COL380 Assignment 4

Aniruddha Deb
2020CS10869

April 2023

1 Approach

Algorithm 1 CUDA Sparse Matrix Computation

```
1: procedure $\text{CUDAMULT}(A, B, C, C_v)$
2:     $bx, by \leftarrow$ Block indexes
3:     $i, j \leftarrow 0, 0$
4:     while $i < A.nidxs[bx]$ do
5:         while $j < B.nidxs[by]$ and $A.idxs[bx][i] \neq B.idxs[by][j]$ do
6:             $j \leftarrow j + 1$
7:         end while
8:         if $j < B.nidxs[by]$ then
9:             break
10:        end if
11:        $C[bx][by] \leftarrow A[bx][i] \times B[i][by]$
12:        $C_v[bx][by] \leftarrow 1$
13:        $i \leftarrow i + 1$
14:    end while
15: end procedure

16: $A \leftarrow$ BCSR matrix read from file
17: $B \leftarrow$ BCSC matrix read from file
18: $n, m, p \leftarrow A.n, A.m, A.n/A.m$

19: $C \leftarrow$ Empty $n \times n$ matrix
20: $C_v \leftarrow$ Empty $p \times p$ bitmap

21: Launch $\text{CUDAMULT}(A,B,C,C_v)$ with $p \times p$ blocks and $m \times m$ threads per block

22: $C_{idx} \leftarrow$ Prefix sum of the row sums of $C_v$
23: Compress $C$ such that all the blocks of $C$ are shifted to the start of their corresponding rows
24: Compress the rows of $C$ into an array on the host, indexed by $C_{idx}$
25: Save $C$ to file.
```

The algorithm that is used reads in two matrices, the first in Block Compressed Sparse Row format and the second in Block Compressed Sparse Column format, and computes the product on the GPU in dense format. After this, the dense matrix is compressed to BCSR on the GPU and sent back to the CPU, where it is saved to file.

2 Results

Six different matrices were tested, with varying values of n and m. The smaller versions were used for debugging/correctness analysis, and the larger versions were used for benchmarking and performance. The times for the larger matrices are given below:

n	m	k_{in}	k_{out}	time
32768	8	16800	68517	1.140 s
32768	4	65300	518873	3.387 s

3 Analysis

The multiplication that we do may be suboptimal if the block sizes are too small and the matrix is too dense. In this case, the cost of scheduling the block, loading the matrix from memory and saving the block to memory may overshadow the cost associated with multiplying the numbers themselves. In this case, we can merge blocks together to increase the block size to ensure better utilization of the GPU. rather than using the given block size that is m, we use a block size of um. For example, if $m = 4$, using $u = 8$ would give us a block size of 1024, which is the maximum number of threads in a block. This ensures much better thread utilization, *Only if we can guarantee that the elements in this expanded block are dense enough to ensure that we multiply them out*. Essentially, we're looking for a guarantee for a certain degree of denseness. That is, if f is above some value, we should expand the blocks and if it is not, we should use the current block size.

Assuming that the blocks in the matrices are sampled uniformly randomly, The probability that a block in the matrix is filled is k/p^2. If $k = fp^2, f \in [0, 1]$, then this probability is f (the fraction of filled blocks in the matrix)

Assuming that A and B have the same block filling fraction f, The expected number of block multiplications that we would need to perform for a single element in $C = AB$ would be

$$\begin{aligned}\mathbb{E} &= \sum_{i=0}^{p} i \binom{p}{i} \left(f^2\right)^i \left(1 - f^2\right)^{p-i} \\ &= pf^2\end{aligned}$$

Therefore, the expected number of block multiplications that we need to do for the complete matrix C would be $\mathbb{E}_1 = p^2\mathbb{E} = p^3 f^2$.

Consider block expansion now. We define a *superblock* as an aggregated $um \times um$ block . We'd have $p/u \times p/u$ superblocks in the matrix, and the probability that such a superblock is empty is $(1 - f)^{u^2}$. The probability that it is nonempty, is therefore $f' = (1 - (1 - f)^{u^2})$. The expected number of nonempty superblocks in the final output would then be

$$\mathbb{E}_u = \frac{p^3}{u^3} \times f'^2 = p^3 \times \frac{(1 - (1 - f)^{u^2})^2}{u^3}$$

It is thus beneficial to use superblocks when $\mathbb{E}_u < \mathbb{E}_1$. Computing the solution numerically, this occurs when $f \gtrapprox 0.04$ for $u = 8$ and $f \gtrapprox 0.10$ for $u = 4$.

Unfortunately, this improvement was not implemented after the clarification on Piazza that the matrices would be sparse (that is, $k \sim O(n/m)$). Hence, f will be less than the given bounds always