Programming paradigms for GPU devices

PATC Course @CINECA
15-17 April 2020

Sergio Orlandini
s.orlandini@cineca.it

Luca Ferraro
l.ferraro@cineca.it

Introduction to General Purpose GPU Computing

Agenda:

- what's a GPU
- how GPU works
- CPU vs GPU comparison
- GPGPU Computing
- the GPGPU model
- GPU programming paradigms

What is a GPU

- **G**raphics **P**rocessing **U**nit

  a device equipped with an highly parallel microprocessor (*many-core*) and a private memory with very high bandwidth

- born in response to the growing demand for high definition 3D rendering graphic applications

GPU are specialized for parallel intensive computation

- GPUs are designed to render complex 3D scenes composed of ***milions of data*** points/vertex in a very fast frame rate (60-120 FPS)
- the rendering process requires a set of transformations based on linear algebra operations and (mostly local) filters
- the ***same set* of operations** are applied on each point/vertex of the scene
- each operation is ***independent*** with respect to data
- all **operations are performed *in parallel*** using a **huge number of *threads*** which process all data independently

Parallelism of Single Program Multiple Data (SPMD)

```
// typical loop over each point with the same set of operation
for each point in collection_of_points:
  output_data = trasformations_on_point(point, input_data)
```

- If the set of tranformations can be applied independently on each point, the output result is independent on the order of point computation
- If transformations are independent, we can speed up the elaboration:
  - apply the same transformation (Single Program) ...
  - ... to each point (Multiple Data)
  - ... using multiple threads concurrently

What is a Thread

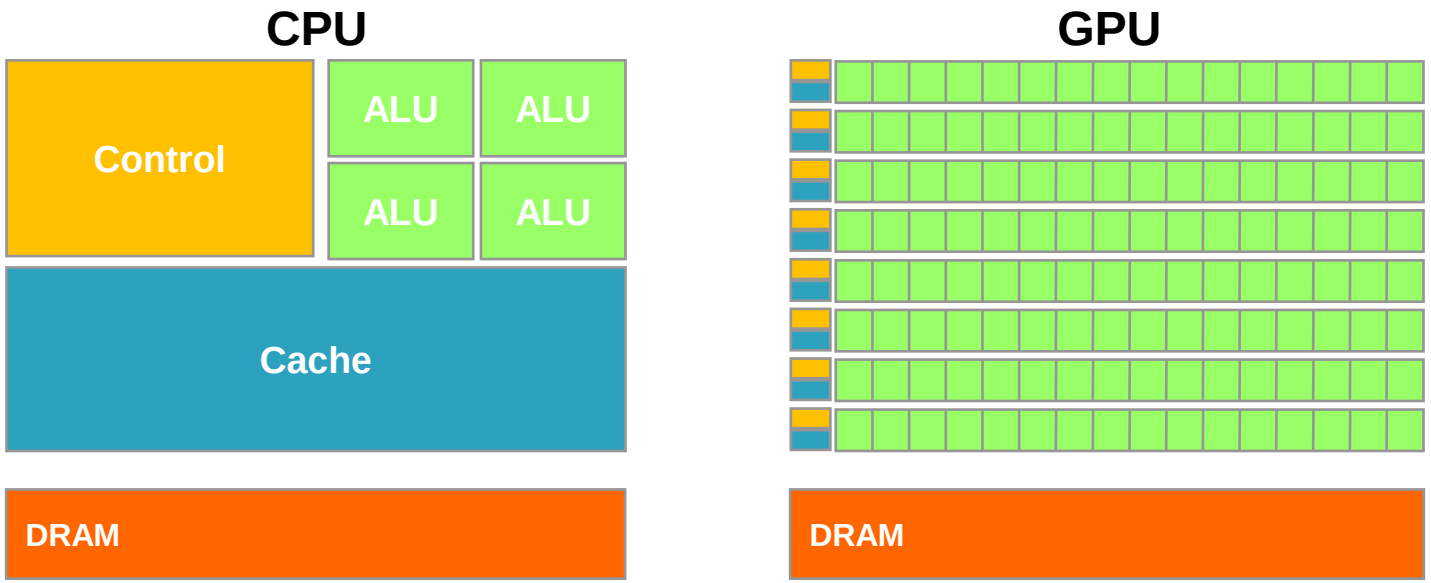
- A *thread* is an independent flow of execution of a program which share the same memory resources of the main process (access to main data)
- *A process* can spawn multiple threads of execution, each can follow an independent flow
- Threads inside a process can:
  - *exchange data* because they share the same memory of the program
  - *use local memory* not shared with other threads
  - *synchronize* with other threads (waiting dependences)

Concurrency using CPU Threads

- Threads can run concurrently as long as there is available hardware to use for their execution (registers and ALUs)
- Modern CPUs often have multiple cores on chip:
  - each core has registers and ALU units to run a thread independently from other cores
- When there are more threads on the fly than available cores, the operating system can make a context switch
  - the running thread is freezed: all information about its status is saved and put back for later restore
  - a new thread take thehardware resources and start to do its flow until a new context switch will take place
- The context switch invole many backup operations, plus the fact that data is no longer in registers and caches

CPU vs GPU Architectures

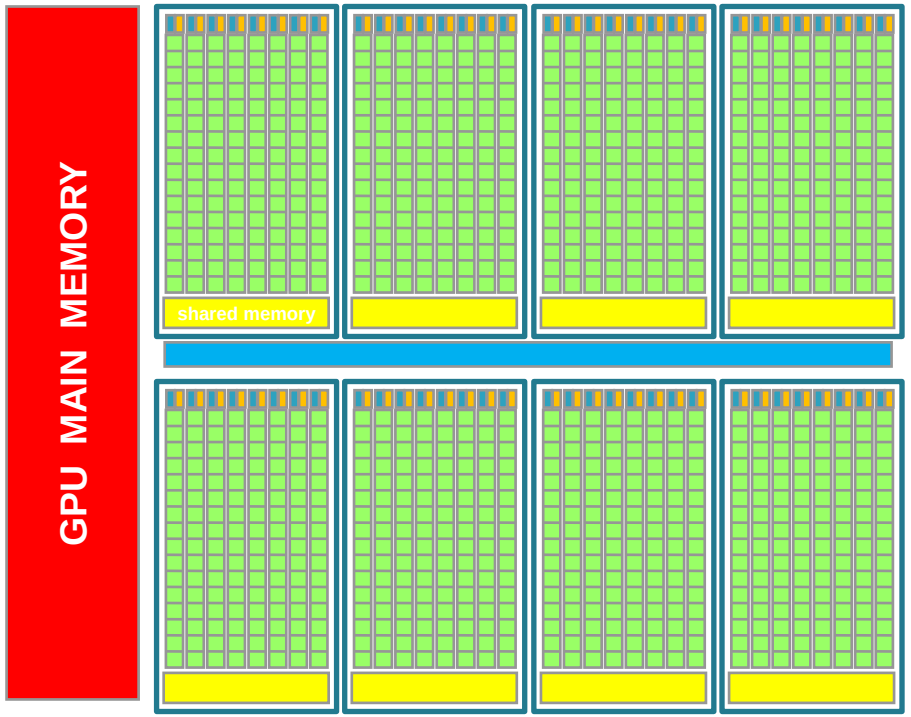
- GPU are specialized for *intense data-parallel computations* where the same set of operations are executed on different data
- GPU hardware is designed so that more transistors are devoted to data processing rather than data caching and flow control



GPU Architecture Scheme

A tipical GPU architecture consists of

- Main Global Memory
  - medium size (16 - 32 GB)
  - very hgh bandwidth (250-800 GB/s)
- Streaming Processors (SM)
  - grouping independent cores and control units
- each SM unit has
  - many ALU cores (> 100 cores)
  - lots of registers (32K-64K)
  - instruction scheduler dispatchers
  - a shared memory with very fast access to data



AMD HPC GPU Solutions

Model	FP32 [TFlops]	cores	RAM [GB]	Bandwidth [GB/s]	Link
Radeon MI8	8.2	4096	4 HBM	512	PCIe 3.0 x 16
Radeon MI25	12.3	4096	16 HBM2	484	PCIe 3.0 x 16
Radeon MI50	13.4	3840	16 HBM2	1024	PCIe 4.0 x 16
Radeon MI60	14.7	4096	32 HBM2	1024	PCIe 4.0 x 16

- VEGA Processor is the AMD top gamma GPU solution for HPC
  - the Radeon RX VEGA/500/400 series are for gaming
- HBM2: gen2 high-performance RAM interface for 3D-stacked DRAM with Error-correcting (ECC)
- Infinity Fabric™ Links per GPU deliver up to 200 GB/s of peer-to-peer bandwidth
- very high TDP factor sustainable on HPC server blades

NVIDIA HPC GPU Solutions

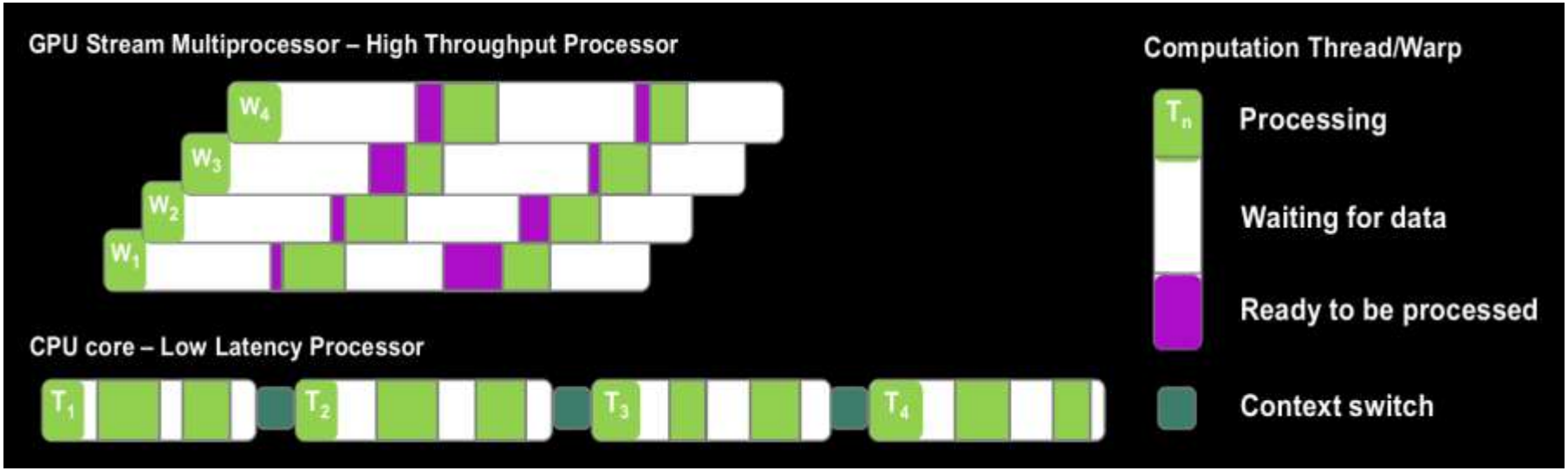
Model	FP32 [TFlops]	cores	RAM [GB]	Bandwidth [GB/s]	Link
Kepler K40	4.3	2280	12 GDDR5	240	PCIe 3.0 x 16
Pascal P100	10.6	3584	16 HBM2	732	PCIe 3.0 x 16
Volta V100	15.7	5120	32 HBM2	900	PCIe 3.0 x 16

- Tesla serie is the NVIDIA top gamma GPU solution for HPC
  - the GeForce series are for gaming
- HBM2: gen2 high-performance RAM interface for 3D-stacked DRAM with Error-correcting (ECC)

GPU Threads vs CPU Threads

GPU are able to manage thousands of threads efficiently

- GPU threads are extremely *light weighted*
  - no penalty in case of a *context-switch*
  - each thread has its own registers
- the more threads are *in flight*, the more the GPU hardware is able to hide memory or computational latencies



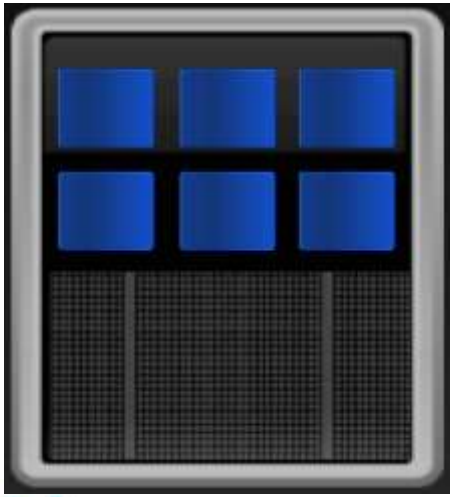
GPGPU (General Purpose GPU) and GPU computing

- Many scientific or tecnical applications that process large data sets can use data-parallel programming models to speed up the computations
- many algorithms outside the field of image rendering are accelerated by data-parallel processing
- ... so why not using GPU power for applications out of the graphic domain?
- many attemps where made by brave programmers and researchers in order to force GPU APIs to treat their scientific data (atoms, signals, events, etc) as pixel or vertex to be crunched by GPUs
- not many survived, yet the era of GPGPU computing was just begun ...

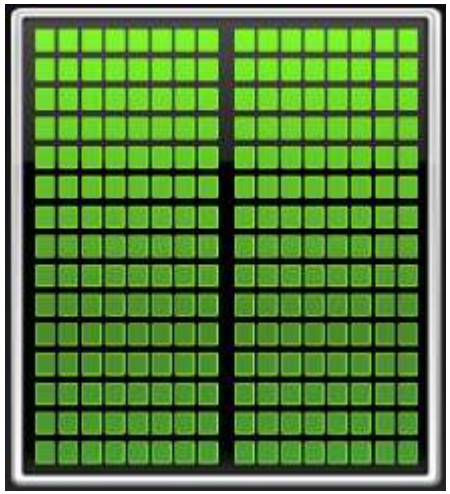
GPGPU Programming Model

- General Purpose GPU Programming relates to use GPU computational power to solve problems other than graphics
- CPU and GPU are **separate devices** with **separate memory** space addresses
- GPU is seen as an auxilirary coprocessor equiped with thousands of cores and a high bandwidth memory
- They should work togheter for best benefit and performances

CPU



GPU

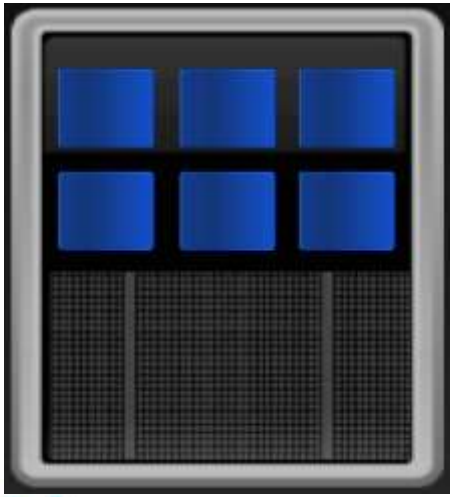


GPGPU Programming Model

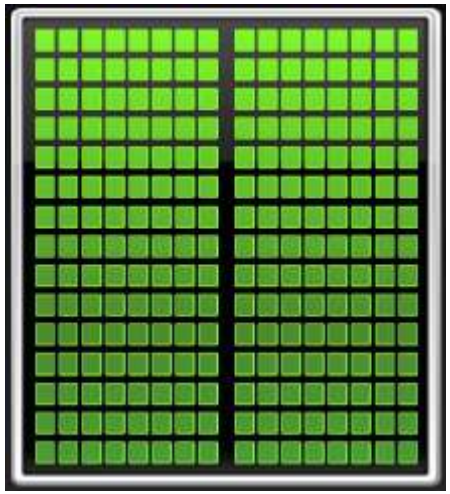
- Optimized for low-latency accesses to cached data sets
- Control logic for out-of-order and speculative execution
- Best for serial or event driven tasks

- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- Best for data-parallel tasks

CPU

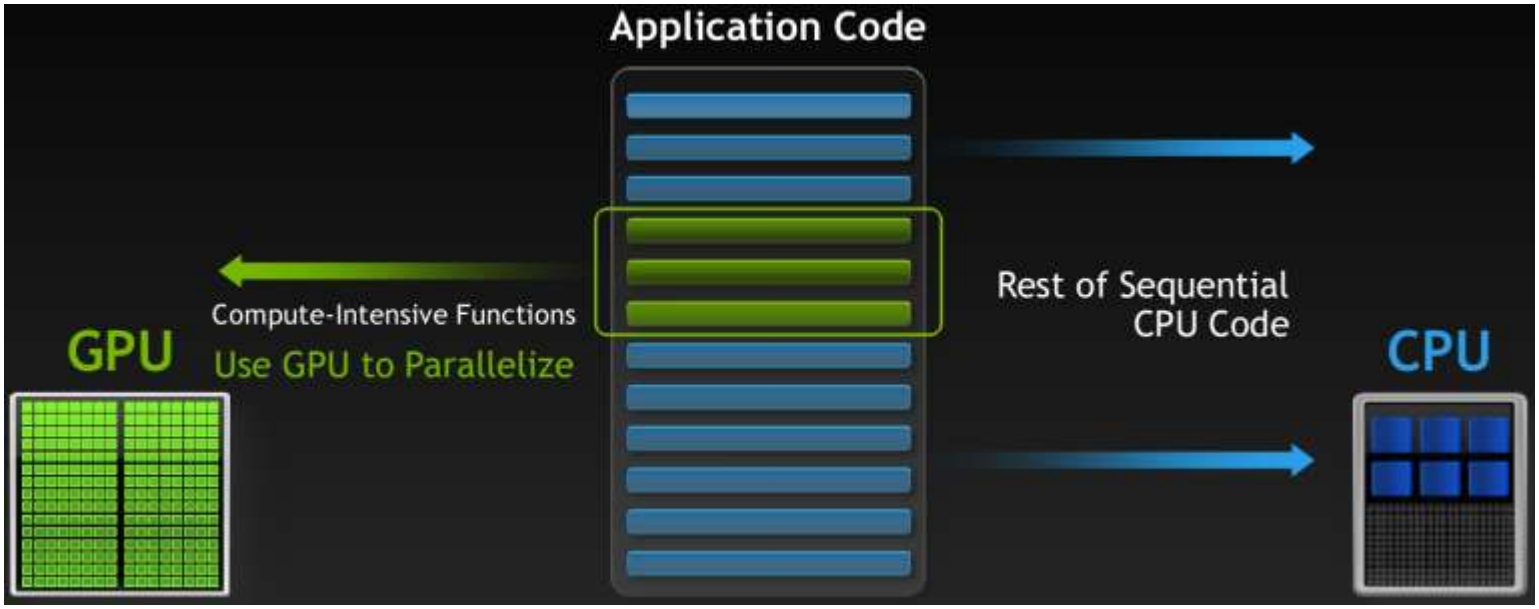


GPU



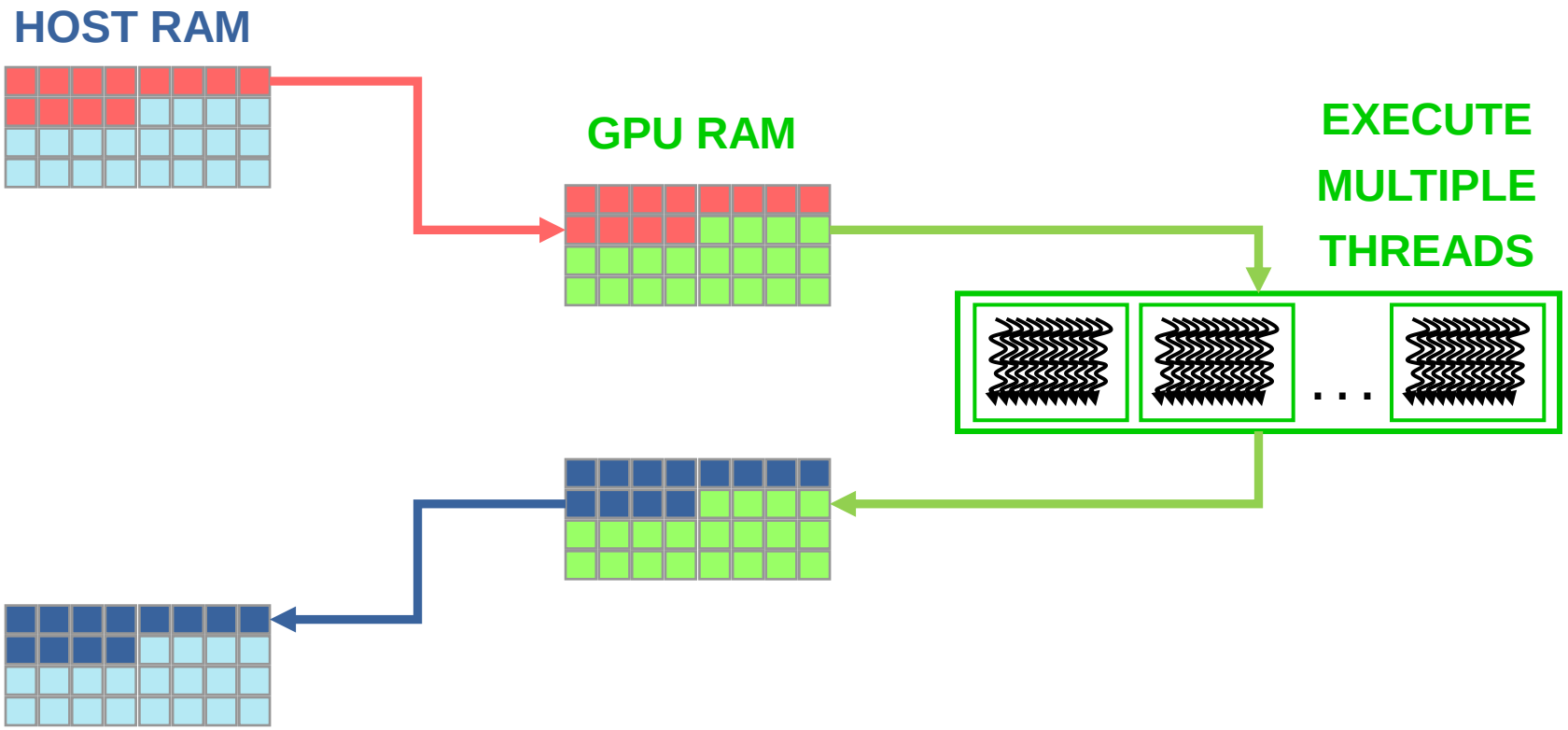
GPGPU Programming Model

- **serial parts** of a program, or those with low level of parallelism, keep running **on the CPU** (host)
- computational-intensive **data-parallel** regions are executed **on the GPU** (device)
- required data is moved on GPU memory and back to HOST memory



Data movement

- data must be moved from HOST to DEVICE memory in order to be processed by a CUDA kernel
- when data is processed, and no more needed on the GPU, it is transferred back to HOST



The data movement bottleneck

- Data movement is often *the bottleneck* of many GPU porting activities or applications
  - many unexperienced GPU developer don't keep the data transfer problem seriously enough or simply ignore it
    - some GPU paradigms/solution "hides" or automate transfers, but the driver or the compiler could make wrong choices
  - the bus transfer can be quite slow with respect to the GPU throughput capacity
    - PCIe v3 provide with a 12-14GB/s average transfer rate
  - sometimes data transfer can take more than the GPU computation if the problem is too "easy"
    - that's way we stressed that GPU are best suited for computational intensive problems, not just "parallel"
    - for example:
      - a vector add is not suited for GPU (order N)
      - matrix matrix multiplication is a good candidate for GPU (order N^3)

GPGPU Programming Model

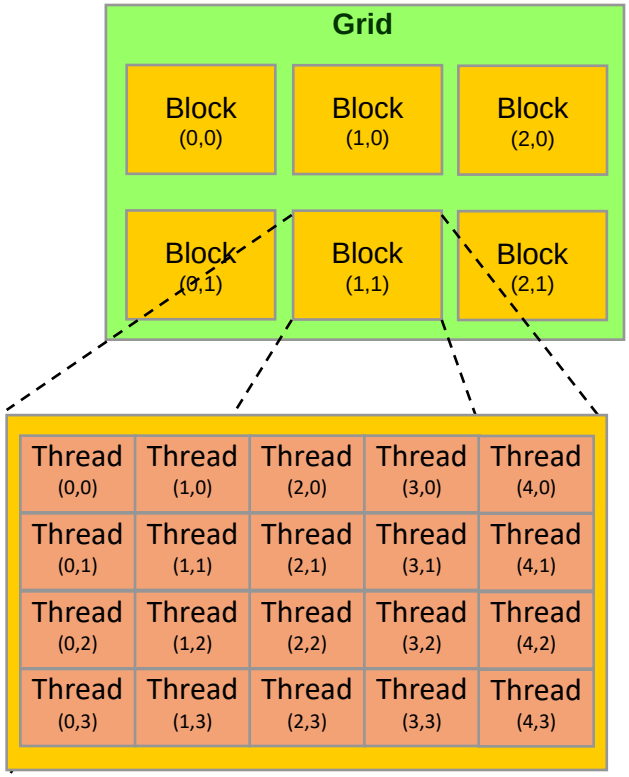
- A function which runs on a GPU is called "**kernel**"
  - when a kernel is launched on a GPU thousands of threads will execute its code
  - programmer chooses the number of threads to run
  - each thread act on a different data element independently
  - the GPU parallelism is very close to the SPMD paradigm

```
void vecAddCPU (int N, const float *A,
                const float *B, float *C)
{
   for ( int i = 0; i < N; i++ )
      c[i] = a[i] + b[i];
}
...
// call vecAddCPU on N elements
vecAddCPU ( N, a, b, c );
```

```
void vecAddGPU (int N, const float *A,
                const float *B, float *C)
{
   int i = .... // use unique thread index
    if ( i < N) c[i] = a[i] + b[i];
}
...
// launch kernel with 1 block of N threads
vecAddGPU<<<1, N>>>( N, a, b, c );
```

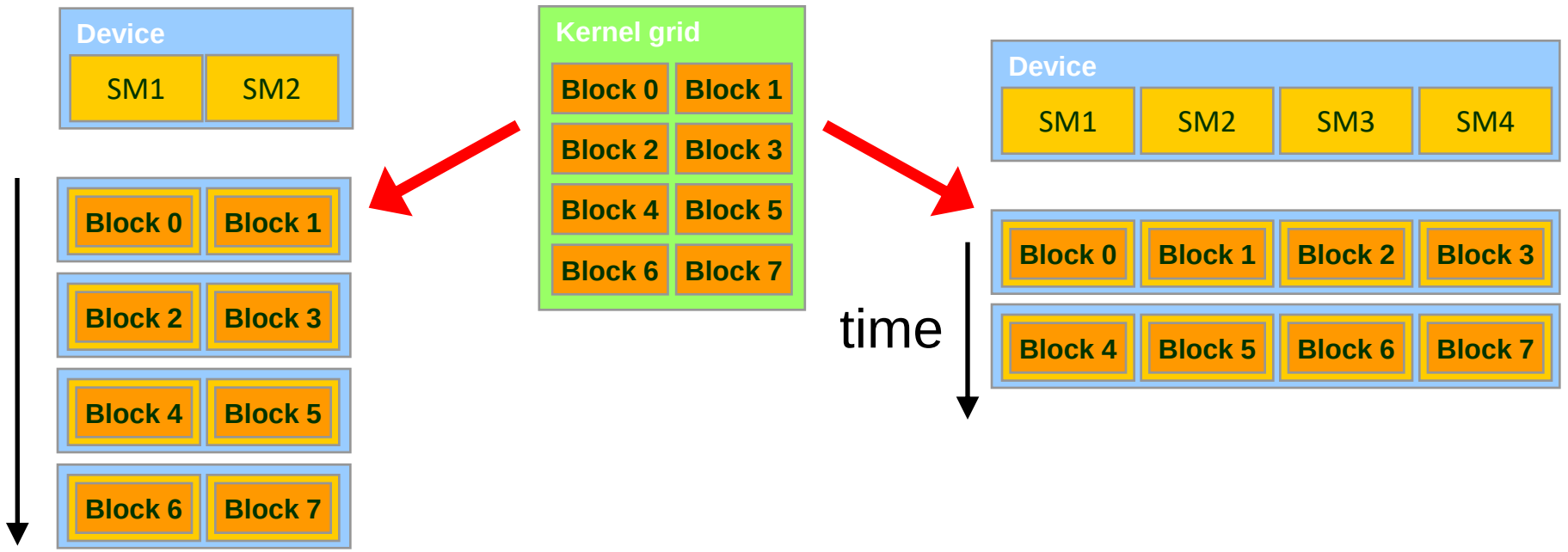
GPU Thread Hierarchy

- In order to compute N elements on the GPU in parallel, at least N concurrent threads must be created on the device
- GPU threads are grouped togheter in *teams* or *blocks* of threads
- Threads belonging to the same block or team can cooperate togheter exchanging data through a shared memory cache area

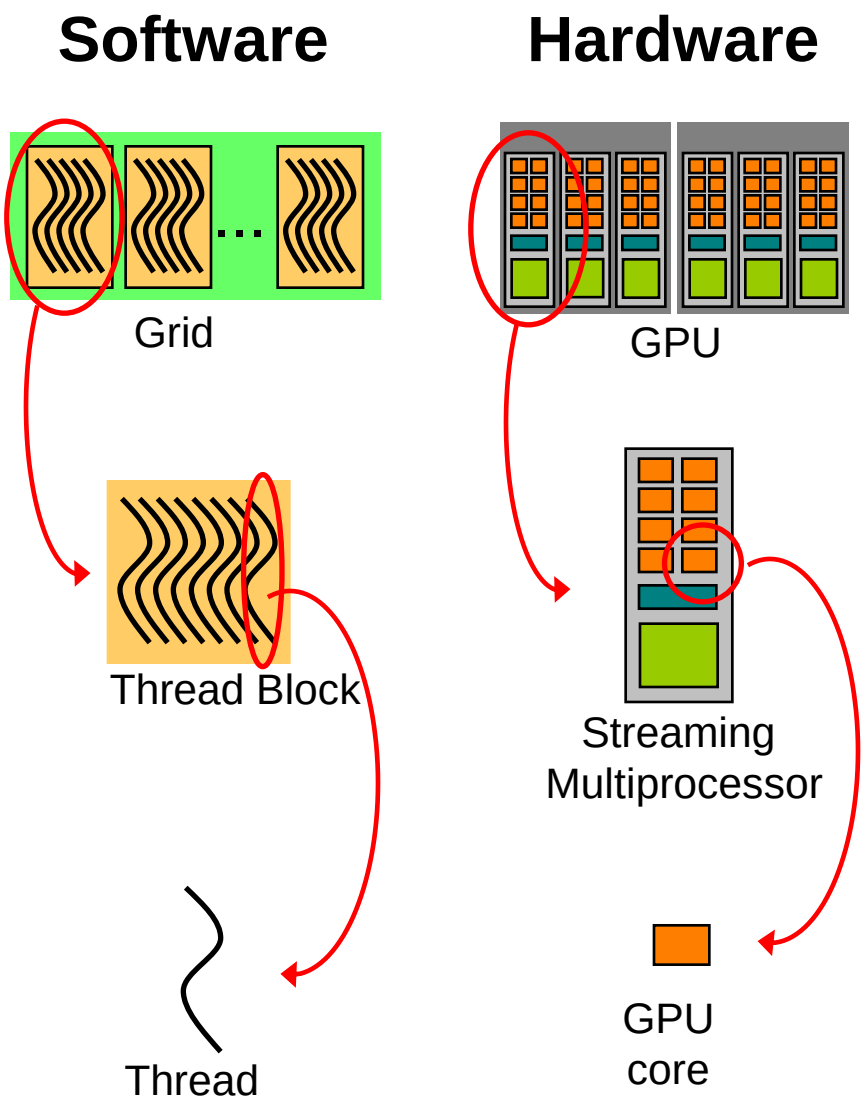


Trasparent Scalability

- the GPU runtime system can execute thread blocks in any order relative to each other
- This flexibility enables to execute the same application code on hardware with different numbers of SM



more on the GPU Execution Model

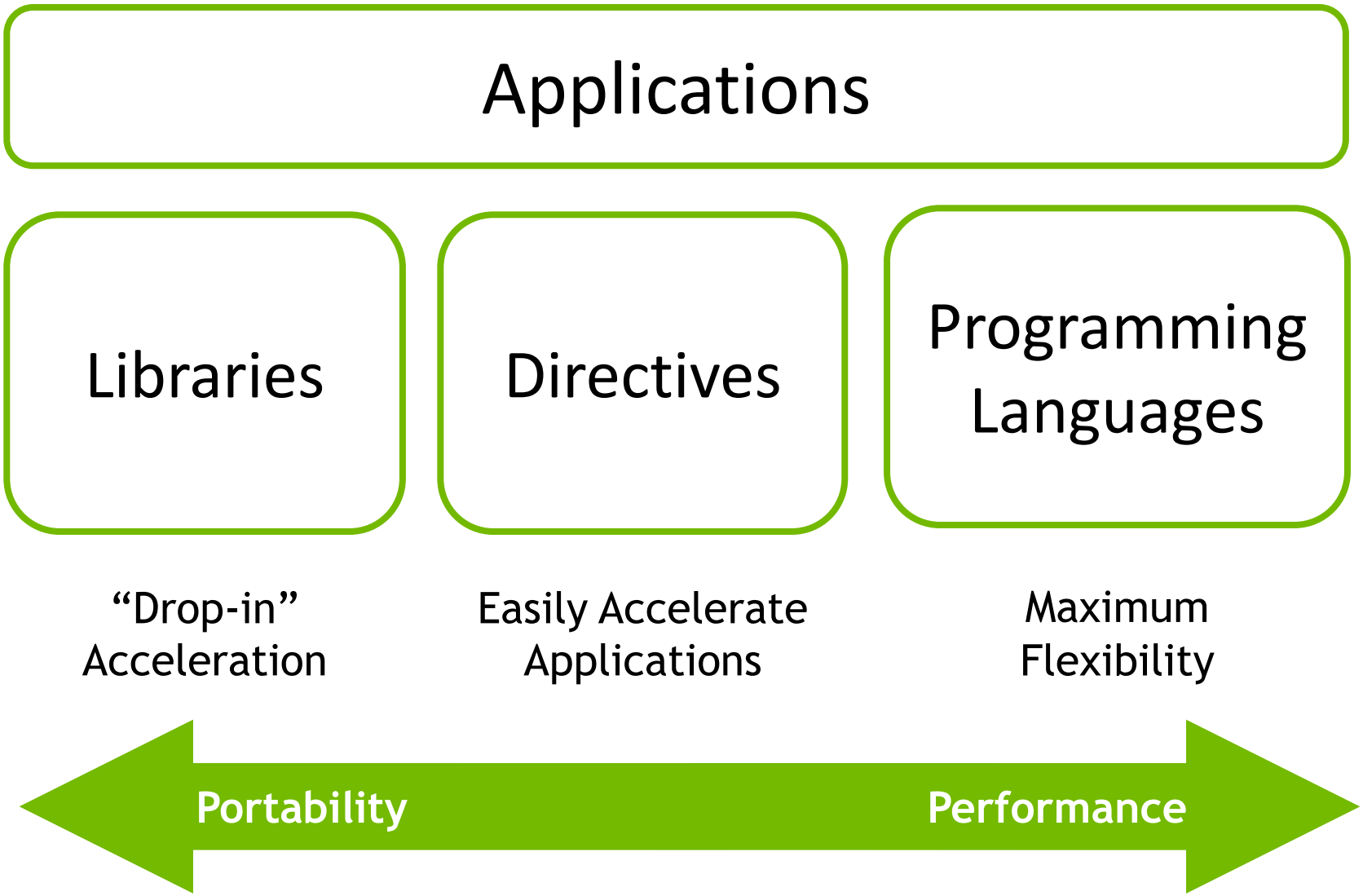


when a GPU kernel is invoked:

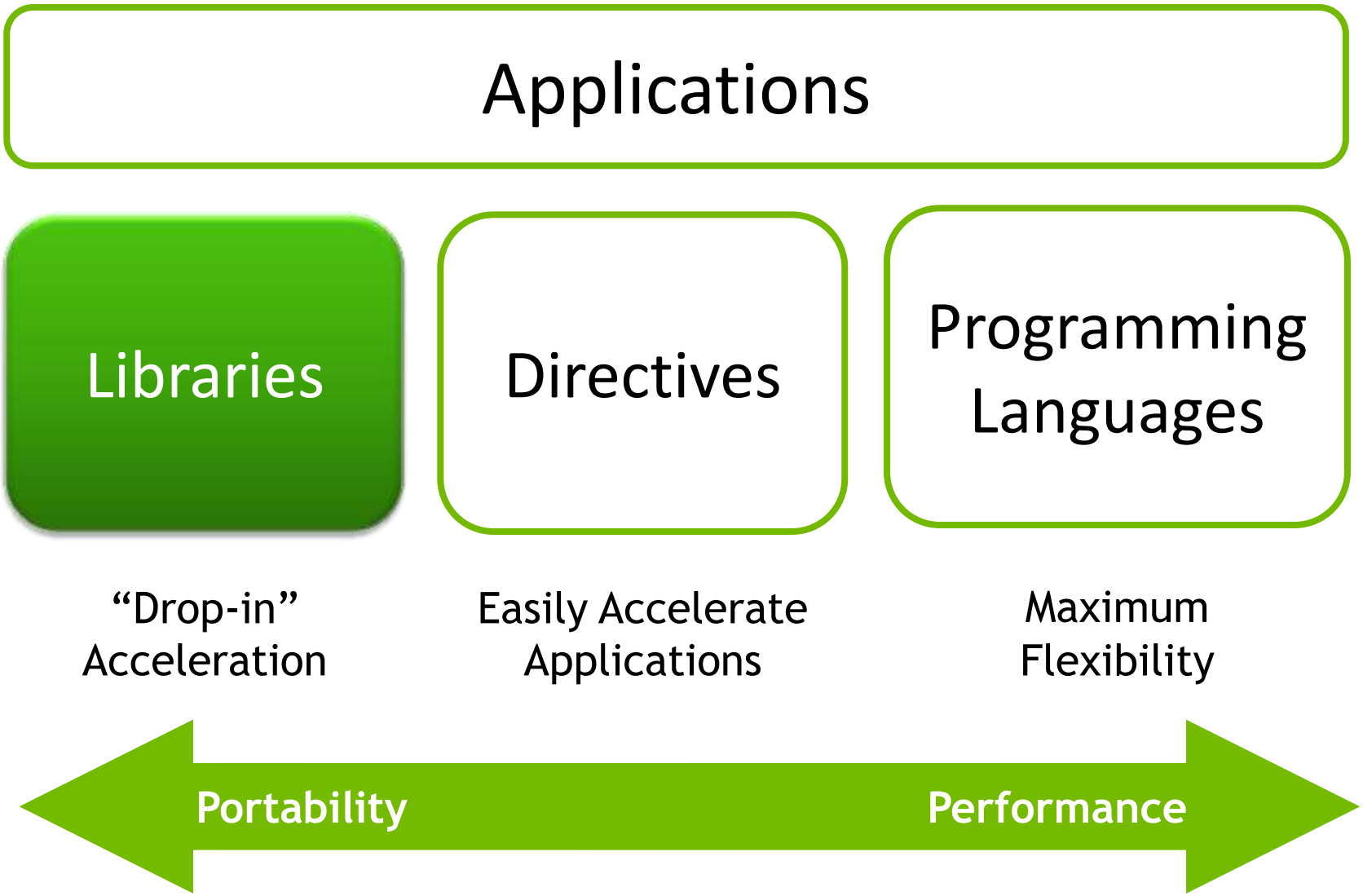
- each thread block is assigned to a SM in a round-robin mode
  - a maximum number of blocks can be assigned to each SM, depending on hardware generation and on how many resorces each requires (registers, shared memory, etc)
  - the runtime system maintains a list of active blocks and assigns new blocks to SMs as they complete
  - once a block is assigned to a SM, it remains on that SM until the work for all threads in the block is completed
  - each block execution is independent from the other (no synchronization is possible among them)
- threads of each block are partitioned into warps of consecutive *threads*
- the scheduler select for execution a warp from one of the residing blocks in each SM
- A warp execute one common set of instruction at a time
  - each GPU core take care of one thread in the warp
  - fully efficiency when all threads agree on their execution path

- GPU programming paradigms
  - GPU enabled libraries
  - High level directives
  - Low level programming languages

3 Ways to Accelerate Applications



3 Ways to Accelerate Applications

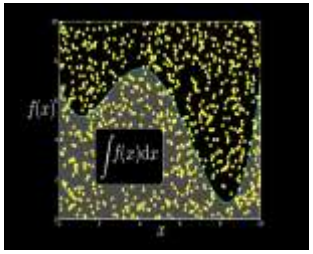


Libraries: Easy, High-Quality GPU Ready

- **Ease of use:** Using libraries enables GPU acceleration without in-depth knowledge of GPU programming
- **"Drop-in":** Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes
- **Quality:** Libraries offer high-quality implementations of functions encountered in a broad range of applications
- **Performance:** libraries are tuned by experts

Some GPU-accelerated Libraries

NVIDIA cuBLAS



NVIDIA cuRAND

NVIDIA cuSPARSE

NVIDIA NPP

GPU VSIPL

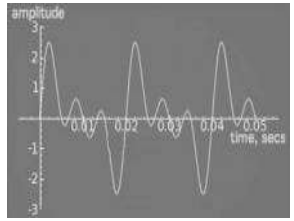
Vector Signal Image Processing

CULA tools

GPU Accelerated Linear Algebra

MAGMA

Matrix Algebra on GPU and Multicore



NVIDIA cuFFT

ROGUE WAVE SOFTWARE

IMSL Library

ArrayFire Matrix Computations

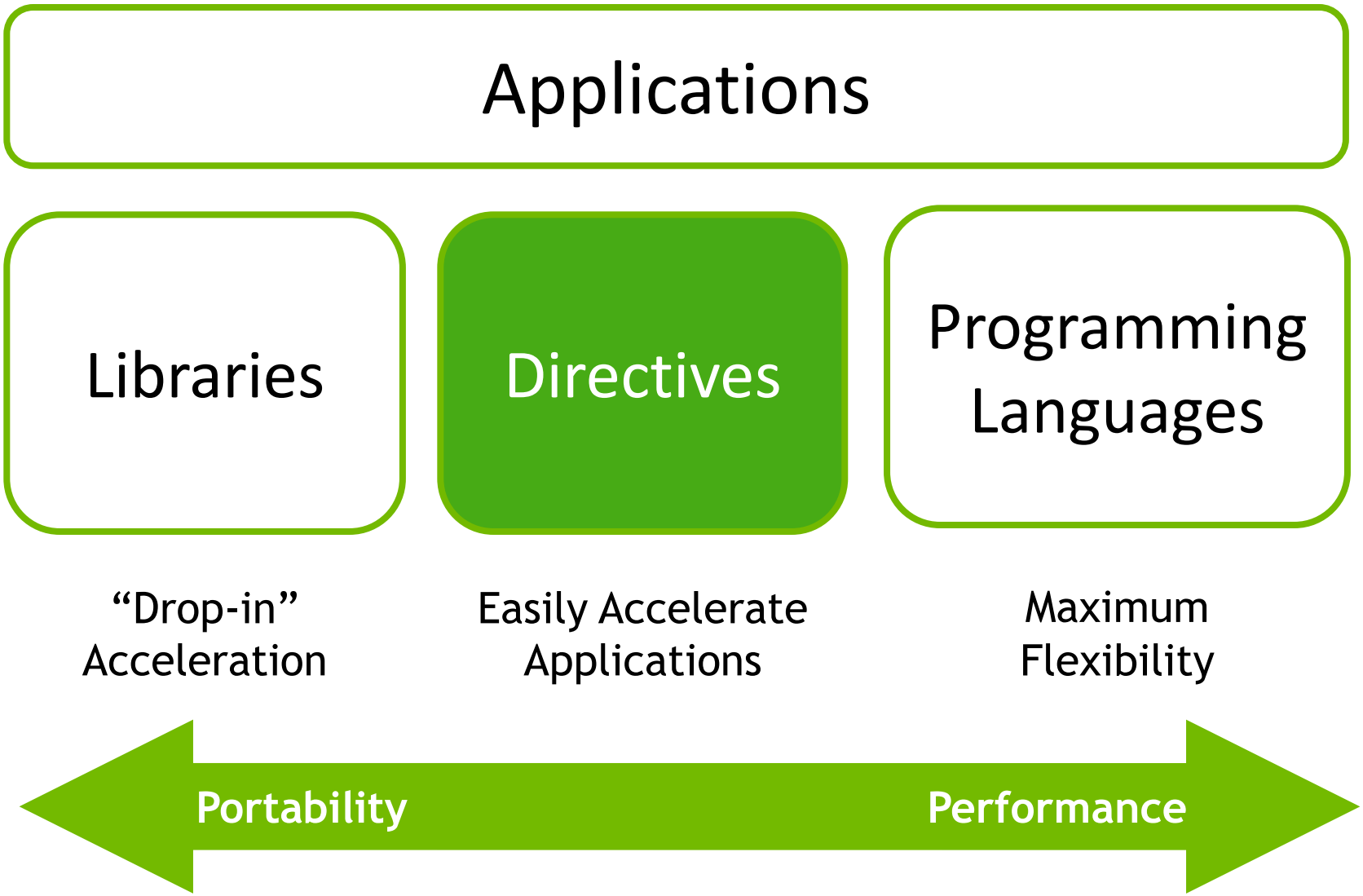
CUSP

Sparse Linear Algebra

Thrust

C++ STL Features for CUDA

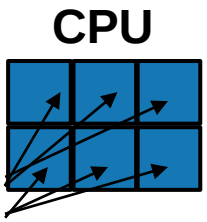
3 Ways to Accelerate Applications

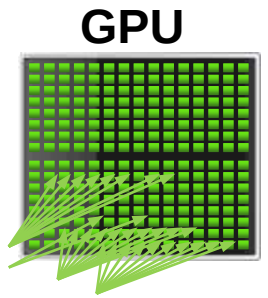


Directive Based Approach

- Directives are added to serial source code
  - Manage loop parallelization
  - Manage data transfer between CPU and GPU memory
- Directives are formatted as comments
  - They don't interfere with serial execution
  - Maintaines portability of original code
- Works with C/C++ or Fortran
- Can be combined with explicit CUDA C/Fortran usage

OpenACC





OpenMP

```
main() {
  double pi = 0.0; long i;

  #pragma omp parallel for reduction(+:pi)
  for (i=0; i<N; i++)
  {
    double t = (double)((i+0.05)/N);
    pi += 4.0/(1.0+t*t);
  }

  printf("pi = %f\n", pi/N);
}
```

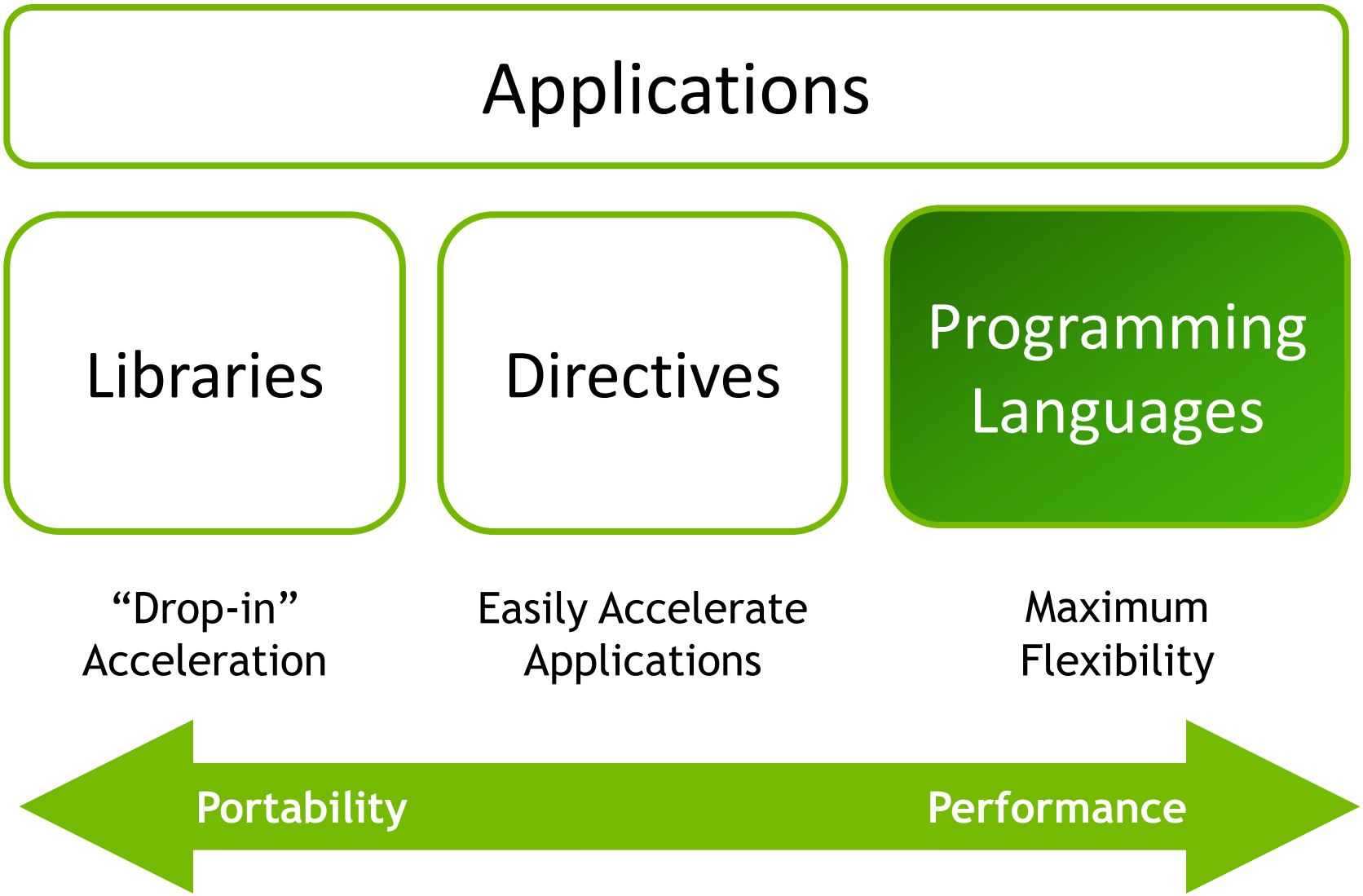
OpenACC

```
main() {
  double pi = 0.0; long i;

  #pragma acc parallel loop reduction(+:pi)
  for (i=0; i<N; i++)
  {
    double t = (double)((i+0.05)/N);
    pi += 4.0/(1.0+t*t);
  }

printf("pi = %f\n", pi/N);
}
```

3 Ways to Accelerate Applications



GPU Programming Languages

- **CUDA** (Compute Unified Device Architecture)
  - a set of extensions to higher level programming language to use GPU as a coprocessor for heavy parallel task
  - a developer toolkit to compile, debug, profile programs and run them easily in a heterogeneous systems
- **OpenCL** (Open Computing Language):
  - a standard open-source programming model developed by major brands of hardware manufacters (Apple, Intel, AMD/ATI, nVIDIA).
    - like CUDA, provides extentions to C/C++ and a developer toolkit
    - extensions for specific hardware (GPUs, FPGAs, MICs, etc)
    - it's very low level (verbose) programming

There are many other approaches and solutions such as SYCL (Krnonos), HIP (AMD), OneAPI (Intel), DirectCompute (Microsoft), ... but current market is basically dominated by CUDA and some OpenCL

Rights & Credits

Slides and examples were authored by:

Isabella Baccarelli, Luca Ferraro, Sergio Orlandini