Optimizing Sparse Matrix-Vector Multiplication on CUDA

Zhuowei Wang
school of computer
wuhan university
Wuhan, China
wangzhuowei0710@163.com

Xianbin Xu
school of computer
wuhan university
Wuhan, China
xbxu@whu.edu.cn

Wuqing Zhao
school of computer
wuhan university
Wuhan, China
whuzhwq@163.com

Yuping Zhang
school of computer
wuhan university
Wuhan, China
yuping.whu@gmail.com

Shuibing He
school of computer
wuhan university
Wuhan, China
hesbingxq@163.com

***Abstract*—in recent years, GPUs have attracted the attention of many application developers as powerful massively parallel system. CUDA as a general purpose parallel computing architecture make GPUs an appealing choice to solve many complex computational problems in a more efficient way. In this paper, we discuss implementing optimizing spare matrix-vector multiplication on NVIDIA GPUs using CUDA programming model. We outline three optimizations include: (1) optimized CSR storage format, (2) optimized threads mapping, and (3) avoiding divergence judgment. We experimentally evaluate our optimizations on GeForce 9600 GTX, connect to Windows xp 64-bit system. In comparison with NVIDIA's SpMV library and NVIDIA's CUDDPA library, the results show that optimizing sparse matrix-vector multiplication on CUDA achieves better performance than other SpMV implementations.**

Keywords- GPUs, CUDA, SpMV, NVIDIA's SpMV library, NVIDIA's CUDDPA library

I. INTRODUCTION

With the high demand market for realtime, high-definition 3D graphics, Graphics Processing Units（GPUs） has evolved into a highly parallel, multithreaded, manycore processor with enormous computational horsepower and very high memory bandwidth. These advances in GPUs open a new epoch of GPU computing. The future of multicore CPUs and manycore GPUs means that mainstream processor chips are now parallel systems. Currently, the use of GPUs for general purpose applications has exceptionally increased in the last few years thanks to the availability of Application Programming Interface (APIs). CUDA's parallel programming model is designed to overcome the challenge that developing application software that transparently scales its parallelism to leverage the increasing number of processor cores and to solve many complex paralleled problems.

It is proved that sparse matrix-vector multiplication (SpMV) to be of particular importance in computational science and practical engineering projects [6]. The SpMV method is that computing a vector y as a result of multiplying a sparse matrix A by vector x ($y=Ax$). Several implementations of SpMV have been developed with CUDA and evaluated on NVIDIA's GPU. In SpMV method calculations involving far more memory accesses per floating point operation, because of irregular and indirect memory accesses. So the optimization of the sparse matrix-vector product is a challenge for improving performance linear systems, partial differential equations form a wide spectrum of scientific and engineering disciplines. Therefore, many efforts must be spent to accelerate the computation of SpMV.

The aim of this approach is to optimize sparse-vector multiplication on a modern GPU, specifically, on the NVIDIA GeForce 9600 GT using the CUDA parallel programming model. Many former related works covers a variety of formats including Diagonal Formats (DIA), ELLPACK Formats (ELL), Coordinate Format (COO), Compressed Sparse Row Format (CSR), Hybrid Format (HYB), and Packet Format (PKT) to store the sparse matrix in order to explore the best possible use of the GPU for a variety of algorithmic parameters. Among of these formats, CSR is the most common sparse matrix storage format. Our work is based on the CSR format. In this paper, at the first, we evaluated a naive non-optimized implementation of the SpMV on NVIDIA GPU architecture. We found that thread mapping and data access strategies are the most important keys improving the performance on SpMV kernel on CUDA. We take account of these problems and propose three optimizations to improve performance of SpMV kernel: （1）optimized CSR storage Format. (2) Optimized threads mapping, (3) avoiding divergence judgment. We compare our approach against two existing SpMV CUDA implementations, namely, NVIDIA's CUDDPA library [3], and NVIDIA's SpMV library [2].

In section 2, we compare the former related work. NVIDIA GPU architecture and CUDA programming model are presented in section3. In section4, the optimizations and implementation of our approach is discussed in detail. Experimental results are presented in section 5. Finally, we give the conclusions and future work in section 6.

II. RELATED WORK

There has been significant amount of work on optimizing sparse matrix computations (SpMV). Most of works have concentrated on optimizing sparse matrix kernel on general-purpose architectures [8].

In [1], Vuduc have discussed descriptions of many storage formats and supported experimental data on CPUs to optimizing SpMV for CPUs. SpMV implementations on CPU can only achieve several percent of CPU's peak performance but have pressure on memory access and data reuse. In [2], 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.

Recently, implementing blocked sparse matrix-vector multiplication for NVIDAI GPUs is investigated [3]. If values or locations of non-zero elements can be efficiently computed, a specialized implementation will likely demonstrate better performance. In [4], the author describes a new hybrid storage format and present experimental results aim to reduce the memory bandwidth required to read coordinates of non-zero elements, by using blocked storage format.

In [7], with indirect and irregular memory accesses resulting in more memory access per floating point operation, the author proposed optimizations to effectively develop a high-performance SpMV kernel on NVIDIA GPUs: exploiting synchronization-free parallelism, optimized thread mapping based on the affinity towards optimal memory access pattern, optimized off-chip memory access to tolerate the high access latency, and exploiting data reuse .

Due to thread mapping and data access strategies are the important keys, our work is based on CSR formats and will try to take account of these problems and propose three optimizations. These features can help to enhance the performance on SpMV kernel on CUDA.

III. Parallel Programming with CUDA

A. *Parallel Programming with CUDA*

CUDA programming model regard CPU as host and GPU as co-processor or device. In this model, CPU and GPU work together and carry out their own duties. CUDA parallel computing function running on GPU is called kernel. Kernel function is not a complete program but one step can be executed in parallel in a whole CUDA program. The CUDA kernel is executed on a set of threads. The threads are organized into groups called thread blocks. One or more than one blocks consist of a grid. There are two parallel levels in a kernel function. That is to say blocks in a grid paralleling and threads in a block paralleling. Each thread in a thread block is solely identified by its thread id (threadIdx) within its block and each thread block is solely identified by its block id (blockIdx).

Every thread has its own memory, register, and local memory. Every thread also has a shared memory. All threads in a grid can access the same piece of global memory. In addition, there are two kinds of only read memory can be accessed by all threads: constant memory and texture memory. They are optimized for different applications. Register is on-chip high-speed memory, execution units can access register in a very low latency. Daters in local memory are existed in graphic memory not in on-chip registers or memory. So it is slow to access local memory. The global memory is a large memory and has a very high latency. The shared memory also is on-chip high-speed memory and is organized into banks. When multiple addresses belonging to the same bank are accessed at the same time, it results in bank conflict. Constant memory is read-only address space. It is favorable when multiple processor cores load the same value from the cache because it has only a single port. Texture cache has higher latency but it has a better accelerate ratio for accessing large amounts of data and non-aligned accessing.

CUDA used SIMT (Single Instruction, Multiple Thread) execution model. To manage large populations of threads efficiently, block can be divided smaller warp which the threads of a block are executed in groups of 32. A warp executes a single instruction at a time across all its threads.

B. *Problem Proposed*

To develop codes for GPUs with CUDA, the programmer has to think about two GPU architectural characteristics: thread mapping and management of the memory hierarchy.

The size of every thread block is defined by the programmer. When all threads of the same warp execute the same instruction sequence, the maximum instruction throughput is got. If any flow control instruction can cause the threads of the same warp to diverge, the different executions paths have to be serialized, so the performance in a sharp decline. In sparse matrix vector multiplication kernel, if all executed rows in sparse matrix can not fill with all thread blocks, it can increase the branch of SpMV kernel function determine and reduce the computing performance [11].

Another key to take advantage of GPUs is related to memory management. Figure1 illustrate the different kinds of memory available on GPUs with different access times and sizes that constitute a memory hierarchy [9, 10]. The off-chip memory latency needs to be efficiently to hidden to fully exploit the massive computing resources of the GPUs. Matrix vector multiplication is a memory-bound application kernel because each element in matrix is brought from memory is used only once in the computation. More than two memory operations for accessing a single non-zero matrix element are heavily memory-bound in sparse matrix vector (SpMV) [5]. As a result of GPUs have different kinds of memory with different access times, it is important to optimized thread mapping to ensure optimized memory access. At the same time, devising appropriate formats to store the sparse matrix have to be taken into account for the parallel computation and the memory accesses are tightly related to the storage format of the sparse matrix.

To sum up, in order to fully use GPUs parallel advantages, we have to consider two main goals: (1) to balance the computation of the set of threads (2) optimized memory-bound applications. Section 4 will discuss optimize SpMV computations on CUDA in detail.

IV. Implementation of the Optimizations

Sparse matrix has several storage formats such as DIA, ELL, COO, CSR, HYB, PKT and so on. These storage formats are described detailed in [1]. In [1], the author proposed two CSR SpMV kernels for the CSR sparse matrix format: scalar SpMV kernel, vector SpMV kernel. Scalar

SpMV kernel is a straightforward CUDA implementation with using one thread per matrix row. Its performance suffers drawbacks like column indices and nonzero values for a row are stored contiguously but not accessed simultaneously. Vector SpMV kernel improve on scalar SpMV kernel drawbacks which using one warp per matrix row. The vector kernel accesses indices and data contiguously and therefore overcomes the principal deficiency of the scalar approach.

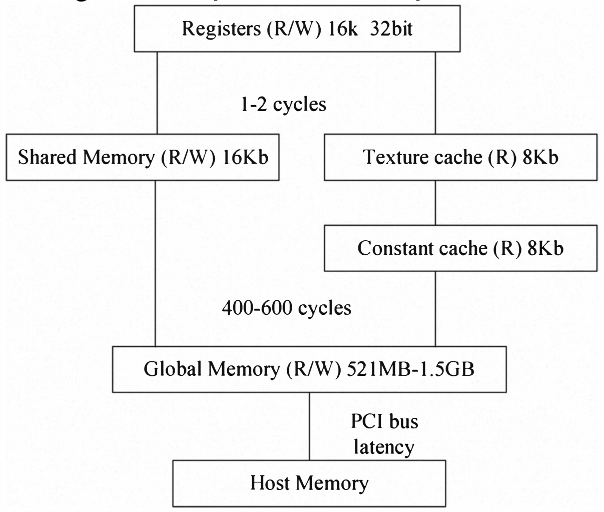

Figure 1. Different access times and sizes of GPU Memories

In the 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. Figure 2 illustrate vector SpMV kernel for CSR sparse kernel.

A. *Optimized CSR storage format*

Sparse matrix's storage and computation can be carried out only for non-zero elements, and now the more popular storage format is CSR format. CSR format is a line of compressed format which alter storing two-dimensional array of sparse matrix into three one-dimensional arrays. There are three arrays stored in COO compressed format: row, col, and element. Row array stores row indices of non-zero elements of each line. Column indices of non-zero elements are deposited in col array and element array stores the value of non-zero elements. CSR compressed format improve on COO compressed format and still storage col array and element array. The important improvement is that a third array of row pointers, ptr, takes the CSR representation. For an M-by-N matrix, ptr stores the offset the i-th row in ptr[i]. The last entry in ptr stores all non-zero numbers of elements in sparse matrix. So the numbers of ptr array have one more than these of row array. CSR compressed format illustrated in figure3.

In the CSR compressed format, col array stores column indices of non-zero elements of each row, in this paper we modify CSR compressed format make col array store the vector value corresponding to the non-zero element multiplying with. The advantage to make modify is that the vector value do not need to copy from the CPU to GPU and reduce the transmit time between CPU and GPU. At the same time it also reduces kernel accessing the global memory. Global memory can provide high bandwidth but also has a higher accessing latency. The modified method reduces the higher accessing latency in global memory and improves computing time of sparse matrix-vector multiplication. The optimized CSR storage format is described in figure4. We resume the vector is [1, 2, 3, 4].

```
__global__ void
Spmv_csr_vecot_kernel( const int nun_rows,
                       const int     *ptr,
                       const int     *indices,
                       const float   *data,
                       const float   *x,
                             float   *y )
{
        __sharred__ 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_row)
        {
            int row_start = ptr[row];
            int row_end  = ptr[row+1];

            vals[threadIdx.x] = 0;
            for (int jj=row_star+lane; jj<row_end; jj +=32)
            vals[threadIdx.x]  += data[jj] * x[indices[jj]];

            if (lane < 16)  vals[threadIdx.x] += vals[threadIdx.x  +16];
            if (lane <  8)  vals[threadIdx.x] += vals[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];
        }
}
```

Figure 2. Vector SpMV kernel for the CSR sparse matrix

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

ptr = [0, 2, 4, 7, 9]
col = [0, 1, 1, 2, 0, 2, 3, 1, 3]
element = [2, 5, 3, 9, 7, 4, 6, 4, 8]

Figure 3. CSR storage format representation of A

ptr = [0, 2, 4, 7, 9]
col = [1, 2, 2, 3, 1, 3, 4, 2, 4]
element = [2, 5, 3, 9, 7, 4, 6, 4, 8]

Figure 4. Optimized CSR storage format representation of A

B. *Optimized threads mapping*

A straightforward CUDA implementation uses one thread per matrix row. Its performance suffers from several drawbacks. The most significant among these problems is the manner in which threads within a warp access the CSR indices and data arrays. While the column indices and non-zero values for a given row are stored contiguously in the CSR data structure, these values are not accessed simultaneously. Instead, each thread reads the elements of its row sequentially. Vector kernel assigns one warp to each matrix row. The CSR vector kernel requires coordination among threads within the same warp. The vector kernel accesses indices and data contiguously and therefore overcomes the principal deficiency of the scalar approach. The given kernel function in the paper [1] using the natural synchronization of each warp take the computing method of each warp deal with daters in a row. A warp consists of 32 warps. That is to say those using 32 threads compute daters in a line. It can improve computing performance. However, in practical application process, we find that the characteristic of sparse vector in reality is the numbers of row is big but non-zero elements of each row are very small. In general, the numbers is usually smaller than 32. If we use the method using 32 threads to deal with elements in a row, you will surprise that many threads in 32 threads of a warp are running validly. In this situation, it waster thread resources and reduce computing time. Our approach is to calculate the average number of non-zero elements each row in sparse matrix. If the numbers of non-zero elements each row in sparse matrix are greater than 32, we will set num_thread to 32. Otherwise set num_thread to slightly more than an average number of non-zero elements in each row. The average number is best stetted to a multiple of 2, because many operations in CUDA are all executed by 16 threads a group. 16 is a magic number in CUDA, especially half-warp is more important than a warp when optimizing instruction stream and memory.

C. *Avoiding divergence judgment*

CUDA uses SIMT （single instruction multiple thread）execution model. If we need to control the behavior of a single thread, we have to use the divergence but reduce efficiency greatly. Threads in the same block start in the same instruction address and are implemented in different branches. But in fact, executing instructions of all threads in a warp are the same due to all SP (Stream Processor) in a same SM (Stream Multiprocessor) using the same access instruction and firing instruction units. The performance of divergence is greatly weakened. For example, when threads of a warp all jump into the same branch, the actual execution time is the implementation time of this branch. If all threads of a warp jump into other different branches, the spent time will be the sum of the implementation time of each branch. In SpMV real operation, a block can deal with several rows of sparse matrix. We assume that there are 512 threads in a block, if the sparse matrix rows divide the number of rows calculating in a block is no equal to an integer; it means that all managing rows in sparse matrix can not be filled with all blocks. Our approach is zeros are padded to fill with the last block and ensure that the all blocks in a grid full of threads. So the method can reduce the check of implementing kernel function divergence and improve computing efficiency

V. EXPERIMENT RESULT

A. *Test platform*

We experimentally evaluated our system using NVIDIA GeForce 9600 GT, connected to Windows xp 64-bit system. The development environment is VS2005 IDE. The CUDA kernels were complied using NVIDIA CUDA Complier (nvcc) to generate the device code that was then launched from the GPU. The host programs were complied using the C language. We used CUDA used version 2.1 for our experiment. The architectural configurations are presented in Table I.

TABLE I. TEST PLATFORM SPECIFICATIONS

GPU	NVIDIA GeForce 966 GT
CPU	Intel (R) Core (TM) 920
OS	Windows xp 64 bit
CUDA	CUDA 2.1 Beta
IDE	Microsoft visual studio 2005
DRAM	6GB

We use 3 sparse matrices from the sparse collection described in [12]. The selected sparse matrices represent a wide variety of real applications including modeling, structural engineering and linear programming. Every matrix has properties of number of rows, columns, and elements of matrix. The properties of 6 matrices are showed in Table II.

TABLE II. 3 SPARSE MATRICES

Matrix	Rows	columns	Nonzeros
m1	3200	3200	18800
m2	93280	93280	652247
m3	659033	659033	5959282

B. *Performance evaluation*

1) Compare our approach with NVIDIA's SpMV library: We first compare the performance of our implementation with that of NVIDIA's SpMV library. As discussed earlier, figures refer to the optimizations such as avoided determine, optimized CSR storage format and thread mapping show that our optimizations out-perform the NVIDIA's SpMV library. NVIDIA's SpMV library shows that a naïve attempt to parallelize CSR SpMV using one warp to each matrix row. But it have a major setback that when a matrix with highly variable number of non-zero per row, it is likely that many threads with a warp will remain idle while the thread with longest row continues iterating. In our approach we assign dynamic threads size to each matrix row to save computing resources. So the GPU kernel time in our approach is less than that in NVIDIA's SpMV library. We also modify CSR compressed format and reduce the transmit time between host and device. The results showed in TableIII allow us to highlight the following major points:

a) The performance obtained by our implementation and NVIDIA's SpMV library increases with the number of non-zero entries in the matrix.

b) With the increase number of non-zero entries in the matrix, the speedup of optimizing sparse matrix is faster than that of NVIDIA's SpMV library.

2) Compare our approach with NVIDIA's CUDPP library: NVIDIA has released a library called CUDDP for data-parallel algorithm primitives, which has an implementation for SpMV for NVIDIA GPUs. The CUDPP implementation has inefficient global memory accesses, shared memory accesses with bank conflicts in some stages of their algorithm, and higher synchronization across threads, leading to poor over-all performance. TableⅣ shows that both GPU-KERNEL time and GPU-HostToDevice Memory Copy Time are less than those in NVIDIA's CUDPP library

VI. Conclusion and Future Work

In this paper, we propose optimizations of sparse matrix-vector multiplication on NVIDIA GPUs using CUDA programming model. First we analyze the various challenges in extracting high-performance from a prominent memory bound kernel. Then, we develop three optimizations that take into account both the application and the architectural characteristics. Finally, we evaluate our approach compare with NVIDIA's SpMV library and NVIDIA's CUDDP library and obtain significant performance improvements over exiting parallel SpMV implementation. In the future, we plan to consider how matrix reordering can be used to reduce cache misses on GPU hardware and built a runtime sophisticated inspection to exploit data reuse and optimize memory access.

References

[1] Vuduc, R.W.: Automatic performance tuning of sparse matrix kernels. Technical report (2003)

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

[3] CUDDP: CUDA Data Parallel Primitives Library. http://www.gpgpu.org/developer/cudpp/.

[4] Alexander Monakov and Arutyun Avetisyan. "Implementing Blocked Sparse Matrix-Vector Multiplication on NVIDIA GPUs", LNCS, vol.5657, pp. 289-297. Springer, Heidelberg (2009).

[5] FATAHAIAN K., SUGERMAN J., HANRAHAN P.: Understanding the efficiency of GPU algorithms for matrix-matrix multiplications. In Graphics Hardware 2004 (Aug. 2004), pp. 133-138.

[6] F Vazquez, E M. Garzon, J. A. Martinez, J J. Fernandex, The sparse matrix vector produce on GPUs, Computer Architecture and Electronics Dep., University of Alimeria, Jun 2009.

[7] M.M. BASKARAN, R. BORDAWEKAR. Optimizing Sparse Matrix-Vector Multiplication on GPUs. IBM Research Reprot RC24704. April 2009.

[8] Aart J. C. Bik and Harry A. G. Wijshoff. Automatic data structure selection and transformation for sparse matrix computations. IEEE Transactions on Parallel and Distributed Systems, 7(2):109-126, February 1996.

[9] DOTSENKO, Y., GOVINDARAJU, N. K., SLOAN, P.-P.,BOYD, C. AND MANFERDELLI, J. 2008. Fast scan algorithms on graphics processors. In ICS'08: Proceedings of the 22nd annual international conference on Supercomputing, ACM, New York, NY, USA, 205-213.

[10] Buatois, L., Caumon, G., Levy, B.: Concurrent number cruncher: An efficient sparse linear solver on the GPU. In: Perrott, R., Chapman, B.M., Subhlok, J., de Mello, R.F., Yang, L.T. (eds.) HPCC 2007. LNCS, vol. 4782, pp. 358-371. Springer, Heidelberg (2007).

[11] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing, pages 1-12, 2007.

[12] http://www.cise.urf.edu/research/sparse/matrices/

TABLE III. Compare our approach with NVIDIA's SpMV library

Problem scale	element	18880	93280	5959282
	row	3200	93280	659033
	column	3200	652247	659033
Optimization approach	GPU-kernel time	0.026145	0.140452	0.723514
	GPU-HostToDevice time	0.052019	0.355491	1.376797
	GPU-total time	0.078764	0.495943	2.120311
NVIDIA's SpMV library	GPU-kernel time	0.027415	0.169951	0.814531
	GPU-HostToDevice time	0.055580	0.339890	1.419073
	GPU-total time	0.082998	0.509841	2.233604

TABLE IV. Compare our approach with NVIDIA's CUDPP library

Problem scale	element	18880	93280	5959282
	row	3200	93280	659033
	column	3200	652247	659033
Optimization approach	GPU-kernel time	0.026145	0.140452	0.723514
	GPU-HostToDevice time	0.052019	0.355491	1.376797
	GPU-total time	0.078764	0.495943	2.120311
NVIDIA's CUDPP library	GPU-kernel time	0.028475	0.199741	0.789324
	GPU-HostToDevice time	0.056956	0.400109	1.578661
	GPU-total time	0.085431	0.599850	2.367985