Programming Massively Parallel Processors

Chapter05 Memory architecture and data locality
&
GPU resources / Warp scheduling

1944023 JiYeong Yi

Chapter04

Compute architecture and scheduling

Table of contents

4.3 Synchronization and transparent scalability

4.4 Warps and SIMD hardware

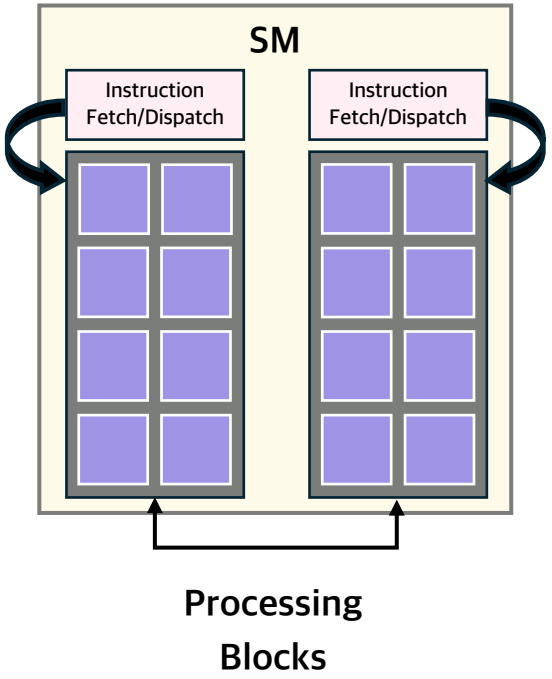
4.5 Control divergence

4.6 Warp scheduling and latency tolerance

4.7 Resource partitioning and occupancy

4.4 Warps and SIMD hardware

- An SM is designed to execute all threads in a **warp** following the single-instruction, multiple-data (**SIMD**) model. That is, at any instant in time, one instruction is fetched and executed for all thread in the warp.



- Cores in SM are grouped into **processing blocks** and share an instruction fetch/ dispatch unit.
- **Threads in the same warp** are assigned to the **same processing block**, which fetches the instruction for the warp and executes it for all threads in the warp at the same time.
- The advantage of SIMD is that the cost of the control hardware, such as the instruction fetch/dispatch unit, is shared across many execution units.

4.5 Control divergence

- SIMD execution works well when all threads within a warp follow the same execution path, more formally referred to as **control flow**, when working on their data.

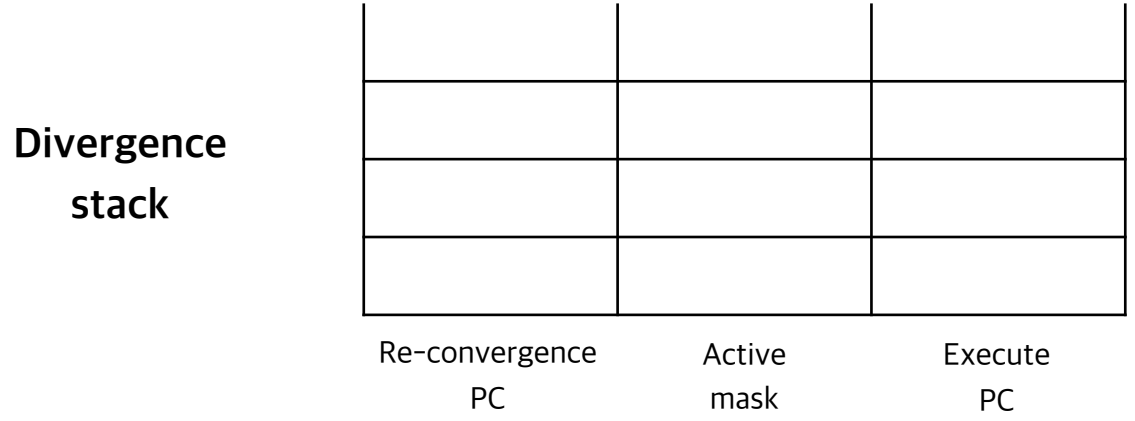
- However, when threads within a warp take different control flow paths, the SIMD hardware will take multiple passes through these paths, one pass for each path.

- When threads in the same warp follow different execution paths, we say that these threads exhibit **control divergence**, that is, they diverge in their execution.

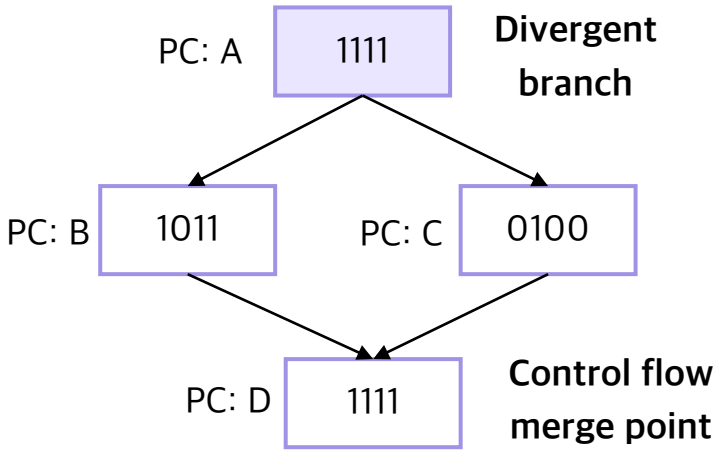
4.5 Control divergence

- Since a warp can only have a single active PC at any given time, when branch divergence occurs, one path must be chosen first and the other is pushed on a divergence stack associated with the warp so that it can be executed later.

- A divergence stack entry consists of three fields: re-convergence PC, an active mask, and an execute PC.



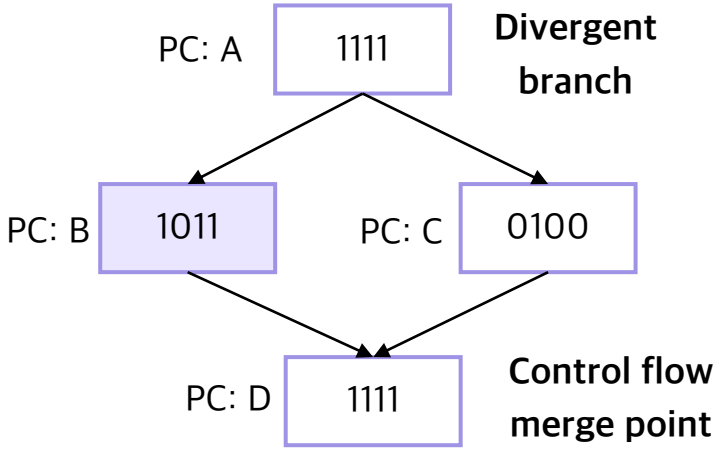
4.5 Control divergence



- When a warp encounters a diverge branch, push a join entry onto the divergence stack.
- This entry has both the re-convergence and execute PCs equal to the compiler identified control flow merge point of the branch.

D	1111	D
Re-convergence PC	Active mask	Execute PC

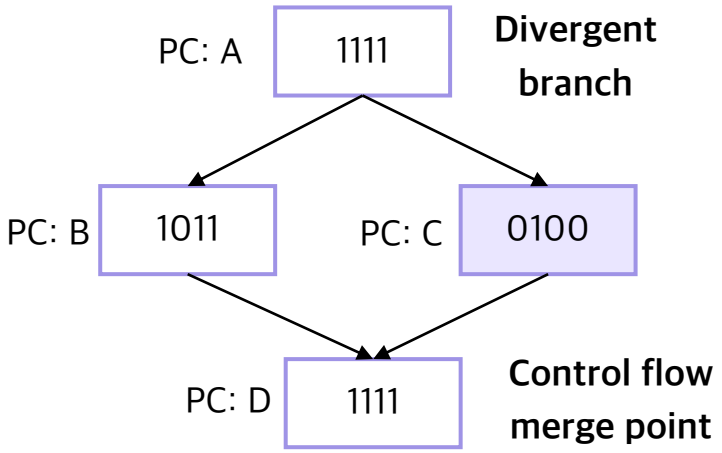
4.5 Control divergence



- One of the two divergent paths is selected to execute first and the current PC and active mask of the warp are updated.
- Another entry, the divergence entry, is pushed on the divergence stack. The re-convergence PC is set equal to the control flow merge point of the divergence branch.

D	0100	C
D	1111	D
Re-convergence PC	Active mask	Execute PC

4.5 Control divergence



- Each time warp reaches the last pipeline stage, the warp's next PC is compared to the re-convergence PC at the top of the stack.
- If equal, the stack is popped, and the active mask and PC of the warp are updated with the active mask and execute PC fields of the popped entry.

D	1111	D
Re-convergence PC	Active mask	Execute PC

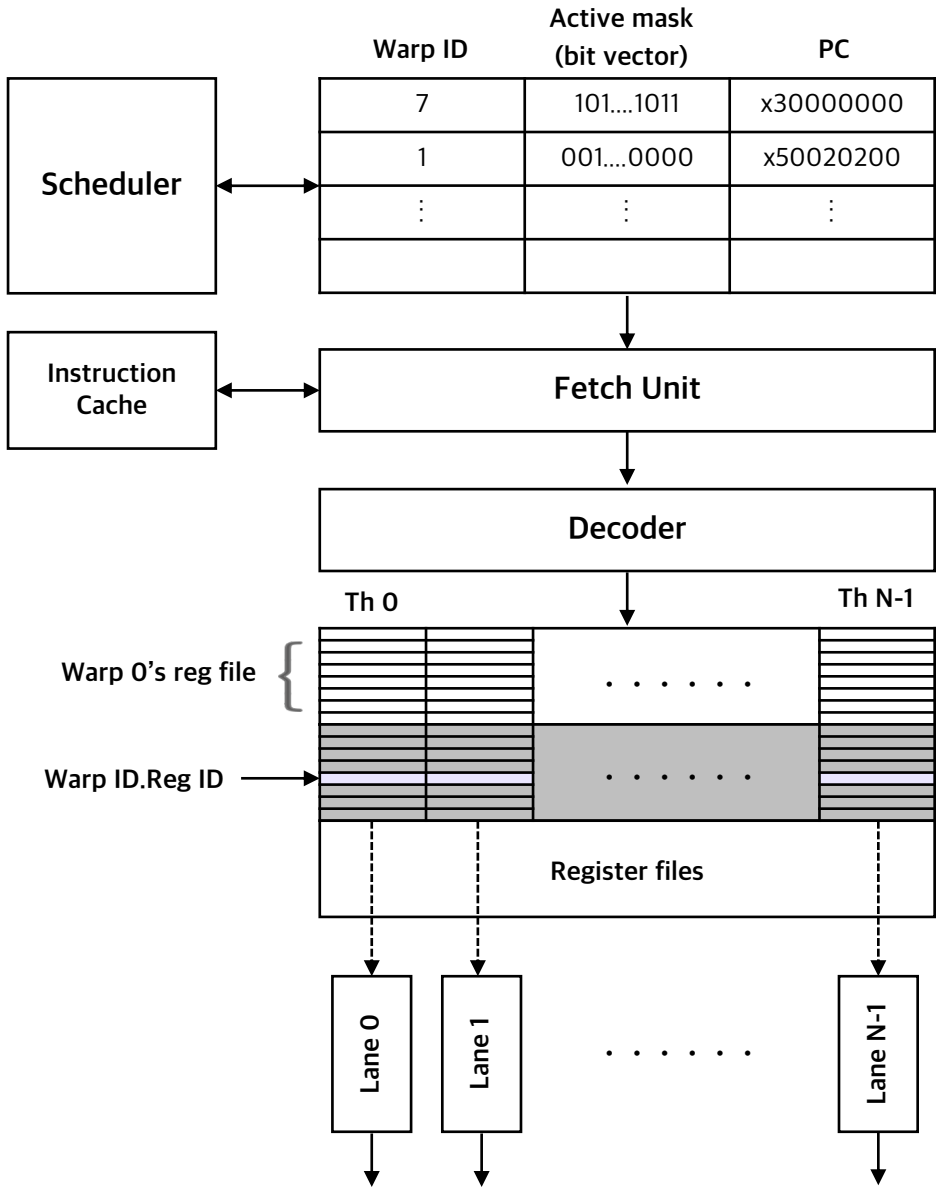
4.6 Warp scheduling and latency tolerance

- When threads are assigned to SMs, there are usually more threads assigned to an SM than there are cores in the SM.

- That is, each SM has only enough execution units to execute a subset of all the threads assigned to it at any point in time.

- The mechanism of filling the latency time of operations from some threads with work from other threads is often called “**latency tolerance**” or “**latency hiding**”.

4.6 Warp scheduling and latency tolerance



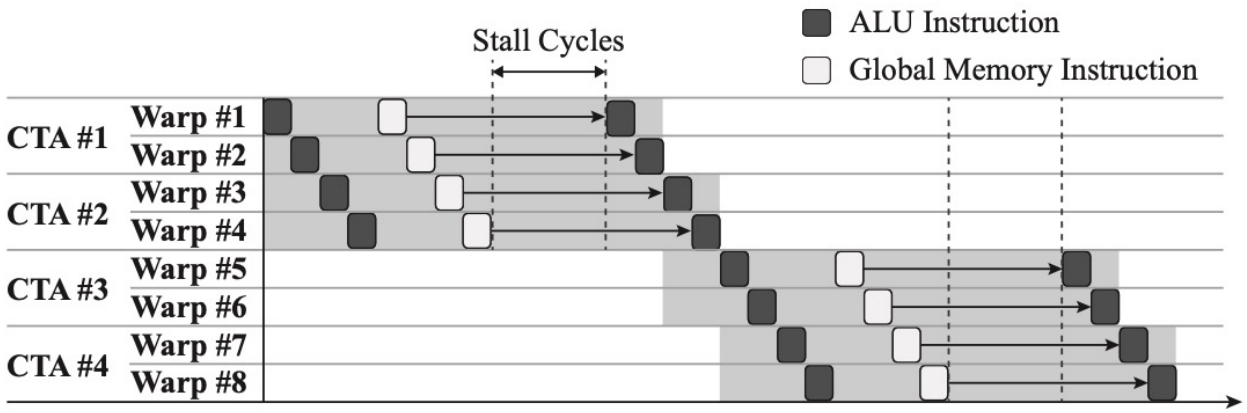
- In the fetch stage, the schedular selects a warp from the list of ready warps using a RR policy.
- After a warp is selected by the scheduler, the instruction cache is accessed at the PC of the warp and the fetch instruction is decoded.
- Register values for all threads in the warp are read in parallel from the register file.
- The register values are processed in parallel across multiple SIMD lanes.
- Once a warp reaches the final stage of the pipelines, its PC and active mask are updated and the warp is again considered for scheduling.

4.6 Warp scheduling and latency tolerance

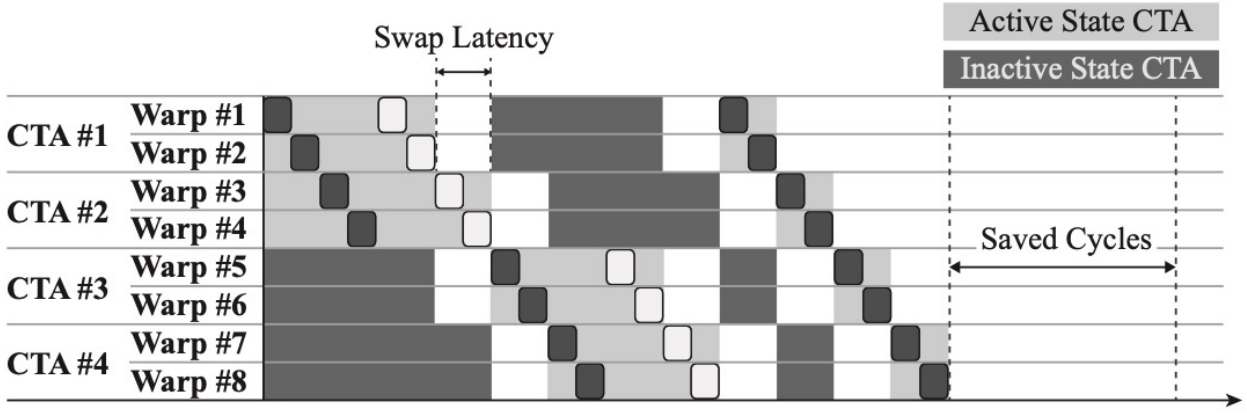
- The selection of warps that are ready for execution does not introduce any idle or wasted time into the execution timeline, which is referred to as **zero-overhead thread scheduling**.

- GPU SMs achieves zero-overhead scheduling by **holding all the execution states for the assigned warps in the hardware registers** so there is no need to save and restore states when switching from one warp to another.

4.6 Warp scheduling and latency tolerance



(a) Execution Order on Baseline GPU



(b) Execution Order using VT Architecture

4.7 Resource partitioning and occupancy

- It is desirable to assign many warps to an SM in order to tolerate long-latency operations. However, it may not always be possible to assign to the SM the maximum number of warps that the SM supports.

- Execution resources are **dynamically partitioned** across threads to support their execution.

  (+) This ability to dynamically partition thread slots among blocks makes SMs versatile.

  (-) Dynamic partitioning of resources can lead to subtle interactions between resource limitations, which can cause underutilization of resources.

4.7 Resource partitioning and occupancy

- The execution resources in an SM include registers, shared memory, thread block slots, and thread slots.

(Ex) NVIDIA Kepler GPU

- Register file size: 64K x 32 bit
- Shared memory per SM: 48KB
- Maximum number of threads per SM: 2048
- Maximum number of blocks per SM: 16

4.7 Resource partitioning and occupancy

$$\text{Occupancy} = \frac{\text{the number of threads assigned to an SM}}{\text{the maximum number of threads supported by an SM}}$$

Chapter05

Memory architecture and data locality

Table of contents

5.1 Importance of memory access efficiency

5.2 CUDA memory types

5.3 Tiling for reduced memory traffic

5.4 A tiled matrix multiplication kernel

5.5 Boundary checks

5.6 Impact of memory using on occupancy

5.7 Summary

5.1 Importance of memory access efficiency

- Global memory, which is typically implemented with off-chip DRAM,
  tends to have long access latency (hundreds of clock cycles) and finite access bandwidth.

- To circumvent traffic congestion in the global memory access paths, GPUs provide a number of **additional on-chip memory resources** for accessing data that can remove the majority of traffic to and from the global memory.

5.1 Importance of memory access efficiency

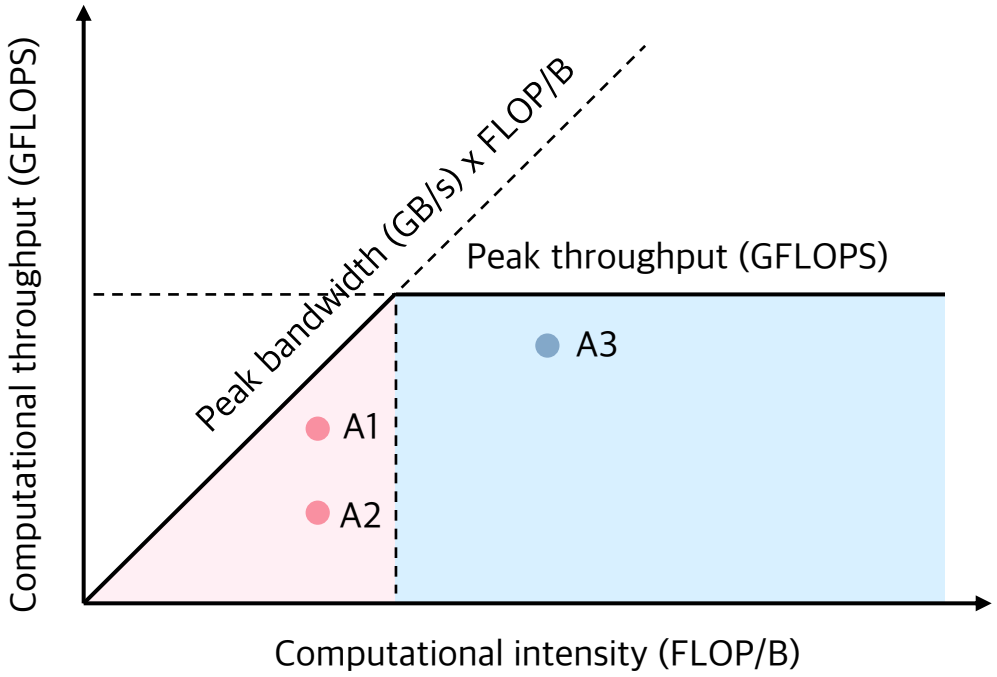
- **Compute to global memory access ratio** (Arithmetic intensity or Computational intensity):
  The number of FLOPs performed for each byte access from the global memory within a region of a program.

```
for (int k = 0; k < Width; k++){
  Pvalue += M[row*Width+K] * N[k*Width+col];
}
```

- Floating-point operations (FLOPs): 2 FLOP
- Bytes (B) accessed from global memory: 8 B
- Compute to global memory access ratio: 0.25 FLOP/B

5.1 Importance of memory access efficiency

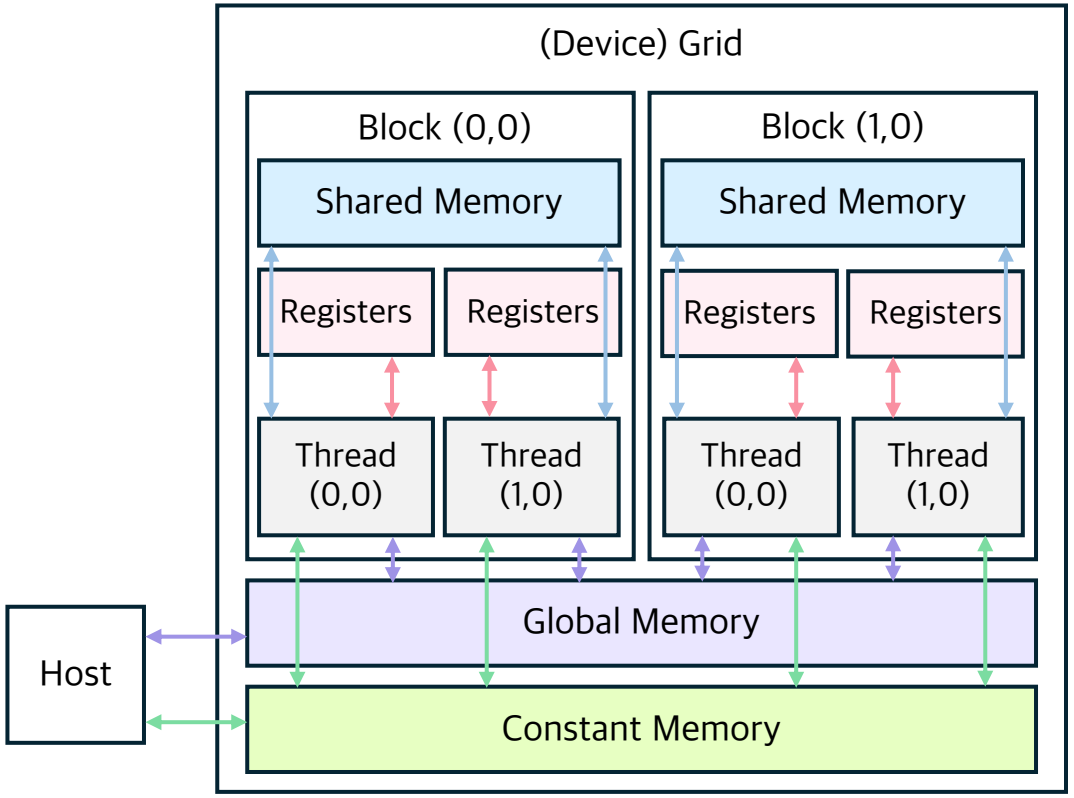
- **The Roofline Model** is a visual model for assessing the performance achieved by an application relative to the limits of the hardware it is running on.



- **Memory-bound Application**:
  Applications with lower computational intensity cannot achieve peak throughput because they are limited by memory bandwidth.

- **Compute-bound Application**:
  Applications with higher computational intensity are not limited by memory bandwidth.

5.2 CUDA memory types



On-chip memory

- Shared Memory
- Registers

Off-chip memory

- Global Memory
- Local Memory
- Constant Memory

5.2 CUDA memory types

- On-chip memory has short access latencies and relatively higher access bandwidth.

Registers

- Allocated to individual threads; each thread can access only its own registers.
- A kernel function typically uses registers to hold frequently accessed variables that are private to each thread.

Shared Memory

- Allocated to thread blocks; all threads in a block can access shared memory.
- It is an efficient means by which threads can cooperate by sharing their data and intermediate results.

5.2 CUDA memory types

- Off-chip memory has long access latencies and relatively low access bandwidth.

Global Memory

- Can be written and read by the host and device.

Local Memory

- Can be written or read by the host and device.
- It is placed in global memory and has similar access latency, but it is not shared across threads.

Constant Memory

- Can be written and read by the host and support short-latency, high-bandwidth read-only access by the device.

5.2 CUDA memory types

- CUDA variable declaration type qualifiers and the properties of each type.

Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Grid
Automatic array variables	Local	Thread	Grid
__device__ __shared__ int SharedVar;	Shared	Block	Grid
__device__ int GlobalVar;	Global	Grid	Application
__device__ __constant__ int ConstVar;	Constant	Grid	Application

5.3 Tiling for reduced memory traffic

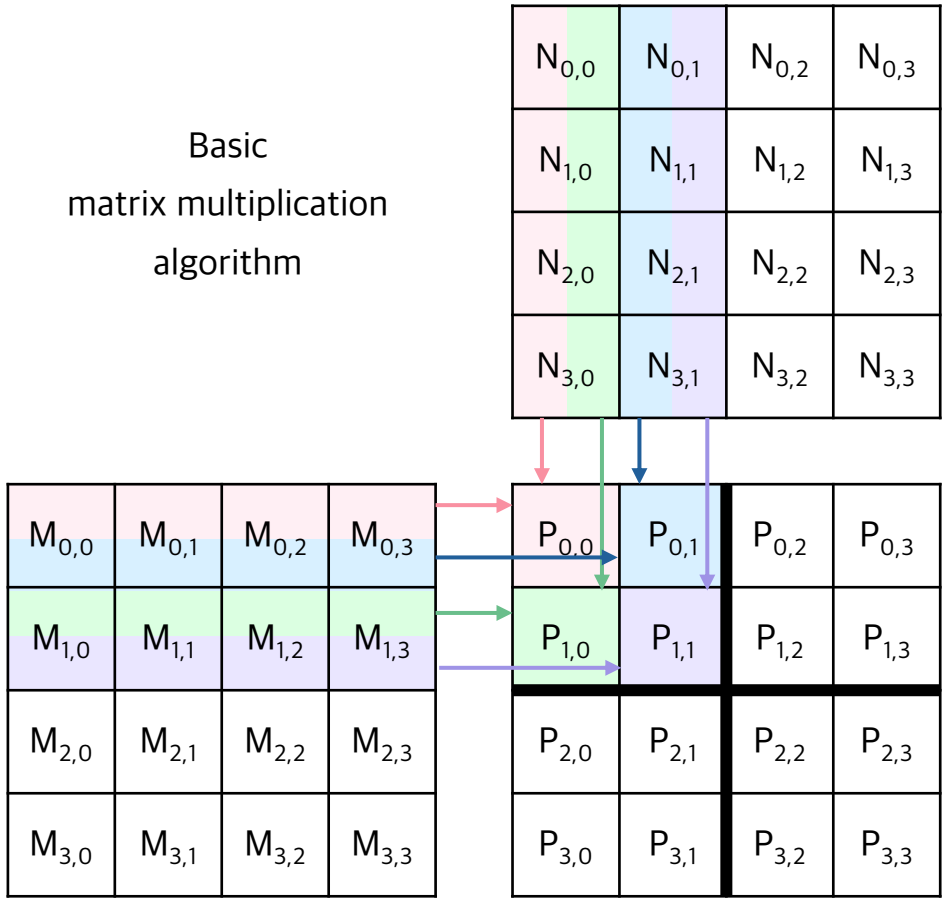
Tiling

- A common strategy is to partition the data into subsets called tiles so that each tile fits into the shared memory.

- The term tile draws on the analogy that a large wall (i.e., the global memory data) can be covered by small tiles (i.e., subsets that can each fit into the shared memory).

5.3 Tiling for reduced memory traffic

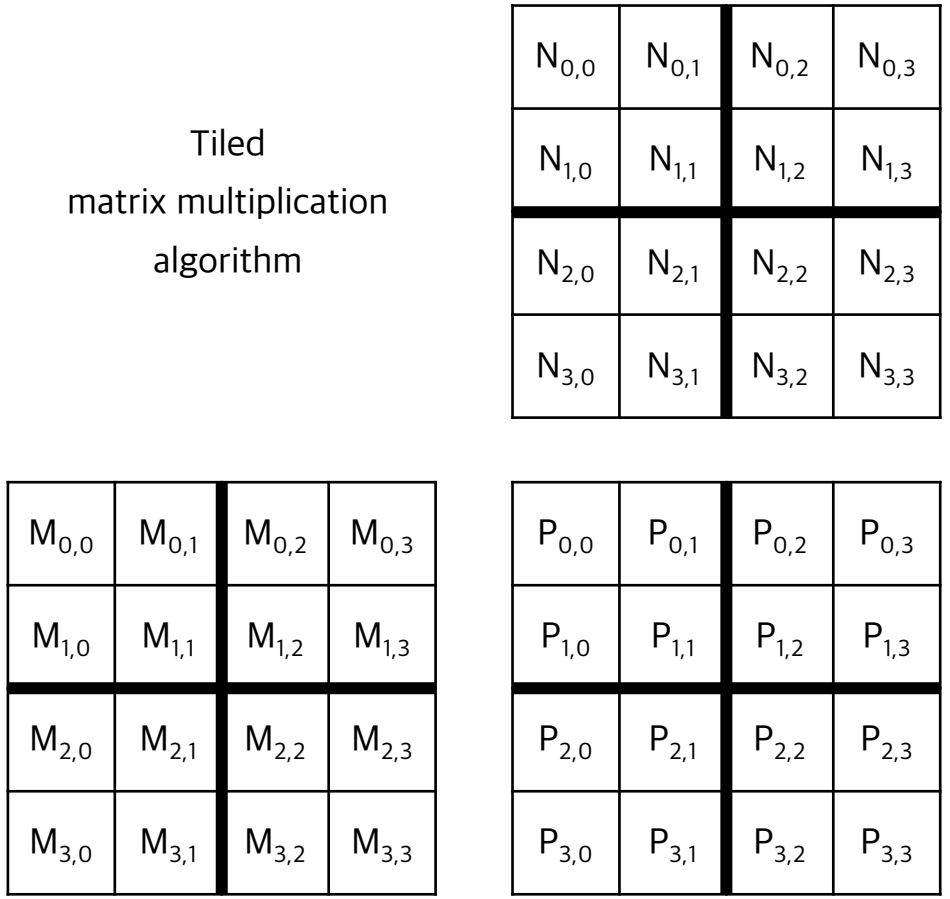
Basic
matrix multiplication
algorithm



- Global memory accesses done by all threads.
- Every M and N element is accessed exactly twice during the execution of $block_{0,0}$.

5.3 Tiling for reduced memory traffic

Tiled
matrix multiplication
algorithm

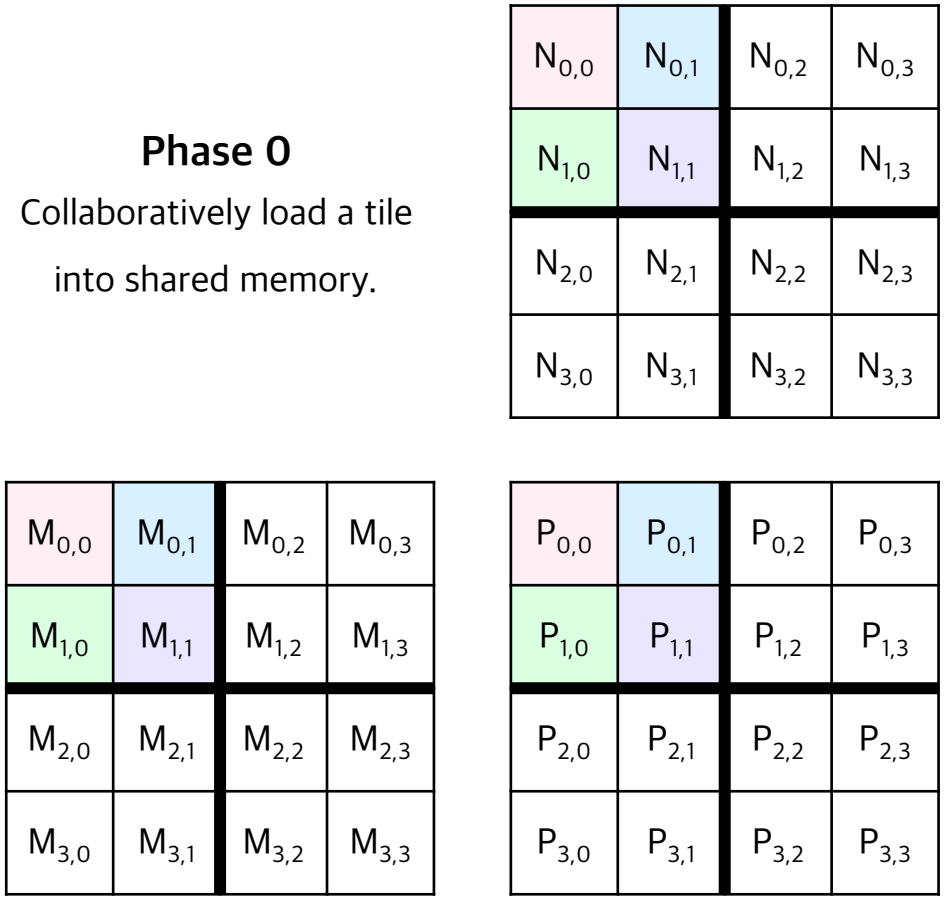


- The basic idea of tiled matrix multiplication algorithm is to have the threads collaboratively load subsets of the M and N elements into the shared memory before they individually use these elements in their dot product calculation.

5.3 Tiling for reduced memory traffic

Phase 0

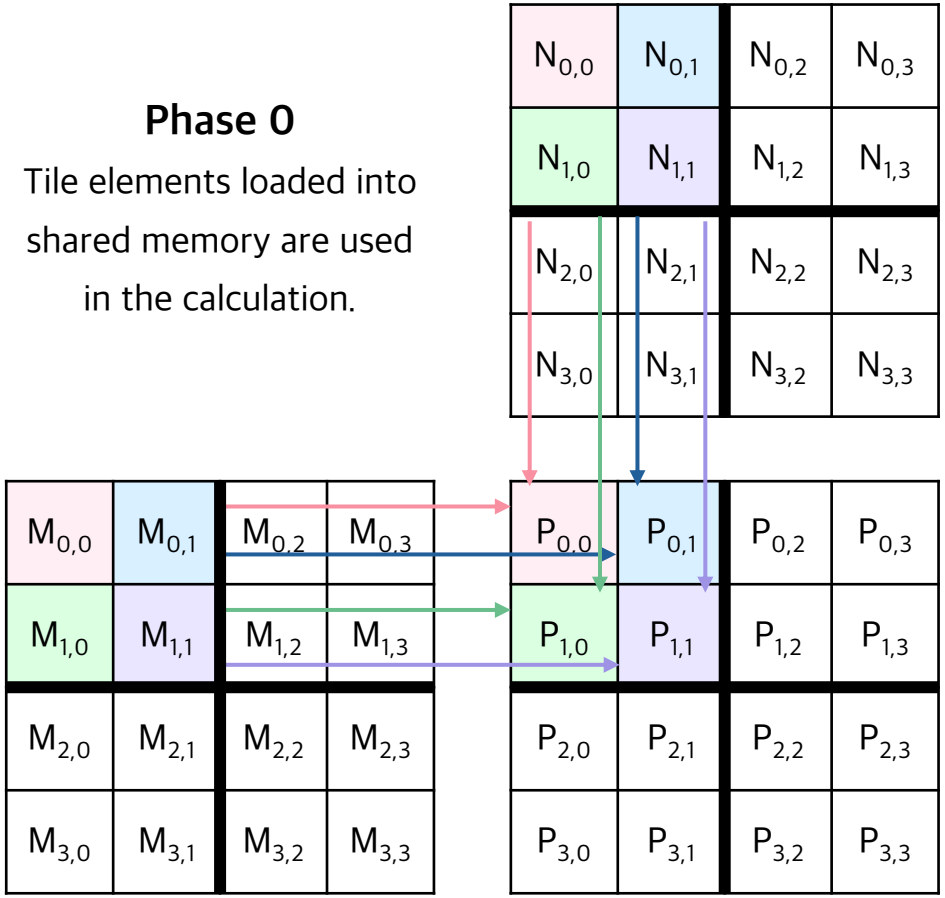
Collaboratively load a tile into shared memory.



5.3 Tiling for reduced memory traffic

Phase 0

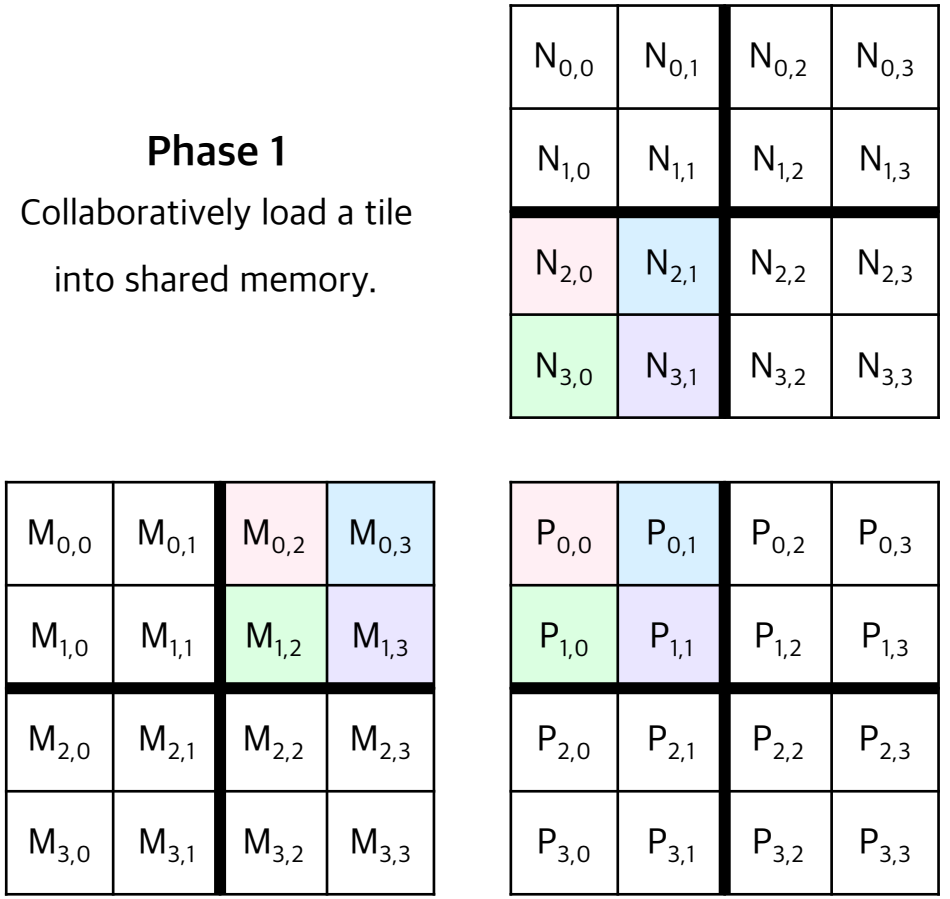
Tile elements loaded into shared memory are used in the calculation.



5.3 Tiling for reduced memory traffic

Phase 1

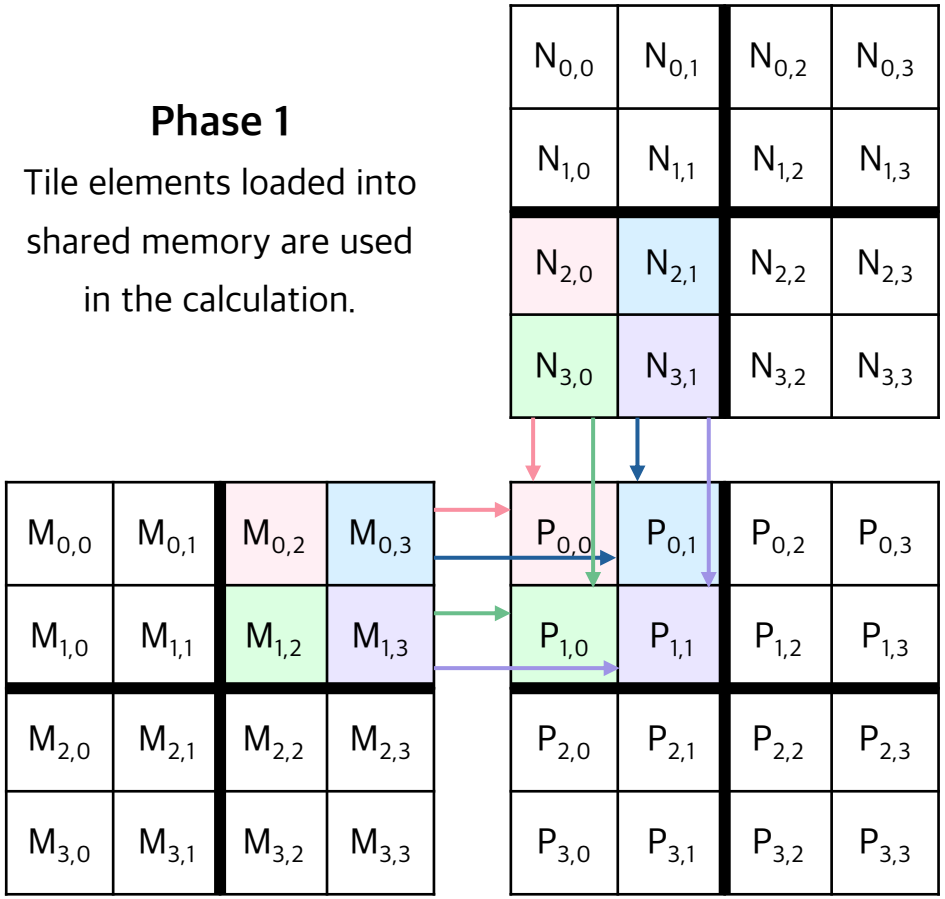
Collaboratively load a tile into shared memory.



5.3 Tiling for reduced memory traffic

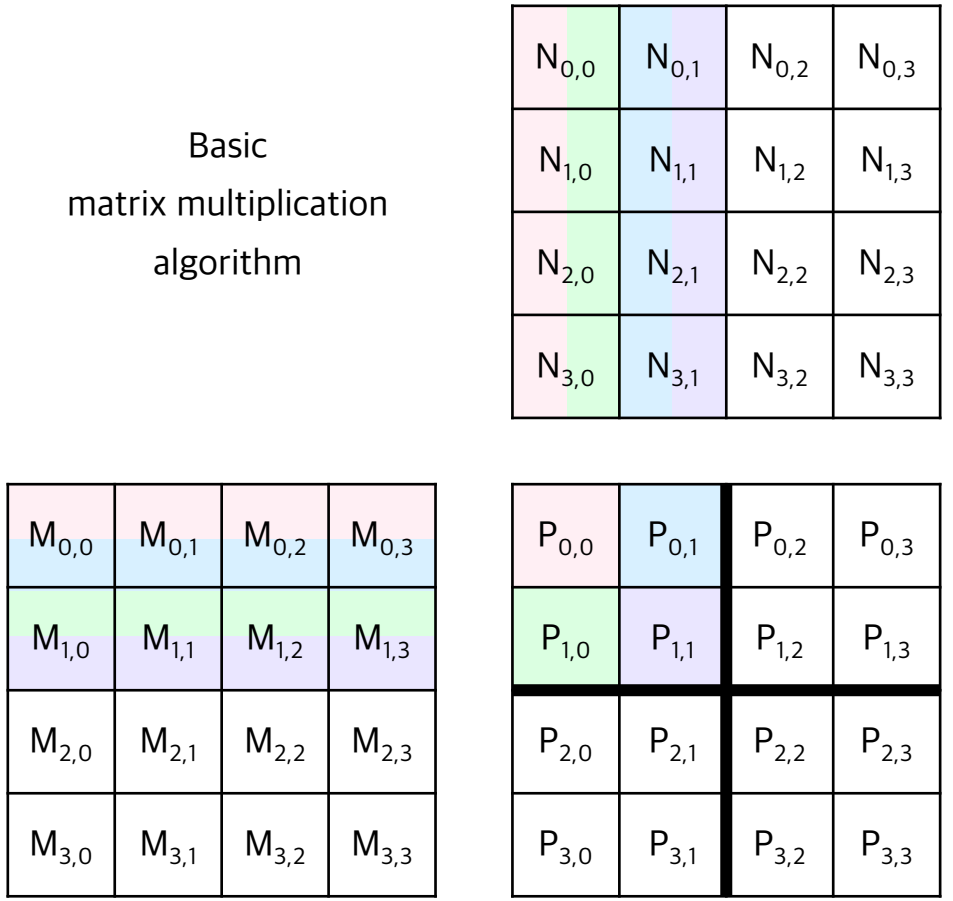
Phase 1

Tile elements loaded into shared memory are used in the calculation.

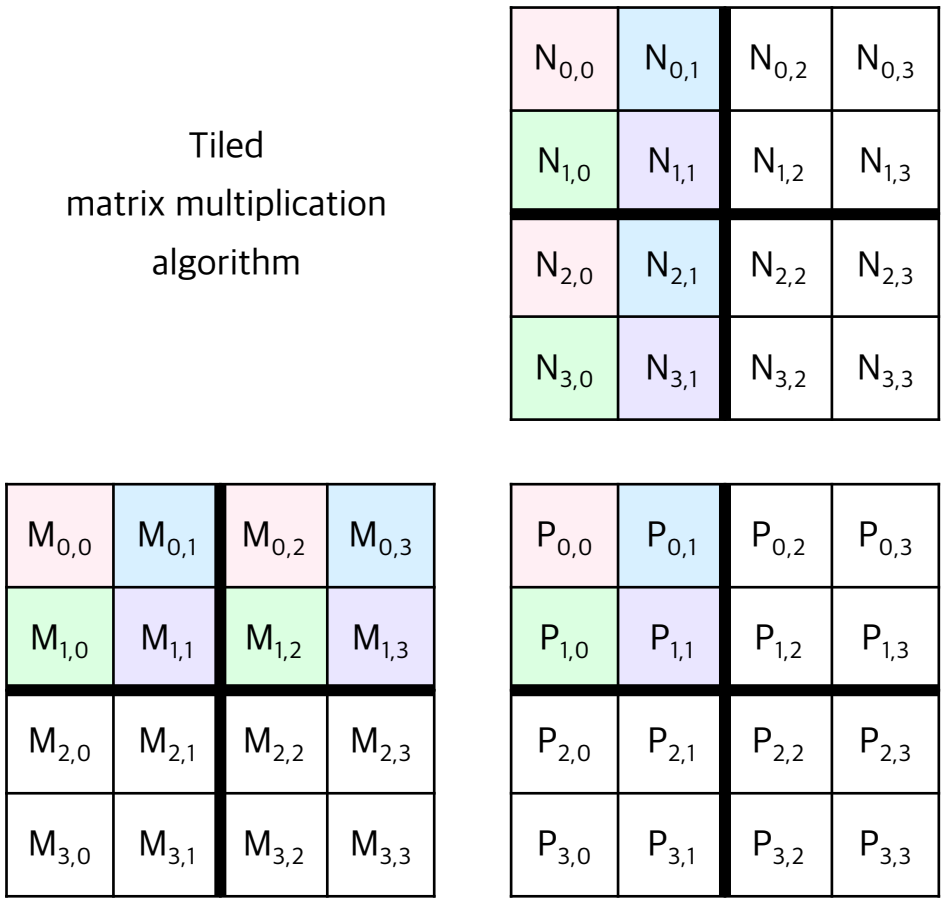


5.3 Tiling for reduced memory traffic

Basic
matrix multiplication
algorithm



Tiled
matrix multiplication
algorithm



The reduction is by a factor of N if the tiles are N x N elements.

5.4 A tiled matrix multiplication kernel

```
// Loop over the M and N tiles required to compute P element
float Pvalue = 0;
for (int ph = 0; ph < Width/TILE_WIDTH; ++ph){

  // Collaborative loading of M and N tiles into shared memory
  Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
  Nds[ty][tx] = N[(ph*TILE_WIDTH + ty) * Width + Col];
  __syncthreads();

  for (int k = 0; k < TILE_WIDTH; ++k){
    Pvalue += Mds[ty][k] * Nds[k][tx];
  }
  __syncthreads();
}
P[Row*Width + Col] = Pvalue;
```

1

2

The data dependence is caused by the fact that they are reusing the same memory location.

5.4 A tiled matrix multiplication kernel

```
// Loop over the M and N tiles required to compute P element
float Pvalue = 0;
for (int ph = 0; ph < Width/TILE_WIDTH; ++ph){

  // Collaborative loading of M and N tiles into shared memory
  Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
  Nds[ty][tx] = N[(ph*TILE_WIDTH + ty) * Width + Col];
1 __syncthreads();

  for (int k = 0; k < TILE_WIDTH; ++k){
    Pvalue += Mds[ty][k] * Nds[k][tx];
  }
2 __syncthreads();
}
P[Row*Width + Col] = Pvalue;
```

Two different types of data dependence

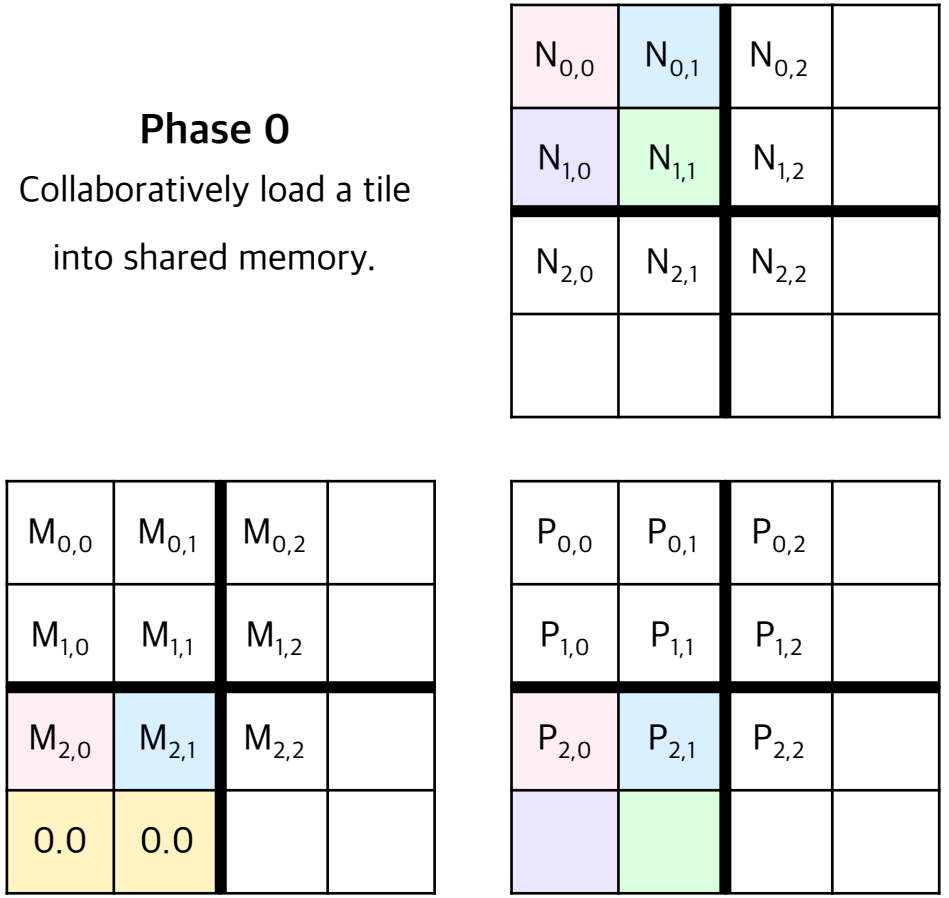
1. Read-after-write dependence (true dependence): Because thread must wait for data to be written to the proper place by other threads before they try to read it.

2. Write-after-read dependence (false dependence): Because a thread must wait for data to be read by all threads that need it before overwriting it.

5.5 Boundary checks

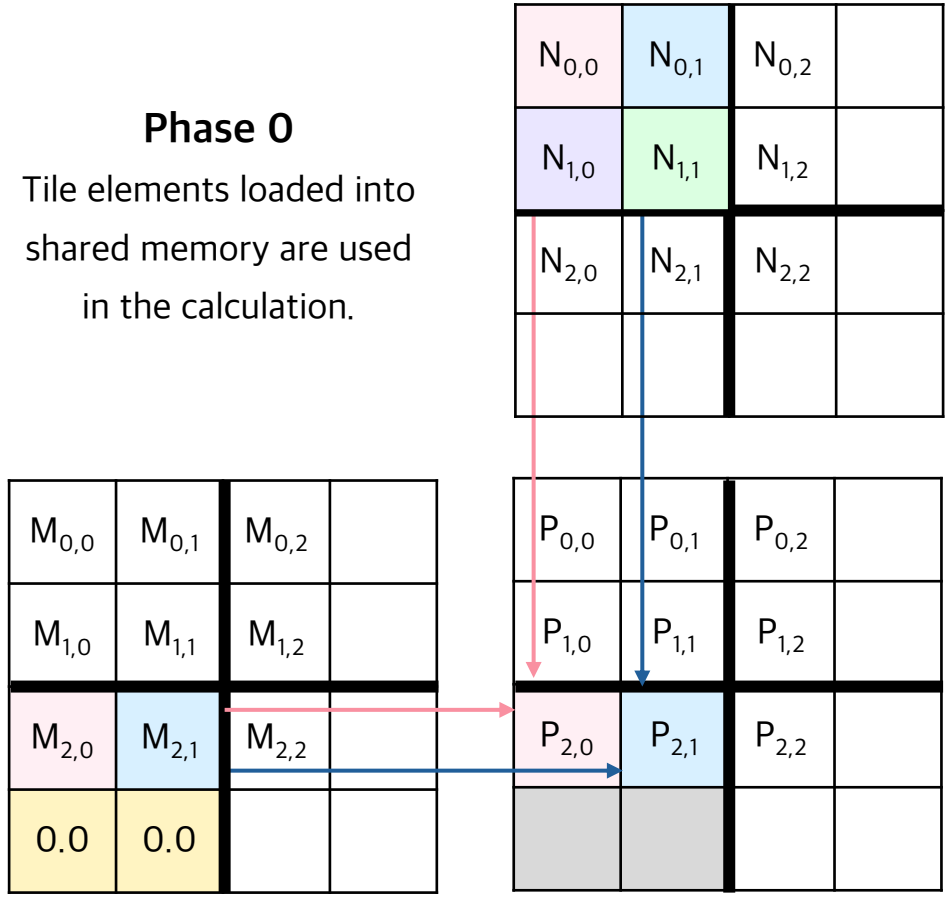
Phase 0

Collaboratively load a tile into shared memory.



Phase 0

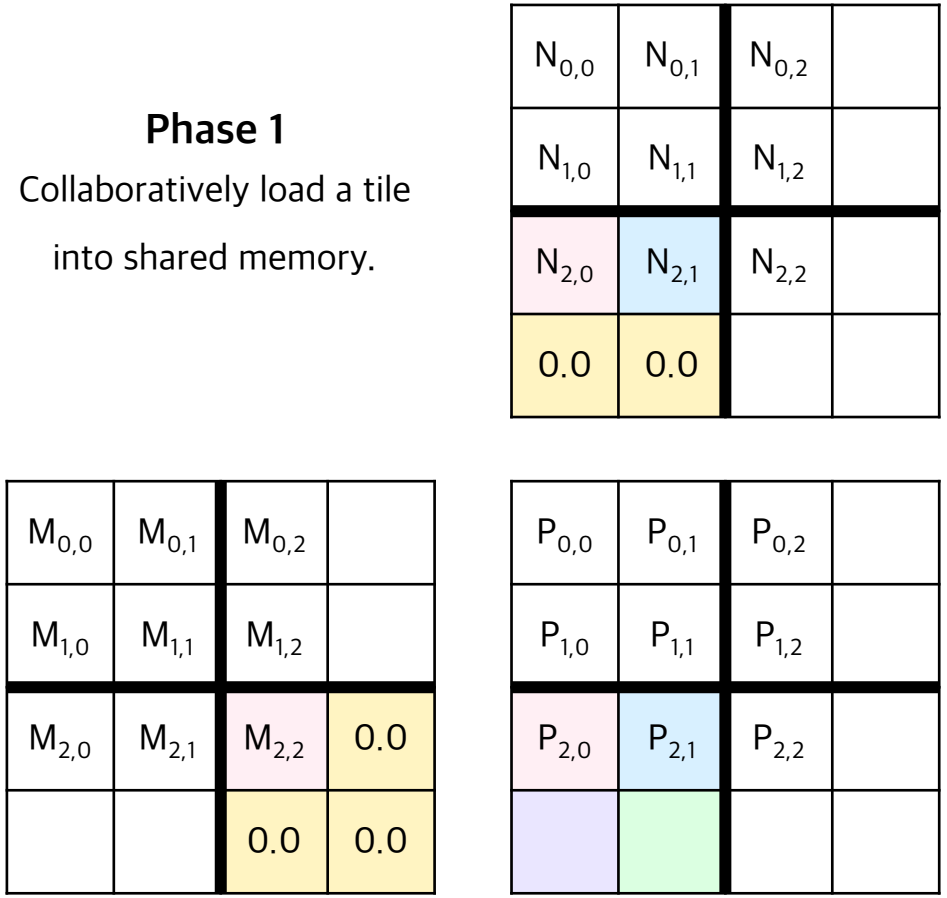
Tile elements loaded into shared memory are used in the calculation.



5.5 Boundary checks

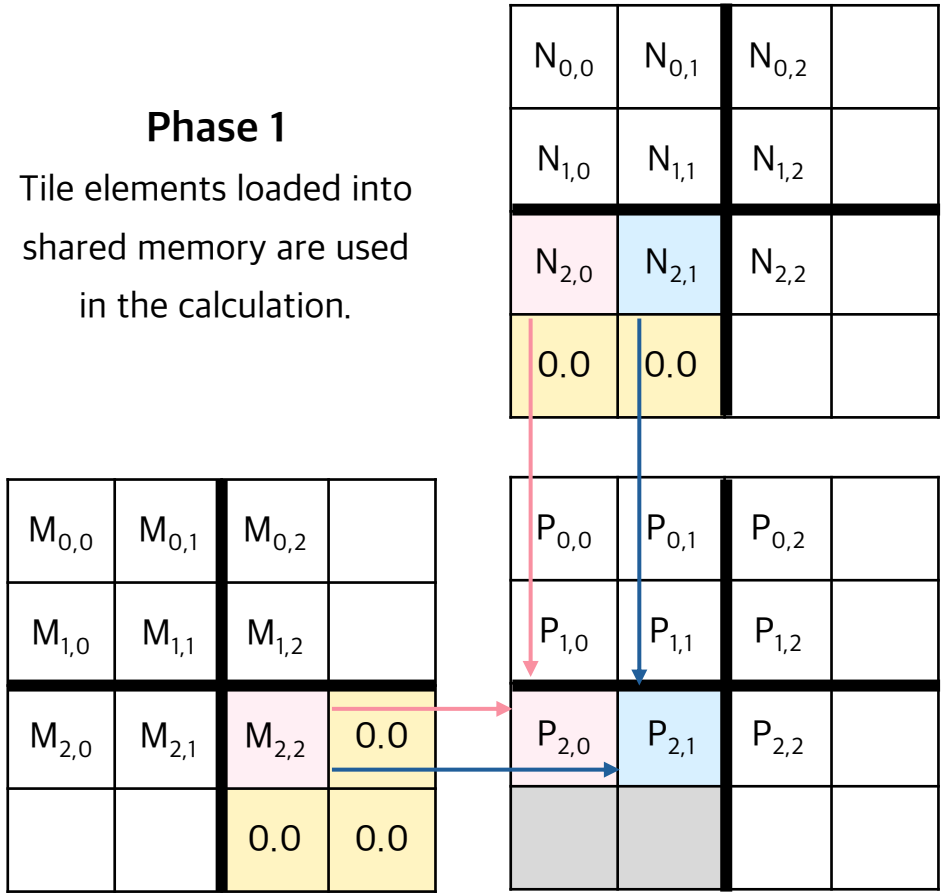
Phase 1

Collaboratively load a tile into shared memory.



Phase 1

Tile elements loaded into shared memory are used in the calculation.



5.7 Summary

- Experimental setup

1) Ubuntu Version: Ubuntu 22.04.2 LTS

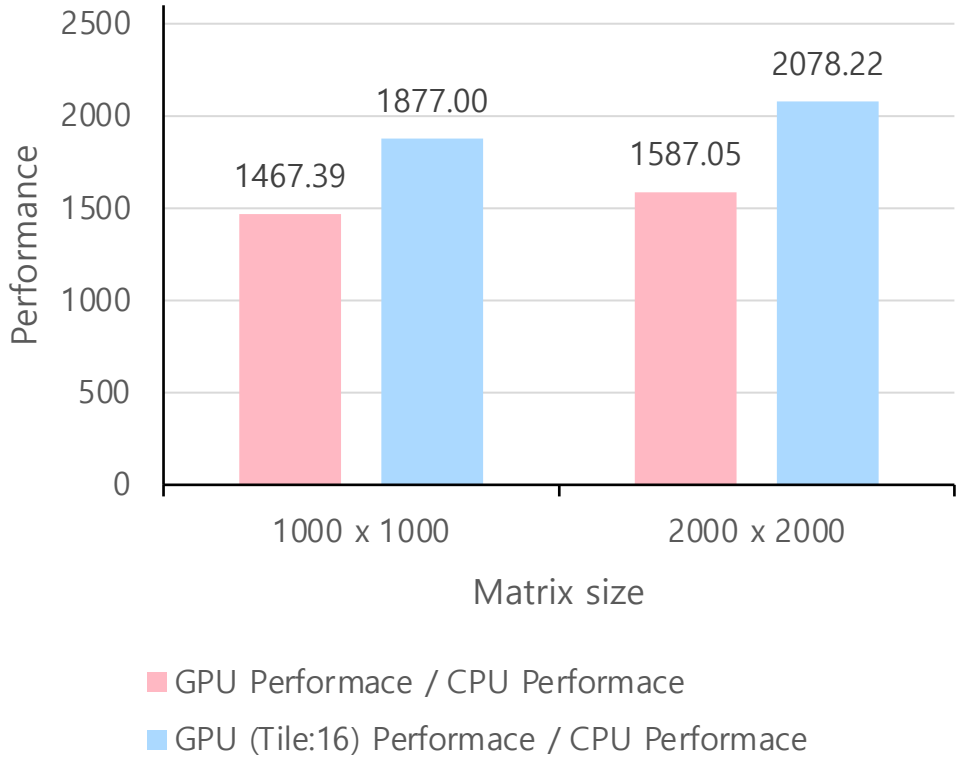
2) CUDA Version: 12.1

3) NVIDIA GeForce RTX 3080
   - 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: (2147483647, 65535, 65535)
   - Shared Mem per SM: 49152 bytes
   - Registers per SM: 65536 bytes
   - Total global Mem: 10495655936 bytes
   - Total constant Mem: 65536 bytes

5.7 Summary

- Results

	CPU Execution Time	GPU Execution Time	GPU (Tile: 16) Execution Time
1000 x 1000	1.564707 (s)	1.066321 (ms)	0.833624 (ms)
2000 x 2000	13.134973 (s)	8.276321 (ms)	6.320292 (ms)



```
jiyeong@IP-CAL-SERVER08:~/pmpp/ch05$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 22.04.2 LTS
Release:        22.04
Codename:       jammy
```

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 530.41.03              Driver Version: 530.41.03    CUDA Version: 12.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name                  Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf            Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |                  MIG M. |
|===============================+======================+======================|
|   0  NVIDIA GeForce RTX 3080         Off| 00000000:01:00.0 Off |                  N/A |
|  0%   50C    P8               36W / 370W|      1MiB / 10240MiB |      0%      Default |
|                                         |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```