202321128

Jeryl Salas

Meng Ai Program

CS 239 Exercise 2

I. Introduction

In the last exercise, we covered the basics of CUDA programming from launching kernels, to getting the GPU's properties to properly identify the optimal block dimensions for matrix addition, and we also learned to compare the performance of these kernels. This time we're going deeper into parallel computing by improving the performance of our parallel program through memory hierarchies. We learned from the previous lectures the global memory is expansive and yet slow to access to compared to shared memory where it offers localized and fast-access cache that can significantly improve the efficiency of our parallel algorithms. We aim to compare two methods of performing matrix multiplications: one using only global memory, and another using both global and shared memory. The guide for the codes and illustrations would mostly come from Kirk, D. B., et al. (2010).

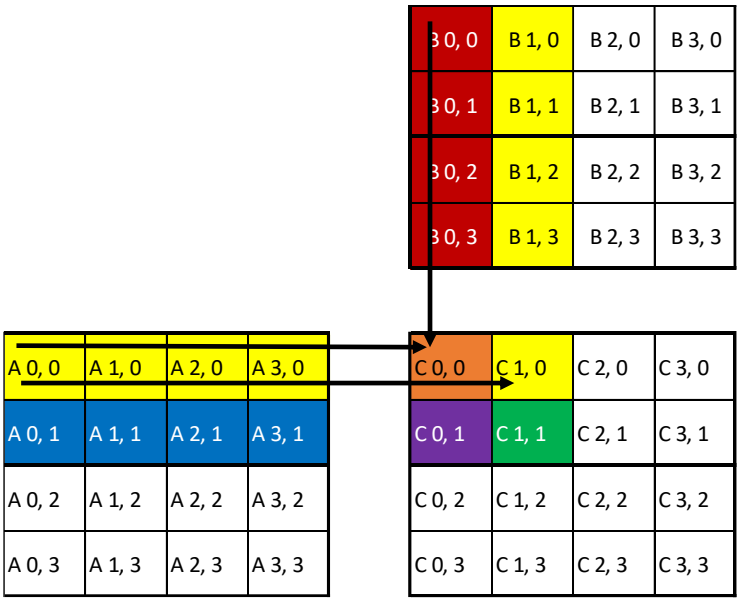

Figure 1: Matrix multiplication using tiling method for shared memory

II. Objectives

Understand concepts of memory hierarchy in parallel computing to enhance performance by utilizing diverse memory types, optimizing access times, patterns, and latency tolerance, while developing skills to design parallel programs considering memory types and hardware constraints.

III. Methodology

The exercise conducted involves performing matrix multiplication where we take two matrices, A and B, to produce an output matrix C which is the product of those two matrices. Each element of matrix C is the dot product of the elements of matrix B and C in their respective assigned row and column. The number of matrix A's columns should equal to the matrix B's number of rows. Thus, the equation for this matrix operation is as follows:

$$C_{nm} = A_{nk} + B_{km} \quad (1)$$

The matrices used in this experiment involve only square matrices with float values but realistically can be used for rectangular matrices. The matrix sizes would involve multiples of 32 which could allow us to better utilize the GPU's computational resources. We were also suggested to use 2D thread of blocks for better efficiency and thus our variable **TILE_SIZE** would be 32. The variables m, n, and k will serve as our variables for defining the sizes of the matrices. *See eqn. 1*.

a. Getting Device Properties

The initial section of our CUDA code involves getting device properties of our GPU. We do this every time to make informed decisions in terms of memory utilization, thread organization, and kernel optimization strategies.

b. Generate matrix A & B and initialize matrix sizes

This code starts by randomly generating matrix A and B using float numbers with a range of $[0, 100]$. The matrix sizes are then defined in an array before starting the computations. The starting matrix size would be 32 and would increase as multiples of 32 for each iteration, i, which gives us an expression of $m, n, k = 32 * i$ where variables n and k is the size of matrix A and variables $k\ and\ m$ is the size of matrix B. We also initialize **num_operations** and **num_globmem_acc** that will serve as our matrices that will store operations and access counts per element of matrix C.

c. Launch global memory and shared memory kernels

We define first the block and grid sizes on our kernels using the **TILE_SIZE** which is set to 32. We then launch **matmul_rec_glob** which is the kernel used to perform matrix multiplication using global memory. We then take the duration time and

compute the CGMA ratio. We launch the **matmul_rec_shar** this time which is a kernel used to compute matrix multiplication using shared memory. We compute the CGMA ratio as well as the duration time.

We repeat these steps on all the matrix sizes and print them on a table. Compute to Global Memory Access ratio (CGMA) is a metric used to evaluate the performance of CUDA kernels as stated from Kirk, D. B., et al. (2010). A higher CGMA would indicate that the kernel is computing more than it is accessing from the Global memory which is a good indication of a better performance since global memory access in GPUs is slower than accessing memory locally. In our case, we manually count the operations and global accesses on each kernel and compute the CGMA ration with this formula:

$$CGMA\ ratio = \frac{number\ of\ operations}{number\ of\ global\ accesses} \qquad (2)$$

IV. Results and Discussion

In this section, we will go over the results of both kernels using global memory and shared memory. We won't present the entire results since it is too many. Instead, we will present the results in a line chart.

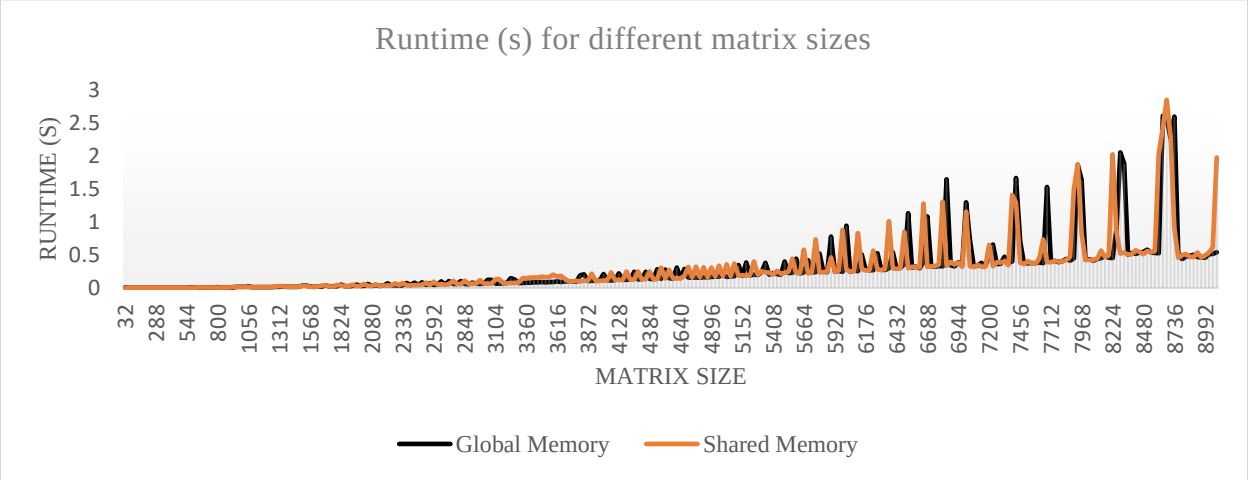

Figure 2: Matrix multiplication runtimes for both global memory and shared memory kernels

```
Device Name: NVIDIA GeForce RTX 2050
  Max threads per block: 1024
  Compute capability: 8.6
  Total global memory: 4294443008 bytes
  Memory Clock Rate (KHz): 7001000

-------------------------Device 0 performance-------------------------

Matrix SizeGlobal Memory Runtime(sec)Shared Memory Runtime(sec) Global Memory CGMA ratio Shared Memory CGMA ratio
-----------------------------------------------------------------------------------------------------------------
   32               0.0008604                9.01e-05                      1                         1
   64               0.0001922                9.63e-05                      1                         1
   96               0.0002167               0.0001341                      1                         1
  128               0.0002914               0.0001711                      1                         1
  160               0.0005852                  0.0004                      1                         1
  192               0.0009878               0.0006986                      1                         1
  224                0.001272               0.0010752                      1                         1
  256                0.001424                0.001255                      1                         1
  288               0.0022559               0.0020565                      1                         1
  320               0.0032197               0.0026731                      1                         1
  352               0.0038923               0.0033422                      1                         1
  384               0.0044019               0.0040896                      1                         1
  416               0.0058582               0.0054358                      1                         1
  448               0.0077327               0.0069475                      1                         1
  480               0.0092689               0.0085465                      1                         1
  512               0.0105789               0.0096964                      1                         1
  544               0.0134061               0.0122771                      1                         1
  576               0.0159192                 0.01435                      1                         1
  608               0.0180402               0.0166107                      1                         1
  640               0.0210877               0.0189536                      1                         1
  672               0.0241732               0.0222991                      1                         1
  704               0.0277901               0.0258487                      1                         1
  736               0.0315459               0.0296396                      1                         1
  768               0.0344634               0.0328083                      1                         1
  800               0.0395318               0.0378619                      1                         1
```

Figure 3: Matrix multiplication using tiling method for shared memory

Matrix Size	Runtime (seconds)	
	Global Memory	Shared Memory
32	0.0048803	0.0003193
64	0.0003039	0.0002731
96	0.0004475	0.0003926
128	0.000436	0.0002899
160	0.0005408	0.0005505
192	0.0008388	0.0009057
224	0.0005608	0.000491
…	…	…
9088	0.539061	1.97175
TOTAL AVERAGE:	0.289444153	0.285041521

Table 1: Matrix multiplication runtime results from global memory and shared memory kernels

From figure 2, we can see how the runtime of both kernels are not that far off which is far from our initial expectations that the shared memory kernel would perform better than the global memory kernel since the shared memory function technically has direct global memory accesses in the loop but it still does involve loading elements from the global memory to the

shared memory and thus the CGMA ratio for shared memory kernel is the same as for global memory kernel which is 1 and hence, our slightly disappointing results.

Global memory	Shared memory
One multiplication	One multiplication
One addition	One addition
Accessing A[i] and B[i]	Accessing A[i] and B[i]

Table 2: Analysis for our CGMA ratio computation

As we can see from table 2, it shows that global and shared memory kernels do two operations per element and also two accesses which is both from matrix A and B where in the shared memory kernel you still have to load matrix A and B to your shared matrix s_A and s_B. The only contributing factor for shared matrix kernel is that you don't have to get values from the global memory every computation, you can just get it from the shared matrices which should result to reduced access latency. As for this exercise, I would like to explore more on what is causing the shared memory kernel to be slower than expected.

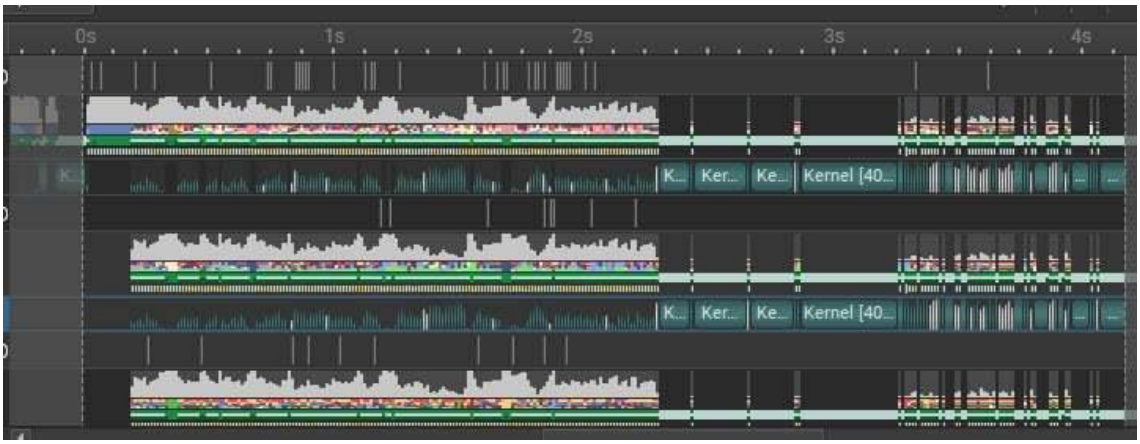

Figure 4: Example of Nsight compute timeline

V. Future study

For future experiments, I would like to see how shared memory kernel can be truly exploited. The current results for this exercise might just be an isolated case and may also come from hardware limitations. As stated from Kirk, D. B., et al. (2010), it's essential for CUDA programmers to be aware to the of the limited sizes of shared memory and other types of memory in CUDA architecture as these memory types have implementation-dependent capacities meaning when you create algorithms that exceed these capacities can become limiting factors to the number of threads that can execute simultaneously on each

Streaming Multi-processor (SM). I would also like to learn to use advanced profiling tools such as Nsight Compute and Nsight Systems as these enable detailed performance analysis at the kernel level. In that way we can find potential bottlenecks in our kernels that can allow us to optimize our code for better efficiency. I was already able to use Nsight compute but I wasn't able to understand the vast amounts of information that it gives out.

VI. References and Code

- Kirk, D. B., & Hwu, W.-m. W. (2010). *Programming Massively Parallel Processors: A Hands-on Approach*. Burlington, MA: Morgan Kaufmann Publishers.

- Nvidia Developer Zone. (n.d.). CUDA Toolkit Documentation: CUDART Device Management [Web page]. Retrieved from https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__DEVICE.html


```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <cuda_runtime.h>
#include "device_launch_parameters.h"
#include <chrono> // For measuring runtime
#include <iomanip> // For displaying results on a table

#define TILE_SIZE 32

// We want to print the device properties
void queryGPUProperties() {
    int deviceCount;
    cudaGetDeviceCount(&deviceCount);

    for (int i = 0; i < deviceCount; ++i) {
        cudaDeviceProp deviceProp;
        cudaGetDeviceProperties(&deviceProp, i);
        std::cout << "Device Name: " << deviceProp.name << std::endl;
        std::cout << "  Max threads per block: " << deviceProp.maxThreadsPerBlock <<
std::endl;
        std::cout << "  Compute capability: " << deviceProp.major << "." << deviceProp.minor
<< std::endl;
        std::cout << "  Total global memory: " << deviceProp.totalGlobalMem << " bytes" <<
std::endl;
        std::cout << "  Memory Clock Rate (KHz): " << deviceProp.memoryClockRate <<
std::endl;
        // Added more a bit of device properties compared to the first exercise (Baed on
Reference 3)
    }
}

__global__ void matmul_rec_glob(float* A, float* B, float* C, int n, int k, int m, float*
num_operations, float* num_globmem_acc) {
    // Matrix multiplication function using global memory (based on page 67)
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < n && col < m) {
        float sum = 0.0f;
        float operations = 0.0f;
```

```
        float accesses = 0.0f;
        for (int i = 0; i < k; ++i) {
            sum += A[row * k + i] * B[i * m + col];
            operations += 2.0f; // One multiplication + addition and therefore +2 operations
per iter. This is my own code
            accesses += 2.0f; // Accessing A[i] and B[i] per iter and thus +2 accesses per
iter
        }
        C[row * m + col] = sum;
        num_operations[row * m + col] = operations; // store number of operations and
accesses on the matrices. This is my own code
        num_globmem_acc[row * m + col] = accesses;
    }
}

__global__ void matmul_rec_shar(float* A, float* B, float* C, int n, int k, int m, float*
num_operations, float* num_globmem_acc) {
    // Matrix multiplication function using shared memory (based on page 87)
    __shared__ float s_A[TILE_SIZE][TILE_SIZE];
    __shared__ float s_B[TILE_SIZE][TILE_SIZE];

    int bx = blockIdx.x, by = blockIdx.y;
    int tx = threadIdx.x, ty = threadIdx.y;

    int Row = by * TILE_SIZE + ty;
    int Col = bx * TILE_SIZE + tx;

    float Cvalue = 0.0f;
    float operations = 0.0f;
    float accesses = 0.0f;

    for (int t = 0; t < (k + TILE_SIZE - 1) / TILE_SIZE; ++t) {
        if (Row < n && t * TILE_SIZE + tx < k) {
            s_A[ty][tx] = A[Row * k + t * TILE_SIZE + tx];
        }
        else {
            s_A[ty][tx] = 0.0f;
        }

        if (Col < m && t * TILE_SIZE + ty < k) {
            s_B[ty][tx] = B[(t * TILE_SIZE + ty) * m + Col];
        }
        else {
            s_B[ty][tx] = 0.0f;
        }

        __syncthreads();

        for (int i = 0; i < TILE_SIZE; ++i) {
            Cvalue += s_A[ty][i] * s_B[i][tx];
            operations += 2.0f; // One multiplication + addition and therefore +2 operations
per iter. This is my own code
            accesses += 2.0f; // Accessing A[i] and B[i] per iter and thus +2 accesses per
iter
        }

        __syncthreads();
    }

    if (Row < n && Col < m) {
        C[Row * m + Col] = Cvalue;
        num_operations[Row * m + Col] = operations; // store number of operations and
accesses on the matrices. This is my own code
        num_globmem_acc[Row * m + Col] = accesses;
    }
}


// We want to generate random matrices
```

```
void generateRandomMatrix(float* matrix, int rows, int cols) {
    for (int i = 0; i < rows * cols; ++i) {
        matrix[i] = static_cast<float>(rand()) / static_cast<float>(RAND_MAX) * 100.0f;
    }
}

// We want a function that computes the CGMA ratio. My own code
float computeCGMARatio(float* num_operations, float* num_globmem_acc, int n, int m) {
    // We calculate total CGMA ratio
    float total_cgma_ratio = 0.0f;
    for (int j = 0; j < n * m; ++j) {
        if (num_globmem_acc[j] != 0) {
            total_cgma_ratio += num_operations[j] / num_globmem_acc[j];
        }
    }

    // We calculate average CGMA ratio for all elements in the matrix
    float average_cgma_ratio = total_cgma_ratio / (n * m);

    return average_cgma_ratio;
}


int main() {
    // Query and print GPU properties
    queryGPUProperties();

    std::cout << "\n";
    std::cout << "-------------------------Device 0 performance-------------------------"
<< std::endl;
    std::cout << "\n";

    std::cout << std::setw(5) << "Matrix Size" << std::setw(25) << "Global Memory
Runtime(sec)" << std::setw(25) << "Shared Memory Runtime(sec)" << std::setw(25) << "Global
Memory CGMA ratio" << std::setw(25) << "Shared Memory CGMA ratio" << std::endl;
    std::cout << std::string(140, '-') << std::endl;

    // Loop through different matrix sizes n, k, m = 32*i where i is the iteration number
    for (int i = 1; i <= 625; ++i) {
        // Initializing matrices and matrix row and column sizes
        float* A, * B, * C;
        int n = 32 * i;
        int k = 32 * i;
        int m = 32 * i;
        float CGMA_ratio_glob; // I call these two "Metric matrices" used to compute CGMA
        float CGMA_ratio_shar;

        size_t size_A = n * k * sizeof(float);
        size_t size_B = k * m * sizeof(float);
        size_t size_C = n * m * sizeof(float);
        cudaMallocManaged(&A, size_A);
        cudaMallocManaged(&B, size_B);
        cudaMallocManaged(&C, size_C);

        // Allocating memory for matrices and setting "Metric matrices" to a matrix full of
zeroes. These matrices will be filled soon
        float* num_operations, * num_globmem_acc;
        cudaMallocManaged(&num_operations, size_C * sizeof(float));
        cudaMallocManaged(&num_globmem_acc, size_C * sizeof(float));
        cudaMemset(num_operations, 0, size_C * sizeof(float));
        cudaMemset(num_globmem_acc, 0, size_C * sizeof(float));

        // We generate random matrices A and B
        generateRandomMatrix(A, n, k);
        generateRandomMatrix(B, k, m);

        // We define block and grid dimensions by the TILE_SIZE that is set
        dim3 blockSize(TILE_SIZE, TILE_SIZE);
        dim3 gridSize((m + TILE_SIZE - 1) / TILE_SIZE, (n + TILE_SIZE - 1) / TILE_SIZE);
```

```
        // We perform matrix multiplication using global memory and find the runtime
duration
        auto start = std::chrono::high_resolution_clock::now();
        matmul_rec_shar << <gridSize, blockSize >> > (A, B, C, n, k, m, num_operations,
num_globmem_acc);
        cudaDeviceSynchronize();
        auto end = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> duration = end - start;

        CGMA_ratio_glob = computeCGMARatio(num_operations, num_globmem_acc, n, m); // We
compute the global memory's CGMA ratio. My own code
        cudaMemset(num_operations, 0, size_C * sizeof(float));
        cudaMemset(num_globmem_acc, 0, size_C * sizeof(float));

        // We perform matrix multiplication using shared memory and find the runtime
duration
        auto start_2 = std::chrono::high_resolution_clock::now();
        matmul_rec_glob << <gridSize, blockSize >> > (A, B, C, n, k, m, num_operations,
num_globmem_acc);
        cudaDeviceSynchronize();
        auto end_2 = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> duration_2 = end_2 - start_2;

        CGMA_ratio_shar = computeCGMARatio(num_operations, num_globmem_acc, n, m); // We
compute the global memory's CGMA ratio. My own code

        // Print results in a table
        std::cout << std::setw(5) << n << std::setw(25) << duration_2.count() <<
std::setw(25) << duration.count() << std::setw(25) << CGMA_ratio_glob << std::setw(25) <<
CGMA_ratio_shar << std::endl;


        // Was adviced to free up memory (page 51)
        cudaFree(A);
        cudaFree(B);
        cudaFree(C);
    }

    return 0;
}
```