COL380 A4 Report

Koduru Suchith (2021CS10572)

1 Implementation Strategy

The goal of the assignment was to compute the product of a sequence of N sparse matrices using MPI and CUDA in a distributed parallel environment. The matrices are stored as sparse blocks of size $k \times k$.

My approach is summarized as follows:

- Each process calculates the **range of matrix indices** it is responsible for based on its `rank` and the total number of MPI processes.
- Each process reads its corresponding matrix files into memory. This ensures fully parallel I/O with no contention.
- Each process performs a **sequential multiplication of its assigned matrices**, where each multiplication is done using **CUDA** on the GPU.
- Once each process has its local result, we perform a **tree-based reduction using MPI** to compute the global result.
- At each reduction step, the matrices are serialized and transmitted. The deserialized result is again multiplied on GPU with the receiver's local matrix.
- The final result matrix is written to file by the process with `rank = 0`.

This approach effectively leverages distributed and device-level parallelism while minimizing communication overhead.

2 Parallelization Details

MPI Parallelization

MPI handles distributed computation:

- Matrix ranges are partitioned across processes as evenly as possible.
- Each process multiplies its local matrix sequence independently.
- A **logarithmic tree reduction** is used to gather and merge the partial results.
- Matrix results are serialized into buffers for MPI communication.

CUDA Acceleration

All matrix multiplications are offloaded to the GPU:

- Each sparse matrix is flattened into coordinate and value arrays and sent to device memory.
- A custom kernel computes block-wise multiplication using multiple threads for each block pair.
- Output blocks are written to device memory and copied back after computation.
- **Correct accumulation** is done on the host side in case multiple block-pairs contribute to the same output block index.
- All arithmetic is performed modulo $2^{63} - 1$ to avoid overflow.

3 Performance

- **I/O Efficiency**: Each process reads only what is necessary. No shared file access.
- **GPU Usage**: All computations (block-wise and matrix-wise multiplications) are GPU-accelerated for maximum throughput.
- **Tree-Based Merging**: The reduction step takes $\mathcal{O}(\log p)$ communication rounds, where p is the number of processes.
- **Modular Arithmetic**: Modular constraints ensure arithmetic correctness without overflow, even during accumulation.
- **Deterministic Output**: Proper accumulation guarantees that the same input yields the same output, despite parallelism and kernel reordering.