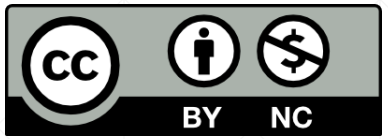
us4us®

IPPT PAN

Introduction to Parallel Programming with NVIDIA CUDA

Performance

License / Attribution



- Materials for the short-course **„Digital Signal Processing with GPUs — Introduction to Parallel Programming”** are licensed by us4us Ltd. the IPPT PAN under the Creative Commons Attribution-NonCommercial 4.0 International License.

- Some slides and examples are borrowed from the course **„The GPU Teaching Kit”** that is licensed by NVIDIA and the University of Illinois under the Creative Commons Attribution-NonCommercial 4.0 International License.
  - All the borrowed slides are marked with NVIDIA ILLINOIS

GPU Memory and Data Locality

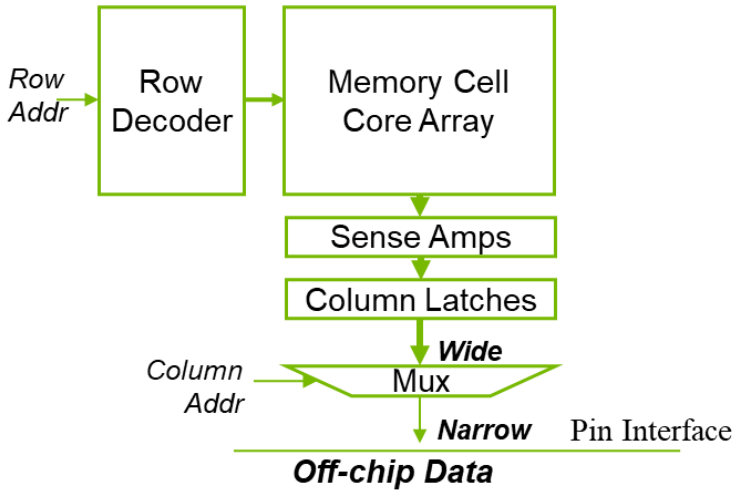
MEMORY is (almost) Everything!

- To learn to effectively use the CUDA memory types in a parallel program
  - Memory hierarchy
  - Importance of memory access efficiency
  - Registers, shared memory, global memory
  - Scope and lifetime

DRAM – Dynamic RAM

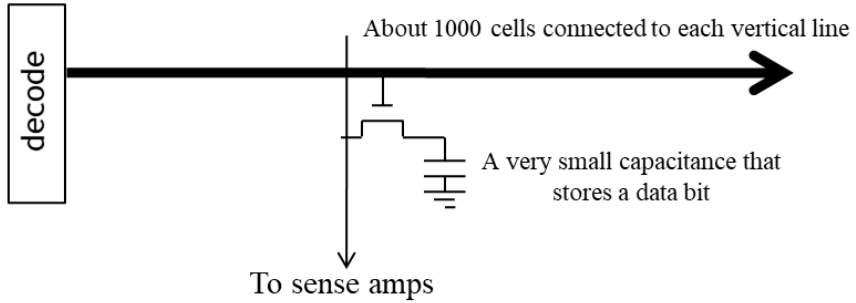
DRAM Core Array Organization

- Each DRAM core array has about 16M bits
- Each bit is stored in a tiny capacitor made of one transistor



DRAM Core Arrays are Slow

- Reading from a cell in the core array is a very slow process
  - DDR: Core speed = ½ interface speed
  - DDR2/GDDR3: Core speed = ¼ interface speed
  - DDR3/GDDR4: Core speed = ⅛ interface speed
  - … likely to be worse in the future

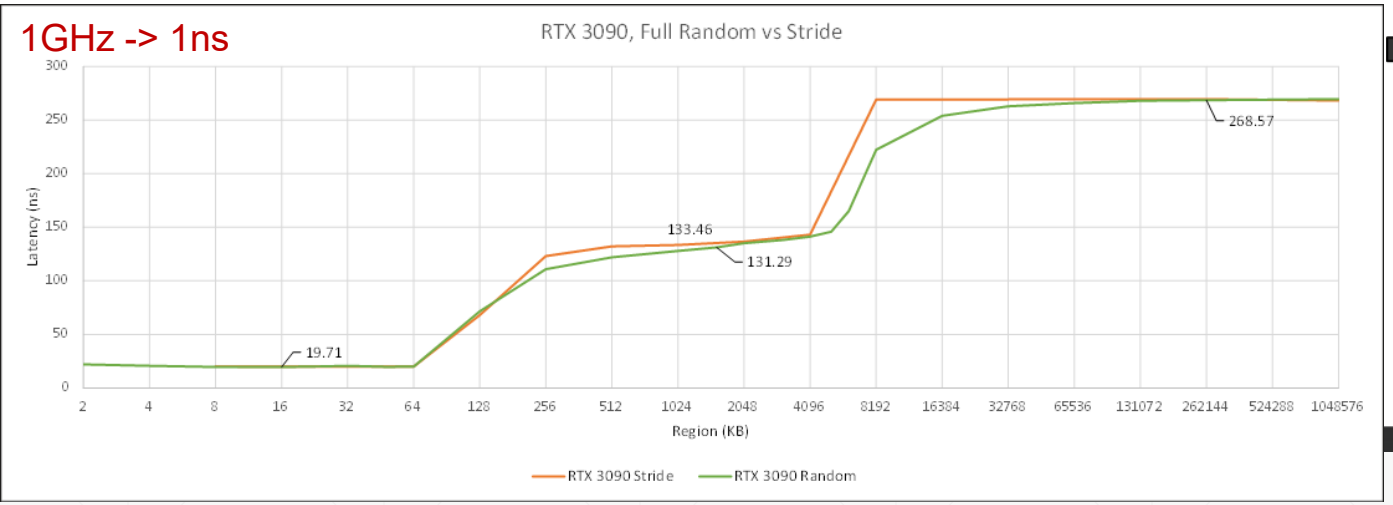


DRAM Bursting

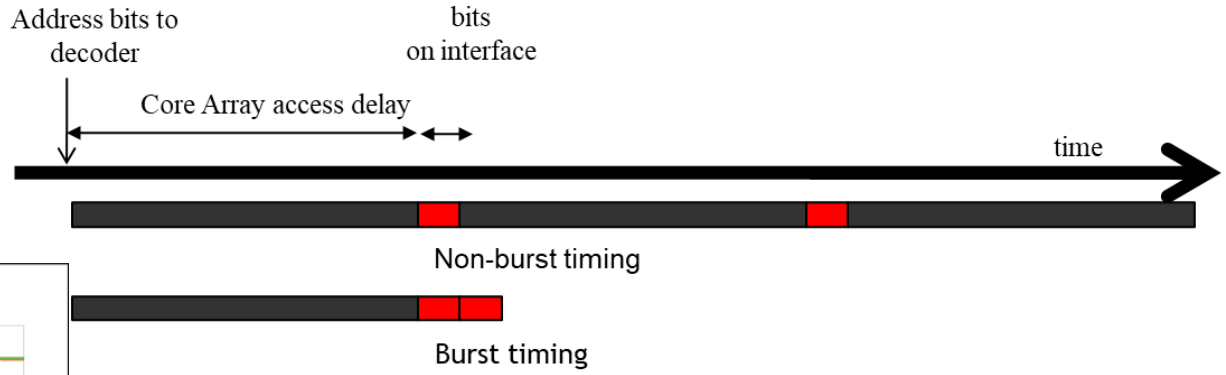
GPU off-chip memory subsystem

- NVIDIA RTX6000 GPU:
  - Peak global memory bandwidth = 672GB/s
- Global memory (GDDR6) interface @ 7GHz
  - 14 Gbps pin speed
  - For GDDR6 32-bit interface, we can sustain only about 56 GB/s
  - We need a lot more bandwidth (672 GB/s) – thus 12 memory channels

1GHz -> 1ns



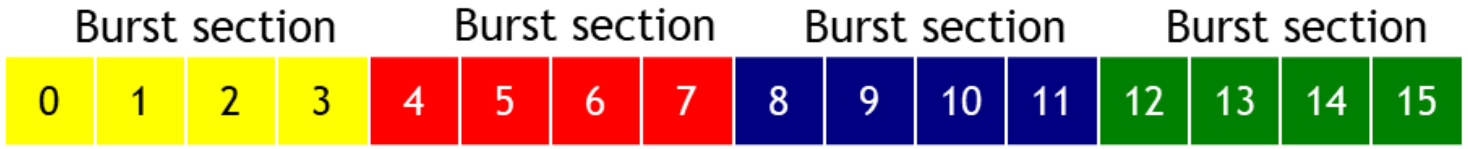
DRAM Bursting Timing Example



Modern DRAM systems are designed to always be accessed in burst mode. Burst bytes are transferred to the processor but discarded when accesses are not to sequential locations.

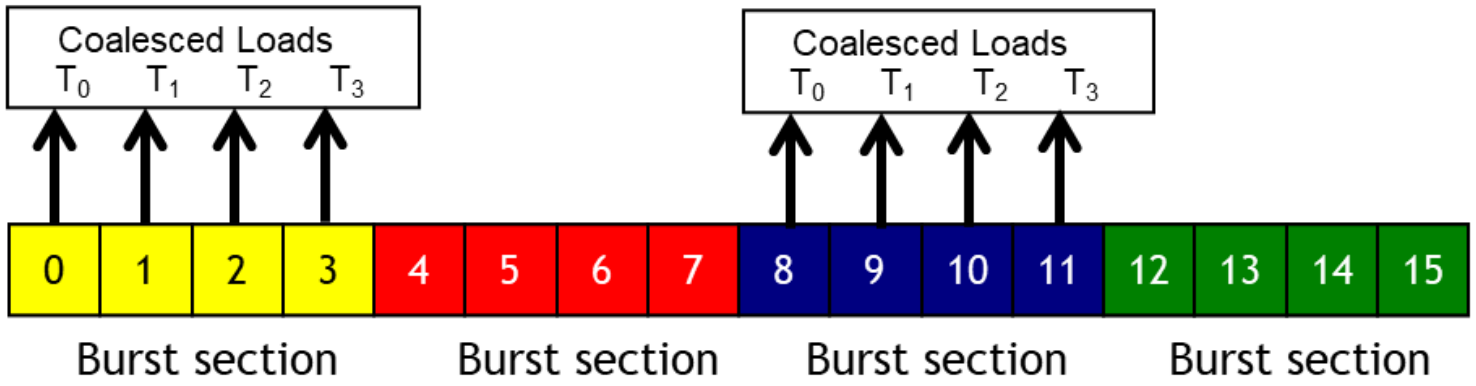
Source: https://chipsandcheese.com/2021/05/13/gpu-memory-latencys-impact-and-updated-test/

DRAM Burst – A System View



- Each address space is partitioned into burst sections
  - Whenever a location is accessed, all other locations in the same section are also delivered to the processor
- Basic example: a 16-byte address space, 4-byte burst sections
  - In practice, we have at least 4GB address space, burst section sizes of 128-bytes or more

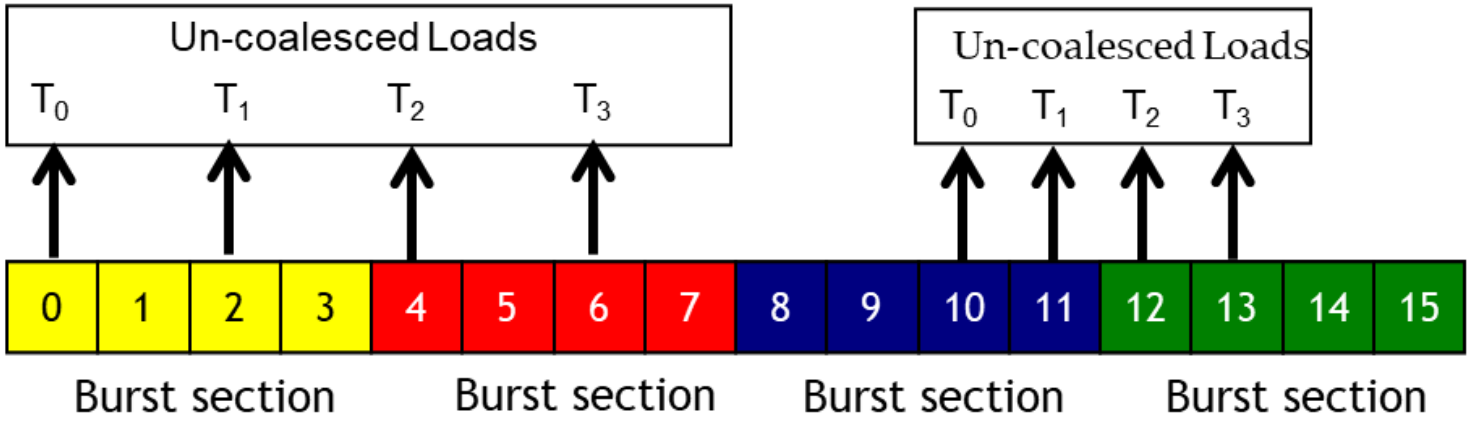
Memory Coalescing



– When all threads of a warp execute a load instruction, if all accessed locations fall into the same burst section, only one DRAM request will be made and the access is fully coalesced.

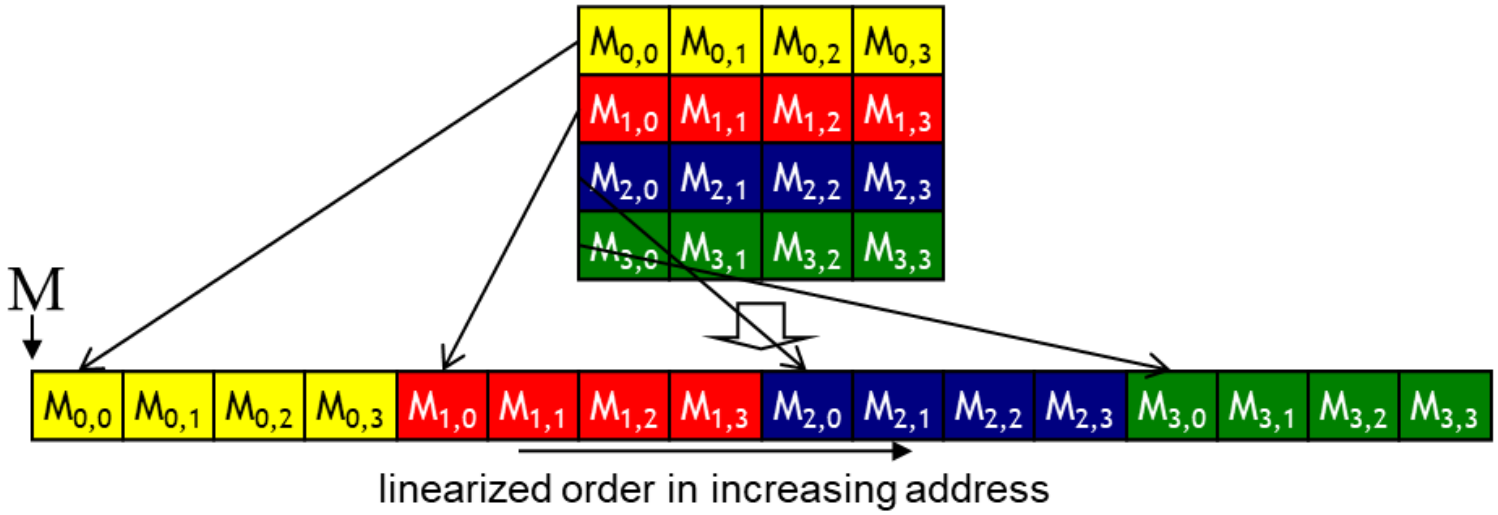
Un-coalesced Accesses

Accesses in a warp are to consecutive locations if the index in an array access is in the form of
A[(expression with terms independent of threadIdx.x) + threadIdx.x];

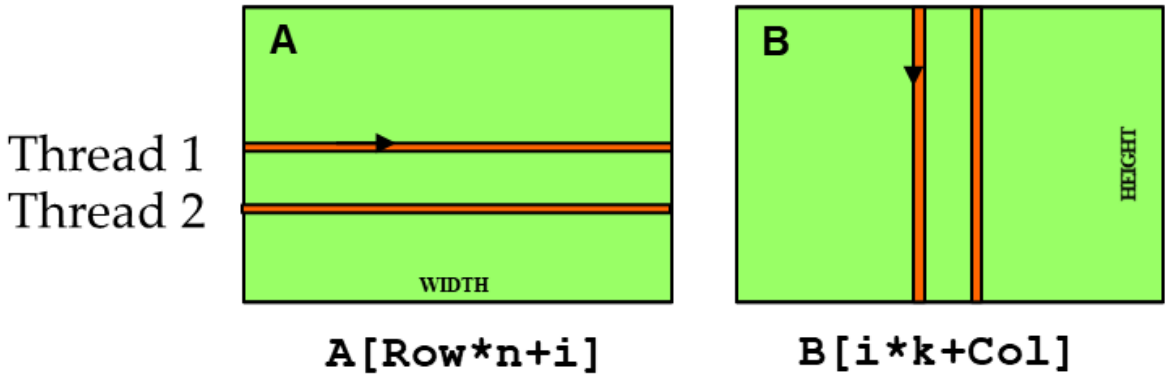


- When the accessed locations spread across burst section boundaries:
  - Coalescing fails
  - Multiple DRAM requests are made
  - The access is not fully coalesced.
- Some of the bytes accessed and transferred are not used by the threads

A 2D C Array in Linear Memory Space



Two Access Patterns of Basic Matrix Multiplication

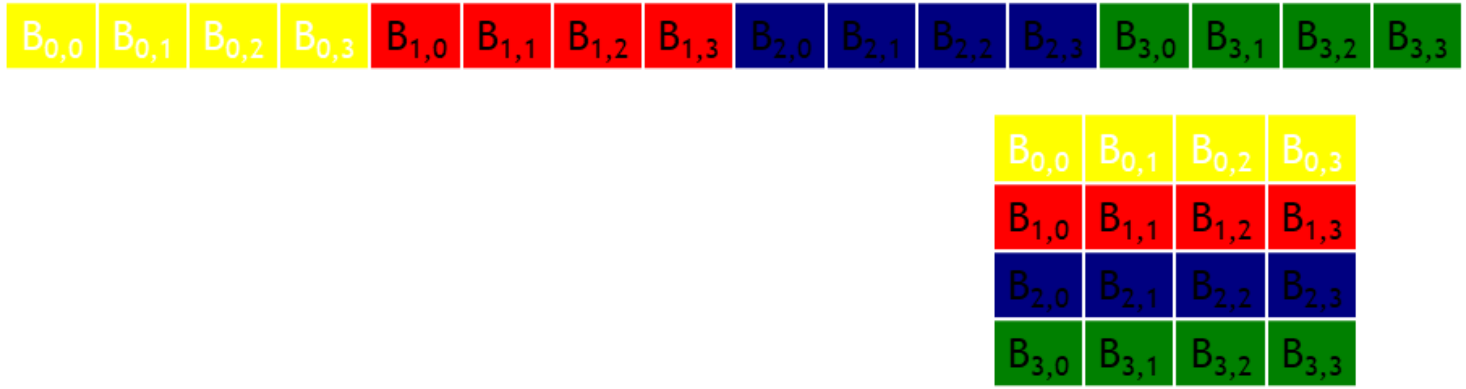


i is the loop counter in the inner product loop of the kernel code

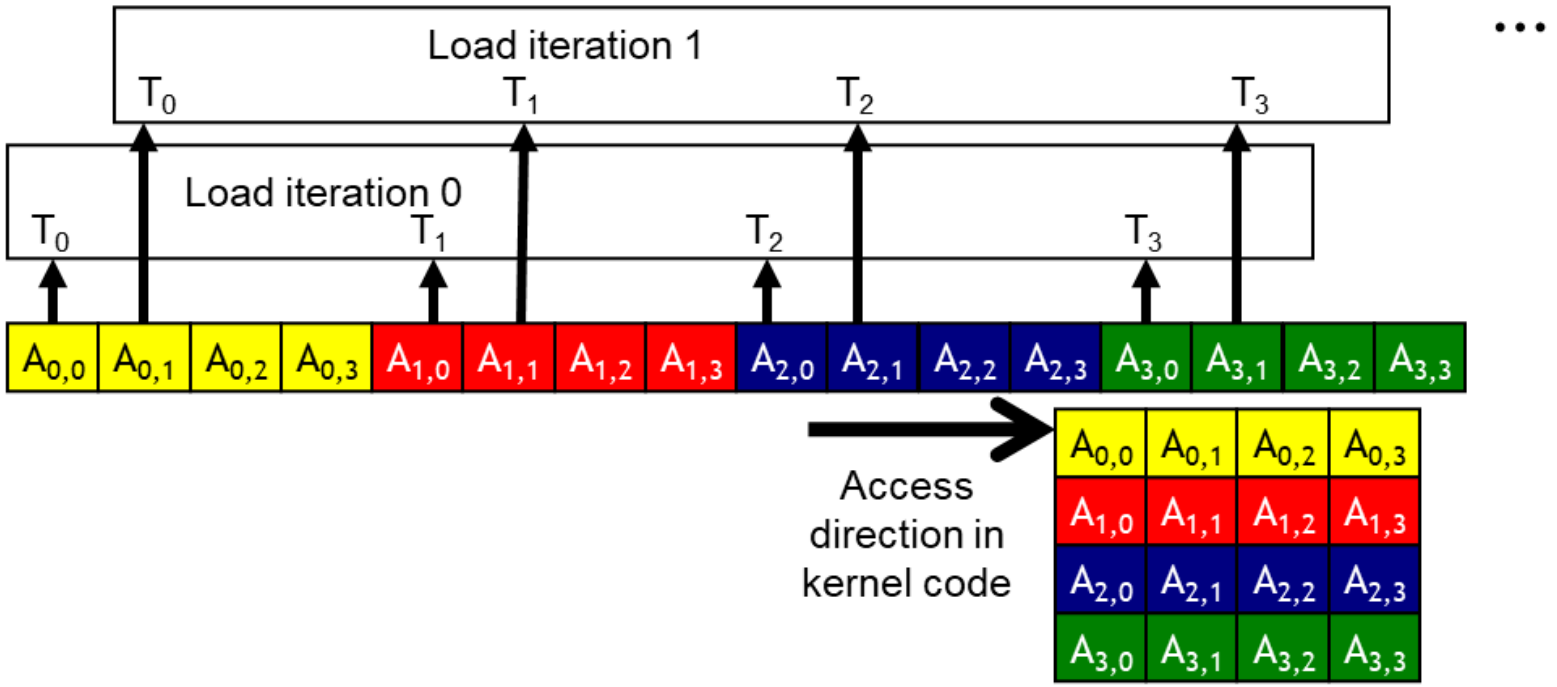
A is m × n, B is n × k

Col = blockIdx.x*blockDim.x + threadIdx.x

B accesses are coalesced



A Accesses are Not Coalesced

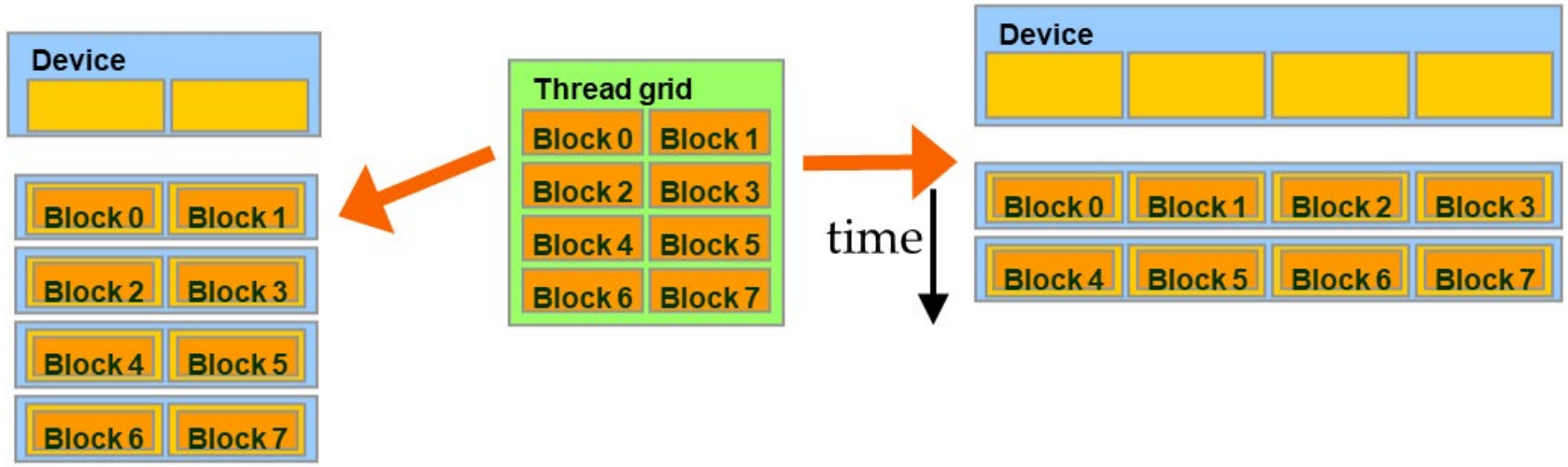


How about performance on a GPU

- All threads access global memory for their input matrix elements
  - One memory accesses (4 bytes) per floating-point addition
  - 4B/s of memory bandwidth/FLOPS
- Assume a GPU with
  - Peak floating-point rate 1,600 GFLOPS with 600 GB/s DRAM bandwidth
  - 4*1,600 = 6,400 GB/s required to achieve peak FLOPS rating
  - The 600 GB/s memory bandwidth limits the execution at 150 GFLOPS
- This limits the execution rate to 9.3% (150/1600) of the peak floating-point execution rate of the device!
- Need to drastically cut down memory accesses to get close to the1,600 GFLOPS

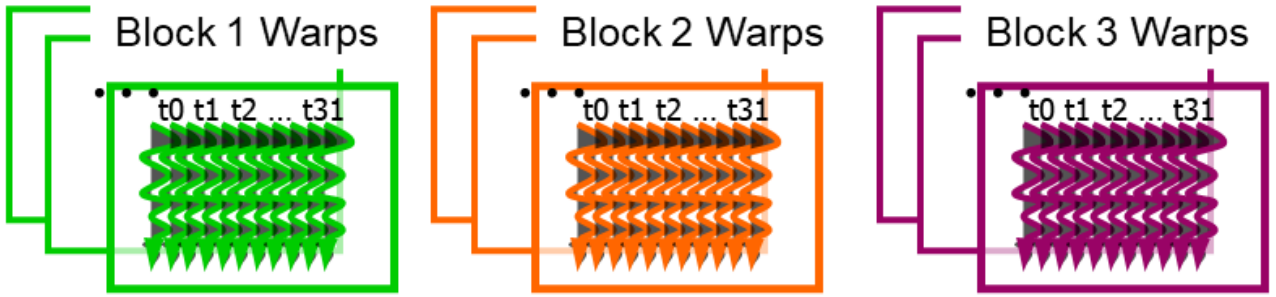
Thread scheduling

Transparent Scalability



- Each block can execute in any order relative to others.
- Hardware is free to assign blocks to any processor at any time
  - A kernel scales to any number of parallel processors

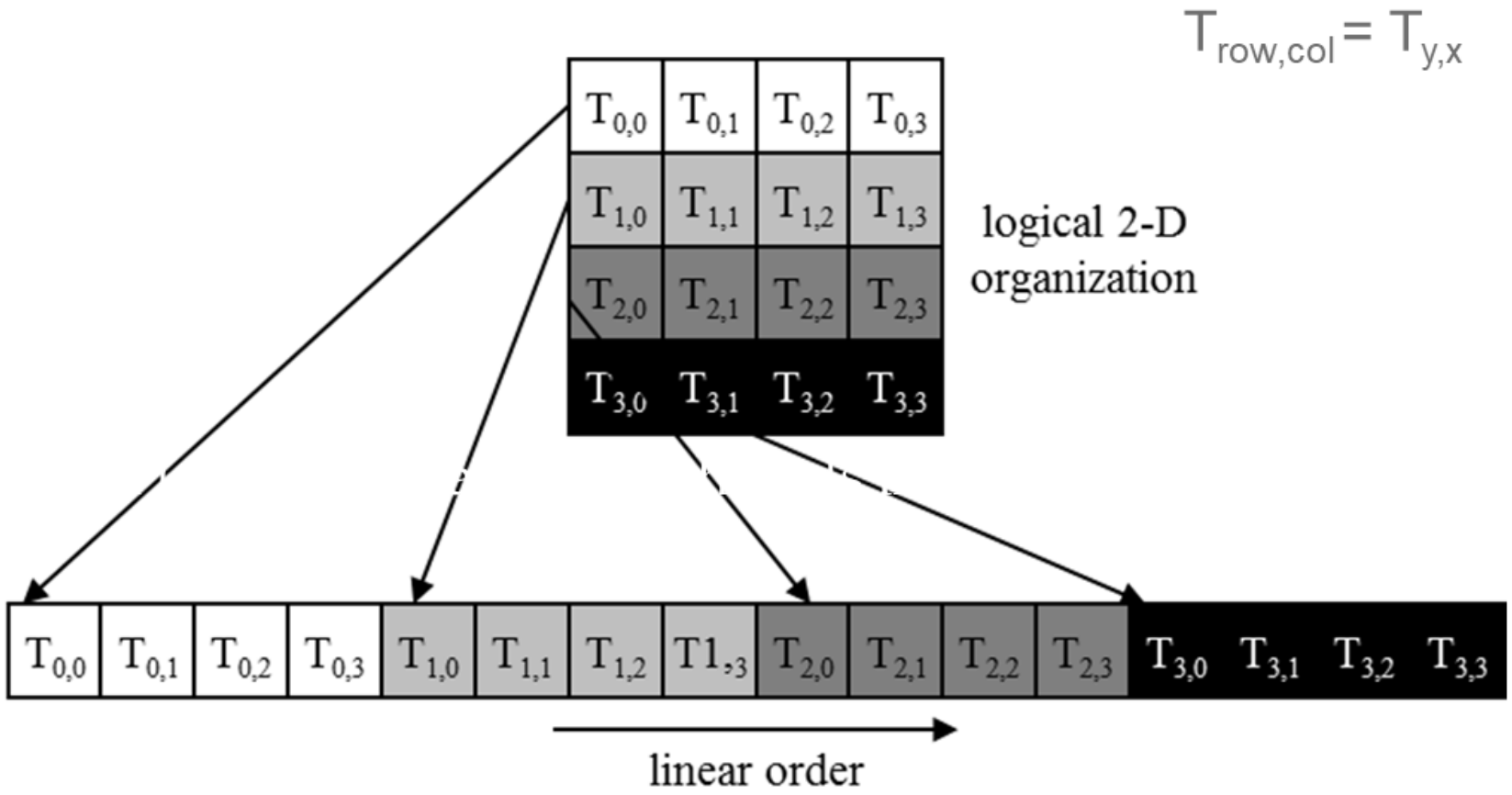
Warps as Scheduling Units



- Each block is divided into 32-thread warps
  - An implementation technique, not part of the CUDA programming model
  - Warps are scheduling units in SM
  - Threads in a warp execute in Single Instruction Multiple Data (SIMD) manner
  - The number of threads in a warp may vary in future generations

Warps in Multi-dimensional Thread Blocks

- The thread blocks are first linearized into 1D in row major order
  - In x-dimension first, y-dimension next, and z-dimension last



Blocks are partitioned after linearization

- Linearized thread blocks are partitioned
  - Thread indices within a warp are consecutive and increasing
  - Warp 0 starts with Thread 0
- Partitioning scheme is consistent across devices
  - Thus you can use this knowledge in control flow
  - However, the exact size of warps may change from generation to generation
- DO NOT rely on any ordering within or between warps
  - If there are any dependencies between threads, you must __syncthreads() to get correct results (more later).

SIMD Execution Among Threads in a Warp

- All threads in a warp must execute the same instruction at any point in time

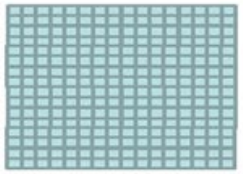
- This works efficiently if all threads follow the same control flow path
  - All if-then-else statements make the same decision
  - All loops iterate the same number of times

Control Divergence

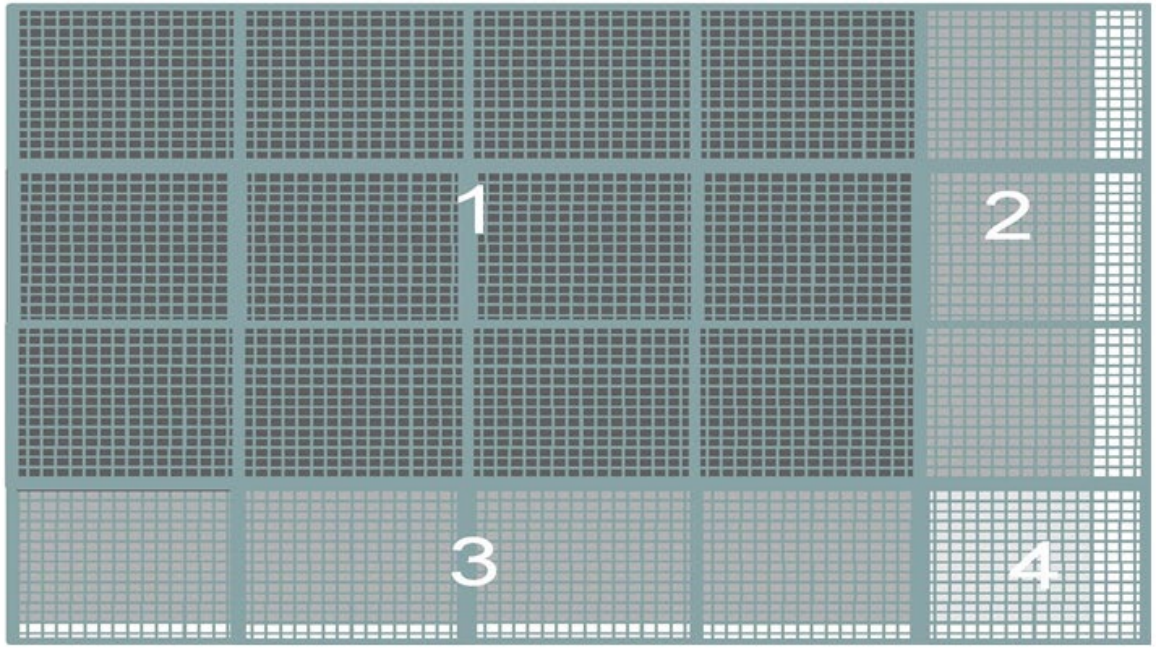
- Control divergence occurs when threads in a warp take different control flow paths by making different control decisions
  - Some take the then-path and others take the else-path of an if-statement
  - Some threads take different number of loop iterations than others

- The execution of threads taking different paths are serialized in current GPUs
  - The control paths taken by the threads in a warp are traversed one at a time until there is no more.
  - During the execution of each path, all threads taking that path will be executed in parallel
  - The number of different paths can be large when considering nested control flow statements

Covering a 62×76 Picture with 16×16 Blocks



16×16 block



Not all threads in a Block will follow the same control flow path.