2022 IEEE IUS
International Ultrasonics Symposium
Venice, Italy || October 10-13, 2022

us4us®

IPPT PAN

Ultrasound Signal Processing with GPUs – Introduction to Parallel Programming

GPU Memory & Performance

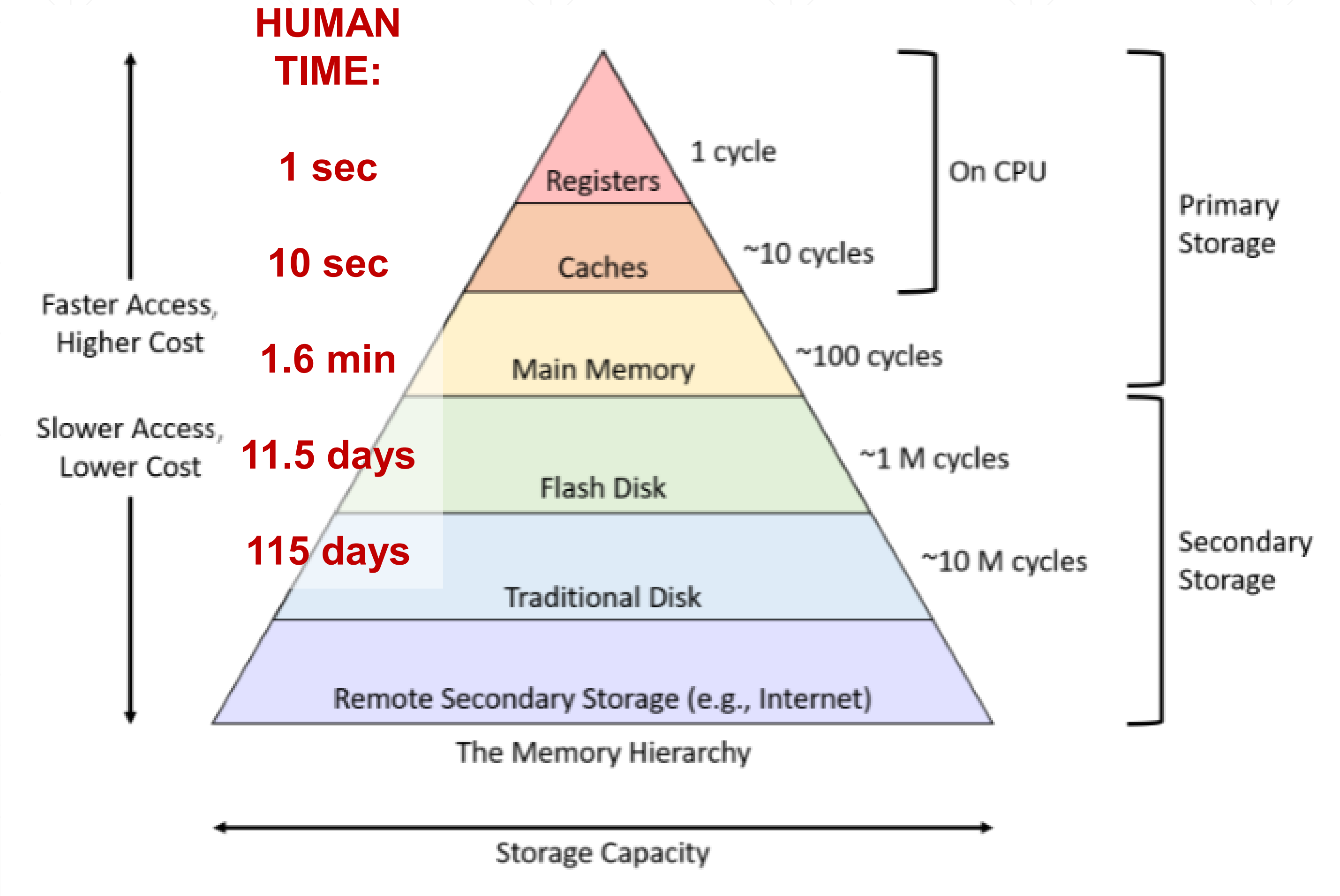
Presenter: Dr Marcin Lewandowski

License / Attribution

- Materials for the short-course **„Ultrasound 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

Computer Memory hierarchy



The Memory Hierarchy

Source: https://diveintosystems.org/antora/diveintosystems/1.0/MemHierarchy/mem_hierarchy.html

GPU Memory Hierarchy

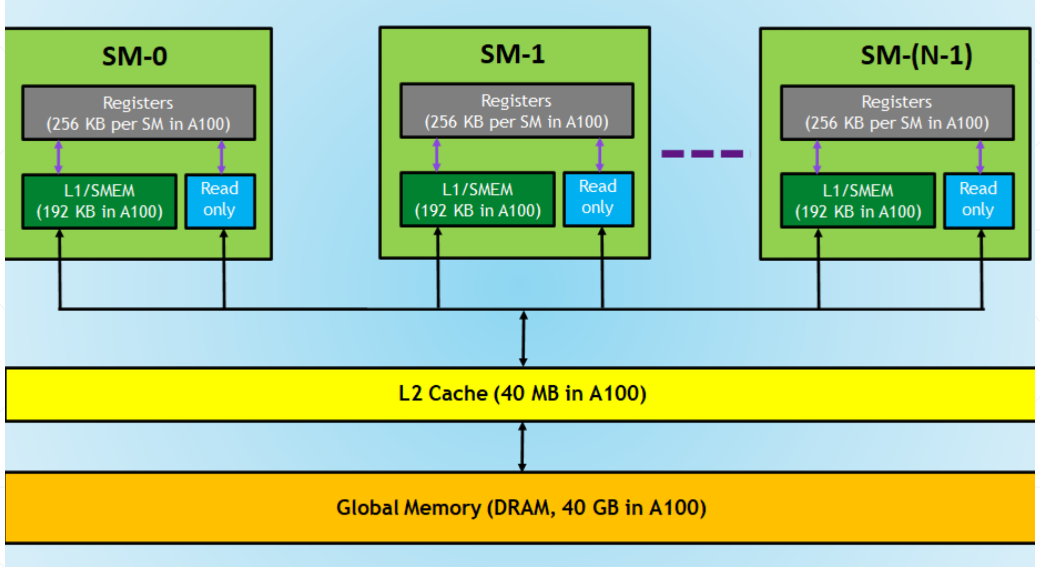
- **Registers –** These are private to each thread, which means that registers assigned to a thread are not visible to other threads. The compiler makes decisions about register utilization.

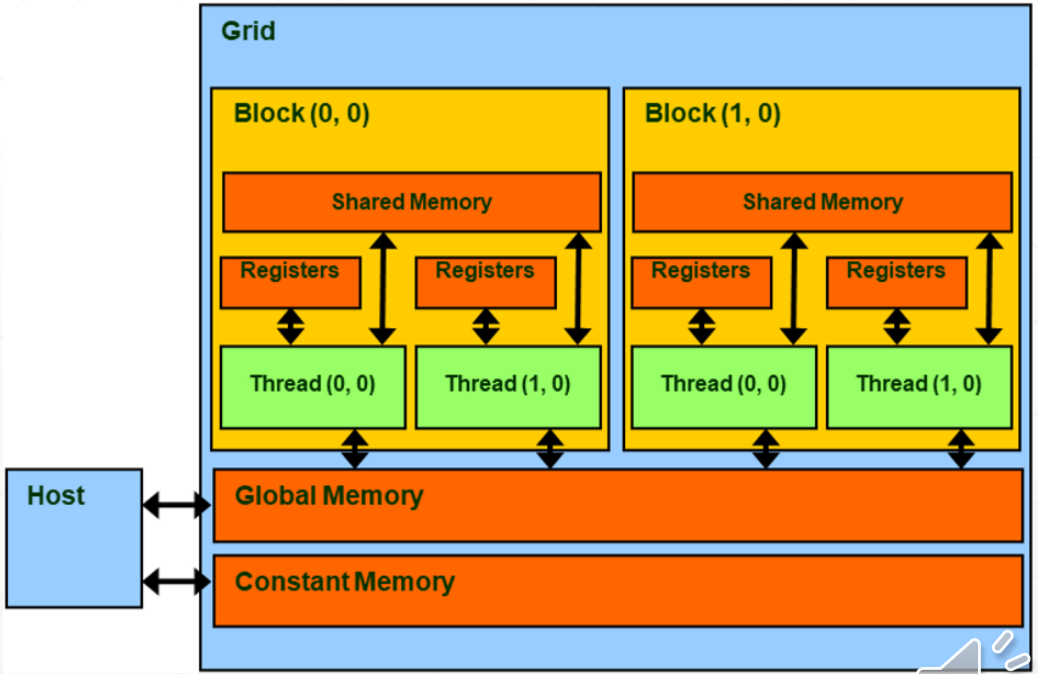
- **L1/Shared memory (SMEM) –** Every SM has a fast, on-chip scratchpad memory that can be used as L1 cache and shared memory. All threads in a CUDA block can share shared memory, and all CUDA blocks running on a given SM can share the physical memory resource provided by the SM.

- **Read-only memory –** Each SM has an instruction cache, constant memory, texture memory and RO cache, which is read-only to kernel code.

- **L2 cache –** The L2 cache is shared across all SMs, so every thread in every CUDA block can access this memory. The NVIDIA A100 GPU has increased the L2 cache size to 40 MB as compared to 6 MB in V100 GPUs.

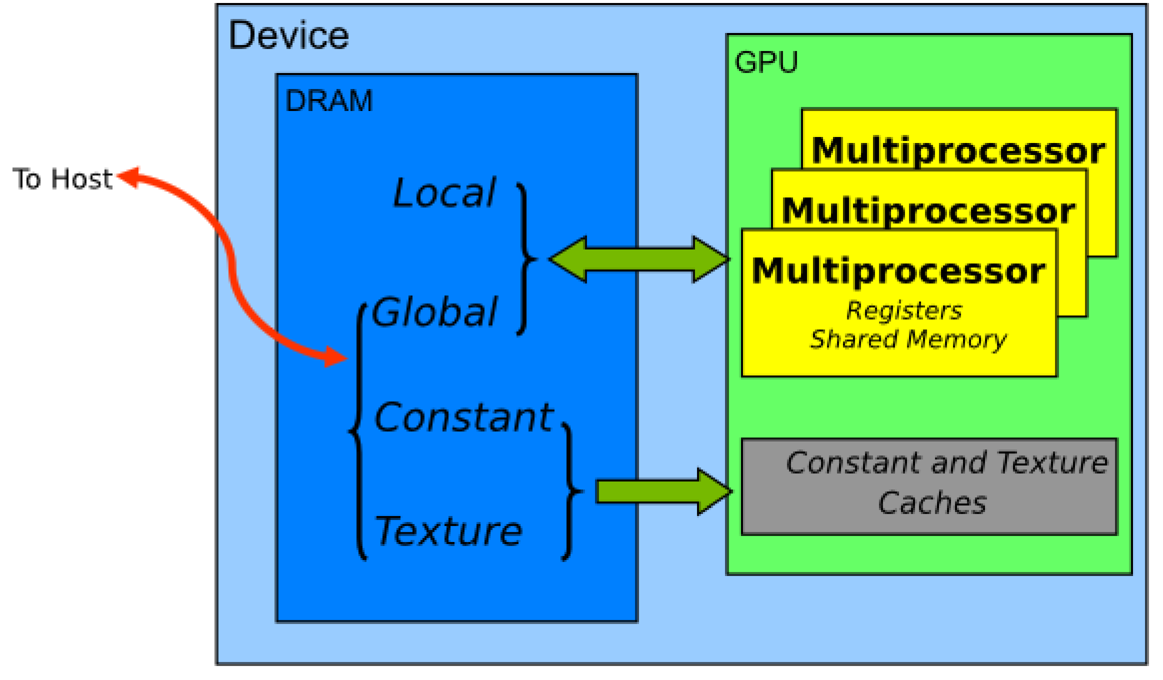
- **Global memory –** This is the framebuffer size of the GPU and DRAM sitting in the GPU.

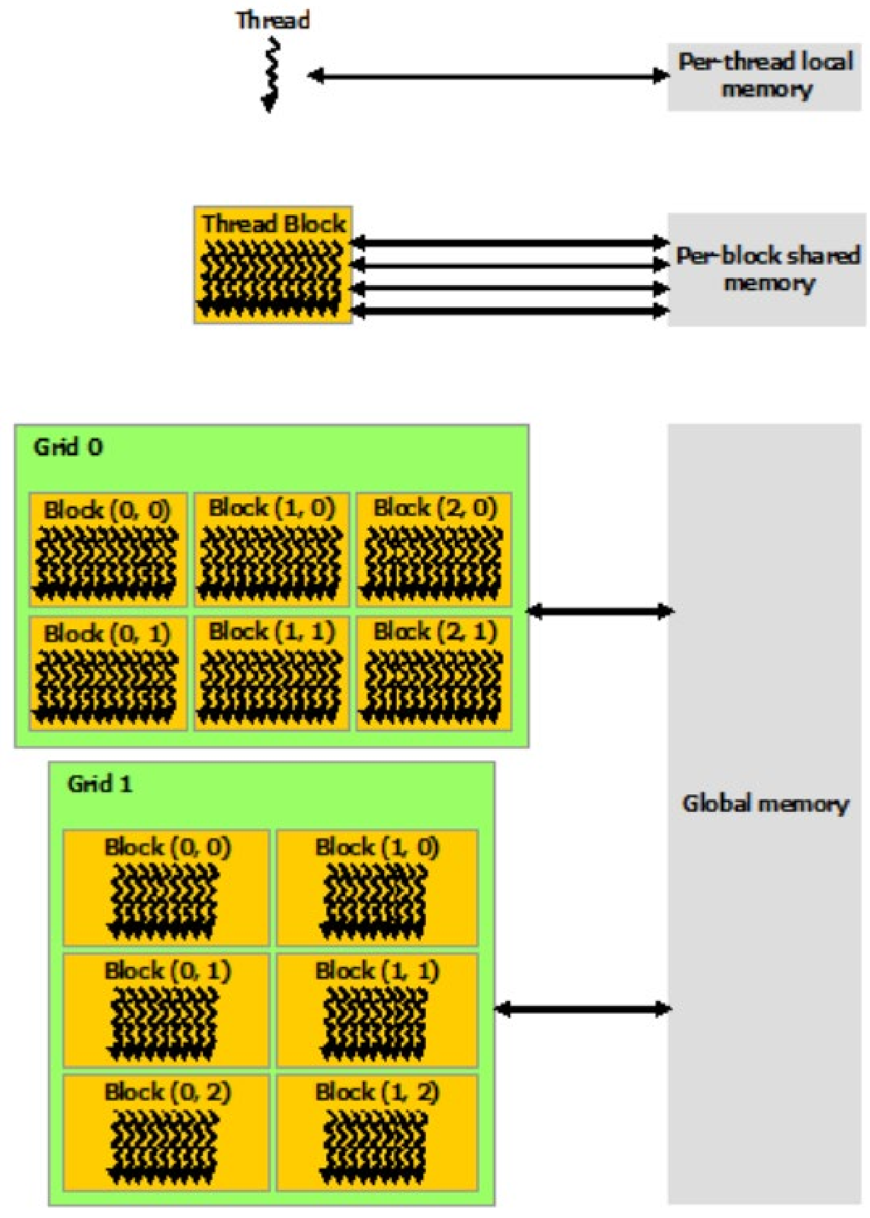




Source: https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

GPU memory access





Source: https://docs.nvidia.com/cuda/

GPU Memory Features by Type

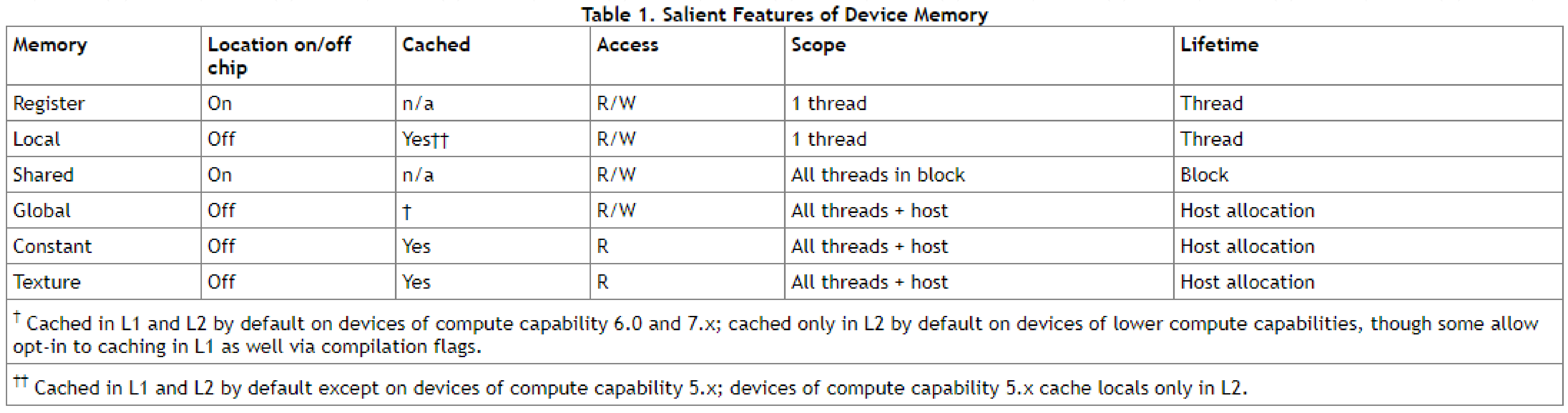


Table 1. Salient Features of Device Memory

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	Yes[††]	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	[†]	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

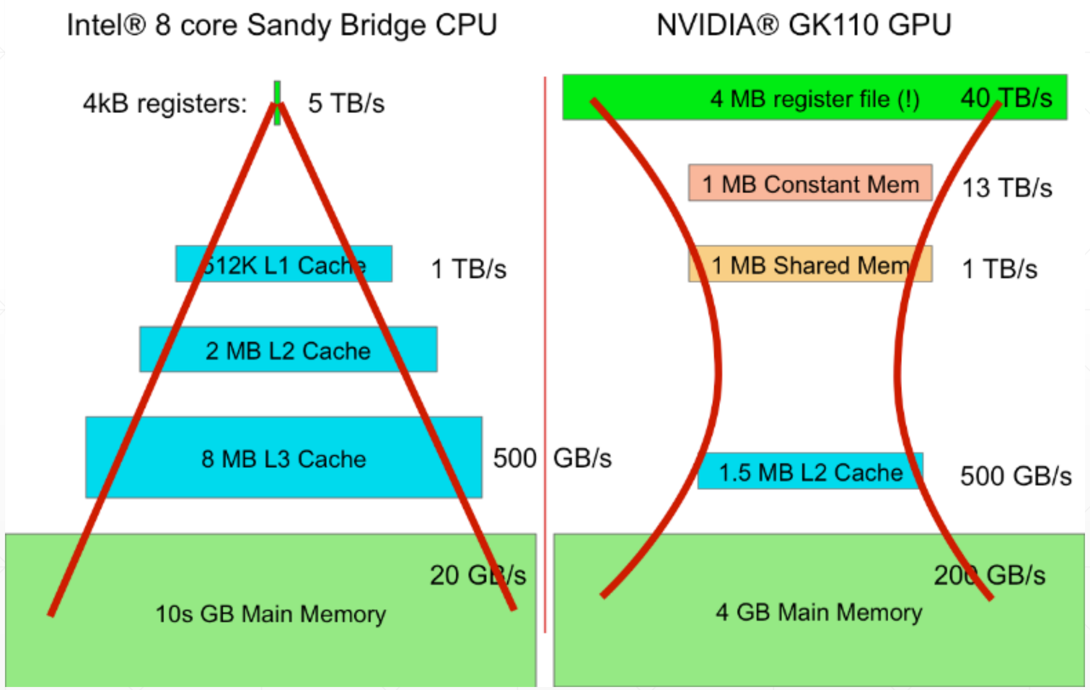
[†] Cached in L1 and L2 by default on devices of compute capability 6.0 and 7.x; cached only in L2 by default on devices of lower compute capabilities, though some allow opt-in to caching in L1 as well via compilation flags.

[††] Cached in L1 and L2 by default except on devices of compute capability 5.x; devices of compute capability 5.x cache locals only in L2.

Source: https://docs.nvidia.com/cuda/

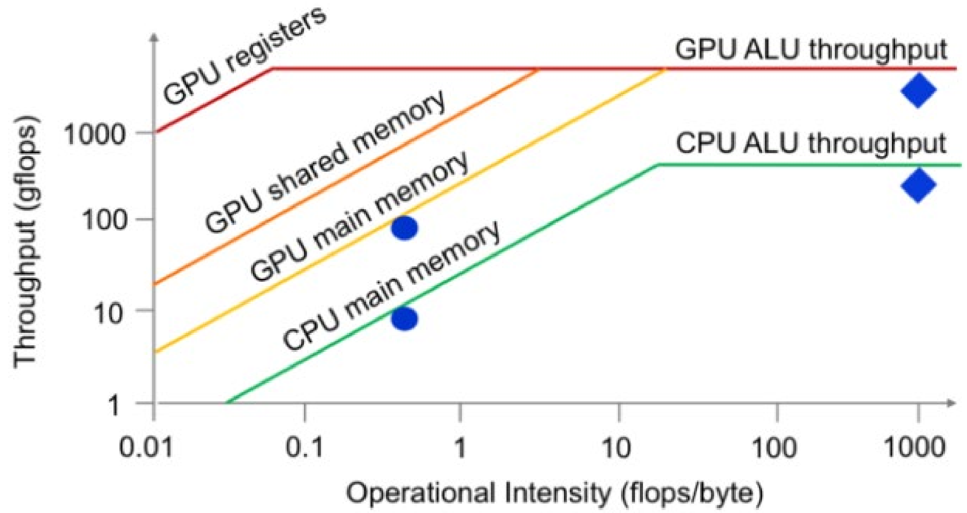
Memory / Arithmetic intensity / Performance

Where is my Memory?



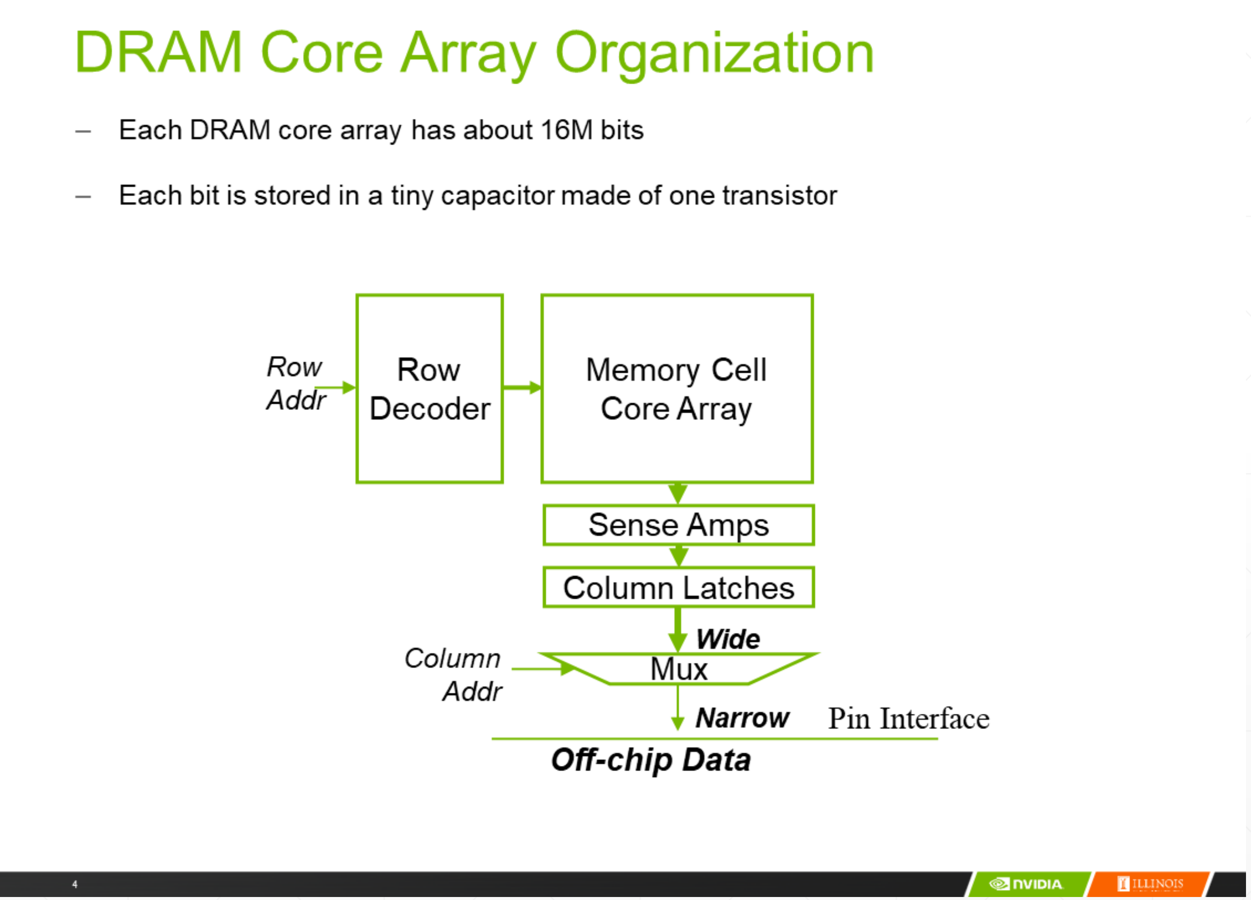
Roofline Design – Matrix kernels

- Dense matrix multiply ◆
- Sparse matrix multiply ●



Source: https://developer.nvidia.com/blog/bidmach-machine-learning-limit-gpus/ | https://en.wikipedia.org/wiki/Roofline_model

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

Row Addr

Row Decoder

Memory Cell Core Array

Sense Amps

Column Latches

Wide

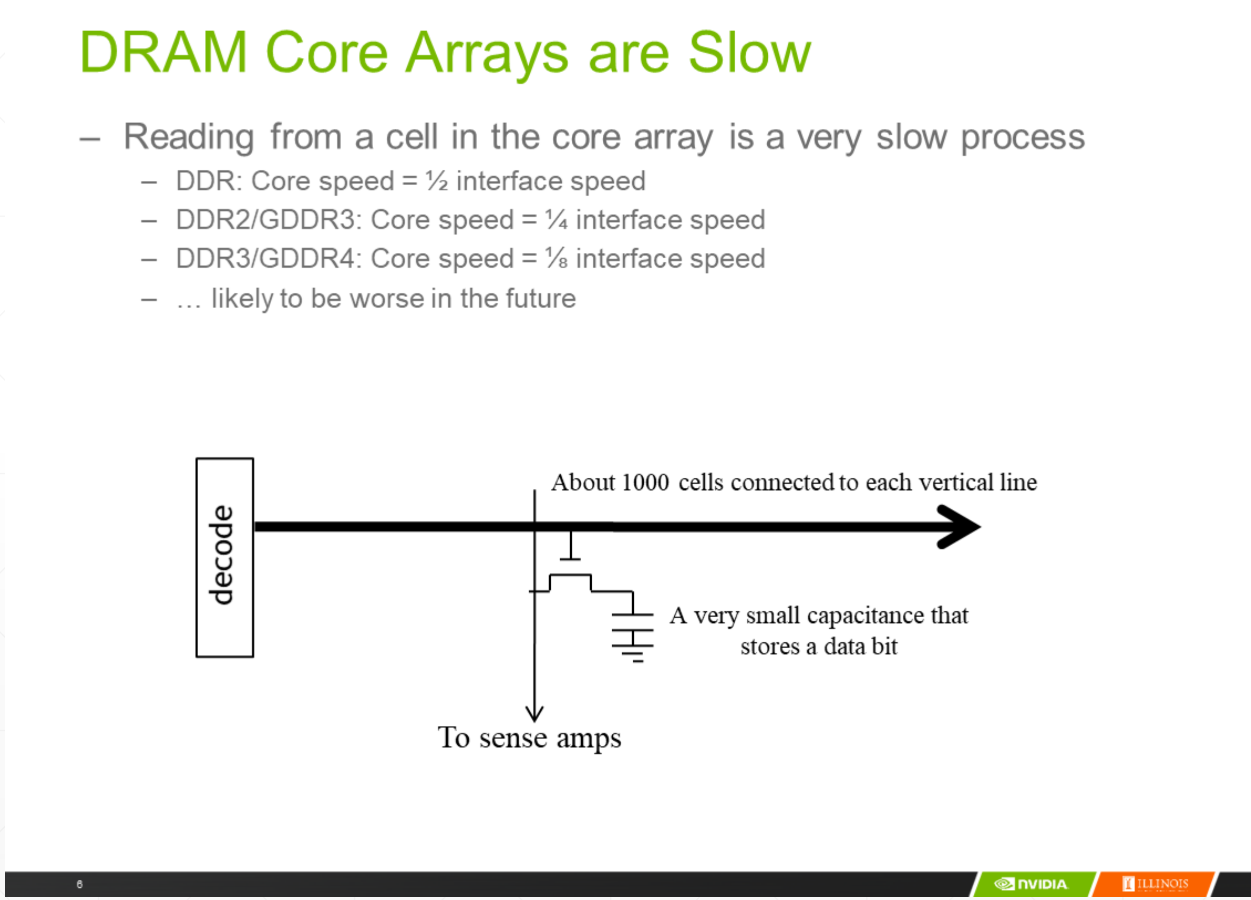
Column Addr

Mux

Narrow

Pin Interface

Off-chip Data



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

decode

About 1000 cells connected to each vertical line

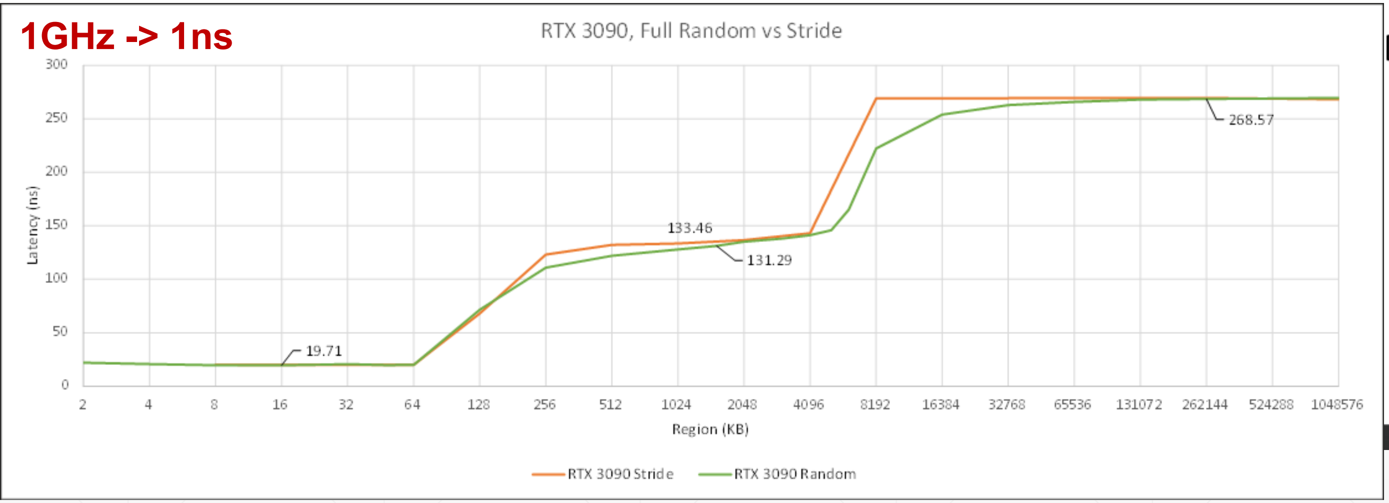
A very small capacitance that stores a data bit

To sense amps

DRAM Bursting

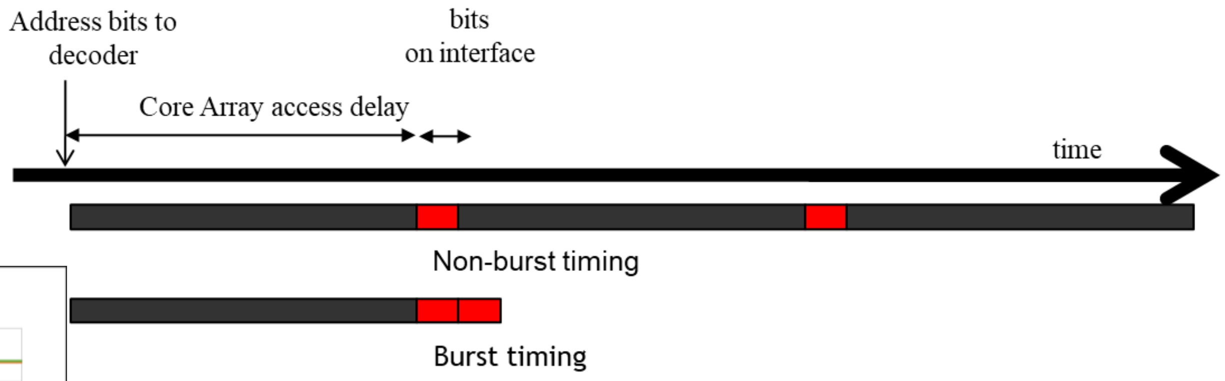
GPU off-chip memory subsystem

- 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


Address bits to decoder

bits on interface

Core Array access delay

time

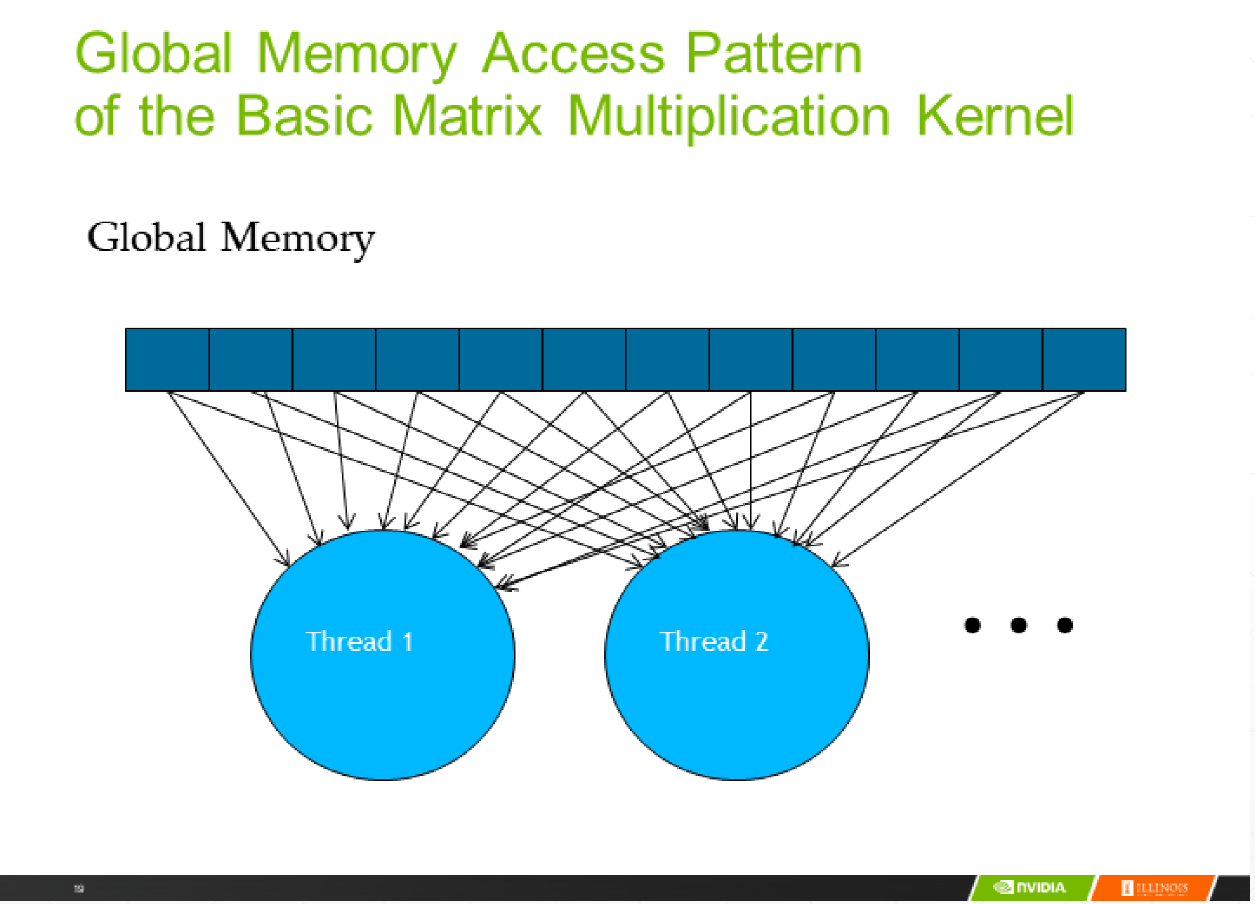
Non-burst timing

Burst timing

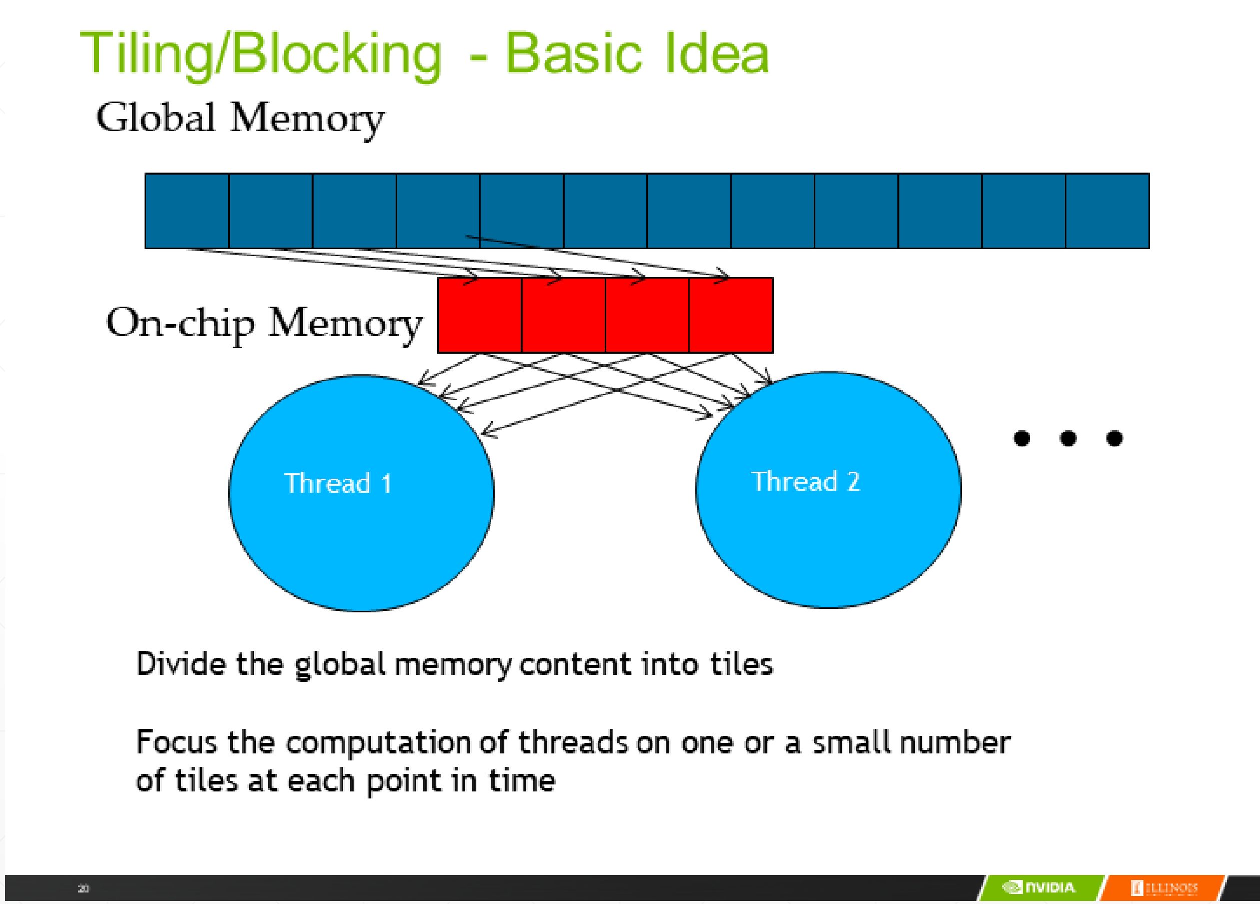
Modern DRAM systems are designed to always be accessed in burst mode. Burst bytes are transferred to the processor but discarded when accesses are not to sequential locations.

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

Tiling/Blocking memory



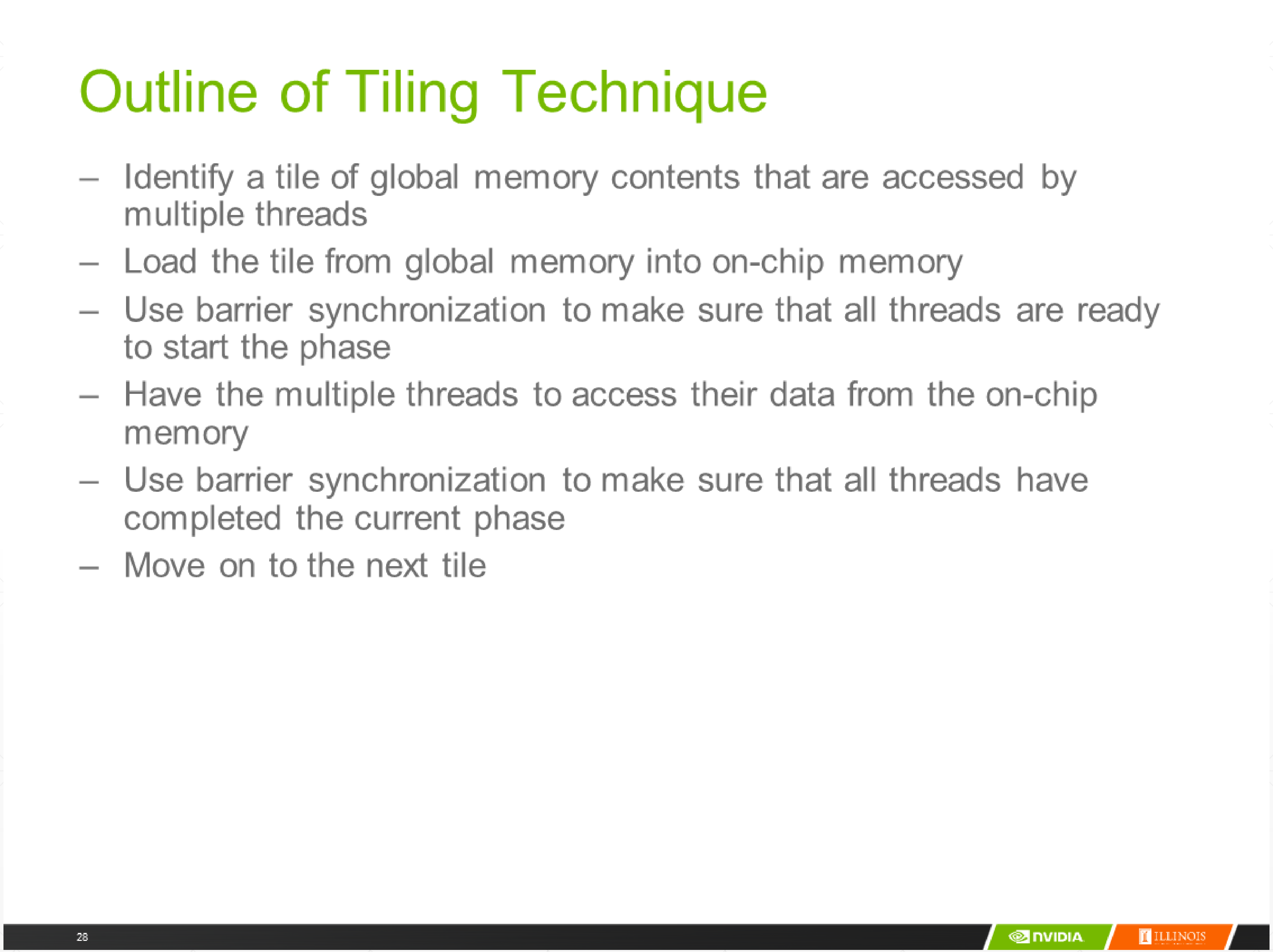
Global Memory Access Pattern of the Basic Matrix Multiplication Kernel



Tiling/Blocking - Basic Idea

Divide the global memory content into tiles

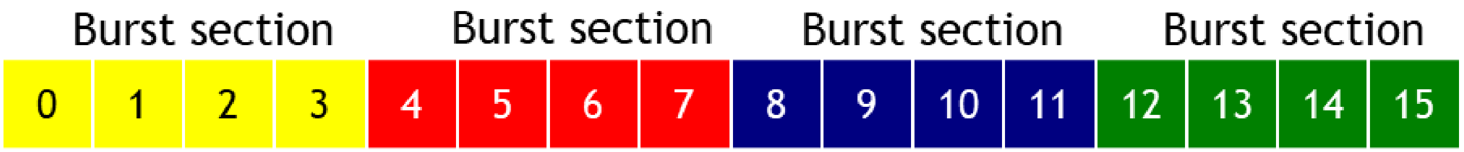
Focus the computation of threads on one or a small number of tiles at each point in time



Outline of Tiling Technique

- Identify a tile of global memory contents that are accessed by multiple threads
- Load the tile from global memory into on-chip memory
- Use barrier synchronization to make sure that all threads are ready to start the phase
- Have the multiple threads to access their data from the on-chip memory
- Use barrier synchronization to make sure that all threads have completed the current phase
- Move on to the next tile

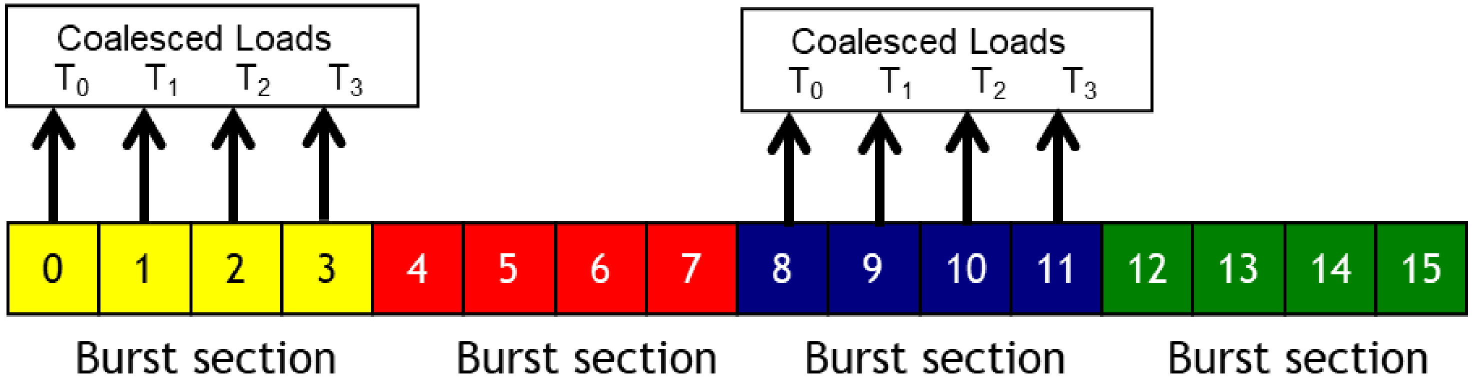
DRAM Burst – A System View



Burst section				Burst section				Burst section				Burst section			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

- 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

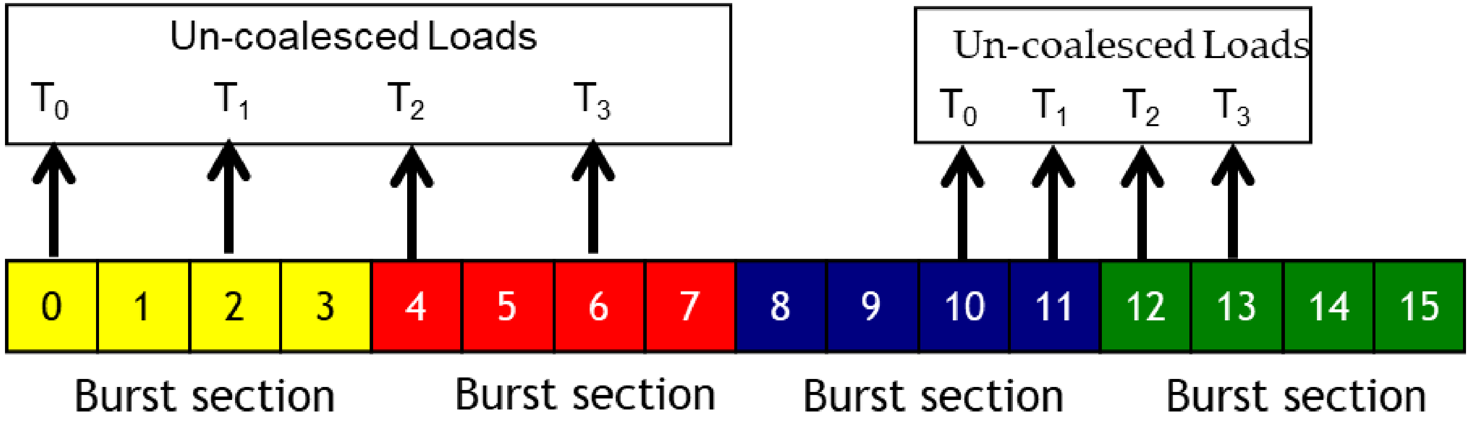
Memory Coalescing



- 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.

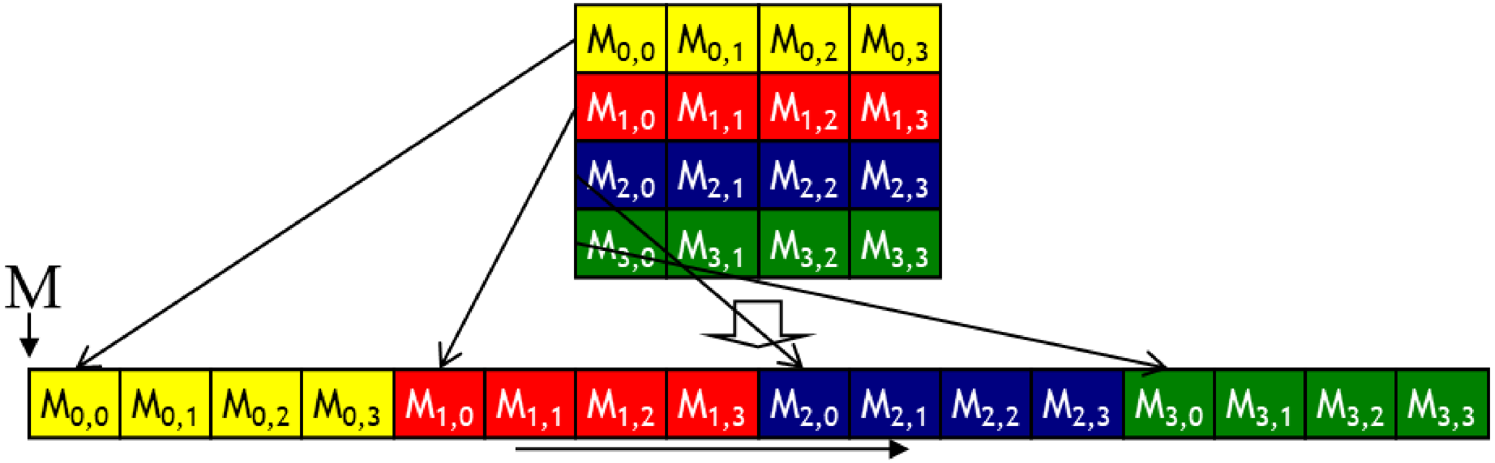
Un-coalesced Accesses

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



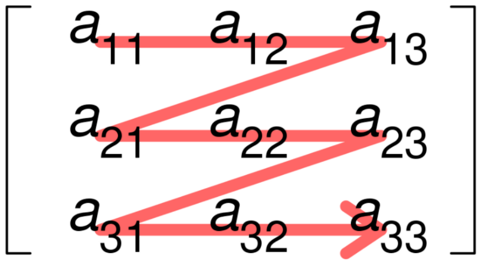
$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

M

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$	$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$	$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$	$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

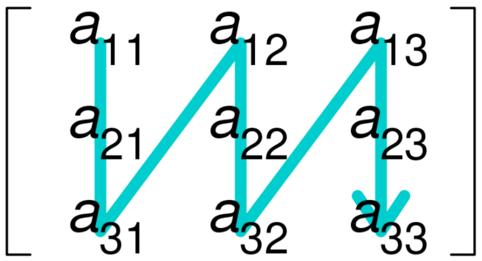
linearized order in increasing address

Row-major order



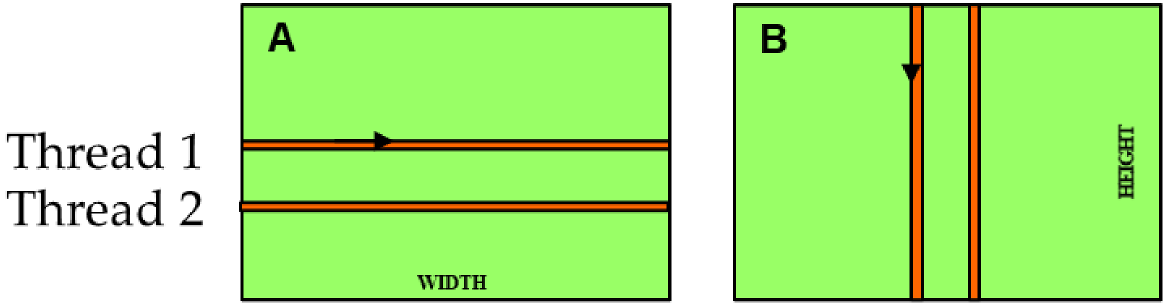
$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Column-major order



$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Two Access Patterns of Basic Matrix Multiplication



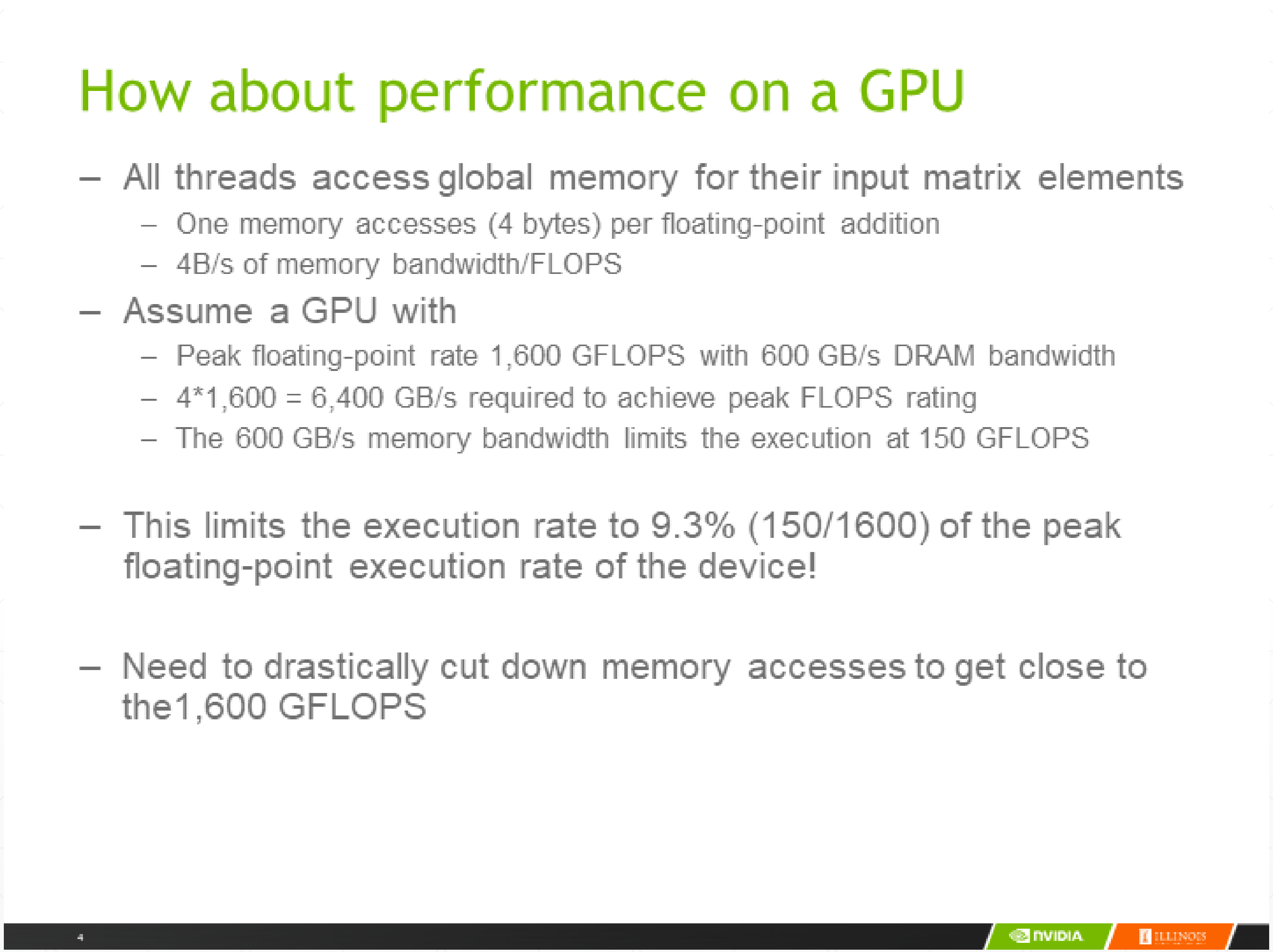
A[Row*n+i]

B[i*k+Col]

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



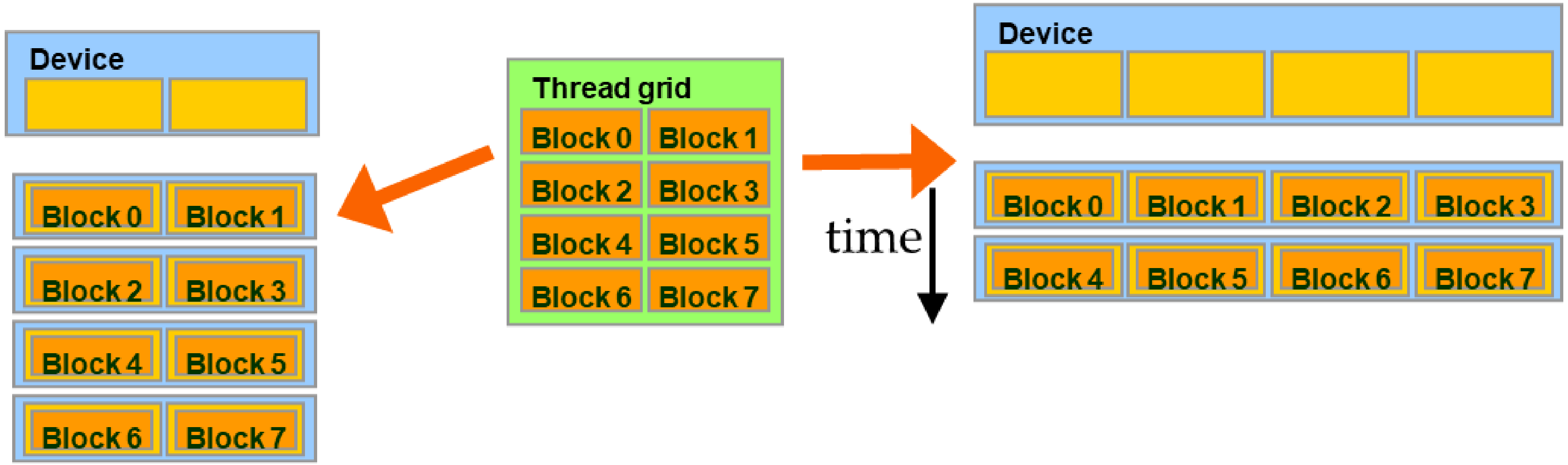
How about performance on a GPU

- 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

LOOK: https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html

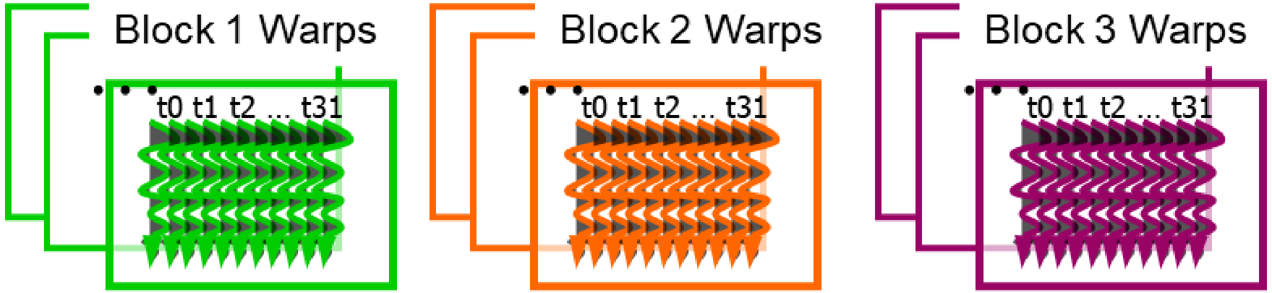
Thread scheduling

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

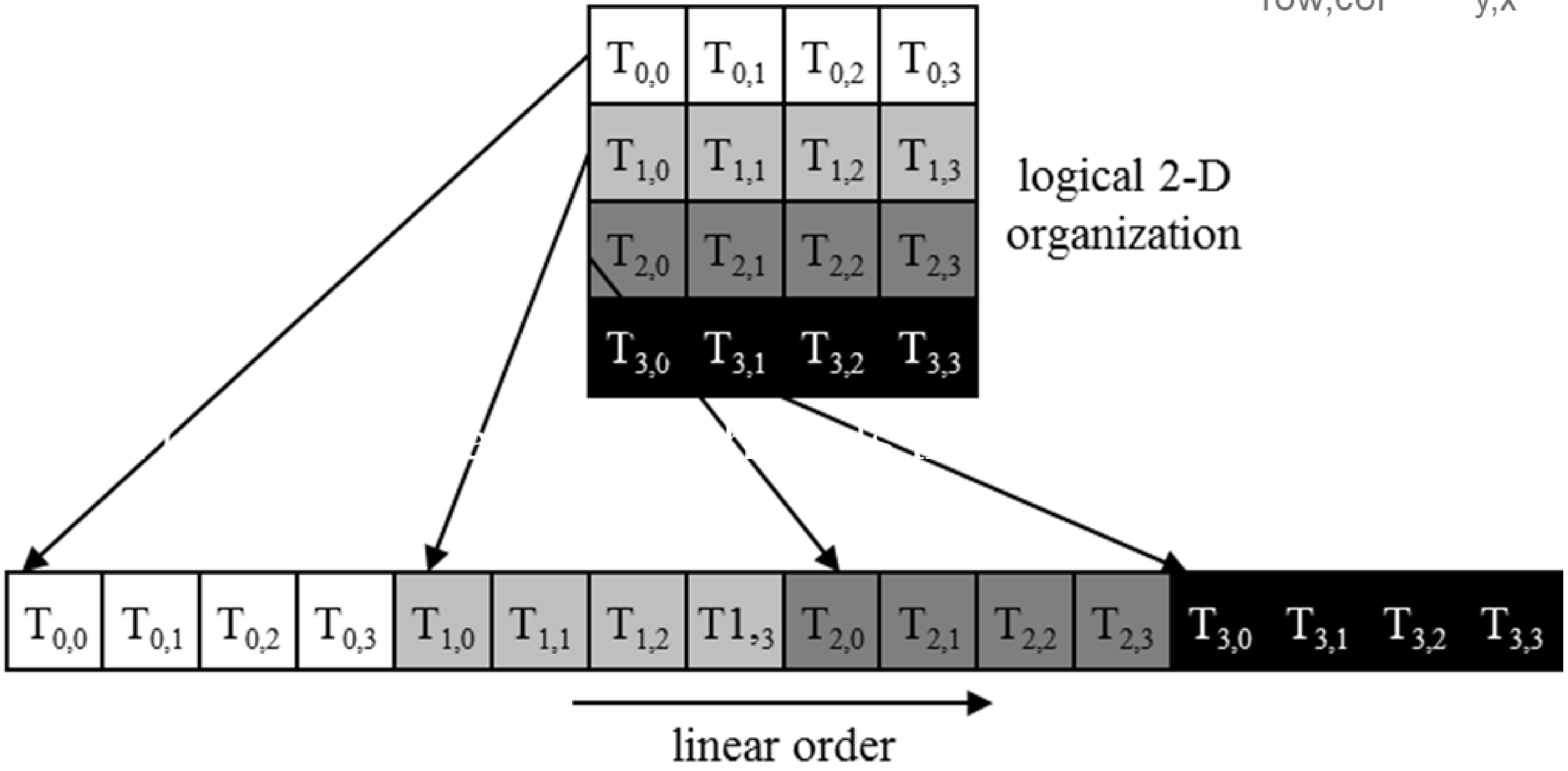


- Each block is divided into 32-thread warps
  - An implementation technique, not part of the CUDA programming model
  - Warps are scheduling units in SM
  - Threads in a warp execute in Single Instruction Multiple Data (SIMD) manner
  - The number of threads in a warp may vary in future generations

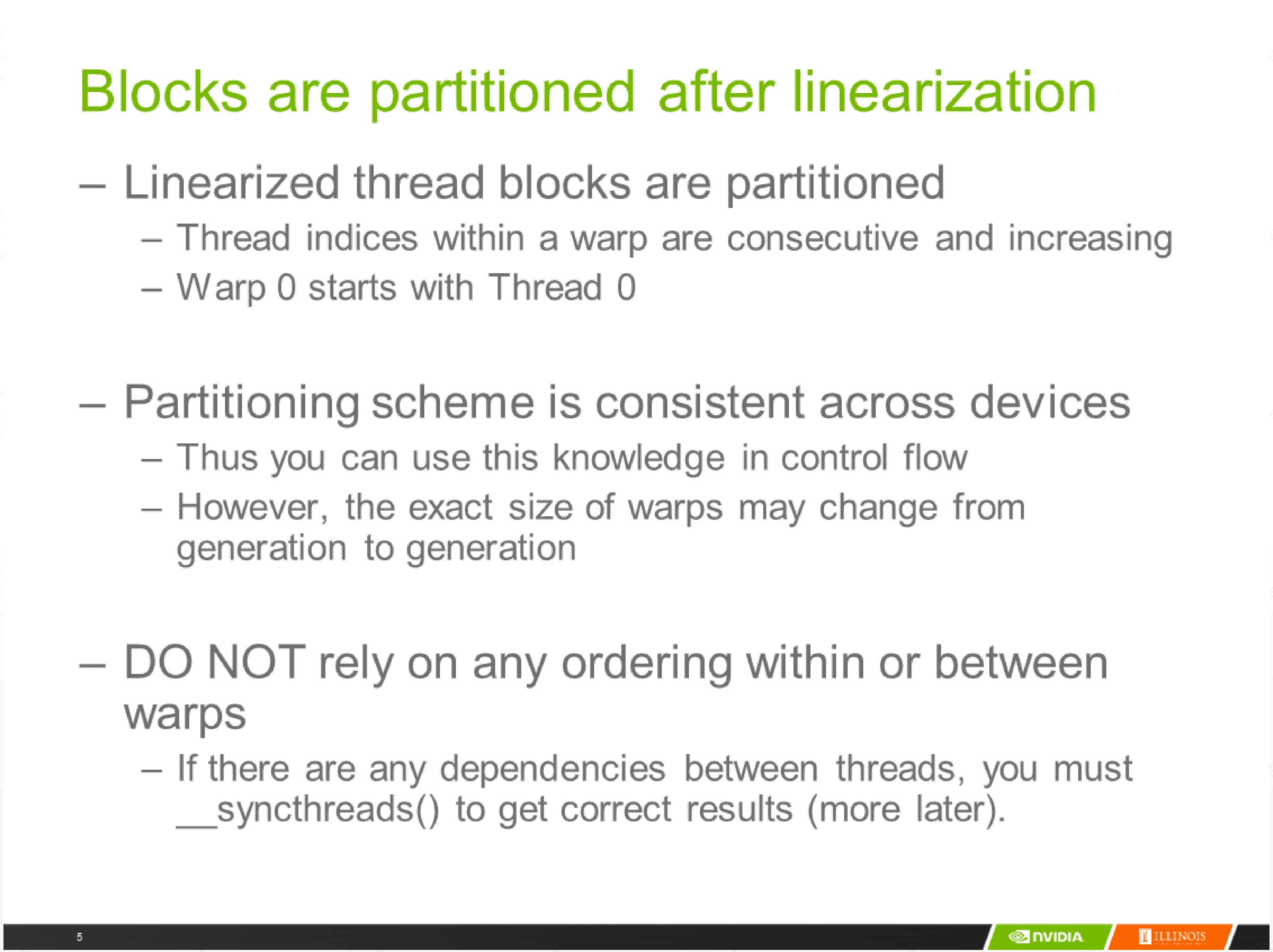
Warps in Multi-dimensional Thread Blocks

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

$T_{row,col} = T_{y,x}$



$T_{0,0}$ $T_{0,1}$ $T_{0,2}$ $T_{0,3}$
$T_{1,0}$ $T_{1,1}$ $T_{1,2}$ $T_{1,3}$
$T_{2,0}$ $T_{2,1}$ $T_{2,2}$ $T_{2,3}$
$T_{3,0}$ $T_{3,1}$ $T_{3,2}$ $T_{3,3}$
logical 2-D organization
$T_{0,0}$ $T_{0,1}$ $T_{0,2}$ $T_{0,3}$ $T_{1,0}$ $T_{1,1}$ $T_{1,2}$ T1,3 $T_{2,0}$ $T_{2,1}$ $T_{2,2}$ $T_{2,3}$ $T_{3,0}$ $T_{3,1}$ $T_{3,2}$ $T_{3,3}$
linear order

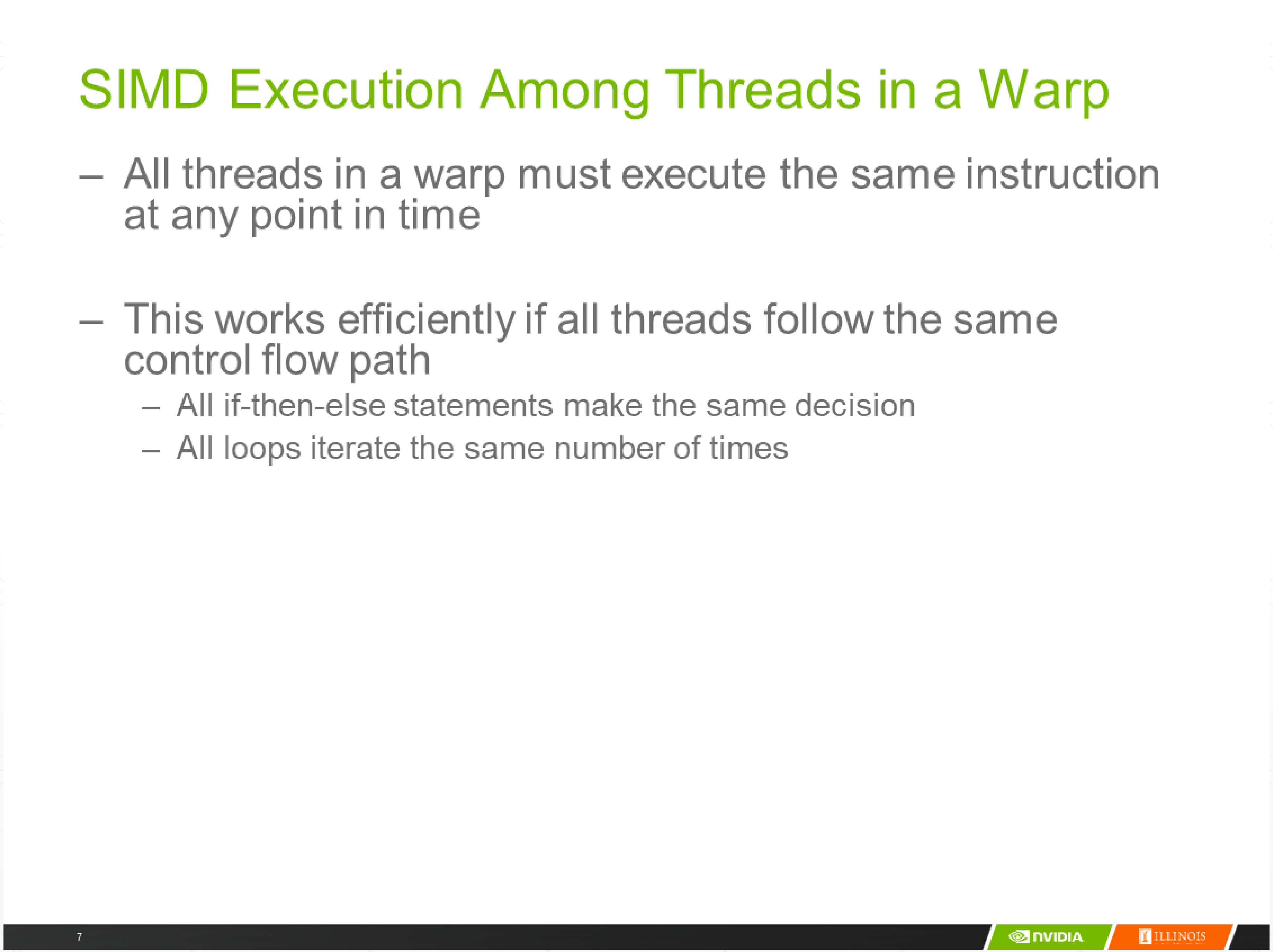


Blocks are partitioned after linearization

- Linearized thread blocks are partitioned
  - Thread indices within a warp are consecutive and increasing
  - Warp 0 starts with Thread 0

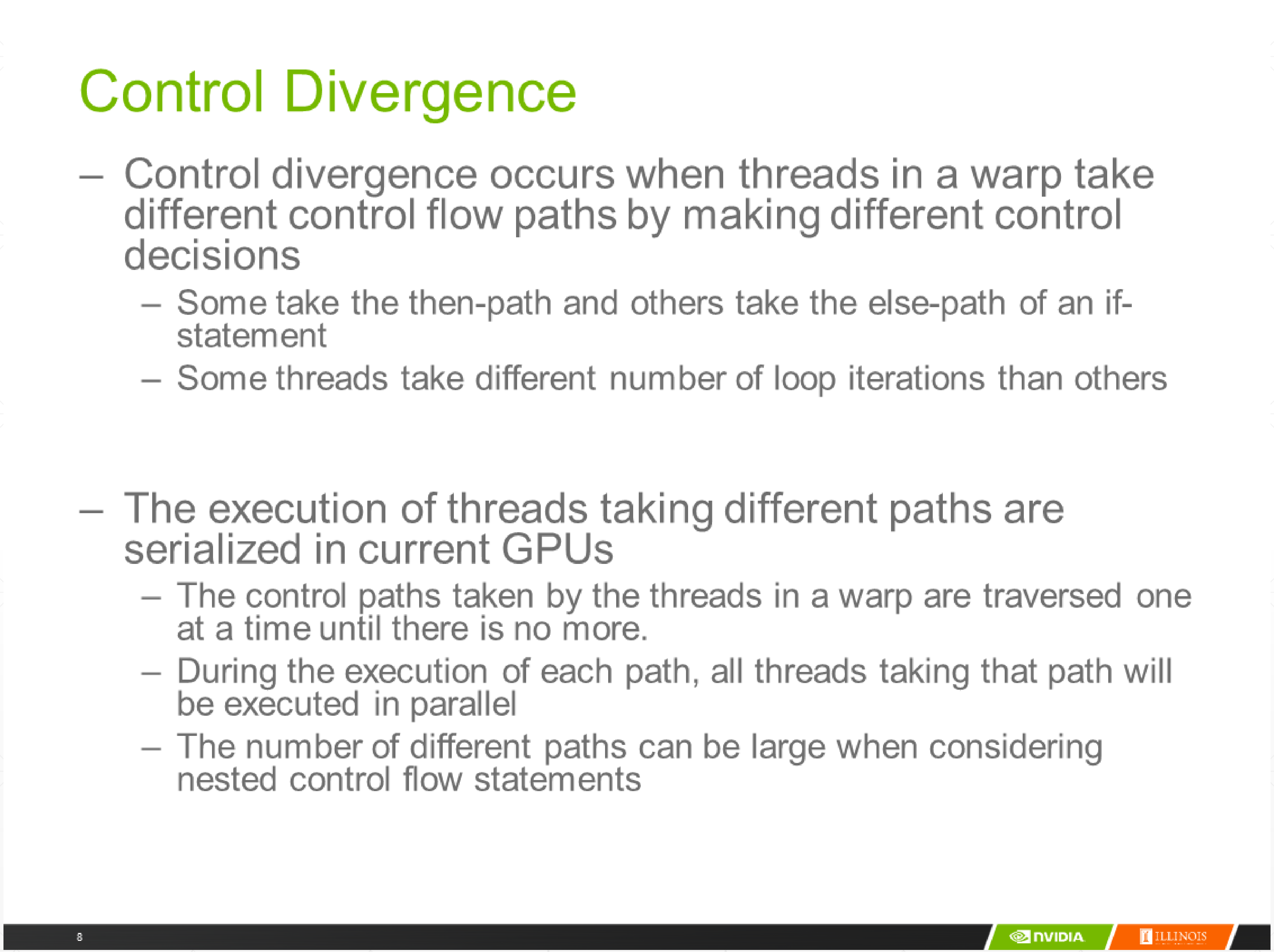
- Partitioning scheme is consistent across devices
  - Thus you can use this knowledge in control flow
  - However, the exact size of warps may change from generation to generation

- 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

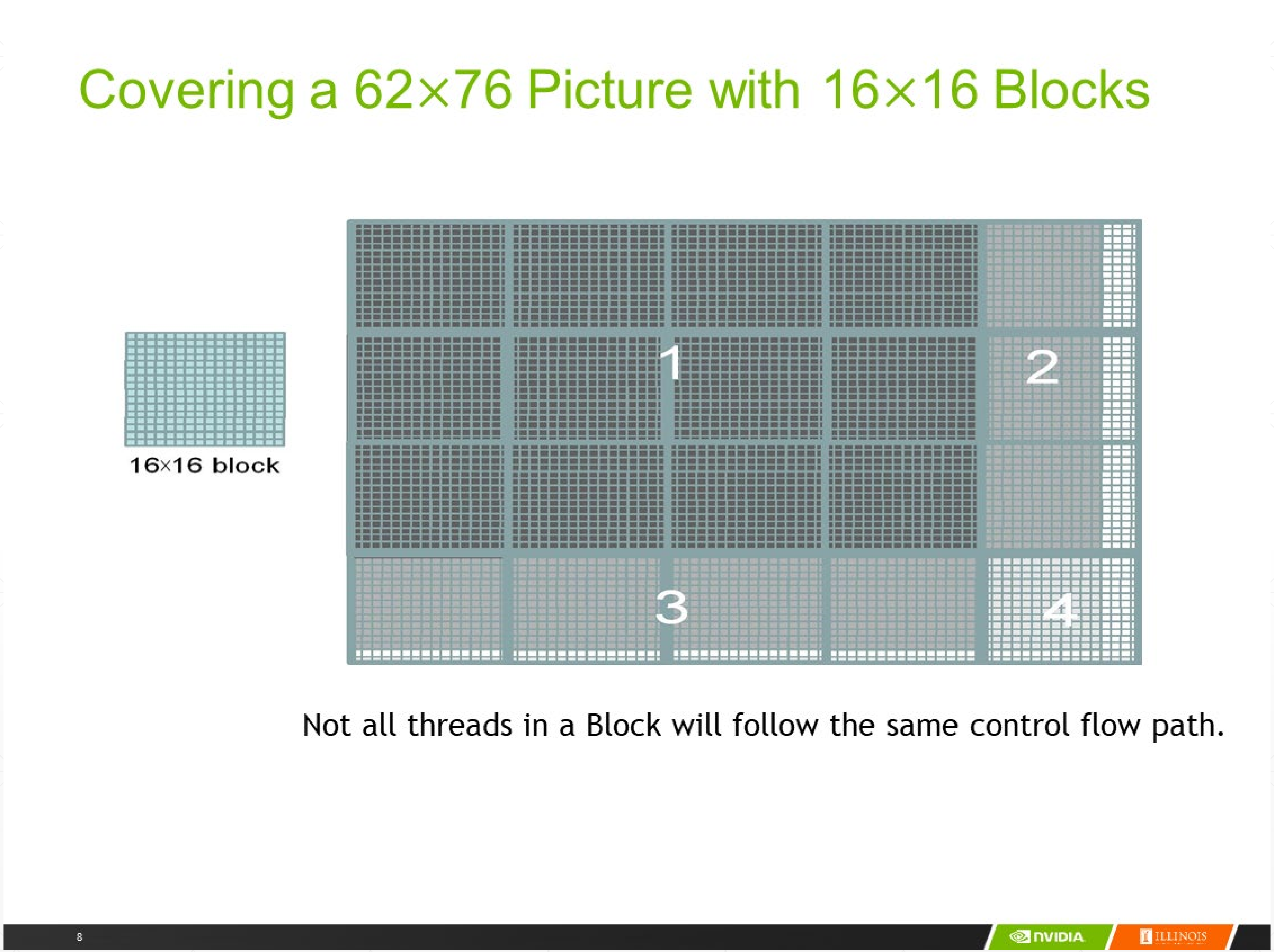
- All threads in a warp must execute the same instruction at any point in time
- This works efficiently if all threads follow the same control flow path
  - All if-then-else statements make the same decision
  - All loops iterate the same number of times



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

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

CUDA TAKE-HOME MESSAGE

- **CUDA, GPU and parallel programming is complex – sorry!!!**
  - In this course we have only shown you the path … but keep in mind that:
  - **There's a difference between knowing the path and walking the path.**
    - Morpheus, The Matrix.
- **THINK PARALLEL** – GPU is a it's a whole different world, change your way of thinking about algorithm implementation.
- **MEMORY is (almost) everything** – even the fastest computing is worthless if you can’t load/store your data!
- **GPU Acceleration** – do not expect x100 speed-up for free;
  - YES, sometimes it is feasible, but requires a lot of optimization and programming effort!

And NOW …

- Consider another break (do not overdo coffee 😉)
- We prepared two Exercises for you:
  - CUDA Memory Model
  - Performance guidelines
- Next … consider a nap or walking a dog …
- Then, continue with the “Ultrasound Hardware & Software” lecture …