Fundamentals of Accelerated Computing with CUDA C/C++

Nitin Shukla
n.shukla@cineca.it

CINECA HPC High Performance Computing | NVIDIA

Who AM I?

About Me

- Master degree in Physics Engineering Instituto Superior Técnico, Lisbon, Portugal
- PhD in Computational Plasma Physics, Instituto Superior Técnico, Lisbon, Portugal
- HPC Scientific Application Engineer, CINECA Italy
- Porting and optimisation of astrophysical, cosmological and plasma physic code

Projects

- PI of Astro and Plasma related research at CINECA
- Leading SPACE-COE
- DevTeam member for DE360
- DevTeam member of Cheese2.0 , Cheese1.0
- EuroFusion (2020-2022)

NVIDIA DLI Instructor and Mentor/TA for GPU BootCamp

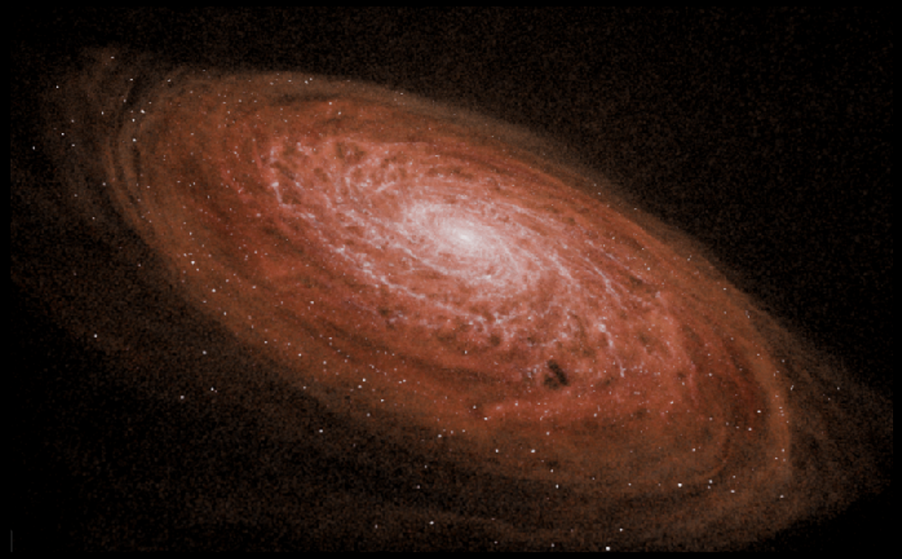
- Fundamentals of CUDA
- OpenACC

https://www.space-coe.eu/

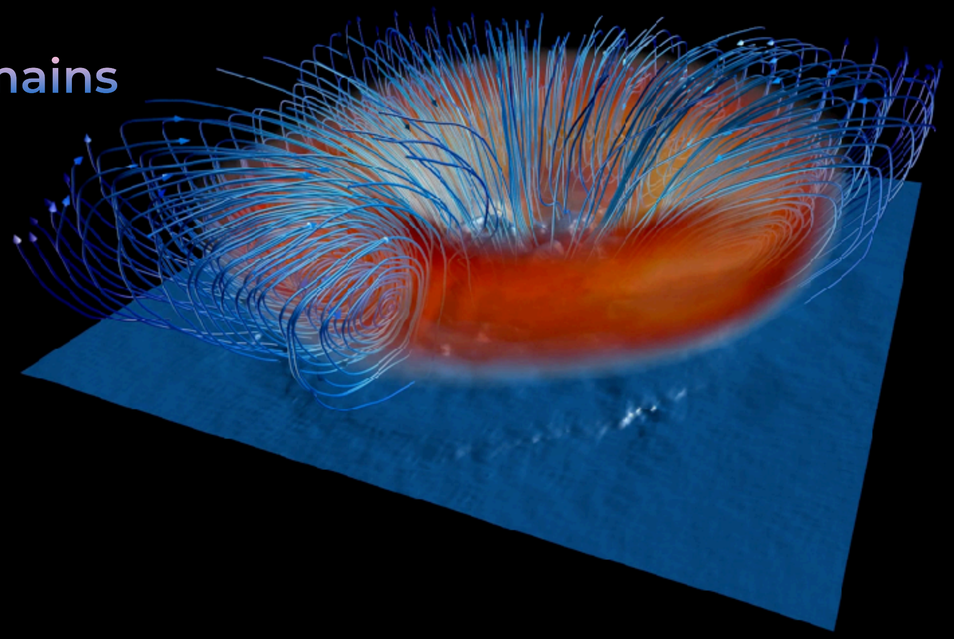
Scalable Parallel Astrophysical Codes for Exascale

1st January 2023 → 31st December 2026

Scientific Domains



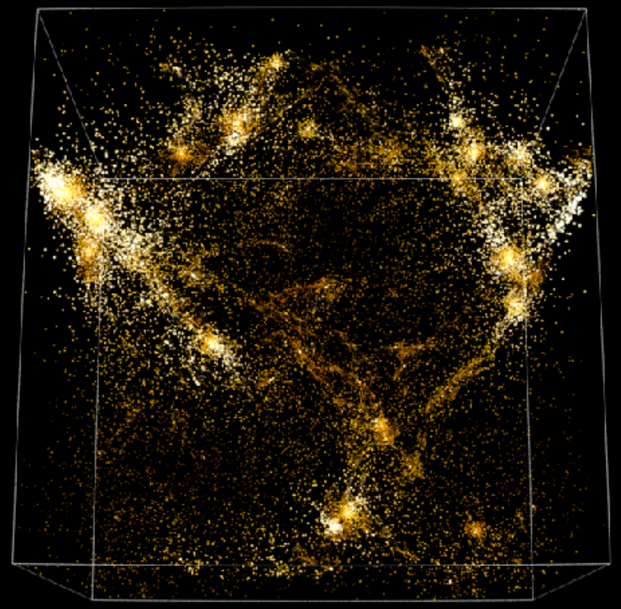
Galaxy b-field



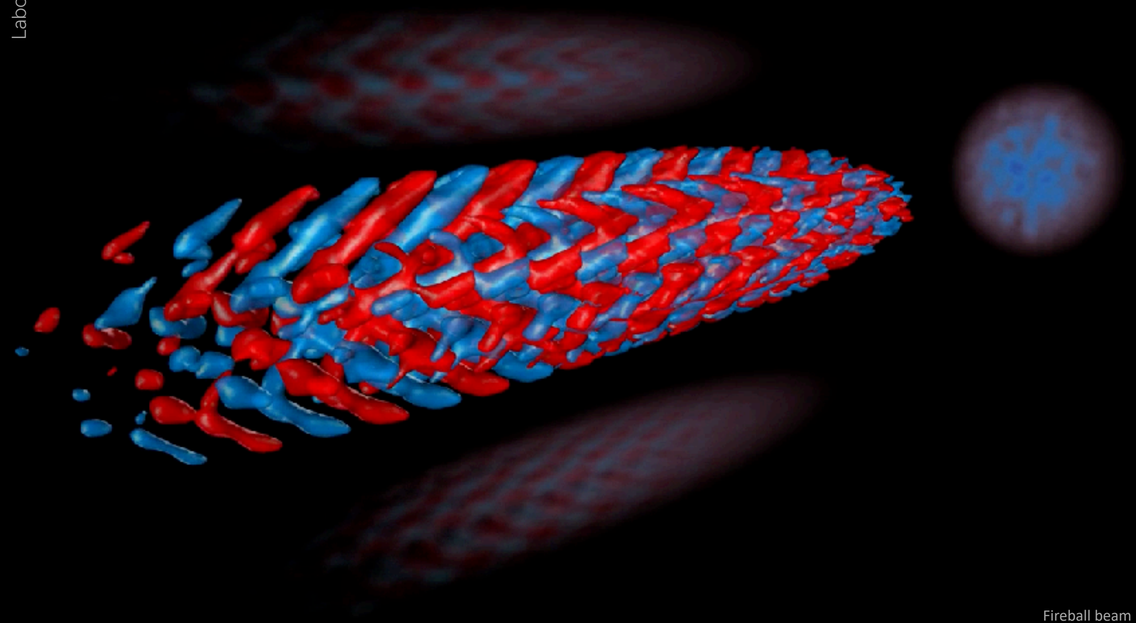
Laser-plasma interactions

Astrophysics and Cosmology

Laboratory plasma



OpenGadget Gass mass



Fireball beam

Team Members

Specialised in PlasmaAstro Physics + HPC

Nitin Shukla

Lead of Research, Astrophysics and Plasma Research

Alessandro Romeo

DevTeam

Michael Redenti

DevTeam

Caterina Caravita

High-Level Support

Access the compute node

```
salloc -N1 --cpus-per-task=1 --ntasks-per-node=1 -A xX -t 00:30:00 -p boost_usr_prod --gres=gpu:1 --qos=boost_qos_dbg
```

```
srun --cpu_bind=none --nodes=1  --gres=gpu:1 --pty bash -i
```

Objective

1. Why heterogenous computing becoming so important in HPC or computer science in general?

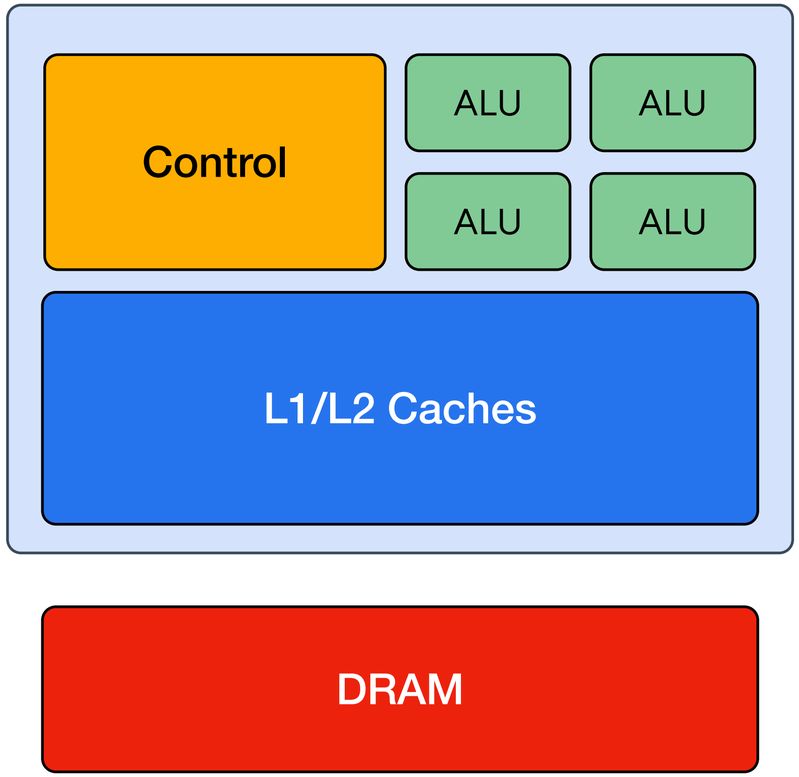
2. What are the major differences between **latency** devices (CPU cores) and **throughput** devices (GPU cores)?

From Silicon to Supercomputing

CPU Design → Memory Systems → HPC Evolution

CPUs: Latency Oriented Design

Called brain of the computer



CPU Architecture

- **Powerful ALU:** Reduced operation latency
- **Large caches:** Convert long latency memory access to short latency cache access
- **Sophisticated control:** Branch prediction & data forwarding

Pipeline Stages:

Fetch → Decode → Execute → Write back

The Performance vs. Capacity Trade-Off

Fast memory is expensive and limited. Large memory is slow and cheap. How do we bridge this fundamental gap?

CPU Registers
~1ns
64 bytes
$$$$$

L1/L2 Cache
~3ns
256KB
$$$$

Main Memory
~100ns
16GB
$$

Storage
~10ms
1TB+
$

The Solution: Virtual Memory

Memory Abstraction
Programs see unlimited, contiguous memory

Demand Paging
Load only needed data into RAM

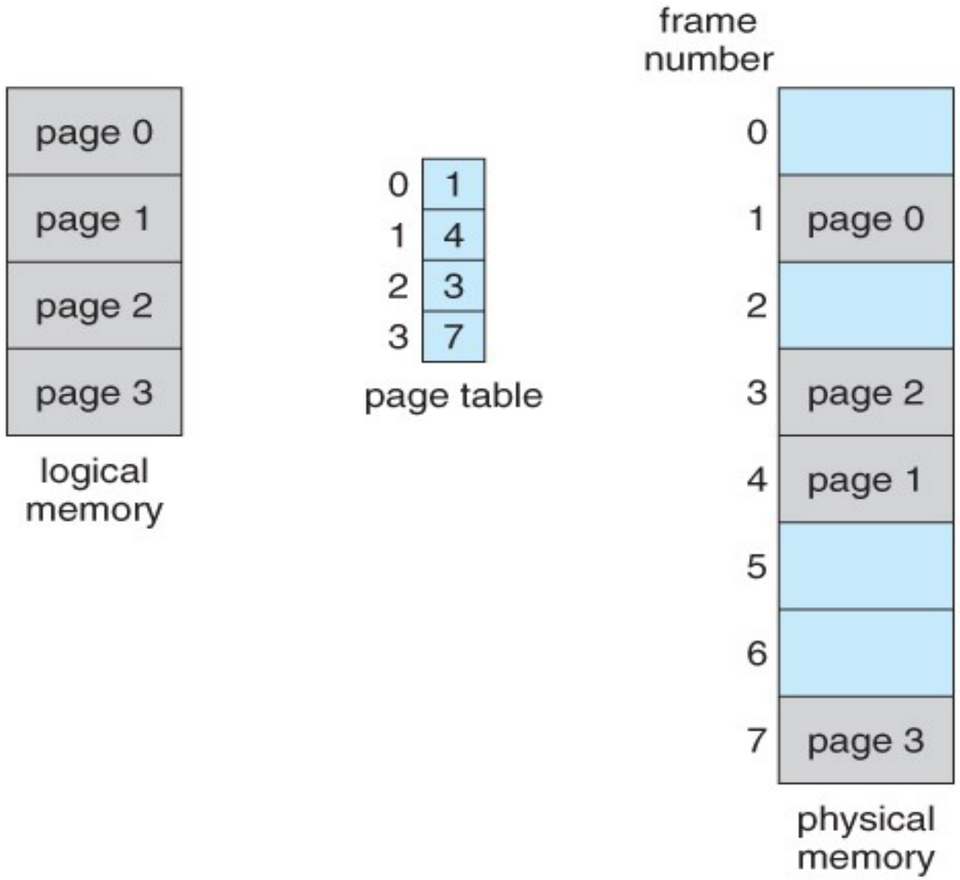
Memory Protection
Isolate processes from each other

Virtual Memory: the Great Abstraction

Memory Paging: Managing Latency

A memory page is a fixed-length contiguous block of virtual memory that serves as the fundamental unit of memory management in modern computing systems.

- Typically 4 KB in size, though this can vary based on operating system and hardware architecture
- With paged memory each program has its own logical memory, which can be broken into consecutive pages
- The program will read and write data to the pages , which are mapped to the physical memory via page table

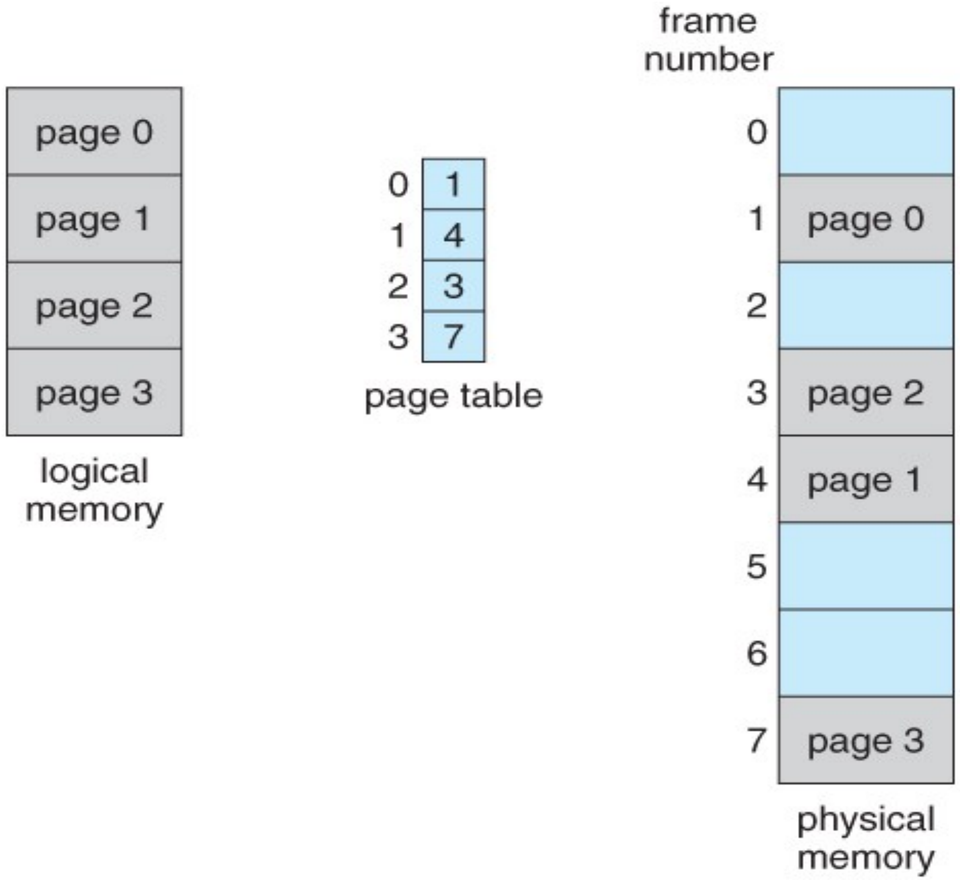


Virtual Memory: the Great Abstraction

Memory Paging: Managing Latency

A memory page is a fixed-length contiguous block of virtual memory that serves as the fundamental unit of memory management in modern computing systems.

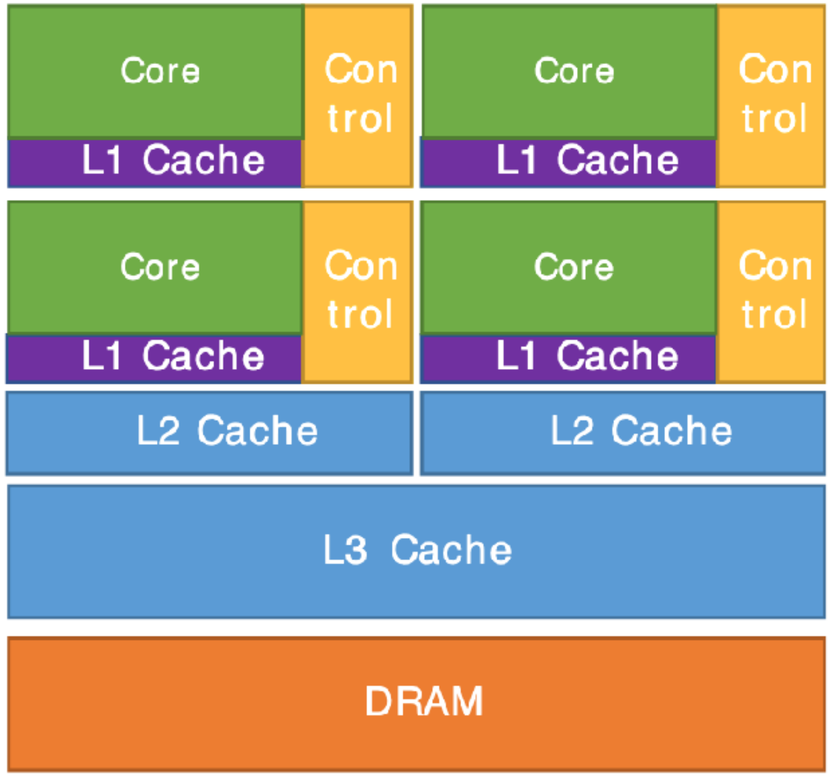
- Although main memory is utilised better with paged memory, its size is limited
- When main memory does't have enough space, and more data needs to be written, some pages will be moved to the hard drive, called **pages out or swapping**
- The process of moving pages from hard drive back to memory is called ***page in***



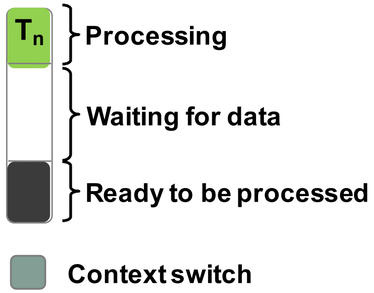
CPUs: Latency Oriented Design

Called brain of the computer

Multi-core CPU



CPU core – Low Latency Processor



CPUs: Latency Oriented Design

Computer architecture drives parallelism at the core level

Bandwidth

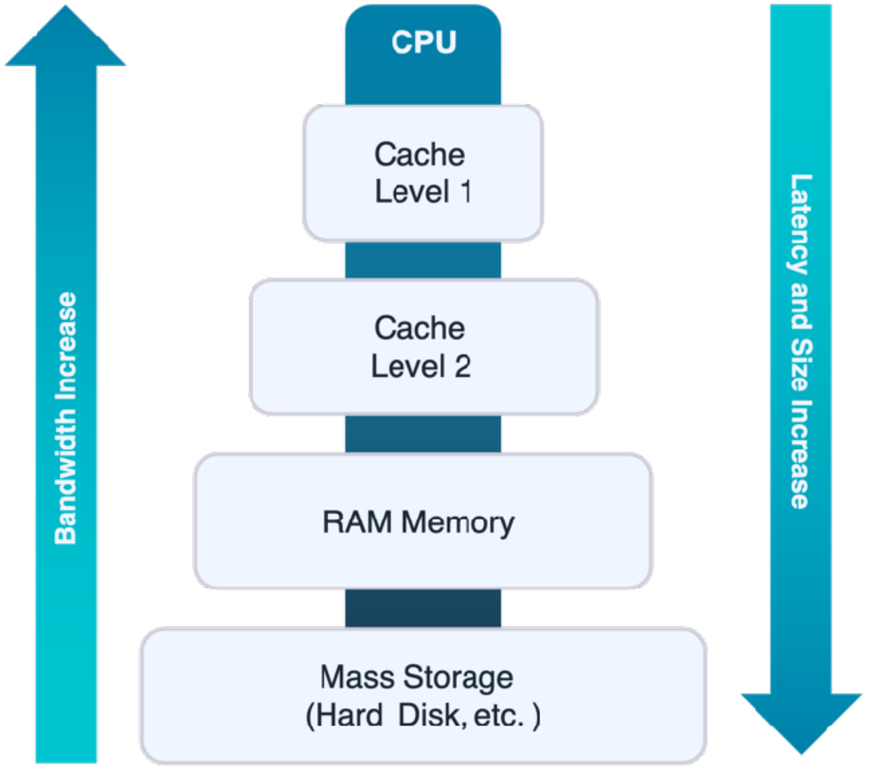
- Refers to the maximum amount of data that can be transferred between the CPU and memory in a given amount of time, usually measured in GB/s

Latency

- Refers to the delay between when the CPU requests data or instructions and when it receives them to start processing

Throughput

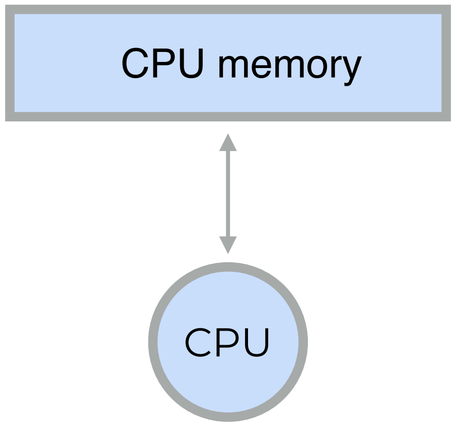
- Refers to the number of processes completed by the hardware in a given per time unit

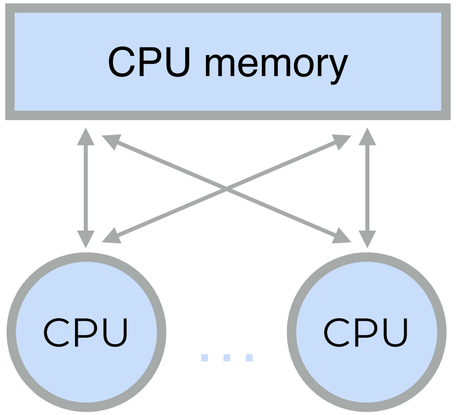


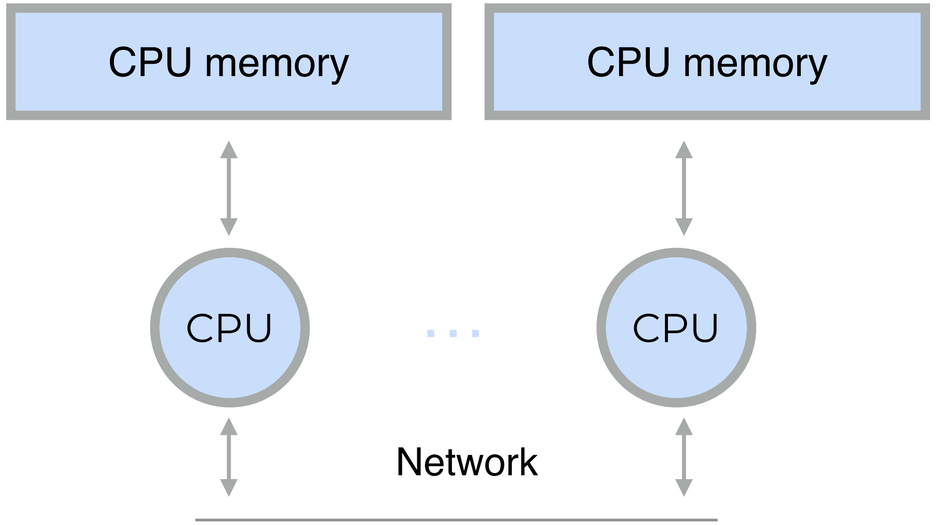
High latency = CPUs sitting idle = wasted resources

HPC System Evolution

Changes in the architecture level also requires changes in the programming models







Single CPU

- Sequential computing
- Performance in Mflop/sec

Multiple CPU

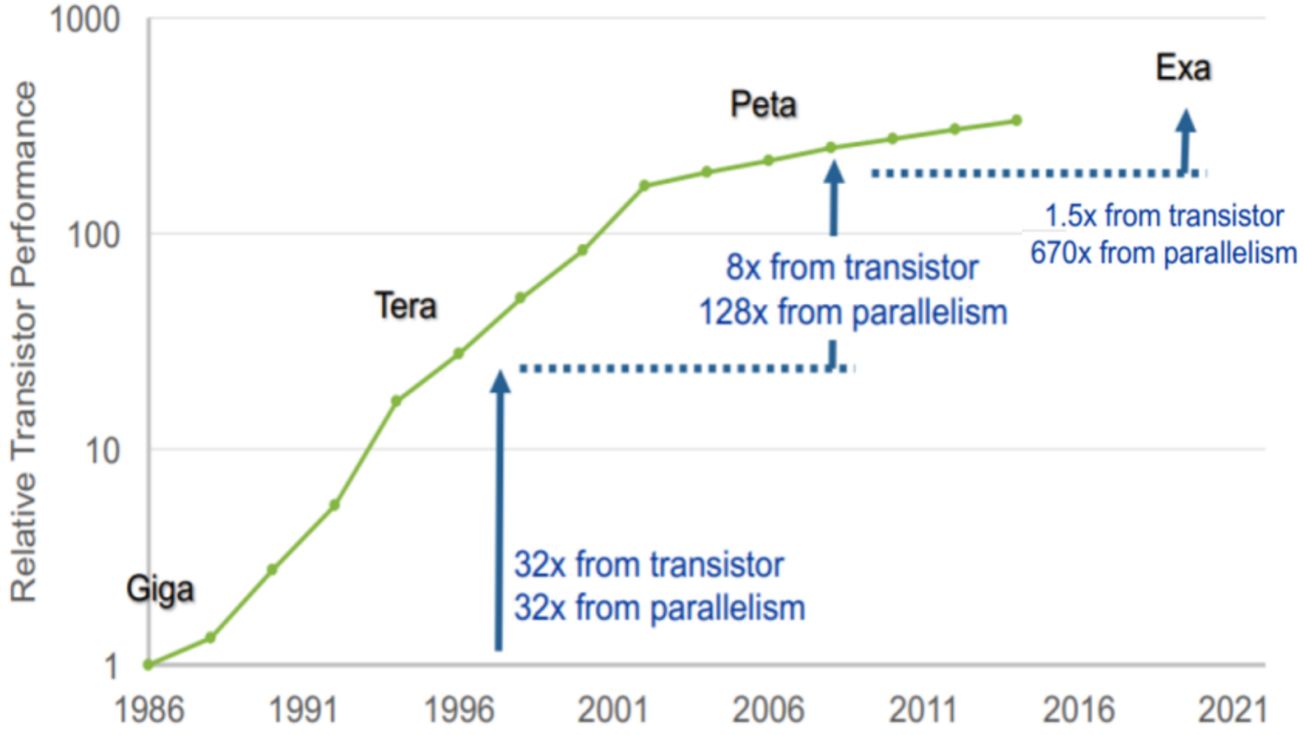
- Shared memory
- Requires change in hardware and software architecture
- New standard: pThread or OpenMP
- Performance in Gflop/sec

Multi-node system

- Connected with high-speed and low latency network (200 GB/s via Infi band network)
- Requires change in hardware and software architecture
- New standard: MPI+OpenMP
- Performance in Tflop/sec

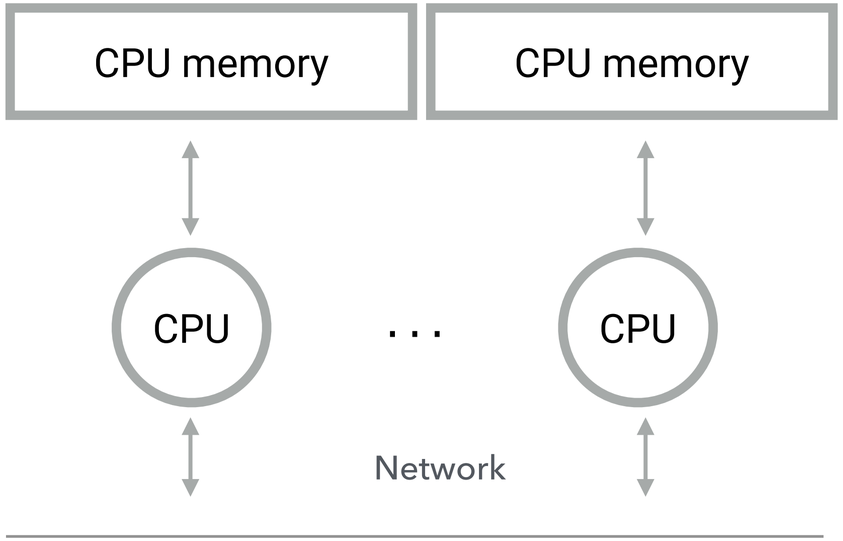
HPC System Evolution

Homogeneous HPC systems: from Giga to Exa and beyond ...



*S. Borkar, Intel

Performance from parallelism



STRENGHTS

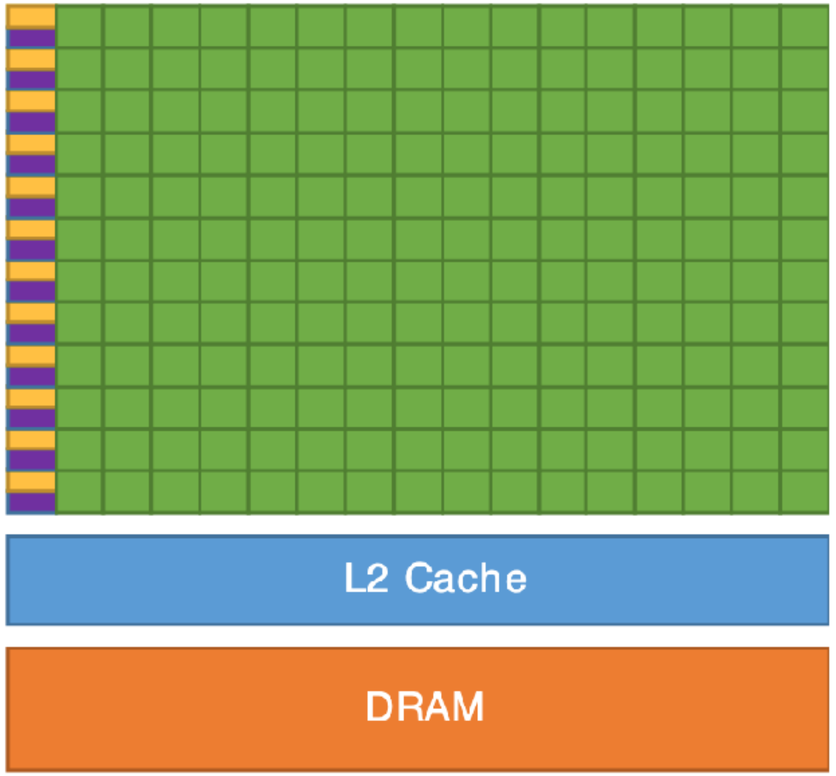
Usability
Flexibility

WEAKNESS

Power hungry

GPUs: Throughput Oriented Design

Called brain of the computer



Simple control

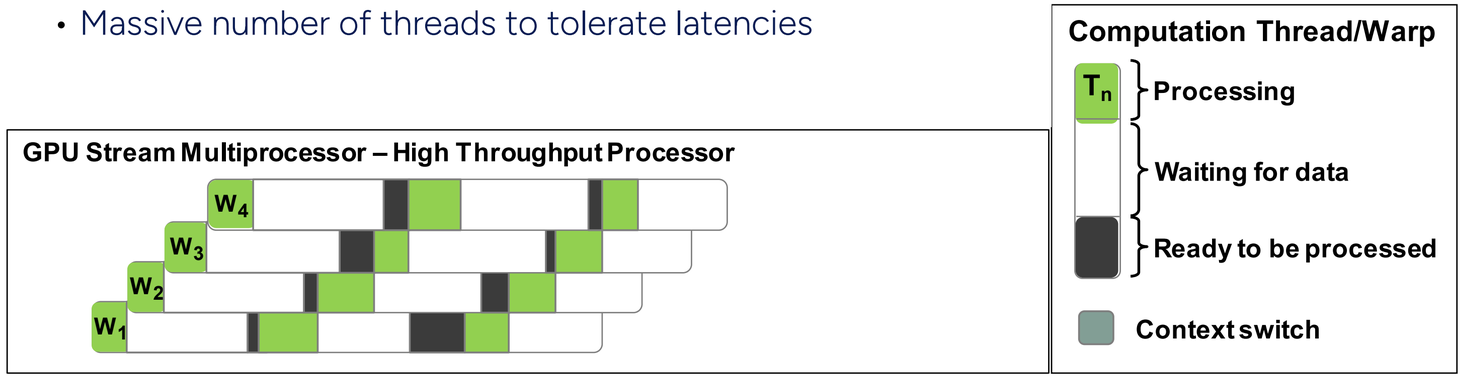
- No branch prediction
- No data forwarding

Small caches

- To boost memory throughput

Energy efficient ALUs

- Many, long latency but heavily pipelined for high throughput
- Massive number of threads to tolerate latencies



Summary: CPUs Vs GPU

The CPU has been called the brain of a PC. The GPU it's soul

CPU: Low Latency Processor

Minimize latency within each thread

Design Focus

- Complex control units
- Large caches (L1, L2, L3)
- Branch prediction
- Out-of-order execution

4-16

Powerful Cores

GPU: High Throughput Processor

Hide latency with massive parallelism

Design Focus

- Simple execution units
- Massive parallelism
- Thread warps/wavefronts
- Context switching

1000s

Simple Cores

Performance Evolution

MFLOPs	GFLOPs	TFLOPs	EFLOPs
Single CPU	Multi-Core	Clusters	Exascale
Millions of operations/sec	Billions of operations/sec	Trillions of operations/sec	Quintillions of operations/sec

Accelerated Computing Has Transformed Supercomputing

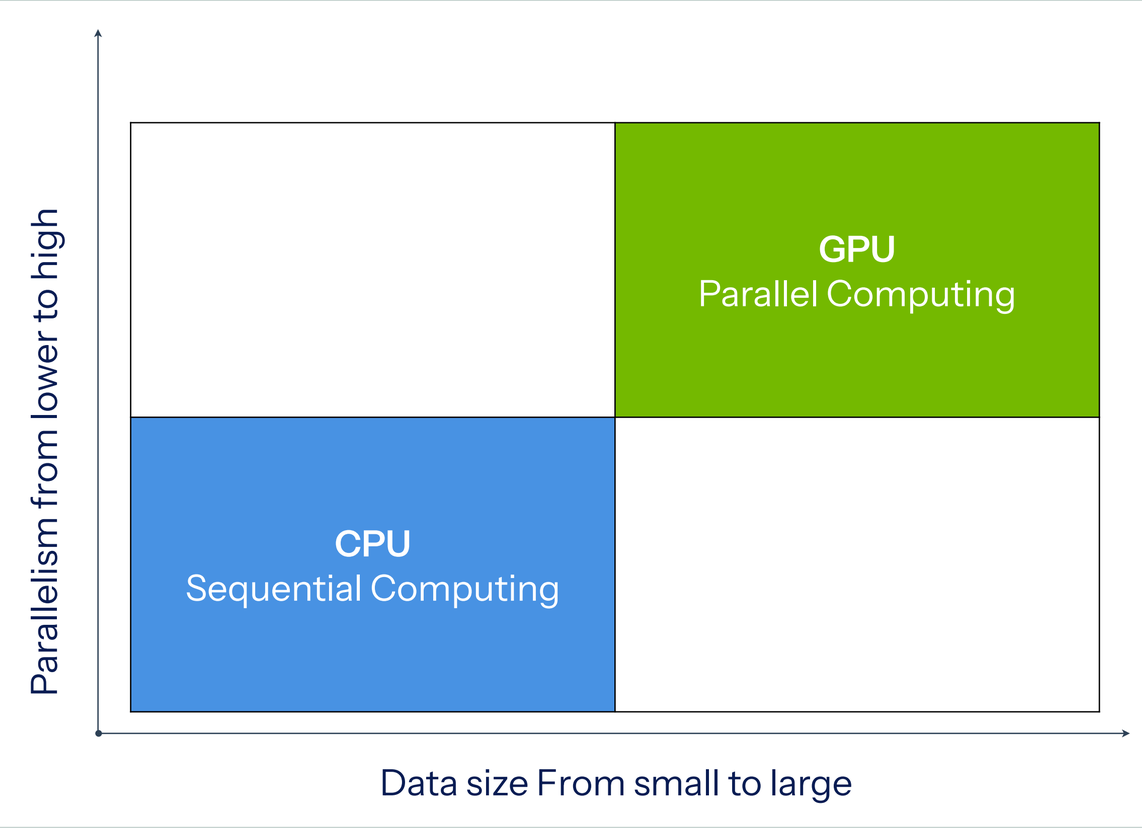
CPUs and GPUs are designed for specific Goal

Task Parallelism

- Multiple independent tasks can run simultaneously, distributing functions across multiple cores
- Could be copies of the same program at different positions in the code

Data Parallelism

- A single program running across multiple threads/processors
- All threads execute the same operation on different elements of data in lockstep



Latency Vs. Throughput

Latency – Time between your first request and the data arrives

Bandwidth – How much data you get in a given time once the transfer starts

Low Latency (left)

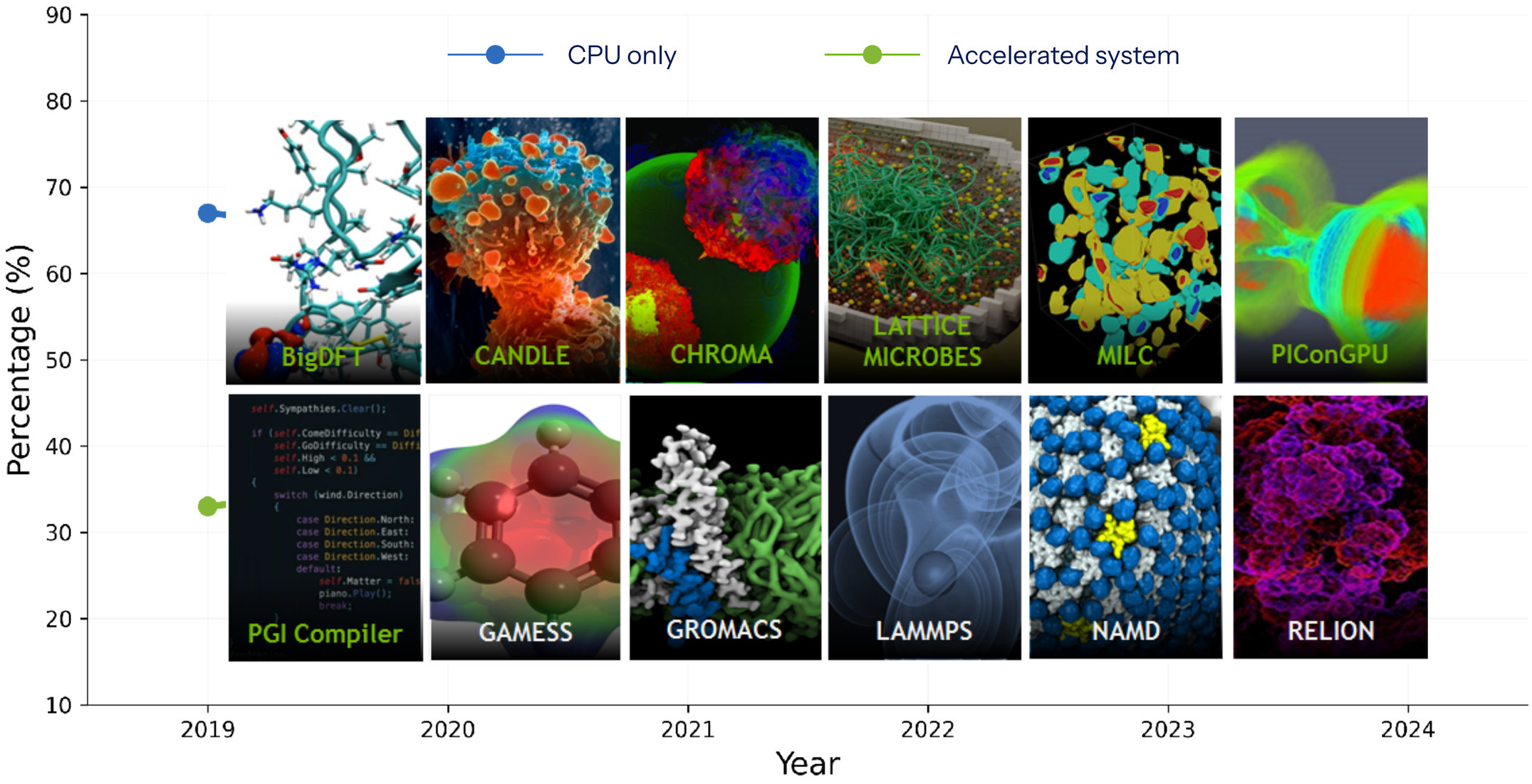
Or

High bandwidth (right)

GPUs have very high bandwidth compared to CPUs (3.3TB/s vs 0.5 TB/s)

But also have higher latency than CPUs (400ns vs 100ns)

Accelerated Computing Has Transformed Supercomputing



EURO HPC SYSTEM

Exascale

Pre-exascale

Petascale / mid-range

Quantum system / simulator

The Next Generation of HPC

Combining specialized processors to achieve unprecedented performance per watt

CPU Partition

Optimized For:

- Complex control flow
- Sequential algorithms
- Irregular memory access
- System orchestration

High-level coordination

GPU Partition

Optimized For:

- Massively parallel workloads
- Regular computation patterns
- High arithmetic intensity
- SIMD operations

Computational heavy lifting

Specialized Accelerators

Examples:

- AI/ML accelerators (TPU)
- Quantum processors
- Neuromorphic chips
- FPGA accelerators

Domain-specific optimization

NVIDIA Ampere Architecture

PCIe interface

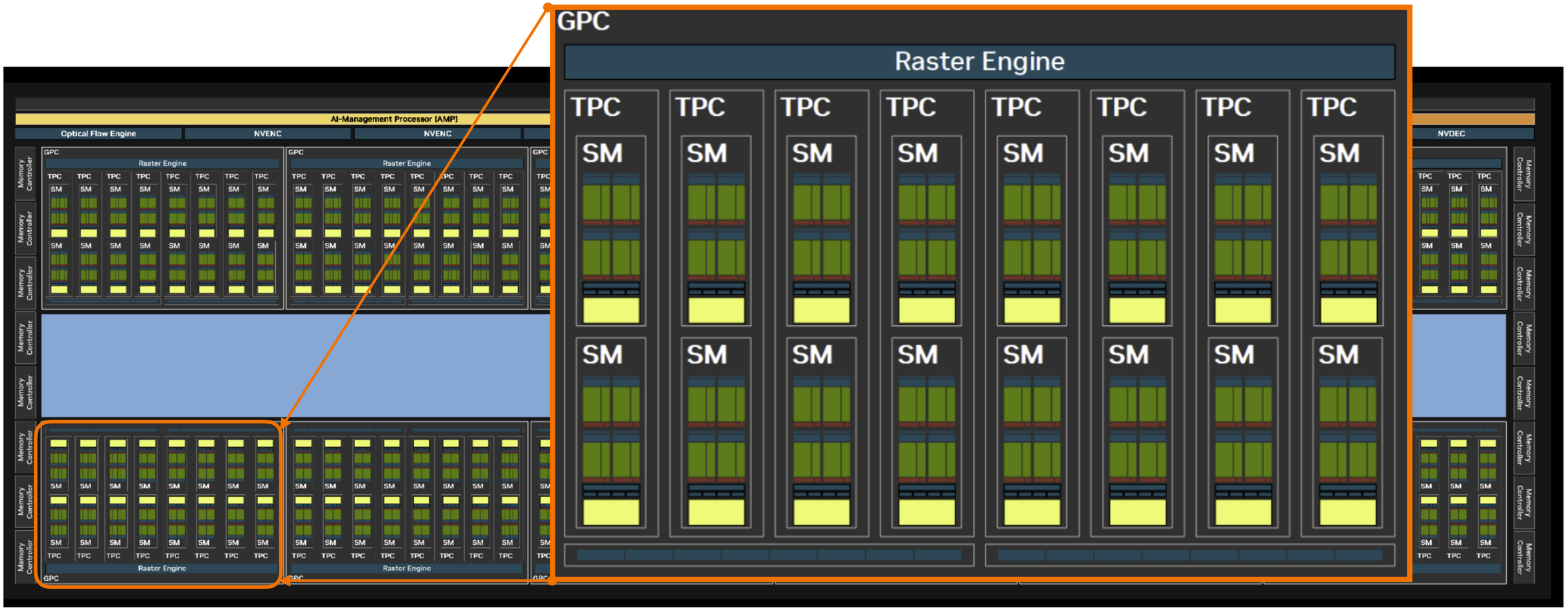
Memory controller

Graphics Processing Cluster

Streaming Multiprocessor

NVLink interface

Graphics Processing Cluster



For RTX 5090 with 11 GPCs, we have:
170 SMs
680 Tensor Cores
170 RT Cores
For a total of 21,760
Paired with 1.7 TB/sec Memory

The NVIDIA Ampere GPU Architecture

SMs	108
Total threads	221,184
Peak FP32 TFlops/s	19.5
Peak FP64 TFLOP/S (non-tensor)	9.7
Peak FP64 TFLOP/S (tensor)	19.5
Tensor Core Precision	FP64, TF32, BF16, FP16, I8 I4, B1
Shared Memory per SM	160 kB
L2 Cache Size	40960 kB
Memory Bandwidth	1555 GB/sec
GPU Boost Clock	1410 MHz
NVLink Interconnect	6000 GB/sec

ASCI white supercomputer Lawrence Livermore National Laboratory Top500 #1 in 2001, 7.9 Teraflop/s

One Level Deeper – Streaming Multiprocessor (SM) in NVIDIA Ampere Architecture”

Composition

- The Streaming Multiprocessor (SM) is the main processing unit.
- CUDA cores handle basic arithmetic and bitwise operations.
- SMs contain Arithmetic Logic Units (ALUs) and memory (e.g., registers, shared memory).
- Special Function Units (SFUs) manage complex functions like square roots and trigonometry.
- SMs may also include units for texture processing, ray tracing, and tensor operations.
- All threads in an SM share the same L1 cache, which can also act as shared memory.
- SMs execute blocks of up to 2,048 threads.

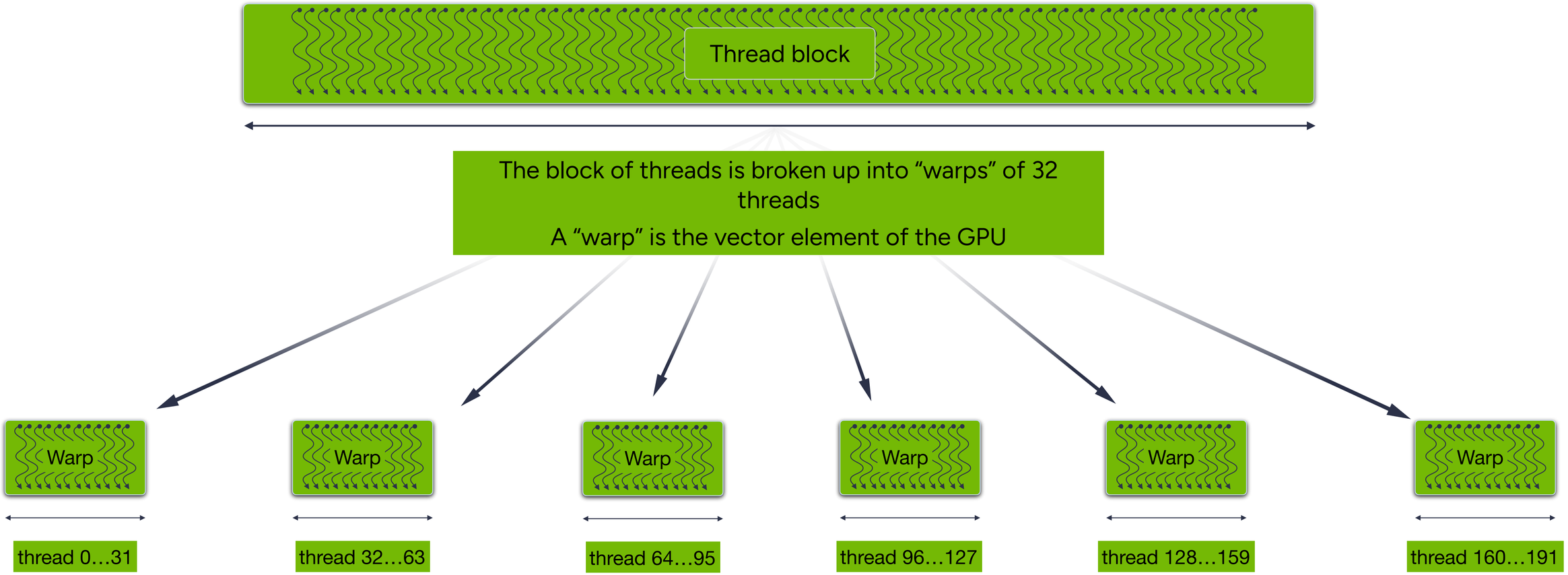
Instruction Parallelism

- Each **SM** has four execution units, each handling a warp of 32 threads using CUDA cores
- **SIMT (Single Instruction, Multiple Threads)**: All threads in a warp execute the same instructions in parallel.
- Unlike strict **SIMD**, **SIMT** allows threads to diverge (e.g., due to conditional branching)

Control Logic

- The SM contains scheduling hardware to manage the execution of multiple warps and switches between them to hide memory latency

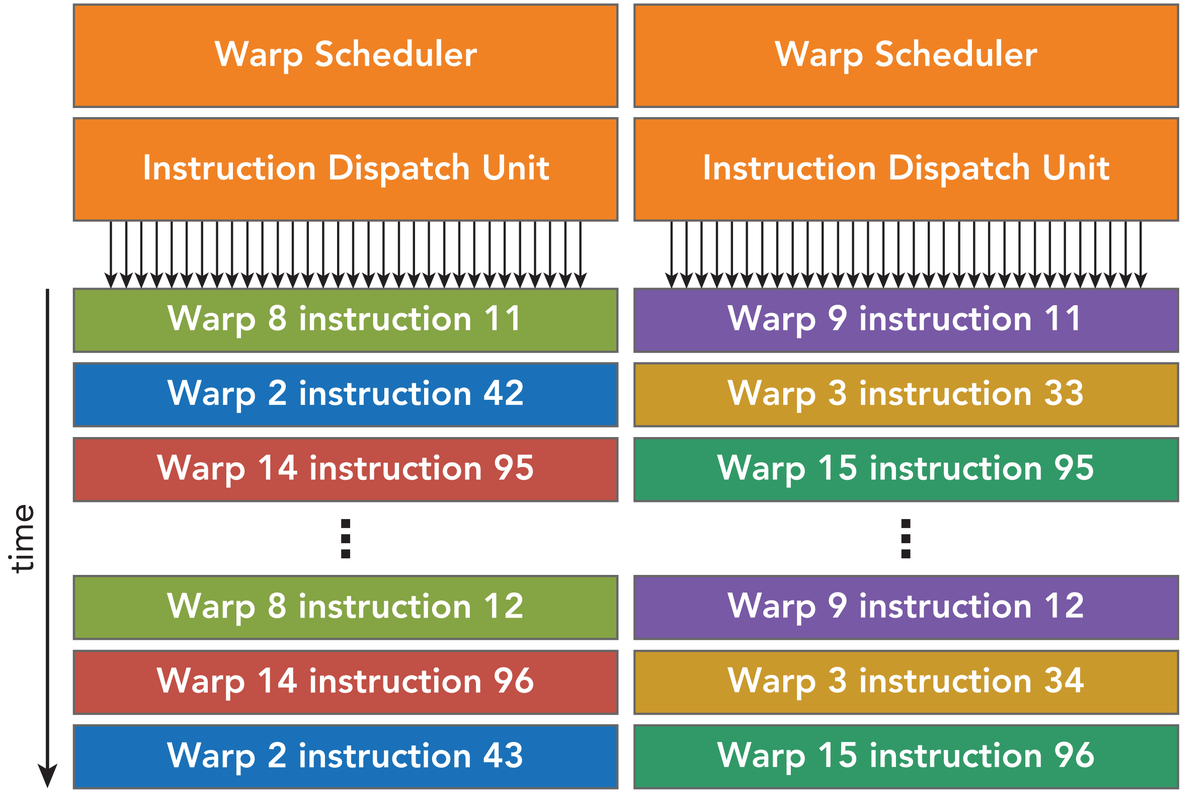
CUDA Launches Arrays of Parallel Threads



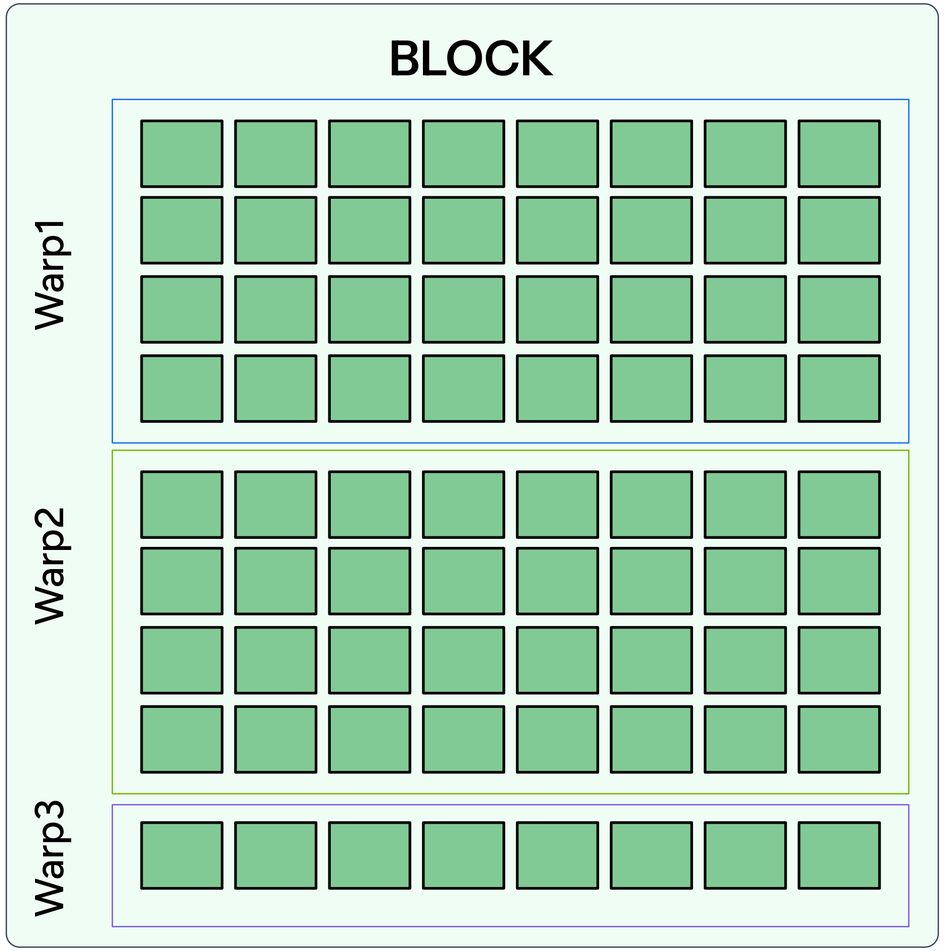
Warps as Scheduling Units

Hardware Multithreading

- NVIDIA SM schedules threads in warps (groups of 32 threads)
- Warp simply means a group of threads that are scheduled together to execute the same instructions in lockstep.
- Execution contest stays on chip
- No overhead for switching warps
- Volta SM has 4 warp schedulers, each one is responsible for
  - feeding 32 CUDA cores
  - 8 load/store units
  - 8 special functions unit



CUDA Launches Arrays of Parallel Threads



Ways to Parallels an Applications on Nvidia GPUs

Libraries

Drop-in Acceleration

- Minimal code changes
- Pre-optimized functions
- Quick implementation

OpenACC/OpenMP

Directives

Easy Accelerations

- Pragma-based approach
- Compiler-guided optimization
- Moderate learning curve

CUDA

Maximum Flexibility

- Full hardware control
- Custom kernel development
- Maximum performance potential

Development Trade-offs

Productivity
Fast development

Portability
Cross-platform

Performance
Maximum speed

Choose your approach based on project requirements and development constraints

Fundamentals of CUDA Programming

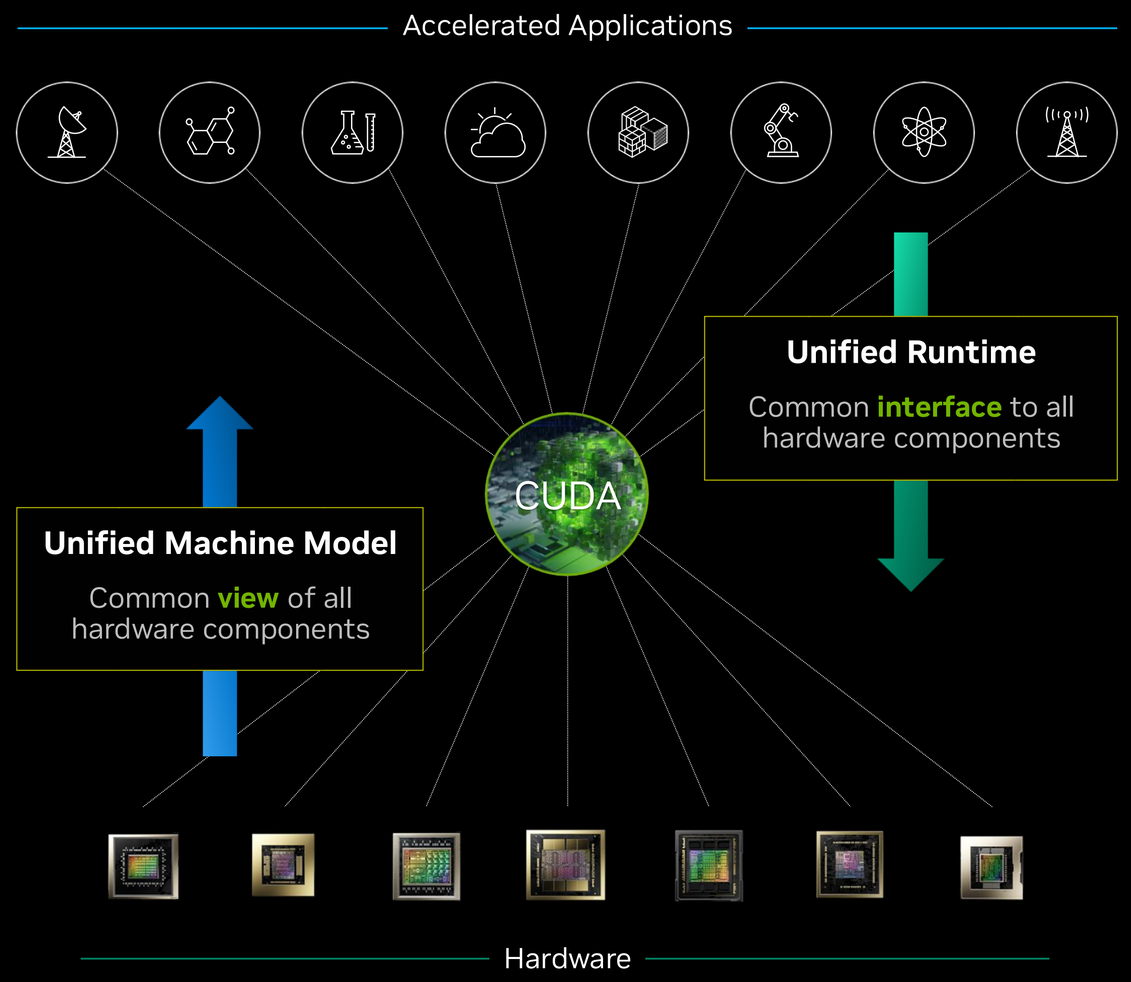
Memory Model, Thread Hierarchy & Advanced Streaming

- CUDA Programming Model
- Memory Hierarchy and Management
- Thread Hierarchy
- Error Handling
- Performance Optimization
- Non-Default Streaming

Compute Unified Device Architecture

Two key ingredients: the Machine Model and the Runtime

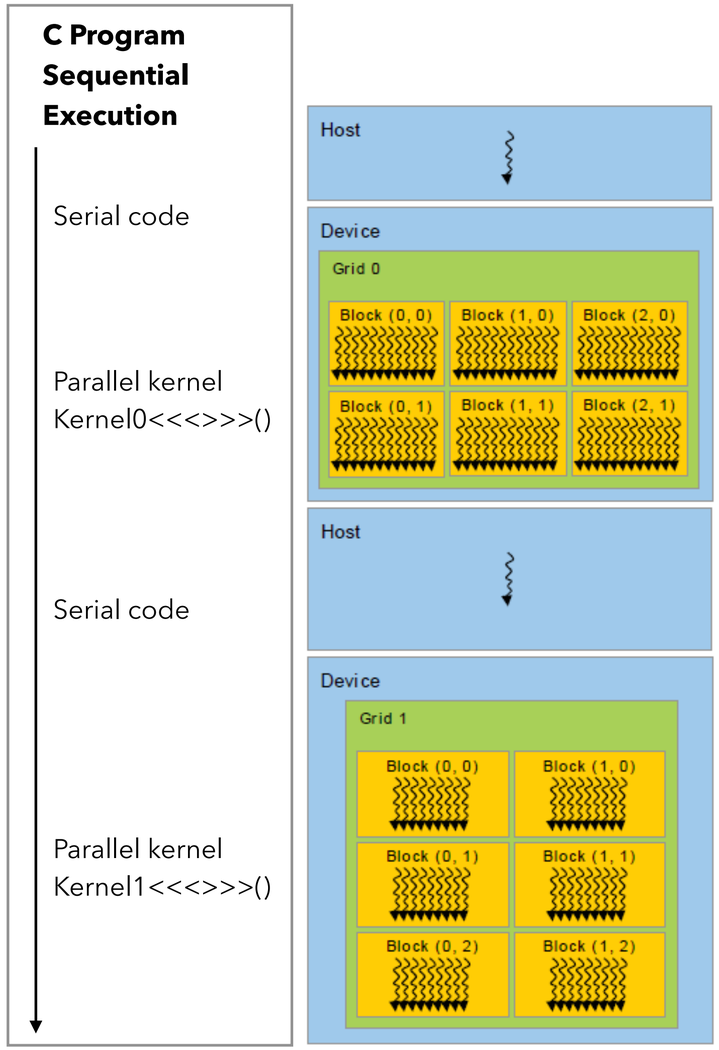
- Frameworks & DSLs
- SDKs
- Domain-Specific Libraries
- Accelerated Libraries
- Communication Libraries
- Device Libraries
- Kernel Authoring
- Compiler Stack
- Host Runtimes & Tools



CUDA Programming Model Overview

Understanding the fundamental of CUDA

1. Host (CPU) and Device (GPU) execution model
2. Kernel functions execute on GPU
3. Massive parallel thread execution
4. Single Instruction Multiple Thread (SIMT)



Execution Policy Vs Specifier

Declaring CUDA functions

1. **__global__**
   Kernel function, called from host, executed on device

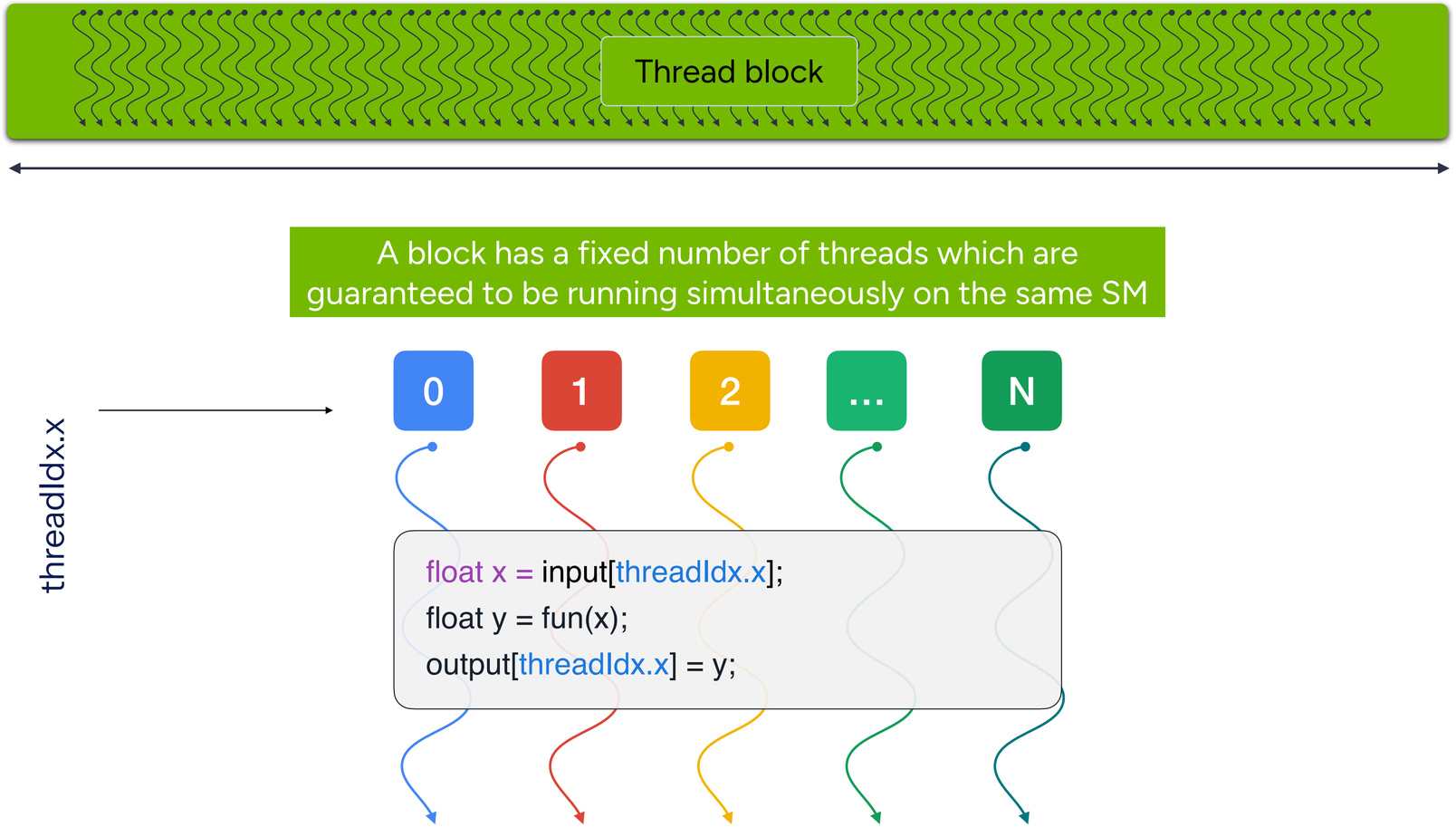
2. **__device__:**
   Device function, called from device, executed on device

3. **__host__**
   Host function (default), called from host, executed on host

4. **__host__ __device__**
   Can be compiled for both host and device

CUDA Launches Arrays of Parallel Threads

Declaring CUDA functions



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

- All threads in a grid run the same kernel code
- Each thread has a unique ID: threadIdx.x
- Threads are similar to data-parallel tasks
- **Execution Configuration:** <<<numBlocks, numThreads>>>(arguments)
- **Synchronisation:** Launching kernel is asynchronous
- **cudaDeviceSynchronize():** wait until device code completeness

Device Function Example

Keyword __global__ void

- Indicates that the function will run on GPU.
- can be invoked globally either from CPU or GPU
- CPU refers as HOST and GPU refers as DEVICE
- called from the host code but run on device

Execution configuration <<<>>>

- Function call to run on GPU
- Must provide an execution configuration with syntax <<< >>> prior to passing kernel argument
- Specify the thread hierarchy for a kernel launch
  <<<nB, NTpb>>>

Synchronization

- Launching kernels is asynchronous CPU code will continue to execute without waiting for kernel to be completed
- cudaDeviceSynchronize() will for the most to wait until device code completeness

```
__global__ void GPUFunction()
{
  printf("function runs on the GPU.\n");
}

int main()
{
  CPUFunction();
  GPUFunction <<<1, 1>>>();
  cudaDeviceSynchronize();
}
```

GPU Computing Is About Massive Parallelism

How do we run code in parallel on the device?

```
onGPU<<<1, 1>>>();
```

Instead of executing once, execute N times in parallel

```
onGPU<<<1, N>>>();
```

How to complete the CUDA enable code?

GPU Computing Is About Massive Parallelism

CUDA Driver

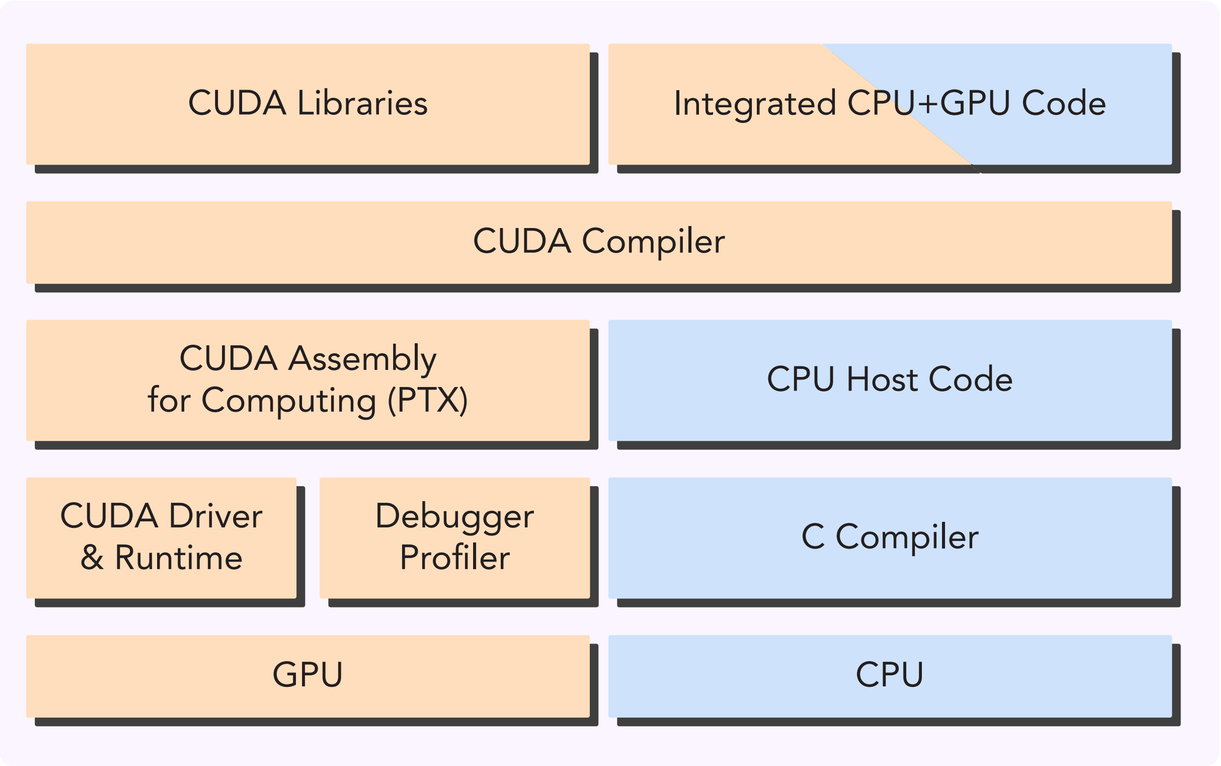
A critical piece of software that acts as the interface between your application and the NVIDIA GPU hardware

The CUDA Toolkit

NVHPC Compiler: translate CUDA into optimised machine instructions for NVIDIA GPUs

Libraries: Comprehensive libraries like cuBLAS and cuDNN are provided

Debugging tools: robust debugging tools



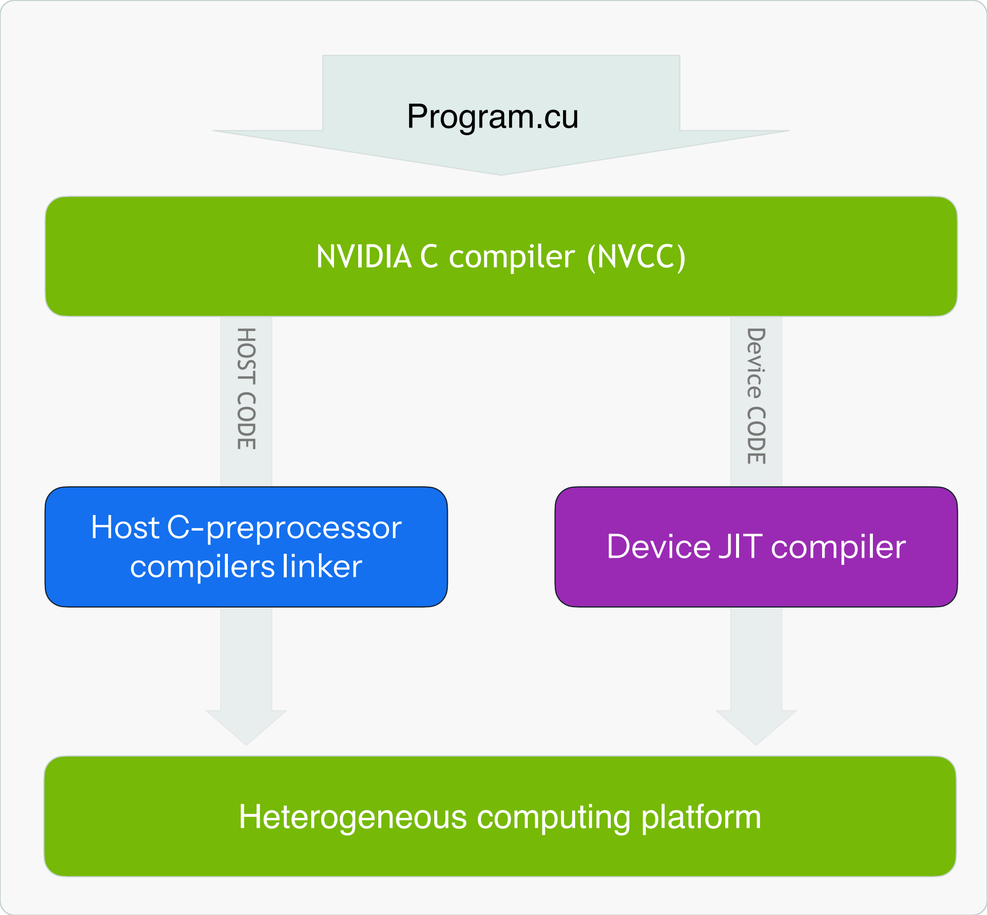
GPU Computing Is About Massive Parallelism

Compilation process

Code for host and device in some.cu file

CUDA compiler separates source code into host and device components

Based LLVM open source compiler infrastructure



nvcc -arch=sm_80 -o out some-CUDA.cu -run

- arch: indicates for which architecture the files must be compiled (sm_80 is for TESLA A100 GPU)
- run: execute the successfully compiled binary
- Information on CUDA device: nvidia-smi, deviceQuery

Exercise-1: Accelerating loops

Serial CPU code

- Defines a function `loop(int N)` that performs a loop from 0 to N - 1
- Uses a `for` loop to iterate `N` times
- Prints the current iteration number using `printf`
- Calls `loop(int N)` to print 10 iteration messages

Things to do

- Declare kernel
- Launch the kernel
- Benchmark: no of threads= 1, 32, 256

// CPU function

```
#include <stdio.h>

void loop(int N)
{
  for (int i = 0; i < N; ++i)
  {
    printf("This is iteration number %d\n", i);
  }
}

int main()
{

  int N = 10;
  loop(N);
}
```

Device function

Device Function Examples

```
// Device function - helper function on GPU
__device__ float computeDistance(float x1, float y1, float x2, float y2) {
    float dx = x2 - x1;
    float dy = y2 - y1;
    return sqrtf(dx * dx + dy * dy);
}

// Kernel using device function
__global__ void calculateDistances(float* x1, float* y1,
                                   float* x2, float* y2,
                                   float* distances, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) {
        distances[idx] = computeDistance(x1[idx], y1[idx],
                                         x2[idx], y2[idx]);
    }
}

// Host function with same signature
__host__ __device__ float clamp(float value, float min_val, float max_val) {
    return fmaxf(min_val, fminf(max_val, value));
}
```

What the Program does

Generates N (1 million) random 2D points:

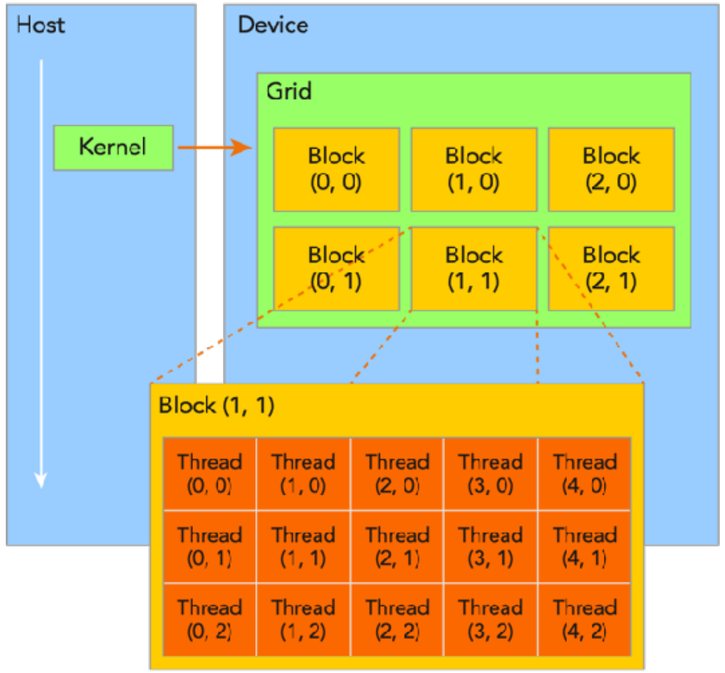
- start points: `x1[i], y1[I]`
- end points `x2[i], y2[i]`

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

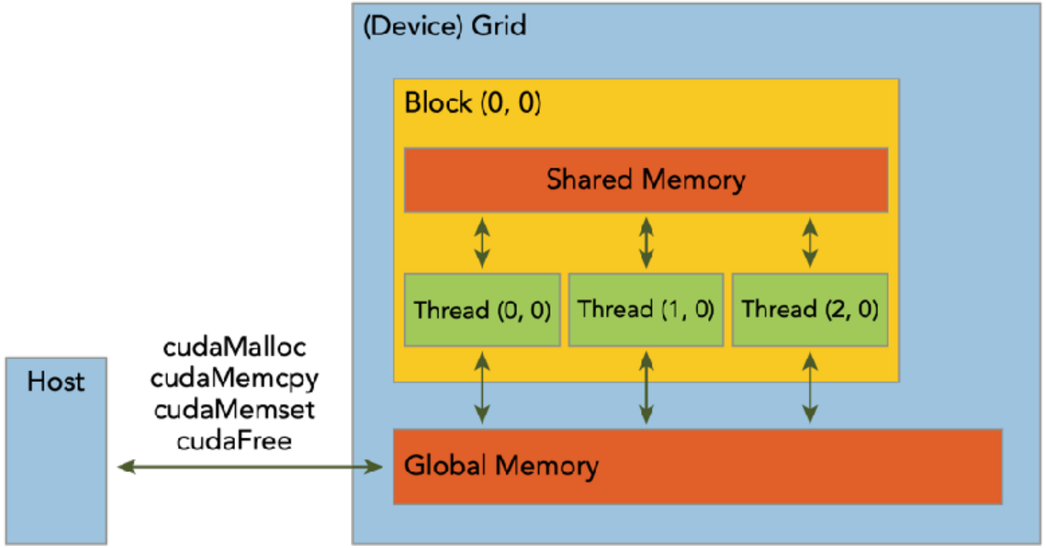
GPUs Serve as a Co-Processor, Not a Standalone Platform

CUDA enhances your control over memory and thread hierarchies, optimizing execution and scheduling with

Thread hierarchy structure



Memory hierarchy structure



SIMT VS. SIMD Execution Model

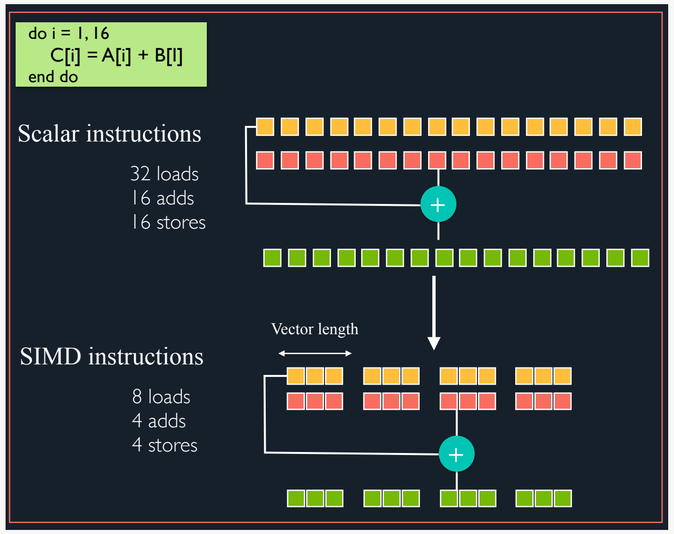
Both SIMD and SIMT achieve parallelism by broadcasting a single instruction to multiple execution units

Consider how computations will be distributed between threads for the following loop (N >> threads count):

```
float *A, *B, *C = …. ; for (int I = 0; I <N; I++ ) A[I] = B[I] + C[I]
```

SIMD: “One instruction, one data chunk.”

- A single instruction operates on multiple data elements simultaneously
- Requires wide vector units in hardware (e.g., 4 or 8 lanes)
- A SIMD register (or a vector register) can hold many values (2 - 16 values or more) of a single type
- Operates on packed data in wide registers (e.g., 128-bit or 256-bit)
- All elements in the vector must follow the same control flow—no divergence
- Vectorisation helps you write code which has good access patterns to maximise bandwidth



CUDA Launches Arrays of Parallel Threads

Both SIMD and SIMT achieve parallelism by broadcasting a single instruction to multiple execution units

SIMT: "One instruction, many threads."

- Uses scalar execution units, not wide vectors, rigid, lockstep vector processing
- 32 threads in a warp share a single instruction fetch, executed over multiple cycles (e.g., 4 cycles on 8 CUDA cores)
- Flexible, thread-level parallelism with divergence support.
- Single instruction, multiple flow paths

  if statements are allowed!

SIMT allows

- CUDA GPU to perform "vector" computations on *scalar cores*
- *Much easier to vectorise than getting compiler to autovectorize on CPU*

https://yosefk.com/blog/simd-simt-smt-parallelism-in-nvidia-gpus.html

SIMT thread registers			
a[I]	a[I+1]	a[I+2]	a[I+3]
b[I]	a[I+1]	b[I+2]	b[I+3]
a	a	a	a
b	b	b	b
I	I+1	I+2	I+3
…	…	…	…

Why Do We Need To Have So Many Warps in an SM?

Both SIMD and SIMT achieve parallelism by broadcasting a single instruction to multiple execution units

Latency hiding

- **Memory Access Latency:** Multiple warps can hide memory access latency by switching to another ready warp when one warp is waiting for data
- **Instruction Pipeline Latency:** Keeps the execution units busy while other warps are stalled due to dependencies or resource constraints

Resource Utilisation

- **Maximizing Throughput:** More warps allow for better utilization of SM resources (ALUs, memory bandwidth)
- **Load Balancing:** Distributes the workload evenly across the available execution units

Parallelism

- **Enhancing Parallel execution:** Multiple warps increase the parallelism, enabling more threads to be processed concurrently
- **Improved Performance:** Higher parallelism leads to better performance and throughput for data-intensive applications

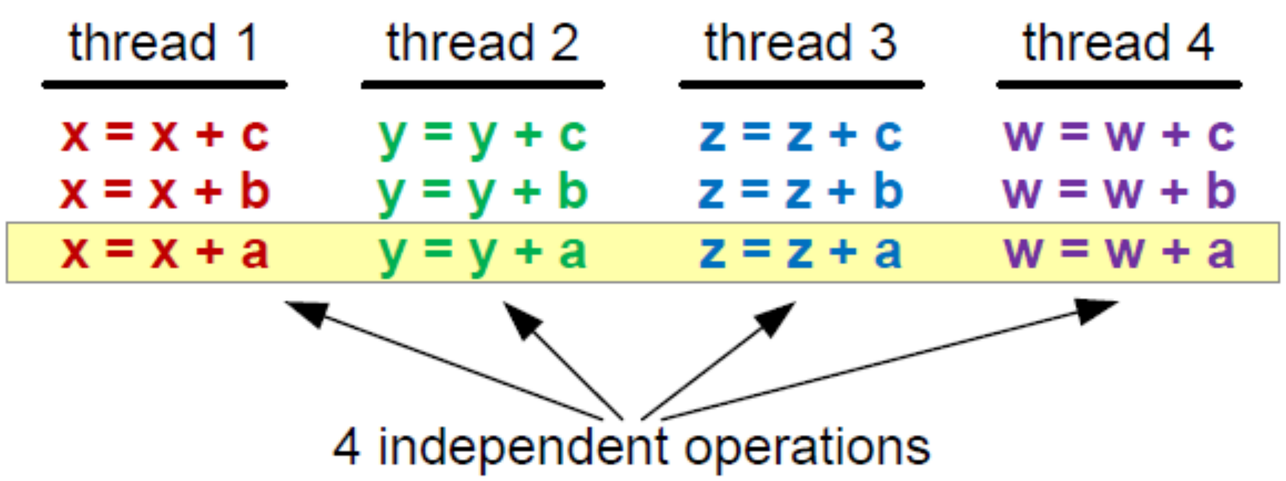
Latency Can Be Hidden in Two Possible Ways

- The code need to be organised to **provide the scheduler a sufficient number of independent operations**
  - i.e. the more warps are available, the more context-switch can hide latency
  - And therefore proceed with other useful operations

- There are two possible ways and paradigms to use (**can be combined too!**)
  - Thread-Level Parallelism (TLP)
  - Instruction-Level Parallelism (ILP)

More Concurrently Eligible Threads

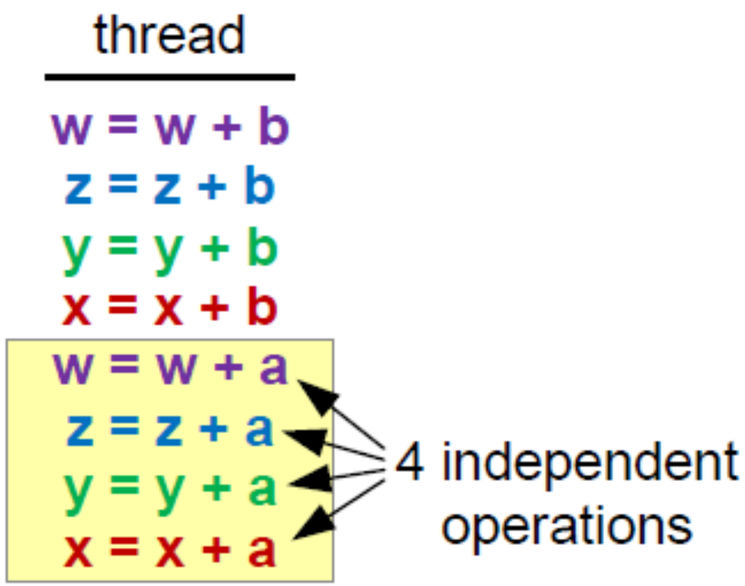
- Thread-level parallelism (TLP)
  - Strive for high SM occupancy: provide as much threads as SM as possible (when a scheduler is free it will find easily a warp to execute)
  - Best approach for low number of independent operations per CUDA kernel



More Concurrently Eligible Threads

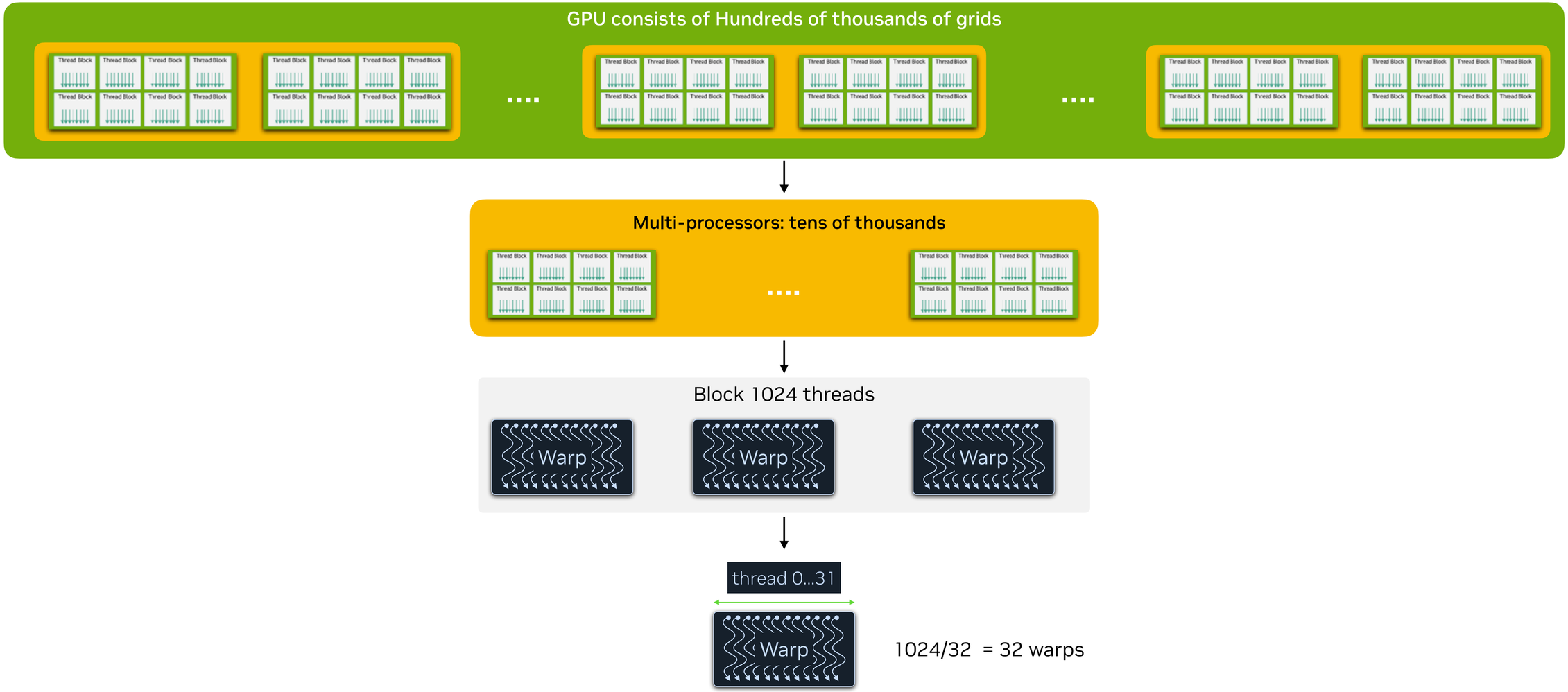
- **Instruction-level parallelism (ILP)**
  - High-number of multiple independent operations inside CUDA kernel (each kernel act on a lot of data)
  - This will grant the scheduler to stay on the same warp and fully load each hardware pipeline

- **NOTE**
  - The scheduler will not select a new warp until there are eligible instructions ready to execute on the current warp



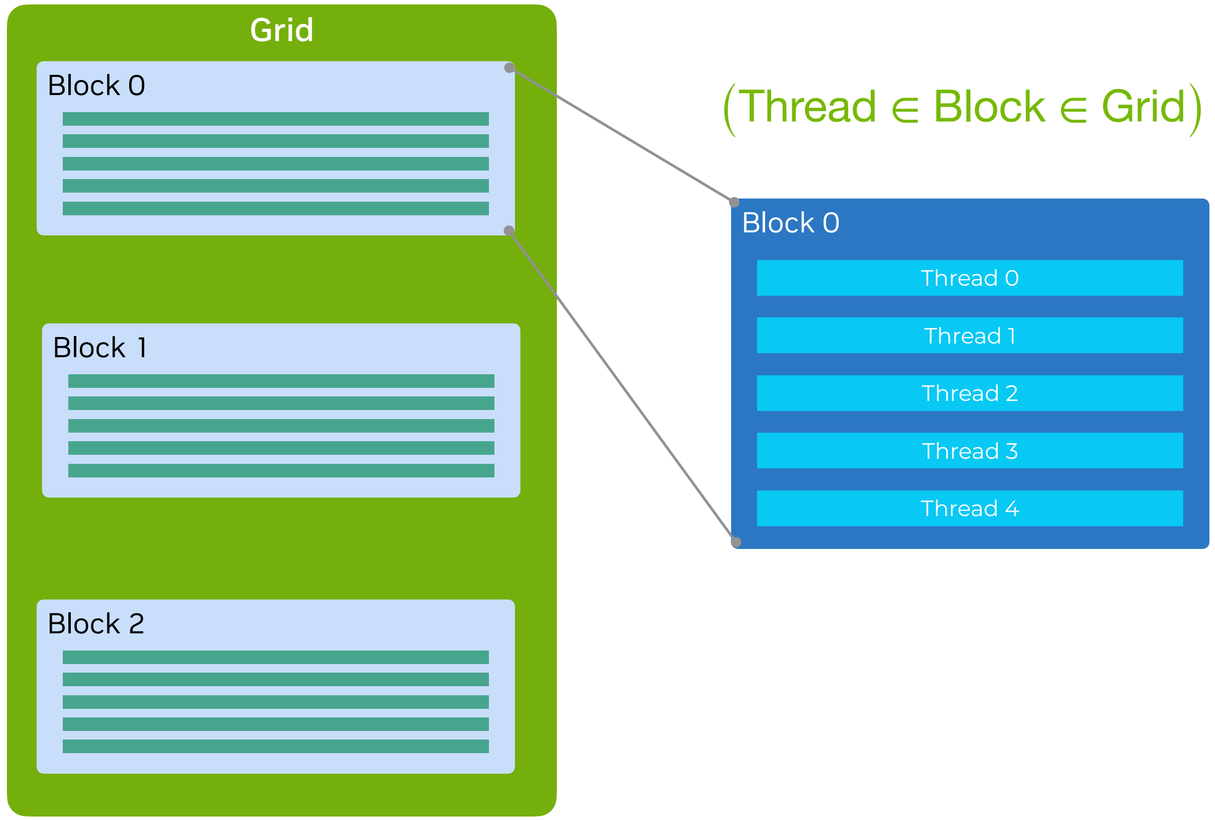
GPU Thread Hierarchy

CUDA launches arrays of parallel threads



Kernel Execution Across Grid, Block and Thread

CUDA launches arrays of parallel threads

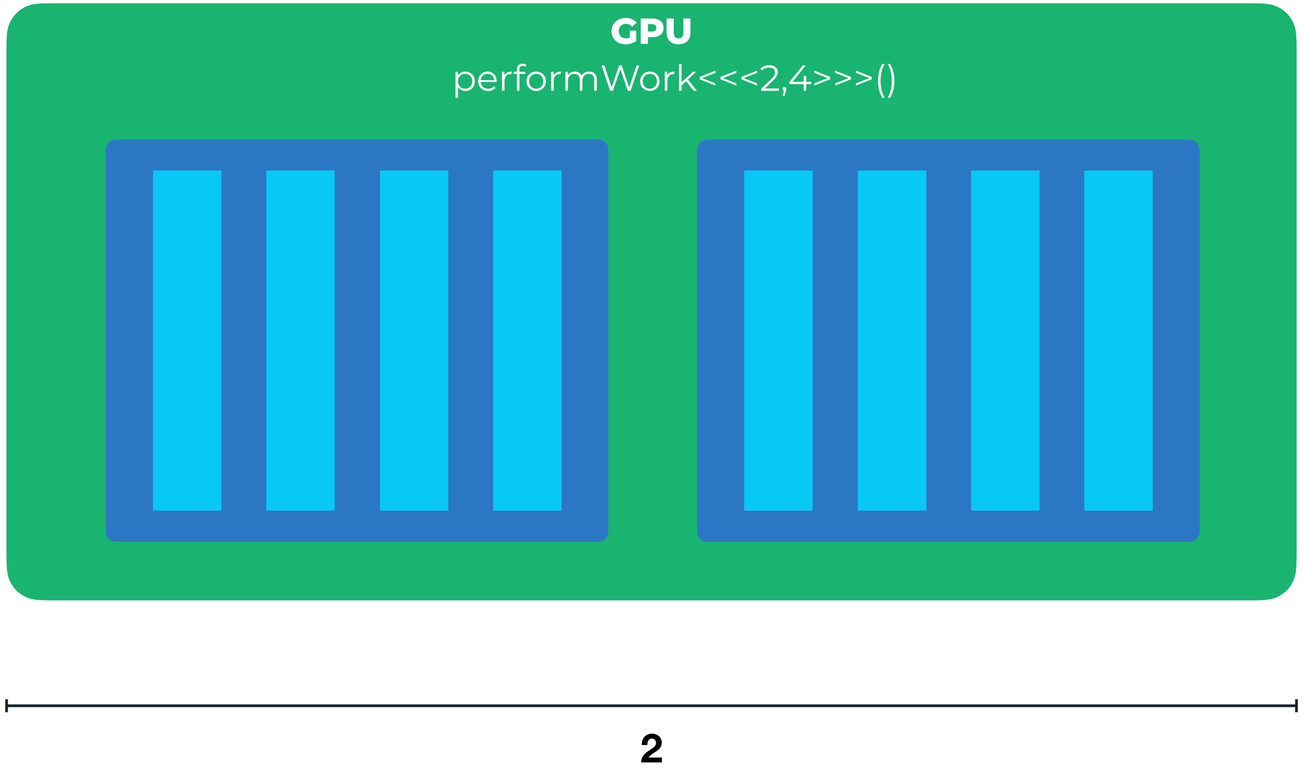


- In order to compute N elements on the GPU in parallel, at least N concurrent threads must be created on the device
- GPU threads are grouped together in teams or blocks of threads
- Threads belonging to the same block or team can cooperate togheter exchanging data through a shared memory cache area
- Each block of threads will be executed independently
- No assumption is made on the blocks execution order

Kernel Execution Across Grid, Block and Thread

CUDA launches arrays of parallel threads

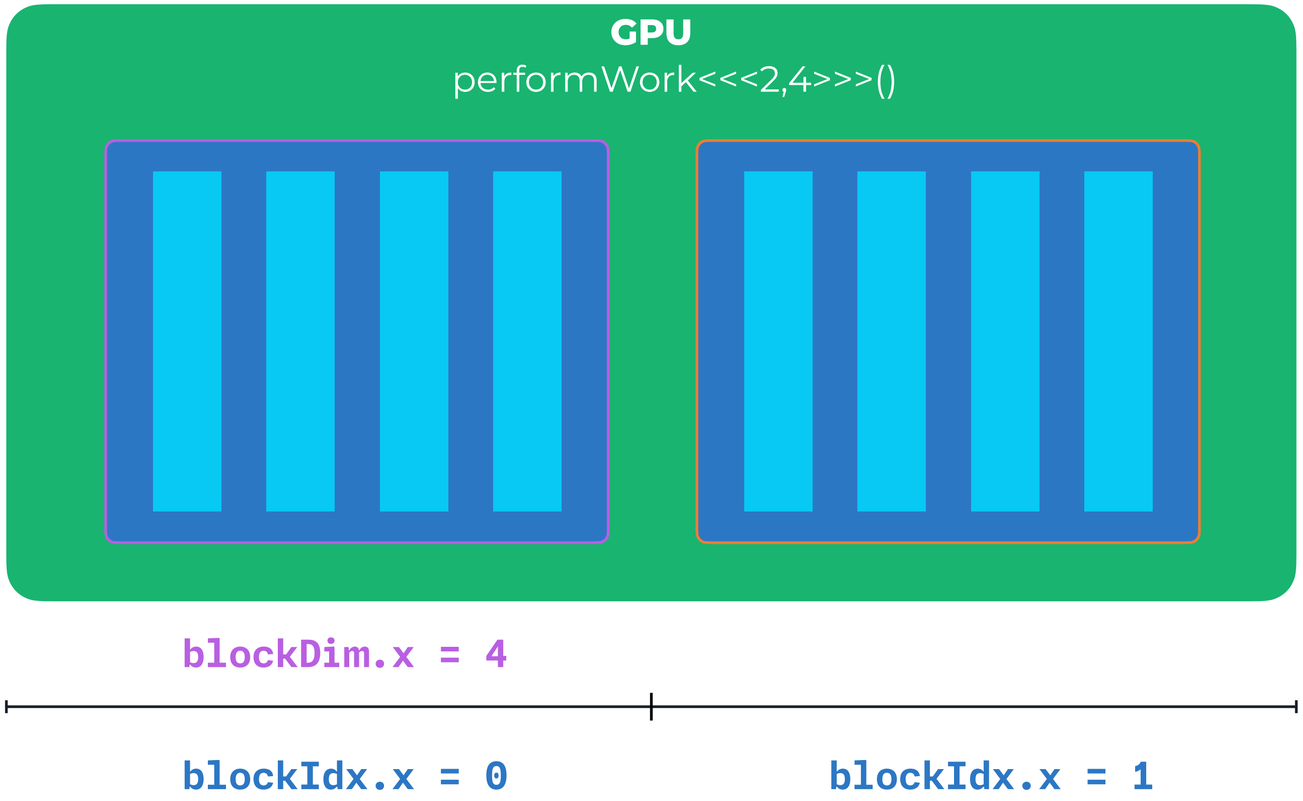
`gridDim.x:` number of blocks in the grid, in this case 2



Kernel Execution Across Grid, Block and Thread

CUDA launches arrays of parallel threads

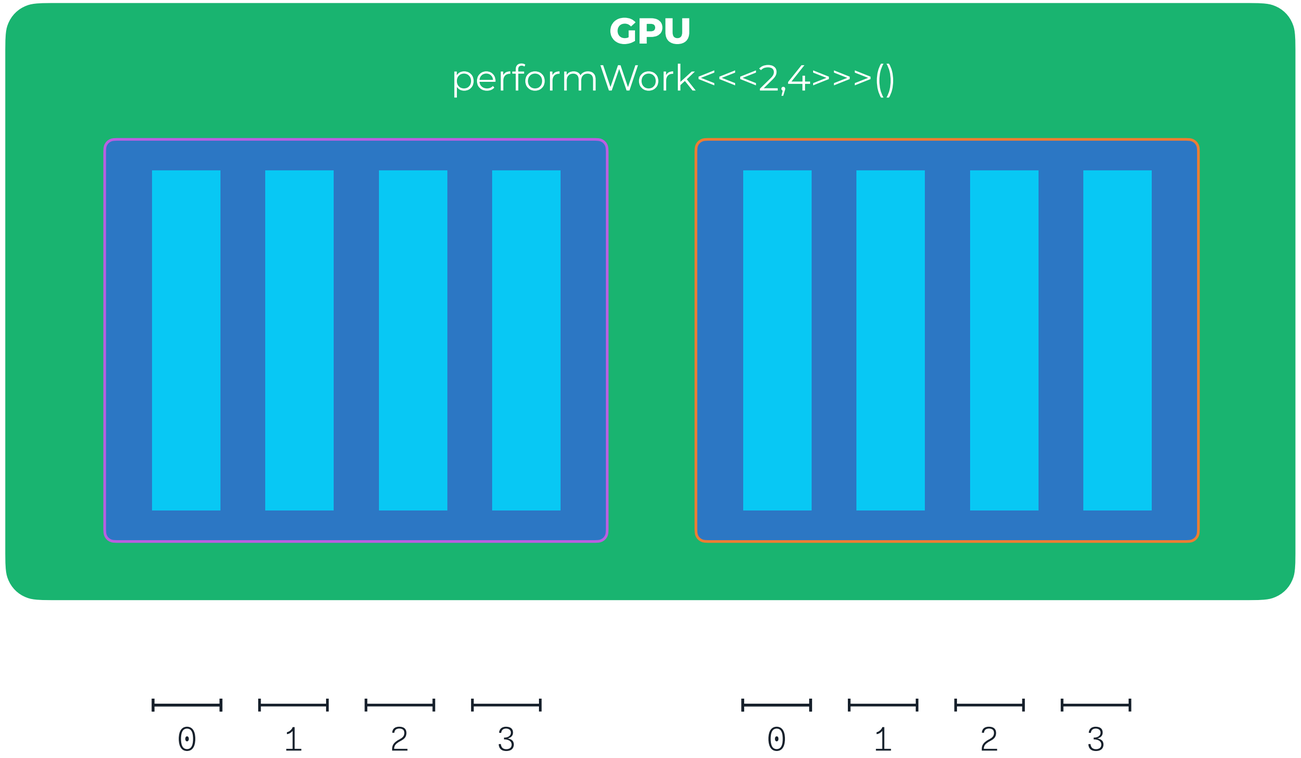
`blockIdx.x`: index of a blocks in a grid
`blockDim.x`: number of threads per block



Kernel Execution Across Grid, Block and Thread

CUDA launches arrays of parallel threads

`threadIdx.x:` index of the thread with a block

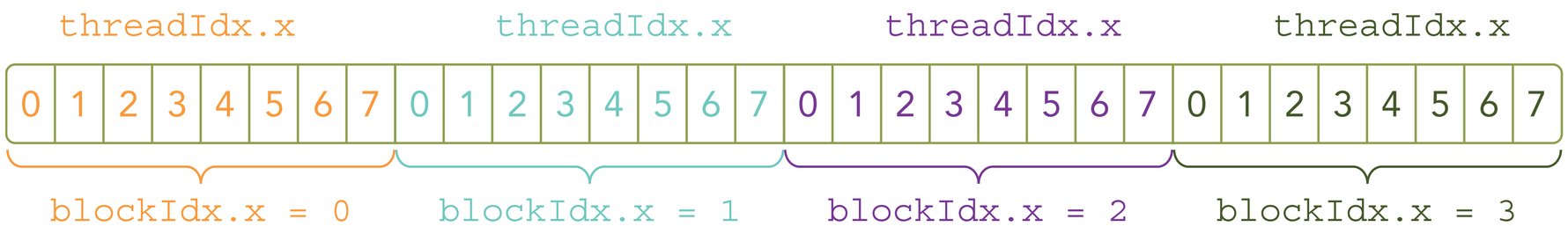


Kernel Execution Across Grid, Block and Thread

CUDA launches arrays of parallel threads

Choose the optimal block size

- A **limited number of threads** (**1024**) can fit inside a thread block
- To increase parallelism, we need to **coordinate** work **among thread blocks**.
- This is achieved by **mapping** element of data vector to threads using **global index** **= threadIdx.x + blockIdx.x*blockDim.x**

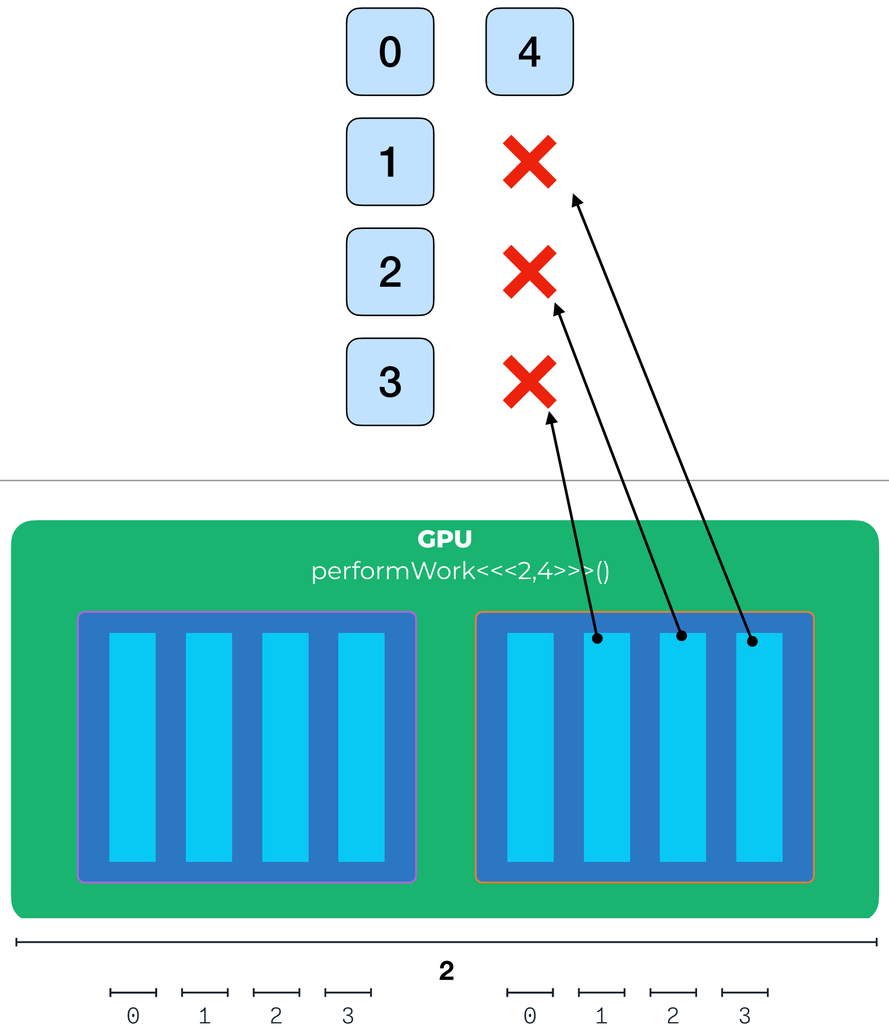



```
for blockIdx.x = 0
  i = 0 * 8 + threadIdx.x  = { 0, 1, 2, ... , 7 }
```

```
for blockIdx.x = 3
  i = 0 * 8 + threadIdx.x  = { 0, 1, 2, ... , 7 }
```

Handling Vector of Arbitrary Size

GPU
DATA



Code must check that the **`dataIndex`** calculated by **`threadIdx.x + blockIdx.x * blockDim.x`** is less than **`N`**, the number of data elements.

GPU

Choosing the Optimal Grid Size

Choose the optimal block size

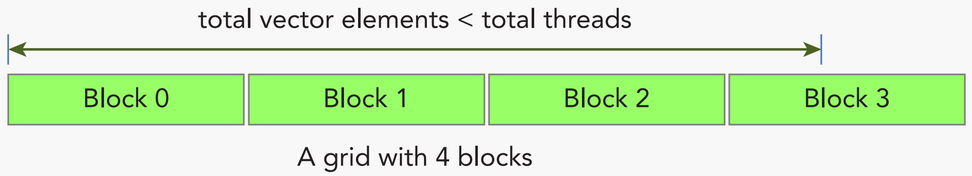
- Write an execution configuration that creates more threads than necessary
- Pass a value as an argument into the kernel (N) that represents that total size if the data set to be processed/total threads needed to complete the work
- Calculate the global index and if it does not exceed N perform the kernel work

```
// Coalesced access example
__global__ vectorSum(int N)
int idx = threadIdx.x + blockIdx.x * blockDim.x;
{
  if(idx < N){ // only do work if it does}
}
```

Know your limitations

Maximum size at each level of the thread hierarchy is device dependent. On A100 typical you get :

- Maximum number of threads per block : 1024
- Maximum sizes of x-, y-, and -z dimensions of threads block 1024 x 1024 x 64
- Maximum sizes of each dimension of grid of thread blocks: 65535 x 65535 x 65535 (about 280,000 billion blocks)



A grid with 4 blocks

Choosing the Optimal Grid Size

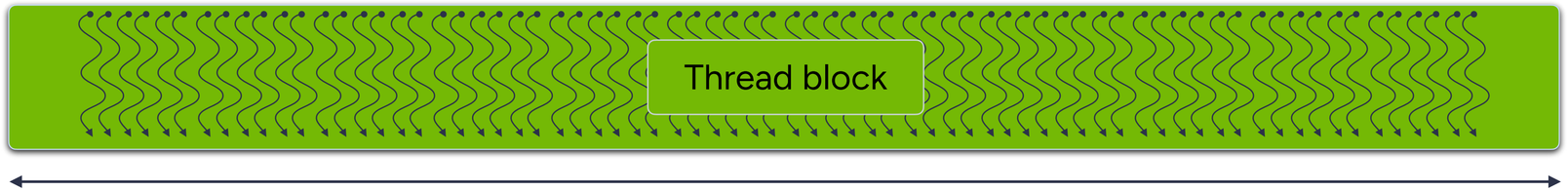
Choose the optimal block size

- best performance for blocks that contain a number of threads that is a **multiple of 32**, due to GPU hardware traits
- Example: we need to run 1000 parallel task with blocks containing 256 threads. How do we choose the optimal block size?

```
int N = 100000; size_t threads_per_blocks = 256;

size_t number_of_blocks = (N + threads_per_block - 1) / threads_per_block;

kernel<<<number_of_blocks, threads_per_block>>>(N);
```

- This calculation ensures that the number of blocks is sufficient to cover all the threads, even if the total number of threads is not evenly divisible by the threads per block
- The “-1” term is added to round up the division if necessary

Every Thread Runs Exactly the Same Program



A **limited number of threads (1024)** can fit inside a thread block

To increase parallelism, we need to **coordinate** work **among thread blocks**

This is achieved by **mapping** element of data vector to threads using **global index**

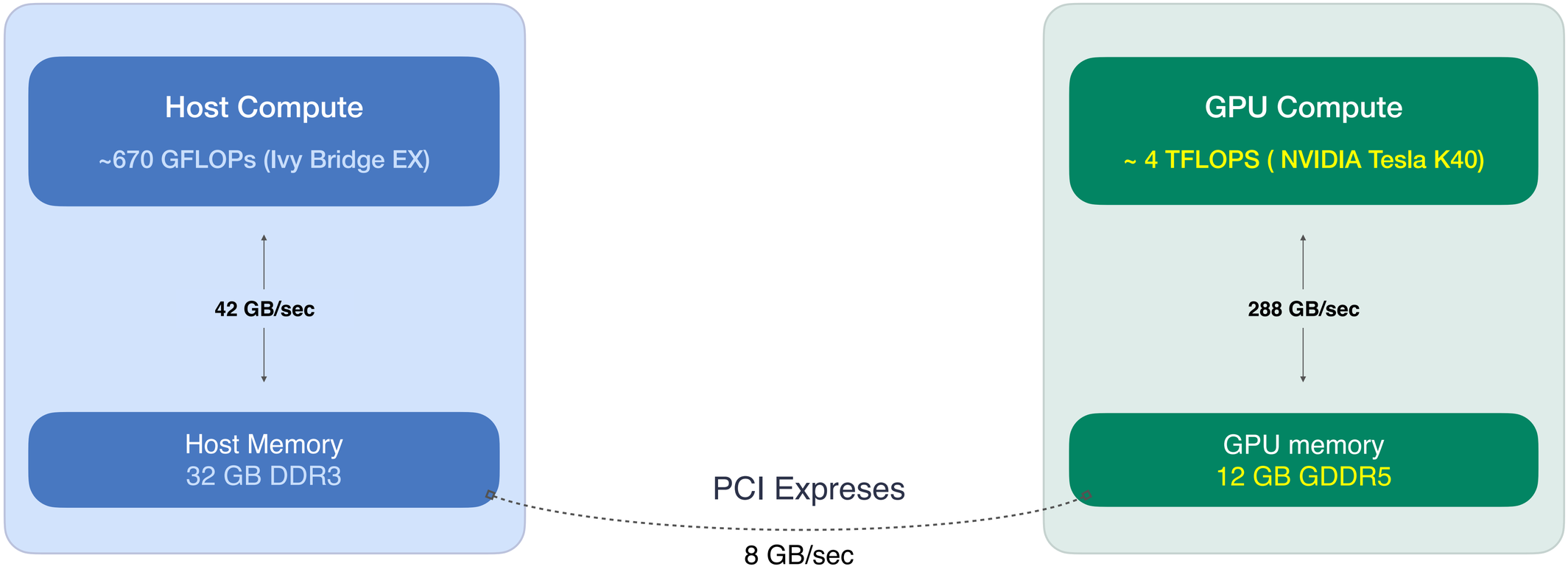
```
int index = threadIdx.x + (blockIdx.x * blockDim.x)
```

All about this one line code

Memory Hierarchy and Management

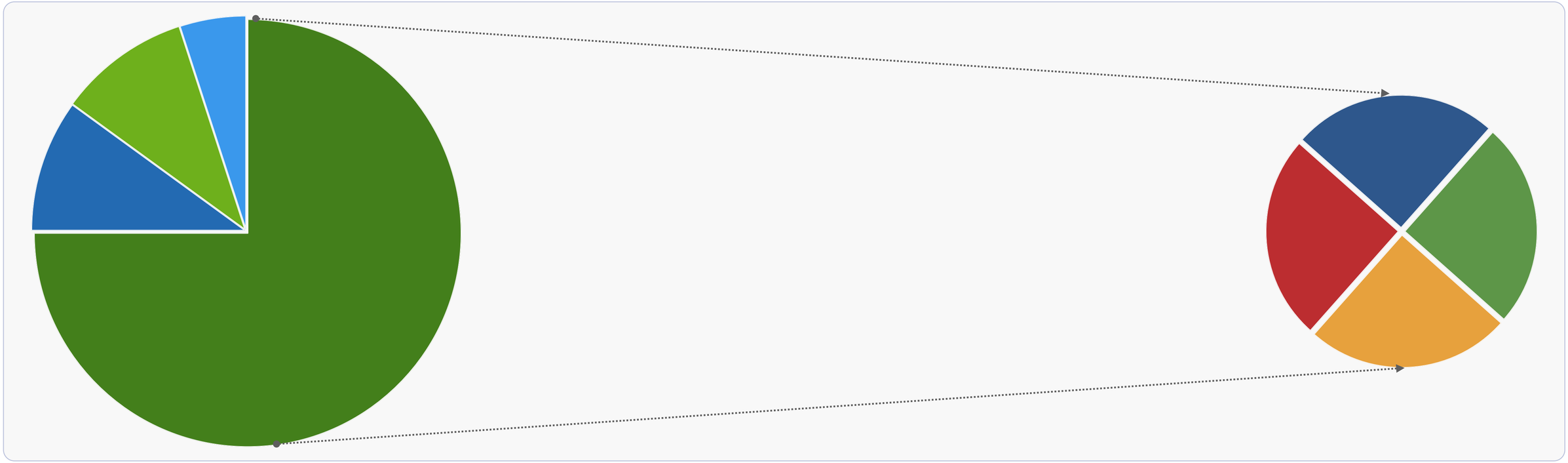
GPU vs. CPU: Understanding Performance Trade-Offs

Impact of data transfer on overall application performance



Data Transfer Impacts on Performance

Impact of data transfer on overall application performance



Minimize data transferred

Host-device data transfers when possible even if GPU kernels offer little or no speed-up over the CPU

Higher bandwidth

Can be achieved by using page-locked (or “pinned”) memory
Avoids paging overhead and allows for direct DMA transfers

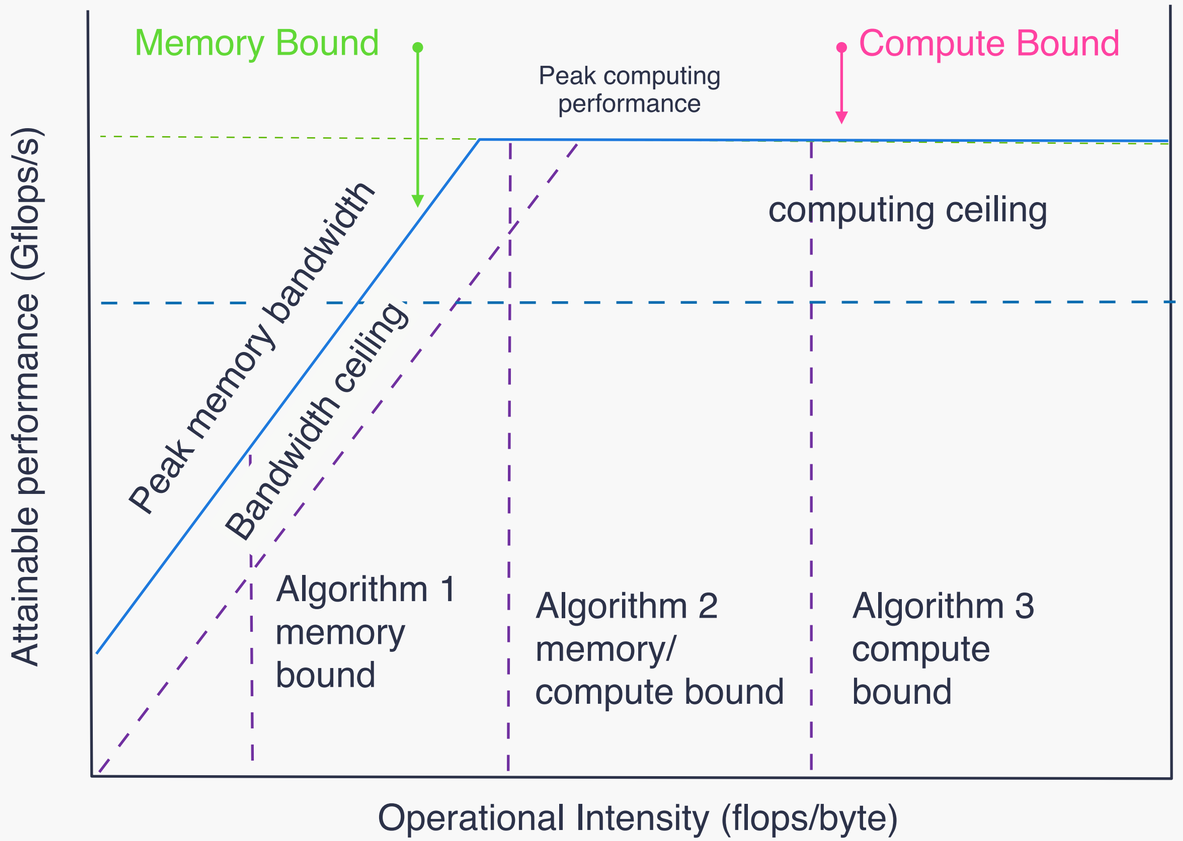
Efficient Data Transfer Tips

Batch small transfers to reduce overhead and improve performance
Overlap data transfers with kernel execution to maximize GPU utilization

Application Performance Constraints

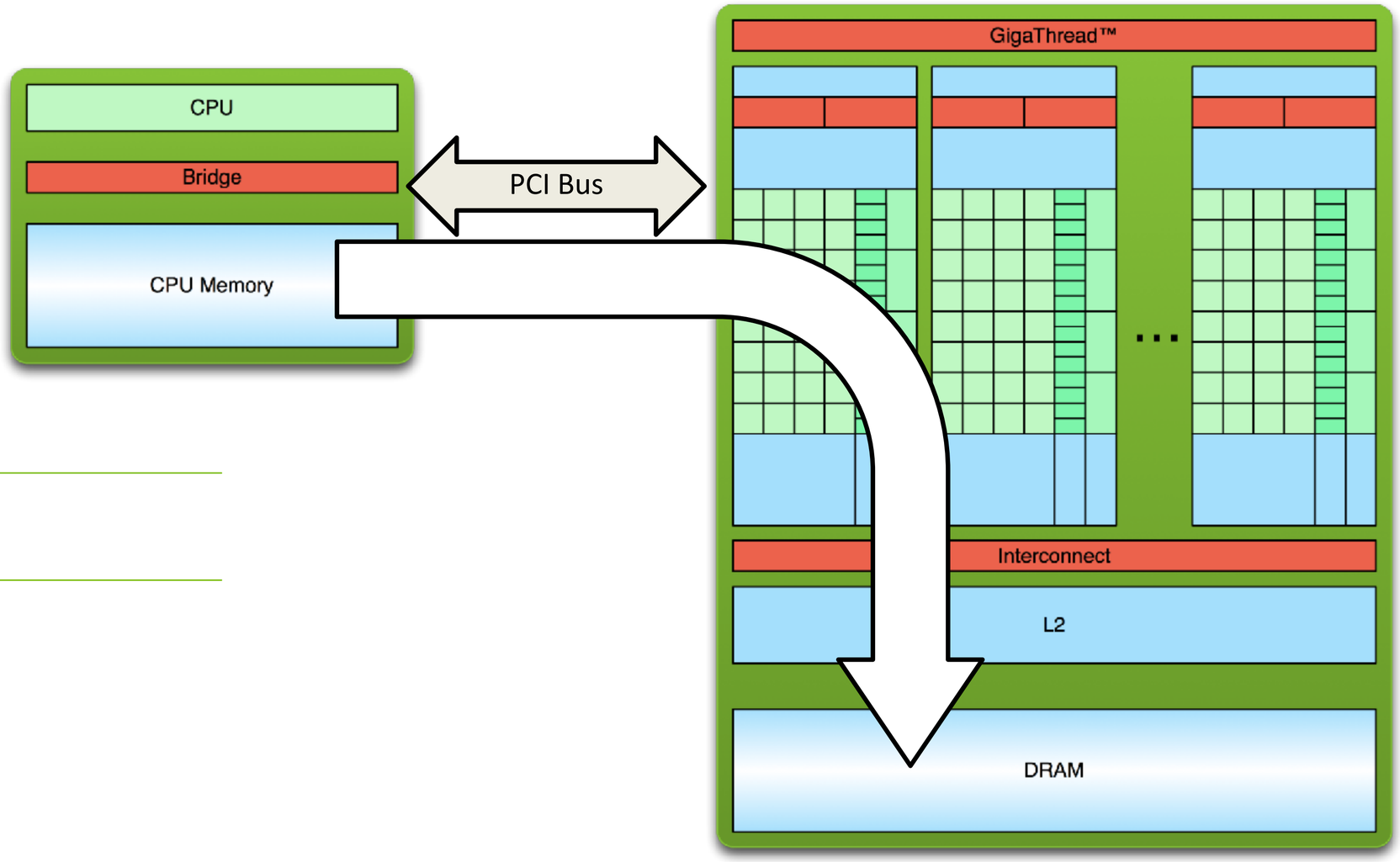
Roofline Model

- Key Concept: Computational Intensity:
  - Defined as FLOP (floating-point operations) per byte of memory transferred
- Latency Hiding:
  - Utilizing multiple warps on a Streaming Multiprocessor (SM) enables concurrent computation.
  - While some warps wait for memory transfers, others can continue executing
- Combined Performance:
  - The model illustrates how computation and memory transfer can overlap, represented as:
    - Performance = max(compute, memory transfer)



Three Simple Processing Steps

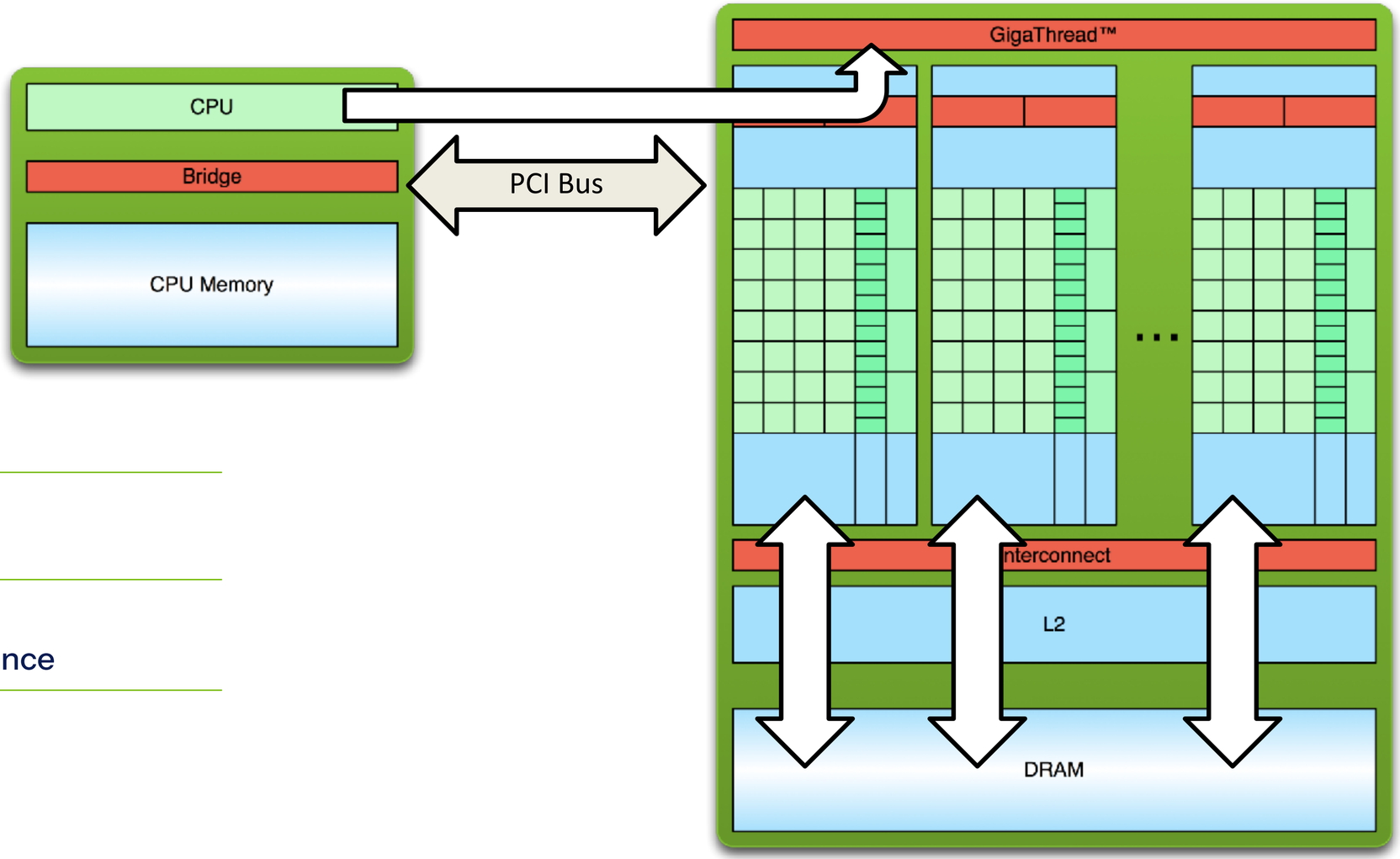
CPUs and GPUs have physically distinct memory regions connected by the PCIe bus



1. Allocate GPU memory
2. Copy data from CPU to GPU

Three Simple Processing Steps

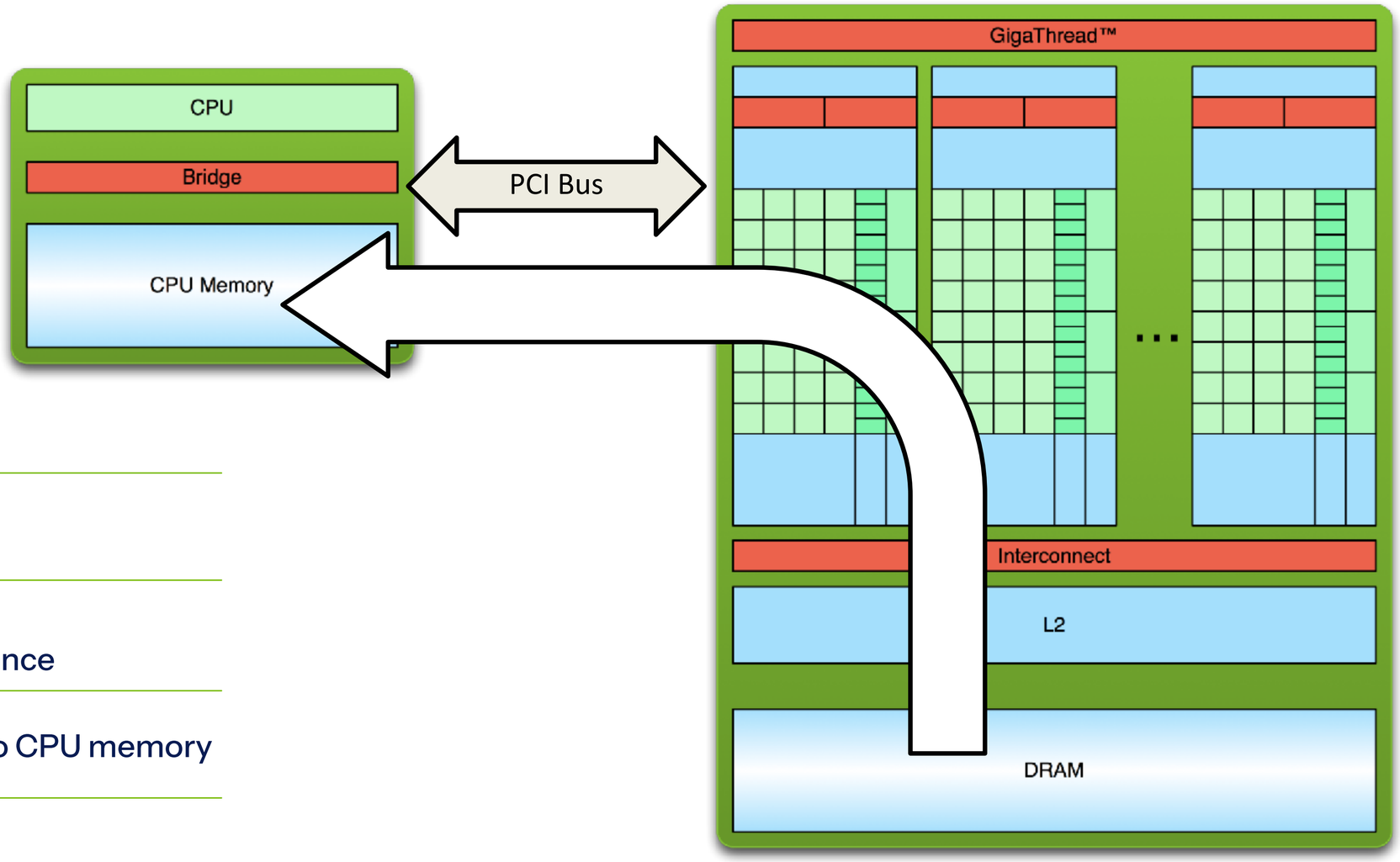
CPUs and GPUs have physically distinct memory regions connected by the PCIe bus



1. Allocate GPU memory
2. Copy data from CPU to GPU
3. Load GPU program and execute, caching data on chip for performance

Three Simple Processing Steps

CPUs and GPUs have physically distinct memory regions connected by the PCIe bus



1. Allocate GPU memory
2. Copy data from CPU to GPU
3. Load GPU program and execute, caching data on chip for performance
4. Copy results from GPU memory to CPU memory
5. Free GPU memory

Unified Virtual Memory (UVM)

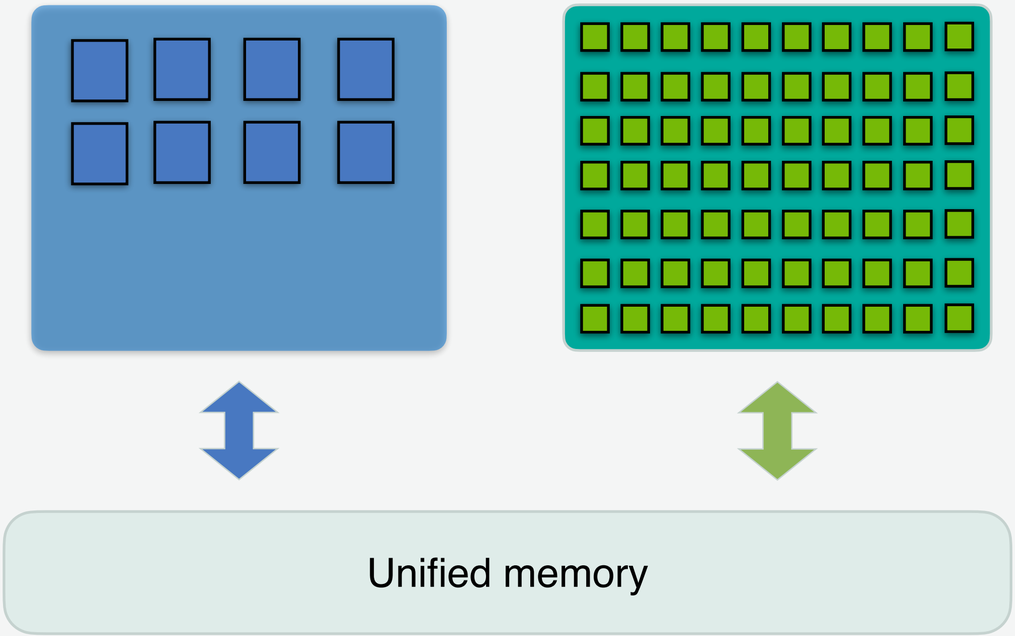
Increased memory latency

- Single allocation, single pointer, accessible everywhere eliminate the need of explicit copy and simplify code porting
- Enables the sharing of memory which reduces overall usage

Limited control over memory placement

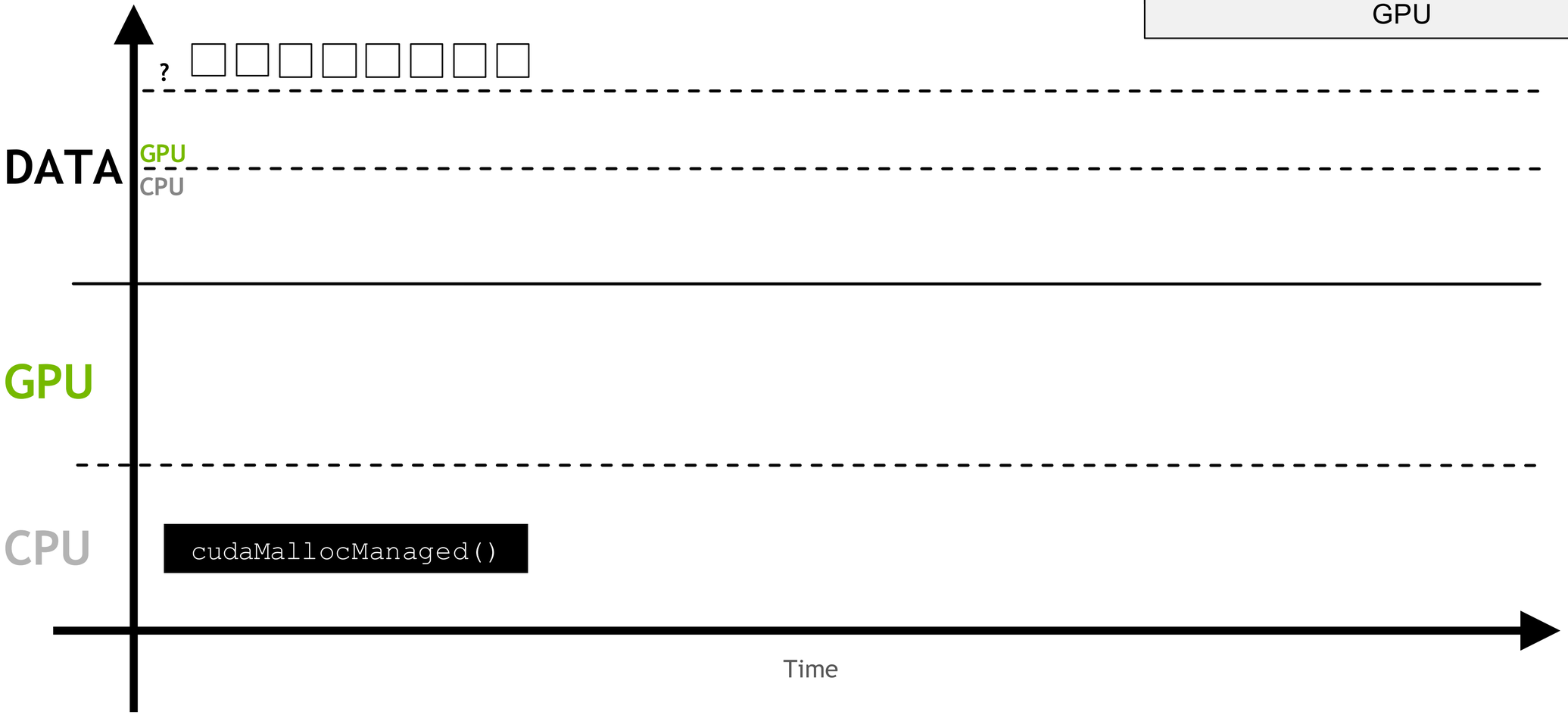
UVM automatically manages memory placement, which may not always be optimal for a given application

Developer view of GPU memory



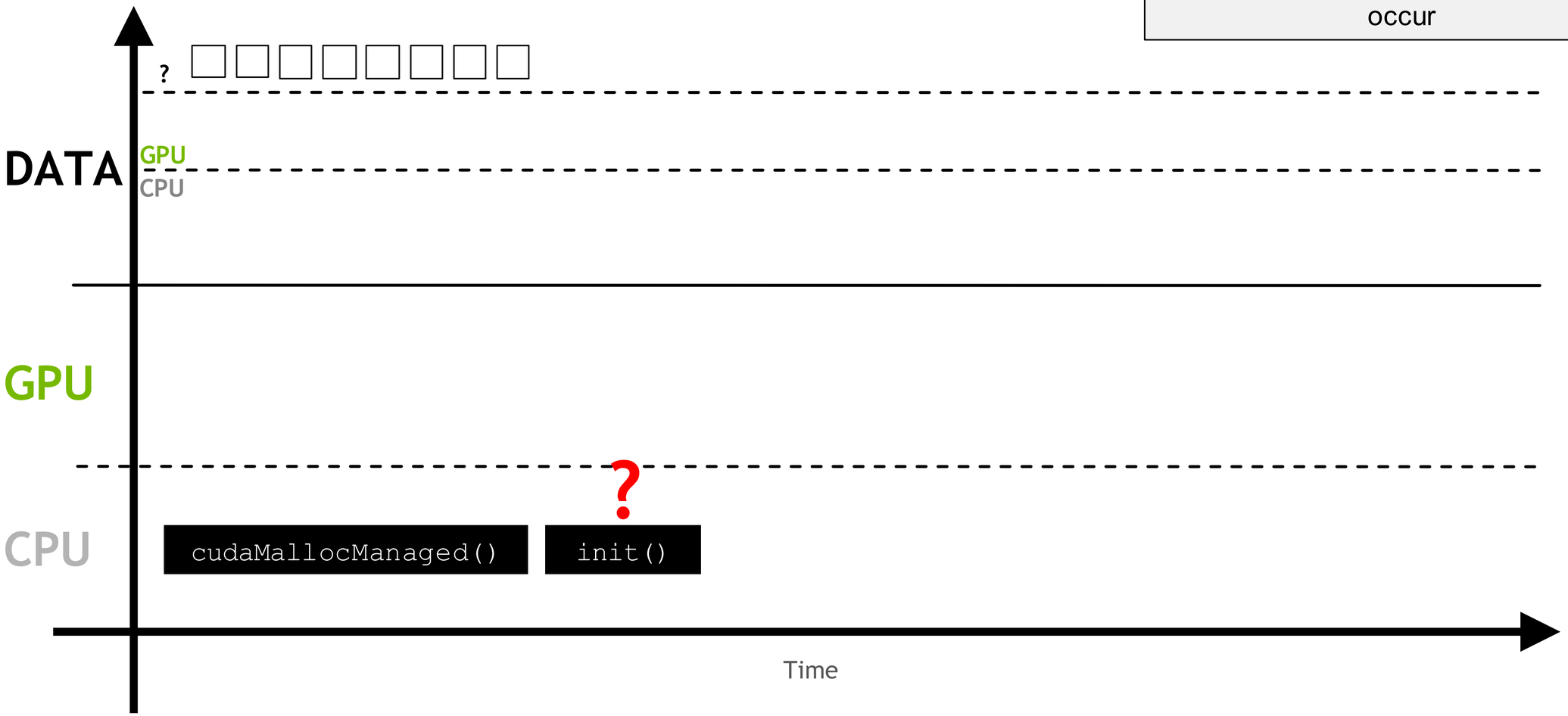
How Does cudaMallocManaged Actually Works?

When **UM** is allocated, it may not be resident initially on the CPU or the GPU



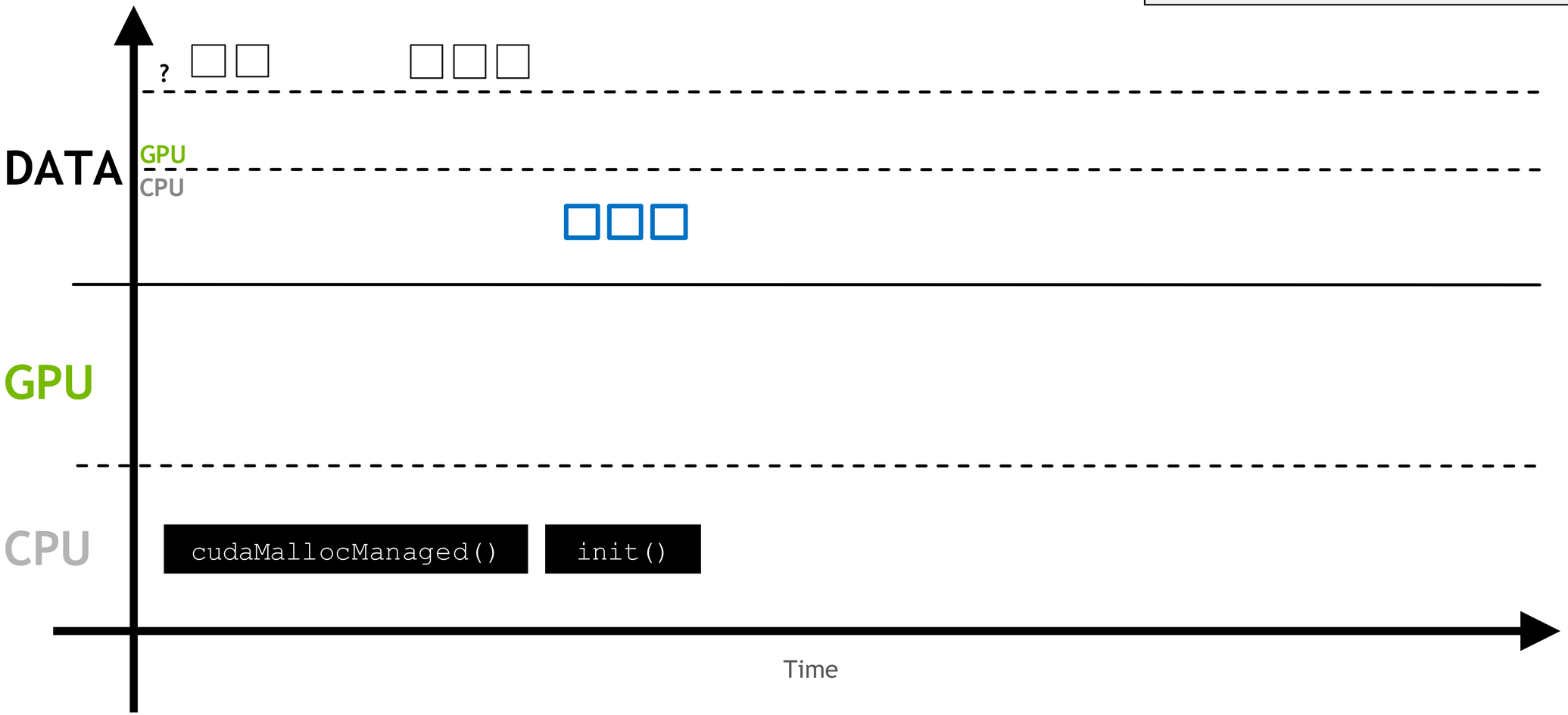
How Does cudaMallocManaged Actually Works?

When some work asks for the memory for the first time, a **page fault** will occur

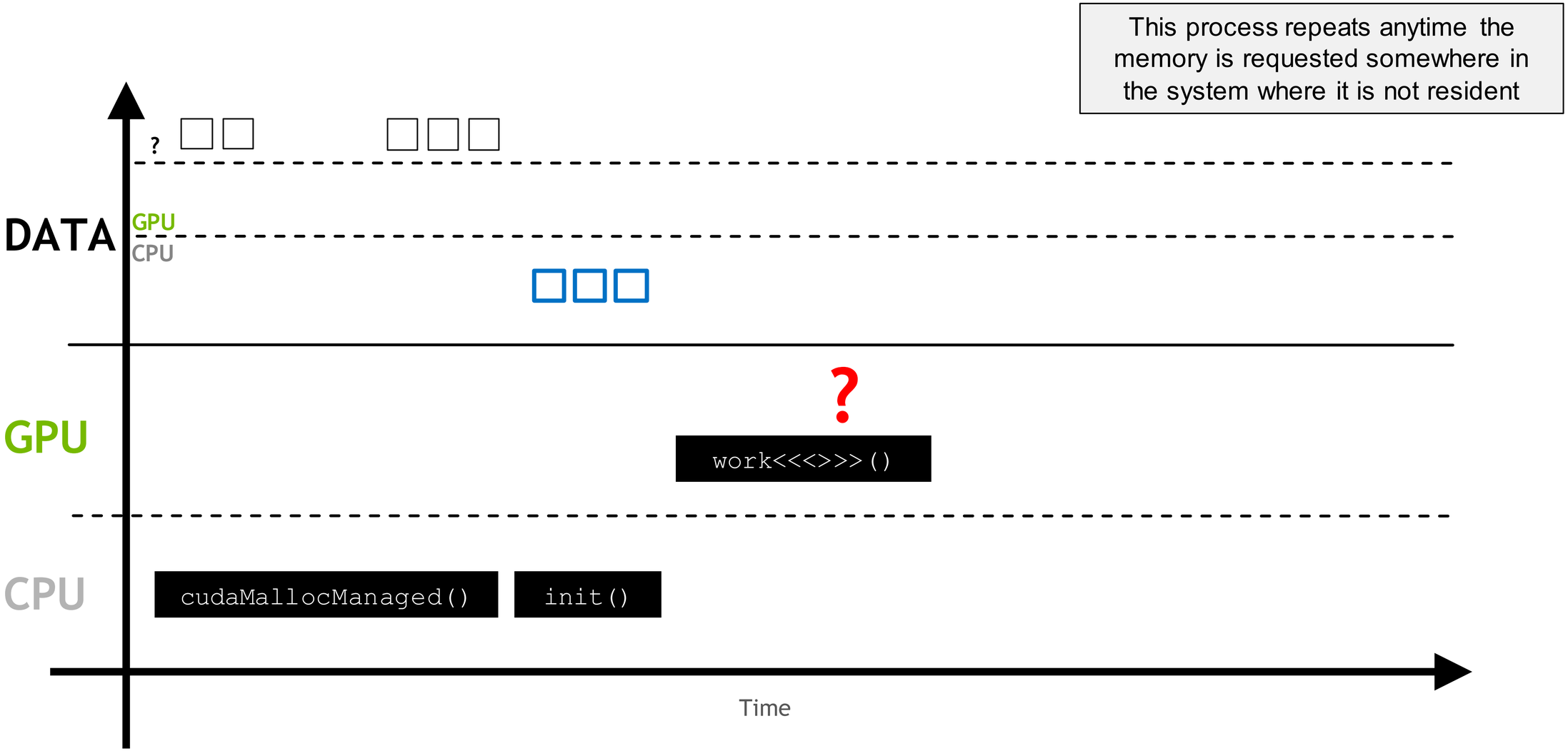


How Does cudaMallocManaged Actually Works?

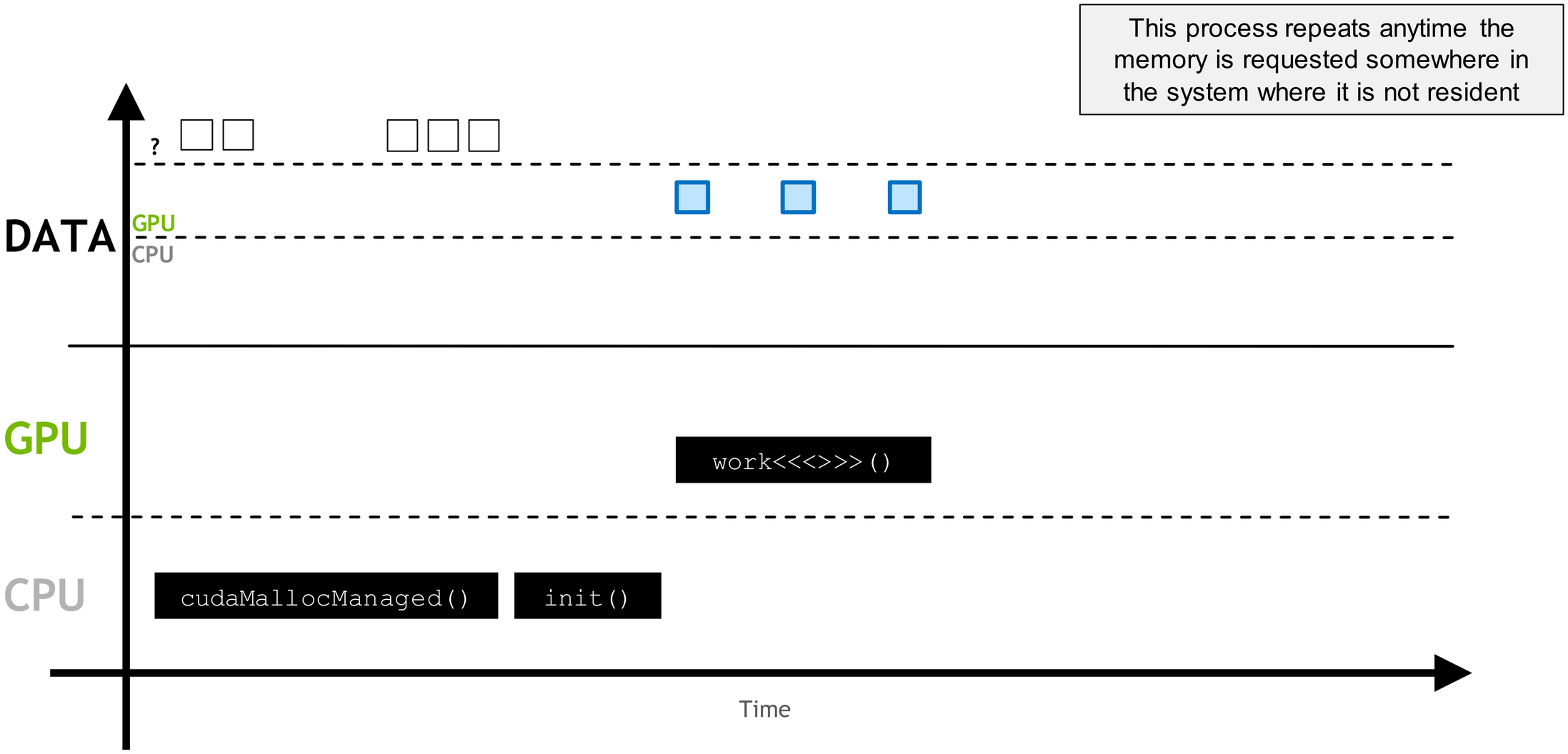
The page fault will trigger the migration of the demanded memory



How Does cudaMallocManaged Actually Works?



How Does cudaMallocManaged Actually Works?



Simplified Memory Management Code

Allow to **allocate** and **free memory**

CPU code

```
int N = 10000;
size_t size = N*sizeof(int);


int *a;
a = (int*)malloc(size);

free(a);
```

CUDA Code with UM

```
int N = 10000;
size_t size = N*sizeof(int);


int *a;
cudaMallocManaged(&a, size);

cudaFree(a);
```

Exercise-2 : Vector Sum

Things to do

The starting code contains a CPU vector addition function. Using what you have learned so far, accelerate the addVectorsInto function to run as a CUDA kernel on the GPU and to do its work in parallel. Consider the following guidelines.

T1. Augment the vecSum definition so that it is a **CUDA kernel**.

T2. Choose a working execution configuration for vecSum.

T3. Use cudaMallocmanaged **to manage the data transfer**

Mapping to Hardware

1

CUDA invokes kernel grid

Host kicks off the execution of a kernel grid which contains blocks of threads

2

Execute concurrently

Each SM runs multiple thread blocks

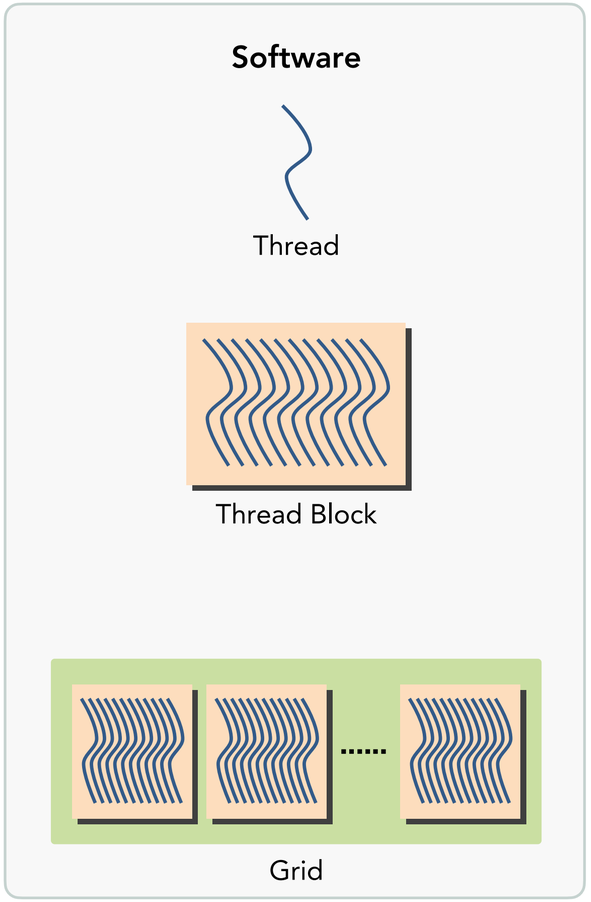
Each SP runs on thread from a thread blocks

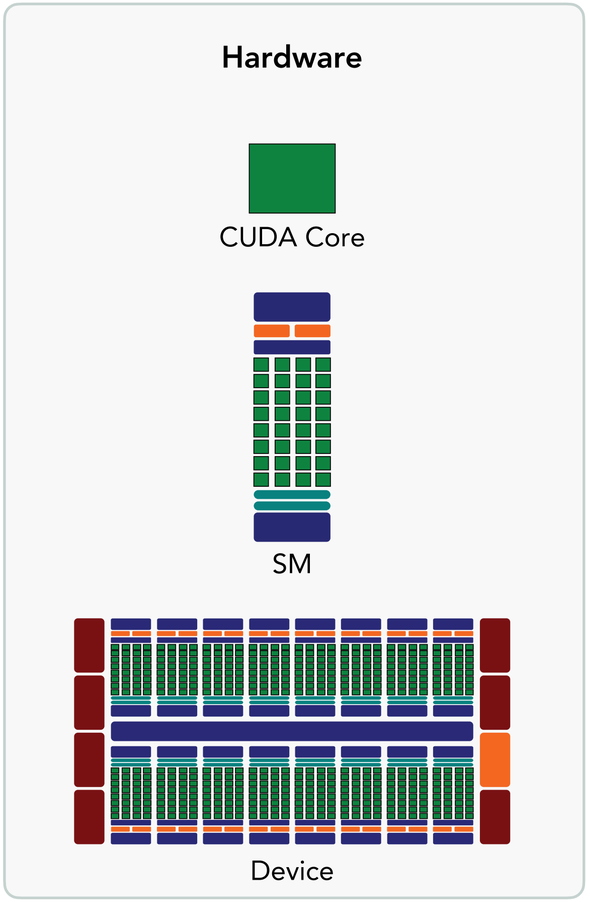
3

Grid blocks distributed to SMs

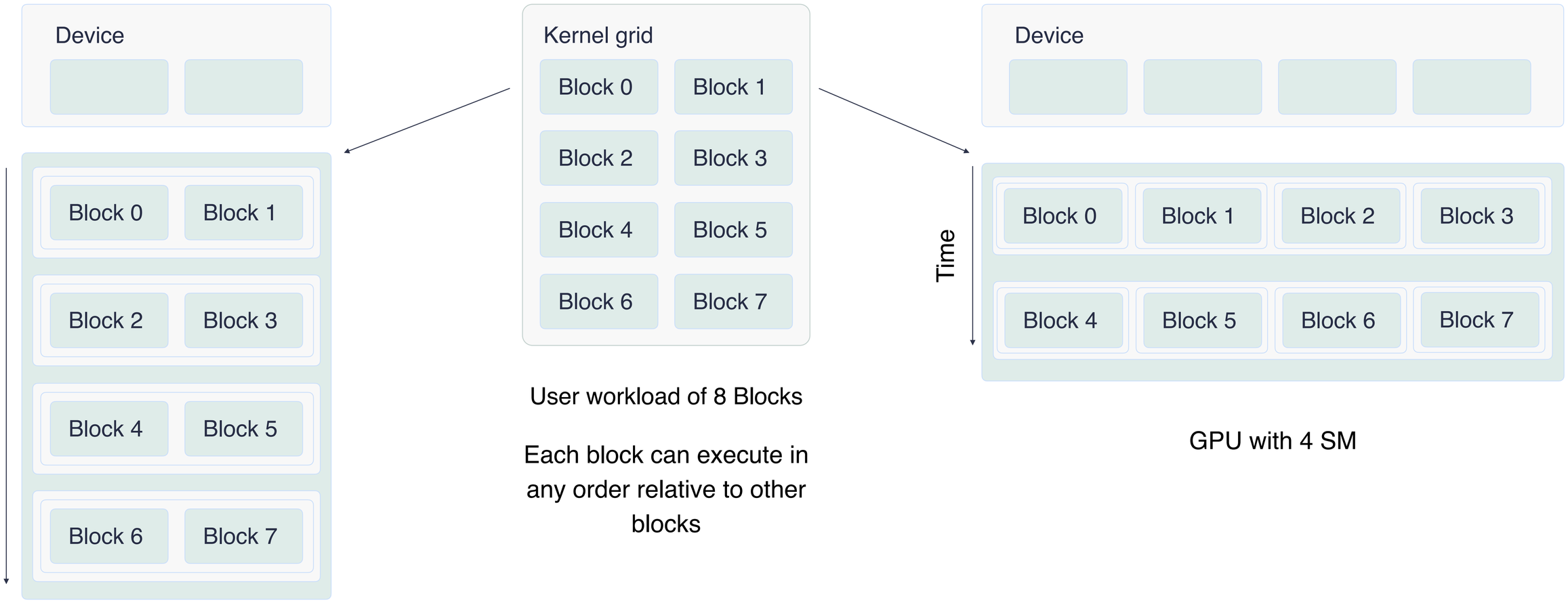
Shared cache, register and memory

Global memory shared by all SMs

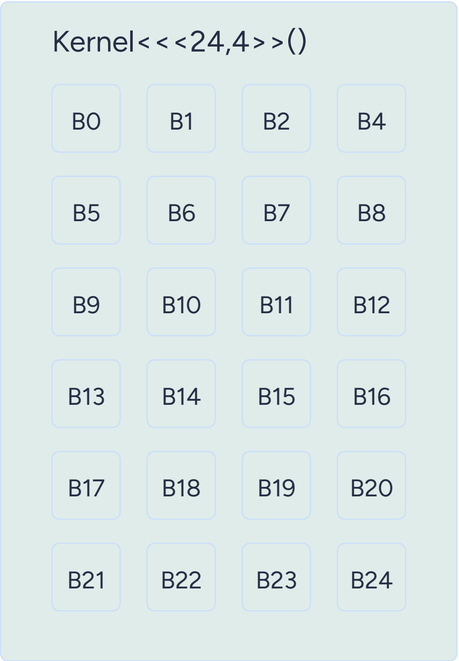


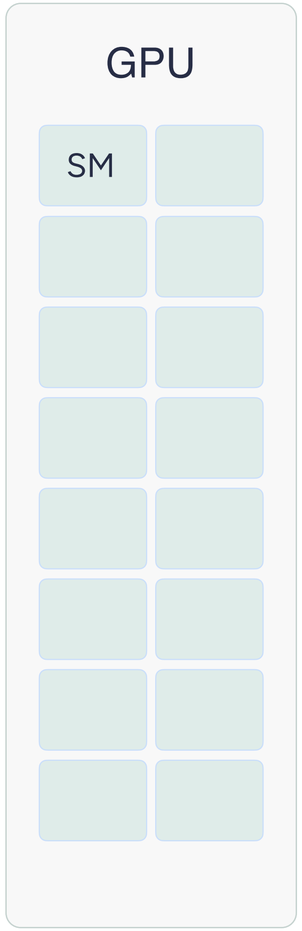


Transparent Scalability



NVIDIA GPUs Contain Functional Units Called SMs

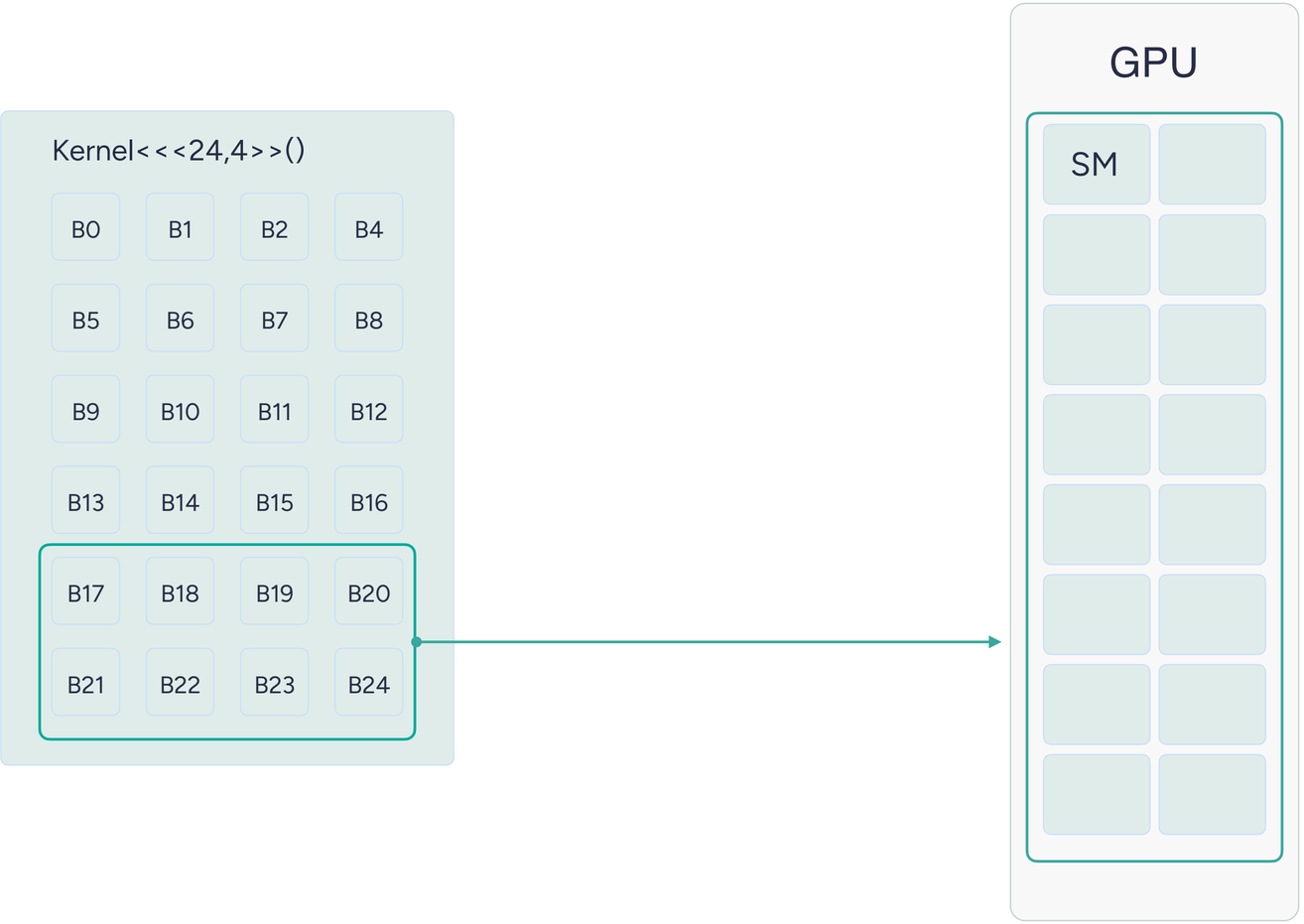




Blocks of threads are scheduled to run on SMs

GPU with 16 SMs

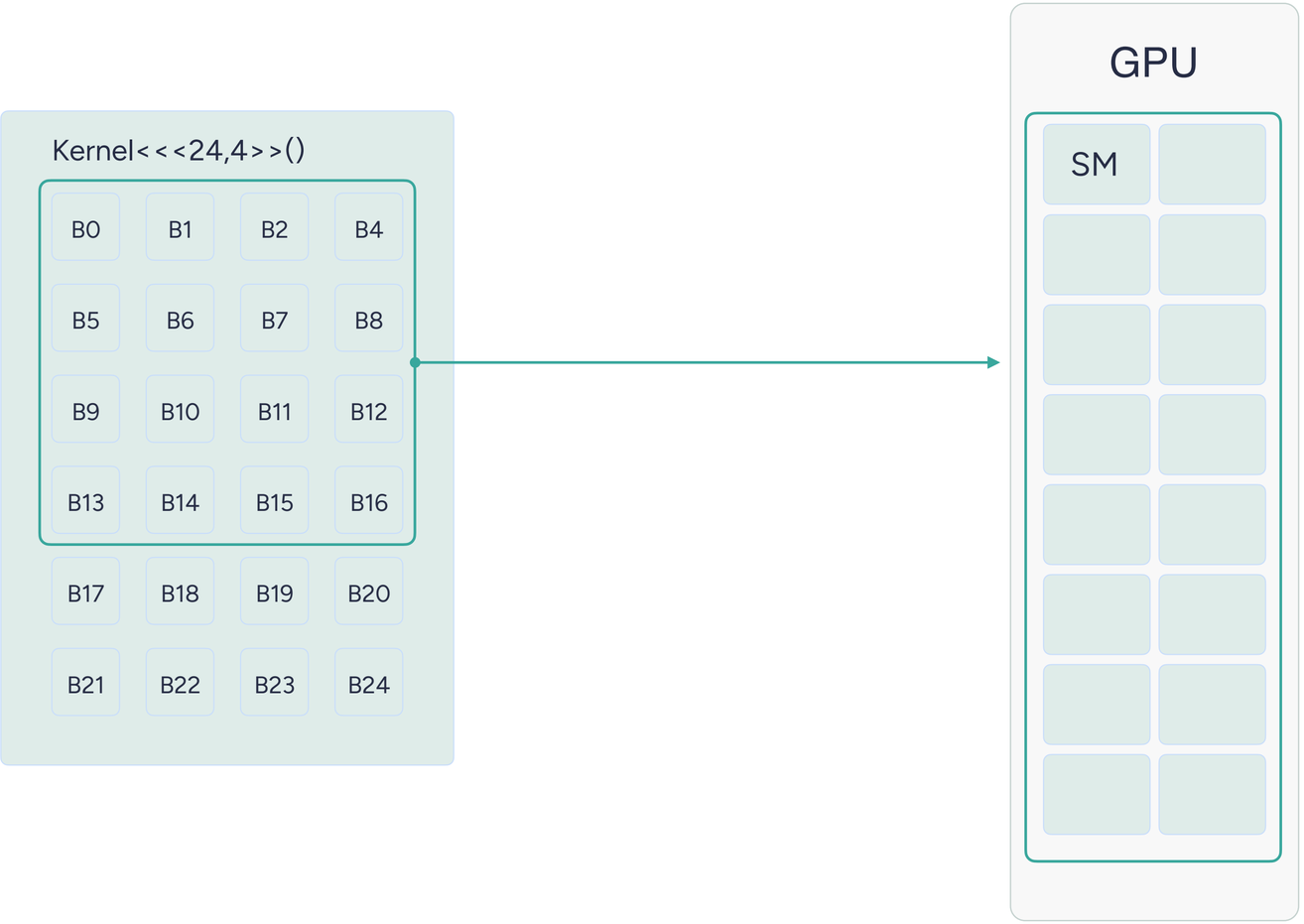
More Than One Block Can Be Schedule on an SM



Depending on the number of SMs on a GPU, and the requirements of a block, more than one block can be scheduled on SM

GPU with 16 SMs

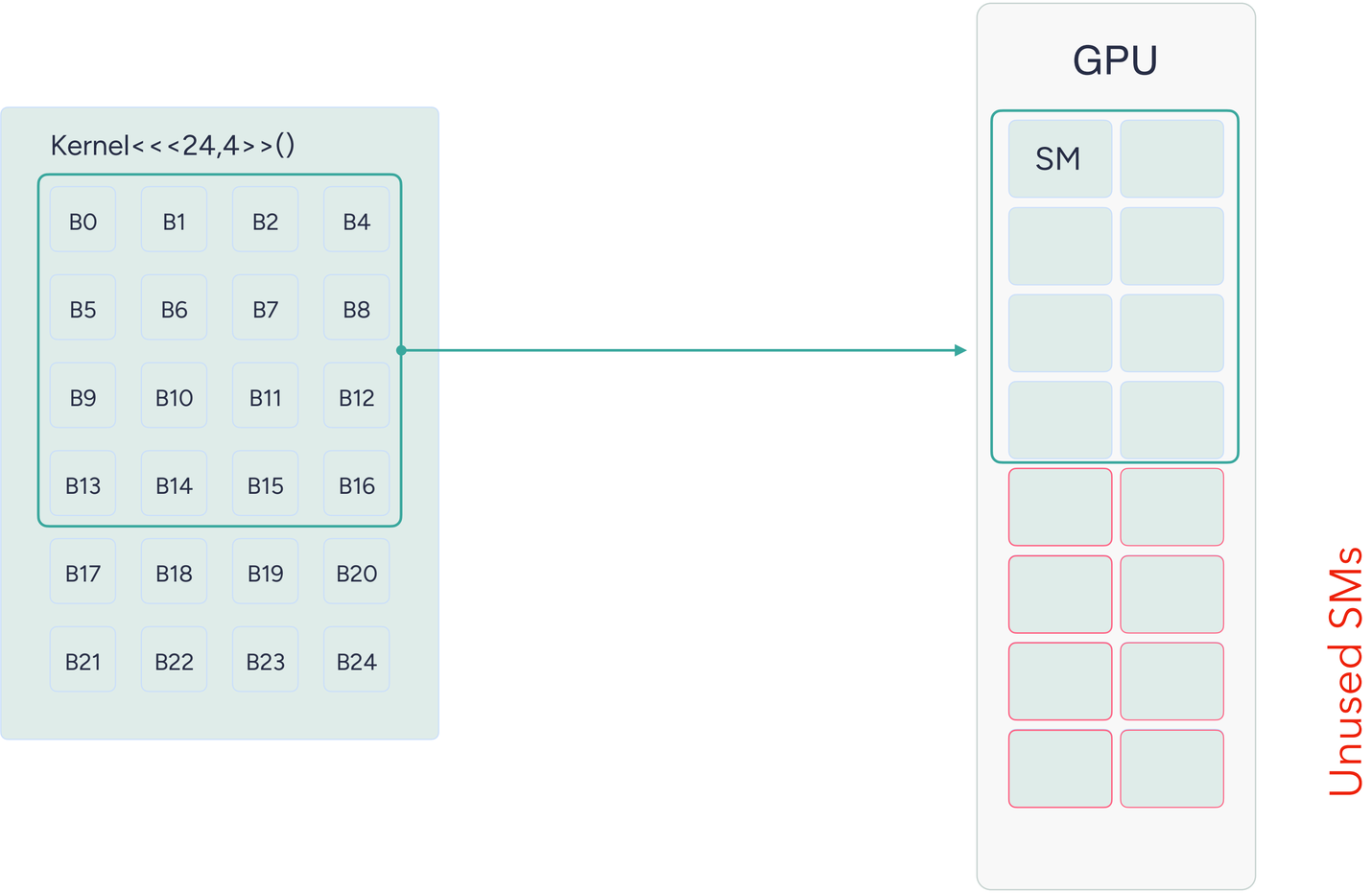
More Than One Block Can Be Schedule on an SM



Depending on the number of SMs on a GPU, and the requirements of a block, more than one block can be scheduled on SM

GPU with 16 SMs

More Than One Block Can Be Schedule on an SM



Grid dimensions divisible by the number of SMs on a GPU can prove full SM utilisation

GPU with 16 SMs

Query Device Properties and Handle Errors in CUDA C/C++

Why Query Device Properties?

Hardware Diversity

CUDA-capable GPUs range widely in specifications. Different models offer varying memory, cores, and capabilities.

Optimization Opportunities

Kernel launches and configurations can be tuned for specific hardware. This maximizes performance on available resources.

Error Prevention

Checking properties prevents runtime failures. Your code can gracefully handle incompatible hardware situations.

Programmatically Querying GPU Device Properties

- **The number of SMs on a GPU** can differ depending on the specific GPU being used, so the number of SMs should **not be hard-coded** into a code bases
- This information should be acquired **programatically**.
- To obtain the id of the currently active GPU:

```
int deviceId;
cudaGetDevice(&deviceId);
```

- To obtain a C struct which contains many properties about the currently active GPU device, including its number of SMs:

```
cudaDeviceProp props;
cudaGetDeviceProperties(&props, deviceId);
```

Querying Device Properties

This code demonstrates the practical enumeration pattern. It loops through all devices and displays key properties for each.

```
#include <stdio.h>
int main() {
int nDevices;
cudaGetDeviceCount(&nDevices);
for (int i = 0; i < nDevices; i++){
cudaDeviceProp prop;
cudaGetDeviceProperties(&prop, i);
printf("Device Number: %d\n", i);
printf("  Device name: %s\n", prop.name);
printf("  Memory Clock Rate (KHz): %d\n",            prop.memoryClockRate);
printf("  Memory Bus Width (bits): %d\n",            prop.memoryBusWidth);
printf("  Peak Memory Bandwidth (GB/s): %f\n\n",     2.0*prop.memoryClockRate*(prop.memoryBusWidth/8)1.0e6);
}
```

Validate GPU Results by Comparing With CPU Results

```
// Validate results
bool validateResults(float *hostRef, float *gpuRef, int nElem) {
   bool correct = true;
   for (int i = 0; i < nElem; i++) {
   if (fabs(hostRef[i] - gpuRef[i]) > 1e-5) {
   correct = false;
   printf("Mismatch at index %d: CPU = %f, GPU = %f\n", i, hostRef[i], gpuRef[I]);
   break;
}
}
   if (correct) {
   printf("Results match!\n”);
   }
   return correct;
}
```

Handling CUDA Errors

- Error handling in accelerated CUDA code is essential.
- **All CUDA API returns an error code of type `cudaError_t`**
  - Special value cudaSuccess means that no error occurred
- **An error message can be printed with `cudaGetErrorString`**

```
cudaError_t err;
err = cudaMallocManaged(&a, N);
if(err != cudaSuccess) { printf("Error: %s \n", cudaGetErrorString(err)); }
```

- **To check for errors occurring at the time of kernel launch, CUDA provides the cudaGetLastError function, which does return a value of type `cudaError_t`**

```
someKernel <<<1, -1 >>>();        // - 1 is not a valid number of threads
cudaError_t err;
err = cudaGetLastError();
if(err != cudaSuccess) { printf("Error: %s \n", cudaGetErrorString(err));}
```

CUDA Error Handling Function

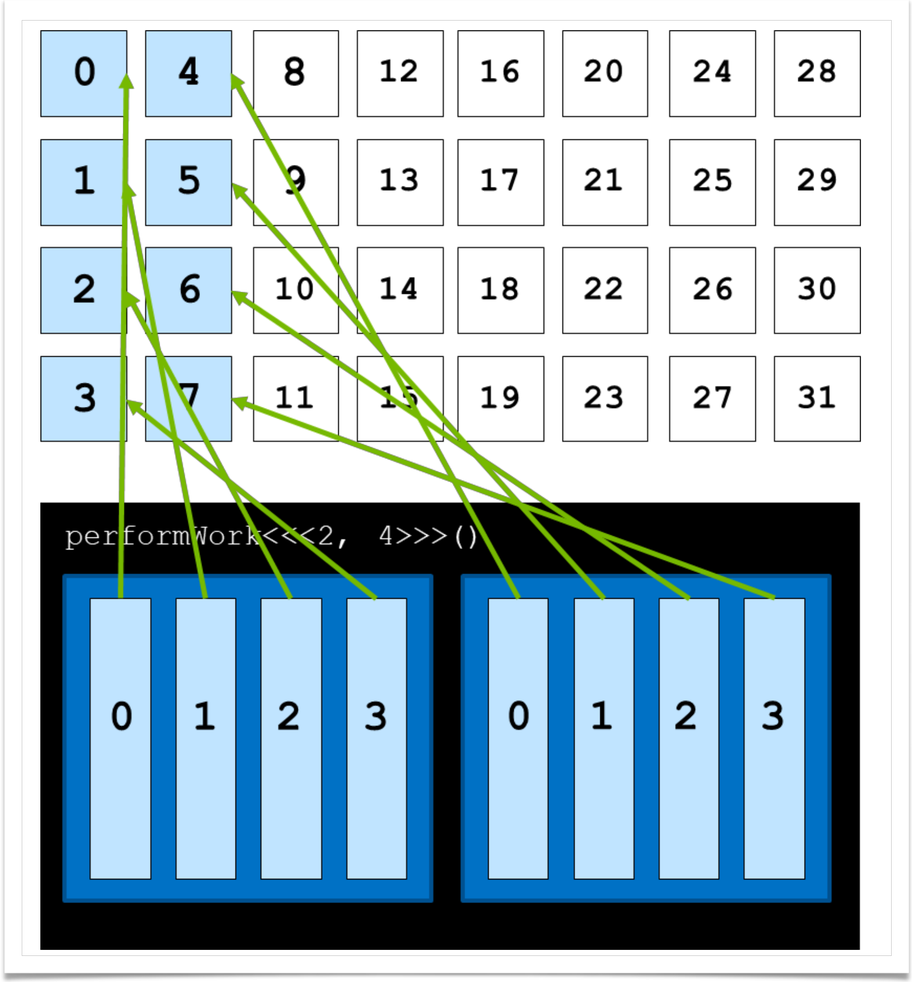
- A macro that wraps CUDA function calls for checking errors could be useful
- Can be wrapped around any function that returns a `cudaError_t`

```
#include <stdio.h>
#include <assert.h>
inline cudaError_t checkCuda(cudaError_t result) {
  if (result != cudaSuccess) {
      fprintf(stderr, "CUDA Runtime Error: %s\n", cudaGetErrorString(result));
      assert(result == cudaSuccess); }
  return result; }

int main() {
/*  The macro can be wrapped around any function returning
* a value of type `cudaError_t`.
*/
checkCuda( cudaDeviceSynchronize() )
}
```

Are there hardware constraints on threads per block and blocks per grid?

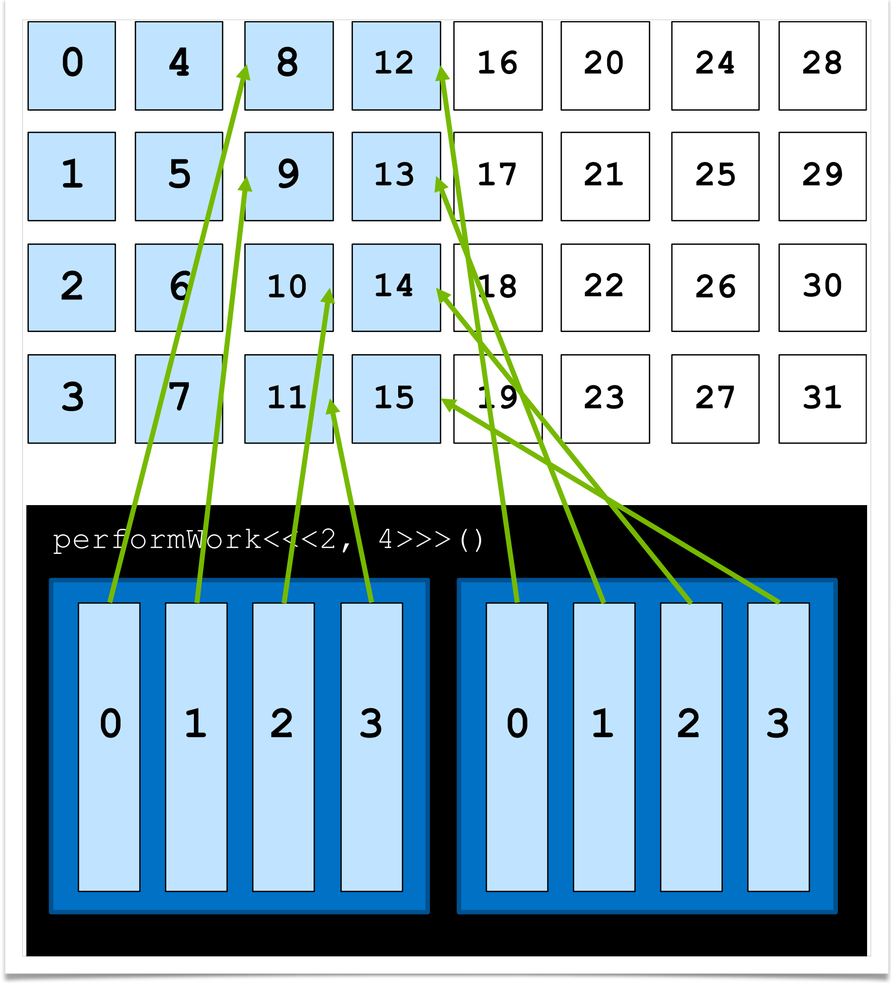
When the Data Set Is Larger Than Grid Size?

- In this scenario, each thread should work on more elements.
- Work can be assigned programmatically with a grid- stride loop.



When the Data Set Is Larger Than Grid Size?

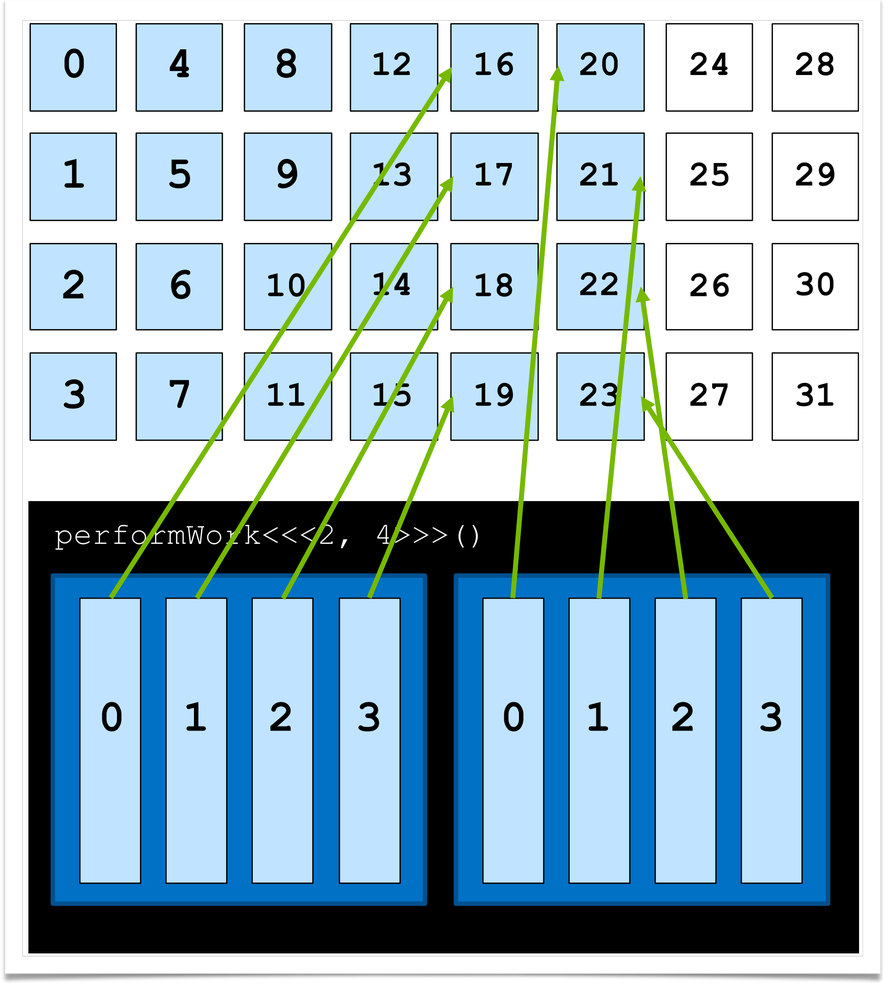
- In this scenario, each thread should work on more elements.
- Work can be assigned programmatically with a grid- stride loop
  - the first element to be assigned to a thread is calculated via the global index



When the Data Set Is Larger Than Grid Size?

- In this scenario, each thread should work on more elements.
- Work can be assigned programmatically with a grid- stride loop
  - the first element to be assigned to a thread is calculated via the global index

```
int globalIndex = threadIdx.x + blockIdx.x *blockDim.x;
```



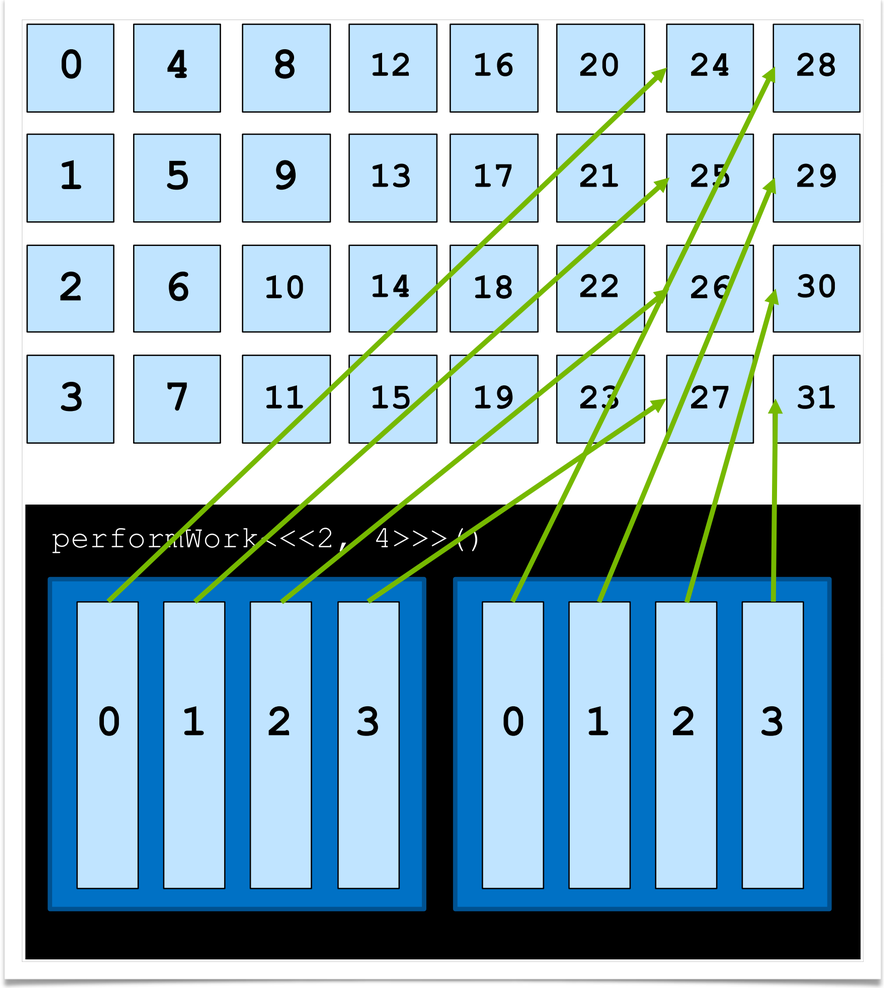
When the Data Set Is Larger Than Grid Size?

- In this scenario, each thread should work on more elements.
- Work can be assigned programmatically with a grid- stride loop
  - the first element to be assigned to a thread is calculated via the global index

```
int globalIndex = threadIdx.x + blockIdx.x *blockDim.x;
```

- the next one is obtained by summing the number of threads in the grid

```
stride = blockDim.x * gridDim.x
```



Data Set Larger Than Grid Size: Grid-Stride Loop

Operation inside the kernel will be executed in a grid-stride loop:

```
__global__ void kernel(int *a, int N)
{
  int indexWithinTheGrid = threadIdx.x + blockIdx.x *blockDim.x;
  int gridStride = blockDim.x * gridDim.x;
  for(int i= indexWithinTheGrid; I <N; I += gridStride)
{
  // do work on a[I];
}
```

Exercise-3 : Vector Sum: grid stride

Things to do

The starting code contains a CPU vector addition function. Using what you have learned so far, accelerate the addVectorsInto function to run as a CUDA kernel on the GPU and to do its work in parallel. Consider the following guidelines.

T1. Augment the vecSum definition so that it is a **CUDA kernel**.

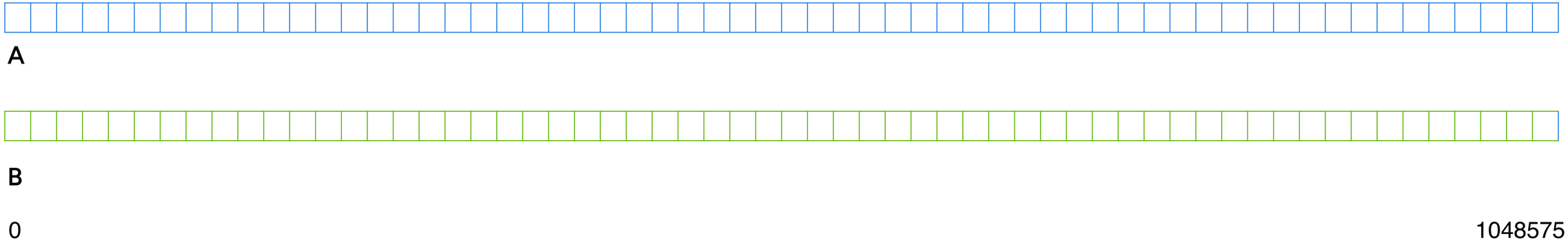
T2. Choose a working execution configuration for vecSum, and use stride

T3. Use cudaMallocmanaged **to manage the data transfer**

T4. Compare it for different problem size N = 1<< 16, 1<<20, 1<< 22, 1<<24

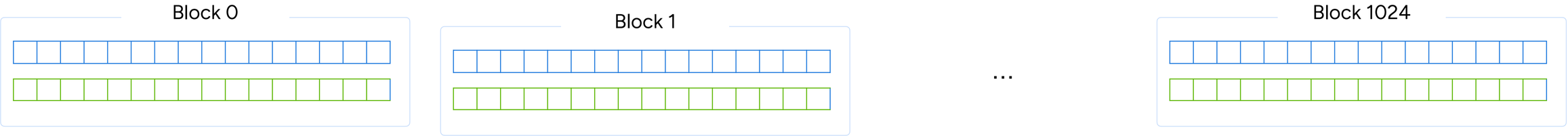
A Subtle but Important Point: Threads vs. Data

If you recall, our vector add had two arrays: A and B. These arrays were 1 million elements each in length



We Then Broke Them Up Into 1024 Separate Blocks of 1024 Threads Each

If you recall, our vector add had two arrays: A and B. These arrays were 1 million elements each in length



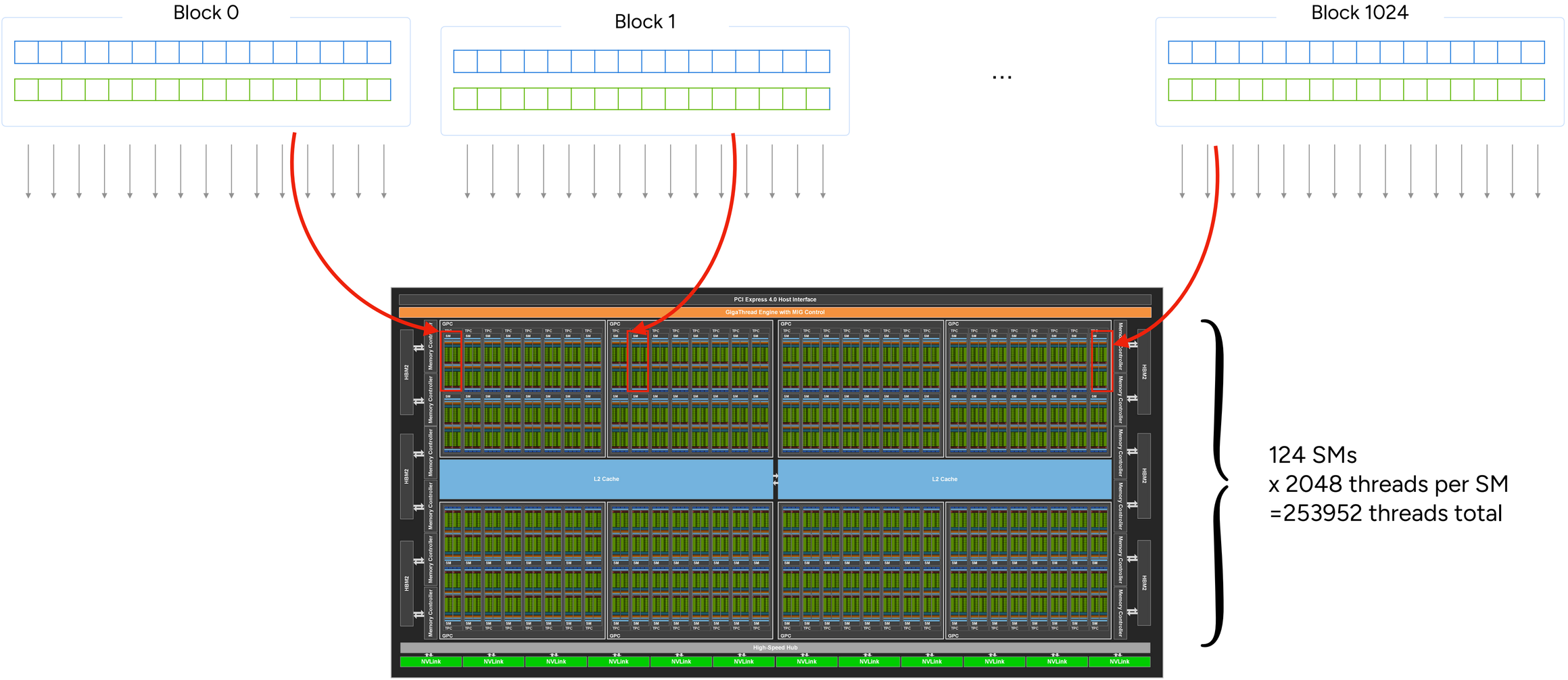

```
_ _global_ _
void vecAdd(float* a, float* b, float* c, int n) {
   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   if (idx < n) {
      c[idx] = a[idx] + b[idx];
   }
}

void run_cuda() {
   ...
   vecAdd<<<1024,1024>>>(d_a, d_b, d_c, n);
  ...
}
```

1024 blocks

1024 thread per block

How the GPU Runs Parallel Programs

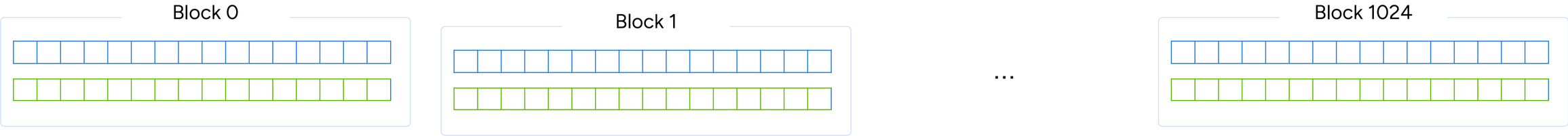
We break the work into “blocks” with a subset of the threads in each block



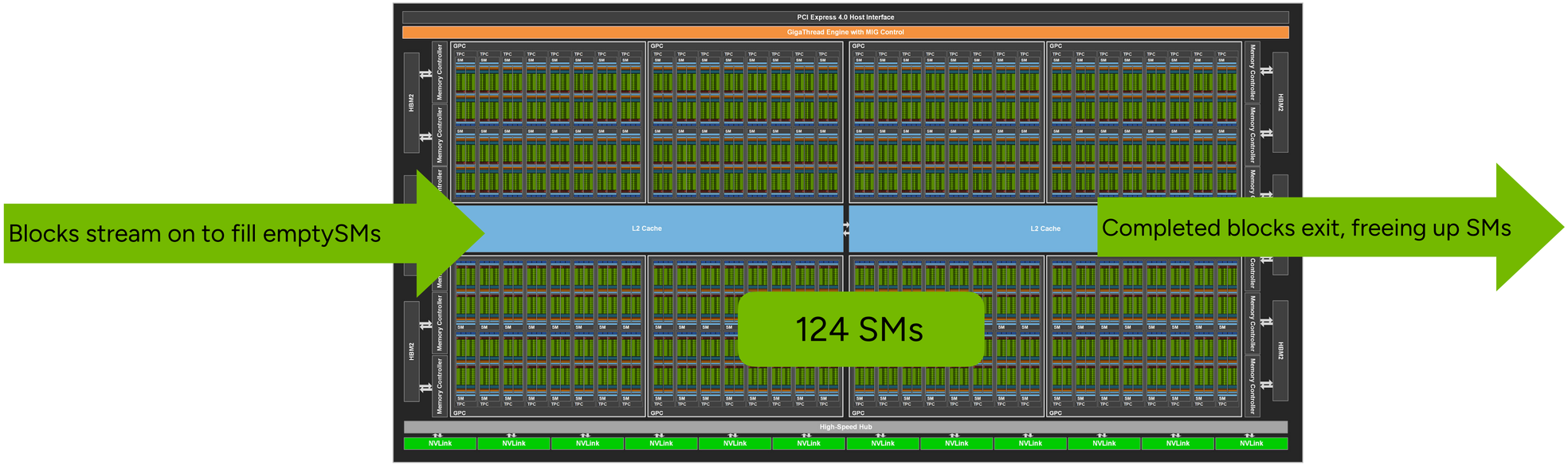
124 SMs
x 2048 threads per SM
=253952 threads total

The GPU Works by Oversubscription

Increasing SM Count = More Concurrency = Higher Performance

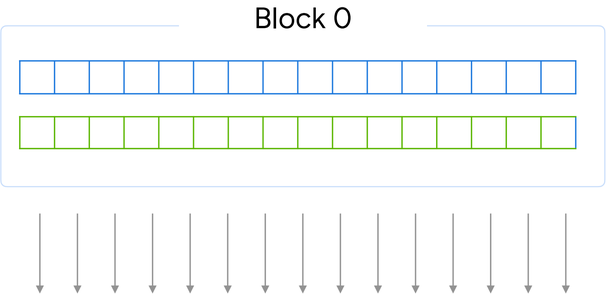


A deep queue of blocks ensures that the GPU is never idle



GA100 Full GPU (2022)

Our Simple Example Processes One Element of Data per Thread



Requires 1048576 threads, or 1024 blocks of 1024 threads

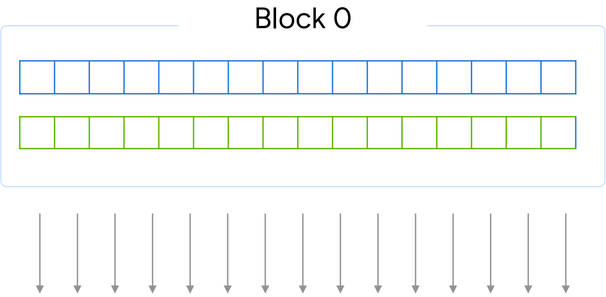
Data index idx is equal to thread index, so number of threads == number of elements

```
_ _global_ _
void vecAdd(float* a, float* b, float* c, int n) {
   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   if (idx < n) {
      c[idx] = a[idx] + b[idx];
   }
}

void run_cuda() {
   ...
   vecAdd<<<1024,1024>>>(d_a, d_b, d_c, n);
  ...
}
```

1024 blocks

1024 thread per block

Our Simple Example Processes One Element of Data per Thread



Requires 1048576 threads, or 1024 blocks of 1024 threads

Data index idx is is now (thread index * 4) We now process 4 data elements per thread

Must now check for out of bounds because 1,048,576 does not divide evenly by 253952

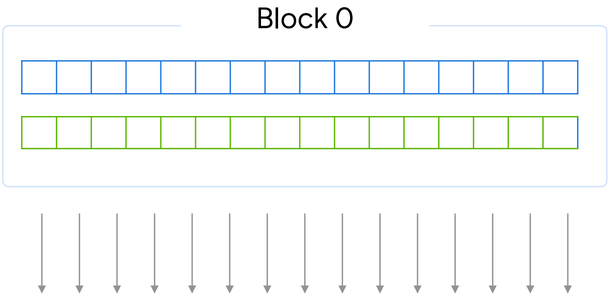
Reduce number of blocks from 1024 → 256 because we no longer need 1 million threads

```
_ _global_ _
void vecAdd(float* a, float* b, float* c, int n) {
   int idx = (blockIdx.x * blockDim.x + threadIdx.x)*4;
   if (idx < n) {
      c[idx] = a[idx] + b[idx];
   }
}

void run_cuda() {
   ...
   vecAdd<<<256 1024,1024>>>(d_a, d_b, d_c, n);
   ...
}
```

1024/4 blocks

1024 thread per block

A Subtle but Important Point: Threads vs. Data



When Using Few Threads (blockDim.x * gridDim.x < n)

- Only a small fraction of the data (e.g., ~0.1%) gets processed.
- Remaining elements are **left untouched**, leading to **incorrect results**.
- Each thread only handles **one element**, which limits coverage.

Benefits of the Grid-Stride Loop Pattern

- Ensures entire data range is covered, even with fewer threads.
- Avoids need for **oversized grids** — more scalable and flexible.
- Highly **efficient memory access** with minimal overhead.
- Helps smooth out **intra-block scheduling jitter**, improving performance consistency

```
--- Case 1: Small Vector (n = 1 << 16) —
=== Running with one element per thread ===
Execution time: 0.030867 ms
Validation: PASSED

=== Running with multiple element per thread ===
Execution time: 0.021679 ms
Validation: PASSED

--- Case 2: Large Vector, Enough Threads (n = 1 << 24) ---
=== Running with one element per thread ===
Execution time: 0.193886 ms
Validation: PASSED

=== Running with multiple element per thread ===
Execution time: 0.194935 ms
Validation: PASSED

--- Case 3: Large Vector, Limited Threads (n = 1 << 24) ---
=== Running with one element per thread ===
Mismatch at index 16384
Execution time: 0.055996 ms
Validation: FAILED

=== Running with multiple element per thread ===
Execution time: 0.722521 ms
Validation: PASSED
```

Which One Is Better?

Kernel1: No stride

Pros

- Straightforward: Each thread handles one element
- Efficient for small to medium vectors.
- Coalesced memory access
- which is good for memory bandwidth

Cons

- Low utilization if n is much larger than total threads launched.
- Requires enough threads to cover the entire vector for full performance.

Kernel2: With stride

Pros

- Scalable: Can process very large vectors with a limited number of threads.
- Good when `n` >> total threads launched — avoids underutilization
- Useful for smaller launches or limited resource availability

Cons

- Can lead to lower cache performance because each thread accesses memory at strided intervals (less spatial locality).
- Potentially less memory coalescing, depending on the stride size.

NVIDIA profile tools

Using Profiling Tools: NVIDIA Nsight and More

Nsight Compute

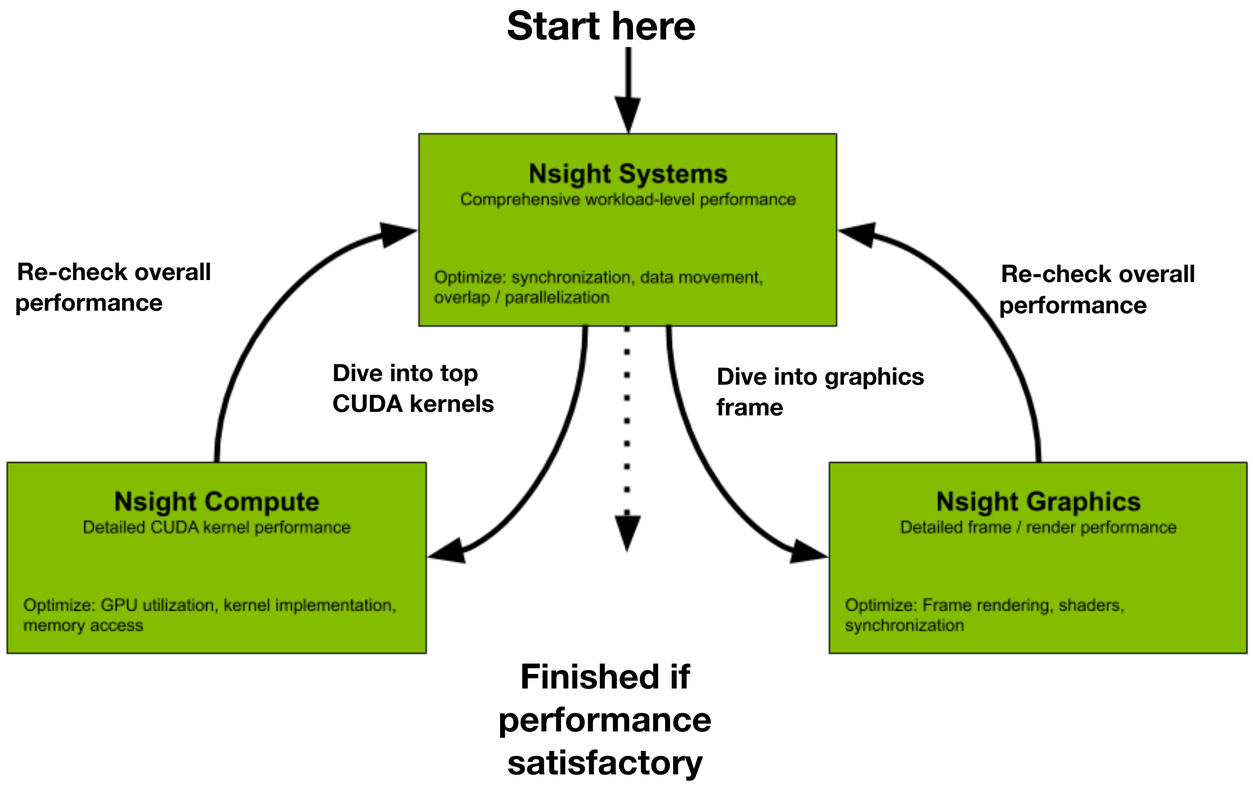
Deep kernel analysis. Shows occupancy, memory usage, and instruction throughput

Nsight Systems

System-wide performance view. Tracks CPU and GPU interactions across time

Profiling Benefits

Compare actual vs. theoretical throughput. Map performance issues to source code



SCAN ME

At First NSIGHT: Recording an Application Timeline

Notable flags for nays profile

```
nsys profile -t cuda,nvtx,mpi,openacc --stats=true --force-overwrite true -o my_report ./myapp
```

```
CUDA API Statistics:

 Time(%)  Total Time (ns)  Num Calls      Average        Minimum        Maximum        StdDev             Name
 -------  ---------------  ---------  --------------  -------------  -------------  -------------  ----------------------
    80.8    2,389,597,601          1  2,389,597,601.0  2,389,597,601  2,389,597,601            0.0  cudaDeviceSynchronize
    18.9      560,567,324          3    186,855,774.7         37,749    560,460,726  323,551,379.2  cudaMallocManaged
     0.3        8,408,411          3      2,802,803.7      2,302,933      3,332,155      515,243.9  cudaFree
     0.0           79,831          1         79,831.0         79,831         79,831            0.0  cudaLaunchKernel

CUDA Memory Operation Statistics (by time):

 Time(%)  Total Time (ns)  Operations  Average  Minimum  Maximum  StdDev              Operation
 -------  ---------------  ----------  -------  -------  -------  -------  --------------------------------
    75.4       10,016,498       1,920  5,216.9    2,846   17,792  4,223.6  [CUDA Unified Memory memcpy HtoD]
    24.6        3,269,861         640  5,109.2    2,815   16,862  4,121.6  [CUDA Unified Memory memcpy DtoH]
```

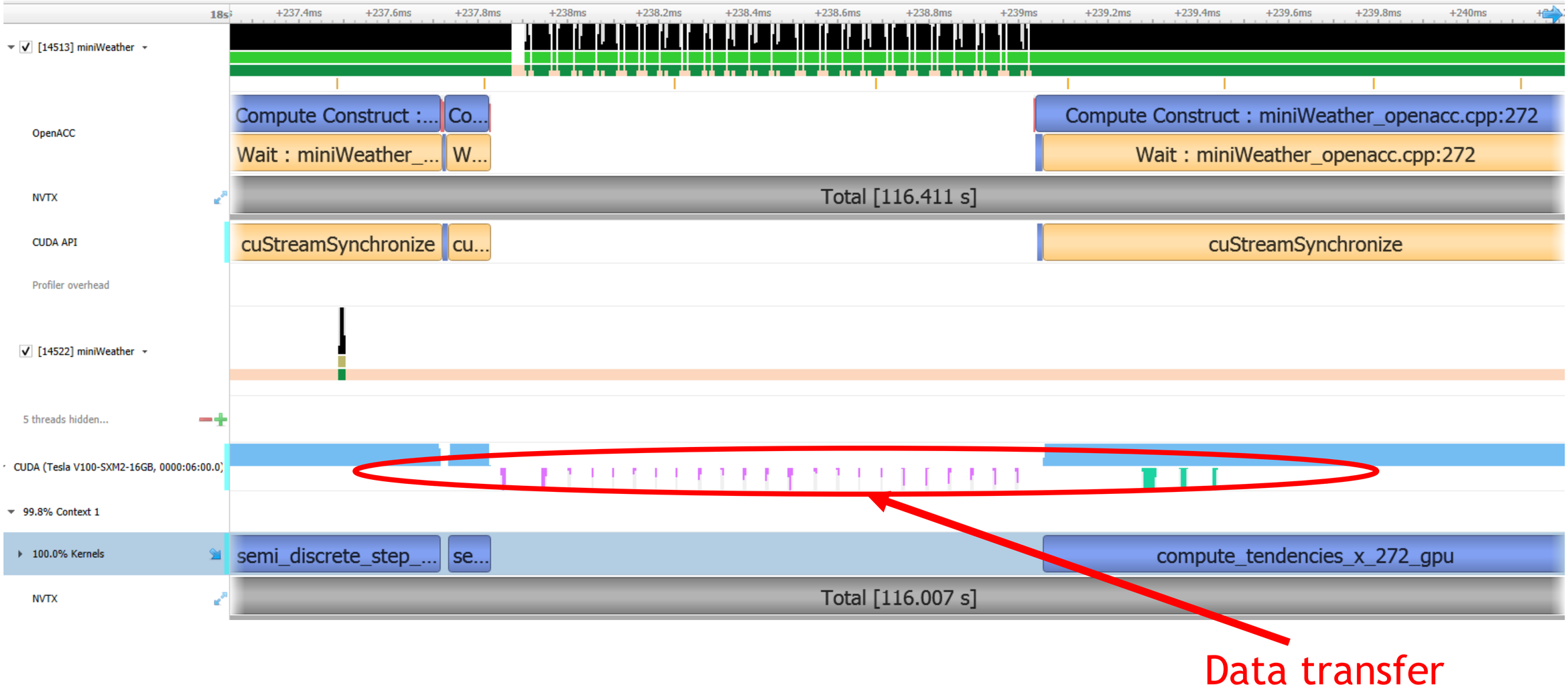
SCAN ME

nsys --help or nsys [specific command] --help

- profile – start a profiling session
- -t: Selects the APIs to be traced (cuda, cublas, nvtx, mpi openmp and openacc in this example)
- —cuda-memory-usage = true or false
- --stats: if true, it generates summary of statistics after the collection
- --force-overwrite: if true, it overwrites the existing generated report
- -o – name for the intermediate result file, created at the end of the collection (.qdrep filename)

See also https://docs.nvidia.com/nsight-systems/

At First NSIGHT: Recording an Application Timeline



Data transfer

NCU Report — Quick Start

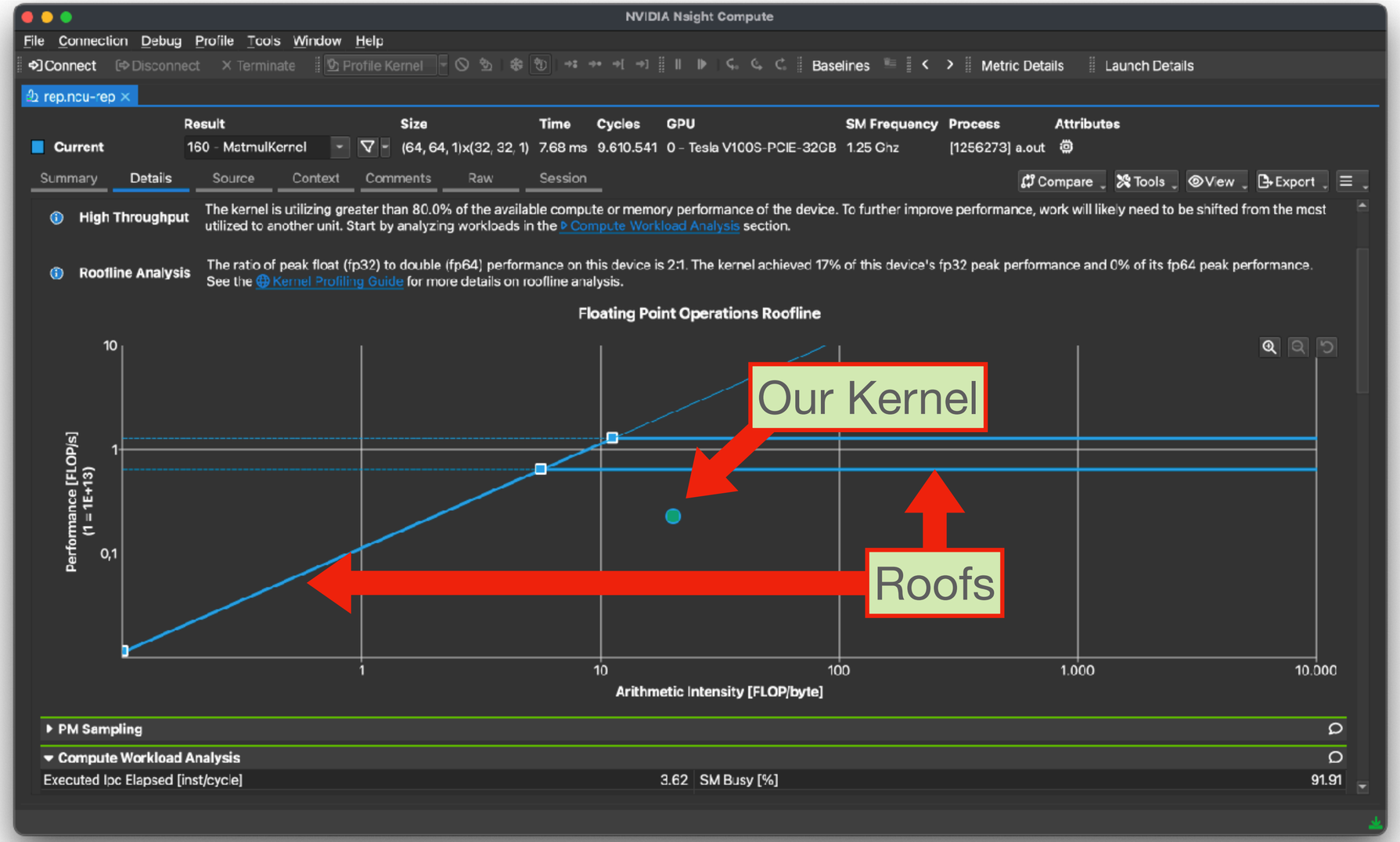
```
$ ncu --set <basic|detailed|full|roofline> ./my_program
```

- The set describes the metrics that you are going to collect.
- The more metrics will be included inside a set, the more time will be required by the profiler to profile each kernel

```
$ ncu --kernel-name <name> --section <sec> --export <name> ./my_program
```

- `--kernel-name`: profile only specific kernels
- `--section`: define the section of metrics to collect
- `--export`: save the output into a .ncu_rep file

Roofline Analysis With NCU



Wrap-Up and Tips

- Nsight Compute is the key for CUDA Kernels Optimization
- Profile first, Optimize second
- Take advantage of guided sections and roofline models
- Combine multiple sets for fine-tuned insights: SpeedOfLight, Roofline, Memory, Scheduler, etc…
- Start small (lightweight profiling) and escalate only if needed

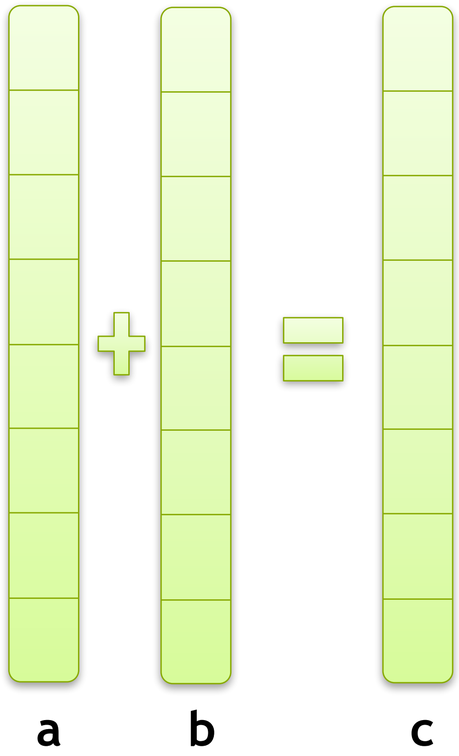
NCU Documentation

Vector Addition on the Device

Vector Addition

- Simple operation: a memory-bound operation
- Natural Fit for GPUs: Each element of a vector are independent
- Scalability: Larger vectors benefit from GPU or multi-core CPU parallelism, offering faster computation than serial processing.

```
sumArraysOnHost(float *A, float *B, float *C, const int N)
 { for (int idx=0; idx<N; idx++)
   C[idx] = A[idx] + B[idx];
}

int main(int argc, char **argv)
{
        ..
        Start = cpuSecond();
        sumArrayOnCPU(h_A, h_B, h_C, N);
        Double cpuTime = cpuSecond() - start;
        printf("CPU Execution Time: %f second \n", cpuTime);
        ..
}
```



Vector Addition on the Device

```
__global__
void sumArraysOnGPU(float* a, float* b, float* c, int n)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
  for (int i = idx; id < n; i += stride)
    c[i] = a[i] + b[i];
}

int main()
{
....
  float *d_a, *d_b, *d_c;
 CHECK_CUDA(cudaMallocManaged(&d_a, bytes));
 CHECK_CUDA(cudaMallocManaged(&d_b, bytes));
 CHECK_CUDA(cudaMallocManaged(&d_c, bytes));


 sumArraysOnGPU<<<1, 1, 0, stream>>>(d_a, d_b, d_c, N);
...
}
```

- Kernel Launch
- Compute the global index

Allocate unified memory (managed)

Triple chevron accepts four parameters:

- You know what stream is at this point
- we'll go over remaining parameters one at a time

Ensure That Data Is GPU Availability During Kernel Execution

CUDA Memory Allocation: Unified Memory

Version	Time [ns]	Speedup	Bandwidth
Single Thread	41195214843	1x	15 MB/s
Single Block (256 threads)	307066894	134.2x	1.95 GB/s
Multiple Blocks	521215	79036.9x	1151.16 GB/s

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Category	Operation
99.9	41,190,049,118	1	41,190,049,118.0	41,190,049,118.0	41,190,049,118	41,190,049,118	0.0	CUDA_KERNEL	vectorAddMultipleBlocks(float *, float *, float *, int)
0.1	21,175,962	2,300	9,206.9	3,631.5	2,206	44,320	11,980.3	MEMORY_OPER	[CUDA memcpy Unified Host-to-Device]
0.0	9,689,060	1,154	8,396.1	2,864.0	1,664	42,367	11,416.0	MEMORY_OPER	[CUDA memcpy Unified Device-to-Host]

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Category	Operation
91.4	327,339,330	1	327,339,330.0	327,339,330.0	327,339,330	327,339,330	0.0	CUDA_KERNEL	vectorAddMultipleBlocks(float *, float *, float *, int)
5.9	21,287,851	2,300	9,255.6	3,647.5	2,206	44,512	12,002.1	MEMORY_OPER	[CUDA memcpy Unified Host-to-Device]
2.7	9,678,734	1,154	8,387.1	2,863.0	1,632	42,495	11,393.4	MEMORY_OPER	[CUDA memcpy Unified Device-to-Host]

Vector Addition Performance Test
Vector size: 50000000 elements (190.73 MB per vector)

Every Thread Runs Exactly the Same Program

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Category	Operation
68.1	79,041,899	1	79,041,899.0	79,041,899.0	79,041,899	79,041,899	0.0	CUDA_KERNEL	vectorAddMultipleBlocks(float *, float *, float *, int)
23.3	27,048,873	8,192	3,301.9	2,689.0	2,303	42,592	2,893.0	MEMORY_OPER	[CUDA memcpy Unified Host-to-Device]
8.6	9,985,691	1,142	8,744.0	3,199.5	1,791	43,071	11,636.0	MEMORY_OPER	[CUDA memcpy Unified Device-to-Host]

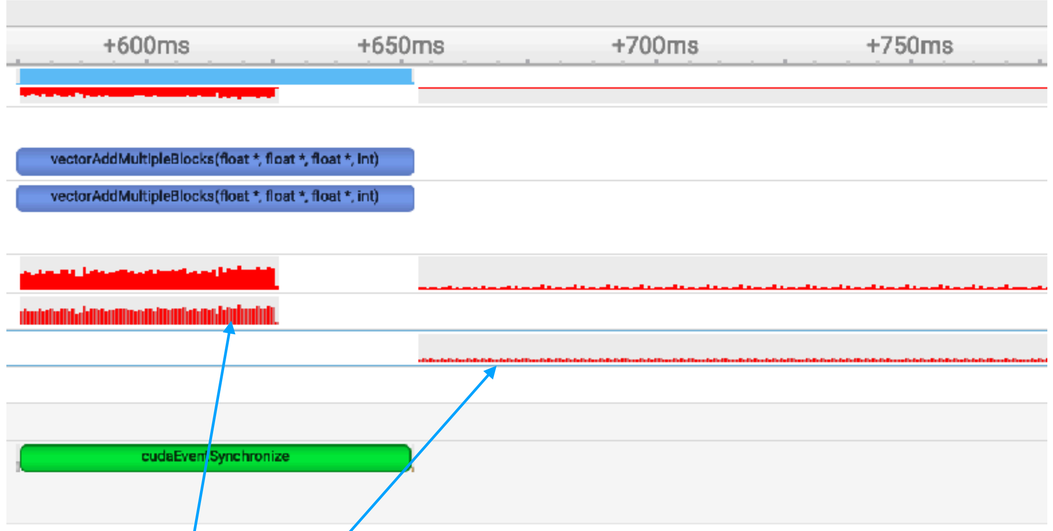
Limitation of Unified Memory

- 8192 instances host-to-device (H2D)
- 1142 instances device-to-host (D2H) “unified” `memcpy` operations
- **No explicit cudaMemcpy calls** exist — transfers are managed automatically
- CUDA **Unified Memory** uses virtual memory; pages can reside in CPU or GPU memory
- Arrays are **initialized on the CPU**, making memory pages CPU-resident initially
- When the GPU accesses them during the kernel, **page faults occur,** triggering on-demand migration
- These migrations cause many small memory transfers, leading to performance bottlenecks
- As a result, **no significant speedup** is observed due to overhead from page migrations
- Page migrations are costly because each fault triggers a separate transfer.
- GPU threads **stall** during faults, causing performance drops.

Data Prefetching To Ensure GPU Availability Before Kernel Execution

Using cudaMemPrefetchAsync() to Preload Data to GPU Before Kernel Execution

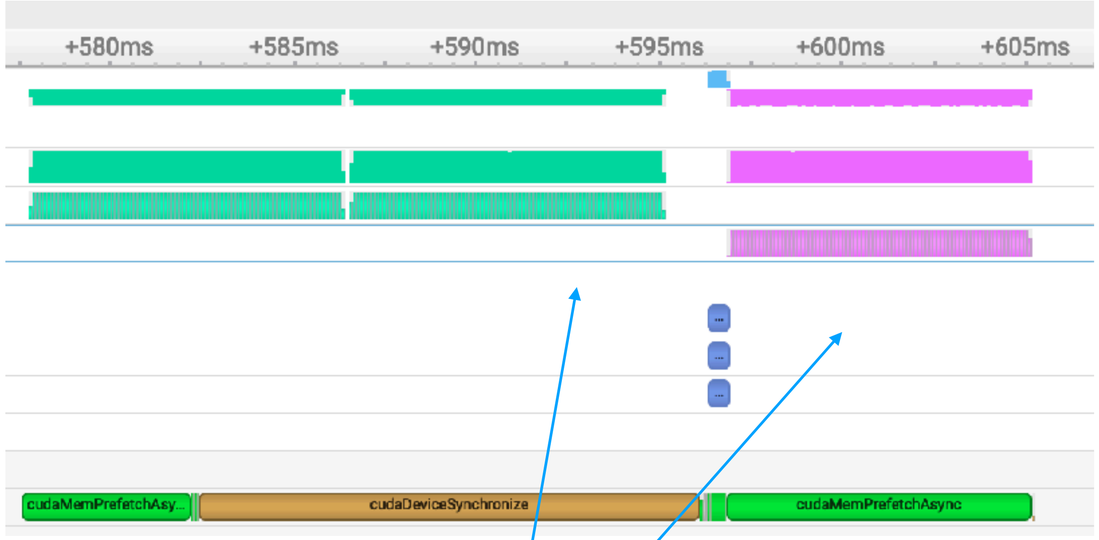
CUDA Memory Allocation: Unified Memory



Data Migration

```
Device: NVS 4200M
Transfer size (MB): 16

Pageable transfers
  Host to Device bandwidth (GB/s): 2.308439
  Device to Host bandwidth (GB/s): 2.316220

Pinned transfers
  Host to Device bandwidth (GB/s): 5.774224
  Device to Host bandwidth (GB/s): 5.958834
```

CUDA Memory Allocation: Unified Memory Prefetching



Data Migration

https://github.com/NVIDIA-developer-blog/code-samples/blob/master/series/cuda-cpp/optimize-data-transfers/profile.cu

Data Prefetching To Ensure GPU Availability Before Kernel Execution

Using cudaMemPrefetchAsync() to Preload Data to GPU Before Kernel Execution

CUDA Memory Allocation: Unified Memory

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Category	Operation
68.1	79,041,899	1	79,041,899.0	79,041,899.0	79,041,899	79,041,899	0.0	CUDA_KERNEL	vectorAddMultipleBlocks(float *, float *, float *, int)
23.3	27,048,873	8,192	3,301.9	2,689.0	2,303	42,592	2,893.0	MEMORY_OPER	[CUDA memcpy Unified Host-to-Device]
8.6	9,985,691	1,142	8,744.0	3,199.5	1,791	43,071	11,636.0	MEMORY_OPER	[CUDA memcpy Unified Device-to-Host]

Look at this pattern

CUDA Memory Allocation: Unified Memory Prefetching

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Category	Operation
66.3	16,667,938	192	86,812.2	87,296.0	33,632	88,832	5,457.1	MEMORY_OPER	[CUDA memcpy Unified Host-to-Device]
31.5	7,929,609	96	82,600.1	82,912.0	31,615	85,696	5,336.7	MEMORY_OPER	[CUDA memcpy Unified Device-to-Host]
2.2	543,550	1	543,550.0	543,550.0	543,550	543,550	0.0	CUDA_KERNEL	vectorAddMultipleBlocks(float *, float *, float *, int)

Data Prefetching To Ensure GPU Availability Before Kernel Execution

Using cudaMemPrefetchAsync() to Preload Data to GPU Before Kernel Execution

CUDA Memory Allocation: Unified Memory

Version	Time [ns]	Speedup	Bandwidth
Single Thread	41195214843	1x	15 MB/s
Single Block (256 threads)	307066894	134.2x	1.95 GB/s
Multiple Blocks	521215	79036.9x	1151.16 GB/s

CUDA Memory Allocation: Unified Memory Prefetching

Version	Time [ns]	Speedup	Bandwidth
Single Thread	42116757812	1x	14 MB/s
Single Block (256 threads)	306890747	137x	2 GB/s
Multiple Blocks	520192	80964x	1153 GB/s

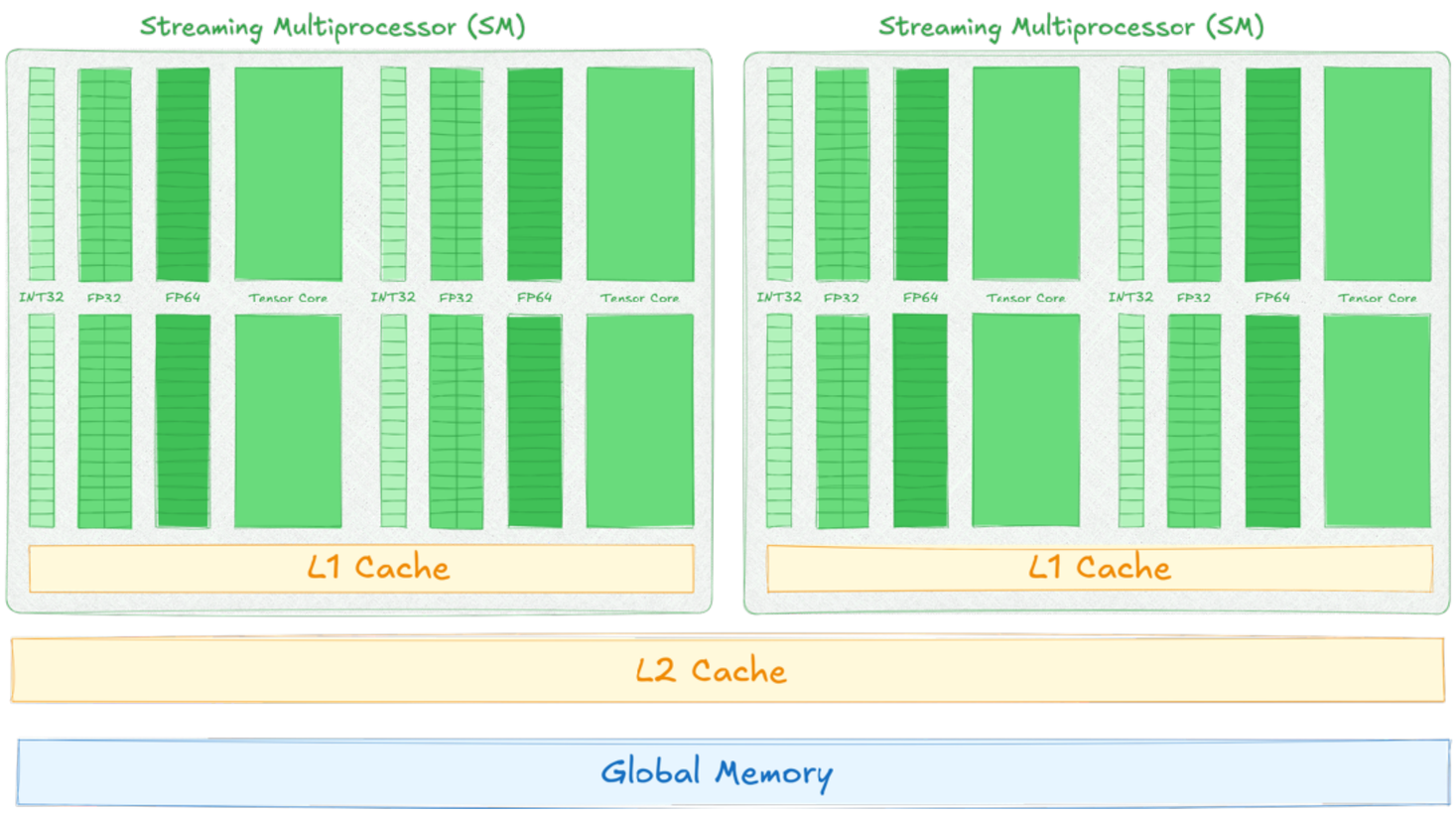
Vector Addition Performance Test
Vector size: 50000000 elements (190.73 MB per vector)

GPU Memory hierarchy

GPU vs. CPU: Understanding Performance Trade-Offs

Impact of data transfer on overall application performance

B/W	Latency
13x	1x
3x	5x
1x	15x



Registers

Fastest, smallest memory

L1 Cache

Fast, small, on-chip

Shared Memory

Medium speed, shared among threads

Global Memory

Slowest, largest, off-chip

CUDA Memory Hierarchy

CUDA Memory Hierarchy

Memory Types

Global Memory

Large, slow, accessible by all threads

```
cudaMalloc(), cudaMemcpy()
```

Shared Memory

Fast, small, shared within block

```
__shared__ float data[256];
```

Registers

Fastest, private to each thread

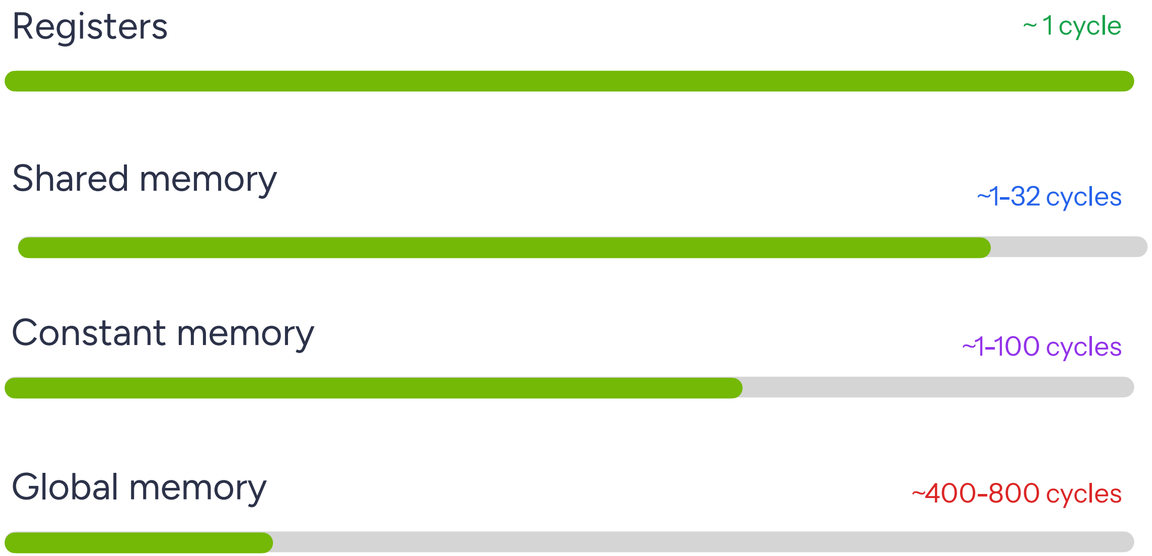
```
Automatic variables
```

Constant Memory

Read-only, cached, broadcast

```
__constant__ float params[64];
```

Memory Performance



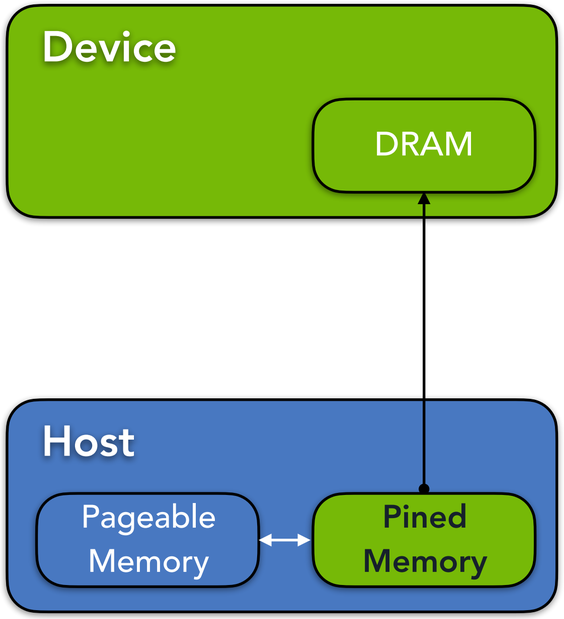
Memory Coalescing

For optimal performance, threads in a warp should access consecutive memory addresses. Coalesced memory access can improve bandwidth utilization by up to 10x compared to non-coalesced access patterns.

Global memory

Data Transfer Between Host and Device

Pageable Data Transfer



Pageable data transfer is default method

- Allocated host memory is *pageable*
- GPU cannot safely access data in pageable host memory
- When transferring data between the host and device, the CUDA driver first copies data from pageable host memory to a *page locked or pinned memory* buffer before sending it to the device
- Pageable memory in CUDA is used for memory allocation when data transfers between the CPU and GPU are infrequent

Memory Management and PCIe Transfers

CPUs and GPUs have physically distinct memory regions connected by the PCIe bus

1. **Explicit memory allocation**

   Traditional CUDA programming requires explicit memory allocation

   ```
   cudaMalloc(&d_data, size);
   ```

2. **Copy from Host to Device**

   ```
   cudaMemcpy(d_data, h_data, size, cudaMemcpyHostToDevice);
   ```

3. **Copy from Device to Host**

   ```
   cudaMemcpy(d_data, h_data, size, cudaMemcpyDeviceToHost);
   ```

4. **Clean up GPU memory**

   ```
   cudaFree(d_data);
   ```

Vector Sum Pageable Memory Transfer

Pageable Data Transfer

```
/* Host memory allocation */
   float *h_A, *h_B, *hostRef, *gpuRef;
   h_A = (float*)malloc(size);
   h_B = (float*)malloc(size);
   hostRef = (float*)malloc(size);      // Result from CPU
   gpuRef = (float *)malloc(size);     // Result from GPU

/* malloc device global memory */
   *float *d_A, *d_B, *d_C;
   checkCuda( cudaMalloc(&d_A, size) );
   checkCuda( cudaMalloc(&d_B, size) );
   checkCuda( cudaMalloc(&d_C, size) );

/* Copy data from host to device*/
   lcudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
   cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

```
/* Define block and grid sizes */
   int blockSize = 256;
   int gridSize = (nElem + blockSize - 1) / blockSize;

/* Measure time for GPU execution */
   start = cpuSecond();
   sumArraysOnGPU<<<gridSize, blockSize>>>(d_A, d_B, d_C, nElem);
   checkCuda( cudaDeviceSynchronize() );  // Ensure GPU kernel finishes
   double gpuTime = cpuSecond() - start;
   printf("GPU Execution Time: %f seconds\n", gpuTime);

/* Copy result from device to host */
  checkCuda( cudaMemcpy(gpuRef, d_C, size, cudaMemcpyDeviceToHost) );
```

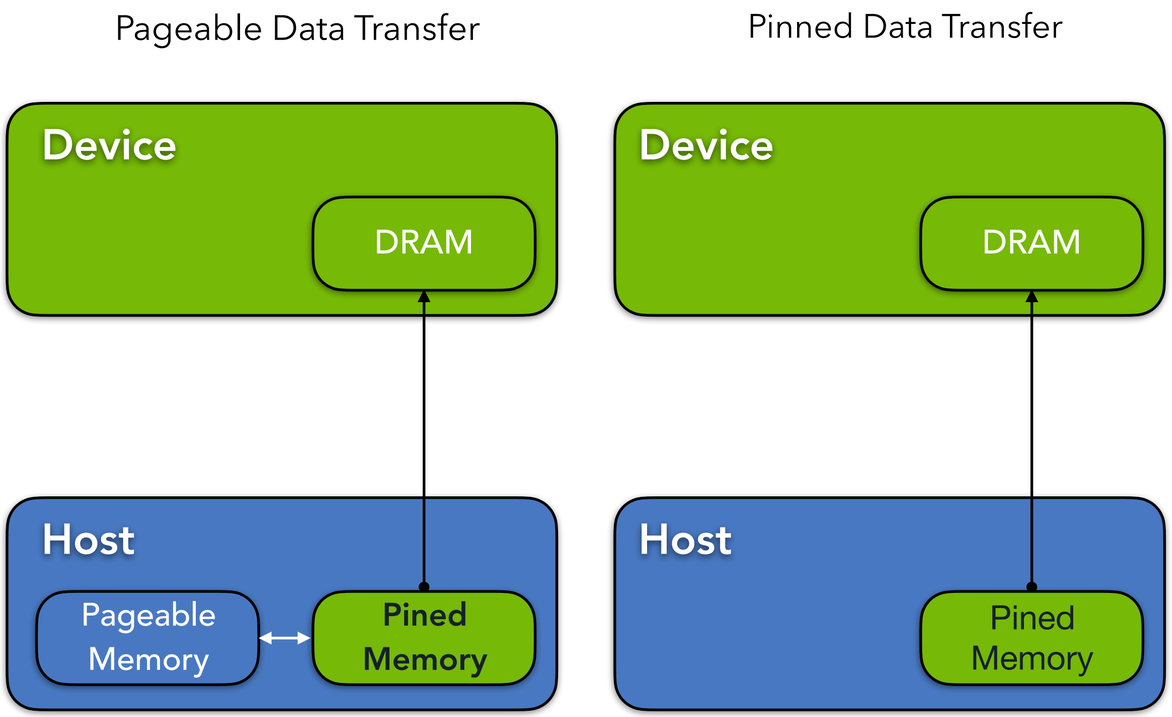
Vector Sum Pageable Memory Transfer

Pageable Data Transfer

```
/* Host memory allocation */
   float *h_A, *h_B, *hostRef, *gpuRef;
   h_A = (float*)malloc(size);
   h_B = (float*)malloc(size);
   hostRef = (float*)malloc(size);      // Result from CPU
   gpuRef = (float *)malloc(size);     // Result from GPU

/* malloc device global memory */
   *float *d_A, *d_B, *d_C;
   checkCuda( cudaMalloc(&d_A, size) );
   checkCuda( cudaMalloc(&d_B, size) );
   checkCuda( cudaMalloc(&d_C, size) );

/* Copy data from host to device*/
   lcudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
   cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

```
/* Define block and grid sizes */
   int blockSize = 256;
   int gridSize = (nElem + blockSize - 1) / blockSize;

/* Measure time for GPU execution */
   start = cpuSecond();
   sumArraysOnGPU<<<gridSize, blockSize>>>(d_A, d_B, d_C, nElem);
   checkCuda( cudaDeviceSynchronize() );  // Ensure GPU kernel finishes
   double gpuTime = cpuSecond() - start;
   printf("GPU Execution Time: %f seconds\n", gpuTime);

/* Copy result from device to host */
  checkCuda( cudaMemcpy(gpuRef, d_C, size, cudaMemcpyDeviceToHost) );
```

Data Transfer Between Host and Device



Pinned data transfer is pinned or locked

- Memory cannot be moved by the operating system
- Pinned memory is memory that is locked in physical memory and is accessible to both the CPU and the GPU
- Allocation and deallocation is expensive than pageable memory
- Provides higher transfer throughput for large data transfers

Pageable and Pinned Memory Transfer

Pageable Data Transfer

```
// allocate and initialize
int *h_a, *d_a;  // host and device specific arrays
h_a = (float*)malloc(nbytes);
cudaMalloc( &d_a, nbytes);

// memcpy H->D
cudaMemcpy( d_a, h_a, nbytes, cudaMemcpyHostToDevice);

// kernel compute
kernelGPU<<<>>>(…, d_a, …);

//cudaMemcpy D->H
cudaMemcpy( h_a, d_a, nbytes, cudaMemcpyDeviceToHost);
verifyOnHost(host_a, N);

//Free host and device memory
cudaFree(device_a); Free(host_a)
```

Pinned Data Transfer

```
// allocate and initialize
cudaMallocHost(nbytes);
cudaMalloc( &d_a, nbytes);

// memcpy H->D
cudaMemcpy( d_a, h_a, nbytes, cudaMemcpyHostToDevice);

// kernel compute
kernelGPU<<<>>>(…, d_a, …);

//cudaMemcpy  D->H
cudaMemcpy( h_a, d_a, nbytes, cudaMemcpyDeviceToHost);
verifyOnHost(host_a, N);

//Free host and device memory
cudaFree(device_a); cudaFreeHost(host_a)
```

Vector Sum Pinned Memory Transfer

Pinned Data Transfer

```
/* malloc device global memory */
   float *h_A, *h_B, *hostRef, *gpuRef;
   cudaMallocHost((void**)&h_A, size);   // Use cudaMallocHost for pinned memory
   cudaMallocHost((void**)&h_B, size);   // Use cudaMallocHost for pinned memory
   cudaMallocHost((void**)&hostRef, size); // Result from CPU
   cudaMallocHost((void**)&gpuRef, size);   // Result from GPU

/* Copy data from host to device */
   checkCuda( cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice) );
   checkCuda( cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice) );

/* malloc device global memory */
   float *d_A, *d_B, *d_C;
   checkCuda( cudaMalloc(&d_A, size) );
   checkCuda( cudaMalloc(&d_B, size) );
   checkCuda( cudaMalloc(&d_C, size) );

/* Copy data from host to device */
   checkCuda( cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice) );
   checkCuda( cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice) );
```

```
/* Define block and grid sizes */
   int blockSize = 256;
   int gridSize = (nElem + blockSize - 1) / blockSize;

/* Measure time for GPU execution */
   start = cpuSecond();
   sumArraysOnGPU<<<gridSize, blockSize>>>(d_A, d_B, d_C, nElem);
   checkCuda( cudaDeviceSynchronize() );  // Ensure GPU kernel finishes
   double gpuTime = cpuSecond() - start;
   printf("GPU Execution Time: %f seconds\n", gpuTime);

/* Copy result from device to host */
   checkCuda( cudaMemcpy(gpuRef, d_C, size, cudaMemcpyDeviceToHost) );
```

Benchmark Results

./vecAdd_managed.x

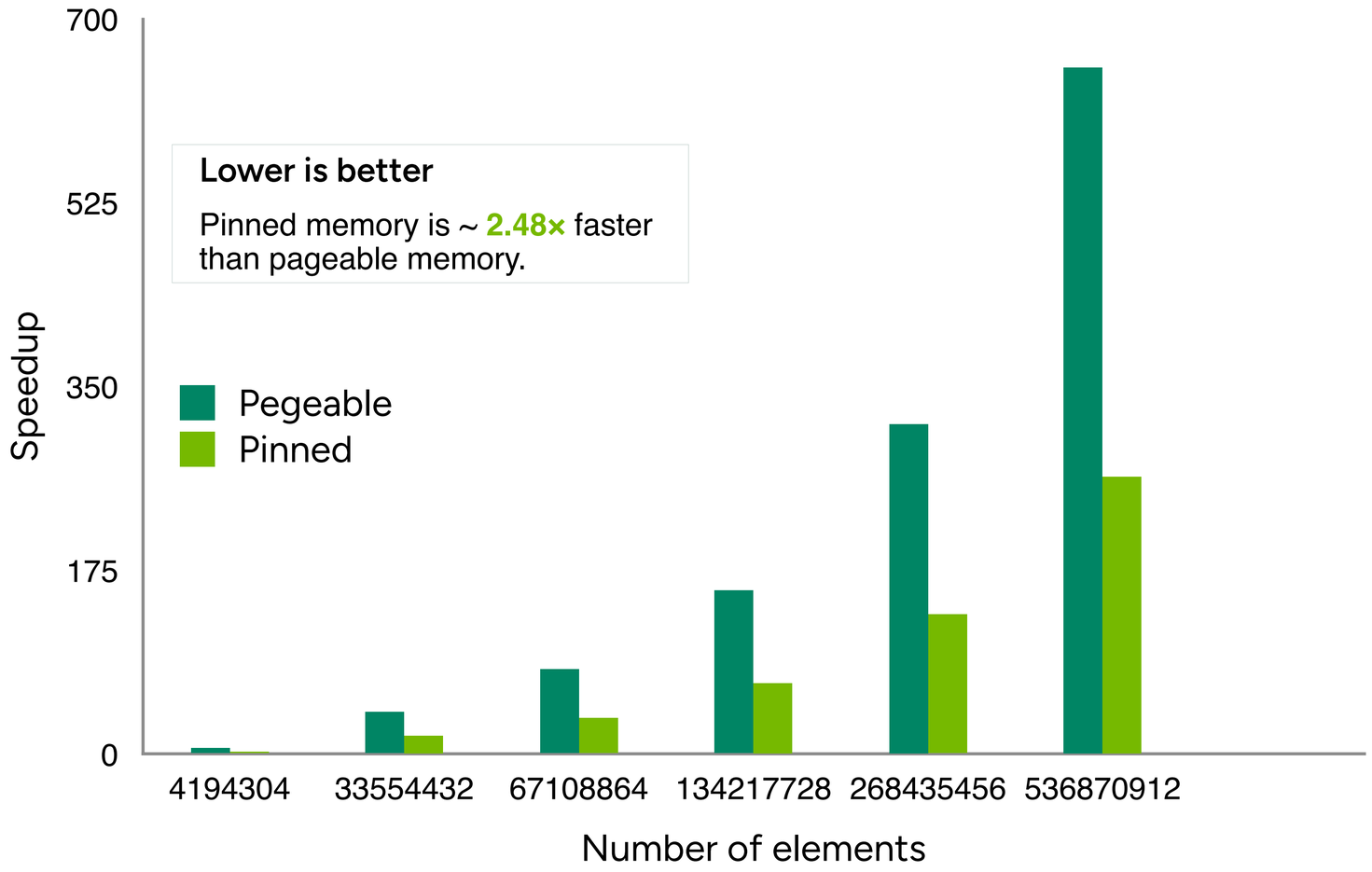
Device: NVIDIA A100-SXM-64GB
Transfer size (MB): 16
Number of elements: 4194304

Pageable transfers
Host to Device bandwidth (GB/s): 8.052961
Device to Host bandwidth (GB/s): 9.962150

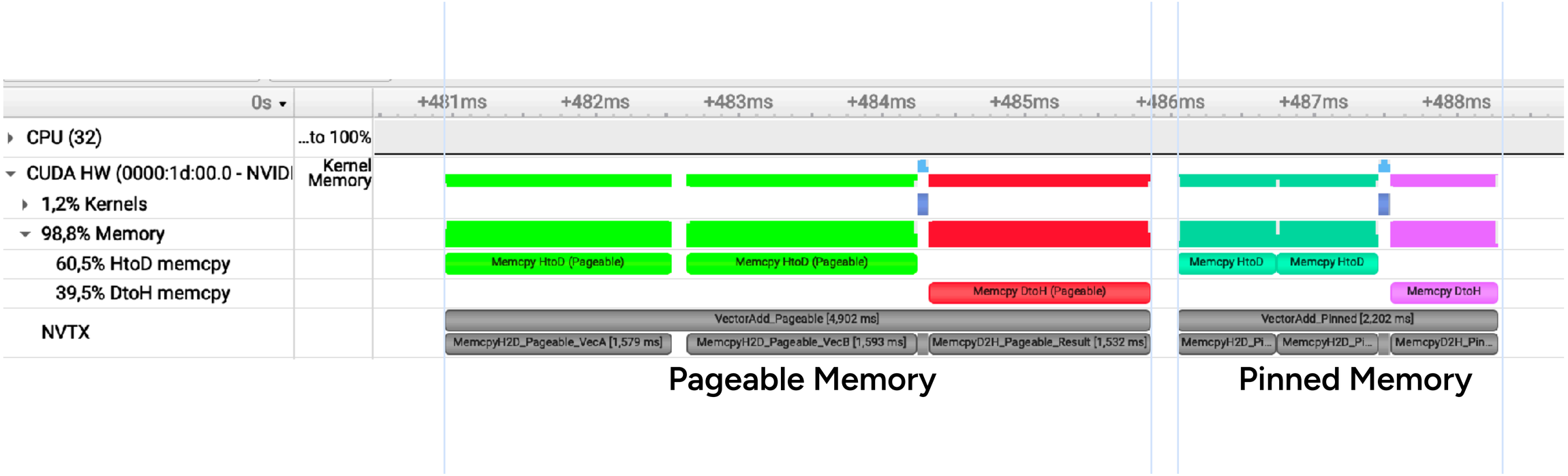
Pinned transfers
Host to Device bandwidth (GB/s): 24.277089
Device to Host bandwidth (GB/s): 24.034473

Pageable vector addition
Total time (ms): 5.395712
Effective bandwidth (GB/s): 9.328083
Vector addition successful!

Pinned vector addition
Total time (ms): 2.195232
Effective bandwidth (GB/s): 22.927713
Vector addition successful!



Application Performance Analysis



Zero-Copy Memory: Bridging Host and Device

Traditional vs Zero-Copy Approach

Zero-copy memory enables GPU threads to directly access pinned host memory without explicit transfers, reducing overhead for certain workloads and simplifying code

Zero-Copy Benefits

Best for small/medium matrices or frequent updates where transfer overhead dominates. Provides unified memory model with simpler programming paradigm

Hardware Considerations

Zero-copy excels on integrated GPU systems where the CPU and GPU share physical memory, eliminating PCI-E bus transfers entirely

```
// Traditional approach
cudaMalloc(&d_data, size);
cudaMemcpy(d_data, h_data, size,
cudaMemcpyHostToDevice);kernel<<>>(d_data);cudaMemcpy(h_data,
d_data, size, cudaMemcpyDeviceToHost);

// Zero-copy approach
cudaHostAlloc(&h_data, size, cudaHostAllocMapped);
cudaHostGetDevicePointer(&d_data, h_data, 0);kernel<<>>(d_data);
// No explicit copy needed
```

Vector Sum Zero Copy Transfer

Zero Data Transfer

```
/* Allocate and initialize host memory for zero-copy*/
   cudaHostAlloc((void**)&h_A, size, cudaHostAllocMapped);
   cudaHostAlloc((void**)&h_B, size, cudaHostAllocMapped);
   cudaHostAlloc((void**)&h_C, size, cudaHostAllocMapped);

/* Get device pointers for zero-copy memory*/
    cudaHostGetDevicePointer(&d_A, h_A, 0);
    cudaHostGetDevicePointer(&d_B, h_B, 0);
    cudaHostGetDevicePointer(&d_C, h_C, 0);

/* malloc device global memory */
    float *d_A, *d_B, *d_C;
    checkCuda( cudaMalloc(&d_A, size) );
    checkCuda( cudaMalloc(&d_B, size) );
    checkCuda( cudaMalloc(&d_C, size) );

/* Copy data from host to device */
    checkCuda( cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice) );
    checkCuda( cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice) );
```

```
/* Define block and grid sizes */
    int blockSize = 256;
    int gridSize = (nElem + blockSize - 1) / blockSize;

/* Measure time for GPU execution */
    start = cpuSecond();
    sumArraysOnGPU<<<gridSize, blockSize>>>(d_A, d_B, d_C, nElem);
    checkCuda( cudaDeviceSynchronize() );  // Ensure GPU kernel finishes
    double gpuTime = cpuSecond() - start;
    printf("GPU Execution Time: %f seconds\n", gpuTime);

/* Copy result from device to host */
   checkCuda( cudaMemcpy(gpuRef, d_C, size, cudaMemcpyDeviceToHost) );
```

Comparison of Zero-Copy Memory vs Device Memory

SIZE	Device memory (ELAPSED TIME [s])	Zero-copy Memory (ELAPSED TIME [s])	SlowDown
1 KB	0,000033	0,000014	0,424242424242424
4 KB	0,007286	0,002334	0,320340378808674
16 KB	0,007289	0,002335	0,320345726437097
64 KB	0,001673	0,002342	1,39988045427376
256 kB	0,002434	0,002358	0,968775677896467
1 MB	0,002446	0,002524	1,03188879803761
4 MB	0,000849	0,000454	0,534746760895171
16 MB	0,004292	0,004123	0,960624417520969
64 MB	0,012136	0,007024	0,578773895847067
256 MB	0,051559	0,029347	0,569192575496034

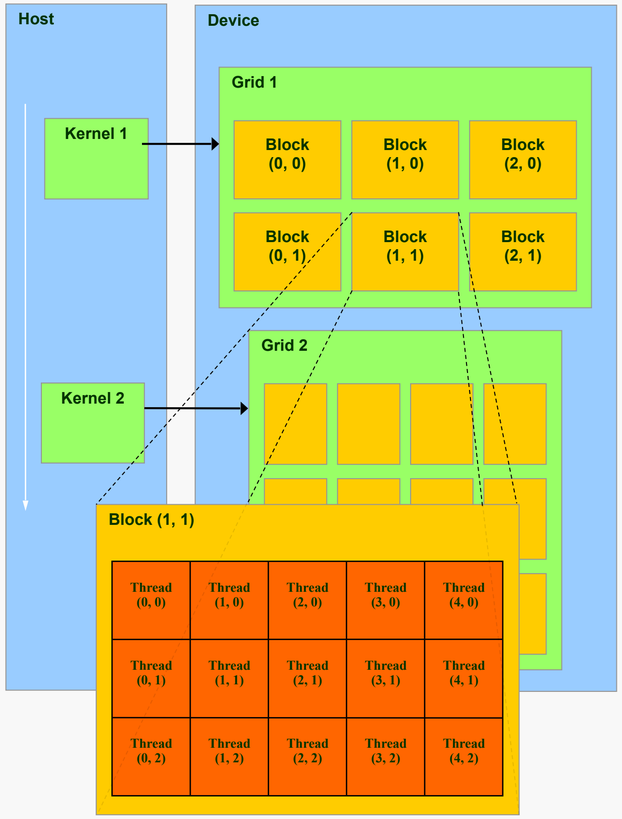
Implementing higher dimensional grid in CUDA

Multidimensional Blocks and Grids

Host program specifies “grid-block-threads” configuration for kernel at run time

- All threads spawned by a single kernel launch are collectively called a *grid*
- All threads in a grid share the same global memory space
- A grid is made up of many thread blocks
- Kernel needs to know run-time configuration
- Built-in-three-dimensional type for threads (uint3) and blocks (dim3)
    - threadIdx.x, threadIdx.y, threadIdx.z
    - blockIdx.x, blockIdx.y, blockIdx.z
    - blockDim.x, blockDim.y, blockDim.z

Grid Dimension: 3x2 = 6 Blocks



Block Dimension: 5x3 = 15 Threads/Blocks
(6 ~~Blocks~~) x(15 Threads/ ~~Blocks~~) = 90 Total threads in Grid

Thread Indexing

```
__global__ void threadIndexExample(float* data) {
    // 1D indexing
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    // 2D indexing
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int idx_2d = row * gridDim.x * blockDim.x + col;

    // 3D indexing
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    int z = blockIdx.z * blockDim.z + threadIdx.z;

    int idx_3d = z * (gridDim.x * blockDim.x) * (gridDim.y * blockDim.y) + y * (gridDim.x * blockDim.x) + x;

    // Built-in variables available:
    // threadIdx.x, threadIdx.y, threadIdx.z
    // blockIdx.x, blockIdx.y, blockIdx.z
    // blockDim.x, blockDim.y, blockDim.z
    // gridDim.x, gridDim.y, gridDim.z
}
```

Device Run-Time Configuration

Type	Variable	Description
dim3	gridDim	Dimensions of grid
uint3	blockIdx	Index of block within grid
dim3	blockDim	Dimensions of block
uint3	ThreadIdx	Index of thread within block

Dimension	Variable	ID
1D	(Dx)	x
2D	(Dx, Dy)	y + y*Dx
3D	(Dx, Dy, Dz)	z + y*Dx + z*DxDy

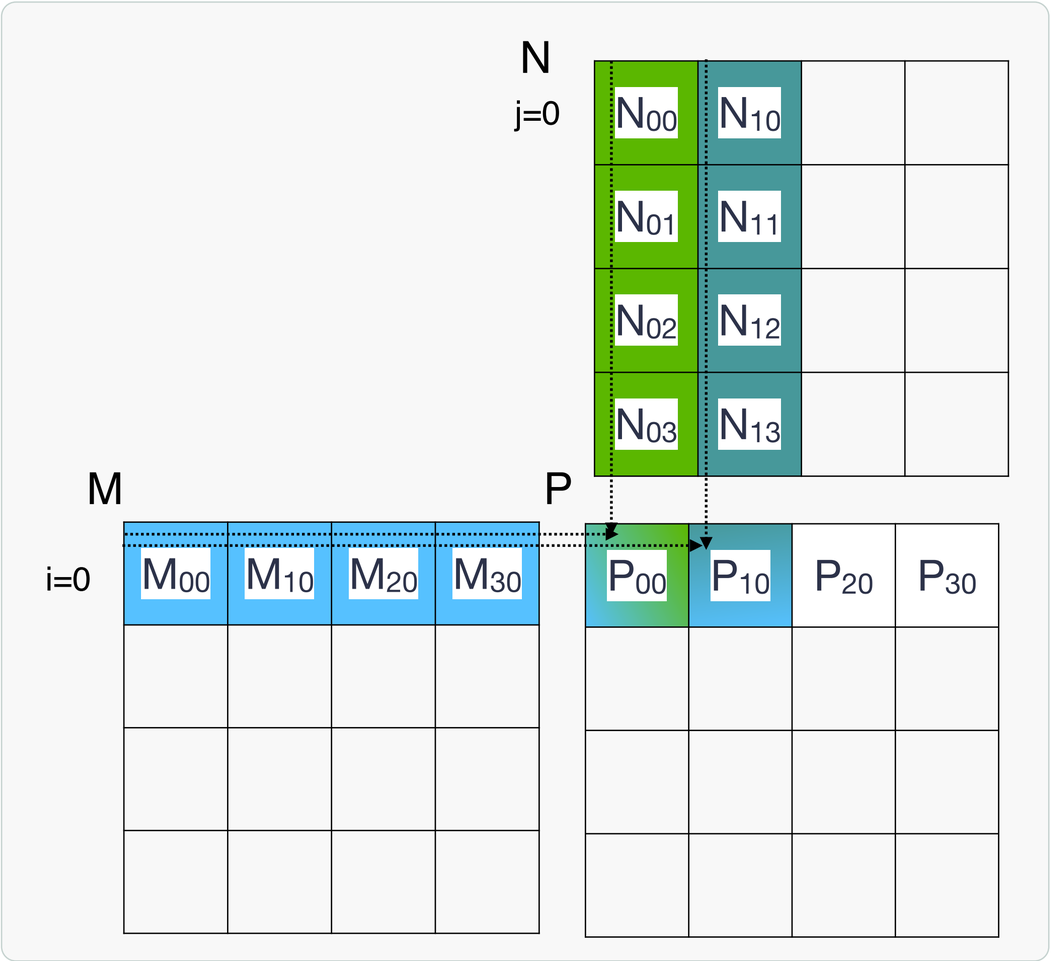
Exercise-3: Two matrix multiplication

Two Matrix Multiplication

$$P_{ij} = \sum_{k=1}^{n} M_{ik} \cdot N_{kj}$$

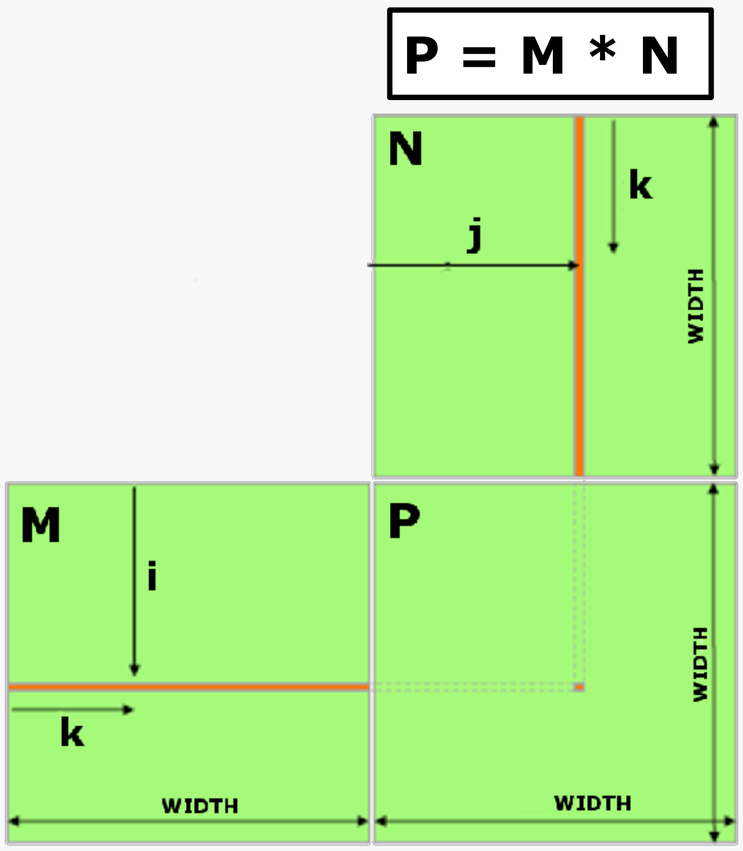
$$P_{10} = M_{00} * N_{10} + M_{10} * N_{11} + M_{20} * N_{12} + +M_{30} * N_{13}$$

$$P_{00} = M_{00} * N_{00} + M_{10} * N_{10} + M_{20} * N_{20} + +M_{30} * N_{30}$$



Two Matrix Multiplication

```
void matrixMultOnHost(float* M, float* N, float* P, int Width){
   for (int row = 0; row < Width; ++row){
      for (int col = 0; col < Width; ++col){

   // accumulate element-wise products
      float pval = 0;
       for (int k = 0; k < Width; ++k){
        float a = M[row*Width + k];
        float b = M[k*Width + col];
        pval += a*b;
        }
        P[row*width + col] = pval;
      }
   }
}
```



Things to do

The starting point of this exercise contains a working host function, called matrixMulCPU. Your task is to build out the matrixMulGPU CUDA kernel. The source code will execute the matrix multiplication with both functions, and compare their answers to verify the correctness of your CUDA kernel.

Follow these guidelines.

T1. Create an execution configuration whose arguments are both dim3 values with the x and y dimensions set to greater than 1.

T2. Inside the body of the kernel, establish the running thread's unique index within the grid as usual, but you should define two indices for the thread: one for the x axis of the grid, and one for the y axis of the grid.

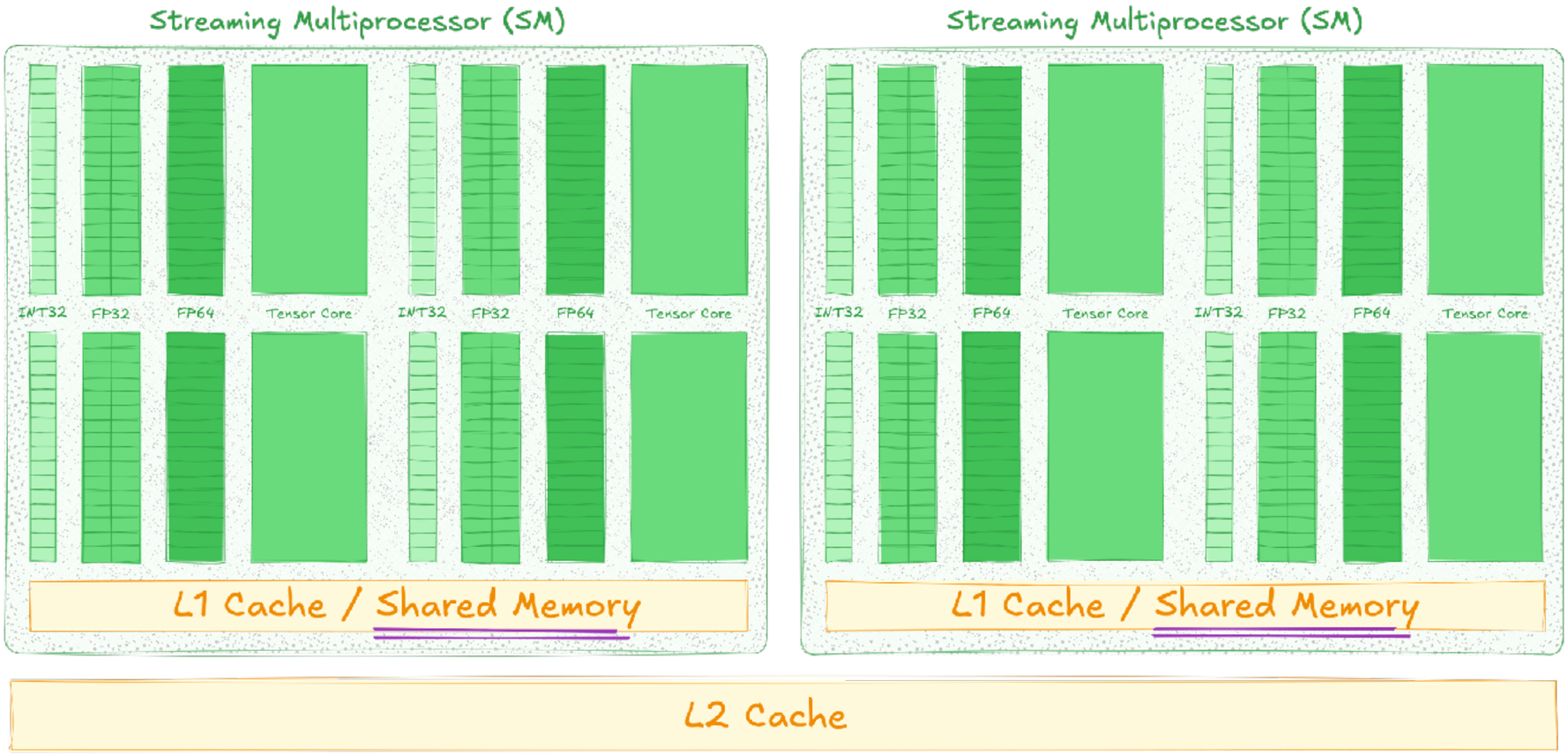
T3. Use pageable and pinned memory,

T4. Optional managed memory

Shared memory

Shared Memory

- CUDA provides a software-defined cache that's called shared memory
- Shared memory is only accessible within a given thread block and is collocated with the L1



Shared Memory

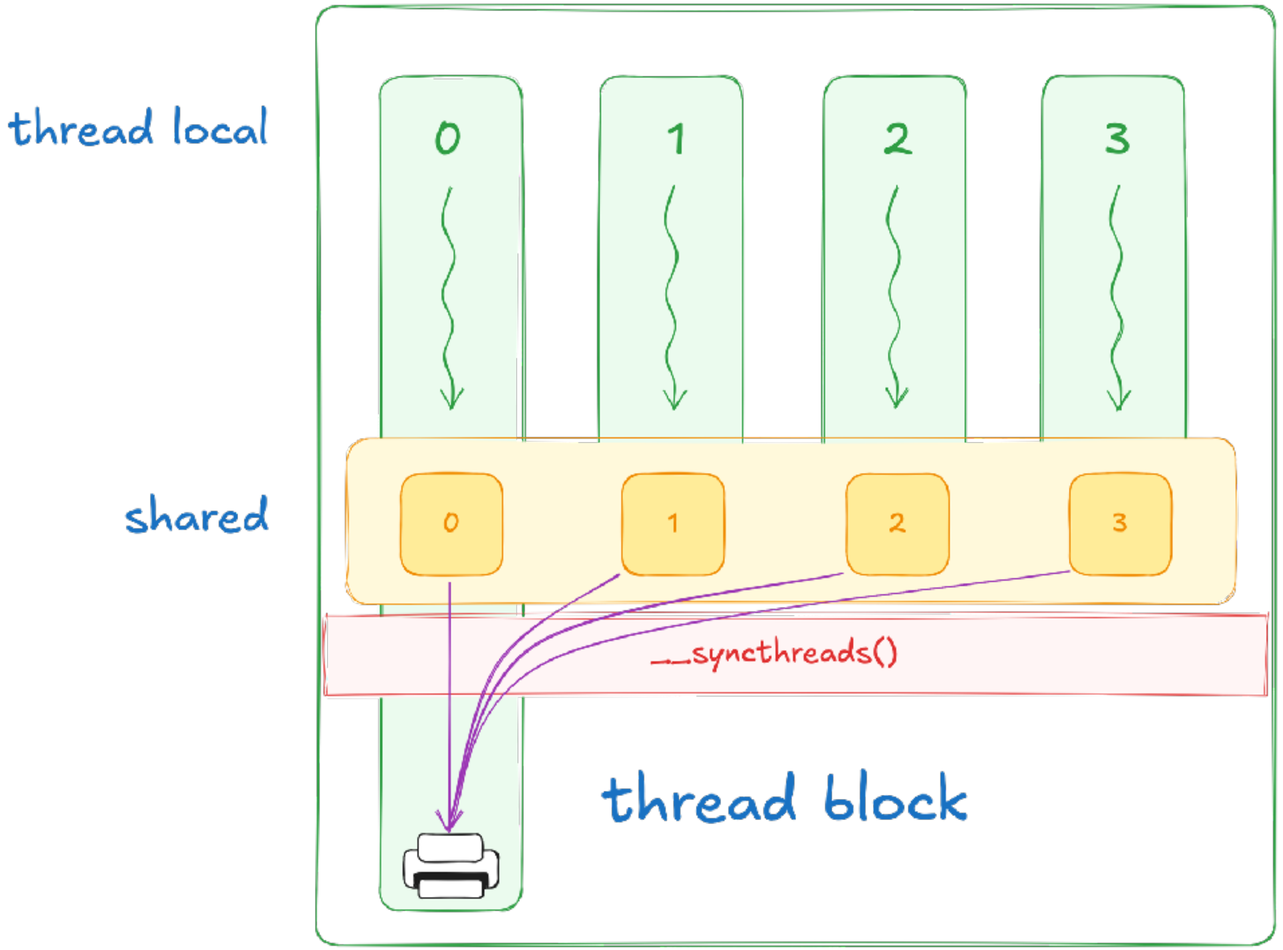
The __shared__ qualifier is used to declare shared memory variables in CUDA kernels

Sharing data between threads

- Each SM has 16 KB of Shared Memory
  - 16 banks of 32bit words
- CUDA uses Shared Memory as shared storage visible to all threads in a thread block
  - read and write access
- Declare using __shared__ keyword, allocated per block
  - data is not visible to threads in other blocks

__syncthreads() – Thread Synchronization

- Synchronizes all threads in a block (barrier)
- Ensures shared memory accesses are complete
- Prevents race conditions and data hazards
- All threads must reach the barrier to avoid hangs
- Conditional use must be uniform across the block



Shared Memory

The __shared__ qualifier is used to declare shared memory variables in CUDA kernels

```
__global__ void sharedMemoryMatrixMultiplicationKernel(float* M, float* N, float* P, int Width) {
    __shared__ float sharedM[BLOCK_SIZE][BLOCK_SIZE]; __shared__ float sharedN[BLOCK_SIZE][BLOCK_SIZE];
    int row = blockIdx.y * blockDim.y + threadIdx.y; int col = blockIdx.x * blockDim.x + threadIdx.x;

    float sum = 0.0f;
    for (int m = 0; m < (Width + BLOCK_SIZE - 1) / BLOCK_SIZE; ++m) {
        // Load elements into shared memory
        if (m * BLOCK_SIZE + threadIdx.x < Width && row < Width) {
            sharedM[threadIdx.y][threadIdx.x] = M[row * Width + m * BLOCK_SIZE + threadIdx.x];
        } else {
            sharedM[threadIdx.y][threadIdx.x] = 0.0f; // Fill with zero if out of bounds
        }

        if (m * BLOCK_SIZE + threadIdx.y < Width && col < Width) {
            sharedN[threadIdx.y][threadIdx.x] = N[(m * BLOCK_SIZE + threadIdx.y) * Width + col];
        } else {
            sharedN[threadIdx.y][threadIdx.x] = 0.0f; // Fill with zero if out of bounds
        }

        __syncthreads(); // Synchronize to make sure all threads have loaded their data

        // Perform the multiplication
        for (int k = 0; k < BLOCK_SIZE; ++k) {
            sum += sharedM[threadIdx.y][k] * sharedN[k][threadIdx.x];
        }
        __syncthreads();// Synchronize before loading the next tile
    }
    // Write the result to global memory
    if (row < Width && col < Width) {
        P[row * Width + col] = sum;
    }
}
```

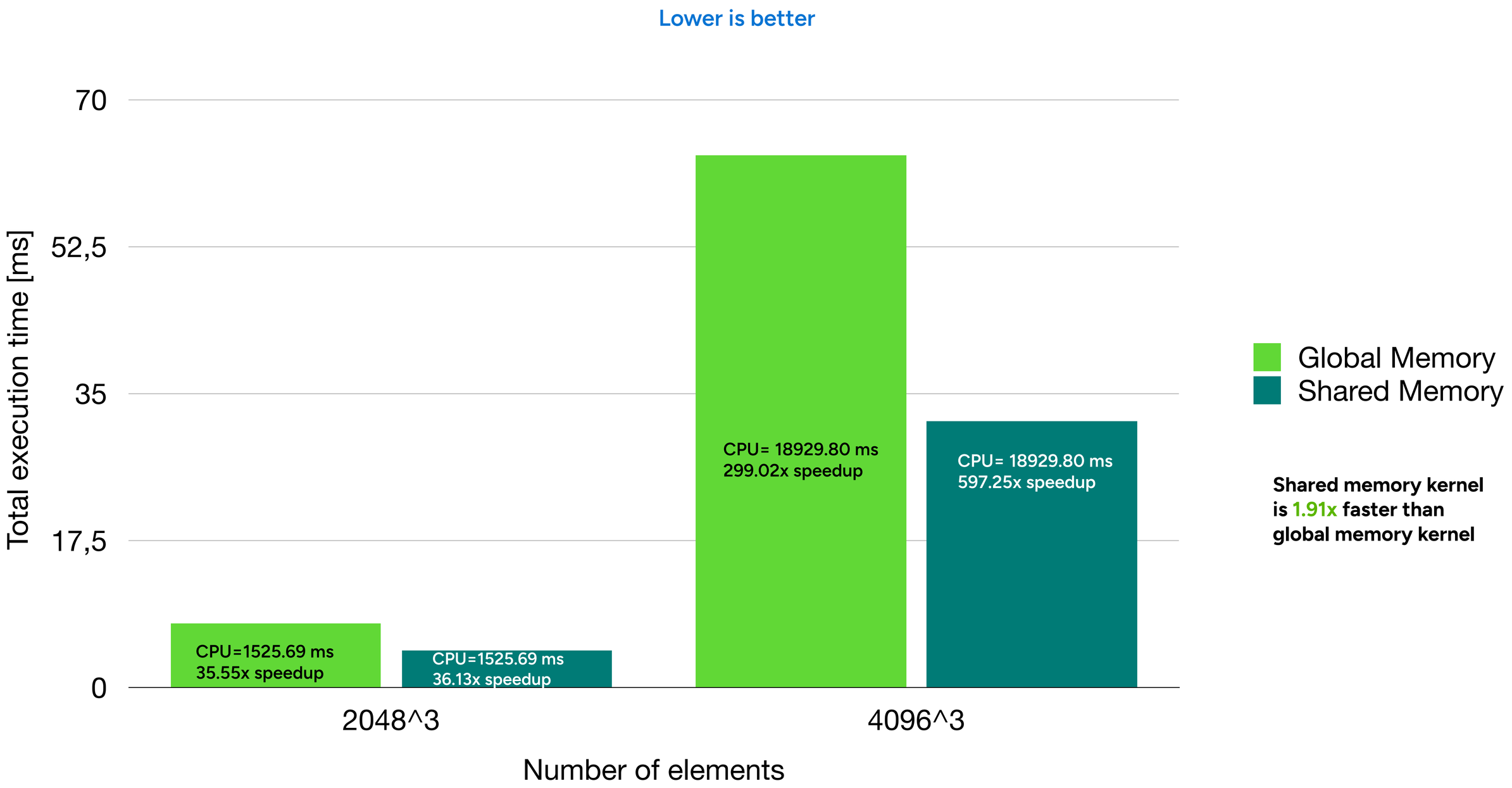
Tile Loading: Each thread loads a small tile (sub-matrix) from M and N into shared memory.

Synchronization: Threads synchronize to make sure all data is loaded before starting multiplication.

Dot Product Calculation: Each thread computes a part of the matrix multiplication by performing a dot product on the loaded tiles.

Write Back: Once the computation is complete, the result is stored in matrix P in global memory.

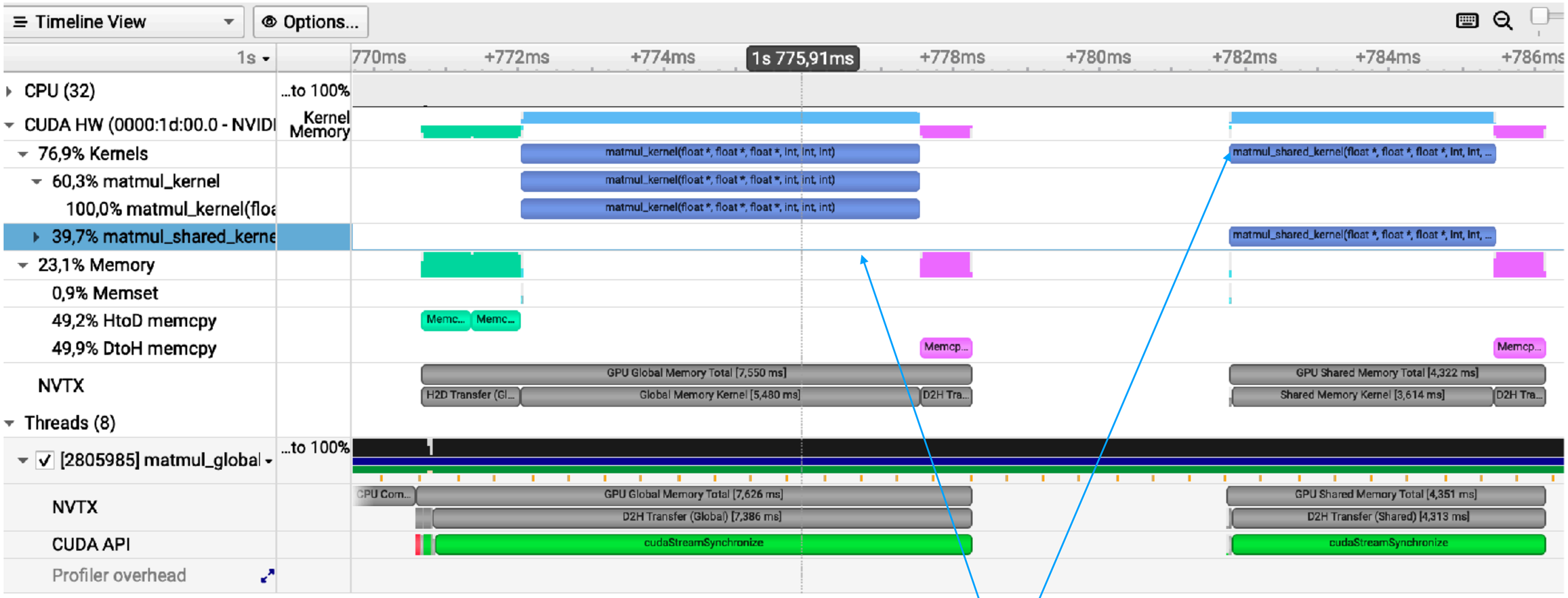
Memory Access Patterns in CUDA: Benchmarking Global and Shared Memory Efficiency



Shared memory kernel is 1.91x faster than global memory kernel

Performance Comparison Between Global and Shared Memory

Lower is better



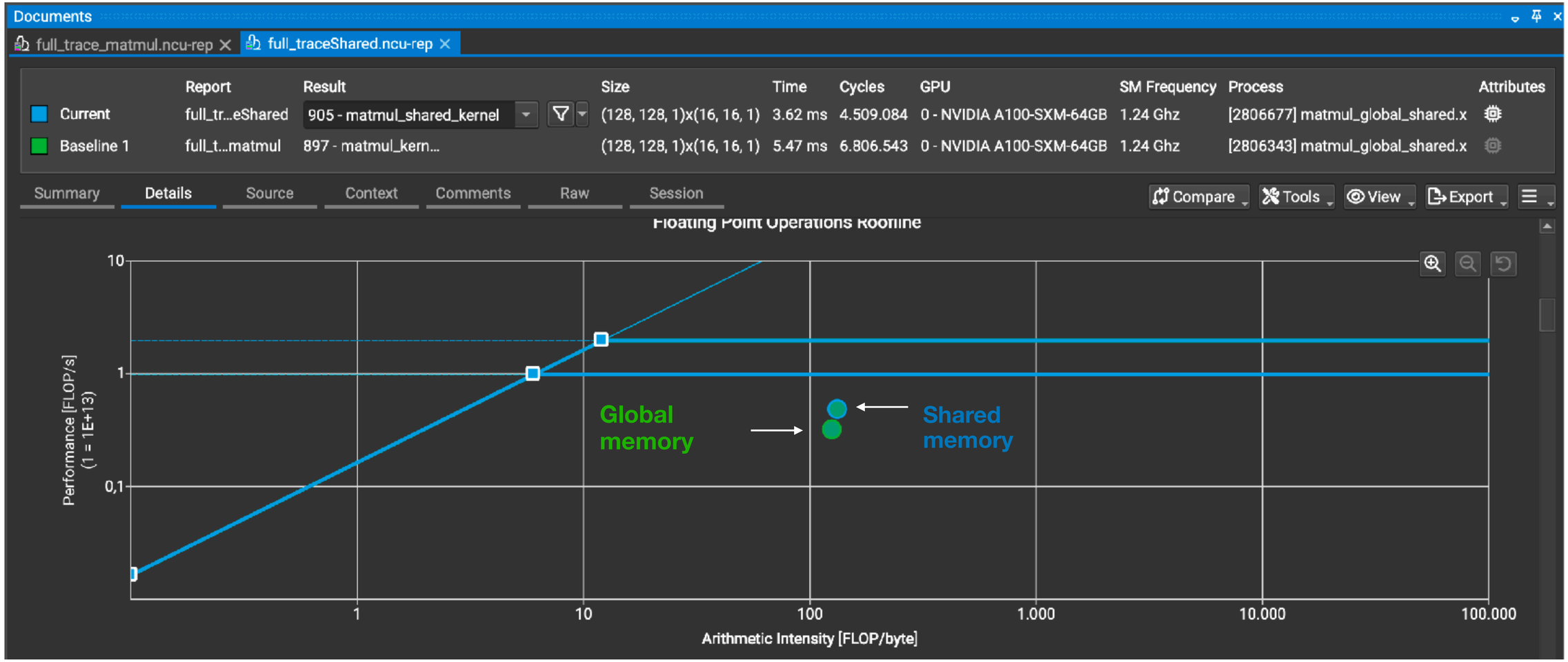
Look at this pattern

Performance Comparison Between Default and Non-Default Stream

Lower is better

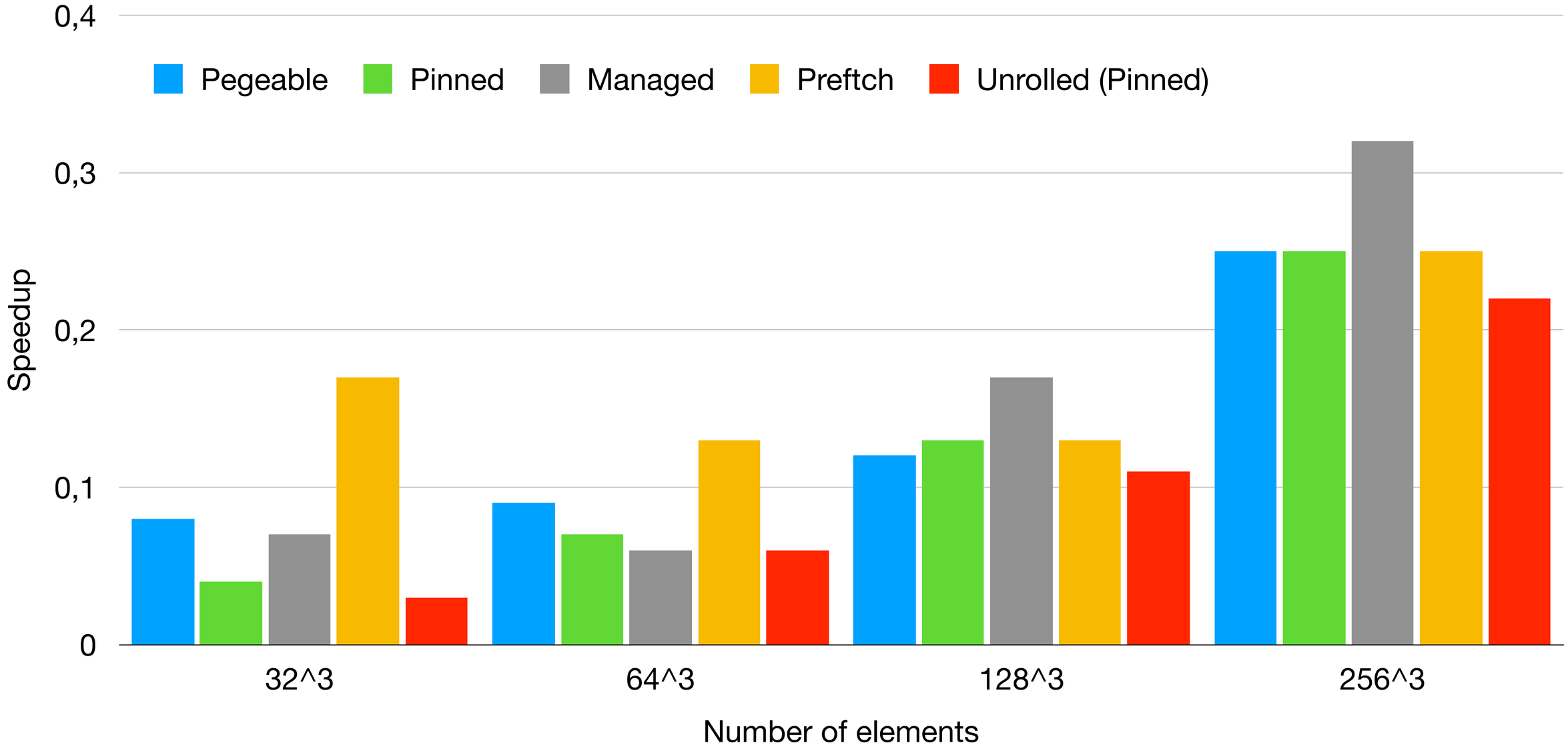
Performance Comparison Between Default and Non-Default Stream

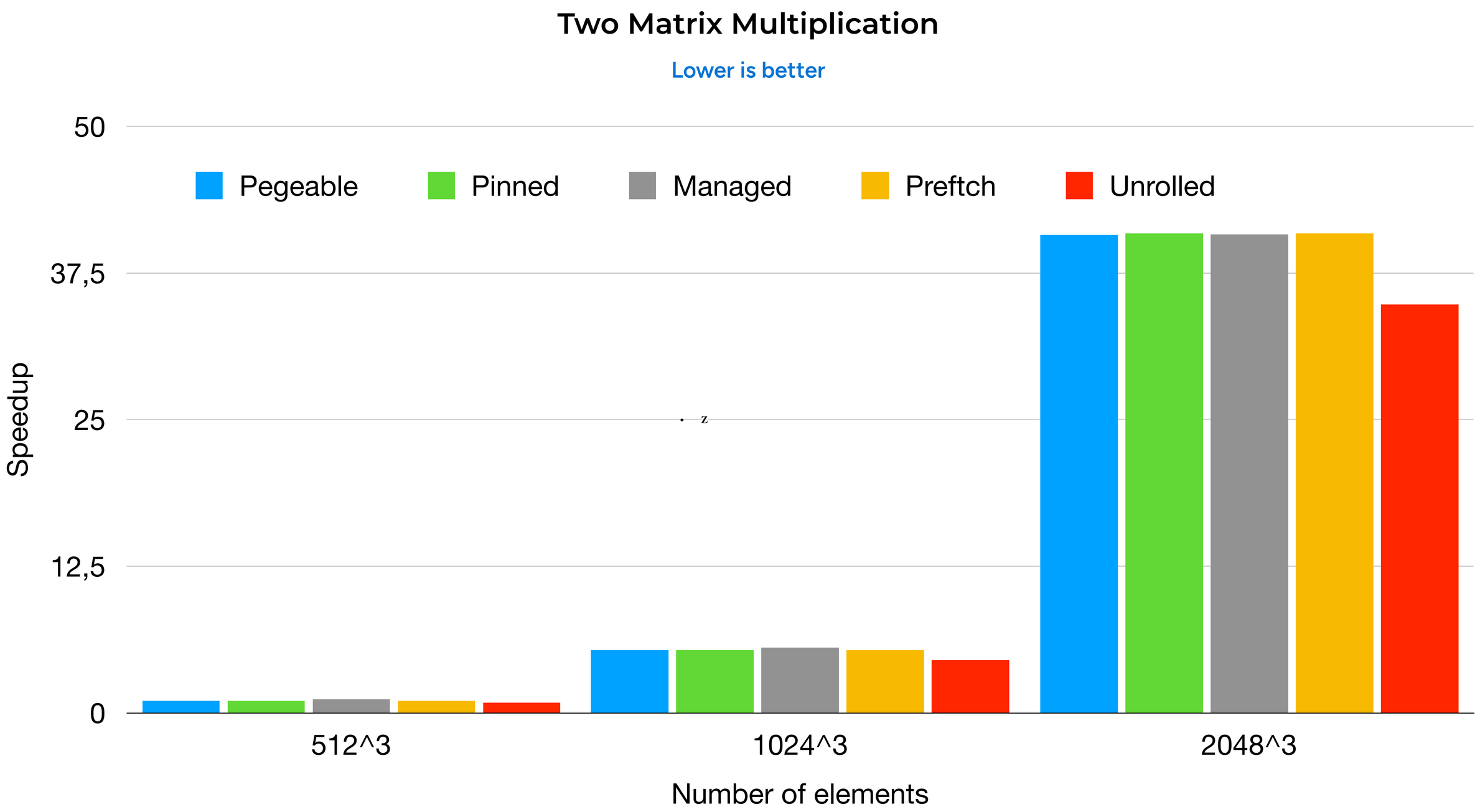
Lower is better



Two Matrix Multiplication

Lower is better




Two Matrix Multiplication
Lower is better
50
37,5
25
12,5
0
Speedup
Pegeable
Pinned
Managed
Preftch
Unrolled
512^3
1024^3
2048^3
Number of elements

How to Overlap Data Transfers in CUDA C/C++

What Is a CUDA STREAM?

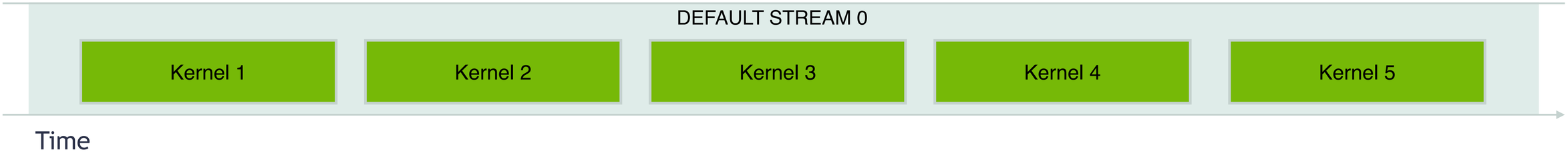
1

Sequence of CUDA operations

kernel execution, memory transfer that execute in issue-order on the GPU

By default, CUDA kernels are executed in a default stream

Instructions are executed in order (in any stream): an instruction must be completed before the next one can begin



Non-Default Stream Behaviour

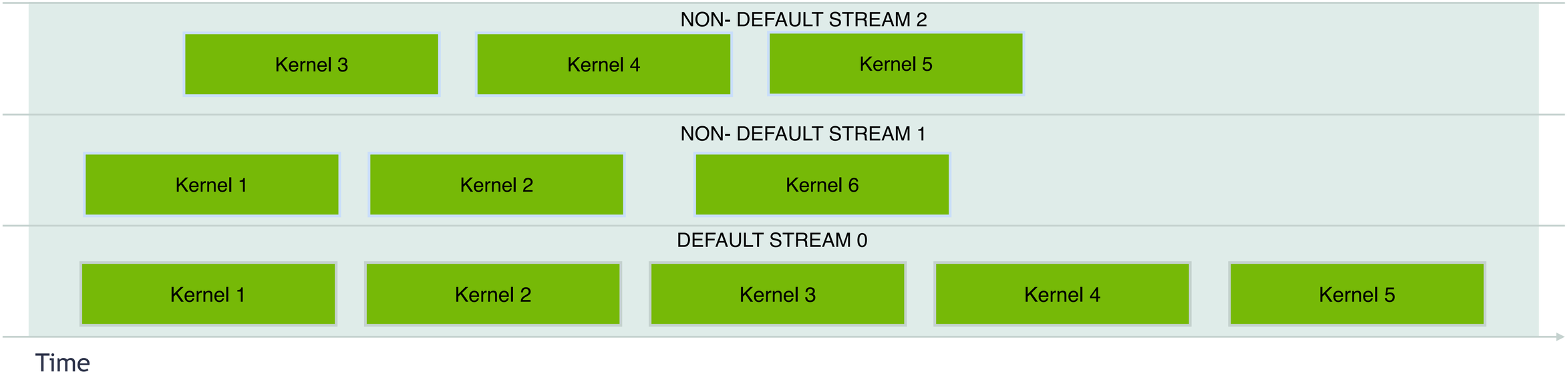
2

Rules of governing the behaviour of streams

Multiple streams or Non-default streams can be created and utilise by CUDA programmers

Kernels, with any single STREAM must execute in order

However, kernels in different, non-default streams, can interact concurrently, have no fixed order of execution



Where It Can Be Useful?

1

Kernel Enqueuing

Kernels are enqueued into a specific stream for execution on the GPU.

2

Memory Transfer

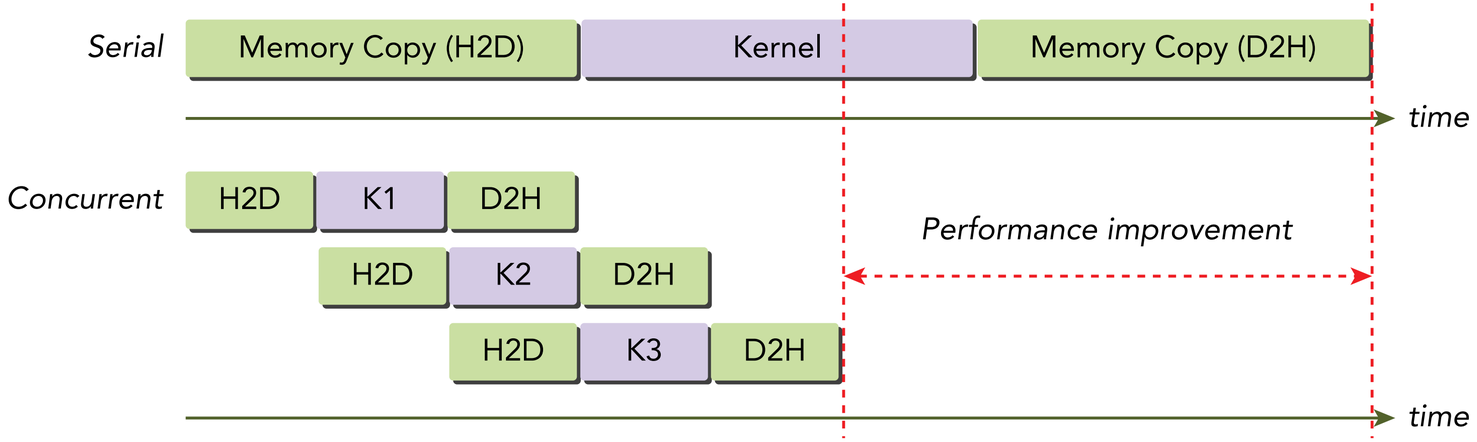
Data transfers between host and device can be enqueued asynchronously into streams.

3

Overlapped Execution

The GPU can execute kernels and memory transfers concurrently in different streams.

Asynchronous Execution with Streams



Overlapping Kernel Execution and Data Transfers

1

How to use streams in a CUDA program?

```
cudaStream_t stream;  cudaStreamCreate(&stream);        // Note that a pointer must be passed to `cudaCreateStream`.
```

2

How to use streams in a CUDA program?

```
someKernel<<<number_of_blocks, threads_per_block, 0,stream>>>();
```

3

How to Destroying Non-Default CUDA Streams?

```
cudaStreamDestroy (stream);
```

4

Blocking and Non-blocking streams

cudaStreamcreate is blocking streams, there is also exists non-blocking streams - But we do not cover it here

CUDA Stream Synchronization

- Explicit
  - cudaDeviceSynchronize()
    - Blocks until all CUDA operations are finished
  - cudaStreamSynchronize(stream))
    - Blocks until all CUDA operations are finished within given stream
  - cudaEvenRecord(event, stream1), cudaStreamWaitEvent(stream2, event)
    - Blocks until all CUDA operations are finished within given stream

- Implicit
  - Page-locked memory allocation
    - cudaMallocHost, cudaHostAlloc
  - Device memory allocation
    - cudaMalloc
  - Blocking version of memory operations
    - cudaMemcpy, cudaMemset
  - Implicit synchronize all CUDA operations

Basic non-Stream Usage

- Split data:
- first half → stream1
- second half → stream2
- Launch two async copies, two kernels, two copies back
- Synchronize both streams

```
// Create two streams
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

// Create timing events
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);

// First half using stream1
cudaMemcpyAsync(d_data, h_data, halfSize, cudaMemcpyHostToDevice, stream1);
int threads = 256;
int blocks1 = (halfN + threads - 1) / threads;
processData<<<blocks1, threads, 0, stream1>>>(d_data, halfN);
cudaMemcpyAsync(h_data, d_data, halfSize, cudaMemcpyDeviceToHost, stream1);

// Second half using stream2
cudaMemcpyAsync(d_data + halfN, h_data + halfN, halfSize, cudaMemcpyHostToDevice, stream2);
int blocks2 = (halfN + threads - 1) / threads;
processData<<<blocks2, threads, 0, stream2>>>(d_data + halfN, halfN);
cudaMemcpyAsync(h_data + halfN, d_data + halfN, halfSize, cudaMemcpyDeviceToHost, stream2);

// Synchronize both streams
cudaStreamSynchronize(stream1);
cudaStreamSynchronize(stream2);
```

Overlapping Computation and Memory Transfer

- Splits 1M elements into 4 streams
- Uses cudaMemcpyAsync and kernel launches per stream
- Waits for all streams to finish
- Includes a working processChunk kernel
- Prints timing and a few result samples

```
int main() {
  int N = 1024 * 1024;
  int num_streams = 4;
  int chunk_size = N / num_streams;
  size_t chunk_bytes = chunk_size * sizeof(float);

  // Allocate page-locked host memory and device memory
  .....

  // Create streams
    cudaStream_t streams[num_streams];
    for (int i = 0; i < num_streams; i++) {
        cudaStreamCreate(&streams[i]);
    }
    // Process data in chunks using multiple streams
    for (int i = 0; i < num_streams; i++) {
        int offset = i * chunk_size;
        // Async copy H2D
        cudaMemcpyAsync(&d_data[offset], &h_data[offset], chunk_bytes,
                        cudaMemcpyHostToDevice, streams[i]);
        // Kernel launch
        int threads = 256;
        int blocks = (chunk_size + threads - 1) / threads;
        processChunk<<<blocks, threads, 0, streams[i]>>>(&d_data[offset],
chunk_size);
        // Async copy D2H
        cudaMemcpyAsync(&h_data[offset], &d_data[offset], chunk_bytes,
                        cudaMemcpyDeviceToHost, streams[i]);
    }

  // Wait for all streams
    for (int i = 0; i < num_streams; i++) {
        cudaStreamSynchronize(streams[i]);
        cudaStreamDestroy(streams[i]);
    }

  return 0;
}
```

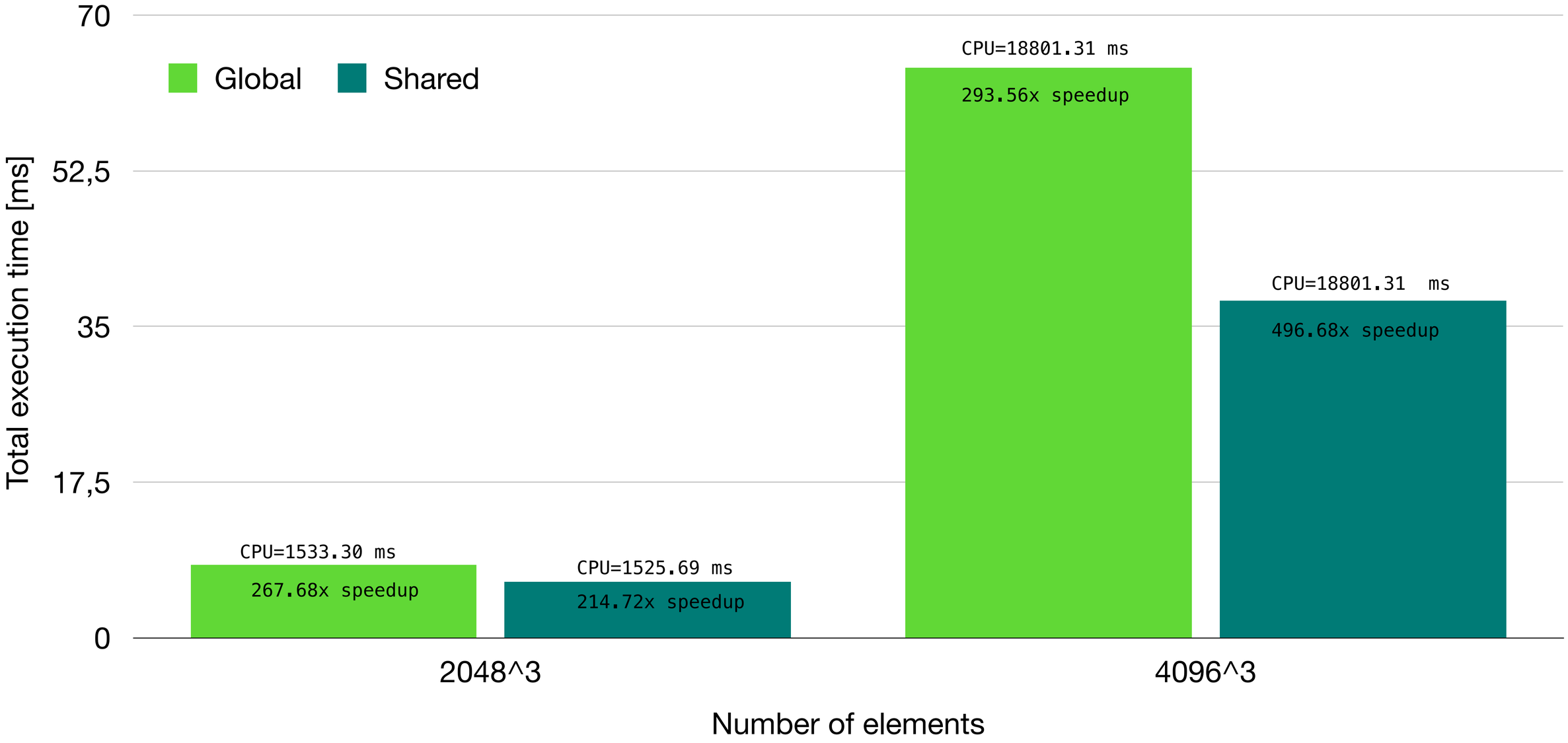
Performance Comparison Between Default and Non-Default Stream

Lower is better

Implementation	Description
CPU (Naive)	Single-threaded triple loop matrix multiplication.
Basic GPU	Single kernel launch, synchronous memory copies, no streams.
Streamed GPU	Multi-stream asynchronous memory transfer and kernel launches with pinned host memory.

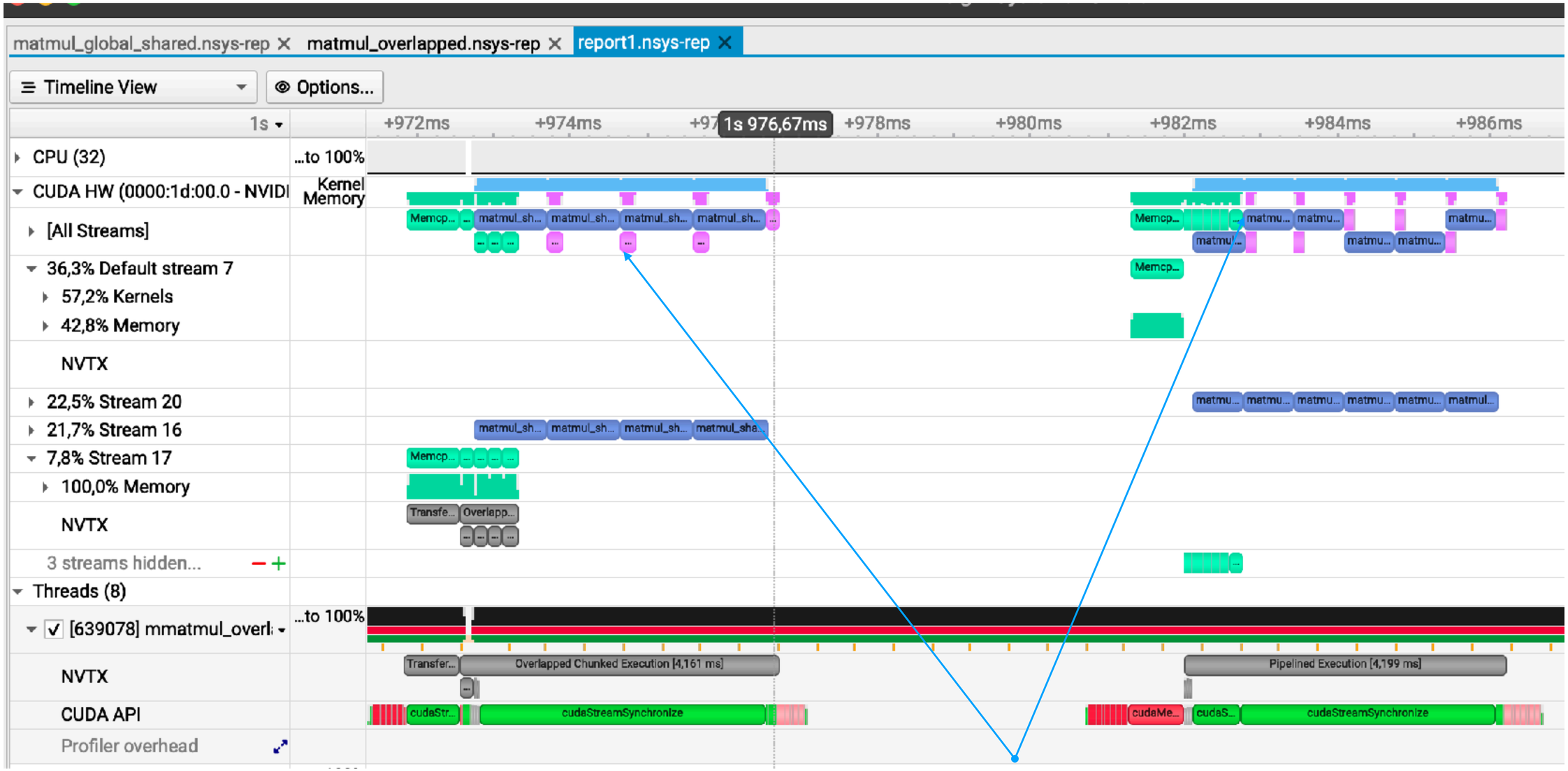
Performance Comparison Between Default and Non-Default Stream

Lower is better



CINECA

Nsight System Report



Look at this pattern

Common Pitfalls and Solutions

1. Race conditions → Use proper synchronization
2. Memory leaks → Implement RAII patterns
3. Uncoalesced memory access → Optimize access patterns
4. Low occupancy → Balance resources and threads
5. Excessive host-device transfers → Use streams and batching
6. Ignoring error checking → Always check CUDA API calls

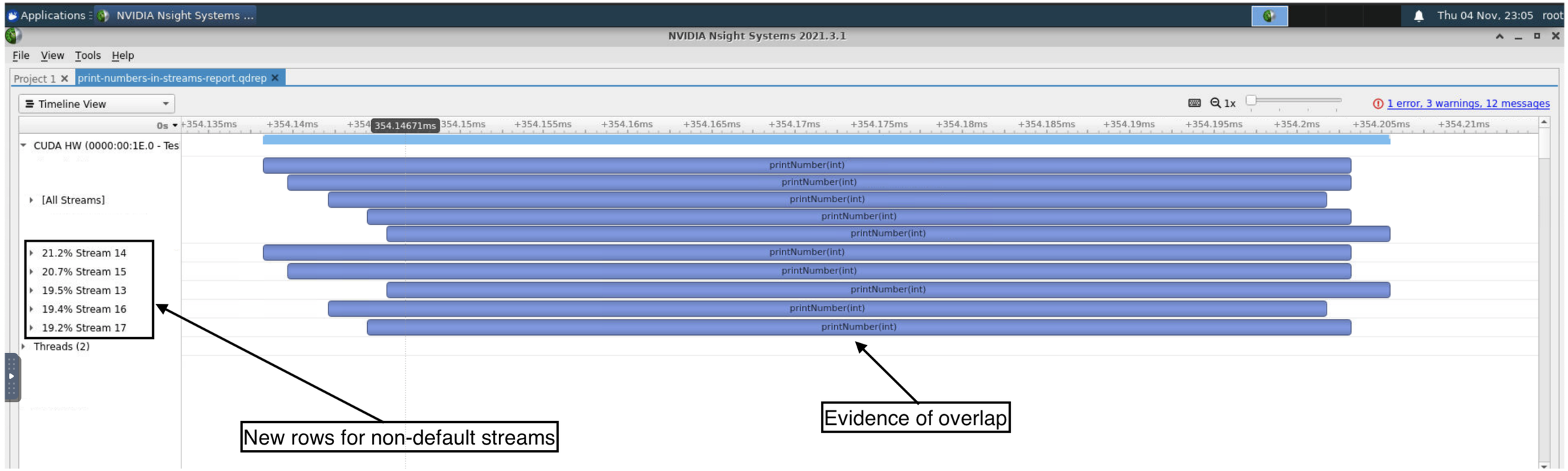
Exercise-4: creating streams

Things To Do

This exercise starts from a very simple printNumber kernel which accepts an integer and prints it. The kernel is being executed 5 times, using a for- loop, and passing each launch the number of the for-loop's iteration. These iterations run serially since they are all in the default stream.

Refactor the code so that each kernel launch occurs in its own non-default streams.

- Will kernels now run in parallel?



Vector_Add_Streams.Cu

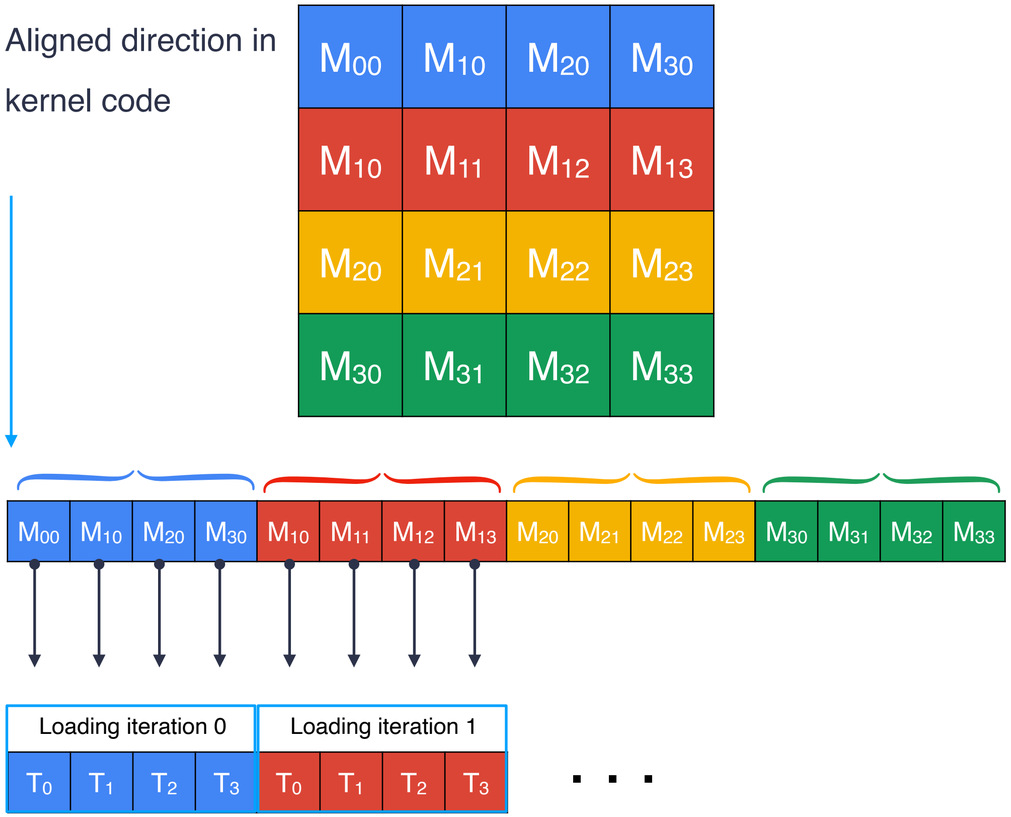
Currently, it launches an initialization kernel 3 times, once for each of the 3 vectors needed in the vectorAdd kernel.

- Refactor it to launch each of the 3 initialization kernel launches in their own non-default stream.
- Be sure to still see the success message print when compiling and running your solution.

Performance Optimization Techniques

Memory Coalescing

Key strategies for maximizing GPU performance

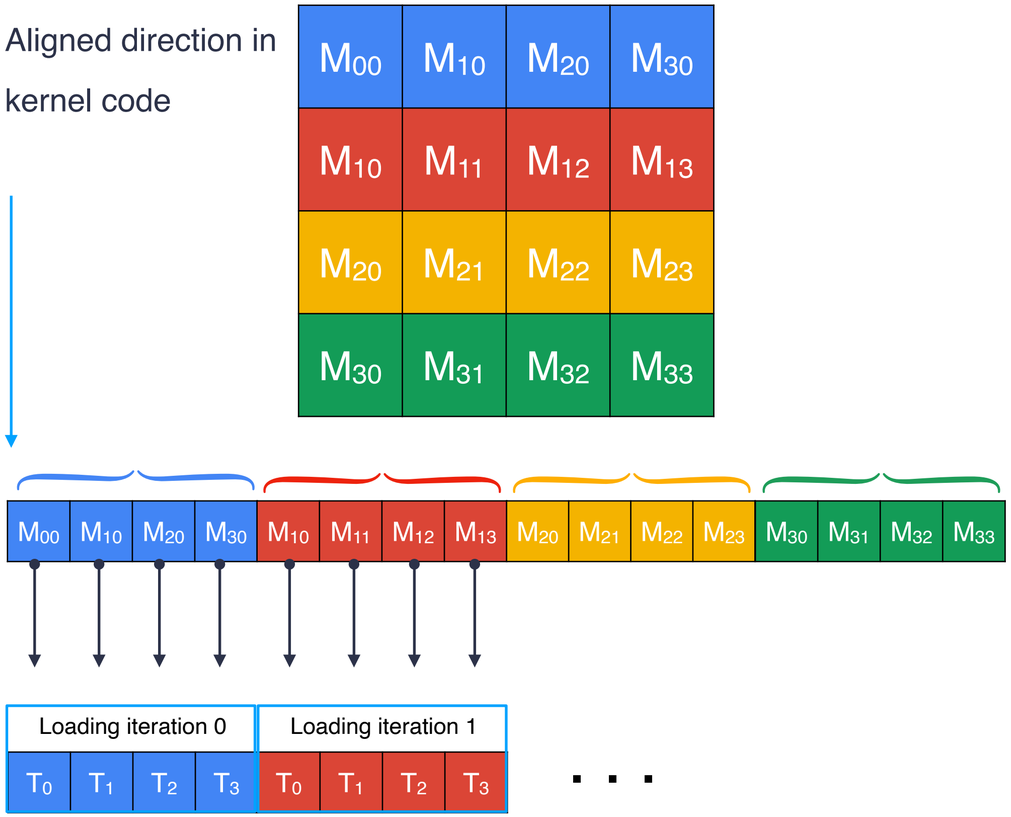


- For blocks that consist of multiple dimensions of threads, the dimensions will be projected into a linear order before partitioning into warps
- Each thread is shown as M(x,y), with x being the threadIdx.x and y being threadIdx.y for the thread
- Cooperatively, the 32 threads in a warp present a single memory access request comprised of the requested addresses, which is serviced by one or more device memory transactions.

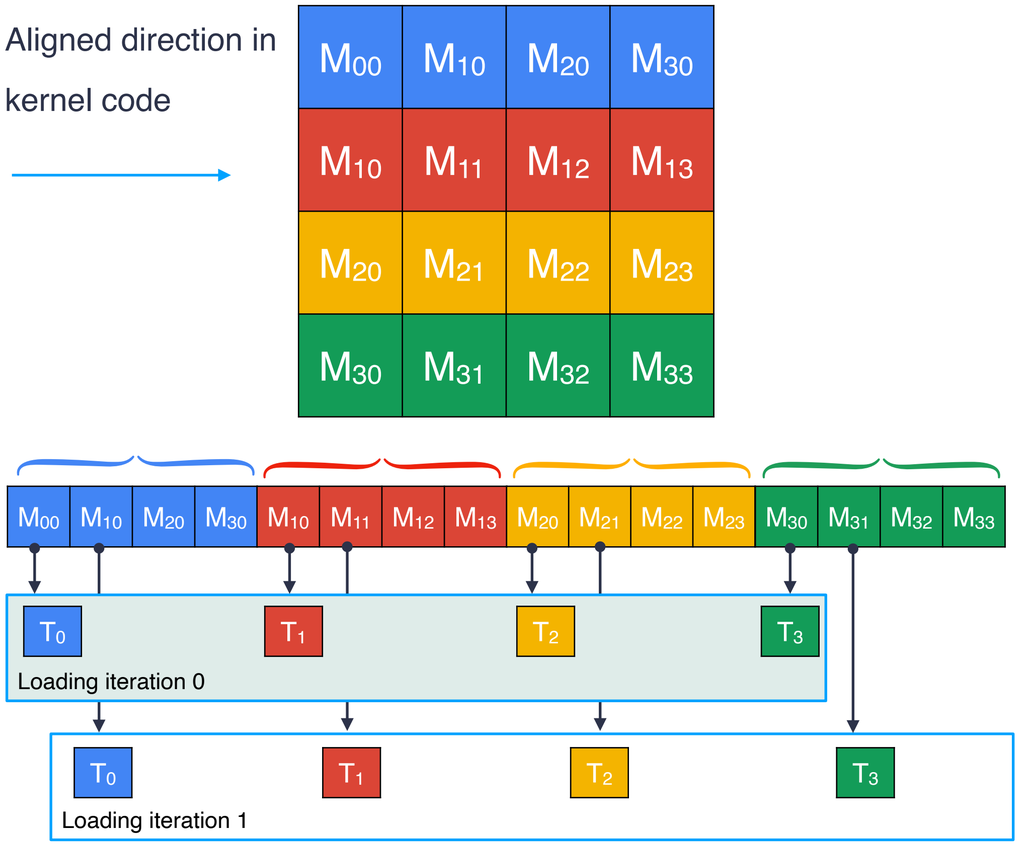
Memory Coalescing

Key strategies for maximizing GPU performance

Aligned direction in kernel code



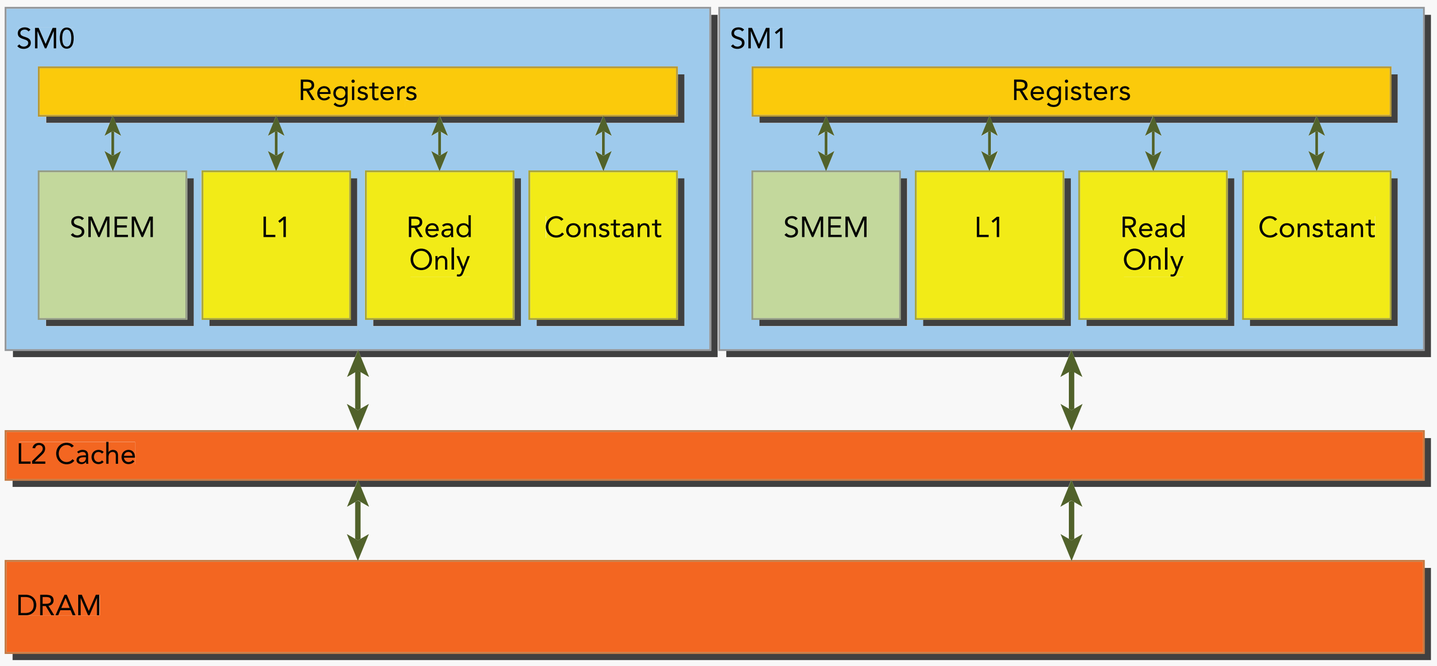
Aligned direction in kernel code



Memory Bandwidth Limits GPU-Enabled Applications

Key strategies for maximizing GPU performance

- Memory operations are issued per warp, with each thread providing its own memory address
- **Global memory loads/stores** are staged through L2 and sometimes L1 caches
- **Global memory accesses** go through L2 cache, with optional L1 cache usage based on architecture
- **Memory transactions** use 128-byte or 32-byte segments, depending on cache involvement
- **L1 cache lines** are 128 bytes and map to 128-byte aligned segments in device memory
- **Perfect mapping** occurs when each thread in a warp requests one 4-byte value, matching the 128-byte cache line size



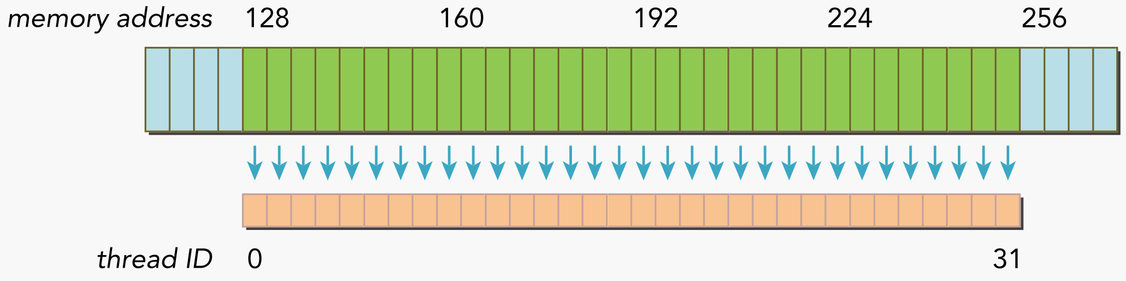
Efficient Memory Access Is Crucial

Key strategies for maximizing GPU performance

Aligned Memory Access

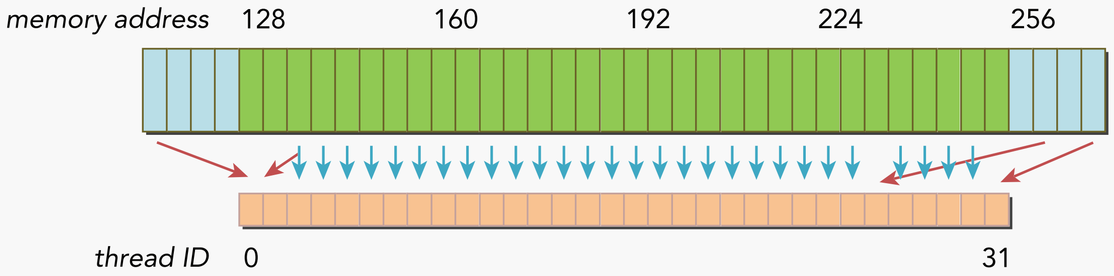
accessed by threads are arranged such that each thread accesses data in consecutive memory locations

L1 and L2 cache granularity: 32 bytes 128 byte



Misaligned Memory Access

accessed by threads are not consecutive or not aligned to memory transaction boundaries



Measuring Memory Bandwidth

Key strategies for maximizing GPU performance

```
__global__ void matrixRowAccess(float* matrix, float* output,
                                int width, int height) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < height && col < width) {
        // GOOD: Row-major access (coalesced)
        int idx = row * width + col;
        output[idx] = matrix[idx];
    }
}

__global__ void matrixColAccess(float* matrix, float* output,
                                int width, int height) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < height && col < width) {
        // BAD: Column-major access (uncoalesced)
        int idx = col * height + row;
        output[row * width + col] = matrix[idx];
    }
```

- Ensure coalesced memory access patterns
- Use appropriate data types and alignment
- Minimize memory transactions
- Utilize shared memory for data reuse
- Consider memory access patterns in algorithm design

Occupancy Optimization

Key strategies for maximizing GPU performance

Occupancy is the ratio of active warps to maximum possible warps per SM.

Theoretical Occupancy	100%
Good Occupancy	75%+
Poor Occupancy	<50%

```
// Check occupancy
cudaOccupancyMaxPotentialBlockSize(&minGridSize, &blockSize,
                                   my_kernel, 0, 0);

// Launch with optimal block size
my_kernel<<<gridSize, blockSize>>>(args);
```

Best Practices & Conclusion

Key takeaways for successful CUDA development

Performance Guidelines

✅ Memory Optimization

- Use coalesced memory access patterns
- Leverage shared memory for data reuse
- Consider unified memory for prototyping
- Minimize host-device transfers

✅ Execution Configuration

- Aim for high occupancy (75%+)
- Use multiples of warp size (32)
- Balance threads per block vs blocks
- Consider register usage limits

Development Workflow

🔄 Iterative Approach

- Start with simple, correct implementation
- Profile to identify bottlenecks
- Optimize incrementally
- Validate correctness at each step

🛠️ Tools & Testing

- Use profiling tools (Nsight)
- Implement comprehensive error checking
- Test with various input sizes
- Compare against CPU reference

Take away message

- **Data movement is a primary bottleneck** in optimizing CUDA applications.
- **Unified Memory (UVM)** simplifies development but may degrade performance due to implicit data transfers.
- **Be aware of intermediate buffering** by the runtime, which can introduce additional copying overhead.
- **Use pinned memory or GPU Direct** to minimize unnecessary data movement and improve transfer efficiency.
- **Overlap data transfer with computation** using asynchronous operations to maximize GPU utilization and throughput.

Recommended Resources

CUDA Programming Guide CUDA API Reference Manual PTX Instruction Set Architecture	Reading
CUDA Compiler Driver NVCC	Building executables
CUDA-MEMCHECK Nsight Documentation Kernel Profiling Guide	Debugging & profiling
NVIDIA Developer Forums	Clarifications, explanations, intricate details