COMPUTER ARCHITECTURE
Assignment-2
LU-Factorization in GPU

Author: Dhwanish
Roll Number: B211266CS

INTRODUCTION

a. Parallel Algorithm Used to Implement the Solution

The parallel algorithm implemented in this solution is based on LU decomposition using Gaussian elimination. LU decomposition breaks a matrix A into a lower triangular matrix L and an upper triangular matrix U, such that A=LU.

The algorithm proceeds in two main steps:

- **Scaling Kernel(computeL)**: For each pivot element in matrix U, the kernel computes the scaling factors for the rows below the pivot, which is used to form matrix L.
- **Row Elimination Kernel(rowElimination)** : Using the scaling factors, the kernel eliminates elements below the pivot row, updating U by subtracting multiples of the pivot row from subsequent rows. This transforms U into an upper triangular form.

In terms of parallelism, the algorithm parallelizes the elimination step where different threads handle different rows of the matrix. The scale step is parallelized across the elements in a column.

b. Kernel Configuration (Grid and Block Sizes)

```
for (int i = 0; i < N; ++i) {
    cudaEventRecord(startLU);

    computeL<<<1, N - i - 1>>>(d_U, d_L, N, i);

    cudaEventRecord(stopLU);
    cudaEventSynchronize(stopLU);
    float l1_time = 0;
    cudaEventElapsedTime(&l1_time,startLU, stopLU);
    l_time += std::chrono::duration<double>(l1_time/1000);

    int sharedMem = N * sizeof(double);
    cudaDeviceSynchronize();
    cudaEventRecord(startLU);

    dim3 gridConfig(((N)/TILE) + ((N%TILE) ? 1 : 0), 1, 1);
    dim3 blockConfig(TILE, 1, 1);
    rowElimination<<<gridConfig, blockConfig, sharedMem>>>(d_L, d_U, N, i);

    cudaEventRecord(stopLU);
    cudaEventSynchronize(stopLU);
    float u1_time = 0;
    cudaEventElapsedTime(&u1_time,startLU, stopLU);
    u_time += std::chrono::duration<double>(u1_time/1000);
}
```

- **Scaling Kernel (computeL)**:
    - Block size: N-i-1(i start from 0 till N) threads per block, where N is the size of the matrix. This means each thread operates on a specific element of the matrix to compute the scale factor.
    - Grid size: A single block <<<1,N-i-1>>> since scaling each row of the matrix only requires one-dimensional parallelization over the elements of the column.
- **Elimination Kernel (rowElimination)**:
    - Block size: TILE=8 threads per block. Each thread is responsible for one row of the matrix.
    - Grid size: ((N/TILE) + (N%TILE) ? 1 : 0) blocks.This ensures that each row in the matrix has a thread responsible for it.

The configuration uses two-dimensional parallelism in the elimination kernel, where blocks operate on chunks of rows, and threads within a block handle different elements of a row.

c. CGMA (Compute-to-Global-Memory Access) Ratio of Each Kernel

The **Compute-to-Global-Memory Access (CGMA)** ratio is a measure of how many arithmetic operations (computes) are performed for each memory access (global memory access). High CGMA indicates good compute efficiency, while low CGMA suggests memory-bound performance.

Scale Kernel (computeL):

```
__global__ void computeL(double* U, double *L, int N, int index) {
    int id = index + threadIdx.x + 1;
    int start = (index * N + index);

    __shared__ double pivot;
    pivot = U[start];
    if (id < N) {
        L[id * N + index] = U[id * N + index] / pivot;
    }
}
```

- **Memory accesses**:
    - Reading one element from the diagonal of U (once per row) is shared which leads to N/TILE * 1 memory accesses.
    - Reading one element from U and writing one element to L which leads to 2 * N total memory access (2 per thread).
- **Computation**: Each thread performs one division operation => N computations.
- **CGMA Ratio**:

CGMA= N / (2*N + (N/TILE))

= 1 / (2 + (1/TILE))

Elimination Kernel (rowElimination):

```
// Row elimination Kernel
__global__ void rowElimination(double* L, double* U, int N, int index) {
    int pivotRow = index * N;
    int currentRow = (blockDim.x * blockIdx.x + threadIdx.x) * N;
    int start = currentRow + index;
    int end = currentRow + N;

    extern __shared__ double Us[];
    __shared__ double pivot;
    pivot = L[start];
    for (int i = 0; i < N; i++) {
        Us[i] = U[pivotRow + i];
    }

    if (currentRow > pivotRow && currentRow < N * N) {
        for (int i = start + 1; i < end; ++i) {
            U[i] = U[i] - (pivot * Us[i - currentRow]);
        }
    }
}
```

- **Memory accesses**:
  - Each thread accesses the pivot row of U which is shared => (N/TILE) * N memory access
  - Updating the row values of U results in 2*N memory operations per thread per iteration => 2*N*N total memory accesses
- **Computation**: Each thread performs 2 * N computations => 2*N*N total computations
- **CGMA Ratio**:

  CGMA = (2*N*N)/(2*N*N + ((N*N)/TILE))

  =2 / (2 + (1/TILE))

d. Synchronizations Used in the Solution and Their Impact on Performance

1. **Kernel Synchronization (Implicit)**:
   - Between the computeL and rowElimination kernels, there is an implicit synchronization because CUDA kernels execute sequentially unless explicitly defined otherwise. This ensures that the scaling factors are computed before the elimination process begins.
   - **Impact**: While necessary to ensure correctness, this introduces overhead as the GPU has to wait for one kernel to complete before launching the next. This limits the parallelism that could be exploited between scaling and elimination.
2. **`cudaDeviceSynchronize()`**:
   - This is explicitly called after each kernel invocation, ensuring that all threads complete before proceeding to the next step.

- **Impact**: These synchronization points add overhead to the execution time as the CPU waits for the GPU to finish, but they are essential for the correct order of operations in LU decomposition.

3. **CUDA Stream Synchronization (`cudaEventSynchronize`)**:
    - Used to measure the elapsed time between kernel executions, ensuring that timing is accurate and preventing the CPU from proceeding until all GPU operations are completed.
    - **Impact**: This synchronization is required for performance measurement but adds no additional performance overhead unless events are overused.