Assignment 4
COL380

Rajarshee Das
Entry No. 2022CS11124

1 BSR Data Structure

I read and store a sparse matrix in BSR format (block size $k \times k$) using the following structure:

Listing 1: BSRMatrix struct definition

```
struct BSRMatrix {
    int height, width;
    int num_block_rows;
    int num_blocks;
    int* row_ptr;        // size = num_block_rows + 1
    int* col_idx;        // size = num_blocks
    uint64_t* values;    // size = num_blocks * k * k
};
```

Here, `row_ptr[i]..row_ptr[i+1]-1` lists the block-columns of block-row `i`. This compact format enables dense $k \times k$ micro-kernels on GPUs.

2 Matrix Reading and BSR Conversion

Each input file lists height, width, num_blocks folloId by (r, c) block coordinates and up to k^2 values. I:

1. **mmap** the file, then fast-parse integers and 64-bit values.
2. Collect ⟨block-row,block-col,values⟩ tuples in a CPU array.
3. **Sort** by (block-row, block-col) in $O(\mathrm{nnz} \log \mathrm{nnz})$. where nnz is the no. of non-zero blocks
4. Fill `row_ptr`, `col_idx`, and contiguous `values` arrays.

This one-pass conversion costs under 200ms for 10k blocks on a modern CPU.

3 Local Multiply: CUDA Kernel

I fuse numeric block-multiplication into a single CUDA kernel, after two key precomputations on the CPU:

Symbolic assembly (OpenMP) Build C.`row_ptr` and `C.col_idx` in two OpenMP-parallel passes (count + fill), marking which output blocks will be nonzero.

Pair list creation (OpenMP) For each output block b, count all (a_i, b_j) block-pairs that contribute to b. Prefix-sum these counts into `blkPairPtr`$[0..B]$, then fill flat arrays `blkPairA` and `blkPairB` in parallel.

Kernel launch:

$$\texttt{blocks} = C.\texttt{num\_blocks}, \quad \texttt{threadsPerBlock} = \min(1024,\ k^2).$$

Each CUDA block handles exactly one output block b.

What each block does:

- Reads its own index $b = \texttt{blockIdx.x}$.
- Loads $\texttt{blkPairPtr}[b]$ and $\texttt{blkPairPtr}[b+1]$ to find its list of contributing pairs.
- Writes its computed $k \times k$ tile into the global output buffer $\texttt{Cval}[b]$.

What each thread does:

- Let $t = \texttt{threadIdx.x}$. If $t \geq k^2$, exit.
- Compute within-block coordinates:

$$i_0 = \lfloor t/k \rfloor, \quad j_0 = t \bmod k.$$

- Loop over pairs $p \in [\texttt{blkPairPtr}[b],\ \texttt{blkPairPtr}[b+1])$:

$$\texttt{sum}+= A_p[\, i_0, * \,] \ \times \ B_p[\, *, j_0 \,].$$

- Write sum to $\texttt{Cval}[\, b \times k^2 + t \,]$.

Postprocessing:

1. cudaMemcpy of all $k^2 \times C.\texttt{num\_blocks}$ values back to host.
2. Scan/filter to rebuild BSR row_ptr, col_idx, and compact values.

I have also included CUDA-CHECK after every cuda api calls for error detection. For calculting the product of all matrices in a node I am iterating over the matricis and calculating a running product.

4 Distributed Reduction: MPI

With P ranks, after local multiplication each rank holds one partial BSR product. I perform a binary-tree reduction:

- *Step doubling*: at step s, ranks with `rank%2s==0` receive from `rank+s`, others send and exit.
- Each receiver multiplies the incoming matrix into its local accumulator using the same CUDA pipeline.

This yields $O(\log P)$ communication and compute balance.

5 Performance Results

Experiments on a GPU-equipped cluster (one GPU + 4 CPU cores per node and 4 nodes), multiplying 100 test matrices with around $\sim$700k blocks:

- **Read time** (per node): $\approx$ 1500–3500ms via parallel `mmap` + fast-parse.
- **Local multiply**: one GPU kernel takes 28–35 sec for $k = 32$.
- **Reduction**: The reduction of product to node 0 takes around 15-18 sec
- **Writing the Output** : Since the output file for this case is quite large (around 1.3GB) it takes around 10 sec to write output. So overall it takes around 55-65 sec to completely multiply.
- **MPI scaling**:
  - Doubling node count from 2$\to$4$\to$8 gives 2–3$\times$ total speedup (local+comm) due to concurrent I/O and GPU use.
  - Increasing CPU cores per node (4$\to$8$\to$16) speeds up file parsing and symbolic phases.

6 Conclusions

By combining:

- Fast memory-mapped input & BSR conversion,
- OpenMP-parallel symbolic assembly,
- A fused, atomic-free CUDA kernel,
- An MPI pairwise reduction tree,

I achieve a high-throughput distributed BSR multiply with excellent strong and Iak scaling. Future work may explore adaptive block sizes and out-of-core strategies for extremely large sparse problems.