IEEE UFFC-JS 2024
IEEE Ultrasonics, Ferroelectrics, and Frequency Control Joint Symposium
Taipei Nangang Exhibition Center, Taipei, Taiwan || September 22 – 26, 2024

CUDA refresher

Short-course: Ultrasound Signal Processing with GPUs

Marcin Lewandowski, CEO
marcin@us4us.eu

LAB 4US

License / Attribution

DISCLAIMER

- 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.

WELCOME!

Ultrasound Signal Processing with GPUs — Introduction to Parallel Programming

IEEE UFFC-JS 2024
IEEE Ultrasonics, Ferroelectrics, and Frequency Control Joint Symposium
Taipei Nangang Exhibition Center, Taipei, Taiwan || September 22 – 26, 2024

us4us®
LAB 4US

UNIVERSITY OF WATERLOO
IPPT PAN
DTU

- **This is our 4rd 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 Y. S. Yiu / yshyi@dtu.dk> / DTU

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

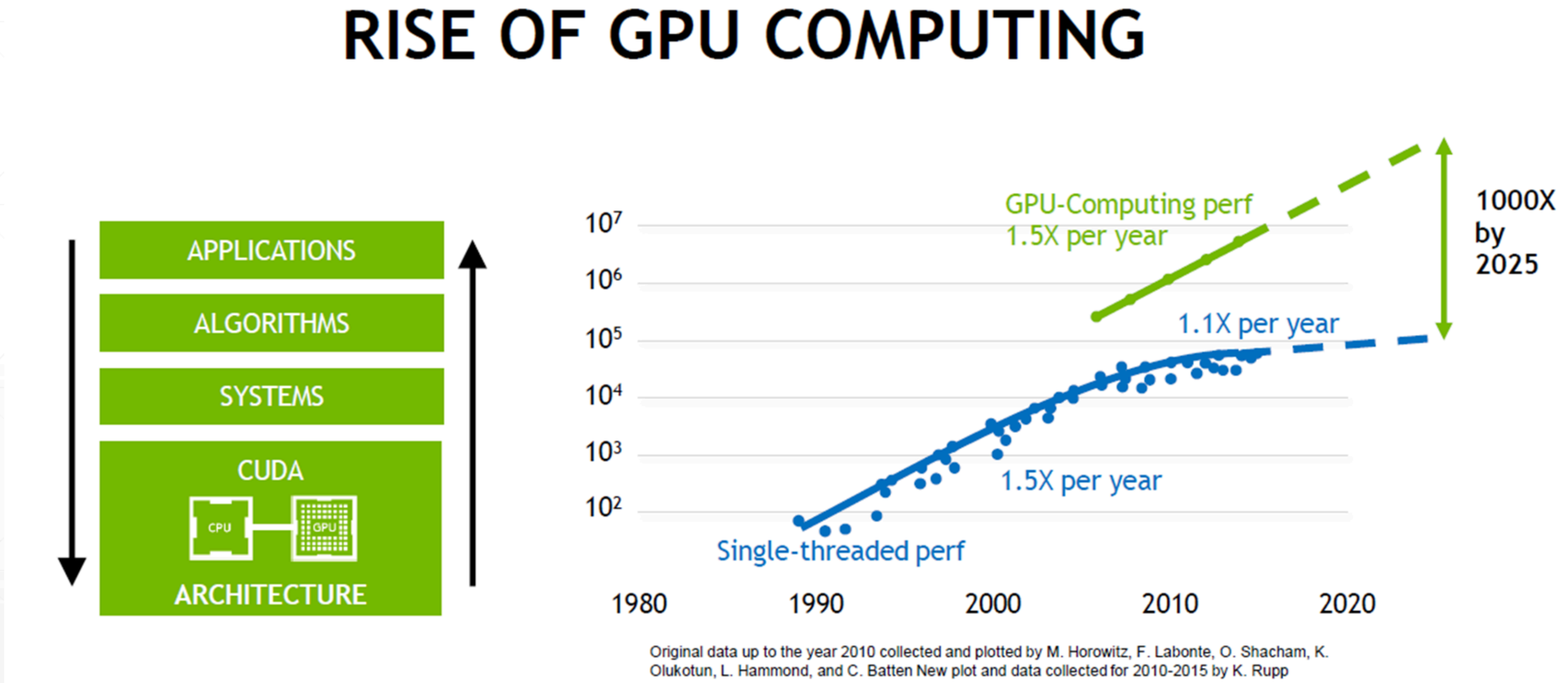
https://vimeo.com/showcase/2022-us-gpu-short-course

PLAN

- **GPU/CUDA ... (MARCIN)**
  - NVIDIA CUDA Refresher
  - Intro to NVIDIA Holoscan
- **US Beamforming Algorithms (PIOTR)**
  - DAS (Delay & Sum) as Interpolation and Reduction
  - CUDA streams, graphs, and NVIDIA Holoscan SDK
- **Speed-of-Sound Estimatin + Deep Learning on GPU (BILLY)**

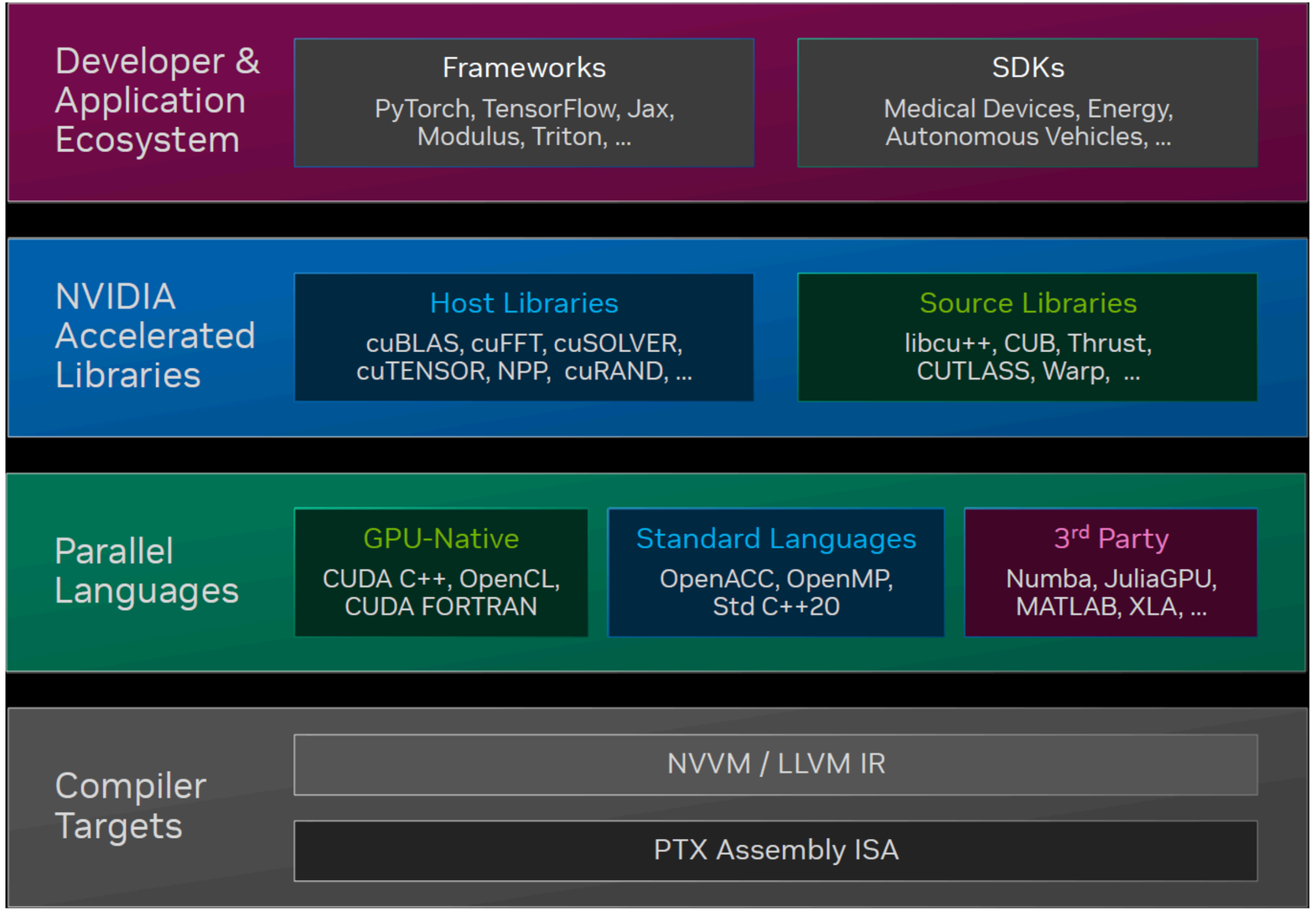
NVIDA CUDA Refresher

Why GPUs?



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

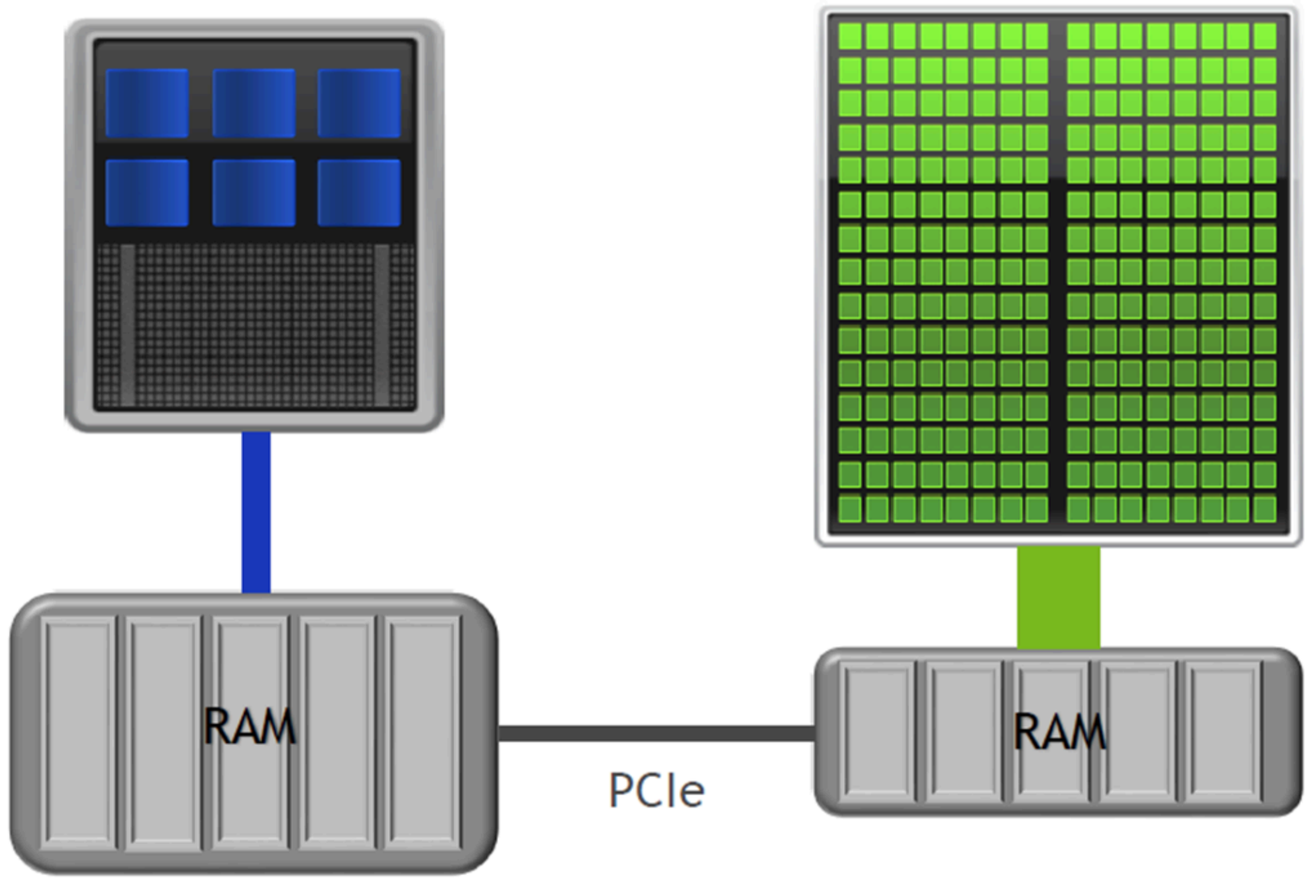
CUDA is more



source: Stephen Jones, How To Write A CUDA Program, GTC Fall 2023

CPU & GPU – heterogeneous computing

CPU
Optimized for
Serial Tasks

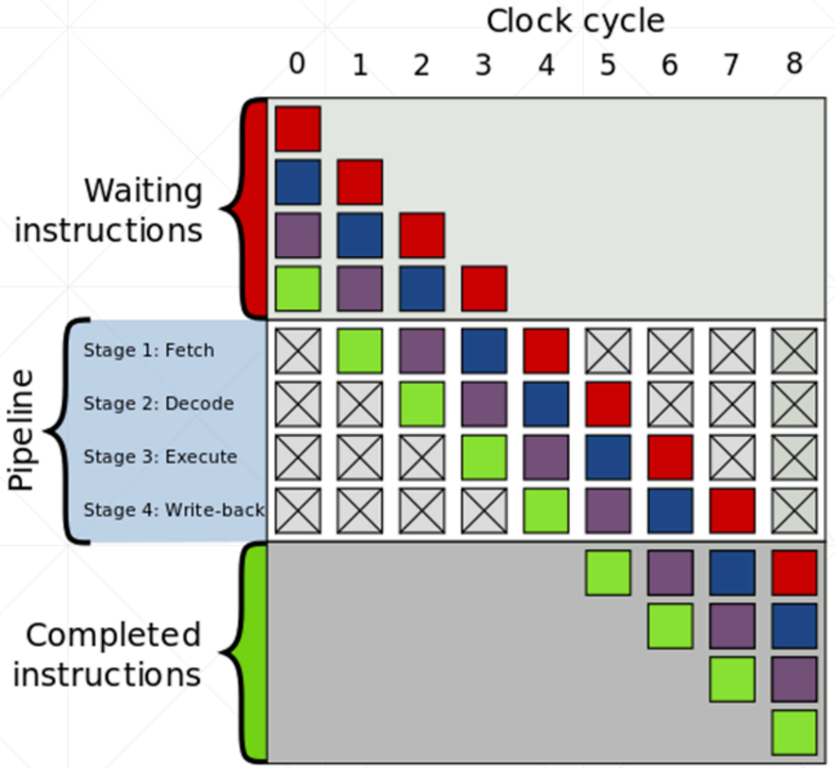


GPU
Optimized for
Parallel Tasks

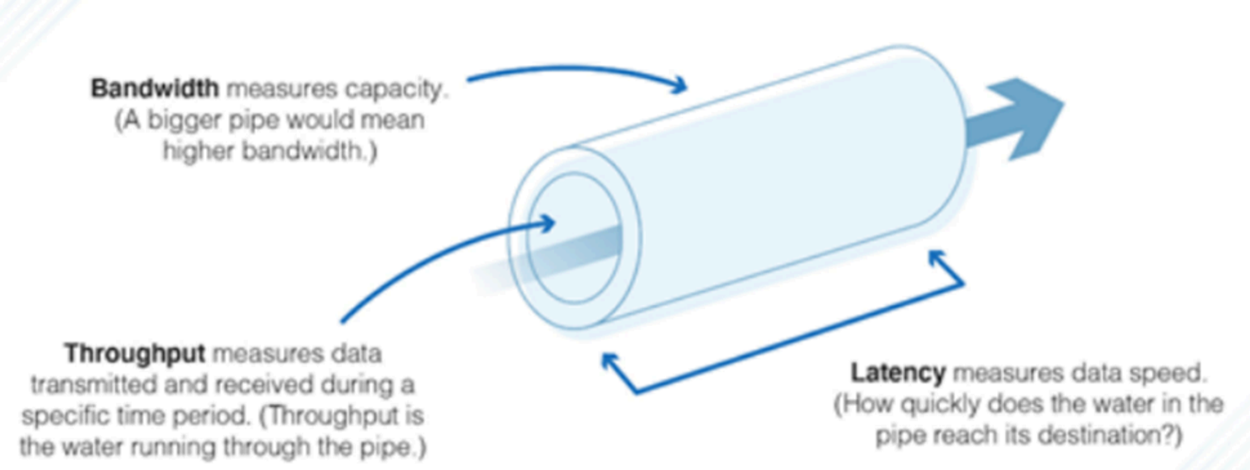
Source: NVIDIA, Andreas Hehn, High Throughput with GPUs, 2018

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



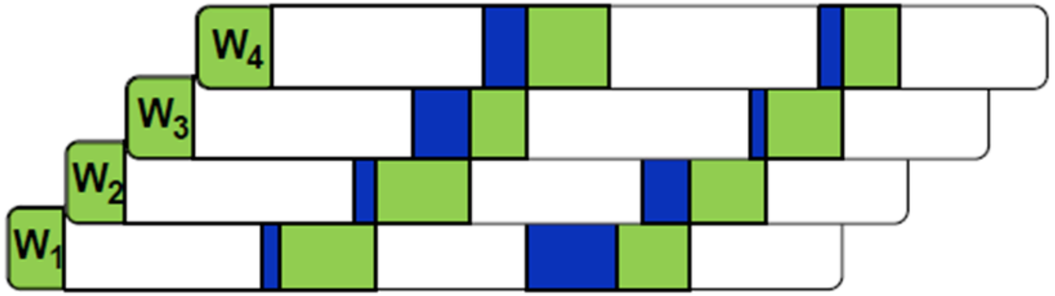
Source: https://www.dnsstuff.com/latency-throughput-bandwidth | https://en.wikipedia.org/wiki/Instruction_pipelining

LOW LATENCY OF HIGH THROUGHPUT?

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

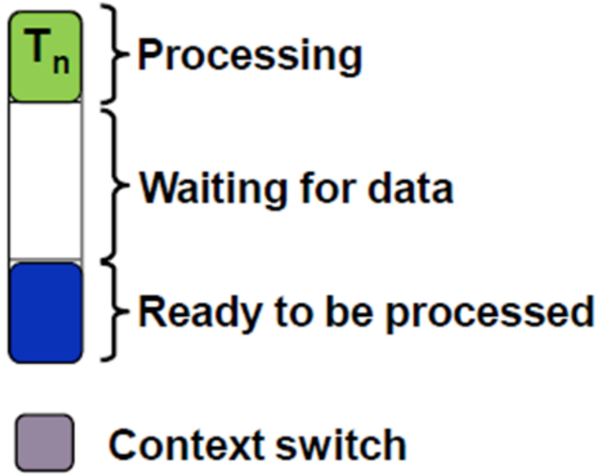
CPU core – Low Latency Processor

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

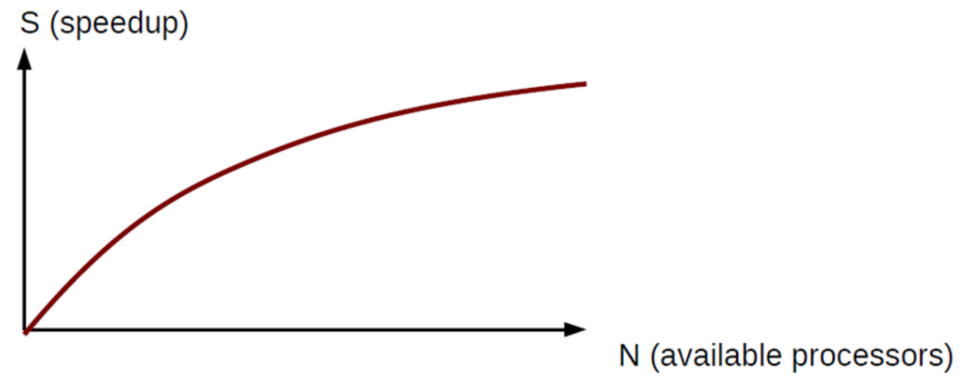
Amdahl's law

- 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

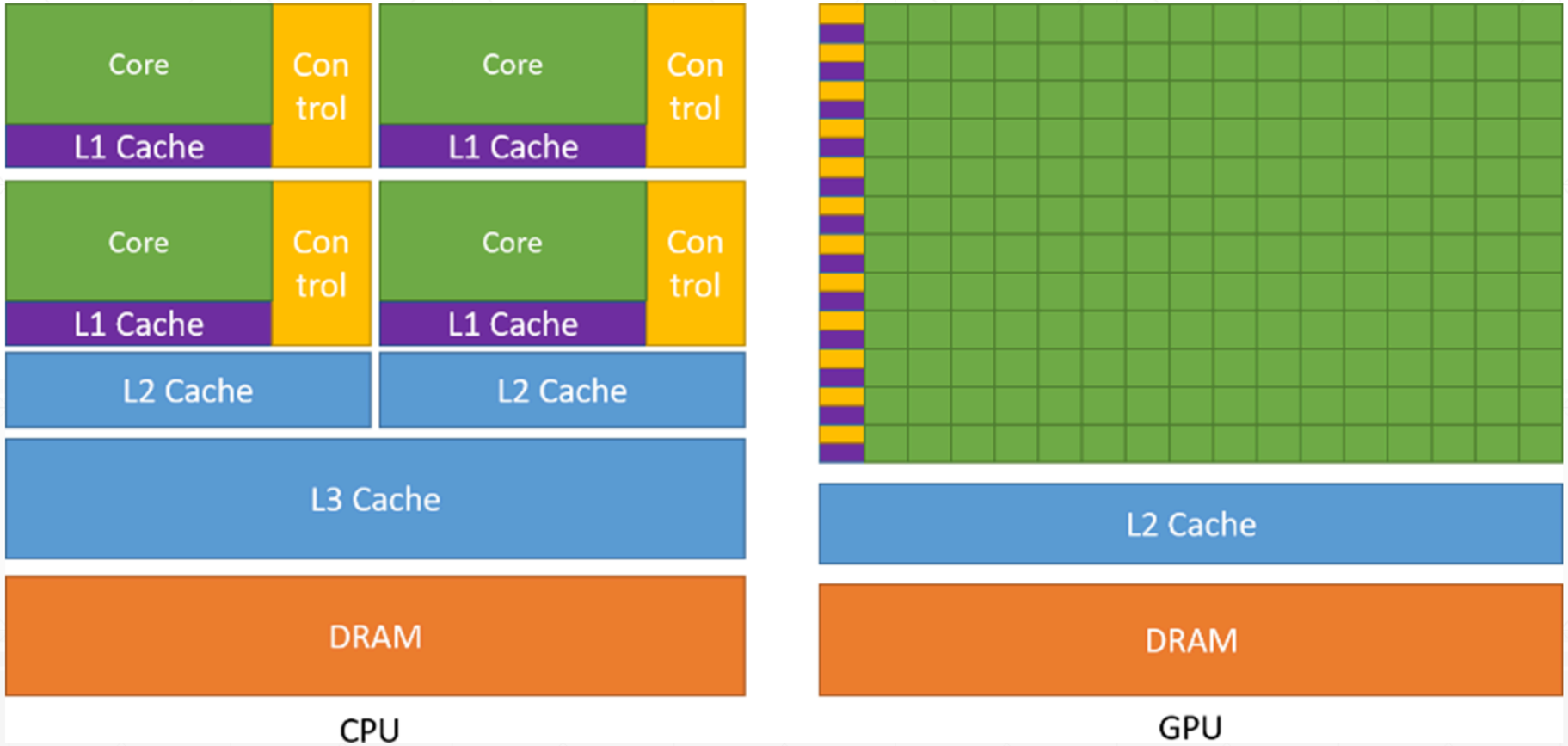
GPU PROGRAMMING FUNDAMENTALS

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

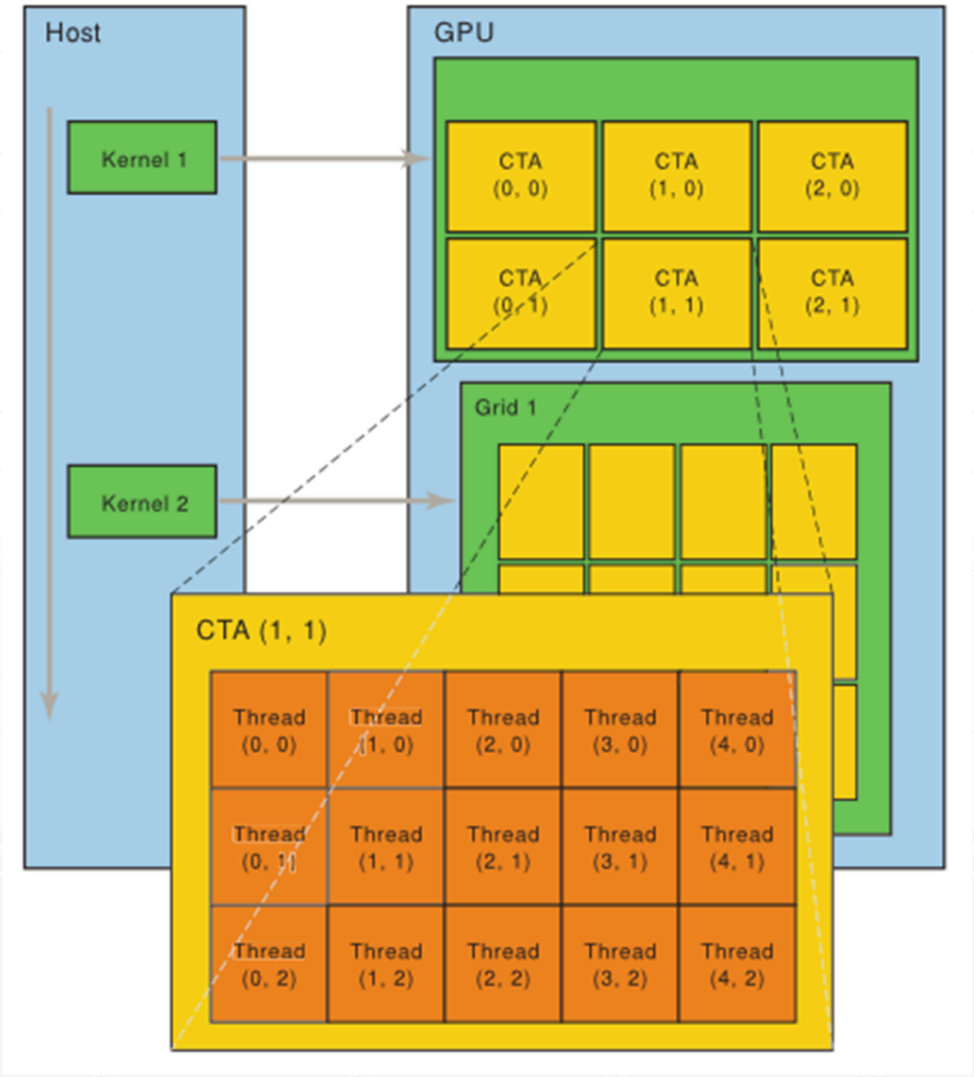


Source: https://docs.nvidia.com/cuda/

NVIDIA Streaming Multiprocessor (SM)

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

CUDA – Grid of Cooperative Thread Arrays

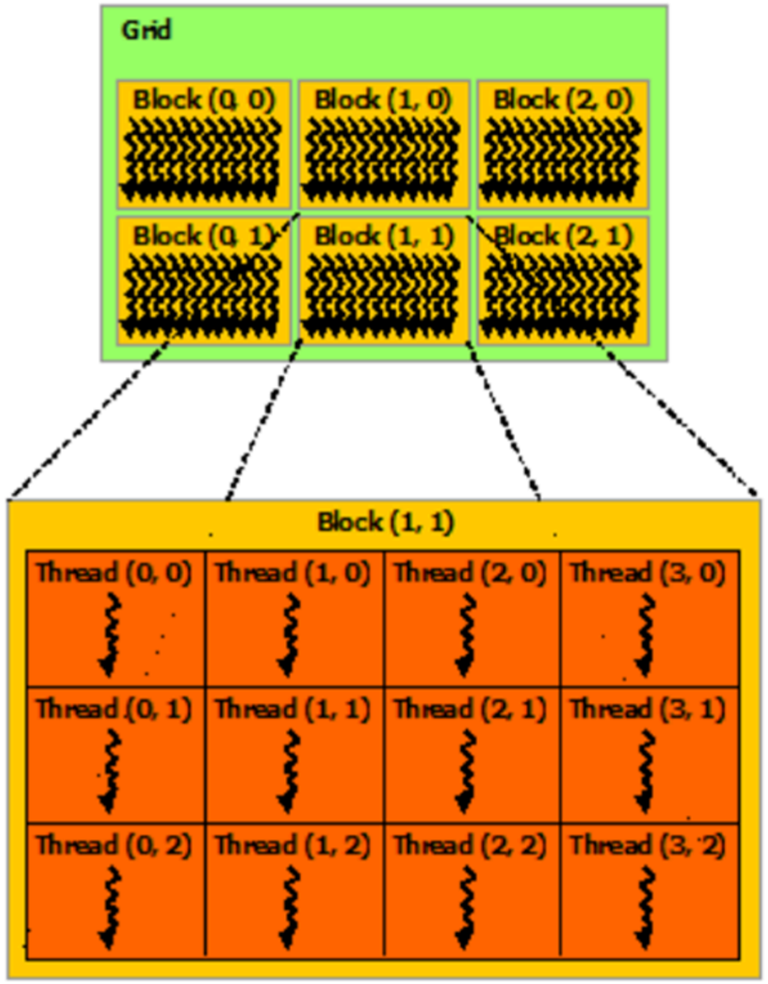


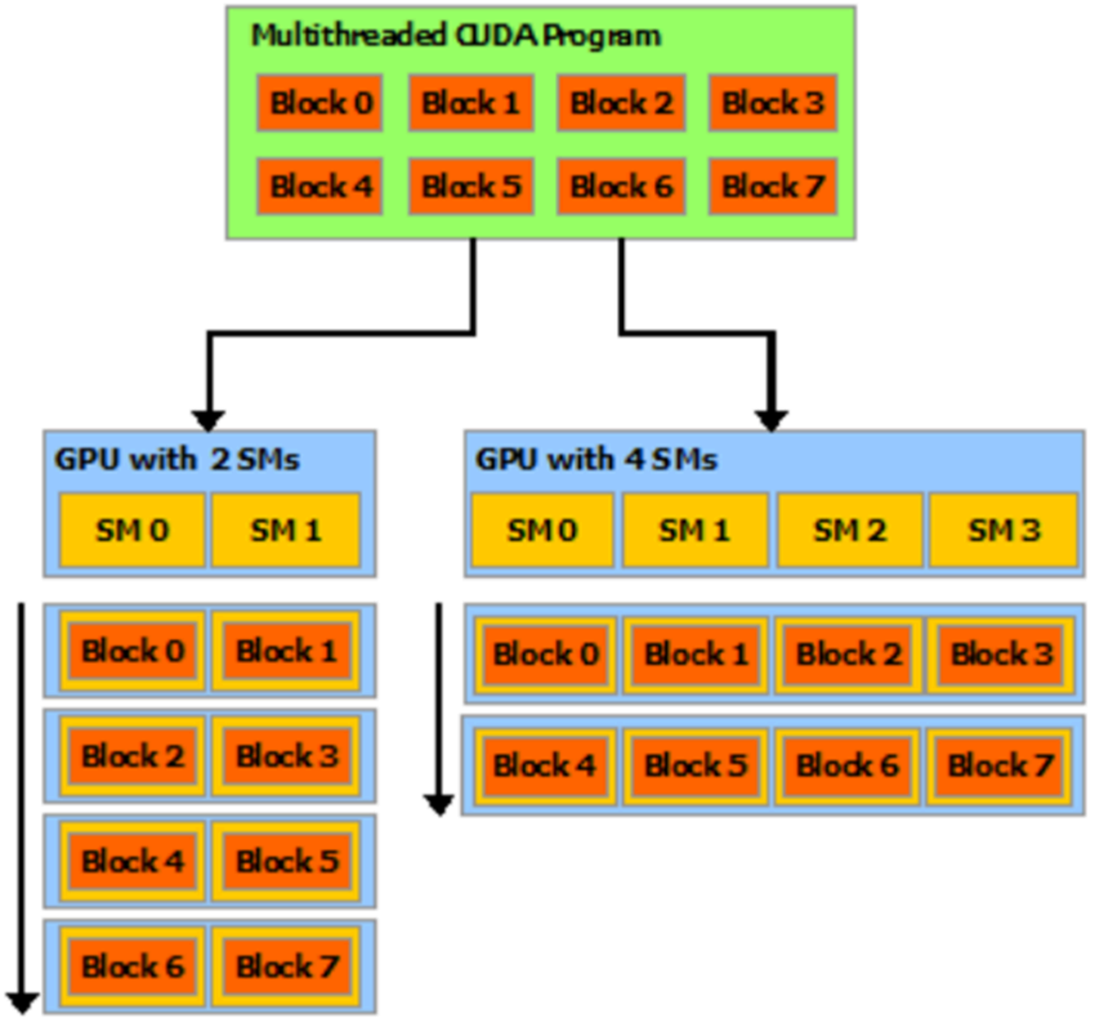
Source: https://docs.nvidia.com/cuda/

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

CUDA / GPU Automatic Scalability

Grid of Thread Blocks



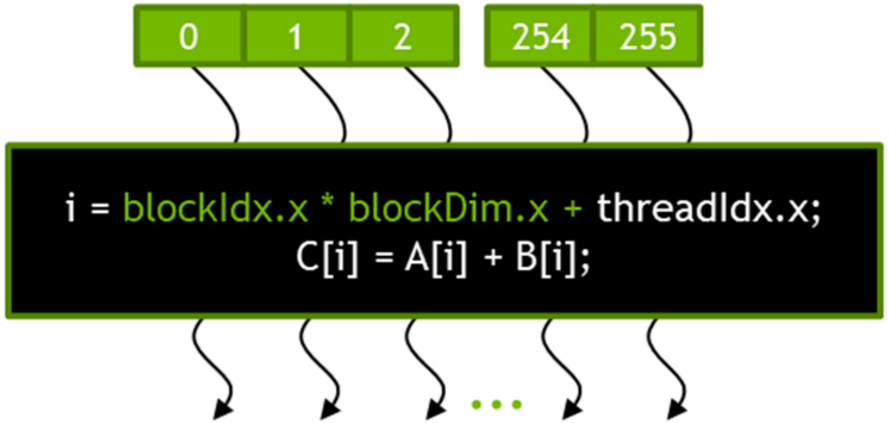


Source: https://docs.nvidia.com/cuda/

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

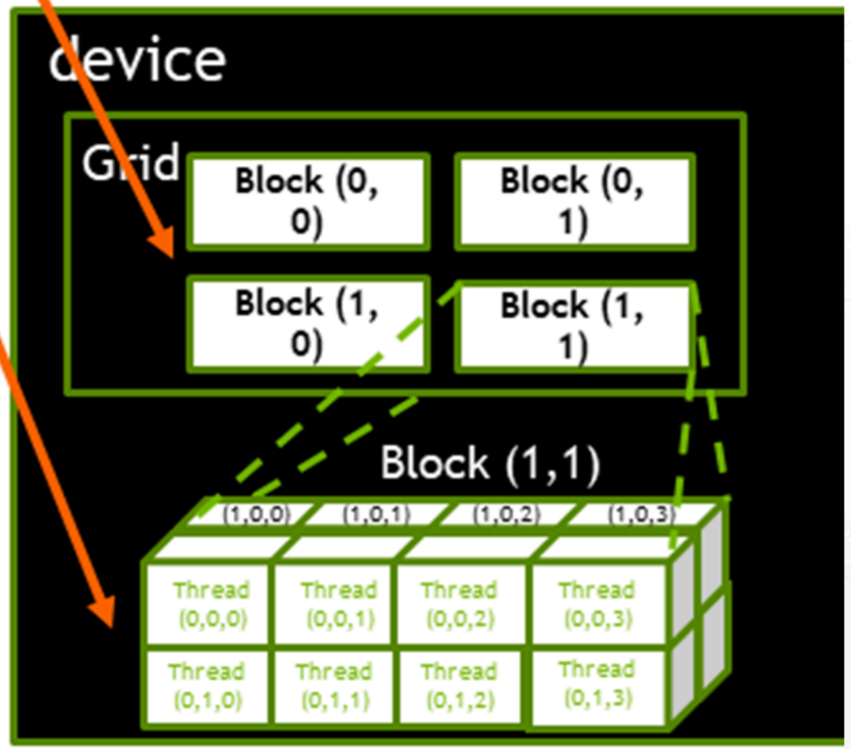
Arrays of Parallel Threads

- 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

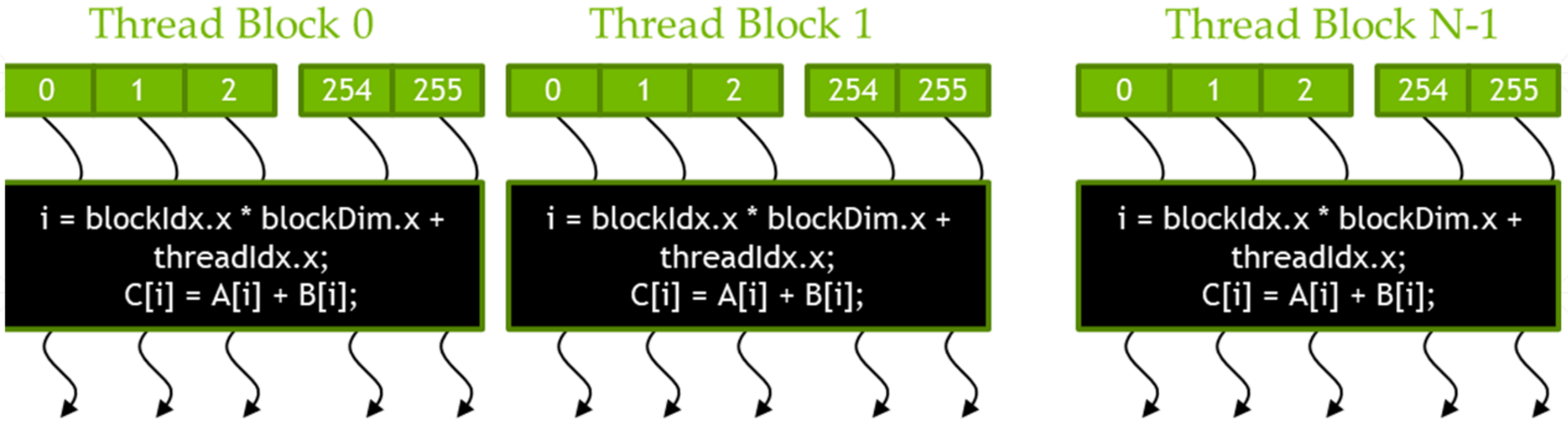


blockIdx and threadIdx

- 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
  - …



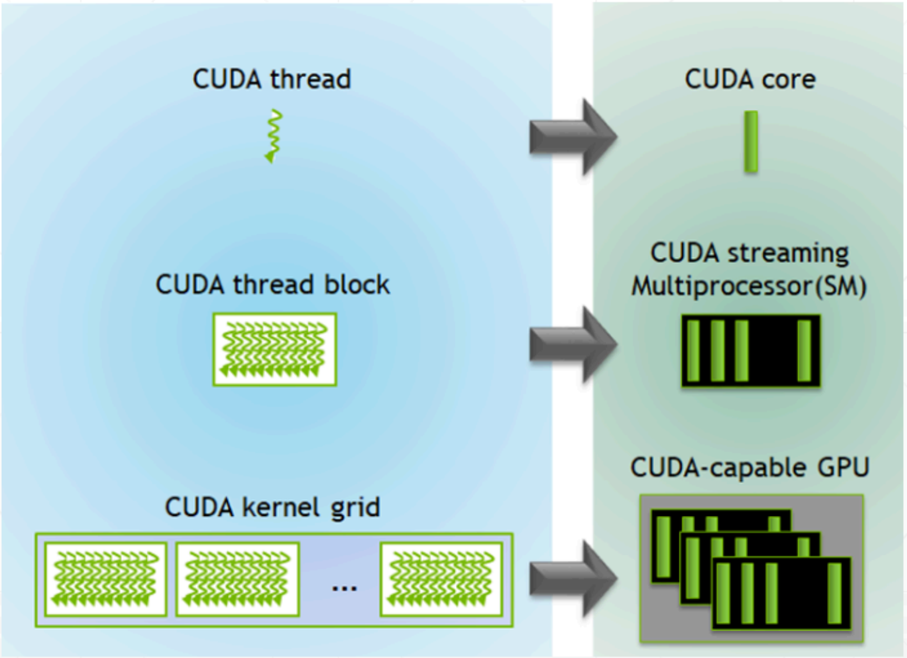
Thread Blocks: Scalable Cooperation



- 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

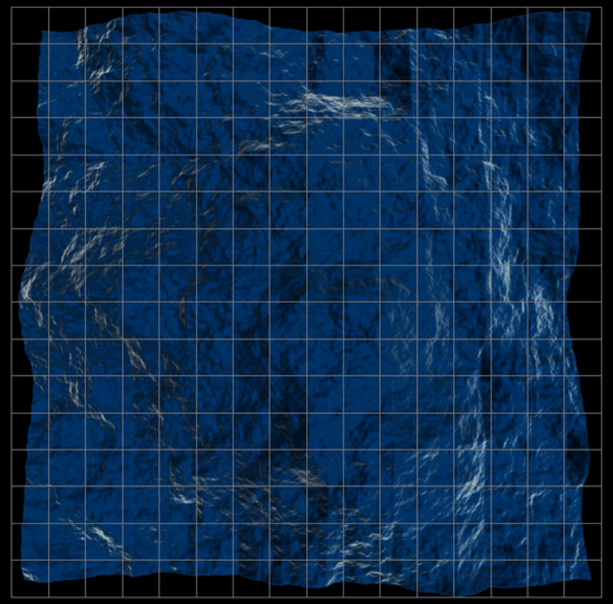
Thread Blocks / Grids

- 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/

Figuring Out How To Divide Up The Work



Total points = 1024 x 1024 = 1048576 = 1M points

Assume one thread per point = 1M threads
GPU max block size = 1024 threads

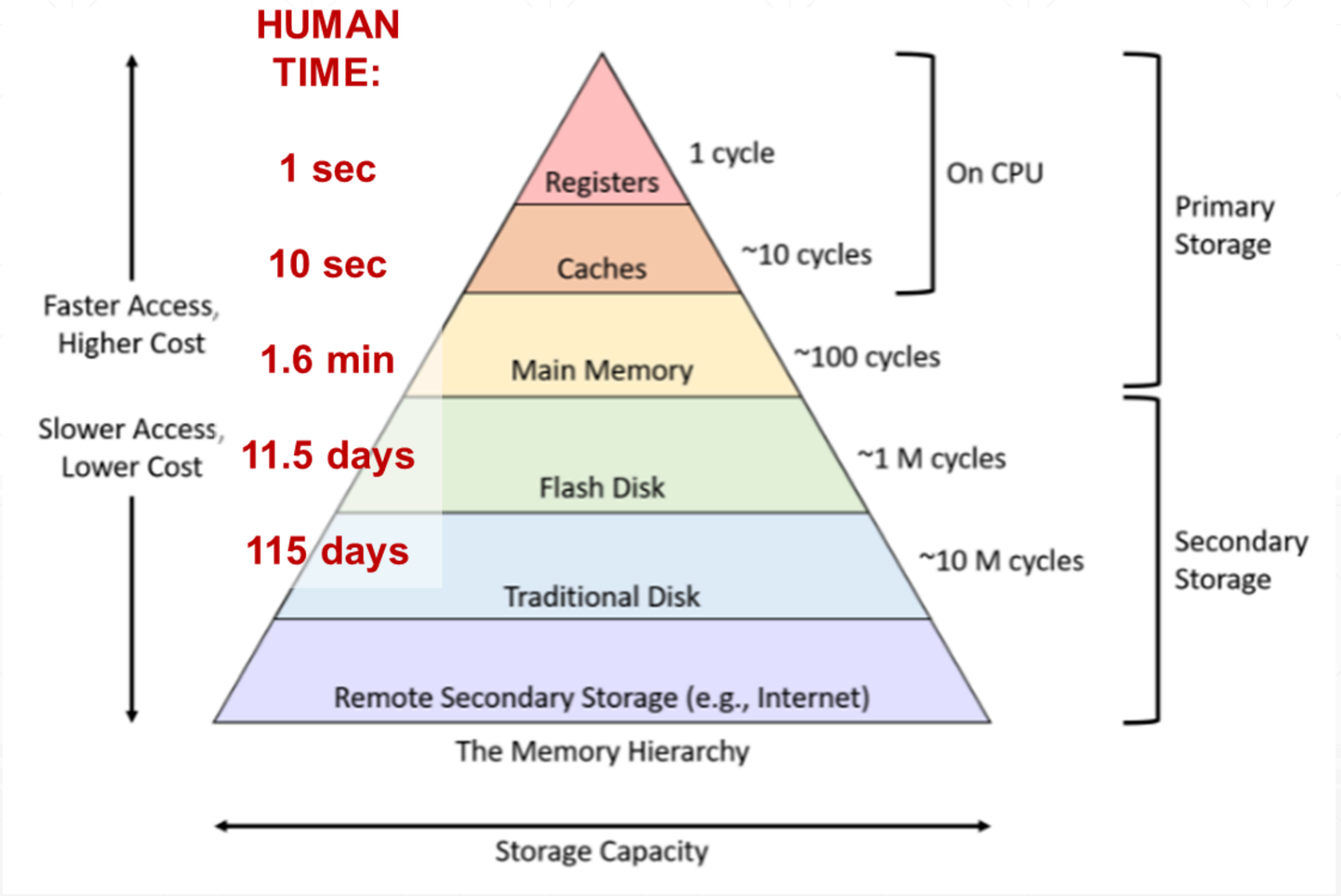
X blocks	Y blocks	Total blocks	Threads per block	
16	16	256	4096	Illegal
32	32	1024	1024	Good
64	64	4096	256	
128	128	16384	64	Too few

Rule of thumb: when in doubt, use a block size of **256**
Corollary: **avoid** block sizes smaller than 128 if you can

NVIDIA

source: Stephen Jones, How To Write A CUDA Program, GTC Fall 2023

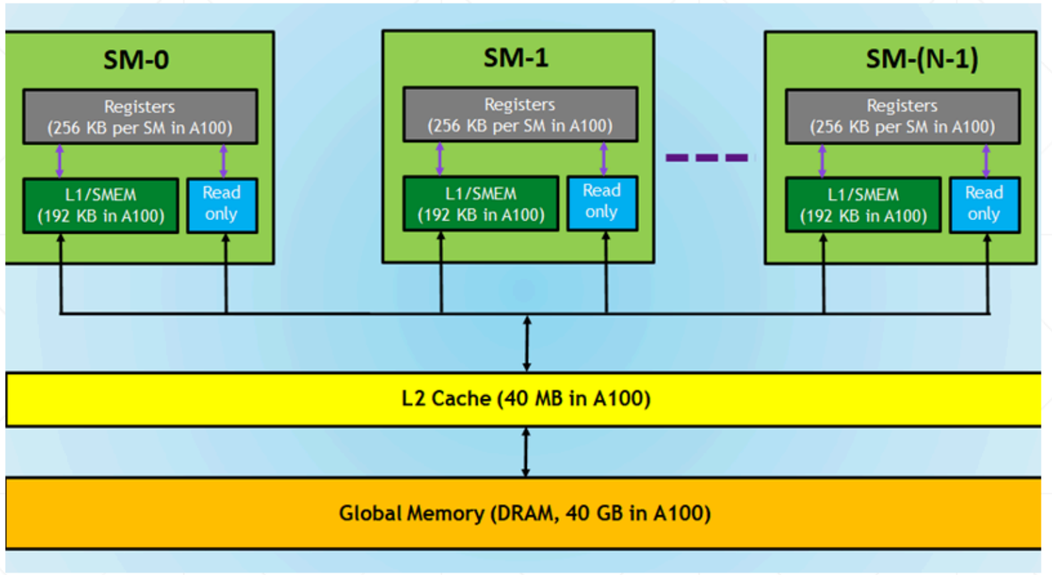
Computer Memory hierarchy

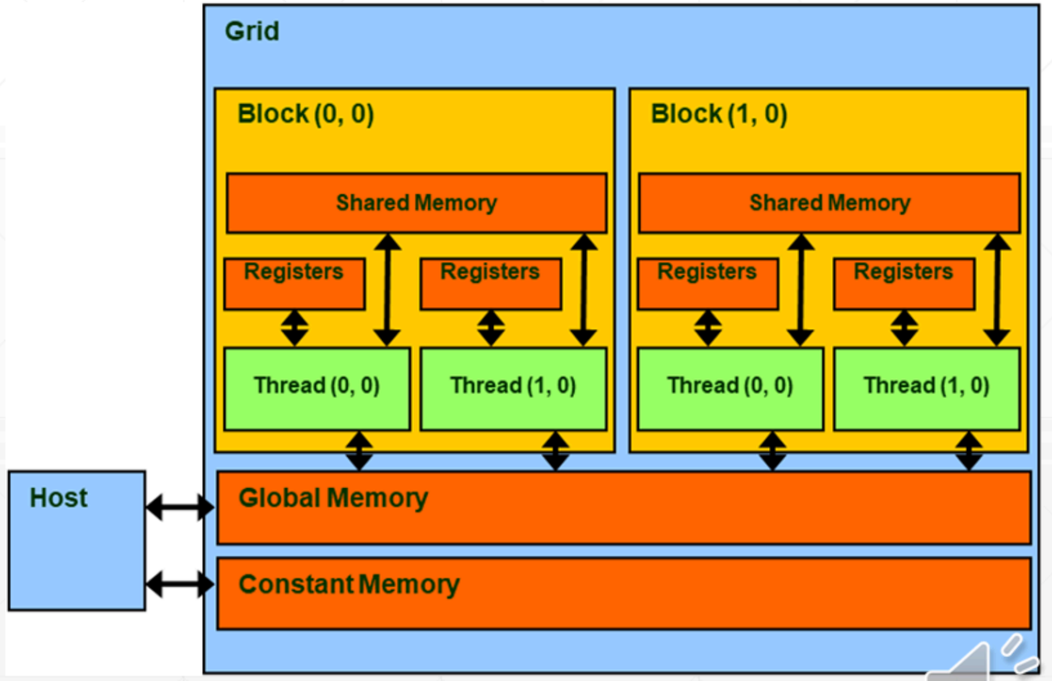


Source: https://diveintosystems.org/antora/diveintosystems/1.0/MemHierarchy/mem_hierarchy.html

GPU Memory Hierarchy

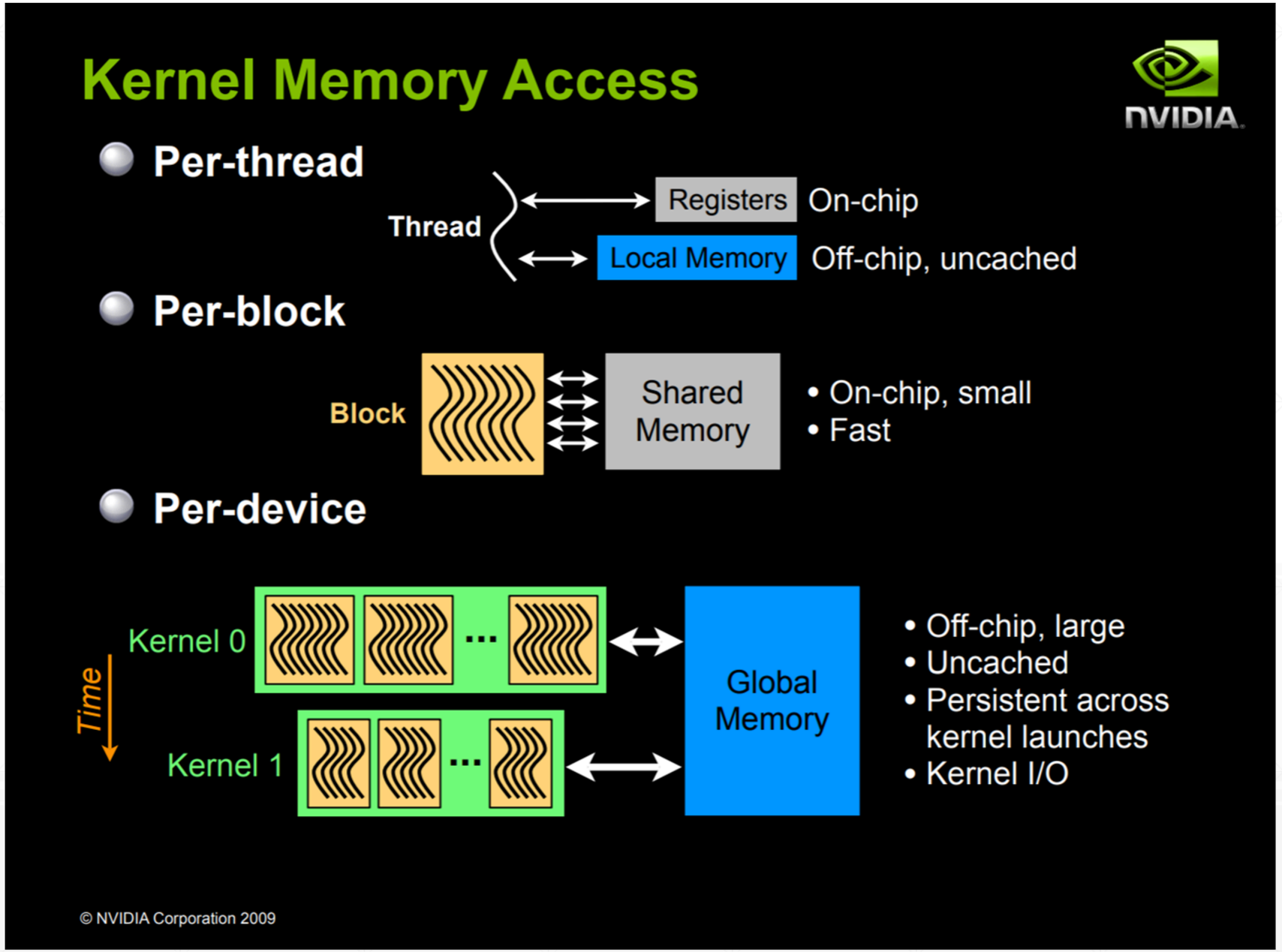
- **Registers** – These are private to each thread, which means that registers assigned to a thread are not visible to other threads. The compiler makes decisions about register utilization.
- **L1/Shared memory (SMEM)** – Every SM has a fast, on-chip scratchpad memory that can be used as L1 cache and 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.





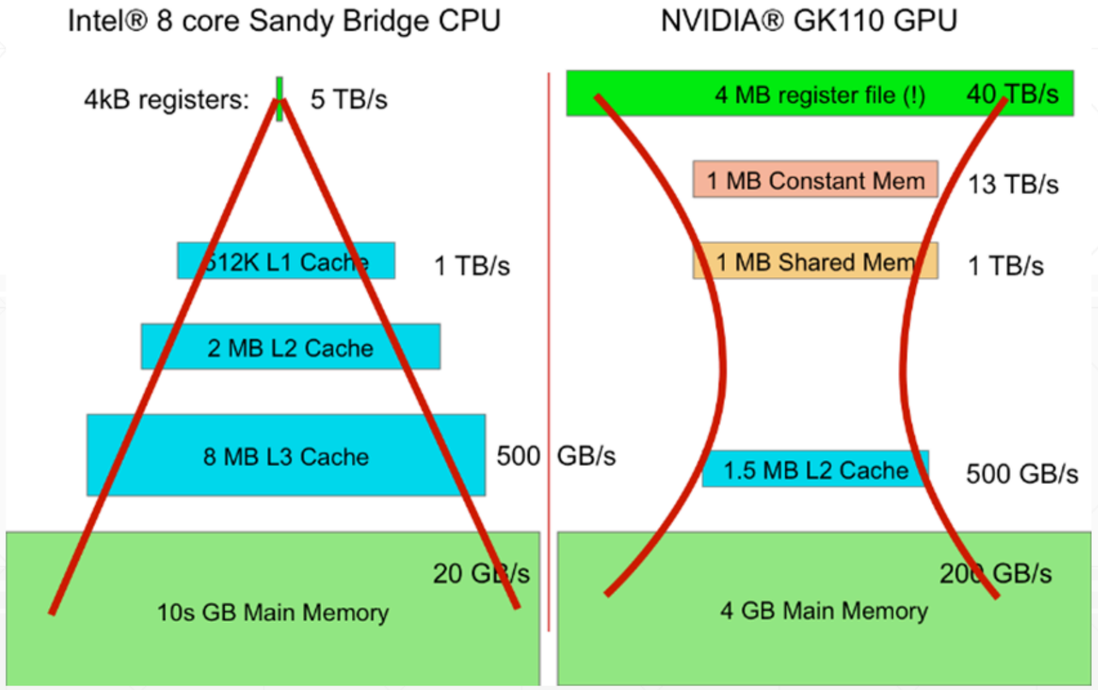
Source: https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

Kernel Memory Access



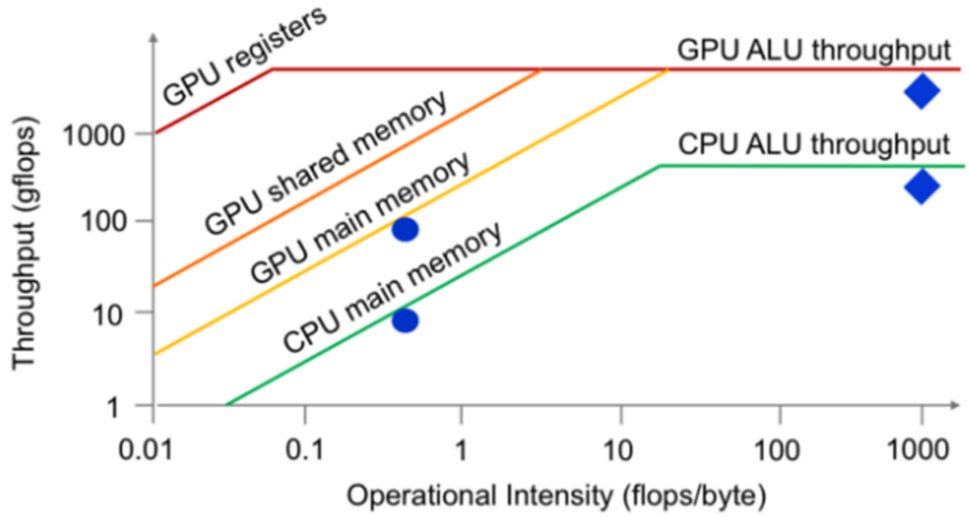
Memory / Arithmetic intensity / Performance

Where is my Memory?

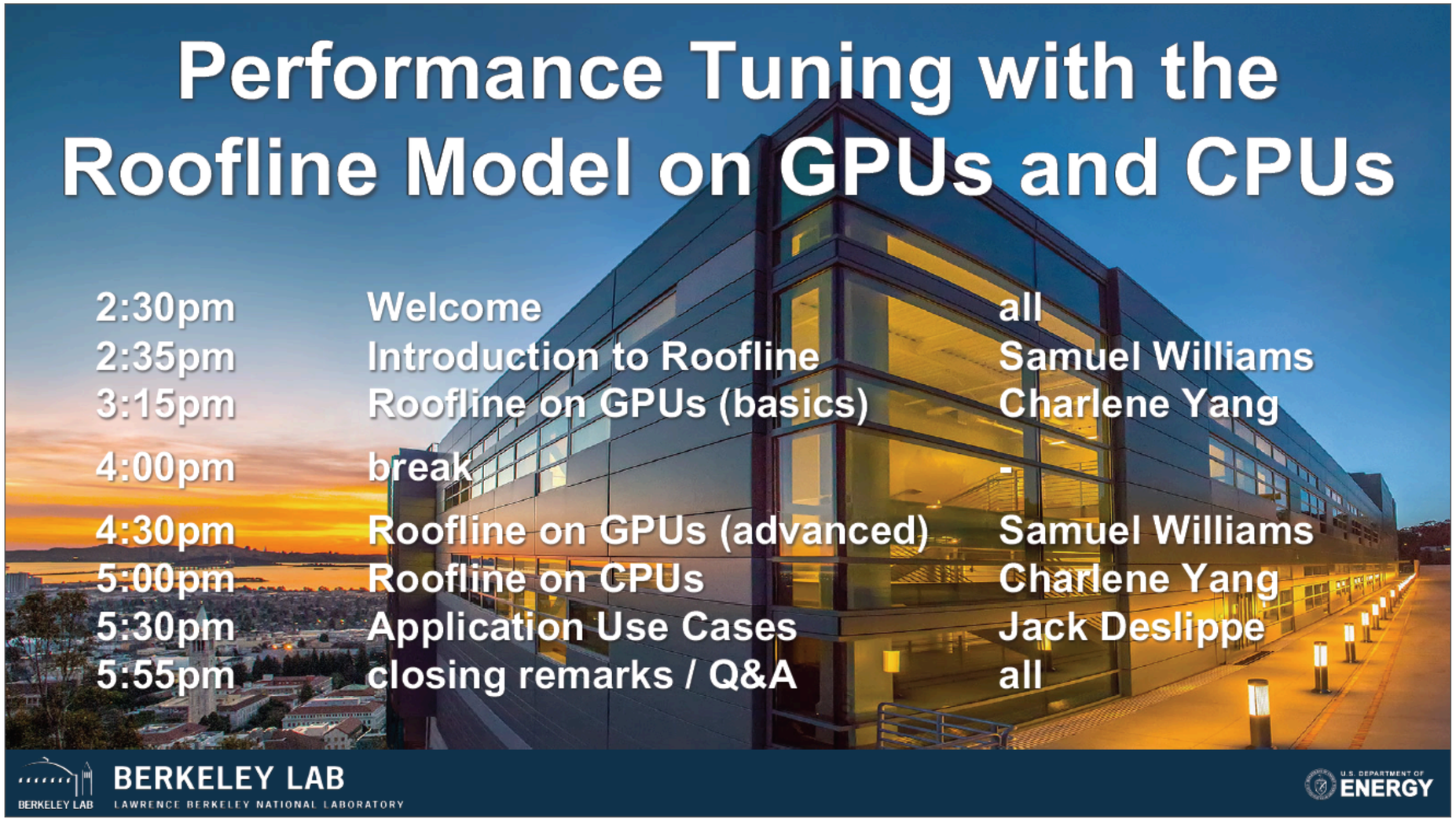


Roofline Design – Matrix kernels

- Dense matrix multiply ◆
- Sparse matrix multiply ●

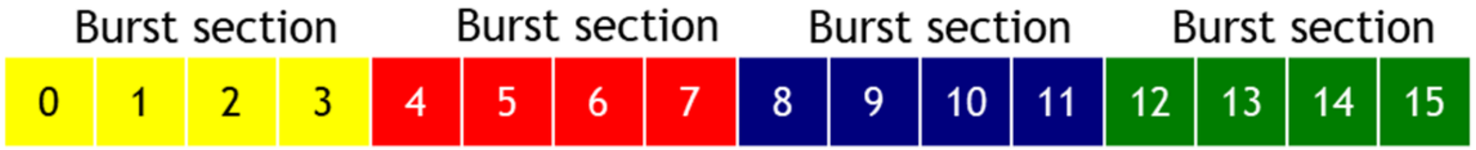


Source: https://developer.nvidia.com/blog/bidmach-machine-learning-limit-gpus/ | https://en.wikipedia.org/wiki/Roofline_model



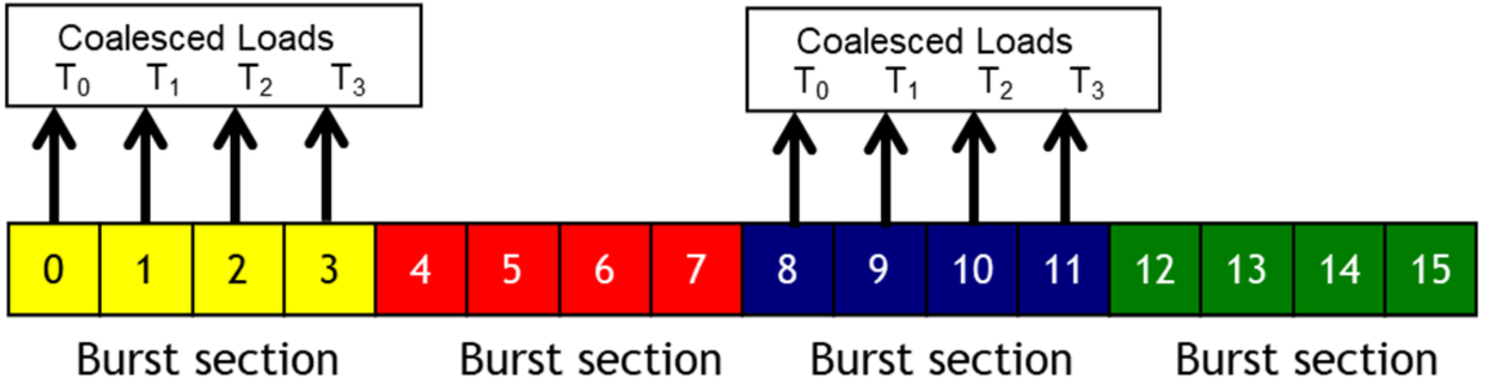
https://crd.lbl.gov/assets/Uploads/ECP20-Roofline-1-intro.pdf
https://crd.lbl.gov/assets/Uploads/ECP21-Roofline-1-intro.pdf

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

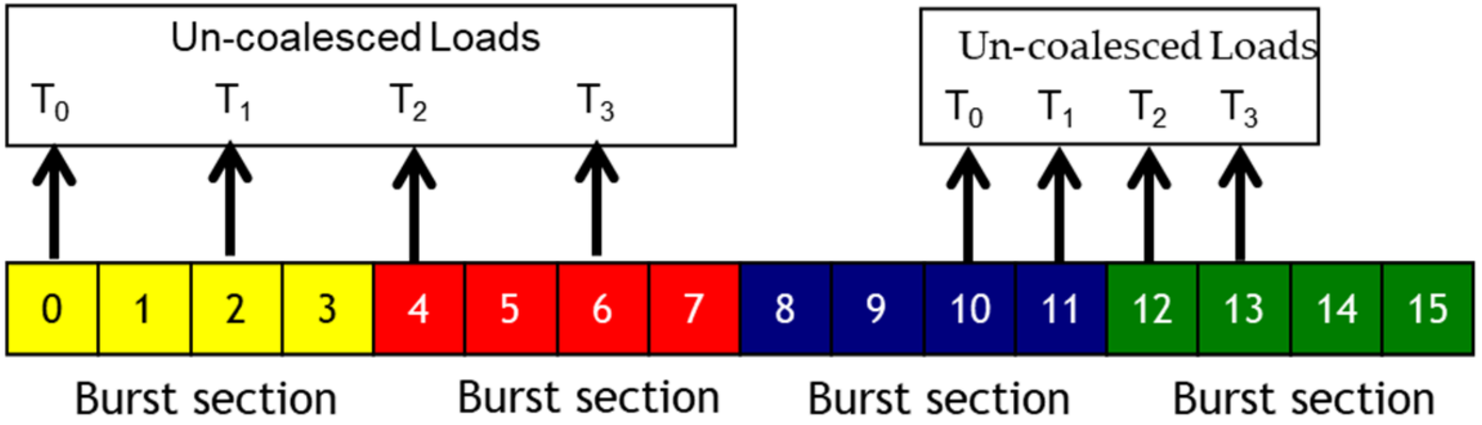


– When all threads of a warp execute a load instruction, if all accessed locations fall into the same burst section, only one DRAM request will be made and the access is fully coalesced.

Un-coalesced Accesses

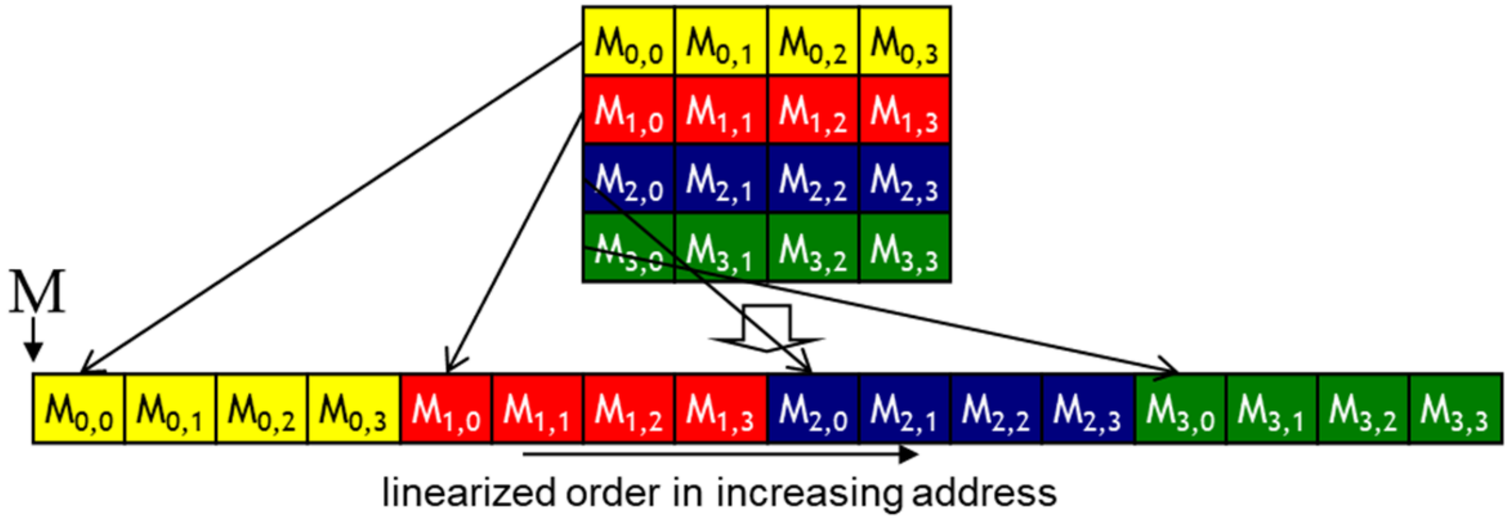
Accesses in a warp are to consecutive locations if the index in an array access is in the form of

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



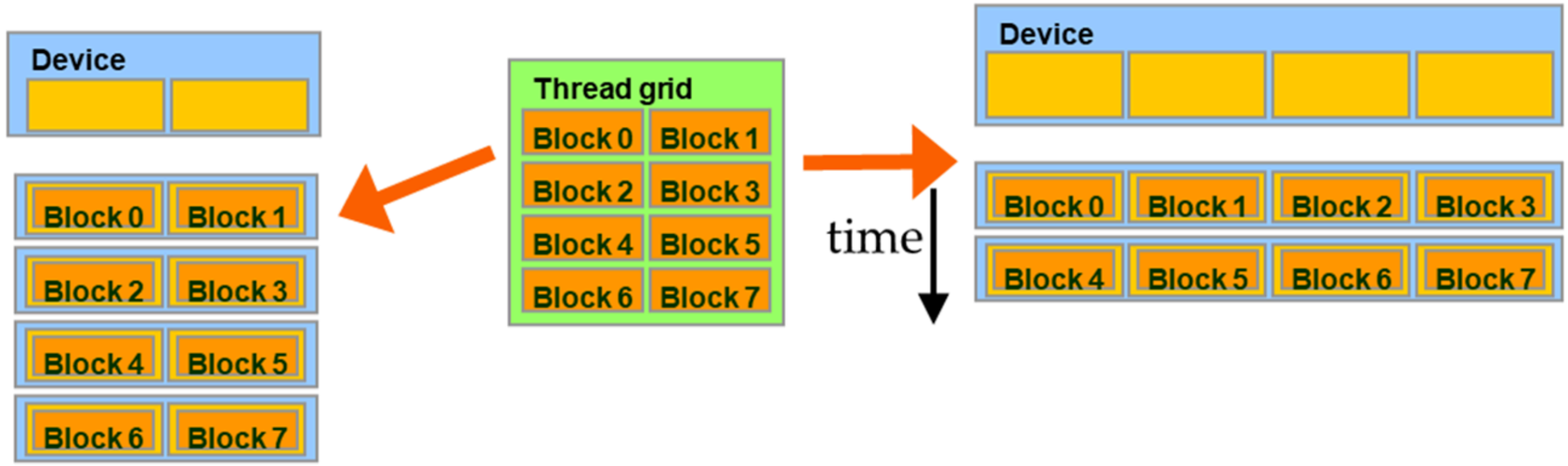
Row-major order

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

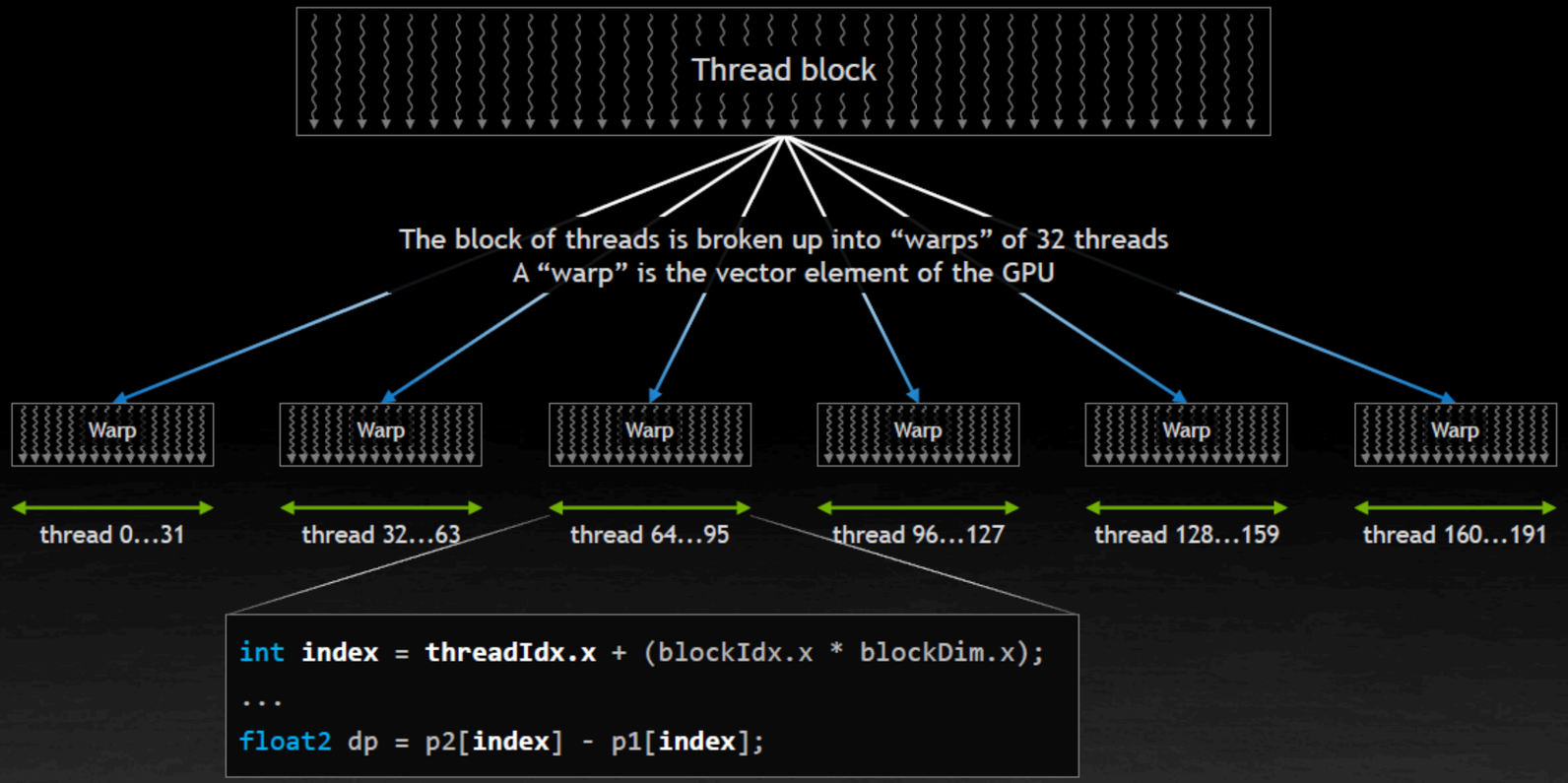
Column-major order

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

Transparent Scalability

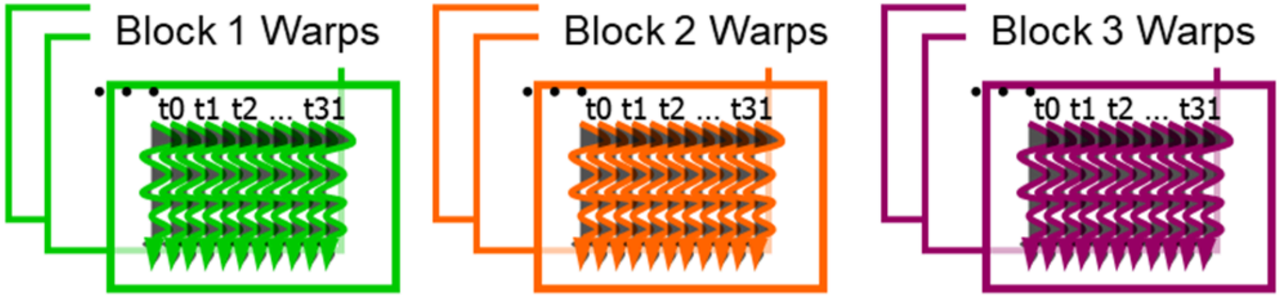


- Each block can execute in any order relative to others.
- Hardware is free to assign blocks to any processor at any time
  - A kernel scales to any number of parallel processors



source: Stephen Jones, HOW CUDA WORKS, GTC Fall 2022

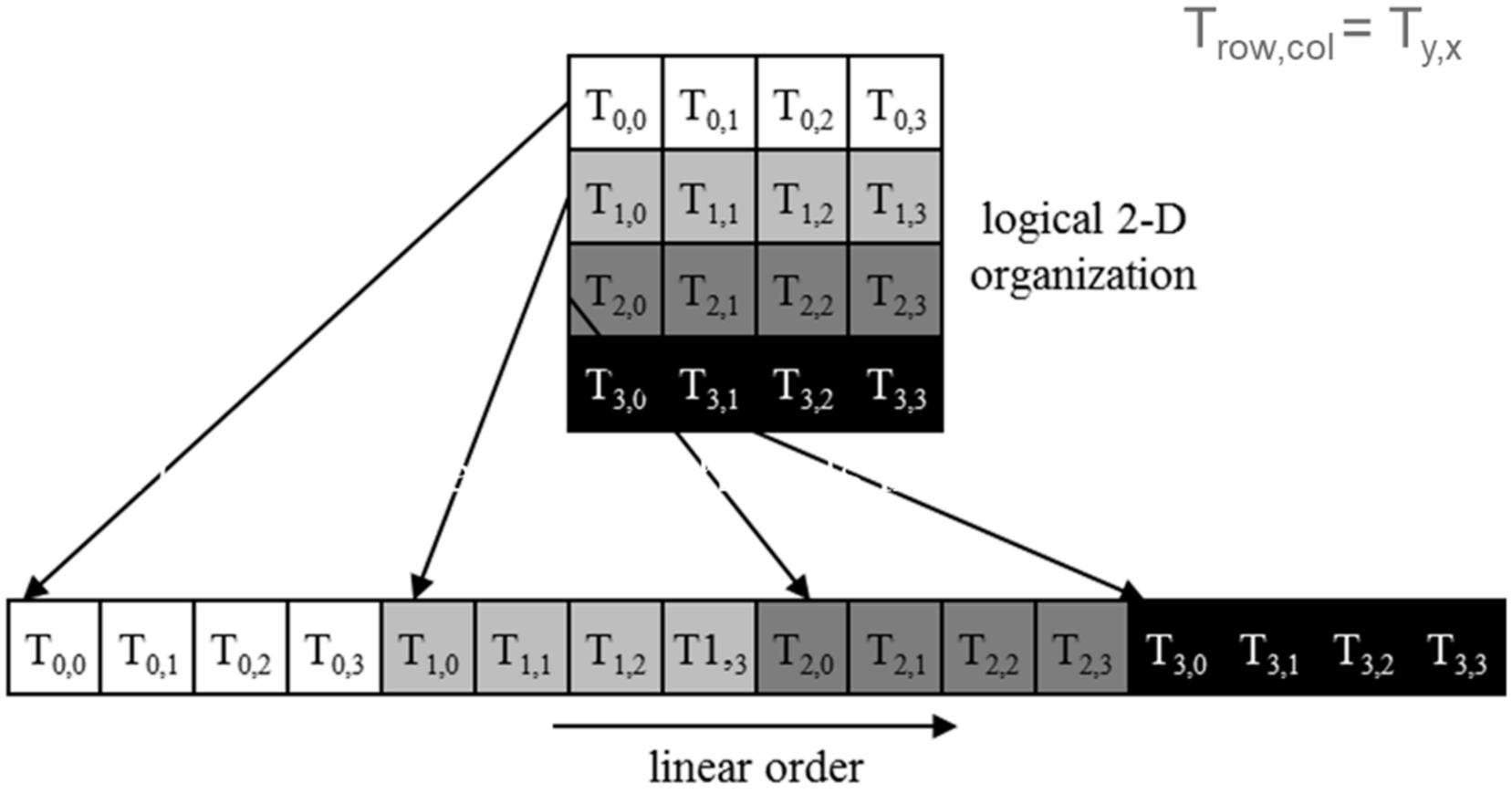
Warps as Scheduling Units



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

Warps in Multi-dimensional Thread Blocks

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



Blocks are partitioned after linearization

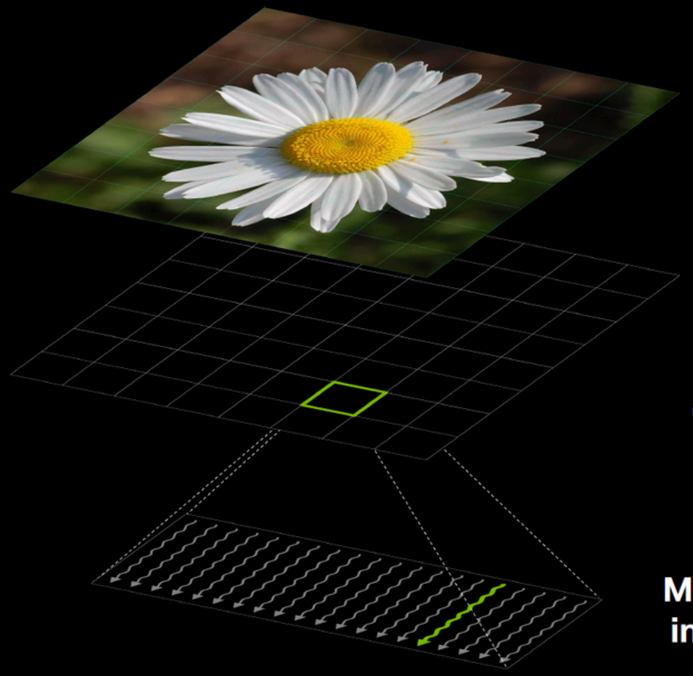
- Linearized thread blocks are partitioned
  - Thread indices within a warp are consecutive and increasing
  - Warp 0 starts with Thread 0
- Partitioning scheme is consistent across devices
  - Thus you can use this knowledge in control flow
  - However, the exact size of warps may change from generation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

SIMD Execution Among Threads in a Warp

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

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

The CUDA Programming Model: Grid → Blocks → Threads



Grid
of work

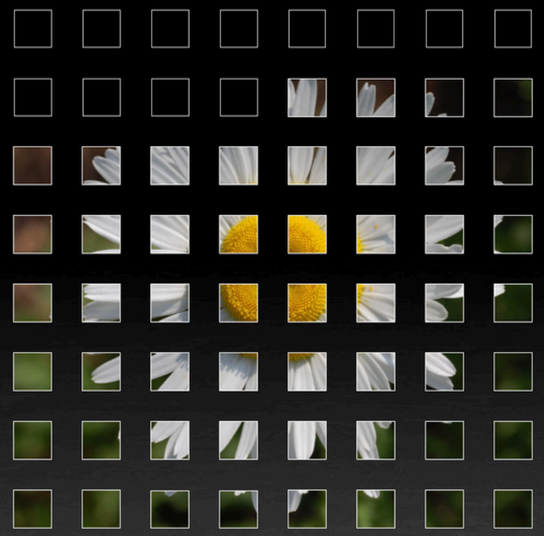
Block
of Threads

**Many Threads
in each Block**

source: Stephen Jones, How To Write A CUDA Program, GTC Fall 2023

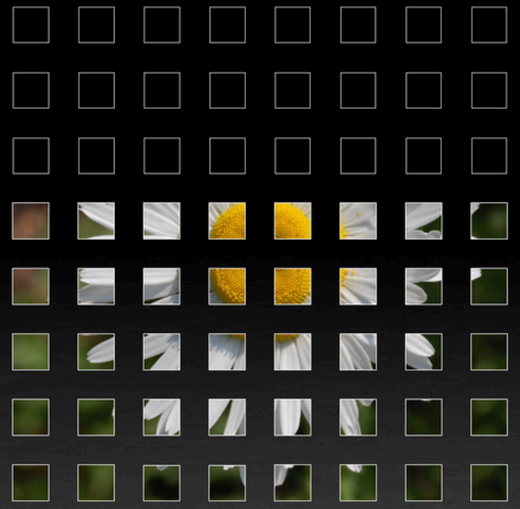
EVERY BLOCK GET PLACED ONTO AN SM

CUDA does not guarantee the order of execution and you cannot exchange data between blocks



BLOCKS CONTINUE TO GET PLACED UNTIL EACH SM IS “FULL”

When a block completes its work and exits, a new block is placed in its spot until the whole grid is done



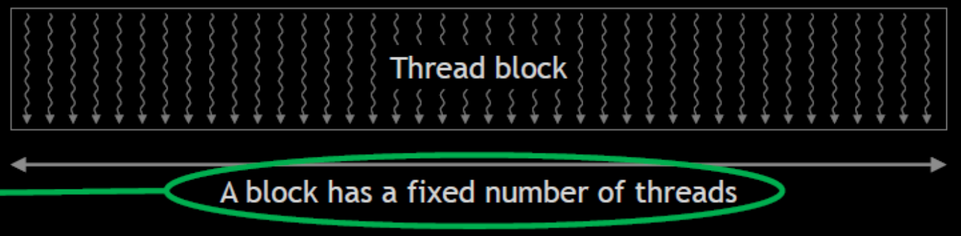
source: Stephen Jones, HOW CUDA WORKS, GTC Fall 2022

ANATOMY OF A THREAD BLOCK

All blocks in a grid run the same program using the same number of threads, leading to 3 resource requirements

1. Block size - the number of threads which must be concurrent
2. Shared memory - common to all threads in a block
3. Registers - depends on program complexity

   Registers are a per-thread resource, so total budget is:
   (threads-per-block x registers-per-thread)

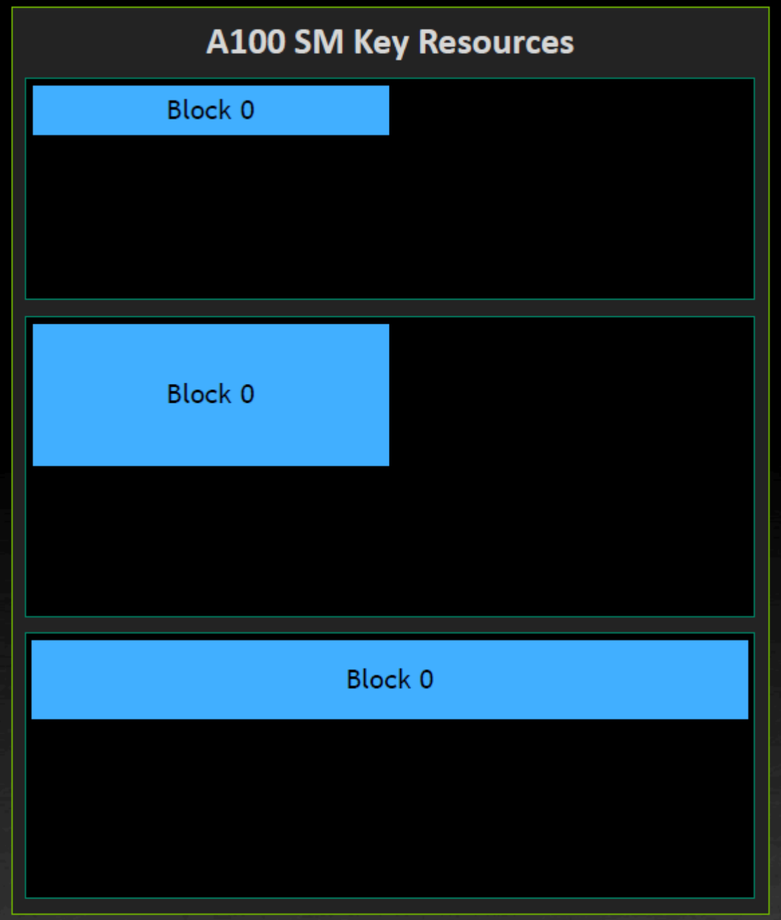



```
__shared__ float mean = 0.0f;
__device__ float mean_euclidian_distance(float2 *p1, float2 *p2) {
    // Compute the Euclidian distance between two points
    float2 dp = p2[threadIdx.x] - p1[threadIdx.x];
    float dist = sqrtf(dp.x * dp.x + dp.y * dp.y);

    // Accumulate the mean distance atomically and return distance
    atomicAdd(&mean, dist / blockDim.x);
    return dist;
}
```

Every thread runs exactly the same program
(this is the "SIMT" model)

source: Stephen Jones, HOW CUDA WORKS, GTC Fall 2022

HOW THE GPU PLACES BLOCKS ON AN SM

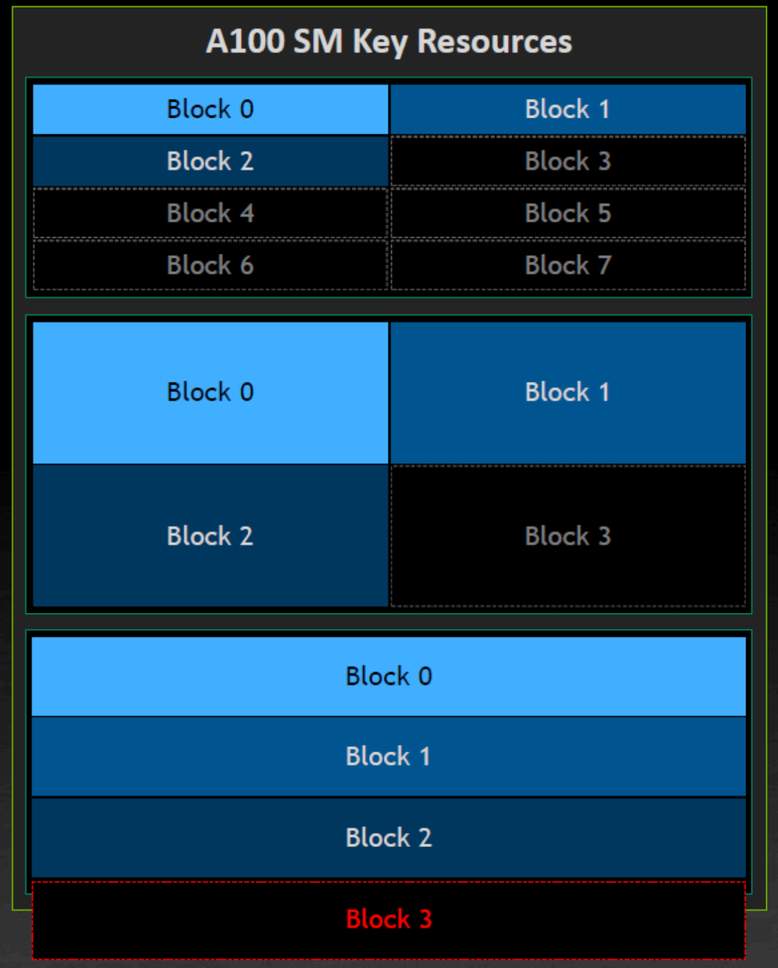


A block has a fixed number of threads always running on a single SM

Example block resource requirements	
256	Threads per block
64	(Registers per thread)
(256 * 64) = 16384	Registers per block
48 kB	Shared memory per block

source: Stephen Jones, HOW CUDA WORKS, GTC Fall 2022

HOW THE GPU PLACES BLOCKS ON AN SM

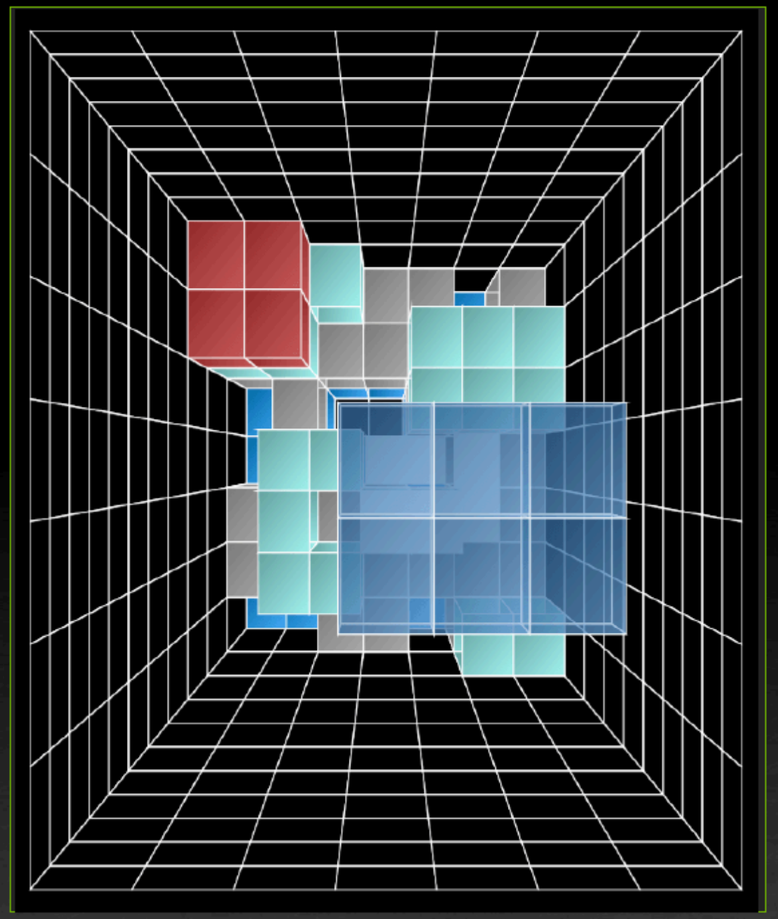


A block has a fixed number of threads always running on a single SM

Example block resource requirements	
256	Threads per block
64	(Registers per thread)
(256 * 64) = 16384	Registers per block
48 kB	Shared memory per block

source: Stephen Jones, HOW CUDA WORKS, GTC Fall 2022

HOW THE GPU PLACES BLOCKS ON AN SM

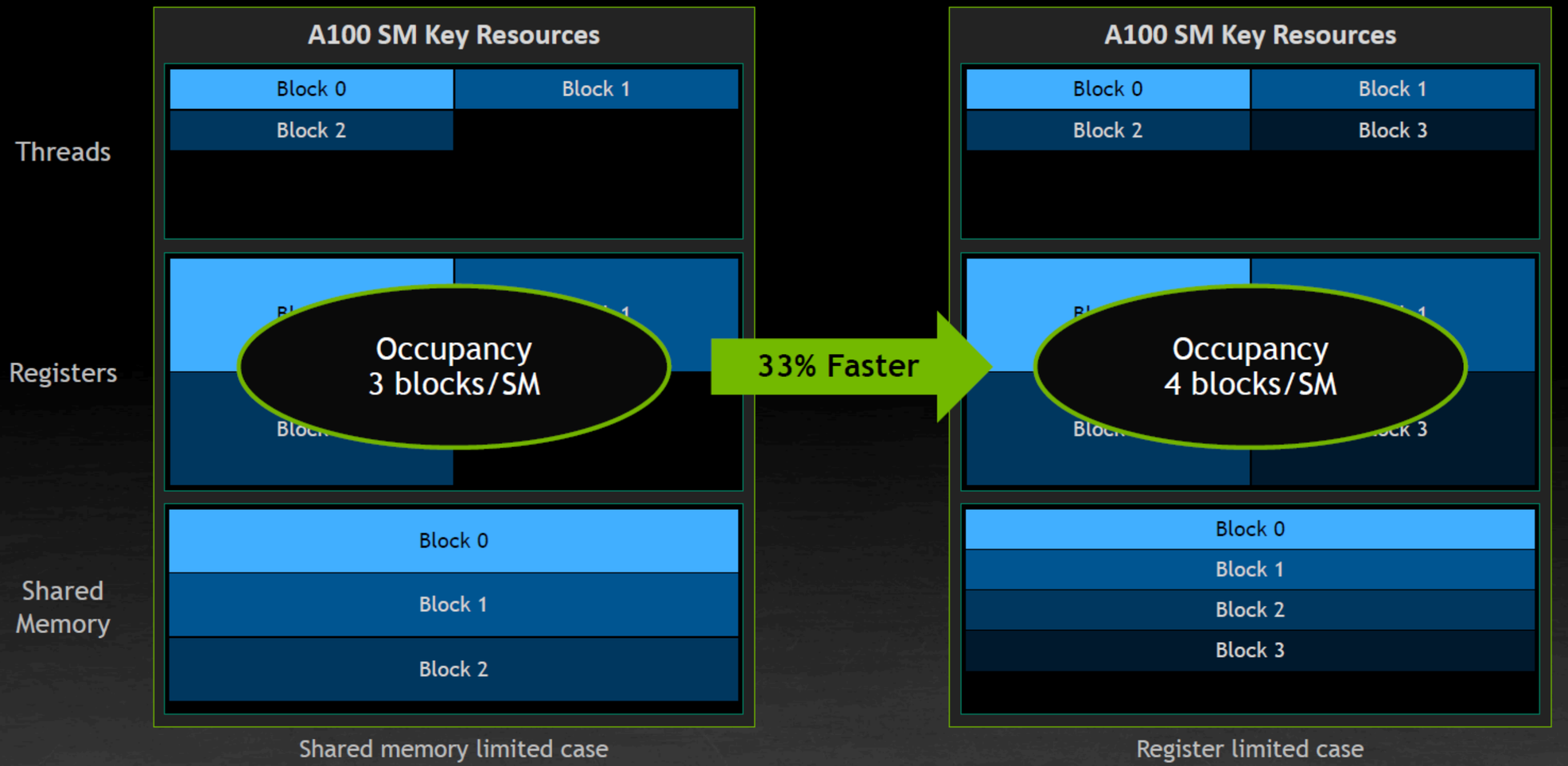


A block has a fixed number of threads
always running on a single SM

Example block resource requirements	
256	Threads per block
64	(Registers per thread)
(256 * 64) = 16384	Registers per block
32 kB	Shared memory per block

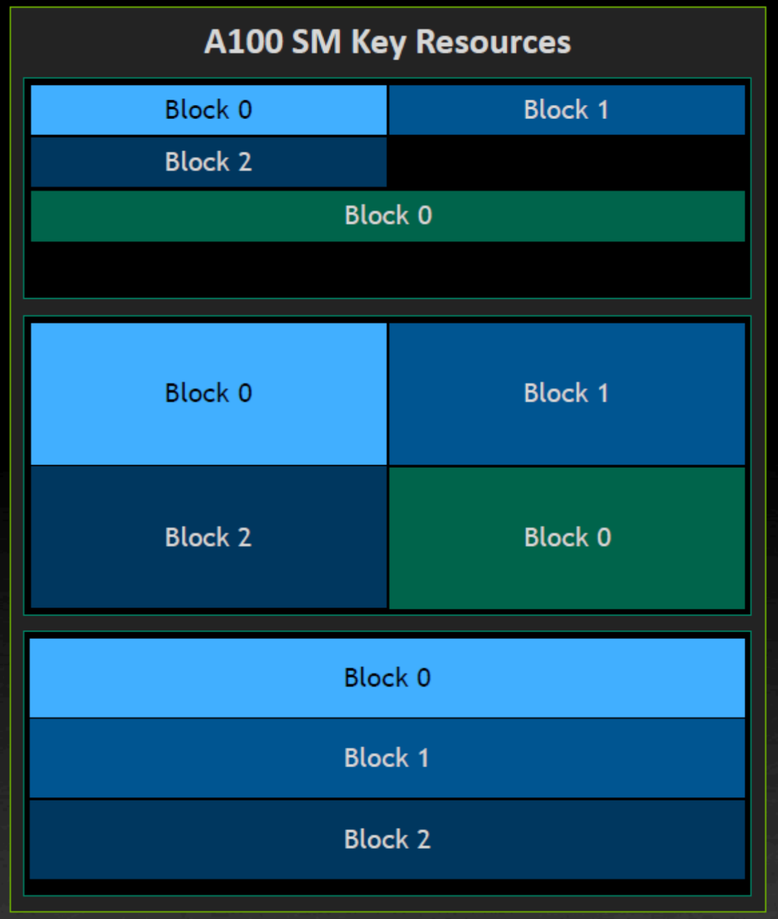
source: Stephen Jones, HOW CUDA WORKS, GTC Fall 2022

OCCUPANCY IS THE MOST POWERFUL TOOL FOR TUNING A PROGRAM



source: Stephen Jones, HOW CUDA WORKS, GTC Fall 2022

FILLING IN THE GAPS

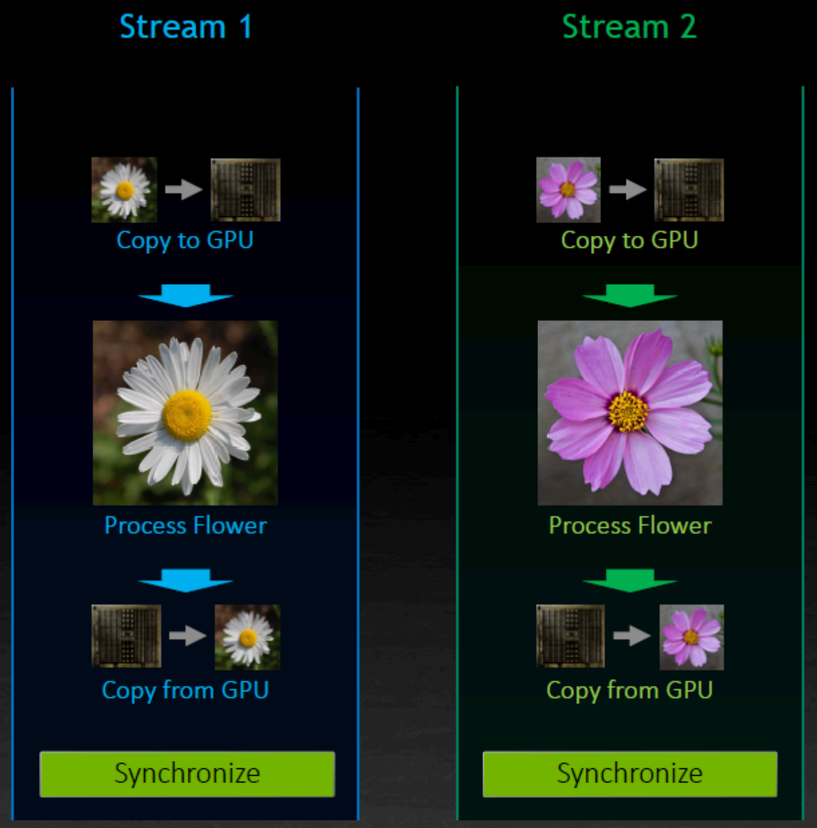


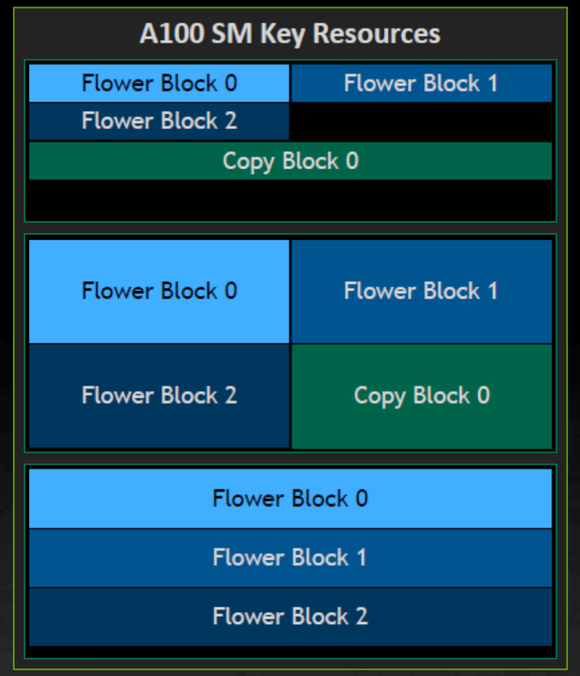
Resource requirements (blue grid)	
256	Threads per block
64	(Registers per thread)
(256 * 64) = 16384	Registers per block
48 kB	Shared memory per block

Resource requirements (green grid)	
512	Threads per block
32	(Registers per thread)
(512 * 32) = 16384	Registers per block
0 kB	Shared memory per block

source: Stephen Jones, HOW CUDA WORKS, GTC Fall 2022

KEEPING THE GPU FULL





source: Stephen Jones, HOW CUDA WORKS, GTC Fall 2022

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!

Deterministic US/sensor processing with NVIDIA HOLOSCAN

Towards Deterministic End-to-end Latency for Medical AI

- Running several AI/ML applications, each with its own visualization components, leads to unpredictable end-to-end latency, primarily due to GPU resource contentions.
- To mitigate this, manufacturers typically deploy separate workstations for distinct AI applications, thereby increasing financial, energy, and maintenance costs.
- NVIDIA's Holoscan platform provides a real-time AI system for streaming sensor data and images.
- It leverages **CUDA MPS (Multi-process Service)** for spatial partitioning of compute workloads and isolates compute and graphics processing onto separate GPUs.

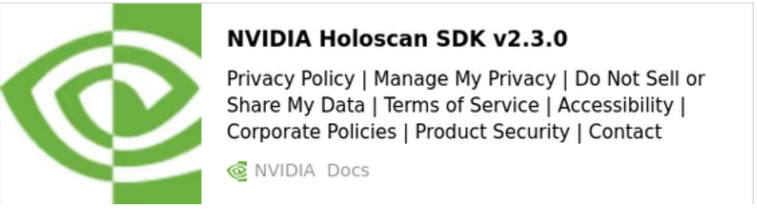
NVIDIA Holoscan SDK is an open-source AI sensor processing platform designed for low-latency real-time steaming data and images

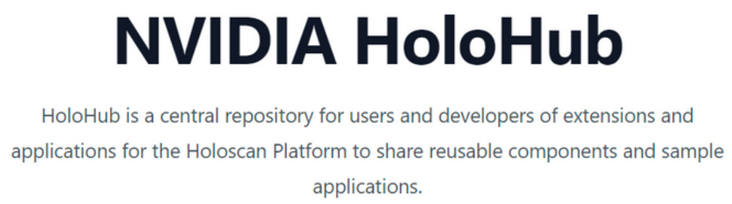
source: Soham Sinha, Shekhar Dwivedi, Mahdi Azizian, Towards Deterministic End-to-end Latency for Medical AI Systems in NVIDIA Holoscan, https://doi.org/10.48550/arXiv.2402.04466

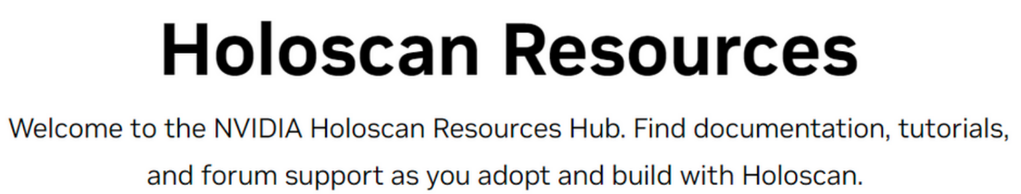
NVIDIA Holoscan SDK

NVIDIA Holoscan is the sensor processing platform that streamlines the development and deployment of AI and high-performance computing (HPC) applications for real-time insights. Accelerating the full workflow, it offers the software and hardware needed to build AI applications and deploy sensor processing capabilities from edge to cloud. From surgery to satellites, Holoscan helps companies explore new capabilities, accelerate time to market, and lower costs.

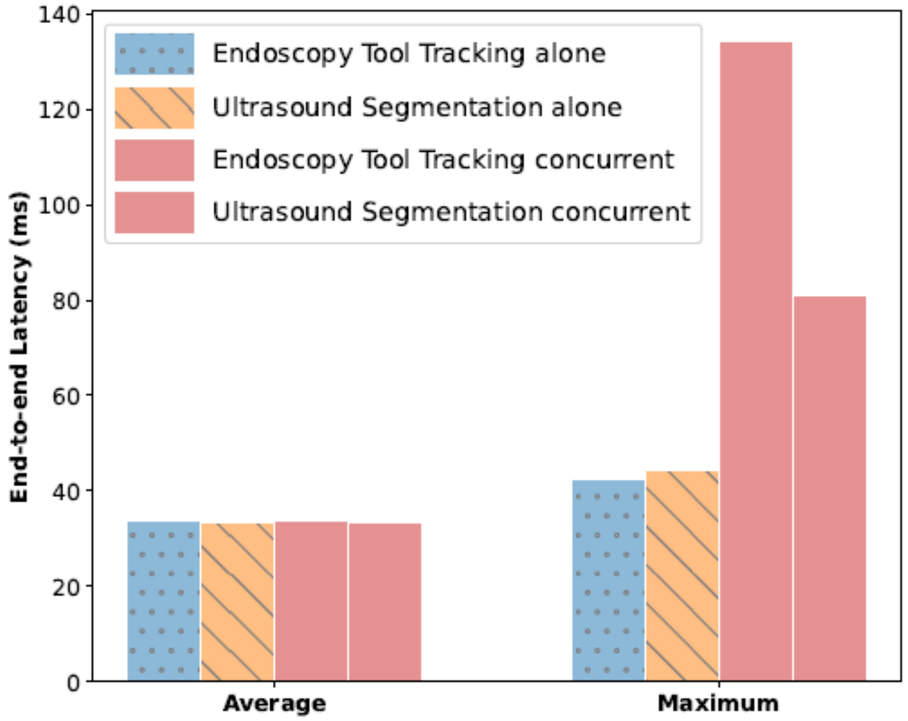
- **Sensor Processing —** Use the Camera Serial Interface protocol and front-end sensors for video capture, ultrasound research, data acquisition, and connection to legacy medical devices.
- **Low Latency —** Tap into the Holoscan SDK's data transfer latency tool to measure complete, end-to-end latency for video processing applications
- **Reference AI Pipelines —** Access AI reference pipelines for radar, high-energy light sources, endoscopy, ultrasound, and other streaming video applications.







GPU processing latency



Average and Maximum End-to-end Latency on an x86 Workstation with an RTX A4000 GPU

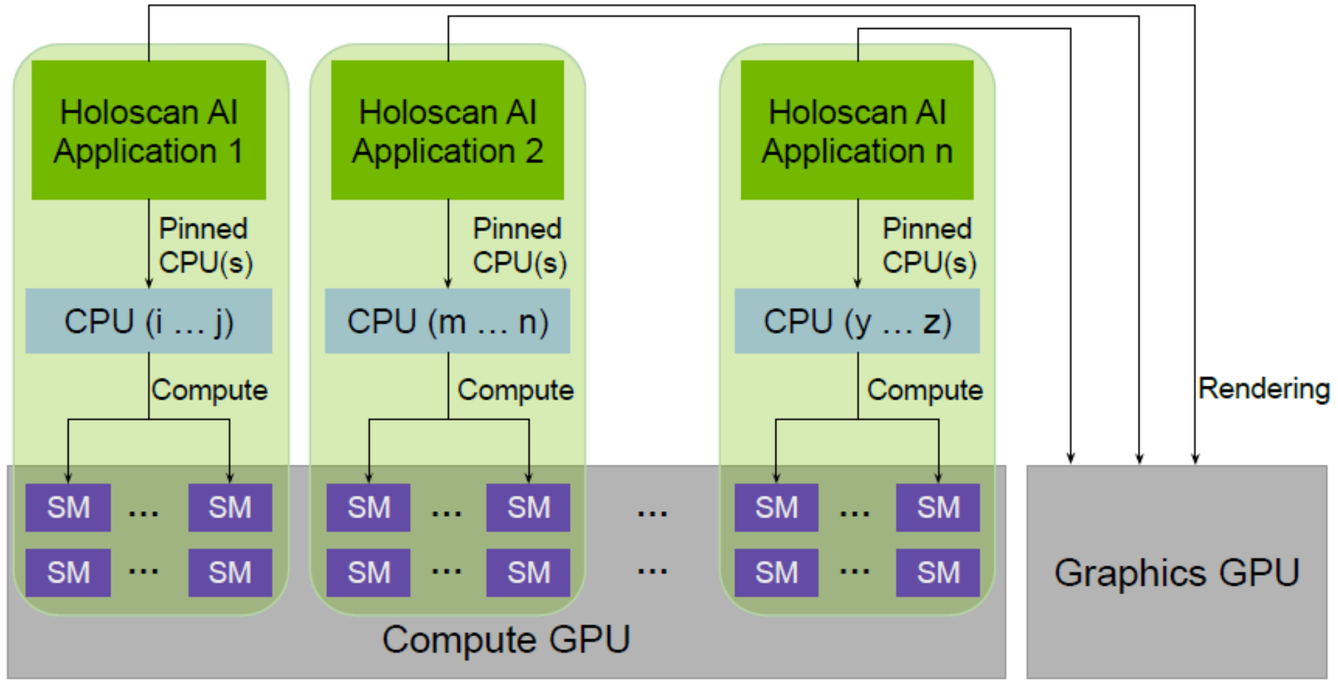


TABLE II: Maximum E2E Latency (ms) for concurrent Endoscopy Tool Tracking and Ultrasound Segmentation

Application	One A4000	One A4000 + MPS	Two A4000s + IMG-MPS
Endoscopy Tool Tracking	133.76	91.24	90.57
Ultrasound Segmentation	80.61	62.49	55.23

source: Soham Sinha, Shekhar Dwivedi, Mahdi Azizian, Towards Deterministic End-to-end Latency for Medical AI Systems in NVIDIA Holoscan, https://doi.org/10.48550/arXiv.2402.04466

Minimizing End-to-end Latency

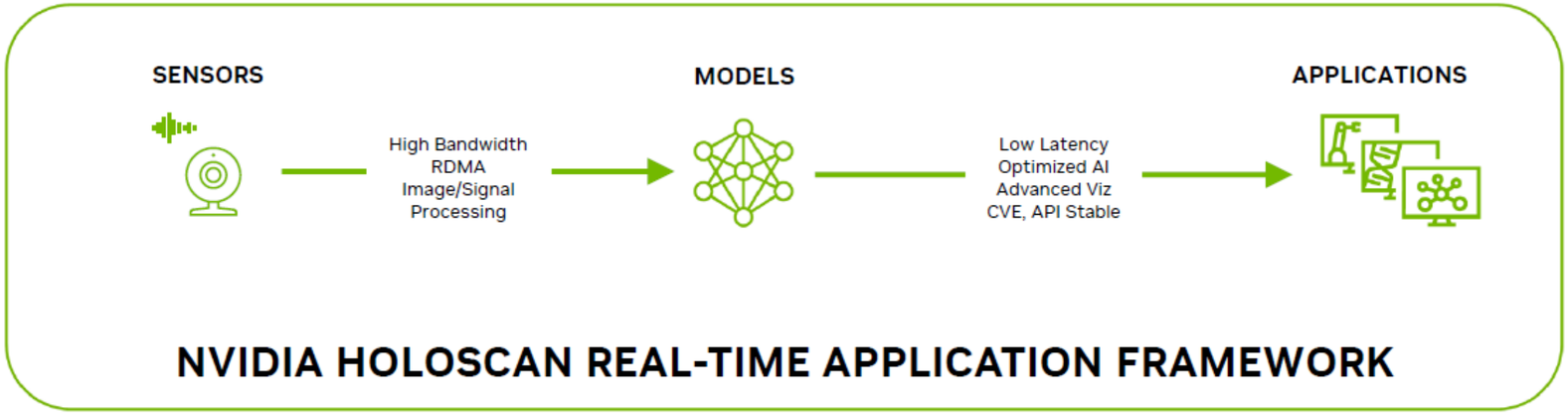
GUIDELINES for managing heterogeneous GPU workloads that include equally important compute and graphics tasks:

- In scenarios where external constraints prohibit the use of multiple GPUs - due to cost, energy, or other factors - CUDA MPS could be utilized to partition a single GPU, after a thorough resource profiling of the involved applications.
- However, benefits only with this setup are limited, due to contention between graphics and multiple compute contexts.
- Deploying distinct GPUs for compute and graphics tasks greatly enhances performance predictability. For cost considerations, a less powerful GPU may handle graphics if it meets the application's memory needs.
    - For instance, an IGX Orin with future support for inter-operable iGPU and discrete GPU would be a suitable platform.

source: Soham Sinha, Shekhar Dwivedi, Mahdi Azizian, Towards Deterministic End-to-end Latency for Medical AI Systems in NVIDIA Holoscan, https://doi.org/10.48550/arXiv.2402.04466

NVIDIA Holoscan: Enabling Real-Time AI for Medical Devices & Beyond

Software-Defined | Realtime & High Bandwidth | Multi-modal | Scalable | Production-ready

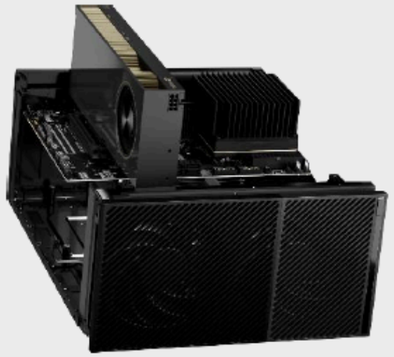


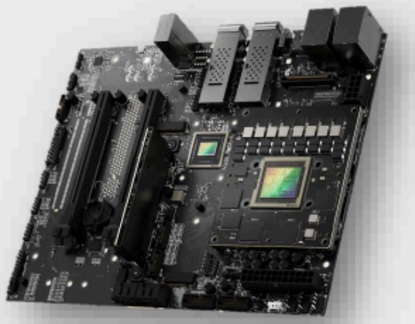
Embedded | Enterprise Edge | Data Center

Super Low Latency	Built-in Scalability	Multi-Modal	Full-stack Long-term	Medical Grade
10ms	30+	I/O	10yr	62304 (SW) 60601 (IGX)
Sensor to Display 4K 240Hz	AI models < 50 ms	Video, Ultrasound, Laser, RF, ...	CVEs + Bug Fixes API Stability	

Enterprise-grade Edge Platform

IGX Orin





Powerful Compute

- GPU: AI Performance: up to 1705 TOPS*
- CPU: 12-core Arm® Cortex®-A78AE v8.2
- An Ampere architecture - 64 Tensor core GPU

Production Ready

- Built for Medical IEC 60601
- Long life HW Support
- Whole-stack long-term SW support

Networking I/O

- NVIDIA ConnectX®-7 SmartNIC 2x100 GbE
- Rich IO HDMI | PCIe | Ethernet | WiFi
- GPUDirect / RDMA

Safety/Security/ Manageability

- OTA and remote monitoring via BMC
- Hardware ROT, FW TPM, Encrypted storage & memory
- SEP, Proactive AI safety support,

Interested in a test drive? Try it now!

Build and run HoloHub app in < 10 minutes

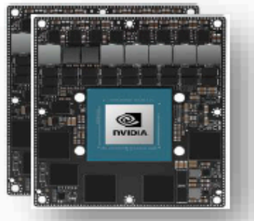
Build Holohub container

```
# Clone the codebase directly
git clone https://github.com/nvidia-holoscan/holohub.git

cd holohub

# Build using Holoscan SDK container from NGC for your iGPU / dGPU setup as base
./dev_container build

# Launch Holohub development container
./dev_container launch
```

Build Holohub app in container

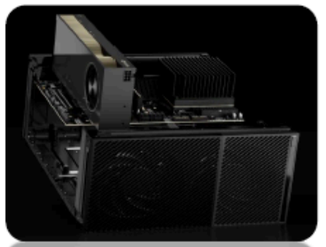
```
# Build endoscopy tool tracking sample app
./run build endoscopy_tool_tracking

# Launch C++ endoscopy tool tracking app
./run build endoscopy_tool_tracking cpp

# Launch Python endoscopy tool tracking app
./run build endoscopy_tool_tracking python
```

Jetson Nano



Jetson AGX

X86 + dGPU



IGX