Design and Implementation of Adaptive SpMV Library for Multicore and Many-Core Architecture

GUANGMING TAN and JUNHONG LIU, State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, University of Chinese Academy of Sciences
JIAJIA LI, Computational Science and Engineering, Georgia Institute of Technology

Sparse matrix vector multiplication (SpMV) is an important computational kernel in traditional high-performance computing and emerging data-intensive applications. Previous SpMV libraries are optimized by either application-specific or architecture-specific approaches but present difficulties for use in real applications. In this work, we develop an auto-tuning system (SMATER) to bridge the gap between specific optimizations and general-purpose use. SMATER provides programmers a unified interface based on the compressed sparse row (CSR) sparse matrix format by implicitly choosing the best format and fastest implementation for any input sparse matrix during runtime. SMATER leverages a machine-learning model and retargetable back-end library to quickly predict the optimal combination. Performance parameters are extracted from 2,386 matrices in the SuiteSparse matrix collection. The experiments show that SMATER achieves good performance (up to 10 times that of the Intel Math Kernel Library (MKL) on Intel E5-2680 v3) while being portable on state-of-the-art x86 multicore processors, NVIDIA GPUs, and Intel Xeon Phi accelerators. Compared with the Intel MKL library, SMATER runs faster by more than 2.5 times on average. We further demonstrate its adaptivity in an algebraic multigrid solver from the Hypre library and report greater than 20% performance improvement.

CCS Concepts: • **Mathematics of computing → Mathematical software performance**; • **Software and its engineering → Application specific development environments**;

Additional Key Words and Phrases: Sparse matrix vector multiplication, auto-tuning, multicore, machine learning

ACM Reference format:
Guangming Tan, Junhong Liu, and Jiajia Li. 2018. Design and Implementation of Adaptive SpMV Library for Multicore and Many-Core Architecture. *ACM Trans. Math. Softw.* 44, 4, Article 46 (August 2018), 25 pages.
https://doi.org/10.1145/3218823

1 INTRODUCTION

Modern high-performance computing technologies are driven simultaneously by Exascale FLOPs (Exaflops) and data-intensive applications. Although the challenges of these applications are

This work is supported by the National Key Research and Development Program of China (2016YFB0201305, 2016YFB0200504, 2017YFB0202105, 2016YFB0200803, 2016YFB0200300) and the National Natural Science Foundation of China under grant no. 61521092, 91430218, 31327901, 61472395, and 61432018.
Authors' addresses: G. Tan and J. Liu, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China; emails: tgm@ict.ac.cn, liujunhong@ncic.ac.cn; J. Li, Computational Science and Engineering at Georgia Institute of Technology, North Avenue, Atlanta, GA, US; email: jiajiali@gatech.edu.

ACM 0098-3500/2018/08-ART46 $15.00
https://doi.org/10.1145/3218823

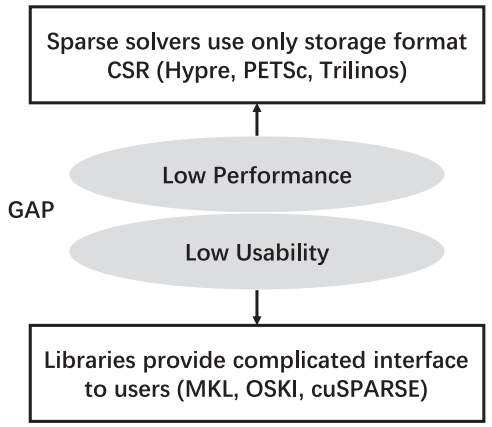


Fig. 1. The dilemma of sparse solvers and libraries.

diverse, they share the requirement for sparse linear system solvers. For example, most execution time is spent solving large-scale sparse linear systems in Exaflops applications, laser fusion in international thermonuclear experimental reactors (ITERs) (Adam Hill 2005), and global cloud systems resolving climate modeling (Khairoutdinov and Randall 2001). Similarly, numerous large-scale graph analysis applications, such as PageRank (Brin and Page 1998; Beamer et al. 2017) and HITS (Spirin and Han 2012), involve sparse matrix solvers to identify relationships. All of these algorithms rely on a core sparse matrix vector multiplication (SpMV) kernel, which consumes a large percent of overall execution time. For example, the algebraic multigrid (AMG) solver (Falgout 2006) from Hypre (Falgout and Yang 2002), an iterative algorithm widely used in laser fusion and climate modeling, reports over 90% SpMV runtime operations of its runtime on Intel Xeon E5-2680.

The diversity of both sparse matrices and processor architectures exacerbates the difficulty of SpMV optimization. In finite element calculations of solid mechanics, the generated sparse matrices have a few diagonals, whereas social network applications can produce sparse matrices with power-law nonzero distributions. Diverse sparse matrix characteristics are generated from both different applications and different stages within a single algorithm, such as the AMG v-cycle. Over the past decades, most studies have focused on developing efficient storage formats specific to either applications or architectures (Kourtis et al. 2011; Bell and Garland 2008; Su and Keutzer 2012; Vuduc et al. 2005; Yang et al. 2011; Sun et al. 2011; Vuduc and Moon 2005; Williams et al. 2009; Buluc et al. 2011; Grewe and Lokhmotov 2011; Srinivasa and Sosonkina 2012; Choi et al. 2010; Nagar and Bakos 2011; Stevenson et al. 2012). Although previous studies adopted hand-tuned libraries or auto-tuning techniques following processor architectures' evolvement (Buluc et al. 2011; Bell and Garland 2008; Kourtis et al. 2011; Williams et al. 2009; Vuduc et al. 2005; Yang et al. 2011; Choi et al. 2010; Su and Keutzer 2012; Byun et al. 2012; Ansel et al. 2009; Belter et al. 2009; Armstrong and Rendell 2008; Liu et al. 2013; Buono et al. 2016; Liu et al. 2018b), a high-performance SpMV library that is adaptive to diverse sparse matrices and processor architectures remains in demand.

Figure 1 shows the current dilemma of sparse matrix computation between basic operation libraries and numerical solvers. Most sparse numerical solvers or applications are not adaptive to diverse sparse matrices, and commonly used solvers, such as Hypre (Falgout and Yang 2002), PETSc (Brown et al. 2012), and Trilinos (Heroux et al. 2005), support only one popular compressed row (CSR) format. Consequently, solvers cannot achieve equal efficiency on diverse sparse matrices due

Table 1. Affiliation of Sparse Matrices in the SuiteSparse Matrix Collection to Their Optimal Formats

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**
counterexample	3	4	1	0	**8**
acoustics	5	0	2	0	**7**
robotics	3	0	0	0	**3**
Percentage	**63%**	**21%**	**9%**	**7%**	**2386**

to their different sparsity patterns. Our experimental evaluation (Table 1 in Section 2) confirms this phenomenon by testing SpMV performance with four matrix formats in the Suit-eSparse matrix collection (Davis and Hu 2011). Compounding the problem, most sparse operation libraries provide complicated interfaces that limit their usability. For example, Intel (2017) and OSKI (Vuduc et al. 2005) libraries provide different interfaces for their supported formats, where users must determine the optimal format and function for their applications. Therefore, it is highly desirable to provide a library that can automatically determine and employ the most suitable sparse matrix format for real-world applications on a given processor architecture.

This article presents the design and implementation of an SpMV library (SMATER) that is co-operatively auto-tuned between algorithms and architectures using our previous state-of-the-art SpMV auto-tuner (SMAT, Li et al. (2013)) and a set of optimized SpMV implementations on various processor architectures as the back-end warehouse.

The proposed SMATER system uses CSR format as the unified interface, since this format is relatively popular and CSR SpMV achieves the best performance in the SuiteSparse (Davis and Hu 2011) sparse matrix collection (Table 1). Users need only prepare sparse matrices in CSR format as input, and SMATER automatically determines the optimal format and implementation based on the sparse matrix structure and the available architecture. SMATER employs auto-tuning based on black-box machine learning, which can be trained by representative features extracted from both sparse matrices and architectural configurations. Once the model is trained on the

target architecture, it is used to evaluate the input sparse matrix during runtime and optimize the format. The back-end warehouse combines the SpMV library (Intel MKL) and our hand-tuned implementation. Other SpMV libraries, such as Kourtis et al. (2011), Williams et al. (2009), Vuduc et al. (2005), Yang et al. (2011), Choi et al. (2010), Su and Keutzer (2012), Byun et al. (2012), and Liu (2015), can be easily included in the back-end warehouse.

SMATER has evolved as an adaptive sparse library from our previously proposed SMAT auto-tuner (Li et al. 2013) by integrating hand-tuned sparse BLAS implementations (spBLAS) as the retargetable back end. The main differences with the previous SMAT auto-tuner and the contributions of this article are as follows:

—We propose a platform-adaptive SpMV library (SMATER) based on auto-tuning techniques for multicore/many-core architectures with a unified interface. The auto-tuner systematically extends the parameter space to be more compatible with blocked sparse formats and many-core architecture configurations. In contrast, SMAT was implemented for only two x86 multicore processors and four storage formats: DIA, ELL, CSR, and COO.
—We design a flexible and extensible retargetable back-end warehouse to provide a unified interface for diverse processor architectures. The warehouse facilitates users to adopt the tuned library to improve output performance with little or no extra programming effort. The spBLAS back-end library offers specific SpMV implementations on multicore CPU, NVIDIA GPU, and Xeon Phi platforms, supplementing SMAT and numerical solvers.
—We implement SMATER and train it using the SuiteSparse sparse matrix collection (Davis and Hu 2011). Experiments show SMATER adaptivity on two x86 multicore processors (Intel Xeon E5-2680 and AMD Opteron 6168) and two many-core coprocessors (NVIDIA Tesla K20 and Intel Xeon Phi 3120A).

With sparse matrices selected from various application areas, SpMV kernels generated by SMATER achieve impressive performance, up to 127GFLOPS and 64GFLOPS for single precision (SP) and double precision (DP), respectively. On many-core coprocessors, SMATER achieves the highest performance of 43GFLOPS (SP) and 26GFLOPS (DP). Compared to SELL-C-σ (Kreutzer et al. 2014; Anzt et al. 2014), SMATER achieves an average 2.39 and 2.41 speedup for SP and DP, respectively, on Intel Xeon E5-2680 v3, and 2.41 and 2.58 on NVIDIA GPU K20 for the best C and σ. Compared to cuSPARSE, SMATER achieves an average 2.0 speedup. Compared with Intel MKL, SMATER achieves 2.87 speedup and 2.83 speedup on average for SP and DP, respectively, on Intel Xeon Phi 3120A. We also evaluate the adaptability of SMAT in the AMG algorithm of the sparse solver Hypre (Falgout and Yang 2002), and the results show that the performance can be improved above 20%.

2 MOTIVATION

Sparse matrices are generally stored in compressed format, and since the sparse format is closely related to compression effectiveness and SpMV performance, more than 15 compressed formats (Bell and Garland 2008; Kourtis et al. 2011; Vuduc et al. 2005; Saad 1994; Su and Keutzer 2012; Stevenson et al. 2012; Yan et al. 2014; Liu and Vinter 2015a, 2015c) have been developed. Unfortunately, it is nontrivial to determine the most suitable sparse matrix format for a given architecture, particularly when nonzero element distribution is random. Extracting sufficient performance characteristics from various sparse matrices and architectures is challenging when building a model that can efficiently and accurately determine the optimal format during runtime. However, overcoming this challenge will provide significant performance benefits for real applications.

To explore sparse characteristics and assess potential benefits arising from tuning storage formats, we perform experiments using the SuiteSparse matrix collection (Davis and Hu 2011), which

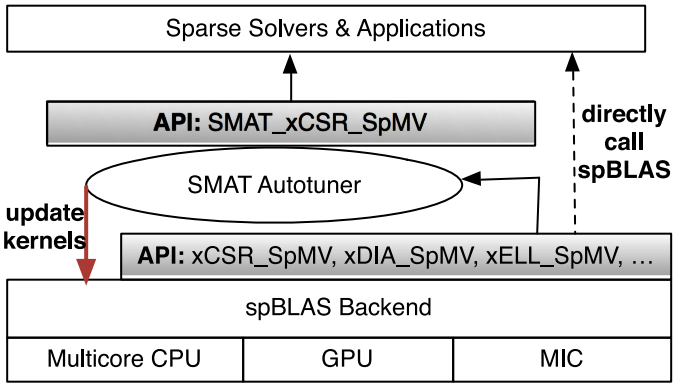


Fig. 2. SMATER framework.

consists of over 2,000 sparse matrices from extensive real-world applications. For simplicity and to enhance accuracy, we study 2,386 sparse matrices, excluding very small ($<100 \times 100$) or complex matrices and considering four sparse formats: CSR, COO, DIA, and ELL. We use naive SpMV implementations of each format to test the performance on Intel Xeon E5-2680. Table 1 shows the number of matrices in which a particular format achieves better SpMV performance than others. For example, in the graph domain, 187 matrices achieve the highest SpMV performance using CSR format, and 114 using COO format, whereas in the theoretical quantum chemistry domain, 26 of 47 matrices obtain the best SpMV performance using DIA format. The bottom row (Percentage) shows the calculated proportions of all matrices affiliated with each format. The number of matrices in each domain is shown in column 6 (Total). Sparse matrices from the same or different domains show distinct affinities to specific formats, highlighting both the necessity and difficulty in determining the optimal format for a given sparse matrix.

3 SMATER FRAMEWORK

Figure 2 shows the major SMATER components, including the SMAT auto-tuner and retargetable back-end library (spBLAS). Two modes are available to support numerical solvers and applications, using auto-tuner or spBLAS interfaces, respectively. The SMATER workflow proceeds as follows:

- —At system initialization, SMATER can select online auto-tuning or directly call spBLAS APIs.
- —Once the online stage of the auto-tuner is activated, SMATER offers the optimal storage format for the sparse matrix.
- —If the optimal format is not CSR, the spBLAS library transforms the matrix storage format as appropriate and implements SpMV computation.

SMATER is very flexible for different application types. Generally, numerical solvers call an SpMV computational kernel hundreds of times during their computation. Hence, it is beneficial to enable the auto-tuner model to choose the optimal storage format, because execution time saved by the optimal format is much larger than that of the one-time format transformation. On the other hand, for applications that only call SpMV kernels several times, we can directly call the spBLAS library to implement the xCSR_SpMV computation, avoiding format transformation. Besides, some applications compute SpMV on sparse matrices with similar patterns, indicating identical optimal storage formats. Therefore, we do not need to choose the optimal format by auto-tuner for every SpMV; rather, we determine the optimal format once and reuse it in the future. These three aspects

enable SMATER to be efficient for different applications. SMATER always hides the complexity of tuning on diverse processor architectures for all cases.

If the SMAT auto-tuner mode is selected, users need only input the sparse matrix in the popular CSR format, and SMAT will select the optimal storage format. Given a sparse matrix in CSR format, SMAT extracts its sparse features first. The feature values train a prediction model, which is used to predict the optimal format and implementation provided by the back-end spBLAS library. A confidence value is set to rule this prediction model. When the confidence value is larger than the threshold, the optimal SpMV format and optimal SpMV implementation have been achieved; otherwise, an execution measure process is triggered to test possible solutions and output the one with the highest performance. Meanwhile, the best-generated implementation will update the corresponding SpMV kernel in the back-end library.

The spBLAS back-end library offers specific SpMV implementations for multicore CPUs, NVIDIA GPUs, and Xeon Phi, providing support for SMAT and numerical solvers. On the NVIDIA GPU platform, the spBLAS library optimizes SpMV by encapsulating the cuSPARSE library, whereas on the Xeon Phi platform, spBLAS fully captures the advantages of multithreading and SIMDization from our implementations and Intel MKL library, where the offload model serves as an acceleration device.

As an auto-tuning-based library, SMATER exhibits reusability, extensibility, and portability. First, on a specific architecture, the auto-tuner generates the learning model in the offline stage, and then uses the model repeatedly for different input matrices. Second, SMATER extends well on diverse architectures due to the architecture features being considered. However, the difference from the existed auto-tuning systems is that SMATER employs the performance of retargetable back-end implementations to reflect the influence of architecture features—instead of using accurate architecture features. This is advantageous in two ways. First, it makes SMATER compatible with new architectures, which is the most significant feature of auto-tuning systems and is the biggest difference from other libraries. Second, it more accurately and simply conducts the learning model using only one performance value instead of numerous architecture features. Ultimately, this narrows the number of factors when considering both application and architecture features.

4 AUTO-TUNER

The auto-tuner leverages machine learning to choose the optimal storage format. This section discusses the main SMAT auto-tuner concepts with the additional functions. Figure 3 shows the architectural framework integrated with the new retargetable spBLAS back end. We consider architecture (i.e., register, cache, TLB sizes, prefetching, and threading policy) and application (i.e., matrix dimension, diagonal occurrence, and nonzero distribution, described in Section 4.1) parameters to optimize SpMV and evaluate its performance on diverse real-world sparse matrices. The large-parameter space makes it impossible to exhaustively search the optimal sparse format and optimization method for a specific architecture. Therefore, we use machine learning to train a tree-based model to assist decision making.

Since sparse matrices have different SpMV behaviors, we first select a set of features to represent different sparse matrices. Then, we test the feature values on the SuiteSparse matrix collection (Davis and Hu 2011) to build a feature database, where each matrix generates a feature record. Each record is labeled by the format that achieves the highest SpMV performance. The decision tree method is used to train a tree-based model on the feature database to predict the optimal sparse format.

However, a predicted format is insufficient because SpMV can have quite a few implementations using different optimization methods and setting different architecture parameters. Considering the optimization methods (i.e., cache blocking, register blocking, and parallelization), a large

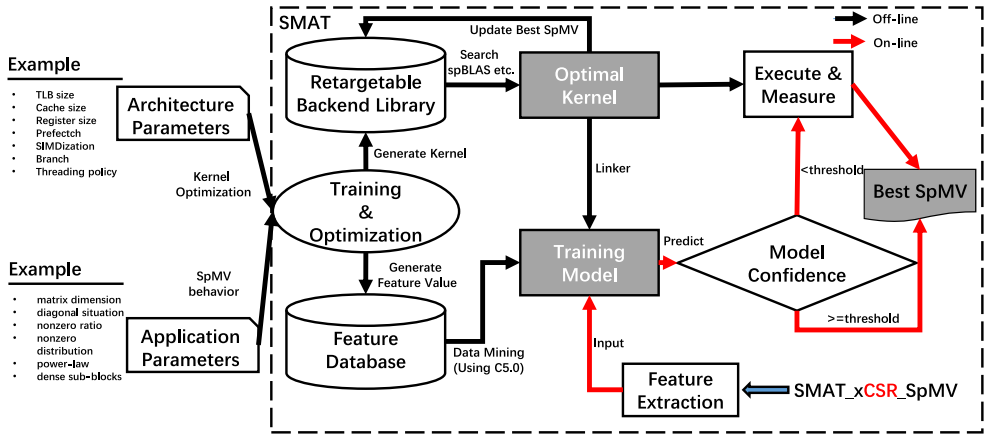


Fig. 3. SMAT auto-tuning system with a self-updating retargetable back-end library.

number of SpMV implementations are built as the back-end warehouse, including spBLAS. Given an architecture, we test a set of benchmarks and use a scoreboard algorithm to mark the best SpMV implementation for each format. We call this process kernel searching, which is particularly useful when multicore CPU and many-core accelerators are both considered. Using kernel searching, our trained model reflects both application features with architecture characteristics. Both kernel searching and model training are depicted as black arrows in Figure 3 as the offline part, whereas the online part of SMAT (red arrows in Figure 3) shows the extracted features of the input sparse matrix. After we extract these features, the trained model predicts the optimal format affiliated with a specific confidence value. If the confidence value is acceptable, we output this SpMV kernel as the optimal one; otherwise, several candidate kernels are tested to determine the optimal one.

We improve the SMAT framework by adding a feedback path between online SMAT and the back-end warehouse. If an input sparse matrix needs to execute the candidate SpMV kernels to determine which have the highest performance, feedback of the execution performance is sent to our back end. The scoreboard compares and records these performance numbers, which may lead to an update of the selected optimal implementation. Adding this feedback path allows the input sparse matrices, particularly those with no obvious features, to help improve future SMAT accuracy.

4.1 Feature Selection

We briefly present the method to extract sparse structure features, which is described in more detail in our previous paper (Li et al. 2013). To build an application-aware auto-tuner, we extract features from more than 2,000 sparse matrices in the SuiteSparse matrix collection (Davis and Hu 2011). Table 2 summarizes the completed feature parameters of sparse structures used in the SMAT auto-tuner. We extend SMAT to support block formats and include two new features of dense subblocks.

Since CSR is the default SMAT format, we design more advanced parameters to describe SpMV behavior of the other formats by analyzing the relationships between parameter values and SpMV performance on matrices from the SuiteSparse matrix collection (Davis and Hu 2011) collection. Hence, we divide the 13 parameters into six categories as follows:

—**Matrix dimension** includes the basic matrix dimension parameters, M and N, which can be applied to all formats (marked as $\surd$ in Table 2).

Table 2. Feature Parameters of a Sparse Matrix and the Relationship with the Formats

Parameter		Meaning	Formula	DIA	(B/S)ELL	(B)CSR	(B)COO
Matrix Dimension	M	#Rows	-	√	√	√	√
	N	#Columns	-	√	√	√	√
Diagonal Situationc	Ndiags	#Diagonals	-	↓			
	NTdiags_ratio	The ratio of "true" to all diagonals	$\frac{\#"true\ diagonals"}{Ndiags}$	↑			
Nonzero Distribution	NNZ	#Nonzeros	-	√	√	√	√
	aver_RD	#Nonzeros per row	$\frac{NNZ}{M}$	√	√	√	√
	max_RD	The maximum #Nonzeros per row	$\max_1^M\{\#nnzs\ per\ row\}$		↓		
	var_RD	The variance of #Nonzeros per row	$\frac{\Sigma_1^M\lvert row\_degree - aver\_RD\rvert^2}{M}$			↓	
Nonzero Ratio	ER_DIA	The ratio of nonzeros in DIA	$\frac{NNZ}{Ndiags \times M}$	↑			
	ER_ELL	The ratio of nonzeros in ELL	$\frac{NNZ}{max\_RD \times M}$		↑		
Power-Law	R	A factor of power-law distribution	$P(k) \sim k^{-R}$				[1, 4]
Block Size	est_rs	The estimated dense subblock's row size	-	↑	↑	↑	↑
	est_cs	The estimated dense subblock's column size	-	↑	↑	↑	↑

—**Diagonal situation** includes the number of diagonals (*Ndiags*) and the ratio of "true" diagonals (*NTdiags_ratio*).
—**Nonzero distribution** includes the total number of nonzero elements (*NNZ*) and the average, maximum, and variance of the number of nonzero elements per row (*aver_RD*, *max_RD*, and *var_RD*, respectively).
—**Nonzero ratio** includes the ratio of nonzero elements in DIA or ELL formats (*ER_DIA* and *ER_ELL*, respectively).
—**Power law** is the power law distribution factor, *R*, which reflects the SpMV behavior of COO format.
—**Block size** includes the estimated block size of dense subblocks (*est_rs* and *est_cs*). Upper or lower arrows in Table 2 indicate the SpMV performance trend as a parameter value increases or decreases, respectively. Since block size affects all blocked or sliced formats, we mark all formats with arrows. We also support Sliced ELL (SELL (Monakov et al. 2010)) format, which shares the same features as ELL format.

We also introduce details regarding the additional block features, `est_rs` and `est_cs`, that represent the row and column size of dense subblocks, respectively. A sparse matrix can be stored in block format with many different dense subblock sizes. Most sparse matrices can achieve close to or the highest performance with block size ≤8 (Vuduc et al. 2005). Thus, we only consider block sizes from 1×1 to 8×8. However, since it is not trivial to determine the best block size, we compute the fill ratio for each of the 64 possible block size combinations using a sampling method (Vuduc et al. 2005) to compute the fill ratio of a small portion (1%) of the sparse matrix for each combination.

Block sizes for the whole matrix (`est_rs` and `est_cs`) are then estimated using the fill ratio of this small portion (see Section 4.2 for details).

4.2 Model Training

4.2.1 Model Generation. Since optimal storage format selection is formulated as a classification problem in supervised learning, we classify a particular sparse matrix to one of the possible candidates (DIA, ELL, CSR, COO, SELL (Monakov et al. 2010), BELL (Choi et al. 2010), BCSR, or BCOO):

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

where $\vec{x_i}(i = 1, \ldots, n)$ is the set of parameter values for a sparse matrix in the training set; $\vec{TH}$ is the set of threshold values for each attribute; and $C_n(DIA, ELL, CSR, COO, SELL, BELL, BCSR, BCOO)$ represents the eight categories. Our goal is to generate $\vec{TH}$ according to the mapping relationship in the training dataset.

With respect to machine learning, we introduce the SMAT auto-tuner with three key components: feature selection, training model, and prediction. We also define an attribute collection {`M, N, Ndiags, NTdiags_ratio, NNZ, max_RD, var_RD, ER_DIA, ER_ELL, R, est_rs, est_cs, Best_Format`}, where `Best_Format` is the target attribute. Training on 2,055 matrices from the SuiteSparse matrix collection (Davis and Hu 2011), we obtain accurate parameter values of the training matrices. For example, matrix t2d_q9 had a record of {9,801, 9,801, 9, 1.0, 87,025, 9, 0.35, 0.99, 0.99, inf, 2, 2, DIA}, where inf means that the power-law distribution cannot calculate an R value for this matrix since it has no scale-free network attribute. The set of parameter values constitutes the matrix feature database.

Using the matrix feature database, SMAT generates a learning model (combining data mining and analytical methods (Vuduc et al. 2005)) in a ruleset pattern with confidence data. The hybrid model is discussed in detail in Section 4.3. Hence, the auto-tuning system extracts informative patterns from the input data and outputs the ruleset pattern as the trained model. Data mining is used to generate a trained model focused on the basic formats, whereas the analytical method is used to determine the optimal block size for the chosen format.

Data mining leverages the RuleQuest Research (2012) tool to generate a ruleset pattern. Since inaccuracy may exist for a rule, we guide (RuleQuest Research 2012) to generate rules with confidence data, adding a confidence value between 0 and 1 to each rule, i.e., the ratio of the number of correctly classified matrices to the number of matrices following this rule. A larger confidence value implies a more reliable rule, and the confidence data role is shown in detail in Section 4.3.

We use the analytical method to decide estimated block sizes, `est_rs` and `est_cs`. Assuming $perf\_dense(r, c)$ is the performance for a dense matrix stored in a blocked format, $fr(\theta)$ is the fill ratio of a portion of a sparse matrix (θ), where $\theta = 1$ is the fill ratio of the whole sparse matrix. We find that when $\theta = 1\%$, the fill ratio is accurate for 80% of the 2,386 matrices, consistent with previous studies (Vuduc et al. 2005). Thus, we choose $\theta = 1\%$ in the SMAT system, and hence the estimated performance for a sparse matrix with block size (r, c) can be expressed as

$$perf(r, c) = perf\_dense(r, c)/fr(\theta). \quad (2)$$

We choose the largest $perf(r, c)$ among the 64 block size combinations, and set `est_rs, est_cs` to be the corresponding block size. We precompute $perf\_dense(r, c)$ for each (r, c) and store them in a table, and then when a matrix is input, we estimate its fill ratio and search for the corresponding $perf\_dense(r, c)$ value in the table. To increase prediction accuracy, we test the performance of different dense matrix sizes that can be fitted into caches and memory and use the corresponding performance numbers according to the input matrix size.

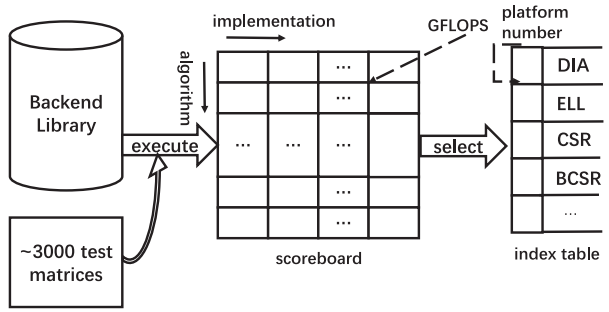


Fig. 4. Kernel search with a scoreboard algorithm.

The data mining method will not be affected by new blocking parameters added from the analytical method, since the blocking parameters only have influence when a basic format is decided. For example, if the ELL format is chosen by the data mining method, we can decide which block size will gain the best performance in BELL (Choi et al. 2010) format, but not in BCSR and BCOO formats.

4.2.2 Kernel Search. It is important to narrow the optimized implementations of the library to search the optimal kernel on a given platform. SMAT can leverage both existing auto-tuning tools and the retargetable back-end library to help search in the offline stage.

The kernel searching process is conducted using a performance record table and scoreboard algorithm, as shown in Figure 4. We run all possible implementations (up to 24, excluding 189 blocked format implementations, because we have to search for the best block size in runtime due to different input matrices) and recorded the corresponding performance. Optimization methods are arranged in a particular order in each record, and the performance number in this record is indexed by all of its optimization strategies. A scoreboard is established from the performance record table to identify the most efficient optimization according to SpMV kernel behavior on this architecture. If the single optimization implementation shows a performance gain compared with the native implementation, the corresponding optimization method is marked with 1 on the scoreboard; otherwise, it is marked by −1. We consider that the optimization has no effect on this architecture when the performance gap between two implementations is <0.01 and ignore this strategy by default. Hence, when more than one optimization strategy is considered, the implementation compares each performance with that for the implementation that has just one less optimization strategy. Thus, we finally obtain the score of each optimization, and the implementation with the highest score is considered to be the best for the corresponding format on the considered architecture. Optimal kernels are called when the data mining method is used to build the trained model and execute/measure module runs.

4.3 Prediction

This section discusses the runtime system components of SMAT (red arrows in Figure 3). This procedure depends on the hybrid model, combining black-box machine learning and analytical models, as shown in Figure 5. When a sparse matrix is input, its features are first extracted using the parameter set and recorded. The parameter values are then input to the trained model and output the predicted format, where the included confidence level determines the prediction reliability. If the confidence is larger than the experimentally derived threshold, the SpMV format with the best implementation is output. Otherwise, an additional execution and measure module

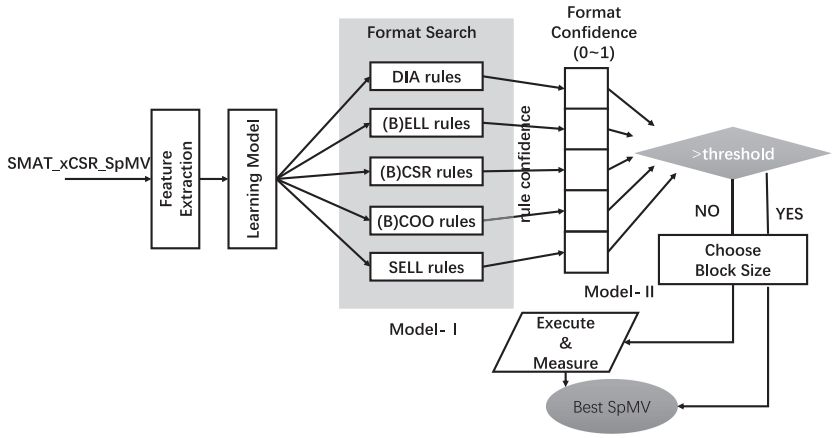


Fig. 5. Runtime procedure.

is triggered to run candidate SpMV kernels with part of the four formats to decide the best SpMV kernel by comparison.

Runtime overhead is critical to online prediction applicability. First, we reduce overhead by considering rule tailoring and grouping. We order the rules according to their estimated contribution to the overall training set: rules that reduce the error rate the most appear first, and rules that contribute the least appear last. We then tailor the rules from the top down until the subset of rules achieves similar predictive accuracy with the whole ruleset, where a 1% accuracy gap is considered acceptable. We choose rules 1 to 15 for Intel platforms, decreasing the feature database error to 9.6%, which is competitive with the 9.0% error achieved by the whole ruleset (40 rules).

Second, we employ an incomplete procedure strategy to accelerate candidate selection. We do not need to go through all the rules if prediction confidence is high after some subset has been processed. This will provide a quicker prediction, saving runtime overhead. For example, suppose that when feature parameter values from the first feature extraction process section are calculated, the learning model predicts the DIA rule group and obtains DIA confidence. If the DIA confidence is larger than the threshold, then DIA format will be the best SpMV format with its best implementation as the output. First, the format group order should be set to realize the incomplete procedure. Since DIA achieves the highest performance when the matrix satisfies its conditions, we arrange the DIA rule group first to pursue high performance. Since ELL format has a regular behavior that is relatively easy to predict, the ELL rule group is placed second after DIA. The CSR rule group parameters are calculated in the first feature extraction process section along with DIA and ELL, and hence CSR is placed third. Thus, COO is the final format to predict in the format search module. Following this order, the prediction runtime time could be significantly reduced while retaining accurate prediction.

To determine the best block size, we first consider the chosen format from Model I using data mining, then extract blocking parameter values in the feature extraction process until after the basic format is decided, thus shortening the runtime prediction process as much as possible. If the format confidence is large enough, we evaluate its blocked format to decide the best block size. Otherwise, if there are several candidate formats, i.e., all candidate confidence values are less than the threshold, we evaluate blocked formats of each basic format to estimate the best block size. Then, we execute SpMV in the blocked formats with the best block size, or the basic formats once and choose the best format.

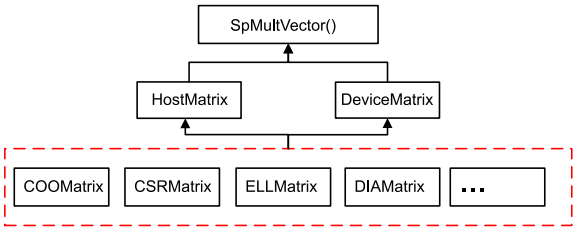


Fig. 6. The algorithm hierarchy.

5 BACK END

The spBLAS library offers specific SpMV implementations on multiple hardware platforms, including CPU, NVIDIA GPU, and Xeon Phi. Considering the difficulty of applying different programming models on diverse platforms, the CPU is abstracted as a host while NVIDIA GPU and Xeon Phi are abstracted as devices in Figure 6. We use a general Matrix class to encapsulate all sparse matrix formats and their SpMV implementations. HostMatrix and DeviceMatrix are inherited from the Matrix class to separately represent host and device format. Thus, users do not need to specifically consider device types, which dramatically reduces programmer workload. HostMatrix and DeviceMatrix classes derive specific sparse matrix storage format classes, such as COOMatrix, CSRMatrix, ELLMatrix, and DIAMatrix. We implement these formats on both the host and devices for various SpMV algorithms. The spBLAS library defines the algorithm interfaces in a high-level abstract class and implements the specific algorithm on the corresponding hardware platform. The `SpMultVector()` function is a member function of the sparse matrix class that computes SpMV. The function has two explicit parameters: `in` is the input dense vector, and `out` is the output vector.

While the spBLAS back-end library is able to automatically detect the hardware platform during compiling or runtime, we use the open-source, cross-platform family of "CMake" tools to probe the hardware platforms. Hence, the library automatically perceives hardware accelerators and programming environments, and the host code can be compiled regardless of whether hardware accelerators are present. When the system includes GPU accelerators and a CUDA programming environment, spBLAS compiles the code and stops probing other accelerators. Otherwise, the system probes for an OpenCL environment, or others.

To fully activate the vector computing units on Xeon Phi, the spBLAS back-end library adopts DIA, CSR, BCSR, COO, HYB (Bell and Garland 2008), BCOO, ELL, BELL, and SELL (Monakov et al. 2010) storage formats for the auto-tuner and numerical solvers. Considering the disadvantage of filling zeros in the SELL (Monakov et al. 2010) storage format, we first split sparse matrices into submatrices and rank them. We continue a second split with 16 or eight elements per row for SP and DP, respectively, so that every data segment fills the 512-bit SIMD FMA unit. The Xeon Phi vector processing capacity is fully used by memory alignment. We also provide blocking formats (BCSR, BCOO, and BELL) for better SpMV memory locality. Block size is chosen from 1×1 to 8×8, and we implement different loop unrolling strategies for different block sizes to increase performance. We also exploit instruction-level optimization, including using the ***gather*** instruction to collect the X vector and improve AVX 512 vector registers' efficiency and leverage to implement SpMV computation.

To ensure efficient usage of computing resources, multithreading is employed to parallelize SpMV implementations. We split sparse matrices into even submatrices for each thread while maintaining an even number of nonzero elements for load balance. Other general optimizations including data alignment and loop unrolling are also adopted.

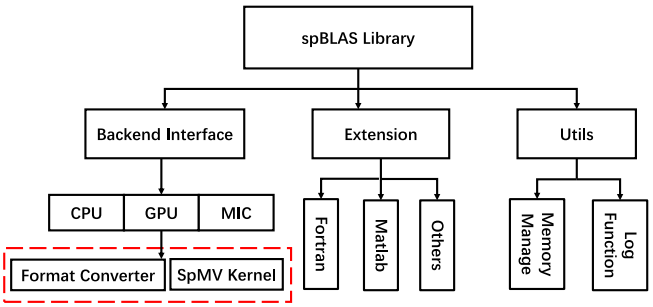


Fig. 7. The architecture of spBLAS.

Figure 7 shows that the spBLAS library framework includes back-end, extension, and utils modules. The back-end module supports primary SpMV algorithms, with sparse matrix files stored in MatrixMarket (MM) format. The spBLAS library recognizes MM format and stores the sparse matrix in a CSR format data structure in its memory. Since different storage formats correspond to different SpMV algorithms with various performances in different platforms, the spBLAS library offers a variety of storage formats and SpMV implementations on diverse platforms, including CPU, NVIDIA GPU, and Xeon Phi. A format transformation interface is provided to convert a sparse matrix from one format to another.

The spBLAS library uses C++ language to implement SpMV and extend third-party languages and libraries, such as Matlab, cuSPARSE library, and Intel MKL library. Users can also integrate the spBLAS library into their own algorithms or libraries. Thus, spBLAS provides a unified interface for different platforms, dramatically decreasing programmer workload. It also has an extension module for users who have special needs for their applications, further extending spBLAS library flexibility.

6 EXPERIMENTS AND ANALYSIS

This section reports experiments to evaluate SMATER performance for benchmark matrices and a real application, including accuracy and overhead analysis.

6.1 Experimental Setup

Platform. Experiments were conducted on different architectures: Intel Xeon E5-2680 v3 and AMD Opteron 6168 multicore processors, and NVIDIA GPU K20 and Intel Xeon Phi 3120A many-core coprocessors, as detailed in Table 3. We used the icc compiler for Intel Xeon E5-2680 v3 and Intel Xeon Phi 3120A, gcc for AMD Opteron 6168, and nvcc for NVIDIA GPU K20. Many-core coprocessors have significantly higher peak performance than multicore processors.

Benchmark. The training dataset for the offline data mining component included 2,055 sparse matrices randomly chosen from the SuiteSparse matrix collection (Davis and Hu 2011). We selected another 331 matrices for performance evaluation. Figure 8 shows 16 representative matrix data graphs, chosen according to their nonzero distribution pattern: matrices 1 to 4 are largely a diagonal pattern, 5 to 8 have similar nonzeros per row, 9 to 12 have the highest performance for the CSR format, and 13 to 16 have power-law nonzero distribution.

Application. To demonstrate SMATER practicality, we integrated it with the Hypre (Falgout and Yang 2002) package, a scalable linear solver library from the Lawrence Livermore National Laboratory (LLNL), and employed the algebraic multigrid (AMG) algorithm as a preconditioner

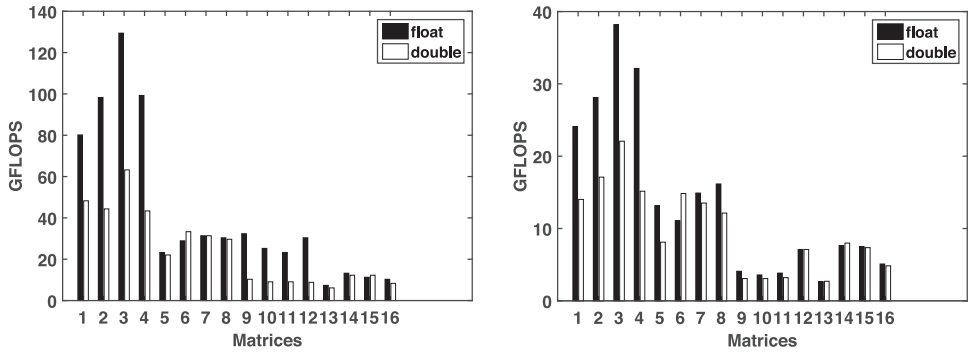
Table 3. Experimental Platform Configuration

Platform	Frequency	#Cores	LLC Size	Memory Size	Bandwidth	GFLOPS
Intel Xeon E5-2680 v3	2.5GHz	24*	30MB	64GB	68GB/s	960 (SP), 480 (DP)
AMD Opteron 6158	1.9GHz	12*	12MB	16GB	42GB/s	91 (SP), 46 (DP)
NVIDIA Tesla K20	900MHz	2496	1.5MB	5GB	208GB/s	3,520 (SP), 1,170 (DP)
Intel Xeon Phi 3120A	1.1GHz	57*	28MB	6GB	240GB/s	1,003 (SP), 502 (DP)

*: Number of physical cores, without considering hyperthreading technology.

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

Fig. 8. Representative matrices.



(a) Intel (b) AMD

Fig. 9. SMATER performance in single and double precision on two CPU platforms.

of conjugate gradients to solve large-scale scientific simulation problems on unstructured grids, which also solves $Au = f$ directly.

6.2 Performance

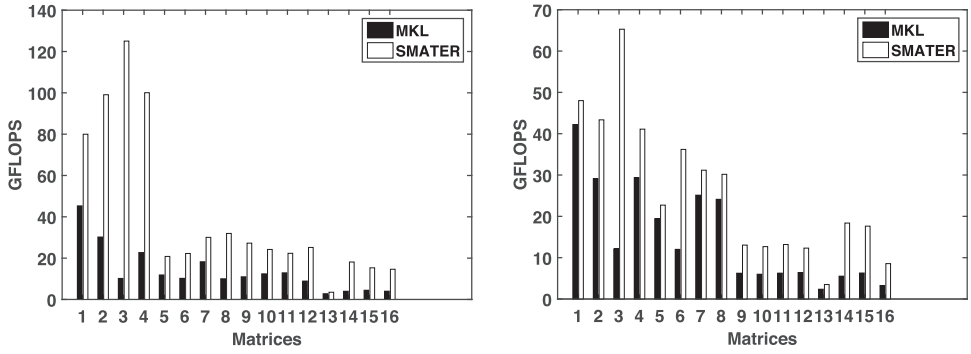
We measure SpMV performance in GFLOPS, the ratio of the number of floating-point operations to execution time.

6.2.1 Multicore CPUs. Figure 9 shows single (SP) and double (DP) precision SpMV performance for the two multicore processors using 16 threads. The highest performance on the Intel platform is 127GFLOPS (13% efficiency) and 64GFLOPS (13%) for SP and DP, respectively (Figure 9(a)). The comparable AMD platform performances are 38GFLOPS (42%) and 22GFLOPS (48%), respectively (Figure 9(b)).

The highest performance is achieved for matrices 1 to 4, because they have banded nonzero distributions. The performance gap between SP and DP is larger for matrices with more regular distributions (e.g., banded nonzero distributions). Overall performance variation is approximately 10 times, strongly supporting adopting a SMATER auto-tuner due to its adaptivity for diverse sparse matrices. The 16 typical matrices achieve this performance using various formats (DIA, ELL, BCSR, COO, BELL (Choi et al. 2010), or SELL (Monakov et al. 2010)), indicating that it is meaningful to implement a high-performance SpMV library aware of sparse structures (applications).

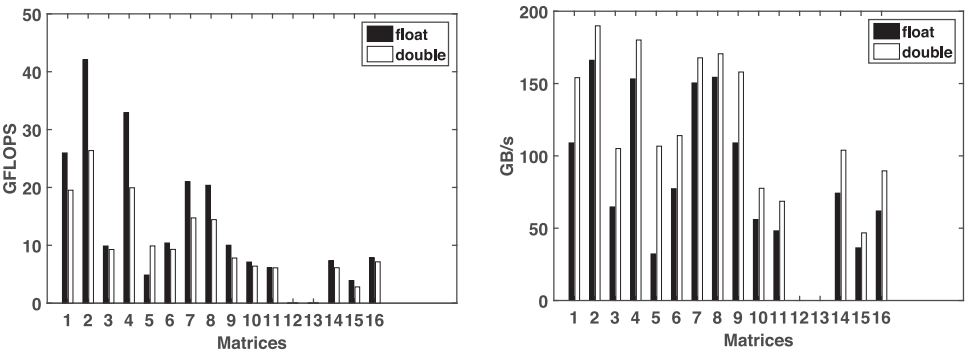
Figure 10 compares SMATER and Intel MKL multithreaded library performance for SP and DP on the Intel platform. SMATER obtains maximum speedup of approximately 10 and 4.9 times for SP and DP, respectively. Although the figure only shows 16 representative matrices, we collect experimental data for all 331 matrices, and average speedup compared to MKL was >3 times for both SP and DP. Thus, SMATER has significant performance advantages due to optimized SpMV implementations with SIMDization and load-balancing policies. Choosing the best format and implementation enables the highest SpMV performances.

6.2.2 Many-Core Accelerators. Figure 11 shows SMATER performance and bandwidth on NVIDIA GPU, where bandwidth is derived from the performance. For example, we use $2 * nnz/time$ to compute the CSR format GFLOPS performance, where 2*nnz is the number of multiplication and addition operations, and $((m + 1 + nnz) * sizeof(int) + (2 * nnz + m) * sizeof(double))/time$ to compute GB/s bandwidth, where nnz is the number of nonzeros and m is the number of rows of the sparse matrix. Bandwidth differs significantly due to matrix irregular



(a) float (b) double

Fig. 10. The performance of SMATER versus MKL.



(a) The performance of SMATER on GPU (b) The bandwidth of SMATER on GPU

Fig. 11. The performance and bandwidth of SMATER on GPU.

memory access (Figure 11(b)), with most test matrices' bandwidth being slightly smaller than the theoretical NVIDIA GPU hardware bandwidth (208GB/s).

Figure 12 shows SMATER performance on Intel Xeon Phi. Since GPU and Xeon Phi memory sizes are both smaller than the CPU memory size, we can only provide performance outcomes for a subset of the sparse matrices.

The highest GPU performance is 43 and 26GFLOPS for SP and DP, respectively; both are achieved for matrix 2 in DIA format, whereas other matrices only achieve 3 to 7GFLOPS, significantly less than its peak performance (Table 3). Thus, the GPU does not show any advantage compared to multicore platform performance. Although SP performance is usually higher than DP, some matrices have similar performance in both precisions, and matrix 5 achieves higher performance in DP. This phenomenon is also present for the multicore platforms (Figure 9). Sparse matrix performance behavior is not as uniform as dense sparse operations due to the diverse irregular memory access patterns generated.

SMATER achieves up to 20GFLOPS (SP) and 16GFLOPS (DP) performance (matrix 1) on Intel Xeon Phi (Figure 12). Therefore, we test SMATER performance using the MKL library for Xeon Phi with 60, 120, 180, and 240 threads, achieving the highest performance for 120 threads. We then analyze SMATER SP and DP performance using 120 threads.

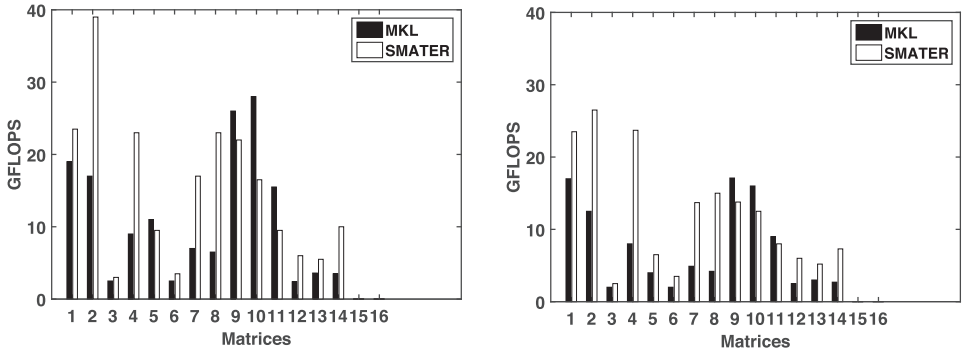


Fig. 12. The performance of SMATER on Intel Xeon Phi coprocessors.

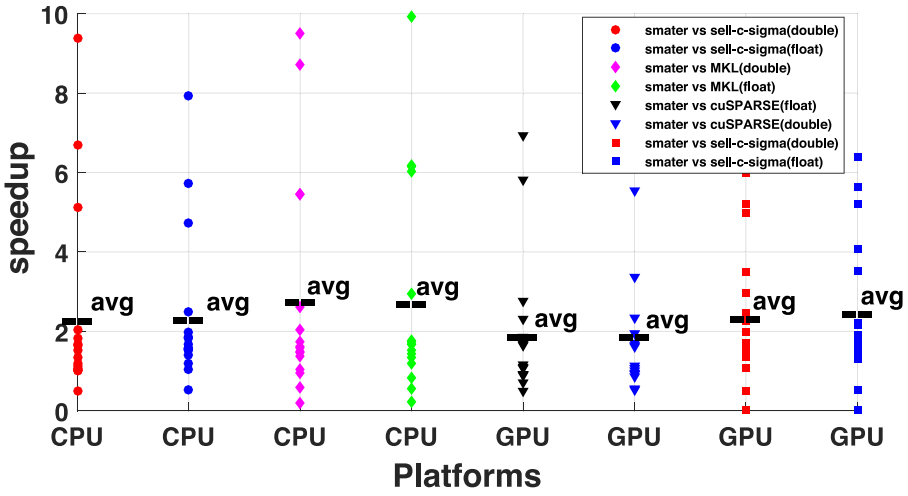


Fig. 13. Relative performance benefit of SMATER over the other libraries.

The best storage format varies between the different platforms. For example, for matrix *bfly*, the best storage format is CSR and ELL for CPU and GPU platforms, respectively. Block size for optimal performance also differs. For example, for matrix *pcrystk02*, optimal format is BCSR with 3×3 block size, whereas for matrix *whitaker3_dual*, optimal format is BELL with 1×2 block size.

6.2.3 Relative Performance. Figure 13 shows the relative performance benefit from SMATER over the SELL-C-σ (Kreutzer et al. 2014; Anzt et al. 2014) format and vendor-supplied SpMV library using the NVIDIA K20 and Intel Xeon E5-2680 v3 (24-core) platforms. Table 3 shows the detailed platform configurations. The data points in Figure 13 represent the ratio of SMATER performance to that for the other libraries. SMATER performance is generally superior, aside from a few cases. The two zero points are due to the matrices being so large that SELL-C-σ (Kreutzer et al. 2014; Anzt et al. 2014) format is unable to deal with them, although other libraries can. SMATER obtains average speedup of approximately 2 times (up to 10 times), compared to performance for other libraries for DP and SP test matrices.

Table 4. Prediction Format of SMAT on GPU

Matrix Number	Matrix Name	SMAT Prediction Format	Model Accuracy
1	pcrystk02	DIA	R
2	denormal	DIA	R
3	cryg10000	ELL	W
4	apache1	DIA	R
5	bfly	ELL	R
6	whitaker3_dual	ELL	R
7	ch7-9-b3	ELL	R
8	shar_te2-b2	ELL	R
9	pkustk14	ELL	R
10	crankseg_2	COO	R
11	Ga3As3H12	COO	R
14	D6-6	CSR	R
15	dictionary28	CSR	R
16	roadNet-CA	CSR	R

"R" and "W" represent right and wrong prediction, respectively.

6.2.4 CPU Scalability. The number of threads for the best sparse matrix performance differs for the CPU-based platforms. Generally, performance improves for an increasing number of threads for matrices *pcrystk02*, *denormal*, *apachel*, *ch7-9-b3*, *shar_te2-b2*, *D6-6*, and *dictionary28*. However, when the number of threads exceed hardware hyperthreading (48 for our CPU platform), performance improvements cease. There are exceptions for some matrices (*pkustk14*, *rankseg_2*, *Ga3As3H12*, *HV15R*, and *europe_osm*) that continue to increase performance when the number of threads is bigger than 48. On the other hand, performances for matrices *cryg1000* and *whitaker3_dual* are relatively constant for different numbers of threads, and performances of *bfly* and *roadNet-CA* fluctuate when the number of threads equals 12 and 48, respectively.

6.3 Accuracy and Overhead

As an auto-tuning system, SMAT performance and usability are both significant. Two important aspects to consider are whether SMAT can find the most accurate format (accuracy) and what is the price needed to pay for good performance (prediction overhead).

For all 331 test matrices, SMAT accuracy equals 92% and 82% on the Intel platform, and 85% and 82% on the AMD platform for SP and DP, respectively. Similarly, SMAT accuracy on GPU equals 89% (SP) and 95% (DP) for 289 testing matrices, respectively. Thus, even if SMAT does not choose the optimal format, it still achieves fairly good performance, sufficient to be used in practical applications. Table 4 shows GPU accuracy details. The only incorrect prediction is matrix 3, which should have used DIA rather than the predicted ELL format. However, correct predictions are made for all other matrices. SMATER optimal formats for matrices 9 to 16 differ from our previous CPU predictions (Li et al. 2013), demonstrating that algorithm behavior on different architectures is not the same. Thus, it is reasonable to inspect algorithm behavior when transforming to a machine with different architecture.

In all tests, if the trained model generates a confident prediction, the overhead is <5 times SpMV runtime using CSR format. When the model fails to decide an exact format, SMAT needs to choose from several candidates, which increases runtime to ≈15 times CSR-SpMV. Compared with other

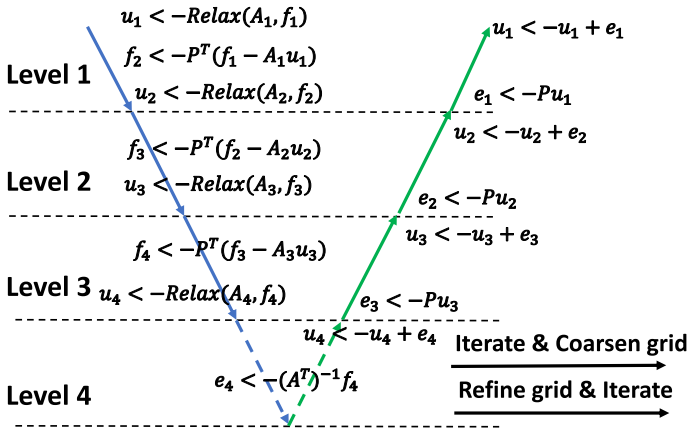


Fig. 14. V-cycle involved with SpMV kernels in AMG solver.

auto-tuner overheads (OSKI (40 times) and clSpMV (1–20 times)), the SMAT overhead is considered acceptable.

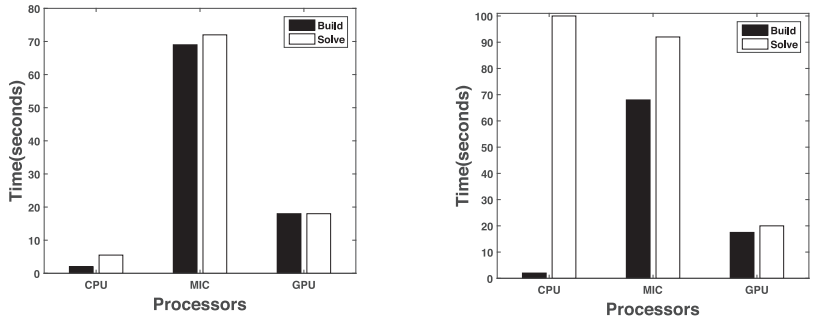
6.4 Application in Algebraic Multigrid

Consider a problem of the form $Au = f$, where A is a large sparse matrix, and u and f are dense vectors. The goal is to solve this equation and obtain u with sufficient precision. Hypre AMG solves this problem by building N levels of virtual grids with a set of grid operators $(A_0, \ldots, A_{N-1})$ and grid transfer operators $(P_0, \ldots, P_{N-2})$ in a setup process where parallel implementation of general sparse matrix-matrix multiplication is essential (Liu and Vinter 2015b; Liu et al. 2018; Hou et al. 2017). Figure 14 illustrates a typical V-cycle in AMG. The P operators perform SpMV between adjacent grids, while A operators perform relaxations, using Jacobi and Gauss-Seidel methods, that also have SpMV kernels. Thus, SpMV consumes the most V-cycle execution time. However, the two sets of sparse matrices show dynamically different matrix features from the original A. Thus, implementing SMATER would be useful to choose the best format and implementation for operators on each level, improving overall performance.

We perform experiments on *cljp* and *rugeL* coarsening methods in the setup process to generate A operators with different structures. Input matrices are generated by 7- and 9-point Laplacian methods. Rather than employing CSR format all the time, SMATER chooses DIA format for A operators in the first few levels, and ELL format for most P operators. We simply replace the SpMV kernel codes with SMAT interfaces without changing the original CSR data structure in the Hypre package. The AMG solution achieves >20% performance gain compared with Hypre AMG on the multicore CPU platform.

Figure 15 compares the AMG iteration execution time on CPU, MIC, and GPU platforms. Each black bar represents the time required to build operators in each level, and white bars indicate the iterated solver runtime. Total time consumed on CPU is the least when the number of iterations equals 10, but the time on MIC is the greatest. As iterations increase, the total time consumed on CPU becomes the greatest, which can be attributed to the following:

—The phase of building operators is not parallelized on the accelerators.
—The building phase on GPU or MIC has extra overhead of transferring matrices from the host to acceleration device, which degrades the total performance.



(a) 10 iterations (b) 100 iterations

Fig. 15. The breakdown of AMG execution time.

However, as the number of iterations increases, the ratio of the solver involving SpMV also increases, demonstrating the advantage of accelerators to improve overall performance.

7 RELATED WORK

Several recent studies have investigated optimizing sparse matrix vector multiplication, and Filippone et al. (2017) provided a review of sparse matrix formats available on NVIDIA GPUs. Although we are unable to provide a comprehensive set, we contrast our outcomes for several key representatives.

Storage format. The proposed SMATER system determines the optimal storage format for sparse matrices, building on our previous work. There are currently more than 15 sparse matrix storage formats (Bell and Garland 2008; Kourtis et al. 2011; Vuduc et al. 2005; Saad 1994; Su and Keutzer 2012; Stevenson et al. 2012; Yan et al. 2014; Liu and Vinter 2015a; Maggioni and Berger-Wolf 2016; Nagasaka et al. 2016; Liu and Vinter 2015c; Xie et al. 2018), most of which are either application or architecture specific, and hence have limited applicable domains.

Several hybrid storage formats have been recently developed. Su and Keutzer (2012) proposed a Cocktail format to split sparse matrices and represent the submatrices in different formats according to relative sparse matrix format advantages. Similarly, cuSPARSE (NVIDIA 2010) on NVIDIA GPUs stores a sparse matrix with hybrid format, HYB, a combination of more than two basic formats. These hybrid storage formats show better performance than conventional formats for some sparse matrices. Their major difference from SMATER is that the hybrid formats are determined statically and are not applicable to dynamic sparse structures, such as required in different AMG solver levels.

Several studies have also proposed relatively flexible formats. Vuduc et al. (2005) improved the blocked compressed sparse row (BCSR) format for VBR to explore dense blocks with different sizes. Kourtis et al. (2011) proposed CSX to exploit dense structures not limited to dense blocks, but also one-dimension bars and dense diagonals by compressing metadata. However, the process to search for subblocks costs too much to be conducted online. Yan et al. (2014) proposed an extended SpMV format, called blocked compressed common coordinate (BCCOO), that used bit flags to store row indices in a blocked common coordinate (COO) format, alleviating bandwidth problems. In contrast, SMATER selects the optimal format from the existing formats, rather than designing a new format. The previously trained machine-learning model makes online decisions feasible, enabling new formats in SMATER simply by plugging them into the feature database and kernel library

components. Other sparse operations, including sparse triangular solve, also require searching for storage formats (Liu et al. 2017; Wang et al. 2018).

Auto-tuning approach. Auto-tuning is a promising tool to develop domain-specific performance-critical libraries, solving performance and portability problems. Several successful auto-tuning libraries have been developed in the scientific computing field, including PAK (Liu et al. 2018a), ATLAS (Whaley and Dongarra 1998), FFTW (Frigo and Johnson 2005), SPIRAL (Püschel et al. 2005), and OSKI (Vuduc et al. 2005). Auto-tuning techniques have been particularly investigated for SpMV applications. jin Im et al. (2004) created BCSR format to better develop dense block performance in sparse matrices. Vuduc et al. (2005) built an auto-tuner (OSKI) to tune matrix block sizes in BCSR or VBR formats. Williams et al. (2009) combined auto-tuning with a hierarchy strategy to choose the best parameter combinations. Choi et al. (2010) implemented BCSR and sliced blocked ELLPACK (SBELL) formats on NVIDIA GPUs to tune block sizes. Yang et al. (2011) proposed a mixed format and automatically chose the partition size of each format using an auto-tuning model. Yan et al. (2014) built an auto-tuning framework to tune parameter values among different strategies and proposed two formats (BCCOO and BCCOO+). Zhao et al. (2018) used deep learning for auto-tuning, which is a development of SMAT but is not as flexible as SMATER. Sedaghati et al. (2015) also used a decision tree approach to build an auto-tuner for GPUs, but they only used approximately 700 matrices. A common feature of previous auto-tuning approaches is the focus on tuning implementations on diverse processing architectures for a given storage format or its variant beforehand. In contrast, in addition to architectural auto-tuning, SMATER extends to cooperatively tune storage formats by extracting key performance parameters from input sparse matrices. Algorithm-architecture cotuning was previously advocated in the PetaBricks compiler (Ansel et al. 2009), and our results further support this as an important auto-tuning trend.

Prediction model. A core SMATER component is the machine-learning-based prediction model employed to search optimal format and implementation. It is a common strategy to apply prediction models for auto-tuning. ATLAS (Whaley and Dongarra 1998) performed an empirical search to determine the optimal parameter values bounded by the architecture features. However, an actual run of the generated code was required to measure and record performance and hence choose the best implementation. This empirical search was efficient to generate high-quality BLAS codes, even though the search process required significant runtime. Many recently proposed auto-tuners have adopted the model-driven approach (Vuduc et al. 2005; Su and Keutzer 2012; Kourtis et al. 2011; Yang et al. 2011; Choi et al. 2010; Neelima et al. 2014; Li et al. 2015, 2017) without requiring code runs. Although the model-driven method decreases prediction time, code performance is generally significantly lower than an empirical search (Yotov et al. 2005). In contrast, the SMATER system combines the learning model and rarely called empirical search to simultaneously ensure code performance and reduce prediction time.

Although clSpMV (Su and Keutzer 2012) also uses a prediction model to tune its Cocktail format, there are some crucial differences from SMATER. The first and most important distinction in online decision making is that clSpMV uses the maximum GFLOPS measured in the offline stage. Unfortunately, our experiments on the SuiteSparse matrix collection (Davis and Hu 2011) (see Table 1) show that the maximum performance of one format is not sufficiently representative to reflect the performance of all matrices suitable for this format. It is more accurate to use the features of each input matrix to predict its own best format rather than a single maximum performance for each format. Second, we extract more features from realistic matrices in the SuiteSparse matrix collection (Davis and Hu 2011) collection, which can feed more training data to data mining tools to generate more reliable rules as the learning model. Armstrong and Rendell (2008) also use reinforcement learning to choose the best format, but users must decide factor values that

influence the learning model accuracy. Ultimately, SMATER offers a more convenient system to automatically generate the model and achieve similar prediction accuracy.

8 CONCLUSION

This article proposes the SMATER system, extending the automatic input adaptive library generation tool SMAT for SpMV. SMATER combines several statistical and machine-learning techniques to build a hybrid model and enable both application- and architecture-dependent black-box prediction. In particular, we provide a unified interface to eliminate tedious work on choosing proper formats and implementation of diverse sparse matrices.

This work indicates that machine-learning techniques are useful to learn irregular algorithm behaviors. As optimization methods and hardware architecture develop, more algorithm performance factors and hardware features must be considered. These factors constitute a large parameter space for building a performance model. This model is difficult to build for irregular algorithms due to the large input space and unclear relationships between them. Machine-learning techniques are useful to determine complicated parameter relationships and devise a model. We can combine the machine-learning-generated model and our algorithm analysis to build a reliable model.

Due to cooperative auto-tuning between algorithms and architectures, SMATER achieves impressive performance in our work. The generated SpMV obtains 127 (38) GFLOPS in single precision and 64 (22) GFLOPS in double precision on an Intel (AMD) multicore processor. The averaged speedup is greater than 3 times that of the Intel MKL sparse library. For many-core architectures, SMATER also achieves 43 (20) GFLOPS in single precision and 26 (16) GFLOPS in double precision on NVIDIA GPU (Intel Xeon Phi). The advantage of SMATER is also demonstrated in a real application in which it improves the Hypre sparse linear system solver by about 20%.

ACKNOWLEDGMENTS

We would like to express our gratitude to all reviewer's constructive comments for helping us polish this article.

REFERENCES

RuleQuest Research. 2012. *Data Mining Tools See5 and C5.0.* Retrieved from http://www.rulequest.com/see5-info.html.

Intel. 2017. *Intel Math Kernel Library.* Retrieved from http://software.intel.com/en-us/intel-mkl.

M. D. Adam Hill. 2005. *The International Thermonuclear Experimental Reactor.* Technical Report.

Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. 2009. PetaBricks: A language and compiler for algorithmic choice. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09).* ACM, 38–49.

Hartwig Anzt, Stanimire Tomov, and Jack Dongarra. 2014. *Implementing a Sparse Matrix Vector Product for the SELL-C/SELL-C-σ Formats on NVIDIA GPUs.* University of Tennessee, Technical Report. ut-eecs-14-727.

Warren Armstrong and Alistair P. Rendell. 2008. Reinforcement learning for automated performance tuning: Initial evaluation for sparse matrix format selection. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing.* 411–420.

S. Beamer, K. Asanovi, and D. Patterson. 2017. Reducing Pagerank communication via propagation blocking. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS'17).* 820–831.

Nathan Bell and Michael Garland. 2008. *Efficient Sparse Matrix-Vector Multiplication on CUDA.* NVIDIA Technical Report NVR-2008-004. NVIDIA Corporation.

Geoffrey Belter, E. R. Jessup, Ian Karlin, and Jeremy G. Siek. 2009. Automating the generation of composed linear algebra kernels. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC'09).* ACM, New York, Article 59, 12 pages.

Sergey Brin and Lawrence Page. 1998. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the 7th International Conference on the World Wide Web 7 (WWW7'98).* Elsevier Science Publishers B. V., 107–117.

Jed Brown, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. 2012. *PETSc Users Manual.* Technical Report ANL-95/11 - Revision 3.3. Argonne National Laboratory.

Aydin Buluc, Samuel Williams, Leonid Oliker, and James Demmel. 2011. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS'11)*. IEEE Computer Society, 721–733.

Daniele Buono, Fabrizio Petrini, Fabio Checconi, Xing Liu, Xinyu Que, Chris Long, and Tai-Ching Tuan. 2016. Optimizing sparse matrix-vector multiplication for large-scale data analytics. In *Proceedings of the 2016 International Conference on Supercomputing (ICS'16)*. ACM, New York, Article 37, 12 pages.

Jee W. Choi, Amik Singh, and Richard W. Vuduc. 2010. Model-driven autotuning of sparse matrix-vector multiply on GPUs. *SIGPLAN Notices* 45, 5 (Jan. 2010), 115–126.

Timothy A. Davis and Yifan Hu. 2011. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.* 38, 1 (2011), 1:1–1:25.

Robert D. Falgout. 2006. An introduction to algebraic multigrid computing. *Computing in Science Engineering* 8, 6 (Nov.-Dec. 2006), 24–33.

Robert D. Falgout and Ulrike Meier Yang. 2002. Hypre: A library of high performance preconditioners. In *Computational Science —ICCS 2002 Part III*, P. M. A. Sloot, C. J. K. Tan, J. J. Dongarra, and A. G. Hoekstra (Eds.). Vol. 2331. Springer-Verlag, Berlin, 632–641.

Salvatore Filippone, Valeria Cardellini, Davide Barbieri, and Alessandro Fanfarillo. 2017. Sparse matrix-vector multiplication on GPGPUs. *ACM Transactions on Mathematical Software* 43, 4, Article 30 (Jan. 2017), 49 pages.

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

Dominik Grewe and Anton Lokhmotov. 2011. Automatically generating and tuning GPU code for sparse matrix-vector multiplication from a high-level representation. In *Proceedings of the 4th Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-4'11)*. ACM, Article 12, 8 pages.

Jong-Ho Byun, Richard Lin, James W. Demmel, and Katherine A. Yelick. 2012. *pOSKI: Parallel Optimized Sparse Kernel Interface Library*. Technical report, University of California, Berkeley.

Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. 2005. An overview of the trilinos project. *ACM Transactions on Mathematical Software* 31, 3 (Sept. 2005), 397–423.

Kaixi Hou, Weifeng Liu, Hao Wang, and Wu-chun Feng. 2017. Fast segmented sort on GPUs. In *Proceedings of the International Conference on Supercomputing (ICS'17)*. ACM, Article 12, 10 pages.

Eun jin Im, Katherine Yelick, and Richard Vuduc. 2004. Sparsity: Optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications* 18 (2004), 2004.

Marat F. Khairoutdinov and David A. Randall. 2001. A cloud resolving model as a cloud parameterization in the NCAR community climate system model: Preliminary results. *Geophysical Research Letters* 28, 18 (2001), 171–178.

Kornilios Kourtis, Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. 2011. 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)*. ACM, 247–256.

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

Jiajia Li, Casey Battaglino, Ioakeim Perros, Jimeng Sun, and Richard Vuduc. 2015. An input-adaptive and in-place approach to dense tensor-times-matrix multiply. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*. ACM, New York, Article 76, 12 pages.

Jiajia Li, Jee Choi, Ioakeim Perros, Jimeng Sun, and Richard Vuduc. 2017. Model-driven sparse CP decomposition for higher-order tensors. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS'17)*. 1048–1057.

Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. 2013. SMAT: An input adaptive auto-tuner for sparse matrix-vector multiplication. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*. ACM, 117–126.

Changxi Liu, Biwei Xie, Xin Liu, Wei Xue, Hailong Yang, and Xu Liu. 2018b. Towards efficient SpMV on sunway many-core architectures. In *Proceedings of the 32nd ACM International Conference on Supercomputing (ICS'18)*.

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

Junhong Liu, Guangming Tan, Yulong Luo, Jiajia Li, Zeyao Mo, and Ninghui Sun. 2018a. An autotuning protocol to rapidly build autotuners. *ACM Transactions on Parallel Computing* (2018).

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

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

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

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

Weifeng Liu and Brian Vinter. 2015c. Speculative segmented sum for sparse matrix-vector multiplication on heterogeneous processors. *Parallel Computing* 49, C (Nov. 2015), 179–193.

Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. 2013. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS'13)*. ACM, New York, 273–282.

Marco Maggioni and Tanya Berger-Wolf. 2016. Optimization techniques for sparse matrix-vector multiplication on GPUs. *Journal of Parallel and Distributed Computing* 93, C (July 2016), 66–86.

Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. 2010. Automatically tuning sparse matrix-vector multiplication for GPU architectures. In *International Conference on High-Performance Embedded Architectures and Compilers*. Springer, 111–125.

Krishna K. Nagar and Jason D. Bakos. 2011. A sparse matrix personality for the convey HC-1. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'11)*. 1–8.

Yusuke Nagasaka, Akira Nukada, and Satoshi Matsuoka. 2016. Adaptive multi-level blocking optimization for sparse matrix vector multiplication on GPU. *Procedia Computer Science* 80 (June 2016), 131–142.

B. Neelima, G. Ram Mohana Reddy, and Prakash S. Raghavendra. 2014. Predicting an optimal sparse matrix format for SpMV computation on GPU. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW'14)*. IEEE, 1427–1436.

NVIDIA. 2010. *CUDA CUSPARSE Library*. NVIDIA.

Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. 2005. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE* 93, 2 (2005), 232–275. Special Issue on "Program Generation, Optimization, and Adaptation."

Youcef Saad. 1994. *SPARSKIT: A Basic Tool Kit for Sparse Matrix Computations*. Technical Report.

Naser Sedaghati, Te Mu, Louis-Noel Pouchet, Srinivasan Parthasarathy, and P. Sadayappan. 2015. Automatic selection of sparse matrix representation on GPUs. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS'15)*. ACM, New York, 99–108.

Nikita Spirin and Jiawei Han. 2012. Survey on web spam detection: Principles and algorithms. *SIGKDD Explorer Newsletter* 13, 2 (May 2012), 50–64.

Avinash Srinivasa and Masha Sosonkina. 2012. Nonuniform memory affinity strategy in multithreaded sparse matrix computations. In *Proceedings of the 2012 Symposium on High Performance Computing (HPC'12)*. Article 9, 8 pages.

John P. Stevenson, Amin Firoozshahian, Alex Solomatnikov, Mark Horowitz, and David Cheriton. 2012. Sparse matrix-vector multiply on the HICAMP architecture. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS'12)*. ACM, New York, 195–204.

Bor-Yiing Su and Kurt Keutzer. 2012. clSpMV: A cross-platform openCL SpMV framework on GPUs. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS'12)*. ACM, 353–364.

Xiangzheng Sun, Yunquan Zhang, Ting Wang, Guoping Long, Xianyi Zhang, and Yan Li. 2011. CRSD: Application specific auto-tuning of SpMV for diagonal sparse matrices. In *Proceedings of the 17th International Conference on Parallel Processing - Part II (Euro-Par'11)*. 316–327.

Richard Vuduc, James W. Demmel, and Katherine A. Yelick. 2005. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc. of SciDAC 2005, J. of Physics: Conference Series*, Vol. 16. Institute of Physics Publishing, 521–530.

Richard W. Vuduc and Hyun-Jin Moon. 2005. Fast sparse matrix-vector multiplication by exploiting variable block structure. In *Proceedings of the 1st International Conference on High Performance Computing and Communications (HPCC'05)*. Springer-Verlag, 807–816.

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

R. Clint Whaley and Jack J. Dongarra. 1998. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (CDROM) (Supercomputing'98)*. IEEE Computer Society, Washington, DC, 1–27.

Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2009. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing* 35, 3 (March 2009), 178–194.

Biwei Xie, Jianfeng Zhan, Zhen Jia, Wanling Gao, Lixin Zhang, and Xu Liu. 2018. CVR: Efficient SpMV vectorization on X86 processors. In *The 2018 International Symposium on Code Generation and Optimization*.

Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. 2014. yaSpMV: Yet another SpMV framework on GPUs. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'14)*. ACM, New York, 107–118.

Xintian Yang, Srinivasan Parthasarathy, and P. Sadayappan. 2011. Fast sparse matrix-vector multiplication on GPUs: Implications for graph mining. *Proceedings of the VLDB Endowment* 4, 4 (Jan. 2011), 231–242.

Kamen Yotov, Xiaoming Li, Gang Ren, Maria Garzaran, David Padua, Keshav Pingali, and Paul Stodghill. 2005. Is search really necessary to generate high-performance BLAS? In *Proc. of the IEEE* 93, 2 (2005), 358–386.

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

Received August 2015; revised December 2017; accepted May 2018