E4
COMPUTER
ENGINEERING

Fundamentals of Accelerated Computing with CUDA C/C++

A comprehensive guide to solving the world’s biggest puzzles

Fundamentals of Accelerated Computing with CUDA C/C++

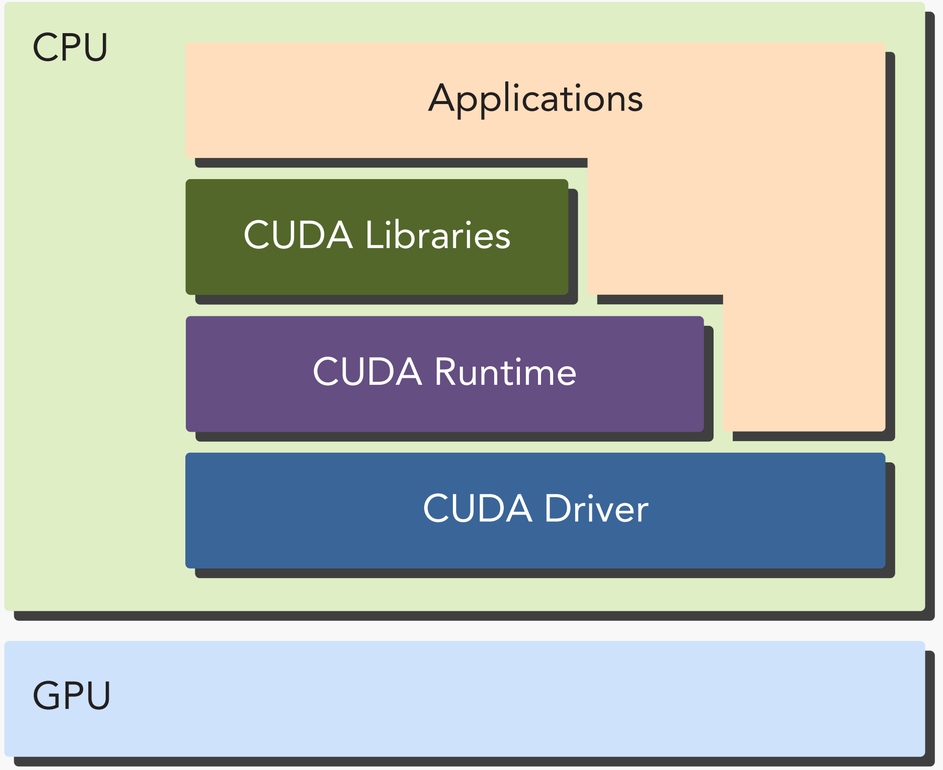
What is CUDA?

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)

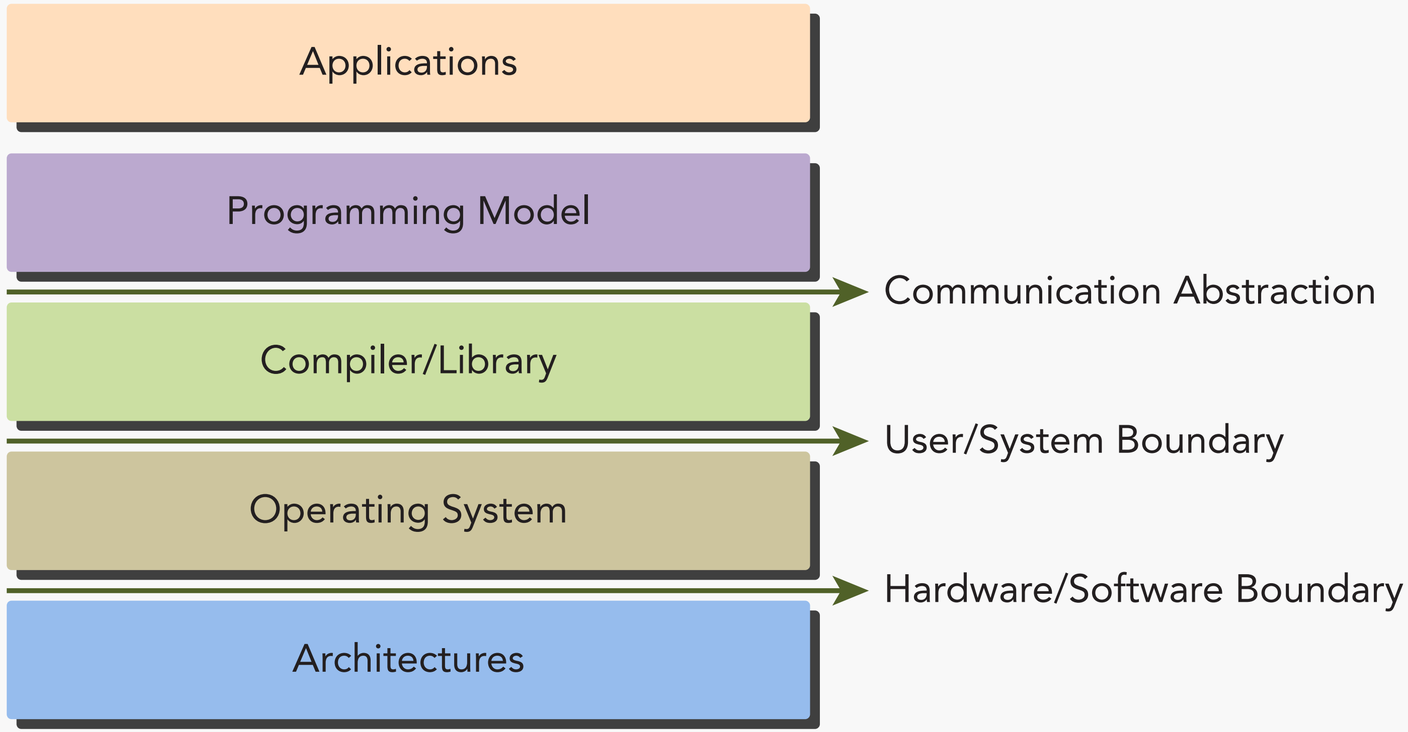
The SDK includes

- 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.JI



CUDA programming model

- Abstraction of computer architectures
- Bridge between app and implementation
- Communication abstraction: program vs. model boundary
- Enabled by compilers/libraries, hardware, and OS
- Program dictates info sharing and coordination
- Offers logical view of computing architectures
- Embodied in languages or environments



Declaring Host-Called, Device-Executed Functions

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

- __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)
- __global__ qualifier for kernels that can be invoked globally

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)
{
   for (int idx=0; idx<N; idx++)
      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);
    ..
}
```

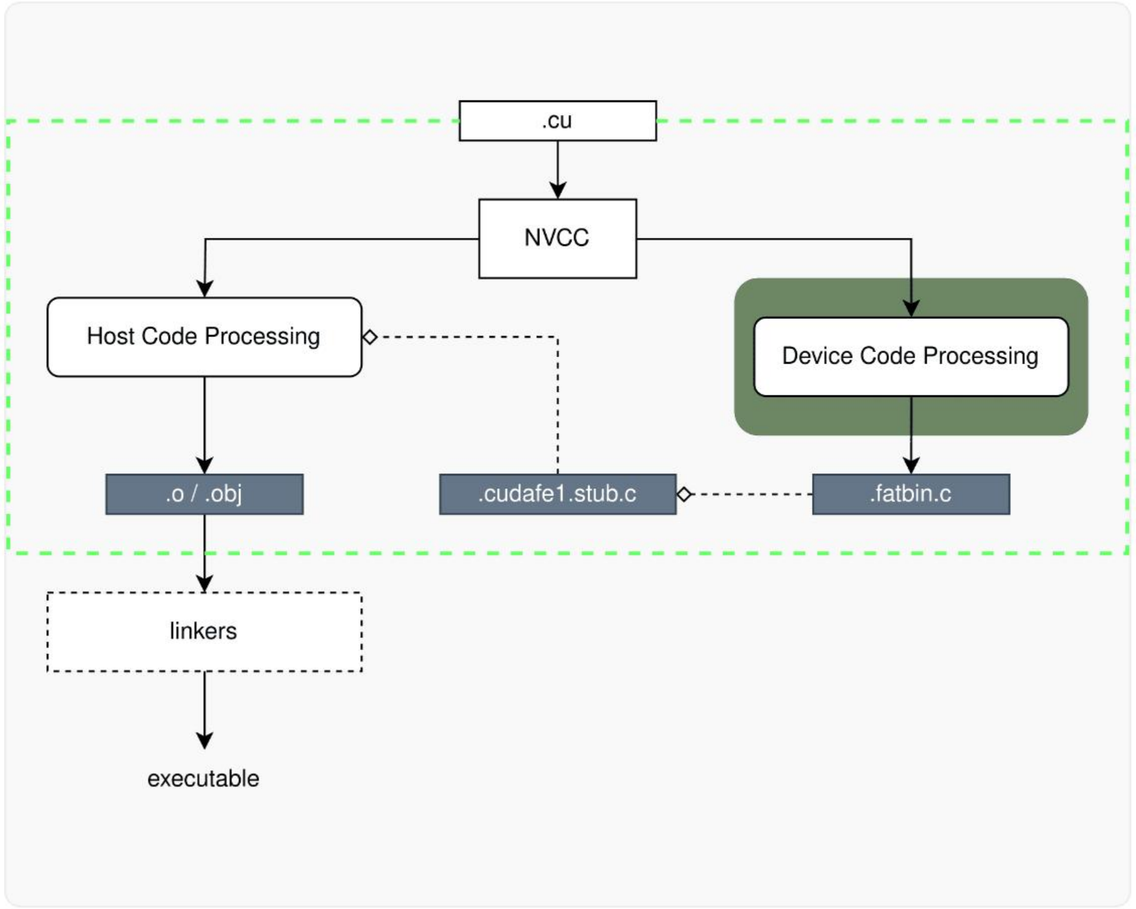
How to compile CUDA enable application?

Source Code Structure

The .cu Extension

CUDA C++ files typically end in `.cu`. This tells the compiler (NVCC) that the file contains both host and device code.

- **Host Code:** Standard C++, compiled by gcc/cl/clang.
- **Device Code:** CUDA kernels, compiled by nvcc to PTX/SASS.



NVHPC compiler

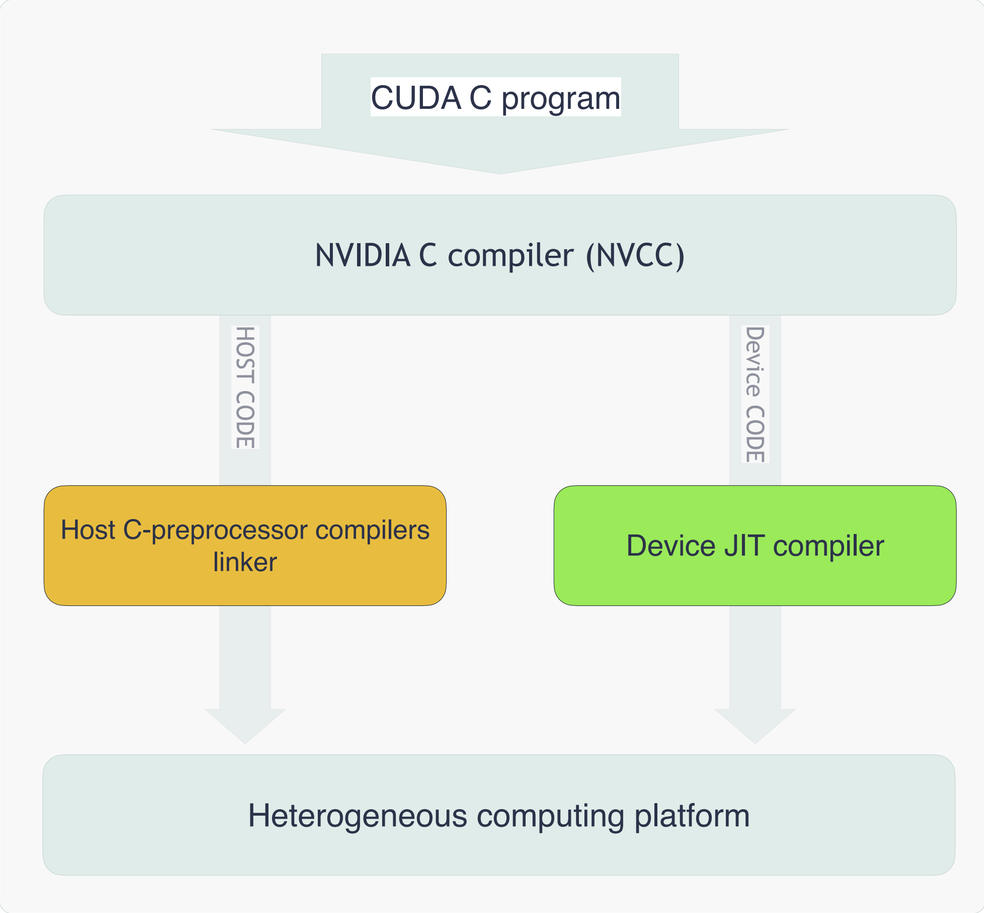
1

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_90 -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

Things to do

- Convert hello.c to hello.cu
- Convert CPU function to GPU function
- Change number of of threads
- Remove Synchronization

Solution

Hello World Kernel

```
__global__ void hello() {
    printf("Hello from GPU!\n");
}

int main() {
    // Launch configuration:
    // 1 Block, 1 Thread
    hello<<<1, 1>>>();

    // Wait for GPU to finish
    cudaDeviceSynchronize();
    return 0;
}
```

Key Takeaways

- **Triple Chevrons:** `<<<...>>>` defines the grid and block dimensions.
- **Async Launch:** Kernels return control to the CPU immediately.
- **Sync Required:** `printf` output buffer is flushed only on synchronization.

Execution & Output

Command

```
$ ./hello
```

Expected Output

```
Hello from CPU!
Hello from thread 0!
Hello from thread 1!
Hello from thread 2!
Hello from thread 3!
Hello from thread 4!
```

Observation

The order of the threads printing is **undefined**. Thread 4 might print before Thread 0.

Why? Parallel execution implies no guaranteed ordering between threads unless we use explicit synchronization barriers.

Common Pitfalls

1. Missing Synchronization

If you forget `cudaDeviceSynchronize()`, the CPU main function might return 0 and exit the program **before** the GPU has even started.

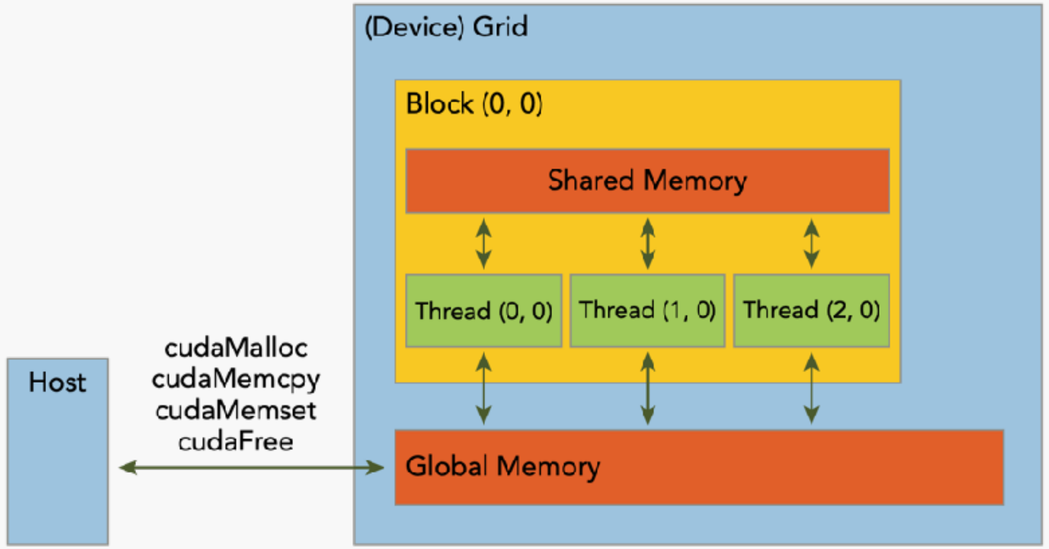
Symptom: "Hello from CPU!" prints, but nothing from GPU.

2. Kernel Launch Failure

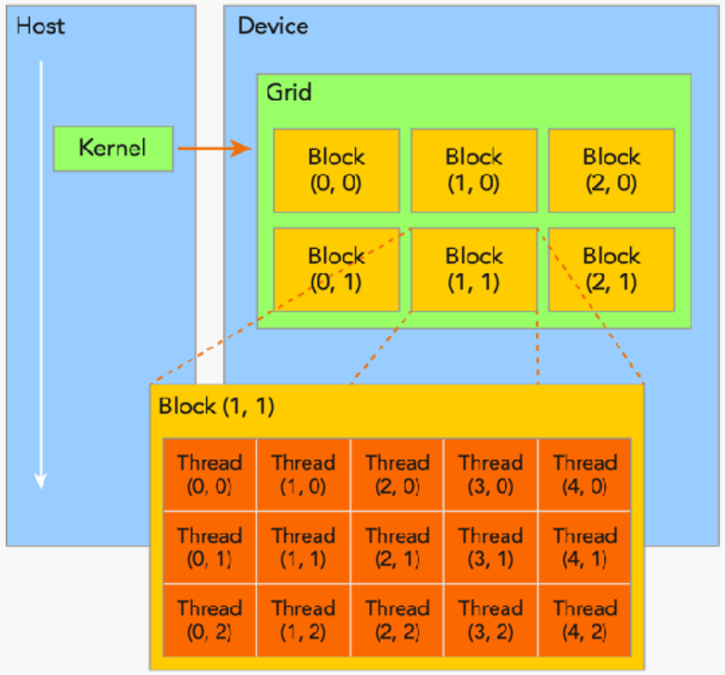
If you launch too many threads (e.g., >1024 per block), the kernel fails silently. Always check for errors (covered in Module 2).

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

Memory hierarchy structure

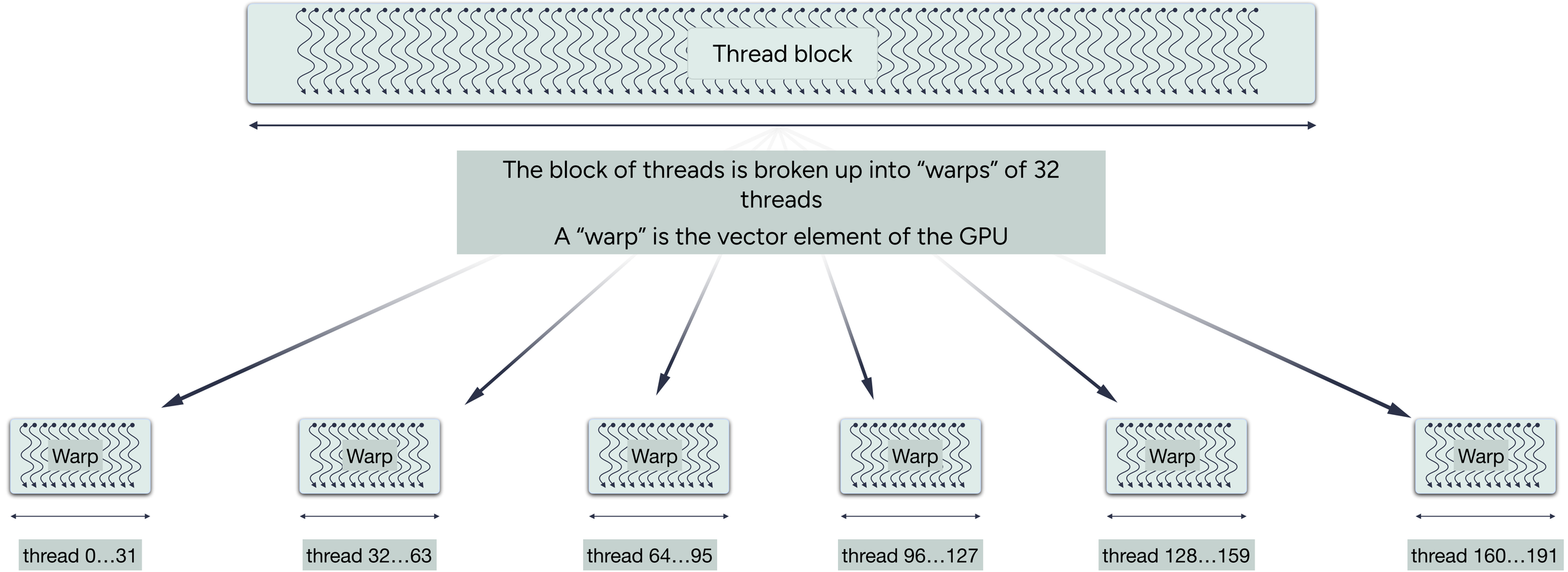


Thread hierarchy structure



Module 1 : GPU Thread hierarchy

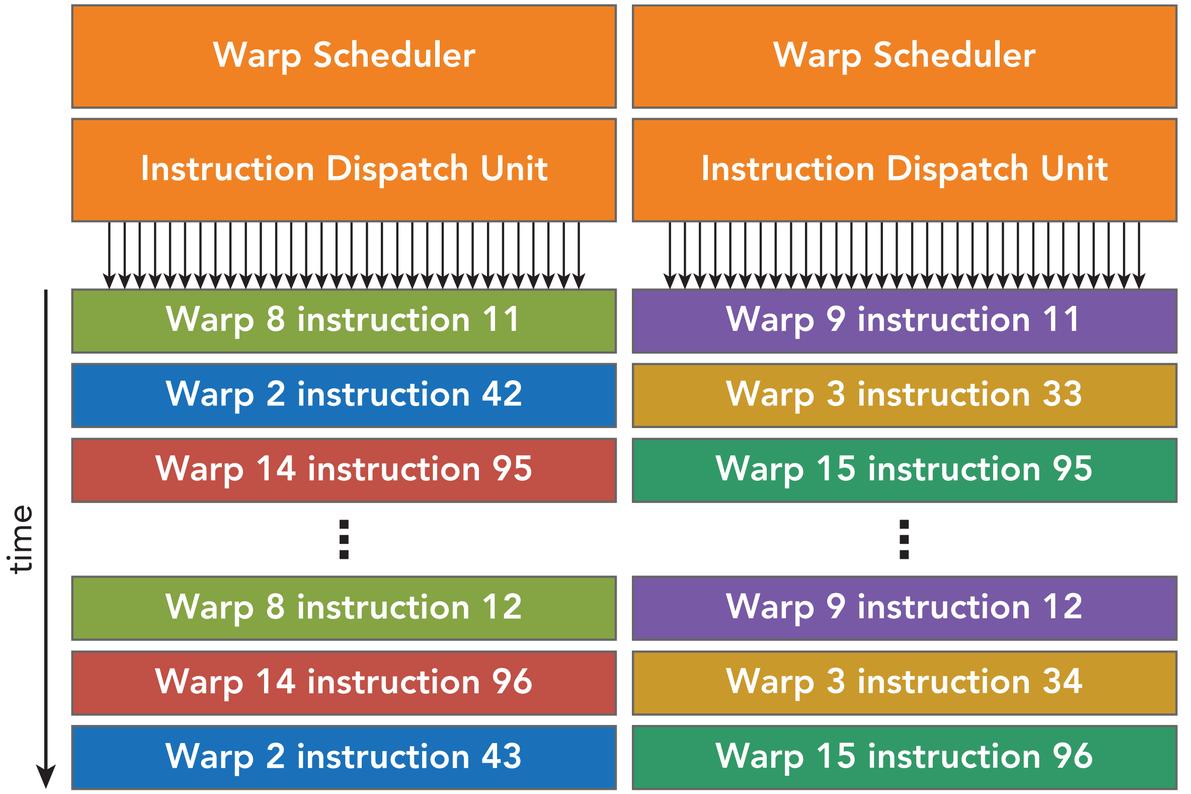
CUDA launches arrays of parallel threads



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



SIMT VS. SIMD execution model

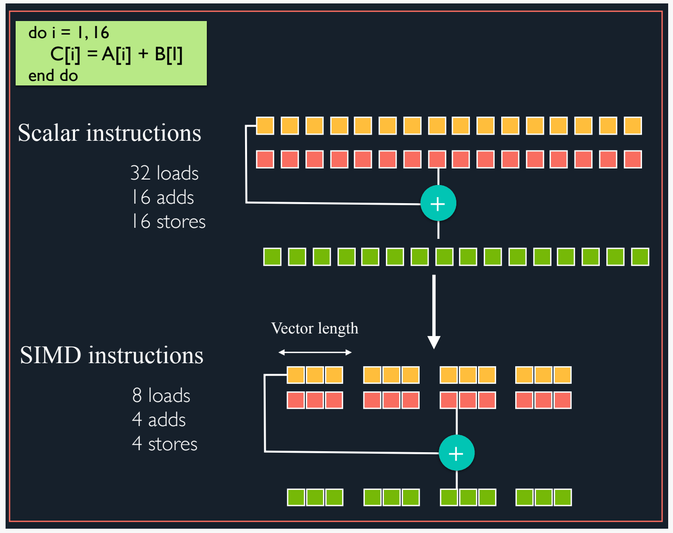
Both SIMD and SIMT achieve parallelism by broadcasting a single instruction to multiple execution units

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?

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

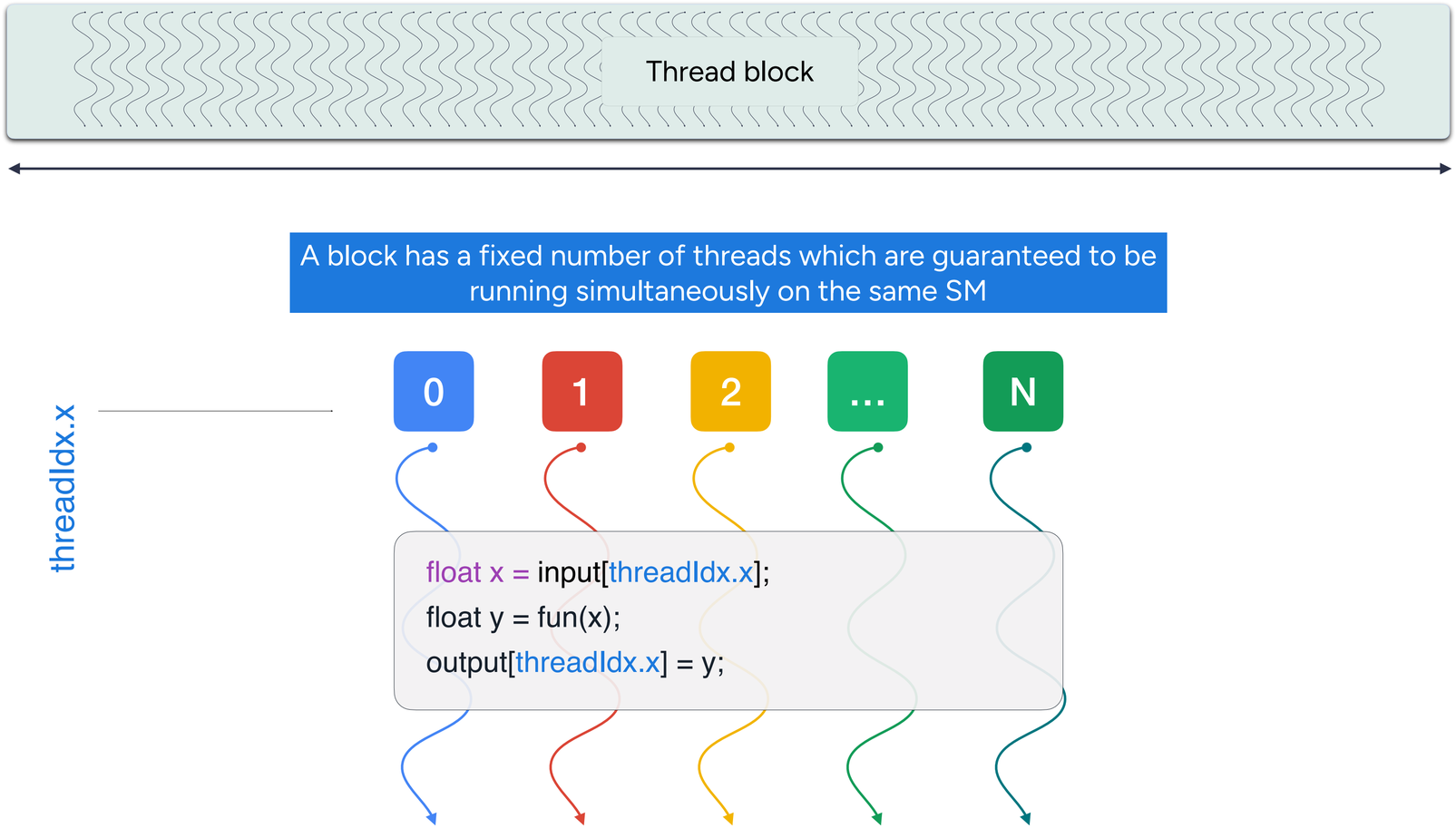
Resource Utilisation

- **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

CUDA launches arrays of parallel threads



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

Exercise -2 : 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);
}
```

Solution

GPU Code

__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

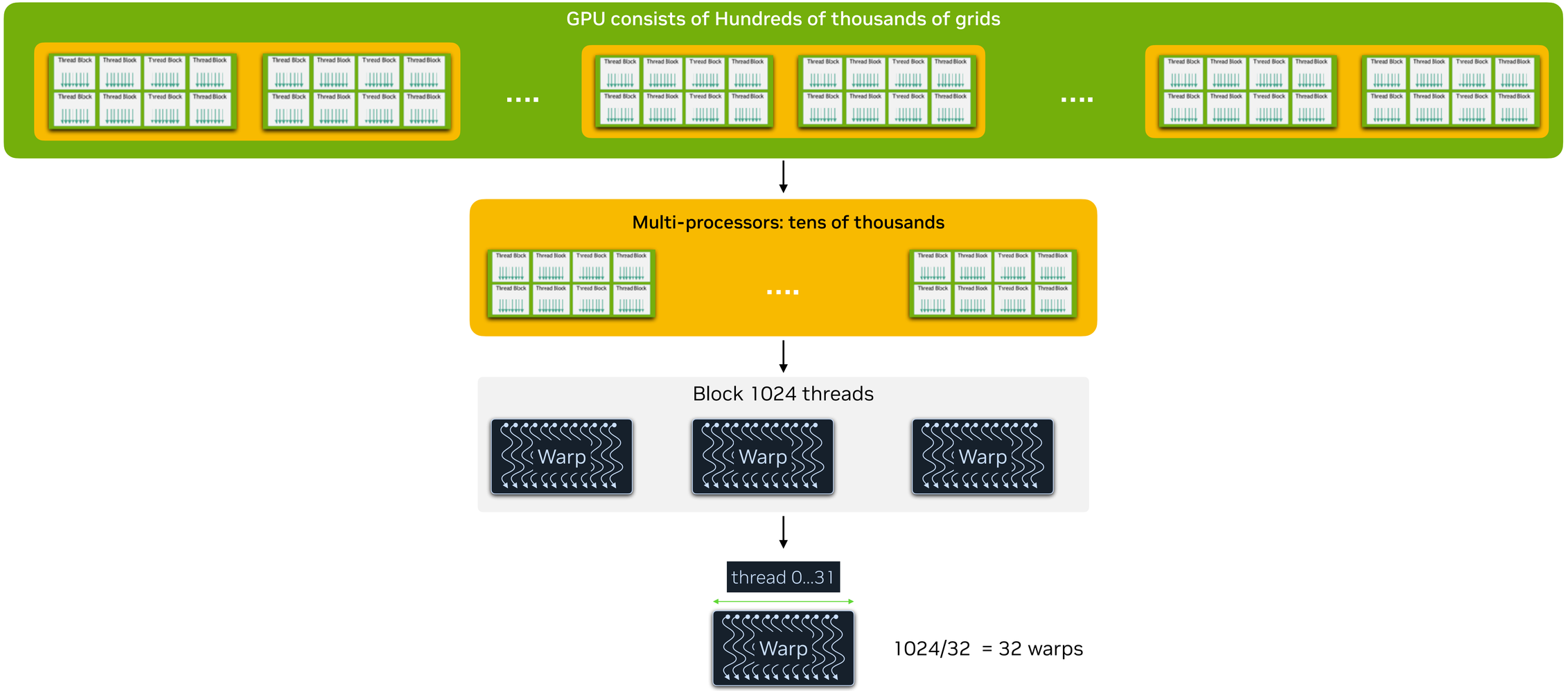
Synchronization

Launching kernel is asynchronous

cudaDeviceSynchronize(): wait until device code completeness

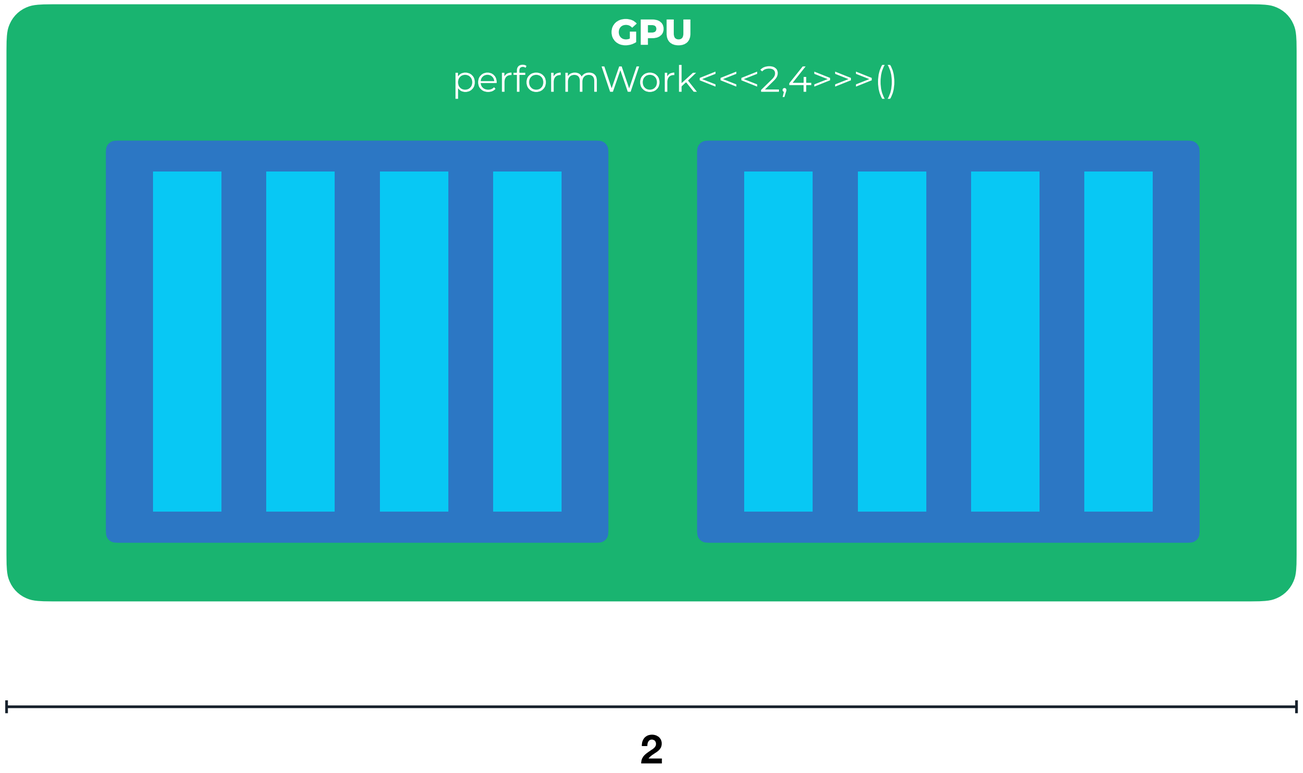
```
#include <stdio.h>
#include <cuda_runtime.h>

// Kernel function with correct parameter syntax
__global__ void loop(int N)
{
    int i = threadIdx.x;
    printf("This is iteration number %d\n", i);

}

int main()
{
    int N = 10;

    // Launch kernel with 1 block and N threads
    loop<<<1, N>>>(N);

    // Wait for GPU to finish
    cudaDeviceSynchronize();

    return 0;
}
```

GPU Thread hierarchy



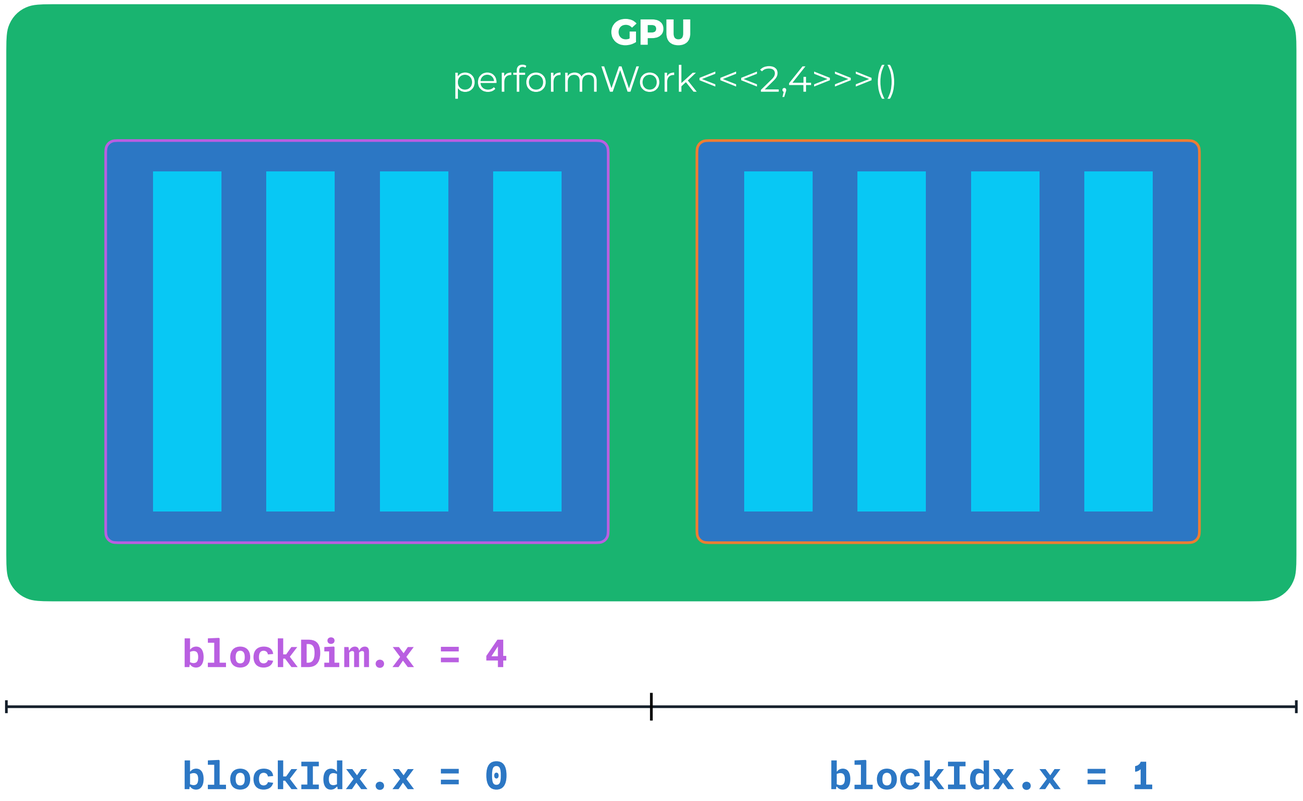
Kernel execution across Grid, Block and Thread

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



Kernel execution across Grid, Block and Thread

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



Kernel execution across Grid, Block and Thread

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

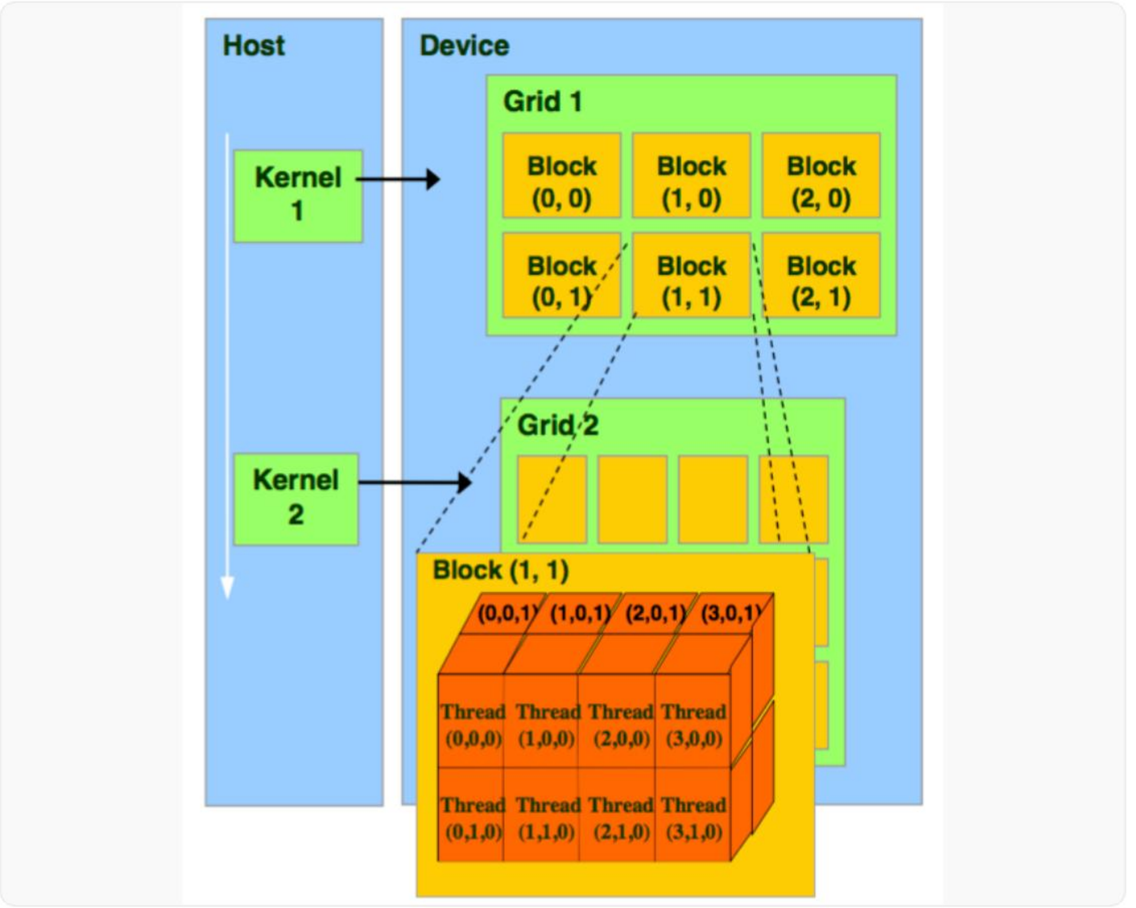
Calculating Global Index

When we use multiple blocks, `threadIdx.x` is not enough. It resets to 0 in every block.

The Magic Formula

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

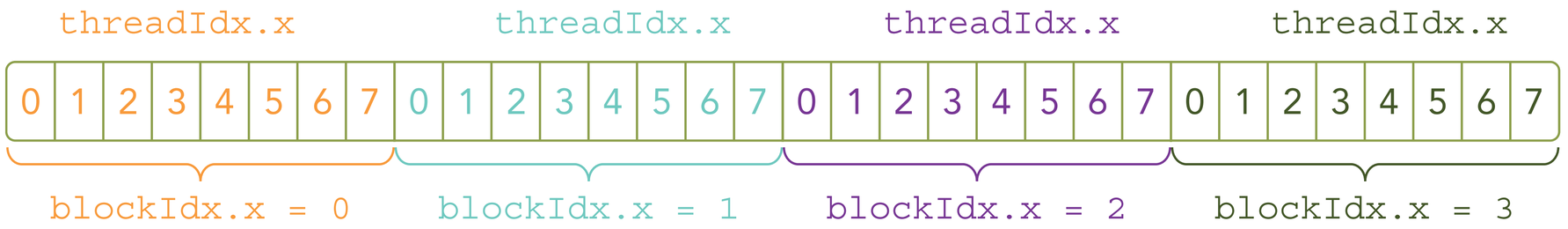
This gives every thread across the entire Grid a unique ID.



Kernel execution across Grid, Block and Thread

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**

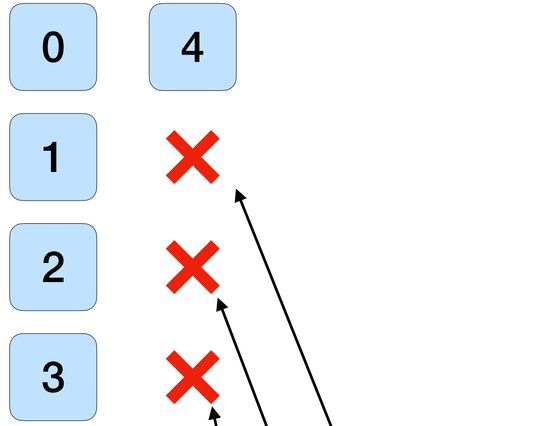



```
for blockIdx.x = 0
  i = 0 * 8 + threadIdx.x  = { 0, 1, 2, ... , 7 }
```

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

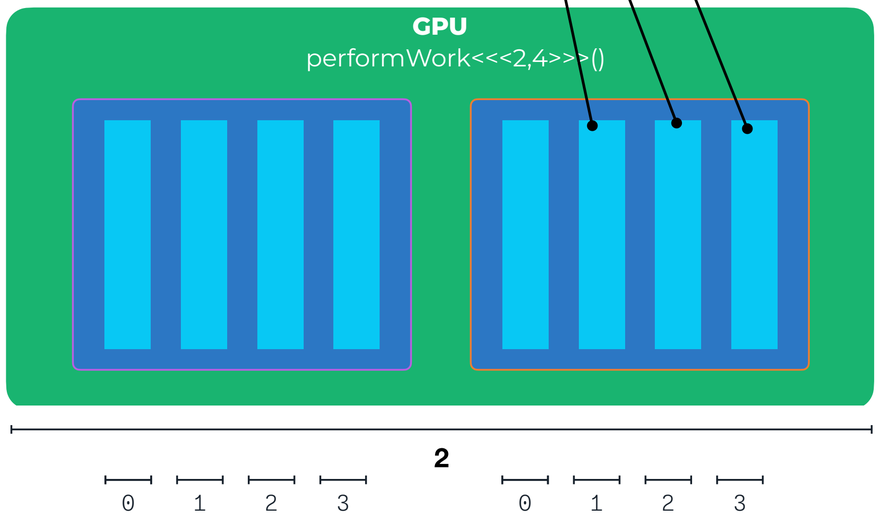
Handling vector of arbitrary size

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

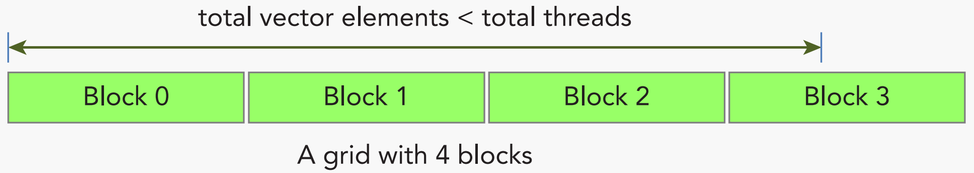
- Write an execution configuration that creates more threads than necessary
- Pass a value as an argument into the kernel (N) that represents that total size if the data set to be processed/total threads needed to complete the work
- Calculate the global index and if it does not exceed N perform the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

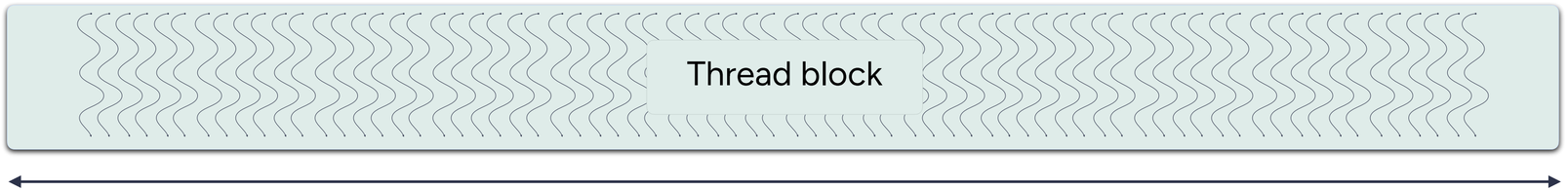
Choose the optimal block size

- 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**

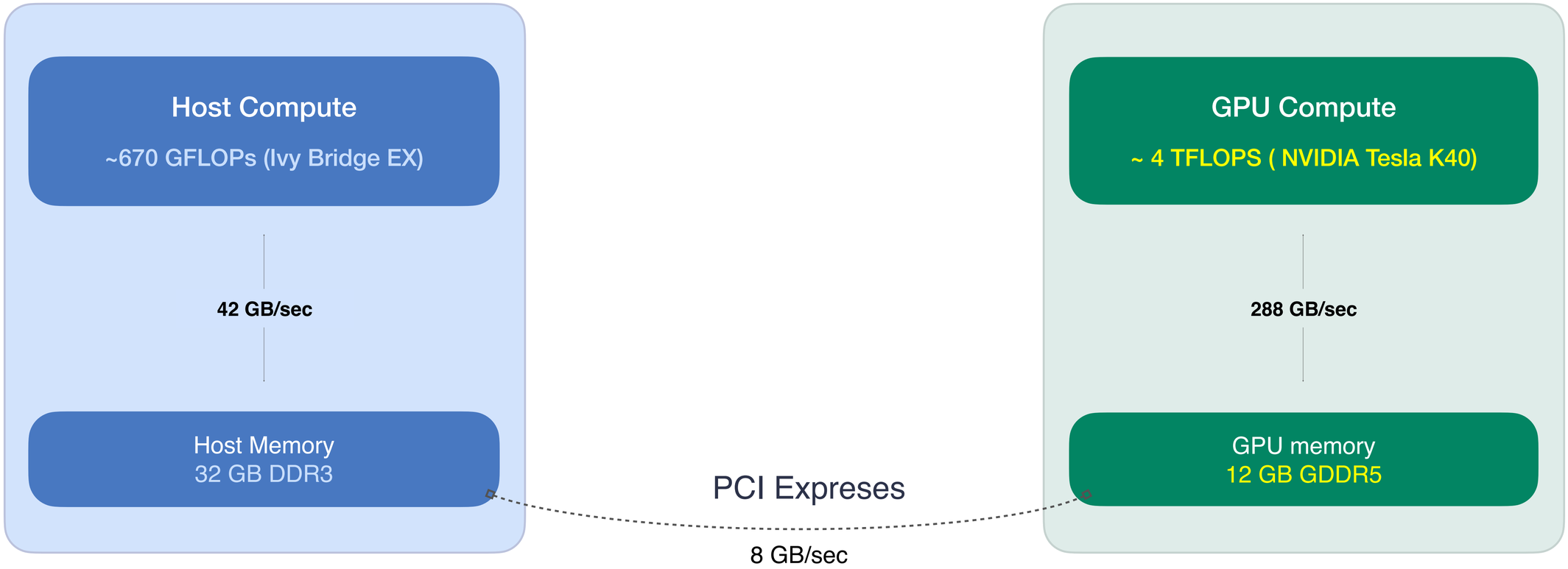
```
int index = threadIdx.x + (blockIdx.x * blockDim.x)
```

↓

All about this one line code

Module 2 : Memory and Data Flow

GPU vs. CPU: Understanding Performance Trade-offs



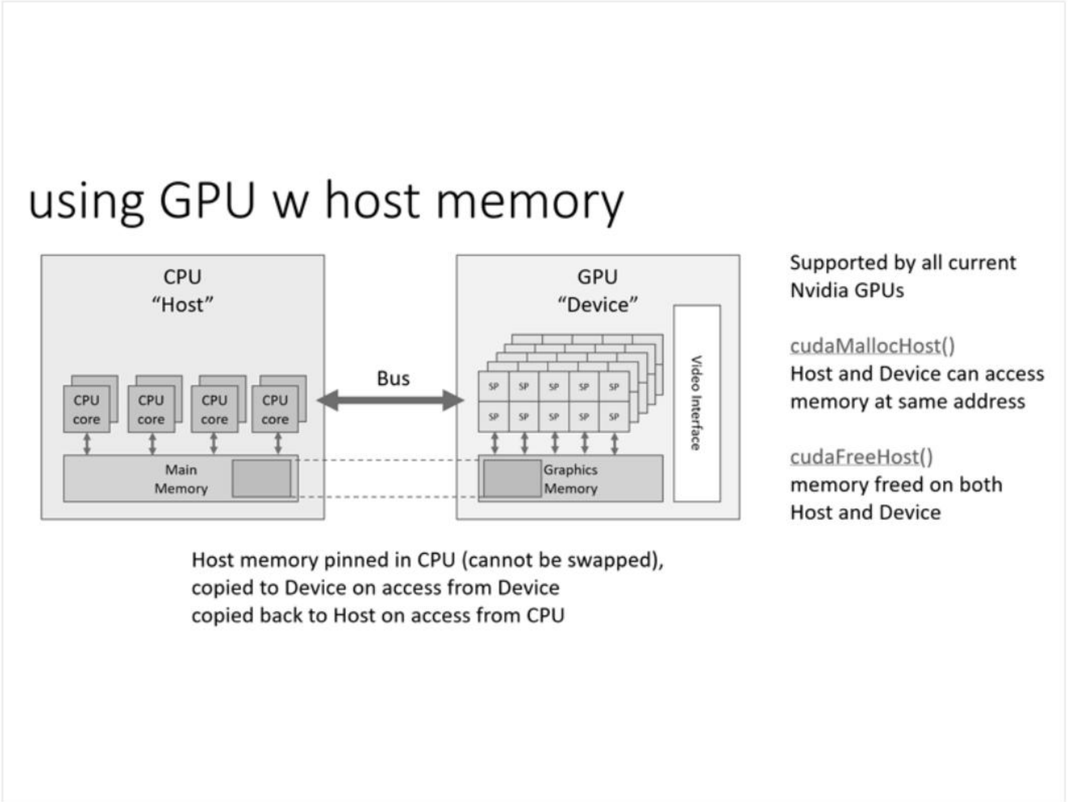
Impact of data transfer on overall application performance

Host vs Device Memory

Separation: The CPU and GPU have physically separate memory spaces (typically connected via PCIe).

Consequence: You cannot dereference a Host pointer on the Device, or vice versa.

Strategy: You must allocate on both, copy data in, compute, and copy data out.



The Five Steps of Every CUDA Program

1. **Prepare Host Data:** Allocate and initialize input arrays on the CPU (RAM).
2. **Allocate Device Memory:** Allocate output and copy-space arrays on the GPU (VRAM) using `cudaMalloc`.
3. **Data Transfer (H→D):** Copy input data from Host to Device using `cudaMemcpy`.
4. **Launch Kernel:** Execute the parallel code on the Device.
5. **Data Transfer (D→H):** Copy results back from Device to Host using `cudaMemcpy`.
6. **Cleanup:** Free all allocated memory on both Host and Device.

Device Allocation: cudaMalloc

cudaError_t cudaMalloc (void devPtr, size_t count);**

- `devPtr`: A Host pointer that will store the address of the newly allocated Device memory.
- `count`: The size in bytes to allocate.

```
float *d_a; // Declare a pointer on the Host
size_t size = N * sizeof(float);

// Allocate N floats on the GPU
cudaError_t err = cudaMalloc(
    (void**)&d_a, // Address of the host pointer
    size          // Size in bytes
);
```

Data Movement: cudaMemcpy

cudaError_t cudaMemcpy(...)

Used for moving data between Host and Device (and vice-versa).

```
cudaMemcpy(
    void* dst,        // Destination address
    const void* src, // Source address
    size_t count,    // Size in bytes
    enum cudaMemcpyKind kind // Direction of transfer
);
```

The `kind` Parameter

- `cudaMemcpyHostToDevice`: H → D (Input data to GPU)
- `cudaMemcpyDeviceToHost`: D → H (Result data back to CPU)
- `cudaMemcpyDeviceToDevice`: D → D (Fast GPU-side copy)
- `cudaMemcpyHostToHost`: H → H (Standard CPU copy)

Data Movement: cudaMemcpy

Moves data between Host and Device. This is a **blocking** (synchronous) call.

```
// Directions:
// cudaMemcpyHostToDevice
// cudaMemcpyDeviceToHost
// cudaMemcpyDeviceToDevice

// Copy Inputs to GPU
cudaMemcpy(d_in, h_in, size, cudaMemcpyHostToDevice);

// ... Run Kernel ...

// Copy Results back to CPU
cudaMemcpy(h_out, d_out, size, cudaMemcpyDeviceToHost);
```

Releasing the GPU memory

cudaError_t cudaFree (void* devPtr);

The equivalent of the Host's `free()`. Must be called for every pointer allocated with `cudaMalloc`.

> **Memory Leak Warning:** If you forget to call `cudaFree`, the VRAM will remain locked by your application until it closes, eventually leading to out-of-memory errors.

```
// Free the memory on the Device
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);

// Free the memory on the Host
delete[] h_a;
```

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;
}
```

The Critical Importance of Error Checking

Silent Failure

Unlike standard C/C++, CUDA operations (especially kernel launches) can fail silently.

All CUDA Runtime API calls return a `cudaError_t` status code.

- Check `cudaMalloc` (e.g., ran out of VRAM).
- Check `cudaMemcpy` (e.g., invalid direction or null pointer).
- Check Kernel Launch (use `cudaGetLastError()` after the launch).

Checking Status Codes

```
cudaError_t err = cudaMalloc(&d_a, size);
if (err ≠ cudaSuccess) {
    printf("Malloc failed: %s\n",
        cudaGetErrorString(err));
    exit(1);
}
```

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() )
}
```

Understanding cudaError_t

Almost all CUDA runtime API calls return an error code.

```
cudaError_t err = cudaMalloc(&ptr, size);

if (err != cudaSuccess) {
    printf("CUDA Error: %s\n", cudaGetErrorString(err));
    // Handle error...
}
```

Note: Kernel launches (`<<<>>>`) do not return a value directly. You must check `cudaGetLastError()` .

Best Practice: The CHECK Macro

```
#define CHECK(call) { \
    const cudaError_t error = call; \
    if (error != cudaSuccess) { \
        printf("Error: %s:%d, ", __FILE__, __LINE__); \
        printf("code:%d, reason: %s\n", error, \
                cudaGetErrorString(error)); \
        exit(1); \
    } \
}

// Usage
CHECK(cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice));
```

Exercise 3

Exercise -3 : Vector Sum

The "Hello World" of GPGPU

We want to calculate $C_i = A_i + B_i$ for every element in vectors A, B, and C.

This is a classic example of an **Embarrassingly Parallel** problem: each element calculation is independent of all others.

A simple CPU implementation requires a single loop.

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

Performance: Time complexity $O(N)$.

Parallel Goal (GPU

N Threads for N Elements

The goal is to eliminate the loop by assigning exactly one CUDA thread to calculate each element C_i.

Performance: Time complexity $O(1)$ (ignoring overhead and memory latency).

Mapping the Index

If Thread ID i is responsible for C_i, then the kernel only needs one line of core logic:

```
C[i] = A[i] + B[i];
```

Solution

The VecAdd Kernel: Index Calculation

The kernel must calculate the global index `i` from its position within the Grid.

```
__global__ void vecAdd(const float* d_A, const float* d_B, float* d_C, int N)
{
    // Calculate the global index i for this thread
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    // Boundary check: only process elements within the vector size N
    if (i < N)
    {
        d_C[i] = d_A[i] + d_B[i];
    }
}
```

Configuring the Launch (Grid Size)

Parameters

- **N (Total Elements):** e.g., $10,000,000$
- `THREADS_PER_BLOCK`: A constant, usually 256 or 512.

We must calculate the number of blocks needed to cover N elements.

Calculating Blocks

Use the Ceiling division formula to ensure all elements are covered, even if N is not a multiple of the Block size.

```
int N = 1024;
int threadsPerBlock = 256; // Db

// Ceiling division: (N + T - 1) / T
int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock; // Dg

// blocksPerGrid = (1024 + 256 - 1) / 256 = 1279 / 256 = 4

vecAdd<<>>( ... );
```

Host Code: Step 1 & 2 (Setup & Malloc)

Setup & Malloc Host/Device

We declare and allocate the necessary pointers on both sides.

```cpp
const int N = 1 << 20; // 1 Million elements
size_t size = N * sizeof(float);

// Host Pointers
float *h_a, *h_b, *h_c;
// Device Pointers
float *d_a, *d_b, *d_c;

// 1. Allocate Host Memory (C++)
h_a = new float[N];
h_b = new float[N];
h_c = new float[N];

// Initialize Host Data (e.g., A[i]=1, B[i]=2)
// (Omitted for brevity)

// 2. Allocate Device Memory (CUDA API)
CHECK(cudaMalloc((void**)&d_a, size));
CHECK(cudaMalloc((void**)&d_b, size));
CHECK(cudaMalloc((void**)&d_c, size));
```

Host Code: Step 3 & 4 (Copy & Launch)

Transferring Data & Launching

We push the data to the Device, configure the execution, and launch the kernel.

```
// 3. Data Transfer H → D
CHECK(cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevic
CHECK(cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevic

// Determine launch parameters
int threadsPerBlock = 256;
int blocksPerGrid = (N + threadsPerBlock - 1) / thread

// 4. Launch Kernel
printf("Launching with %d blocks of %d threads\n",
    blocksPerGrid, threadsPerBlock);

// Launch the kernel function
vecAdd<<>>(d_a, d_b, d_c, N);

// Check for kernel launch errors immediately after la
CHECK(cudaGetLastError());
```

Host Code: Step 5 & 6 (Retrieve & Free)

Retrieval and Cleanup

We wait for the GPU, retrieve the result, and release all resources.

```
// Wait for the Device to complete all work
CHECK(cudaDeviceSynchronize());

// 5. Data Transfer D → H (Get Result)
CHECK(cudaMemcpy(h_c, d_c, size,
    cudaMemcpyDeviceToHost));

// (Validation/Verification step is usually here)
// (Omitted for brevity)

// 6. Cleanup Device Memory
CHECK(cudaFree(d_a));
CHECK(cudaFree(d_b));
CHECK(cudaFree(d_c));

// Cleanup Host Memory
delete[] h_a; delete[] h_b; delete[] h_c;

printf("Success!\n");
```

Importance of the Boundary Check

The Over-Launch Problem

Using ceiling division to calculate blocks often results in launching **more threads** than there are elements (N).

Example: $N=1025$, Block Size $= 256$. We launch 5 blocks (1280 threads total).

> Threads 1025 through 1279 will try to access memory outside the array bounds if the check is missing.

```
// The Guard in the Kernel
int i = blockIdx.x * blockDim.x + threadIdx.x;
if (i < N)
{
    // Safe code execution
    d_C[i] = d_A[i] + d_B[i];
}
```

The Performance Hit of cudaDeviceSynchronize

Blocking Call

`cudaDeviceSynchronize()` is a **blocking call**. The CPU stops and waits for *all* pending GPU work to finish.

This is necessary for correctness (to ensure results are ready before retrieval), but it kills potential CPU-side work overlap.

Asynchronicity is Key

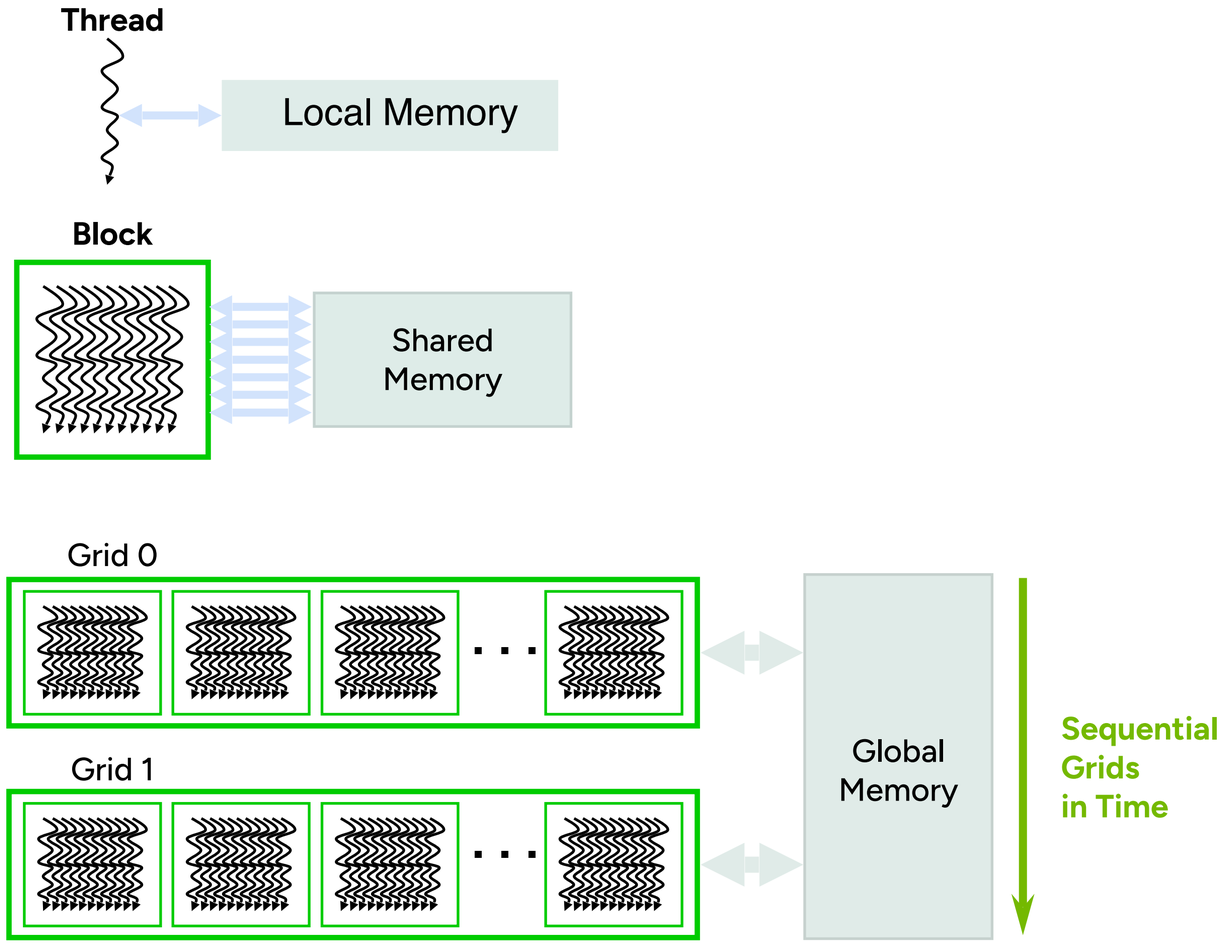
Most CUDA API calls (`cudaMemcpy` H→D, Kernel Launches) are **asynchronous** (non-blocking).

The Host immediately moves to the next instruction while the Device works.

Module 3 will cover Streams, which manage multiple asynchronous tasks and reduce synchronization pain.

The CUDA Memory Hierarchy

GPU memory breakdown



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

- Local Memory: per-thread
  - Private per thread
  - Auto variables, register spill
- Shared Memory: per-Block
  - Shared by threads of the same block
  - Inter-thread communication
- Global Memory: per-application
  - Shared by all threads
  - Inter-Grid communication

Global Memory (__device_)

Characteristics

- **Scope:** Global (all threads, all blocks, all kernels).
- **Lifetime:** From `cudaMalloc` to `cudaFree`.
- **Allocation:** Allocated on the Host via `cudaMalloc`.
- **Speed:** High latency (slow to access relative to cache). Bandwidth matters most here.

Why is it "Slow"?

It is physically located far from the SM cores, requiring access through the L2 cache and DRAM controller.

Goal: Hide this latency by launching massive numbers of threads (high occupancy).

Simplified Data Management: Unified Memory

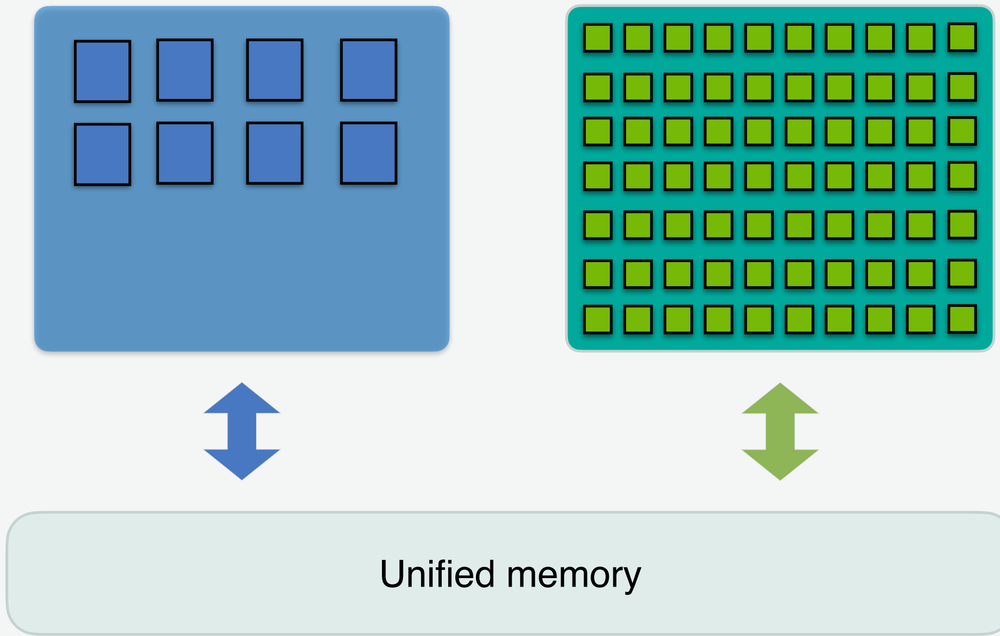
Increased memory latency

- 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

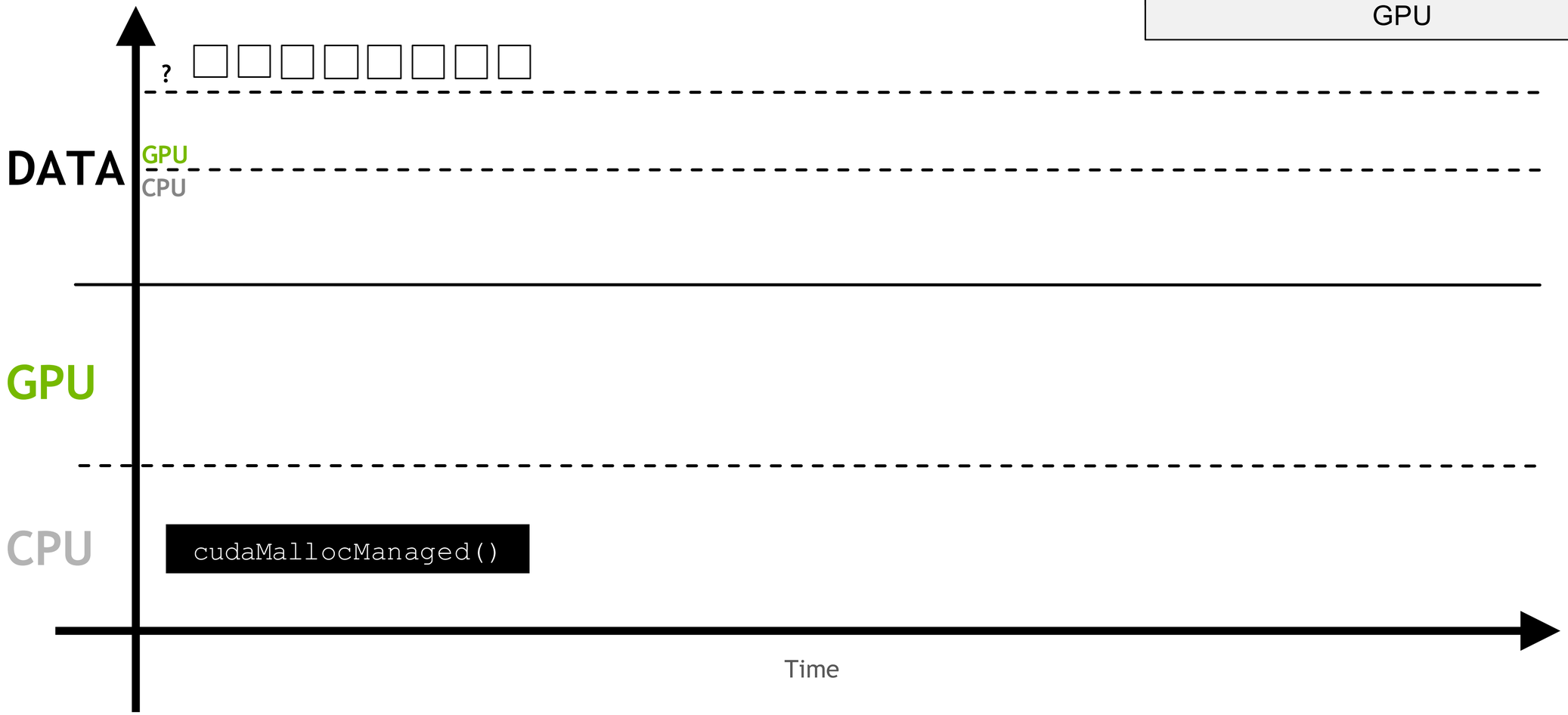
UVM automatically manages memory placement, which may not always be optimal for a given application

Developer view of GPU memory



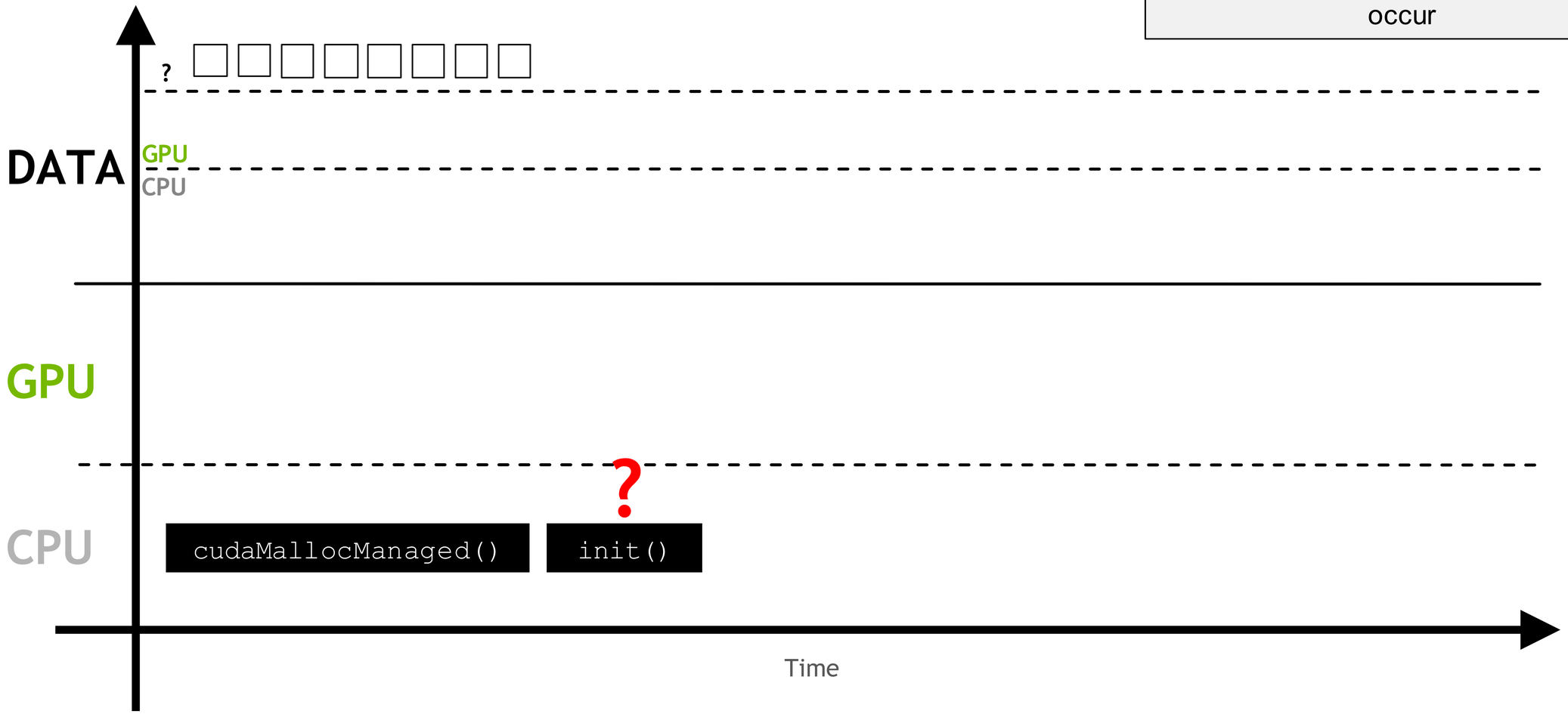
How does cudaMallocManaged actually works?

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



How does cudaMallocManaged actually works?

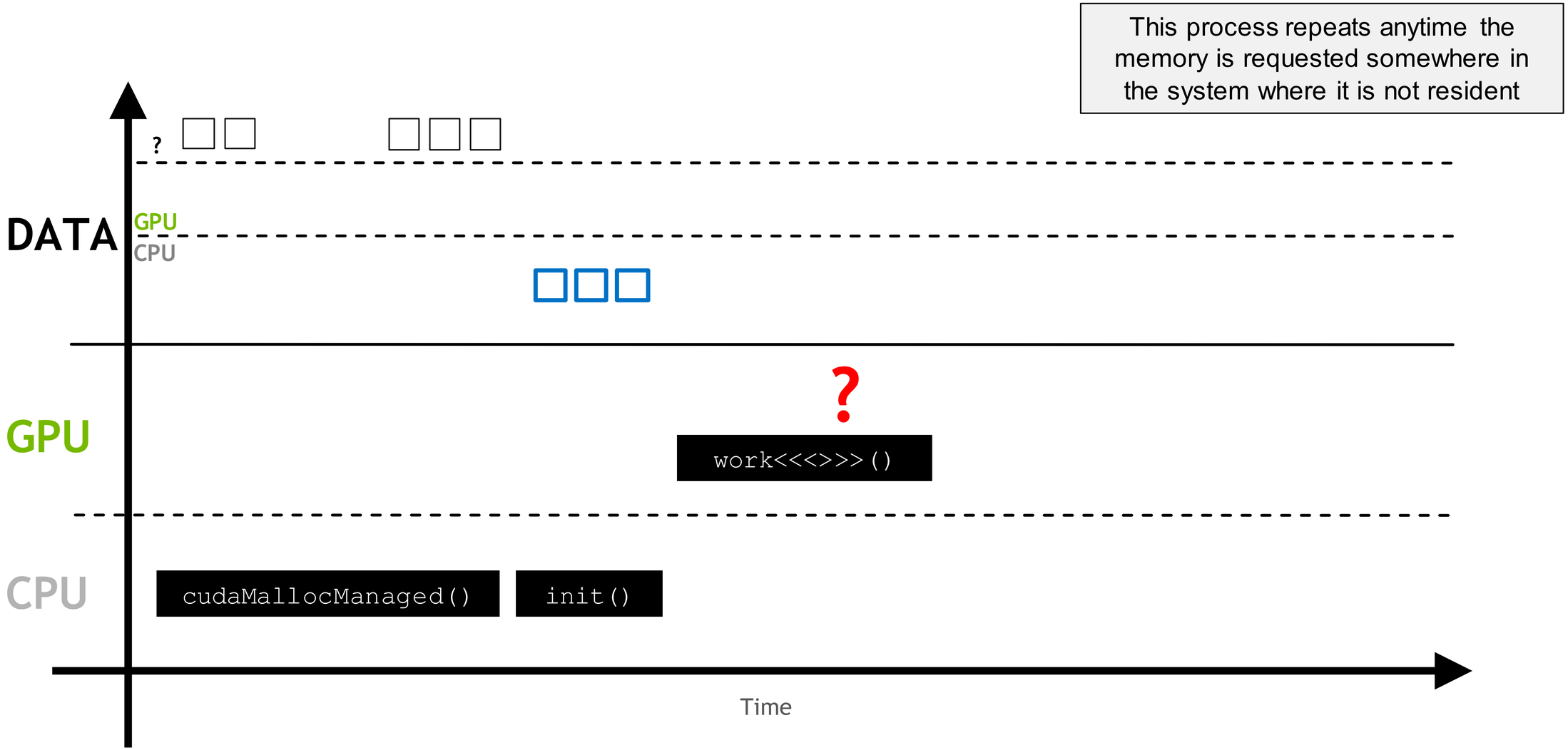
When some work asks for the memory for the first time, a **page fault** will occur



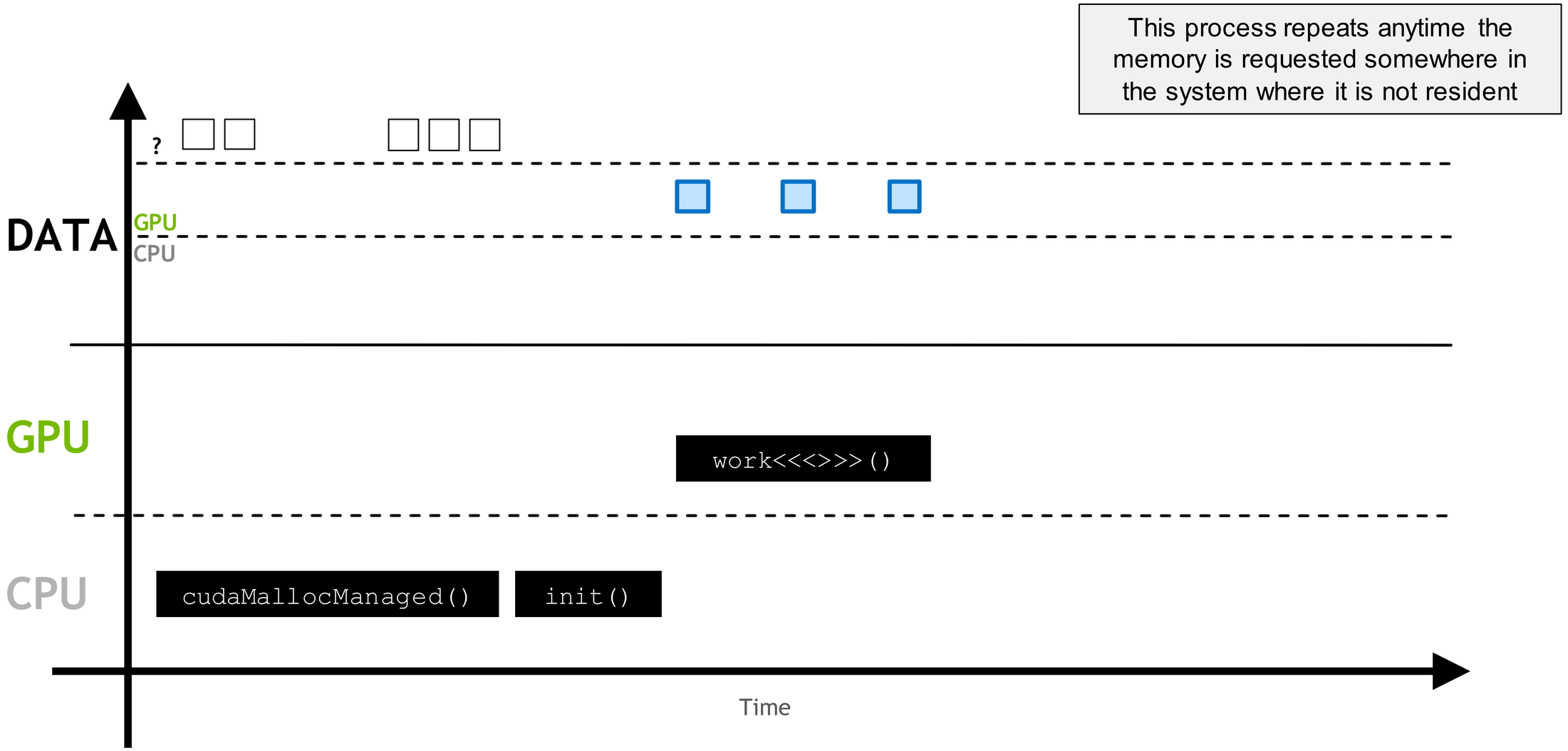
How does cudaMallocManaged actually works?

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

The Managed Memory Model

Introduced to simplify the separation. It allows a single pointer to be accessed by both the CPU and GPU.

```
// Allocation API changes:
cudaMallocManaged((void**)&h_a, size);
```

How it Works

The CUDA Runtime performs "On-Demand Paging": data is automatically migrated by the driver between Host and Device memory as needed.

It removes explicit `cudaMemcpy` calls, but the data transfer penalty still exists!

Exercise 4

Parallel Goal (GPU

- Take the previous exercise
- Use unified memory
- Compile and run the code
- Profile and compare it exercise 3

Pros/Cons of Unified Memory

Pros

- Simpler code.
- No manual `cudaMemcpy`.
- Allows "Oversubscription" (allocating more than GPU RAM).

Cons

- Page faults are expensive (latency).
- Harder to hide latency with overlap
- Driver magic hides performance bottlenecks.

Review: Execution Configuration Dims

`dim3` Type

Both the Grid (`Dg`) and the Block (`Db`) dimensions are defined using the `dim3` struct (which has `x`, `y`, and `z` fields).

```
dim3 threads(32, 8, 1); // 32*8=256 threads
dim3 blocks(16, 16, 1); // 256 blocks

// Total threads: 256 * 256 = 65,536
```

Mapping to Index

For 2D or 3D datasets (like images or 3D simulation grids), using 2D/3D blocks/grids is intuitive.

Example: 2D Index i, j

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

2D Indexing (Matrix Multiplication)

Matrix $C[R][C]$

A single thread calculates one element $C[row][col]$.

We map the thread's 2D position directly to the matrix indices.

```
__global__ void matrixMulKernel( ... )
{
    // Column index (y-dimension)
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    // Row index (y-dimension)
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    if (row < R && col < C) {
        // C[row*width + col]
        // Do the computation ...
    }
}
```

Measuring Performance: The CUDA Timer

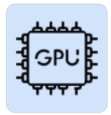
`cudaEvent` API

Used to measure the time elapsed for Device operations (Kernels and `cudaMemcpy` D→H).

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start, 0);

// Kernel Launch or Memcpy
//  ...

cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);

float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
printf("Time: %f ms\n", milliseconds);
```

Why Use `cudaEvent`?

Standard Host timers like `std::chrono` can only measure Host time. Since kernel launches are asynchronous, the CPU will finish timing before the GPU starts! `cudaEvent` forces the GPU to record the actual Device time.



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

Asynchronicity: CUDA Streams

The Need for Parallelism

By default, all CUDA operations use Stream 0 (the default stream) and execute sequentially.

A Stream is a sequence of operations that execute in issue-order on the Device. Operations in different streams may execute concurrently.

Benefits of Streams

- **Overlap:** Compute (Kernel) can overlap with Data Transfer (Memcpy).
- **Concurrent Execution:** Multiple independent Kernels can run at the same time.
- **Batching:** Dividing a large problem into smaller batches for efficient queuing.

What Is a CUDA STREAM?

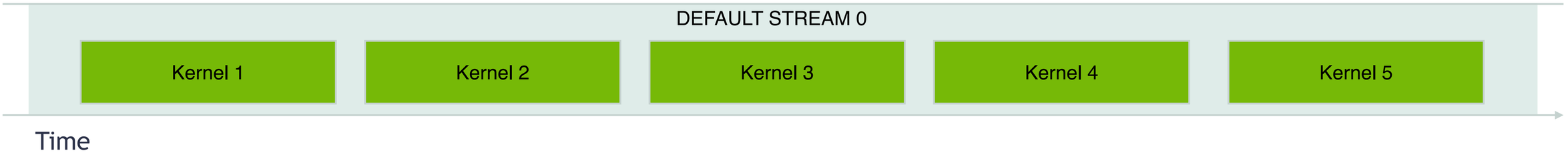
1

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



Non-Default Stream Behaviour

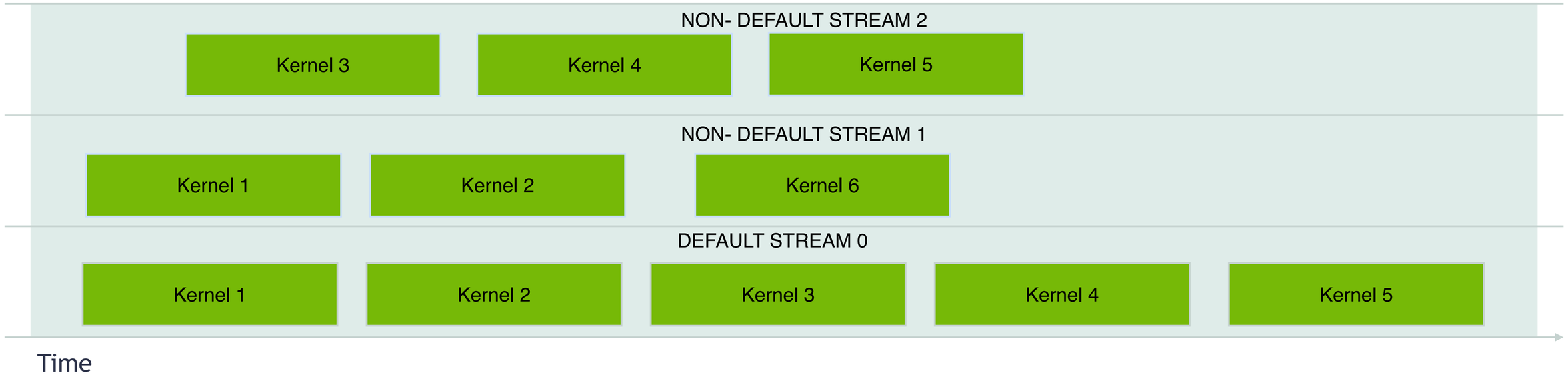
2

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



Where It Can Be Useful?

1

Kernel Enqueuing

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

2

Memory Transfer

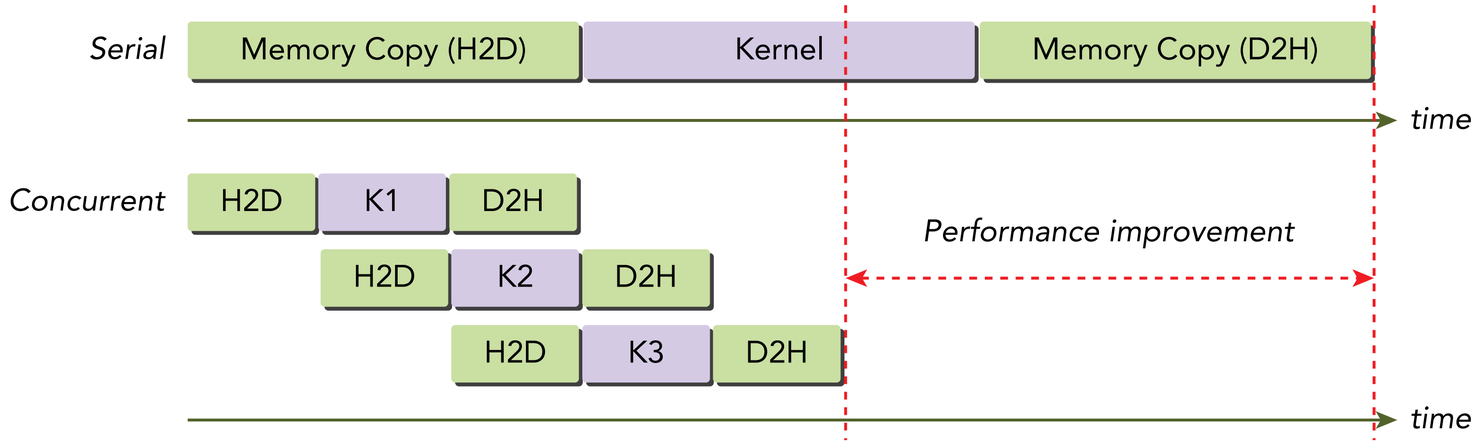
Data transfers between host and device can be enqueued asynchronously into streams.

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 API

Creation and Destruction

```
cudaStream_t stream1;
CHECK(cudaStreamCreate(&stream1));
//  ... use stream1 ...
CHECK(cudaStreamDestroy(stream1));
```

Enqueueing Operations

Operations are enqueued by passing the stream handle as the final parameter.

```
// Asynchronous Memcpy H→D
cudaMemcpyAsync(d_a, h_a, size,
    cudaMemcpyHostToDevice, stream1);

// Kernel Launch using stream1
kernel<<>>(d_a);

// Asynchronous Memcpy D→H
cudaMemcpyAsync(h_b, d_b, size,
    cudaMemcpyDeviceToHost, stream1);
```

Stream Synchronization

Two Levels of Synchronization

To ensure correctness, you must wait for the specific stream or all activity to complete.

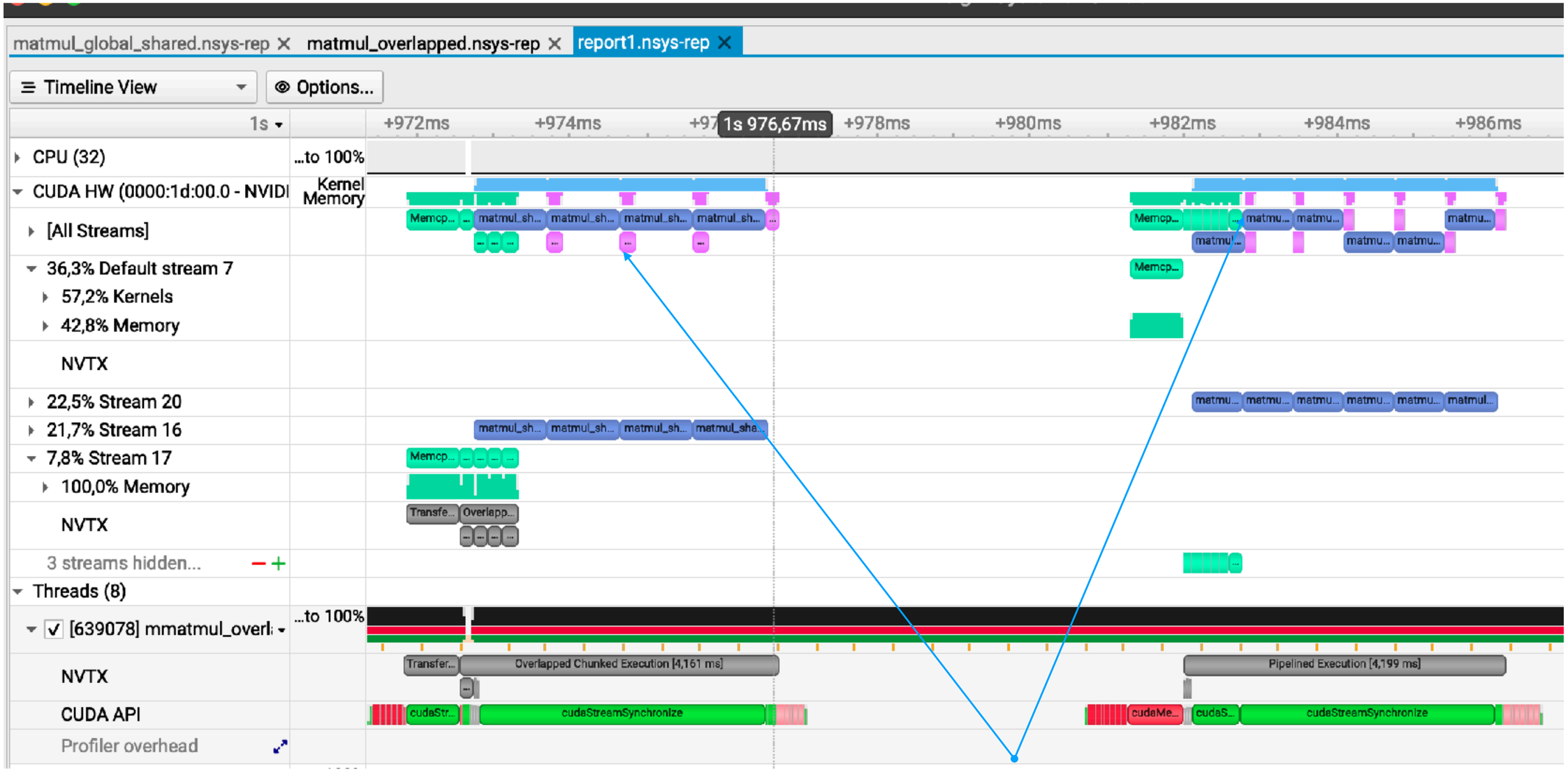
- **Single Stream:** `cudaStreamSynchronize(stream)` (Host waits for specific stream).
- **All Streams:** `cudaDeviceSynchronize()` (Host waits for all work on the GPU).

`cudaEventRecord` (Stream-to-Stream Sync)

Events allow one stream to wait for a specific point in another stream to complete, avoiding full stream blocking.

```
cudaEventRecord(event, stream1);
cudaStreamWaitEvent(stream2, event, 0);

// Stream 2 will wait for 'event' in Stream 1 before starting.
```

Nsight System Report



Look at this pattern