

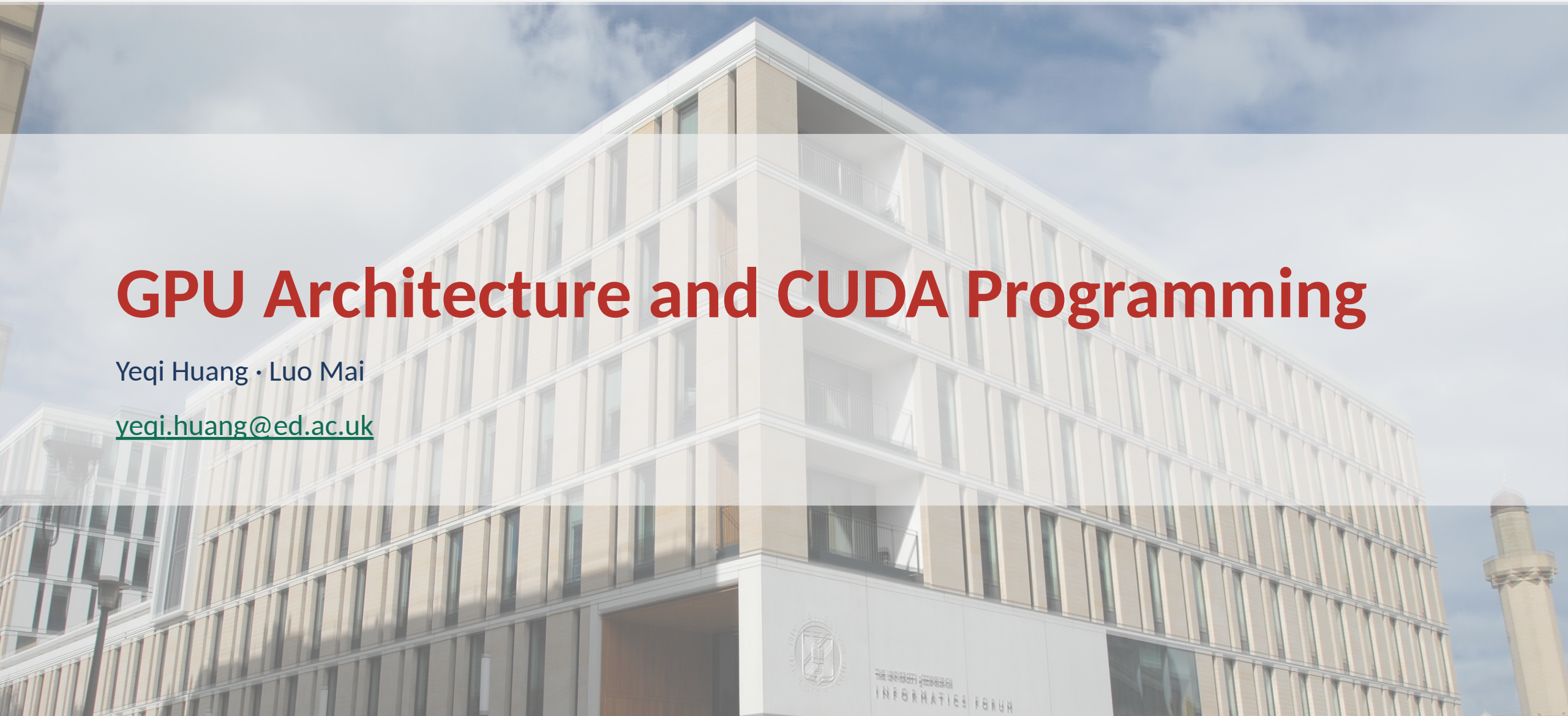


THE UNIVERSITY of EDINBURGH
informatics

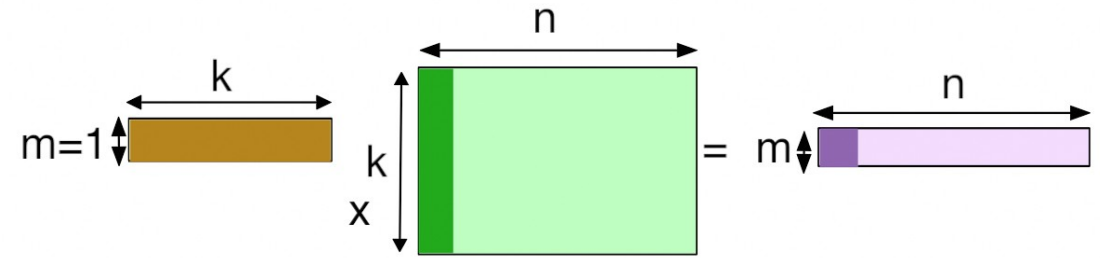
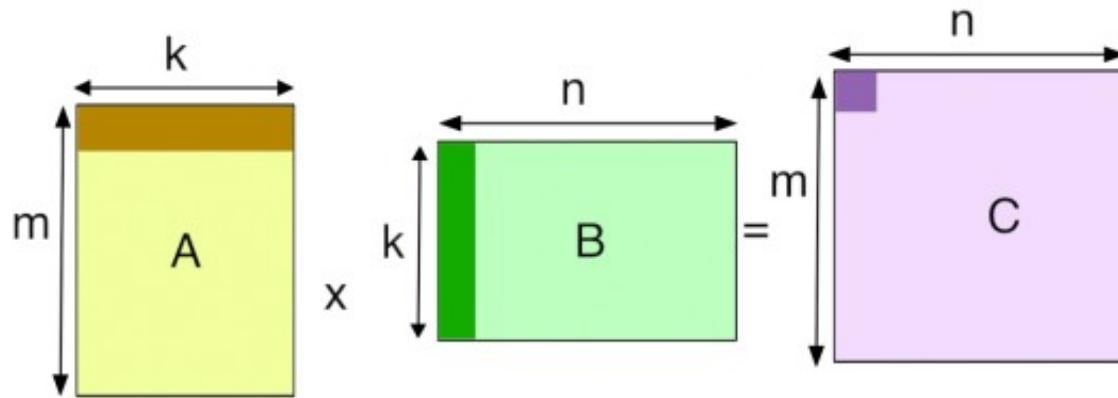
GPU Architecture and CUDA Programming

Yeqi Huang • Luo Mai

yeqi.huang@ed.ac.uk



GEMM and GEMV



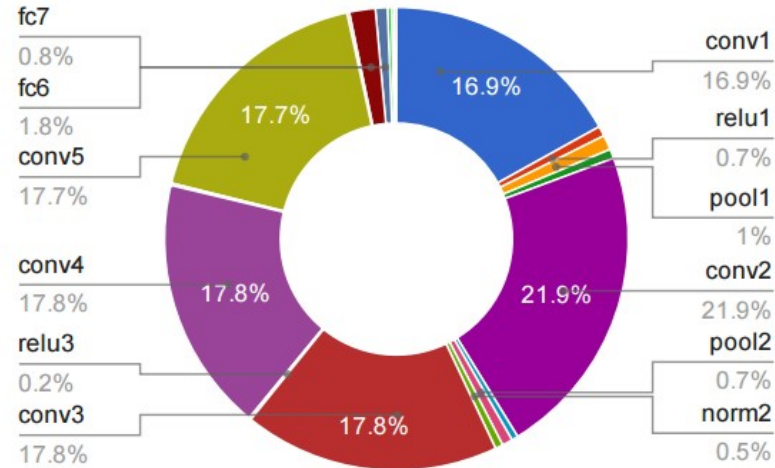
GEMM $C = A \times B$: matrix $\times$ matrix

GEMV $y = x \times A$: row vector $\times$ matrix

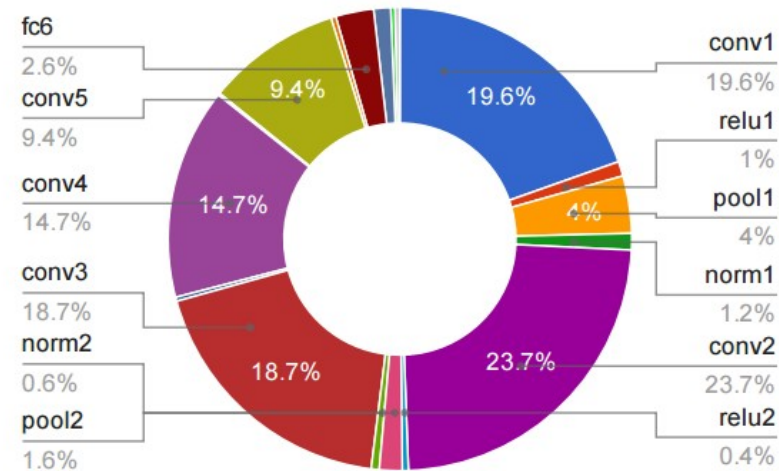
GEMV is GEMM with $m = 1$: same math, but each matrix element is used only once

Why GEMM is at the heart of deep learning?

GPU Forward Time Distribution



CPU Forward Time Distribution



Conv + fc layers take $\approx 90\%$ of forward time (both charts) — and both run as GEMM

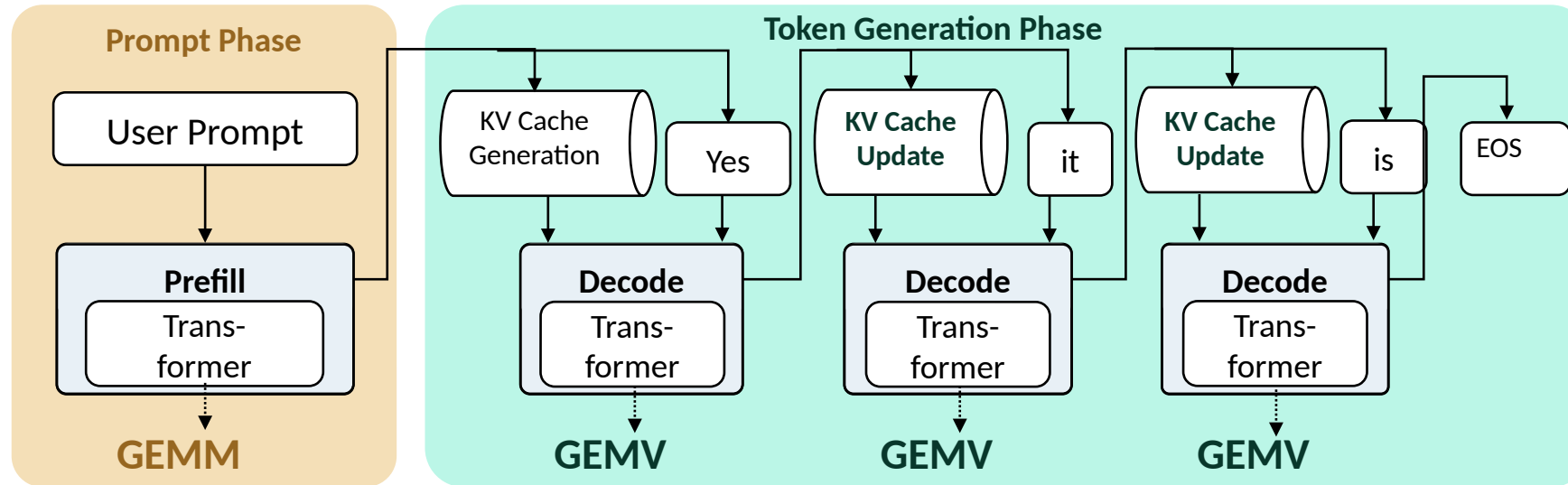
Fully connected.
one GEMM per layer

Convolution.
lowered to GEMM (im2col)

But also:
data movement and reuse
decide speed

Further reading: “Why GEMM is at the heart of deep learning” (P. Warden); Berkeley DNN analysis (Y. Jia, 2014)

GEMM and GEMV in modern AI applications



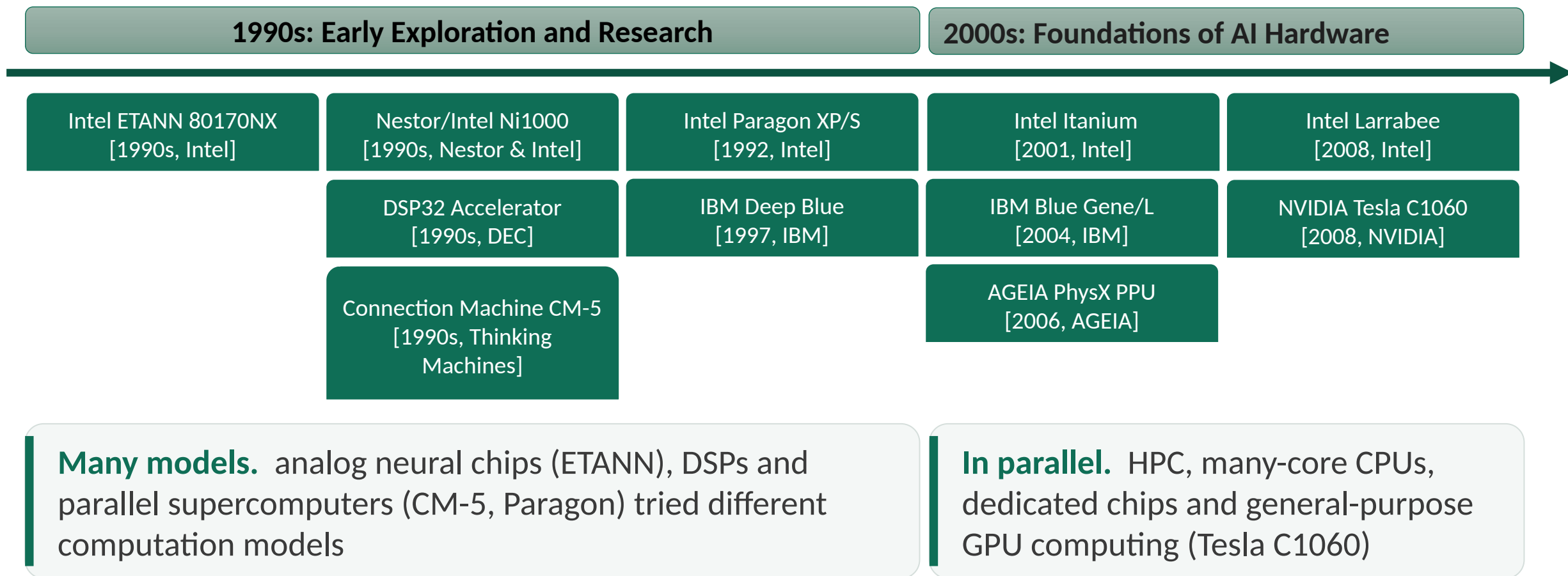
Prefill → GEMM all prompt tokens at once:
matrix × matrix

Decode → GEMV one token per step:
matrix × vector at small batch

Same model, two kernel shapes: prefill is compute-bound, decode is memory-bound.

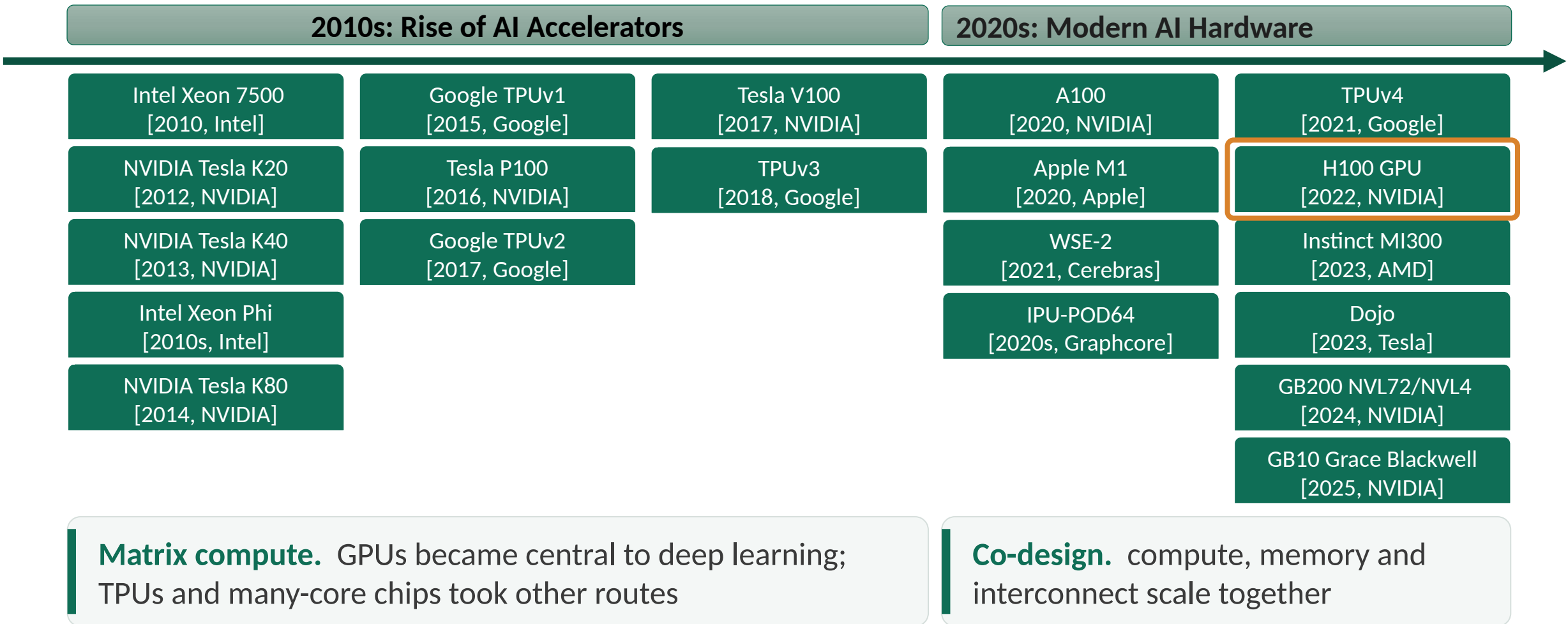
Kernel research for these shapes: FlashDecoding++, LUT-GEMM, T-MAC (details in speaker notes)

A brief history of AI accelerators (I)



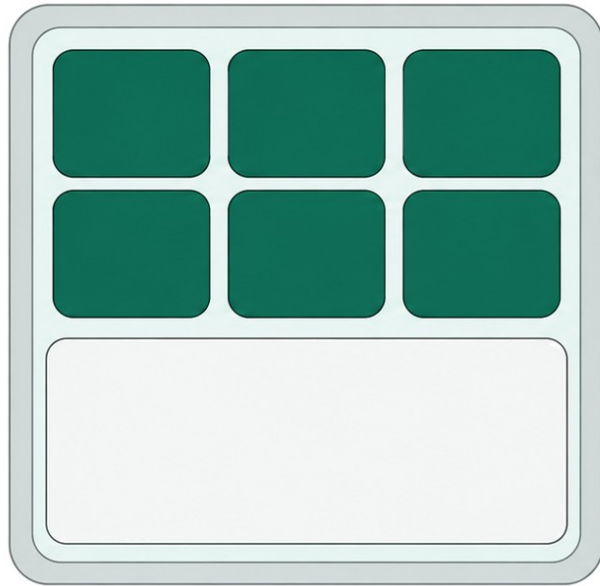
A broad AI / HPC history, not a single GPU family line.

A brief history of AI accelerators (II)

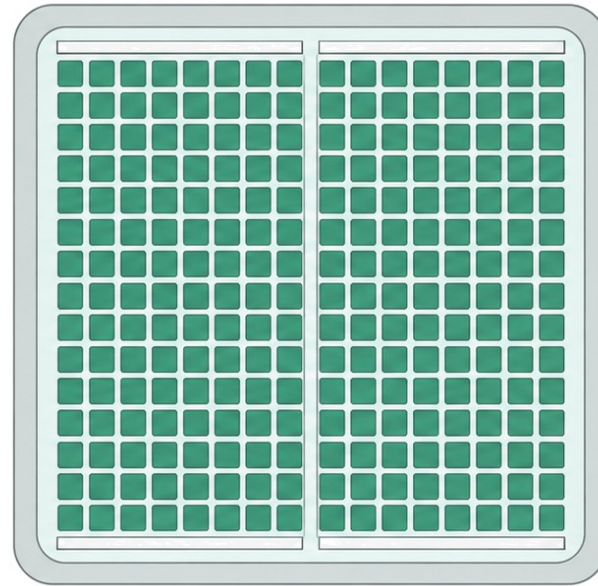


H100 (2022, outlined) is our concrete example of a modern GPU architecture.

CPUs and GPUs: fundamentally different



CPU
optimized for serial tasks



GPU
optimized for parallel tasks

Silicon budget

	CPU	GPU
ALU	Less	More
Control	More	Less
Cache	More	Less

Source: NVIDIA, Introduction to GPU Computing (P. Graham); figures redrawn.

A CPU spends its area on control and cache; a GPU spends it on arithmetic units.

CPU reduces latency; GPU hides latency

CPU: reduce latency

Strengths

- Very large main memory
- Very fast clock speeds
- Large caches cut memory latency
- A few threads run very quickly

Weaknesses

- Lower memory bandwidth
- Cache misses are costly
- Lower performance per watt

GPU: hide latency

Strengths

- High-bandwidth main memory
- Much more compute
- Tolerates latency through parallelism
- High throughput and performance per watt

Weaknesses

- Smaller memory capacity
- Low per-thread performance

Source: NVIDIA, Introduction to GPU Computing (P. Graham).

A GPU keeps many threads in flight: while some wait for memory, others compute.

Single-chip inference performance: 1000× in 10 years

Where the speedup comes from

Smaller numbers ~16× faster

FP32 → FP16 → INT8: fewer bits per value, more math per cycle

Matrix instructions ~12.5× faster

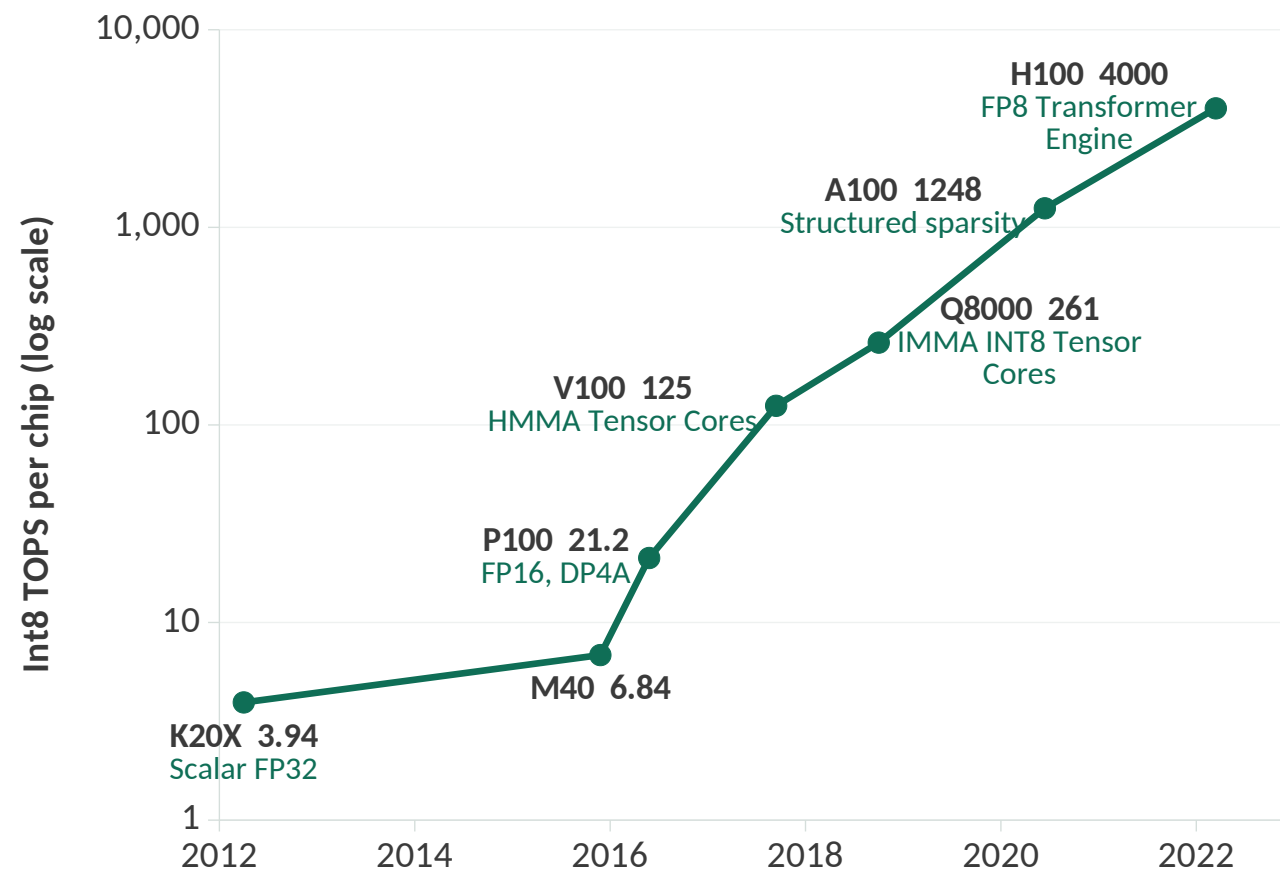
DP4A, HMMA, IMMA: many multiply-adds per instruction

Smaller transistors ~2.5× faster

28 → 16 → 7 → 5 nm: more units fit on one chip

Sparsity ~2× faster

skip multiplications by zero weights



Source: NVIDIA, Introduction to GPU Computing (P. Graham).

Multiply them: $16 \times 12.5 \times 2.5 \times 2 \approx 1000\times$. Most of it is smaller numbers and matrix instructions.

Common features of AI accelerators

Key trends

General-purpose CPUs → specialized AI chips (GPUs, ASICs)

Parallelism and scalability for complex AI workloads

Energy-efficient AI accelerators

Edge AI: chips for real-time, low-latency tasks

What they all share

Compute units with high parallelism for specific tasks

In deep learning: **GEMM / GEMV**

Why we use NVIDIA GPUs in this course

Three reasons: suited to AI • available to us • practical to learn

Suited to AI. large tensor operations expose parallelism and data reuse

Accessible. we run CUDA on the university teaching GPU cluster

Practical ecosystem. CUDA C++, cuBLAS and profilers for hands-on learning

Other vendors also offer programmable platforms.

H100 architecture: the white paper

Our map: the full GH100 diagram from the [H100 white paper](#)



Full GH100 GPU with 144 SMs (NVIDIA H100 Tensor Core GPU Architecture white paper)



NVIDIA H100 Tensor Core GPU Architecture

EXCEPTIONAL PERFORMANCE, SCALABILITY, AND SECURITY FOR THE DATA CENTER

NVIDIA H100 GPU Architecture In-Depth



Figure 6. GH100 Full GPU with 144 SMs

H100 SM Architecture

Building upon the NVIDIA A100 Tensor Core GPU SM architecture, the H100 SM quadruples A100's peak per-SM floating point computational power, due to the introduction of FP8, and doubles A100's raw SM computational power on all previous Tensor Core and FP32 / FP64 data types, clock-for-clock.

The new Transformer Engine, combined with Hopper's FP8 Tensor Cores, delivers up to 9x faster AI training and 30x faster AI inference speedsups on large language models compared to the prior generation A100. Hopper's new DPX instructions enable up to 7x faster Smith-Waterman algorithm processing for genomics and protein sequencing.

Hopper's new fourth-generation Tensor Core, Tensor Memory Accelerator, and many other new SM and general H100 architecture improvements together deliver up to 3x faster HPC and AI performance in many other cases.

NVIDIA H100 Tensor Core GPU Architecture

19

The full GH100 design

Units in the full GH100 implementation

144 SMs 8 GPCs $\times$ 9 TPCs $\times$ 2 SMs per TPC

6 HBM stacks HBM3 / HBM2e, 12 memory controllers (512-bit)

18,432 FP32 CUDA cores: 128 per SM

60 MB L2 cache

576 4th-generation Tensor Cores: 4 per SM

NVLink 4 GPU-to-GPU links; PCIe Gen 5 to the host

Full chip $\neq$ product: shipping H100s enable fewer units (next slide).

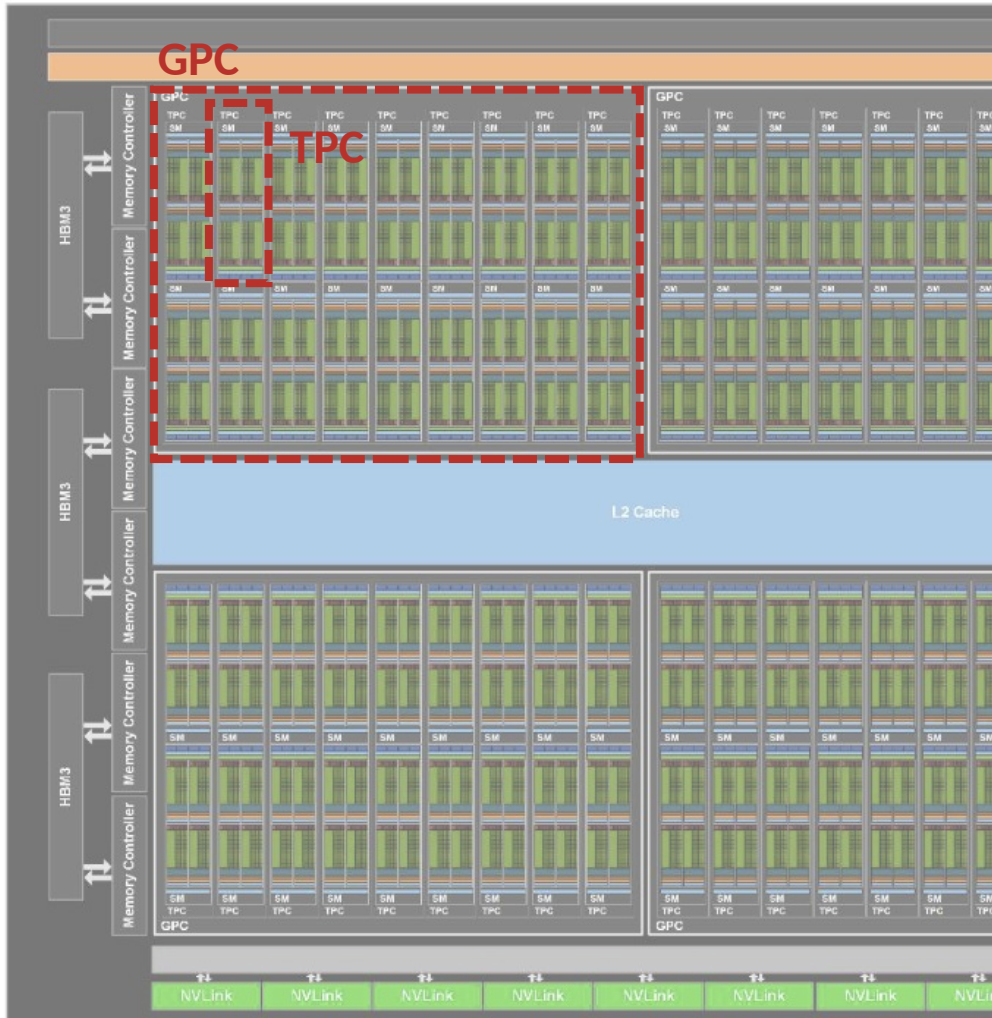
Full GH100 design and H100 products

Configuration	SMs	L2	Memory context
Full GH100 design	144	60 MB	6 HBM3 / HBM2e stacks
H100 SXM, original 80 GB	132	50 MB	80 GB HBM3
H100 PCIe, original 80 GB	114	50 MB	80 GB HBM2e

Configurations from NVIDIA's 2022 Hopper architecture material

Diagram ≠ product: full GH100 has 144 SMs; H100 SXM 132, H100 PCIe 114

GH100: GPC, TPC and SM



Compute hierarchy

GPC Graphics Processing Cluster: 9 TPCs

TPC Texture Processing Cluster: 2 SMs

SM Streaming Multiprocessor: runs blocks

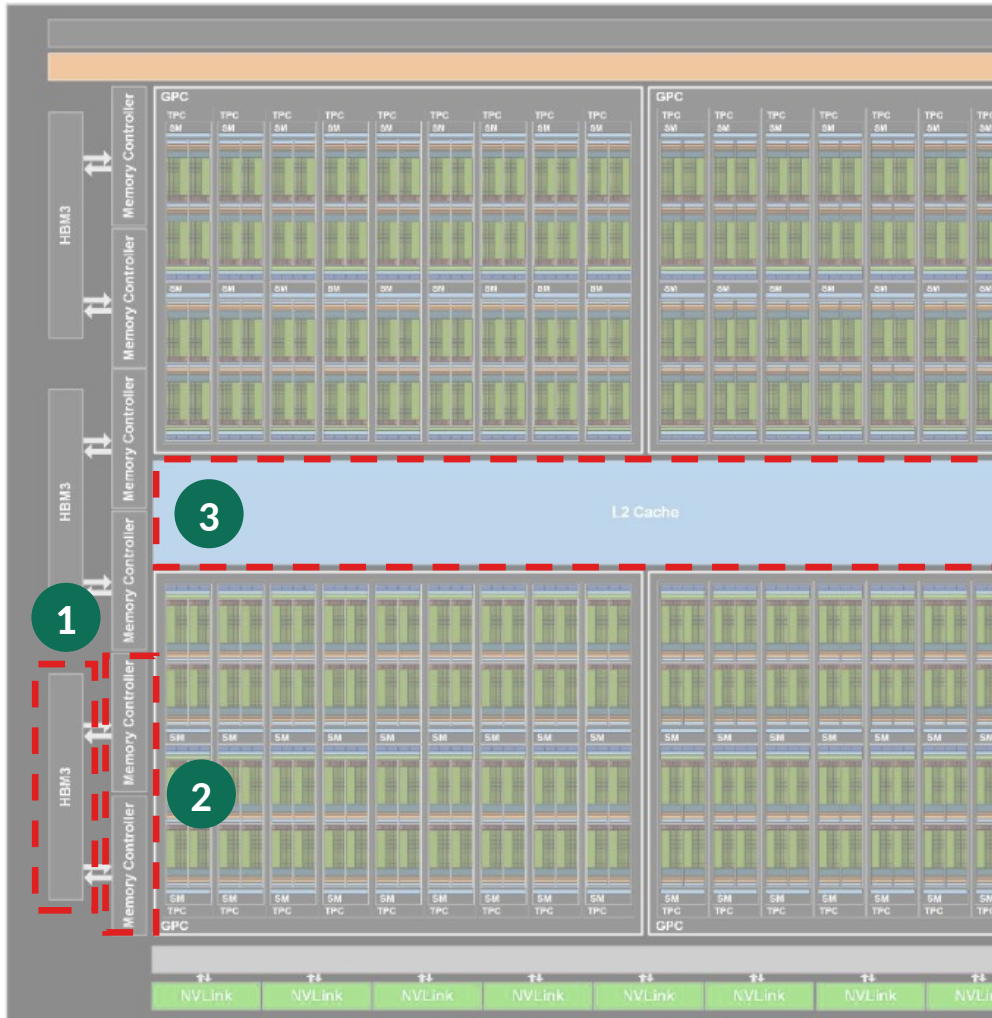
Full design vs products

Full GH100 8 GPCs · 72 TPCs · 144 SMs

Products H100 SXM 132 SMs · H100 PCIe 114 SMs

Products enable fewer SMs than the full design.

GH100: memory controllers and HBM



Data path

SM → L2 → memory controller → HBM

- 1 **HBM3** stacked high-bandwidth device memory
 - 2 **Memory controllers** 12 × 512-bit, drive the HBM stacks
 - 3 **L2 cache** one cache shared by all SMs
- Host & peer copies:** PCIe / NVLink via copy engines, not the memory controllers

Inside a Hopper SM

An SM executes our CUDA instructions

Its resources

- 1 Warp schedulers and register files
- 2 CUDA cores: INT32, FP32, FP64 units
- 3 Tensor Cores (4th generation)
- 4 Load/store and special-function units
- 5 L1 cache / shared memory: 256 KB per SM



CUDA Cores and Tensor Cores

In every Hopper SM

128

FP32 CUDA Cores

4

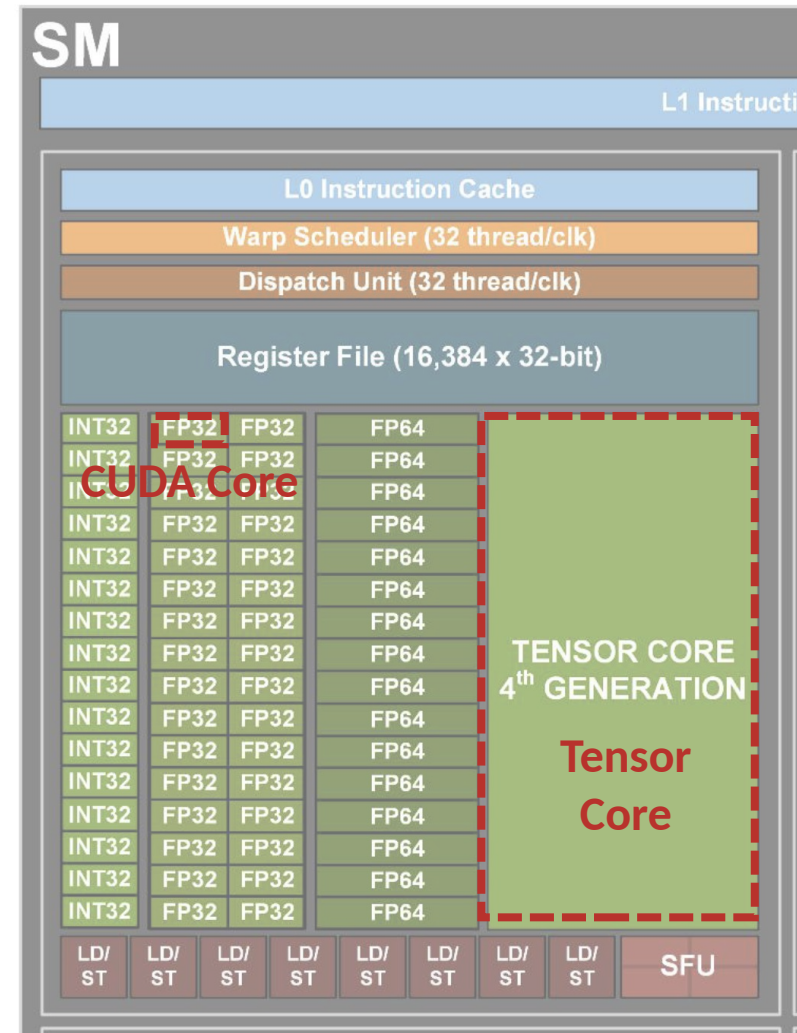
Tensor Cores (4th gen)

What they do

CUDA Cores execute scalar arithmetic instructions
 $c = a * b + c$ // one FMA on single numbers

Tensor Cores work on entire matrices with each instruction
 $D = A * B + C$ // A 16×16, B 16×8 (one mma)

Scalar math → CUDA Cores; matrix multiply → Tensor Cores.



Tensor Cores across generations

Each generation adds formats

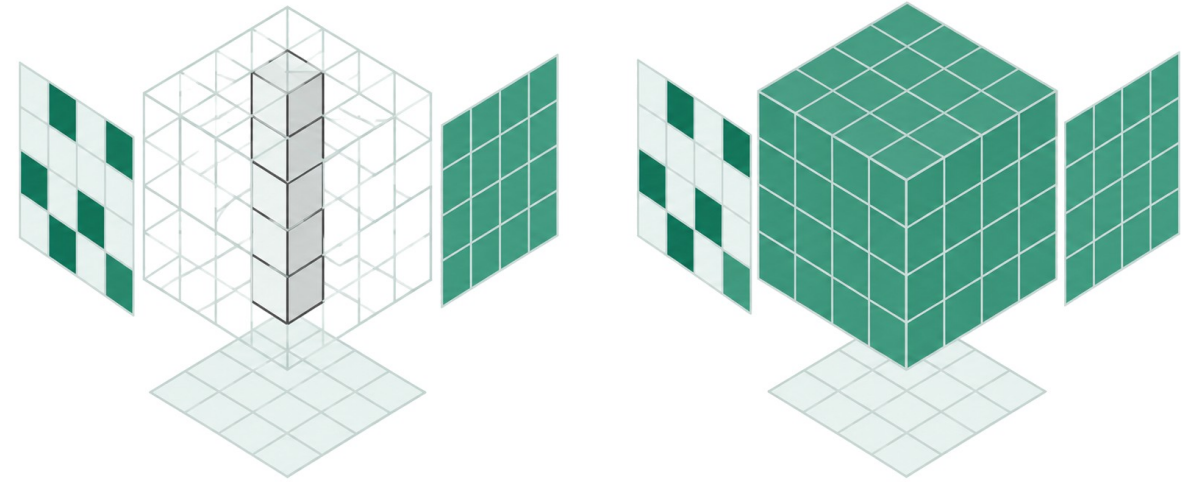
Volta (V100) FP16 multiply, FP32 accumulate

Turing INT8, INT4, INT1

Ampere (A100) FP64 MMA, BF16, TF32

Hopper (H100/H200) FP8, Transformer Engine

Blackwell FP4, 2nd-gen Transformer Engine



Pascal
scalar FMAs on CUDA Cores

Volta Tensor Core
4x4x4 MMA per clock

One instruction: $D = A \times B + C$ on a small tile

Source: NVIDIA, Introduction to GPU Computing (P. Graham); figure redrawn.

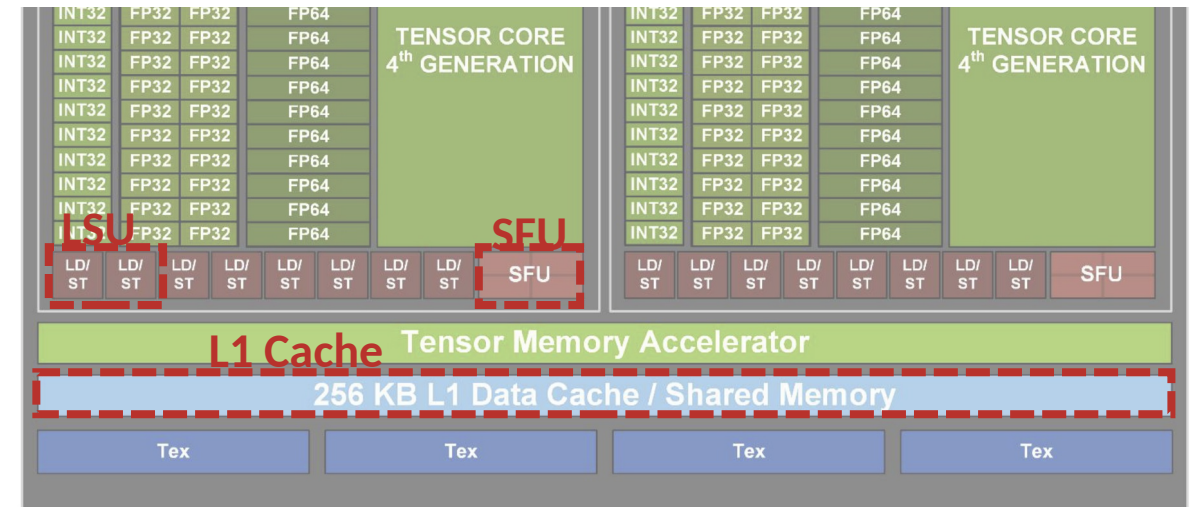
Lower precision + matrix instructions = most of the 1000× inference gain.

Special functions, loads and stores

SFU fast math functions such as exp, sin, cos

LSU sends loads and stores to the GPU memory system

L1 cache per SM; one physical array shared with shared memory (configurable split), programmed differently

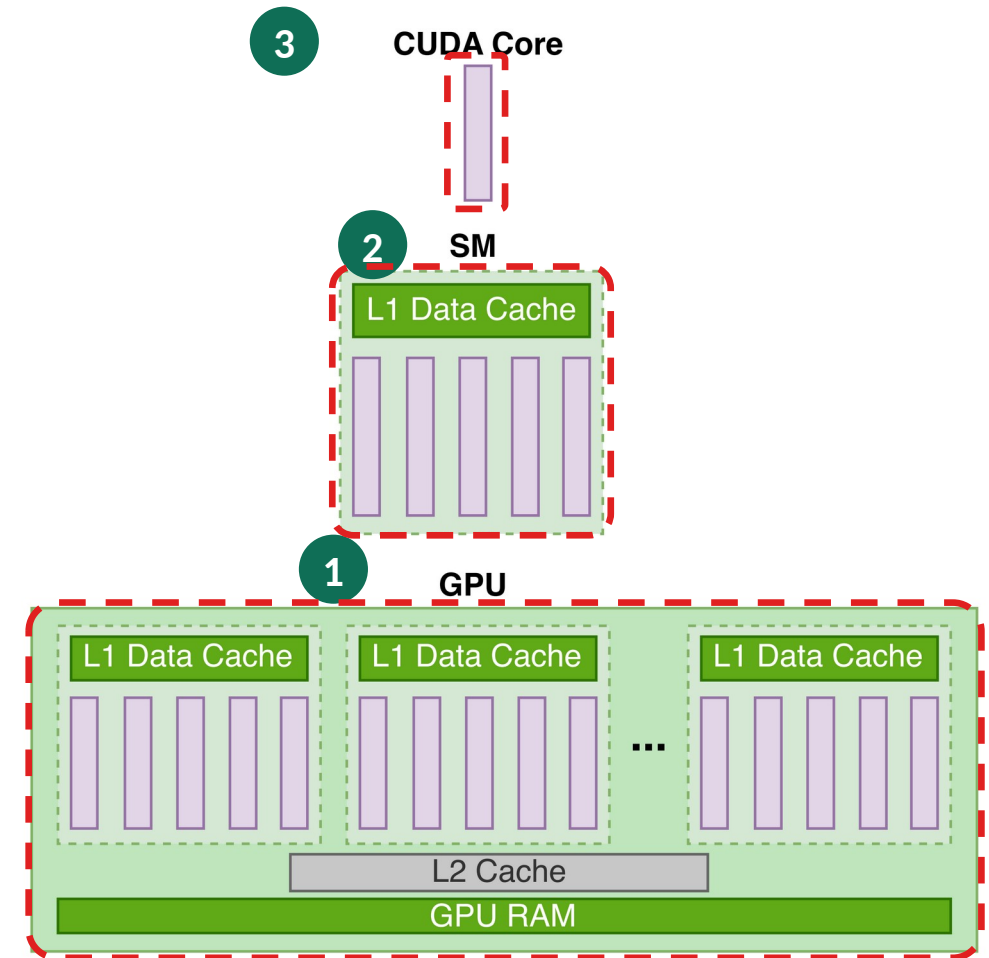


Not every instruction runs on a CUDA core: **math functions → SFUs**, **memory accesses → LSUs**

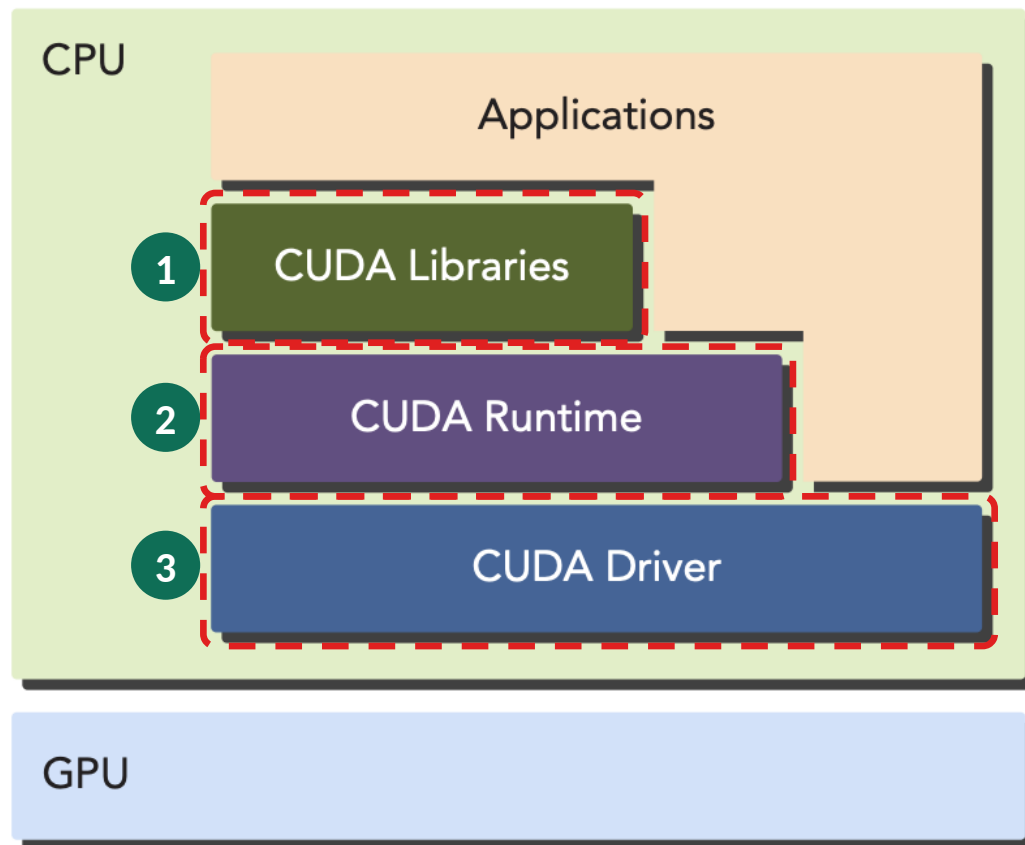
A simplified GPU hardware model

- 1 **GPU** many SMs + shared L2 cache + device DRAM
 - 2 **SM** warp schedulers, execution units, registers, shared memory / L1
 - 3 **CUDA core** executes scalar arithmetic
- Tensor Core** matrix operations inside the SM, on Volta (2017) and later GPUs

Simplified: a GPU is many SMs; each SM holds many execution units.



What is CUDA?



CUDA = NVIDIA's parallel computing platform and C++-like programming model

The software stack

- 1 **Libraries:** ready-made kernels (cuBLAS, ...)
- 2 **Runtime:** the API we call (cudaMalloc, ...)
- 3 **Driver:** loads kernels, talks to the GPU

General-purpose GPU computing: scientific computing, AI, data analysis.

A first CUDA kernel

```

__global__ void kernel() {
    printf("Block %d of %d, Thread %d of %d\n",
        blockIdx.x, gridDim.x, threadIdx.x, blockDim.x); 1
}

int main() {
    kernel<<<4, 3>>>(); 2
    cudaDeviceSynchronize(); 3
    return 0;
}

```

<<<4, 3>>> = 4 blocks × 3 threads = 12 threads

Output (order may vary)

```

Block 0 of 4, Thread 0 of 3
Block 0 of 4, Thread 1 of 3
Block 0 of 4, Thread 2 of 3
...
Block 3 of 4, Thread 2 of 3

```

1

`blockIdx.x, threadIdx.x`: which block and thread prints

2

`<<<4, 3>>>`: 4 blocks × 3 threads

3

`cudaDeviceSynchronize()`: wait for the GPU before exit

Kernel code and thread

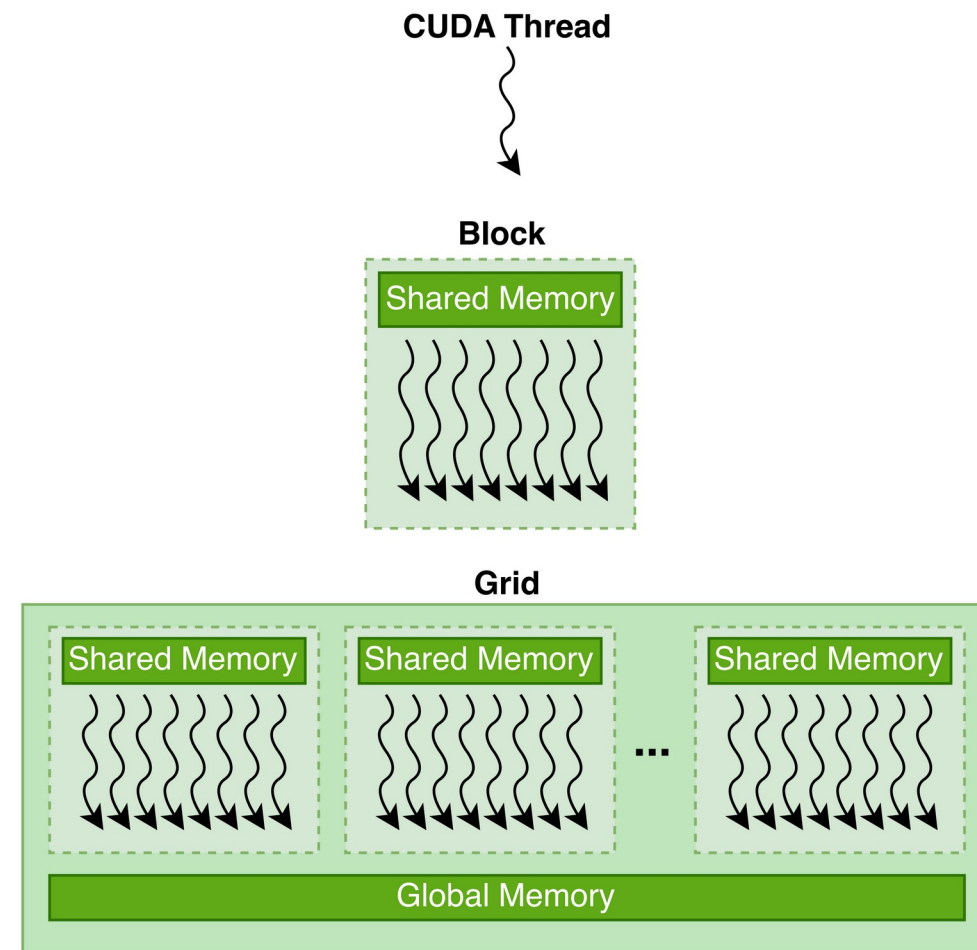
CUDA kernel

A special function that runs on the GPU

Called from CPU code, executed on the GPU

CUDA thread

A *thread of execution* (“thread”) is the lowest unit of programming for GPUs



Block and grid

Block

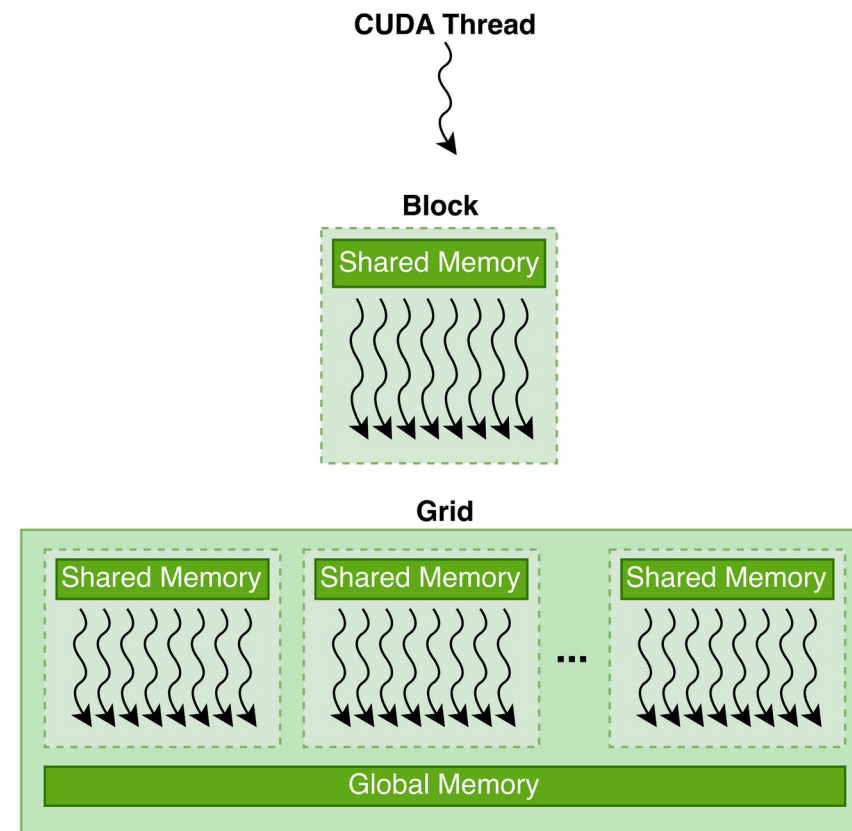
A group of warps whose threads can coordinate

Blocks run independently, in any order

Grid

One kernel launch = one grid of blocks (1D, 2D, 3D)

Matching memory level: global memory



Block and grid are both logical concepts: the GPU decides where blocks run.

Block and thread parallelism

Two levels of parallelism

blocks in a grid · threads in a block

```
kernel<<<arg1, arg2>>>();
```

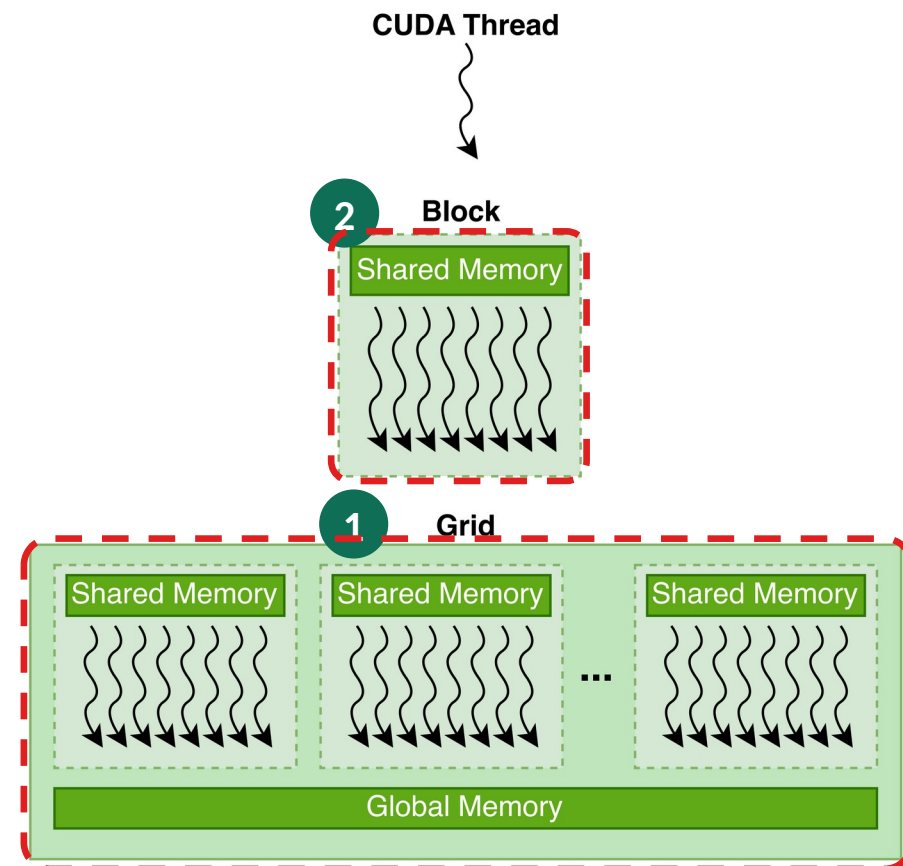
1

2

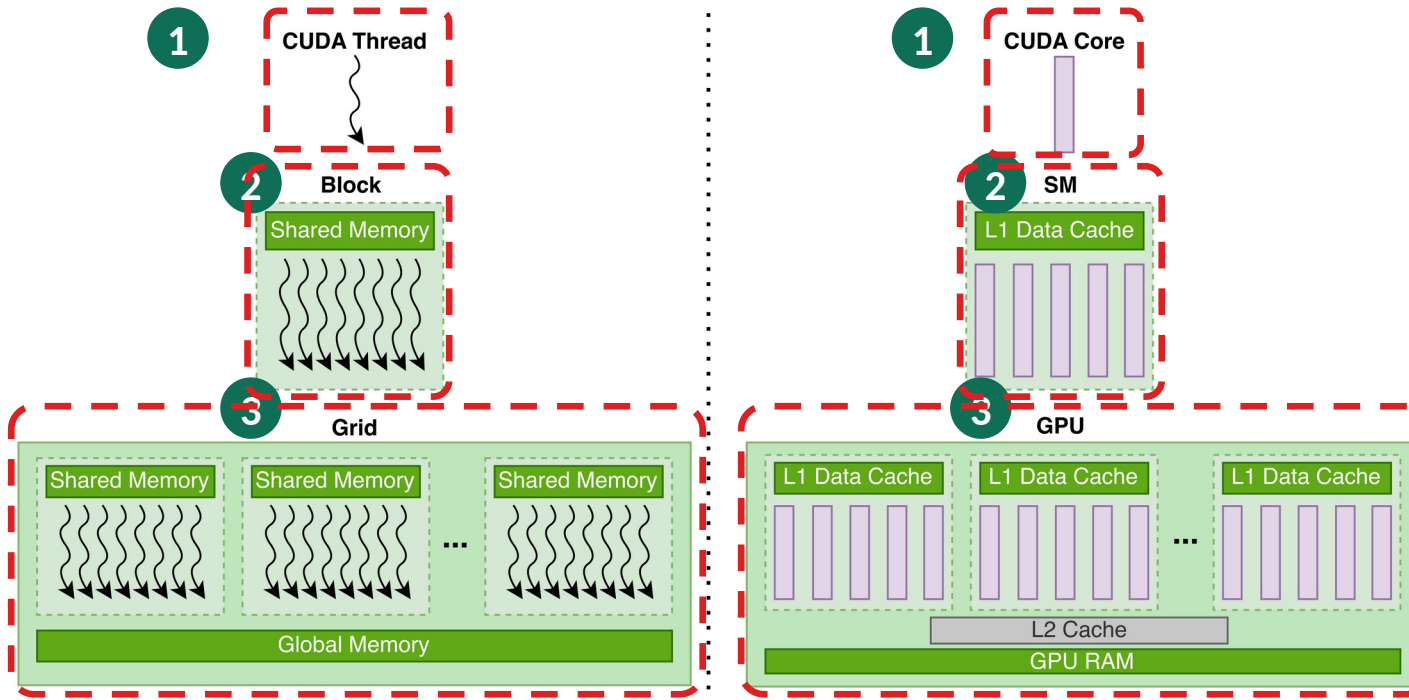
1 **arg1** = blocks in the grid = **gridDim.x**

2 **arg2** = threads per block = **blockDim.x**

Total threads = **gridDim.x** × **blockDim.x** (<<<4, 3>>> → 12)



The programming model and GPU hardware



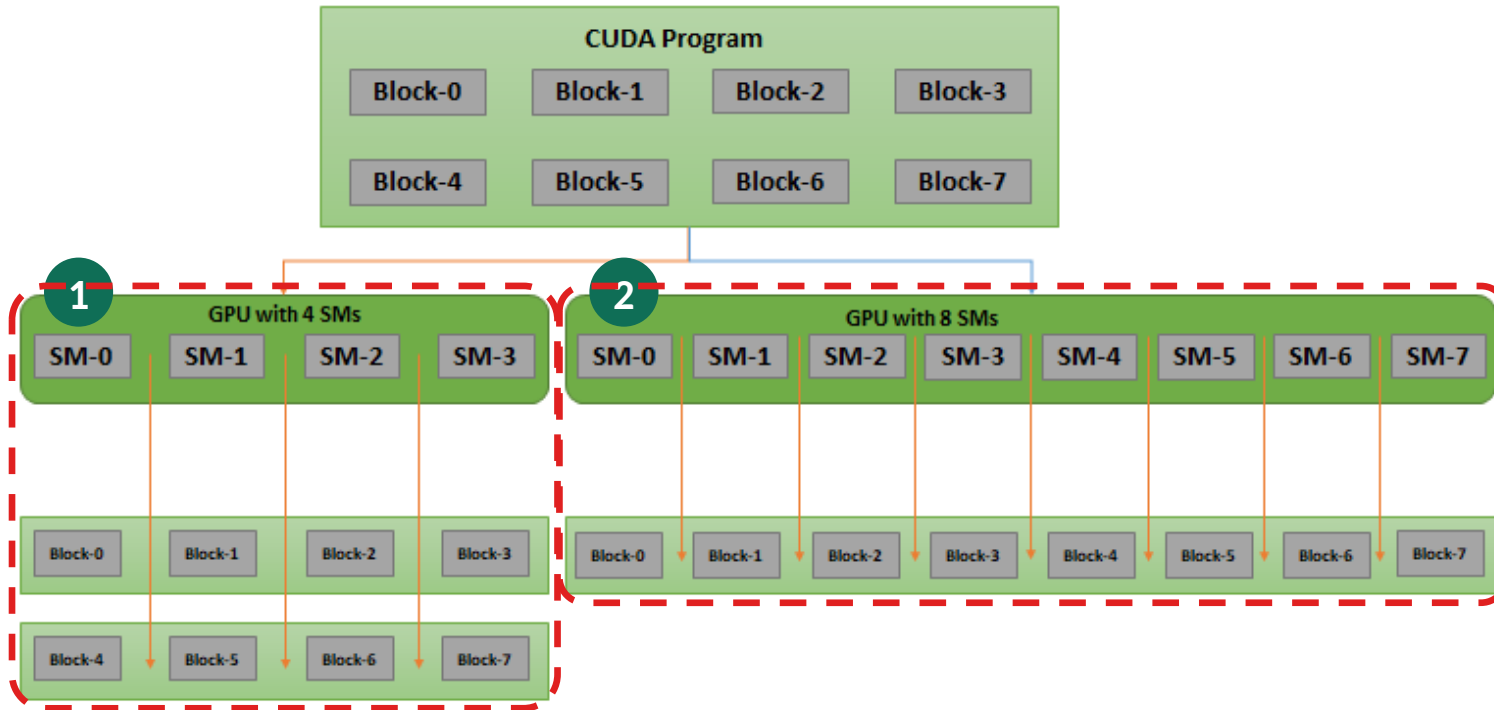
Thread ↔ core. threads run in warps of 32; they do not own fixed cores

Block ↔ SM. a block runs on one SM; an SM may host several blocks; shared memory belongs to a block

Grid ↔ GPU. a launch creates a grid of blocks spread over the GPU

You choose the grid and block sizes; the GPU decides which SM runs each block.

Block and grid: mapping blocks to SMs



1 4 SMs: 2 waves of 4 blocks

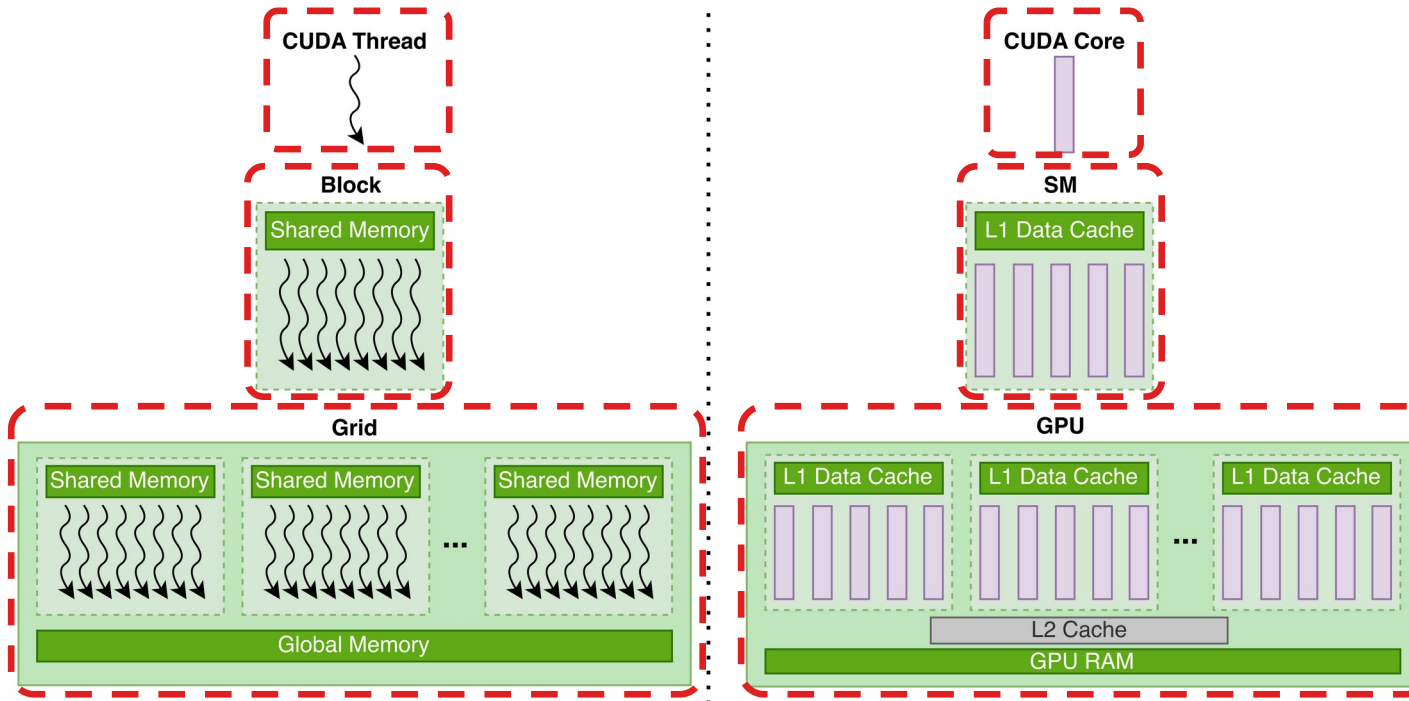
2 8 SMs: 1 wave of 8 blocks

Same kernel, same 8 blocks: the program never picks an SM

One possible assignment: real placement depends on free SMs and resources

More SMs → more blocks at once: the same code scales without changes.

Recap: the programming model and GPU hardware



1

An **SM executes** warps of 32 threads, from the blocks resident on it

2

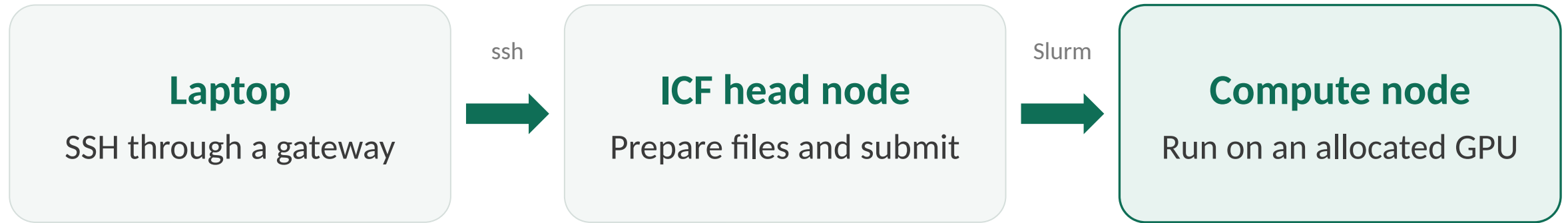
Hierarchy: a grid is made of blocks; each block is split into warps of 32 threads

3

A **block shares** its shared memory: visible to all its threads, private to the block

Logical grid → block → warp → thread runs on physical GPU → SM → cores.

Login, scheduling and GPU execution



Slurm allocates the compute node and its GPU in the **Teaching** partition

Head node. prepare files and submit jobs;
seeing no GPU here is normal

Compute node. test the GPU only after Slurm
has allocated it

Logging in and receiving a GPU allocation are separate steps.

DICE account and cluster access

Log in: two ssh hops

```
# From an approved campus/VPN network:  
1 [ssh YOUR_UUN@student.ssh.inf.ed.ac.uk]  
# On the gateway:  
2 [ssh icf.inf.ed.ac.uk]
```

- 1 Gateway student.ssh: first hop from your laptop
- 2 ICF head node: prepare files and submit jobs

Check before class

Account. Your DICE password

Course. Teaching course access

Network. Campus or University /
School VPN

Your University Login and DICE account have separate passwords.

macOS Kerberos and student login

~/.ssh/config

```
Host *.inf.ed.ac.uk
    GSSAPIAuthentication yes
    GSSAPIDelegateCredentials yes
    PreferredAuthentications gssapi-with-mic,keyboard-interactive
```

Laptop terminal

```
kinit -f YOUR_UUN@INF.ED.AC.UK
klist
ssh YOUR_UUN@student.ssh.inf.ed.ac.uk
```

On the gateway

```
ssh icf.inf.ed.ac.uk
```

Use your DICE password. Renew expired tickets with kinit.

DICE and cluster files

DICE home on the gateway

/afs/inf.ed.ac.uk/user/.../UUN

Run `echo "$HOME"` on the gateway.

AFS files stay in your DICE home.

Cluster home on icf

/home/UUN

Clone the course repository here.

Compute nodes read this directory.

On icf: get the lecture code (cuda-tutorial/)

```
git clone https://github.com/ed-aisys/edin-mls-26-autumn.git
cd edin-mls-26-autumn/cuda-tutorial
```

Compute nodes use the cluster home. Keep a backup on your laptop.

Student DICE documentation

DICE account and password

<https://opencourse.inf.ed.ac.uk/DICE-UG/accounts>

Student gateway and VPN

<https://computing.help.inf.ed.ac.uk/external-login>

macOS Kerberos ticket

<https://computing.help.inf.ed.ac.uk/kerberos-mac-os-x>

ICF access and file storage

<https://computing.help.inf.ed.ac.uk/teaching-cluster>

Slurm job commands

<https://computing.help.inf.ed.ac.uk/slurm>

Cluster files and scratch

<https://computing.help.inf.ed.ac.uk/cluster-tips>

Before class: check your account, network and Teaching course access.

A Slurm allocation for a GPU shell

```
1 sinfo -p Teaching
2 srun -p Teaching --gres=gpu:1 \
3     --cpus-per-task=1 --mem=2G \
4     --time=00:40:00 --pty bash
hostname
nvidia-smi -L
export PATH=/opt/cuda-12.8.0/bin:$PATH
nvcc --version
```

Tested: H200 MIG (saxa) and RTX 2080 Ti (opencast), CUDA 12.8.93; your allocation may use a different device

- 1 **sinfo**: partition state, not a guaranteed slot
- 2 **srun**: queue for 1 GPU and a 40 min shell
- 3 **hostname, nvidia-smi -L**: which node, which GPU
- 4 **nvcc**: CUDA 12.8 compiler on PATH

Check the compute node, GPU and CUDA compiler.



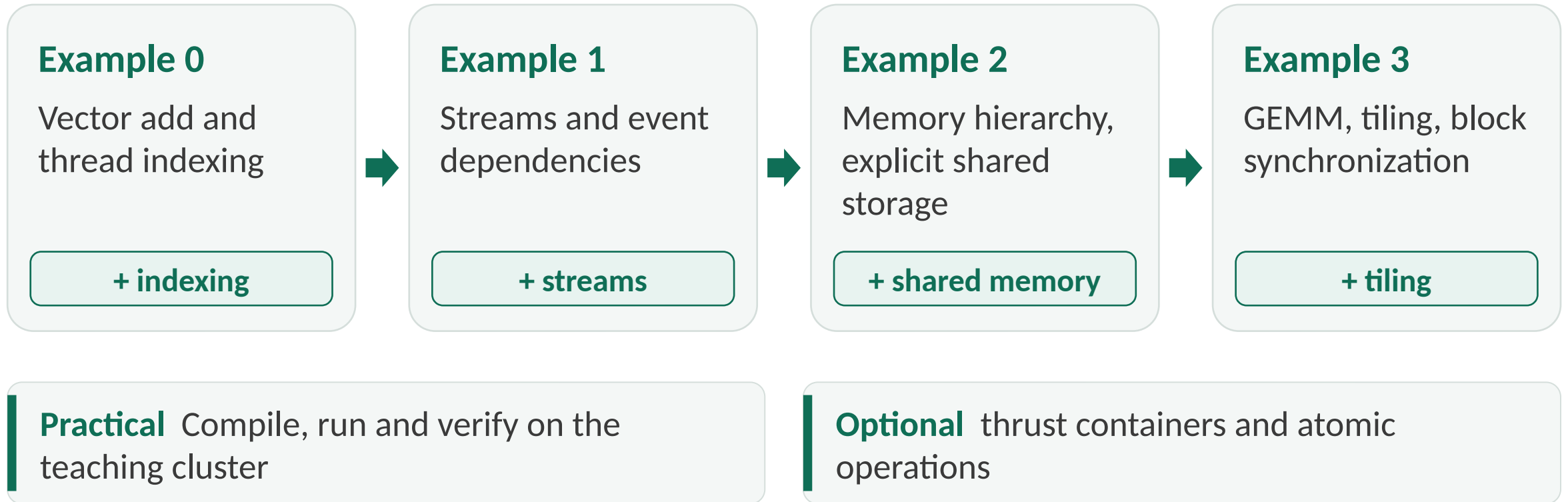
THE UNIVERSITY of EDINBURGH
informatics

CUDA Programming by Examples

Qualifiers · indexing · memory · streams · shared memory · GEMM



CUDA examples in this lecture



Each example adds one idea: indexing → streams → shared memory → tiling.

A kernel is a `__global__` function

```
1 __global__ void hello() {  
    printf("gpu hello!\n");  
}  
  
int main() {  
2     hello<<<1, 2>>>();  
3     cudaDeviceSynchronize();  
    printf("cpu hello!\n");  
}
```

Functions without `__global__` are ordinary CPU code.

Output

```
gpu hello!  
gpu hello!  
cpu hello!
```

- 1 `__global__`: runs on the GPU, launched from the CPU
- 2 `<<<1, 2>>>` = 1 block × 2 threads → two GPU lines
- 3 Launch is asynchronous: `cudaDeviceSynchronize()` waits, so the GPU lines print first

Example 0: Vector add (CUDA version)

```

CUDA Vector Add

__global__ void vector_add(const float* a, const float* b, float* c, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x; 1
    if (idx < n) { 2
        c[idx] = a[idx] + b[idx];
    }
}

// c = a + b
vector_add<<<gridSize, blockSize>>>(a, b, c, n); 3

```

One thread per element

Vector add is the canonical 1-D parallel task

a, b, c: 1-D arrays of length n

1

idx: one global index per thread

2

idx < n: spare threads skip the work

3

gridSize × **blockSize** ≥ n threads

Thread coordinates and data indices

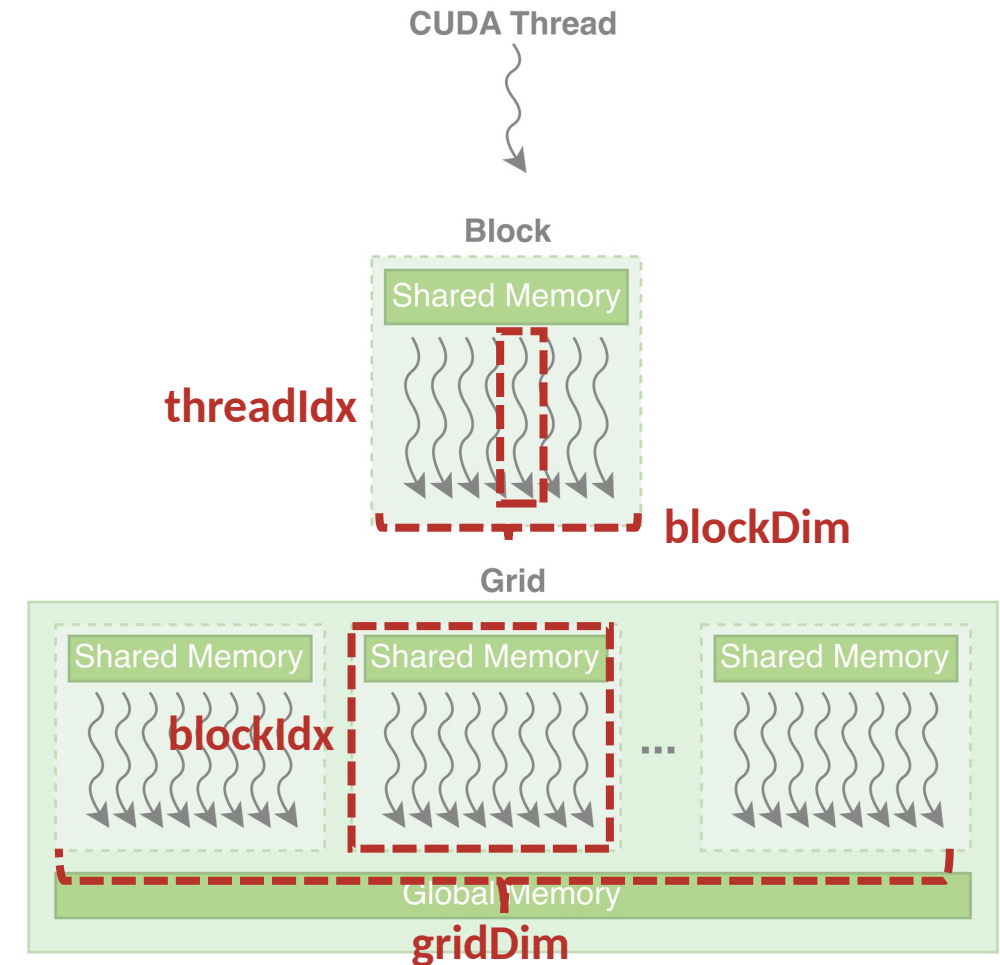
A kernel launch runs as one **grid** of blocks

gridDim blocks per grid; **blockIdx** locates a block

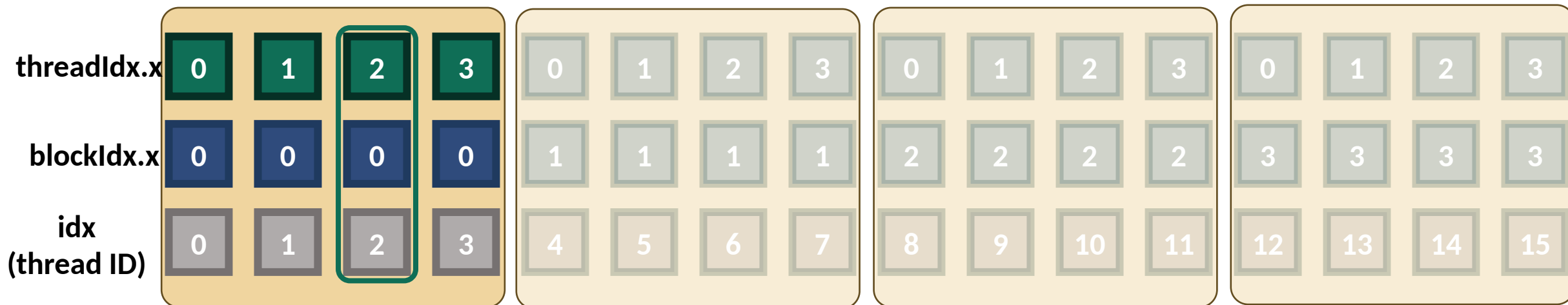
blockDim threads per block; **threadIdx** locates a thread

1-D, 2-D or 3-D: 1-D for arrays, 2-D for matrices

In a 1-D array

$$\text{idx} = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$$


Thread indices: one block



$$\text{idx} = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$$

Here: blockDim.x = 4 threads per block

Example: block 0, thread 2 $\rightarrow$ idx = $0 \times 4 + 2 = 2$

Inside one block, idx is simply threadIdx.x.

Thread indices: the whole grid

threadIdx.x	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
blockIdx.x	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3
idx	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

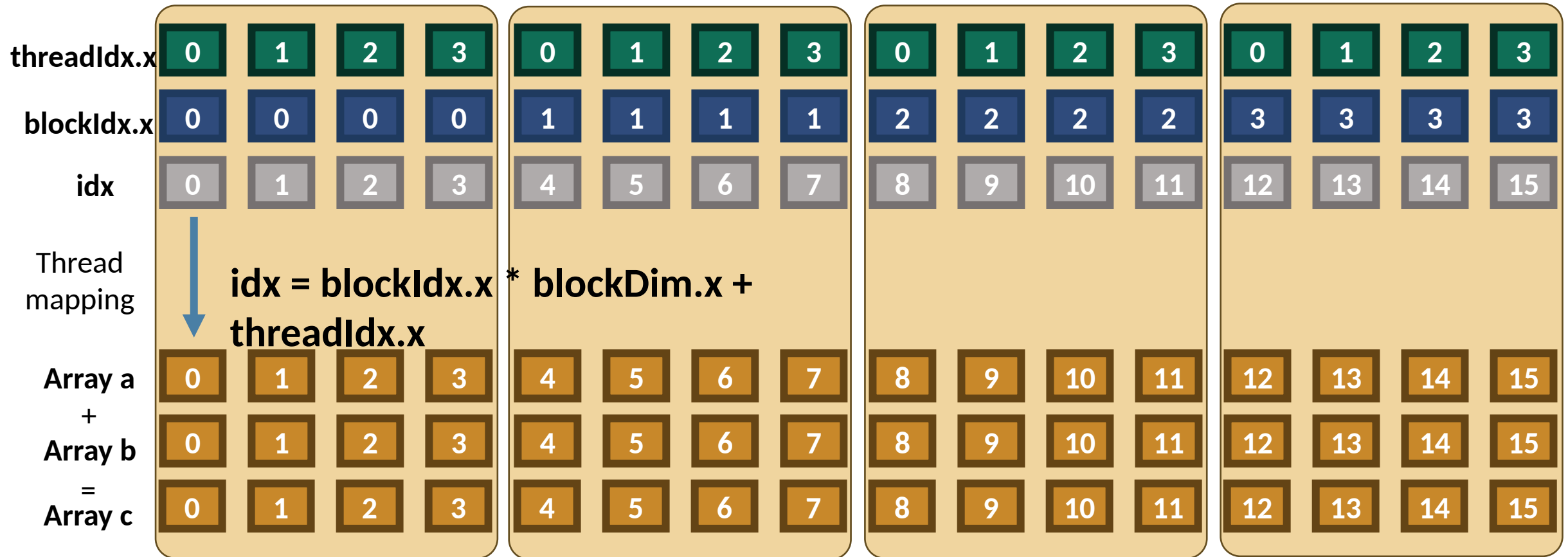
$$\text{idx} = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$$

Here: blockDim.x = 4 threads per block

Example: block 2, thread 1 → $\text{idx} = 2 \times 4 + 1 = 9$

Every thread in the grid gets its own idx: 0 ... 15.

Thread indices: mapping to data



blockDim.x fixed, chosen for convenience

gridDim.x many blocks for a large workload

Grid size, block size and residency

Grid size: how many blocks are launched

Block size: threads in each block

Too little work: leaves the GPU idle

Residency: how many blocks one SM runs at the same time

Blocks use registers, shared memory and thread slots; when the SM runs out, no more blocks fit.

More resident threads do not always mean higher performance.

Inspect limits: deviceQuery

```
Device 0: "NVIDIA GeForce GTX 1080 Ti"
CUDA Driver Version / Runtime Version      11.5 / 11.5
CUDA Capability Major/Minor version number: 6.1
Total amount of global memory:              11178 MBytes (11721506816 bytes)
(028) Multiprocessors, (128) CUDA Cores/MP: 3584 CUDA Cores
GPU Max Clock rate:                         1582 MHz (1.58 GHz)
Memory Clock rate:                          5505 Mhz
Memory Bus Width:                           352-bit
L2 Cache Size:                              2883584 bytes
Maximum Texture Dimension Size (x,y,z)      1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
Maximum Layered 1D Texture Size, (num) layers 1D=(32768), 2048 layers
Maximum Layered 2D Texture Size, (num) layers 2D=(32768, 32768), 2048 layers
Total amount of constant memory:             65536 bytes
Total amount of shared memory per block:     49152 bytes
Total shared memory per multiprocessor:      98304 bytes
Total number of registers available per block: 65536
Warp size:                                   32
Maximum number of threads per multiprocessor: 2048
Maximum number of threads per block:         1024
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)
```

NVIDIA CUDA Samples, GeForce GTX 1080 Ti; red box = the limits below

threads / SM	2048
threads / block	1024
block (x,y,z)	1024, 1024, 64
grid (x,y,z)	2 ³¹ -1, 65535, 65535

Choosing a launch configuration

256 threads = 8 warps of 32: a common starting point

Check limits: ≤ 1024 threads per block; the kernel's registers and shared memory

Then measure 128, 256, 512: the best size is found empirically

Worked example: $n = 1000$

$\text{gridSize} = (1000 + 255) / 256 = 4$

$4 \text{ blocks} \times 256 = 1024 \text{ threads} \geq n$

24 spare threads exit at if $(i < n)$

```
Grid/Block Size for Kernel

// Host memory
float *a, *b, *c;
a = (float*)malloc(size);
b = (float*)malloc(size);
c = (float*)malloc(size);

// Initialize host data
for(int i = 0; i < n; i++){
    a[i] = 1.0f;
    b[i] = 2.0f;
}

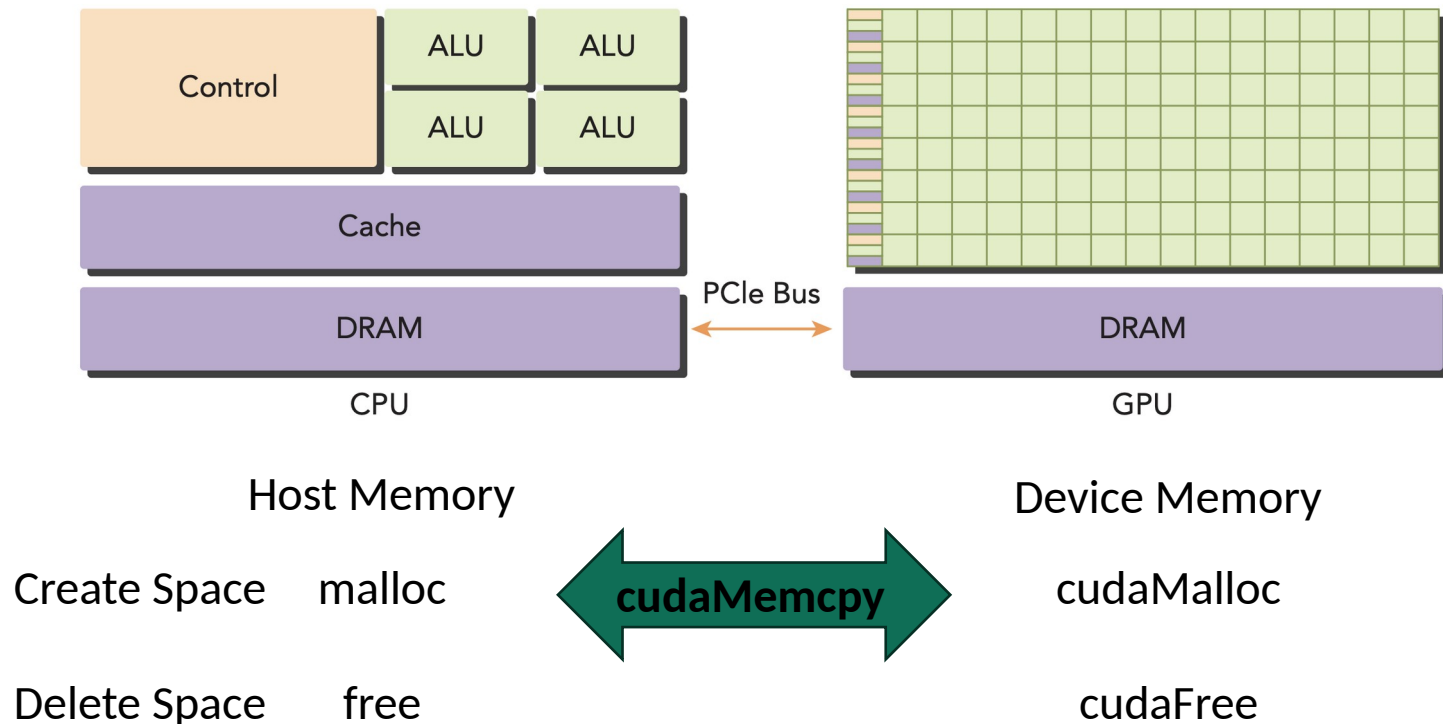
// Launch kernel
int blockSize = 256;
int gridSize = (n + blockSize - 1) / blockSize;

vector_add<<<gridSize, blockSize>>>(a, b, c, n);

// print first 10 elements of c
for(int i = 0; i < 10; i++){
    printf("%f ", c[i]);
}
printf("\n");

// Clean up
free(a); free(b); free(c);
```

Host and device memory



Output of the previous program (first 10 elements of c):

```
0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.
000000 0.000000 0.000000
```

Two separate memories

CPU DRAM and GPU DRAM, joined by the PCIe bus

This example: ordinary device allocations with `cudaMalloc`, data moved with `cudaMemcpy`

All zeros? the previous kernel got `malloc` (host) pointers, so it never wrote `c`

Other models: managed and mapped memory (next slides)

Host, device and managed memory

One virtual address space (unified addressing): the bytes still live on one side



Separate DRAM. A pointer is valid only on the side that allocated it

Explicit copy. cudaMemcpy moves bytes over PCIe/NVLink and waits for earlier work in the stream

Managed memory. One pointer for both sides; pages migrate on demand, at a time cost

Usual program shape: generate data → copy H→D → compute → copy D→H → check.

Mistake 1: a host pointer handed to a kernel

```

__global__ void kernel(int *arr) {
    arr[0] = 0;
    for (int i = 1; arr[i] != 0; ++i)
        arr[0] += arr[i];
}

1 int *a = (int*)malloc(12*sizeof(int));
  for (int i = 1; i <= 10; ++i) a[i] = i;
  a[11] = 0;                      // end marker
2 kernel<<<1, 1>>>(a);             // host pointer!
3 checkCudaErrors(
    cudaDeviceSynchronize());
printf("%d\n", a[0]);

```

Output: code=700 (cudaErrorIllegalAddress)

1

Host memory: `malloc` returns an address only the CPU can use

2

The kernel receives that host address and reads `arr[i]` on the GPU

3

The check reports error 700; without it the program prints a wrong result

Kernels need device pointers. Check errors: a silent wrong answer is worse than a crash.

Mistake 2: a device pointer used on the host

```
1  int *a;  
   [cudaMalloc(&a, 12 * sizeof(int));]  
   // a now points into GPU memory  
  
2  [for (int i = 1; i <= 10; ++i)  
   [  a[i] = i;                      // host write  
   kernel<<<1, 1>>>(a);  
3  [printf("%d\n", a[0]);] // host read
```

```
[Output: Segmentation fault (core dumped)]
```

- 1 `cudaMalloc` fills `a` with a device address
- 2 The CPU writes through that address: segmentation fault
- 3 Reading `a[0]` on the host fails too: copy it back with `cudaMemcpy` first

Device pointers go to kernels and `cudaMemcpy`; never dereference them on the host.

Explicit copies with cudaMemcpy

```
size_t bytes = 12 * sizeof(int);
int *a = (int *)malloc(bytes);
for (int i=1; i<=10; ++i) a[i]=i;
a[11] = 0; // end marker
1 int *d_a; cudaMalloc(&d_a, bytes);
2 cudaMemcpy(d_a, a, bytes,
             cudaMemcpyHostToDevice);
3 kernel<<<1, 1>>>(d_a);
4 cudaMemcpy(a, d_a, bytes,
             cudaMemcpyDeviceToHost);
printf("%d\n", a[0]);
free(a); cudaFree(d_a);
```

Output: 55

- 1 A separate device buffer d_a
- 2 Copy host → device before the launch
- 3 The kernel only ever sees d_a
- 4 Copy device → host; it waits for the kernel, no extra sync

Every byte that crosses the bus is visible and under your control.

Managed memory with cudaMallocManaged

```
size_t bytes = 12 * sizeof(int);  
int *a;  
1 cudaMallocManaged(&a, bytes);  
2 for (int i=1; i<=10; ++i) a[i]=i;  
  a[11] = 0; // end marker  
  kernel<<<1, 1>>>(a);  
3 cudaDeviceSynchronize();  
  printf("%d\n", a[0]);  
  cudaFree(a);
```

Output: 55

- 1 One pointer, valid on both the host and the device
- 2 The host writes directly; pages migrate to the GPU on first touch
- 3 The launch is asynchronous: synchronize before the host reads

Less code, but the copies are hidden: page migration costs time.

Vector add: the complete host path

1	Allocate host and device buffers	<pre>float *h_a, *h_b, *h_c; h_a = (float*)malloc(size); h_b = (float*)malloc(size); h_c = (float*)malloc(size); float *d_a, *d_b, *d_c; cudaMalloc(&d_a, size); cudaMalloc(&d_b, size); cudaMalloc(&d_c, size);</pre>
2	Copy inputs to the GPU	<pre>cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice); cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);</pre>
3	Launch [n/256] blocks of 256 threads	<pre>int blockSize = 256; int gridSize = (n + blockSize - 1) / blockSize; vector_add<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);</pre>
4	Copy results back and check them	<pre>cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost); // print first 10 elements of h_c</pre>
5	Free both sides	<pre>cudaFree(d_a); cudaFree(d_b); cudaFree(d_c); free(h_a); free(h_b); free(h_c);</pre>

Kernel submission and completion

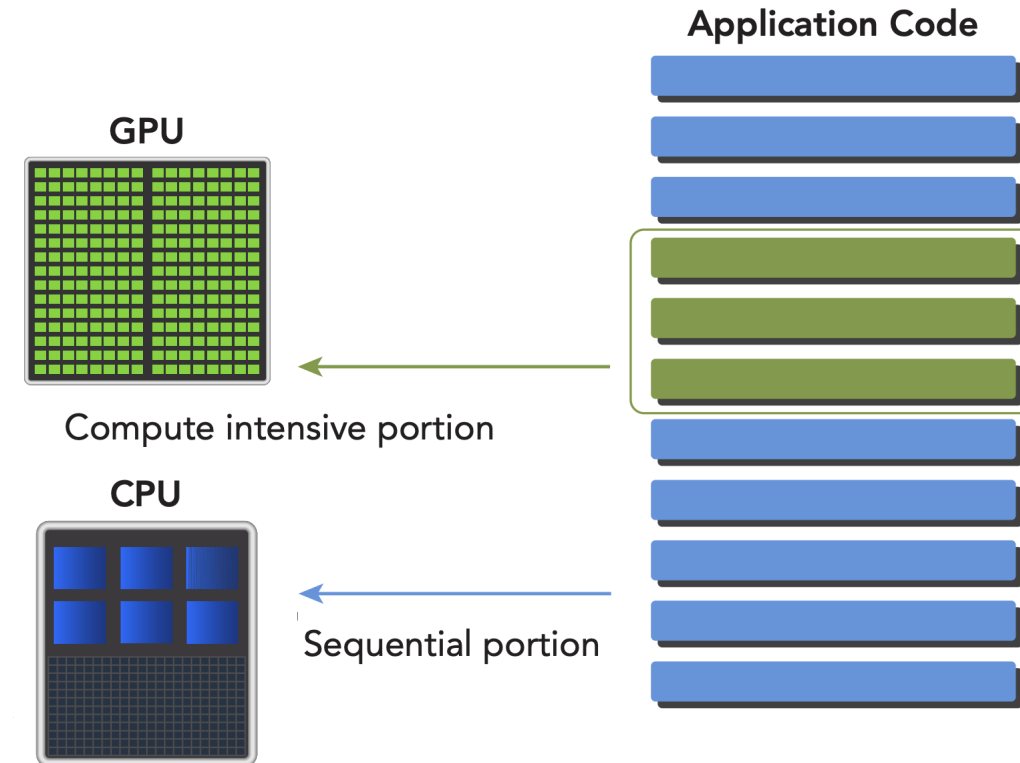
A kernel is **called by the CPU** but **runs on the GPU**

The launch is **asynchronous**: the CPU continues without waiting

CPU code may run ahead before the GPU has finished its work

`cudaDeviceSynchronize()`
blocks the CPU until all GPU work is done

Launch, then synchronize before the CPU reads GPU results.



Student checkpoint: vector add

Build and run on the GPU node, in `cuda-tutorial/2-vectoradd`

```
nvcc -O2 -std=c++17 -arch=native \  
    vector_add.cu -o vector_add  
./vector_add 1003 256  
./vector_add 257 128
```

Expected output

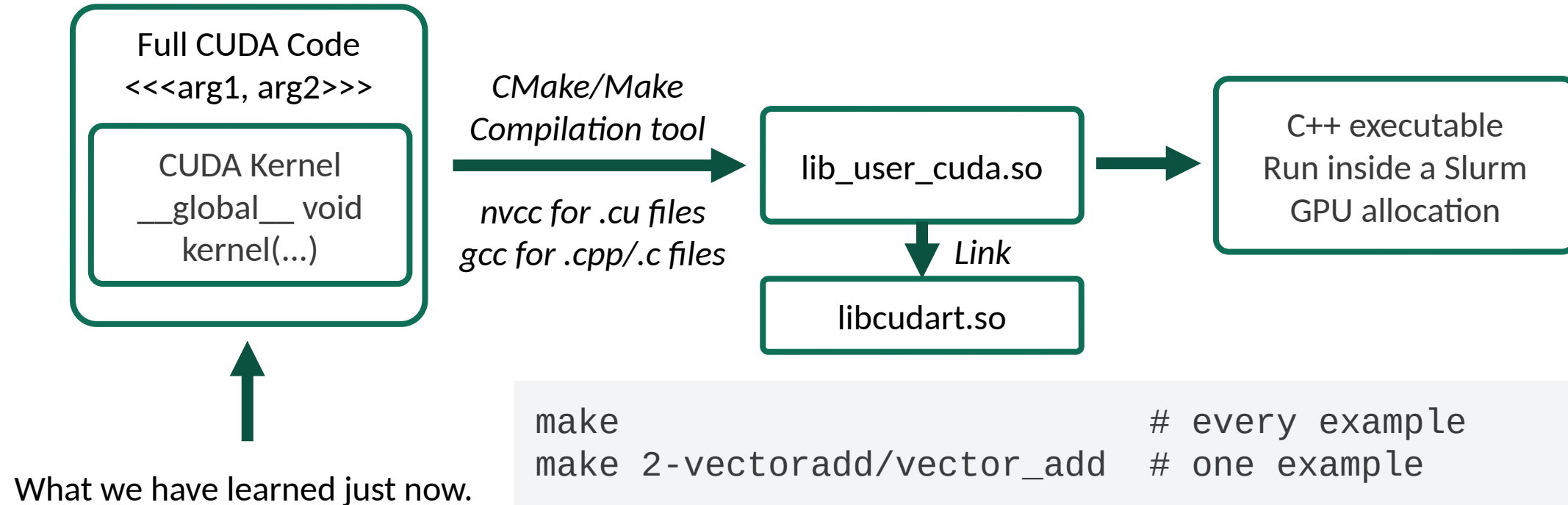
```
n=1003 block=256 grid=4  
checked=1003 failures=0  
PASS vector add
```

Then

- 1 Record node and GPU
- 2 Explain the bounds check
- 3 Exit the GPU shell

Why do both configurations produce the same answer?

Compiling a CUDA program

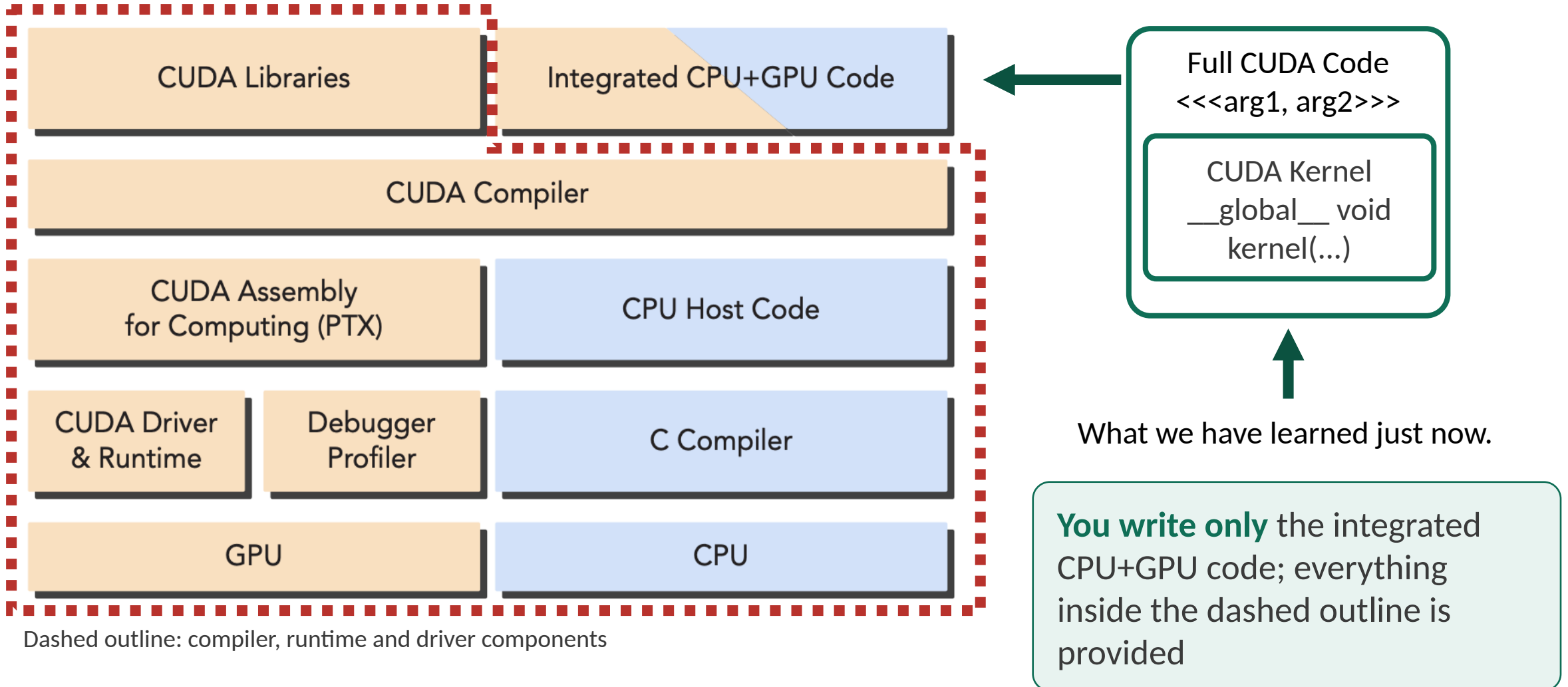


Build files. Our Makefile runs nvcc with the flags and the GPU target

Verify. Check the kernel launch and synchronize before reading host results

Compile with nvcc → link the CUDA runtime → run on an allocated GPU node.

CUDA compilation and execution



Vector add: CPU and CUDA versions

CPU: one thread loops over i

```
void add_cpu(const float* a,
             const float* b,
             float* c, int n) {
  1  for (int i = 0; i < n; ++i) {
        c[i] = a[i] + b[i];
    }
  3  add_cpu(a, b, c, n);
}
```

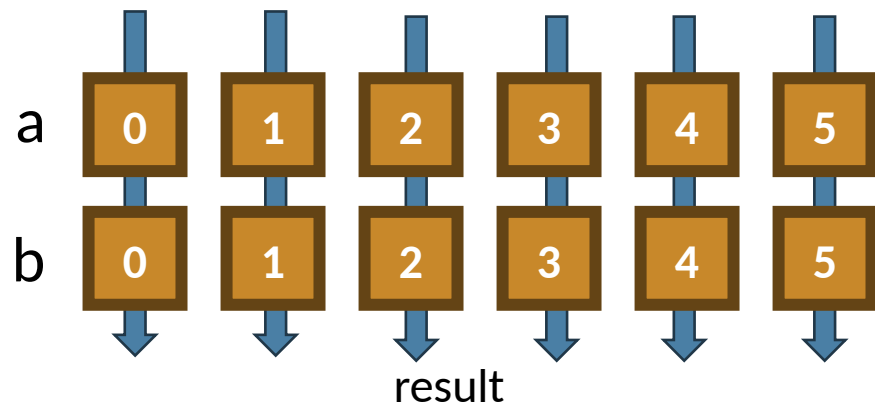
CUDA: one thread per element

```
__global__ void add(const float* a,
                    const float* b,
                    float* c, int n) {
  1  int i = blockIdx.x * blockDim.x
        + threadIdx.x;
  2  if (i < n) c[i] = a[i] + b[i];
}
  3  add<<<(n+255)/256, 256>>>(a, b, c, n);
}
```

- 1 The for loop becomes the grid: each thread computes its own *i*
- 2 `if (i < n)`: the last block is partial, spare threads skip the work
- 3 The host submits the kernel with `<<<blocks, threads>>>` and the same arguments

Common operators on GPU

Elementwise



Same operator on every item, independently

```
result[i] = a[i] + b[i]  
result[i] = a[i] * k + a[i] * a[i]
```

Reduction



Many inputs combine into one output value

```
result = max(a[i])  
result = sum(a[i])
```

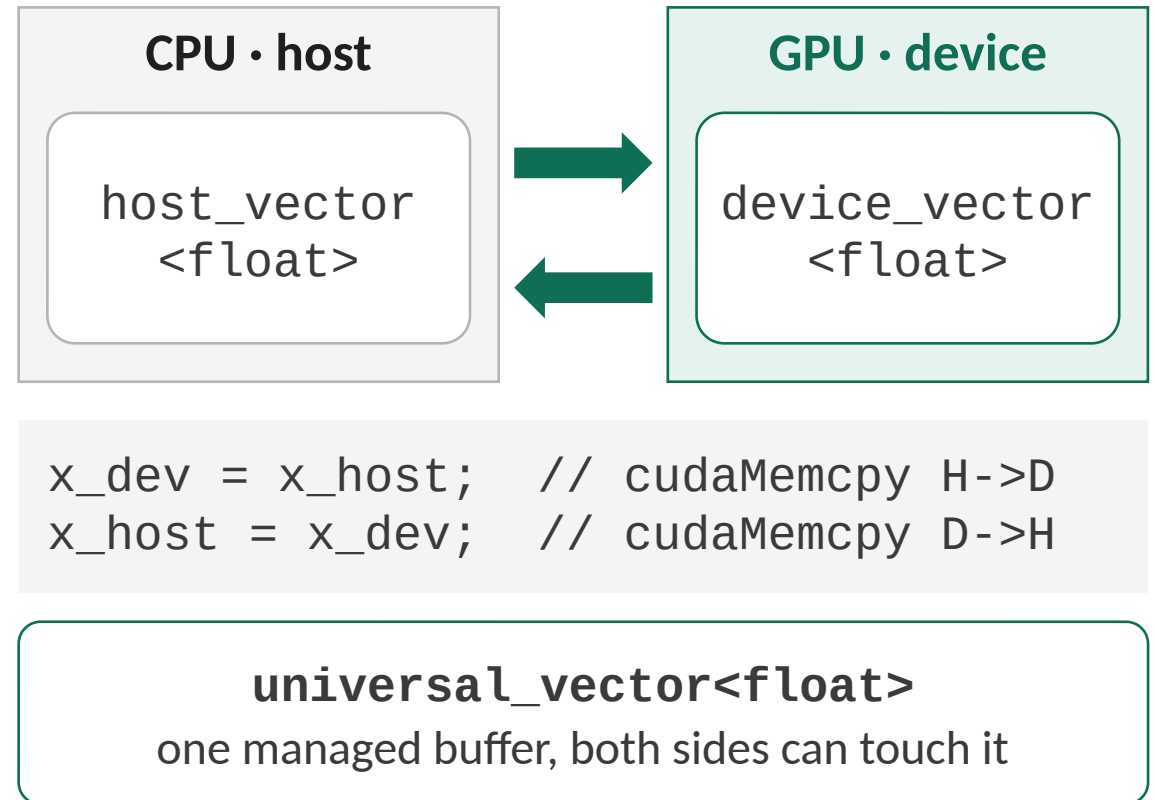
Elementwise: one thread per item. Reduction: threads must combine partial results.

Optional: thrust, a template library for CUDA

Why. STL containers do not work in device code; thrust is the STL-style replacement shipped with the toolkit (CCCL)

Algorithms. Mirror `<algorithm>`: generate, transform, sort, reduce, count_if, ...

Kernels. Still take raw pointers:
`raw_pointer_cast(v.data())`



thrust algorithms such as generate take lambdas, so passing operators to the GPU matters.

Docs: nvidia.github.io/cccl/thrust

Optional: thrust example, vector add

Grid-stride kernel and host data

1

```
template <class F>
__global__ void kernel(int n, F f) {
    int i = blockIdx.x * blockDim.x
        + threadIdx.x;
    int step = blockDim.x * gridDim.x;
    for (; i < n; i += step) f(i);
}

int n = 65536, block = 128;
int grid = (n + block - 1) / block;
thrust::host_vector<float> x_h(n);
thrust::host_vector<float> y_h(n);
// fill both with thrust::generate
```

Output: 752851072.000000

Copy in, launch, copy back

2

```
thrust::device_vector<float>
    x_d = x_h, y_d = y_h; // H -> D
kernel<<<grid, block>>>(n,
    [x = x_d.data(), y = y_d.data()]
    __device__ (int i) {
        x[i] = x[i] + y[i];
    });
cudaDeviceSynchronize();
x_h = x_d; // D -> H
printf("%f\n", x_h[0]);
```

1 Grid-stride loop: one kernel serves any n

2 Device lambda needs - - extended - lambda

Two copies, one kernel: the same vector add as Example 0 without a single cudaMalloc.

Example 1: CUDA streams

What is a CUDA stream?

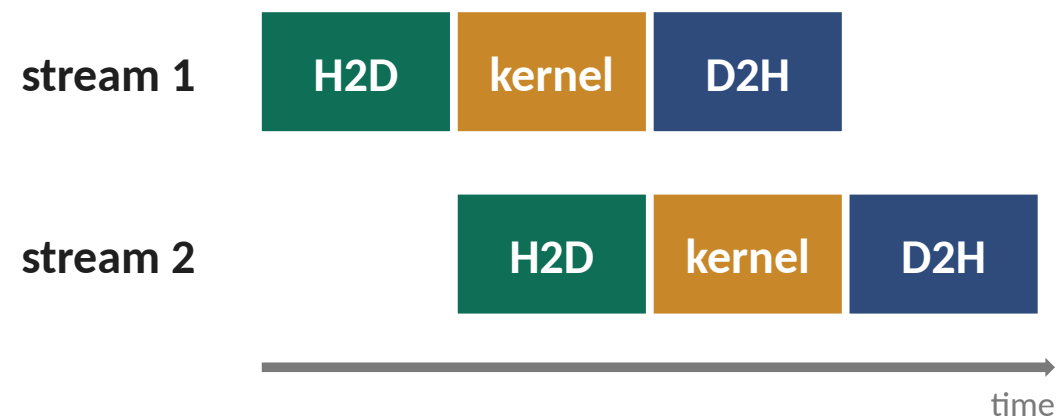
A queue of GPU operations that run in the order they are issued

When we need streams

Concurrent kernels. independent kernels in different streams can run at the same time

Pipelining. overlap data transfers with computation

Two streams over time



H2D / D2H = copy host → device / device → host. Each stream runs its own work in order; stream 2's copy overlaps stream 1's kernel.

Within a stream: in order. Across streams: no ordering, so work can overlap.

[Source: NVIDIA, CUDA Streams and Concurrency webinar \(slide 1\)](#)

A stream async example

```

x = x1 * x1      # step 1
y = x2 * x2      # step 2
z = x * sin(x1)  # step 3
w = y * cos(x2)  # step 4
z = z + w        # step 5

```

- A** | **Chain A.** steps 1 → 3 need only x1
- B** | **Chain B.** steps 2 → 4 need only x2
- 5** | **Join.** step 5 waits for both z and w

A and B can run concurrently: one stream each

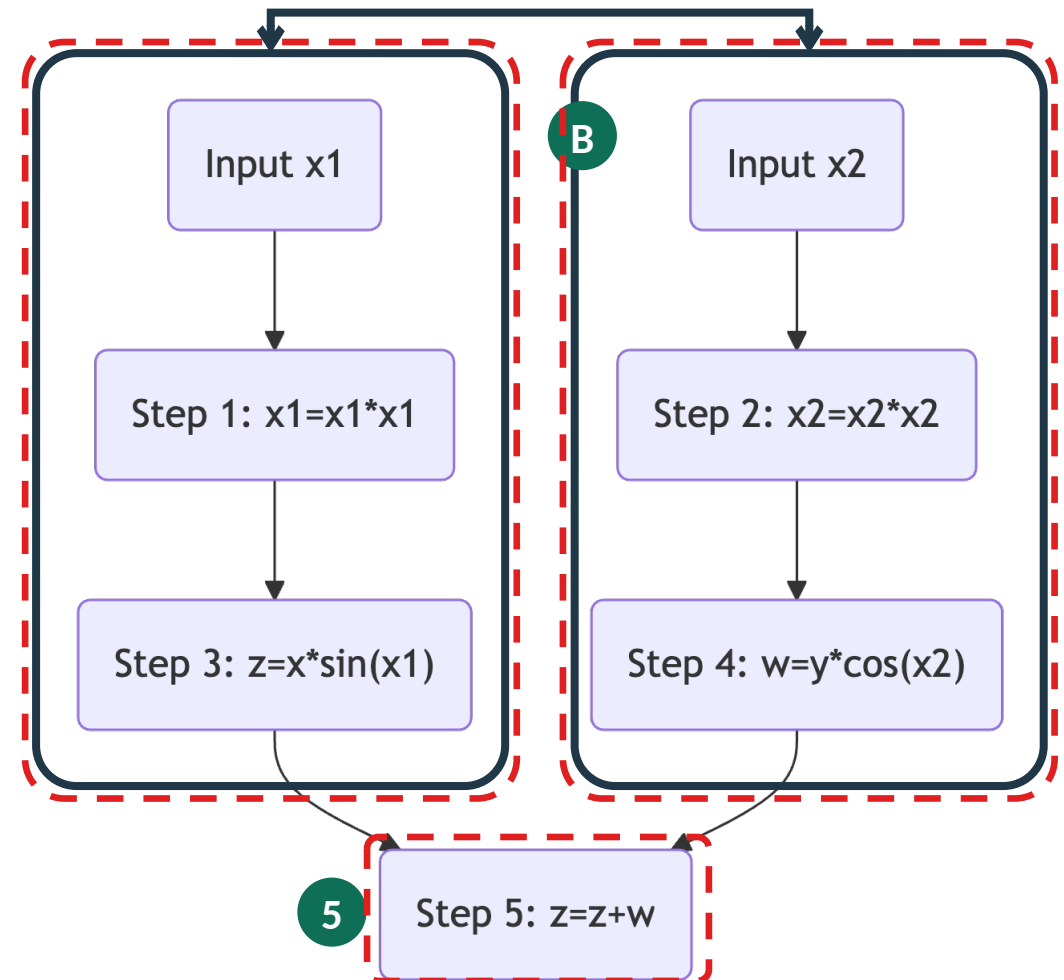


Figure: $x1=x1*x1$ means $x = x1*x1$, $x2=x2*x2$ means $y = x2*x2$

CUDA streams: independent branches

```

1  cudaStream_t s1, s2;
   cudaStreamCreate(&s1);
   cudaStreamCreate(&s2);
A  square<<<g, b, 0, s1>>>(x1, x, n);
B  square<<<g, b, 0, s2>>>(x2, y, n);
A  left<<<g, b, 0, s1>>>(x1, x, z, n);
B  right<<<g, b, 0, s2>>>(x2, y, w, n);

```



1 **Create.** one in-order queue per chain

A **Chain A.** steps 1 → 3 are queued in s1

B **Chain B.** steps 2 → 4 are queued in s2

The 4th launch argument picks the stream
 <<<grid, block, smem, stream>>>

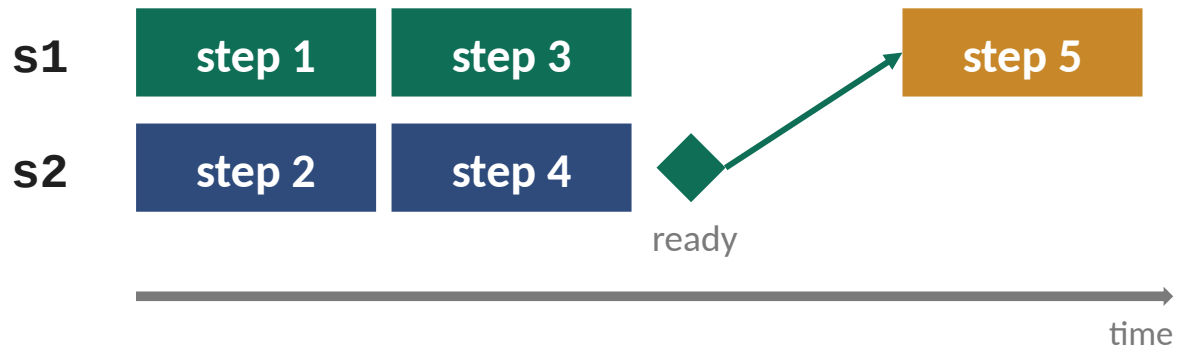
Same stream: in order. Different streams: no ordering, so they may overlap.

CUDA streams: joining the branches

```

cudaEvent_t ready;
cudaEventCreate(&ready);
1 cudaEventRecord(&ready, s2);
2 cudaStreamWaitEvent(s1, ready, 0);
3 combine<<<g, b, 0, s1>>>(z, w, n);
4 cudaStreamSynchronize(s1);
checkCudaErrors(cudaGetLastError());
// Check every API return value.

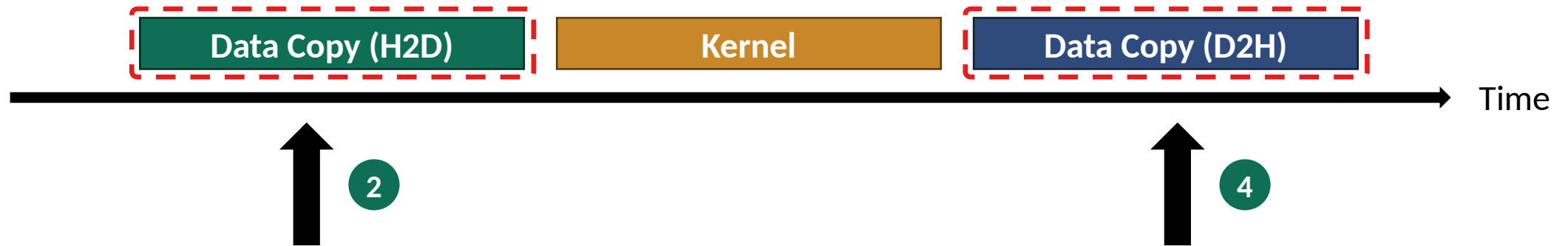
```



- 1 **Record.** mark the end of step 4 in s2
- 2 **Wait.** s1 waits for ready on the GPU; the host does not block
- 3 **Combine.** step 5 sees both z and w
- 4 **Sync.** host waits for s1, which now covers both chains

Streams are unordered with each other: an event is how one stream waits for another.

Copies and computation



Programming pattern on CUDA

- 1 Find the sections that can run in parallel
- 2 **Copy** the input data to the GPU
- 3 Run the CUDA kernel
- 4 **Copy** the results back to the CPU

The GPU sits idle during steps 2 and 4
It waits for data. Run the kernel many times and the idle time adds up.

Fix: overlap copies with computation (next slide).

Copies and computation across batches

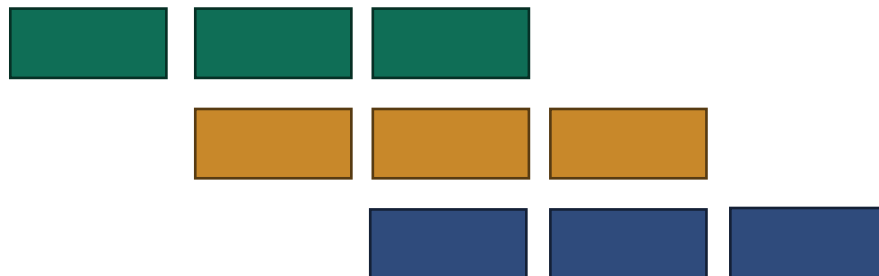
Sequential



Time

Cut the task into small batches
copy and compute part by part

Pipelined



Time

Batch k+1 copies in while batch k computes

time saved

CUDA stream ordering for batched copies

```
1 cudaMemcpyAsync(d_in, h_in, bytes,  
    cudaMemcpyHostToDevice, s);  
2 kernel<<<grid, block, 0, s>>>(  
    d_in, d_out, n);  
3 cudaMemcpyAsync(h_out, d_out, bytes,  
    cudaMemcpyDeviceToHost, s);  
4 cudaStreamSynchronize(s);
```

- 1 queues the copy, returns at once
- 2 kernel runs after copy 1
- 3 copy back runs after the kernel
- 4 host waits once for the batch

Many batches → overlap

One stream per batch. batch k+1's copy can overlap batch k's kernel

Pinned host memory. `cudaMallocHost` buffers, so async copies really run asynchronously

Order within a batch comes from the stream; overlap comes from using several streams.

Example 2: matrix-vector multiplication

1	2	3
4	5	6
7	8	9

*

10	20	30
----	----	----

=

1*10	2*20	3*30
4*10	5*20	6*30
7*10	8*20	9*30

=

10	40	90
40	100	180
70	160	270

Shapes. A: $M \times K$, x: K , y: M

One output = one row $\cdot$ x: $y[0] = 1 \cdot 10 + 2 \cdot 20 + 3 \cdot 30 = 140$, $y = (140, 320, 500)$

CPU reference

```

1  for (int row = 0; row < M; ++row) {
    float sum = 0.0f;
    2  for (int k = 0; k < K; ++k)
        sum += A[row*K+k] * x[k];
    y[row] = sum;
  }

```

1 Outer loop: one output per row

2 Inner loop: dot product: row of A $\cdot$ x

Matrix-vector multiplication on CUDA

```
__global__ void matvec(  
    const float* A, const float* x,  
    float* y, int M, int K) {  
1   int row = blockIdx.x * blockDim.x  
    + threadIdx.x;  
2   if (row >= M) return;  
    float sum = 0.0f;  
    for (int k = 0; k < K; ++k)  
3       sum += A[row*K+k] * x[k];  
    y[row] = sum;  
}
```

- 1 one thread computes one output row
- 2 spare threads in the last block exit
- 3 each thread reads its row of A and **all** of x

Reused input: x

every output row reads all of x; each element of A is read once

Matrix-vector indexing and work assignment

```
// Grid-stride loop over rows
1 int stride = blockDim.x * gridDim.x;
  int first = blockIdx.x * blockDim.x
    + threadIdx.x;
2 for (int row = first; row < M;
   row += stride) {
3   float sum = 0.0f;
   for (int k = 0; k < K; ++k)
     sum += A[row*K+k] * x[k];
   y[row] = sum;
}
```

1

stride = number of threads in the whole grid

2

rows first, first + stride, ...: any grid size covers all M rows

3

every row still reads all of x from global memory

New work assignment $\neq$ data reuse

Shared memory must be loaded explicitly (next slide).

Matrix-vector multiplication with input reuse

```

__shared__ float sx[TILE];
// for each tile of x (all threads):
1  sx[threadIdx.x] = k < K ? x[k] : 0.0f;
2  __syncthreads();
3  if (row < M) {
    // accumulate the valid tile into sum
  }
4  __syncthreads();
// next tile only after readers finish

```

1

Load. every thread loads one element of x , even if $\text{row} \geq M$

2

Sync. wait until the whole tile is in shared memory

3

Use. only valid rows accumulate (no early return)

4

Sync again. do not overwrite the tile while others still read it

Reuse: a block reads each tile of x from global memory once; all its rows then read it from shared memory

Cache and shared memory

Cache	Shared memory
Hardware manages reuse	The program loads and indexes the data
Loads may hit or miss	Threads explicitly cooperate
Replacement is a hardware decision	The program controls when a buffer is reused

Shared memory reuse needs explicit loads and correct synchronization.

Input reuse: a cache analogy (1)

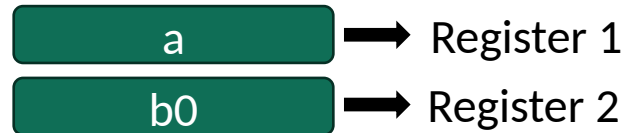
Cache model: holds three vectors

Step 1: load a , $b[0]$ — a miss, $b[0]$ miss

Cold cache: both vectors come from memory

Step 1

Cache



Cost = 2 * load time

```
for (int row = 0; row < M; ++row)
    y[row] = dot(A + row*K, x, K);
```

a = x : read by every row (reused)

b[i] = row i of A : read once

Input reuse: a cache analogy (2)

Cache model: holds three vectors

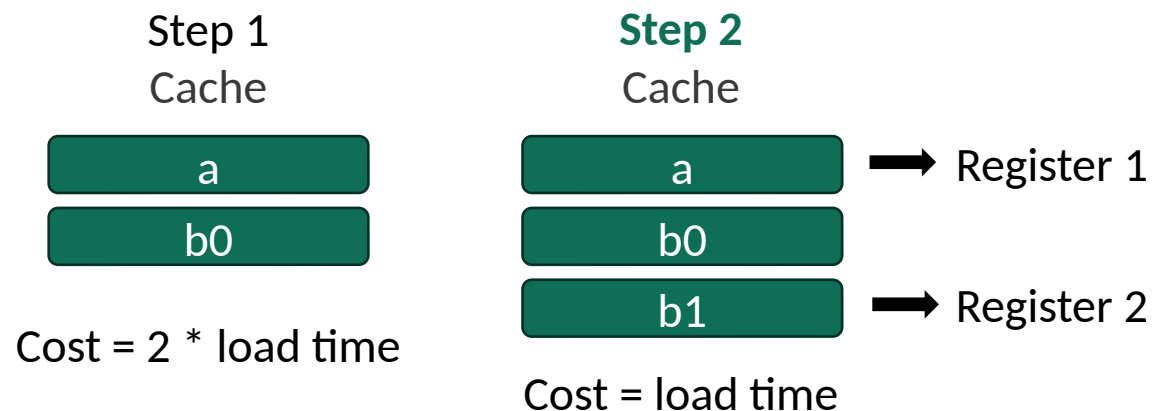
Step 1: load a, b[0] — a miss, b[0] miss

Step 2: load a, b[1] — a hit, b[1] miss
a is still cached: only b[1] is loaded

```
for (int row = 0; row < M; ++row)
    y[row] = dot(A + row*K, x, K);
```

a = x: read by every row (reused)

b[i] = row i of A: read once



Input reuse: a cache analogy (3)

Cache model: holds three vectors

Step 1: load a, b[0] — a miss, b[0] miss

Step 2: load a, b[1] — a hit, b[1] miss

Step 3: load a, b[2] — a hit, b[2] miss

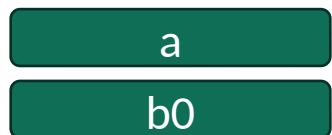
Cache full: b[2] replaces the oldest entry, a

```
for (int row = 0; row < M; ++row)
    y[row] = dot(A + row*K, x, K);
```

a = x: read by every row (reused)

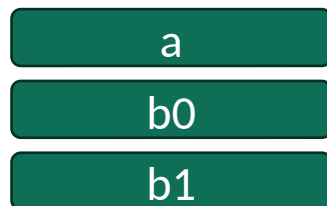
b[i] = row i of A: read once

Step 1
Cache



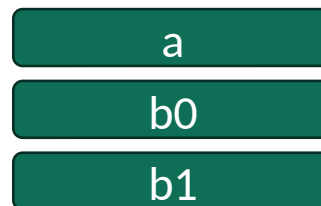
Cost = 2 * load time

Step 2
Cache

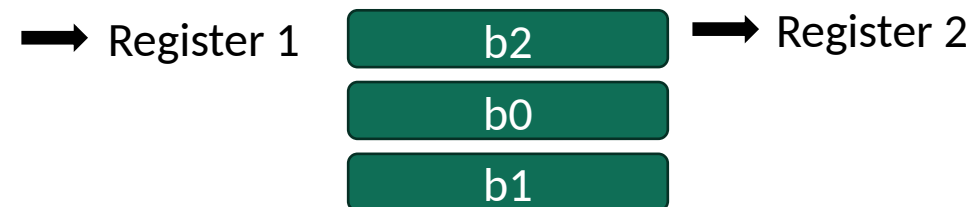


Cost = load time

Step 3
Cache



Cost = load time



Cache after the replacement

Input reuse: a cache analogy (4)

Cache model: holds three vectors

Step 1: load a, b[0] — a miss, b[0] miss

Step 2: load a, b[1] — a hit, b[1] miss

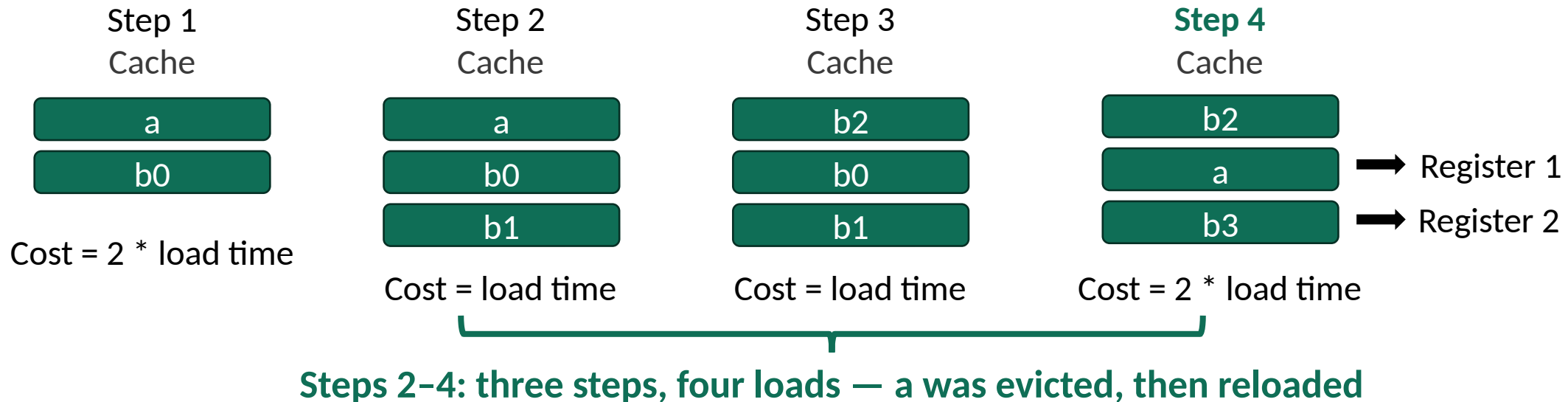
Step 3: load a, b[2] — a hit, b[2] miss; a evicted

Step 4: load a, b[3] — a miss, b[3] miss

```
for (int row = 0; row < M; ++row)
    y[row] = dot(A + row*K, x, K);
```

a = x: read by every row (reused)

b[i] = row i of A: read once



Input reuse: explicit shared storage

Shared memory: explicit storage

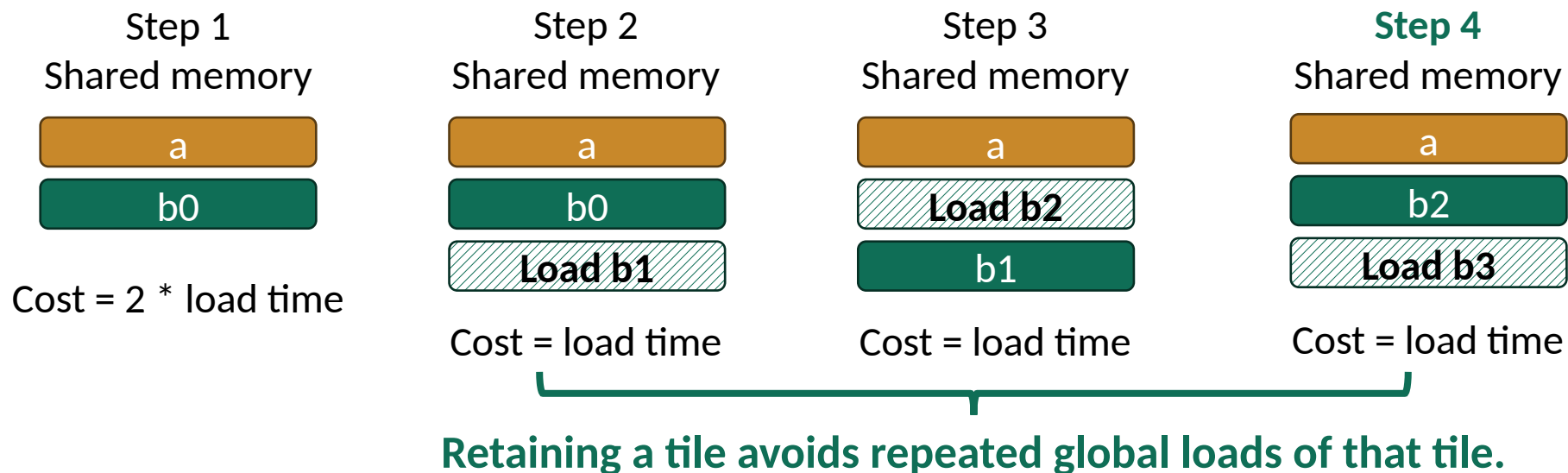
a stays in shared memory: no automatic eviction

Each **b** tile is loaded explicitly, over an old one

Synchronize before use and before overwriting

Step 4: 1 load, not 2

a was never evicted, so only **b3** is loaded



CUDA shared-memory declaration

```
1  __shared__ float tile[TILE][TILE];  
   // Cooperative load, edge padding = 0:  
2  tile[ty][tx] = valid ? input[index] : 0.0f;  
3  __syncthreads();  
   // ... every thread reads the tile ...  
4  __syncthreads();  
   // The tile may now be overwritten.
```

All threads of the block must reach every
__syncthreads()

No early return before a barrier

- 1 **__shared__**: one tile per block, seen by all its threads
- 2 Each thread loads one element; out-of-range ones store 0
- 3 **Barrier 1**: the whole tile is loaded before anyone reads it
- 4 **Barrier 2**: every read is done before the tile is overwritten

GEMM: optimized libraries and CUDA kernels

cuBLAS: tuned library

Tuned matrix multiplication implementations

The baseline to compare any kernel with

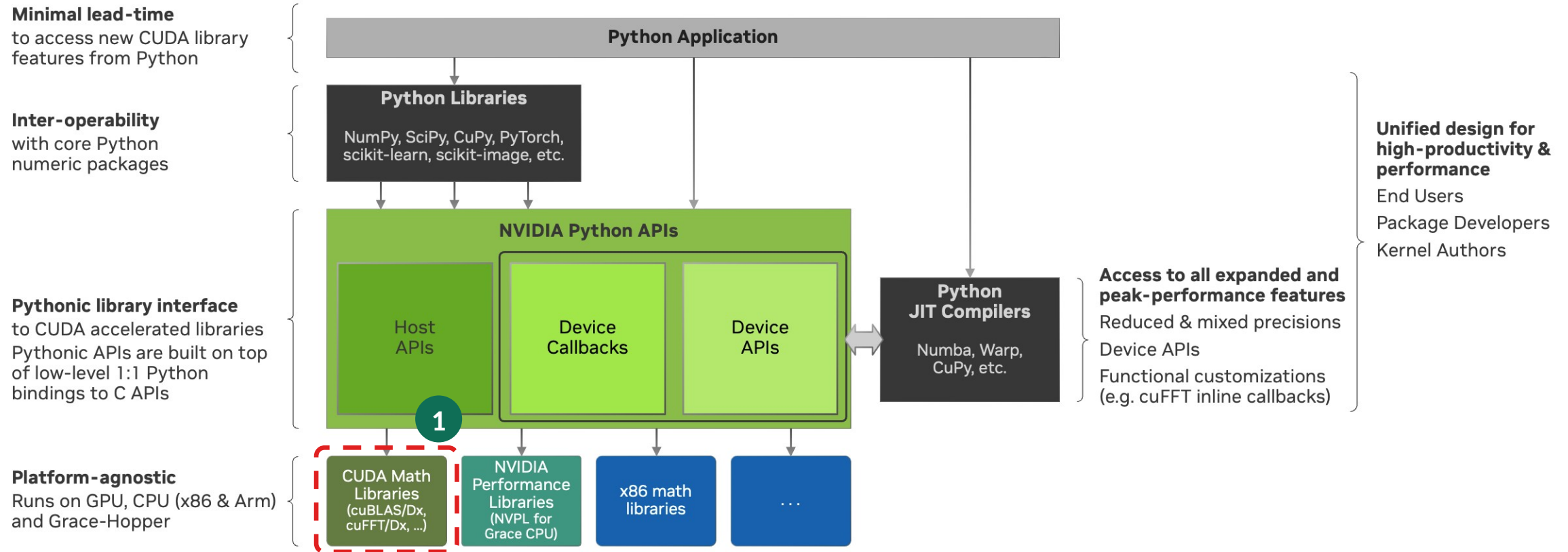
Handwritten CUDA kernel

Explains the indexing and data reuse

Does not generally outperform cuBLAS

Measure: compare any custom implementation with a tuned library baseline

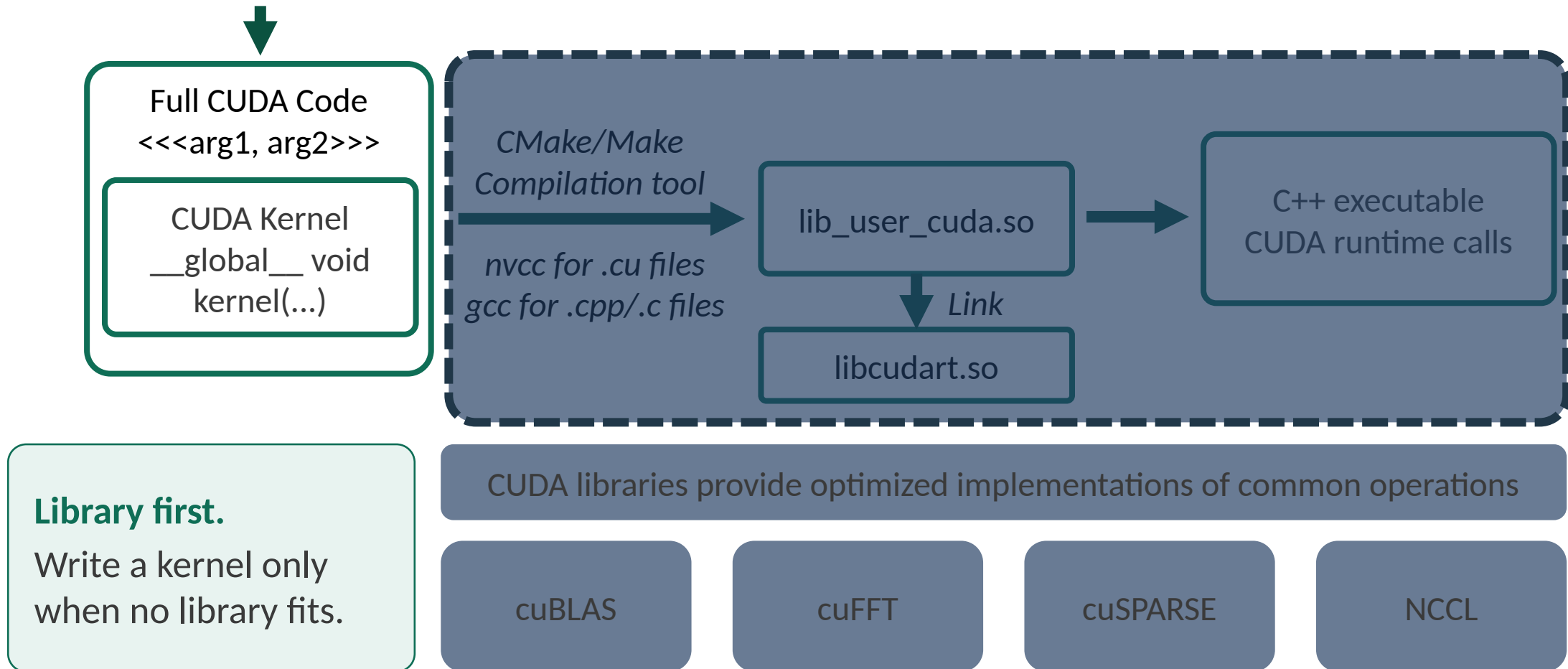
GEMM: the CUDA library stack



1 GPU matmul in CuPy or PyTorch ends in a CUDA math library such as cuBLAS.

CUDA libraries and application code

What we have learned just now.



GEMM optimization: tiling

Tiling: load a tile once, reuse it for many outputs

Block loop. Load input tiles → synchronize → compute → advance along K

Libraries add more. Specialized instructions, data layouts and scheduling

No single best tile. Tile size and instructions depend on the GPU and matrix shape

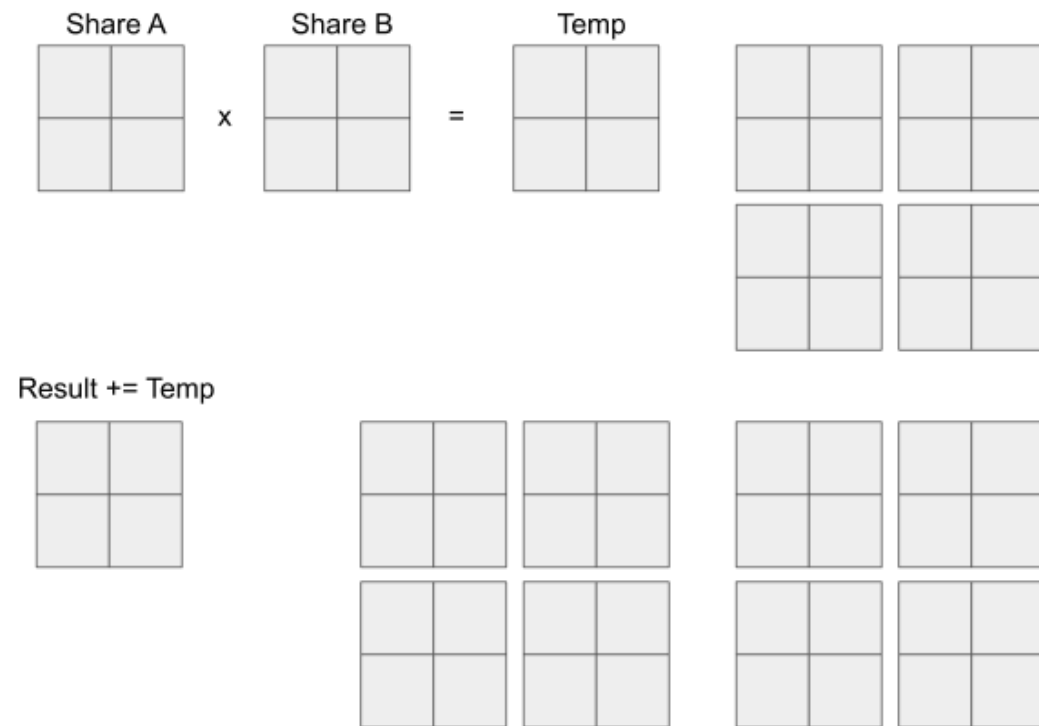
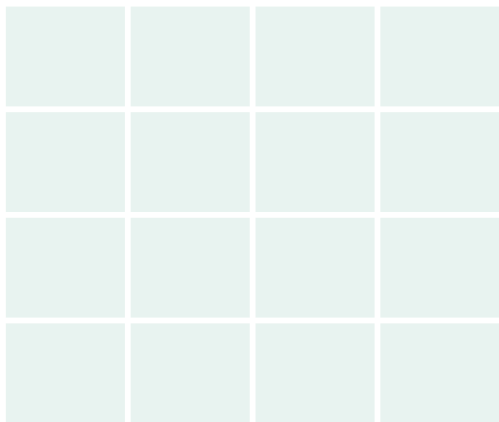


Figure: [penny-xu.github.io, tiled matrix multiplication](https://penny-xu.github.io/tiled-matrix-multiplication)

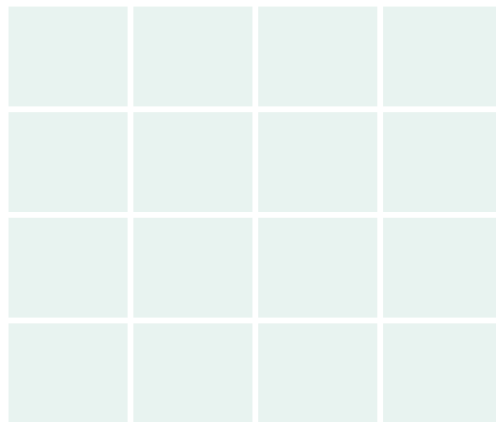
Tiled GEMM and input reuse

A tile



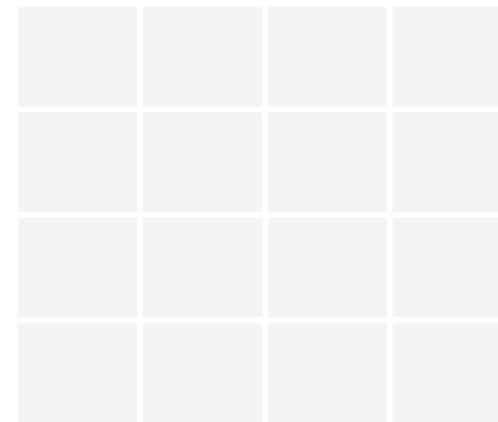
16 × 16 (schematic)

B tile



16 × 16 (schematic)

C output tile



Register accumulators

Load cooperatively, synchronize, reuse, repeat over K.

512 input values serve 4,096 multiply-adds per tile pair.

The two barriers in tiled GEMM

```
// Every thread loads a valid value or zero.  
as[ty][tx] = valid_a ? a[a_index] : 0;  
bs[ty][tx] = valid_b ? b[b_index] : 0;  
1  __syncthreads();  
   for (int q = 0; q < TILE; ++q)  
       acc += as[ty][q] * bs[q][tx];  
2  __syncthreads();
```

1 Data is ready before any thread consumes it

2 Consumers finish before the shared buffer is overwritten

Edge tiles load zeros: never return early — every thread must reach both barriers.

Timing GPU work with events

```
1 cudaEventRecord(start, stream);  
  // Launch the measured work in this stream.  
1 cudaEventRecord(stop, stream);  
2 cudaEventSynchronize(stop);  
3 cudaEventElapsedTime(&ms, start, stop);
```

1 start and stop are queued in the stream, around the work

2 CPU waits for stop; a CPU timer around an async launch sees mostly submission

3 GPU time between the two events, in ms

Measure honestly

Warm up

Repeat

Check results

State what the measured interval includes: kernel only, or copies and host work too.

Batch results and allocation cleanup

Check the result, in [cuda-tutorial/2-vectoradd](#)

```
sbatch smoke.sbatch
queue -u "$USER"
1 sacct -j JOB_ID --format=JobID,State,ExitCode
2 cat smoke-JOB_ID.out
```

1 State COMPLETED,
ExitCode 0:0

2 PASS in the output file

Clean up

Exit the interactive shell: that releases its allocation

`scancel JOB_ID` cancels a job you no longer need

“Submitted” is not success: check the final state, the exit code and the output.

After this lecture

You can now

Write a CUDA C++ kernel

Choose a launch configuration

Move data between host and device

Tell resident threads from executing instructions

Next step



Correct first, then fast, and always against a baseline.

CUDA exercises

Task 1 CUDA vector add with $n = 1003$ and $n = 257$

Task 2 Verify a non-square tiled GEMM against a CPU reference

Extension Precise KNN top 10 or K-means with a validated CUDA baseline

Why these sizes?

1003 and **257** are not multiples of the block size: the last block has spare threads, so the $i < n$ guard must work

A non-square GEMM exposes swapped row and column indices

Check against a CPU reference before you measure speed.

Further reading on CUDA

Code and tutorials

Must read: [An Even Easier Introduction to CUDA](#) — NVIDIA Technical Blog

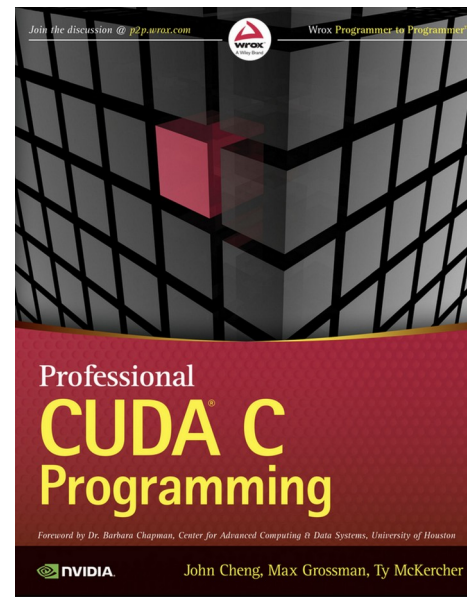
[NVIDIA/cuda-samples](#) — official sample programs for every CUDA feature

[HMUNACHI/cuda-repo](#) — from zero to hero CUDA for maths and machine learning on GPU

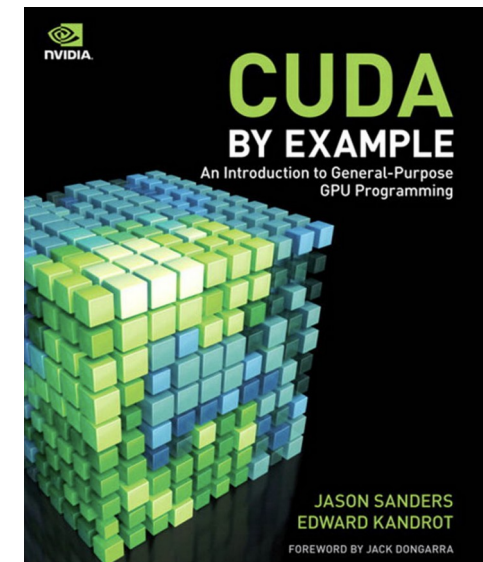
[ed-aisys/edin-mls-26-autumn](#) — every CUDA example in this lecture, in cuda-tutorial/

Start with the blog, then run and modify a sample.

Books



Professional CUDA C Programming — Cheng, Grossman, McKercher



CUDA by Example — Sanders, Kandrot

CUDA runtime calls and error checks

Explicit CUDA runtime calls

```

1 cudaMalloc(&d_a, bytes);
  cudaMalloc(&d_b, bytes);
  cudaMalloc(&d_c, bytes);
  cudaMemcpy(d_a, h_a, bytes,
             cudaMemcpyHostToDevice);
2 add<<<grid,block>>>(d_a,d_b,d_c,n);

```

1 Every runtime call returns a `cudaError_t`: check it

2 A kernel launch returns no error code: query it afterwards

Checked host path

```

3 auto err = cudaGetLastError();
  if (err != cudaSuccess)
    fprintf(stderr, "%s\n",
            cudaGetErrorString(err));
4 err = cudaDeviceSynchronize();
  // check err, then read results

```

3 Catches launch errors, e.g. an invalid configuration

4 Waits for the kernel, then reports errors raised while it ran

Check every CUDA API call and launch; synchronize before reading results on the host.

CUDA operations and launch configurations

Operation patterns

Elementwise. One output per thread

Reduction. Threads must communicate and synchronize

GEMM. Tiling lets threads reuse loaded data

Rule. The launch configuration follows the kernel and its resource requirements

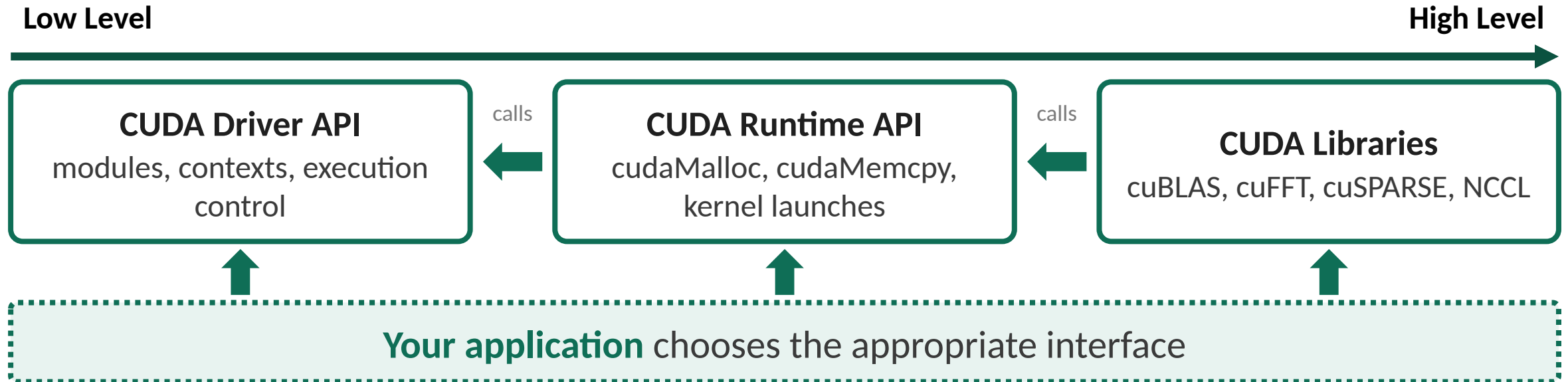
Elementwise launch

```
1 int block = 256;  
2 int grid = (n + block - 1) / block;  
   add<<<grid,block>>>(d_a,d_b,d_c,n);  
3 // The kernel still checks i < n.
```

1 Round up: enough blocks to cover all n elements

2 The last block has spare threads; the guard keeps them out of bounds

CUDA interfaces and tools



Tools: profilers

Nsight Systems. Timeline of the whole program: CPU, kernels, copies

Nsight Compute. Detailed metrics for one kernel

Alternative GPU environments

For this class: use the university teaching GPU cluster

Alternatives: cloud services

Options. Colab, Lightning and Runpod can provide alternative GPU environments

Terms change. GPU availability, pricing and free quotas vary by service and date: check the current terms first

Fair use. Do not rely on multiple accounts to expand a quota

Further CUDA optimization topics

Bank conflict. Threads of one warp hit different addresses in the same shared-memory bank, so the accesses serialize

[Stack Overflow: What is a bank conflict? \(CUDA / OpenCL programming\)](#)

Register spill. A kernel needs more registers than it gets, so values spill to slow local memory

[NVIDIA training: Local Memory and Register Spilling \(PDF\)](#)

Memory coalescing. Neighbouring threads read neighbouring addresses, so a warp's loads merge into few transactions

[Stack Overflow: In CUDA, what is memory coalescing, and how is it achieved?](#)

Nsight Compute reports all three: measure before you tune.

CUDA environment configuration

Toolkit, driver and compute capability are three different version numbers

- 1 **CUDA Toolkit.** Use the version installed on the compute node
- 2 **Driver.** Must be new enough for the toolkit version
- 3 **GPU architecture.** Read the device properties; choose a supported compiler target
- 4 **Build tools.** nvcc or CMake; keep the classroom executable and source together

Check on the compute node

- 1 `nvcc --version`
- 2 `nvidia-smi`
- 3 `nvidia-smi \`
`--query-gpu=compute_cap \`
`--format=csv`
- 4 `nvcc -arch=native app.cu`

Further reading: Tensor Cores

WMMA API. Warp-level $16 \times 16 \times 16$ matrix multiply-accumulate in CUDA C++; cuBLAS opt-in
[NVIDIA Technical Blog: Programming Tensor Cores in CUDA 9](#)

mma.sync step by step. Six kernels from tiling to 96% of cuBLAS HGEMM (Turing T4)
[A. Armbruster: How To Write A Fast Matrix Multiplication From Scratch With Tensor Cores](#)

Hopper wgmma + TMA. Warp-group Tensor Core instructions on H100, from 32 to over 700 TFLOPs
[P. Shankhdhar: Outperforming cuBLAS on H100: a Worklog](#)

How libraries do it. CUTLASS / CuTe templates and the CuTe DSL for Tensor Core GEMMs
[GitHub: NVIDIA/cutlass](#)

Start with WMMA, then follow a worklog; read CUTLASS to see production kernels.

Optional: atomics (1), a race on a shared sum

GPU: every thread adds to one global

```

__device__ float sum = 0; // device mem

kernel<<<grid, block>>>(n,
    [=] __device__ (int i) {
1      sum += sinf(arr[i]); // race!
    });

float result = 0;
2  cudaMemcpyFromSymbol(&result, sum,
    sizeof(float));
printf("GPU %f\n", result);

```

CPU reference

```

result = 0;
for (int i = 0; i < n; ++i)
    result += sinf(i);
printf("CPU %f\n", result);

```

Run 1 $\neq$ Run 2 $\neq$ CPU: the GPU value changes on every run

1

sum += x is load, add, store: threads interleave and overwrite

2

__device__ variables: copy with cudaMemcpyFromSymbol

Which thread wins is undefined: a data race, not a rounding problem.

Optional: atomics (2), atomicAdd and atomicCAS

Fix: a hardware read-modify-write

```
kernel<<<grid, block>>>(n,
    [=] __device__ (int i) {
        atomicAdd(&sum, sinf(arr[i]));
    });
```

Result. Now differs from the CPU only by summation order

Family. atomicAdd, Sub, Min, Max, And, Or, Xor, Exch, Inc, Dec, CAS all return the old value

Build your own with atomicCAS

```
__device__ float
my_atom_add(float *dst, float src) {
    int old = __float_as_int(*dst), expect;
    do {
        expect = old;
        old = atomicCAS((int *)dst, expect,
            __float_as_int(
                __int_as_float(expect) + src));
    } while (expect != old); // retry
    return __int_as_float(old);
}
```

atomicCAS(addr, expect, new) writes only if
*addr == expect; integer types only, hence
__float_as_int

Fork and merge: sum per block first (shared memory), then one atomicAdd per block.