the quality of their predictions. We use three metrics. The first is overall accuracy, defined as the number of correct predictions (i.e., the predicted best format is indeed the best) over the total number of matrices. The other two assess the quality of the predictions for each format: precision on format X is the fraction of all predicted X that is correct; recall on format X is the fraction of S_X that are predicted as X, where S_X is the set of matrices on which format X is indeed the best.

Table 6.2 reports the results of our three CNN-based models (with Binary, Binary+Density, or Histogram representations) and the previous Decision Tree-based model (DT) [Li13; Sed15] on the Intel® CPU platform. The “Binary” and “Binary+Density” use 128 × 128 as the size of the representations, while the “Histogram” uses 128 × 50 (these sizes were picked empirically.) The second column in Table 6.2 gives the number of matrices of each format label from the overall 9200 sparse matrices. The results show that all three CNN models outperform the DT model, in almost all metrics. CNN with histograms achieves the best results, with an overall accuracy of 93%; DT gets only 85% accuracy. CNN also shows higher recall rates and precisions for all the formats.
Table 6.3 Prediction results on the GPU platform.

<table>
<thead>
<tr>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ground Truth</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>CSR</td>
</tr>
<tr>
<td>ELL</td>
</tr>
<tr>
<td>HYB</td>
</tr>
<tr>
<td>BSR</td>
</tr>
<tr>
<td>CSR5</td>
</tr>
<tr>
<td>COO</td>
</tr>
<tr>
<td>Overall</td>
</tr>
</tbody>
</table>

The results differ for different formats. There are two major factors: the amount of training data and the complexity of patterns. CSR, for instance, has much better prediction results than other formats have thanks to the largest number of matrices carrying the CSR label, which allows CNN to learn it more sufficiently. COO and ELL have the similar amounts of data, but COO has much worse prediction results than ELL has. Our detailed analysis shows that matrices favoring ELL tend to have a similar pattern (rows in the matrix have similar numbers of non-zeros), while COO does not show clear patterns.

Table 6.3 reports the results on the GPU platform. As GPU memory is more limited, only 4218 of the matrices can run on it. As GPU cuSPARSE supports more formats than SMATLib, the table shows results on six formats. For space limit, we include only CNN with histograms and DT results. Among the six formats, format COO never wins on GPU, while the other five all win on some matrices\(^2\). Overall, CNN again outperforms DT significantly in both the overall accuracy (90% versus 83%) and the per-format metrics.

6.6.2 Speedup

This section reports the speedups of SpMVs by using our CNN model predicted formats over using the DT model's predictions. Figure 6.6 shows the speedup distribution over testing matrices on which the two models give different predictions of the suitable formats. The horizontal line at 1 explicitly separates the matrices achieving speedups using CNN model and those not showing speedups compared to the DT model. The CNN model helps improve the SpMV performance on 86% matrices over the DT model. The SpMVs using the CNN model predicted formats achieve an average of 1.73× and the maximum of 5.2× speedups over those of the DT model. This result further confirms that sparse formats are critical to SpMV performance. This comparison shows the performance improvement of our work over the state of the art[Li13]. Furthermore, we also tested

\(^2\)BSR uses a 4 × 4 block size [Vud03].
the SpMV speedups with our CNN model over the default CSR format, which are 2.23× on average and 14.9× in maximum. For the GPU platform, the CNN model achieves an average of 1.7× and the maximum of 22.5× speedup of the default CSR format.

![Figure 6.6](image)

Figure 6.6 Speedups from CNN over DT-based predictions.

6.6.3 Model Migrations

Figure 6.7 reports the effect of the two transfer learning methods. The methods try to migrate the CNN model trained on Intel® platform to AMD® platform. The benefits of the methods are obvious. On less training data, they achieve much better accuracy than the “from scratch” method does. Note that the time to migrate a model consists of the retraining time of the CNN model as well as the time to collect the labels on the new platform. The latter takes most of the time. In our experiments, the collection of the labels for the 9200 matrices takes about 75 hours as it requires the executions of SpMV on each of the matrices many times.

Therefore, the large reduction of the needed retraining data by the transfer learning methods significantly shortens the model migration process. Suppose that we want to get a model on the new platform with a 90% accuracy. From Figure 6.7, we can see that with “top evolvement”, it takes only about a quarter of the time the “from scratch” method takes, and two thirds of the time the “continuous evolvement” method takes. Overall, “continuous evolvement” achieves a slightly higher accuracy than “top evolvement” after relearning 4000 inputs, while “top evolvement” provides a faster learning process.
Figure 6.7 Prediction accuracies of different retraining methods on a new platform (AMD® platform). (Accuracies at x=0 are without retraining.)

Figure 6.8 CNN late-merging structure (layer shapes in parentheses).

6.6.4 Impact of CNN Structures

This part compares late-merging and early-merging (Figures 6.4 and 6.5 in Section 6.4.) Figure 6.8 shows the CNN of the late-merging structure. It involves 13 layers. The early-merging model has the same structure except that the networks merge the use of the different channels at the beginning of the networks (the input layer uses one INPUT(128 x 128 x 2) rather than two separate INPUTs(128 x 128)).

Figure 6.9 shows the loss function curves of the two CNN models. Here, loss function is defined as the cross-entropy between the true labels and the predicted ones [Mur12]. Cross-entropy is a measure of the similarity between the distributions; the smaller it is, the closer the two distributions are. The loss function of late-merging structure decreases much faster than that of the early-merging structure. The late-merging structure uses about 7000 steps to converge to the loss value around 0.1,
while early-merging structure converges to 0.4 after 10000 steps. By using only 1000 time steps, the
loss function's value of late-merging is half of that of early-merging. In addition, the late-merging
structure behaves more steadily than early-merging does. These results confirm that the late mixing
of the different sources of information simplifies the prediction problem, and avoids the negative
impact from the mixture.

![Figure 6.9](image)

**Figure 6.9** Comparison of the loss function convergence for two different structural designs.

### 6.6.5 Discussion

Training a CNN model based on collected data is about 27min. It is a one-time effort for a given
platform.

Prediction overhead is worth more discussions. In both our CNN method and the previous
DT-based method, the prediction for a matrix includes two steps: 1) input representation or feature
extraction; 2) feeding the feature vector to the predictive model to get the prediction. In terms of
the time taken by one iteration of SpMV (on the CSR format), the overhead for CNN method is
(on average) 0.96× for step 1 and 0.13× for step 2, 1.09× in total. The overhead for the DT method
is 3.4× for step 1 and 0.0085× for step 2, 3.4× in total. Another source of overhead is the format
conversions. Converting one existing format of a matrix to another format could take a number of
SpMV iterations' time.

These overhead entails different ways to use the CNN method in different settings. First, the
runtime prediction overhead of the CNN method is much smaller than the DT-based method, which
gives the CNN method an extra edge in addition to its better prediction accuracy. However, because
SpMV is often called on a matrix repeatedly (e.g., a linear solver takes hundreds or thousands of iterations [BT09]), the 1–3 iterations of overhead is negligible compared to the time the better formats help save (see more proof in [Tan18]).

Both methods are subject to format conversion overhead, which could be more significant than the prediction overhead. If the collection of matrices are to be used by many users (on the same or different platforms) for many SpMV-based computations, a one-time process can be used to generate and store all the candidate formats of the matrices. That can save the runtime needs for format conversions, and hence avoid the influence of format conversion overhead on the usage of the predictive models.

6.7 Insights

We summarize three key insights and their general implications for other HPC tasks. The first is that CNN has some good potential for helping improve the decisions in HPC. Beyond the sparse matrix format selection problem, we envision that CNN may help other problems: selecting the proper solvers for solving a linear system (which also typically take sparse matrices as inputs) [Ans09; Bho09; Anz16], choosing the proper compilation flags, finding good scheduling policies, and so on. Using CNN for them could face some challenges as those encountered in this work, such as cross-architecture portability, CNN structure designs, input representation issues. The findings reported in this work can shed some light on solving the issues in these other HPC scenarios.

The second insight is the importance of tailoring the representations of HPC data inputs according to the nature of the problem in the HPC domain. The representations we have proposed in this work could be useful for other sparse matrix algorithms beyond SpMV. Meanwhile, this work also shows the importance of thinking out of box when applying CNN to HPC, reflected by the benefits of the unconventional CNN structure (late-merging structure) driven by the special properties of the input representations.

The third insight is the benefits of transfer learning in saving time required for porting learned models across architectures. This technique is especially useful for HPC due to the widely existing architectural sensitivity.

6.8 Related Work

The work closest to this study is the SMAT work [Li13; Tan18] which builds up a decision tree for selecting the best storage format for a sparse matrix storage. The previous sections have provided quantitative comparisons with that work. A similar classification-tree–based model was built in another recent study [Sed15] with a 65–84% accuracy. In addition, a previous work has proposed a hybrid format for sparse matrices [SK12; BG09]. It allows the use of different formats for repre-
senting different subsets of a large sparse matrix. To select the format for each sub-matrix, it uses interpolation over a large offline collected performance database to estimate the performance if a certain format is used, based on its local features.

There are some other efforts trying to optimize the computations over sparse matrices, including hand-tuning input- or architecture-related features [BG09; Wil09; Liu13], designing new sparse formats [SK12; BG09; Kou11], and building automatically performance tuning (auto-tuning) systems [Li13; Cho10; Sed15; Vud05].

In a broader scope, there has been a large body of work applying machine learning techniques to solve program optimization difficulties. Examples include some that focus on improving lower-level compiler optimizations [Alm04; Fur08; Par11; Aga06], some on algorithmic selections [Ans09; Din15], and some on dynamic compilations and adaptations [Tia10].

To the best of our knowledge, this work is the first that explores the special challenges of applying CNN to HPC problems. Due to the special attributes of CNN, its usage in HPC faces some special complexities that prior applications of machine learning to HPC do not have, including the CNN structure designs and the input representations. It also brings some special opportunities, such as the use of transfer learning for alleviating cross-architecture portability difficulties. Explorations on these novel aspects are where the key contributions of this current work reside.
CHAPTER

7

OVERHEAD-CONSCIOUS SPARSE MATRIX FORMAT SELECTION

7.1 Motivation

Although current studies on sparse matrix format selection (including our work in Chapter 6) have given promise and valuable insights, they are all subject to an important limitation. They are overhead oblivious—that is, none of them have considered the implications of the overhead in employing the predictors and in adopting their predictions including the substantial overhead of the required format conversions.

Figure 7.1 illustrates the principled issue. In practical settings, the input matrix to an SpMV-based application often comes in only one default storage format. To use a better format, two steps have to happen, the prediction on which format to use, and the conversion of the matrix into that new format. In many cases, these two steps need to happen during the execution of the application. The overall time is hence the sum of these overhead ($T_p$ for prediction and $T_c$ for conversion) and the execution time ($T_e$) of the SpMV on the new format.

Prior studies have all tried to build predictors to predict the format that minimizes $T_e$, ignoring the influence of $T_p$ and $T_c$, which together can be even longer than the execution time of SpMV. As a result, even though previous predictors appear to have quite good prediction accuracies, the formats predicted by them frequently give inferior performance in practical usage. The problem
is fundamental. As Figure 7.2 shows, even if the prior method could achieve a perfect prediction accuracy, the results from them could still cause significant slowdowns to the overall executions.

The goal of this work is to solve the problem by putting the overhead into consideration for SpMV storage format prediction. The solution would need to create predictors that can accurately predict the overall effects (rather than just SpMV performance) of a new storage format.

Creating such a predictor is challenging. It faces a strand of new complexities.

First, it has more factors to consider. Previous predictive models make predictions based on only the features of the input matrix because those are the only factor determining which format makes SpMV run the fastest on the given architecture. However, when our objective becomes minimizing the overall time, matrix features are not sufficient anymore, because we need to compare the benefits with runtime overhead. The quantitative benefits from a new format of a matrix depend on how many times SpMV gets called on the matrix, which depends on both the matrix and the program code itself. Sparse matrix is already difficult to characterize; adding program code makes the development of the solution even harder.

Second, the creation of overhead-conscious predictors requires an appropriate design strategy to deal with the overhead. The overhead could be dealt with explicitly, by for instance, building up one predictor for each kind of overhead and then subtracting it from the predicted benefits. It could also be dealt with implicitly, by for instance, building up a single predictor that takes the features of the input matrix and the program and predicts which format gives the overall best performance. Different strategies have different pros and cons. Understanding them and finding the suitable strategy is the second open research problem.

Finally, no matter what strategy is used, we eventually create some kind of predictor such that it can predict the best format to use for a given sparse matrix. The problem is that because such a
Figure 7.2 The histogram of the overall speedups of program PageRank when it uses a previous decision tree-based predictor with an idealized 100% accuracy (for minimizing $T_e$ in Figure 7.1). The annotated numbers indicate percentage of samples with speedup $<1$, $=1$, $>1$ respectively; $<1$ means slowdown. CSR is the default format.

predictor often has to run at the program runtime and its overhead is non-trivial (as it often requires extracting matrix features), its own overhead could also get in the way. If its own overhead already exceeds the benefits, the program would suffer slowdowns. If we create another predictor $Y'$ to predict whether it is worth to run the format predictor, how do we deal with the overhead of that new predictor $Y'$? Creating yet another predictor $Y''$? How about its own overhead? This could lead us into a chicken-egg dilemma. How to effectively address this dilemma is the third research problem.

### 7.2 Design Choices and Challenges

As Figure 7.1 has mentioned, the run time of an SpMV-based program execution consists of the time to predict what format of matrices is desirable, the time to convert the matrix into that format, and the time to run the SpMV-based program on the matrix in the new format. The first two parts are runtime overhead.

We have considered various designs to treat the overhead for minimizing the overall execution time.

#### 7.2.1 Implicit Versus Explicit Treatment

The first set of designs we considered treat the overhead implicitly. They directly predict the overall execution time of the SpMV-based program.
For instance, one design we had is to train a predictor \( p \) that takes a program code, the original matrix, and a certain matrix format as input, and predicts the overall run time of the program. It can be represented as the following \( T_{overall} = p(G, A, f) \). The difficulty is that the predictor must apprehend the influence of the features of the program \( G \), the matrix \( A \), and the format \( f \) at the same time. All three components could have many dimensions to consider, with program being the most complex component.\(^1\)

An alternative design is to build a predictive model \( p_G \) specific to each given program \( G \). As it is specific to a particular program, it needs to learn only about the influence of \( A \) and \( f \): \( T_{overall} = p_G(A, f) \), which simplifies the construction process. However, the catch is the loss of the generality of the constructed predictor. One would need to go through the time-consuming process of predictor construction for each program.

Overall, methods with an implicit treatment to overhead suffer a tension between model generality and construction complexity.

To address the tension, we use a design that takes an explicit treatment to the overhead. This design is based on a more detailed view of the overall program run time. For a given matrix used by one or more SpMV statements in a program, if we allow format conversion of the matrix based on some format selector, the whole program execution time can be regarded as follows:

\[
T_{overall} = T_{predict} + T_{convert} + \left( \sum_i T_{spmv(i)} \times N_i \right) + T_{other},
\]

where,

- \( T_{overall} \): the overall execution time of the whole program.
- \( T_{predict} \): the time to predict what format to use for a given matrix.
- \( T_{convert} \): the time to convert the matrix into the desired format.
- \( T_{spmv(i)} \): the run time of the \( i \)th SpMV statement on the matrix.
- \( N_i \): the number of times the \( i \)th SpMV statement is invoked on the matrix.
- \( T_{other} \): the time spent by other parts of the program.

In all SpMV-based applications that we have examined, \( T_{other} \) is largely independent to the matrix format as those parts of code tend to use the SpMV results rather than the matrix itself. For them, \( T_{other} \) can be ignored in the format selection; we need to consider only the first three terms.

\(^1\)A variant of the predictor is to directly predict which format is the best (rather than time), which is still subject to the influence of all the components.
on the right-hand-side of $T_{overall}$. So the goal becomes to minimize

$$T_{affected} = T_{\text{predict}} + T_{\text{convert}} + \left( \sum_i T_{spmv(i)} \times N_i \right), \quad (7.2)$$

Our design is to build separate predictors to directly predict the overhead, single SpMV time $T_{spmv(i)}$, and $N_i$ respectively. They each work across programs, matrices, and formats, providing good generality. At the same time, they avoid many complexities the implicit designs face. For instance, the overhead and $T_{spmv(i)}$ are not affected by the features of the program $G$; $T_{\text{convert}}$ is determined by the matrix and formats only; $T_{spmv(i)}$ is determined by the matrix and the used format only (for a given SpMV library); $N_i$ is influenced by the features of the program $G$ and the matrix, but is independent of the matrix formats.

Because of the strengths in both simplicity and generality of the explicit design, we use it as the base for our developed solution.

### 7.2.2 Challenges

To effectively materialize the design, there are three major challenges.

- The first is that unlike previously built predictors that give out qualitative prediction results (i.e., which format works the best), the four predictors we need to build are all quantitative predictors, giving out numerical predictions. Consequently, the machine learning methods that showed effectiveness in the previous predictors cannot be applied to our problem, and in the same vein, the features of matrices/formats/programs found useful in the previous studies may not necessarily fit our predictors. Identifying the suitable machine learning methods and features to use is the first question we must answer. This challenge relates with the constructions of all our predictors.

- The second challenge is specific to the prediction time $T_{\text{predict}}$. It is the chicken-egg dilemma mentioned in the introduction. The prediction time $T_{\text{predict}}$ could be substantial as it typically requires the extraction of matrix features. Because the prediction happens during runtime, if the prediction result is to keep the format unchanged, the prediction would incur only runtime overhead and result in slowdowns to the program execution. If we add a predictor for predicting $T_{\text{predict}}$, that added prediction itself suffers the same problem.

- The third challenge is specific to the prediction of $N_i$. The value of $N_i$ is usually determined by the tripcount(s) of the loop(s) surrounding the call of SpMV. Predicting the tripcount of a loop is not easy as it depends on the algorithm implemented in the loop and the data involved in the loop execution. The former varies from program to program, and the latter varies even across the different runs of the same program. How to automatically model the relations
between programs and loop tripcounts is a difficult problem—it is in general equivalent to the halting problem the answer to which is undecidable.

In the next section, we present our solution in detail and explain how it addresses all the challenges.

7.3 Overhead-Conscious Predictor

This section first describes the choice of machine learning method we made for constructing the predictors, then presents our two-stage lazy-and-light scheme and how it helps with risk control, and finally explains some other important details in the predictors construction.

7.3.1 Learning Method

Previous format selection systems are oblivious to the overhead terms, and only need to make qualitative predictions (i.e., which format gives the shortest SpMV time). Hence, they all formalize the problem as a classification problem. In contrast, as the previous section has explained, when taking the overhead into consideration, we need the predictors to provide quantitative predictions. We can no longer model it as a classification problem. Instead, we need to build regression models to predict numerical values.

Regression models are models that take in some feature values (which could be numerical or categorical) and output some numerical predictions. There are many machine learning algorithms for constructing regression models, such as linear regression, Support Vector Regression (SVR), and so on. We select the learning method for our problem based on the following principles:

- The algorithm should give good prediction results;
- Due to the complexity of sparse matrices, the model should be robust and flexible in handling data with complex features;
- As our predictors are for on-line predictions, the algorithm should be efficient;
- It would be better if the produced models are interpretable.

 Among the machine learning methods used in practice, regression tree-based models meet the four requirements well. They have advantages in simple data preparation, robust performance to nonlinear relationship, and easy interpretable results (as the created trees are composed of questions on input features). The regression models they produce are also fast to run as they involve only a small number of questions on the data features and several linear algebraic operations (the leaf nodes in the trees are typically linear functions of data features). Moreover, when combined with
boosting methods, they have shown best prediction results and robustness in a variety of problems. In our work, we select the open-source package XGBoost [CG16], an efficient tree boosting system, to build our models. XGBoost is one of the most widely used tree boosting package, and has proven its effectiveness in many previous machine learning tasks [SC16; Vol17]².

7.3.2 Two-Stage Lazy-and-Light Scheme

As the previous section has mentioned, one of the barriers for deploying format predictors during runtime is that the predictor’s own overhead could already cause large slowdowns to the overall execution if the SpMV is called for only a few times on a given matrix. It is primarily due to the time needed for the predictor to extract the features from the given matrix (running the predictor takes little time as mentioned in the previous subsection.)

Our solution is a two-stage lazy-and-light scheme. As Figure 7.3 shows, the scheme consists of two stages of predictions. The first one is a lightweight predictor of the number of times the SpMV gets called on a matrix. The second stage does more sophisticated predictions and decides what format is the best format to use. The first predictor serves as a gateway; by comparing the predicted value \( LC \) to a threshold \( TH \), it decides whether it is worthwhile to invoke the second stage of prediction. If \( LC < TH \), no further prediction will be done and the program runs with the default format; otherwise, the second-stage prediction will be done, and depending on the prediction result, the matrix may remain as it is or be converted to a new format and used in the rest of the program execution.

²Although deep learning is now popular, it is best for classification rather than regression.
Such a scheme is designed to prevent the large prediction overhead from causing significant slowdowns when the executions are short-run executions. For the scheme to work effectively, it is important to ensure that the first stage takes virtually no time but still can provide reasonable prediction accuracies, and at the same time, the influence of its prediction errors could be controlled as much as possible.

We have explored several ways to implement the first-stage predictor. One method is to build it by modeling the relations between the features of matrices and the tripcounts of the SpMV-containing loops, such that the model can predict the tripcounts for an arbitrary matrix given to that loop. The relations are too complicated to capture; our results based on XGBoost show disappointing prediction accuracies. The method also incurs substantial runtime overhead as it needs the collection of matrix features.

We eventually settled on the following time-series–based lazy predictor and found it meeting our needs well. It is based on an observation that the loop surrounding SpMV in an SpMV-based application is often a loop for convergence. In a linear solver, for instance, each iteration of the loop computes the error from the current solution and the loop terminates when the error is smaller than a predefined threshold. We call the error the progress indicator of the loop. Other kinds of applications may have other kinds of progress indicators. Our solution is to build up a predictor that predicts the total number of iterations based on the sequences of the progress indicators of the first $k$ iterations of the loop. Its usage will be lazy in the sense that it is invoked in an execution of the target program only after the first $k$ iterations of the loop have passed and their progress indicators are revealed.

Such a lazy design has three benefits. First, it avoids the slowdowns that the prediction (and prediction errors) may cause to the loop if the loop has very few (less than $k$) iterations. For such short loops, even small runtime overhead could have some significant impact. Second, as the collected progress indicators reflect the dynamic behaviors of this current execution, they provide the predictors with clues specific to this run. Finally, the constructed predictor runs quickly. It does not need extracting features from matrices. It only needs to record several values of the progress indicator and then do several linear algebraic calculations. They take virtually no time compared to a SpMV execution. The predictors is built as a simple but efficient time series analysis model based on the RNN (recurrent neural network).

We make three notes. First, the goal of the two-stage scheme is not to ensure no slowdown (which could be easily achieved by avoiding any prediction or format conversion), but to maximize the overall speedups while avoiding large slowdowns in the unfavorable cases. The second stage of the prediction gets used only if the first stage predicts, after the first $k$ iterations of the loop, that the loop will run for at least another $T_H$ iterations. We empirically set both $k$ and $T_H$ to 15 in our implementations. Second, we do not claim the novelty of the time-series method for loop tripcount prediction. The approach shares lots of commonality with many other time-series–
based program behavior predictions. The main contributions in this part is that we go around the prediction overhead dilemma through this lazy-and-light two-stage design. Finally, our discussion focuses on convergence loops. For regular for loops, the problem is easier; the loop tripcount can be directly attained from the loop bounds at runtime.

7.3.3 The Second-Stage Prediction and Feature Selection

The second stage predicts the best format to use. Recall that our original goal is to find the format to minimize $T_{affected}$ in Equation 7.2. By this time, the first-stage prediction has already decided to run the second-stage prediction. So the same $T_{predict}$ is incurred no matter what format the predictor chooses. Therefore, the original goal is equivalent to minimize

$$T_{convert} + \sum_i T_{spmv(i)} * N_i.$$

We observe that if we divide the goal formula with a constant (normalization), the result of minimizing the normalized value is equivalent to minimizing the original formula.

This observation is useful because in our explorations, we found that directly predicting $T_{convert}$ and $T_{spmv(i)}$ is not as easy as predicting their normalized values (e.g., by $T_{spmv(0)}$ on the original matrix format). The plausible reason is that some environment biases common to the three times are canceled out by the divisions.

So, we build two predictors for the second stage prediction. They respectively predict the normalized conversion time and the normalized SpMV time on each matrix format. For a given old matrix format and a new format, both normalized times are primarily determined by the matrix features. The predictors are regression models on matrix features, constructed with XGBoost.

The construction process is straightforward. For instance, for the predictor of normalized $T_{convert}$ from given old to new formats, we collect the normalized $T_{convert}$ values on many matrices. For each, we create a tuple $(feature_1, feature_2, \ldots, feature_m, normalized\ T_{convert})$, where $feature_i$ is the value of the $i$th feature of the matrix. XGBoost then takes these tuples as training data and automatically constructs the predictor that predicts the normalized $T_{convert}$ from the features of an arbitrary given matrix.

The main challenge in building the regression models is on the identification of the important features of a sparse matrix. The feature should capture the characteristics of the matrix for the studied problem well. On the other hand, more features may increase the feature extraction overhead and artificially increase the needed number of training data to form the predictor. So the optimal choice of features is a trade-off among expressiveness, cost, and simplicity.

Prior studies [EF03; Li13; Vud03] have proposed more than 60 features to store the meta information about a matrix. It is essential to find those that are important for our predictors. Fortunately,
Table 7.1 The full set of feature candidates of a matrix.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>number of rows</td>
</tr>
<tr>
<td>N</td>
<td>number of columns</td>
</tr>
<tr>
<td>NNZ</td>
<td>number of nonzeros (NZ)</td>
</tr>
<tr>
<td>Ndiags</td>
<td>number of diagonals</td>
</tr>
<tr>
<td>NTdiags_ratio</td>
<td>the ratio of &quot;true&quot; diagonals to total diagonals, true diagonals represents one occupied mostly with NZ.</td>
</tr>
<tr>
<td>aver_RD</td>
<td>average number of NZ per row</td>
</tr>
<tr>
<td>max_RD</td>
<td>maximum number of NZ per row</td>
</tr>
<tr>
<td>min_RD</td>
<td>minimum number of NZ per row</td>
</tr>
<tr>
<td>dev_RD</td>
<td>the deviation of number of NZ per row</td>
</tr>
<tr>
<td>aver_CD</td>
<td>average number of NZ per column</td>
</tr>
<tr>
<td>max_CD</td>
<td>maximum number of NZ per column</td>
</tr>
<tr>
<td>min_CD</td>
<td>minimum number of NZ per column</td>
</tr>
<tr>
<td>dev_CD</td>
<td>the deviation of number of NZ per column</td>
</tr>
<tr>
<td>ER_DIA</td>
<td>the ratio of nonzeros in DIA data structure</td>
</tr>
<tr>
<td>ER_RD</td>
<td>the ratio of nonzeros in row-packed (ELL) structure</td>
</tr>
<tr>
<td>ER_CD</td>
<td>the ratio of nonzeros in column-packed structure</td>
</tr>
<tr>
<td>row_bounce</td>
<td>average difference between NNZs of adjacent rows</td>
</tr>
<tr>
<td>col_bounce</td>
<td>average difference between NNZs of adjacent columns</td>
</tr>
<tr>
<td>d</td>
<td>density of NNZ in the matrix</td>
</tr>
<tr>
<td>cv</td>
<td>normalized variation of NNZ per row</td>
</tr>
<tr>
<td>max_mu</td>
<td>$\text{max}<em>\text{RD} - \text{aver}</em>\text{RD}$</td>
</tr>
<tr>
<td>blocks</td>
<td>number of non_zero blocks</td>
</tr>
<tr>
<td>mean_neighbor</td>
<td>average number of NZ neighbors of an element</td>
</tr>
</tbody>
</table>

A benefit of using ensembles of decision tree methods like XGBoost is that they can automatically determine feature importance from a trained predictive model. To be specific, an XGBoost model is built using the whole feature set in Table 7.1. As a side product, the importance score of each feature can be calculated by the algorithm, allowing features to be ranked and compared to each other. The importance score indicates how useful each feature is in the construction of the tree-boosting model. Features with low importance score can be automatically pruned until the minimal set of features is retained without sacrificing the prediction performance.

In addition to the feature selection, some other standard methods are used in our predictors constructions, including grid search for parameter auto-tuning and cross validation for overfitting prevention.

The deployment of the overhead-conscious method is currently through library. For a given application, to use the method for runtime format selection and conversion, the user just needs
to replace the original SpMV call with our customized SpMV call, and insert code to record the values of the *progress indicator* of the surrounding loop. The customized SpMV call adds a wrapper to SpMV such that when the appropriate conditions are met, it calls the construction of Stage-1 predictor, or calls the Stage-2 predictor (which needs to be built only once on the system of interest) and the format conversion when necessary.

### 7.4 Evaluations

In this section, we report the evaluations of the efficacy of the proposed overhead-conscious format selection for SpMV-based applications. A quick summary is that it predicts the normalized format conversion time and the SpMV time with an average accuracy greater than 88% in most cases. It improves the overall performance of applications by 1.14X to 1.43X, significantly larger than the 0.82X to 1.24X upperbound speedups overhead-oblivious methods could give. We next describe the methodology and the full results.

#### 7.4.1 Methodology

**Hardware**

The evaluations are on a CPU-GPU platform as detailed in Table 7.2.

<table>
<thead>
<tr>
<th></th>
<th>Intel® CPU</th>
<th>NVIDIA® GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>Xeon E5-1607</td>
<td>GeForce GTX TITAN X</td>
</tr>
<tr>
<td>Freq.</td>
<td>3.00 GHz</td>
<td>1.08 GHz</td>
</tr>
<tr>
<td>Cores</td>
<td>4</td>
<td>3072</td>
</tr>
<tr>
<td>Memory</td>
<td>16GB DDR3 1.9 GHz</td>
<td>12GB GDDR5 3.5 GHz</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>34.1 GB/s</td>
<td>168 GB/s</td>
</tr>
<tr>
<td>OS/Driver</td>
<td>Ubuntu 16.04</td>
<td>CUDA 8.0</td>
</tr>
<tr>
<td>Compiler</td>
<td>GCC (6.2)</td>
<td>NVCC (8.0)</td>
</tr>
</tbody>
</table>

**Library, Formats, Applications**

In our evaluation, we focus on the CUDA-based SpMV libraries on the NVIDIA GPU. We adopt the NVIDIA® CUSP library [Dal14], which supports COO, CSR, DIA, ELL, HYB formats. We supplement it with the cuSPARSE library [Nau10] to support the BSR format. A previous work reports some promising performance of a new format CSR5 over some alternative formats and publishes the
implementation [Liu16]. We include its CUDA implementation to support CSR5 format. Format conversions are through the functions included in CUSP which runs on GPU.

The set of formats covered in this study are limited to the formats supported by these existing libraries. The support of these covered formats by these (commercial) libraries indicates their competitiveness and general applicability. The set unavoidably leaves some formats uncovered. With the idea verified, the approach can be easily extended to the selection of other formats.

CSR is the most commonly used default format in SpMV-based applications. It is hence used as the default format in our experiments.

We create a simple software framework, named SpMVframe, to help with a focused study of SpMV performance. It consists a loop with adjustable upperbounds that surrounds a call to SpMV. In addition, we also evaluate the technique on four real-world SpMV-based applications: PageRank [Pag99] is the popular web page ranking algorithm, BiCGSTAB [Vor92] implements the bi-conjugate gradient stabilized method, CG [HS52] is a conjugate gradient method, GMRES [SS86] is a linear equation system solver based on the generalized minimum residual method.

**Dataset**

Our experiments use the dataset from the SuiteSparse matrix collection [DH11]. These matrices include the 2757 real-world matrices (which were also used in the previous studies [Li13; Sed15]).

In our evaluation of the prediction model, we run the SpMV on all the matrices of all formats. However, not all runs are valid as some formats impose extra limitations on matrices. For example, the DIA and ELL require the fill ratio (the ratio of zeros to be padded in the storage) is within some threshold. And some applications (e.g., linear solver) works on only matrices meeting certain conditions. Only valid runs are considered in the performance comparisons; more details are given during our following result discussions.

**Cross Validation**

We use the same cross validation method as described in Section 6.6.

**7.4.2 Impact of the Overhead on Format Selection**

We first measure the impact of the format conversion overhead on format selection. This part uses no prediction but actual performance measurements. As Table 7.3 shows, converting a matrix to a different format takes a lot of time, equaling 9-270 calls of SpMV. And the time differs across different formats. As a result, the format minimizing SpMV often does not give the best overall performance. It is reflected by the largely different distributions of matrices shown in Table 7.4 in terms of their favorite formats in overhead-conscious (OC) and overhead-oblivious (OO) cases. Table 7.4 also
Table 7.3 The conversion (CSR to other formats) time normalized by a single SpMV on CSR.

<table>
<thead>
<tr>
<th>Other format</th>
<th>Conversion cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>COO</td>
<td>9</td>
</tr>
<tr>
<td>DIA</td>
<td>270</td>
</tr>
<tr>
<td>ELL</td>
<td>102</td>
</tr>
<tr>
<td>HYB</td>
<td>147</td>
</tr>
<tr>
<td>BSR</td>
<td>37</td>
</tr>
<tr>
<td>CSR5</td>
<td>26</td>
</tr>
</tbody>
</table>

Table 7.4 The number of matrices that favor each of the formats in the conversion overhead-oblivious (OO) and overhead-conscious cases (when the loop has 100 or 1000 iterations).

<table>
<thead>
<tr>
<th>Format</th>
<th>OO</th>
<th>OC (Iter=100)</th>
<th>OC (Iter=1000)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSR</td>
<td>195</td>
<td>1490</td>
<td>965</td>
</tr>
<tr>
<td>DIA</td>
<td>30</td>
<td>6</td>
<td>27</td>
</tr>
<tr>
<td>ELL</td>
<td>107</td>
<td>0</td>
<td>76</td>
</tr>
<tr>
<td>HYB</td>
<td>54</td>
<td>4</td>
<td>13</td>
</tr>
<tr>
<td>BSR</td>
<td>943</td>
<td>201</td>
<td>632</td>
</tr>
<tr>
<td>CSR5</td>
<td>582</td>
<td>210</td>
<td>198</td>
</tr>
</tbody>
</table>

shows that when overhead is considered, the best format changes with the number of iterations of the SpMV-surrounding loop (format conversion is done only once for a matrix in a run). These results confirms the importance of being overhead conscious in selecting matrix storage format.

7.4.3 Performance Comparison of Primary Predictors

As mentioned, our overhead-conscious solution consists of two stages. The first stage is a gateway mostly based on loop tripcount prediction. It is application specific. The second stage consists of our primary predictors that deal with the trade-off between format conversion overhead and conversion benefits. In this part, we give a focused study on the quality of the primary predictors, and compare the format they select with those from previous overhead-oblivious methods. The next section will report the performance of the whole overhead-conscious solution on real applications.

We use our SpMVframe for this study; by allowing easy change of loop bounds, it makes it convenient to examine the tradeoff between conversion overhead and benefits.
Table 7.5 Prediction errors of normalized format conversion time and SpMV time

<table>
<thead>
<tr>
<th>Format</th>
<th>No. of matrices</th>
<th>Error of conversion time</th>
<th>Error of SpMV time</th>
</tr>
</thead>
<tbody>
<tr>
<td>COO</td>
<td>1911</td>
<td>9.6%</td>
<td>18.0%</td>
</tr>
<tr>
<td>CSR</td>
<td>1911</td>
<td>8.1%</td>
<td>7.0%</td>
</tr>
<tr>
<td>DIA</td>
<td>630*</td>
<td>8.8%</td>
<td>8.3%</td>
</tr>
<tr>
<td>ELL</td>
<td>1331*</td>
<td>8.6%</td>
<td>10.0%</td>
</tr>
<tr>
<td>HYB</td>
<td>1911</td>
<td>8.3%</td>
<td>8.0%</td>
</tr>
<tr>
<td>BSR</td>
<td>1911</td>
<td>10.7%</td>
<td>15.0%</td>
</tr>
<tr>
<td>CSR5</td>
<td>1911</td>
<td>13.9%</td>
<td>11.5%</td>
</tr>
</tbody>
</table>

*The library allows only matrices meeting certain conditions (e.g., number of diagonals exceeds 20% for DIA) for DIA and ELL.

7.4.3.1 Prediction Accuracy and Speedup Comparisons

Recall that the primary predictors predict the normalized format conversion time and the normalized SpMV time on the new format. We use relative error as the metric, defined as

$$\frac{|\text{predicted value} - \text{actual value}|}{\text{actual value}}.$$

Table 7.5 reports the relative errors of the predictions by our primary predictors on each of the studied matrix formats. The accuracies are over 88% in most cases.

Figure 7.4 shows that the predictions are sufficient for the primary predictors to select the best matrix formats correctly in most cases, and produces significant speedups. The bars in Figure 7.4 report the speedups obtained in three ways. The Speedup<sub>OC</sub> bars are the speedups from our primary predictors. The T<sub>B<sub>OC</sub> bars are the upperbounds of the overhead-conscious method—that is, when the primary predictors have a perfect prediction accuracy. The T<sub>B<sub>OO</sub> bars are the upperbounds of the results from overhead-oblivious methods, which are obtained by picking the format that actually minimizes SpMV time (without considering format conversion time).

As the results in Figure 7.4 show, because of the impact of the conversion overhead, the decisions from T<sub>B<sub>OO</sub> cause large slowdowns when the SpMV-enclosing loop has a small number of iterations. When the number of iterations gets large, the speedups from that method is still lower than what the overhead-conscious method achieves. The differences between the Speedup<sub>OC</sub> bars and and the T<sub>B<sub>OC</sub> bars show that the speedup loss due to the prediction errors of our primary predictors is small across all different numbers of loop iterations. The results indicate the importance of being overhead-conscious, and suggest that the predictions from our primary predictors are accurate enough to keep most benefits of overhead-conscious matrix format selection.
Figure 7.4 The comparison of speedups based on SpMVframe. The baseline is the performance on the default CSR format.

7.4.3.2 Feature importance

A side product from tree-based predictors, XGBoost in particular, is that the predictor construction process can report the importance of the features of the input data. Figure 7.5 and 7.6 show the ranked list of the features that are the most important for the DIA and BSR formats. Different formats favor different features. For the DIA format, the most important feature is Ndiags, while for the BSR format, the most important features are aver_RD and block_ratio due to its block based storage format.

7.4.4 Performance on Entire Applications

Next, we report the overall speedups our solution brings to four real world applications, along with the evaluation of our first stage predictor.

Algorithm 2 shows the pseudo code of the first application, PageRank. The most time consuming part is the SpMV operation in line 4. It is surrounded by a convergence loop. Such a code pattern is representative for SpMV-based applications, also shown in the other three applications we tested.

The convergence checking statements in those applications give the progress indicators for the Stage-1 predictor to use for its prediction of loop tripcounts. The four applications exhibit different patterns in loop tripcounts. PageRank has a quite stable pattern, with loop tripcounts in the range of [1, 93], while BiCGSTAB shows a much larger range [1, 100000] (100000 is the preset upper bound). The errors in the exact numbers of predicted loop iterations vary, from 17% average on PageRank
Figure 7.5 Feature ranking for $T_{spm}$ of DIA format.

Figure 7.6 Feature ranking for $T_{spm}$ of BSR format.
Algorithm 2 PageRank algorithm

Input: Sparse matrix $A^{N \times N}$, damping factor $d$, convergence error $\varepsilon$
Output: Vector $R^{(n)}$

1: \[ \text{procedure PAGE RANK} \]
2: \[ R^{(0)}[i] \leftarrow 1/N, \forall i \in \{1...n\} \]
3: repeat
4: \[ R^{(k+1)} \leftarrow (1-\alpha)R^{(k)} + d(A^T \times R^{(k)}) \]
5: until $|P R^{(k+1)} - P R^{(k)}| < \varepsilon$

Table 7.6 (the Speed up OC column) shows the average speedups our method brings to the whole program executions (all runtime overhead is counted.) We also report the upperbound speedups (the $TB_{OC}$ column) of our method when there are no prediction errors, and the upperbound speedups (the $TB_{OO}$ column) of overhead-oblivious methods. Again, the performance of the default CSR format is used as the baseline. The prediction errors reduce the speedups of overhead-conscious method to a certain degree. It is most visible on GMRES which has the lowest prediction accuracies. However, even with that influence, the overhead-conscious method still significantly outperforms the upperbounds of the overhead-oblivious methods. The average speedups are significant, ranging from 1.14X to 1.43X.

Table 7.7 shows the distributions of the selected formats. Figure 7.7 shows the histogram of the speedup for PageRank. Compared with Figure 7.2, our selector largely avoids performance slowdowns caused by unnecessary format conversions. Selected new formats work well for most cases.

Table 7.8 reports the details of the predictions on several matrices of different sizes and densities.

to 62% on BiCGSTAB, 78% on CG, and 102% on GMRES. It is important to note that our ultimate goal is to decide whether to do the predictions and format conversions, rather than to get the precise loop tripcounts. Hence, there is a substantial amount of slack for tolerating tripcount prediction errors. For instance, even though a prediction of 10 for a 2-iteration loop means a 400% prediction error, the predictor still makes the right decision that no further predictions or conversions should be done as the loop is too small (smaller than the threshold). And for a loop, tripcounts of 1000 and 2000 could lead to the same conclusion for our ultimate questions as they are both so large that the overhead in predictions and format conversions is trivial compared to the benefits. More specifically, for our Stage-1 predictor, the predicted tripcount is taken to compare with the threshold $TH=15$ to decide whether it is worth proceeding into the more costly Stage-2 prediction. So despite the sometimes quite big errors in the predicted tripcounts, our Stage-1 predictor correctly predicts whether the tripcounts (after the first stage) of the loop exceed the threshold in most of the times: PageRank 93%, BiCGSTAB 82%, CG 76%, GMRES 65%.

The prediction errors have some influence on the overall benefits of the predictions, but the overall results still remain positive. Table 7.6 (the Speed up OC column) shows the average speedups our method brings to the whole program executions (all runtime overhead is counted.) We also report the upperbound speedups (the $TB_{OC}$ column) of our method when there are no prediction errors, and the upperbound speedups (the $TB_{OO}$ column) of overhead-oblivious methods. Again, the performance of the default CSR format is used as the baseline. The prediction errors reduce the speedups of overhead-conscious method to a certain degree. It is most visible on GMRES which has the lowest prediction accuracies. However, even with that influence, the overhead-conscious method still significantly outperforms the upperbounds of the overhead-oblivious methods. The average speedups are significant, ranging from 1.14X to 1.43X.

Table 7.7 shows the distributions of the selected formats. Figure 7.7 shows the histogram of the speedup for PageRank. Compared with Figure 7.2, our selector largely avoids performance slowdowns caused by unnecessary format conversions. Selected new formats work well for most cases.

Table 7.8 reports the details of the predictions on several matrices of different sizes and densities.
Table 7.6 Speedup of applications.

<table>
<thead>
<tr>
<th>Application</th>
<th>Speedup</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$TB_{OO}$</td>
<td>$TB_{OC}$</td>
<td>Speedup</td>
<td></td>
</tr>
<tr>
<td>PageRank</td>
<td>1.0762</td>
<td>1.4368</td>
<td>1.4307</td>
<td></td>
</tr>
<tr>
<td>BiCGSTAB</td>
<td>1.2454</td>
<td>1.3975</td>
<td>1.3375</td>
<td></td>
</tr>
<tr>
<td>CG</td>
<td>0.8246</td>
<td>1.1449</td>
<td>1.1416</td>
<td></td>
</tr>
<tr>
<td>GMRES</td>
<td>1.0136</td>
<td>1.2505</td>
<td>1.2034</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.7 The numbers of matrices that favor each of the formats in the four applications.

<table>
<thead>
<tr>
<th>Applications</th>
<th>PageRank</th>
<th>BiCG</th>
<th>CG</th>
<th>GMRES</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OO</td>
<td>OC</td>
<td>OO</td>
<td>OC</td>
</tr>
<tr>
<td>F</td>
<td>CSR</td>
<td>71</td>
<td>320</td>
<td>32</td>
</tr>
<tr>
<td>o</td>
<td>DIA</td>
<td>12</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>r</td>
<td>ELL</td>
<td>38</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>m</td>
<td>HYB</td>
<td>24</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>a</td>
<td>BSR</td>
<td>367</td>
<td>129</td>
<td>166</td>
</tr>
<tr>
<td>t</td>
<td>CSR5</td>
<td>86</td>
<td>149</td>
<td>31</td>
</tr>
</tbody>
</table>

The consideration of overhead leads our predictor to select entirely different formats from those selected by the ideal overhead-oblivious method on four out of the five matrices. For instance, on matrix shallow_water2, even though HYB can make the SpMV run faster, our stage-1 predictor correctly predicts that it is not worthwhile because the runtime overhead will likely outweigh the benefits. By leaving the format unchanged, it avoids the substantial slowdowns the overhead-oblivious method incurs. Because stage-1 predictor takes virtually no time, it adds no noticeable overhead to the program execution. The stage-2 predictor has some larger overhead (about 2X–4X SpMV time) as it needs to extract matrix features. However, the two-stage design and the lazy-and-light scheme ensure that it does not get invoked in short program executions, as in the shallow_water2 case.

Table 7.8 Speedup comparison for selected matrices. (PR: PageRank, Sp: Speedup)

<table>
<thead>
<tr>
<th>App.</th>
<th>Matrix</th>
<th>NNZ</th>
<th>Row(Col)</th>
<th>Iter.</th>
<th>Fmt$_{OO}$</th>
<th>Fmt$_{OC}$</th>
<th>Sp$_{OO}$</th>
<th>Sp$_{OC}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR</td>
<td>nlpkkt120</td>
<td>95117792</td>
<td>3542400</td>
<td>49</td>
<td>ELL</td>
<td>CSR5</td>
<td>0.89</td>
<td>1.297</td>
</tr>
<tr>
<td>PR</td>
<td>shipsec1</td>
<td>3568176</td>
<td>140874</td>
<td>48</td>
<td>CSR5</td>
<td>CSR5</td>
<td>1.17</td>
<td>1.17</td>
</tr>
<tr>
<td>PR</td>
<td>pwtk</td>
<td>11524432</td>
<td>217918</td>
<td>46</td>
<td>BSR</td>
<td>CSR5</td>
<td>0.95</td>
<td>1.245</td>
</tr>
<tr>
<td>BiCG</td>
<td>shallow_water2</td>
<td>81920</td>
<td>327680</td>
<td>11</td>
<td>HYB</td>
<td>CSR5</td>
<td>0.03</td>
<td>1.0</td>
</tr>
<tr>
<td>CG</td>
<td>torso3</td>
<td>259156</td>
<td>4429042</td>
<td>45</td>
<td>ELL</td>
<td>CSR5</td>
<td>0.93</td>
<td>1.49</td>
</tr>
</tbody>
</table>
Overall, the upperbounds of speedups that previous overhead-oblivious methods can bring are only 0.82X–1.24X. (Recall that less than one means slowdown.) In contrast, our method, on average, improves the overall performance of applications by 1.14X–1.43X. The results demonstrate that the proposed method is effective in overcoming the limitations of prior methods in selecting the storage format for SpMV-based applications. The experiments focus on GPU, but as SpMV selection is also important for CPU [Li13; Yil16], we expect that the technique could help CPU executions as well; the exploration is left for future studies.

Regarding the prediction overhead, the time of the time-series model and the XGBoost model is constant with value of 2ms and 5ms, respectively. The feature extraction overhead varies with a range of 2X–4X of an SpMV call.

### 7.5 Related work

There has been a number of studies on format selection for SpMV. For example, the SMAT work [Li13] built up a decision tree for selecting the best storage format for a sparse matrix storage. A similar classification-tree-based model was used in [Sed15] and an SVM classification model was used in [Ben16].

None of these studies have taken the overhead into account when predicting the best format. A recent work [Yil16] discusses the “break-even point”, which is the minimal number of SpMV calls needed for the conversion benefits to outweigh the overhead. This concept is overhead conscious, but the work does not integrate it into the prediction model, leaving the decision to users.
There are some other efforts trying to optimize the computations over sparse matrices, including building automatically performance tuning (auto-tuning) systems [Li13; Cho10; Sed15; Vud05], designing new sparse formats [SK12; BG09; Kou11], and hand-tuning input- or architecture-related features [BG09; Wil09; Liu13].

In a broader scope, there have been a large body of work applying machine learning techniques to solve program optimization difficulties. Examples include some on algorithmic selections [Ans09; Din15], some on improving lower-level compiler optimizations [Alm04; Fur08; Par11; Aga06], and some on dynamic compilations and adaptations [Tia10].
Part IV

Epilogue
8.1 Conclusions

The PATO work in Chapter 3 demonstrates the promise of ontology-based knowledge engineering to overcome the three major limitations of today’s declarative program analysis. The four types of data analyses on PATO show that a single ontology is able to support multiple different program analyses efficiently. The experience shows some promise of the integration of ontology into program analysis. Establishing this new way of program analysis, however, requires the development of an ontology for program analysis and some deep investigation of the opportunities that ontology may bring to program analysis and optimizations.

In Chapter 4, we extend the ontology-based knowledge engineering to address a more ambitious problem – how to efficiently accumulate, share, and reuse various knowledge of the HPC community. Therefore, we have introduced OpenK, the first infrastructure that offers some systematic support to this problem. 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 code optimization and for enhancing the extensibility of software optimizers. For example, the GPU
data placement experiment shows the promise of ontology for seamlessly linking knowledge from different sources, extending declarative program analysis with the capability to effectively guide program optimizations. The cooperative Liveness analysis demonstrates that with ontology-based OpenK framework, collaborations among different compiler tools become simpler and the synergy turns out to be quite beneficial.

Part III exploits how to leverage machine learning techniques to acquire and capitalize HPC knowledge. We choose a pillar HPC problem – the sparse matrix format selection problem – as the case study. In Chapter 6, we present a systematic exploration on bridging the gap between deep learning and sparse matrix format selection. It points out three-fold special challenges sparse matrix format selection poses to the application of CNN: input matrix representation, CNN structure design, and the needs for cross-architecture migrations of the learned models. To tackle each of the challenges, it makes a set of innovations—including several novel matrix representations, use of late-merging CNN structure, and the use of transfer learning for reducing the large cost of cross-architecture model migrations.

In Chapter 7, this thesis further points out the importance and challenges of being overhead-conscious for sparse matrix format selection. We present the first end-to-end overhead-conscious solution for sparse matrix format selection, including an ensemble of predictive models for cost-benefit analysis and a lazy-and-light scheme for slowdown prevention. The evaluation on 2757 matrices and four real-world applications demonstrates the significant benefits of the proposed method. It shows that the overhead-conscious solution is an optimal treatment for the sparse matrix format selection problem.

8.2 Future Studies

To lower the barrier for modern HPC development, the PATO and the OpenK work are just preliminary steps. For example, the ontology implemented in the PATO work is for only the representations of C programs and the GPU hardware knowledge is partially implemented in the OpenK work. A full development of ontology-based program analysis would require also the formal definitions of common concepts in program analysis (e.g., dominator, post order, alias), their relations and some deep investigations of the opportunities that ontology may bring to program analysis. Besides, a comprehensive development of ontology about the HPC hardware information, expert knowledge is required to enable more sophisticated program analysis, which ideally can reveal insights beyond the capability of traditional compiler tools. Therefore, we hope that the work in Part II will encourage further investigations into this direction.

For the machine learning based knowledge engineering, the SpMV format selection problem is just one application. There are many other uses of sparse matrices. We foresee that the potential of proposed techniques and methods (e.g., the deep learning, transfer learning or the lazy-and-light
scheme) may go well for more sparse matrix applications.

Besides, a more interesting direction is to use deep learning techniques for understanding and analyzing the source code directly.


Wielemaker, J. *SWI-Prolog Semantic Web Library 3.0*.


