A Performance Modeling and Optimization Analysis Tool for Sparse Matrix-Vector Multiplication on GPUs

Ping Guo, *Student Member, IEEE* , Liqiang Wang, *Member, IEEE* , and Po Chen

Abstract—This paper presents a performance modeling and optimization analysis tool to predict and optimize the performance of sparse matrix-vector multiplication (SpMV) on GPUs. We make the following contributions: 1) We present an integrated analytical and profile-based performance modeling to accurately predict the kernel execution times of CSR, ELL, COO, and HYB SpMV kernels. Our proposed approach is general, and neither limited by GPU programming languages nor restricted to specific GPU architectures. In this paper, we use CUDA-based SpMV kernels and NVIDIA Tesla C2050 for our performance modeling and experiments. According to our experiments, for 77 out of 82 test cases, the performance differences between the predicted and measured execution times are less than 9 percent; for the rest five test cases, the differences are between 9 and 10 percent. For CSR, ELL, COO, and HYB SpMV CUDA kernels, the average differences are 6.3, 4.4, 2.2, and 4.7 percent, respectively. 2) Based on the performance modeling, we design a dynamic-programming based SpMV optimal solution auto-selection algorithm to automatically report an optimal solution (i.e., optimal storage strategy, storage format(s), and execution time) for a target sparse matrix. In our experiments, the average performance improvements of the optimal solutions are 41.1, 49.8, and 37.9 percent, compared to NVIDIA's CSR, COO, and HYB CUDA kernels, respectively.

Index Terms—Performance modeling, sparse matrix-vector multiplication, GPU, CUDA

1 Introduction

Sparse matrix-vector multiplication (SpMV) is an essential operation in solving linear systems and partial differential equations. For many scientific and engineering applications, the matrices can be very large and sparse. It is a challenging issue to accurately predict and optimize SpMV performance. This paper addresses this challenge by presenting a performance modeling and optimization analysis tool to predict and optimize SpMV performance on GPUs.

Bell and Garland [1] proposed and implemented SpMV CUDA kernels for multiple storage formats, including CSR, ELL, COO, and HYB. Based on our experiments, CSR usually has good performance on sparse matrices with large number of non-zero elements; ELL is usually good for a sparse matrix with nearly equal and small number of non-zero elements per row; HYB has better performance when the matrix has small number of non-zero elements per row, and many rows are nearly equal but there may be few irregular rows with many more non-zero elements; COO is the most intuitive storage format, but usually has worse performance than other formats. We observed that different matrices may have their own most appropriate single storage formats to achieve the best performance. Besides, we also notice that there exists a possibility, although not always, that when a matrix is partitioned and each block is stored in an appropriate format, the performance achieved can outperform that of any single storage format. All these observations motivate us to design an automatic tool, to accurately predict the execution times of multiple SpMV kernels, further, to help choose an SpMV optimal solution (i.e., storage strategy, storage format(s), and execution time) for a target sparse matrix.

This paper makes the following contributions:

1. We present an integrated analytical and profile-based performance modeling to accurately predict the kernel execution times of CSR, ELL, COO, and HYB SpMV kernels. Given a target sparse matrix and storage format, its SpMV kernel execution time can be reported.
2. Based on the performance modeling, we design an SpMV optimal solution auto-selection algorithm to automatically report an SpMV optimal solution for a target sparse matrix. We aim to find a storage format (single or multiple formats combined) from current available ones to maximize performance improvement, rather than devise a new single storage format or an SpMV kernel.

Our performance modeling consists of two phases: instrumenting and modeling. In the phase of instrumenting, benchmark matrices are generated according to GPU's

- *P. Guo and L. Wang are with the Department of Computer Science, Department 3315, 1000 E. University Ave., University of Wyoming, Laramie, WY 82071. E-mail: {pguo, lwang7}@uwyo.edu.*
- *P. Chen is with the Department of Geology and Geophysics, Department 3006, 1000 E. University Avenue, University of Wyoming, Laramie, WY 82071. E-mail: pchen@uwyo.edu.*

Manuscript received 13 Aug. 2012; revised 1 Feb. 2013; accepted 10 Apr. 2013.; date of publication 18 Apr. 2013; date of current version 21 Mar. 2014. Recommended for acceptance D. Kaeli.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2013.123

architecture features, then SpMV computations with these benchmark matrices are conducted on the GPU to obtain the execution times. The properties and the execution times of benchmark matrices are recorded as the input in the phase of modeling. In the phase of modeling, we instantiate our parameterized performance models according to the experimental results of benchmark matrices. Finally, we utilize the instantiated models to estimate SpMV kernel execution time for a target sparse matrix.

Our innovative performance modeling approach combines two major techniques: profiling and analytics. Dividing modeling into two phases follows the profile-based technique; and it follows the analytical technique to generate benchmark matrices and performance models according to hardware properties. The integration of both analytical and profile-based modeling has the following advantages: 1) Compared to analytical models, our model is easy to use. 2) Compared to traditional profile-based models, which are usually inaccurate for parallel architectures [2], our model can effectively capture the performance effects of GPUs.

Based on our accurate SpMV performance modeling, to report optimal solutions and optimize performance, we propose a dynamic-programming based algorithm. Given a target sparse matrix, we partition it into strips by row. The algorithm may involve combining some neighboring strips into matrix blocks. According to the predicted execution times for all matrix blocks, the algorithm runs in a bottom-up way and searches all potential storage strategies. If the SpMV performance can be improved when the target matrix is partitioned, the optimal solution, including the storage strategy, the storage format for each block, and the predicted overall execution time, will be reported; otherwise, for the entire matrix, a single storage format with the least predicted execution time will be reported as the optimal solution.

In this paper, we use SpMV CUDA kernels developed by NVIDIA [1] and NVIDIA Tesla C2050 for our performance modeling and experiments. According to our experiments on 22 matrices (totally 82 test cases), our performance modeling is very accurate. Specifically, the average performance differences between the predicted and measured execution times are 6.3, 4.4, 2.2, and 4.7 percent for CSR, ELL, COO, and HYB SpMV CUDA kernels, respectively. The SpMV optimal solutions of the 22 matrices are reported by our tool, where six matrices are suggested to be partitioned to obtain their optimal solutions. The performance improvement of our algorithm is also effective. Specifically, the average performance improvements of the optimal solution suggested by our tool are 41.1, 49.8, and 37.9 percent, compared to NVIDIA's CSR, COO, and HYB CUDA kernels, respectively.

The proposed approach in this paper is general, and neither limited by GPU programming languages nor restricted to specific GPU architectures. Specifically, our modeling approach is applicable to both CUDA-based and OpenCL-based SpMV kernels and can achieve desired accuracy. For different SpMV kernels on the same GPU, no matter CUDA-based or OpenCL-based, only the execution times of benchmark matrices need to be retested. For different GPU architectures, additionally, the benchmark matrices also need to be regenerated. However, these changes only affect the phase of instrumenting. Our parameterized performance models can be reused in the phase of modeling.

2 Related Work

This paper extends our previous work [3] and [4]. The current paper includes: a refined formalization for performance modeling, a new dynamic-programming based algorithm for SpMV optimization analysis, and an extensive experiment for the accuracy of performance modeling and the benefit of performance optimization.

Bolz et al. [5] proposed one of the first SpMV CUDA [6] kernel implementations. Bell and Garland [1] implemented SpMV CUDA kernels for some well-known sparse matrix formats. Our modeling approach utilizes their implementation. Optimizing SpMV computation has been a challenge because SpMV computation is irregular and the fine-grained parallelism is hard to explore [7]. Im et al. [8] described optimization techniques to improve memory efficiency in SpMV for one or more vectors. Baskaran and Bordawekar [9] proposed optimizations including: synchronization-free parallelism, optimized thread mapping, optimized off-chip memory access, and data reuse, to speed up SpMV kernel. Demmel et al. [10] explored AEOS approach to automate the kernel optimization. We [11] proposed an auto-tuning framework that can automatically compute and select CUDA parameters for SpMV to obtain the optimal performance on specific GPUs. Vazquez et al. [12] proposed a new format, called ELLR-T, to achieve high performance on GPUs. Monakov et al. [13] proposed a sliced ELL format and used auto-tuning to find the optimal configuration, for example, the number of rows in a slice, to improve SpMV performance on GPUs. Grewe and Lokhmotov [14] presented a framework consisting of three components: a high-level representation for describing sparse matrix formats, a compiler for generating low-level code, and an automatic tuner, to improve SpMV performance. Wang et al. [15] outlined three optimizations including: optimized CSR storage format, optimized threads mapping, and avoid divergence judgment. Pichel et al. [16] explored the performance optimization of SpMV on GPUs using reordering techniques. Yang et al. [17] presented a novel non-parametric and self-tunable approach to data representation for computing SpMV, particularly targeting sparse matrices representing power-law graphs.

There are extensive work on performance models. Ryoo et al. [18] introduced two metrics (i.e., efficiency and utilization) to reduce optimization space. Their model focuses on pruning optimization space to reduce tuning time for a program. Choi et al. [19] designed a blocked ELLPACK format and proposed a model to predict matrix-dependent tuning parameters. Schaa and Kaeli [20] designed a methodology to accurately predict the execution time for a multi-GPU system according to that of a single GPU. Xu et al. [21] proposed the optimized SpMV based on ELL format and a SpMV CUDA performance model. Zhang and Owens [22] adopted a microbenchmark-based approach to develop a throughput model. Their model focuses on identifying

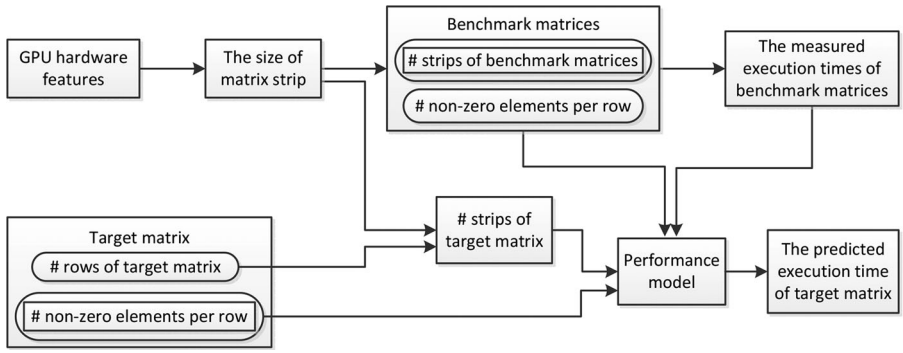


Fig. 1. Modeling workflow for SpMV CSR & ELL kernels.

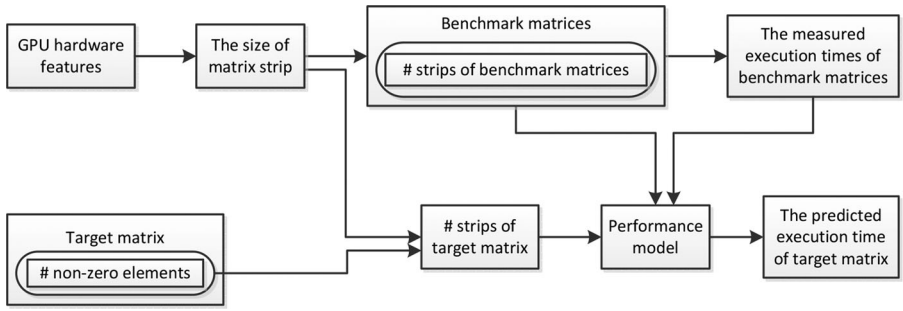


Fig. 2. Modeling workflow for SpMV COO kernel.

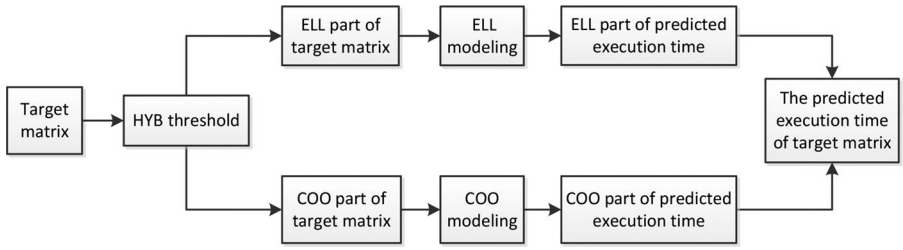


Fig. 3. Modeling workflow for SpMV HYB kernel.

performance bottlenecks and guiding programmers for optimization; our model focuses on predicting the execution time, which is similar to [23], [24], [25]. Baghsorkhi et al. [23] presented a compiler-based GPU performance modeling approach with accurate prediction using program analysis and symbolic evaluation techniques. Hong and Kim [24] proposed a simple analytical GPU model to estimate the execution time of massively parallel programs. Their model estimates the number of parallel memory requests by taking into account the number of running threads and memory bandwidth. Kothapalli et al. [25] presented a performance model by combining several known models of parallel computation: BSP, PRAM, and QRQW. However, their proposed analytical models are based on the abstraction of GPU architecture. Unlike these analytical performance models, our model is based on both analytical and profile-based modeling techniques.

3 SpMV Performance Modeling

The modeling workflows for CSR & ELL, COO, and HYB SpMV kernels are shown in Figs. 1, 2, and 3, respectively. As introduced in Section 1, our performance modeling consists of two phases:

Phase 1. Instrumenting:

- Compute the size of matrix strip (Section 3.1).
- Generate the benchmark matrices (Section 3.2).
- Test the execution times of the benchmark matrices (Section 3.3).
- Compute the number of matrix strips and non-zero elements per row (it is not applicable to COO) for a target matrix (Section 3.4).

Phase 2. Modeling:

- Instantiate parameterized models according to the experimental results of benchmark matrices.
- Estimate SpMV kernel execution times for a target matrix using performance models (Section 3.5).

The symbols used in our model are shown in Table 1.

3.1 The Size of Matrix Strip (S)

A *strip* is a maximum submatrix that can be handled by a GPU with a full load of thread blocks within one iteration [19]. For a large matrix, it may contain multiple strips. The size of matrix strip is determined by the physical limitations of GPUs (i.e., *NVIDIA GPUs*: # *Warps/Multiprocessor* and # *Threads/Multiprocessor*; *AMD*

TABLE 1
Symbols Used in Our Performance Modeling

N_{SM}	The number of streaming multiprocessors (SMs). (NVIDIA)
N_{CU}	The number of compute units (CUs). (AMD/ATI)
R	The number of rows of a benchmark matrix.
S	The size of a matrix strip, which is the maximum number of rows (for CSR and ELL) or non-zero elements (for COO) that can be processed by a GPU within one iteration.
I	The number of strips of a benchmark or target matrix.
$\mathcal{N}$	The set of natural numbers.
C	The number of columns of a benchmark matrix.
P_{NZ}	The number of non-zero elements per row of a benchmark or target matrix.
G_M	The size (bytes) of GPU global memory.
$M_{R\times C}$	A benchmark matrix, where $R \times C$ indicates the dimension of the benchmark matrix.
V_C	A random vector, where C indicates the dimension of the random vector.
$\phi_{(M\times V)}$	The execution time of matrix-vector multiplication, where M denotes the benchmark matrix and V denotes the random vector.
T	The execution time of a benchmark matrix.
N_R	The number of rows of a target matrix.
N_{NZ}	The number of non-zero elements of a target matrix.

(ATI) GPUs: # *Wavefronts/ComputeUnit* and # *WorkItems/ComputeUnit*) and SpMV kernel granularity, as shown in Table 2. The modeling approach presented in the remainder of Section 3 is general and can be applied to both NVIDIA and AMD GPU architectures. For SpMV CSR, ELL, and COO kernels, the sizes of matrix strip are computed as follows:

$$S_{CSR} = \begin{cases} N_{SM} \times Warps/Multiprocessor \\ N_{CU} \times Wavefronts/ComputeUnit. \end{cases}$$

$$S_{ELL} = S_{COO} = \begin{cases} N_{SM} \times Threads/Multiprocessor \\ N_{CU} \times WorkItems/ComputeUnit. \end{cases}$$

3.2 The Benchmark Matrices

3.2.1 The Criteria for Generating Benchmark Matrices

- The number of rows (R): $R = S \times I$
  - CSR: $S = S_{CSR}$, $I \in \mathcal{N}$
  - ELL: $S = S_{ELL}$, $I \in \mathcal{N}$
  - COO: $S = S_{COO}$, $I \in \mathcal{N}$
- The number of columns (C): $C > P_{NZ}$ is required
  - The value of C does not affect the performance since the sparse matrices are stored in compressed formats.
- The number of non-zero elements per row (P_{NZ}):
  - CSR: $P_{NZ} \in [1, \frac{G_M - sizeof(int)\times(R+1)}{(sizeof(float)+sizeof(int))\times R})$
  - ELL: $P_{NZ} \in [1, \frac{G_M}{(sizeof(float)+sizeof(int))\times R})$
  - COO: $P_{NZ} \in [1, \frac{G_M}{(sizeof(float)+2\times sizeof(int))\times R})$

  In benchmark matrices, we let each row have the same number of non-zero elements. In the above equations, we assume that the non-zero elements are in single-precision ($float$). For double-precision, the equations are similar. The maximum P_{NZ} is derived according to the maximum non-zero elements that can be stored in the GPU global memory in the corresponding sparse matrix format. The actual values of P_{NZ} used in our benchmarks are introduced in Section 3.2.2.
- The value of each non-zero element is random.

TABLE 2
SpMV Kernel Granularity

SpMV Kernel	Granularity
CSR	One warp (wavefront) per row
ELL	One thread (work-item) per row
COO	One thread (work-item) per non-zero element

3.2.2 The Experimental Setup

To obtain accurate performance models, we generate a series of benchmark matrices. A benchmark matrix is determined by R and P_{NZ}. Since $R = S \times I$, where S is fixed, we just enumerate values of I and P_{NZ} according to the above criteria to obtain combinations. Each combination indicates a benchmark matrix.

- The number of strips (I):
  - CSR and ELL: Let $I = 1, 2, 3, \ldots, 10$
    In our experiment, the largest benchmark matrix contains 10 strips, which is accurate enough to measure the performance.
  - COO: Let $I = 1$
    Since each non-zero element is handled by one thread, we just need to increase the number of non-zero elements per row for different benchmark matrices to duplicate strips instead of increasing the values of the number of strips.
- The number of non-zero elements per row (P_{NZ}):
  - CSR and ELL: Let $P_{NZ} = 4, 16 \ldots 1024, 2048 \ldots$
  - COO: Let $P_{NZ} = 10, 20, 30 \ldots 100$

  In our experiments, the above values are chosen for generating linear relationships introduced in Section 3.5.

3.3 The Execution Times of Benchmark Matrices (T)

We remove the effect of long initialization delay and average the execution time of a benchmark matrix as follows, where α and β are the number of executions, and $\alpha < \beta$.

$$T = \frac{\sum_{j=1}^{\beta} \phi_{((M_{R\times C})\times V_C)} - \sum_{j=1}^{\alpha} \phi_{((M_{R\times C})\times V_C)}}{\beta - \alpha}.$$

3.4 The Target Matrix

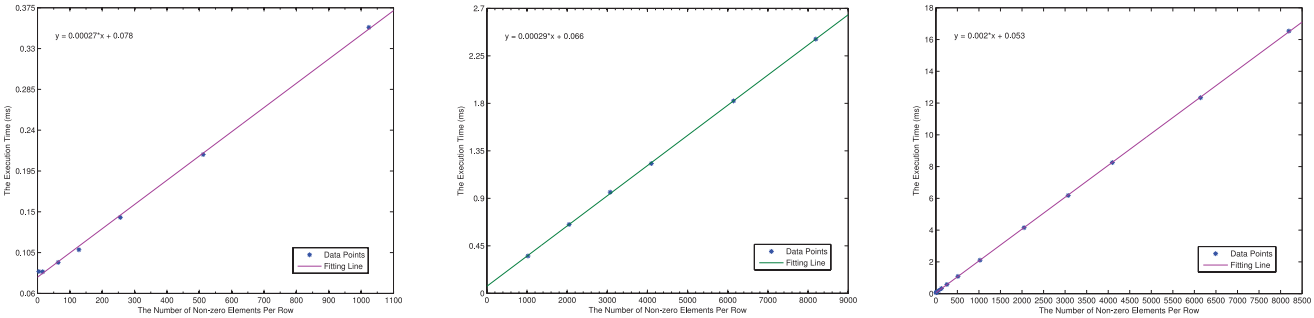
3.4.1 The Number of Strips (I)

Given a target matrix with N_R rows and N_{NZ} non-zero elements, the number of strips can be computed as follows:

$$I_{CSR} = \left\lceil \frac{N_R}{S_{CSR}} \right\rceil$$

$$I_{ELL} = \left\lceil \frac{N_R}{S_{ELL}} \right\rceil$$

$$I_{COO} = \left\lceil \frac{N_{NZ}}{S_{COO}} \right\rceil$$



(a) CSR: $P_{NZ} <= threshold$ (b) CSR: $P_{NZ} >= threshold$ (c) ELL

Fig. 4. The number of non-zero elements per row versus the execution time when I is fixed ((a)(b): $I = 5$, (c): $I = 1$).

3.4.2 Number of Non-Zero Elements Per Row (P_{NZ})

Let D be a set consisting of the number of non-zero elements in each row of a target matrix.

- CSR: P_{NZ} is set to be mode (in statistics) of a set D.
- ELL: P_{NZ} is set to be maximum value of a set D.

3.5 Performance Modeling and Estimating

There exists relationships between the number of strips, the number of non-zero elements per row, and the execution times of the benchmark matrices. Hence, we can estimate the SpMV kernel execution time of a target matrix according to these relationships.

3.5.1 CSR Kernel

Our method contains the following steps:

Step 1. Establish the following relationships:

- Relationship-1 ($T = E(x)$): For a set of benchmark matrices with the same number of strips (it can be any arbitrary value within the range defined in Section 3.2), we establish the relationship between the number of non-zero elements per row (x) and the execution time of the benchmark matrices (T), as shown in Figs. 4a and 4b.
- Relationship-2 ($T' = E'(y)$): For a set of benchmark matrices with the same number of non-zero elements per row, we establish the relationship between the number of strips (y) and the execution time of the benchmark matrices (T'), as shown in Figs. 5a and 5b.

By studying the physical limitations of NVIDIA Tesla C2050, we discovered that its number of max threads per block, i.e., 1,024, is exactly a threshold: when the number of non-zero elements per row is smaller or larger than it, the linear relationships are different.

Step 2. Estimate the execution time of a target matrix:

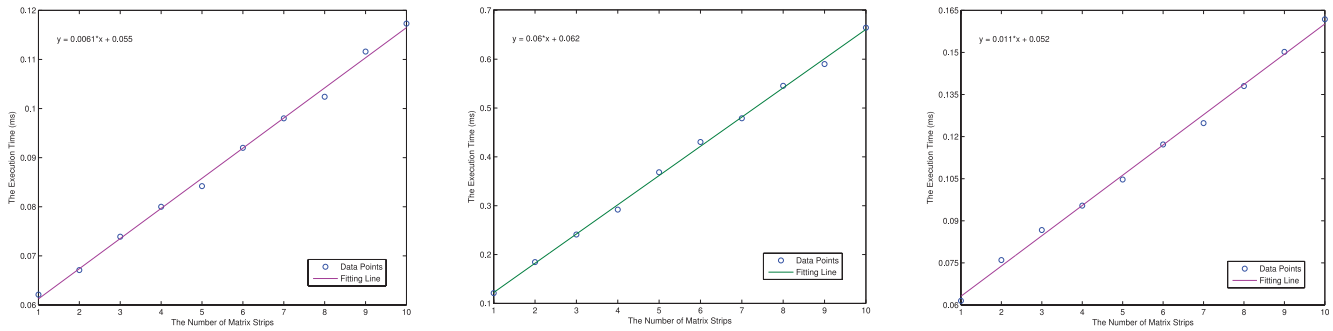
- According to the derived number of non-zero elements per row of the target matrix (denoted by x_0), derive T_1 by $T_1 = E(x_0)$ from Relationship-1, and the execution time T_2 of any previously tested benchmark matrix M.
- According to the number of strips of the target matrix (denoted by y_0), derive T_3 by $T_3 = E'(y_0)$ from a corresponding linear equation in Relationship-2. Note that, the number of non-zero elements per row of matrix M is set to be the number of non-zero elements per row in Relationship-2.
- Estimate the execution time of the target matrix (T_0) by $T_0 = (T_1/T_2) \times T_3$.

3.5.2 ELL Kernel

Our method works as follows:

Step 1. Establish the following relationships:

- Relationship-1 ($T = f(y_1) \times x + g(y_1)$): For a set of benchmark matrices with the same number of strips (it can be any arbitrary value within the range defined in Section 3.2 and denoted by y_1), we establish the relationship between the number of non-zero elements per row (x) and the execution time of the benchmark matrices (T), as shown in Fig. 4c.



(a) CSR: $P_{NZ} <= threshold$ (b) CSR: $P_{NZ} >= threshold$ (c) ELL

Fig. 5. The number of strips versus the execution time when P_{NZ} is fixed ((a): $P_{NZ} = 4$, (b): $P_{NZ} = 1{,}024$, (c): $P_{NZ} = 4$).

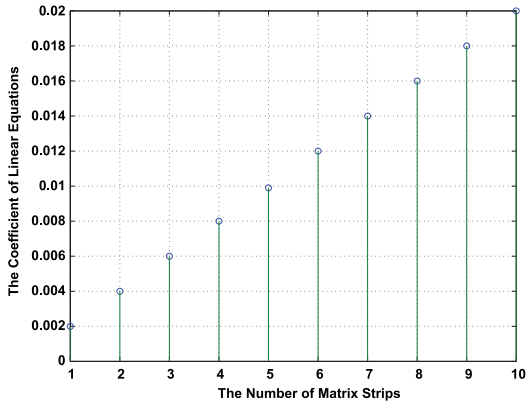


Fig. 6. The number of strips versus the linear coefficient (ELL).

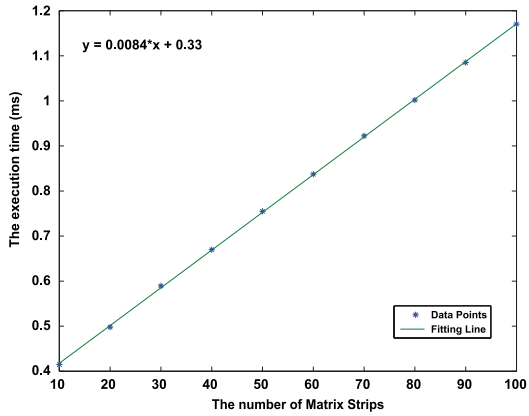


Fig. 7. The number of strips versus the execution time (COO).

- Relationship-2 ($f(y)$): For sets of benchmark matrices with different number of strips, we establish the relationship between the number of strips of the benchmark matrices (y) and the corresponding coefficient of the linear equations (f) in Relationship-1, as shown in Fig. 6.
- Relationship-3 ($e(y) = f(y) \times x_1 + g(y)$): For a set of benchmark matrices with the same number of non-zero elements per row (it can be any arbitrary value within the range defined in Section 3.2 and denoted by x_1), we establish the relationship between the number of strips (y) and the execution time of the benchmark matrices (e), as shown in Fig. 5c. Thus, $g(y) = e(y) - f(y) \times x_1$.

Step 2. Estimate the execution time of a target matrix:

- Given a target matrix, in order to estimate its execution time, we need to obtain the coefficient $f(Y)$ and the intercept $g(Y)$ of the linear equation, where Y is the number of strips of the target matrix. This can be done as follows:
  - According to the number of strips (Y) of the target matrix, obtain the coefficient of the linear equation from Relationship-2, i.e., $f(Y)$.
  - To obtain the intercept of the linear equation of the target matrix (i.e., $g(Y)$), we find the execution time (e) from Relationship-3 according to Y. Thus, $g(Y) = e(Y) - f(Y) \times Y$.
- Estimate the execution time of the target matrix (T_0) by $T_0 = f(Y) \times X + g(Y)$, where X and Y are the number of non-zero elements per row and the number of strips of the target matrix, respectively.

3.5.3 COO Kernel

Our method contains the following steps:

Step 1. Establish the following relationships:

- Relationship-1 ($T = E(x)$): We establish the relationship between the number of strips (x) and the execution time of the benchmark matrices (T), as shown in Fig. 7.

Step 2. Estimate the execution time of a target matrix:

- Count the total number of non-zero elements of the target matrix, then calculate the number of strips (x_0) according to Section 3.4.1.
- Estimate the execution time of the target matrix (T_0) using Relationship-1 by $T_0 = E(x_0)$.

3.5.4 HYB Kernel

Our method works as follows:

Step 1. Establish the following relationships:

- Since HYB kernel is the combination of ELL and COO kernels, as shown in Fig. 3, we can reuse the relationships in Sections 3.5.2 and 3.5.3.

Step 2. Estimate the execution time of a target matrix:

- Compute HYB threshold [1] to divide the target matrix into two parts: ELL and COO.
- Count the total number of non-zero elements of COO part of the target matrix.
- Calculate the number of strips of ELL (x_1) and COO parts (x_2) of the target matrix, respectively.
- Use HYB threshold (z_0) as the number of non-zero elements per row of ELL part (y_1) of the target matrix by $y_1 = z_0$.
- Estimate the execution times of ELL (T_1) and COO parts (T_2) of the target matrix by $T_1 = f(y_1) \times x_1 + g(y_1)$ and $T_2 = E(x_2)$, respectively.
- Estimate the execution time of the target matrix (T_0) by $T_0 = T_1 + T_2$.

4 SPMV Optimization Analysis

Based on the accurate performance modeling, we design a dynamic-programming based SpMV optimal solution auto-selection algorithm to automatically report an SpMV optimal solution for a given target sparse matrix. Specifically, given a target matrix, our algorithm searches its all potential storage strategies and checks whether the SpMV performance can be improved when the target matrix is partitioned into more than one matrix blocks and each one is stored in an appropriate storage format. If such a situation exists, the optimal solution, including the storage strategy, the storage format for each matrix block, as well as the predicted overall execution time, will be reported by the algorithm; otherwise, for the entire matrix, if a single storage format has the best SpMV performance, such a storage format, as well as its corresponding predicted overall execution time, will be reported as the optimal solution.

$$T[1,j] = \begin{cases} E[1,1], & \text{if j = 1} \\ \min\left\{ \min_{1 \le k < j} \{T[1,k] + E[k+1,j]\}, E[1,j] \right\}, & \text{if } 2 \le \text{j} \le \text{N} \end{cases}$$

Fig. 8. The recursive equation for computing $T[1,j]$.

Recall a target matrix can be partitioned into strips by rows. In the dynamic-programming based SpMV optimal solution auto-selection algorithm, one matrix block may contain one or more neighboring strips. Given a matrix M, let N and $[i,i]$ denote its number of strips and the i^{th} strip, respectively. Thus, the matrix block $[i,j]$ consists of continuous $\#(j-i+1)$ matrix strip(s), i.e., $[i,i]$, $[i+1, i+1], \ldots, [j,j]$, where $i = 1, 2, \ldots, N$ and $j = i,\ i+1, \ldots, N$. Given a matrix with N strips, there are at most $1/2 \times [N \times (N+1)]$ matrix blocks in total. In our experiments, we adopt 672 rows (i.e., S_{CSR} for NVIDIA Tesla C2050) as the size of a strip. However, if the number of rows of a target matrix is very large, instead, we can use 672×32 rows (i.e., S_{ELL} for NVIDIA Tesla C2050) as an alternative size to avoid huge number of matrix blocks. For each matrix block $[i,j]$, we first predict the kernel execution times for SpMV CSR, ELL, COO, and HYB kernels by our performance modeling, respectively. Here, each matrix block is used as a target matrix in the performance modeling approach presented in Section 3. Then, we store the least predicted execution time and its corresponding storage format into $E[i,j]$ and $F[i,j]$, respectively, which will be used as input of the algorithm to report an SpMV optimal solution for the entire target matrix.

The recursive equation is shown in Fig. 8. It defines the value of an optimal solution recursively in terms of the optimal solution to subproblems. Deriving recursive solution is an important step in developing a dynamic-programming based algorithm. Let $T[1,j]$ denote the predicted best performance of $\#j$ matrix strips starting from $[1,1]$ to $[j,j]$. To keep track of how to construct an optimal solution, we define $S[1,j]$ to store the value k at which $\#j$ matrix strips are split into two matrix blocks such that $T[1,j] = T[1,k] + E[k+1,j]$.

Based on the recursive equation, we could easily write an exponential-time recursive algorithm to compute optimal solution. However, the recursive algorithm repeatedly solves each subproblem, which involves lots of redundant computations. For example, in the recursive tree for computing $T[1,4]$, as shown in Fig. 9, $T[1,2]$ is computed twice and $T[1,1]$ is computed four times. The

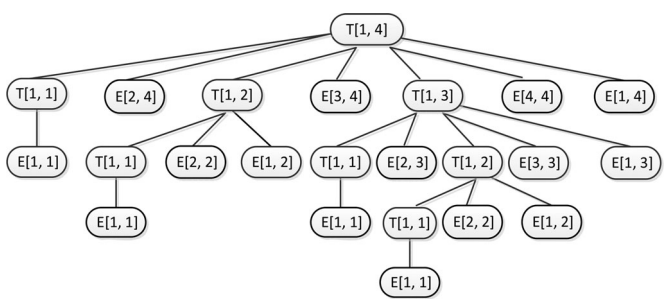


Fig. 9. The recursive tree for computing $T[1,4]$.

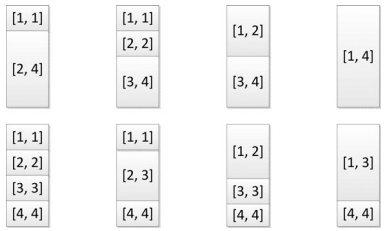


Fig. 10. Eight possible strategies for computing $T[1,4]$.

total time to compute $T[1,N]$ is $O(2^{N-1})$, which is exponential in N. The cost of such a recursive algorithm is very expensive when $\#N$ becomes large.

Fig. 11 shows our dynamic-programming based SpMV optimal solution auto-selection algorithm. The procedure SPMV_OPTIMAL_SOLUTION works as follows: Lines (01)-(02) compute $T[1,1]$ and $S[1,1]$. Then the *for* loop of lines (03)-(13) computes $T[1,j]$ and $S[1,j]$ for $j = 2, 3, \ldots, N$. Initially, $T[1,j]$ and $S[1,j]$ are set to be $E[1,j]$ and j in lines (04)-(05), respectively. Then, lines (06)-(12) replace the value of $T[1,j]$ and $S[1,j]$ with the minimum value of $T[1,k] + E[k+1,j]$ and its corresponding k,

```
SPMV_OPTIMAL_SOLUTION (E, N)
(01)   T[1,1] ← E[1,1]
(02)   S[1,1] ← 1
(03)   for j ← 2 to N do
(04)       T[1,j] ← E[1,j]
(05)       S[1,j] ← j
(06)       for k ← 1 to j − 1 do
(07)           q ← T[1,k] + E[k+1,j]
(08)           if q < T[1,j] then
(09)               T[1,j] ← q
(10)               S[1,j] ← k
(11)           endif
(12)       endfor
(13)   endfor
(14)   Print "Optimal Time: T[1,N]"
(15)   PRINT_OPTIMAL_STRATEGY (N, S)

PRINT_OPTIMAL_STRATEGY (j, S)
(01)   if j = 1 then
(02)       Print "Optimal Strategy: [1,1]"
(03)       Print "Optimal Format: F[1,1]"
(04)   else if S[1,j] = 1 then
(05)       Print "Optimal Strategy: [1,1], [2,j]"
(06)       Print "Optimal Format: F[1,1], F[2,j]"
(07)       else if S[1,j] = j then
(08)           Print "Optimal Strategy: [1,j]"
(09)           Print "Optimal Format: F[1,j]"
(10)           else
(11)             PRINT_OPTIMAL_STRATEGY (S[1,j],S);
(12)             Print ",[S[1,j]+1,j], F[S[1,j]+1,j]"
(13)           endif
(14)       endif
(15)   endif
```

where:

- N is the number of matrix strips;
- $E[i,j]$ stores the minimum predicted execution time of the matrix block $[i,j]$;
- $F[i,j]$ stores optimal storage format (*e.g.*, CSR, ELL, COO, or HYB) of the matrix block $[i,j]$;
- $T[1,j]$ stores the minimum predicted execution time of $\#j$ matrix strips starting from $[1,1]$ to $[j,j]$;
- $S[1,j]$ records the index k which splits $\#j$ matrix strips starting from $[1,1]$ to $[j,j]$ into two matrix blocks: $[1,k]$ and $[k+1,j]$. Note that if k equals to j, all strips from $[1,1]$ to $[j,j]$ are in the same matrix block.

Fig. 11. SpMV optimal solution auto-selection algorithm.

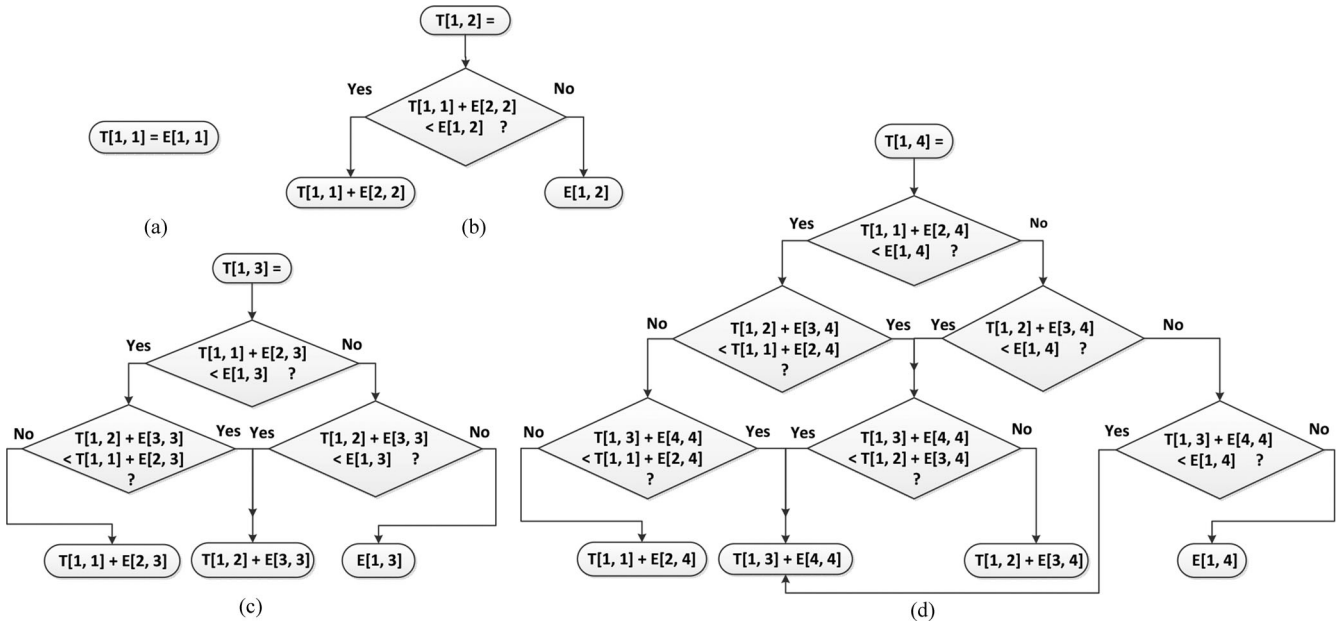


Fig. 12. The demo diagram of SpMV optimal solution auto-selection algorithm for computing $T[1, 4]$.

TABLE 3
Optimal Solutions of 22 Sparse Matrices

Matrix	Optimal Strategy	Optimal Format (s)	Matrix	Optimal Strategy	Optimal Format (s)
linverse	[1, 18]	ELL/HYB	FEM/Cantilever	[1, 93]	ELL
ex11	[1, 25]	ELL	FEM/Harbor	[1, 70]	CSR
neos	[1, 713]	HYB	QCD	[1, 74]	ELL/HYB
finan512	[1, 74], [75, 112]	ELL, ELL	FEM/Ship	[1, 210]	HYB
nasasrb	[1, 1], [2, 82]	CSR, ELL	Epidemiology	[1, 783]	ELL/HYB
hcircuit	[1, 28], [29, 158]	CSR, ELL/HYB	Wind Tunnel	[1, 325]	HYB
OPF_6000	[1, 42], [43, 45]	ELL, CSR	Economics	[1, 308]	HYB
OPF_10000	[1, 59], [60, 66]	ELL, CSR	FEM/Accelerator	[1, 32], [33, 181]	ELL, ELL/HYB
Dense	[1, 3]	CSR	Circuit	[1, 255]	HYB
Protein	[1, 55]	CSR	Webbase	[1, 1489]	HYB
FEM/Spheres	[1, 125]	ELL/HYB	LP	[1, 7]	CSR

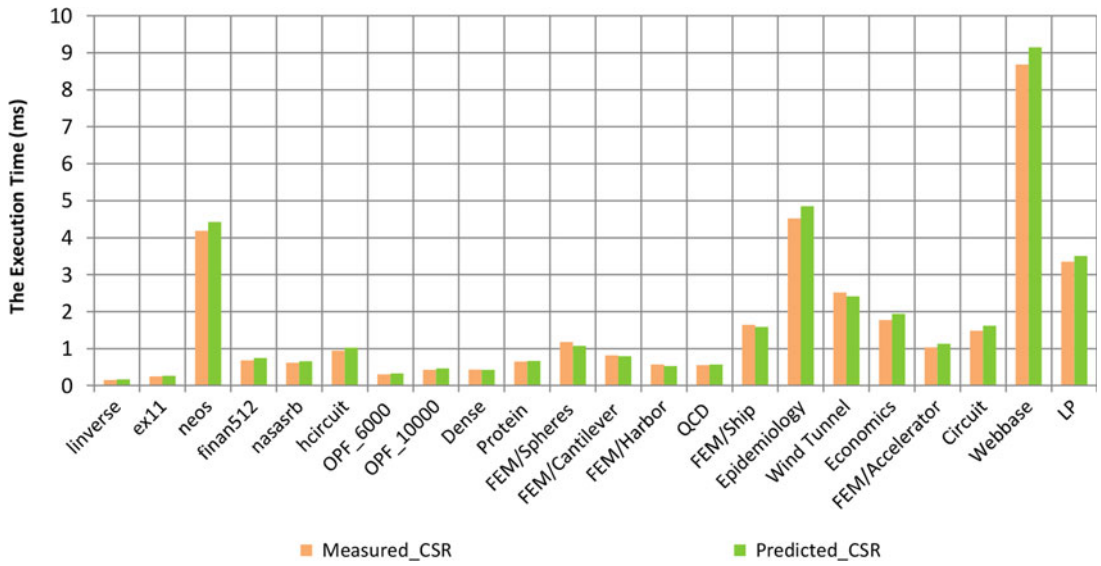


Fig. 13. Accuracy evaluation on CSR kernel.

respectively, if the minimum value of $T[1, k] + E[k + 1, j]$ is less than the value of $T[1, j]$. Finally, lines (14)-(15) print the best performance $T[1, N]$ and its corresponding storage strategy, using the procedure PRINT_OPTIMAL_STRATEGY. The time complexity of the entire procedure SPMV_OPTIMAL_SOLUTION is $O(N^2)$.

Compared to the recursive algorithm, which is in top-down fashion, the bottom-up dynamic-programming

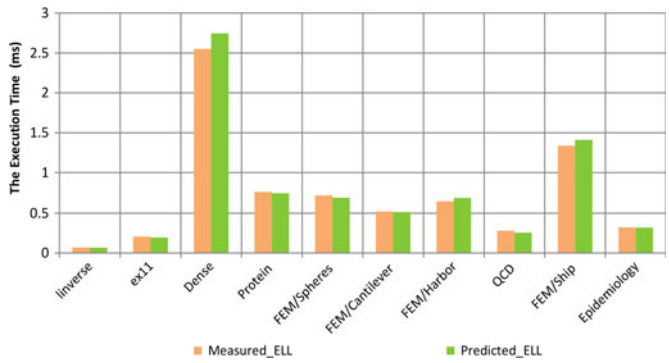


Fig. 14. Accuracy evaluation on ELL kernel.

based algorithm is more efficient because it solves each subproblem exactly once. Fig. 12 shows an example of computing $T[1,4]$ using the bottom-up dynamic-programming based algorithm. As shown in Figs. 12a, 12b, and 12c, before computing $T[1,4]$, the values of $T[1,1]$, $T[1,2]$, and $T[1,3]$ should be computed exactly once and stored in table T. Hence, we can determine the value of $T[1,4]$ by looking up table T, as shown in Fig. 12d. Compared to the enumerative algorithm, the dynamic-programming based algorithm is more efficient since it searches for the optimal solution from N partitioning strategies for a matrix with $\#N$ matrix strips, instead of searching all of 2^{N-1} possible strategies. For example, given $T[1,1]$, $T[1,2]$, and $T[1,3]$, to compute $T[1,4]$, the dynamic-programming based algorithm only needs to check four partitioning strategies, i.e., $T[1,1]+E[2,4]$, $T[1,2]+E[3,4]$, $T[1,3]+E[4,4]$, and $E[1,4]$, as shown at the bottom of Fig. 12d, instead of check all of eight possible strategies, as shown in Fig. 10.

5 Experimental Evaluation

Our experimental evaluation focuses on two aspects: 1) the accuracy of performance modeling; 2) the benefit of SpMV optimization analysis tool. Our experiments are performed on NVIDIA Tesla C2050 with 3 GB global memory. The version of CUDA library we use in our experiments is 4.1. We evaluate our tool on 22 matrices from the sparse matrices collection [26], [27], as shown in Table 3. All SpMV CUDA kernels are based on NVIDIA's implementation [1]. However, the experiments of ELL SpMV CUDA kernel are conducted only on 10 out of 22 sparse matrices on NVIDIA Tesla C2050 because of the limitation of "*num_cols_per_row*" in the code.

5.1 Accuracy of Performance Modeling

The accuracy of performance modeling is critical since it is the basis for accurately reporting optimal solutions and performance optimization. To evaluate it, we focus on two aspects: the execution time and the performance difference rate. Figs. 13, 14, 15, and 16 show the comparisons between the measured execution times of CSR, ELL, COO, and HYB SpMV kernels and the times predicted by our performance models, respectively. The measured execution times are obtained by averaging the total measured execution times of the SpMV kernel for 500 times. Note that, in our experiments, the GPU's warm up time is excluded. Therefore, the measured execution times are relative stable and accurate. The measured and predicted execution times of optimal solutions (denoted by OPT, which is elaborated in Section 5.2) are shown in Fig. 18. Fig. 17 shows the absolute performance difference percentage rates (i.e., the difference between the predicted and measured execution times divided by measured execution times). There are totally 82 test cases, which include the evaluation of CSR, COO, and HYB kernels on a collection of 22 sparse matrices, the evaluation of ELL kernel on 10 sparse matrices within the collection, and the evaluation of optimal solutions on six sparse matrices within the collection. The execution times predicted by our model match the measured times very well. Specifically, for 77 out of 82 test cases, the performance differences between the predicted and measured execution times are less than 9 percent. For the rest five test cases, the differences are between 9 and 10

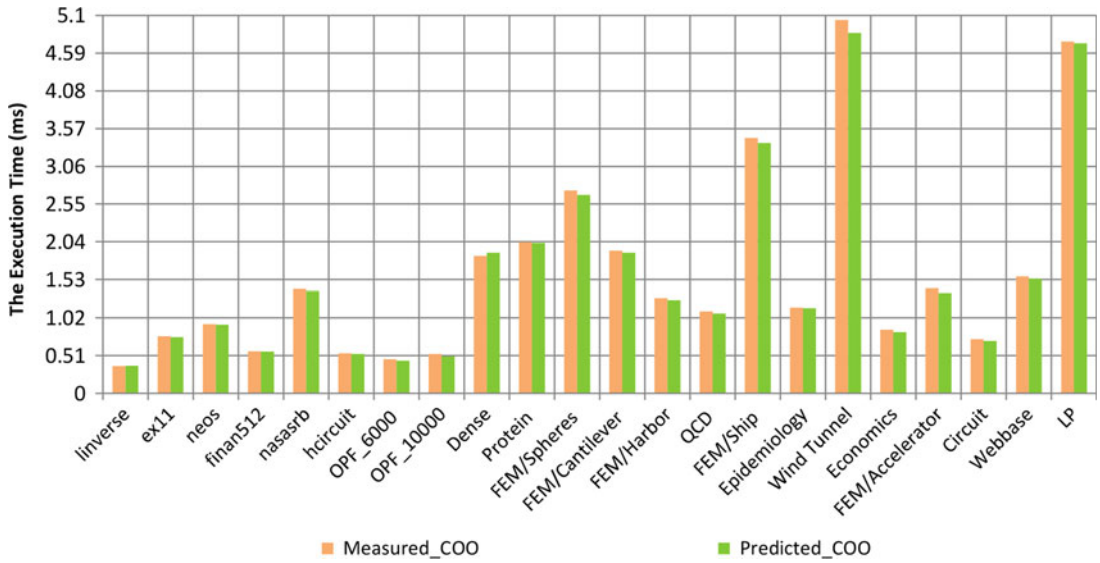


Fig. 15. Accuracy evaluation on COO kernel.

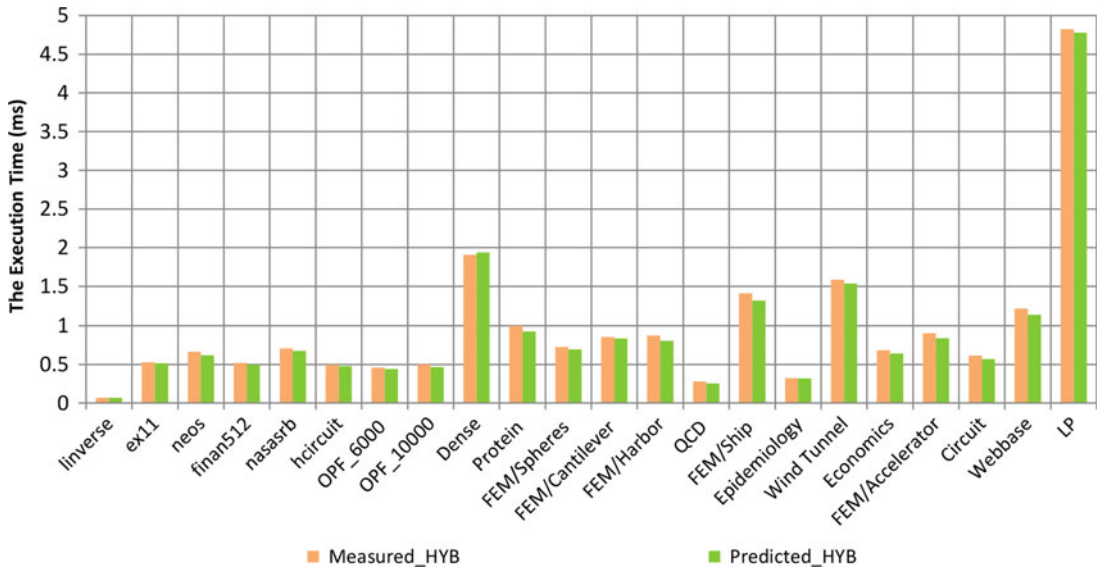


Fig. 16. Accuracy evaluation on HYB kernel.

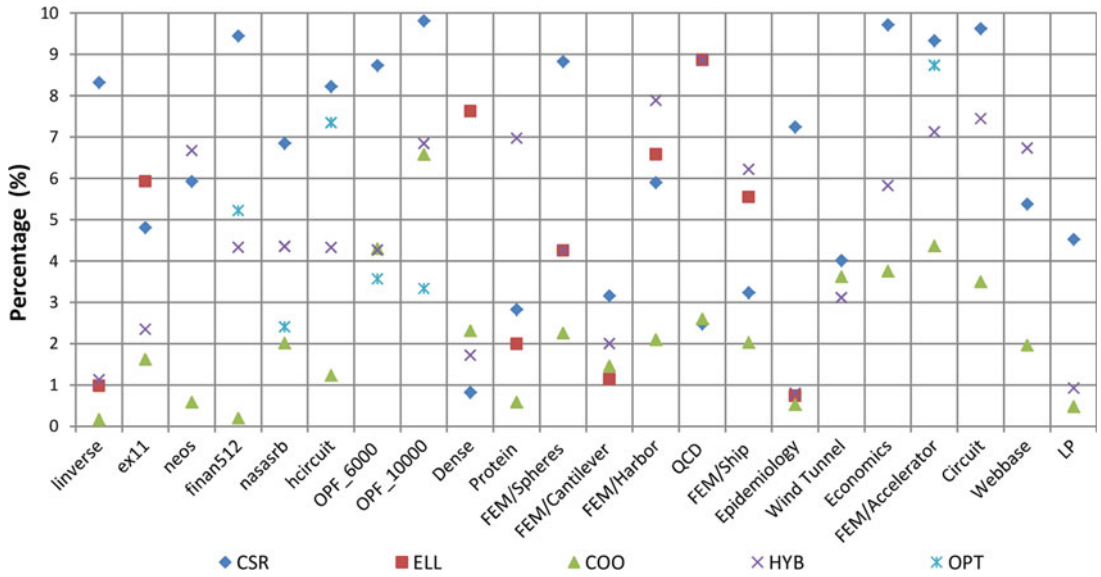


Fig. 17. Comparision of performance difference rates.

percent. For CSR, ELL, COO, and HYB SpMV kernels, the average differences are 6.3, 4.4, 2.2, and 4.7 percent, respectively. For the optimal solutions, the average difference is 5.1 percent.

5.2 Benefit of SpMV Optimization Analysis Tool

5.2.1 *Reporting Optimal Solutions*

Table 3 shows OPT (i.e., optimal storage strategy and storage format(s)) for 22 target matrices, which are predicted by the SpMV optimal solution auto-selection algorithm in Fig. 11. In the reported optimal solutions, matrices "finan512", "nasasrb", "hcircuit", "OPF_6000", "OPF_10000", and "FEM/Accelerator" are partitioned into two blocks. For example, matrix "OPF_10000" has 66 matrix strips in total. Our tool suggests partitioning it into two blocks: [1, 59] and [60, 66], where the partitioning position is the last row of the 59th matrix strip. In addition, the optimal formats reported by our tool for

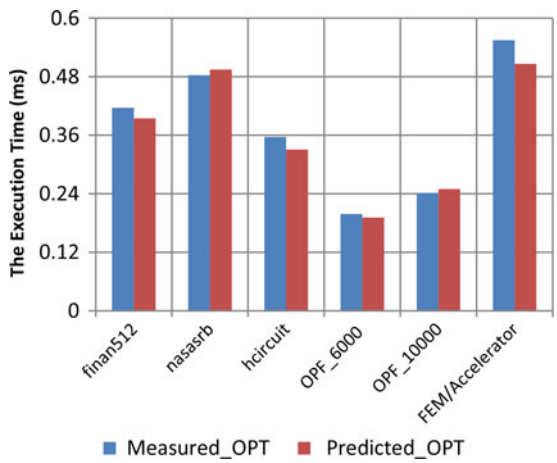


Fig. 18. Accuracy evaluation on OPT.

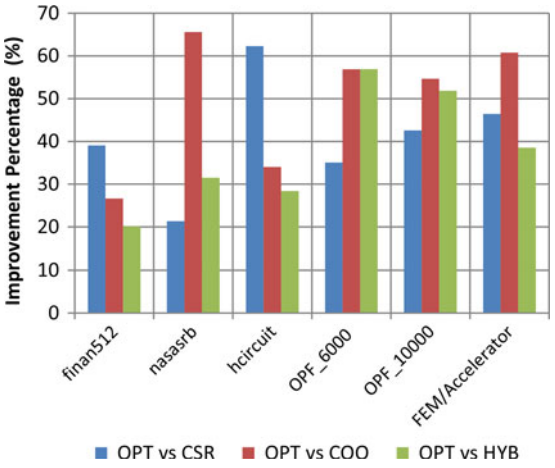


Fig. 19. OPT performance improvements.

blocks [1, 59] and [60, 66] are ELL and CSR, respectively. Except for the above six matrices, our tool reports that all other matrices only have a single storage format as their own optimal solutions. Specifically, for matrices "Dense", "Protein", "FEM/Harbor", and "LP", the optimal storage format is CSR; for matrix "neos", "FEM/Ship", "Wind Tunnel", "Economics", "Circuit", and "Webbase", it is HYB; for matrices "linverse", "FEM/Spheres", "QCD", and "Epidemiology", it is ELL or HYB (denoted by ELL/HYB); for matrices "ex11" and "FEM/Cantilever", it is ELL. Note that, for matrix "FEM/Ship", because the performance difference between the measured execution times of ELL and HYB kernels is very small, and additionally, there exists small difference between the measured and predicted execution times, although its optimal storage format reported by our tool is HYB, ELL has the best performance in the real execution. However, the real performance difference for ELL and HYB is very small, i.e., 5.5 versus 6.2 percent.

5.2.2 Optimizing SpMV Performance

Fig. 19 shows the performance improvement evaluation on matrices "finan512," "nasasrb," "hcircuit," "OPF_6000," "OPF_10000," and "FEM/Accelerator" by comparing the measured execution times of optimal solutions and CSR, COO, and HYB kernels. For example, the optimal solution on matrix "hcircuit" can achieve 62.3, 34.0, and 28.4 percent performance improvement, respectively, compared to the measured execution times of CSR, COO, and HYB kernels. Generally, the average performance improvements on six matrices are 41.1, 49.8, and 37.9 percent, respectively.

6 Conclusion and Future Work

In this paper, we present a performance modeling and optimization analysis tool to predict and optimize SpMV performance on GPUs. Our tool includes four performance models, CSR, ELL, COO, and HYB models, to accurately predict the execution times of SpMV kernels by utilizing an integrated analytical and profile-based performance modeling approach. It also includes an efficient dynamic-programming based SpMV optimal solution auto-selection algorithm to automatically report an SpMV optimal solution (i.e., optimal storage strategy, storage format(s), and execution time) for a target matrix. The proposed approach in our tool is general, and neither limited by GPU programming languages nor restricted to specific GPU architectures. In the future work, we will extend the current SpMV performance modeling to handle multi-GPU SpMV kernels.

Acknowledgments

The work was supported in part by NSF under Grants 0941735, CAREER-1054834, and by the Graduate Assistantship of the School of Energy Resources at the University of Wyoming.

References

[1] N. Bell and M. Garland, "Implementing Sparse Matrix-Vector Multiplication On Throughput-Oriented Processors," *Proc. Conf. High Performance Computing Networking, Storage and Analysis (SC '09)*, pp. 1-11, 2009.

[2] A. Resios and V. Holdermans, "GPU Performance Prediction Using Parametrized Models,", Master's thesis, Utrecht Univ., 2011.

[3] P. Guo and L. Wang, "Accurate CUDA Performance Modeling for Sparse Matrix-Vector Multiplication," *Proc. IEEE Int'l Conf. High Performance Computing and Simulation (HPCS '12)*, pp. 496-502, July 2012.

[4] P. Guo, H. Huang, Q. Chen, L. Wang, E.-J. Lee, and P. Chen, "A Model-Driven Partitioning and Auto-Tuning Integrated Framework for Sparse Matrix-Vector Multiplication on GPUs," *Proc. TeraGrid Conf. Extreme Digital Discovery (TG '11)*, pp. 2:1-2:8, 2011.

[5] J. Bolz, I. Farmer, E. Grinspun, and P. Schroder, "Sparse Matrix Solvers on The GPU: Conjugate Gradients and Multigrid," *ACM Trans. Graphics*, vol. 22, no. 3, pp. 917-924, 2003.

[6] *NVIDIA CUDA C Programming Guide, Version 4.0*, May 2011.

[7] J. Kurzak, W. Alvaro, and J. Dongarra, "Optimizing Matrix Multiplication for a Short-Vector Simd Architecture-Cell Processor," *J. Parallel Computing*, vol. 35, no. 3, pp. 138-150, 2009.

[8] E.-J. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization Framework for Sparse Matrix Kernels," *Int'l J. High Performance Computing Applications*, vol. 18, no. 1, pp. 135-158, 2004.

[9] M.M. Baskaran and R. Bordawekar, "Optimizing Sparse Matrix-Vector Multiplication on GPUs," Research Report RC24704, IBM TJ Watson Research Center, Dec. 2008.

[10] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R.C.W.R. Vuduc, and K. Yelick, "Self-Adapting Linear Algebra Algorithms and Software," *Proc. IEEE*, vol. 93, no. 2, pp. 293-312, Feb. 2005.

[11] P. Guo and L. Wang, "Auto-Tuning CUDA Parameters for Sparse Matrix-Vector Multiplication on GPUs," *Proc. Int'l Conf. Computational and Information Sciences (ICCIS '10)*, pp. 1154-1157, 2010.

[12] F. Vazquez, G. Ortega, J.J. Fernandez, and E.M. Garzon, "Improving the Performance of the Sparse Matrix Vector Product with GPUs," *Proc. 10th IEEE Int'l Conf. Computer and Information Technology (CIT '10)*, pp. 1146-1151, 2010.

[13] A. Monakov, A. Lokhmotov, and A. Avetisyan, "Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures," *Proc. Fifth Int'l Conf. High Performance Embedded Architectures and Compilers (HiPEAC '10)*, pp. 111-125, 2010.

[14] D. Grewe and A. Lokhmotov, "Automatically Generating and Tuning Gpu Code for Sparse Matrix-Vector Multiplication from a High-Level Representation," *Proc. ACM Fourth Workshop General Purpose Processing on Graphics Processing Units (GPGPU-4)*, pp. 12:1-12:8, 2011.

[15] Z. Wang, X. Xu, W. Zhao, Y. Zhang, and S. He, "Optimizing Sparse Matrix-Vector Multiplication on CUDA," *Proc. Second Int'l Conf. Education Technology and Computer (ICETC '10)*, vol. 4, pp. V4-109-V4-113, June 2010.

[16] J.C. Pichel, F.F. Rivera, M. Fernandez, and A. Rodriguez, "Optimization of Sparse Matrix-Vector Multiplication Using Reordering Techniques on GPUs," *Microprocessors and Microsystems*, vol. 36, no. 2, pp. 65-77, 2012.

[17] X. Yang, S. Parthasarathy, and P. Sadayappan, "Fast Sparse Matrix-Vector Multiplication on GPUs: Implications for Graph Mining," *Proc. VLDB Endowment*, vol. 4, no. 4, pp. 231-242, Jan. 2011.
[18] S. Ryoo, C.I. Rodrigues, S.S. Stone, S.S. Baghsorkhi, S.-Z. Ueng, J.A. Stratton, and W.-m.W. Hwu, "Program Optimization Space Pruning for a Multithreaded GPU," *Proc. ACM Sixth Ann. IEEE/ACM Int'l Symp. Code Generation and Optimization (CGO '08)*, pp. 195-204, 2008.
[19] J.W. Choi, A. Singh, and R.W. Vuduc, "Model-Driven Autotuning of Sparse Matrix-Vector Multiply on GPUs," *Proc. 15th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP '10)*, pp. 115-126, 2010.
[20] D. Schaa and D. Kaeli, "Exploring the multiple-GPU Design Space," *Proc. IEEE Int'l Parallel & Distributed Processing Symp. (IPDPS '09)*, pp. 1-12, May 2009.
[21] S. Xu, W. Xue, and H. Lin, "Performance Modeling and Optimization of Sparse Matrix-Vector Multiplication on NVIDIA CUDA Platform," *J. Supercomputing*, vol. 63, pp. 710-721, 2013.
[22] Y. Zhang and J. Owens, "A Quantitative Performance Analysis Model for GPU Architectures," *Proc. IEEE 17th Int'l Symp. High Performance Computer Architecture (HPCA '11)*, pp. 382-393, Feb. 2011.
[23] S.S. Baghsorkhi, M. Delahaye, S.J. Patel, W.D. Gropp, and W.-M.W. Hwu, "An Adaptive Performance Modeling Tool for GPU Architectures," *Proc. 15th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP '10)*, pp. 105-114, 2010.
[24] S. Hong and H. Kim, "An Analytical Model for a GPU Architecture with Memory-Level and Thread-Level Parallelism Awareness," *Proc. 36th ACM Ann. Int'l Symp. Computer Architecture (ISCA '09)*, pp. 152-163, 2009.
[25] K. Kothapalli, R. Mukherjee, M. Rehman, S. Patidar, P. Narayanan, and K. Srinathan, "A Performance Prediction Model for the CUDA GPGPU Platform," *Proc. Int'l Conf. High Performance Computing (HiPC '09)*, pp. 463-472, Dec. 2009.
[26] T.A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Software*, vol. 38, no. 1, pp. 1:1-1:25, 2011.
[27] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms," *Proc. ACM/IEEE Conf. Supercomputing*, 2007.

Ping Guo received the BS degree in computer science from Harbin University of Science and Technology, China, in 2005, and the MS degree in computer science from the University of Kentucky, in 2008. She is working toward the PhD degree in the Department of Computer Science at the University of Wyoming. Her research interests include designing high-performance computing (HPC) system on graphics processing unit (GPU) platform. She is a student member of the IEEE.

Liqiang Wang received the BS degree in mathematics from Hebei Normal University, China, in 1995, the MS degree in computer science from Sichuan University, China, in 1998, and the PhD degree in computer science from Stony Brook University in 2006. He is an associate professor in the Department of Computer Science at the University of Wyoming. His research interests include the design and analysis of data-intensive parallel computing systems, including GPU and cloud computing. He received a US National Science Foundation CAREER Award in 2011. He is a member of the IEEE.

Po Chen received the PhD degree in geological sciences from the University of Southern California in 2005. He is currently an assistant professor at the Department of Geology and Geophysics, University of Wyoming. His research interests include the theoretical and computational aspects of seismological inverse problems related to energy exploration, earthquake hazard analysis/mitigation, and the dynamics of the solid earth.