Computação Paralela

Mest. Engenharia Computacional
Mest. Int. Engenharia Computacional

Ano letivo 2022/2023

Rui Costa, Nuno Lau

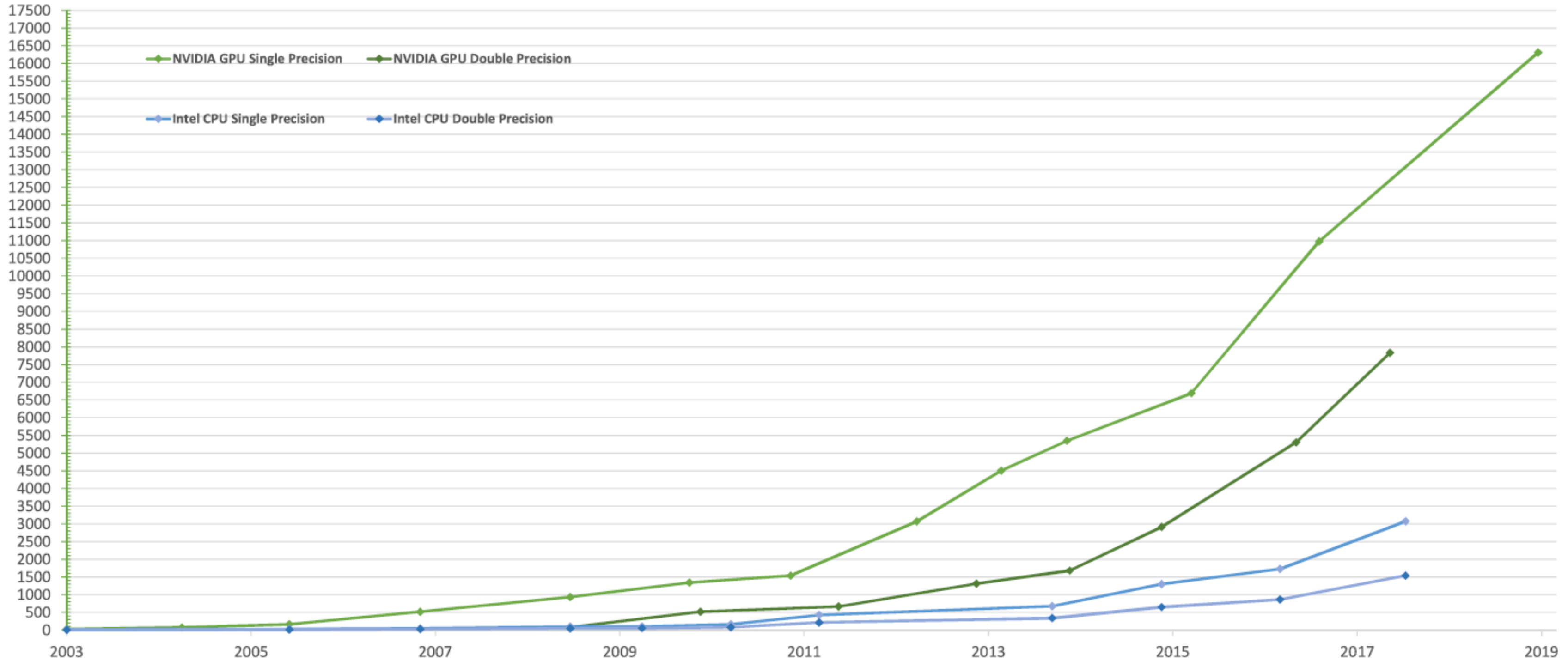
CUDA

- Parallel general purpose computation model
- Introduced by NVIDIA in 2006
- Allows using the GPU for the execution of general purpose applications

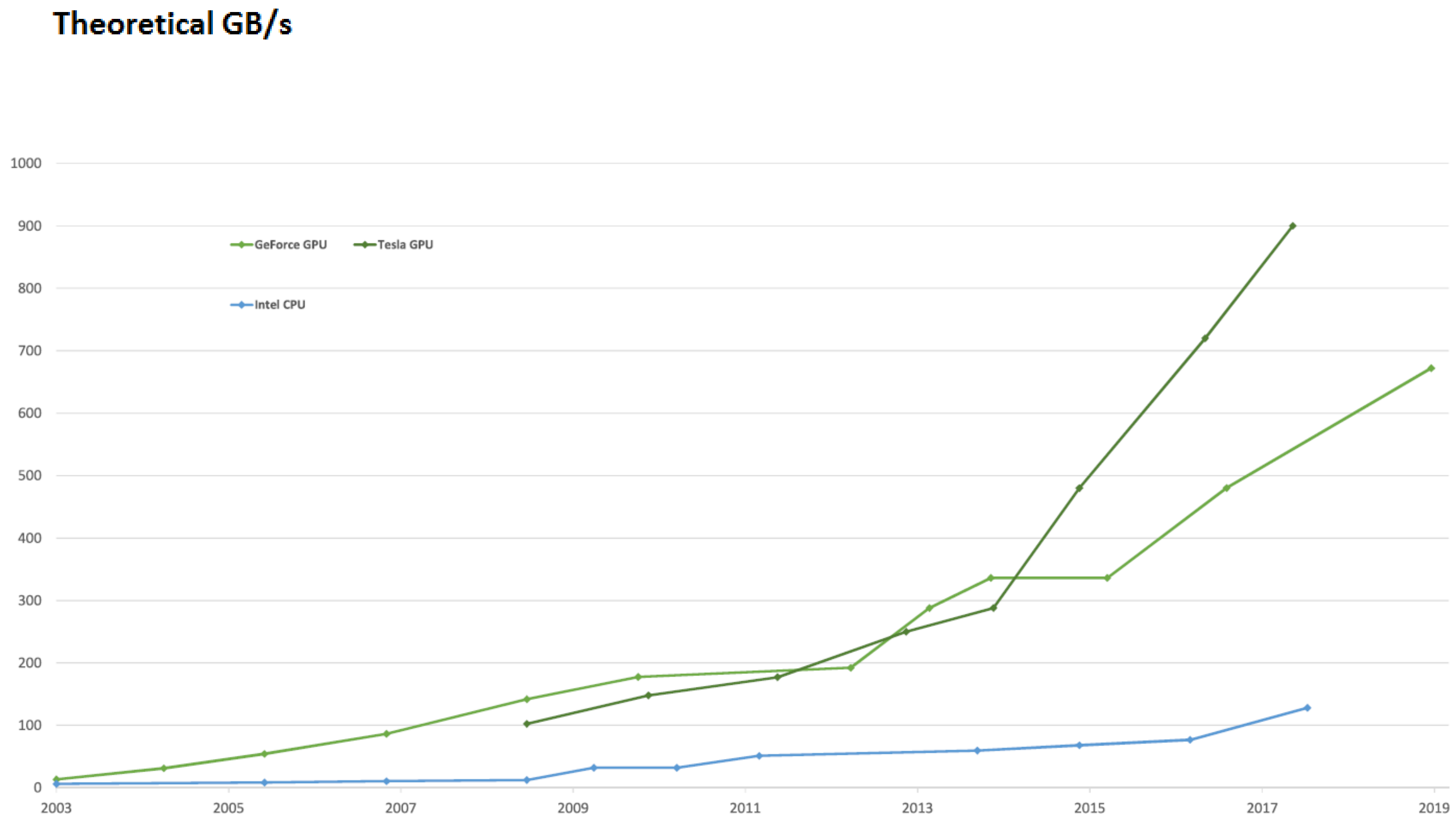
Acknowledgement: This lecture is based on “CUDA C++ Programming Guide”, v11.6, NVIDIA and older versions

CUDA

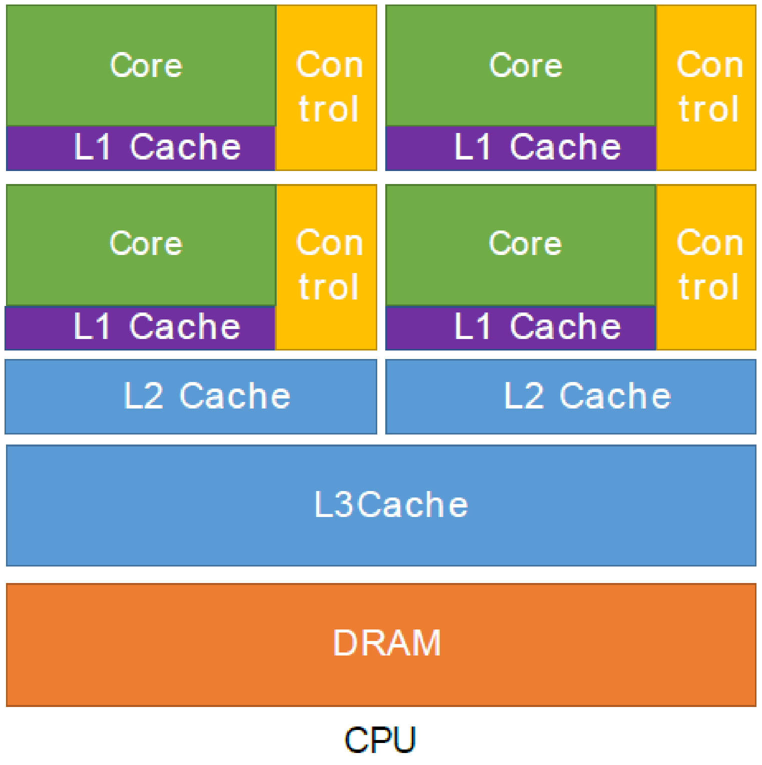
Theoretical GFLOP/s

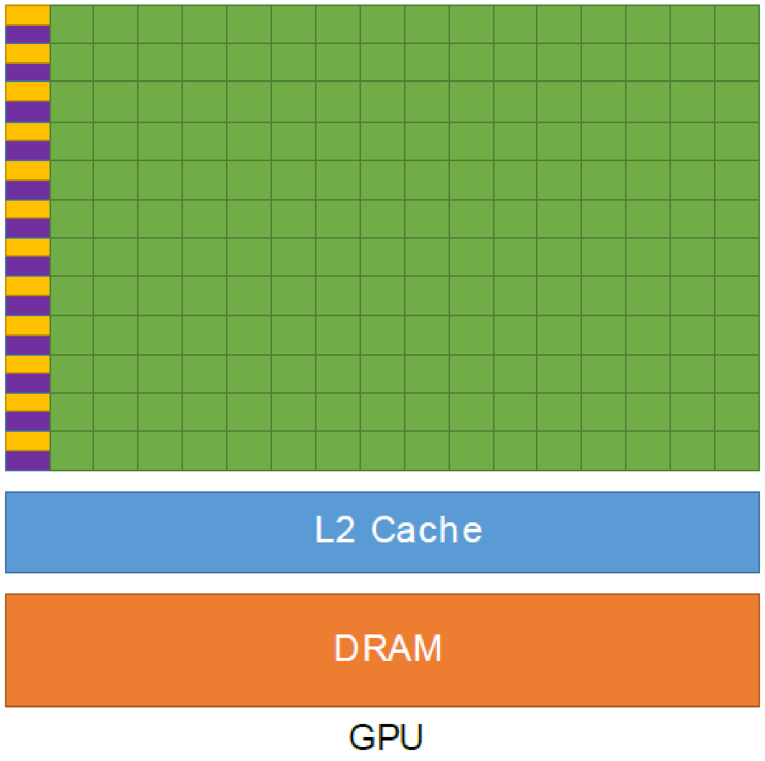


CUDA

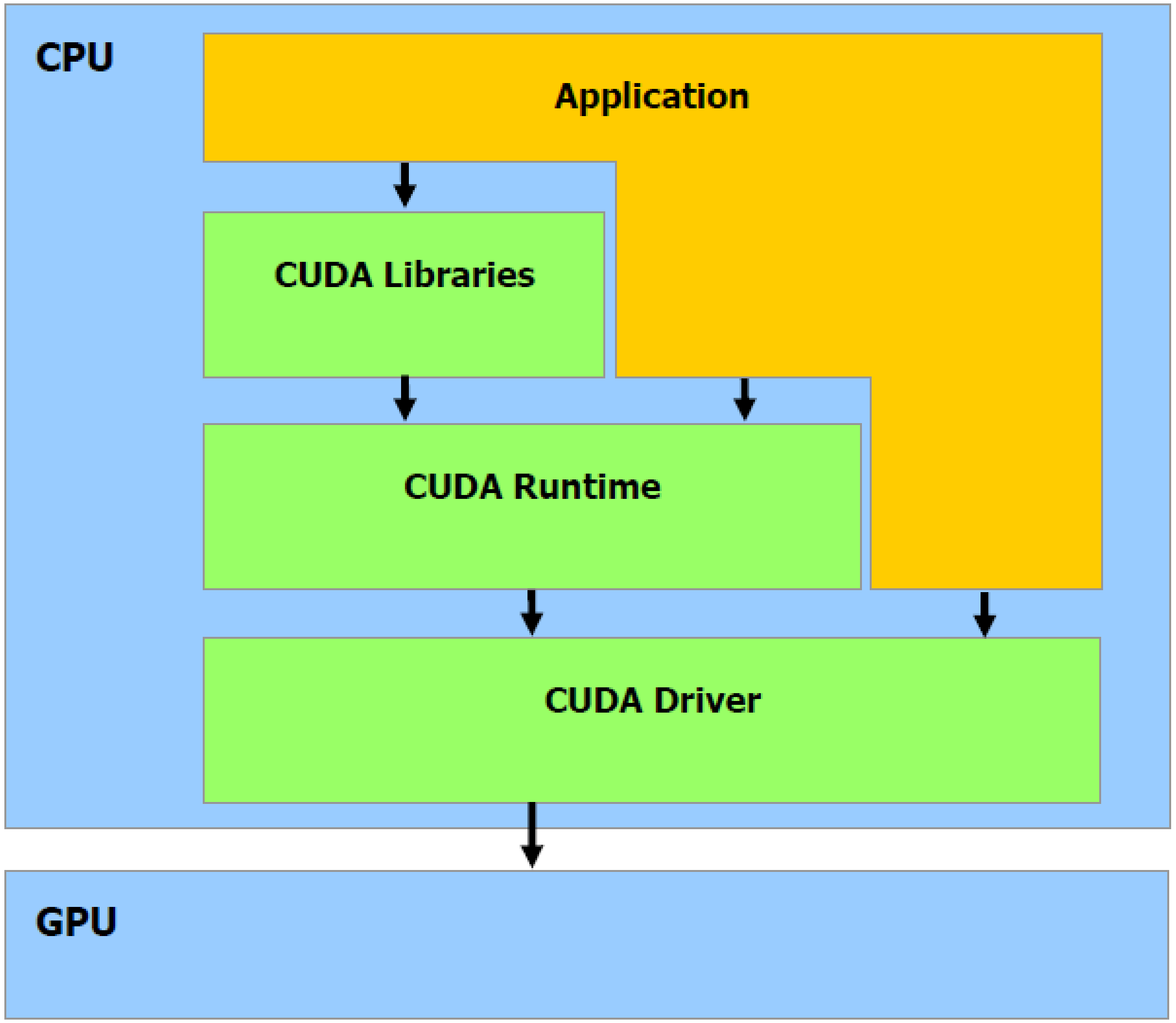

Theoretical GB/s
GeForce GPU
Tesla GPU
Intel CPU
1000
900
800
700
600
500
400
300
200
100
0
2003
2005
2007
2009
2011
2013
2015
2017
2019

CUDA – CPU and GPU Architecure

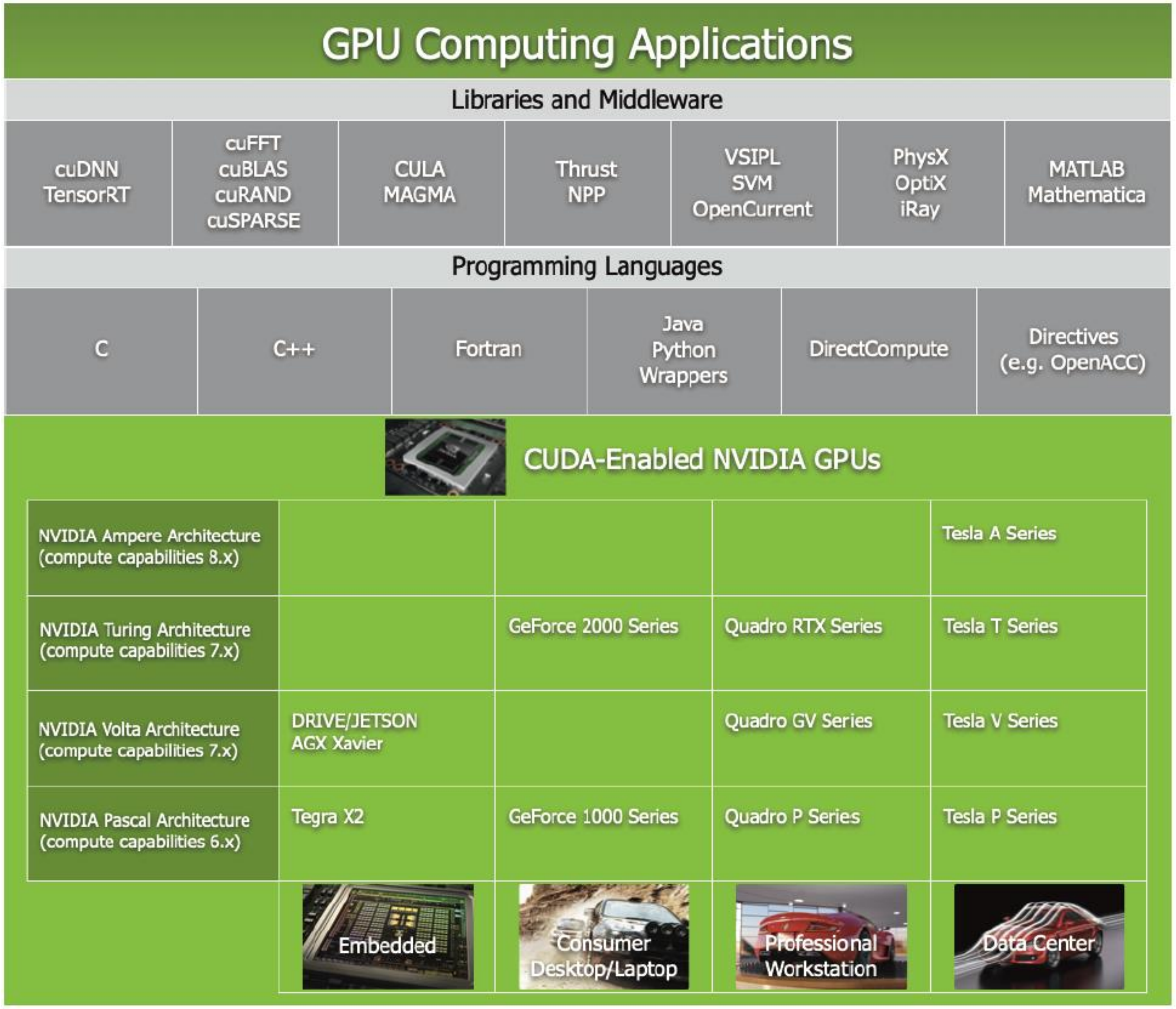




CUDA Architecture



CUDA Computing Applications



GPU Computing Applications

Libraries and Middleware

cuDNN TensorRT	cuFFT cuBLAS cuRAND cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL SVM OpenCurrent	PhysX OptiX iRay	MATLAB Mathematica

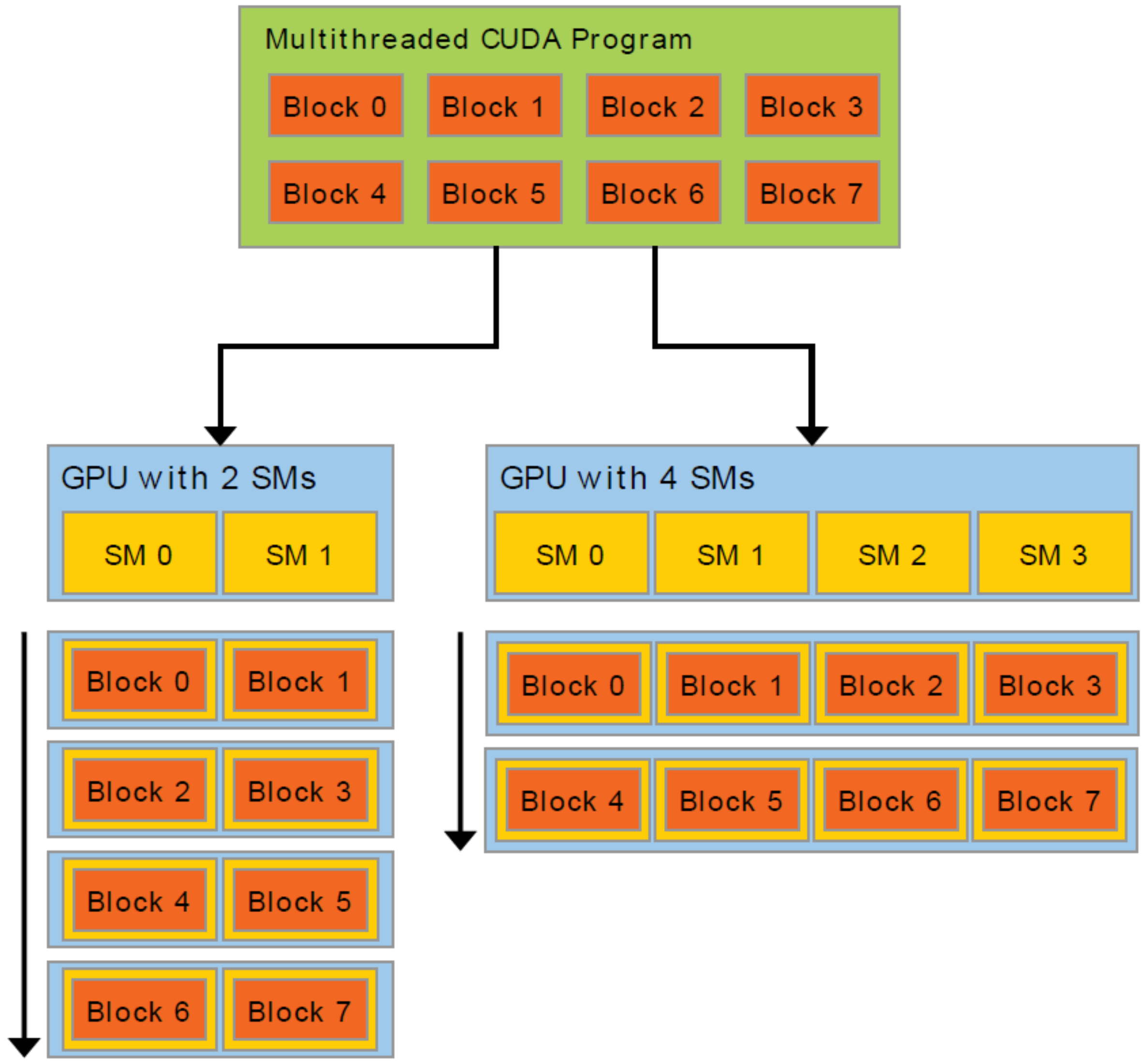
Programming Languages

C	C++	Fortran	Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)

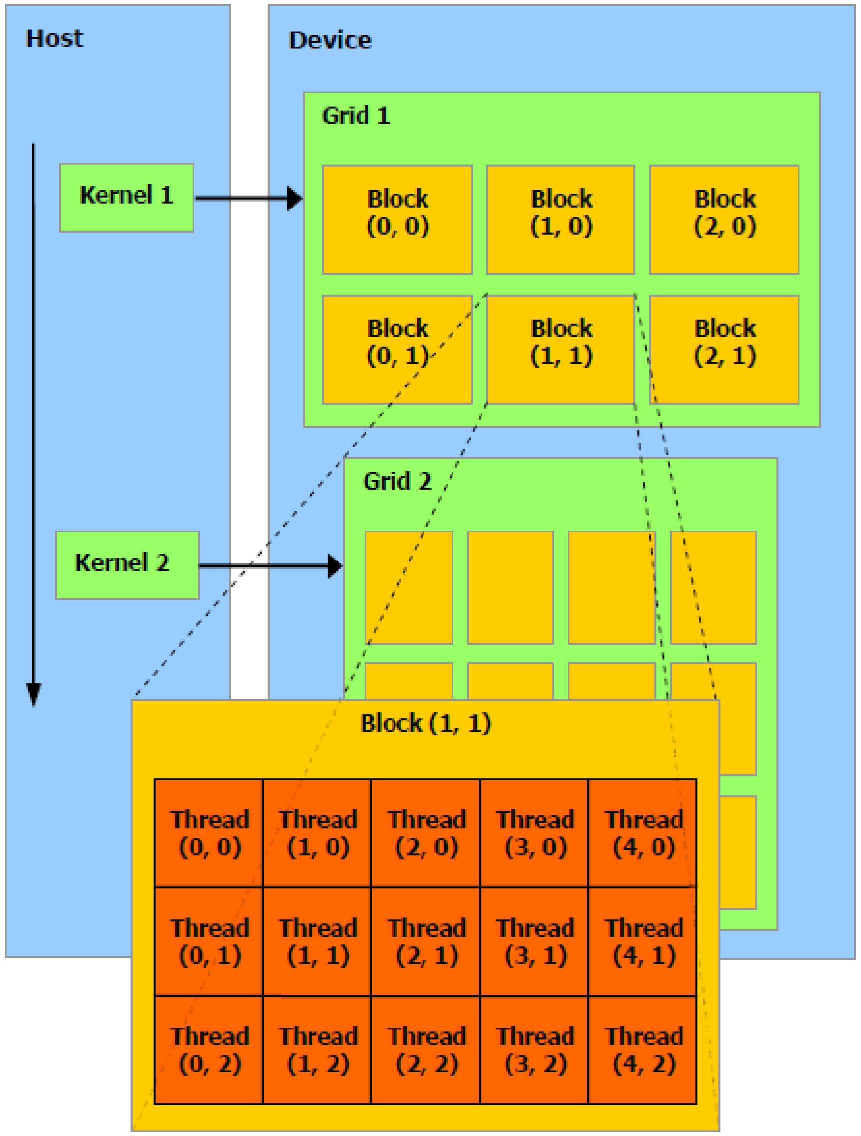
CUDA-Enabled NVIDIA GPUs

NVIDIA Ampere Architecture (compute capabilities 8.x)				Tesla A Series
NVIDIA Turing Architecture (compute capabilities 7.x)		GeForce 2000 Series	Quadro RTX Series	Tesla T Series
NVIDIA Volta Architecture (compute capabilities 7.x)	DRIVE/JETSON AGX Xavier		Quadro GV Series	Tesla V Series
NVIDIA Pascal Architecture (compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series	Tesla P Series
	Embedded	Consumer Desktop/Laptop	Professional Workstation	Data Center

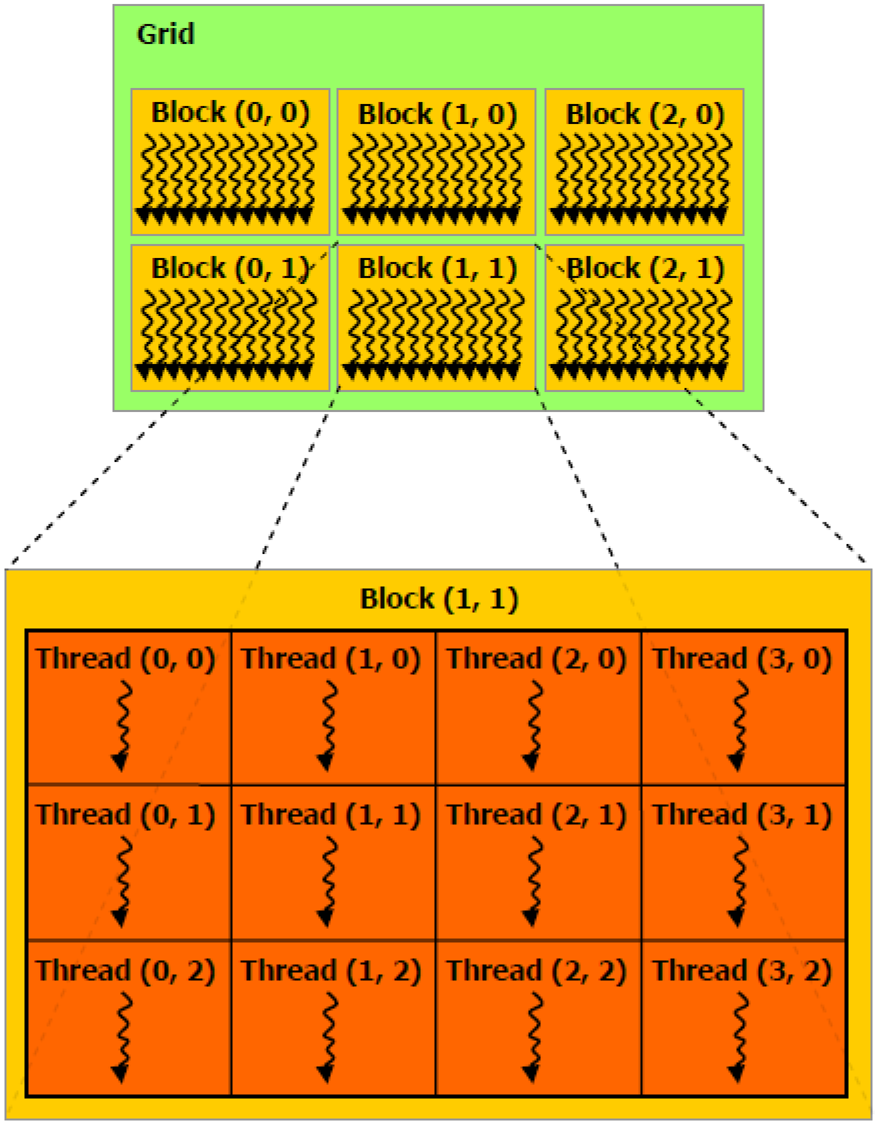
CUDA Automatic Scalability



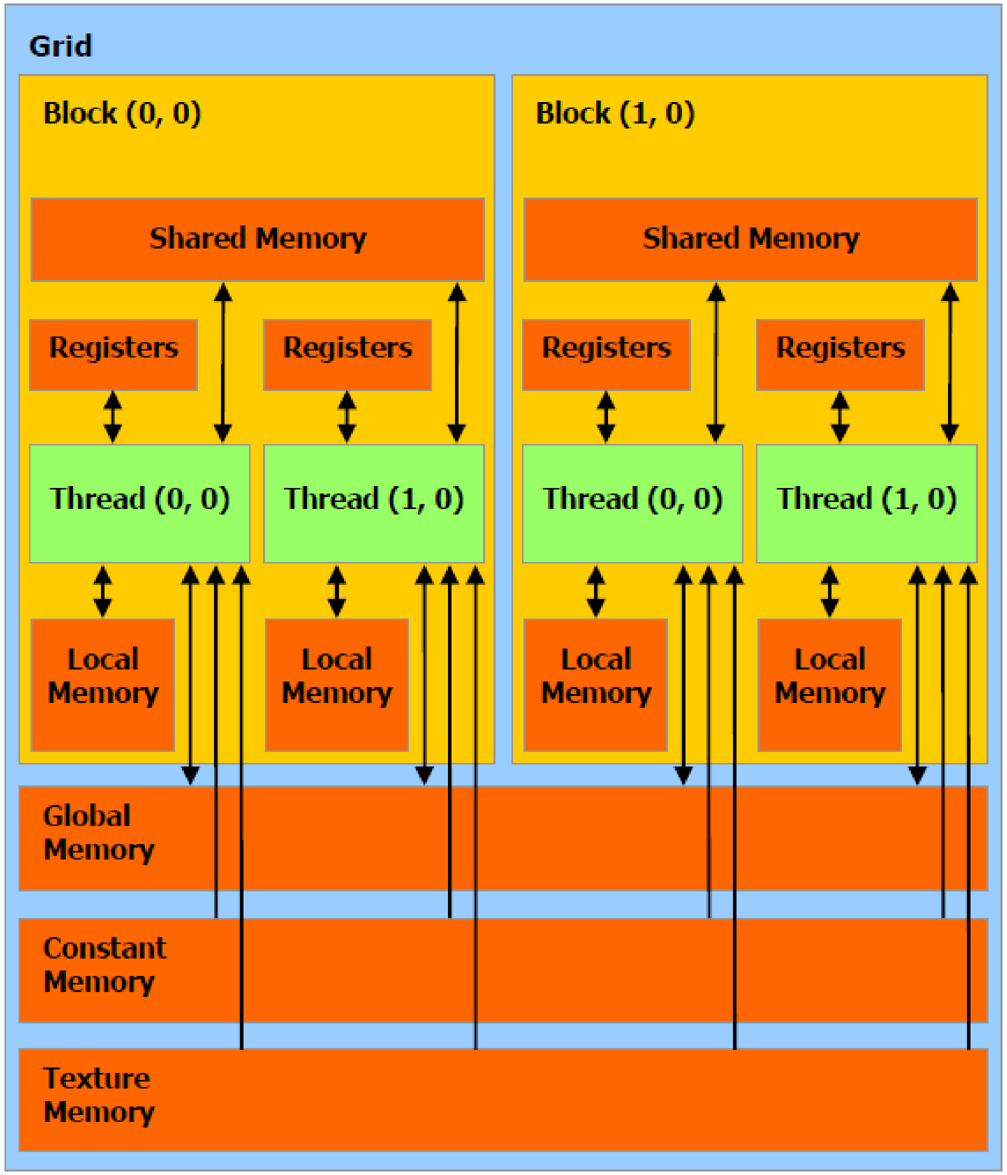
CUDA Kernel, Grid, Block, Thread



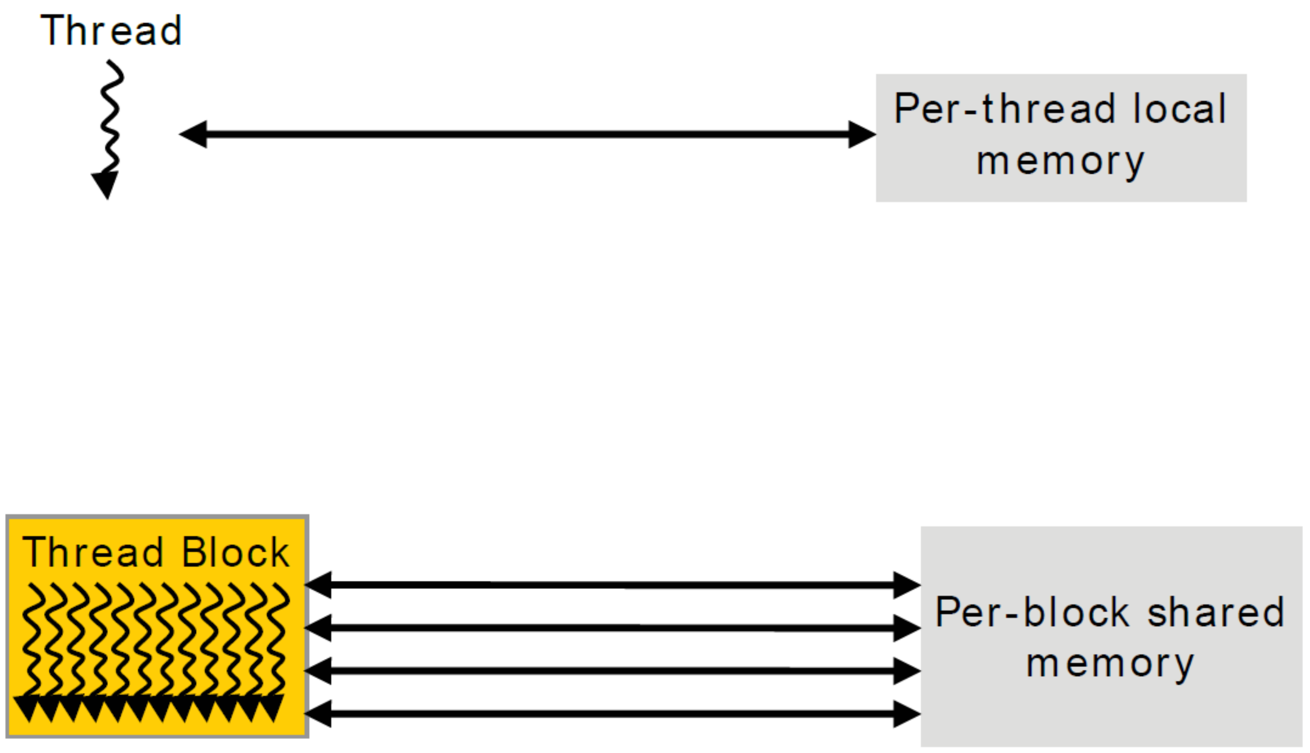
CUDA Grid of Threads



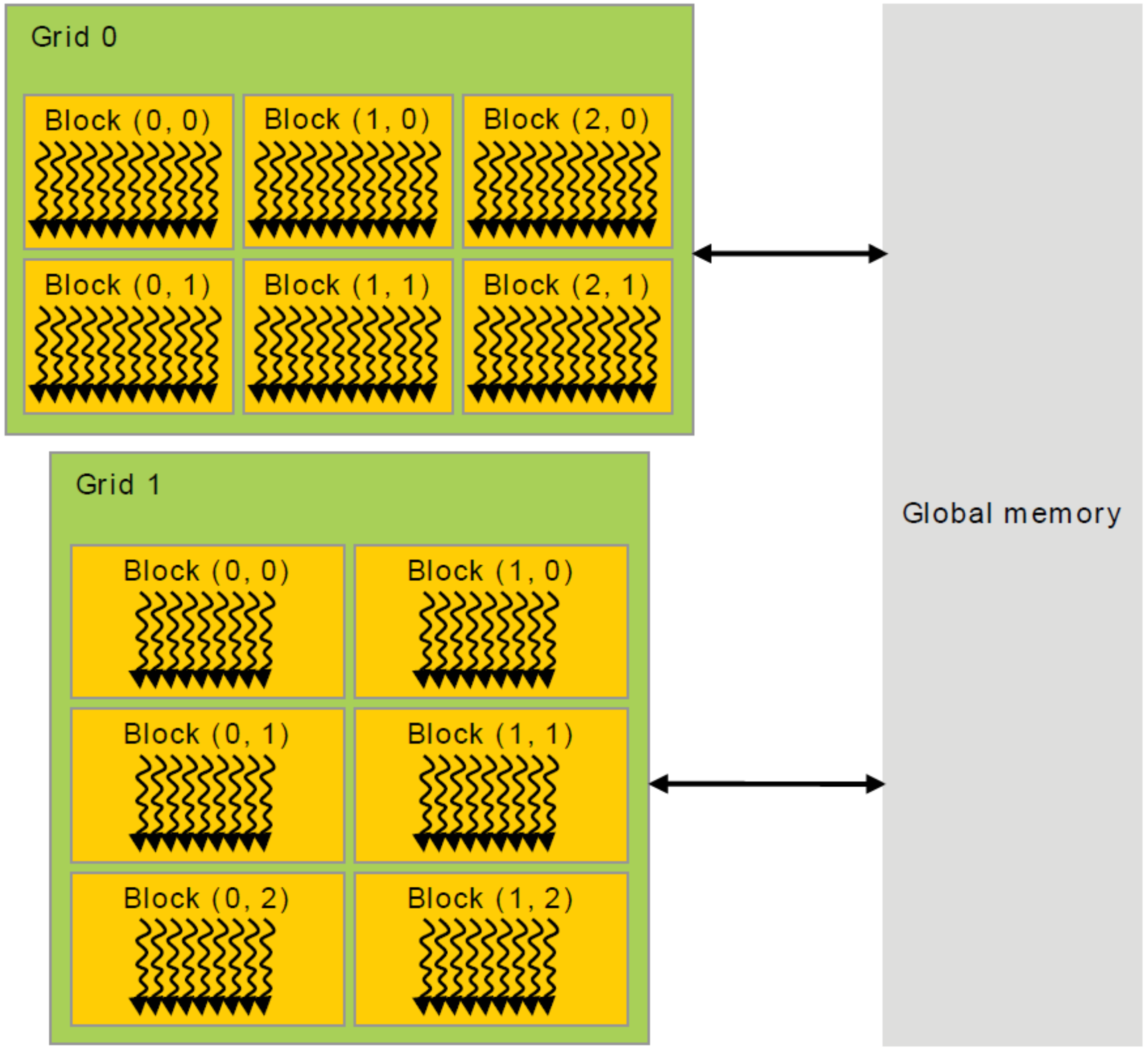
CUDA memory


Grid
Block (0, 0)
Block (1, 0)
Shared Memory
Shared Memory
Registers
Registers
Registers
Registers
Thread (0, 0)
Thread (1, 0)
Thread (0, 0)
Thread (1, 0)
Local Memory
Local Memory
Local Memory
Local Memory
Global Memory
Constant Memory
Texture Memory

CUDA memory


Thread
Per-thread local memory
Thread Block
Per-block shared memory

CUDA memory


Grid 0
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Grid 1
Block (0, 0)
Block (1, 0)
Block (0, 1)
Block (1, 1)
Block (0, 2)
Block (1, 2)
Global memory

CUDA Compute Capability

Compute Capability 1.x

- Global memory (read and write)
  - Slow and uncached
  - Requires sequential and aligned 16 byte reads and writes to be fast
- Texture memory (read only)
  - Cache optimized for 2D spatial access pattern
- Constant memory
  - This is where constants and kernel arguments are stored
  - Slow, but with small cache
- Shared memory (16 kb per MP)
  - Fast, but take care of bank conflicts
  - Can be used to exchange data between threads in a block
- Local memory (used for data that does not fit into registers)
  - Slow and uncached
- Registers
  - Fastest, scope is thread local

CUDA Compute Capability

Compute Capability 2.x

- Global memory (read and write)
  - Slow, but now with cache
- Texture memory (read only)
  - Cache optimized for 2D spatial access pattern
- Constant memory
  - Slow, but with cache (8 kb)
- Shared memory
  - Fast, but slightly different rules for bank conflicts now
- Local memory
  - Slow, but now with cache
- Registers (32768 32-bit registers per MP)

CUDA Compute Capability

Compute Capability 3.x

- Global memory (read and write)
  - Generally cached in L2, but not in L1
- Constant memory
  - Cache shared by all functional units
- Shared memory
  - 32 banks with two addressing modes
- Local memory
  - Cached in L1

Compute Capability 5.x

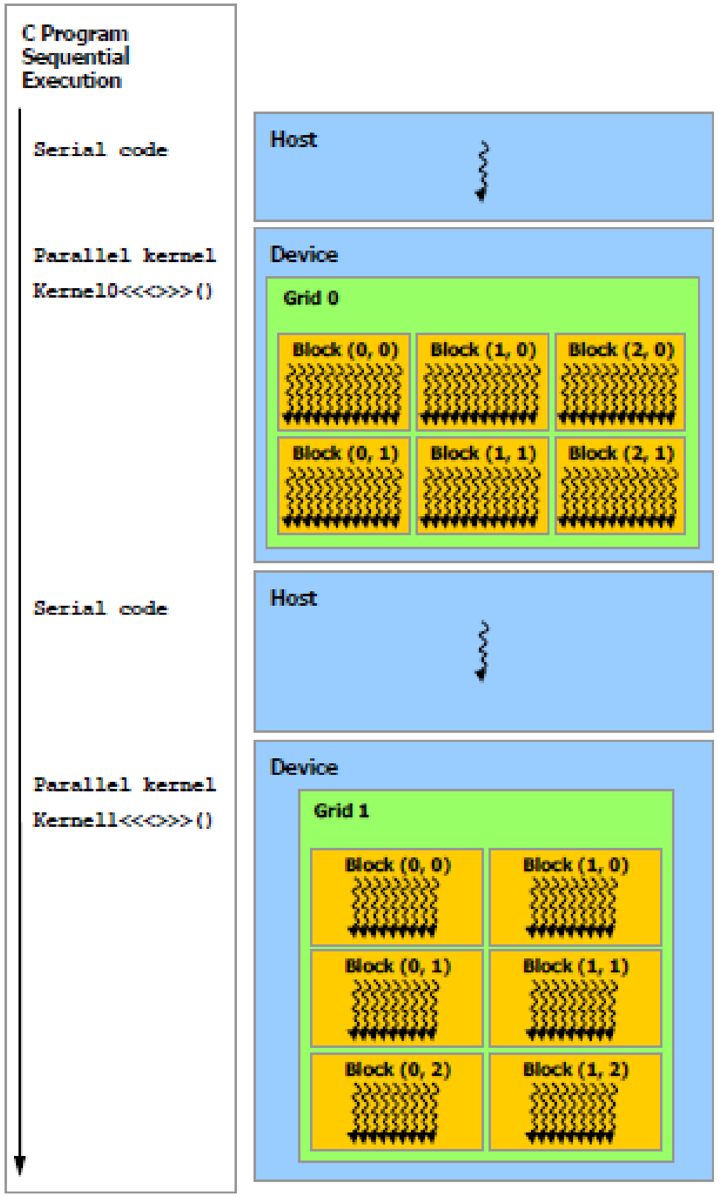
- Global memory
  - Identical to cp 3.x
- Shared memory
  - 32 banks organized such that successive 32-bit words map to successive banks

Compute Capability 6.x

Compute Capability 7.x

Compute Capability 8.x

CUDA Heterogeneous Programming



CUDA

- Kernel invocation

  `kernel<<<` Dg, Db, Ns, S >>>

  - **Dg** is the grid dimension (dim3 type)
  - **Db** is the block dimension (dim3 type)
  - **Ns** is the number of shared memory bytes
  - **S** is the stream

CUDA

- Block
  - `threadIdx.x`, `threadIdx.y`, `threadIdx.z`
    - Identify the thread in the block
  - `blockDim.x`, `blockDim.y`, `blockDim.z`
  - `threadID = x+y*Dx+z*Dx*Dy`
  - Threads are executed in *warps*
    - 32 threads of the same block with consecutive ids
    - Blocks should have more than 32 threads

CUDA

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
}
```

CUDA

- Grid
  - `blockIdx.x`, `blockIdx.y`, `blockIdx.z`
    - Identify the block in the grid
  - `gridDim.x`, `gridDim.y`, `gridDim.z`

CUDA

- **`__global__`**
  - Used to define kernel functions
  - Called by CPU; Run in GPU
- **`__device__`**
  - Used to define device functions
  - Called by GPU; Run in GPU
- **`__shared__`**
  - Used to declare variables that reside in shared memory

CUDA main()

```
int  main(void)
{
    float A[SIZE], B[SIZE], C[SIZE];
    float *devPtrA; float *devPtrB; float *devPtrC;
    int memsize = SIZE * sizeof(float);

    // Initialize arrays
    srand (time(NULL));
    for(int i=0; i < SIZE; i++) {
        A[i]=rand() % 100; B[i]=rand() % 100;
    }

    cudaSetDevice(0);     // Select GPU device (can be 0 to 1)

    // Allocate device memory for A, B and C arrays
    cudaMalloc((void**)&devPtrA, memsize);
    cudaMalloc((void**)&devPtrB, memsize);
    cudaMalloc((void**)&devPtrC, memsize);

    // Copy data (data to process) from host to device (from CPU to GPU)
    cudaMemcpy(devPtrA, A, memsize,  cudaMemcpyHostToDevice);
    cudaMemcpy(devPtrB, B, memsize,  cudaMemcpyHostToDevice);

    // Execute the Kernel
    arrAdd <<<1, SIZE>>> (devPtrA,  devPtrB, devPtrC); // launch 1 block with SIZE threads

    // Copy data from device (results) back to host
    cudaMemcpy(C, devPtrC, memsize,  cudaMemcpyDeviceToHost);

    // Show results
    printf("     A       B        C\n");
    for (int i=0; i < SIZE; i++) {
        printf("%2d: %4.1f + %4.1f = %5.1f\n", i, A[i], B[i], C[i]);
    }

    // Free device memory
    cudaFree(devPtrA); cudaFree(devPtrB); cudaFree(devPtrC);
}
```

CUDA kernel

```
//  Kernel definition, see also section 2.1 of NVIDIA CUDA Programming Guide
__global__  void arrAdd(float *A, float *B, float *C)
{
    // threadIdx.x is a built-in variable provided by CUDA at runtime
    // It represents the thread index inside the block

    int id = threadIdx.x; // id: unique thread identifier

    C[id] = A[id] + B[id];
}
```