An Autotuning Protocol to Rapidly Build Autotuners

JUNHONG LIU, GUANGMING TAN, and YULONG LUO, State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, University of Chinese Academy of Sciences
JIAJIA LI, Computational Science and Engineering, Georgia Institute of Technology
ZEYAO MO, Institute of Applied Physics and Computational Mathematics
NINGHUI SUN, State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, University of Chinese Academy of Sciences

Automatic performance tuning (Autotuning) is an increasingly critical tuning technique for the high portable performance of Exascale applications. However, constructing an autotuner from scratch remains a challenge, even for domain experts. In this work, we propose a performance tuning and knowledge management suite (PAK) to help rapidly build autotuners. In order to accommodate existing autotuning techniques, we present an autotuning protocol that is composed of an extractor, producer, optimizer, evaluator, and learner. To achieve modularity and reusability, we also define programming interfaces for each protocol component as the fundamental infrastructure, which provides a customizable mechanism to deploy knowledge mining in the performance database. PAK's usability is demonstrated by studying two important computational kernels: stencil computation and sparse matrix-vector multiplication (SpMV). Our proposed autotuner based on PAK shows comparable performance and higher productivity than traditional autotuners by writing just a few tens of code using our autotuning protocol.

CCS Concepts: • **Computing methodologies** → *Parallel programming languages*; • **Software and its engineering** → *Application specific development environments*;

Additional Key Words and Phrases: Autotuner, knowledge database, protocol, stencil, SpMV

ACM Reference format:
Junhong Liu, Guangming Tan, Yulong Luo, Jiajia Li, Zeyao Mo, and Ninghui Sun. 2019. An Autotuning Protocol to Rapidly Build Autotuners. *ACM Trans. Parallel Comput.* 5, 2, Article 9 (January 2019), 25 pages.
https://doi.org/10.1145/3291527

1 INTRODUCTION

The multi/many-core technique results in more complexity and diversity of architectures and programming models, increasing the difficulty to develop high performance programs with reasonable efficiency. Traditionally, tuning code by hand is trick-intensive and requires programmers to be

This work is supported by the National Key Research and Development Program of China (2016YFB0201305, 2016YFB0200504, 2017YFB0202105, 2016YFB0200803, 2016YFB0200300) and National Natural Science Foundation of China, under Grant nos. 61521092, 91430218, 31327901, 61472395, and 61432018.
Authors' addresses: J. Liu, G. Tan, Y. Luo, J. Li, and N. Sun, Institute of Computing Technology, Chinese Academy of Sciences; Z. Mo, Institute of Applied Physics and Computational Mathematics; emails: liujunhong@ncic.ac.cn, tgm@ict.ac.cn, luoyulong@ncic.ac.cn, jiajiali@gatech.edu, zeyao_mo@iapcm.ac.cn, snh@ict.ac.cn.

2329-4949/2019/01-ART9 $15.00
https://doi.org/10.1145/3291527

highly knowledgeable about the relation between software and its mapping to hardware. Although such a hand-tuned code can achieve extremely high performance, it is usually not performance portable across different execution contexts due to the following reasons:

— *Input Variation:* Many applications have varying behaviors (i.e., locality and parallelism) depending on their inputs. For example, a seven-point stencil application (without any optimization) exhibits different data locality behaviors as the input data size changes from $256 \times 256 \times 512$ to $512 \times 512 \times 1024$, leading to a 10% performance variation on ×86 CPUs. The performance variation is caused by the input, but correlated with many other factors of architecture and application, which are complicated to analyze and optimize alone.
— *Compiler Variation:* On currently used computer systems, compilers or programming models play pivotal roles in performance tuning. Regardless of the performance gap caused by different compilers (i.e., Intel C compiler and GNU GCC), a meticulous difference of one compiler's options may lead to large variations in performance. Considering the huge number of modern compiler options, it is prohibitive to tune the optimal configuration by hand.
— *Hardware Variation:* With respect to the evolution of hardware, the optimal code on the latest architecture is almost certain to be suboptimal in the future. In fact, even on different implementations of the same ISA (i.e., Intel and AMD x86 CPUs), the optimal codes of an application may change significantly from each other. Obviously, it is extremely time- and labor-consuming to hand-tune codes on every platform.

As of recently, autotuning is becoming more mainstream as an important technique for optimizing the performance portability of Exascale applications (Basu et al. 2013a). Autotuners hide complexity and diversity of architectures and programming models by either autotuned libraries (Whaley and Dongarra 1998; Vuduc et al. 2005), adaptive performance tuning frameworks (Datta et al. 2008; Matthias et al. 2011; Chen et al. 2005; Hou et al. 2018, 2016; Wang et al. 2017; Liu and Vinter 2015b; Tan et al. 2018; Zhao et al. 2018), or autotuned algorithms (Hou et al. 2017; Liu and Vinter 2015a; Liu et al. 2017, 2018; Wang et al. 2016, 2018; Li et al. 2017a) for particular application domains. In general, a core part of autotuning is to tune optimizing strategies by considering variant program implementations and execution contexts (such as input datasets, compilers, and architectures). These strategies automatically explore a search space of possible optimization solutions by learning useful knowledge to help choose the optimal strategy (Matthias et al. 2011; Datta et al. 2008, 2009; Lutz et al. 2013; Khan et al. 2013; Cooper et al. 1999; Choi et al. 2010; Meng and Skadron 2009; Qasem et al. 2012; Williams et al. 2009; Christen et al. 2012; Ganapathi et al. 2009; Mametjanov et al. 2012). The knowledge refers to the relations among applications, execution contexts, and the above performance behaviors.

Although many auxiliary tools and algorithms are available, constructing an autotuning framework or autotuner for a specific demand presents an enormous challenge.

— *Constructing autotuners is challenging due to the lack of a modular, reusable framework.* Until now, researchers have developed a series of autotuning libraries and frameworks, some of which (e.g., FFTW (Frigo and Johnson 2005) and ATLAS (Whaley and Dongarra 1998)) have been extensively used in accelerating various applications. However, the implementations of one autotuning framework cannot be directly reused by another. Thus, we have to re-implement some autotuning strategies from scratch to construct a new autotuner, even though they have already been successfully applied in existing ones. Some researches have focused their initial efforts on simplifying the construction. For example, Ansel et al. proposed OpenTuner (Ansel et al. 2014), a fully customizable configuration representation,

as an extensible technique to allow domain-specific techniques simply using interfaces to communicate with the program to be autotuned.

— *Preserving performance knowledge is difficult because of the lack of an extensible, customizable approach.* Performance optimization/tuning is nontrivial work, especially for the performance-essential algorithms (i.e., stencil computation and sparse matrix operations in the following case studies). Numerous papers have been published about performance tuning on every generation of processors over the past several decades. However, a successful performance tuning is often determined by prior knowledge of programmers, which includes performance data, optimization parameters, characteristics of a specific program, and hardware features of a machine. The main goal in developing an autotuning protocol is to lower the knowledge requirements for programmers. For example, search-based autotuning (Kamil et al. 2010) acquires knowledge of the best optimization strategies using empirical trials, while machine-learning (ML)-based autotuners (Li et al. 2013; Tan et al. 2018) save the knowledge in a statistical model during the training phase. Unfortunately, different formats and organizations of the knowledge are employed among tools/algorithms, making it difficult to facilitate a comprehensive optimization search in an autotuning framework.

Therefore, most components of an autotuner, such as static analysis tools (Luo et al. 2015), search algorithms (Ansel et al. 2014), ML models (Li et al. 2013; Tan et al. 2018), domain-specific compilers (Matthias et al. 2011; Henretty et al. 2013; Maruyama et al. 2011), highly tuned algorithm libraries (Luo et al. 2015), and measurement tools (Malony et al. 1999), cannot be easily integrated together because of their non-uniform input and output interfaces. This fact obstructs the popularity of autotuning techniques.

In this work, we propose a performance tuning and knowledge-managing suite (**PAK**) to overcome the above-stated problems and to construct an infrastructure to facilitate building modular autotuners. The novelty of our work is highlighted by an autotuning protocol of five abstraction modules, *extractor*, *producer*, *optimizer*, *evaluator*, and *learner* to rapidly assemble an autotuner. We define their interfaces and specifications of corresponding inputs/outputs to customize modules with existing tools. With the five modules, users can easily build an autotuning skeleton and instantiate it by either using some available modules implemented inside the infrastructure or adding their own implementation as plugins. Performance knowledge shows the inherent relation between a given instance (e.g., a program or platform) and its optimization variants. Thus, a cornerstone of our methodology is a performance knowledge database, which stores and expresses the knowledge produced during the course of autotuning in a uniform way for the four basic data types. The organization of the database is extensible and supports importing new knowledge types. Moreover, PAK uses input and output interfaces for new tools to automatically recognize the customized knowledge and store them in the database. The main contributions of this article are as follows:

— We abstract a protocol for building an autotuner, which is composed of five basic procedures representing commonality of an autotuning framework. We further define autotuning programming interface (API) for constructing an autotuner with the proposed protocol.
— We design a performance knowledge database, which provides a customized, extensible, and uniform way to automatically preserve knowledge generated in the process of autotuning. The database provides interfaces to describe performance data by defining customized tools.
— We illustrate the usability of PAK by re-constructing two autotuners (Li et al. 2013; Luo et al. 2015; Tan et al. 2018) with the autotuning programming protocol and interface. Only a few tens of lines of code are needed to assemble the autotuner by leveraging the existing

modules. Compared to corresponding traditional autotuning approaches, PAK shows much higher productivity and comparable performance.

To the best of our knowledge, *this work is the first to present a generalized autotuning methodology for composable procedures which can be modularized as infrastructure by leveraging performance knowledge database*. However, this article is only an initiative to advocate a unified protocol for developing autotuners. A comprehensive implementation relies on the effort of the whole community, and therefore, we have opened the PAK source codes to the public (Liu et al. 2015).

The rest of this article is organized as follows. Section 2 introduces several basic terms of autotuning techniques. Section 3 presents the design and implementation of the autotuning infrastructure. Section 4 describes the integration of the performance knowledge database in the autotuning system. In Section 5, we present two application examples of PAK to construct an autotuner. Section 7 relates our system to other autotuning systems.

2 BACKGROUND AND MOTIVATION

2.1 Autotuning

Conceptually, autotuning explores a search space of possible *code variants* that are functionally equivalent, but different in the algorithms or implementations, and *optimization parameters* that represent various combinations of execution of variants. Usually, the autotuner appears as either an auto-tuning library (Vuduc et al. 2005; Whaley and Dongarra 1998) or adaptive performance tuning framework (Chen et al. 2005; Datta et al. 2008; Matthias et al. 2011).

Based on results from previous works on autotuners and their applications to domain-specific problems, most autotuners consist of four general phases, which are characterizing the running instance, generating an optimization candidate, performing the optimization strategies, and evaluating the optimization result.

The input to an autotuning system is referred to as the *running instance*, which is a combination of a program code, input, target platform, and back-end compiler. The program code and execution context are encapsulated as a whole to directly determine the actual performance of a program.

Autotuning is a process of finding the best optimization strategy from the optimization space for a given running instance. Each step in autotuning is an empirical trial of performing an optimization strategy on a running instance. The whole process is closely related to knowledge concerning the running instance and its performance behaviors which obviously requires that all components cooperate and share the knowledge amongst each other.

2.2 Autotuning Hierarchy

By dissecting the internal structure of most autotuning systems, we observe that they can be decomposed into several components and can be hierarchically structured as in Figure 1.

Logically, domain-specific autotuners can be formulated as *solutions* to specific *problems*. Here, the problem is defined as follows: given an application *domain*, find its optimal implementation on any computing *platform* and execution *environment*. In order to help solve the problem, researchers have developed a series of *tools* and *algorithms*. Theoretically, an autotuner should be easily built with the available tools and algorithms, although it is still built case-by-case and needs re-engineering for real-world implementations. Therefore, we aim to design a protocol for constructing autotuners in which a unified interface is defined to program any autotuner.

2.3 Autotuning Speed

Most autotuners employ a search-based approach to select a suitable optimization solution from a huge optimization space. Although a number of pruning strategies and search heuristics are

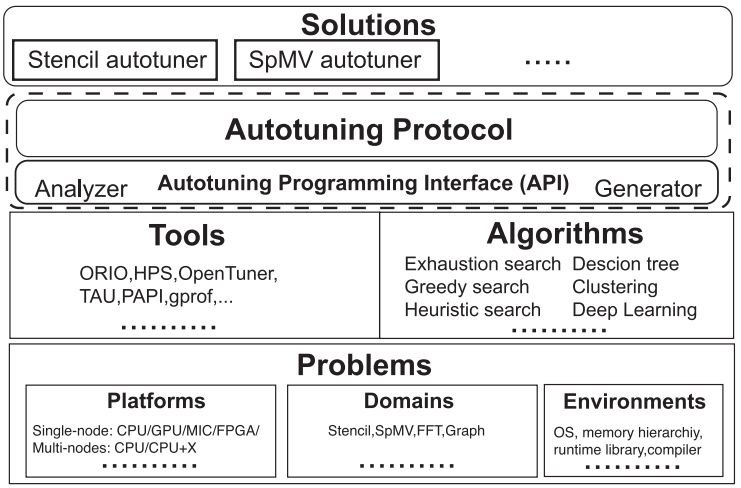


Fig. 1. An abstracted hierarchy of autotuning systems.

adopted to reduce optimization space, it still results in unacceptable overhead for some complicated problems. For example, our previous conference paper (Li et al. 2013; Luo et al. 2015; Tan et al. 2018) investigated several state-of-the-art stencil autotuners that require 10 or more hours tuning overhead. The main contribution of our previous work was to prove that a learning technique based on knowledge database is extremely efficient in speeding up the searching process. Specifically, the proposed optimal space model improves autotuning speed by two orders of magnitude while achieving comparable performance.

Therefore, these observations inspired us to develop an infrastructure to modularly construct an autotuner and comprehensively preserve the autotuning knowledge. By integrating existing and emerging tools and algorithms and employing a customizable, extensible, and uniform knowledge database, existing works could be reused to simplify and decrease the efforts of building an autotuner. Given a specific problem, traditional methodologies require a series of tedious procedures, which include analyzing the problem, selecting/creating tools, implementing algorithms, defining interfaces, and so on. In contrast, our methodology requires that developers only focus on analyzing the problem, selecting/customizing modules, and assembling autotuners.

3 AN AUTOTUNING PROTOCOL

The fundamental aspect of our PAK framework is an autotuning protocol composed of five customizable modules abstracted from current autotuning systems. In the protocol, we provide a set of abstract interfaces for the five elemental components to be customized by users to construct an autotuner. These interfaces imply workflow of an autotuner with PAK, as shown in Figure 2. The core modules, *extractor*, *producer*, *optimizer*, *evaluator*, and *learner*, cooperatively work as follows.

— *Extractor* characterizes a given running instance from different levels of the algorithm, architecture, and input and then extracts a set of performance features through one or more loosely coupled analyzers.
— *Producer* generates the optimization search space to determine one solution candidate for each tuning step. An input interface is used to receive the features, a tuning step, and its tuning score and has the ability to realize search algorithms, including both search-based and ML-based approaches.

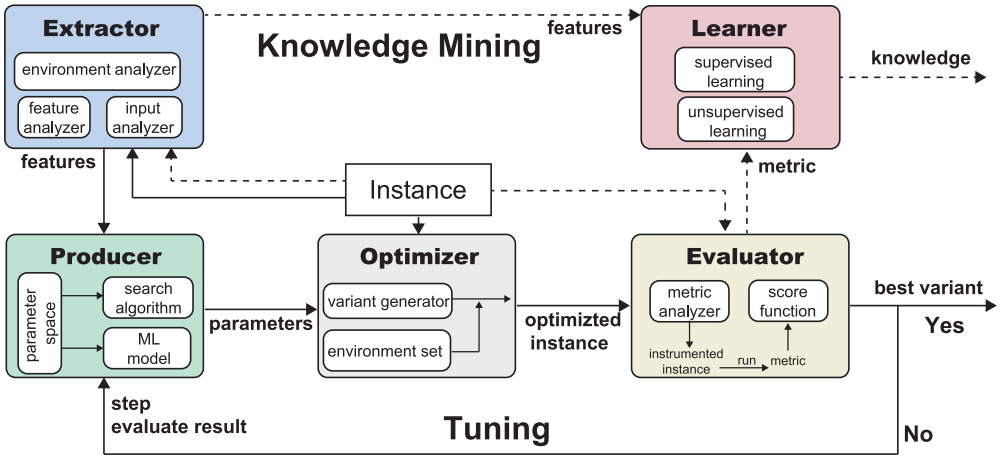


Fig. 2. The workflow of autotuning protocol in PAK.

```
#customize an Extractor object 'ext' using a set of analyzer objects
ext=PAK.Extractor$new(analyzers)

#invoke extractFeatures() and extract features from an application
ext$extractFeatures(app)
```

Fig. 3. Integrating analyzers in extractor.

— *Optimizer* employs some generators to perform optimizations with the produced optimization parameters.
— *Evaluator* measures the optimization result with one or more dynamic analyzers by implementing the optimized running instance as the input. Each of the features in these dynamic analyzers is related to a specific score function, which appraises the optimization result according to the preset object. If the appraisement satisfies the object, the autotuning stops and outputs the optimization result. Otherwise, the appraisement result is sent to the *Producer*, and a new tuning step begins.
— *Learner* obtains the tuning data from the performance knowledge database to build ML models, which facilitate the parameter generation of the ML-based *Producer*s. The knowledge data generated during the course of autotuning are persevered in a uniform format in the database.

In the following context, we present the internal mechanism of each module.

3.1 Extractor

Based on ML, the optimization parameters in traditional autotuning systems are produced according to the features of the running instance. These features are key in prompting the autotuner to achieve high performance. PAK provides an abstract class `Extractor` to define a feature extractor.

The extractor requires a customized analyzer to perform a comprehensive analysis on a given running instance prior to optimization. It takes a running instance as the input and outputs a list of features. These features characterize the running instance from multiple aspects and are used to facilitate the prediction of optimization parameters and mining knowledge for performance tuning. Figure 3 illustrates an example of an instantiating extractor. The analyzers associate the

```
# an exhaustion search Producer class
PAK.Producer.Exhaustion<−setRefClass(
  "PAK.Producer.Exhaustion",
  contains="PAK.Producer",
  fields = list(parameter.space="data.frame"),
  methods = list(
    #init function
    initialize=function(parameter.space){
    parameter.space<<−parameter.space
    },
    #implemente the interface method
    getParameter=function(step,extractor.result,score)
    {
      if(step<nrow(parameter.space))
        return(parameter.space[step,])
      else
        return (NULL)
    }
  )
)
```

Fig. 4. A producer instance of exhaustion search.

names of the required analyzer with features, and initialize an extractor object `ext`, that invokes `extractFeatures` to extract user-specified features from the running instance app.

3.2 Producer

There is a significant diversity among existing autotuning systems in terms of parameter generation for which several methods include exhaustive, greedy, and machine-learning algorithms. In general, optimization parameters are determined by features of a running instance, tuning step, and tuning result. In our proposed methodology, PAK abstracts a `Producer` class to generate parameter candidates.

The producer is a module that generates optimization parameters according to the features. The module uses a pure virtual method defined as `getParameter(step,extractor.result,score)`, which is an interface of both the input and output of the producer. More specifically, the argument `step` is the current tuning step, `extractor.result` contains features of the running instance, and `score` is the tuning result.

Figure 4 describes a producer of exhaustion search. The class `PAK.Producer.Exhaustion` inherits the abstract class `PAK.Producer`, and defines a constructor that inputs an argument of `parameter.space`. The `parameter.space` is a `data.frame` object, containing all candidate optimization parameters. It implements an interface `getParameter(...)`, which returns the optimization parameters in optimization space in sequences.

3.3 Optimizer

In an autotuning system, optimization strategies are often instantiated as a parameterized compiler, collection of algorithms, or highly tuned library. PAK abstracts the `Optimizer` class to implement this functionality by taking an optimization parameter that represents one or more optimization strategies and instantiates them using a custom generator. Figure 5 displays an example of optimizer. The `hpsGen` (Luo et al. 2015) is a code generator for stencil applications that implements multiple optimization algorithms. In the example, we first use the constructor of `PAK.Optimizer` with argument `hpsGen` to initialize an optimizer `opt`, which accepts arguments from the running

```
#customize an Optimizer object 'opt' uses the hpsGen
opt=PAK.Optimizer$new("hpsGen")

#instances the optimization strategies on the running instance
opt$optimize(app,parameters)
```

Fig. 5. An optimizer with a stencil code generator.

instance and optimization parameters. Then, the `opt` invokes `hpsGen` to perform optimization. The optimized running instance is returned and saved in variable `optimized.instance`.

3.4 Evaluator

After optimization, the autotuning system evaluates the current solution. Traditionally, the optimization object is expressed as a single-objective (execution time) minimization problem. Given the current and projected changes in computer architectures, this formulation is insufficient due to a wide variety of emerging autotuning problems. To amend such issues, metrics, such as power, performance, energy, and resiliency, need to be targeted together. The autotuning system is supposed to have the ability to consider multiple autotuning objects together and choose the most suitable one under some tradeoff threshold.

The evaluator is a module that applies multiple analyzers and score functions to appraise an optimization solution. It manages one or more analyzer objects and profiles features of power, performance, and energy on an optimized running instance. According to the optimization object, the relative features are enabled and individually associated with a specific score function $\mathcal{F}_{score_i}$.

$$\mathcal{F}_{score_i} = \begin{cases} -distance & worse \\ 0 & equal\ or\ better. \end{cases}$$

This function inputs the profiling result on the $feature_i$ and returns a negative score that represents the distance between the current optimization solution and optimization object. Moreover, when the $feature_i$ satisfies the optimization object, the function $\mathcal{F}_{score_i}()$ returns zero.

$$Score = \sum_{i}^{n} \mathcal{F}_{score_i}.$$

For a multi-objective autotuning system, the score of an optimization solution is the sum of all results of the score functions. When all optimization objects are satisfied, the score turns to zero, and autotuning is completed.

Figure 6 presents a multi-objective autotuning problem of performance and energy. Since the energy is related to the data movement, we measure this feature by the number of load/store instructions. We first create two variables to store the score functions. The first score function sets the threshold of time at $100ms$, and the second score function sets the threshold of the number of load/store instructions at 10^7. Then, we use tuning and analysis utilities (TAU) (Malony et al. 1999) as the performance and energy analyzer and associate the features of P_WALL_CLOCK_TIME and PAPI_LST_INS with the previous score functions. We chose TAU in our article for two reasons. First, TAU performance system is a portable profiling and tracing toolkit for performance analysis of parallel programs, which supports comprehensive performance profiling and tracing and targets all parallel programming/execution paradigms. Second, TAU integrated performance toolkit, including instrumentation, measurement, analysis, visualization, widely ported performance profiling/tracing system, performance data management and data mining, and open source (BSD-style license).

```
#define two score functions
s1=function(x){if(x>100) return (100−x) else return(0)}
s2=function(x){if(x>10^7) return (10^7−x) else return(0)}

#wraps the analyzers with the score functions
subevaluators=list(tau=list(P_WALL_CLOCK_TIME=s1,
                            PAPI_LST_INS=s2))

#customize an Evaluator object 'eva' using the analyzers
eva=PAK.Evaluator$new(subevaluators)

# evaluate an running instance and return the result score
score=eva$evaluate(optimized.instance)
```

Fig. 6. An evaluator object that customized with TAU.

The `Evaluator` is created and appraises the optimized running instance, and the result is returned and saved as the variable `score`. If the `score` is zero, the optimized running instance satisfies the optimization object. Otherwise, a performance gap exists between the current running instance and its optimization object, whereby further optimization is needed.

3.5 Learner

ML allows us to draw inferences from models automatically constructed from large quantities of data. In contrast to search-based approaches, ML-based autotuning generates optimization solutions directly with relatively low overhead. Another advantage is that it does not rely on either the application or architecture knowledge.

The learner, building a ML model to facilitate ML-based producers, is an abstract class with interface `learnModel`, which contains arguments `(training.data,idv,dv)` that represents training data, list of independent variables, and dependent variables, respectively. The user can also implement other machine-learning models, such as Convolutional Neural Networks (CNNs). The training dates are provided by the users of the PAK. The features are able to be listed by the users of the PAK using the format of the PAK. The accuracy of the trained models is related to the trained models of the applications.

A learner `PAK.Learner.DecisionTree` for a decision tree model is illustrated in Figure 7. As a subclass of `PAK.Learner`, it implements the interface `learnModel(training.data,idv,dv)`, which creates a string to store the commands for building a model. Then, it executes the command and employs the function `rpart` to learn a model. The `rpart` represents recursive partitioning for classification, regression, and survival trees. Then, the generated model is packed with the names of independent and dependent variables.

3.6 Assembler

In addition to the five core modules, the autotuning protocol provides a mechanism to assemble an autotuner upon them. In this layer, users integrate analysis and predefined optimization tools. There are two base classes of assemblers to be implemented, including an `analyzer` and `generator`.

The `analyzer` is used to analyze the features of a running instance, including both static and dynamic behaviors. To make use of the built-in analysis tools in PAK, users need a launch script and format definition file (FDF), which describes the format information of the features produced by analyzers. In our implementation, the FDF adopts XML specification and each feature is defined with four attributes in a FDF, including

```
# an decision tree Learner Class
PAK.Learner.DecisionTree<−setRefClass(
  "PAK.Learner.DecisionTree",
  contains="PAK.Learner",
  fields = list(model="list",dv.name="character",
                idv.name="character"),
  methods = list(
    #init function
    initialize=function(){
    },
    learnModel=function(training.data,idv,dv){
      buildstr<−sprintf("rpart(%s~.,training.data)",dv)
      model[["rp"]]<<−eval(parse(text=buildstr) )
      model[["dv.name"]]<<−dv
      model[["idv.name"]]<<−idv
      return (model)
    }
  )
)
```

Fig. 7. A learner that trains a decision tree model.

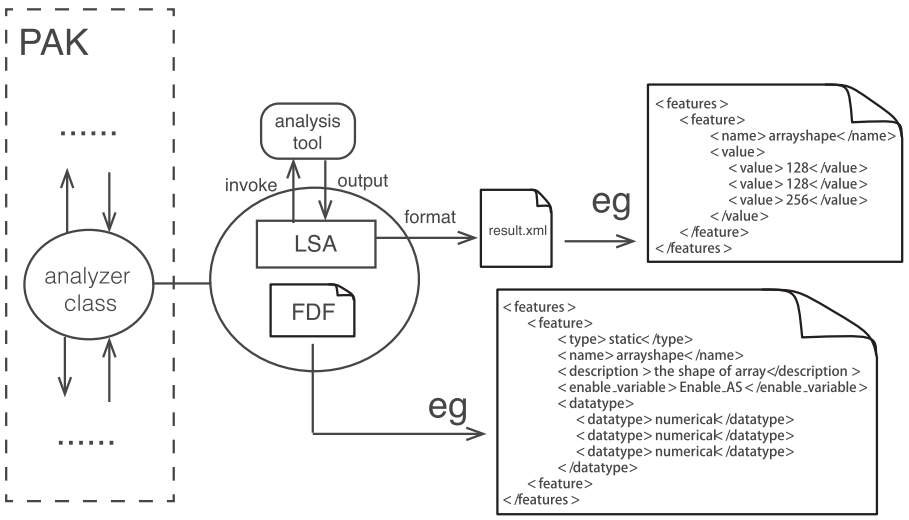


Fig. 8. An example of FDF for a customized analyzer.

—*Name*: a word or a term used for identifying a feature.
—*Description*: presents the characteristics of a feature.
—*Enable variable*: the environment variable that is used to control feature analysis. PAK notices the launch script to produce the specific feature by setting the enable variables.
—*Data type*: determines the data type of a feature. (In PAK, we use five data types which will be discussed in Section 4.)

A launch script of an analyzer (LSA) wraps existing analysis tools, where the input consists of a running instance and a series of enable variables. It outputs an analysis report in XML format. Figure 8 shows an example of FDF which defines a feature called `arrayshape` and characterizes a running instance in terms of the array's shape. PAK enables the features by setting the

environment variable `Enable_ arrayshape`. The data type is a combination type that represents a nested structure containing three numerical data. As mentioned before, the output file contains an analysis for feature `arrayshape`, which is (128, 128, 256) in this example.

The `generator` is used to perform optimization strategies for a running instance. In our implementation of the autotuning protocol, it shares the same mechanism with the `analyzer`. Therefore, users should also provide a launch script and FDF as the input of the generator to describe the format information of the optimization parameters. The optimization parameter of a generator is also defined with four attributes, which are the same features in the analyzer. The launch script of generator (LSG) wraps an existing external optimization tool, has the same input interface with LSA, and provides an approach for PAK to perform optimization strategies using an external optimization tool.

4 PERFORMANCE KNOWLEDGE

A remarkable advantage of PAK is the extensibility that promises an easy employment of third-party tools. These tools provide measurement, analysis, and optimization capabilities for a running instance in a tuning step to produce valuable data. We refer to them as *knowledge data* because they contain the characterization of the running instances, parameters of optimization, and the measurement of the optimized running instance. The knowledge data encodes a tuning step with three aspects of characterization of a running instance, optimization strategy, and measurement results. However, most tools use fixed knowledge data with their own formats, which cannot be recognized by each other.

PAK provides an interface for customizing tools, including a FDF describing knowledge data and a launch script that standardizes the input and output files. The knowledge data are understood automatically by PAK and converted to the format that is compatible with the knowledge database. The knowledge database then employs a uniform format that consists of four data types to express the knowledge data.

4.1 Uniformity

The knowledge data produced by external tools are transformed into a uniform format and then stored in the database. To express the unified knowledge, we define four data types:

- *Category:* expresses non-countable data including CPU/GPU type, compiler version, compiler options, and so on.
- *Numerical:* expresses quantifiable data, such as the number of cores/instructions, time, and energy.
- *Boolean:* expresses data that only can be assigned as two values (usually denoted true and false), including optimization switch, algorithm selection, and so forth.
- *Combination:* expresses complicated data that contain multiple sub-data. These sub-data can have either identical or different data types. As shown in Figure 9, a combination data type may contain the category, numerical, Boolean, or other combination types. For example, the dimension feature could be a 3-numerical combination data, and the tiling parameter could be a 9-numerical combination data.

4.2 Customizability

To use external tools and the knowledge data comprehensively in a customized framework, we provide an input and output interface to the external tools and pack them into two basic classes in PAK.

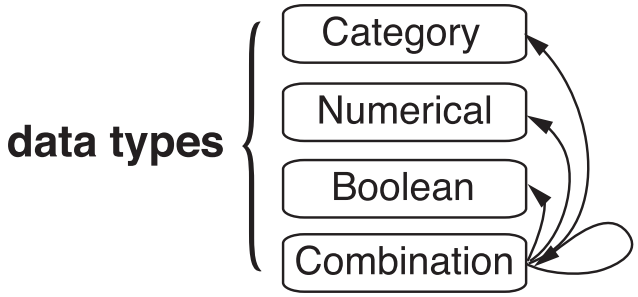
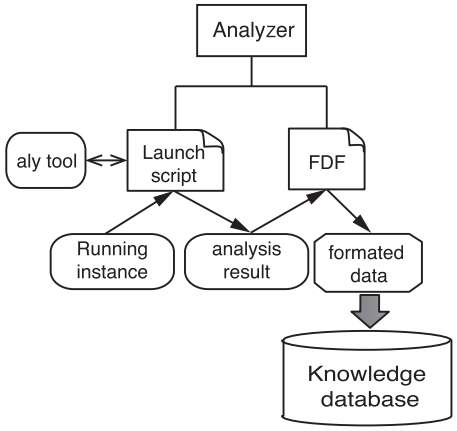
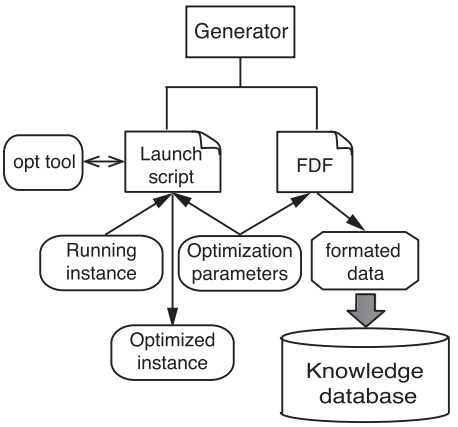


Fig. 9. The basic data type of knowledge data.



(a) The analyzer performs static/dynamic analysis and formats features as knowledge data.



(b) The generator applies optimization strategies and formats optimization parameters as knowledge data.

Fig. 10. An illustration of interaction with performance knowledge database.

As shown in Figure 10(a), an analysis tool is encapsulated in an `Analyzer` object. Given a running instance, the `Analyser` employs analysis tools through the launch script. Then, it reads the FDF and uses the format information to automatically convert the analysis result into the uniform knowledge and store it to the knowledge database. On the other hand, at each tuning step, optimization tools receive a set of parameters and perform the corresponding optimization strategies to optimize a given running instance. To preserve these optimization parameters, PAK abstracts them as an optimizer class. In Figure 10(b), an optimization tool is encapsulated in a `Generator` object. The `Generator` receives a set of parameters and passes them to the optimization tools through the launch script. To preserve these optimization parameters, the `Generator` parses them to the uniform knowledge using the FDF and stores them to the knowledge database.

4.3 Extensibility

The database structure consists of a main table and multiple sub-tables. The main table stores the main key of sub-tables, and each record represents a tuning step. The sub-tables keep knowledge data generated by external tools, and each record represents knowledge data generated by external tools in a tuning step.

During each tuning step, PAK reads the format information from the FDF of each external tool and recognizes the generated knowledge data. As shown in Figure 11, PAK creates a record in

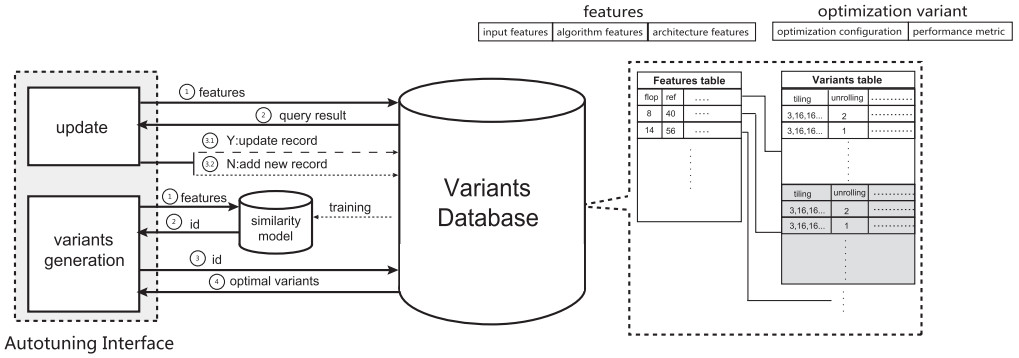


Fig. 11. Knowledge database supports extensible knowledge data preservation.

the main table to store knowledge. When the knowledge data is initially stored to the database, a registration is invoked. Then PAK creates and initializes a table according to the format of the knowledge data, and adds a field in the main table for the foreign key that is connected to the new table. When the tool has been registered, the knowledge data are stored into the sub-table and a new record is created. The primary key of this record is added to the field in the main table.

Once a training running instance has been executed, a running record is updated in the database by an **`Update`** function. For each running record, there is one feature vector that is extracted by the feature analysis module, and one solution vector containing optimization parameters, as well as the corresponding performance results, is collected by the evaluator. If there is no matched record in either of the two tables, the **`Update`** function adds the corresponding feature vector or solution vector. Otherwise, the new training record is updated in the database.

5 STENCIL AUTOTUNER

In this section, we apply PAK to implement a stencil autotuner (*PAK-Stencil*), shown in Figure 12. In fact, we implement all components of PAK autotuning protocol by integrating our previous work (Luo et al. 2015) into the PAK infrastructure. According to the autotuning protocol, PAK-stencil is composed of five modules: the extractor, producer, optimizer, evaluator, and learner. Figure 13 shows the implementation of the stencil autotuner based on the autotuning programming interface.

As an overview of the internal structure, we adopt the high performance stencil (HPS) frontend (Luo et al. 2015) to extract the features of the running instance, which is registered as an analyzer class by providing two interface files. We use the 19 feature parameters defined in Luo et al. (2015) for the extractor. Then, we employ the Optimal-Solution Space (OSS) (Luo et al. 2015), which is a module used to find optimal solutions comprised of a model and database as the tuning algorithm. The HPS backend is responsible for code generation and performing optimizations. We provide two interface files to register it as a generator class and construct the `Optimizer`. At last, we use TAU (Malony et al. 1999) to evaluate the optimization results, which measures the running time and provides a score.

5.1 Experimental Setup

We conducted the experiments on a 16-core symmetric multiprocessing (SMP) system, integrating two Intel Xeon E5-2670 multicore CPUs (116.4 GFlops double-precision and 332.8 GFlops single-precision) and a NVIDIA Tesla K20c GPU (1.17 TFlops double-precision and 3.52 TFlops

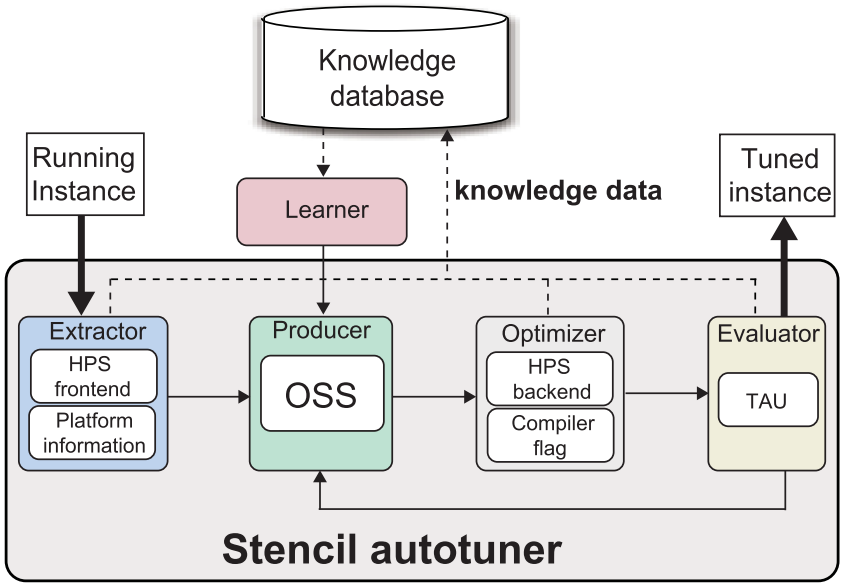


Fig. 12. Architecture of stencil autotuner with PAK protocol.

single-precision). An Intel compiler version 13.1 and an NVCC version 6.0 were used as the backend compilers.

5.1.1 Stencil Autotuners. Here, we compared PAK-Stencil with other stencil autotuning frameworks, including *SDSL and Patus*. For simplicity, we only report the experimental results for the $128 \times 128 \times 256$ problem size.

— *SDSL (Henretty et al. 2013)*: The stencil domain-specific language (SDSL) can be embedded in C/C++/MATLAB and focuses on polyhedral compiler optimization for short-vector SIMD and CUDA. There is a built-in autotuning module that implements an improved brute-force search with the help of tuning annotations in the language.
— *Patus (Matthias et al. 2011)*: The Patus stencil optimization framework focuses on the autotuning strategy itself, which is the most similar counterpart to PAK-Stencil. There is a flexible autotuner that integrates several efficient heuristic algorithms and provides language support to annotate the tuning strategy.

5.1.2 Stencil Applications. We evaluated the stencil autotuners with a benchmark set composed of five stencil computation applications:

— **HEAT** (Henretty et al. 2013): a 3D seven-point stencil with order-1 in which the distance of all stencil points from the center point is 1. It contains 15 floating point operations for each point, and there is one array for reading, and one array for writing:

$$\begin{aligned} u'_{i,j,k} = {} & \alpha(u_{i+1,j,k} - \beta u_{i,j,k} + u_{i-1,j,k}) \\ & + \alpha(u_{i,j+1,k} - \beta u_{i,j,k} + u_{i,j-1,k}) \\ & + \alpha(u_{i,j,k+1} - \beta u_{i,j,k} + u_{i,j,k-1}) + u_{i,j,k}). \end{aligned}$$

```
#Extractor
myextract<−PAK.Extractor$new(
list(hps_frontend("FD","RD","AN","LR","AC","DT","PS","IT"),
     platforminfo("core","freq","cache","bd")))

#Producer
myproducer<−PAK.Producer.OSS$new(parameter.list)

#Optimizer
myoptimizer<−PAK.Optimizer$new(generator.name="hps_gen")

#Evaluator
myevaluator<−PAK.Evaluator$new(
sub.evaluators=list(tau=list(
                          P_WALL_CLOCK_TIME=function(x){
                          if(x>0) return (−x) else return(0)}
                          )))

#Learner
for(app in training.stencils)
{
  tuning<−PAK.Tuner$new(app=app,optimizer=myoptimizer,
                        evaluator=myevaluator,producer=myproducer,
                        need.store=TRUE)
  tuning$tune()
}
mylearner<−PAK.Learner.OptimalSpace$new(trainingData)
myproducer<−PAK.Producer.OptimalSpace$new(mylearner$learn())

#Autotuning using the oos model
for(app in test.stencils)
{
  t<−targettime[[app]]
  myevaluator<−PAK.Evaluator$new(
  sub.evaluators=list(tau=list(
                          P_WALL_CLOCK_TIME=function(x){
                          if(x>t)return (−x) else return(0)}
                          )))
  tuning<−PAK.Tuner$new(app=app,extractor=myextract,
                        optimizer=myoptimizer,evaluator=myevaluator,
                        producer=myproducer)
  tuning$tune()
}
```

Fig. 13. Implementation of stencil autotuner with PAK programming interface.

—**HIMENO** (Himeno 2011): a 19-point stencil with order-1, which has 13 input arrays and 1 output array. It is a linear solver for the 3D pressure Poisson equation, which appears in an incompressible Navier-Stokes solver:

$$\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} + \frac{\partial^2 p}{\partial z^2} + \alpha \frac{\partial^2 p}{\partial xy} + \beta \frac{\partial^2 p}{\partial xz} + \gamma \frac{\partial^2 p}{\partial yz} = \rho.$$

—**FDTD** (Kunz and Luebbers 1993): an application that consists of three order-1 stencil kernels, 75 floating point operations, and 8 read/write arrays. We list one of the three stencils

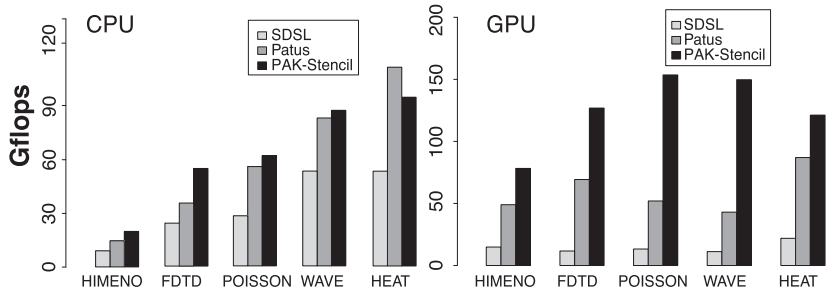


Fig. 14. Comparison of performance on both CPU and GPU.

below:

$$\begin{aligned} miu_8 &= 0.5miu0(miur_{i,j,k-1} + miur_{i,j,k}),\\ msigma_8 &= 0.5(msigma_{i,j,k-1} + msigma_{i,j,k}),\\ CP &= (2miu_8 - dt_8 msigma_8)/\\ &\quad (2miu_8 + dt_8 msigma_8),\\ CQ &= 2dt_8/(2miu_8 + dt_8 msigma_8),\\ Hx'_{i,j,k} &= CPHx_{i,j,k} + CQ((Ey_{i+1,j,k} - Ey_{i,j,k})\\ &\quad /dz - (Ez_{i,j+1,k} - Ez_{i,j,k})/dy). \end{aligned}$$

—**WAVE** (Micikevicius 2009): a 3D 25-point stencil kernel with order-4 that reads and writes one array during computing and has 30 floating point operations. It is made up of a center point and 24 points that are a distance of 1 to 4 from the center of the coordinate axis:

$$\begin{aligned} pa'_{i,j,k} = [&\alpha(pa_{i\pm1,j,k} + pa_{i,j\pm1,k} + pa_{i,j,k\pm1})\\ &+ \beta(pa_{i\pm2,j,k} + pa_{i,j\pm2,k} + pa_{i,j,k\pm2})\\ &+ \gamma(pa_{i\pm3,j,k} + pa_{i,j\pm3,k} + pa_{i,j,k\pm3})\\ &+ \delta(pa_{i\pm4,j,k} + pa_{i,j\pm4,k} + pa_{i,j,k\pm4})\\ &+ \epsilon pa_{i,j,k}]vsq_{i,j,k} + 2pa_{i,j,k}. \end{aligned}$$

—**POISSON** (Unat et al. 2011): a 3D 19-point stencil with order-1 and 20 float operations per stencil updating. The total number of arrays to be accessed is three, two for reading and rest and one for writing:

$$\begin{aligned} u'_{i,j,k} = c_0[&b_{i,j,k} + c_1(u_{i,j,k\pm1} + u_{i,j\pm1,k} + u_{i\pm1,j,k})\\ &+ u_{i\pm1,j\pm1,k} + u_{i\pm1,j,k\pm} + u_{i,j\pm,k\pm})]. \end{aligned}$$

5.2 Performance

We report the performance in Gflops for the five benchmark applications described above. On average, *PAK-Stencil* improves performance by 3.28 times over *Baseline*, 100% times over *SDSL*, 18% times over *Patus* on the CPU, while on GPU it increases performance by 4.86 times over *Baseline*, 8.3 times over *SDSL*, and 1.25 times over *Patus*.

CPU: The left bar graph in Figure 14 compares the performances of the CPU implementations. First, PAK-Stencil improves performance by 1.4–6.2 times over *Baseline*, 64%–120% over *SDSL*, and −15%–53% over *Patus*. *PAK-Stencil* employs the 2.5D blocking strategy (Nguyen et al. 2010) to reuse data between two adjacent blocks. For memory bound applications, such as FDTD and HIMENO, we obtained better performance results than *Patus*. However, for applications with high operational intensity, such as HEAT, Patus surpasses PAK-Stencil due to its explicit SIMD optimization using intrinsic functions. *SDSL* also works well on short-vector SIMD architectures and performs tiling optimizations on temporal and spatial dimensions. However, the time-tiling used is restricted

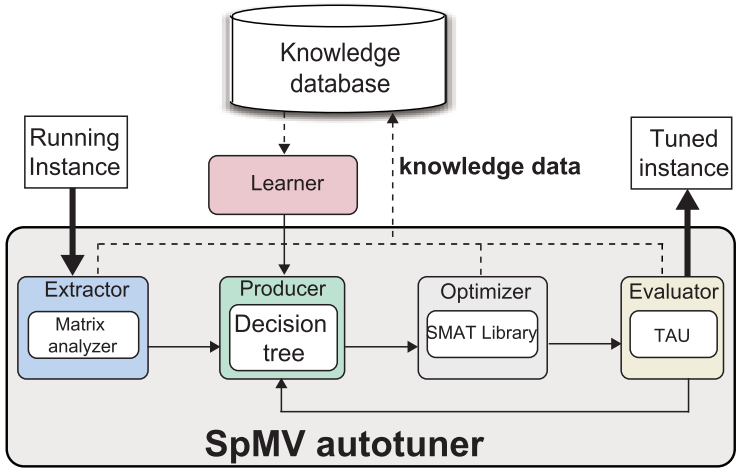


Fig. 15. Architecture of SpMV autotuner with PAK protocol.

by the growing size of tiles when the dimension is larger than two. This leads to its unsatisfactory performance on five 3D applications. The performance improvement differs with the five stencil applications. For example, PAK-Stencil achieves higher speedups compared to speedups for HEAT and WAVE than for HIMENO and FDTD. The difference is related to the algorithmic feature of *Operational intensity*, which sets the upper bound performance according to the roofline model (Williams et al. 2009).

GPU: The right bar graph in Figure 14 shows the performance comparison of the GPU implementations. PAK-Stencil improves performance by 58% to 7 times over *Baseline*, 4.24 to 12.3 times over *SDSL*, and 39% to 2.48 times over *Patus*. Unlike the implementations on CPU, *PAK-Stencil* outperforms the GPU counterparts. One major reason is that our optimization solutions adopt the 2.5D blocking method, through which we can put the block date into the shared memory and constant memory, and implement texture reference optimization strategies. Contrarily, both *SDSL* and *Patus* only employ straightforward parallelization and blocking strategies.

6 SPMV AUTOTUNER

We also apply PAK to implement a sparse matrix-vector multiplication (SpMV) autotuner (*PAK-SpMV*). Here, we integrate the major components of the SpMV autotuner (SMAT) (Li et al. 2013; Tan et al. 2018) into the PAK infrastructure by following our autotuning protocol, which is composed of an extractor, producer, optimizer, evaluator, and learner. Figure 15 depicts the architecture of our new SpMV autotuner. Figure 16 demonstrates the implementation of autotuner codes based on the autotuning programming interface.

The extractor extends the feature extractor from our previous work (Li et al. 2013; Tan et al. 2018) to analyze blocking matrices. There are 13 features to be extracted which include the numbers of rows (M), columns (N), Diagonals (Ndiags), non-zeros (NNZ), and the ratio of "true" diagonals to all diagonals, average number of non-zeros per row (aver_RD), maximum number of non-zeros per row (max_RD), variation of non-zeros per row (var_RD), ratio of non-zeros in DIA (ER_DIA), ratio of non-zeros in ELL (ER_ELL), a factor of power-law distribution (R), the estimated dense sub-block's row size (est_rs), and the estimated dense sub-block's column size (est_cs). Since the PAK infrastructure provides most of the machine-learning algorithms, we use the decision tree module to implement the producer. The tuned SpMV library by SMAT (Li et al. 2013;

```
#Extractor
ext=PAK.Extractor$new(
list(spmv_extrator=c(
"M","N","NNZ","Ndiags","NTdiags_ratio","ER_DIA")))

#Producer
pro1=PAK.Producer.Exhaustion$new(list(
method_name="'MV_COO','MV_CSR','MV_CSC','MV_DIA';"))

#Optimizer
opt=PAK.Optimizer$new(generator.name="smatlib")

#Evaluator
eva=PAK.Evaluator$new(sub.evaluators=list(
tau=list(P_WALL_CLOCK_TIME=function(x){
if(x>0) return (0-x) else return(0)})))

#Learner
pro2=PAK.Producer.DecisionTree$new()
pro2$trainModel(trainingData,"","method_name")
for(app in training.matrixs)
{
  tuning=PAK.Tuner$new(app=app,optimizer=opt,evaluator=eva,
  producer =pro1,need.store=TRUE)
  tuning$tune()
}

#Autotuning using the generated decision-tree model
for(app in test.matrixs)
{
  t=objectives[[app]]
  eva2=PAK.Evaluator$new(sub.evaluators=list(tau=list(
  P_WALL_CLOCK_TIME=function(x){
  if(x>t)return (t-x) else return(0)})))
  tuning=PAK.Tuner$new(app=app,extractor=ext,optimizer=opt,
  evaluator=eva,producer =pro2)
  tuning$tune()
}
```

Fig. 16. Implementation of SpMV autotuner with PAK programming interface.

Tan et al. 2018) is adopted as the optimizer. At last, we also use TAU (Malony et al. 1999) to evaluate the optimization results and update the knowledge database.

6.1 Experimental Setup

The experimental platforms are the same CPU and GPU as those used in the stencil autotuner. The PAK-based SpMV autotuner is compared with other counterpart autotuners for several typical sparse matrices, which are used in many related literatures (Li et al. 2013; Tan et al. 2018; Liu 2015; Kreutzer et al. 2014; Zhao et al. 2018).

6.1.1 SpMV Autotuners. Although there are many sparse matrices computation autotuners, most are developed based on obsolete architectures. Recently, Kreutzer et al. (Kreutzer et al. 2014) proposed a unified sparse matrix data format SELL-C-σ, which is oblivious to the architecture. They leverage performance engineering and models to achieve high performance SpMV

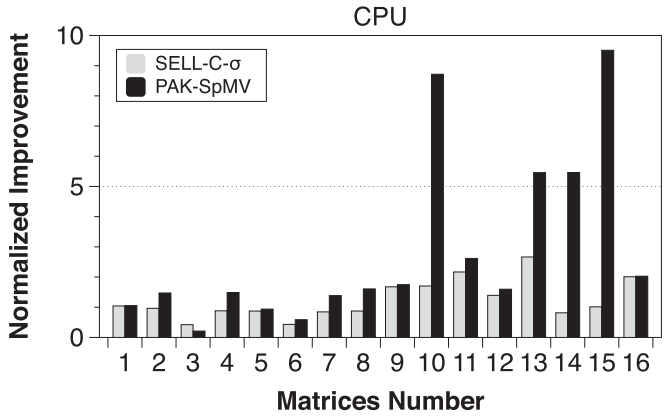
No.	Graph	Name	Dimensions	Nonzeros (NNZ / M)	Application area
1		pcrystk02	14K×14K	491K (35)	duplicate materials problem
2		denormal	89K×89K	623K (7)	counter-example
3		cryg10000	10K×10K	50K (5)	materials problem
4		apache1	81K×81K	311K (4)	structural problem
5		bfly	49K×49K	98K (2)	undirected graph sequence
6		whitaker3 _dual	19K×19K	29K (1)	2D/3D problem
7		ch7-9-b3	106K×18K	423K (4)	combinatorial problem
8		shar_te2-b2	200K×17K	601K (3)	combinatorial problem
9		pkustk14	152K×152K	15M (98)	structural problem
10		crankseg_2	64K×64K	14M (222)	structural problem
11		Ga3As3H12	61K×61K	6M (97)	theoretical/quantum chemistry
12		HV15R	2M×2M	283M (140)	computational fluid dynamics
13		europe_osm	51M×51M	108M (2)	undirected graph
14		D6-6	121K×24K	147K (1)	combinatorial problem
15		dictionary28	53K×53K	178K (3)	undirected graph
16		roadNet-CA	2M×2M	6M (3)	undirected graph

Fig. 17. Representative matrices.

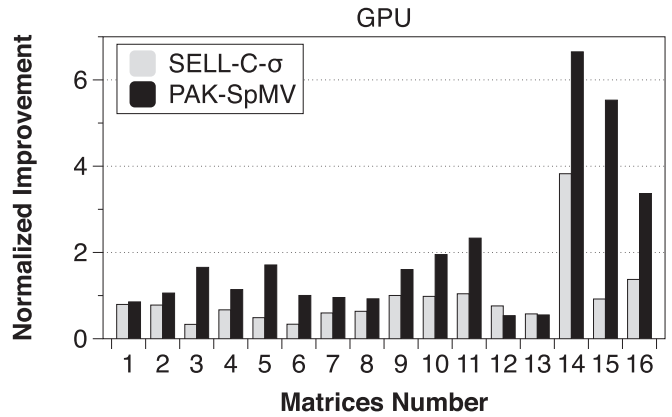
implementations on both CPU and manycore accelerators (GPU and MIC). We select SELL-C-σ to compare performances for several reasons:

— A remarkable feature of our SpMV autotuner is the automatic selection of the optimal format for a given sparse matrix. SELL-C-σ can switch between CSR and SELL-like formats, while other autotuners only focus on the best implementation of some format.
— SELL-C-σ performs extensive optimizations and is well suited for all modern, threaded architectures with SIMD/SIMT execution such as the Intel Xeon multicore processors and Nvidia Kepler.

6.1.2 Sparse Matrices. For the offline data mining component, its training dataset includes 2,055 sparse matrices, which are randomly chosen from the UF sparse matrix collection. By excluding the training matrices, we select 331 other matrices for performance evaluation. However, for the sake of plotting a more readable graph, we only present the experimental data with the visual graphs for 16 representatives in the following context. The 16 matrices are described in Figure 17.



(a) Performance improvement on CPU



(b) Performance improvement on GPU

Fig. 18. Comparison of performance on both CPU and GPU.

6.2 Performance

Because the vendors have tuned libraries on both CPU and GPU, we measure the SpMV performance of autotuners by normalizing them to the corresponding libraries, which include Intel MKL on multicore CPU and NVIDIA cuSPARSE on Kepler GPU, respectively.

CPU: Figure 18(a) plots the respective performance improvements normalized to Intel MKL using 16 threads. The X-axis represents the matrix number according to the order listed in Figure 17, and the Y-axis represents the speedups of SpMV generated by the two autotuners. PAK-SpMV outperforms Intel MKL for most matrices with a highest speedup reaching a factor of more than 9. For the three matrices (Nos. 3, 5, and 6), PAK-SpMV cannot perform better because the size of matrix or the number of non-zeros is relatively small. For example, the size of the matrix No. 3 is 0.38Mbyte for double precision format. Hence, the matrix No. 3 can be held in the cache of CPU and the distribution of the matrix is relatively regular. Intel MKL can directly achieve better performance while SELL-C-σ and PAK-SpMV have some overhead such as reordering the matrix for SELL-C-σ and choosing the optimal format for PAK-SpMV. Although SELL-C-σ achieves speedups for only half of the sparse matrices, it displays comparable performance with Intel MKL.

The highest performance is achieved on Nos. 10, 14, and 15 matrices because they have either a more explicit pattern or regular distributions of non-zeros. It is also apparent to observe that performance variation was greater than nine times. The significant variation indicates that it is worth adopting the autotuner in different applications, owing to its adaptivity to diverse sparse matrices. From our experiments, the 16 matrices achieve this high performance by using DIA, ELL, BCSR, COO, and BELL, respectively. This indicates a high performance SpMV library should be implemented with the acknowledgment of sparse structures (applications).

GPU: Figure 18(b) shows the performance improvement on NVIDIA Kepler GPU, which is in accordance with that on multi-core CPU. Compared to CPU, PAK-SpMV outperforms NVIDIA cuSPARSE for most matrices, while SELL-C-σ performs worse. An interesting observation is that the highest speedup is also achieved on Nos.14 and 15 matrices because of the explicit pattern with small dense blocks. Although SELL-C-σ is aware of the sparse format choice to some extent (e.g., CSR is the special format of SELL-C-σ), it still pertinent to pay more attention to traditional autotuning only focused on the architecture. By leveraging our previous work on tuning the sparse storage format, PAK-SpMV achieves the best performance in this presented work.

7 RELATED WORK

Numerous studies have reported successful autotuning to achieve high performance on modern architectures for diverse applications (Chen et al. 2005; Basu et al. 2013a; Li et al. 2017b). For instance, Hall et al. (Basu et al. 2013a) reviewed autotuning systems with respect to generality, managing overhead, and usability. Because PAK mainly addresses the usability issue in this work, we attempted to contrast our work with several key representatives promoting usability, while referring to Basu et al. (2013a) for a comprehensive survey of state-of-the-art autotuning techniques. With advancements in autotuning systems, efforts have been invested on domain-specific library/language, the adaptive framework, and performance knowledge to enhance usability.

Domain-Specific Library/Language: The domain-specific libraries and languages allow programmers to express computation at a high level and leverage code generation and autotuning to produce optimized codes. Undoubtedly, these autotuning systems are specially designed for some application algorithms, such as ATLAS (Whaley and Dongarra 1998) and PHiPAC (Bilmes et al. 1997) for dense matrix multiplication, FFTW (Frigo and Johnson 2005) and UHFFT (Ali et al. 2007) for FFT, and OSKI (Vuduc et al. 2005) and SMAT (Li et al. 2013; Tan et al. 2018) for sparse matrix kernels. In fact, these systems share a commonality of tuning space searching algorithms, which are encapsulated by the `Producer` interface in PAK. Thus, redundant engineering work can be eliminated when other libraries are developed. On the other hand, domain-specific languages have been recently used to hide the complexity of autotuning, especially for computations (i.e., stencil) that cannot be abstracted as libraries. PATUS (Matthias et al. 2011) and work by (Kamil et al. 2010) and (Hou et al. 2017) automatically generate autotuned code for stencil computations or a subset of geometric multigrids. They provide domain-specific language to users for writing optimization configurations, which facilitate searches in predefined optimization space. In addition, PAK-Stencil (Luo et al. 2015) and Pochoir (Tang et al. 2011) have defined domain-specific embedded languages for stencil computations in which the tuning strategies are based on an optimal-space model and cache oblivious algorithm. PetaBricks (Ansel et al. 2009) is a new implicitly parallel language and compiler that has multiple implementations for multiple algorithms in order to solve problems and it has an autotuning system and framework to find optimal choices. The limitation of PetaBricks (Ansel et al. 2009) is that users can only use optimizations that have been built into the autotuning systems and adding new optimizations will require substantial efforts. Using PAK eliminates such issues, whereby users only need to instantiate the APIs instead of re-architecting the whole autotuning system.

Adaptive Framework: Given the successes of autotuning, people have developed several frameworks to make autotuning a mainstream technology. Active Harmony (Ţăpuş et al. 2002; Tiwari et al. 2009) provides a framework for tuning configurable libraries and exploring different compiler optimizations. It acts as a search engine capable of rapidly exploring the parameter search space by testing multiple hypotheses in parallel. Another search engine framework is OpenTuner (Ansel et al. 2014), which allows many search techniques to work together and supports customizable configuration representations of a sophisticated search technique. CHiLL (Tiwari et al. 2009) is a code variant generator that allows the user to specify a series of high-level loop transformations to be applied together. However, these frameworks only focus on one phase of autotuning and have to be integrated into other autotuning systems. Orio (Hartono et al. 2009) achieves extensibility to some extent because it allows user-defined skeleton functions to test the performance of generated variants. Basu et al. (2017) explores the use of a compiler-based autotuning framework based on CUDA-CHiLL. On multigrid domain, Cy Chan et al. (2009) describes the techniques which allow the user to automatically generate tunned multigrid cycles of different shapes targeted to the user's specific combination of problem, hardware, and accuracy requirements. Surge (Muralidharan et al. 2016) is a nested data-parallel programming system designed to simplify the porting and tuning of parallel applications to multiple target architectures. GMG (Basu et al. 2013b) describes a compiler approach to introducing communication-avoiding optimizations in geometric multigrid. Protonu Basu et al. (Basu et al. 2017) applies CHiLL and CUDA-CHiLL to the operators of the miniGMG benchmark. PAK is a high-level abstraction that provides a protocol to encapsulate these tools.

Performance Knowledge: Although performance knowledge is important to perform tuning, it is commonly overlooked. Malony et al. (1999) developed a TAU (Malony et al. 1999) parallel performance system suite of tools, which creates a robust, flexible, portable, and integrated framework and toolset for performance instrumentation, measurement, analysis, and visualization of large-scale parallel computer systems and applications. It also provides a database to store knowledge about performance, but only collects fixed knowledge about some specific performance metric and has nothing to do with the other modules in autotuning. In fact, recent literature by Malony et al. (1999) presents a case study of integrating TAU with other autotuning frameworks, including Active Harmony, CHiLL, and Orio. Inspired by their work, PAK is a major step forward in the progress of encapsulation, in which it is used as a mechanism to automate integration and make it more extensible.

8 CONCLUSION

Although there is a consensus that autotuning is a critical strategy to achieve high performance as Exascale computing is approaching, more advancements need to be made to initiate autotuning as a mainstream technology that is more effective and available to a broader class of users. One of the barriers in doing so is the low usability of current autotuning techniques. In this article, we generalize an autotuning protocol (PAK) of five procedures corresponding to an extractor, producer, optimizer, evaluator, and learner. Based on this protocol, a general performance tuning and knowledge-managing infrastructure was developed to construct autotuners with high usability. PAK enables the composition of abstraction layers in the autotuning system and is able to exploit whatever knowledge a user provides. We present two cases using PAK to construct autotuners for stencil and sparse matrix computation to test its efficiency and performance. Compared to the traditional approach of developing autotuners, PAK provides a modularized autotuning-programming interface that allows users to rapidly assemble an efficient autotuner by seamlessly leveraging performance knowledge mining.

ACKNOWLEDGMENTS

We would like to express our gratitude to all reviewers' for their constructive comments, helping us to polish this article.

REFERENCES

Ayaz Ali, Lennart Johnsson, and Jaspal Subhlok. 2007. Scheduling FFT computation on SMP and multicore systems. In *Proceedings of the 21st Annual International Conference on Supercomputing*. ACM, 293–301.

Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. 2009. PetaBricks: A language and compiler for algorithmic choice. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Vol. 44. ACM.

Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*. http://groups.csail.mit.edu/commit/papers/2014/ansel-pact14-opentuner.pdf

P. Basu, M. Hall, M. Khan, S. Maindola, S. Muralidharan, S. Ramalingam, A. Rivera, M. Shantharam, and A. Venkat. 2013a. Towards making autotuning mainstream. *The International Journal of High Performance Computing Applications* 27, 4 (2013), 379–393.

P. Basu, A. Venkat, M. Hall, S. Williams, B. Van Straalen, and L. Oliker. 2013b. Compiler generation and autotuning of communication-avoiding operators for geometric multigrid. In *Proceedings of the 20th Annual International Conference on High Performance Computing*. 452–461.

Protonu Basu, Samuel Williams, Brian Van Straalen, Leonid Oliker, Phillip Colella, and Mary Hall. 2017. Compiler-based code generation and autotuning for geometric multigrid on GPU-accelerated supercomputers. *Parallel Computing* 64 (2017), 50–64. http://www.sciencedirect.com/science/article/pii/S0167819117300376 High-End Computing for Next-Generation Scientific Discovery.

Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. 1997. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In *Proceedings of the 11th International Conference on Supercomputing*. ACM, 340–347.

Cy Chan, Jason Ansel, Yee Lok Wong, Saman Amarasinghe, and Alan Edelman. 2009. Autotuning multigrid with PetaBricks. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC'09)*. ACM, New York, Article 5, 12 pages.

Chun Chen, Jacqueline Chame, and Mary Hall. 2005. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *International Symposium on Code Generation and Optimization*. IEEE, 111–122.

Jee W. Choi, Amik Singh, and Richard W. Vuduc. 2010. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Vol. 45. ACM, 115–126.

Matthias Christen, Olaf Schenk, and Yifeng Cui. 2012. Patus for convenient high-performance stencils: Evaluation in earthquake simulations. In *Proceedings of the 2012 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'12)*. IEEE, 1–10.

Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. 1999. Optimizing for reduced code space using genetic algorithms. *ACM SIGPLAN Notices* 34, 7 (1999), 1–9.

Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. 2009. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Review* 51, 1 (2009), 129–159.

Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. 2008. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. IEEE Press, 4.

Matteo Frigo and Steven G. Johnson. 2005. The design and implementation of FFTW3. *Proceedings of the IEEE* 93, 2 (2005), 216–231.

Archana Ganapathi, Kaushik Datta, Armando Fox, and David Patterson. 2009. A case for machine learning to optimize multicore performance. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Parallelism (HotPar'09)*.

Albert Hartono, Boyana Norris, and Ponnuswamy Sadayappan. 2009. Annotation-based empirical performance tuning using Orio. In *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing (IPDPS'09)*. IEEE, 1–11.

Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. 2013. A stencil compiler for short-vector simd architectures. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*. ACM, 13–24.

R. Himeno. 2011. Himeno benchmark. Retrieved from http://accc.riken.jp/2444.htm.

K. Hou, H. Wang, and W. C. Feng. 2016. AAlign: A SIMD framework for pairwise sequence alignment on x86-based multi- and many-core processors. In *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS'16)*. 780–789.

Kaixi Hou, Hao Wang, and Wu-chun Feng. 2017. GPU-UniCache: Automatic code generation of spatial blocking for stencils on GPUs. In *Proceedings of the Computing Frontiers Conference (CF'17)*. ACM, New York, 107–116.

K. Hou, H. Wang, and W. C. Feng. 2018. A framework for the automatic vectorization of parallel sort on x86-based processors. *IEEE Transactions on Parallel and Distributed Systems* 29, 5 (2018), 958–972.

Shoaib Kamil, Cy Chan, Leonid Oliker, John Shalf, and Samuel Williams. 2010. An auto-tuning framework for parallel multicore stencil computations. In *Proceedings of the 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS'10)*. IEEE, 1–12.

Malik Khan, Protonu Basu, Gabe Rudy, Mary Hall, Chun Chen, and Jacqueline Chame. 2013. A script-based autotuning compiler system to generate high-performance CUDA code. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 31.

Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop. 2014. A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units. *SIAM Journal on Scientific Computing* 36, 5 (2014), C401–C423.

Karl S. Kunz and Raymond J. Luebbers. 1993. *The Finite Difference Time Domain Method for Electromagnetics*. CRC Press.

Ang Li, Weifeng Liu, Mads R. B. Kristensen, Brian Vinter, Hao Wang, Kaixi Hou, Andres Marquez, and Shuaiwen Leon Song. 2017b. Exploring and analyzing the real impact of modern on-package memory on HPC scientific kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'17)*. 26:1–26:14.

J. Li, J. Choi, I. Perros, J. Sun, and R. Vuduc. 2017a. Model-driven sparse CP decomposition for higher-order tensors. In *Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS'17)*. 1048–1057.

Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. 2013. SMAT: An input adaptive auto-tuner for sparse matrix-vector multiplication. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 117–126.

Junhong Liu, Xin He, Weifeng Liu, and Guangming Tan. 2018. Register-based implementation of the sparse general matrix-matrix multiplication on GPUs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'18)*. ACM, New York, 407–408.

Junhong Liu, Guangming Tan, Yulong Luo, Zeyao Mo, and Ninghui Sun. 2015. PAK. Retrieved from https://github.com/PAA-NCIC/PAK.

Weifeng Liu. 2015. *Parallel and Scalable Sparse Basic Linear Algebra Subprograms*. Ph.D. dissertation. University of Copenhagen.

Weifeng Liu, Ang Li, Jonathan D. Hogg, Iain S. Duff, and Brian Vinter. 2017. Fast synchronization-free algorithms for parallel sparse triangular solves with multiple right-hand sides. *Concurrency and Computation: Practice and Experience* 29, 21 (2017), e4244–n/a.

Weifeng Liu and Brian Vinter. 2015a. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM International Conference on Supercomputing (ICS'15)*. ACM, 339–350.

Weifeng Liu and Brian Vinter. 2015b. A framework for general sparse matrix-matrix multiplication on GPUs and heterogeneous processors. *Journal of Parallel and Distributed Computing* 85, C (Nov. 2015), 47–61.

Yulong Luo, Guangming Tan, Zeyao Mo, and Ninghui Sun. 2015. FAST: A fast stencil autotuning framework based on an optimal-solution space model. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS'15)*. ACM, New York, 187–196.

Thibaut Lutz, Christian Fensch, and Murray Cole. 2013. Partans: An autotuning framework for stencil computation on multi-gpu systems. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 59.

Allen D. Malony, Jan Cuny, and Sameer Shende. 1999. *TAU: Tuning and Analysis Utilities*. Technical Report. LALP-99–205. Los Alamos National Laboratory Publication.

Azamat Mametjanov, Daniel Lowell, Ching-Chen Ma, and Boyana Norris. 2012. Autotuning stencil-based computations on GPUs. In *Proceedings of the 2012 IEEE International Conference on Cluster Computing (CLUSTER'12)*. IEEE, 266–274.

Naoya Maruyama, Tatsuo Nomura, Kento Sato, and Satoshi Matsuoka. 2011. Physis: An implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*. IEEE, 1–12.

Christen Matthias, Schenk Olaf, and Burkhart Helmar. 2011. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS'11)*. IEEE, 676–687.

Jiayuan Meng and Kevin Skadron. 2009. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In *Proceedings of the 23rd International Conference on Supercomputing*. ACM, 256–265.

Paulius Micikevicius. 2009. 3D finite difference computation on GPUs using CUDA. In *Proceedings of the 2nd Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 79–84.

Saurav Muralidharan, Michael Garland, Albert Sidelnik, and Mary Hall. 2016. Designing a tunable nested data-parallel programming system. *ACM Transactions on Architecture and Code Optimization* 13, 447 (Dec. 2016), Article, 24 pages.

Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 2010. 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 1–13.

Apan Qasem, Michael Jason Cade, and Dan Tamir. 2012. Improved energy efficiency for multithreaded kernels through model-based autotuning. In *Proceedings of the 2012 IEEE Green Technologies Conference*. IEEE, 1–6.

Guangming Tan, Junhong Liu, and Jiajia Li. 2018. Design and implementation of adaptive SpMV library for multicore and manycore architecture. *ACM Transactions on Mathematical Software* 44, 4 (Aug. 2018), Article 46.

Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. 2011. The pochoir stencil compiler. In *Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 117–128.

Cristian Ţăpuş, I-Hsin Chung, Jeffrey K. Hollingsworth, et al. 2002. Active harmony: Towards automated performance tuning. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society Press, 1–11.

A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth. 2009. A scalable autotuning framework for compiler optimization. In *Proceedings of the International Parallel and Distributed Processing Symposium*.

Didem Unat, Xing Cai, and Scott B. Baden. 2011. Mint: Realizing CUDA performance in 3D stencil methods with annotated C. In *Proceedings of the International Conference on Supercomputing*. ACM, 214–224.

Richard Vuduc, James W. Demmel, and Katherine A. Yelick. 2005. OSKI: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, Vol. 16. IOP Publishing, 521.

Hao Wang, Weifeng Liu, Kaixi Hou, and Wu-chun Feng. 2016. Parallel transposition of sparse data structures. In *Proceedings of the 2016 International Conference on Supercomputing (ICS'16)*. 33:1–33:13.

H. Wang, J. Zhang, D. Zhang, S. Pumma, and W. C. Feng. 2017. PaPar: A parallel data partitioning framework for big data applications. In *Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS'17)*. 605–614.

Xinliang Wang, Weifeng Liu, Wei Xue, and Li Wu. 2018. swSpTRSV: A fast sparse triangular solve with sparse level tile layout on sunway architectures. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'18)*. ACM, New York, 338–353.

R. Clint Whaley and Jack J. Dongarra. 1998. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society, 1–27.

Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM* 52, 4 (2009), 65–76.

Yue Zhao, Jiajia Li, Chunhua Liao, and Xipeng Shen. 2018. Bridging the gap between deep learning and sparse matrix format selection. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'18)*. ACM, New York, 94–108.

Received January 2017; revised January 2018; accepted June 2018