COL380

Lab 4 Report

Aditi Singla (2014CS50277)
Aayan Kumar (2014CS10201)

COO Format:

This is a simple storage scheme known as *Coordinate Format* (COO). The sparse matrix is represented by 3 arrays: **row, col** & **data** for non zero entries. This is the most general form of representation and works on any sparsity pattern.

CSR Format:

This stands for *Compressed Sparse Row* format. The sparse matrix (m*n dimension) is represented by 3 arrays:

1. **ptr** : It is (m+1) sized array and stores the offset into the ith row in ptr[i]. So, (ptr[i+1] - ptr[i]) represents the number of non zero elements in ith row.
2. **indices** : It stores the column number for each non-zero element in the matrix. For ith row, indices gives the columns for the (ptr[i+1]-ptr[i]) non-zero elements.
3. **data** : It stores the data corresponding to the indices array.

Note: 'indices' and 'data' arrays in CSR format are same as 'col' and 'data' arrays in COO format.

Algorithm:

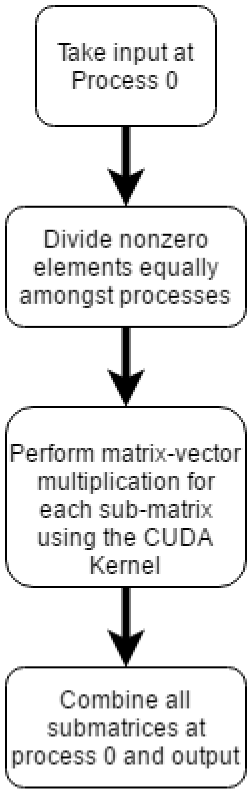
1. Take in the matrix input in COO format and convert it to CSR format
2. Split the data into multiple MPI processes in the following way:
   - Send to each process a part of the matrix A and the complete vector B. The distribution is explained below.
3. Each process does the multiplication on its part of the data, to generate an output vector, which are later collected at one process and combined to get the final answer

Design Decisions:

1. MPI has been used to implement this distribution of data amongst processes to maximise the parallelism.
2. CSR Scalar Kernel : This kernel takes one row of the matrix per thread and computes the multiplication of the row with the vector and returns. Different kernels were tried like DIA and ELL Kernel, but CSR kernel has been chosen since it allows variable number of non-zero elements per row, and hence gives more flexibility.
3. The given COO format has been converted to CSR format (instead of using COO directly), to exploit the benefits of CSR kernel.

4. Some experimentation was done with respect to the grid size and the block size to come up with the optimal values.

Schematic of the Algorithm:


Take input at Process 0
Divide nonzero elements equally amongst processes
Perform matrix-vector multiplication for each sub-matrix using the CUDA Kernel
Combine all submatrices at process 0 and output

Parallelisation Strategy:

1. The scalar kernel for CSR format has been employed here. Multiplication is carried out by one row per CUDA thread approach. These threads run in parallel to yield the output for the corresponding sub-matrix, which are then later merged to yield the final output.
2. The data split was done so as to ensure each thread gets almost an equal number of non-zero data elements as this is likely to yield a better distribution (than row wise split to different processes) and hence parallelisation.

Source:

"Efficient Sparse Matrix-Vector Multiplication on CUDA" :
(Link : http://www.nvidia.in/docs/IO/66889/nvr-2008-004.pdf**)**