

GPU Programming

Lubomír Říha, Kristian Kadlubiak, Jakub Homola
IT4Innovations, VSB-TU Ostrava

Univerza v Ljubljani



Co-funded by the
Erasmus+ Programme
of the European Union

This project has been funded with support from the European Commission.

This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Course outline

	Beginning	End	Description			Slot duration
1	9:00	10:30	<ol style="list-style-type: none"> Heterogeneous Parallel Computing GPU Architecture Hands-on: Accessing GPU accelerated nodes Hands-on: Benchmark HW properties CUDA Programming 	<ol style="list-style-type: none"> 9 slides 15 slides -- -- 21 slides 	<ol style="list-style-type: none"> 15 minutes 20 minutes 10 minutes 10 minutes <u>25 - 30 minutes</u> Total: 80 – 85 minutes	90 min
	10:30	10:45	Coffee break			
2	10:45	11:45	<ol style="list-style-type: none"> Hands-on: Hello World in CUDA CUDA Programming cont. Hands-on: Vector Addition (single GPU, two versions) Multi-GPU programming 	<ol style="list-style-type: none"> . 10 slides -- 10 slides 	<ol style="list-style-type: none"> 10 minutes 10 – 15 minutes 15 minutes <u>10 – 15 minutes</u> Total: 45 – 55 minutes	60 min
	11:45	12:00	Coffee break			
3	12:00	13:00	<ol style="list-style-type: none"> Hands-on: Vector Addition (multi-GPU, two versions) Multi-Dimensional Grids Hands-on: Image Blur Thread Execution CUDA Memories 	<ol style="list-style-type: none"> . 10 slides -- 9 slides 5 slides 	<ol style="list-style-type: none"> 15 minutes 10 - 15 minutes 10 minutes 10 - 12 minutes <u>5 minutes</u> Total: 50 – 57 minutes	60 min
	13:00	14:00	Lunch break			
4	14:00	15:15	<ol style="list-style-type: none"> Global Memory Hands-on: Matrix Sum Shared Memory Memory and Data Locality: Tiling Technique Hands-on: Tiled Matrix Multiplication 	<ol style="list-style-type: none"> 12 slides -- 13 slides 45 slides -- 	<ol style="list-style-type: none"> 15 minutes <u>10 minutes</u> 10 minutes 35 - 45 minutes <u>10 minutes</u> Total: 70 – 80 minutes	75 min
	15:15	15:30	Coffee break			
5	15:30	16:45	<ol style="list-style-type: none"> Parallel Computation Patterns: Stencil Parallel Computation Patterns: Reduction Parallel Computation Patterns: Histogram Efficient Host-Device Data Transfer and CUDA Streams Hands-on: Heat Transfer mini-apt 	<ol style="list-style-type: none"> 23 slides 9 slides 15 slides 10 slides . 	<ol style="list-style-type: none"> 25 – 30 minutes 10 – 15 minutes 15 – 20 minutes 10 – 15 minutes <u>30 minutes</u> Total: 65 – 75 minutes	75 min
	16:45	17:00	Closing remarks			Total: 360 minutes (6 hours)

Heterogeneous Parallel Computing

Univerza v Ljubljani



TECHNISCHE
UNIVERSITÄT
WIEN



VSB TECHNICAL
UNIVERSITY
OF OSTRAVA

IT4INNOVATIONS
NATIONAL SUPERCOMPUTING
CENTER

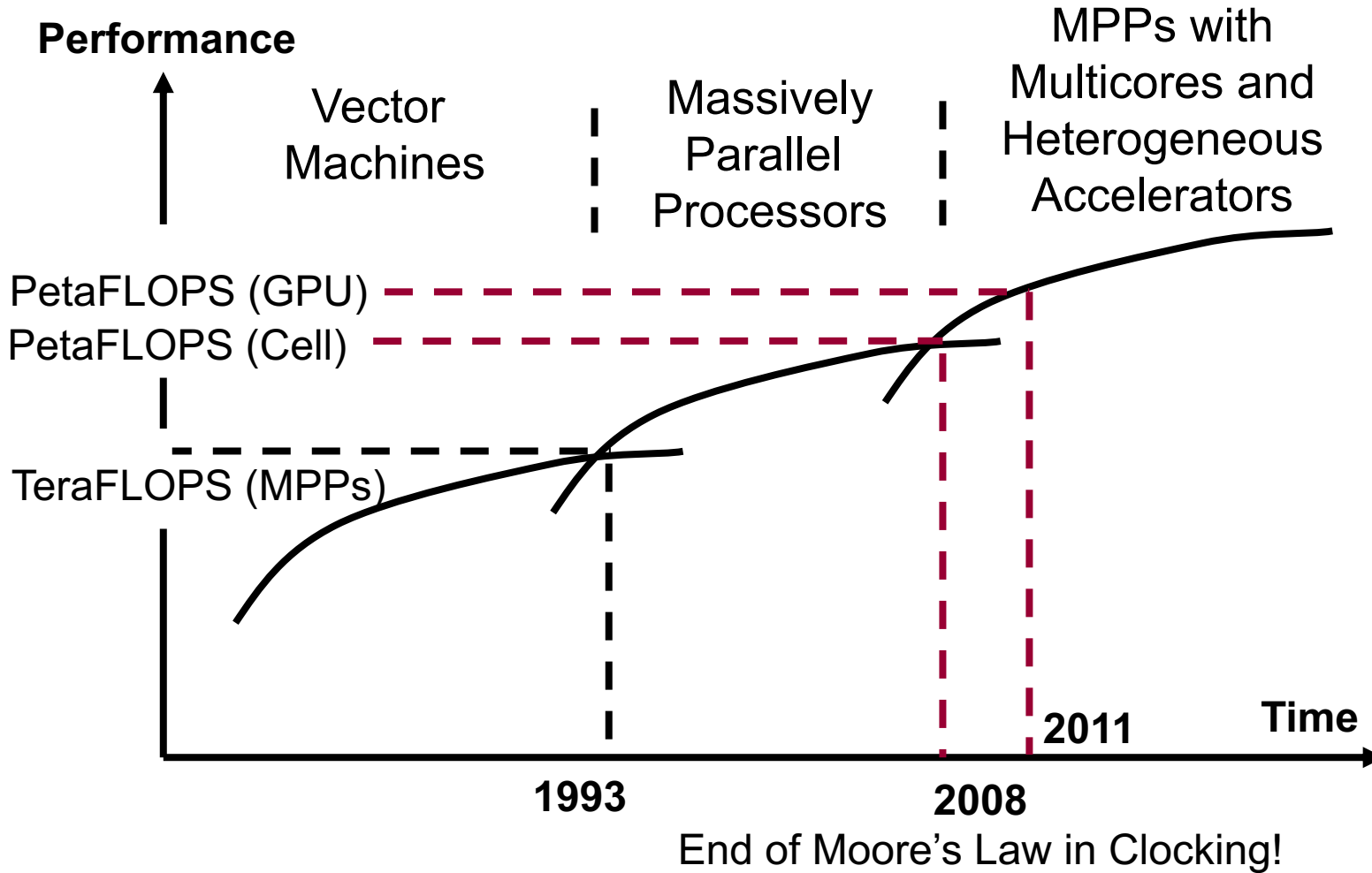


Co-funded by the
Erasmus+ Programme
of the European Union

This project has been funded with support from the European Commission.

This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Accelerators in HPC Historical Analysis

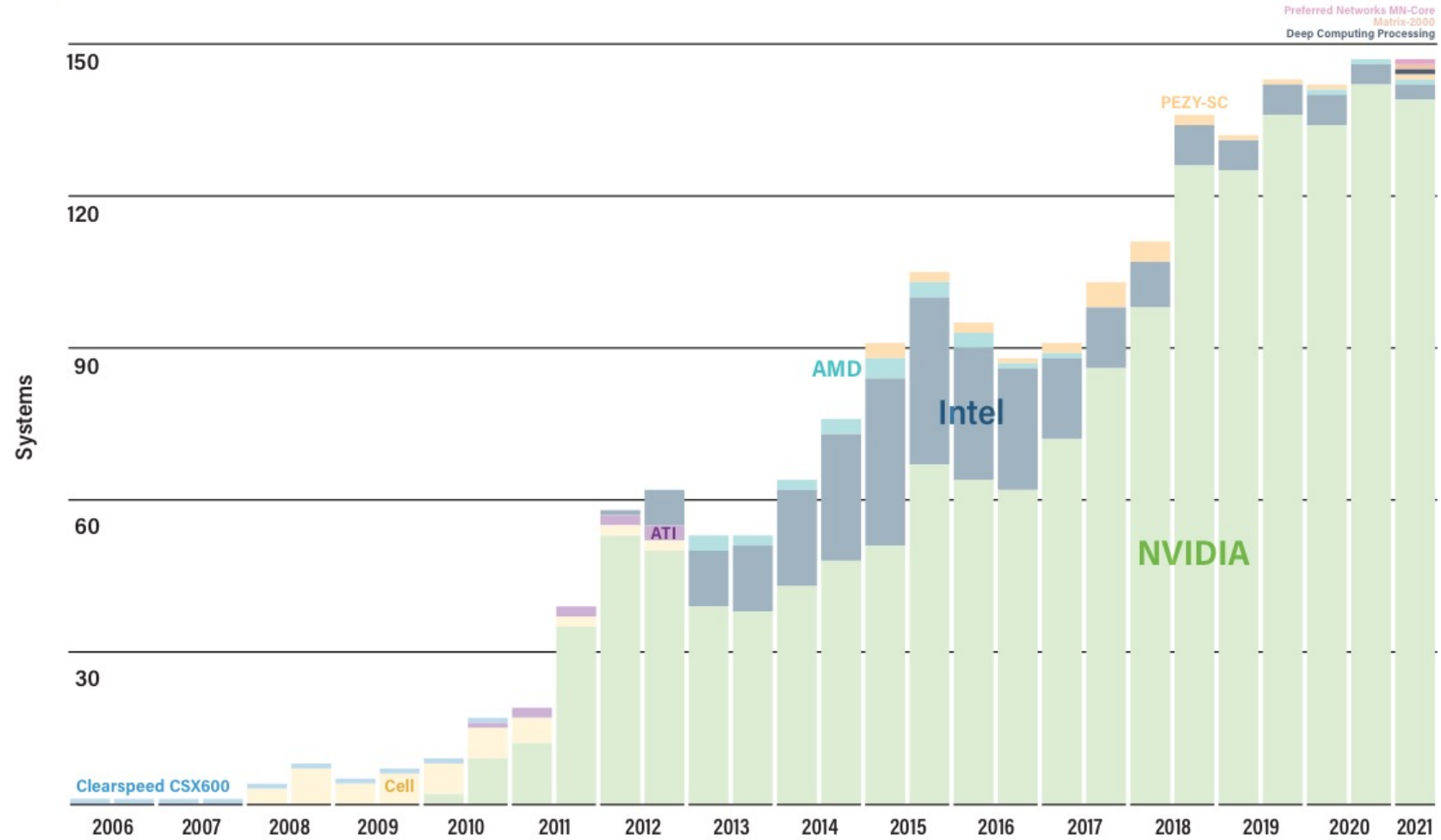


IBM Roadrunner (2008)

- the first heterogeneous supercomputer
- installed in Los Alamos National Lab
- 6,480 AMD Opteron processors
 - with 52 TB RAM
- 12,960 PowerXCell 8i processors
- 296 racks - 2.35 MW power consumption

Accelerators in HPC Historical Analysis

Computer	# CPU cores	Year
Fugaku, Japan	7 630 848	2020
Summit, USA	2 414 592	2018
Sunway TAIHULIGHT	10 649 600	2016
TIANHE-2, CHINA	3 120 000	2015
Titan, USA	560 640	2012
Sequoia, USA (BlueGene/Q)	1 572 864	2012
K-Computer, Japan	548 352	2011
Tianhe-1A, China	186 368	2010
Jaguar, Cray	224 162	2009
Roadrunner, USA	122 400	2008
BlueGene/L	212 992	2007



Accelerators in HPC

ORNL Summit Supercomputer

Feature	
Number of Nodes	4,608
Performance	200 PF Peak, 148 Linpack (FP64) 3.3 ExaOps (FP16)
Node performance	42 TF
Memory per Node	512 GB DDR4 + 96 GB HBM2
NV memory per Node	1600 GB
Total System Memory	>10 PB DDR4 + HBM2 + Non-volatile
System Interconnect	Dual Rail Infiniband EDR (25 GB/s)
Interconnect Topology	Non-blocking Fat Tree
Processors	2x IBM POWER9 6x NVIDIA Volta
File System	250 PB, 2.5 TB/s, GPFS
Power Consumption	13 MW



Summit: DOE/SC/Oak Ridge National Laboratory

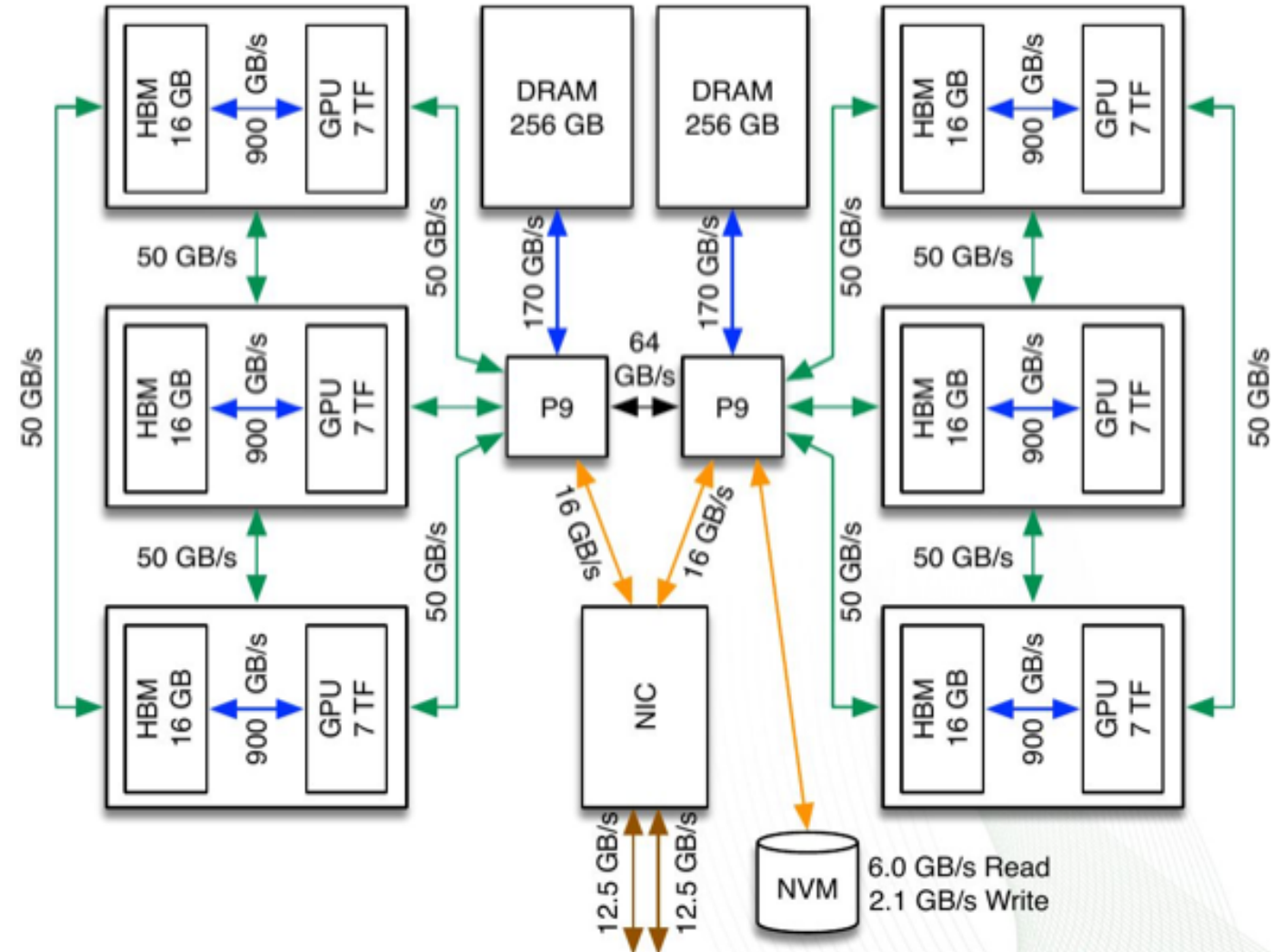
No.1 from Jun 2018 until Nov 2019

Currently no. 2

Accelerators in HPC

ORNL Summit Supercomputer

- Coherent memory across entire node
- NVLink v2 fully interconnects three GPUs and one CPU on each side node
- PCIe Gen4 connects NVMe and NIC
- Single shared NIC with dual EDR ports

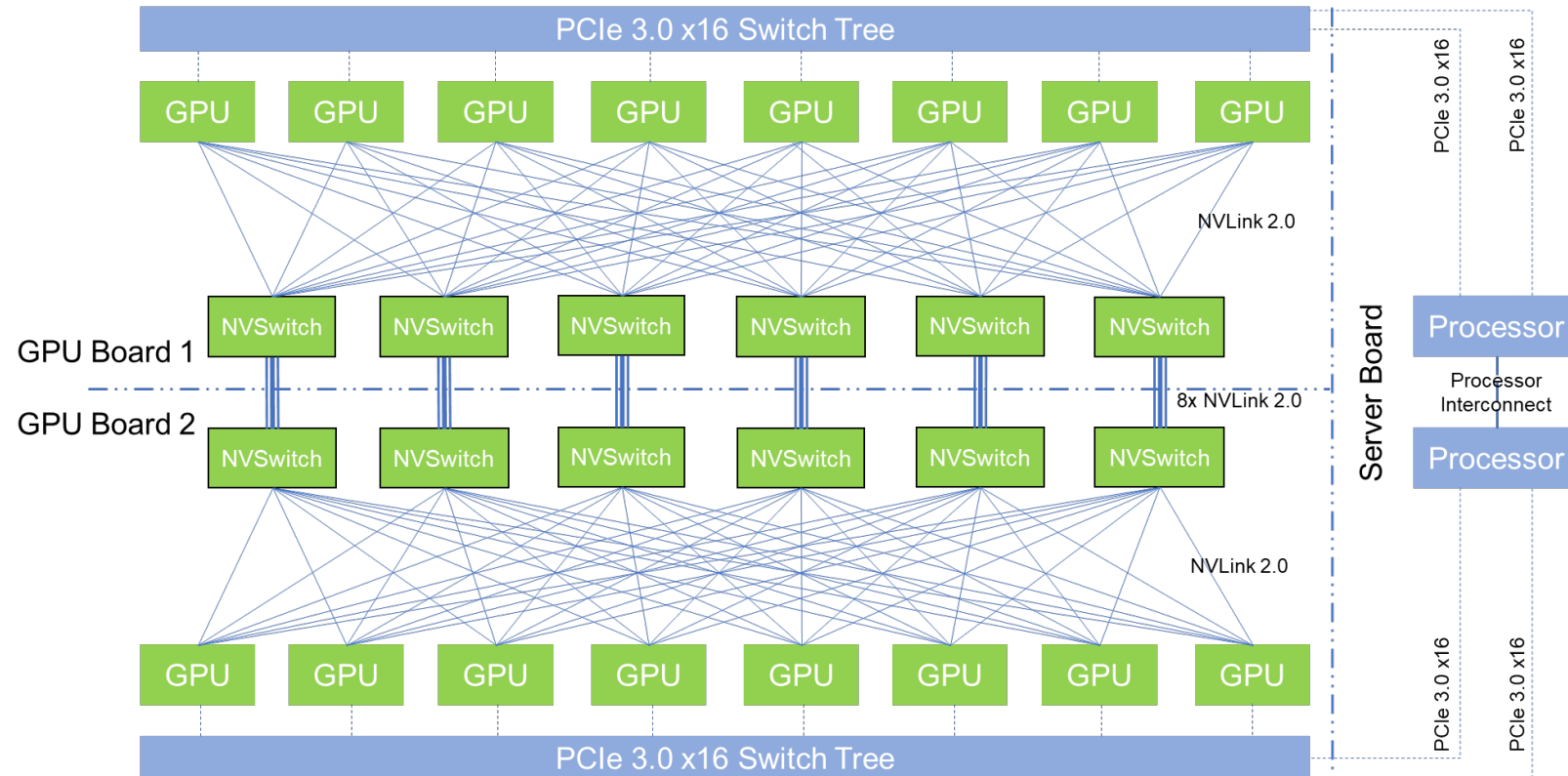


Accelerators in HPC

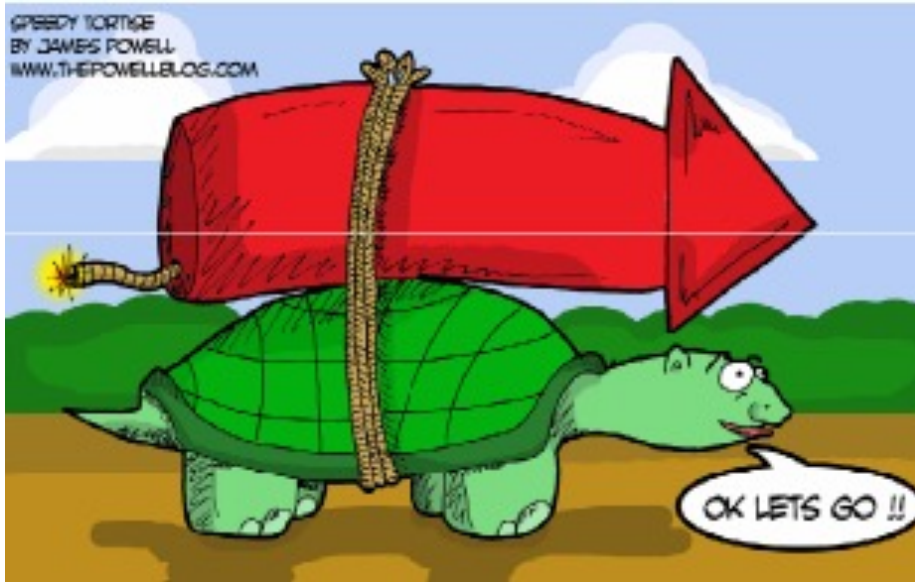
Distributed Shared Memory GPU Systems

NVIDIA DGX-2

- 2x x86 CPUs with 1.5 TB RAM
- 16x Nvidia Tesla V100 GPU (Volta architecture)
 - 2560 FP64 cores
 - 5120 FP32 cores
 - 640 tensor cores
 - 32 GB HBM2 memory @ 900GB/s
- 512GB HBM total GPU memory
- 6x NVlink @ 25+25GB/s = 150+150 GB/s total
- NVLINK network interconnecting GPGPU
- 12x NVSwitch, throughput 2.4TB/s in bisection
- 8x 100Gb/s Infiniband
- NVMe SSD storage 30TB
- 130TF Peak!



Accelerators in HPC Heterogeneous Computing

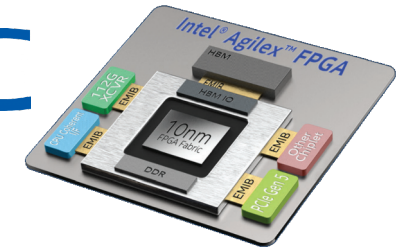
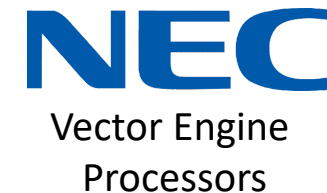
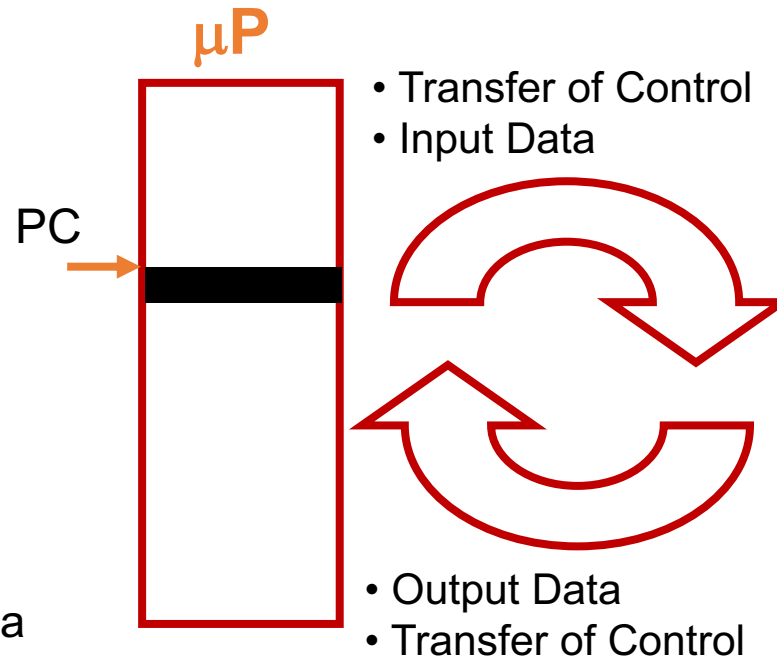


Main Features

- Coprocessor to the CPU
- PCIe based interconnection
- Separate GPU memory
- Provide high bandwidth access to local data
- Slow access to the CPU memory

Hardware Accelerators - Speeding up the Slow Part of the Code

- Enable higher performance through fine-grained parallelism
- Offer higher computational density than CPUs
- Accelerators present heterogeneity!



Accelerators

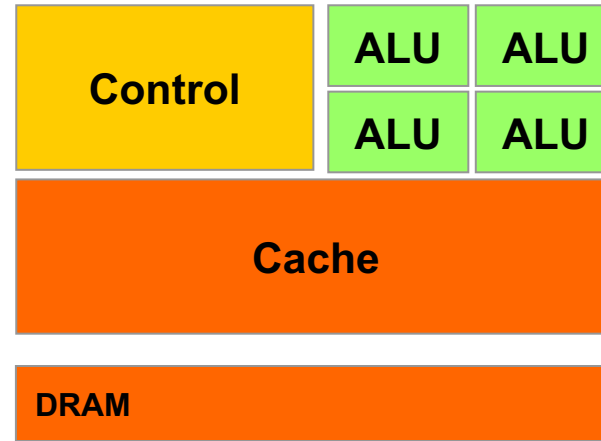
- tailored for compute-intensive, highly data parallel computation
- many parallel execution units
- have significantly faster and more advanced memory interfaces
- more transistors is devoted to data processing
- less transistors for data caching and flow control

Very Efficient For

- Fast Parallel Floating Point Processing
- High Computation per Memory Access

Not As Efficient For

- Branching-Intensive Operations
- Random Access,
- Memory-Intensive Operations



CPUs

Powerful ALU

- reduced operation latency

Large caches

- convert long latency memory accesses to short latency cache accesses

Sophisticated control with branch prediction for reduced branch latency

GPUs

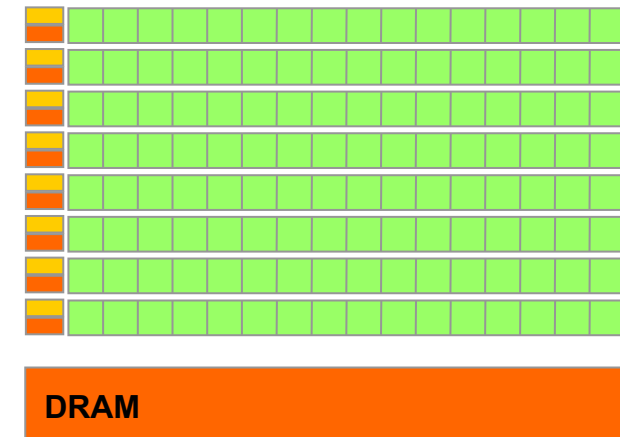
Small caches to boost memory throughput

Simple control with no branch prediction

Energy efficient ALUs

- many, long latency but heavily pipelined for high throughput

Require massive number of threads to tolerate latencies



Accelerators

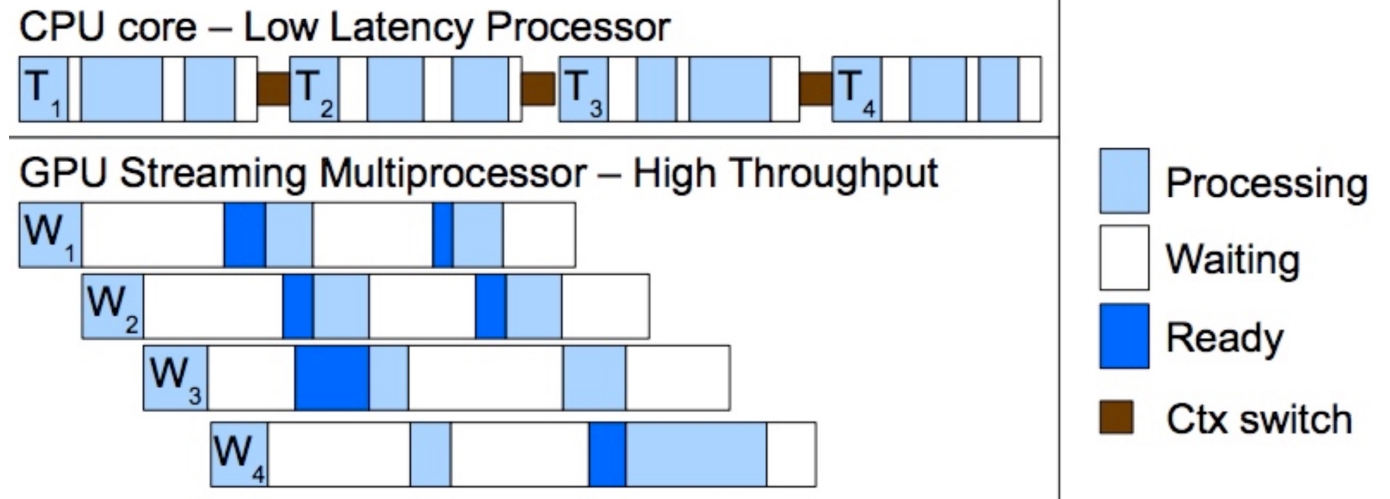
- tailored for compute-intensive, highly data parallel computation
- many parallel execution units
- have significantly faster and more advanced memory interfaces
- more transistors is devoted to data processing
- less transistors for data caching and flow control

Very Efficient For

- Fast Parallel Floating Point Processing
- High Computation per Memory Access

Not As Efficient For

- Branching-Intensive Operations
- Random Access,
- Memory-Intensive Operations



NVIDIA Corporation 2010

GPU are throughput devices

- CPU cores are optimized to minimize latency between operations.
- GPUs aim to minimize latency between operations by scheduling multiple warps (thread bundles).

Accelerators in HPC

Device	Fabrication process [nm]	Clock frequency [GHz]	No. of cores	Peak floating point performance SP/DP [GFLOPs]	Peak power consumption [W]	Perf. Per Watt SP/DP [GFLOPs/W]	Theoretical Memory Bandwidth [GB/s]	Memory type
Intel Xeon Platinum 8180	14	1.7	28	3046/1523	205	15/7	128	DDR4
nVidia Tesla V100	12	1.246	5120DP 2560SP	15700/7800	300	52/26	900	HBM2
Intel Xeon Phi KNL	14	1.3	64	5324/2662	215	25/12	400/102.4	MCDRAM/ DDR4
Matrix-2000	?	1.2	128	4914/2457	240	20/10	143.1	DDR4
NEC SX-Aurora	16	1.6	8	4900/2450	?	?	1200	HBM2
Intel Stratix 10 DX	14	1?	11520 DSPs	8600 (SP)	?	?	512	HBM2
Intel Agilex	10	?	?	40000 (FP16)	?	?	512	HBM2
Xilinx Alveo U280	16	?	9024 DSP	24.5 (INT8 TOPs)	225	109 GOPs/W	38/460	DDR4/ HBM2
Xilinx Alveo U250	16	?	12288	33.3 (INT8 TOPs)	225	148 GOPs/W	77	DDR4

GPU Architecture

Univerza v Ljubljani



Co-funded by the
Erasmus+ Programme
of the European Union

This project has been funded with support from the European Commission.
This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Accelerators in HPC

Evolution of Graphics Processors

Till 90s

- VGA controllers used to accelerate some display functions

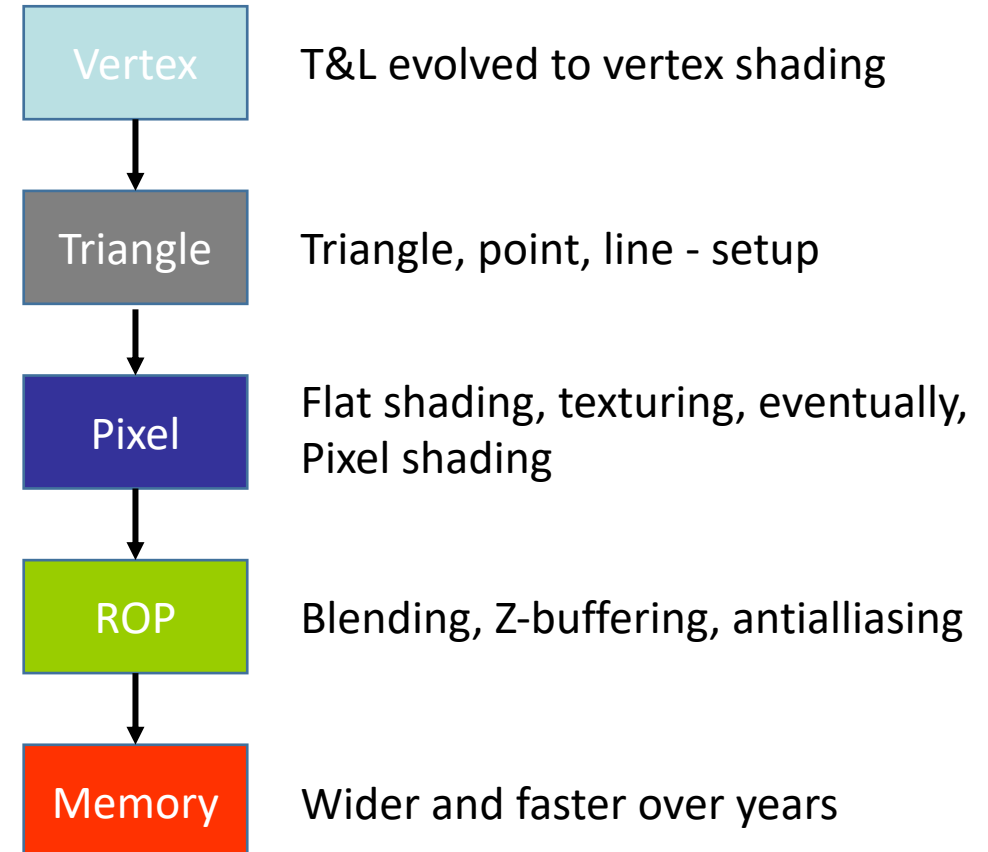
Mid 90s to mid 00s

- Fixed-function graphic accelerators for the OpenGL and DirectX APIs
 - Some GP-GPU capabilities on top of the interface
- 3D graphic: triangle setup & rasterization, texture mapping & shading

Modern GPUs

- Programmable multiprocessors (optimized for data-parallel ops)
 - OpenGL/DirectX and general purpose language
- Some fixed function hardware (texture, raster, ops,)

Graphic Pipeline (for last 20 years)

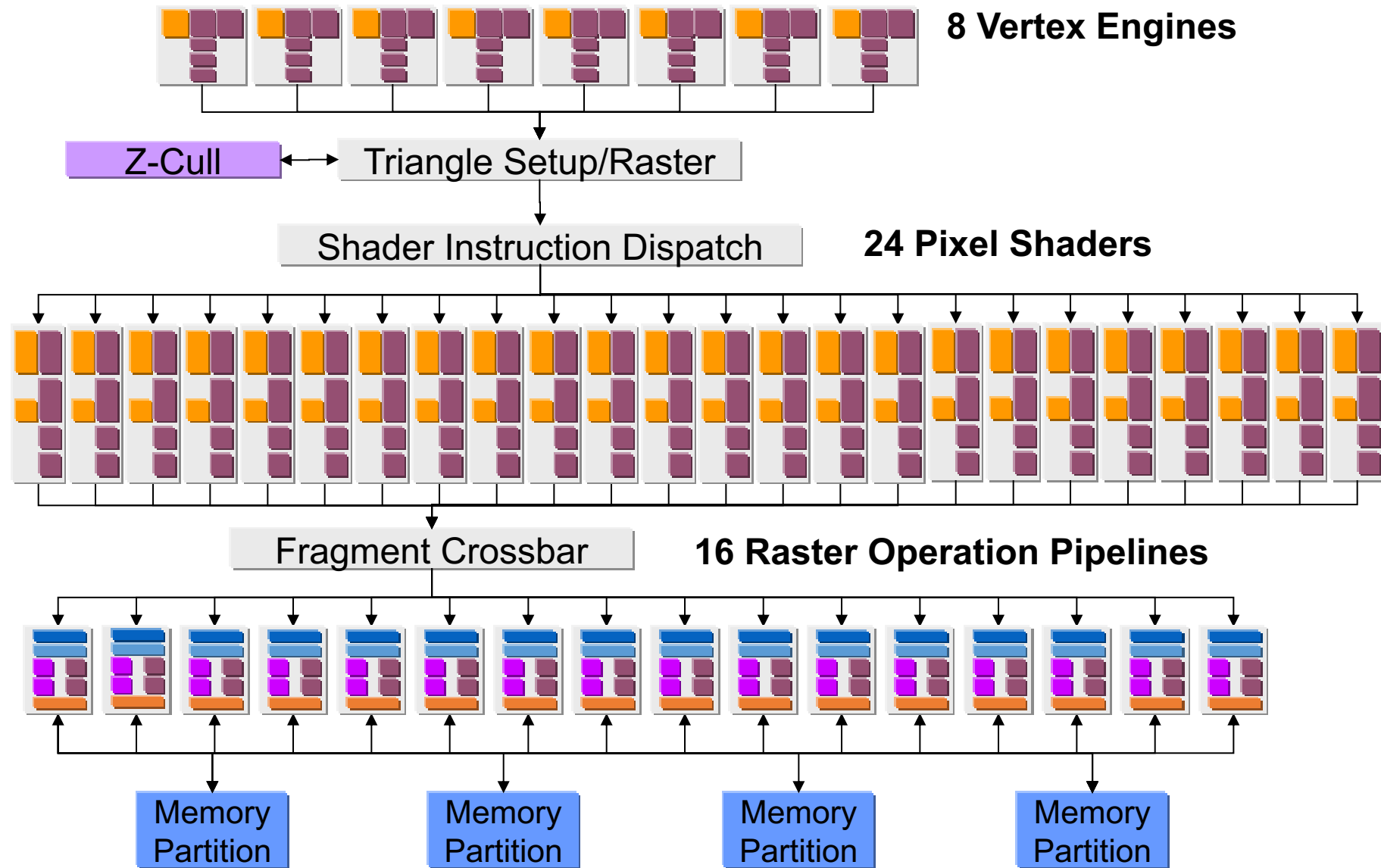


Accelerators in HPC

Non-unified GPU Architecture GeForce 7800 GTX

SCtrain

SUPERCOMPUTING
KNOWLEDGE
PARTNERSHIP



Accelerators in HPC

Why Unify Shader Processors?

Vertex Shader



Pixel Shader



Heavy Geometry
Workload Perf = 4

Vertex Shader



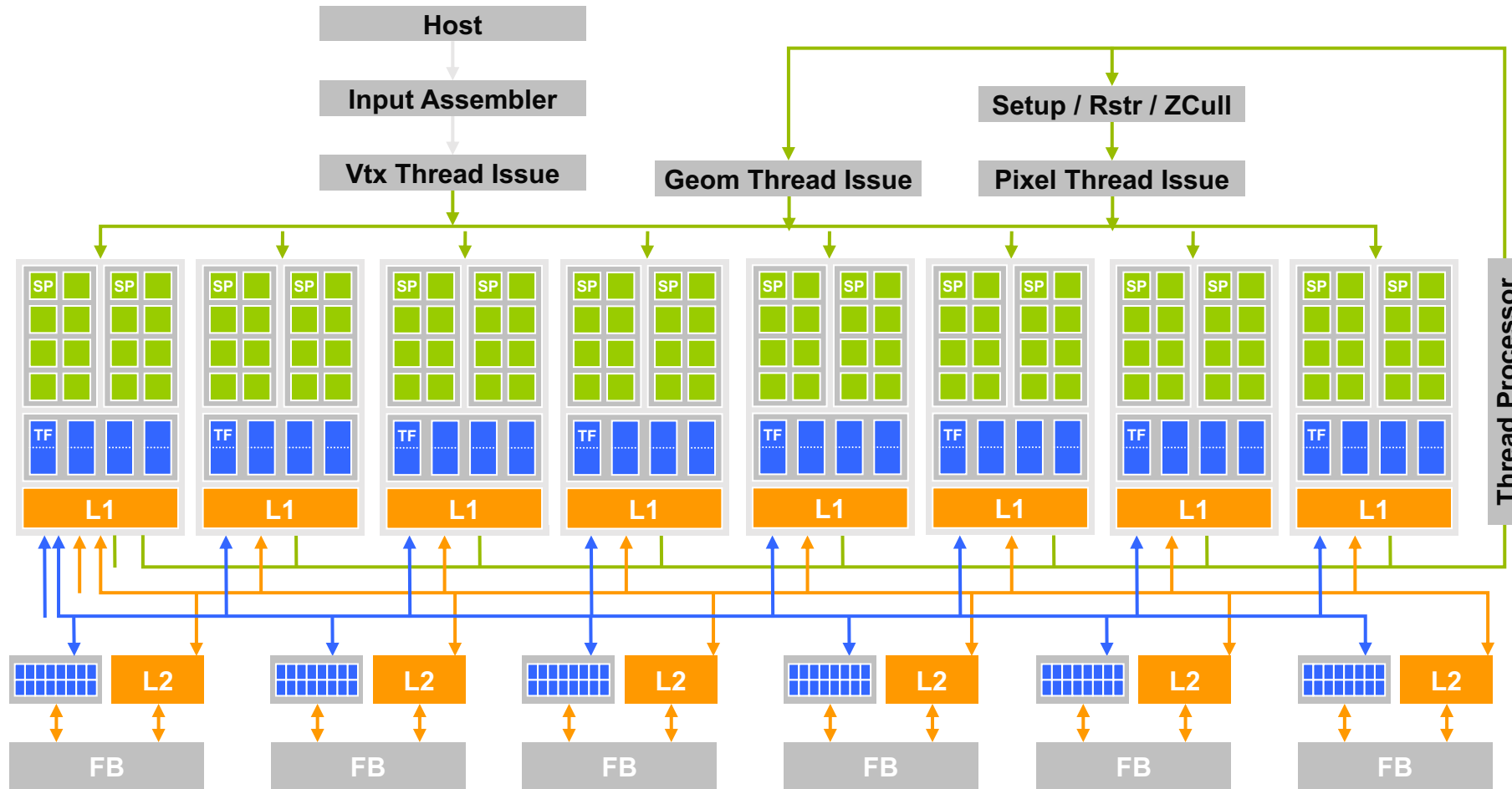
Pixel Shader



Heavy Pixel
Workload Perf = 8

Accelerators in HPC

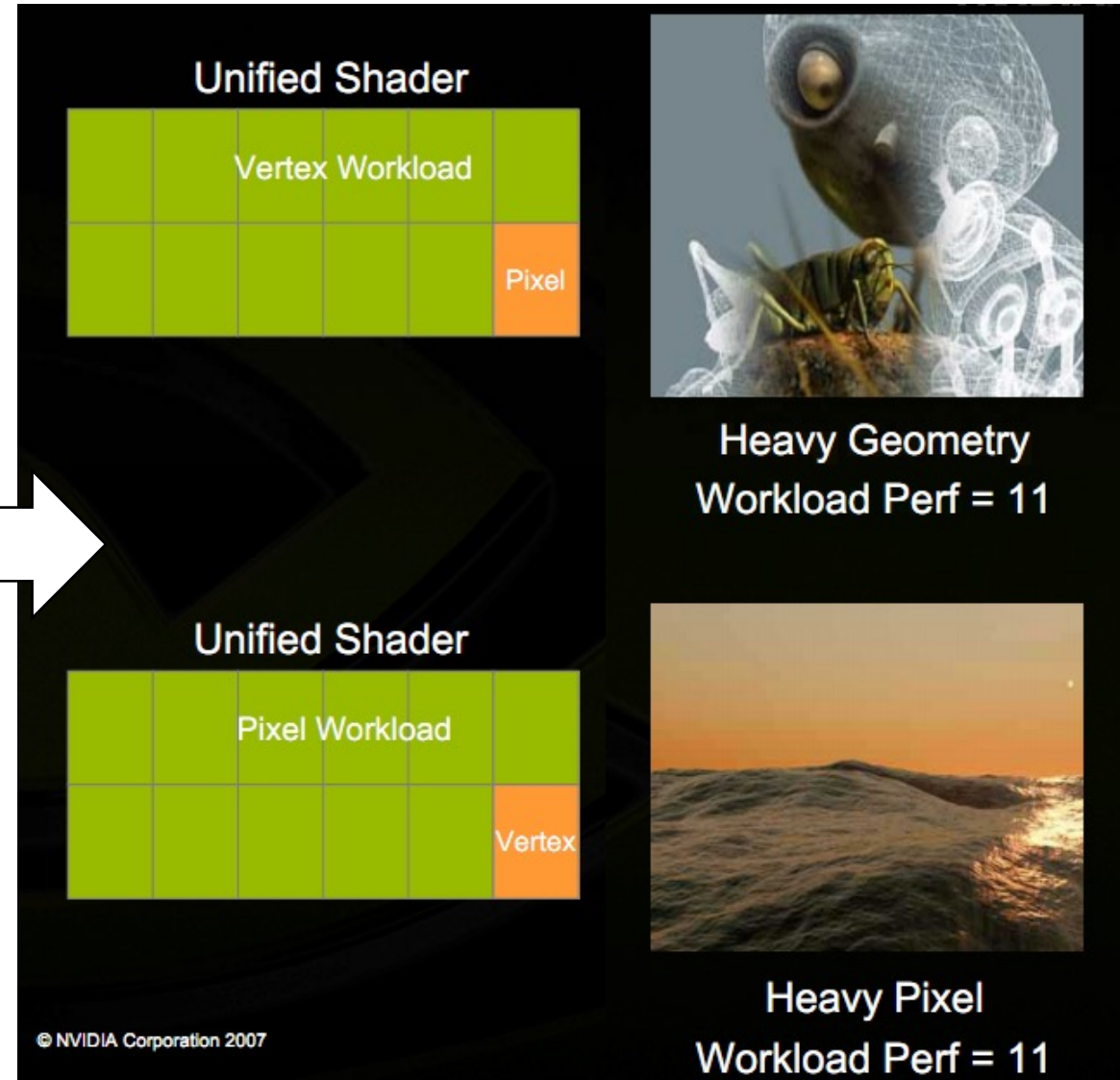
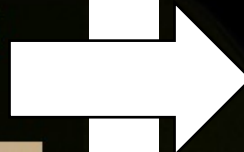
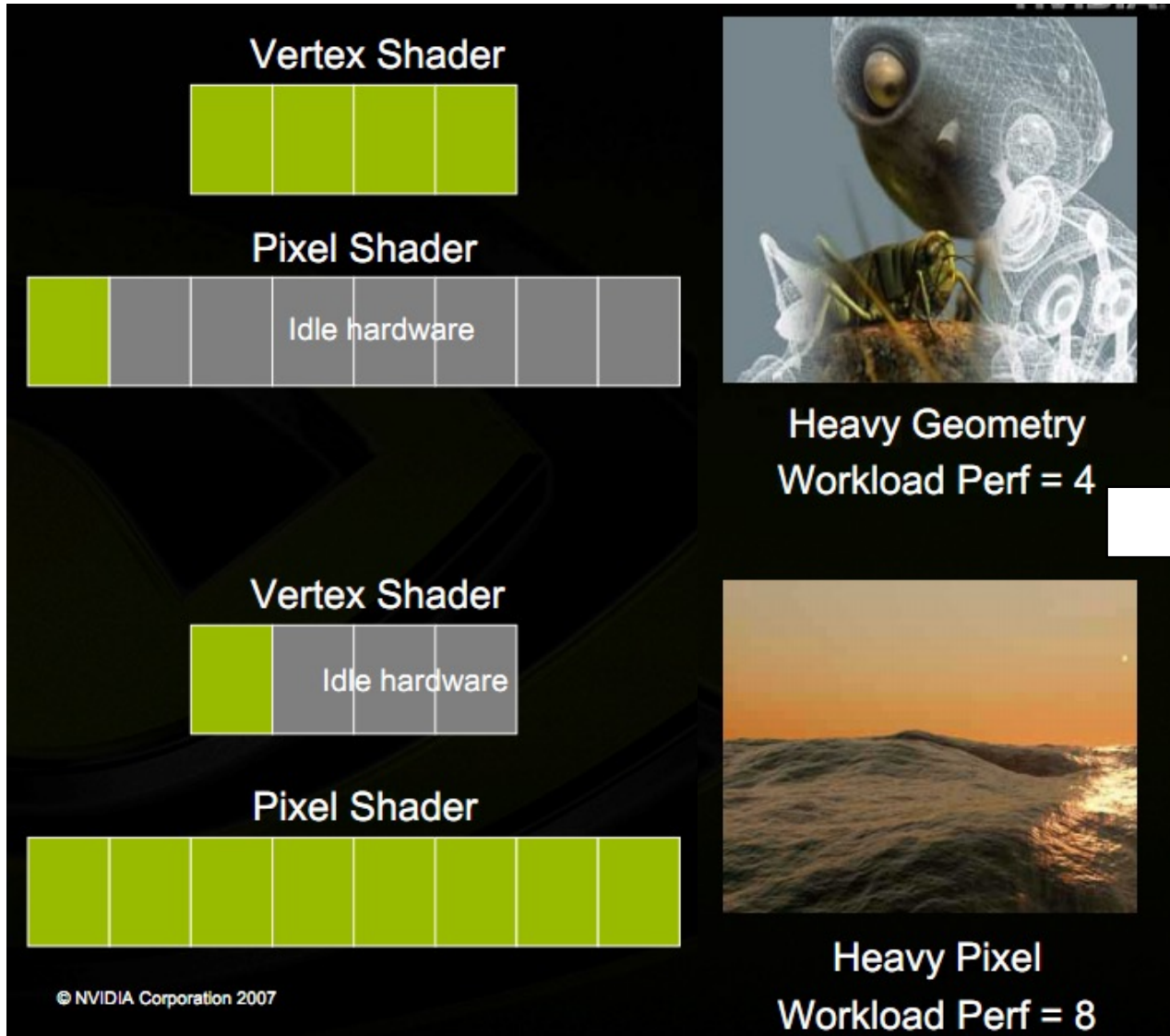
Unified Architecture G80 - Graphics Mode



The future of GPUs is programmable processing architecture built around the processor.

Accelerators in HPC


Why Unify Shader Processors?



Accelerators in HPC


Why Unify Shader Processors?

Unified Shader




Vertex Workload

Pixel




Heavy Geometry
Workload Perf = 11

Unified Shader



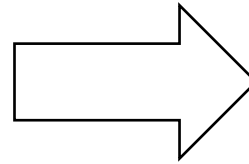
Pixel Workload

Vertex



Heavy Pixel
Workload Perf = 11

© NVIDIA Corporation 2007



Dynamic resource reallocation

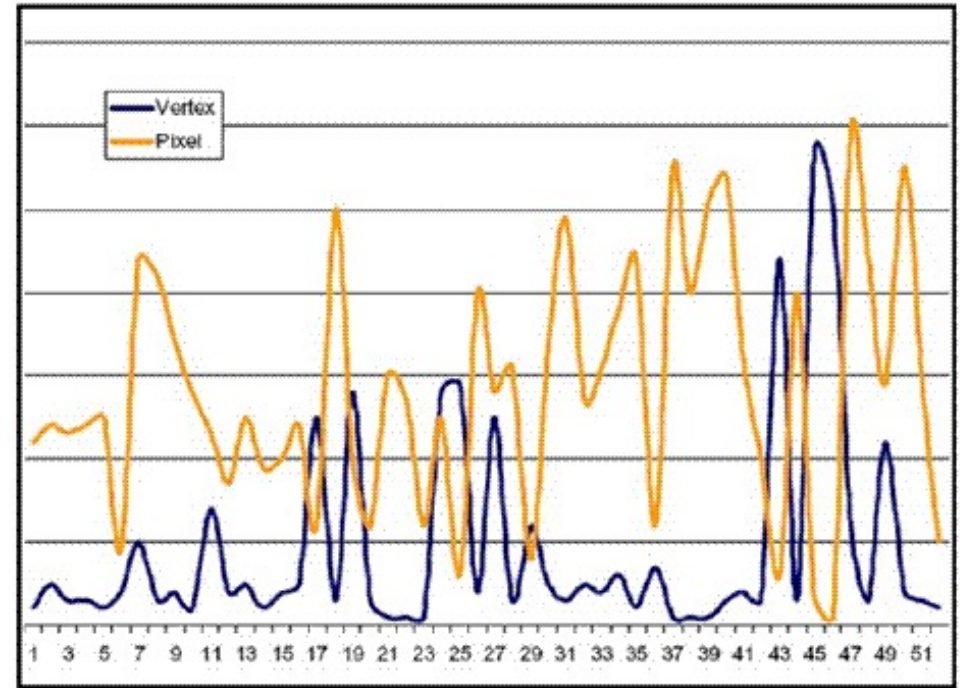
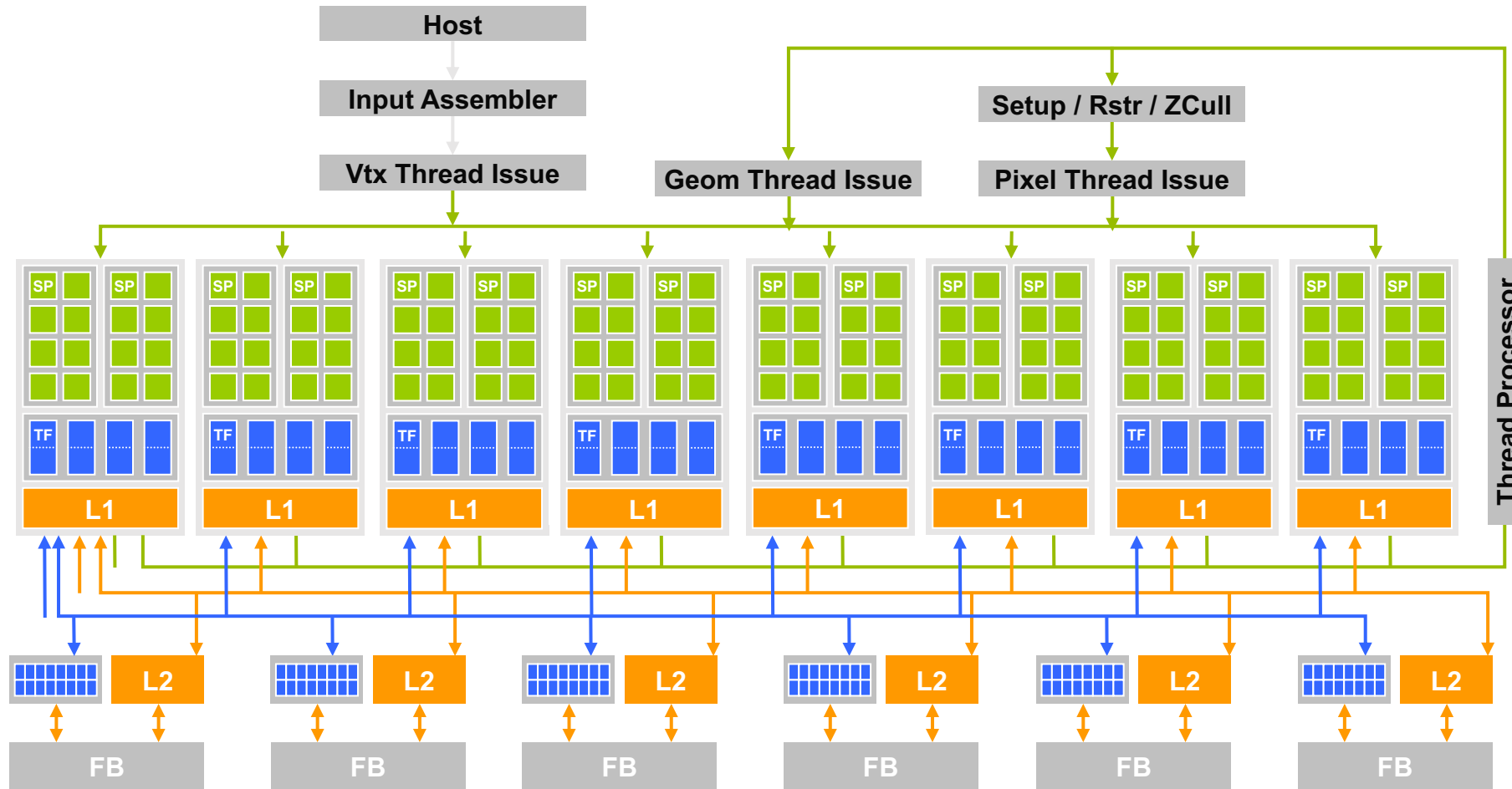


Figure 14. Characteristic pixel and vertex shader workload variation over time

Accelerators in HPC

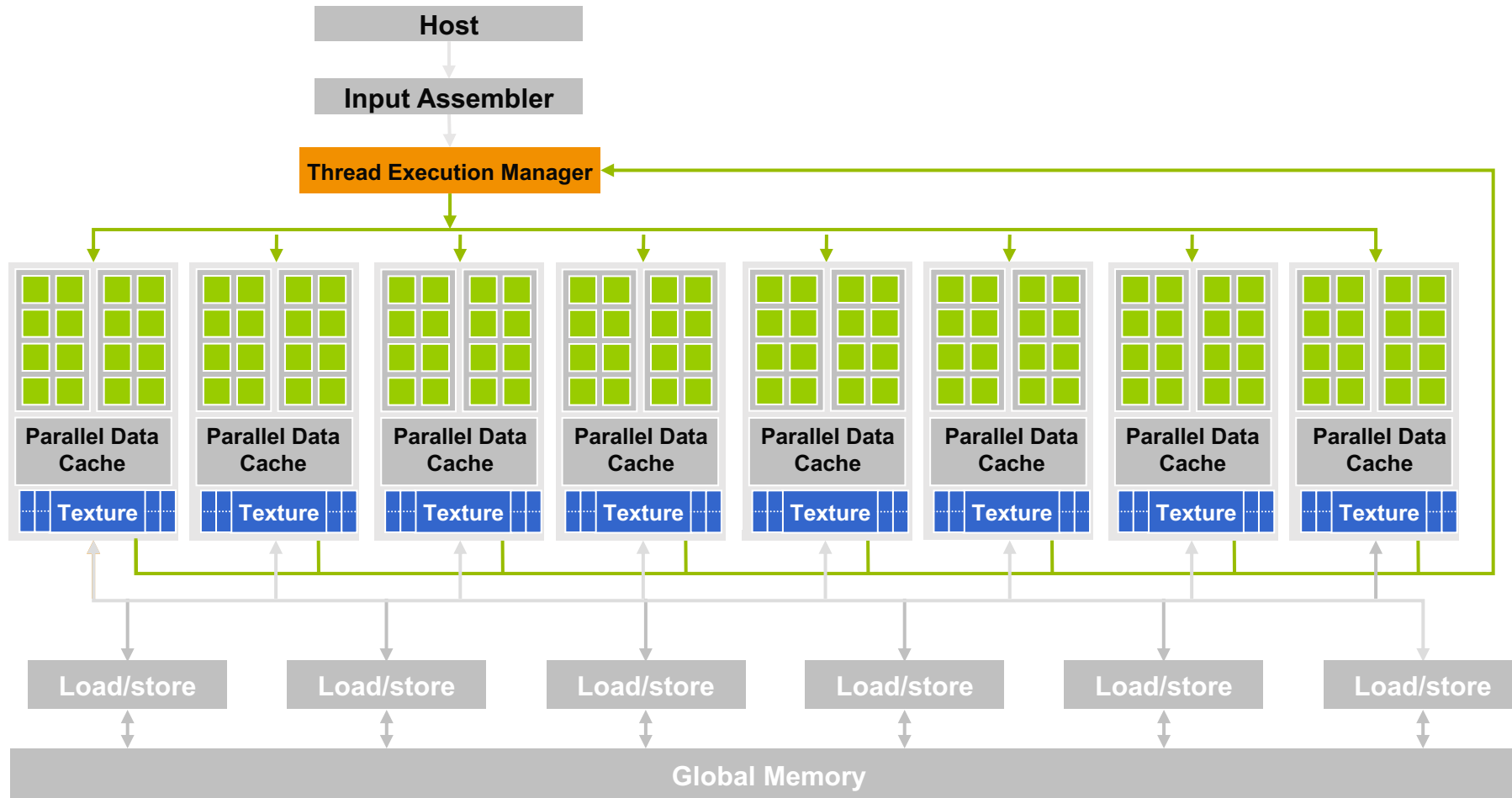
Unified Architecture G80 - Graphics Mode



The future of GPUs is programmable processing architecture built around the processor.

Accelerators in HPC

Unified Architecture G80 - Compute Mode

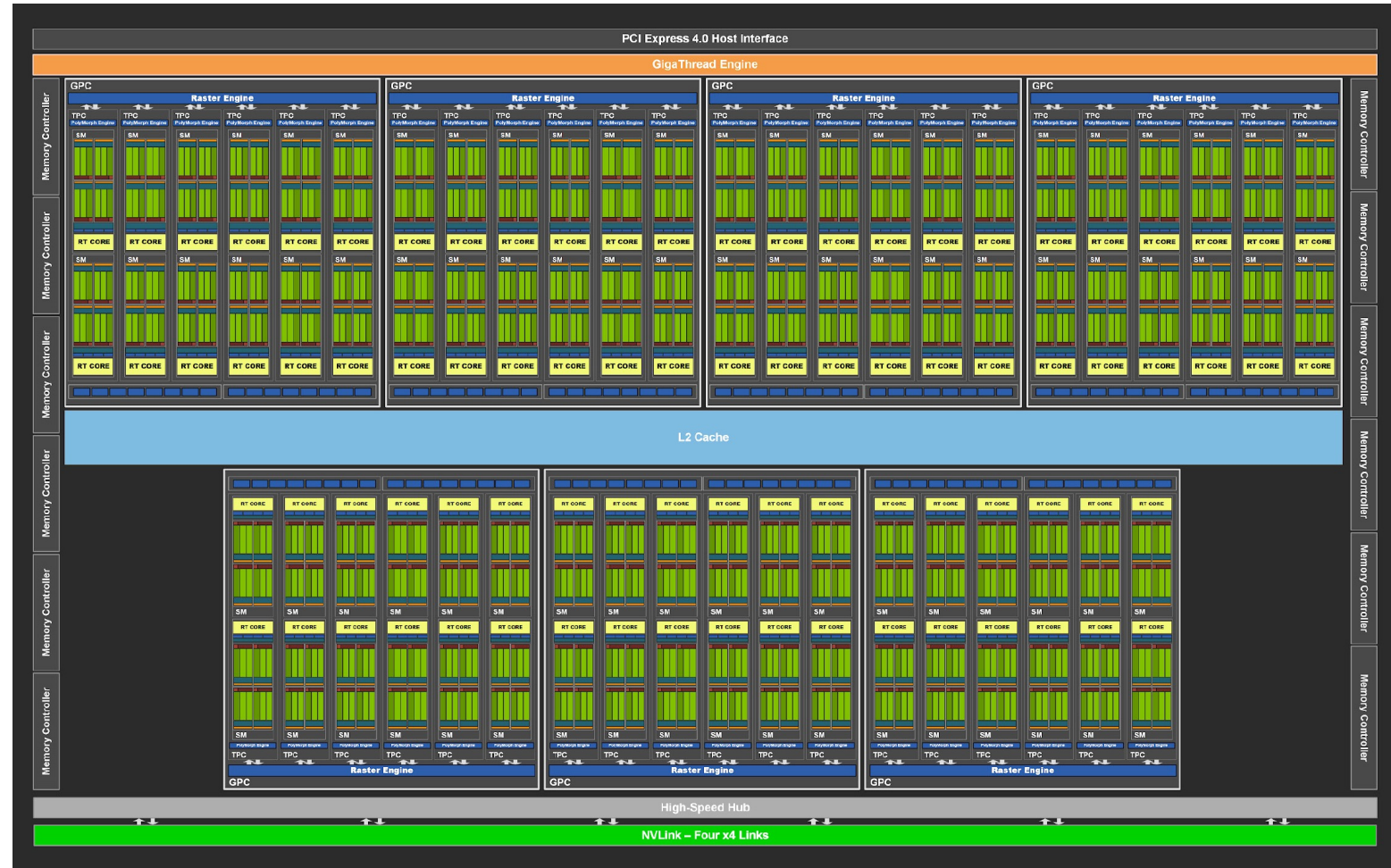


- processors execute computing threads
- new operating mode - HW interface for computing or accelerator

Accelerators in HPC

NVIDIA A40 Architecture

- Based on Ampere architecture GA102 chip designed for 3D graphics rather than scientific computing
 - GA102 GPU also features 168 FP64 units (two per SM),
 - FP64 TFLOP rate is 1/64th the TFLOP rate of FP32 operations.**
 - the small number of FP64 hardware units are included to ensure any programs with FP64 code operate correctly



GA102 Full GPU with 84 SMs

Accelerators in HPC

NVIDIA A40 Architecture

- Based on Ampere architecture GA102 chip designed for 3D graphics rather than scientific computing
 - GA102 GPU also features 168 FP64 units (two per SM),
 - **FP64 TFLOP rate is 1/64th the TFLOP rate of FP32 operations.**
 - the small number of FP64 hardware units are included to ensure any programs with FP64 code operate correctly

SPECIFICATIONS

GPU architecture	NVIDIA Ampere architecture
GPU memory	48 GB GDDR6 with ECC
Memory bandwidth	696 GB/s
Interconnect interface	NVIDIA® NVLink® 112.5 GB/s (bidirectional) ³ PCIe Gen4: 64GB/s
NVIDIA Ampere architecture-based CUDA Cores	10,752
NVIDIA second-generation RT Cores	84
NVIDIA third-generation Tensor Cores	336
Peak FP32 TFLOPS (non-Tensor)	37.4
Peak FP16 Tensor TFLOPS with FP16 Accumulate	149.7 299.4*
Peak TF32 Tensor TFLOPS	74.8 149.6*
RT Core performance TFLOPS	73.1
Peak BF16 Tensor TFLOPS with FP32 Accumulate	149.7 299.4*
Peak INT8 Tensor TOPS Peak INT 4 Tensor TOPS	299.3 598.6* 598.7 1,197.4*

<https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf>

Accelerators in HPC

NVIDIA A40 Architecture

GA10x Streaming Multiprocessor (SM)

- includes four SM processing blocks (also called partitions)
 - 32 FP32 operations per clock, or
 - 16 FP32 and 16 INT32 operations per clock
- In compute mode, the GA10x SM will support the following configurations:
 - 128 KB L1 + 0 KB Shared Memory
 - 120 KB L1 + 8 KB Shared Memory
 - 112 KB L1 + 16 KB Shared Memory
 - 96 KB L1 + 32 KB Shared Memory
 - 64 KB L1 + 64 KB Shared Memory
 - 28 KB L1 + 100 KB Shared Memory



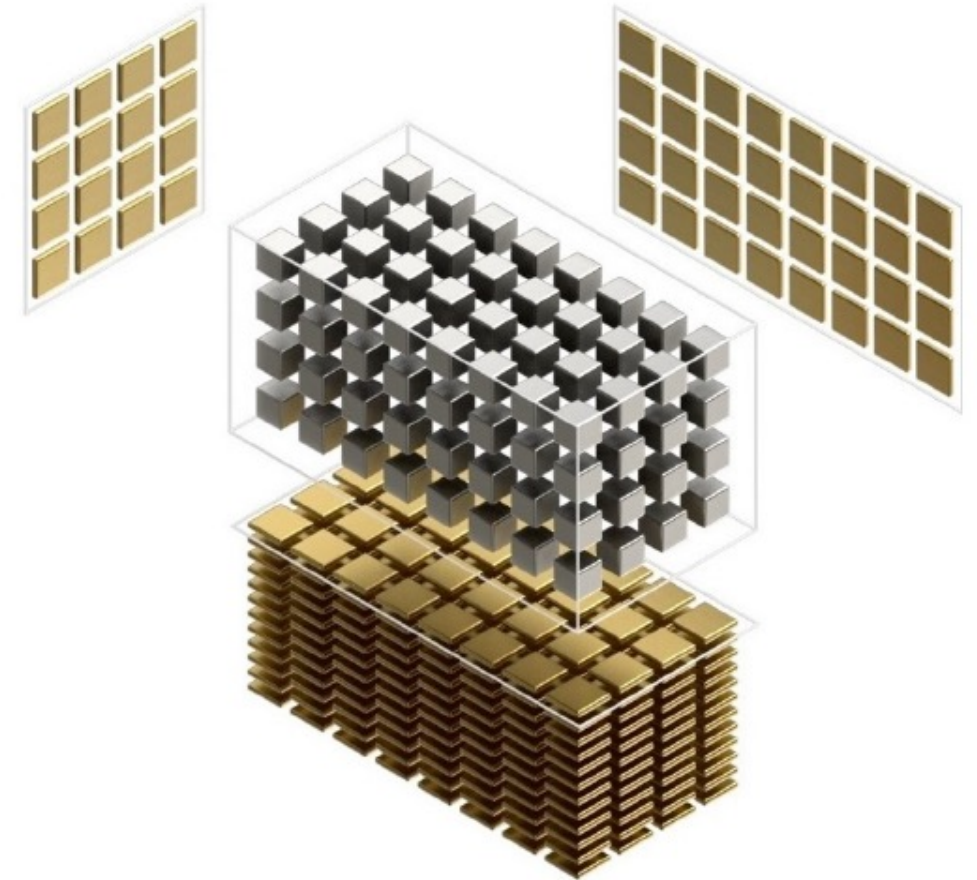
Accelerators in HPC

NVIDIA A40 Architecture

Tensor Cores

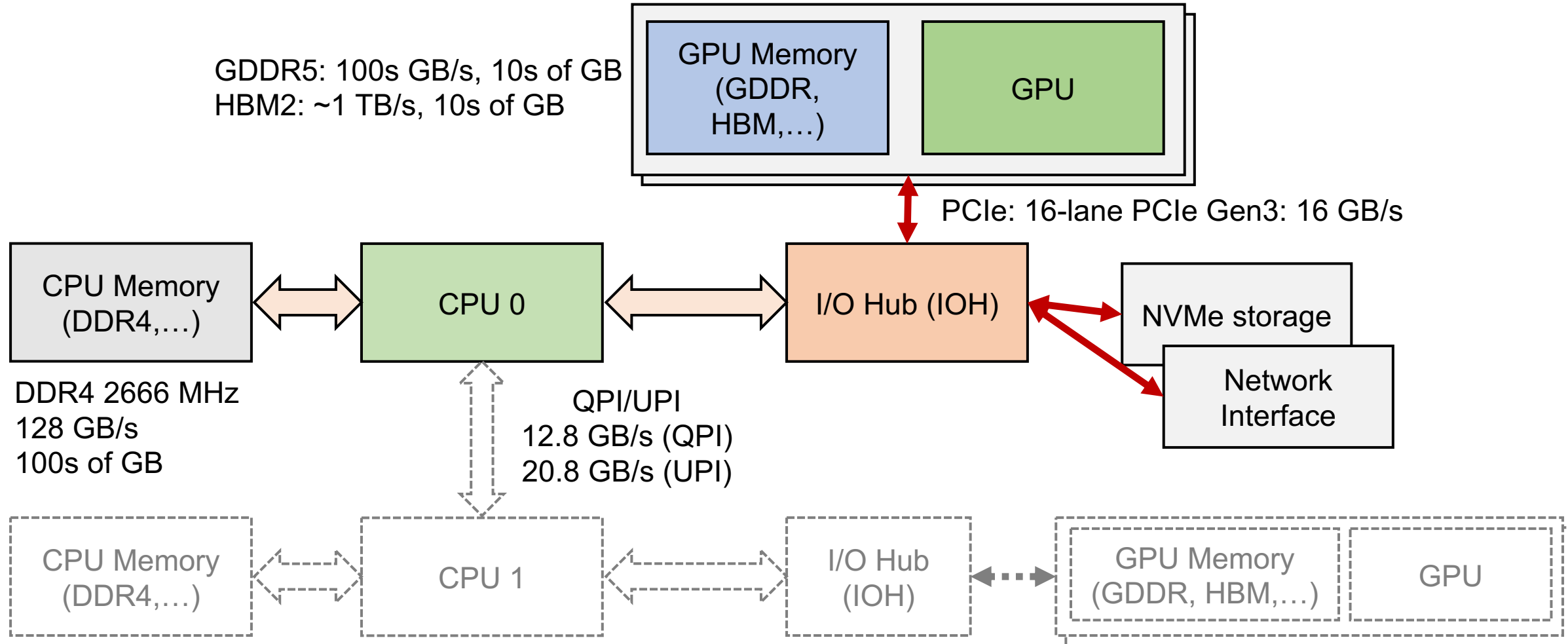
- specialized execution units designed specifically for performing the tensor / matrix operations that are the core compute function used in Deep Learning
- accelerate the matrix-matrix multiplication

GPU Architecture	NVIDIA Ampere
Tensor Cores per SM	4
FP16 FMA operations per Tensor Core	Dense: 128 Sparse: 256
Total FP16 FMA operations per SM	Dense: 512 Sparse: 1024



Ampere architecture tensor core

Architecture of GPU Accelerated Compute Node



Accelerators in HPC

Compute node evaluation

```
$ nvidia-smi topo -m
```

	GPU0	GPU1	m1x5_0	CPU Affinity	NUMA Affinity
GPU0	X	SYS	NODE	0-7,16-23	0
GPU1	SYS	X	SYS	8-15,24-31	1
m1x5_0	NODE	SYS	X		

Legend:

X = Self

SYS = Connection traversing PCIe as well as the SMP interconnect between NUMA nodes (e.g., QPI/UPI)

NODE = Connection traversing PCIe as well as the interconnect between PCIe Host Bridges within a NUMA node

PHB = Connection traversing PCIe as well as a PCIe Host Bridge (typically the CPU)

PXB = Connection traversing multiple PCIe bridges (without traversing the PCIe Host Bridge)

PIX = Connection traversing at most a single PCIe bridge

NV# = Connection traversing a bonded set of # NVLinks

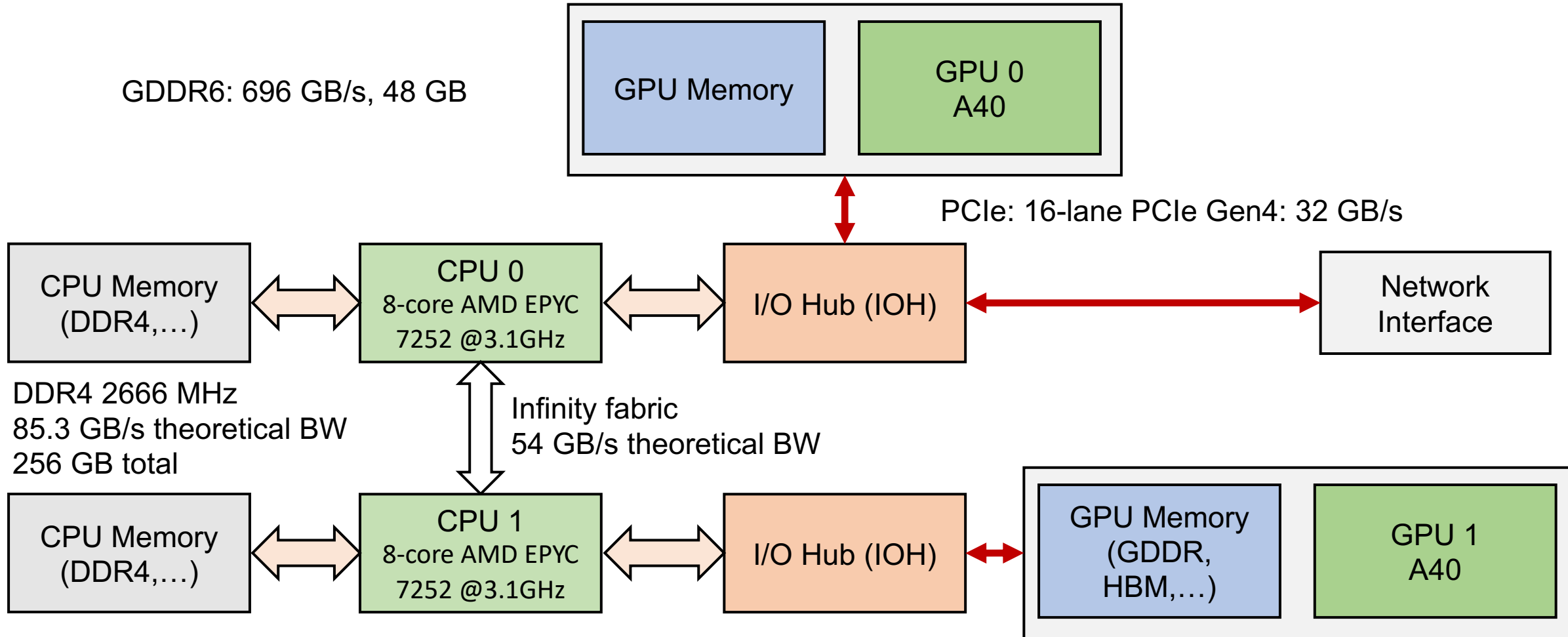
Note:

CPU: 2x 8-core AMD EPYC 7252 @3.1GHz

GPU: 2x NVIDIA A40 GPUs

Accelerators in HPC

Compute node evaluation



Connecting to VSC3 cluster for hands-on exercises

Univerza v Ljubljani



TECHNISCHE
UNIVERSITÄT
WIEN



VSB TECHNICAL
UNIVERSITY
OF OSTRAVA

IT4INNOVATIONS
NATIONAL SUPERCOMPUTING
CENTER



Co-funded by the
Erasmus+ Programme
of the European Union

This project has been funded with support from the European Commission.

This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Accessing the GPU Accelerated Compute Nodes of the Cluster

- ssh to the Vienna Scientific Cluster 3 (VSC3), via a jump host vmos
 - `ssh -t trainee99@vmos.vsc.ac.at vsc3`
 - **Everyone logs in under the same shared user trainee99 – TAKE CARE**
- In Zoom you will be provided a password, enter it TWICE (for vmos and vsc3)
 - If the second prompt for password does not show, `ctrl+C` and try connecting again (might happen multiple times)
- No need to allocate a slurm job, the job is already running
 - you just need to `ssh` to the correct node
- Instructions how to find out your GPU and node will be provided in Zoom
- For Windows users, Putty instructions are on the next slide

```
me@my-home-pc:~$ ssh -t trainee99@vmos.vsc.ac.at vsc3
trainee99@vmos.vsc.ac.at's password: <the_password>
...some-stuff-you-can-ignore...
Password: <the_password>
[trainee99@131 ~]$
```

- Connecting to VSC3 on Windows using Putty
 - Download at <https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html>
 - HostName: vmos.vsc.ac.at
 - Port: 22
 - Connection type: SSH
 - Left menu --> SSH --> Remote command: vsc3
 - Open --> in terminal - login as: trainee99
- After that, everything is the same as on the previous slides

- For future, I recommend checking out WSL2 (Windows subsystem for Linux)

- Hands-on sources also available at https://code.it4i.cz/training/sc_train_2
- Editing the source code files to complete the tasks:
- vim, emacs, nano, ..., directly on VSC3
- **Visual Studio Code and Remote SSH extension**
 - Use `ssh -J trainee99@vmos.vsc.ac.at trainee99@vsc3.vsc.ac.at` command line for connecting
- Edit files locally on your PC, then `scp` or `rsync` to VSC3 (replace 123 with your ID)
 - `scp -o "ProxyJump trainee99@vmos.vsc.ac.at" my_file.txt trainee99@vsc3.vsc.ac.at:~/my_home_dir/CUDA/path/`
 - `rsync -r -e "ssh -J trainee99@vmos.vsc.ac.at" . trainee99@vsc3.vsc.ac.at:~/my_home_dir/CUDA/`
 - When you are located inside the folder cloned from the git
- Again, enter the password twice when connecting
 - In case the second prompt for password to VSC3 does not show, cancel and try again
 - Same weird behavior as described on previous slide
- **Again, everyone is logged in under the same user, so BE CAREFUL**

Hands on: Benchmark Hardware Properties

Univerza v Ljubljani



TECHNISCHE
UNIVERSITÄT
WIEN



VSB TECHNICAL
UNIVERSITY
OF OSTRAVA

IT4INNOVATIONS
NATIONAL SUPERCOMPUTING
CENTER



Co-funded by the
Erasmus+ Programme
of the European Union

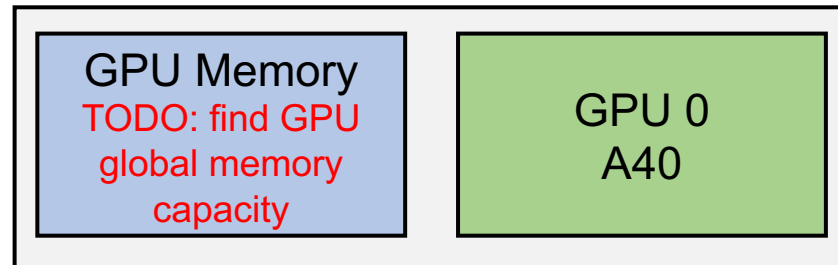
This project has been funded with support from the European Commission.

This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

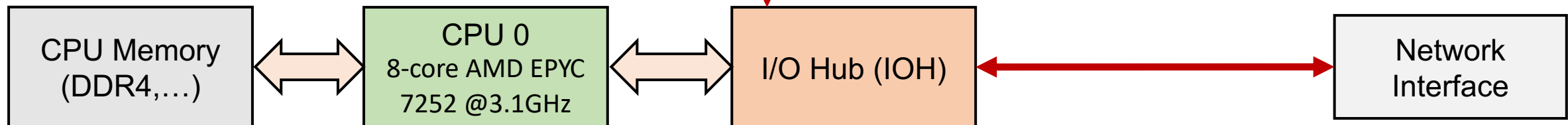
- `cd 00_gpu_info`
- Run the following benchmarks and complete the TODO values on next 2 slides
- Note: there are two participants on each node, if both run the benchmark at the same time, performance might be lower
- Retrieve information about the available GPUs, find global memory capacity
 - `./run_1_device_query.sh`
- Measure CPU memory (RAM) bandwidth
 - `./run_2_memory_bw_cpu.sh`
- Measure GPU memory bandwidth, compare it with CPU memory bandwidth
 - `./run_3_memory_bw_gpu.sh`
- Measure CPU-GPU data transfer bandwidth
 - `./run_4_copy_bw_cpu_gpu.sh`
- Measure GPU-GPU data transfer bandwidth, compare with CPU-GPU data transfer bandwidth
 - `./run_5_copy_bw_gpu_gpu.sh`

Hands on Benchmark Hardware Properties

GDDR6: Theoretical bandwidth: 696 GB/s
Benchmark: BabelStream
TODO: Measure global mem. bandwidth

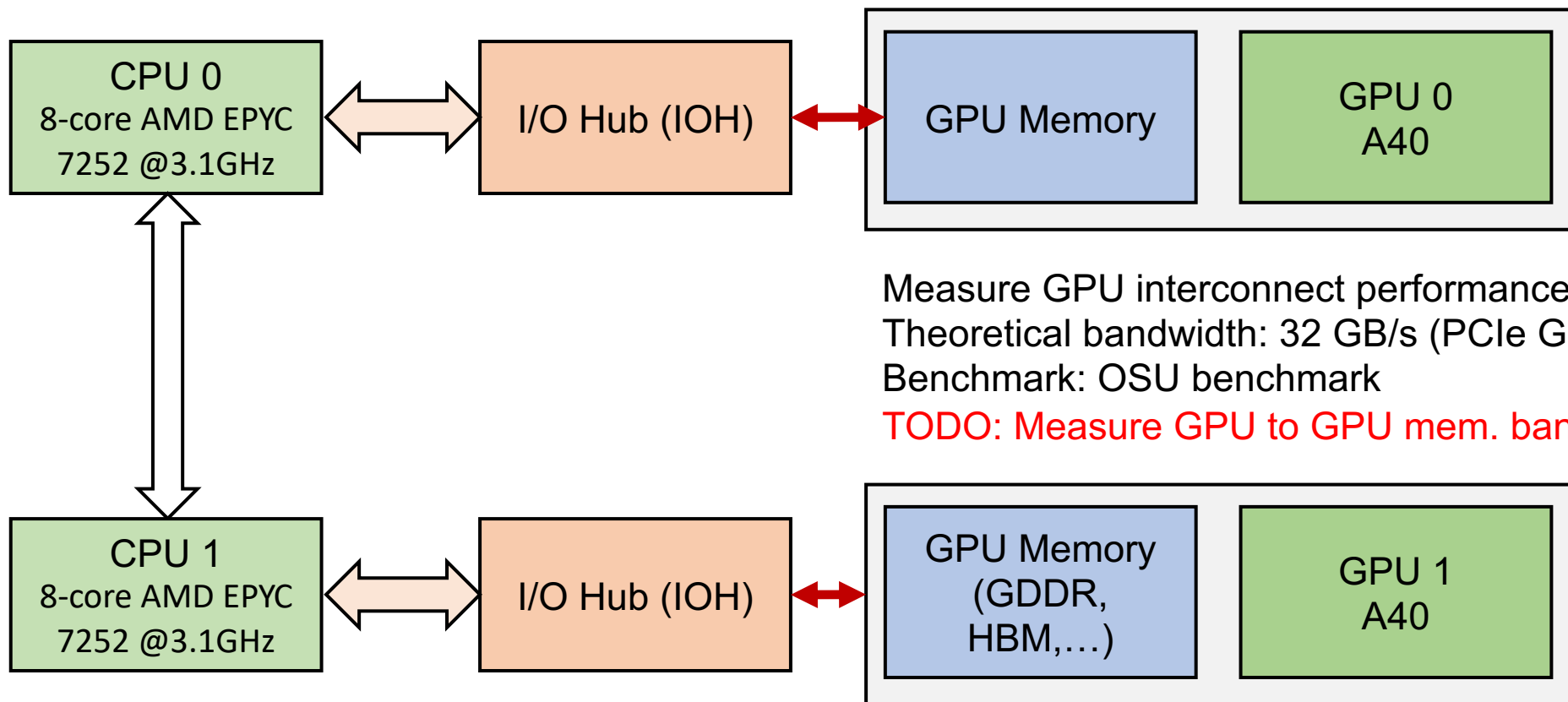


PCIe: 16-lane PCIe Gen4: 32 GB/s theoretical bandwidth
Benchmark: bandwidthTest from CUDA samples
TODO: Measure PCIe bandwidth



Theoretical 85.3 GB/s
Benchmark: STREAM benchmark
TODO: Measure CPU memory bandwidth

Hands on Benchmark Hardware Properties



Measure GPU interconnect performance
Theoretical bandwidth: 32 GB/s (PCIe Gen4)
Benchmark: OSU benchmark

TODO: Measure GPU to GPU mem. bandwidth

Hands on – solution, output Benchmark Hardware Properties

```
$ ./run_5_copy_bw_gpu_gpu.sh
...
# OSU MPI-CUDA Bandwidth Test
# Send Buffer on DEVICE (D) and Receive Buffer on DEVICE (D)
# Size          Bandwidth (MB/s)
1024             188.05
2048             394.65
4096             817.17
8192             1732.43
16384            3551.33
32768            6935.73
65536            12408.54
131072           17182.50
262144           20774.41
524288           23185.55
1048576          24647.77
2097152          25442.60
4194304          25857.67
8388608          26051.70
16777216         26170.93
33554432         26231.38
67108864         26260.76
```

```
$ ./run_4_copy_bw_cpu_gpu.sh
...
Host to Device Bandwidth, 1 Device(s)
Pinned Memory Transfers
Transfer Size (Bytes)          Bandwidth(GB/s)
32000000                       26.2

Device to Host Bandwidth, 1 Device(s)
Pinned Memory Transfers
Transfer Size (Bytes)          Bandwidth(GB/s)
32000000                       25.7
```

```
$ ./run_3_memory_bw_gpu.sh
...
Function      MBytes/sec  Min (sec)  Max      Average
Copy          580629.471  0.00185   0.00187  0.00186
Mul           579703.149  0.00185   0.00188  0.00186
Add           584039.499  0.00276   0.00279  0.00277
Triad         584855.371  0.00275   0.00278  0.00276
Dot           572203.933  0.00188   0.00189  0.00188
```

```
$ ./run_2_memory_bw_cpu.sh
...
-----
Function  Best Rate MB/s  Avg time  Min time  Max time
Copy:    79567.2        0.050298  0.050272  0.050332
Scale:   53722.3        0.074535  0.074457  0.074618
Add:     57829.1        0.103843  0.103754  0.103920
Triad:   57774.5        0.103947  0.103852  0.104022
-----
Solution Validates: avg error less than 1.000000e-13 on all three arrays
-----
```

```
$ ./run_1_device_query.sh
...
Detected 2 CUDA Capable device(s)

Device 0: "A40"
  CUDA Driver Version / Runtime Version      11.2 / 11.4
  CUDA Capability Major/Minor version number: 8.6
  Total amount of global memory:            45634 MBytes (47850782720 bytes)
  (084) Multiprocessors, (128) CUDA Cores/MP: 10752 CUDA Cores
  GPU Max Clock rate:                       1740 MHz (1.74 GHz)
  Memory Clock rate:                         7251 Mhz
  Memory Bus Width:                          384-bit
  L2 Cache Size:                             6291456 bytes
  Maximum Texture Dimension Size (x,y,z)    1D=(131072), 2D=(131072, 65536),
  3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers 1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers 2D=(32768, 32768), 2048 layers
  Total amount of constant memory:          65536 bytes
  Total amount of shared memory per block:   49152 bytes
  Total shared memory per multiprocessor:    102400 bytes
  Total number of registers available per block: 65536
  Warp size:                                 32
  Maximum number of threads per multiprocessor: 1536
  Maximum number of threads per block:      1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                     2147483647 bytes
  Texture alignment:                         512 bytes
  Concurrent copy and kernel execution:      Yes with 2 copy engine(s)
  Run time limit on kernels:                 No
  Integrated GPU sharing Host Memory:        No
  Support host page-locked memory mapping:   Yes
  Alignment requirement for Surfaces:        Yes
  Device has ECC support:                    Enabled
  Device supports Unified Addressing (UVA):   Yes
  Device supports Managed Memory:            Yes
  Device supports Compute Preemption:        Yes
  Supports Cooperative Kernel Launch:        Yes
  Supports MultiDevice Co-op Kernel Launch:  Yes
  Device PCI Domain ID / Bus ID / location ID: 0 / 65 / 0
  Compute Mode:
    < Default (multiple host threads can use ::cudaSetDevice() with device
    simultaneously) >

Device 1: "A40"
...
> Peer access from A40 (GPU0) -> A40 (GPU1) : Yes
> Peer access from A40 (GPU1) -> A40 (GPU0) : Yes
```

CUDA Programming

Univerza v Ljubljani



TECHNISCHE
UNIVERSITÄT
WIEN



VSB TECHNICAL
UNIVERSITY
OF OSTRAVA

IT4INNOVATIONS
NATIONAL SUPERCOMPUTING
CENTER



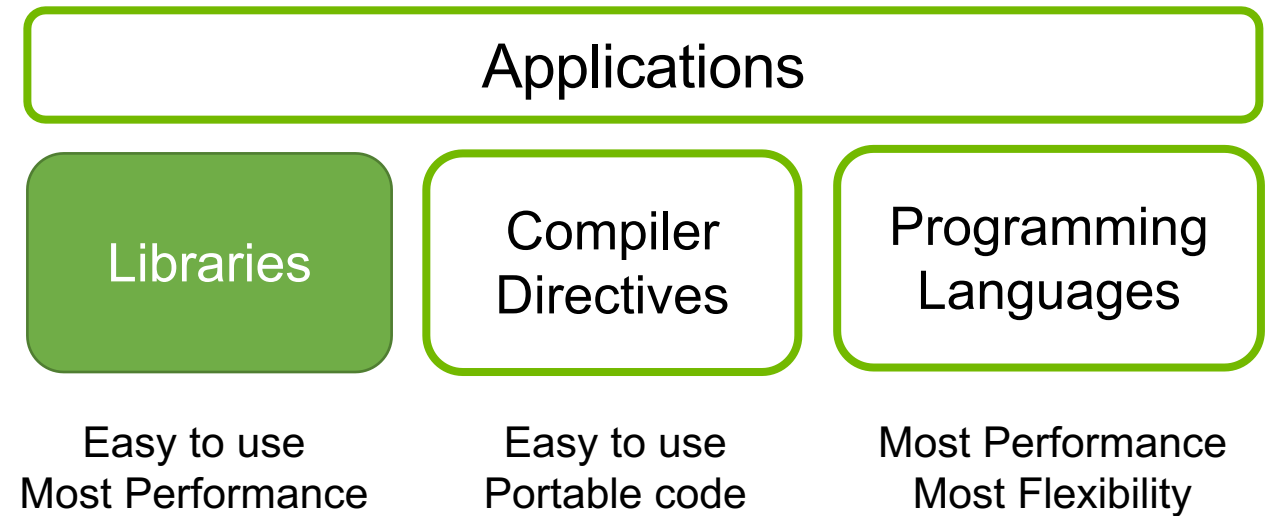
Co-funded by the
Erasmus+ Programme
of the European Union

This project has been funded with support from the European Commission.

This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Libraries

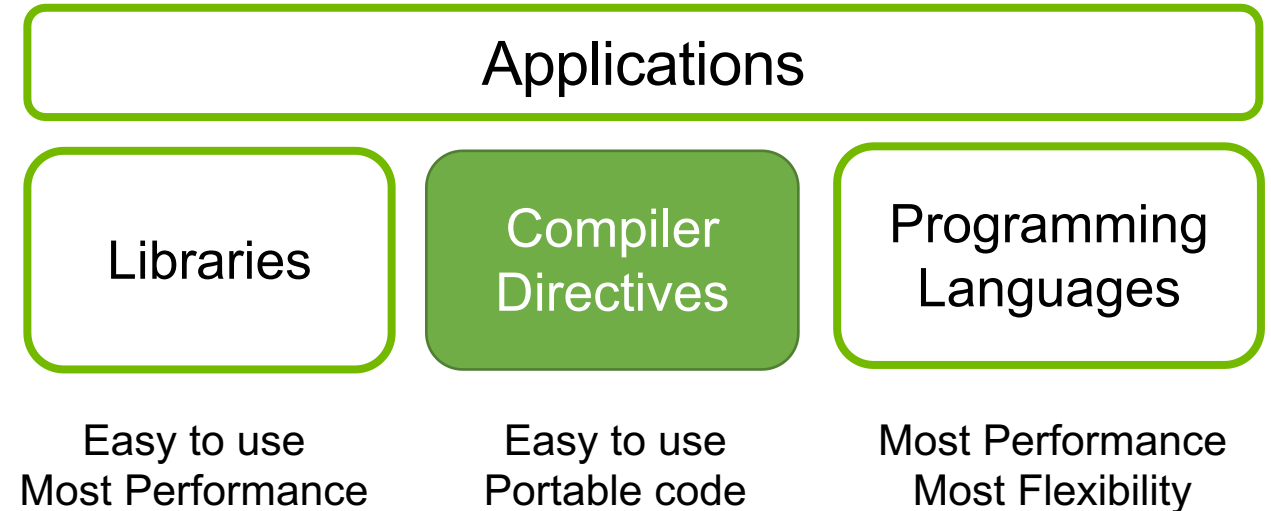
- ease of use:
 - enables GPU acceleration without any GPU programming
- drop-in:
 - follow standard APIs
 - minimal code changes
- quality:
 - high-quality implementations



Compiler Directives

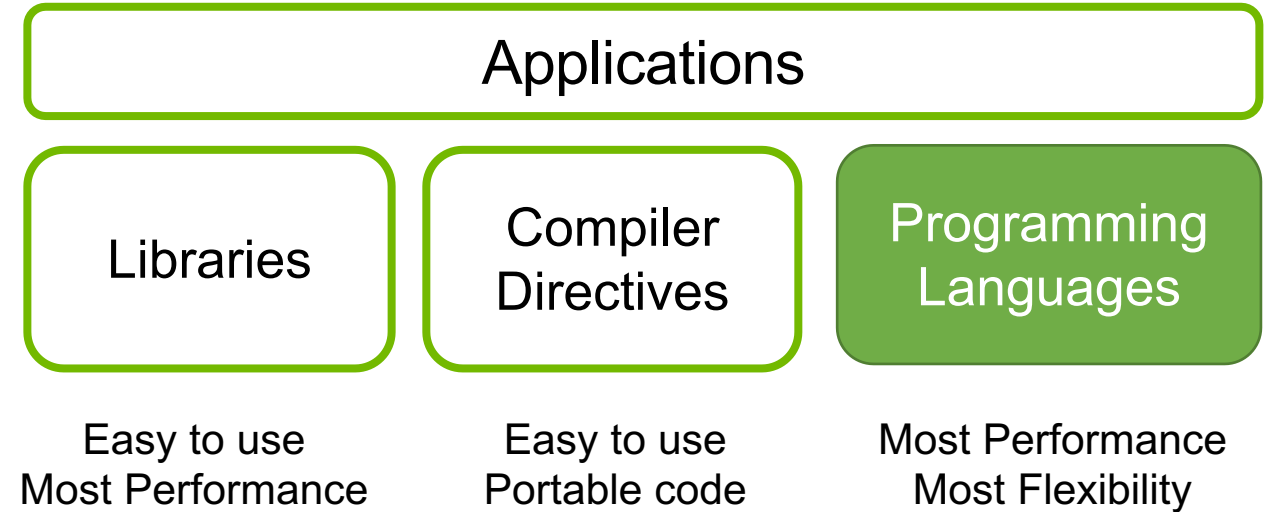
- ease of use
 - compiler takes care of details of parallelism management and data movement
- portable
 - code is generic, not specific to any type of hardware
- Example: OpenACC
 - Compiler directives for C, C++, and FORTRAN

```
#pragma acc parallel loop  
copyin(input1[0:inputLength], input2[0:inputLength]),  
copyout(output[0:inputLength])  
for(i = 0; i < inputLength; ++i) {  
    output[i] = input1[i] + input2[i];  
}
```



Programming Languages

- Performance: best control of parallelism and data movement
- Flexible: the computation does not need to fit into a limited set of library patterns or directives
- Complex: programmer often needs to express more details



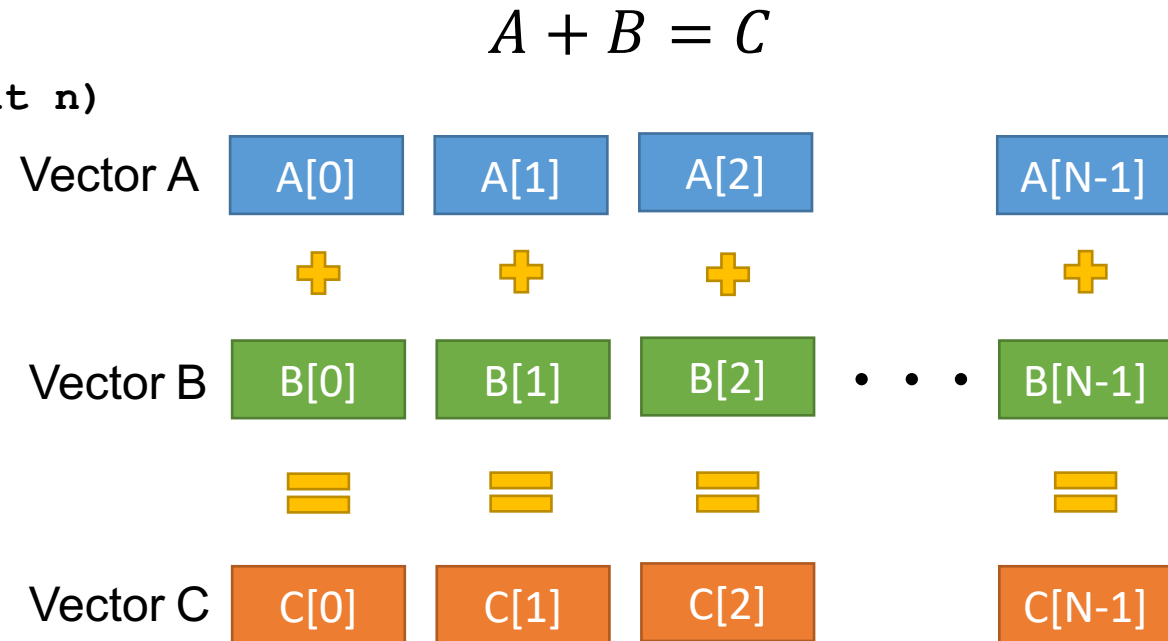
GPU Programming Languages

Numerical analytics	MATLAB, Mathematica	C	CUDA C, OpenACC
Python	PyCUDA, Numba	C++	CUDA C++, Thrust
Fortran	CUDA Fortran, OpenACC	C#	Hybridizer

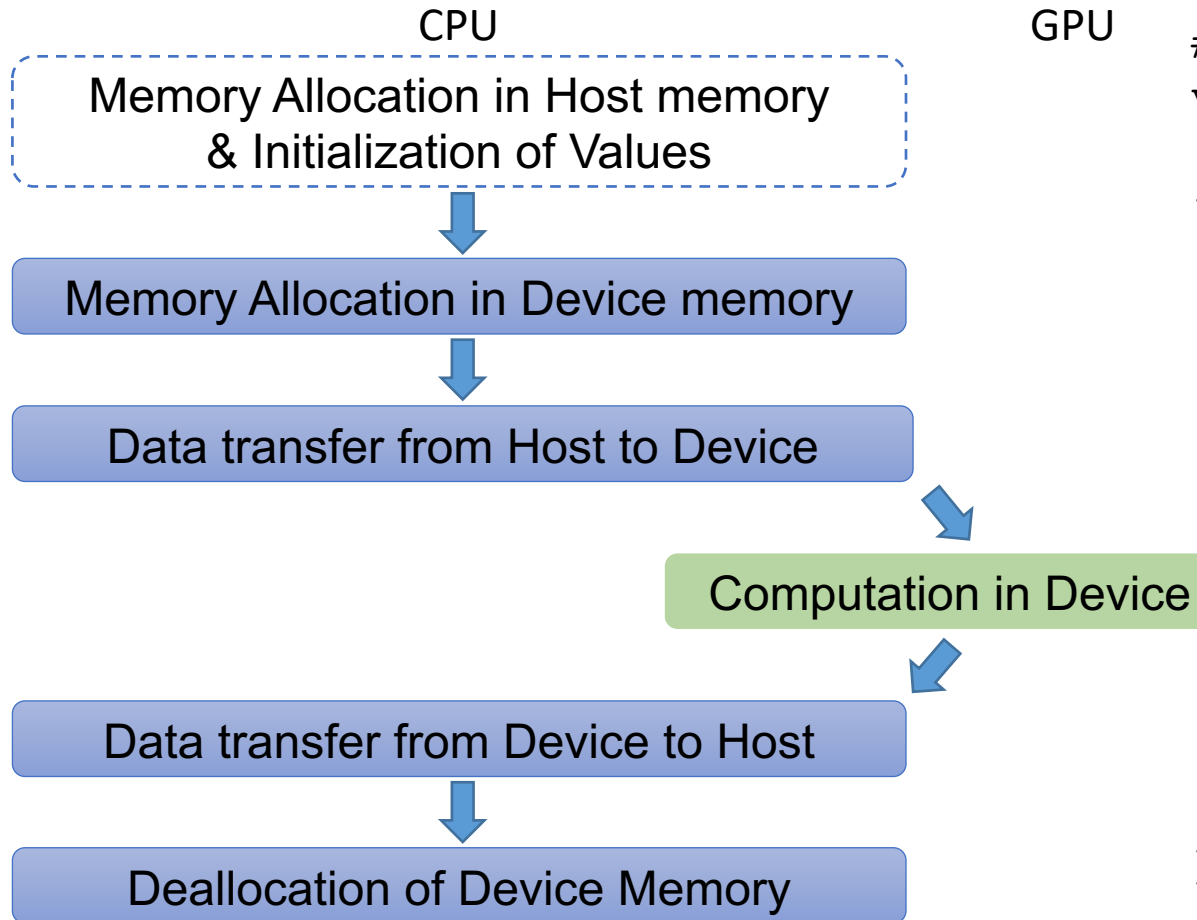
Vector addition example

```
// Compute vector sum C = A + B
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int i;
    for (i = 0; i < n; i++)
        h_C[i] = h_A[i] + h_B[i];
}

int main()
{
    // Memory allocation for h_A, h_B, and h_C
    // read h_A and h_B from file for N elements
    ...
    vecAdd(h_A, h_B, h_C, N);
}
```



CUDA programming Heterogenous Program



GPU

```
#include <cuda.h>
void vecAdd(float *h_A, float *h_B, float *h_C,
            int n)
{
    int size = n* sizeof(float);
    float *d_A, *d_B, *d_C;

    // allocate device memory for A, B, and C
    // copy A and B to device memory

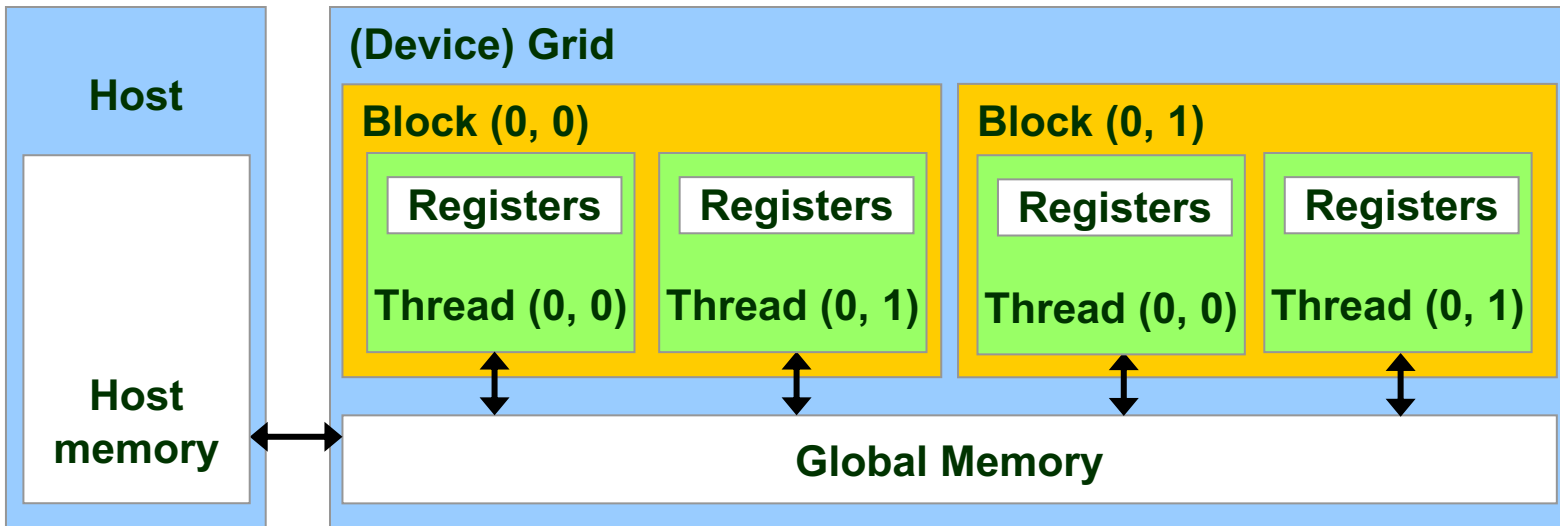
    // kernel launch code
    // – GPU performs the actual vector addition

    // copy C from the device memory

    // Free device vectors
}
```

CUDA programming

Partial Overview of CUDA Memories



Device code (kernel) can:

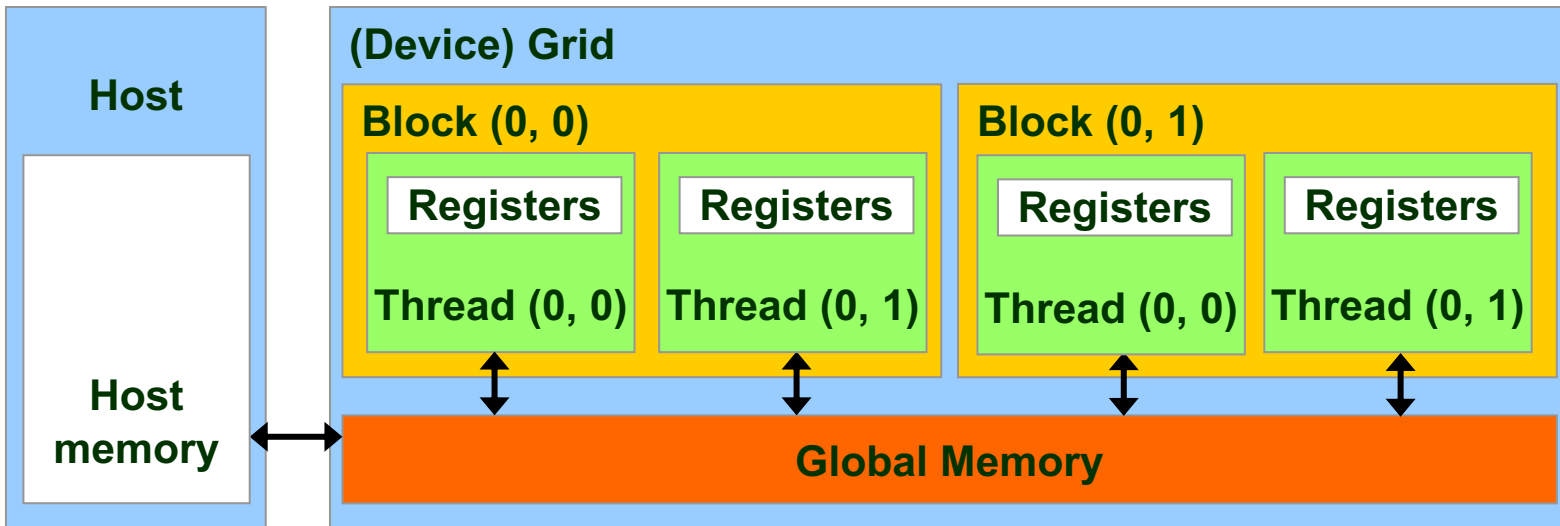
- R/W per-thread **registers**
- R/W all-shared **global memory**

Host code can

- Transfer data to/from per grid **global memory**

CUDA programming

Partial Overview of CUDA Memories



cudaMalloc()

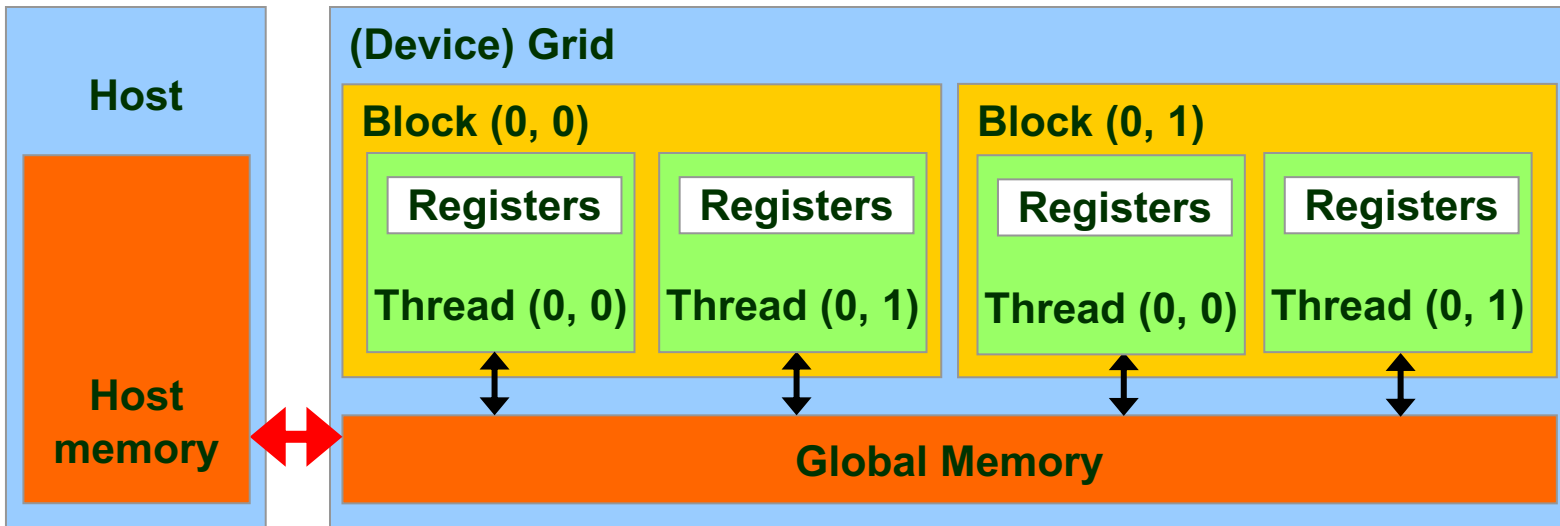
- Allocates an object in the device global memory
- Two parameters
 - Address of a pointer to the allocated object
 - Size of allocated object in terms of bytes

cudaFree()

- Frees object from device global memory
- One parameter
 - Pointer to freed object

CUDA programming

Partial Overview of CUDA Memories



`cudaMemcpy()`

- memory data transfer
- Requires four parameters
 - Pointer to destination
 - Pointer to source
 - Number of bytes copied
 - Type/Direction of transfer
- Transfer to device is synchronous with respect to the host

CPU

Memory Allocation in Host memory
& Initialization of Values

```
int main(){

    float *h_A, *h_B, *h_C;

    int n = 10000000 // size of an array
    int size = n * sizeof(float);

    h_A = (float*)malloc(size);
    h_B = (float*)malloc(size);
    h_C = (float*)malloc(size);

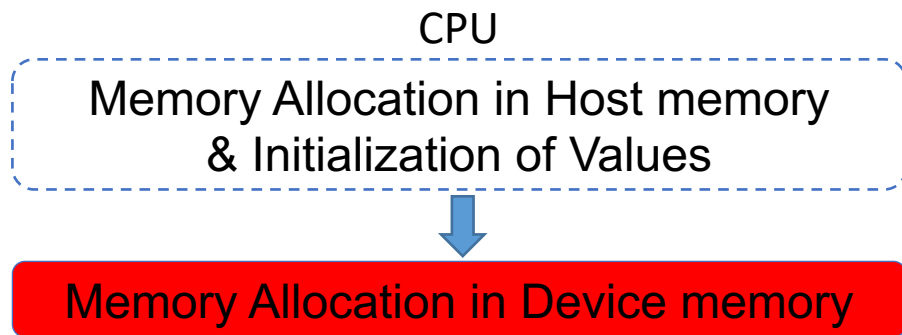
    // Initialize array
    for(int i = 0; i < array_size; i++){
        h_A[i] = 1.0f;
        h_B[i] = 2.0f;}

    vecAdd(h_A, h_B, h_C, n);

    // Deallocate host memory
    free(h_A); free(h_A); free(h_C);
}
```

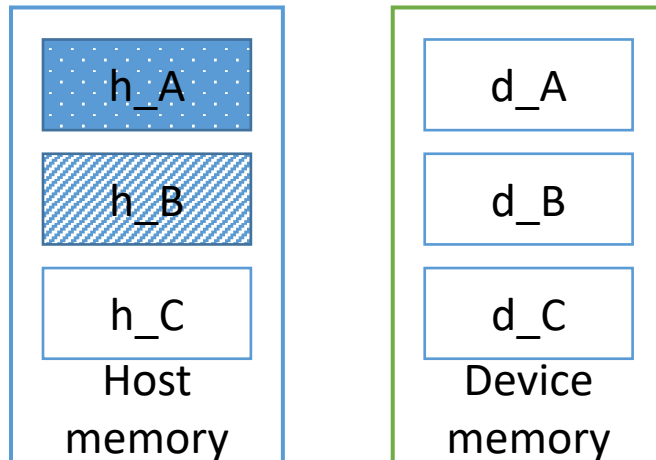
CUDA programming

Explicit Memory Management



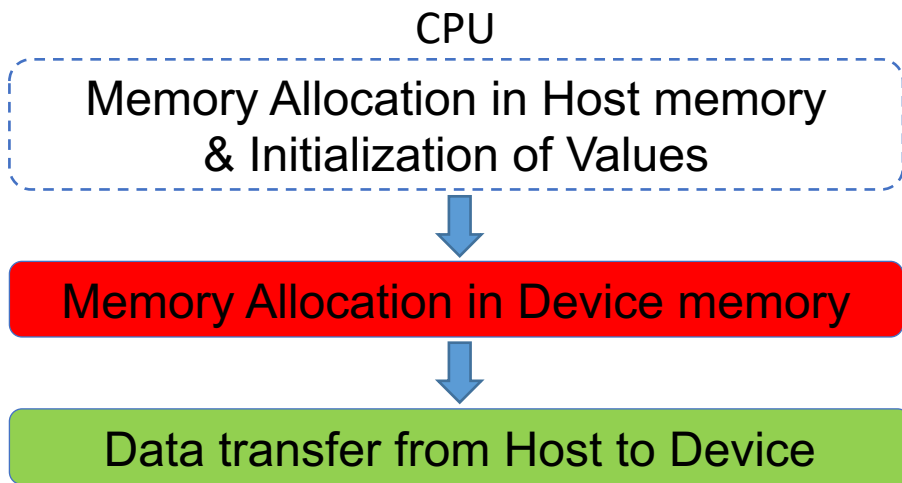
GPU

```
void vecAdd(float *h_A, float *h_B, float *h_C,  
int n)  
{  
    int size = n * sizeof(float);  
    float *d_A, *d_B, *d_C;  
    cudaMalloc((void **) &d_A, size);  
    cudaMalloc((void **) &d_B, size);  
    cudaMalloc((void **) &d_C, size);  
}
```



CUDA programming

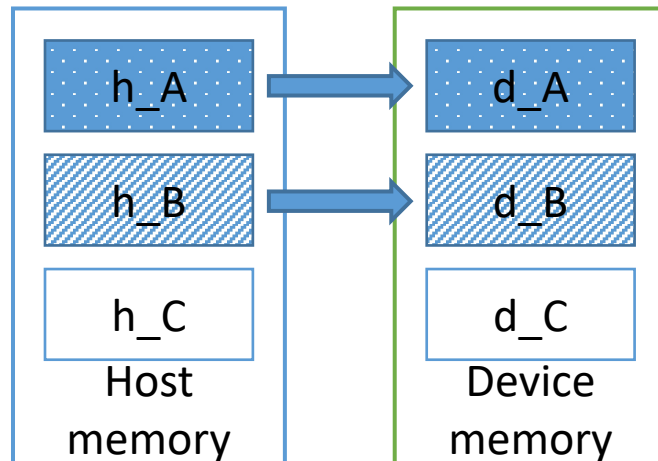
Explicit Memory Management



GPU

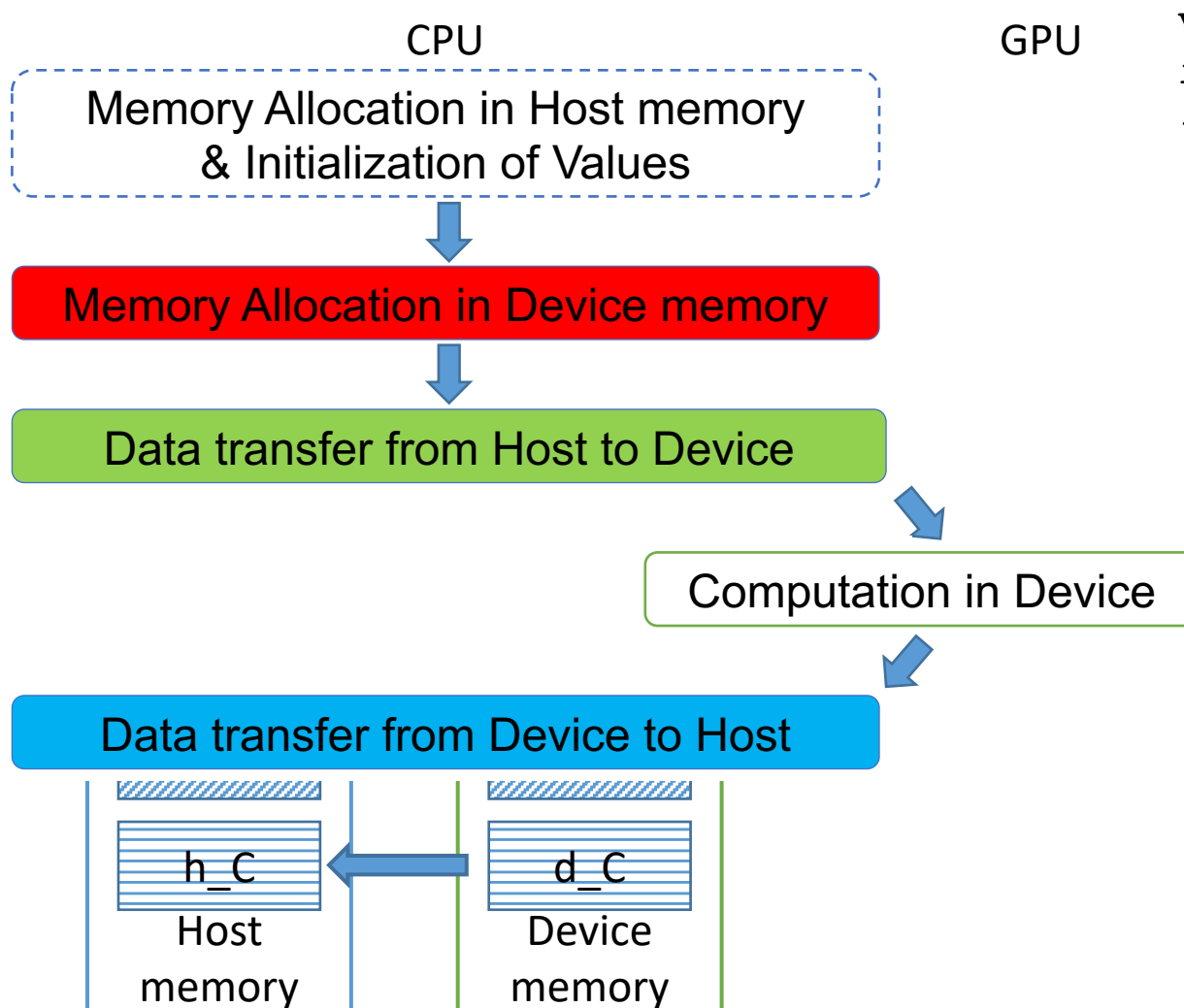
```
void vecAdd(float *h_A, float *h_B, float *h_C,
int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;
    cudaMalloc((void **) &d_A, size);
    cudaMalloc((void **) &d_B, size);
    cudaMalloc((void **) &d_C, size);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
}
```



CUDA programming

Explicit Memory Management



```
void vecAdd(float *h_A, float *h_B, float *h_C,
int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;
    cudaMalloc((void **) &d_A, size);
    cudaMalloc((void **) &d_B, size);
    cudaMalloc((void **) &d_C, size);

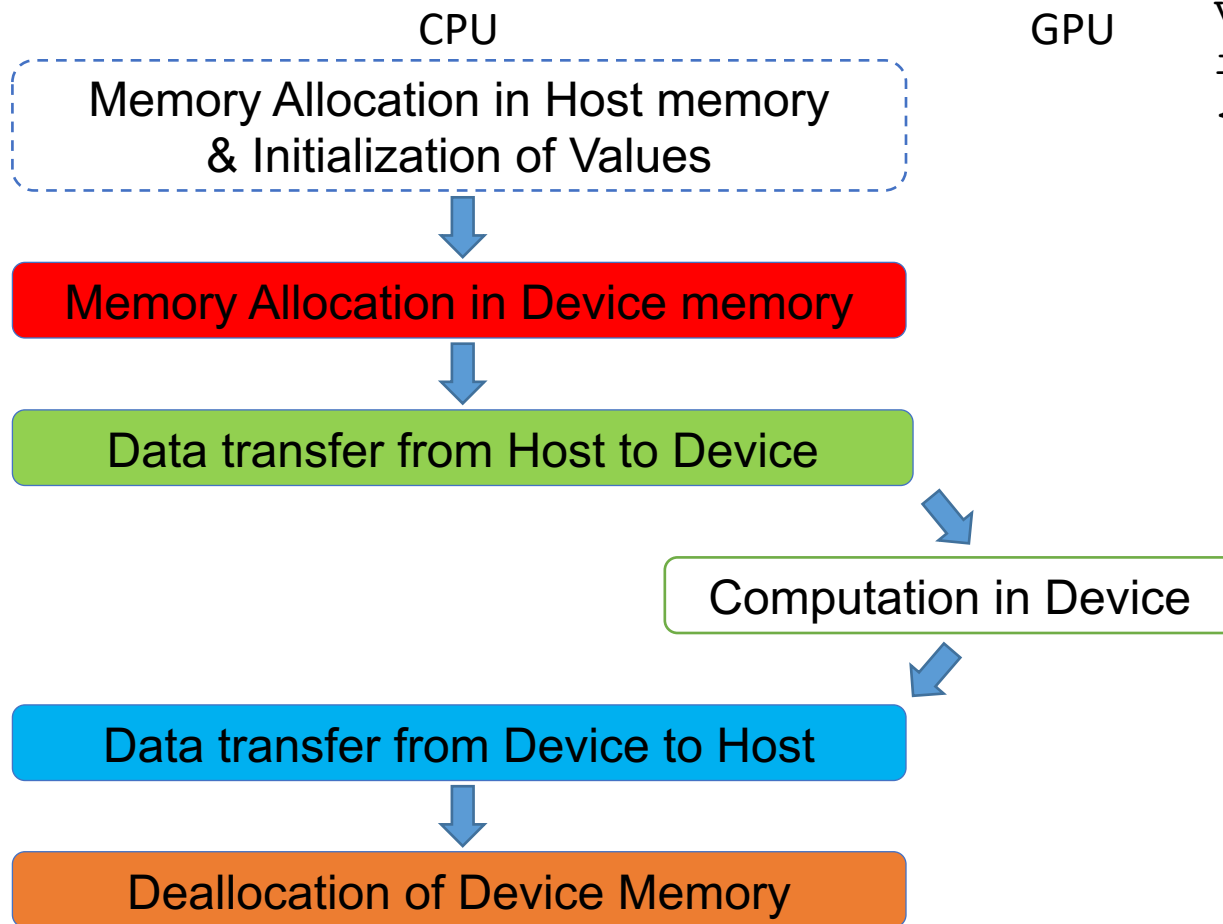
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Kernel run

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
}
```

CUDA programming

Explicit Memory Management



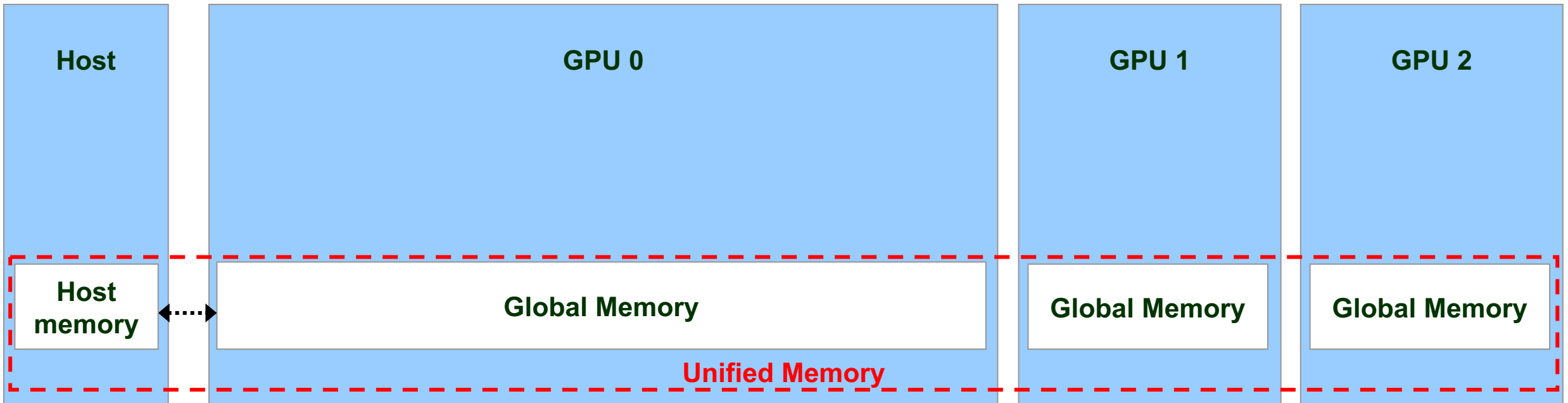
```
void vecAdd(float *h_A, float *h_B, float *h_C,
int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;
    cudaMalloc((void **) &d_A, size);
    cudaMalloc((void **) &d_B, size);
    cudaMalloc((void **) &d_C, size);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Kernel run

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

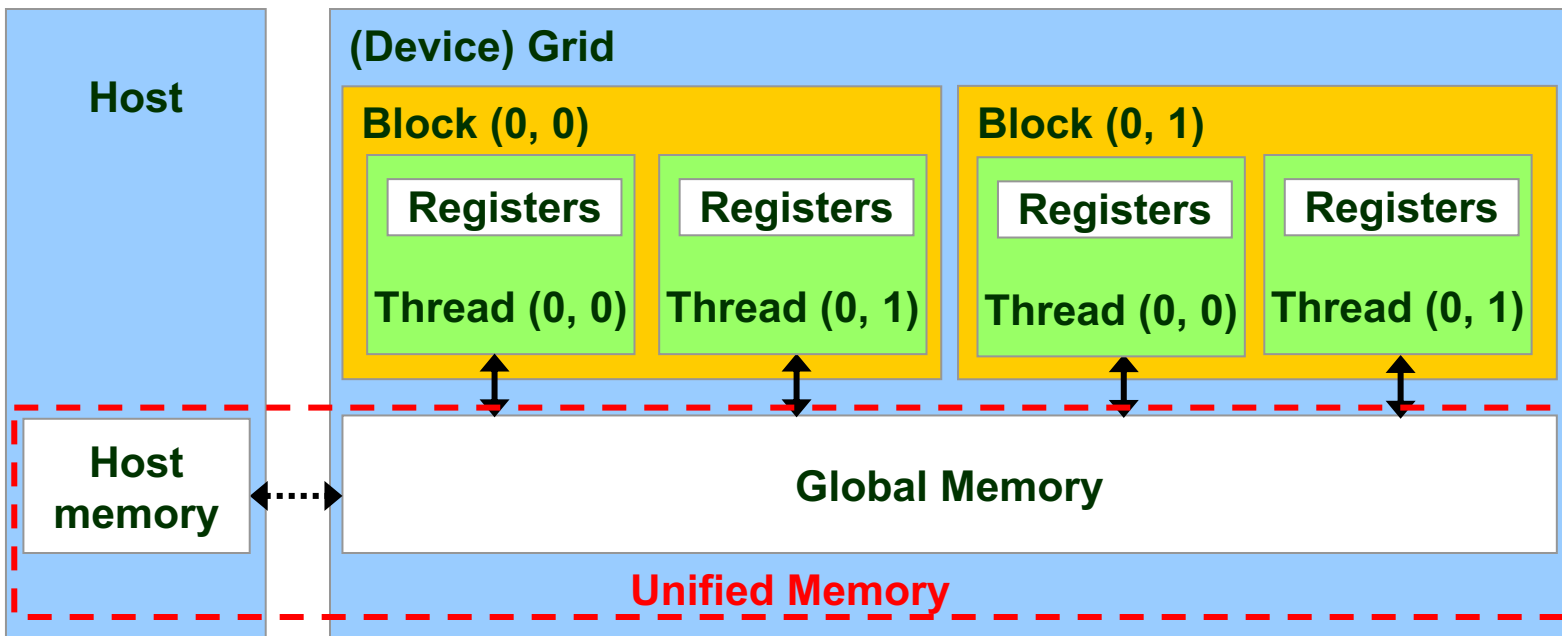
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

- Single memory address space accessible from all CPUs/GPUs in a single server
 - maintain single copy of data
- On-demand page migration - hardware/software handles automatically the data migration between the host and the device maintaining consistency between them

CUDA programming

Unified Memory



Device code (kernel) can:

- R/W per-thread **registers**
- R/W all-shared **global memory**
- **R/W managed memory (Unified Memory)**

Host code can

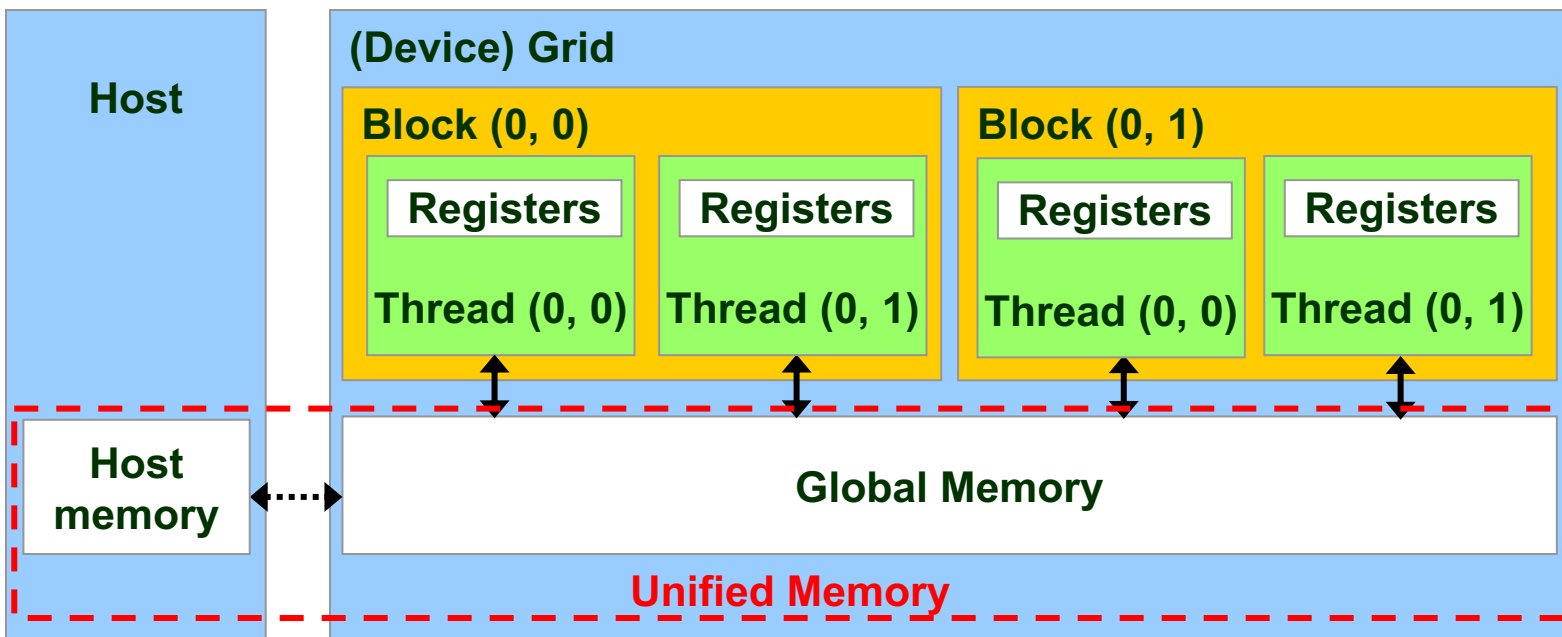
- Transfer data to/from per grid **global memory**
- **R/W managed memory (Unified Memory)**

In modern GPUs:

- there are specialized hardware units managing page faulting
- data is migrated on demand, meaning that data gets copied only on page fault
- possibility to oversubscribe memory, enabling larger arrays than the device memory size

CUDA programming

Unified Memory



Can be optimized

- `cudaMemAdvise()`,
- `cudaMemPrefetchAsync()`,
- `cudaMemcpyAsync()`

`cudaMallocManaged(void** ptr, size_t size)`

- Allocates an object in the Unified Memory address space.
- Two parameters, with an optional third parameter.
 - Address of a pointer to the allocated object
 - Size of the allocated object in terms of bytes
 - [Optional] Flag indicating if memory can be accessed from any device or stream

`cudaFree()`

- Frees object from unified memory.
- One parameter
 - Pointer to freed object

CPU

Memory Allocation in Host memory
& Initialization of Values

```
int main(){

    float *h_A, *h_B, *h_C;

    int n = 10000000 // size of an array
    int size = n * sizeof(float);

    h_A = (float*)malloc(size);
    h_B = (float*)malloc(size);
    h_C = (float*)malloc(size);

    // Initialize array
    for(int i = 0; i < array_size; i++){
        h_A[i] = 1.0f;
        h_B[i] = 2.0f;}

    vecAdd(h_A, h_B, h_C, n);

    // Deallocate host memory
    free(h_A); free(h_A); free(h_C);
}
```

```
int main(){

    float *h_A, *h_B, *h_C;

    int n = 10000000 // size of an array
    int size = n * sizeof(float);

    h_A = (float*)malloc(size);
    h_B = (float*)malloc(size);
    h_C = (float*)malloc(size);

    // Initialize array
    for(int i = 0; i < array_size; i++){
        h_A[i] = 1.0f;
        h_B[i] = 2.0f;}

    vecAdd(h_A, h_B, h_C, n);

    // Deallocate host memory
    free(h_A); free(h_A); free(h_C);
}
```

```
int main(){

    float *A, *B, *C;

    int n = 10000000 // size of an array
    int size = n * sizeof(float);

    cudaMallocManaged(&A, size);
    cudaMallocManaged(&B, size);
    cudaMallocManaged(&C, size);

    // Initialize array
    for(int i = 0; i < array_size; i++){
        A[i] = 1.0f;
        B[i] = 2.0f;}

    vecAdd(A, B, C, n);

    // Deallocate host memory
    cudaFree(h_a); cudaFree(h_b); cudaFree(h_c);
}
```

```
void vecAdd(float *h_A, float *h_B, float *h_C,
int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;
    cudaMalloc((void **) &d_A, size);
    cudaMalloc((void **) &d_B, size);
    cudaMalloc((void **) &d_C, size);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Kernel run

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

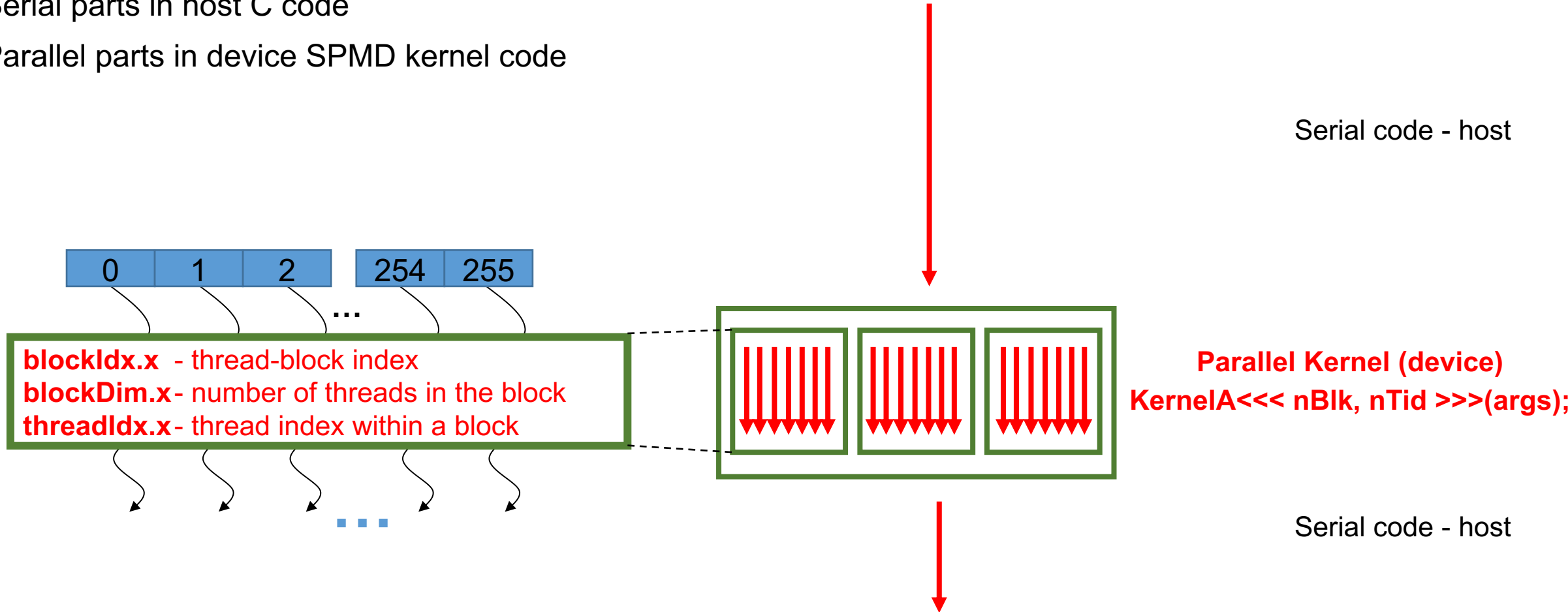
```
void vecAdd(float *A, float *B, float *C, int n)
{
    // Kernel run
}
```


CUDA programming

CUDA Execution Model

Heterogeneous host (CPU) + device (GPU) application C program

- Serial parts in host C code
- Parallel parts in device SPMD kernel code



Device code or kernel

- `__global__` defines a kernel function 

Kernel Code

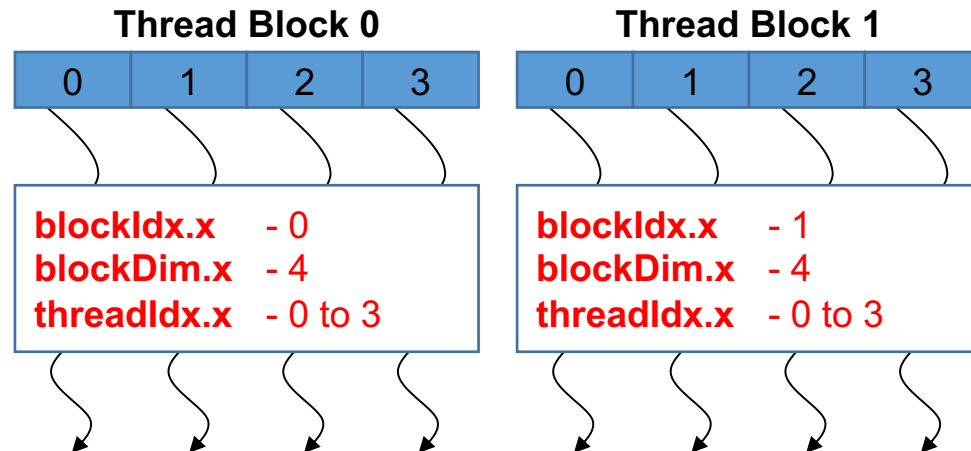
```
__global__ void say_hello()  
{  
    int global_index = blockIdx.x * blockDim.x + threadIdx.x;  
    int total_threads = blockDim.x * gridDim.x;  
    printf("Hello from thread %d,  
          block %d,  
          my global index is %d,  
          total number of threads is %d\n",  
          threadIdx.x,  
          blockIdx.x,  
          global_index,  
          total_threads);  
}
```

Host code – kernel execution

- `say_hello<<< 2, 4 >>>()`

Grid dimension = # of blocks

Block dimension = # of threads per block



Each thread uses indices to decide what data to work on

- **blockIdx.x** – block index in x direction
- **threadIdx.x** – thread index in x direction
- **blockDim.x** – block size (# of threads per block) in x dir.

Hands-On Hello world in CUDA

Univerza v Ljubljani



TECHNISCHE
UNIVERSITÄT
WIEN



VSB TECHNICAL
UNIVERSITY
OF OSTRAVA

IT4INNOVATIONS
NATIONAL SUPERCOMPUTING
CENTER



Co-funded by the
Erasmus+ Programme
of the European Union

This project has been funded with support from the European Commission.

This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Hands-On Hello world in CUDA

- Start simple with a classic hello world
- Choose your language of preference (C++, Fortran)
 - C++ is highly recommended
- `cd 01_hello_world/<lang>/Task`
- Open the source code `hello_world.{cu, CUF}`
- Finish the TODO tasks
- Compile using
 - `nvcc hello_world.cu -o hello_world.x`
 - `nvfortran hello_world.CUF -o hello_world.x`
- And run as usual
 - `./hello_world.x`

C++ sample output (might be in different order):

```
Launching the kernel with 2 blocks, each with 4 threads
Kernel was launched, waiting for its completion
Hello from thread 0/4, block 0/2, my global index is 0/8
Hello from thread 1/4, block 0/2, my global index is 1/8
Hello from thread 2/4, block 0/2, my global index is 2/8
Hello from thread 3/4, block 0/2, my global index is 3/8
Hello from thread 0/4, block 1/2, my global index is 4/8
Hello from thread 1/4, block 1/2, my global index is 5/8
Hello from thread 2/4, block 1/2, my global index is 6/8
Hello from thread 3/4, block 1/2, my global index is 7/8
Kernel execution completed
```

Fortran sample output (might be in different order):

```
Launching the kernel with 2 blocks, each with 4 threads
Kernel was launched, waiting for its completion

thread_index  block_size  block_index  grid_size  global_idx  total_threads
           1             4             1             2             1             8
           2             4             1             2             2             8
           3             4             1             2             3             8
           4             4             1             2             4             8
           1             4             2             2             5             8
           2             4             2             2             6             8
           3             4             2             2             7             8
           4             4             2             2             8             8

Kernel execution completed
```

CUDA Programming cont.

Univerza v Ljubljani



TECHNISCHE
UNIVERSITÄT
WIEN



VSB TECHNICAL
UNIVERSITY
OF OSTRAVA

IT4INNOVATIONS
NATIONAL SUPERCOMPUTING
CENTER



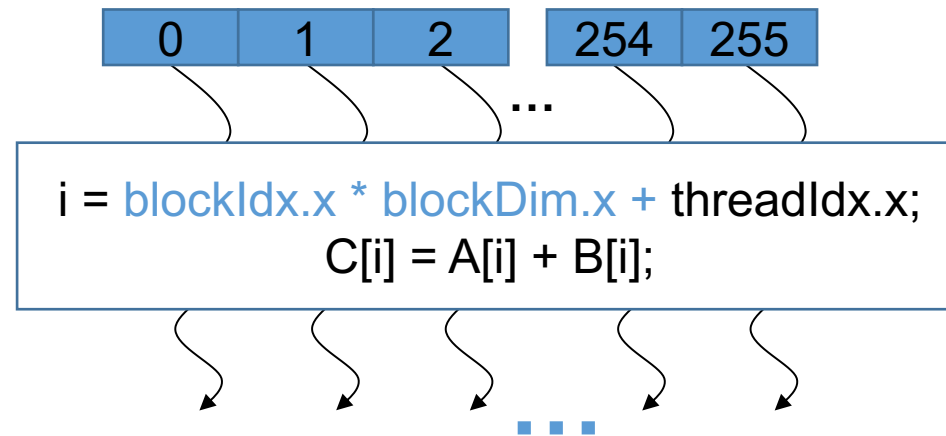
Co-funded by the
Erasmus+ Programme
of the European Union

This project has been funded with support from the European Commission.

This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

A CUDA kernel is executed by a grid (array) of threads

- All threads in a grid run the same kernel code (Single Program Multiple Data)
- Each thread has indexes that it uses to compute memory addresses and make control decisions



Divide thread array into multiple blocks

- **Threads within a block cooperate** via
 - shared memory,
 - atomic operations and
 - barrier synchronization
- **Threads in different blocks do not interact**

blockIdx and threadIdx

- Each thread uses indices to decide what data to work on
 - blockIdx: 1D, 2D, or 3D
 - threadIdx: 1D, 2D, or 3D

Thread Block 0



```
i = blockIdx.x * blockDim.x + threadIdx.x;  
C[i] = A[i] + B[i];
```

Thread Block 1



```
i = blockIdx.x * blockDim.x + threadIdx.x;  
C[i] = A[i] + B[i];
```

Thread Block N-1



```
i = blockIdx.x * blockDim.x + threadIdx.x;  
C[i] = A[i] + B[i];
```


Device code or kernel

- compute vector sum $C = A + B$
- each thread performs one pair-wise addition

```
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C[i] = A[i] + B[i];
}
```

__global__ defines a kernel function

- each “__” consists of two underscore characters
- kernel function must return void

Each thread uses indices to decide what data to work on

- **blockIdx.x** – block index in x direction
- **threadIdx.x** – thread index in x direction
- **blockDim.x** – block size (# of threads per block) in x dir.
- Note: 1D indexing uses .x only, 2D uses .x, .y and 3D uses .x, .y, .z

Host code

- Kernel execution – host code that launches kernel
- GPU hardware creates a grid of threads
- each thread executes the kernel function from previous slide

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    // d_A, d_B, d_C allocations and memory copies are done
    //      x y z direction
    dim3 DimGrid (2, 1, 1);    // number of blocks per grid to be launched
    dim3 DimBlock(4, 1, 1);    // number of threads per block to be launched
    vecAddKernel<<<DimGrid,DimBlock>>>(d_A, d_B, d_C, n);
}
```

Host code

- Kernel execution – host code that launches kernel
- GPU hardware creates a grid of threads
- each thread executes the kernel function from previous slide

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    // d_A, d_B, d_C allocations and memory copies are done
    // launches 2 block in a grid and 4 threads per block
    vecAddKernel<<<2,4>>>(d_A, d_B, d_C, n);
}
```

CUDA programming

Vector Addition Kernel Launch

Host code

- Executes $\text{ceil}(n/256.0)$ blocks of 256 threads each
- the ceiling function makes sure that there are enough threads to cover all elements.

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    // d_A, d_B, d_C allocations and memory copies are done
    vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);
}
```

Host code

- This is an equivalent way to express the ceiling function.

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    // d_A, d_B, d_C allocations and memory copies are done
    dim3 DimGrid((n-1)/256 + 1, 1, 1);
    dim3 DimBlock(256, 1, 1);
    vecAddKernel<<<DimGrid,DimBlock>>>(d_A, d_B, d_C, n);
}
```

CUDA programming

Vector Addition Kernel Launch

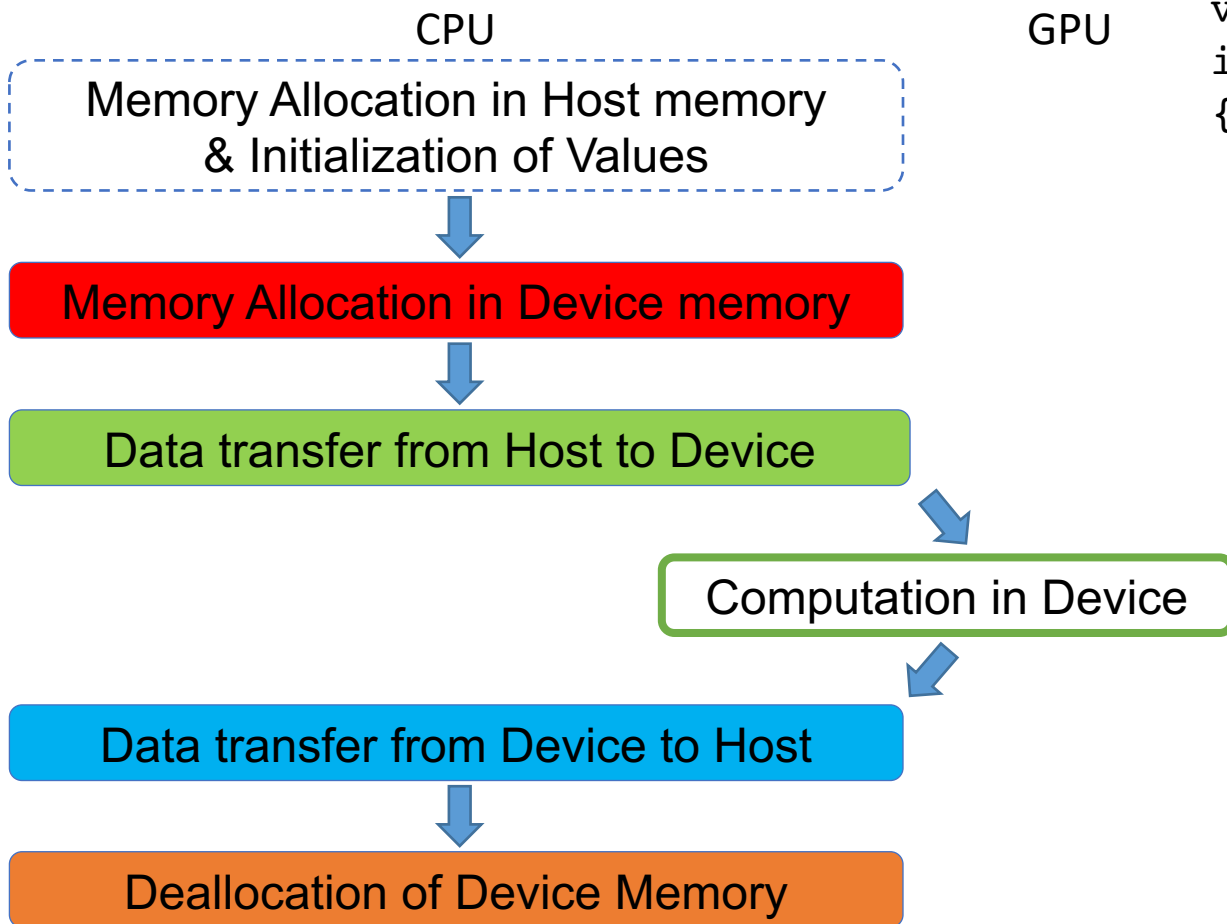
- Host: launches „extra“ block to cover all elements – ensures that there is enough threads to process all elements
- Kernel: controls that thread does not read unallocated memory
- Host: DimBlock equals to
- Kernel: blockDim
- Kernel: threadIdx is in range <0, DimBlock)
- Kernel: blockIdx is in range <0, DimGrid)

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    dim3 DimGrid(ceil(n/256.0), 1, 1);
    dim3 DimBlock(256, 1, 1);
    vecAddKernel<<<DimGrid,DimBlock>>>(d_A, d_B, d_C, n);
}
```

```
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C[i] = A[i] + B[i];
}
```

CUDA programming

Vector Addition – with kernel exec.



```
void vecAdd(float *h_A, float *h_B, float *h_C,
int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;
    cudaMalloc((void **) &d_A, size);
    cudaMalloc((void **) &d_B, size);
    cudaMalloc((void **) &d_C, size);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    vecAddKernel<<<ceil(n/256.0),256>>>
    (d_A, d_B, d_C, n);

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```


CUDA programming

Kernel timing using events

Use CUDA events for timing CUDA related execution time.

- Works as "markers" in execution queue
- Besides timing, they are crucial for GPU synchronization
- **Important!** In order to compute elapsed time correctly. Both events must "happen". That is, they need to reach the end of execution queue
- Can be ensured by waiting for the event to "happen" using `cudaEventSynchronize()` or synchronization with entire GPU by `cudaDeviceSynchronize()`

```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    ...
    float timeInMs;
    cudaEvent_t startEvent, endEvent;

    cudaEventCreate(&startEvent);
    cudaEventCreate(&endEvent);

    cudaEventRecord(startEvent);

    vecAddKernel<<<ceil(n/256.0),256>>>
        (d_A, d_B, d_C, n);

    cudaEventRecord(endEvent);

    cudaDeviceSynchronize();
    cudaEventElapsedTime
        (&timeInMs, startEvent, endEvent);

    cudaEventDestroy(endEvent);
    cudaEventDestroy(startEvent);

    ...
}
```

Macro and definition:

```
#define gpuErrchk(ans) { gpuAssert((ans), __FILE__, __LINE__); }

inline void gpuAssert(cudaError_t code, const char *file, int line, bool abort=true)
{
  if (code != cudaSuccess)
  {
    fprintf(stderr,"GPUassert: %s %s %d\n", cudaGetErrorString(code), file, line);

    if (abort) exit(code);
  }
}
```

Usage:

- API calls:

```
gpuErrchk( cudaMalloc(&a_d, size*sizeof(int)) );
gpuErrchk( cudaMemcpy(a_h, a_d, size * sizeof(int), cudaMemcpyDeviceToHost) );
```
- Kernel Execution

```
kernel<<<1,1>>>(a);
gpuErrchk( cudaPeekAtLastError() );
gpuErrchk( cudaDeviceSynchronize() );
```

Hands-on Vector Addition

Univerza v Ljubljani



TECHNISCHE
UNIVERSITÄT
WIEN



VSB TECHNICAL
UNIVERSITY
OF OSTRAVA

IT4INNOVATIONS
NATIONAL SUPERCOMPUTING
CENTER



Co-funded by the
Erasmus+ Programme
of the European Union

This project has been funded with support from the European Commission.

This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Vector addition – single GPU

- `cd 02_vector_add/<lang>/Task`
- Task 2a: Using explicit memory management
 - Open file `vec_add{.cu, .CUF}` and search for TODOs:
 - Implement vector addition computation in CUDA kernel (slide 74)
 - Fill in explicit data copy from GPU to CPU after computation (slide 75)
 - Compile with `--std=c++11` and run
- Task 2b: Using unified memory
 - Open file `vec_add_managed{.cu, .CUF}` and search for TODOs:
 - Allocate managed memory (slide 60)
 - Compile with `--std=c++11` and run

Task 2a

```
__global__  
void cudaVecAdd (...){  
    int i = threadIdx.x + blockDim.x *  
            blockDim.x;  
    if(i<N) C[i] = A[i] + B[i];  
}  
...  
cudaMemcpy(h_C, d_C, size,  
           cudaMemcpyDeviceToHost);  
...
```

Task 2b

```
...  
cudaMallocManaged(&A, size);  
cudaMallocManaged(&B, size);  
cudaMallocManaged(&C, size);  
...
```

Task 2a

```
idx = blockDim%x * (blockIdx%x - 1)
      + threadIdx%x
if (idx <= n) then
    C(idx) = A(idx) + B(idx)
end if
...
h_C = d_C
...
```

Task 2b

```
...
real, allocatable, managed :: A(:),
    B(:), C(:)
    allocate(A(N))
    allocate(B(N))
    allocate(C(N))
...

```

Multi-GPU Programming

Univerza v Ljubljani



TECHNISCHE
UNIVERSITÄT
WIEN



VSB TECHNICAL
UNIVERSITY
OF OSTRAVA

IT4INNOVATIONS
NATIONAL SUPERCOMPUTING
CENTER

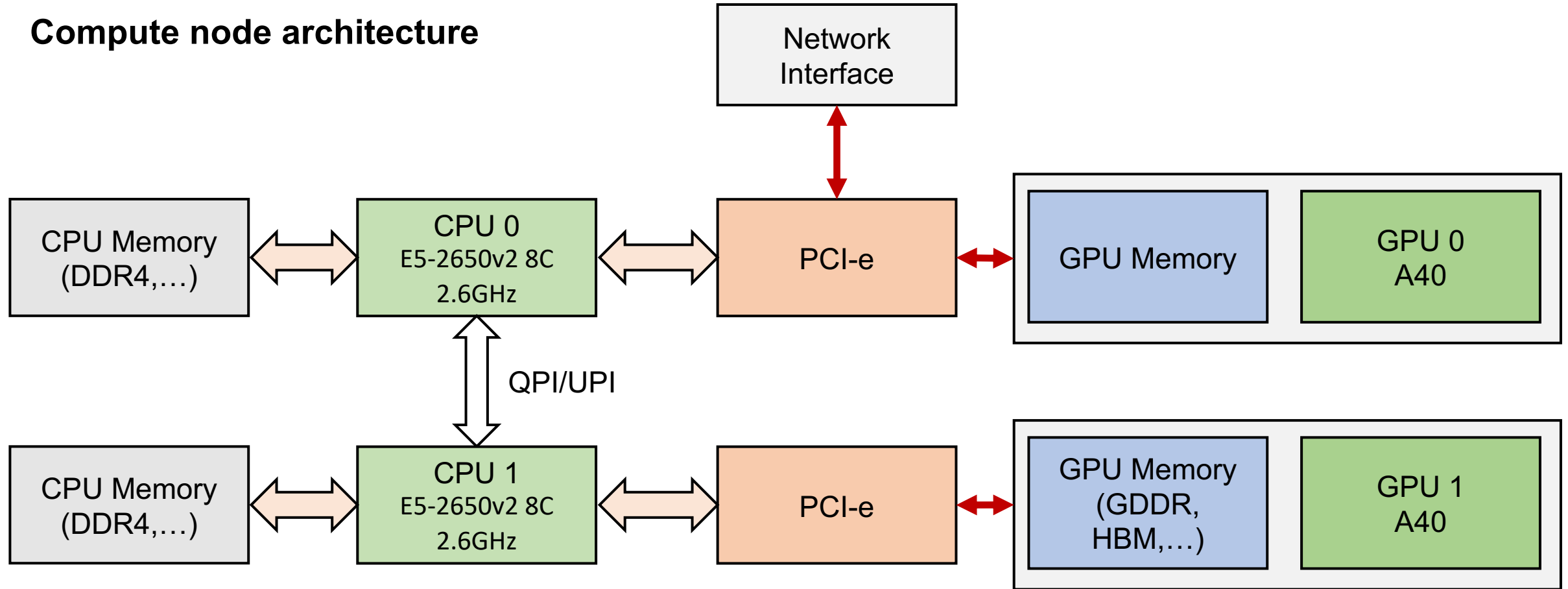


Co-funded by the
Erasmus+ Programme
of the European Union

This project has been funded with support from the European Commission.

This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Compute node architecture

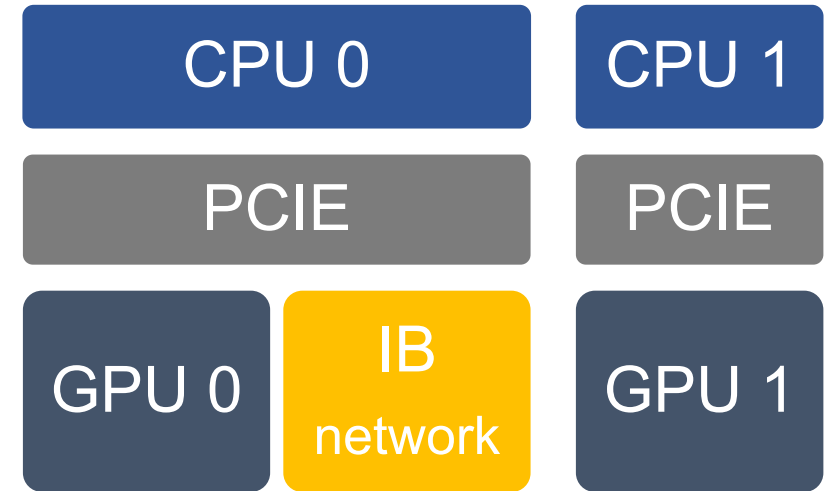


CUDA programming

MultiGPU programming basics

Multi-GPU system

- GPU's are numbered from 0 to n-1, where n is the number of GPU's.
- The CUDA driver always starts with a default active device.
- There are two broad types of Multi GPU communication:
 - Through the PCIE bus
 - Through NVLINK



```
$ nvidia-smi topo -m
```

```
GPU0    GPU1    mlx5_0    CPU Affinity    NUMA Affinity
GPU0    X      SYS      NODE          0-7,16-23       0
GPU1    SYS    X      SYS          8-15,24-31      1
mlx5_0  NODE   SYS      X
```

SYS = Connection traversing PCIE as well as the SMP interconnect between NUMA nodes (e.g., QPI/UPI)

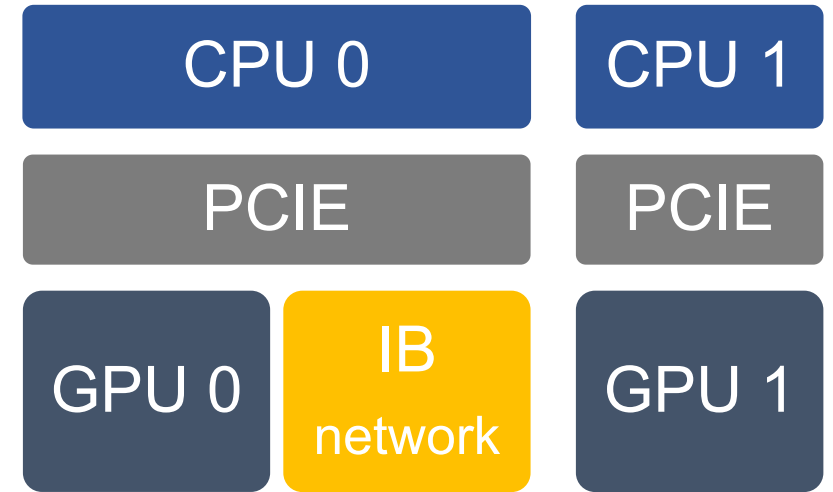
NODE = Connection traversing PCIE as well as the interconnect between PCIE Host

CUDA programming

CUDA host API calls for Multi GPU's

cudaSetDevice()

- Set GPU device to use for device code execution on the active host thread.
- Requires one parameter:
 - An int with the device id number
- This function doesn't affect other host threads, meaning that setting the device on one thread will not set the device in other host threads. Also doesn't affect previous async calls.



cudaGetDevice()

- Get GPU device being currently used by the active host thread
- Requires one parameter:
 - An int pointer to store the device id

cudaGetDeviceCount()

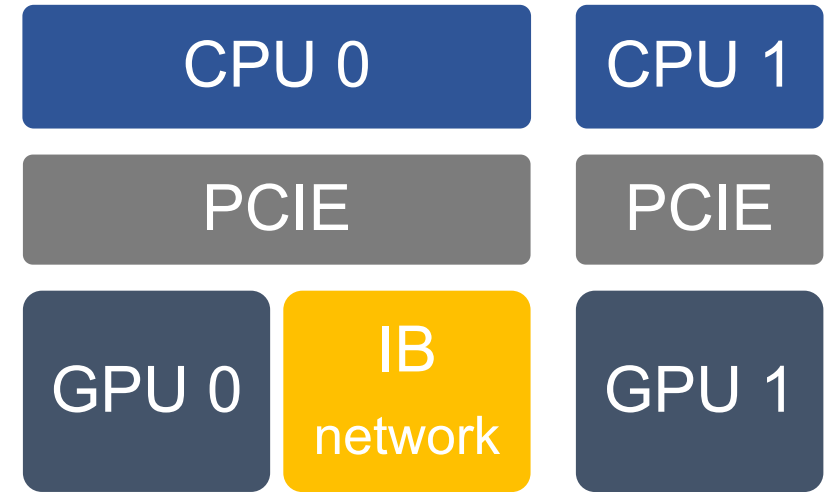
- Get the number of CUDA-capable devices in the system.
- Requires one parameter:
 - An int pointer to store the device count

CUDA programming

CUDA host API calls for Multi GPU's

cudaSetDevice()

- Set GPU device to use for device code execution on the active host thread.
- Requires one parameter:
 - An int with the device id number
- This function doesn't affect other host threads, meaning that setting the device on one thread will not set the device in other host threads. Also doesn't affect previous async calls.



Memory allocation

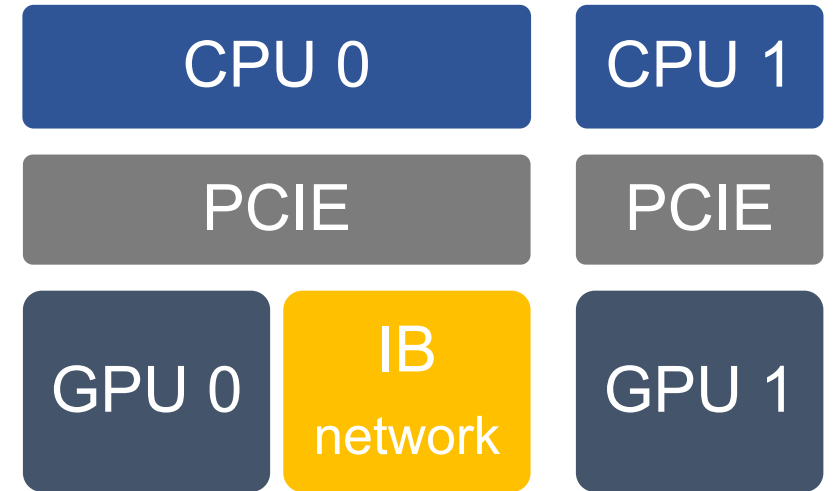
To allocate or associate memory with a specific device using non-Managed CUDA-API calls, it's necessary to call **cudaSetDevice()** before doing the allocation call.

- cudaMalloc() - allocates an object in the device global memory
- cudaHostAlloc() - allocates pinned memory on the host

CUDA programming

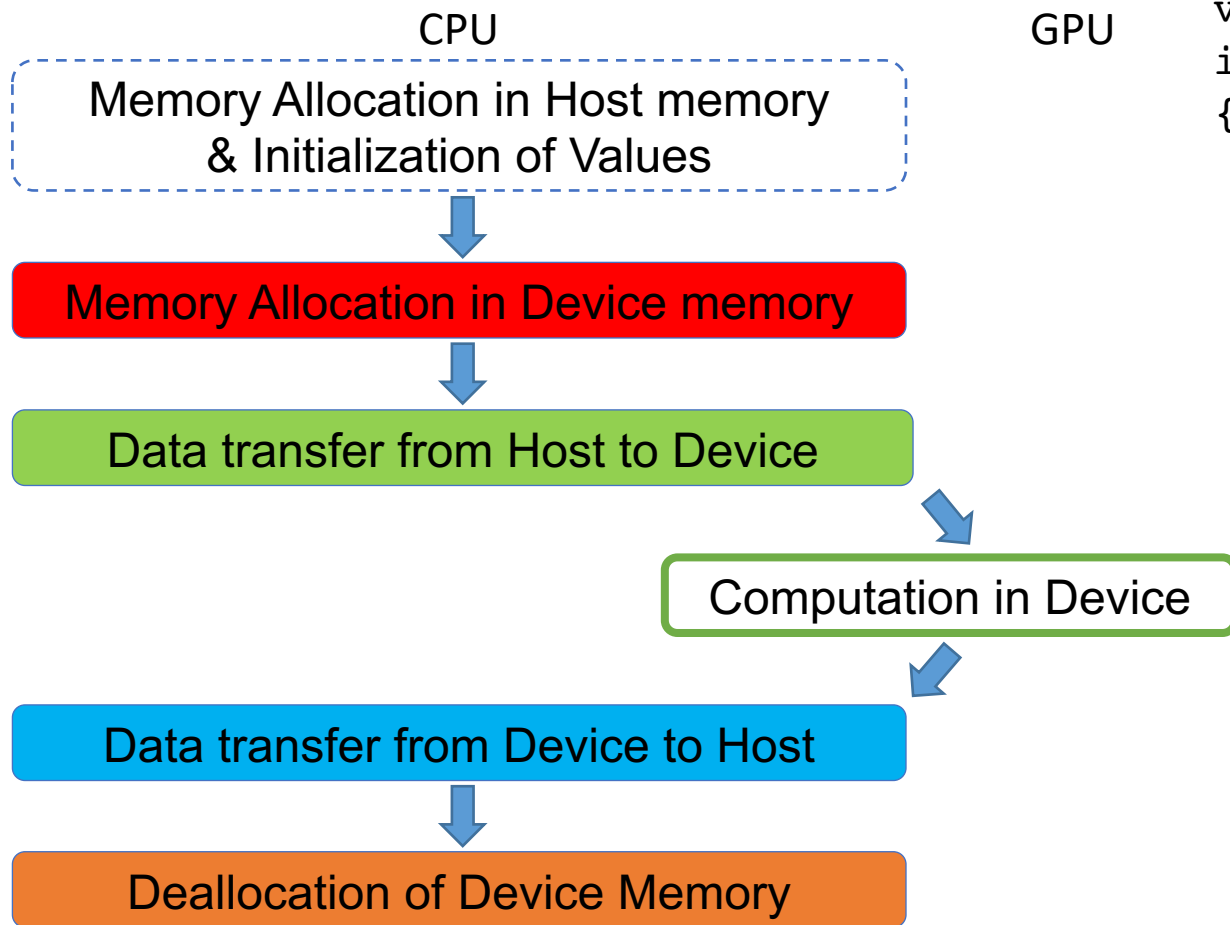
CUDA runtime calls affected by `cudaSetDevice`

- If **`cudaSetDevice()`** was called before a kernel launching call, the kernel will execute in the active device.
 - It's crucial that every non managed memory being used in the kernel resides in the active device, otherwise an error will occur.
- If **`cudaSetDevice()`** was called before a **`cudaStreamCreate()`**, then the stream will be associated with the active device.
- The synchronization functions: **`cudaDeviceSynchronize()`**, **`cudaStreamSynchronize()`** are also affected by **`cudaSetDevice()`**, synchronizing tasks only for the active device on the active host thread



CUDA programming

Vector Addition – with kernel exec.



```
void vecAdd(float *h_A, float *h_B, float *h_C,
int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;
    cudaMalloc((void **) &d_A, size);
    cudaMalloc((void **) &d_B, size);
    cudaMalloc((void **) &d_C, size);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

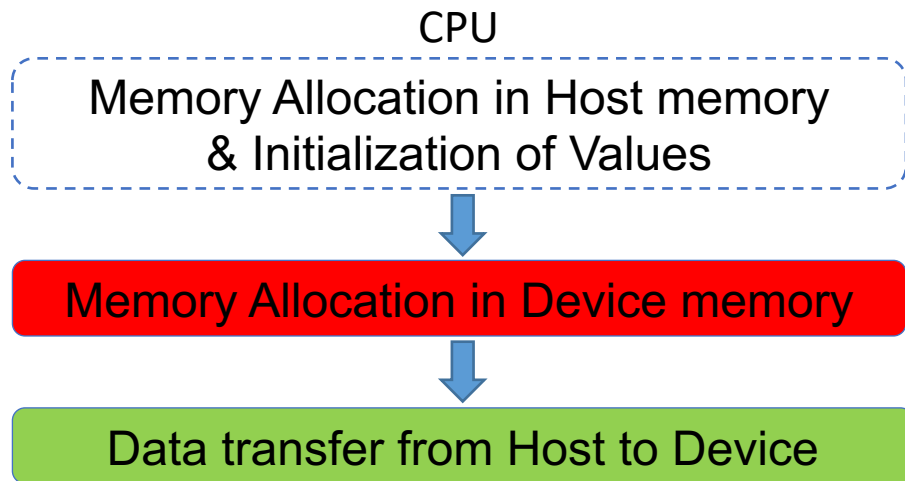
    vecAddKernel<<<ceil(n/256.0),256>>>
    (d_A, d_B, d_C, n);

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

CUDA programming

Multi-GPU Vector Addition – Part 1



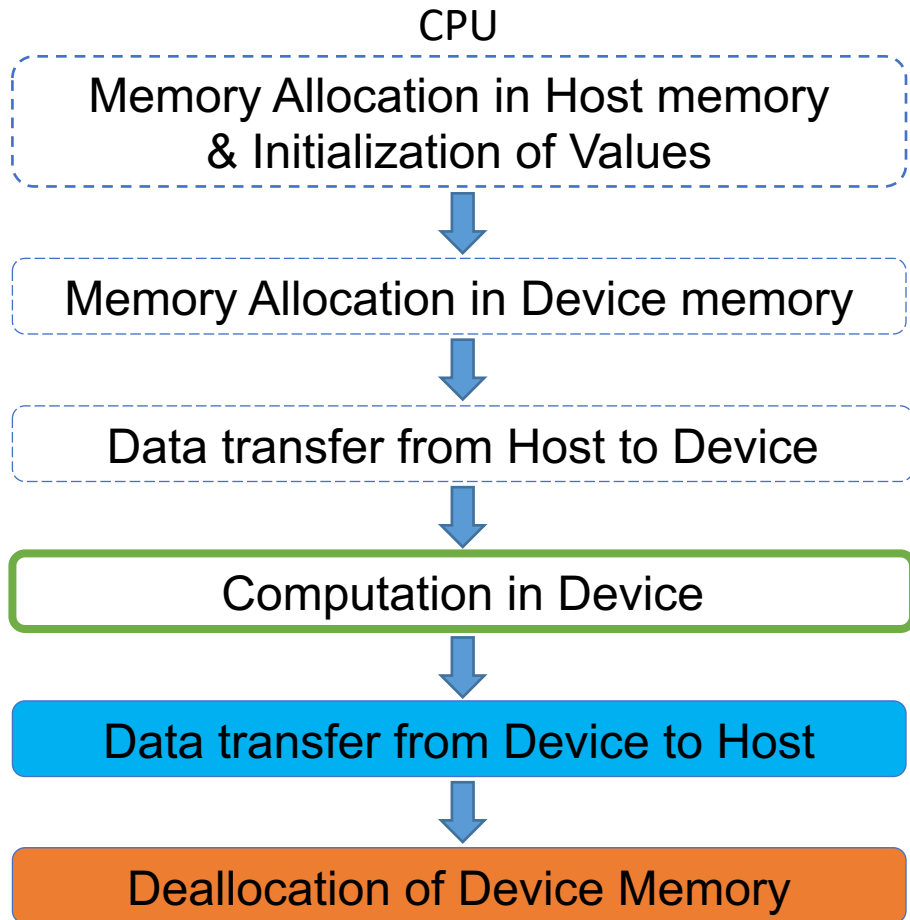
```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int n0 = n / 2;
    int n1 = n - n0;
    int size0 = n0 * sizeof(float);
    int size1 = n1 * sizeof(float);
    float *d_A0, *d_B0, *d_C0;
    float *d_A1, *d_B1, *d_C1;

    cudaSetDevice(0);
    cudaMalloc((void **) &d_A0, size0);
    cudaMalloc((void **) &d_B0, size0);
    cudaMalloc((void **) &d_C0, size0);
    cudaMemcpy(d_A0, &h_A[0], size0, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B0, &h_B[0], size0, cudaMemcpyHostToDevice);

    cudaSetDevice(1);
    cudaMalloc((void **) &d_A1, size1);
    cudaMalloc((void **) &d_B1, size1);
    cudaMalloc((void **) &d_C1, size1);
    cudaMemcpy(d_A0, &h_A[n0], size1, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B0, &h_B[n0], size1, cudaMemcpyHostToDevice);
}
```

CUDA programming

Multi-GPU Vector Addition – Part 2



```
int n0 = floor(n/2.0);
int n1 = ceil(n/2.0);
int size0 = n0 * sizeof(float);
int size1 = n1 * sizeof(float);

cudaSetDevice(0);
vecAddKernel<<<ceil(n0/256.0),256>>> (d_A0, d_B0, d_C0, n0);

cudaSetDevice(1);
vecAddKernel<<<ceil(n1/256.0),256>>> (d_A1, d_B1, d_C1, n1);

cudaMemcpy(&h_C[0], d_C0, size, cudaMemcpyDeviceToHost);
cudaMemcpy(&h_C[n0], d_C1, size, cudaMemcpyDeviceToHost);

cudaFree(d_A0); cudaFree(d_A1);
cudaFree(d_B0); cudaFree(d_B1);
cudaFree(d_C0); cudaFree(d_C1);
}
```

CUDA programming

MultiGPU Vector Addition

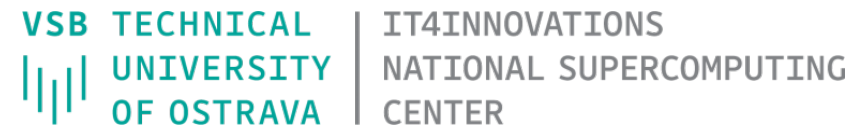
```
float *m_A0, float *m_B0, *m_A1, float *m_B1, int n;  
int size = n * sizeof(float);  
  
cudaSetDevice(0); // Will set the active device to 0  
cudaMalloc((void**) &m_A0, size); // Will allocate memory on device 0  
cudaMalloc((void**) &m_B0, size); // Will allocate memory on device 0  
  
cudaSetDevice(1); // Will set the active device to 1  
cudaMalloc((void**) &m_A1, size); // Will allocate memory on device 1  
cudaMalloc((void**) &m_B1, size); // Will allocate memory on device 1  
  
// Memory initialization on the Host and memory transfers  
  
cudaSetDevice(0); // Set the device for kernel execution  
vecAdd<<<gridDim, blockDim>>>(m_A0,m_B0);  
  
cudaSetDevice(1); // Set the device for kernel execution  
vecAdd<<< gridDim, blockDim>>>(m_A1,m_B1);  
  
cudaFree(m_A0); cudaFree(m_B0);  
cudaFree(m_A1); cudaFree(m_B1);
```


Environment variable controlling devices visibility

- Useful for selecting or restricting the set of available GPUs for specific application even without the access to the source code
- Execute `export CUDA_VISIBLE_DEVICES=<comma separated list of GPU IDs>` before running the app
- To list all available GPU IDs run `nvidia-smi` from command line
- Single GPU applications (might cooperate with a peer you share a node with to select different GPU):
`export CUDA_VISIBLE_DEVICES=0 ./app`
- Multi GPU applications:
`export CUDA_VISIBLE_DEVICES=0,1 ./app`

Hands-on Multi-GPU Vector Addition

Univerza v Ljubljani



Co-funded by the
Erasmus+ Programme
of the European Union

This project has been funded with support from the European Commission.
This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Vector addition – multi-GPU

- `cd 03_vector_add_multigpu/<lang>/Task`
- Task 3a - Using explicit memory management
 - Open file `vec_add_multi_GPU{.cu, .CUF}` and search for TODOs:
 - Rewrite host allocation so there is only single copy of host arrays.
 - Allocate arrays A, B and C with the correct size and set up `h_A0`, `h_A1`, etc. as pointers into host arrays for particular GPU (e.g. `h_A0 -> A[0]`, `h_A1 -> A[size0]`)
 - Compile with `--std=c++11` and run (don't forget the `export CUDA_VISIBLE_DEVICES=0,1 ./app`)
- Task 3b - Using unified memory
 - Open file `vec_add_multi_GPU_managed{.cu, .CUF}` and search for TODOs:
 - Same task as in 3a, but allocate managed memory for host arrays and set up the pointers correctly
 - Uncomment and think about implications of prefetching
 - Compile with `--std=c++11` and run (don't forget the `export CUDA_VISIBLE_DEVICES=0,1 ./app`)

Task 3a

```
cudaHostAlloc(&h_A, size, cudaHostAllocDefault);
cudaHostAlloc(&h_B, size, cudaHostAllocDefault);
cudaHostAlloc(&h_C, size, cudaHostAllocDefault);
h_A0 = &h_A[0]; h_A1 = &h_A[N0];
h_B0 = &h_B[0]; h_B1 = &h_B[N0];
h_C0 = &h_C[0]; h_C1 = &h_C[N0];

// Setting device 0 as current device.
gpuErrchk(cudaSetDevice(0));
// Allocation of device memory on device 0.
gpuErrchk(cudaMalloc(&d_A0, size0));
// Setting device 1 as current device.
...
gpuErrchk(cudaSetDevice(1));
// Allocation of device memory on device 0.
gpuErrchk(cudaMalloc(&d_A1, size1));
...
```

Task 3b

```
cudaMallocManaged(&A, size);
cudaMallocManaged(&B, size);
cudaMallocManaged(&C, size);
A0 = &A[0]; A1 = &A[N0];
B0 = &B[0]; B1 = &B[N0];
C0 = &C[0]; C1 = &C[N0];
```

Task 3a

```
real, allocatable, pinned, target :: h_A(:),  
    h_B(:), h_C(:)  
real, pointer :: h_A0(:), h_B0(:), h_C0(:)  
real, pointer :: h_A1(:), h_B1(:), h_C1(:)  
h_A0 => h_A(1:N0) h_A1 => h_A(N0+1:N)  
h_B0 => h_B(1:N0) h_B1 => h_B(N0+1:N)  
h_C0 => h_C(1:N0) h_C1 => h_C(N0+1:N)  
...  
result = cudaSetDevice(0)  
allocate(d_A0(N0))  
...  
result = cudaSetDevice(1)  
allocate(d_A1(N1))
```

Task 3b

```
real, allocatable, managed, target ::  
    h_A(:), h_B(:), h_C(:)  
real, managed, pointer :: h_A0(:),  
    h_B0(:), h_C0(:)  
real, managed, pointer :: h_A1(:),  
    h_B1(:), h_C1(:)  
h_A0 => h_A(1:N0) h_A1 => h_A(N0+1:N)  
h_B0 => h_B(1:N0) h_B1 => h_B(N0+1:N)  
h_C0 => h_C(1:N0) h_C1 => h_C(N0+1:N)
```

Multi-Dimensional Grids

Univerza v Ljubljani



TECHNISCHE
UNIVERSITÄT
WIEN



VSB TECHNICAL
UNIVERSITY
OF OSTRAVA

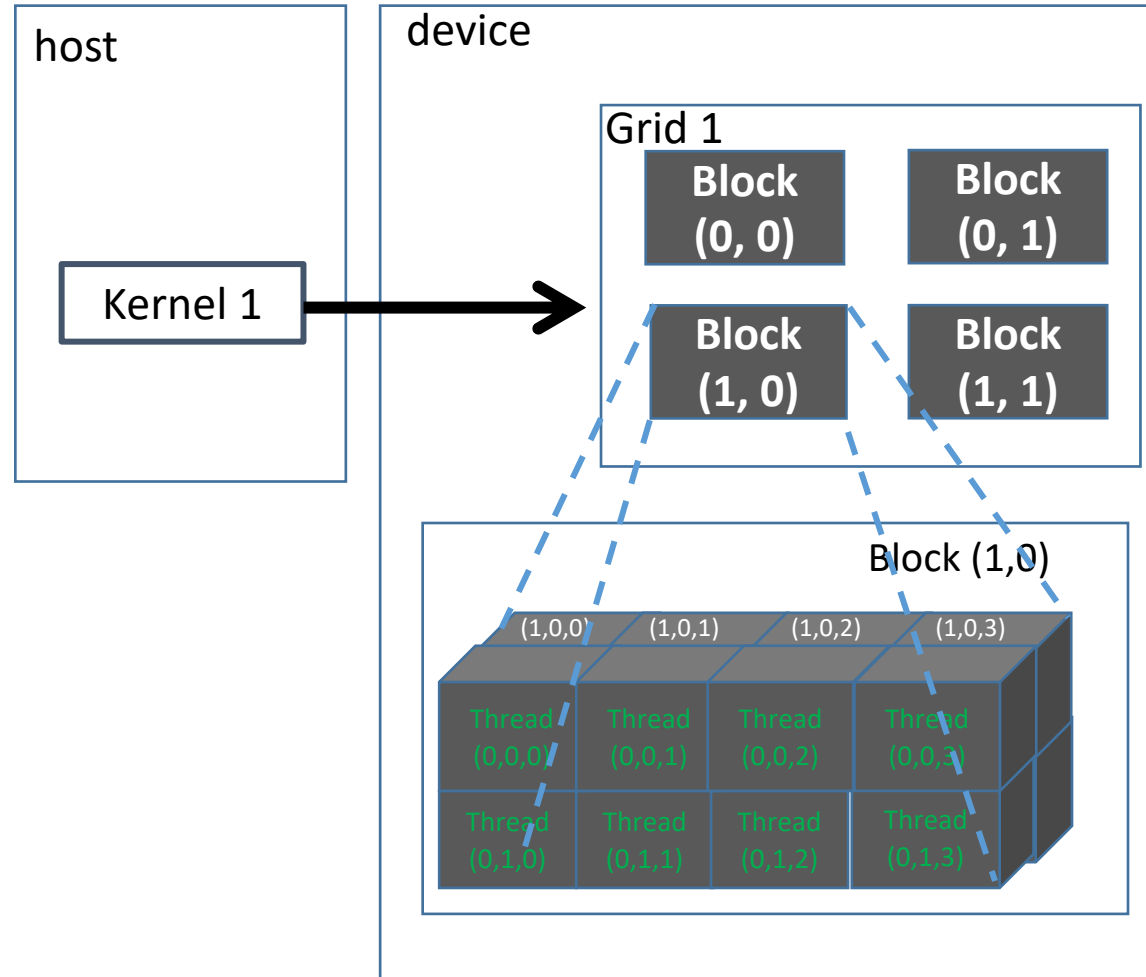
IT4INNOVATIONS
NATIONAL SUPERCOMPUTING
CENTER



Co-funded by the
Erasmus+ Programme
of the European Union

This project has been funded with support from the European Commission.

This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

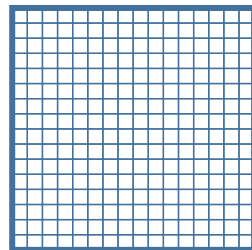


CUDA programming

Processing a Picture with a 2D Grid

Work distribution

- image will be addressed in 2D blocks of size
 - 16x16 threads
- some threads, highlighted in orange, will be idle



1 block:
16×16
threads
per block

Control flow divergence

- not all threads in a Block will follow the same control flow path

62×76 picture

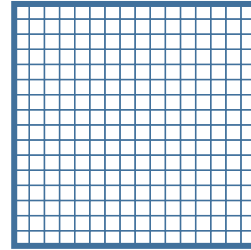


CUDA programming

Processing a Picture with a 2D Grid

Work distribution

- image will be addressed in 2D blocks of size
 - 16x16 threads
- some threads, highlighted in orange, will be idle

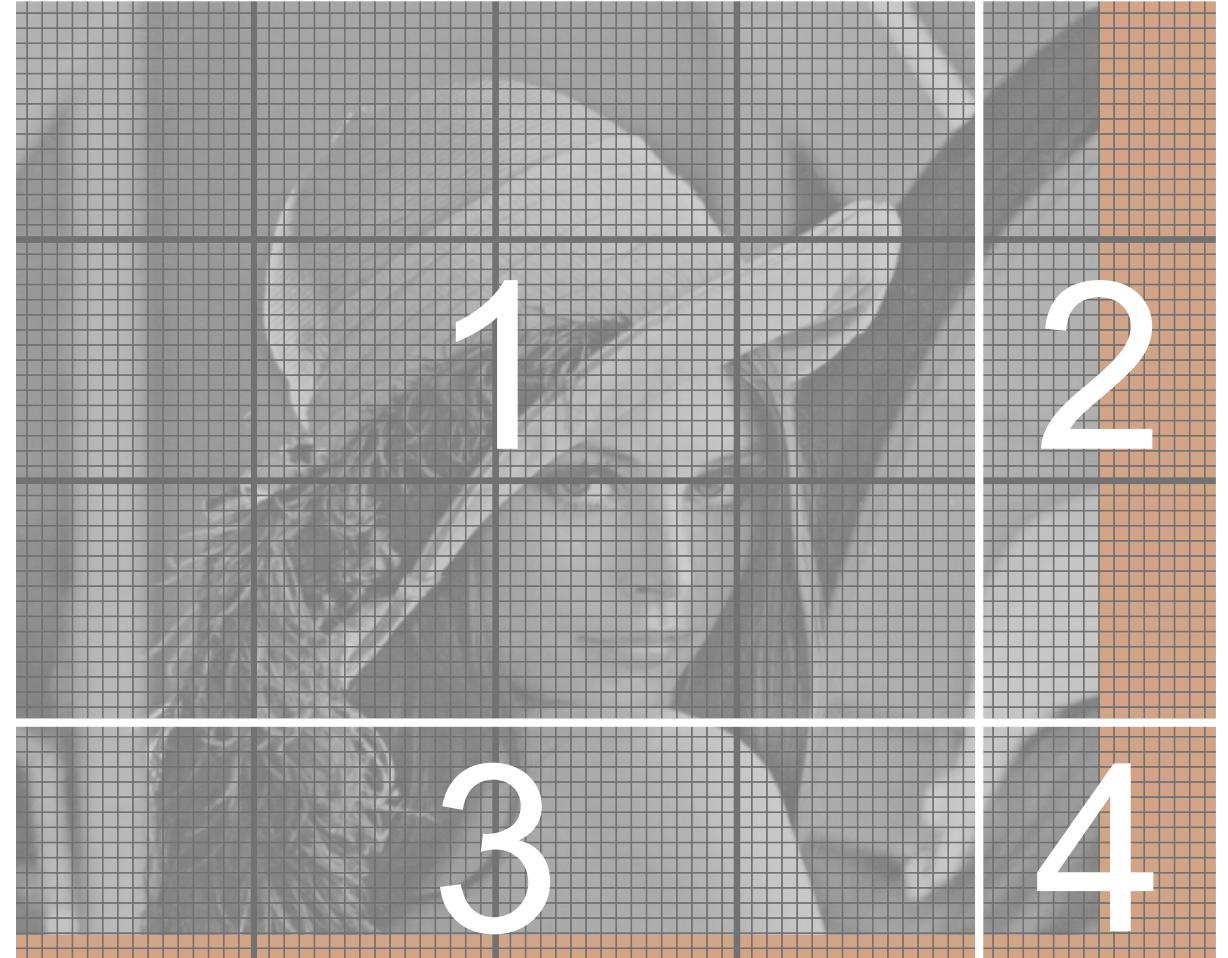


1 block:
16×16
threads
per block

Control flow divergence

- not all threads in a block will follow the same control flow path
- 4 different paths in this case

62×76 picture



CUDA programming

Processing a Picture with a 2D Grid

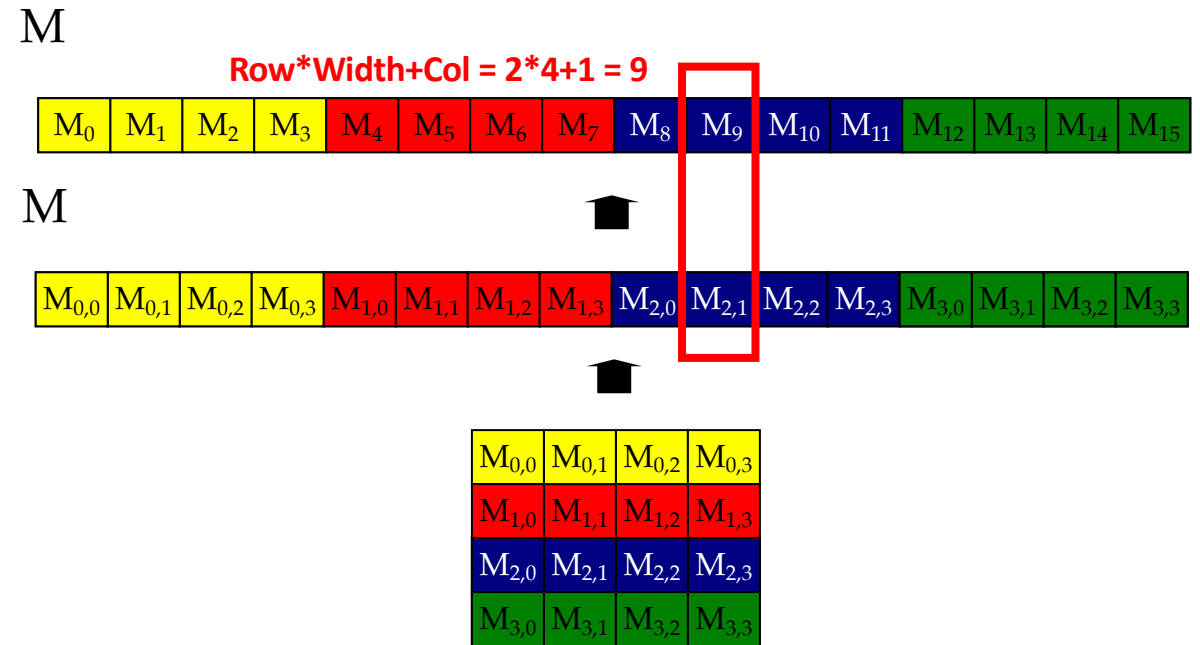
Kernel

```
__global__ void PictureKernel(float* d_Pin,
                             float* d_Pout,
                             int height,
                             int width)
{
    // Calculate the row # of
    // the d_Pin and d_Pout element
    int Row = blockIdx.y*blockDim.y + threadIdx.y;

    // Calculate the column # of
    // the d_Pin and d_Pout element
    int Col = blockIdx.x*blockDim.x + threadIdx.x;

    // each thread computes one
    // element of d_Pout if in range
    if ((Row < height) && (Col < width)) {
        d_Pout[Row*width+Col] = 2.0*d_Pin[Row*width+Col];
    }
}
```

Row-Major Layout in C/C++



CUDA programming

Processing a Picture with a 2D Grid

Kernel

```
__global__ void PictureKernel(float* d_Pin,
                             float* d_Pout,
                             int height,
                             int width)
{
    // Calculate the row # of
    // the d_Pin and d_Pout element
    int Row = blockIdx.y*blockDim.y + threadIdx.y;

    // Calculate the column # of
    // the d_Pin and d_Pout element
    int Col = blockIdx.x*blockDim.x + threadIdx.x;

    // each thread computes one
    // element of d_Pout if in range
    if ((Row < height) && (Col < width)) {
        d_Pout[Row*width+Col] = 2.0*d_Pin[Row*width+Col];
    }
}
```

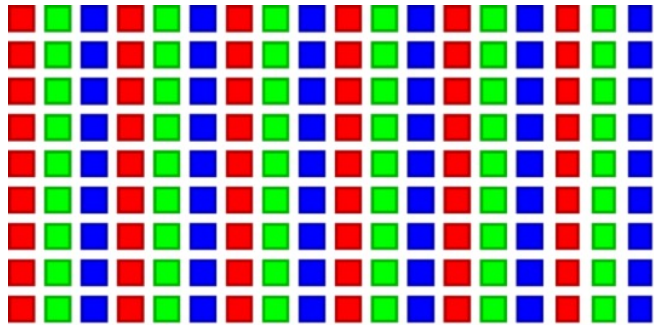
Host Code for Launching 2D kernel

- assume that the picture is $m \times n$,
- m pixels in y dimension and n pixels in x dimension
- input d_Pin has been allocated on and copied to device
- output d_Pout has been allocated on device

```
dim3 DimGrid((n-1)/16 + 1, (m-1)/16+1, 1);
dim3 DimBlock(16, 16, 1);
PictureKernel<<<DimGrid,DimBlock>>>(d_Pin, d_Pout, m, n);
```

CUDA programming

Converting color image to grayscale



RGB color image

- 3 values per pix
 - r - red
 - g - green
 - b - blue

Grayscale image

- only intensity



$$\text{grayPixel}[I,J] = 0.21*r + 0.71*g + 0.07*b$$

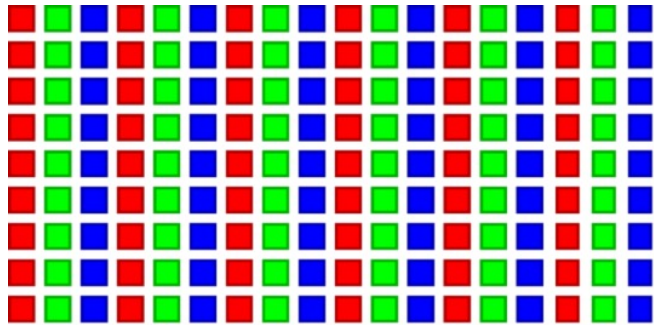
RGB Kernel:

```
// we have 3 channels corresponding to RGB  
// The input image is encoded as unsigned characters [0, 255]  
__global__ void colorConvert(unsigned char * grayImage,  
                             unsigned char * rgbImage,  
                             int width, int height) {  
  
    int col = threadIdx.x + blockIdx.x * blockDim.x;  
    int row = threadIdx.y + blockIdx.y * blockDim.y;  
  
    if (col < width && row < height) {  
        // get 1D coordinate for the grayscale image  
        int grayOffset = row*width + col;  
  
    }  
}
```

Host code for launching the kernel is the same as in previous slide.

CUDA programming

Converting color image to grayscale



RGB color image

- 3 values per pix
 - r - red
 - g - green
 - b - blue

Grayscale image

- only intensity



$$\text{grayPixel}[I,J] = 0.21*r + 0.71*g + 0.07*b$$

RGB Kernel:

```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__ void colorConvert(unsigned char * grayImage,
                             unsigned char * rgbImage,
                             int width, int height) {

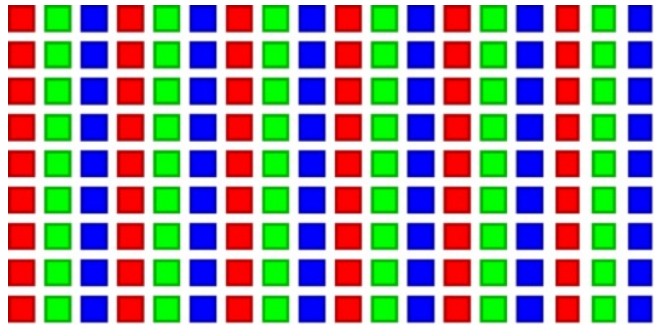
    int col = threadIdx.x + blockIdx.x * blockDim.x;
    int row = threadIdx.y + blockIdx.y * blockDim.y;

    if (col < width && row < height) {
        // get 1D coordinate for the grayscale image
        int grayOffset = row*width + col;
        // one can think of the RGB image having
        // CHANNEL times columns than the gray scale image
        int  rgbOffset = grayOffset*CHANNELS;
        unsigned char r = rgbImage[rgbOffset + 0]; // red value for pix
        unsigned char g = rgbImage[rgbOffset + 1]; // green value for pix
        unsigned char b = rgbImage[rgbOffset + 2]; // blue value for pix
    }
}
```

Host code for launching the kernel is the same as in previous slide.

CUDA programming

Converting color image to grayscale



RGB color image

- 3 values per pix
 - r - red
 - g - green
 - b - blue

Grayscale image

- only intensity



$$\text{grayPixel}[I,J] = 0.21*r + 0.71*g + 0.07*b$$

RGB Kernel:

```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__ void colorConvert(unsigned char * grayImage,
                             unsigned char * rgbImage,
                             int width, int height) {

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

    if (col < width && row < height) {
        // get 1D coordinate for the grayscale image
        int grayOffset = row*width + col;
        // one can think of the RGB image having
        // CHANNEL times columns than the gray scale image
        int rgbOffset = grayOffset*CHANNELS;
        unsigned char r = rgbImage[rgbOffset + 0]; // red value for pix
        unsigned char g = rgbImage[rgbOffset + 1]; // green value for pix
        unsigned char b = rgbImage[rgbOffset + 2]; // blue value for pix
        // perform the rescaling and store it
        // We multiply by floating point constants
        grayImage[grayOffset] = (unsigned char)(0.21f*r + 0.71f*g + 0.07f*b);
    }
}
```

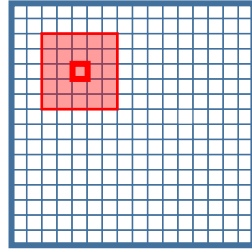
Host code for launching the kernel is the same as in previous slide.

CUDA programming

Image Blur

Blur Filter

- calculates average value inside the mask
 - BLUR_SIZE** value



1 block:
16×16
threads
per block

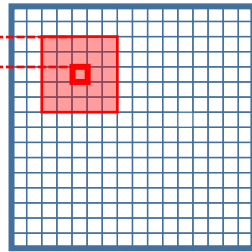


BlurPixel[I,J] = Average value of all pixel in a mask

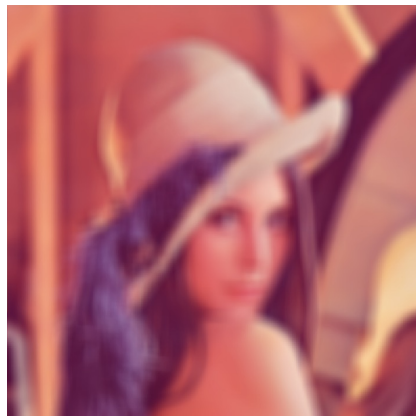


Hands-on Image Blur

BLUR_SIZE=2



1 block:
16×16
threads
per block



**BlurPixel[I,J] = Average value of all pixel in
a mask**

```
// we have 1 channel, therefore a grayscale image
// The input image is encoded as unsigned characters [0, 255]
__global__ void BlurKernel(unsigned char * inImage,
                           unsigned char * outImage,
                           int width, int height) {

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

    if (col < width && row < height) {
        int pixVal = 0; int pixels = 0;

        // Get the average of the surrounding 2xBLUR_SIZE x 2xBLUR_SIZE box
        for(int blurRow = row - BLUR_SIZE; blurRow <= row + BLUR_SIZE; blurRow++) {
            for(int blurCol = col - BLUR_SIZE; blurCol <= col + BLUR_SIZE; blurCol++) {

                int curRow = row + blurRow - row;
                int curCol = col + blurCol - col;

                // Verify we have a valid image pixel
                if(curRow > 0 && curRow < height && curCol > 0 && curCol < width) {
                    pixVal = inImage[curRow * width + curCol];
                    pixels++; // Total number of accumulated pixels
                }
            }
        }

        // Write our new pixel value out
        outImage[col * height + row] = (unsigned char)(pixVal / pixels);
    }
}
```


Hands-on Image Blur

Univerza v Ljubljani



TECHNISCHE
UNIVERSITÄT
WIEN



VSB TECHNICAL
UNIVERSITY
OF OSTRAVA

IT4INNOVATIONS
NATIONAL SUPERCOMPUTING
CENTER



Co-funded by the
Erasmus+ Programme
of the European Union

This project has been funded with support from the European Commission.

This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

- Finish the missing code in the kernel on the previous slide
- Source code in `04_image_blur/<lang>/Task/image_blur.<ext>`
- Tasks are annotated with TODO, only in the kernel
- No actual image (to simplify the code), just some pattern which is easy to check for correctness
- Compile and run with
 - `nvcc image_blur.cu -o image_blur.x && ./image_blur.x`
 - `nvfortran image_blur.CUF -o image_blur.x && ./image_blur.x`

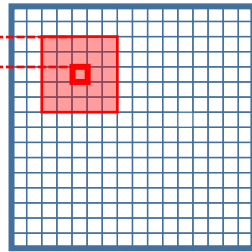
Correct output:

Everything seems OK

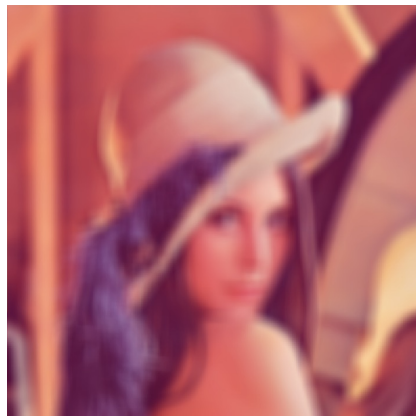
CUDA programming

Image Blur - Solution

BLUR_SIZE=2



1 block:
16x16
threads
per block



**BlurPixel[i,J] = Average value of all pixel in
a mask**

```
// we have 1 channel, therefore a grayscale image
// The input image is encoded as unsigned characters [0, 255]
__global__ void BlurKernel(unsigned char * inImage,
                           unsigned char * outImage,
                           int width, int height) {

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

    if (col < width && row < height) {
        int pixVal = 0; int pixels = 0;

        // Get the average of the surrounding 2xBLUR_SIZE x 2xBLUR_SIZE box
        for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {
            for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol) {
                int curRow = row + blurRow;
                int curCol = col + blurCol;

                // Verify we have a valid image pixel
                if(curRow > -1 && curRow < height && curCol > -1 && curCol < width) {
                    pixVal = pixVal + inImage[curRow * width + curCol];
                    pixels = pixels + 1; // Total number of accumulated pixels
                }
            }
        }

        // Write our new pixel value out
        outImage[Row * width + Col] = (unsigned char)(pixVal / pixels);
    }
}
```

Thread Execution

Univerza v Ljubljani



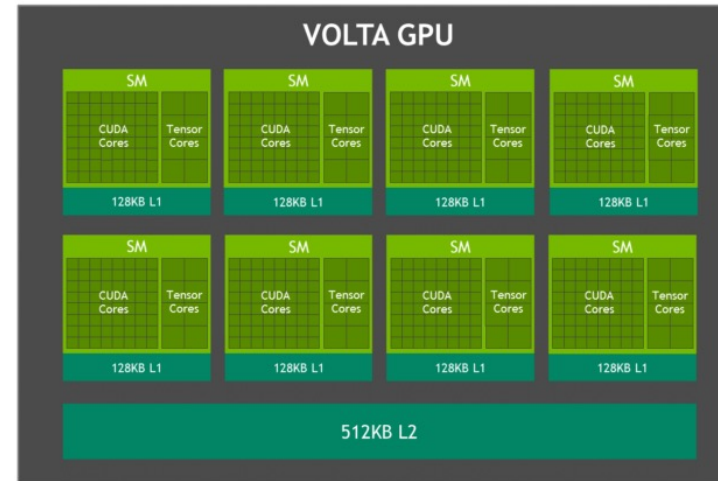
Co-funded by the
Erasmus+ Programme
of the European Union

This project has been funded with support from the European Commission.

This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Transparent scaling of GPU kernels

- Kernel execution is broken in Grid of Blocks
 - blocks can be executed in any order relative to others
 - hardware is free to assign blocks to any Streaming Multiprocessor (SM) at any time
 - **a kernel scales to any number of parallel processors**
- this property ensures correct execution on GPUs with
 - different number of Streaming Multiprocessors (different performance, different model of GPU accelerators (A100, A40, ...))
 - different GPU architectures



NVIDIA Jetson AGX Xavier

- ARM based embedded single board computer with on-chip GPU
- GPU with 8 SMs



NVIDIA V100

- HPC accelerator
- GPU with 80 SMs

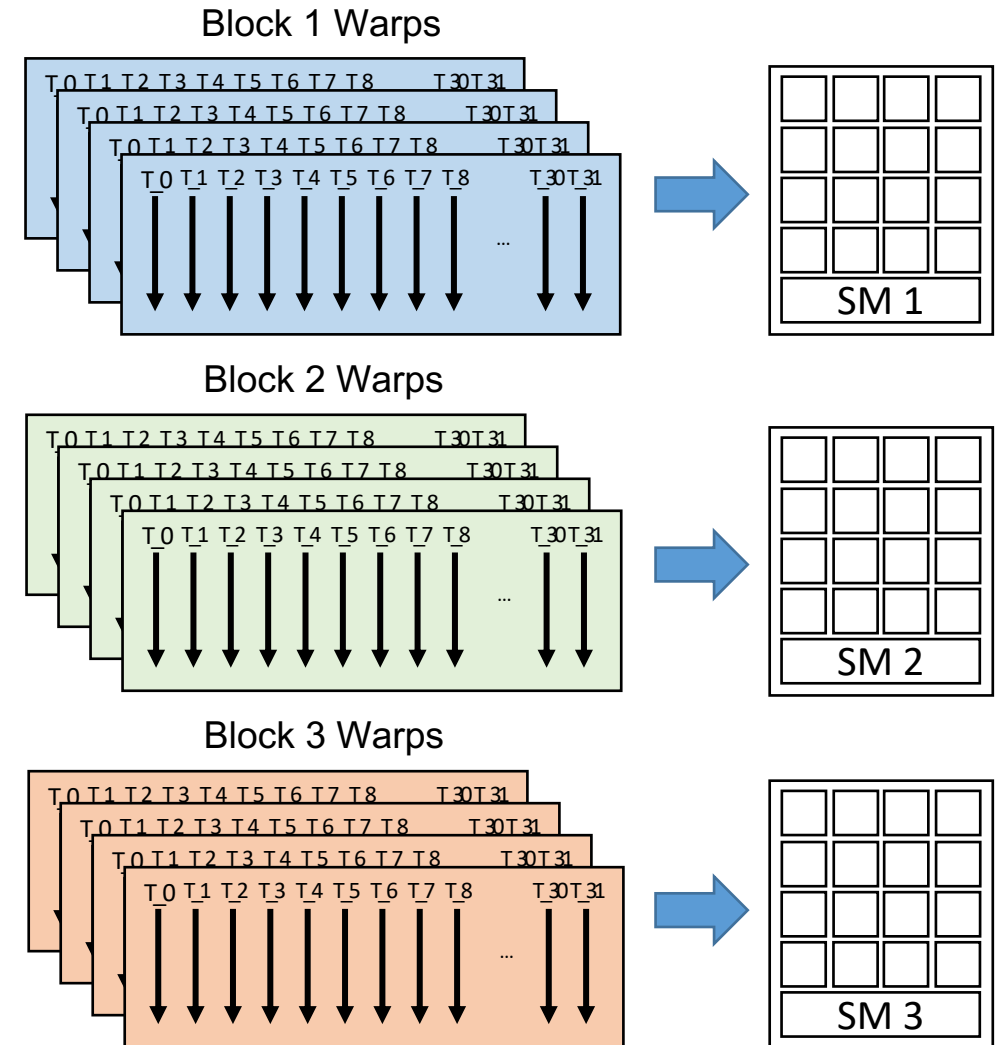
Thread Execution and Warps

Thread Execution

- blocks are assigned to Streaming Multiprocessors (SM)
 - up to 32 blocks can be assigned to one SM as resources allow
 - Ampere generation SM can take up to 2048 threads
 - could be $256 \text{ (threads/block)} * 8 \text{ blocks}$
 - or $512 \text{ (threads/block)} * 4 \text{ blocks, etc.}$
- SM maintains thread/block idx #s
- SM manages/schedules thread execution

Warps as Scheduling Units

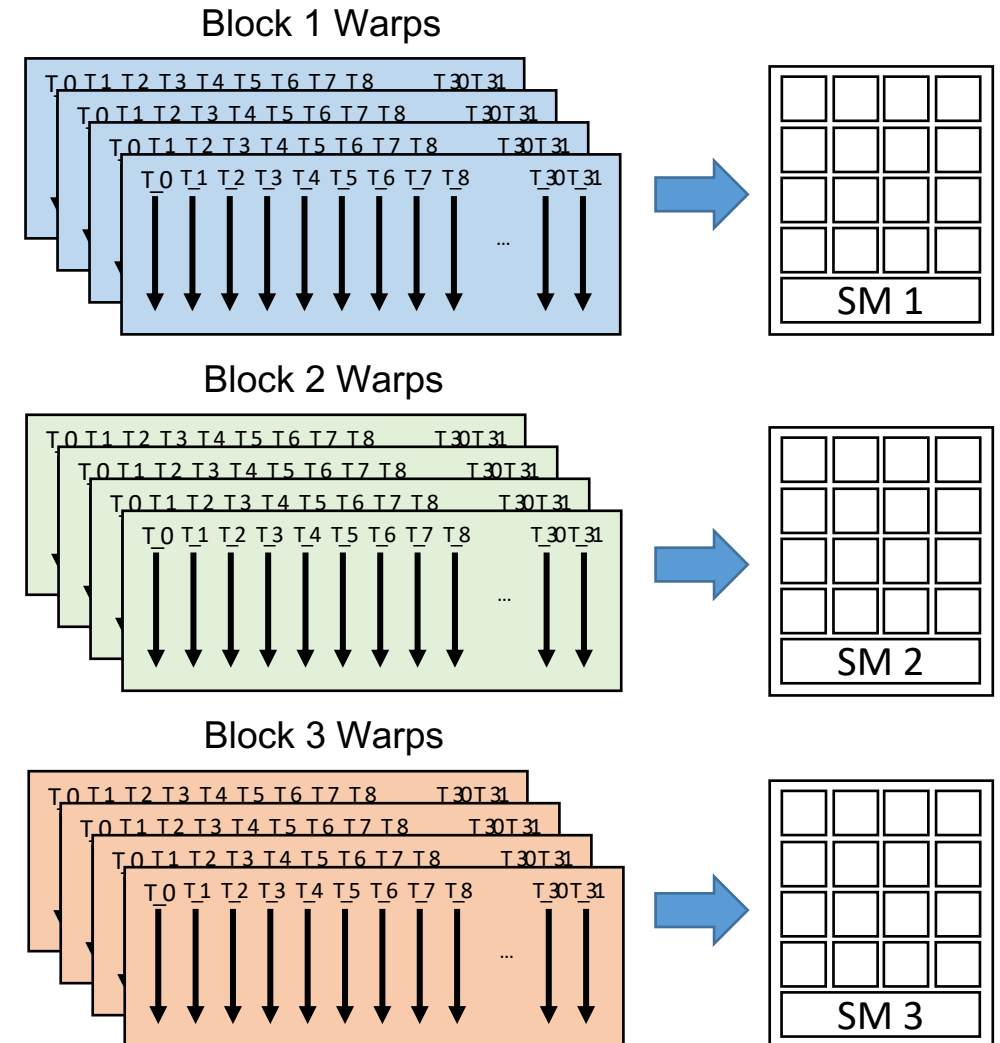
- each Block is divided and executed as 32-thread Warps
 - an implementation decision, not part of the CUDA programming model
- warps are scheduling units in SM
- threads in a warp execute in SIMD fashion
- future GPUs may have different number of threads in each warp
 - for instance, AMD GPUs have warp size 64 threads



Thread Execution and Warps

Thread Execution cont.

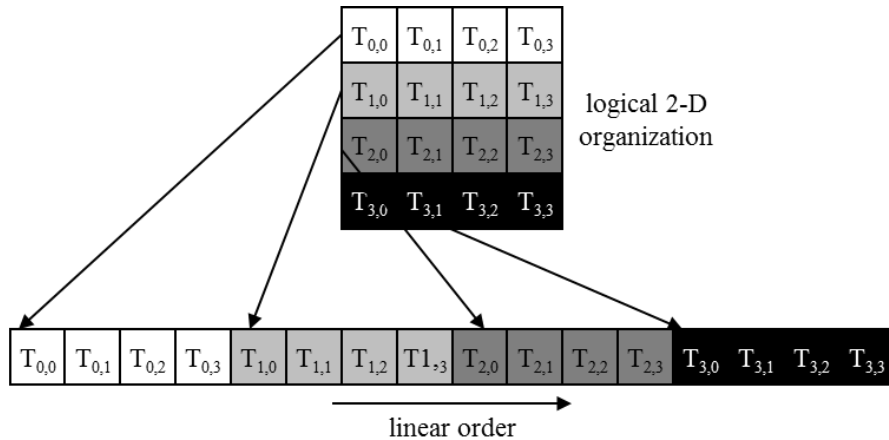
- SM implements zero-overhead warp scheduling
 - Warps whose next instruction has its operands ready for consumption are eligible for execution
 - Eligible Warps are selected for execution based on a prioritized scheduling policy
 - All threads in a warp execute the same instruction when selected



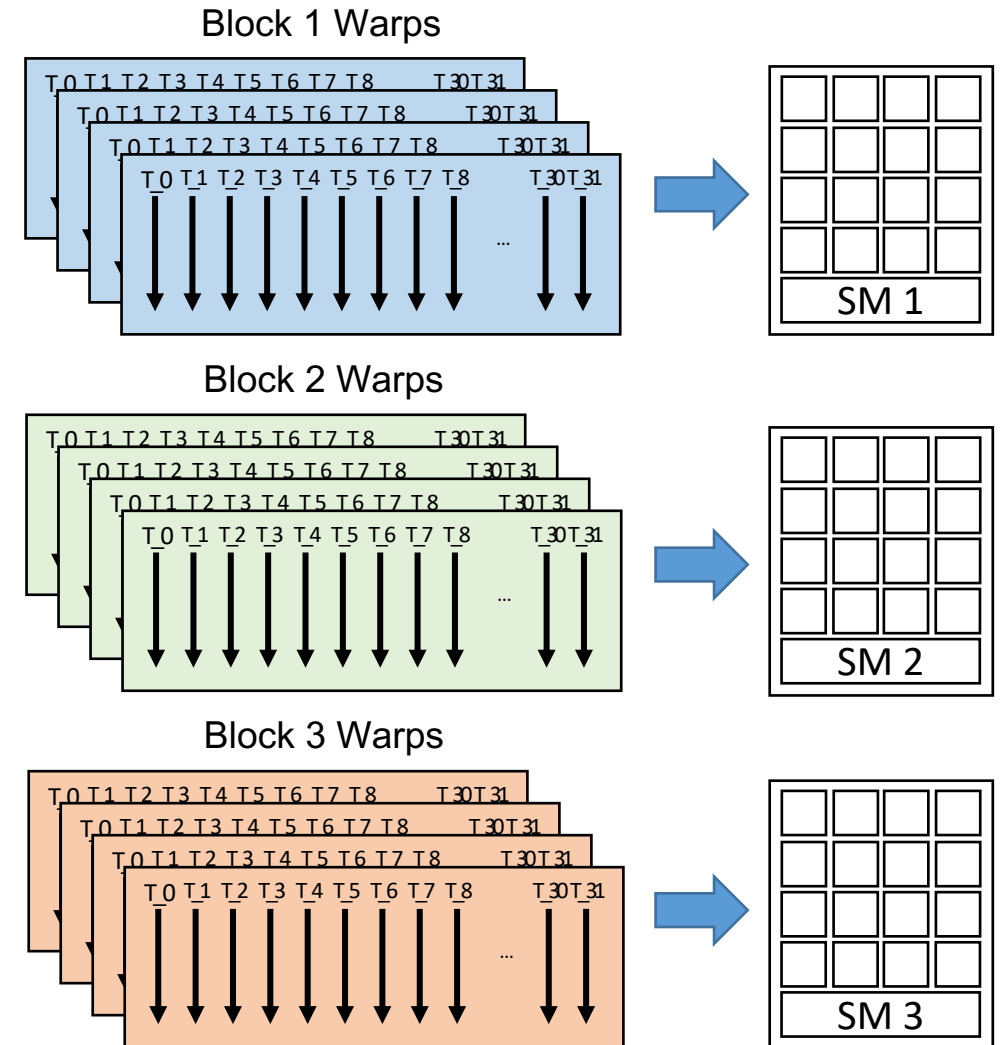
Thread Execution and Warps

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



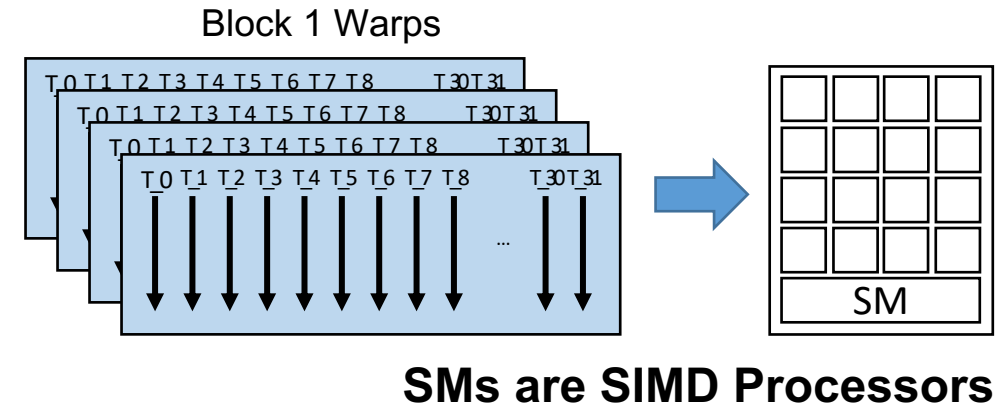
- Linearized thread blocks are partitioned in warps
 - Thread indices within a warp are consecutive and increasing
 - Warp 0 starts with Thread 0
- **DO NOT** rely on any ordering within or between warps
 - If there are any dependencies between threads, you must `__syncthreads()` to get correct results (more later)



Thread Execution and Warps

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



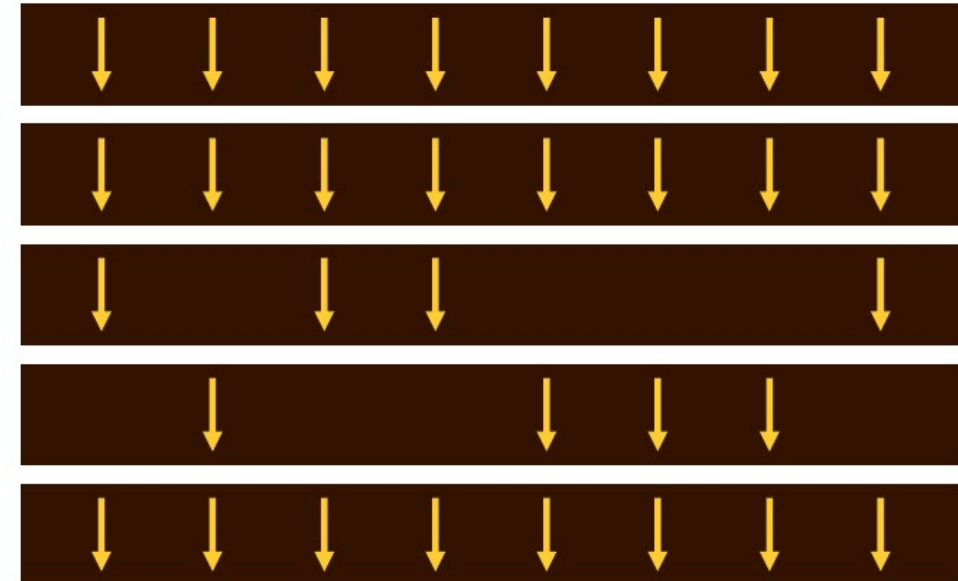
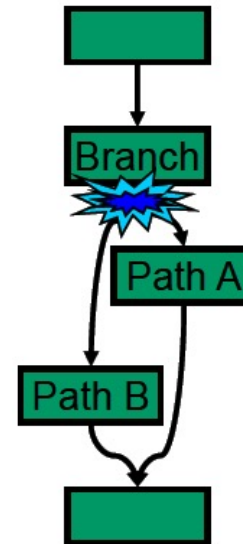
Example of a SIMD code:

```
__global__  
void vecAddKernel(float* A, float* B, float* C, int n)  
{  
    int i = threadIdx.x + blockDim.x * blockIdx.x;  
  
    C[i] = A[i] + B[i];  
}
```

Control Divergence

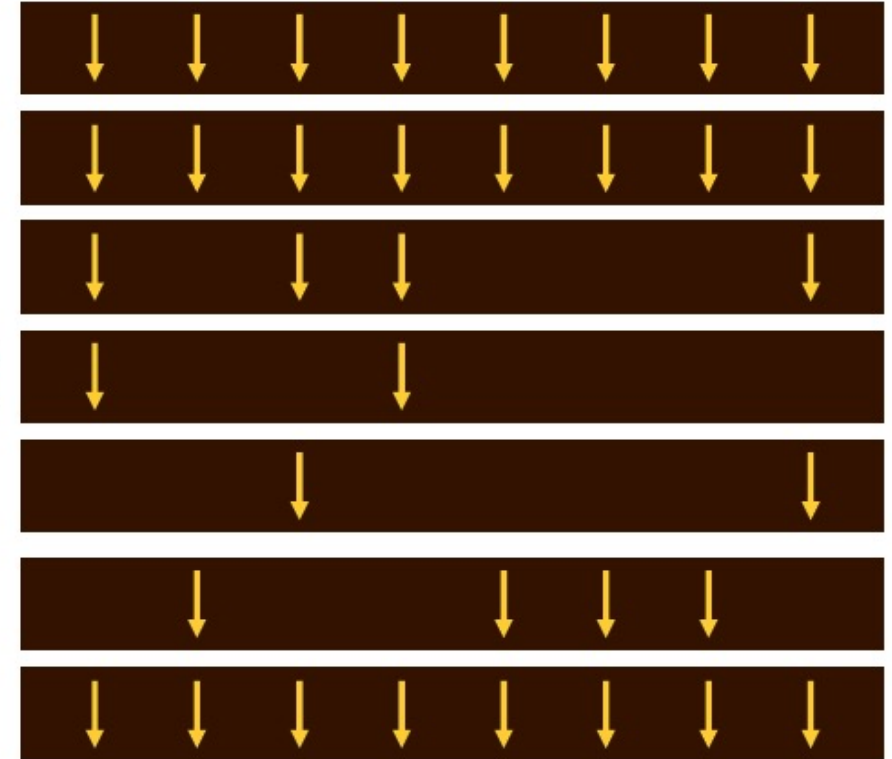
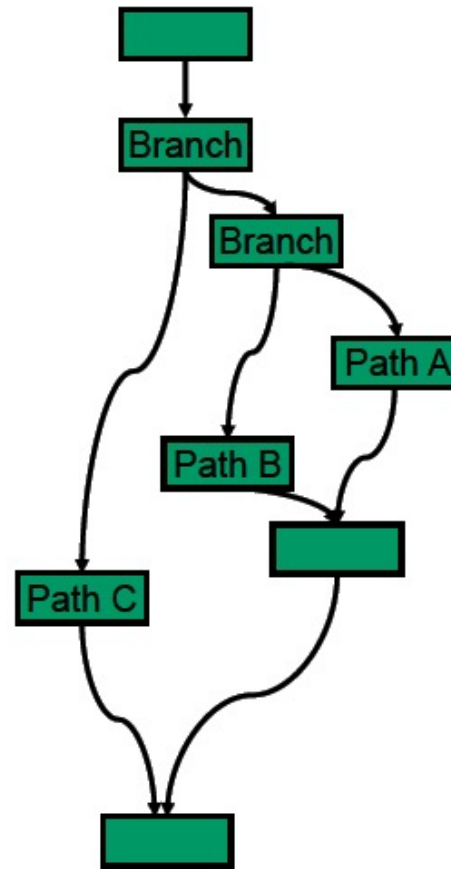
- control divergence occurs when threads in a warp take different control flow paths by making different control decisions
 - some take the then-path and others take the else-path of an if-statement
 - some threads take different number of loop iterations than others
- The execution of threads taking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

```
if (foo(threadIdx.x))  
{  
    do_A();  
}  
else  
{  
    do_B();  
}
```



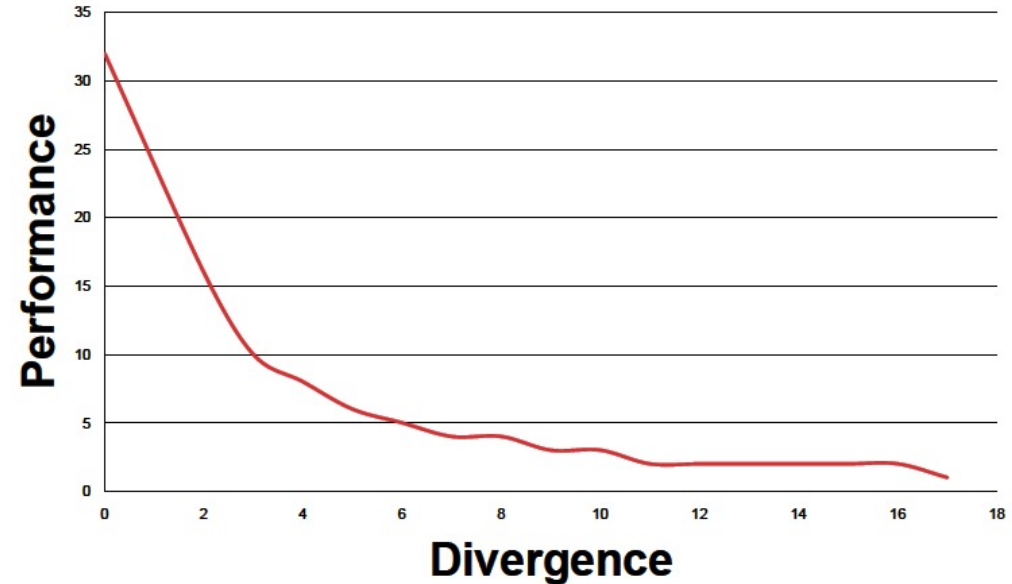
Control Divergence

- 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**



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**



The control diverges is problem only among threads within a warp.

The control divergence among warps is perfectly fine as long as all threads within a warp execute the same instruction.

Divergence can arise when branch or loop condition is a function of thread indices

```
__global__  
void vecAddKernel(float* A, float* B, float* C, int n)  
{  
    int i = threadIdx.x + blockDim.x * blockIdx.x;  
  
    if (i < n) C[i] = A[i] + B[i];  
}
```

CUDA Memories

Univerza v Ljubljani



TECHNISCHE
UNIVERSITÄT
WIEN



VSB TECHNICAL
UNIVERSITY
OF OSTRAVA

IT4INNOVATIONS
NATIONAL SUPERCOMPUTING
CENTER

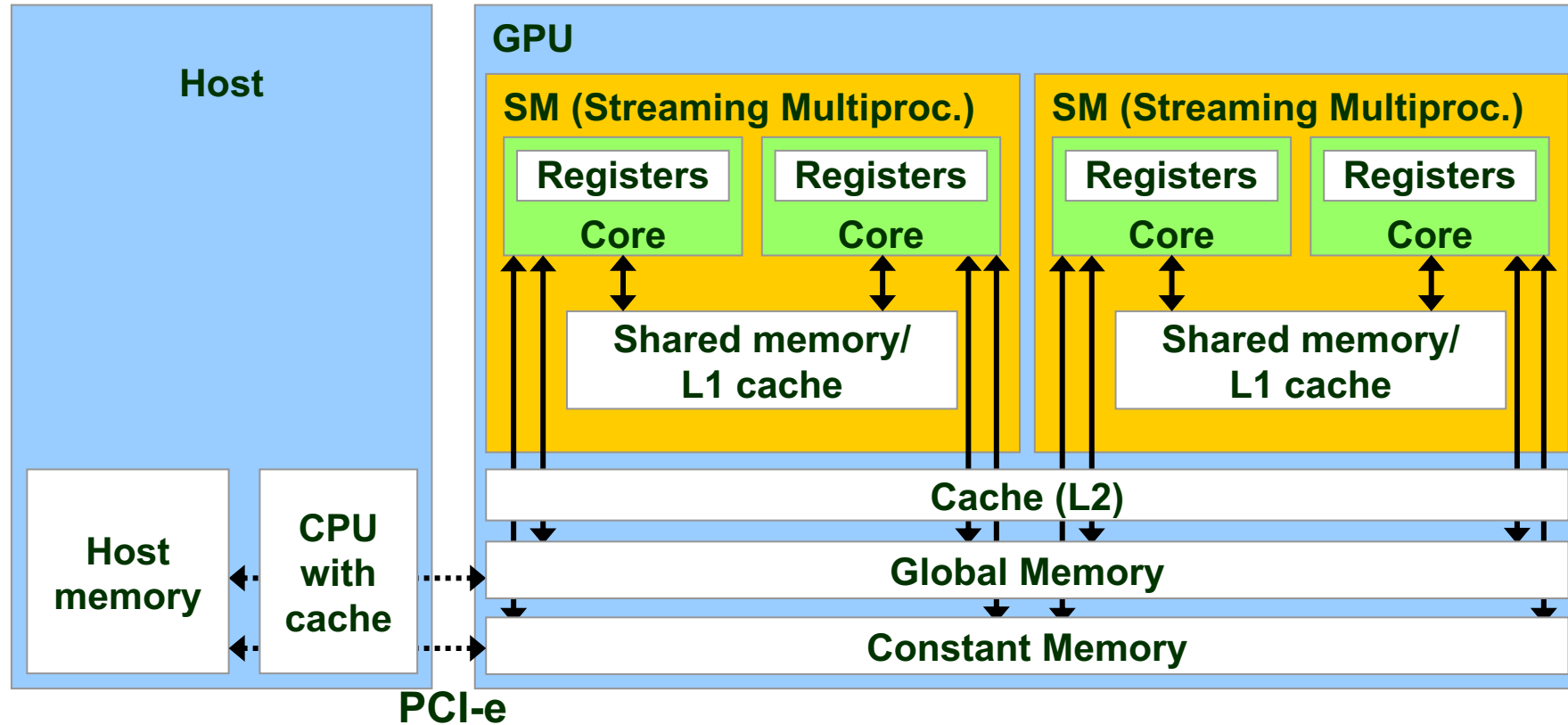


Co-funded by the
Erasmus+ Programme
of the European Union

This project has been funded with support from the European Commission.

This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

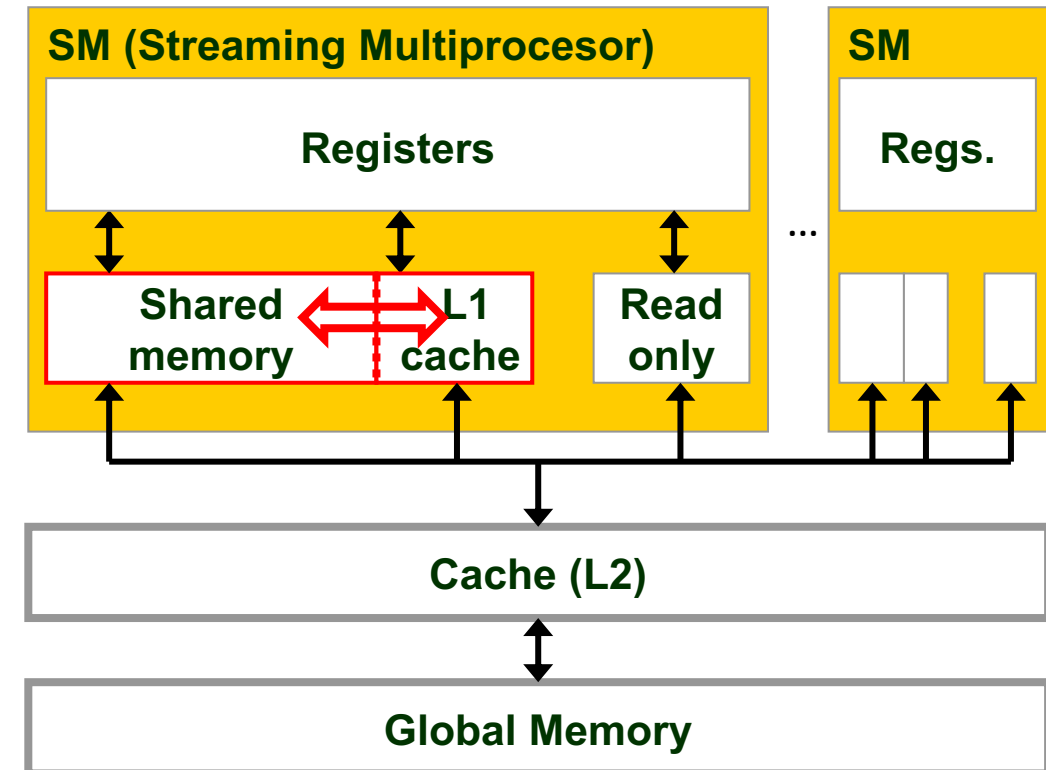
CUDA Memories Hardware View



CUDA Memories Hardware View

Memory hierarchy in Ampere generation (GA100)

- Registers
 - 256 kB per SM
 - Storage local to each threads
- Shared memory / L1 (192KB total)
 - configurable up to 164KB for SM;
 - remainder for L1 Cache
 - low latency: ~22 cycles (SM), 34 cycles (L1d)
 - high bandwidth: ~18 TB/s
- Read-only cache
 - Up to 128 kB per SM
- L2 - 40 MB
 - latency: ~ 200 or 350 cycles
 - BW: ~ 7000 GB/s
- Global memory – 40 or 80 GB HBM2
 - BW ~ 1500 GB/s



CUDA Memories Caches

Why do GPU have caches?

In general, not for cache blocking

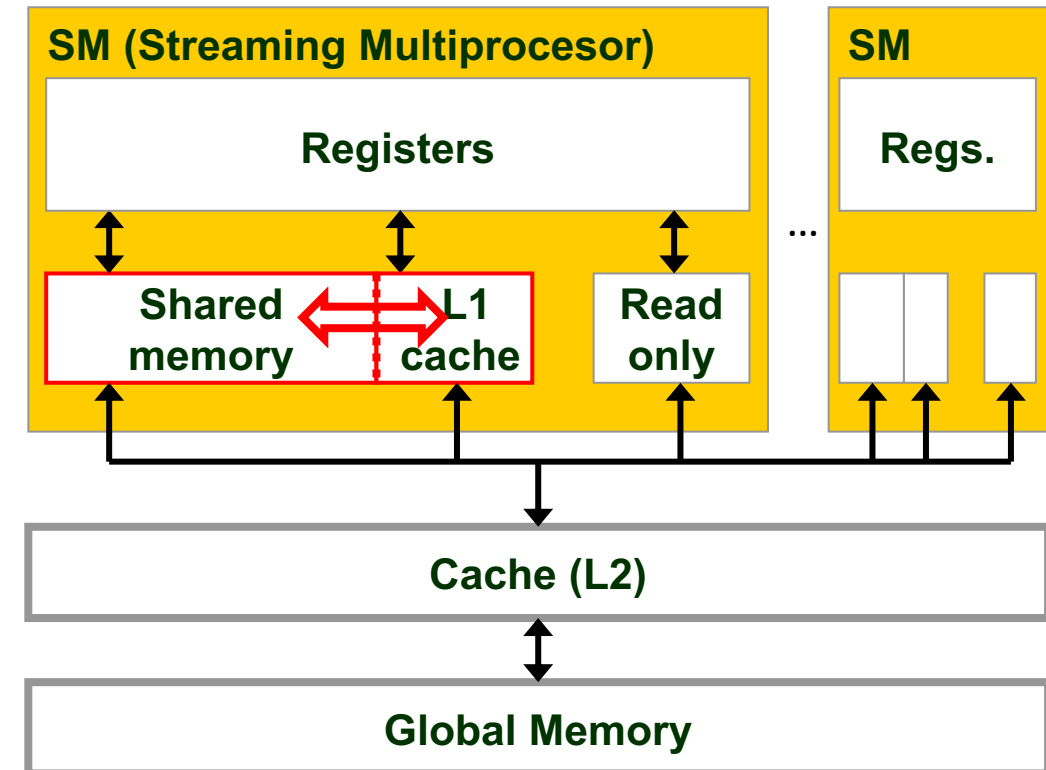
- 100s ~ 1000s of threads running per SM
- tens of thousands of threads sharing the L2 cache
- L1, L2 are small per thread.
- **Example:** at 2048 threads/SM, with 80 SMs:
 - 64 bytes L1,
 - 38 Bytes L2 per thread

Shared Memory is usually better option to cache data explicitly:

- user managed, no evictions out of your control.

Caches on GPUs are useful for:

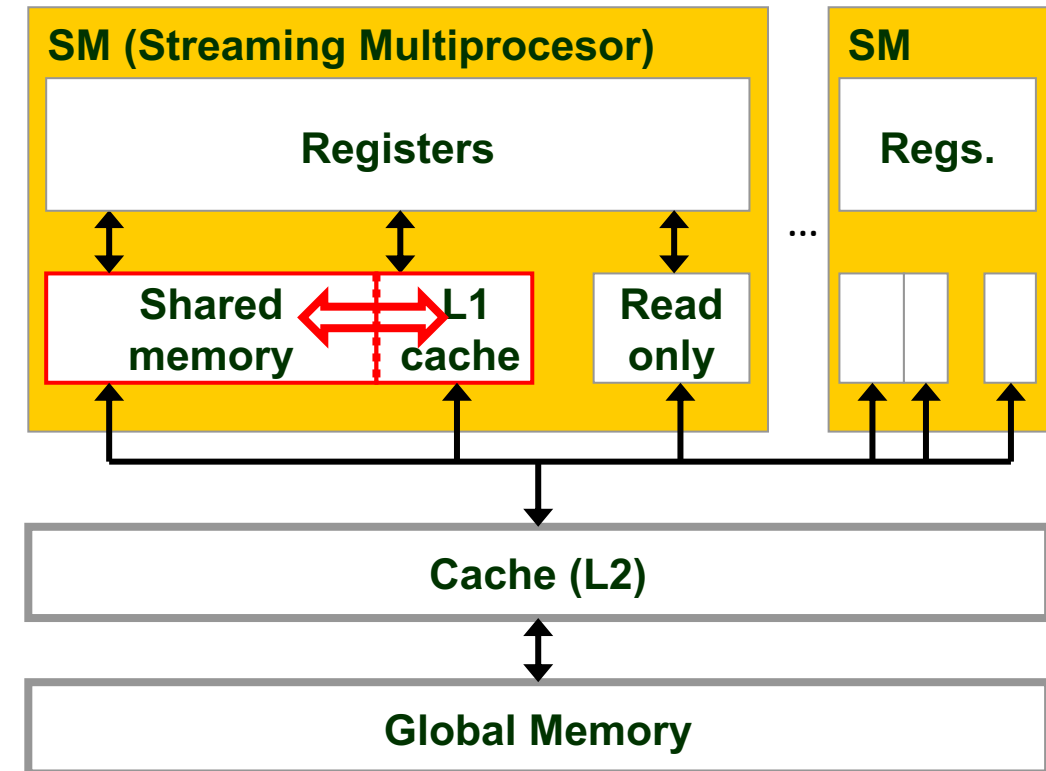
- “Smoothing” irregular, unaligned access patterns
- Caching common data accessed by many threads
- Faster register spills, local memory
- Fast atomics
- Codes that don’t use shared memory (naïve code, OpenACC, ...)



CUDA Memories Hardware View

Constant memory

- Read-only variables or arrays of global scope
- Qualified with `__constant__` keyword
- Capacity 64 KiB
- Cached in 8 KiB constant (read-only) cache
- Very fast if all threads within a warp read the same address
 - If the address is cached, throughput of constant cache
 - If not cached, throughput of device memory
- If different threads read different addresses, the accesses are serialized
- Example use: stencil coefficients



Global Memory

Univerza v Ljubljani



Co-funded by the
Erasmus+ Programme
of the European Union

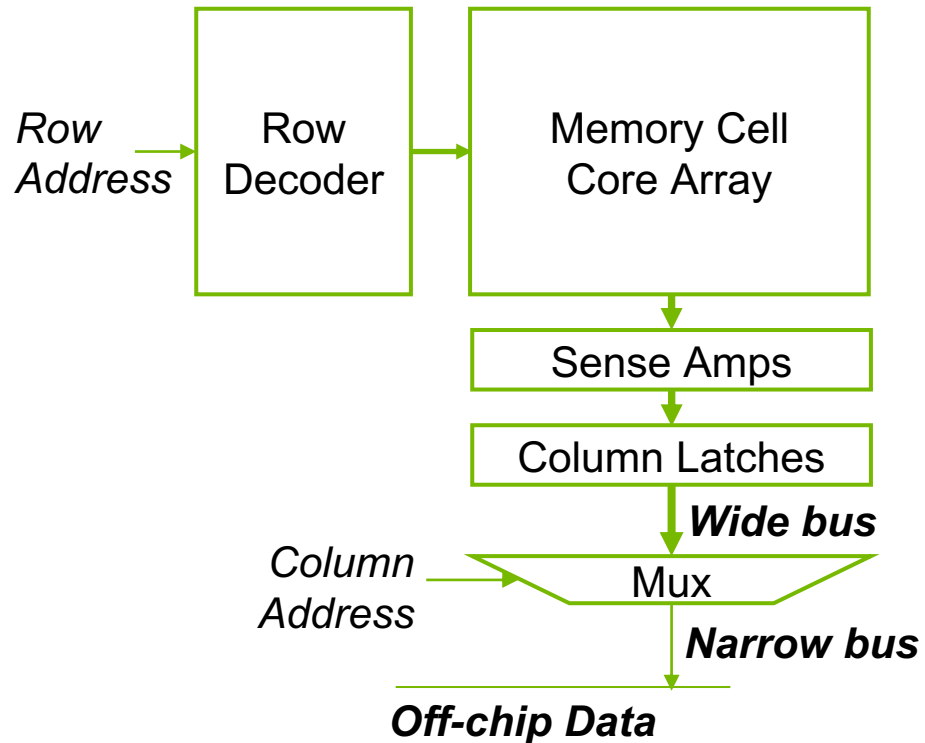
This project has been funded with support from the European Commission.
This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

CUDA Memories

Architecture of DRAM (Global) Memory

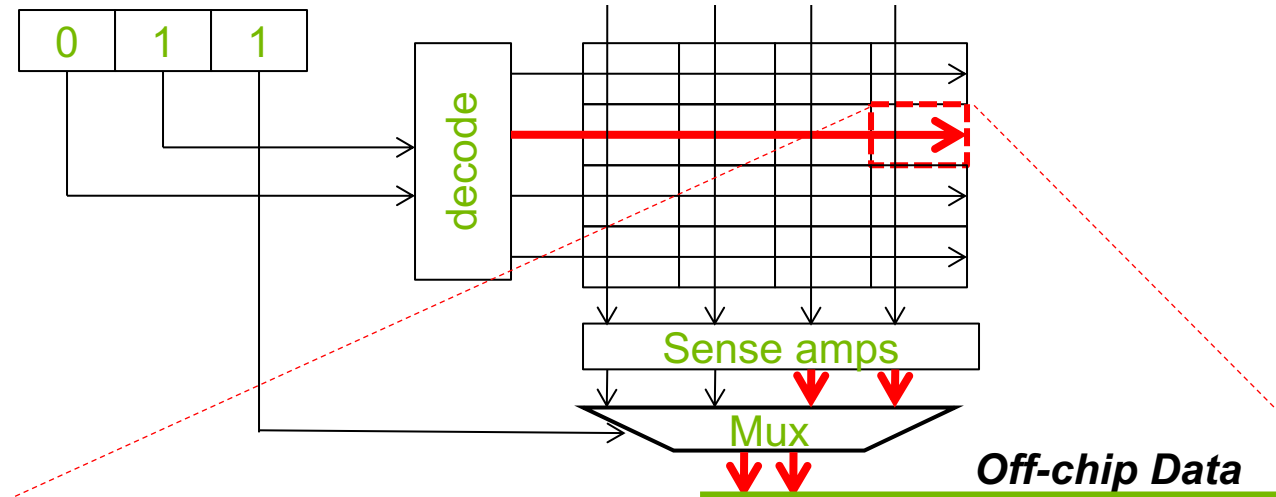
DRAM Core Array Organization

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

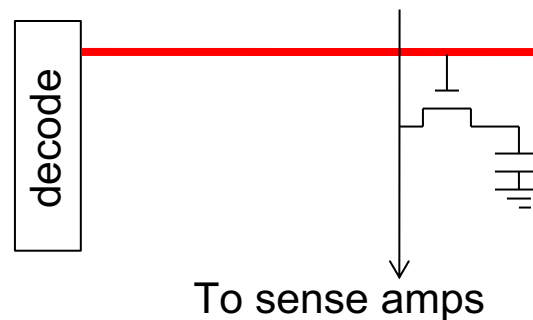


Example of a very small (8x2-bit) DRAM Core Array

- each bit is stored in a tiny capacitor made of one transistor



A cell in the core array



A very small capacitance that stores a data bit

Reading from a cell in the core array is a very slow process

- DDR3/GDDR4: Core speed = $\frac{1}{8}$ interface speed
- likely to be worse in the future

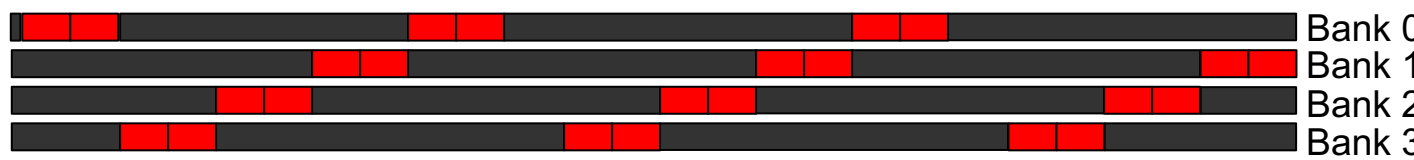
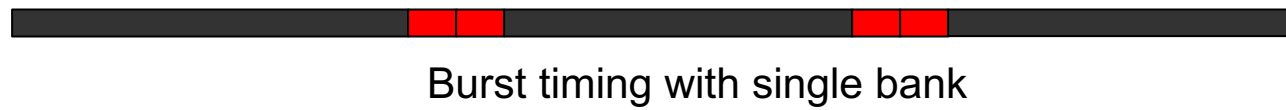
CUDA Memories

Architecture of DRAM (Global) Memory

DRAM Bursting

- For DDR{2,3,...} SDRAM the cores are clocked at 1/N speed of the interface
- DRAM Burst means to load ($N \times$ interface width) of DRAM bits from the same row at once to an internal buffer, then transfer in N steps at interface speed (i.e. DDR3/GDDR4: buffer width = 8X interface width)

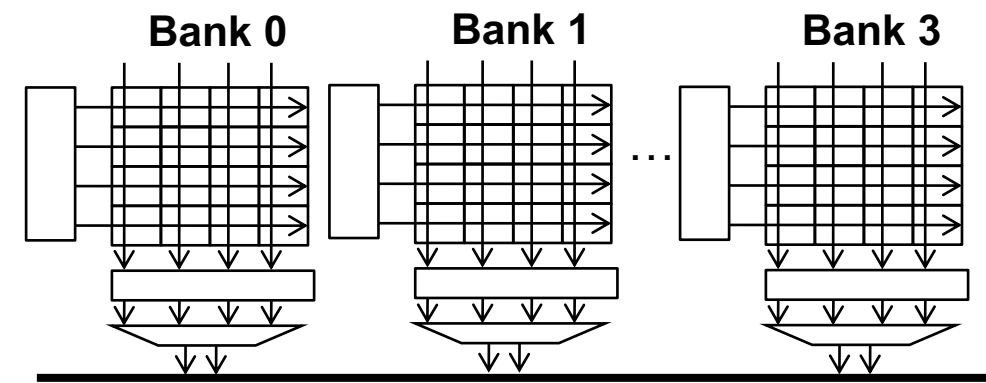
Address bits
to decoder



Multi-Bank burst timing, reduced dead time

- 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.

Multi-Bank



CUDA Memories

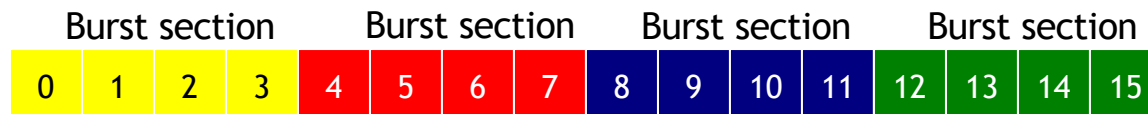
Global Memory Efficient Access

Memory Coalescing

- memory coalescing is important for effectively utilizing memory bandwidth in CUDA
 - its origin in DRAM burst
- for good performance CUDA memory access is coalesced

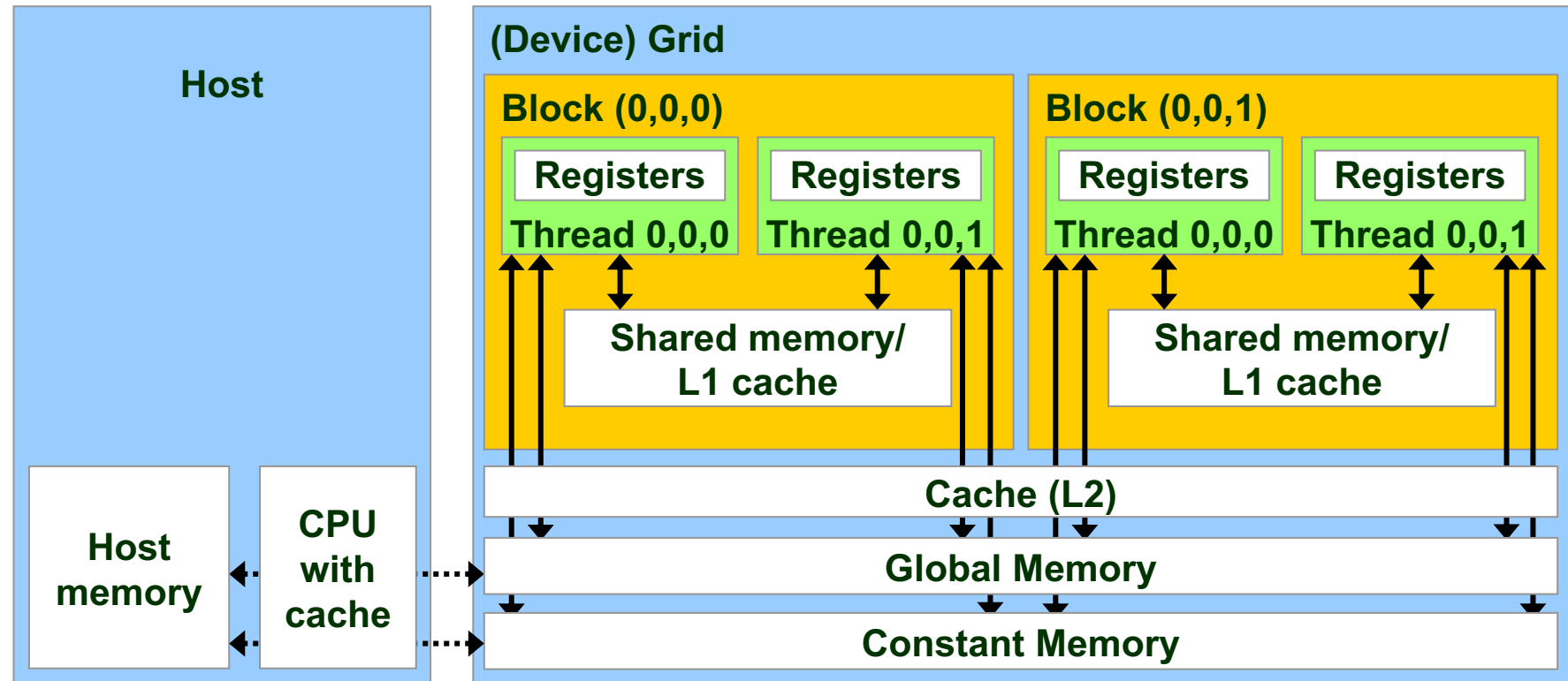
DRAM Burst – A System View

- 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



CUDA Memories Programmer View

- `__device__` is optional when used with `__shared__`, or `__constant__`
- Automatic variables reside in a register
 - Except per-thread arrays that reside in global memory



Variable declaration	Memory	Scope	Lifetime
<code>int LocalVar;</code>	register	thread	thread
<code>__device__ __shared__ int SharedVar;</code>	shared	block	block
<code>__device__ int GlobalVar;</code>	global	grid	application
<code>__device__ __constant__ int ConstantVar;</code>	constant	grid	application

CUDA Memories

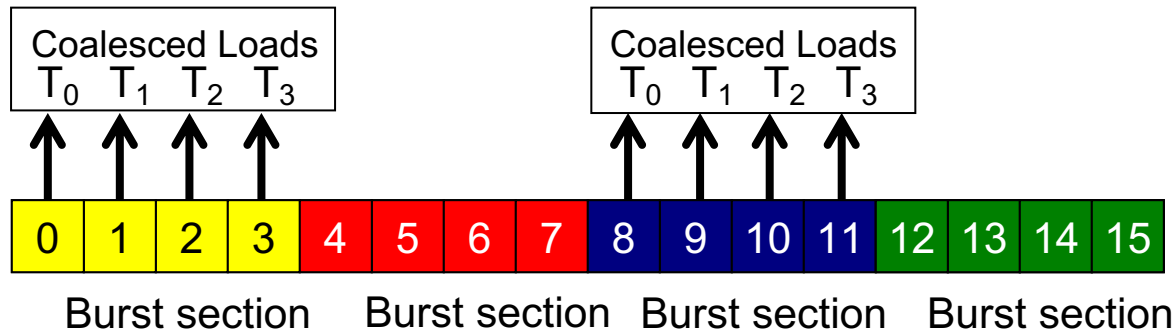
Global Memory Efficient Access

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.

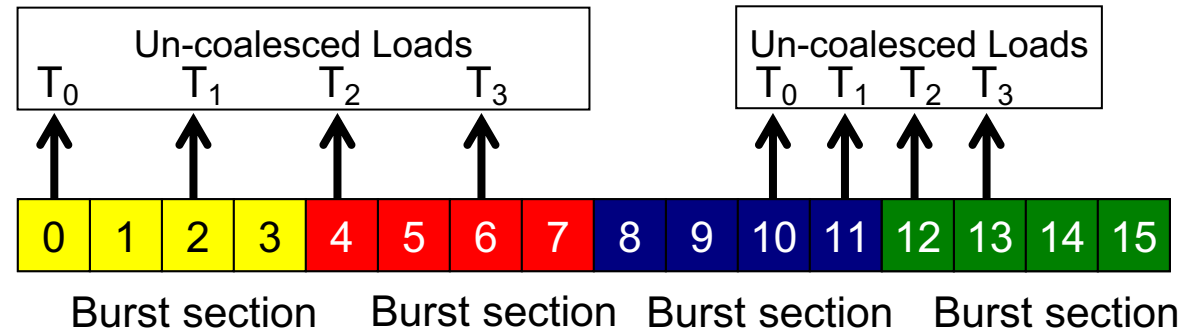
How to judge if an access is coalesced?

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



Un-coalesced Accesses

- 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

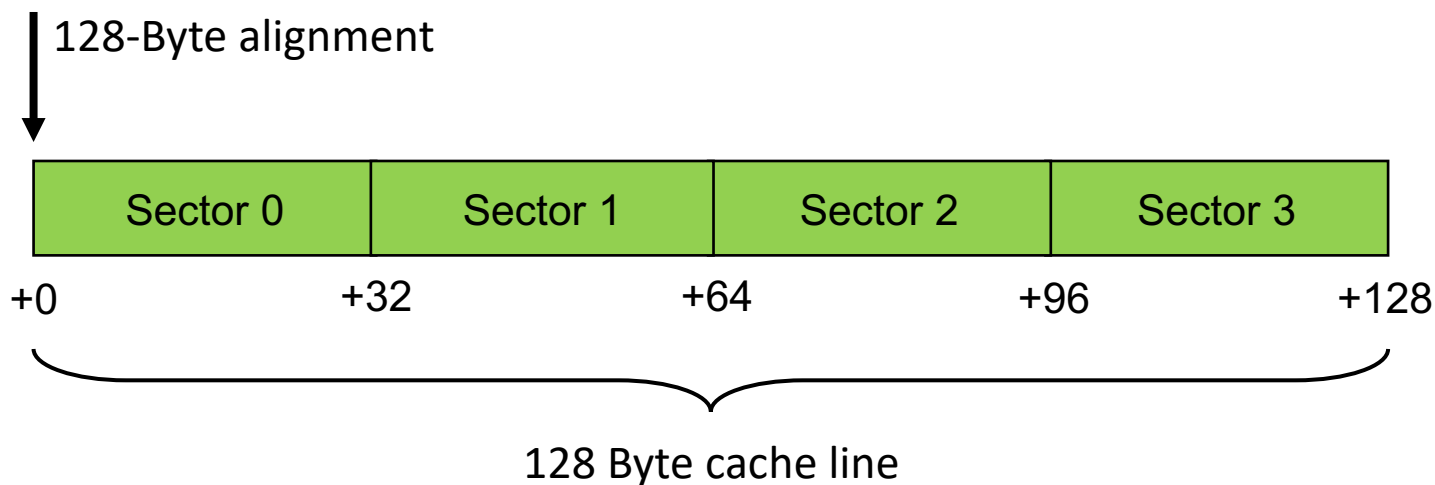


CUDA Memories

Global Memory Efficient Access

Cache lines and Sectors

- Moving data between L1, L2 and DRAM



Memory access granularity

- **32 Bytes – 1 sector**
 - for Maxwell and Pascal
- **Volta architecture**
 - **64 Bytes**
 - 2 sectors is default – second sector is prefetched
- **Ampere architecture**
 - **granularity can be set to**
 - **32, 64 and 128 Bytes**

Cache line size

- **128 Bytes** – made of 4 sectors

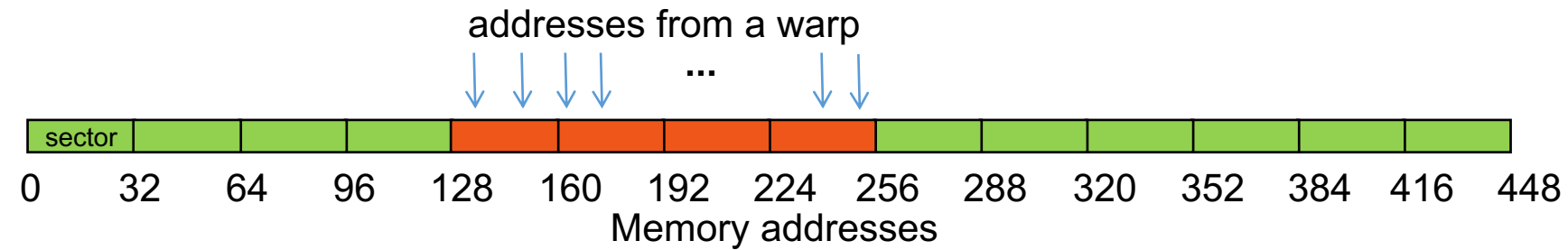
Cache management granularity

- 1 cache line

```
cudaDeviceSetLimit(cudaLimitMaxL2FetchGranularity, 32)
```

CUDA Memories

Global Memory Efficient Access

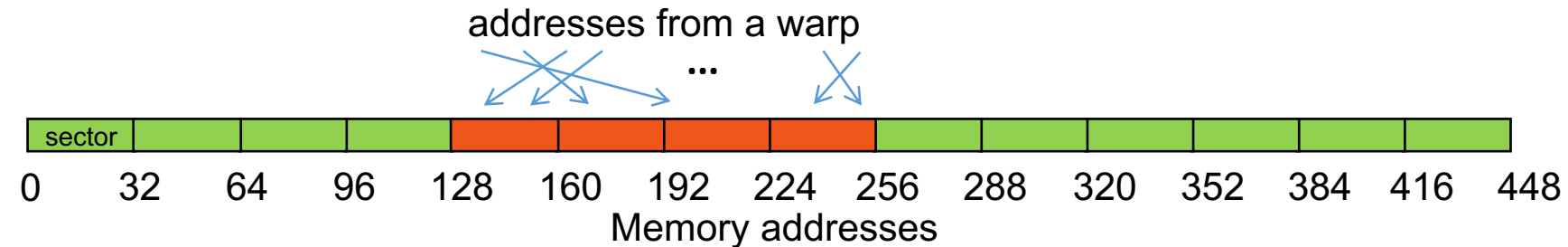


Scenario 1:

- Warp requests 32 aligned, **consecutive** 4-byte words

Addresses fall within 4 sectors

- Warp needs 128 bytes
- 128 bytes move across the bus
- **Bus utilization: 100%**



Scenario 2:

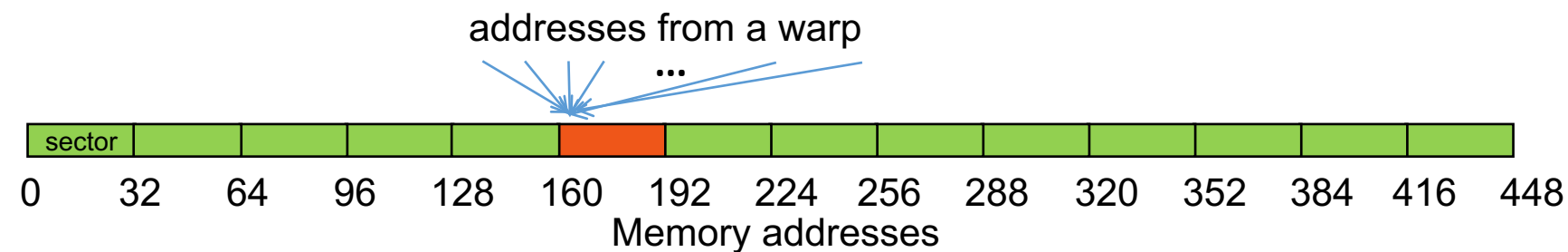
- Warp requests 32 aligned, **permuted** 4-byte words

Addresses fall within 4 sectors

- Warp needs 128 bytes
- 128 bytes move across the bus
- **Bus utilization: 100%**

CUDA Memories

Global Memory Efficient Access

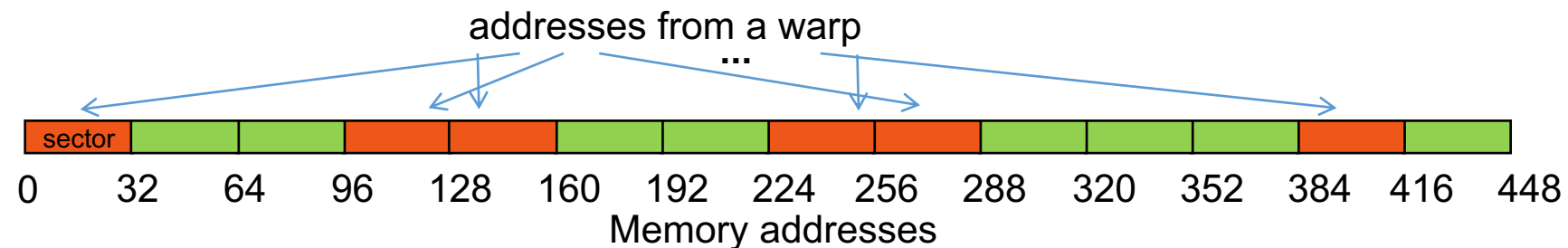


Scenario 3:

- All threads in a warp request the same 4-byte word

Addresses fall within 4 sectors

- Warp needs 4 bytes
- 32 bytes move across the bus
- **Bus utilization: 12.5%**



Scenario 4:

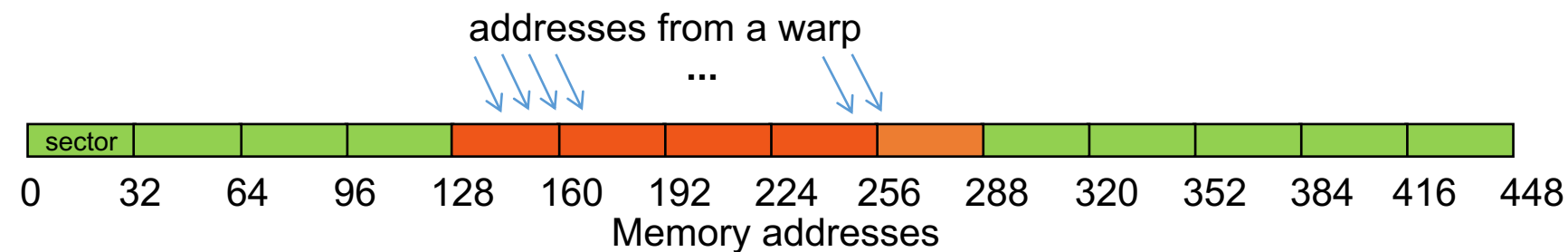
- Warp requests 32 scattered 4-byte words

Addresses fall within 4 sectors

- Warp needs 128 bytes
- $N*32$ bytes move across the bus
- **Bus utilization: $128 / (N*32)$**

CUDA Memories

Global Memory Efficient Access

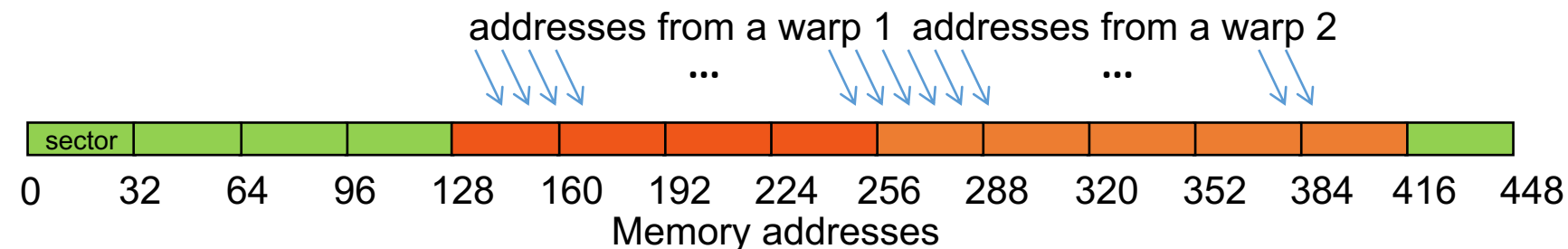


Scenario 5:

- Warp requests 32 **unaligned, consecutive** 4-byte words

Addresses fall within 5 sectors

- Warp needs 128 bytes
- 160 bytes move across the bus
- **Bus utilization: 80%**



Scenario 6:

- 2 Warps request 32 **unaligned, consecutive** 4-byte words

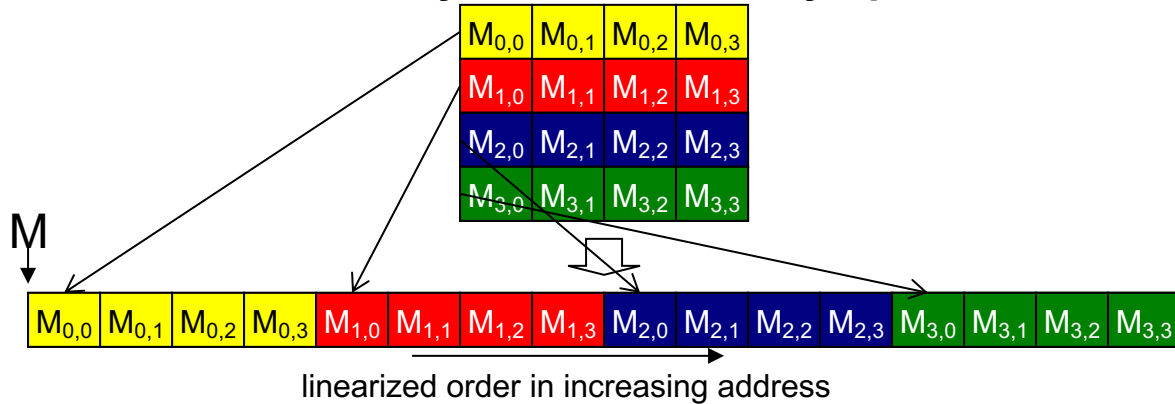
Addresses fall within 9 sectors

- 2 Warps need 256 bytes
- 288 or 320 bytes move across the bus (depends on presence of data in cache)
- **Bus utilization: 88% or 80%**

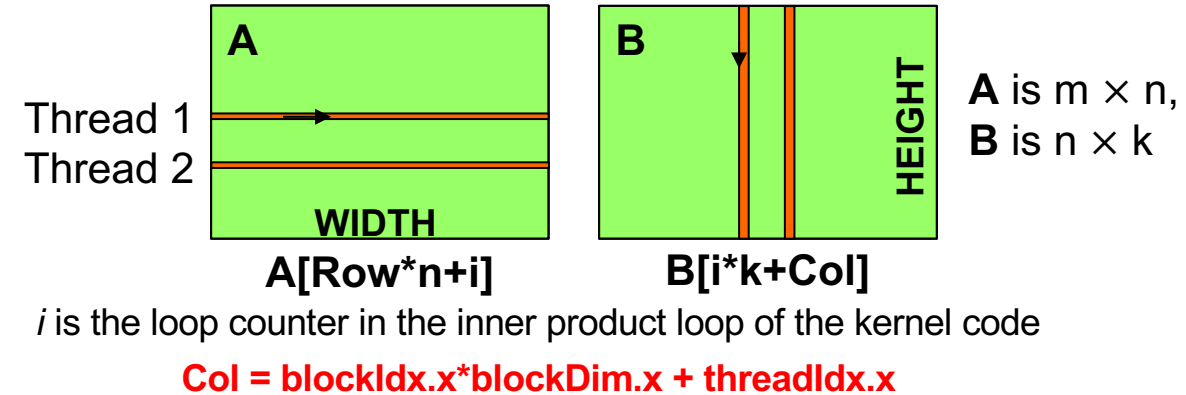
CUDA Memories

Global Memory Access for Mat. Mult.

2D C Array in Linear Memory Space



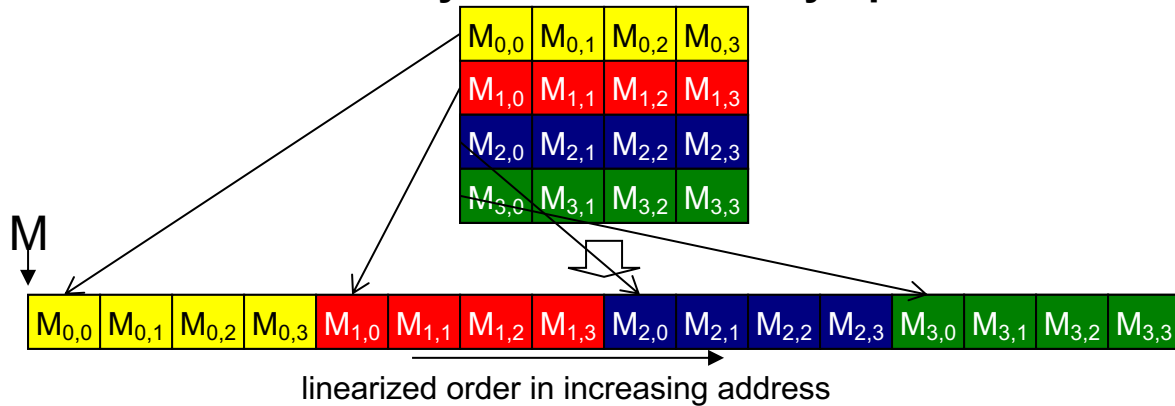
Two Access Patterns of Basic Matrix Multiplication



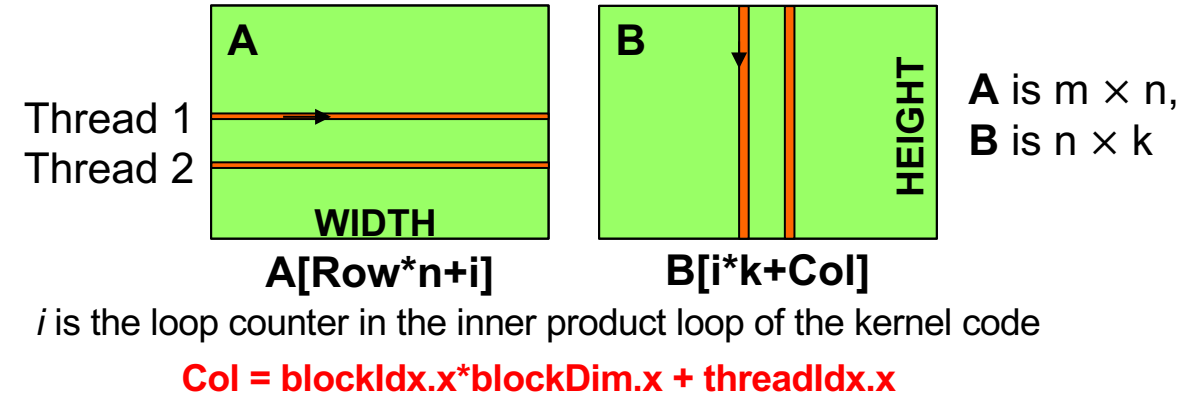
CUDA Memories

Global Memory Access for Mat. Mult.

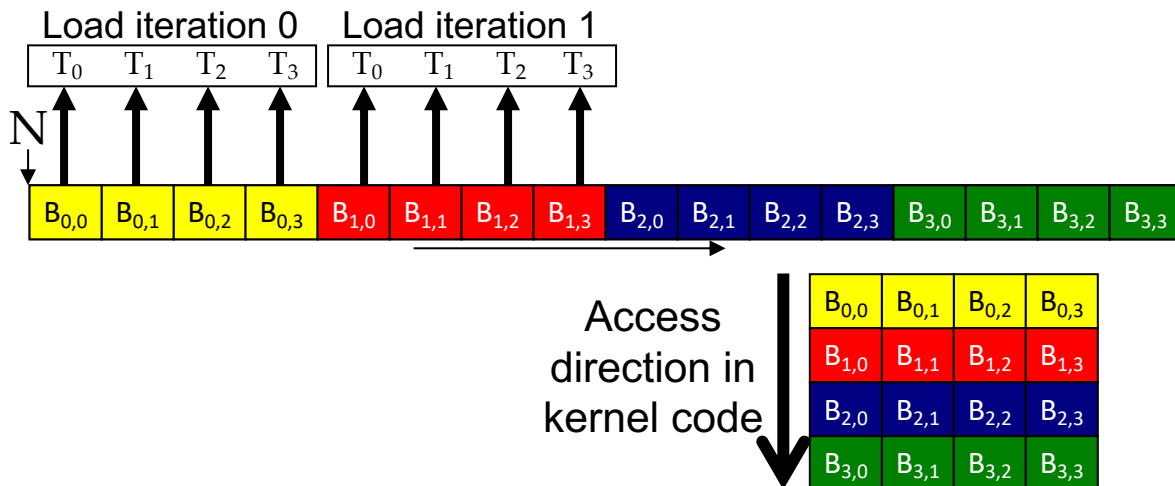
2D C Array in Linear Memory Space



Two Access Patterns of Basic Matrix Multiplication



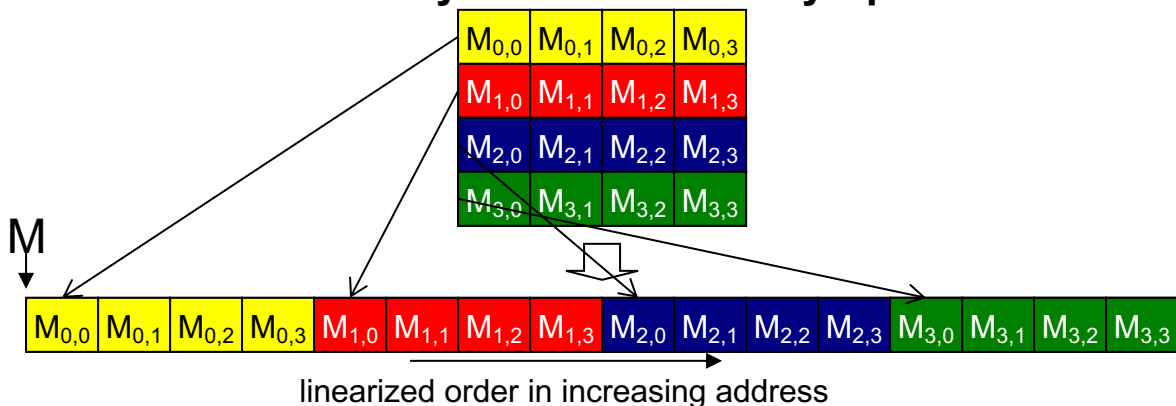
Matrix B accesses are coalesced



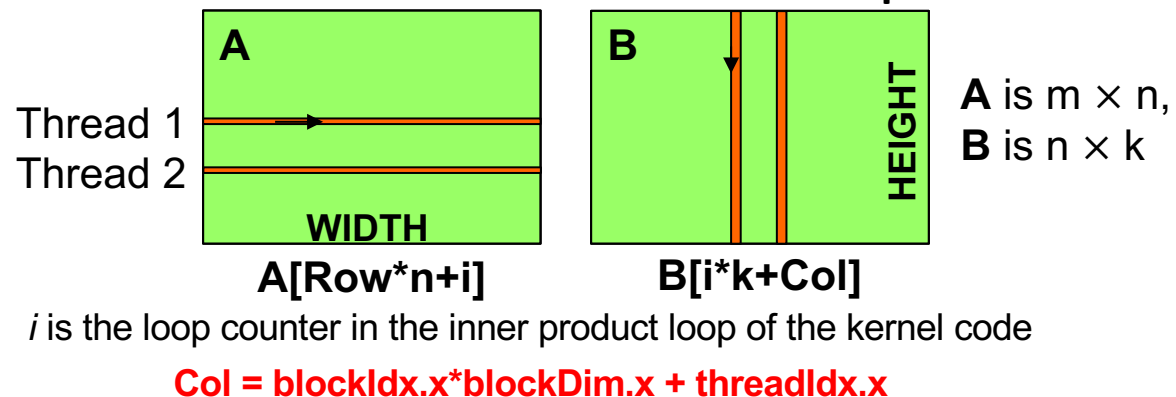
CUDA Memories

Global Memory Access for Mat. Mult.

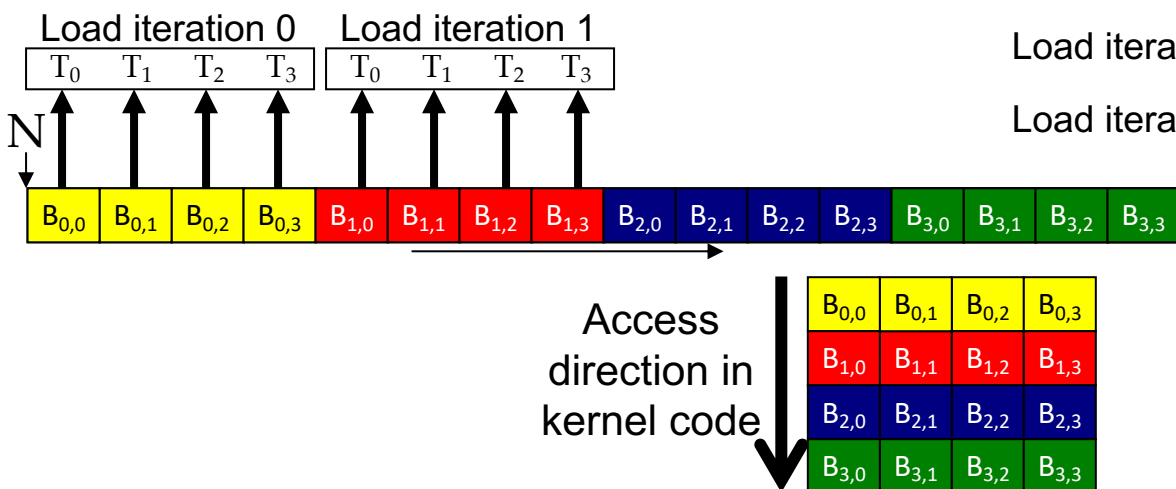
2D C Array in Linear Memory Space



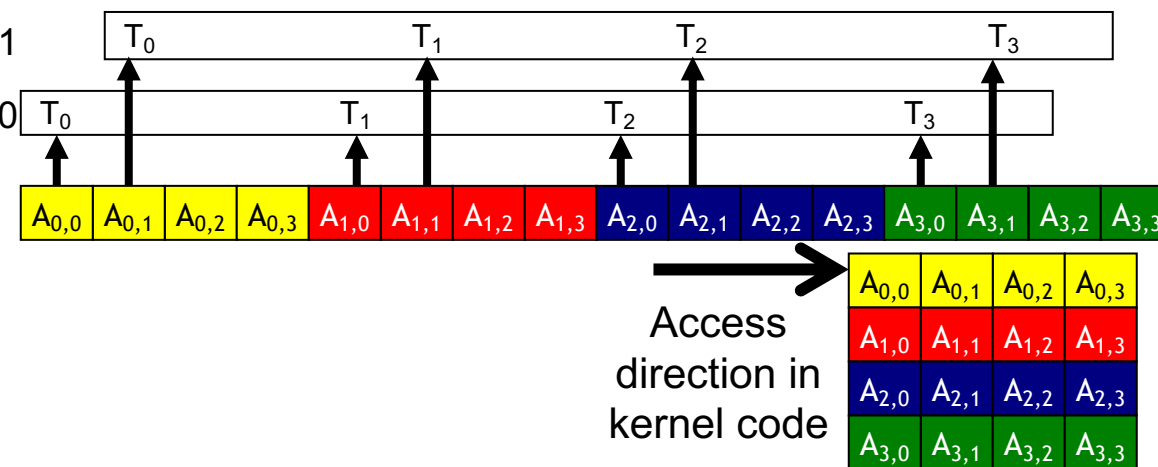
Two Access Patterns of Basic Matrix Multiplication



Matrix B accesses are coalesced



Matrix A Accesses are Not Coalesced



Hands-on Matrix sum

Univerza v Ljubljani



TECHNISCHE
UNIVERSITÄT
WIEN



VSB TECHNICAL
UNIVERSITY
OF OSTRAVA

IT4INNOVATIONS
NATIONAL SUPERCOMPUTING
CENTER



Co-funded by the
Erasmus+ Programme
of the European Union

This project has been funded with support from the European Commission.

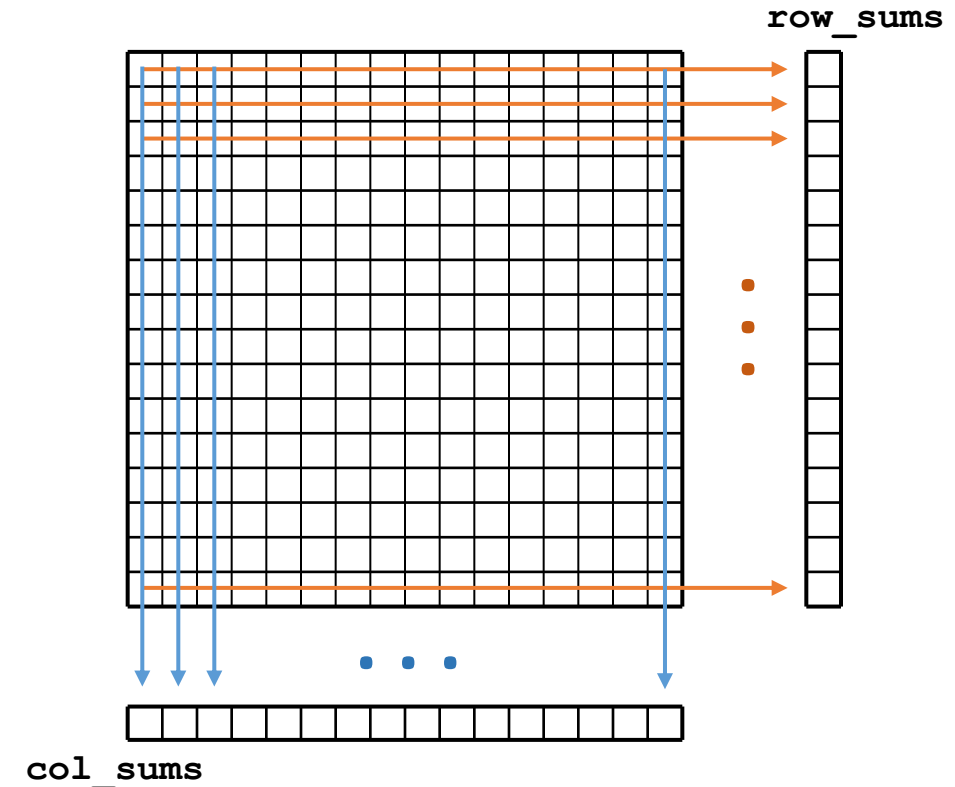
This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Hands-on Matrix sum

- `05_matrix_sum/<lang>/Task/matrix_sum.<ext>`
- Sum of values in a matrix
 - In each row (`matrix_sum_each_row` kernel)
 - In each column (`matrix_sum_each_col` kernel)
- Complete the TODO task
 - Implement the second kernel
- Think about the memory access pattern
 - Do not think about each thread individually, think about the threadblock (or rather warp) as a whole
- Beware C vs Fortran conventions for storing a matrix in memory
 - Row-major vs column-major order

Correct output (C++):

```
Summation time in each row:      19.320 ms
Summation time in each column:    7.801 ms
Using coalesced memory accesses was 2.48 times faster
```



Shared Memory

Univerza v Ljubljani



TECHNISCHE
UNIVERSITÄT
WIEN



VSB TECHNICAL
UNIVERSITY
OF OSTRAVA

IT4INNOVATIONS
NATIONAL SUPERCOMPUTING
CENTER



Co-funded by the
Erasmus+ Programme
of the European Union

This project has been funded with support from the European Commission.

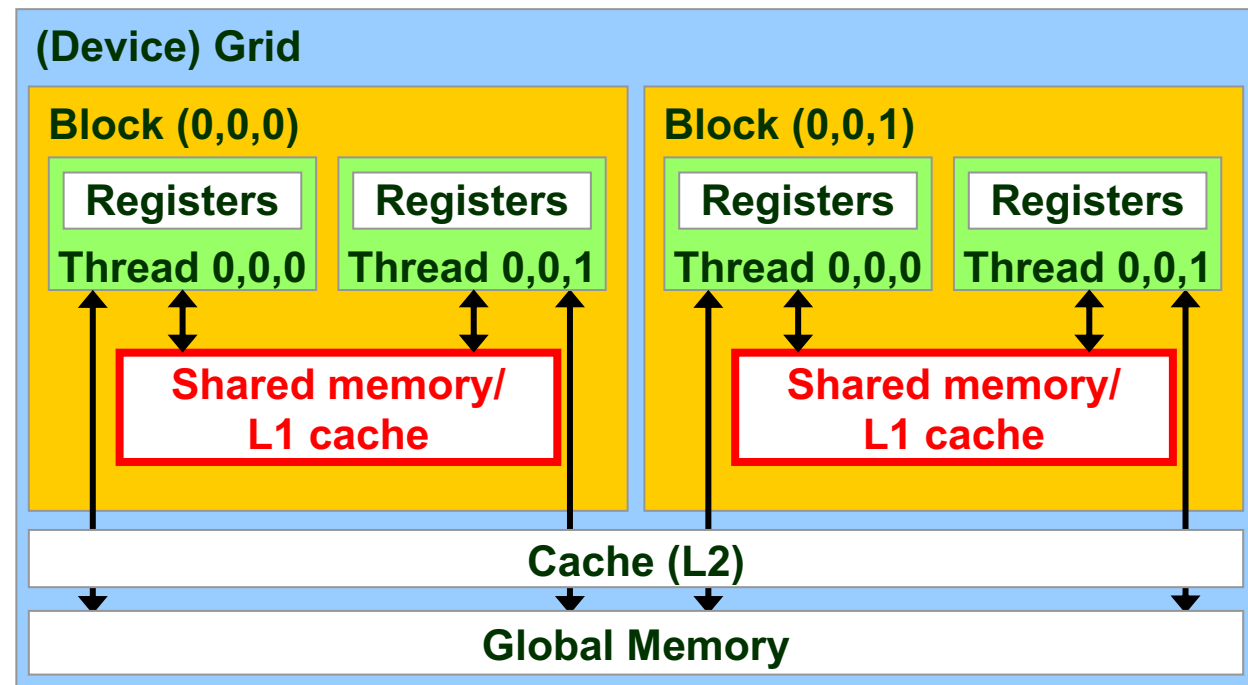
This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

CUDA Memories

Shared Memory in CUDA

Special type of memory whose contents are explicitly defined and used only in the kernel source code

- **one independent chunk in each SM**
- **accessed at much higher speed** (in both latency and throughput) than global memory
- **scope of access and sharing – all threads in a block**
- lifetime – thread block, contents will disappear after the corresponding thread finishes terminates execution
- accessed by memory load/store instructions
- a form of scratchpad memory in computer architecture



CUDA Memories

Shared Memory in CUDA

Performance benefits compared to DRAM:

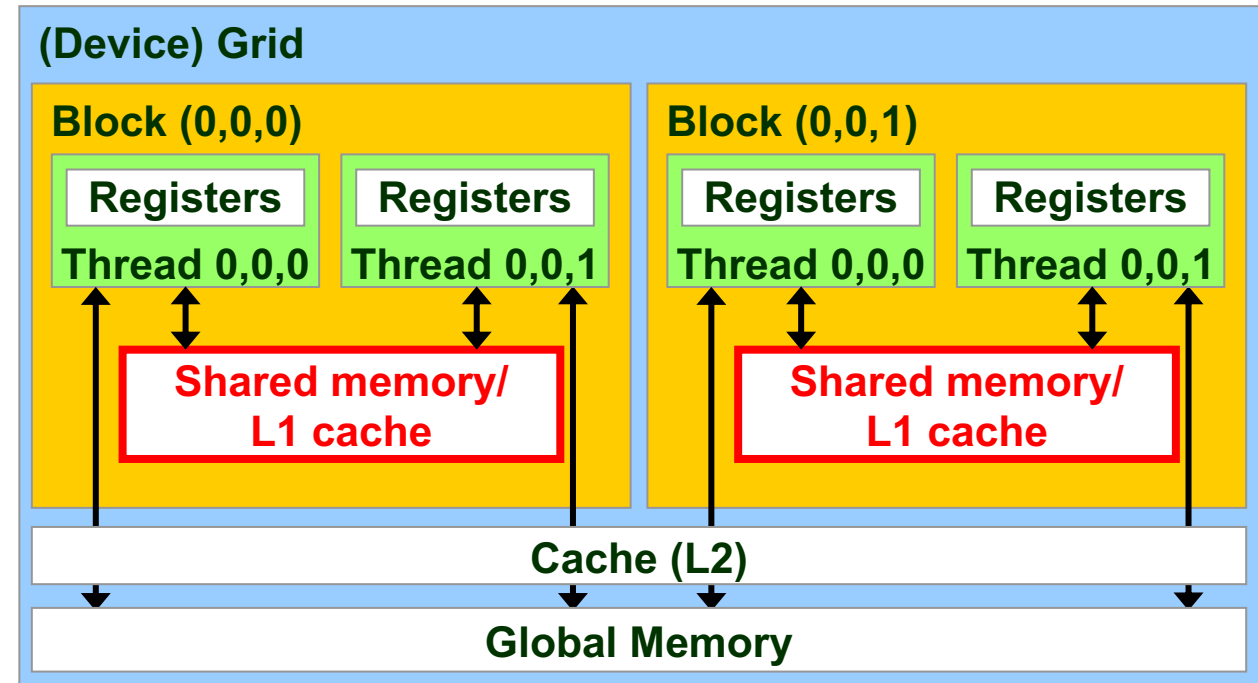
- 20-40x lower latency
- ~15x higher bandwidth
- accessed at 4-byte granularity
- Global Memory granularity is 32 Bytes

Ampere generation shared memory + L1 cache

- GA102 – 128 KB (used by A40 - mainly for graphics)
 - Configurable up to 100 KB
- GA100 – 192 KB (used by A100 - HPC)
 - Configurable up to 164 KB

Organization

- organized in 32 banks, each 4 Bytes wide
 - bandwidth: 4 Bytes per bank per clock per SM
 - 128 Bytes per clk per SM
- successive 4-byte words go to successive banks



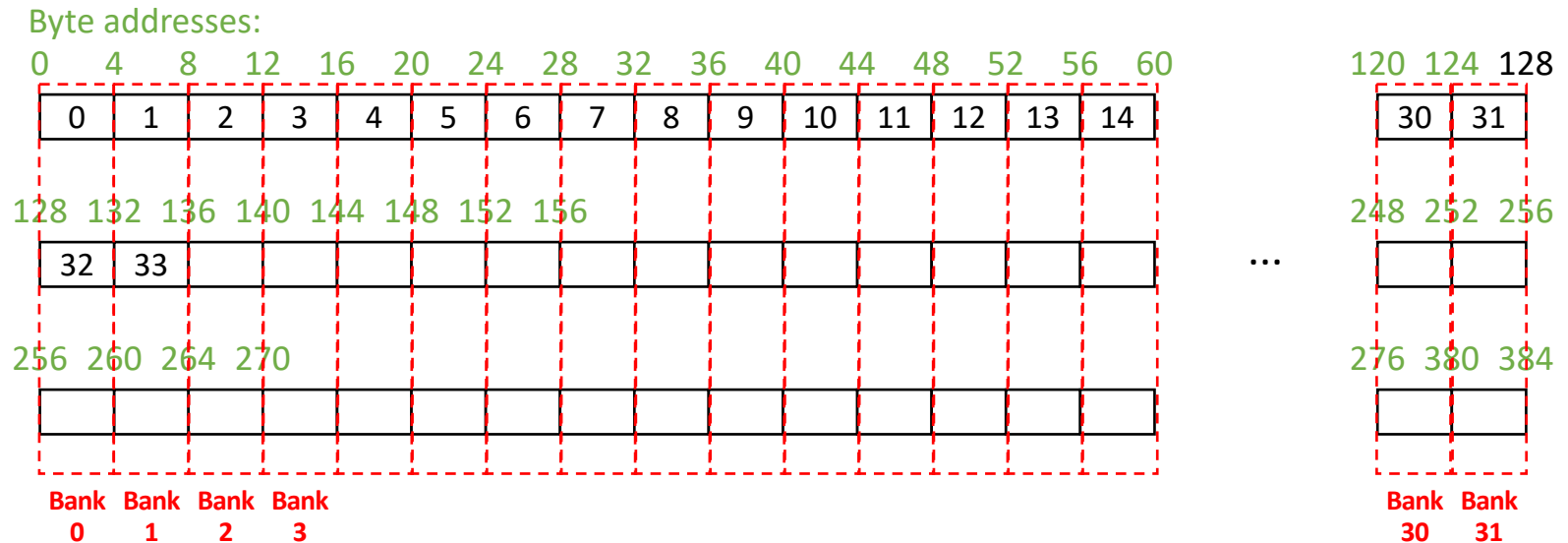
Bank index computation examples:

- $(4\text{B word index}) \% 32$
- $((1\text{B word index}) / 4) \% 32$
- 8B word spans two successive banks

CUDA Memories

Shared Memory in CUDA

Logical View of Shared Memory banks



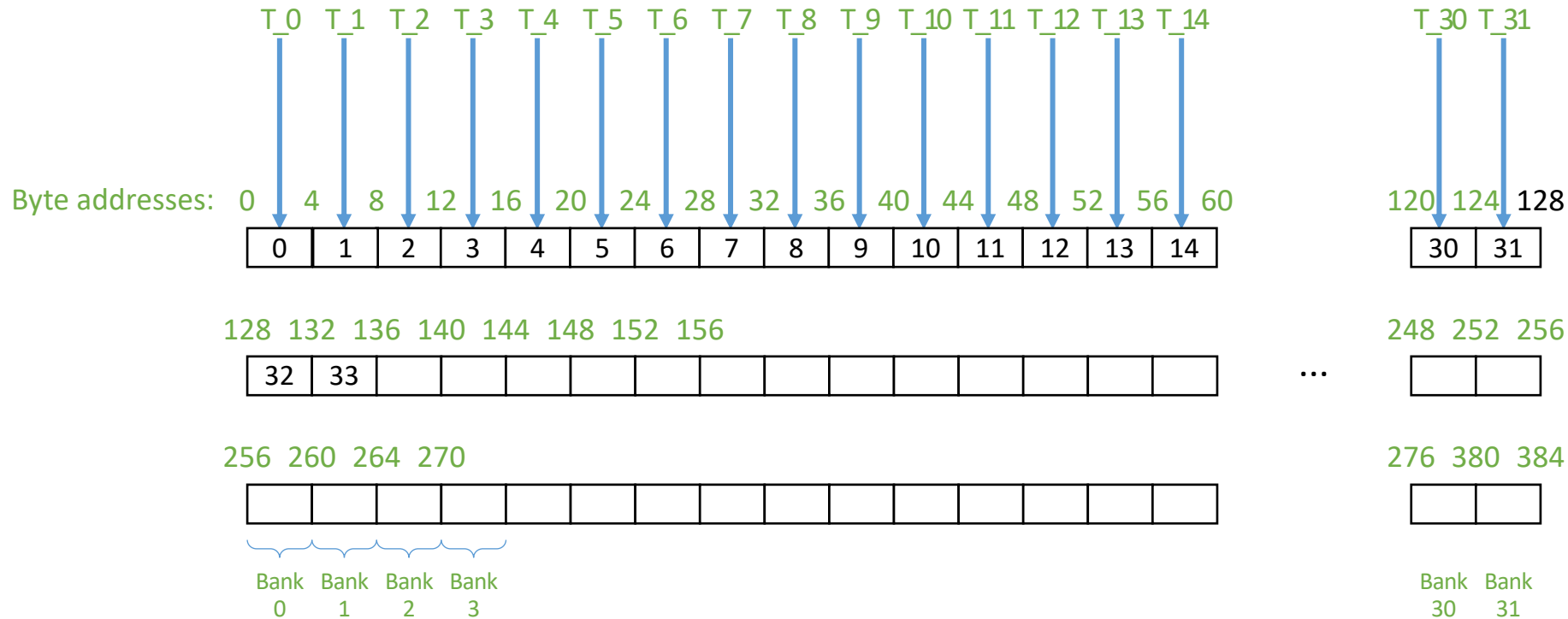
Banks Conflicts

- A **bank conflict** occurs when, **inside a warp**:
 - **2 or more** threads access within **different 4B words in the same bank**
 - Think: 2 or more threads access different “rows” in the same bank
- **N-way** bank conflict: **N threads** in a warp conflict
 - Increases latency
 - Worst case: 32-way conflict → 31 replays
 - Each replay adds a few cycles of latency
- There is **no bank conflict** if:
 - Several threads access the same 4-byte word
 - Several threads access different bytes of the same 4-byte word

CUDA Memories

Shared Memory in CUDA

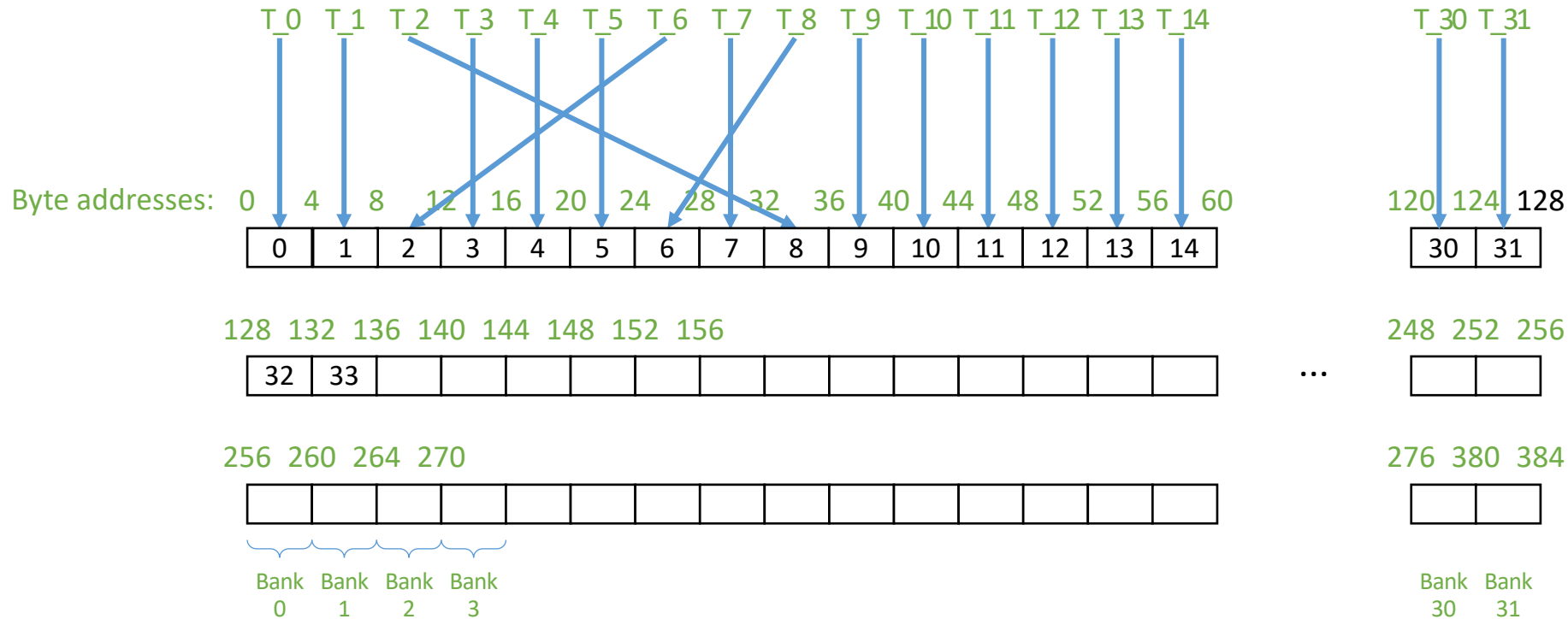
No Bank Conflict



CUDA Memories

Shared Memory in CUDA

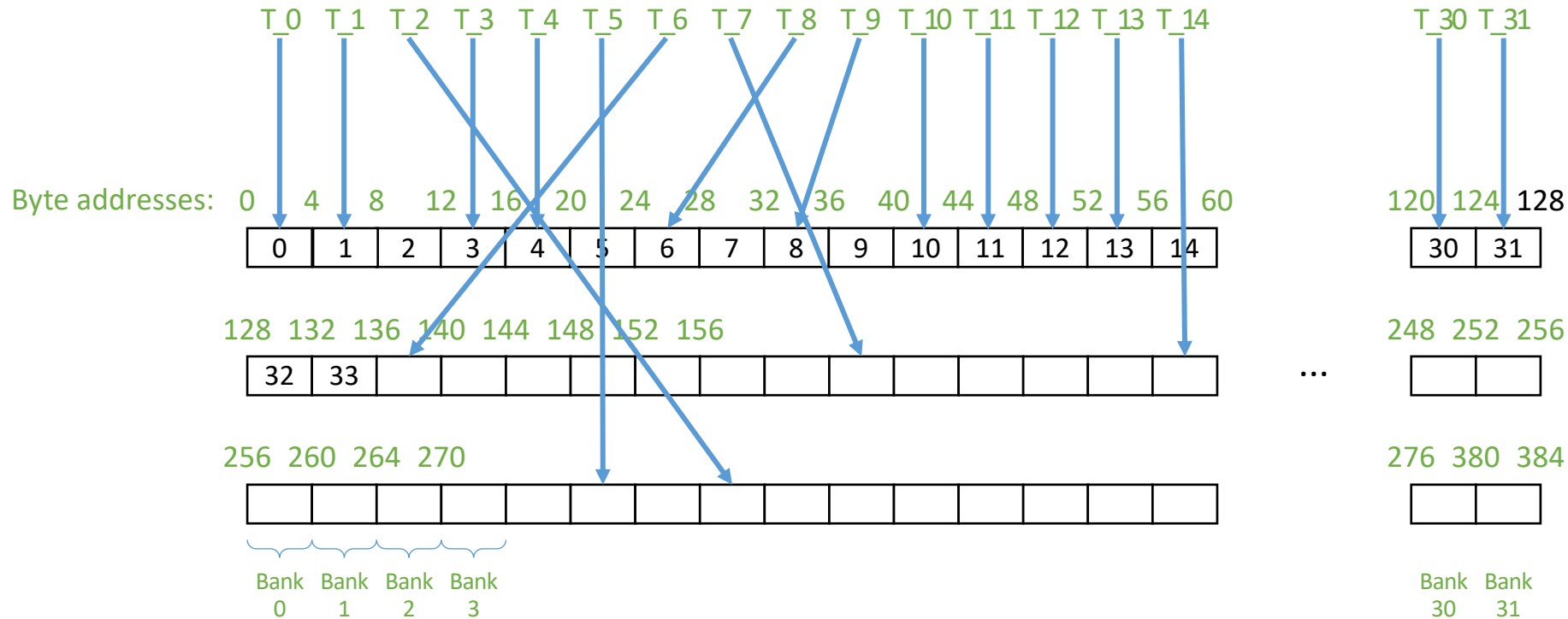
No Bank Conflict



CUDA Memories

Shared Memory in CUDA

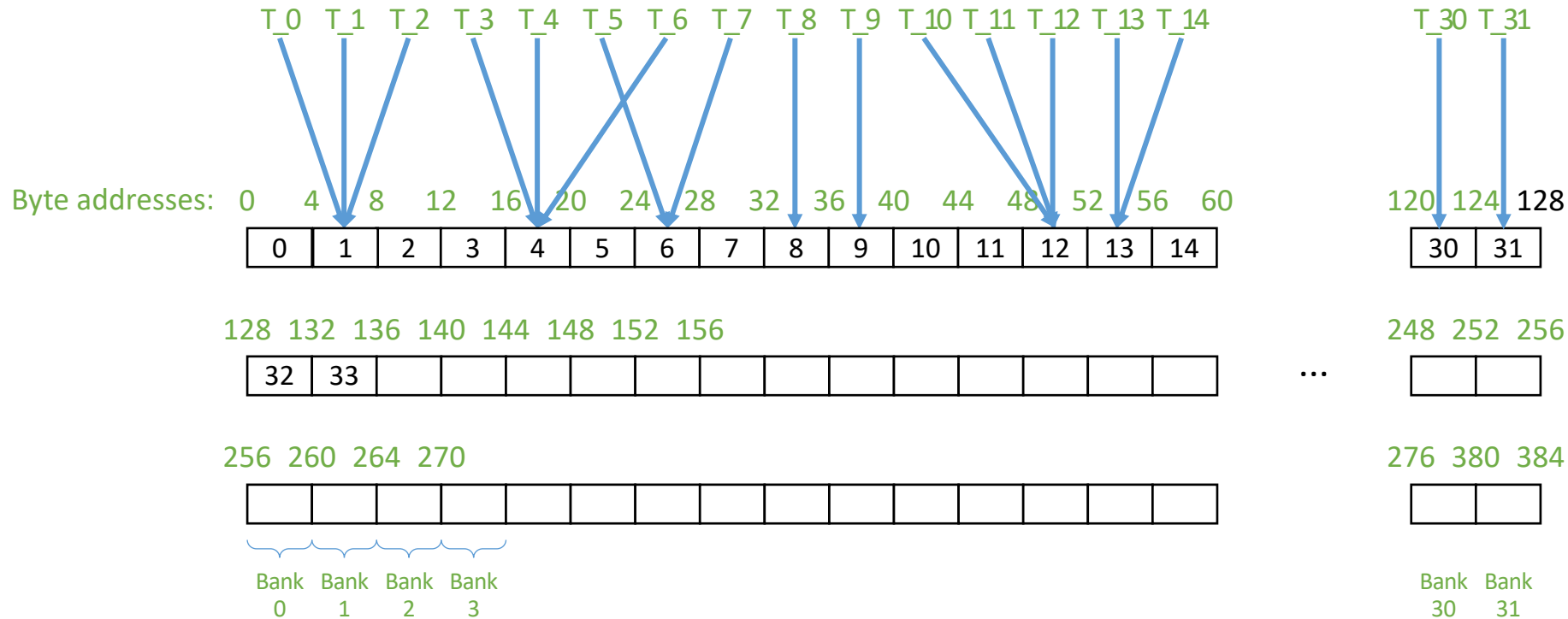
No Bank Conflict



CUDA Memories

Shared Memory in CUDA

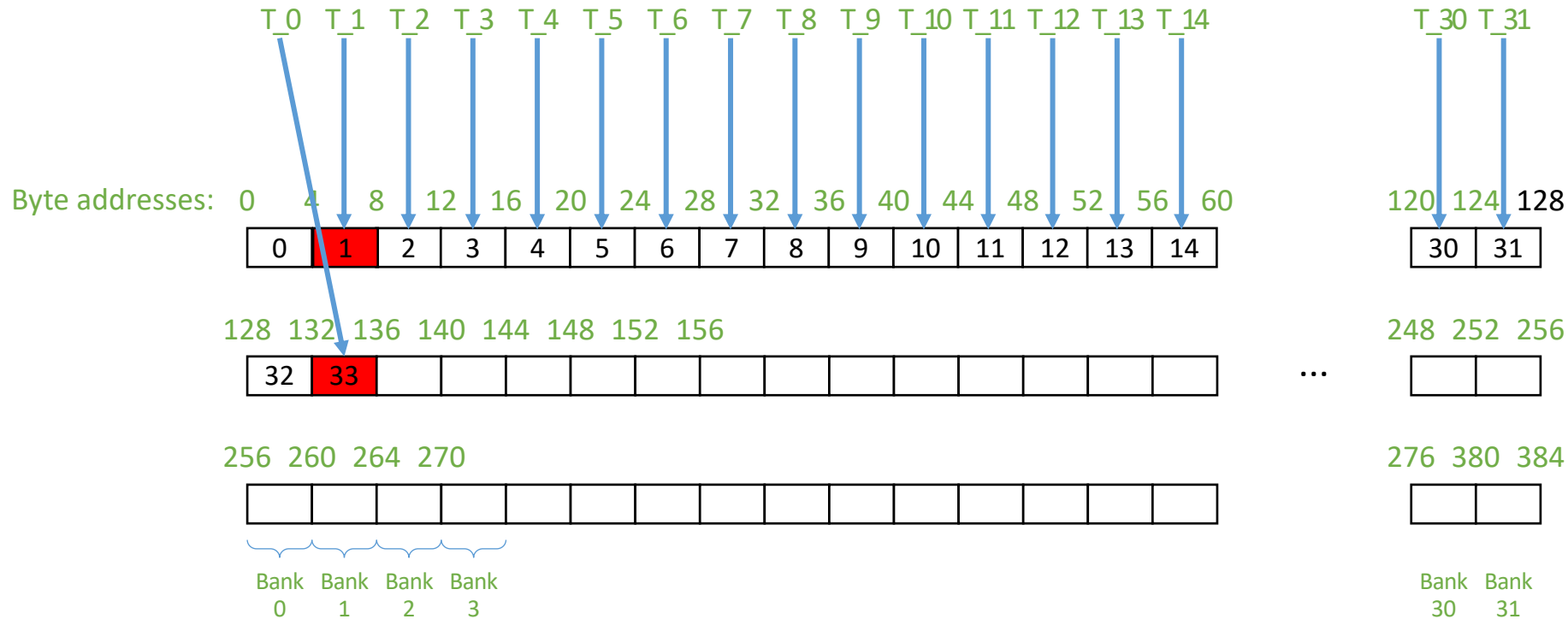
No Bank Conflict



CUDA Memories

Shared Memory in CUDA

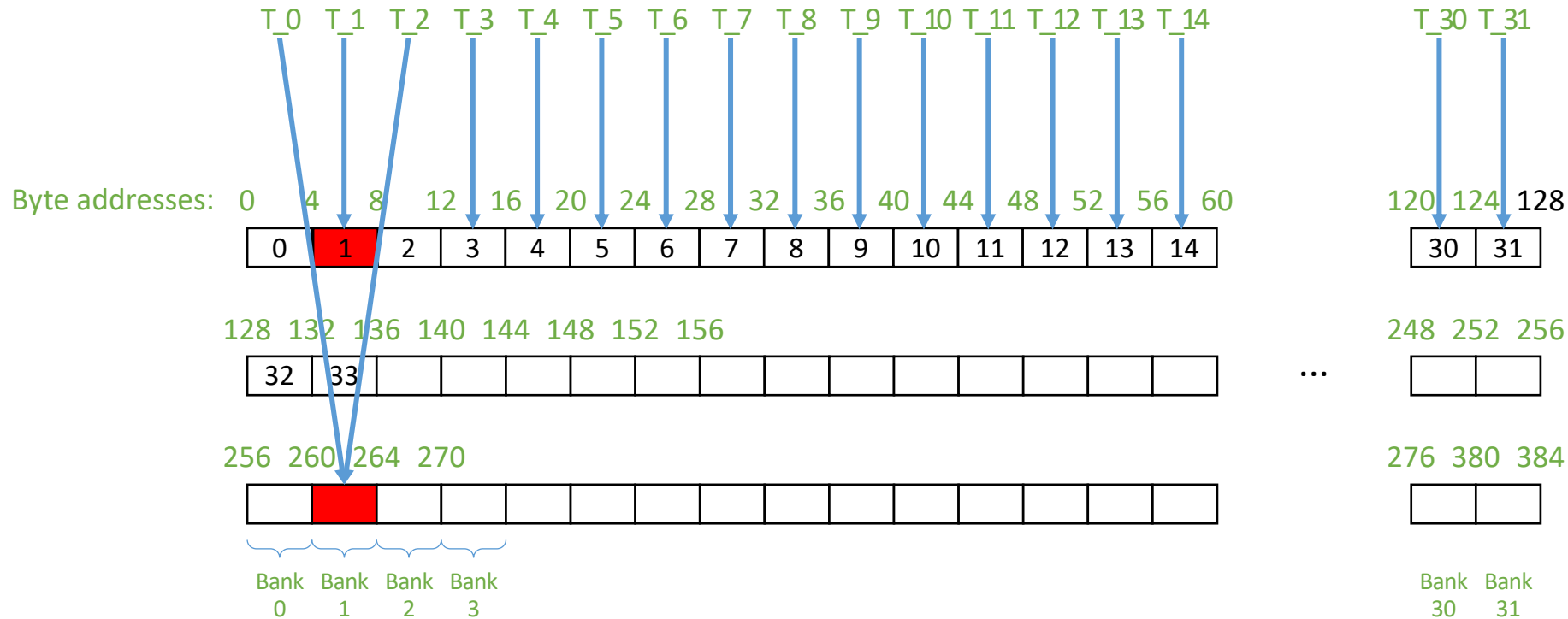
2-way Bank Conflict



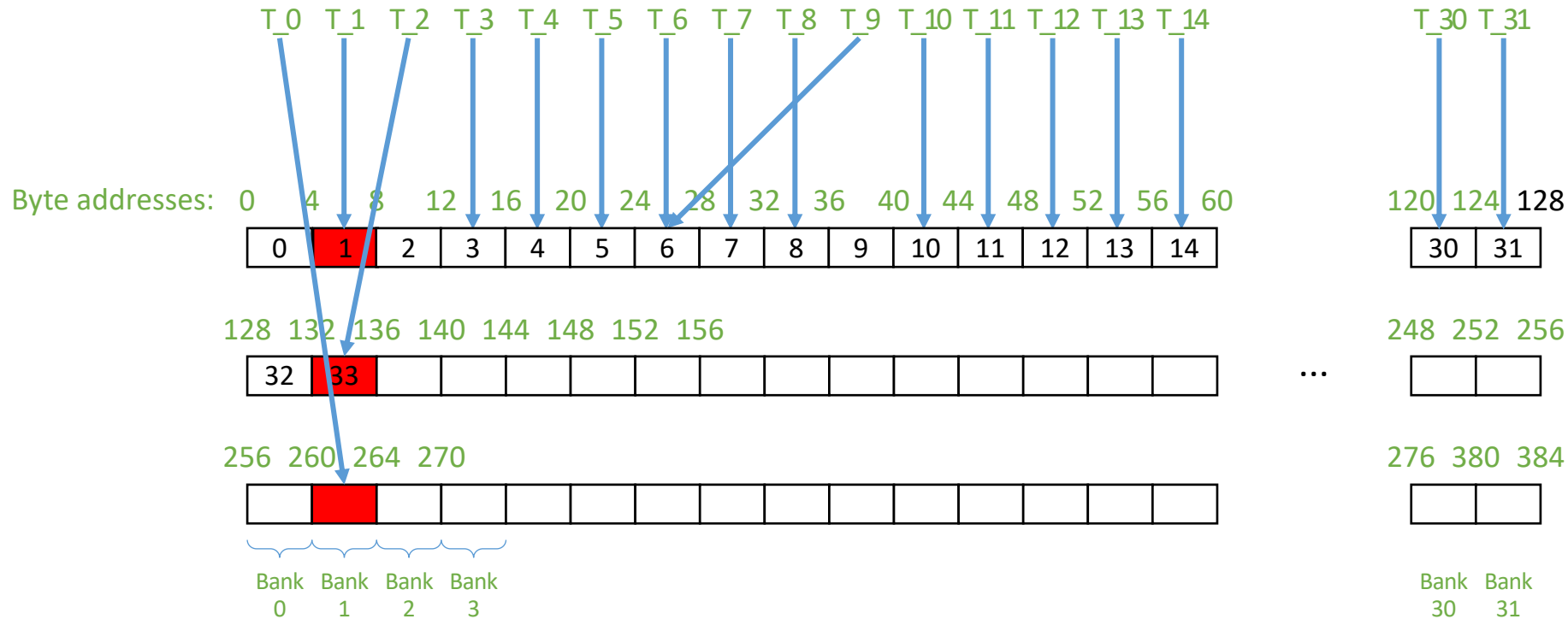
CUDA Memories

Shared Memory in CUDA

2-way Bank Conflict

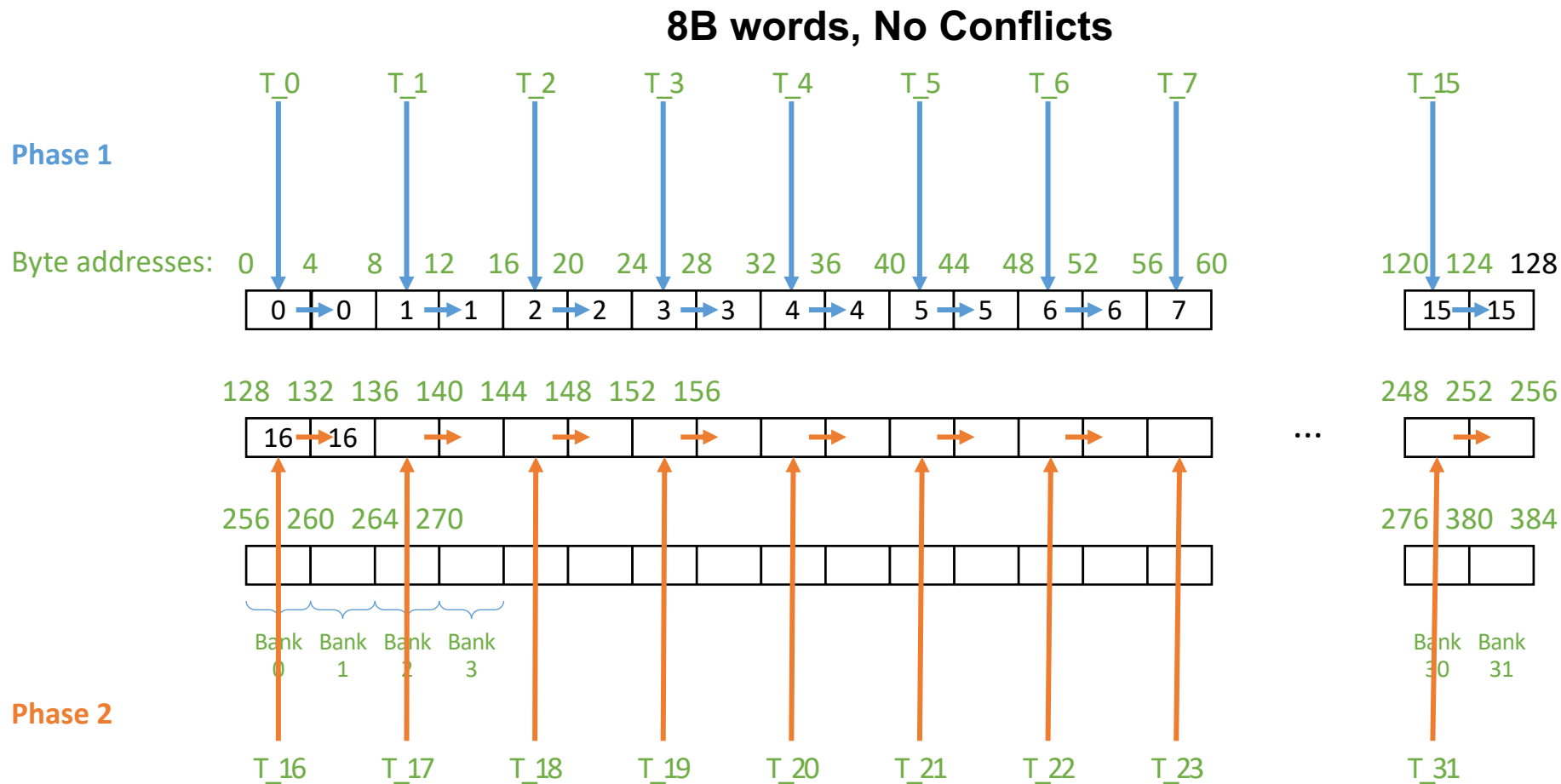


3-way Bank Conflict



CUDA Memories

Shared Memory in CUDA

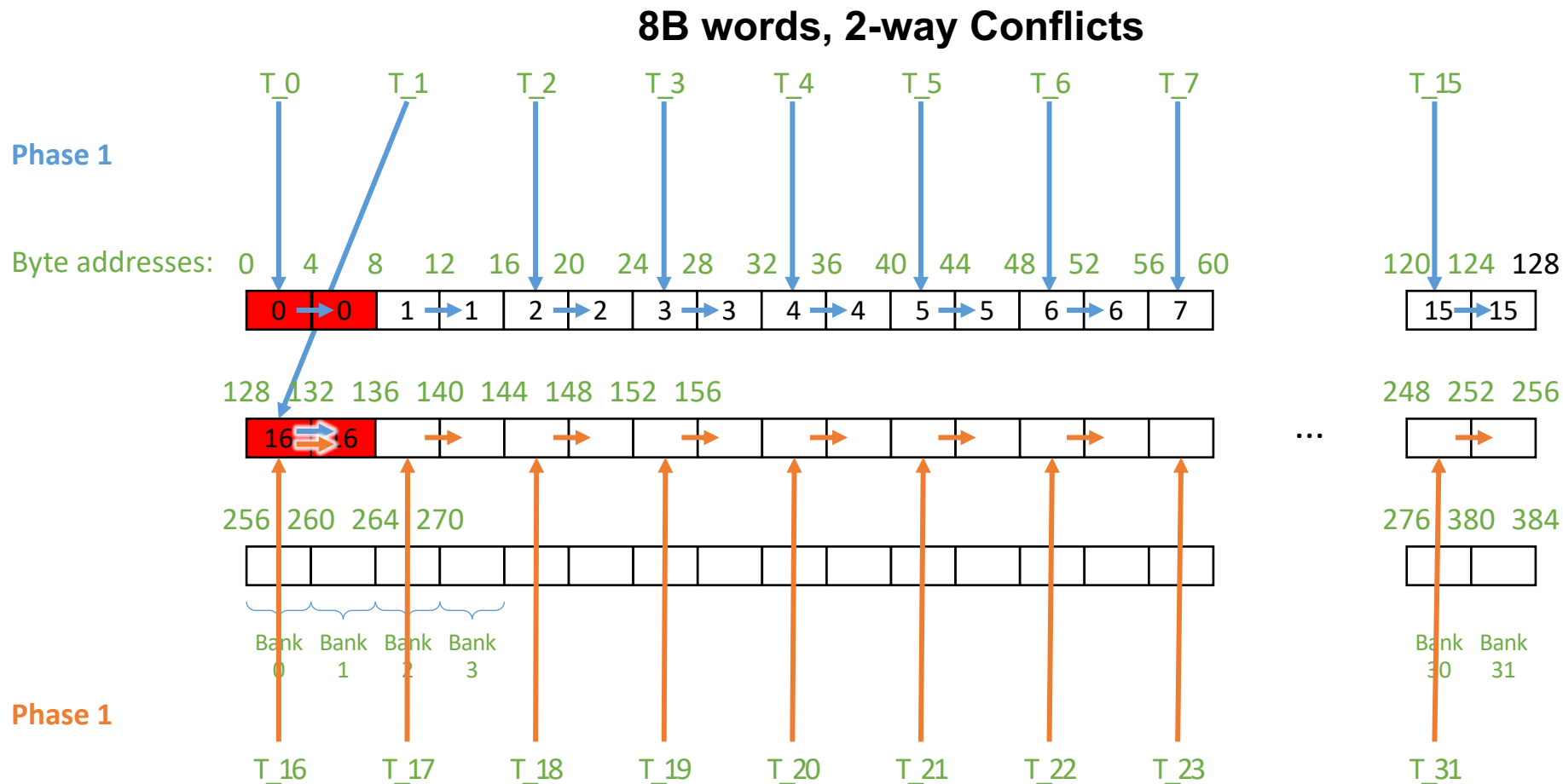


8B words are accessed in 2 phases:

- **Phase 1:** Process addresses of the **first 16 threads** in a warp
- **Phase 2:** Process addresses of the **second 16 threads** in a warp

CUDA Memories

Shared Memory in CUDA



8B words are accessed in 2 phases:

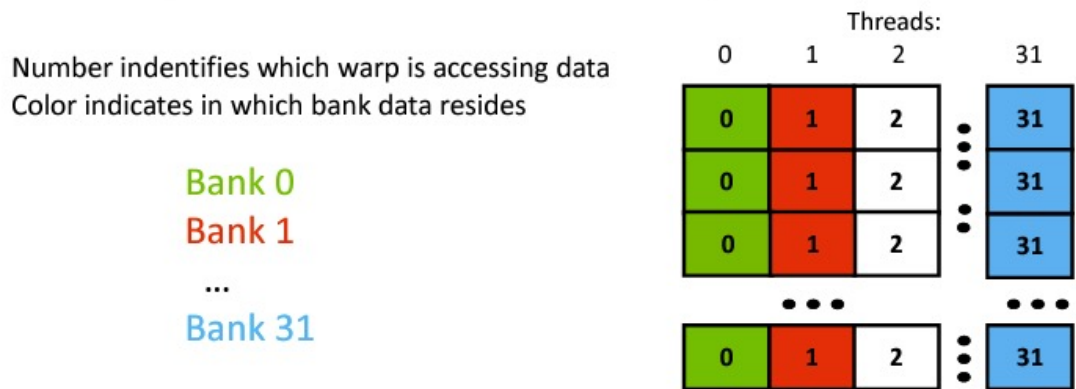
- **Phase 1:** Process addresses of the **first 16 threads** in a warp
- **Phase 2:** Process addresses of the **second 16 threads** in a warp

Case Study: Matrix Transpose

32x32 SMEM array (.e.g. `__shared__ float sm[32][32]`)

Warp accesses a row : No conflict

Warp accesses a column : 32-way conflict



Case Study: Matrix Transpose

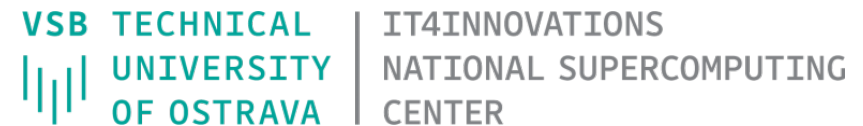
Solution: add a column for padding: 32x33
(.e.g. `__shared__ float sm[32][33]`)

Warp accesses a row or a column: no conflict



Memory and Data Locality: Tiling Technique

Univerza v Ljubljani



Co-funded by the
Erasmus+ Programme
of the European Union

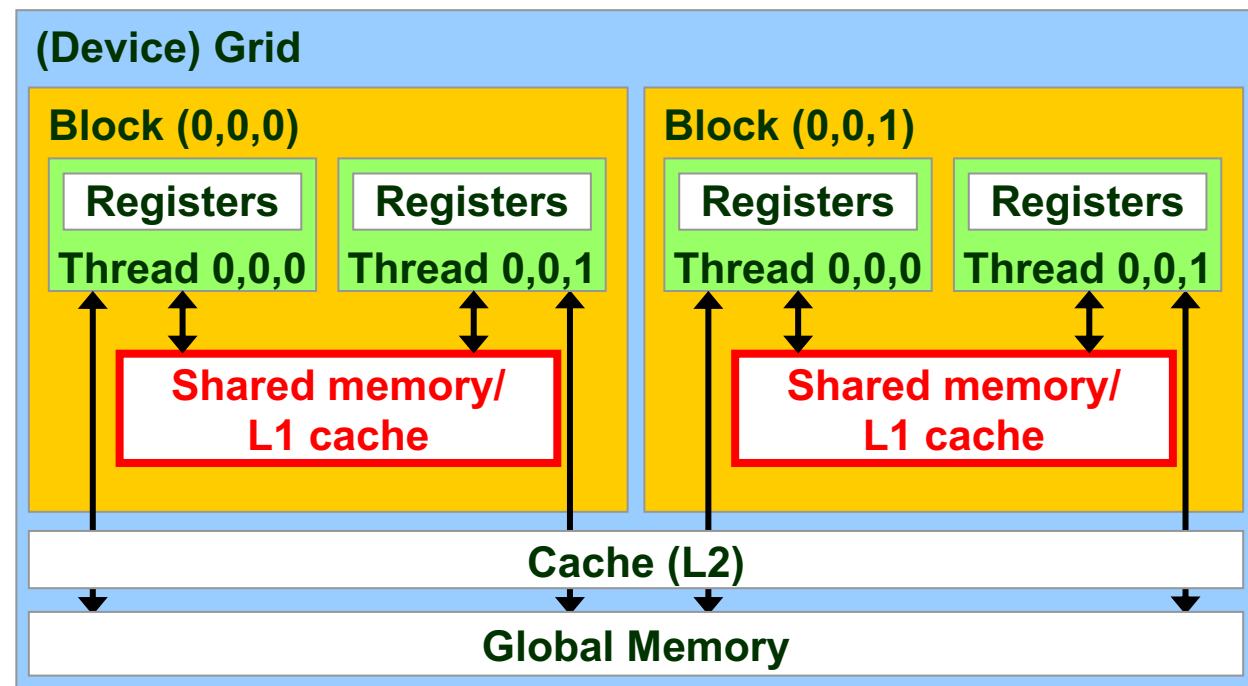
This project has been funded with support from the European Commission.
This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

CUDA Memories

Shared Memory in CUDA

Declaration:

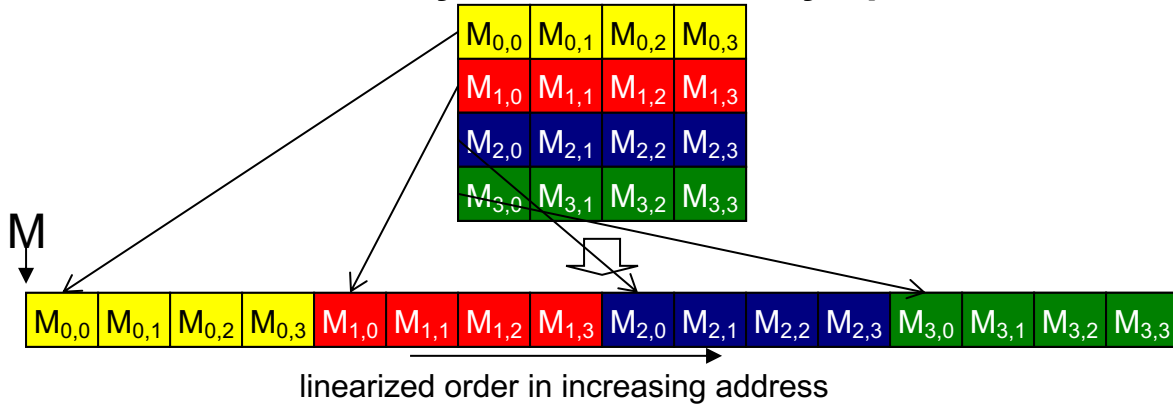
```
void CUDA_Kernel(unsigned char * in,  
unsigned char * out, int w, int h)  
{  
    __shared__ float  
    ds_in[TILE_WIDTH][TILE_WIDTH];  
    ...  
}
```



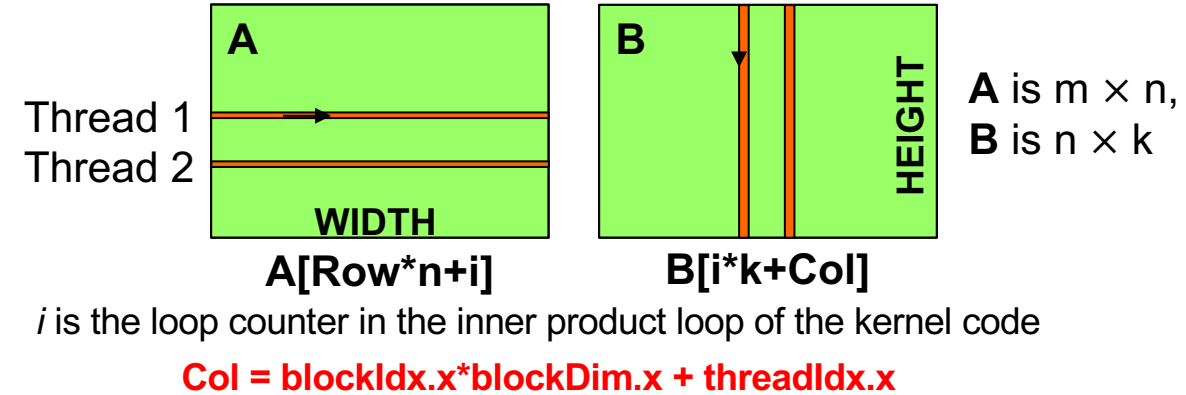
CUDA Memories

Matrix Multiplication – Memory access problem

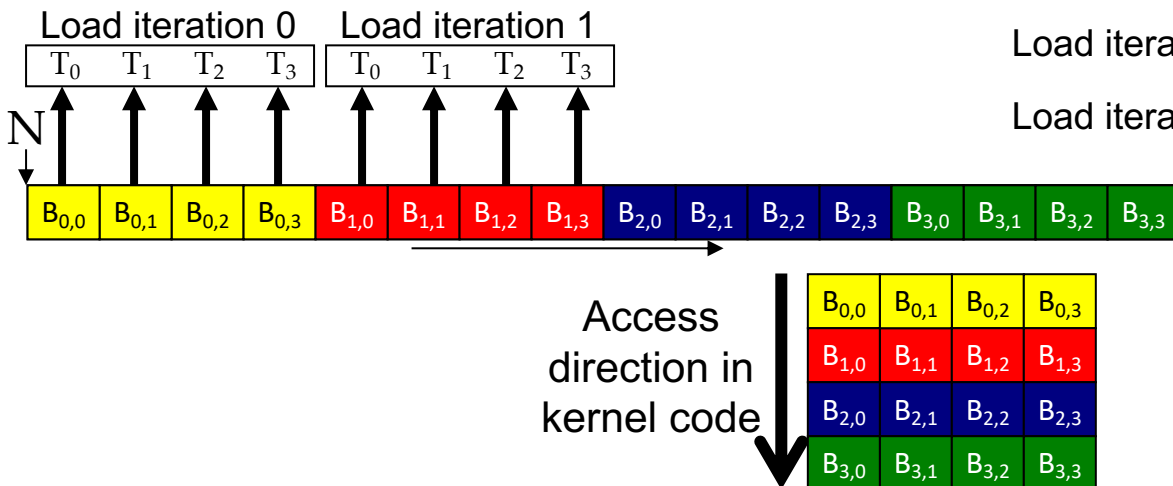
2D C Array in Linear Memory Space



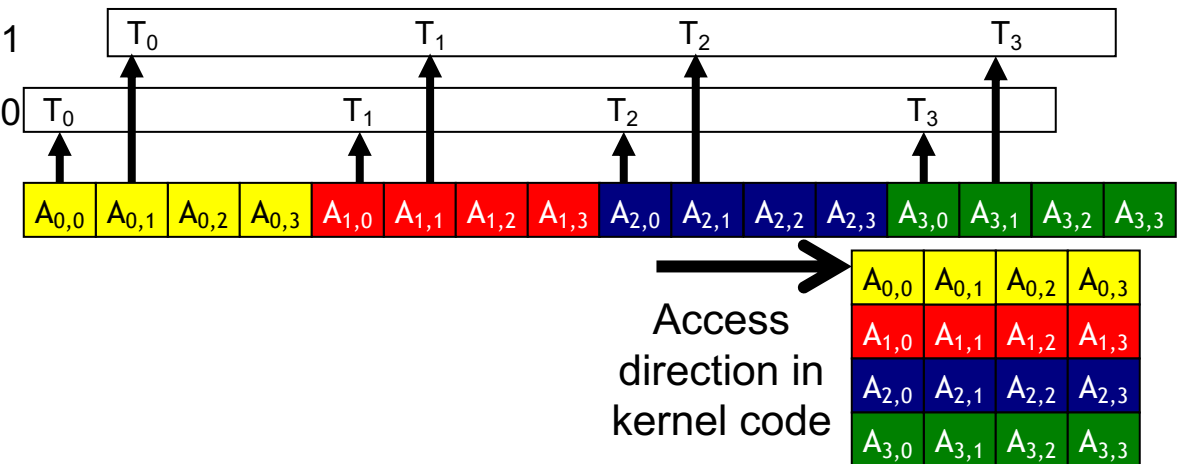
Two Access Patterns of Basic Matrix Multiplication



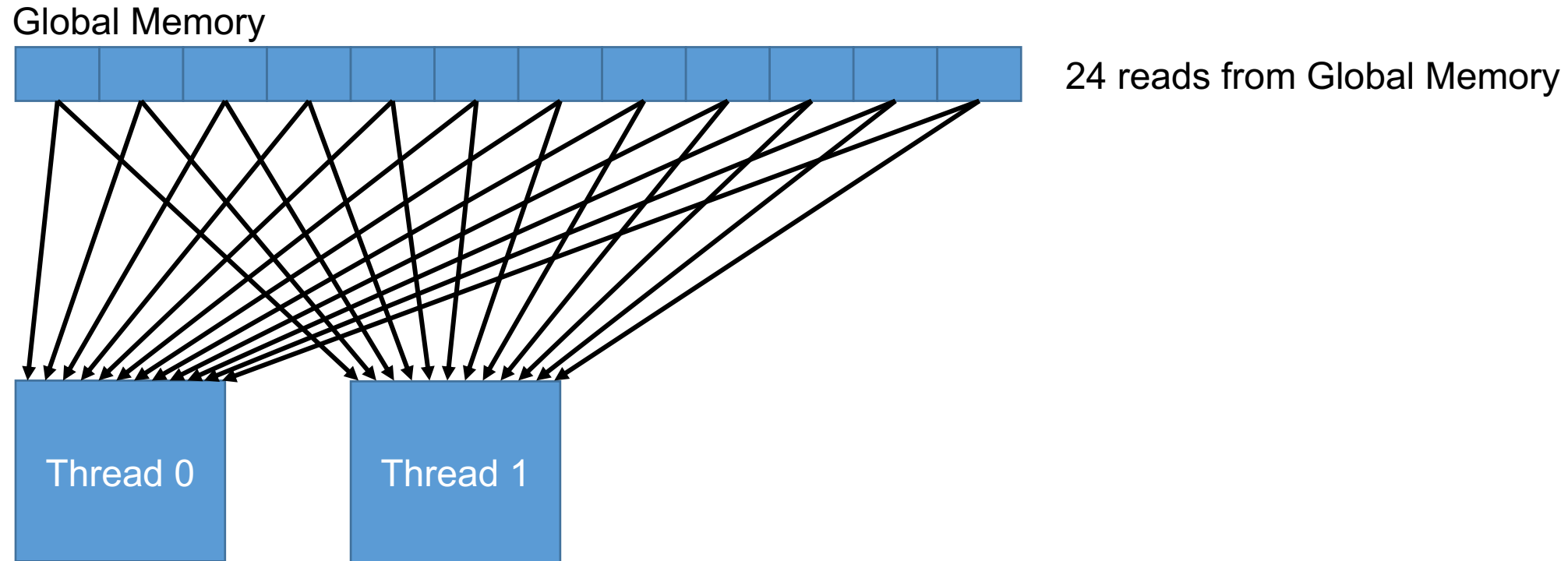
Matrix B accesses are coalesced



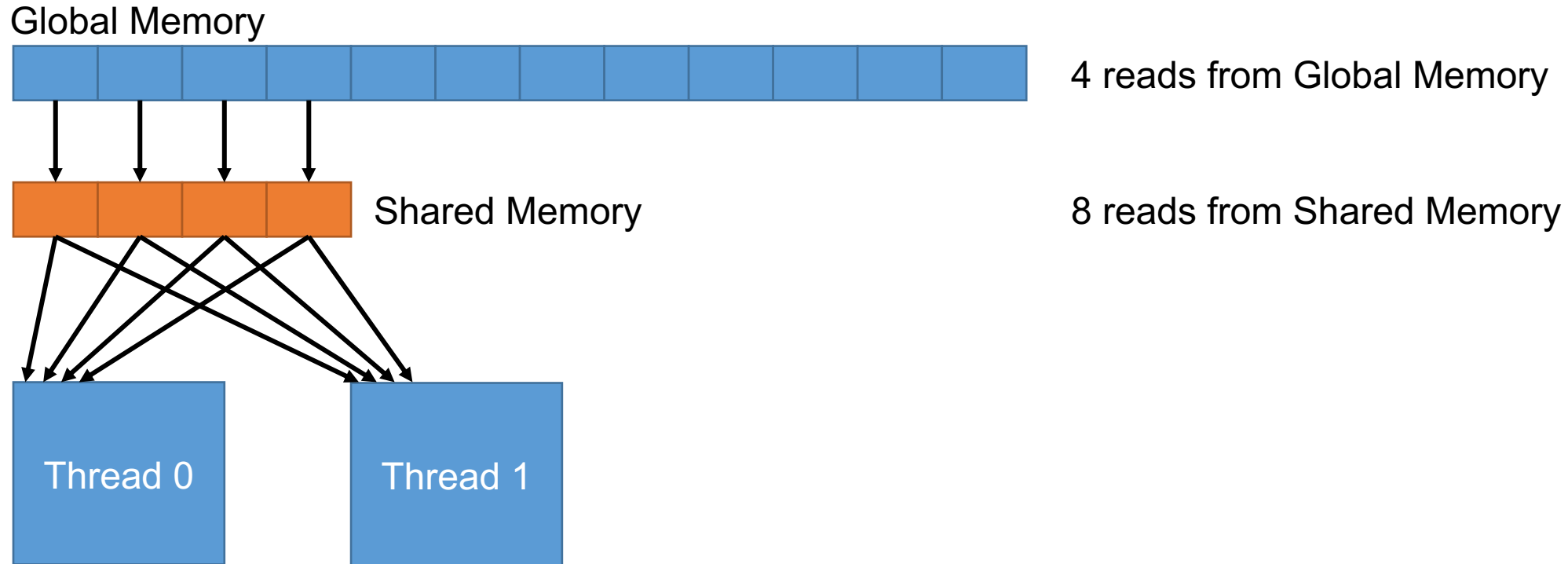
Matrix A Accesses are Not Coalesced



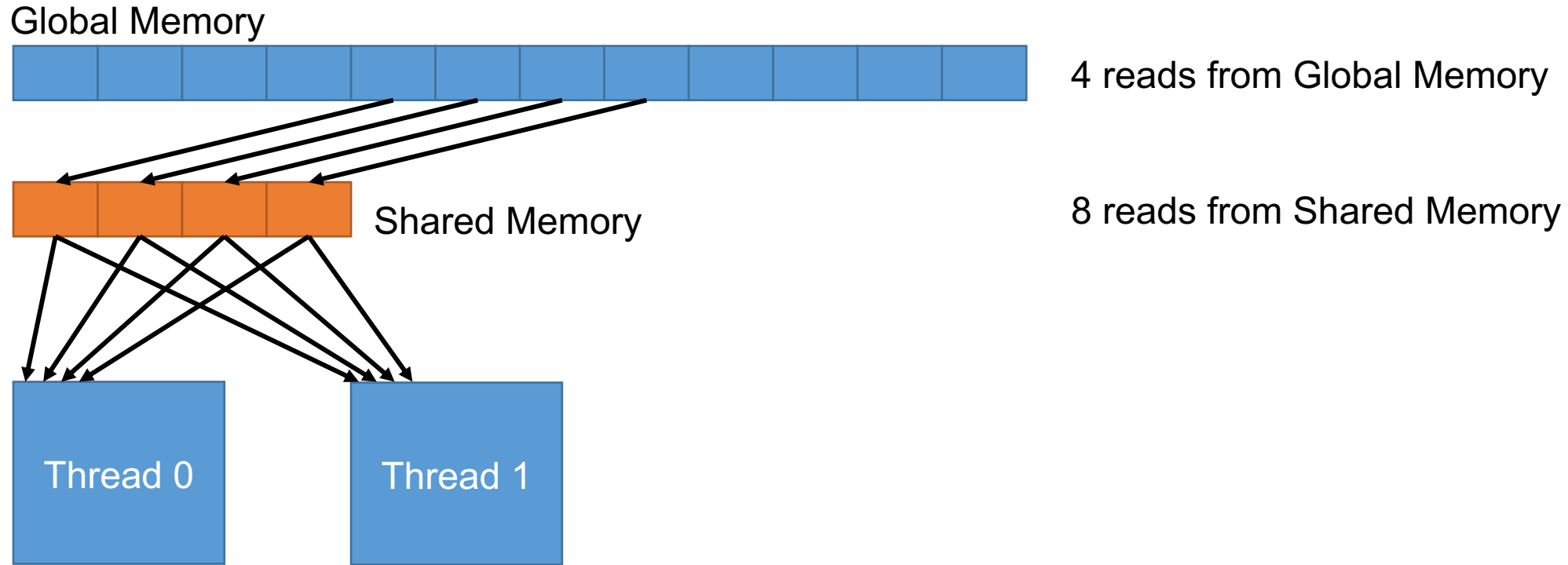
CUDA Memories Tiling Technique



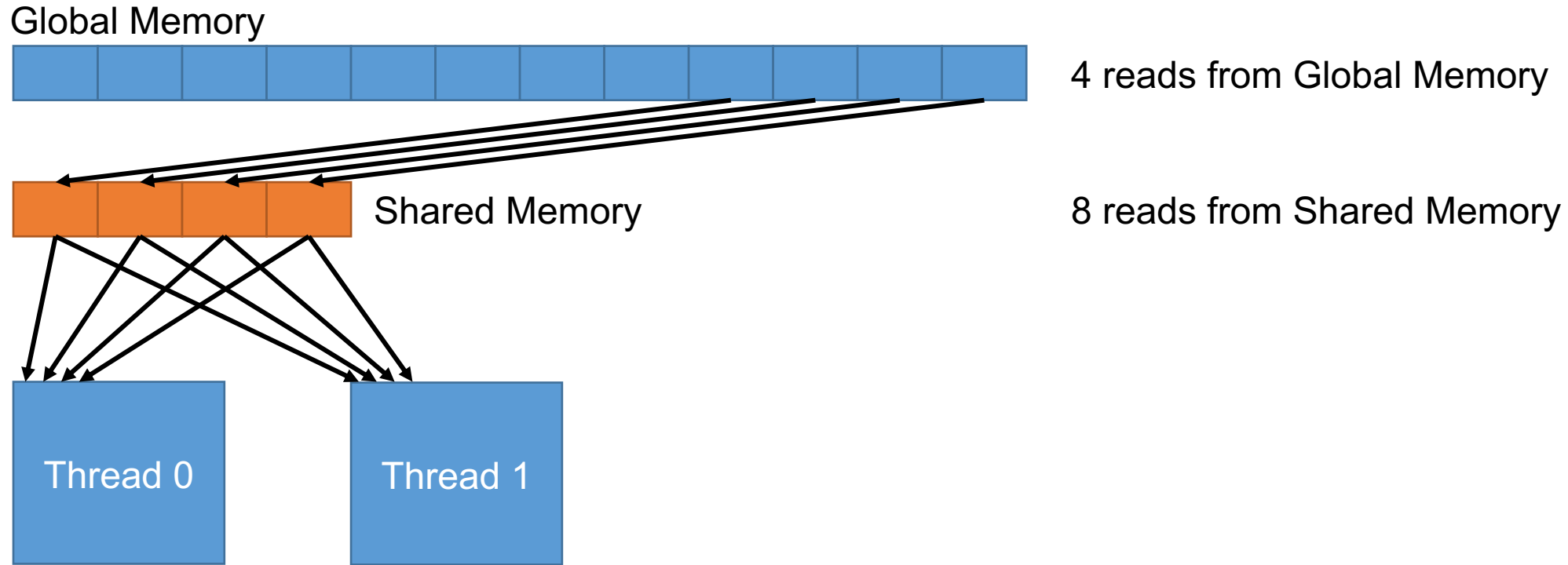
CUDA Memories Tiling Technique



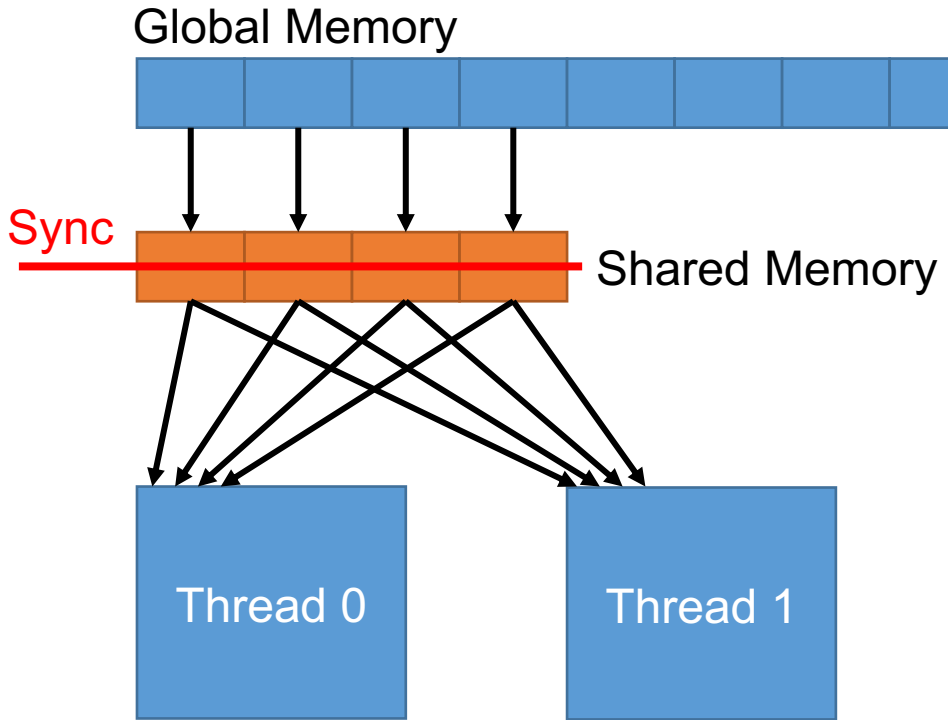
CUDA Memories Tiling Technique



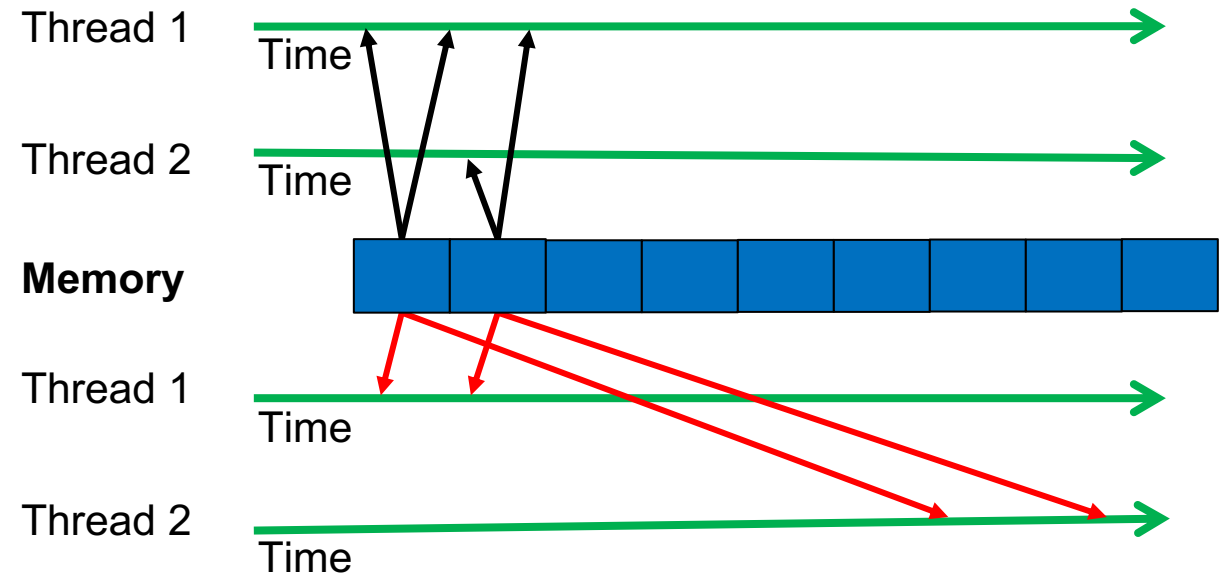
CUDA Memories Tiling Technique



Tiling needs synchronization



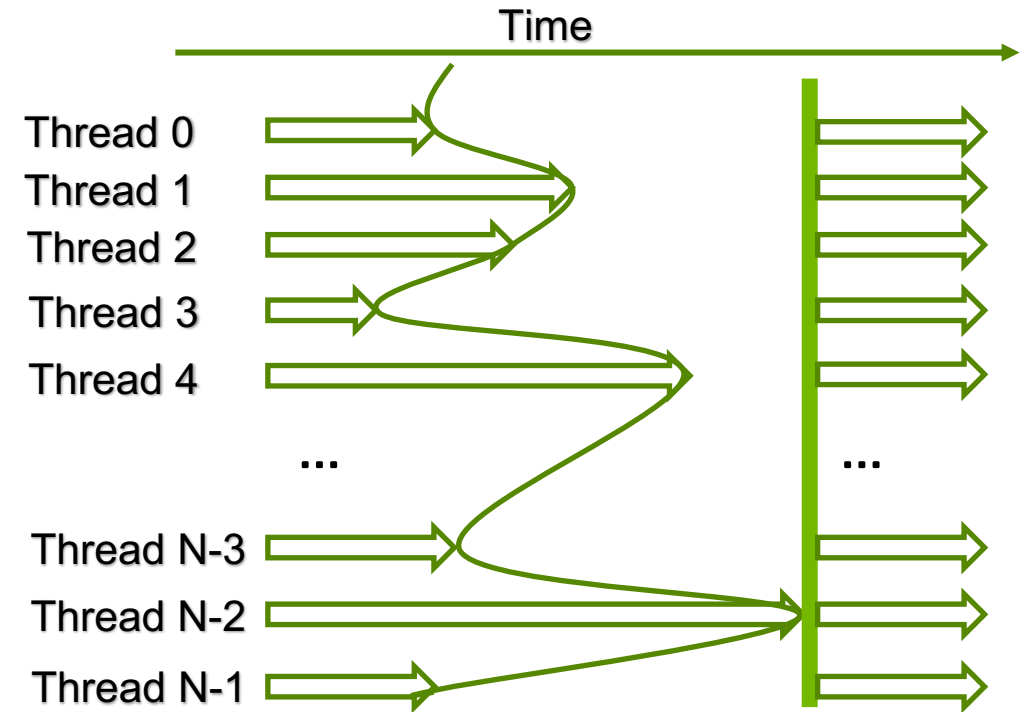
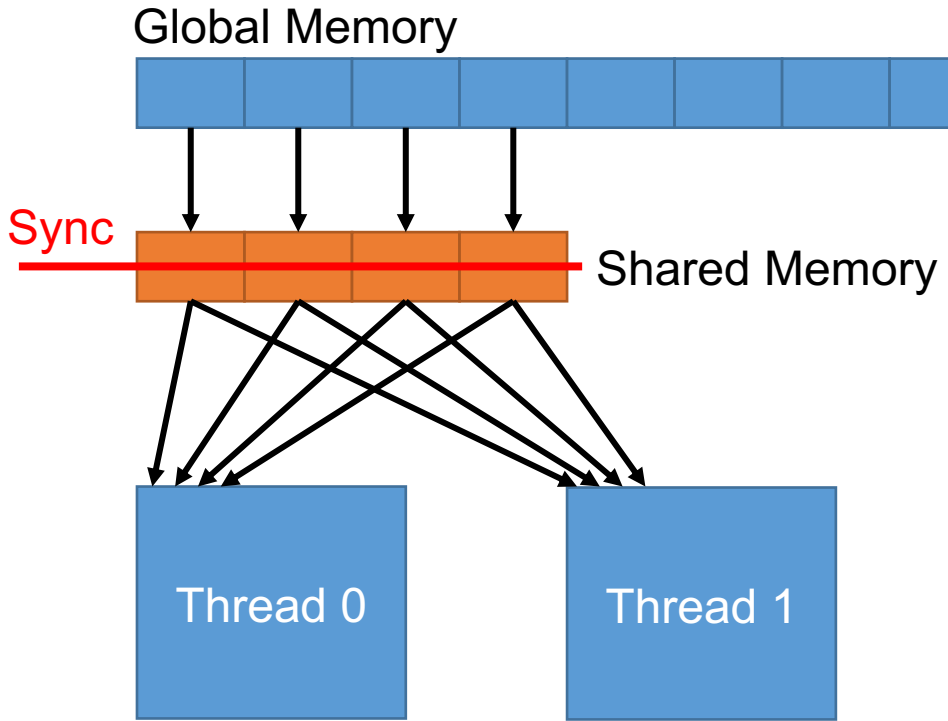
Good: when threads have similar access timing



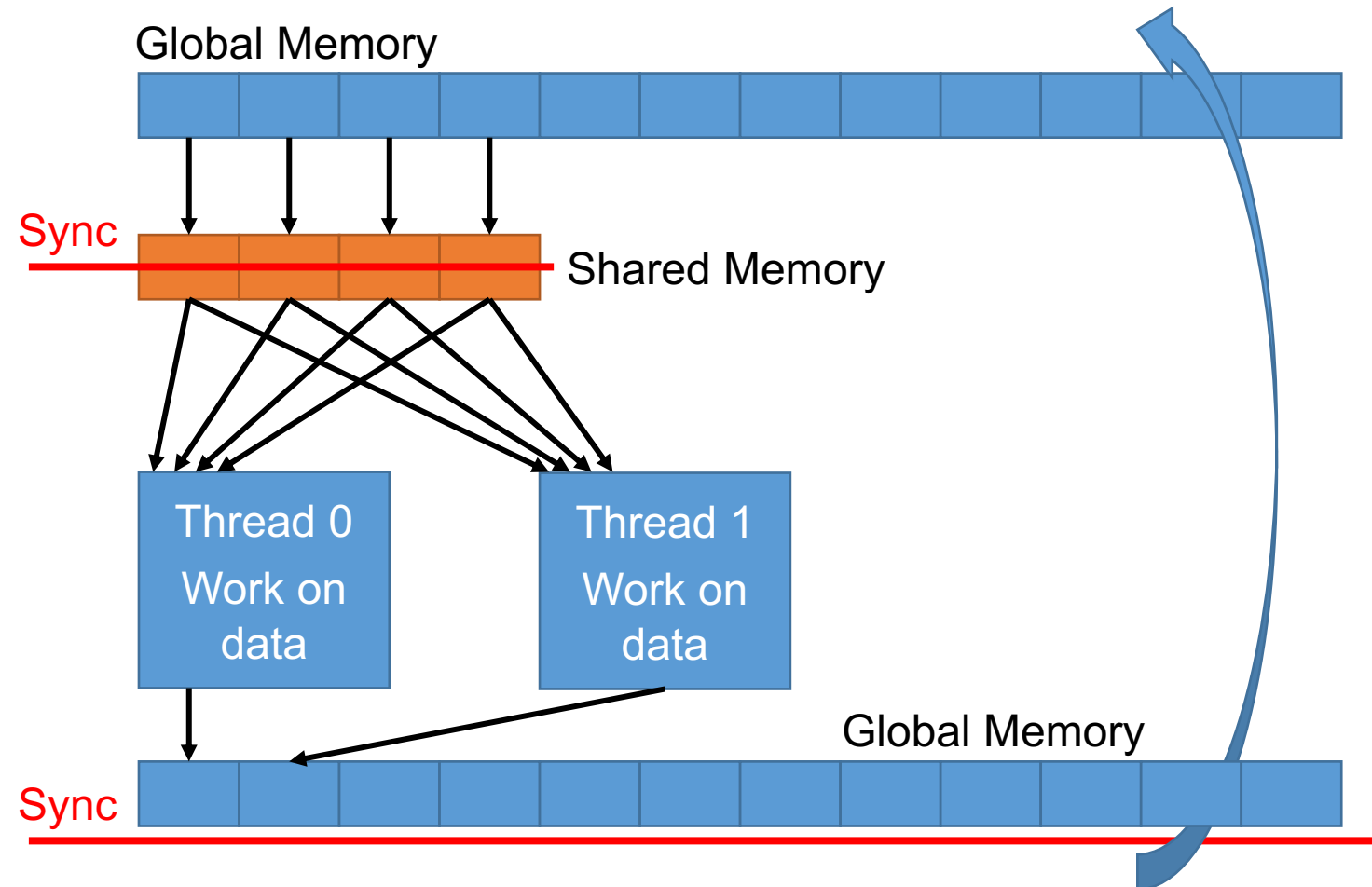
Bad: when threads have very different timing

CUDA Memories Tiling Technique

Tiling needs synchronization



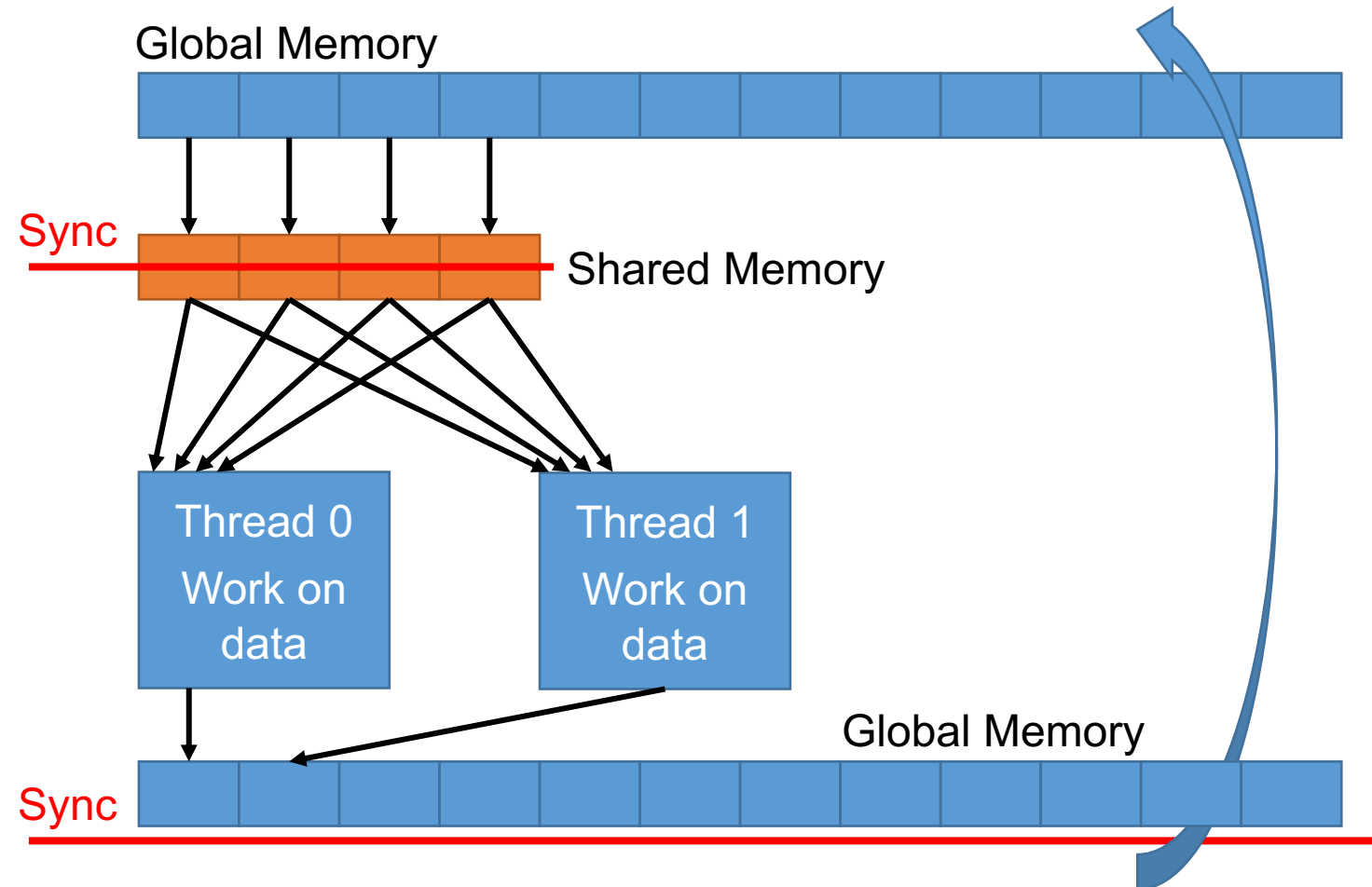
Tiling needs synchronization



Tiling Techniques step by step

- 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

Tiling needs synchronization



Barrier Synchronization

- CUDA call to synchronize all threads in a block

`__syncthreads()`

- all threads in the same block must reach the **`__syncthreads()`** before any of the them can move on
- best used to coordinate the phased execution of a tiled algorithms
 - to ensure that all elements of a tile are loaded at the beginning of a phase
 - to ensure that all elements of a tile are consumed at the end of a phase

CUDA Memories

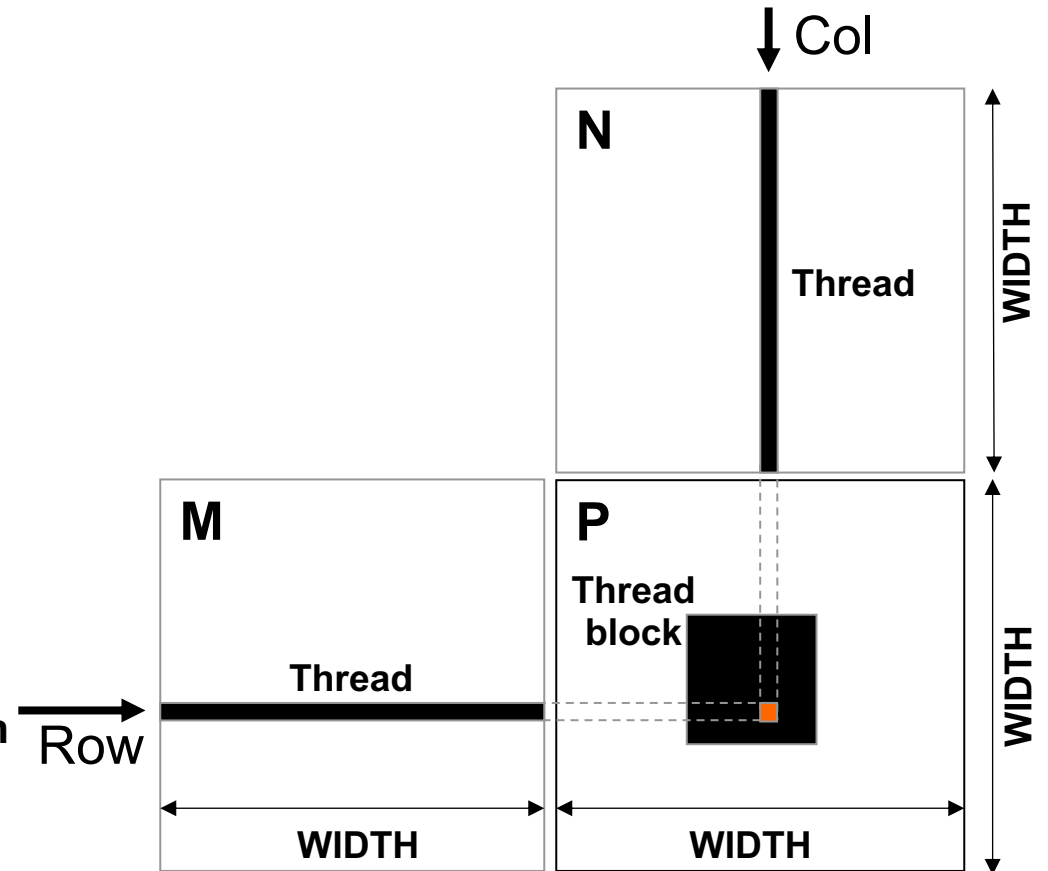
Tiled Matrix Multiplication

Global Memory is not fast enough !!!

Example: Matrix multiplication from Global Memory

- all threads access global memory for their input matrix elements
 - **two** memory accesses (4 bytes) per **two** floating-point operations (multiplication and addition)
 - algorithm needs **4B for every FLOP**
- Assume a GPU with
 - peak floating-point rate **1,600 GFLOP/second**
 - **4B*1,600 = 6,400 GB/s** required to achieve peak **FLOPS** rating
 - **600 GB/s DRAM bandwidth**
 - 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 the 1,600 GFLOPS**

Thread



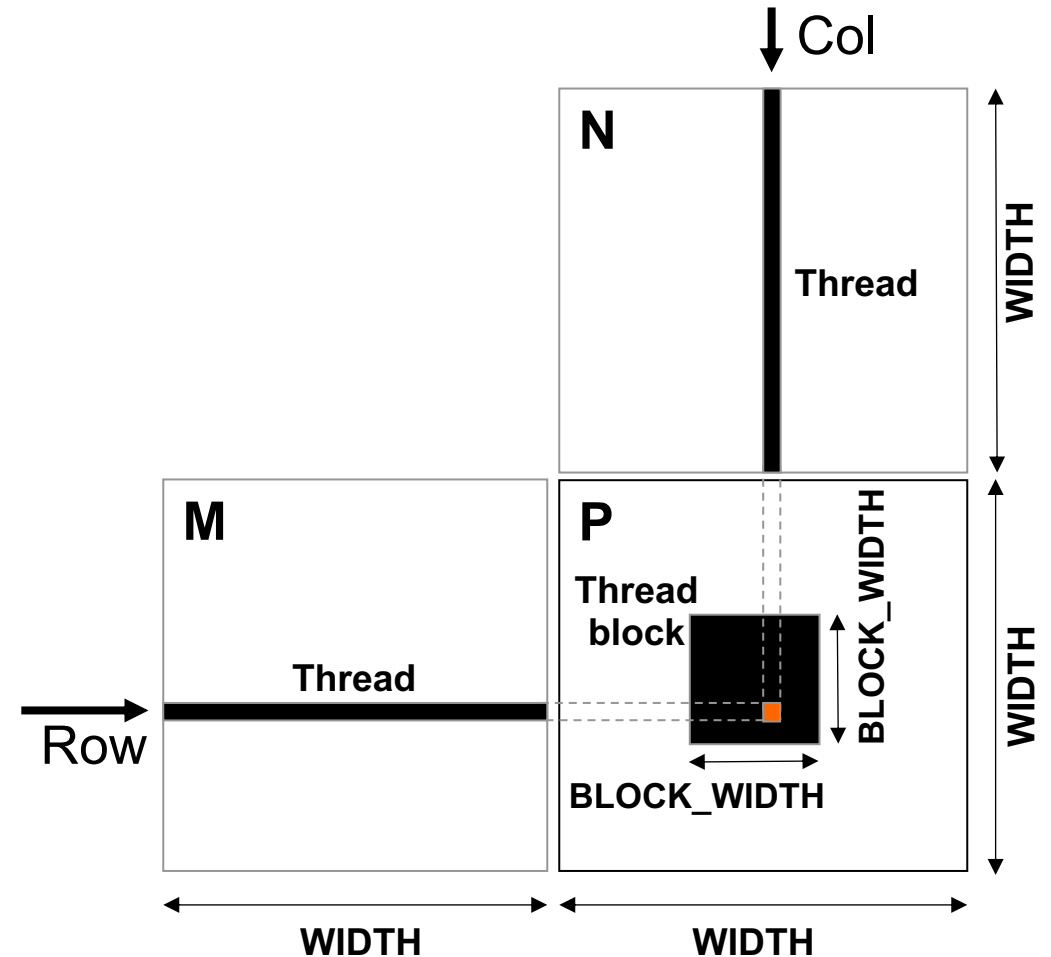
```
for (int k = 0; k < Width; ++k) {  
    Pvalue += M[Row*Width+k] * N[k*Width+Col];  
}
```

CUDA Memories

Tiled Matrix Multiplication

A Basic Matrix Multiplication

```
__global__ void MatrixMulKernel(float* M, float* N,  
                               float* P, int Width)  
{  
    // Calculate the row index of the P element and M  
    int Row = blockIdx.y*blockDim.y+threadIdx.y;  
  
    // Calculate the column index of P and N  
    int Col = blockIdx.x*blockDim.x+threadIdx.x;  
  
    if ((Row < Width) && (Col < Width)) {  
        float Pvalue = 0;  
        // each thread computes one element of the  
        // block sub-matrix  
        for (int k = 0; k < Width; ++k) {  
            Pvalue += M[Row*Width+k]*N[k*Width+Col];  
        }  
        P[Row*Width+Col] = Pvalue;  
    }  
}
```



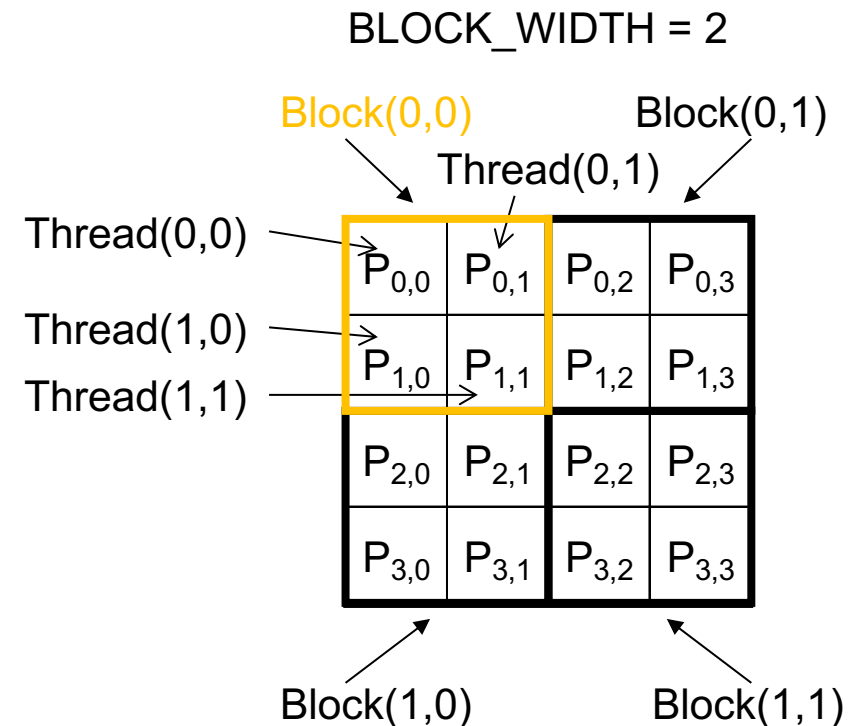
Solution – use tiling to reuse data in Shared Memory

CUDA Memories

Tiled Matrix Multiplication

A Basic Matrix Multiplication

```
__global__ void MatrixMulKernel(float* M, float* N,  
                                float* P, int Width)  
{  
    // Calculate the row index of the P element and M  
    int Row = blockIdx.y*blockDim.y+threadIdx.y;  
  
    // Calculate the column index of P and N  
    int Col = blockIdx.x*blockDim.x+threadIdx.x;  
  
    if ((Row < Width) && (Col < Width)) {  
        float Pvalue = 0;  
        // each thread computes one element of the  
        // block sub-matrix  
        for (int k = 0; k < Width; ++k) {  
            Pvalue += M[Row*Width+k]*N[k*Width+Col];  
        }  
        P[Row*Width+Col] = Pvalue;  
    }  
}
```

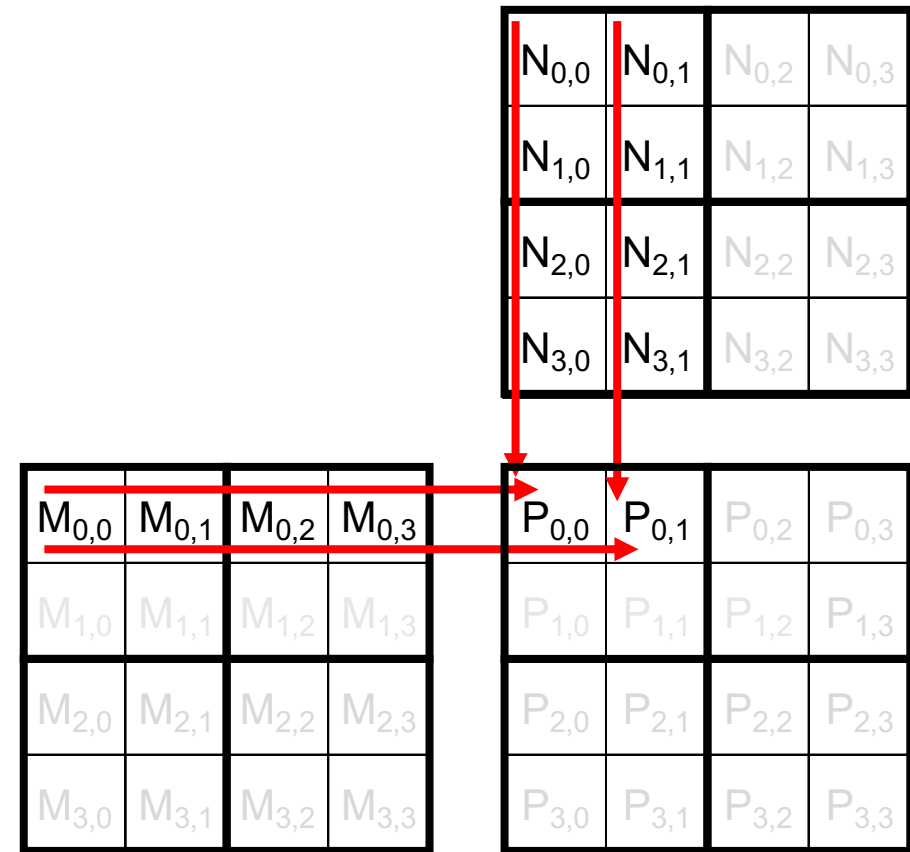


CUDA Memories

Tiled Matrix Multiplication

A Basic Matrix Multiplication

```
__global__ void MatrixMulKernel(float* M, float* N,  
                               float* P, int Width)  
{  
    // Calculate the row index of the P element and M  
    int Row = blockIdx.y*blockDim.y+threadIdx.y;  
  
    // Calculate the column index of P and N  
    int Col = blockIdx.x*blockDim.x+threadIdx.x;  
  
    if ((Row < Width) && (Col < Width)) {  
        float Pvalue = 0;  
        // each thread computes one element of the  
        // block sub-matrix  
        for (int k = 0; k < Width; ++k) {  
            Pvalue += M[Row*Width+k]*N[k*Width+Col];  
        }  
        P[Row*Width+Col] = Pvalue;  
    }  
}
```

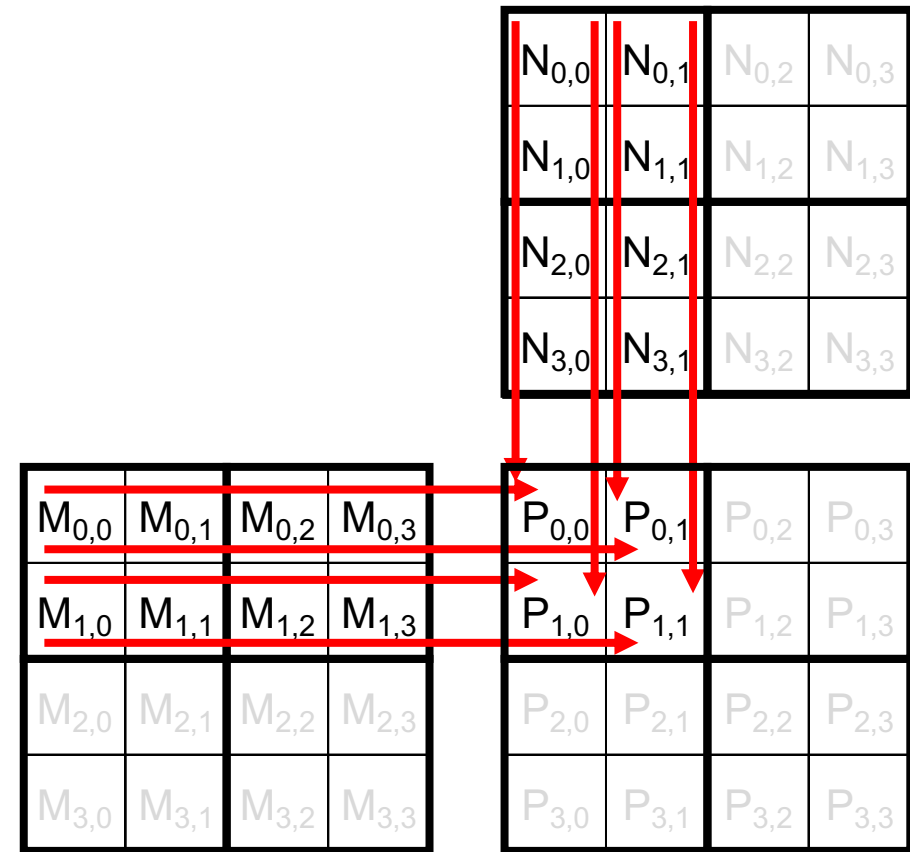


CUDA Memories

Tiled Matrix Multiplication

A Basic Matrix Multiplication

```
__global__ void MatrixMulKernel(float* M, float* N,  
                                float* P, int Width)  
{  
    // Calculate the row index of the P element and M  
    int Row = blockIdx.y*blockDim.y+threadIdx.y;  
  
    // Calculate the column index of P and N  
    int Col = blockIdx.x*blockDim.x+threadIdx.x;  
  
    if ((Row < Width) && (Col < Width)) {  
        float Pvalue = 0;  
        // each thread computes one element of the  
        // block sub-matrix  
        for (int k = 0; k < Width; ++k) {  
            Pvalue += M[Row*Width+k]*N[k*Width+Col];  
        }  
        P[Row*Width+Col] = Pvalue;  
    }  
}
```

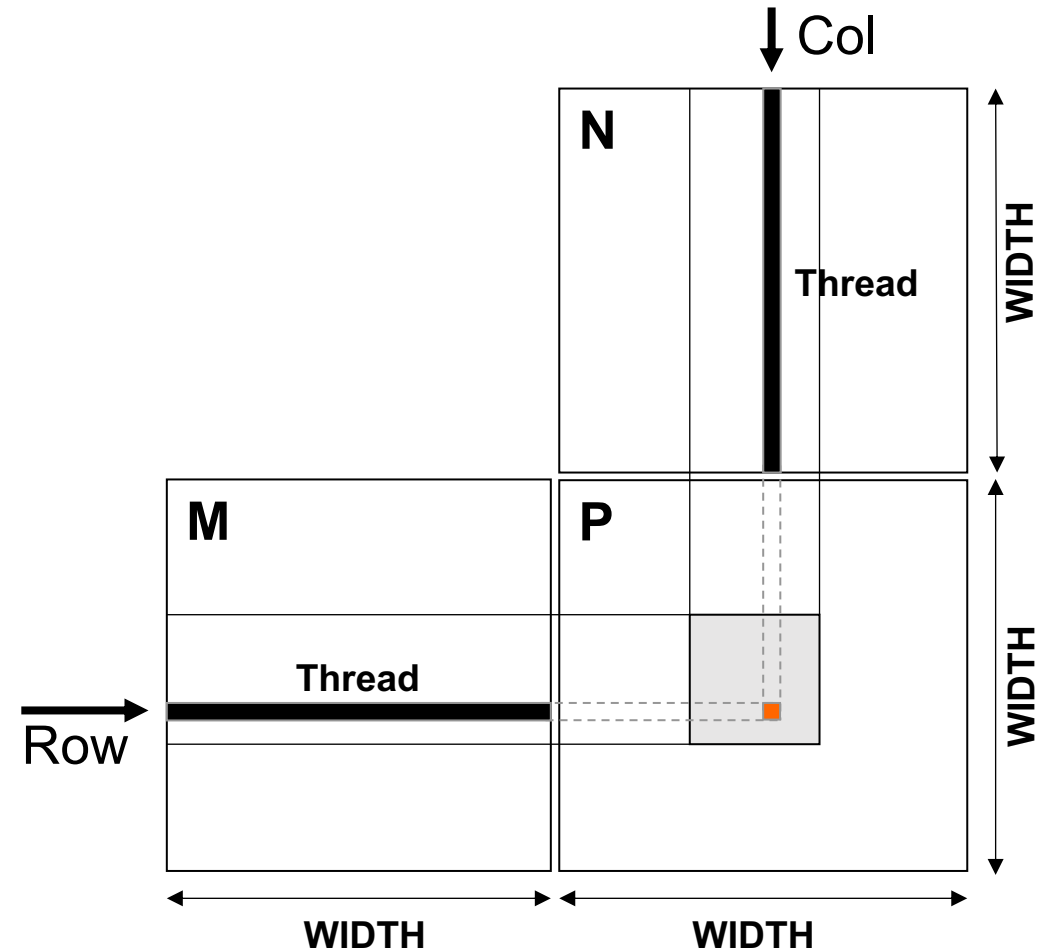


CUDA Memories

Tiled Matrix Multiplication

Data access pattern

- each thread - a row of M and a column of N
- each thread block – a strip of M and a strip of N



CUDA Memories

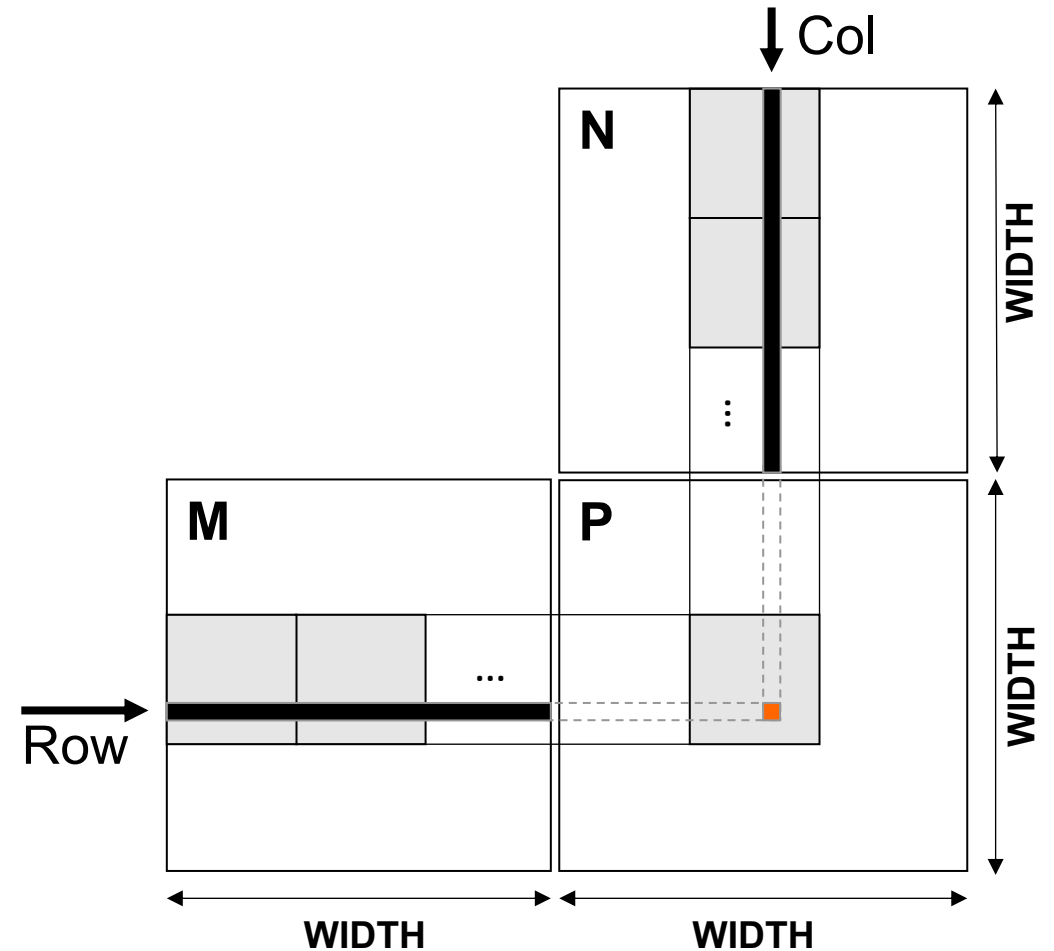
Tiled Matrix Multiplication

Data access pattern

- each thread - a row of M and a column of N
- each thread block – a strip of M and a strip of N

Tiled Matrix Multiplication

- break up the execution of each thread into phases
- so that the data accesses by the thread block in each phase are focused on one tile of M and one tile of N
- the tile is of BLOCK_SIZE elements in each dimension



CUDA Memories

Tiled Matrix Multiplication

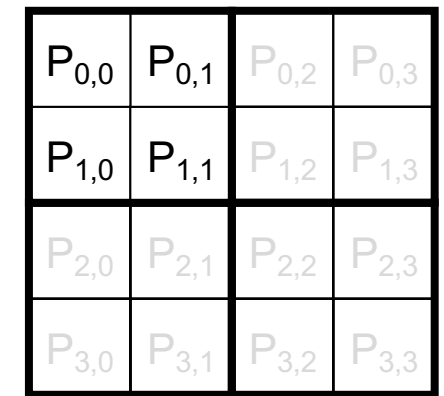
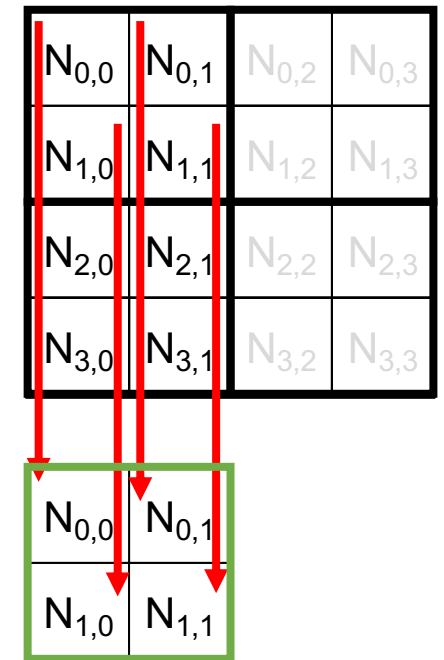
Data access pattern

			Phase 0
Thread (0,0)	$M_{0,0}$ ↓ $MS_{0,0}$	$N_{0,0}$ ↓ $NS_{0,0}$	
Thread (0,1)	$M_{0,1}$ ↓ $MS_{0,1}$	$N_{0,1}$ ↓ $NS_{0,1}$	
Thread (1,0)	$M_{1,0}$ ↓ $MS_{1,0}$	$N_{1,0}$ ↓ $NS_{1,0}$	
Thread (1,1)	$M_{1,1}$ ↓ $MS_{1,1}$	$N_{1,1}$ ↓ $NS_{1,1}$	

Phase 0 Load for Block (0,0)



Shared
Memory



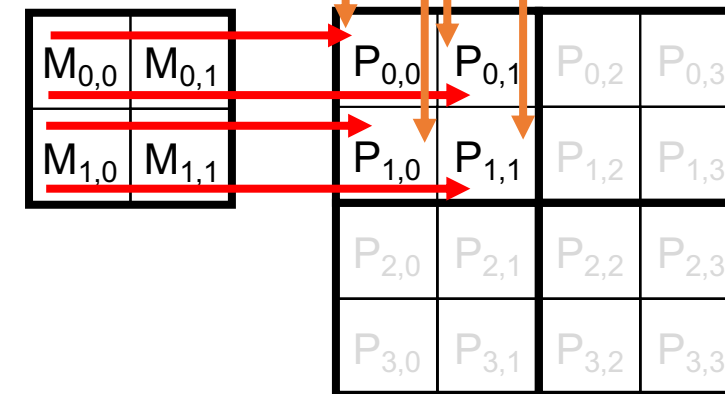
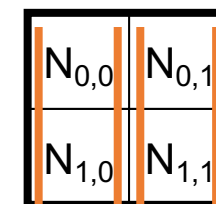
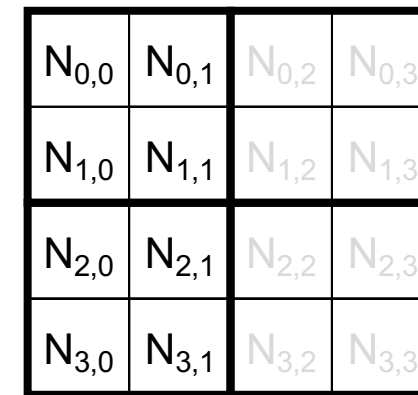
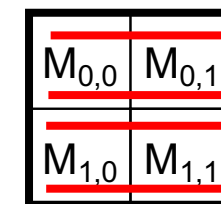
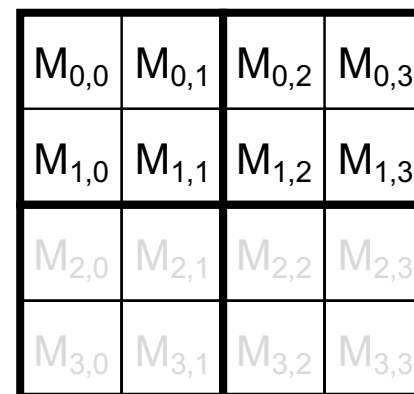
CUDA Memories

Tiled Matrix Multiplication

Data access pattern

			Phase 0
Thread (0,0)	$M_{0,0}$ ↓ $MS_{0,0}$	$N_{0,0}$ ↓ $NS_{0,0}$	$PValue_{0,0} += MS_{0,0} * NS_{0,0}$
Thread (0,1)	$M_{0,1}$ ↓ $MS_{0,1}$	$N_{0,1}$ ↓ $NS_{0,1}$	$PValue_{0,1} += MS_{0,0} * NS_{0,1}$
Thread (1,0)	$M_{1,0}$ ↓ $MS_{1,0}$	$N_{1,0}$ ↓ $NS_{1,0}$	$PValue_{1,0} += MS_{1,0} * NS_{0,0}$
Thread (1,1)	$M_{1,1}$ ↓ $MS_{1,1}$	$N_{1,1}$ ↓ $NS_{1,1}$	$PValue_{1,1} += MS_{1,0} * NS_{0,1}$

Phase 0 Use for Block (0,0) (iteration 0)



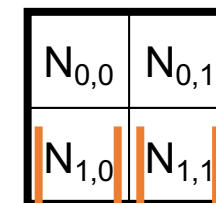
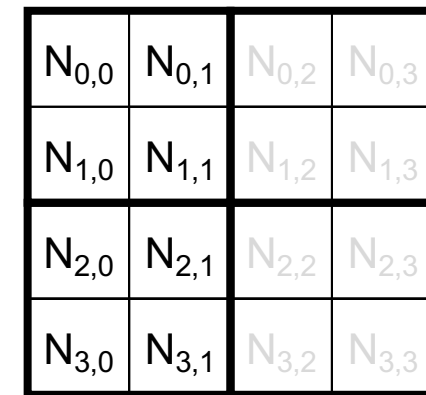
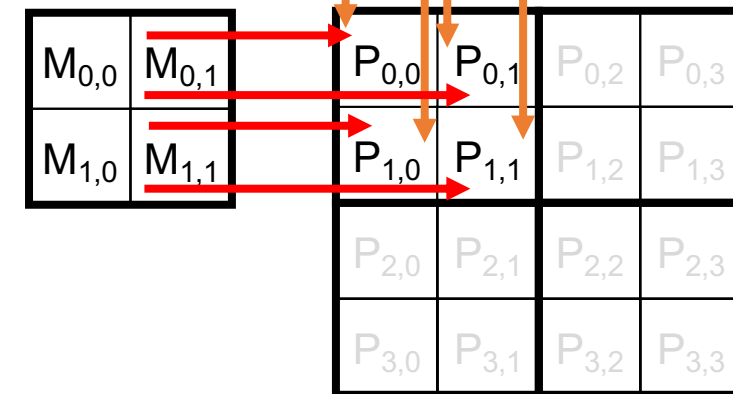
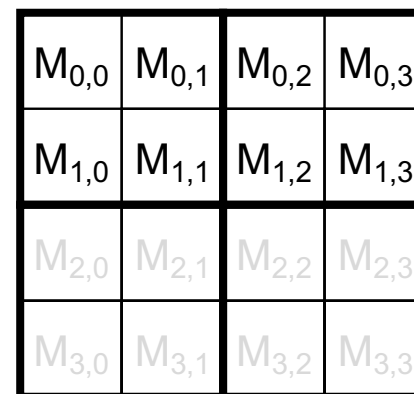
CUDA Memories

Tiled Matrix Multiplication

Data access pattern

			Phase 0
Thread (0,0)	$M_{0,0}$ ↓ $MS_{0,0}$	$N_{0,0}$ ↓ $NS_{0,0}$	$PValue_{0,0} += MS_{0,0} * NS_{0,0} + MS_{0,1} * NS_{1,0}$
Thread (0,1)	$M_{0,1}$ ↓ $MS_{0,1}$	$N_{0,1}$ ↓ $NS_{0,1}$	$PValue_{0,1} += MS_{0,0} * NS_{0,1} + MS_{0,1} * NS_{1,1}$
Thread (1,0)	$M_{1,0}$ ↓ $MS_{1,0}$	$N_{1,0}$ ↓ $NS_{1,0}$	$PValue_{1,0} += MS_{1,0} * NS_{0,0} + MS_{1,1} * NS_{1,0}$
Thread (1,1)	$M_{1,1}$ ↓ $MS_{1,1}$	$N_{1,1}$ ↓ $NS_{1,1}$	$PValue_{1,1} += MS_{1,0} * NS_{0,1} + MS_{1,1} * NS_{1,1}$

Phase 0 Use for Block (0,0) (iteration 1)



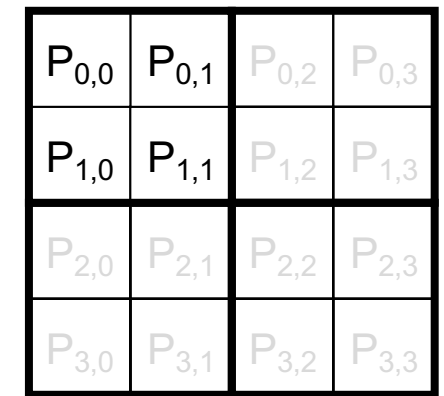
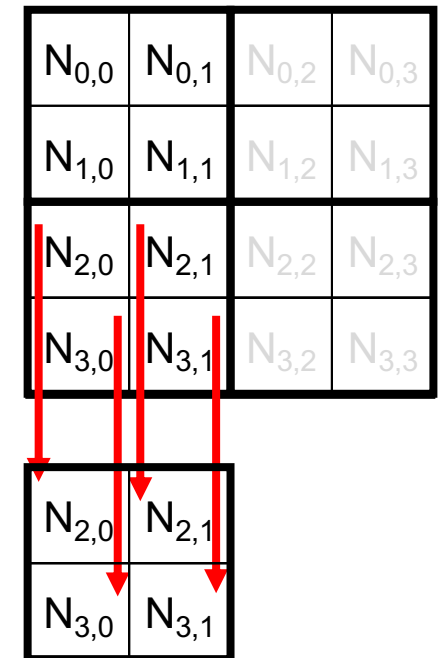
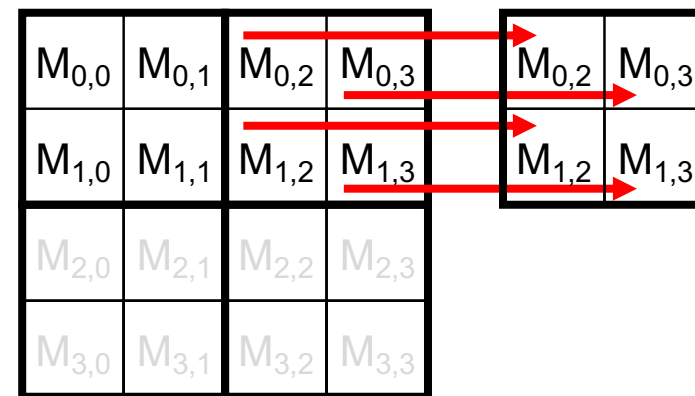
CUDA Memories

Tiled Matrix Multiplication

Data access pattern

			Phase 1
Thread (0,0)	$M_{0,2}$ ↓ $MS_{0,0}$	$N_{2,0}$ ↓ $NS_{0,0}$	
Thread (0,1)	$M_{0,3}$ ↓ $MS_{0,1}$	$N_{2,1}$ ↓ $NS_{0,1}$	
Thread (1,0)	$M_{1,2}$ ↓ $MS_{1,0}$	$N_{3,0}$ ↓ $NS_{1,0}$	
Thread (1,1)	$M_{1,3}$ ↓ $MS_{1,1}$	$N_{3,1}$ ↓ $NS_{1,1}$	

Phase 1 Load for Block (0,0)



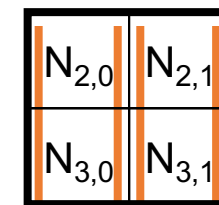
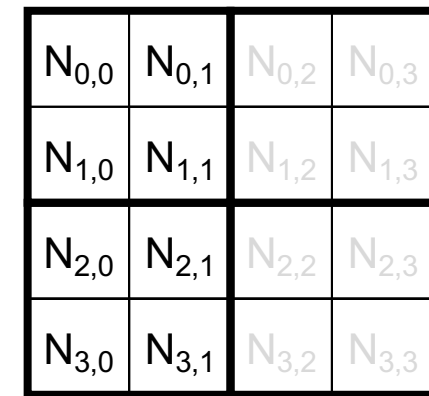
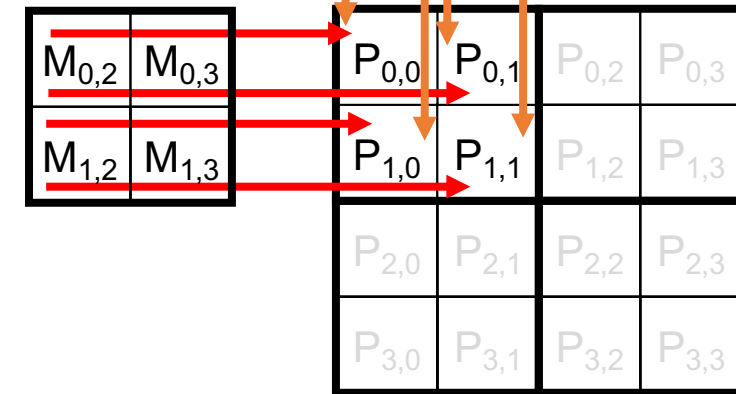
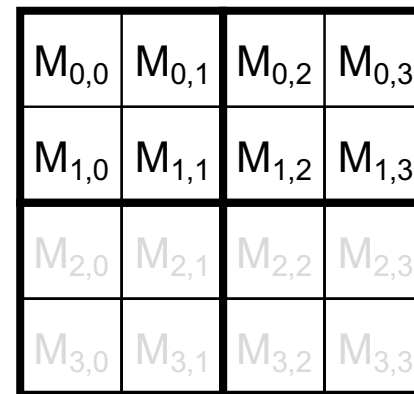
CUDA Memories

Tiled Matrix Multiplication

Data access pattern

			Phase 1
Thread (0,0)	$M_{0,2}$ ↓ $MS_{0,0}$	$N_{2,0}$ ↓ $NS_{0,0}$	$PValue_{0,0} += MS_{0,0} * NS_{0,0}$
Thread (0,1)	$M_{0,3}$ ↓ $MS_{0,1}$	$N_{2,1}$ ↓ $NS_{0,1}$	$PValue_{0,1} += MS_{0,0} * NS_{0,1}$
Thread (1,0)	$M_{1,2}$ ↓ $MS_{1,0}$	$N_{3,0}$ ↓ $NS_{1,0}$	$PValue_{1,0} += MS_{1,0} * NS_{0,0}$
Thread (1,1)	$M_{1,3}$ ↓ $MS_{1,1}$	$N_{3,1}$ ↓ $NS_{1,1}$	$PValue_{1,1} += MS_{1,0} * NS_{0,1}$

Phase 1 Use for Block (0,0) (iteration 0)



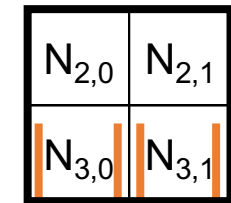
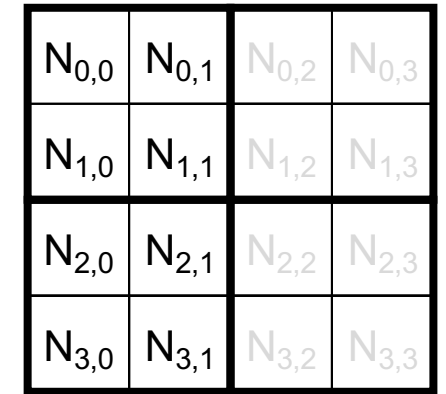
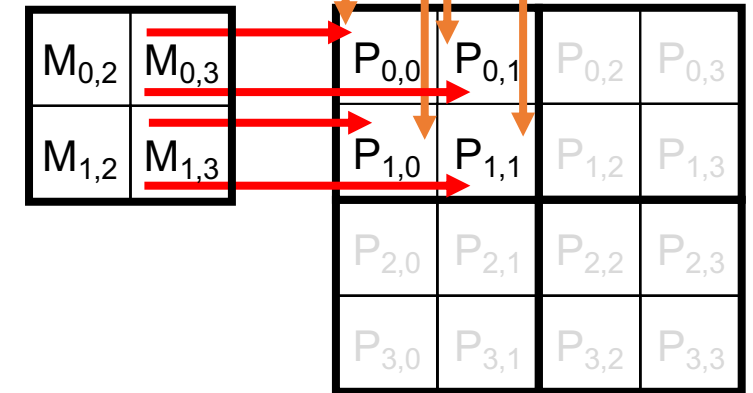
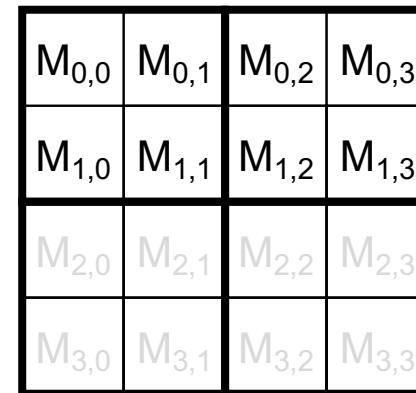
CUDA Memories

Tiled Matrix Multiplication

Data access pattern

			Phase 1
Thread (0,0)	$M_{0,2}$ ↓ $MS_{0,0}$	$N_{2,0}$ ↓ $NS_{0,0}$	$PValue_{0,0} += MS_{0,0} * NS_{0,0} + MS_{0,1} * NS_{1,0}$
Thread (0,1)	$M_{0,3}$ ↓ $MS_{0,1}$	$N_{2,1}$ ↓ $NS_{0,1}$	$PValue_{0,1} += MS_{0,0} * NS_{0,1} + MS_{0,1} * NS_{1,1}$
Thread (1,0)	$M_{1,2}$ ↓ $MS_{1,0}$	$N_{3,0}$ ↓ $NS_{1,0}$	$PValue_{1,0} += MS_{1,0} * NS_{0,0} + MS_{1,1} * NS_{1,0}$
Thread (1,1)	$M_{1,3}$ ↓ $MS_{1,1}$	$N_{3,1}$ ↓ $NS_{1,1}$	$PValue_{1,1} += MS_{1,0} * NS_{0,1} + MS_{1,1} * NS_{1,1}$

Phase 1 Use for Block (0,0) (iteration 1)



CUDA Memories

Tiled Matrix Multiplication

			Phase 0
Thread (0,0)	$M_{0,0}$ ↓ $MS_{0,0}$	$N_{0,0}$ ↓ $NS_{0,0}$	$PValue_{0,0} +=$ $MS_{0,0}$ * $NS_{0,0} + MS_{0,1} * NS_{1,0}$
Thread (0,1)	$M_{0,1}$ ↓ $MS_{0,1}$	$N_{0,1}$ ↓ $NS_{0,1}$	$PValue_{0,1} +=$ $MS_{0,0}$ * $NS_{0,1} + MS_{0,1} * NS_{1,1}$
Thread (1,0)	$M_{1,0}$ ↓ $MS_{1,0}$	$N_{1,0}$ ↓ $NS_{1,0}$	$PValue_{1,0} +=$ $MS_{1,0} * NS_{0,0} + MS_{1,1} * NS_{1,0}$
Thread (1,1)	$M_{1,1}$ ↓ $MS_{1,1}$	$N_{1,1}$ ↓ $NS_{1,1}$	$PValue_{1,1} +=$ $MS_{1,0} * NS_{0,1} + MS_{1,1} * NS_{1,1}$

			Phase 1
Thread (0,0)	$M_{0,2}$ ↓ $MS_{0,0}$	$N_{2,0}$ ↓ $NS_{0,0}$	$PValue_{0,0} +=$ $MS_{0,0} * NS_{0,0} + MS_{0,1} * NS_{1,0}$
Thread (0,1)	$M_{0,3}$ ↓ $MS_{0,1}$	$N_{2,1}$ ↓ $NS_{0,1}$	$PValue_{0,1} +=$ $MS_{0,0} * NS_{0,1} + MS_{0,1} * NS_{1,1}$
Thread (1,0)	$M_{1,2}$ ↓ $MS_{1,0}$	$N_{3,0}$ ↓ $NS_{1,0}$	$PValue_{1,0} +=$ $MS_{1,0} * NS_{0,0} + MS_{1,1} * NS_{1,0}$
Thread (1,1)	$M_{1,3}$ ↓ $MS_{1,1}$	$N_{3,1}$ ↓ $NS_{1,1}$	$PValue_{1,1} +=$ $MS_{1,0} * NS_{0,1} + MS_{1,1} * NS_{1,1}$

Shared memory allows each value to be accessed by multiple threads.

CUDA Memories

Tiled Matrix Multiplication Kernel

Loading Input Tile 0 of M and N (Phase 0)

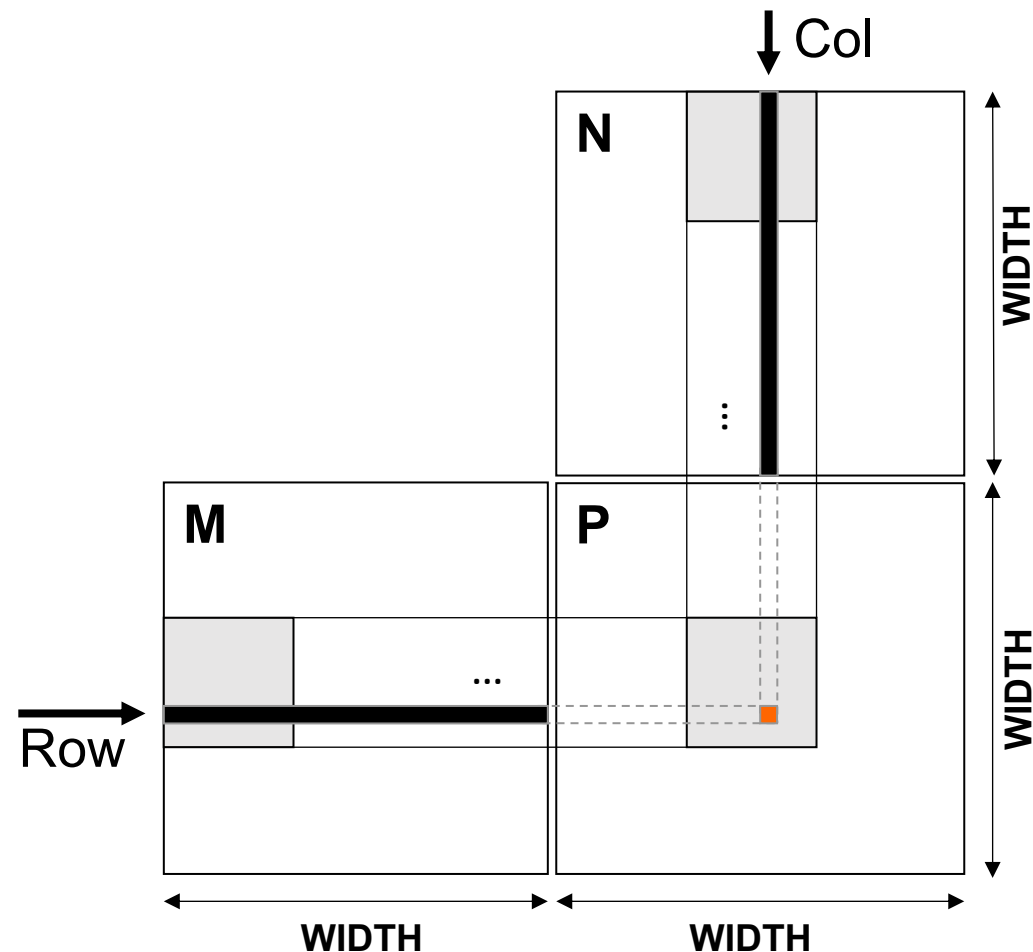
- each thread loads an M element and an N element at the same relative position as its P element

```
int bx = blockIdx.x;  int by = blockIdx.y;  
int tx = threadIdx.x; int ty = threadIdx.y;
```

```
int Row = by * blockDim.y + ty;  
int Col = bx * blockDim.x + tx;
```

2D indexing for accessing Tile 0:

```
M [Row] [tx]  
N [ty] [Col]
```



CUDA Memories

Tiled Matrix Multiplication Kernel

Loading Input Tile 1 of M and N (Phase 1)

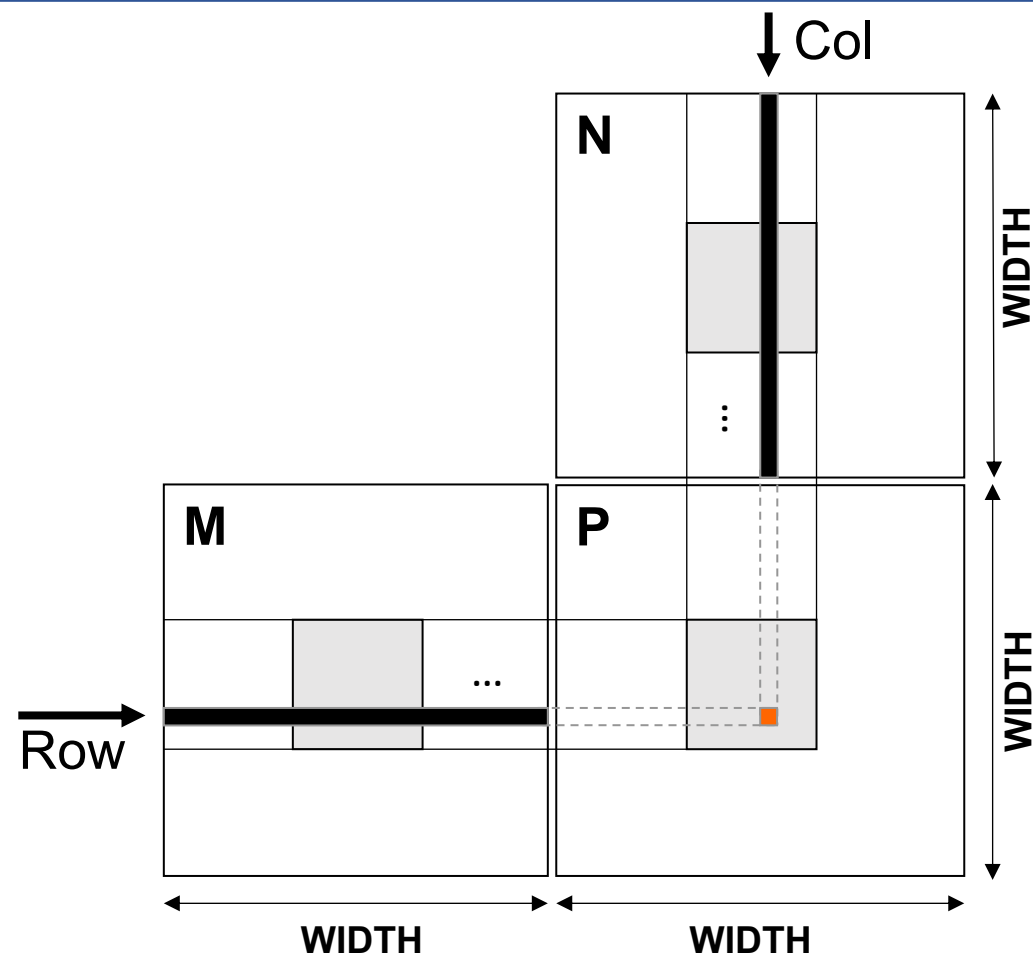
- each thread loads an M element and an N element at the same relative position as its P element

```
int bx = blockIdx.x;  int by = blockIdx.y;  
int tx = threadIdx.x; int ty = threadIdx.y;
```

```
int Row = by * blockDim.y + ty;  
int Col = bx * blockDim.x + tx;
```

2D indexing for accessing Tile 1:

```
M [Row]           [1*TILE_WIDTH + tx]  
N [1*TILE*WIDTH + ty] [Col]
```



CUDA Memories

Tiled Matrix Multiplication Kernel

Loading Input Tile **p** of M and N (Phase 1)

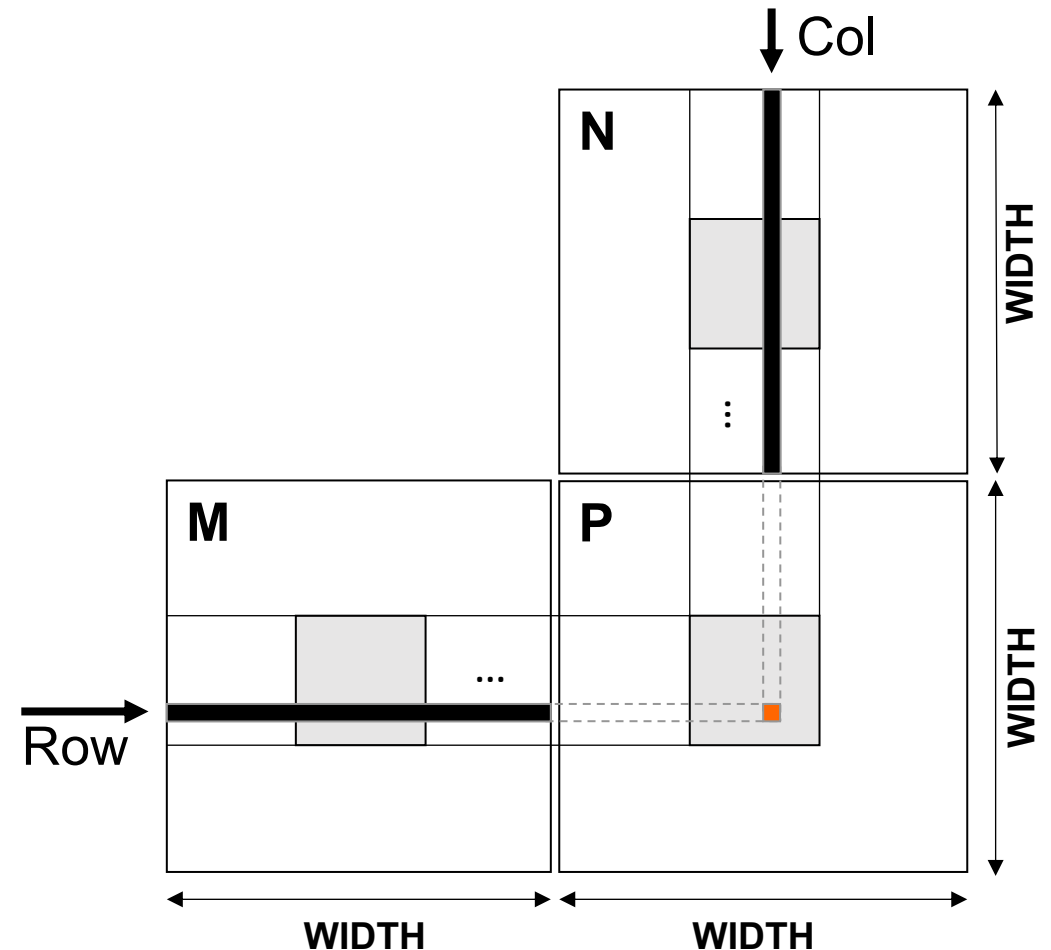
- each thread loads an M element and an N element at the same relative position as its P element

```
int bx = blockIdx.x;  int by = blockIdx.y;  
int tx = threadIdx.x; int ty = threadIdx.y;
```

```
int Row = by * blockDim.y + ty;  
int Col = bx * blockDim.x + tx;
```

2D indexing for accessing Tile **p**:

```
M [Row]          [p*TILE_WIDTH + tx]  
N [p*TILE*WIDTH + ty] [Col]
```



CUDA Memories

Tiled Matrix Multiplication Kernel

Loading Input Tile **p** of **M** and **N** (Phase 1)

- each thread loads an **M** element and an **N** element at the same relative position as its **P** element

```
int bx = blockIdx.x;  int by = blockIdx.y;  
int tx = threadIdx.x; int ty = threadIdx.y;
```

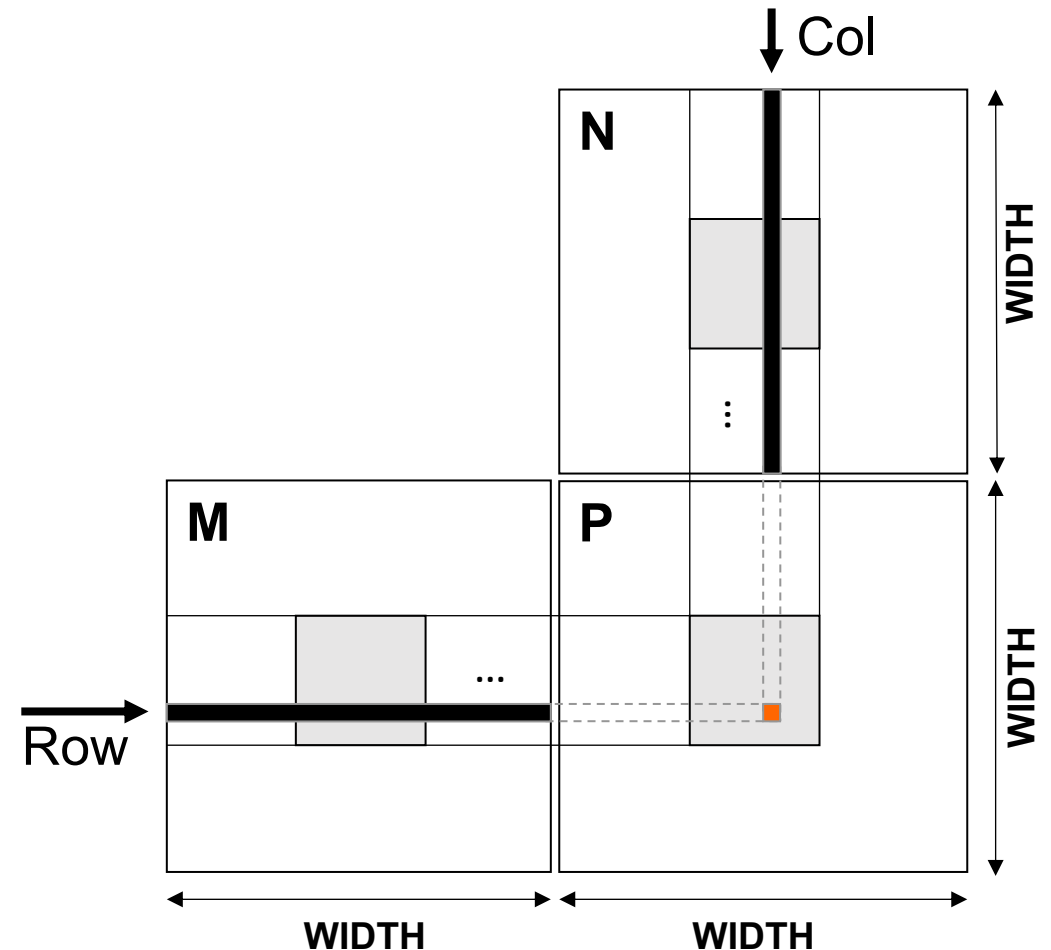
```
int Row = by * blockDim.y + ty;  
int Col = bx * blockDim.x + tx;
```

2D indexing for accessing Tile **p**:

```
M [Row]                [p*TILE_WIDTH + tx]  
N [p*TILE*WIDTH + ty] [Col]
```

1D indexing for accessing Tile **p**:

```
M [Row*Width + p*TILE_WIDTH + tx]  
N [(p*TILE_WIDTH+ty)*Width + Col]
```



CUDA Memories

Tiled Matrix Multiplication Kernel

Matrix Multiplication kernel

- two additional declaration of SM array ds_M a ds_N
 - Tiles of M and N

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width)
{
    __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
    __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x; int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

    int Row = by * blockDim.y + ty;
    int Col = bx * blockDim.x + tx;
    float Pvalue = 0;

    // Loop over the M and N tiles required to compute the P element
    for (int p = 0; p < n/TILE_WIDTH; ++p) {
        // Collaborative loading of M and N tiles into shared memory
        ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
        ds_N[ty][tx] = N[(p*TILE_WIDTH+ty)*Width + Col];
        __syncthreads();

        for (int i = 0; i < TILE_WIDTH; ++i)
            Pvalue += ds_M[ty][i] * ds_N[i][tx];

        __syncthreads();
    }
    P[Row*Width+Col] = Pvalue;
}
```

CUDA Memories

Tiled Matrix Multiplication Kernel

Matrix Multiplication kernel

- loop defines the phases of the mat. mult. kernel
 - each iteration corresponds to a phase
 - P variable – indicates the number of the current phase
- each thread loads one element of M and N
- these elements are moved to SM arrays ds_M and ds_N
- `__syncthreads()` is needed because threads can execute in different timings

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width)
{
    __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
    __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x; int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

    int Row = by * blockDim.y + ty;
    int Col = bx * blockDim.x + tx;
    float Pvalue = 0;

    // Loop over the M and N tiles required to compute the P element
    for (int p = 0; p < n/TILE_WIDTH; ++p) {
        // Collaborative loading of M and N tiles into shared memory
        ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
        ds_N[ty][tx] = N[(p*TILE_WIDTH+ty)*Width + Col];
        __syncthreads();

        for (int i = 0; i < TILE_WIDTH; ++i)
            Pvalue += ds_M[ty][i] * ds_N[i][tx];
        __syncthreads();
    }
    P[Row*Width+Col] = Pvalue;
}
```

CUDA Memories

Tiled Matrix Multiplication Kernel

Matrix Multiplication kernel

- selected loop performs the execution of the inner product in a current phase based on the content of the SM
- `__syncthreads()`
 - makes sure all threads finished the calculation
 - and no threads need content of SM anymore
 - after the sync, we can rewrite the content of the SM with new data for next phase
- Finally, each thread writes its output value in the P matrix

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width)
{
    __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
    __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x; int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

    int Row = by * blockDim.y + ty;
    int Col = bx * blockDim.x + tx;
    float Pvalue = 0;

    // Loop over the M and N tiles required to compute the P element
    for (int p = 0; p < n/TILE_WIDTH; ++p) {
        // Collaborative loading of M and N tiles into shared memory
        ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
        ds_N[ty][tx] = N[(p*TILE_WIDTH+ty)*Width + Col];
        __syncthreads();

        for (int i = 0; i < TILE_WIDTH; ++i)
            Pvalue += ds_M[ty][i] * ds_N[i][tx];
        __syncthreads();
    }
    P[Row*Width+Col] = Pvalue;
}
```

CUDA Memories

Tiled Matrix Multiplication Kernel

Performance considerations

- Each thread block should have many threads
 - TILE_WIDTH of **16** gives **$16*16 = 256$ threads**
 - TILE_WIDTH of **32** gives **$32*32 = 1024$ threads**
- **for TILE_WIDTH = 16,**
 - in each phase, each block performs
 - $2*256 = 512$ float loads from global memory for
 - $256 * (2*16) = 8,192$ mul/add operations
 - **16 floating-point operations for each memory load**
- **for TILE_WIDTH = 32,**
 - in each phase, each block performs
 - $2*1024 = 2048$ float loads from global memory for
 - $1024 * (2*32) = 65,536$ mul/add operations
 - **32 floating-point operation for each memory load**

CUDA Memories

Tiled Matrix Multiplication Kernel

Performance considerations

- Each thread block should have many threads
 - TILE_WIDTH of **16** gives **$16*16 = 256$ threads**
 - TILE_WIDTH of **32** gives **$32*32 = 1024$ threads**
- **for TILE_WIDTH = 16,**
 - in each phase, each block performs
 - $2*256 = 512$ float loads from global memory for
 - $256 * (2*16) = 8,192$ mul/add operations
 - **16 floating-point operations for each memory load**
- **for TILE_WIDTH = 32,**
 - in each phase, each block performs
 - $2*1024 = 2048$ float loads from global memory for
 - $1024 * (2*32) = 65,536$ mul/add operations
 - **32 floating-point operation for each memory load**

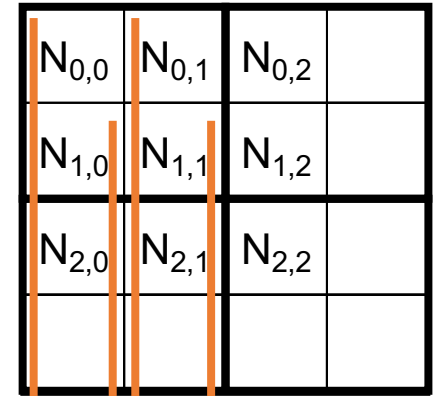
Shared Memory impact

- For example, **let's have an SM with 16KB shared memory**
 - Shared memory size is implementation dependent!
 - GA102 – up to 100kB per SM
 - GA100 – up to 164kB per SM
- **For TILE_WIDTH = 16,**
 - each thread block uses $2*16*16*4B = 2KB$ of shared memory
 - for 16KB shared memory per SM, one SM
 - can have up to 8 thread blocks executing
 - this allows up to $8*512 = 4,096$ pending loads
 - 2 per thread, 256 threads per block
- **For TILE_WIDTH = 32**
 - each thread block uses $2*32*32*4B = 8KB$ of shared memory
 - one SM can have 2 thread blocks active at the same time
 - one have to check maximum number of threads per block (1024, 1536 or 2048) architecture dependent

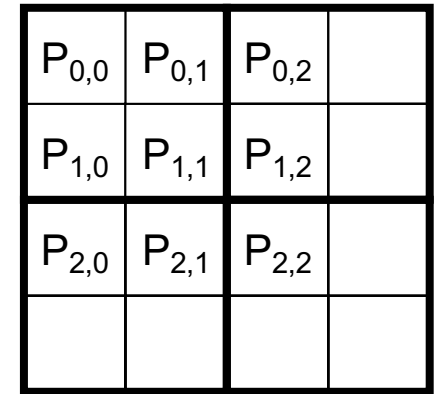
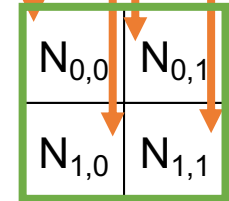
Tiled Matrix Multiplication Matrices of Arbitrary Size

Data access pattern

Phase 0 Load for Block (0,0)



Shared
Memory



Tiled Matrix Multiplication Matrices of Arbitrary Size

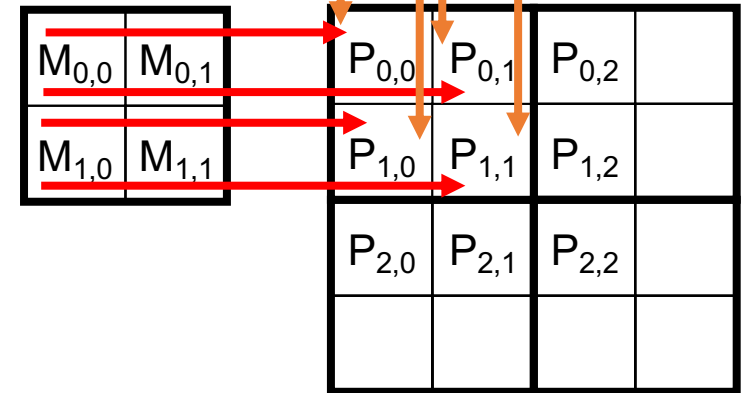
Data access pattern

Phase 0 Use for Block (0,0)
(iteration 0)

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	

$N_{0,0}$	$N_{0,1}$
$N_{1,0}$	$N_{1,1}$

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



Tiled Matrix Multiplication Matrices of Arbitrary Size

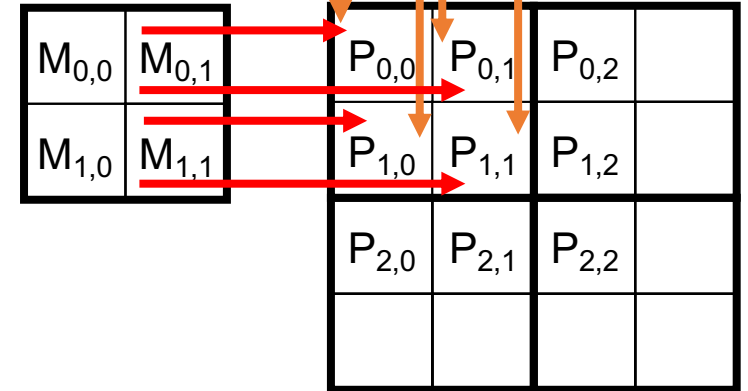
Data access pattern

Phase 0 Use for Block (0,0)
(iteration 1)

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	

$N_{0,0}$	$N_{0,1}$
$N_{1,0}$	$N_{1,1}$

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

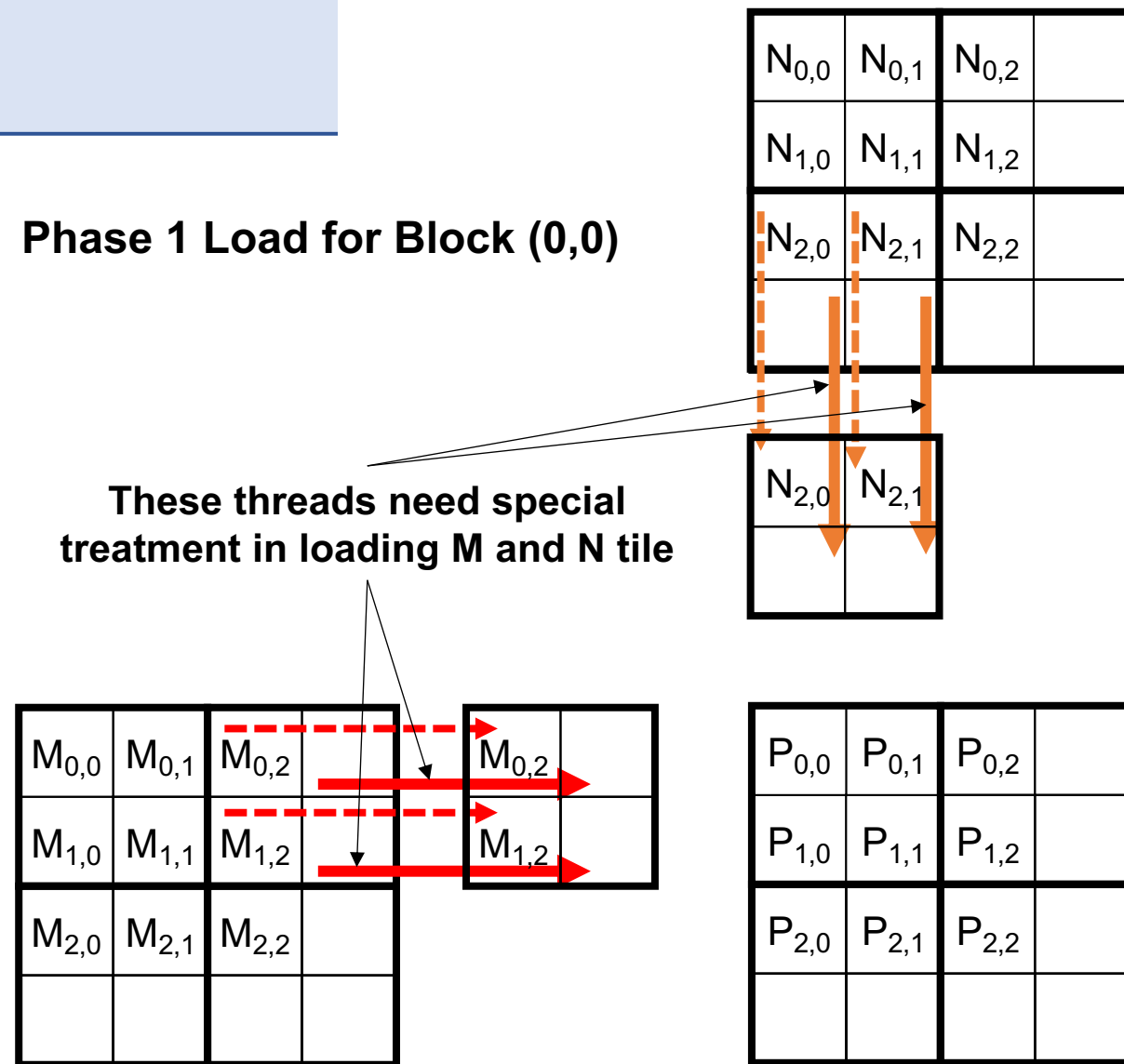


Tiled Matrix Multiplication Matrices of Arbitrary Size

Basic kernel limitations

- the tiled matrix multiplication kernel can handle only
 - square matrices
 - dimensions are multiples of the TILE_WIDTH
- real applications need to handle arbitrary sized matrices

Phase 1 Load for Block (0,0)

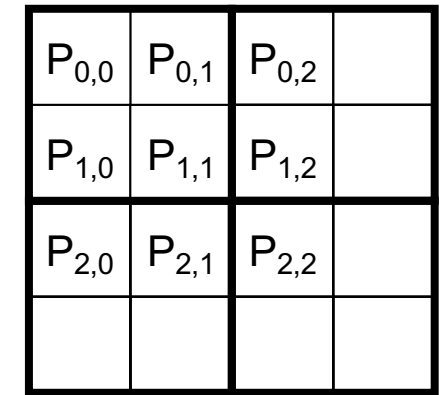
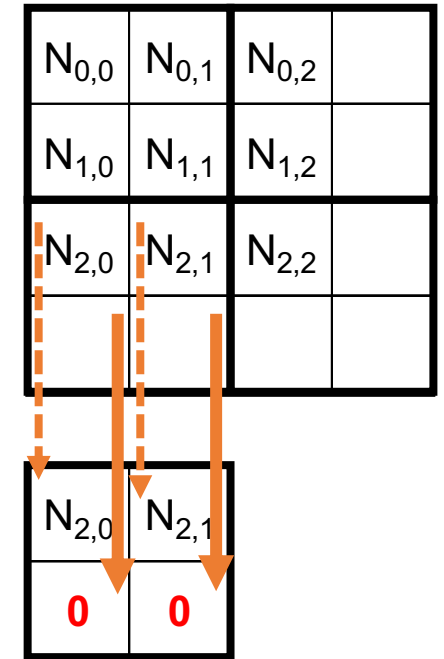
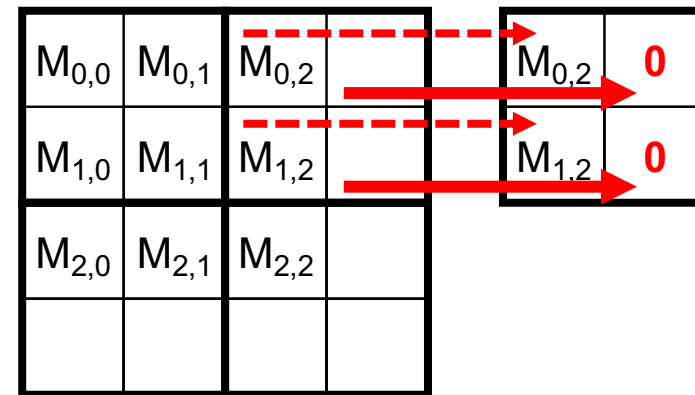


Tiled Matrix Multiplication Matrices of Arbitrary Size

Solving problem during loading data into tile

- when a thread is to load an input element
 - **test if it is in the valid index range**
 - **if valid, proceed to load**
 - **else, do not load, and write a 0 to SM**
- **Rationale:** a 0 value will ensure that the multiply-add step does not affect the final value of the output element
- the condition tested for loading input elements is different from the test for calculating output P element
- a thread that does not calculate valid P element can still participate in loading input tile elements

Phase 1 Load for Block (0,0)



Tiled Matrix Multiplication Matrices of Arbitrary Size

Tile processing

- if SM buffers are loaded correctly, the Tile processing remains unaffected

Phase 1 Use for Block (0,0) (iteration 0)

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

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

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	

$N_{2,0}$	$N_{2,1}$
0	0

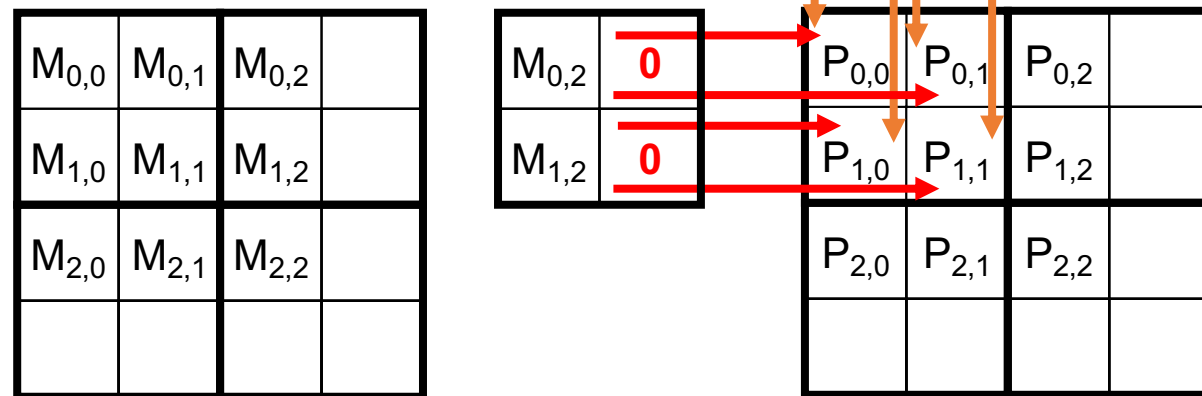
$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	

Tiled Matrix Multiplication Matrices of Arbitrary Size

Tile processing

- if SM buffers are loaded correctly, the Tile processing remains unaffected

Phase 1 Use for Block (0,0) (iteration 1)



CUDA Memories

Tiled Matrix Multiplication Kernel

Loading Input Tile **p** of **M**

- each thread loads an M element and an N element at
- the same relative position as its P element

```
int bx = blockIdx.x;  int by = blockIdx.y;  
int tx = threadIdx.x; int ty = threadIdx.y;  
int Row = by * blockDim.y + ty;  
int Col = bx * blockDim.x + tx;
```

2D indexing for accessing Tile **p**:

```
M [Row][p*TILE_WIDTH + tx]
```

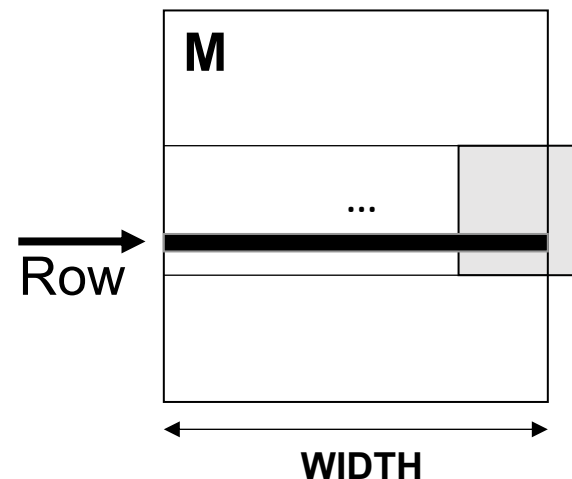
1D indexing for accessing Tile **p**:

```
M [Row*Width + p*TILE_WIDTH + tx]
```

Boundary condition

```
if(Row < Width) && (p*TILE_WIDTH+tx < Width)
```

- true: load M element
- else: use 0



CUDA Memories

Tiled Matrix Multiplication Kernel

Loading Input Tile **p** of **N**

- each thread loads an **M** element and an **N** element at
- the same relative position as its **P** element

```
int bx = blockIdx.x;  int by = blockIdx.y;  
int tx = threadIdx.x; int ty = threadIdx.y;  
int Row = by * blockDim.y + ty;  
int Col = bx * blockDim.x + tx;
```

2D indexing for accessing Tile **p**:

```
N [p*TILE*WIDTH + ty][Col]
```

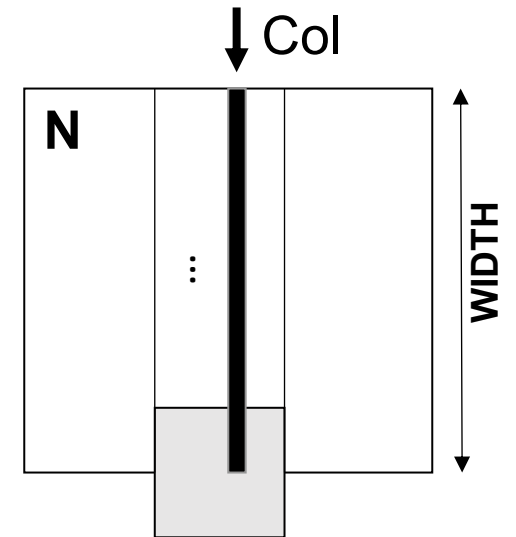
1D indexing for accessing Tile **p**:

```
N [(p*TILE_WIDTH+ty)*Width + Col]
```

Boundary condition

```
if (p*TILE_WIDTH+ty < Width) && (Col < Width)
```

- true: load **M** element
- else: use **0**



CUDA Memories

Tiled Matrix Multiplication Kernel

Loading Elements to Shared Memory

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width)
{
    __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
    __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x; int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

    int Row = by * blockDim.y + ty;
    int Col = bx * blockDim.x + tx;
    float Pvalue = 0;

    // Loop over the M and N tiles required to compute the P element
    for (int p = 0; p < n/TILE_WIDTH; ++p) {
        // Collaborative loading of M and N tiles into shared memory
        ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
        ds_N[ty][tx] = N[(t*TILE_WIDTH+ty)*Width + Col];
        __syncthreads();

        for (int i = 0; i < TILE_WIDTH; ++i)
            Pvalue += ds_M[ty][i] * ds_N[i][tx];

        __syncthreads();
    }
    P[Row*Width+Col] = Pvalue;
}
```

CUDA Memories

Tiled Matrix Multiplication Kernel

Loading Elements to Shared Memory

- with boundary check

```
// Loop over the M and N tiles required to compute the P element
for (int p = 0; p < n/TILE_WIDTH; ++p) {
    // Collaborative loading of M and N tiles into shared memory
    ds_M[ty][tx] = M[Row * Width + p * TILE_WIDTH + tx];
    ds_N[ty][tx] = N[(p * TILE_WIDTH + ty) * Width + Col];
    __syncthreads();
}
```

```
// Loop over the M and N tiles required to compute the P element
for (int p = 0; p < (Width-1)/TILE_WIDTH + 1; ++p) {
    // Collaborative loading of M and N tiles into shared memory
    if(Row < Width && p * TILE_WIDTH+tx < Width) {
        ds_M[ty][tx] = M[Row * Width + p * TILE_WIDTH + tx];
    } else {
        ds_M[ty][tx] = 0.0;
    }
    if (p*TILE_WIDTH+ty < Width && Col < Width) {
        ds_N[ty][tx] = N[(p * TILE_WIDTH + ty) * Width + Col];
    } else {
        ds_N[ty][tx] = 0.0;
    }
    __syncthreads();
}
```

CUDA Memories

Tiled Matrix Multiplication Kernel

Inner Product

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width)
{
    __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
    __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x; int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

    int Row = by * blockDim.y + ty;
    int Col = bx * blockDim.x + tx;
    float Pvalue = 0;

    // Loop over the M and N tiles required to compute the P element
    for (int p = 0; p < n/TILE_WIDTH; ++p) {
        // Collaborative loading of M and N tiles into shared memory
        ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
        ds_N[ty][tx] = N[(t*TILE_WIDTH+ty)*Width + Col];
        __syncthreads();

        for (int i = 0; i < TILE_WIDTH; ++i)
            Pvalue += ds_M[ty][i] * ds_N[i][tx];

        __syncthreads();
    }
    P[Row*Width+Col] = Pvalue;
}
```

CUDA Memories

Tiled Matrix Multiplication Kernel

Inner Product – Before and After

```
for (int i = 0; i < TILE_WIDTH; ++i) {
    Pvalue += ds_M[ty][i] * ds_N[i][tx];
}
__syncthreads();
}
P[Row*Width+Col] = Pvalue;
} /* end of kernel */
```

```
if(Row < Width && Col < Width) {
    for (int i = 0; i < TILE_WIDTH; ++i) {
        Pvalue += ds_M[ty][i] * ds_N[i][tx];
    }
    __syncthreads();
}
if (Row < Width && Col < Width)
    P[Row*Width + Col] = Pvalue;
} /* end of kernel */
```

Hands-On Tiled Matrix Multiplication

Univerza v Ljubljani



TECHNISCHE
UNIVERSITÄT
WIEN



VSB TECHNICAL
UNIVERSITY
OF OSTRAVA

IT4INNOVATIONS
NATIONAL SUPERCOMPUTING
CENTER



Co-funded by the
Erasmus+ Programme
of the European Union

This project has been funded with support from the European Commission.

This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

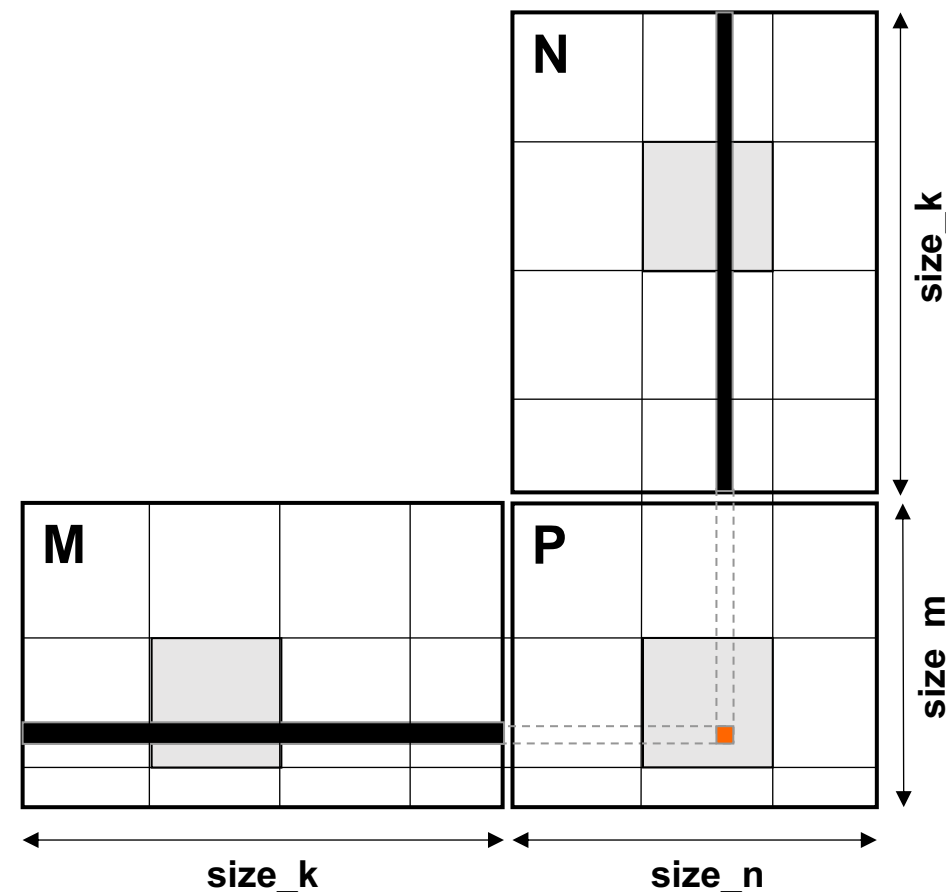
Hands-On Tiled Matrix Multiplication

- `06_matrix_multiplication/<lang>/Task/matrix_multiplication.<ext>`
- Matrix multiplication of two non-square matrices
- Finish the TODO tasks in kernels
 - Naïve implementation
 - Tiled implementation
- Compare the execution times

Correct output:

```
Matrix multiplication naive seems OK  
Matrix multiplication tiled seems OK
```

```
Time multiplication naive:   xxxx.xxx ms  
Time multiplication tiled:   yyyy.yyy ms  
Speedup is   z.zz
```



Parallel Computation Patterns: Stencil

Univerza v Ljubljani



TECHNISCHE
UNIVERSITÄT
WIEN



VSB TECHNICAL
UNIVERSITY
OF OSTRAVA

IT4INNOVATIONS
NATIONAL SUPERCOMPUTING
CENTER



Co-funded by the
Erasmus+ Programme
of the European Union

This project has been funded with support from the European Commission.

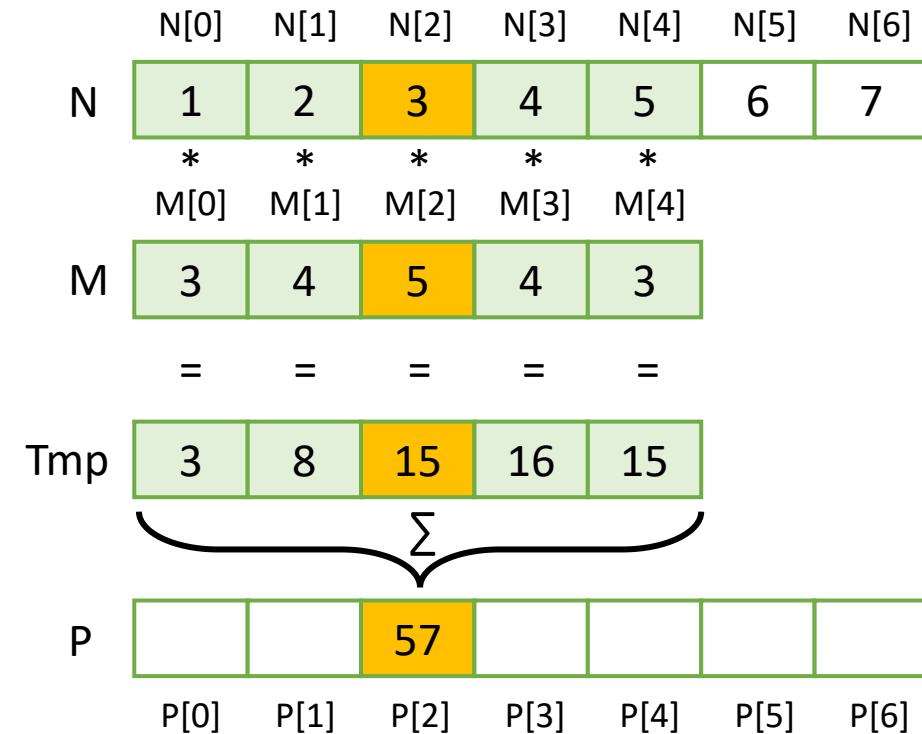
This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Parallel Computation Patterns

Stencil

Convolution

- basic example for stencil computation pattern
- an array operation where each output data element is a weighted sum of a collection of neighboring input elements
- the weights used in the weighted sum calculation are defined by an input mask array, commonly referred to as the convolution kernel
 - we will refer to these mask arrays as convolution masks to avoid confusion.
 - the value pattern of the mask array elements defines the type of filtering done
- Image Blur example is a special case where all mask elements are of the same value and hard coded into the source code.



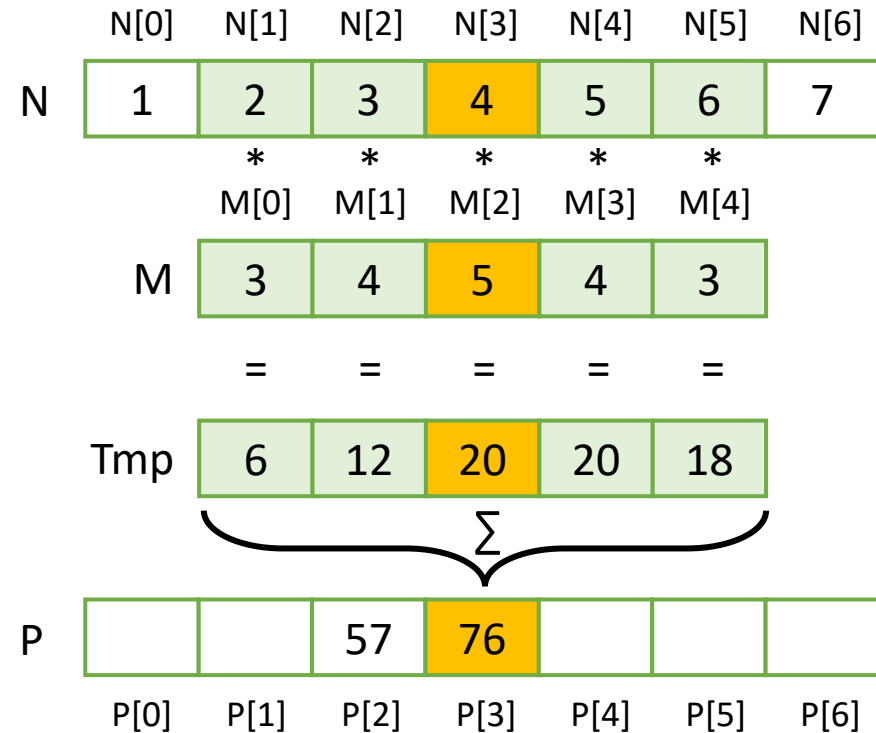
$$P[2] = N[0]*M[0] + N[1]*M[1] + N[2]*M[2] + N[3]*M[3] + N[4]*M[4]$$

Parallel Computation Patterns

Stencil

Convolution

- basic example for stencil computation pattern
- an array operation where each output data element is a weighted sum of a collection of neighboring input elements
- the weights used in the weighted sum calculation are defined by an input mask array, commonly referred to as the convolution kernel
 - we will refer to these mask arrays as convolution masks to avoid confusion.
 - the value pattern of the mask array elements defines the type of filtering done
- Image Blur example is a special case where all mask elements are of the same value and hard coded into the source code.



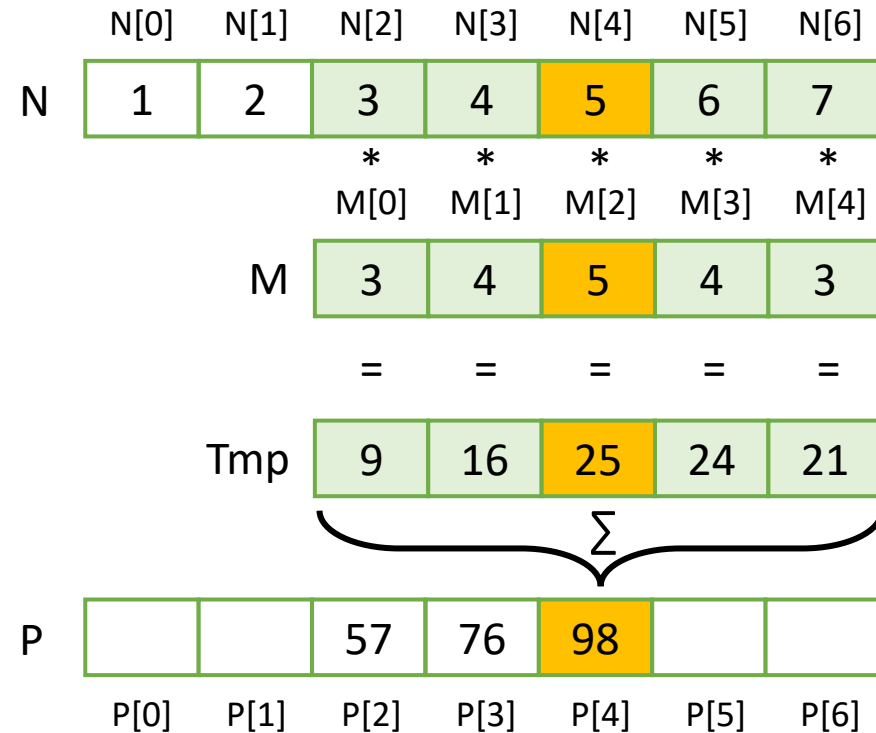
$$P[3] = N[1]*M[0] + N[2]*M[1] + N[3]*M[2] + N[4]*M[3] + N[5]*M[4]$$

Parallel Computation Patterns

Stencil

Convolution

- basic example for stencil computation pattern
- an array operation where each output data element is a weighted sum of a collection of neighboring input elements
- the weights used in the weighted sum calculation are defined by an input mask array, commonly referred to as the convolution kernel
 - we will refer to these mask arrays as convolution masks to avoid confusion.
 - the value pattern of the mask array elements defines the type of filtering done
- Image Blur example is a special case where all mask elements are of the same value and hard coded into the source code.



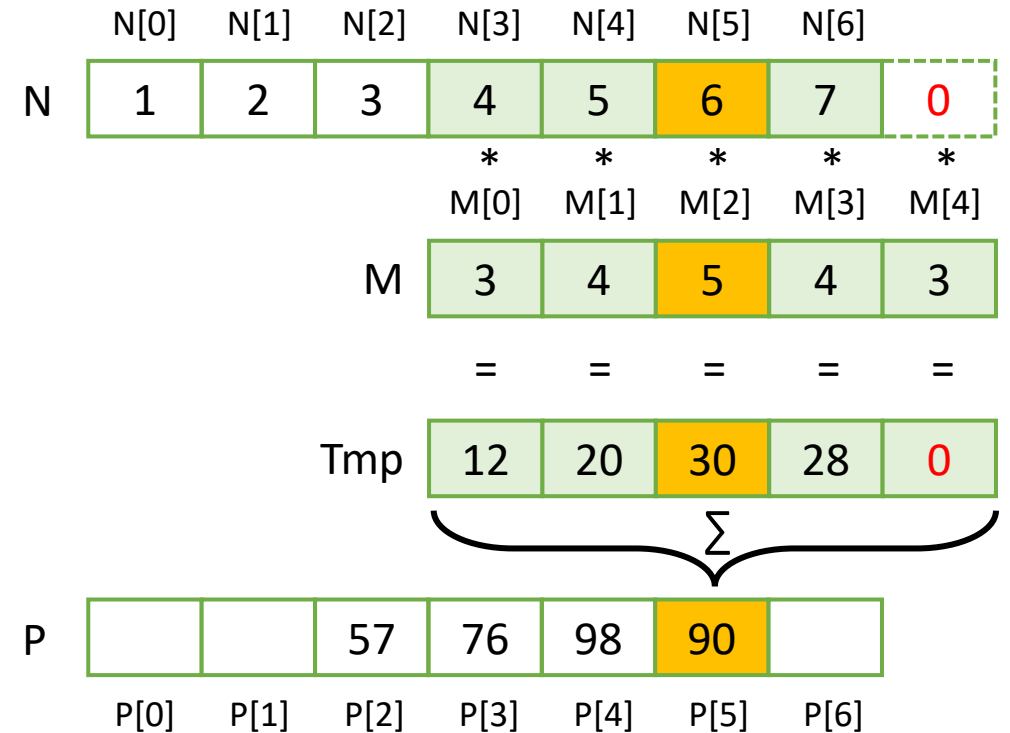
$$P[4] = N[2]*M[0] + N[3]*M[1] + N[4]*M[2] + N[5]*M[3] + N[6]*M[4]$$

Parallel Computation Patterns

Stencil

Boundary condition

- calculation of output elements near the boundaries (beginning and end) of the array need to deal with “ghost” elements
 - different policies (0, replicates of boundary values, etc.)



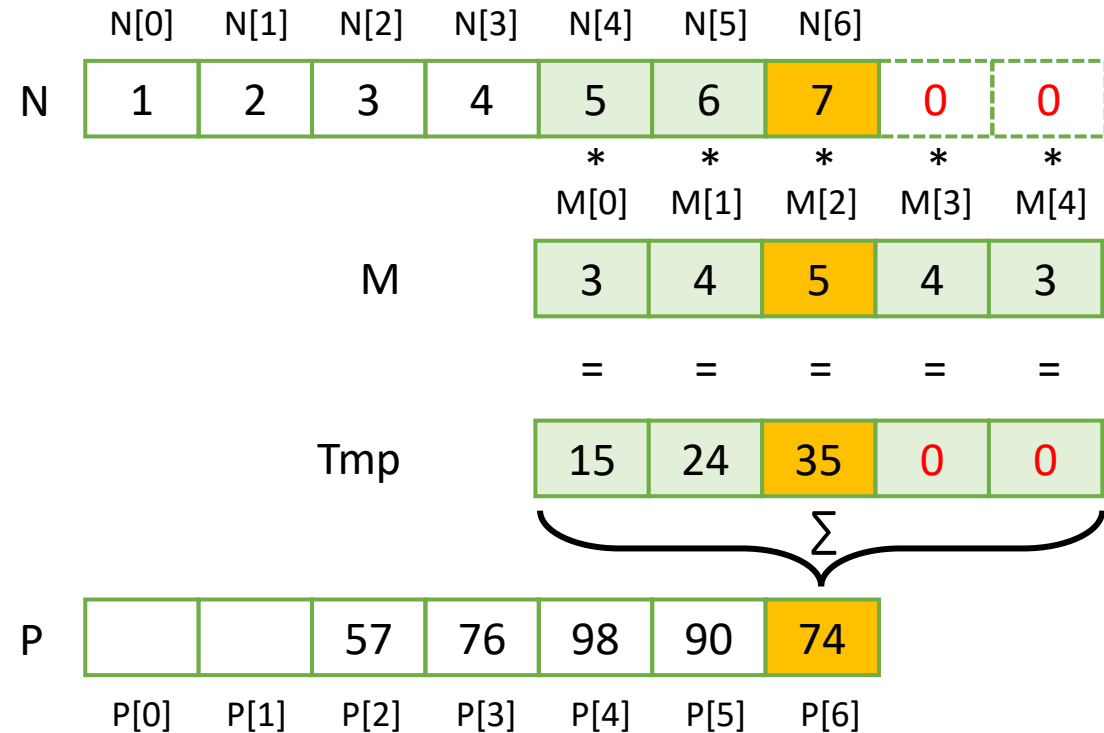
$$P[5] = N[3]*M[0] + N[4]*M[1] + N[5]*M[2] + N[6]*M[3] + 0*M[4]$$

Parallel Computation Patterns

Stencil

Boundary condition

- calculation of output elements near the boundaries (beginning and end) of the array need to deal with “ghost” elements
 - different policies (0, replicates of boundary values, etc.)



$$P[3] = N[4]*M[0] + N[5]*M[1] + N[6]*M[2] + 0*M[3] + 0*M[4]$$

Boundary condition

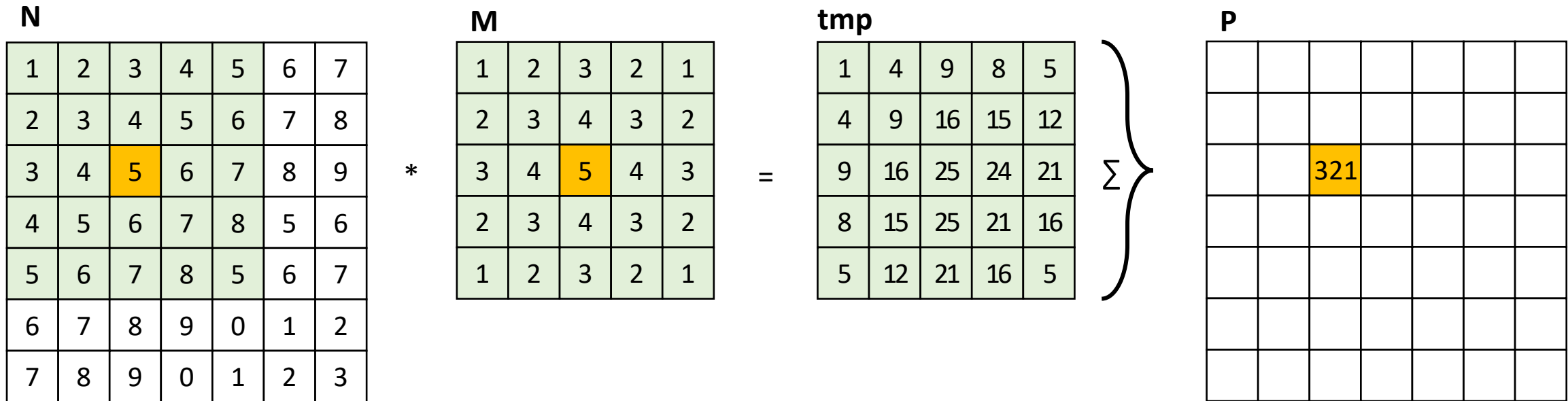
- calculation of output elements near the boundaries (beginning and end) of the array need to deal with “ghost” elements
 - different policies (0, replicates of boundary values, etc.)

```
__global__ void convolution_1D_basic_kernel(  
    float *N, float *M, float *P,  
    int Mask_Width, int Width)  
{  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
  
    float Pvalue = 0;  
    int N_start_point = i - (Mask_Width/2);  
  
    for (int j = 0; j < Mask_Width; j++) {  
        if (N_start_point + j >= 0 && N_start_point + j < Width)  
        {  
            Pvalue += N[N_start_point + j] * M[j];  
        }  
    }  
  
    P[i] = Pvalue;  
}
```

Parallel Computation Patterns

Stencil

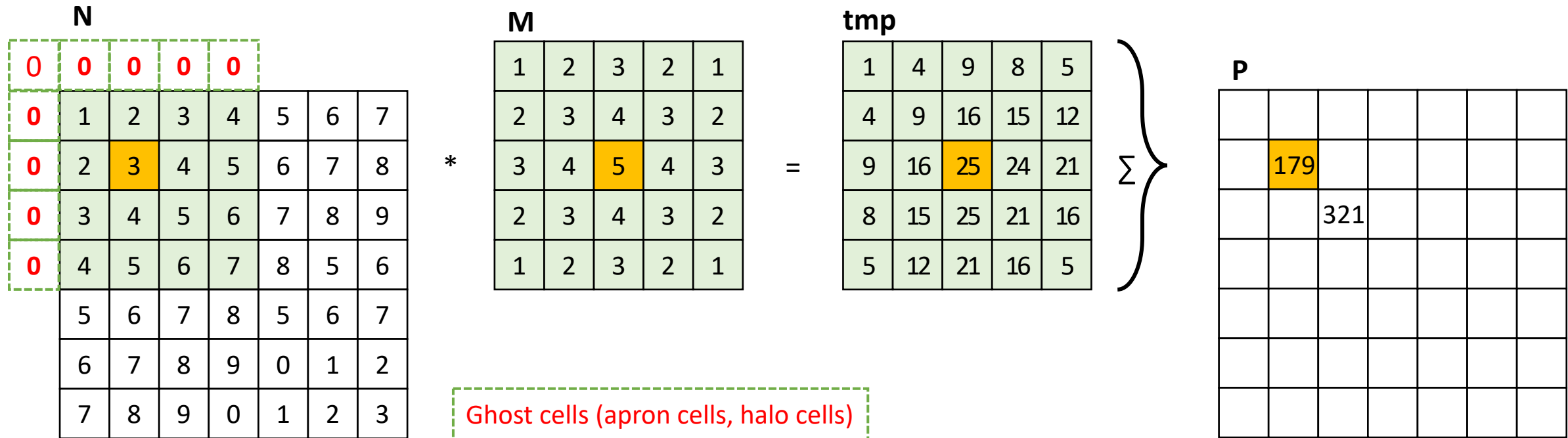
2D Convolution



Parallel Computation Patterns

Stencil

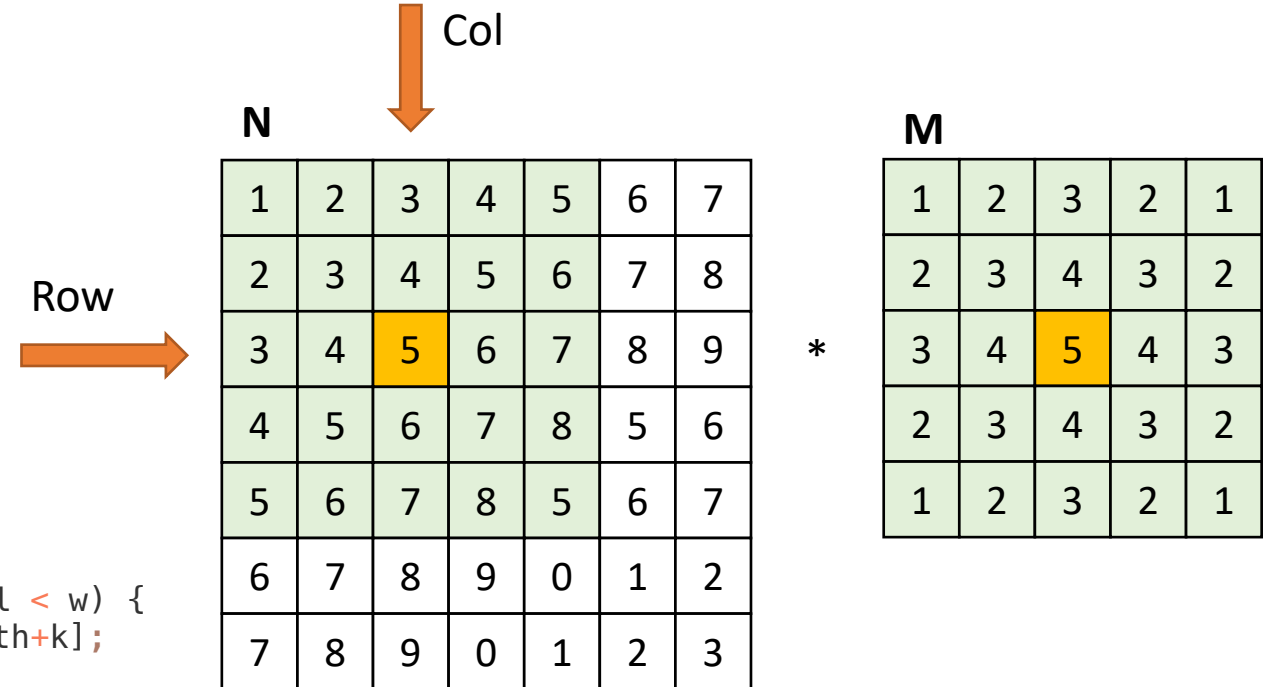
2D Convolution – boundaries with ghost cells



Parallel Computation Patterns

Stencil

```
__global__  
void convolution_2D_basic_kernel(unsigned char * in, unsigned char * mask, unsigned char * out, int maskWidth, int w, int h) {  
    int Col = blockIdx.x * blockDim.x + threadIdx.x;  
    int Row = blockIdx.y * blockDim.y + threadIdx.y;  
  
    if (Col < w && Row < h) {  
        int pixVal = 0;  
  
        N_start_col = Col - (maskwidth/2);  
        N_start_row = Row - (maskwidth/2);  
  
        // Get the of the surrounding box  
        for(int j = 0; j < maskWidth; ++j) {  
            for(int k = 0; k < maskWidth; ++k) {  
  
                int curRow = N_start_row + j;  
                int curCol = N_start_col + k;  
                // Verify we have a valid image pixel  
                if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {  
                    pixVal += in[curRow * w + curCol] * mask[j*maskWidth+k];  
                }  
            }  
        }  
  
        // Write our new pixel value out  
        out[Row * w + Col] = (unsigned char)(pixVal);  
    }  
}
```



Parallel Computation Patterns

Stencil

```
__global__  
void convolution_2D_basic_kernel(unsigned char * in, unsigned char * mask, unsigned char * out, int maskWidth, int w, int h) {  
    int Col = blockIdx.x * blockDim.x + threadIdx.x;  
    int Row = blockIdx.y * blockDim.y + threadIdx.y;
```

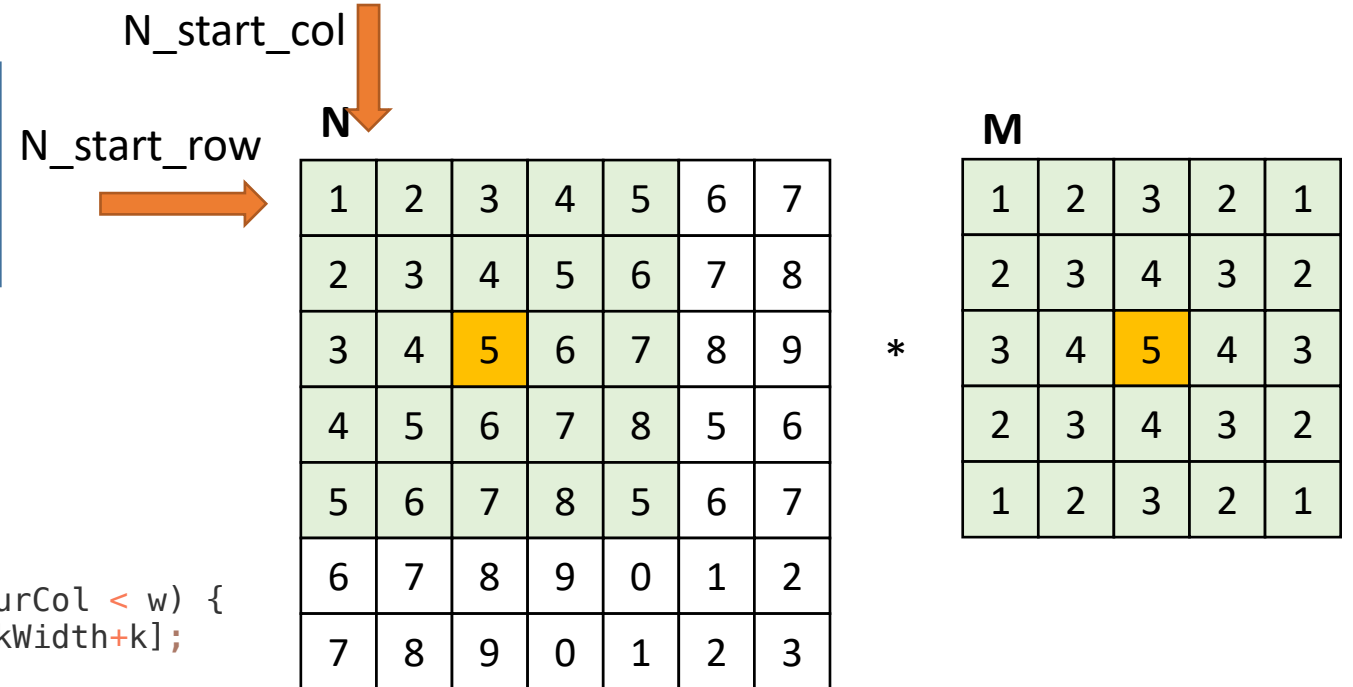
```
    if (Col < w && Row < h) {  
        int pixVal = 0;
```

```
        N_start_col = Col - (maskwidth/2);  
        N_start_row = Row - (maskwidth/2);
```

```
        // Get the of the surrounding box  
        for(int j = 0; j < maskWidth; ++j) {  
            for(int k = 0; k < maskWidth; ++k) {
```

```
                int curRow = N_start_row + j;  
                int curCol = N_start_col + k;  
                // Verify we have a valid image pixel  
                if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {  
                    pixVal += in[curRow * w + curCol] * mask[j*maskWidth+k];  
                }  
            }  
        }
```

```
        // Write our new pixel value out  
        out[Row * w + Col] = (unsigned char)(pixVal);  
    }
```



Parallel Computation Patterns

Stencil

```
__global__
void convolution_2D_basic_kernel(unsigned char * in, unsigned char * mask, unsigned char * out, int maskWidth, int w, int h) {
    int Col = blockIdx.x * blockDim.x + threadIdx.x;
    int Row = blockIdx.y * blockDim.y + threadIdx.y;
```

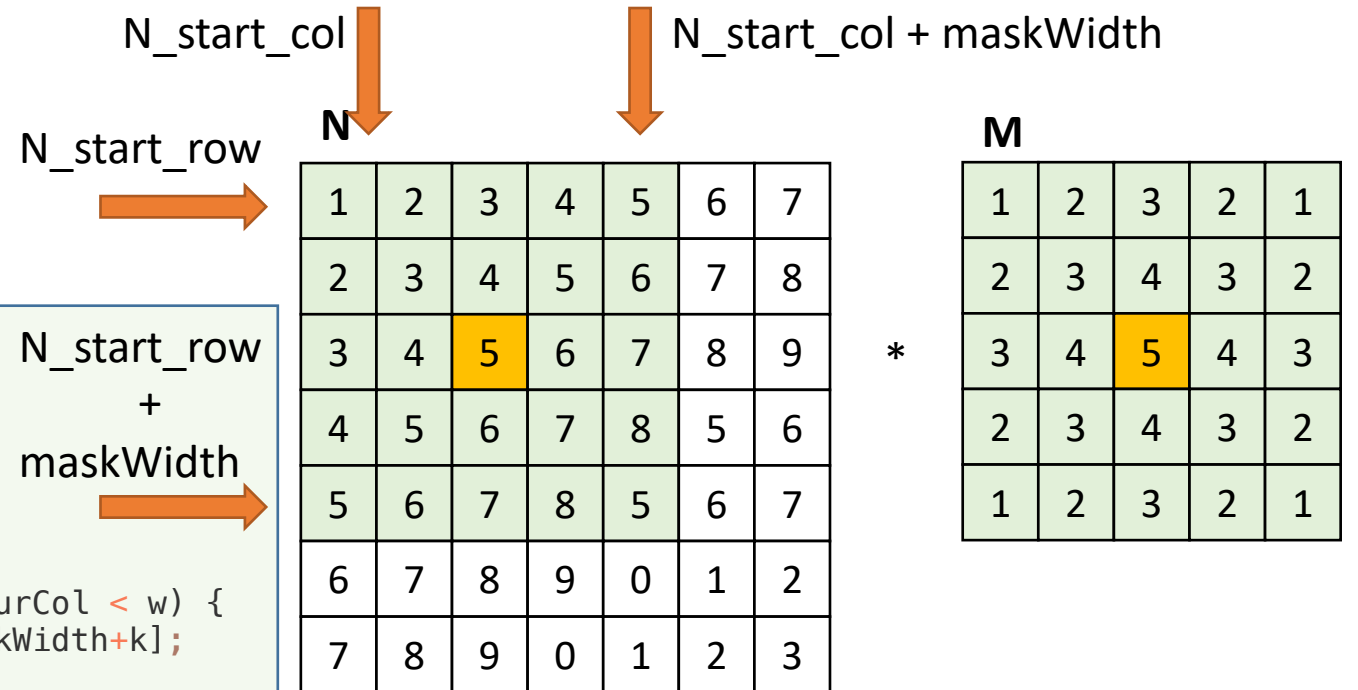
```
    if (Col < w && Row < h) {
        int pixVal = 0;
```

```
        N_start_col = Col - (maskwidth/2);
        N_start_row = Row - (maskwidth/2);
```

```
        // Get the of the surrounding box
        for(int j = 0; j < maskWidth; ++j) {
            for(int k = 0; k < maskWidth; ++k) {
```

```
                int curRow = N_start_row + j;
                int curCol = N_start_col + k;
                // Verify we have a valid image pixel
                if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
                    pixVal += in[curRow * w + curCol] * mask[j*maskWidth+k];
                }
            }
        }
```

```
        // Write our new pixel value out
        out[Row * w + Col] = (unsigned char)(pixVal);
    }
```



Parallel Computation Patterns

Stencil

Using constant memory and caching for Mask

- mask is used by all threads but not modified in the convolution kernel
 - all threads in a warp access the same locations at each point in time
- CUDA devices provide constant memory whose contents are aggressively cached
 - cached values are broadcast to all threads in a warp
 - effectively magnifies memory bandwidth without consuming shared memory
- use of **const __restrict__** qualifiers for the mask parameter informs the compiler that it is eligible for constant caching, for example:

```
__global__ void convolution_2D_kernel(  
    float *P,  
    float *N,  
    int height, int width,  
    const float __restrict__ *M)  
{ ... }
```

More info: <https://developer.nvidia.com/blog/cuda-pro-tip-optimize-pointer-aliasing/>

Mask

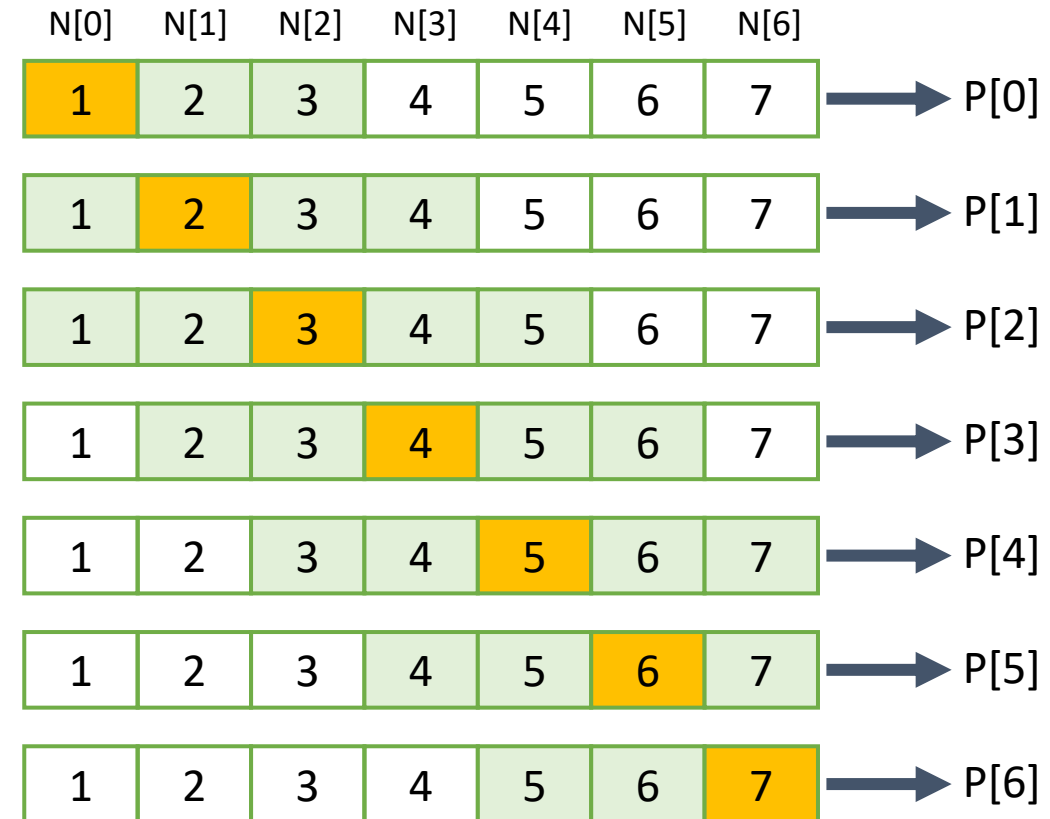
1	2	3	2	1
2	3	4	3	2
3	4	5	4	3
2	3	4	3	2
1	2	3	2	1

Parallel Computation Patterns

Stencil

Tiling Opportunity Convolution

- calculation of adjacent output elements involve shared input elements
 - e.g., N[2] is used in calculation of P[0], P[1], P[2]. P[3] and P[5] assuming a 1D convolution Mask_Width of width 5
- we can load all the input elements required by all threads in a block into the shared memory to reduce global memory accesses



Tile considerations

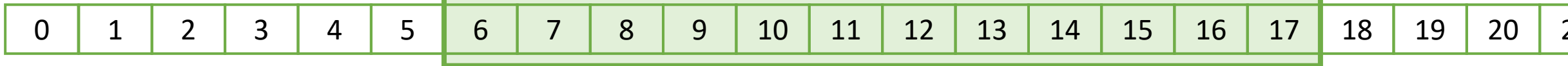
M

Mask_Width / 2 (integer arithmetics)

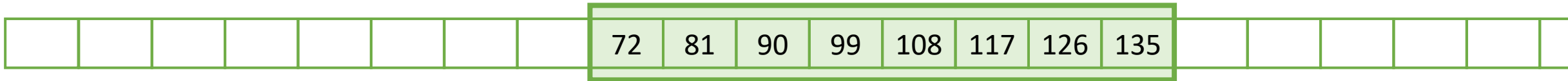


Input tile size = $T + \text{Mask_Width} - 1$

N



P

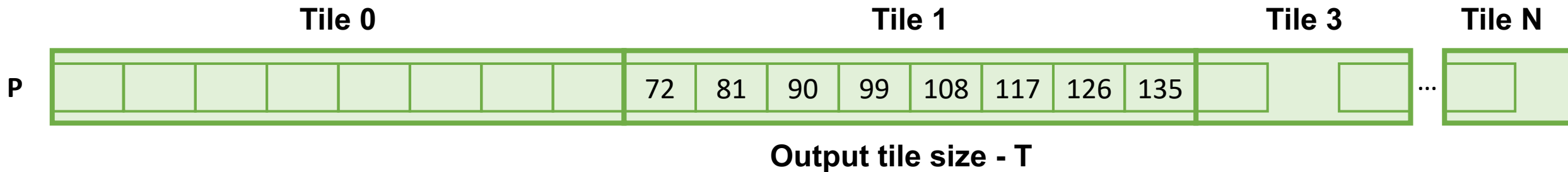


Output tile size - T

Assume that we want to have each block to calculate T output elements

- $T + \text{Mask_Width} - 1$ input elements are needed to calculate T output elements
- $T + \text{Mask_Width} - 1$ is usually not a multiple of T, except for small T values
- T is usually significantly larger than Mask_Width

Output tile definition



- each thread block calculates one output tile
- each output tile width is T
 - T is 4 in this example

Parallel Computation Patterns

Stencil

Input Tile in Shared Memory

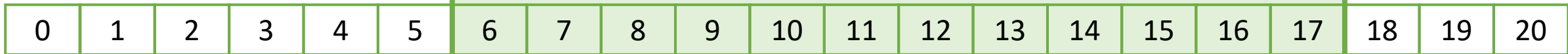
M

Mask_Width / 2 (integer arithmetics)

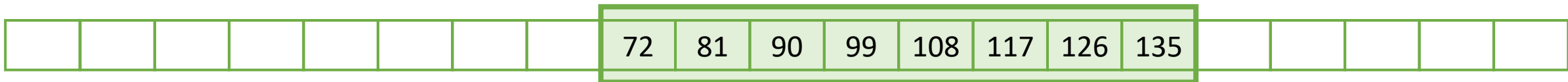


Input tile size = $T + \text{Mask_Width} - 1$

N



P



Output tile size - T

Tile in a shared memory: Ns



- each input tile has all values needed to calculate the corresponding output tile.

Parallel Computation Patterns

Stencil

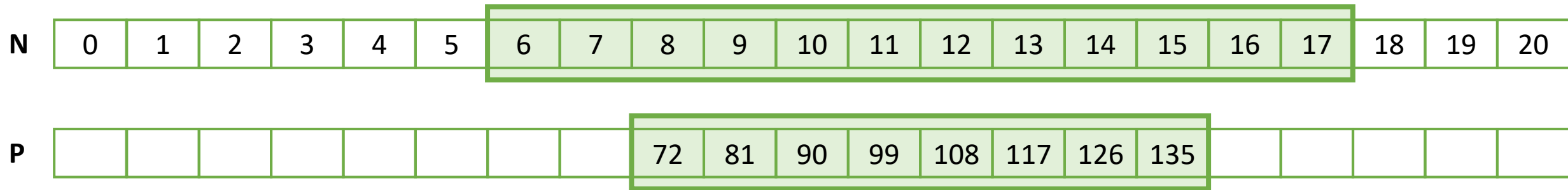
Design 1: The size of each thread block matches the size of an output tile

- All threads participate in calculating output elements
- blockDim.x would be 8 in our example
- Some threads need to load more than one input element into the shared memory

Design 2: The size of each thread block matches the size of an input tile

- Some threads will not participate in calculating output elements
- blockDim.x would be 12 in our example
- Each thread loads one input element into the shared memory

Input tile size = $T + \text{Mask_Width} - 1$



Output tile size - T

Tile in a shared memory: N_s



- each input tile has all values needed to calculate the corresponding output tile.

Parallel Computation Patterns

Stencil

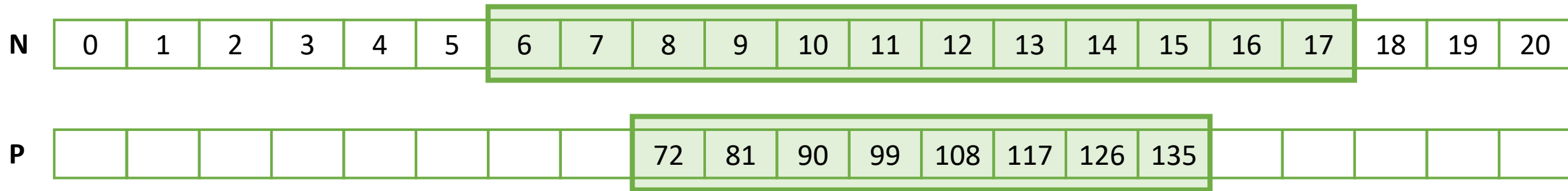
Design 1: The size of each thread block matches the size of an output tile

- All threads participate in calculating output elements
- blockDim.x would be 8 in our example
- Some threads need to load more than one input element into the shared memory

Design 2: The size of each thread block matches the size of an input tile

- Some threads will not participate in calculating output elements
- blockDim.x would be 12 in our example
- Each thread loads one input element into the shared memory

Input tile size = $T + \text{Mask_Width} - 1$



Output tile size - T

Tile in a shared memory: N_s

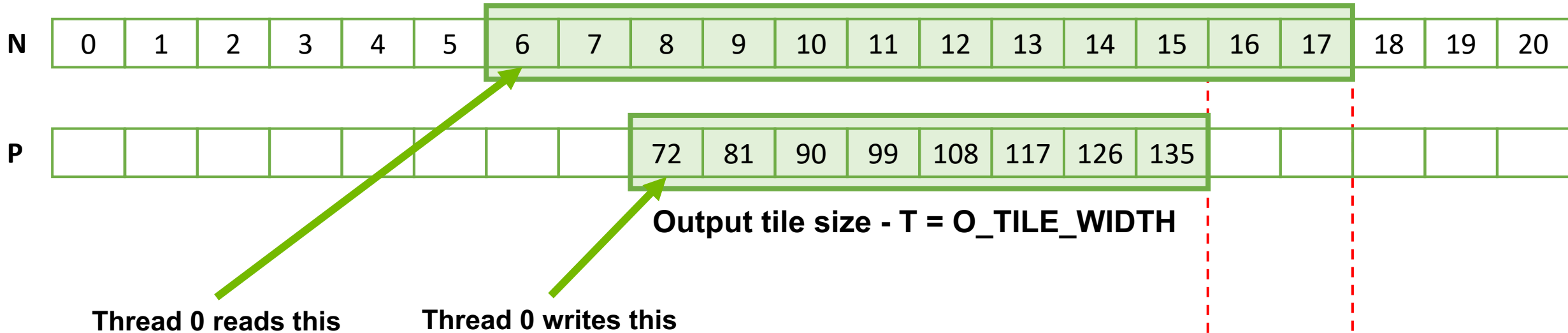


- each input tile has all values needed to calculate the corresponding output tile.

Parallel Computation Patterns

Stencil

Thread to Input and Output Data Mapping

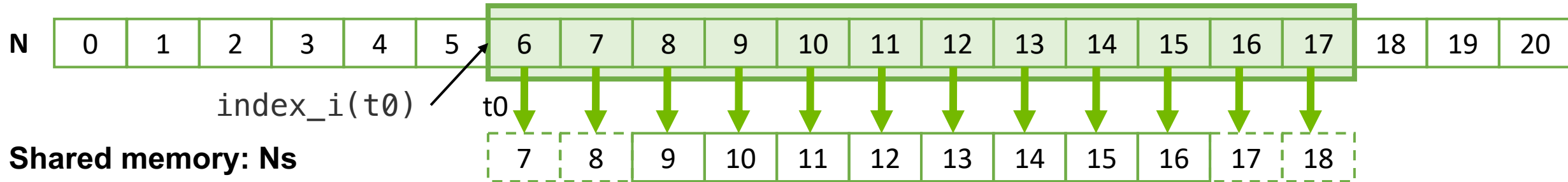


For each thread:

- $index_i = index_o - n$,
- where:
 - n is $Mask_Width/2$
 - n is 2 in this example



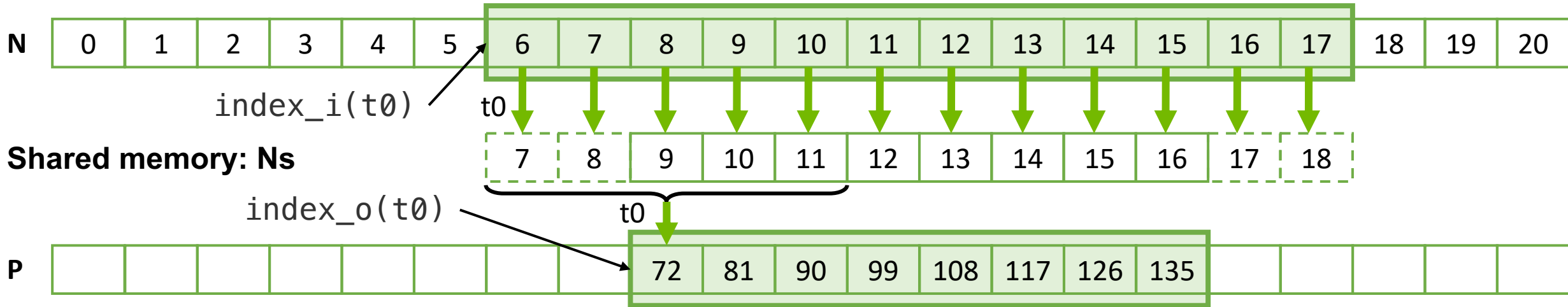
Thread to Input and Output Data Mapping



- all threads participate in loading input tiles

```
float output = 0.0f;  
  
if((index_i >= 0) && (index_i < Width)) {  
    Ns[tx] = N[index_i];  
}  
else{  
    Ns[tx] = 0.0f;  
}
```

Thread to Input and Output Data Mapping



- some threads do not participate in calculating output

```

index_o = blockIdx.x * 0_TILE_WIDTH + threadIdx.x;
index_i = index_o - Mask_Width/2;
if (threadIdx.x < 0_TILE_WIDTH){
    output = 0.0f;
    for(j = 0; j < Mask_Width; j++) {
        output += M[j] * Ns[j+threadIdx.x];
    }
    P[index_o] = output;
}
    
```

Setting Block Size

```
#define O_TILE_WIDTH 1020
#define BLOCK_WIDTH (O_TILE_WIDTH +
                    (Mask_Width-1))

dim3 dimBlock(BLOCK_WIDTH,1, 1);

dim3 dimGrid((Width-1)/O_TILE_WIDTH+1, 1, 1)
```

Kernel code (partial)

```
...
index_o = blockIdx.x * O_TILE_WIDTH +
          threadIdx.x;
index_i = index_o - n - Mask_Width/2;

if((index_i >= 0) && (index_i < Width)) {
    Ns[tx] = N[index_i];
}
else{
    Ns[tx] = 0.0f;
}

if (threadIdx.x < O_TILE_WIDTH){
    float output = 0.0f;
    for(j = 0; j < Mask_Width; j++) {
        output += M[j] * Ns[j+threadIdx.x];
    }
    P[index_o] = output;
} ...
```

The Efficiency of Tiling

- Significant reduction of Global Memory bandwidth

1D Convolution

- **The reduction ratio – how many times tiling reduces accesses to Global Memory**
- $\text{MASK_WIDTH} * (\text{O_TILE_WIDTH}) / (\text{O_TILE_WIDTH} + \text{MASK_WIDTH} - 1)$

O_TILE_WIDTH	16	32	64	128	256
MASK_WIDTH= 5	4.0	4.4	4.7	4.9	4.9
MASK_WIDTH = 9	6.0	7.2	8.0	8.5	8.7

2D Convolution

- The reduction ratio is:
 - $\text{O_TILE_WIDTH}^2 * \text{MASK_WIDTH}^2 / (\text{O_TILE_WIDTH} + \text{MASK_WIDTH} - 1)^2$

O_TILE_WIDTH	8	16	32	64
MASK_WIDTH = 5	11.1	16	19.7	22.1
MASK_WIDTH = 9	20.3	36	51.8	64

Tile size has significant effect on of the memory bandwidth reduction ratio.

This often argues for larger shared memory size.

Parallel Computation Patterns: Reduction

Univerza v Ljubljani



TECHNISCHE
UNIVERSITÄT
WIEN



VSB TECHNICAL
UNIVERSITY
OF OSTRAVA

IT4INNOVATIONS
NATIONAL SUPERCOMPUTING
CENTER



Co-funded by the
Erasmus+ Programme
of the European Union

This project has been funded with support from the European Commission.

This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Parallel Reduction

- a commonly used strategy for processing large input data sets
- there is no required order of processing elements in a data set (associative and commutative)

Approach:

- partition the data set into smaller chunks
- have each thread to process a chunk
- use a reduction tree to summarize the results from each chunk into the final answer
- **we will focus on the reduction tree step for now**

Reduction also enables other techniques

- reduction is also needed to clean up after some commonly used parallelizing transformations
- Example: privatization
 - multiple threads write into an output location
 - replicate the output location so that each thread has a private output location (privatization)
 - use a reduction tree to combine the values of private locations into the original output location

Parallel Computation Patterns

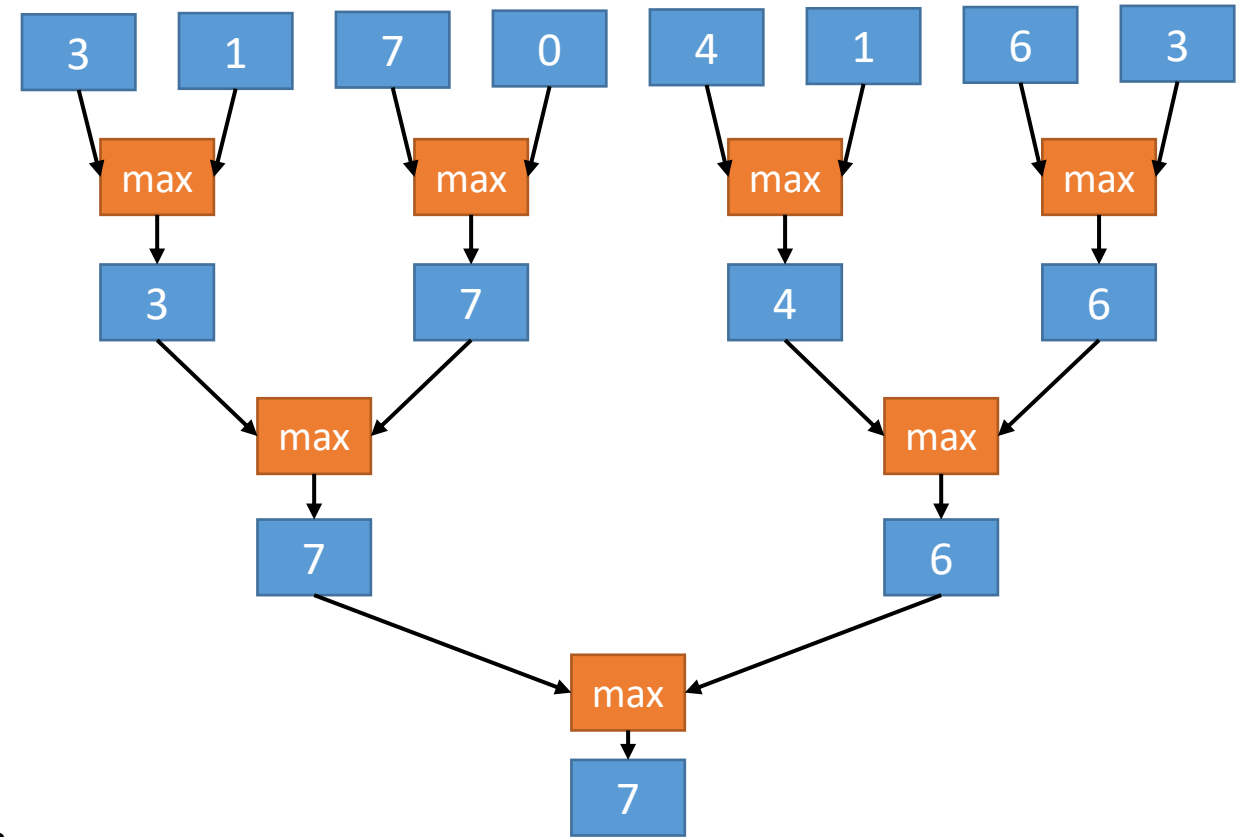
Reduction

Parallel Reduction

- summarize a set of input values into one value using a “reduction operation”
 - Max, Min, Sum, Product, ...
- can be used with a user defined reduction operation function if the operation:
 - is associative and commutative
 - has a well-defined identity value (e.g., 0 for sum)

An Efficient Sequential Reduction $O(N)$

- initialize the result as an identity value for the reduction operation
 - Smallest possible value for max reduction
 - Largest possible value for min reduction
 - 0 for sum reduction
 - 1 for product reduction
- iterate through the input and perform the reduction operation between the result value and the current input value
- **N reduction operations performed for N input values**
- each input value is only visited once – an $O(N)$ algorithm



A parallel reduction tree algorithm performs $N-1$ operations in $\log(N)$ steps

Parallel Computation Patterns

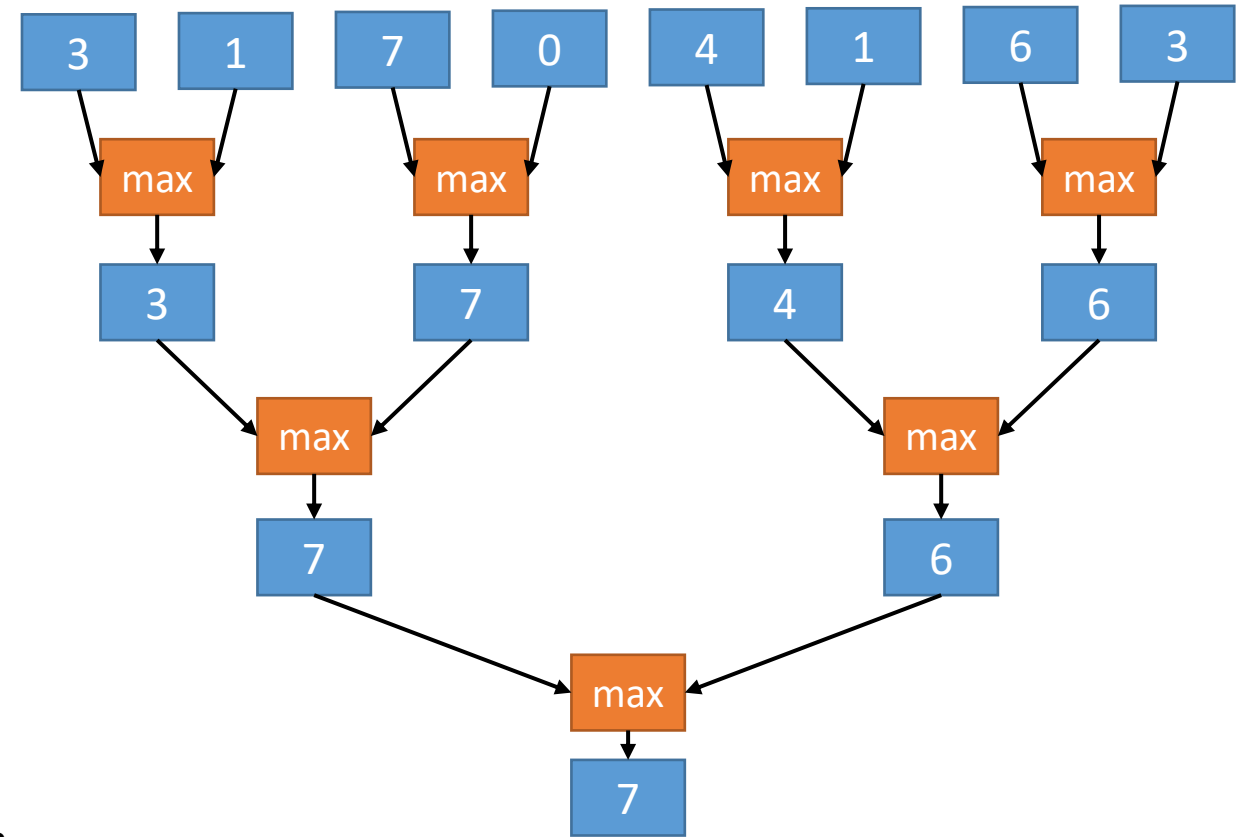
Reduction

Parallel Reduction

- summarize a set of input values into one value using a “reduction operation”
 - Max, Min, Sum, Product, ...
- can be used with a user defined reduction operation function if the operation:
 - is associative and commutative
 - has a well-defined identity value (e.g., 0 for sum)

An Efficient Sequential Reduction $O(N)$

- initialize the result as an identity value for the reduction operation
 - Smallest possible value for max reduction
 - Largest possible value for min reduction
 - 0 for sum reduction
 - 1 for product reduction
- iterate through the input and perform the reduction operation between the result value and the current input value
- **N reduction operations performed for N input values**
- each input value is only visited once – an $O(N)$ algorithm



A parallel reduction tree algorithm performs $N-1$ operations in $\log(N)$ steps

Parallel Computation Patterns

Reduction

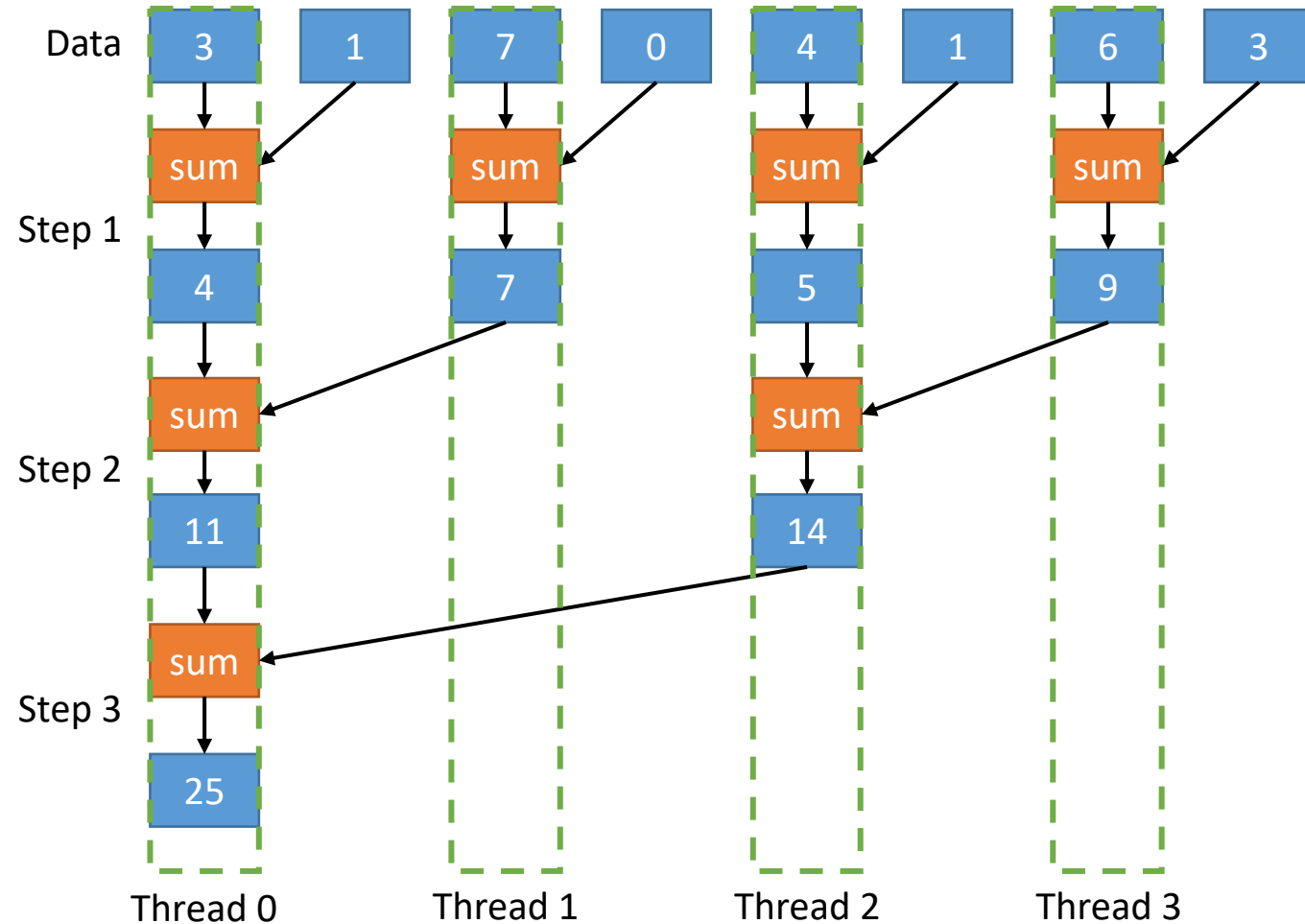
Parallel Sum Reduction on GPU

Parallel implementation

- each thread adds two values in each step
- recursively halve # of threads
- takes $\log(n)$ steps for n elements, requires $n/2$ threads

Assume an in-place reduction using shared memory

- the original vector is in device global memory
- the shared memory is used to hold a partial sum vector
 - initially, the partial sum vector is simply the original vector
- each step brings the partial sum vector closer to the sum
- the final sum will be in element 0 of the partial sum vector
- reduces global memory traffic due to reading and writing partial sum values
- thread block size limits n to be less than or equal to 2,048



Parallel Computation Patterns

Reduction

A Simple Thread Block Design

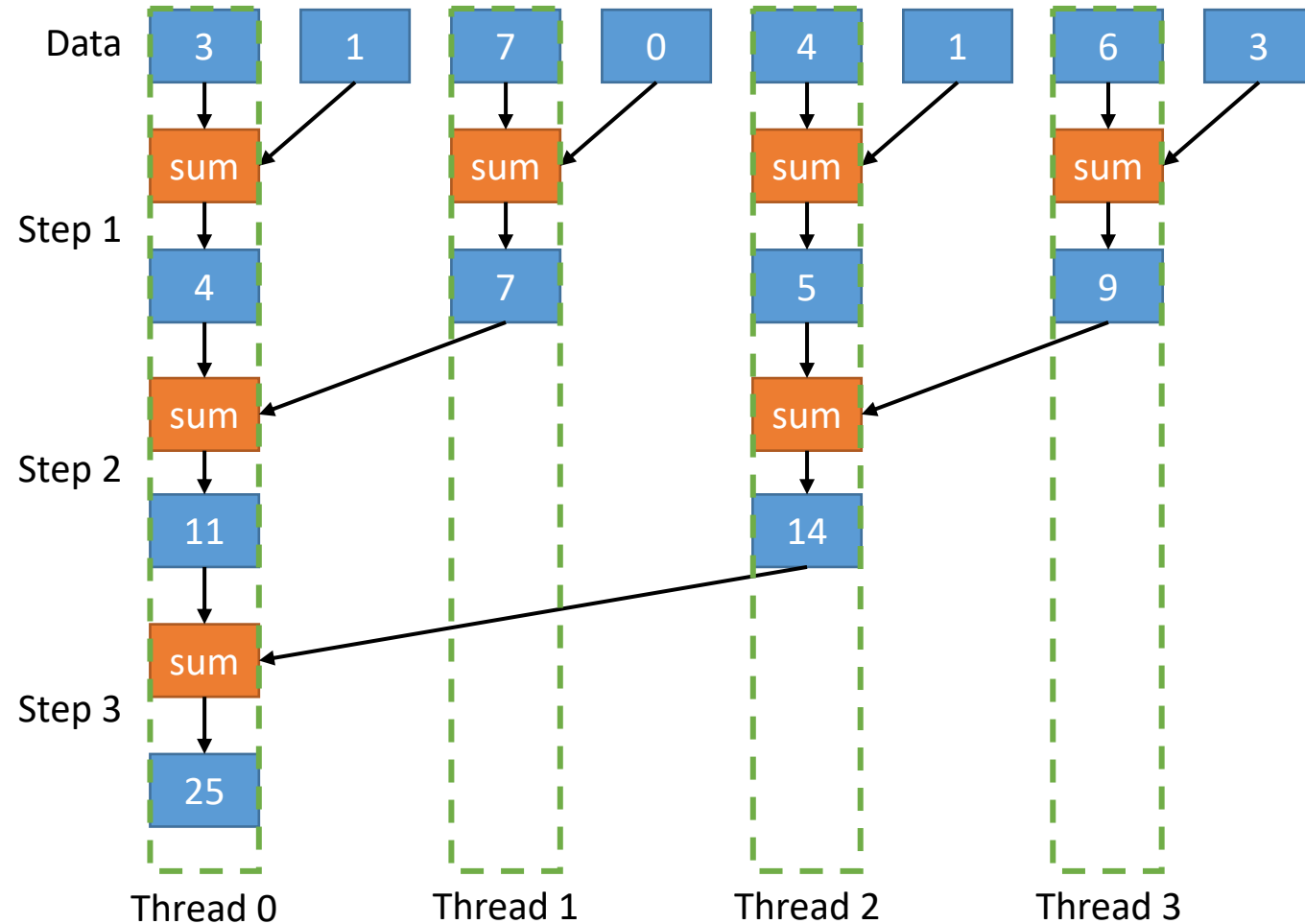
- each thread block takes $2 \times \text{BlockDim.x}$ input elements
- each thread loads 2 elements into shared memory

```
__shared__ float partialSum[2*BLOCK_SIZE];

unsigned int t = threadIdx.x;
unsigned int start = 2*blockIdx.x*blockDim.x;

partialSum[t] = input[start + t];
partialSum[blockDim+t] = input[start +
                             blockDim.x+t];

// The reduction step
for (unsigned int stride = 1;
     stride <= blockDim.x;
     stride *= 2)
{
    __syncthreads();
    if (t % stride == 0)
        partialSum[2*t] += partialSum[2*t+stride];
}
```



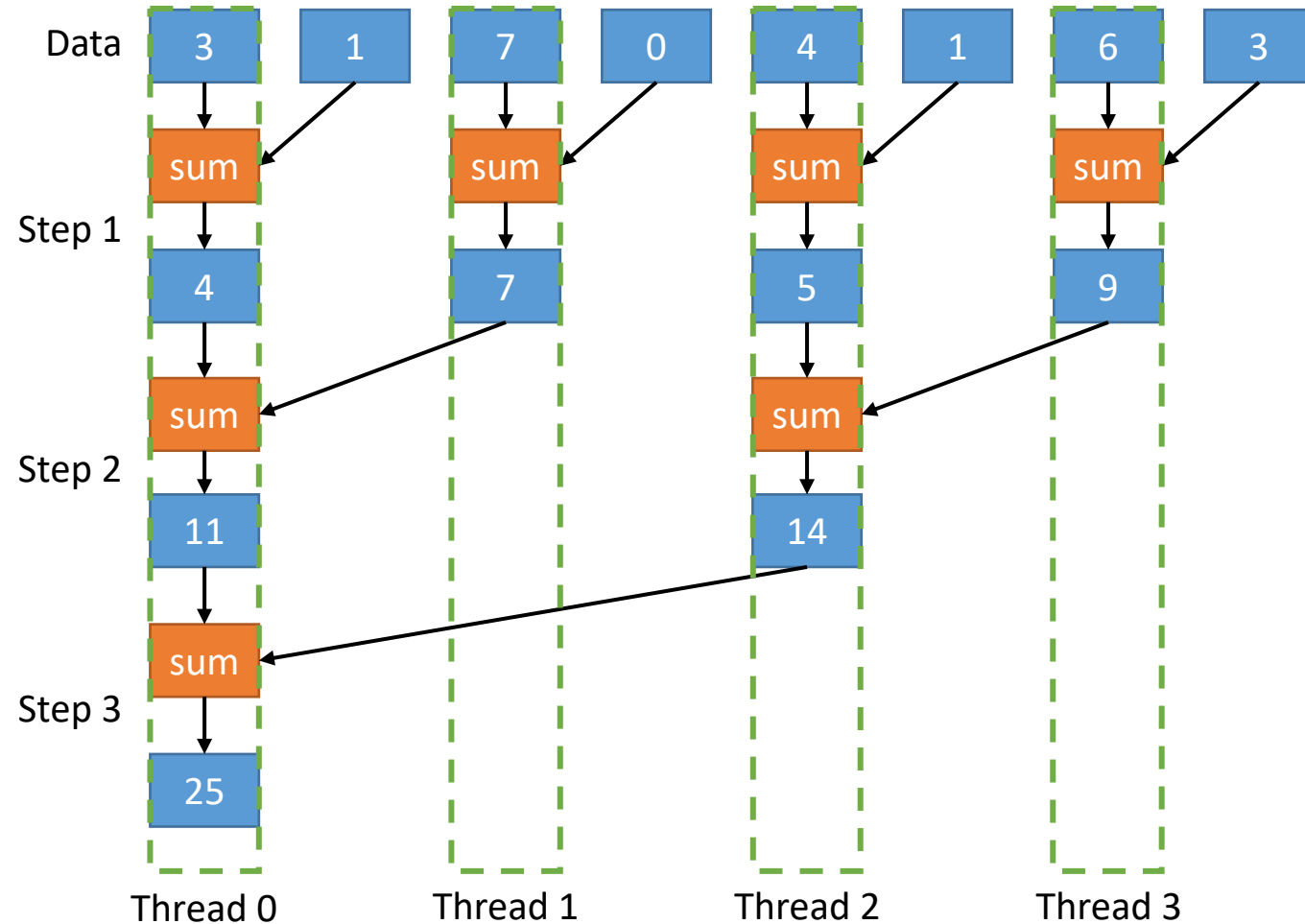
`__syncthreads()` is needed to ensure that all elements of each step of partial sums have been generated before the next step

Parallel Computation Patterns

Reduction

Global Picture

- at the end of the kernel, Thread 0 in each block writes the sum of the thread block in partialSum[0] into a vector indexed by the blockIdx.x
- there can be a large number of such sums if the original vector is very large
- the host code may iterate and launch another kernel
- if there are only a small number of sums, the host can simply transfer the data back and add them together
- alternatively, Thread 0 of each block could use atomic operations to accumulate into a global sum variable.



Parallel Computation Patterns

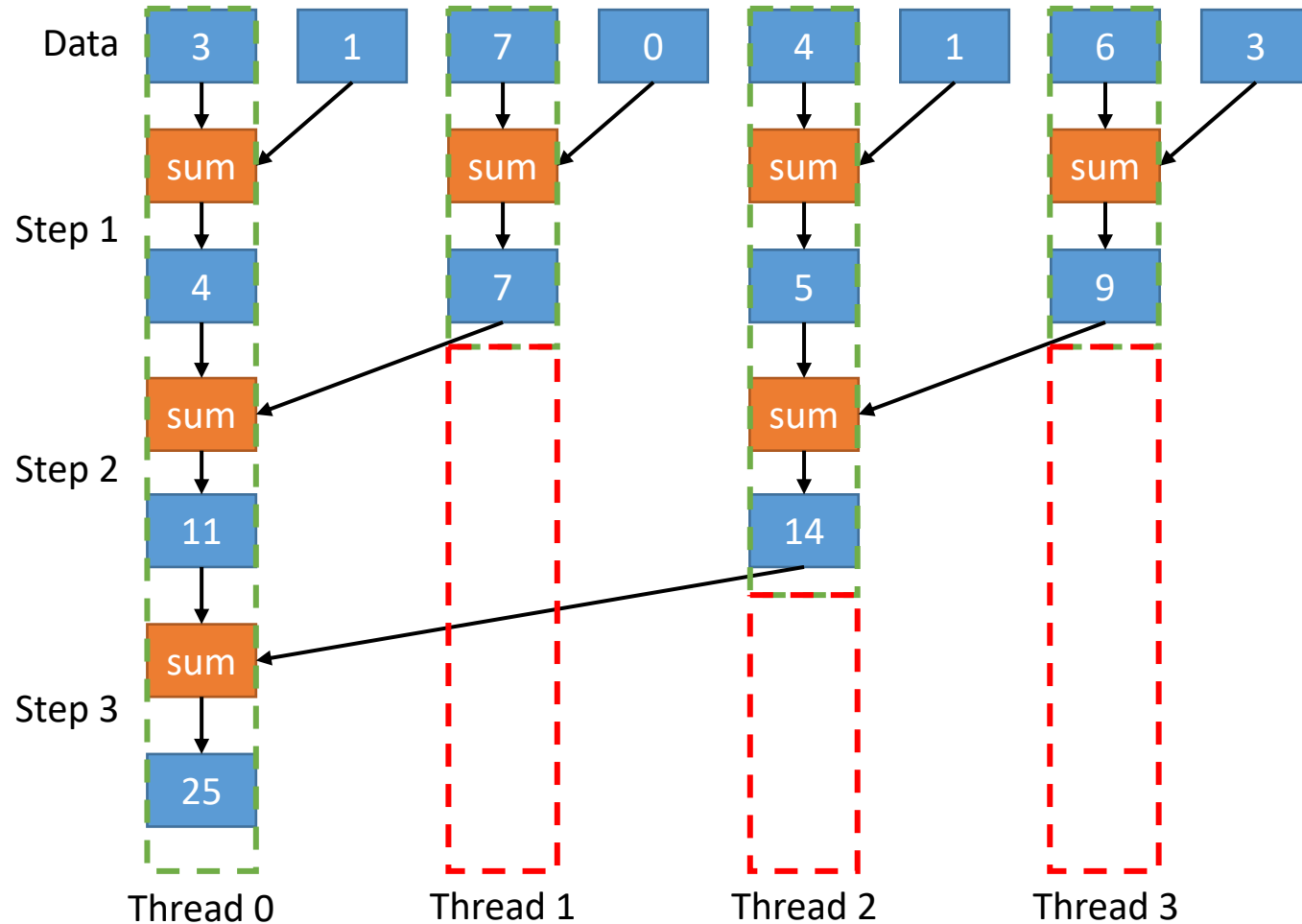
Reduction

Naive Thread to Data Mapping

- each thread is responsible for an even-index location of the partial sum vector (location of responsibility)
- after each step, half of the threads are no longer needed
- one of the inputs is always from the location of responsibility
- in each step, one of the inputs comes from an increasing distance away

Control Divergence of Naïve Kernel

- in each iteration, two control flow paths will be sequentially traversed for each warp
- threads that perform addition and threads that do not
- threads that do not perform addition still consume execution resources
- half or fewer of threads will be executing after the first step
- all odd-index threads are disabled after first step
- after the 5th step, entire warps in each block will fail the if test, poor resource utilization but no divergence
- this can go on for a while, up to 6 more steps (stride = 32, 64, 128, 256, 512, 1024), where each active warp only has one productive thread until all warps in a block retire



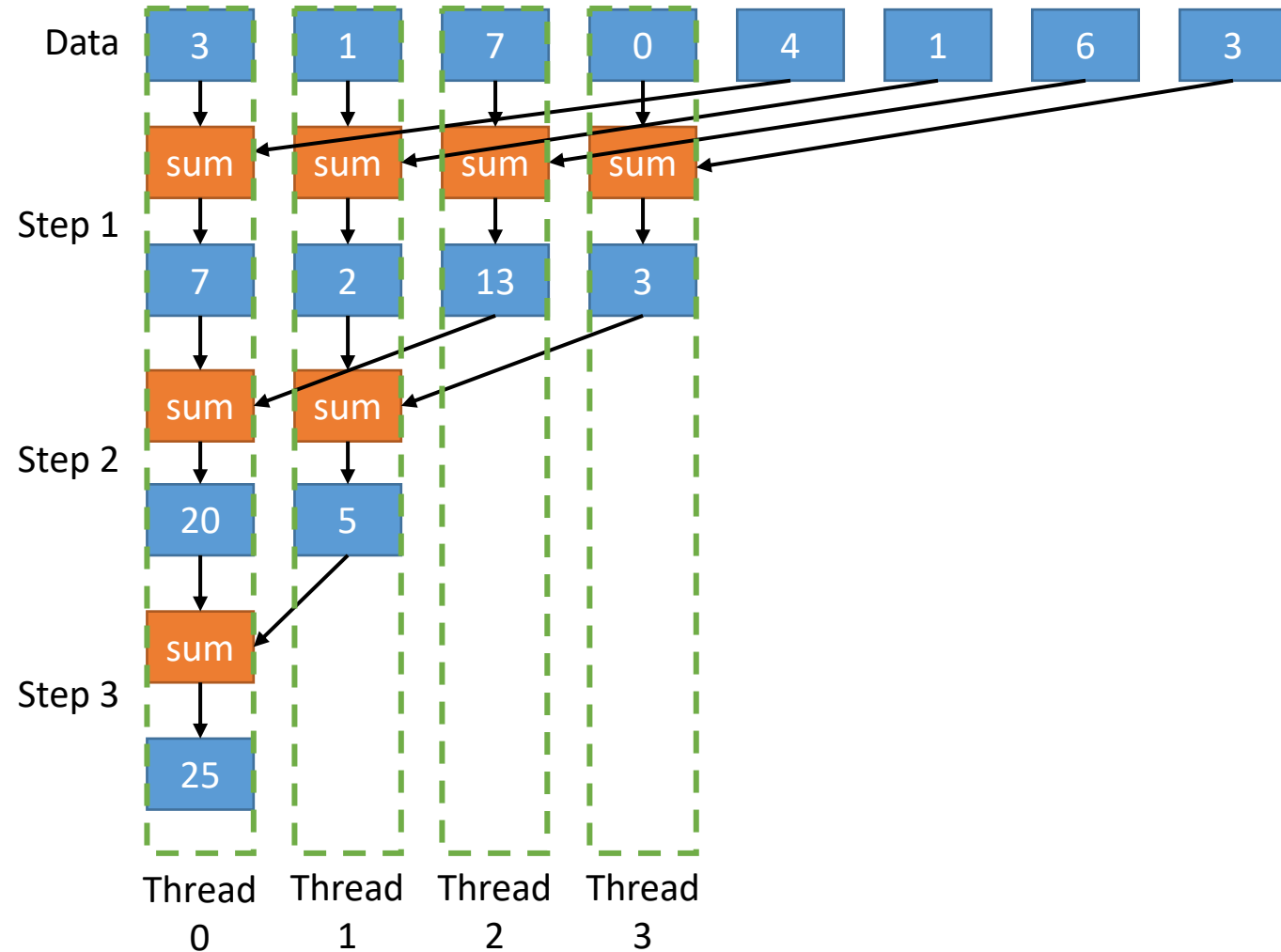
Parallel Computation Patterns

Reduction

Better Thread to Data Mapping

- in some algorithms, one can shift the index usage to improve the divergence behavior
 - Commutative and associative operators
- always compact the partial sums into the front locations in the partialSum[] array
- keep the active threads consecutive

```
for (unsigned int stride = blockDim.x;  
     stride > 0;  
     stride /= 2)  
{  
    __syncthreads();  
    if (t < stride)  
        partialSum[t] += partialSum[t+stride];  
}
```



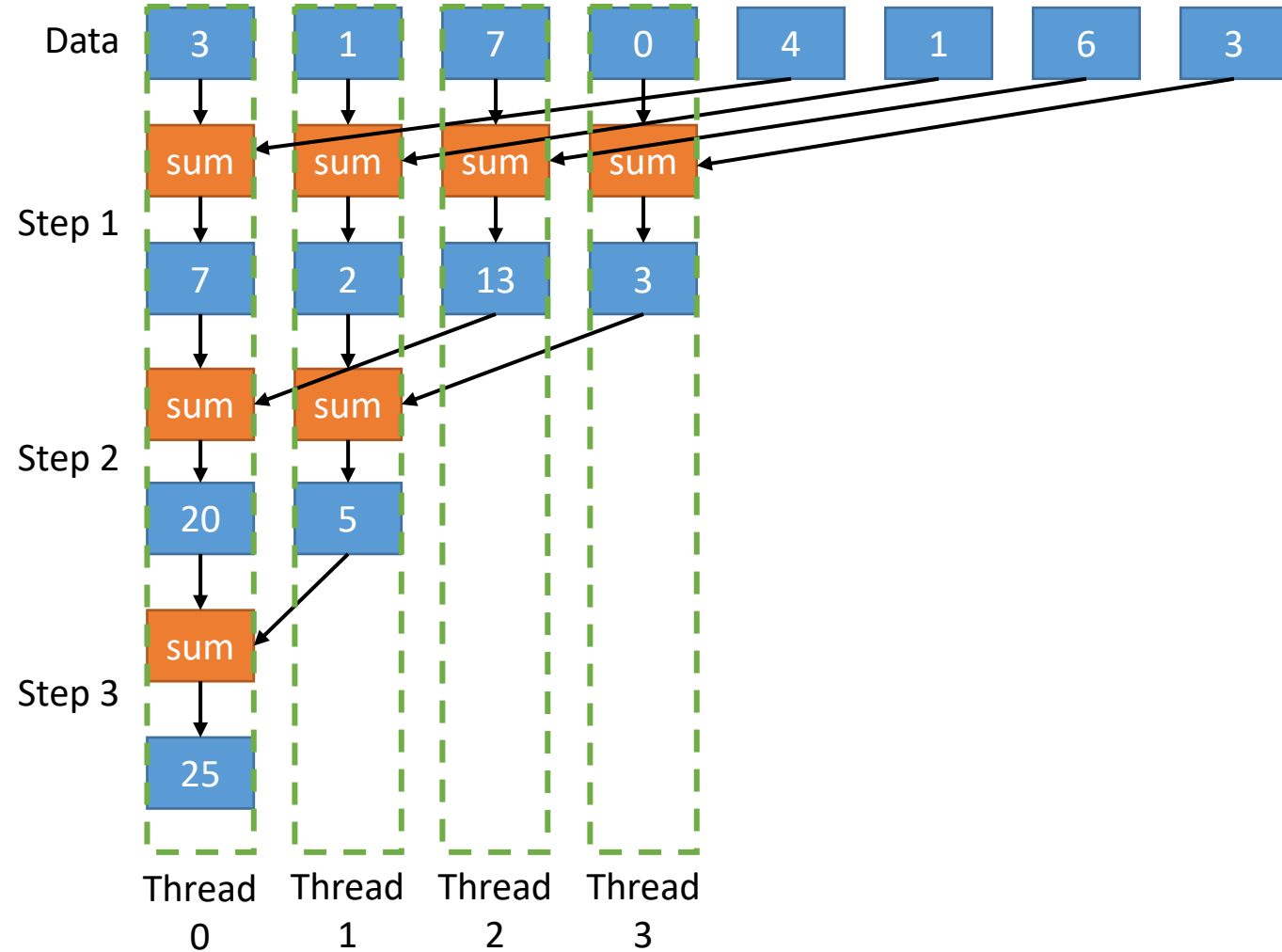
Parallel Computation Patterns

Reduction

A Quick Analysis for a 1024 thread block

- no divergence in the first 5 steps
 - 1024, 512, 256, 128, 64, 32 consecutive threads are active in each step
 - All threads in each warp either all active or all inactive
- the final 5 steps will still have divergence

```
for (unsigned int stride = blockDim.x;  
     stride > 0;  
     stride /= 2)  
{  
    __syncthreads();  
    if (t < stride)  
        partialSum[t] += partialSum[t+stride];  
}
```



Parallel Computation Patterns: Histogram (Atomic Operations)

Univerza v Ljubljani



TECHNISCHE
UNIVERSITÄT
WIEN



VSB TECHNICAL
UNIVERSITY
OF OSTRAVA

IT4INNOVATIONS
NATIONAL SUPERCOMPUTING
CENTER



Co-funded by the
Erasmus+ Programme
of the European Union

This project has been funded with support from the European Commission.

This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Parallel Computation Patterns

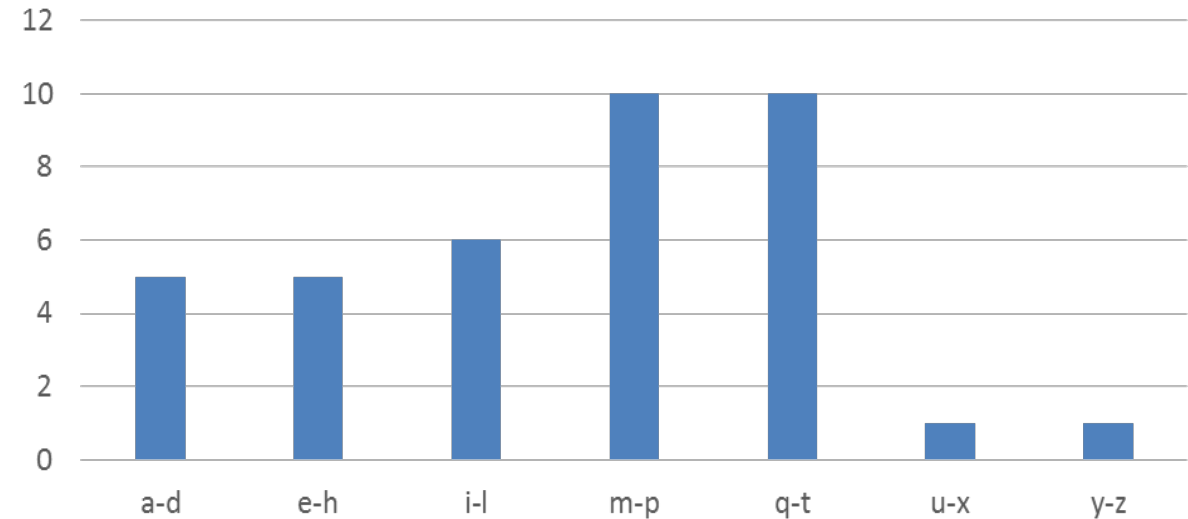
Histogram

Histogram

- A method for extracting notable features and patterns from large data sets
- Basic histograms - for each element in the data set, use the value to identify a “bin counter” to increment

A Text Histogram Example

- define the bins as four-letter sections of the alphabet: a-d, e-h, i-l, n-p, ...
- for each character in an input string, increment the appropriate bin counter.
- in the phrase “Programming Massively Parallel Processors” the output histogram is shown below:



Parallel Computation Patterns

Histogram

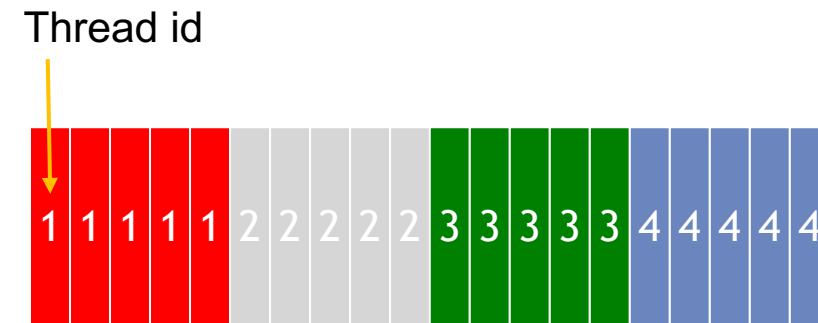
A simple parallel histogram algorithm

- partition the input into sections
- have each thread to take a section of the input
- each thread iterates through its section.
- for each letter, increment the appropriate bin counter

Input Partitioning Affects Memory Access Efficiency

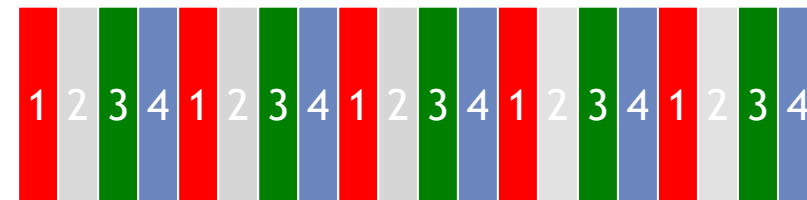
Sectioned partitioning

- results in poor memory access efficiency
- adjacent threads do not access adjacent memory locations
- accesses are not coalesced
- DRAM bandwidth is poorly utilized



Interleaved partitioning

- all threads process a contiguous section of elements
- they all move to the next section and repeat
- the memory accesses are coalesced

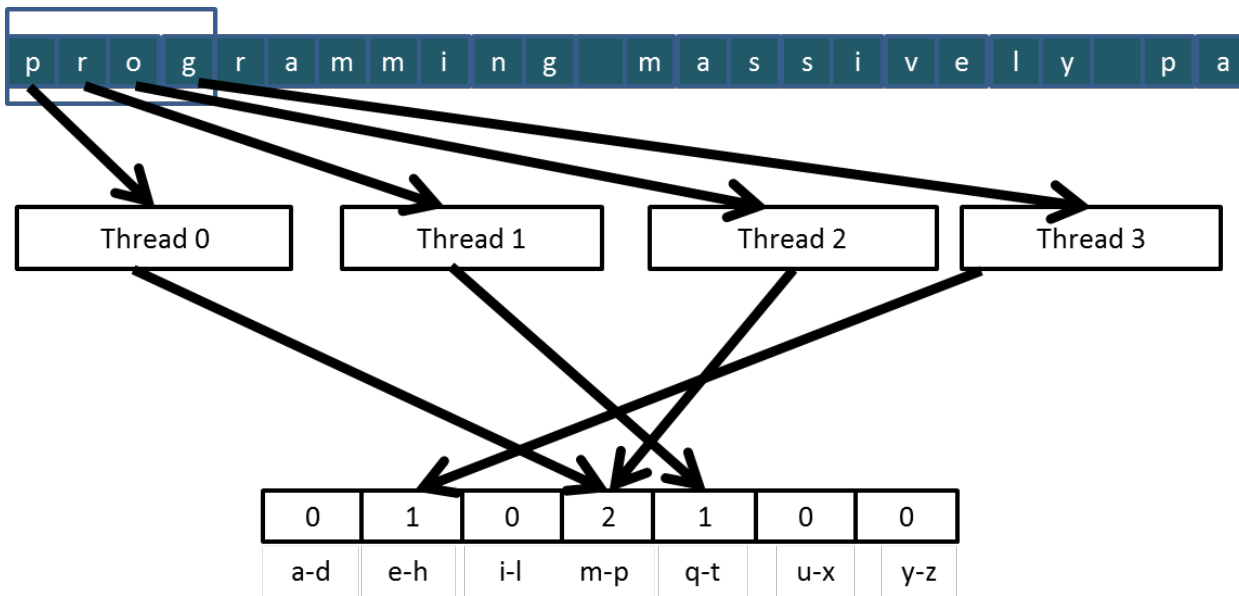


Parallel Computation Patterns

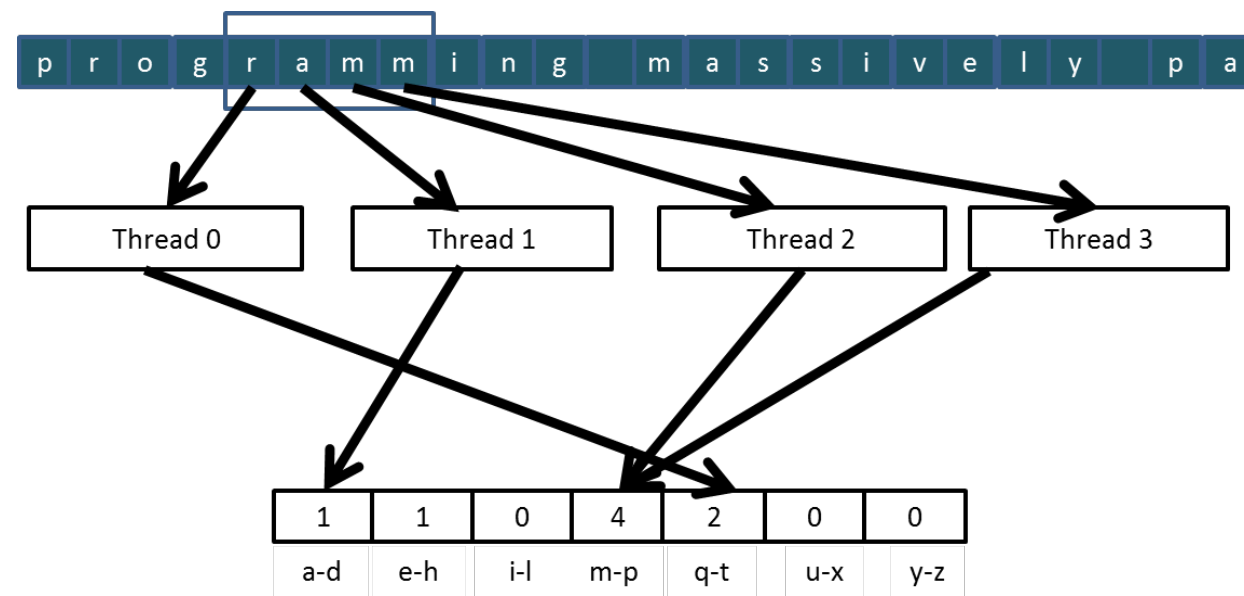
Histogram

Interleaved partitioning of input

Iteration 1



Iteration 2



Parallel Computation Patterns

Histogram

Interleaved partitioning of input

- for every input element thread increments selected bin
- bin incrementation results in
 - **Read-modify-write** operation
 - **can result in Data Race**

Data Race in Parallel Thread Execution

thread1: Old \leftarrow Mem[x]
New \leftarrow Old + 1
Mem[x] \leftarrow New

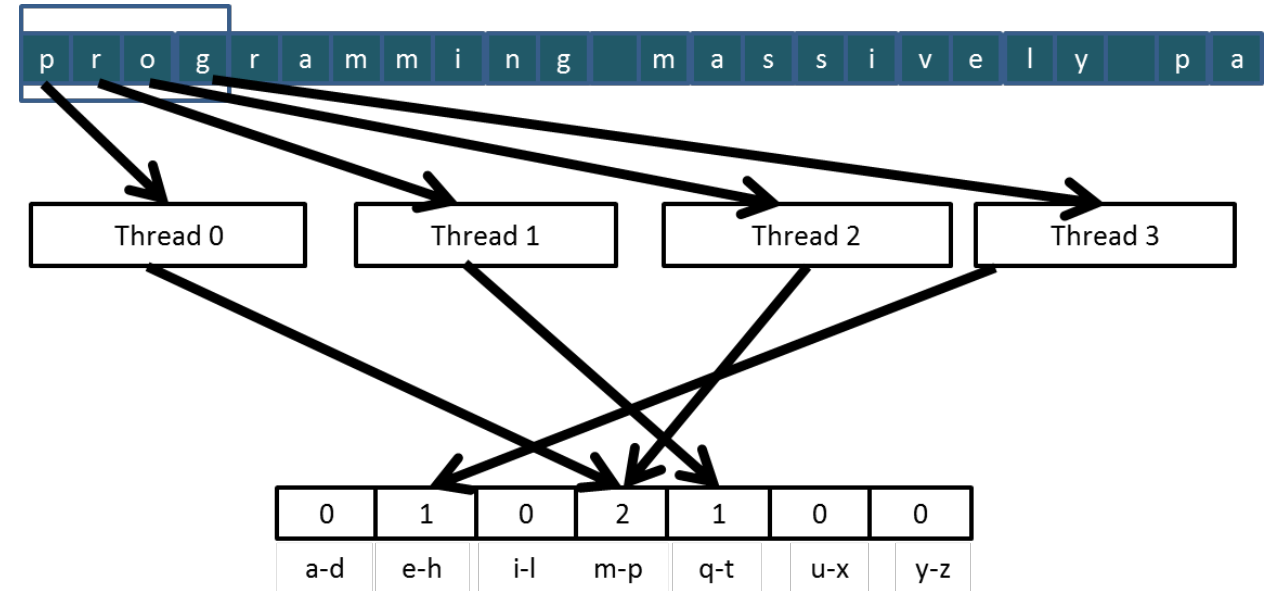
thread2: Old \leftarrow Mem[x]
New \leftarrow Old + 1
Mem[x] \leftarrow New

- **Old** and **New** are per-thread register variables.

Question 1: If Mem[x] was initially 0, what would the value of Mem[x] be after threads 1 and 2 have completed?

Question 2: What does each thread get in their Old variable?

Unfortunately, the answers may vary according to the relative execution timing between the two threads, which is referred to as a **data race**.



Parallel Computation Patterns

Histogram

Data race examples

Time	Thread 1	Thread 2
1	(0) Old \leftarrow Mem[x]	
2	(1) New \leftarrow Old + 1	
3	(1) Mem[x] \leftarrow New	
4		(1) Old \leftarrow Mem[x]
5		(2) New \leftarrow Old + 1
6		(2) Mem[x] \leftarrow New

Timing Scenario #1

- Thread 1 Old = 0
- Thread 2 Old = 1
- Mem[x] = 2 after the sequence

Time	Thread 1	Thread 2
1		(0) Old \leftarrow Mem[x]
2		(1) New \leftarrow Old + 1
3		(1) Mem[x] \leftarrow New
4	(1) Old \leftarrow Mem[x]	
5	(2) New \leftarrow Old + 1	
6	(2) Mem[x] \leftarrow New	

Timing Scenario #2

- Thread 1 Old = 1
- Thread 2 Old = 0
- Mem[x] = 2 after the sequence

Parallel Computation Patterns

Histogram

Data race examples

Timing Scenario #3

- Thread 1 Old = 0
- Thread 2 Old = 0
- Mem[x] = 1 after the sequence

Time	Thread 1	Thread 2
1	(0) Old \leftarrow Mem[x]	
2	(1) New \leftarrow Old + 1	
3		(0) Old \leftarrow Mem[x]
4	(1) Mem[x] \leftarrow New	
5		(1) New \leftarrow Old + 1
6		(1) Mem[x] \leftarrow New

Timing Scenario #4

- Thread 1 Old = 0
- Thread 2 Old = 0
- Mem[x] = 1 after the sequence

Time	Thread 1	Thread 2
1		(0) Old \leftarrow Mem[x]
2		(1) New \leftarrow Old + 1
3	(0) Old \leftarrow Mem[x]	
4		(1) Mem[x] \leftarrow New
5	(1) New \leftarrow Old + 1	
6	(1) Mem[x] \leftarrow New	

Parallel Computation Patterns

Histogram

Atomic Operations Ensure Good Outcomes

```
thread1: Old ← Mem[x]
         New ← Old + 1
         Mem[x] ← New
```

```
thread2: Old ← Mem[x]
         New ← Old + 1
         Mem[x] ← New
```

Or

```
thread2: Old ← Mem[x]
         New ← Old + 1
         Mem[x] ← New
```

```
thread1: Old ← Mem[x]
         New ← Old + 1
         Mem[x] ← New
```

Timing Scenario #3

- Thread 1 Old = 0
- Thread 2 Old = 0
- Mem[x] = 1 after the sequence

Timing Scenario #4

- Thread 1 Old = 0
- Thread 2 Old = 0
- Mem[x] = 1 after the sequence

Time	Thread 1	Thread 2
1	(0) Old ← Mem[x]	
2	(1) New ← Old + 1	
3		(0) Old ← Mem[x]
4	(1) Mem[x] ← New	
5		(1) New ← Old + 1
6		(1) Mem[x] ← New

Time	Thread 1	Thread 2
1		(0) Old ← Mem[x]
2		(1) New ← Old + 1
3	(0) Old ← Mem[x]	
4		(1) Mem[x] ← New
5	(1) New ← Old + 1	
6	(1) Mem[x] ← New	

Atomic Operations

```
thread1: Old ← Mem[x]  
         New ← Old + 1  
         Mem[x] ← New
```

```
thread2: Old ← Mem[x]  
         New ← Old + 1  
         Mem[x] ← New
```

Or

```
thread2: Old ← Mem[x]  
         New ← Old + 1  
         Mem[x] ← New
```

```
thread1: Old ← Mem[x]  
         New ← Old + 1  
         Mem[x] ← New
```

Key Concepts of Atomic Operations

- a read-modify-write operation performed by a single hardware instruction on a memory location address
 - read the old value, calculate a new value, and write the new value to the location
- the hardware ensures that no other threads can perform another read-modify-write operation on the same location until the current atomic operation is complete
 - any other threads that attempt to perform an atomic operation on the same location will typically be held in a queue
 - all threads perform their atomic operations serially on the same location

Atomic Operations

```
thread1: Old ← Mem[x]
         New ← Old + 1
         Mem[x] ← New
```

```
thread2: Old ← Mem[x]
         New ← Old + 1
         Mem[x] ← New
```

Or

```
thread2: Old ← Mem[x]
         New ← Old + 1
         Mem[x] ← New
```

```
thread1: Old ← Mem[x]
         New ← Old + 1
         Mem[x] ← New
```

Atomic Arithmetic Operations in CUDA

- performed by calling functions that are translated into single instructions (a.k.a. intrinsic functions or intrinsics)
 - Atomic add, sub, inc, dec, min, max, exch (exchange), CAS (compare and swap)
 - Read CUDA C programming Guide for details

Example: Atomic Add

```
int atomicAdd(int* address, int val);
```

- reads the 32-bit word old from the location pointed to by address in global or shared memory, computes (old + val), and stores the result back to memory at the same address.
- these three operations are performed in one atomic transaction. The function returns old.

More Atomic Adds in CUDA

- unsigned 32-bit integer atomic add - *unsigned int atomicAdd*
- unsigned 64-bit integer atomic add, single-precision floating-point atomic add, double-precision floating-point atomic add, 16-bit floating-point atomic add, ...

Parallel Computation Patterns

Histogram

A Basic Text Histogram Kernel

- The kernel receives a pointer to the input buffer of byte values
- Each thread process the input in a strided pattern

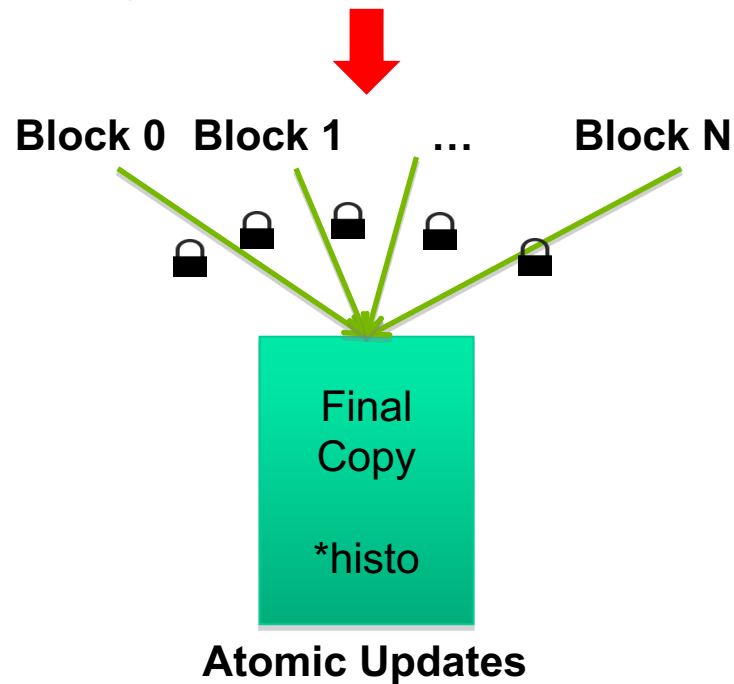
```
__global__ void histo_kernel(  
    unsigned char *buffer,  
    long size,  
    unsigned int *histo)  
{  
    int i = threadIdx.x + blockIdx.x * blockDim.x;  
  
    // stride is total number of threads  
    int stride = blockDim.x * gridDim.x;  
  
    // All threads handle blockDim.x * gridDim.x  
    // consecutive elements  
    while (i < size) {  
        int alphabet_position = buffer[i] - "a";  
        if (alphabet_position >= 0 && alphabet_position < 26)  
            atomicAdd(&(histo[alphabet_position/4]), 1);  
        i += stride;  
    }  
}
```

Parallel Computation Patterns

Histogram

A Basic Text Histogram Kernel

Heavy contention and serialization



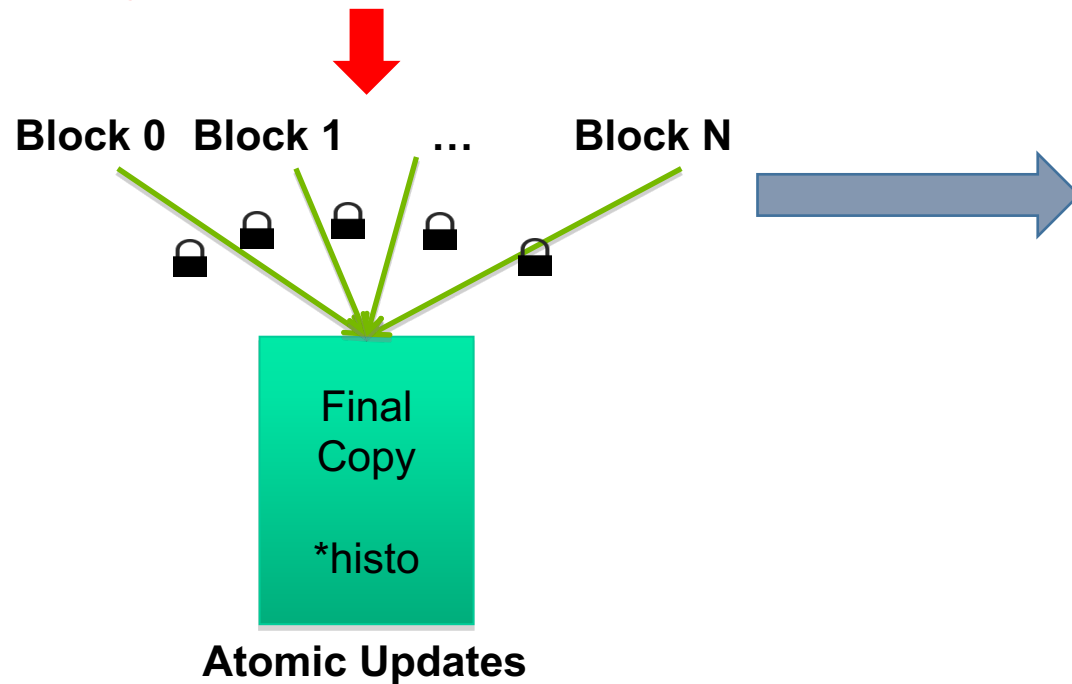
```
__global__ void histo_kernel(  
    unsigned char *buffer,  
    long size,  
    unsigned int *histo)  
{  
    int i = threadIdx.x + blockIdx.x * blockDim.x;  
  
    // stride is total number of threads  
    int stride = blockDim.x * gridDim.x;  
  
    // All threads handle blockDim.x * gridDim.x  
    // consecutive elements  
    while (i < size) {  
        int alphabet_position = buffer[i] - "a";  
        if (alphabet_position >= 0 && alphabet_position < 26)  
            atomicAdd(&(histo[alphabet_position/4]), 1);  
        i += stride;  
    }  
}
```

Parallel Computation Patterns Histogram

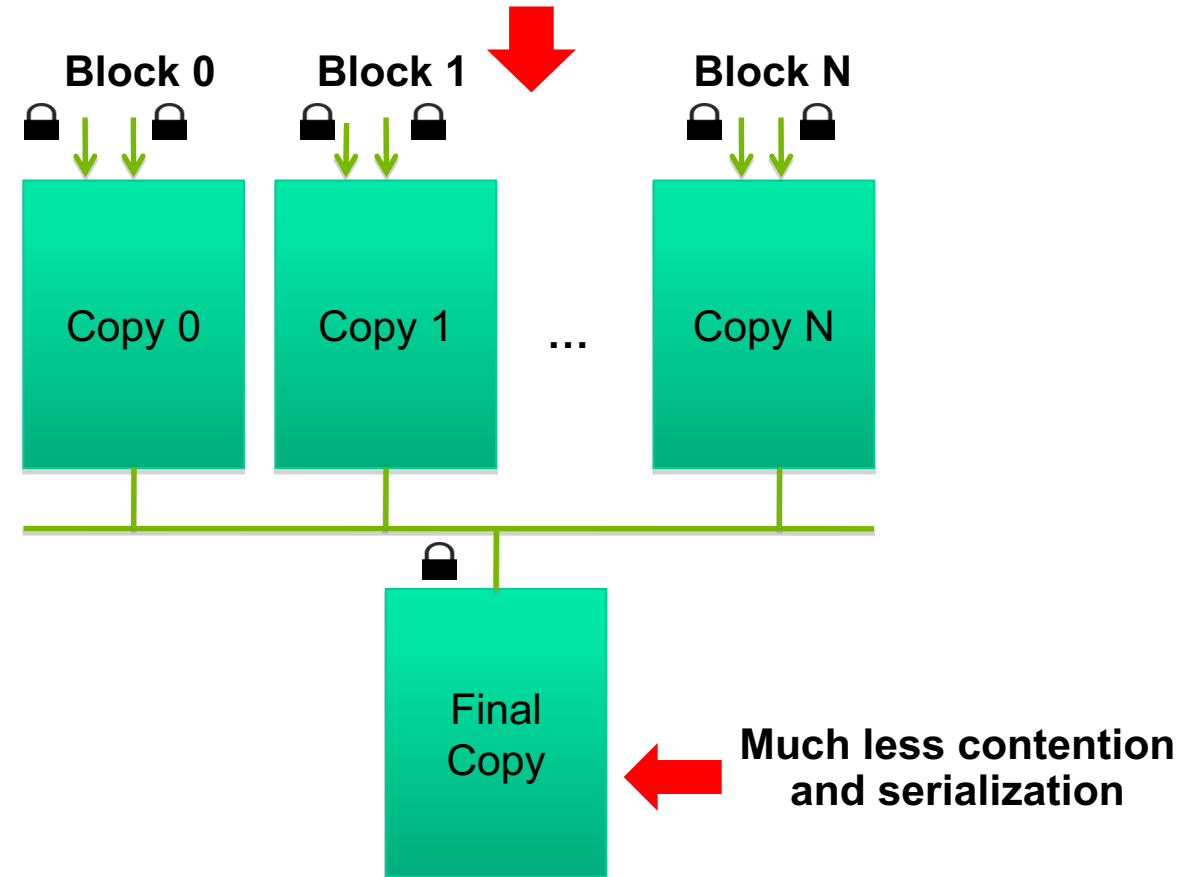
Privatization

- Privatization is a technique for reducing latency, increasing throughput, and reducing serialization

Heavy contention and serialization



Much less contention and serialization



Parallel Computation Patterns

Histogram

Privatization

- privatization is a technique for reducing latency, increasing throughput, and reducing serialization

Cost and Benefit of Privatization

Cost

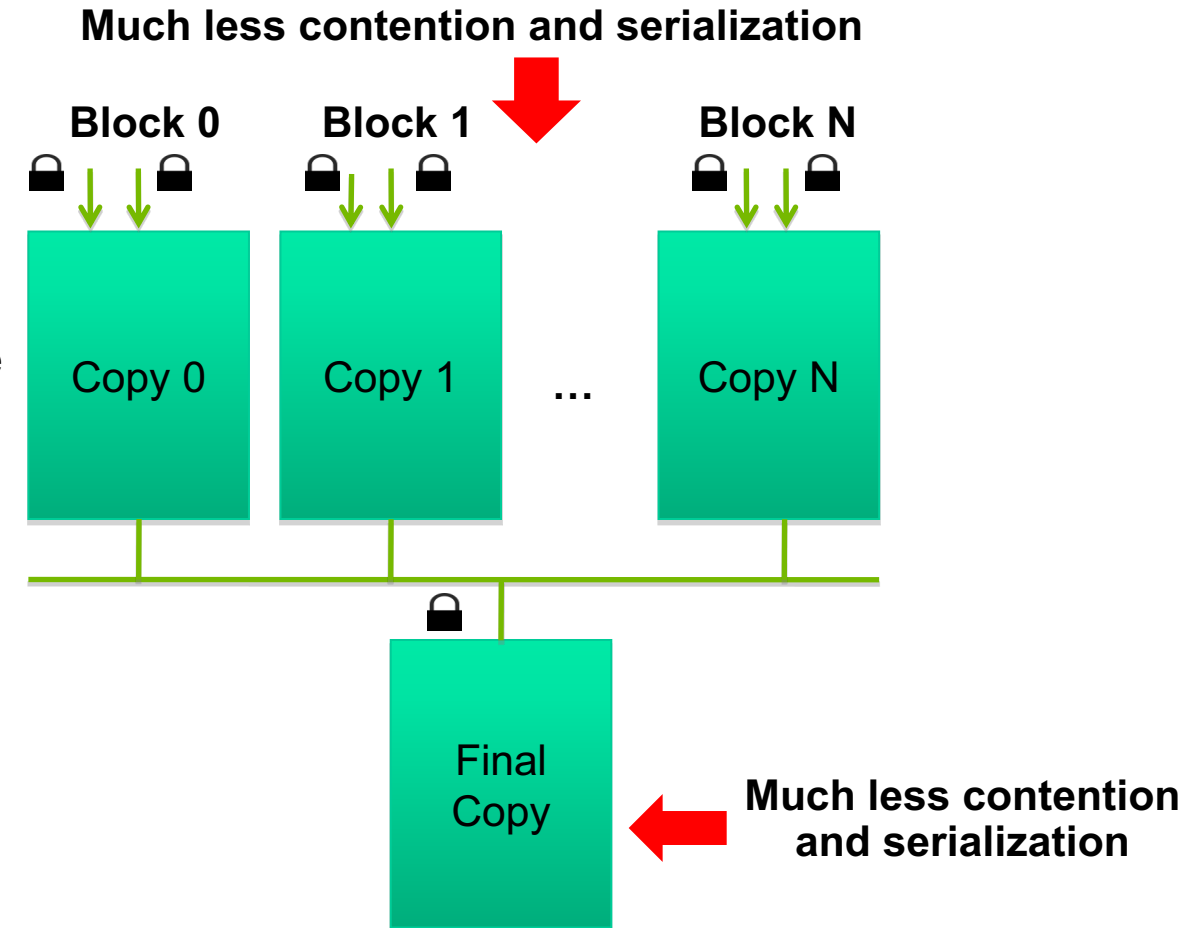
- overhead for creating and initializing private copies
- overhead for accumulating the contents of private copies into the final copy

Benefit

- much less contention and serialization in accessing both the private copies and the final copy
- the overall performance can often be improved more than 10x

Shared Memory Atomics for Histogram

- each subset of threads are in the same block
- much higher throughput than DRAM (100x) or L2 (10x) atomics
- less contention – only threads in the same block can access a shared memory variable
- this is a very important use case for shared memory!



Parallel Computation Patterns

Histogram

Privatized Histogram kernel

Create private copies of the histo[] array for each thread block

Initialize the bin counters in the private copies of histo[]

Build Private Histogram

Build Final Histogram

```
__global__ void histo_kernel(unsigned char *buffer,
                             long size, unsigned int *histo)
{
    __shared__ unsigned int histo_private[7];

    if (threadIdx.x < 7) histo_private[threadIdx.x] = 0;

    __syncthreads();

    int i = threadIdx.x + blockIdx.x * blockDim.x;
    // stride is total number of threads
    int stride = blockDim.x * gridDim.x;
    while (i < size) {
        int alphabet_position = buffer[i] - "a";
        if (alphabet_position >= 0 && alphabet_position < 26)
            atomicAdd(&(private_histo[alphabet_position/4]), 1);
        i += stride;
    }

    // wait for all other threads in the block to finish
    __syncthreads();

    if (threadIdx.x < 7) {
        atomicAdd(&(histo[threadIdx.x]), private_histo[threadIdx.x] );
    }
}
```

More on Privatization

- privatization is a powerful and frequently used technique for parallelizing applications
- the operation needs to be associative and commutative
 - histogram add operation is associative and commutative
 - no privatization if the operation does not fit the requirement
- the private histogram size needs to be small
 - fits into shared memory
- What if the histogram is too large to privatize?
 - sometimes one can partially privatize an output histogram and use range testing to go to either global memory or shared memory

Efficient Host-Device Data Transfer and CUDA Streams

Univerza v Ljubljani



TECHNISCHE
UNIVERSITÄT
WIEN



VSB TECHNICAL
UNIVERSITY
OF OSTRAVA

IT4INNOVATIONS
NATIONAL SUPERCOMPUTING
CENTER



Co-funded by the
Erasmus+ Programme
of the European Union

This project has been funded with support from the European Commission.

This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

CPU-GPU Data Transfer using DMA

- DMA (Direct Memory Access) hardware is used by `cudaMemcpy()` for better efficiency
 - CPU is not used and perform useful calculations
 - DMA is hardware unit used to transfer given number of bytes
 - between physical memory address space regions
 - uses system interconnect: in current systems PCI-Express

Virtual Memory Management

- **Problem for DMA:** *not all variables and data structures are always located in the physical memory*

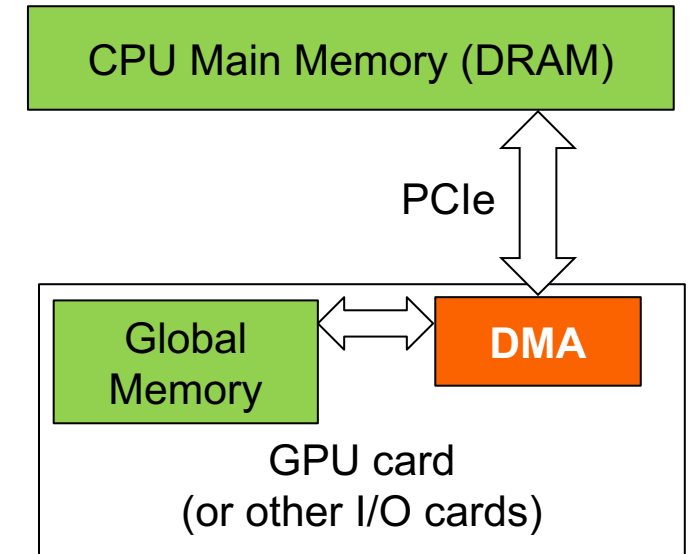
Data Transfer and Virtual Memory

- DMA uses **ONLY** physical addresses
- **when `cudaMemcpy()` copies an array, it is implemented as one or more DMA transfers**

Solution: Pinned Memory

- pinned memory are virtual memory pages that are specially selected, and they cannot be paged out (removed from physical memory)
- pinned memory is allocated with a special system API function call

CPU memory that serve as the source or destination of a DMA transfer must be allocated as pinned memory

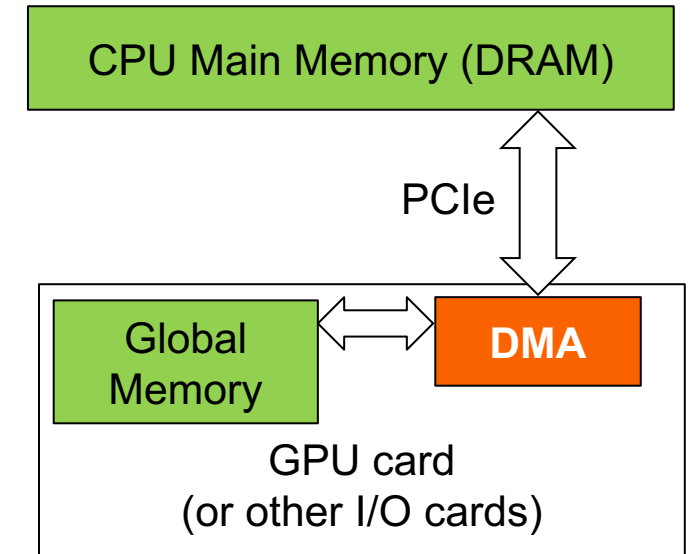


CUDA data transfer uses pinned memory.

- the DMA used by `cudaMemcpy()` requires that any source or destination in the host memory is allocated as pinned memory
- if a source or destination of a `cudaMemcpy()` in the host memory is not allocated in pinned memory, it needs to be first copied to a pinned memory – extra overhead
- `cudaMemcpy()` is faster if the host memory source or destination is allocated in pinned memory since no extra copy is needed

Using Pinned Memory in CUDA

- use the allocated pinned memory and its pointer the same way as those returned by `malloc()`;
- the only difference is that the allocated memory cannot be paged by the OS
- the `cudaMemcpy()` function should be about 2X faster with pinned memory
- pinned memory is a limited resource
- over-subscription can have serious consequences



Allocate/Free Pinned Memory

cudaHostAlloc(), three parameters

- Address of pointer to the allocated memory
- Size of the allocated memory in bytes
- Option – use `cudaHostAllocDefault` for now

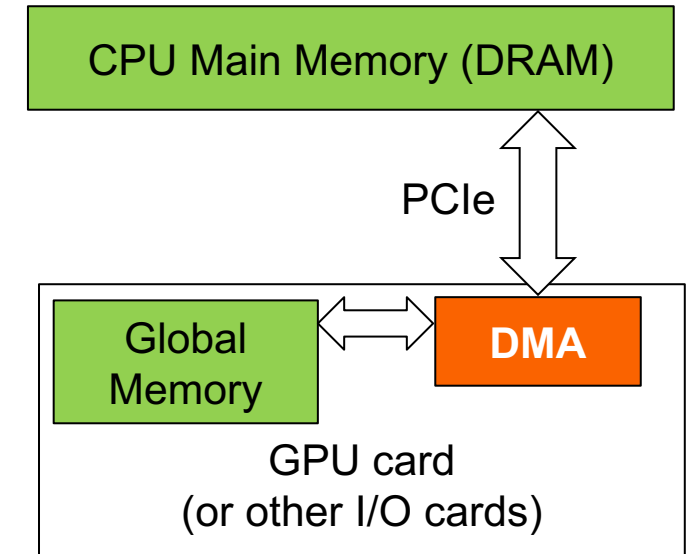
cudaFreeHost(), one parameter

- Pointer to the memory to be freed

Pinned Memory

Example: Vector Addition Host Code

```
int main()
{
    float *h_A, *h_B, *h_C;
    ...
    cudaHostAlloc((void **) &h_A, N* sizeof(float), cudaHostAllocDefault);
    cudaHostAlloc((void **) &h_B, N* sizeof(float), cudaHostAllocDefault);
    cudaHostAlloc((void **) &h_C, N* sizeof(float), cudaHostAllocDefault);
    ...
    // cudaMemcpy() runs 2X faster
}
```



Concurrency using CUDA Streams

System can perform multiple CUDA operations simultaneously:

- multiple CUDA kernels on GPU
- one cudaMemcpyAsync from Host to Device
- one cudaMemcpyAsync from Device to Host
- computation on the CPU

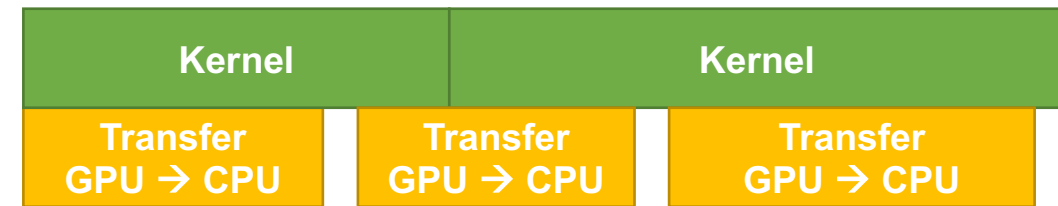
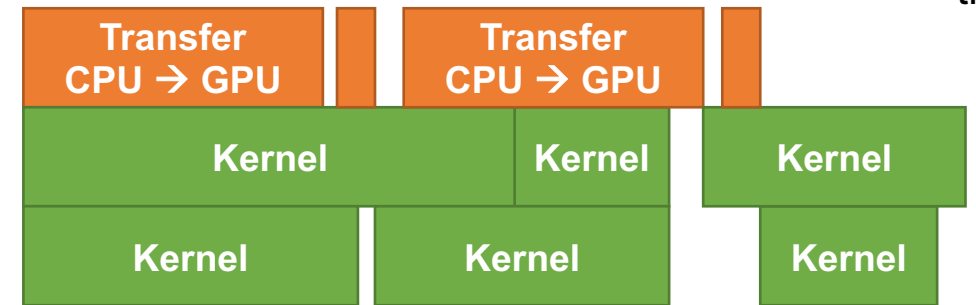
CUDA Stream

- a sequence of operations that execute in issue-order on the GPU

Stream Semantics

- Two operations issued into the same stream will execute in issue-order. Operation B issued after Operation A will not begin to execute until Operation A has completed.
- Two operations issued into separate streams have no ordering prescribed by CUDA. Operation A issued into stream 1 may execute before, during, or after Operation B issued into stream 2.
- Operation: Usually, cudaMemcpyAsync or a kernel call. More generally, most CUDA API calls that take a stream parameter, as well as stream callbacks.

Sequential execution



Concurrent execution

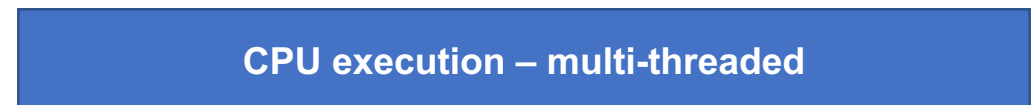
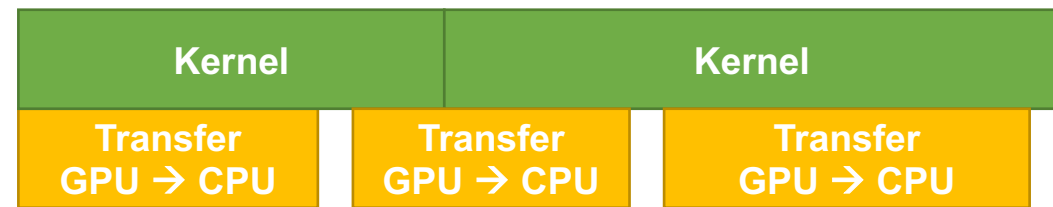
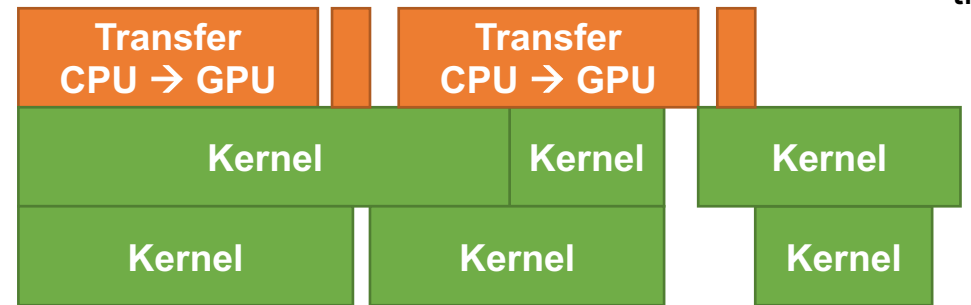
Concurrency using CUDA Streams

Default Stream (aka Stream '0')

- Stream used when no stream is specified
- Completely synchronous w.r.t. host and device
 - As if `cudaDeviceSynchronize()` inserted before and after every CUDA operation
- Exceptions – asynchronous w.r.t.
 - `hostKernel` launches in the default stream
 - `cudaMemcpy*Async`
 - `cudaMemset*Async`
 - `cudaMemcpy` within the same device
 - H2D `cudaMemcpy` of 64kB or less

Requirements for Concurrency

Sequential execution

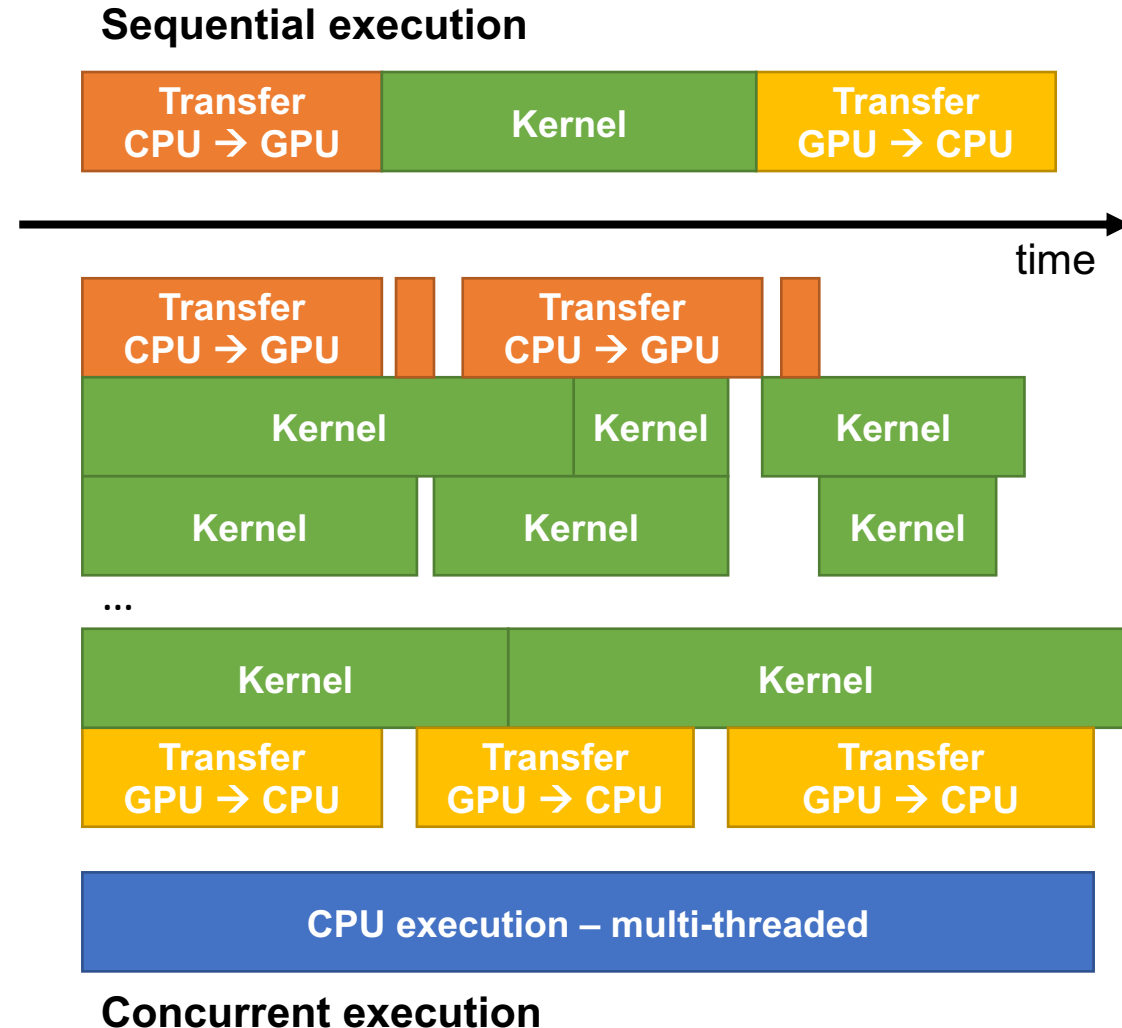


Concurrent execution

Concurrency using CUDA Streams

CUDA Streams – How to use them?

- **Create/Destroy**
 - `cudaStream_t stream;`
 - `cudaStreamCreate(&stream);`
 - `cudaStreamDestroy(stream);`
- **Launch**
 - `my_kernel<<<grid,block,0,stream>>>(...);`
 - `cudaMemcpyAsync(..., stream);`
- **Synchronize**
 - `cudaStreamSynchronize(stream);`



Concurrency using CUDA Streams

Basic Example 1: KERNEL CONCURRENCY

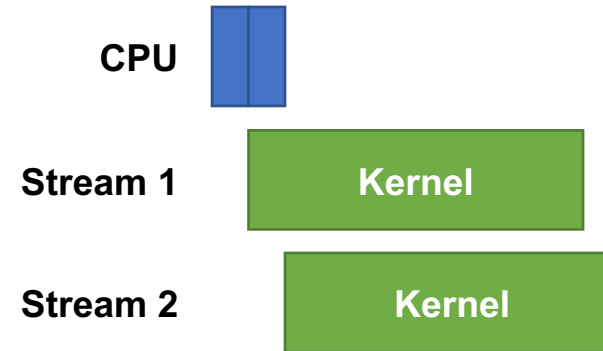
- assume foo only utilizes 50% of the GPU
- using user streams

```
cudaStream_t stream1, stream2;
```

```
cudaStreamCreate(&stream1);  
cudaStreamCreate(&stream2);
```

```
foo<<<blocks, threads, 0, stream1>>>();  
foo<<<blocks, threads, 0, stream2>>>();
```

```
cudaStreamDestroy(stream1);  
cudaStreamDestroy(stream2);
```



Concurrency using CUDA Streams

Basic Example 2: CONCURRENT MEMORY COPIES

- assume pinned memory

Synchronous

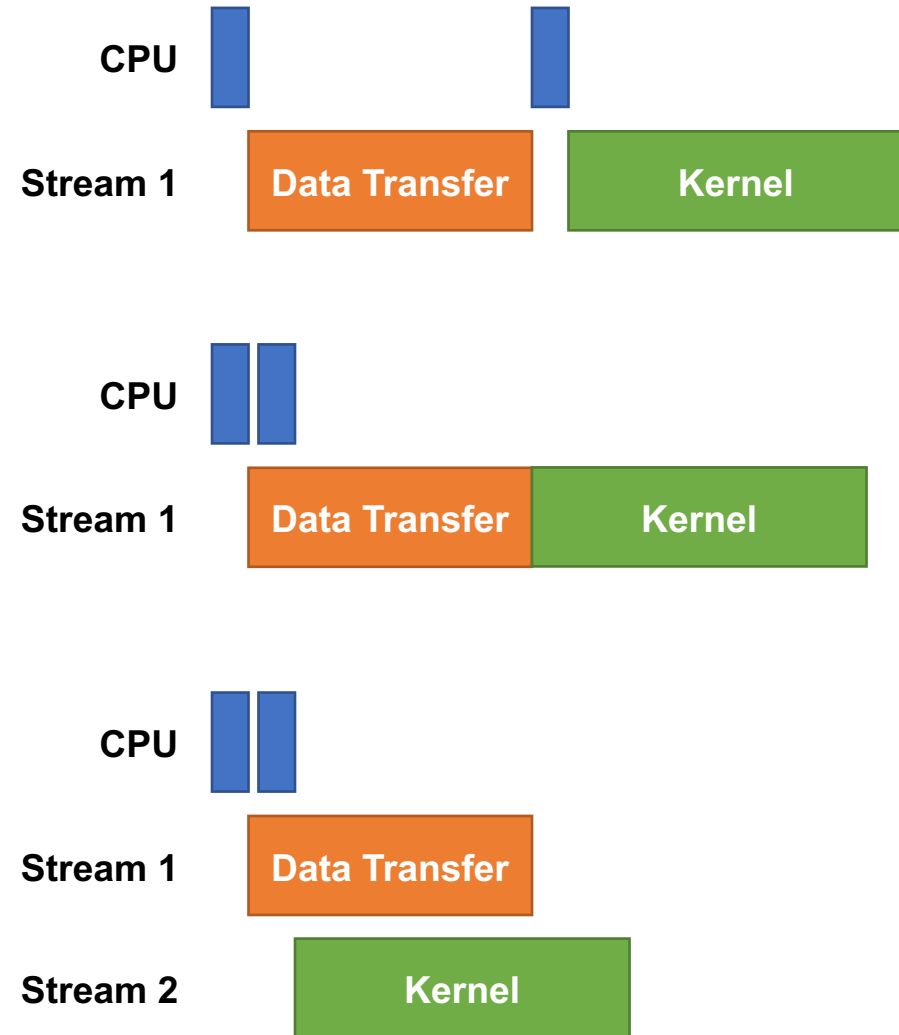
```
cudaMemcpy(...);  
foo<<<...>>>();
```

Asynchronous Same Stream

```
cudaMemcpyAsync(..., stream1);  
foo<<<..., stream1>>>();
```

Asynchronous Different Streams

```
cudaMemcpyAsync(..., stream1);  
foo<<<..., stream2>>>();
```



CPU-GPU Data Transfer using DMA

SCtrain

SUPERCOMPUTING
KNOWLEDGE
PARTNERSHIP

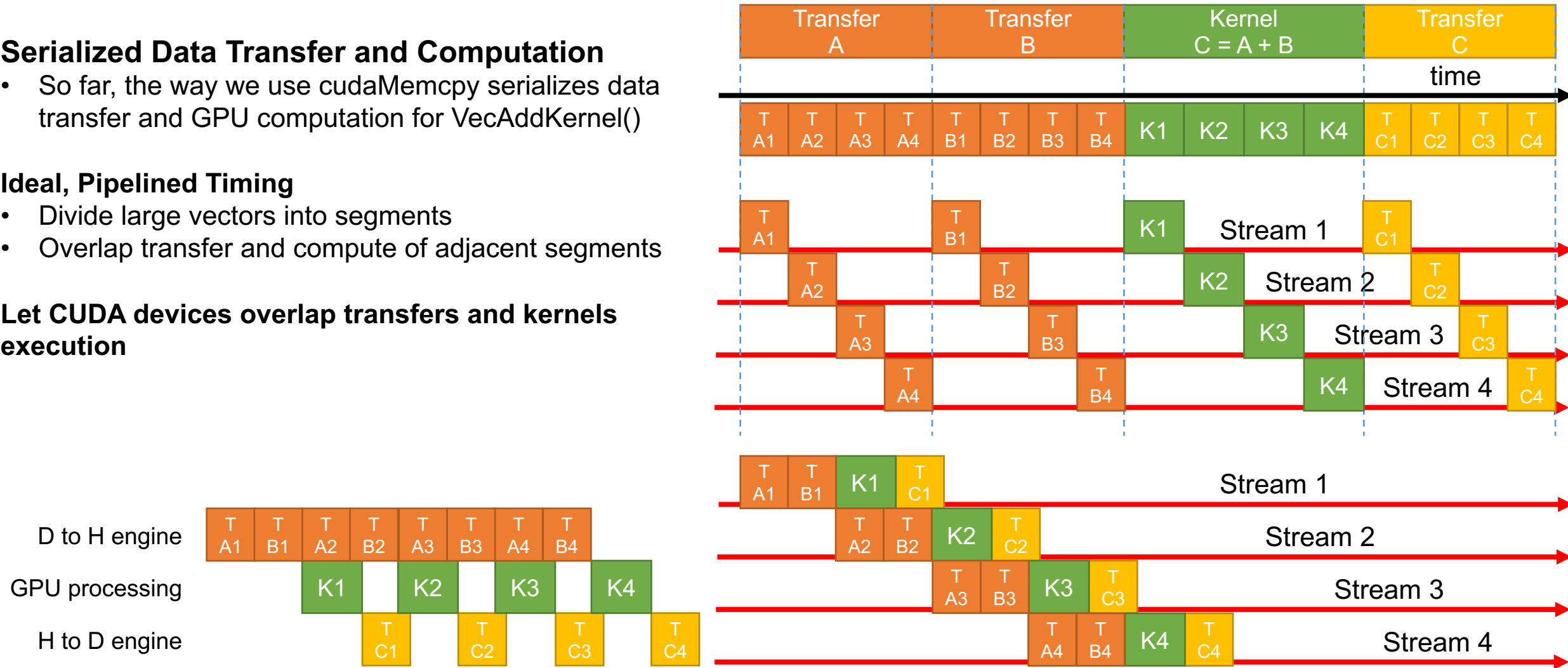
Serialized Data Transfer and Computation

- So far, the way we use cudaMemcpy serializes data transfer and GPU computation for VecAddKernel()

Ideal, Pipelined Timing

- Divide large vectors into segments
- Overlap transfer and compute of adjacent segments

Let CUDA devices overlap transfers and kernels execution



Serialized Data Transfer and Computation

//non-streamed version

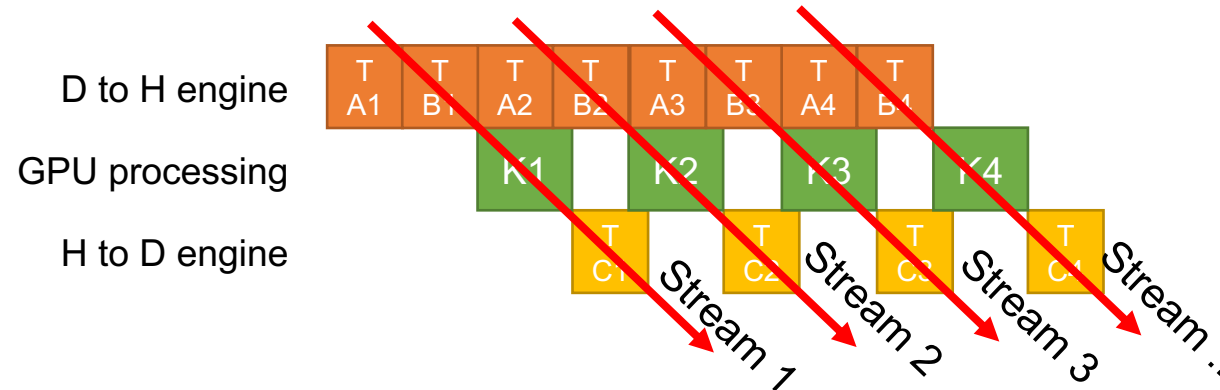
```
cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);
Kernel<<<b, t>>>(d_a, d_b, d_c, N);
cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);
```

//streamed version

```
// c - number of pipeline phases
// ns - total number of streams used
// size - size of input arrays
```

```
cudaStream_t stream[ns];
for (int i = 0; i < ns; ++i)
    cudaStreamCreate(&stream[i]);

for (int i = 0, i < c; i++){
    size_t off = (size/c)*i;
    cudaMemcpyAsync(d_a+off, h_a+off, size/c, cudaMemcpyHostToDevice, stream[i%ns]);
    cudaMemcpyAsync(d_b+off, h_b+off, size/c, cudaMemcpyHostToDevice, stream[i%ns]);
    Kernel<<<b/c, t, 0, stream[i%ns]>>>(d_a+off, d_b+off, d_c+off, N/c);
    cudaMemcpyAsync(h_c+off, d_c+off, size/c, cudaMemcpyDeviceToHost, stream[i%ns]);
}
```



HeatFlow: GPU Accelerated Version

Univerza v Ljubljani



TECHNISCHE
UNIVERSITÄT
WIEN



VSB TECHNICAL
UNIVERSITY
OF OSTRAVA

IT4INNOVATIONS
NATIONAL SUPERCOMPUTING
CENTER



Co-funded by the
Erasmus+ Programme
of the European Union

This project has been funded with support from the European Commission.

This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Thank you for your attention!

<http://sctrain.eu/>

Univerza v Ljubljani

