VIVEK K P
B210473CS

Implement an efficient CUDA program to solve a system of linear equations using LU factorization method. Assume that the system is represented as "AX = B" matrices.

1 Parallel Algorithm Implementation for LU Decomposition

1.1 a. Parallel Algorithm Used to Implement the Solution

The parallel algorithm implemented in the code is the LU decomposition of a matrix A into its lower triangular matrix L and upper triangular matrix U.

1. **LU Decomposition**:
   - The `luop` kernel performs the actual LU decomposition in parallel.
   - It computes elements of U and L using the formulas for forward and backward substitutions.
   - It utilizes CUDA threads to perform calculations for different elements of the matrices L and U concurrently.
   - **Doolittle Method**: This method computes the LU decomposition of a matrix A such that $A = LU$, where L is a lower triangular matrix and U is an upper triangular matrix.
     - The algorithm starts with L as the identity matrix and iteratively updates L and U as follows:
       * For each row k, the diagonal entry of U is computed as:

$$U_{kk} = A_{kk} - \sum_{j=1}^{k-1} L_{kj} U_{jk}$$

       * The entries of L below the diagonal are computed using:

$$L_{ik} = \frac{1}{U_{kk}} \left(A_{ik} - \sum_{j=1}^{k-1} L_{ij} U_{jk} \right) \quad \text{for } i > k$$

   - **Forward and Backward Substitution**:
     - The `linearfind` function performs forward substitution to compute an intermediate solution and then backward substitution to find the final solution of the linear system $Ax = b$.

1.2 b. Kernel Configuration (Grid and Block Sizes) for Each Kernel Called

(a) `initialize` Kernel:

- **Grid Size**:

$$\text{dim3 grid} = \text{dim3} \left(\lceil \frac{\text{size}}{(double)\text{BLOCK\_SIZE}} \rceil, \lceil \frac{\text{size}}{(double)\text{BLOCK\_SIZE}} \rceil \right);$$

- **Block Size**:

$$\text{dim3 block} = \text{dim3}(\text{BLOCK\_SIZE}, \text{BLOCK\_SIZE});$$

- This configuration allows the kernel to launch a 2D grid of threads to initialize matrices L and U to zeros, setting the diagonal of L to ones.

(b) `luop` Kernel:

- **Grid Size**:
$$\text{int grid1} = \lceil \frac{\text{size}}{\text{block1}} \rceil;$$
where `block1` = `hfact(size, 128);`.
- **Block Size**: `block1`, determined by the highest factor of `size` that is less than or equal to 128.
- This kernel is called in a loop for each row of the matrices, performing the LU decomposition concurrently across the size of the matrix.

1.3 c. CGMA Value of Each Kernel Called

Calculating the **Computational Gains per Memory Access (CGMA)** for each kernel involves estimating the number of computations performed in relation to the number of memory accesses. Here's how we can analyze the CGMA for each kernel:

1.3.1 1. `initialize` Kernel

Functionality: This kernel initializes the matrices L and U.

- **Memory Accesses**:
  - Each thread accesses memory to read/write to matL and matU.
  - For every thread, it writes to both matL and matU.
  - For a matrix of size n, there are n^2 total threads and therefore $2n^2$ memory accesses (2 for each thread).
- **Computations**:
  - Each thread performs a small constant number of computations (setting values).
  - Specifically:
    - 1 comparison for checking if $i == j$.
    - A few assignments.
  - Therefore, for each thread, we can consider 1 or 2 computations, leading to approximately n^2 computations.
- **CGMA Calculation**:
$$\text{CGMA} = \frac{\text{Total Computations}}{\text{Total Memory Accesses}} = \frac{n^2}{2n^2} = \frac{1}{2}$$

1.3.2 2. `luop` Kernel

Functionality: This kernel performs the actual LU decomposition of the matrix A.

- **Memory Accesses**:
  - Each thread accesses the matrices L, U, and A.
  - For each row curr:
    - Each thread accesses matA once, then matU and matL in a loop for calculations. The number of accesses can be approximated as:
      - 1 access for reading matA.

        - Up to i accesses for reading matU (in the loop).
        - Up to j accesses for reading matL (in the loop).
    - The worst case for each thread accessing in the inner loop is $O(n)$.
    - Overall, for n rows and each thread processing potentially n accesses, we can assume $O(n^3)$ total memory accesses in the worst case.
- **Computations**:
    - Each thread performs several arithmetic operations (subtractions, multiplications).
    - For a full decomposition:
        - Each element of U involves operations with up to n elements.
        - Each element of L similarly involves up to n operations.
    - Therefore, in total, each thread performs $O(n^2)$ computations (each row processing up to n elements).
- **CGMA Calculation**:

$$\text{CGMA} = \frac{\text{Total Computations}}{\text{Total Memory Accesses}} = \frac{O(n^2)}{O(n^3)} = O(n^{-1}) \approx \frac{1}{n}$$

1.4 Summary of CGMA Values

- `initialize` Kernel: CGMA $= \frac{1}{2}$
- `luop` Kernel: CGMA $= O(n^{-1}) \approx \frac{1}{n}$

1.5 d. Discuss Different Types of Synchronizations Used in the Solution and Their Impact on Performance

(a) `cudaDeviceSynchronize()`:

- **Purpose**: This function is called after each kernel launch (`luop`) to ensure that the host waits for the completion of the device operations before proceeding.
- **Impact**: While necessary for correctness (to ensure the results are available before the next iteration or subsequent computations), it can lead to performance bottlenecks if overused. In high-performance applications, minimizing these synchronizations is crucial to achieve maximum throughput.

(b) Atomic Operations:

- The code uses `atomicAdd` to update timing variables for the upper and lower triangular calculations.
- **Purpose**: These atomic operations ensure that simultaneous writes from multiple threads to shared memory do not lead to race conditions.
- **Impact**: Atomic operations introduce serialization, which can significantly slow down performance, especially if there are high levels of contention among threads trying to write to the same memory location. They can introduce significant latency due to data transfer times, especially with large matrices. Optimizing memory transfer (e.g., using asynchronous copies, overlapping computation with communication) is essential for performance.