SMAT: An Input Adaptive Auto-Tuner for Sparse Matrix-Vector Multiplication

Jiajia Li[1,2], Guangming Tan[1], Mingyu Chen[1], Ninghui Sun[1]

[1]State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
[2]University of Chinese Academy of Sciences
{lijiajia, tgm, cmy, snh}@ict.ac.cn

Abstract

Sparse Matrix Vector multiplication (SpMV) is an important kernel in both traditional high performance computing and emerging data-intensive applications. By far, SpMV libraries are optimized by either application-specific or architecture-specific approaches, making the libraries become too complicated to be used extensively in real applications. In this work we develop a Sparse Matrix-vector multiplication Auto-Tuning system (SMAT) to bridge the gap between specific optimizations and general-purpose usage. SMAT provides users with a unified programming interface in compressed sparse row (CSR) format and automatically determines the optimal format and implementation for any input sparse matrix at runtime. For this purpose, SMAT leverages a learning model, which is generated in an off-line stage by a machine learning method with a training set of more than 2000 matrices from the UF sparse matrix collection, to quickly predict the best combination of the matrix feature parameters. Our experiments show that SMAT achieves impressive performance of up to 51GFLOPS in single-precision and 37GFLOPS in double-precision on mainstream x86 multi-core processors, which are both more than 3 times faster than the Intel MKL library. We also demonstrate its adaptability in an algebraic multigrid solver from Hypre library with above 20% performance improvement reported.

Categories and Subject Descriptors F.2.1 [*Numerical Algorithms and Problems*]: Computations on matrices; C.1.2 [*Multiple Data Stream Architectures (Multiprocessors)*]: Parallel processors

General Terms Algorithms, Performance

Keywords sparse matrix-vector multiplication, SpMV, auto-tuning, data mining, algebraic multi-grid

1. Introduction

Nowadays, high performance computing technologies are driven simultaneously by both Exascale FLOPs (ExaFlops) and data-intensive applications. Although the grand challenges of these applications are diverse, an interesting observation is that they share an intersection in terms of sparse linear system solvers. For example, the well-known ExaFlops applications, laser fusion in international thermonuclear experimental reactor (ITER) [3] and global cloud system resolving climate modeling [20], spend most of the execution time on solving large-scale sparse linear systems. Similarly, lots of large-scale graph analysis applications like PageRank [9] and HITS [26] involve sparse matrix solvers in identifying relationships. All of these algorithms rely on a core Sparse Matrix Vector multiplication (SpMV) kernel consuming a large percentage of their overall execution time. For example, the algebraic multigrid (AMG) solver [13], an iterative algorithm widely used in both laser fusion and climate modeling, reports above 90% SpMV operation of its overall iterations.

Since 1970s, plenty of researches have been dedicated to optimizing SpMV performance for its fundamental importance, which are generally separated into two paths, one for developing new application-specific storage formats [7, 21, 29–32, 36], and the other for tuning on emerging processor architectures [10, 11, 16, 22, 27, 28, 34, 35]. This separation leads to low performance and low productivity in SpMV solvers and applications:

- *Low Performance: sparse solvers are not aware of the diversity of input sparse matrices.*
  It is well-known that for a specific sparse matrix SpMV performance is sensitive to storage formats. More than ten formats have been developed during the last four decades. However, most of them show good performance only for a few specific applications, and are rarely adopted in widely used numeric solvers such as Hypre [14], PETSc [6], Trilinos [18] etc. In fact, these solvers mainly support one popular storage format CSR (compressed sparse row, explained in Section 2.1). Thus, they usually perform poorly in some applications whose sparse matrices are not appropriate for the supported CSR format. Besides, even considering only one application, patterns of sparse matrices may change at runtime. Take AMG as an example again, it generates a series of different sparse matrices by the coarsen algorithm on successive grid levels. Figure 1 illustrates an example of the need for dynamic storage formats in Hypre AMG solver. At the first few levels, SpMVs desire DIA or COO format (see Section 2.1) for the optimal performance. While at the coarser-level grids, other formats like CSR may be more desirable due to high zero-filling ratio in DIA format. Therefore, a numerical solver should be adaptive to different sparse matrix formats for achieving better performance.
- *Low Productivity: sparse libraries provide multiple interfaces to users*
  An alternative solution to the above low performance issue is to rewrite these solvers for every storage format. Unfortunately, such changes are too time-consuming for legacy software. An easier option may be to provide an SpMV library, which can automatically adapt to various sparse matrix formats for

PLDI'13, June 16–19, 2013, Seattle, WA, USA.
Copyright © 2013 ACM 978-1-4503-2014-6/13/06...$15.00

numerical solvers and real applications. However, none of the available sparse libraries accommodate us with a consistent programming interface to replace the original SpMV kernels. All these libraries including Intel MKL [1] and OSKI [31] provide different interfaces for every format supported, which leaves the burden of determining and integrating the optimal format to application programmers. As a consequence, although these libraries are highly optimized by either hand-tuning or auto-tuning techniques [4, 5, 7, 8, 10, 11, 17, 21, 29, 31, 35, 36] with processor architectures evolving, it is still difficult to integrate them into high level solvers or applications. This fact results in an embarrassing situation: on one hand, there are several high performance sparse libraries available; on the other hand, solver or application developers seldom use them in practice.

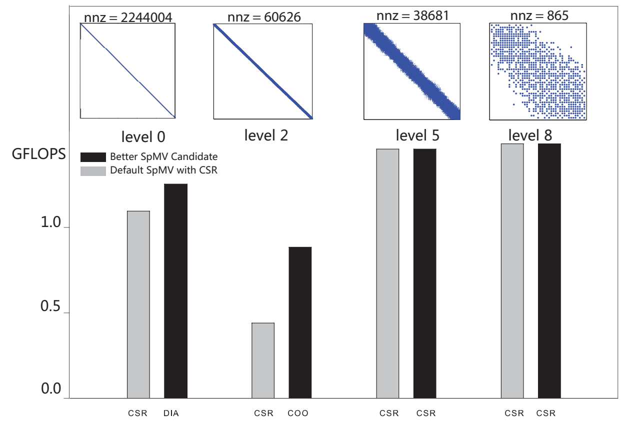


Figure 1. An example of dynamic sparse matrix structures in AMG solver and their SpMV performance using different formats. The *nnz* represents the number of nonzero elements in matrices.

In this paper we propose a novel Sparse Matrix-vector multiplication Auto-Tuner (SMAT) to provide both application- and architecture-aware SpMV kernels. A distinct feature of SMAT is the unified programming interface for different storage formats. Based on a comprehensive investigation (in Section 2) that shows CSR performs the best for most sparse matrices, we define the unified interface as CSR format. In this way users only need to prepare their sparse matrices in CSR format as the input, and SMAT automatically determines the optimal storage format and implementation on a given architecture. Our auto-tuning approach takes advantage of a black-box machine learning model to train representative performance parameters extracted from the sparse matrix pattern and architectural configuration. Once the model is trained on the target architecture, SMAT evaluates candidate storage formats and determines the optimal format and implementation by leveraging either existing auto-tuning work [11, 17, 21, 29, 31, 35, 36] or available hand-tuned codes.

Apart from the conventional auto-tuning techniques considering architectural parameters, the novelty of SMAT system is *cooperatively auto-tuning between algorithms and architectures*. In other words, SMAT searches for the optimal storage format based on sparse matrix structures at the algorithm level while generating the optimal implementation based on the processor architecture. Though auto-tuning techniques for optimizing libraries and applications have been explored for a long time, the originality of our work lies in an input adaptive mechanism that identifies characteristics of both applications and architectures. Besides, the main contributions of this paper are:

- We enable reusable training by adopting more than 2000 sparse matrices from real applications and systematically sampling the parameter space, including sparse structure characteristics and architecture configuration.
- We propose a unified interface on top of SpMV kernels to avoid the tedious work of manually choosing among sparse storage formats. SMAT facilitates users to adopt the tuned library for improving performance with little extra programming effort.
- We develop a flexible and extension-free framework, with which users can add not only new formats and novel implementations in terms of their needs, but also more features and larger datasets.

We implement SMAT and train it using sparse matrix collection from the university of Florida [12] on two x86 multi-core processors, Intel Xeon X5680 and AMD Opteron 6168. With sparse matrices selected from various application areas, SpMV kernels generated by SMAT achieve impressive performance of up to 51GFLOPS in single-precision (SP) and 37GFLOPS in double-precision (DP). The improvement of SMAT over Intel MKL is more than 3.2 times on average. We also evaluate the adaptability of SMAT in AMG algorithm of sparse solver Hypre [14], and the results show above 20% performance improvement.

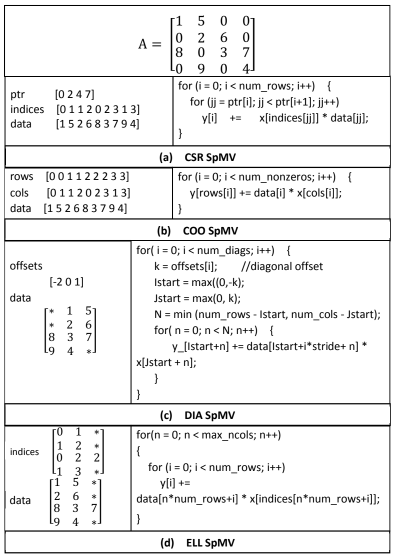
The rest of the paper is organized as follows. Section 2 presents several storage formats of sparse matrices and our motivation. In Section 3 we propose an overview of SMAT system. Then the three important contributions are described in the following sections. Section 4 shows the parameter extraction process to build the feature database. After that, we illustrate the machine learning and kernel searching process in Section 5, and the runtime procedure in Section 6. We give the performance results on representative sparse matrices and a real application in Section 7, as well as analyze the accuracy and overhead of SMAT. Related work is presented in Section 8, and conclusion in Section 9.

2. Background

2.1 Sparse Matrix Storage Format

In order to reduce the complexity of both space and computation, a compressed data structure is commonly used to only store nonzero elements of a sparse matrix. Since sparse structure is closely related to compression effectiveness and SpMV behavior on specific processor architectures (i.e.,vector machine, cache, etc.), more than ten compressed data structures (i.e. formats) have been developed since 1970s. However, most of them were customized only for certain special cases. By far four basic storage formats CSR, DIA, ELL, COO are extensively used in mainstream applications, from which most other formats can be derived. For example, when there exist many dense sub-blocks in a sparse matrix, the corresponding blocking variants (i.e. BCSR, BDIA, etc.) may perform better if an appropriate blocking factor is selected. Here, COO is specially noted because it usually performs better in large scale graph analysis applications [36]. In this proof-of-concept work, we start from the four basic formats and make it possible to extend for supporting other formats in our auto-tuning system.

Figure 2 illustrates a sparse matrix example represented in the four basic formats respectively. CSR (Compressed Sparse Row) format contains three arrays: "data" stores nonzero elements of the sparse matrix, "indices" stores their corresponding column indices, and the beginning positions of each row are stored in "ptr". COO (COOrdinate) format explicitly stores the row indices. The "rows", "cols" and "data" arrays store the row indices, the column indices, and the values of all nonzero elements respectively. DIA (Diagonal) format stores non-zeros by the order of diagonals. "data" array stores the dense diagonals, and "offsets" records the offsets of each diagonal to the principal one. The idea of the ELL (ELLPACK) format is to pack all non-zeros towards left, and store the packed dense matrix. “data” array stores this dense matrix, and “indices” stores the corresponding column indices. Please refer to [25] for more detailed illustration. The corresponding basic implementations of the four formats are also presented in Figure 2.



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

```
ptr        [0 2 4 7]
indices    [0 1 1 2 0 2 3 1 3]
data       [1 5 2 6 8 3 7 9 4]
```

```
for (i = 0; i < num_rows; i++)    {
    for (jj = ptr[i]; jj < ptr[i+1]; jj++)
        y[i]    +=     x[indices[jj]] * data[jj];
}
```

(a) CSR SpMV

```
rows    [0 0 1 1 2 2 2 3 3]
cols    [0 1 1 2 0 2 3 1 3]
data    [1 5 2 6 8 3 7 9 4]
```

```
for (i = 0; i < num_nonzeros; i++)    {
    y[rows[i]] += data[i] * x[cols[i]];
}
```

(b) COO SpMV

offsets

[-2 0 1]

data

$$\begin{bmatrix} * & 1 & 5 \\ * & 2 & 6 \\ 8 & 3 & 7 \\ 9 & 4 & * \end{bmatrix}$$

```
for( i = 0; i < num_diags; i++)    {
     k = offsets[i];       //diagonal offset
     Istart = max((0,-k);
     Jstart = max(0, k);
     N = min (num_rows - Istart, num_cols - Jstart);
     for( n = 0; n < N; n++)    {
          y_[Istart+n] += data[Istart+i*stride+ n] *
x[Jstart + n];
     }
}
```

(c) DIA SpMV

indices $\begin{bmatrix} 0 & 1 & * \\ 1 & 2 & * \\ 0 & 2 & 2 \\ 1 & 3 & * \end{bmatrix}$

data $\begin{bmatrix} 1 & 5 & * \\ 2 & 6 & * \\ 8 & 3 & 7 \\ 9 & 4 & * \end{bmatrix}$

```
for(n = 0; n < max_ncols; n++)
{
   for (i = 0; i < num_rows; i++)
       y[i] +=
data[n*num_rows+i] * x[indices[n*num_rows+i]];
}
```

(d) ELL SpMV

Figure 2. Data structures and basic SpMV implementations

2.2 Motivation

Although it is a common sense to store sparse matrices in a proper format, it is non-trivial to figure out how to automatically select the combination of appropriate storage format and its optimal implementation on a specific architecture. A distinct difference is that previous auto-tuning techniques [11, 17, 21, 29, 31, 35, 36] only take architectural impact into account while the storage format is predefined by users. Actually, it is really a challenging job because sufficient performance characteristics must be extracted from diverse sparse matrices and then used to quickly determine the optimal format and implementation.

For extracting characteristics and showing potential benefits from tuning storage format based on sparse structures, we conduct comprehensive experiments on the UF collection [12], which consists of more than 2000 sparse matrices from real applications covering extensive application areas. For simplicity without loss of accuracy, we exclude the matrices with complex values and too small size, and eventually studied 2386 sparse matrices in total in this work. Table 1 lists their application areas spread in more than 20 domains.

Table 1 also summarizes the distribution of the optimal storage formats for all 2386 sparse matrices. Column $2-5$ represent the number of sparse matrices that have affinity with CSR, COO, DIA, and ELL formats, respectively. The last row calculates the proportion of the matrices have affinity with each format to the whole matrix set respectively. Apparently, CSR is favored by most sparse matrices. Furthermore, for highlighting performance variance and plotting legible graphs, we select 16 representatives from the 2386 matrices and compare the performance in GFLOPS among the four storage formats without meticulous implementations. As shown in Figure 3 the largest performance gap is about 6 times. The reasons for the variance will be explained in Section 4. Obviously it is not reasonable to apply only one storage format in sparse numeric solvers.

Given a specific format, there are already well-studied auto-tuning tools or libraries [11, 17, 21, 29, 31, 35, 36], which can be leveraged to generate the optimal SpMV implementation on a specific platform. If the matrix structure stays the same during the whole lifetime of an application, even a brute-force search algorithm might be reasonable to generate the best candidate. As noted in Section 1, certain applications like AMG dynamically change sparse matrix structures (see Figure 1). Therefore, two key issues will be addressed in this paper: (i). *determine the optimal storage format for any input sparse matrix*; (ii). *generate the optimal combination of format and implementation at runtime with low overhead.*

Table 1. Application and distribution of affinity to each format

Application Domains	CSR	COO	DIA	ELL	Total
graph	187	114	6	27	334
linear programming	267	52	3	5	327
structural	224	14	35	4	277
combinatorial	122	50	10	84	266
circuit simulation	110	149	0	1	260
computational fluid dynamics	110	8	47	3	168
optimization	113	15	8	2	138
2D_3D	64	21	19	17	121
economic	67	4	0	0	71
chemical process simulation	47	14	2	1	64
power network	45	15	0	1	61
model reduction	29	34	6	1	60
theoretical quantum chemistry	21	0	26	0	47
electromagnetics	17	1	12	3	33
semiconductor device	28	1	3	1	33
thermal	19	3	3	4	29
materials	12	3	11	0	26
least squares	10	2	0	9	21
computer graphics vision	8	1	1	2	12
statistical mathematical	2	1	3	4	10
counter-example	3	4	1	0	8
acoustics	5	0	2	0	7
robotics	3	0	0	0	3
Percentage	63%	21%	9%	7%	2386

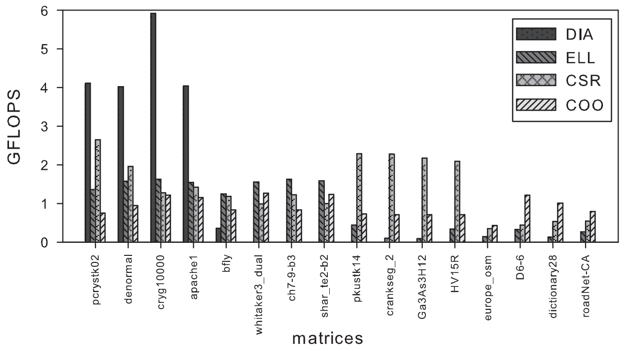


Figure 3. Performance variance among different storage formats for 16 representative matrices.

3. SMAT Overview

We develop SpMV Auto-Tuner (SMAT) to choose the optimal storage format and implementation of SpMV, whose framework is shown in Figure 4. We consider both architecture parameters (such as TLB size, cache and register size, prefetch, threading policy, etc., details in [35]) and application parameters (such as matrix dimension, diagonal situation, nonzero distribution, etc., to be described in Section 4) to optimize an SpMV kernel and evaluate its performance on sparse matrices from different applications. The large parameter space makes it difficult to find the most suitable sparse matrix format and optimization method on a specific architecture. For this reason, we generate a learning model using data mining to find the optimal result.

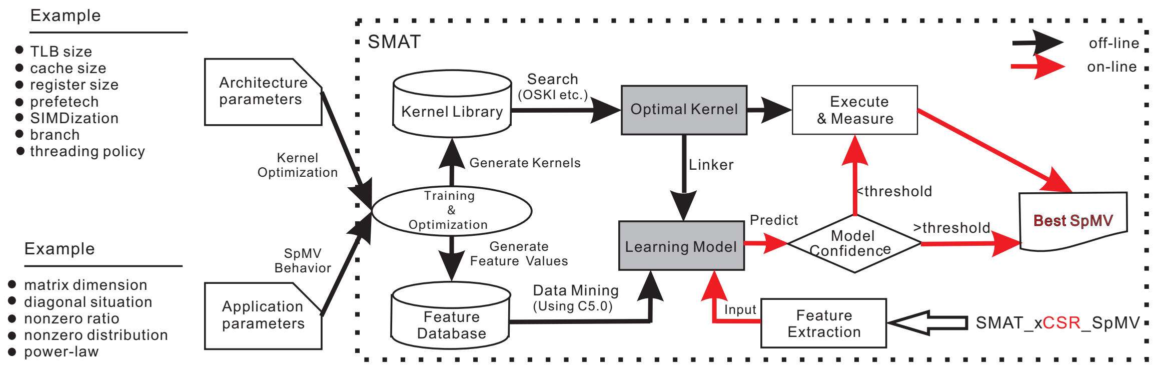


Figure 4. SMAT architecture

Since each sparse matrix has different SpMV behaviors, we train the corresponding parameter values using UF collection to generate a feature record for each matrix. Data mining is used to generate a learning model from the feature database. Besides, since a variety of kernel optimization methods consider architecture parameters (like cache blocking, register blocking, parallelization, etc.), a large kernel library is built. The optimal kernel implementation is searched based on particular architecture characteristics. Thus, a learning model, which combines application features and architecture characteristics together, is built to generate high performance kernels quickly.

Given a sparse matrix in the most popular CSR format, SMAT extracts its sparse features first. The feature values are used to predict the optimal format and implementation of SpMV. However, there are still some cases that the learning model may fail to make the right prediction. A confidence factor is added to rule the model. When the confidence factor is larger than the threshold, we determine the optimal SpMV format with the optimal implementation through the model. Otherwise, an execution-measure process is triggered to benchmark available candidates and outputs the one with the highest performance. In this way, users do not need to think about which format and implementation should be chosen. All they need to do is to input the sparse matrix in CSR format and then SMAT will give the optimal answer. Figure 5 compares application programming interfaces between Intel MKL [1] and SMAT. MKL implements six storage formats and provides the same number of function calls to users, while SMAT exposes only one function in CSR format, as most sparse matrices favor CSR according to statistical results (see Table 1).

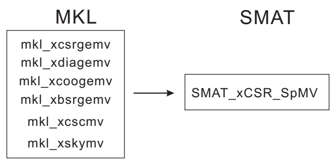


Figure 5. Application programming interface. The character *"x"* denotes numerical precision (single- or double-precision).

In addition to the benefit of the unified programming interface, there are two more advantages of SMAT – reusability and portability: First, on a specific architecture, SMAT generates the learning model once in off-line stage, and use this model repeatedly for different input matrices; Second, SMAT extends well to different architectures as the architecture features have been considered already. Besides, SMAT performs beyond existing auto-tuners by employing the performance of SpMV implementations to quantize architecture features, instead of using architecture features directly. This innovation brings two advantages: First, it makes SMAT smoothly compatible to new architectures, which is the most important thing to auto-tuners and the biggest difference from other ones; Second, it is more accurate to conduct learning using performance values instead of a band of architecture features, to narrow down the parameter space while considering both application and architecture features.

Another main advantage of SMAT is extensibility. With the nature of data mining methods, SMAT is really easy for users to add new algorithms and/or more meticulous implementation. Besides, it is also open to add new matrices and corresponding records into the database to improve the prediction accuracy. Third, if there is a need to balance accuracy and training time, it is also convenient to add or remove parameters from the learning model with SMAT.

In the following sections, three compositions of SMAT will be discussed, which are respectively parameter extraction, learning model generation, and runtime process.

4. Parameter Extraction

SMAT is an application and architecture aware auto-tuner, and it can apparently leverage the well-studied architecture-aware auto-tuners [17, 21, 31, 36] to extract architecture parameters. In this section, we present how to extract sparse structure feature parameters. To build an application-aware auto-tuner, we extract feature parameters from more than 2000 sparse matrices of UF collection [12]. Table 2 summaries a complete list of sparse structure parameters used in SMAT system.

Firstly, four parameters are used to represent the basic structure of a sparse matrix applicable to all four formats (marked by $\surd$): `M` (the number of rows), `N` (the number of columns), `NNZ` (the number of non-zeros), and `aver_RD` (the average number of non-zeros per row). Besides, a couple of advanced parameters are added to describe SpMV behavior for non-default formats in SMAT such as DIA, ELL and COO. Note: the up(down) arrows indicate that the corresponding SpMV performance gets higher once the parameter becomes larger(smaller), .

The advantages of DIA and ELL formats come from regular data access behavior to the matrix, especially the X-vector. Since DIA and ELL store matrices in a diagonal and column major layout respectively, DIA reads continuous elements of X-vector, while ELL has a large possibility to reuse X-elements. However, these two formats also suffer from the following drawbacks: First, when there are only a few elements distributed in a diagonal(row), DIA(ELL) will fill up these diagonals(rows) with zero-padding and has to do useless computation on zero elements, which will dramatically hurt the overall performance. Second, the diagonal(column)-order loop requires Y-elements to be written once per diagonal(column), which will produce frequent cache evict and memory write back operation for large sparse matrices. In order to address these issues, four parameters ({`Ndiags, max_RD, ER_DIA, ER_ELL`} in Table 2) are extracted to represent the DIA and ELL kernels' behavior.

Table 2. Feature parameters of a sparse matrix and the relationship with the formats

Parameter		Meaning	Formula	DIA	ELL	CSR	COO
Matrix Dimension	M	the number of rows	-	$\surd$	$\surd$	$\surd$	$\surd$
	N	the number of columns	-	$\surd$	$\surd$	$\surd$	$\surd$
Diagonal Situation	Ndiags	the number of diagonals	-	$\downarrow$			
	NTdiags_ratio	the ratio of "true" diagonals to total diagonals	*NTdiags_ratio*=$\frac{number\ of\ \text{"}true\ diagonals\text{"}}{Ndiags}$	$\uparrow$			
Nonzero Distribution	NNZ	the number of nonzeros	-	$\surd$	$\surd$	$\surd$	$\surd$
	aver_RD	the number of nonzeros per row	*aver_RD* = $\frac{NNZ}{M}$	$\surd$	$\surd$	$\surd$	$\surd$
	max_RD	the maximum number of nonzeros per row	*max_RD*=$\max_1^M\{number\ of\ nonzeros\ per\ row\}$		$\downarrow$		
	var_RD	the variation of the number of nonzeros per row	*var_RD* = $\frac{\Sigma_1^M \lvert row\_degree - aver\_RD\rvert^2}{M}$			$\downarrow$	
Nonzero Ratio	ER_DIA	the ratio of nonzeros in DIA data structure	*ER_DIA*=$\frac{NNZ}{Ndiags \times M}$	$\uparrow$			
	ER_ELL	the ratio of nonzeros in ELL data structure	*ER_ELL* = $\frac{NNZ}{max\_RD \times M}$		$\uparrow$		
Power-Law	R	a factor of power-law distribution	$P(k) \sim k^{-R}$				$[1, 4]$

Our methodology is to perform a statistical analysis based on experiments on UF sparse matrix collection, by plotting the distribution of beneficial matrices to reflect the parameter influence and then defining their ranks accordingly. In Figure 6(a) X-axis gives the intervals of `Ndiags` and `max_RD`, and the distribution of DIA and ELL matrices among the intervals is presented on Y-axis in percentage. When `Ndiags or max_RD` becomes larger, less number of matrices gain performance benefits from DIA or ELL format, which means small `Ndiags` and `max_RD` indicate good performance for DIA and ELL. Similarly, Figure 6(b) shows the beneficial matrix distribution among nonzero ratio intervals, from which we can conclude that large nonzero ratio is good for DIA (less obvious) and ELL formats.

However, we also observe exceptions from these two figures above. For `ER_DIA` parameter there are still nearly half of matrices benefiting from DIA format even when the nonzero ratio is no more than 50%. To improve the combinational accuracy of the rules, we introduce two more parameters – `NTdiags_ratio` and `var_RD`. We firstly define a new type of diagonal – "true diagonal", to represent one occupied mostly with non-zeros. A "true diagonal" features continuous X access pattern, minor part of zero-padding, and writing Y-vector only once, which will show competitive SpMV performance. We use the ratio of "true diagonals" out of all diagonals (`NTdiags_ratio`) as another parameter. The beneficial DIA matrix distribution is shown in Figure 6(c), which indicates DIA also gains higher performance with larger `NTdiags_ratio` just as `ER_DIA` parameter. However, `NTdiags_ratio` shows more obvious rules than `ER_DIA` so that it captures application features more accurately. Similarly, we add `var_RD` parameter for ELL format. As we know, if a sparse matrix has a significantly variable number of non-zeros per row, its SpMV performance degrades due to a lot of zero-padding operations. So we introduce `var_RD` to represent the variation of the number of non-zeros per row. Figure 6(d) indicates that small `var_RD` value is good for ELL SpMV performance. Finally, we learn from [36] that COO format gains good performance on small-world network. Thus, we choose power-law distribution $P(k) \sim k^{-R}$ as its criterion. Figure 6(e) depicts the distribution on `R`, of which the interval $[1, 4]$ is preferred by COO matrices.

So far, we extract 11 feature parameters to abstract sparse matrix structure and draw several rules to achieve higher SpMV performance with a certain format. However, it is still far from predicting the optimal format and implementation only based on rules with the thresholds generated from the simple observations in Table 2. First, for each format there are multiple parameters which are not completely orthogonal to each other. That makes it hard to set accurate threshold values for decision making. Second, from Figure 6, there are quite a few exceptions to the simple rules, which may bring too many wrong predictions. Last, a careful balance is needed for determining the threshold values. Since a rule has both advantages and disadvantages, although it can predict accurately for its favored format, it might hurt the prediction accuracy of other formats. For example, if we conclude a rule of $var\_RD < 0.5$ from Figure 6(d), even when the matrix satisfies this rule SpMV performance with ELL format still may be lower than CSR format. In this case, it is necessary to balance the overall accuracy of four formats rather than only one. Therefore, we introduce a data mining method based on the parameters in Table 2 to build a learning model, with which SMAT can generate practical and reliable prediction.

5. Machine Learning Model

In this section, we discuss details of the data mining method used to the generate learning model and search for optimal implementation from the kernel library.

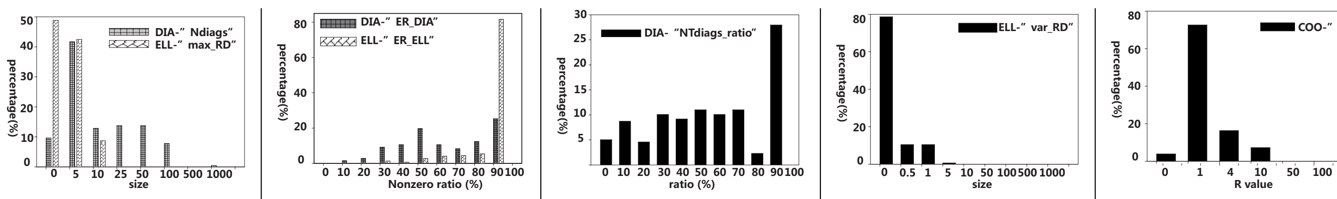
5.1 Model Generation

We define an attribute collection {`M, N, Ndiags, NTdiags_ratio, NNZ, max_RD, var_RD, ER_DIA, ER_ELL, R, Best_Format`} for each matrix, where "Best_Format" is the target attribute. Through training with 2055 UF matrices, we get accurate parameter values of all training matrices. For example, matrix t2d_q9 (in training set) has a record of parameter values as {9801, 9801, 9, 1.0, 87025, 9, 0.35, 0.99, 0.99, inf, DIA}, where "inf" means the power-law distribution cannot calculate the R value from this matrix since it has no attribute of scale-free network. All of these records together constitute the matrix feature database.

As the target attribute, "Best_Format" is used to classify possible candidates (DIA, ELL, CSR, COO) for a particular sparse matrix. Thus, a mapping from the parameters set to the "Best_Format" needs to be built. This falls into a classification problem that can be solved by data mining tools. The formulation of mapping is described in Equation 1, where $\vec{x_i}(i = 1, \ldots, n)$ represents the set of parameters of a sparse matrix in the training set, and $\vec{TH}$ stands for the set of thresholds for each attribute. $C_n(DIA, ELL, CSR, COO)$ represents one of the four categories. With the help of data mining method, $\vec{TH}$ is generated along with the decision tree.

$$f(\vec{x_1}, \vec{x_2}, \ldots, \vec{x_n}, \vec{TH}) \rightarrow C_n(DIA, ELL, CSR, COO) \qquad (1)$$

SMAT system uses C5.0 [2] data mining tool to generate the decision tree from the training matrix set. With its user-friendly design, little knowledge besides the feature database is needed to generate the model. The tool extracts informative patterns from the input data and outputs the learning model.



(a) Ndiags and max_RD (b) ER_DIA and ER_ELL (c) NTdiags_ratio (d) var_RD (e) R

Figure 6. The distribution of the beneficial matrices (gain SpMV performance benefit using the corresponding formats) with different parameter values, where Y-axis shows the percentage of matrices falling into each parameter value interval.

As we know, C5.0 can generate either decision tree or ruleset pattern. Though the two patterns represent similar information, we eventually chose ruleset for two reasons. First, ruleset classifiers are most likely more accurate than decision tree. Although this higher accuracy increases the execution time of the data mining method, it only impacts the off-line stage of SMAT and is reusable after one time decision. Second, it is convenient to convert the rules to IF-THEN sentences, so ruleset is more straightforward to integrate with our system. In order to reduce model inaccuracy, we guide C5.0 to generate a confidence factor along with each rule. The confidence factor is the ratio of the number of correctly classified matrices to the number of matrices falling in this rule, the range of which is between 0 and 1. The larger the confidence factor is, the more reliable the rule is. We will show the role of this confidence factor in Section 6.

Using a matrix feature database, SMAT generates a learning model in ruleset pattern with a confidence factor. The usage of the ruleset model will also be discussed in Section 6.

5.2 Kernel Searching

Previous literature [10, 11, 22, 31, 35] discussed architectural optimization strategies (such as blocking methods, software prefetching, SSE intrinsic usage, multi-thread, etc.) for specific architectures. In SMAT a collection of strategies are grouped together as a large kernel library in Figure 4. It is important to narrow down the optimization implementations of the library to find potential optimal kernels on a given architecture. SMAT can leverage existing auto-tuning tools to facilitate search in off-line stage.

Kernel searching is conducted with a performance record table and scoreboard algorithm. We run all possible implementations (up to 24 in current SMAT system) and record corresponding performance in a table. The implementations are arranged in a specific order in this table, and each performance number in a record can be indexed by all of the optimization strategies it used. Based on the performance table, a scoreboard is created to find the most efficient strategy according to SpMV behavior on the target architecture. Beginning from the implementation with a single optimization method, if it shows performance gain compared with basic implementation, the corresponding optimization method is marked as 1 on the scoreboard, otherwise it is marked by -1. When performance gap between two implementations is less than 0.01, we consider this optimization strategy showing no effect on this architecture and neglect it by default. In this way, an implementation with multiple optimization strategies should compare its performance with those that have just one less optimization strategy. Thus, we eventually obtain the score of each optimization strategy, and then the score of each implementation can be calculated by summing the scores of strategies used in it. Through these two algorithms, the implementation with the highest score is considered to be the optimal one for the corresponding format on the architecture. The optimal kernels will be invoked when the data mining method builds the learning model and the execute-measure module runs.

6. Runtime Process

In this section, we discuss runtime components (labeled by red arrows in Figure 4) of SMAT. This procedure depends on a black-box machine learning model with details depicted in Figure 7. We also discuss several optimization methods to reduce runtime overhead.

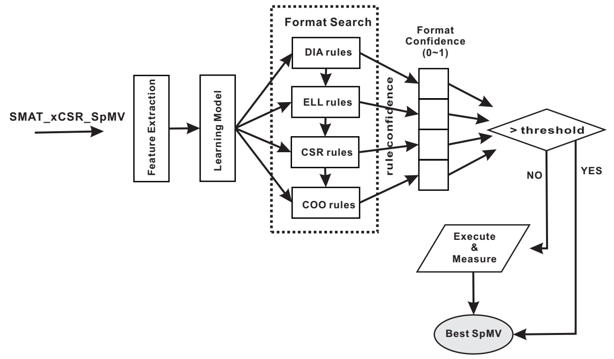


Figure 7. Runtime procedure of SMAT

Feature Extraction In this component the parameters listed in Table 2 are calculated without actually running SpMV in two separated steps: First, we extract diagonal information for DIA format. For reducing the number of times we traverse the whole matrix, we count the diagonals and nonzero distribution together. Thus, we extract the parameter values of DIA, ELL, and CSR formats in this step. Second, we need to evaluate parameter R for COO format, which will spends non-trivial time due to the heavy computation of power-law distribution. From the feature extraction process, we learn that the two separated steps are independent, which will be useful to accelerate the runtime process as described in the next paragraph.

Rule Tailoring and Grouping C5.0 generates tens of rules in the learning model. As many of these rules are suitable for only a small group of matrices and inaccuracy, we re-order them according to corresponding estimated contribution (a concept from C5.0) to the overall training set. That is, rules reducing error rate the most appear first, and rules contributing the least appear last. Then we tailor rules top-down until the subset of rules achieve predictive accuracy close to whole ruleset (1% accuracy gap is acceptable). From our experiments, we choose rules No.1-15 on Intel platform, which decrease the error rate to 9.6%, which is very close to the result 9.0% achieved by the whole ruleset of 40 rules.

After finishing the rule tailoring step, we assign the rules to different format groups. According to the rule confidence factor generated with learning model, we pick the largest one within the same format group as format confidence factor. The format confidence factor reflects the reliability when a corresponding format is chosen by the model, and we compare it with the threshold to decide the reliability of the prediction.

Since it is quite time-consuming when executing all steps in order, we employ an optimistic strategy to accelerate this process. The basic idea is that we can make the decision and do not need to go through all rules if the prediction has sufficiently high confidence. For example, after the feature parameters are calculated in the first step, the learning model will check DIA rule group and get DIA confidence. If DIA confidence is larger than the threshold, DIA format and its optimal implementation will be output as the result. With this strategy, the format group order should be carefully set. As we know, DIA achieves the highest performance (Figure 3) once a matrix satisfies specific conditions, so we arrange DIA rule group in the first place for high performance. ELL takes the second place for its regular and easy-to-predict behavior, which is followed by CSR as the parameters needed have been calculated already (with DIA and ELL), and COO takes the last place. With this order, the prediction procedure reduces quite a lot of time while retaining prediction accuracy.

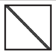
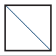
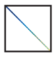
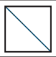
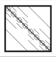
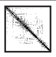
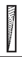
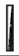
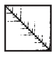
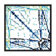
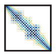
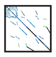
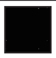
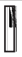
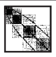
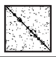
No.	Graph	Name	Dimensions	Nonzeros (NNZ / M)	Application area
1		pcrystk02	14K×14K	491K (35)	duplicate materials problem
2		denormal	89K×89K	623K (7)	counter-example
3		cryg10000	10K×10K	50K (5)	materials problem
4		apache1	81K×81K	311K (4)	structural problem
5		bfly	49K×49K	98K (2)	undirected graph 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

Figure 8. Representative matrices

7. Experiments and Analysis

In this section, we evaluate the performance of SMAT by running benchmark matrices and a real application, followed by accuracy and overhead analysis.

7.1 Experimental Setup

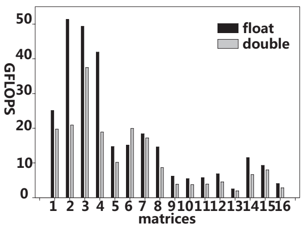
Platform The experiments are conducted on two x86 multi-core processors: One is a Intel Xeon X5680 with 12 3.3GHz cores, 12MB shared last-level cache, and 31GB/s memory bandwidth; and the other one is AMD Opteron 6168 with 12 1.9GHz cores, 12MB shared last-level cache, and 42GB/s memory bandwidth. All executions are configured to run with 12 threads.

Benchmark The off-line data mining component processes a training set of 2055 sparse matrices randomly chosen from UF sparse matrix collection. Excluding the training matrices, we use the rest 331 matrices for performance evaluation, and 16 matrices (described in Figure 8) of them for plotting visualized graphs of their contents.

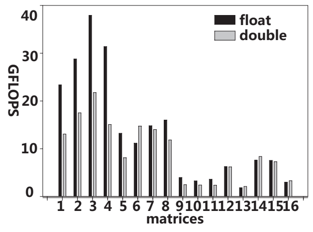
Application To demonstrate the usability of SMAT, we integrate it with Hypre [14], a scalable linear solver library from LLNL. Wherein, Algebraic Multigrid (AMG) is used as a preconditioner such as conjugate gradients to solve large-scale scientific simulation problems on unstructured grids. As noted in Section 1, the sparse matrices show dynamic patterns to some extent among different levels. We anticipate it will benefit from SMAT's co-tuning approach.

7.2 Performance

We measure SpMV performance in GFLOPS, which is a ratio of the number of floating-point operations to execution time. Figure 9 plots GFLOPS of both single- and double-precision on the two platforms. X-axis represents the matrix number as listed in Figure 8 and Y-axis gives the performance of SpMV generated by SMAT. The highest SMAT performance on Intel platform is 51GFLOPS with an efficiency of 32% in single-precision and 37GFLOPS (23%) in double-precision. On AMD platform, SMAT achieves 38GFLOPS (42%) in single-precision and 22GFLOPS (24%) in double-precision. This peak performance is even higher than the reported maximum performance of 18 GFLOPs on GPU [7].



(a) Intel



(b) AMD

Figure 9. SMAT performance in single- and double-precision on the two platforms

Figure 9 shows up to 5 times performance variation among the matrices, which proves it is worth adopting SMAT in different applications owing to its adaptability to diverse sparse matrices. For matrices that have affinity to DIA, ELL or COO format (No.1-8 and No.13-16), the corresponding SpMVs achieve higher performance than those in CSR format (No.9-12). The performance gap indicates that it is meaningful to implement a high performance SpMV library being aware of sparse structures (applications).

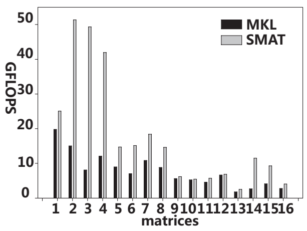
Moreover, we give performance comparison between SMAT and Intel MKL multi-threaded library in Figure 10 in single- and double-precision on Intel platform. MKL performance shown in this figure is the maximum performance number of DIA, CSR, and COO SpMV functions in this library. Compared with MKL, SMAT obtains the maximum speedup of 6.1 times in single-precision and 4.7 times in double-precision. Although this figure only shows 16 representative matrices, we do collect experimental data for all 331 matrices and the average speedup over MKL is **3.2** times in single-precision and **3.8** times in double-precision. SMAT shows big advantages in performance due to the following reasons. First, we take advantage of optimized SpMV implementations us-

Table 3. Analysis of SMAT

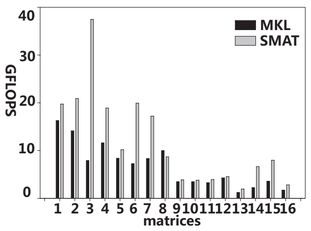
Matrix Number	Matrix Name	Model Prediction Format	Execution	SMAT Prediction Format	Actual Best Format	Model Accuracy	SMAT Overhead (times of CSR-SpMV)
1	pcrystk02	DIA	-	DIA	DIA	R	2.28
2	denormal	DIA	-	DIA	DIA	R	2.09
3	cryg10000	DIA	-	DIA	DIA	R	2.11
4	apache1	DIA	-	DIA	DIA	R	1.94
5	bfly	ELL	-	ELL	ELL	R	1.18
6	whitaker3_dual	ELL	-	ELL	ELL	R	4.89
7	ch7-9-b3	ELL	-	ELL	ELL	R	2.25
8	shar_te2-b2	ELL	-	ELL	ELL	R	2.24
9	pkustk14	$confidence < TH$	CSR+COO	CSR	CSR	W	16.39
10	crankseg_2	$confidence < TH$	CSR+COO	CSR	CSR	W	16.28
11	Ga3As3H12	$confidence < TH$	CSR+COO	CSR	CSR	W	16.2
12	HV15R	$confidence < TH$	CSR+COO	CSR	CSR	W	15.43
13	europe_osm	COO	-	COO	COO	R	2.3
14	D6-6	COO	-	COO	COO	R	5.79
15	dictionary28	COO	-	COO	COO	R	2.05
16	roadNet-CA	COO	-	COO	COO	R	2.38

"R" and "W" represent Right and Wrong prediction respectively.

ing SIMDization, branch optimization, data prefetch, and task parallelism policy. Second, as SMAT can determine the optimal format and implementation, the highest SpMV performance achieved by these optimization methods is eventually revealed. We also observe one matrix (No.8) shows lower performance than MKL due to inaccuracy of SMAT. We will analyze accuracy and prediction overhead of SMAT in the next section.



(a) float



(b) double

Figure 10. The performance of SMAT V.S. MKL

7.3 Analysis

Accuracy Table 3 shows the details of on-line decision making process and its accuracy. When a sparse matrix is input to SMAT system, the learning model performs prediction on-line and outputs the predicted format (noted by "Model Prediction Format"). However, there are some exceptions that the learning model cannot predict their formats confidently. SMAT actually executes SpMV kernels for once and measures the performance for part of formats, which is noted as "Execution" in this table. For example, the execution process runs CSR- and COO- based SpMVs to determine the final format for matrix No.9-12. In this table "Best Format" represents the best result of exhaustive search. Though the learning model cannot predict correctly in some cases due to relatively intricate features of CSR as the most general format, SMAT still can obtain good results on the 16 matrices. For all 331 matrices, the accuracy is 92% (SP) and 82% (DP) on Intel platform, and 85% (SP) and 82% (DP) on AMD platform respectively.

Prediction Overhead Table 3 also shows the "SMAT Overhead" in the last column, which is represented by ratio of the overall execution time to the basic CSR-SpMV execution time. In most cases, the overhead is no more than 5 times, which is very competitive comparing to existing auto-tuning tools [11, 17, 17, 21, 29, 31, 35, 36] and practical for real applications. But when the learning model fails to get a confident prediction, SMAT overhead increases to about 15 times. This is acceptable when an application executes an SpMV kernel hundreds of times. In fact, compared with the overhead of existing auto-tuners – OSKI (40 times), clSpMV (1-20 times), the overhead of SMAT appears acceptable.

As a straightforward way to search for the optimal result, one option is to run SpMV kernels for all formats one by one. Despite plenty of implementation variants of one format, the simplest search across basic implementations of the four formats takes more time than SMAT. The overhead of this search method comes from format conversion and SpMV execution. For example, the conversion from CSR to ELL consumes 39.6 times of CSR-SpMV for the No.11 matrix. The overhead of simple search reaches up to 45 times even if only four implementations are explored, which is much higher than the case (16.2) in Table 3. Remember that SMAT is only a proof of concept system, and its adoption of reusable machine learning model makes it feasible to extend for more formats and implementations.

7.4 SMAT-based AMG

Considering a problem of solving equation $Au = f$ with certain precision criterion, where A is a large sparse matrix, u and f are dense vectors, Hypre AMG solves this by building N levels of virtual grids with a series of grid operators $(A_0, ...A_{N-1})$ and grid transfer operators $(P_0, ...P_{N-2})$ in a setup process. Figure 11 illustrates a typical V-cycle in AMG. Apparently, P-operators perform SpMV between adjacent grids, and A-operators are used to do relaxations like Jacobi and Gauss-Seidel methods with SpMV kernel, too. Without a doubt, SpMV consumes most of the V-cycle's execution time. We observe an interesting phenomenon that the two series of sparse matrices dynamically show different sparse features from the original input matrix A. Thus, SMAT is useful on determining the optimal format and implementation for the operators at each level, and improving the overall performance.

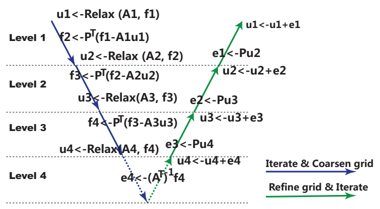


Figure 11. V-cycle involved with SpMV kernels in AMG solver.

We perform experiments on *cljp* and *rugeL* coarsen methods in the setup process to generate A-operators with different structures. The input matrices are generated by 7-point and 9-point Laplacian methods, respectively. Instead of using CSR format all the time, SMAT chooses DIA format for A-operators at the first few levels, and ELL format for most P-operators. To do this, we simply replace the SpMV kernel codes with SMAT interfaces with no changes to the original CSR data structure in Hypre. As shown in Table 4, the solving part of SMAT AMG achieves more than 20% performance gain than Hypre AMG.

Table 4. SMAT-based AMG execution time (millisecond)

Coarsen	Rows	Hypre AMG	SMAT AMG	Speedup
cljp_7pt_50	125K	3034	2487	1.22
rugeL_9pt_500	250K	388	300	1.29

8. Related Work

There has been a flurry of work related to optimizing sparse matrix vector multiplication performance. While we could not hope to provide a comprehensive set, we attempt to contrast our work with several key representatives.

Storage Format Basically, SMAT discovers optimal storage formats for sparse matrices. This idea is reflected in previous work. By far, more than ten storage formats [7, 21, 25, 28, 29, 31] have been proposed, most of which are either application-specific or architecture-specific so that their applicable domains are limited. Recently, several hybrid storage formats have been developed. Bor-Yiing Su et al. [29] proposed a Cocktail format that splits a matrix and represents these sub-matrices in different formats according to relative advantages of different sparse matrix formats. Similarly, on NVIDIA GPU CuSparse [23] stores one sparse matrix with a hybrid format HYB, which is a combination of two basic formats. These new hybrid storage formats show much better performance than conventional formats for certain sparse matrices. The major difference from SMAT is that the hybrid formats are determined statically and not applicable to dynamic sparse structures as shown at different levels of AMG solver. In the other aspect, researchers are developing relatively flexible formats. Richard Vuduc et al. [31] improved BCSR format to VBR to explore dense blocks with different sizes. Kourtis et al. [21] proposed CSX to exploit dense structures not limited to dense blocks, but also 1-dimension bars and dense diagonals by compressing metadata. However, the process of searching sub-blocks costs too much to be conducted on-line. In contrast, in terms of storage format optimization, SMAT selects the optimal format from the existing formats, instead of designing a new one. The machine learning model makes an on-line decision feasible. It is possible to add new formats by extracting novel parameters and integrating its implementations in kernel library in SMAT.

Auto-tuning Approach For developing a domain-specific performance-critical library, auto-tuning approach is promising to resolve both performance and portability problems. There are several successful auto-tuning libraries, such as ATLAS [33], FFTW [15], SPIRAL [24], and OSKI [31], widely used in scientific computing area. Specifically for SpMV, auto-tuning techniques are actively investigated. Eun-jin Im et al. [19] created BCSR format to better develop the performance of dense blocks in a sparse matrix. Richard Vuduc et al. [31] built OSKI to tune the block size for a matrix in BCSR or VBR formats. Williams et al. [35] deployed a hierarchical strategy to choose the optimal architectural parameter combinations. Choi et al. [11] implemented Blocked Compress Sparse Row (BCSR) and Sliced Blocked ELLPACK (SBELL) formats on Nvidia GPUs, and tuned the block sizes of them. X.Yang et al. [36] proposed a mixed format and automatically chose the partition size for each format with the help of a model. A common feature of these auto-tuning designs is to focus on implementation tuning on diverse processing architectures only for a single pre-defined storage format. In addition to architectural auto-tuning, SMAT extends to cooperatively tune storage formats by extracting key performance parameters from input sparse matrices. In fact, algorithm and architecture co-tuning was advocated in PetaBricks compiler [4]. Our work further proves the value of auto-tuning techniques.

Prediction Model A core component of SMAT is the machine-learning based prediction model for determining the optimal format and implementation. It is a common strategy to apply prediction model in auto-tuning approach. ATLAS [33] performed an empirical search to determine optimal parameter values bounded by architecture features. But an actual run of generated code is needed to measure and record its performance to choose the best implementation. This empirical search has been proven efficient to generate high quality BLAS codes, although the search process takes a lot of time. Recently, many arising auto-tuners adopt model-driven method such as [11, 21, 29, 31, 36] without the need of actually running the code. Though the model-driven method decreases the prediction time, the generated code performance is considered lower than empirical search in most cases. Kamen Yotov et. al. [37] did experiments on ATLAS and got this conclusion on ten platforms. Our SMAT system combines learning model and empirical search invoked rarely that ensures the code performance and reduces prediction time at the same time. Although clSpMV [29] also used a prediction model to tune its Cocktail format, there are certain crucial differences from SMAT. First and foremost, in on-line decision making stage, clSpMV uses the maximum GFLOPS measured in offline stage. Unfortunately, as our experiments on UF collection shows (see Table 1 and Figure 3) the maximum performance of one format is not representative enough to reflect the SpMV performance of all the matrices suitable in this format. It is more accurate to use the features of each input matrix to determine its own best format rather than using a single maximum performance for each format. Second, we extract more features from real matrices in UF collection, which can feed more training data to data mining tools so as to generate more reliable rules for the learning model. W. Armstrong et al. [5] uses reinforcement learning to choose the best format, but users should decide the factor values which influence the accuracy of the learning model. SMAT system is more convenient to automatically generate the model and still achieve similar prediction accuracy.

9. Conclusion

In this paper, we propose SMAT, an input adaptive SpMV auto-tuner, which encompasses several statistical and machine learning techniques to enable both application- and architecture-dependent, incremental model training and black-box performance prediction. In particular, we provide a unified interface to eliminate tedious work on determining optimal formats and implementations for diverse sparse matrices. Due to cooperatively auto-tuning with both algorithms and architectures, SMAT achieves impressive performance of up to 51 (38) GFLOPS in single-precision and 37 (22) GFLOPS in double-precision on Intel (AMD) multi-core processor. The average speedup is above 3 times over Intel MKL sparse library. SMAT is also used to improve the performance of algebraic multi-grid algorithm from Hypre sparse linear system solver by about 20%. Our work suggests that algorithm and architecture co-tuning is a promising approach for developing domain-specific auto-tuning libraries or tools.

Acknowledgments

We would like to express our gratitude to Yin Li from Intel China and Dr. Trishul Chilimbi for helping us polishing this paper. This work is supported by National Natural Science Foundation of China (61272134, 61033009, 61003062, 60925009), the National Basic Research Program of China (973 Program) (2011CB302502, 2012CB316502), and the joint lab of Institute of Computing Technology and Intel.

References

[1] *Intel Math Kernel Library*. URL `http://software.intel.com/en-us/intel-mkl`.

[2] *Data Mining Tools See5 and C5.0*. URL `http://www.rulequest.com/see5-info.html`.

[3] M. D. Adam Hill. The international thermonuclear experimental reactor. Technical report, 2005.

[4] 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*, PLDI '09, pages 38–49. ACM, 2009. ISBN 978-1-60558-392-1.

[5] W. Armstrong and A. Rendell. Reinforcement learning for automated performance tuning: Initial evaluation for sparse matrix format selection. In *Cluster Computing, 2008 IEEE International Conference on*, pages 411 –420, 2008.

[6] S. Balay, J. Brown, , K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.3, Argonne National Laboratory, 2012.

[7] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.

[8] G. Belter, E. R. Jessup, I. Karlin, and J. G. Siek. Automating the generation of composed linear algebra kernels. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 59:1–59:12, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-744-8.

[9] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the seventh international conference on World Wide Web 7*, WWW7, pages 107–117. Elsevier Science Publishers B. V., 1998.

[10] A. Buluc, S. Williams, L. Oliker, and J. Demmel. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 721–733. IEEE Computer Society, 2011.

[11] 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*, PPoPP '10, pages 115–126. ACM, 2010.

[12] T. A. Davis. The university of florida sparse matrix collection. *NA DIGEST*, 92, 1994.

[13] R. Falgout. An introduction to algebraic multigrid computing. *Computing in Science Engineering*, 8(6):24 –33, nov.-dec. 2006. ISSN 1521-9615.

[14] R. D. Falgout and U. M. Yang. hypre: a library of high performance preconditioners. In *Preconditioners, Lecture Notes in Computer Science*, pages 632–641, 2002.

[15] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[16] D. Grewe and A. Lokhmotov. Automatically generating and tuning gpu code for sparse matrix-vector multiplication from a high-level representation. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 12:1–12:8. ACM, 2011.

[17] J. Harris. poski: An extensible autotuning framework to perform optimized spmvs on multicore architectures. 2009.

[18] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An overview of the trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, Sept. 2005. ISSN 0098-3500.

[19] E. jin Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications*, 18:2004, 2004.

[20] M. F. Khairoutdinov and D. A. Randall. A cloud resolving model as a cloud parameterization in the ncar community climate system model: Preliminary results. *Geophys. Res. Lett.*, 28(18):3617C3620, 2001.

[21] K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris. Csx: an extended compression format for spmv on shared memory systems. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 247–256. ACM, 2011.

[22] K. Nagar and J. Bakos. A sparse matrix personality for the convey hc-1. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 1 –8, may 2011.

[23] *CUDA CUSPARSE Library*. NVIDIA, 2010.

[24] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[25] Y. Saad. Sparskit : a basic tool kit for sparse matrix computations. *Technical Report*, 1994.

[26] N. Spirin and J. Han. Survey on web spam detection: principles and algorithms. *SIGKDD Explor. Newsl.*, 13(2):50–64, May 2012. ISSN 1931-0145.

[27] A. Srinivasa and M. Sosonkina. Nonuniform memory affinity strategy in multithreaded sparse matrix computations. In *Proceedings of the 2012 Symposium on High Performance Computing*, HPC '12, pages 9:1–9:8, 2012.

[28] J. P. Stevenson, A. Firoozshahian, A. Solomatnikov, M. Horowitz, and D. Cheriton. Sparse matrix-vector multiply on the hicamp architecture. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 195–204, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1316-2.

[29] B.-Y. Su and K. Keutzer. clspmv: A cross-platform opencl spmv framework on gpus. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 353–364. ACM, 2012.

[30] X. Sun, Y. Zhang, T. Wang, G. Long, X. Zhang, and Y. Li. Crsd: application specific auto-tuning of spmv for diagonal sparse matrices. In *Proceedings of the 17th international conference on Parallel processing - Volume Part II*, Euro-Par'11, pages 316–327, 2011.

[31] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc. SciDAC, J. Physics: Conf. Ser.*, volume 16, pages 521–530, 2005.

[32] R. W. Vuduc and H.-J. Moon. Fast sparse matrix-vector multiplication by exploiting variable block structure. In *Proceedings of the First international conference on High Performance Computing and Communications*, HPCC'05, pages 807–816. Springer-Verlag, 2005.

[33] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-89791-984-X.

[34] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 38:1–38:12. ACM, 2007.

[35] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Comput.*, 35(3):178–194, Mar. 2009.

[36] X. Yang, S. Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on gpus: implications for graph mining. *Proc. VLDB Endow.*, 4(4):231–242, Jan. 2011.

[37] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance blas?, 2005.