Lower Triangular Sparse Matrix Kernel

Ibrahim Hasan

September 2020

1 Introduction

We tried optimizing the performance of the sparse triangular solve: $Lx = b$. L is a lower triangular sparse matrix, b is a vector, and x is the vector we attempt to find. All tests were run on an **AMD Ryzen 3600X 6-Core Processors**. We attempted the following:

- Serial sparse-matrix dense-vector solve
- Serial sparse-matrix sparse-vector solve
- Parallel sparse-matrix dense-vector solve
- Parallel sparse-matrix sparse-vector solve

For the sparse matrix we used the `af_0_k101` matrix. For the dense vector we used the `b_dense_af_0_k101` vector and for the sparse vector we used the `b_sparse_af_0_k101` vector.

2 Validation of Results

To test correctness we use the provided serial multiplication code, and assert that the solution vector x that we get was such that $Lx = b$. Because of issues with the stability of double precision values we define equality of two double's as:

```
(Abs(x - y) <= EPSILON * Max(1.0f, Abs(x), Abs(y)))
```

We defined `EPSILON` = 1E9. We got this equality check from here.

3 Implementations and Results

3.1 Serial sparse-matrix dense-vector solve

We used the naive algorithm provided in the starter code to solve this and we got the following run times:

Table 1: Naive serial sparse-matrix dense-vector solve.

Trial	**Time**
	ms
1	19.310
2	17.592
3	18.336
4	18.556
5	17.987

This gives us an average run time of **18.356 ms**.

Based on a Stackoverflow post linked here, we tried making a small optimization by making a copy of our unknown x_j in iteration j and making it a `const`. This is because the compiler is unable to determine that x_j won't be mutated at run time, but by making it a `const` the compiler can now put it in a register and improve access times. With this optimization we get the following run times:

Table 2: Optimized serial sparse-matrix dense-vector solve.

Trial	**Time**
	ms
1	18.063
2	17.964
3	17.772
4	17.966
5	17.831

This gives us an average run time of **17.919 ms**.

As you can see the optimization barely makes any difference. Perhaps I should spend some more time investigating other potential improvements.

3.2 Serial sparse-matrix sparse-vector solve

We once again use the naive algorithm provided in the starter code and get the following run times:

Table 3: Naive serial sparse-matrix sparse-vector solve.

Trial	**Time**
	ms
1	17.780
2	16.396
3	17.532
4	17.525
5	17.707

This gives us an average run time of **17.388 ms**.

This seems somewhat in line with the sparse-matrix dense-vector solve. This is because we don't take advantage of the fact that the right hand side is sparse. Not all columns of L take part in the computation and the jth iteration could potentially be ignored. In [1] there exists an outline of a method that involves building a *Reach* set that consists of j for which x_j won't be 0, and hence we can only consider these. I started implementing this but ran into some issues; I might retry this method if provided with more time.

Based on the structure of the `af_0_k101` matrix this optimization doesn't seem promising. Some analysis on the matrix indicated that all x_j depends on x_{j-1}, and since the `b_sparse_af_0_k101` vector has a non-zero in its first row, the *Reach* set will contain all j anyways.

3.3 Parallel sparse-matrix vector solve

Note that we're combining the implementation and analysis for both the dense and sparse vectors.

To identify a good way to parallelize this code we first inspected some properties of the `af_0_k101` matrix:

Table 4: `af_0_k101` properties

Property	**Value**
numColumns/numRows	503,625
avg nz in Col	17.92
max nz in Col	25

An simple way to parallelize would be the following:

```
lsolve(Lp, Li, Lx, b):
    x = b
    for j in 0..(n - 1):
        x[j] /=  / Lx[Lp[j]]
        #begin parallel
        for p in (Lp[j] + 1) ... (Lp[j + 1] - 1):
            x[Li[p]] -= Lx[p] * x[j]
        #end parallel
```

However as we notice from *Table 4* there are on average only 17 non-zero entries per column, and in the max there are 25 non-zero entries per column, hence in parallel each thread would only have a small amount of computation to do and most probably the overheads of parallelism would out weigh any benefits of the parallelism.

We then attempt to use a level scheduling algorithm as outlined in [2]. The idea is basically that for each (i, j) non-zero entry in L, we have an implicit edge from node j to node i i.e. i depends on j. In this way we can form a level ordering of the nodes and we can compute all the entries of x in a particular level in parallel and process all the levels in a non-decreasing order. The algorithm is something like:

```
lsolve(Lp, Li, Lx, b):
    x = b
    for m in 0 ... numLevels:
        #begin parallel
        for j in levels[m]:
            x[j] /= Lx[Lp[j]]

            for p in (Lp[j] + 1) ... (Lp[j+1] - 1):
                #begin critical section
                x[Li[p]] -= Lx[p] * x[j]
                #end critical section
        #end parallel
```

Doing some quick analysis on the level ordering we get the following insights:

Table 5: `level ordering` properties

Property	**Value**
maxLevel	13529
max nodes in a level	85
min nodes in a level	1

Another thing we noted is that half the levels are empty (levels are essentially the number of dependencies a node has, and all nodes seem to have an odd number of dependencies, so the even numbered levels don't have any nodes). On average a level should have 74 nodes. This provides good potential for parallelizing the computation on the nodes per level. Based on some experimentation the best scheduling seemed to be dynamic with a chunk size of 4. These are our results for the sparse matrix (note it does not include the time to build the level set):

Table 6: parallel sparse-matrix sparse-vector solve.

Trial	**Time**
	ms
1	27.256
2	27.124
3	27.607
4	27.681
5	27.346

This gives us an average run time of **27.402 ms**.

As is obvious this is a significant slow down from the serial version. One potential reason for this could be that when processing the columns in level order you are no longer necessarily processing columns in a contiguous manner, and hence you lose benefits of **spacial locality**. It may be worthwhile to investigate methods of rearranging the order in which the columns are stored to continue to exploit spacial locality even in the level order ordering. Furthermore you need to pay the overhead of the critical section around the concurrent updates to x.

References

[1] Sivasankran Rajamanickam Timothy A.Davis and Wissam M. Sid-Lakhdar. *A survey of direct methods for sparse linear systems.*

[2] Ruipeng Li. On parallel solution of sparse triangular linear systems in cuda, 2017.