Chapter 71
Parallel Optimization for Sparse Matrix–Vector on GPU

Mengjia Yin, Xianbin Xu, Hua Chen, Shuibing He and Jing Hu

Abstract Graphics processing units (GPUs) have been used in the general-purpose computation field. Sparse matrix–vector multiplication (SpMV) algorithm is one of the most important scientific computing kernel algorithms. In this paper, we discuss implementing optimizing sparse matrix–vector multiplication on GPUs using CUDA programming model. We used methods and strategy which including mapping thread, merging access, reusing data, and avoiding the branch. The experimental results show that the optimizations strategy to improve SpMV performance.

Keywords Sparse matrix–vector multiplication (SpMV) • Compute unified device architecture (CUDA) • Graphics processing unit (GPU) • Performance optimizations strategy (POS)

M. Yin (✉) · X. Xu · S. He · J. Hu
School of Computer, Wuhan University, Wuhan 430074, People's Republic of China
e-mail: hbyinmj@163.com

X. Xu
e-mail: xbxu@whu.edu.cn

S. He
e-mail: hesbingxq@163.com

J. Hu
e-mail: hujing031115@126.com

M. Yin
School of Computer and Information Science, Hubei Engineering University, XiaoGan, People's Republic of China

X. Xu
School of Computer Science, Wuhan Donghu University, Wuhan, People's Republic of China

H. Chen
Agricultural Bank of China Software Development Center Guangzhou Sub-center, Yingbin Road, Dashi section 519 Agricultural Bank of China Science and Technology Park, Panyu, Guangzhou, Guangdong, People's Republic of China
e-mail: chenhuakfgd@abchina.com

S. He
Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing 10000, People's Republic of China

Z. Zhong (ed.), *Proceedings of the International Conference on Information Engineering and Applications (IEA) 2012*, Lecture Notes in Electrical Engineering 217,
DOI: 10.1007/978-1-4471-4850-0_71,

71.1 Introduction

With the rapid growth of computing complexity and data, general CPU computing power has failed to meet its needs. The rapid development of GPU is greatly exceed the speed of Moore's Law, the development of computing power, memory bandwidth is far exceed CPU. As modern GPUs have become increasingly powerful, inexpensive and relatively easier to program through high level API functions, they are increasingly being used for non-graphic or general-purpose applications (called GPGPU computing).

Sparse matrix–vector multiplication (SpMV) operation is widely used in solving large-scale linear system and solving matrix eigenvalues problems [1], especially in iterative method, it is key steps of influence the arithmetic performance. SpMV is typical of memory bottleneck operations, namely computing/memory is low, ALU seriously unsaturated, it is difficult to achieve the throughput of high floating point operations. SpMV has the nature of parallelism, and use of modern multi-processor platform to improve the performance is one of the feasible direction.

According to the deficiency of the traditional parallel strategies, we present the more efficient performance optimization strategy: mapping thread, merging access, reusing data, avoiding branch, optimization thread block. The experimental results show that these strategies can realize SpMV efficient parallel computing, and effectively improve the performance of the system.

The rest of this paper is organized as follows: Sect. 71.2 introduces related work, SpMV parallel mode is detailed in Sect. 71.3, performance optimization strategy is present in Sects. 71.4, 71.5 contains our results and evaluation, conclusion and future work is shown in Sect. 71.6.

71.2 Related Work

The bottleneck problems of memory are those algorithm that each floating point operation needs to multiple access memory, SpMV is a kind of this algorithm [2]. In the past 20 years, there have been a lot of work for the optimization of the SpMV algorithm [3, 4], from the point of view of the memory; optimization is mainly to improve the computational performance [5], which most of the optimization work is focused on generalization system structure similar to CPU [6]. But the optimization technology cannot be directly used in GPU system structure. GPU is massively parallel system; it has multi-stage storage system structure. In order to play the advantages of GPU memory high bandwidth, we need to accord with the characteristics to design different optimization strategies.

In reference [7], Nathan Bell and Michael Garland provide data structures and algorithms for SpMV that are efficiently implemented on CUDA platform for the fine-grained parallel architecture of the GPU. They emphasize memory

bandwidth efficiency and compact storage formats when given the memory-bound nature of SpMV. They also develop methods to exploit several common forms of matrix structure while offering alternatives which accommodate greater irregularity.

In reference [8], with indirect and irregular memory accesses resulting in more memory access per floating point operation, Baskraran proposed optimizations to effectively develop a high-performance SpMV kernel on NVIDIA GPUs. The optimizations include exploiting synchronization-free parallelism, optimized thread mapping based on the affinity toward optimal memory access pattern, optimized off-chip memory access to tolerate the high access latency, exploiting data reuse.

Based on the above, this paper emphasizes its optimization strategy in the process of SpMV algorithm on GPU, the optimization strategy is aimed at the system structure of the GPU, and considers the GPU complex storage management and the mapping optimization between threads.

71.3 Parallel Acceleration for SpMV Model Based on GPU

71.3.1 GPU Programming with CUDA

CUDA is a parallel computing architecture developed by NVIDIA Corporation [3] and allows writing and running general-purpose applications on the NVIDIA GPU's. CUDA uses threads for parallel execution, and GPU allows 1,000 of threads for parallel execution at the same time.

On the GPU, there is a hierarchy of memory architecture to program on it; we present the memories in our implementation: Registers, Shared Memory, Global Memory, Constant Memory, and Texture Memory. In the memory architecture, the fastest memories are the shared memories and registers. The other memories are all located on the GPUs main RAM. The constant memory is favorable when multiple processor cores load the same value from cache. Texture cache has higher latency, but it has a better acceleration ratio for accessing large amount of data and non-aligned accessing. The memory architecture of GPU is described in Fig. 71.1. To gain better performance, we must manage the shared memory, registers, and global memory usage.

The CUDA programming model greatly simplified the difficulty of using the GPU for general-purpose computing, but compared to isomorphic to the systems only included CPU; it is more complicated to program in the heterogeneous system based on the CPU–GPU; the program's performance optimization is even more difficult. Generally affect the performance of CUDA program includes the main three factors: memory access latency, load balance, and global synchronous spending [9]. In different computing platforms,the causes and the corresponding optimization methods of these factors are not same.

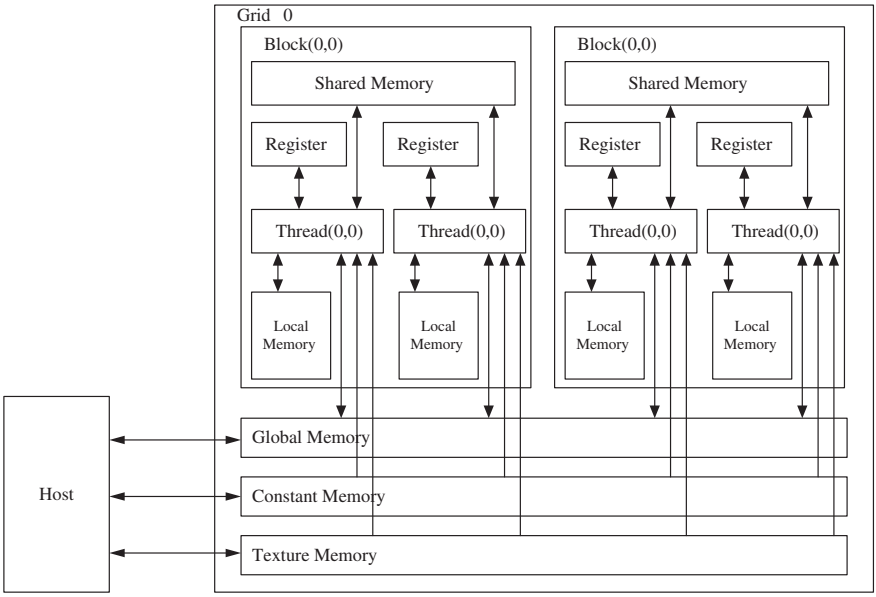


Fig. 71.1 Memory architecture of CUDA

71.3.2 Sparse Matrix Format

In scientific computing, SpMV has been proven to be a special important of numerical algorithm [10], it has the characteristics of high intensity calculation, high parallel degree, and simply control, so matrix calculation is very suitable for GPU for parallel computing. How to play the powerful computing ability of GPU in sparse matrix–vector algorithm is need to deal with.

Sparse matrix has several storage formats such as DIA, ELL, COO, CSR, HYB, and PKT. These storage formats are described detailed in [11]. Each format is different in storage requirements, calculation characteristics, access and operation of the matrix element method. Different storage formats are determined by the sparse matrix mode, that is, the distribution of non-zero elements in the coefficient matrix. In this paper, we base our approach on the vector SpMV kernel for CSR sparse matrix and discuss optimization to adapt CSR storage format to suit the GPU architecture.

CSR format is the more popular storage format [11, 12]; it is a line of compressed format which alter storing two-dimensional array of sparse matrix into 3 one-dimensional arrays: A,Col_Idx,Row_Ptr. Scan follows the line width for the sparse matrix, and will stored the zero elements in array A; An array of Col_Idx stored column index of non-zero elements in array A corresponding location in the original matrix; An array of Row_Ptr stored an index of every row in the first non-zero elements in an array of A Col_Idx in primitive sparse matrix. For M * N matrix, the length of Row_Ptr array is M + 1, the offset of ith row stored in

Row_Ptr [I], the last Row_Ptr [M] in sparse matrix stored the total number of non-zero elements. We give an example of 5 * 4 sparse matrixes, as shown in Figs. 71.2, 71.3, how to use the CSR storage formats to show the original sparse matrix.

71.3.3 Parallel Computing for SpMV Model Based on GPU

The serial algorithm based on the Compression row storage (CSR) format as follows: this algorithm is realized its parallelization in multiple processors, parallelism is realized in outer loop, so the single processors is responsible for computing the row of matrix.
Serial algorithm based on the CSR format:

{for i = 0 to rows
{y(i) = 0;
for j = Row_Ptr(i) to (Row_Ptr(i + 1)-1)
{y(i) = y(i) + A(j-1)*x(Col_Idx(j-1));}}}

N. Bell and M. Garland proposed two CSR format of SpMV kernel: Scalar-CSR kernel and Vector-CSR kernel [8]. The realization of Scalar-CSR kernel in CUDA is simple and direct: use a thread is responsible for computing one line element in sparse matrix. The performance of the Scalar-CSR is affected by various factors, non-zero elements in each row and its column index is stored, but cannot be accessed at the same time. Vector-CSR overcomes this shortcoming of the Scalar-CSR: using a Warp thread responsible for computing one line element in sparse matrix. The kernel of Vector-CSR continuously accesses to the index and data, so overcome the problem of inefficiency in the Scalar method. In this paper, the realization of the CSR format SpMV kernel is also a reference to Vector SpMV kernel.

Here, the parallelism of the method is as follows: use a Warp thread to responsible for computing non-zero elements in sparse matrix, do not need filling zero elements to

Fig. 71.2 5 * 4 sparse matrix

0	2	0	4	0
1	2	3	0	0
0	1	0	0	0
0	0	0	1	1

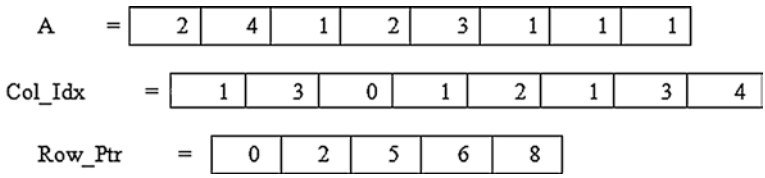


Fig. 71.3 CSR storage format of sparse matrix

align, intermediate results put on sharing memory, and then accumulate the intermediate results through reduction summation, finally through thread 0 to get the final results. Vector SpMV kernel for the CSR sparse matrix:

```
{_global_void spmv_csr_Kernel(const int num_rows,const int * Row_Ptr, const
   int *Col_Idx, const Float * A, const Float * x, Float* y)
{__shared__ Float vals[];
int thread_id = blockDim.x*blockIdx.x + threadIdx.x;
int Warp_id = thread_id/32;
int lane = thread_id & (32-1);
int row = Warp_id;
 if(row < num_rows)
{int row_start = Row_Ptr[row], row_end = Row_Ptr[row + 1];
       vals [threadIdx.x] = 0;
       for(int j = row_start +lane; j < row_end; j +=32)
         {vals [threadIdx.x] += data[j] * x[indices[j]];}
        if (lane < 16) vals [threadIdx.x] += vals [threadIdx.x + 16];
if (lane < 8) vals [threadIdx.x] += sdata[threadIdx.x + 8];
if (lane < 4) vals [threadIdx.x] += vals [threadIdx.x + 4];
if (lane < 2) vals [threadIdx.x] += vals [threadIdx.x + 2];
if (lane < 1) vals [threadIdx.x] += vals [threadIdx.x + 1];
if (lane == 0) y[row] += vals [threadIdx.x];}}
```

71.4 Optimizations for SpMV Model Based on GPU

71.4.1 Optimization Method

According to circle, CSR-Vector with a Warp complete computing one line of elements, and in the process of calculation, in order to get the results of output vector, we reduction summation in sharing storage. However, if the number of non-zero elements in the row is less than 32, the performance of the CSR-Vector will drop. When the number of non-zero elements is bigger, often more than 32, it can be get the best computing performance in non-zero elements of matrix each row contained relatively large, often greater than 32. According to the various insufficient of CSR-Vector kernel, based on the proposed all sorts of optimization strategy, here a few of the optimization of the CSR format SpMV: threads mapping, merging access, data reuse, avoiding divergence, optimization thread block.

71.4.2 Improve CSR SpMV Optimization Algorithm

The achievements of kernel first need several threads that responsible for computing an element of the output vector. When the number of non-zero elements that

one line contains is less, or is not multiple of 16, this strategy will cause wasting the thread to calculate the resources. In this paper, we proposed a new calculation method: array A in CSR sparse matrix is divided into certain length fragments, the length of the fragment is an integer multiple of the number of threads in the thread blocks, a thread block calculates element of an array fragment. The intermediate results stored in shared memory, and finally through accumulated calculation the intermediate results to complete the output element y [13]. This method is equally distributed computing tasks, and can effectively improve the operation efficiency.

As there is difference in the number of non-zero elements of sparse matrix each row, CSR SpMV kernel is difficult to the average computing tasks are assigned to each thread and cause computing resources free. To solve this problem, this paper takes the method that each thread block calculates the 1,024 non-zero elements; the last fragment is filled with 0, as shown in Fig. 71.4.

On the base of the CSR data structure, we added a int2 type array Bound, the length of array Bound is the number of array fragments that are divided (the number of thread block). Bound [i] corresponds to thread blocks that index is i, the members of x stored row number where the first element corresponding thread block, the members of y stored row number where tail element. This paper only generates Bound array through a simple judgment on each element value in Row_Ptr, as shown in Fig. 71.5.

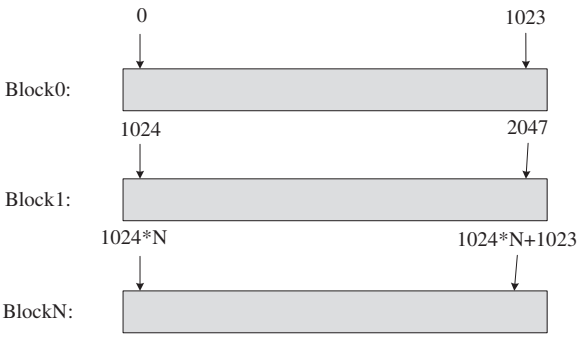
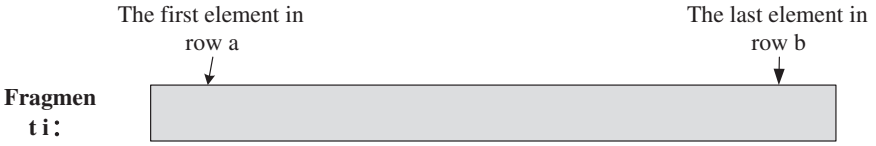


Fig. 71.4 Each thread block calculates the 1,024 non-zero elements



bound[i].x=a, bound[i].y=b

Fig. 71.5 Generate bound array

The above process by the two kernel function: the first step calculates the incomplete result and auxiliary vector result_aid; the second step merged the result_aid into the result, so obtain the final result.
The first kernel:

1. Calculate the product of 1,024 elements and the corresponding vector elements, saved to the shared memory.
2. According to the boundary row number that Bound recorded, read the value of rpos. Assume the fragment contains 100 lines, then the adjacent 100 Half-Warp thread read. If the number of line is more than Half-Warp number, through the cycle solution.
3. Assuming that this fragment contains 100 lines, so first 100 Half-Warp will products accumulate corresponding single element to the registers. Then, the first and last row that corresponding Half-Warp will accumulate results in a result_aid, the rest of the corresponding Half-Warp will write the result. The number of line is more than the number of Half-Warp, also used the method of cycle.

The second kernel:

1. thread i corresponding Bound[i]
2. if(Bound[i].x == Bound[i-1].y) thread is not work;
else if(Bound[i].x! = Bound[i-1].y)
{thread work;
while (Bound[i].y == Bound[i + 1].x) i ++;}

71.5 Experimental Results

We experimentally evaluated our system using NVIDIA Tesla C1060, connected to Windows 7 system. The development environment is VS2010 IDE. The CUDA kernels were compiled using NVIDIA CUDA Complier (nvcc) to generate the device code that was then launched from the GPU. The host programs were compiled using the C language. We used CUDA used version 4.0 for our experiment. The architectural configurations are presented in Table 71.1.

According to the difference in the SpMV sparse matrix format, we, respectively, marked SpMV as CSR-R-GPU, CSR-B-GPU. CSR-R-GPU is not optimized SpMV kernel. CSR-B-GPU is realized by a new algorithm that proposed in this paper, which is introduced a new data structure contains Bound (Figs. 71.6, 71.7).

Through the analysis, we can get the following conclusion: for CSR-B-GPU kernel, the performance in the matrix Protein, PEM/Spheres, FEM/Cantilever, and FEM/Accelerator is obviously lower than other kernel. But, in Economics, Epidemiology, and Web base matrix, the performance of CSR-B-GPU kernel is higher than other kernel, especially in Web base matrix. When the average number of non-zero elements in each line is little, the performance is higher than other kernel. But when the average number of non-zero elements in each line is more,

Table 71.1 Test matrix sets

Matrix	Row (column)	The number of non-zero	The number of non-zero each line
Protein	36,417	4,344,765	119.3
PEM/spheres	83,334	6,010,480	72.1
FEM/cantilever	62,451	4,007,383	64.1
Economics	206,500	1,273,389	6.1
Epidemiology	525,825	2,100,225	3.9
FEM/accelerator	121,192	2,624,331	21.6
Web base	1,000,005	3,105,536	3.1

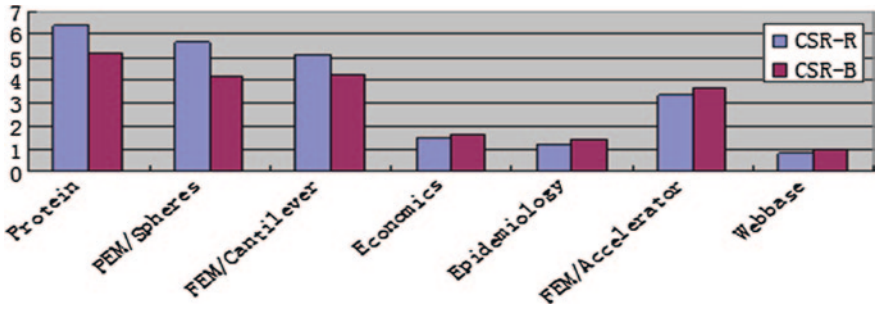


Fig. 71.6 Computing performance based on CSR SpMV kernel on GPU

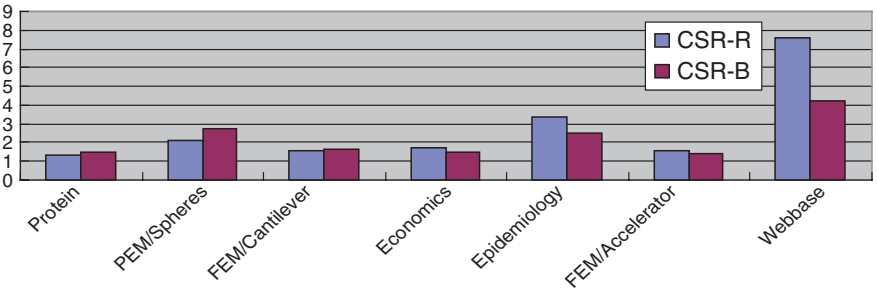


Fig. 71.7 Execution time based on CSR SpMV kernel on GPU

its performance is not ideal. One of the reasons is the kernel use many shared memory, so the optimization algorithm also needs to improve

71.6 Conclusion

This paper takes memory bottlenecks algorithm SpMV as an example, combines the characteristics of the problem with a special system of the GPU, using thread mapping, merging access, data reuse, avoiding branch and thread optimization,

optimization GPU computing on the sparse matrix storage format CSR. From the experiment, we can see these optimization strategies to be effective, there are still areas for improvement, need to be more refined.

References

1. Shereshevsky M, Cukic B, Crowel J, et al (2003) Software aging and multifractality of memory resources proceedings of DSN 2003, vol 21. IEEE Computer Society, USA, pp 721–730
2. V'azquez F, Garz'on EM, Fern'andez JJ (2010) A matrix approach to tomographic reconstruction and its implementation on GPUs. J Struct Biol 170:146–151
3. Bik AJC, Wijshoff HAG (1996) Automatic data structure selection and transformation for sparse matrix computations. IEEE Trans Parallel Distrib Syst 7(2):109–126
4. Im EJ, Yelick K (2010) Optimizing sparse matrix-vector multiplication on SMPs, vol 21(04). Computer Science Division, University of California, Berkeley, pp 56–58
5. Lee BC, Yelick RW, Demmel JW, Katherine A (2006) Yelick performance model for evaluation and automatic tuning of symmetric sparse matrix-vector multiply, vol 14(03). Computer Science Division, University of California, Berkeley, pp 79–84
6. Fatahaian K, Sugerman J, Hanrahan P (2004) Understanding the efficiency of GPU algorithms for matrix-matrix multiplications. In: Graphics hardware, vol 04, pp 133–138
7. Bell N, Garland M (2008) Efficient sparse matrix-vector multiplication on CUDA NVIDIA technical report NVR-20080004, NVIDIA Corporation
8. Baskaran MM, Bordawekar R (2009) Optimizing sparse matrix-vector multiplication on GPUs. IBM research report RC24704 04:5405–5408
9. Chen B (2010) Research on performance optimization of heterogeneous platform based on CPU-GPU and multicore parallel programming model, University of Science and Technology of China, Anhui, pp 30–45
10. Shahnaz R, Usman A, Chugtai IR (2011) Review of storage techniques for sparse matrices. Pakistan Inst Eng Appl Sci 9:118–123
11. Barrett R et al (1994) Templates for the solution of linear systems: building blocks for iterative methods, vol 35. SIAM Press, Philadelphia, pp 623–627
12. Chen H (2012) Parallel technology for implementing sparse matrix vector on GPU, vol 11. School of Computer. Wuhan University, Wuhan, pp 44–47
13. Williams S et al (2009) Optimization of sparse matrix-vector multiplication on emerging multicore platforms. Parallel Comput 35(3):178–194