Assignment 3 - Sparse Matrix Vector Multiplication

Kristian Sørdal

May 7, 2024

Contents

1 Sparse Matrix Vector Multiplication (SpMV)

Sparse Matrix Vector Multiplication (SpMV) is a common operation used in scientific computing. It lends itself very well to parallelization, both in shared and distributed memory systems. The matrices used in SpMV are usually very big, typically on the order of at least 10^8 rows, with at least 10^9 non-zeros.

Although there are many definitions as to what a sparse matrix is, they all generally describe matrices where its worth treating non-zeros differently to zeros. For the matrices used in this project, we will be working with matrices with $\mathcal{O}(n)$ non-zeros, i.e. $\mathcal{O}(1)$ non-zeros per row. Below is an example of a sparse matrix.

$$\underbrace{\begin{bmatrix} 1 & 5 & 0 & 0 & 4 & 0 & 0 & 1 \\ 0 & 7 & 0 & 9 & 0 & 12 & 0 & 0 \\ 0 & 8 & 7 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 & 3 & 17 \\ 0 & 0 & 0 & 2 & 2 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 6 & 0 & 0 \end{bmatrix}}_{\text{A}} \quad \underbrace{\begin{bmatrix} 1 & 5 & & & 4 & & & 1 \\ & 7 & & 9 & & 12 & & \\ & 8 & 7 & & & & & \\ & & & & & 3 & 3 & 17 \\ & & & 2 & 2 & & 1 & \\ & & & & & 6 & & \end{bmatrix}}_{\text{A}}$$

Figure 1: (Left) Dense representation of A , (Right) Sparse representation of A

1.1 Representing a Sparse Matrix effectively

There are many different ways to represent a sparse matrix. The key takeway from all of them is that they don't store the zeros. For this project, the Compressed Sparse Row (CSR) format has been utilized. Storing a matrix using this format, we need 3 vectors. The `row_ptr` vector has size $n+1$, where the ith entry represents the starting index in the `col_idx` vector of the `i`th row. The `col_idx` vector has size nnz, where nnz is the number of non-zeros in the matrix. It represents the presence of a non-zero values in the matrix. If `col_idx[4]` $= 5$, then the matrix has a non-zero value at index $[4, 5]$. Finally, the `vals` vector also has size nnz, and it stores the numerical value of the non-zeros. Below is an example of a sparse matrix, and the CSR representation of the same matrix.

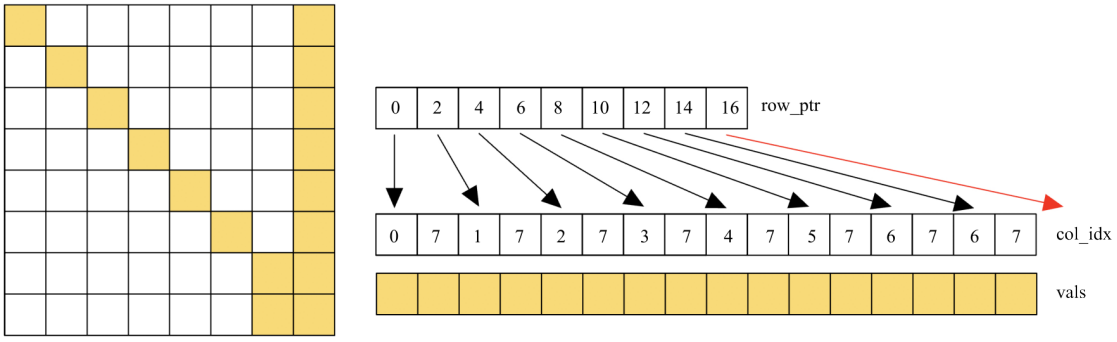

Figure 2: Compressed Sparse Row Format

1.2 SpMV Kernel

The kernel used for computing the result of an iteration of SpMV on a CSR matrix is outlined in the algorithm below. This is what will be parallelized in order to (hopefully) achieve better performance.

Algorithm 1: SpMV Kernel

Input : Row pointer: *rowPtr*, Column indices: *colIdx*, Values: *vals*, Input vector x, Result vector y
Output: Output Vector y

for $row \leftarrow 0$ **to** n **do**
  for $idx \leftarrow rowPtr[row]$ **to** $rowPtr[row+1]$ **do**
    $y[row] \leftarrow y[row] + vals[idx] \times x[colIdx[idx]]$

2 STREAM Benchmark

When performing SpMV with the CSR format, we have to be aware that the performance of our program will likely not scale with the number of threads we throw at the program, but rather the memory bandwidth of the system we are running on. Memory bandwidth is the rate at which data can be read or stored in memory. The reason for this becomes evident when we look at the kernel.

When we execute the kernel, we are repeatedly accessing memory, and do very few operations (just 2 floating point operations) per memory access. In addition to this, we are accessing the input vector `x` in a manner that leads to a large amount of cache misses. This is because it is indexed by the value stored in `col_idx`. Although this vector is sorted in increasing order when looking at a single row, the indices stored in this vector can, and usually will have a large difference in their numerical value. When we access any value at a given index in the `x` vector, the processor will prefetch successive elements in this vector and load them into the cache, allowing for very fast access. But since the numerical difference between the indices can be large, most likely the next element we need will not be prefetched, thus requiring the processor to read the value from memory, instead of reading from the cache. It is for this reason that SpMV is limited by the memory bandwidth of the system.

The STREAM is an industry standard benchmark that is used for measuring sustained memory bandwidth of the CPU in a shared memory system. The benchmark computes four different kernels named `Copy`, `Scale`, `Sum` and `Triad`. The kernels are defined as the following:

- Copy: $y[i] = x[i]$
- Scale: $y[i] = \alpha \times x[i]$
- Sum: $y[i] = x[i] + y[i]$
- Triad: $y[i] = x[i] + \alpha \times z[i]$

We are interested in the results from the `Triad` kernel, as this is the only kernel which uses 2 FLOPS per iteration, which is the same as we do in SpMV.

By running the STREAM benchmark on Brake and plotting the results, we can see the sustained memory bandwidth achieved for each thread. This will be important for comparing the results of the different implementations of parallel SpMV later on. Below we can see the results of the benchmark.

As we can see, we are able to achieve at most ≈ 33 GB/s in sustained memory bandwidth, and a peak of ≈ 2.5 GFLOPS, so we should not expect that the SpMV results will exceed this number.

3 Parallelizing SpMV

For this project, 3 different strategies were implemented for parallelizing SpMV. The following sections will discuss their implementation details.

3.1 Method 1

This method is the simplest of all methods, as it simply parallelizes the outmost for-loop of the kernel, using OpenMP's `#pragma omp parallel for` directive. From testing different scheduling methods, it was found that `dynamic` scheduling with a block size of 1024 was the best choice. The kernel now looks like the following:

Algorithm 2: SpMV Kernel

Input : Row pointer: *rowPtr*, Column indices: *colIdx*, Values: *vals*, Input vector x, Result vector y
Output: Output Vector y

for $row \leftarrow 0$ **to** n **do in parallel**
  for $idx \leftarrow rowPtr[row]$ **to** $rowPtr[row+1]$ **do**
    $y[row] \leftarrow y[row] + vals[idx] \times x[colIdx[idx]]$

3.2 Method 2

The next method is slightly more involved, as it makes use of manually assigning rows to each thread, in an attempt at achieving a better load balance for each thread, trying to ensure that no thread does significantly more work than any of the other threads.

The load balancing strategy used in this method involves computing the ideal number of non-zeros each thread should work on, by dividing the total number of non-zeros by the number of threads. Then for each thread, we greedily assign rows (without splitting them) until the sum of non-zeros in each row is at or just above the ideal number of non-zeros. The algorithm used for this is implemented in the following manner:

Algorithm 3: Naive Load Balancing

Input : Row pointer: *rowPtr*, Number of non-zeros: *nnz*, Number of threads: *numThreads*, Number of rows: *N*,
Output: Start and end indicies for each rank
partitionSize $\leftarrow$ list of size p, assigned tuples $(0, 0)$
avgNnz $\leftarrow \frac{nnz}{p}$
threadIdx $\leftarrow 0$
prevIdx $\leftarrow 0$
for $i \leftarrow 0$ **to** N **do**
  if *threadIdx* = *numThreads* -1 **then**
    partition[threadIdx] $\leftarrow$ (prevIdx, N) **break**
  if *rowPtr[i+1] - rowPtr[prevIdx] ¿ avgNnz* **then**
    partition[threadIdx] $\leftarrow$ (prevIdx,i) prevIdx $\leftarrow$ i threadIdx $\leftarrow$ threadIdx + 1

Then, for each thread, we compute the following kernel:

Algorithm 4: SpMV Kernel - Load Balanced

Input : Row pointer: *rowPtr*, Column indices: *colIdx*, Values: *vals*, Input vector x, Result vector y, Start index: startIdx, End index: endIdx
Output: Output Vector y

for $row \leftarrow startIdx$ **to** $endIdx$ **do**
  for $idx \leftarrow rowPtr[row]$ **to** $rowPtr[row+1]$ **do**
    $y[row] \leftarrow y[row] + vals[idx] \times x[colIdx[idx]]$

3.3 Method 3

The final method implemented makes use of the METIS graph partitioning library, this exposes two methods for paritioning a graph into k parts. We have the choice of using either multilevel recursive bisection, or k-way partitioning. After some testing, k-way partitioning was found to be the most stable, as the multilevel recursive bisection sometimes resulted in segmentation faults.

By using the METIS library, the goal is to hopefully achieve an even better load balance than what the greedy approach described earlier can provide. The algorithm used for load balancing with METIS is outlined below.

Algorithm 5: Metis Load Balancing

Input : Number of partitions: k, Row pointer: *rowPtr*, Column indices: *colIdx*, Values: *vals*,Input vector: inputVec, Number of non-zeros: *nnz*, Number of rows: N
Output: Start and end indicies for each rank

startIdx ← 0 initialized list of size $p + 1$
partitionMap ← 0 initialized list of size N
objVal ← NULL
ncon ← 1

```
//ubvec constrains the load imbalance, a value of 1.01 indicates that the load
  imbalance is at most 1%.  Note that there is no guarantee that this is
  achieved
```
ubvec ← 1.01

if $k = 1$ **then**
  startIdx[1]=N **return**

```
//METIS partitioning
```
rc ← METISPartGraphKway(&N, &ncon, *rowPtr, *colIdx, nullptr, nullptr, nullptr, &k, nullptr, &ubvec, nullptr, &objval, *partitionMap)

newId ← 0 initialized list of size N
oldId ← 0 initialized list of size N
id ←0
```
//Reenumerate the rows according to the partitionMap
```
for $r \leftarrow 0$ **to** k **do**
  for $i \leftarrow 0$ **to** N **do**
    if *partitionMap[i]* $= r$ **then**
      newId[id] ← i oldId[i] ← id id ← id + 1
  startIdx[r+1] ← id

newRowPtr ← 0 initialized list of size $N + 1$
newInputVec ← 0 initialized list of size $N + 1$
newColIdx ← 0 initialized list of size nnz
newVals ← 0 initialized list of size nnz

for $i \leftarrow 0$ **to** N **do**
  degree ← rowPtr[i+1] - rowPtr[i]
  newRowPtr[i+1] = newRowPtr[i] + degree
  colIdxStart ← *colIdx + rowPtr[oldId[i]]
  valsStart ← *vals + rowPtr[oldId[i]]

  ```
  //Copy the values and indices to the new arrays
  ```
  copy(colIdxStart, colIdxStart + degree, newColIdx + newRowPtr[i])
  copy(valsStart, valsStart + degree, newVals + newRowPtr[i])

  for $r \leftarrow$ *newRowPtr[i]* **to** *newRowPtr[i+1]* **do**
    newColIdx[r] ← newId[newColIdx[r]]
for $i \leftarrow 0$ **to** N **do**
  newInputVec[i] = inputVec[oldId[i]]
for $i \leftarrow 0$ **to** k **do**
  partition[i] ← (startIndices[i],startIndices[i+1])
rowPtr ← newRowPtr
colIdx ← newColIdx
vals ← newVals
inputVec ← newInputVec

3.4 Load Balance comparison