CUDA Deep Dive: From Fundamentals to Advanced Techniques

Nitin Shukla

HPC Application Engineer

October 27th 2024

Co-funded by the European Union

EPICURE CINECA

Contents: topics explored

1

Why heterogeneous computing?

Grasping the basic elements of GPU programming

2

CUDA programming mode

Kernel launch, Thread and Memory hierarchy

3

Performance consideration

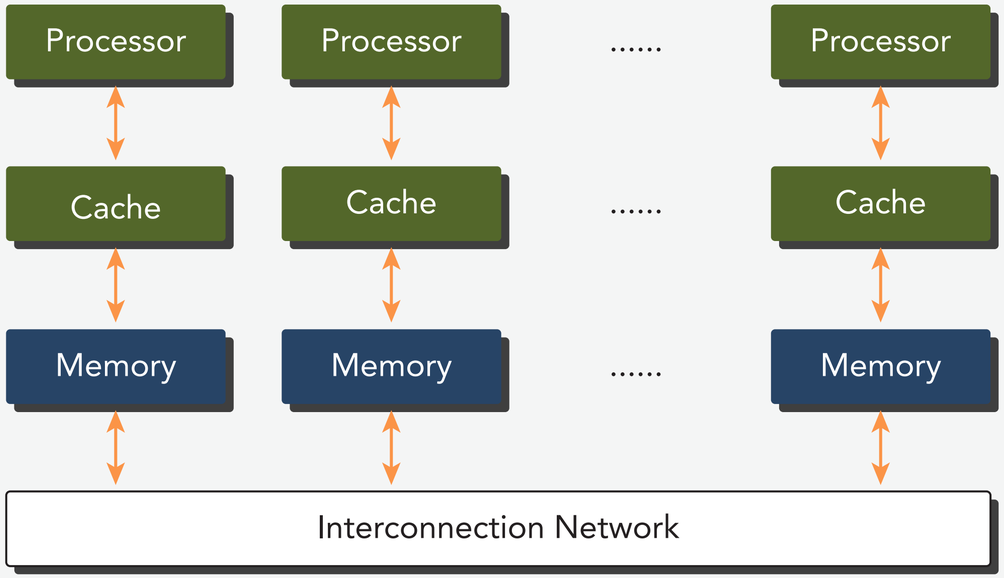
Memory management, analysis Nsight and Nvidia

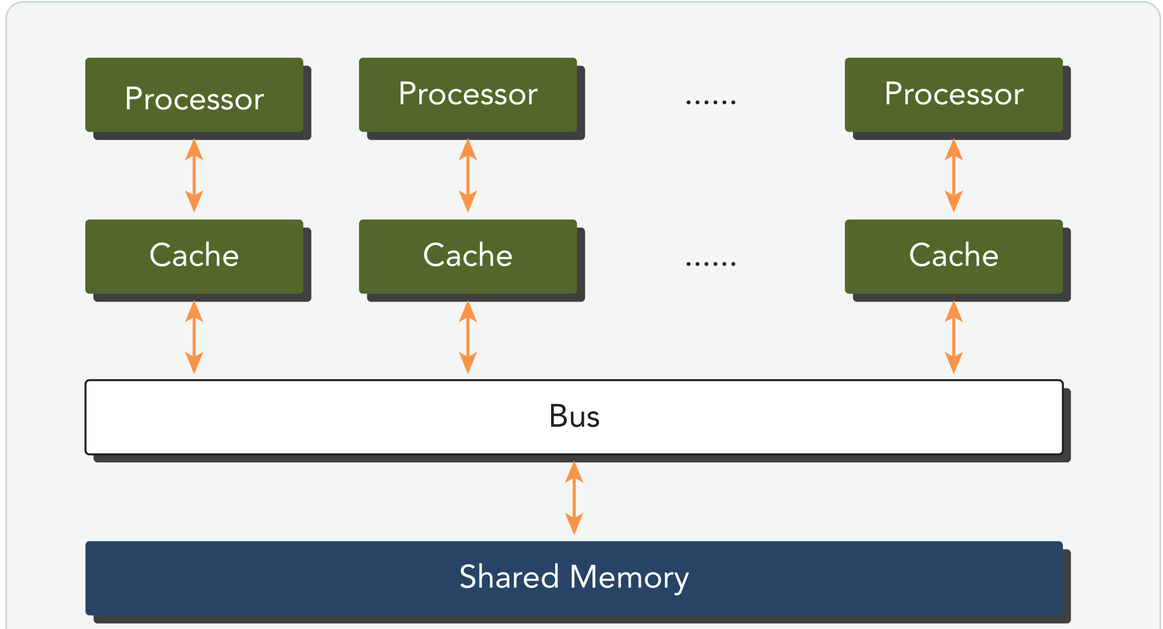
4

Streams and Concurrency

Overlapping kernel execution & data transfer on Single/Multi GPU

Computer architecture drives parallelism at the core level





Most modern processors implement

Memory (instruction memory and data memory)
Central processing unit (control unit and arithmetic logic unit)
Input/Output interfaces

Parallel computing two core technologies

Computer architecture i.e Hardware aspect
Parallel programming i.e Software aspect

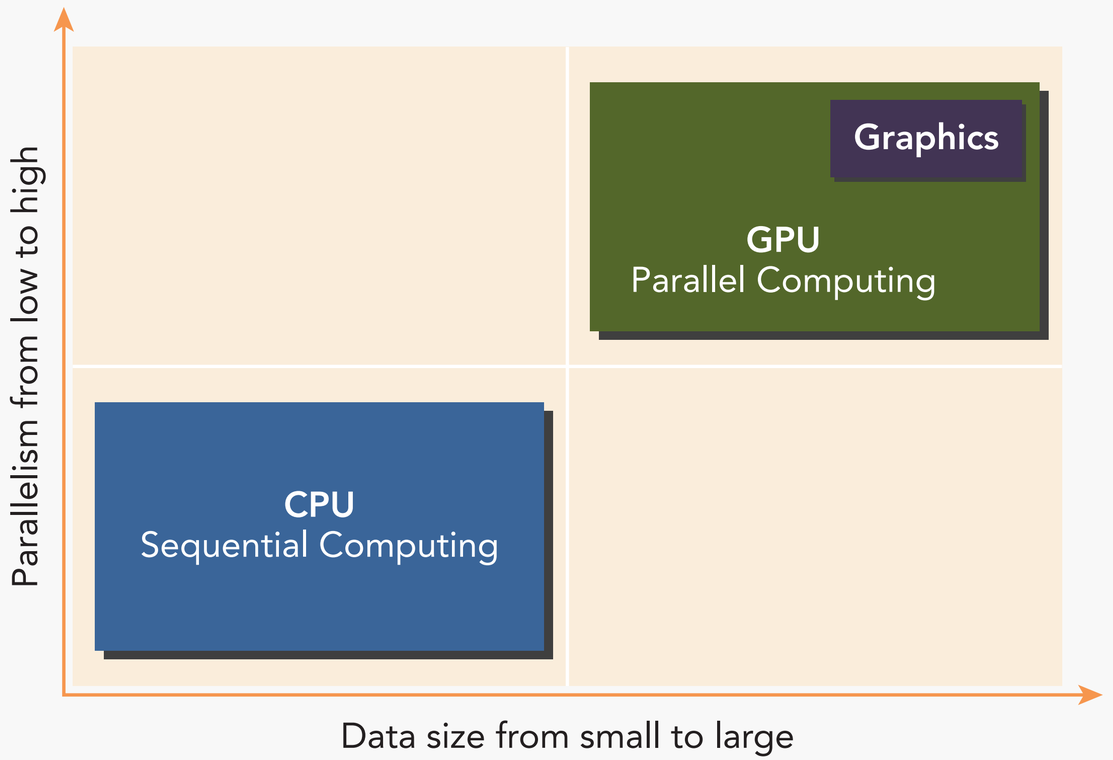
Computer architecture drives parallelism at the core level

Fundamentals types of parallelism

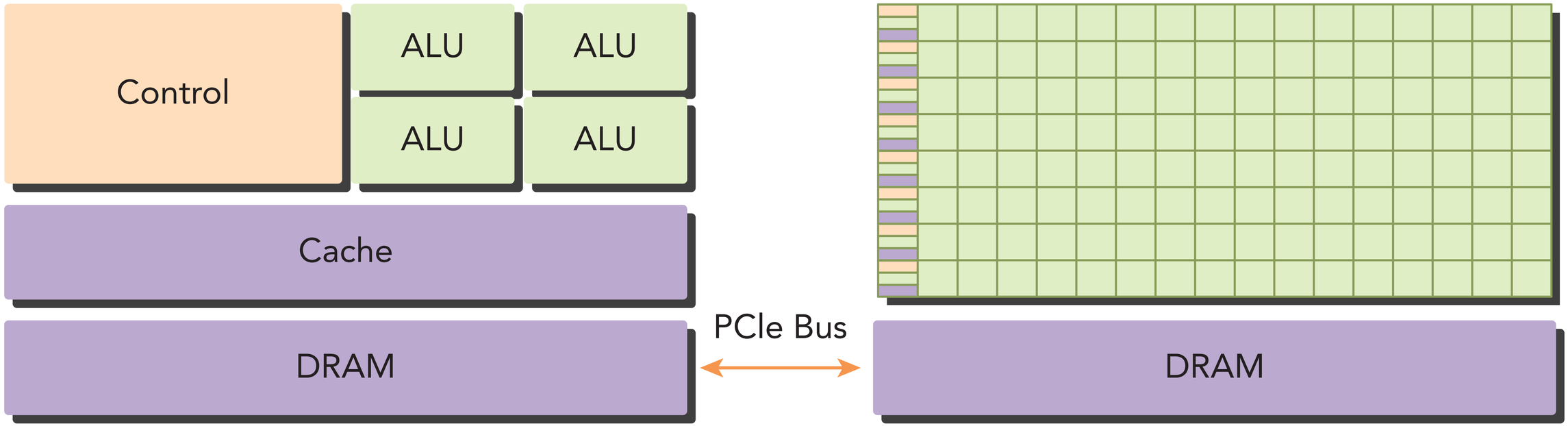
- Task parallelisms: multiple independent tasks can run simultaneously, distributing functions across multiple cores
- Data parallelisms: multiple data items can be processed simultaneously, distributing the data across multiple cores

Heterogeneous computing

- CUDA programming: well-suited to address problems that can be expressed as data-parallel computations



How GPUs are different than CPUs?



CPU (host): minimize latency

GPU(Device): maximize throughput

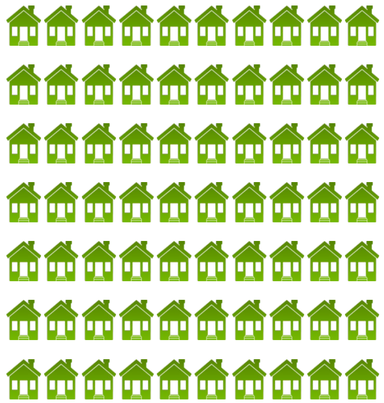
Why computing perf/Watt matters?

Traditional CPUs are not economically feasible

2.3 PFlops

7.0 Megawatts

7000 homes

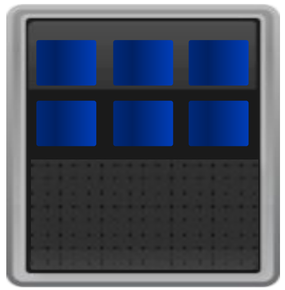


7.0 Megawatts

GPU-accelerated computing started a new era

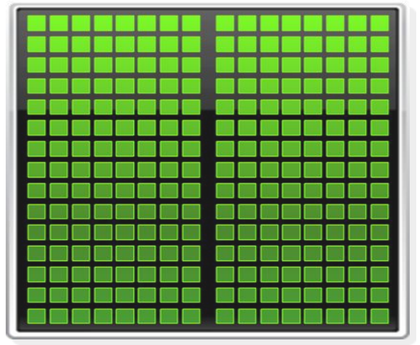
CPU

Optimized for Serial Tasks



GPU Accelerator

Optimized for Many Parallel Tasks

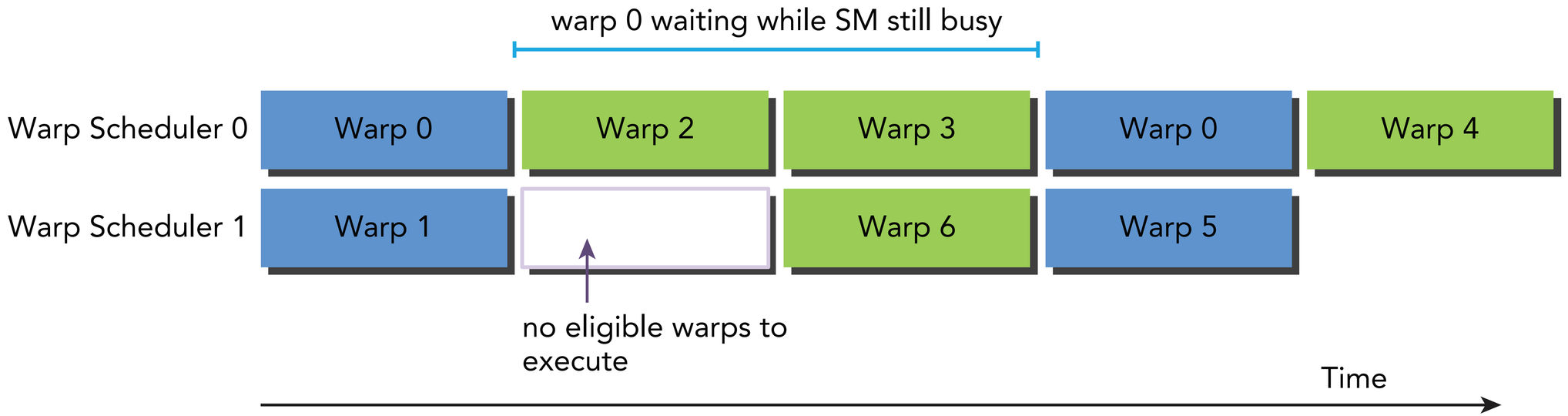


GPU architecture

GPU architecture is built around a scalable array of SM

- CUDA cores
- Shared Memory/L1 Cache
- Register File
- Load/Store Units
- Special Function Units
- Warp Scheduler

Latency Hiding



GPU acceleration for data-parallel tasks

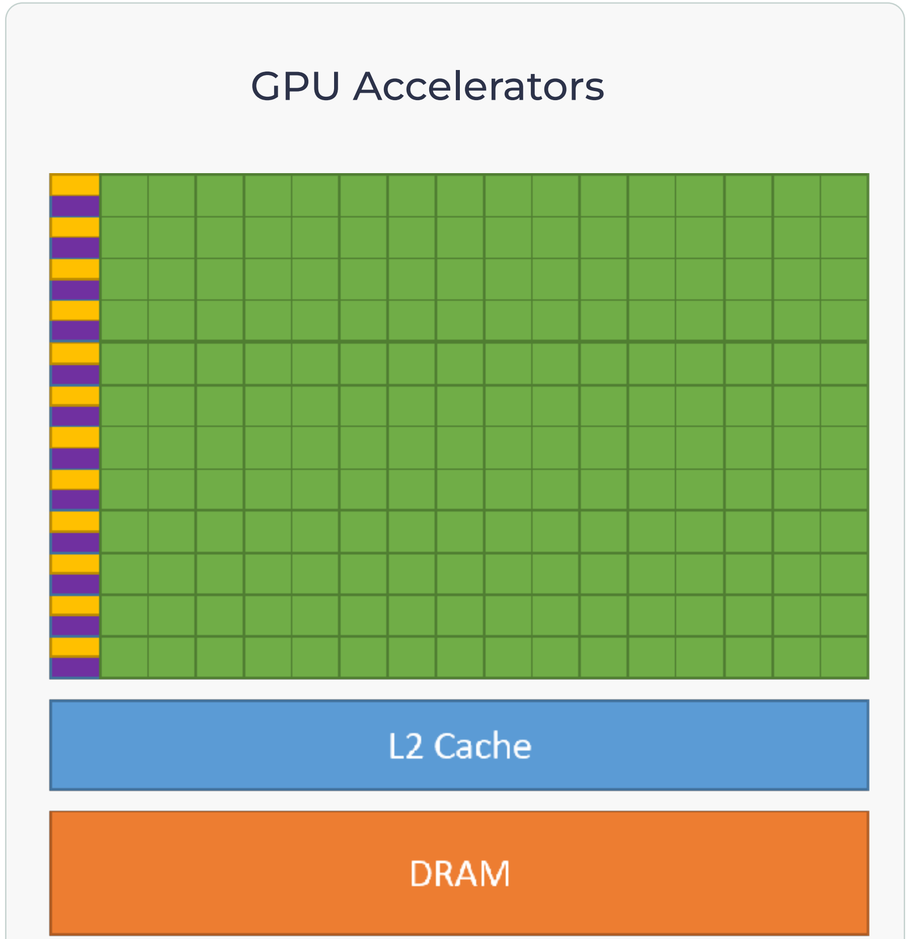
Two important features that describe GPU capability

- Number of CUDA cores
- Memory size

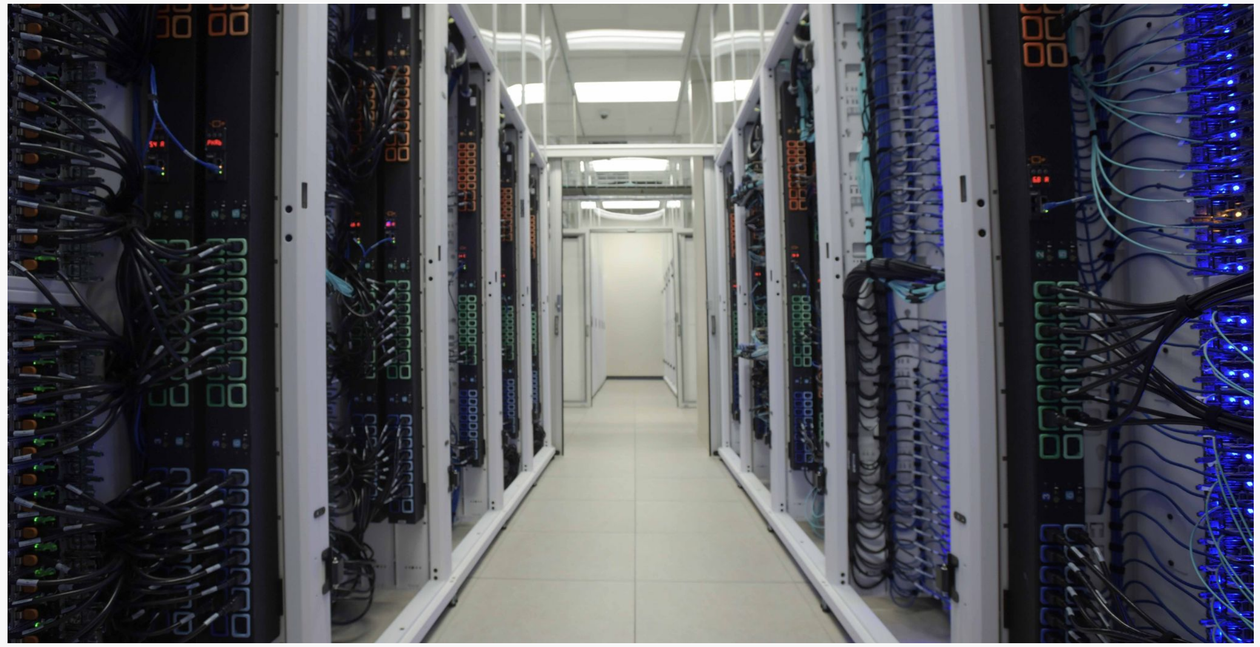
GPU Performance Metrics: Throughput vs. Latency

- Peak computational performance
  measures in Tflops or Pflops, reflects a device's ability to perform floating-point calculations rapidly and efficiently

- Memory bandwidth
  the rate at which data can be transferred between the CPU and memory, measured in gigabytes per second (GB/s). It directly impacts the speed of data-intensive applications.



NVIDIA Tesla A100 with 54 Billion Transistor



- With 7nm technologies
- 19.5 teraflops of FP32 performance
- 6912 CUDA cores, 40GB of graphics memory, and 1.6TB/s of graphics memory bandwidth
- The A100 80GB model announced in Nov 2020, has 2.0TB/s graphics memory bandwidth

TOP10 System – November 2023

1. Frontier ORNL

AMD CPUs
AMD GPUs
HPE Slingshot
1679 pflops

2. Aurora ANL

Intel CPUs
Intel GPUs
HPE Slingshot
1059 pflops

3. Eagle Microsoft

Intel CPUs
Nvidia GPUs
Nvidia Inf
846 pflops

4. Fugaku RIKEN

Fujitsu ARM

Fujitsu Tofu
537 pflops

5. Lumi CSC

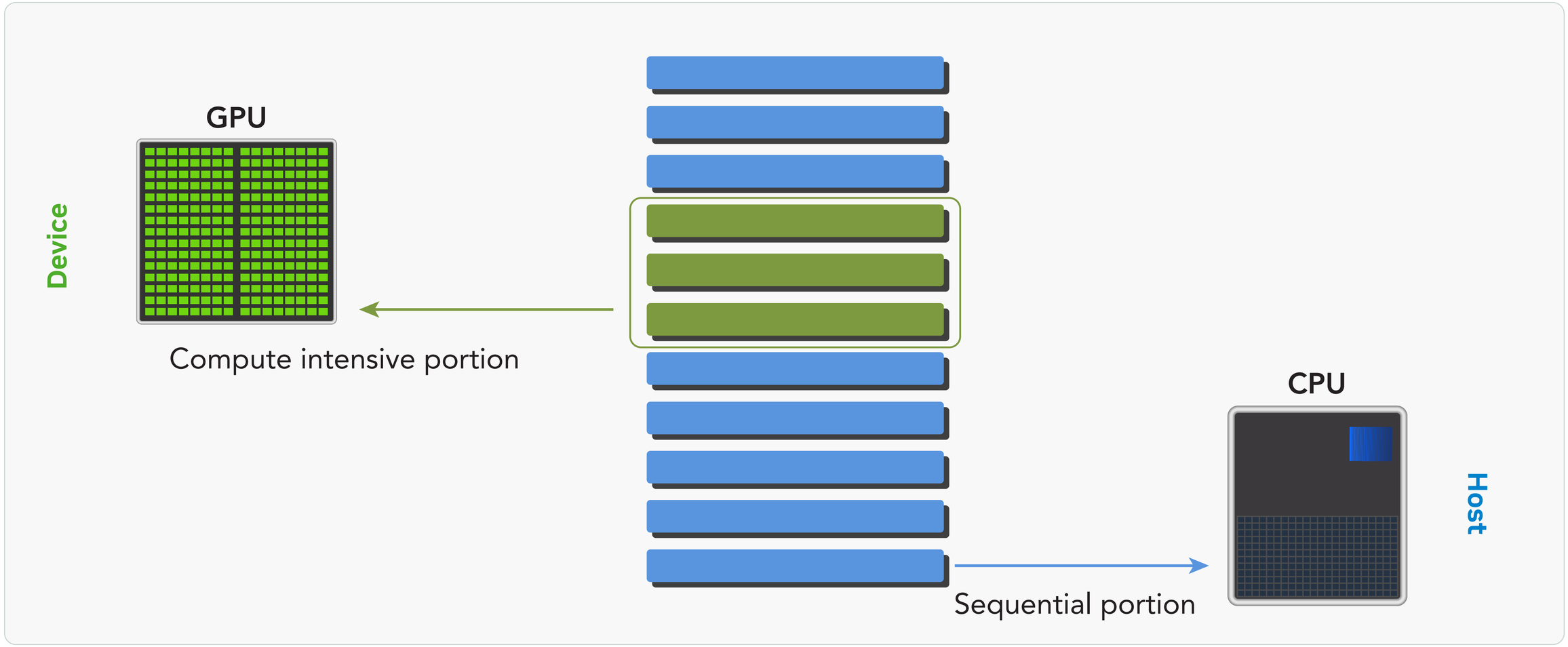
AMD CPUs
AMD GPUs
HPE Slingshot
531 pflops

6. Leonardo CINECA

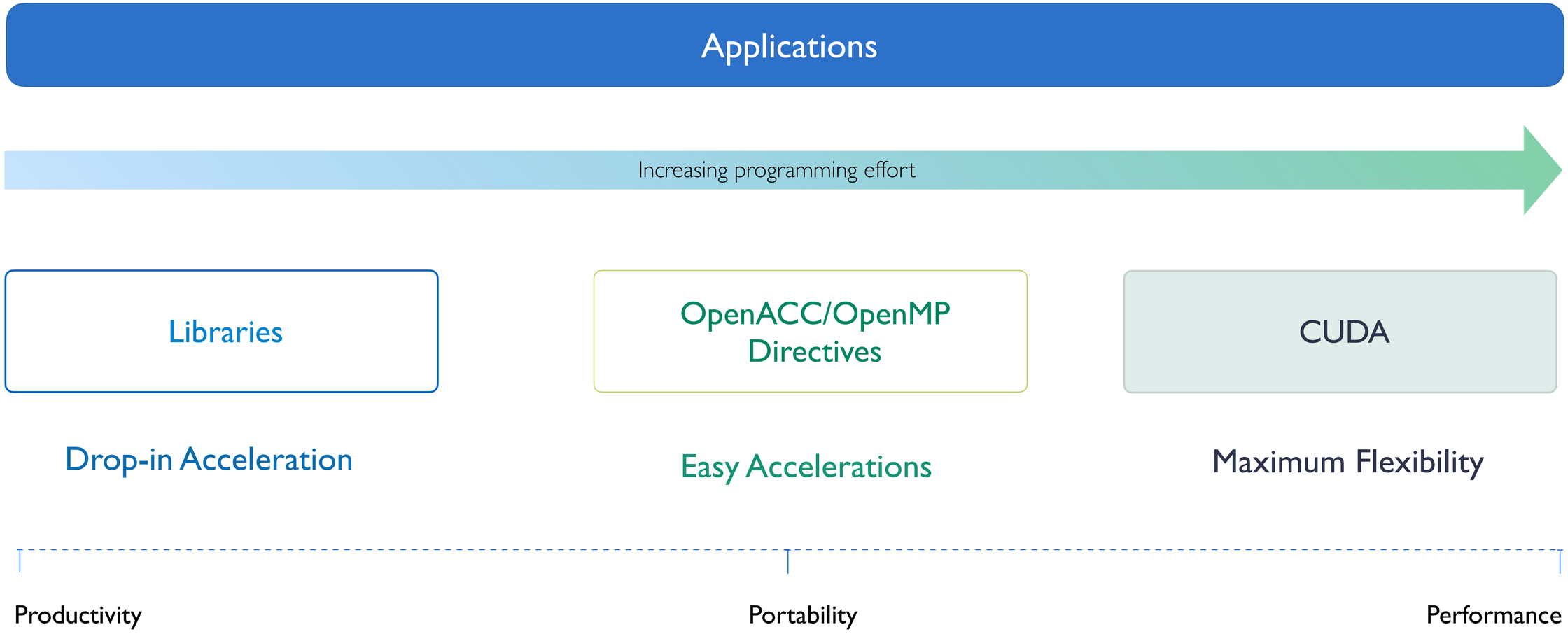
Intel CPUs
Nvidia GPUs
Nvidia Inf
304 pflops

70 % of FLOP/s by GPUs, > 100 000 GPUs in Frontier+Aurora

GPUs serve as a co-processor, not a standalone platform



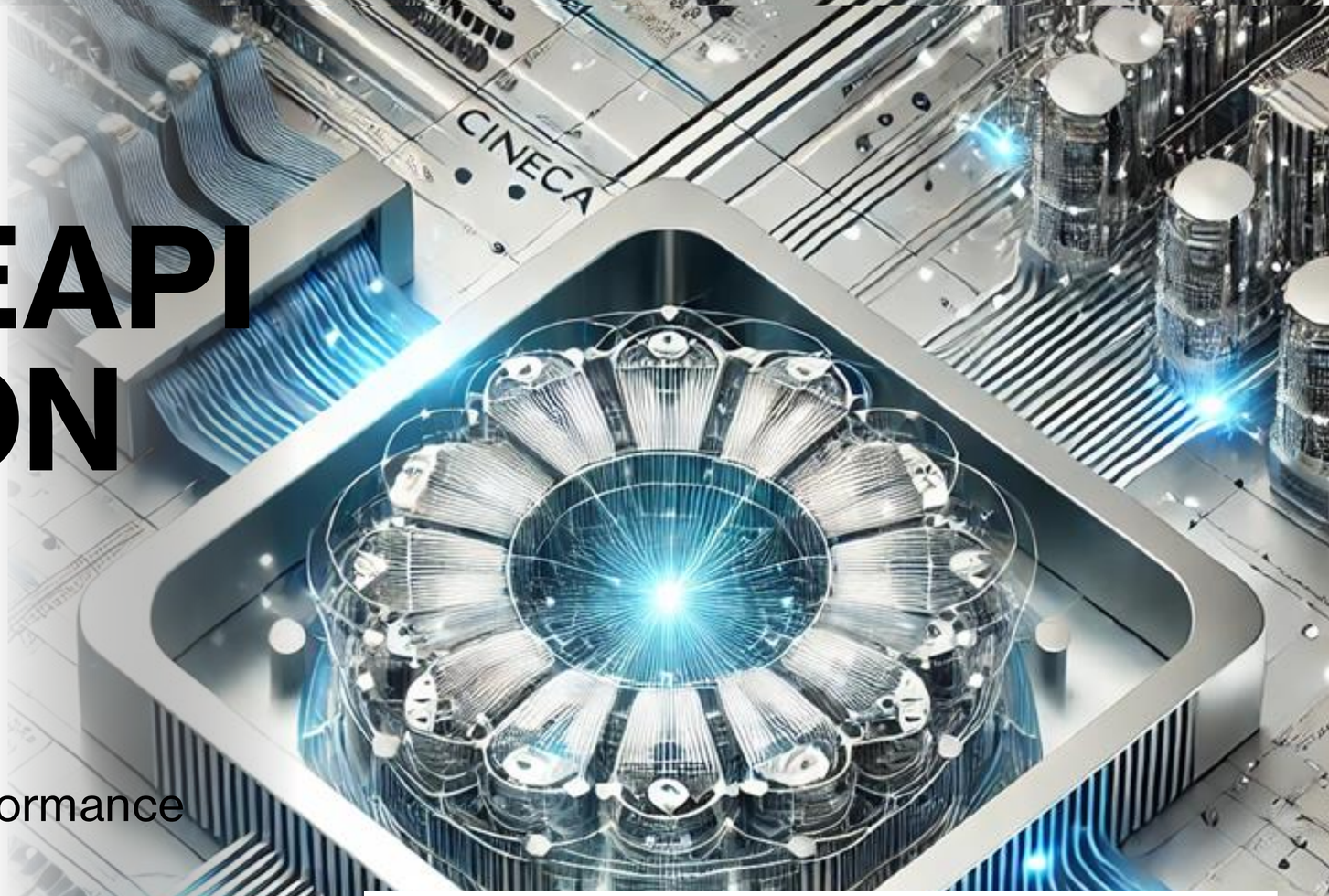
Ways to parallels an applications on Nvidia GPUs



SYCL / ONEAPI HACKATHON @ CINECA

Empowering the Future of High-Performance Computing with SYCL

CINECA intel



Register now!

Follow the link: https://hpc-portal.eu/node/2190

... or scan the QR code

For further info / questions: a.masini@cineca.it

Why CUDA?

Performance

- Massive Parallelism: scale to 1000’s of cores, 10000000’s of parallel thread
- Massive Gain: substantial performance improvements in tasks that can be divided into smaller, concurrent operations

Scalability

- Efficiently maps to the GPU architecture: well-suited for leveraging GPU capabilities
- Wide Range of Hardware: applications can scale from small embedded devices to large supercomputers

Flexibility

- Programming Languages: supports various programming languages
- Easy to use: let programmers strip away complexity associated with parallel computing and focus on parallel algorithms

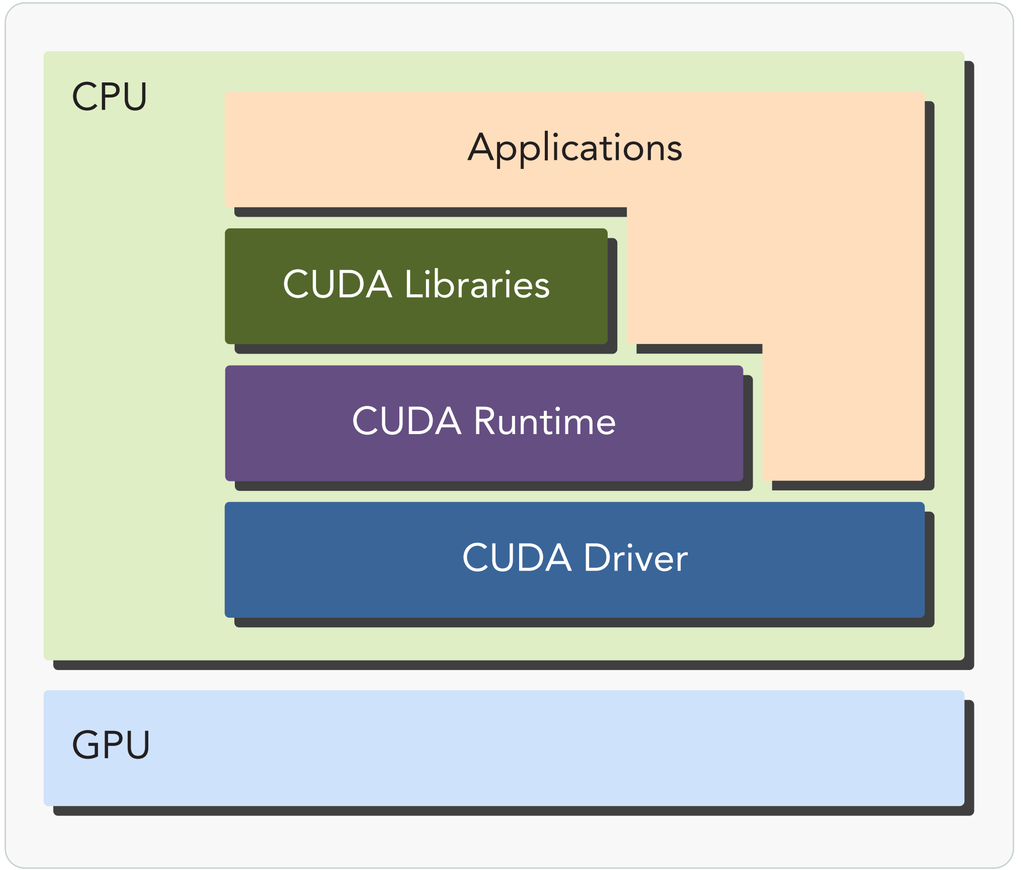
What is CUDA?

CUDA : Compute Unified Device Architecture

- Enable heterogeneous systems (i.e., CPU+GPU)
- A new architecture instruction set called PTX (Parallel Thread eXecution) to match GPU typical hardware
- Parallelism allows developers to use GPUs for general purpose processing (GPGPU)

The SDK includes

- A Drivers, runtimes and API
- Compiler wrappers for complain coda code (nvcc)
- Libraries (cuBLAS, cuFFT, cuSolver) debuggers (cuda-gdb, cuda-memcheck), profilers (nvprof, nView), etc
- CUDA-aware languages C/C++, Fortran, PyCUDA, CUDA.Jl



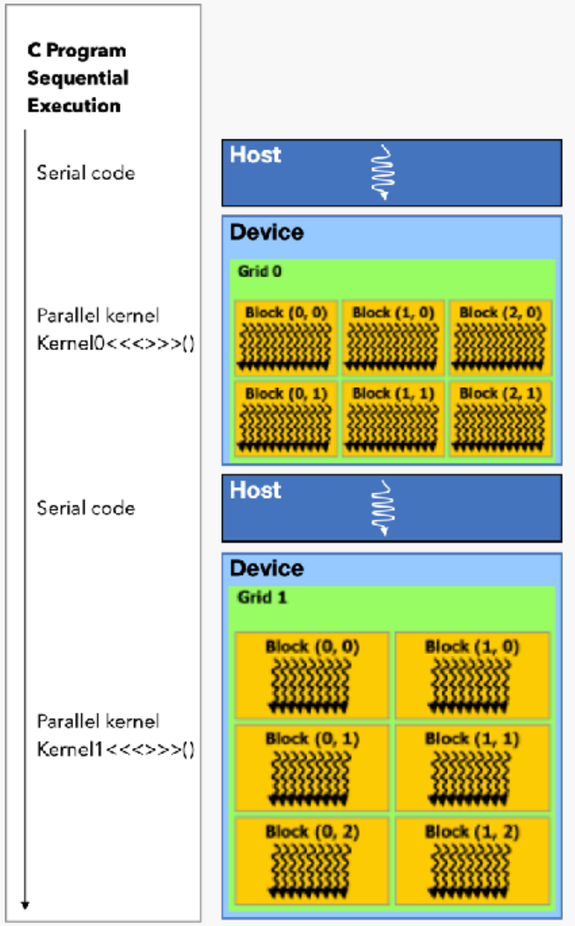
CUDA execution model

CUDA programmer perspective

- **Heterogenous computing:** combination of CPU and GPU
  - **Host:** The CPU and its memory
  - **Device:** The GPU and its memory
- **Execution:** Programs run a on the host and launch parallel code (kernels) on the device by many threads

Programming model view

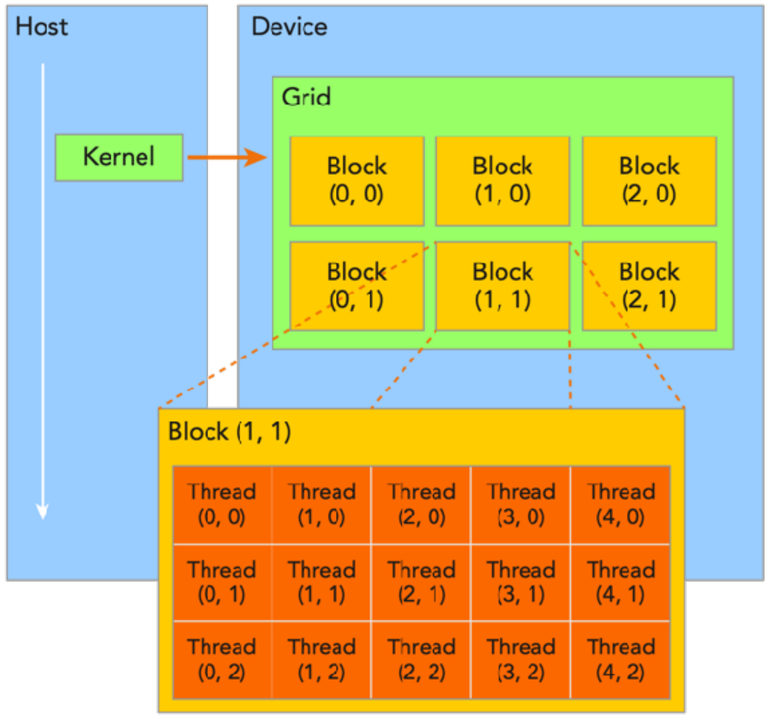
- **Kernels:** A function written in CUDA C/C++ and executed on the GPU
- **Launch configurations:**
  - **Threads:** Smallest unit of execution in CUDA
  - **Block:** A collection of threads
  - **Grid:** A collection of blocks
- **Memory management:** Allocate and transfer data between host (CPU) and device (GPU)



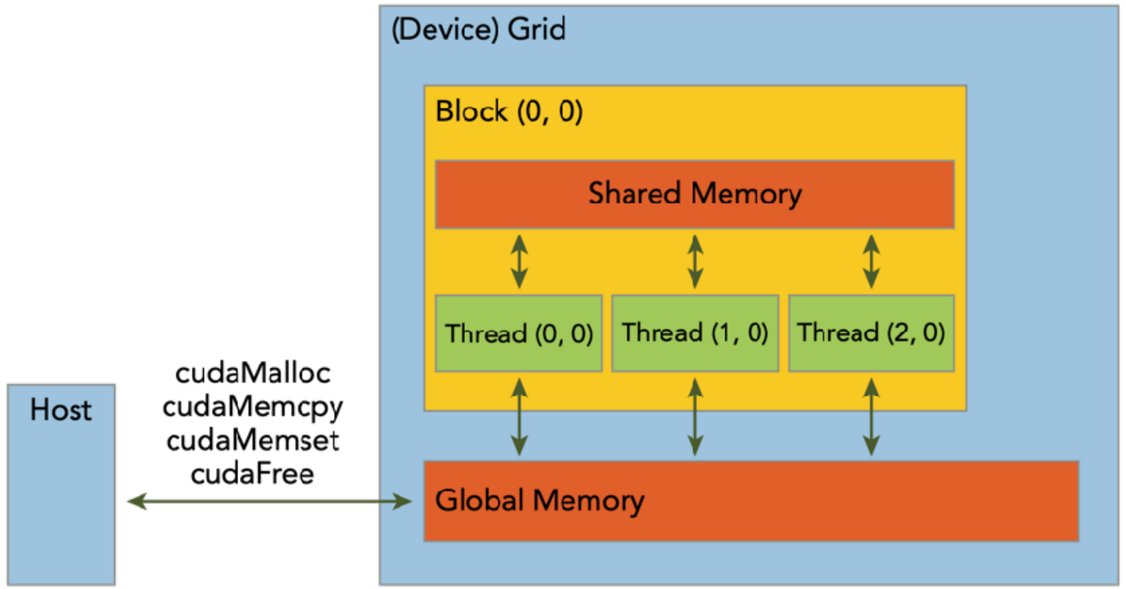
Compiling and running CUDA enable application

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

Thread hierarchy structure



Memory hierarchy structure



Embarrassing parallel code

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.

// CPU function

```
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);
      ..
  }
```

Declaring Host-Called, Device-Executed Functions

CUDA differentiates between these functions by using one of the following function type qualifiers as a prefix

- __global__ qualifier for kernels that can be invoked globally
- __host__ functions called from host and executed on the host
- __device__ functions called from device and execute on the device (a function that is called from a kernel needs the __device__ qualifier)

Step to Launching a CUDA Kernel

__global__ void()

Defines a kernel
can be invoked globally either from CPU or GPU

Execution configuration

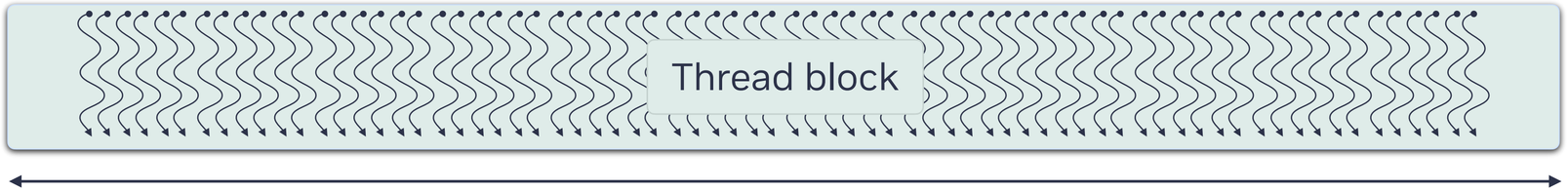
Kernel_name <<<numBlocks, numThreads>>> (arguments);
Specifies grid and block dimensions

Synchronization

Launching kernel is asynchronous
cudaDeviceSynchronize(): wait until device code completeness

```
// Kernel
__global__
 sumArraysOnDevice(float *A, float *B, float *C, const int N)
{
    int idx = threadIdx.x + (blockIdx.x * blockDim.x)
    if(idx<N)
      C[idx] = A[idx] + B[idx];
}

int main(int argc, char **argv)
{
  ..
  start = cpuSecond();
  sumArraysOnGPU<<<gridSize, blockSize>>>(d_A, d_B, d_C, N);
  cudaDeviceSynchronize();
  double gpuTime = cpuSecond() - start;
  printf("GPU Execution Time: %f seconds\n", gpuTime);
  ..
}
```

CUDA launches arrays of parallel threads



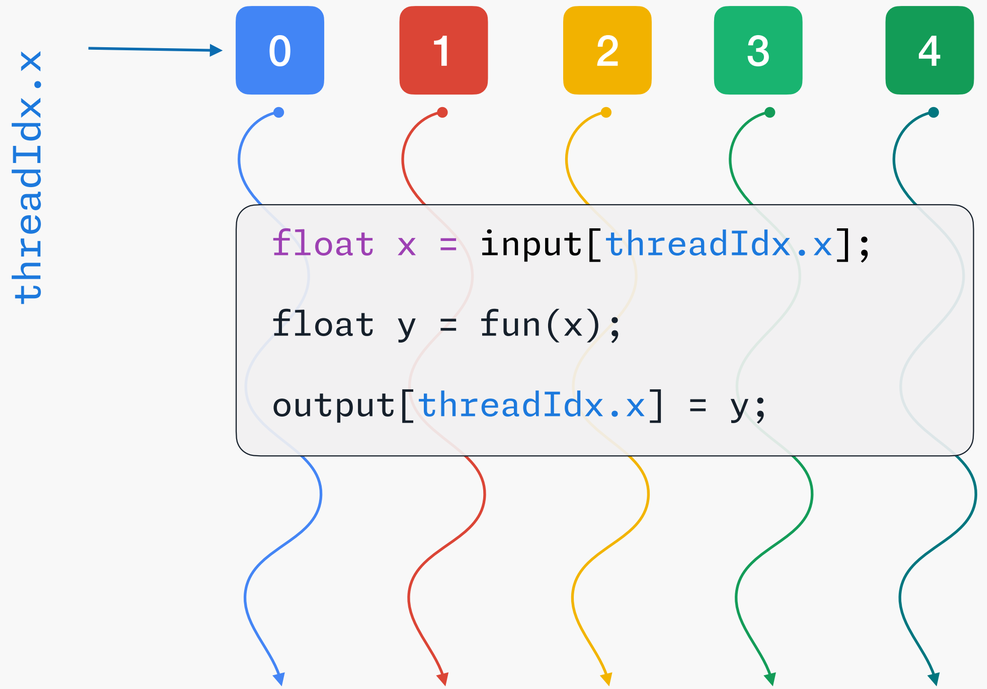
A block has a fixed number of threads which are guaranteed to be running simultaneously on the same SM

CUDA launches arrays of parallel threads

For fully utilisation of the parallel processing power of the GPU

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
- Threads are similar to data-parallel tasks.
- Threads independently execute the same operation on a data subset
- Follows SPMD model i.e the Single Program Multiple Data => SIMT Single Instructions Multiple threads



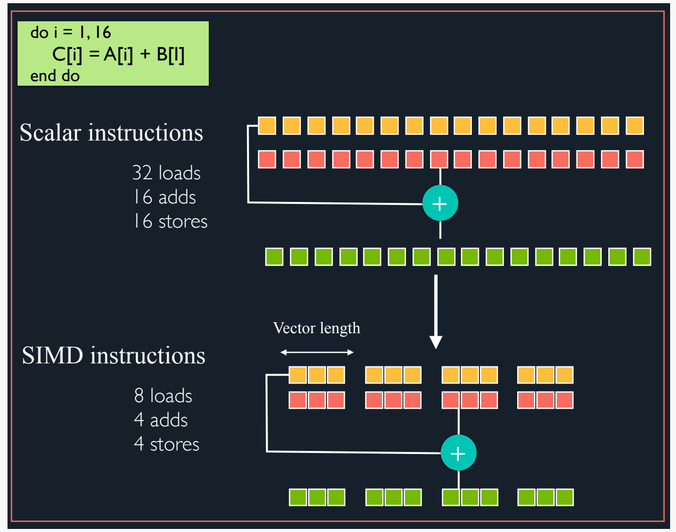
SIMT VS. SIMD execution model

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 describes a class of instructions which perform the same operations on multiple registers simultaneously
- Converting an algorithm to use SIMD is usually called “Vectorizing”
- a SIMD register (or a vector register) can hold many values (2 - 16 values or more) of a single type
- Vectorisation helps you write code which has good access patterns to maximise bandwidth



SIMT VS. SIMD execution model

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

A loose extension of SIMD which is what CUDA's computational model is, although there is key differences

- Single instruction, multiple registers
- Single instructions multiple addresses
  i.e. parallel memory access!
- 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

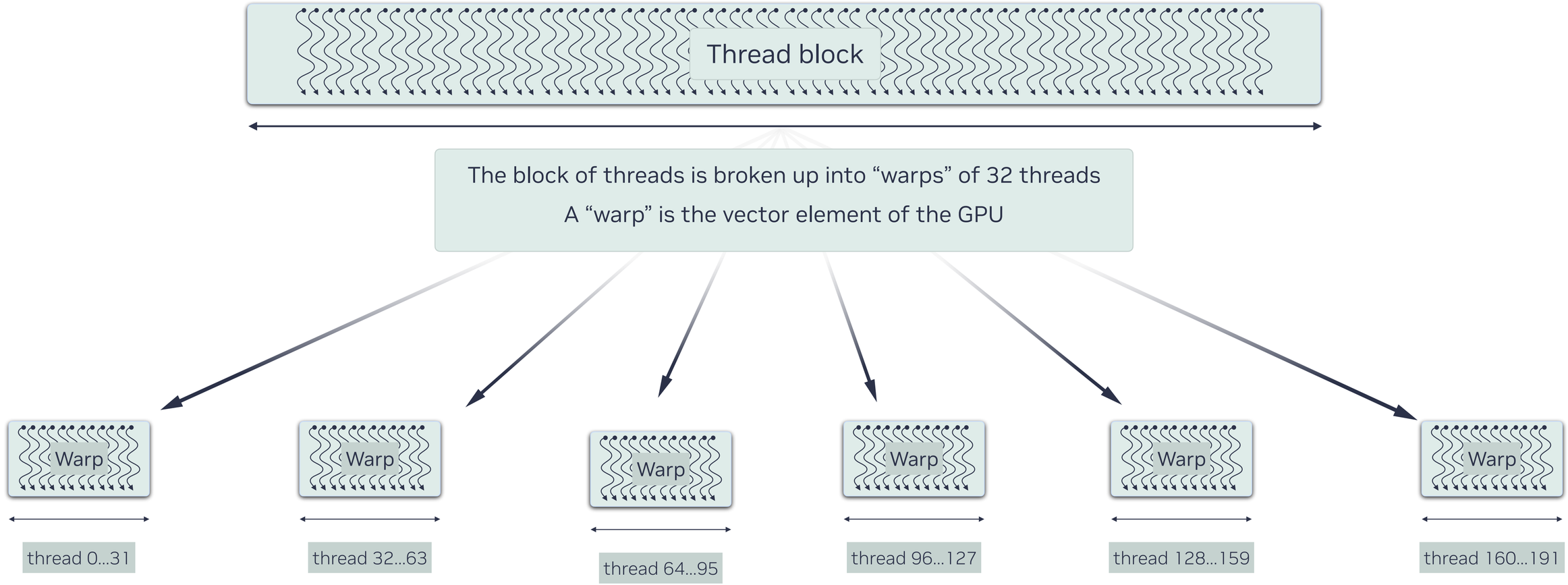
SIMT thread registers			
a[I]	a[I+1]	a[I+2]	a[I+3]
b[I]	a[I+1]	b[I+2]	b[I+3]
a	a	a	a
b	b	b	b
I	I+1	I+2	I+3
…	…	…	…

SIMT VS. SIMD execution model

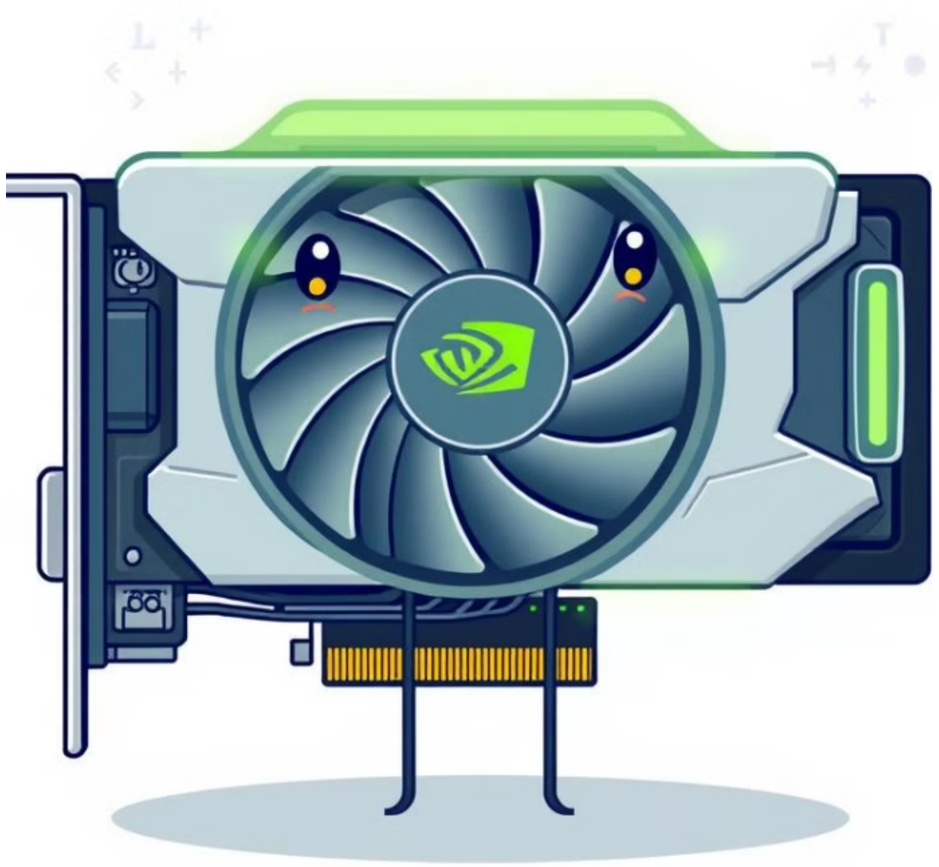
Both SIMD and SIMT achieve parallelism by broadcasting a single instruction to multiple execution units

Feature	SIMD	SIMT
Architecture	Traditional CPUs	Utilized by NVIDIA GPUs
Execution Unit	Multiple data lanes	Multiple threads (warps)
Flexibility	Low	High
Branch Handling	No support for divergence	Supports thread divergence
Best Suited For	Homogeneous data operations	Dynamic control flow applications
Common Usage	CPU computing	Vector processing on GPUs

CUDA launches arrays of parallel threads



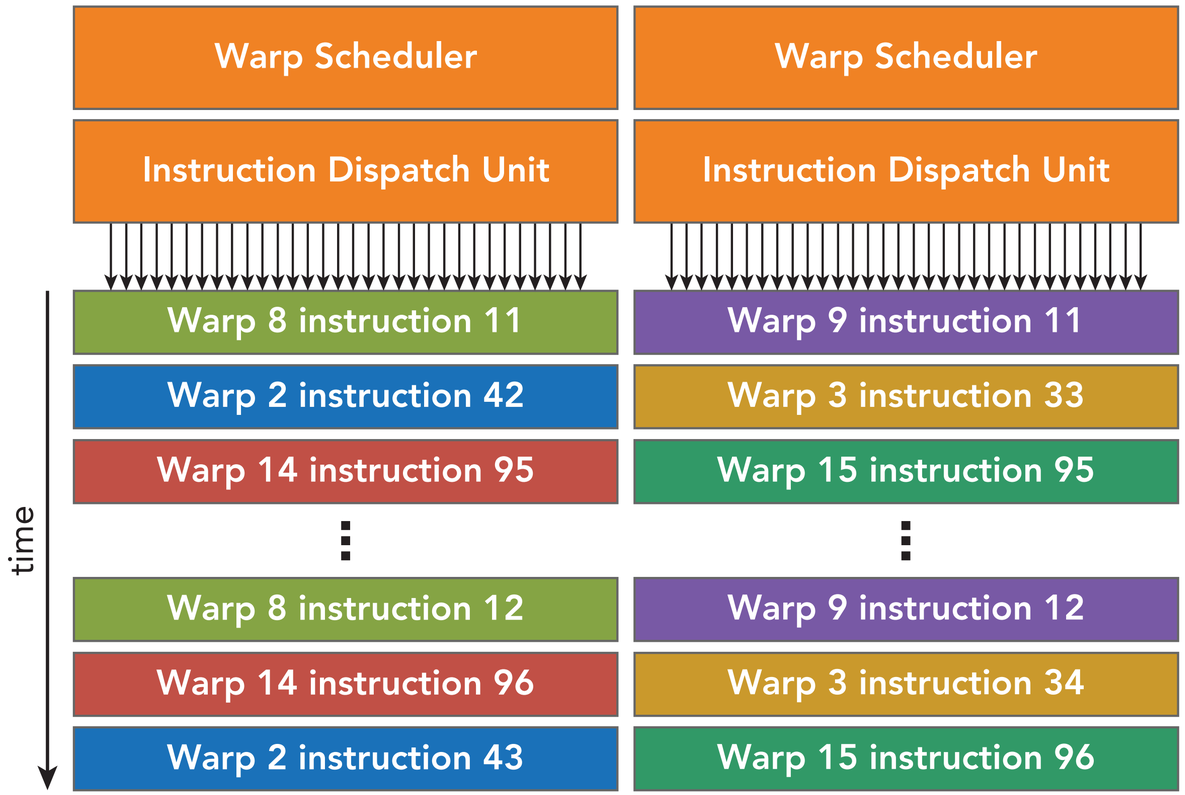
What is warp, and why is it important?



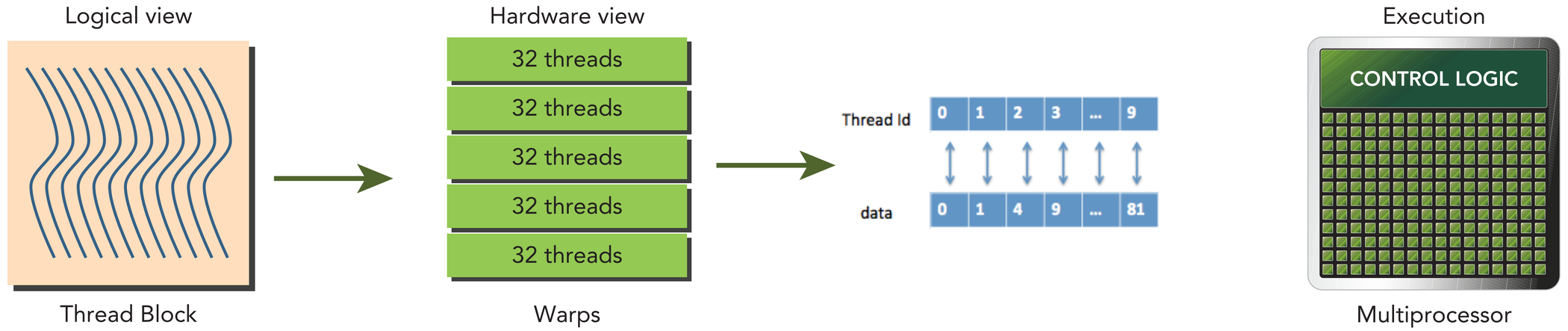
What is WARP?

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



Warps as Scheduling Units



Groups (vectors) of 32 consecutive threads of a block that are executed in parallel in hardware

- An implementation technique, not part of the CUDA programming model
- basic unit of execution in an SM

```
Warp 0: thread  0, thread  1, thread  2, ... thread 31
Warp 1: thread 32, thread 33, thread 34, ... thread 63
Warp 3: thread 64, thread 65, thread 66, ... thread 95
Warp 4: thread 96, thread 97, thread 98, ... thread 127
```

Why do we need to have so many warps in an SM?

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

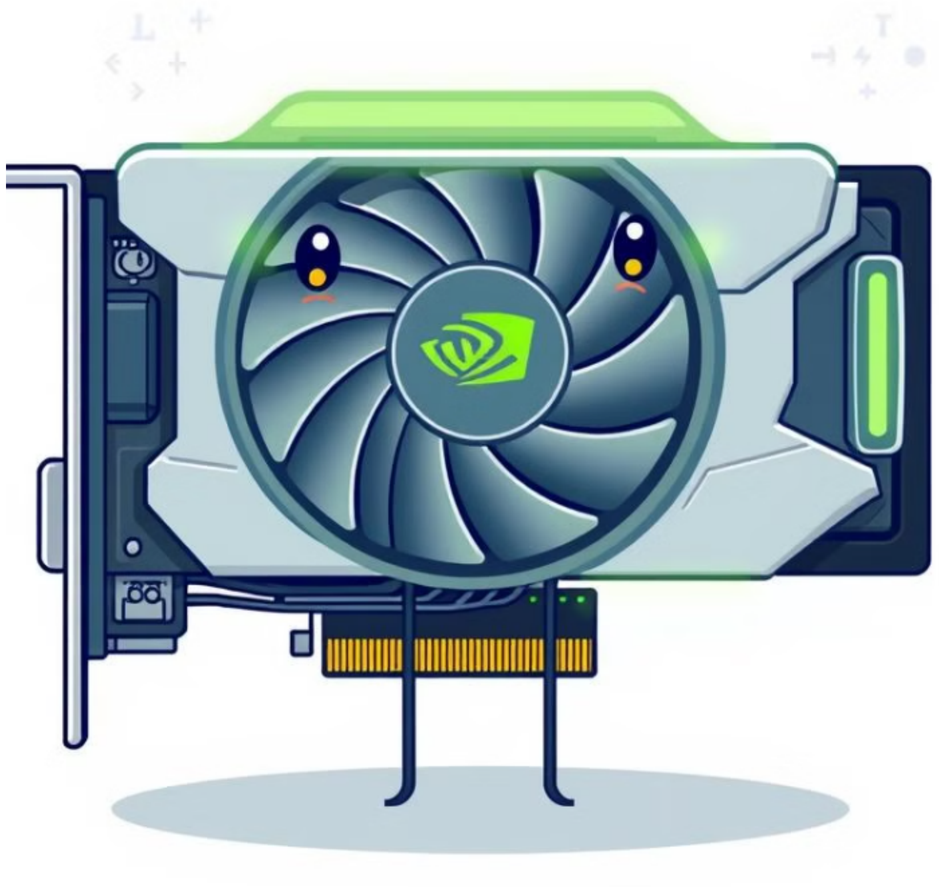
Resource Utilisation

- **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

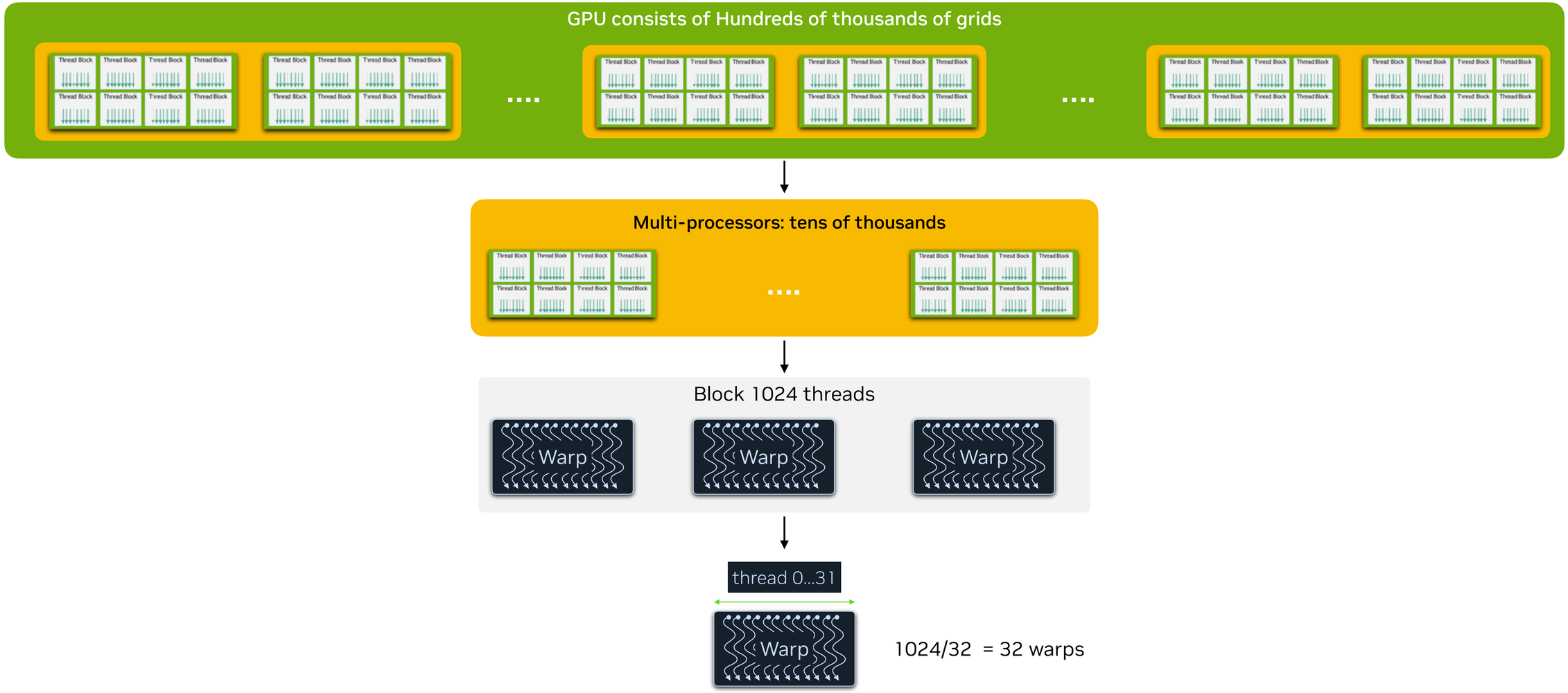
Parallelism

- **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

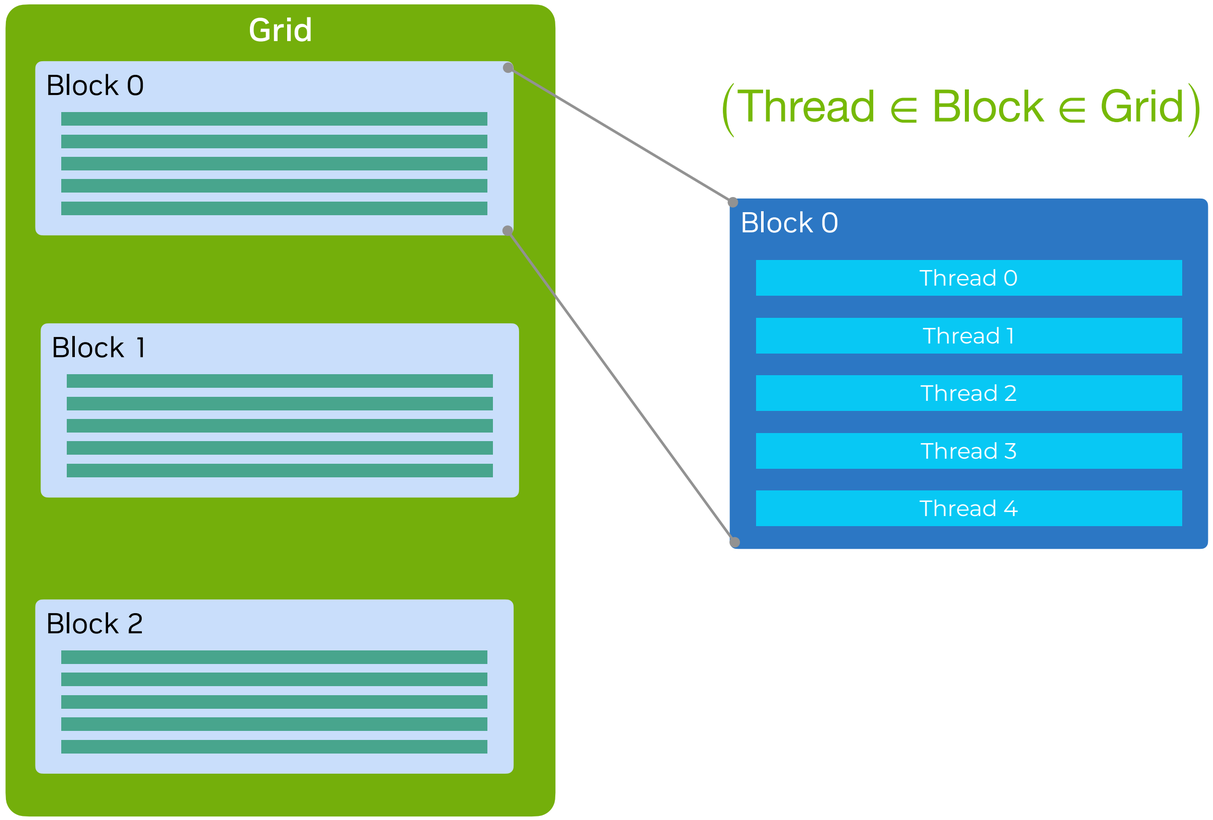
GPU Thread hierarchy



GPU Thread hierarchy



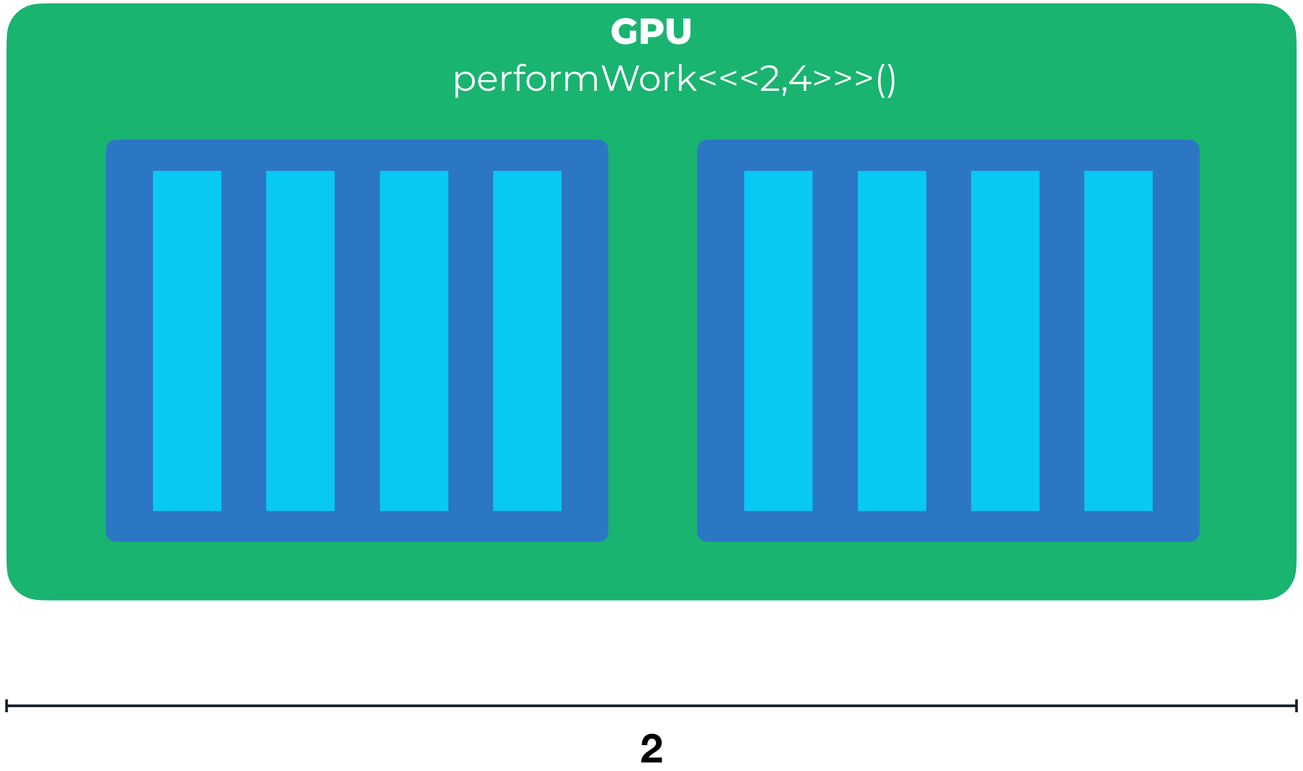
Kernel execution across Thread, Block, and Grid



- 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

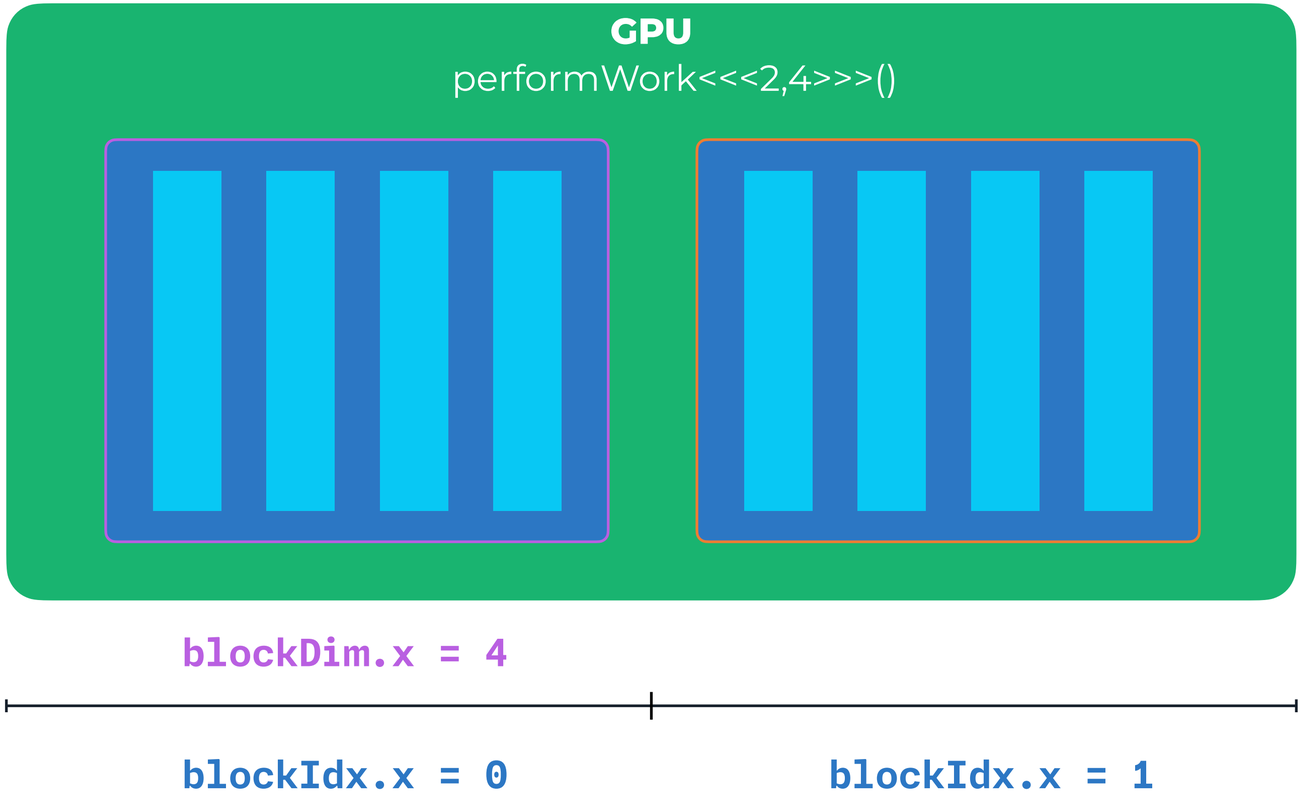
Kernel execution across Thread, Block, and Grid

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



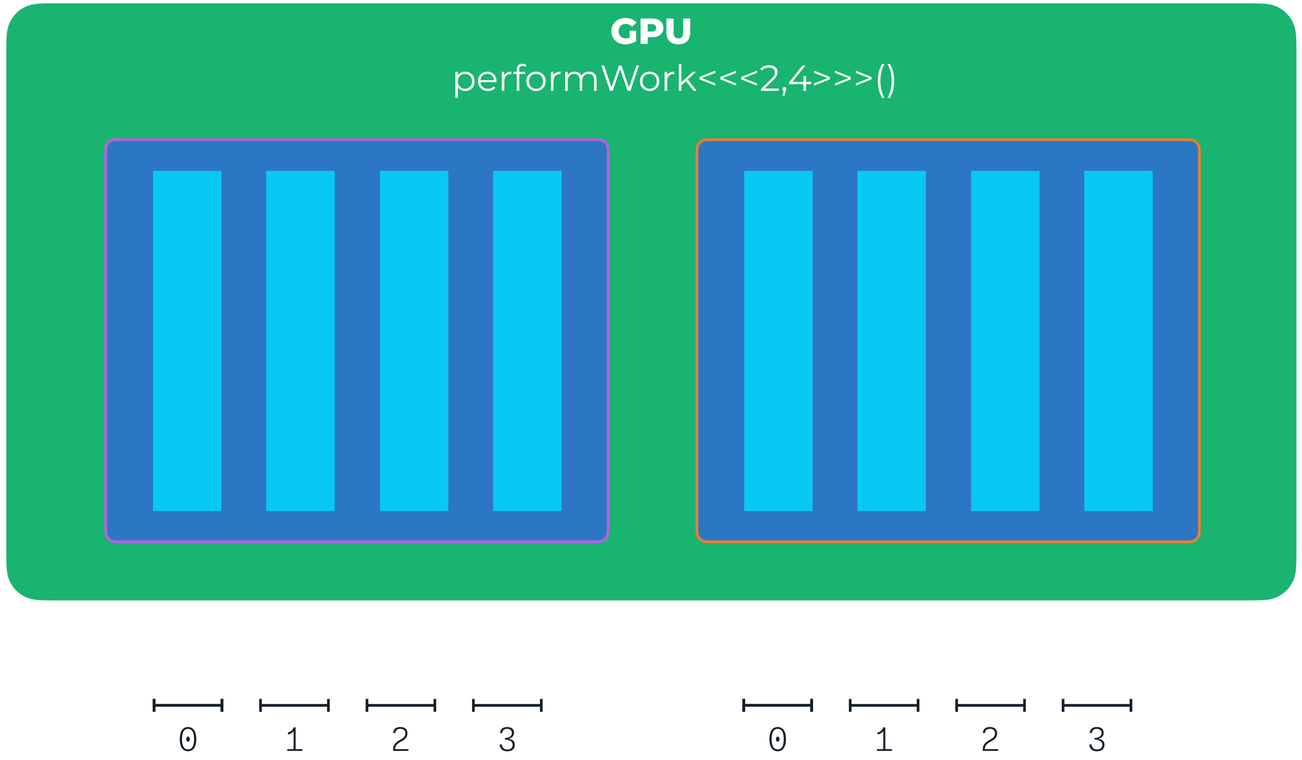
Kernel execution across Thread, Block, and Grid

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



Kernel execution across Thread, Block, and Grid

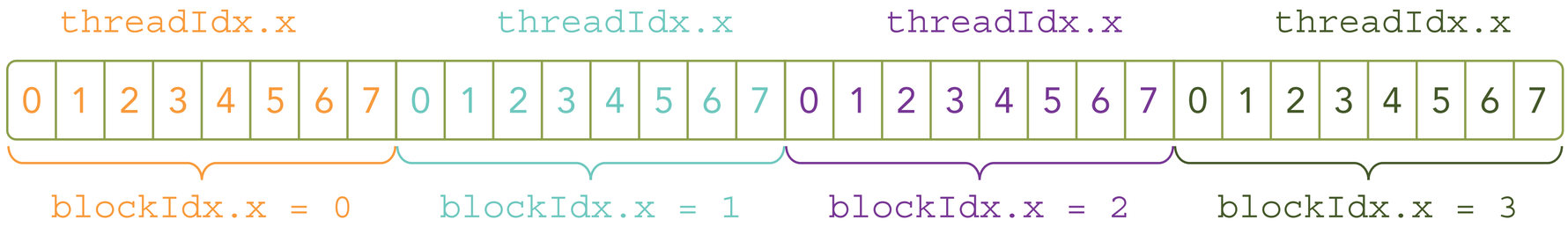
`threadIdx.x:` index of the thread with a block



Kernel execution across Thread, Block, and Grid

Choose the optimal block size

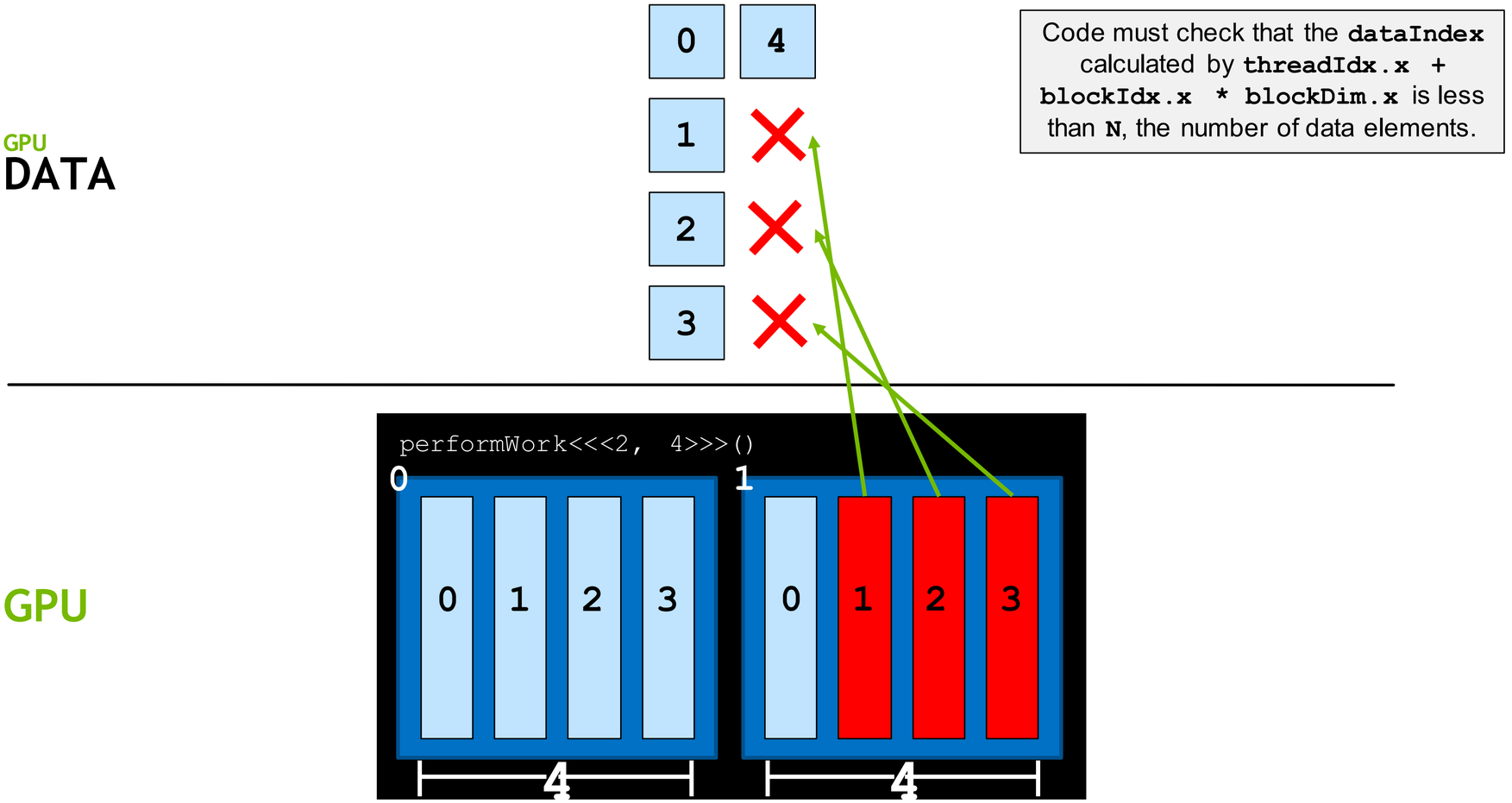
- A **limited number of threads** (**1024**) can fit inside a thread block
- To increase parallelism, we need to **coordinate** work **among thread blocks**.
- This is achieved by **mapping** element of data vector to threads using **global index = threadIdx.x + blockIdx.x*blockDim.x**




```
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 }
```

Grid size larger than data set



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

Choosing the optimal grid size

Choose the optimal block size

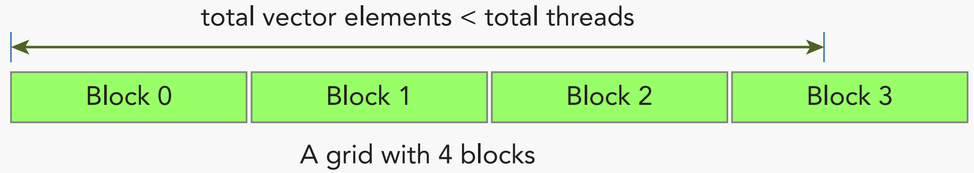
- Write an execution configuration that creates more threads than necessary
- Pass a value as an argument into the kernel (N) that represents that total size if the data set to be processed/total threads needed to complete the work
- Calculate the global index and if it does not exceed N perform the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

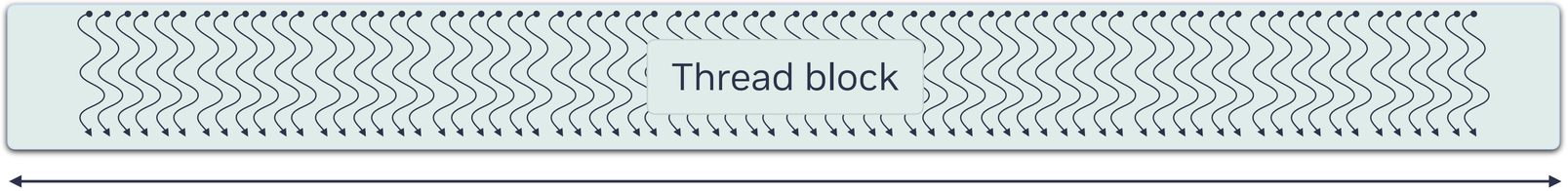
Maximum size at each level of the thread hierarchy is device dependent. On A100 typical you get :

- 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

Every thread runs exactly the same program



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

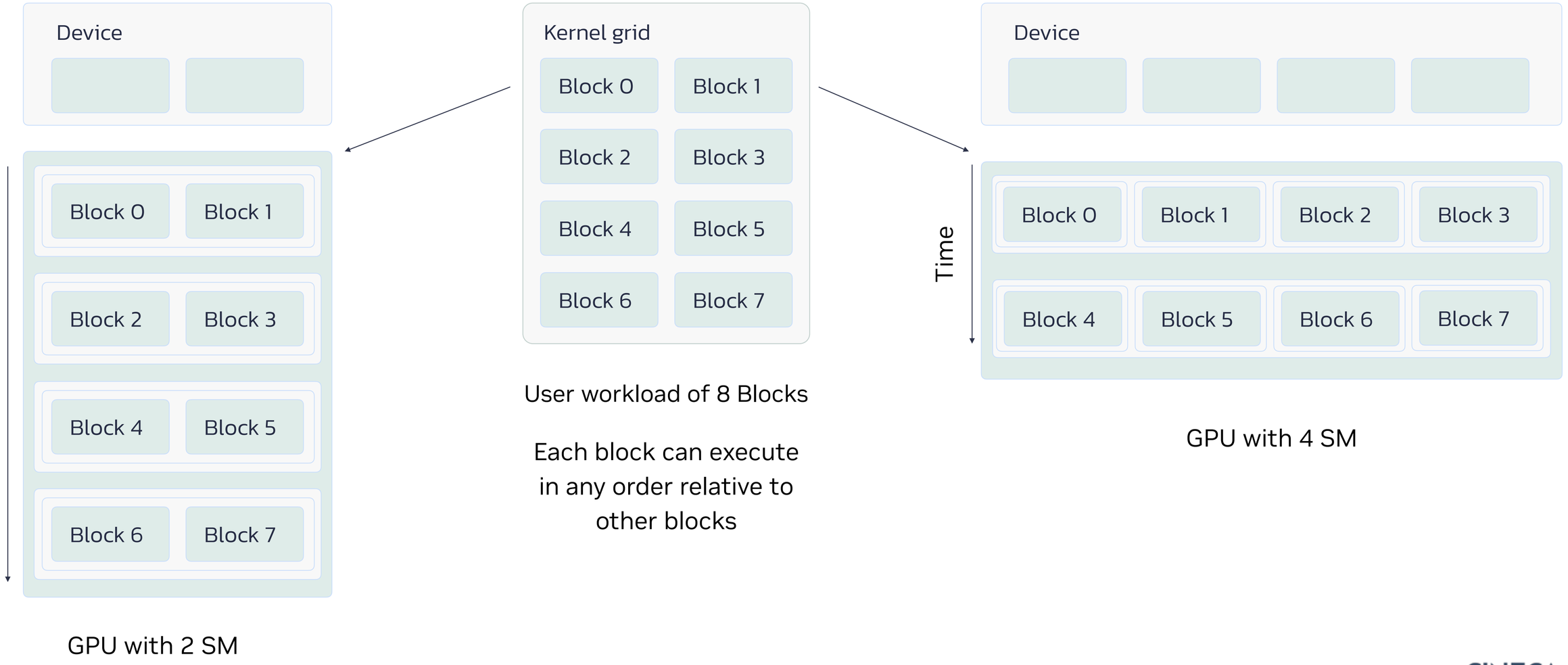
To increase parallelism, we need to **coordinate** work **among thread blocks**

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

Transparent scalability



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**

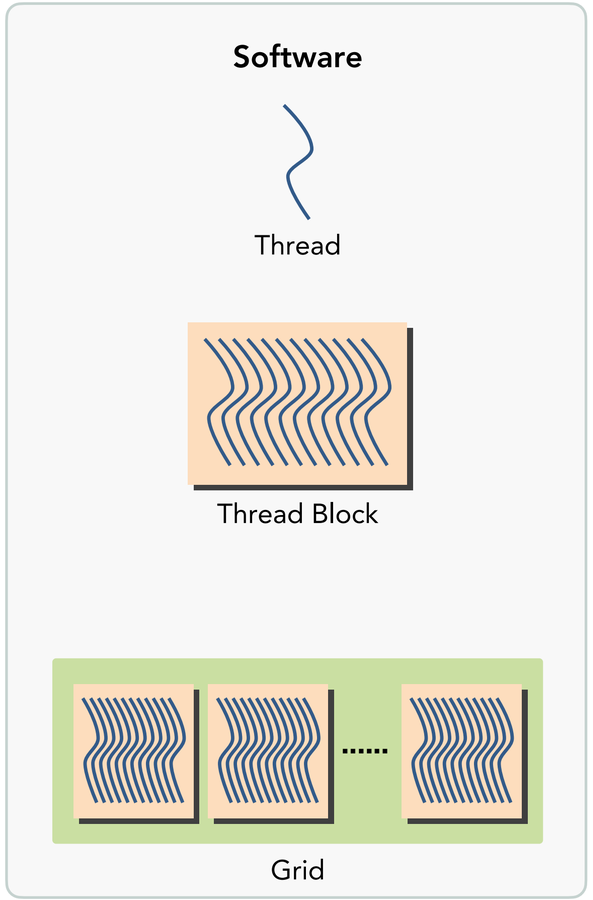
Each SM runs multiple thread blocks

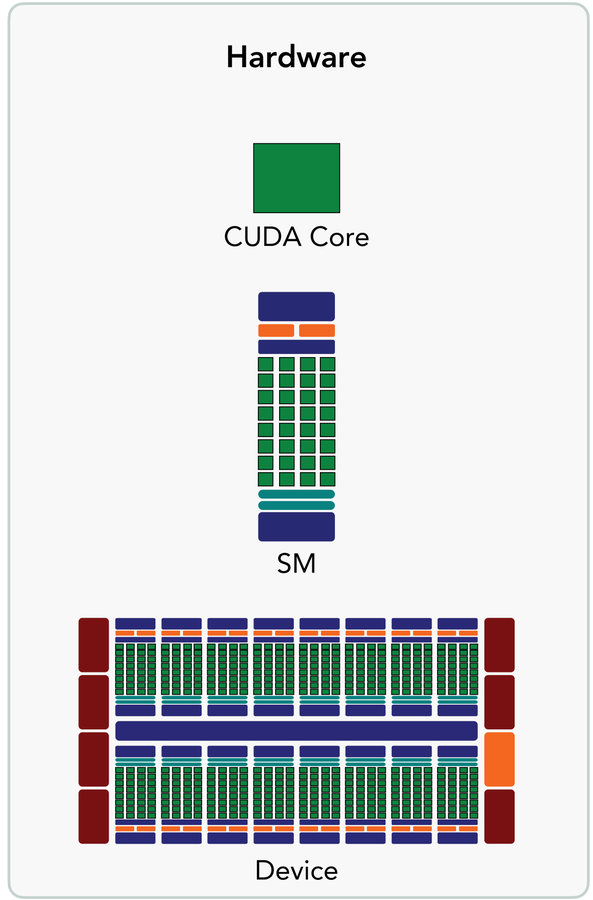
Each SP runs on thread from a thread blocks

3 **Grid blocks distributed to SMs**

Shared cache, register and memory

Global memory shared by all SMs

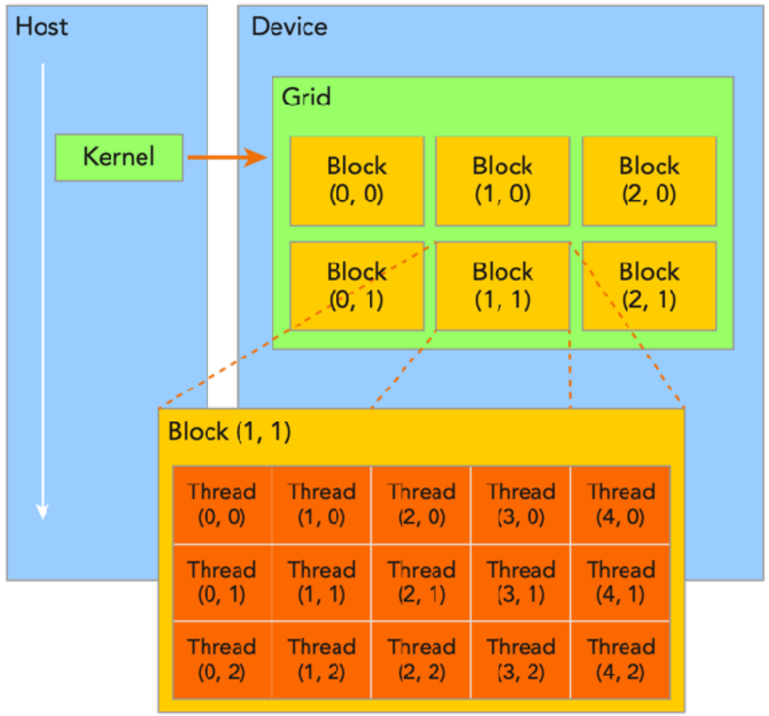




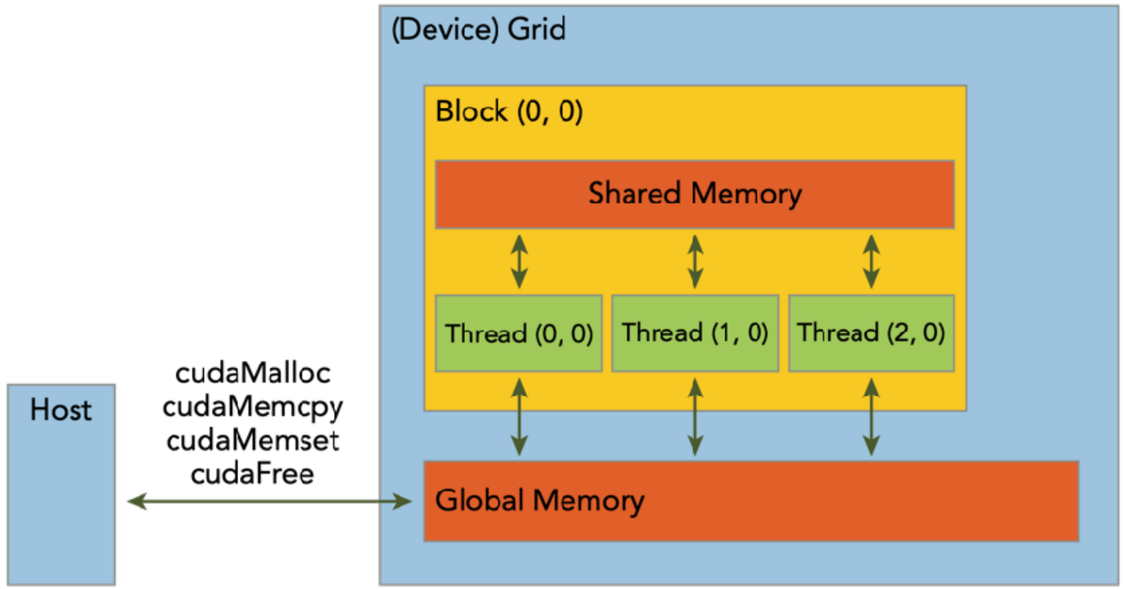
Compiling and running CUDA enable application

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

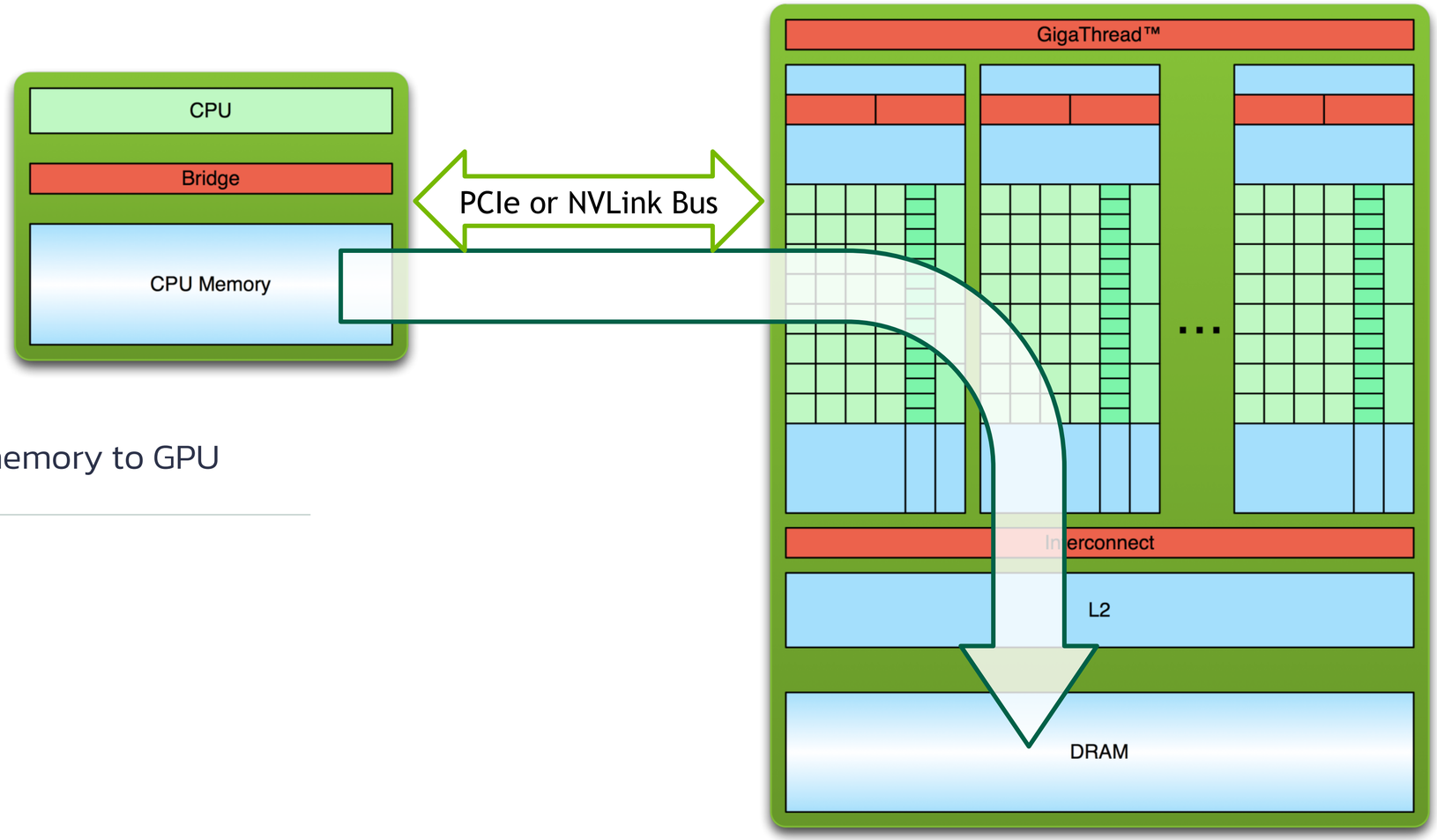
Thread hierarchy structure



Memory hierarchy structure

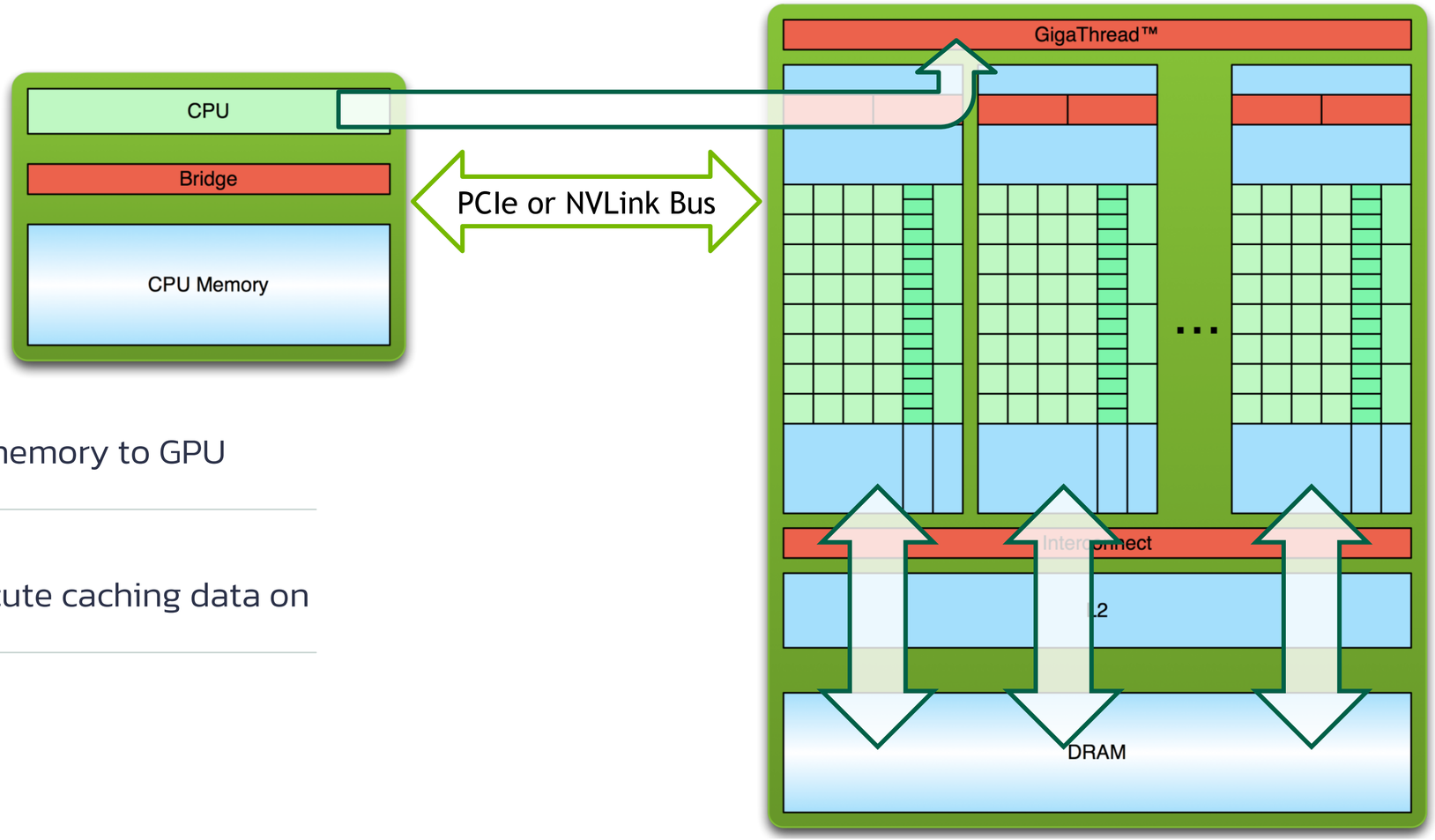


Three simple processing steps



1 Copy input data from CPU memory to GPU

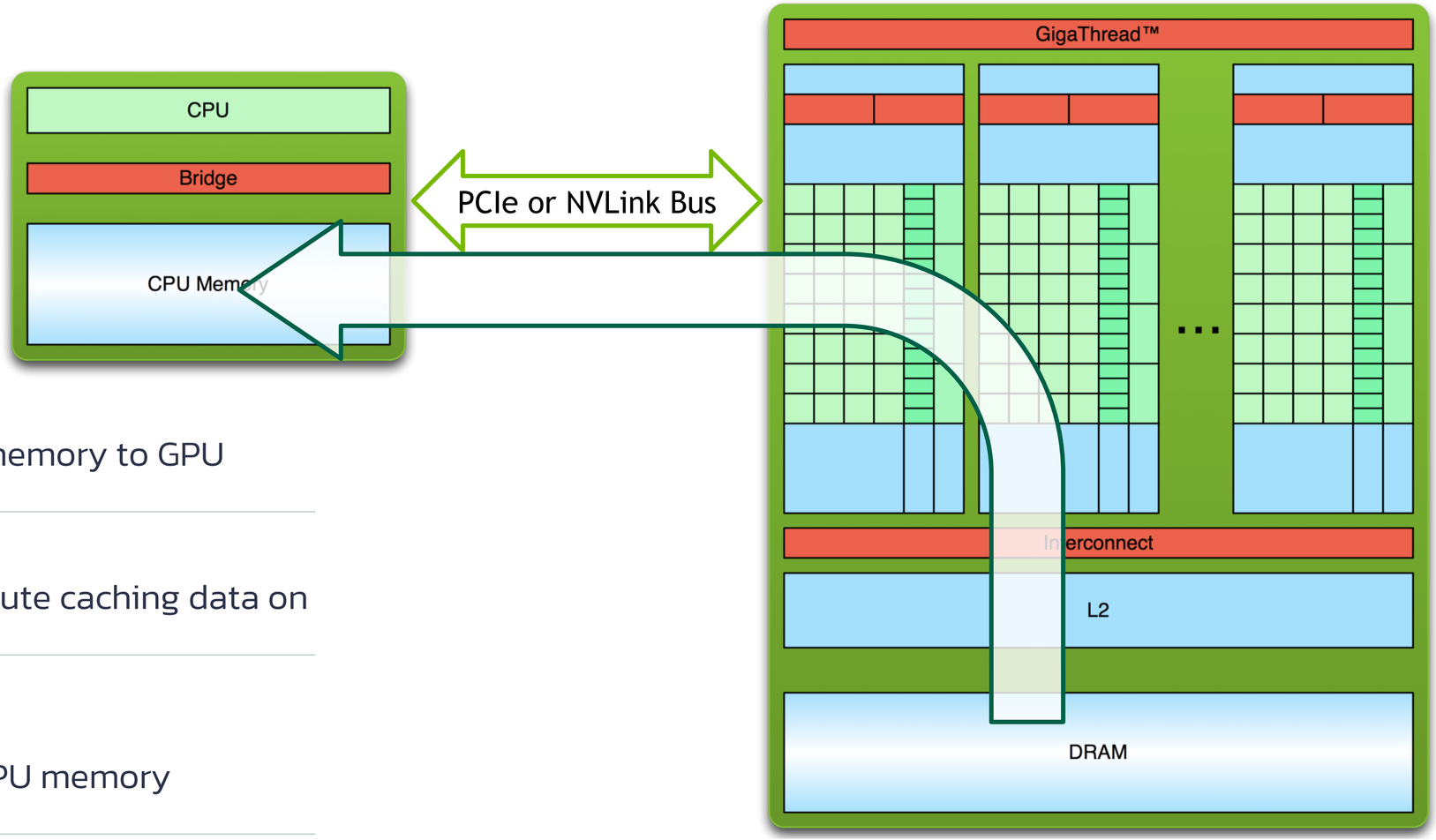
Three simple processing steps



1. Copy input data from CPU memory to GPU

2. Load GPU program and execute caching data on chip for performance

Three simple processing steps



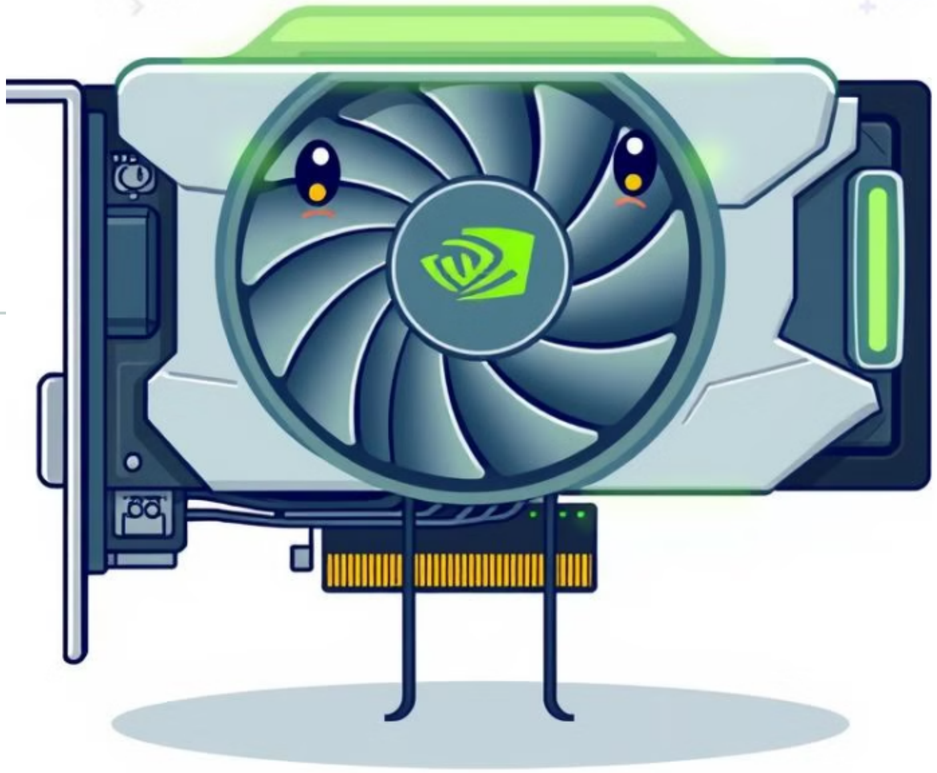
1. Copy input data from CPU memory to GPU
2. Load GPU program and execute caching data on chip for performance
3. Copy results From GPU to CPU memory

Data movement

1. Copy host to Device
2. Copy Device to host
3. Clean up memory for host and device

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

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

// Clean up memory
  checkCuda( cudaFree(d_A) );
    checkCuda( cudaFree(d_B) );
    checkCuda( cudaFree(d_C) );
    cleanup(h_A, h_B, h_C, h_C_ref);
```

1 How to compile CUDA enable application?



CUDA components

1

CUDA Driver

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

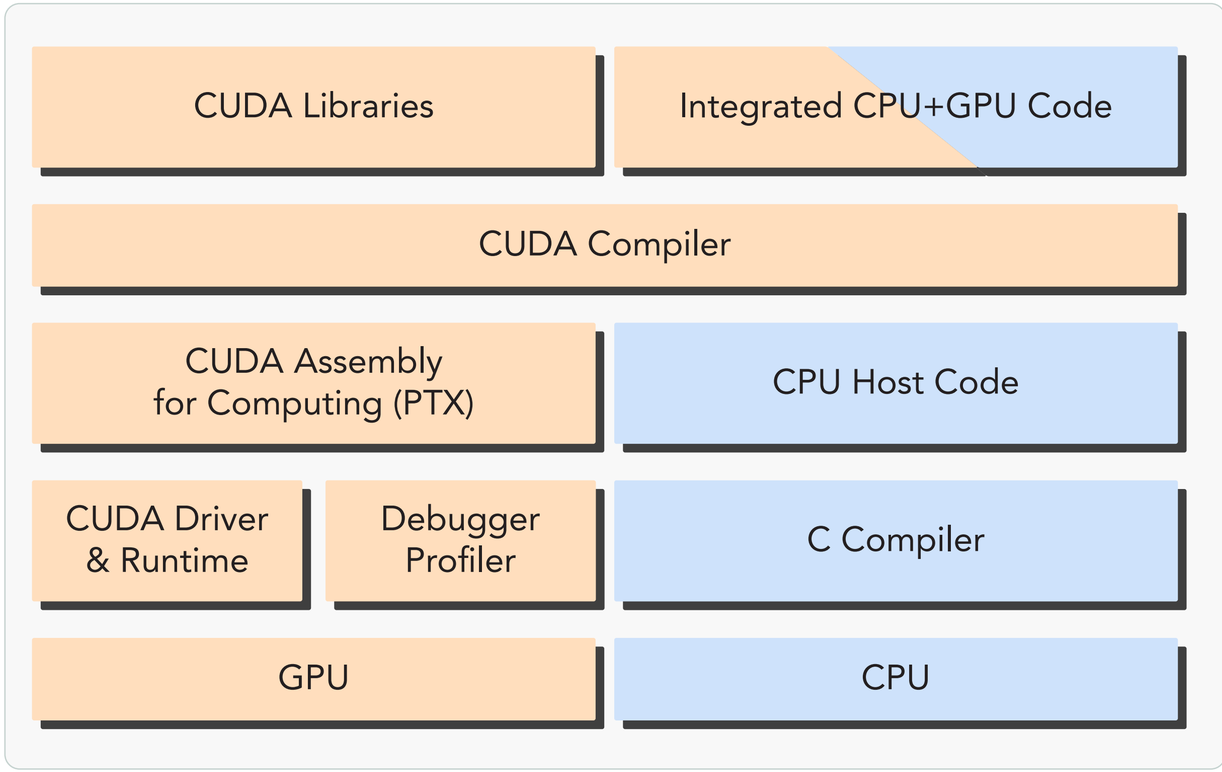
2

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



CUDA components

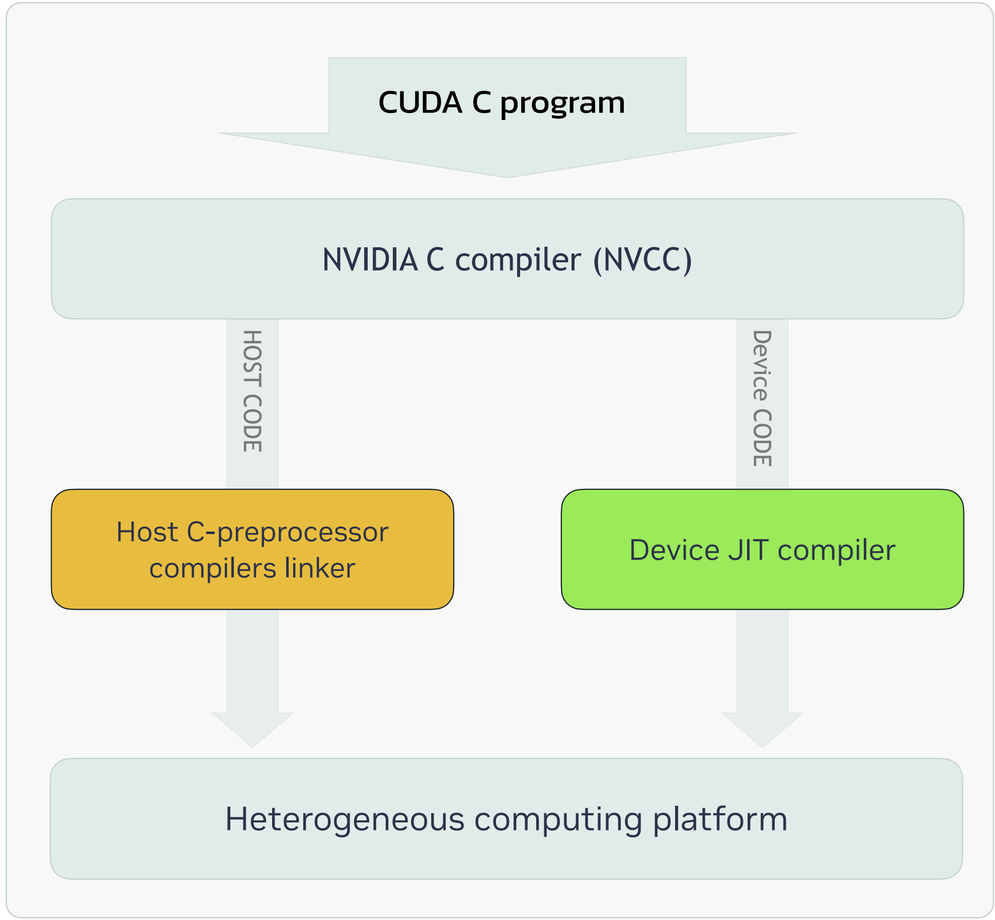
1

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



2

nvcc –arch=sm_70 –o out some–CUDA.cu –run

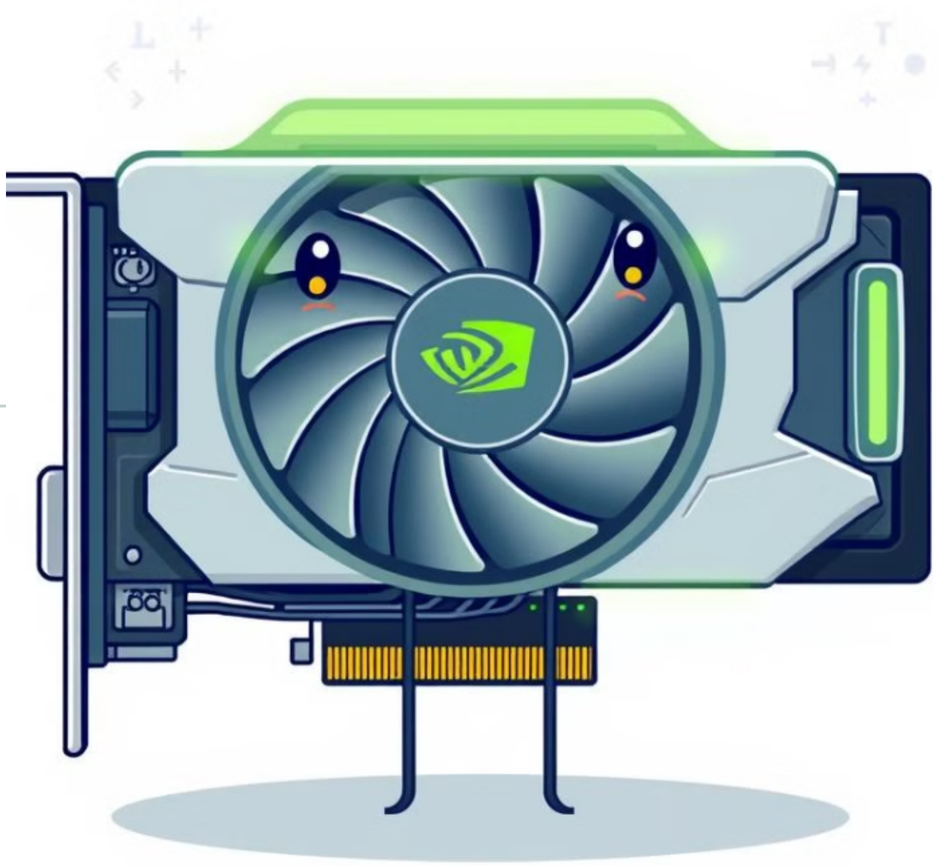
- arch: indicates for which architecture the files must be compiled (sm_80 is for TESLA A100 GPU)

- run: execute the successfully compiled binary

- Information on CUDA device: nvidia-smi, deviceQuery

2

Measuring performance and Error handling



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;
}
```

Kernel Launch 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() )
}
```

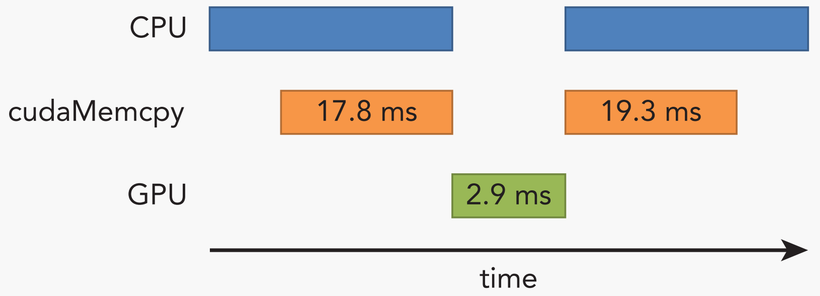
Asynchronous errors

To catch errors that occur in **asynchronous part** of the code (for example during the execution of an asynchronous kernel), check the **status returned by a subsequent synchronizing CUDA runtime API call,** such as cudaDeviceSynchronize.

```
cudaError_t asynchErr;
asynchErr = cudaDeviceSynchronize(); if (asynchErr != cudaSuccess)
{
      printf("Error: %s\n", cudaGetErrorString(err));
 }
```

Timing your kernel

```
double cpuSecond() {
    struct timespec ts;
    timespec_get(&ts, TIME_UTC);
    return ((double)ts.tv_sec + (double)ts.tv_nsec * 1.e-9);
}
/* Measure time for CPU execution */
    double start = cpuSecond();
    sumArraysOnCPU(h_A, h_B, hostRef, nElem);
    double cpuTime = cpuSecond() - start;
    printf("CPU Execution Time: %f seconds\n", cpuTime);
/* Measure time for GPU execution
    double 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);
```



Measuring performance with events

An event in CUDA is essentially a GPU time stamp that is recorded at a user-specified point in time. The API calls that create and destroy events, record events and convert timestamp difference into a floating-point value in milliseconds

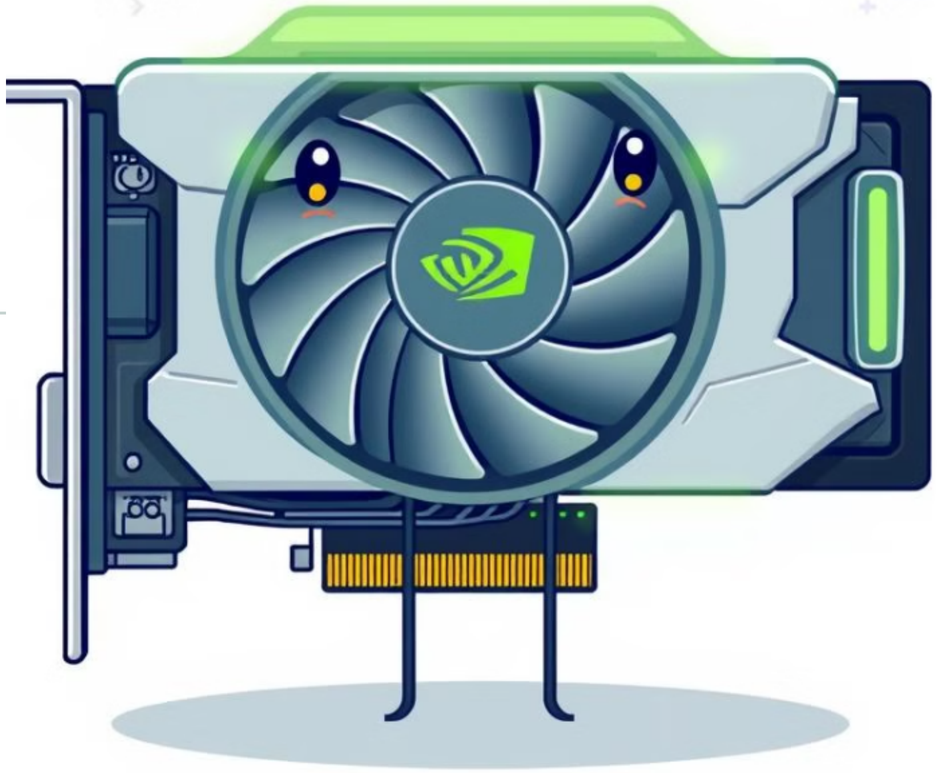
How to time code using CUDA events

```
cudaEvent_t start, stop;
float time;
cudaEventCreate(&start);
cudaEventRecord(&stop);
cudaEventRecord( start, 0 );
kernel<<<grid, threads>>> ( d_odata, d_idata, size_x, size_y, NUM_REPS);

// do some work on the GPU
cudaEventRecord( stop, 0 );
cudaEventSynchronize( stop );

cudaEventElapsedTime( &time, start, stop );
cudaEventDestroy( start );
cudaEventDestroy( stop );
```

Time your kernels

N	Elapsed Time on Host	Kernel Configuration	Elapsed Time on Device	Speed up [Second]
1 <<20	0.000757	(4096, 256)	0.000206	3.67
1 << 24	0.00013451	(4096, 256)	0.000447	30.12
1 << 26	0.052383	(524288, 128)	0.001013	51.72
1 << 29	0.424363	(524288, 128)	0.008173	51.92

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

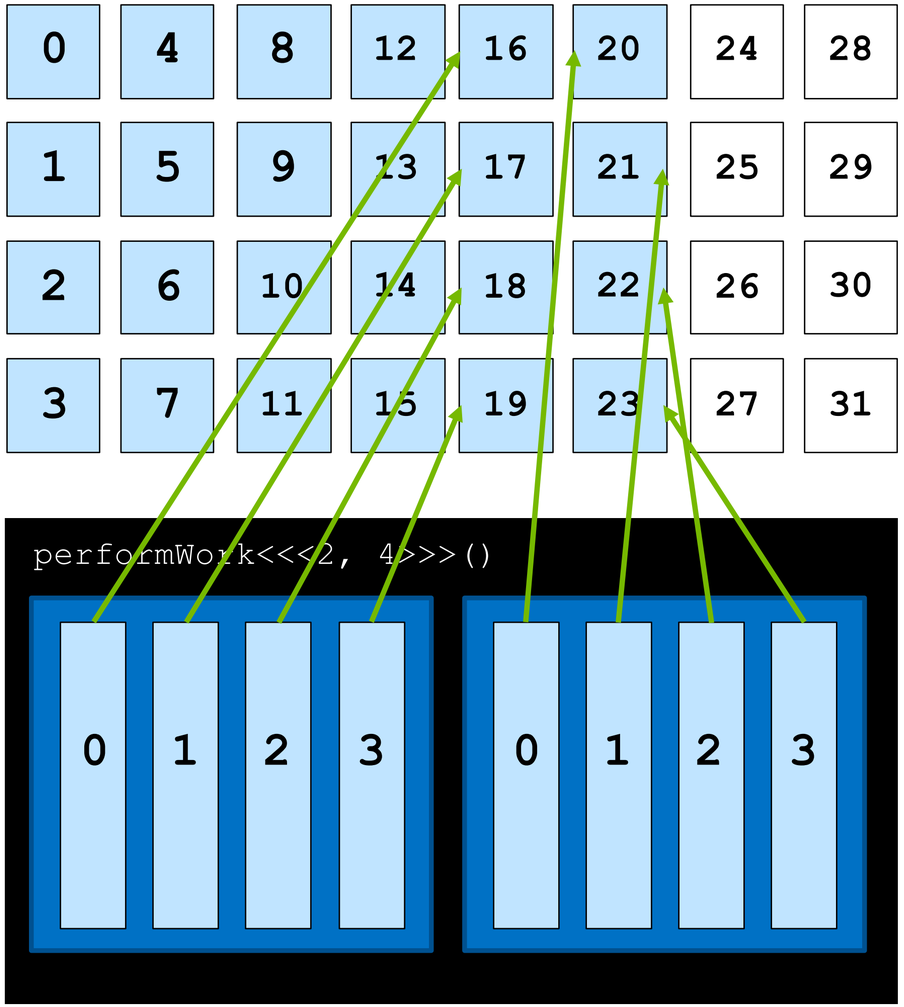


When the data set is larger than grid size?

Advantages of Grid-stride loops

- Scalability: handles any size of input data regardless of hardware contains. It ensures all the data is processed
- Efficient resource utilisations: It allows the kernels to utilise all available threads efficiently by feeding more jobs
- Simplicity: straight forward implementation, without needing any complex logic to manage the devision of the work

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



Ways to improve your code

1

Types of Data transfer

Pageable and Pinned memory

Unified memory and Asynchronous Prefetching

2

Global memory reads/writes

Aligned and coalesced memory accesses that reduce wasted bandwidth

Array of Structure versus Structure of Array

Overlapping Kernel and Data movement by using non-default streams

3

Performance tuning

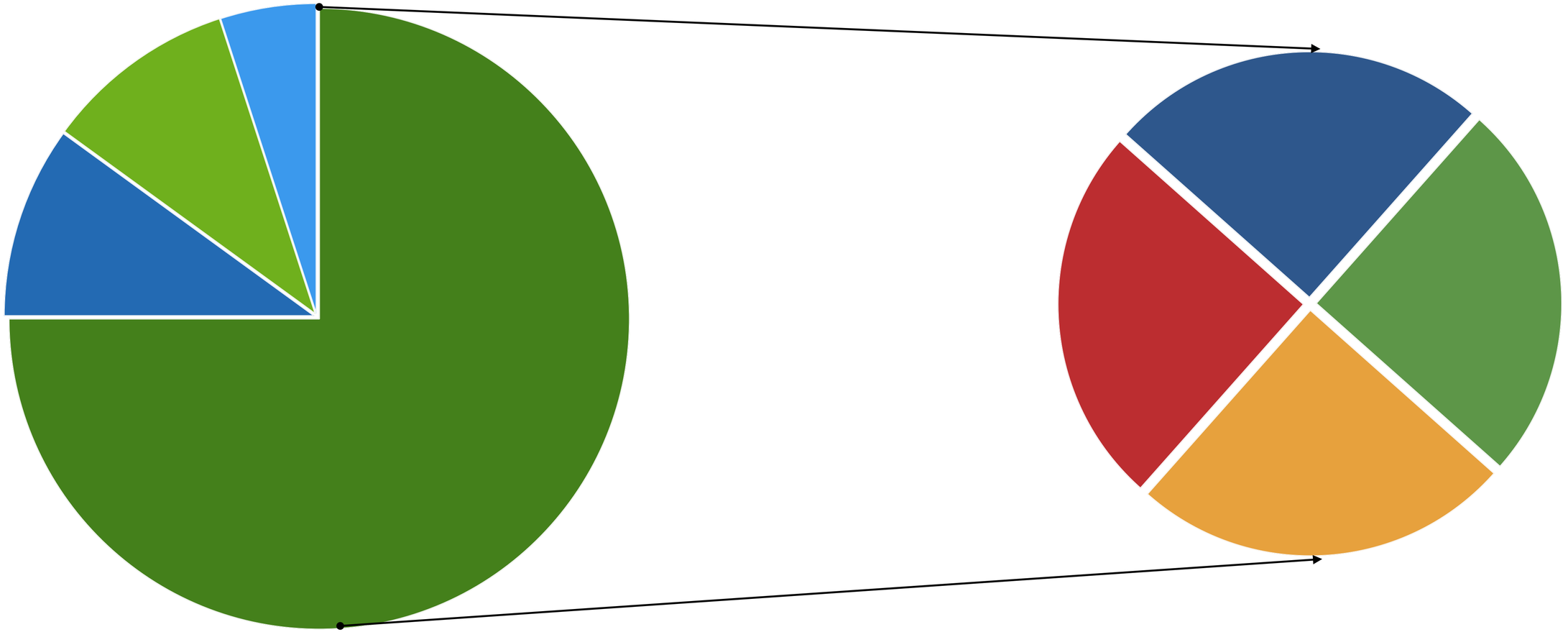
Parallelising higher dimensions-2D

Unrolling techniques

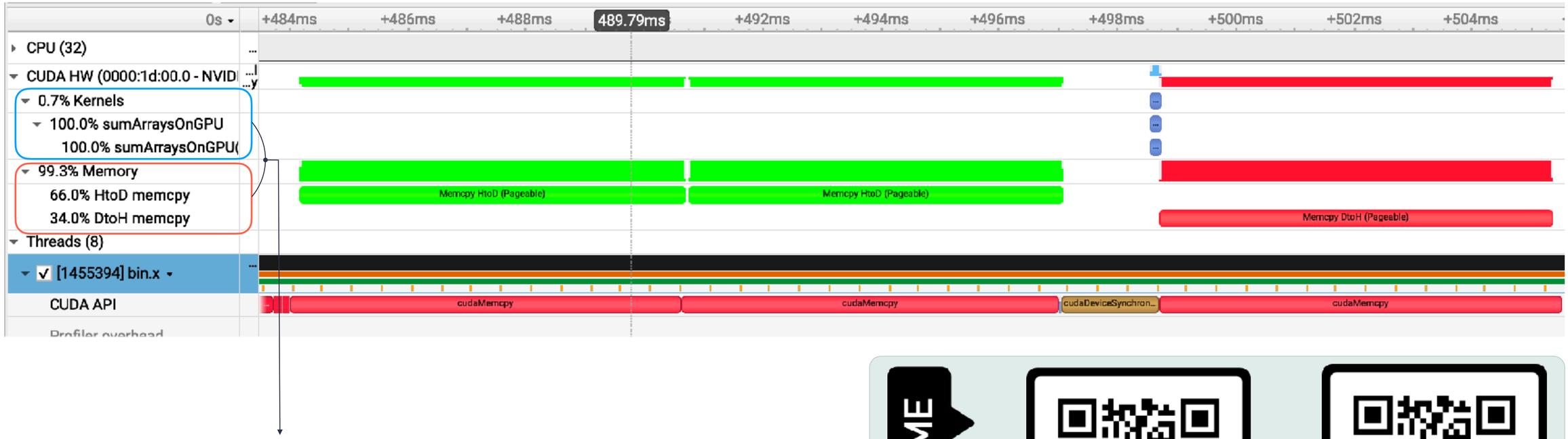
Matrix Transpose Problem

Shared memory

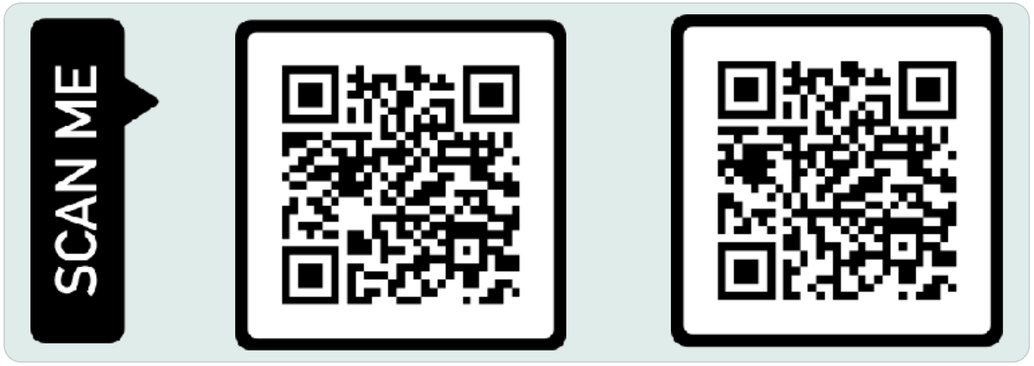
Data transfer impacts on performance



Measuring performance with events



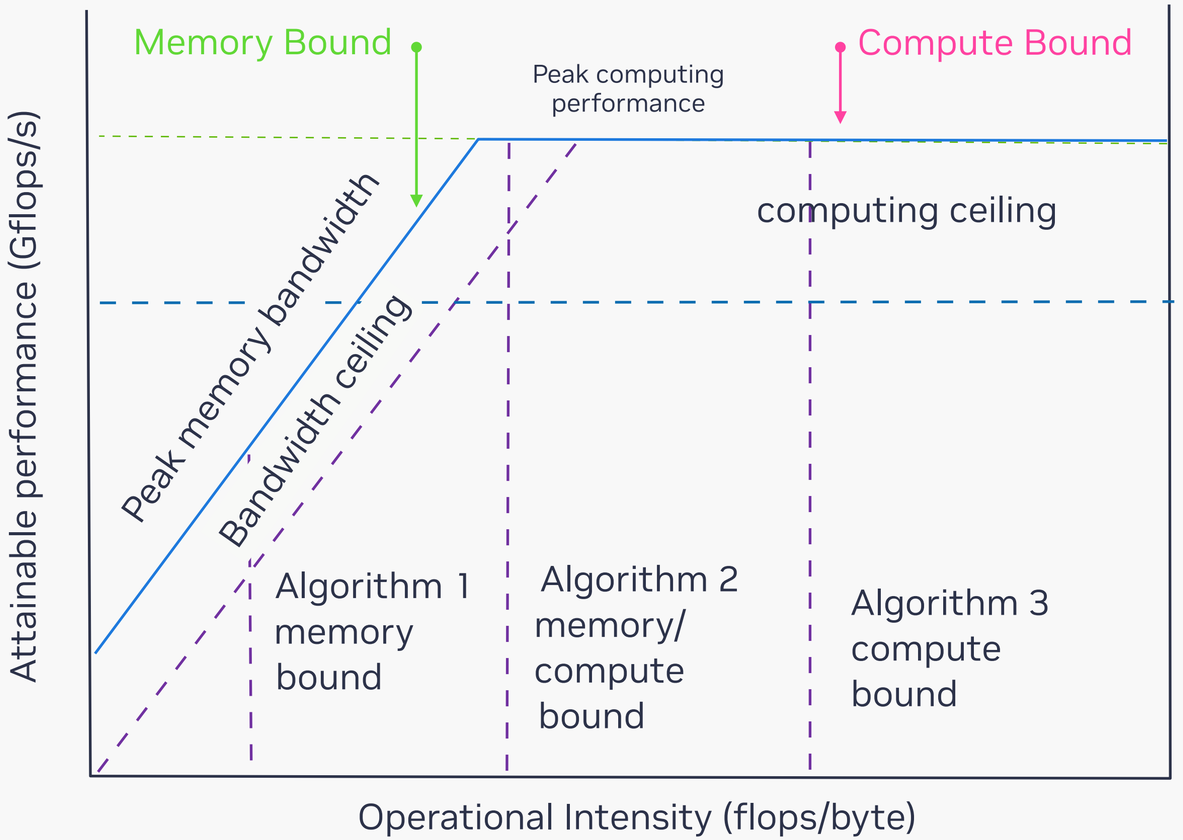
Important to minimise the transfer between the host and device



Application Performance constraints

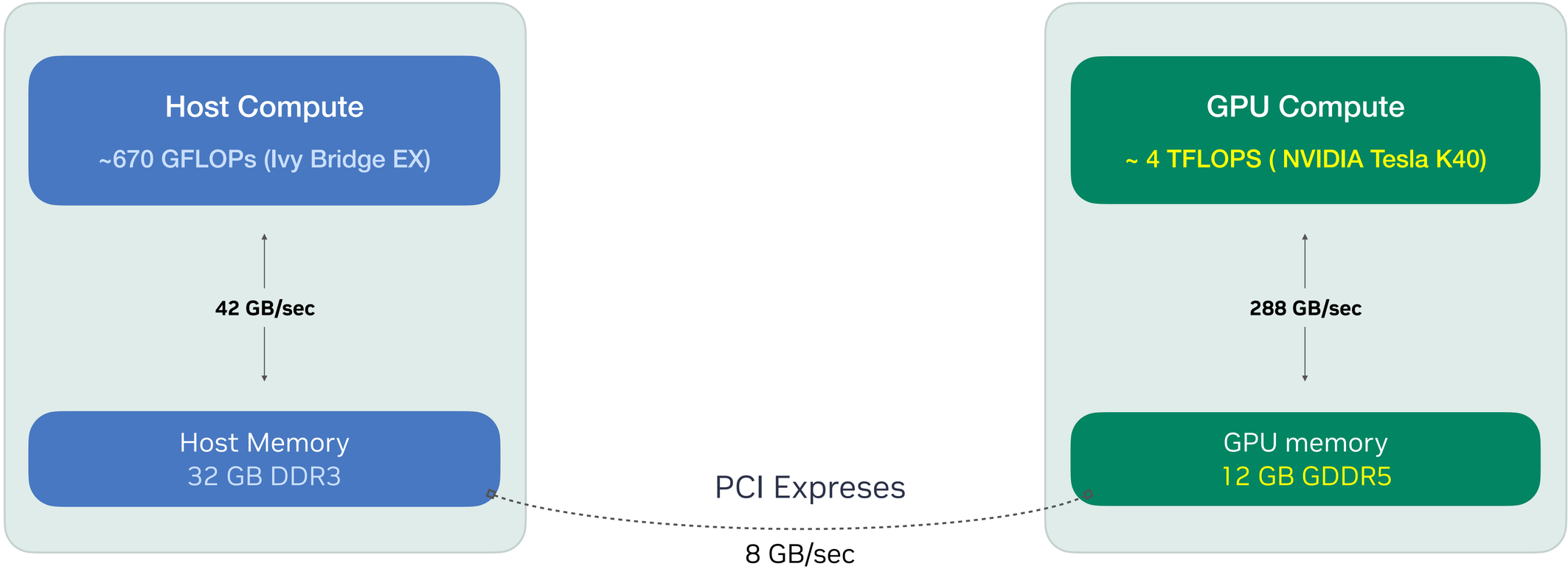
Roofline Model

- 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)



GPU vs. CPU: Understanding Performance Trade-offs

Impact of data transfer on overall application performance



Understanding CUDA Memory Hierarchy

Registers

Fastest, smallest memory

L1 Cache

Fast, small, on-chip

Shared Memory

Medium speed, shared among threads

Global Memory

Slowest, largest, off-chip

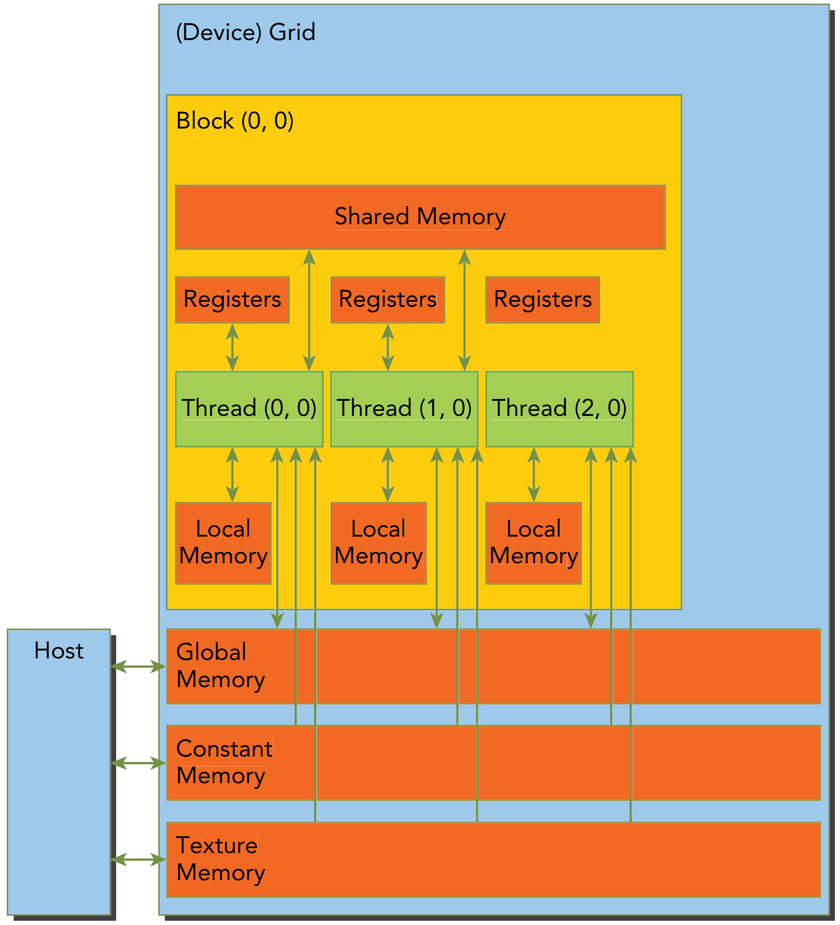
GPU memory breakdown

Device code can

- R/W per-thread registers
- R/W per-thread Local Memory
- R/W per-block Shared Memory
- R/W per-grid global Memory
- Read only per-grid Constant Memory
- Read only per-grid Texture Memory

Host code can

- Transfer data to/from per-grid global and constant memories



CUDA Variable Declaration Summary

QUALIFIER	VARIABLE NAME	MEMORY	SCOPE	LIFESPAN
	`float var`	Register	Thread	Thread
	`float var[100]`	Local	Thread	Thread
`__shared__`	`float var` †	Shared	Block	Block
`__device__`	`float var` †	Global	Global	Application
`__constant__`	`float var` †	Constant	Global	Application

CUDA memory management

1. Memory allocation

Process of reserving memory space for a variable or data structure Memory allocation can be performed using different memory types, such as global, shared and constant memory

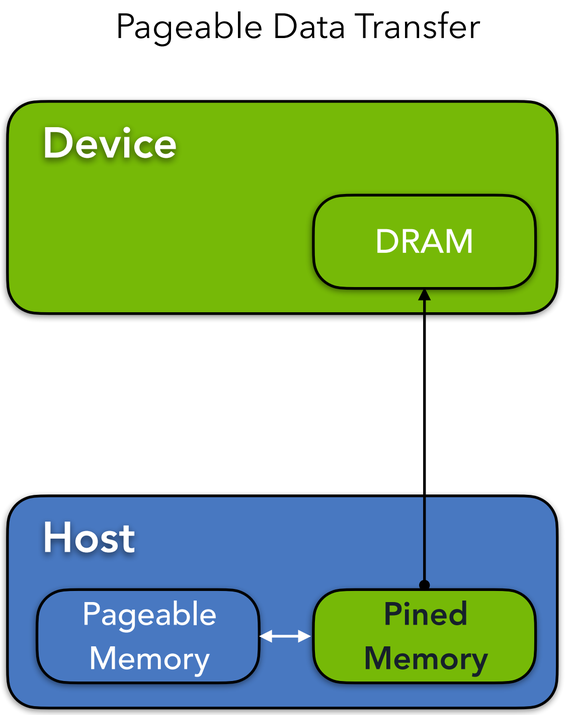
2. Memory transfer

Process of copying data from one memory location to another
Memory copy can be performed using different memory types, such as host memory and device memory

3. Memory synchronization

Process of coordinating the access of multiple threads to shared memory or global memory
Synchronization primitives: atomic operations, barriers, and locks

Data transfer between host and device



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

Data memory allocation/release

- cudaMemcpy (void* dst, void *src, size_t nbytes, **cudaMemcpyKind** kind)
  - Direction specifies locations (host or device) of src and dst
  - Blocks CPU thread (returns after the copy is complete)
  - Does not start copy until previous CUDA calls complete
- Kind: specifies the direction of the memory copy
  - cudaMemcpyHostToHost
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost

- CudaFree(devPtr)
  - Free memory from device Global memory
  - Pointer to free object

Data memory allocation/release

Refers to the coordination of threads accessing global memory or shared memory

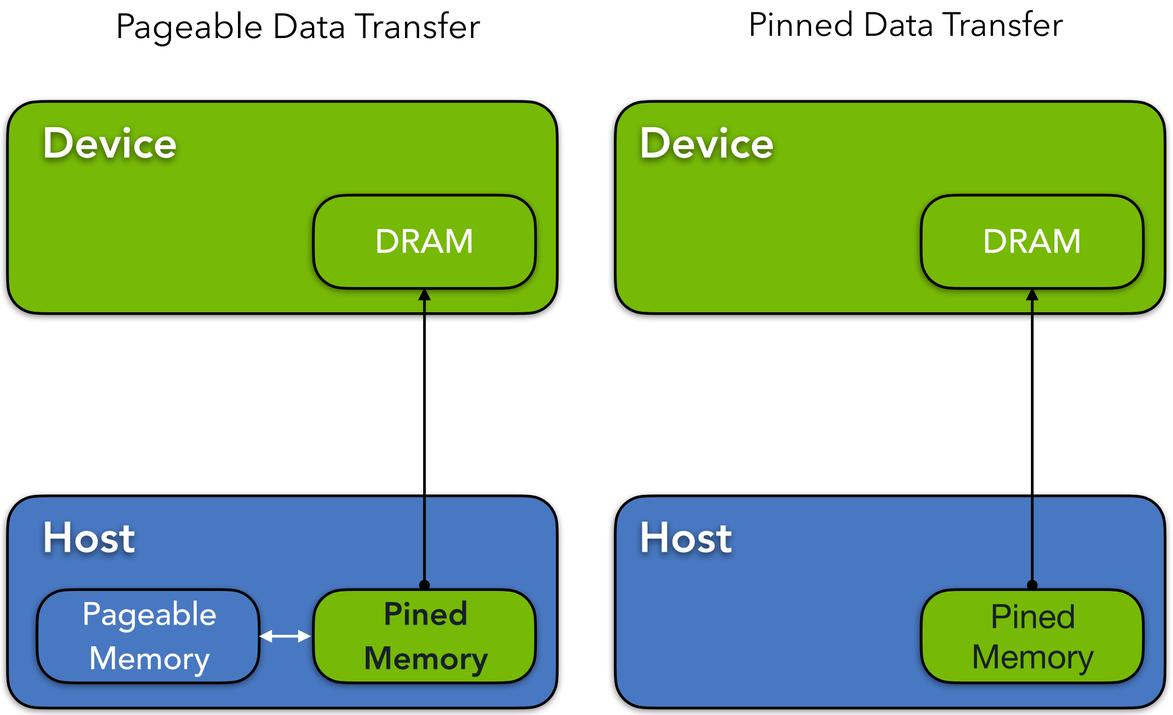
- Device synchronization
  - In CUDA, the CPU and the GPU operate asynchronously
  - Synchronization is necessary to ensure that the GPU has finished executing before continuing with the CPU code

```
cudaMemcpy(d_data, h_data, size * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpyAsync(h_result, d_result, size *  sizeof(float), cudaMemcpyDeviceToHost);
cudaDeviceSynchronize();
```

- Thread synchronization
  - Threads within a block can access shared memory, which is a memory space shared among all threads in a block
  - Ensure that threads accessing shared memory do not interfere with each other

```
__syncthreads();
// compute using shared memory
```

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 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 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) );
```

Vector sum pageable and pinned memory transfer

N	Pageable mem transfer	Pinned mem transfer	SlowDown
1 << 20	0.000500	0.00036	0.72
1 << 22	0.000486	0.000225	0.462962962962963
1 << 24	0.001842	0.002379	1.29153094462541
1 << 26	0.003168	0.001021	0.322285353535354
1 << 28	0.004015	0.007195	1.7920298879203
1 << 30	0.029974	0.019884	0.663374924934944

Zero-copy memory

Host cannot access device variables and device cannot access host variables directly, one exception rule to this : zero copy memory

1

GPU threads can directly access zero-copy memory

- Leveraging host memory when there is insufficient device memory
- Avoiding explicit data transfer between the host and device
- Improving PCIe transfer rates
- When using zero-copy memory to share data between the host and device, you must synchronise memory access across the host and device

2

CUDA API call

- cudaHostAlloc(void **ptr, size_t size, unsigned int flags);
- flags = cudaHostAllocMapped, cudaHostAllocDefault, cudaHostAllocPortable
- Most relevant flag to zero-copy memory is cudaHostAllocMapped, which returns host memory that is mapped into the device address space

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

Unified virtual memory (UVM)

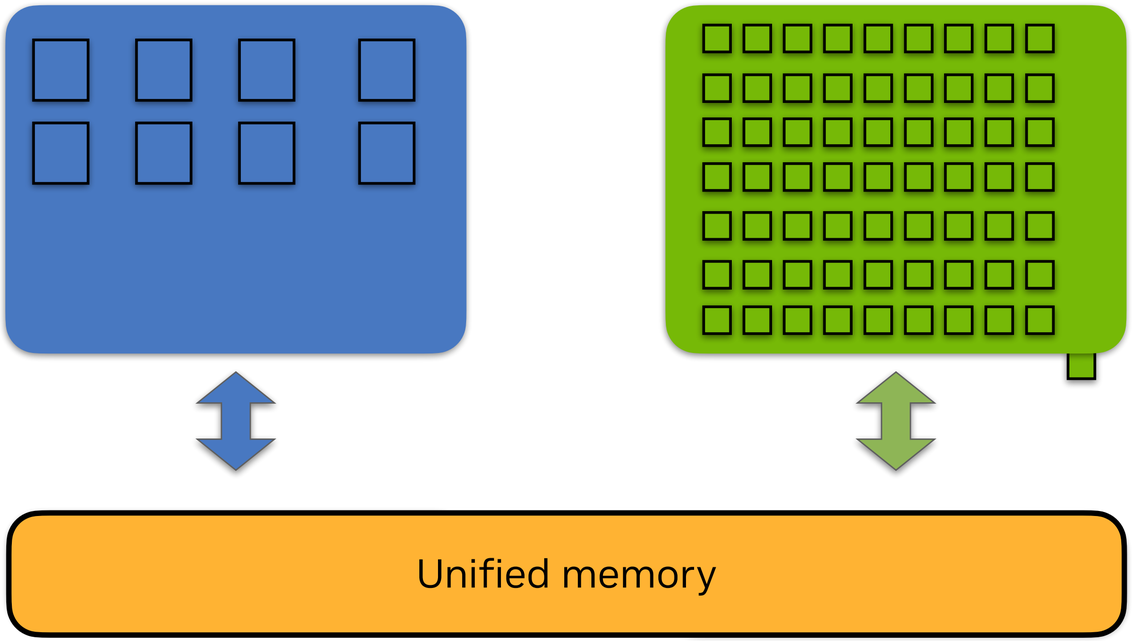
Increased memory latency

- 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

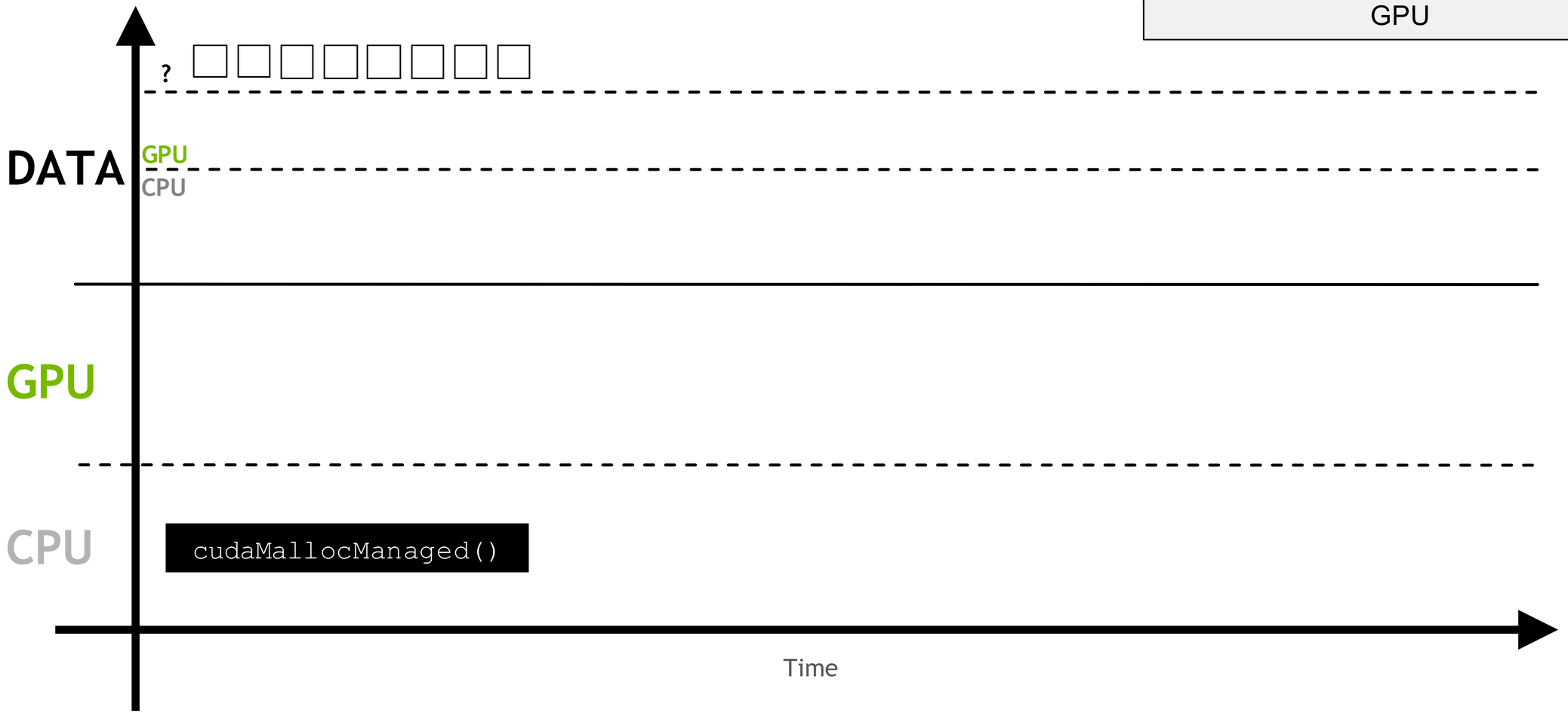
UVM automatically manages memory placement, which may not always be optimal for a given application

Developer view of GPU memory



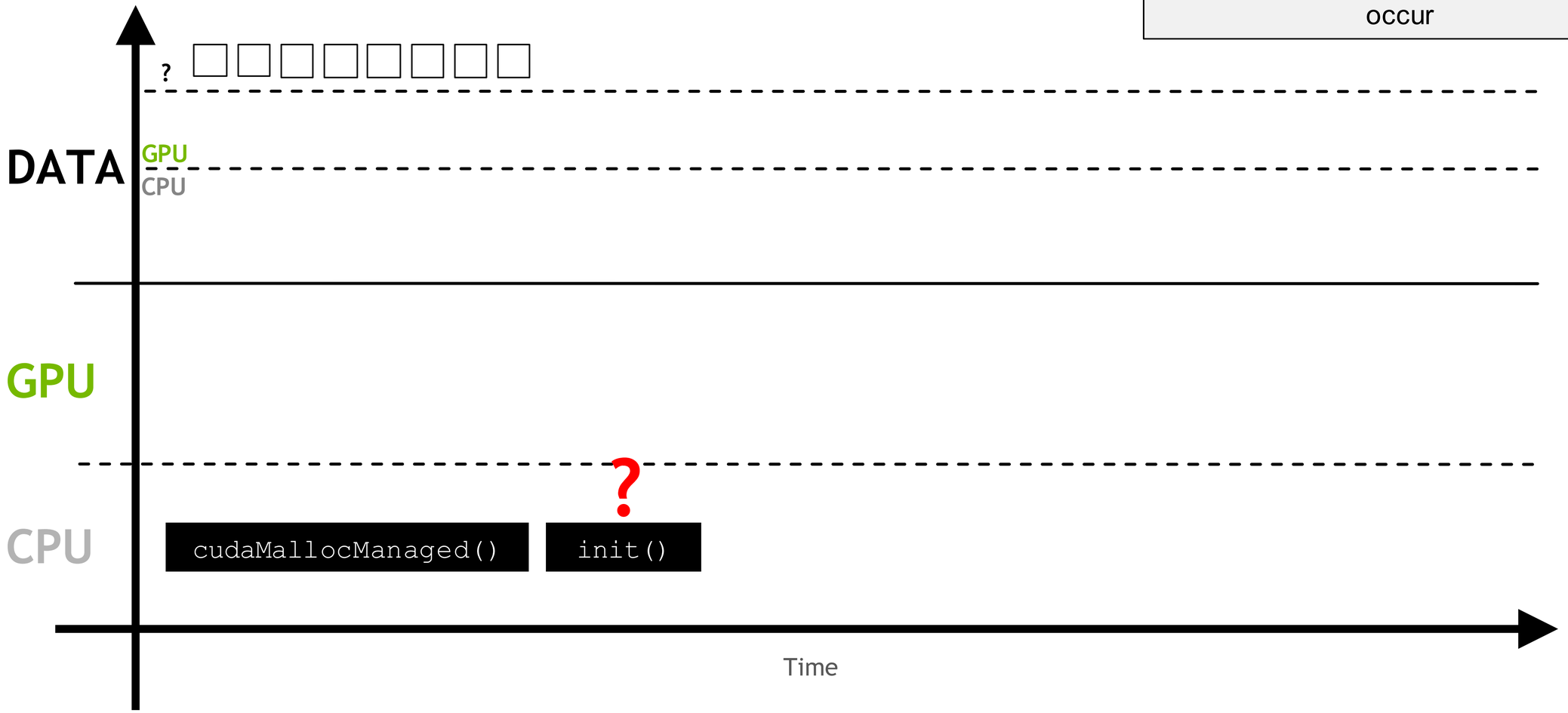
How does cudaMallocManaged actually works?

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



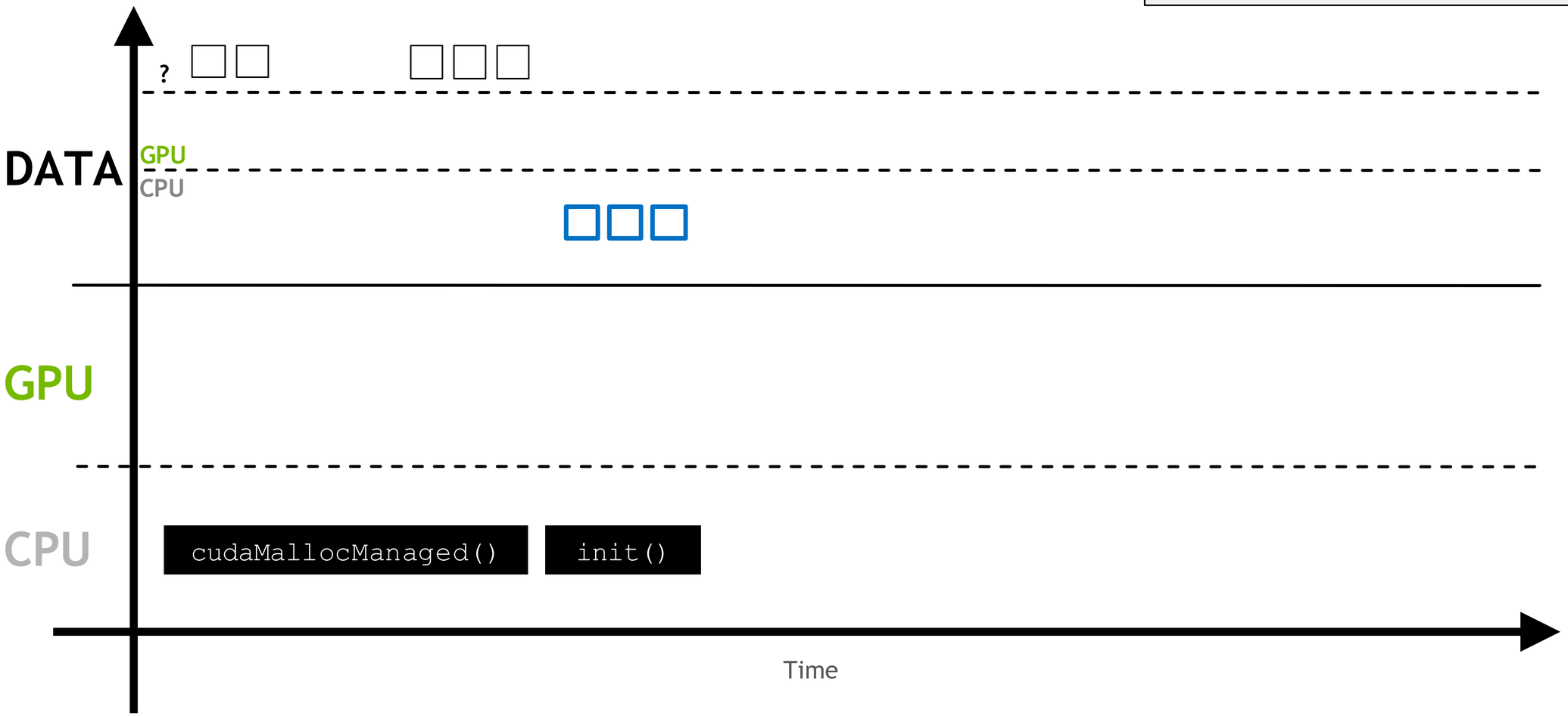
How does cudaMallocManaged actually works?

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



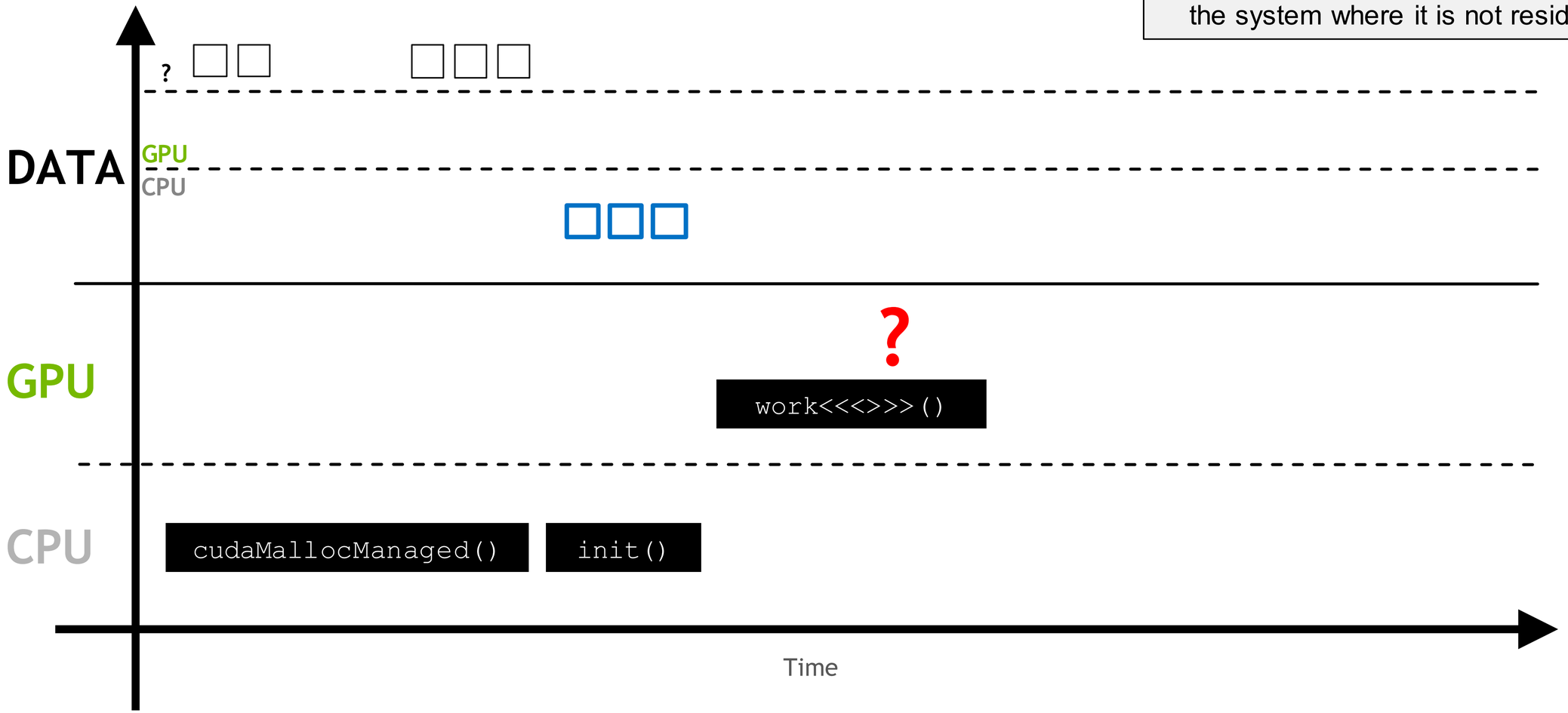
How does cudaMallocManaged actually works?

The page fault will trigger the migration of the demanded memory

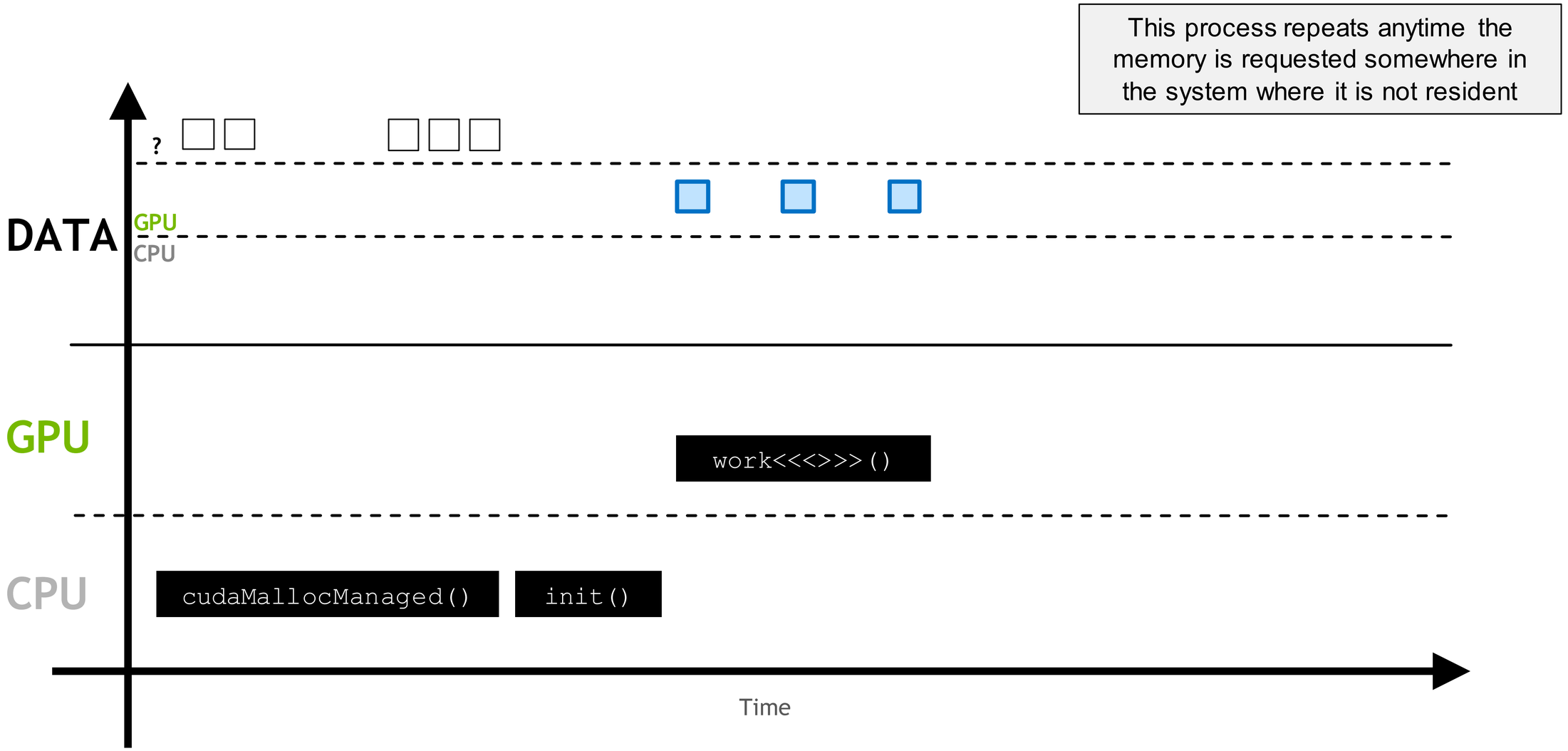


How does cudaMallocManaged actually works?

This process repeats anytime the memory is requested somewhere in the system where it is not resident



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);
```

Vector sum Unified memory transfer

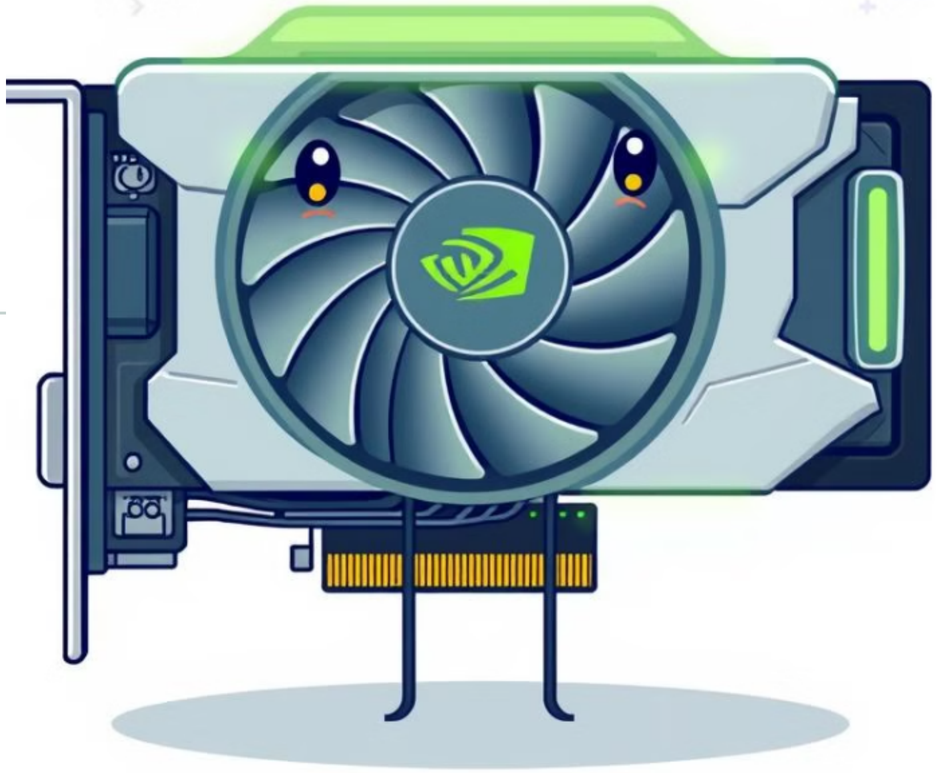
Unified memory Transfer

```
/* Unified Memory allocation */
   float *a, *b, *hostRef, *gpuRef;
   checkCuda(cudaMallocManaged(&a, size));
   checkCuda(cudaMallocManaged(&b, size));
   checkCuda(cudaMallocManaged(&hostRef, size));
   checkCuda(cudaMallocManaged(&gpuRef, size));
```

```
/* 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() );
    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) );
```

4

Performance consideration



Best Practices for porting a code

1

Understand the application

Mini app, Understand if the kernel is memory or compute bound

2

Identify Hot Spots

Analyze your application's memory access patterns and identify the critical data that should be prefetched

3

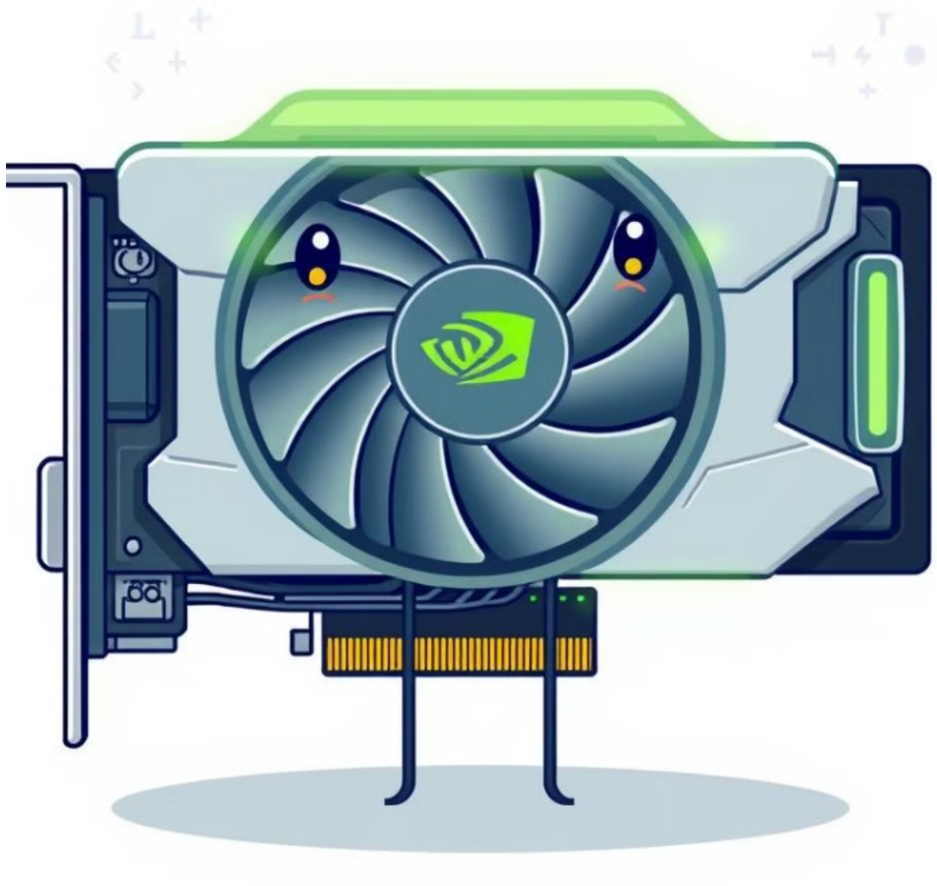
Time Prefetching

Carefully time the prefetch operations to overlap with kernel execution and minimize latency

4

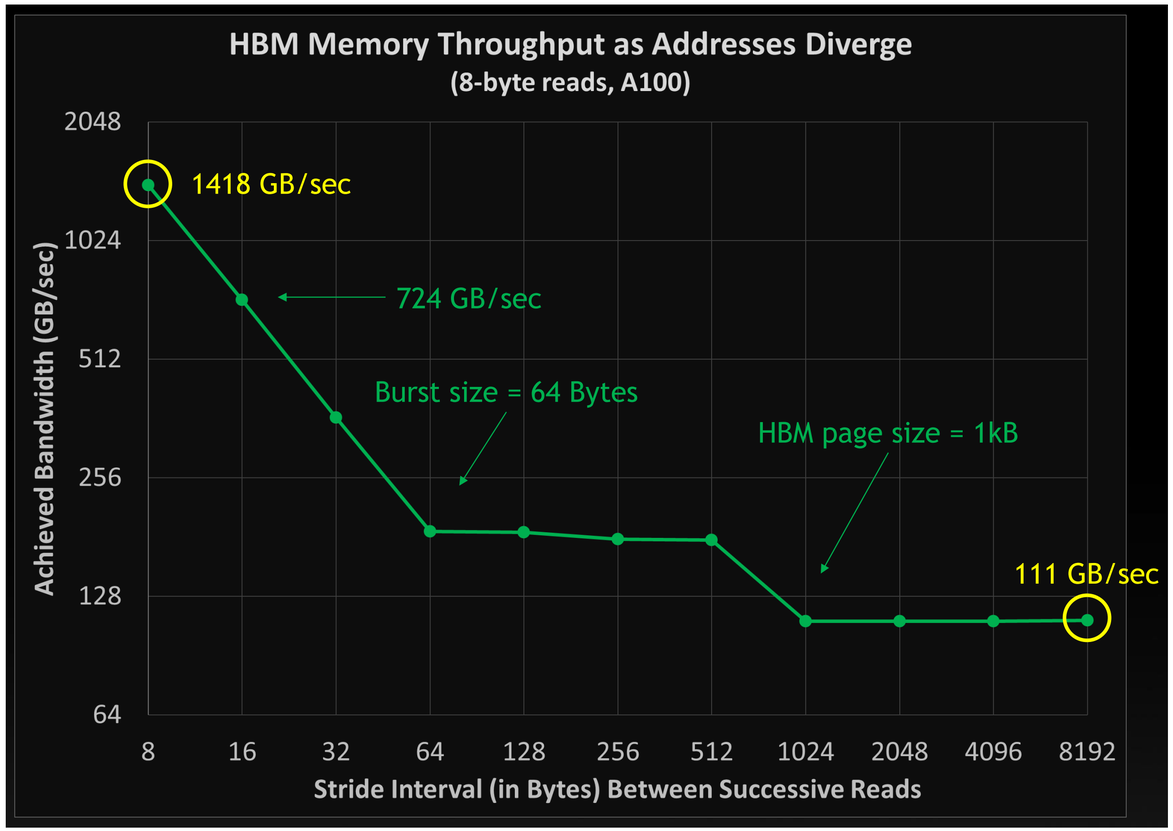
Monitor Performance

Use profiling tools to measure the impact and fine-tune its usage: profiler the code with Nsight-system + NVTX, Nsight compute

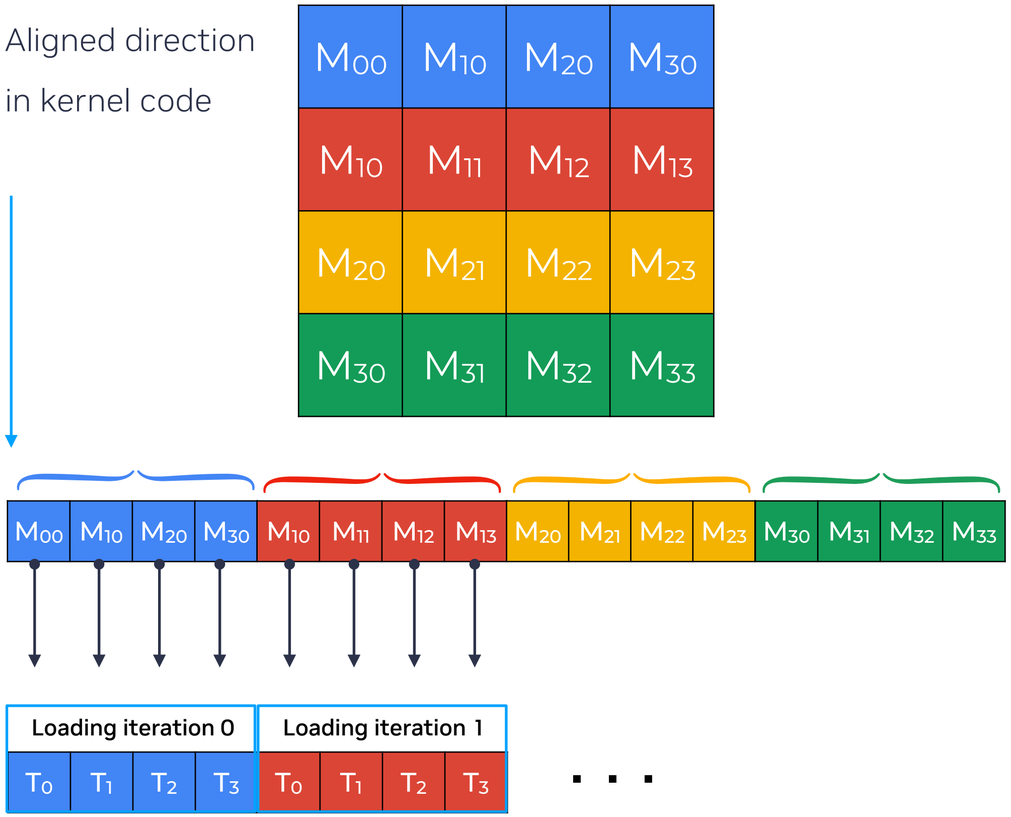


It’s all about memory access patterns

Depending on how you access memory bandwidth can very greatly!



Memory access patterns

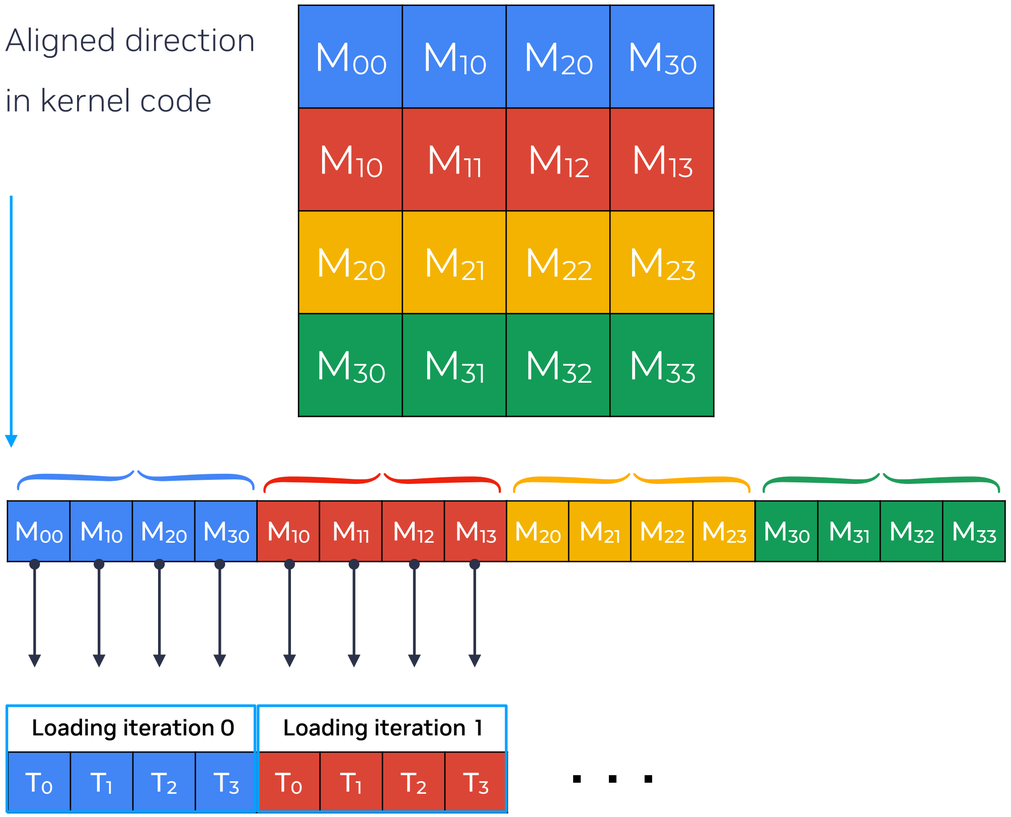


- For blocks that consist of multiple dimensions of threads, the dimensions will be projected into a linear order before partitioning into warps
- Each thread is shown as M(x,y), with x being the threadIdx.x and y being threadIdx.y for the thread
- Cooperatively, the 32 threads in a warp present a single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

Memory access patterns

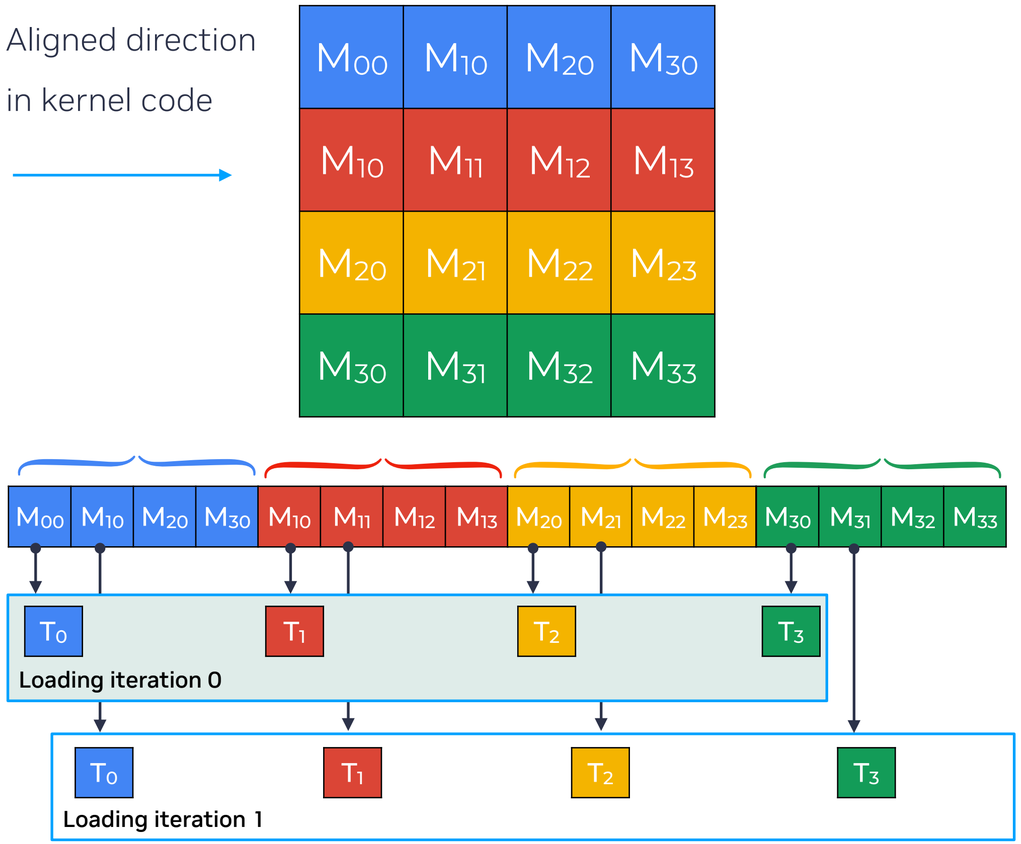
Aligned direction

in kernel code



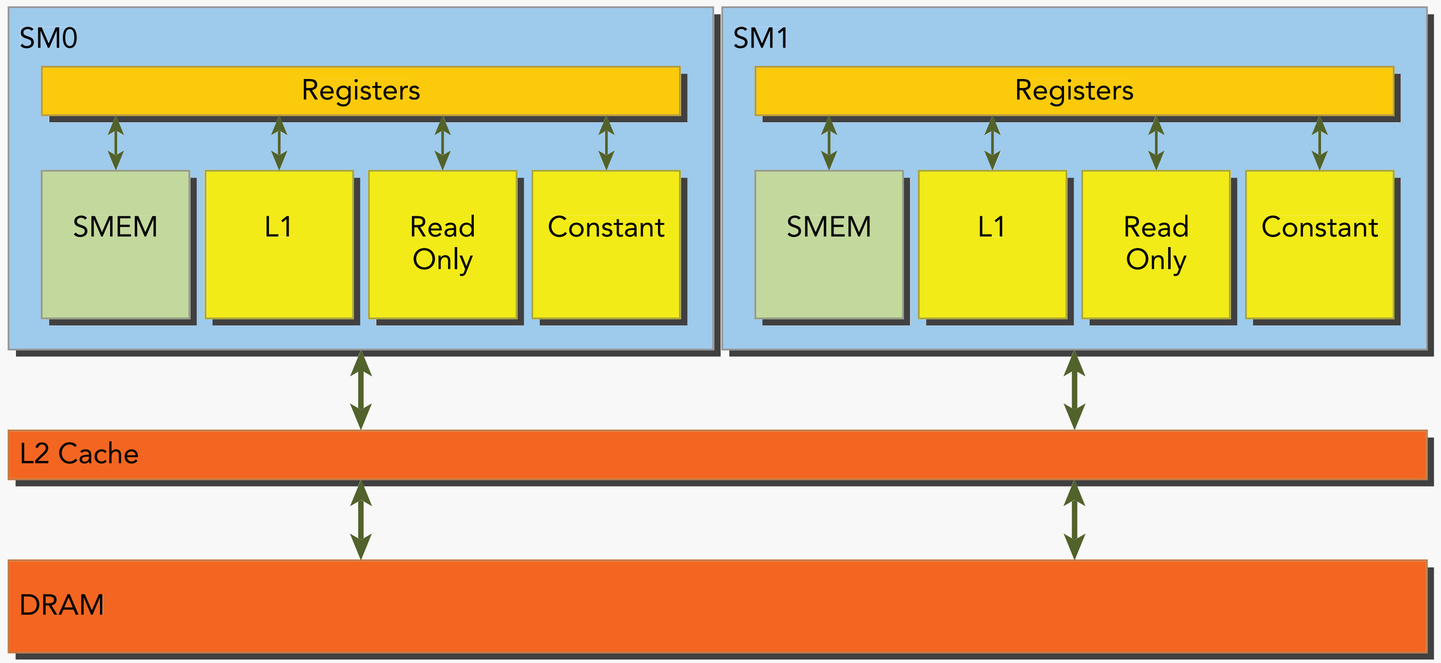
Aligned direction

in kernel code



Memory bandwidth limits GPU-enabled applications

- 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
- P**erfect mapping** occurs when each thread in a warp requests one 4-byte value, matching the 128-byte cache line size

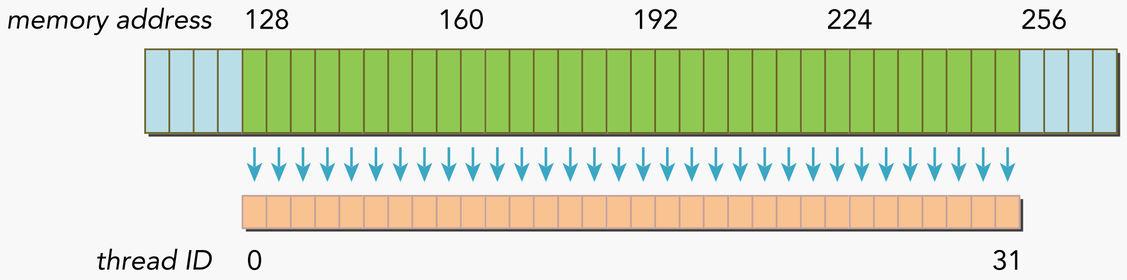


Efficient memory access is crucial

Aligned Memory Access

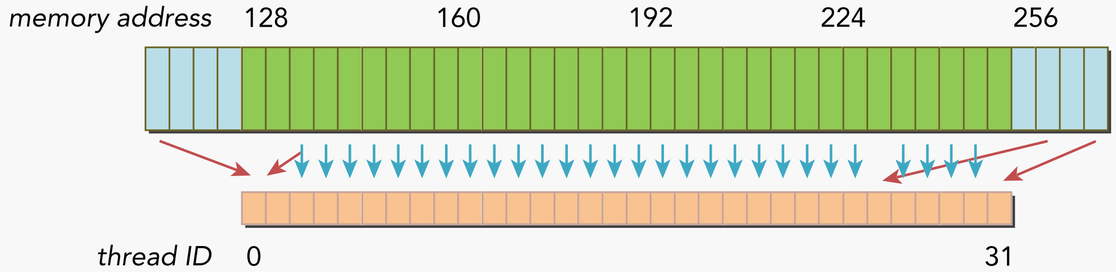
accessed by threads are arranged such that each thread accesses data in consecutive memory locations

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



Efficient memory access is crucial

```
__global__ void sumAddalignedacces(float *a, float *b, float *c, int n, int offset) {
    for (int idx = offset, k = 0; idx < n; idx++, k++)
      C[k] = A[idx] + B[idx];
}
```

```
__global__ void missedAlignedAccessed(float *a, float *b, float *c, int n) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    int k = i + offset;
    if (int i < k) { c[i] = a[i] + b[I]; }
}
```

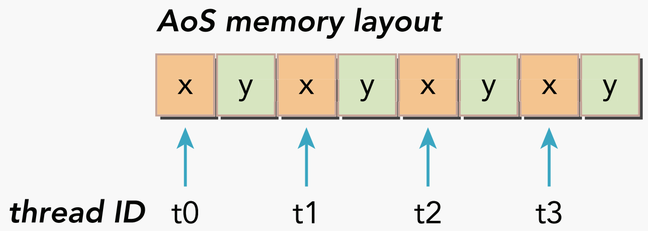
Time your kernels

Offset	SIMULATION TIME (SECONDS)
0	0.003968
12	0.004011
33	0.004024

Array of Structure versus Structure of Arrays

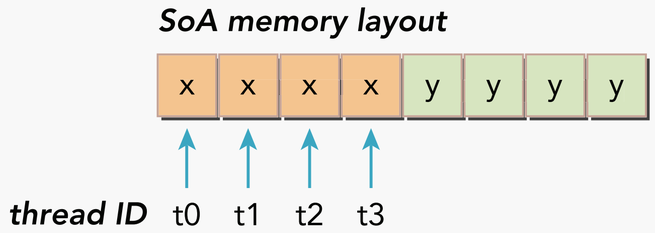
Array of Structures (AOS)

```
struct innerStruct {
  float x;
  float y;
};

struct innerStruct myAoS[N];
```



Structure of Arrays (SOA)

```
struct innerStruct {
  float x[N];
  float y[N];
};

struct innerArray moa;
```



Sample code: EPIC in a predefined electric field

Basic assumptions

Only compute the force from electric field

Neglect magnetic field

Main function

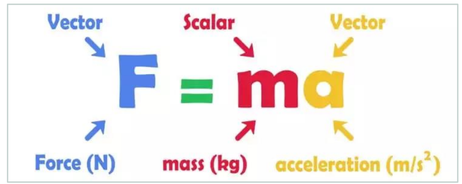
Particle position

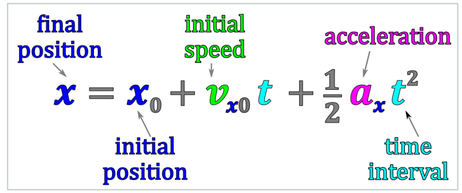
Particle velocity

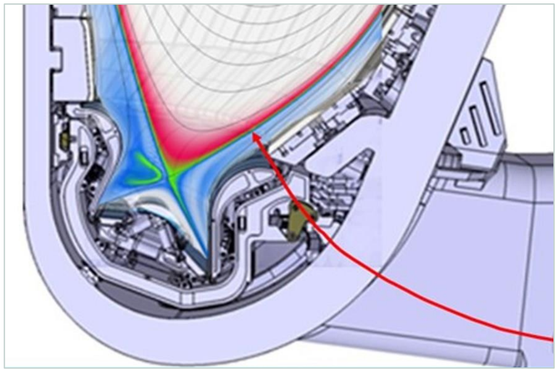
Electric field

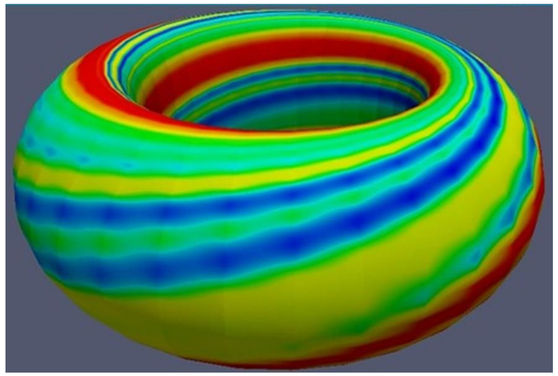
$$\vec{F} = q\vec{E} + q\vec{v} \times \vec{B}$$

Electric force — Magnetic force









CINECA

AOS: EPIC in a predefined electric field

Struct for ParticleList

```
struct ParticleList {

  // An array of particles Structures

   struct Particle* parts;

   // This represents the number of particles in the array

   int n;

};

pl->parts = (struct Particle*)malloc(n * sizeof(struct Particle));


// Set the electric field for each particle
void setE(struct ParticleList* pl, int DIM) {
    for (int i = 0; i < pl->n; ++i) {
        for (int j = 0; j < DIM; ++j) {
            pl->parts[i].E[j] = sin(M_PI * pl->parts[i].pos[j]);
        }
    }
}

// Accelerate particles by updating their velocity
void accel(struct ParticleList* pl, double dt, int DIM) {
    for (int i = 0; i < pl->n; ++i) {
        for (int j = 0; j < DIM; ++j) {
            pl->parts[i].vel[j] += dt * pl->parts[i].q / pl->parts[i].m * pl->parts[i].E[j];
        }
    }
}
```

```
// Move particles by updating their position
void move(struct ParticleList* pl, double dt, int DIM) {
    for (int i = 0; i < pl->n; ++i) {
        for (int j = 0; j < DIM; ++j) {
            pl->parts[i].pos[j] += dt * pl->parts[i].vel[j];
            // Apply periodic boundary conditions
            if (pl->parts[i].pos[j] > 1.0) {
                pl->parts[i].pos[j] -= 1.0;
            }
            if (pl->parts[i].pos[j] < 0.0) {
                pl->parts[i].pos[j] += 1.0;
            }
        }
    }
}
    // Main simulation loop
    int step = 0;
    for (double t = 0; t < 1; t += dt, ++step) {
        nvtxRangePush("Time Step");
        nvtxRangePush("setE");
        setE(&p, DIM);  // Update electric field for all particles
        nvtxRangePop(); //SetE
        nvtxRangePush("accel");
        accel(&p, dt, DIM);  // Update velocities of all particles
        nvtxRangePop(); // Accel
        nvtxRangePush("move");
        move(&p, dt, DIM);  // Update positions of all particles
        nvtxRangePop();
        nvtxRangePop(); // Time Step

        // Save data every ndumps steps
        if (step % ndumps == 0) {
            printData(&p, t, outFile, DIM);  // Save particle data
        }
    }
```

SOA: EPIC in a predefined electric field

Struct for ParticleList

```
struct ParticleList {
    double *pos[MAX_DIM];  // Array of pointers for position
    double *vel[MAX_DIM];  // Array of pointers for velocity
    double *E[MAX_DIM];    // Array of pointers for electric field
    double *q;             // Array for charges
    double *m;             // Array for masses
    int n;                 // Number of particles

};
```

```
for (int i = 0; i < DIM; ++i) {
    pl->pos[i] = (double*)malloc(n * sizeof(double));
    pl->vel[i] = (double*)malloc(n * sizeof(double));
    pl->E[i] = (double*)malloc(n * sizeof(double));
}
pl->q = (double*)malloc(n * sizeof(double));
pl->m = (double*)malloc(n * sizeof(double));
```

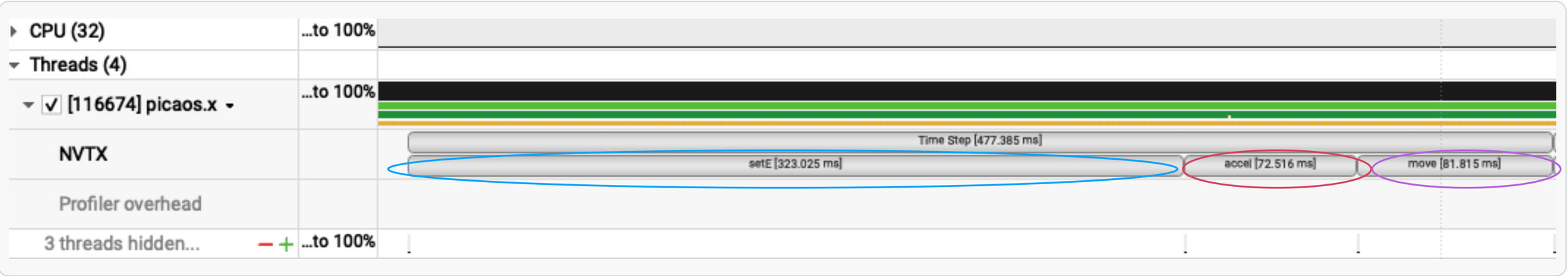
Data access pattern in functions like `setE`, `accel`, and `move`:

```
for (int j = 0; j < DIM; ++j) {
    for (int i = 0; i < pl->n; ++i) {
        pl->E[j][i] = sin(M_PI * pl->pos[j][i]);
    }
}

// Accelerate particles by updating their velocity
void accel(struct ParticleList* pl, double dt, int DIM) {
    for (int j = 0; j < DIM; ++j) {
        for (int i = 0; i < pl->n; ++i) {
            pl->vel[j][i] += dt * pl->q[i] / pl->m[i] * pl->E[j][i];
        }
    }
}

// Move particles by updating their position
void move(struct ParticleList* pl, double dt, int DIM) {
    for (int j = 0; j < DIM; ++j) {
        for (int i = 0; i < pl->n; ++i) {
            pl->pos[j][i] += dt * pl->vel[j][i];
            // Apply periodic boundary conditions
            if (pl->pos[j][i] > 1.0) {
                pl->pos[j][i] -= 1.0;
            }
            if (pl->pos[j][i] < 0.0) {
                pl->pos[j][i] += 1.0;
            }
        }
    }
}
```

Time your kernels

Input parameters
number of Particles = 40000000 dimensions = 2 dt = 0.1 ndumps = 1000

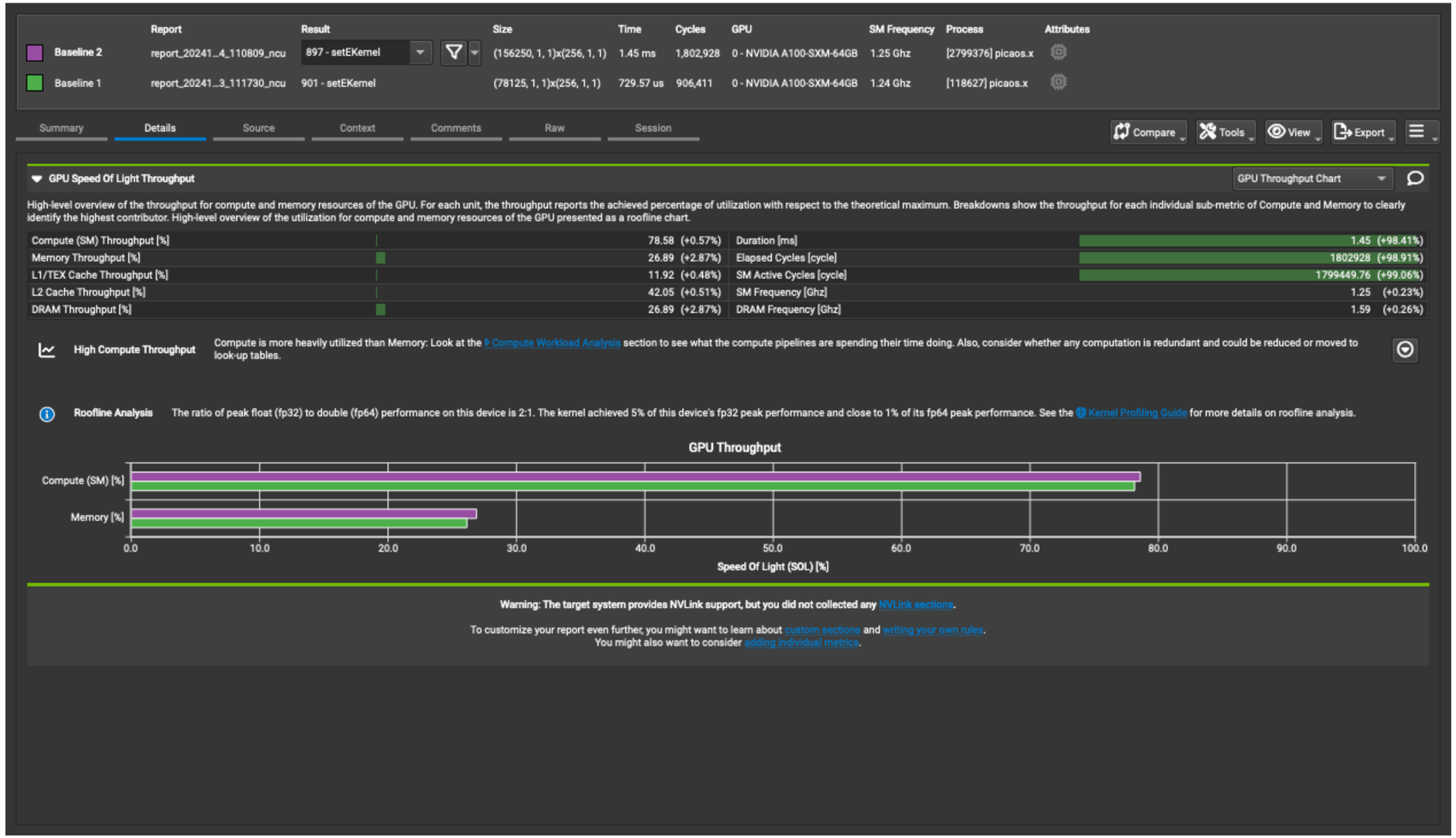
RUNS	SIMULATION TIME (SECONDS)
AOS	38.33
SOA	35. 93



Time your kernels

Runs	N	Kernel Configuration	Elapsed Time on Device
Pageable memory	40000000	(156250, 256)	19.93
Pinned memory	40000000	(156250, 256)	19.21
CudaMallocManaged	40000000	(156250, 256)	19.59

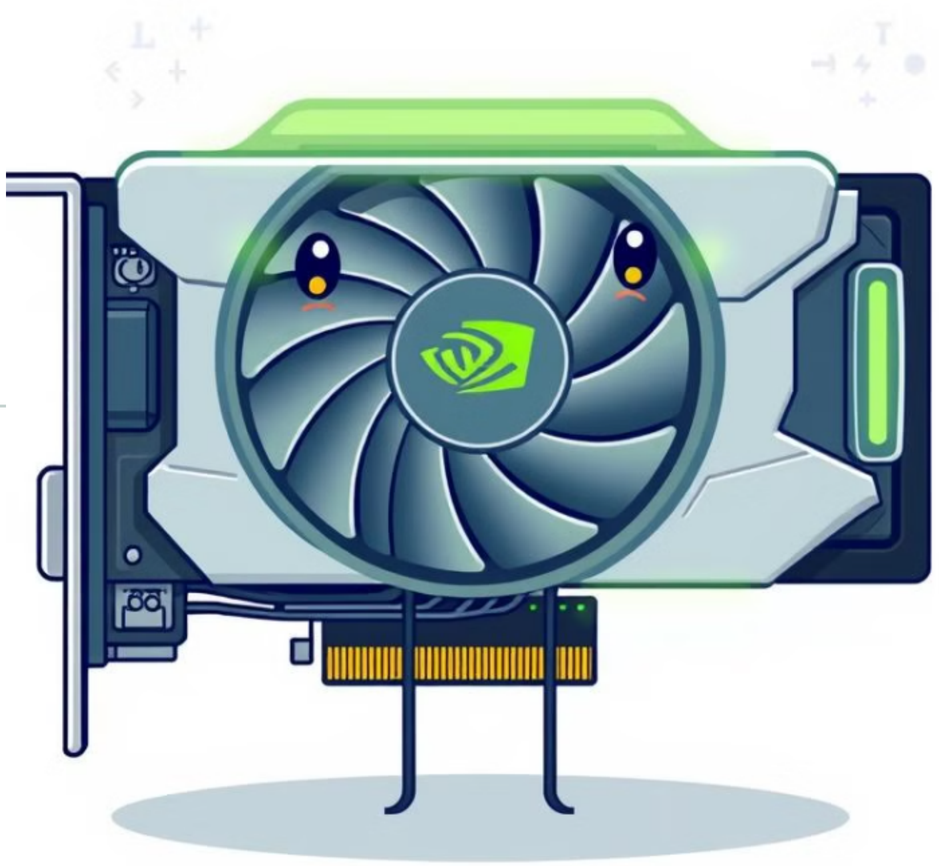
Nsight Compute



Peageable Memory

Unified Memory

5 Optimising memory transfers: cudaMemPrefetchAsync



What is cudaMemPrefetchAsync?

1

cudaMemPrefetchAsync

CUDA function that allows you to explicitly move data to a specific memory location before it is actually needed

2

Supported Platforms

works on both CPU and GPU memory and is supported on NVIDIA GPUs

How to use cudaMemPrefetchAsync?

1

Before Kernel Launch

Call `cudaMemPrefetchAsync` to prefetch data into the cache before the kernel that will use it runs

2

Syntax

```
cudaMemPrefetchAsync(particles.pos, N * DIM * sizeof(float), device_id);
```

When use cudaMemPrefetchAsync?

1. **Memory Bound Kernels**

   most beneficial for kernels that are limited by memory access latency or bandwidth

2. **Irregular Access Patterns**

   particularly useful for workloads with unpredictable or scattered memory access patterns

3. **Asynchronous Execution**

   designed to be used in asynchronous programming models, where data transfers and computations can overlap

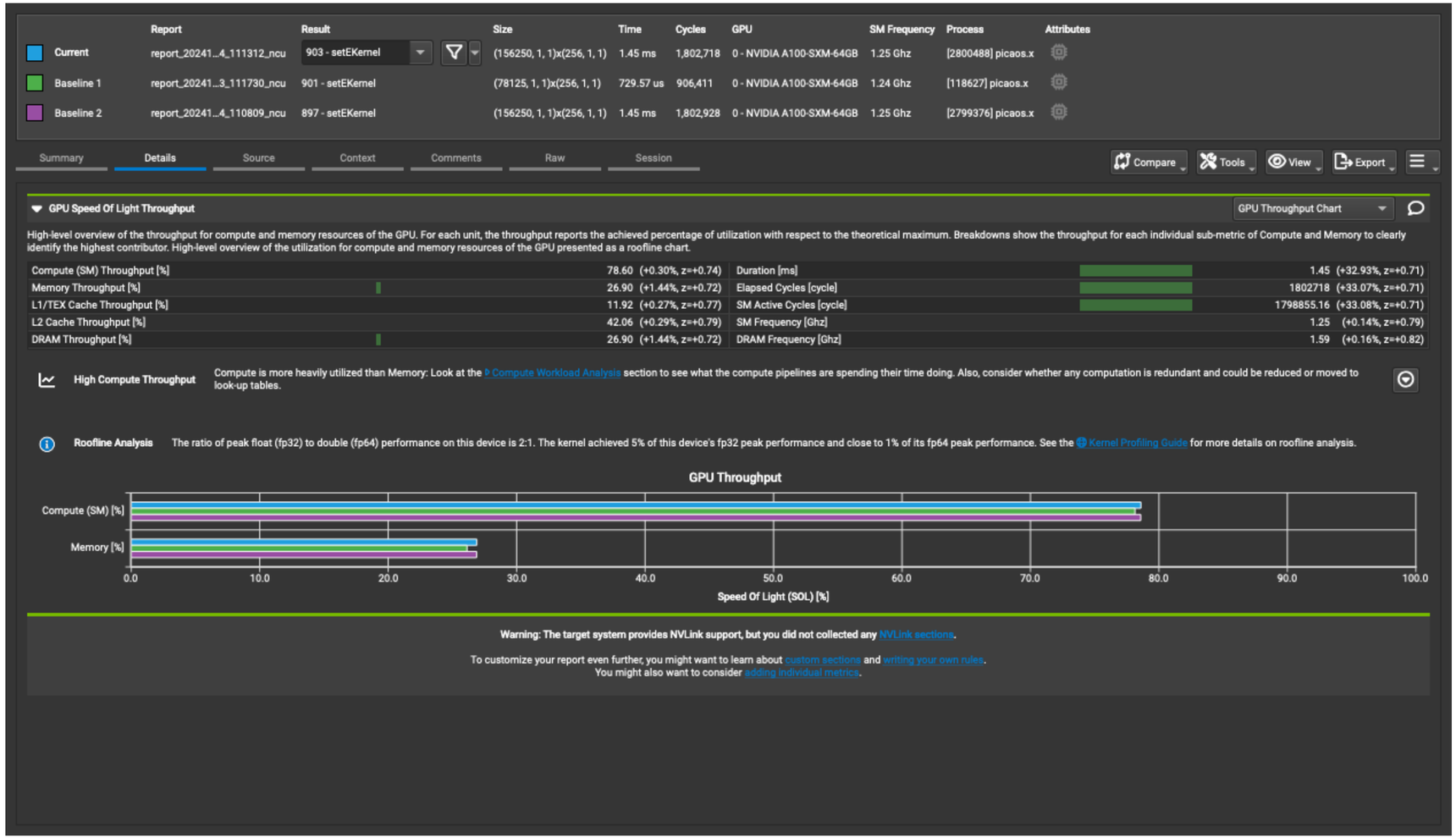
4. **Multi-GPU Environments**

   help optimize data movement between multiple GPUs or between the CPU and GPU

Time your kernels

Runs	N	Kernel Configuration	Elapsed Time on Device
Pageable memory	40000000	(156250, 256)	19.93
Pinned memory	40000000	(156250, 256)	19.21
CudaMallocManaged	40000000	(156250, 256)	19.59
Prefetching	40000000	(524288, 128)	19.33

Nsight Compute

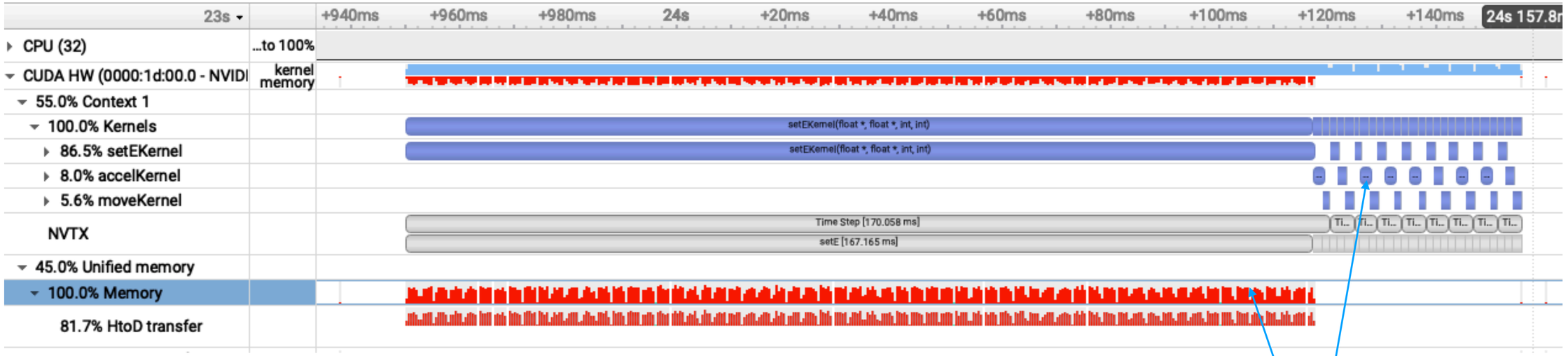


Peageable Memory

Unified Memory

Prefetch Memory

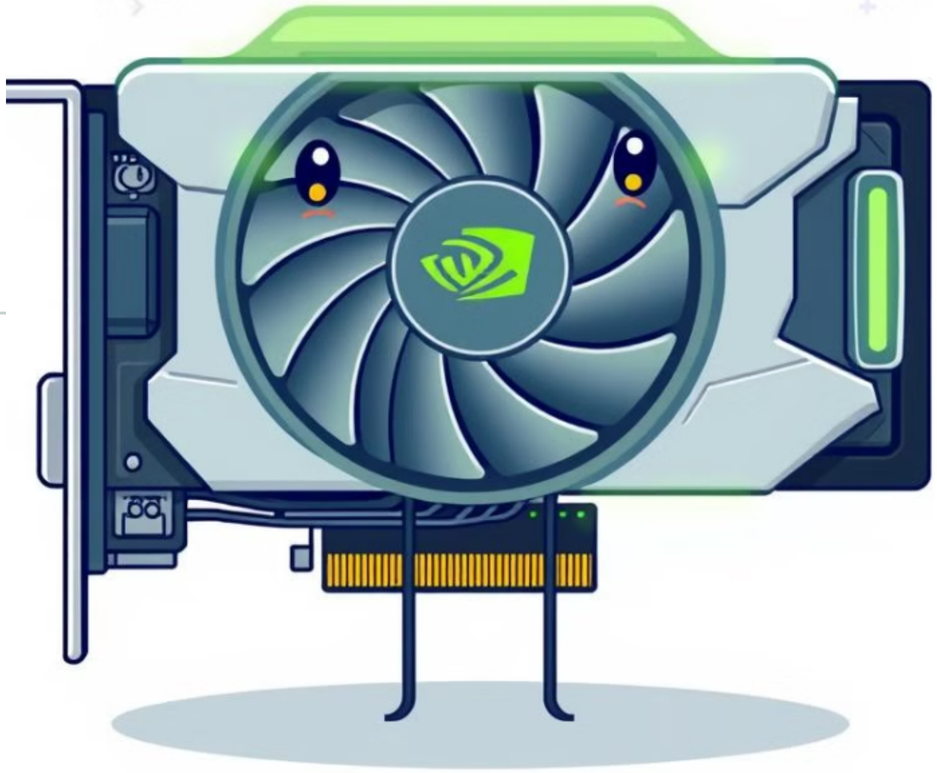
Nsight system report



Look at this pattern

6

How can we overlap kernel and data transfer?



What is a STREAM?

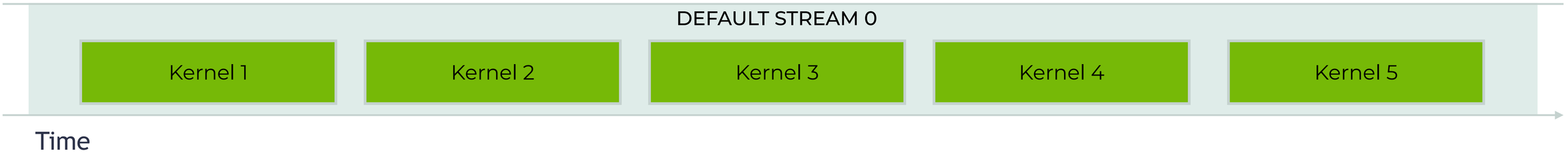
1

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



Non-default Stream behaviour

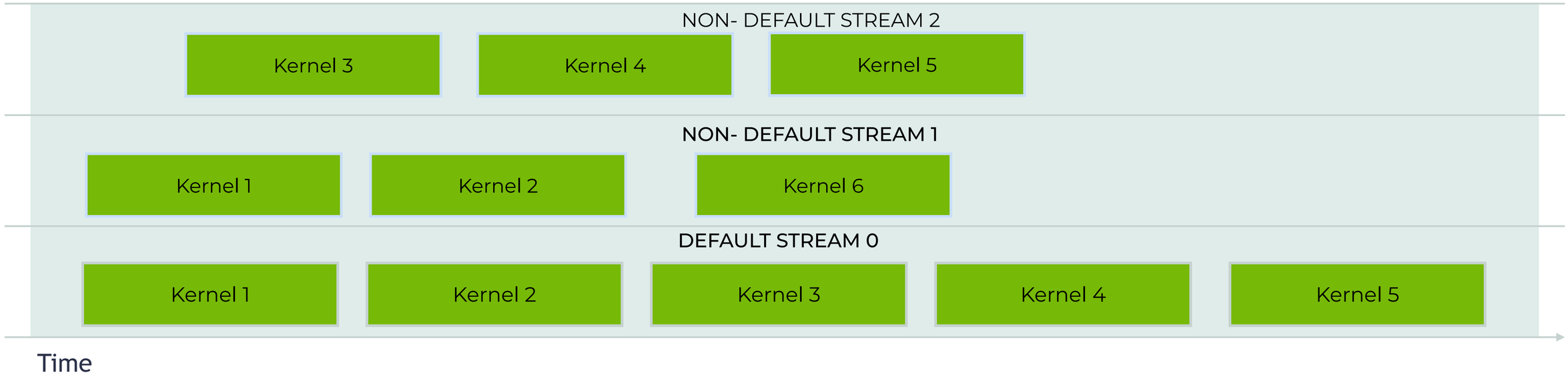
2

Rules of governing the behaviour of streams

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

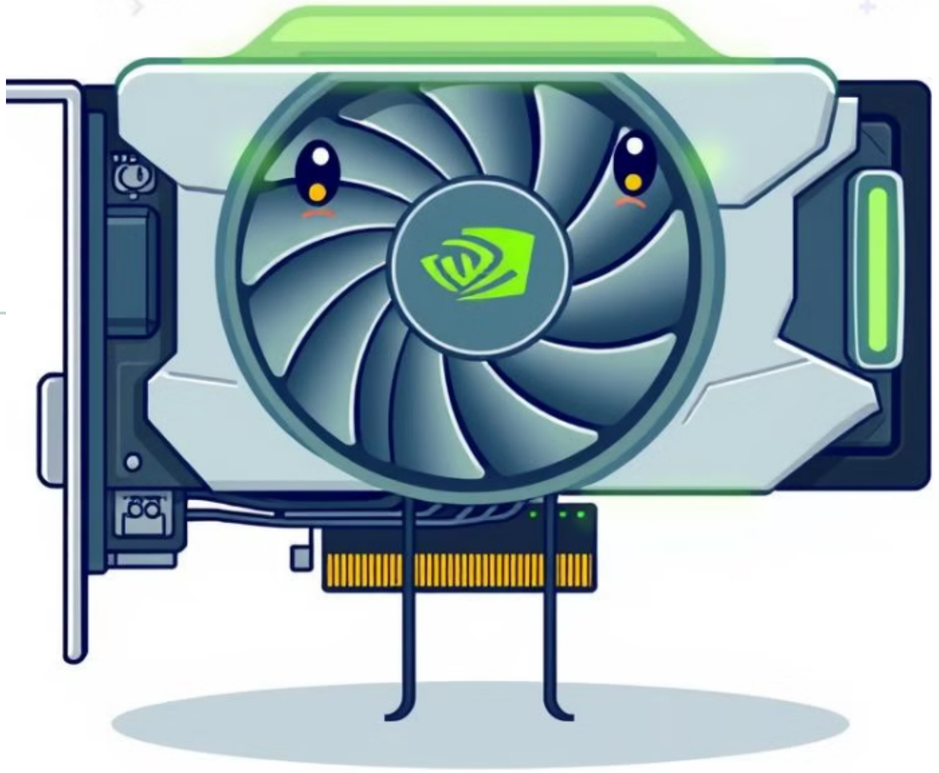
Kernels, with any single STREAM must execute in order

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



7

Understanding CUDA Non-Streams behaviour



Where it can be useful?

1

Kernel Enqueuing

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

2

Memory Transfer

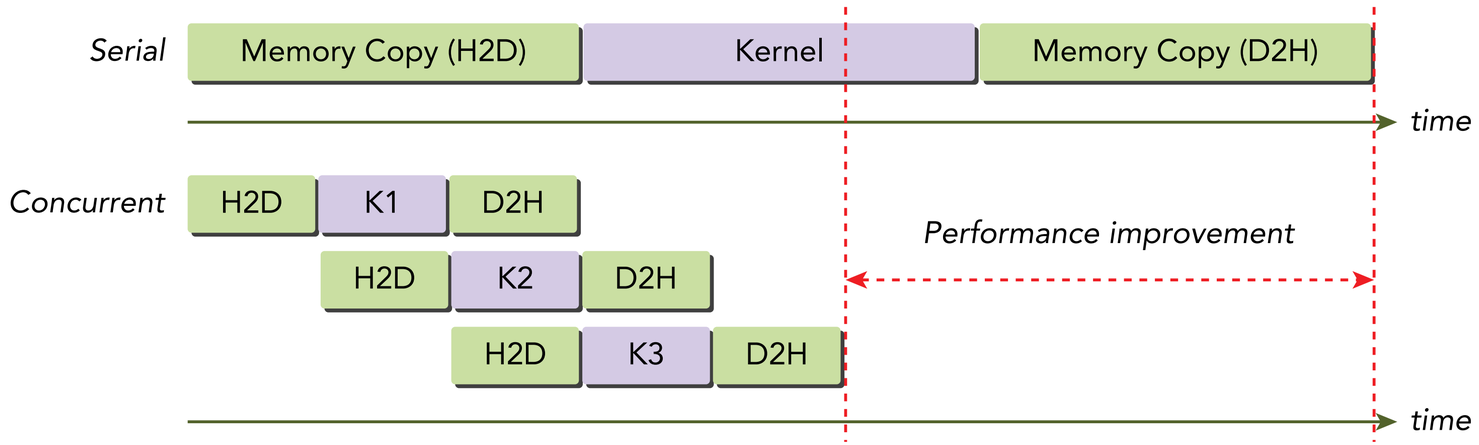
Data transfers between host and device can be enqueued asynchronously into streams.

3

Overlapped Execution

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

Asynchronous Execution with Streams



When use cudaMemPrefetchAsync?

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

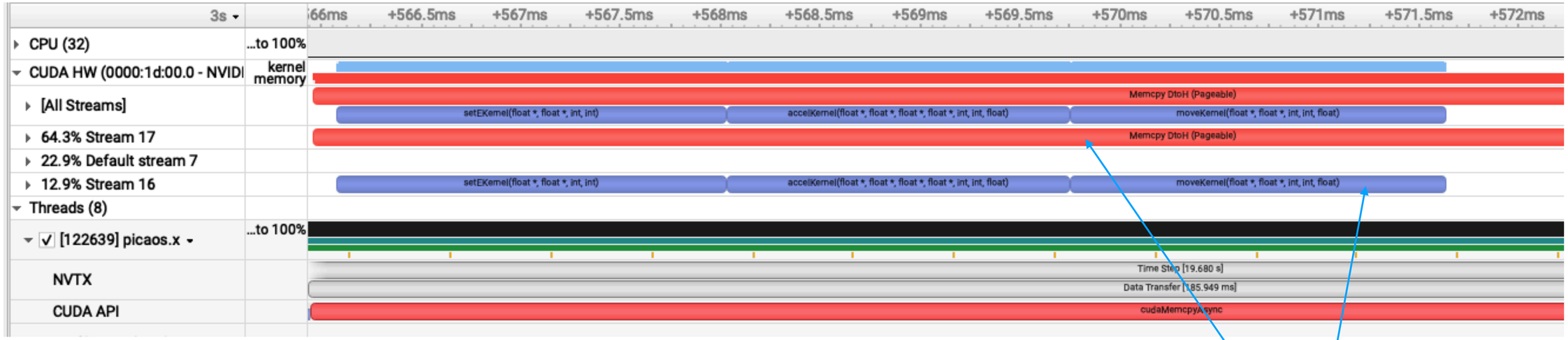
cudaStreamcreate is blocking streams, there is also exists non-blocking streams - But we do not cover it here

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

Nsight system report

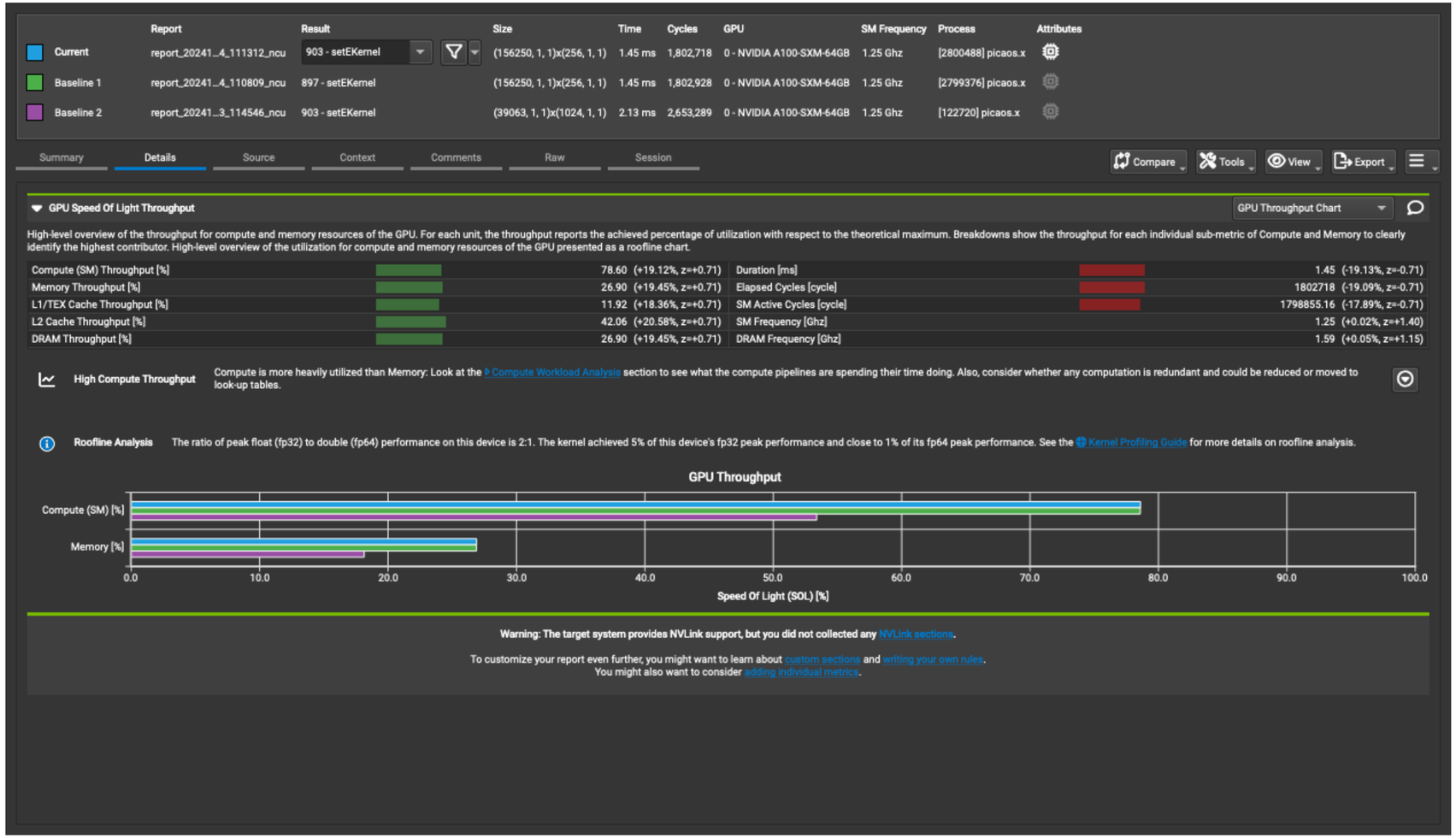


Look at this pattern

Time your kernels

Runs	N	Kernel Configuration	Elapsed Time on Device
Pageable memory	40000000	(156250, 256)	19.93
Pinned memory	40000000	(156250, 256)	19.21
CudaMallocManaged	40000000	(156250, 256)	19.59
Prefetching	40000000	(524288, 128)	19.33
Streams	40000000	(524288, 128)	20.06

Nsight Compute

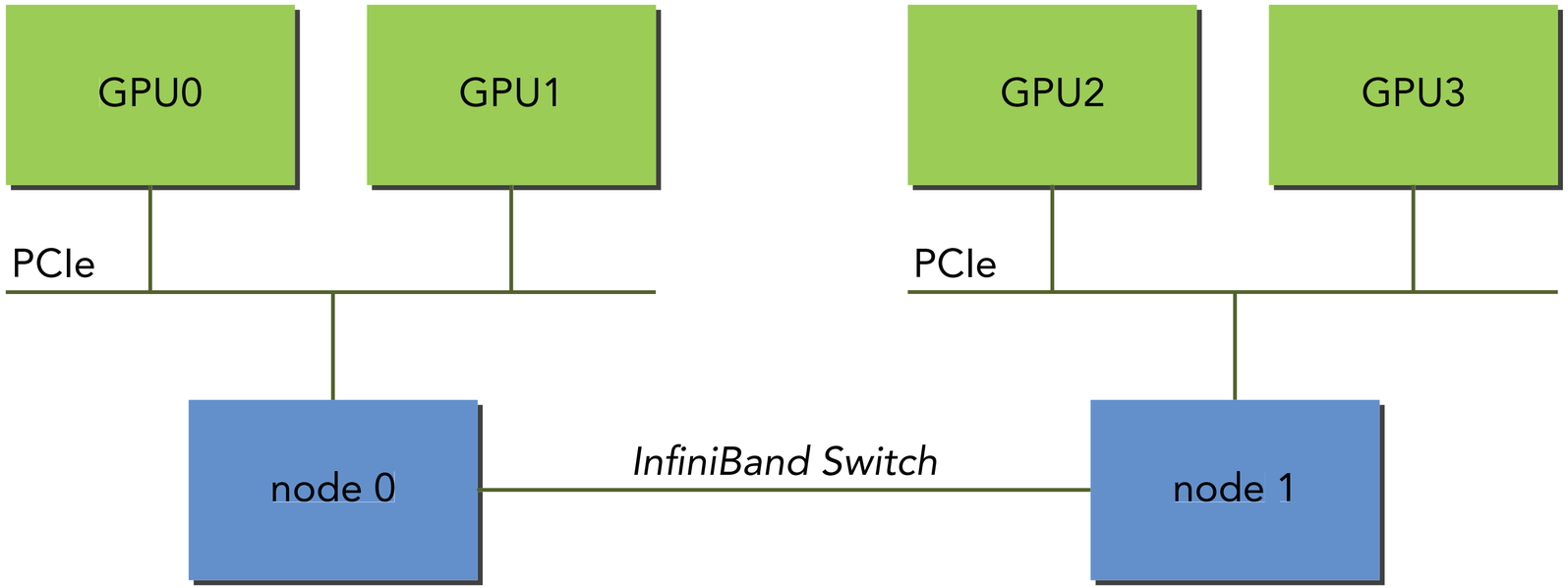


Peageable Memory

Unified Memory

Stream

Multiple streaming-GPU



When use cudaMemPrefetchAsync?

1. Get number of GPUs

   int numGPUs; cudaGetDeviceCount(&numGPUs);

2. Determine the number of particles per GPU

   int particlesPerGPU = N / numGPUs;

3. For each GPU, allocate memory, create streams, and launch kernels

   cudaStream_t streams1[numGPUs], streams2[numGPUs];

4. Final data transfer and synchronization

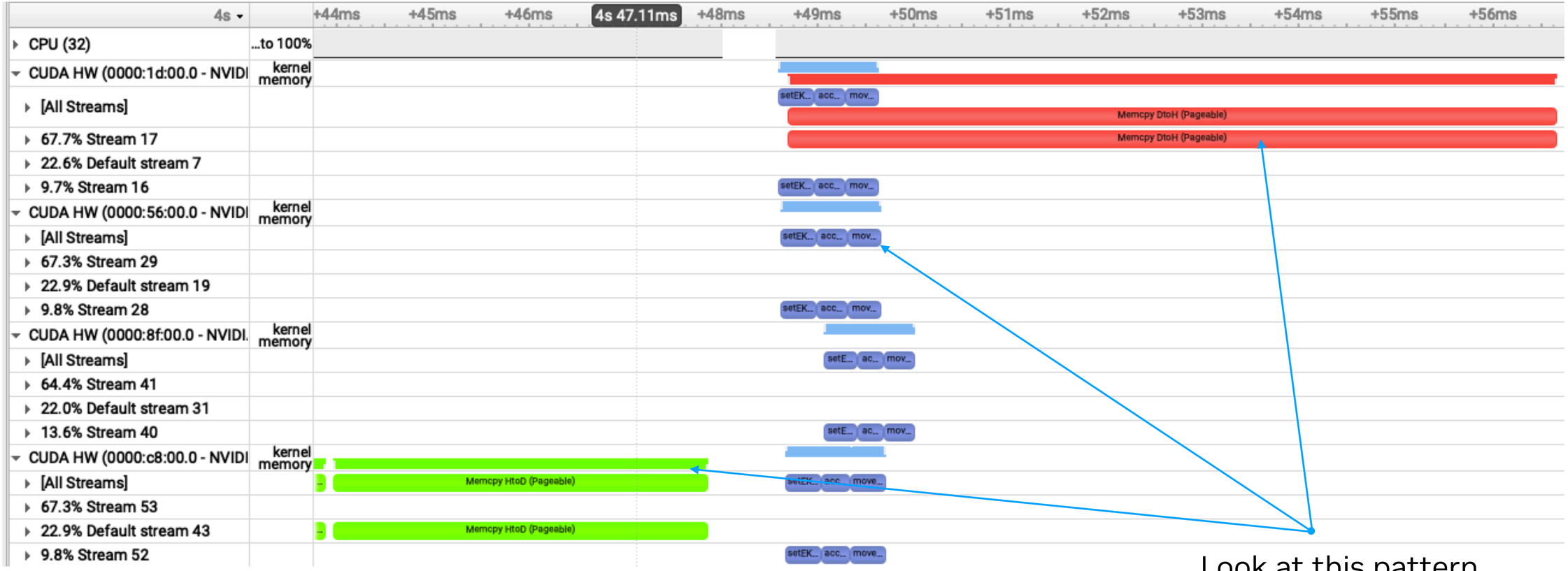
   cudaStreamSynchronize

```
// Non-coalesced access example
for (int gpu = 0; gpu < numGPUs; ++gpu) {
  cudaSetDevice(gpu);  // Set the GPU
  int numBlocks = (particles[gpu].n + BLOCK_SIZE - 1) /
  BLOCK_SIZE;
  setEKernel<<<numBlocks, BLOCK_SIZE, 0,
  streams1[gpu]>>>(particles[gpu].d_pos, particles[gpu].d_E,
  particles[gpu].n, DIM); }
  cudaStreamSynchronize(streams2[gpu]);  // Ensure all data is
  transferred
```

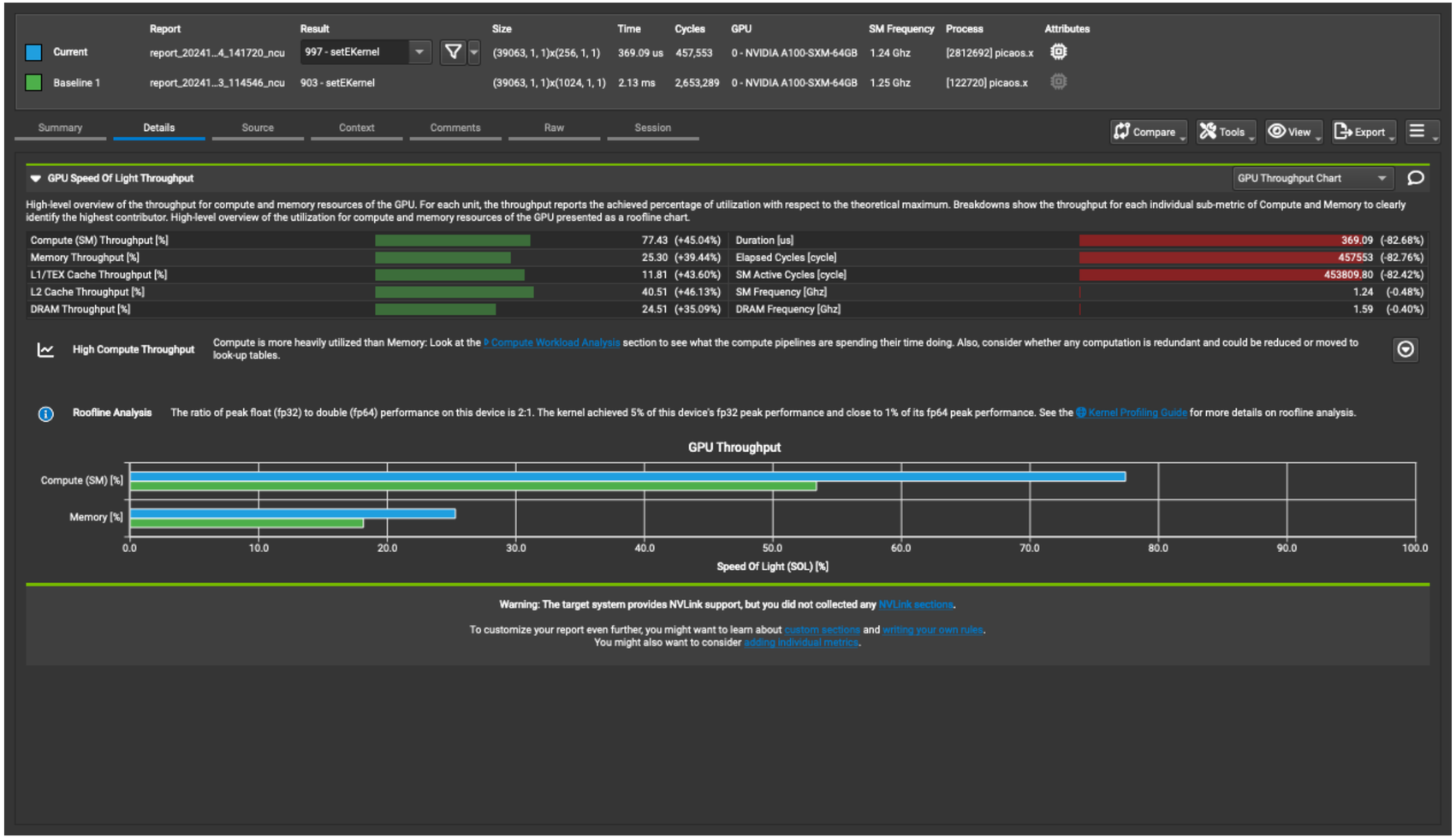
Time your kernels

Runs	N	Kernel Configuration	Elapsed Time on Device
Peag-able memory	40000000	(156250, 256)	19.93
Pinned memory	40000000	(156250, 256)	19.21
CudaMallocManaged	40000000	(156250, 256)	19.59
Prefetching	40000000	(524288, 128)	19.33
Multiple Streams	40000000	(524288, 256)	20.06
Multiple Streams-GPU	40000000	(39063, 256)	20.23

Nsight system report



Nsight system report

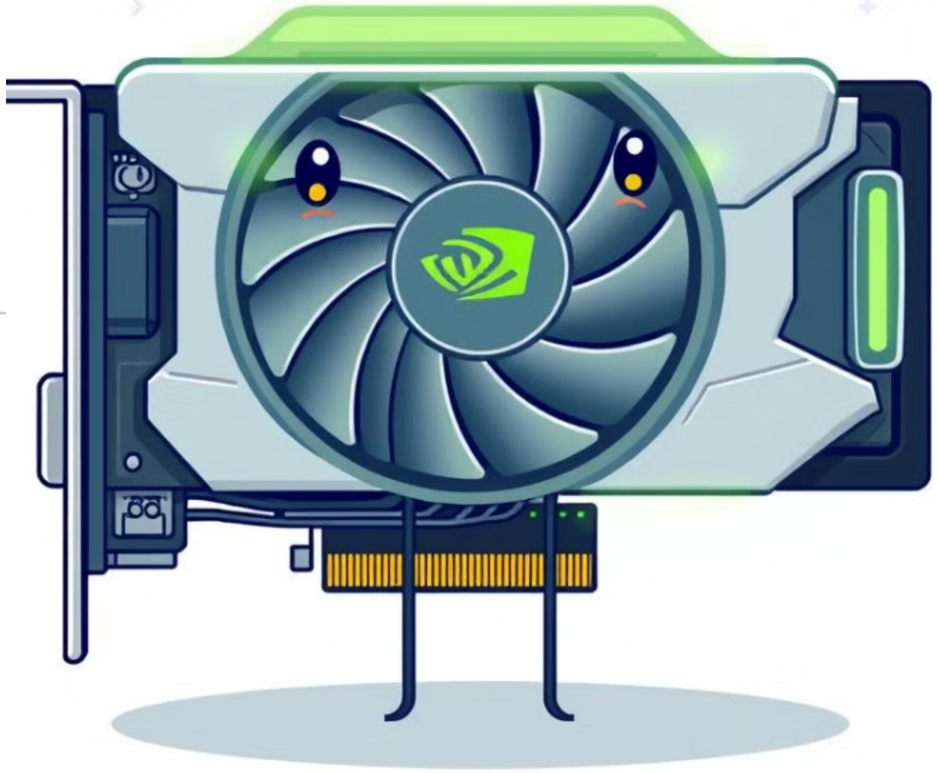


Multiple stream single gpu

Multiple stream-gpu

8

Implementing higher dimensional grid in CUDA

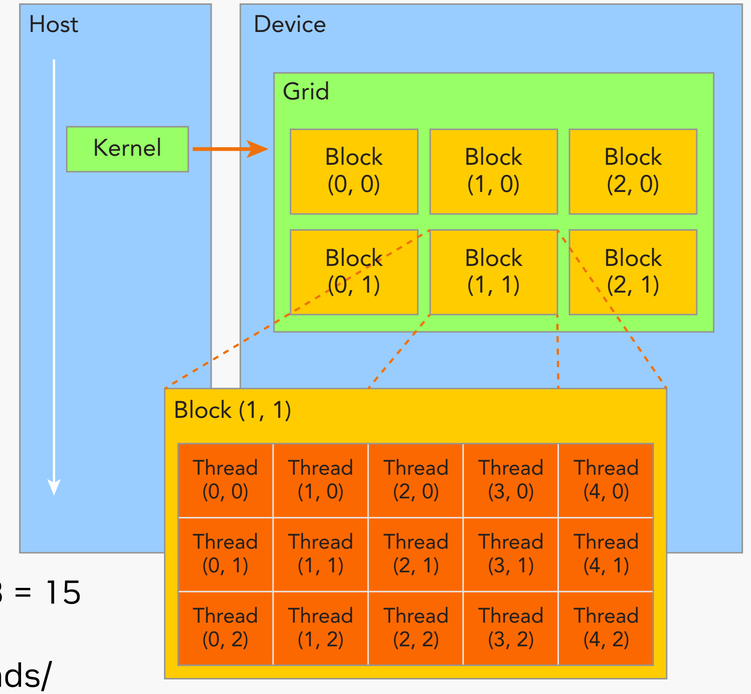


Multidimensional Blocks and Grids

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

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

CUDA compute grid

CUDA compute grid supports 1–3 dimensions

gpu_kernel<<<4,2>>>(...)

gpu_kernel<<<dim3(8, 4 , 1), dim3(4,2,1) >>>(...)

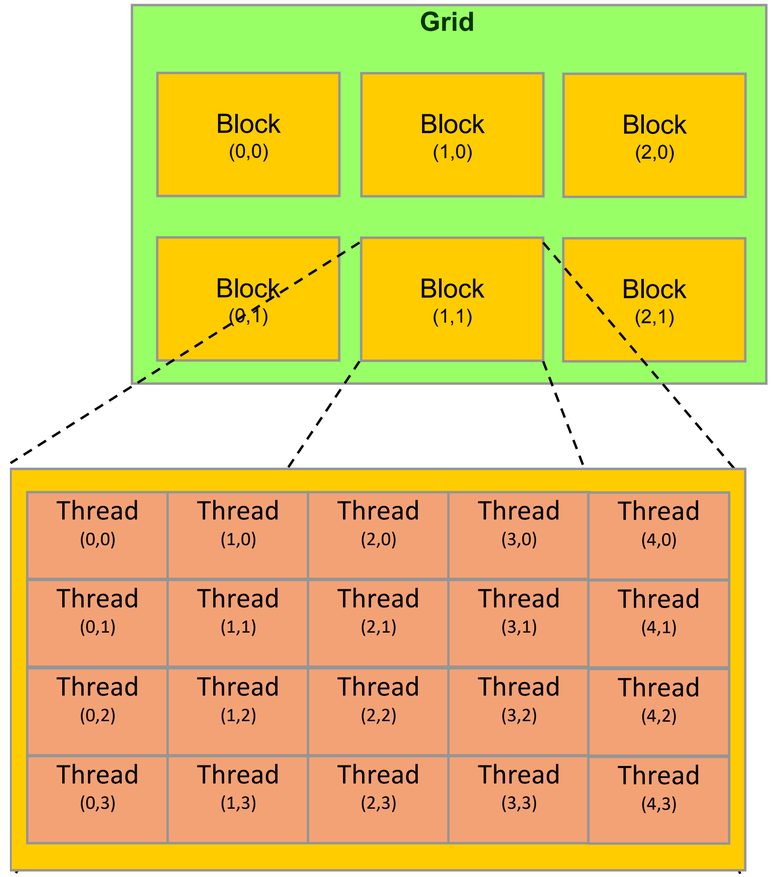
gpu_kernel<<<dim3(16, 8 , 4), dim3(8, 4, 2) >>>(...)

Useful for when

Dealing with multidimensional data

CUDA's dim3 type for both 2D and 3D grids and blocks

CUDA variables: gridDim.x, gridDim.y, gridDim.z, gridBlock.z,...

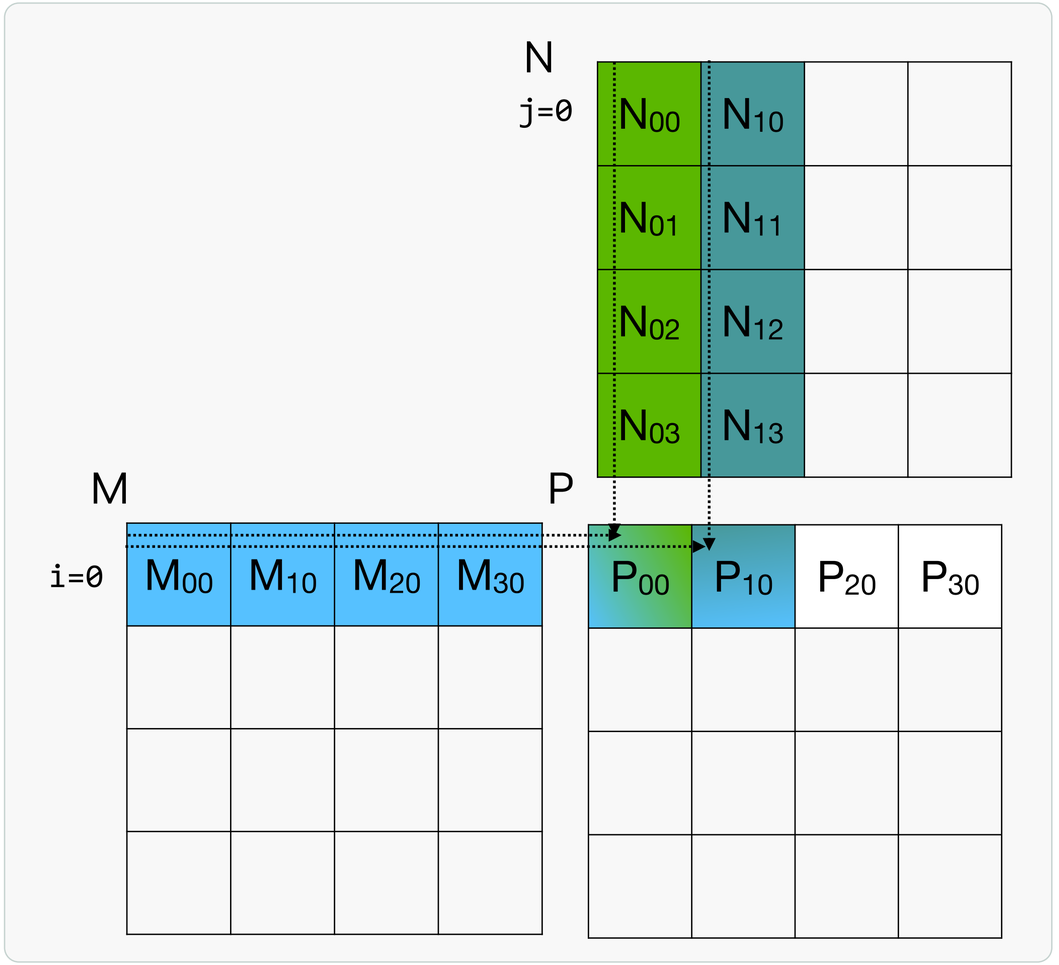


Two matrix multiplication

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

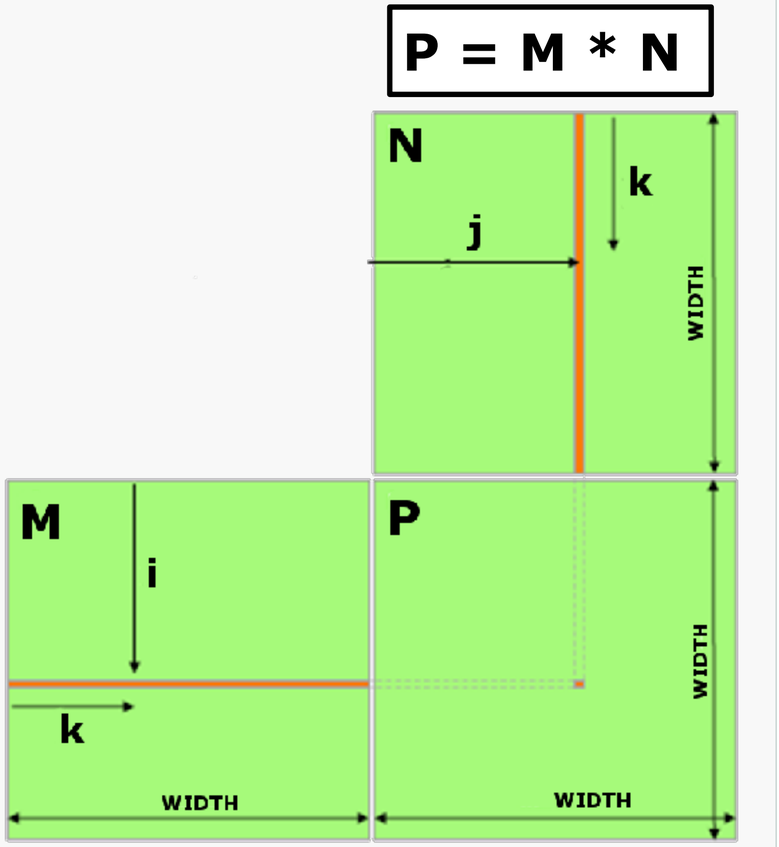
$$P_{10} = M_{00} * N_{10} + M_{10} * N_{11} + M_{20} * N_{12} + + M_{30} * N_{13}$$

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



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;
      }
   }
}
```



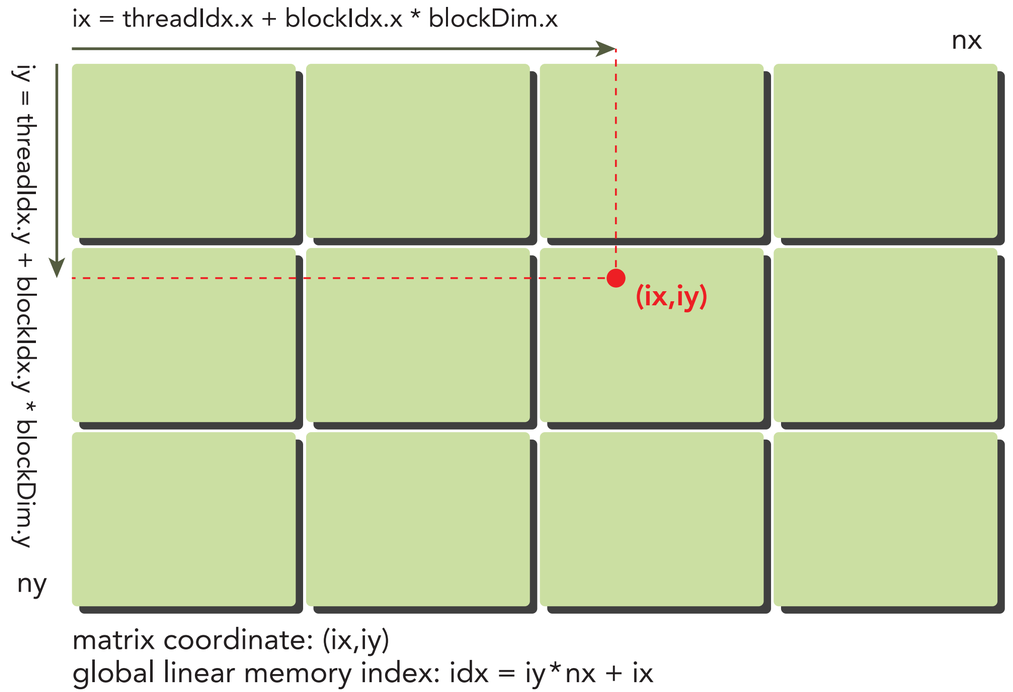
CUDA compute grid supports 1–3 dimensions

2D

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.z;
```

3D

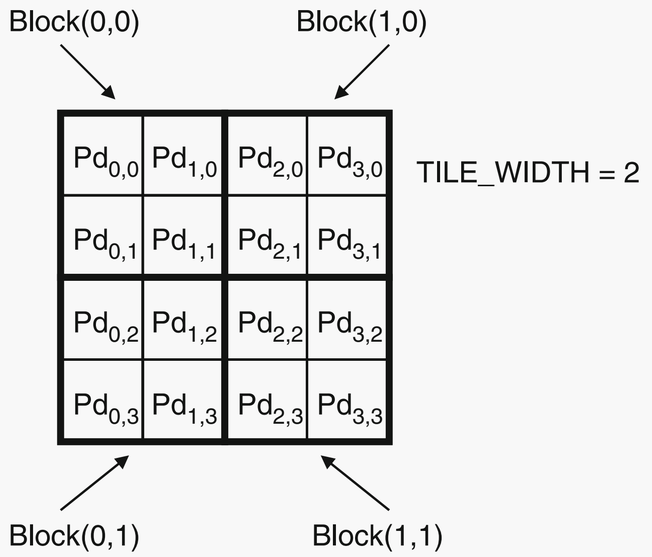
```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.z;
int k = blockIdx.z * blockDim.z + threadIdx.z;
```



- CUDA “hides” loop headers into kernel launch parameters
- Ranges are distributed between threads and blocks of threads
- Blocks number is rounded up to handle the remainder

Two matrix multiplication on GPU

```
// Kernel for matrix multiplication
__global__
void matrixMultiplicationKernel(float* M, float* N, float* Pd, int Width)
{
        int row = blockIdx.y * blockDim.y + threadIdx.y;
        int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < Width && col < Width) {
        float sum = 0;
        for (int k = 0; k < Width; ++k) {
            sum += M[row * Width + k] * N[k * Width + col];
        }
        Pd[row * Width + col] = sum;
    }
}
```

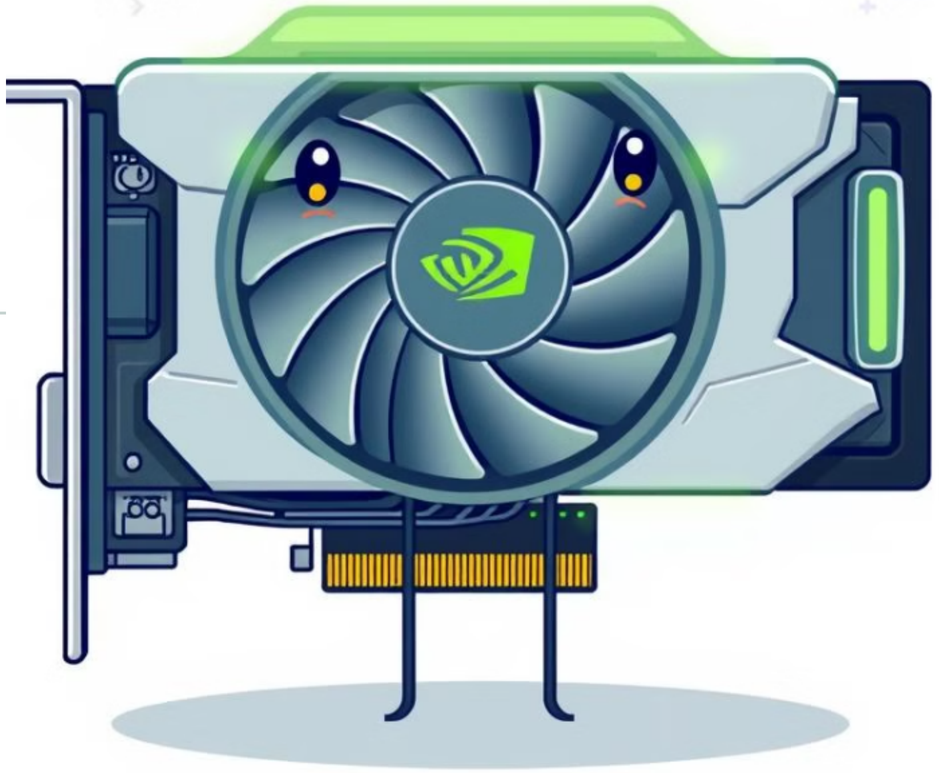


Two matrix multiplication on GPU

N	Methods	Time execution	Speedup
2048x2048	Serial	25.18	1
	CUDA	0.063	398.29

9

Unrolling loops



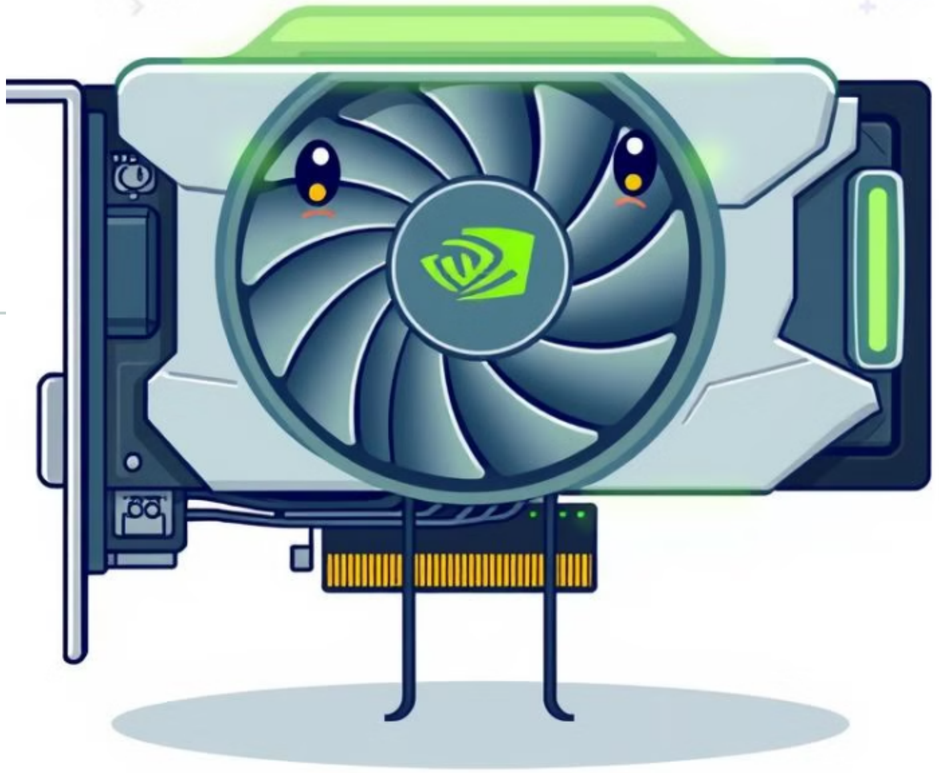
Unrolling loops

```
__global__ void unrolledMatrixMultiplicationKernel(float *A, float *B, float *C, int n, int m, int p) {
    int i = blockIdx.x * blockDim.x + threadIdx.x; // Row index of C
    int j = blockIdx.y * blockDim.y + threadIdx.y; // Column index of C

    if (i < n && j < p) {
        float sum = 0; // Changed to float
        for (int k = 0; k < m - 3; k += 4) {
            sum += A[i * m + k] * B[k * p + j] + A[i * m + k + 1] * B[(k + 1) * p + j] +
                   A[i * m + k + 2] * B[(k + 2) * p + j] + A[i * m + k + 3] * B[(k + 3) * p + j];
        }
        // Handle remaining elements
        for (int k = (m / 4) * 4; k < m; k++) {
            sum += A[i * m + k] * B[k * p + j];
        }
        C[i * p + j] = sum;
    }
}
```

Two matrix multiplication on GPU

N	Methods	Time execution	Speedup
2048x2048	Serial	25.18	1
	CUDA	0.063	398.29
	Unrolled loop	0.055491	453.92

10 What Bandwidth can a kernel achieve?



Theoretical Bandwidth vs. Effective Bandwidth

Theoretical Bandwidth

The absolute maximum bandwidth achievable with the hardware.

Performance Gap

Effective bandwidth is often lower than theoretical bandwidth due to various factors.

Effective Bandwidth

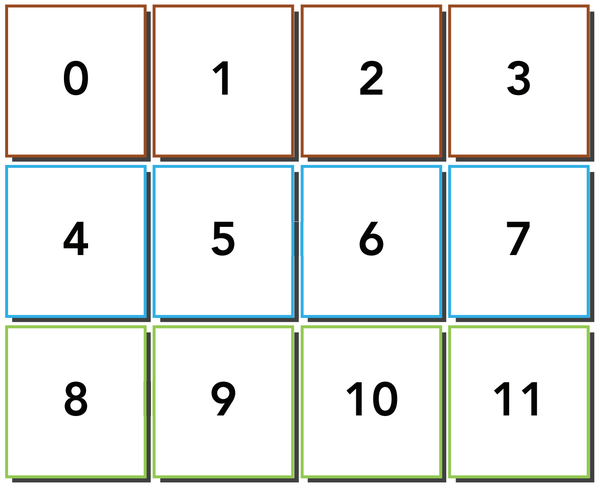
The measured bandwidth that a kernel actually achieves

$$\text{effective bandwidth (GB/s)} = \frac{(\text{bytes read} + \text{bytes written}) \times 10^{-9}}{\text{time elapsed}}$$

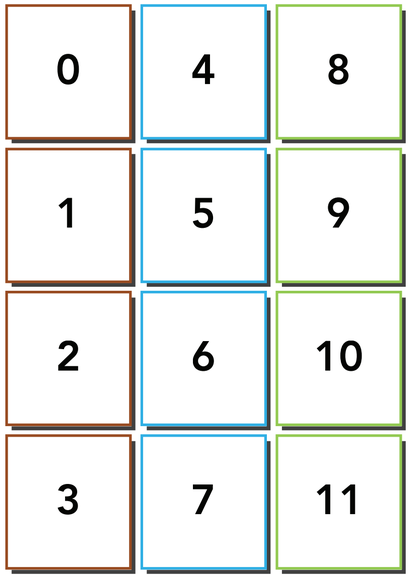
Optimization Importance

Bridging the gap between theoretical and effective bandwidth is a key optimization goal.

Matrix transpose problem

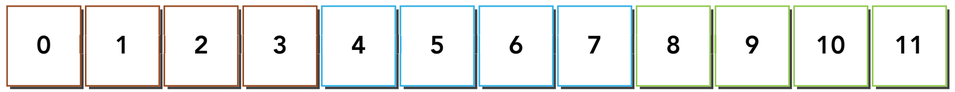



```
void transposeHost(float *out, float *in, const int nx, const int ny) {
    for (int iy = 0; iy < ny; ++iy)  {
    for (int ix = 0; ix < nx; ++ix)  {
     out[ix*ny+iy] = in[iy*nx+ix];
     }
     }
}
```



transposed

data layout of original matrix

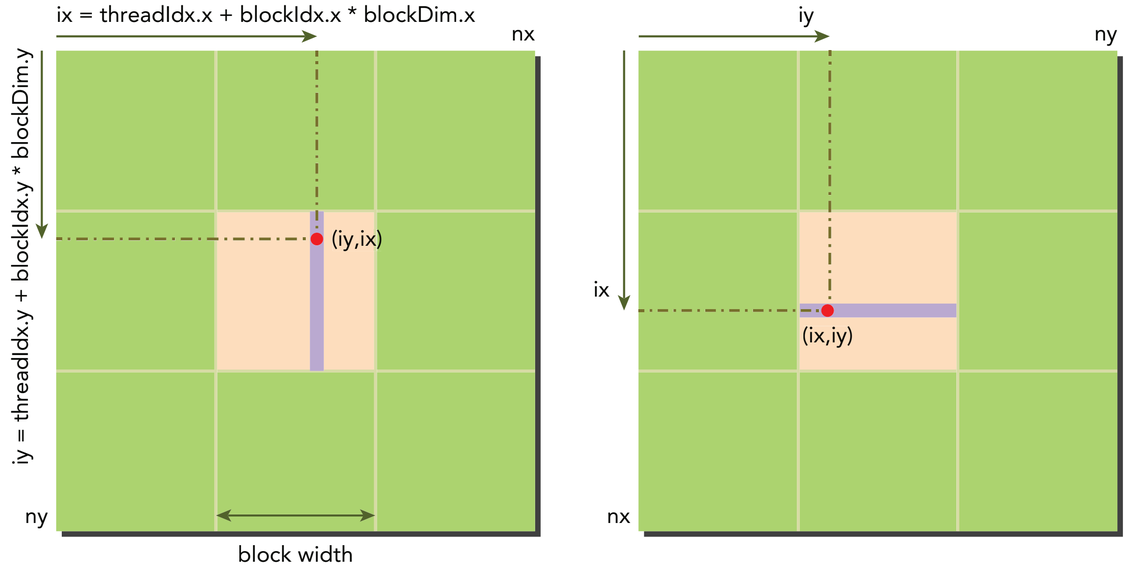


data layout of transposed matrix

CUDD Matrix transpose

```
__global__
void tranposeRow(float *out, float *in, const int nx, const int ny) {
    unsigned int ix = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;

    if (ix < nx && iy < ny) { out[iy*nx + ix] = in[iy*nx + ix];}
}
```

```
__global__
void tranposeCol(float *out, float *in, const int nx, const int ny) {
    unsigned int ix = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;

    if (ix < nx && iy < ny) { out[ix*ny + iy] = in[ix*ny + iy]; }
}
```



Effective Bandwidth of Kernels

BLOCKSIZE	KERNEL	BANDWIDTH [GB/s]	RATIO TO PEAK BANDWITDH (%)
	Theoretical peak bandwidth	900.0	
16 X16	copyRow: Load/store using rows	626.60	69.62
	copyCol: Load/store using cols	275.42	30.60
32X32	copyRow: Load/store using rows	376.32	41.81
	copyCol: Load/store using cols	170.14	18.90

Naive Transpose: Reading Rows versus Reading Columns

```
__global__
void tranposeNRow(float *out, float *in, const int nx, const int ny) {
    unsigned int ix = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;

    if (ix < nx && iy < ny) { out[ix * ny + iy] = in[iy * nx + ix]; }
}
```

```
__global__
void tranposeNCol(float *out, float *in, const int nx, const int ny) {
    unsigned int ix = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;

    if (ix < nx && iy < ny) { out[iy*nx + ix] = in[ix*ny + iy]; }
}
```

BLOCKSIZE	KERNEL	BANDWIDTH [GB/s]	RATIO TO PEAK BANDWITDH (%)
	Theoretical peak bandwidth	900.0	
16 X16	copyRow: Load/store using rows	273.09	30.34
	copyCol: Load/store using rows	296.09	32.90

Unrolling Transpose: Reading Rows versus Reading Columns

```
__global__ void transposeUnroll4Row(float *out, float *in, const int nx,
const int ny) {
    unsigned int ix = blockDim.x * blockIdx.x*4 + threadIdx.x;
    unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;
    unsigned int ti = iy*nx + ix; unsigned int to = ix*ny + iy;

    // access in columns
    if (ix+3*blockDim.x < nx && iy < ny) {
        out[to] = in[ti];
        out[to + ny*blockDim.x] = in[ti+blockDim.x];
        out[to + ny*2*blockDim.x] = in[ti+2*blockDim.x];
        out[to + ny*3*blockDim.x] = in[ti+3*blockDim.x];
    }
}
```

```
__global__ void transposeUnroll4Col(float *out, float *in, const int nx,
const int ny) {
    unsigned int ix = blockDim.x * blockIdx.x*4 + threadIdx.x;
    unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;
    unsigned int ti = iy*nx + ix; unsigned int to = ix*ny + iy;

    // access in columns
    if (ix+3*blockDim.x < nx && iy < ny) {
        out[ti] = in[to];
        out[ti + blockDim.x] = in[to+ blockDim.x*ny];
        out[ti + 2*blockDim.x] = in[to+ 2*blockDim.x*ny];
        out[ti + 3*blockDim.x] = in[to+ 3*blockDim.x*ny];
    }
}
```

Effective Bandwidth of Kernels

BLOCKSIZE	KERNEL	BANDWIDTH [GB/s]	RATIO TO PEAK BANDWITDH (%)
	Theoretical peak bandwidth	900.0	
16 X16	NaiveRow: Load/store using rows	317.29	35.25
	NaiveCol: Load/store using rows	742.74	82.53
32X32	NaiveRow: Load/store using rows	160.73	17.86
	NaiveCol: Load/store using rows	492.21	54.69

Take away message

1

GPU is throughput Horsepower

Offer fast memory access and significant computing power

Importance of compute intensity and memory access patterns

2

Minimize the available data

Wasting bandwidth can severely impact performance

Use structured arrays and maintain proper data order

3

Optimizing Performance

About 75% of issues in code adaptation stem from memory access problems

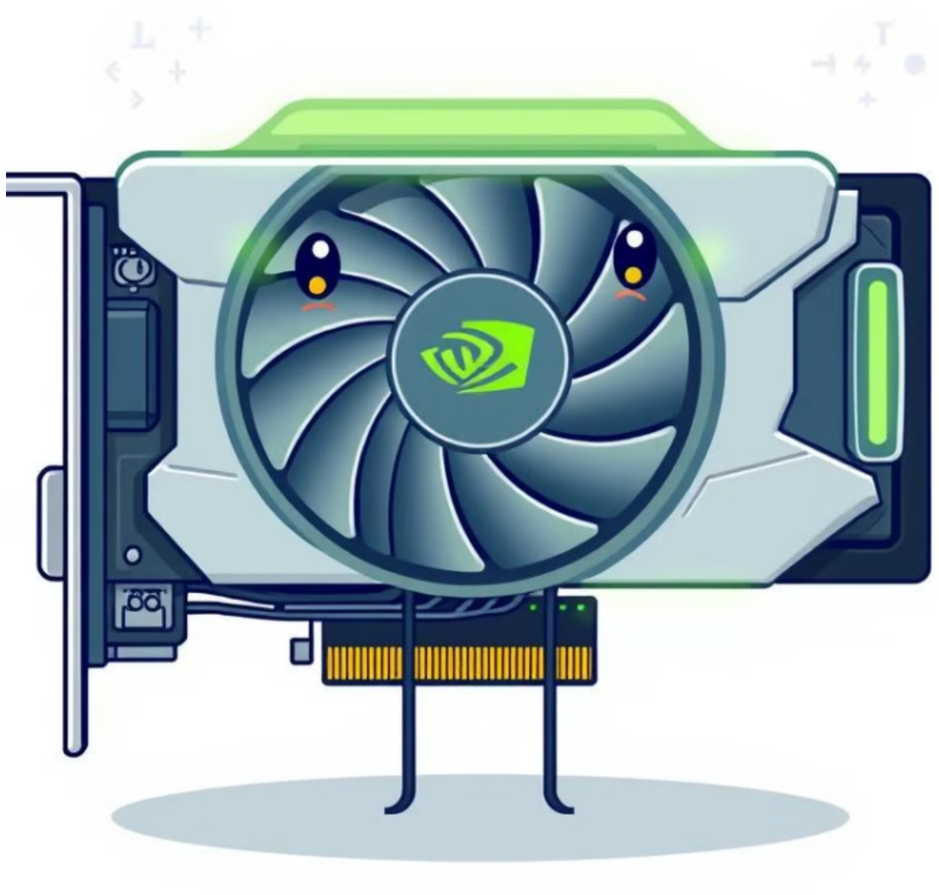
Techniques for improving occupancy and latency hiding

4

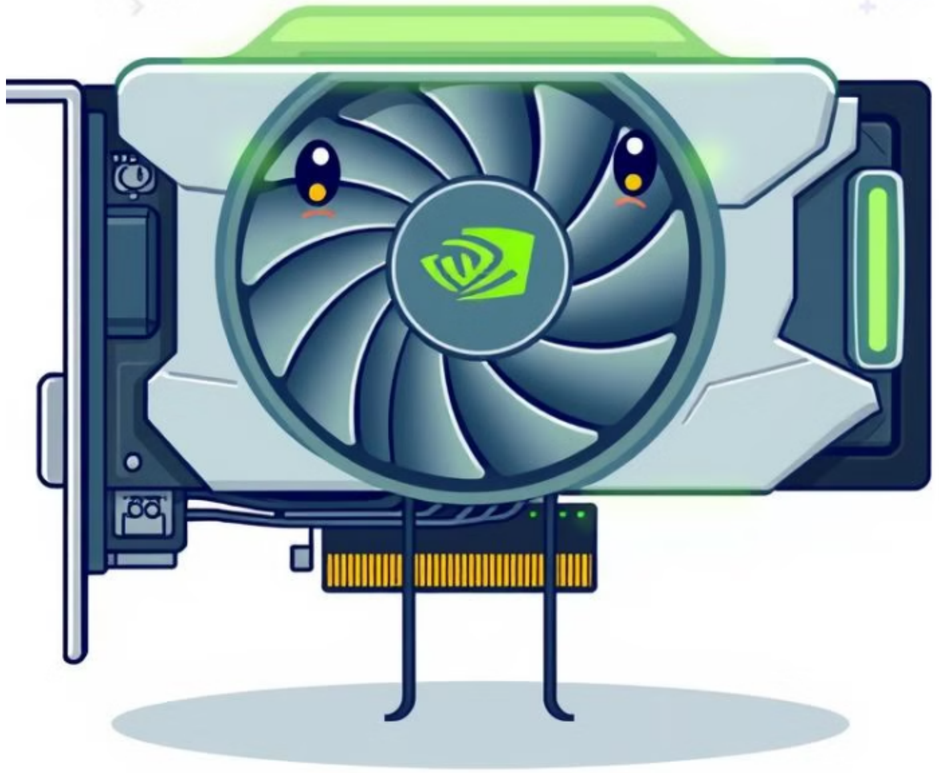
Advanced Techniques

Efficient use of shared memory

Utilizing CUDA streams for concurrent execution



Extra-Slide



GPU Memory Hierarchy

Global Memory

Large, off-chip memory with high latency and lower bandwidth compared to shared memory.

Shared Memory

Small, on-chip memory shared by all threads within a thread block, offering low latency and high bandwidth.

Register Memory

Private memory for each individual thread, with the fastest access but limited capacity.

Shared Memory Basics

Low Latency

Shared memory has much lower access latency compared to global memory, allowing for faster data processing.

High Bandwidth

Shared memory offers significantly higher bandwidth, enabling more efficient data transfer between threads.

Limited Capacity

Shared memory is limited in size, typically ranging from 16KB to 96KB per Streaming Multiprocessor (SM).

Thread Block Scope

Shared memory is shared among all threads within a thread block, allowing for efficient inter-thread communication.

The __shared__ Qualifier

Declaration

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

Scope

Shared memory variables are only accessible to threads within the same thread block

Thread Sync

Threads in a thread block can synchronize using the __syncthreads() intrinsic
Synchronization enables safe data exchange between threads within a block.

Shared memory matrix multiplication kernel

```
__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;
    }
}
```

Two matrix multiplication on GPU

N	Methods	Time execution	Speedup
2048x2048	Serial	25.18	1
	CUDA	0.063	398.29
	Shared memory	0.055491	453.92