**Simple Project – Optimizing Memory Usage in CUDA for Matrix Multiplication using cmake**

**Objective**

To understand and apply memory optimization techniques in CUDA using shared memory, coalesced global memory access, and constant memory, by implementing and comparing performance of multiple CUDA kernels for matrix multiplication.

**Background**

Matrix multiplication is a classic CUDA problem, but naive implementations often lead to poor performance due to inefficient memory access. This lab focuses on:

* Global memory access (baseline)
* Shared memory tiling
* Using constant memory
* Memory coalescing

**Files to be Created**

A screen shot of a computer program

AI-generated content may be incorrect.

**Step 1: Define Header File matmul\_utils.h**

#pragma once

void matMulNaive(const float\* A, const float\* B, float\* C, int N);

void matMulShared(const float\* A, const float\* B, float\* C, int N);

void matMulConst(const float\* A, float\* C, int N);

void fillMatrix(float\* mat, int N);

void printMatrix(const float\* mat, int N);

**Step 2: Implement Utility Functions matmul\_utils.cu**

#include <iostream>

#include <cstdlib>

#include "matmul\_utils.h"

void fillMatrix(float\* mat, int N) {

for (int i = 0; i < N \* N; ++i)

mat[i] = static\_cast<float>(rand()) / RAND\_MAX;

}

void printMatrix(const float\* mat, int N) {

for (int i = 0; i < std::min(N, 8); ++i) {

for (int j = 0; j < std::min(N, 8); ++j)

std::cout << mat[i \* N + j] << " ";

std::cout << "\n";

}

}

**Step 3: Naive Kernel matmul\_naive.cu**

#include "matmul\_utils.h"

\_\_global\_\_ void matMulKernelNaive(const float\* A, const float\* B, float\* C, int N) {

int row = blockIdx.y \* blockDim.y + threadIdx.y;

int col = blockIdx.x \* blockDim.x + threadIdx.x;

float sum = 0;

if (row < N && col < N) {

for (int k = 0; k < N; ++k)

sum += A[row \* N + k] \* B[k \* N + col];

C[row \* N + col] = sum;

}

}

void matMulNaive(const float\* A, const float\* B, float\* C, int N) {

float \*d\_A, \*d\_B, \*d\_C;

size\_t size = N \* N \* sizeof(float);

cudaMalloc(&d\_A, size);

cudaMalloc(&d\_B, size);

cudaMalloc(&d\_C, size);

cudaMemcpy(d\_A, A, size, cudaMemcpyHostToDevice);

cudaMemcpy(d\_B, B, size, cudaMemcpyHostToDevice);

dim3 threads(16, 16);

dim3 blocks((N + 15) / 16, (N + 15) / 16);

matMulKernelNaive<<<blocks, threads>>>(d\_A, d\_B, d\_C, N);

cudaMemcpy(C, d\_C, size, cudaMemcpyDeviceToHost);

cudaFree(d\_A); cudaFree(d\_B); cudaFree(d\_C);

}

**Step 4: Shared Memory Kernel matmul\_shared.cu**

#include "matmul\_utils.h"

\_\_global\_\_ void matMulKernelShared(const float\* A, const float\* B, float\* C, int N) {

\_\_shared\_\_ float tileA[16][16];

\_\_shared\_\_ float tileB[16][16];

int row = threadIdx.y + blockDim.y \* blockIdx.y;

int col = threadIdx.x + blockDim.x \* blockIdx.x;

float sum = 0;

for (int tile = 0; tile < N / 16; ++tile) {

tileA[threadIdx.y][threadIdx.x] = A[row \* N + tile \* 16 + threadIdx.x];

tileB[threadIdx.y][threadIdx.x] = B[(tile \* 16 + threadIdx.y) \* N + col];

\_\_syncthreads();

for (int k = 0; k < 16; ++k)

sum += tileA[threadIdx.y][k] \* tileB[k][threadIdx.x];

\_\_syncthreads();

}

C[row \* N + col] = sum;

}

void matMulShared(const float\* A, const float\* B, float\* C, int N) {

float \*d\_A, \*d\_B, \*d\_C;

size\_t size = N \* N \* sizeof(float);

cudaMalloc(&d\_A, size); cudaMemcpy(d\_A, A, size, cudaMemcpyHostToDevice);

cudaMalloc(&d\_B, size); cudaMemcpy(d\_B, B, size, cudaMemcpyHostToDevice);

cudaMalloc(&d\_C, size);

dim3 threads(16, 16);

dim3 blocks(N / 16, N / 16);

matMulKernelShared<<<blocks, threads>>>(d\_A, d\_B, d\_C, N);

cudaMemcpy(C, d\_C, size, cudaMemcpyDeviceToHost);

cudaFree(d\_A); cudaFree(d\_B); cudaFree(d\_C);

}

**Step 5: Constant Memory Kernel matmul\_const.cu**

#include "matmul\_utils.h"

\_\_constant\_\_ float Bc[1024 \* 1024]; // 1024x1024 max

\_\_global\_\_ void matMulKernelConst(const float\* A, float\* C, int N) {

int row = blockIdx.y \* blockDim.y + threadIdx.y;

int col = blockIdx.x \* blockDim.x + threadIdx.x;

float sum = 0;

if (row < N && col < N) {

for (int k = 0; k < N; ++k)

sum += A[row \* N + k] \* Bc[k \* N + col];

C[row \* N + col] = sum;

}

}

void matMulConst(const float\* A, float\* C, int N) {

float \*d\_A, \*d\_C;

size\_t size = N \* N \* sizeof(float);

cudaMalloc(&d\_A, size);

cudaMalloc(&d\_C, size);

cudaMemcpy(d\_A, A, size, cudaMemcpyHostToDevice);

// Assume matrix B is static, provided by host (const use case)

static float Bhost[1024 \* 1024]; // make sure N ≤ 1024

fillMatrix(Bhost, N);

cudaMemcpyToSymbol(Bc, Bhost, size);

dim3 threads(16, 16);

dim3 blocks((N + 15) / 16, (N + 15) / 16);

matMulKernelConst<<<blocks, threads>>>(d\_A, d\_C, N);

cudaMemcpy(C, d\_C, size, cudaMemcpyDeviceToHost);

cudaFree(d\_A); cudaFree(d\_C);

}

**Step 6: Main Program main.cu**

#include <iostream>

#include <chrono>

#include "matmul\_utils.h"

int main() {

const int N = 512;

float \*A = new float[N \* N];

float \*B = new float[N \* N];

float \*C = new float[N \* N];

fillMatrix(A, N);

fillMatrix(B, N);

std::cout << "[Naive Kernel]\n";

auto start = std::chrono::high\_resolution\_clock::now();

matMulNaive(A, B, C, N);

auto end = std::chrono::high\_resolution\_clock::now();

std::cout << "Time: "

<< std::chrono::duration<double, std::milli>(end - start).count()

<< " ms\n";

std::cout << "[Shared Memory Kernel]\n";

start = std::chrono::high\_resolution\_clock::now();

matMulShared(A, B, C, N);

end = std::chrono::high\_resolution\_clock::now();

std::cout << "Time: "

<< std::chrono::duration<double, std::milli>(end - start).count()

<< " ms\n";

std::cout << "[Constant Memory Kernel]\n";

start = std::chrono::high\_resolution\_clock::now();

matMulConst(A, C, N);

end = std::chrono::high\_resolution\_clock::now();

std::cout << "Time: "

<< std::chrono::duration<double, std::milli>(end - start).count()

<< " ms\n";

delete[] A;

delete[] B;

delete[] C;

return 0;

}

**CMakeLists.txt**

cmake\_minimum\_required(VERSION 3.10)

project(CudaMemoryOpt LANGUAGES CUDA CXX)

set(CMAKE\_CUDA\_STANDARD 14)

add\_executable(CudaMemoryOpt

main.cu

matmul\_naive.cu

matmul\_shared.cu

matmul\_const.cu

matmul\_utils.cu

)

# Allow larger constant memory usage

set\_target\_properties(CudaMemoryOpt PROPERTIES CUDA\_SEPARABLE\_COMPILATION ON)

1. **Create a build directory:**

mkdir build

cd build

1. Run CMake to configure:

cmake ..

1. Build the project:

cmake --build .

cuda\_project.exe

This will compile main.cpp and kernel.cu and produce an executable cuda\_project.exe

**Expected Output**

[Naive Kernel]

Time: 85.6312 ms

[Shared Memory Kernel]

Time: 21.3948 ms

[Constant Memory Kernel]

Time: 18.7623 ms

**Explanation**

* **Naive Kernel**:
  + Each thread directly accesses global memory without optimization.
  + Slowest due to non-coalesced memory accesses and redundant loads.
* **Shared Memory Kernel**:
  + Uses block-level shared memory to reduce redundant global memory accesses.
  + Faster than naive due to better memory access patterns and reduced latency.
* **Constant Memory Kernel**:
  + Uses \_\_constant\_\_ memory (which is cached and optimized for broadcast).
  + Very efficient when the same data (e.g., matrix B) is reused frequently by many threads.
  + Fastest if properly implemented for specific workloads.