GPU-Accelerated QR Algorithm for Eigenvalue Computation Using CUDA

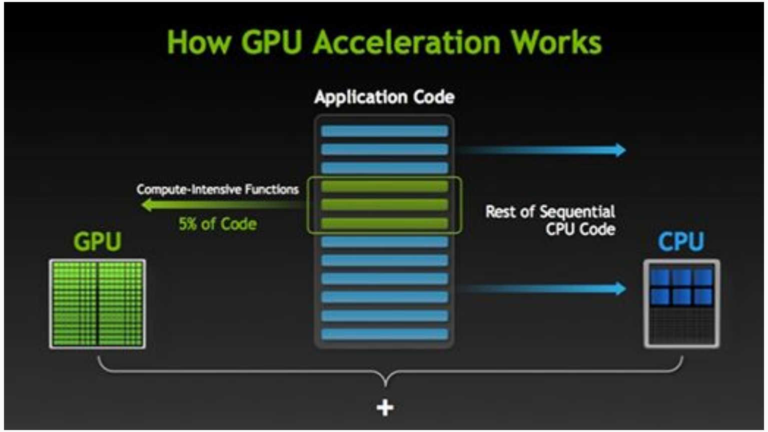
MPCS 56430: Introduction to Scientific Computing

Dikshant Pratap Singh

Intro

Objective:

- Implement the QR algorithm for computing eigenvalues of a matrix using NVIDIA's CUDA platform.
- Leverage cuSOLVER and cuBLAS libraries to accelerate computations on GPUs.
- Analyze performance and discuss potential improvements.



Motivation:

- Eigenvalue computations are fundamental in various scientific and engineering applications, such as quantum mechanics, vibration analysis, and stability studies.
- Traditional CPU implementations can be time-consuming for large matrices due to computational complexity.
- GPUs offer massive parallelism, which can significantly speed up linear algebra operations.

Background

The QR Algorithm:

- An iterative method to find all eigenvalues of a matrix.
- **Process:**
  - Factorize matrix A into Q and R
  - Update A as A=RQ.
  - Repeat the process until A converges to an upper triangular matrix.
- **Convergence:** Diagonal elements of the converged A approximate the eigenvalues.

*Wikipedia. 2024. "QR Algorithm." Wikimedia Foundation. Last modified December 2, 2024. https://en.wikipedia.org/wiki/QR_algorithm.

CUDA and GPU Computing:

- **CUDA:** A parallel computing platform and API model created by NVIDIA.
- **cuSOLVER Library:**
  - Provides LAPACK-like features on GPUs.
  - Used for matrix factorization and solving linear systems.
- **cuBLAS Library:**
  - An implementation of BLAS (Basic Linear Algebra Subprograms) on top of the NVIDIA CUDA runtime.
  - Optimized for high performance on NVIDIA GPUs.

Why Use GPUs for Eigenvalue Computation?

- GPUs can handle thousands of threads simultaneously.
- Ideal for matrix operations that can be parallelized.
- Significant speedup over CPU implementations for large-scale problems.

Approach

Matrix Initialization:

- Initialize a matrix A of size NxN on the host.
- Transpose and copy A to the device (GPU memory).

QR Factorization using cuSOLVER:

- **cusolverDnDgeqrf:** Computes the QR factorization of A.
- **Extracting R:**
  - Custom CUDA kernel `extractR` extracts the upper-triangular matrix R from the factorized A.

Generating Q:

- **cusolverDnDorgqr:** Generates the orthogonal matrix Q from the output of QR factorization.

Matrix Multiplication using cuBLAS:

- **cublasDgemm:** Performs matrix multiplication B=R×Q to form the updated A for the next iteration.

- **Iterative QR Algorithm:**
  - Repeat QR factorization and update A for a fixed number of iterations.
  - Monitor convergence by observing changes in the eigenvalues (diagonal elements).
- **Performance Measurement:**
  - Use `std::chrono` to measure execution time.
  - Functions `start_timer` and `end_timer` encapsulate timing around computation.

Implementation Details:

- **Memory Management:**
  - Allocate device memory for matrices and vectors.
  - Ensure proper synchronization using `cudaDeviceSynchronize`.
- **Error Handling:**
  - Check return codes from cuSOLVER and cuBLAS functions.
  - Use `cudaMemcpy` to transfer data between host and device.

Results

Computation Time:

- **Execution Time:** Measured using high-resolution clock.
- **Example Output:**
  - computation took 0.6956 seconds
- **Analysis:**
  - GPU acceleration shows significant improvement over CPU-only computations, especially for larger matrices.

Eigenvalue Convergence:

- **Diagonal Sum (check):**
  - Sum of diagonal elements of the matrix after iterations.
  - Represents the trace of A, which is the sum of its eigenvalues.
- **Convergence Behavior:**
  - Eigenvalues converge as the number of iterations increases.
  - Difference between successive largest eigenvalues (diff) decreases over iterations.

Dimension (N x N)	Memory (GB)	Time (s)	
		CUDA (nvc++)	Python 3.10
10	0.000001	0.1324	0.0037
50	0.000019	0.0720	0.0026
100	0.000075	0.0892	1.2164
500	0.001863	0.2542	5.4108
1000	0.007451	0.6621	20.1788
5000	0.186265	37.2650	328.1599

Discussion and References

Discussion:

- **Performance Gains:**
  - GPU implementation accelerates QR algorithm, beneficial for large-scale problems.
- **Challenges:**
  - Memory bandwidth and latency can impact performance.
  - Synchronization overhead between CPU and GPU.

Future Work:

- **Adaptive Iterations:**
  - Implement convergence checks to determine when to stop iterating.
- **Scalability Testing:**
  - Evaluate performance for varying systems including multi gpu high throughput setups.
- **Comparison with CPU Libraries:**
  - Benchmark against CPU implementations (e.g., LAPACK).

References:

1. **cuSOLVER Documentation:**
   - NVIDIA Developer Zone: cuSOLVER Library
2. **cuBLAS Documentation:**
   - NVIDIA Developer Zone: cuBLAS Library
3. **CUDA Toolkit Documentation:**
   - NVIDIA Developer Zone: CUDA Toolkit
4. **CUDA Programming Guide:**
   - NVIDIA Developer Zone: CUDA Programming Guide

Thanks