Durham
University

GPU Computing

SCC Training

Laura Morgenstern
laura.morgenstern@durham.ac.uk

History of GPU Computing

1970s	Specialized graphics circuits in arcade video game hardware
1981	First dedicated GPU chip NEC7220
1990s	Dissemination of OpenGL for GPU programming
2000s	(Mis)use of graphics pipeline for matrix computations
2008	Nvidia introduces G80, the first general purpose GPU along with CUDA
Today	GPU computing common in AI and HPC applications

Durham University

Concurrency vs. Parallelism

Types of Concurrency

→ Time multiplexing

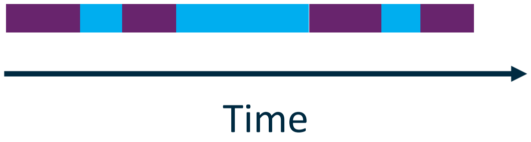
→ Space multiplexing

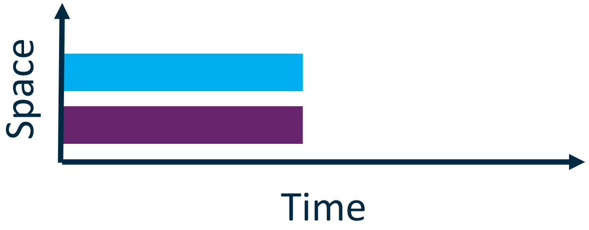
Interleaving

Two *concurrent* processes *A* and *B* are executed in an *interleaved* manner iff *A* and *B* are executed *alternatingly* on the *same execution unit*.

Parallelism

Two *concurrent* processes *A* and *B* are executed in *parallel* iff *A* and *B* are executed *simultaneously* on *different execution units*.



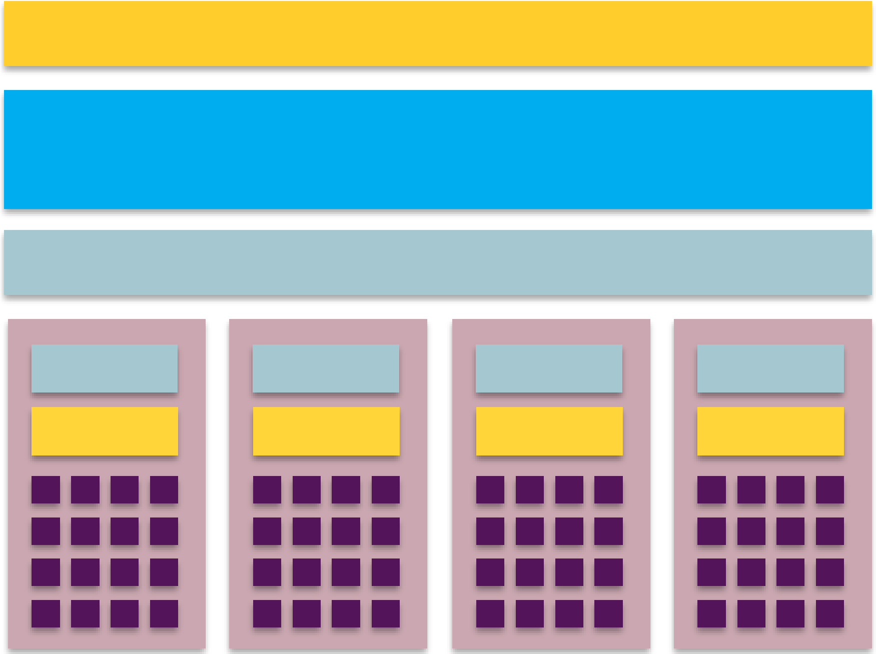


Durham University

GPU Architecture

GPU Architecture

- Different GPU vendors, AMD, Intel, Nvidia use different terminology but architectures are similar
- Hierarchy of schedulers
- Memory and cache hierarchy
- Compute units consisting of processing elements



GPU Architecture: Nvidia GA100 (Ampere) Chip

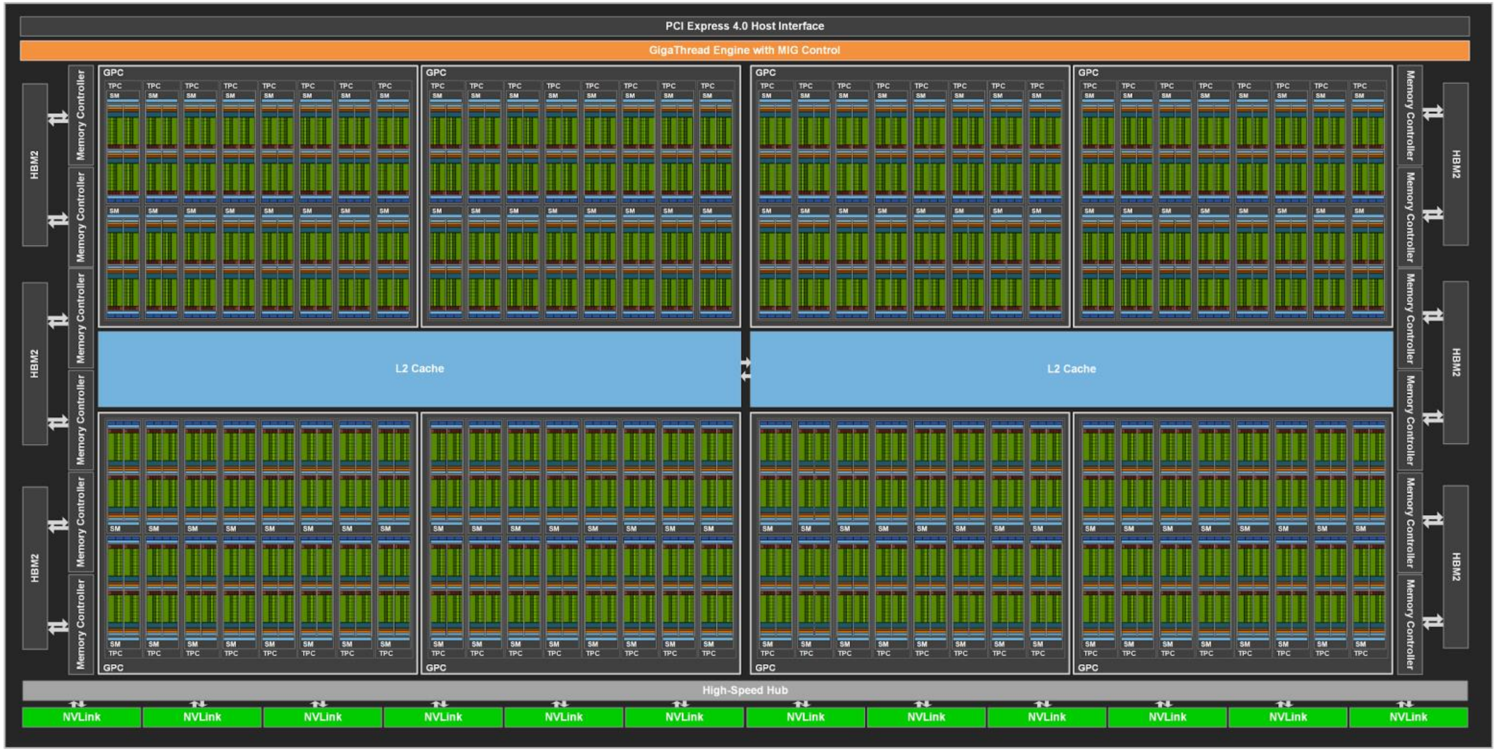

PCI Express 4.0 Host Interface
GigaThread Engine with MIG Control
GPC
TPC
SM
L2 Cache
Memory Controller
HBM2
High-Speed Hub
NVLink

Image credit: A100 Whitepaper, Nvidia

GPU Architecture: Nvidia GA100 SM

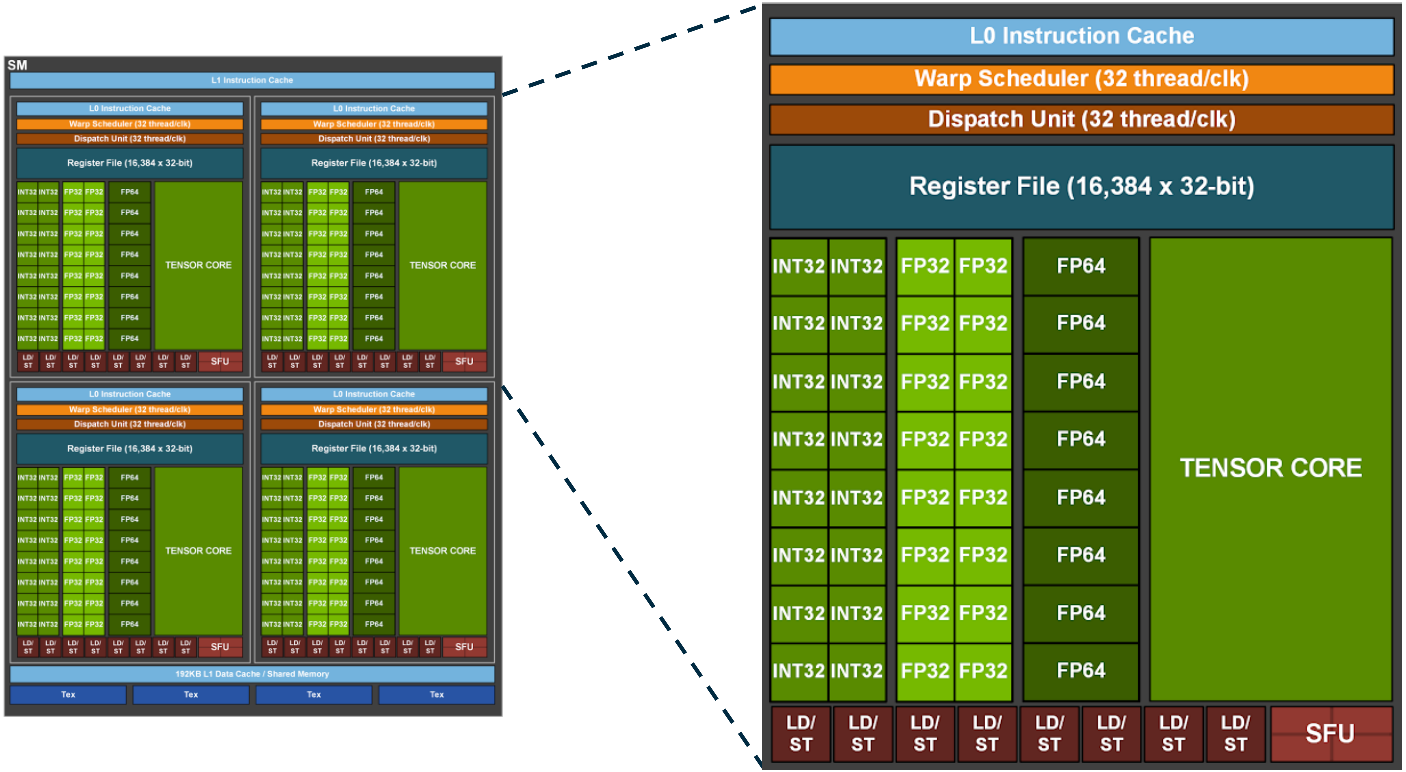
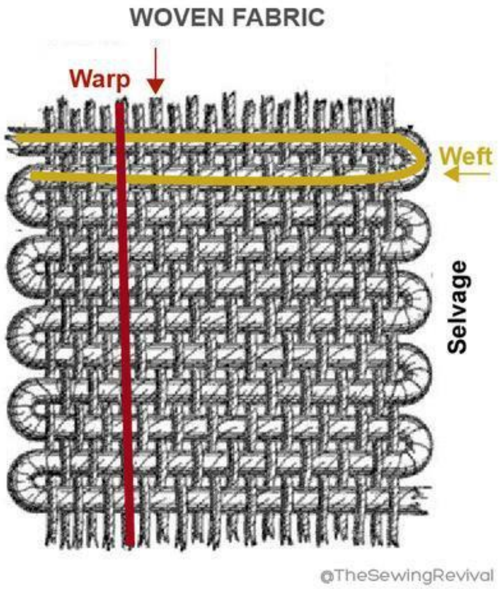

SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE
LD/ ST
SFU
192KB L1 Data Cache / Shared Memory
Tex

Image credit: A100 Whitepaper, Nvidia

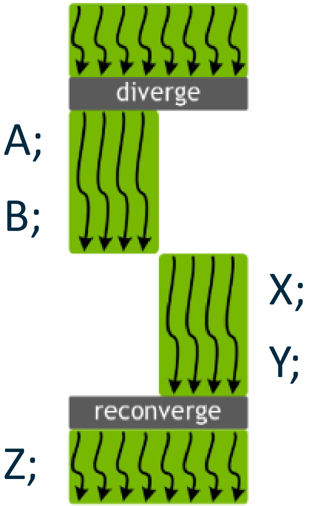
SIMT Execution Model

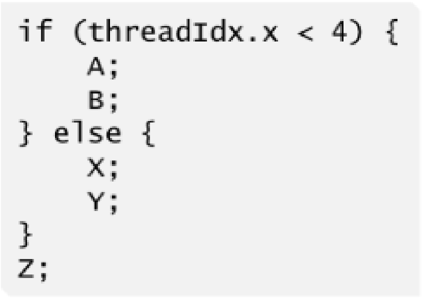
- **S**ingle **I**nstruction, **M**ultiple **T**hread (Nvidia speak)
- Warp-based execution:
  - Warp: group of threads that execute the same instruction on different data elements concurrently
  - Lanes in a warp diverge at conditional statements by masking lanes dependent on the execution path they take

  → SIMD model, but hidden from programmer by the programming model

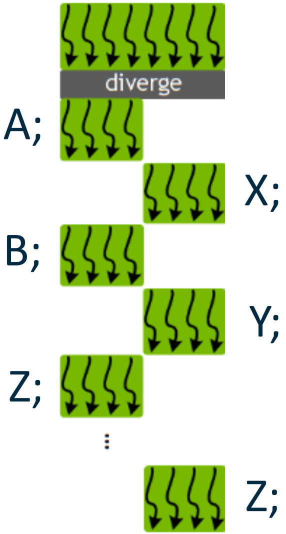


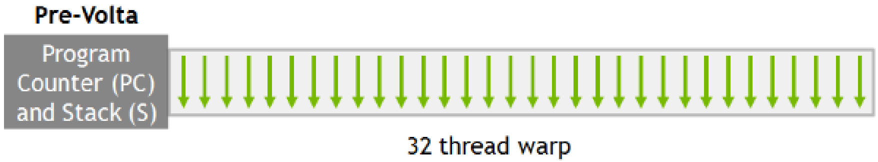
SIMT Execution Model





```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```





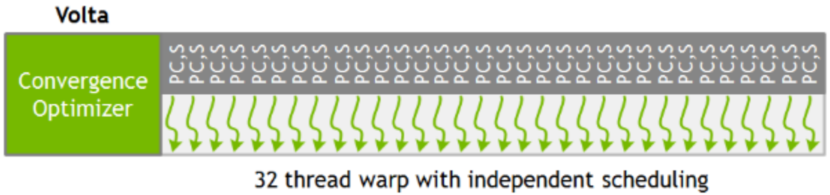
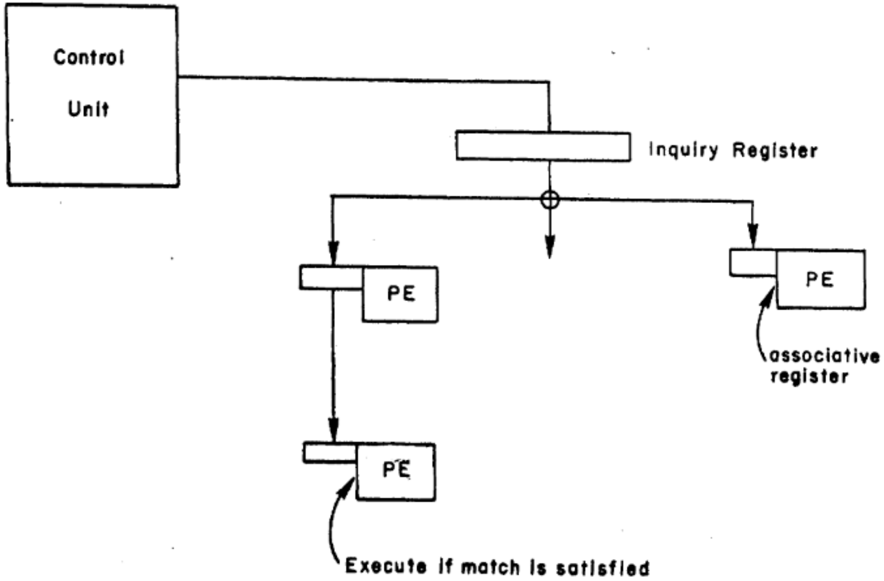


Image credit: V100 Whitepaper, Nvidia

SIMT Execution Model

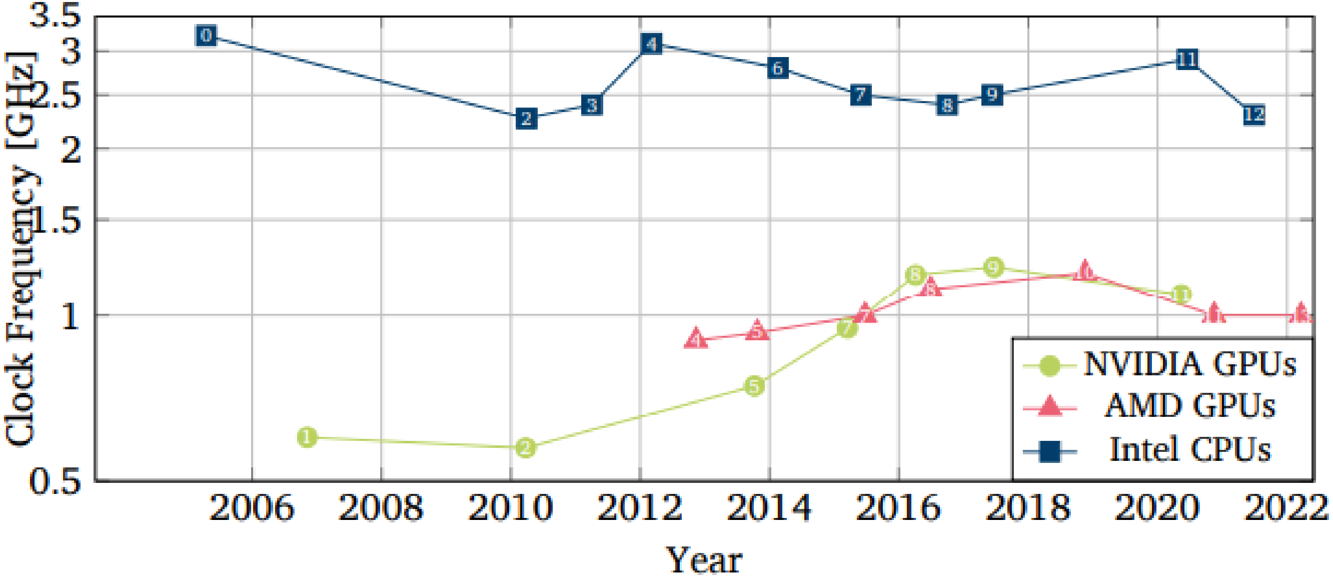


→ GPU is basically the successor of associative array processors

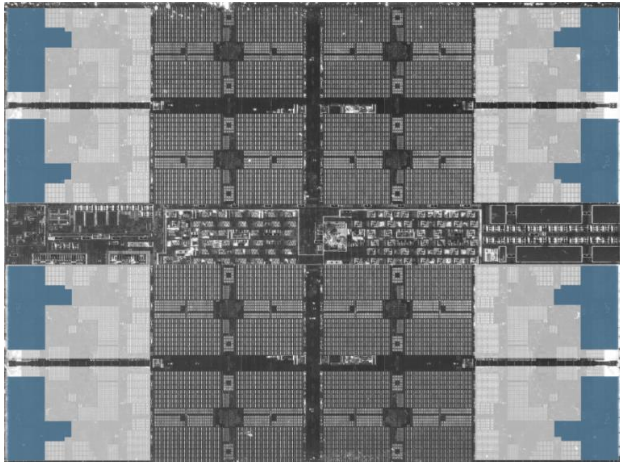
Image credit: https://users.cs.utah.edu/~hari/teaching/paralg/Flynn72.pdf

CPU vs. GPU Architectures

CPU vs. GPU Architectures: Clock Frequency

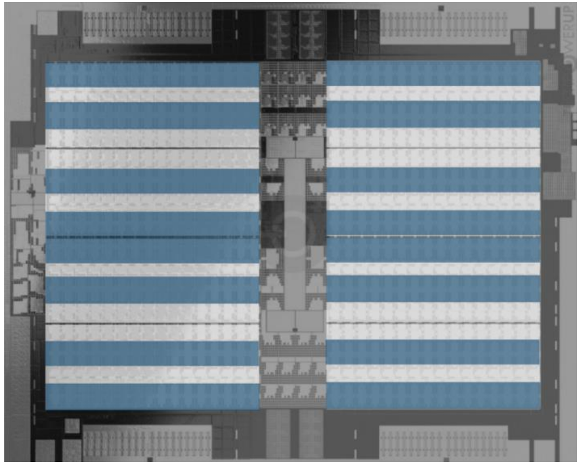


CPU vs. GPU Architectures: Compute Density



AMD EPYC 7702 CPU CCD:

- 8 Zen 2 cores (white) and 4 L3 cache slices
- Compute logic (blue) per core including ALU, FPU, SIMD units



AMD MI100:

- 8 arrays (white) à 16 compute units
- Compute logic (blue) per array including ALU, FPU, matrix cores

Credit for the underlying CPU die shot goes to *Fritzchens Fritz. AMD Epyc 7702 ES. 2019. url: https://en.wikichip.org/wiki/File:AMD_Zen_2_CCD.jpg* and for the underlying GPU die shot to *TechPowerUp. AMD Arcturus. 2020. url: https://www.techpowerup. com/gpu-specs/amd-arcturus.g927#gallery-2.*

CPU vs. GPU Architectures: Compute Density

	CPU	GPU
Chip	AMD EPYC 7702	AMD MI 100
Die size [mm2]	592	750
Cores	64	128
Base Clock [GHz]	2.0	1.0
Instructions per cycle	16 (2x AVX2 FMA)	64 (32x FMA)
FP64 Peak Performance [GFLOP/s]	2048	8192
Compute density [GFLOP/s/mm2]	**3.46**	**10.9**

- Compute density of GPUs is (at least) a factor 3 higher since:
  - CPUs are optimized for latency → Do one thing as fast as possible.
  - GPUs are optimized for throughput → Do as many things as possible at once.

Durham University

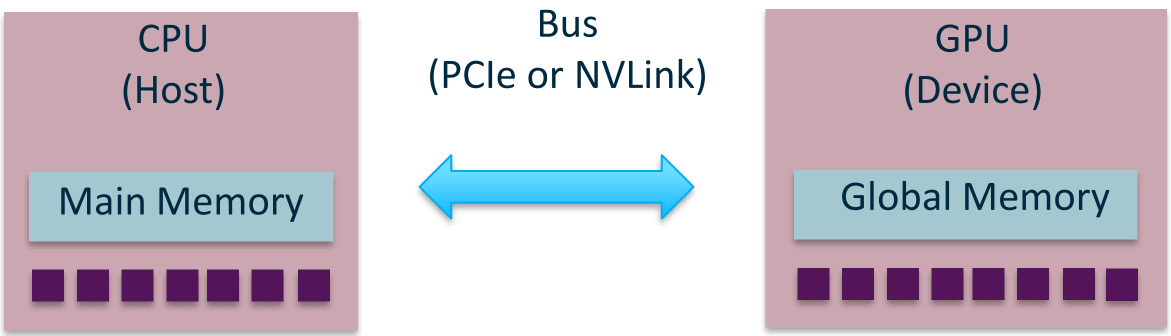
Parallel Programming Models: CUDA

COMP52315 GPU Programming

Laura Morgenstern

laura.morgenstern@durham.ac.uk

Architecture Model



Execution Model

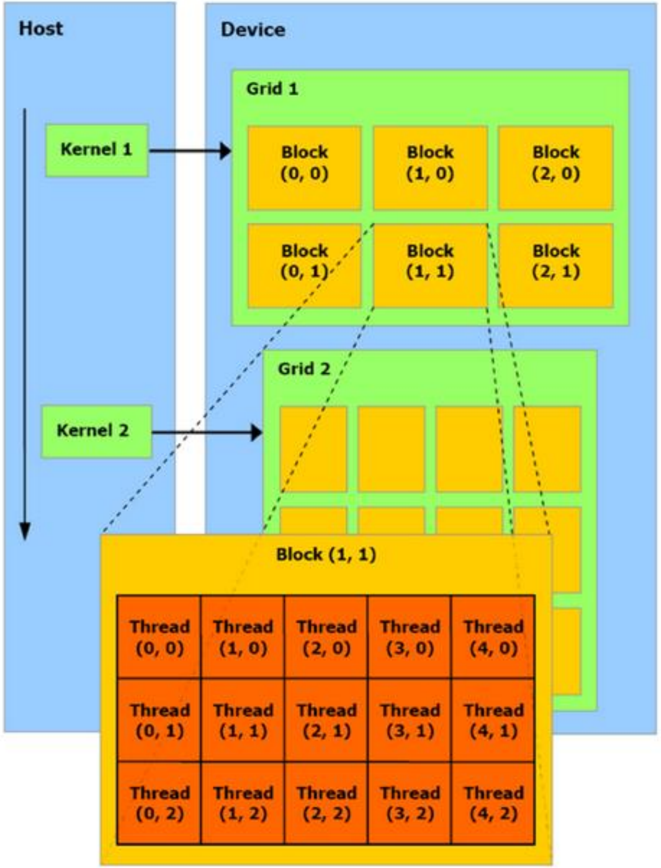
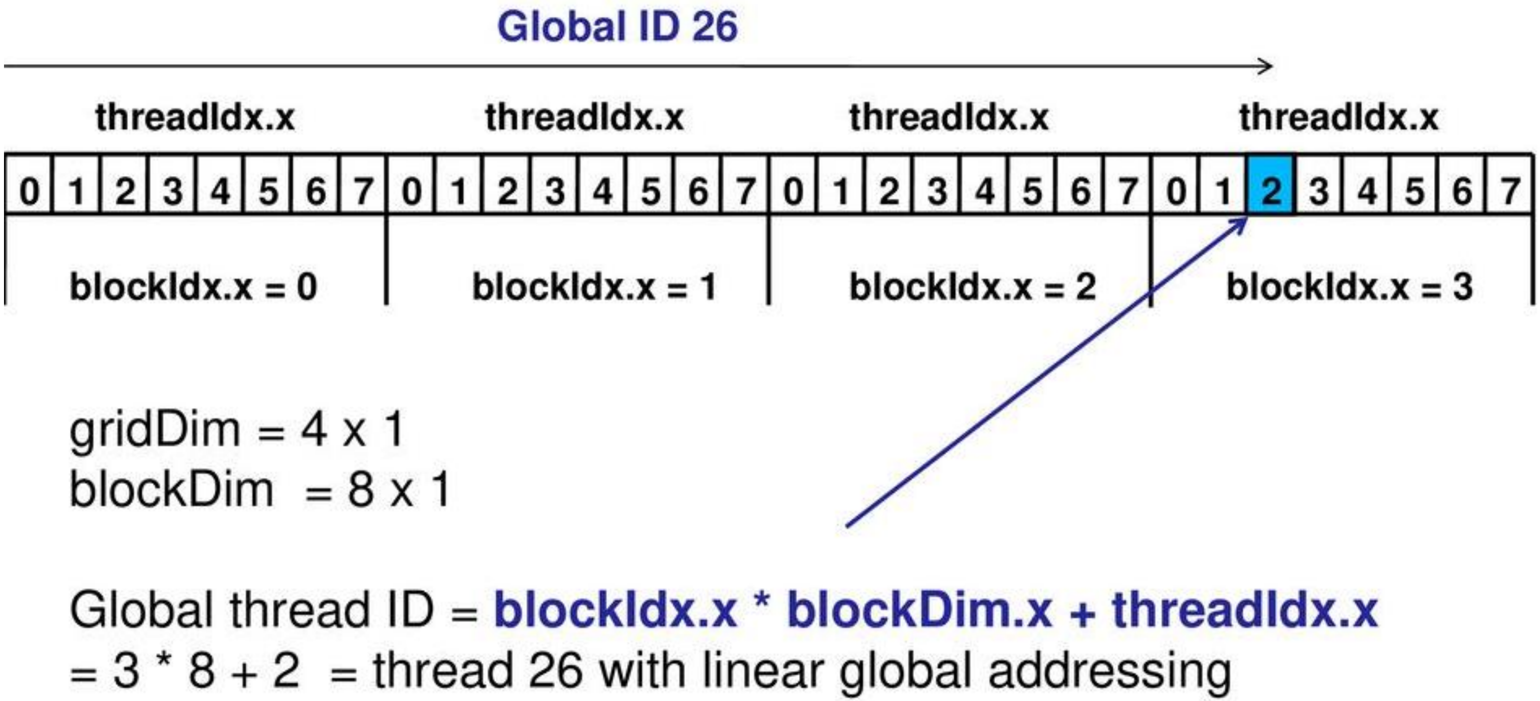


Image credit: CUDA programming guide

Execution Model



gridDim = 4 x 1
blockDim = 8 x 1

Global thread ID = **blockIdx.x * blockDim.x + threadIdx.x**
= 3 * 8 + 2 = thread 26 with linear global addressing

Image credit: Jason Sanders, Introduction to CUDA C

Memory Model

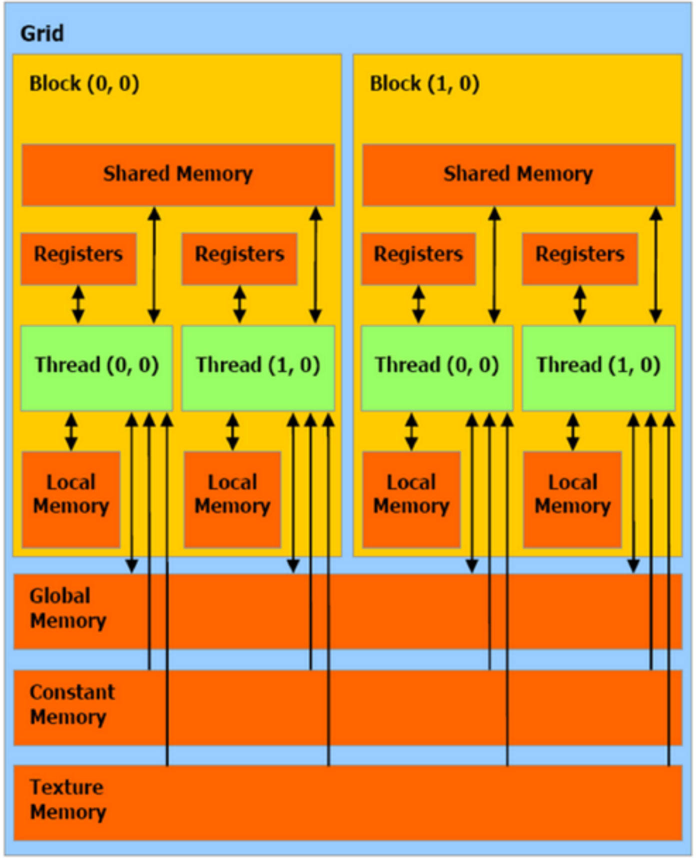


Image credit: CUDA programming guide

DEMO
Vector addition in CUDA

TASK

Write a CUDA program that takes two
matrices A and **B**, and
two **scalars x** and **y** as input, and
computes matrix **C = x*A + y*B**.

Code at: https://github.com/DUSCC/GPU-Training

Durham University

GPU Computing

SCC Training

Q & A

Laura Morgenstern
laura.morgenstern@durham.ac.uk