COA LAB ASSIGNMENT 3

MATRIX MULTIPLICATION

Group 9:

Ponnanna A H

Priyam Saha

Arihant Garg

Shubh Modi

Tejas Singh Rajput

AUGUST 21, 2023

CUDA CODE FOR MATRIX MULTIPLICATION USING GLOBAL MEMORY:

```
// This program computes a simple version of matrix multiplication
(global)

#include <algorithm>
#include <cassert>
#include <cstdlib>
#include <functional>
#include <iostream>
#include <vector>

using std::cout;
using std::generate;
using std::vector;

__global__ void matrixMul(const int *a, const int *b, int *c, int N)
{
    // Compute each thread's global row and column index
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    // Iterate over row, and down column
    c[row * N + col] = 0;
    for (int k = 0; k < N; k++)
    {
        // Accumulate results for a single element
        c[row * N + col] += a[row * N + k] * b[k * N + col];
    }
}

// Check result on the CPU
void verify_result(vector<int> &a, vector<int> &b, vector<int> &c,
int N)
{
    // For every row...
    for (int i = 0; i < N; i++)
    {
        // For every column...
        for (int j = 0; j < N; j++)
        {
            // For every element in the row-column pair
            int tmp = 0;
            for (int k = 0; k < N; k++)
            {
                // Accumulate the partial results
                tmp += a[i * N + k] * b[k * N + j];
            }

            // Check against the CPU result
            assert(tmp == c[i * N + j]);
        }
    }
}

int main()
{
    // Matrix size of 128 X 128;
    int N = 1 << 7;

    // Size (in bytes) of matrix
    size_t bytes = N * N * sizeof(int);

    // Host vectors
    vector<int> h_a(N * N);
    vector<int> h_b(N * N);
    vector<int> h_c(N * N);

    // Initialize matrices
    generate(h_a.begin(), h_a.end(), []()
             { return rand() % 100; });
    generate(h_b.begin(), h_b.end(), []()
             { return rand() % 100; });

    // Allocate device memory
    int *d_a, *d_b, *d_c;
    cudaMalloc(&d_a, bytes);
    cudaMalloc(&d_b, bytes);
    cudaMalloc(&d_c, bytes);

    // Copy data to the device
    cudaMemcpy(d_a, h_a.data(), bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b.data(), bytes, cudaMemcpyHostToDevice);
    // Threads
    int THREADS = 32;

    // Blocks per grid dimension
    int BLOCKS = N / THREADS;

    dim3 threads(THREADS, THREADS);
    dim3 blocks(BLOCKS, BLOCKS);

    // Launch kernel
    matrixMul<<<blocks, threads>>>(d_a, d_b, d_c, N);

    // Copy back to the host
    cudaMemcpy(h_c.data(), d_c, bytes, cudaMemcpyDeviceToHost);

    // Check result
    verify_result(h_a, h_b, h_c, N);

    cout << "COMPLETED SUCCESSFULLY\n";

    // Free memory on device
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    return 0;
}
```

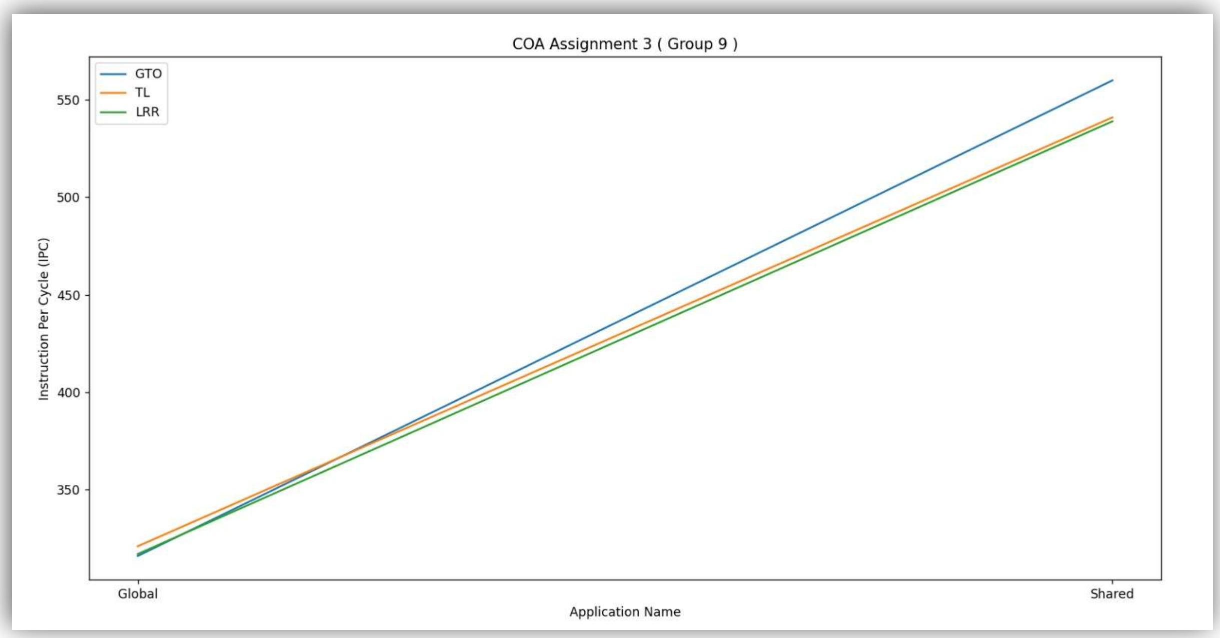
CUDA CODE FOR MATRIX MULTIPLICATION USING SHARED MEMORY:

```
// This program computes a simple version of matrix
multiplication (shared)

#include <iostream>
#include <cassert>
#include <ctime>
#include <cstdlib>
#include <cuda_runtime.h>

const int N = 128;
const int BLOCK_SIZE = 16;

__global__ void matrixMultiplyShared(float *A, float *B,
float *C, int n)
{
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int row = blockIdx.y * blockDim.y + ty;
    int col = blockIdx.x * blockDim.x + tx;

    __shared__ float shared_A[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float shared_B[BLOCK_SIZE][BLOCK_SIZE];

    float sum = 0.0f;

    for (int i = 0; i < n / BLOCK_SIZE; ++i)
    {
        shared_A[ty][tx] = A[row * n + i * BLOCK_SIZE + tx];
        shared_B[ty][tx] = B[(i * BLOCK_SIZE + ty) * n +
col];
        __syncthreads();

        for (int k = 0; k < BLOCK_SIZE; ++k)
        {
            sum += shared_A[ty][k] * shared_B[k][tx];
        }
        __syncthreads();
    }

    C[row * n + col] = sum;
}

void verifyMatrixMultiplication(float *A, float *B, float
*C, int n)
{
    for (int i = 0; i < n; ++i)
    {
        for (int j = 0; j < n; ++j)
        {
            float sum = 0.0f;
            for (int k = 0; k < n; ++k)
            {
                sum += A[i * n + k] * B[k * n + j];
            }
            assert(fabs(C[i * n + j] - sum) < 1e-5);
        }
    }
}

int main()
{
    float *h_A, *h_B, *h_C;
    float *d_A, *d_B, *d_C;
    int matrixSize = N * N * sizeof(float);

    h_A = new float[N * N];
    h_B = new float[N * N];
    h_C = new float[N * N];

    srand(static_cast<unsigned int>(time(0)));
    for (int i = 0; i < N * N; ++i)
    {
        h_A[i] = static_cast<float>(rand() % 100 + 1);
        h_B[i] = static_cast<float>(rand() % 100 + 1);
    }

    cudaMalloc(&d_A, matrixSize);
    cudaMalloc(&d_B, matrixSize);
    cudaMalloc(&d_C, matrixSize);

    cudaMemcpy(d_A, h_A, matrixSize,
cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, matrixSize,
cudaMemcpyHostToDevice);

    dim3 threadsPerBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 numBlocks(N / BLOCK_SIZE, N / BLOCK_SIZE);

    matrixMultiplyShared<<<numBlocks,
threadsPerBlock>>>(d_A, d_B, d_C, N);

    cudaMemcpy(h_C, d_C, matrixSize,
cudaMemcpyDeviceToHost);

    verifyMatrixMultiplication(h_A, h_B, h_C, N);

    std::cout << "Matrix multiplication result is correct."
<< std::endl;

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    delete[] h_A;
    delete[] h_B;
    delete[] h_C;

    return 0;
}
```

RUN TIME FOR VARIOUS CONFIGURATIONS:

Warp Schedular	Implementation Way	GPGPU Simulation Time(in secs)
TL(Two Level)	Global	24
TL(Two Level)	Shared	10
LRR(Loose Round Robin)	Global	27
LRR(Loose Round Robin)	Shared	10
GTO(Greedy Then Others)	Global	25
GTO(Greedy Then Others)	Shared	10

Plot showing the IPC on Y-axis and application name on X-axis, legend:different warp schedulers.



2.STATISTICS:

Greedy Then Other:

Metrics	Global	Shared
L1D_total_cache_accesses	591872	34816
L1D_total_cache_misses	434299	34816
L1D_total_cache_miss_rate	0.7338	1.0000
L2_total_cache_accesses	280576	34816
L2_total_cache_misses	2048	2048
L2_total_cache_miss_rate	0.0073	0.0588

Loose Round Robin:

Metrics	Global	Shared
L1D_total_cache_accesses	591872	34816
L1D_total_cache_misses	436737	34816
L1D_total_cache_miss_rate	0.7379	1.0000
L2_total_cache_accesses	280576	34816
L2_total_cache_misses	2048	2048
L2_total_cache_miss_rate	0.0073	0.0588

Two Level:

Metrics	Global	Shared
L1D_total_cache_accesses	591872	34816
L1D_total_cache_misses	435018	34816
L1D_total_cache_miss_rate	0.7350	1.0000
L2_total_cache_accesses	280576	34816
L2_total_cache_misses	2048	2048
L2_total_cache_miss_rate	0.0073	0.0588

JUSTIFICATION:

We observed that Global has higher cache accesses for both L1D and L2 caches as compared to shared memory for all three warp schedulers- Loose Round Robin, Greedy Then Others and Two-Level.

We also observed that the miss rate of both L1D and L2 cache is lower in global memory than shared memory for all three warp schedulers.