Introduction to CUDA Parallel Programming Homework Assignment 1

- March, 2025
- NTNU
- 41173058H ## Problem Statement

Write a CPU+GPU implementation for the modified matrix addition operation defined by **c(i,j) = 1/a(i,j) + 1/b(i,j)**. Using input NxN matrices A and B (where N=6400) with entries of random numbers between 0.0 and 1.0, determine the optimal block size by running your code. The implementation should be based on the provided template `vecAdd.cu`.

Source Code Analysis

Core Algorithm Implementation

The implementation consists of three main components: CPU baseline computation, GPU kernel implementation, and performance optimization through block size testing.

CPU Implementation

```
void addVectorsCPU(const float *a, const float *b, float *c, int n) {
    for (int i = 0; i < n; ++i) {
        c[i] = 1.0f / a[i] + 1.0f / b[i];
    }
}
```

The CPU version provides a sequential baseline implementation that computes the reciprocal sum for each element pair. This serves as both a performance baseline and validation reference.

GPU Kernel Implementation

```
__global__ void addKernel(const float *a, const float *b, float *c, int n) {
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    if (idx < n) {
        c[idx] = 1.0f / a[idx] + 1.0f / b[idx];
    }
}
```

The GPU kernel uses the standard CUDA thread indexing pattern with `threadIdx.x + blockDim.x * blockIdx.x` to ensure each thread processes a unique element[2]. The boundary check `if (idx < n)` prevents out-of-bounds memory access when the total number of threads exceeds the array size[2].

Memory Management Strategy The implementation uses explicit memory management with `cudaMalloc()` and `cudaMemcpy()` operations[2]. Memory transfers are timed separately to analyze the performance bottleneck:

```
// Input transfer timing
auto start_input = std::chrono::high_resolution_clock::now();
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
auto end_input = std::chrono::high_resolution_clock::now();
```

Block Size Optimization The code systematically tests block sizes of 32, 64, 128, 256, 512, and 1024 threads to find the optimal configuration[2]. This approach follows CUDA best practices where block sizes should be multiples of 32 (the warp size)[2].

Results and Analysis

Performance Comparison

CPU Performance Baseline

- **Processing Time**: 99.51 ms
- **GFLOPS**: 1.65
- **Computational Intensity**: Single-threaded sequential execution

GPU Performance Analysis by Block Size

Block Size	Kernel Time (ms)	GFLOPS	Total Time (ms)	Efficiency
32	9.76	16.79	179.54	Poor
64	6.63	24.72	139.31	Good
128	6.53	25.08	139.13	Better
256	**6.49**	**25.24**	**139.94**	**Optimal**
512	6.52	25.14	139.22	Good
1024	6.59	24.85	138.96	Good

Key Performance Insights

Optimal Configuration Analysis

- **Best Block Size**: 256 threads
- **Peak Kernel Performance**: 25.24 GFLOPS
- **Speedup vs CPU**: 15.33x
- **Kernel Efficiency**: 6.49 ms execution time

The optimal block size of 256 threads represents the sweet spot between parallelism and resource utilization[2]. This configuration maximizes occupancy while avoiding resource contention that occurs with larger block sizes.

Memory Transfer Bottleneck

- **Input Transfer Time**: ~91-92 ms (52% of total time)
- **Output Transfer Time**: ~41 ms (23% of total time)
- **Kernel Computation**: ~6.5 ms (4% of total time)
- **Memory Overhead**: 96% of total execution time

The analysis reveals that memory transfers dominate the execution time, which is typical for memory-bound operations[5]. The PCIe bandwidth limitation creates a significant bottleneck compared to the GPU's computational throughput.

Block Size Performance Characteristics Small Block Sizes (32 threads): - **Poor Performance**: 9.76 ms kernel time - **Low Occupancy**: Insufficient parallelism to saturate GPU resources - **Warp Underutilization**: Only one warp per block limits instruction throughput

Medium Block Sizes (64-128 threads): - **Improved Performance**: ~6.5-6.6 ms kernel time - **Better Occupancy**: Multiple warps per block improve resource utilization - **Balanced Resource Usage**: Good balance between parallelism and memory bandwidth

Optimal Block Size (256 threads): - **Peak Performance**: 6.49 ms kernel time - **Maximum Efficiency**: Optimal balance of occupancy and resource utilization - **Warp Efficiency**: 8 warps per block maximize instruction-level parallelism[2]

Large Block Sizes (512-1024 threads): - **Slight Degradation**: Marginally slower than optimal - **Resource Contention**: Higher register and shared memory pressure - **Diminishing Returns**: Additional threads don't improve performance

Numerical Accuracy Validation

The implementation maintains perfect numerical accuracy with `norm(h_C - h_D) = 0.000000000000000e+00`, confirming that the GPU computation produces identical results to the CPU baseline.

Discussion

Algorithm Efficiency

The reciprocal addition operation `c[i] = 1/a[i] + 1/b[i]` is computationally lightweight, making this a memory-bound problem rather than compute-bound[5]. The 15.33x speedup demonstrates the GPU's effectiveness even for simple operations when sufficient parallelism is available.

Performance Optimization Strategies

1. **Memory Optimization**: The current implementation could benefit from unified memory (`cudaMallocManaged`) to simplify memory management[5]
2. **Streaming**: Overlapping computation with memory transfers using CUDA streams could reduce total execution time
3. **Memory Coalescing**: The current access pattern already ensures coalesced memory access for optimal bandwidth utilization[2]

Scalability Considerations

The optimal block size of 256 threads is well-suited for modern GPU architectures, providing good occupancy across different GPU generations[2]. This configuration scales effectively with the number of streaming multiprocessors available on the device.

Real-World Applications

This pattern of element-wise operations with reciprocals is common in: - **Scientific Computing**: Harmonic mean calculations - **Image Processing**: Brightness and contrast adjustments - **Financial Modeling**: Risk assessment calculations - **Machine Learning**: Normalization operations

Conclusion

The homework successfully demonstrates CUDA programming fundamentals and performance optimization techniques. The optimal block size of 256 threads achieves 25.24 GFLOPS with a 15.33x speedup over CPU execution. While memory transfers currently dominate execution time, the kernel optimization shows the importance of proper thread block sizing for maximizing GPU utilization. The implementation provides a solid foundation for understanding parallel computing principles and CUDA optimization strategies.

Submission Guidelines

Submit your homework report including source codes, results, and discussions. Prepare the discussion file using a typesetting system (LaTeX, Word, etc.) and convert to PDF. Compress all files into a gzipped tar file named with your student number and problem set number (e.g., `r05202043_HW1.tar.gz`). Send your homework with the title "your_student_number_HW1" to twchiu@phys.ntu.edu.tw before 17:00, June 11, 2025.

[1] https://github.com/CisMine/Parallel-Computing-Cuda-C/blob/main/README.md
[2] https://docs.nvidia.com/cuda/cuda-c-programming-guide/ [3] https://github.com/aaditya29/Parallel-Computing-And-CUDA [4] https://developer.nvidia.com/cuda-education
[5] https://developer.nvidia.com/blog/even-easier-introduction-cuda/ [6] https://gitlab.com/nvidia/container-images/cuda/blob/master/doc/README.md

[7] https://ipython-books.github.io/58-writing-massively-parallel-code-for-nvidia-graphics-cards-gpus-with-cuda/ [8] https://gitlab.com/nvidia/container-images/cuda/-/blob/master/doc/README.md