Parallel Vector Addition and Multiplication of Double Precision Floating-Point Numbers: Serial C Code vs. CUDA

Problem Statement

The task is to perform two operations on two vectors of size at least 1 million using both serial and parallel CUDA programming. The operations to be performed are:

1. Vector addition of two vectors A and B.
2. Vector multiplication of two vectors A and B.

The serial and parallel codes are provided for both operations, and the performance of the serial and parallel codes is compared by measuring the time taken for execution. The speedup is calculated by dividing the serial execution time by the parallel execution time.

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 for Vector Operations

The serial code implements two operations: vector addition and vector multiplication. Here is the code:

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

#define N 10000000  // Number of elements (10 million)

// Serial function for vector addition
void vector_add(double *A, double *B, double *C) {
    for (int i = 0; i < N; i++) {
        C[i] = A[i] + B[i];
    }
}

// Serial function for vector multiplication
void vector_multiply(double *A, double *B, double *C) {
    for (int i = 0; i < N; i++) {
        C[i] = A[i] * B[i];
    }
}

int main() {
    double *A, *B, *C;
    A = (double*)malloc(N * sizeof(double));
    B = (double*)malloc(N * sizeof(double));
    C = (double*)malloc(N * sizeof(double));

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

    // Time serial vector addition
    clock_t start = clock();
    vector_add(A, B, C);
    clock_t end = clock();
```

```
    double add_time = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Vector Addition C[%d] = %f\n", 9999999, C[9999999]);
    printf("Serial Vector Addition Time: %f seconds\n", add_time);

    // Time serial vector multiplication
    start = clock();
    vector_multiply(A, B, C);
    end = clock();
    double multiply_time = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Vector Multiplication C[%d] = %f\n", 9999999, C[9999999]);
    printf("Serial Vector Multiplication Time: %f seconds\n", multiply_time);

    // Cleanup
    free(A);
    free(B);
    free(C);

    return 0;
}
```

Explanation of Serial Code

The serial code defines two functions, `vector_add()` and `vector_multiply()`, to perform the respective operations on two vectors A and B, storing the result in vector C. It measures the time taken for each operation using the `clock()` function. The vectors are initialized with values such that:

$$A[i] = i + 1, \quad B[i] = N - i \quad \text{for } i = 0 \text{ to } N - 1$$

The time for both operations is printed as output.

Output

The output of the serial summation code is as follows:

```
gcc vector_add_mul.c
./a.out
Vector Addition C[9999999] = 10000001.000000
Serial Vector Addition Time: 0.059371 seconds
Vector Multiplication C[9999999] = 10000000.000000
Serial Vector Multiplication Time: 0.033171 seconds
```

Parallel CUDA Code for Vector Operations

The parallel code implements the same vector addition and multiplication operations but using CUDA to leverage parallel processing capabilities of GPUs. Here is the CUDA code for both operations:

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

#define N 10000000  // Number of elements (10 million)
#define THREADS_PER_BLOCK 256

// CUDA kernel for vector addition
__global__ void vector_add_kernel(double *A, double *B, double *C) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) {
        C[idx] = A[idx] + B[idx];
    }
}

// CUDA kernel for vector multiplication
__global__ void vector_multiply_kernel(double *A, double *B, double *C) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) {
        C[idx] = A[idx] * B[idx];
    }
}

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

    // Allocate memory for the vectors
    A = (double*)malloc(N * sizeof(double));
    B = (double*)malloc(N * sizeof(double));
    C = (double*)malloc(N * sizeof(double));

    // Initialize the vectors with random values
    for (int i = 0; i < N; i++) {
        A[i] = i + 1;
```

```
    B[i] = N - i;
}

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

// Copy input vectors from host to device
cudaMemcpy(d_A, A, N * sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B, N * sizeof(double), cudaMemcpyHostToDevice);

// Launch kernel for vector addition
int blocks = (N + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);

vector_add_kernel<<<blocks, THREADS_PER_BLOCK>>>(d_A, d_B, d_C);

cudaEventRecord(stop);
cudaEventSynchronize(stop);
float add_time = 0.0f;
cudaEventElapsedTime(&add_time, start, stop);

// Copy output vector from device to host
cudaMemcpy(C, d_C, N * sizeof(double), cudaMemcpyDeviceToHost);

printf("CUDA Vector Addition C[%d] = %f\n", N-1, C[N-1]);
printf("CUDA Vector Addition Time: %f seconds\n", add_time / 1000.0);

// Launch kernel for vector multiplication
cudaEventRecord(start);
vector_multiply_kernel<<<blocks, THREADS_PER_BLOCK>>>(d_A, d_B, d_C);
cudaEventRecord(stop);
cudaEventSynchronize(stop);
float multiply_time = 0.0f;
cudaEventElapsedTime(&multiply_time, start, stop);

// Copy output vector from device to host
```

```
    cudaMemcpy(C, d_C, N * sizeof(double), cudaMemcpyDeviceToHost);

    printf("CUDA Vector Multiplication C[%d] = %f\n", N-1, C[N-1]);
    printf("CUDA Vector Multiplication Time: %f seconds\n", multiply_time / 1000

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

    return 0;
}
```

Explanation of CUDA Parallel Code

The parallel CUDA code includes two CUDA kernels, `vector_add_kernel()` and `vector_multiply_kernel()`, to perform vector addition and multiplication in parallel on the GPU. The code divides the work among multiple threads:

$$\text{Thread index} = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}$$

where each thread computes one element of the resulting vector.

The number of blocks is computed based on the total number of elements and the number of threads per block, and the execution time is measured using CUDA events.

```
nvcc vector_add_mul_cuda.cu
./a.out
CUDA Vector Addition C[9999999] = 10000001.000000
CUDA Vector Addition Time: 0.004121 seconds
CUDA Vector Multiplication C[9999999] = 10000000.000000
CUDA Vector Multiplication Time: 0.001644 seconds
```

Performance Results

The execution times for the serial and parallel implementations are as follows:

- Serial Vector Addition Time: 0.059371 seconds

- Serial Vector Multiplication Time: 0.033171 seconds
- CUDA Vector Addition Time: 0.004121 seconds
- CUDA Vector Multiplication Time: 0.001644 seconds

Speedup Calculation

The speedup for each operation is calculated using the formula:

$$\text{Speedup} = \frac{\text{Serial Time}}{\text{Parallel Time}}$$

- Speedup for Vector Addition:

$$\text{Speedup} = \frac{0.059371}{0.004121} \approx 14.4$$

- Speedup for Vector Multiplication:

$$\text{Speedup} = \frac{0.033171}{0.001644} \approx 20.2$$

Conclusion

By utilizing parallel programming with CUDA, the vector addition and multiplication operations show significant performance improvements, with speedups of approximately 14.4x for addition and 20.2x for multiplication compared to the serial implementation.