Matrix Multiplication: Serial C Code vs. Parallel CUDA Implementation

Problem Statement

The task is to perform matrix multiplication for two $N \times N$ matrices, where each element of the matrix is a double-precision number. The matrix size N is sufficiently large, at least 10000, to test the performance of both serial and parallel implementations. The goal is to compare the time taken for matrix multiplication in both serial and parallel approaches and compute the speedup achieved by using CUDA for parallel execution.

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.

Matrix Multiplication: Serial Implementation

The serial implementation of matrix multiplication computes the product of two matrices A and B to produce matrix C. The multiplication algorithm involves three nested loops where each element of matrix C is computed by taking the dot product of the corresponding row of matrix A and column of matrix B.

Code: Serial Implementation

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

#define N 1000  // Matrix size (1K)

void matrix_multiply_serial(double **A, double **B, double **C) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            C[i][j] = 0.0;
            for (int k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
```

```
    }
}

int main() {
    double **A, **B, **C;

    // Allocate memory for matrices
    A = (double **)malloc(N * sizeof(double *));
    B = (double **)malloc(N * sizeof(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));
    }

    // Initialize matrices A and B with random values
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            A[i][j] = i + 1;
            B[i][j] = N - i;
        }
    }

    // Record start time
    clock_t start = clock();

    // Perform matrix multiplication
    matrix_multiply_serial(A, B, C);

    // Record end time
    clock_t end = clock();

    // Print execution time
    double serial_time = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Serial C[%d][%d] = %f\n", N-1, N-1, C[N-1][N-1]);
    printf("Serial Time: %f seconds\n", serial_time);

    // Free allocated 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;
}
```

Explanation of Serial Code

The serial code begins by allocating memory for the matrices A, B, and C. The matrices are initialized with simple values for testing: $A[i][j] = i + 1$ and $B[i][j] = N - i$. The matrix multiplication is carried out by the function `matrix_multiply_serial()`, which uses three nested loops to calculate each element of matrix C. The total execution time is measured using the `clock()` function and printed at the end.

Execution Time for Serial Code

The serial code executes the matrix multiplication for matrices of size 1000×1000 and takes approximately 5.85 seconds to complete.

$$\text{Serial Time} = 5.852081 \text{ seconds}$$

Matrix Multiplication: Parallel Implementation Using CUDA

In the parallel implementation, the matrix multiplication is performed on the GPU using CUDA. The kernel is designed to compute the product of matrices A and B in parallel, where each thread computes one element of the result matrix C.

Code: Parallel Implementation (CUDA)

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

#define N 1000  // Matrix size (1K)
#define TILE_SIZE 16  // Tile size for block size

__global__ void matrix_multiply_kernel(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) {
        double value = 0.0;
        for (int k = 0; k < n; k++) {
            value += A[row * n + k] * B[k * n + col];
        }
        C[row * n + col] = value;
    }
}

void matrix_multiply_cuda(double *A, double *B, double *C, int n) {
    double *d_A, *d_B, *d_C;

    // Allocate memory on device
    cudaMalloc((void **)&d_A, n * n * sizeof(double));
    cudaMalloc((void **)&d_B, n * n * sizeof(double));
    cudaMalloc((void **)&d_C, n * n * sizeof(double));

    // Copy matrices A and B from host to device
    cudaMemcpy(d_A, A, n * n * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, n * n * sizeof(double), cudaMemcpyHostToDevice);

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

    // Record start time
    double start = (double)clock() / CLOCKS_PER_SEC;

    // Launch the kernel
    matrix_multiply_kernel<<<gridDim, blockDim>>>(d_A, d_B, d_C, N);

    // Synchronize the device to ensure kernel completes
    cudaDeviceSynchronize();

    // Record end time
    double end = (double)clock() / CLOCKS_PER_SEC;

    // Copy the result matrix from device to host
    cudaMemcpy(C, d_C, N * N * sizeof(double), cudaMemcpyDeviceToHost);

    // Print execution time
    double cuda_time = end - start;
    printf("CUDA C[%d][%d] = %f\n", N-1, N-1, C[(N-1) * N + (N-1)]);
    printf("CUDA Time: %f seconds\n", cuda_time);

    // Free device memory
```

```
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

Explanation of Parallel Code

The CUDA implementation uses a kernel function to parallelize the matrix multiplication. The kernel is launched with a grid of blocks, where each block contains threads that compute elements of matrix C. Each thread computes one element of the resulting matrix C by performing a dot product between a row of matrix A and a column of matrix B. Memory for matrices is allocated on the GPU, and the matrices are copied from the host to the device. After the kernel completes, the result is copied back to the host, and the execution time is measured.

Execution Time for Parallel Code (CUDA)

The CUDA code executes the matrix multiplication for matrices of size 1000×1000 and takes approximately 0.027 seconds.

$$\text{CUDA Time} = 0.027291 \text{ seconds}$$

Speedup Calculation

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

$$\text{Speedup} = \frac{\text{Serial Time}}{\text{Parallel Time}} = \frac{5.852081}{0.027291} \approx 214.85$$

Thus, the speedup achieved by using CUDA for matrix multiplication is approximately 214.85 times faster than the serial implementation.

Conclusion

The parallel implementation of matrix multiplication using CUDA provides a significant performance improvement compared to the serial implementation. With the given problem size, the parallel code achieves a speedup of approximately 214.85, demonstrating the effectiveness of GPU acceleration for large-scale matrix operations.