Parallel Matrix Addition: Serial C Code vs. CUDA Performance Comparison

Problem Statement

The problem involves performing matrix addition on two $N \times N$ matrices where each element is of double precision (64-bit). We are to implement the matrix addition both serially and in parallel using CUDA, and then compare the execution times to compute the speedup achieved by parallelization. For this report, we have chosen $N = 10000$, which results in $10000 \times 10000 = 10^8$ operations.

We will compare:

- Serial execution time using a CPU.
- Parallel execution time using CUDA on a GPU.
- Speedup achieved by parallelization.

Compute Power of the System

The system is powered by the **NVIDIA GeForce GTX 1650** graphics card, based on the *TU117* GPU architecture. Below are the detailed specifications of the GPU:

Graphics Processor Specifications

- **GPU Name:** TU117
- **GPU Variant:** TU117-300-A1
- **Architecture:** Turing
- **Process Size:** 12 nm
- **Die Size:** 200 mm^2
- **Transistors:** 4.7 billion
- **DirectX Support:** 12
- **Shader Model:** 6.8

- **CUDA:** 7.5
- **NVENC:** 5th Gen
- **NVDEC:** 4th Gen
- **PureVideo HD:** VP10
- **VDPAU:** Feature Set J

Core Configuration

- **Shading Units (CUDA Cores):** 896
- **Texture Mapping Units (TMUs):** 56
- **Render Output Units (ROPs):** 32
- **Streaming Multiprocessors (SM):** 14
- **L1 Cache (per SM):** 64 KB
- **L2 Cache:** 1024 KB

Memory Specifications

- **Memory Size:** 4 GB GDDR5
- **Memory Bus:** 128 bit
- **Memory Clock:** 2001 MHz (8 Gbps effective)
- **Memory Bandwidth:** 128.1 GB/s

Clock Speeds

- **Base Clock:** 1485 MHz
- **Boost Clock:** 1665 MHz
- **Memory Clock:** 2001 MHz (8 Gbps effective)

Performance Metrics

- **Pixel Rate:** 53.28 GPixel/s
- **Texture Rate:** 93.24 GTexel/s
- **FP32 Performance (float):** 2.984 TFLOPS
- **FP16 Performance (half):** 5.967 TFLOPS
- **FP64 Performance (double):** 93.24 GFLOPS

Power and Thermal Design

- **Thermal Design Power (TDP):** 75 W
- **Recommended PSU:** 250 W
- **Slot Width:** Dual-slot
- **Power Connectors:** None

Display and Connectivity

- **Outputs:** 1x DVI, 1x HDMI 2.0, 1x DisplayPort 1.4a
- **Bus Interface:** PCIe 3.0 x16
- **Supported Resolutions:** 1920x1080, 2560x1440, 3840x2160

Relative Performance

At launch, the GeForce GTX 1650 performed similarly to other mid-range graphics cards in the market, with a launch price of 149 USD. It offers an excellent balance of price and performance for mainstream gaming and light productivity tasks.

Serial Code Implementation

The serial implementation of matrix addition involves iterating over the elements of two $N \times N$ matrices, summing the corresponding elements, and storing the result in a third matrix. Below is the serial code:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 10000  // 10K

void matrix_addition(double **A, double **B, double **C) {
    int i, j;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
}

int main() {
    // Initialize matrices
    double **A = (double **)malloc(N * sizeof(double *));
    double **B = (double **)malloc(N * sizeof(double *));
    double **C = (double **)malloc(N * sizeof(double *));
    for (int i = 0; i < N; i++) {
        A[i] = (double *)malloc(N * sizeof(double));
        B[i] = (double *)malloc(N * sizeof(double));
        C[i] = (double *)malloc(N * sizeof(double));
    }

    // Fill matrices A and B with values
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            A[i][j] = i;
            B[i][j] = N - i;
        }
    }

    // Measure time for serial execution
    clock_t start = clock();

    matrix_addition(A, B, C);

    clock_t end = clock();
    double serial_time = (double)(end - start) / CLOCKS_PER_SEC;

    printf("Serial Execution C[%d][%d] = %lf\n", N-1, N-1, C[N-1][N-1]);
    printf("Serial Execution Time: %lf seconds\n", serial_time);

    // Free memory
    for (int i = 0; i < N; i++) {
        free(A[i]);
        free(B[i]);
        free(C[i]);
    }
    free(A);
    free(B);
    free(C);

    return 0;
}
```

Parallel Code Implementation (CUDA)

For the parallel implementation, we use CUDA to offload the matrix addition task to the GPU. The matrix is divided into smaller blocks, and each block is processed by multiple threads in parallel. The following is the parallel CUDA code:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 10000  // 10K
#define TILE_SIZE 16

// CUDA Kernel to perform matrix addition
__global__ void matrix_addition(double *A, double *B, double *C, int n) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < n && col < n) {
        C[row * n + col] = A[row * n + col] + B[row * n + col];
    }
}

int main() {
    int n = N;  // Size of the matrix
    size_t size = n * n * sizeof(double);

    // Host matrices
    double *A = (double*)malloc(size);
    double *B = (double*)malloc(size);
    double *C = (double*)malloc(size);

    // Initialize matrices A and B with values
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            A[i * n + j] = i;
            B[i * n + j] = N - i;
        }
    }

    // Device matrices
    double *d_A, *d_B, *d_C;

    // Allocate memory on the device
    cudaMalloc((void**)&d_A, size);
    cudaMalloc((void**)&d_B, size);
    cudaMalloc((void**)&d_C, size);

    // Copy data from host to device
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

    // Set up grid and block dimensions
    dim3 threadsPerBlock(TILE_SIZE, TILE_SIZE);
    dim3 numBlocks((n + TILE_SIZE - 1) / TILE_SIZE, (n + TILE_SIZE - 1) / TILE_SIZE);

    // Measure time for parallel execution on the GPU
    double start_time = (double)clock();

    matrix_addition<<<numBlocks, threadsPerBlock>>>(d_A, d_B, d_C, n);

    // Check for errors in kernel launch
    cudaError_t err = cudaGetLastError();
    if (err != cudaSuccess) {
        printf("CUDA error: %s\n", cudaGetErrorString(err));
        return -1;
    }

    // Wait for GPU to finish before accessing the result
    cudaDeviceSynchronize();

    double end_time = (double)clock();
    double parallel_time = (end_time - start_time) / CLOCKS_PER_SEC;

    // Copy data from device to host
    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

    // Print results
    printf("Parallel (CUDA) Execution C[%d] = %lf\n", N-1, C[(N-1) * N + (N-1)]);
    printf("Parallel (CUDA) Execution Time: %lf seconds\n", parallel_time);

    // Clean up
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    free(A);
    free(B);
    free(C);

    return 0;
}
```

Results and Performance Analysis

The results of the execution are as follows:

- **Serial Execution Time:** 0.540691 seconds
- **Parallel (CUDA) Execution Time:** 0.013554 seconds

The speedup is calculated as the ratio of the serial execution time to the parallel execution time:

$$\text{Speedup} = \frac{\text{Serial Time}}{\text{Parallel Time}} = \frac{0.540691}{0.013554} \approx 39.91$$

Thus, the parallel execution using CUDA achieved a speedup of approximately 39.91 times compared to the serial execution.

Conclusion

In this report, we have demonstrated the use of CUDA to parallelize the matrix addition operation. The serial implementation took 0.540691 seconds, while the parallel CUDA implementation took only 0.013554 seconds. The observed speedup is significant, highlighting the power of parallel computing using GPUs for large matrix operations. This approach can be applied to other computationally intensive tasks as well.