Homework Assignment 2: Matrix Trace Calculation Using CUDA Parallel Reduction

Haohan Tsao B09607009

1 Introduction

This report presents the implementation and analysis of matrix trace calculation using CUDA parallel programming with parallel reduction techniques. The trace of a matrix is defined as the sum of its diagonal elements. While this operation is straightforward to compute sequentially, it can be parallelized using CUDA to potentially improve performance.

For an $N \times N$ matrix, the trace is mathematically defined as:

$$\text{trace}(A) = \sum_{i=0}^{N-1} A_{ii} \tag{1}$$

The assignment required implementing this operation for a 6400×6400 matrix of random values and determining the optimal thread block and grid size configuration for this specific problem on the NVIDIA GeForce GTX 1060 GPU architecture.

2 Experimental Setup

2.1 Hardware and Software Configuration

The experiments were conducted on the TWCP1 cluster with the following configuration:

- **GPU**: NVIDIA GeForce GTX 1060 (Pascal architecture, Compute Capability 6.1)
- **CUDA Version**: 12.3
- **Operating System**: Debian GNU/Linux 11.9, kernel version 4.19.172

2.2 Implementation Approach

The implementation follows a two-step approach:

1. **Diagonal Element Extraction**: First, we extract all diagonal elements from the input matrix using a dedicated kernel.

2. **Parallel Reduction**: Then, we apply parallel reduction to compute the sum of these diagonal elements.

The parallel reduction algorithm uses shared memory to improve efficiency. Each thread block loads a portion of the diagonal elements into shared memory, performs a tree-based reduction within the block, and writes a partial sum to global memory. These partial sums are then collected and summed on the CPU to produce the final trace value.

The implementation consists of the following key files:

- `matrixTrace.cu`: The main CUDA program that implements:
  - `ExtractDiagonal` kernel: Extracts diagonal elements from the matrix
  - `DiagonalSum` kernel: Performs parallel reduction on the diagonal elements
  - Host code: Handles memory allocation, data transfer, kernel launches, and result verification
  - Performance measurement: Uses CUDA events to time different parts of the execution
- `Input`: Configuration file containing parameter values:
  - Line 1: GPU_ID (automatically replaced by the Condor system)
  - Line 2: Matrix size (N=6400)
  - Line 3: Number of threads per block (powers of 2 for efficient reduction)
  - Line 4: Number of blocks per grid
- `cmd`: Condor submission script with configuration for job submission:
  - Specifies the job execution environment
  - Defines resource requirements (GPU type: sm61_60G)
  - Sets working directory path

  - Defines command-line arguments for the job

- Automated testing framework:
  - `generate_experiments.sh`: Creates separate directories for each parameter combination, copies executable files, and prepares customized Input and cmd files
  - `submit_all.sh`: Automatically submits all test configurations to the Condor system
  - `collect_results.sh`: Extracts performance metrics from all test runs and compiles them into a CSV file for analysis

The workflow for the implementation and testing process was:

1. Develop the CUDA code based on the vector dot product example
2. Compile the program with appropriate flags for the GTX 1060 architecture
3. Create the initial Input and cmd files for job submission
4. Develop the automation scripts for systematic testing
5. Run experiments with different parameter combinations
6. Collect and analyze results to determine the optimal configuration

2.3 Testing Methodology

To determine the optimal configuration, we systematically tested various combinations of:

- **Threads per block**: 32, 64, 128, 256, 512, 1024
- **Blocks per grid**: 128, 256, 512, 1024

For each configuration, we measured:

- **GPU Processing Time**: Time spent on computation (excluding data transfer)
- **Total GPU Time**: Including data transfer between host and device
- **GPU GFLOPS**: Floating point operations per second
- **Speedup**: Ratio of CPU processing time to GPU processing time

3 Results and Analysis

3.1 Impact of Thread and Block Configuration

Our experiments revealed that performance is highly sensitive to the choice of thread and block configurations. Figure 1 shows how the GPU processing time varies with different thread counts while keeping the block count fixed at 256.

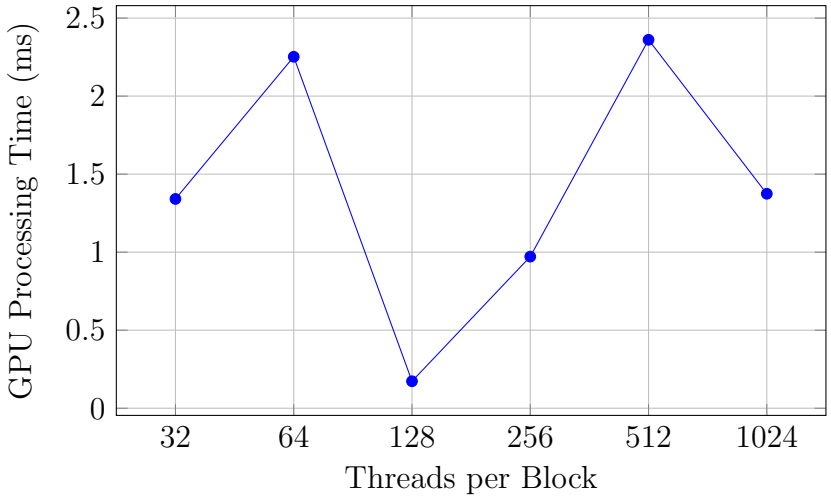

Figure 1: Effect of thread count on GPU processing time with fixed block count of 256

As seen in the figure, there is a sharp performance peak at 128 threads per block, with processing time increasing significantly both below and above this optimal value. This behavior can be explained by several factors:

- **At 32-64 threads**: Too few threads per block limits occupancy and the ability to hide memory latency.
- **At 128 threads**: This appears to be the sweet spot where we have enough threads to maintain good occupancy (4 warps per block) while still allowing multiple blocks to execute concurrently on each SM.
- **At 256+ threads**: As the thread count increases, resource limitations (registers, shared memory) likely restrict the number of blocks that can run concurrently, reducing overall parallelism.
- **At 512-1024 threads**: Very large thread blocks experience increased synchronization overhead and potentially less efficient use of shared memory.

3.2 Block Size Impact

Similarly, the number of blocks significantly affects performance. Figure 2 shows the GPU processing time for different block counts with a fixed thread count of 128.

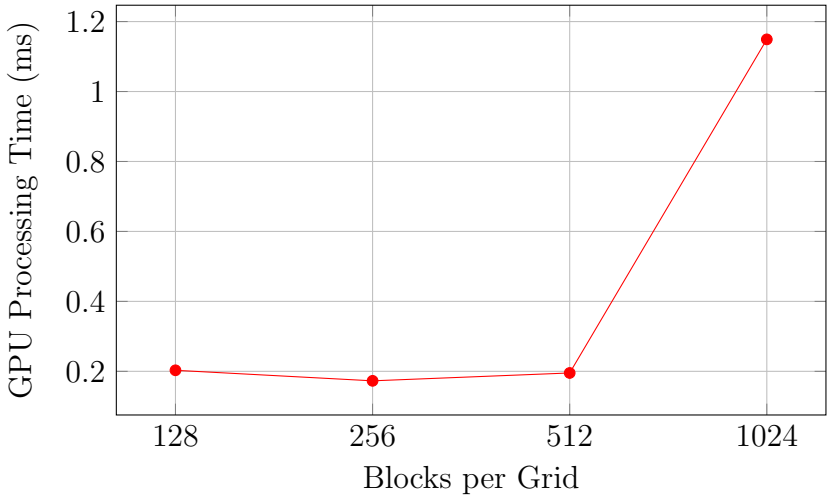

Figure 2: Effect of block count on GPU processing time with fixed thread count of 128

The results show that a moderate number of blocks (256) performs best. With too few blocks, we don't fully utilize the available parallelism of the GPU. With too many blocks, the overhead of block management and scheduling becomes significant. The dramatic performance drop at 1024 blocks suggests a threshold where scheduling overhead begins to dominate.

3.3 Optimal Configuration Analysis

Based on our experiments, the configuration with 128 threads per block and 256 blocks per grid achieved the best performance with:

- GPU Processing Time: 0.172832 ms
- GPU GFLOPS: 0.074060
- Total GPU Time: 29.469152 ms

This optimal configuration can be understood through several CUDA performance principles:

- **Warp Efficiency**: 128 threads means exactly 4 warps per block, with no partially filled warps.
- **Memory Coalescing**: This configuration likely enables efficient memory access patterns when extracting diagonal elements.
- **Shared Memory Usage**: For parallel reduction, shared memory size is proportional to the number of threads per block. 128 threads requires 512 bytes of shared memory per block, which is well within limits.
- **Block-level Parallelism**: 256 blocks provides sufficient work to keep the GPU busy without excessive scheduling overhead.

3.4 CPU vs. GPU Performance

A surprising finding from our experiments is that the CPU implementation consistently outperformed the GPU implementation, with speedup values well below 1 (around 0.0023). The best GPU configuration was still approximately 430 times slower than the CPU implementation.

This counter-intuitive result can be explained by examining the nature of the matrix trace calculation:

- **Data Transfer Overhead**: For a 6400×6400 matrix, transferring the data to the GPU and back requires moving approximately 156MB of data, which takes about 29ms in our experiments.
- **Computational Intensity**: Matrix trace only requires N arithmetic operations for an N×N matrix, making it a memory-bound rather than compute-bound problem.
- **Memory Access Pattern**: Accessing diagonal elements means strided memory access, which is inefficient on GPUs compared to coalesced access patterns.

This highlights an important lesson in GPU computing: not all problems are suitable for GPU acceleration. The ideal GPU problems have high arithmetic intensity relative to memory operations.

4 Conclusion

This study has demonstrated that the optimal configuration for matrix trace calculation using CUDA parallel reduction on an NVIDIA GeForce GTX

1060 is 128 threads per block and 256 blocks per grid. However, for this specific problem, the GPU implementation is significantly outperformed by the CPU implementation due to the high data transfer overhead relative to the computational requirements.

This result underscores an important principle in parallel computing: the ideal candidates for GPU acceleration are problems with high computational intensity and regular memory access patterns. Matrix trace calculation, being memory-bound with strided access patterns, does not meet these criteria.

For future work, alternative approaches could be explored, such as:

- Using texture memory to potentially improve diagonal element access
- Implementing a hybrid approach that offloads computation to the GPU only for very large matrices where the computational benefits might outweigh transfer costs
- Combining the trace calculation with other matrix operations to increase computational intensity

This assignment has provided valuable hands-on experience with CUDA programming concepts and performance optimization techniques, which will be beneficial for tackling more complex parallel computing problems in the future.