# Programming paradigms for GPU devices

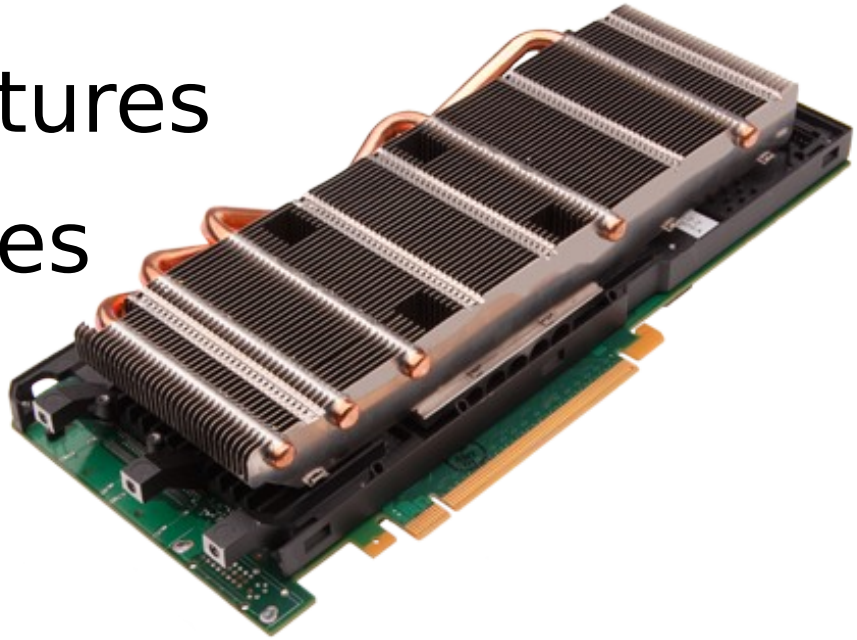

28th Summer School on Parallel Computing

*1-12 July 2019*

***Sergio Orlandini***
s.orlandini@cineca.it

CINECA SCAI
SuperComputing Applications and Innovation

- General Purpose GPU Computing
- CPU vs GPU architectures
- HPC GPU architectures
- GPU programming models

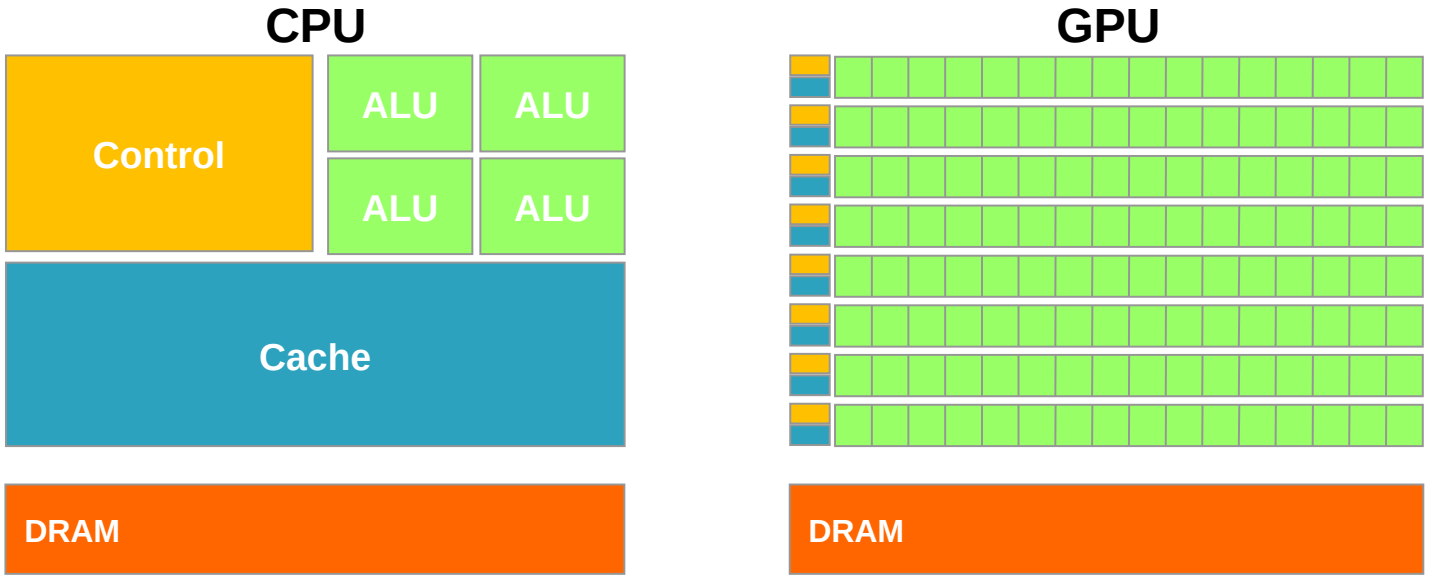


# What is a GPU ?

- **G**raphics **P**rocessor **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

# CPU vs GPU Architectures

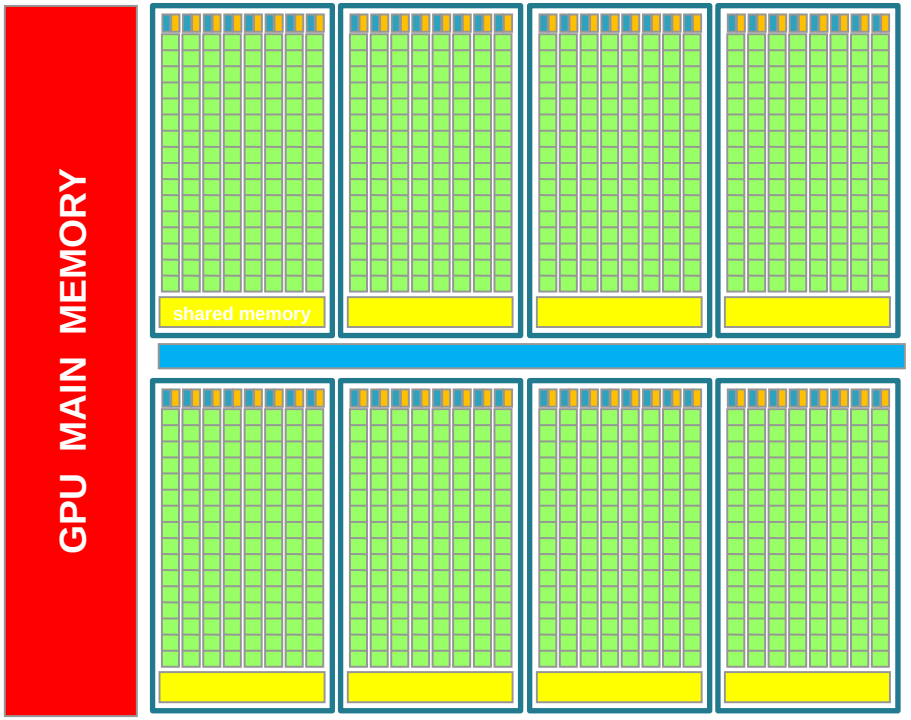
- GPU hardware is specialized for problems which can be classified as *intense data-parallel computations*
  - the same set of operation is executed many times in parallel on different data
  - designed such that more transistors are devoted to data processing rather than data caching and flow control




*"The GPU devotes more transistors to Data Processing"*
(NVIDIA CUDA Programming Guide)

# GPU Architectures

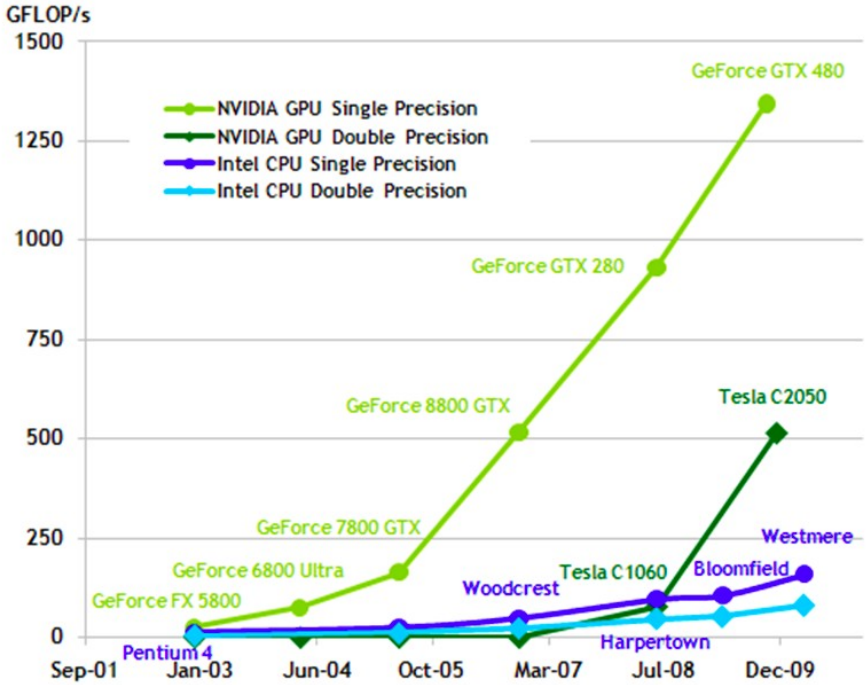
- A typical GPU architecture consists of:
  - Main global memory
    - high bandwidth
  - Streaming Processor
    - grouping independent cores and control units
- Each SM unit has
  - Many ALU cores
  - Instruction scheduler dispatchers
  - Shared memory with very fast access to data

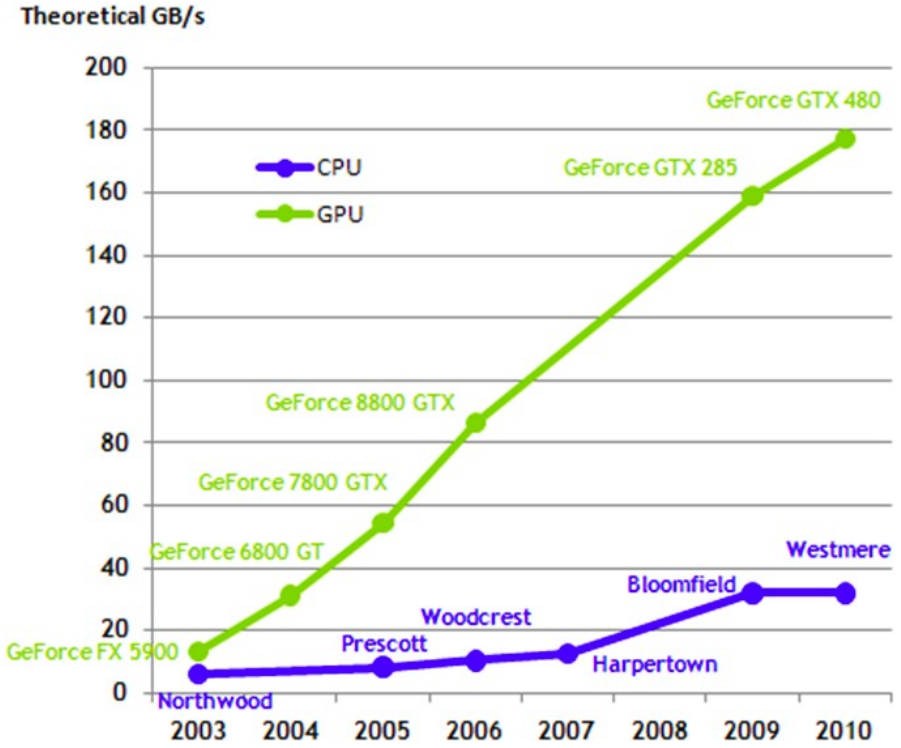


# The concurrency revolution

A new direction in microprocessor design roadmaps

- CPU vendors tend to increase the computational power of single processing unit by increasing the working frequency and adding more higher level control logic and pipelines
- GPU increased the number of processing units, less logic, lowering frequency and dropping down power consumption




Peak GFlops (left) and bandwidth (right) trends of some nVIDIA GPU compared to Intel CPU products

# GPGPU (General Purpose GPU) and GPU computing

- many applications that process large data sets can use a data-parallel programming model 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 3D graphics domain?
- many attemps where made by brave programmers and researchers in order to force GPU APIs to threat their scientific data (atoms, signals, events, etc) as pixel or vertex in order to be computed by the GPU.
- not many survived, still the era of GPGPU computing was just begun ...

# 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 | 14.0 | 5120 | 32 HBM2 | 900 | PCIe 3.0 x 16 |

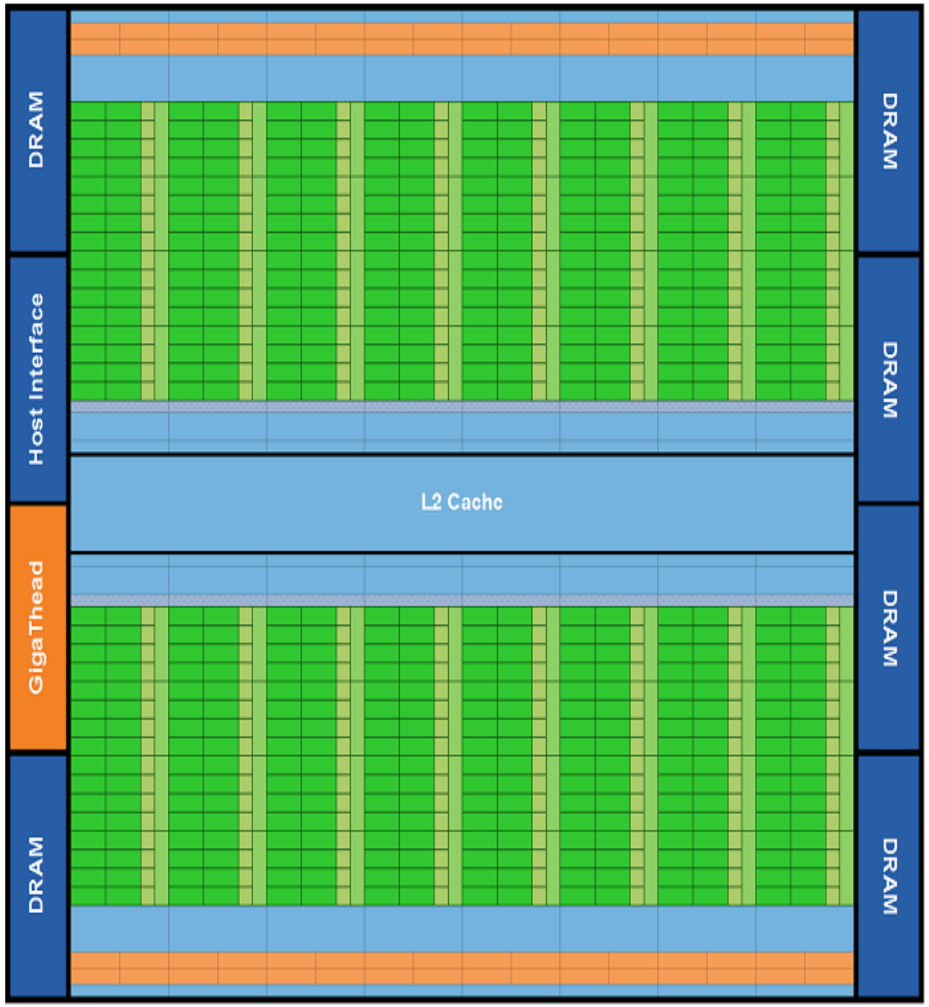
- Tesla serie is the NVIDIA top gamma GPU solution for HPC
  - the GeForce series are for gaming
- HBM2: gen2 high-performance RAM interface for 3D-stacked DRAM with Error-correcting (ECC)

# NVIDIA Architectures naming

- Mainstream & laptops: GeForce
  - Target: videogames and multi-media
- Workstation: Quadro
  - Target: professional graphic applications such as CAD, modeling 3D, animation and visual effects
- GPGPU: Tesla
  - Target: High Performance Computing

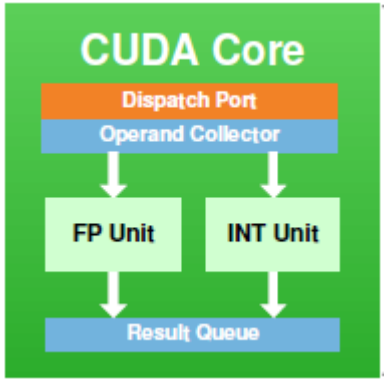
# NVIDIA Fermi Architecture (2009)

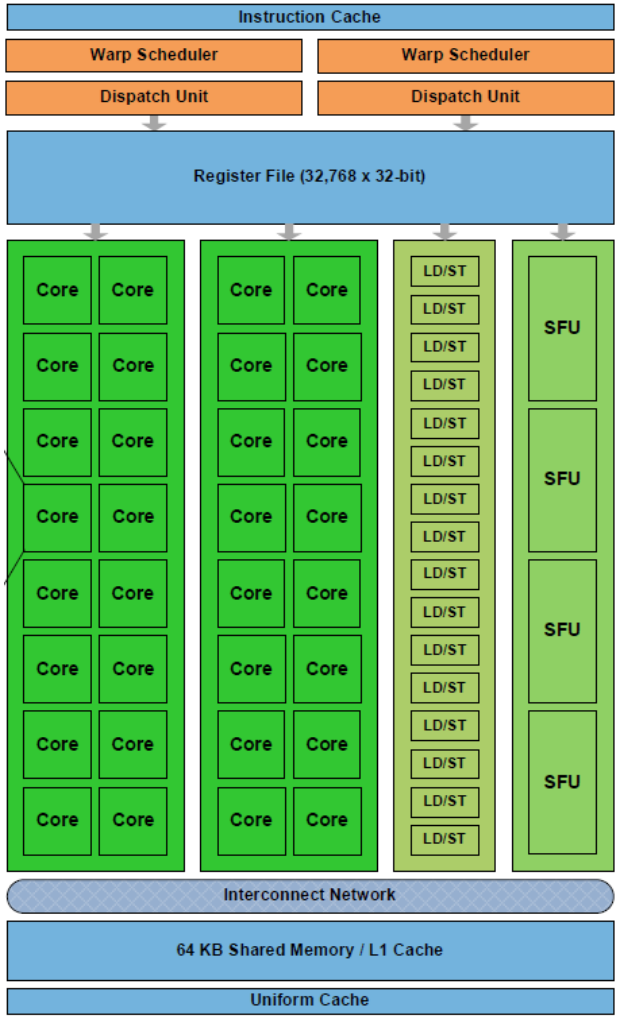
- 16 Streaming Multiprocessors (SM)
- DDR3 Memory
  - 4-6 GB global memory with ECC
- First model with a cache hierarchy:
  - L1 (16-48KB) per SM
  - L2 (768KB) shared among all SM
- 2 independent controllers for data transfer from/to host through PCI-Express
- Global thread scheduler (GigaThread global scheduler) which manage and distribute thread blocks to be processed on SM resources



# Fermi Streaming Multiprocessor (SM)

- 32/48 CUDA cores with an arithmetic logic unit (ALU) and a floating point unit (FPU) fully pipelined
- floating point operations are fully IEEE 754-2008 a 32-bit e a 64-bit
- fused multiply-add (FMA) for both single and double precision
- 32768 registers (32-bit)
- 64KB configurable L1
- shared-memory/cache
- 48-16KB or 16-48KB shared/L1 cache
- 16 load/store units
- 4 Special Function Unit (SFU) to handle trascendental mathematical functions (sin, sqrt, recp-sqrt,..)



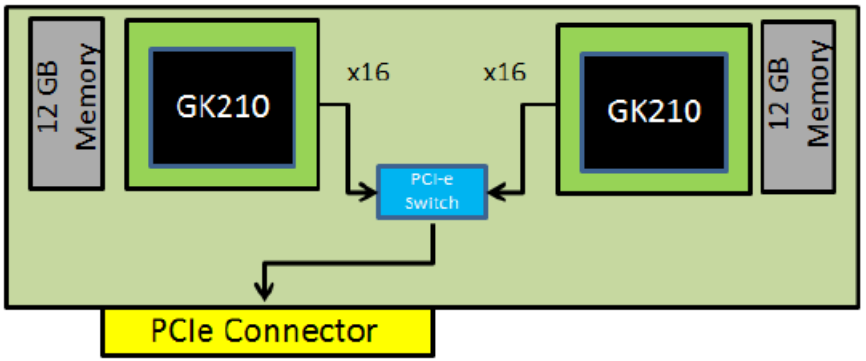
# NVIDIA Kepler Architecture (2012)

- x3 performance/watt with respect to FERMI
- 28nm litography
- 192 CUDA cores
- 4 warp scheduler (2 dispatcher)
  - 2 independent instruction/warp
- 65536 registers per SM (32-bit)
- 32 load/store units
- 32 Special Function Unit
- 1534KB L2 cache (x2 vs Fermi)
- 64KB shared-memory/cache + 48KB read-only L1 cache
- 16 texture units (x4 vs Fermi)
  - Read-only cache
- Hyper-Q technology
  - Enable dynamic parallelism

# GPU nVIDIA K80 (2013)

- Two GPUs (K40) per device
  - 12GB RAM per GPU
  - 480 GB/s memory bandwidth

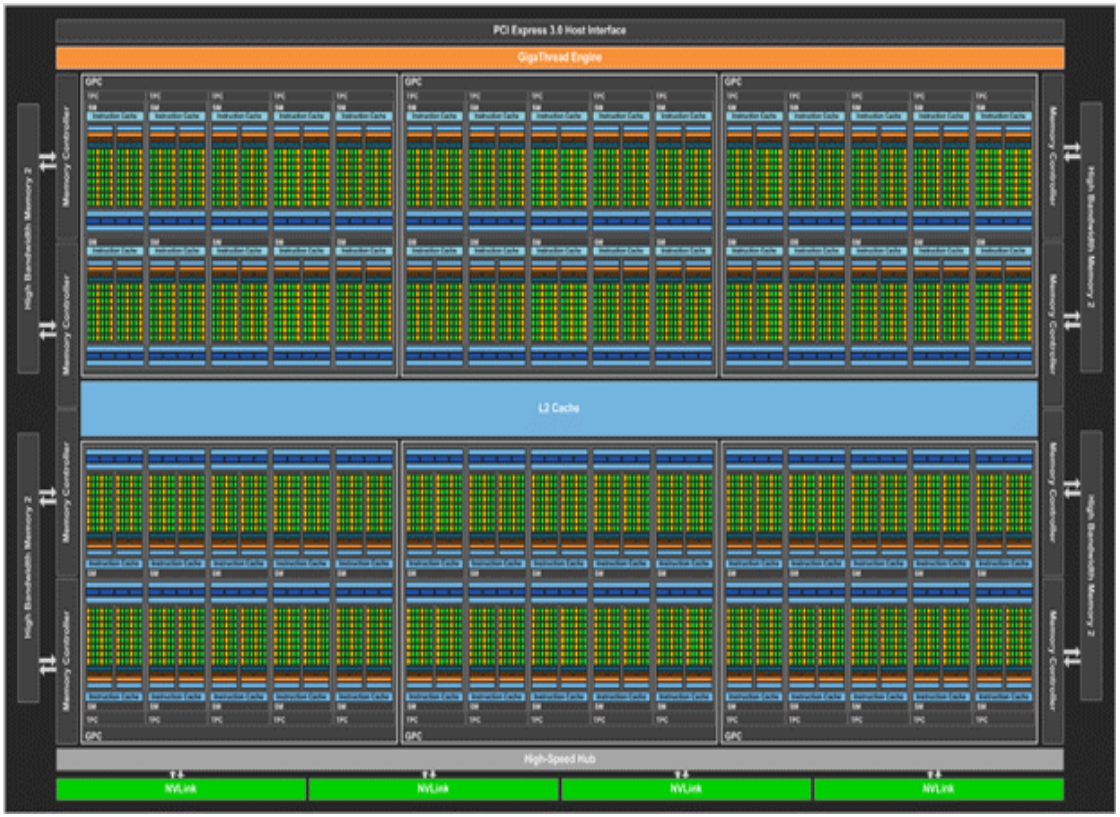

  - 15 SM per GPU
  - 192 CUDA cores/SM
    - total of 2880 cuda cor




  - 500-800 MHz clock
  - 250W

# NVIDIA Pascal Architecture (2016)

- 6 Compute Graphic Clusters (CGC) with 10 SM each
- 16nm litography
  - 2X Watt/Flop respect Kepler architecture
- 4MB L2 cache
- 3D stacked RAM HDDR5
- High Bandwidth Memory
  - 16GB RAM
  - 760 GB/s bandwidth
- NVLink tecnology
  - 80GB/s bandwidth to host data transfers
  - 5X respect PCIe Gen3 16x




Peak Performance: 5,7 TFlops

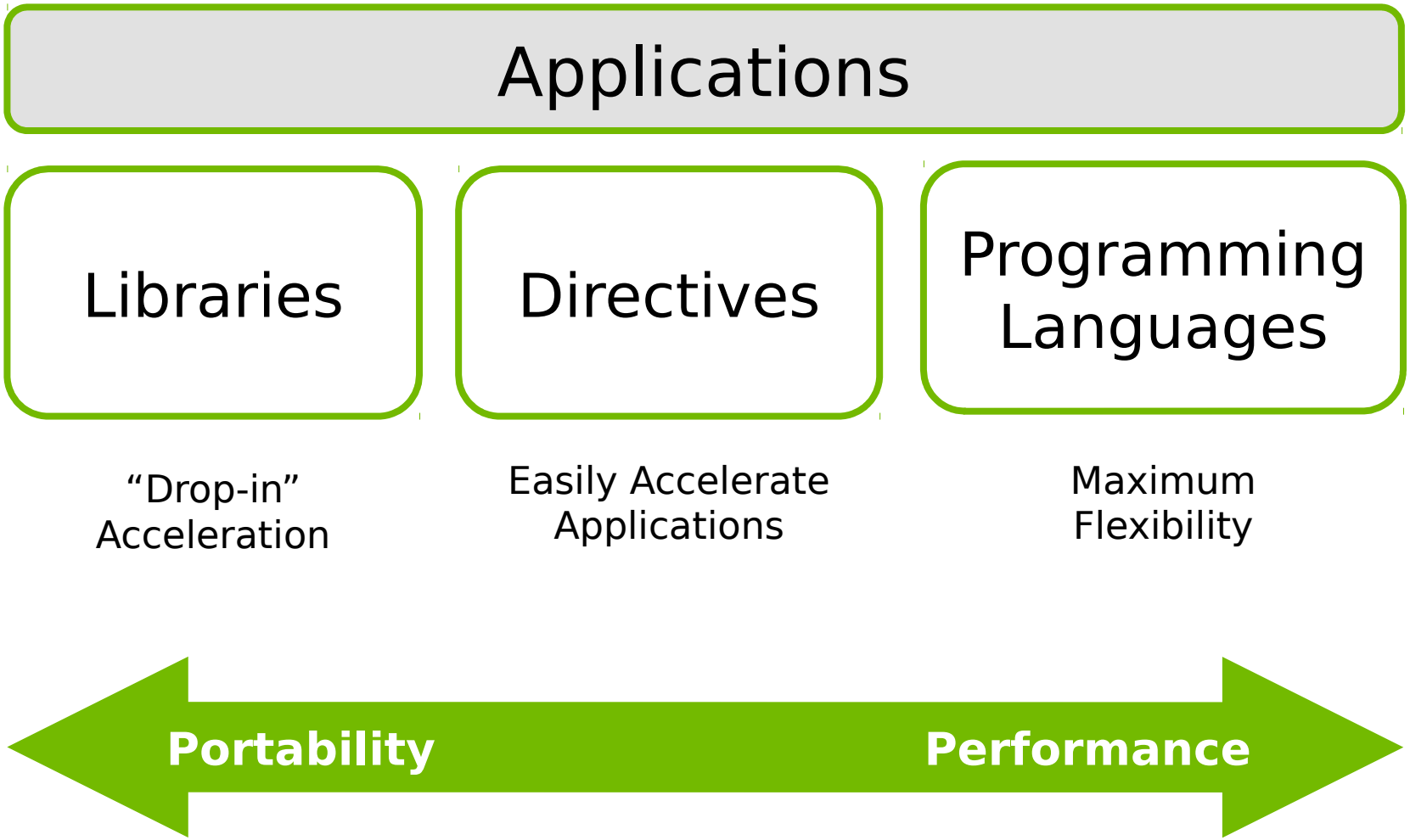
# NVIDIA Pascal Architecture (2016)

- SM composed of two independent blocks
- Each block sports:
  - 1 warps x 2 dispatchers
  - 32 ALU SIMD units
  - 16FP64 units
  - 8 Load/Store units
  - 8 SFU units
  - 32768 32bits registers
- Each block accesses:
  - 64KB shared memory
  - L1 64KB cache
  - 4 texture units

# GPGPU Programming Model

- GPU is seen as an auxiliary coprocessor equipped with
  - thousands of cores
  - global memory with high bandwidth
- *computational-intensive data-parallel* regions of a program can be executed on the GPU device
  - thousands of threads will be executed on the GPU
  - each thread will insist on a different GPU core
  - each thread can acts on a different data element independently
  - the GPU parallelism is very close to the SPMD paradigm
- the more the working thread, the better are the performances
  - GPU threads are very *light*
    - no penalty is paid in case of *context-switch* (each thread has its own registers)
    - the more the threads, the more the chance to hide memory or computational latencies

# 3 Ways to Accelerate Applications



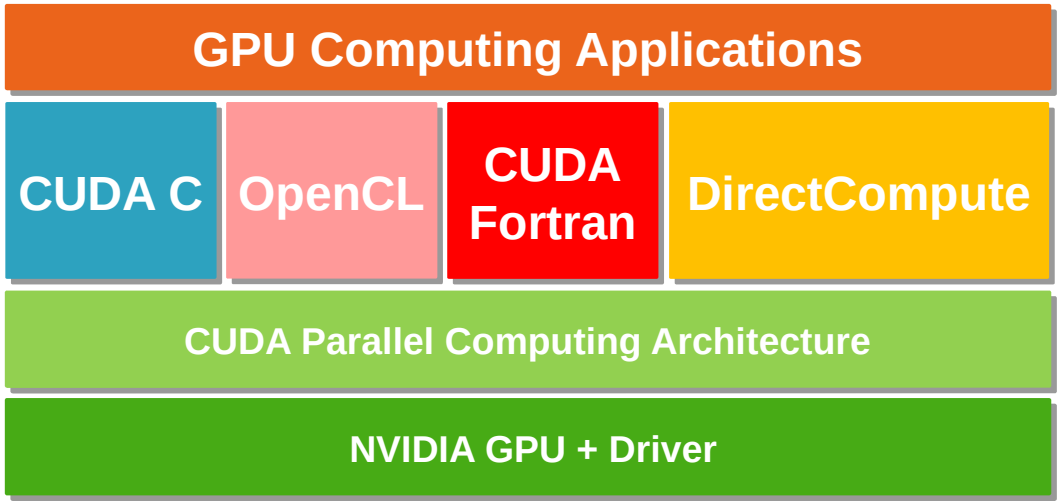
# GPGPU Programming Approaches

- nVIDIA 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
- Accelerator Directives Approach
  - OpenACC
  - OpenMP v4.x  accelerator directives
  - you hope your compiler understand what you want, and do a good job
- Library Based:
  - MAGMA, CUDA Libraries, StarPu, ArrayFire, etc

# General-Purpose Parallel Computing Architecture

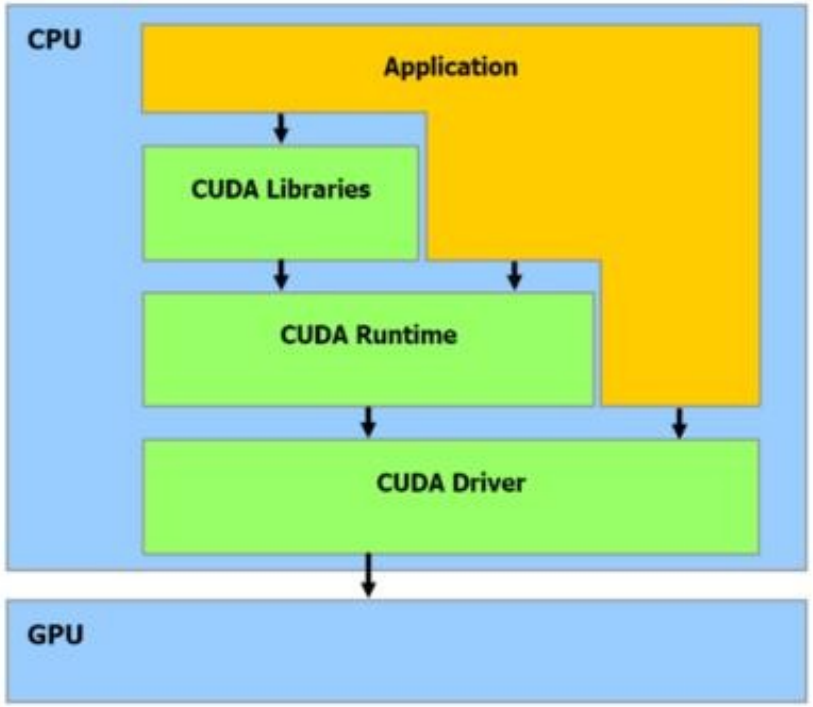
Compute Unified Device Architecture (CUDA)

- a general purpose parallel computing platform and programming model that easy GPU programming, which provides:
  - a new hierarchical multi-threaded programming paradigm
  - a new architecture instruction set called PTX (Parallel Thread eXecution)
  - a small set of syntax extensions to higher level programming languages (C, Fortran) to express thread parallelism within a familiar programming environment
  - A complete collection of development tools to compile, debug and profile CUDA programs.



# CUDA Driver Vs Runtime API

- CUDA is composed of two APIs:
  - the CUDA runtime API
  - the CUDA driver API
- They are mutually exclusive
- Runtime API:
  - easier to program
  - it eases device code management: it's where the C-for-CUDA language lives
- Driver API:
  - requires more code: no syntax sugar for the kernel launch, for example
  - finer control over the device expecially in multithreaded application
  - doesn't need nvcc to compile the host code.



# CUDA Driver API

- The driver API is implemented in the nvcuda dynamic library. All its entry points are prefixed with cu.
- It is a handle-based, imperative API: most objects are referenced by opaque handles that may be specified to functions to manipulate the objects.
- The driver API must be initialized with cuInit() before any function from the driver API is called. A CUDA context must then be created that is attached to a specific device and made current to the calling host thread.
- Within a CUDA context, kernels are explicitly loaded as PTX or binary objects by the host code**.
- Kernels are launched using API entry points.

- **by the way, any application that wants to run on future device architectures must load PTX, not binary code

# Compute Capability

- the *compute capability* of a device describes its architecture
  - *registers, memory sizes, features and capabilities*
- the compute capability is identified by a code like "`compute_Xy`"
  - major number (X): identifies base line chipset architecture
  - minor number (y): indentifies variants and releases of the base line chipset
- a compute capability select the set of usable PTX instructions

| *compute capability* | *feature support* |
|---|---|
| **compute_20** | FERMI architecture |
| **compute_30** | KEPLER K10 architecture (only single precision) |
| **compute_35** | KEPLER K20, K20X, K40 architectures |
| **compute_37** | KEPLER K80 architecture (two K40 on a single board) |
| **compute_53** | MAXWELL GM200 architecture (only single precision) |
| **compute_60** | PASCAL GP100 architecture |

# Capability: resources constraints

| Technical Specifications | Compute Capability | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1.0 | 1.1 | 1.2 | 1.3 | 2.x | 3.0 | 3.5 |
| Maximum dimensionality of grid of thread blocks | 2 | | | | 3 | | |
| Maximum x-dimension of a grid of thread blocks | 65535 | | | | | $2^{31}$-1 | |
| Maximum y- or z-dimension of a grid of thread blocks | 65535 | | | | | | |
| Maximum dimensionality of thread block | 3 | | | | | | |
| Maximum x- or y-dimension of a block | 512 | | | | 1024 | | |
| Maximum z-dimension of a block | 64 | | | | | | |
| Maximum number of threads per block | 512 | | | | 1024 | | |
| Warp size | 32 | | | | | | |
| Maximum number of resident blocks per multiprocessor | 8 | | | | | 16 | |
| Maximum number of resident warps per multiprocessor | 24 | | 32 | | 48 | 64 | |
| Maximum number of resident threads per multiprocessor | 768 | | 1024 | | 1536 | 2048 | |
| Number of 32-bit registers per multiprocessor | 8 K | | 16 K | | 32 K | 64 K | |
| Maximum number of 32-bit registers per thread | 128 | | | | 63 | | 255 |
| Maximum amount of shared memory per multiprocessor | 16 KB | | | | 48 KB | | |
| Number of shared memory banks | 16 | | | | 32 | | |
| Amount of local memory per thread | 16 KB | | | | 512 KB | | |
| Constant memory size | 64 KB | | | | | | |
| Cache working set per multiprocessor for constant memory | 8 KB | | | | | | |
| Cache working set per multiprocessor for texture memory | Device dependent, between 6 KB and 8 KB | | | | | | |
| Maximum width for a 1D texture reference bound to a CUDA array | 8192 | | | | 65536 | | |

- CUDA programming model
  - Heterogeneous execution
  - Writing a CUDA Kernels
  - Thread Hierarchy
- Getting started with CUDA programming:
  - Vector-Vector Add
  - Handling data transfers from CPU to GPU and back
  - Write and launch a CUDA program

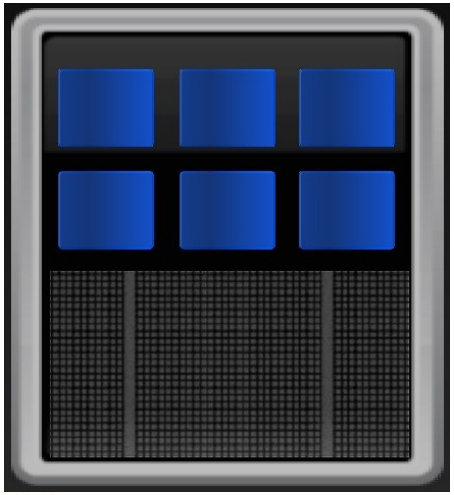
# GPU Programming Model

- GPU is seen as an auxilirary coprocessor with its own memory space
- *data-parallel*, *computational-intensive* portions of a program can be executed on the GPU
  - each *data-parallel* computational portion can be isolated into a function, called CUDA kernel, that is executed on the GPU
  - CUDA kernels are executed by many different threads in parallel
  - each thread can compute different data elements independently
  - the GPU parallelism is very close to the SPMD (Single Program Multiple Data) paradigm. Single Instruction Multiple Threads (SIMT) according to the Nvidia definition.
- GPU threads are extremely *light weight*
  - no penalty in case of a *context-switch* (each thread has its own registers)
  - the more are the threads *in flight*, the more the GPU hardware is able to hide memory or computational latencies, i.e better overall performances at executing the kernel function

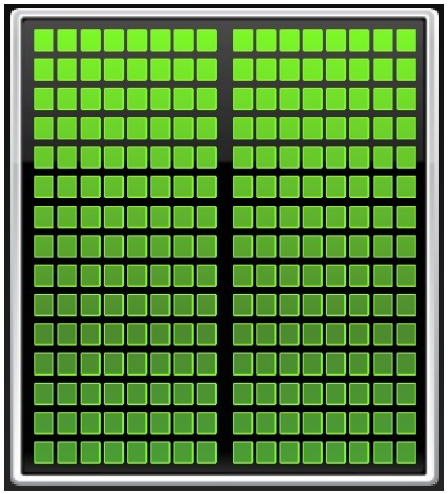
# GPGPU Programming Model

- 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

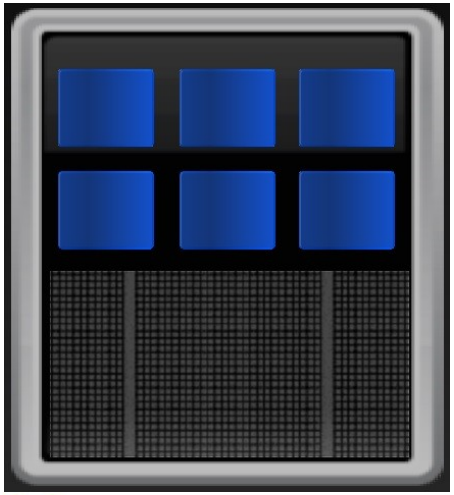


# GPGPU Programming Model

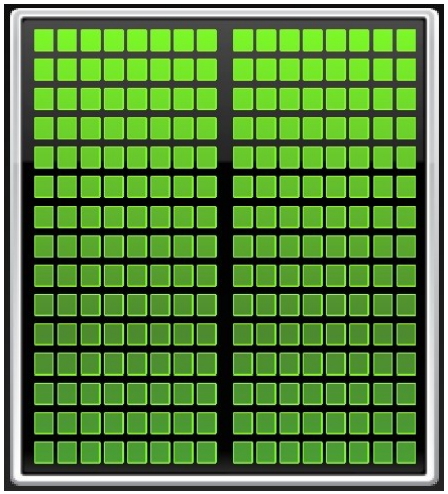
- Optimized for low-latency accesses to caches data sets
- Control logic for out-of-order and speculative execution

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

CPU

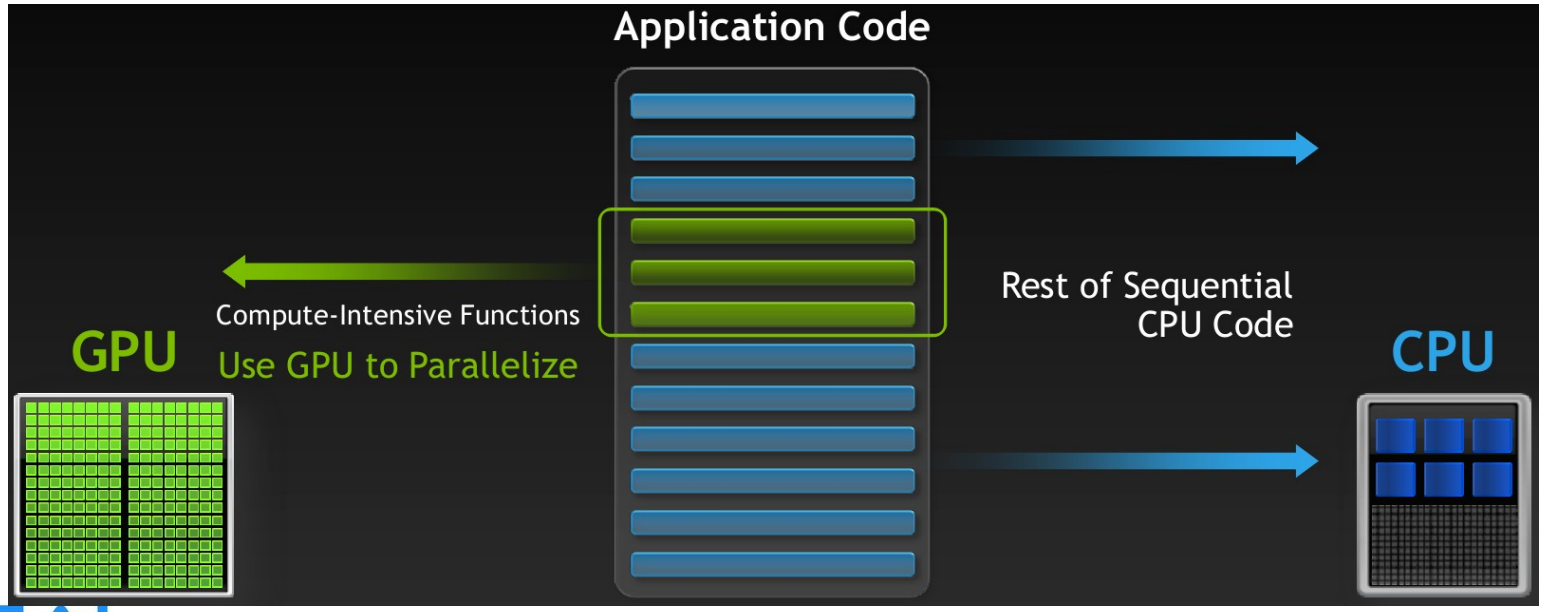


GPU



# 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



# GPGPU Programming Model

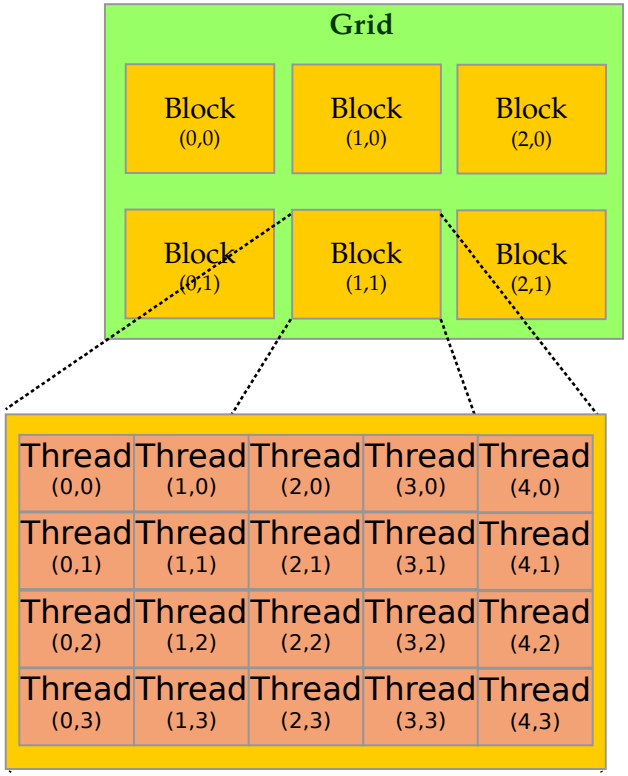
- A function which runs on a GPU is called "**kernel**"
  - when a kernel is launched on a GPU thousands of threads will execute its code
  - programmer chooses the number of threads to run
  - each thread act on a different data element independently
  - the GPU parallelism is very close to the SPMD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

```
void vecAddCPU (int N, float *A,
                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 );
```

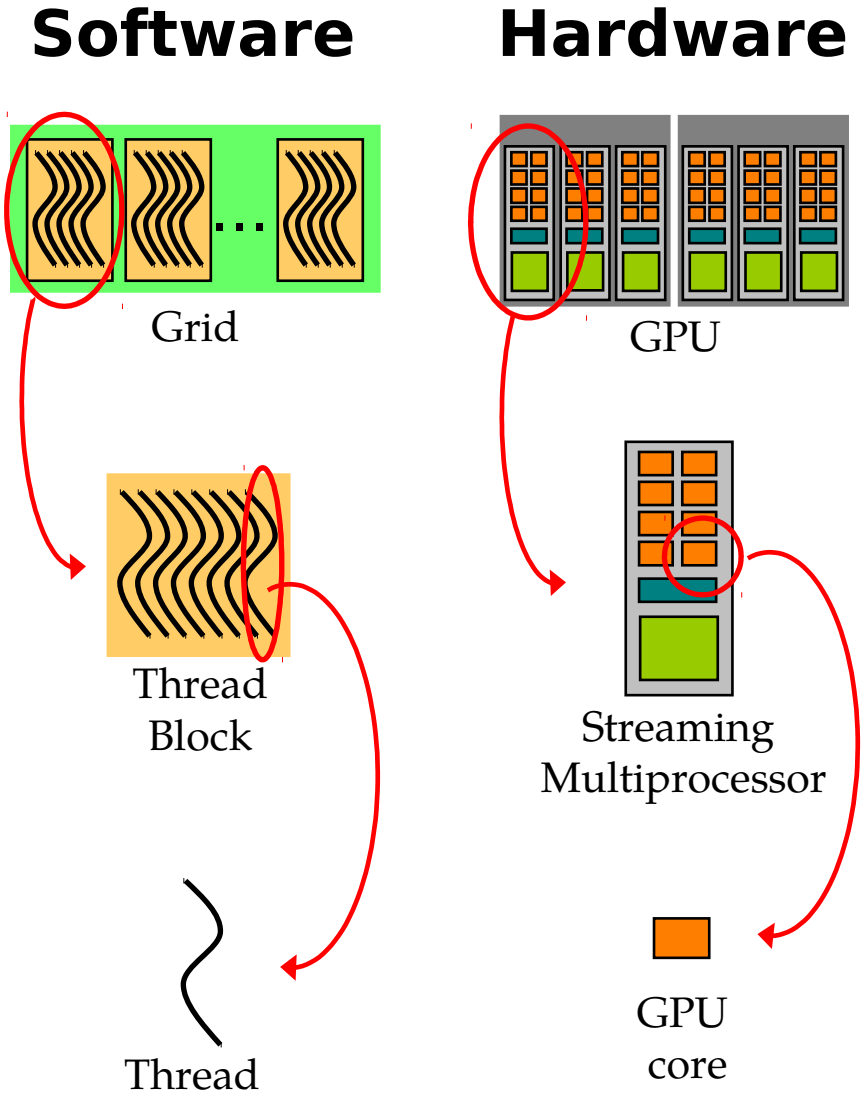
```
void vecAddGPU (int N, float *A,
                float *B, float *C)
{
    int i = threadIdx.x;
    if ( i < N)
       c[i] = a[i] + b[i];
}
...
// CUDA kernel vecAddGPU on N elems
vecAddGPU<<<1, N>>>( N, a, b, c );
```

# 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 together in *teams* or *blocks* of threads
- Threads belonging to the same block or team can cooperate together exchanging data through the shared memory area



# more on the GPU Execution Model

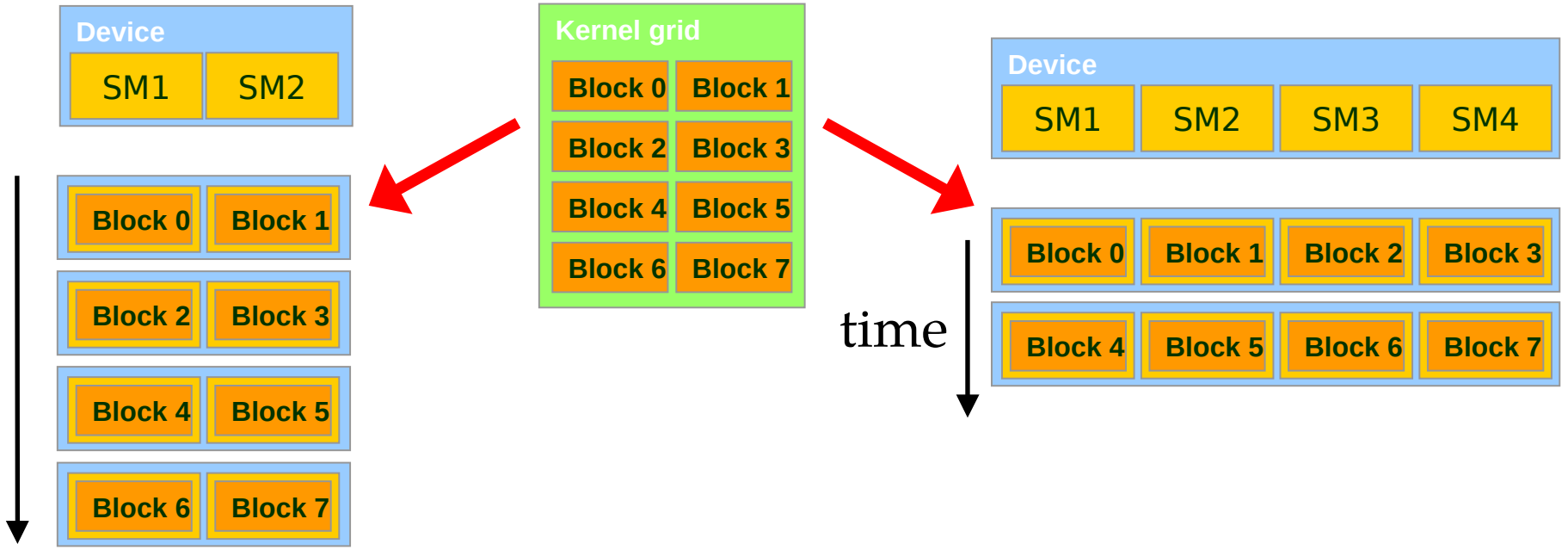



when a kernel is invoked:

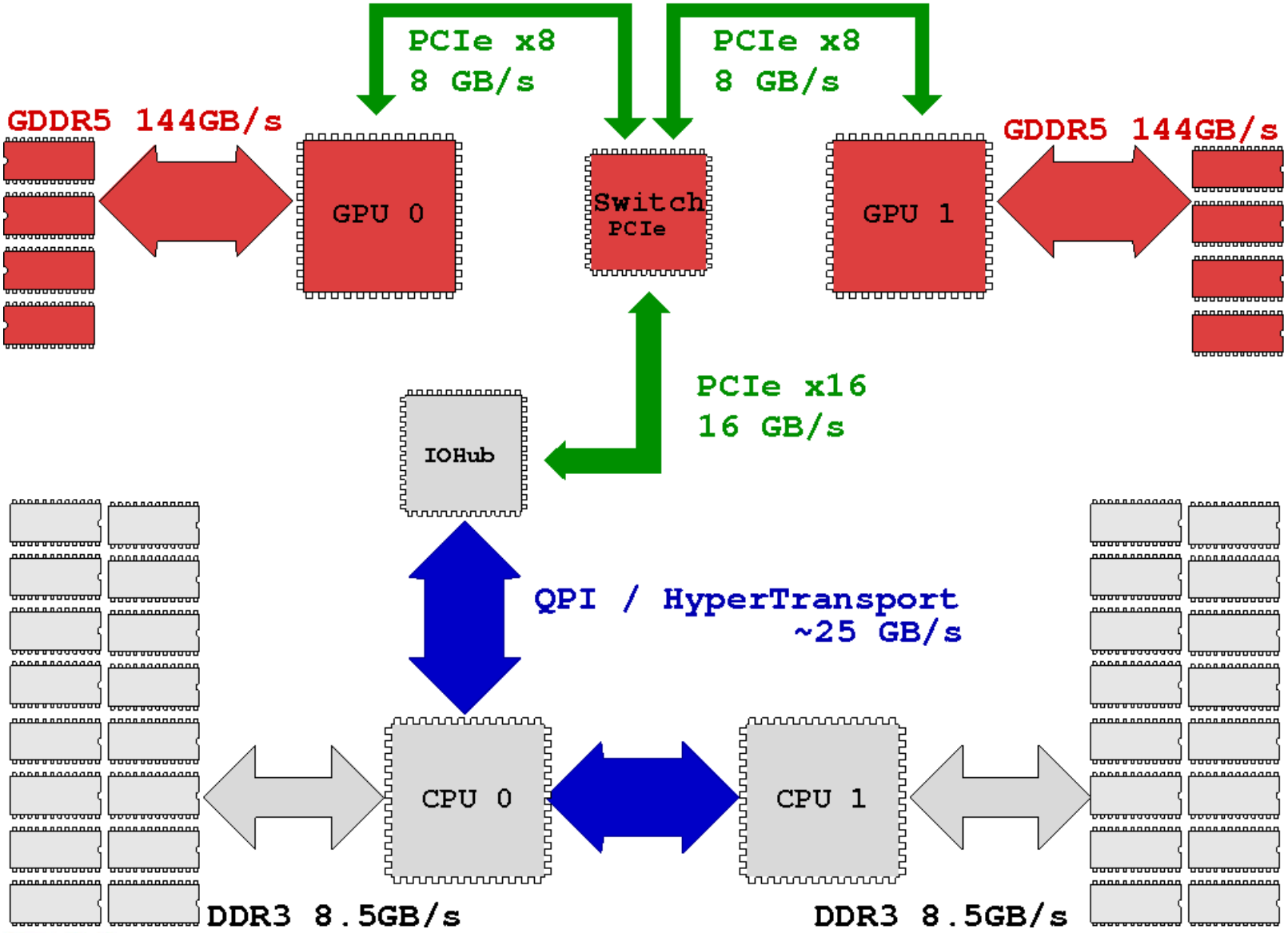
- each thread block is assigned to a SM in a round-robin mode
  - a maximum number of blocks can be assigned to each SM, depending on hardware generation and on how many resorses each block needs to be executed (registers, shared memory, etc)
  - the runtime system maintains a list of blocks that need to execute and assigns new blocks to SMs as they complete the execution of blocks previously assigned to them.
  - 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)
- thread of each block are partitioned into warps of 32 *threads* each, so to map each thread with a unique consecutive thread index in the block, starting from index 0.
- the scheduler select for execution a warp from one of the residing blocks in each SM.
- A warp execute one common 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

# Transparent Scalability

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

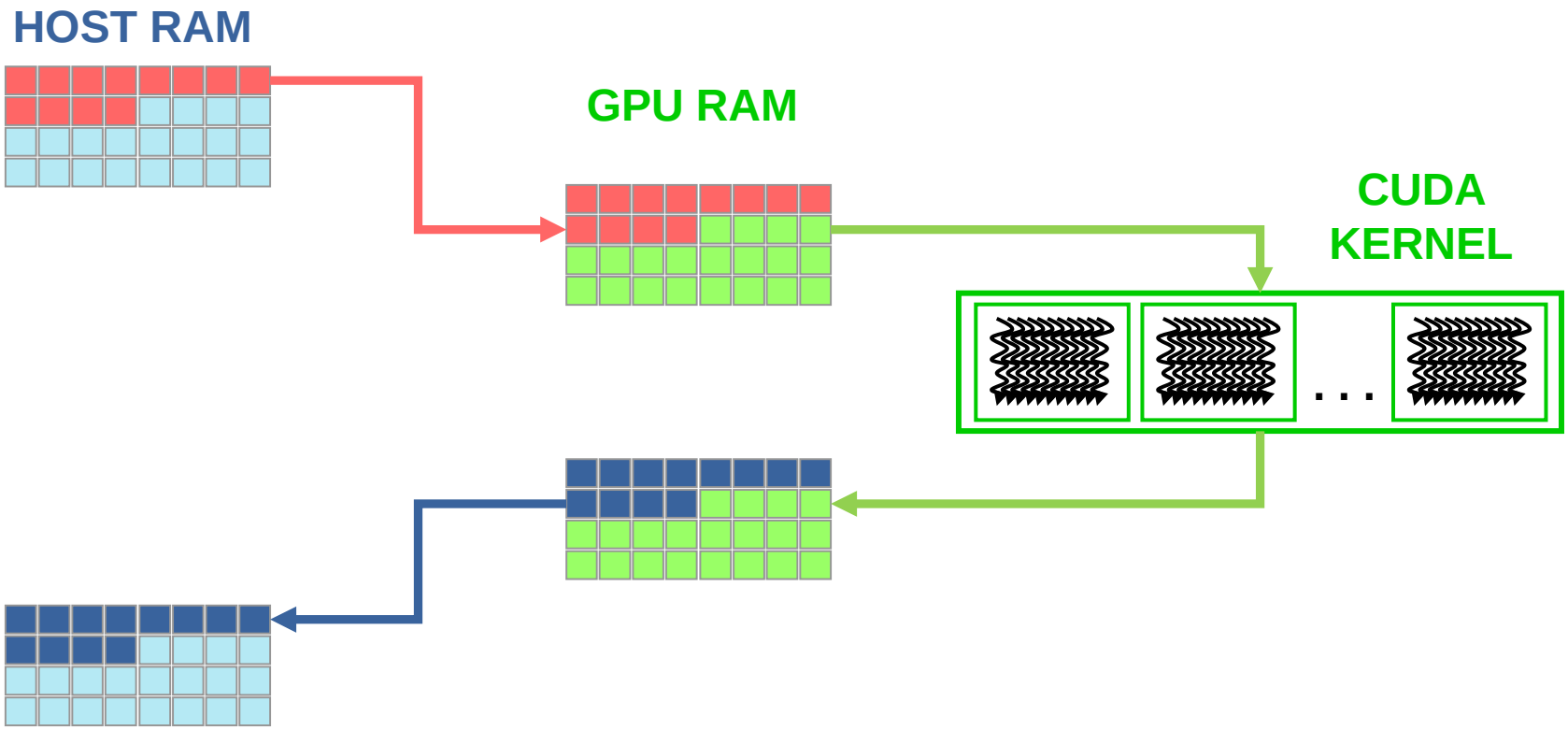


# Connection Scheme of *host/device*



# 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



# D2H and H2D Data Transfers

- GPU devices are connected to the host with a PCIe bus
  - PCIe bus is characterized by very low latency, but also by a low bandwidth with respect to other bus

| Technology | Peak Bandwidth |
|---|---|
| PCIex GEN2 (16x, full duplex) | 8 GB/s (peak) |
| PCIex GEN3 (16x, full duplex) | 16 GB/s (peak) |
| DDR3 (full duplex) | 26 GB/s (single channel) |

- data transfer can easily become a bottleneck in heterogeneous environment equipped with accelerators
  - strive to minimize transfers or execute them in overlap with computations (advanced technique, more on this later)

# NVLink NVIDIA Technology

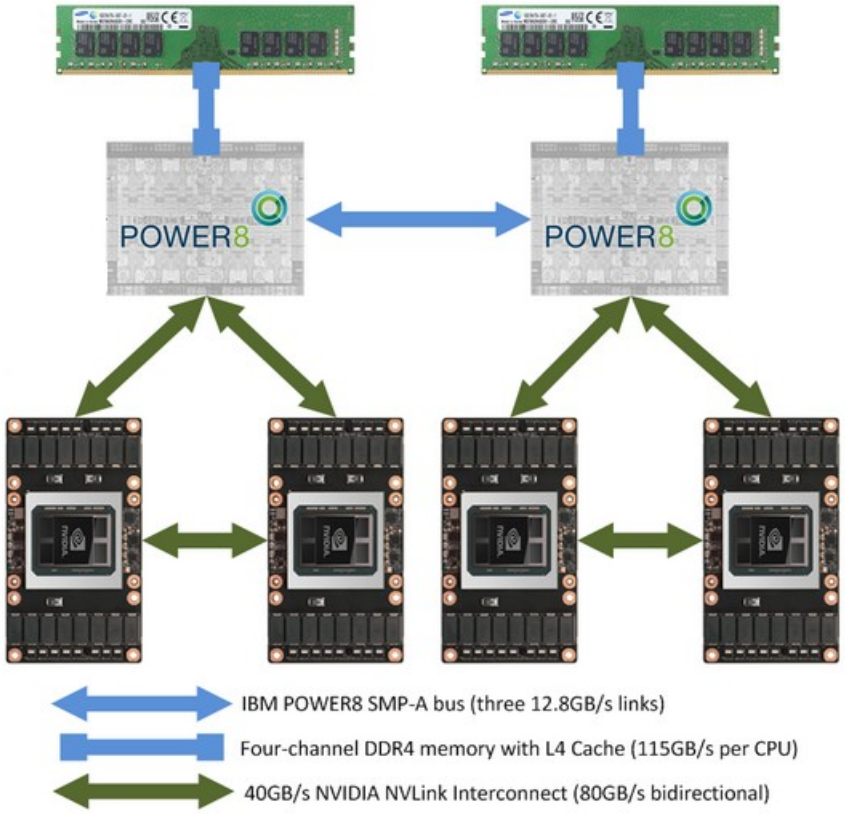
- NVLink is a wire-based communications protocol serial multi-lane near-range communication link developed by Nvidia.
- NVLink specifies a point-to-point connections with data rates of 20 and 25 Gbit/s (v1.0/v2.0) per data lane per direction.
- Total data rates in real world systems are 160 and 300 GByte/s (v1.0/v2.0) for the total system sum of input and output data streams.

# NVLink Transfers on D.A.V.I.D.E.

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 457.93 | 35.30 | 20.37 | 20.40 |
| 2 | 35.30 | 454.78 | 20.16 | 20.14 |
| 3 | 20.19 | 20.16 | 454.56 | 35.29 |
| 4 | 18.36 | 18.42 | 35.29 | 454.07 |

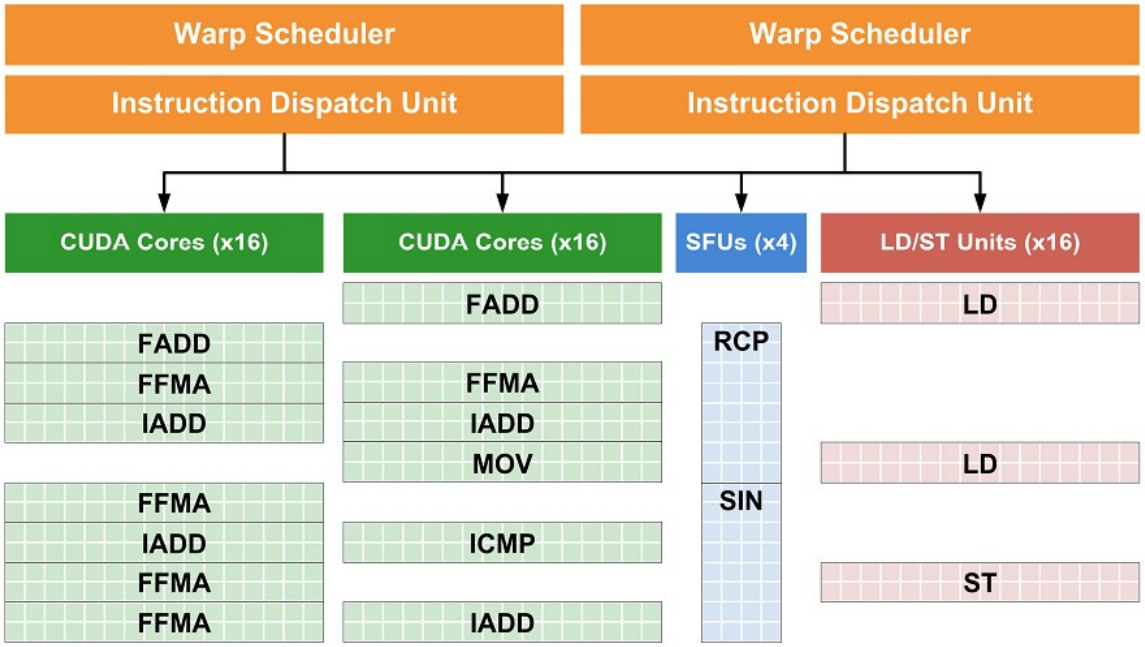
Bandwidth in GB/s transferring 1GB of data from device-to-device with peer-to-peer communications.

On each GPU, the available 80GB/s bandwidth was divided in half. One link goes to a POWER8 CPU and one link goes to the adjacent P100 GPU.



# Warps

- The GPU multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*.
- Individual threads composing a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently
- each *warp* can execute instructions on
  - SM cores
  - load/store units
  - SFUs units

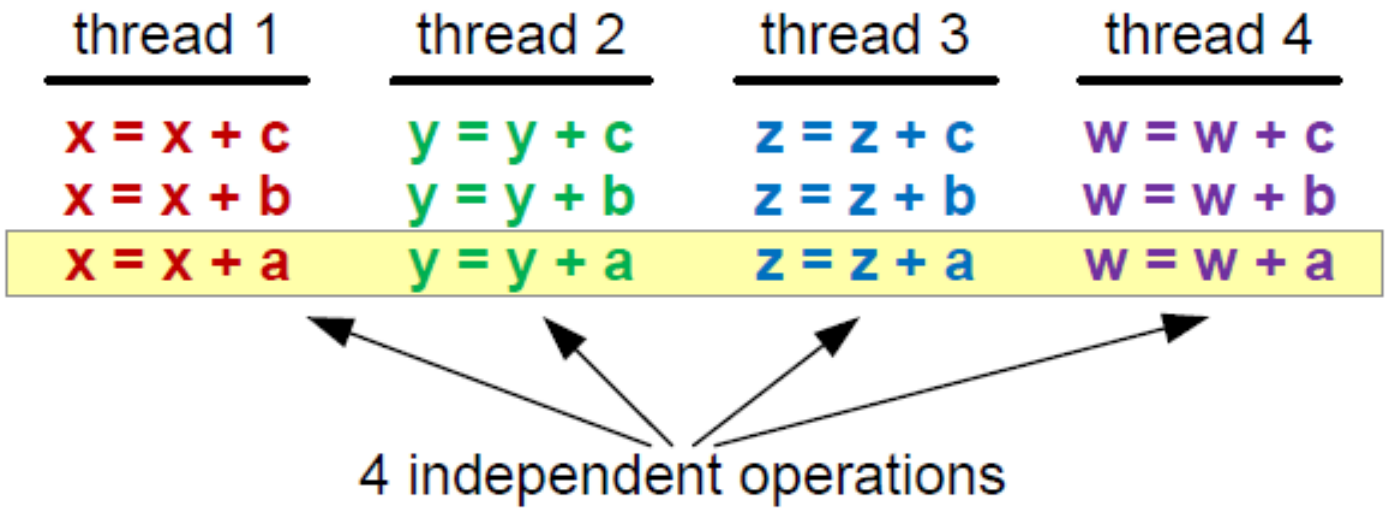


# Hiding Latencies

- What is latency?
  - the number of clock cycles needed to complete an istruction
  - ... that is, the number of cycles I need to wait for before another **dependent operation** can start
    - arithmetic latency (~ 18-24 cycles)
    - memory access latency (~ 400-800 cycles)
- We cannot discard latencies (it's an hardware design effect), but we can lesser their effect and hide them.
  - saturating computational pipelines in computational bound problems
  - saturating bandwidth in memory bound problems
- We can organize our code so to provide the scheduler a sufficient number of **independent operations,** so that the more the warp are available, the more content-switch can hide latencies and proceed with other useful operations
- There are two possible ways and paradigms to use (can be combined too!)
  - Thread-Level Parallelism (TLP)
  - Instruction-Level Parallelism (ILP)

# Thread-Level Parallelism (TLP)

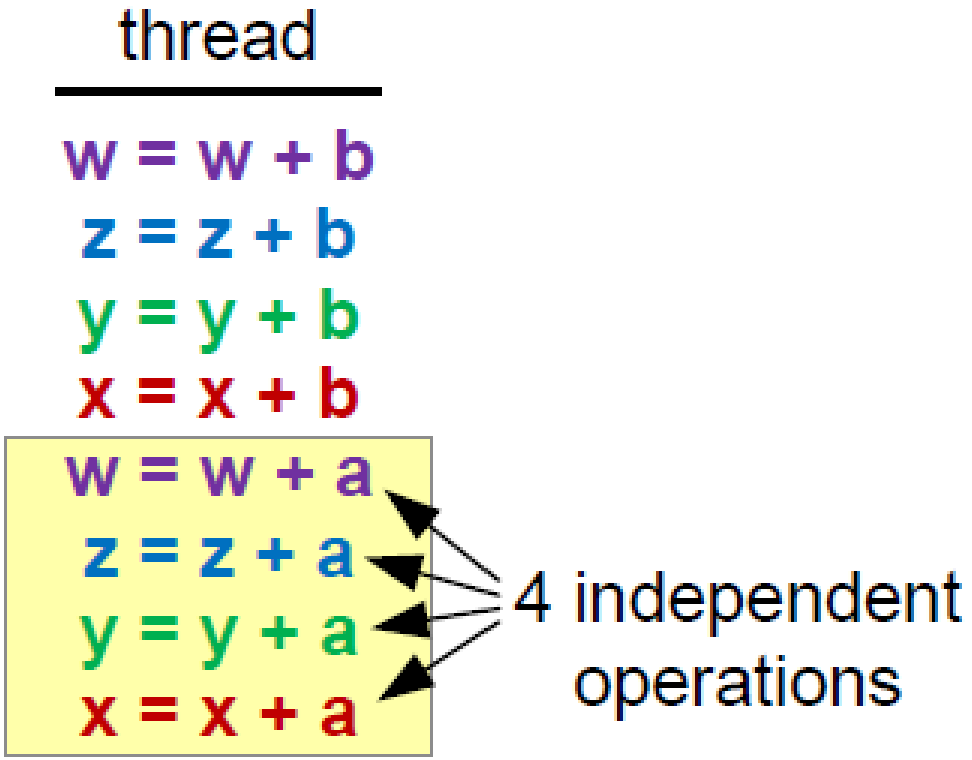
- Strive for high SM occupancy: that is try to provide as much threads per SM as possible, so to easy the scheduler find a warp ready to execute, while the others are still busy
- This kind of approach is effective when there is a low level of independet operations per CUDA kernels



# Instruction-Level Parallelism (ILP)

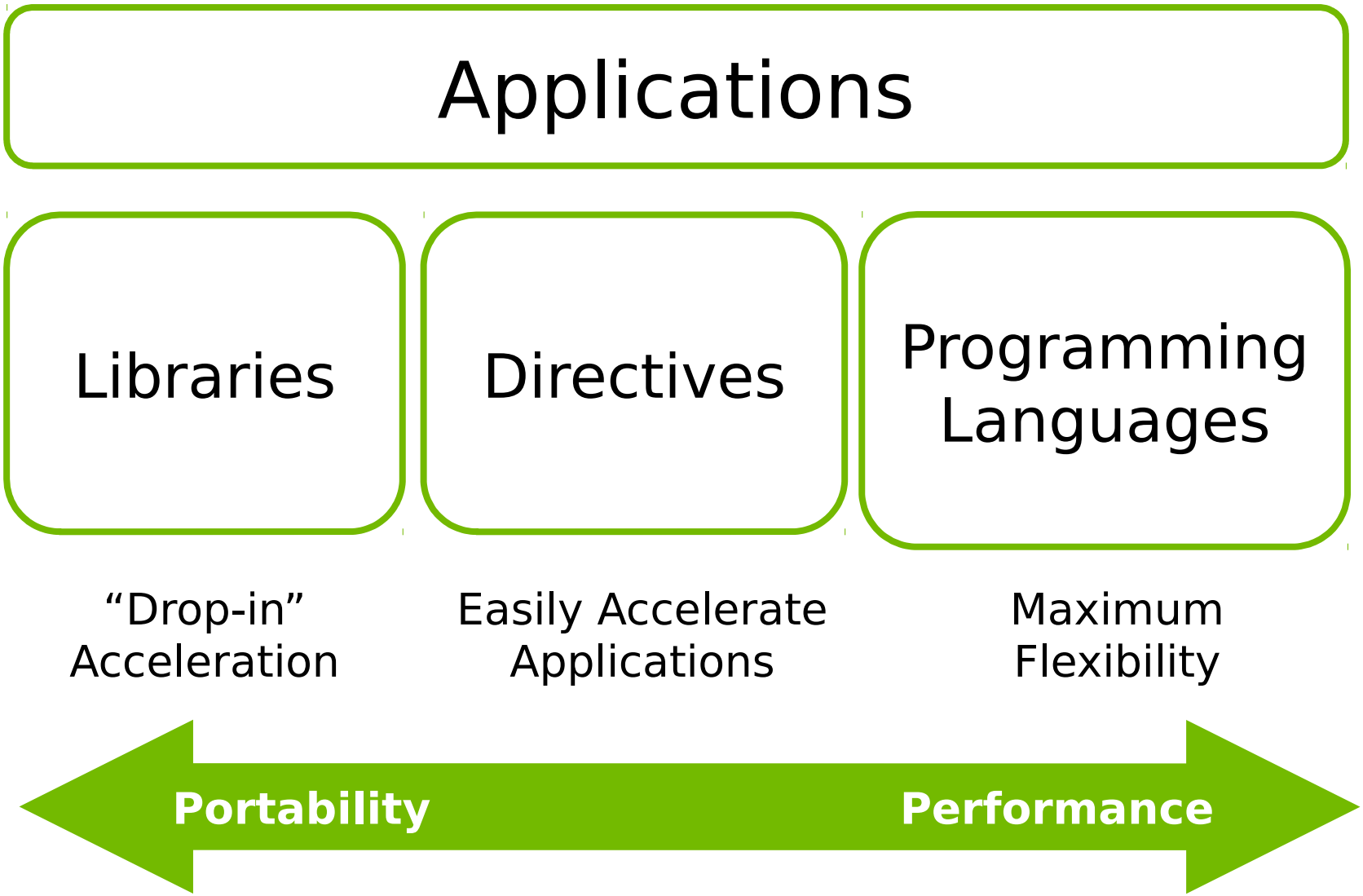
- Strive for multiple independent operations inside you CUDA kernel: that is, let your kernel act on more than one data
- this will grant the scheduler to stay on the same warp and fully load each hardware pipeline

- note: the scheduler will not select a new warp until there are eligible instructions ready to execute on the current warp

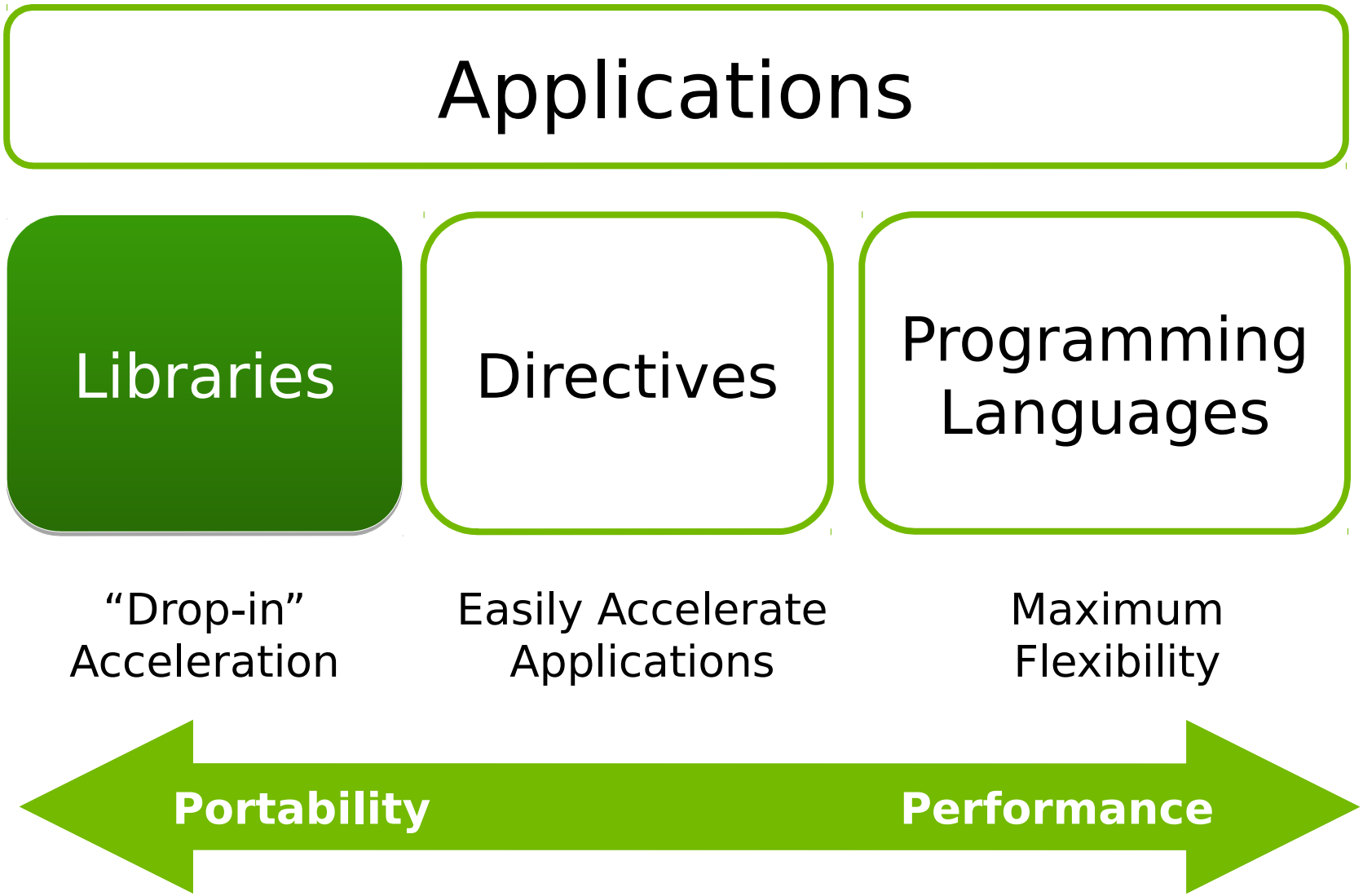


- GPU programming models
  - GPU enabled libraries
  - high level directives
  - low level programming languages

# 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:**

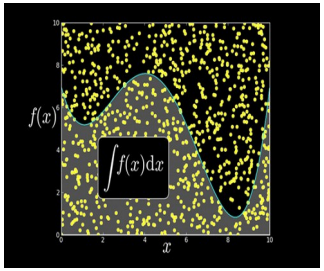
  Libraries offer high-quality implementations of functions encountered in a broad range of applications

- **Performance:**

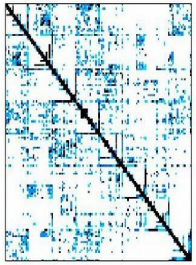
  libraries are tuned by experts

# Some GPU-accelerated Libraries

cuBLAS




NVIDIA cuRAND



NVIDIA cuSPARSE

NVIDIA NPP

GPU VSIPL

Vector Signal Image Processing

CULA tools

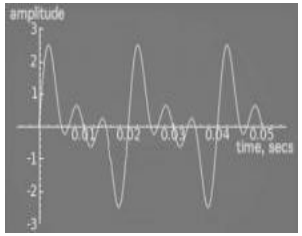
GPU Accelerated Linear Algebra

MAGMA

ICL UT

Matrix Algebra on GPU and Multicore

open source initiative




NVIDIA cuFFT

ROGUE WAVE SOFTWARE

IMSL Library

**ArrayFire Matrix Computations**

CUSP

Sparse Linear Algebra

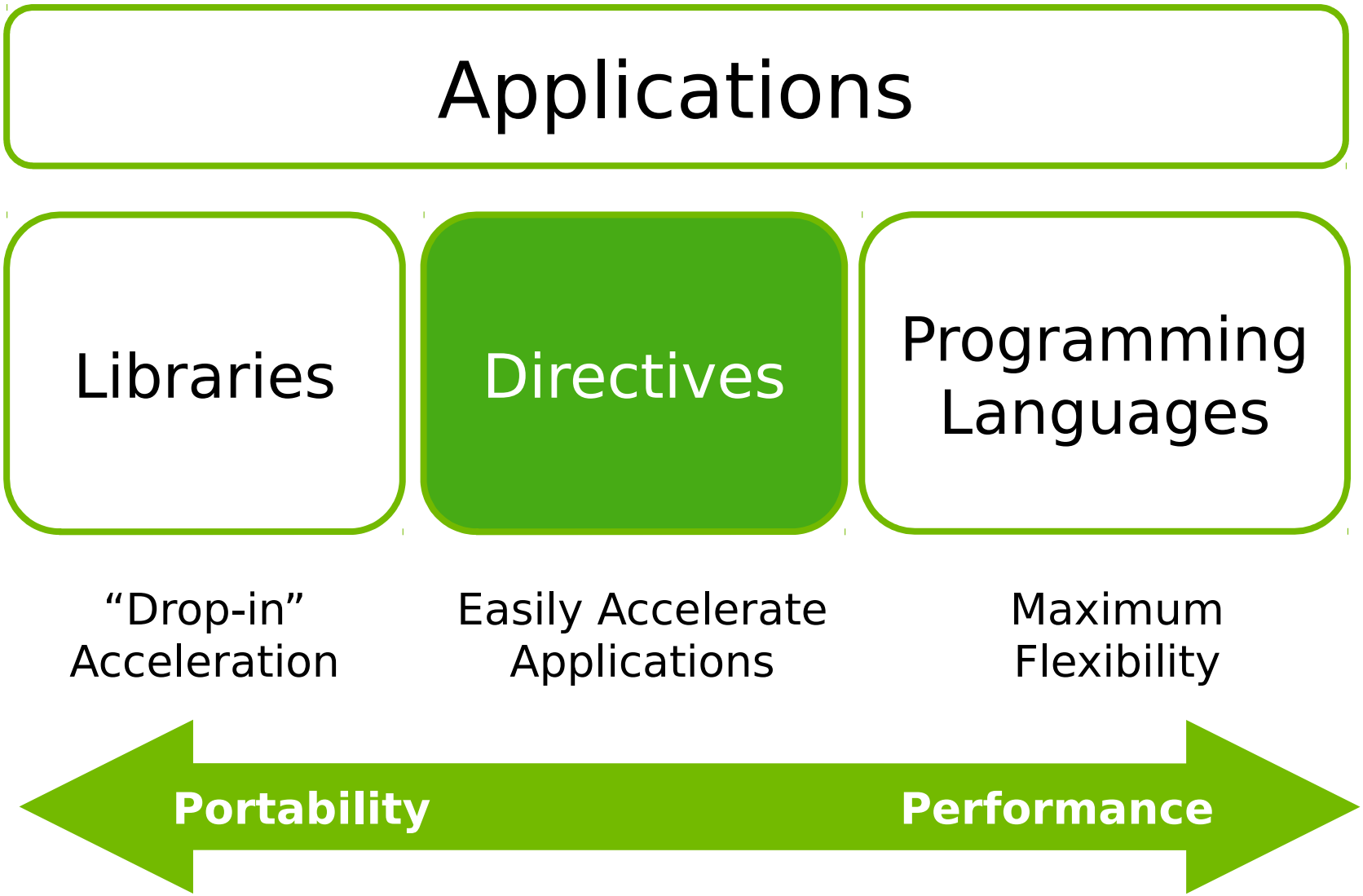
open source initiative

Thrust

C++ STL Features for CUDA

open source initiative

# 3 Ways to Accelerate Applications



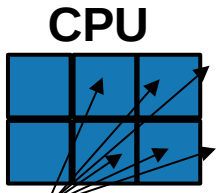
# Directive Based Approach

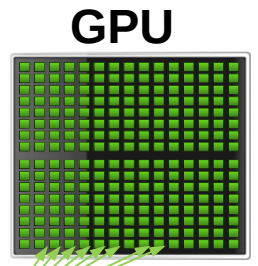
- Directives are added to serial source code
  - Manage loop parallelization
  - Manage data transfer between CPU and GPU memory
- Directives are formatted as comments
  - They don't interfere with serial execution
  - Maintaines portability of original code
- Works with C/C++ or Fortran
- Can be combined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

# OpenACC

- OpenACC is a high-level specification with compiler directives for expressing parallelism for accelerators.
  - Portable to a wide range of accelerators.
  - One specification for Multiple Vendors and Multiple Devices
- OpenACC specification was released in November 2011.
  - Original members: CAPS, Cray, NVIDIA, Portland Group
- OpenACC 2.0 was released in June 2013
  - More functionality
  - Improve portability
- OpenACC 2.5 in November 2015
- OpenACC had more than 10 member organizations

# Familiar to OpenMP Programmers




## 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);
}
```

# OpenMP 4.x

- Starting from OpenMP 4.x begins support for accelerator offload of high-parallel regions
- data movement and placement was refined with OpenMP v4.5
- OpenMP accelerator suites very well GPUs and efficient implementation runs on NVIDIA and AMD GPUs
- “OpenMP will be at a point where everything OpenACC has will have a direct analogy in OpenMP other than the kernels directive, which will probably never be in OpenMP.” *Bronis de Supinski - CTO and chair of the OpenMP language committee*

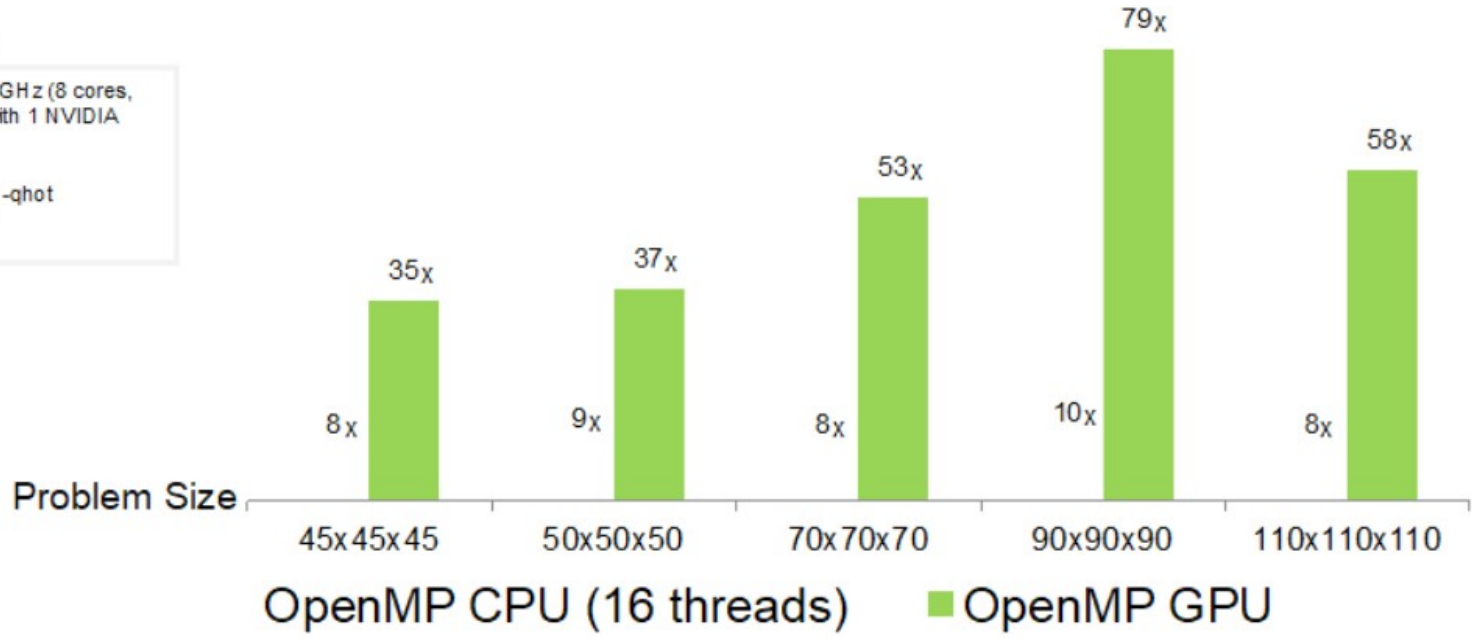
# OpenMP v4.5 Benchmark on GPU

- LULESH represents a typical hydrocode, like ALE3D. LULESH approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh. A node on the mesh is a point where mesh lines intersect. LULESH is built on the concept of an unstructured hex mesh. Instead, indirection arrays that define mesh relationships are used. The default test case for LULESH appears to be a regular cartesian mesh, but this is for simplicity only – it is important to retain the unstructured data structures as they are representative of what a more complex geometry will require.

**Test Specs**

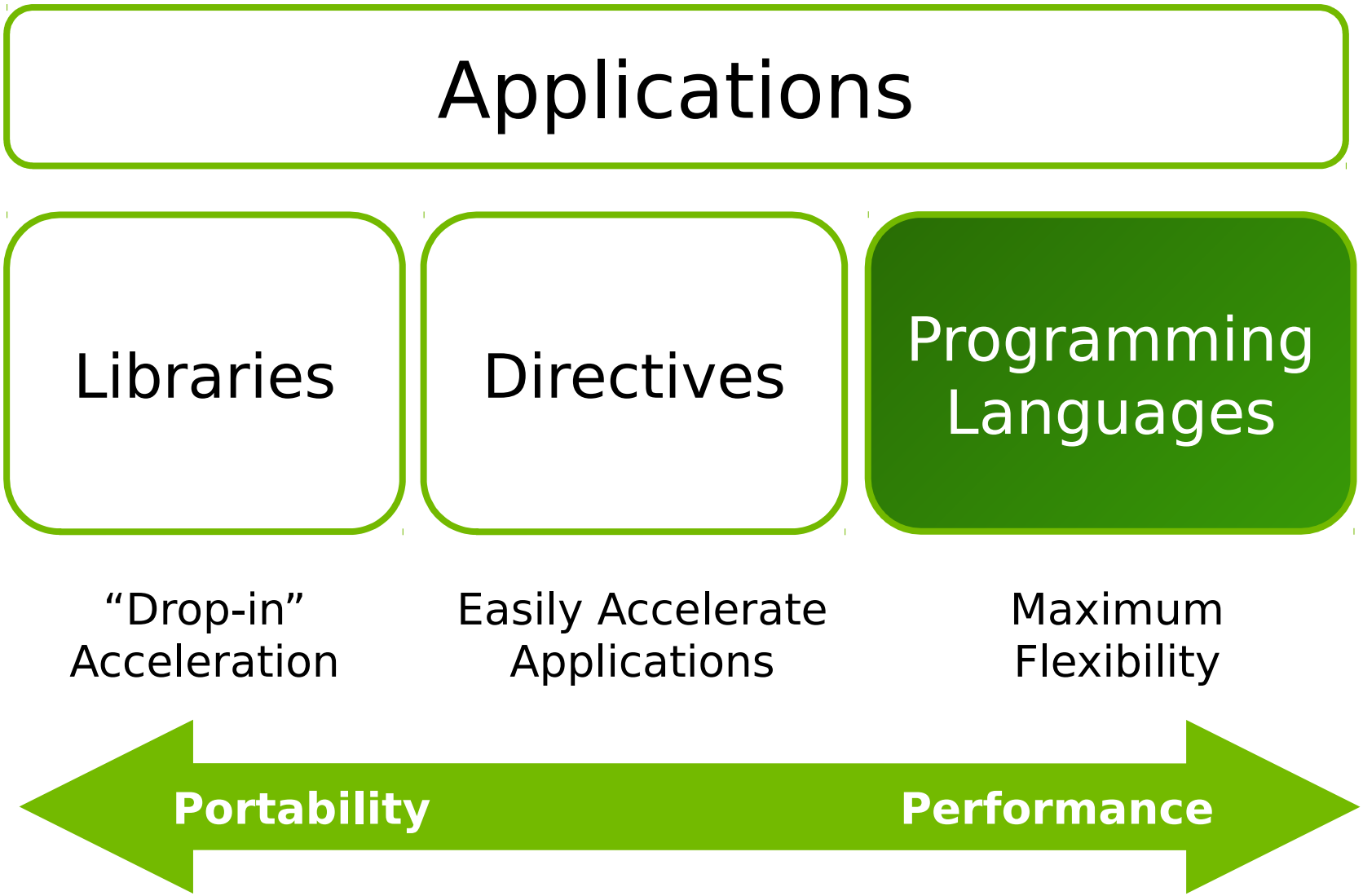
2 Power8 sockets @ 4GHz (8 cores, with 8 threads each) with 1 NVIDIA Pascal P100 GPU.

Compiler Options: -O3 -qhot -qsmp=omp -qoffload*
* Where applicable




https://codesign.llnl.gov/lulesh.php

# 3 Ways to Accelerate Applications



# GPU Programming Languages

- nVIDIA 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

# GPU Programming Languages

| | |
|---|---|
| **Fortran** | OpenMP, OpenACC, CUDA Fortran |
| **C** | OpenMP, OpenACC, CUDA C, OpenCL |
| **C++** | Thrust, CUDA C++, OpenCL |
| **Python** | PyCUDA, Copperhead |
| **Numerical analytics** | MATLAB, Mathematica, LabVIEW |

# Rights & Credits



Slides and examples were authored by:

Isabella Baccarelli, Luca Ferraro, Sergio Orlandini