An Automatic Synthesizer of Advising Tools for High Performance Computing

Hui Guan, *Member, IEEE*, Xipeng Shen, *Senior Member, IEEE*, and Hamid Krim, *Fellow, IEEE*

Abstract—This article presents Egeria, the first automatic synthesizer of advising tools for High-Performance Computing (HPC). When one provides it with some HPC programming guides as inputs, Egeria automatically constructs a text retrieval tool that can advise on what to do to improve the performance of a given program. The advising tool provides a concise list of essential rules automatically extracted from the documents and can retrieve relevant optimization knowledge for optimization questions. Egeria is built based on a distinctive multi-layered design that leverages natural language processing (NLP) techniques and extends them with HPC-specific knowledge and considerations. This article presents the design, implementation, and both quantitative and qualitative evaluation results of Egeria.

Index Terms—Performance tools, natural language processing, code optimization

1 Introduction

Achieving high performance on computing systems is challenging. It requires programmers to have a deep understanding of the underlying computing systems and make proper implementations to effectively harness the computing power. The problem becomes more complicated with the rapid changes and increasing complexity of modern systems (e.g., many-core heterogeneous systems equipped with Graphic Processing Units) because the set of knowledge and specifications programmers have to master grows fast and continuously. Although performance profiling tools (e.g., HPCToolkit [1], NVProf [2]) alleviate the problem by identifying the potential issues, they do not provide many guidelines on how to optimize the code to address the issues. Coming up with available solutions still demands lots of expertise specific to the underlying architecture.

Programming and optimization guides usually contain optimization rules. For example, both NVIDIA and AMD have published guides [3], [4] explaining the many intricate features of their Graphic Processing Units (GPUs) and programming models, the detailed guidelines and methods for developing code that runs efficiently on each major GPU model. Programmers could read them and try to apply what they've learned to optimize their code. Such documents, however, are often hundreds of pages long. It is difficult for application programmers to master and memorize all the knowledge, and quickly come up with all the relevant guidelines to apply when they encounter a specific program optimization problem.

In this work, we propose a framework named Egeria[1] to bridge the gap between programmers' demands for optimization guidelines and the hard-to-master programming guides. Egeria consists of two stages. The first stage is *advising sentence recognition*. When one provides Egeria with some HPC programming guides as inputs, it extracts a concise list of essential rules, called *advising sentences*, from the documents. The second stage is *knowledge recommendation*, which builds a text retrieval (TR) agent to interactively offer suggestions for specific optimization questions. The TR agent together with the list of advising sentences compose an advising tool synthesized by Egeria.

With such advising tools, programmers no longer need to memorize every optimization guideline or spend time to search. When encountering an optimization problem, they can just feed the advising tool either a performance profiling report of an execution of interest or some queries on how to solve certain specific performance issues. The tool will immediately provide a list of guidelines for solving those performance problems.

Recognitions of advising sentences require the analysis of the semantic and syntax of the sentences through some Natural Language Processing (NLP) techniques. Egeria adopts a multi-layered scheme guided by HPC domain-specific properties. Advising sentences in programming guides for HPC share some common syntactic and semantic patterns and some special words and phrases related to performance improvements in HPC. Exploiting such features helps significantly simplify the problems. The multi-layered design integrates the HPC domain properties into the NLP techniques in each of the layers. Through treatments at the levels of keywords, syntactic structures, and semantic roles guided by the HPC special features, Egeria is able to successfully recognize advising sentences from raw programming guide documents. Coupled with some text retrieval techniques (VSM [5] and

• *Hui Guan is with University of Massachusetts Amherst, MA 01002. E-mail: huiguan@cs.umass.edu .*
• *Xipeng Shen and Hamid Krim are with North Carolina State University, Raleigh, NC 27695. E-mail: {xshen5, ahk}@ncsu.edu.*

Manuscript received 23 Sept. 2019; revised 15 Aug. 2020; accepted 18 Aug. 2020. Date of publication 21 Aug. 2020; date of current version 4 Sept. 2020.
(Corresponding author: Hui Guan.)
Recommended for acceptance by R. Yahyapour.
Digital Object Identifier no. 10.1109/TPDS.2020.3018636

1. The name comes from a nymph Egeria in Greek mythology who gives wisdom and prophecy.

TF-IDF [5]), Egeria accurately finds the relevant advising sentences for users' queries.

It is worth mentioning that Egeria itself is not a TR system but *a generator* of TR systems for various HPC domains. Having an easy-to-use generator of advising tools is essential for meeting the needs of HPC, thanks to its many domains and the fast changes in each of them. To our best knowledge, Egeria is the first auto-synthesizer of advising tools for HPC.

We conduct both quantitative and qualitative experiments to evaluate Egeria. In the experiments, advising tools are generated for CUDA programming on NVIDIA GPUs, OpenCL programming on AMD GPUs, and Xeon Phi programming on Intel Xeon Phi coprocessors. Egeria is able to recognize the advising sentences from these programming guides with over 80 percent precision rates, significantly higher than other alternative methods. Its two-stage design makes it able to answer CUDA program optimization queries with a 80-100 percent accuracy, substantially higher than a single-stage design. Two user studies also demonstrate the overall usefulness of Egeria in easing their efforts in optimizing programs. This work extends our conference paper [6] in several aspects: (1) An adjustable relevance factor is added for users to control the number of retrieved results in Section 4; (2) A sensitivity study on the factor is reported in Section 5.3. (2) Several semantic-based techniques are explored for improving both knowledge recommendation and advising sentence recognition components in Sections 6 and 7. (3) A dependency-parsing selector is proposed to replace the SRL-based selector in Section 7.

2 Overview

Our goal is to enable automatic synthesis of advising tools that can give advice on what to do to improve the performance of a given program. We call those advice "relevant advising sentences". Formally, we define "relevant advising sentences" as sentences in a given document that can serve as actionable solutions for an input query on improving certain performance aspects of a program (e.g., "how to improve memory throughput"). To determine whether each of the sentences in the given document belongs to the category of "relevant advising sentences" for the given query is a binary classification problem. This section gives an overview of our solution.

Egeria uses a two-stage design. As the top row in Fig. 1 shows, the two stages consider the "advising" and "relevance" aspects respectively. The two boxes at the bottom part of Fig. 1 give the more detailed illustrations of the two stages. The first stage, *advising sentence recognition*, recognizes all advising sentences from the given document. The second stage, *knowledge recommendation*, retrieves, from the set of advising sentences collected in the first stage, the sentences relevant to the input query through *text retrieval* methods, and returns them as answers to the user. The output from the first stage can also be directly reviewed by the user as a reminding summary of all the essential guidelines contained in the input document.

The first stage is more challenging due to the limited efficacy of existing NLP techniques. Egeria overcomes the difficulties by adopting a multi-layered scheme guided by some HPC domain-specific properties, as the left bottom box in Fig. 1 shows. It builds its second stage upon two key text retrieval techniques, namely the VSM representations and

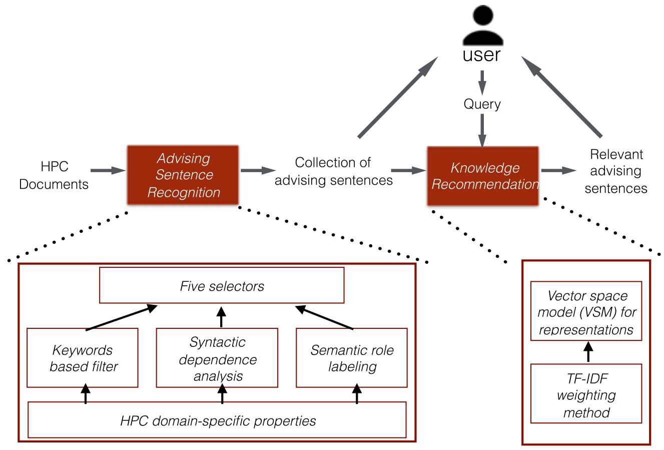


Fig. 1. Overview of Egeria. The two boxes at the bottom illustrate the two stages of Egeria, respectively.

the TF-IDF weighting method. We provide a detailed explanation on the first stage in Section 3 and the second stage in Section 4.

3 Advising Sentence Recognition

Recognizing advising sentences requires the analysis of the semantic and syntax of the sentences through some NLP techniques. The main challenge is the limited efficacy of each individual existing NLP techniques.

Two key features of Egeria help it circumvent those difficulties. (1) It leverages some important properties of HPC domains, including the common patterns in the suggesting sentences in programming guides for HPC, and the importance of some special words and phrases related with performance improvements in HPC. These significantly simplify the problem. (2) It adopts a multi-layered design, employing techniques at the levels of keywords based filtering, syntactic dependence analysis, and semantic role labeling. The combination creates a synergy for one technique to complement the weaknesses of another. Meanwhile, it effectively integrates the HPC domain knowledge into the NLP techniques at each of the layers. Together, these techniques lead to five selectors that work as an assembly to recognize advising sentences with a high accuracy. We next explain these two features in more detail.

3.1 HPC Domain-Specific Properties

According to our observations on some HPC documents, advising sentences of HPC are often featured with certain patterns along with some key words. We crystallize the observations into six categories as shown in Table 1 and five sets of keywords as shown in Table 2.

As Table 1 shows, the first category corresponds to sentences that contain some critical keywords (e.g., "good choice" in the example sentence). Our observation shows that appearances of such keywords can usually offer a sufficient indication, regardless of the forms of the sentences. We put together a collection of such keywords as FLAGGING_WORDS shown in Table 2.

The second category includes sentences that involve comparative relations that are formed with certain optimization-related words (part of XCOMP_GOVERNORS in Table 2).

TABLE 1
HPC Advising Sentence Categories

Categories	Patterns	Example Sentences (w/ key words underlined)	Selection Rules*	Key Techniques
I	Contains certain keywords	This can be a good choice when the host does not read the memory object to avoid the host having to make a copy of the data to transfer.	#1: $\exists$ w in S, w$\in$ FLAGGING_WORDS	Keyword Matching
II	Certain kind of comparative sentences	Thus, a developer may prefer using buffers instead of images if no sampling operation is needed.	#2: lemma(g) $\in$ XCOMP_GOVERNORS where g is the governor in a xcomp or ccomp relation	Syntactic Dependence Parsing
III	Certain kind of passive sentences	This synchronization guarantee can often be leveraged to avoid explicit clWaitForEvents() calls between command submissions.		
IV	Certain kind of imperative sentences	Pinning takes time, so avoid incurring pinning costs where CPU overhead must be avoided.	#3: $\exists v$ in S, v's governor is ROOT and v $\in$ IMPERATIVE_WORDS	
V	Sentences with certain subjects	For peak performance on all devices, developers can choose to use conditional compilation for key code loops in the kernel, or in some cases even provide two separate kernels.	#4: lemma(d) $\in$ KEY_SUBJECTS where d is the dependent in a nsubj relation	
VI	Sentences with certain purposes	The first step in maximizing overall memory throughput for the application is to minimize data transfers with low bandwidth.	#5: $\exists v$ as the predicate of a component c in S, p $\in$ KEY_PREDICATES and c doesn't have a A0 tag.	Semantic Role Labeling

* *(S: a given sentence; Upper-cased words: sets of keywords shown in Table 2).*

The third category includes some passive sentences that involve certain optimization-related keywords (part of XCOMP_GOVERNORS in Table 2).

The fourth category includes imperative sentences that involve words included in IMPERATIVE_WORDS shown in Table 2. Such a form of sentence is a frequent form used by suggesting sentences, and those keywords hint on their relevance with performance optimizations.

The fifth category includes sentences whose subjects are developer, programmer, or other special words contained in KEY_SUBJECTS in Table 2.

The final category consists of sentences with a purpose clause related with performance optimizations.

Except the first category, the patterns in the other categories are related with either the syntactic or semantic structure of the given sentence. We employ a series of NLP techniques to construct five selectors to help recognize the six patterns from an arbitrarily given sentence, as explained next.

3.2 Five Selectors

The five selectors we have developed work in a series. From the first to the fifth, they try to check whether the given sentence meets a certain condition. As long as the sentence meets the condition of one of the selectors, it is considered to be an "advising sentence".

3.2.1 Keyword Marching and Selector 1

The first selector is for the recognition of the first category in Table 1. It is a simple keyword matching process. One minor complexity is that one word could be in many different variations of form, such as, "argue", "argued", "argues", and "argument". We use the standard stemming technique in NLP to reduce all the forms into the stem of the word (e.g., "argu"). We do that for all the words in FLAGGING_WORDS and those in the given sentence before conducting the keyword matching. The principal rule of this selector can be formally expressed as follows:

Rule 1. *A sentence is an advising sentence if it contains at least one of the keywords in the FLAGGING_WORDS.*

3.2.2 Dependency Parsing and Selectors 2,3,4

The next three selectors are for categories 2, 3, 4, and 5. As these categories are all about syntactic structures of the sentence, these selectors are all based on *syntactic dependency parsing*. Dependency parsing is an automatic syntactic analysis approach that analyzes the grammatical structure of a sentence. It focuses on analyzing binary asymmetrical relations (called dependency relations) between words within a sentence [7]. Dependency parsing has been successfully

TABLE 2
Sets of Keywords Used in the Selectors

FLAGGING_WORDS	'better', 'best performance', 'higher performance', 'maximum performance', 'peak performance', 'improve the performance', 'higher impact', 'more appropriate', 'should', 'high bandwidth', 'benefit', 'high throughput', 'prefer','effective way', 'one way to', 'the key to', 'contribute to', 'can be used to', 'can lead to', 'reduce', 'can help', 'can be important', 'can be useful', 'is important', 'help avoid', 'can avoid', 'instead', 'is desirable', 'good choice', 'ideal choice', 'good idea', 'good start', 'encouraged'
XCOMP_GOVERNORS	'prefer', 'best', 'faster', 'better', 'efficient', 'beneficial', 'appropriate', 'recommended', 'encouraged', 'leveraged', 'important', 'useful', 'required', 'controlled'
IMPERATIVE_WORDS	'use', 'avoid', 'create', 'make', 'map', 'align', 'add', 'change', 'ensure', 'call', 'unroll', 'move', 'select', 'schedule', 'switch', 'transform', 'pack'
KEY_SUBJECTS	'programmer', 'developer', 'application', 'solution', 'algorithm', 'optimization', 'guideline', 'technique'
KEY_PREDICATES	'maximize', 'minimize', 'recommend', 'accomplish', 'achieve', 'avoid'

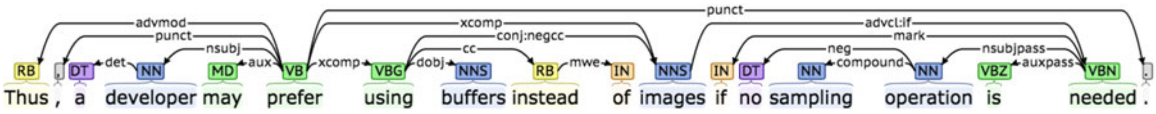


Fig. 2. Dependency structure for a sentence in Comparative Sentence category. xcomp(prefer, using).

applied to information extraction and text analysis [8]. A dependency relation is composed of a subordinate word (called the *dependent*), a word on which it depends (called the *governor*), and an asymmetrical grammatical relation between the two words.

Fig. 2 shows the dependency structure for an example sentence generated by the Stanford CoreNLP dependency parser [9]. The dependency relations are represented as arrows pointing from a governor to a dependent. Each arrow is labeled with a dependency type. For example, the noun *developer* is a dependent of the verb *prefer* with the dependency type *nominal subject* (nsubj) while it is a governor of the article *a* with the dependency type *determiner* (det). Dependency relations are usually written in the format: relation(governor, dependent) [10]. The relations in the two aforementioned examples are written as nsubj(prefer, developer) and det(developer, a). For the uniformity of representation, a virtual governor ROOT and a virtual relation "root" are used when expressing a word without an actual governor in the sentence. For example, for the verb *prefer* in the sentence Fig. 2, one may write the following: root(ROOT, prefer).

Selector 2 takes advantage of dependency parsing to detect sentences in category II (certain comparative sentences) and category III (certain passive sentences). It specifically checks a dependency relation *open clausal complement* (xcomp) and *clausal complement* (ccomp). The definition of xcomp relations is as follows: The governor of an xcomp relation is a verb or an adjective while the dependent is a predicative or clausal complement without its own subject [10]. For example, in Table 1, the given sentences in categories II and III have relations xcomp(prefer, using) and xcomp(leveraged, avoid) respectively. ccomp is similar to xcomp. The principal rule used by Selector 2 is as follows:

Rule 2. *A sentence is an advising sentence if it contains the following dependency relation: xcomp(*g*, *) or ccomp(*g*, *), where,* g $\in$ *XCOMP_GOVERNORS.*

Selector 3 is about the relevant imperative sentences. An imperative sentence is a type of sentence that gives advice or instructions or that expresses a request or command, as illustrated by the example sentence in Table 1 Category IV. Such sentences can be recognized based on such a feature: The root verb (i.e., the principal verb) in the sentence shall have no subject dependent. There are two complexities to note. First, the subject of a verb could have two types: *nominal subject* (nsubj) and *passive nominal subject* (nsubjpass). A nominal subject is a noun phrase which is the syntactic subject of a clause, such as "instructions" in the sentence "the scalar instructions can use up to two SGPR sources per cycle". A passive nominal subject is that of a passive clause [7], such as "allocations" in the sentence "all allocations are aligned on the 16-byte boundary". Both types of subjects should be checked and neither should appear in the sentence. Second, the sentence must at the same time be relevant to HPC optimizations. We notice that the root verb in such sentences provide good hints in this aspect. Specifically, the selector checks whether the root verb is part of the IMPERATIVE_WORDS in Table 2, and label the imperative sentence as an HPC advising sentence if so. To address the complexities in the various verb tenses, we use the lemma of a verb, which is the verb's canonical form (e.g., "run" for "runs", "ran", "running"). The principal rule used by Selector 3 is as follows:

Rule 3. *A sentence is an advising sentence if its* root *verb v meets both of the following conditions:*

1) *lemma(v) $\in$ IMPERATIVE_WORDS;*
2) *v is not in* nsubj *or* nsubjpass *dependency relations.*

Selector 4 is for category V, sentences with certain kinds of subjects (e.g., "developers" in the category V example sentence in Table 1). It finds out the dependent of nsubj relations and then checks whether they belong to the KEY_SUBJECTS set. The principal rule used by this selector is as follows (lemma gets the canonical form of the words):

Rule 4. *A sentence is an advising sentence if it contains the nsubj dependency relation and the lemma of the dependent $\in$ KEY_SUBJECTS.*

3.2.3 Semantic Role Labeling and Selector 5

Selector 5 treats category VI. This category involves the semantic roles (e.g., purpose) of the parts of the sentence. The selector hence employs semantic role labeling (SRL). Because SRL is generally a more complex task compared with dependency parsing and thus more error-prone, we will discuss the possibilities of getting rid of SRL by considering specific dependency patterns in Section 7.

Semantic role labeling, also called shallow semantic parsing, is an approach to detecting the semantic arguments associated with predicates or verbs of a sentence and classifying them into specific semantic roles. Semantic arguments refer to the constituents or phrases in a sentence. Semantic roles are representations that express the abstract roles that arguments of a predicate take that reveal the general semantic properties of the arguments in the sentence.

Fig. 3 shows an example attained through a SRL Demo[2] [11]. The demo follows the definition of semantic roles encoded in the lexical resource PropBank [12] and CoNLL-2004 shared task [13]. There are six different types of arguments labeled as A0-A5. These labels have different semantics for each verb as specified in the PropBank Frames scheme. In addition, there are also 13 types of adjuncts labeled as AM-XXX where XXX specifies the adjunct type. In the example, V is the predicate, A0 the subject, A1 the object, A2 the indirect object, AM-PNC the purpose. The example shows three "SRL" columns, with each corresponding to one semantic role relation centered on one verb. The first "SRL" column, for instance, centers around

2. http://cogcomp.cs.illinois.edu/page/demo_view/srl

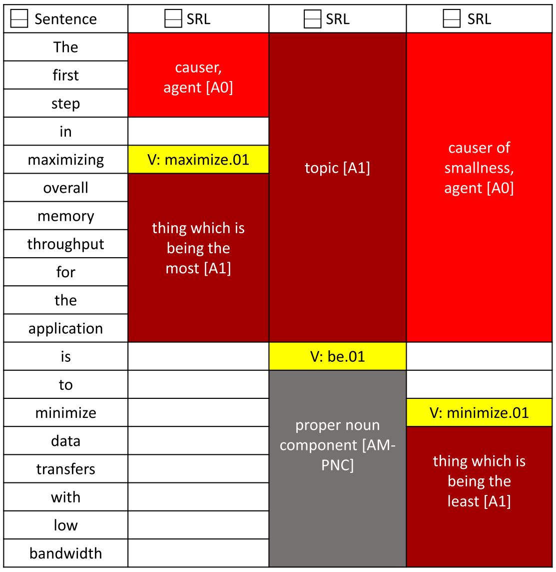


Fig. 3. Semantic role labeling results for a sentence.

the verb 'maximize'. This verb takes the meaning of maximize.01 in the PropBank and has a subject 'The first step' and an object 'overall memory throughput for the application'. The purpose argument for the verb 'be' also contains a predicate 'minimize' and its object 'data transfer with low bandwidth'.

Selector 5 uses SRL to detect sentences with purpose clauses. It particularly seeks for the purposes related to HPC optimizations. The predicate of the purpose clause usually offers good hints on the relevance. Our empirical study shows that, instead of finding the purpose clause of the sentence, the selector can simply check whether the predicate of a labeled argument (sentence component) belongs to KEY_PREDICATES shown in Table 2. The principal rule of this selector is put as follows:

Rule 5. *A sentence is HPC advising sentence if it meets all the following conditions:*

1) *the sentence contains an argument arg whose predicate $v \in$ PREDICATE_SET.;*
2) *the arg doesn't have any word with label A0.*

We implement the selectors based on several NLP tools. We use Stanza [14] for dependency parsing, AllenNLP [15] for SRL, and NLTK [16] for tokenization, word stemming and lemmatization. The design of the selection rules and keywords and NLP uses, are currently based on our observations about advising sentences found in HPC guides. The approach is possible to apply to non-HPC domains; some extensions in the design (keywords, rules, NLP uses) might be necessary.

4 Knowledge Recommendation

The second stage of Egeria is modeled as a text retrieval problem. It builds a recommendation engine that tries to identify advising sentences that are closely related with a given query. Our exploration shows that two techniques, vector space model (VSM) and term frequency-inverse document frequency (TF-IDF), suit the problem well.

VSM [5] is used to represent a sentence (the query or an advising sentence) in a feature vector form. It prepares for the relevancy calculations. VSM represents a piece of text as a vector of indexed terms. Each dimension corresponds to a separate term. If a term occurs in the text, its value in the vector is non-zero—the exact value is computed based on TF-IDF [5], one of the best-known weighting methods. In TF-IDF, the weight vector for a sentence s is $\mathbf{v}_s = [w_{1,s}, w_{2,s}, \ldots, w_{N,s}]^T$. Each entry is computed as

$$w_{t,s} = tf_{t,s} * \log \frac{|S|}{|\{s' \in S | t \in s'\}|}, \tag{1}$$

where $tf_{t,s}$ is the term frequency of term t in the sentence and $\log \frac{|S|}{|\{s' \in S | t \in s'\}|}$ is the inverse sentence frequency. $|S|$ is the total number of sentences in the sentence set and $|\{s' \in S | t \in s'\}|$ is the number of sentences containing the term t. The sentence similarity between a sentence s and a query q is calcuated as cosine similarity

$$sim(s,q) = \frac{\mathbf{v}_s^T \mathbf{v}_q}{\|\mathbf{v}_d\| \|\mathbf{v}_q\|}. \tag{2}$$

Our implementation of VSM is based on Gensim [17].

An advising tool produced by Egeria reports the top-ranked sentences (having a similarity score higher than an *adjustable* similarity threshold) as the answer to user's query. We use 0.15 as the default similarity threshold. Users can easily adjust the similarity threshold through the interface to control the number of advising sentences they get. To make the sentences easy to understand, the answer is shown in an HTML web page with the hyper references associated with the sentences that link to the paragraph in the original document. The advising tool contains an interface for inputting queries. Besides directly inputting queries, users may also upload a performance report of a program execution as the query. Egeria currently supports GPU performance reports (a PDF file output from NVIDIA NVPP),[3] from which, the advising tools by Egeria can find the described key performance issues through simple regular expression based search according to the report format.

5 Evaluations

We conduct a set of experiments to examine the efficacy of Egeria. Our experiments are designed to answer the following four major questions: 1) Is Egeria useful for programmers in easing their efforts in optimizing programs? 2) Do we really need the recognition of advising sentences for easing the use of programming guides? How much does it help compared to simple keyword search or other methods? 3) How does the similarity threshold in knowledge recommendation stage affect the performance? 4) Do we really need the sophisticated NLP-based design to recognize advising sentences? Howe much does it help compared to other designs?

Due to space limit, please refer to our conference paper [6] for the comparisons between our multi-layered design and alternative methods (Question 4). Here, we just briefly mention the key observations. Experiments on CUDA [18], OpenCL [19], and Xeon Phi [20] programming guides show that Egeria can recognize the advising sentences from these

3. https://developer.nvidia.com/nvidia-visual-profiler

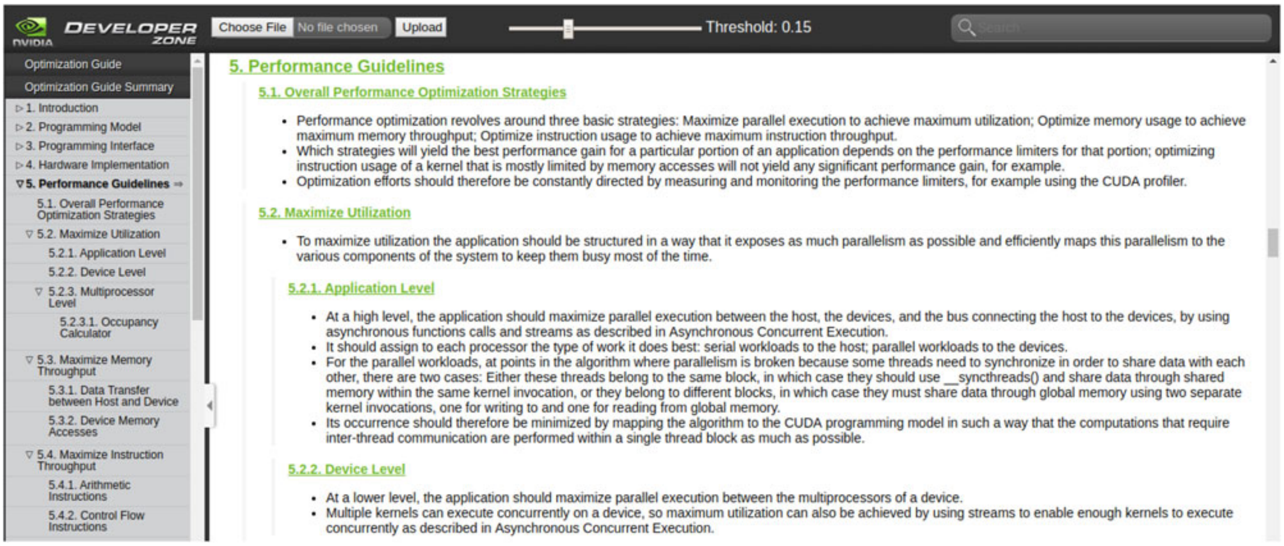


Fig. 4. The initial webpage of the CUDA Adviser, displaying the advising sentences of CUDA Programming Guide. The two buttons on top allow users to upload a performance report in PDF as a query. The search box at the right top corner allows users to directly input queries. The range bar in the middle allow users to adjust the number of retrieved sentences.

guides with over 80 percent precision rates, significantly higher than other alternative methods.

We next report our experiments and results on the first three questions. We start with a user study, showing how useful Egeria is to help programmers address some performance issues of CUDA programs. We then provide some detailed examinations of the benefits of the two-staged design of Egeria in Section 5.2 and a sensitivity study of the similarity threshold in Section 5.3.

5.1 User Study

The user study focuses on an advising tool generated by Egeria to show how one can use NVIDIA profiler data or questions to retrieve relevant and helpful tuning advice. We got the advising tool by applying Egeria on the NIVIDA CUDA Programming Guide [18], which was created to guide the development or optimizations of code to run on NVIDIA GPUs. We call the tool *CUDA Adviser*. The interface of the tool is shown in Fig. 4.

Given a query, either an Nvidia Visual Profiler (NVVP) report or a natural language-based query, our *CUDA Adviser* responds with recommended sentences. (Users can optionally ask it to also list all other advising sentences in the subsections containing those recommended sentences. In that case, the recommended ones will be highlighted) We do not limit the number of sentences the tool can suggest. An advising sentence is suggested as long as it is sufficiently relevant (the similarity threshold is 0.15 as stated in Section 4). In our experiments, the number of suggested sentences for a query is typically 5–25. In the extreme case that no good answers exist, the advising tool gives “No relevant sentences found”.

In the user study, 37 graduate students were asked to manually optimize a sparse matrix manipulation program written using CUDA. The program contains a kernel that makes some normalization to values in a matrix. The original program has optimization potential in multiple aspects, including memory accesses, thread divergences, loop controls, and cache performance. All students were given the original CUDA programming guide and were allowed to use any other resources and tools (including NVIDIA GPU profiling tools) in the process, while Egeria were provided to 22 randomly chosen students out of the 37. There are two ways that students could use *CUDA Adviser*. One is to feed it with an NVVP report, the other is to directly query it with questions. We gave no restrictions on how the students can use the tool. They typically started with the first approach and then used the second approach when they had other questions. As a course project, the students were asked to submit the optimized code and report in two weeks.

An NVVP report usually has four sections. The first section provides an overview of the performance issues while the later three sections each describe the problems in each of the three main aspects: instruction and memory latency; compute resources; memory bandwidth. Some of the later three sections could be empty if no issues exist in those aspects. Each performance issue in a section contains three parts ‘title’, ‘description’, and optional ‘optimization’. We use all the three parts to compose our queries.

When fed with an NVVP report, our *CUDA Adviser* searches within each section and take subsections that contain the “Optimization:” identifier as performance issue-related contents. It then extracts those subsections as performance issue-related contents. Table 3 shows the extracted performance issues for the sparse matrix program used in this case study. Each title and its description are combined to form a query to our *CUDA Adviser*.

Fig. 5 shows the sentences suggested by our *CUDA Adviser* given the example NVVP report. For space limitations, it shows only the sentences selected from Chapter 5 of the CUDA Guide (eight other sentences were chosen in the other 14 chapters). Besides the recommended sentences, the figure also shows some of the other advising sentences residing in the same subsections as the suggested sentences do. The recommended ones are highlighted in the figure.

Among the eight recommended sentences, we can see that the following sentence directly provides suggestions on handling the “register usage” issue:

TABLE 3
Subsections From an Example NVVP Report for Indicating Performance Issues (With the Descriptions Abridged)

Subsection	Description
GPU Utilization May Be Limited By Register Usage	Theoretical occupancy is less than 100% but is large enough that increasing occupancy may not improve performance.... The kernel uses 31 registers for each thread (7936 registers for each block)...
Divergent Branches	Compute resource are used most efficiently when all threads in a warp have the same branching behavior. When this does not occur the branch is said to be divergent. Divergent branches lower warp execution efficiency which leads to inefficient use of the GPU's compute resources....

5. Performance Guidelines

5.1. Overall Performance Optimization Strategies

- Performance optimization revolves around three basic strategies: Maximize parallel execution to achieve maximum utilization; Optimize memory usage to achieve maximum memory throughput; Optimize instruction usage to achieve maximum instruction throughput.
- Which strategies will yield the best performance gain for a particular portion of an application depends on the performance limiters for that portion; optimizing instruction usage of a kernel that is mostly limited by memory accesses will not yield any significant performance gain, for example.
- Optimization efforts should therefore be constantly directed by measuring and monitoring the performance limiters, for example using the CUDA profiler.

5.2. Maximize Utilization

5.2.3. Multiprocessor Level

- At an even lower level, the application should maximize parallel execution between the various functional units within a multiprocessor.

......
- Register usage can be controlled using the maxrregcount compiler option or launch bounds as described in Launch Bounds.

......
- Applications can also parameterize execution configurations based on register file size and shared memory size, which depends on the compute capability of the device, as well as on the number of multiprocessors and memory bandwidth of the device, all of which can be queried using the runtime (see reference manual).
- The number of threads per block should be chosen as a multiple of the warp size to avoid wasting computing resources with under-populated warps as much as possible.

5.4. Maximize Instruction Throughput

- To maximize instruction throughput the application should: Minimize the use of arithmetic instructions with low throughput; this includes trading precision for speed when it does not affect the end result, such as using intrinsic instead of regular functions (intrinsic functions are listed in Intrinsic Functions), single-precision instead of double-precision, or flushing denormalized numbers to zero; Minimize divergent warps caused by control flow instructions as detailed in Control Flow Instructions Reduce the number of instructions, for example, by optimizing out synchronization points whenever possible as described in Synchronization Instruction or by using restricted pointers as described in __restrict__.

5.4.1. Arithmetic Instructions

- cuobjdump can be used to inspect a particular implementation in a cubin object.

......
- As the slow path requires more registers than the fast path, an attempt has been made to reduce register pressure in the slow path by storing some intermediate variables in local memory, which may affect performance because of local memory high latency and bandwidth (see Device Memory Accesses).

......
- This last case can be avoided by using single-precision floating-point constants, defined with an f suffix such as 3.141592653589793f, 1.0f, 0.5f.

5.4.2. Control Flow Instructions

- To obtain best performance in cases where the control flow depends on the thread ID, the controlling condition should be written so as to minimize the number of divergent warps.

......
- The programmer can also control loop unrolling using the #pragma unroll directive (see #pragma unroll).

Fig. 5. Retrieved Sentences from Chapter 5 of CUDA Guide for a Given NVVP Report. (Highlighted are recommended sentences; others, including omitted ones, are advising sentences in the same subsections as the recommended ones are.)

> *Register usage can be controlled using the maxrregcount compiler option or launch bounds as described in Launch Bounds.*

The following sentence is closely related to the "divergent branches" issue:

> *To obtain best performance in cases where the control flow depends on the thread ID, the controlling condition should be written so as to minimize the number of divergent warps.*

With the response, if users want to learn more details, they can easily access the corresponding subsections in the original document through hyper-links associated with each section/subsection title in the summary (these titles are underlined in Fig. 5). For example, by examining Section *5.4.2. Control Flow Instruction*, which contains the aforementioned recommended sentence on "divergent branches", users can find the following sentences that explain warp divergence:

> *Any flow control instruction (...) can significantly impact the effective instruction throughput by causing threads of the same warp to diverge (...). If this happens, the different executions paths have to be serialized, increasing the total number of instructions executed for this warp...*

The reports we received from the students in the user study indicated that the retrieved advising sentence along with its context from the original document helped them identify an optimization opportunity on the if-else block shown in Fig. 6a. The optimized version of the block is shown in Fig. 6b which has the if-else branches removed.

In addition to NVVP reports, students also posted queries to the advising tool. Some example queries were "warp execution efficiency", "How to avoid thread divergence", "memory access coalescence", and so on.

According to students' report and optimized code, optimizations by the Egeria group included memory optimizations

```
if(tx % 2 == 0 && ty % 2 == 0)
    out[tx * width + ty] = 2.0 * in[tx * width + ty]/sum;
else if(tx % 2 == 1 && ty % 2 == 0)
    out[tx * width + ty] = in[tx * width + ty]/sum;
else if(tx % 2 == 1 && ty % 2 == 1)
    out[tx * width + ty] = (-1.0) * in[tx * width + ty]/sum;
else
    out[tx * width + ty] = 0.0f;
```

(a) The If-else Block from the Original Program.

```
out[tx * width + ty] = (((tx+1)%2)+ 1 - (ty%2)*2) * in[tx * width + ty]/sum;
```

(b) The Optimized Block.

Fig. 6. Optimization to minimize thread divergence.

TABLE 4
Speedups on a GPU Program

	GeForce GTX 780		GeForce GTX 480	
	Average	Median	Average	Median
Group 1: Egeria used	**6.27**X	**5.93**X	**4.15**X	**4.43**X
Group 2: Egeria not used	4.09X	3.58X	2.59X	2.39X

(e.g., "rearrange memory access instructions"), minimize thread divergences (e.g., "remove if-else"), increase the amount of parallelism (e.g., "tuning the dimensions of thread blocks and grids"), and minimize the number of instructions a thread needs to do (e.g., "loop unrolling"). The non-Egeria group as a whole covered most of these optimizations, but an individual in that group typically implemented fewer optimizations than an individual in the Egeria group did, as with Egeria, it is easier to identify a comprehensive set of relevant optimizations. We did not see a significant difference in the amount of prior GPU experience between the two groups of students. A quantitative examination of responses' accuracy and comparison is in the next subsection.

Table 4 reports the speedups that the students' optimizations have achieved on two GPUs of different models over the original CUDA program. The much larger speedups obtained by the students that have used Egeria suggest the usefulness of the advising tool by Egeria: With its advice, the students were able to better target the set of suitable optimizations in their explorations, which has saved them time in searching in the original documents or other resources and has helped prevent them from trying many irrelevant optimizations.

5.2 Effectiveness of the Two-Level Design

In this part, we report a deeper examination of the effectiveness of the two-level design featured by Egeria, and compare it with some alternative methods.

Recall that the key idea of the two-stage design is to first recognize advising sentences, and then from them, find the sentences related with the input query. We compare it with two one-stage methods:

- *Keywords method:* This method uses keywords in the input query to directly search the original programming guide to find relevant sentences. Both the keywords and the words in the document are reduced to their stem forms to allow matchings among different variants of a word.
- *Full-doc method:* This method also queries the original programming guide without first extracting advising sentences. Unlike the *keywords method*, this method does not use keywords, but uses the same knowledge recommendation method as Egeria uses—that is, through the use of VSM and TF-IDF techniques as Section 4 describes.

We applied the several methods to four GPU performance profiling reports. These reports were collected through an NVIDIA GPU profiling tool (NVPP)[4], with each containing a detailed description of the performance issues of a GPU program execution. The four reports are for the following four CUDA programs:

- knnjoin.cu: a K-Nearest Neighbor (KNN) program that has thread divergence problems in the kernel;
- knnjoin-opt.cu: knnjoin.cu after some task reordering to reduce the thread divergence for the kernel;
- trans.cu: a matrix transpose that has a large number of non-coalesced memory accesses;
- trans-opt.cu: trans.cu after optimizing the memory accesses through the use of 2D surface memory.

The second column in Table 5 lists the top issue(s) of the most time-consuming kernel of each of the four programs.

We fed the reports into our CUDA advising tool and the *full-doc method*; they each returned a set of sentences for each of the reports as their answers on how to resolve the performance issues in that report. For the *keywords method*, we tried a number of keywords for each performance issue as listed below:

- knnjoin (issue 1): warp, execution, efficiency, warp efficiency, warp execution efficiency;
- knnjoin (issue 2): divergence, branch, divergent branch;
- knnjoin_opt: memory, alignment, memory alignment, access pattern;
- trans (issue 1): utilization, memory, instruction, memory instruction;
- trans (issue 2): instruction, latency, instruction latency;
- trans_opt: memory, bandwidth, memory bandwidth;

TABLE 5
Quality of Answers on Performance Queries

NVVP Report	Performance Issues	#gnd truth	Egeria Method			Full-doc Method			Keywords Method		
			P	R	F	P	R	F	P	R	F
knnjoin	P1	6	0.667	1.0	0.8	0.146	1.0	0.255	0.154	1.0	0.267
	P2	2	0.667	1.0	0.8	0.167	1.0	0.286	0.333	1.0	0.5
knnjoin_opt	P3	7	1.0	0.857	0.923	0.304	1.0	0.467	0.571	0.571	0.571
trans	P4	8	0.667	1.0	0.8	0.211	1.0	0.348	0.571	0.5	0.533
	P5	11	0.667	0.909	0.769	0.182	0.909	0.303	0.364	0.364	0.364
trans_opt	P6	18	0.652	0.833	0.732	0.308	0.889	0.457	0.545	0.333	0.414

P1: Low Warp Execution Efficiency; P2: Divergent Branches; P3: Global Memory Alignment and Access Pattern; P4: GPU Utilization is Limited by Memory Instruction Execution; P5: Instruction Latencies may be Limiting Performance; P6: GPU Utilization is Limited by Memory bandwidth.
(P: precision; R: recall; F: F-measure).

The underlined are the keywords that yield the best overall results in terms of F-measure (defined in the next paragraph).

Table 5 reports the quality of the results by the three methods. For *keywords method*, the table shows only the results by the aforementioned best keywords. The three metrics we use are commonly used in information retrieval: precision P (#true positive/#answers), recall R (#true positive/#groundTruth), and the combined metric F-measure $F = 2 * P * R/(P + R)$. We asked three domain experts to manually label all the sentences in the CUDA programming guide regarding whether they are advising sentences relevant for resolving each of the performance issues listed in Table 5. The Fleiss' kappa values [21] (a standard measure for assessing the reliability of agreement of a number of raters) of the labeling results are all above 0.8, indicating large agreements among the raters. Majority vote is used to generate the ground truth answers for each of the performance issues.

As the "Egeria" column in Table 5 shows, our advising tool returns most relevant advising sentences, with the recall rates at 83-100 percent. The small number of missing sentences are mostly due to some difficulties in advising sentence recognitions. A fraction (0-35 percent) of the answers are false positives for some limitations of the VSM/TF-IDF technique used for similarity computations. But overall, the advising tool gives answers significantly better than both alternative methods give.

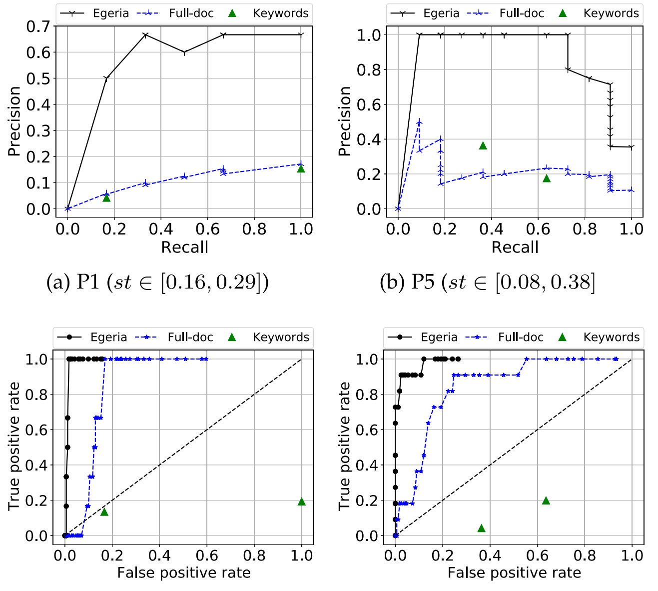
Because the "full-doc" method uses the same knowledge recommendation method as the Egeria-based advising tool uses and advising sentences are part of the original document, this method finds all the sentences returned by the Egeria-based CUDA advising tool. However, it also yields many sentences that are not advising sentences because it works on the original document. Some of these sentences, for instance, are detailed explanations of some terms or concepts, and some are details of some example architectures. Although these may have some relevancy to the input queries, they do not give suggestions on how to optimize the program to resolve the performance issues specified in the queries. As Table 5 shows, the precision of the returned results by the *full-doc method* is only 30 percent or below.

The "keywords" method is inferior in both precision and recall. The reason is that lots of sentences containing the keywords are not advising sentences, but explanations of some details or examples. At the same time, many relevant advising sentences do not contain the keywords. Please refer to [6] for example sentences.

We applied stemming to the keywords and documents to allow matchings between variants of words. Without stemming, the false positives of the "keywords" method could get reduced slightly, but the recall rate would get much lower; the overall results would be even worse.

5.3 Sensitivity Analysis

In knowledge recommendation stage (Section 4), the default similarity threshold is set to 0.15 as shown in Fig. 4. The advising tools only recommend advising sentences with a similarity score higher than the default threshold. To investigate the influence of the similarity threshold, we evaluated



(a) P1 ($st \in [0.16, 0.29]$) (b) P5 ($st \in [0.08, 0.38]$)

(c) P1 (d) P5

Fig. 7. PRCs and ROCs. The "Egeria" and "Full-doc" curves correspond to a spectrum of sampled similarity thresholds (st) as shown in the subgraph captions. P1, P5 refer to performance issues listed in Table 5.

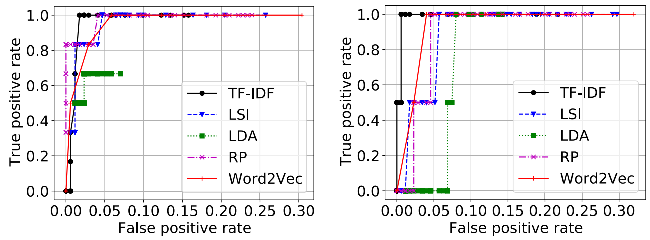
the performance of the three methods (Egeria, Full-doc, and Keywords) under different threshold settings.

We vary the similarity threshold from zero to 0.5 with a step size of 0.01. Fig. 7 shows the Precision and Recall curves (PRCs) and Receiver Operating Characteristic curves (ROCs) for two benchmarks used in Table 5. The PRCs and ROCs for other benchmarks are similar. Since the Keywords method does not use the knowledge recommendation algorithm as Egeria and Full-doc method use, it is not affected by different similarity thresholds. The different triangle points correspond to different keywords in the input query. The smaller the similarity threshold is, the larger number of recommended sentences and higher false-positive rate and true positive rates we see.

According to the PRCs in Fig. 7, with the same recall, the Egeria method gives the highest precision consistently. Also, it is worth mention that our default similarity threshold (i.e., 0.15) achieves a good balance between recall and precision: it yields, in most cases, a recall rate of 100 percent and also the highest precision. With a smaller threshold, more sentences are recommended at the expense of a decrease in precision since the query results are diluted by advising sentences that may not be solutions to the specific query. For instance, in Fig. 7b, a similarity threshold of 0.8 can give a recall rate of 100 percent but a precision rate of 35.5 percent (31 recommended sentences). This means that a user needs to go through more information to find potential solutions. In practice, with our advising tools, users can adjust the similarity threshold to control the number of recommended sentences to meet their needs.

6 Extensions for Semantic Sensitivity

We adopted the term frequency-inverse document frequency to represent advising sentences. This representation allows sentence ranking according to their possible relevance based on the number of overlapped words and the



(a) P1 (b) P5

Fig. 8. ROC curves for P1 knnjoin(issue 1) and P5 trans (issue 2) listed in Table 5.

importance of those words. The main limitation is that it cannot recognize the relevance between sentences with a similar meaning but in different term vocabularies. It is called the *semantic sensitivity* problem.

Several models (e.g., Latent Semantic Indexing (LSI) [22], Latent Dirichlet Allocation (LDA) [23], Random Projection (RP) [24]) have been proposed to avoid the semantic sensitivity problem by learning representation for a document in a latent semantic space with lower dimensionality. Each latent dimension corresponds to a latent topic. Each document is represented in terms of latent topics rather than words. These models rely on different techniques to determine the relationship between words and latent topics. LSI, also called Latent Semantic Analysis, uses a mathematical technique called Singular Value Decomposition (SVD) for dimension reduction. For real corpora, the recommended number of target dimensions is 200-500 [25]. LDA is a probabilistic extension of LSI, which means that the latent topics of LDA are probability distributions over words and also that each document is a soft mixture of topics. RP is a more computationally efficient, yet sufficiently accurate method for dimension reduction, compared with LSI. In RP, the original high dimensional data is projected onto a lower-dimensional subspace using a random matrix.

Recent proposed word embeddings (e.g., Word2Vec [26], [27], [28] and GloVe [29] learn a low-dimensional vector representation, called *embedding*, for each word. These embeddings capture the semantic relationships among words. For example, *vec*(Berlin) - *vec*(Germany) + *vec*(France) is close to *vec* (Paris), where $vec(.)$ is the embedding function. Based on the embeddings, one can calculate the distance between two documents by Word Mover's Distance (WMD) [30].

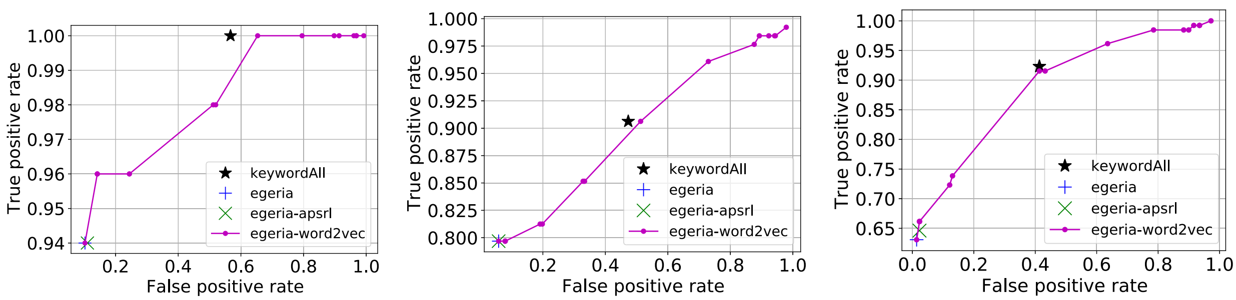
We compared these advanced models and Word2Vec with TF-IDF. For methods LSI, LDA, and RP, we set the latent dimension to 50, 100, and 200. For Word2Vec, we used word embeddings of dimension 100 pre-trained on Wikipedia and Gigaword [29] and finetuned these embeddings on the CUDA programming guide using Gensim [17]. The ROCs for the benchmark knnjoin (issue 1) and trans (issue 2), with different models and a latent dimension 100 are shown in Fig. 8. Other benchmarks and latent dimensions have similar observations. Given the same false-positive rate, these advanced models yield similar or even worse true positive rate compared with TF-IDF. This may result from the limited size of the training corpus (i.e., sentences from CUDA Programming Guide). Further explorations with larger training data sizes can be more meaningful.

7 Extensions for Advising Sentence Recognition

Advising sentence recognition takes advantage of HPC domain-specific properties, including advising sentence patterns and corresponding keywords, to simplify the problem into five simpler ones. This results in five selectors working as an ensemble to identify advising sentences with high accuracy. Although this multi-layered design has shown much better results over the alternatives in our conference paper [6], the five selectors rely on *exact matching* with the sets of keywords listed in Table 2. The first open question is whether we can further improve the classification accuracy if Egeria can identify advising sentences that contain semantic-equivalent or semantic-similar words. Also, the fifth selector (Rule #5) uses semantic role labeling (SRL) which is generally a more complex task than dependency parsing and thus more error-prone. The second open question is whether we can replace semantic role labeling with the more accurate dependency parsing technique by considering specific dependency patterns. This section reports our explorations to answer the two open questions.

Keyword Expansion. We leveraged pre-trained word2-vec [27] to expand the sets of keywords in Table 2. We add a word w from the programming guide into a set of keywords S, where $S \in$ {FLAGGING_WORDS, XCOMP_GOVERNORS, IMPERATIVE_WORDS, KEY_SUBJECTS, KEY_PREDICATES} if the cosine similarity between w and any word in the set S is larger than a threshold. We vary the threshold from 0.8 to 0.95. The five selectors then use the expanded sets of keywords to classify advising sentences.

We call the method *egeria-word2vec* and show its classification performance in Fig. 9 as a ROC curve. We compare *egeria-word2vec* with two other methods *egeria* and *keywordAll*. For



(a) CUDA (b) OpenCL (c) Xeon

Fig. 9. ROCs for advising sentence recognition on three programming guides: CUDA [18], OpenCL [19], and Xeon [20].

the *keywordAll* approach, we used the same experiment setting: we apply the first selector (the keyword-based selector) but use the union of all the keywords used in all selectors as the replacement of the FLAGGING WORDS.

According to Fig. 9, *egeria-word2vec* with a high similarity threshold can achieve the same accuracy as *egeria* in recognizing advising sentences. When we lower the similarity threshold to include more semantic-similar keywords, it is worse than the *keywordAll* approach under the same false-positive rate. This means incorporating semantic-similar words into the sets of keywords lower the precision of the advising sentence recognition.

Selector Approximation. We replaced the fifth selector (Rule #5) introduced in Section 3.2.3 using the following simpler dependency parsing-based rule:

Rule 6. *A sentence is a HPC advising sentence if it meets all the following conditions:*

1) *The sentence contains a verb* v *and* $lemma(v) \in$ *KEY_PREDICATES.*
2) v *is not in any* $subj$ *dependency relation.*

We use *egeria-apsrl* to refer to the advising sentence recognition method using the five selectors, Rule #1-#4 and Rule #6. Its classification performance is shown in Fig. 9. *egeria-apsrl* is able to achieve similar precision and recall compared with *egeria*.

8 Related Work

The importance of tools for HPC has been well recognized. Through the years, many high quality HPC tools have been developed. HPCToolkit [1] provides a set of tools for profiling and analyzing HPC program executions. Other tools for performance profiling include some code-centric tools (e.g., VTune [31], Oprofile [32], CodeAnalyst [33], and Gprof [34]) and some other data-centric tools [35], [36], [37], [38]. Just for GPU, there are numerous performance profiling tools (e.g., NVVP [2], NVProf [2], CodeXL [39], GPU PerfStudio [40], Snapdragon [41]. There have also been many profiling tools developed for data centers and cloud (e.g., PerfCompass [42]). All these tools have concentrated on measuring and identifying the main performance issues, rather than creating advising tools for offering advice on how to fix the issues.

NLP has been used in software engineering broadly. For instance, it has been used for some bug report classification [43], bug report summarization [44], bug severity prediction [45], and relevant source files retrieval [46]. The goals of those work differ from the recognition of advising sentences. For instance, report summarization aims at creating a representative summary or abstract of a report [47]. It focuses on finding the most informative sentences, which may not be advising sentences. The different goals of Egeria motivate its unique design and distinctive ways to leverage NLP techniques.

9 Conclusion

We developed the framework Egeria for automatic synthesis of HPC advising tools. Advising tools generated by Egeria can provide users with a list of important optimization guidelines to remind them of available optimization rules, and suggest related optimization advice based on the performance issues of a program or questions from a user. Egeria is made possible by integrating HPC domain properties with NLP techniques for recognizing advising sentences with a high accuracy. Both qualitative and quantitative experiments demonstrate the usefulness of Egeria for HPC.

Acknowledgments

John Mellor-Crummey gave us some valuable suggestions on this work. The authors would like to thank Lars Nyland and Huiyang Zhou for their comments at the early stage of this work. This material is based upon work supported by DOE Early Career Award (DE-SC0013700), and the National Science Foundation (NSF) Grants No. 1455404, 1455733 (CAREER), and 1525609. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DOE or NSF.

References

[1] L. Adhianto *et al.*, "HPCTOOLKIT: Tools for performance analysis of optimized parallel programs," *Concurrency Comput., Pract. Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[2] NVIDIA. Profiler User's Guide. PDF file. 2017. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf

[3] S. Cook, *CUDA Programming: A Developer's Guide to Parallel Computing With GPUs*. Burlington, MA, USA: Morgan Kaufmann, 2012.

[4] A. Munshi, B. Gaster, T. G. Mattson, and D. Ginsburg, *OpenCL Programming Guide*. London, U.K.: Pearson Education, 2011.

[5] P. D. Turney *et al.*, "From frequency to meaning: Vector space models of semantics," *J. Artif. Intell. Res.*, vol. 37, no. 1, pp. 141–188, 2010.

[6] H. Guan, X. Shen, and H. Krim, "Egeria: A framework for automatic synthesis of HPC advising tools through multi-layered natural language processing," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2017, Art. no. 10.

[7] S. Kübler, R. McDonald, and J. Nivre, "Dependency parsing," *Synthesis Lectures Hum. Lang. Technol.*, vol. 1, no. 1, pp. 1–127, 2009.

[8] C. Niklaus, M. Cetto, A. Freitas, and S. Handschuh, "A survey on open information extraction," in *Proc. 27th Int. Conf. Comput. Linguistics*, Aug. 2018, pp. 3866–3878.

[9] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky, "The stanford CoreNLP natural language processing toolkit," in *Proc. 52nd Annu. Meeting Assoc. Comput. Linguistics: Syst. Demonstrations*, 2014, pp. 55–60.

[10] M.-C. De Marneffe and C. D. Manning, "Stanford typed dependencies manual," Stanford University, Stanford, CA, 2008. [Online]. Available: https://nlp.stanford.edu/software/dependencies_manual.pdf

[11] V. Punyakanok, D. Roth, and W. Yih, "The importance of syntactic parsing and inference in semantic role labeling," *Comput. Linguistics*, vol. 34, no. 2, pp. 257–287, 2008. [Online]. Available: http://cogcomp.cs.illinois.edu/papers/PunyakanokRoYi07.pdf

[12] M. Palmer, D. Gildea, and P. Kingsbury, "The proposition bank: An annotated corpus of semantic roles," *Comput. Linguistics*, vol. 31, no. 1, pp. 71–106, 2005.

[13] X. Carreras and L. Màrquez, "Introduction to the CoNLL-2005 shared task: Semantic role labeling," in *Proc. 9th Conf. Comput. Natural Lang. Learn.*, 2005, pp. 152–164.

[14] P. Qi, Y. Zhang, Y. Zhang, J. Bolton, and C. D. Manning, "Stanza: A Python natural language processing toolkit for many human languages," in *Proc. 58th Annu. Meeting Assoc. Comput. Linguistics: Syst. Demonstrations*, 2020, pp. 101–108.

[15] M. Gardner *et al.*, "AllenNLP: A deep semantic natural language processing platform," in *Proc. Workshop NLP Open Source Softw.*, 2017, pp. 1–6.

[16] S. Bird, "NLTK: The natural language toolkit," in *Proc. COLING/ACL Interactive Presentation Sessions*, 2006, pp. 69–72.

[17] R. Řehůřek and P. Sojka, "Software framework for topic modelling with large corpora," in *Proc. LREC Workshop New Challenges NLP Frameworks*, May 2010, pp. 45–50, [Online]. Available: http://is.muni.cz/publication/884893/en

[18] NIVIDA CUDA programming guide, 2016, Accessed: Jul. 19, 2017. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[19] AMD OpenCL optimization guideline, 2013, Accessed: Jul. 19, 2017. [Online]. Available: http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-par allel-processing-app-sdk/opencl-optimization-guide/

[20] Intel Xeon Phi best practice guide, 2017, Accessed: Jul. 19, 2017. [Online]. Available: http://www.prace-ri.eu/best-practice-guide-intel-xeon-phi-html/

[21] J. L. Fleiss, "Measuring nominal scale agreement among many raters," *Psychol. Bull.*, vol. 76, no. 5, 1971, Art. no. 378.

[22] C. H. Papadimitriou, H. Tamaki, P. Raghavan, and S. Vempala, "Latent semantic indexing: A probabilistic analysis," in *Proc. 17th ACM SIGACT-SIGMOD-SIGART Symp. Princ. Database Syst.*, 1998, pp. 159–168. [Online]. Available: http://doi.acm.org/10.1145/275487.275505

[23] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, Mar. 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=944919.944937

[24] E. Bingham and H. Mannila, "Random projection in dimensionality reduction: Applications to image and text data," in *Proc. 7th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2001, pp. 245–250.

[25] R. B. Bradford, "An empirical study of required dimensionality for large-scale latent semantic indexing applications," in *Proc. 17th ACM Conf. Inf. Knowl. Manage.*, 2008, pp. 153–162.

[26] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *Proc. Int. Conf. Learn. Representations*, 2013.

[27] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. 26th Int. Conf. Neural Inf. Process. Syst.*, 2013, pp. 3111–3119.

[28] T. Mikolov, W.-T. Yih, and G. Zweig, "Linguistic regularities in continuous space word representations," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics: Hum. Lang. Technol.*, 2013, pp. 746–751.

[29] J. Pennington, R. Socher, and C. D. Manning, "GloVe: Global vectors for word representation," in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2014, pp. 1532–1543.

[30] M. Kusner, Y. Sun, N. Kolkin, and K. Weinberger, "From word embeddings to document distances," in *Proc. 32nd Int. Conf. Mach. Learn.*, 2015, pp. 957–966.

[31] J. Reinders, *VTune Performance Analyzer Essentials*. Santa Clara, CA, USA: Intel Press, 2005.

[32] J. Levon and P. Elie, "OProfile: A system profiler for linux," 2004, Accessed: Aug. 28, 2020. [Online]. Available: https://oprofile.sourceforge.io

[33] Amd. An introduction to analysis and optimization with AMD CodeAnalyst Performance Analyzer. PDF file. 2008. [Online]. Available: https://developer.amd.com/wordpress/media/2012/10/Introduction_to_CodeAnalyst.pdf

[34] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," *ACM SIGPLAN Notices*, vol. 17, no. 6, pp. 120–126, 1982.

[35] B. R. Buck and J. K. Hollingsworth, "Data centric cache measurement on the Intel ltanium 2 processor," in *Proc. ACM/IEEE Conf. Supercomputing*, 2004, p. 58.

[36] R. Lachaize, B. Lepers, and V. Quéma, "MemProf: A memory profiler for NUMA multicore systems," in *Proc. USENIX Annu. Tech. Conf.*, 2012, Art. no. 5.

[37] X. Liu and J. Mellor-Crummey, "Pinpointing data locality problems using data-centric analysis," in *Proc. 9th Annu. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2011, pp. 171–180.

[38] C. McCurdy and J. Vetter, "Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2010, pp. 87–96.

[39] Codexl quick start guide, 2012, Accessed: Dec. 14, 2016. [Online]. Available: http://developer.amd.com/tools-and-sdks/opencl-zone/codexl

[40] GPU PerfStudio, 2016, Accessed: Dec. 14, 2016. [Online]. Available: http://developer.amd.com/tools-and-sdks/graphics-development/gpu-perfstudio

[41] Qualcomm. Snapdragon Profiler Quick Start Guide. 2016. [Online]. Available: https://developer.qualcomm.com/download/sdprofiler/sdprofiler-quick-start-guide.pdf

[42] D. J. Dean, H. Nguyen, P. Wang, X. Gu, A. Sailer, and A. Kochut, "PerfCompass: Online performance anomaly fault localization and inference in infrastructure-as-a-service clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 6, pp. 1742–1755, Jun. 2016.

[43] Y. Zhou, Y. Tong, R. Gu, and H. Gall, "Combining text mining and data mining for bug report classification," *J. Softw.: Evol. Process*, vol. 28, pp. 150–176, 2016.

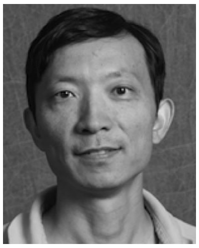
[44] S. Mani, R. Catherine, V. S. Sinha, and A. Dubey, "AUSUM: Approach for unsupervised bug report summarization," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, 2012, Art. no. 11.

[45] Y. Tian, D. Lo, and C. Sun, "Information retrieval based nearest neighbor classification for fine-grained bug severity prediction," in *Proc. 19th Work. Conf. Reverse Eng.*, 2012, pp. 215–224.

[46] X. Ye, R. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 689–699.

[47] D. Das and A. F. Martins, "A survey on automatic text summarization," *Literature Survey Lang. Statist. II Course at CMU*, vol. 4, pp. 192–195, 2007.

Hui Guan (Member, IEEE) is currently an assistant professor at the College of Information and Computer Sciences (CICS), University of Massachusetts Amherst. Her research interests include the intersection between machine learning and programming systems.



Xipeng Shen (Senior Member, IEEE) is a professor with the Computer Science Department, North Carolina State University, Raleigh, North Carolina. He is an ACM distinguished speaker. His current research focuses on heterogeneous massively parallel computing, high performance machine learning, and high-level large-scale program optimizations.

Hamid Krim (Fellow, IEEE) is presently on the faculty in the ECE Department, North Carolina State University, Raleigh, North Carolina, leading the Vision, Information and Statistical Signal Theories and Applications group, whose research interests are in statistical signal and image analysis and mathematical modeling with a keen emphasis on applied problems.