COMS4040A & COMS7045A Project – Report
Sparse Matrix-Vector Multiplication

8 June 2020

1 Introduction

Sparse matrix-vector multiplication (SpMV) is an important topic in linear algebra and can be found in multiple computational disciplines [Bell and Garland 2009]. SpMV represents a large portion of the cost involved in multiple iterative methods which enable the solutions of linear systems and eigenvalue related problems to be found in the scientific and engineering field. Sparse operations deal with a variety of matrices which may range from regular matrices to highly irregular matrices. When required to multiply a *n*n* matrix, which is large and has *nnz* non-zeros, by a dense vector the associated memory bandwidth when of reading the *n*n* matrix can affect performance negatively. A simple representation could store each non-zero value, plus associated row and column index as a triple. The downside of the approach is that it would require the storage of two non-zero (2*nnz*) column and row indices as well as the non-zero elements themselves [Buluç *et al.* 2009]. Therefore in this paper the standard Compressed Sparse Rows (CSR) format will be utilized for the representation of our sparse matrices since it only stores *n+nnz* row pointers and column indices.

CSR storage is one of the common matrix storage and representation methods. The method is able to store the matrix as a sequence of compressed rows. Three arrays are used to store the information, one array contains contains row pointers which are the indices of the first non-zero elements of each row, the second array contains column indices which represent the column index for each non-zero element, and the third array contains the actual non-zero elements [Pinar and Heath 1999]. An example of the representation method can be seen below. In this paper we will then utilize the CSR representation method for our matrices and perform SpMV using parallel models such as CUDA and MPI. A normal matrix representation will also be used and SpMV will be performed, this method will act as baseline when comparing results.

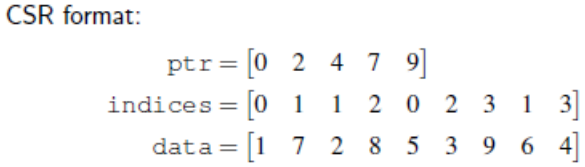

Figure 1: Example of matrix in CSR format [Bell and Garland 2009]

2 Methodology

2.1 Device information

CUDA Driver Version / Runtime Version 10.1 / 10.0
—Device 0—
Name: "GeForce GTX 1060 6GB"
CUDA Capability Major/Minor version number: 6.1
— Memory information for device —
Total global mem: 6076 MB
Total constant mem: 65536 B
The size of shared memory per block: 49152 B
The maximum number of registers per block: 65536
The number of SMs on the device: 10
The number of threads in a warp: 32
The maximal number of threads allowed in a block: 1024
Max thread dimensions (x,y,z): (1024, 1024, 64)
Max grid dimensions (x,y,z): (2147483647, 65535, 65535)

The above information relates to the GPU device used in this paper. For CPU use an i7 7700 intel CPU in addition with 16GB of RAM is used for all experiments to get results.

2.2 Method

2.2.1 Normal matrix representation and regular CPU SpMV

For this method the memory for square matrices of size are first allocated (requires $n*n$ storage), memory is then allocated for the dense vectors and the resultant vectors. The matrices and dense vectors are stored in a 1D format and the matrices are initially filled with zeros of float types. The dense vectors are then filled with ones and 1% of the matrices are filled with a non-zero value. SpMV is then performed and timed on the CPU, using row-major order for the matrices. The multiplication function can be seen below. The time taken and resultant vector is printed out as the output. The number of operations that occur is n^2.

```
void matrix_multiply_seq(float *a, float *b, float *ab, size_t width){
  int i, j, k;
  for(i=0; i<width; i++)
    for(j=0; j<width; j++){
      ab[i]=0.0;
      for(k=0; k<width; k++){
        ab[i] += a[i*width+k] * b[i];
      }
    }
}
```

2.2.2 CSR matrix representation and CUDA SpMV

Square sparse matrices matrices which can be found at SuiteSparse Matrix Collection of sizes 317*317, 1074*1074, 1612*1612 and 3876*3876 will be used in the CSR experiments. They are read in and converted to CSR format as described in the introduction. Dense vectors are created as seen above in the previous method. Memory is then allocated on the GPU for the row pointer array, column index array, value array and dense vector. The values of all the required arrays and vectors are then copied to the device and passed to the kernel which performs CSR SpMV. The time taken for the kernel to run and the resultant vector is printed out as output. The global cuda kernel code can be seen below. Code for this method was adapted from this repository.

```
1  //SpMV kernel using CSR
2  __global__ void csr_spMV(const int *row_pointer, const int *column_index,
       const float *values, const int nrows, const float *x, float *y) {
3      // Uses a grid-stride loop to perform dot product
4      int i = blockIdx.x * blockDim.x + threadIdx.x;
5      for (int a = i ; a < nrows; a = a + (blockDim.x * gridDim.x)) {
6
7          float prod = 0;
8          const int row_start = row_pointer[a];
9          const int row_end = row_pointer[a + 1];
10
11         for (int b = row_start; b < row_end; b++) {
12
13             prod += values[b] * x[ column_index[b] ];
14
15         }
16
17         y[a] = (float)prod;
18     }
19 }
```

2.2.3 CSR matrix representation and MPI SpMV

Dense vectors are created as seen above in the previous method. The MPI environment is then initialized, the size of the group associated with a communicator is found and the rank of the calling process in the communicator is also found. If the process ID is 0 then the square sparse matrices are read in and converted to CSR format as described in the introduction and the dense vectors are filled with ones to match number of rows of the input matrix. Data size and displacement vectors for every processor are computed as well as sparse matrix entry counts and displacement vectors for the processors, this enables the splitting of the work. The number or rows of the matrices are then broadcasted and elements of the row pointer array, column index array, value array and dense vector are scattered to all processors. The load for each processor is then determined and each processor then shares information with the other remote vector entries. Each processor will then send required indices of remote vector entries needed to all other processors, then all processors will share actual vector entries with each other. Results are then computed using the CSR SpMV kernel, gathered and then written to an output file. The output file will contain the dense resultant vector and time taken.

If the process ID is not 0, code for the other processors are set up similarly as described above, with main difference being that they have to first receive their required portions of data from process 0 before scattering their data, computing results and gathering results. The CSR SpMV kernel is similar to the kernel displayed in the previous method. Code for this method was adapted from this repository.

3 Results and Discussion

	317*317	1074*1074	1612*1612	3876*3876
Normal+CPU_SpMV	**77.3493**	**2867.7000**	**9831.0000**	**135220.00**
CSR+Cuda_SpMV	**0.0373**	**0.0465**	**0.0906**	**0.1640**
CSR+MPI_SpMV	**1.7860**	**1.5350**	**4.3160**	**4.868**

Table 1: Time taken (milliseconds) for matrix representation method and algorithm for varying matrix sizes

	317*317	1074*1074	1612*1612	3876*3876
CSR+Cuda_SpMV over Normal+CPU_SpMV	**2073.71**	**61670.97**	**108509.93**	**824524.3902**
CSR+MPI_SpMV over Normal+CPU_SpMV	**47.28**	**1873.09**	**4011.02**	**27777.3213**
CSR+MPI_SpMV over CSR+Cuda_SpMV	**0.02**	**0.03**	**0.04**	**0.0337**

Table 2: Speedup of methods for varying matrix sizes

In Table 1 the time taken for the normal matrix storage and CPU SpMV multiplication (Normal+CPU_SpMV) method increases drastically for an increase in matrix size, this is due to memory required to store the matrices and extra amount of computation that occurs due to all the zero elements present when the matrix vector multiplication occurs on the CPU. The CSR matrix storage format with the Cuda SpMV implementation (CSR+Cuda_SpMV) achieves impressive results when compared to Normal+CPU_SpMV, in Table 2 we see that the speedup achieved by CSR+Cuda_SpMV over Normal+CPU_SpMV are thousands of times faster with the most impressive result being 824524.39 times faster on a matrix of size 3876*3876. In Table 1 the time taken for the CSR matrix storage format with the MPI SpMV implementation (CSR+MPI_SpMV) is shorter than the Normal+CPU_SpMV method but longer than the CSR+Cuda_SpMV method. In Table 2 the speedup achieved by CSR+MPI_SpMV over Normal+CPU_SpMV is large and also impressive with the most impressive result being 27777.32 times faster on a matrix of size 3876*3876. In Table 2 we also see that the speed of CSR+MPI_SpMV over CSR+Cuda_SpMV is very low, which indicates that the CSR+Cuda_SpMV method is much faster. Therefore the CSR+Cuda_SpMV method achieves the best results.

While the CSR matrix storage format is efficient it does have a few drawbacks. The flexibility of CSR introduces thread divergence and the method suffers from non-coalesced memory accesses [Pinar and Heath 1999]. The difference in time taken between our Cuda and MPI implementations may be due to communication times between processors in the MPI implementation but may outperform Cuda on larger matrices, but the Cuda implementation is superior in this paper.

4 Conclusion

The CSR+Cuda_SpMV implementation has the best performance and the CSR matrix storage format is more efficient than regular matrix storage. Although more research needs to be conducted with different sparse matrix representation methods that address the shortcomings of the CSR format and results will have to be compared to determine the best sparse matrix representation method since this paper only focused on one representation method and how it differs across the Cuda and MPI implementations. The use of the MPI implementation also requires refining to ensure optimal results are required and possible use of shared memory may help speed up the Cuda implementation even more.

5 Appendix

Below there is output for the MPI implementation which shows the implementation was run on multiple nodes on a cluster.

Matrix size: 317*317

```
-----------------------------------------------------
Job is running on node  mscluster[16,18]-----------------------------------------------------
SLURM: sbatch is running on mscluster0.ms.wits.ac.za
SLURM: job ID is 79431
SLURM: submit directory is /home-mscluster/cpillay
SLURM: number of nodes allocated is 2
SLURM: number of cores is 8
SLURM: job name is mpi
-----------------------------------------------------

MPI Run time:  1.7860 ms
```

Matrix size: 1074*1074

```
-----------------------------------------------------
Job is running on node  mscluster[16,18]-----------------------------------------------------
SLURM: sbatch is running on mscluster0.ms.wits.ac.za
SLURM: job ID is 79435
SLURM: submit directory is /home-mscluster/cpillay
SLURM: number of nodes allocated is 2
SLURM: number of cores is 8
SLURM: job name is mpi
-----------------------------------------------------

MPI Run time:  1.5350 ms
```

Matrix size: 1612*1612

```
-----------------------------------------------------
Job is running on node  mscluster[16,18]-----------------------------------------------------
SLURM: sbatch is running on mscluster0.ms.wits.ac.za
SLURM: job ID is 79434
SLURM: submit directory is /home-mscluster/cpillay
SLURM: number of nodes allocated is 2
SLURM: number of cores is 8
SLURM: job name is mpi
-----------------------------------------------------

MPI Run time:  4.3160 ms
```

Matrix size: 3876*3876

```
-----------------------------------------------------
Job is running on node  mscluster[16,18]-----------------------------------------------------
SLURM: sbatch is running on mscluster0.ms.wits.ac.za
SLURM: job ID is 79437
SLURM: submit directory is /home-mscluster/cpillay
SLURM: number of nodes allocated is 2
SLURM: number of cores is 8
SLURM: job name is mpi
-----------------------------------------------------

MPI Run time:  4.8680 ms
```

References

[Bell and Garland 2009] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the conference on high performance computing networking, storage and analysis*, pages 1–11, 2009.

[Buluç *et al.* 2009] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 233–244, 2009.

[Pinar and Heath 1999] Ali Pinar and Michael T Heath. Improving performance of sparse matrix-vector multiplication. In *SC'99: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, pages 30–30. IEEE, 1999.