CS 239 Exercise 2 Report

Gerry Agluba Jr.

CS 239 WZZQ- Parallel Computing
Date of Submission: March 26, 2025

1. OBJECTIVES

This report highlights demonstration of (a) improving performance of parallel programs by exploiting different memory strategies namely: shared memory access, access times, patterns, and latency tolerance (b) ability to design reasonable parallel programs based on hardware memory limitations. This exercise revolves around implementing a matrix multiplication program with two matrices of arbitrary shapes inputs leveraging different types of memory access paradigms.

2. METHODOLOGY

The following methods lies on the assumption that a heterogonous computer system is already setup. In particular, a working CUDA [Cor25a] environment with compatible C++ compiler should be set up prior to design, programming and tests.

2.1 Environment GPU Device Properties

Initially, different device properties were queried to plan different experimental setup that can be made as well as determining the limitations of current environment. A utility function to print device properties were written with the following values observed in table 1

2.2 Kernel Design and Implementation [1]

As stated, two programs of matrix multiplication will be implemented. Kernel function named **matmul_rec_glob** is implemented which calculates the product of two matrices $\mathbf{A} \in \mathbb{R}^{n\times k}$ and $\mathbf{B} \in \mathbb{R}^{k\times m}$ producing an output $\mathbf{C} \in \mathbb{R}^{n\times m}$ implemented using global memory access. Another kernel **matmul_rec_shard** is implemented using tiling algorithm with shared memory access.
For both kernels function; **thread block** of size **32** x **32** is fixed with **TILE_WIDTH** also fixed at the same size. Blocks and tile of said dimensions exactly fits the **1024 max threads per SM** and takes advantage of full utilization of **warp size (32)** and maximization of threads per block.

2.3 Experimental Setup

2.3.1 Experiments

The following items summarized the experiment setup for this report.

- Each experiment will be replicated 10 times.

[1] The whole implementation can be found on github repository

Device Count	1
NVIDIA GeForce GTX 1650 SUPER Properties	
Clock Rate (KHz)	1740000
Memory Clock Rate (KHz)	6001000
Total Global Memory (MB)	4095
Shared Memory / Block (KB)	48
Warp Size	32
Pitch (MB)	2047
Max Threads / Block	1024
Max Dimension Size of Block	1024, 1024, 64
Max Dimension Size of Grid	2147483647, 65535, 65535
Multiprocess Count	20
Max Blocks / Multiprocessor	16
Concurrent Kernels	1
Max Threads / Multiprocessor	1024
Shared Memory (KB) / Multiprocessor	64

Table 1: NVIDIA GeForce GTX 1650 SUPER Properties

- Each experiment input matrices will be randomly generated from a uniform distribution $M[i][j] \sim \mathcal{U}(0, 100)$ with matrix size parameters: **n,k and m** all sampled from normal distribution $\mathcal{N}(\mu = \mathbf{b}, \sigma = \mathbf{0.1b}$, $\mathbf{b} \in \{2^4, 2^5, 2^6, , 2^7, 2^8, 2^9, 2^12, 2^{13}\}$ providing a more robust rectangular matrix shapes.
- For this experiment, necessary metrics primarily **runtime (s)** and **compute throughput (%)** will be acquired and sample average over 10 runs will be computed.
- Profiling and Performance Monitoring Tool **NVIDIA Nsight Compute [Cor25b]** will also be utilized for main and other supporting metrics.

3. RESULTS

3.1 Runtime and GPU Throughput

Figure 1 and 3.1 details the kernel runtime results and gpu throughput of both kernel implementations. Runtime seems to show a comparable runtime performances in both implementations with the matmul_rec_glob kernel function inching over matmul_rec_shar. On the other hand, a notable performance difference can be seen in the trends of GPU compute throughput, as the tiled matrix multiplication with shared memory access significantly better than the naive implementation.

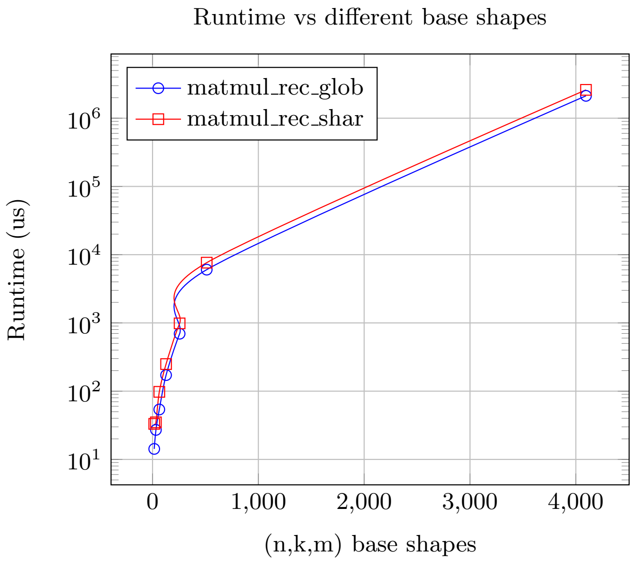


Figure 1: Relative runtime of both kernel implementations: matmul_rec_glob and matmul_rec_shar over 10 runs acquired from NVIDIA Nsight Compute

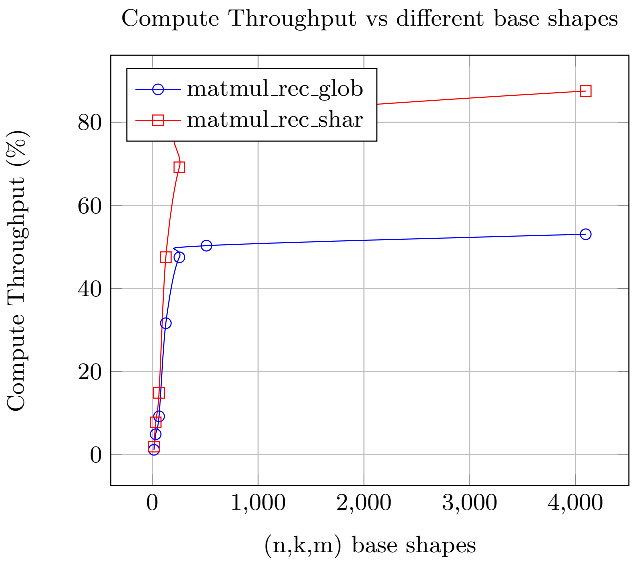


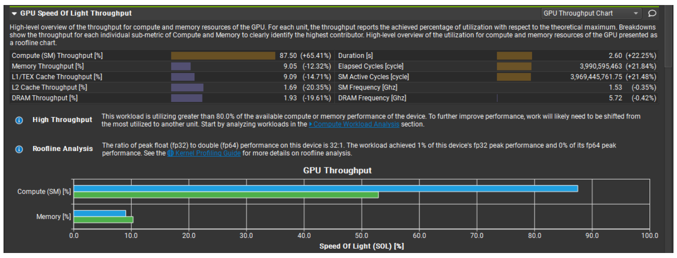
Figure 2: Comparison between the GPU throughputs of matmul_rec_glob and matmul_rec_shar over 10 runs acquired from NVIDIA Nsight Compute

3.2 Additional NSight Results

An instance of the experiment profiling result is presented below to show empirical observations of both kernel implementations. matmul_rec_glob is labelled in the following results as *baseline* (mostly represented in green color) and *current* is labelled unto matmul_rec_shar (mostly represented in blue color). The sample instance is taken from the experiment with the highest matrix size setup.

Figure 3 indicates a **+65.41%** relative increase on **Compute (SM) Throughput** of matmul_rec_shar ove matmul_rec_glob but worse performance can be seen for **Memory Throughput** with **-12.32%** and Runtime (**+22.25%**).

Additionally, matmul_rec_shar exhibites a **Very High Utilization of SM ALU** with **87%** compared matmul_rec_glob of **53.1%** and have **busier SM (+65.37%)** as shown in figure 4.



GPU Speed Of Light Throughput

GPU Throughput Chart

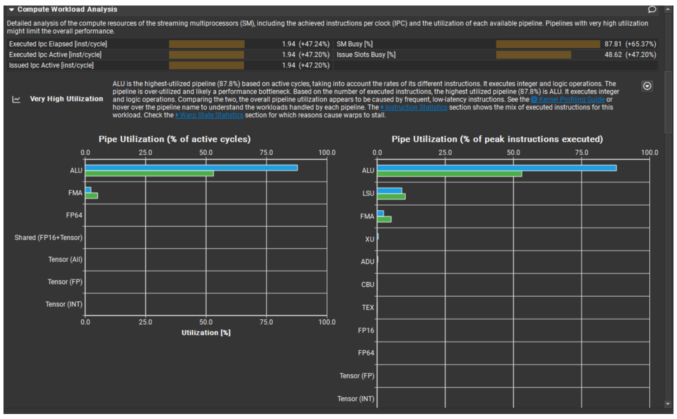
High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	87.50 (+65.41%)	Duration [s]	2.60 (+22.25%)
Memory Throughput [%]	9.05 (-12.32%)	Elapsed Cycles [cycle]	3,990,595,463 (+21.84%)
L1/TEX Cache Throughput [%]	9.09 (-14.71%)	SM Active Cycles [cycle]	3,969,445,761.75 (+21.48%)
L2 Cache Throughput [%]	1.69 (-20.35%)	SM Frequency [Ghz]	1.53 (-0.35%)
DRAM Throughput [%]	1.93 (-19.61%)	DRAM Frequency [Ghz]	5.72 (-0.42%)

High Throughput This workload is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing workloads in the Compute Workload Analysis section.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 32:1. The workload achieved 1% of this device's fp32 peak performance and 0% of its fp64 peak performance. See the Kernel Profiling Guide for more details on roofline analysis.

Figure 3: A sample snapshot of the throughput analysis of matmul_rec_glob vs matmul_rec_shar



Compute Workload Analysis

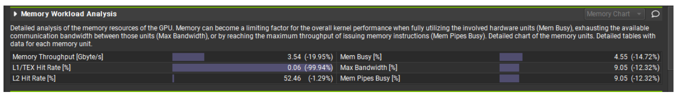
Detailed analysis of the compute resources of the streaming multiprocessors (SM), including the achieved instructions per clock (IPC) and the utilization of each available pipeline. Pipelines with very high utilization might limit the overall performance.

Executed Ipc Elapsed [inst/cycle]	1.94 (+47.24%)	SM Busy [%]	87.81 (+65.37%)
Executed Ipc Active [inst/cycle]	1.94 (+47.20%)	Issue Slots Busy [%]	48.62 (+47.20%)
Issued Ipc Active [inst/cycle]	1.94 (+47.20%)		

Very High Utilization ALU is the highest-utilized pipeline (87.8%) based on active cycles, taking into account the rates of its different instructions. It executes integer and logic operations. The pipeline is over-utilized and likely a performance bottleneck. Based on the number of executed instructions, the highest utilized pipeline (87.8%) is ALU. It executes integer and logic operations. Comparing the two, the overall pipeline utilization appears to be caused by frequent, low-latency instructions. See the Kernel Profiling Guide or hover over the pipeline name to understand the workloads handled by each pipeline. The Instruction Statistics section shows the mix of executed instructions for this workload. Check the Warp State Statistics section for which reasons cause warps to stall.

Figure 4: A sample snapshot of the compute workload analysis of matmul_rec_glob vs matmul_rec_shar

On the contrary, matmul_rec_shar performs worse with regards to memory through **-19.95%** relative decrease 5.



Memory Workload Analysis

Memory Chart

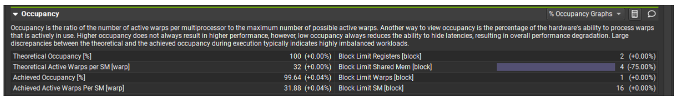
Detailed analysis of the memory resources of the GPU. Memory can become a limiting factor for the overall kernel performance when fully utilizing the involved hardware units (Mem Busy), exhausting the available communication bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput of issuing memory instructions (Mem Pipes Busy). Detailed chart of the memory units. Detailed tables with data for each memory unit.

Memory Throughput [Gbyte/s]	3.54 (-19.95%)	Mem Busy [%]	4.55 (-14.72%)
L1/TEX Hit Rate [%]	0.06 (-99.94%)	Max Bandwidth [%]	9.05 (-12.32%)
L2 Hit Rate [%]	52.46 (-1.29%)	Mem Pipes Busy [%]	9.05 (-12.32%)

Figure 5: A sample snapshot of the memory workload analysis of matmul_rec_glob vs matmul_rec_shar

Shown in 6, Both kernel functions has achieved high marks on **occupancy** with almost **100%** as well as fully utilizing all warps per SM.

3.3 Computations per Global Memory Access (CGMA)



Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]	100 (+0.00%)	Block Limit Registers [block]	2 (+0.00%)
Theoretical Active Warps per SM [warp]	32 (+0.00%)	Block Limit Shared Mem [block]	4 (-75.00%)
Achieved Occupancy [%]	99.64 (+0.04%)	Block Limit Warps [block]	1 (+0.00%)
Achieved Active Warps Per SM [warp]	31.88 (+0.04%)	Block Limit SM [block]	16 (+0.00%)

Figure 6: A sample snapshot of the occupancy of matmul_rec_glob vs matmul_rec_shar

In the subsection, we calculate both kernels' CGMA.
For matmul_rec_glob; each thread in a block requires row(column) of matrix inputs **A** and **B** of size **k** totalling to **2k**. It performs multiplication and addition of single precision (fp32) floating point operations on these memory access aggregating to **2k** operations. Computing CGMA will yield a ratio of $\frac{2k}{2k} = 1$.
On the other hand, kernel matmul_rec_shared will load **2** element of associated row(col) of inputs **A** and **B** in collaboration with other threads in a block. Succeeding this is a total operation of **2*TILE_WIDTH=2*32** addition and multiplication operations yielding to a CGMA of $\frac{2*TILE\_WIDTH}{2} = TILE\_WIDTH = 32$.
Theoretical computations shows **significantly better** CGMA for tiled implementation with shared memory access by a factor of **TILE_WIDTH**.

4. OBSERVATIONS AND NSIGHT (NO PUN INTENDED)

In this report we explore two implementations of matrix multiplication with different memory access strategies. Below are the following insights and observations that were gathered while implementing the program, computing theoretical and analysing empirical results.

4.1 Notes on matmul_rec_shar high throughput

Implementations on shared memory enables higher compute eficicency because of data loaded in a shared memory are reused by multiple computations. Computed theoretical CGMA as well as empirical results on ALU utilization manifests this inherent property.

4.2 Arguments on Insignificant Runtime Speed

Contrary to general notion of better performance on shared memory access techniques. Results shows that there are no significant difference between the wall clock runtime of both kernels. These can be attributed to several factors.
Physical Device Shared Memory Capacity can have a significant impact on the runtime of matmul_rec_shar; an homage to the ideas presented on **Chapter 5.4 - Memory as a Limiting Factor to Parallelism**[KH12]. Physical shared memory limit of a single SM is **64KB**. Say we have two 4096 × 4096 single precision float matrix to load. Assuming we use all **20** SM on our current setup. That would take around $\frac{2 \times 4096 \times 4096 Kb}{20 \times 64 Kb} \approx$ **102** cycles to load. If shared memory access is around that **100x** faster or lower, its not suprising that the empircal results we have for runtime speed is not sufficiently significant. This is not yet considering other factors like other memory access latency, thread synchronization etc.
As mentioned, excessive use of **__syncthreads()** can also be one factor though on this report, the influence is not yet measured.

5. RECOMMENDATION

The following items were not yet explored and is pushed as candidate for recommendation that are relevant to this report's objectives.

1. Coalesced Implementation Measures
2. Thread Synchronization
3. Memory Access Latency Observations
4. Other relevant metrics used by NVIDIA Nsight compute
5. Higher variation of matrix sizes
6. Run on environment with **higher shared memory capacity**

6. REFERENCES

References

[Cor25a] NVIDIA Corporation. *CUDA C++ Programming Guide*. 2025. URL: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/` (visited on 02/28/2025).

[Cor25b] NVIDIA Corporation. *NVIDIA Nsight Compute*. 2025. URL: `https://developer.nvidia.com/nsight-compute`.

[KH12] Kirk and Hwu. *Programming Massively Parallel Processors A Hands-on Approach*. Morgan Kaufmann, 2012.

APPENDIX

Listing 1: Kernel Function Implementation

```
__global__ void matmul_rec_glob(float* A, float* B, float* C, int n, int k, int m) {
        // this thread computs C[x][y];
        int row = blockIdx.y * blockDim.y + threadIdx.y;
        int col = blockIdx.x * blockDim.x + threadIdx.x;


        if (row < n && col < m) {
                float cVal = 0;
                for (int _k = 0; _k < k;_k++) {
                        cVal += A[row * k + _k] * B[_k * m + col];
                }
                C[row * m + col] = cVal;
        }
}

__global__ void matmul_rec_shar(float* A, float* B, float* C, int n, int k, int m) {
        __shared__ float ds_A[TILE_WIDTH][TILE_WIDTH];
        __shared__ float ds_B[TILE_WIDTH][TILE_WIDTH];

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

        int row = by * TILE_WIDTH + ty;
        int col = bx * TILE_WIDTH + tx;

        if (row < n && col < m) {
                float pValue = 0; // partial sum
                for (int q = 0; q < (k + TILE_WIDTH - 1) / TILE_WIDTH; q++) {
                        // Load collaboratively a row/col on a submatrix
                        ds_A[ty][tx] = A[(row * k) + (q * TILE_WIDTH + tx)];
                        ds_B[ty][tx] = B[col + (q * TILE_WIDTH + ty) * k];
                        // ds_B[ty][tx] = B[(q * TILE_WIDTH + ty)*m + col]; // more coalesced??

                        __syncthreads();
                        for (int r = 0; r < TILE_WIDTH;r++) {
                                pValue += ds_A[ty][r] * ds_B[r][tx];
                                // pValue += ds_A[ty][r] * ds_B[tx][r];
                        }
                        __syncthreads();
                }
                C[row * m + col] = pValue;
        }
}
```