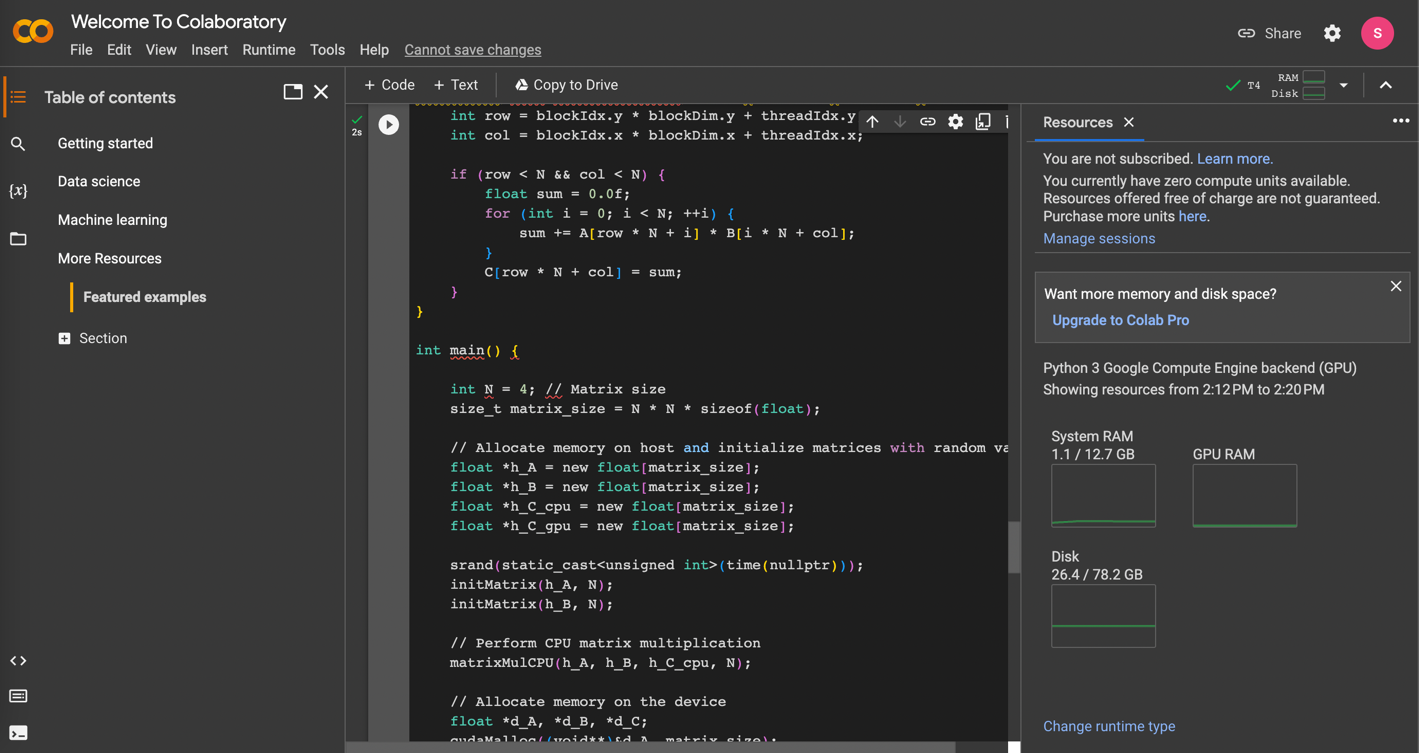
**Assignment-2**

**Assigned GPU in Google Collab:**

****

**A screenshot of a computer

Description automatically generated**

**Comparison of CPU Matrix Multiplication with GPU Matrix Multiplication using iterative method:**

**A screenshot of a computer program

Description automatically generated**

**Native GPU version of matrix multiplication with CPU Matrix multiplication for comparison:**

%%cu

#include <iostream>

#include <cstdlib>

#include <ctime>

#include <cmath>

#include <chrono> // For CPU timing

#include <cuda\_runtime.h>

#define TILE\_SIZE 16

// Function to initialize a matrix with random values

void initMatrix(float \*matrix, int N) {

for (int i = 0; i < N; ++i) {

for (int j = 0; j < N; ++j) {

matrix[i \* N + j] = static\_cast<float>(rand()) / RAND\_MAX;

}

}

}

// CPU Matrix Multiplication

void matrixMulCPU(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];

}

C[i \* N + j] = sum;

}

}

}

// Function to display a matrix to the console

void displayMatrix(float \*matrix, int N) {

for (int i = 0; i < N; ++i) {

for (int j = 0; j < N; ++j) {

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

}

std::cout << std::endl;

}

}

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

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

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

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

float sum = 0.0f;

for (int i = 0; i < N; ++i) {

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

}

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

}

}

int main() {

int N = 4; // Matrix size

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

//Max Shared Memory per block

int maxSharedMemoryPerBlock;

cudaDeviceGetAttribute(&maxSharedMemoryPerBlock, cudaDevAttrMaxSharedMemoryPerBlock,0);

std::cout << "Max Shared Memory Per Block: " << maxSharedMemoryPerBlock << " bytes" << std::endl;

// Find and print the maximum threads per block

int maxThreadsPerBlock;

cudaDeviceProp deviceProp;

cudaGetDeviceProperties(&deviceProp, 0); // Assuming device 0

maxThreadsPerBlock = deviceProp.maxThreadsPerBlock;

std::cout << "Max Threads Per Block: " << maxThreadsPerBlock << std::endl;

// Allocate memory on host and initialize matrices with random values

float \*h\_A = new float[matrix\_size];

float \*h\_B = new float[matrix\_size];

float \*h\_C\_cpu = new float[matrix\_size];

float \*h\_C\_gpu = new float[matrix\_size];

srand(static\_cast<unsigned int>(time(nullptr)));

initMatrix(h\_A, N);

initMatrix(h\_B, N);

// Create CPU timers for CPU timing

std::chrono::high\_resolution\_clock::time\_point cpu\_start, cpu\_stop;

// Start measuring CPU time

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

// Perform CPU matrix multiplication

matrixMulCPU(h\_A, h\_B, h\_C\_cpu, N);

// Stop measuring CPU time

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

auto cpu\_duration = std::chrono::duration\_cast<std::chrono::milliseconds>(cpu\_stop - cpu\_start);

float cpu\_seconds = cpu\_duration.count() ; // Convert to seconds

std::cout << "CPU Execution Time: " << cpu\_seconds << "ms" << std::endl;

// Allocate memory on the device

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

cudaMalloc((void\*\*)&d\_A, matrix\_size);

cudaMalloc((void\*\*)&d\_B, matrix\_size);

cudaMalloc((void\*\*)&d\_C, matrix\_size);

// Copy data from host to device

cudaMemcpy(d\_A, h\_A, matrix\_size, cudaMemcpyHostToDevice);

cudaMemcpy(d\_B, h\_B, matrix\_size, cudaMemcpyHostToDevice);

// Define thread block and grid dimensions

/\* dim3 blockDim(16, 16);

dim3 gridDim((N + blockDim.x - 1) / blockDim.x, (N + blockDim.y - 1) / blockDim.y); \*/

// Define thread block and grid dimensions

dim3 blockDim(TILE\_SIZE, TILE\_SIZE); //Block Dimensions

dim3 gridDim((N + TILE\_SIZE - 1) / TILE\_SIZE, (N + TILE\_SIZE - 1) / TILE\_SIZE); //Grid Dimensions

// Create CUDA events for GPU timing

cudaEvent\_t gpu\_start, gpu\_stop;

cudaEventCreate(&gpu\_start);

cudaEventCreate(&gpu\_stop);

// Start measuring GPU time

cudaEventRecord(gpu\_start);

// Launch the kernel

matrixMulNaive<<<gridDim, blockDim>>>(d\_A, d\_B, d\_C, N);

// Stop measuring GPU time

cudaEventRecord(gpu\_stop);

cudaEventSynchronize(gpu\_stop);

// Calculate and print GPU execution time in seconds

float gpu\_milliseconds = 0.0f;

cudaEventElapsedTime(&gpu\_milliseconds, gpu\_start, gpu\_stop);

float gpu\_seconds = gpu\_milliseconds ; // Convert to seconds

std::cout << "GPU Execution Time: " << gpu\_seconds << "ms" << std::endl;

// Copy the result back to the host

cudaMemcpy(h\_C\_gpu, d\_C, matrix\_size, cudaMemcpyDeviceToHost);

// Display the output matrix

/\* std::cout << " GPU Output Matrix:" << std::endl;

displayMatrix(h\_C\_gpu, N);

std::cout << " \n------------End of GPU Matrix----------\n" << std::endl;

std::cout << " CPU Output Matrix:" << std::endl;

displayMatrix(h\_C\_gpu, N);

std::cout << "\n ------------End of CPU Matrix---------- \n" << std::endl; \*/

// Compare GPU and CPU results for verification

bool verificationPassed = true;

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

if(fabs(h\_C\_gpu[i] - h\_C\_cpu[i]) > 1e-5){

std::cout << "\n Verification Failed!! At " << i <<" "<<h\_C\_gpu[i] << " " <<h\_C\_cpu[i] << std::endl;

verificationPassed = false;

break;

}

}

if(verificationPassed){

std::cout << "\n Verification Sucess!! \n" <<std::endl;

}

// Free device memory

cudaFree(d\_A);

cudaFree(d\_B);

cudaFree(d\_C);

// Free host memory

delete[] h\_A;

delete[] h\_B;

delete[] h\_C\_cpu;

delete[] h\_C\_gpu;

return 0;

}

**Tiled GPU version of matrix multiplication with CPU Matrix multiplication for comparison:**

%%cu

#include <iostream>

#include <cstdlib>

#include <ctime>

#include <cmath>

#include <chrono> // For CPU timing

#define TILE\_SIZE 16

// Function to initialize a matrix with random values

void initMatrix(float \*matrix, int N) {

for (int i = 0; i < N; ++i) {

for (int j = 0; j < N; ++j) {

matrix[i \* N + j] = static\_cast<float>(rand()) / RAND\_MAX;

}

}

}

// CPU Matrix Multiplication

void matrixMulCPU(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];

}

C[i \* N + j] = sum;

}

}

}

// Function to display a matrix to the console

void displayMatrix(float \*matrix, int N) {

for (int i = 0; i < N; ++i) {

for (int j = 0; j < N; ++j) {

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

}

std::cout << std::endl;

}

}

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

int tx = threadIdx.x;

int ty = threadIdx.y;

int bx = blockIdx.x;

int by = blockIdx.y;

int row = by \* TILE\_SIZE + ty;

int col = bx \* TILE\_SIZE + tx;

\_\_shared\_\_ float shared\_A[TILE\_SIZE][TILE\_SIZE];

\_\_shared\_\_ float shared\_B[TILE\_SIZE][TILE\_SIZE];

float sum = 0.0f;

for (int i = 0; i < N / TILE\_SIZE; ++i) {

shared\_A[ty][tx] = A[row \* N + (i \* TILE\_SIZE + tx)];

shared\_B[ty][tx] = B[(i \* TILE\_SIZE + ty) \* N + col];

\_\_syncthreads();

for (int k = 0; k < TILE\_SIZE; ++k) {

sum += shared\_A[ty][k] \* shared\_B[k][tx];

}

\_\_syncthreads();

}

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

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

}

}

int main() {

int N = 16; // Matrix size (power of 2)

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

//Max Shared Memory per block

int maxSharedMemoryPerBlock;

cudaDeviceGetAttribute(&maxSharedMemoryPerBlock, cudaDevAttrMaxSharedMemoryPerBlock,0);

std::cout << "Max Shared Memory Per Block: " << maxSharedMemoryPerBlock << " bytes" << std::endl;

// Find and print the maximum threads per block

int maxThreadsPerBlock;

cudaDeviceProp deviceProp;

cudaGetDeviceProperties(&deviceProp, 0); // Assuming device 0

maxThreadsPerBlock = deviceProp.maxThreadsPerBlock;

std::cout << "Max Threads Per Block: " << maxThreadsPerBlock << std::endl;

// Allocate memory on host and initialize matrices with random values

float \*h\_A = new float[matrix\_size];

float \*h\_B = new float[matrix\_size];

float \*h\_C\_cpu = new float[matrix\_size];

float \*h\_C\_gpu = new float[matrix\_size];

srand(static\_cast<unsigned int>(time(nullptr)));

initMatrix(h\_A, N);

initMatrix(h\_B, N);

// Create CPU timers for CPU timing

std::chrono::high\_resolution\_clock::time\_point cpu\_start, cpu\_stop;

// Start measuring CPU time

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

// Perform CPU matrix multiplication

matrixMulCPU(h\_A, h\_B, h\_C\_cpu, N);

// Stop measuring CPU time

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

auto cpu\_duration = std::chrono::duration\_cast<std::chrono::milliseconds>(cpu\_stop - cpu\_start);

float cpu\_seconds = cpu\_duration.count() ; // Convert to seconds

std::cout << "CPU Execution Time: " << cpu\_seconds << "ms" << std::endl;

// Allocate memory on the device

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

cudaMalloc((void\*\*)&d\_A, matrix\_size);

cudaMalloc((void\*\*)&d\_B, matrix\_size);

cudaMalloc((void\*\*)&d\_C, matrix\_size);

// Copy data from host to device

cudaMemcpy(d\_A, h\_A, matrix\_size, cudaMemcpyHostToDevice);

cudaMemcpy(d\_B, h\_B, matrix\_size, cudaMemcpyHostToDevice);

// Define thread block and grid dimensions

dim3 blockDim(TILE\_SIZE, TILE\_SIZE);

dim3 gridDim((N + TILE\_SIZE - 1) / TILE\_SIZE, (N + TILE\_SIZE - 1) / TILE\_SIZE);

// Create CUDA events for GPU timing

cudaEvent\_t gpu\_start, gpu\_stop;

cudaEventCreate(&gpu\_start);

cudaEventCreate(&gpu\_stop);

// Start measuring GPU time

cudaEventRecord(gpu\_start);

// Launch the kernel

matrixMulTiled<<<gridDim, blockDim>>>(d\_A, d\_B, d\_C, N);

// Stop measuring GPU time

cudaEventRecord(gpu\_stop);

cudaEventSynchronize(gpu\_stop);

// Calculate and print GPU execution time in seconds

float gpu\_milliseconds = 0.0f;

cudaEventElapsedTime(&gpu\_milliseconds, gpu\_start, gpu\_stop);

float gpu\_seconds = gpu\_milliseconds ; // Convert to seconds

std::cout << "GPU Execution Time: " << gpu\_seconds << "ms" << std::endl;

// Copy the result back to the host

cudaMemcpy(h\_C\_gpu, d\_C, matrix\_size, cudaMemcpyDeviceToHost);

// Display the output matrix

std::cout << " GPU Output Matrix:" << std::endl;

displayMatrix(h\_C\_gpu, N);

std::cout << " \n------------End of GPU Matrix----------\n" << std::endl;

std::cout << " CPU Output Matrix:" << std::endl;

displayMatrix(h\_C\_gpu, N);

std::cout << "\n ------------End of CPU Matrix---------- \n" << std::endl;

// Compare GPU and CPU results for verification

bool verificationPassed = true;

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

if(fabs(h\_C\_gpu[i] - h\_C\_cpu[i]) > 1e-5){

std::cout << "\n Verification Failed!! At " << i <<" "<<h\_C\_gpu[i] << " " <<h\_C\_cpu[i] << std::endl;

verificationPassed = false;

break;

}

}

if(verificationPassed){

std::cout << "\n Verification Sucess!! \n" <<std::endl;

}

// Free device memory

cudaFree(d\_A);

cudaFree(d\_B);

cudaFree(d\_C);

// Free host memory

delete[] h\_A;

delete[] h\_B;

delete[] h\_C\_cpu;

delete[] h\_C\_gpu;

return 0;

}

**Note:**

Plots of execution time with input sizes of 2 \* 2, 4 \* 4, ..., 8192 \* 8192 for the GPU versions (naive and tiled) Vs CPU versions of the matrix multiplication can be found here in this link below:

<https://sjsu0-my.sharepoint.com/:x:/g/personal/saikashyap_kurella_sjsu_edu/EWZpul4sFb1JuZlCd7yZfooBVA94Gkfx5QBZjZgk21h1iQ?e=OKe3Br>

**Excessive shared memory scenario:**

If we have an excessive amount of shared memory compared to your block size in a CUDA kernel, it means you have more shared memory available than what is required by the threads in the block. In such a scenario, you can optimize your kernel to make more efficient use of shared memory and other resources. Here are some possible optimizations and explanations for their benefits:

1. Increase Block Size:

* You can increase the number of threads per block to better utilize the available shared memory and other resources which might benefit in the following ways:
  + Improved thread-level parallelism: More threads can be executed in parallel, which can help hide memory access latency and increase GPU utilization.
  + Better occupancy: A larger block size can lead to higher occupancy, which can improve overall GPU performance.

2. Increase Data Reuse:

* Take advantage of the extra shared memory to store and reuse additional data that can help optimize your algorithm. This method has the following advantages:
* Reduced global memory accesses: By caching more data in shared memory, you can minimize global memory accesses, which tend to be slower.
* Improved memory access patterns: Caching additional data can help you read and write data in a more coalesced manner, reducing memory access latency.

3. Parallelize Additional Computations:

* If your kernel has unused shared memory, you can parallelize additional computations that can benefit your algorithm by increasing throughput by running additional computations in parallel can enhance the throughput of your algorithm and make better use of available GPU resources.

4. Optimize for Algorithmic Efficiency:

* Use the extra shared memory to optimize the core logic of your algorithm, such as reducing redundant calculations or improving data access patterns which can improve the efficiency of algorithms by leveraging the additional shared memory for optimization which can lead to more efficient algorithms with reduced execution time.

5. Load Data for Future Iterations:

* If your kernel has multiple iterations or stages, you can preload data into shared memory for future iterations to reduce global memory accesses. This method can reduce memory latency by caching data in shared memory for future iterations thereby help reduce memory latency and improve overall execution time.

**Increased Block Size in the previous implementation:**

#include <iostream>

#include <cstdlib>

#include <ctime>

#include <cmath>

#include <cuda\_runtime.h>

// Function to initialize a matrix with random values

void initMatrix(float \*matrix, int N) {

for (int i = 0; i < N; ++i) {

for (int j = 0; j < N; ++j) {

matrix[i \* N + j] = static\_cast<float>(rand()) / RAND\_MAX;

}

}

}

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

// The naive matrix multiplication kernel remains the same as before

// ...

}

int main() {

int N = 512; // Matrix size (power of 2)

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

// Allocate memory on host and initialize matrices with random values

float \*h\_A = new float[matrix\_size];

float \*h\_B = new float[matrix\_size];

float \*h\_C = new float[matrix\_size];

srand(static\_cast<unsigned int>(time(nullptr));

initMatrix(h\_A, N);

initMatrix(h\_B, N);

// Allocate memory on the device

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

cudaMalloc((void\*\*)&d\_A, matrix\_size);

cudaMalloc((void\*\*)&d\_B, matrix\_size);

cudaMalloc((void\*\*)&d\_C, matrix\_size);

// Copy data from host to device

cudaMemcpy(d\_A, h\_A, matrix\_size, cudaMemcpyHostToDevice);

cudaMemcpy(d\_B, h\_B, matrix\_size, cudaMemcpyHostToDevice);

// Increase the block size

dim3 blockDim(32, 32); // Increase the number of threads per block

// Define grid dimensions

dim3 gridDim((N + blockDim.x - 1) / blockDim.x, (N + blockDim.y - 1) / blockDim.y);

// Launch the kernel

matrixMulNaive<<<gridDim, blockDim>>>(d\_A, d\_B, d\_C, N);

// Copy the result back to the host

cudaMemcpy(h\_C, d\_C, matrix\_size, cudaMemcpyDeviceToHost);

// Free device memory

cudaFree(d\_A);

cudaFree(d\_B);

cudaFree(d\_C);

// Free host memory

delete[] h\_A;

delete[] h\_B;

delete[] h\_C;

return 0;

}