Laboratory exercises (GPU)

Introduction HPC, TU Delft, course 2016–2017
Designed by Jianbing Jin and Guangliang Fu
December 2, 2016

1 Sparse Matrix-Vector Multiplication (SpMV) on Cuda

Sparse matrix structures arise in numerous computational disciplines, and as a result, methods for efficiently manipulating them are ofter critical to the performance of many applications. Sparse matrix-vector multiplication (SpMV) operations have proven to be of particular importance in computational science. They represent the dominant cost in many iterative methods for solving large-scale linear systems and eigenvalue problems that arise in a wide variety of scientific and engineering applications. The remaining part of these iterative methods (e.g., the conjugate gradient method), typically reduce to linear algebra operations that are readily handled by optimized BLAS and LAPACK implementations.

Mordern NVIDIA GPUs are throughput-oriented manycore processors that offer very high peak computational throughput. Realizing this potential requires exposing large amount of fine-grained parallelism and stucturing computations to exhibit sufficient regularity of execution paths and memory access patterns. Dense operations are quite regular and are consequently often limited by floating point throughput. In contrast, sparse matrix operations are typically much less regular in their access patterns and consequently are generally limited purely by bandwidth.

In the CUDA parallel programming model, an application consists of a sequential *host* program that may execute parallel programs known as kernels on a parallel *device*. A kernel is a SPMD (Single Program Multiple Data) computation that is executed using a potentially large number of parallel threads. Each thread runs the same scalar sequential program. The programmer organizes the threads of a kernel into a grid of *thread* blocks. The threads of a given block can cooperate amongst themselves using barrier synchronization and have a per-block shared memory space private to that block.

We focus on the kernels for sparse matrix-vector multiplication. For sparse martix formats, there are a multitude of sparse matrix representations, each with different storage requirements, computational characteristics, and methods of accessing and manipulating entries of the matrix. Some well known matrix formats are Diagonal Format (DIA), ELL Format (ELL), Compressed Sparse Row Format (CSR), Coordinate Format (COO). The Compressed Sparse Row (CSR) format is perhaps the most popular general-purpose sparse matrix representation. Recently, A new storage format Compressed Sparse Row 5 (CSR5) based on CSR was proposed by W. Liu and B. Vinter 2015 (CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication). In this exercise, we focus on comparing SpMV performance between the CSR and CSR5 format. The details of both formats are described in the following.

2 CSR and CSR5

2.1 The CSR Format

The CSR format for sparse matrices consists of three arrays: (1) **row_ptr** array which saves the start and the end pointers of the nonzeros of the rows. It has size

$m+1$, where m is the number of rows of the matrix, (2) **col_idx** array of size nnz stores column indices of the nonzeros, where nnz is the number of nonzeros of the matrix, and (3) *val* array of size nnz stores values of the nonzeros. Fig. 1 shows an example.

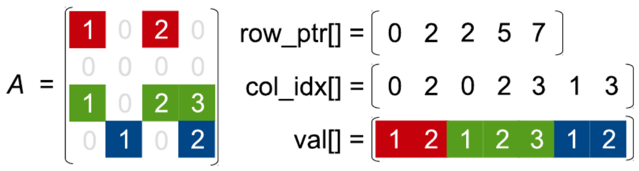

Fig. 1. A sparse matrix and its CSR format.

A few algorithms have concentrated on accelerating CSR-based SpMV with either row block methods [1] or segmented sum methods [2]. However, each of the two types of methods has its own drawbacks. As for the row block methods, despite their good performance for regular matrices, they may provide very low performance for irregular matrices due to unavoidable load imbalance. In contrast, the segmented sum methods can achieve near perfect load balance, but suffer from high overhead due to more global synchronizations and global memory accesses. Furthermore, none of them can avoid an overhead from preprocessing, since certain auxiliary data for the basic CSR format have to be generated for better load balancing or established primitives.

Therefore, to be practical, an efficient format must satisfy two criteria: (1) it should limit format conversion cost by avoiding structure-dependent parameter tuning, and (2) it should support fast SpMV for both regular and irregular matrices. To meet these two criteria, CSR5 (Compressed Sparse Row 5), was proposed. It is a format directly extending the classic CSR format. (The reason it is called CSR5 is that it has five groups of data, instead of three in the classic CSR.)

2.2 The CSR5 Format

To achieve a near-optimal load balance for matrices with any sparsity structures, all nonzero entries are first evenly partitioned to multiple 2D tiles of the same size. Thus when executing parallel SpMV operation, a compute core can consume one or more 2D tiles, and each SIMD lane of the core can deal with one column of a tile. Then the same skeleton of the CSR5 format is simply a group of 2D tiles. The CSR5 format has two tuning parameters: ω and σ, where ω is a tile's width and σ is its height. In fact, the CSR5 format *only* has these two tuning parameters. It has been investigated, for the double precision SpMV, $\omega = 32$ is optimal for the NVidia GPUs, $\omega = 64$ for the AMD GPUs, and $\omega = 8$ for Intel Xeon Phi with 512-bit SIMD units. The other parameter σ is decided by mapping the value nnz/row to σ. For the mapping, three bounds are defined: r, s and t. The first bound r is designed to prevent a too small σ. The second bound s is used for preventing a too big σ. But when nnz/row is much larger than the third bound t, σ is set to a small value u. Then,

[1] A. Ashari et al, Fast Sparse Matrix-Vector Multiplication on GPUs for Graph Applications, 2014

[2] G.E. Blelloch et al, Segmented Operations for Sparse Matrix Compution on Vector Multiprocessors, 1993

$$\sigma = \begin{cases} r & \text{if} \quad nnz/\text{row} \le r \\ nnz/\text{row} & \text{if} \quad r < nnz/\text{row} \le s \\ s & \text{if} \quad s < nnz/\text{row} \le t \\ u & \text{if} \quad t < nnz/\text{row}. \end{cases}$$

For NVidia GPUs and AMD GPUs, $(r,\ s,\ t,\ u) = (4,\ 32,\ 256,\ 4)$ and $(4,\ 7,\ 256,\ 4)$ are obtained from experiments [3].

Furthermore, extra information is needed to efficiently compute SpMV. For each tile, a tile pointer *tile_ptr* and a tile descriptor *tile_desc* are introduced. Meanwhile, the three arrays, i.e., row pointer *row_ptr*, column index *col_idx* and value *val*, of the classic CSR format are directly integrated. The only difference is that the *col_idx* data and the *val* data in each complete tile are in-place transposed (i.e., from row-major order to column-major order) for coalesced memory access from contiguous SIMD lanes. If the last entries of the matrix do not fill up a complete 2D tile (i.e., $nnz \bmod (\omega\sigma) \neq 0$), they just remain unchanged and discard their *tile_desc*.

In Fig. 2 an example matrix A of size 8×8 with 34 nonzero entries is stored in the CSR5 format. When $\omega = 4$ and $\sigma = 4$, the matrix is divided into three tiles including two complete tiles of size 16 and one incomplete tile of size 2. The arrays *col_idx* and *val* in the two complete tiles are stored in tile-level column-major order now. Moreover, only the first two tiles have *tile_desc*, since they are complete.

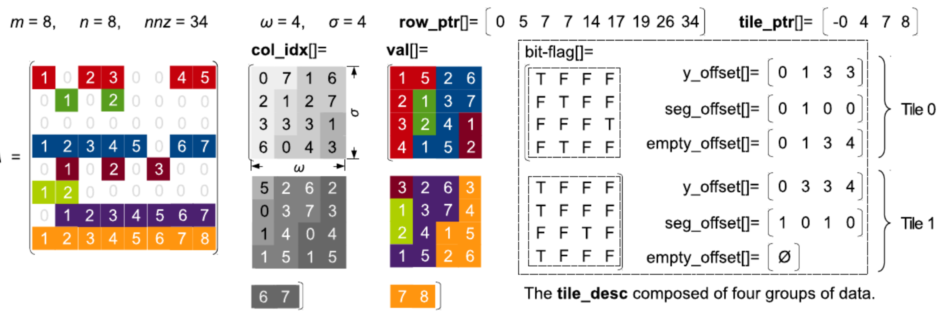

Fig. 2. The CSR5 storage format of a sparse matrix A of size 8×8. The five groups of information include *row_ptr*, *tile_ptr*, *col_idx*, *val* and *tile_desc*.

The added tile pointer information *tile_ptr* stores the row index of the first matrix row in each tile, indicating the starting position for storing its partial sums to the vector y. By introducing *tile_ptr*, each tile can find its own starting position, allowing tiles to execute in parallel. The size of the *tile_ptr* array is $p{+}1$, where $p = \lceil nnz/(\omega\sigma) \rceil$ is the number of tiles in the matrix. For the example in Fig. 2, the first entry of Tile 1 is located in the 4th row of the matrix, and thus 4 is set as its tile pointer.

Only having the tile pointer is not sufficient for a fast SpMV operation. For each tile, we also need four extra hints: (1) bit_flag of size $\omega \times \sigma$, which indicates whether an entry is the first nonzero of a matrix row, (2) y_offset of size ω used to let each column know where the starting point to store its local partial sums

[3] W. Liu and Brian Vinter, CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication, 2015

is, (3) seg_offset of size ω used to accelerate the local segmented sum inside a tile, and (4) $empty\_offset$ of unfixed size (but smaller than $\omega \times \sigma$) constructed to help the partial sums to find correct locations in y if the tile includes any empty rows. The tile descriptor *tile_desc* is defined to describe a combination of the above four groups of data.

The CSR5 format leaves one of the three arrays of the CSR format unchanged, stores the other two arrays in an in-place tile-transposed order, and adds two groups of extra auxiliary information. The format conversion from the CSR to the CSR5 merely needs two tuning parameters: one is hardware-dependent and the other is sparsity-dependent (but structure-independent). Because the added two groups of information are usually much shorter than the original three in the CSR format, very limited extra space is required. Furthermore, the CSR5 format is SIMD-friendly and thus can be easily implemented on all mainstream processors with the SIMD units. Because of the structure-independence and the SIMD utilization, the CSR5-based SpMV algorithm can bring stable high throughput for both regular and irregular matrices.

3 Exercise

3.1 Description

Since the CSR and CSR5 formats are introduced, this exercise focuses on comparing the SpMV performance on Cuda between the CSR format and CSR5 format. The performance data can be measured either with **Speed of execution** (GFLOP/s) or **Memory bandwidth utilization** (GBytes/s) or both. The questions are as follows:

(1) Compute $y = y + Ax$ first with CSR format, and then with CSR5 format. Compare and analyze the results. Here

$$A = \begin{vmatrix} 1\,7\,0\,0 \\ 0\,2\,8\,0 \\ 5\,0\,3\,9 \\ 0\,6\,0\,4 \end{vmatrix}, \qquad x = \begin{vmatrix} 1.0 \\ 2.0 \\ 3.0 \\ 4.0 \end{vmatrix} \quad \text{and} \quad y = \begin{vmatrix} 1.0 \\ 2.0 \\ 3.0 \\ 4.0 \end{vmatrix}$$

(2) similar with (1), but the matrix A changes to a structured matrix **rdb450**, and elements of x, y are set as 1. Analyze the results and check whether the previous conclusions with (1) still yields ?

3.2 Useful Reference

Code for CSR-based SpMV This code is complete, the students can directly adjust it to do analysis.

```
// parallel SPMV CSR
// compute y = y + Ax
// A =   | 1 7 0 0 |                Av = [1 7 2 8 5 3 9 6 4] =
    non zero elements
//       | 0 2 8 0 |                Aj = [0 1 1 2 0 2 3 1 3] =
    column indices of elements
//       | 5 0 3 9 |
//       | 0 6 0 4 |                Ap = [0 2 4 7 9] = pointers to
     the first element in each row
//
// x = | 1 |       y = | 1 |
//     | 2 |           | 2 |
//     | 3 |           | 3 |
//     | 4 |           | 4 |
//
// G. Fu, Dec 31, 2015
//
//

#include <stdio.h>
#include <cuda.h>

// compute multiply_row   used in csrmul_kernel
__device__ float multiply_row(int rowsize,
                     int *Aj,      // column indices for row
       float *Av,    // non-zero entries for row
       float *x)     // the RHS vector
{
float sum = 0;
for (int column=0; column < rowsize; ++column)
   sum += Av[column] * x[Aj[column]];
return sum;
}

// compute CSR format, kernel for Matrix-vector multiplication
       used in main
__global__ void csrmul_kernel(int *Ap, int *Aj, float *Av, int
     num_rows_A,
                              float *x, float *y)
{
   int row = blockIdx.x * blockDim.x + threadIdx.x ;
   if (row < num_rows_A)
   {
   int row_begin = Ap[row];
   int row_end = Ap[row+1];
   y[row] = multiply_row(row_end - row_begin, Aj+row_begin, Av
   +row_begin,x);
   }
}

// main function
int main(void)
{
     const int num_Ap = 5;
     const int num_Av = 9;
     const int num_rows_A = 4;
     const int num_rows_x = 4;
     const int num_rows_y = 4;

  int Ap_h[] = {0, 2, 4, 7, 9};
  int Aj_h[] = {0, 1, 1, 2, 0, 2, 3, 1, 3};
  float Av_h[] = {1.0, 7.0, 2.0, 8.0, 5.0, 3.0, 9.0, 6.0,
   4.0};
  float x_h[] = {1.0, 2.0, 3.0, 4.0};
  float y_h[] = {1.0, 2.0, 3.0, 4.0};

        int *Ap;
  int *Aj;
  float *Av;
  float *x;
  float *y;

  int size_Ap = num_Ap*sizeof(int);
  int size_Aj = num_Av*sizeof(int);
  float size_Av = num_Av*sizeof(float);
  float size_x = num_rows_x*sizeof(float);
  float size_y = num_rows_y*sizeof(float);

   for( int ii = 0 ; ii <= num_rows_y-1; ii++ )
      {
          printf("input y(%i) = %f\n", ii , y_h[ii]);
      }

  cudaMalloc( (void**)&Ap, size_Ap );
  cudaMalloc( (void**)&Aj, size_Aj );
  cudaMalloc( (void**)&Av, size_Av );
  cudaMalloc( (void**)&x, size_x );
  cudaMalloc( (void**)&y, size_y );

  cudaMemcpy( Ap, Ap_h, size_Ap , cudaMemcpyHostToDevice );
  cudaMemcpy( Aj, Aj_h, size_Aj , cudaMemcpyHostToDevice );
  cudaMemcpy( Av, Av_h, size_Av , cudaMemcpyHostToDevice );
  cudaMemcpy( x, x_h, size_x , cudaMemcpyHostToDevice );
  cudaMemcpy( y, y_h, size_y , cudaMemcpyHostToDevice );


       unsigned int blocksize = 128; //
       unsigned int nblocks = (num_rows_A + blocksize - 1) /
    blocksize;
       csrmul_kernel<<<nblocks,blocksize>>>(Ap,Aj,Av,
    num_rows_A,x,y);

       cudaMemcpy(y_h, y, size_y , cudaMemcpyDeviceToHost);

     // cleanup memory
        cudaFree(y);
        cudaFree(x);
        cudaFree(Av);
        cudaFree(Aj);
        cudaFree(Ap);

   for( int ii = 0 ; ii <= num_rows_y-1; ii++ )
      {
          printf("output y(%i) = %f\n", ii , y_h[ii]);
      }
     return EXIT_SUCCESS;
}
```

Program flow chart for CSR5-based SpMV The flow chart for CSR5 is given in Fig. 3. Students need to implement this by themselves, because it is an important part of exercise. You may get more information from the article (CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication).
(http://arxiv.org/abs/1503.05032)

```
1: MALLOC(*tmp, $\omega$)
2: MEMSET(*tmp, 0)
3: MALLOC(*last_tmp, $\omega$)
4: /*use empty_offset[y_offset[$i$]] instead of
   y_offset[$i$] for a tile with any empty rows*/
5: for $i = 0$ to $\omega - 1$ in parallel do
6:     sum ← 0
7:     for $j = 0$ to $\sigma - 1$ do
8:         $ptr \leftarrow tid \times \omega \times \sigma + j \times \omega + i$
9:         sum ← sum + val[$ptr$] × x[col_idx[$ptr$]]
10:        /*check bit_flag[$i$][$j$]*/
11:        if /*end of a red sub-segment*/ then
12:            tmp[$i - 1$] ← sum
13:            sum ← 0
14:        else if /*end of a green segment*/ then
15:            y[tile_ptr[$tid$] + y_offset[$i$]] ← sum
16:            y_offset[$i$] ← y_offset[$i$] +1
17:            sum ← 0
18:        end if
19:    end for
20:    last_tmp[$i$] ← sum //end of a blue sub-segment
21: end for
22: FAST_SEGMENTED_SUM(*tmp, *seg_offset)    ▷ Alg. 6
23: for $i = 0$ to $\omega - 1$ in parallel do
24:     last_tmp[$i$] ← last_tmp[$i$] + tmp[$i$]
25:     y[tile_ptr[$tid$] + y_offset[$i$]] ← last_tmp[$i$]
26: end for
27: FREE(*tmp)
28: FREE(*last_tmp)
```

Fig. 3. The CSR5-based SpMV for one tile.

3.3 Cuda on DAS4

CUDA is supported by Nvidia GPUs. The current CUDA 5.5 implementation can be added to your environment as follows:
module load cuda55/toolkit.
An SGE job script **spmv-csr-cuda.job** to submit a CUDA application on a host with a GTX480 GPU could then look like this:

```
#!/bin/sh
#$ -S /bin/sh
#$ -l gpu=GTX480
#$ -l h_rt=00:15:00
#$ -cwd
#$
. /etc/bashrc
module load cuda55/toolkit
file='spmv-csr-cuda'
nvcc $file.cu -o $file
./$file
```

Submit the job by command:
qsub spmv-csr-cuda.job,
the result will be in output file **spmv-csr-cuda.job.o********.