FAST: A Fast Stencil Autotuning Framework Based on an Optimal-solution Space Model

Yulong Luo†, Guangming Tan†, Zeyao Mo‡, Ninghui Sun†
†State Key Laboratory of Computer Architecture, Institute of Computing Technology
‡Institute of Applied Physics and Computational Mathematics
Beijing, China
{luoyulong,tgm,snh}@ncic.ac.cn, zeyao_ mo@iapcm.ac.cn

ABSTRACT

Stencil computations comprise an important class of kernels in many scientific computing applications. As the diversity of both architectures and programming models grow, autotuning is emerging as a critical strategy for achieving portable performance across a broad range of execution contexts for stencil computations. However, costly tuning overhead is a major obstacle to its popularity. In this work, we propose a fast stencil autotuning framework FAST based on an Optimal-Solution Space (OSS) model to significantly improve tuning speed. It leverages a feature extractor that comprehensively characterizes stencil computation. Using the extracted features, FAST constructs an OSS database to train an off-line model which provides an on-line prediction. We evaluate FAST with five important stencil computation applications on both an Intel Xeon multicore CPU and an NVIDIA Tesla K20c GPU. Compared with state-of-the-art stencil autotuners like Patus and SDSL, FAST improves autotuning speed by $10-2697$ times without any user annotation, while achieving comparable performance.

Categories and Subject Descriptors

D.1.3 [**Programming techniques**]: Concurrent programming—*Parallel Programming*; D.3.4 [**Programming Languages**]: Processors—*Code Generation, Compilers*

General Terms

Algorithms, Performance, Languages

Keywords

Stencil; Autotuning; OSS

1. INTRODUCTION

Stencil computations are of critical importance for scientific computing, engineering, image processing, particle simulation, and geometric modeling. For example, grand challenge applications including climate, weather, and ocean modeling use explicit time-integration methods of partial differential equations that require Exascale stencil computations. Other important examples include computational electromagnetics and quantum dynamics codes using the finite-difference time-domain (FDTD) method, medical and image-processing applications that perform smoothing and other neighbor pixel-based computations, and certain cellular automata and seismic simulations. Due to the importance of stencil computations, it has resulted in extensive programming model research efforts, as well as architectural support.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICS'15, June 8–11, 2015, Newport Beach, CA, USA.
Copyright © 2015 ACM 978-1-4503-3559-1/15/06 ...$15.00.
http://dx.doi.org/10.1145/2751205.2751214.

Table 1: The reported autotuning speed in existing stencil autotuning systems.

Autotuner	Search Time	Search Strategy
PATUS [35]	8 hours	Greedy
SDSL [20]	>33 hours	Exhaustive
Halide [41]	2 hours - 2 days	Stochastic
PARTANS [31]	2.5 hours - 32 days	Hill climbing

The advent of multi/many-core processors brings more complexity and diversity in architectures and programming models. As a consequence, it increases the difficulty of developing high performance programs with reasonable efficiency. Recently, a critical technology to deal with the significant changes is autotuning [4] that acts as either an auto-tuning library [51, 50] or adaptive performance tuning framework [12, 35, 7] for some specific domain. In general, autotuning explores a search space of possible optimization solutions, which reflect functional equivalence but employ different algorithms or implementations, and optimization parameters. A core problem addressed by recent technical approaches is to tune optimization strategies considering program and execution context together. The term ***running instance*** is used within autotuning to refer to the combination of program code, input, target platform, and back-end compiler. It encapsulates the program code and execution context as a whole, and directly determines actual performance of an application.

Two major approaches to autotuning are empirical search-based and prediction-based. Search-based autotuning is a universal but time consuming approach which measures the runtime of all the candidate optimization solutions. For most stencil autotuners, the search-based approach is employed to select a suitable optimization solution from a huge optimization space. A number of pruning strategies and search heuristics are adopted to reduce optimization space and speed up the tuning process[43, 35, 12, 31, 27, 10, 11, 40,

8, 52, 36]. In addition, analytical models derived by composing analysis of code with architectural models are also used to reduce an optimization space to more profitable solutions. However, for some complicated tuning problems, these techniques may still result in unacceptable overhead. Table 1 surveys the autotuning search time reported by existing stencil autotuning systems. In contrast to search-based approaches, prediction-based autotuning gives suitable optimization solutions directly and shows very low overhead, relatively. This approach relies on building a machine learning model that maps from a feature space to an optimal solution space and is usually applied to algorithm selection. However, for stencil autotuning, there are three challenges preventing prediction-based techniques from being applied: i) the optimal optimization solutions are affected by the input data, platform, and stencil itself, and vary a lot when these features are changing. ii) the number of optimization solutions is extremely large, and a prohibitive amount of training data is required. iii) the performance gap between the optimal and a near-optimal solution is not always distinct, resulting in low accuracy of the model.

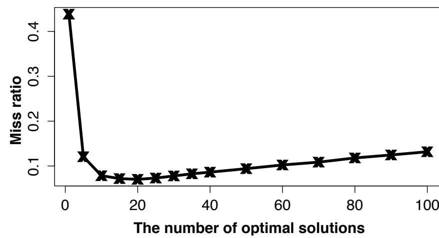

Figure 1: The miss radio of this machine learning model is related to the number of predicted optimal solutions.

In this paper, we propose a *Framework based on optimAl-solution Space modeling for sTencil autotuning* (FAST), integrating techniques of static analysis, domain-specific languages, and machine learning methods to overcome these challenges above. First, FAST adopts static analysis and domain-specific techniques to extract inherent features from target running instances to assist autotuning. Second, a self-learning database is set up to collect training data during either training or tuning phases, lowering the requirement for initial training data. Furthermore, Figure 1 presents the average miss ratio of a machine learning model that predicts a set of optimal solutions, where a smaller miss ratio means higher accuracy. When the number of predicted optimal solutions equals 1, it indicates an approach directly predicting the best solution, and the miss ratio is about 50%. When the number of predicted optimal solutions grows, the miss ratio decreases until the number is 25, where a minimum point is reached. Instead of predicting the best solution directly, it is more feasible to predict a set of optimal solutions (or near-optimal). According to this observation, FAST uses a prediction-based approach to generate highly-tuned solutions by predicting a set of optimal solutions (refer to as an ***optimal-solution space***), thereby it significantly improves the tuning speed.

More specifically, we make the following contributions:

- We develop a feature extractor that extracts inherent features of a running instance, including aspects of algorithm, architecture and input, to characterize stencil computation. Afterwards, an embedded domain-specific language for stencil is proposed to facilitate feature extraction, which is implemented using the ROSE compiler infrastructure (Section 4).
- We formulate a machine learning model to correlate features with an optimal-solution space. The core of this model is that the OSSs of two running instances are similar if their features are of high similarity (Section 5).
- We set up a self-learning database to train the model and generate OSSs. It automatically records optimization recipes and constructs OSSs during either training or tuning phases (Section 5).
- We implement a code generator that supports C/Fortran/CUDA back-end code generation and optimization. It performs code translation and transformation according to the back-end compiler and selected optimization solution (Section 6).

Compared with state-of-the-art stencil autotuning systems like Patus and SDSL, FAST accomplishes the autotuning process in minutes and the tuning step is reduced by $10-2697$ times without any user annotations while achieving comparable performance.

2. MOTIVATION

A stencil is a geometric structure and defines the value of a grid point $v(R)$ in a d-dimensional spatial grid R at time t as a function of neighboring grid points at recent times before t. A stencil computation sweeps over R to update $v(R)$ as a mapping function f of its neighbors $v(R')$, where $R' \in \mathit{neighbor}(R)$. It computes the stencil repeatedly for each grid point over many time steps. Stencil order H is defined as the distance between the central grid point R and the farthest grid points R' in $\mathit{neighbor}(R)$ along a certain axis. Stencil size N is defined as the cardinality $|\mathit{neighbor}(R)|$, that is the number of grid points R' in $\mathit{neighbor}(R)$ including R itself. The definition of a stencil computation doesn't specify how the map function f exactly performs, its concrete implementation depends on a physical model in practice. Therefore, it is impossible to define a common library like BLAS routines for stencil computation. In other words, optimization solutions applied to stencil computations are either input-sensitive or application-specific.

The design principle of our autotuning framework is motived by observations on the relationship between algorithm features and optimization solutions. Figure 2 demonstrates an example of four stencil computation kernels in two groups. Intuitively, it is obvious that the difference between $kernel1$ and $kernel2$ in $GroupA$ is less than that between $kernel3$ and $kernel4$ in $GroupB$. In fact, we can characterize their algorithm features using some quantitative metrics. As an illustrative instance, we simply use several metrics including *array number*(AN), *flop density*(FD), and *reference density*(RD), to characterize the four kernels. For this example, it is easy to calculate these metrics by hand. The values are summarized in the tables of Figure 2. The differences of all metrics in $GroupA$ are extremely low while the difference in $GroupB$ are much higher. A straightforward conclusion from the feature difference is that there is a high probability of applying the same optimization solutions to all kernels in $GroupA$, but different optimization solutions should be applied to each kernel in $GroupB$.

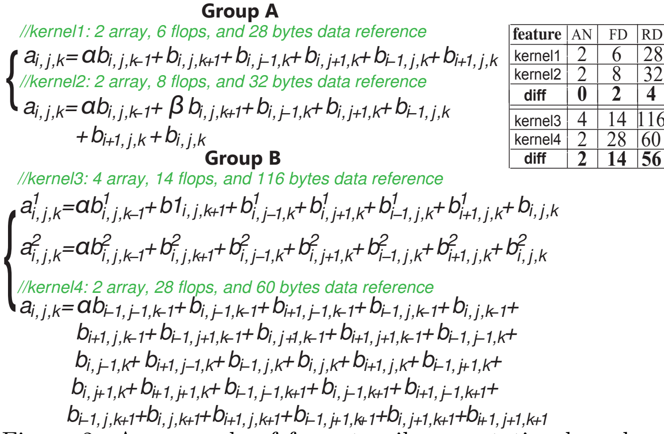

Figure 2: An example of four stencil computation kernels and several typical computing features. AN: the number of arrays. FD: the number of floating-point operations per stencil point. RD: the number of data references per stencil point.

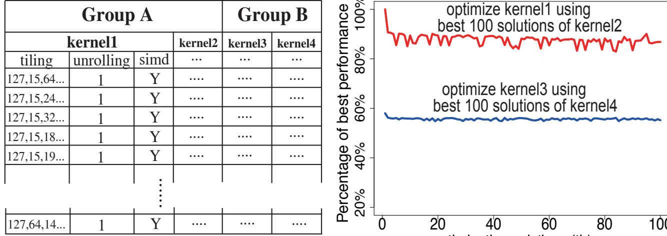

Figure 3: An illustration of the relationship between optimization solution and similarity of computing feature for the two groups of stencil kernels.

In our experiments, we first construct the best 100 optimization solutions for $kernel2$ and $kernel4$ by brute-force search, then apply these optimization solutions to $kernel1$ and $kernel3$, respectively. For this motivating example we adopt three optimization strategies— *tiling*, *unrolling*, and *SIMD* to search optimal solutions (the table in the left of Figure 3). The right picture in Figure 3 plots performance normalized to the best solution [1] for $kernel1$ and $kernel3$. For $GroupA$, $kernel1$ reaches more than 90% of its best performance when using the same optimization solutions as $kernel2$. For $GroupB$, $kernel3$ reaches only 60% of its best performance when using the same optimization with $kernel4$. The experimental result indicates a key attribute where two stencil computations share the same (near-)optimal solutions if they have high similarity in computing features. Further, an autotuning system can leverage this attribute to quickly find optimal solutions based on a priori knowledge instead of a time-consuming empirical search-based approach.

3. AUTOTUNING FRAMEWORK

Figure 4 highlights the major constituent parts of FAST, including (1) a feature extractor that characterizes inherent features of an incoming running instance with the support of a domain specific language, (2) an OSS module used to find optimal solutions that is comprised of a model and a database, (3) a code generator applying translation and optimization according to the back-end compiler and selected optimization solution. We give a brief introduction to each component by going through the autotuning process.

[1]It's achieved by brute-force search.

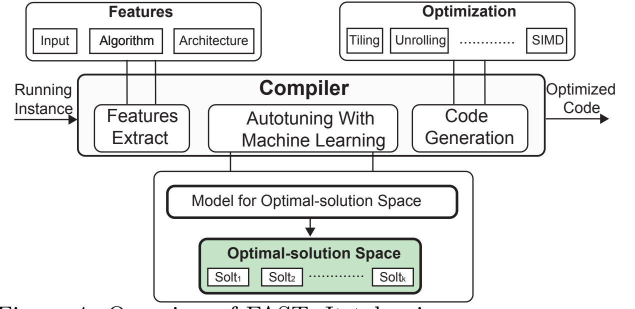

Figure 4: Overview of FAST. It takes in a program source written in domain-specific language, input data and architecture description, outputs an optimized program binary for target architecture.

At the first step, the feature extractor performs an analysis on an incoming running instance comprised of input data, program code and architectural description. In order to comprehensively characterize a running instance, we develop an embedded domain-specific language (eDSL) for stencil computations. The extracted features are then formed as a feature vector.

With the extracted feature vector, the next step is to obtain the optimal solution. FAST first sets up a self-learning database that saves optimization recipes and forms an OSS. Then it builds a model that correlates the behavior of a new running instance with previous knowledge. The model takes the difference of the feature vector from the new running instance and each saved feature vector as the input and predicts the similarity of their OSSs. With the model, an OSS that has maximum similarity with that of the incoming case is picked out from the OSS database. According to a preset threshold, the chosen OSS is validated through an empirical approach, and the first optimization solution satisfying the threshold is returned as the result.

Finally, the code generator performs source-to-source translation according to the detected programming model and the resulting optimization solution. This step is comprised of two phases. In the first phase, a naive implementation without optimization is generated in the form of a high-level language (C or CUDA) according to the back-end compiler. In the second phase, optimization strategies are applied to the naive implementation based on the resulting optimization solution, and binary code is generated for a given execution context.

4. FEATURE ANALYSIS

4.1 Features

The feature extractor focuses on three feature types— *architecture*, *algorithm* and *input*. Table 2 summarizes these features. The architecture features include several basic configurations like processor core (e.g., frequency and vector length.) and memory hierarchy (e.g., cache/shared memory, bandwidth). The programming model is also considered as an architecture feature, where tuning is a combination of compiler options in a back-end compiler [13, 35, 3]. Although the target of our autotuning system is stencil computations, the algorithm feature is independent of stencils and

Table 2: Features extracted by FAST.

Feature Name	Optimization solution
Architecture	
Cores number	parallelism
Frequency	computing upper bound
L1/L2/L3 cache size	blocking
Shared memory (CUDA)	blocking & parallelism
Global memory	data layout
Register number	unrolling & parallelism
Memory bandwidth	blocking
Memory access latency	parallelism
Programing model	compiler options
Algorithm	
Operational intensity	parallelism
Flop density	instruction pipeline
Reference density	unrolling & blocking & texture (CUDA)
Array number	blocking & data layout
Loop radius	blocking & data reuse
Active cache lines	data reuse
Data type	data alignment & blocking
Input	
Problem size	all
Problem shape	loop transformation
Step/iteration time	tradeoff optimization overhead

can be used to characterize any program or problem. These algorithm features mixed with the architecture and input features are used to explore optimization solutions listed in Table 4. For example, the loop radius together with cache parameters is related to cache blocking. The input features are specific to stencil computations. These feature types include problem size, shape and iterations. To some extent, the problem size is related to all optimization solutions.

4.2 Compiler Support

In order to facilitate feature extraction, we design and implement a stencil embedded domain-specific language (eDSL) that is similar to other stencil DSLs [35, 20, 43, 34]. It is a minimal extension of the C/C++/Fortran high-level language, which expresses stencil domain information, such as the grid array referenced in sweeping, and the domain area that determines the range of sweeping. The extension structures are listed in Table 3, and we give a example of 3D 7-point in Figure 5.

```
1 void 7P_stencil(float*** a,float*** b,float alpha,float beta,int t)
2 { RegisterGridData (a,N,N,N);
3   RegisterGridData (b,N,N,N);
4   RegisterDomain ("dom",1,N−1,1,N−1,1,N−1);
5   STENCIL("dom",t)
6     b[0][0][0]=(a[0][0][−1]+a[0][0][1]+a[0][−1][0]+a[0][1][0]+
7                 a[−1][0][0]+a[1][0][0])*beta+a[0][0][0]*alpha;
8 }
```

Figure 5: An example of 3D 7-point stencil.

During feature extraction, the DSL code is first translated to an AST (Abstraction Syntax Tree) through front-end compiling. The functions *RegisterGridData()* and *RegisterDomain()* in the AST are identified to construct internal objects of **GridData** and **Domain**. Each internal object is inserted into a global table to finish "registration". The structure *STENCIL(domain,T)* is transformed to a **stencil** object using the previous global table, which represents a stencil computation. Then, a feature analysis pass is performed on **stencil** and outputs formatted feature data.

Table 3: The eDSL extends C/C++/Fortran with two data types, two intrinsic functions and one special structure to facilitate stencil representation and feature extraction.

Specification	Description
GridData	It wraps the array address, dimension and dimensionality into a stencil data object, and enables the compiler to do array index overflow checks and code generation on multiple platforms.
Domain	It expresses a grid area that the stencil computation is sweeping on and wraps N pairs of boundary arguments that specify the upper boundary and the lower boundary for each dimension of the grid area.
RegisterGridData (source,n$_1$...)	It registers an array as a **GridData** object to be identified in the **STENCIL** body. The function has $N+1$ arguments, where the first one determines the data source of **GridData**, and the next N arguments describe the shape of the n-dimensional grid.
RegisterDomain (id,l$_1$,u$_1$...)	It registers a grid area as a **Domain** object with a unique id that identifies the **Domain** object and $2*N$ arguments to describe the shape of the sweeping domain for their upper boundaries and lower boundaries.
STENCIL (domain,T) {..}	It is composed of three parts: (i) a unique **Domain**; (ii) a integer variable T representing the number of repeated stencil computations; (iii) the shape of the stencil defining the actual computations. A language restriction is that every referenced point is an element of one registered **GridData**. The array indices represent offsets away from the central point of the stencil.

5. OPTIMAL-SOLUTION SPACE MODULE

5.1 Optimal-Solution Space

Formally, given an optimization space $R = \{v_1, v_2, ..., v_N\}$ of N optimization solutions v_i ($i = 1..N$), let $M(v_i)$ be a metric to evaluate an optimization solution v_i. An *optimal-solution space* containing K solutions can be expressed as:

$$OSS^K = \{v|M(v) \geq M(v_K^*), v, v_K^* \in R\} \quad (1)$$

where v_K^* is the *K-th* best optimization solution in R and each v in OSS^K is not worse than v_k^*. In other words, OSS^K is a subset containing the best K optimization solutions. Further, we define a notion of *Overlapping Ratio* OR to measure the similarity of two OSS^Ks:

$$OR = \frac{|OSS_x^K \cap OSS_y^K|}{K} \quad (2)$$

It represents the ratio of the similar part between two OSSs. Our aim is to build a module that provides a mechanism leveraging feature vector $\vec{f}$ =<*architecture, algorithm, input*> to generate an OSS. We approach this problem by learning mapping (Section 5.2) from the feature vector difference of two running instance, $\vec{x} = \Delta\vec{f}$, to a probability distribution over OR of their OSS, $y = OR$. Once this distribution has been trained, to predict the OSS for a new running instance is achieved by traversing a OSS database (Section 5.3) and returning an OSS with a maximum predicted overlap ratio OR^*,

$$OR^* = \underset{x}{\operatorname{argmax}}\, q(y|\vec{x})$$

To provide a design principle to this machine learning model, we use a set of analytical experiments to study three key attributes of the OSS, including:

Overlapping ratio: We conduct experiments on 288 run-

ning instances (details in Section7.1) on both CPU and GPU platforms and achieve their OSSs by evaluating and ranking the optimization solutions listed in Table 4. Figure 6 presents the average ORs of these OSSs, and the x-axis presents their sizes (K). On both platforms, the average ORs drop when K decreases. For the CPU instance, when the size is 2100, the average OR is 78%. When the size becomes 100, the average OR drops to 20%. The GPU platform is the same. This result indicates a straightforward observation that *larger OSSs have higher OR they share more optimal (near-optimal) solutions with each other*. However, this leads to a dilemma that a larger OSS requires higher overhead for its empirical validation.

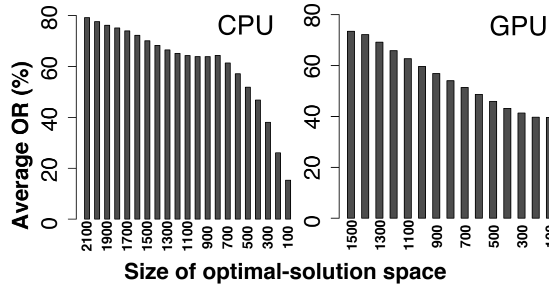

Figure 6: Relationship between OR and the size of OSSs.

Performance coverage: In an OSS^K, the performance lower bound is determined by the K-*th* best optimization solution p_K,

$$p_{lower}(OSS^K) = p_K$$

To study the performance coverage of OSSs, we construct the OSSs with different sizes (K) for 288 running instances and evaluate their average performance lower bound $\mathcal{D}_K$ by computing,

$$\mathcal{D}_K = \frac{\sum_{m=1}^{N} \frac{p_{m_K}}{p_m^*}}{N}$$

For a running instance m, p_m^* is its best optimization solution, and p_{m_K} is its K-*th* best optimization solution. We normalize the p_{m_k} value to p_m^*, and calculate the average value. As Figure 7 shows, $\mathcal{D}_K$ slowly degrades when K grows. More specifically, the $\mathcal{D}_K$ decreases 10% in performance compared to the best optimization solutions on the CPU when K is 100. On the GPU, a 10% performance slowdown occurs when K is 20. This observation indicates that *a small OSS covers most of the solutions with the highest performance*.

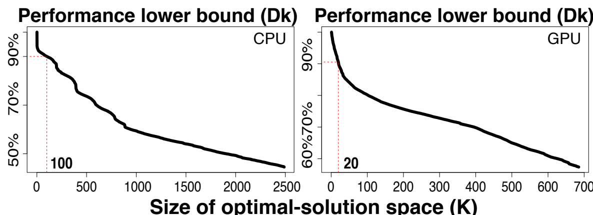

Figure 7: Performance lower bound of OSSs with different size.

Sensitivity to features: For two running instances with different features, their OSSs are different. To evaluate sensitivity of OSS to feature difference, we conduct two experiments by varying features of *flop density* and *problem size*, respectively. In each experiment, one feature is fixed and the other one varies. We measure the variability of features in each group by computing their coefficient of variation:

$$c_v = \frac{\sigma}{\mu} = \frac{\sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2}}{\overline{x}}$$

c_v is a normalized measure of the dispersion of a probability distribution. It is defined as the ratio of the standard deviation σ to the mean μ. Figure 8 presents the average OR for four groups of kernels with c_v from 80% to 3%. The result shows that when the coefficient of variation is smaller, the OR between two OSSs is higher. This observation indicates that *the OSSs of two running instances have a high OR if their features are of high similarity*. Therefore, it is reasonable that we can achieve an OSS having high OR with incoming running instances using a recorded running instance that has similar features.

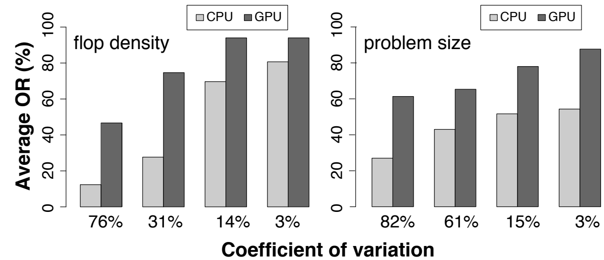

Figure 8: Averaged OR of OSS^{100}s for the four groups of example kernels. Stencils in each group have different features of *flop density*(left) and *problem size*(right) , and use c_v to denote extent of variability.

5.2 Build a Model

Based on the previous observations, we construct an optimal-space model inputting feature vector difference of two running instances, $\vec{x} = \Delta\vec{f}$, and predicting a probability distribution over OR of their OSS, $y = OR$. We let the feature vector difference $\vec{x}$ be the result of subtracting two given feature vectors,

$$x_i = \begin{cases} f_i - f_i' & numeric \\ 0/1 & non-numeric \end{cases}$$

where x_i is vector components of $\vec{x}$. For the numeric feature f_i, we perform a subtraction on them. For non-numeric f_i, x_i is 1 if the two features are the same. Otherwise x_i is 0. For each vector component x_i, the difference of two running instances in the i-*th* feature is a predictor variable and the OR is a response variable. To fit a machine learning model, we make use of the training records stored in a relational database to generate a training set $< \mathcal{X}, \mathcal{Y} >$ where each vector $\vec{x} \in \mathcal{X}$ is a feature vector difference $(x_1, x_2.., x_D)$ and each target $y \in \mathcal{Y}$ denotes the OR of the OSS of two running instances. Because there is more than one predictor variable in our model, we have used multiple linear regression and assumed the form of the model as:

$$y = \sum_{i=1}^{D} \mathcal{F}(x_i)$$

D is the dimensionality of the feature vector difference, and it equals to number of features. For each vector component x_i, the $\mathcal{F}(x_i)$ denotes the effect of the feature difference x_i on OR. Considering the nonlinear relationship between feature

vector differences and OR, we employ a polynomial regression and represent each $\mathcal{F}(x_i)$ as a three-order regression function:

$$\mathcal{F}(x_i) = \alpha_i \Delta x_i + \beta_i \Delta x_i^2 + \gamma_i \Delta x_i^3 + \delta_i$$

The α_i, β_i, γ_i are partial regression coefficients of x_i, and δ_i is its intercept.

$$y = \sum_{i=1}^{D} (\alpha_i \Delta x_i + \beta_i \Delta x_i^2 + \gamma_i \Delta x_i^3 + \delta_i)$$

It is easy to build the model with a solver in the R language, assuming that $\hat{\alpha}$, $\hat{\beta}$, $\hat{\gamma}$ are estimations for the parameters, and $\epsilon_i = y' - y$ are residuals between estimations and real values. We can get $\hat{\alpha}$, $\hat{\beta}$, $\hat{\gamma}$ by minimizing residuals for each ϵ_i.

5.3 OSS Database

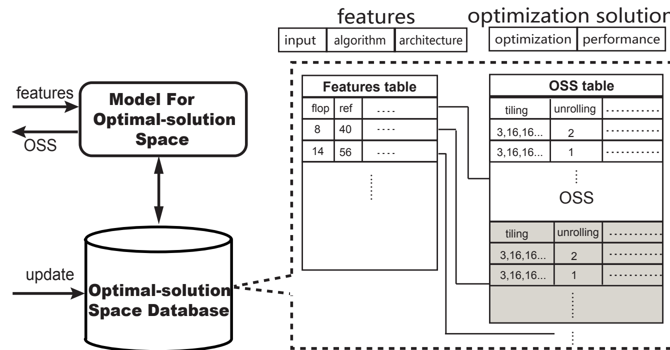

Figure 9: The database is used for the training model and for generating OSSs. It is indexed by the feature vector, and returns the corresponding OSS.

Figure 9 illustrates the structure of the OSS database and its workflow. It is indexed by the feature vector, and returns the corresponding OSS if it exists. The feature vectors and OSS are stored in two tables, and are connected by a foreign key, i.e., `feature.id` equals to `optimalspace.featureid`.

During the training phase, the OSS database stores and updates the OSS for each training instance. Once a training running instance finishes executing, an **`Update`** function is invoked to update its running record in the database. 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 collected by TAU [32]. If either one of the two tables has no matched record, **`Update`** adds the corresponding feature vector or solution vector. Otherwise, the new training record is updated in the OSS database.

5.4 Generate And Validate OSS

During the autotuning phase, the module is responsible for inputting the feature vector of the incoming case and returning its OSS. For an incoming test case, its feature vector is compared with that of running instances in the OSS database. The module calculates their differences and uses the OSS model to predict the OR of their OSSs. Then, the OSS with maximum OR is returned as the result. The result contains near-optimal solutions and we validate these solutions using an empirical approach. In addition, we provide an option to set a performance threshold, and validation is stopped when reaching the threshold. Considering the rationale of this model, the optimization solutions of high quality accomplish the empirical validation rapidly and make the whole autotuning overhead much lower than with other strategies.

If the performance threshold is not satisfying after finishing validation, a traditional search-based approach will be performed as a backup. One thing to note is that the traditional autotuning procedure also updates the OSS database with its running record. As a consequence, the iteration may generate a self-learning database that trains a more powerful model.

6. CODE GENERATION

Table 4: The parameterized optimization space.

Optimization		Parameter	Range
Technology	Description		
Cache Blocking	blocking for Level 1	L_{1i} L_{1j} L_{1k}	{1,2..N} {1,2..N} {1,2..N}
	blocking for Level 2	L_{2i} L_{2j} L_{2k}	{1,2..N} {1,2..N} {1,2..N}
	blocking for Level 3	L_{3i} L_{3j} L_{3k}	{1,2..N} {1,2..N} {1,2..N}
CUDA Blocking	blocking for CUDA threads	t_x t_y t_z	{1,2..N} {1,2..N} {1,2..N}
	blocking for CUDA blocks	b_x b_y b_z	{1,2..N} {1,2..N} {1,2..N}
OpenMP	thread-level parallelism	$number$	{1,2..16}
Unrolling	unrolling stencil loop	$factor$	{1,2..8}
SIMDization	data-level parallelism	$dosimd$	{0,1}
Compiler	flag for icc or nvcc	$cflag$	{O0,O1,O2,-O3,-xSSE3,-xAVX...}

According to the detected programming model or compiler back-end, the eDSL codes are translated to the corresponding high level language. For each **`STENCIL`** structure, the translator first acquires domain information and computation times through **`Domain`** and **`T`**, and creates a loop body for the stencil computation. Second, **`GridData`** variables are replaced by memory objects corresponding to an array and its iterator variables in the memory of a particular programming model. The generated code is a naive code solution and needs to be optimized. We apply optimization strategies based on previous autotuning results. Some of our optimization strategies are summarized in Table 4. Finally, a highly-tuned code solution is generated and the autotuing process finishes.

7. EVALUATION

7.1 Experimental Setup

We conduct the experiments on a 16-core 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 single-precision). The back-end compiler is an Intel compiler version 13.1 and an NVCC version 6.0, respectively.

Test Dataset: We evaluate FAST with a benchmark set composed of five stencil computation applications. FDTD [29] represents a 3D 5-point stencil with order-1, which is used in modeling computational electrodynamics. HEAT [20] represents a 3D 7-point stencil with order-1 which is used in chemical diffusion, WAVE [37] represents 3D 25-points stencil with order-4 which is used in fluid dynamics, POISSON [48] represents 3D 19-points stencil with order-1 that used in mechanical engineering. HIMENO [21] represents 3D 19-points with order-1.

Initializing Database: We initialize the OSS database from various stencil running instances using a Python script and generate stencil kernels by varying stencil order (n_o) from 1 to 3, parameter number (n_p) from 1 to 4 and array number (n_a) in $\{2, 4, 8, 16\}$. That is, the number of training kernels is $3 \times 4 \times 4 = 48$. The *flop density* is equal to $(1 + (6 + n_p) * o) * n_a/2 - 1$, and *reference density* is equal to $(1 + 6 * n_o) * n_a/2 * 4$ when the data type is single floating point. We use three input sizes $\{128^2 \times 256, 256^2 \times 512, 512^2 \times 1024\}$ and two target platforms {Sandy Bridge, K20}. Thus, the total number of running instances is $3 \times 2 \times 48 = 288$.

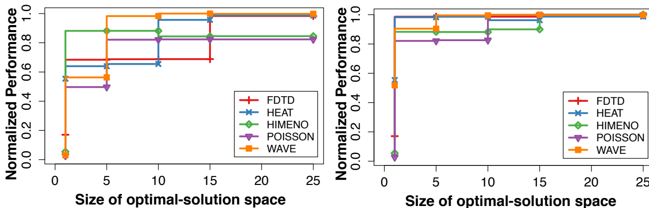

Figure 10: A properly sized OSS is a key factor to achieving high performance in FAST. The left graph shows the performance result using different OSS sizes. In the right graph, we further combine five closest OSSs as the result to improve robustness.

Training Model: An appropriate OSS size is important to FAST. As shown in the left graph in Figure 10, the performance achieved for five applications (described in the test dataset) is far away from the optimal solution when the size of the OSS is one. When increasing the size, the performance becomes better and stable once it reaches 20. However, there are still two applications whose performance is less than 90% of the optimal solution. We found that the prediction model is not so robust, where the returned OSS is not the closest one all the time. We improve FAST further by combining five closest OSSs predicted by the model and returning it as the target OSS. The right picture of Figure 10 shows that all cases achieve 99% of their best performance when the size of the OSS is larger than 15. As a result, we set the size of the OSS to be 25 in FAST's training process and combine the five closest OSSs to create each resulting OSS.

We compare *FAST* with baseline programs (*Baseline*) and two other stencil optimization frameworks (*SDSL, Patus*) with the support of the DSL compiler and autotuning techniques. For simplicity of presentation we only report experimental results for the $128 \times 128 \times 256$ problem size.

- *Baseline*: A straightforward implementation of five benchmark applications according to their stencil formulations. They are compiled by the back-end compiler with default optimization flags, i.e., `'-O3 -ipo'` for the Intel compiler, `'-O3 -Xcompiler -O3'` for CUDA.

Table 5: Autotuning running time in seconds.

Program	Feature Analysis	Tuning	Total
HIMENO	4.6	551	555.6
FDTD	4.8	99	103.8
POISSON	4.4	107	111.4
WAVE	4.6	47	51.6
HEAT	4.2	199	203.2

- *SDSL [20]*: 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 as well. There is a built-in autotuning module that implements an improved brute-force search with the help of tuning annotations in the language.

- *Patus [35]*: The Patus stencil optimization framework focuses on the autotuning strategy itself, which is the most similar counterpart to FAST. There is a flexible autotuner that integrates several efficient heuristic algorithms and provides language support to annotate the tuning strategy.

We evaluate the autotuning frameworks in terms of autotuning speed and performance. In summary, our framework achieves the best performance for most of the benchmark applications. The most remarkable advantage of our framework is that it generates optimized codes in minutes instead of either hours or days (Table 1). More specifically, it reduces tuning steps by $10 - 2697$ times compared with the conventional search-based strategies used extensively in the counterpart autotuning frameworks.

7.2 Autotuning Speed

Table 5 profiles the execution time of the whole autotuning process. Compared to the existing stencil computation autotuning systems, which require either hours or days to perform tuning, FAST finishes the autotuning process in minutes. A major source of this improvement comes from the significantly reduced tuning steps. Due to the brute-force-like search algorithm used in SDSL, we evaluate autotuning speed in terms of the tuning step by comparing with Patus. In fact, Patus adopts several representative auto-tuning search strategies that appeared in most of autotuners [3]. Figure 11 plots the best performance achieved at each tuning step for both CPU and GPU implementations. The blue line represents the greedy-based autotuning used in Patus and the red line represents FAST. Two dotted lines are used to mark the tuning steps' satisfying threshold, which is 95% of best performance. For Patus, the fastest one costs 225 steps in the case of FDTD and the slowest one costs 2725 steps in the case of Poisson. For FAST, the fastest one costs one step, and the slowest one only costs 23 steps in the case of FDTD. In summary, the autotuning overhead is significantly reduced by an average two orders of magnitude with FAST. The rapid convergence of FAST is due to the high quality solutions contained in the predicted OSS. Figure 12 presents normalized performance of the optimization solutions in the predicted OSS. The y-axis is the performance normalized to the best optimization solution and the x-axis presents optimization solutions in the predicted OSS. For HIMENO and WAVE, all optimization solutions fall in the interval of 90%-100% to of the best performance. For POISSON and FDTD, there are half of the optimization solutions achieved at least 90% of the best performance. For

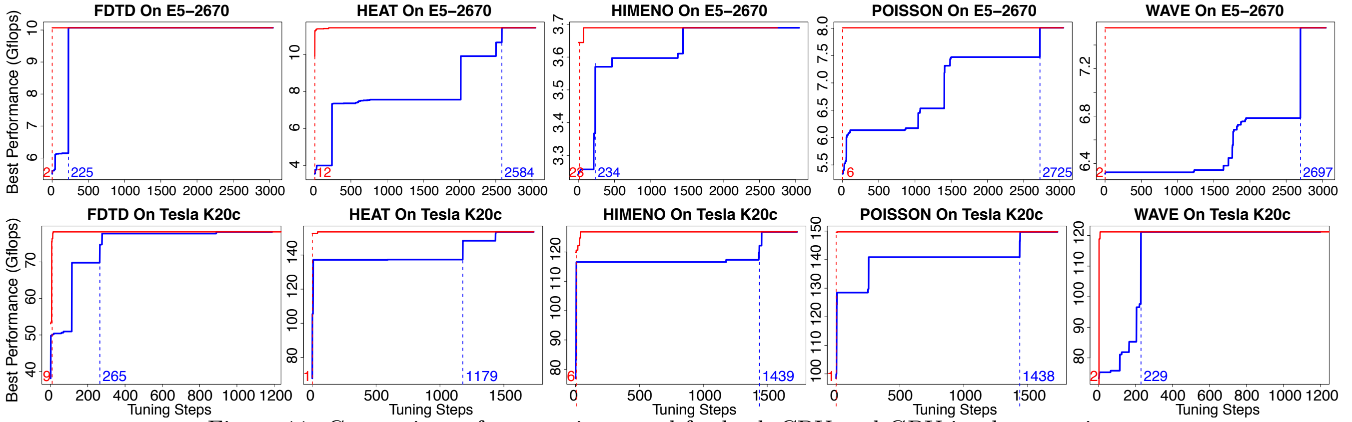

Figure 11: Comparison of autotuning speed for both CPU and GPU implementations.

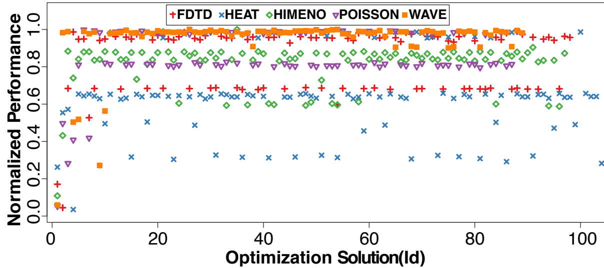

Figure 12: Performance of optimization solutions in the predicted OSS.

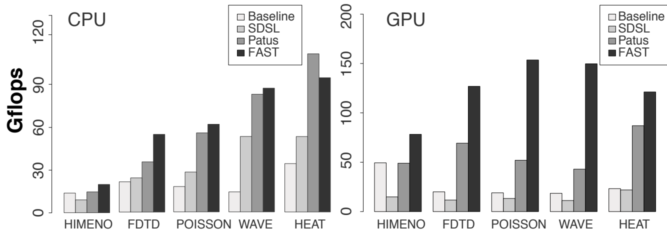

Figure 13: Comparison of performance in Gflops on for both CPU and GPU implementations.

HIMENO, only one-third of optimization solutions are better than 90% of the best performance. The behaviors of these optimization solutions are due to the differences between the applications and the running instances in initial OSS database. Obviously, WAVE is the most similar case, which shares most optimal solutions with these running instances in the OSS database. In spite of this, for all five benchmark applications, their OSS definitely contains several optimization solutions that achieve more than 95% of best performance. This verifies that FAST results in high accuracy of prediction for optimal solutions.

7.3 Performance

We report performance in Gflops for these five benchmark applications in Figure 13 on both the CPU and GPU. On average, *FAST* improves performance by 3.28 times over *Baseline*, 100% times over *SDSL* and 18% times over *Patus* on the CPU, and by 4.86 times over *Baseline*, 8.3 times over *SDSL* and 1.25 times over *Patus* on the GPU.

CPU: The left bar graph in Figure 13 shows performance comparison for the CPU implementations. First, for the CPU implementations, *FAST* improves performance by 1.4-6.2 times over *Baseline*, 64%-120% over *SDSL* and -15%-53% over *Patus*, respectively. *FAST* employs the 2.5D blocking strategy [38] to reuse data between two adjacent blocks. As a consequence, for memory bound applications, such as FDTD and HIMENO, we obtain better performance results than *Patus*. However, for applications with high operational intensity, such as HEAT, Patus surpasses *FAST* due to its explicit SIMD optimization by 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 by the growing size of tiles when the dimension is larger than two. That leads to its unsatisfactory performance on five three-dimension applications. Second, the performance improvement differs with the five stencil applications. For example, *FAST* achieves higher speedups for HEAT and WAVE than for HIMENO and FDTD. The difference is related to the algorithmic feature of *Operational intensity*, which sets the performance upper bound according the roofline model [52].

GPU: The right bar graph in Figure 13 shows the performance comparison for the GPU implementations. *FAST* improves performance by 58% – 7 times over *Baseline*, 4.24 – 12.3 times over *SDSL* and 39% – 2.48 times over *Patus*, respectively. Unlike the implementations on the CPU, *FAST* outperforms the GPU counterparts. One major reason is that our optimization solutions adopt 2.5D blocking, shared memory, constant memory and texture reference optimization strategies, but both *SDSL* and *Patus* only employ straightforward parallelization and blocking strategies.

8. RELATED WORK

Autotuning has emerged as a critical strategy for achieving high performance as architectural complexity grows. A number of autotuning frameworks have been developed for building efficient, portable libraries or frameworks in specific domains. For example, PHiPAC [5] is an autotuning system for dense matrix multiplication. ATLAS [51] utilizes empirical autotuning to produce a cache-contained matrix multiply. FFTW [16] uses empirical autotuning to combine solvers for FFTs. Other autotuning systems include SPARSITY [23] for sparse matrix computations, SPIRAL [39, 49, 15] for digital signal processing, UHFFT [1] for FFT on multicore systems, and OSKI [50] for sparse matrix kernels. ActiveHarmony [44, 46] provides a general framework for tuning configurable libraries and exploring different compiler optimizations. Two major approaches of autotuning are search-based and prediction-based. Search-based autotuning is a generic but time consuming approach, which empirically evaluates candidates in the optimization space. For this reason, a number of strategies have been applied to speed up search, such as pruning, heuristics and analytical models [17, 9, 25, 52, 31, 19, 41, 20, 33, 12]. Prediction-

based autotuning leverages prior knowledge to achieve the best optimization solution directly and has very low overhead. It is always applied to the problem of code solution or algorithm selection [30, 42, 45, 28, 18, 14, 24].

On the other hand, as a higher level of abstraction, DSLs are promising trend to improve usability since many application domains do not have a clear abstraction as a library interface. They leverage a specialized compiler to help application experts easily write high performance codes and facilitate optimization. Several successful attempts include graph [22], multigrid [2] and stencil [34] computations.

FAST combines techniques of DSL and autotuning for optimizing stencil computation. Recent work [10, 2, 43, 35, 12, 31, 27, 26, 20] has demonstrated their strength in automatic optimization on diverse architectures. FAST shares the same programming productivity of DSLs with these works for domain application experts and further exploits using the DSLs to extract features that help with autotuning. The features extraction is an important aspect of our autotuning approach compared to existing ones. For example, some general frameworks like Intelligent Compilers [6], CHiLL [47], Patus [35] and Orio [19] are designed for autotuning compiler or library developers instead of domain application experts. They force an experienced user to write a script for tuning configurations. Afterwards, FAST takes a prediction-based approach to deal with stencil autotuning that differs from previous work. First, previous autotuning systems employ pruning and heuristic algorithms to search the optimization space. They establish the optimization space using all configurable parameters exposed in the strategy definition and some internal default parameters (loop nest unrolling factors, padding). Then, multiple search algorithms are applied, such as Nelder-Mead simplex search, dynamic programming and stochastic optimization methods including random, evolutionary, and hill climbing. Recently, OpenTuner [3] allows many search techniques to work together and supports customizable configuration representations of a sophisticated search technique. Instead of using these search-based approaches, FAST builds a self-learning OSS database to automatically learn an OSS prediction model and predicts optimal solutions directly, which significantly decreases tuning overhead.

9. CONCLUSION

In this paper, we propose and implement a framework based on the OSS model for stencil autotuning (FAST). Compared to previous work, FAST adopts a prediction-based approach to autotune stencil computations, and greatly decreases tuning overhead. The key insight behind FAST is that that two running instances have a small OSS with high overlap ratio if their features are of high similarity. The experiments on both x86 multicore CPU and NVIDIA GPU show that FAST outperforms its counterparts in terms of both performance and autotuning speed. In particular, the autotuning speed is improved by $10 - 2697$ times over the conventional autotuning strategy.

10. ACKNOWLEDGMENTS

We would like to express our gratitude to all reviewer's constructive comments and Dr. Brad Chamberlain for helping us polishing this paper. This work is supported by National Natural Science Foundation of China, under grant no. (61272134, 31327901, 91430218, 60921002, 60925009, 61472395), National 863 Program (2009AA01A129) and 973 Program (2012CB316502 and 2011CB302502).

References

[1] A. Ali, L. Johnsson, and J. Subhlok. Scheduling fft computation on smp and multicore systems. In *Proceedings of the 21st annual international conference on Supercomputing*, pages 293–301. ACM, 2007.

[2] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. Petabricks: a language and compiler for algorithmic choice. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, volume 44. ACM, 2009.

[3] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe. Opentuner: An extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*, Edmonton, Canada, August 2014.

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

[5] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. In *Proceedings of the 11th international conference on Supercomputing*, pages 340–347. ACM, 1997.

[6] J. Cavazos. Intelligent compilers. In *Cluster Computing, 2008 IEEE International Conference on*, pages 360–368. IEEE, 2008.

[7] C. Chen, J. Chame, and M. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *International Symposium on Code Generation and Optimization*, pages 111–122. IEEE, 2005.

[8] J. W. Choi, A. Singh, and R. W. Vuduc. 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*, volume 45, pages 115–126. ACM, 2010.

[9] M. Christen, O. Schenk, and Y. Cui. Patus for convenient high-performance stencils: evaluation in earthquake simulations. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–10. IEEE, 2012.

[10] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. *ACM SIGPLAN Notices*, 34(7):1–9, 1999.

[11] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM review*, 51(1):129–159, 2009.

[12] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 4. IEEE Press, 2008.

[13] C. Dubach, T. Jones, and M. O'Boyle. Microarchitectural design space exploration using an architecture-centric approach. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 262–271. IEEE Computer Society, 2007.

[14] E. Fink. How to solve it automatically: Selection among problem solving methods. In *AIPS*, pages 128–136, 1998.

[15] F. Franchetti, F. de Mesmay, D. McFarlin, and M. Püschel. Operator language: A program generation framework for fast kernels. In *Domain-Specific Languages*, pages 385–409. Springer, 2009.

[16] M. Frigo and S. G. Johnson. The design and implementation of fftw3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[17] A. Ganapathi, K. Datta, A. Fox, and D. Patterson. A case for machine learning to optimize multicore performance. In *First USENIX Workshop on Hot Topics in Parallelism (HotPar'09)*, 2009.

[18] A. Guerri and M. Milano. Learning techniques for automatic algorithm portfolio selection. In *ECAI*, volume 16, page 475, 2004.

[19] A. Hartono, B. Norris, and P. Sadayappan. Annotation-based empirical performance tuning using orio. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–11. IEEE, 2009.

[20] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector simd architectures. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 13–24. ACM, 2013.

[21] R. Himeno. Himeno benchmark, 2011.

[22] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-marl: a dsl for easy and efficient graph analysis. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, volume 40, pages 349–362. ACM, 2012.

[23] E.-J. Im and K. Yelick. Optimizing sparse matrix computations for register reuse in sparsity. In *Computational ScienceâĂŤICCS 2001*, pages 127–136. Springer, 2001.

[24] N. Jain, A. Bhatele, M. P. Robson, T. Gamblin, and L. V. Kale. Predicting application performance using supervised learning on communication features. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, page 95. ACM, 2013.

[25] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[26] S. Kamil, D. Coetzee, and A. Fox. Bringing parallel performance to python with domain-specific selective embedded just-in-time specialization. In *Python for Scientific Computing Conference (SciPy)*, 2011.

[27] M. Khan, P. Basu, G. Rudy, M. Hall, C. Chen, and J. Chame. A script-based autotuning compiler system to generate high-performance cuda code. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):31, 2013.

[28] L. Kotthoff, I. P. Gent, and I. Miguel. A preliminary evaluation of machine learning in algorithm selection for search problems. In *Fourth Annual Symposium on Combinatorial Search*, 2011.

[29] K. S. Kunz and R. J. Luebbers. *The finite difference time domain method for electromagnetics.* CRC press, 1993.

[30] J. Li, G. Tan, M. Chen, and N. Sun. Smat: an input adaptive auto-tuner for sparse matrix-vector multiplication. In *ACM SIGPLAN Notices*, volume 48, pages 117–126. ACM, 2013.

[31] T. Lutz, C. Fensch, and M. Cole. Partans: An autotuning framework for stencil computation on multi-gpu systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):59, 2013.

[32] A. D. Malony, J. Cuny, and S. Shende. Tau: Tuning and analysis utilities. Technical report, LALP-99–205, Los Alamos National Laboratory Publication, 1999.

[33] A. Mametjanov, D. Lowell, C.-C. Ma, and B. Norris. Autotuning stencil-based computations on gpus. In *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pages 266–274. IEEE, 2012.

[34] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. Physis: an implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12. IEEE, 2011.

[35] C. Matthias, S. Olaf, and B. Helmar. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676–687. IEEE, 2011.

[36] J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus. In *Proceedings of the 23rd international conference on Supercomputing*, pages 256–265. ACM, 2009.

[37] P. Micikevicius. 3d finite difference computation on gpus using cuda. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 79–84. ACM, 2009.

[38] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 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*, pages 1–13. IEEE Computer Society, 2010.

[39] M. Püschel, J. M. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing alogorithms. *International Journal of High Performance Computing Applications*, 18(1):21–45, 2004.

[40] A. Qasem, M. J. Cade, and D. Tamir. Improved energy efficiency for multithreaded kernels through model-based autotuning. In *Green Technologies Conference, 2012 IEEE*, pages 1–6. IEEE, 2012.

[41] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[42] J. R. Rice. The algorithm selection problem. 1975.

[43] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The pochoir stencil compiler. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 117–128. ACM, 2011.

[44] C. Ţăpuş, I.-H. Chung, J. K. Hollingsworth, et al. Active harmony: Towards automated performance tuning. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–11. IEEE Computer Society Press, 2002.

[45] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger. A framework for adaptive algorithm selection in stapl. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 277–288. ACM, 2005.

[46] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

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

[48] D. Unat, X. Cai, and S. B. Baden. Mint: realizing cuda performance in 3d stencil methods with annotated c. In *Proceedings of the international conference on Supercomputing*, pages 214–224. ACM, 2011.

[49] Y. Voronenko, F. de Mesmay, and M. Püschel. Computer generation of general size linear transform libraries. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 102–113. IEEE Computer Society, 2009.

[50] R. Vuduc, J. W. Demmel, and K. A. Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, volume 16, page 521. IOP Publishing, 2005.

[51] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–27. IEEE Computer Society, 1998.

[52] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.