Sparse Matrix-Vector Multiplication on CUDA

(by WEIDA ZHU and FENG MI)

1.Introduction:

1.1 Sparse Matrix:

A sparse matrix is a matrix in which most of the elements are zero. By contrast, if most of the elements are nonzero, then the matrix is considered dense. (1)

1.2 Sparse Matrix-Vector Multiplication:

Sparse matrix-vector multiplication (SpMV) of the form $y = Ax$ is a widely used computational kernel existing in many scientific applications. The input matrix A is sparse. The input vector x and the output vector y are dense. In case of repeated $y = Ax$ operation involving the same input matrix A but possibly changing numerical values of its elements, A can be preprocessed to reduce both the parallel and sequential run time of the SpMV kernel.(2)

1.3 New Opportunity using Cuda

Large sparse matrices often appear in scientific or engineering applications when solving partial differential equations. By this means, if we store every element of the matrix it may waste much space. Besides, when we want to use this matrix to calculate, it can also waste much time.(3)

And we can use cuda to help us speed up the Sparse Matrix-vector multiplication on GPU. By decompose the calculation in to several independent part, it will difinitely save much time.

However, we need to find a way to store the sparse matrix efficientely in order to make sparse matrix-vector multiplication at high speed. In short，we need to find a certain algorithm to compact data.

2. Compressed Sparse Row Format

We choose Compressed Sparse Row Format to implement our project.

CSR stores column indices and nonzero values in arrays indices[ ] and data[].

For an M-by-N matrix, ptr has length M+1 and stores the offset into the i-th row in ptr[i]. The last entry in ptr, which would otherwise correspond to the (M + 1)-st row, stores the number of nonzeros in the matrix. By this way, we can easily find that ptr[i+1] - ptr[i] represents the number of nonzero elements in i-th row.(3)

For example:

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

$$\begin{aligned} \mathtt{ptr} &= \begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix} \\ \mathtt{indices} &= \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix} \\ \mathtt{data} &= \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix} \end{aligned}$$

The advantage of using CSR format as our input to do the project is that it can extremely release the memory bandwidth bound when we deliver the context in the sparse matrix from cpu to gpu because it abandoned the storage of huge amounts of 0.

3.Our algorithm to solve the problem:

3.1 We first find an algorithm to calculate the sparse vector matrix multiplication which input is in CSR format in cuda.

This is the kernel:

__global__ void mv(int num_rows,int *ptr,int *indices,float *data,float *x,float *y)

```
{      int jj;
       int row= threadIdx.x + blockIdx.x*blockDim.x;
     if(row< num_rows){
          float dot=0;

          int row_start = ptr[row];
          int row_end = ptr[row+1];
          for ( jj = row_start; jj<row_end; jj++)
               dot += data[jj] * x[indices[jj]];
          y[row] = dot;

     }
}
```

We use each thread to calculate each row of the sparse matrix. By this way, we implement running a Sparse Matrix-vector multiplication parallel program on GPU.

Notice : y is the result, x is vector. ptr, indicies and data are introduced in previous paragraph.

3.2 Some drawbacks of the previous program and the optimization method

1. If there are too many nonzero elements in one row , this method can cause much time because each iteration each thread can only compute the product of one nonzero element and one element of the vector.

2. The global memory speed is much more lower than the sharememory. So if we use take use of the sharememory, the runtime on GPU will certainly decrease.

3.3.Optimization:

1. We use one warp to compute each row of the sparse matrix, which means the products of the nonzero elements in the sparse matrix and the elements in the vector are calculated and stored simultaneously.

2.We use the array named v[ ] in sharedmemory to store the outcome of the products of the nonzero element in the sparse matrix and the corresponding element in the vector, and then we add them up according the warp id which represents the row in the sparse matrix.

3. warp_id = thread_id / 32; We can use this method to find the warp_id according to the thread_id. The reason is that one warp contains 32 threads sequentially.

```
__global__ void mv(int num_rows,int *ptr,int *indices,float *data,float *x,float *y)
{   __shared__ float vals[128];
    int thread_id = threadIdx.x + blockIdx.x*blockDim.x;   // global thread index
    int warp_id = thread_id / 32;                          // global warp index
    int lane = thread_id & (32-1);                          // thread index within the warp
    // one warp per row
    int row = warp_id;

    if(row< num_rows){

        int row_start = ptr[row];
        int row_end = ptr[row+1];

        //compute running sum per thread
        vals[threadIdx.x] = 0;
        for (int jj = row_start + lane; jj<row_end; jj+=32)
            vals[threadIdx.x] += data[jj] * x[indices[jj]];
        //parallel reduction in shared memory
        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];

        // first thread writes the result
        if(lane == 0)
        y[row] = vals[threadIdx.x];

    }
}
```

(the optimizaed kernel):

However, though the program can be used in most cases, this program has one limitation that if the nonzero elements in one row is more than 32, we cannot use this program to calculate Sparse Matrix-vector multiplication because the number of nonzero elements in one row exceeds the number of the threads in the warp. (The spase matrix means that the percent of the nonzero elements is lower than 5%. So this program is usually practical in most cases.)

4.Our implementation and debugging

4.1 We verify the algorithm by a 4*4 sparse matrix vector multiplication to make sure the correctness.

$$A=\begin{bmatrix} 0 & 2 & 6 & 0 \\ 1 & 0 & 7 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 5 & 0 & 3 \end{bmatrix} \qquad vector=\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

Ptr = [0 , 2 , 4 , 7 , 9]

Indices = [1 , 2 , 0 , 2 , 0 , 2 , 3 , 1 , 3]

Data = [2, 6 ,1 ,7 , 5 , 3 , 9, 5 , 3]

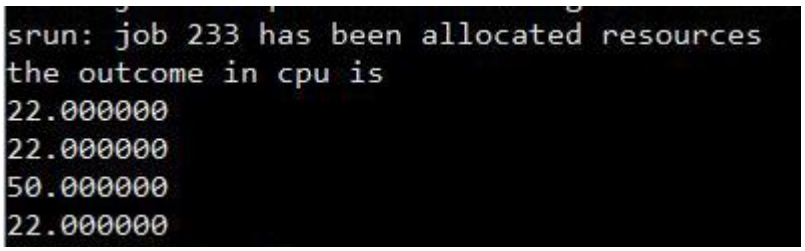
Iteration1=[0, 1, 2, 3,]

Iteration2=[0, 1, 2, 3]

Iteration3=[2,]

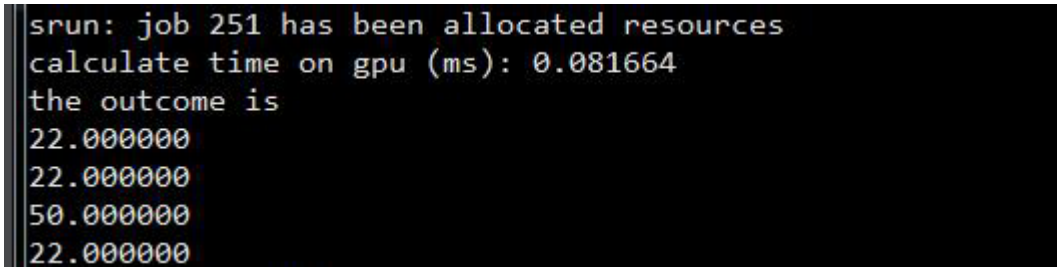
vector[4]=[1,2,3,4];

(by csrsinglethread.cu)



The result is correct.

4.2 We used this 4*4 matrix to verity our optimization algorithm



The result is also correct.(csrwarpandmemory.cu in testing folder)

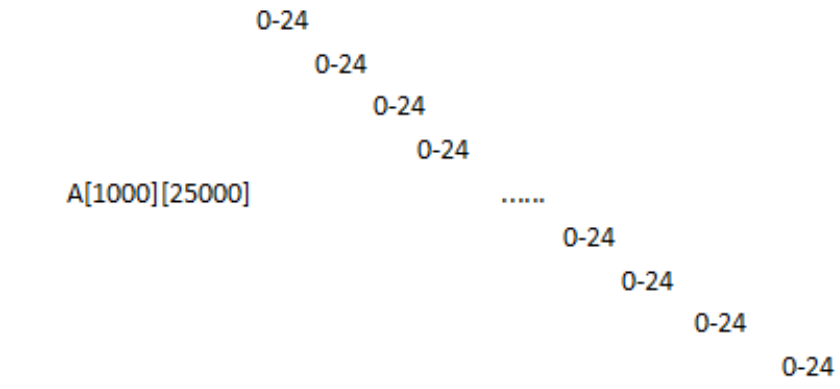
4.3We designed a sparse matrix which has 1000 rows and 25,000 columns and a vector which has 25,000 rows in 3 programs.

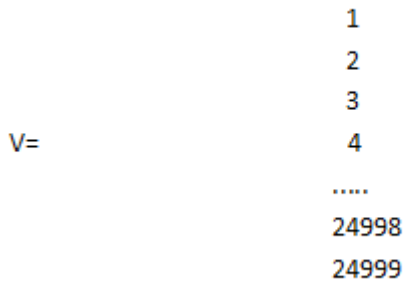
The first program is that we use only cpu to do the calculation.(6.c)

The second program is that we use gpu to calculate each row in each threads.(99.cu) (the same algorithm as 3.1)

The third program is that we calculate each row in each warps and we use the sharememory to document the outcome. (12.cu)(the same algorithm as 3.3)

The sparse matrix ,the vector and the generate method are listed below:





(v is the vector)

```
int num_rows=1000;
int ptr[1001];
int indices[25000];
float data[25000];
int i,j,m,n,q;
for( i=0; i<1001; i++)
{
    ptr[i]=25*i;
}
for( j=0; j<25000;j++)
{
    indices[j]=j;
}
for(m=0;m<1000;m++)
{
    for(n=0;n<25;n++)
    {
        data[25*m+n]=n;
    }
}

float v[25000];
for(q=0;q<25000;q++)
{
    v[q]=q;
}
```

Result:

```
24978 24979 24980 24981 24982 24983 2498
./a.out          gcc 6.c
6.c: In function ‘main’:
6.c:60: warning: incompatible implicit de
[mifeng@cuda ~]$ ./a.out
time run on cpu(ms)  = 0.199000
```

```
[mifeng@cuda ~]$ nvcc -o 99 99.cu
[mifeng@cuda ~]$ srun -N1 99
calculate time on gpu (ms): 0.134720
[mifeng@cuda ~]$
```

```
[mifeng@cuda ~]$ nvcc -o 12 12.cu
[mifeng@cuda ~]$ srun -N1 12
calculate time on gpu (ms): 0.077440
```

By measuring the time difference, we can certainly prove that the third

program has a better performance.

5.Outcome summary:

4.1 and 4.2 verified the correctness of our algorithm while 4.3 validated the effectiveness of our program.

By calculating the 1000*2500 sparse matrix vector multiplication which we designed ourselves in CSR format.

The overhead in cpu:0.199ms (6.c)

The overhead in gpu:0.1347ms (99.cu)

The overhead in optimized algorithm in gpu(using one warp to calculate each row and using shared memory):0.0774ms (12.cu)

We can get almost 62.6% performance improvement in the final result by using the optimized algorithm in gpu vs calculating the result in cpu.

(4.1 and 4.2 verified the correctness of our algorithm while 4.3 validated the effectiveness of our program.)

Reference:

https://en.wikipedia.org/wiki/Sparse_matrix(1)

https://en.wikipedia.org/wiki/Sparse_matrix-vector_multiplication(2)

http://www.nvidia.com/docs/IO/66889/nvr-2008-004.pdf(3)

http://icps.u-strasbg.fr/people/latu/public_html/wavelet/course_note.pdf

PS: We consult the algorithm of sparse Matrix vector Multiplication from internet but we fully understand it and then design the program and the huge sparse Matrix and how to input this Matrix into our program by ourselves.