POLITECNICO DI MILANO

GPGPU Computing

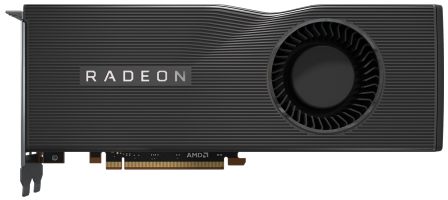
Advanced Computer Architectures

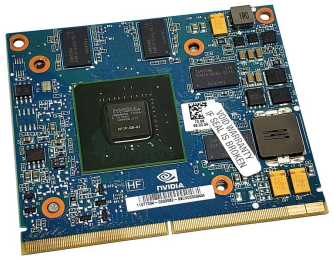
Serena Curzel

Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria
serena.curzel@polimi.it

What is a GPU?

- A Graphics Processing Unit (GPU) is a full device equipped with a highly parallel microprocessor (many-core) and high-bandwidth private memory
  - High streaming throughput
  - Fine-grain SIMD parallelism
  - Low-latency floating point (FP) computation
- Born in response to the growing demand for high-definition 3D rendering graphic applications





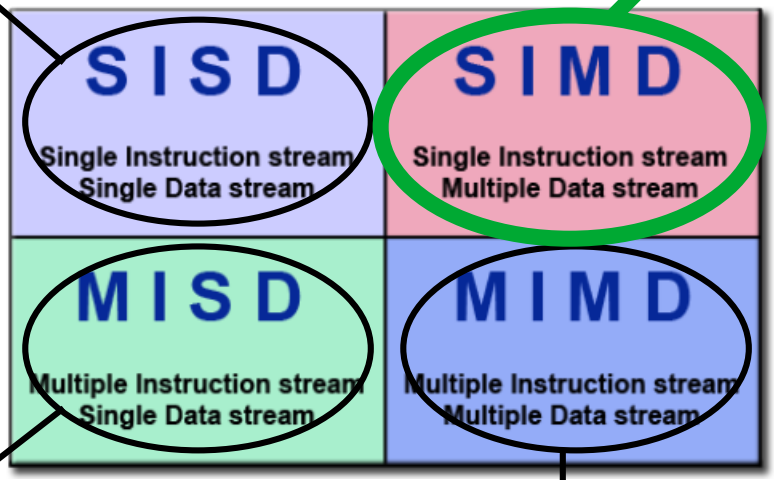
What is a GPU?

- GPUs are specialized for parallel intensive computation
- Rendering complex 3D scenes implies:
  - Millions of data points
  - Fast frame rate (60-120 FPS)
  - Sequence of transformations based on linear algebra operations and filters
  - Operations are applied on each point/vertex of the scene independently
- All operations are performed in parallel by the GPU using a huge number of **threads** processing all data independently

What is a GPU?

- Flynn's Taxonomy

Von Neumann architecture



Vector processors and **GPUs**

- Single Instruction or Single Program
- Multiple Data
- Same set of operations on each input point using multiple threads

Multi-core with redundant computation

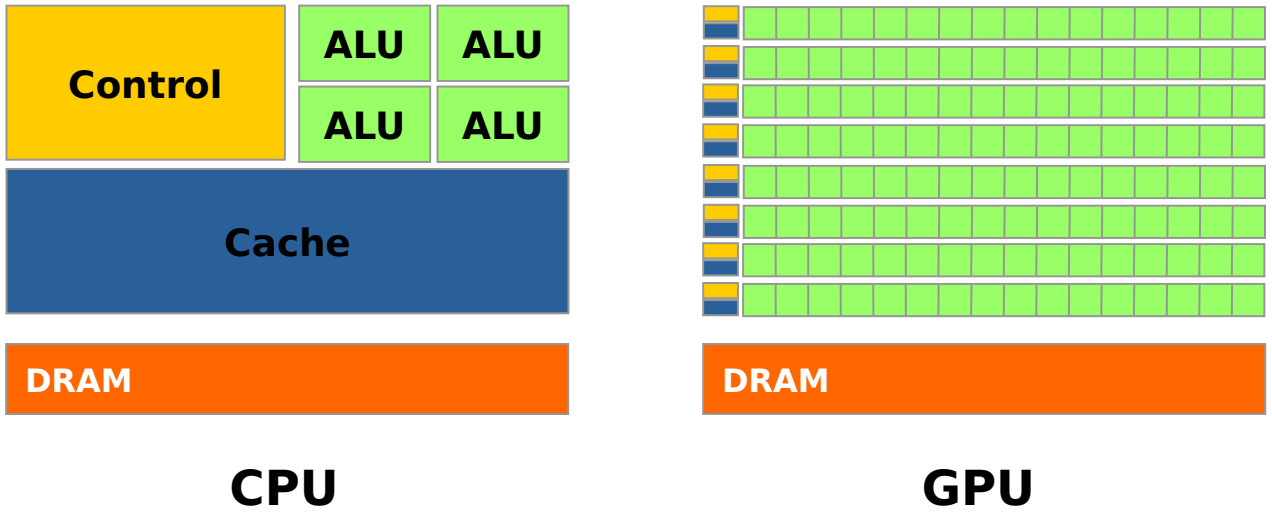
Multi-core architecture

CPU vs GPU

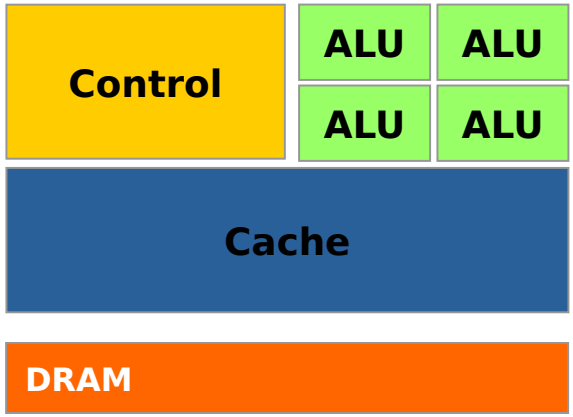
- Recap:
  - A thread is an independent unit of work that has local resources and shares resources of the main process
  - Threads communicate by exchanging data
  - Multiple threads run independently and synchronize at specific points
- Threads can run in parallel on a CPU if there is available hardware to execute them
  - Multiple cores in most modern chips
  - Context switch if there are more threads than available cores (expensive!)

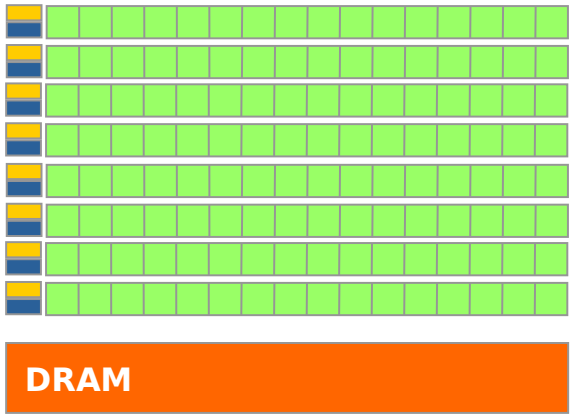
CPU vs GPU

- GPUs are specialized for intense data-parallel computations
  - Hardware is designed with a completely different perspective
  - Target is not *latency*, but *throughput*



CPU vs GPU





CPU

- Powerful ALU
  - Reduced operation latency
- Large hierarchical caches
  - Fast cache access instead of slow memory access
- Sophisticated control
  - Branch prediction
  - Data forwarding

GPU

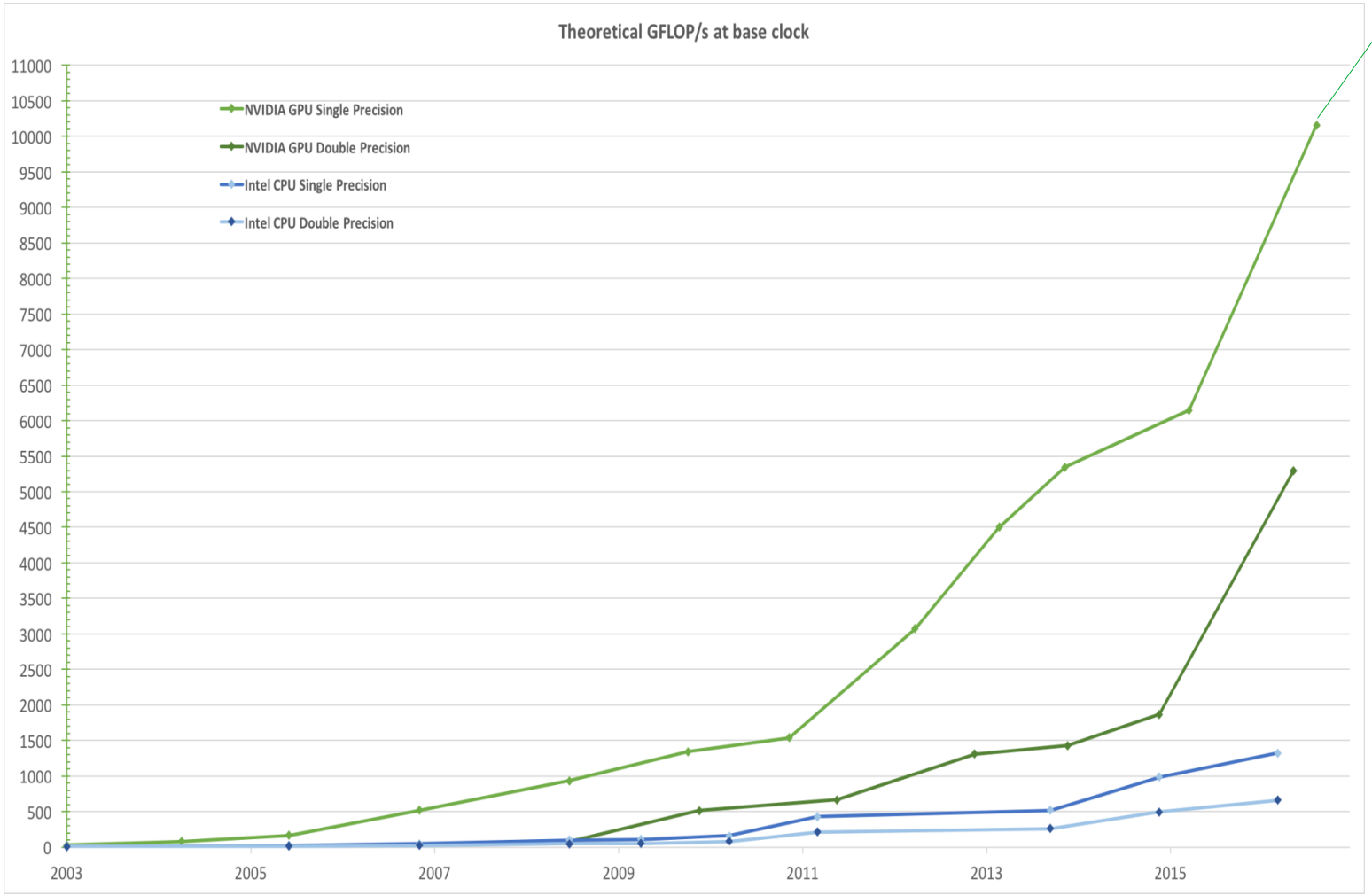
- Pipelined ALUs
  - Long latency, but high throughput
  - Rely on massive number of processing elements
- Small caches
  - High memory throughput
- Simple control
- Easy context switch

CPU vs GPU

- Visual demonstration: https://www.youtube.com/watch?v=-P28LKWTzrI&list=PLA6F24FAFFEB8F01B

- GPUs have evolved into powerful processors (and co-processors)
  - GFLOPS/TFLOPS performance
  - Programmable from high-level languages
  - IEEE 754 single and double precision floating point (half precision with some tweaks)

CPU vs GPU

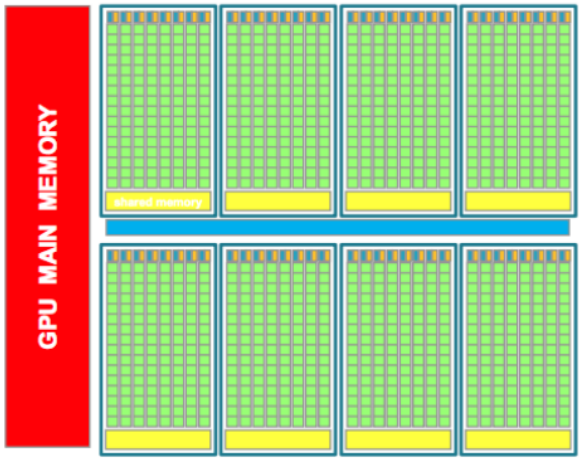


2020:

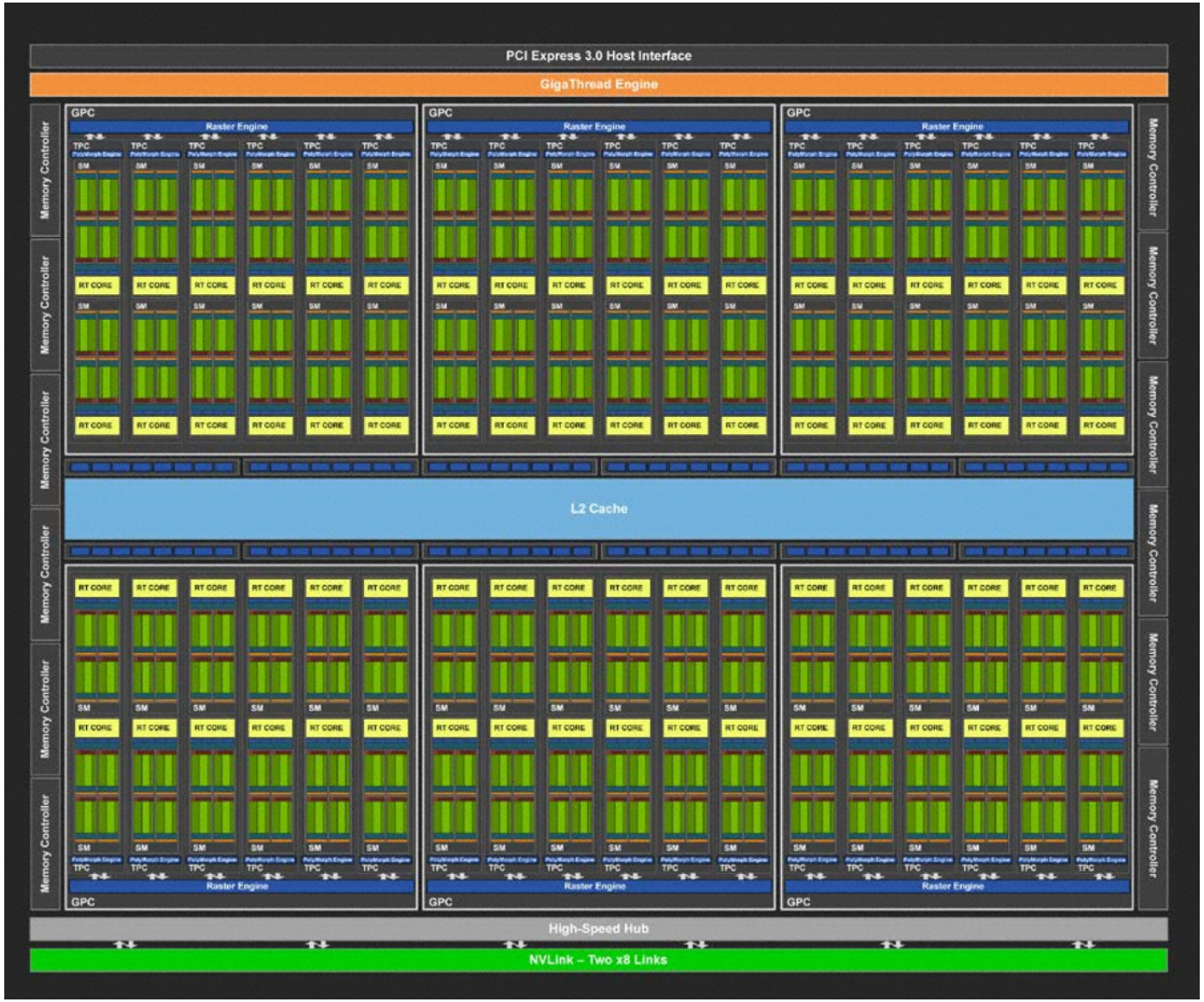
- AMD Radeon MI60 at 14.7 TFLOPS
- NVIDIA Ampere A100 at 19.5 TFLOPS

NVIDIA GPU Architecture

- Main Global Memory
  - Medium size (16 – 32 GB)
  - Very high bandwidth (250-800 GB/s)
- **Streaming Multiprocessors (SM)**
  - Multiple ALU cores (>100)
    - Floating point unit
    - Logic unit (add, sub, mul, madd)
    - Move, compare unit
    - Branch unit
  - Many registers (32k – 64k)
  - Shared memory with fast access
  - Instruction scheduler dispatchers



NVIDIA GPU Architecture



- NVIDIA Turing
  - 16.3 TFLOPS (FP32)
  - 72 SMs
  - 4608 cores
  - 175 W

NVIDIA GPU Architecture

- Recent High-End GPU models:

Vendor	MODEL	FP32 [Tflops]	Cores	Ram [GB]	Bw [GB/s]	Link
AMD	Radeon MI8	8.2	4096	4 HBM	512	PCIe3.0x16
AMD	Radeon MI25	12.3	4096	16 HBM2	484	PCIe3.0x16
AMD	Radeon MI50	13.3	3840	16 HBM2	1024	PCIe4.0x16
AMD	Radeon MI60	14.7	4096	32 HBM2	1024	PCIe4.0x16
NVIDIA	Kepler K40	4.3	2280	12 GDDR5	240	PCIe3.0x16
NVIDIA	Pascal P100	10.6	3584	16 HBM2	732	PCIe3.0x16
NVIDIA	Volta V100	15.7	5120	32 HBM2	900	PCIe3.0x16
NVIDIA	Ampere A100	19.5	6912	40GB HBM2	1600	PCIe4.0x16

GPUs vs Vector Processors

- Similarities:
  - Suitable for data-parallel problems
  - Scatter-gather transfers
  - Mask registers for control dependencies
  - Large register files
- Differences:
  - No scalar processor
  - Multithreading (*streams*) to hide memory latency
  - Many functional units instead of few fully pipelined ones

GPGPU computing

- Many scientific/technical applications process large data sets and use data-parallel programming models to speed up the computation
  - They include image processing and image rendering, but not only!
  - *Why not using the computational power of GPUs to accelerate other types of applications?*
- GPU graphics API were not suitable to scientific applications, but the idea remained valid
  - General Purpose GPU computing – GPGPU

GPGPU computing

Massive Data Parallelism

Instruction Level Parallelism

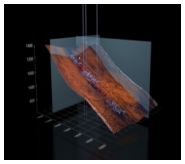
Graphics

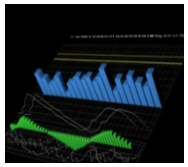
GPU
(Parallel Computing)

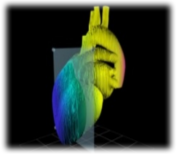
CPU
(Sequential Computing)

Data Fits in Cache

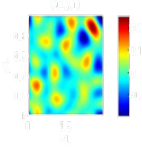
Larger Data Sets


Oil & Gas


Finance


Medical


Biophysics


Numerics


Audio


Video


Imaging

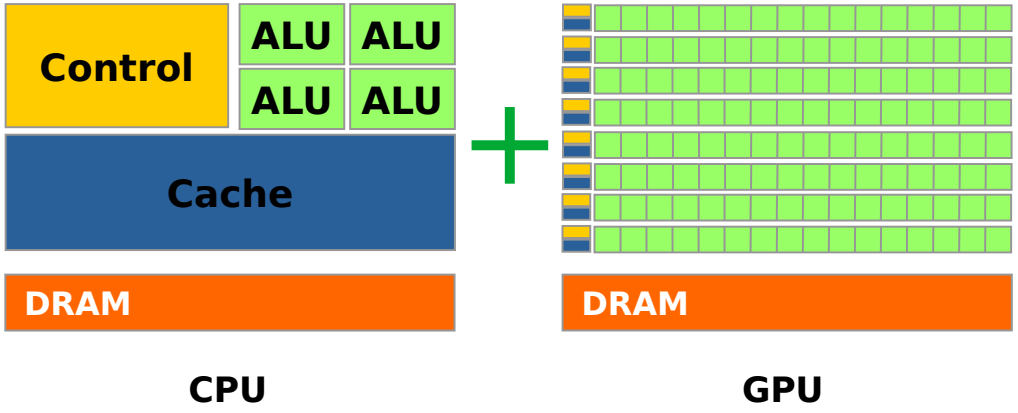
GPGPU computing

- GPGPU applications should have similar characteristics to graphics applications:
  - Extensive data parallelism
  - Few control instructions
  - Many math operations
  - Few synchronization points
  - Some task parallelism

GPGPU Programming Model

- The GPU (*device*) serves as a **coprocessor** for the CPU (*host*)
  - CPU and GPU are separate devices, with separate memory space addresses
  - The GPU has its own high-bandwidth memory
  - CPU and GPU should work together for maximum performance

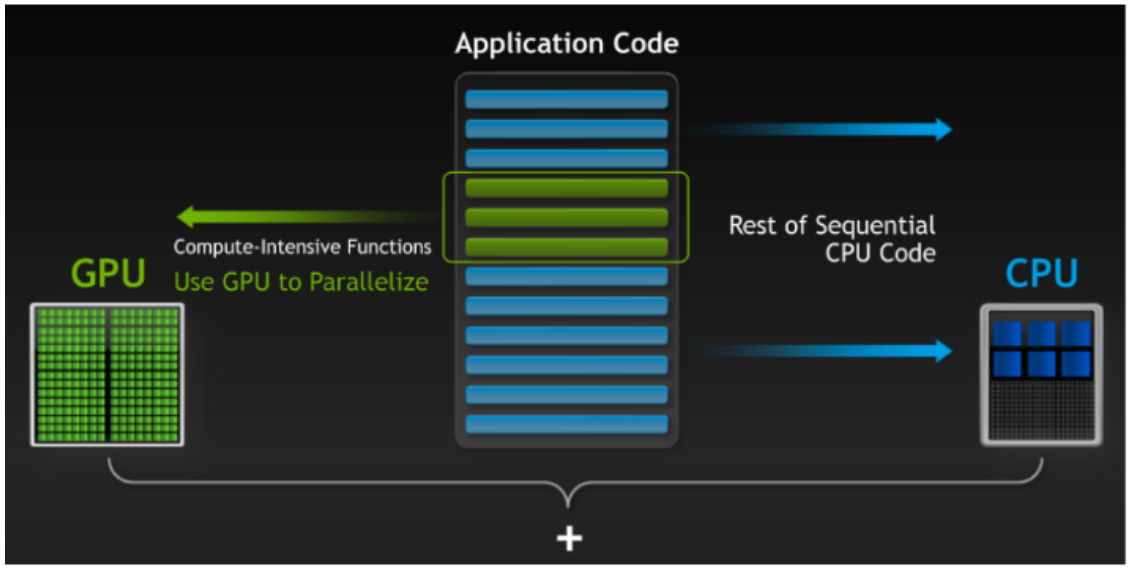
- Low-latency access to cached data
- Out-of-order and speculative execution
- Serial or event-driven tasks



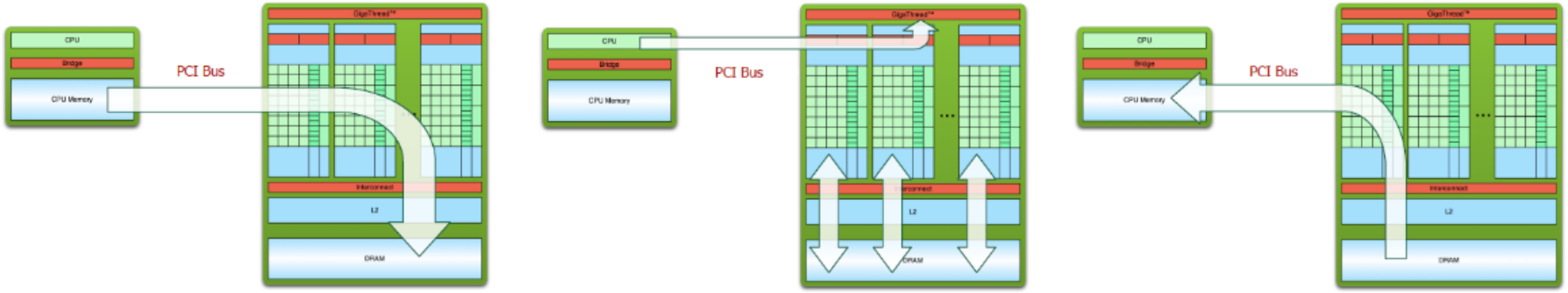
- High-throughput computation
- Hide memory latency
- Data-parallel tasks

GPGPU Programming Model

- Serial parts of a program run on the CPU (host)
- *Computation-intensive* and *data-parallel* parts are offloaded to the GPU (device)
- Data is moved between device memory and host memory when needed



GPGPU Programming Model



1) Copy input data from CPU memory to GPU memory

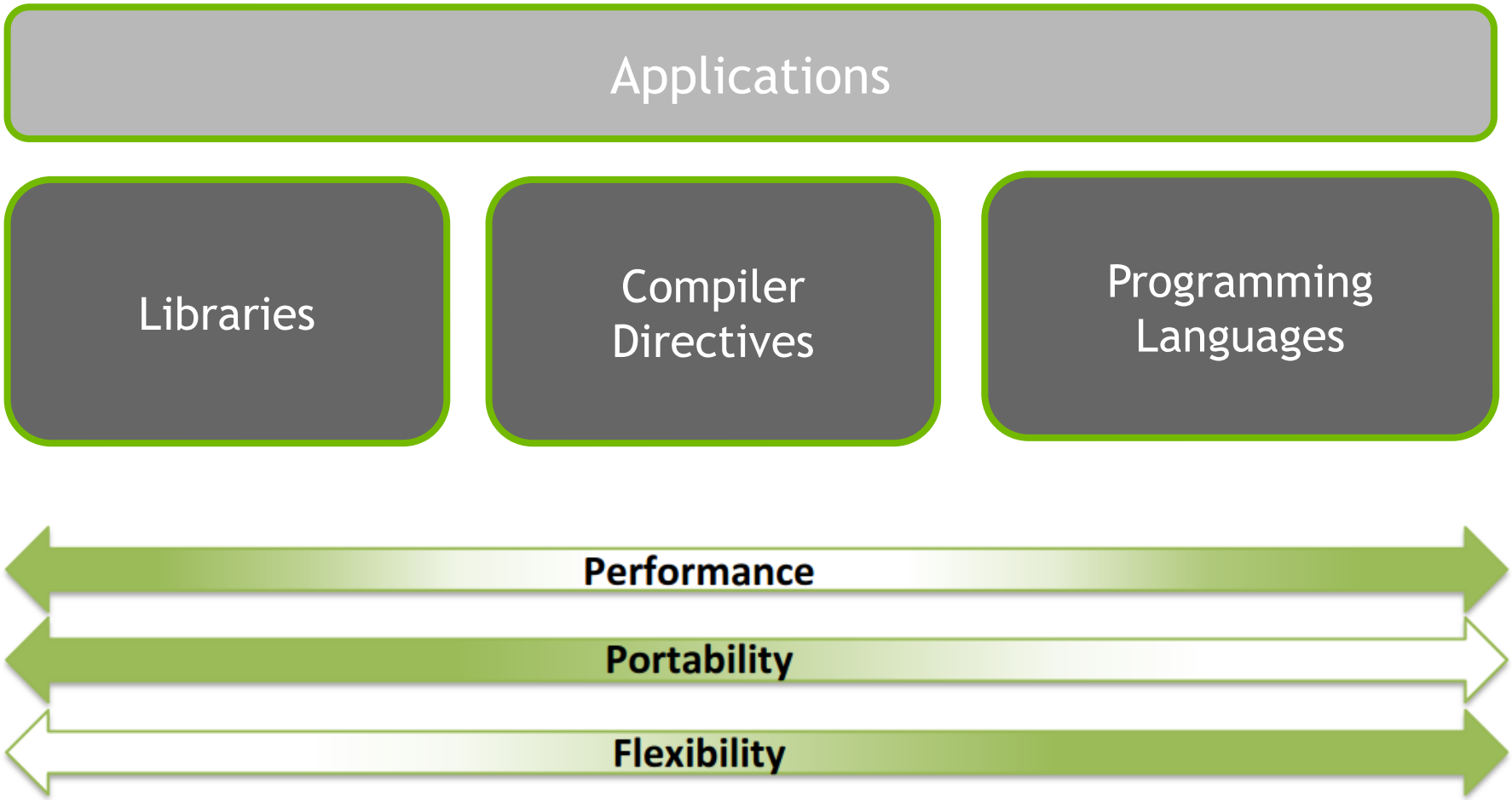
2) Load GPU program and execute, caching data on chip for performance

3) Copy results from GPU memory to CPU memory

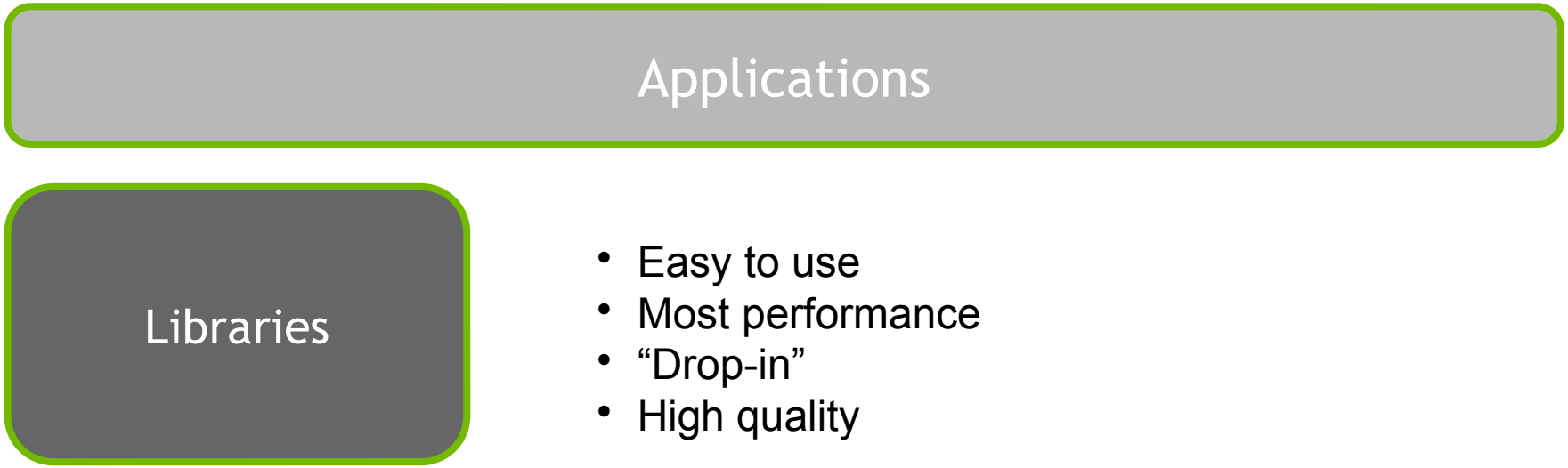
GPGPU Programming Model

- Data movement between CPU and GPU is the main bottleneck
  - Low bandwidth with respect to internal CPU and GPU since it exploits PCI Express (12-14GB/s)
  - Relatively high latency
  - *Data transfer can take more than the actual computation*
- This is especially a problem when porting CPU applications to GPGPU
  - Ignoring or not treating data movement seriously destroys GPU performance benefits
  - Some programming solutions may hide/automate data transfer

GPGPU Programming Paradigms

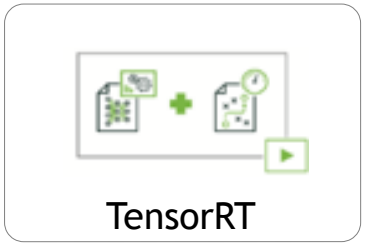


Libraries

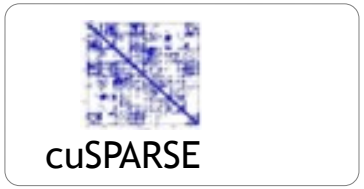

Applications
Libraries
Easy to use
Most performance
“Drop-in”
High quality
Performance
Portability
Flexibility

Libraries

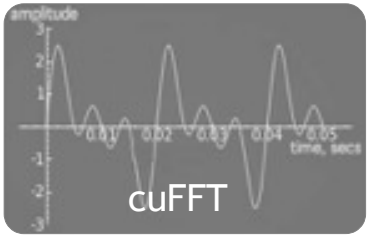
DEEP LEARNING



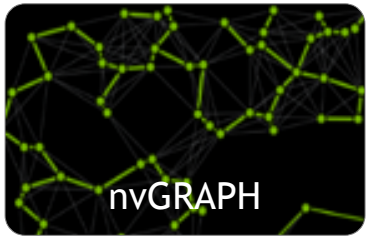
LINEAR ALGEBRA

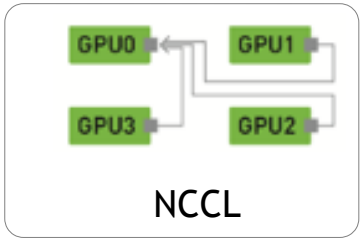


SIGNAL, IMAGE, VIDEO

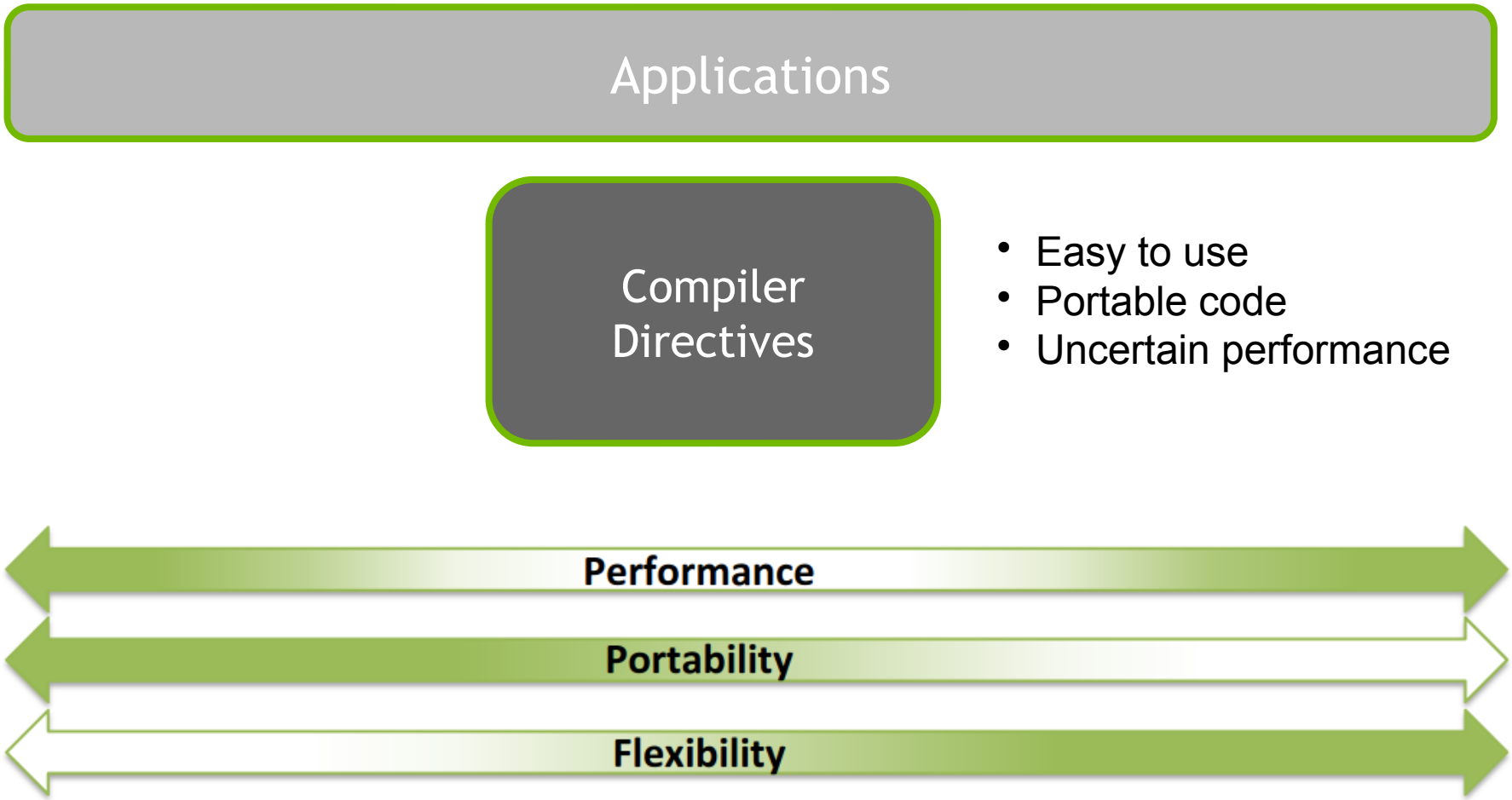


PARALLEL ALGORITHMS



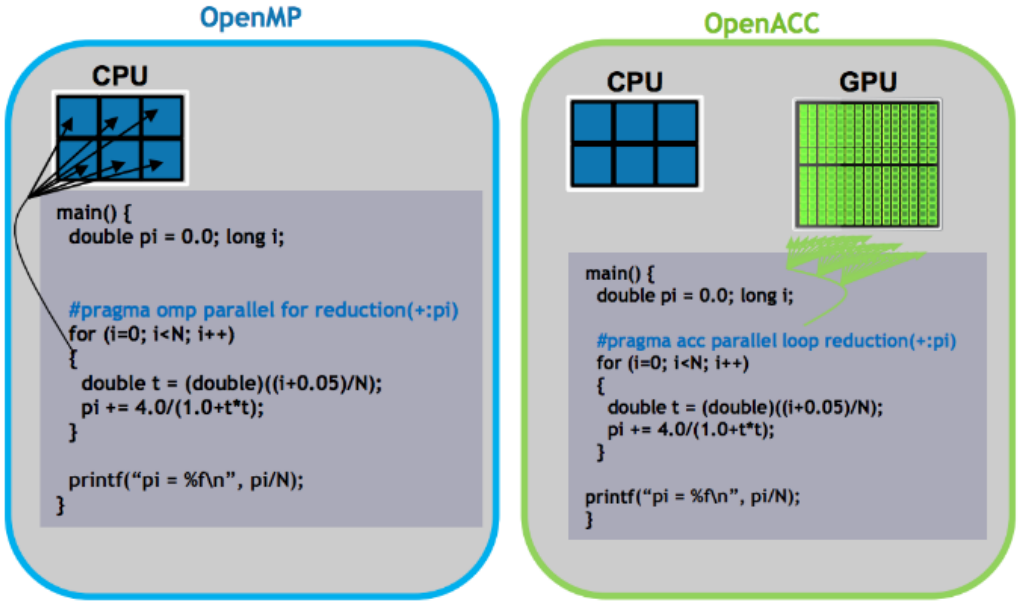


Compiler directives



Compiler directives

- OpenACC example (very similar to OpenMP):



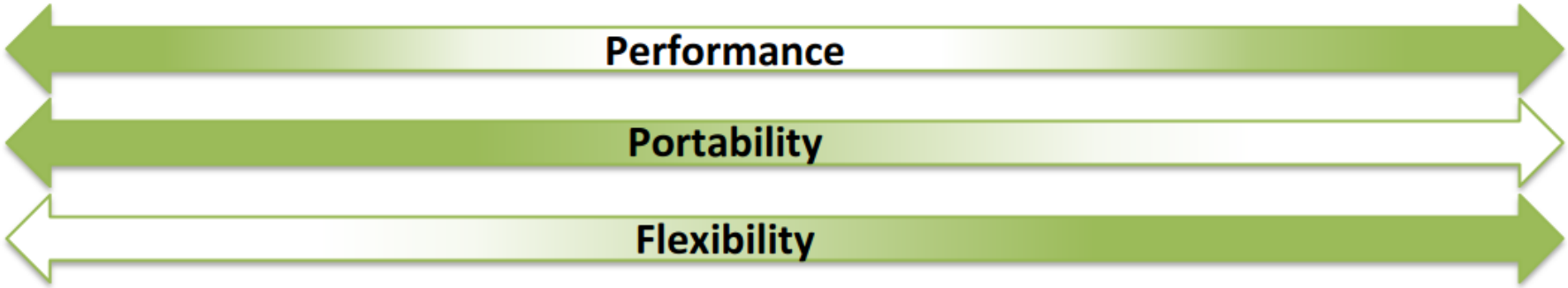

```
#pragma acc data copy(A), create(Anew)
while ( error > tol && iter < iter_max ) {
  error=0.0;

#pragma acc kernels loop gang(32), vector(16)
  for( int j = 1; j < n-1; j++) {
#pragma acc loop gang(16), vector(32)
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);
      error = max(error, abs(Anew[j][i] - A[j][i]);

    }
  }

#pragma acc kernels loop gang(16), vector(32)
  for( int j = 1; j < n-1; j++) {
#pragma acc loop
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
```

Programming Languages

Applications

- Most performance
- Most flexibility
- Verbose

Programming Languages



Programming Languages

Numerical analytics	MATLAB, Mathematica, LabVIEW
Python	PyCUDA, Copperhead, Numba
Fortran	CUDA Fortran
C	CUDA C, OpenCL
C++	CUDA C++, OpenCL, SYCL, OneAPI
C#	Hybridizer

CUDA basics

- Compute Unified Device Architecture
- Parallel computing architecture and programming model that expose the computational horsepower of **NVIDIA GPUs**
- Has its own a C/C++/Fortran compiler
- Enables GPGPU computing with minimal effort:
  - Write a program for one thread
  - Instantiate it on many parallel threads
  - Low learning curve, no knowledge of graphics is required
- The programming model evolves to reflect the underlying hardware architecture

CUDA basics

- A function which runs on a GPU is called **kernel**
  - When a kernel is launched on a GPU thousands of threads will execute its code
  - The programmer decides the number of threads
  - Each thread acts on different data elements independently (SIMD/SPMD/SIMT parallelism)

```
void vsum(int* a, int* b, int* c){
  int i;
  for (i=0;i<N;i++){
    c[i]=a[i]+b[i];
  }
}

void main(){
   int va[N], vb[N], vc[N];
   ...
   vsum(va,vb,vc);
   ...
}
```

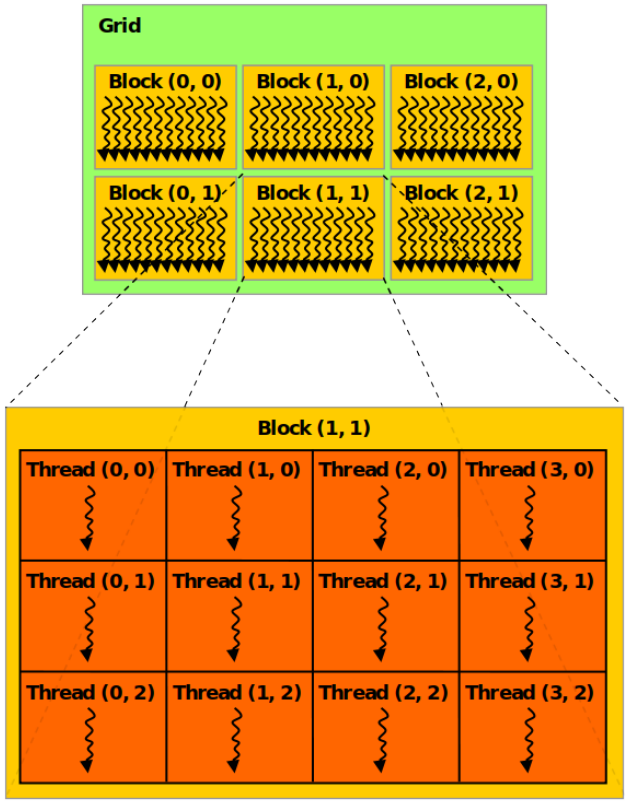
C program

```
__global__
void vsum(int* a, int* b, int* c){
  int i = ... // get unique thread ID
  c[i]=a[i]+b[i];
}

void main(){
   int va[N], vb[N], vc[N];
   ...
   vsum<<<1, 1>>>(va,vb,vc);
   ...
}
```

CUDA C program

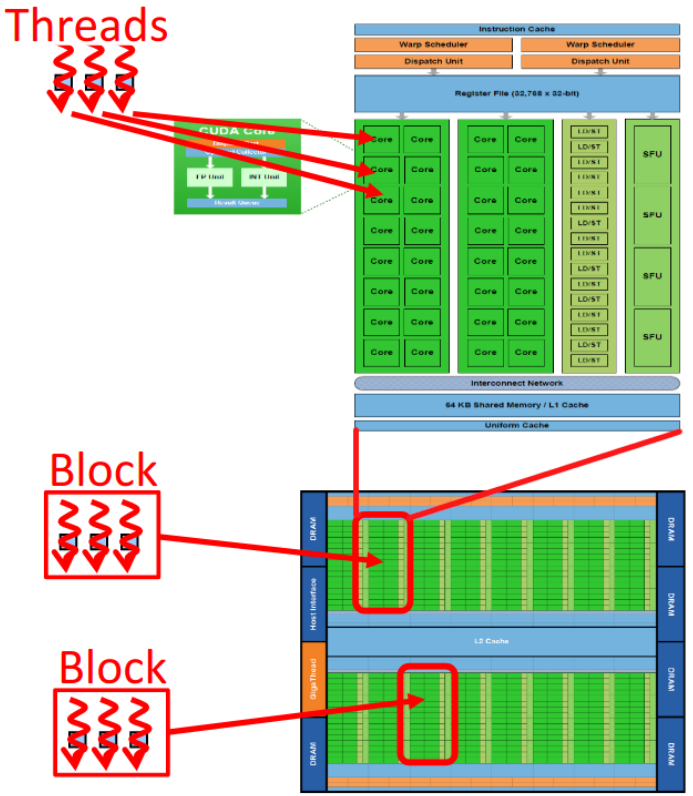
CUDA thread hierarchy

- CUDA **thread hierarchy**
  - *Threads* are grouped together in *blocks*
  - *Blocks* are grouped together in *grids*
  - Blocks and grids are organized as N-dimensional (up to 3) arrays
- Threads that belong to the same block can cooperate through a shared memory
- IDs are used to identify threads and blocks



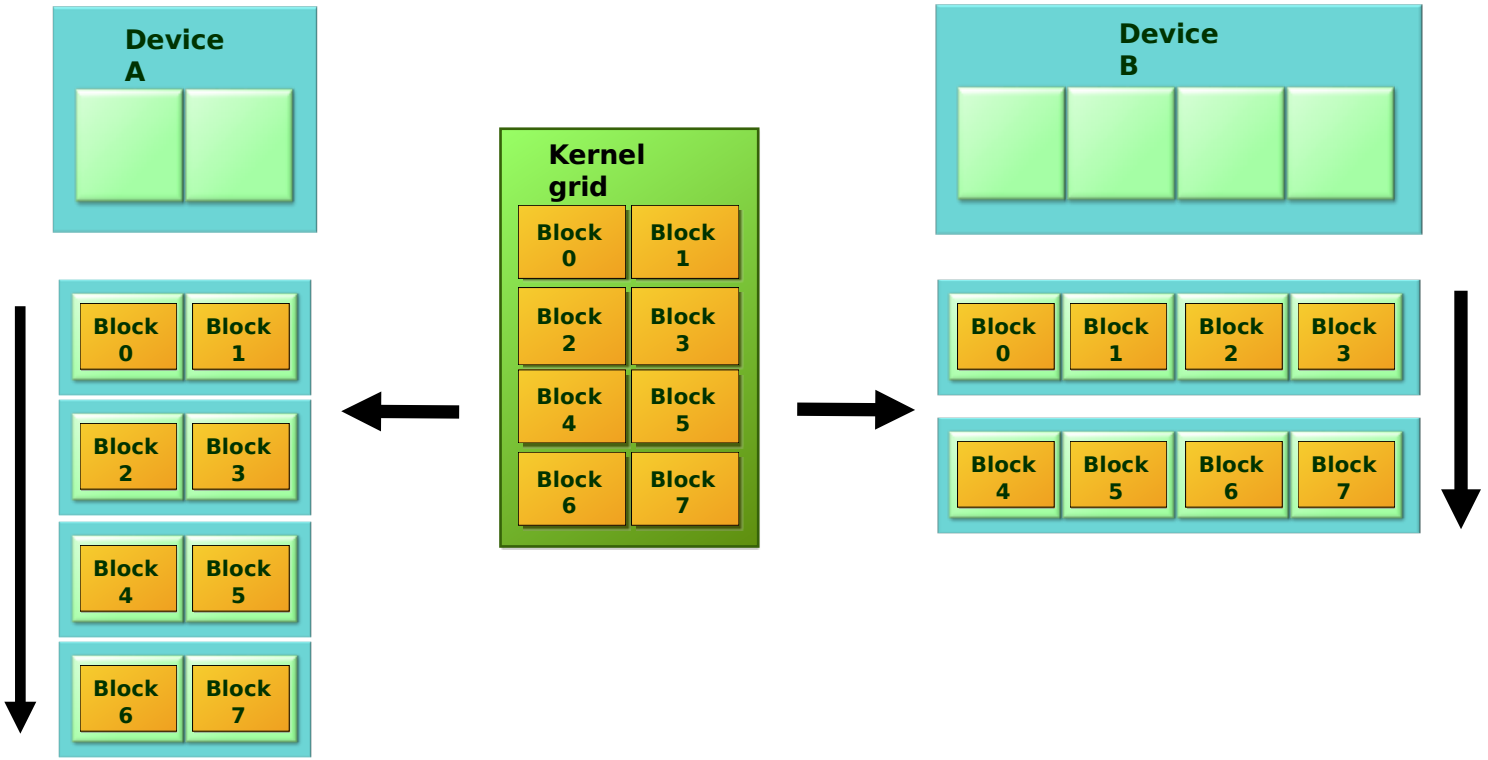
CUDA thread hierarchy

- CUDA thread hierarchy matches the underlying NVIDIA **architecture**
  - Each *core* executes a thread
  - Each *SM* executes one or more blocks
  - A *GPU* executes one or more grids
- Each SM executes threads of the same block in groups of 32 (*warps*)
- If there are more blocks than SMs, execution is interleaved

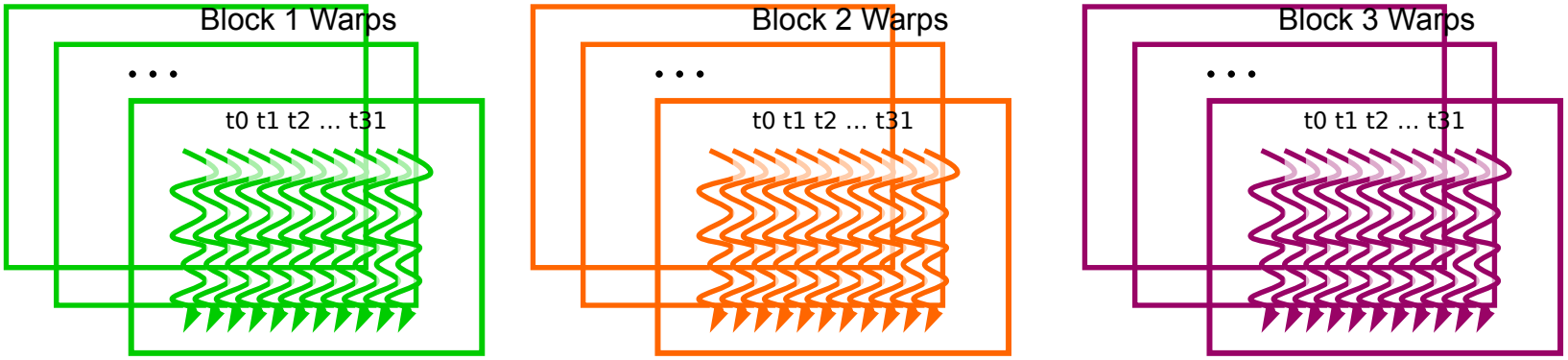


CUDA thread hierarchy

- Dividing threads into blocks improves scalability
  - GPU runtime transparently assigns blocks to available SMs



Warp scheduling



- Blocks are divided in warps
- Warps are scheduling units on the SM
- All threads in a warp execute the same instruction (SIMD)
  - Control unit for instruction fetch, decode, and control is shared among multiple processing units
  - Control overhead is minimized

Control divergence

- Control divergence occurs when threads in a warp take different control flow paths by making different control decisions
  - If path/else path
  - Different number of loop iterations
- The execution of threads taking different paths are serialized in current GPUs
  - The control paths taken by the threads in a warp are traversed one at a time
  - During the execution of each path, only the threads taking that path will be executed in parallel

Control divergence

- Example with control divergence:

```
if (threadIdx.x > 2) { }
```

  - Decision granularity less than warp size
  - Threads 0, 1 and 2 follow different path than the rest of the threads in the first warp

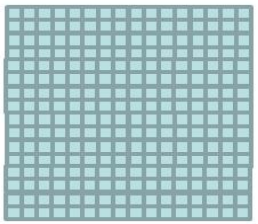
- Example without control divergence:

```
if (blockIdx.x > 2) { }
```

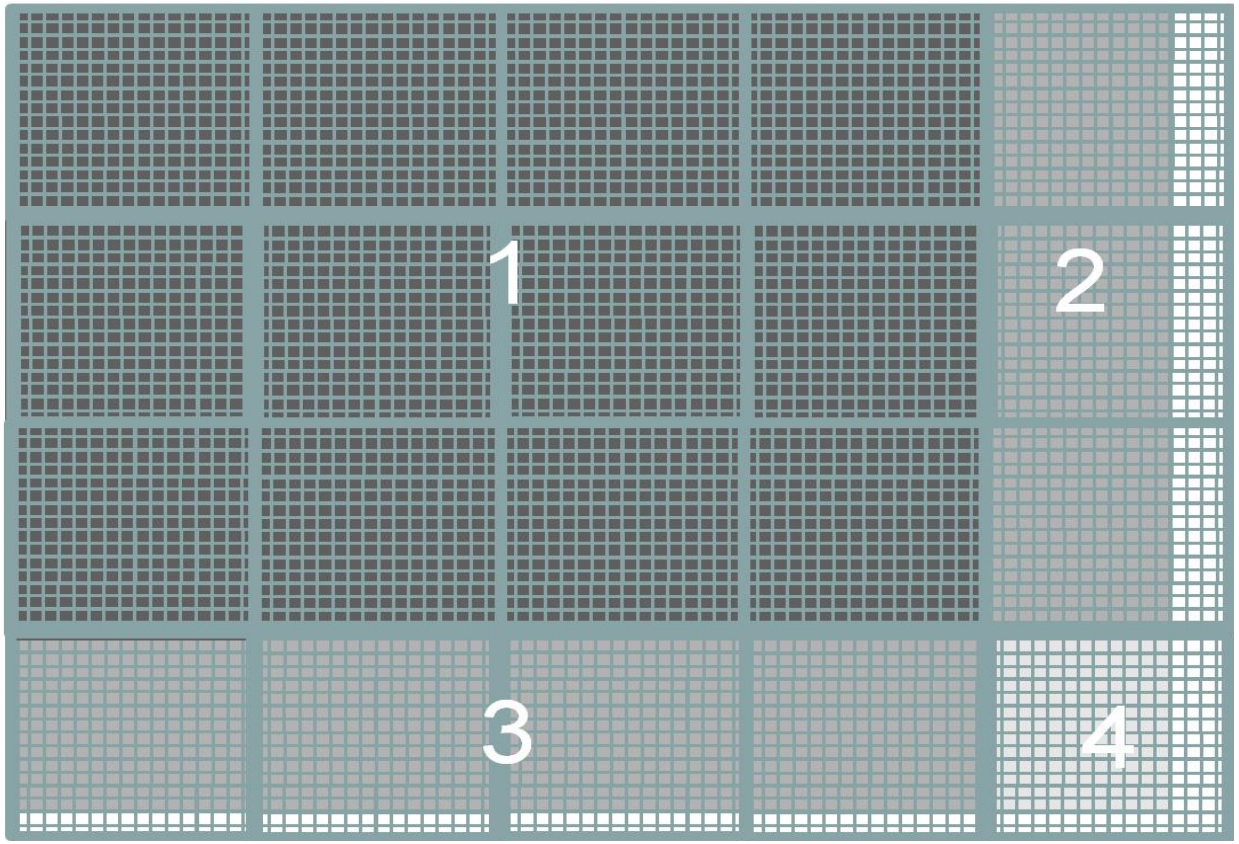
  - Decision granularity is a multiple of block size
  - All threads in any given warp follow the same path

Control divergence

- Covering a 62x76 picture with 16x16 blocks:

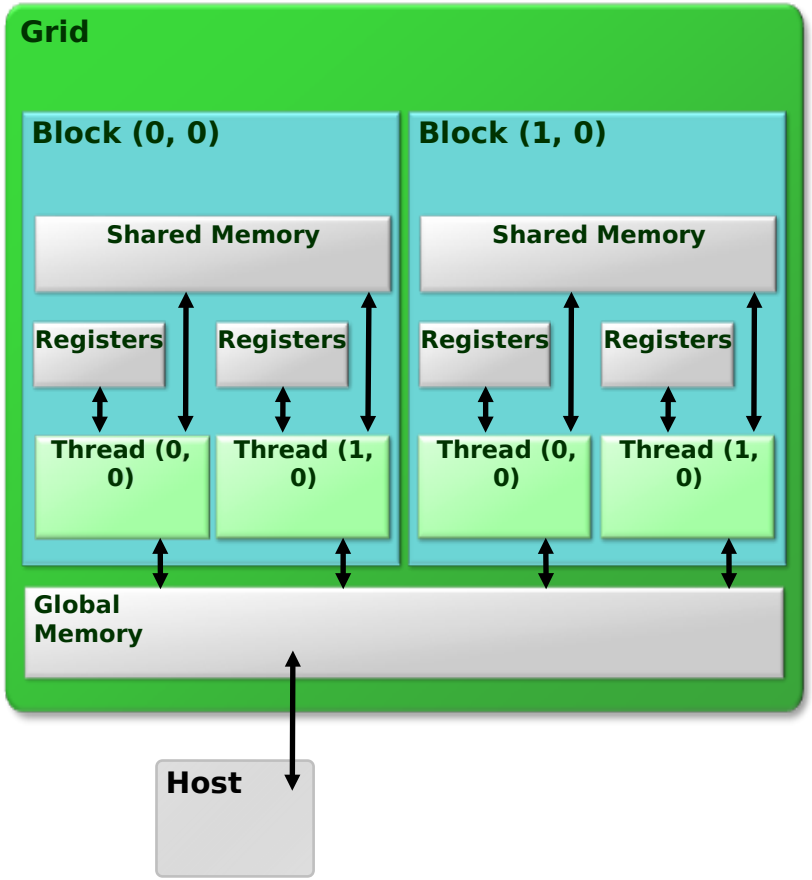


16x16 block

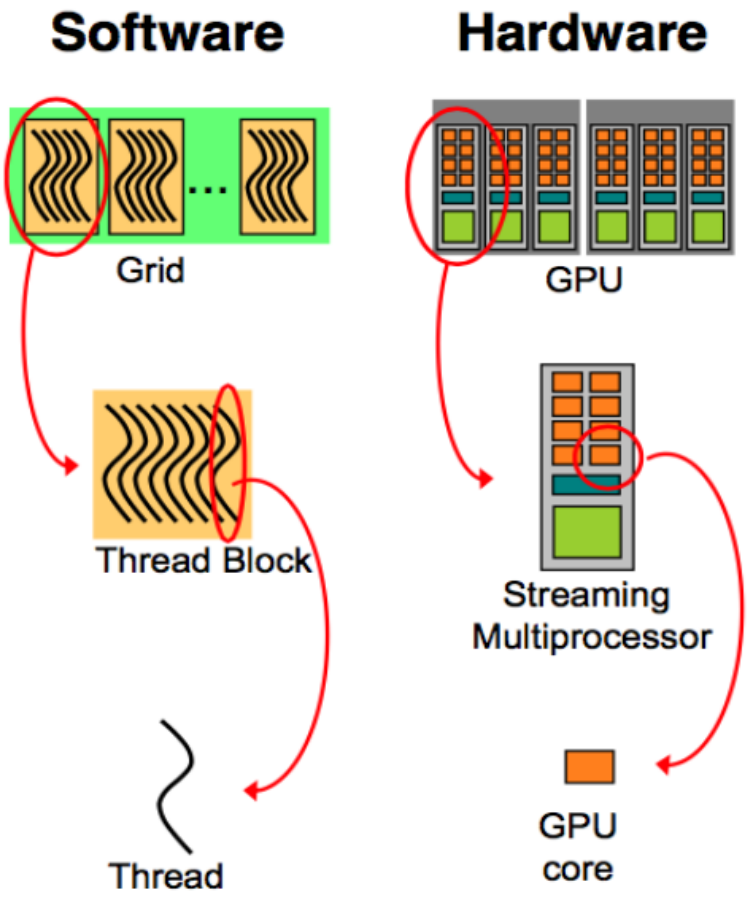


CUDA memory hierarchy

- Three types of memory are available, matching the hardware architecture
  - Per-thread Private Local Memory (Registers)
  - Per-block Shared Memory (Cache)
  - Global Memory (off-chip DRAM)
- Host can transfer data to/from global memory

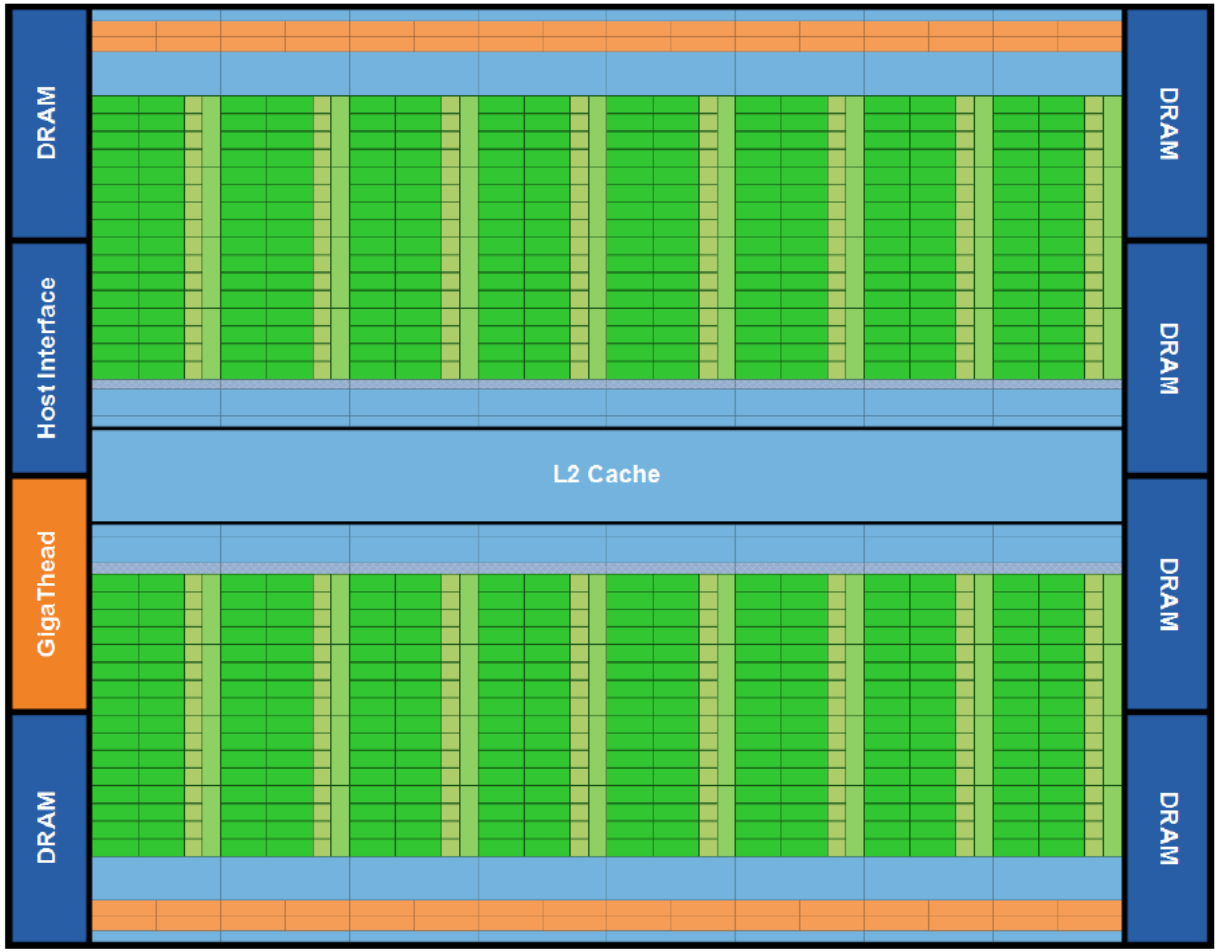


GPU execution model



- A GPU kernel is invoked:
  - Each thread block is assigned to a SM
  - When a block completes, the runtime system assigns another one to the SM
  - Threads within a block are divided in warps
  - The scheduler selects warps for execution on the SM cores

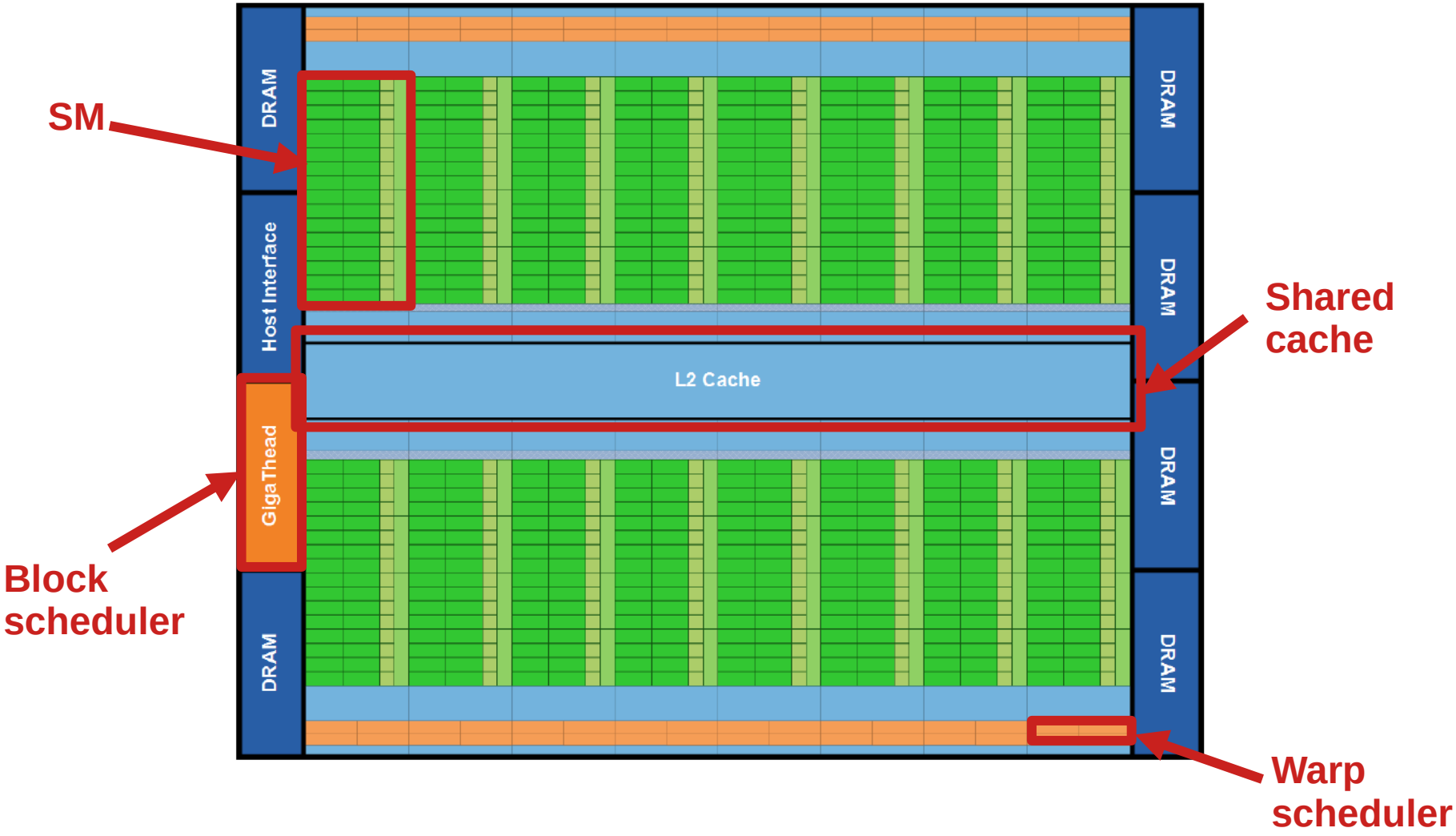
NVIDIA Fermi Architecture (2010)



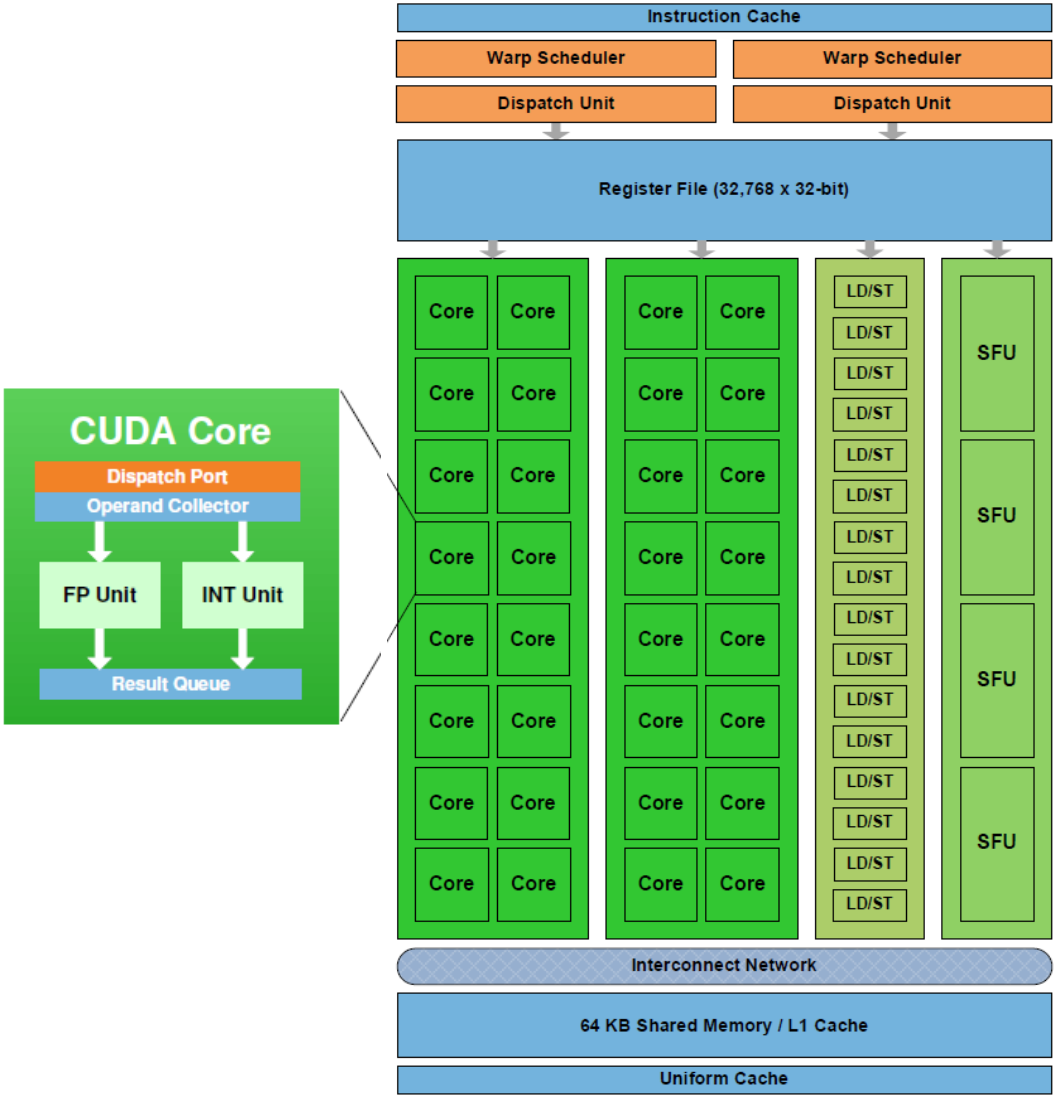
NVIDIA Fermi Architecture (2010)

- 16 Streaming Multiprocessors (multithreaded, SIMD)
- Each SM contains
  - 32 CUDA cores
  - 16 load/store units
  - Two warp schedulers
  - 4 special function units (SFUs)
- 512 CUDA cores in total
- Each Warp is composed of 32 threads
- Two warps can be scheduled in parallel on the same SM

NVIDIA Fermi Architecture (2010)



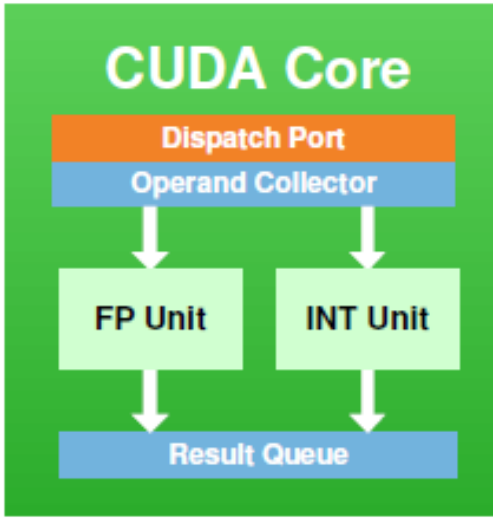
NVIDIA Fermi Architecture (2010)



Fermi Streaming Multiprocessor (SM)

NVIDIA Fermi Architecture (2010)

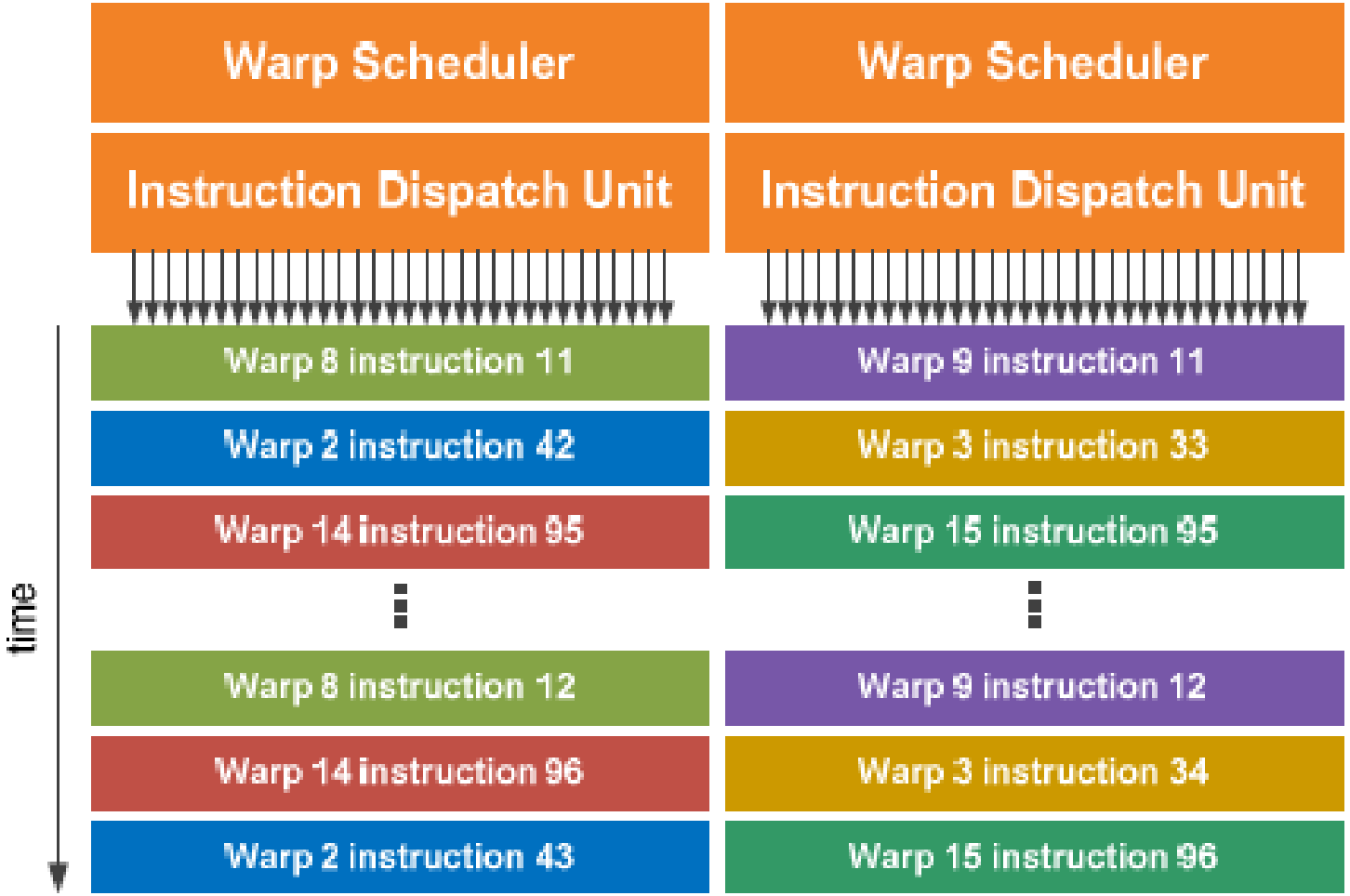
- Fully pipelined integer ALU
- Fully pipelined FPU
  - IEEE 754-2008 FP standard
  - Fused multiply-add (FMA) instruction in one cycle
  - 2 single precision operations or 1 double precision operation



Scheduling threads on the Fermi GPU

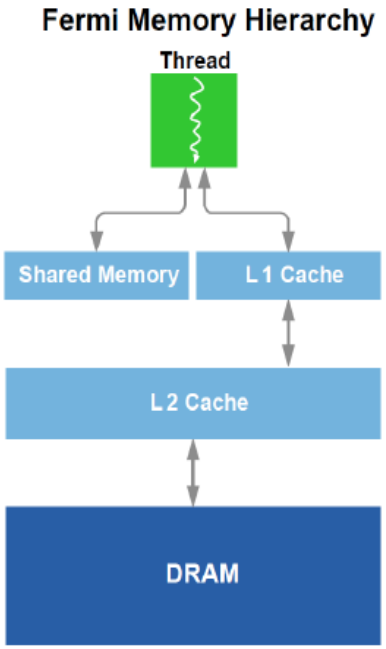
- Two levels of hardware schedulers
  - Giga Thread Engines (schedule blocks to SMs – also from different kernel)
  - Dual Warp Schedulers (schedule instructions from two warps to execution units)
- Threads are independent by definition, so there is no need to check for dependencies in the instruction stream
- Similar to a multi-threaded vector processor

Scheduling threads on the Fermi GPU



Fermi GPU memory hierarchy

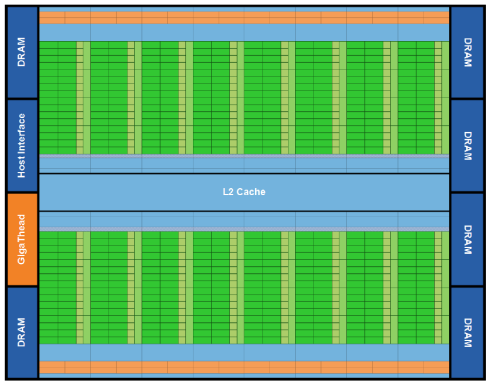
- Global memory (DRAM)
  - Accessible by host and kernel code
  - High latency
- L2 cache, 768 KB
  - Available to all 16 SMs
  - Used for data that is shared across blocks
- L1 cache/shared memory, 64KB
  - Available to a single block
  - Can hold spilled registers
- Registers, 32768x32b
  - 64 registers per thread, no overlap

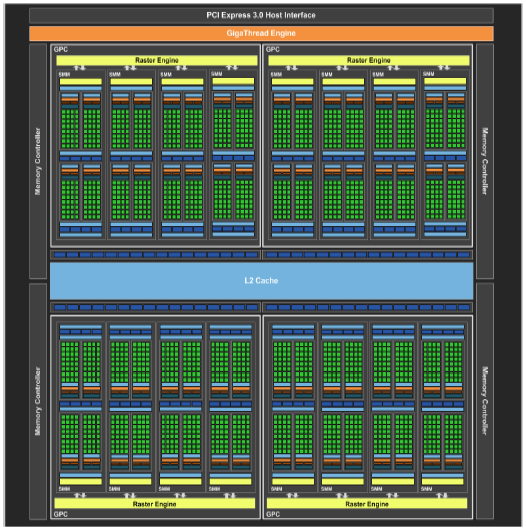


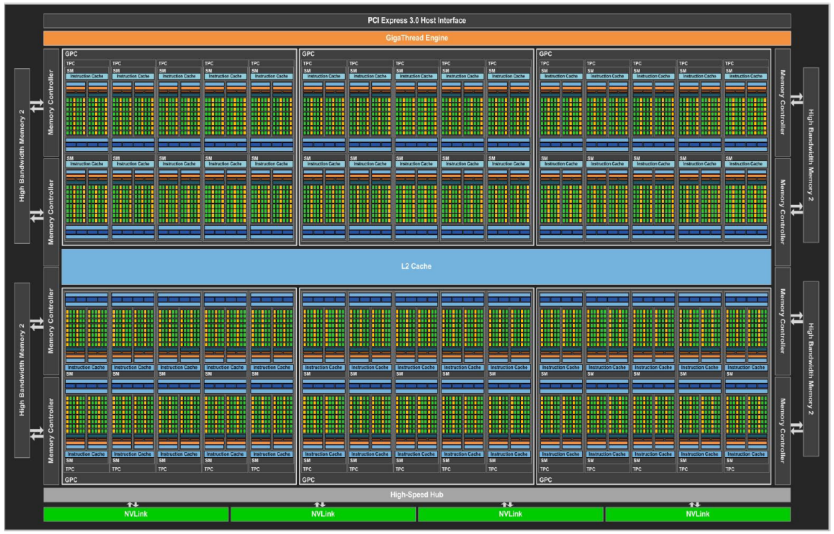
NVIDIA Fermi GPU

GPU	G80	GT200	Fermi
Transistors	681 million	1.4 billion	3.0 billion
CUDA Cores	128	240	512
Double Precision Floating Point Capability	None	30 FMA ops / clock	256 FMA ops /clock
Single Precision Floating Point Capability	128 MAD ops/clock	240 MAD ops / clock	512 FMA ops /clock
Warp schedulers (per SM)	1	1	2
Special Function Units (SFUs) / SM	2	2	4
Shared Memory (per SM)	16 KB	16 KB	Configurable 48 KB or 16 KB
L1 Cache (per SM)	None	None	Configurable 16 KB or 48 KB
L2 Cache (per SM)	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit

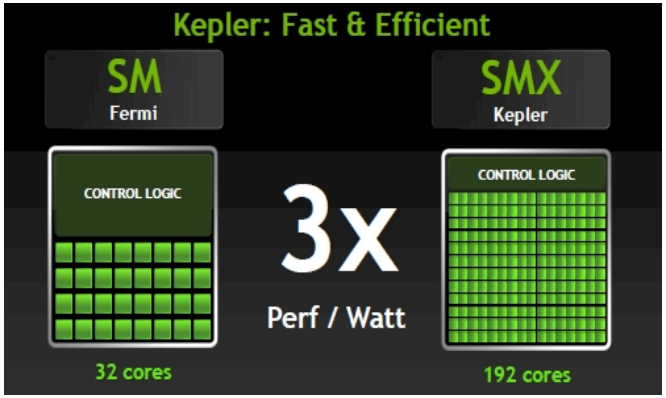
Evolution of NVIDIA GPUs







NVIDIA Kepler GPU

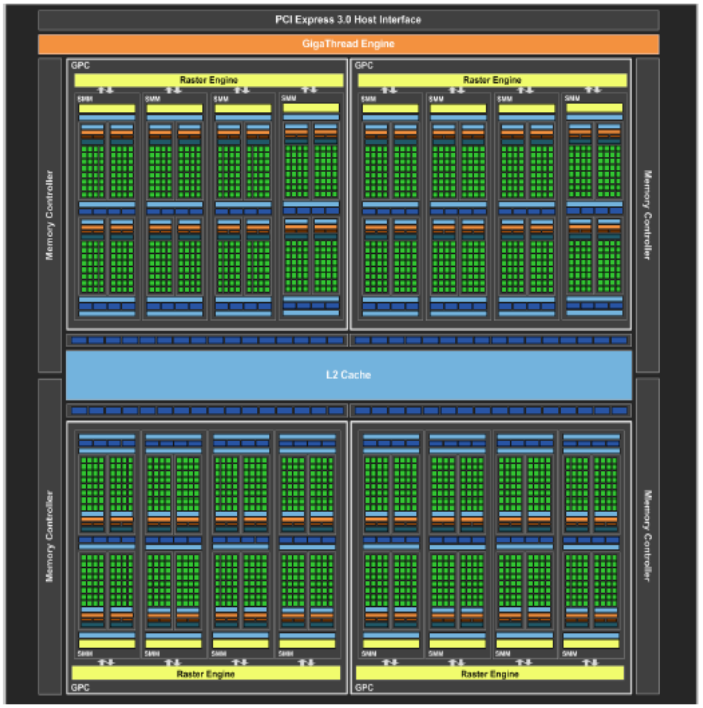


- GK110 architecture
- 15x192 = 2880 CUDA cores
- Larger caches
- Four warp schedulers

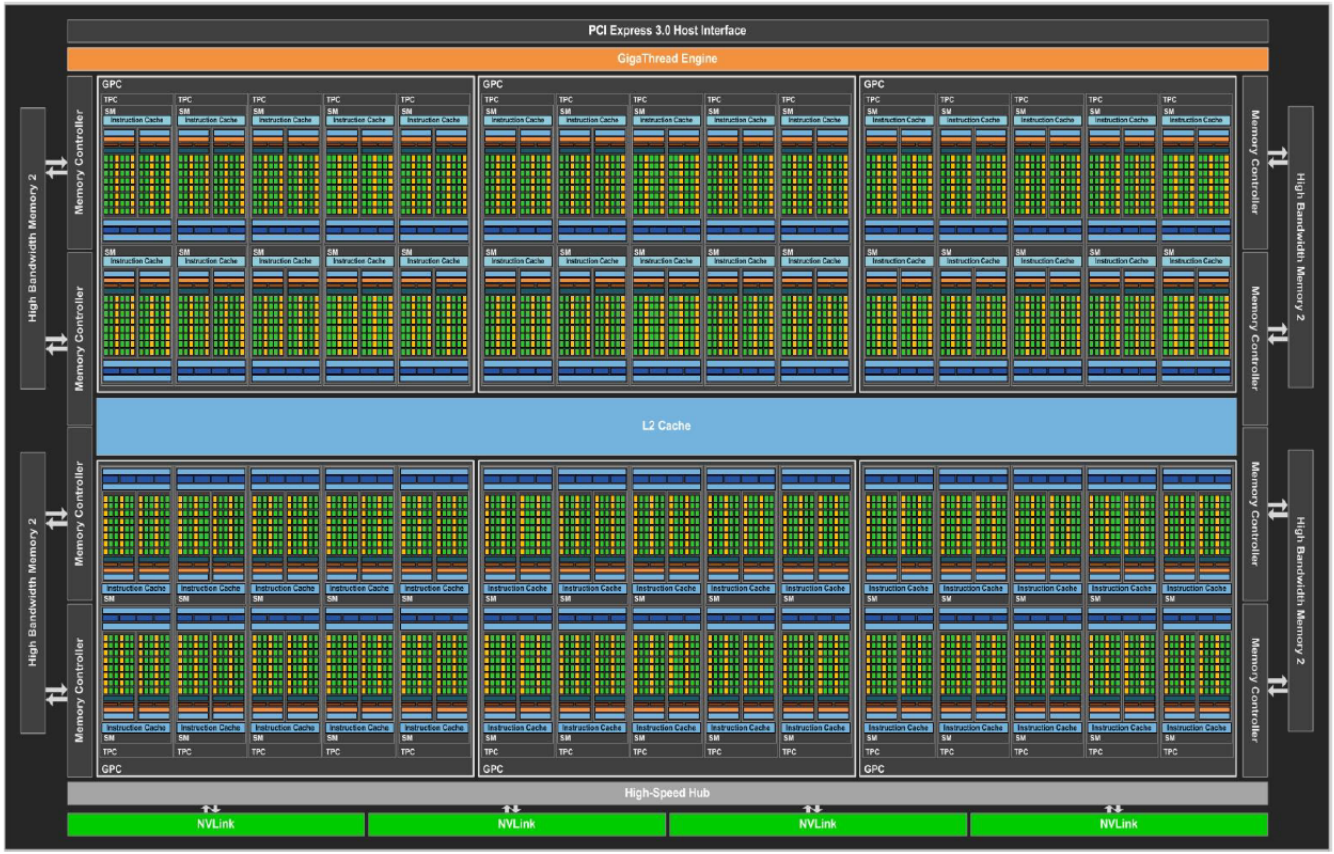
- Optimized for double-precision calculations
- Dynamic parallelism

NVIDIA Maxwell GPU

GPU	GeForce GTX 680 (Kepler)	GeForce GTX 980 (Maxwell)
SMs	8	16
CUDA Cores	1536	2048
Base Clock	1006 MHz	1126 MHz
GPU Boost Clock	1058 MHz	1216 MHz
GFLOPs	3090	4612[1]
Texture Units	128	128
Texel fill-rate	128.8 Gigatexels/sec	144.1 Gigatexels/sec
Memory Clock	6000 MHz	7000 MHz
Memory Bandwidth	192 GB/sec	224 GB/sec
ROPs	32	64
L2 Cache Size	512KB	2048KB
TDP	195 Watts	165 Watts
Transistors	3.54 billion	5.2 billion
Die Size	294 mm^2	398 mm^2
Manufacturing Process	28-nm	28-nm



NVIDIA Pascal GPU



- 3584 FP32 CUDA cores, 1792 FP64 CUDA cores

NVIDIA Volta GPU

- Optimized for Deep Learning

NVIDIA Volta GPU

- Optimized for Deep Learning
  - New SM architecture with higher performance and energy efficiency
  - Tensor Cores for matrix-matrix multiplications
  - Half-precision FP16
- Higher-bandwidth NVLink for multi-GPU systems

Evolution of NVIDIA GPUs

Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / SM	NA	NA	NA	8
Tensor Cores / GPU	NA	NA	NA	640
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz	1530 MHz
Peak FP32 TFLOPS[1]	5	6.8	10.6	15.7
Peak FP64 TFLOPS[1]	1.7	.21	5.3	7.8
Peak Tensor TFLOPS[1]	NA	NA	NA	125
Texture Units	240	192	224	320
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB	6144 KB
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB	Configurable up to 96 KB
Register File Size / SM	256 KB	256 KB	256 KB	256KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB	20480 KB
TDP	235 Watts	250 Watts	300 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion	21.1 billion
GPU Die Size	551 mm²	601 mm²	610 mm²	815 mm²
Manufacturing Process	28 nm	28 nm	16 nm FinFET+	12 nm FFN

Evolution of NVIDIA GPUs

- After Volta: Turing (2018) and Ampere (2020)
  - More acceleration for graphics (ray tracing)
  - Support for sparse matrix operations
  - More data types support in tensor cores
- Innovation still driven by graphics + deep learning

References

- NVIDIA Deep Learning Institute, “Accelerated Computing” Teaching Kit
- More self-learning material from NVIDIA can be found at: https://developer.nvidia.com/cuda-education
- D.B. Kirk and W.W. Hwu, “Programming Massively Parallel Processors. A hands-on approach”, 3rd edition, Morgan Kaufmann 2017