Overview on GPU Computing

Silvio Stanzani , Raphael Cóbe and Jefferson Fialho

UNESP – Center for Scientific Computing

silvio.stanzani@sprace.org.br, raphael.cobe@sprace.org.br, jefferso
n.fialho@sprace.org.br

Agenda

- Heterogeneous Architectures
- GPU Architecture
- Programming Model
- Libraries
- OpenACC
- Cuda

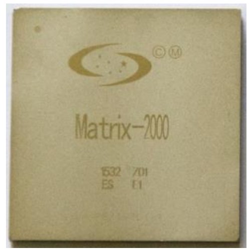
Heterogeneous Architectures

- GPU
  - It is a parallel/multithreaded multiprocessor optimized for visual computing.
    - ❑ Graphic processing is a **massively parallel application**
- GPGPU
  - General Purpose computation using GPU
- GPU serves as both a programmable graphics processor and a scalable parallel computing platform.
- Systems can combine CPU+GPU to execute applications

Heterogeneous Architectures

Host: CPU Processors

Device: Coprocessors or Accelerators

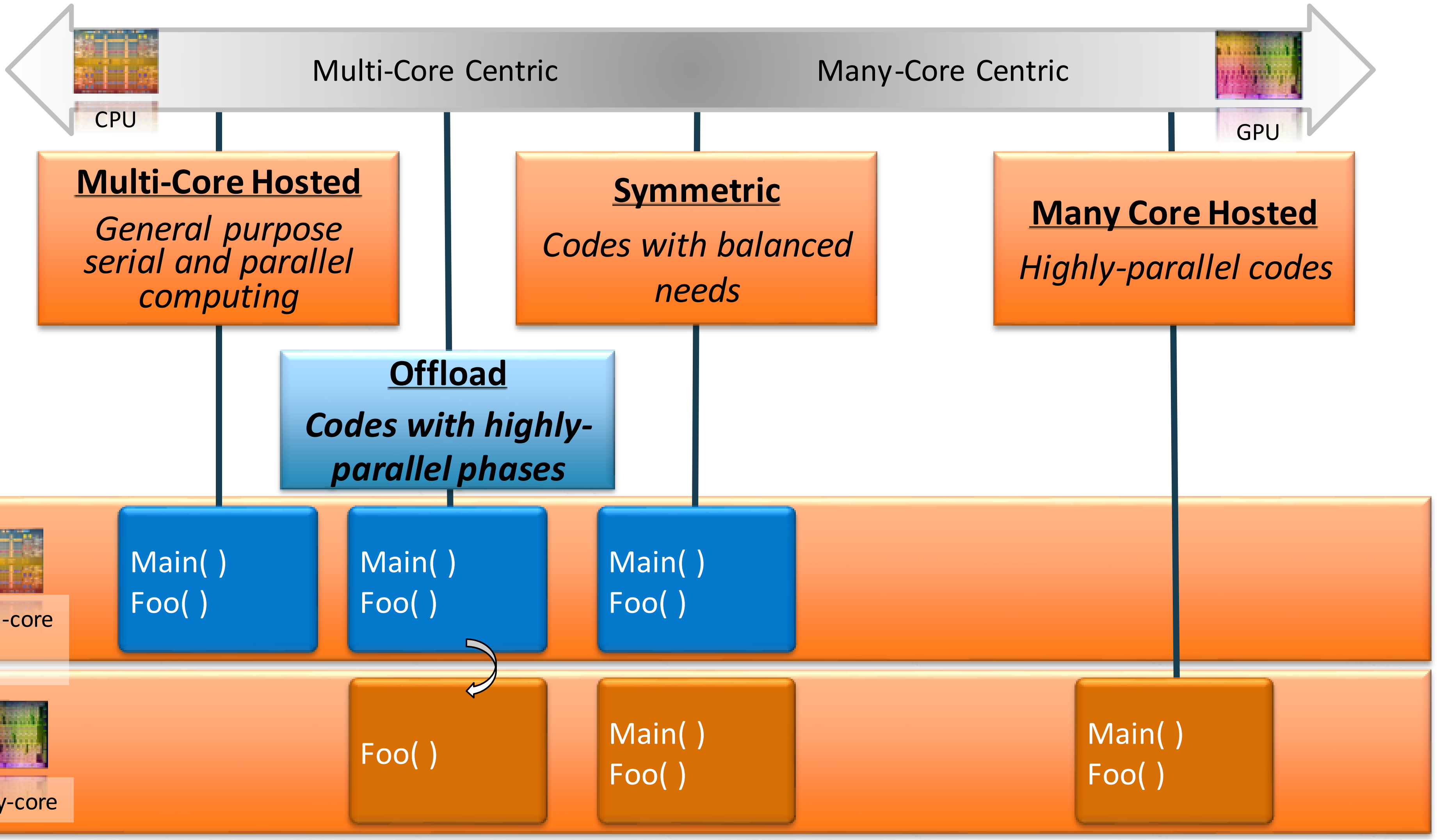


Host

Device (Network)

Device (PCI)

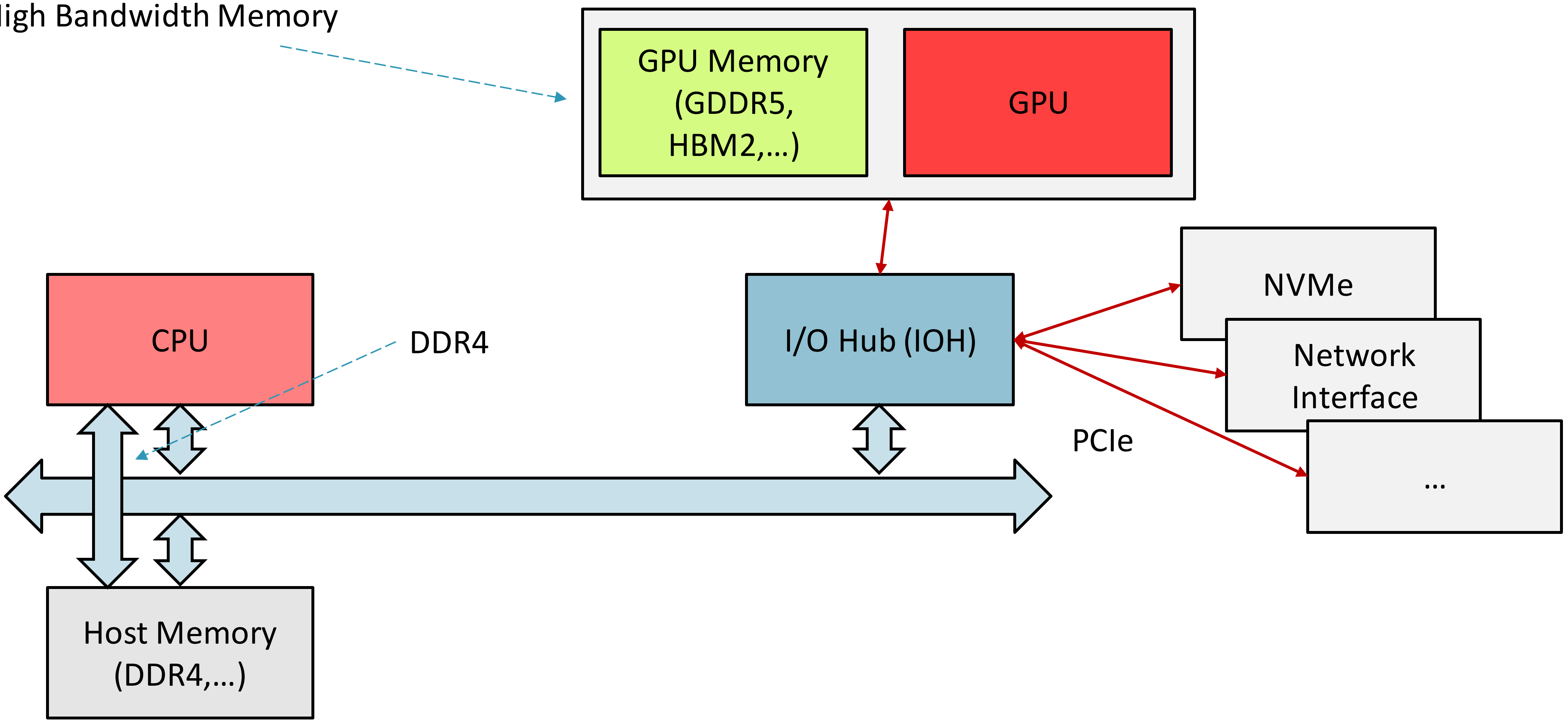
Heterogeneous Architectures



Range of models to meet application needs

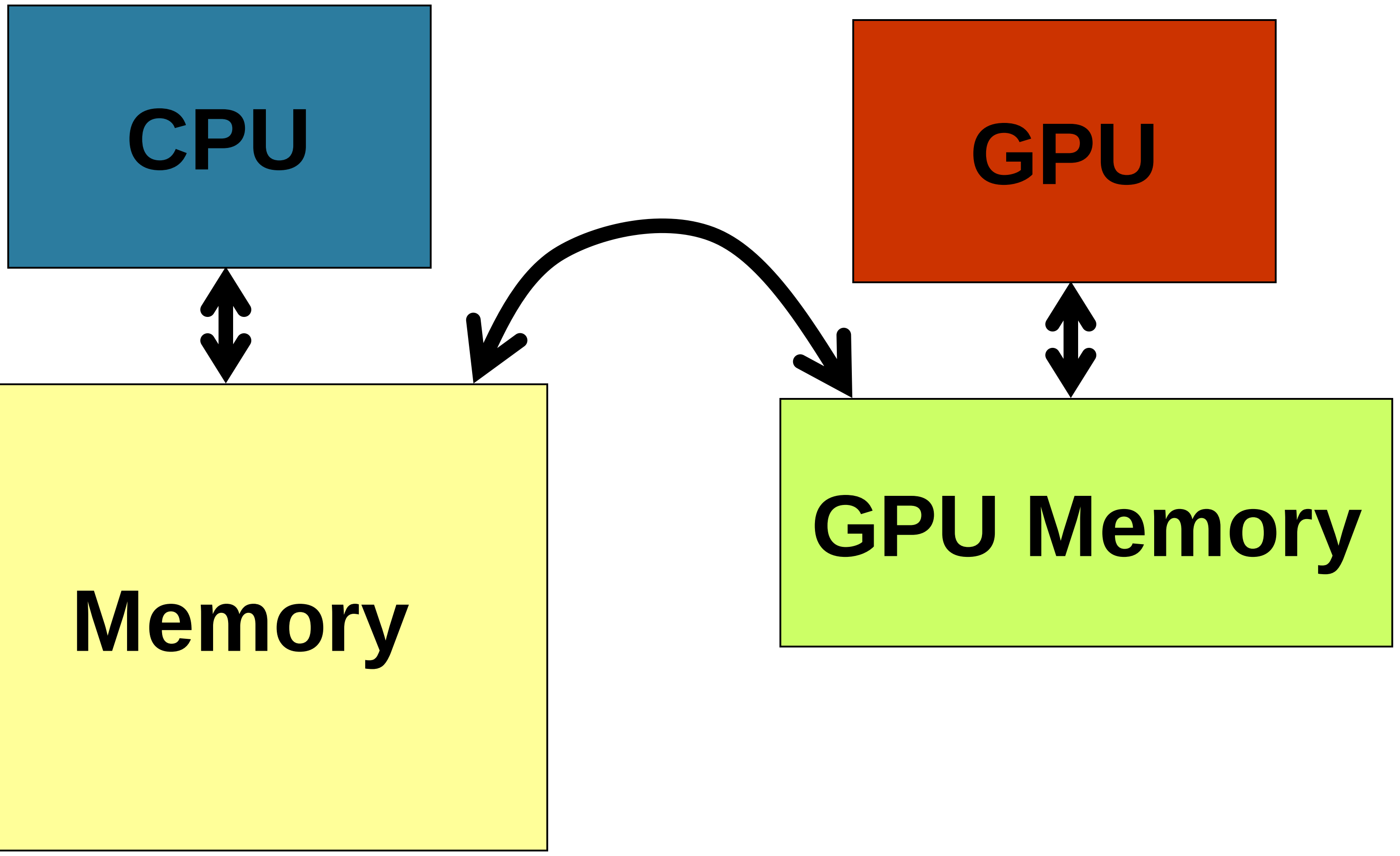
GPU Architecture

GDDR5: Graphics DDR
HBM: High Bandwidth Memory

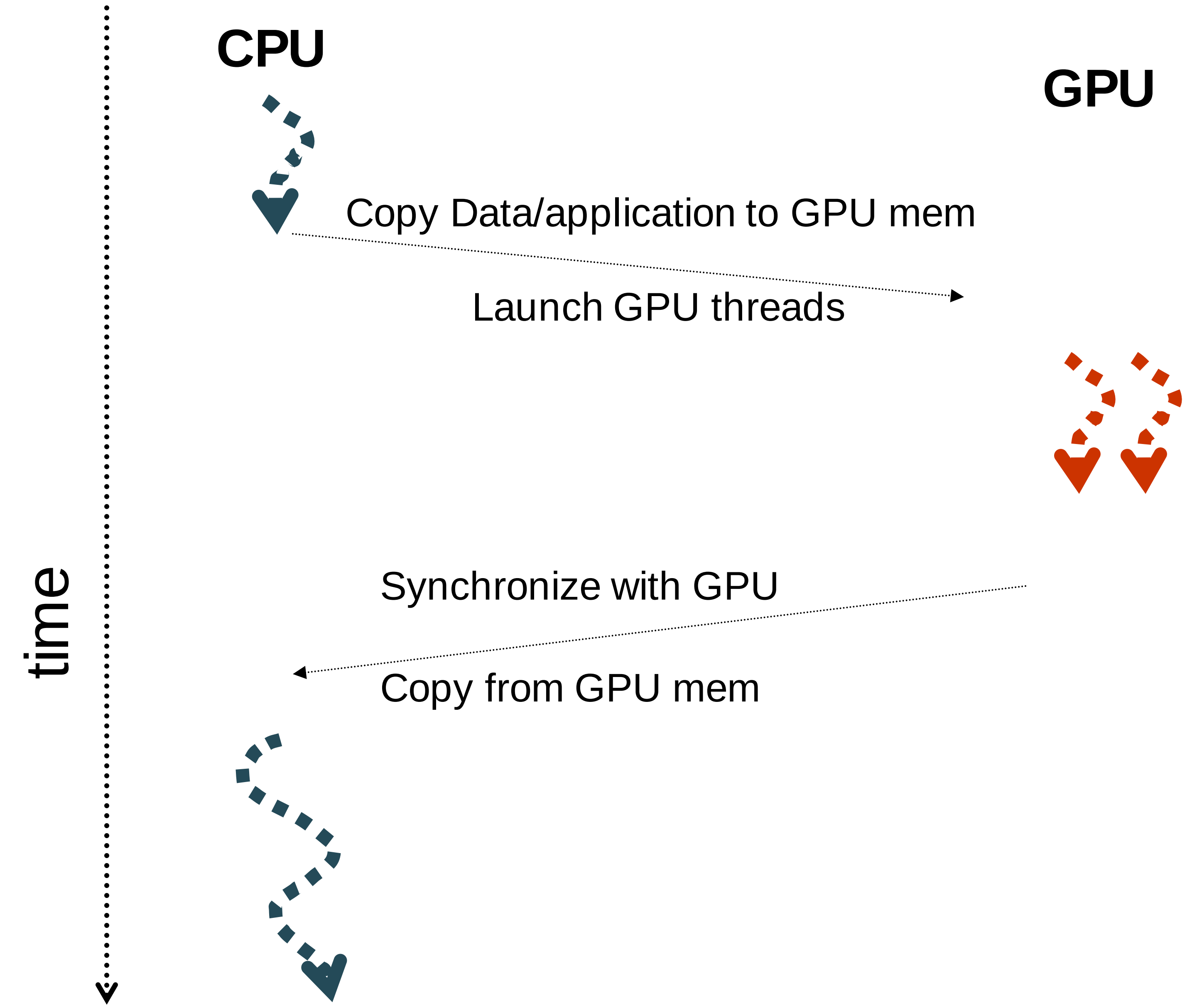


GPU Architecture

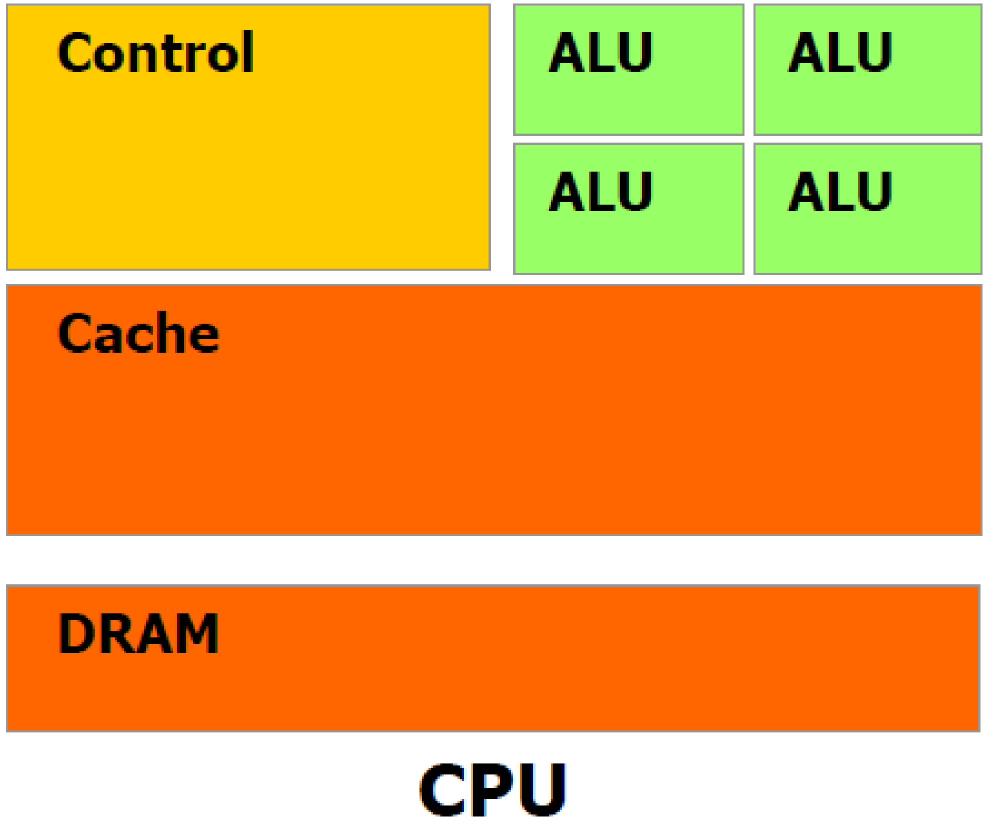
- Independent Memory Units
- Transfer overhead between CPU/GPU memory must compesate performance gains

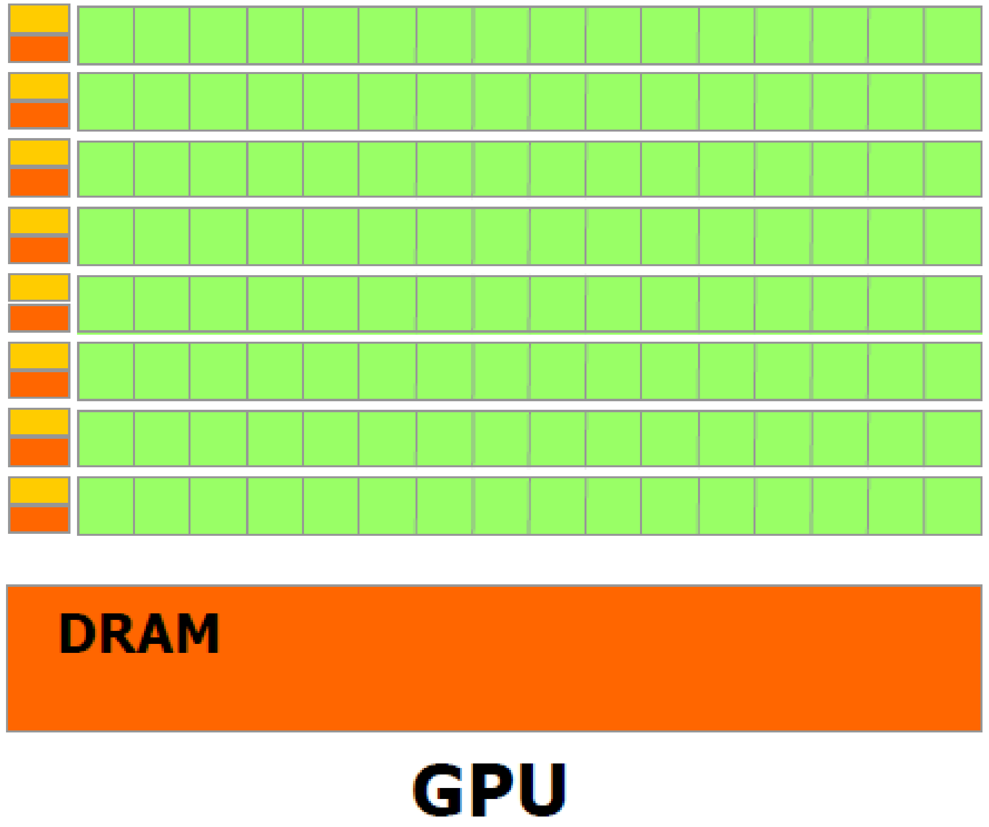


GPU Architecture



GPU Architecture

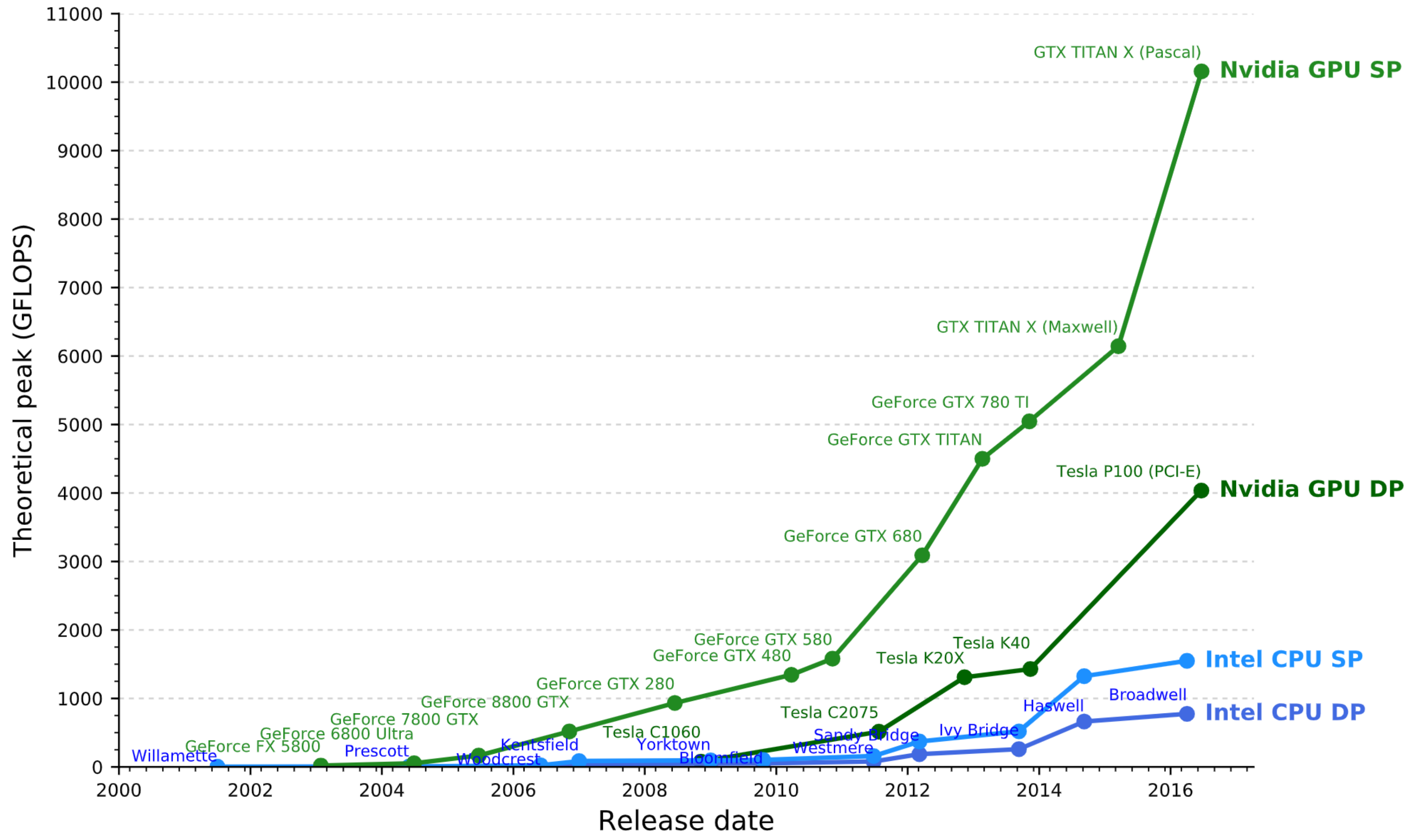




Cache Coherence

Data Locality

GPU Architecture



GPU Architecture

- Simplified logic (no out-of-order execution, no branch prediction) means much more of the chip is devoted to floating-point computation
  - CPU Core ≠ GPU Core

- Arranged as multiple units with each unit being effectively a vector unit, all cores doing the same thing at the same time
  - **Kernel**: a parallel routine to run on the parallel hardware

- Higher memory bandwidth than CPU

- Not general purpose
  - Massively parallel applications
    - ❑ Graphic processing
  - Applications that exploit memory locality
    - ❑ Each parallel unit perform access to its own subset of data
  - Data parallel algorithms leverage GPU attributes
    - ❑ Large data arrays, streaming throughput
    - ❑ Low-latency floating point (FP) computation

GPU Architecture

- SIMD: A single sequential instruction stream of SIMD instructions
  - each instruction specifies multiple data inputs
    - ❑ [VLD, VLD, VADD, VST], VLEN

- SIMT: Multiple instruction streams of scalar instructions
  - threads grouped dynamically
    - ❑ [LD, LD, ADD, ST], NumThreads
    - ❑ Data within Thread Group cannot be acessed by threads from other groups
  - Can treat each thread separately
    - ❑ Execute each thread independently (on any type of scalar pipeline)

Programming Model

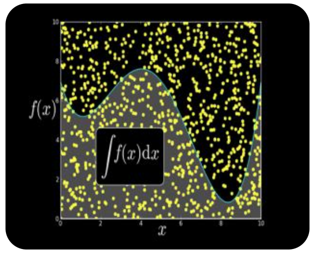
- How to use GPU resources
  - Libraries
    - ❑ Cublas
    - ❑ Tensorflow
  - Language extensions (directives)
    - ❑ OpenMP, OpenACC, OpenCL
    - ❑ Easy to accelerate code
    - ❑ Minimum flexibility
  - Programming Languange
    - ❑ Cuda API
    - ❑ Maximum flexibility
    - ❑ Low Level access

Programming Model

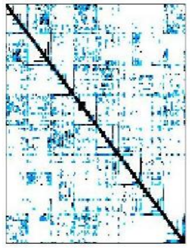
- Using libraries enables GPU acceleration without in-depth knowledge of GPU programming
- Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes
- Quality: Libraries offer high-quality implementations of functions encountered in a broad range of applications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

Programming Model

NVIDIA cuBLAS



NVIDIA cuRAND



NVIDIA cuSPARSE

NVIDIA NPP

GPU VSIPL

Vector Signal Image Processing

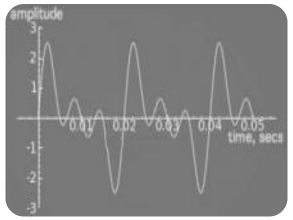
CULA tools

GPU Accelerated Linear Algebra

MAGMA

Matrix Algebra on GPU and Multicore

open source initiative



NVIDIA cuFFT

ROGUE WAVE SOFTWARE

IMSL Library

ArrayFire Matrix Computations

CUSP

Sparse Linear Algebra

Thrust

C++ STL Features for CUDA

Programming Model

- **Step 1:** Substitute library calls with equivalent CUDA library calls
- `saxpy ( … )` `cublasSaxpy ( … )`

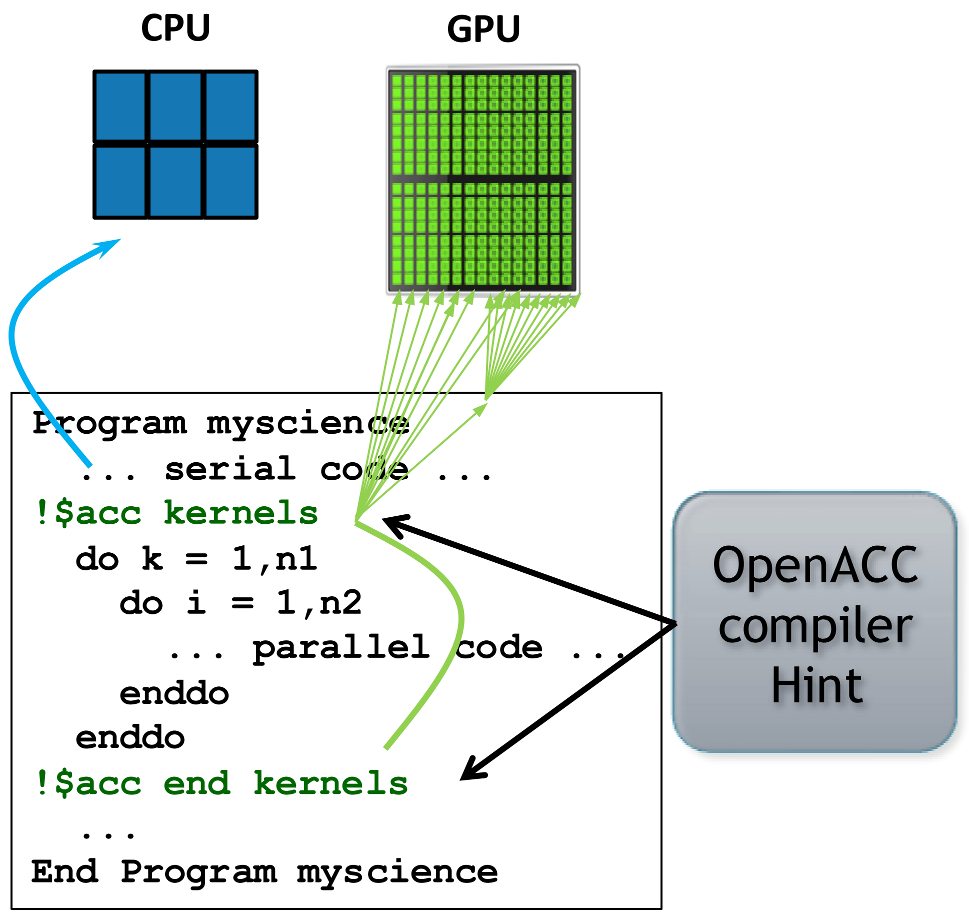
- **Step 2:** Manage data locality
- - with CUDA: `cudaMalloc(), cudaMemcpy(), etc.`
  - with CUBLAS: `cublasAlloc(), cublasSetVector(), etc.`

- **Step 3:** Rebuild and link the CUDA-accelerated library
  - ❑ `nvcc myobj.o –l cub`

Programming Model

OpenACC

- Directives are the easy path to accelerate compute intensive applications
- OpenACC is an open GPU directives standard,making GPU programming straightforward and portable across parallel and multi-core processors
- PGI compiler supports OpenACC
  - Professional version ($) and community version



```
Program myscience
   ... serial code ...
!$acc kernels
  do k = 1,n1
    do i = 1,n2
      ... parallel code ...
    enddo
  enddo
!$acc end kernels
   ...
End Program myscience
```

Your original Fortran or C code

OpenACC

We request that each loop execute as a separate *kernel* on the GPU.

```
!$acc kernels
    do i=1,n
       a(i) = 0.0
       b(i) = 1.0
       c(i) = 2.0
    end do

    do i=1,n
       a(i) = b(i) + c(i)
    end do
!$acc end kernels
```

$\left.\begin{matrix}\\ \\ \\ \\ \end{matrix}\right\}$ **kernel 1**

$\left.\begin{matrix}\\ \\ \end{matrix}\right\}$ **kernel 2**

OpenACC

- general directive syntax and scope

C

```
#pragma acc kernels [clause …]
```

Fortran

```
!$acc kernels [clause …]
    structured block
!$acc end kernels
```

- Compile and run:
  - C: pgcc –acc saxpy.c
  - Fortran: pgf90 –acc saxpy.f90
  - Run: ./a.out

OpenACC

- general directive syntax and scope

C

```
#pragma acc kernels [clause …]
```

Fortran

```
!$acc kernels [clause …]
    structured block
!$acc end kernels
```

- Compile and run:
  - C: pgcc –acc saxpy.c
  - Fortran: pgf90 –acc saxpy.f90
  - Run: ./a.out

OpenACC

- **pragma acc kernel** : indicates to the compiler a loop that can be automatically parallelized

- **pragma acc parallel loop** : ensure that the loop can be parallelized

- **Data transfer:**
  - **copyin(list)** - Allocates memory on GPU and copies data from host to GPU when entering region.
  - **copyout(list)** - Allocates memory on GPU and copies data to the host when exiting region.

- In some cases Kernel automatically implements CPU <-> GPU transfer

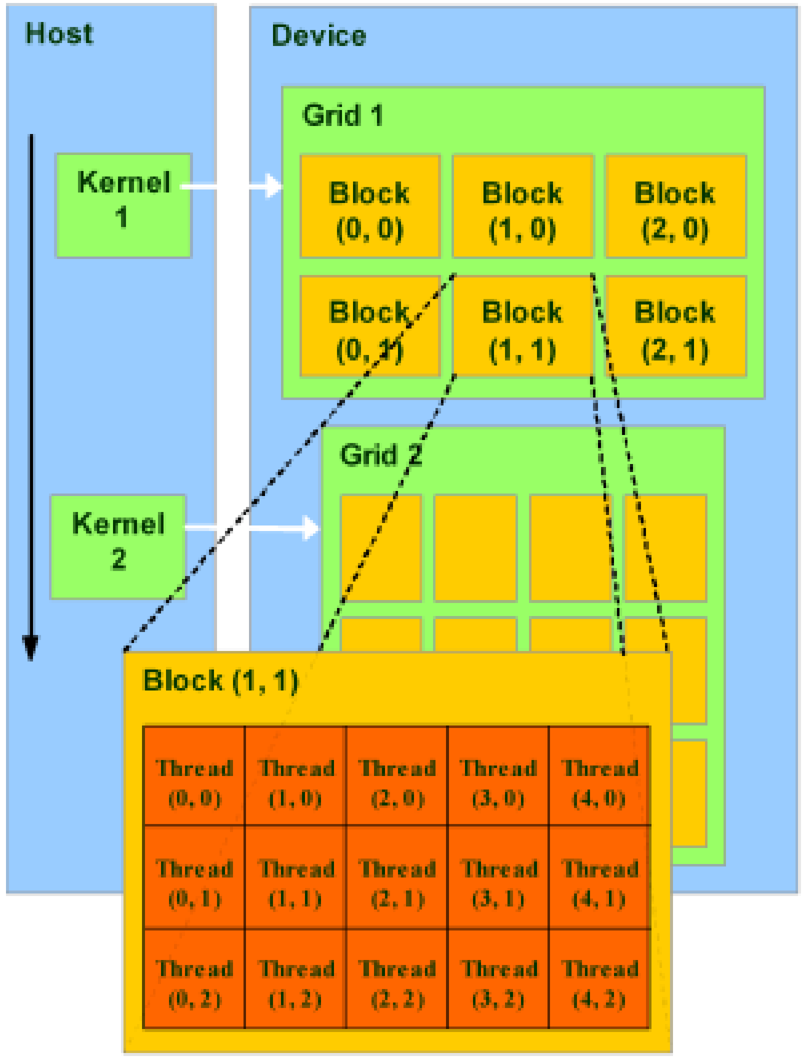
OpenACC

Matrix Multiplication Example

```
#pragma acc kernels copyin(A[:SIZE*SIZE],B[:SIZE*SIZE]), copyout(C[:SIZE*SIZE])
{
  for (row=0; row<row_len; row++) {
   for (col=0; col<col_len; col++) {
    for (i=0; i<SIZE; i++)
     C[row*row_len+col] += A[row*row_len+i] * B[col+i*row_len];
   }
  }
}
```

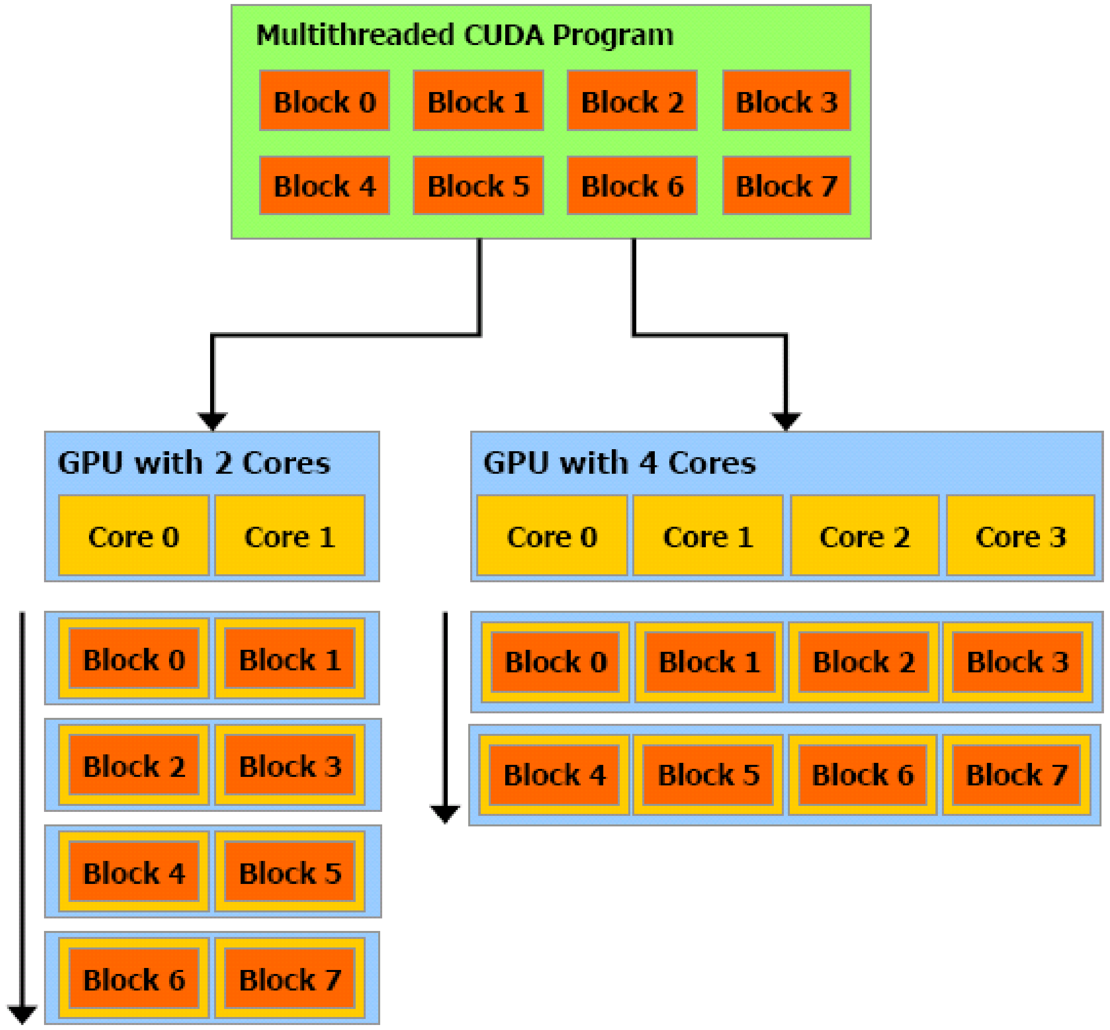
Cuda

- CUDA: Compute Unified Device Architecture
- Low Level API to GPU
- Scale code to hundreds of cores running thousands of threads
  - Threads are grouped into thread blocks
  - Blocks are grouped into a single grid
  - The grid is executed on the GPU as a kernel



Cuda

- Blocks map to cores on the GPU
- Allows for portability when changing hardware



Cuda

- Terms and Concepts
  - Each block and thread has a unique id within a block.
    - threadIdx – identifier for a thread
    - blockIdx – identifier for a block
    - blockDim – size of the block
  - Unique thread id:
    - (blockIdx*blockDim)+threadIdx

- Development: Basic Idea
  - Allocate equal size of memory for both host and device
    - Transfer data from host to device
  - Execute kernel to compute on data
    - Transfer data back to host

Cuda

- Kernel Function Qualifiers
  - __global__ Runs on the GPU, called from the CPU.
  - C program:
    - void increment_cpu(float *a, float b, int N)
  - CUDA program
    - **__global__** void increment_gpu(float *a, float b, int N)

Cuda

- __device__
  - Stored in global memory (large, high latency, no cache)
  - Allocated with cudaMalloc
  - Accessible by all threads
  - Lifetime: application

- __shared__
  - Stored in on-chip shared memory (very low latency)
  - Specified by execution configuration or at compile time
  - Accessible by all threads in the same thread block
  - Lifetime: thread block

Cuda

- Calling a kernel function is different from calling a regular function

```
Void main(){
    Int blocks = 256;
    Int threadsperblock = 512;
    mycudafunc<<<blocks,threadsperblock>>>(some parameter);
}
```

Cuda

- **GPU Memory Allocation / Release**
- Host (CPU) manages GPU memory:
  - cudaMalloc (void ** pointer, size_t nbytes)
  - cudaMemset (void * pointer, int value, size_t count);
  - cudaFree (void* pointer)

```
Void main(){
    int n = 1024;
    int nbytes = 1024*sizeof(int);
    int * d_a = 0;
    cudaMalloc( (void**)&d_a,  nbytes );
    cudaMemset( d_a, 0, nbytes);
    cudaFree(d_a);
}
```

Cuda

- Memory Transfer
  - cudaMemcpy(void *dst,   void *src,   size_t nbytes, enum cudaMemcpyKind direction);
    - ❑ returns after the copy is complete blocks CPU
    - ❑ thread doesn’t start copying until previous CUDA calls complete
  - enum cudaMemcpyKind
    - ❑ cudaMemcpyHostToDevice
    - ❑ cudaMemcpyDeviceToHost
    - ❑ cudaMemcpyDeviceToDevice

Cuda

```
void increment_cpu(float *a, float b, int
    N)
{
 for (int idx = 0; idx<N; idx++)
a[idx] = a[idx] + b;
}

void main()
{
...
increment_cpu(a, b, N);
}
```

```
__global__ void increment_gpu(float *a, float b,
int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if( idx < N)
        a[idx] = a[idx] + b;
}
void main() {
.....
dim3 dimBlock (blocksize);

dim3 dimGrid( ceil( N / (float) blocksize) )

increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```