Egeria: A Framework for Automatic Synthesis of HPC Advising Tools through Multi-Layered Natural Language Processing

Hui Guan
North Carolina State University
Raleigh, NC 27695
hguan2@ncsu.edu

Xipeng Shen
North Carolina State University
Raleigh, NC 27695
xshen5@ncsu.edu

Hamid Krim
North Carolina State University
Raleigh, NC 27695
ahk@ncsu.edu

ABSTRACT

Achieving high performance on modern systems is challenging. Even with a detailed profile from a performance tool, writing or refactoring a program to remove its performance issues is still a daunting task for application programmers: it demands lots of program optimization expertise that is often system specific.

Vendors often provide some detailed optimization guides to assist programmers in the process. However, these guides are frequently hundreds of pages long, making it difficult for application programmers to master and memorize all the rules and guidelines and properly apply them to a specific problem instance.

In this work, we develop a framework named Egeria to alleviate the difficulty. Through Egeria, one can easily construct an advising tool for a certain high performance computing (HPC) domain (e.g., GPU programming) by providing Egeria with a optimization guide or other related documents for the target domain. An advising tool produced by Egeria provides a concise list of essential rules automatically extracted from the documents. At the same time, the advising tool serves as a question-answer agent that can interactively offers suggestions for specific optimization questions. Egeria is made possible through a distinctive multi-layered design that leverages natural language processing techniques and extends them with knowledge of HPC domains and how to extract information relevant to code optimization Experiments on CUDA, OpenCL, and Xeon Phi programming guides demonstrate, both qualitatively and quantitatively, the usefulness of Egeria for HPC.

CCS CONCEPTS

•General and reference →Performance; •Computing methodologies →Natural language processing;

KEYWORDS

program optimization, high performance computing, natural language processing

ACM Reference format:
Hui Guan, Xipeng Shen, and Hamid Krim. 2017. Egeria: A Framework for Automatic Synthesis of HPC Advising Tools through Multi-Layered Natural Language Processing. In *Proceedings of SC17, Denver, CO, USA, November 12–17, 2017,* 14 pages.
DOI: 10.1145/3126908.3126961

SC17, Denver, CO, USA

DOI: 10.1145/3126908.3126961

1 INTRODUCTION

Achieving high performance on computing systems is a complicated process. It requires a deep understanding of the underlying computing systems, the architectural properties, and proper implementations to take a full advantage of the computing systems.

Performance profiling tools (e.g., HPCToolkit [1], NVProf [24]) have been developed to help. Although these tools help identify performance issues, how to optimize the code to address the issues still demands lots of expertise specific to the underlying architecture and may require significant code refactoring.

The problem is exacerbated by the rapid changes and increasing complexity of modern systems (e.g., many-core heterogeneous systems equipped with Graphic Processing Units), causing a fast continuous growth of the set of knowledge and specifications programmers have to master in order to effectively harness the computing systems.

Vendors typically provide some documents to assist users in programming new architectures. For example, both NVIDIA and AMD have published programming guides [6, 23] 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.

Such documents, however, are often hundreds of pages long. Programmers could read them in their entirety and try to apply what they've learned; however, 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. As an NVIDIA architect observes, only a small subset of GPU application developers have showed a good familiarity with the NVIDIA GPU programming guide. Consequently, the potential value of the detailed programming guides remains largely untapped.

To address the problem, we propose a framework named Egeria[1] to bridge the gap between programmers' demands for optimization guidelines and the hard-to-master programming guides. Through Egeria, one can easily construct an advising tool for a certain HPC domain (e.g., GPU programming) by providing Egeria with a programming guide or other documents of that type. The advising tool synthesized by Egeria provides a concise list of essential rules, automatically extracted from the documents. At the same time, the advising tool can serve as a question-answer (QA) agent to interactively offer suggestions for specific optimization questions. 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

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

issues. The tool will immediately provide a list of guidelines for solving those performance problems.

QA systems exist in other domains. Egeria differs from such systems in two key aspects. First, Egeria itself is not a QA system but *a generator* of QA 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. Second, traditional construction of a QA system usually requires lots of manual work. Domain experts need to manually develop the ontology (i.e., conceptual framework and terminology) of the domain, and collect a large amount of labeled documents for systematic training. For example, in the IBM Watson system [11, 12], candidate answers are ranked based on the contribution from hundreds of evidence dimensions. The formation of these dimensions requires a manually defined taxonomy of evidence types to group features. Weighting and combining these features for a final ranking score also need a well-trained machine-learning model. Egeria, on the other hand, produces the advising tools with virtually no manual inputs needed, making it easy to adopt in HPC. To our best knowledge, Egeria is the first auto-constructor of advising tools for HPC or other domains.

The key reason for making Egeria a success is a distinctive multi-layered design that closely leverages the properties of HPC domains to overcome the weaknesses of existing Natural Language Processing (NLP) techniques. NLP has achieved some impressive advancements in recent years. However, some complex NLP analyses needed for general advising tool constructions are yet to get mature. The state-of-the-art semantic role labeling, for instance, gives only 76.6% accuracy for general test sets [30]. The yet-to-improve quality of NLP techniques is the reason for the heavy reliance on manual efforts in the construction of existing advising tools.

Two key features of Egeria help it circumvent those difficulties. (1) It leverages some important features of HPC domains, including the common syntactic and semantic patterns in the advising sentences in programming guides for HPC, and the importance of some special words and phrases related with performance improvements in HPC. Its ability to exploit such features helps significantly simplify the problems. (2) Egeria adopts a multi-layered design, which 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 [34] and TF-IDF [34]), Egeria accurately finds the relevant advising sentences for users' queries.

We evaluate Egeria through several experiments, in which, Egeria produces an advising tool 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% precision recall rates, significantly higher than other alternative methods. Its two-stage design makes it able to answer CUDA program optimization queries with a 80-100% accuracy, substantially higher than a single-stage design. A user study on 37 students show that by using the advising tools produced by Egeria, students can more effectively optimize GPU programs, yielding code with a much higher performance.

Overall, this work makes the following major contributions:

- *Egeria.* It introduces the first auto-constructor of HPC advising tools.
- *Approach.* It presents an effective approach, which circumvents the limitations of current NLP techniques for constructing advising tools through a multi-layered design that incorporates HPC special properties into each NLP layer.
- *Evaluation.* It describes a systematic evaluation of the novel framework and approach, and demonstrates their promise for bridging the gap between programmers' demands for HPC optimization advises and the hard-to-master rapid changing programming guide documentations.

2 PROBLEM OVERVIEW AND POTENTIAL SOLUTIONS

The problem that an HPC advising tool faces is to identify the sentences in a given document that can serve as suggested solutions for an input query on improving certain performance aspects of a program (e.g., "how to improve memory throughput"). We call those sentences "relevant advising sentences". The problem could be regarded as a binary classification problem: to determine whether each of the sentences in the given document belongs to the category of "relevant advising sentences" for the given query. This section discusses the potential solutions and their issues, and then gives an overview of our solution.

Potential Solutions. A commonly used method for classification is supervised machine learning. By statistically learning upon a set of training data, the method builds up a predictive model. For an input query, the model predicts which sentence in the document is an advising sentence relevant to that query. This method requires a large volume of labeled data, which, in our case, would be many queries and at least many thousands of sentences labeled as "relevant advising sentences" or "other sentences" for each of many queries. Given the scarcity of labeled data in HPC advising and the large amount of manual labeling efforts this method requires, this method is not a practical option for our problem.

We hence focus on unsupervised methods. In *text retrieval*, there is a common unsupervised method for retrieving sentences relevant to a query. It represents both the query and each sentence as a feature vector, and then computes the relevance between the query and each sentence by calculating their vector distances. This method considers relevance only. When applying this method to our problem, it finds many sentences that are relevant to the query, but are not advising sentences (as Section 4 reports). These sentences may explain some architectural details or examples rather than provide potential solutions for the performance problem expressed in the query.

Design of Egeria. Egeria employs a distinctive unsupervised design. To address the shortcomings of the relevance-only method in text retrieval, it uses a two-stage design. As the top row in Figure 1 shows, the two stages consider the "advising" and "relevance" aspects respectively. The two boxes at the bottom part of Figure 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

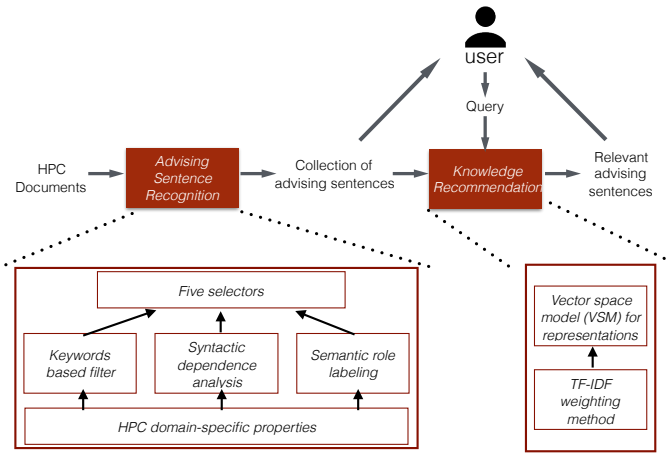


Figure 1: Overview of Egeria. The two boxes at the bottom give more detailed illustrations of the two stages of Egeria respectively.

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 Figure 1 shows. It builds its second stage upon two key text retrieval techniques, namely the VSM representations and the TF-IDF weighting method. We provide a detailed explanation next.

3 THE INTERNAL OF EGERIA

This section describes the design of Egeria and how it addresses the difficulties in each of the two stages.

3.1 Stage I: Advising Sentence Recognition

It is worth noting the differences between advising sentence recognition and a common text retrieval task namely document summarization. Document summarization aims at creating a representative summary or abstract of one or more documents [7]. It focuses on finding the most informative sentences, which may not be advising sentences. We are not aware of any prior studies specifically on recognizing advising sentences.

Recognitions of advising sentences require 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. As aforementioned, for instance, the state-of-the-art semantic role labeling (which labels what semantic role each phrase associated with verbs plays in a sentence) gives only 76.6% accuracy for general test sets [30].

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.

3.1.1 HPC Domain-Specific Properties. By examining some HPC documents, we observe that advising sentences of HPC are often featured with certain patterns in its sentential form coupled 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).

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.

3.1.2 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".

(1) First Selector

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 principle 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.*

(2) Dependency Parsing and Selectors 2,3,4

Table 1: HPC Advising Sentence Categories.

Categories	Patterns	Example Sentences (w/ key words underlined)	Selection Rules (S: a given sentence; Uppercased words: sets of keywords shown in Table 2)	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: xcomp(governor, *), lemma(governor) $\in$ XCOMP_GOVERNORS	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 has no subject 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: subject(S) $\in$ KEY_SUBJECTS	
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: $\exists p$ in the predicate of purpose(S), $p \in$ KEY_PREDICATES	Semantic Role Labeling

Table 2: Sets of Keywords Used in the Selectors.

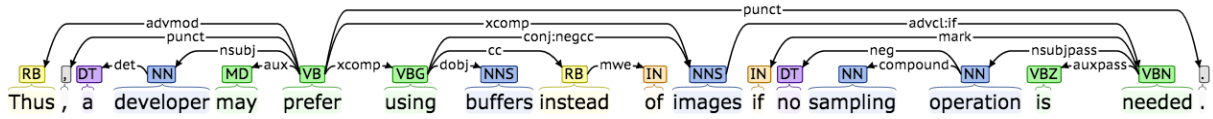
FLAGGING_WORDS	{'better', 'best performance', 'higher performance', 'maximum performance', 'peak performance', 'improve the performance', 'higher impact', 'more appropriate', 'should', 'high bandwidth', 'benefit', 'high throughput', 'prefer','effective way', 'one way to', 'the key to', 'contribute to', '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'}

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 [15]. Dependency parsing has recently attracted considerable interest in the NLP community due to its successful applications on problems such as information extraction and machine translation. 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.

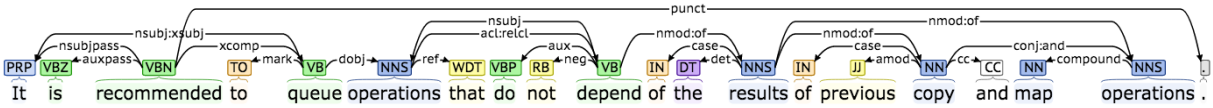
Figure 2 shows the dependency structures for two example sentences generated by the Stanford CoreNLP dependency parser[2]. The dependency relations are represented as arrows pointing from a governor to a dependent. Each arrow is labeled with a dependency type. For example, in Figure 2a, the noun *developer* 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) [8]. 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 Figure 2a, 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). 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 [8]. For example, in Table 1, the given sentences in categories II and III have relations xcomp(prefer, using) and xcomp(leveraged, avoid) respectively (the full dependency structures are shown in Figure 2). The principle rule used by Selector 2 is as follows:

[2]http://corenlp.run/



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



(b) Dependency structure for a sentence in Passive Sentence category. xcomp(recommended, queue).

Figure 2: Dependency Structure.

Rule 2. *A sentence is an advising sentence if it contains the following dependency relation: xcomp(governor, *), where, governor ∈ 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 or governor. 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 [15], 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 principle 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) *root(ROOT, v), lemma(v) ∈ 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 subjects of a sentence through the dependency parsing and then checks whether they belong to the KEY_SUBJECTS set. The principle 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 following dependency relation: nsubj(governor, n), where, lemma(n) ∈ KEY_SUBJECTS.*

(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.

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.

Figure 3 shows an example attained through a Semantic Role Labeling Demo[3] [26]. The demo follows the definition of semantic roles encoded in the lexical resource PropBank [25] and CoNLL-2004 shared task [4]. 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 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 semantic role labeling to detect sentences with purpose clauses. It particularly seeks for the purposes related with HPC optimizations. The predicate of the purpose clause usually offers good hints on the relevance. After finding the purpose clause of the sentence, the selector checks whether the predicate of the clause 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 with the semantic role AM-PNC;*
(2) *arg contains a predicate v;*
(3) *lemma(v) ∈ PREDICATE_SET.*

We implement the selectors based on some NLP tools. We use Stanford CoreNLP [21] for dependency parsing, SENNA [5] for semantic role labeling, and NLTK [2] for word and sentence tokenization, word stemming and lemmatization.

These tools represent the state-of-the-art of NLP development. Some steps of these tools (e.g., stemming) are mature with a very

[3] http://cogcomp.cs.illinois.edu/page/demo_view/srl

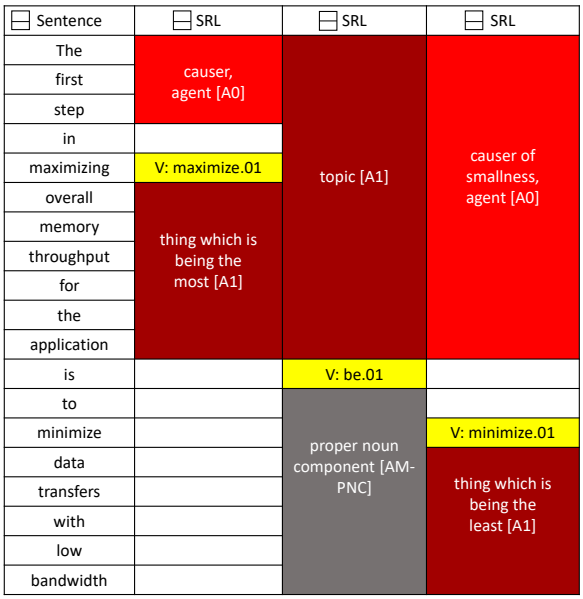


Figure 3: Semantic Role Labeling Results for a Sentence.

high accuracy. However, some more complicated steps have a much less satisfying accuracy (e.g., 75% for the semantic role labeling by SENNA [5]). The design of the selectors largely circumvents the limitations by simplifying the NLP tasks through the integration of the HPC domain properties. The categorizations of the patterns divide the overall problem into five simpler ones, reducing the complexities needed to handle by each of the selectors. That also reduces the influence of the errors from the NLP tools. For instance, although general semantic role labeling is hard to be accurate, the design of the selectors only relies on the "purpose" roles, thanks to the insights attained from the examinations of the HPC documents. Such roles can be recognized by the tool with a much larger accuracy (88.2%). In addition, each of the selectors incorporates the hints from some keywords, which also greatly reduces the complexity of the tasks and the reliances on the NLP analysis.

3.2 Stage II: Knowledge Recommendation

The second stage of Egeria is relatively easier. We model it as a text retrieval problem: From the advising sentences found by the first stage, it tries to identify those 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 [34] 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 [34], one of the best-known weighting methods. In TF-IDF, the weight vector for a sentence s is $\mathbf{v}_s = [w_{1,s}, w_{2,s}, \cdots, w_{N,s}]^T$. Each entry is computed as:

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

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:

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

Our implementation of VSM is based on Gensim [28].

An advising tool produced by Egeria reports the top-ranked sentences (having the similarity score no less than 0.15) as the answer to user's query. 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[4]), 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. (Support to other commonly used profiling reports will be added in the future.)

Egeria itself is a web-based tool. It is equipped with a document loader (which, for now, is customized for certain HTML documents). The loader extracts out all the contained sentences, and at the same time, infers the document structure (e.g., chapter, section, etc.) based on the indices or the HTML header tags. The structure allows the produced advising tools to be able to provide a richer context of the advising sentences.

The design of Egeria, including 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 EVALUATIONS

We conduct a set of experiments to examine the efficacy of Egeria. Our experiments are designed to answer the following three 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) Do we really need the sophisticated NLP-based design to recognize advising sentences? How much does it help compared to other designs?

We next report our experiments and results on each of the three questions. We start with a case study, showing how Egeria helps programmers address some performance issues of a CUDA program. We then provide some detailed examinations of the benefits of the two-staged design of Egeria, and give some comparisons to alternative methods.

4.1 A Case Study

The case 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[5], which was created to guide the development or optimizations of code to run on NVIDIA GPUs. We call the tool *CUDA Adviser*.

[4]https://developer.nvidia.com/nvidia-visual-profiler
[5]https://docs.nvidia.com/cuda/cuda-c-programming-guide/

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.

.....

Figure 4: Retrieved Sentences from Chapter 5 of CUDA Guide for a Given NVVP Report. (Highlighted are recommended sentences; others, including those omitted ones, are advising sentences in the same subsections as the recommended ones are.)

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 3.2). 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 case 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.

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.

```
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.

Figure 5: Optimization to Minimize Thread Divergence.

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*.

Figure 4 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:

> 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 Figure 4). 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 (if, switch, do, for, while) can significantly impact the effective instruction throughput by causing threads of the same warp to diverge (i.e., to follow different execution paths). 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 Figure 5a. The optimized version of the block is shown in Figure 5b which has the if-else branches removed.

In addition to NVVP reports, students also posted queries to the advising tool. One example query is "reduce instruction and memory latency". The sentences and corresponding section numbers returned by the advising tool are listed in Table 4. The retrieved nine sentences covered all three aspects of optimizations: maximize utilization, maximize memory throughput, and maximize instruction throughput, indicating that a diverse set of optimizations are available to reduce the instruction and memory latency. Some other 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 (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 5 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.

4.2 Detailed Examination and Comparisons

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 3.2 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:

Table 4: Retrieved Sentences from Chapter 5 for the Query: "reduce instruction and memory latency".

Section	Sentences
5.2.3. Multiprocessor Level	The number of clock cycles it takes for a warp to be ready to execute its next instruction is called the latency, and full utilization is achieved when all warp schedulers always have some instruction to issue for some warp at every clock cycle during that latency period, or in other words, when latency is completely "hidden".
	The number of instructions required to hide a latency of L clock cycles depends on the respective throughputs of these instructions (see Arithmetic Instructions for the throughputs of various arithmetic instructions); assuming maximum throughput for all instructions, it is: L for devices of compute capability 2.0 since a multiprocessor issues one instruction per warp over two clock cycles for two warps at a time, as mentioned in Compute Capability 2.x, 2L for devices of compute capability 2.1 since a multiprocessor issues a pair of instructions per warp over two clock cycles for two warps at a time, as mentioned in Compute Capability 2.x, 8L for devices of compute capability 3.x since a multiprocessor issues a pair of instructions per warp over one clock cycle for four warps at a time, as mentioned in Compute Capability 3.x.
	The number of warps required to keep the warp schedulers busy during such high latency periods depends on the kernel code and its degree of instruction-level parallelism.
	Having multiple resident blocks per multiprocessor can help reduce idling in this case, as warps from different blocks do not need to wait for each other at synchronization points.
5.3.2. Device Memory Accesses	For example, for global memory, as a general rule, the more scattered the addresses are, the more reduced the throughput is.
	In general, the more transactions are necessary, the more unused words are transferred in addition to the words accessed by the threads, reducing the instruction throughput accordingly.
	Also, it is designed for streaming fetches with a constant latency; a cache hit reduces DRAM bandwidth demand but not fetch latency.
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	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).

Table 5: 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

- 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 6 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;

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

Table 6 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 6. The Fleiss' kappa values [13] (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 6 shows, our advising tool returns most relevant advising sentences, with the recall rates at 83-100%. The small number of missing sentences are mostly due to some difficulties in advising sentence recognitions as detailed in the next sub-section. A fraction (0-35%) 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.

Table 6: Quality of Answers on Performance Queries.

NVVP Report	Performance Issues	#ground truth	Egeria Method			Full-doc Method			Keywords Method		
			P	R	F	P	R	F	P	R	F
knnjoin	Low Warp Execution Efficiency	6	0.667	1.0	0.8	0.146	1.0	0.255	0.154	1.0	0.267
	Divergent Branches	2	0.667	1.0	0.8	0.167	1.0	0.286	0.333	1.0	0.5
knnjoin_opt	Global Memory Alignment and Access Pattern	7	1.0	0.857	0.923	0.304	1.0	0.467	0.571	0.571	0.571
trans	GPU Utilization is Limited by Memory Instruction Execution	8	0.667	1.0	0.8	0.211	1.0	0.348	0.571	0.5	0.533
	Instruction Latencies may be Limiting Performance	11	0.667	0.909	0.769	0.182	0.909	0.303	0.364	0.364	0.364
trans_opt	GPU Utilization is Limited by Memory bandwidth	18	0.652	0.833	0.732	0.308	0.889	0.457	0.545	0.333	0.414

(P: precision; R: recall; F: F-measure)

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. An example sentence returned by "full-doc" on how to improve warp execution efficiency is

> Execution time varies depending on the instruction, but it is typically about 22 clock cycles for devices of compute capability 2.x and about 11 clock cycles for devices of compute capability 3.x, which translates to 22 warps for devices of compute capability 2.x and 44 warps for devices of compute capability 3.x and higher (still assuming that warps execute instructions with maximum throughput, otherwise fewer warps are needed).

It contains a lot of details but no advice for addressing the warp efficiency problem. As Table 6 shows, the precision of the returned results by the *full-doc method* is only 30% 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. For instance, consider the following sentence:

> To maximize global memory throughput, it is therefore important to maximize coalescing by: Following the most optimal access patterns based on Compute Capability 2.x and Compute Capability 3.x, Using data types that meet the size and alignment requirement detailed in Device Memory Accesses, Padding data in some cases, for example, when accessing a two-dimensional array as described in Device Memory Accesses.

It is a useful sentence on how to improve memory bandwidth. However, it contains no "bandwidth" in it. Although it contains "memory", that is such a common word that using it to search, the answer would contain 60 (four times of useful ones) sentences that are not helpful advices on improving memory bandwidth.

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.

Table 7: Statistics in Two Case Studies.

Documentation	Original document sentences (pages)	Egeria's selection sentences	Ratio
CUDA Guide[5]	2140 (275)	273	7.8
OpenCL Guide[6]	1944 (178)	440	4.4
Xeon Guide[7]	558 (47)	94	5.9

4.3 Effects of the Multilayered Design

The comparison with the "full-doc" method in the previous subsections indicate the important benefits of using the advising sentences that Egeria identifies. Recall that Egeria features a multilayered NLP-based design for recognizing advising sentences. In this part, we examine the detailed results of the recognition step, and assess the benefits yielded from that design through some comparisons.

In addition to CUDA Programming Guide[5], we applied Egeria to two more documents to show its robustness. The first is the AMD OpenCL Optimization Guideline[6], which is written by AMD for GPUs and fused accelerators it produces. The second is the Intel Xeon Phi Best Practice Guide[7], which is for the guiding programming on Intel Xeon Phi coprocessors. Table 7 reports the statistics of the original documents and those of the output of the first stage (advising sentence recognition) of Egeria. On the three documents, Egeria selects about 13%-23% of the original sentences as the advising sentences.

To examine the quality of the recognition, we conducted a deeper examination on these documents (only one chapter from the first two documents). We asked three domain experts to manually label all the sentences from *Chapter 5 Performance Guidelines* of CUDA Programming Guide, *Chapter 2 OpenCL Performance and Optimization for GCN Devices* of OpenCL Optimization Guide, and the entire Xeon document as advising sentences or non-advising sentences.

[6] http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/opencl-optimization-guide/
[7] http://www.prace-ri.eu/best-practice-guide-intel-xeon-phi-html/

Table 8: Evaluation of Advising Sentence Recognition on Three Guides.

Methods	CUDA (chapter 5)					OpenCL (chapter 2)					Xeon				
	Sel. Sents	Correct	P	R	F	Sel. Sents	Correct	P	R	F	Sel. Sents	Correct	P	R	F
Keyword	39	31	0.795	0.596	0.681	60	51	0.850	0.398	0.543	60	55	0.917	0.458	0.611
Comparative	5	4	0.800	0.077	0.140	18	15	0.833	0.117	0.205	2	2	1.0	0.017	0.033
Imperative	0	0	0	0	0	76	22	0.880	0.172	0.288	11	8	0.727	0.067	0.122
Subject	12	10	0.833	0.192	0.312	28	19	0.679	0.148	0.244	14	14	1.0	0.117	0.209
Purpose	23	21	0.913	0.404	0.560	17	15	0.882	0.117	0.207	17	16	0.941	0.133	0.234
KeywordAll	107	52	0.486	1.00	0.654	258	103	0.399	0.805	0.534	260	100	0.385	0.833	0.526
Egeria	59	48	0.814	0.923	0.865	126	102	0.810	0.797	0.803	94	85	0.904	0.708	0.794

(P: precision; R: recall; F: F-measure)

As some sentences appear vague in whether they provide advice on optimizations, there are slight discrepancies among the labels by different experts. The Fleiss' kappa value [13] of the labeling results is above 0.85 for the three guides, indicating large agreements among the raters. Through majority voting, we identify 52 out of 177 sentences of CUDA, 128 out of 556 sentences of OpenCL, and 120 out of 558 as the actual advising sentences; we use them as the ground truth.

We compared the recognition results of Egeria with the ground truth. The bottom row in Table 8 gives the results of Egeria. It successfully selects most of the advising sentences with only few false positives. The precisions are over 81%, the recall rates are 92%, 80%, 71% on the three guides respectively, and the F-measure values are 86%, 80%, 79%. Note that Egeria uses the same sets of configurations (selectors in Table 1 and keywords in Table 2) for advising sentence recognition from the three distinct documents. A fine tuning of the list of keywords can further improve the performance. For example, given the Xeon guide, after we added one extra keyword into the FLAGGING_WORDS list ('have to be') and two extra keywords into KEY_SUBJECTS list ('user', 'one'), the recall is improved to 0.892 with precision equaling 0.877.

The false negatives and false positives come from two main reasons. (1) Some sentences are ambiguous in whether they are advising sentences. An example is: "Native_functions are generally supported in hardware and can run substantially faster, although at somewhat lower accuracy." The ambiguity even causes some discrepancy among human raters as mentioned earlier. (2) The second reason is the mistakes made by NLP techniques. NLP techniques, especially semantic analysis, are yet to be improved in effectiveness. For example, an advising sentence is "As shown below, programmers must carefully control the bank bits to avoid bank conflicts as much as possible.". Theoretically speaking, this sentence meets the criteria of our selector VI (in Table 1) as a sentence with a purpose to avoid bank conflicts. However, the semantic role labeling tool failed to recognize "avoid bank conflicts" as a purpose argument in the sentence. The sentence was hence misclassified. Our design of selectors have already alleviated the limitations by simplifying the NLP tasks through the integration of the HPC domain properties and by combining the strengths of the multi-layered analysis. But avoiding all errors is difficult. As NLP techniques get better, the errors will be further reduced.

For comparison, the top five rows in Table 8 report the results if each of the five selectors in the multilayered design of Egeria is used alone, and the sixth row (*KeywordAll*) reports the result when 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. Some of these methods could achieve a good precision or recall, but at the time, suffer seriously on the other metric. *KeywordAll*, for instance, finds all the true sentences, but at the same time, finds some sentences which contain some of the keywords but are not actual advising sentences. An example sentence is "This section provides some guidance for experienced programmers who are programming a GPU for the first time". The sentence contains the keyword *programmer* but is not an advising sentence. By combining keywords with syntactic and semantic constraints effectively, Egeria avoids most of the false positives while missing only few true sentences. The results show that Egeria offers significantly better results than each of the alternatives.

5 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 [29], Oprofile [17], CodeAnalyst [10], and Gprof [14]) and some other data-centric tools [3, 16, 18, 19, 22]. Just for GPU, there are numerous performance profiling tools (e.g., NVVP [24], NVProf [24], CodeXL [31], GPU PerfStudio [32], Snapdragon [27]. There have also been many profiling tools developed for data centers and cloud (e.g., PerfCompass [9]). All these tools have concentrated on measuring the various performance aspects of an execution and identifying the main performance issues, rather than creating advising tools for offering advice on how to fix the issues.

The advising tools produced by Egeria are some kind of question-answer (QA) systems. QA systems have been developed in some other domains. Watson, for instance, is a QA system developed in IBM's DeepQA project [11]. The system was specifically developed to answer questions on the quiz show Jeopardy!. It has been later extended to the health care domain [12]. Egeria differs from those QA systems in two key aspects. First, Egeria itself is not a QA system but *a generator* of QA systems for various HPC domains. Second, traditional constructions of a QA system usually require lots of manual work. Egeria, on the other hand, produces the advising tools with no or minimum manual inputs (if the user decides to extend the set of keywords with some domain-specific ones). These appealing features are especially important for HPC because of the many subdomains it contains and the continuous fast evolvement of these domains.

NLP has been used in software engineering broadly. For instance, it has been used for some bug report classification [36], bug

report summarization [20], bug severity prediction [33], and relevant source files retrieval [35]. 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 [7]. 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.

6 CONCLUSIONS

We developed a framework named Egeria, which integrates existing NLP tools and domain knowledge, for auto-construction of HPC advising tools. Such advising tools provide users with a list of important optimization guidelines to remind them of available optimization rules, and can also suggest related optimization advice based on the performance issues of a program or questions from a user. We propose an unsupervised approach to recognizing advising sentences that distinctively integrates HPC domain properties with NLP techniques for a multilayered treatment of the problem. We performed our experiments on CUDA, OpenCL, and Xeon Phi optimization documents to demonstrate the usage of Egeria. A series of quantitative and qualitative evaluations demonstrate the effectiveness of Egeria in producing useful advising tools.

ACKNOWLEDGEMENTS

We are thankful to John Mellor-Crummey for shepherding the revision of this paper; his comments were extremely helpful. We also 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] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. 2010. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 685–701.

[2] Steven Bird. 2006. NLTK: the natural language toolkit. In *Proceedings of the COLING/ACL on Interactive presentation sessions*. Association for Computational Linguistics, 69–72.

[3] Bryan R Buck and Jeffrey K Hollingsworth. 2004. Data centric cache measurement on the Intel ltanium 2 processor. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 58.

[4] Xavier Carreras and Lluís Màrquez. 2005. Introduction to the CoNLL-2005 shared task: Semantic role labeling. In *Proceedings of the Ninth Conference on Computational Natural Language Learning*. Association for Computational Linguistics, 152–164.

[5] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. 2011. Natural language processing (almost) from scratch. *Journal of Machine Learning Research* 12, Aug (2011), 2493–2537.

[6] Shane Cook. 2012. *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes.

[7] Dipanjan Das and André FT Martins. 2007. A survey on automatic text summarization. *Literature Survey for the Language and Statistics II course at CMU* 4 (2007), 192–195.

[8] Marie-Catherine De Marneffe and Christopher D Manning. 2008. *Stanford typed dependencies manual*. Technical Report. Technical report, Stanford University.

[9] Daniel J Dean, Hiep Nguyen, Peipei Wang, Xiaohui Gu, Anca Sailer, and Andrzej Kochut. 2016. PerfCompass: Online Performance Anomaly Fault Localization and Inference in Infrastructure-as-a-Service Clouds. *IEEE Transactions on Parallel and Distributed Systems* 27, 6 (2016), 1742–1755.

[10] Paul J Drongowski, AMD CodeAnalyst Team, and Boston Design Center. 2008. An introduction to analysis and optimization with AMD CodeAnalyst Performance Analyzer. *Advanced Micro Devices, Inc* (2008).

[11] David Ferrucci, Eric Brown, Jennifer Chu-Carroll, James Fan, David Gondek, Aditya A Kalyanpur, Adam Lally, J William Murdock, Eric Nyberg, John Prager, and others. 2010. Building Watson: An overview of the DeepQA project. *AI magazine* 31, 3 (2010), 59–79.

[12] David Ferrucci, Anthony Levas, Sugato Bagchi, David Gondek, and Erik T Mueller. 2013. Watson: beyond jeopardy! *Artificial Intelligence* 199 (2013), 93–105.

[13] Joseph L Fleiss. 1971. Measuring nominal scale agreement among many raters. *Psychological bulletin* 76, 5 (1971), 378.

[14] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. 1982. Gprof: A call graph execution profiler. In *ACM Sigplan Notices*, Vol. 17. ACM, 120–126.

[15] Sandra Kübler, Ryan McDonald, and Joakim Nivre. 2009. Dependency parsing. *Synthesis Lectures on Human Language Technologies* 1, 1 (2009), 1–127.

[16] Renaud Lachaize, Baptiste Lepers, and Vivien Quéma. 2012. MemProf: A Memory Profiler for NUMA Multicore Systems. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 53–64.

[17] John Levon and Philippe Elie. 2004. Oprofile: A system profiler for linux. (2004).

[18] Xu Liu and John Mellor-Crummey. 2011. Pinpointing data locality problems using data-centric analysis. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*. IEEE, 171–180.

[19] Xu Liu and John Mellor-Crummey. 2013. Pinpointing data locality bottlenecks with low overhead. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*. IEEE, 183–193.

[20] Senthil Mani, Rose Catherine, Vibha Singhal Sinha, and Avinava Dubey. 2012. Ausum: approach for unsupervised bug report summarization. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 11.

[21] Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit.. In *ACL (System Demonstrations)*. 55–60.

[22] Collin McCurdy and Jeffrey Vetter. 2010. Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*. IEEE, 87–96.

[23] Aaftab Munshi, Benedict Gaster, Timothy G Mattson, and Dan Ginsburg. 2011. *OpenCL programming guide*. Pearson Education.

[24] CUDA NVidia. 2014. CUDA Profiler Users Guide (Version 6.5): NVIDIA. *Santa Clara, CA, USA* (2014), 87.

[25] Martha Palmer, Daniel Gildea, and Paul Kingsbury. 2005. The proposition bank: An annotated corpus of semantic roles. *Computational linguistics* 31, 1 (2005), 71–106.

[26] V. Punyakanok, D. Roth, and W. Yih. 2008. The Importance of Syntactic Parsing and Inference in Semantic Role Labeling. *Computational Linguistics* 34, 2 (2008). http://cogcomp.cs.illinois.edu/papers/PunyakanokRoYi07.pdf

[27] Inc. Qualcomm Technologies. 2016. Qualcomm Snapdragon Profiler Quick Start Guide. (2016).

[28] Radim Řehůřek and Petr Sojka. 2010. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. ELRA, Valletta, Malta, 45–50.

[29] James Reinders. 2005. *VTune performance analyzer essentials*. Intel Press.

[30] Michael Roth and Mirella Lapata. 2016. Neural Semantic Role Labeling with Dependency Path Embeddings. *CoRR* abs/1605.07515 (2016). http://arxiv.org/abs/1605.07515

[31] AMD Developer Tools Team. 2013. CodeXL Quick Start Guide. (2013). Retrieved Dec. 14, 2016 from http://developer.amd.com/tools-and-sdks/opencl-zone/codexl

[32] AMD Developer Tools Team. 2016. GPU PerfStudio. (2016). Retrieved Dec. 14, 2016 from http://developer.amd.com/tools-and-sdks/graphics-development/gpu-perfstudio

[33] Yuan Tian, David Lo, and Chengnian Sun. 2012. Information retrieval based nearest neighbor classification for fine-grained bug severity prediction. In *2012 19th Working Conference on Reverse Engineering*. IEEE, 215–224.

[34] Peter D Turney, Patrick Pantel, and others. 2010. From frequency to meaning: Vector space models of semantics. *Journal of artificial intelligence research* 37, 1 (2010), 141–188.

[35] Xin Ye, Razvan Bunescu, and Chang Liu. 2014. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 689–699.

[36] Yu Zhou, Yanxiang Tong, Ruihang Gu, and Harald Gall. 2016. Combining text mining and data mining for bug report classification. *Journal of Software: Evolution and Process* (2016).

A ARTIFACT DESCRIPTION: EGERIA: A FRAMEWORK FOR AUTOMATIC SYNTHESIS OF HPC ADVISING TOOLS THROUGH MULTI-LAYERED NATURAL LANGUAGE PROCESSING

A.1 Abstract

The description contains the information needed to launch/use our Egeria system. We explain how to launch the CUDA advisor used in our case study and also how to extend it to other HPC documents.

A.2 Description

This artifact consists of two parts: the Egeria framework for creating advising tools, and several advising tools that we have already built with Egeria. This description is mainly about the Egeria framework. The file README.md inside this artifact package (downloadable from Github as instructed below) explains the advising tools.

A.2.1 Check-list (artifact meta information).

- **Algorithm:** Semantic Role Labeling (SRL), Dependency Parsing, Word Tokenization, Stemming, Normalization, Term Frequency-Inverse Document Frequency (TF-IDF), Vector Space Model (VSM)
- **Program:** Python 2.7.10, HTML.
- **Data set:** CUDA Programming Guide(HTML), OpenCL Optimization Guide(HTML), Xeon Best Practice Guide(HTML).
- **Output:** suggestions (i.e., advising sentences) on program optimization given a query.
- **Experiment customization:** set of keywords used in the selectors, number of worker processes, IP address, port number.

A.2.2 How software can be obtained (if available). The code can be cloned from Github: https://github.com/guanh01/Egeria-demo. The code contains the three advising tools(cuda, opencl, and xeon) generated using Egeria and also the Egeria utilities.

A.2.3 Software dependencies. Our system uses the following packages: CoreNLP 3.7.0, Pycorenlp 0.3.0, Practnlptools 1.0, NLTK 3.2.1, Gensim 0.12.4, Textract 1.5.0, Gunicorn 19.6.0, Flask 0.11.1, BeautifulSoup 4.4.1.

- CoreNLP: We use the dependency parsing annotator in CoreNLP to mark dependencies among different parts of a sentence as part of the advising sentence recognition process, the first stage of Egeria. Enable CoreNLP server first to use the python wrapper *Pycorenlp*.
- Practnlptools: It provides a fast implementation of SENNA, one of the state-of-the-art Semantic Role Labeling algorithms.
- NLTK: English SnowballStemmer, WordNetLemmatizer, stopwords, word_tokenize and sent_tokenize are used for basic text preprocessing.
- Gensim: We use the TF-IDF and VSM model for sentence similarity search (knowledge recommendation).
- Textract: The package handles the parsing of pdf (NVVP reports generated from the NIVIDA Visual Profiler)
- Others: The packages provide web functionalities.

A.2.4 Datasets. We have three demo advising tools; each is for one of the HTML-based HPC documents: CUDA Programming Guide, OpenCL Optimization Guide, and Xeon Best Practice Guide. Other HPC documents can be easily processed with little extra work (need the parser to convert the document into a sequence of text blocks).

A.3 Installation

To launch a demo advising tool (e.g. the CUDA advisor):

- Install the following dependencies: Gunicorn, Flask, Textract, Gensim, NLTK, BeautifulSoup.
- Open the project folder and setup the host IP address (host) and the port number (port) in configuration files.
- Run in the command line (Linux only):

```
./run.sh
```

The default set of keywords used in the selectors are shown in Table 2 in the paper. The demo will automatically create a website with a webpage presenting the advising sentences generated by Egeria for CUDA Programming Guide as Figure 6. On top are two buttons (`Choose File` and `Upload`) through which users can upload performance reports (e.g., NVVP reports) as queries. They can also directly type in queries into the `search` box at the upper right corner of the webpage. The answers to the queries will then be displayed on the webpage as Figure 7 illustrates.

The source package includes Egeria utilities for construction of advising tools for other documents. The main process is as follows:

(1) Preprocess the documents into a sequence of text blocks. Since a raw document can be in various formats (e.g., txt, pdf, HTML, JSON, etc.), we do not provide a general API for this conversion.
(2) Enable CoreNLP according to the description on the official website [8].
(3) (Optional) Customize the set of keywords used in the selectors by modifying the configuration file: *Config.py*.
(4) Invoke Egeria, which will synthesize an advising tool for the new documents and display a webpage similar to Figure 6 but with the new advising sentences contained.

Users can then input queries to the advising tool through the web interface.

A.4 Experiment workflow

In Section 4 of the paper, we describe two ways in which we evaluate the quality of Egeria's optimization advice. Firstly, we asked domain experts to manually label sentences as advising or non-advising to generate the ground truth. The labeled data is used to evaluate the result of our advising sentence recognition method and the quality of the answers on performance queries. The labeled data, NVVP reports (used as queries), and the programs to generate reports are available on Github.

Secondly, we performed one case study which focused on an advising tool generated by Egeria with CUDA Programming Guide to show how one can use NVIDIA profiler data or questions to retrieve relevant and helpful tuning advice. In the case study, 37 graduate student were asked to manually optimize a program written using CUDA. Only a subset of students were given access to the advising tool. The program used in the experiment: *norm.cu*, is available on Github. The website that is available to students is shown in Figure 6. An example response is shown in Figure 7. The highlighted sentences are the recommended advising sentences. For a better understanding of these optimization instructions, we

[8] http://stanfordnlp.github.io/CoreNLP/corenlp-server.html

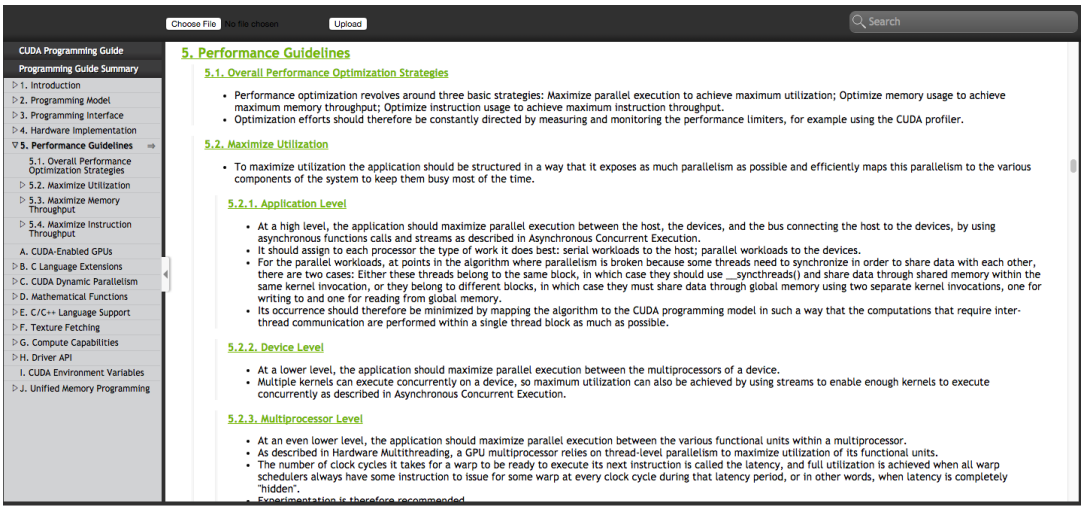


Figure 6: The initial webpage created by the demo, displaying the advising sentences of CUDA Programming Guide. The two buttons on top allow users to upload a performance report in PDF as a query. The search box at the right top corner allows users to directly input queries.

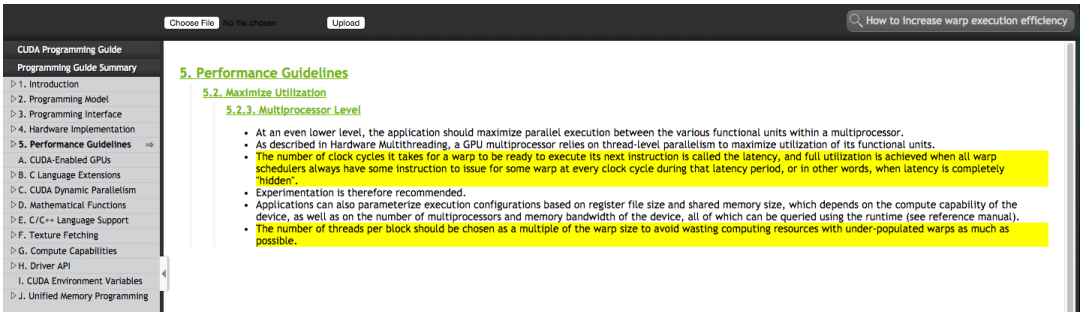


Figure 7: An example response to a query: How to increase warp execution efficiency. The tool automatically generates a HTML file to report the answers to the query. The file shows both the answers (highlighted sentences) and some context sentences of the answers. By clicking the hyper-links, users can easily view the part in the original document containing the answers.

provide links from these sentences back to the original document. The response webpage also shows other advising sentences in the same section.

A.5 Evaluation and expected result

The overall usefulness of Egeria is evaluated qualitatively through one case study and further examined quantitatively through comparison with other two methods. Details of these methods are described in Section 4.2. An example of expected results is shown in Figure 7 with the recommended advising sentences highlighted in yellow.

A.6 Experiment customization

The vocabulary is constructed based on the summary while the TF-IDF model is built on the whole document for more accurate weights for each indexed word.

The default similarity threshold to recommend a sentence is 0.15. A smaller threshold will lead to more sentence suggestions.