IEEE IUS 2023
International Ultrasonics Symposium
Palais des congrès de Montréal || September 3-8, 2023

us4us®

CUDA refresher

Short-course: Ultrasound Signal Processing with GPUs

Marcin Lewandowski, CEO
marcin@us4us.eu

LAB 4US

License / Attribution

- Materials for the short-course „**Ultrasound Signal Processing with GPUs – Introduction to Parallel Programming**” are licensed by us4us Ltd. the IPPT PAN under the Creative Commons Attribution-NonCommercial 4.0 International License.
- Some slides and examples are borrowed from the course „The GPU Teaching Kit” that is licensed by NVIDIA and the University of Illinois under the Creative Commons Attribution-NonCommercial 4.0 International License. All the borrowed slides are marked with NVIDIA ILLINOIS

DISCLAIMER

- Content and works included in this educational materials/presentations are subject to the rules resulting from the Polish regulations on copyright and related rights. Downloading to a computer and copying of all or part of educational materials/presentations is permitted only for non-commercial use under the terms of the CC BY-NC-ND 4.0 licence.
- Reproduction, processing in whole or in part and any type of use going beyond that referred to in the preceding sentence requires the written permission of the creator/entity holding the copyright to the educational material/presentation.
- In the educational material/presentation, content and works copyrighted by others have been used within the scope of permitted use of protected works — such content and works have been marked as such. If you nevertheless notice a copyright infringement, please inform us accordingly. In the event of an infringement, we will remove the relevant content immediately.

- Every effort has been made to ensure that all borrowed material has its source and/or author correctly stated.
- The information contained in the material has been prepared to the best knowledge of the authors and is taken from sources believed to be reliable, although the authors do not guarantee its accuracy or completeness.

LAB4US

| Ultrasound Edu | Research | Innovation |

CC BY NC

WELCOME!

Ultrasound Signal Processing with GPUs — Introduction to Parallel Programming

us4us®
LAB 4US

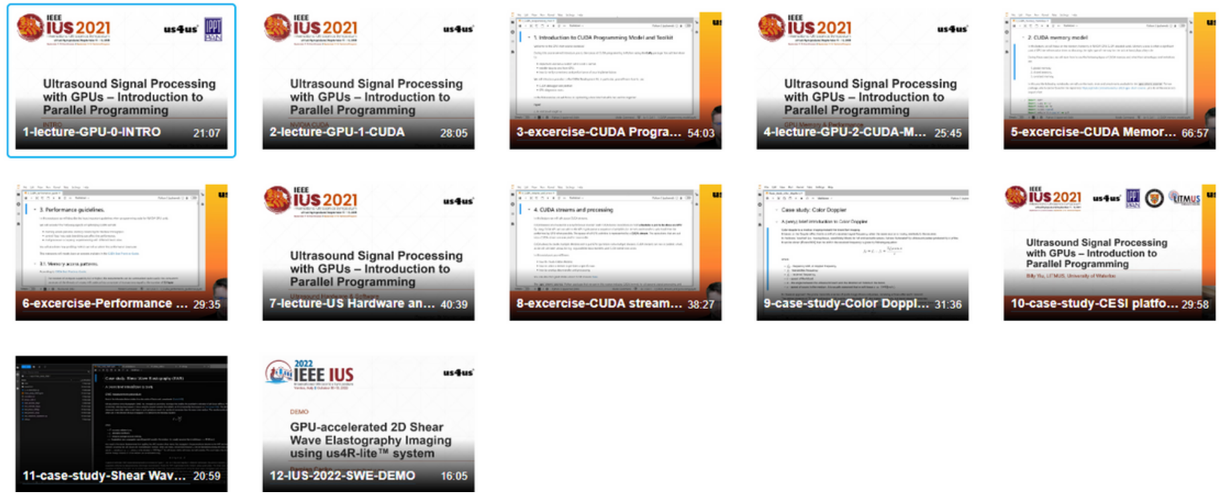
UNIVERSITY OF WATERLOO

IPPT PAN

- **This is our 3rd edition of the Ultrasound-GPU short-course!**
  - We are going to modify the course to focus more on practical ultrasound use-cases.
  - Thus, the participants will need to familiarize themselves with the on-line lecture materials before attending the short-course!
- **SAME great Team!**
  - Dr Marcin Lewandowski <marcin@us4us.eu> / us4us Ltd. / IPPT PAN.
  - Piotr Jarosik <piotr.jarosik@us4us.eu> / us4us Ltd. / IPPT PAN.
  - Dr Billy Yiu <billy.yiu@uwaterloo.ca> / University of Waterloo

INTRO & PRE-REQUISITE

- In 2023, we would like to upgrade our short-course to show more practical US algorithms.
- We believe that you go over our recorder 2022 US-GPU short-course!?



https://github.com/Lab4US/gpu-short-course

https://vimeo.com/showcase/2022-us-gpu-short-course

PLAN

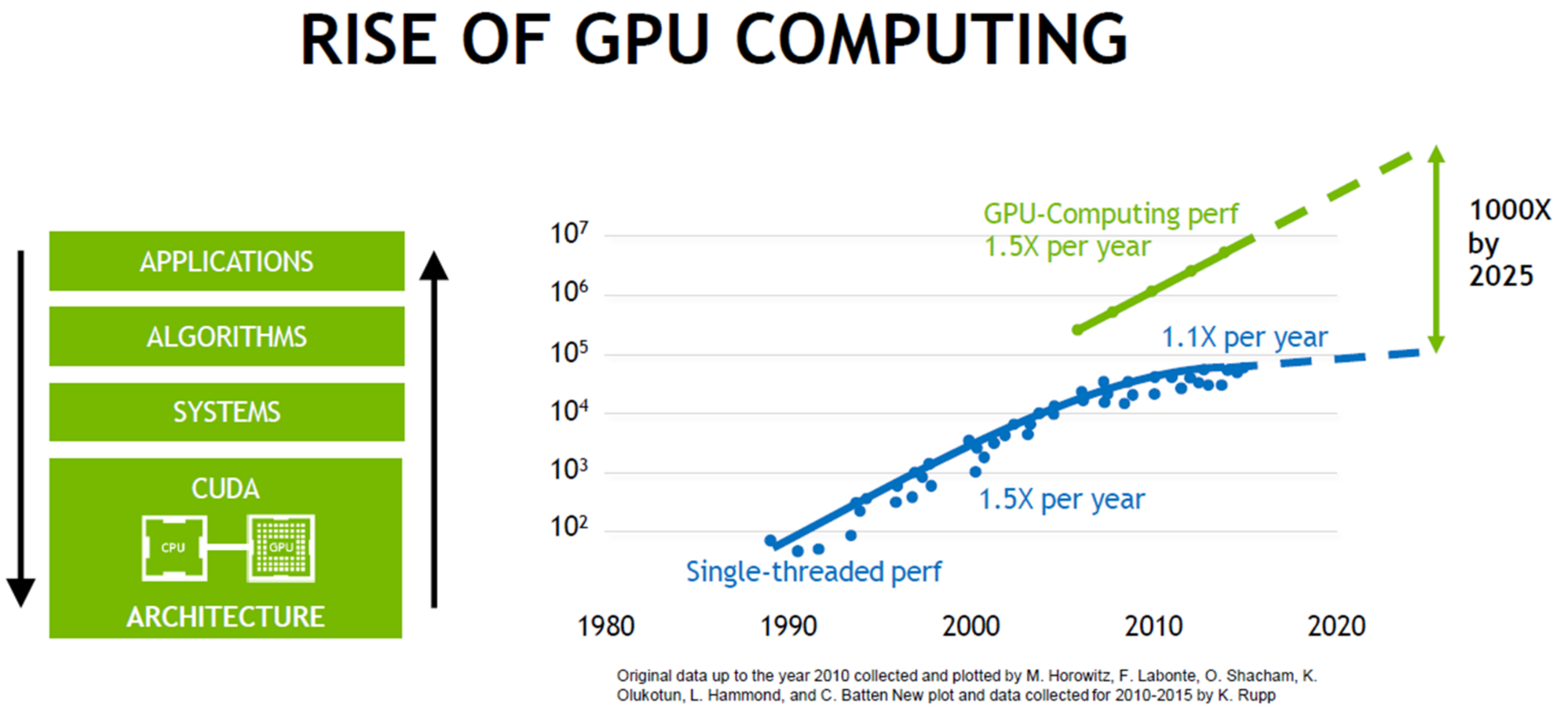
- **More or less CUDA ... (MARCIN)**
  - NVIDIA CUDA Refresher
  - More than CUDA & NVIDIA

- **US Beamforming Algorithms (PIOTR)**
  - DAS (Delay & Sum) as Interpolation and Reduction
  - Matrix-Array & Row-Column Arrays Beamforming

- **Speed-of-Sound Estimatin + Deep Learning on GPU (BILLY)**

NVIDA CUDA Refresher

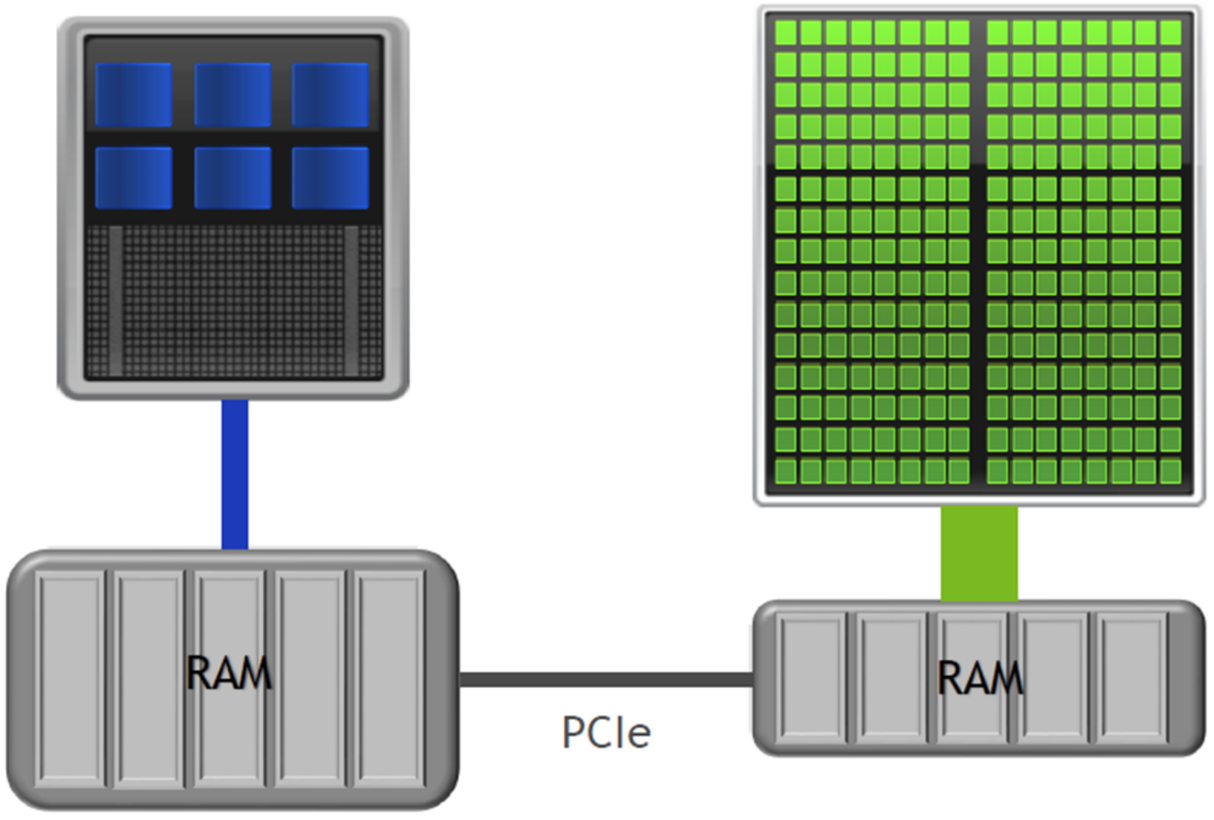
Why GPUs?



Source: NVIDIA, Andreas Hehn, High Throughput with GPUs, 2018

CPU & GPU – heterogeneous computing

CPU
Optimized for
Serial Tasks

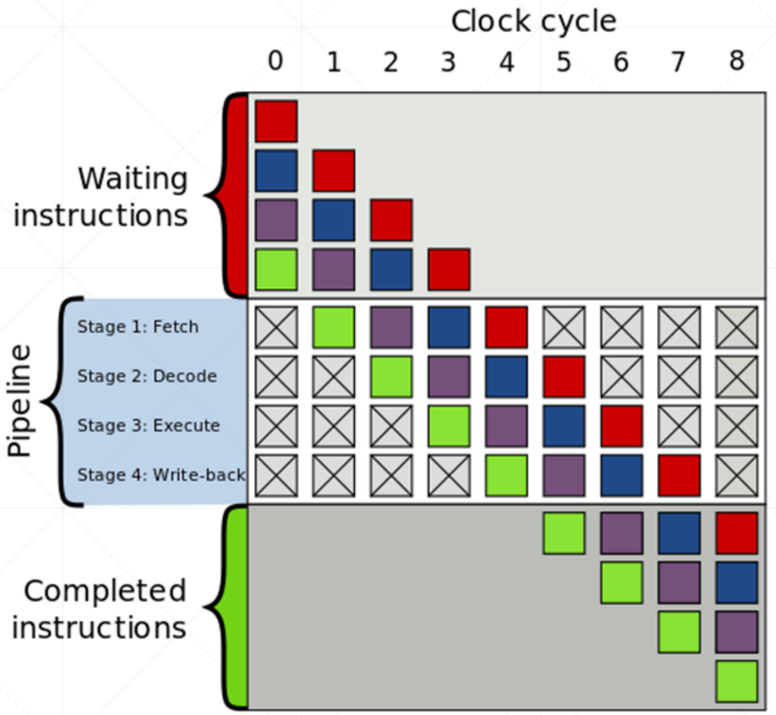


GPU
Optimized for
Parallel Tasks

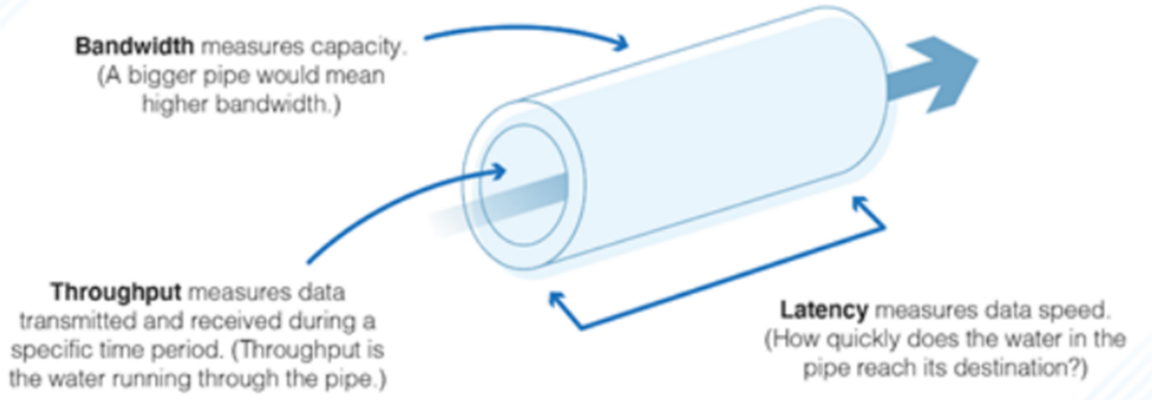
Source: NVIDIA, Andreas Hehn, High Throughput with GPUs, 2018

Latency and Throughput

- **Latency** – time to get a solution
  - metric is time [sec]
    - minimize time, at the expense of power
- **Throughput** – number of operation (tasks) processed per unit of time
  - metric: ops/time [GFLOPS]
    - minimize energy per operation



- CPU: optimized for latency
- GPU: optimized for throughput



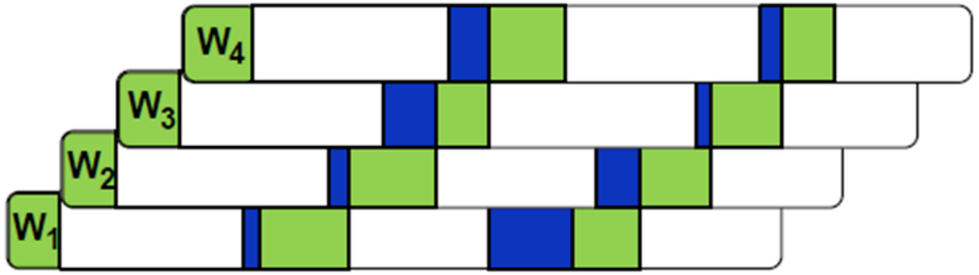
Source: https://www.dnsstuff.com/latency-throughput-bandwidth | https://en.wikipedia.org/wiki/Instruction_pipelining

LOW LATENCY OF HIGH THROUGHPUT?

CPU architecture must **minimize latency** within each thread

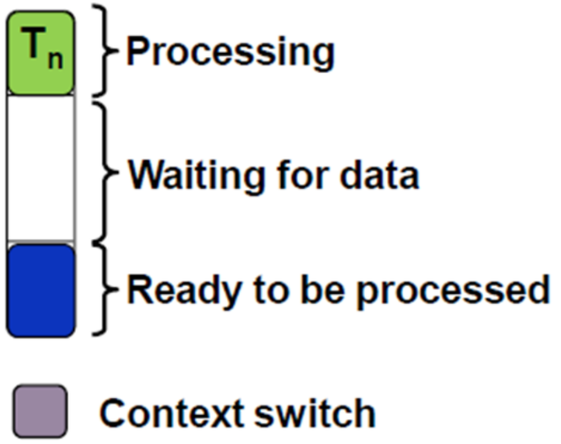
CPU core – Low Latency Processor

GPU architecture **hides latency**
with computation from other threads (warps)



GPU Stream Multiprocessor – High Throughput Processor

Computation Thread/Warp



Source: NVIDIA, Andreas Hehn, High Throughput with GPUs, 2018

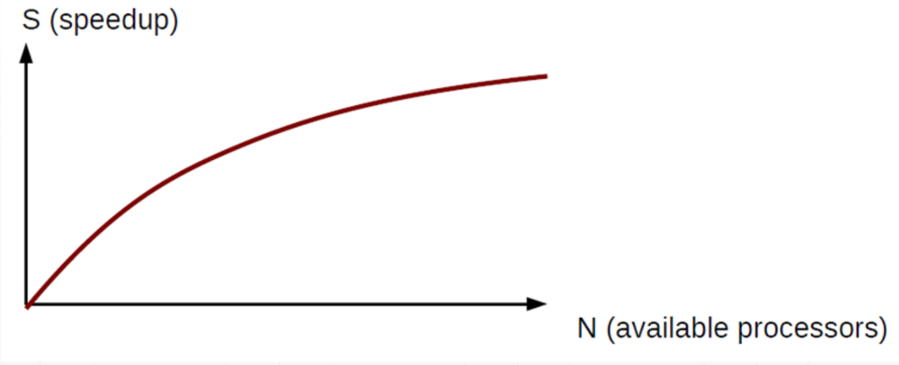
Amdahl's law

- Bounds of speed-up achievable by parallelization:
  - S – Speed-up
  - P – Ratio of parallel portions
  - N – Number of processors.

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

Time to run sequential portion

Time to run parallel portion



Source: https://en.wikipedia.org/wiki/Amdahl%27s_law

3 Rules to Rule them All … GPUs

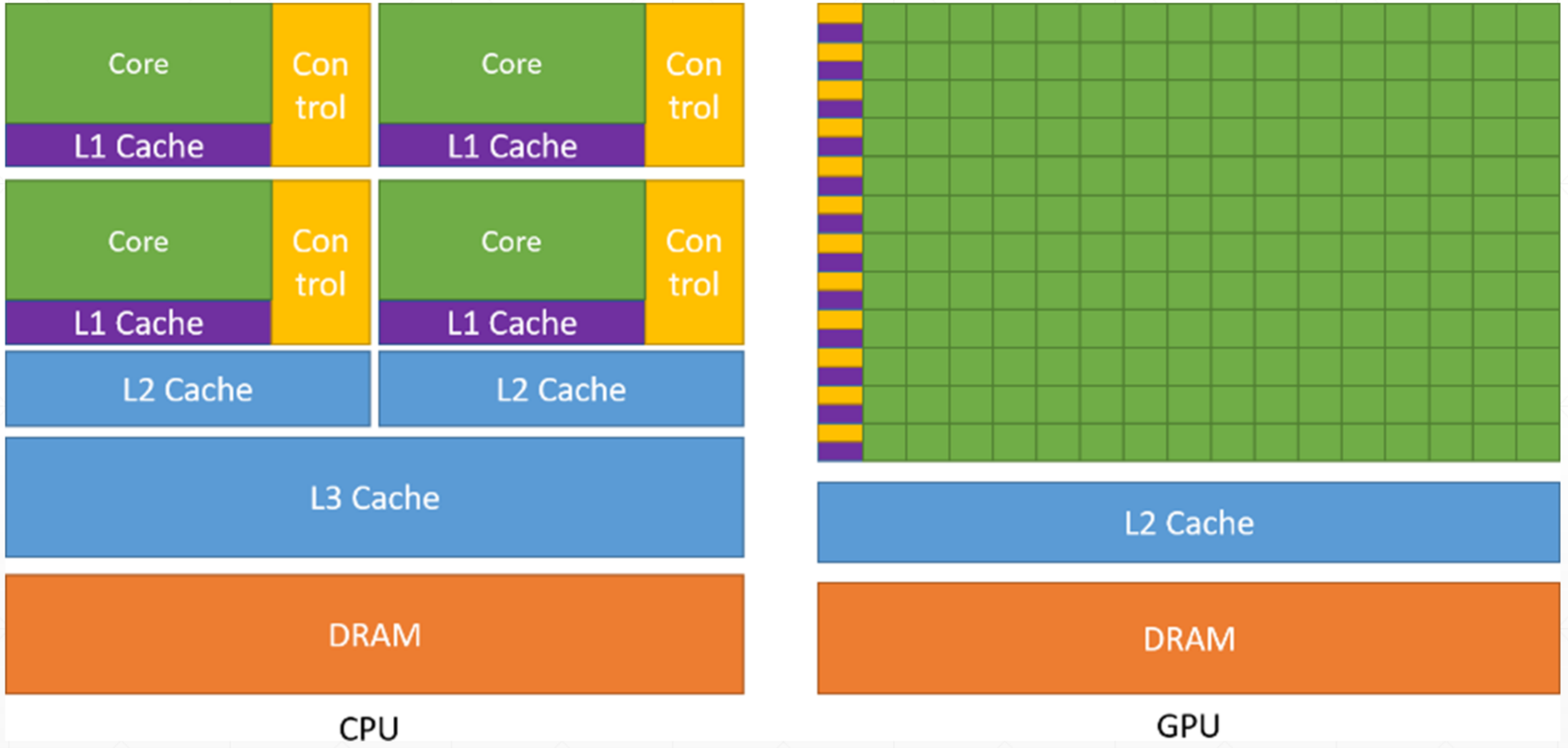
GPU PROGRAMMING FUNDAMENTALS

3 Important Rules

1. Think parallel! Feed 1000s of threads
2. Minimize and overlap CPU<->GPU transfers
3. GPU-friendly data-layout

Source: NVIDIA, Andreas Hehn, High Throughput with GPUs, 2018

Architecture CPU vs. GPU

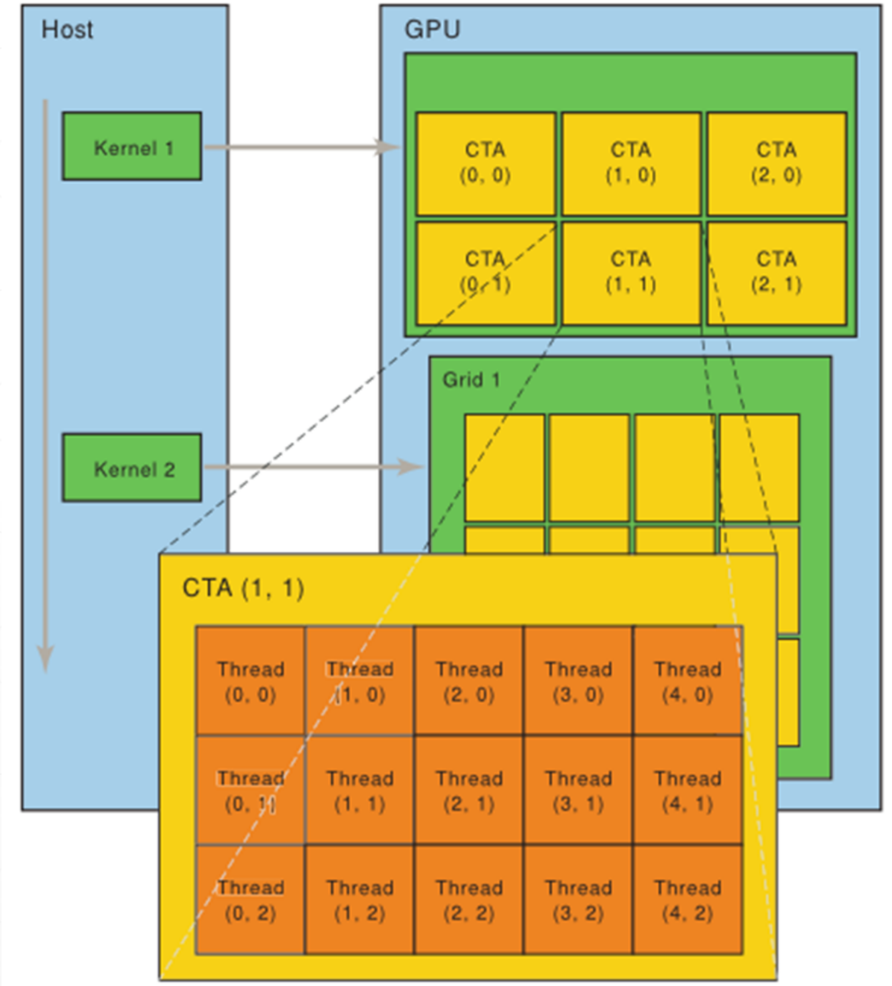


Source: https://docs.nvidia.com/cuda/

NVIDIA Streaming Multiprocessor (SM)

Source: https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

CUDA – Grid of Cooperative Thread Arrays

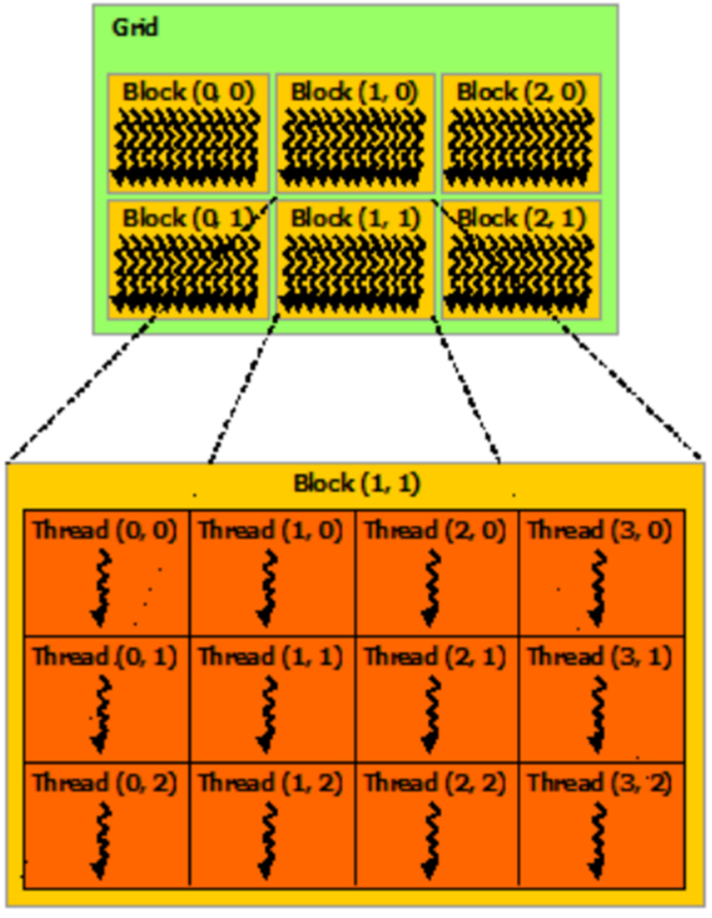


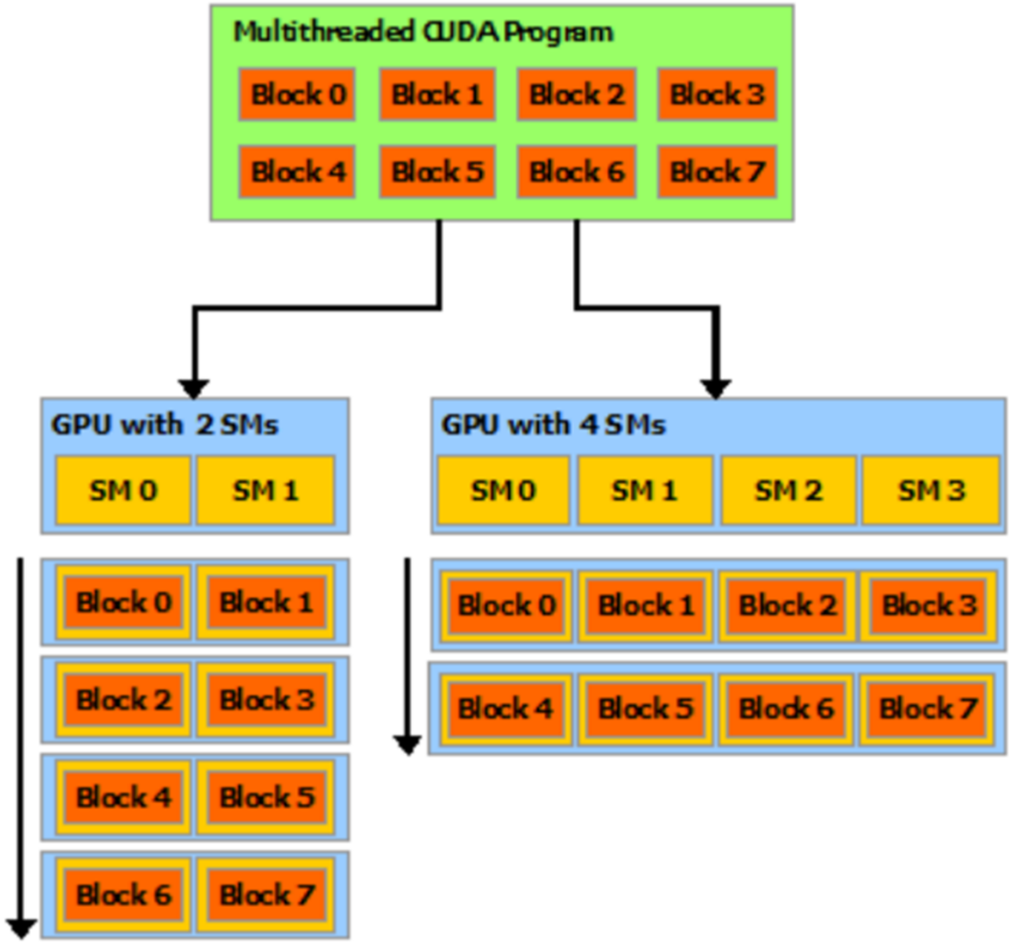
Source: https://docs.nvidia.com/cuda/

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

CUDA / GPU Automatic Scalability

Grid of Thread Blocks



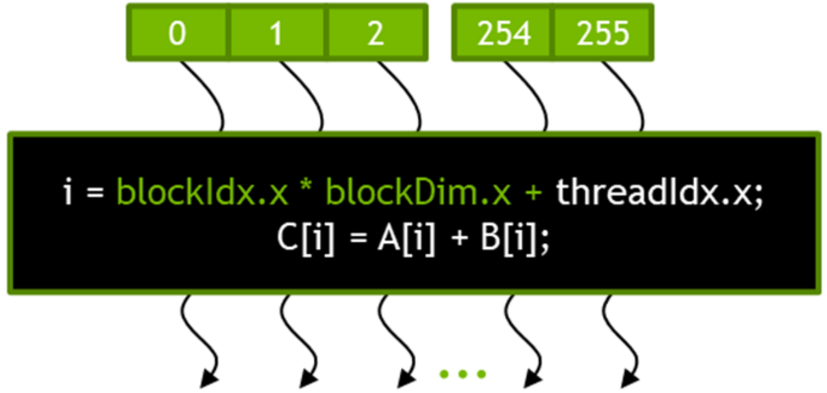


Source: https://docs.nvidia.com/cuda/

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

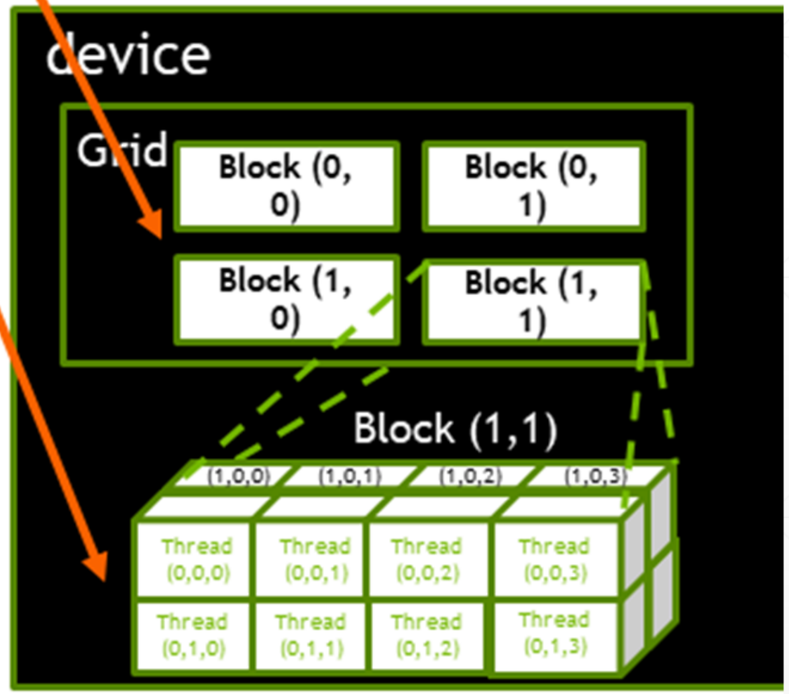
Arrays of Parallel Threads

- 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

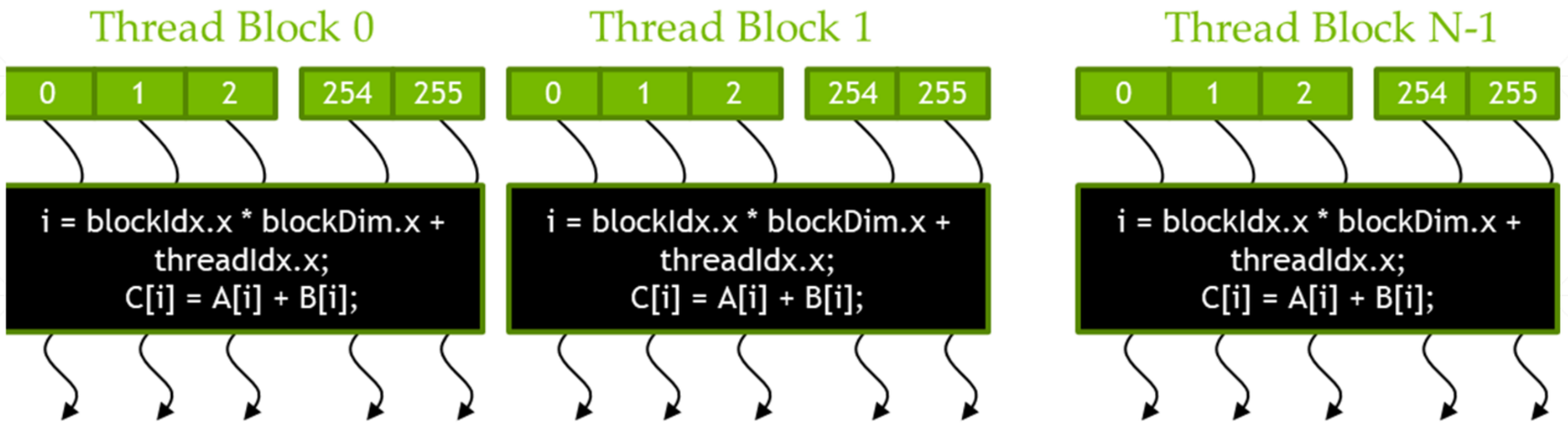


blockIdx and threadIdx

- Each thread uses indices to decide what data to work on
  - blockIdx: 1D, 2D, or 3D (CUDA 4.0)
  - threadIdx: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



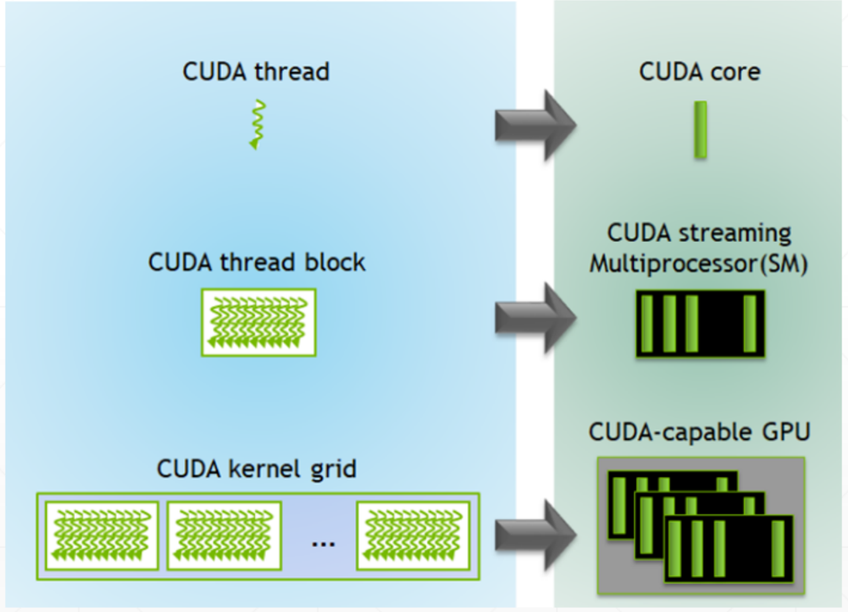
Thread Blocks: Scalable Cooperation



- 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

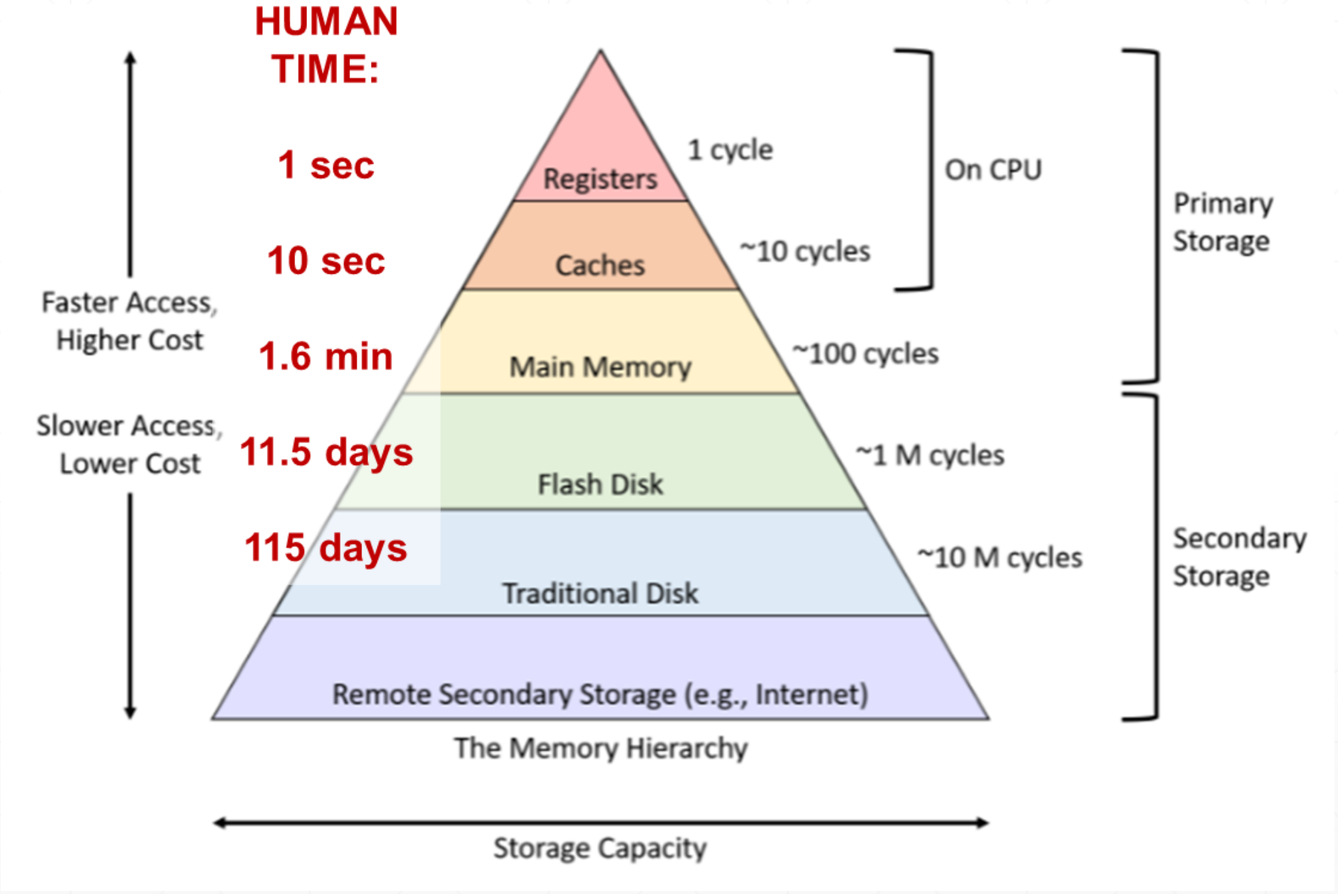
Thread Blocks / Grids

- Each CUDA block is executed by one streaming multiprocessor (SM) and cannot be migrated to other SMs in GPU.
- One SM can run several concurrent CUDA blocks depending on the resources needed by CUDA blocks.
- Each kernel is executed on one device and CUDA supports running multiple kernels on a device at one time.
- CUDA defines built-in 3D variables for threads and blocks
- CUDA architecture limits the numbers of threads per block (1024 threads per block limit).



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

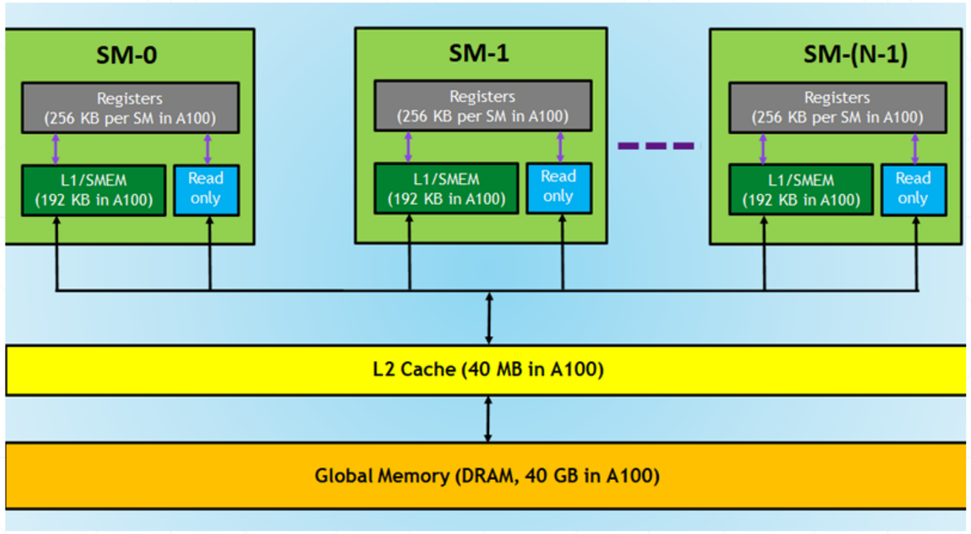
Computer Memory hierarchy

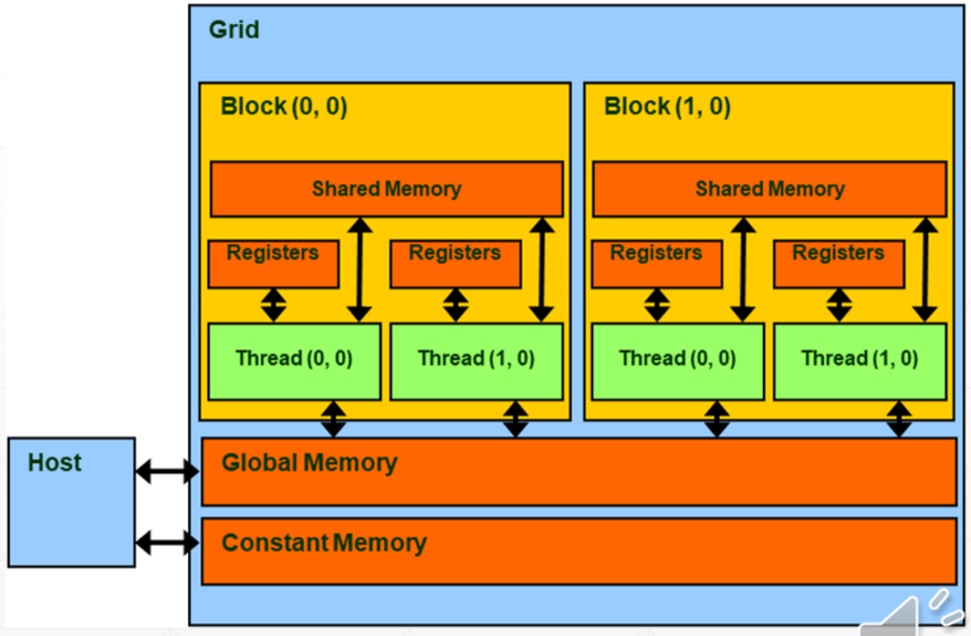


Source: https://diveintosystems.org/antora/diveintosystems/1.0/MemHierarchy/mem_hierarchy.html

GPU Memory Hierarchy

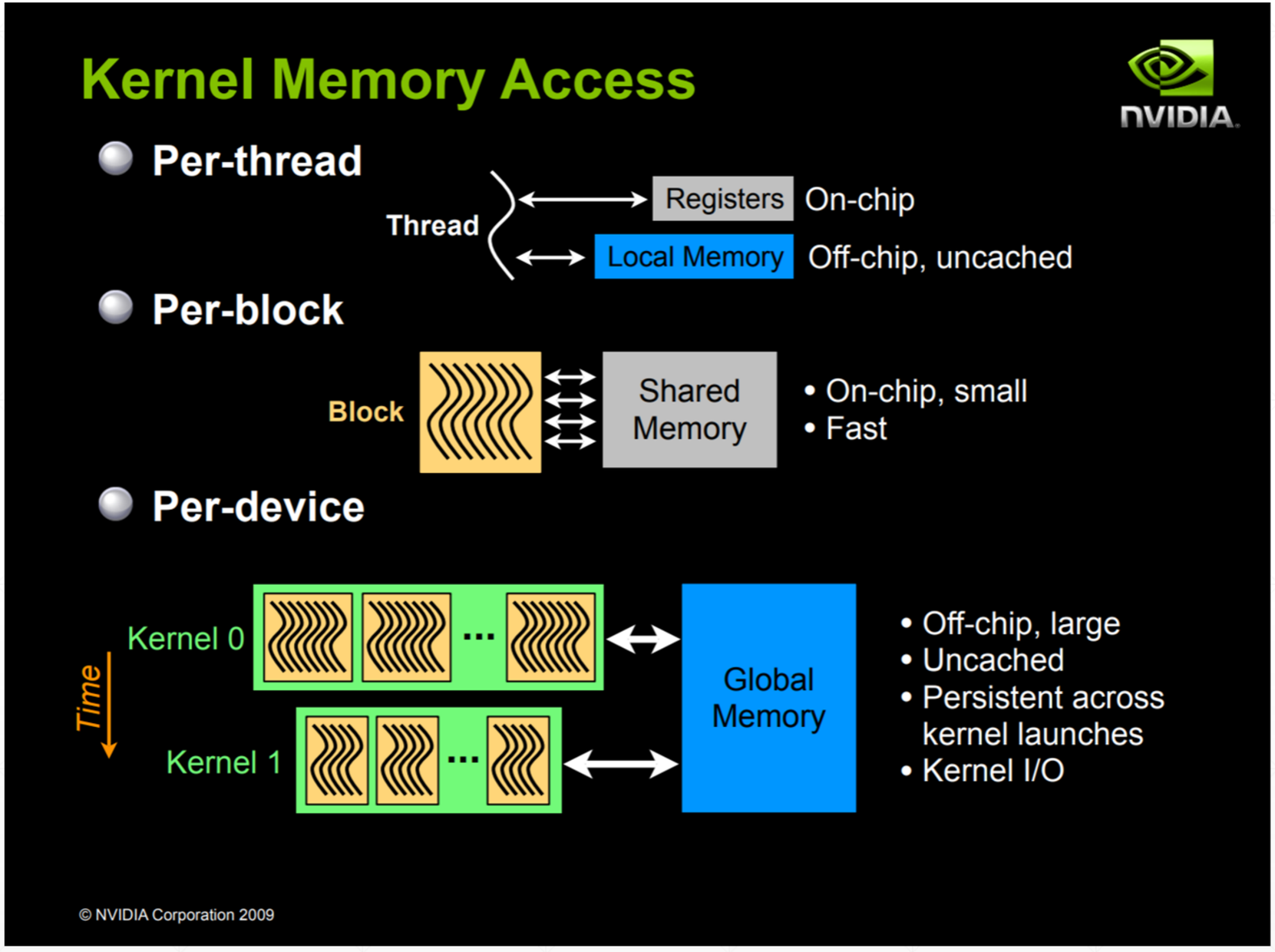
- **Registers** – These are private to each thread, which means that registers assigned to a thread are not visible to other threads. The compiler makes decisions about register utilization.
- **L1/Shared memory (SMEM)** – Every SM has a fast, on-chip scratchpad memory that can be used as L1 cache and shared memory. All threads in a CUDA block can share shared memory, and all CUDA blocks running on a given SM can share the physical memory resource provided by the SM.
- **Read-only memory** – Each SM has an instruction cache, constant memory, texture memory and RO cache, which is read-only to kernel code.
- **L2 cache** – The L2 cache is shared across all SMs, so every thread in every CUDA block can access this memory. The NVIDIA A100 GPU has increased the L2 cache size to 40 MB as compared to 6 MB in V100 GPUs.
- **Global memory** – This is the framebuffer size of the GPU and DRAM sitting in the GPU.





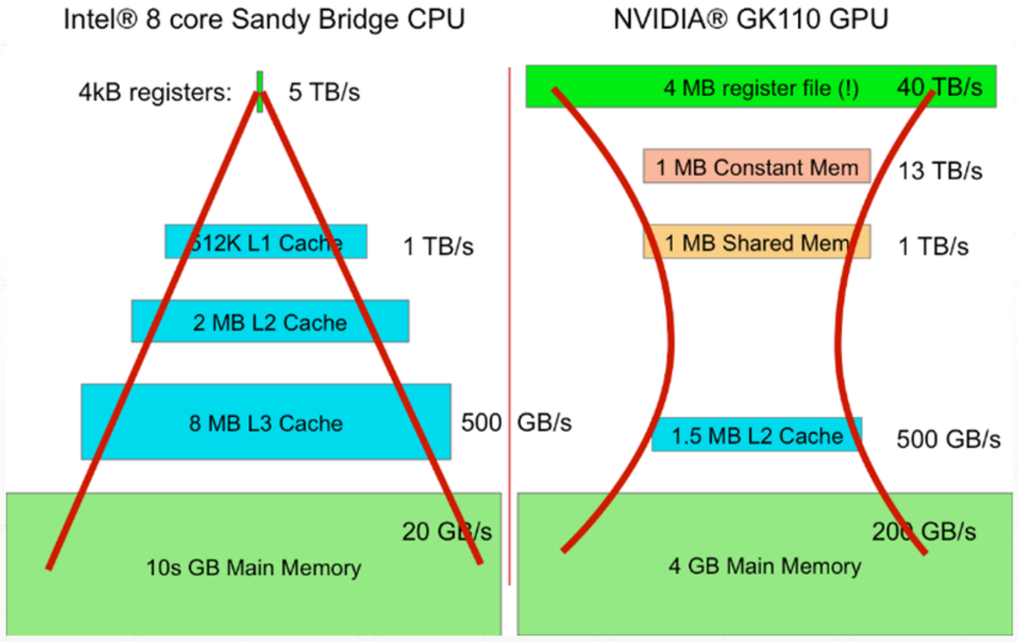
Source: https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

Kernel Memory Access



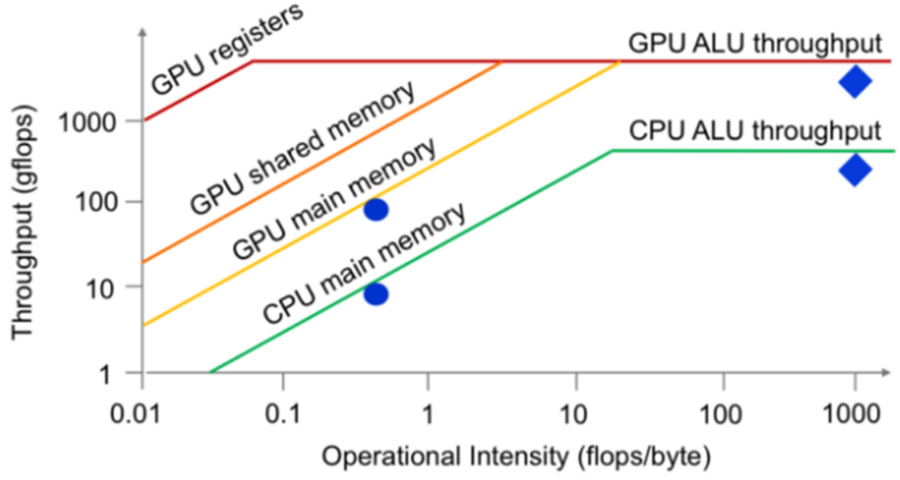
Memory / Arithmetic intensity / Performance

Where is my Memory?

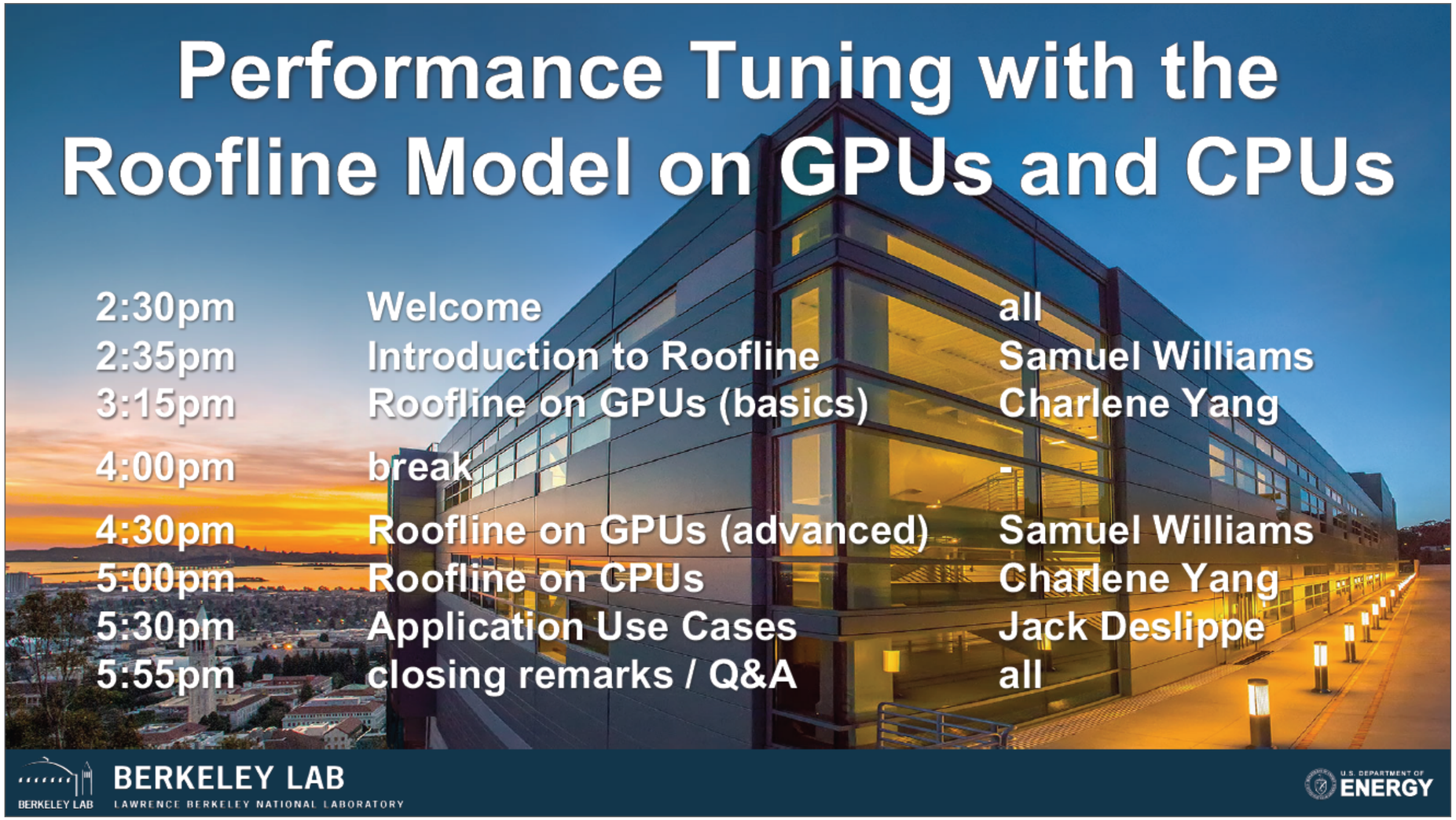


Roofline Design – Matrix kernels

- Dense matrix multiply
- Sparse matrix multiply

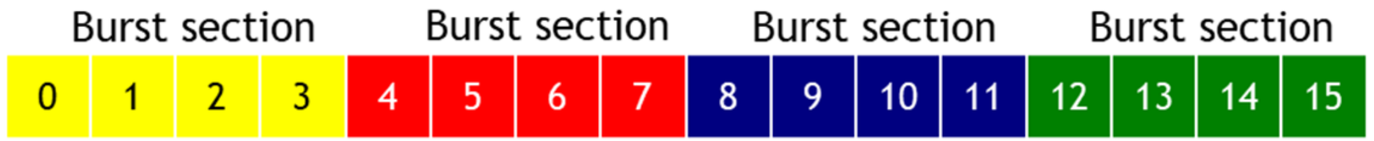


Source: https://developer.nvidia.com/blog/bidmach-machine-learning-limit-gpus/ | https://en.wikipedia.org/wiki/Roofline_model



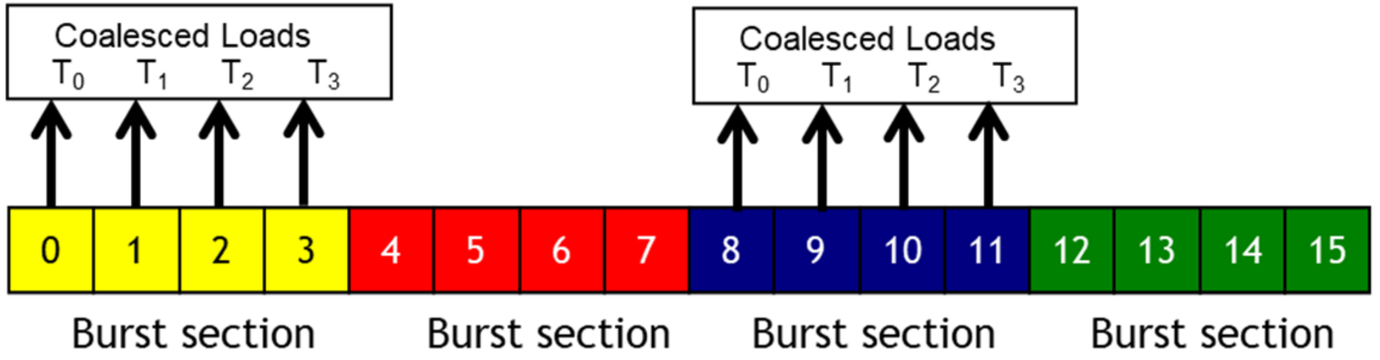
https://crd.lbl.gov/assets/Uploads/ECP20-Roofline-1-intro.pdf
https://crd.lbl.gov/assets/Uploads/ECP21-Roofline-1-intro.pdf

DRAM Burst – A System View



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

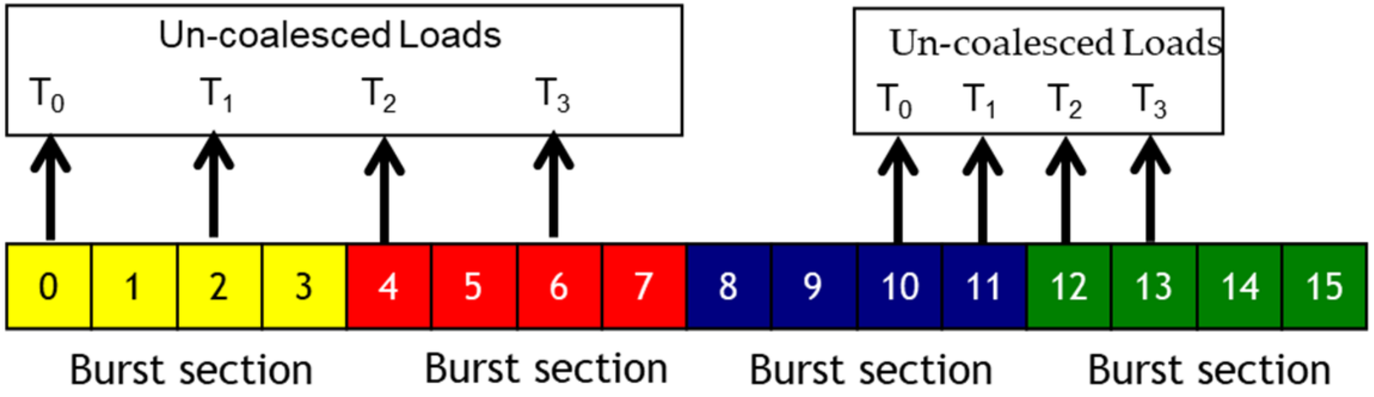
Memory Coalescing



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

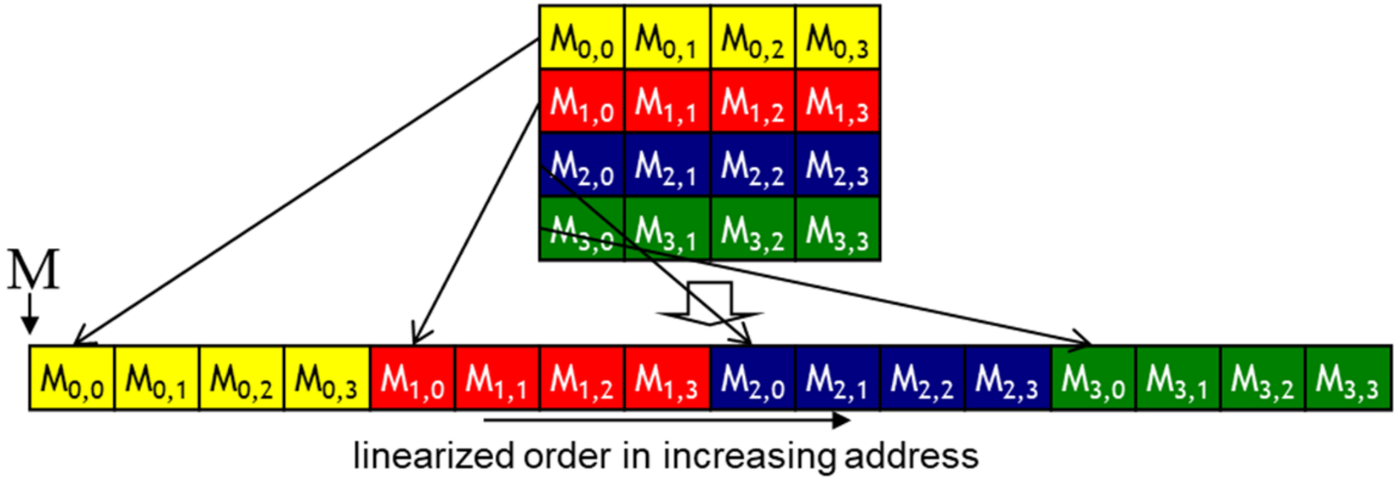
Un-coalesced Accesses

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

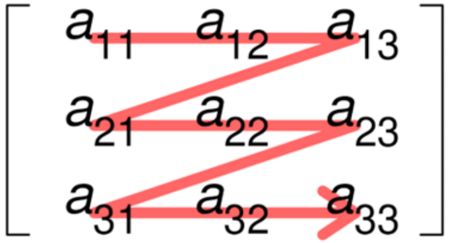


- When the accessed locations spread across burst section boundaries:
  - Coalescing fails
  - Multiple DRAM requests are made
  - The access is not fully coalesced.
- Some of the bytes accessed and transferred are not used by the threads

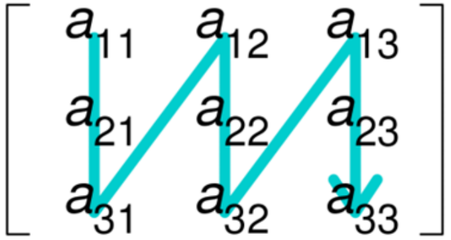
A 2D C Array in Linear Memory Space



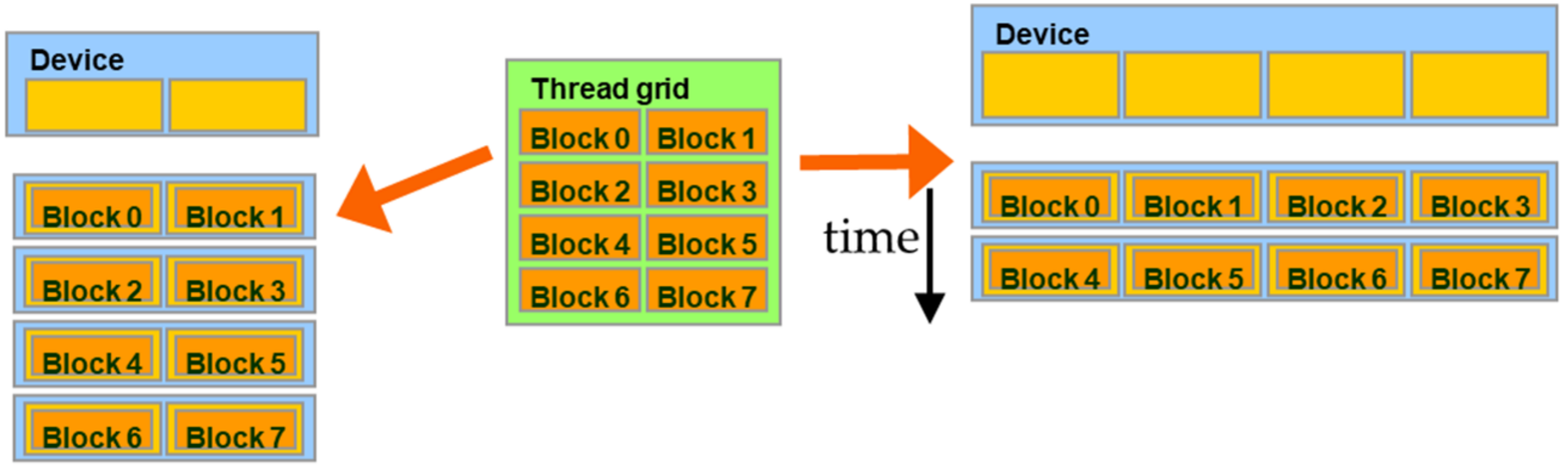
Row-major order



Column-major order

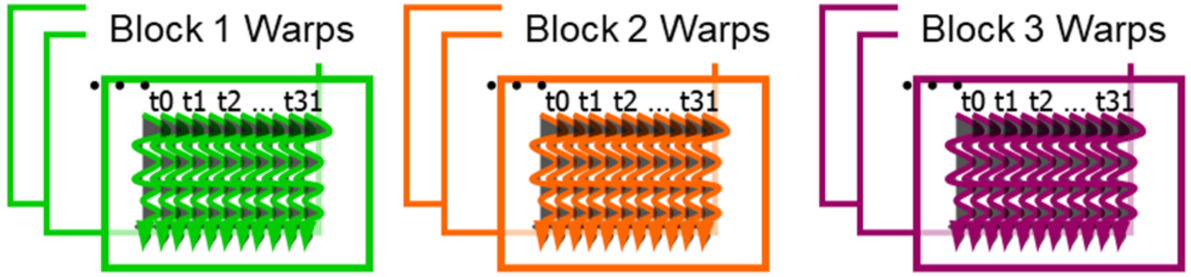


Transparent Scalability



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

Warps as Scheduling Units

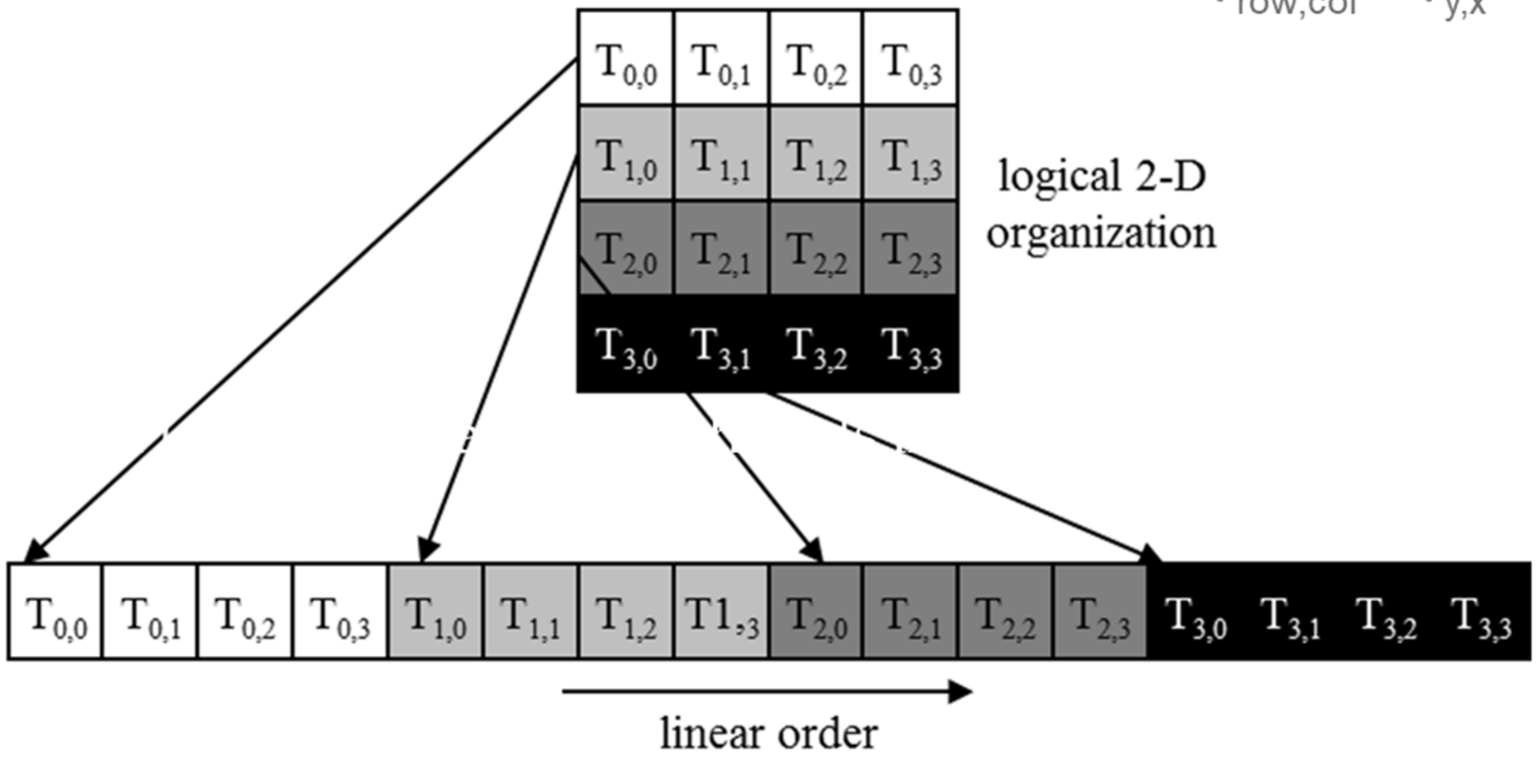


- Each block is divided into 32-thread warps
  - An implementation technique, not part of the CUDA programming model
  - Warps are scheduling units in SM
  - Threads in a warp execute in Single Instruction Multiple Data (SIMD) manner
  - The number of threads in a warp may vary in future generations

Warps in Multi-dimensional Thread Blocks

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

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



Blocks are partitioned after linearization

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

SIMD Execution Among Threads in a Warp

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

Control Divergence

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

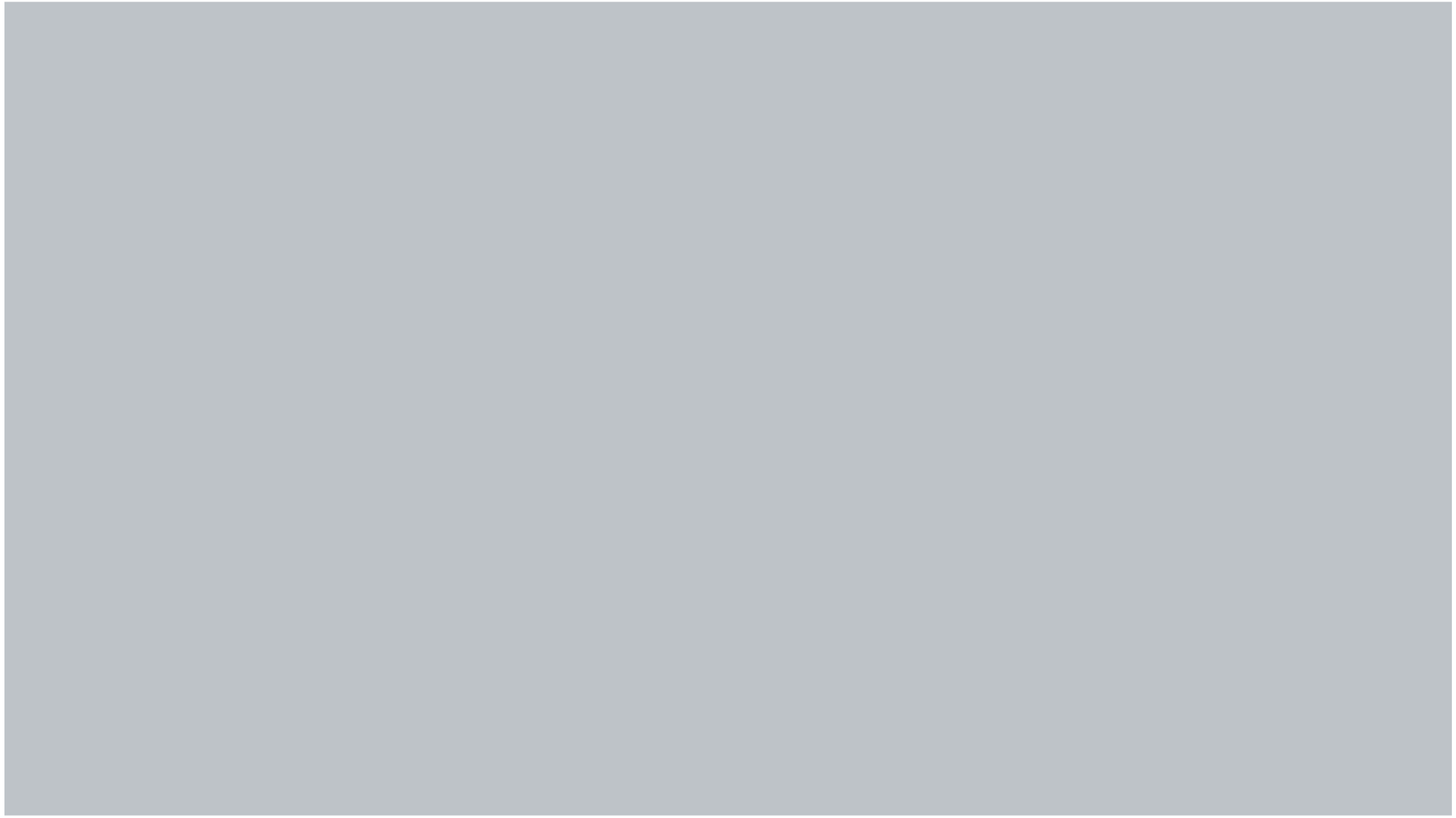
- The execution of threads taking different paths are serialized in current GPUs
  - The control paths taken by the threads in a warp are traversed one at a time until there is no more.
  - During the execution of each path, all threads taking that path will be executed in parallel
  - The number of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

CUDA TAKE-HOME MESSAGE

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

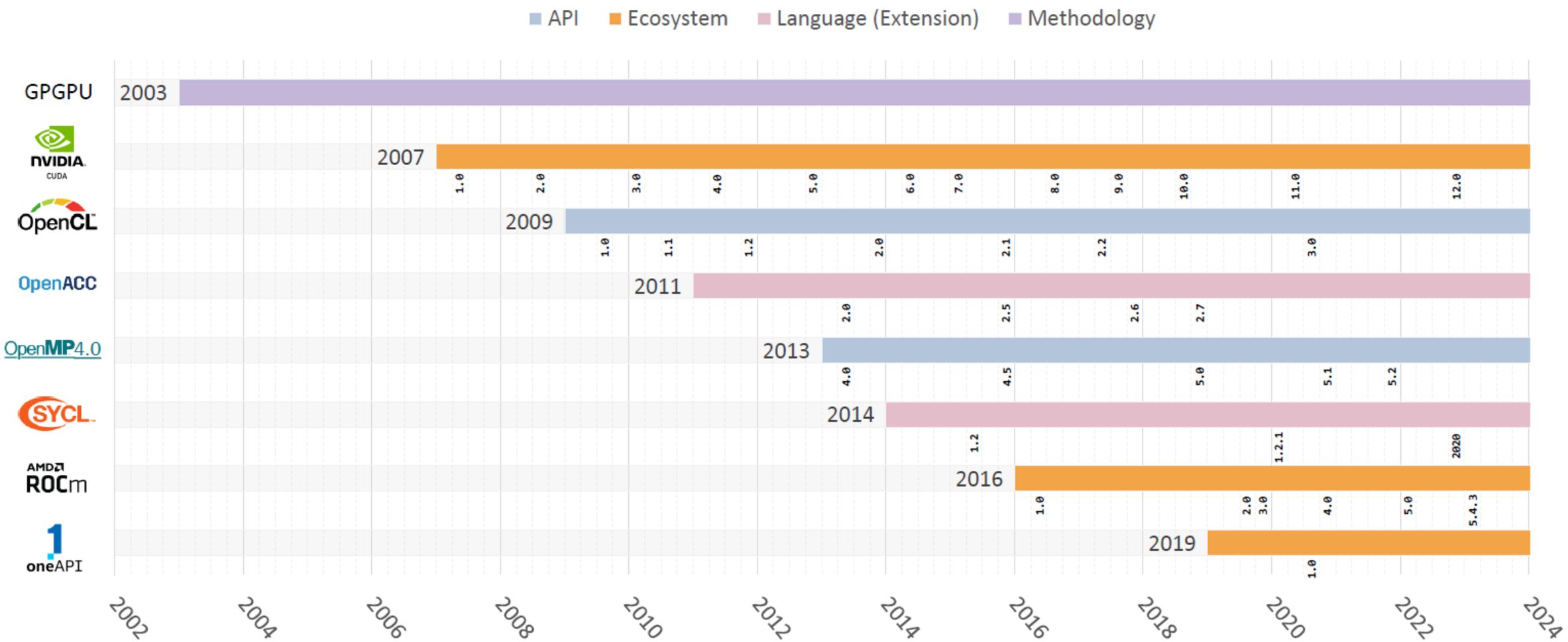
More than CUDA & NVIDIA

CUDA, ROCm, oneAPI – All for One or One for All?



sources: https://helpwiki.sharcnet.ca/wiki/Online_Seminars

GPU Computing Timeline



sources: Armin Sobhani, CUDA, ROCm, oneAPI – All for One or One for All?

NVIDIA CUDA Ecosystem

Deployment Tools	NVIDA SMI		Data Center GPU Manager (DCGM)		GPU REST Engine (GRE)
Libraries	cuBLAS	cuFFT	cuSPARSE	cuSOLVER	AMG-X
	Thrust	CUB	cuDNN	cuRAND	NCCL
Compilers & Tools	Compilers nvcc, nvc, nvc++, nvfortran	CUDA-GDB	NVIDIA Nsight	NVIDIA Visual Profiler	PAPI CUDA
Programming Models	CUDA	OpenMP API	OpenACC	OpenCL	PyCUDA
Drivers / Runtimes	*Linux* and *Windows* Device Drivers and Runtime (no macOS anymore)				

sources: Armin Sobhani, CUDA, ROCm, oneAPI – All for One or One for All?

AMD ROCm Ecosystem

Deployment Tools	ROCm SMI		ROCm Data Center Tool		ROCm Validation Suite
Libraries	rocBLAS	rocFFT	rocSPARSE	rocSOLVER	rocALLUTION
	rocThrust	rocPRIM	MIOpen	rocRAND	RCCL
Compilers & Tools	Compilers hipcc, hipfc	rocGDB	rocProfiler	hipify gpufort	TENSILE
Programming Models	HIP API		OpenMP API		OpenCL
Drivers / Runtimes	*Linux* (RedHat, SLES and Ubuntu) Device Drivers and Runtime				

Developing Applications with the AMD ROCm Ecosystem - Day 1

sources: Armin Sobhani, CUDA, ROCm, oneAPI – All for One or One for All?

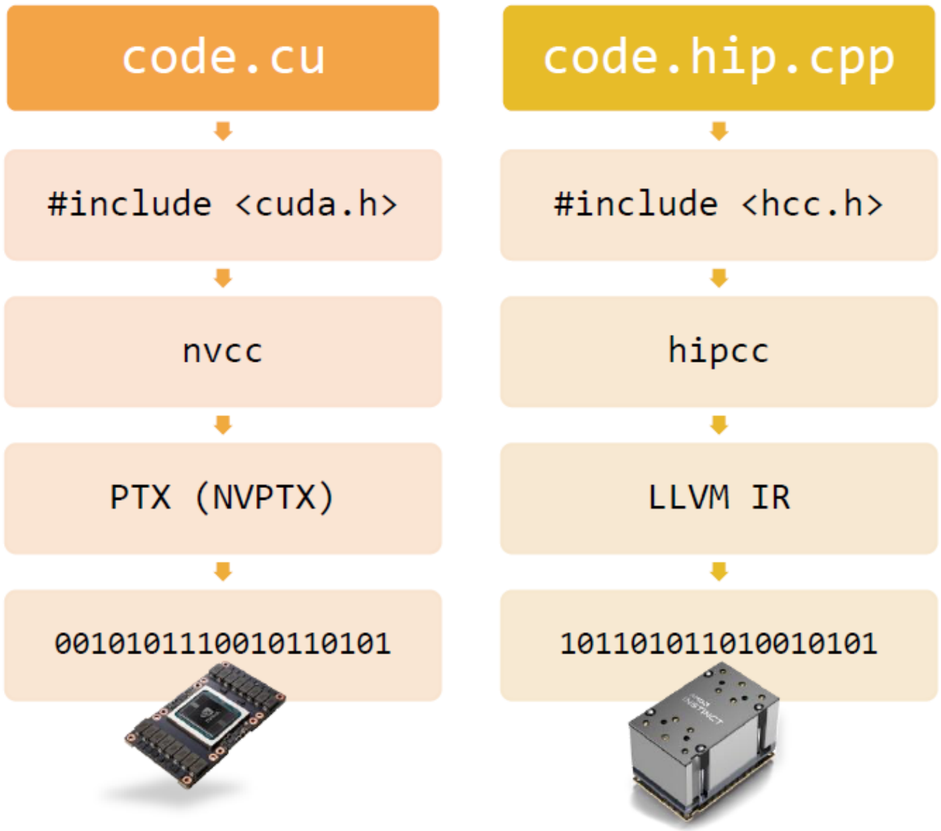
INTEL oneAPI Ecosystem

Deployment Tools	sycl-ls		Intel® Cluster Checker		Intel® MPI Library
Libraries	oneMKL	oneMKL	oneMKL	oneMKL	oneMKL
	oneDPL	oneTBB	oneDNN	oneMKL	oneCCL
Compilers & Tools	Compilers icx, icpx, ifx, dpcpp	Intel® GDB	VTune Profiler	c2s (dpct)	Intel® Inspector
Programming Models	SYCL		OpenMP API		OpenCL
Drivers / Runtimes	*Linux*, *Windows* and *macOS* Device Drivers and Runtime				

sources: Armin Sobhani, CUDA, ROCm, oneAPI – All for One or One for All?

Compile Time vs. Run Time

Compile-time (CUDA / ROCm)



Run-time (oneAPI / SYCL / OpenCL)

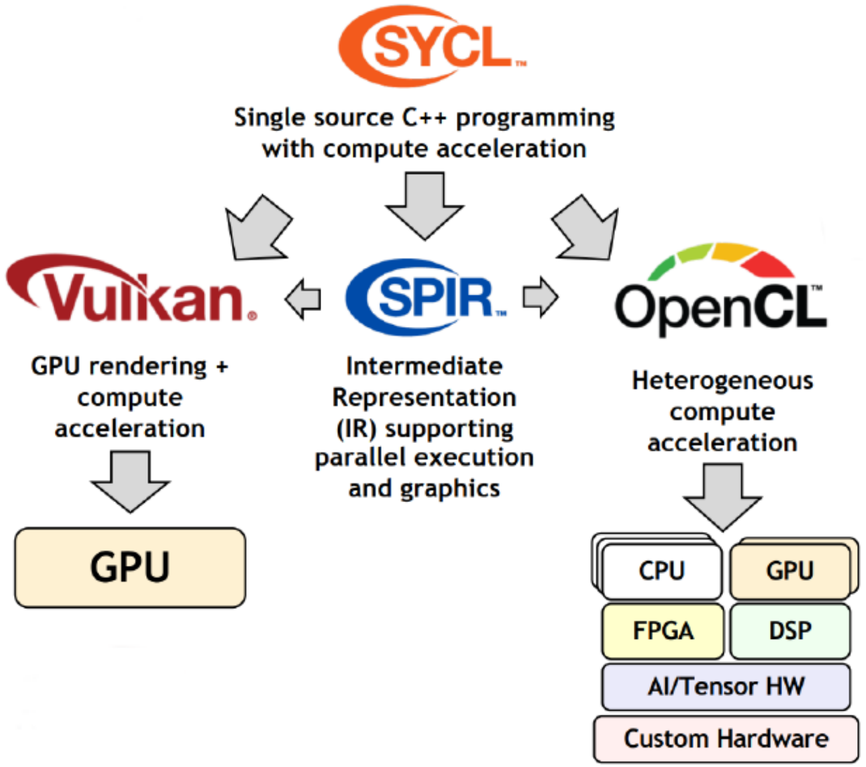


Image courtesy of khronos.org

sources: Armin Sobhani, CUDA, ROCm, oneAPI – All for One or One for All?

Syntax ‘Cheat Sheet’

TERM	CUDA	ROCm	OpenCL	SYCL
Device	int deviceId	int deviceId	cl_device	sycl::device
Queue	cudaStream_t	hipStream_t	cl_command_queue	sycl::queue
Event	cudaEvent_t	hipEvent_t	cl_event	sycl::event
Memory	void*	void*	cl_mem	sycl::buffer
Grid of threads	grid	grid	NDRange	NDRange
Subgroup of threads	block	block	work-group	work-group
Thread	thread	thread	work-item	work-item
Scheduled execution	warp	Warp	sub-group (warp, wavefront, etc.)	sub-group (warp, wavefront, etc.)
Thread-index	threadIdx.x	hipThreadIdx_x, threadIdx.x*	get_local_id(0)	sycl::nd_item::get_local_id(0)
Block-index	blockIdx.x	hipBlockIdx_x, blockIdx.x*	get_group_id(0)	sycl::nd_item::get_group(0)
Block-dim	blockDim.x	hipBlockDim_x, blockDim.x*	get_local_size(0)	sycl::nd_item::get_local_range(0)
Grid-dim	gridDim.x	hipGridDim_x, gridDim.x*	get_global_size(0)	sycl::nd_item::get_global_range(0)
Device Kernel	__global__	__global__	__kernel__	C++ lambda, sycl::kernel
Device Function	__device__	__device__	N/A. Implied in device compilation	N/A. Implied in device compilation
Host Function	__host__ (default)	__host__ (default)	N/A. Implied in host compilation	N/A. Implied in host compilation
Host + Device Function	__host__ __device__	__host__ __device__	N/A	N/A
Kernel Launch	<<< >>>	hipLaunchKernel, <<< >>>*	clEnqueueNDRangeKernel	sycl::queue::submit()
Global Memory	__global__	__global__	__global	__global
Group Memory	__shared__	__shared__	__local	__local
Private Memory	(default)	(default)	__private	__private
Constant	__constant__	__constant__	__constant	__constant
Thread Synchronisation	__syncthreads	__syncthreads	barrier(CLK_LOCAL_MEMFENCE)	sycl::queue::wait()
Precise Math	cos(f)	cos(f)	cos(f)	cos(f)
Fast Math	__cos(f)	__cos(f)	native_cos(f)	native_cos(f)

sources: Armin Sobhani, CUDA, ROCm, oneAPI – All for One or One for All?

* Since ROCm 3.5

Library 'Cheat Sheet'

CUDA Library	ROCm Library	oneAPI Library	Description
cuBLAS	rocBLAS	oneMKL	Basic Linear Algebra Subroutines
cuFFT	rocFFT	oneMKL	Fast Fourier Transfer Library
cuSPARSE	rocSPARSE	oneMKL	Sparse BLAS + SPMV
cuSolver	rocSOLVER	oneMKL	Lapack library
AMG-X	rocALUTION	oneMKL	Sparse iterative solvers and preconditioners with Geometric and Algebraic MultiGrid
Thrust	rocThrust	oneDPL	C++ parallel algorithms library
CUB	rocPRIM	oneTBB	Low Level Optimized Parallel Primitives
cuDNN	MIOpen	oneDNN	Deep learning Solver Library
cuRAND	rocRAND	oneMKL	Random Number Generator Library
EIGEN	EIGEN – HIP port	oneMKL	C++ template library for linear algebra: matrices, vectors, numerical solvers
NCCL	RCCL	oneCCL	Communications Primitives Library based on the MPI equivalents

sources: Armin Sobhani, CUDA, ROCm, oneAPI – All for One or One for All?