UNIVERSITÀ di VERONA

Dipartimento di INFORMATICA

PARCO Parallel Computing Lab

NVIDIA CUDA

Part II

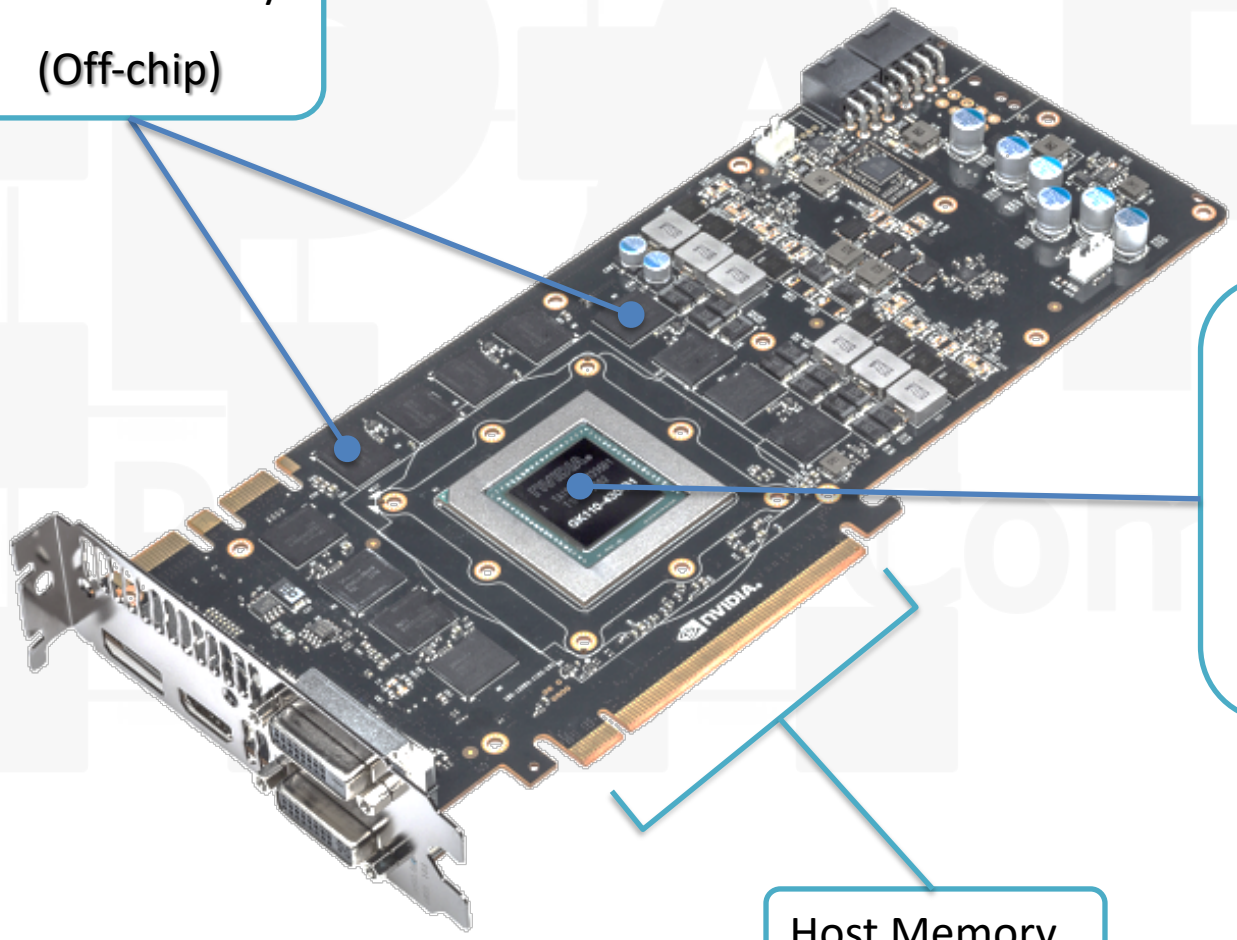
Laurea magistrale in Ingegneria e Scienze Informatiche
Laurea Magistrale in Medical Bioinformatics

Nicola Bombieri – Federico Busato

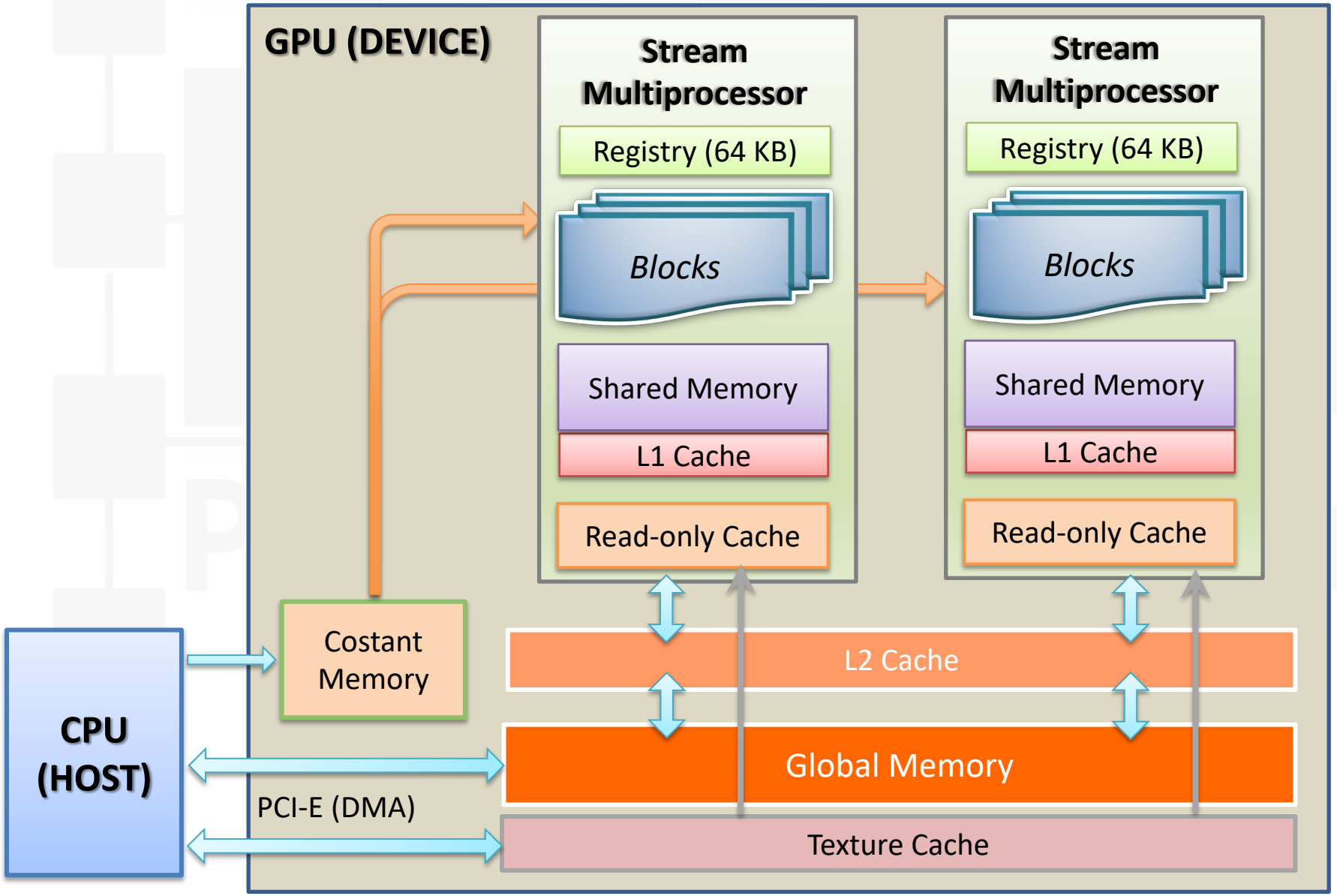
Agenda

- CUDA Memories and Memory Hierarchy
- Warp and Parallelism Hierarchy
- Occupancy
- Synchronization
- Exercises
  - Matrix Multiplication
  - 1D Stencil
  - Matrix Transpose
- References

CUDA Memories (I)



CUDA Memories Hierarchy



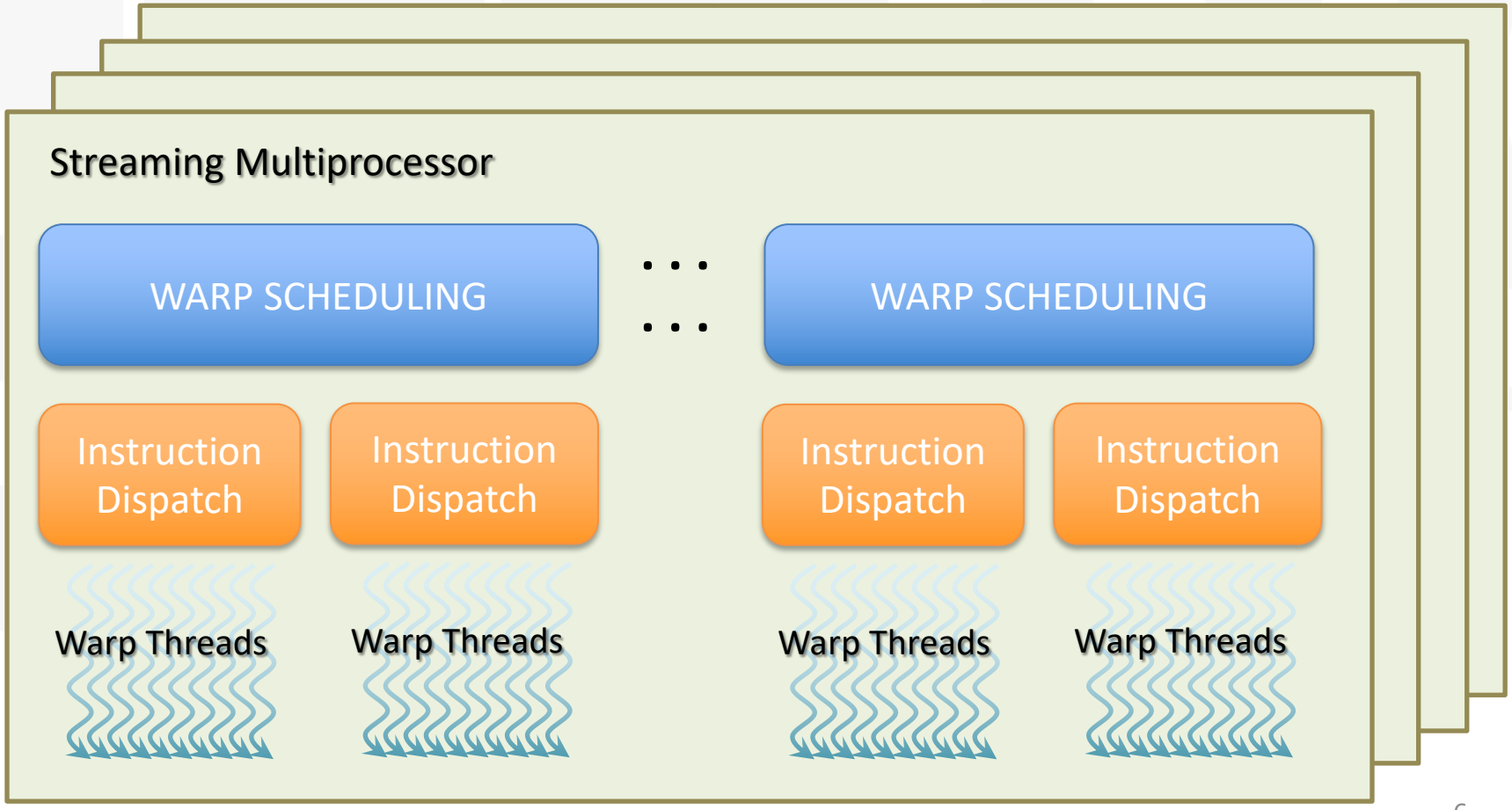
CUDA Warp

- Each Streaming Multiprocessor (SM) contains many Streaming Processors (SP) also called **CUDA Cores**.
- **Warp**: groups of 32 threads that execute in SIMD-like fashion
  - Instructions are SIMD synchronous within a warp: All threads in a warp execute the same instruction
  - Each warp is executed in a single CUDA Core.
- SM schedules and executes Warps that are ready to run
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - Many warp per SMs but only one executing
    - seems a waste of time…
    - … increase efficiency of long-latency operations!
- Kepler's quad warp scheduler selects four warps, and two independent instructions per warp can be dispatched each cycle

CUDA Parallelism Hierarchy

Three Levels of Parallelism:

- Threads
- Warps
- Blocks



Occupancy

- The **GPU Occupancy** as the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps
- Remember: resources are allocated for the entire block
  - Resources are finite
  - Utilizing too many resources per thread may limit the occupancy
- Potential occupancy limiters:
  - Register usage
  - Shared memory usage
  - Block size
- High occupancy does not necessarily translate into a fastest application performance.
- Instruction-level parallelism (ILP) can be equally effective in hiding arithmetic latency by keeping the SIMD cores busy with fewer threads that consume fewer resources and introduce less overhead
- Utilizing fewer threads also benefit kernels that use shared memory/registers by allowing data reuse within a thread

Occupancy Calculator

In the CUDA Toolkit directory (es. /usr/local/cuda-7.5/tools/)

CUDA GPU Occupancy Calculator

Click Here for detailed instructions on how to use this occupancy calculator.

For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Just follow steps 1, 2, and 3 below! (or click here for help)

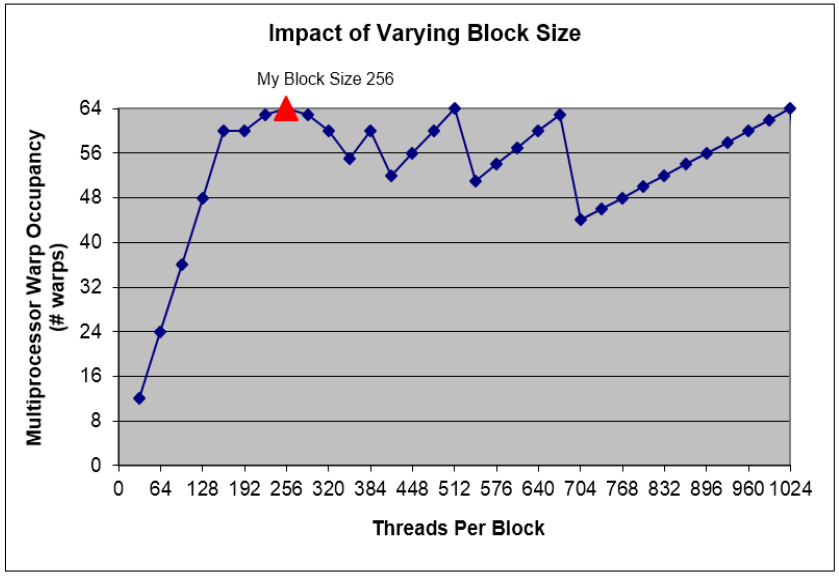
1.) Select Compute Capability (click):	3.0	(Help)
1.b) Select Shared Memory Size Config (bytes)	49152	
2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	32	
Shared Memory Per Block (bytes)	4096	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	2048	(Help)
Active Warps per Multiprocessor	64	
Active Thread Blocks per Multiprocessor	8	
Occupancy of each Multiprocessor	100%	

Physical Limits for GPU Compute Capability:	3.0
Threads per Warp	32
Warps per Multiprocessor	64
Threads per Multiprocessor	2048
Thread Blocks per Multiprocessor	16

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.



Synchronization (II)

Block Synchronization:

CUDA provides a synchronization barrier routine for threads within blocks

```
__syncthreads();
```

"__" indicates hardware intrinsic instruction

- The __SYNCTHREADS() command is a block level synchronization barrier.
- All threads must reach a particular __SYNCTHREADS() routine otherwise deadlock occurs or produce unintended side effects.

- Each warp is executed in a SIMD-like fashion then within a warp is not required explicit synchronization (warp-synchronous programming paradigm)
- More in detail: __SYNCTHREADS() are warp counters.
  - Barriers are executed on a per-warp basis as if all the threads in a warp are active.
  - Thus, if any thread in a warp executes a bar instruction, it is as if all the threads in the warp have executed the bar instruction

Synchronization (III)

Kernel Synchronization:

- Unfortunately no global kernel barrier routine available in CUDA
  - There is no robust way to do inter-block synchronization in the CUDA programming model, as blocks could even execute serially under that model, leading to deadlocks
- To do that, have to use workarounds such as returning from kernel

- Host-to-device data transfers are synchronous, the CPU thread wait until the host-to-device transfer is completed
- Kernels are asynchronous non-blocking with respect to the host code

- CUDA includes two types of host-device synchronization:
  - EXPLICIT: cudaDeviceSynchronize(). In most cases this is overkill, and can really hurt performance due to stalling the entire device and host thread.
  - IMPLICIT: Only after the first kernel is finished the second one will execute Consecutive kernel calls on the same stream are executed sequentially.

Exercises

Memory Hierarchy

- Matrix Multiplication
- 1D Stencil
- Matrix Transpose

Matrix Multiplication (I)

Matrix multiplication is an important application in HPC and appears in many applications

Idea: Use Shared Memory to reuse global memory data

Observations:

1) All threads access global memory for their input matrix elements
2) Two memory accesses per arithmetic operation

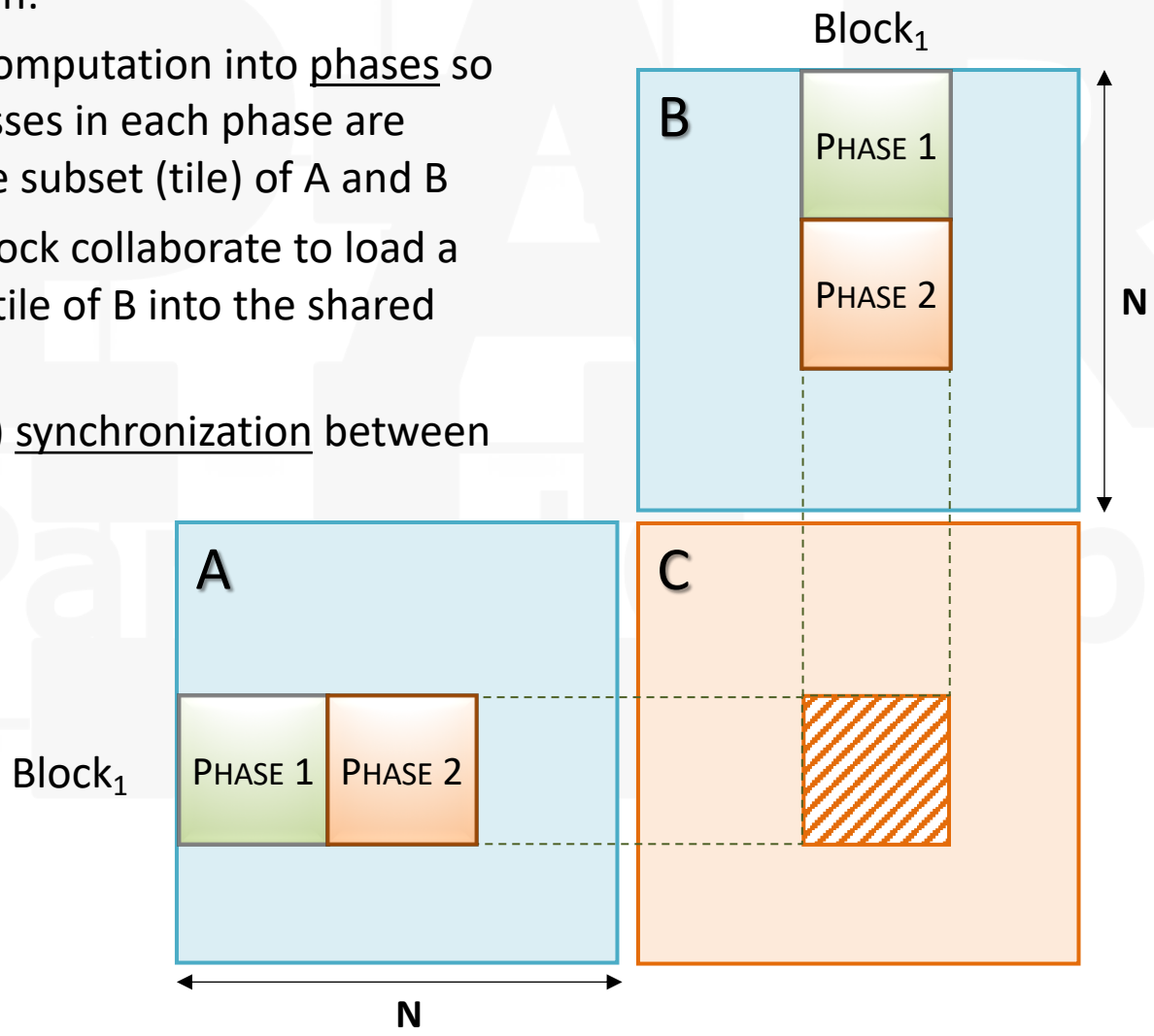
- Shared memory is fast but small → partition data into tiles so that each tile fits into shared memory
- This is feasible because kernel computations on each tile can be done independently of each other
- Locality is extremely important to achieve high performance both in multicore CPUs and GPUs

Matrix Multiplication (II)

Tiling

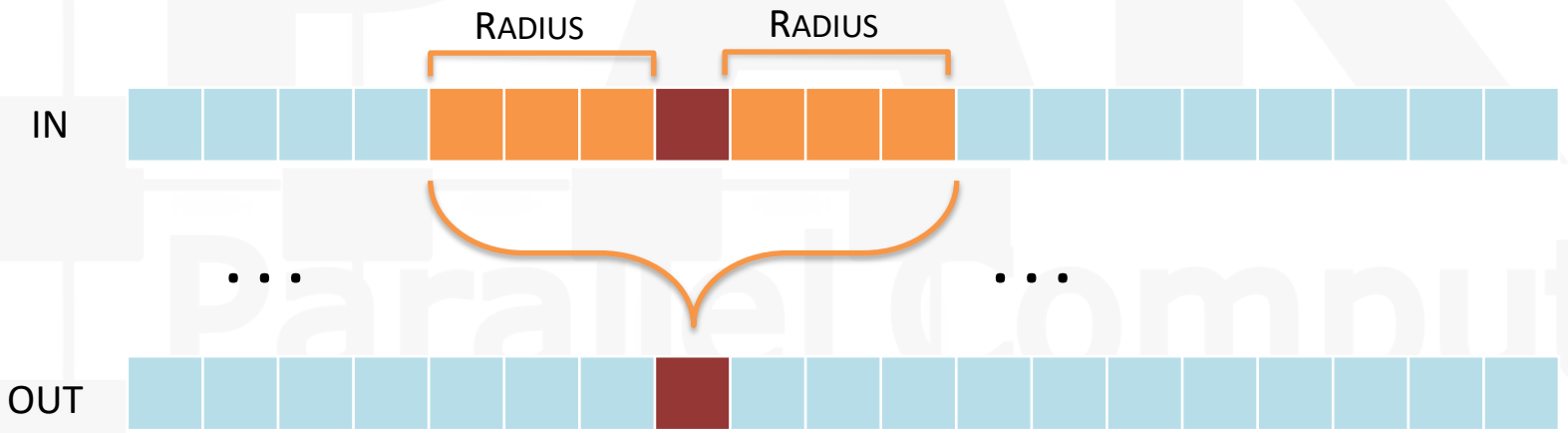
Tiled Multiplication:

- Break up the computation into phases so that data accesses in each phase are focused on one subset (tile) of A and B
- Threads in a block collaborate to load a tile of A and a tile of B into the shared memory
- Require (block) synchronization between each phase



1D Stencil (I)

- Consider applying a 1D stencil to a 1D array of elements
  - Each output element is the sum of input elements within a radius
  - If radius is 3, then each output element is the sum of 7 input elements

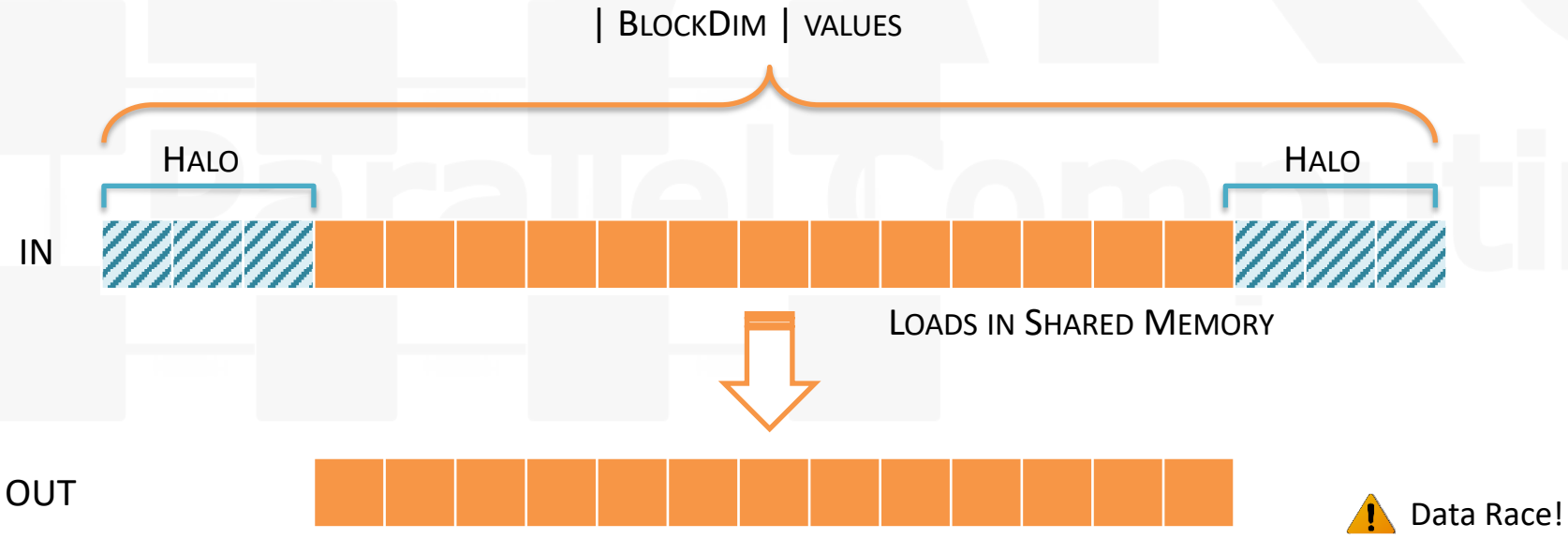


- Fundamental to many algorithms Standard discretization methods, interpolation, convolution, filtering

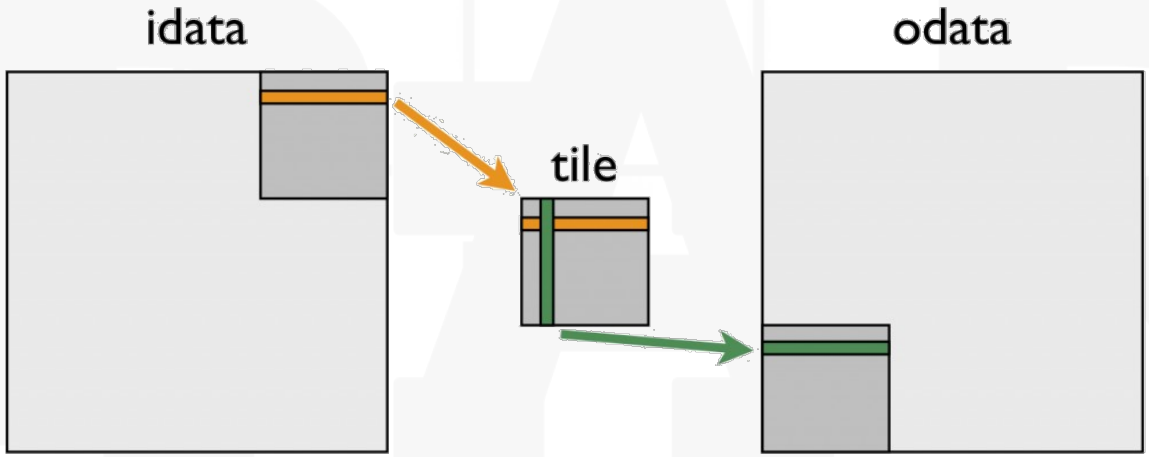
1D Stencil (II)

Cache data in shared memory

- Read (blockDim.x + 2 * RADIUS) input elements from global memory to shared memory
- Compute | BLOCKDIM | output elements
- Write | BLOCKDIM | output elements to global memory
- Each block needs a halo of radius elements at each boundary



Matrix Transpose



- It is very similar to the simple copy
- Use shared memory to avoid the large strides through global memory.
- Conceptually partition the input matrix into square tiles

References

- CUDA C Programming Guide

  http://docs.nvidia.com/cuda/cuda-c-programming-guide/

- CUDA Runtime API

  http://docs.nvidia.com/cuda/cuda-runtime-api/

  http://developer.download.nvidia.com/compute/cuda/4_1/rel/toolkit/docs/online/modules.html

  (Memory Management)

- CUDA Math API

  http://docs.nvidia.com/cuda/cuda-math-api/