Programming Massively Parallel Processors

Chapter06 Performance considerations

1944023 JiYeong Yi

Chapter06

Performance considerations

Table of contents

6.1 memory coalescing

- The global memory of a CUDA device is implemented with DRAM.
- Data bits are stored in DRAM cells that are small capacitors, in which the presence or absence of a tiny amount of electrical charge distinguishes between a 1 and a 0 value.
- Reading data from a DRAM cell requires the small capacitor to use its tiny electrical charge to drive a highly capacitive line leading to a sensor.
- Because this process is very slow relative to the desired data access speed, modern DRAM designs use parallelism to increase their rate of data access, commonly referred to as memory access throughput.

6.1 memory coalescing

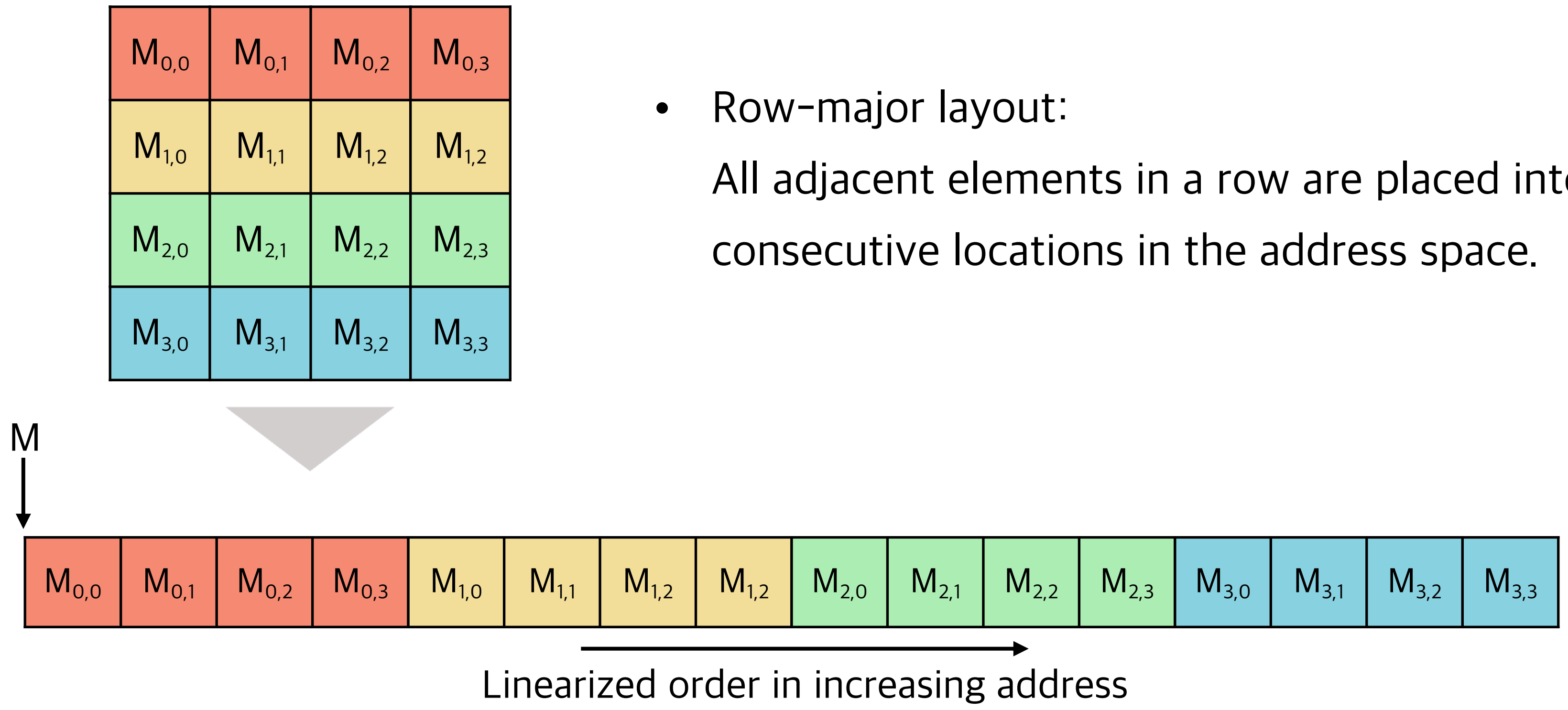
DRAM bursting

1. Accessing consecutive locations
   - Each time a DRAM location is accessed, a range of consecutive locations that include the requested location are accessed.

2. Delivered data from consecutive locations
   - Many sensors which are provided in each DRAM chip are worked in parallel and each senses the content of bit within consecutive locations.
   - Once detected by the sensors, the data from all these consecutive locations can be transferred at high speed to the processor.

6.1 memory coalescing

- The most favorable access pattern is achieved when all threads in a warp access consecutive global memory locations. In this case, all these accesses will be coalesced, or combined into a single request for consecutive locations when accessing the DRAM.

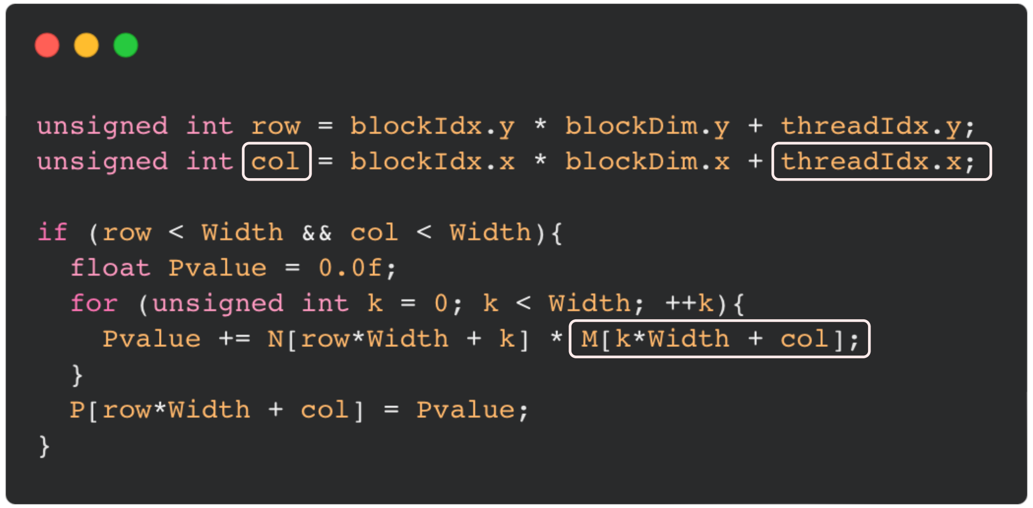


$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,2}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
M
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$ $M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,2}$ $M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$ $M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
Linearized order in increasing address

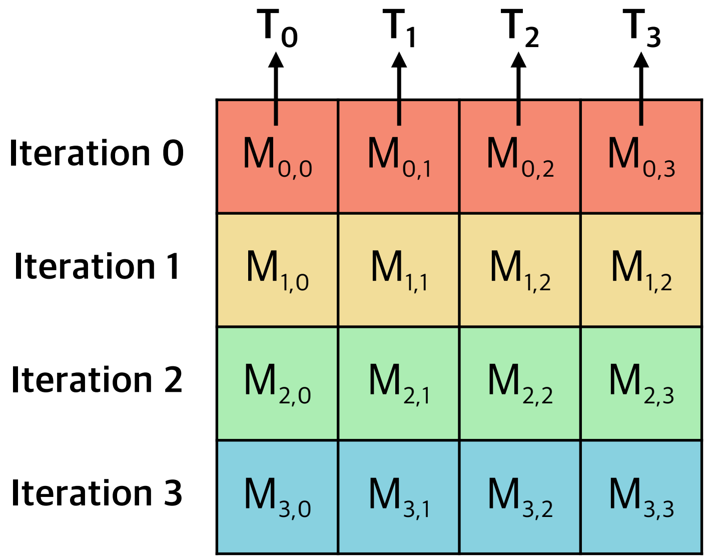
- Row-major layout:
  All adjacent elements in a row are placed into consecutive locations in the address space.

6.1 memory coalescing

- Row-major layout



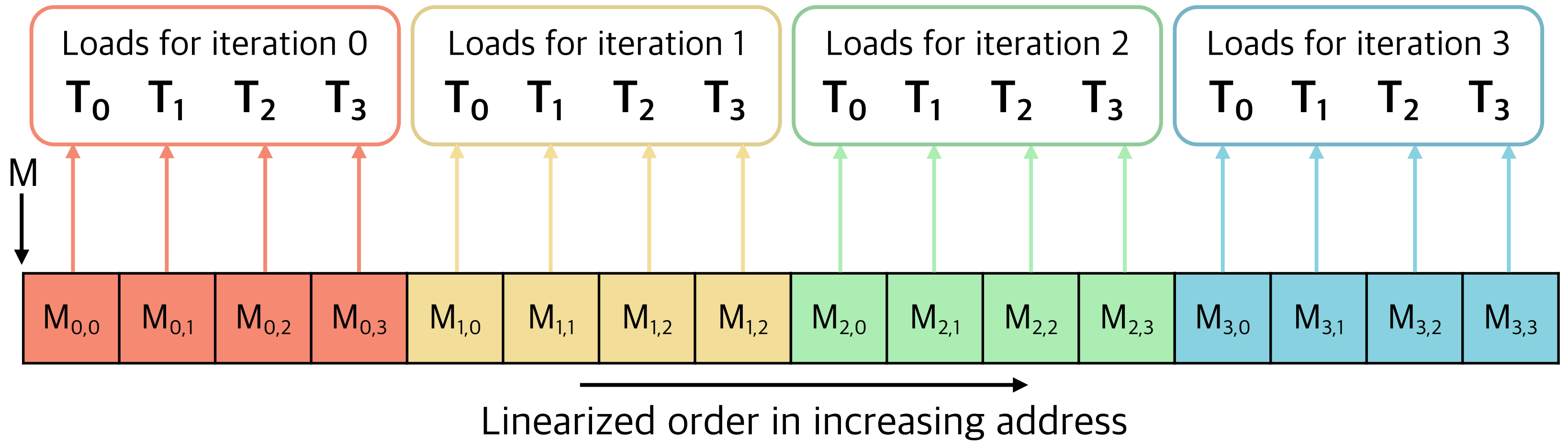
```
unsigned int row = blockIdx.y * blockDim.y + threadIdx.y;
unsigned int col = blockIdx.x * blockDim.x + threadIdx.x;

if (row < Width && col < Width){
  float Pvalue = 0.0f;
  for (unsigned int k = 0; k < Width; ++k){
    Pvalue += N[row*Width + k] * M[k*Width + col];
  }
  P[row*Width + col] = Pvalue;
}
```



T_0
T_1
T_2
T_3
Iteration 0
Iteration 1
Iteration 2
Iteration 3
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,2}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$

6.1 memory coalescing

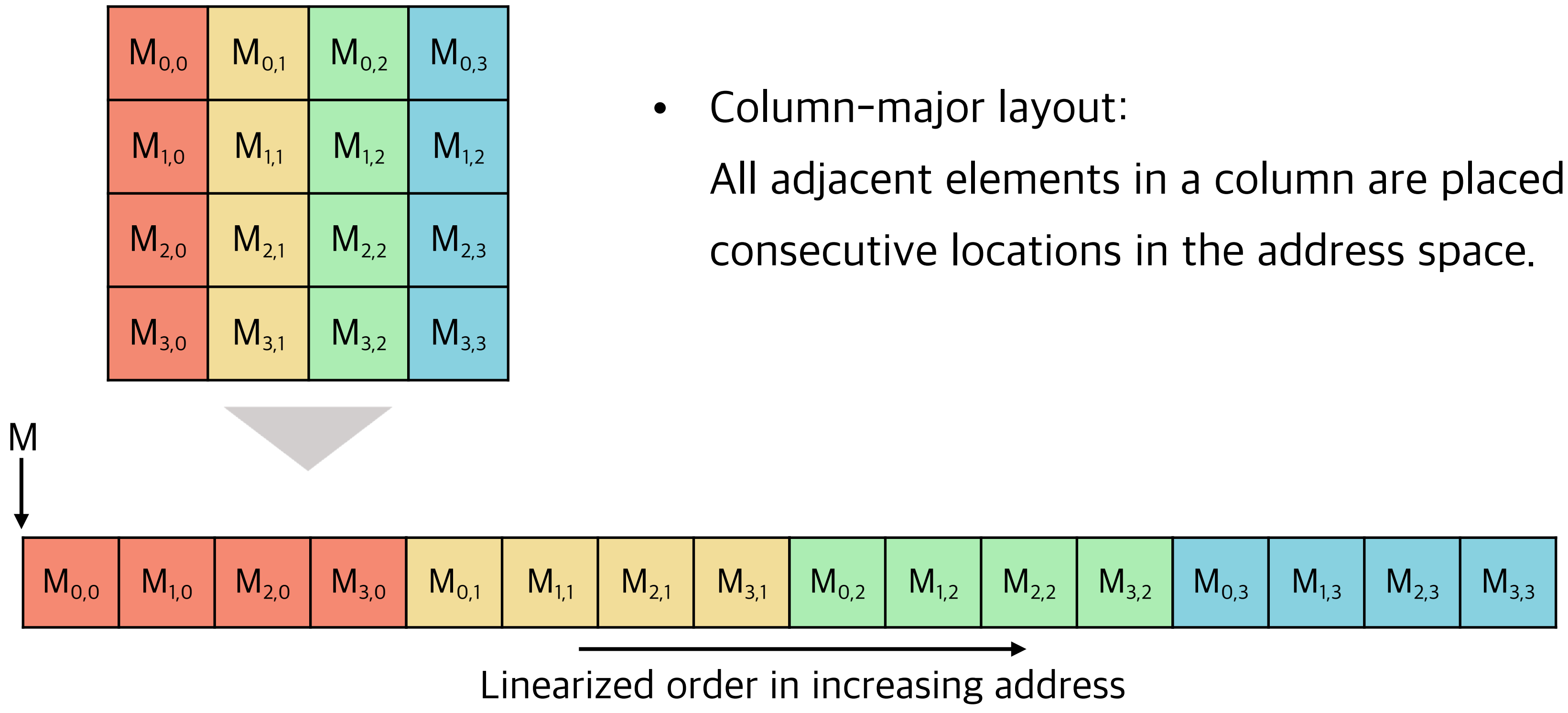
- Row-major layout: A coalesced access pattern.



Loads for iteration 0
T_0 T_1 T_2 T_3
Loads for iteration 1
T_0 T_1 T_2 T_3
Loads for iteration 2
T_0 T_1 T_2 T_3
Loads for iteration 3
T_0 T_1 T_2 T_3
M
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$ $M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,2}$ $M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$ $M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
Linearized order in increasing address

6.1 memory coalescing

- In linear algebra we often need to use both the original and transposed forms of a matrix.
- When the transposed form is needed, its elements can be accessed by accessing the original form by switching the roles of the row and column indices. It is equivalent to viewing the transposed matrix as a column-major layout of the original matrix.

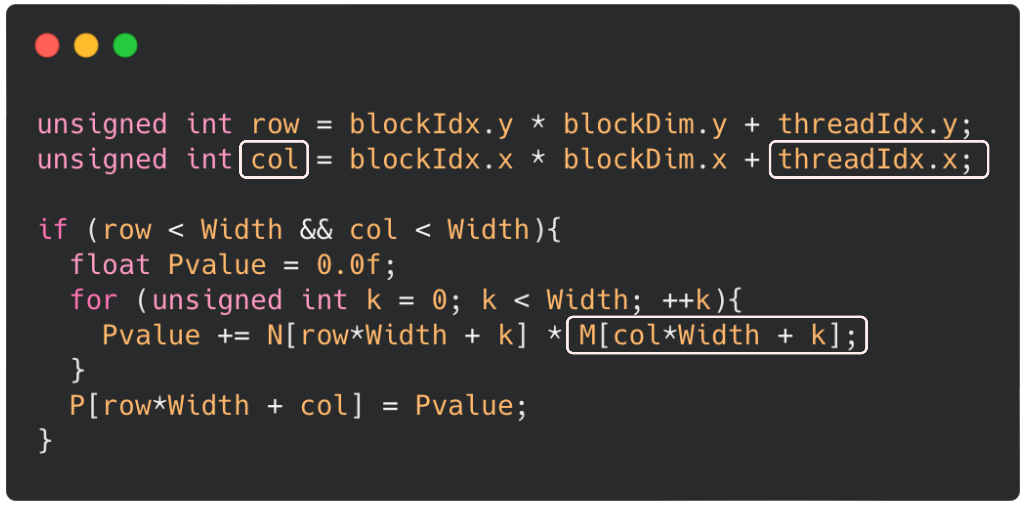


$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,2}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
M
$M_{0,0}$ $M_{1,0}$ $M_{2,0}$ $M_{3,0}$ $M_{0,1}$ $M_{1,1}$ $M_{2,1}$ $M_{3,1}$ $M_{0,2}$ $M_{1,2}$ $M_{2,2}$ $M_{3,2}$ $M_{0,3}$ $M_{1,3}$ $M_{2,3}$ $M_{3,3}$
Linearized order in increasing address

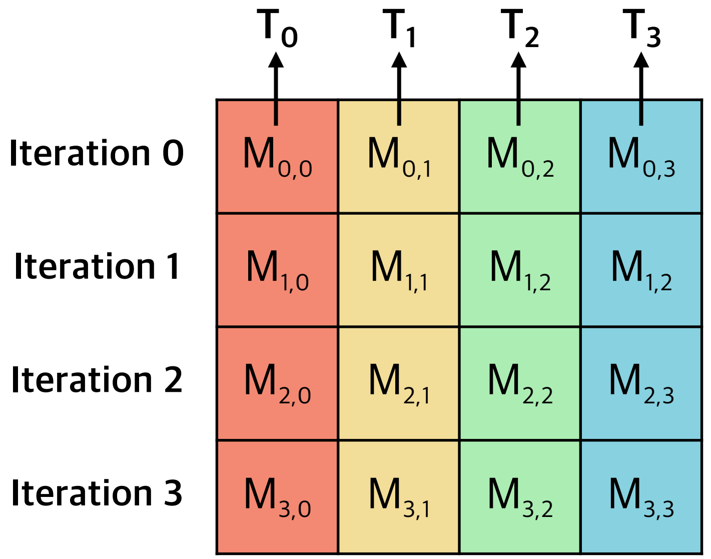
- Column-major layout:
  All adjacent elements in a column are placed into consecutive locations in the address space.

6.1 memory coalescing

- Column-major layout

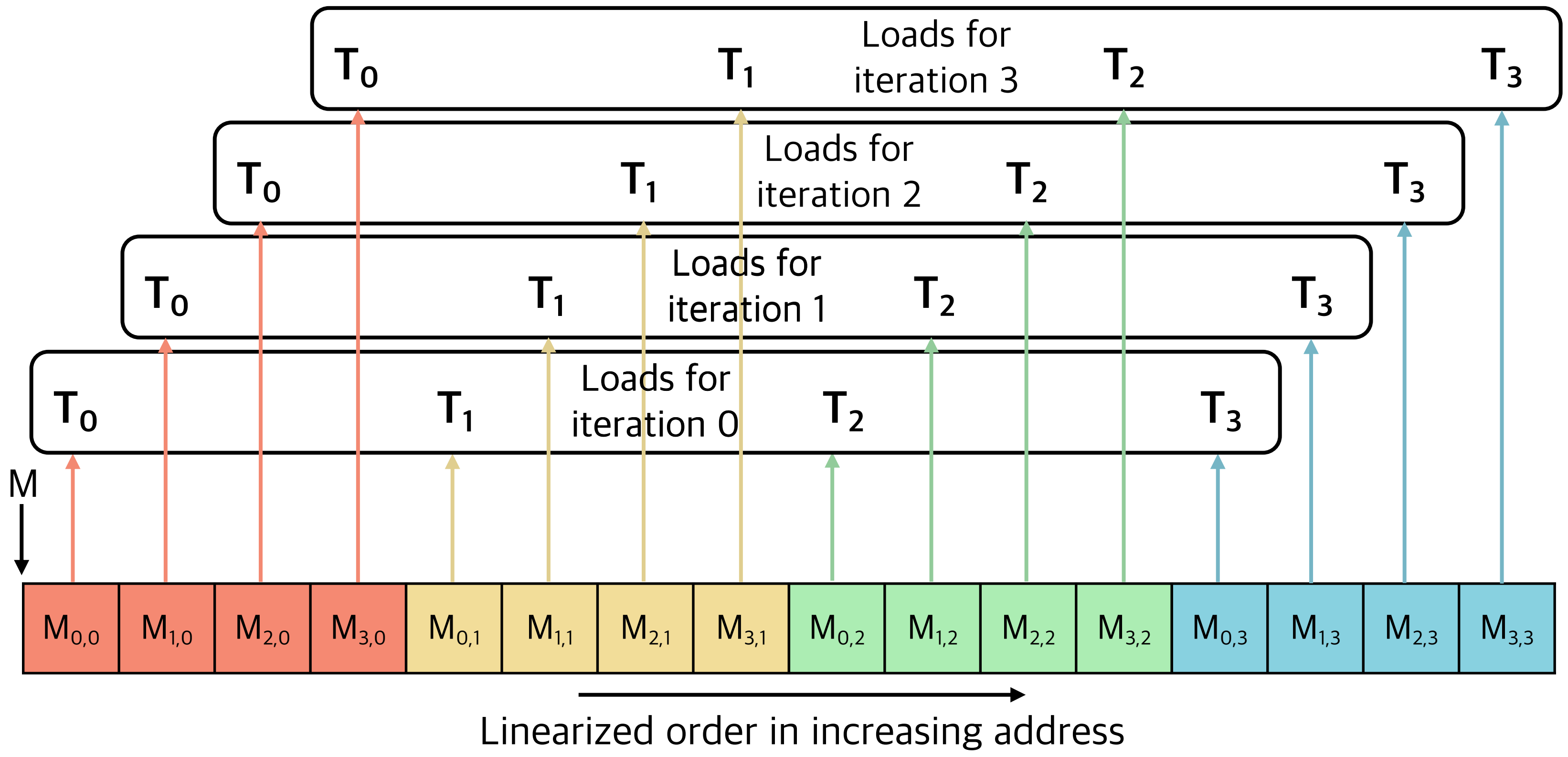


```
unsigned int row = blockIdx.y * blockDim.y + threadIdx.y;
unsigned int col = blockIdx.x * blockDim.x + threadIdx.x;

if (row < Width && col < Width){
  float Pvalue = 0.0f;
  for (unsigned int k = 0; k < Width; ++k){
    Pvalue += N[row*Width + k] * M[col*Width + k];
  }
  P[row*Width + col] = Pvalue;
}
```



6.1 memory coalescing

- Column-major layout: An uncoalesced access pattern.



Loads for iteration 3
T_0 T_1 T_2 T_3
Loads for iteration 2
T_0 T_1 T_2 T_3
Loads for iteration 1
T_0 T_1 T_2 T_3
Loads for iteration 0
T_0 T_1 T_2 T_3
M
$M_{0,0}$ $M_{1,0}$ $M_{2,0}$ $M_{3,0}$ $M_{0,1}$ $M_{1,1}$ $M_{2,1}$ $M_{3,1}$ $M_{0,2}$ $M_{1,2}$ $M_{2,2}$ $M_{3,2}$ $M_{0,3}$ $M_{1,3}$ $M_{2,3}$ $M_{3,3}$
Linearized order in increasing address

6.1 memory coalescing

- There are various strategies for optimizing code to achieve memory coalescing when the computation is not naturally amenable to it.

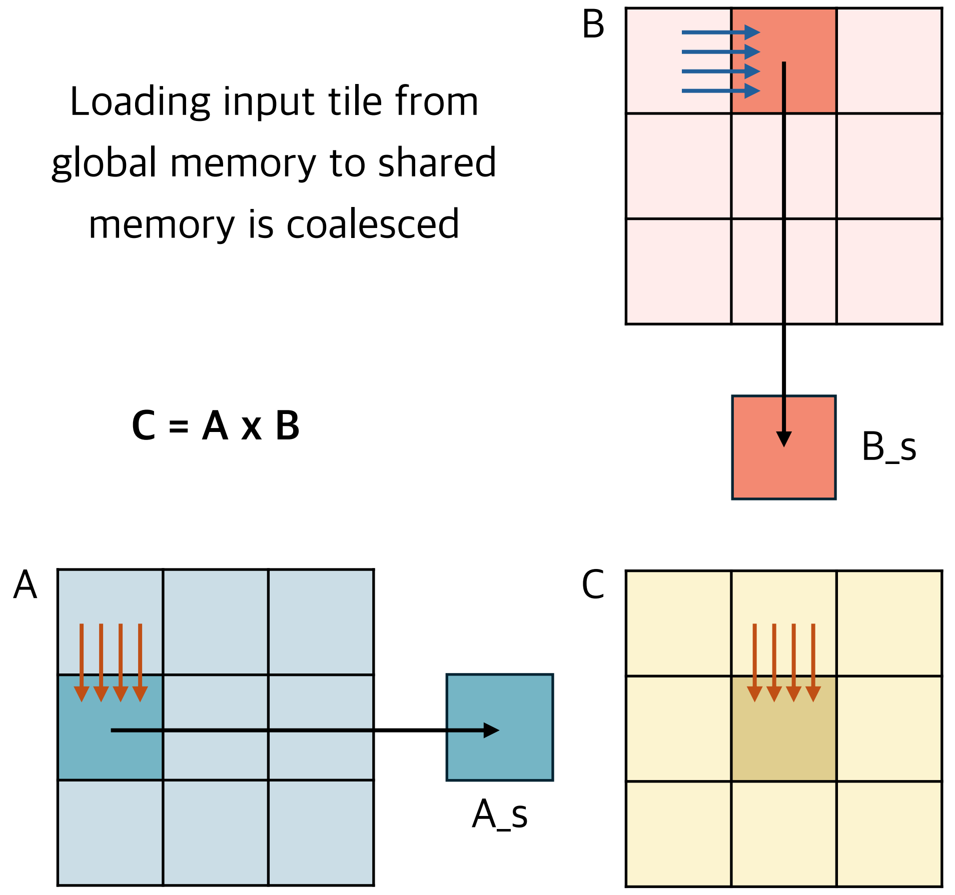
- Corner turning
  Strategy to transfer the data between global memory and shared memory in a coalesced manner and carry out the unfavorable access pattern in shared memory, which provides faster access latency.

6.1 memory coalescing

- Applying corner tuning to coalesce accesses to matrix B, which is stored in column-major layout

- A: Input matrix which is stored in row-major layout in global memory
  B: Input matrix which is stored in column-major layout in global memory
  C: Output matrix which is stored in row-major layout in global memory

Loading input tile from global memory to shared memory is coalesced



6.2 Hiding memory latency

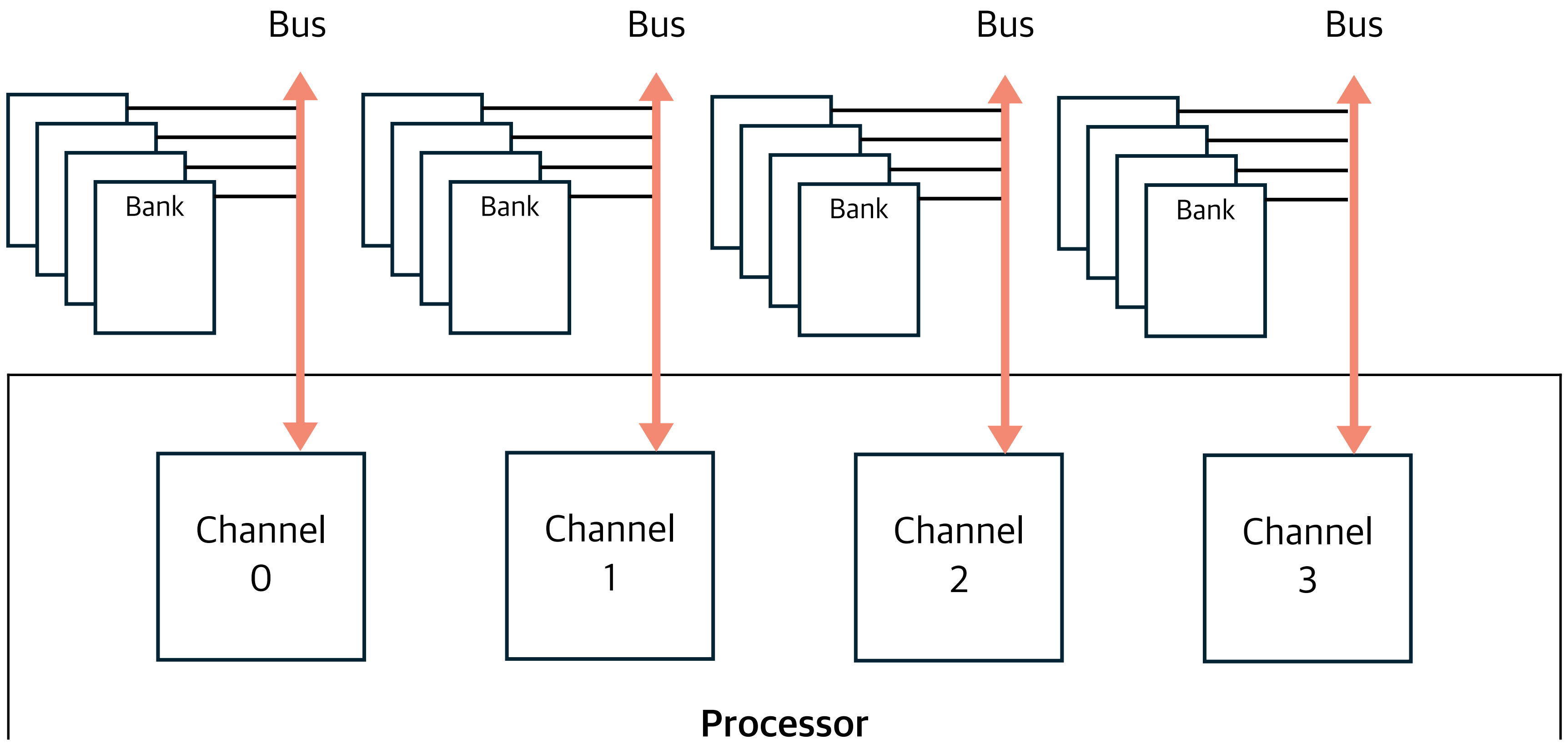
- Form of parallel organization in DRAM

1. DRAM bursting:
   Multiple locations are accessed in the DRAM core array in parallel.

2. Banks and channel:
   - Processor contains one or more channels. Each channel is a memory controller with a bus that connects a set of DRAM banks to the processor.
   - For each channel, the number of banks that is connected to it is determined by the number of banks required to fully utilized the data transfer bandwidth of the bus.

6.2 Hiding memory latency

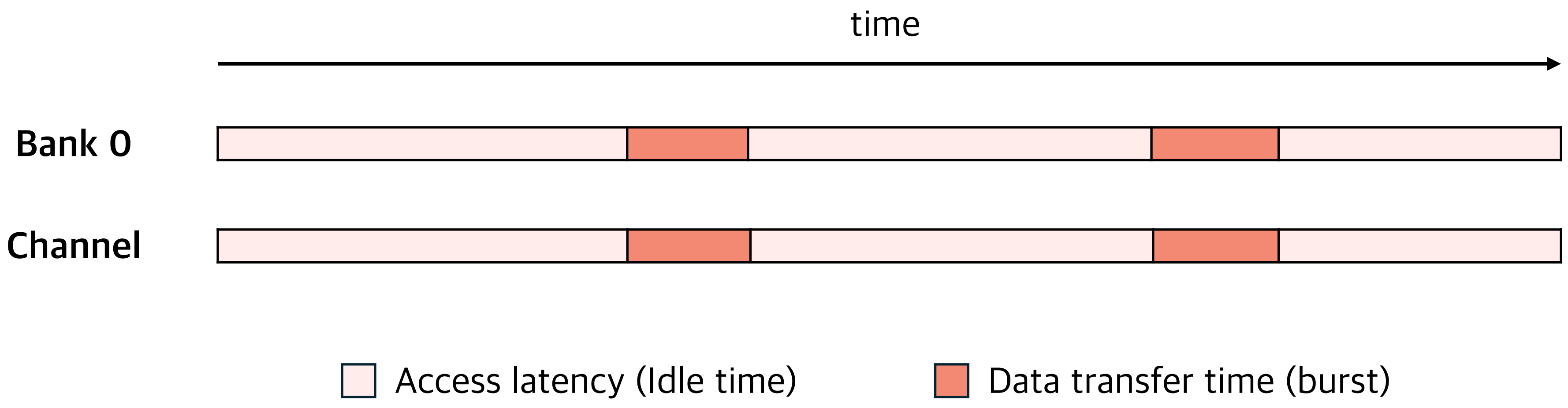
Channels and banks in DRAM system



6.2 Hiding memory latency

Single-bank organization

- Access latency is much longer than the data transfer time.
  Access transfer timing of a one-bank organization would grossly underutilize the data transfer bandwidth of the channel bus.

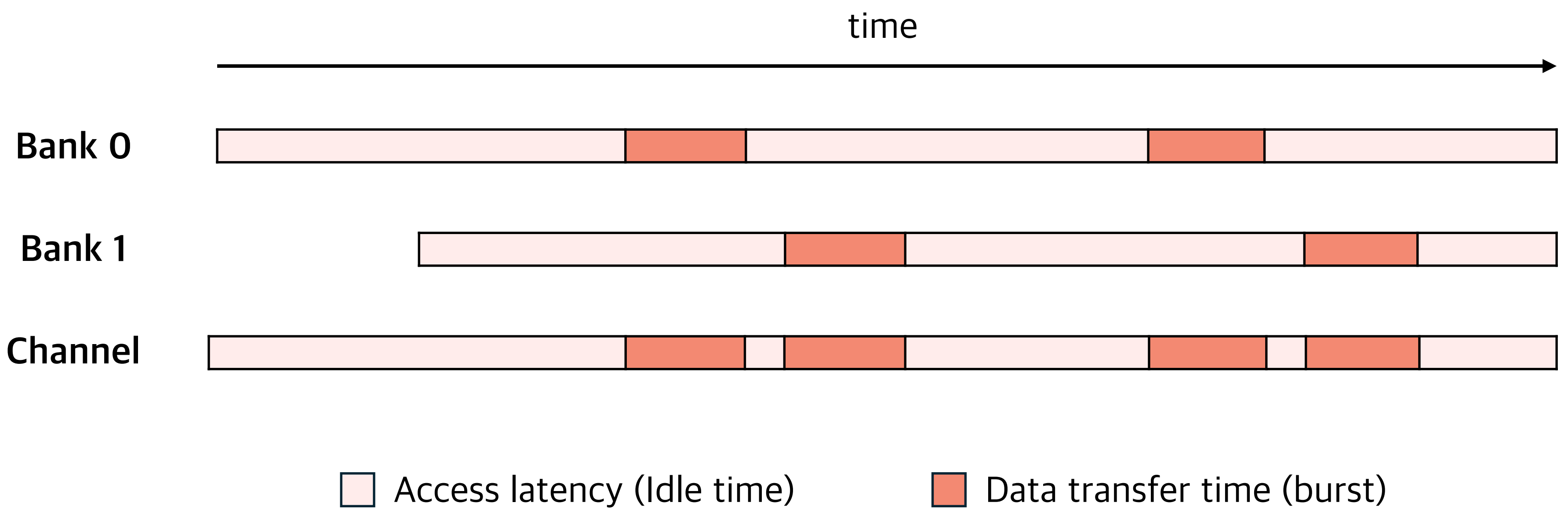


[Single-Bank channel access timeline]

6.2 Hiding memory latency

Multiple-bank organization

- Multiple-bank organization can **overlap the latency** for accessing the DRAM cell arrays.

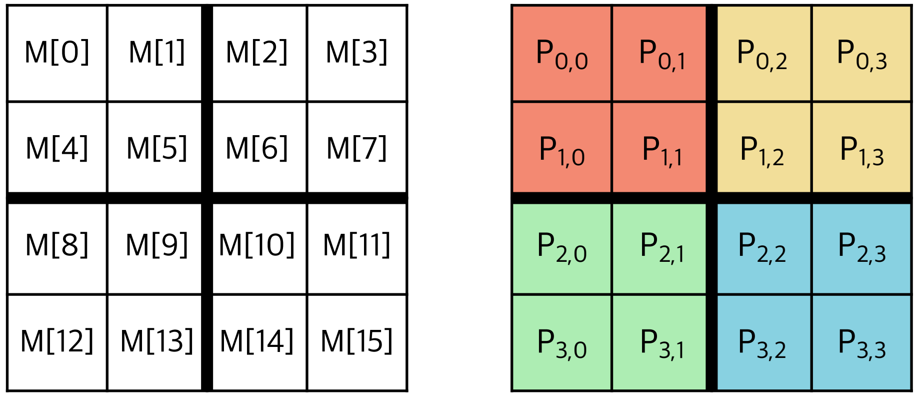


[Two-Bank channel access timeline]

6.2 Hiding memory latency

- To achieve the best bandwidth utilization, memory accesses must be evenly distributed across channels and banks, and each access to a bank must also be a coalesced access.

[Tiled matrix multiplication]


M[0] M[1] M[2] M[3]
M[4] M[5] M[6] M[7]
M[8] M[9] M[10] M[11]
M[12] M[13] M[14] M[15]
$P_{0,0}$ $P_{0,1}$ $P_{0,2}$ $P_{0,3}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$ $P_{1,3}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$ $P_{2,3}$
$P_{3,0}$ $P_{3,1}$ $P_{3,2}$ $P_{3,3}$

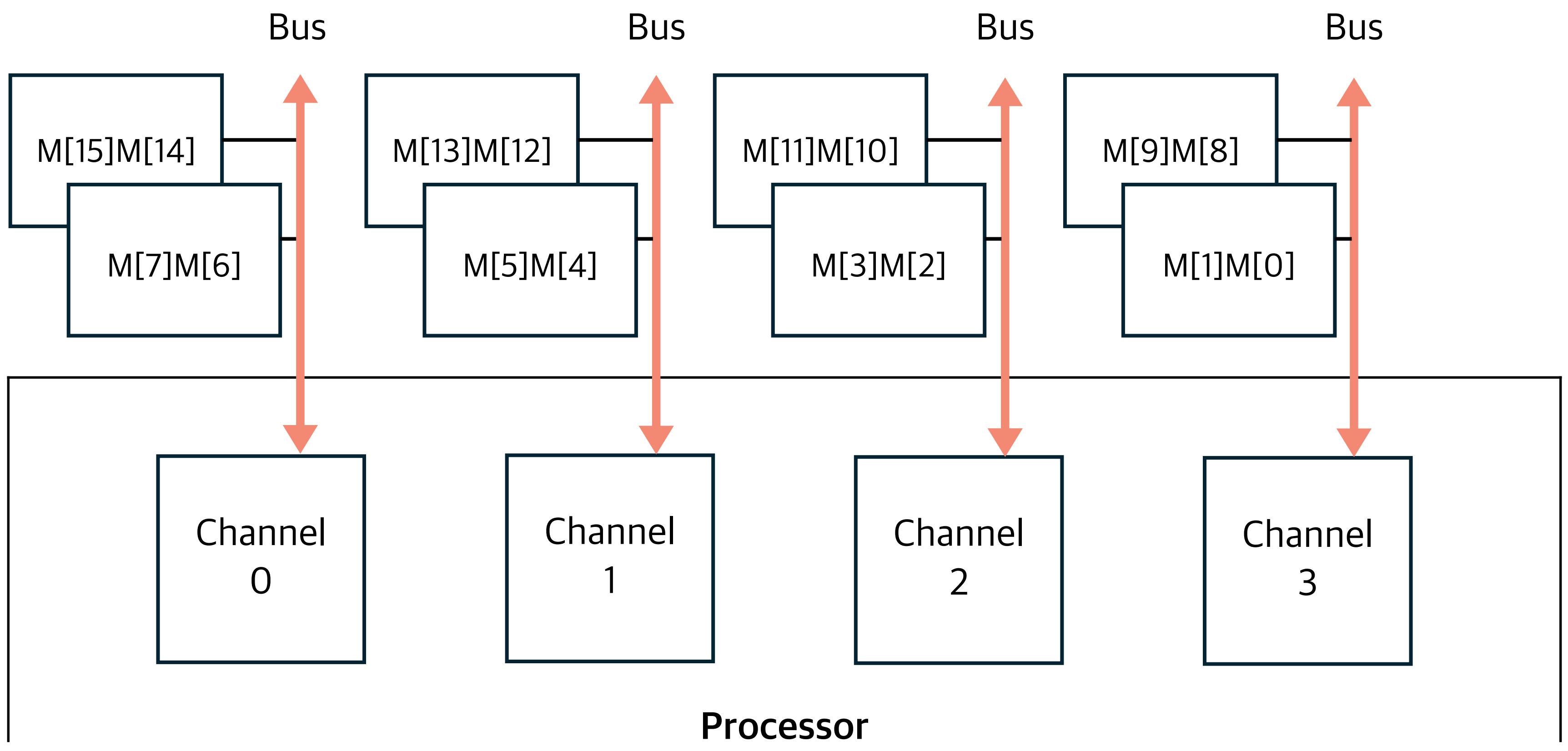
Input matrix M (linearized index)

output matrix P

	Block 0,0	Block 0,1	Block 1,0	Block 1,1
Phase 0	M[0], M[1], M[4], M[5]	M[0], M[1], M[4], M[5]	M[8], M[9], M[12], M[13]	M[8], M[9], M[12], M[13]
Phase 1	M[2], M[3], M[6], M[7]	M[2], M[3], M[6], M[7]	M[10], M[11], M[14], M[15]	M[10], M[11], M[14], M[15]

6.2 Hiding memory latency

	Block 0,0	Block 0,1	Block 1,0	Block 1,1
Phase 0	M[0], M[1], M[4], M[5]	M[0], M[1], M[4], M[5]	M[8], M[9], M[12], M[13]	M[8], M[9], M[12], M[13]
Phase 1	M[2], M[3], M[6], M[7]	M[2], M[3], M[6], M[7]	M[10], M[11], M[14], M[15]	M[10], M[11], M[14], M[15]



6.3 Thread coarsening

- The price of parallelism can take many forms, such as redundant loading of data by different thread blocks, redundant work, synchronization overhead, and others. The price of parallelism is often worth paying.

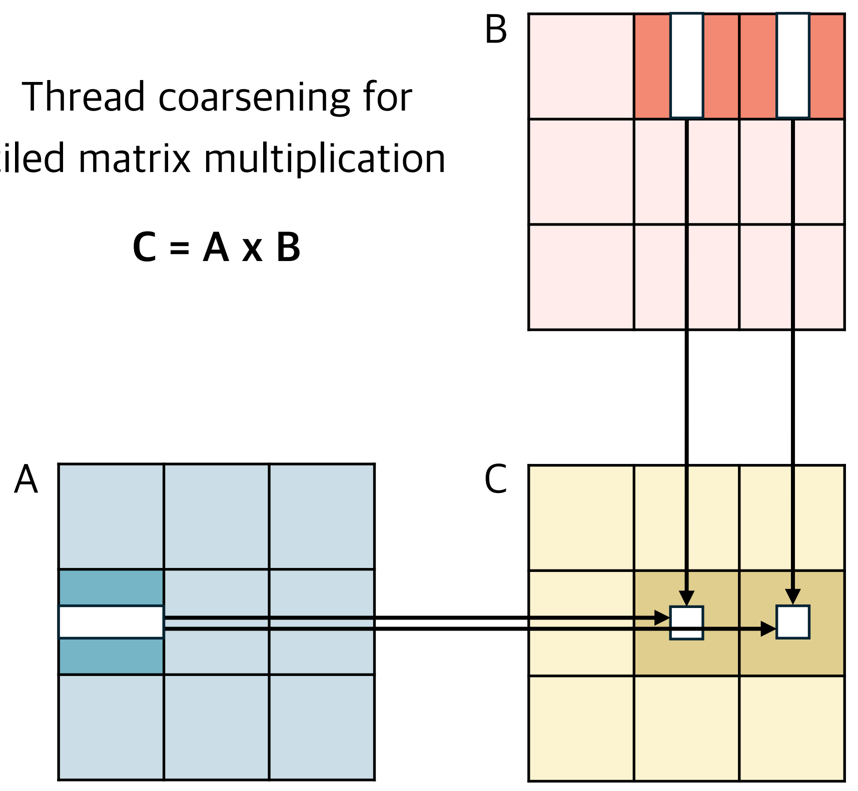
- However, if the hardware ends up serializing the work as a result of insufficient resource,
then this price has been paid unnecessarily.

- Thread coarsening, which assigns multiple units of work to each thread, can serialize the work and reduce the price that is paid for parallelism.

6.3 Thread coarsening

Thread coarsening for
tiled matrix multiplication

C = A x B



- Although having different thread blocks load the same input tile is redundant, it is a price that we pay to be able to process the two output tiles in parallel using different blocks.

- If these thread blocks are serialized by the hardware, the price is paid in vain. It is better to have a single thread block process the multiple output tiles.

6.4 A checklist of optimizations

Optimization	Benefit to compute cores	Benefit to memory	Strategies
Maximizing occupancy	More work to hide pipeline latency	More parallel memory accesses to hide DRAM latency	Tuning usage of resources such as threads per block, shared memory per block, and registers per thread
Enabling coalesced global memory accesses	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic and better utilization of bursts/ cache lines	Transfer between global memory and shared memory in a coalesced manner and performing uncoalesced accesses in shared memory (e.g., corner turning)
Minimizing control divergence	High SIMD efficiency (fewer idle cores during SIMD execution)	-	1. Rearranging the mapping of threads to work and/or data 2. Rearranging the layout of the data

6.4 A checklist of optimizations

Optimization	Benefit to compute cores	Benefit to memory	Strategies
Tiling of reused data	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic	Placing data that is reused within a block in shared memory or registers so that it is transferred between global memory and the SM only once
Privatization (covered later)	Fewer pipeline stalls waiting for atomic updates	Less contention and serialization of atomic updates	Applying partial updates to a private copy of the data and then updating the universal copy when done
Thread coarsening	Less redundant work, divergence, or synchronization	Less redundant global memory traffic	Assigning multiple units of parallelism to each thread to reduce the price of parallelism when it is incurred unnecessarily

6.5 Knowing your computation's bottleneck

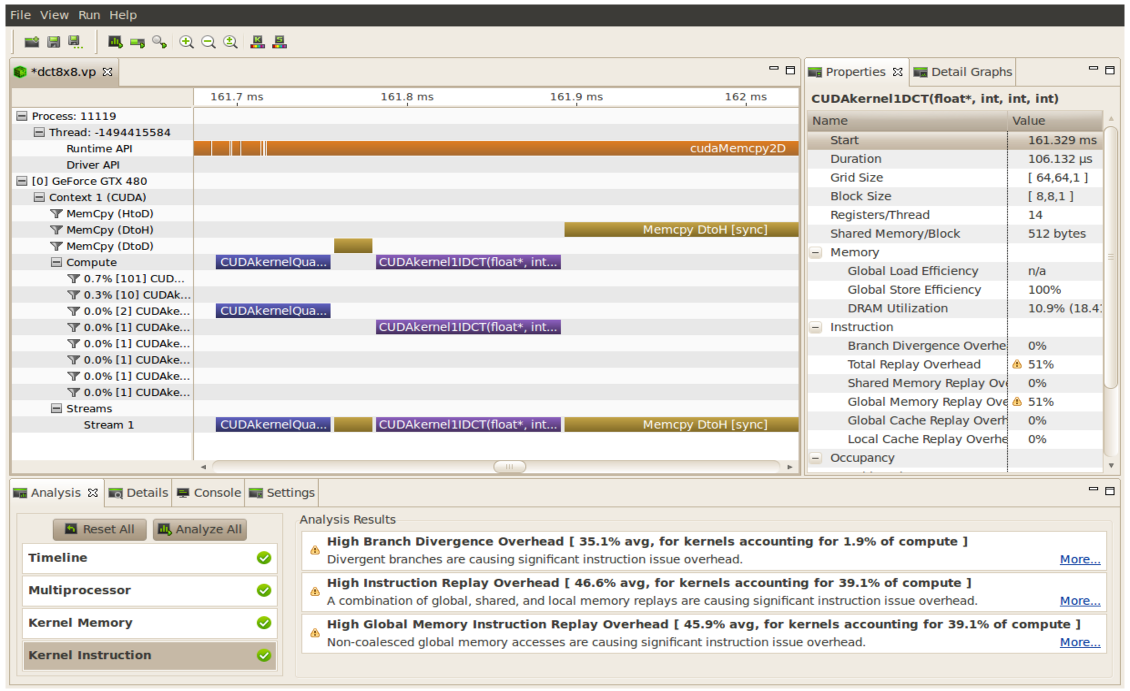
- The resource that limits the performance of a computation is often referred to as a performance **bottleneck**.
- If the optimization that is applied does not target the bottleneck resource, there are may be no benefit from the optimization.
- Performance bottlenecks may be **hardware-specific**, meaning that the same computation may encounter different bottlenecks on different devices.

6.5 Knowing your computation’s bottleneck

NVIDIA Visual Profiler

- Automated performance analysis:

  Perform automated analysis of your application to identify performance bottlenecks and get optimization suggestions that can be used to improve performance



CUDAkernel1DCT(float*, int, int, int)

Name	Value
Start	161.329 ms
Duration	106.132 μs
Grid Size	[64,64,1]
Block Size	[8,8,1]
Registers/Thread	14
Shared Memory/Block	512 bytes
Memory	
Global Load Efficiency	n/a
Global Store Efficiency	100%
DRAM Utilization	10.9% (18.4
Instruction	
Branch Divergence Overhe	0%
Total Replay Overhead	⚠ 51%
Shared Memory Replay Ove	0%
Global Memory Replay Ove	⚠ 51%
Global Cache Replay Overh	0%
Local Cache Replay Overhe	0%
Occupancy	

Analysis Results

High Branch Divergence Overhead [35.1% avg, for kernels accounting for 1.9% of compute]
Divergent branches are causing significant instruction issue overhead. More...

High Instruction Replay Overhead [46.6% avg, for kernels accounting for 39.1% of compute]
A combination of global, shared, and local memory replays are causing significant instruction issue overhead. More...

High Global Memory Instruction Replay Overhead [45.9% avg, for kernels accounting for 39.1% of compute]
Non-coalesced global memory accesses are causing significant instruction issue overhead. More...

https://developer.nvidia.com/nvidia-visual-profiler

6.6 Summary

- Methodology

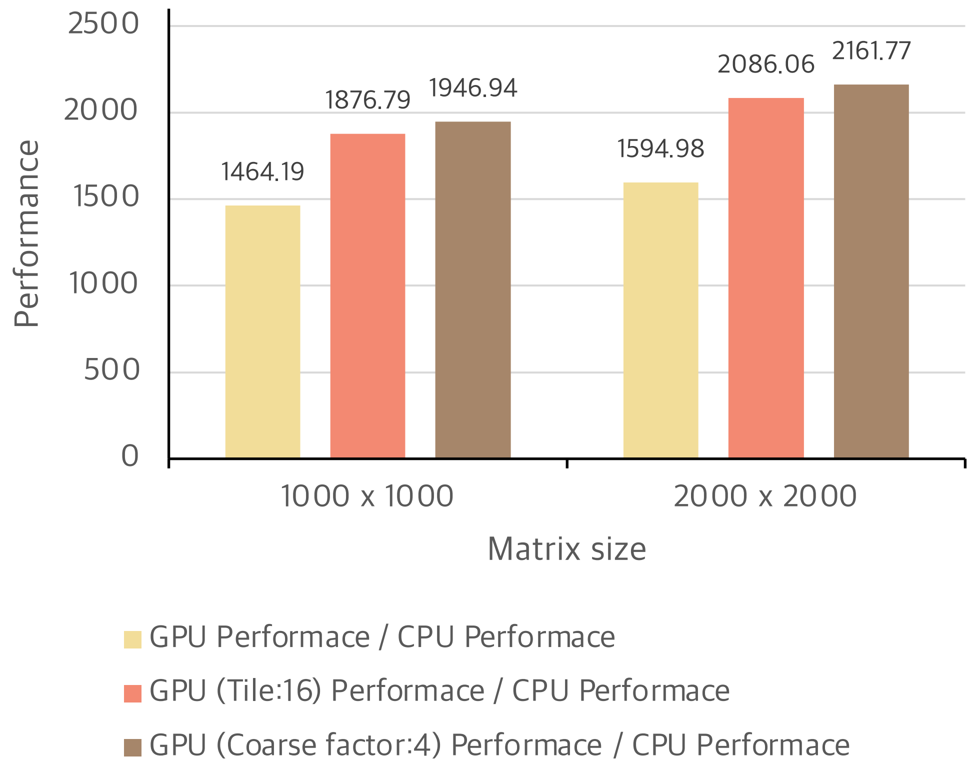
1. Ubuntu Version: Ubuntu 22.04.2 LTS
2. CUDA Version: 12.1
3. NVIDIA GeForce RTX 3080
4. Comparing the performance of matrix multiplication algorithms on CPU and GPU, where matrices of sizes 1000x1000 and 2000x2000 are employed, and the GPU block size is set to 16x16.
5. The algorithm using 'thread coarsening' set coarse factor to 4 and blocks four times less than other algorithms are used.

SM Count	68
Max resident threads per SM	1536
Max number of resident blocks per SM	16
Threads in warp	32
Max threads per block	1024
Max thread dimensions	(1024, 1024, 64)
Max grid dimensions	(2^{31}–1, 2^{16}–1, 2^{16}–1)
Shared Mem per SM	48 KB
Registers per SM	64 KB
Total constant Mem	64 KB
Total global Mem	10 GB

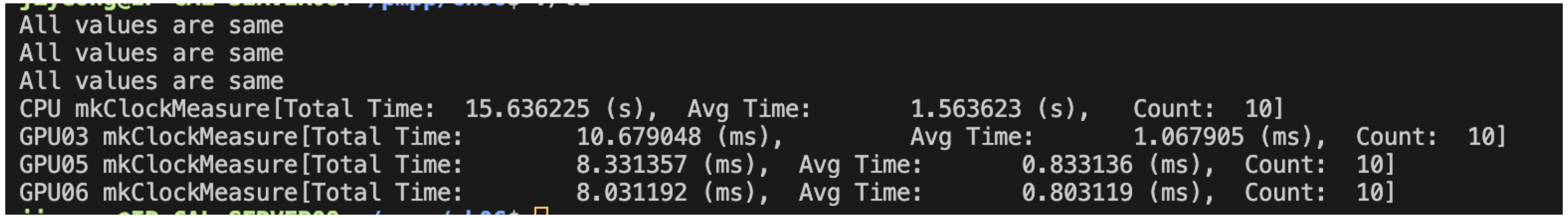
6.6 Summary

- Results

	CPU Execution Time	GPU Execution Time	GPU (Tile: 16) Execution Time	GPU (Coarse factor: 4) Execution Time
1000 x 1000	1.563623 (s)	1.067905 (ms)	0.833136 (ms)	0.803119 (ms)
2000 x 2000	13.195638 (s)	8.273207 (ms)	6.325630 (ms)	6.104082 (ms)



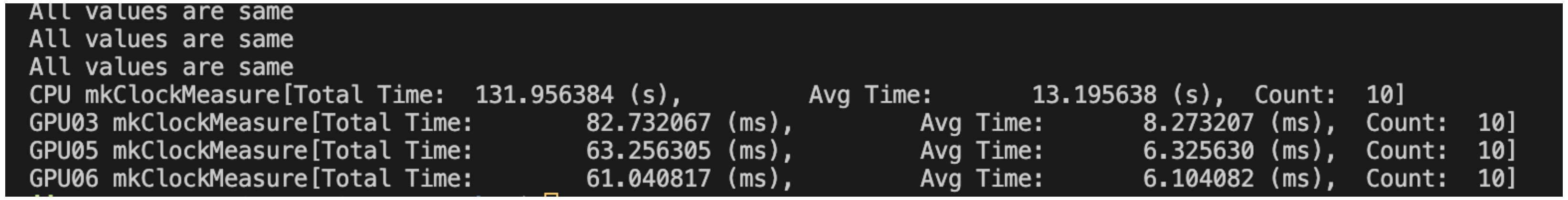
1000x1000
block size = 16X16
tile size = 16x16
COARSE_FACTOR = 4



```
All values are same
All values are same
All values are same
CPU mkClockMeasure[Total Time:  15.636225 (s),  Avg Time:       1.563623 (s),   Count:  10]
GPU03 mkClockMeasure[Total Time:        10.679048 (ms),         Avg Time:       1.067905 (ms),  Count:  10]
GPU05 mkClockMeasure[Total Time:        8.331357 (ms),  Avg Time:       0.833136 (ms),  Count:  10]
GPU06 mkClockMeasure[Total Time:        8.031192 (ms),  Avg Time:       0.803119 (ms),  Count:  10]
```

CPU mkClockMeasure[Total Time: 15.636225 (s), Avg Time: 1.563623 (s),   Count: 10]
GPU03 mkClockMeasure[Total Time: 10.679048 (ms),   Avg Time: 1.067905 (ms), Count: 10]
GPU05 mkClockMeasure[Total Time: 8.331357 (ms), Avg Time: 0.833136 (ms), Count: 10]
GPU06 mkClockMeasure[Total Time: 8.031192 (ms), Avg Time: 0.803119 (ms), Count: 10]

2000x2000
block size = 16X16
tile size = 16x16
COARSE_FACTOR = 4



```
All values are same
All values are same
All values are same
CPU mkClockMeasure[Total Time:  131.956384 (s),         Avg Time:       13.195638 (s),  Count:  10]
GPU03 mkClockMeasure[Total Time:        82.732067 (ms),         Avg Time:       8.273207 (ms),  Count:  10]
GPU05 mkClockMeasure[Total Time:        63.256305 (ms),         Avg Time:       6.325630 (ms),  Count:  10]
GPU06 mkClockMeasure[Total Time:        61.040817 (ms),         Avg Time:       6.104082 (ms),  Count:  10]
```

CPU mkClockMeasure[Total Time: 131.956384 (s),   Avg Time: 13.195638 (s), Count: 10]
GPU03 mkClockMeasure[Total Time: 82.732067 (ms),   Avg Time: 8.273207 (ms), Count: 10]
GPU05 mkClockMeasure[Total Time: 63.256305 (ms),   Avg Time: 6.325630 (ms), Count: 10]
GPU06 mkClockMeasure[Total Time: 61.040817 (ms),   Avg Time: 6.104082 (ms), Count: 10]