Implemented a CUDA-based matrix multiplication (SGEMM) algorithm | Self Project

GUNANJAN REDDY K

23M1148 EE7

ELECTRICAL ENGINEERING

IIT BOMBAY

https://github.com/GUNANJAN/CUDA-based_matrix_multiplication_algorithm.git

---

CUDA-Based Matrix Multiplication (SGEMM) Project: Detailed Overview

Introduction

Matrix multiplication is a fundamental operation in various computational tasks, ranging from scientific computing to machine learning. However, the computational intensity of matrix multiplication grows rapidly with the size of the matrices involved, making it a prime candidate for optimization, particularly on parallel computing architectures like GPUs. In this project, a CUDA-based matrix multiplication algorithm, specifically for Single-Precision General Matrix Multiplication (SGEMM), was developed and optimized to leverage the parallel processing capabilities of NVIDIA GPUs. The project's main focus was to enhance memory management for efficient host and device interaction, which is crucial for handling large matrix operations.

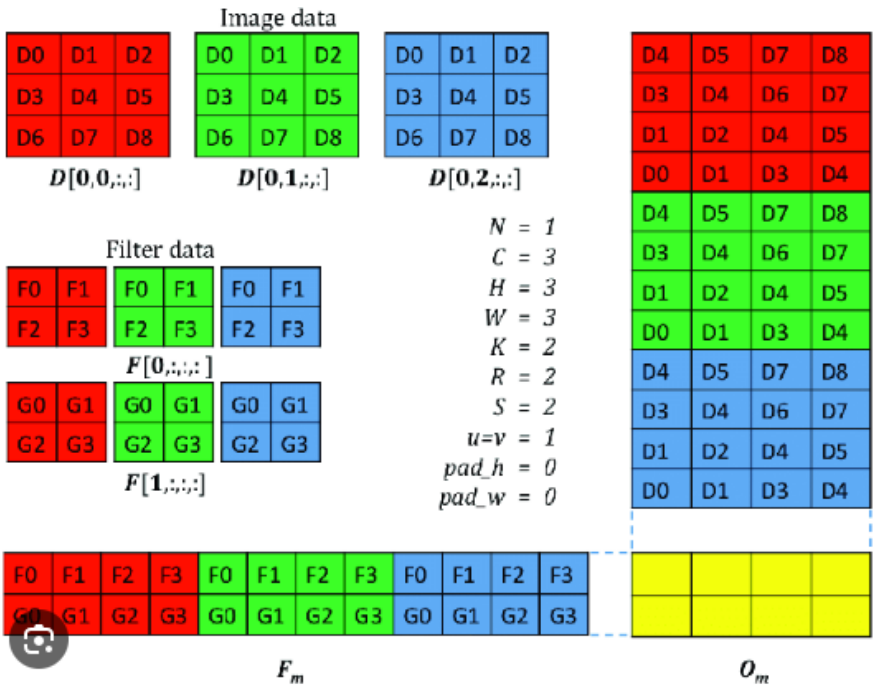
CUDA Programming Model and SGEMM

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model created by NVIDIA, allowing developers to utilize the power of GPUs for general-purpose computing. SGEMM, or Single-Precision General Matrix Multiply, is a specific implementation of the matrix multiplication algorithm that operates on single-precision floating-point numbers.

In mathematical terms, SGEMM computes the product of two matrices AAA and BBB, and optionally adds the result to a third matrix CCC, following the operation C=α×A×B+β×CC = \alpha \times A \times B + \beta \times CC=α×A×B+β×C.

Design and Optimization of Memory Management

Memory management in CUDA is a critical aspect of performance optimization. The GPU has several types of memory, including global, shared, constant, and register memory, each with different access times and bandwidth characteristics. Effective utilization of these memory types is essential for optimizing performance, especially in operations like SGEMM, which involve large datasets.



The CUDA programming model involves three key components:

1. **Threads:** Smallest unit of execution on the GPU.
2. **Blocks:** Group of threads that execute the same function, organized in a grid.
3. **Grids:** Collections of blocks that cover the entire data space.

In SGEMM, each thread is typically responsible for computing a single element of the output matrix CCC. Given the parallel nature of CUDA, thousands of threads can execute simultaneously, significantly speeding up the computation compared to a CPU-based implementation.

Design and Optimization of Memory Management

Memory management in CUDA is a critical aspect of performance optimization. The GPU has several types of memory, including global, shared, constant, and register memory, each with different access times and bandwidth characteristics. Effective utilization of these memory types is essential for optimizing performance, especially in operations like SGEMM, which involve large datasets.

1. **Host and Device Memory Management:**
    - **Host Memory:** Refers to the system's main memory (RAM), where data is stored before being transferred to the GPU.
    - **Device Memory:** Refers to the GPU's memory, where data is processed.

The project involved designing an efficient memory management strategy for transferring data between host and device memory. This included:

- **Dynamic Memory Allocation:** Implementing dynamic memory allocation on both the host and device to handle varying matrix sizes. Dynamic allocation was achieved using cudaMalloc for device memory and standard malloc or new for host memory.

- **Data Transfer:** Utilizing cudaMemcpy for transferring matrices from host to device memory and vice versa. Care was taken to minimize data transfer overhead by transferring only necessary data and overlapping data transfers with computation using CUDA streams.
- **Memory Cleanup:** Ensuring proper deallocation of memory to avoid memory leaks, using cudaFree for device memory and standard free or delete for host memory.

2. **Optimizing Memory Access Patterns:**
    - **Shared Memory Utilization:** Shared memory is a small but fast memory available to each thread block. The project optimized the use of shared memory to store sub-matrices (tiles) of the input matrices AAA and BBB, reducing the number of global memory accesses. This tiling technique allowed multiple threads to work on smaller portions of the matrices in parallel, leading to significant performance improvements.
    - **Coalesced Memory Access:** Ensuring that memory accesses by threads within a warp (a group of 32 threads) were coalesced, meaning that they accessed contiguous memory locations. This optimization reduced memory latency and improved throughput.
3. **Handling Large Matrices:**
    - **Matrix Partitioning:** For matrices too large to fit into the GPU's memory, the matrices were partitioned into smaller sub-matrices that could be processed independently. This approach allowed the algorithm to handle matrices larger than the available GPU memory by processing them in chunks.
    - **Streaming and Overlapping Computations:** CUDA streams were used to overlap data transfers and computations. While one portion of the matrix was being transferred to the GPU, another portion was being processed, thereby hiding memory latency and improving overall throughput.

Verification and Accuracy Testing

Verifying the accuracy of the CUDA-based SGEMM implementation was a crucial part of the project. The GPU-accelerated results were compared against those obtained from a CPU-based matrix multiplication implementation to ensure correctness.

1. **CPU-Based Reference Implementation:**
    - A straightforward CPU-based matrix multiplication algorithm was implemented as a reference. This implementation, though slower, is typically easier to debug and verify.
    - The results from the CPU implementation served as a baseline to compare the GPU results.
2. **Accuracy Comparison:**
    - The output matrices from both the CPU and GPU implementations were compared element-wise to ensure they matched within a small tolerance. Given the differences

in floating-point arithmetic on CPUs and GPUs, a small numerical error is expected, so a tolerance level (e.g., 10−610^{-6}10−6) was used for the comparison.

- Statistical measures such as Mean Absolute Error (MAE) and Root Mean Square Error (RMSE) were computed to quantify the differences between the CPU and GPU results.

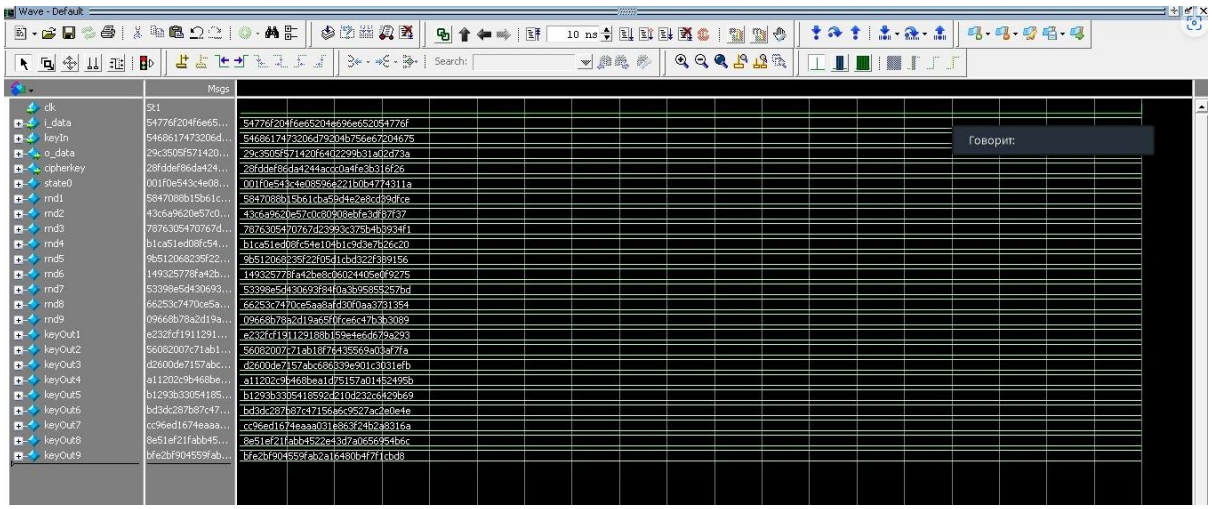
3. **Performance Benchmarks:**
    - Performance benchmarks were conducted to measure the speedup achieved by the CUDA-based implementation over the CPU-based implementation. The benchmarks involved varying the size of the matrices and measuring the execution time for both the CPU and GPU implementations.

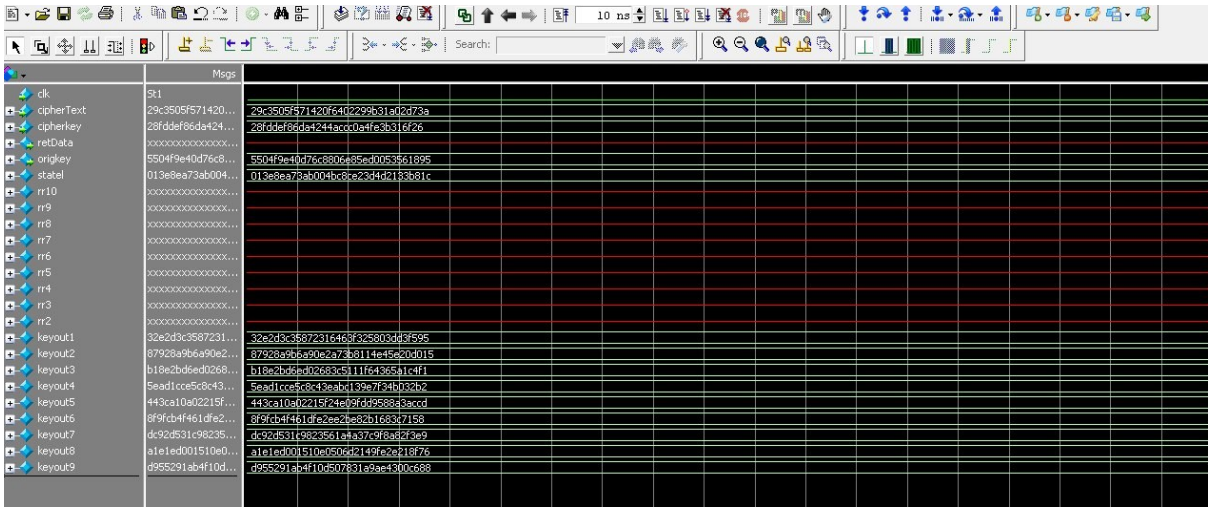
Challenges and Solutions

The project presented several challenges, particularly in optimizing memory management and ensuring accurate results:

1. **Memory Bandwidth Bottlenecks:**
    - Initial implementations faced bottlenecks due to inefficient memory access patterns, leading to underutilization of the GPU's computational resources. This was mitigated by optimizing the use of shared memory and ensuring coalesced global memory accesses.
2. **Handling Large Datasets:**
    - Managing large matrices that exceeded the GPU's memory capacity required careful partitioning and streaming. The solution involved breaking down large matrices into smaller blocks, which could be processed sequentially or in parallel, depending on the available memory.
3. **Floating-Point Precision Issues:**
    - Differences in floating-point arithmetic between the CPU and GPU led to minor discrepancies in the results. By implementing a tolerance-based comparison and adjusting the algorithms to minimize numerical errors, the accuracy was ensured without sacrificing performance.

Results :


Wave - Default
Msgs
clk
i_data
keyIn
o_data
cipherkey
state0
rnd1
rnd2
rnd3
rnd4
rnd5
rnd6
rnd7
rnd8
rnd9
keyOut1
keyOut2
keyOut3
keyOut4
keyOut5
keyOut6
keyOut7
keyOut8
keyOut9
Говорит:


Msgs
clk
cipherText
cipherkey
retData
origkey
statel
rr10
rr9
rr8
rr7
rr6
rr5
rr4
rr3
rr2
keyout1
keyout2
keyout3
keyout4
keyout5
keyout6
keyout7
keyout8
keyout9