CUDA Matrix Addition Report

Your Name

March 2, 2024

1 Introduction

This report presents a performance comparison between CPU and CUDA implementations of matrix addition.

2 Code Implementation

The CUDA code and cpu code for matrix addition are shown below:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <cuda_runtime.h>

#define N 6400


// CUDA kernel for matrix addition
__global__ void matrixAddition(float **a, float **b, float **result
    , int rows, int cols) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < rows && col < cols) {
        result[row][col] = a[row][col] + b[row][col];
    }
    __syncthreads();
}

void execute(int block_size) {
    // Matrix dimensions
    int rows = N;
    int cols = N;

    // Host arrays and initialization with random values
    float **h_a = new float*[rows];
    float **h_b = new float*[rows];
    float **h_result = new float*[rows];

    for (int i = 0; i < rows; ++i) {
        h_a[i] = new float[cols];
```

```
32          h_b[i] = new float[cols];
33          h_result[i] = new float[cols];
34      }
35
36      srand(time(NULL));
37      for (int i = 0; i < rows; ++i) {
38          for (int j = 0; j < cols; ++j) {
39              h_a[i][j] = static_cast<float>(rand()) / RAND_MAX;
40              h_b[i][j] = static_cast<float>(rand()) / RAND_MAX;
41          }
42      }
43
44      // Device arrays
45      float **d_a, **d_b, **d_result;
46      cudaMalloc(&d_a, rows * sizeof(float *));
47      cudaMalloc(&d_b, rows * sizeof(float *));
48      cudaMalloc(&d_result, rows * sizeof(float *));
49
50      float **d_a_data, **d_b_data, **d_result_data;
51      cudaMalloc(&d_a_data, rows * cols * sizeof(float));
52      cudaMalloc(&d_b_data, rows * cols * sizeof(float));
53      cudaMalloc(&d_result_data, rows * cols * sizeof(float));
54
55      cudaMemcpy(d_a, h_a, rows * sizeof(float *),
        cudaMemcpyHostToDevice);
56      cudaMemcpy(d_b, h_b, rows * sizeof(float *),
        cudaMemcpyHostToDevice);
57      cudaMemcpy(d_result, h_result, rows * sizeof(float *),
        cudaMemcpyHostToDevice);
58
59      cudaMemcpy(d_a_data, h_a[0], rows * cols * sizeof(float),
        cudaMemcpyHostToDevice);
60      cudaMemcpy(d_b_data, h_b[0], rows * cols * sizeof(float),
        cudaMemcpyHostToDevice);
61
62      // Timing
63      clock_t start, stop;
64      start = clock();
65
66      // Launch kernel for matrix addition and measure time
67      dim3 threadsPerBlock(block_size, block_size);
68      dim3 numBlocks((cols + threadsPerBlock.x - 1) / threadsPerBlock
        .x,
69                     (rows + threadsPerBlock.y - 1) / threadsPerBlock
        .y);
70
71      matrixAddition<<<numBlocks, threadsPerBlock>>>(d_a, d_b,
        d_result, rows, cols);
72
73      cudaDeviceSynchronize(); // Ensure all kernel launches are
        complete
74
75      stop = clock();
76      double timer_seconds = ((double)(stop - start)) /
        CLOCKS_PER_SEC * 1000; // Convert to milliseconds
77      std::cout << "Block size: " << block_size << ", Time taken: "
        << timer_seconds << " ms\n";
```

```
78
79      // Copy result back to host
80      cudaMemcpy(h_result, d_result, rows * sizeof(float *),
        cudaMemcpyDeviceToHost);
81      cudaMemcpy(h_result[0], d_result_data, rows * cols * sizeof(
        float), cudaMemcpyDeviceToHost);
82
83      // Free device memory
84      cudaFree(d_a);
85      cudaFree(d_b);
86      cudaFree(d_result);
87      cudaFree(d_a_data);
88      cudaFree(d_b_data);
89      cudaFree(d_result_data);
90
91      // Free host memory
92      for (int i = 0; i < rows; ++i) {
93          delete[] h_a[i];
94          delete[] h_b[i];
95          delete[] h_result[i];
96      }
97      delete[] h_a;
98      delete[] h_b;
99      delete[] h_result;
100 }
101
102 int main() {
103     int a[] = {4, 8, 10, 16, 20, 32};
104     for (int i = 0; i < 6; i++) {
105         execute(a[i]);
106     }
107
108     return 0;
109 }
```

Listing 1: CUDA Matrix Addition Code

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <ctime>
4
5  #define N 6400
6
7  void matrixAdd(float *a, float *b, float *c, int n) {
8      for (int i = 0; i < n; ++i) {
9          for (int j = 0; j < n; ++j) {
10             int index = i * n + j;
11             c[index] = a[index] + b[index];
12         }
13     }
14 }
15
16 int main() {
17     float *a, *b, *c; // Matrices
18
19     int size = N * N * sizeof(float);
20
21     // Allocate memory for matrices on host
```

```
    a = (float *)malloc(size);
    b = (float *)malloc(size);
    c = (float *)malloc(size);

    // Initialize matrices with random numbers
    srand(time(NULL));
    for (int i = 0; i < N * N; ++i) {
        a[i] = static_cast<float>(rand()) / RAND_MAX;
        b[i] = static_cast<float>(rand()) / RAND_MAX;
    }

    // Start timing
    clock_t start = clock();

    // Perform matrix addition on CPU
    matrixAdd(a, b, c, N);

    // Stop timing
    clock_t end = clock();

    // Calculate elapsed time in milliseconds
    double elapsed_time = double(end - start) / CLOCKS_PER_SEC *
    1000.0;

    // Print execution time
    std::cout << "CPU execution time: " << elapsed_time << " ms" <<
     std::endl;

    // Free host memory
    free(a);
    free(b);
    free(c);

    return 0;
}
```

Listing 2: CUDA Matrix Addition Code

3 Hardware Specification

The CUDA matrix addition implementation was executed on a system with the following specifications:

- CPU: Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz (12 cores, 24 threads)
- GPU: NVIDIA GeForce RTX 3090

4 Performance Analysis

The performance of the CUDA matrix addition program was evaluated using different block sizes. The results are summarized in Table 1.

4.1 GPU performance

```
b12902015@meow1 [~/cuda_programming/hw1] ./gpu
Block size: 4, Time taken: 30.224 ms
Block size: 8, Time taken: 0.007 ms
Block size: 10, Time taken: 0.01 ms
Block size: 16, Time taken: 0.005 ms
Block size: 20, Time taken: 0.005 ms
Block size: 32, Time taken: 0.011 ms
```

Table 1: Performance Analysis of CUDA Matrix Addition

Block Size	**Time Taken (ms)**
4	30.224
8	0.007
10	0.010
16	0.005
20	0.005
32	0.011

4.2 CPU performance

```
b12902015@meow1 [~] ./cpu
CPU execution time: 168.315 ms
```

5 Discussion

The execution time data shows variation in performance with different block sizes. The execution time decreases drastically as the block size increases from 4 to 16, reaching its lowest value at a block size of 16 and 20. However, further increasing the block size to 20 and 32 results in a slight increase in execution time.

Reasons for Performance Variation:

1. Increasing the block size allows for better utilization of GPU resources, such as registers and shared memory. This improved resource utilization can lead to more efficient execution and shorter execution times.

2. However , when further increases the block size . It may reach a point where the resources are exhausted, leading to resource contention and decreased performance .

6 Conclusion

The CUDA matrix addition implementation outperformed the CPU-based counterpart, showcasing the GPU's superior computational power. The performance was notably affected by block size selection. Smaller block sizes led to longer execution times due to lower resource utilization. Performance improved significantly with larger block sizes up to a point, beyond which further increases resulted in diminishing returns, likely due to resource limitations and scheduling overhead. Optimizing block size is crucial for maximizing GPU performance in matrix addition operations.