Auto-Tuning CUDA Parameters for Sparse Matrix-Vector Multiplication on GPUs

Ping Guo
Department of Computer Science
University of Wyoming, USA
pguo@uwyo.edu

Liqiang Wang
Department of Computer Science
University of Wyoming, USA
lwang7@uwyo.edu

***Abstract*—Graphics Processing Unit (GPU) has become an attractive coprocessor for scientific computing due to its massive processing capability. The sparse matrix-vector multiplication (SpMV) is a critical operation in a wide variety of scientific and engineering applications, such as sparse linear algebra and image processing. This paper presents an auto-tuning framework that can automatically compute and select CUDA parameters for SpMV to obtain the optimal performance on specific GPUs. The framework is evaluated on two NVIDIA GPU platforms: GeForce 9500 GTX and GeForce GTX 295.**

Keywords- GPU; CUDA; sparse matrix-vector multiplication; performance

I. INTRODUCTION

The sparse matrix-vector multiplication (SpMV) is a critical operation in a wide variety of scientific and engineering applications. Optimizing SpMV computation has always been a challenge because SpMV computation is irregular and requires many indirect and irregular memory accesses [8]. In addition, the fine-grained parallelism is hard to explore [10]. Bell and Garland demonstrated that the SpMV computation can be successfully mapped to the fine-grained parallel architecture of GPUs [4].

Graphics Processing Unit (GPU) has become an attractive coprocessor for scientific computing due to its massive processing capability. GPU is especially well-suited to address problems that can be expressed as data-parallel computations [6]. GPU has been applied to SpMV and can significantly accelerate the performance by executing many dot products in parallel [9]. Launching a CUDA kernel will create a grid of threads. All threads in a grid execute the same kernel function. These threads are organized into a two-level hierarchy. A grid is organized as a two dimensional array of blocks at the top level and all blocks are organized into a three dimensional array of threads at the underline level [9]. A warp is a group of threads executed physically in parallel. Typically, 16 threads (half warp) are executed simultaneously.

This paper makes the following contributions: (1) We investigate how CUDA parameters (Num_Threads, Block_Size and Warp_Size) affect the performance of SpMV kernels. (2) We design and implement an auto-tuning framework that can automatically adjust and choose CUDA parameters for SpMV to obtain the optimal performance on specific GPUs.

This work was supported in part by TeraGrid Pathways Fellowship, NSF under Grant 0930040 and 0941735, and the Graduate Assistantship of the School of Energy Resources at the University of Wyoming.

II. RELATED WORKS

Bolz et al. first apply GPU computing to SpMV [7]. Bell and Garland implement SpMV kernels in CUDA for several sparse matrix formats, including DIA, ELL, COO, CSR, and Hybrid (ELL/COO) [5]. Our SpMV kernel is based on their implementation.

Baskaran and Bordawekar propose a framework for optimizing SpMV on GPUs [11]. Their framework consists of three components: two modules and one runtime inspector. One module performs compile-time optimization. The runtime inspector analyzes the sparse matrix structure. And the other module executes the optimized kernel on GPU device.

Nukada and Matsuoka present an auto-tuning framework that chooses the optimal number of threads for the CUDA-based 3-D FFT library automatically [2]. Demmel and Dongarra have explored AEOS approach to automate the kernel optimization [1]. They present two software systems, ATLAS and BeBOP, for dense and sparse linear algebra kernels, respectively. Their optimized kernels achieve a considerable speedup.

The rest of this paper is organized as follows: Section III reviews a widely-used sparse matrix format. Section IV presents our auto-tuning framework. Section V presents the performance evaluations. Section VI summaries the conclusions and the future work.

III. SPARSE MATRIX FORMAT

Sparse matrix is usually stored in a compact format, *i.e.*, only non-zero elements are preserved. Fig. 1 shows an example for a widely-used sparse matrix format called CSR (*Compressed Sparse Row*) or CRS (*Compressed Row Storage*). CSR format contains three arrays, i.e., *ptr*, *indices*, *and data*, which store row pointers to the offset of each row, the column indices, and the values of non-zero entries, respectively.

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

Sample Sparse Matrix

CSR Format:

ptr=[0 4 7 9 14 17]

indices=[0 1 3 4 0 2 4 1 3 0 1 2 3 4 1 2 3]

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

Figure 1. CSR sparse matrix format.

IV. SPMV AUTO-TUNING FRAMEWORK

What CUDA parameters do we need to auto-tune for SpMV computation on GPUs? Before designing our auto-tuning framework, we need to address this question first. Studying the real code of CSR vector kernel for SpMV, we get two important equations (1) and (2) as follows:

$$\text{NUM\_BLOCKS} = \text{NUM\_THREADS} / \text{BLOCK\_SIZE} \quad (1)$$

$$\text{NUM\_WARPS}=(\text{BLOCK\_SIZE} / \text{WARP\_SIZE}) * \text{gridDim.x} \quad (2)$$

Where NUM_BLOCKS is the first special parameter which specifies the number of blocks in the grid and NUM_WARPS is the total number of active warps. More specifically, from equations (1) and (2), we know that three CUDA parameters (NUM_THREADS, BLOCK_SIZE and WARP_SIZE) affect the performance of SpMV kernels on GPUs remarkably. Tuning these parameters can significantly affect the performance of SpMV computation.

The workflow of our auto-tuning framework is shown in Fig. 2. During the execution, our auto-tuning framework reads the properties of the specific GPU device first, then enumerates the feasible values for NUM_THREADS and BLOCK_SIZE according to the properties of the specific GPU device and selects the value of WARP_SIZE (16 or 32) according to the specific matrix, then combines the different parameters to invoke the corresponding CSR vector kernel for SpMV computation. Since scientific computations usually contain many iterations of SpMV for the same matrix, after the first iteration, our auto-tuning framework can automatically choose a combination of NUM_THREADS, BLOCK_SIZE, and WARP_SIZE with the optimal performance. The rest iterations will utilize such a combination of parameters to obtain the optimal performance.

A. Selecting NUM_THREADS

The value of MAX_THREADS, which is the maximum number of threads, is a pre-determined value depending on the specific GPUs. It is determined by the following formula:

$$\text{MAX\_THREADS} = N * T_{SM}$$

Where N is the number of Streming Multiprocesssors (SMs) in a specific GPU. T_{SM} is the maximum number of threads that can be assigned to each SM. The specific GPUs with different compute capability have different values of N and T_{SM}. For GeForce 9500 GTX, the value of N is 4 and the value of T_{SM} is 768. For GeForce GTX 295, the value of N is 30 and the value of T_{SM} is 1024. The range of NUM_THREADS for our auto-tuning framework is [MAX_THREADS /2, MAX_THREADS] since the peak performance usually appears between the half load and the full load of MAX_THREADS (See Figure 3 and Figure 4). In addition, we set T_{SM} as the step length for tuning NUM_THREADS.

B. Selecting BLOCK_SIZE

The value of BLOCK_SIZE depends on the specific GPUs. For a specific GPU, since T_{SM} is constant, the value of BLOCK_SIZE is inversely proportional to the number of blocks which can co-exist on the same SM. In addition, the BLOCK_SIZE values that fail to satisfy the following two criterions will be excluded:

- The value of BLOCK_SIZE can not exceed 512 .
- The value of BLOCK_SIZE must be the integral multiple of the value of WARP_SIZE.

C. Selecting WARP_SIZE

The CSR kernel contains two implementations: vector-based and scalar-based. In CSR vector kernel, each matrix row is processed by a whole warp (32 GPU threads), while the CSR scalar kernel processes each row by one thread. The CSR vector kernel accesses indices and data contiguously, therefore outperforms the CSR scalar kernel [4]. The performance of the vector kernel is sensitive to the number of non-zero elements in matrix rows. If the number of non-zero elements in most matrix rows is greater than the warp size, the CSR vector kernel can usually obtain high performance. In Nvidia's implementation [4], they define the warp size as 32. However, one important point neglected by Nvidia's implementation is: if most rows of the sparse matrix have a small number of non-zero elements (*e.g.* less than 32), then many threads in the warp are idle. In this situation, the performance of the vector kernel will drop down significantly. By tuning the warp size (32 threads) to half warp size (16 threads) to utilize the resource effectively, our approach can get significant performance improvement in contrast to Nvidia's implementation (See Figure 6).

In our auto-tuning tool, CSR kernels may be invoked with different values of NUM_THREADS, BLOCK_SIZE, and WARP_SIZE. In Nvidia's implementation [4], these parameters are defined as constant. In our implementation, we predefine several variants of CSR kernels according to different combinations of these parameters since changing the values of parameters dynamically cannot be supported by the system.

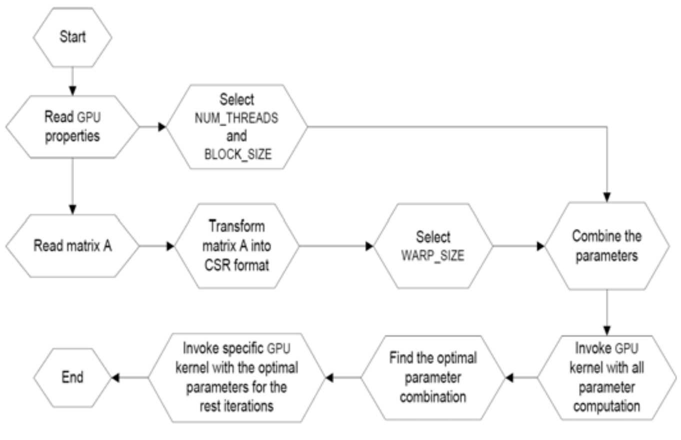

Start
Read GPU properties
Select NUM_THREADS and BLOCK_SIZE
Read matrix A
Transform matrix A into CSR format
Select WARP_SIZE
Combine the parameters
Invoke GPU kernel with all parameter computation
Find the optimal parameter combination
Invoke specific GPU kernel with the optimal parameters for the rest iterations
End

Figure 2. The workflow of our auto-tuning framework.

V. PERFORMANCE EVALUATION

Our experiments are evaluated on 14 unstructured sparse matrices [3], as shown in Table 1, by using two NVIDIA GPUs: GeForce 9500 GTX and GeForce GTX 295, where the compute capability for 9500 GTX is v1.1 and the compute capability for GTX 295 is v1.3. For each sparse matrix, we randomly generate an input vector for SpMV whose values do not affect the performance. To measure the performance of SpMV for each matrix, we execute the SpMV kernel for 500 times, and take the averaged performance. Note that, in our experiment, the GPU's warm up time is excluded.

A. Test NUM_THREADS

Fig. 3 and Fig. 4 show how the performance varies with NUM_THREADS by assuming BLOCK_SIZE and WARP_SIZE are constant. Let BLOCK_SIZE=128 and WARP_SIZE=16. The performance increases sharply before the half load of allowed MAX_THREADS. While, after the half load of allowed MAX_THREADS, the performance change slightly. For GeForce GTX 295, the peak performance appears at point "15360". For GeForce 9500 GTX, the peak performance appears at point "2304".

B. Test BLOCK_SIZE

For GeForce GTX 295, 128, 256, 512 are the only three feasible BLOCK_SIZE values. Assuming that NUM_THREADS and WARP_SIZE are constant, Fig. 5 shows how the performance varies with them. Let NUM_THREADS=16384 and WARP_SIZE=32. From Fig. 5, we found that BLOCK_SIZE=128 has the best performance compared to the others. However, for GeForce GTX 9500, the best performance comes from BLOCK_SIZE=192 instead of BLOCK_SIZE=128. The reason is: since the resource of GeForce GTX 9500 is insufficient to satisfy the needs of the simultaneous execution of 8 blocks, the CUDA runtime has to reduce NUM_BLOCKS per SM automatically to satisfy that the resource usage is under the limit. Thus, the value of BLOCK_SIZE has to increase correspondingly since we have assumed that NUM_THREADS is constant.

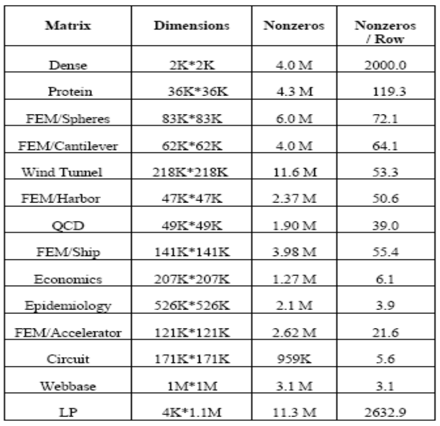
C. Test WARP_SIZE

For GeForce GTX 295, Fig. 6 shows how the performance varies with WARP_SIZE=16 and WARP_SIZE=32, respectively by assuming that NUM_THREADS and BLOCK_SIZE are constant. Let NUM _THREADS=15360 and BLOCK_SIZE=128. By comparing the performance, we group these 14 matrices into two groups. Dense, Protein, FEM/Spheres, FEM/Cantilever, Wind Tunnel, FEM/Harbor, FEM/Ship and LP are in group 1. QCD, Economics, Epidemiology, FEM/Accelerator, Circuit and Webbase are in group 2. For all group 1 members, the performance of WARP_SIZE=32 outperforms the performance of WARP_SIZE=16 since the number of non-zero elements in most matrix rows is greater than the warp size. While, for all group 2 members, the performance of WARP_SIZE=16 outperforms the performance of WARP_SIZE=32 since the number of non-zero elements in most matrix rows is small (*e.g.* less than 45).

D. Overall Performance Evaluations

Sections A, B and C have shown how the performance varies with NUM_THREADS, BLOCK_SIZE and WARP_SIZE, respectively, by assuming that the other two parameters are constant. Fig. 7 and Fig. 8 illustrate how the performance varies if when tuning all three parameters. For GeForce 9500 GTX, compared to Nvidia's implementation, our auto-tuning framework has 237% performance improvement on the average, and the median improvement is 278%. For GeForce GTX 295, compared to Nvidia's implementation, our auto-tuning framework has 33% performance improvement on the average, and the median improvement is 25.6%. The performance of Nvidia's implementation in Fig. 8 is much lower than our auto-tuned performance since GeForce 9500 GTX cannot launch as many as 30*1024 threads requested by Nvidia's implementation. BLOCK_SIZE=128, as defined in Nvidia's implementation, cannot always guarantee to obtain the best performance compared to other values of BLOCK_SIZE because of the resource usage in specific GPU device (*e.g.* 9500 GTX).

Table 1. Unstructured matrices used for our evaluations.



Matrix	Dimensions	Nonzeros	Nonzeros / Row
Dense	2K*2K	4.0 M	2000.0
Protein	36K*36K	4.3 M	119.3
FEM/Spheres	83K*83K	6.0 M	72.1
FEM/Cantilever	62K*62K	4.0 M	64.1
Wind Tunnel	218K*218K	11.6 M	53.3
FEM/Harbor	47K*47K	2.37 M	50.6
QCD	49K*49K	1.90 M	39.0
FEM/Ship	141K*141K	3.98 M	55.4
Economics	207K*207K	1.27 M	6.1
Epidemiology	526K*526K	2.1 M	3.9
FEM/Accelerator	121K*121K	2.62 M	21.6
Circuit	171K*171K	959K	5.6
Webbase	1M*1M	3.1 M	3.1
LP	4K*1.1M	11.3 M	2632.9

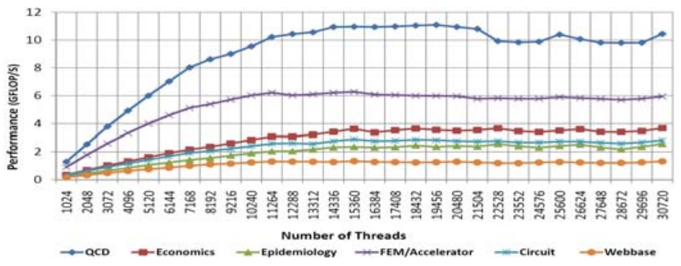


Figure 3. Test NUM_THREADS on GTX 295.

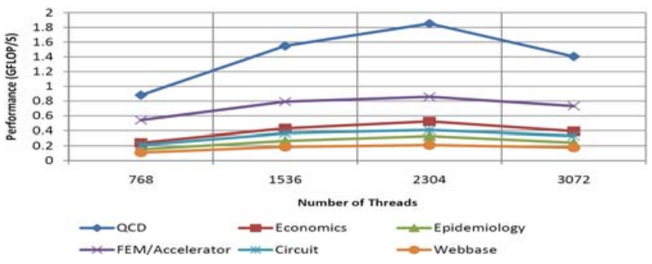


Figure 4. Test NUM_THREADS on 9500 GTX.

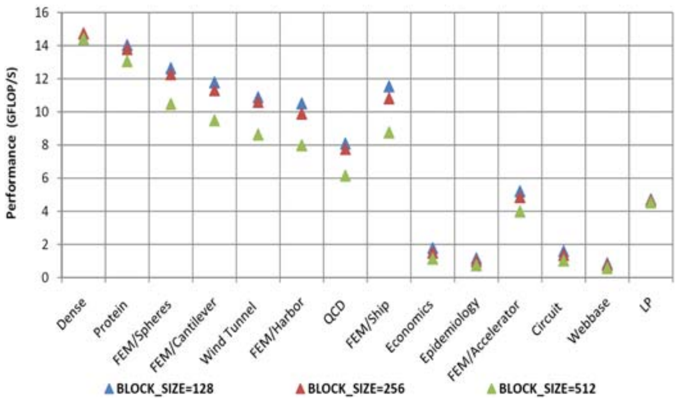


Figure 5. Test BLOCK_SIZE on GTX 295.

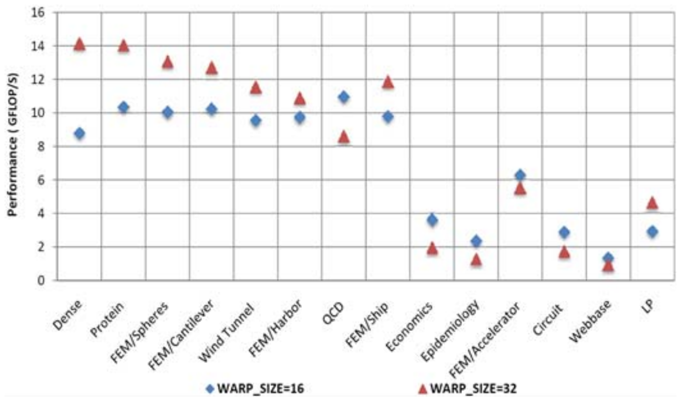


Figure 6. Test WARP_SIZE on GTX 295.

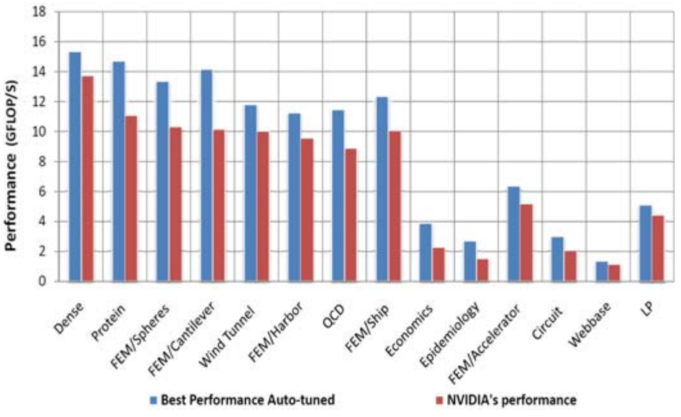


Figure 7. Overall performance evaluation on GTX 295.

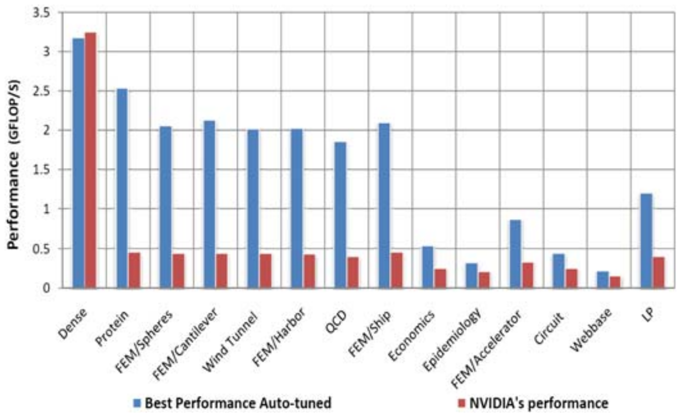


Figure 8. Overall performance evaluation on 9500 GTX.

VI. CONCLUSION AND FUTUREWORK

We design an auto-tuning framework that can automatically compute and select CUDA parameters to obtain the optimal performance on specific GPUs. Our performance evaluations are conducted on two NVIDIA GPU platforms: GeForce 9500 GTX and GeForce GTX 295. Compared to Nvidia's original implementation, the experimental results show that our auto-tuning framework significantly improves the performance of SpMV computation. In the future work, we will explore more optimization methods, such as overlapping the executions of CPU and GPU, to obtain better performance for SpMV on GPUs. We will also extend our auto-tuning framework to handle other CUDA kernels.

REFERENCES

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

[2] 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.

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

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

[5] 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.

[6] NVIDIA CUDA (Compute Unified Device Architecture): Programming Guide, Version 2.0, and June 2008.

[7] 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.

[8] 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.

[9] David B. Kirk and Wen mei W. Hwu. Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann, Burlington, MA, USA, 2010.

[10] 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.

[11] 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, 2008.