FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

Lecture 2

Logo

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

Current HPC Architectures

Supercomputers, GPUs

Logo

TOP 500 list

- List of current best supercomputers in the world (updated twice a year)
- https://www.top500.org/lists/top500/list/2020/11/
- Facts
  - 3 supercomputers of TOP 20 located Germany (Jülich SC, LRZ Garching, HLRS Stuttgart)
  - ExaFLOP/s performance (10^{18} floating point operations per second) is close
  - Several MW power consumption
  - Millions of cores in one supercomputer
  - TOP countries: Japan, USA, China, Europe
- TOP 1:
  - https://blog.de.fujitsu.com/produkte-services-loesungen/rundumsrechenzentrum/fugaku-der-aktuell-weltweit-leistungsstaerkste-supercomputer/
  - https://en.wikipedia.org/wiki/Fugaku_(supercomputer)

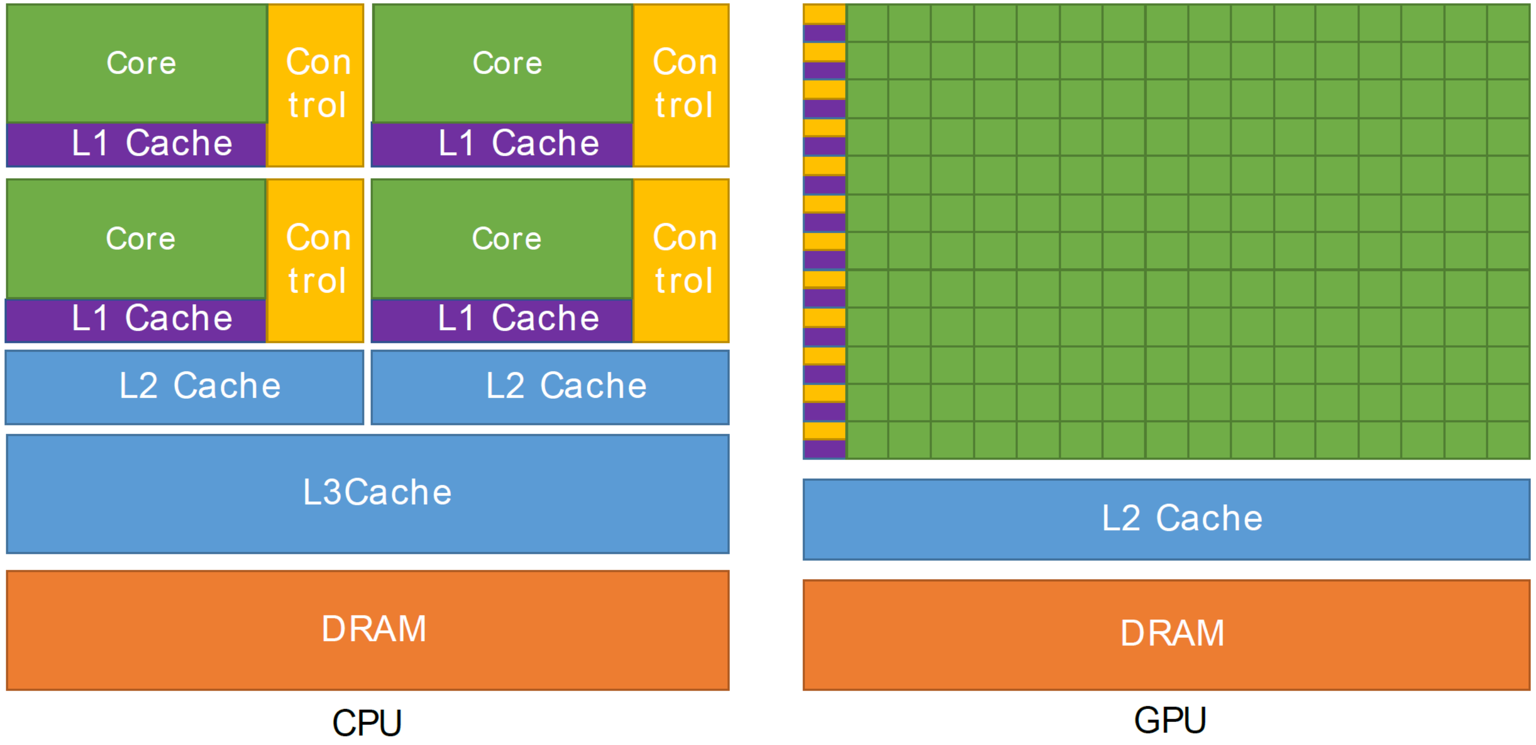
Where Does HPC Computer Architecture Go?

- It becomes more and more difficult to exploit progress in semiconductor technology for automatic performance improvements e.g.
  - Simple increase number of transistors (complexity)
  - Increase clock rate (power considerations and physical limits)
- Solution is to introduce parallelism on all hardware levels
  - On-Node level
    - Instruction level (on core)
    - SIMD-like vectorization (on core)
    - Multi-core (with shared memory)
  - Inter-node level
    - Clusters (with distributed memory and fast interconnects)

Why GPUs?

- Graphics Processing Units (GPUs) are a massively parallel computer architecture
- GPUs offer high computational performance at relatively low costs
- GPGPU
  - General-Purpose computing on a GPU
  - Using graphic hardware for non-graphic computations
  - Programming Tools like CUDA or OpenCL
- **CUDA C++ Programming Guide (nvidia.com)**
  - Later Figures on slides are from this programming guide!
  - https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units

GPU devotes more transistors to data processing



GPU vs CPU

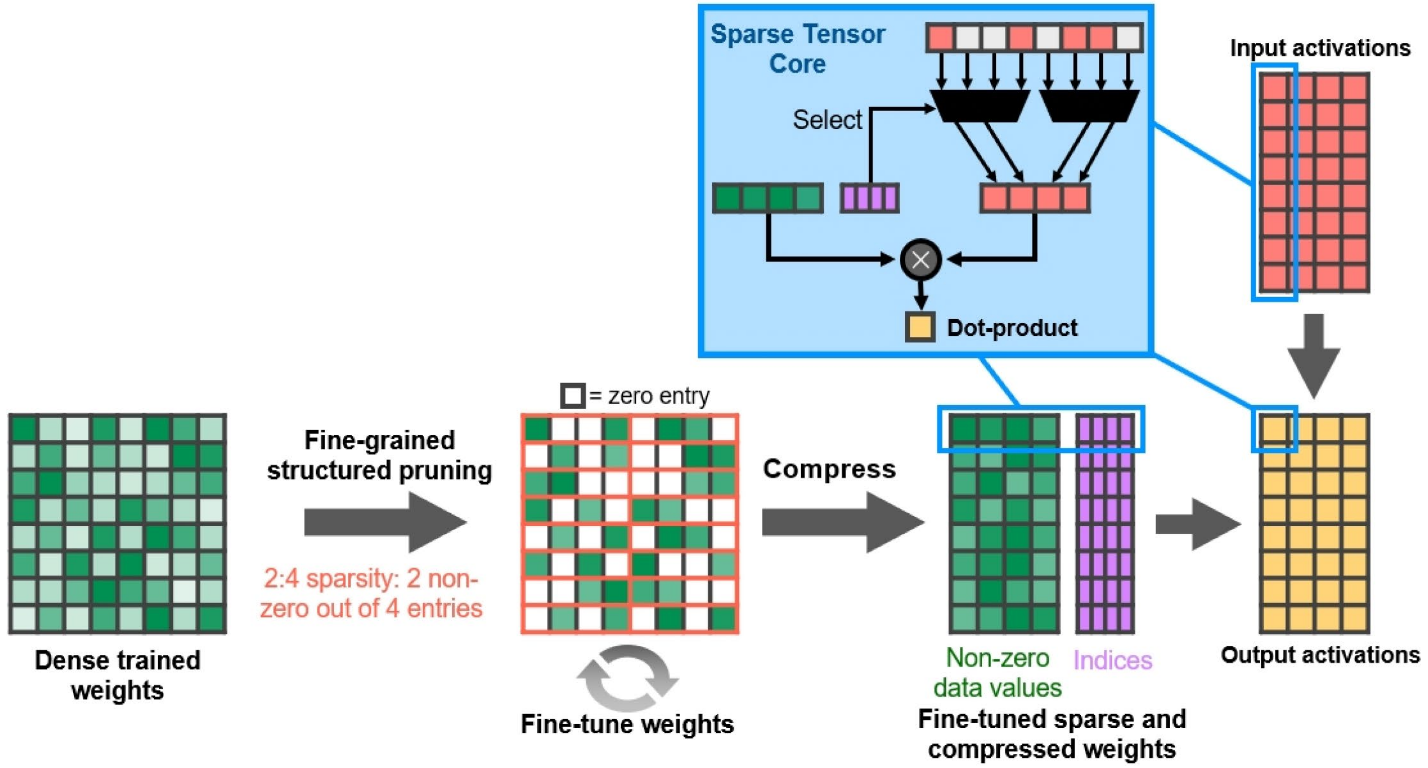
- CPUs are great for task parallelism
  - Big and fast caches
  - Branching adaptability
- GPUs are great for data parallelism
  - Multiple ALUs
  - Fast onboard memory
  - High throughput on parallel tasks
  - Executes program on each fragment

Example: Nvidia GeForce 30 series

- https://en.wikipedia.org/wiki/GeForce_30_series
- Based on Ampere architecture
  - Samsung 8 nm (semiconductor fabrication)
  - Doubled FP32 performance per Streaming Multiprocessor (SM) containing **CUDA cores** on Ampere GPUs
  - Third-generation **Tensor Cores** with FP16, bfloat16, TensorFloat-32 (TF32) and sparsity acceleration (for small matrix-vector operations)
  - Second-generation **Ray Tracing (RT) Cores**, plus concurrent ray tracing and shading and compute
  - GDDR6X memory support (RTX 3080 and RTX 3090)
  - PCI Express 4.0 (up to 64 GB/sec of peak bandwidth)
  - NVLink 3.0 (RTX 3090, 112.5 GB/sec total bandwidth between two GPUs)
  - CUDA Compute Capability 8.6 (discussed later)

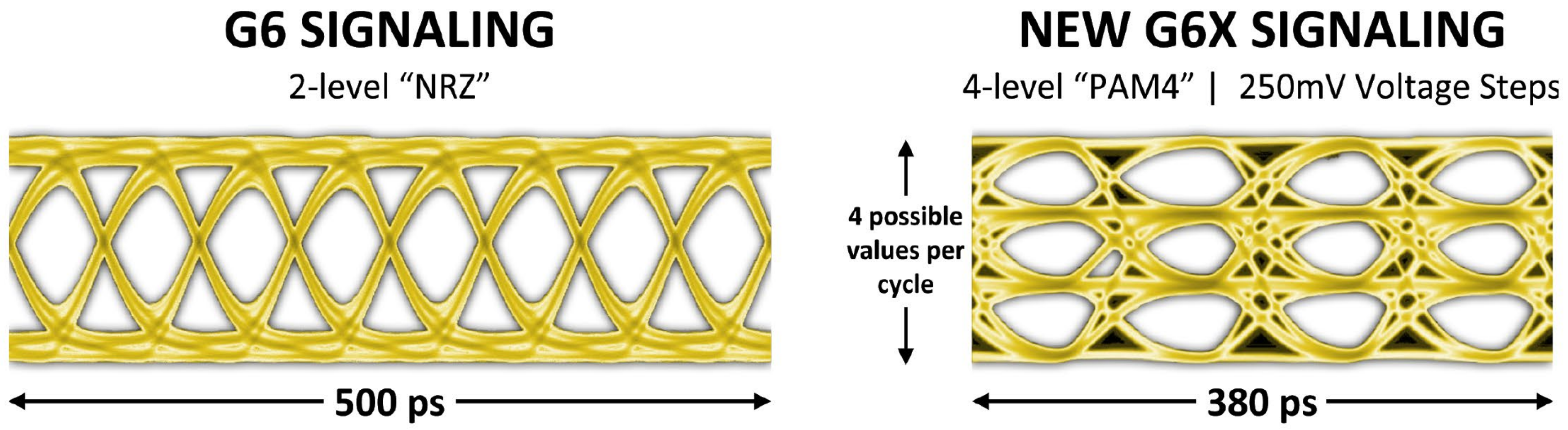
FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

Fine-grained Structured Sparsity in Tensor Cores



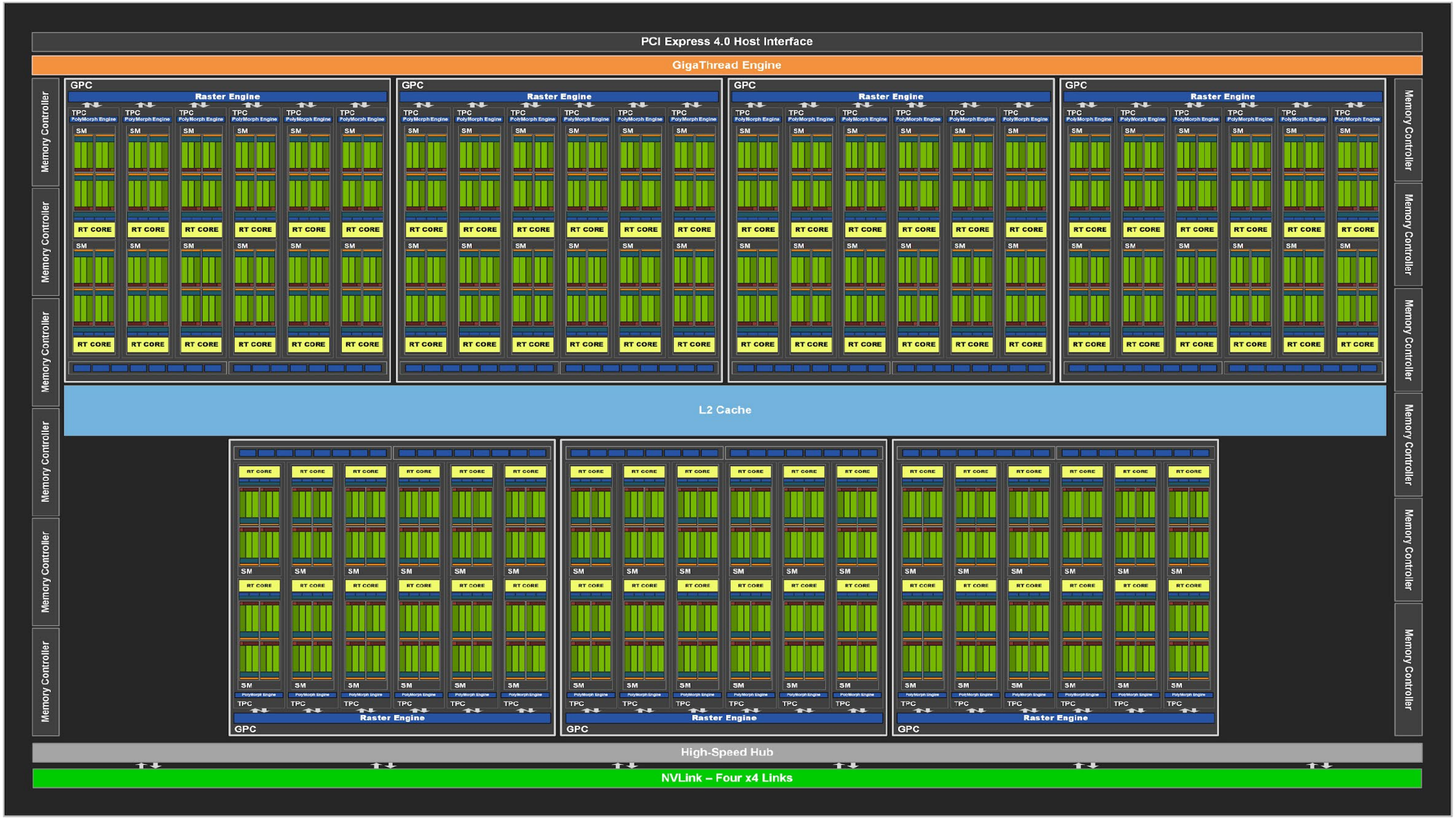
GDDR6X

- Based on four-level pulse amplitude modulation (PAM4) and doubling the I/O data rate of the previous PAM2/NRZ (non return to zero) signaling scheme.
- Instead of transmitting two binary bits of data each clock cycle (one bit on the rising edge and one bit on the falling edge of the clock), PAM4 sends two bits each clock edge, encoded using four different voltage levels. The voltage levels are divided into 250 mV steps with each level representing two bits of data - 00, 01, 10, or 11 sent on each clock edge.

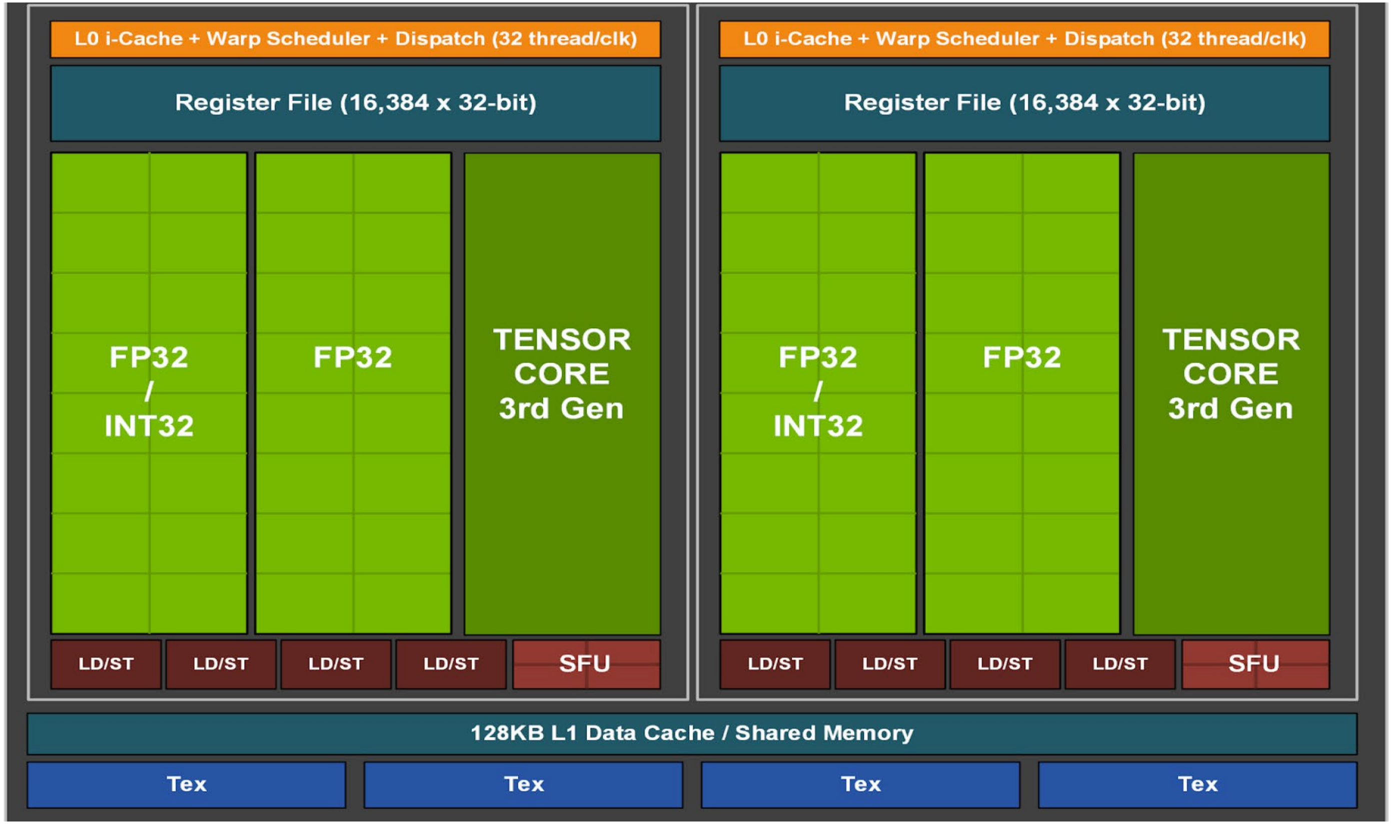

G6 SIGNALING
2-level “NRZ”
500 ps
NEW G6X SIGNALING
4-level “PAM4” | 250mV Voltage Steps
4 possible values per cycle
380 ps

Ampere GPU Architecture In-Depth

- https://www.nvidia.com/content/dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf
- GA102 includes 28.3 billion transistors with a die size of 628.4 mm^2 and is composed of
  - Graphics Processing Clusters (GPCs),
  - Texture Processing Clusters (TPCs),
  - Streaming Multiprocessors (SMs),
  - Raster Operators (ROPS), and
  - memory controllers.
- The full GA102 GPU contains seven GPCs, 42 TPCs, and 84 SMs.
- This sums up to 10752 CUDA cores, 84 RT cores, and 336 Tensor Cores (4 per SM)


PCI Express 4.0 Host Interface
GigaThread Engine
GPC
Raster Engine
TPC
PolyMorph Engine
SM
RT CORE
L2 Cache
Memory Controller
High-Speed Hub
NVLink – Four x4 Links

GA10x Streaming Multiprocessor (SM)


L0 i-Cache + Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
FP32 / INT32
FP32
TENSOR CORE 3rd Gen
LD/ST
LD/ST
LD/ST
LD/ST
SFU
L0 i-Cache + Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
FP32 / INT32
FP32
TENSOR CORE 3rd Gen
LD/ST
LD/ST
LD/ST
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex
Tex
Tex
Tex

Nvidia RTX 3090

- GPU Architecture: NVIDIA Ampere
- GPCs/TPCs/SMs: 7/41/82
- CUDA cores/Tensor Cores/RT Cores: 10496/328/82
- GPU Boost Clock: 1695 MHz
- Peak FP32 TFLOP/s: 1695 MHz * 10496 * 2 op/cycle = 35,6 TFLOP/s
- Memory Clock (data rate): 19,5 Gbps
- Memory Interface: 384-bit
- Memory bandwidth: 384/8 Byte * 19,5 Gbps = 936 GB/s
- Global memory: 24 GB GDDR6X
- Power consumption: 350 W

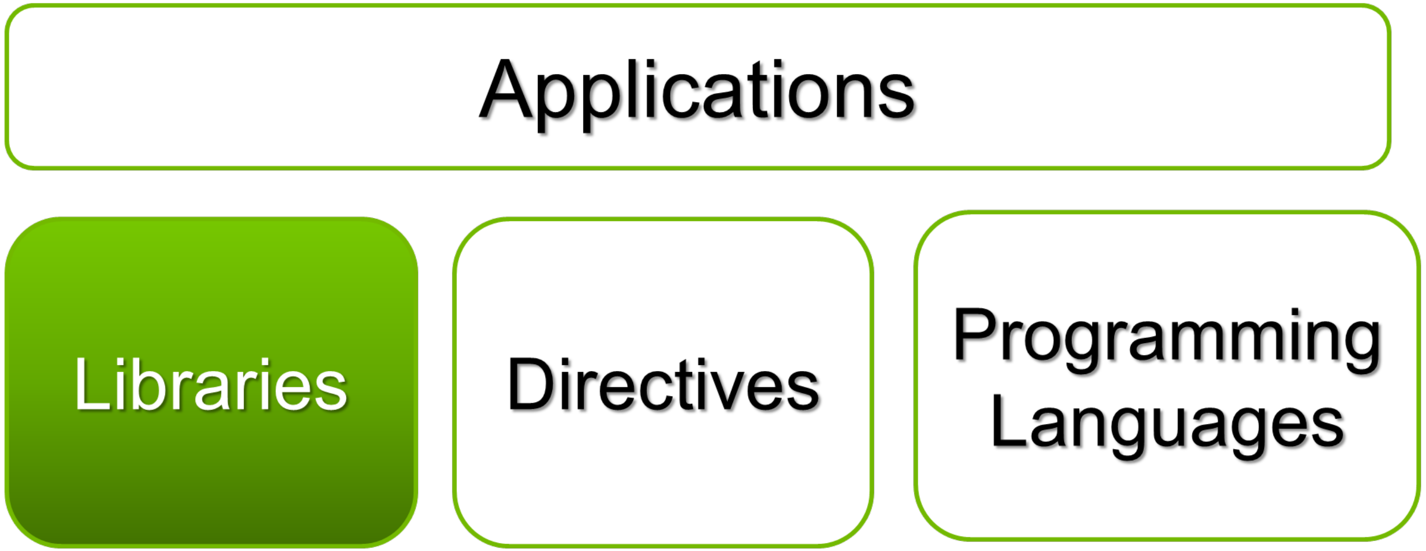
FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

Basic Concepts of GPU Programming

CUDA and OpenCL

Logo

3 Ways to Accelerate Applications on GPU



“Drop-in” Acceleration

Easily Accelerate Applications

Maximum Flexibility

Data-parallel Programming: CUDA

- What is CUDA?
  - Compute Unified Device Architecture
  - Software architecture for managing data-parallel programming
- Write kernels (functions) that execute on the device and process multiple data elements in parallel
- use massive threading

OpenCL

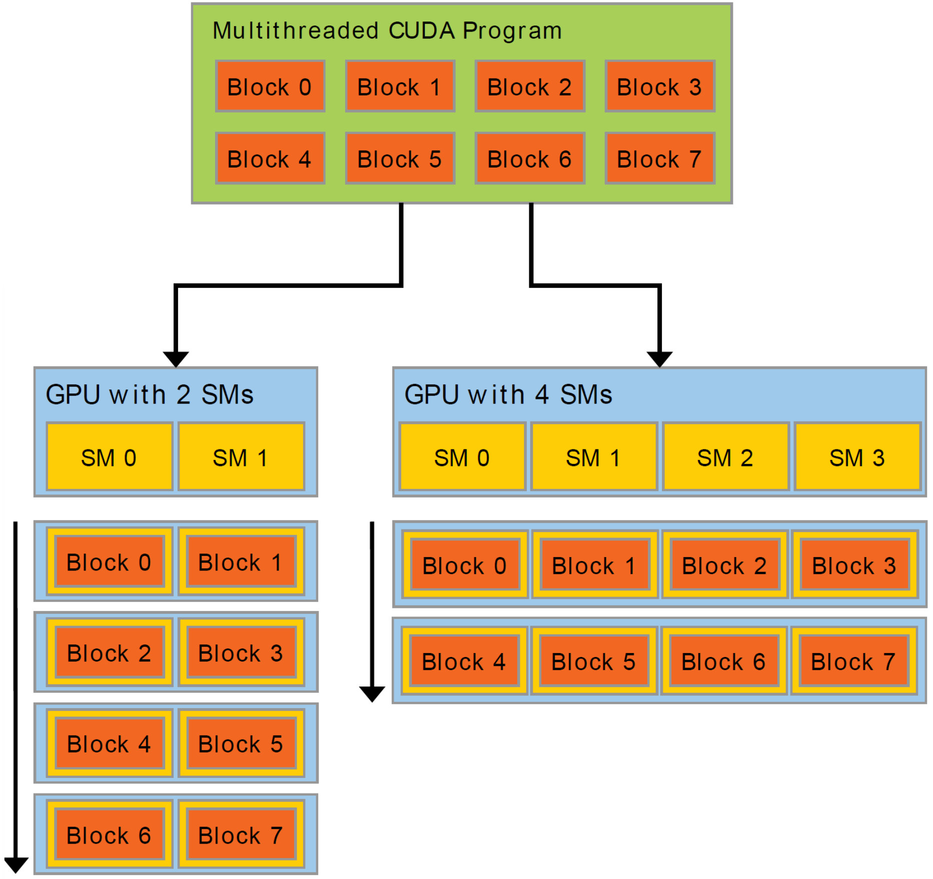
- OpenCL (Open Computing Language) is an open standard for heterogeneous parallel computing
- Managed by non-profit technology consortium Khronos group
  - www.khronos.org/opencl
- Analogous to open industry standards OpenGL for 3D graphics
- OpenCL exploits task-based and data-based parallelism
- Similar to CUDA, OpenCL includes a language (based on C99) for writing kernels executing on devices like GPU, cell, or multi-core processors

CUDA vs. OpenCL

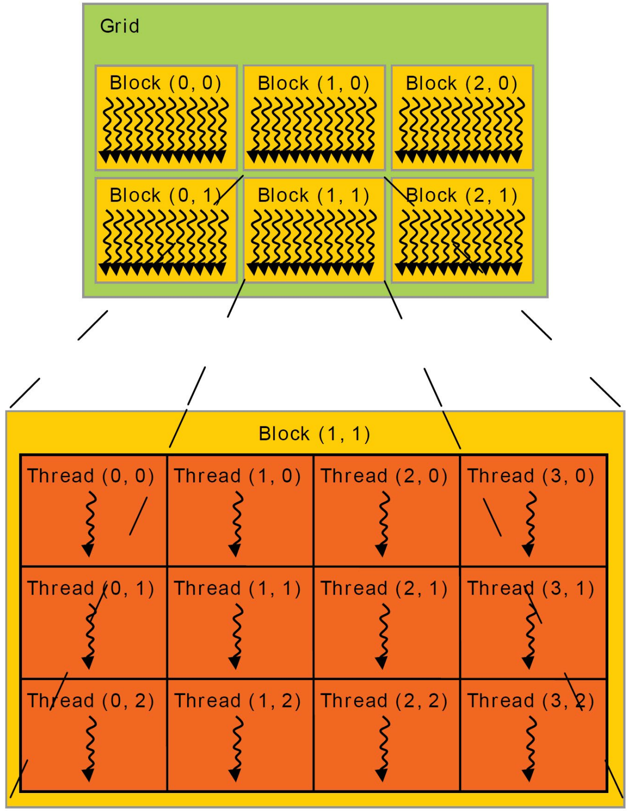
- Both paradigms are similar: write small, scalable kernels
- Kernel performance of both is very similar
- CUDA:
  - Runs best on Nvidia GPUs, but also supports all new features of them
  - Supports C++
  - Little call overhead and no device management necessary
- OpenCL:
  - Runs on GPU, CPU, and others like Cell, but to exploit performance the code has to be adapted to specific architecture
  - Code can be compiled at runtime
  - Can be easily integrated in existing code since no special compiler is necessary (like nvcc)
  - An open standard exists

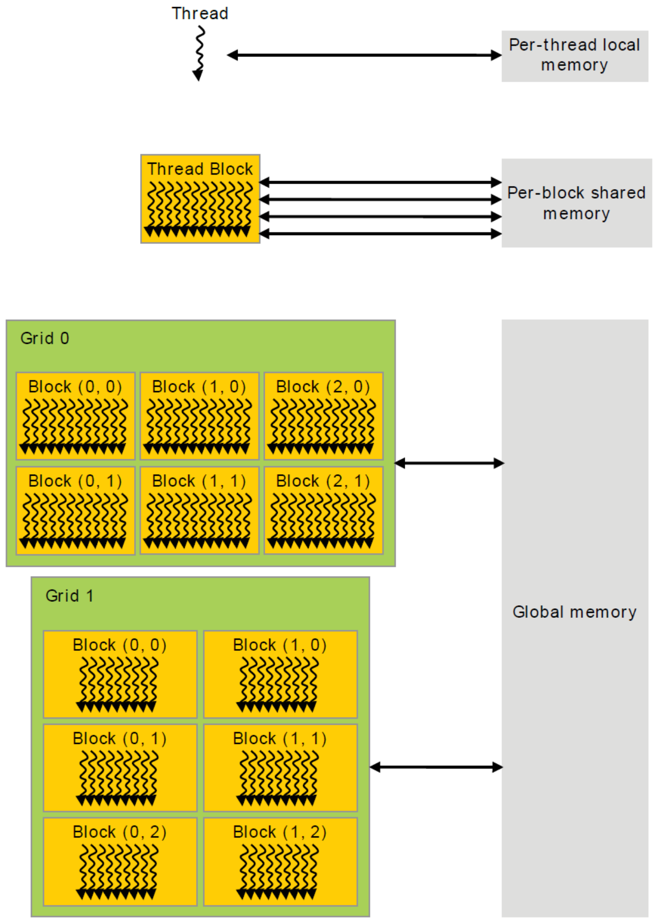
Automatic Scalability

GPU Features	NVIDIA Tesla P100	NVIDIA Tesla V100	NVIDIA A100
GPU Codename	GP100	GV100	GA100
GPU Architecture	NVIDIA Pascal	NVIDIA Volta	NVIDIA Ampere
Compute Capability	6.0	7.0	8.0
Threads / Warp	32	32	32
Max Warps / SM	64	64	64
Max Threads / SM	2048	2048	2048
Max Thread Blocks / SM	32	32	32
Max 32-bit Registers / SM	65536	65536	65536
Max Registers / Block	65536	65536	65536
Max Registers / Thread	255	255	255
Max Thread Block Size	1024	1024	1024
FP32 Cores / SM	64	64	64
Ratio of SM Registers to FP32 Cores	1024	1024	1024
Shared Memory Size / SM	64 KB	Configurable up to 96 KB	Configurable up to 164 KB



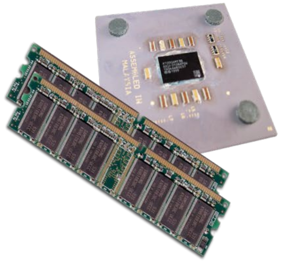
Grid of Thread Blocks



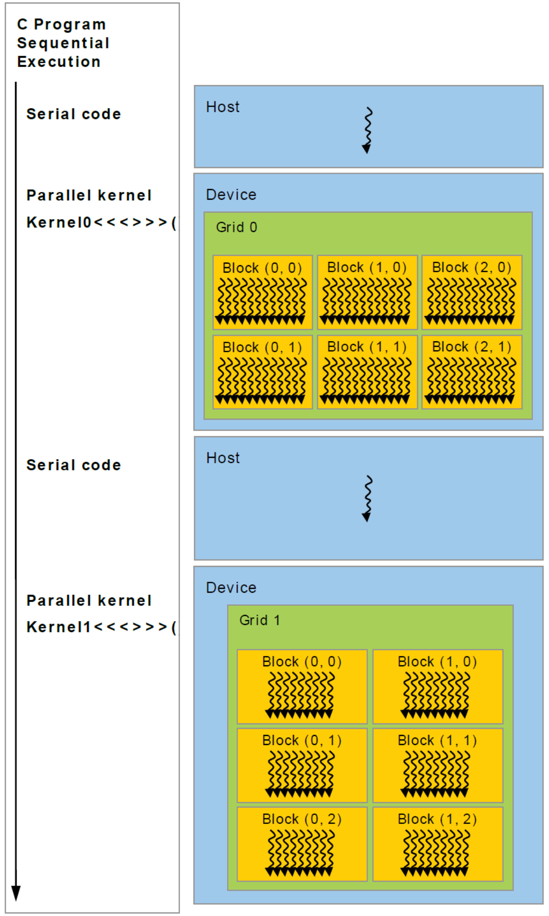


Heterogeneous Computing

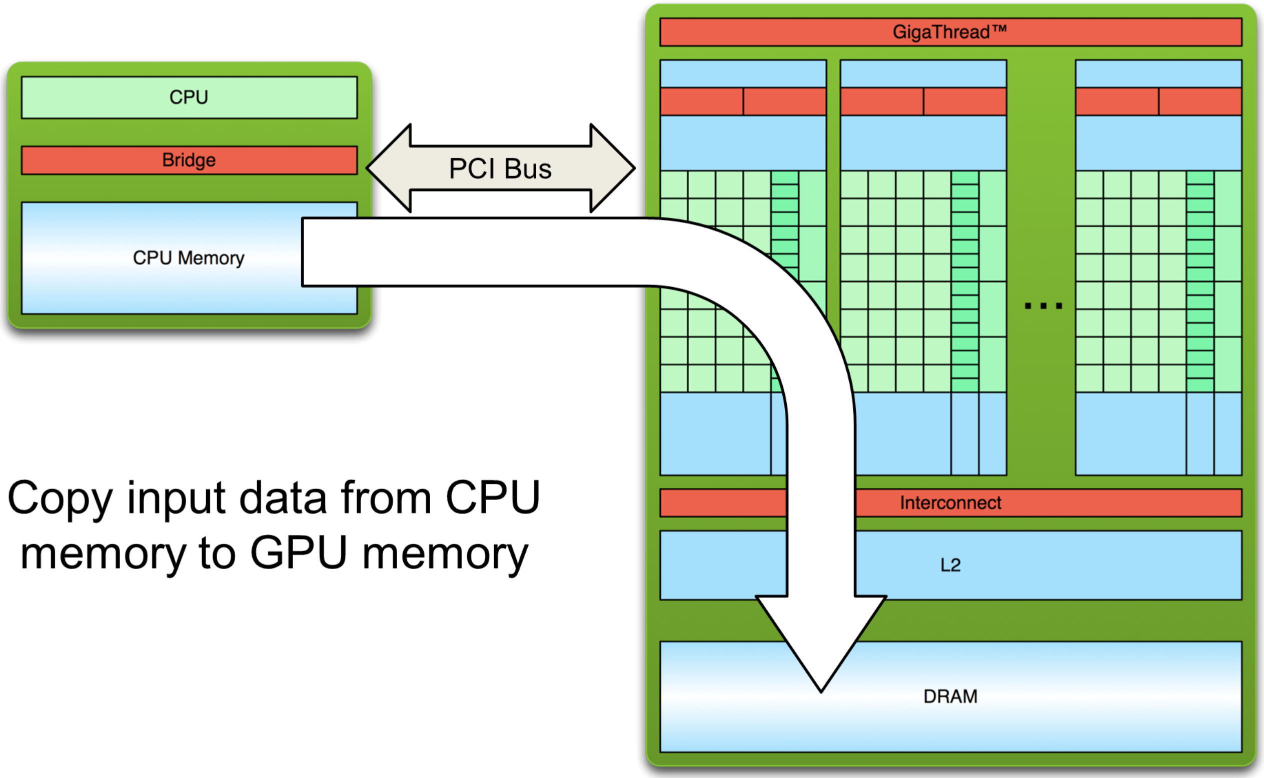
- Terminology:
  - *Host* The CPU and its memory (host memory)
  - *Device* The GPU and its memory (device memory)


Host

Device

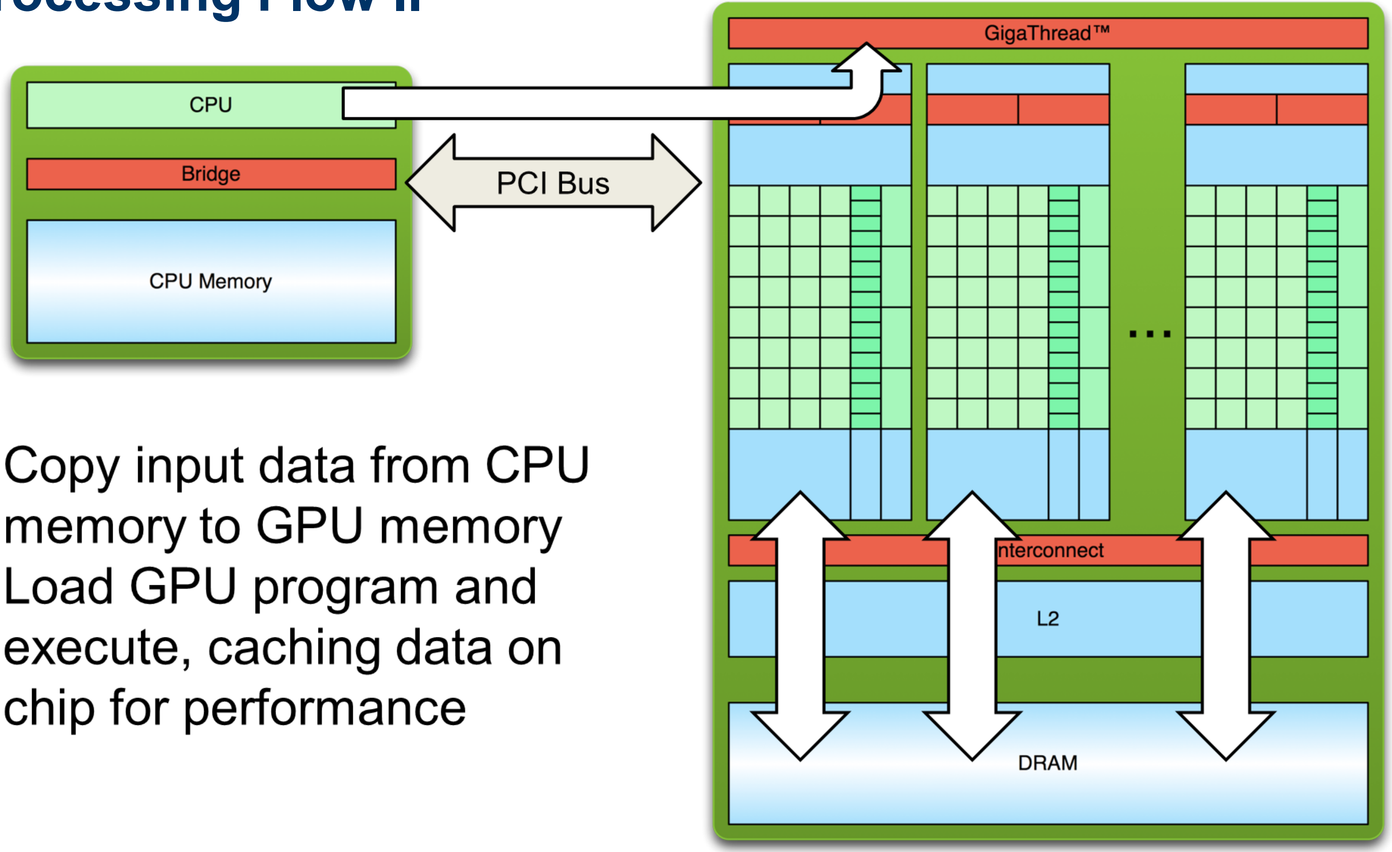


Simple Processing Flow I



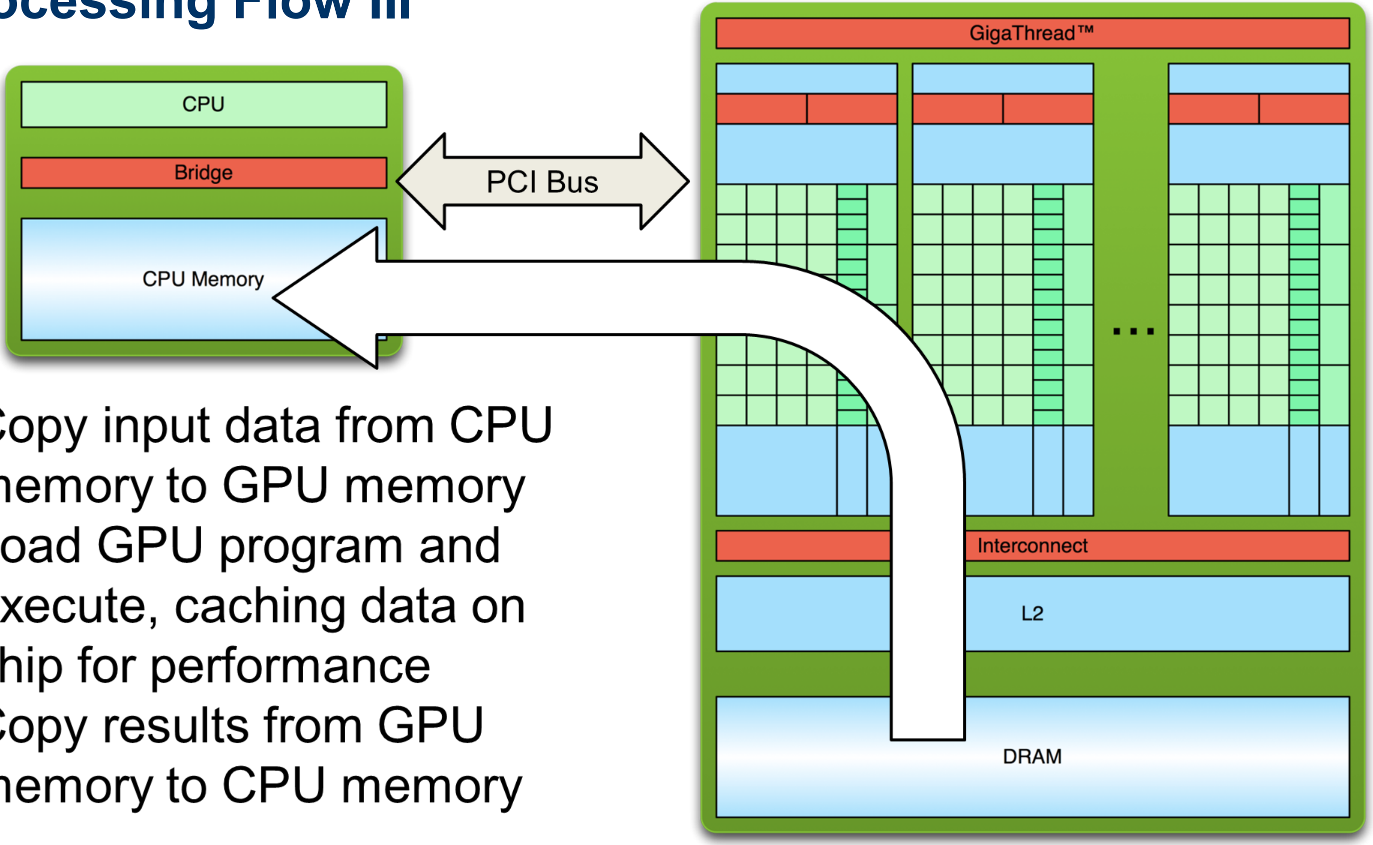
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow II



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

Simple Processing Flow III



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

CUDA Hello World!

```
int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no *device* code

Output:

```
$ nvcc hello_world.cu
$ a.out
Hello World!
```

CUDA Hello World! with Device Code

```
__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<<1,1>>>(); // grid dim, block dim
    printf("Hello World!\n");
    return 0;
}
```

- CUDA C/C++ keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host code

Next time…

- More on CUDA
- Introduction to OpenCL