CUDA LU Factorization Solver

Praval Pattam
B220057CS

1. Implementation Overview

The implementation solves systems of linear equations **AX = B** using LU factorization with partial pivoting on the GPU. The algorithm decomposes matrix **A** into a lower triangular matrix **L** and an upper triangular matrix **U**, then solves the system through forward and backward substitution.

Key Features:

- Parallel LU factorization using CUDA kernels
- Partial pivoting for numerical stability
- Double precision floating-point arithmetic
- Shared memory optimization
- Comprehensive timing measurements

---

2. Parallel Algorithm Explanation

The solution is based on a parallel LU decomposition algorithm with partial pivoting. The decomposition proceeds column by column, with each stage parallelized where possible:

1. **Pivot Finding:** A parallel reduction is used to identify the maximum element in the current column, ensuring numerical stability.
    - Purpose: Find row with maximum absolute value in column k
    - Method: Parallel reduction using shared memory
    - Kernel: findPivotKernel
    - Threads: (N-k) threads in parallel
    - Algorithm:
        1. Each thread loads one element from column k
        2. Store absolute values and indices in shared memory

3. Perform reduction using binary tree pattern
4. Thread 0 writes pivot index to global memory

- Why: Ensures numerical stability and handles zero diagonal elements

2. **Row Swapping:** Once the pivot row is identified, a parallel row exchange is performed across all columns.
    - Purpose: Exchange pivot row with current row
    - Method: Parallel column-wise swapping
    - Kernel: swapRowsKernel
    - Threads: N threads (one per column)
    - Algorithm:
        1. Each thread handles one column
        2. Swap elements in both A matrix and B vector
        3. Also swap already-computed L matrix entries if k > 0
    - Why: Brings largest element to diagonal position
3. **Multiplier Computation:** The multipliers used to eliminate entries below the pivot are computed in parallel, filling the **L** matrix.
    - Purpose: Calculate L matrix entries for column k
    - Method: Parallel computation with zero-check
    - Kernel: computeMultipliersKernel
    - Threads: (N-k-1) threads in parallel
    - Algorithm:
        1. Each thread computes one multiplier: L[i,k] = A[i,k] / A[k,k]
        2. Check if pivot is too small (< 1e-10) and set to 0 if so
        3. Write directly to L matrix
    - Why: These multipliers are used to eliminate subdiagonal elements

4. **Submatrix Update:** The trailing submatrix is updated in parallel using a 2D grid of threads with shared memory caching to reduce global memory traffic.
    - Purpose: Update remaining matrix: A[i,j] -= L[i,k] × A[k,j]
    - Method: 2D parallel threads with shared memory caching
    - Kernel: updateSubmatrixKernel
    - Threads: $(N-k-1)^2$ threads in 16×16 blocks
    - Algorithm:
        1. Load A[k,:] (row k) into shared memory
        2. Each thread computes: A[i,j] -= L[i,k] × A[k,j]
        3. Shared memory reduces redundant global memory accesses
    - Why: Most computationally intensive part; shared memory critical for CGMA
5. **Forward Substitution:** Once factorization is complete, the system **Ly = Pb** is solved sequentially.
    - Purpose: Solve Ly = Pb for intermediate vector y
    - Method: Sequential iteration (data dependencies)
    - Kernel: forwardSubstitutionKernel
    - Threads: 1 thread per iteration, N iterations
    - Algorithm:

      For i = 0 to N-1:

      y[i] = B[i] - Σ(L[i,j] × y[j]) for j = 0 to i-1
    - Why: Strong data dependencies prevent parallelization
6. **Backward Substitution:** Finally, the system **Ux = y** is solved sequentially to obtain the solution vector.
    - Purpose: Solve Ux = y for solution vector x
    - Method: Sequential iteration (data dependencies)
    - Kernel: backwardSubstitutionKernel

- Threads: 1 thread per iteration, N iterations
- Algorithm:
- For i = N-1 down to 0:
- x[i] = (y[i] - Σ(U[i,j] × x[j])) / U[i,i] for j = i+1 to N-1
- Why: Strong data dependencies prevent parallelization

This design balances parallelism in the computationally heavy stages (pivoting, multipliers, submatrix updates) with sequential steps where dependencies prevent parallel execution.

3. Kernel Configuration Details

The kernel launch configurations were carefully chosen to balance occupancy, memory access, and synchronization requirements.

Kernel	**Block Size**	**Grid Size**	**Launch Frequency**
findPivotKernel	256 threads	⌈(N–k)/256⌉	N–1 times
swapRowsKernel*	256 threads	⌈N/256⌉	When pivoting needed
computeMultipliersKernel	256 threads	⌈(N–k–1)/256⌉	N–1 times
updateSubmatrixKernel	16×16 (256)	2D Grid**	N–1 times
forwardSubstitutionKernel	1 thread	1 block	N times
backwardSubstitutionKernel	1 thread	1 block	N times

* swapRowsKernel: Only called when pivot row ≠ k (approximately 50% of time)

** 2D Grid: ⌈(N-k-1)/16⌉ × ⌈(N-k-1)/16⌉ blocks

The **updateSubmatrixKernel** dominates runtime, and its 2D grid configuration ensures efficient coverage of the trailing submatrix with shared memory reuse.

Detailed Configuration By Matrix Size

N = 50:

- findPivotKernel: Up to 1 block, 50 threads
- updateSubmatrixKernel: Up to 4×4 = 16 blocks, 2,401 peak threads
- Total Thread Invocations: 44,249

N = 100:

- findPivotKernel: Up to 1 block, 100 threads
- updateSubmatrixKernel: Up to 7×7 = 49 blocks, 9,801 peak threads
- Total Thread Invocations: 343,499

N = 200:

- findPivotKernel: Up to 1 block, 200 threads
- updateSubmatrixKernel: Up to 13×13 = 169 blocks, 39,601 peak threads
- Total Thread Invocations: 2,706,999

N = 500:

- findPivotKernel: Up to 2 blocks, 500 threads
- updateSubmatrixKernel: Up to 32×32 = 1,024 blocks, 249,001 peak threads
- Total Thread Invocations: 41,917,499

4. CGMA Analysis

The **Compute-to-Global-Memory-Access (CGMA)** ratio measures computational intensity. Higher CGMA indicates the kernel is more compute-bound, while lower CGMA indicates memory-bound behavior.

Matrix Size	CGMA Ratio	Performance Characteristic
50	7.01	Moderate, partially memory-bound
100	13.68	Balanced
200	27.01	Compute-intensive
500	67.01	Highly compute-intensive

Key Observations:

- CGMA scales approximately linearly with **N** (~0.133N).
- Shared memory optimization improves CGMA by a factor of ~3.
- The **updateSubmatrixKernel** achieves a CGMA of 1.0–1.5 after optimization, making it the dominant performance driver.

5. Synchronization Analysis

Synchronization is critical for correctness but introduces overhead. Two types are used:

1. **Device Synchronization (cudaDeviceSynchronize):**
    - Used after each kernel call to enforce sequential dependencies.
    - Overhead: 5–15% of total execution time.
    - Essential for correctness, especially between pivoting and submatrix updates.
2. **Block Synchronization (__syncthreads):**
    - Used within kernels (e.g., parallel reduction for pivot finding).
    - Overhead: minimal (~1–2 clock cycles).
    - Ensures safe shared memory access.

3. N Total Sync Calls Estimated Overhead (ms) % of Total Time

50	~200	4–10	3–8%
100	~400	8–20	3–8%
200	~800	16–40	3–8%
500	~2,000	40–100	5–15%

This table highlights how synchronization costs scale with problem size. While the absolute overhead increases with **N**, the percentage of total runtime remains modest, demonstrating that the algorithm maintains good efficiency even at larger scales.

Performance Impact:

- Synchronization overhead grows with matrix size. For **N=500**, synchronization costs ~40–100 ms.
- Despite overhead, synchronization is a necessary trade-off to guarantee correctness in parallel execution.

6. Performance Results

Timing Measurements

Matrix Size	Read Time (s)	LU Time (s)	Solve Time (s)	Total Time (s)
50	**0.000456**	**0.008786**	**0.012502**	**0.012958**
80	**0.001201**	**0.014001**	**0.019390**	**0.020591**
100	**0.001815**	**0.017361**	**0.024435**	**0.026250**
200	**0.007178**	**0.036077**	**0.051959**	**0.059137**
500	**0.045743**	**0.086512**	**0.136516**	**0.182259**

CGMA and Thread Utilization

Matrix Size	CGMA Ratio	Total Threads	Peak Threads
50	7.01	44,249	2,401
80	11.01	177,199	6,241
100	13.68	343,499	9,801
200	27.01	2,706,999	39,601
500	67.01	41,917,499	249,001

Timing Breakdown Analysis (for N=500)

- I/O Time: 25.1% (File reading – CPU bound)
- LU Computation: 47.5% (Main parallel computation)
- Substitution: 27.4% (Sequential solve phase)

Observations:

1. I/O time grows quadratically with N (as expected for N^2 elements).
2. LU time scales sub-cubically due to parallelization.
3. Substitution time is relatively small but remains sequential.