Sparse Matrix-Vector Multiplication using Pthreads
CS 380P - Parallel Systems - Assignment 3
Nikhil Panpalia, Saral Jain
{nikhil, saral}@cs.utexas.edu
University of Texas at Austin

Base algorithm:
A simple matrix vector multiplication with P threads, where each thread is assigned a block of rows. The matrix is stored in the CSR format (Compressed Sparse Row).

Note: We evaluated the tradeoff of storing the matrix in the CSC (Compressed Sparse Column, better suited for Matrix Transpose* Vector computation) and COO (worse memory footprint and performance) and settled on CSR for its better performance.

Optimizations:
1) Loop Optimization [1]:
A typical matrix-vector multiplication (matrix in CSR format) consists of a nested loop where the outer loop iterates over all the rows and the inner loop iterates over columns in those rows. Since the data is stored in a sequential fashion in CSR (one row after the other), the data can be accessed by the nested loop using a single loop variable instead of two (in the typical algorithm). As expected, loop unrolling gave a good performance boost to our initial simple CSR-based algorithm.

2) Load Balancing:
We sort all the rows in the ascending order of number of non-zero elements present in those rows. Thus, after distributing the rows among threads, all the threads have almost equal amount of work since they have almost the same number of non-zero elements to process. In those matrices with a great disparity in the number of non-zero elements in each row, this method clearly outperforms the base algorithm although if the number of non-zeroes in each row across the matrix are roughly constant, this method is seen to perform worse, because of the added overhead of finding the correct row index for each thread to work on.

3) Load Balancing – Heuristic II
We combine load balancing with block allocations in this strategy, to be able to efficiently optimize the cache. In case of block allocations, random accessing of rows is not required and hence increases the performance. Here, we try to divide the number of non-zeroes each thread works on in a fair way such that each thread works on roughly the same number of non-zeroes. Depending on the nature of the input matrix, there can be differences between the number of rows each thread works on.
4) Reduction in memory footprint [1]:

We have used unsigned short (2 bytes) instead of int (4 bytes) wherever possible in the program. Since, we are also using unsigned short for column indices, this reduces the footprint of the program for larger matrices. (For example, in a 10000x10000 matrix with sparsity 70%, there are about 30,000,000 non-zero elements. And we are using 2 bytes for each of them instead of 4 bytes.) This simple trick, gave us a nice boost in the performance of our base algorithm, presumably due to lower memory footprint leading to higher chances of more effective caching.

5) Cache blocking [2]:

We also implemented cache blocking, but didn’t find it to be very effective. We converted the given matrix in MMEF format to Block Compressed Row Storage (BCRS) format. The given matrix was divided into r x c sized blocks. We tried to block the cache for r elements from the source vector and c elements from the output vector as a block of size r x c from the input matrix is being multiplied by the source vector. (Note: we only implemented cache blocking for symmetric matrices.)

We found that the performance of the algorithm did not improve much with cache blocking. One of the primary reasons could be that cache-blocking is effective only when the input matrix contains lots of dense blocks of non-zero elements. It does not work very well when the non-zeros are uniformly distributed throughout the matrix. Also, when the matrix is divided in to blocks of equal size, each thread is assigned a row of these blocks. This makes load balancing harder since the distribution of work is more coarse-grained with cache blocking than if groups of individual rows are assigned to threads.

6) Hybrid Format – a combination of Ellpack and COO storage [3]

According the [3] Ellpack is a storage scheme well suited for Vector machines and gives better performance than CSR. However, in case the number of non-zeros differ significantly in each row, it tends to occupy a large amount of memory and performs worse than simple formats. The ideal solution is to implement a hybrid format which uses EllPack and COO to store the matrix. The purpose of the hybrid format is to store the typical number of nonzeros per row in the ELL data structure and the remaining entries of exceptional rows in the COO format. We implemented all the three formats given in the paper. The performance numbers published suggest Hyb format to consistently outperform all the other formats (by a great margin), however our implementation performs worse than the standard algorithm. Our guess is that it largely depends on the hardware in place, as the paper is specifically on Spmv performance on GPU and CUDA. Also, they have much larger resources in terms of machines and are testing on matrices (~1million rows*1 million columns). The hyb matrix could outshine the simpler CSR when talking at this scale due to its better storage formats.
Instructions for compiling and executing:

To compile:
- Module load papi
- run “make clean” in the source directory (containing the “Makefile”)
- run “make”

To execute:
```
./all <matrix_file> <source_vector_file> <output_vector_file> <num_threads>
```

`matrix_file`: File containing the input matrix in MMEF format.
`source_vector_file`: File containing the source vector.
`output_vector_file`: File where the output vector will be written in MMEF format.
`num_threads`: Number of threads to be used to the computation.

For example: `./all matrix.mm inputVector.txt output.txt 32`

Performance

We measured the time it takes to calculate the matrix vector product for the test inputs provided by the TA.

<table>
<thead>
<tr>
<th>Number of threads</th>
<th>Input matrix 1</th>
<th>Input matrix 2</th>
<th>Input matrix 3</th>
<th>Input matrix 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>13173</td>
<td>6241</td>
<td>3570</td>
<td>5625</td>
</tr>
<tr>
<td>16</td>
<td><strong>9089</strong></td>
<td>7942</td>
<td>5272</td>
<td>7126</td>
</tr>
<tr>
<td>32</td>
<td>10089</td>
<td>8223</td>
<td>5820</td>
<td>7076</td>
</tr>
<tr>
<td>64</td>
<td>12073</td>
<td>9293</td>
<td>8563</td>
<td>10042</td>
</tr>
</tbody>
</table>

References:


3) “Efficient Sparse Matrix-Vector Multiplication on CUDA”