Matrix Multiply (MTMT) Report

Summary

Two CUDA implementations were created:

1. **mtmt-xy-cr.cu**
   - Grid **x-dimension → column index j**
   - Grid **y-dimension → row index i**
2. **mtmt-xy-rc.cu**
   - Grid **x-dimension → row index i**
   - Grid **y-dimension → column index j**

Both versions use **16×16 thread blocks** and implement the same arithmetic, differing only in how threads map to indices.

Thread Index Mappings

1. XY-CR (x → column j, y → row i)

```
int i = blockIdx.y * blockDim.y + threadIdx.y;
int j = blockIdx.x * blockDim.x + threadIdx.x;
```

2. XY-RC (x → row i, y → column j)

```
int j = blockIdx.y * blockDim.y + threadIdx.y;
int i = blockIdx.x * blockDim.x + threadIdx.x;
```

Memory Coalescing Analysis

Each thread computes:

```
sum += A[k*n + i] * B[j*n + k];
```

We evaluate how warps access memory in both mappings.

XY-RC (x → i, y → j) — FAST VERSION

Warp behavior

- Threads vary in **i**
- Threads share same **j**

Access patterns

Matrix	Access	Coalesced?	Notes
A[k*n+i]	i varies	Yes	Contiguous row access → fully coalesced
B[j*n+k]	j constant	Broadcast	All threads load same element; efficient
C[i*n+j]	i varies	No	Strided write

Why it's fast

- A is accessed inside the k-loop and is fully coalesced → major benefit
- Broadcast access to B is cheap
- Strided writes to C matter far less than strided reads

Result: high bandwidth efficiency.

XY-CR (x → j, y → i) — SLOW VERSION

Warp behavior

- Threads vary in **j**
- Threads share same **i**

Access patterns

Matrix	Access	Coalesced?	Notes
A[k*n+i]	i constant	Broadcast	Cheap, but no reuse
B[j*n+k]	j varies	No	Huge stride of n → non-coalesced loads
C[i*n+j]	j varies	Yes	Good write coalescing

Why it's slow

- Accesses to B are strided by `n` elements
- No reuse across threads → constant cache misses
- This becomes the bottleneck inside the k-loop

Result: significantly lower performance.

Experimental Results

Tests run on: **RTX A6000 (rtxpr6000bl)**

Matrix size: **999 × 999** Block size: **16 × 16 (256 threads)**

Version	Time (ms)	Performance (GFLOP/s)
mtmt-xy-rc	**0.35 ms**	**5718.86 GFLOPs**
mtmt-xy-cr	**0.59 ms**	**3351.77 GFLOPs**

Performance Difference

- XY-RC is ~**70% faster** than XY-CR
- Matches coalescing analysis

Final Conclusions

- **XY-RC (x → row, y → col)** is the better mapping.
- Its advantage comes from **coalescing A**, which is accessed every loop iteration and dominates memory traffic.
- In contrast, XY-CR causes **severely strided accesses to B**, leading to poor memory bandwidth utilization.
- The experimental results matched theoretical predictions exactly, confirming that **coalesced reads in the inner loop dominate performance**, even when writes or other reads are uncoalesced.

Files Included

- `mtmt-xy-rc.cu` – row/column mapping (fast)
- `mtmt-xy-cr.cu` – column/row mapping (slow)

- `mtmt.cu` – baseline starter code
- `stencil.cu` - 1-D stencil computation code