A Model-Driven Partitioning and Auto-tuning Integrated Framework for Sparse Matrix-Vector Multiplication on GPUs[*]

Ping Guo, He Huang, Qichang Chen, Liqiang Wang
Department of Computer Science
University of Wyoming, USA
{pguo, hhuang1, qchen2, lwang7}@uwyo.edu

En-Jui Lee, Po Chen
Department of Geology and Geophysics
University of Wyoming, USA
{elee8, pchen}@uwyo.edu

ABSTRACT

Sparse Matrix-Vector Multiplication (SpMV) is very common to scientific computing. The Graphics Processing Unit (GPU) has recently emerged as a high-performance computing platform due to its massive processing capability. This paper presents an innovative performance-model driven approach for partitioning sparse matrix into appropriate formats, and auto-tuning configurations of CUDA kernels to improve the performance of SpMV on GPUs. This paper makes the following contributions: (1) Propose an empirical CUDA performance model to predict the execution time of SpMV CUDA kernels. (2) Design and implement a model-driven partitioning framework to predict how to partition the target sparse matrix into one or more partitions and transform each partition into appropriate storage format, which is based on the fact that the different storage formats of sparse matrix can significantly affect the performance of SpMV. (3) Integrate the model-driven partitioning with our previous auto-tuning framework to automatically adjust CUDA-specific parameters to optimize performance on specific GPUs. Compared to the NVIDIA's existing implementations, our approach shows a substantial performance improvement. It has 222%, 197%, and 33% performance improvement on the average for CSR vector kernel, ELL kernel and HYB kernel, respectively.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling techniques; D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Measurement, Performance

Keywords

GPU, performance modeling, auto-tuning, Sparse Matrix-Vector Multiplication

1. INTRODUCTION

Sparse matrix-vector multiplication (SpMV) is an essential operation in solving linear systems and partial differential equations in many scientific and engineering applications. For many applications, the matrices can be very large and sparse. Optimizing SpMV performance is a challenging problem because data access locality is often not observed and fine-grained loop parallelism is hard to explore [1]. Researchers have investigated many ways to speed up SpMV using hardware advances, such as multi-core processors [2] and many-core Graphics Processing Units (GPUs) [3].

GPUs are specifically designed to provide high throughput on parallel computations using many-core chips. GPU supports graphics-related computations, which can be adopted for numerical analysis in scientific computations. Due to the high computation power, GPU has become an attractive coprocessor for general purpose computing. GPGPU (General Purpose Computation on Graphics Processing Units) is especially well-suited to address problems that can be expressed as data-parallel computations, *i.e.*, the same program is executed on many data elements in parallel. Because the same program is executed on many data elements and the kernel has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches. The kernel does not automatically have high arithmetic intensity. It should have high intensity to mask data load latency. If the kernel is not arithmetically intense, the load latency will be exposed and kernel performance will suffer.

Recently, Bell and Garland [3], and Baskaran and Bordawekar [4] proposed and implemented SpMV kernels using CUDA [5] for different sparse matrix formats, including DIA, ELLPACK, CSR, COO, and a hybrid ELL/COO format. The information in detail about these matrix formats is introduced in Section 3. Based on our experiments on unstructured sparse matrices, CSR usually has the best performance on sparse matrices with large number of non-zero elements. ELLPACK is usually good for a sparse matrix with nearly

[*]The work was supported in part by NSF under Grants 0941735, 0930040, CAREER-1054834, and by the Graduate Assistantship of the School of Energy Resources at the University of Wyoming.

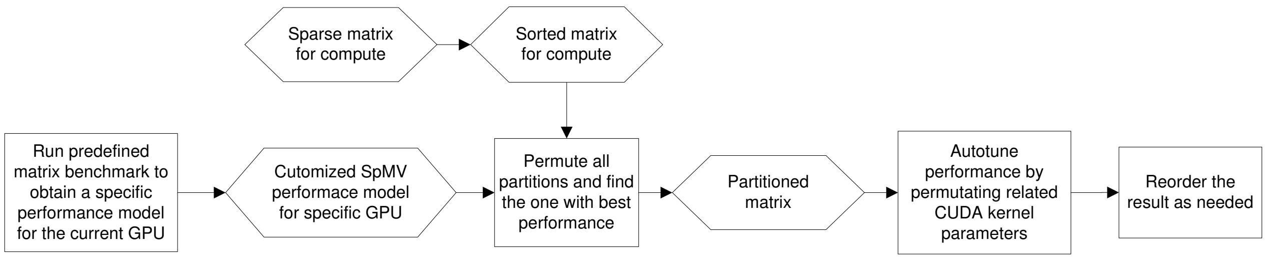
Sparse matrix for compute
Sorted matrix for compute
Run predefined matrix benchmark to obtain a specific performance model for the current GPU
Cutomized SpMV performace model for specific GPU
Permute all partitions and find the one with best performance
Partitioned matrix
Autotune performance by permutating related CUDA kernel parameters
Reorder the result as needed

Figure 1: The workflow of our tool to automatically partition and tune SpMV CUDA kernels. A rectangle represents a certain operation in our framework; all the rest represent the input matrix or intermediate results.

equal and small number of non-zero elements per row. The hybrid ELL/COO format 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 much larger non-zero elements. The above observation motivates us to design an automatic approach to improve SpMV performance by partitioning and formatting sparse matrices. Specifically, we design a GPU-specific performance model to estimate the running time for a matrix in a specific storage format. Thus, given a sparse matrix, we may partition it, then predict the execution time for each matrix strip in different formats. Hence, we are able to obtain the optimal partitioning and formatting to optimize the performance.

The workflow of our tool is shown in Figure 1. Specifically, it contains the following steps. Given a specific GPU, we execute a series of matrix benchmarks predefined in our performance model. The experimental results are used to instantiate the parameters of our performance model. Thus, the customized performance model can be used for predicting the execution time of SpMV in CSR, ELL, and HYB formats on the given GPU. Given a matrix, we first sort it based on the number of non-zero elements per row. Then we use the above performance model to find an optimal partitioning and transform each partition into a specific sparse storage format. During the computation, the tool automatically adjusts CUDA parameters (*e.g.*, *Num_Threads*, *Block_Size*, and $Warp\_size$) to optimize performance. The final results can be adjusted back as needed according to the original order of rows. Actually, this step is often unnecessary because we usually do not care about the order of row, where a row often represents an equation in the linear system.

This paper makes the following contributions: (1) Propose an empirical CUDA performance model to predict the execution time of SpMV CUDA kernels. (2) Design a model-driven partitioning framework to predict how to partition the target sparse matrix into one or more partitions and transform each partition into appropriate storage format, which is based on the fact that the different storage formats of sparse matrix can significantly affect the performance of SpMV. (3) Integrate the model-driven partitioning with our previous auto-tuning framework to automatically adjust CUDA-specific parameters to optimize performance on specific GPUs. We evaluated our implementation on a collection of widely-used sparse matrix benchmarks. Compared to NVIDIA's CSR, ELL, and HYB kernels [3], our approach has 222%, 197%, and 33% performance improvement on the average, respectively.

The rest of this paper is organized as follows: Section 2 surveys the related work. Section 3 discusses the widely-used space-efficient sparse matrix formats including CSR, COO, ELL, as well as HYB format. Section 4 introduces our empirical GPU performance model. Section 5 discusses how to find an optimal matrix partitioning. Section 6 presents how to auto-tune the parameters to optimize performance of SpMV. Section 7 evaluates the performance of our tool on a collection of benchmarks. Section 8 gives the conclusion and future work.

2. RELATED WORK

There are extensive work on SpMV optimization and tuning. Interested readers may refer to some surveys [6, 7].

As an emerging powerful massively parallel computing platform, GPU has also been used to speed up SpMV. Bolz *et al.* first applied GPU computing to SpMV [8]. Their experiment shows that the performance of their implementation on NVIDIA's GeForce FX is nearly two times faster than the execution on a 3GHz Pentium 4.

The most related work to this paper are [9] and [3]. Bell and Garland implemented SpMV kernel in CUDA for different sparse matrix formats, including DIA, ELLPACK, CSR, COO, and HYB format [3]. Our SpMV kernels are based on their implementation [3, 10]. Choi proposed and implemented a blocked ELLPACK (BELLPACK) storage formats and a performance model [9]. By directly modeling the structure of the streaming multiprocessors (SMs), the GPU-specific execution time model can accurately predict suitable tuning parameters for SpMV using the BELLPACK format. Compared to it, our model can handle more storage formats and guide matrix partitioning and auto-tune GPU runtime parameters.

Due to the hardware characteristics and restrictions, the performance of CUDA kernels is instable. Auto-tuning technique could be applied to optimize the performance. Nukada and Matsuoka [11] presented an auto-tuning framework that can automatically choose the optimal number of threads for the CUDA-based 3-D FFT library by exhausting the state space of all possible numbers of threads to find out the best number of threads. In addition, they also optimized memory access by inserting padding to 3-D FFT matrices to reduce the memory bank conflicts. Their experiments show that the approach achieves at least 5.0 times speedup over the NVIDIA's CUFFT library. Our approach can not only find out an optimal GPU runtime configuration, such as the number of threads, but also find out an optimal partition and storage format.

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

(a) An example Matrix

$$\text{data} = \begin{bmatrix} 2 & 5 & 3 & * \\ 6 & 1 & * & * \\ 8 & 2 & 7 & 9 \\ 7 & 4 & * & * \\ 8 & 4 & * & * \end{bmatrix} \quad \text{indices} = \begin{bmatrix} 0 & 1 & 3 & * \\ 1 & 2 & * & * \\ 0 & 1 & 2 & 3 \\ 0 & 1 & * & * \\ 1 & 3 & * & * \end{bmatrix}$$

(b) The ELL Storage Representation

ptr=[0 3 5 9 11 13]
indices=[0 1 3 1 2 0 1 2 3 0 1 1 3]
data=[2 5 3 6 1 8 2 7 9 7 4 8 4]

(c) The CSR Storage Representation

row=[0 0 0 1 1 2 2 2 2 3 3 4 4]
indices=[0 1 3 1 2 0 1 2 3 0 1 1 3]
data=[2 5 3 6 1 8 2 7 9 7 4 8 4]

(d) The COO Storage Representation

Figure 2: Four sparse matrix storage formats. $*$ denotes the padding zero.

3. SPARSE MATRIX COMPRESSION FORMAT

SpMV is widely used for many scientific and engineering computing applications. We investigate four major formats for unstructured sparse matrix, *i.e.*, COO, CSR, ELL, and HYB (ELL/COO). An example matrix with the four formats is illustrated in Figure 2.

- *Coordinate Storage (COO)* is the most simple and intuitive storage scheme to store a sparse matrix in computer memory. It only saves the non-zero elements and the coordinates (row index and column index) of each non-zero element.

- *Compressed Sparse Row (CSR)* (also known as Compressed Row Storage, *i.e.*, CRS) is the most popular format used in SpMV operations due to its space-efficiency and fast access latency. The CSR format consists of three arrays: *ptr*, *indices*, and *data*. The integer array *ptr* stores row pointers to the offset of each row. The integer array *indices* stores the column indices of the non-zero elements. The array *data* stores the values of non-zero elements. Assume a matrix has the dimension of $M \times N$ and the number of non-zero elements is P, then the length of *ptr* is $M + 1$ and *indices* and *data* has the same length P.

- *ELLPACK (ELL)* (also known as ITPACK) stores a sparse matrix in two arrays: *indices* and *data*. The integer array *indices* stores the column indices of each non-zero element. The array *data* stores the non-zero values. Assume a matrix has the dimension of $M \times N$ and the maximum number of non-zero elements in a single row is Q, then the arrays *indices* and *data* have the same size of $M \times Q$ because those rows with fewer non-zero elements than Q are padded with zeros. If the number of non-zero elements on the rows is relatively small and average, ELL may have better performance than CSR. If the number of non-zeros of different rows varies significantly, the performance of ELL is worse than CSR.

- *Hybrid ELL/COO (HYB)* stores the majority of non-zero elements (on the left) in ELL format and the remaining non-zero elements (on the right) in COO format [3]. Specifically, the typical number of non-zero elements per row are stored in the ELL format and the remaining elements of exceptional rows are stored in the COO format. [3] computes a histogram of the row sizes and determines a threshold value. All non-zero elements at the columns on the left of the threshold value are in the ELL format and the rest non-zero elements are in the COO format.

4. AN EMPIRICAL CUDA PERFORMANCE MODEL FOR SPMV

4.1 Overview

Different from host memory, GPU has a very complex hierarchy of memories. The techniques of this paper are based on NVIDIA's GPUs, which contain global memory, constant memory, texture memory, shared memory, and registers. Each memory hierarchy has different size and different access control policy. Global memory, constant memory, and texture memory are relatively abundant but slow compared to shared memory and registers. They are visible for all threads. Shared memory is relatively small but fast. It can be accessed by threads within the same block. Registers are even smaller and work as the local memory for threads. The different sizes and access control policies of the hierarchical memories complicate the prediction of the execution time of SpMV CUDA kernel. In this paper, we assume that global memory, constant memory, and texture memory are abundant enough, and only the size of shared memory and register limit the numbers of threads, warps, and blocks during the execution of CUDA kernels.

Our model is specialized to estimate the execution time for commonly used SpMV CUDA kernels on NVIDIA's GPUs. In the paper, we focus on very large sparse matrices and the time we calculate in this paper is the statistical average time. The workflow of our model is as follows:

- Obtain hardware information of GPU according to its compute capability. (See Section 4.2.1.)

- Compute the number of active thread blocks for a streaming multiprocessor. (See Section 4.2.2.)

- Compute the total number of thread blocks for the target matrix. (See Section 4.2.3.)

- Compute the number of iterations. (See Section 4.2.4.)

- Choose several sample matrices and test their execution time. Then obtain the relationship between the number of non-zero elements and the execution time. (See Section 4.3.1.)

- Partition the target matrix into strips. For each matrix strip, appropriately enlarge the number of non-zero elements for some rows. Then estimate the execution time of each matrix strip. (See Section 4.3.1.)

- The total execution time of the target matrix is the sum of each strip's estimated execution time. (See Section 4.3.1.)

4.2 GPU memory limitations

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

Table 1: Symbols and functions used in our model

w_{pTB}	The number of warps per thread block
th_{pW}	The warp size
th_{pTB}	The block size
N_{TB}	The number of thread blocks
P	The number of active thread blocks per SM
M	The number of SMs
I	The total number of iterations
$t(i)$	The execution time of the ith iteration
$t_{SM\_PTB}(i)$	The execution time for one SM executes P thread blocks in the ith iteration
$\phi_{r\times c}()$	The execution time for a sample matrix with r rows and c columns
α, β	The arbitrary integers
c	The number of columns in a sample matrix
t_{Sample_CSR}	The execution time of sample matrix in CSR format

Table 2: Symbols of resource usage in .cubin file

Registers / Block	reg
Shared Memory / Block (bytes)	smem

4.2.1 Basic information of GPUs

The kernel specific information can be obtained by compiling and analyzing the kernel code. The block size (th_{pTB}) is required to be specified before invoking the SpMV kernel. We can obtain the value of *Registers / Thread* and *Shared Memory / Block* by using "*-cubin*" option when compiling the kernel. Their corresponding symbols in the "code" section of .cubin file are listed in Table 2. The value of *Registers / Block* can be obtained from multiplying the value of *Registers / Thread* by the value of block size (th_{pTB}).

The physical limitations of a specific GPU are determined by its compute capability. For NVIDIA's GeForce GTX 295, its compute capability is 1.3. The physical limitations of NVIDIA's GeForce GTX 295 are listed in Table 3.

4.2.2 The number of active thread blocks per SM

The number of active thread blocks is determined by three factors: the number of active warps, the number of registers and the size of shared memory.

The number of active thread blocks, denoted by TB_{SM}, is determined by the number of active warps.

Table 3: Physical limitations of GPUs with compute capability 1.3

Threads / Warp	32
Warps / Multiprocessor	32
Threads / Multiprocessor	1024
Thread Blocks / Multiprocessor	8
Total # of 32bit registers / Multiprocessor	16384
Shared Memory / Multiprocessor (bytes)	16384

Table 4: SpMV kernel granularity

Kernel	Granularity
CSR (vector)	One warp per row
ELL	One thread per row

$$TB_{SM} = \lfloor \frac{Warps/Multiprocessor}{w_{pTB}} \rfloor$$

The number of active thread blocks, denoted by TB_{regis}, is determined by the number of registers.

$$TB_{regis} = \lfloor \frac{Total\,\#\,of\ 32bit registers/Multiprocessor}{Registers/Block} \rfloor$$

The number of active thread blocks, denoted by TB_{shared_mem}, is determined by the size of shared memory.

$$TB_{shared\_mem} = \lfloor \frac{SharedMemory/Multiprocessor}{SharedMemory/Block} \rfloor$$

The number of active thread blocks, denoted by P, is the minimum of the above three values.

$$P = MIN\{TB_{SM}, TB_{regis}, TB_{shared\_mem}\}$$

4.2.3 The total number of thread blocks

We assume the target matrix has m rows. The SpMV kernel granularity is shown in Table 4.

For CSR vector kernel, one warp (32 threads) is responsible for computing the multiplication of one row of the matrix and the vector. Thus, the total number of required warps is equal to the number of rows of the matrix. Hence, the total number of thread blocks for CSR vector kernel is,

$$N_{TB} = \lceil \frac{m \times th_{pW}}{th_{pTB}} \rceil \qquad (CSR)$$

For ELL kernel, one thread corresponds to one row of the matrix. Thus, the total number of threads required by ELL kernel is equal to the number of rows in the matrix. Hence, the total number of thread blocks for ELL kernel is,

$$N_{TB} = \lceil \frac{m}{th_{pTB}} \rceil \qquad (ELL)$$

4.2.4 *The number of iterations*

Assume that the large-scale target matrix can not be processed completely by a GPU within one iteration. Instead, the matrix is processed by several iterations.

The total number of iterations is,

$$I = \lceil \frac{N_{TB}}{P \times M} \rceil$$

4.3 Estimation of execution time

As all the SMs execute concurrently, we assume that all the SMs finish each iteration at the same time approximately. We use one SM's execution time to represent the entire GPU's execution time. Hence, the total execution time of the target matrix, denoted by T, is equivalent to the sum of one SM's execution time in each iteration.

$$T = \sum_{i=1}^{I} t(i) \qquad (*)$$

In each iteration, each SM holds P thread blocks. Thus, the execution time is equivalent to the time that one SM executes P thread blocks.

$$t(i) = t_{SM\_PTB}(i)$$

where $t_{SM\_PTB}(i)$ is the time that one SM executes P thread blocks in the ith iteration.

To estimate the execution time of each iteration, we first evaluate some sample matrices which are similar to the matrix strip of the target matrix executed on GPUs in each iteration. Then we estimate the execution time of the matrix strip of the target matrix in each iteration according to those sample matrices. Finally, we use equation (*) to obtain the total execution time.

By analyzing the kernel code, we find that the time complexity of CSR vector kernel is proportional to the number of operations. Furthermore, the number of operations is also proportional to the number of non-zero elements of the matrix. Therefore, the execution time should be proportional to the number of non-zero elements of the matrix if the number of non-zero elements is large enough; otherwise the compute capability of GPU is not fully employed. For ELL kernel, the execution time is closely related to both the number of non-zero elements per row and the number of rows. According to [3], one thread corresponds to one non-zero element in COO kernel, hence, the execution time is proportional to the total number of non-zero elements of the matrix.

4.3.1 *Estimation for CSR vector kernel*

The principle of selecting sample matrices for CSR vector kernel is as follows:

- The sample matrices should utilize all the active thread blocks in all SMs, which ensures that the GPU's compute capability is fully utilized. For CSR vector kernel, one warp corresponds to one row, so the number of rows of the sample matrix is $(M \cdot P \cdot w_{pTB})$.

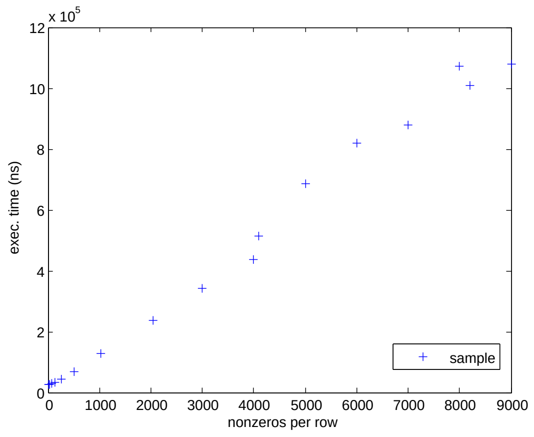

Figure 3: Number of non-zero elements per row vs execution time of sample matrices (CSR).

- To find out the threshold value of the linear relation between the execution time and the number of non-zero elements, the number of non-zero elements per row should be the same as a sample matrix.

To effectively avoid the affection of a long initialization delay and accurately measure the execution time of a sample matrix, we first invoke the SpMV kernel β times, where β is an arbitrary integer between 2 and 100. Then we invoke the SpMV kernel for another α times, where $\alpha = 1, 2, 3 \cdots$. Note that $\alpha < \beta$ is required.

The execution time of the sample matrix for CSR is

$$t_{Sample\_CSR} = \frac{\sum_{j=1}^{\beta} \phi_{(M \cdot P \cdot w_{pTB}) \times c} - \sum_{j=1}^{\alpha} \phi_{(M \cdot P \cdot w_{pTB}) \times c}}{\beta - \alpha}$$

where $w_{pTB} = \frac{th_{pTB}}{th_{pW}}$, and ϕ indicates the execution time of the sample matrix and its subscript indicates the dimension of the sample matrix.

The relation between the number of non-zero elements per row and the execution time of the sample matrices is shown in Figure 3. We find that there exists a threshold: when the number of non-zero elements in each row of the sample matrices is smaller than the threshold, the relationship is nearly conic curve (these nodes are crowded at the beginning of Figure 3); while the number is larger than the threshold, the relationship is nearly linear. Therefore, we use a conic curve and a linear line to fit these two parts respectively.

The linear relation between the total number of non-zero elements of the sample matrices and the corresponding execution time is illustrated in Figure 4. Here the total number of non-zeros in the sample matrix must be greater than a minimal value, which is calculated from the threshold obtained from Figure 3.

The steps to estimate the execution time are as follows:

- Partition: We partition our target matrix into matrix strips by row. The number of rows for each strip, $(M \cdot P \cdot w_{pTB})$, is the same as that of sample matrices.

- Extend and Count: We measure the execution time of each matrix strip to estimate execution time in each

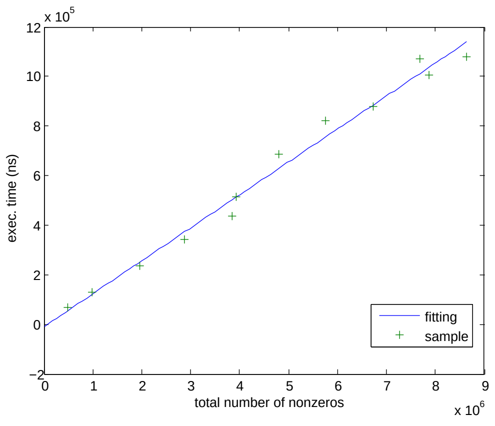

Figure 4: Fitting result for sample matrices (CSR).

iteration. For the matrix strip in each iteration, we extend the rows, where the number of non-zero elements is less than the threshold value, to the point which has the same execution time in the linear line in Figure 3 for easier calculation. Then we count the total number of non-zero elements of the extended matrix strip. Figure 4 reveals the proportional relation between the total number of non-zeros elements and the execution time. We can estimate the execution time according to the linear curve in Figure 4.

- Summarize: We finally estimate the total execution time by equation (*).

4.3.2 Estimation for ELLPACK kernel

The method to estimate the execution time for ELL kernel is similar to that of CSR vector kernel. By analyzing the ELL kernel code and the experiment results, we find that the execution time of ELL kernel is related to both the number of non-zero elements per row and the number of rows. Since the execution time depends on the two factors, we use two-dimension interpolation to estimate the execution time. The relation between the execution time and the two factors is illustrated in Figure 5. X, Y, and Z-coordinates are the logarithm of number of non-zero elements per row, the logarithm of number of rows, and the logarithm of execution time, respectively. After interpolation, we obtain Z, a two-dimension array, as the basis for our estimation.

To estimate the execution time of a target matrix in ELL format, we first scan the number of non-zero elements per row and find the row with the maximum value. Then we pad other rows to ensure that all the rows have the same number of non-zero elements. We assign the number of non-zero elements per row to x, *i.e.*, the maximum value obtained above, and assign the number of rows to y. To obtain $log(z)$ value, we calculate the logarithm of x and y, respectively, and find out the nearest point in Z array according to $log(x)$ and $log(y)$. Finally, we calculate the exponential of $log(z)$ to obtain estimate execution time z.

4.3.3 Estimation for HYB kernel

The HYB kernel is the combination of ELL and COO kernels. Since we have already discussed the model of ELL, here, we only discuss the COO model. The linear relation

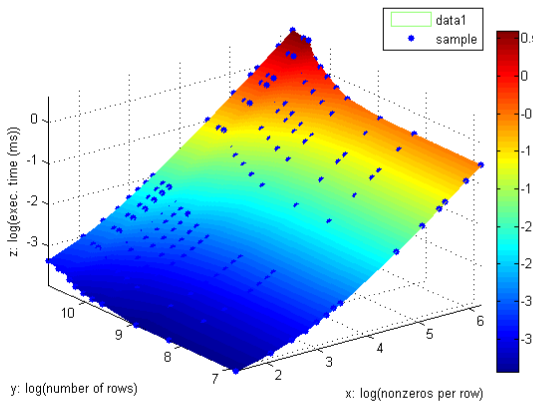

Figure 5: Interpolation result for sample matrices (ELL).

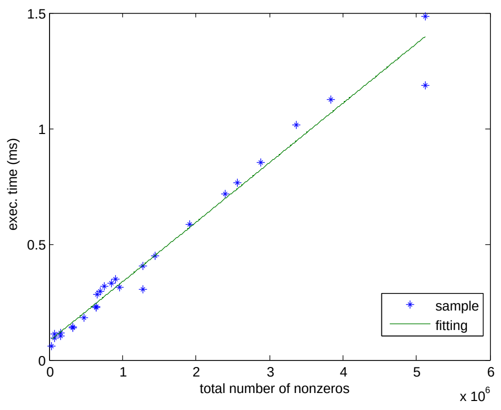

Figure 6: Fitting result for sample matrices (COO).

between the total number of non-zero elements and the execution time is demonstrated in Figure 6. So given a matrix, we can count its total number of non-zero elements, then estimate the execution time using this linear relation.

To estimate the execution time for a given matrix in HYB format, we first calculate the threshold value that separates ELL and COO, which is the same method used in [3]. Then the left part of the matrix is assigned to ELL model, the right part is assigned to COO model. We estimate the execution time of the entire matrix using the methods discussed above for ELL and COO. Finally, the total execution time of the matrix in HYB storage format is the sum of the two parts' execution time.

5. MODEL-DRIVEN SPARSE MATRIX PARTITIONING

The target matrix is sorted in descending order by the number of non-zero elements per row before partitioning. Our performance prediction model will benefit from the sorted target matrix.

To simplify the partitioning procedure, we first consider two popular space-saving matrix formats, *i.e.*, CSR and ELL. As we mentioned in Section 1, CSR storage format is usually good for the sparse matrices with large number of non-zero elements, and ELL storage format is usually good for sparse matrices with nearly equal and small number of non-zero elements per row.

For a sorted target matrix, we infer a separator to partition the matrix into two partitions by row. Each partition consists of several matrix strips. We adopt a uniform 960-row as the size of one matrix strip when partitioning the matrix. For NVIDIA's GeForce GTX 295, the granularity of the CSR vector kernel in Table 4 is one warp per row and 960 is the maximum number of warps that all the SMs can hold once, hence 960-row is a full-load for CSR vector kernel on NVIDIA's GeForce GTX 295. The performance model estimates the execution time of the CSR vector kernel by using 960-row as a basic unit of matrix strip.

The partitioning procedure works as follows: Initially, the separator starts at the bottom of the first matrix strip, then moves toward the end of target matrix step by step. The whole matrix strip (960 rows) is the step length which the separator moves. Every time when the separator moves to a new position, we estimate the execution time of the matrix partitions above and below the separator by using the methods in Sections 4.3.1 and 4.3.2, respectively. The total estimated execution time is equal to the sum of each partition's estimated execution time. The procedure continues till the separator moves to the bottom of the target matrix. Since the procedure to find the separator works in a top-down fashion, we exhaust all possible partitioning scenarios. The best partition will be chosen as the strategy for subsequent SpMV computation.

If the partition strategy cannot bring significant performance improvement, instead, our tool will report which storage format (*e.g.*, CSR, ELL, or HYB) can bring the best execution time based on the performance model. In our experiment of Section 7, the HYB storage format can bring some matrices significant benefit.

6. AUTO-TUNING SPMV CUDA KERNELS

Some CUDA parameters can affect the performance of SpMV notably. According to our experiment, there are three parameters playing decisive roles for SpMV kernels on GPUs, which are *Num_Threads*, *Block_Size* and *Warp_Size*. Tuning these parameters can remarkably affect the performance of SpMV CUDA kernels. Table 5 shows the tuning parameters for different SpMV kernels. For CSR vector kernel, all of three parameters affect the performance. For ELL kernel, only *Block_Size* affects the performance notably. For COO kernel, only *Num_Threads* matters. Since scientific computations usually contain many iterations, given a matrix with a specific storage format, the auto-tuning framework uses some loops to try all feasible combinations of these parameters in the first iteration, then chooses a combination with the best performance for the rest iterations.

Figure 7, Figure 8, and Figure 9 compare the performance of auto-tuning three parameters. Our experiment is conducted on the 14 matrices used by [3]. By auto-tuning parameters, the best performance of our CSR vector kernel is average 33% faster than NVIDIA's CSR kernel [3]; our HYB kernel is average 9% faster than NVIDIA's HYB kernel; our COO kernel is average 16% faster than NVIDIA's COO kernel. Auto-tuning parameters for ELL does not have much effect.

The details about how to choose the values of *Num_Threads*, *Block_Size*, and *Warp_Size* appeared in our preliminary work [12], where only CSR format is considered.

Table 5: Tuning parameters for SpMV kernels

Kernel	Tuning Parameters		
	Num_Threads	*Block_Size*	*Warp_Size*
CSR (vector)	×	×	×
ELL		×	
HYB	×	×	

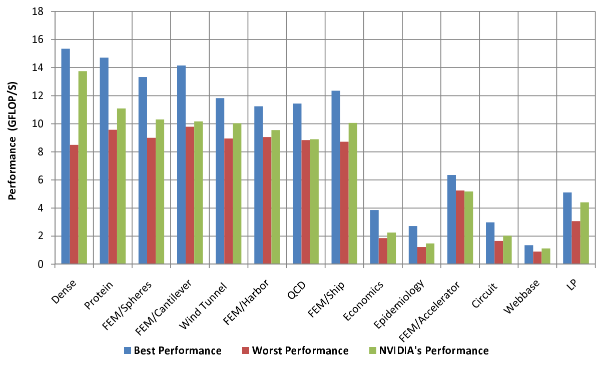

Figure 7: Performance evaluation on CSR kernel.

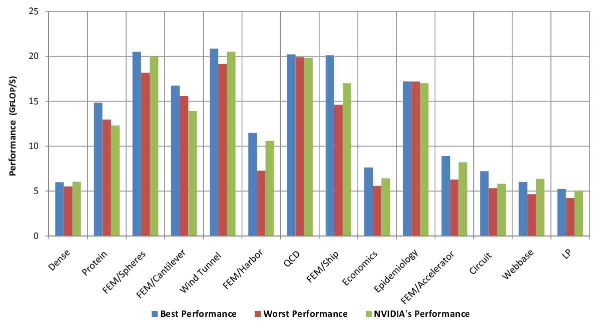

Figure 8: Performance evaluation on HYB kernel.

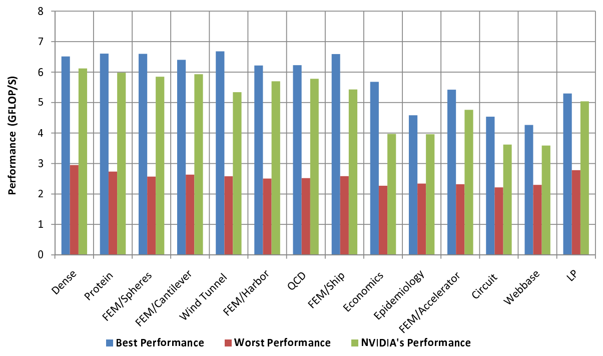

Figure 9: Performance evaluation on COO kernel.

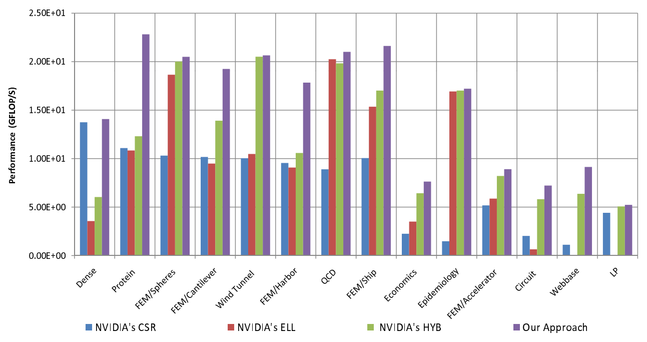

Figure 10: Performance evaluation of our approach compared to NVIDIA's CSR, ELL and HYB kernel.

7. PERFORMANCE EVALUATION

Similar to the experiment in Section 6, the partition approach is evaluated on the 14 matrices used by [2] by using NVIDIA's GeForce GTX 295. For SpMV, a randomly generated vector is used in our experiment, since the actual content of the vector will not affect the performance.

The comparison of our approach with NVIDIA's implementation (*i.e.*, SpMV CUDA kernels for CSR, ELL, and HYB [3]) on single-precision SpMV is shown in Figure 10. We did not consider the time to sort the target matrix and adjust the result. The ELL kernel cannot execute the matrices "Webase" and "LP" because their memory requirement is beyond our current GPU memory size. Our partition schema is usually the combination of CSR and ELL. The combination of CSR and HYB does not have benefit because the matrix to be processed has been sorted. In our generated partitions, the majority of non-zero elements are in CSR format, and the rows with few non-zero elements are in ELL. Some matrices may not need any partition. For example, our approach reports that "HYB" is already the best format for the matrices "FEM/Spheres", "Economics", "Epidemiology", "FEM/Accelerator", "Circuit" and "LP". In conclusion, compared to NVIDIA's CSR vector kernel, our approach has 222% performance improvement on the average on the 14 matrices. Compared to NVIDIA's ELL kernel, our approach has 197% performance improvement on the average. Compared to NVIDIA's HYB kernel, our approach has 33% performance improvement on the average.

8. CONCLUSION AND FUTURE WORK

In this paper, we have presented a framework that computes SpMV efficiently. The framework is based on an empirical GPU model that estimates the execution time for a given matrix in various formats. We design and implement a model-driven partitioning approach to predict how to partition the target sparse matrix and transform each partition into appropriate storage format based on different storage characteristics. Our partition explores different combinations of formats to find an optimal partition for large-scale matrices using the performance prediction model. The designing principle is based on the fact that the different storage formats of sparse matrix can significantly affect the performance of SpMV. The auto-tuning tool can automatically adjust CUDA-specific parameters to optimize performance. We integrate model-driven partitioning and auto-tuning to maximize the benefit of performance. Our experiment shows that this approach has substantial performance improvement compared to the existing SpMV CUDA kernels [3].

In the future work, we will refine the performance model to predict the matrix partitioning more accurately. We will improve and extend the current framework to include more SpMV kernels. In addition, we will apply our approach to more areas of scientific and engineering applications.

9. REFERENCES

[1] J. Kurzak, W. Alvaro, and J. Dongarra. Optimizing matrix multiplication for a short-vector simd architecture-cell processor. *Parallel Comput.*, 35(3):138–150, 2009.

[2] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *In Proc. 2007 ACM/IEEE Conference on Supercomputing*, 2007.

[3] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009.

[4] M. M. Baskaran and R. Bordawekar. Optimizing sparse matrix-vector multiplication on gpus using compile-time and run-time strategies. Technical report, Research Report RC24704, IBM TJ Watson Research Center, december 2008.

[5] *NVIDIA CUDA (Compute Unified Device Architecture): Programming Guide, Version 2.1*, December 2008.

[6] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. C. Whaley R. Vuduc, and K. Yelick. Self-adapting linear algebra algorithms and software. *Proceeding of IEEE*, 93(2):293–312, 2005.

[7] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *Int. J. High Perform. Comput. Appl.*, 18(1):135–158, 2004.

[8] J. Bolz, I. Farmer, E. Grinspun, and P. Schroder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, 2003.

[9] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 115–126, New York, NY, USA, 2010.

[10] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on cuda. Technical report, Nvidia Technical Report NVR-2008-004, 2008.

[11] A. Nukada and S. Matsuoka. Auto-tuning 3-d fft library for cuda gpus. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–10, New York, NY, USA, 2009.

[12] Ping Guo and Liqiang Wang. Auto-tuning cuda parameters for sparse matrix-vector multiplication on gpus. *Computational and Information Sciences, International Conference on*, 0:1154–1157, 2010.