International Journal of Grid and Distributed Computing
Vol. 9, No. 10 (2016), pp.99-106
http://dx.doi.org/10.14257/ijgdc.2016.9.10.09

Shuffle Reduction Based Sparse Matrix-Vector Multiplication on Kepler GPU

Yuan Tao[1] and Huang Zhi-Bin[2]

[1]*College of Mathematics, Jilin Normal University, Siping Jilin 136000, China*
[2]*Beijing Key Lab of Intelligent Telecommunication Software and Multimedia,Beijing University of Posts and Telecommunications, Beijing China,100876*
[1]*taoyuanjl@foxmail.com*
[2]*Huangzb@bupt.edu.cn*

Abstract

GPU is the suitable equipment for accelerating computing-intensive applications in order to get the higher throughput for High Performance Computing (HPC). Sparse Matrix-Vector Multiplication (SpMV) is the core algorithm of HPC, so the SpMV's throughput on GPU may affect the throughput on HPC platform. In the paper, we focus on the latency of reduction routine in SpMV included in CUSP, such as accessing shared memory and bank conflicting while multiple threads simultaneously accessing the same bank. We provide shuffle method to reduce the partial results instead of reducing in the shared memory in order to improve the throughput of SpMV on Kepler GPU. Experiments show that shuffle method can improve the throughput up to 9% of the original routine of SpMV in CUSP on average.

Keywords: *gpu; sparse matrix; vector; shuffle; reduction*

1. Introduction

Sparse matrix can reflect the exact properties of some applications and is widely used in HPC [1]. SpMV is the core algorithm of most sparse matrix computing, the throughput of SpMV on GPU may impact the performance of HPC immediately. GPU has been widely used in HPC to accelerate the computing-intensive algorithms, because it could provide much more throughput than CPU for some applications, many researchers and scientists have done a lot of research for SpMV on GPU and got some prefer results.

The seminal work of Nathan Bell [2] could get 16 GFLOPS on single GPU. The primary contribution is that different threads store the partial sum of productions in the shared memory and reduce in the shared memory for the final result by means of the different warp switching and this can avoid explicit synchronized operation, so we can get much more throughput on GPU. Limited to the GPU's architecture, there may be the much more latency while each thread accessing the shared memory or meeting bank conflict during one more threads concurrently accessing the same bank of the shared memory at the same time, the two factors above should not be ignored for current GPU for higher throughput. In the paper we focus on the issues on Kepler GPU, experiments show that the proposed method can provide at most 9% much more throughput than the original one of CUSP.

The paper is organized as followed. Section 2 gives the preliminaries of this work. In section 3 we provide shuffle reduction based CSR's SpMV on GPU. Section 4 reports the experimental result and discussion. At last we get conclusion in section 5.

ISSN: 2005-4262 IJGDC
Copyright ⓒ 2016 SERSC

2. Preliminaries

2.1. General Purpose Computing with GPU

The original GPU is used for rendering the graphics. Being pursuit for higher computing throughput of HPC, programmability has been made by hardware vendors, such as NVIDIA and AMD. In the paper, we used NVIDIA GPU and programming model, CUDA, for our experiments.

There are three ties for the CUDA's program. A thread is a basic parallel unit, multiple threads constitute a block, and multiple blocks constitute a grid, however multiple grids should be launched one more times. A CUDA program is Single Program Multiple Data (SPMD), warp of CUDA is the basic unit to be launched, a warp is consisted of 32 threads, the thread ID within warp is called lane, one block may be consisted of multiple warps. Different warp in the same block should be required to be employed in the same Stream Multi-processor (SM), and different blocks may be employed in the same SM too. Upon the current CUDA's data organization, the threads in the same block can share the data with shared memory, however the explicit synchronized command should be required in CUDA for most applications. Bank conflict will meet while multiple threads accessing the same bank at the same time, this will introduce additional latency overhead. All the threads should exchange data by means of global memory. For Kepler GPU, the threads in the same warp can exchange register's value directly by means of shuffle command [3], which can save to access the shared memory.

2.2. Compressed Sparse Row

Figure 1, shows the format of Compressed Sparse Row (CSR) for sparse matrix. It means to store the non-zeros along the row, such as val for storing the non-zeros, the length of col_ptr is the same as val, and the col_ptr is to store the non-zero's column, row_ptr is the position of first non-zeros of each row, its length is rows+1, in which rows is the row's number of sparse matrix. The method can save to store the row index of each non-zero, so it can save much more storage and communication overhead.

```
T * val              //Non-zeros of sparse matrix
integer * col_ptr  //Column's index of non-zeros
integer * row_ptr  //Row pointer
```

Figure 1. CSR Structure

2.3. Shared Memory Reducing Based SpMV

GPU is a parallel platform, different logic cores compute different elements in parallel (or we call threads in CUDA). For SpMV, the general routine is that all the threads within a warp should iterate along a non-zeros row, we should do reduction after the threads finished the iterating all the non-zeros of current row.

Figure 2, lists pseudo-code of CSR based SpMV on GPU which is provided by CUSP [2], the library of CUSPARSE [4] is the extended of CUSP. It launches the fixed number of threads, such as the max number of threads, and one warp compute a row of sparse matrix, the threads in the warp iterate all the non-zeros in a row (line 12 and 13). After the warp finished the current row, the threads store the partial sum of products into the shared memory (line 14), and they do the reduction for the shared memory from line 15 to line 24. They store the final result once they finished computing (line 25 and line 26). After all the warps iterate the rows of sparse matrix (line 8), the program is over.

 Copyright ⓒ 2016 SERSC

```
Input:
  row_ptr,col_ptr,val //Sparse matrix
  num_rows       //number of row of sparse matrix
    vector x        //vector x
Output:
  vector y        //modified vector y
begin
1  THREADS_PER_BLOCK = the number of threads per block;
2  thread_id = global thread's ID;
3  thread_lane = thread's ID of intra-warp;
4  vector_id = global warp's ID;
5  vector_lane = warp's ID of intra-block;
6  num_vectors = number of warp;

7  sum=0.0f;    //save production's partial sum for each thread
8  for ( row = vector_id to num_rows step=num_vectors)
     {

9      row_start = the first non-zero's offset of the current row;
10     row_end = the last non-zero's offset of the current row;

11     sum=0.0f;

12     for (jj = row_start+thread_lane to row_end
                    step=THREADS_PER_VECTOR)
     {
13         sum+=Ax[jj]*x[Aj[jj]];
     }

14     sdata[threadIdx.x]=sum;

                    //reduce local sums to row sum
15     if (THREADS_PER_VECTOR>16)
16       sdata[threadIdx.x]=sum=sum+sdata[threadIdx.x+16];
17     if (THREADS_PER_VECTOR>8)
18       sdata[threadIdx.x]=sum=sum+sdata[threadIdx.x+8];
19     if (THREADS_PER_VECTOR>4)
20       sdata[threadIdx.x]=sum=sum+sdata[threadIdx.x+4];
21     if (THREADS_PER_VECTOR>2)
22       sdata[threadIdx.x]=sum=sum+sdata[threadIdx.x+2];
23     if (THREADS_PER_VECTOR>1)
24       sdata[threadIdx.x]=sum=sum+sdata[threadIdx.x+1];

       //store the final result
25     if (thread_lane == 0)
26       y[row]+=sdata[threadIdx.x];
     }
End
```

Figure 2. Pseudo-Code of Shared Memory Based Reduction for SpMV on GPU

3. Shuffle Reduction Based CSR's SpMV on GPU

Reduction is required for SpMV in parallel because the partial sum is stored in distribution, the key is how to access and sum them in a short time. For Kepler, we can use the shuffle command to exchange the register's value within the warp, this can avoid of accessing the shared memory, and save the bank conflicts which will bring much more latency. We will discuss the reduction for SpMV in parallel with shuffle in the section.

Copyright ⓒ 2016 SERSC

As Figure 3, shows that we use the __shfl_xor() to reduce the different threads' partial sum [3], which are from line 14 to line 23, such as line 15-f is single-precision data and 15-d is double-precision data. However double-precision data is not supported by __shfl_xor(), we resort to PTX instruction [5], which is named __shfl().

The double-precision data's implementation's details are provided in Figure 4. We convert double-precision data into two integers at line 3, which are high bits and low bits, and do __shfl_xor() for them respectively which are at line 4 and line 5, at last we convert high bits and low bits into a double-precision data at line 6, and then return the result.

```
Input:
  row_ptr,col_ptr,val //Sparse matrix
   num_rows      //number of row of sparse matrix
   vector x       //vector x
Output:
  vector y       //modified vector y
begin
1  THREADS_PER_BLOCK = the number of threads per block;
2  thread_id = global thread's ID;
3  thread_lane = thread's ID of intra-warp;
4  vector_id = global warp's ID;
5  vector_lane = warp's ID of intra-block;
6  num_vectors = number of warp;

7  sum=0.0f;    //save production's partial sum for each thread
8  for ( row = vector_id to num_rows step=num_vectors)
     {

9     row_start = the first non-zero's offset of the current row;
10    row_end = the last non-zero's offset of the current row;

11    sum=0.0f;

12    for (jj = row_start+thread_lane to row_end
                  step=THREADS_PER_VECTOR)
     {
13      sum+=Ax[jj]*x[Aj[jj]];
     }

        //reduce local sums to row sum
14    if (THREADS_PER_VECTOR>16)
     {
15-f     sum +=__shfl_xor(sum,16,32); //single-precision data
15-d     sum +=__shfl(sum,16,32);    //double-precision data
     }
16    if (THREADS_PER_VECTOR>8)
     {
17-f     sum +=__shfl_xor(sum,8,32); //single-precision data
17-d     sum +=__shfl(sum,8,32);    //double-precision data
     }
18    if (THREADS_PER_VECTOR>4)
     {
19-f     sum +=__shfl_xor(sum,4,32); //single-precision data
19-d     sum +=__shfl(sum,4,32);    //double-precision data
     }
20    if (THREADS_PER_VECTOR>2)
     {
21-f     sum +=__shfl_xor(sum,2,32); //single-precision data
21-d     sum +=__shfl(sum,2,32);    //double-precision data
```

 Copyright ⓒ 2016 SERSC

```
      }
22    if (THREADS_PER_VECTOR>1)
      {
23-f      sum +=__shfl_xor(sum,1,32); //single-precision data
23-d      sum +=__shfl(sum,1,32);    //double-precision data
      }

            //store the final result
24    if (thread_lane == 0)
25      y[row]+=sum;
      }
End
```

Figure 3. Shuffle Based Reduction for SpMV on GPU

```
__device__ __inline__ double shfl(double x, Integer lane)
Input:
x: register variable for shuffling
lane: XOR operand with the current lane to compute target lane
Output:
   x: value of lane's x

begin
1   Integer WARP = 32;
    // Split the double number into 2 32b registers.
2   Integer lo, hi;
3   asm volatile("mov.b64 {%0,%1}, %2;":"=r"(lo),"=r"(hi):"d"(x));
    // Shuffle the two 32b registers.
4   lo = __shfl_xor(lo,lane,WARP);
5   hi = __shfl_xor(hi,lane,WARP);
    // Recreate the 64b number.
6   asm volatile("mov.b64 %0,{%1,%2};":"=d"(x):"r"(lo),"r"(hi));
   return x;
end
```

Figure 4. Shuffle for Double Precision Data on GPU [5]

4. Experimental Results and Discussion

In the section, we evaluate the effectiveness of our shuffle reduction, and compare with the SpMV routine of CUSP.

4.1. Experimental Setup

In this work, all the experiments are done on Kepler GPU listed in Table 1. All the benchmarking matrices are downloaded from Florida University [6], and most matrices are used by Nathan Bell *et. al.*, [2], in the related work, which are provided in Table 2.

Copyright ⓒ 2016 SERSC

Table 1. Test platform used in this work

Platform	GPU	CPU
Name	K20M	Intel(R) Xeon(R) CPU E5-2620
Architecture	Kepler	x86_64
Compute Capability	3.5	N/A
Cores	13×192=2496	4 way×6 cores
Compiler and Runtime	CUDA 5.0	gcc 4.4.6
Operation System Kernel	Red Hat Enterprise Linux Server release 6.2 X86_64 2.6.32-220.el6.x86_64	

Table 2. Benchmarking Matrices Used in this Work

Matrix Name	Description	Rows	Columns	Non-zeros	Non-zeros Max/Mean
dense2_app	Dense matrix	2000	2000	4000K	2K/2K
cant	FEM cantilever	62K	62K	2035K	40/32
consph	FEM concentric spheres	83K	83K	3048K	66/36
cop20k_A	Accelerator cavity design	121K	121K	1362K	24/11
mac_econ_fwd500	Macroeconomic model	207K	207K	1273K	44/6
pdb1HYS	protein data bank	36K	36K	2191K	184/60
qcd5_4_app	Quark propagators	49K	49K	1917K	39/39
rail4284	Italian railways	4K	1097K	11284K	56K/3K
scircuit	Motorola. Circuit	171K	171K	959K	353/5
shipsec1	Ship section	141K	141K	3977K	84/28
webbase-1M	Web connectivity matrix	1000K	1000K	3106K	5K/3

4.2. Experimental Results and Discussion

Figure 3, and Figure 4, give the single precision data and double precision data throughput on GPU respectively. From the experimental result we can get that our shuffle reduction implementation are surpass the reduction with shared memory for all benchmarking matrices. On the average, our shuffle reducing surpass the reduction with shared memory for single precision data by 5.3% and for double precision data by 9.1%.

 Copyright ⓒ 2016 SERSC

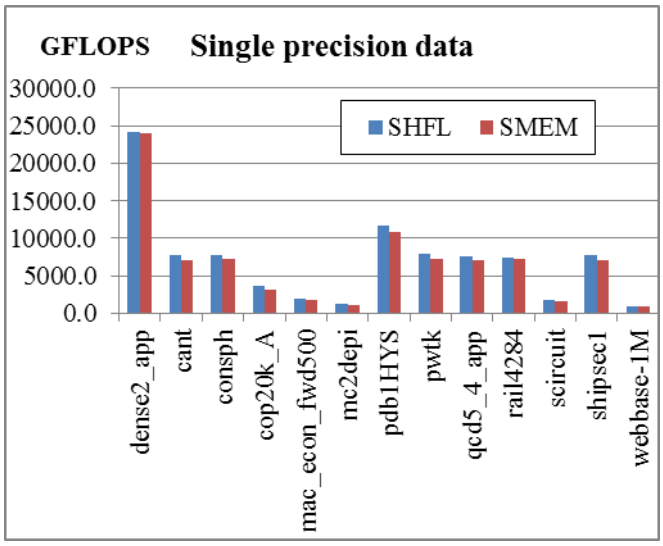

Figure 3. Throughput of Single-Precision Data

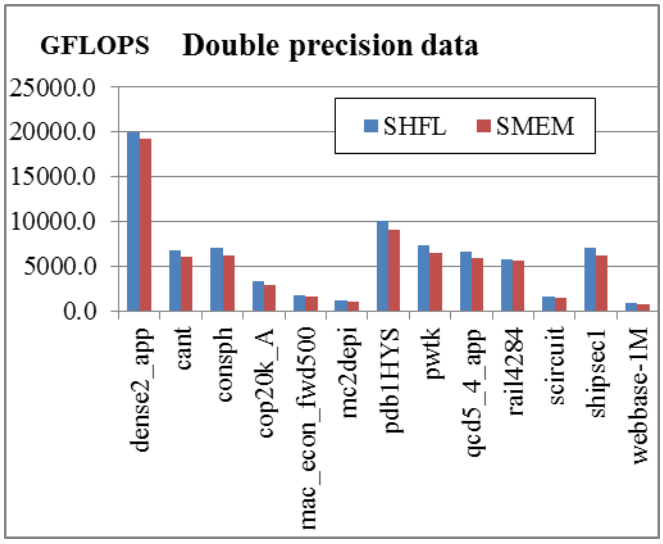

Figure 4. Throughput of double-Precision Data

Upon the experimental results, for Kepler GPU, shuffle can be more effective than shared memory for exchanging the value of register variable between threads within warp. The mainly reasons are that shuffle reduction could save to read from or write to shared memory and avoid bank conflict while multiple threads within warp accessing the shared memory simultaneously. They exchange register variable and do reduction directly, so it save a lot of time.

5. Conclusion

In this work, we focus on the reducing routine of SpMV of CUSP, we use shuffle reducing instead of reducing with shared memory. Experimental results show that shuffle reducing can provide much more throughput than reduction with shared memory within warp for Kepler GPU. Especially there is one more times iterating of SpMV for some applications, the improvement will be tremendous, so the latency of additional accessing shared memory should not be ignored.

Copyright ⓒ 2016 SERSC

Acknowledgements

We thank for Supercomputing Center, Beihang University providing the platform to do the experiment, and we thank for Zhen Lin to discuss with me. The work is supported by the China Postdoctoral Science Foundation under Grant No. 2014M550662; the fund of the State Key Laboratory of Software Development Environment under Grant No. SKLSDE-2014KF-04; the fund of the Beijing Key Lab of Intelligent Telecommunication Software and Multimedia under the project "Key technology research for the framework of Dealing with the scientific Big Data"; 2015 IBM SUR Project.

References

[1] I. S. Duff, "A Survey of Sparse Matrix Research", Proceedings of the IEEE, vol. 65,no. 500, **(1977)**.
[2] B. Nathan and M. Garland, "Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors", Portland, USA SC, (**2009**) November 14-20.
[3] CUDA C PROGRAMMING GUIDE, **(2012)**.
[4] CUDA Toolkit 5.0 CUSPARSE Library, **(2012)**.
[5] J. Demouth, "Shuffle: Tips and Tricks", GPU Technology Conference, ; California, USA **(2013)** March 18-21.
[6] T. Davis and Y. F. Hu, "The University of Florida Sparse Matrix". Collection.http://www.cise.ufl.edu/research/sparse/matrices/.

Authors

Yuan Tao, He received his ME degree in College of Information Science and Engineering from North East University in 2004, and the Ph.D. degree in computer system and architecture in Beihang University in 2015. His main research areas are parallel algorithm, parallel and distributed systems, computer system software and computer architecture.

Huang Zhi-Bin, He received his Ph.D. degree in Beihang University, China in 2013. He is now a postdoctoral fellow at Beijing University of Posts and Telecommunications. He is a member of IEEE and ACM. His research interests include high performance cache architecture and multicore architecture, memory hierarchy design, high performance computation and parallel computation.

 Copyright ⓒ 2016 SERSC