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;
}
```

CUDA Kernel for Matrix Addition

The CUDA kernel for matrix addition divides the task of adding two matrices into smaller, independent tasks. Each thread computes the sum of one corresponding element from matrices A and B, storing the result in matrix C. The kernel function is as follows:

```
__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];
    }
}
```

This kernel performs matrix addition by summing the corresponding elements of matrices A and B and storing the result in matrix C. Each thread is responsible for computing one element of the result matrix.

Key Concepts and Performance Improvements

1. **Parallelization of Computation**: Each thread computes the sum of one element from matrices A and B, which is stored in the corresponding element of matrix C. The row and column indices for each element are calculated using:

```
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
```

This calculation maps each thread to a unique element in the matrix, enabling parallel execution across thousands of threads. The use of block and thread indices ensures that the workload is evenly distributed, significantly speeding up the computation compared to a serial implementation.

2. **Efficient Memory Usage**: CUDA organizes threads into blocks, and each block processes a small tile of the matrices. This approach maximizes memory throughput by reducing the frequency of global memory accesses. Each thread reads its required values from matrices A and B, performs the addition, and writes the result to matrix C. This minimizes memory contention and optimizes memory access patterns.

3. **Load Balancing**: By dividing the matrix into smaller blocks, the kernel ensures that each thread block handles a small section of the matrices. This division allows for better load balancing across the GPU's cores, reducing idle time and ensuring that all threads work independently. As a result, the kernel achieves high parallel efficiency.

4. **Scalability**: The kernel scales efficiently with larger matrices. As the matrix size increases, more threads are launched, and the workload is distributed

more effectively. The number of threads grows linearly with the matrix size, allowing the GPU to maintain high utilization even for large matrices.

5. **Efficient Use of CUDA Hardware**: The kernel leverages the GPU's hardware resources by distributing the work across multiple streaming multiprocessors (SMs). This ensures that a large number of threads run simultaneously, utilizing the many cores available on the GPU. The result is a significant performance boost, particularly for large matrices where parallelization has the greatest impact.

Role of `threadIdx.x`, `threadIdx.y`, `blockIdx.x`, and `blockIdx.y`

The calculation of `row` and `col` is central to the kernel's performance:

```
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
```

Here, `blockIdx` and `threadIdx` represent the block and thread indices in the grid, while `blockDim` specifies the number of threads per block. This calculation ensures that:

- Each thread is assigned a unique element in the matrix.
- The workload is evenly distributed across all threads.
- The kernel can handle matrices of arbitrary size by adjusting the grid and block dimensions.

Block-Wise Division and Performance Improvement

The use of `threadIdx.x`, `threadIdx.y`, `blockIdx.x`, and `blockIdx.y` enables block-wise division of the matrix, which is a key factor in improving performance. Here's how it works:

1. **Thread Organization**: - CUDA organizes threads into a grid of blocks, where each block contains a fixed number of threads. - Each thread has a unique identifier within its block, given by `threadIdx.x` and `threadIdx.y`. - Each block also has a unique identifier within the grid, given by `blockIdx.x` and `blockIdx.y`.

2. **Block-Wise Division**: - The matrix is divided into smaller tiles, where each tile is processed by one thread block. - The size of each tile is determined by the block dimensions (`blockDim.x` and `blockDim.y`). - For example, if the block size is 16×16, each block processes a 16×16 tile of the matrix.

3. **Performance Improvement**: - **Parallel Execution**: By dividing the matrix into smaller tiles, multiple blocks can work on different parts of the matrix simultaneously. This maximizes parallelism and reduces computation time. - **Memory Locality**: Threads within a block can cooperate to load data into shared memory, which is much faster than global memory. This reduces the latency of memory accesses and improves overall performance. - **Load Balancing**: Block-wise division ensures that the workload is evenly distributed across

all blocks, minimizing idle time and maximizing GPU utilization. - **Scalability**: As the matrix size increases, more blocks can be launched to handle the additional workload, ensuring that the kernel remains efficient even for very large matrices.

The CUDA kernel for matrix addition optimizes performance by leveraging parallel threads, improving memory access patterns, and efficiently distributing work across the GPU. The use of `threadIdx.x`, `threadIdx.y`, `blockIdx.x`, and `blockIdx.y` ensures that each thread processes a unique element, enabling high scalability and performance. Block-wise division further enhances performance by maximizing parallelism, improving memory locality, and ensuring load balancing. This approach results in a significant speedup compared to serial implementations, especially for large matrices. The kernel is highly scalable and well-suited for computationally intensive tasks in fields such as scientific computing, machine learning, and data analysis.

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.