Problem Statement:

Comparing CUDA and C implementations for matrix operations to determine the impact of CUDA's parallelism on computational speed.

CUDA Code (Google colab {T4 GPU}):

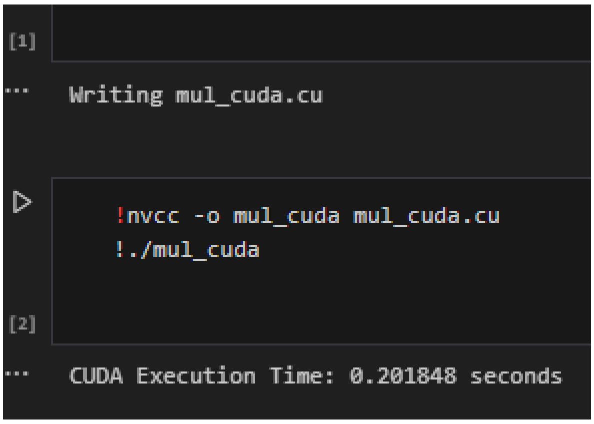
```
%%writefile mul_cuda.cu
#include <iostream>
#include <chrono>

// CUDA Kernel for adding constant to matrix
__global__ void addConstant(float *matrix, float constant, int size) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < size) {
        matrix[idx] += constant;
    }
}

// CUDA Kernel for multiplying matrix by scalar
__global__ void multiplyByScalar(float *matrix, float scalar, int size) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < size) {
        matrix[idx] *= scalar;
    }
}

// CUDA Kernel for matrix multiplication
__global__ void matrixMultiply(float *a, float *b, float *c, int size) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < size && col < size) {
        float sum = 0.0f;
```

```
        for (int i = 0; i < size; ++i) {
            sum += a[row * size + i] * b[i * size + col];
        }
        c[row * size + col] = sum;
    }
}

int main() {
    const int size = 3000;
    const float constant = 5.0f;
    const float scalar = 2.0f;

    // Allocate memory for matrices on host
    float *h_A, *h_B, *h_C;
    h_A = new float[size * size];
    h_B = new float[size * size];
    h_C = new float[size * size];

    // Initialize matrices
    for (int i = 0; i < size * size; ++i) {
        h_A[i] = 1.0f;
        h_B[i] = 2.0f;
        h_C[i] = 0.0f;
    }

    // Allocate memory for matrices on device
    float *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, size * size * sizeof(float));
    cudaMalloc(&d_B, size * size * sizeof(float));
    cudaMalloc(&d_C, size * size * sizeof(float));
```

```
// Copy matrices from host to device
cudaMemcpy(d_A, h_A, size * size * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size * size * sizeof(float), cudaMemcpyHostToDevice);

// Add constant to matrices
addConstant<<<(size * size + 255) / 256, 256>>>(d_A, constant, size * size);
addConstant<<<(size * size + 255) / 256, 256>>>(d_B, constant, size * size);

// Multiply matrices by scalar
multiplyByScalar<<<(size * size + 255) / 256, 256>>>(d_A, scalar, size * size);
multiplyByScalar<<<(size * size + 255) / 256, 256>>>(d_B, scalar, size * size);

// Matrix multiplication
dim3 threadsPerBlock(16, 16);
dim3 numBlocks((size + threadsPerBlock.x - 1) / threadsPerBlock.x,
        (size + threadsPerBlock.y - 1) / threadsPerBlock.y);

auto start = std::chrono::steady_clock::now();

matrixMultiply<<<numBlocks, threadsPerBlock>>>(d_A, d_B, d_C, size);

// Copy result matrix from device to host
cudaMemcpy(h_C, d_C, size * size * sizeof(float), cudaMemcpyDeviceToHost);

// Print execution time
auto end = std::chrono::steady_clock::now();
std::chrono::duration<double> elapsed = end - start;
std::cout << "CUDA Execution Time: " << elapsed.count() << " seconds\n";

// Clean up
delete[] h_A;
```

```
    delete[] h_B;
    delete[] h_C;

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    return 0;
}
```

Execution output:



This CUDA program showcases efficient matrix operations using Google colab's CUDA platform, focusing on **thread-level parallelism** and optimal block configurations. The program begins by initializing two **3000x3000** matrices, h_A and h_B, both filled with 1.0. It proceeds to add 5.0 to each element of the matrices and then multiplies them by a scalar value of 2.0 using CUDA kernels **addConstant** and **multiplyByScalar**. These kernels are executed with a block size of **256** threads per block, calculated according to the matrix size. For the crucial matrix multiplication operation, the **matrixMultiply** kernel is utilized with a block configuration of **16x16 threads** per block. The program also measures and displays the execution time.

C Code (System {CPU}):

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

// Function for adding constant to matrix
```

```
void addConstant(float *matrix, float constant, int size) {
    for (int i = 0; i < size; ++i) {
        matrix[i] += constant;
    }
}

// Function for multiplying matrix by scalar
void multiplyByScalar(float *matrix, float scalar, int size) {
    for (int i = 0; i < size; ++i) {
        matrix[i] *= scalar;
    }
}

// Function for matrix multiplication
void matrixMultiply(float *a, float *b, float *c, int size) {
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            float sum = 0.0f;
            for (int k = 0; k < size; ++k) {
                sum += a[i * size + k] * b[k * size + j];
            }
            c[i * size + j] = sum;
        }
    }
}

int main() {
    const int size = 3000;
    const float constant = 5.0f;
    const float scalar = 2.0f;
```

```
// Allocate memory for matrices on heap
float *h_A = (float *)malloc(size * size * sizeof(float));
float *h_B = (float *)malloc(size * size * sizeof(float));
float *h_C = (float *)malloc(size * size * sizeof(float));

// Initialize matrices
for (int i = 0; i < size * size; ++i) {
    h_A[i] = 1.0f;
    h_B[i] = 2.0f;
    h_C[i] = 0.0f;
}

clock_t start = clock();

// Add constant to matrices
addConstant(h_A, constant, size * size);
addConstant(h_B, constant, size * size);

// Multiply matrices by scalar
multiplyByScalar(h_A, scalar, size * size);
multiplyByScalar(h_B, scalar, size * size);

// Matrix multiplication
matrixMultiply(h_A, h_B, h_C, size);

clock_t end = clock();
double elapsed = ((double) (end - start)) / CLOCKS_PER_SEC;
printf("CPU Execution Time: %f seconds\n", elapsed);

// Free allocated memory
free(h_A);
```

```
    free(h_B);
    free(h_C);

    return 0;
}
```

Execution output:

```
C:\Users\Dell\Documents>gcc -o C_prog C_prog.c

C:\Users\Dell\Documents>C_prog CPU Execution Time: 117.823000 seconds
```

Inference:

The analysis demonstrates that CUDA outperforms C in execution time for matrix operations, showing faster completion of tasks like matrix addition and multiplication. This suggests the efficiency of CUDA's parallel processing for speeding up computations compared to traditional C programming.