DATA SCIENCE &
SCIENTIFIC COMPUTING

Lecture 16

introduction to GPU programming part 1

Stefano Cozzini
Area Science Park
14.12.2021/16.12.2021

Agenda

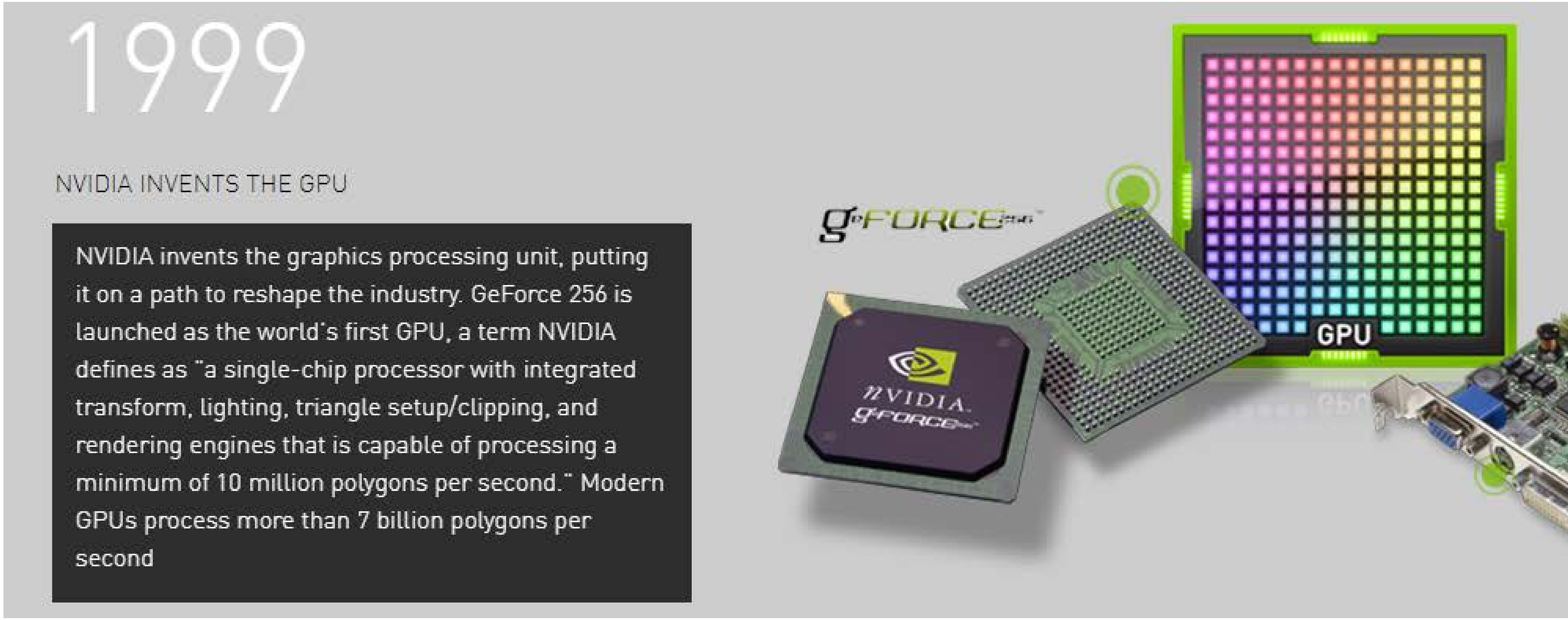
- Why GPU ? (a little bit of history)
- Basic of GPU architecture
- How to use/programming GPUs ?

DISCLAIMER: many of these slides are coming from Nvidia training materials

A little bit of history 1: the rising (90s..)

- The CPU has always been slow for Graphics Processing
  - Visualization
  - Games
- Graphics processing is inherently parallel and there is a lot of parallelism
  - O(pixels)
- GPUs were built to do graphics processing only
- Initially, hardwired logic replicated to provide parallelism
  - Little to no programmability

From Nvidia web site



1999

NVIDIA INVENTS THE GPU

NVIDIA invents the graphics processing unit, putting it on a path to reshape the industry. GeForce 256 is launched as the world's first GPU, a term NVIDIA defines as "a single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second." Modern GPUs process more than 7 billion polygons per second

To better read..

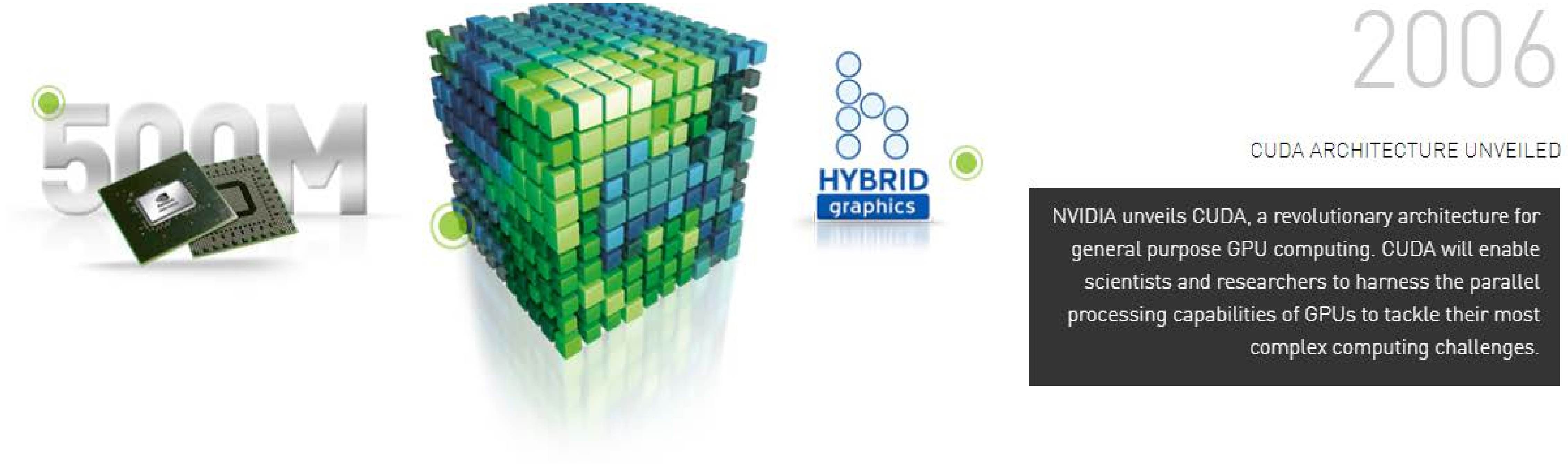
- Nvidia defined the term *graphics processing unit* as

“a single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second.”

A little bit of history 2: the 00’s

- Like CPUs, GPUs benefited from Moore's Law
  - Evolved from fixed-function hardwired logic to flexible, programmable ALUs
- Around 2003/2004, GPUs were programmable “enough” to do some non-graphics computations
- Severely limited by graphics programming model
- In 2006, GPUs became “fully” programmable: NVIDIA releases “CUDA” language to write non-graphics programs that will run on GPUs

From Nvidia web site



2006

CUDA ARCHITECTURE UNVEILED

NVIDIA unveils CUDA, a revolutionary architecture for general purpose GPU computing. CUDA will enable scientists and researchers to harness the parallel processing capabilities of GPUs to tackle their most complex computing challenges.

A little bit of history 3: the 10’s (the present)

- GPUs are widely deployed as accelerators
- GPUs so successful that other CPU alternatives are dead
  - Sony/IBM Cell BE
  - Clearspeed RSX
  - Intel MIC
- GPU enabled the ML/DL/AI revolution and started the HPC/AI convergence
  - Tensorcore
- There is ONE winner: NVIDIA

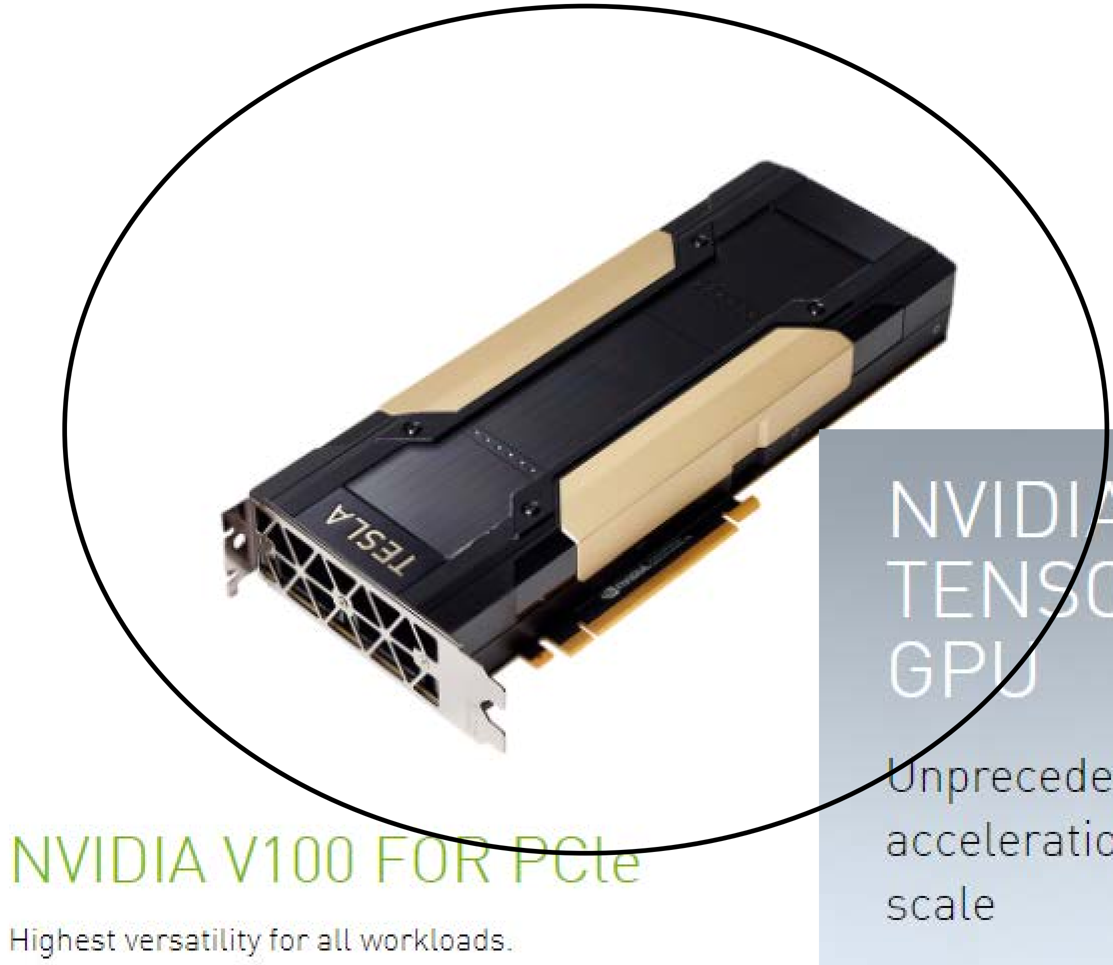
From Nvidia web site

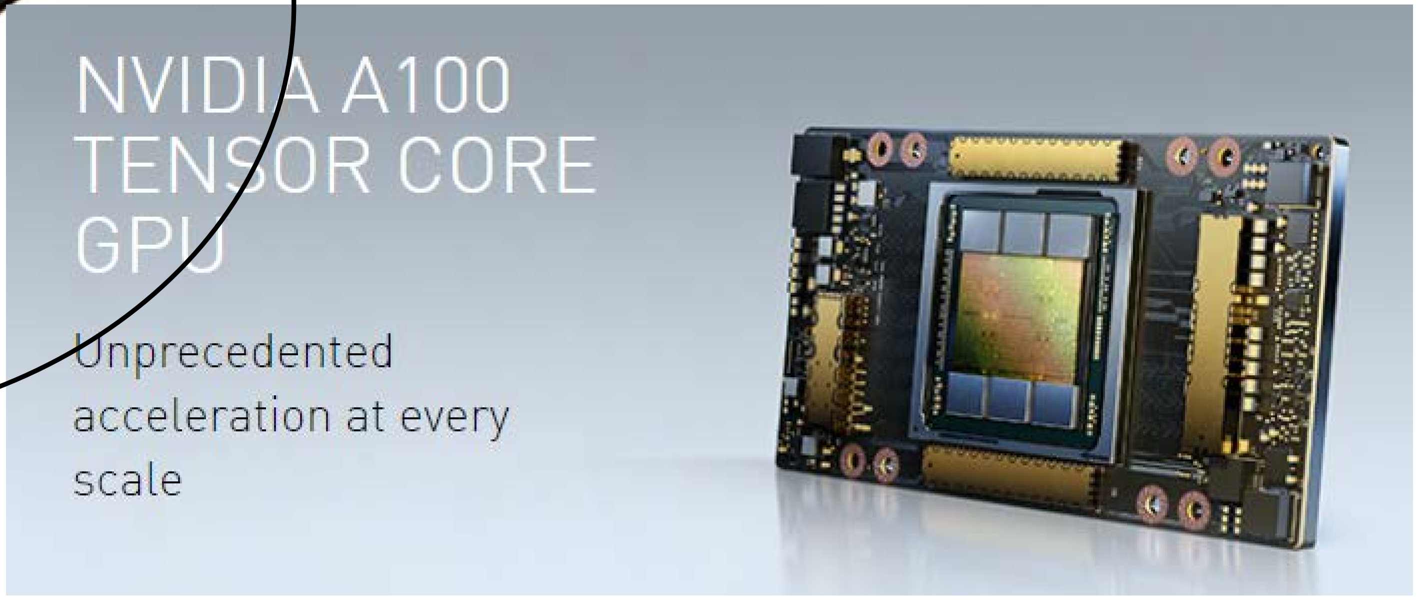
2019

NVIDIA ANNOUNCES ADVANCES ACROSS HPC, EMBEDDED, DATA CENTER, AUTONOMOUS VEHICLE, AND PRO GRAPHICS MARKETS

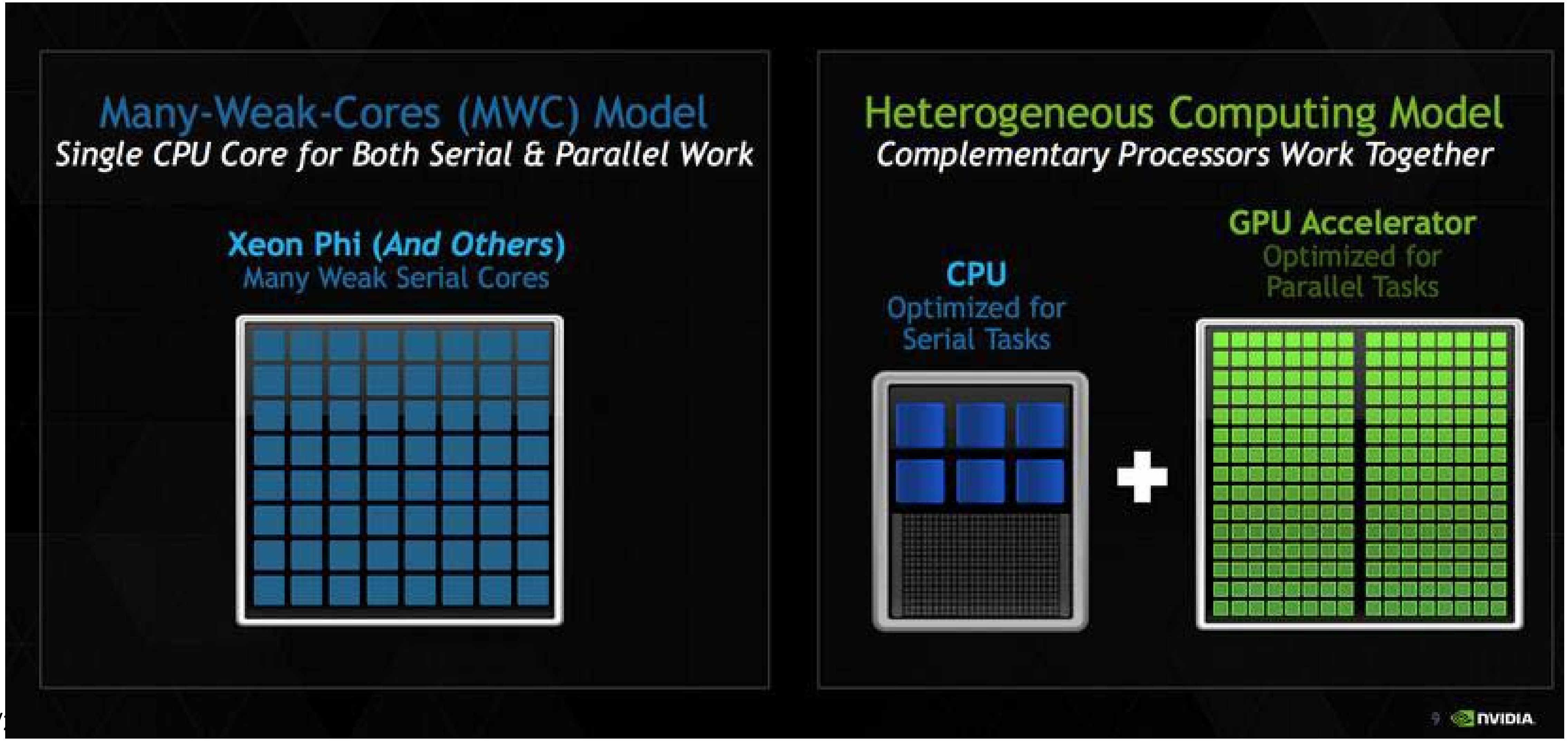
NVIDIA reference design platform is introduced, enabling companies to quickly build GPU-accelerated Arm®-based servers for a growing range of HPC applications.

Nvidia cards for HPC/AI



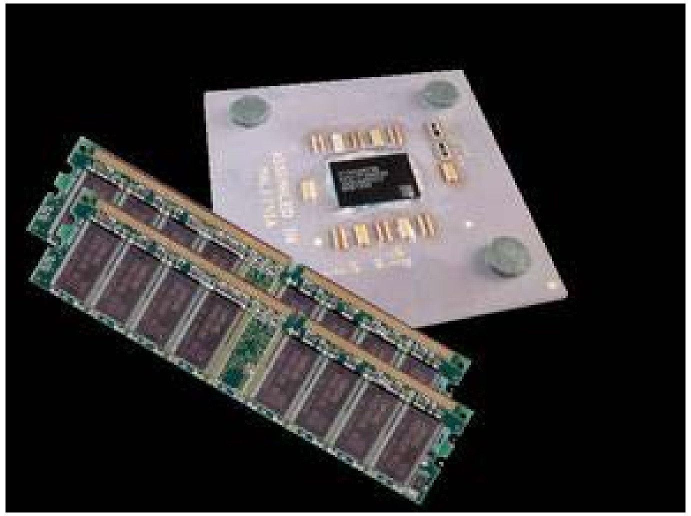

NVIDIA A100 TENSOR CORE GPU
Unprecedented acceleration at every scale

Nvidia slide: computing mode for accelerator

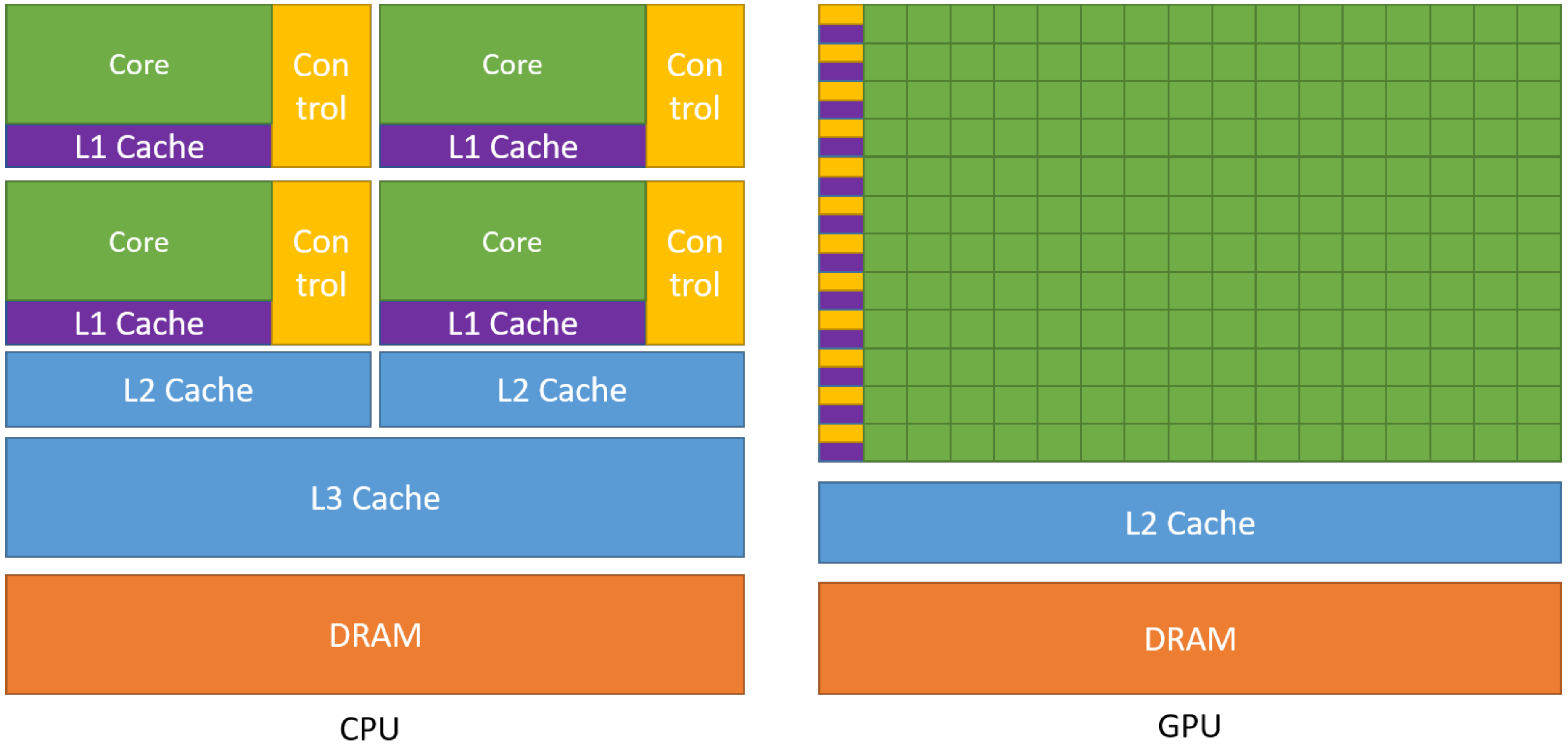

Many-Weak-Cores (MWC) Model
Single CPU Core for Both Serial & Parallel Work
Xeon Phi (And Others)
Many Weak Serial Cores
Heterogeneous Computing Model
Complementary Processors Work Together
CPU
Optimized for Serial Tasks
GPU Accelerator
Optimized for Parallel Tasks
9
NVIDIA

Heterogenous computing

- Terminology:
  - Host The CPU and its memory (host memory)
  - Device The GPU and its memory (device memory)



CPU vs GPU

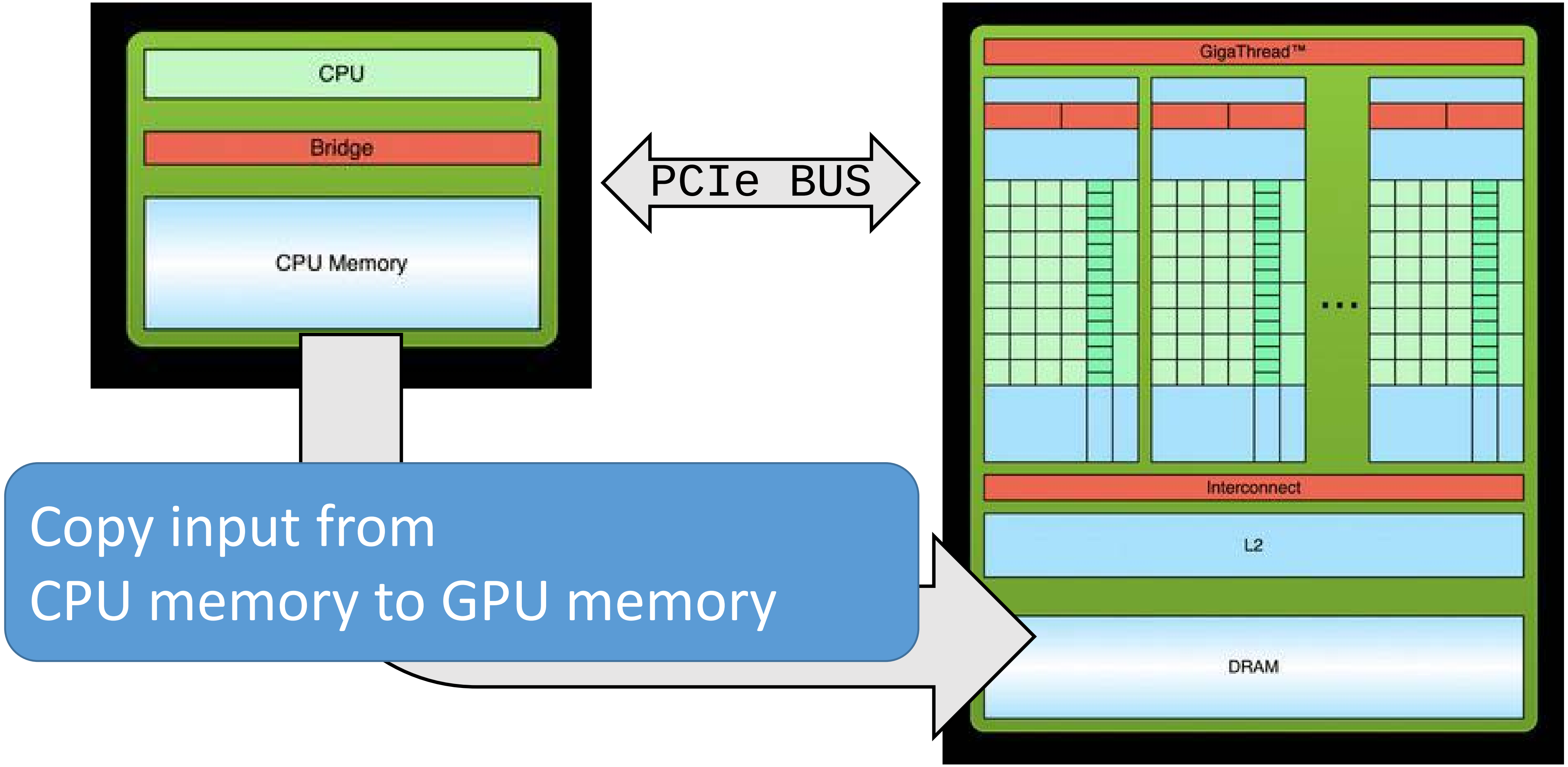

Core
Con trol
L1 Cache
Core
Con trol
L1 Cache
Core
Con trol
L1 Cache
Core
Con trol
L1 Cache
L2 Cache
L2 Cache
L3 Cache
DRAM
CPU
L2 Cache
DRAM
GPU

Nvidia cards for HPC/AI

Nvidia Datacenter GPU	Nvidia Tesla V100	Nvidia A100
GPU codename	GV100	GA100
GPU architecture	Volta	Ampere
Launch date	May 2017	May 2020
GPU process	TSMC 12nm	TSMC 7nm
Die size	815mm2	826mm2
Transistor Count	21.1 billion	54 billion
FP64 CUDA cores	2,560	3,456
FP32 CUDA cores	5,120	6,912
Tensor Cores	640	432
Streaming Multiprocessors	80	108
Peak FP64	7.8 teraflops	9.7 teraflops
Peak FP64 Tensor Core	-	19.5 teraflops
Peak FP32	15.7 teraflops	19.5 teraflops
Peak FP32 Tensor Core	-	156 teraflops/312 teraflops*
Peak BFLOAT16 Tensor Core	-	312 teraflops/624 teraflops*
Peak FP16 Tensor Core	-	312 teraflops/624 teraflops*
Peak INT8 Tensor Core	-	624 teraflops/1,248 TOPS*
Peak INT4 Tensor Core	-	1,248 TOPS/2,496 TOPS*
Mixed-precision Tensor Core	125 teraflops	312 teraflops/624 teraflops*
Max TDP	300 watts	400 watts

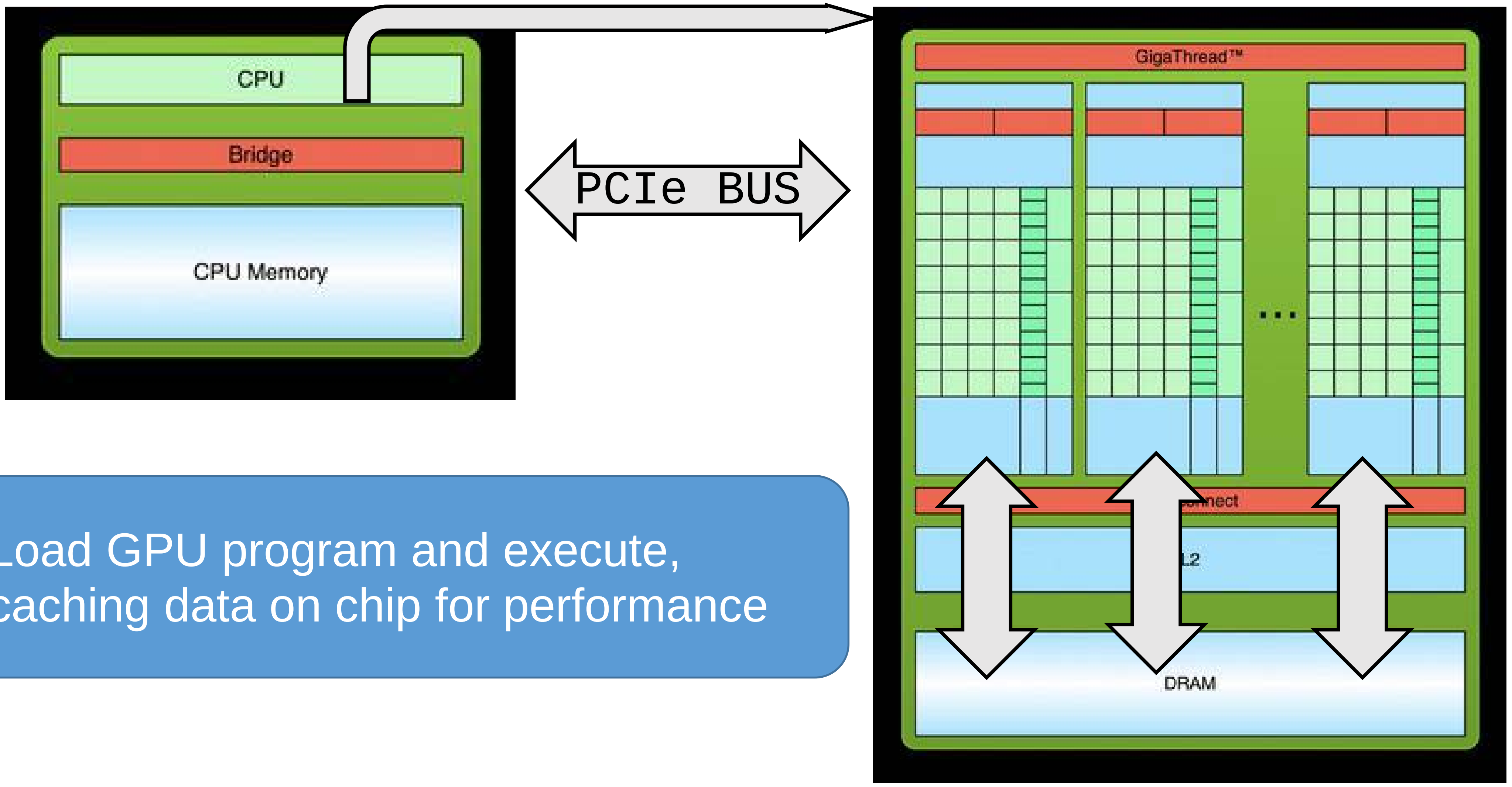
*Effective TOPS / TFLOPS using the new Sparsity feature

Simple process flow



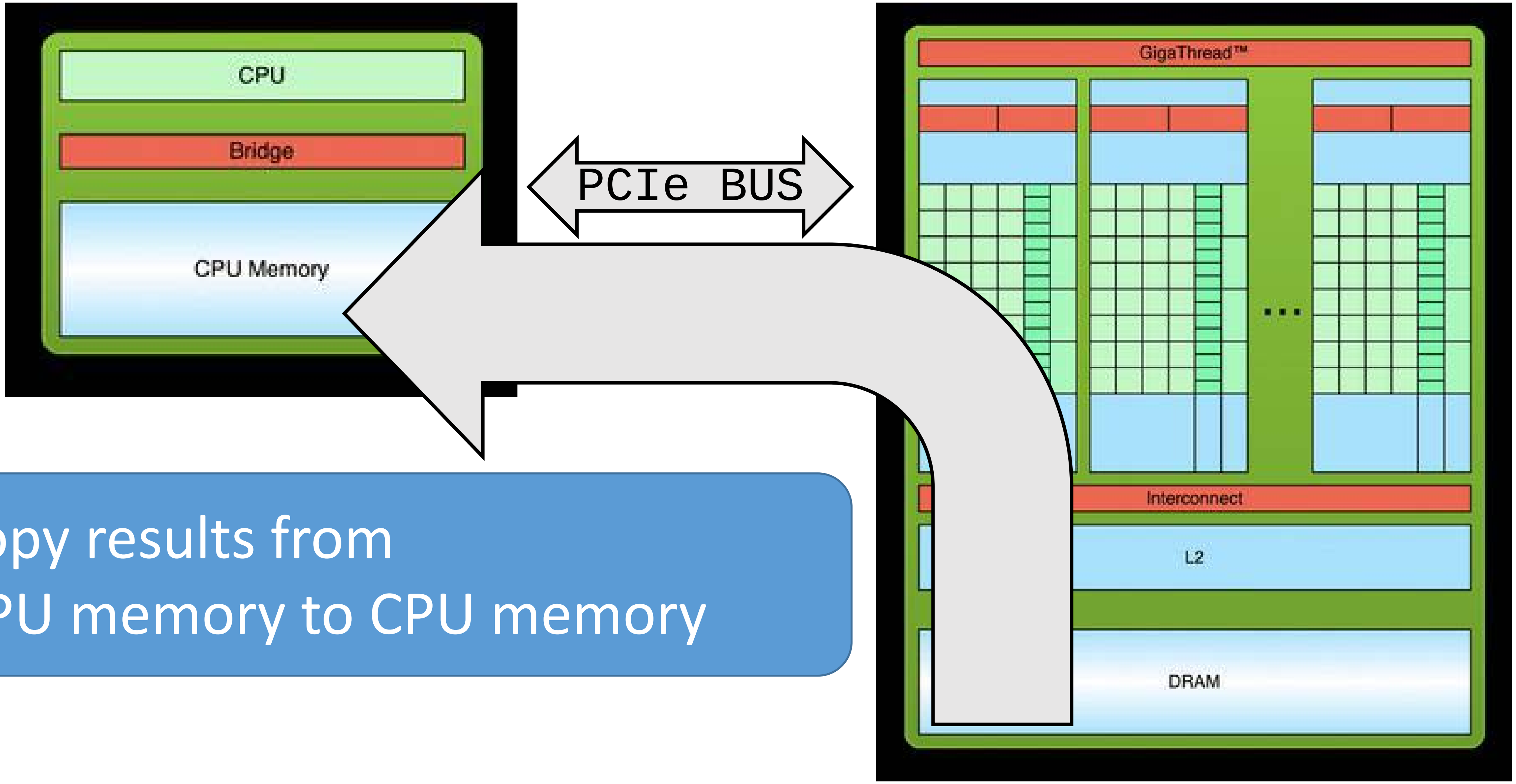
Copy input from
CPU memory to GPU memory

Simple process flow



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

Simple process flow



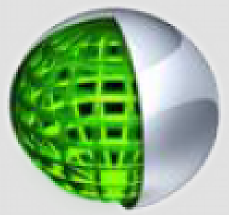
Copy results from
GPU memory to CPU memory

CUDA Parallel Computing Platform

Programming Approaches

Libraries	OpenACC Directives	Programming Languages
"Drop-in" Acceleration	**Easily Accelerate Apps**	**Maximum Flexibility**

Development Environment

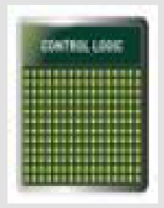


Nsight IDE
Linux, Mac and Windows
GPU Debugging and Profiling

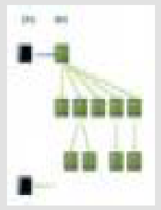
CUDA-GDB debugger
NVIDIA Visual Profiler

Hardware Capabilities

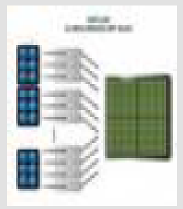
SMX



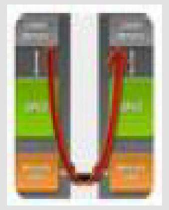
Dynamic Parallelism



HyperQ

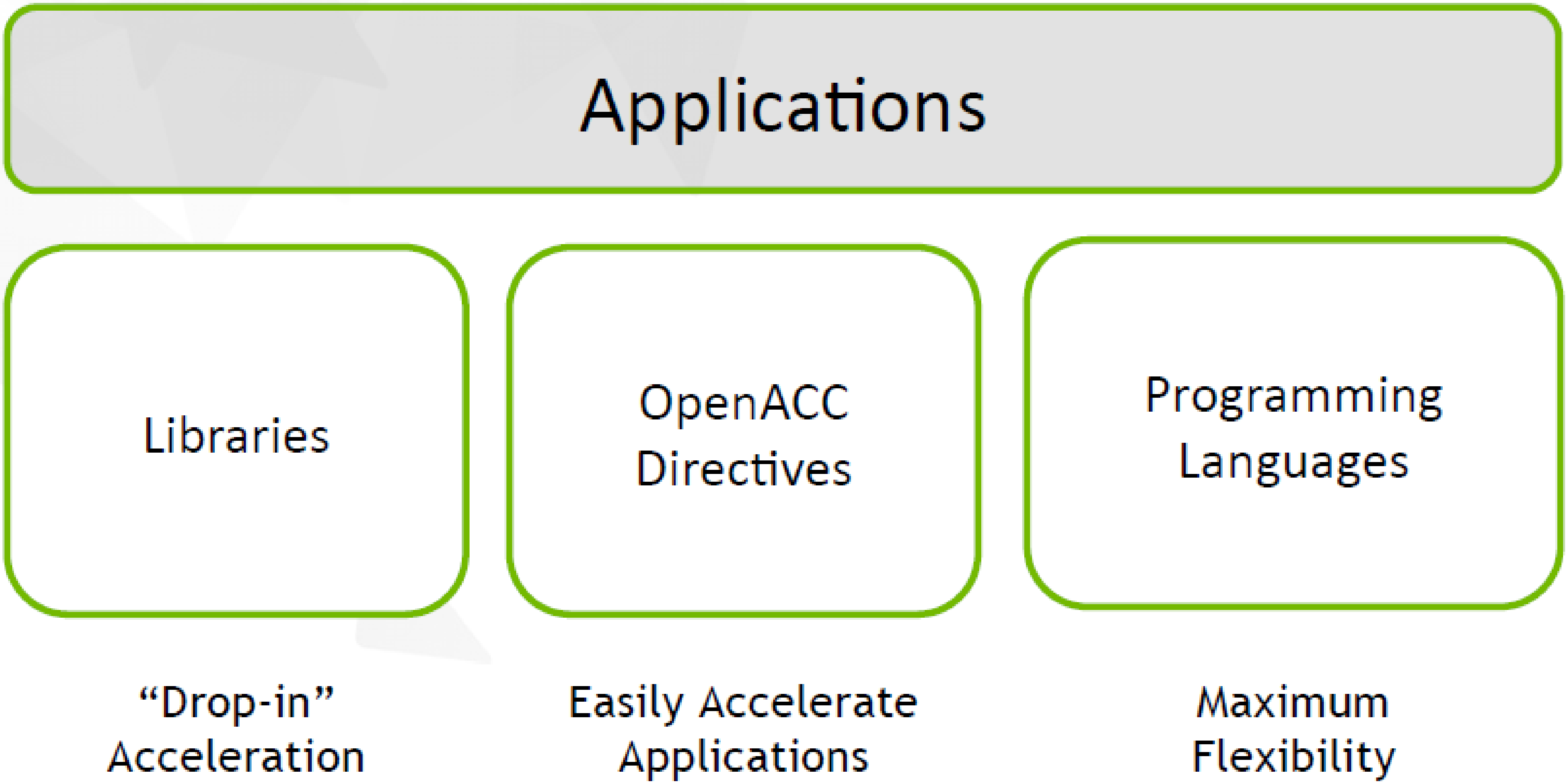


GPUDirect

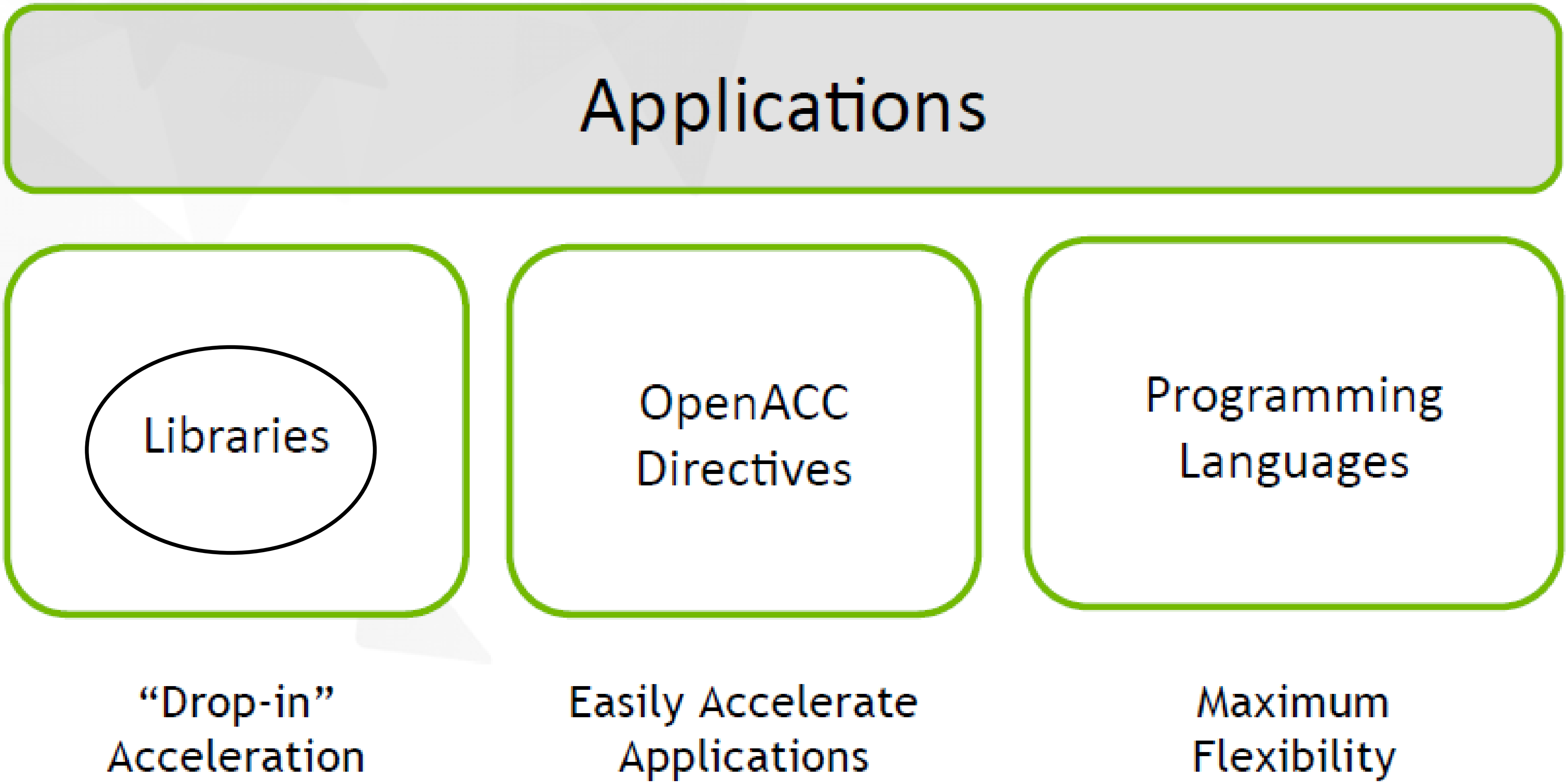


From: www.nvidia.com/getcuda

3 ways to accelerate your application



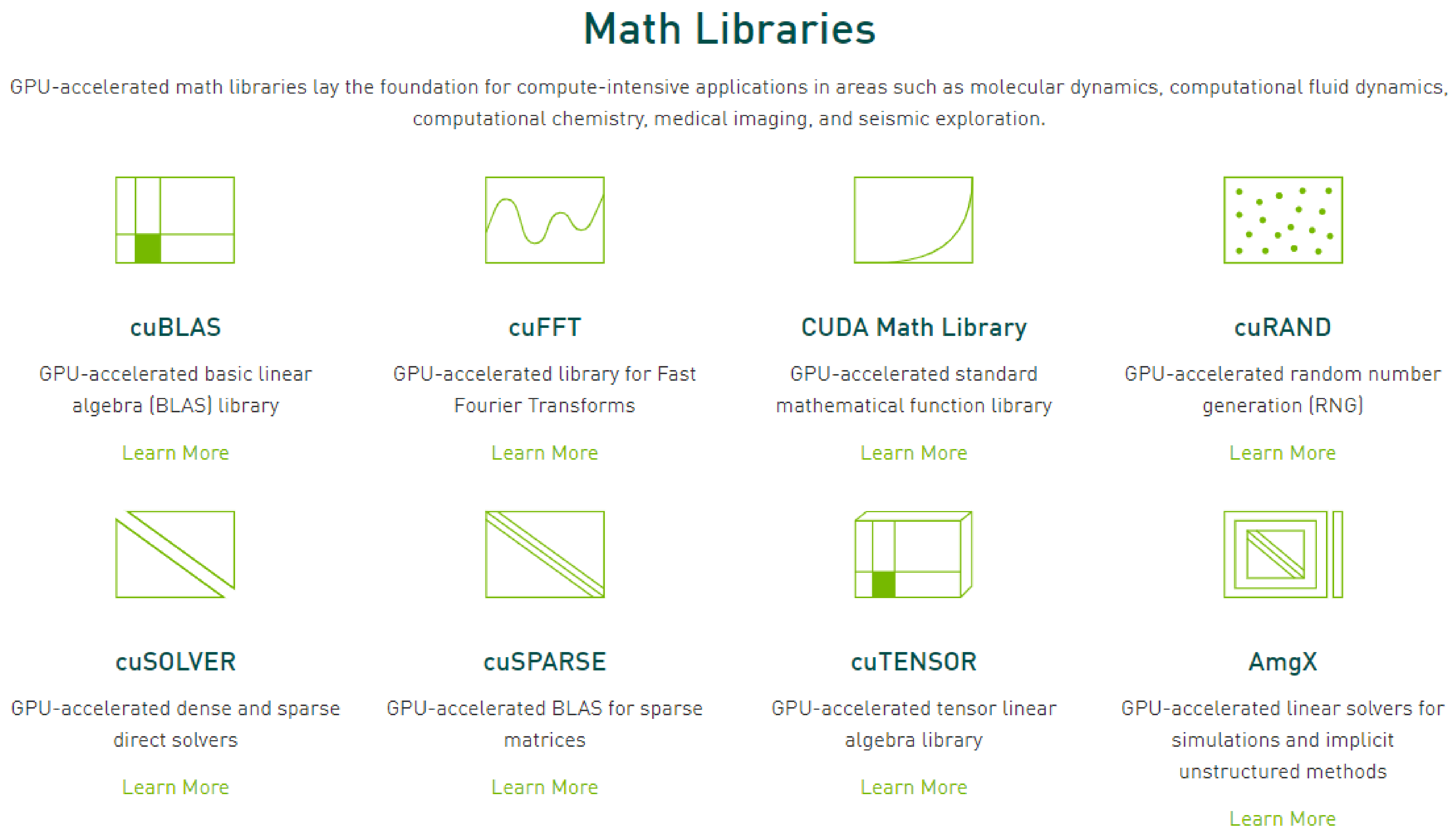
3 ways to accelerate your application



Libraries: Easy, High-Quality Acceleration

- 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:
  - NVIDIA libraries are tuned by experts

Some GPU-accelerated libraries



Math Libraries

GPU-accelerated math libraries lay the foundation for compute-intensive applications in areas such as molecular dynamics, computational fluid dynamics, computational chemistry, medical imaging, and seismic exploration.

cuBLAS

GPU-accelerated basic linear algebra (BLAS) library

Learn More

cuFFT

GPU-accelerated library for Fast Fourier Transforms

Learn More

CUDA Math Library

GPU-accelerated standard mathematical function library

Learn More

cuRAND

GPU-accelerated random number generation (RNG)

Learn More

cuSOLVER

GPU-accelerated dense and sparse direct solvers

Learn More

cuSPARSE

GPU-accelerated BLAS for sparse matrices

Learn More

cuTENSOR

GPU-accelerated tensor linear algebra library

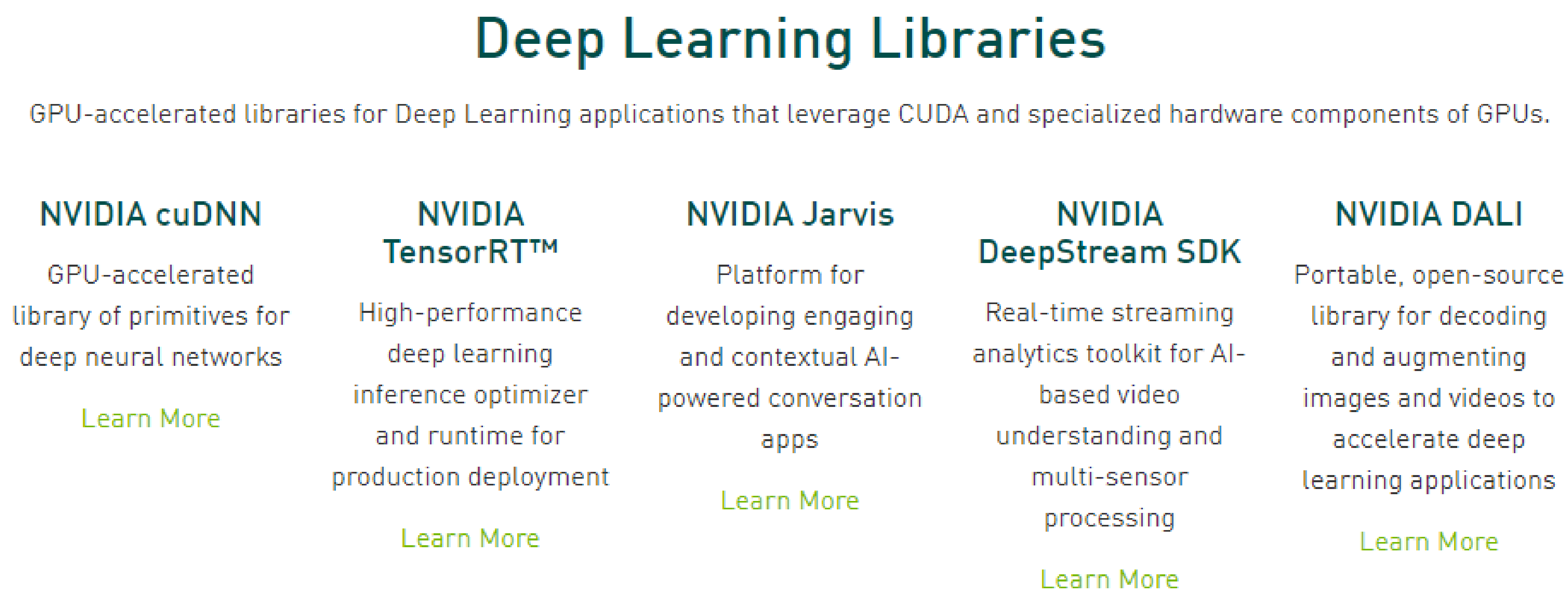
Learn More

AmgX

GPU-accelerated linear solvers for simulations and implicit unstructured methods

Learn More

Some GPU-accelerated libraries



Deep Learning Libraries

GPU-accelerated libraries for Deep Learning applications that leverage CUDA and specialized hardware components of GPUs.

NVIDIA cuDNN

GPU-accelerated library of primitives for deep neural networks

Learn More

NVIDIA TensorRT™

High-performance deep learning inference optimizer and runtime for production deployment

Learn More

NVIDIA Jarvis

Platform for developing engaging and contextual AI-powered conversation apps

Learn More

NVIDIA DeepStream SDK

Real-time streaming analytics toolkit for AI-based video understanding and multi-sensor processing

Learn More

NVIDIA DALI

Portable, open-source library for decoding and augmenting images and videos to accelerate deep learning applications

Learn More

3 Steps to CUDA-accelerated application

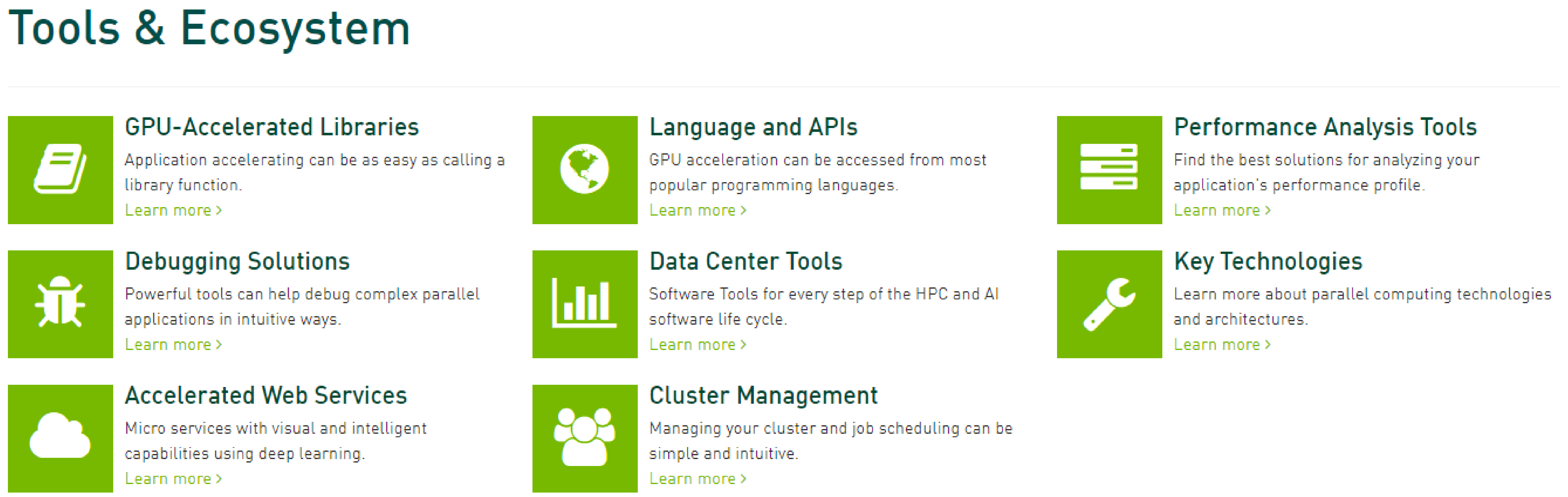
- Step 1: Substitute library calls with equivalent CUDA library calls

```
    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 cublas`

Explore the CUDA (Libraries) Ecosystem

CUDA Tools and Ecosystem described in detail on NVIDIA Developer Zone:
developer.nvidia.com/cuda-tools-ecosystem



Tools & Ecosystem

GPU-Accelerated Libraries
Application accelerating can be as easy as calling a library function.
Learn more ›

Language and APIs
GPU acceleration can be accessed from most popular programming languages.
Learn more ›

Performance Analysis Tools
Find the best solutions for analyzing your application's performance profile.
Learn more ›

Debugging Solutions
Powerful tools can help debug complex parallel applications in intuitive ways.
Learn more ›

Data Center Tools
Software Tools for every step of the HPC and AI software life cycle.
Learn more ›

Key Technologies
Learn more about parallel computing technologies and architectures.
Learn more ›

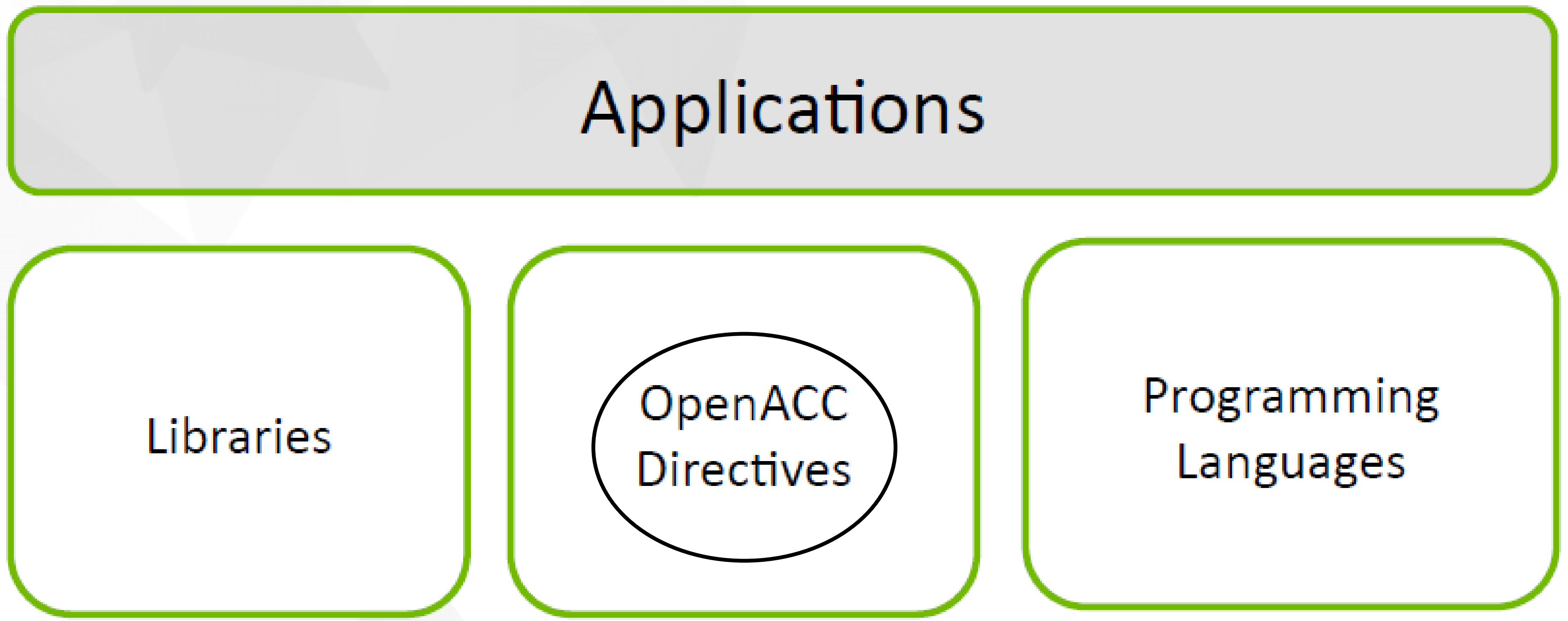
Accelerated Web Services
Micro services with visual and intelligent capabilities using deep learning.
Learn more ›

Cluster Management
Managing your cluster and job scheduling can be simple and intuitive.
Learn more ›

Tutorial: performing DGEMM on CPU and GPU

- See github repo

3 ways to accelerate your application



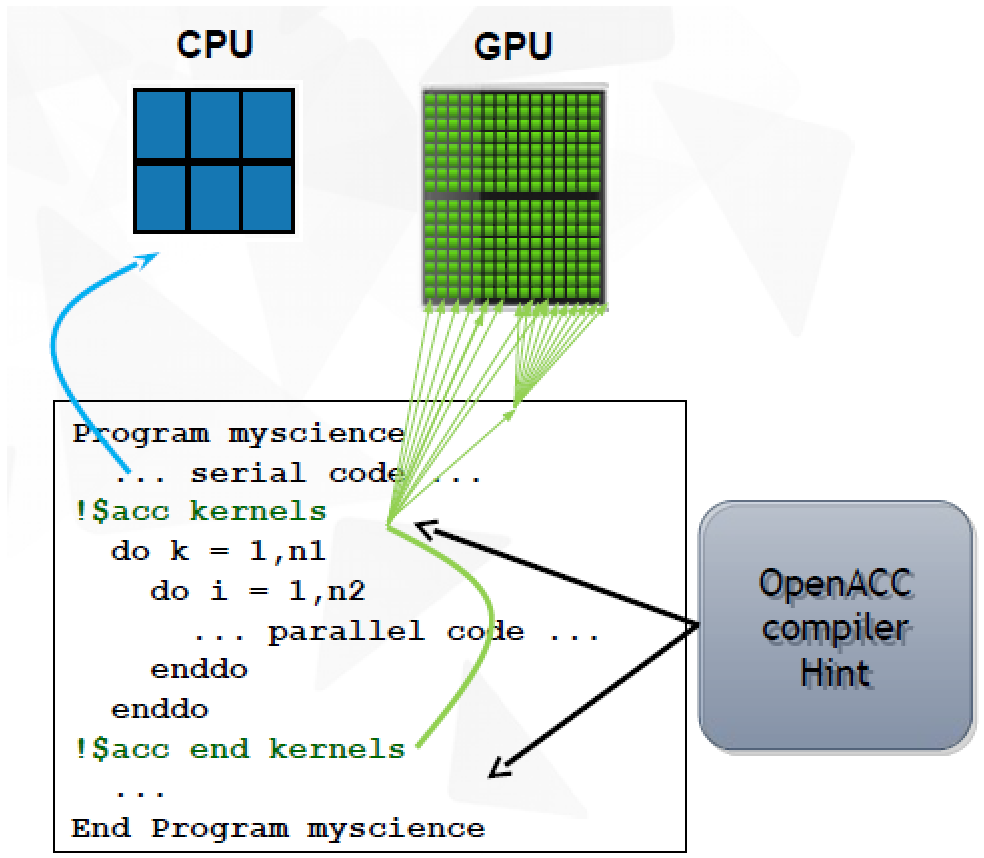
"Drop-in" Acceleration

Easily Accelerate Applications

Maximum Flexibility

OpenACC directives

- Simple Compiler hints
- Compiler parallelizes code
- Works on many-core GPUs v& multicore CPUs



```
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

- Easy:
  - Directives are the easy path to accelerate computer intensive applications
- Open:
  - OpenACC is an open GPU directives standard, making GPU programming straightforward and portable across parallel and multi-core processors
- Powerful:
  - GPU Directives allow complete access to the massive parallel power of a GPU

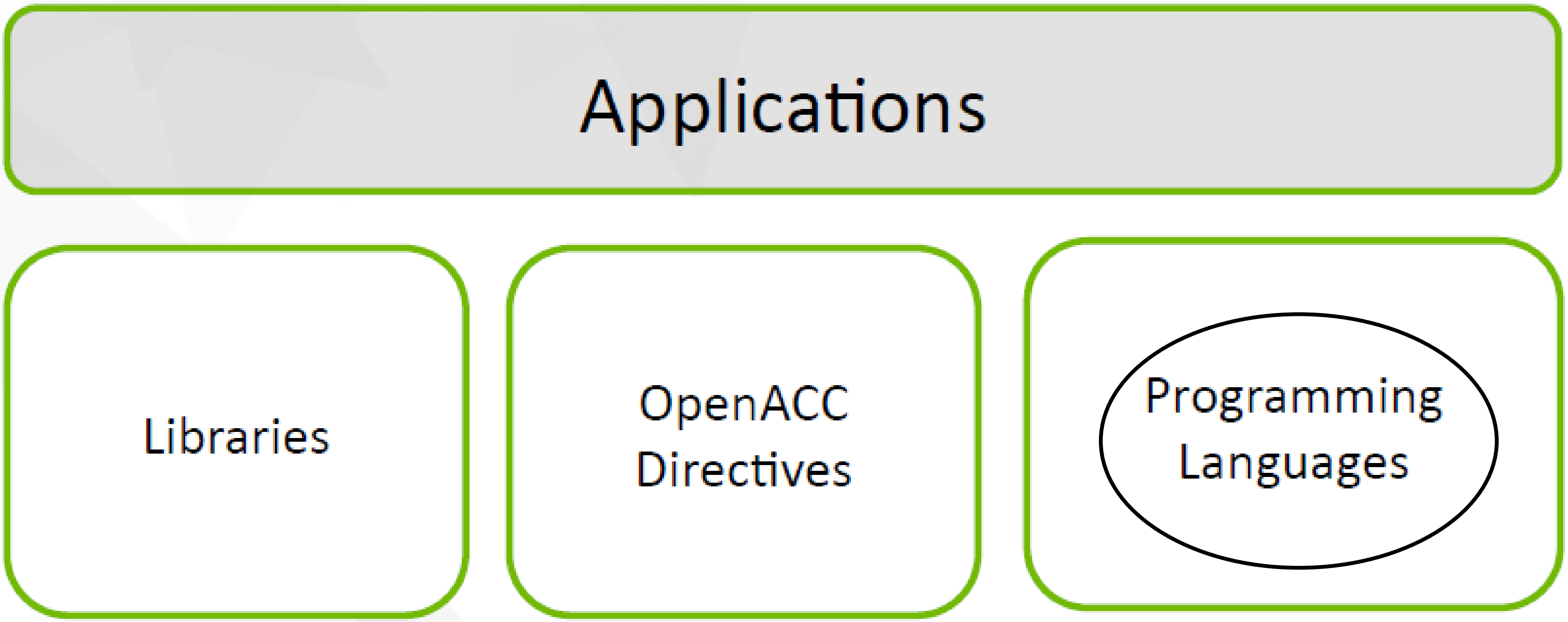
Is openACC available ?

- OpenACC is at version 2.7
- PGI compiler fully implements it
- GCC 9 includes initial support for OpenAcc 2.6

OpenACC –Directive Based Approach

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

3 ways to accelerate your application



"Drop-in" Acceleration

Easily Accelerate Applications

Maximum Flexibility

GPU Programming Languages

Numerical analytics	MATLAB, Mathematica, LabVIEW
Fortran	OpenACC, CUDA Fortran
C	OpenACC, CUDA C
C++	Thrust, CUDA C++
Python	PyCUDA, Copperhead

CUDA programming

- CUDA = Compute Unified Device Architecture
  - Expose general-purpose GPU computing as first-class capability4
  - Retain traditional DirectX/OpenGL graphics performance
- CUDA C
  - Based on industry-standard C
  - A handful of language extensions to allow heterogeneous programs
  - Straightforward APIs to manage devices, memory, etc.

CUDA basic concepts

- The GPU is viewed as a compute device that:
  - has its own RAM (device memory)
  - runs data-parallel portions of an application as kernels by using many threads
- GPU vs. CPU threads
  - GPU threads are extremely lightweight
  - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
  - multi-core CPU needs only a few (basically one thread per core)

Hello world

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no device code
- At its simplest, CUDA C is just C!

```
int main(void) {
printf("Hello World!\n");
return 0;
}
>nvcc hello_world.cu
```

Hello world on device

- To compile: `nvcc –o simple_kernel simple_kernel.cu`
- To execute: `./simple_kernel`

```
__global__ void mykernel(void) {
}
int main(void) {
mykernel<<<1,1>>>();
printf("Hello World!\n");
return 0;
}
```

Hello world on device

```
__global__ void mykernel(void) {
}
int main(void) {
mykernel<<<1,1>>>();
printf("Hello World!\n");
return 0;
}
```

- CUDA C keyword __global__ indicates that a function
  - Runs on the device
  - Called from host code
- nvcc splits source file into host and device components:
  - NVIDIA's compiler handles device functions like kernel()
  - Standard host compiler handles host functions like main()
  - gcc, icc, …

Hello world on device

```
__global__ void mykernel(void) {
}
int main(void) {
mykernel<<<1,1>>>();
printf("Hello World!\n");
return 0;
}
```

- Triple angle brackets mark a call from host code to device code
  - Also called a “kernel launch”
  - We’ll return to the parameters (1,1) in a moment

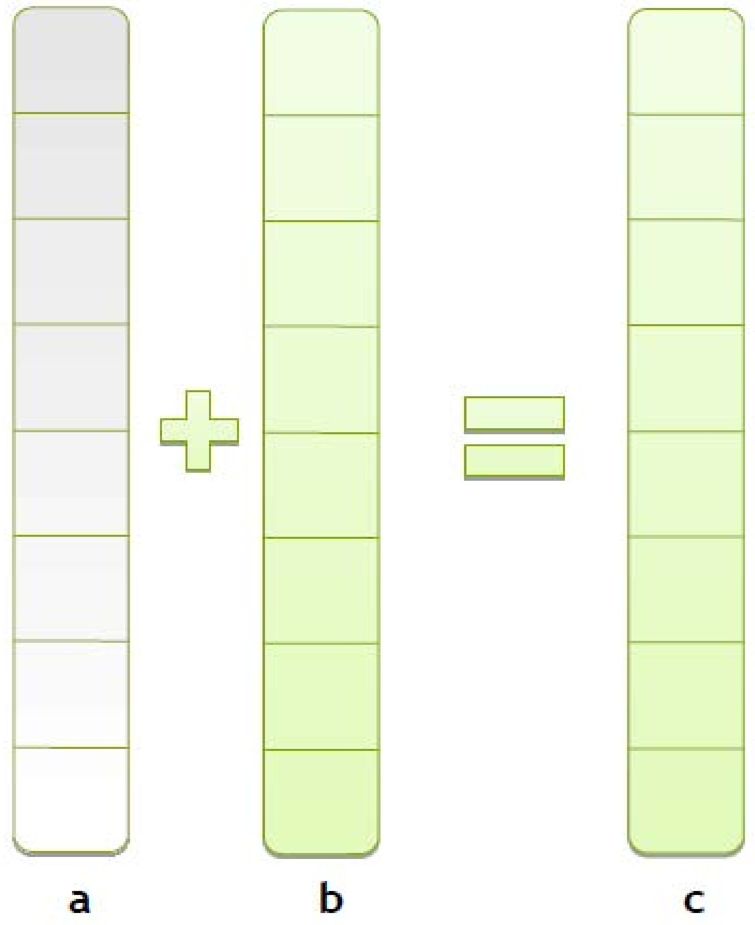
That’s all that is required to execute a function on the GPU

Parallel programming on GPU

- Adding to vectors on GPU
- A simple kernel:

```
__global__ void add(int *a, int *b, int *c)
{
*c = *a + *b;
}
```

- As before __global__ is a CUDA C/C++ keyword meaning:
  - add() will execute on the device
  - add() will be called from the host



Memory management (1)

- Host and device memory are separate entities
  - Device pointers point to GPU memory
  - May be passed to/from host code
  - May not be dereferenced in host code
- Host pointers point to CPU memory
  - May be passed to/from device code
  - May not be dereferenced in device code

Memory management (2)

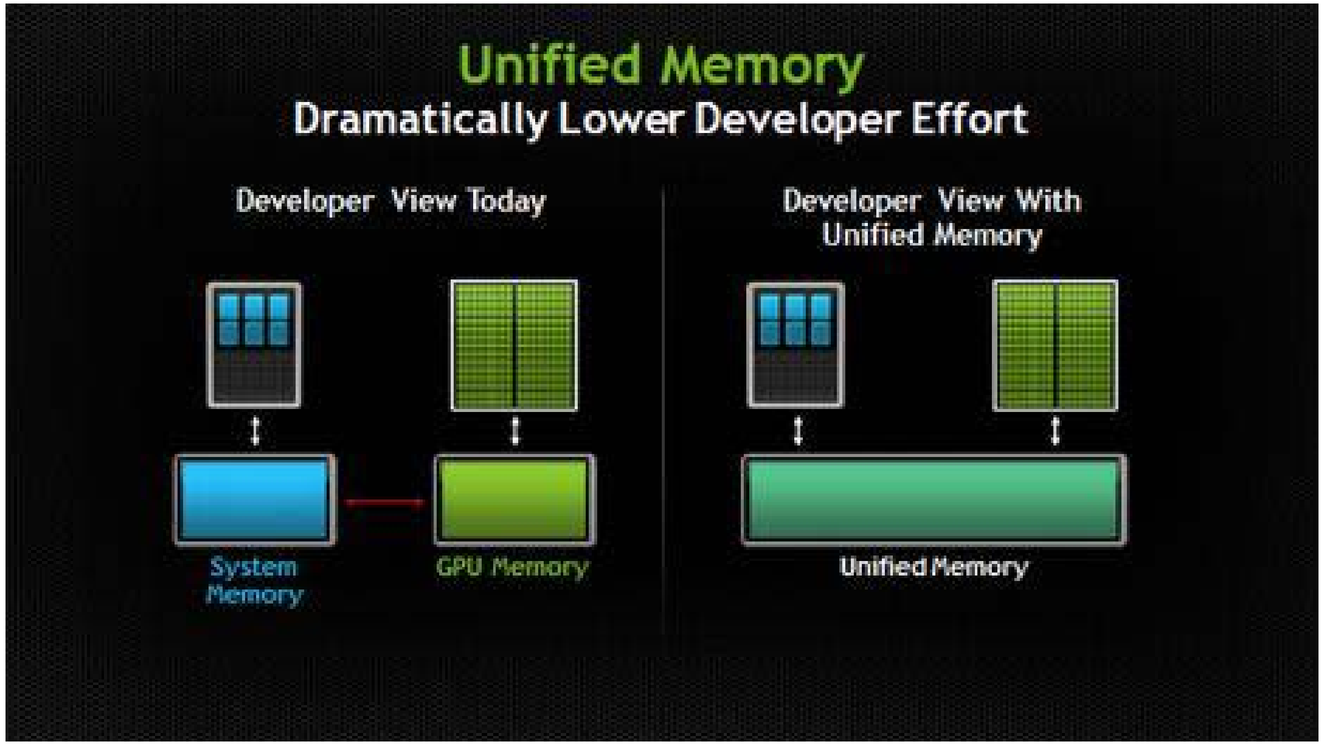
- Simple CUDA API for handling device memory
  - `cudaMalloc(&p, size),`
  - `cudaFree(&p),`
  - `cudaMemcpy(t, s, size, direction)`
- Similar to the C equivalents
  - `malloc(), free(), memcpy()`

Starting on CUDA 6.0 there is
a **Unified Memory** feature

Memory management (3)

- Unified Memory creates a pool of managed memory that is shared between the CPU and GPU.
- Managed memory is accessible to both the CPU and GPU using a single pointer.
- System automatically migrates data allocated in Unified Memory between host and device
- API:

`cudaMallocManaged()`



Addition on the Device: add()

```
__global__ void add(int *a, int *b, int *c)
{
*c = *a + *b;
}
```

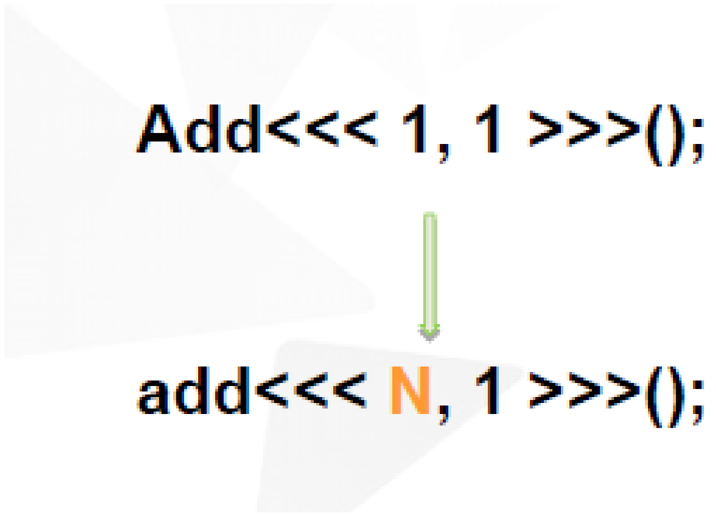
Let’s take a look at main()…

Addition on the Device: main()

```
___int main(void) {
int a, b, c; // host copies of a, b, c
int *d_a, *d_b, *d_c; // device copies of a, b, c
int size = sizeof(int);

// Allocate space for device copies of a, b, c
cudaMalloc((void **)&d_a, size);
cudaMalloc((void **)&d_b, size);
cudaMalloc((void **)&d_c, size);

// Setup input values
a = 2;
b = 7;
```

Moving to parallel…

- So how do we run code in parallel on the device?



```
Add<<< 1, 1 >>>();
```

```
add<<< N, 1 >>>();
```

- Instead of executing add() once, execute N times in parallel

Vector Addition on the Device

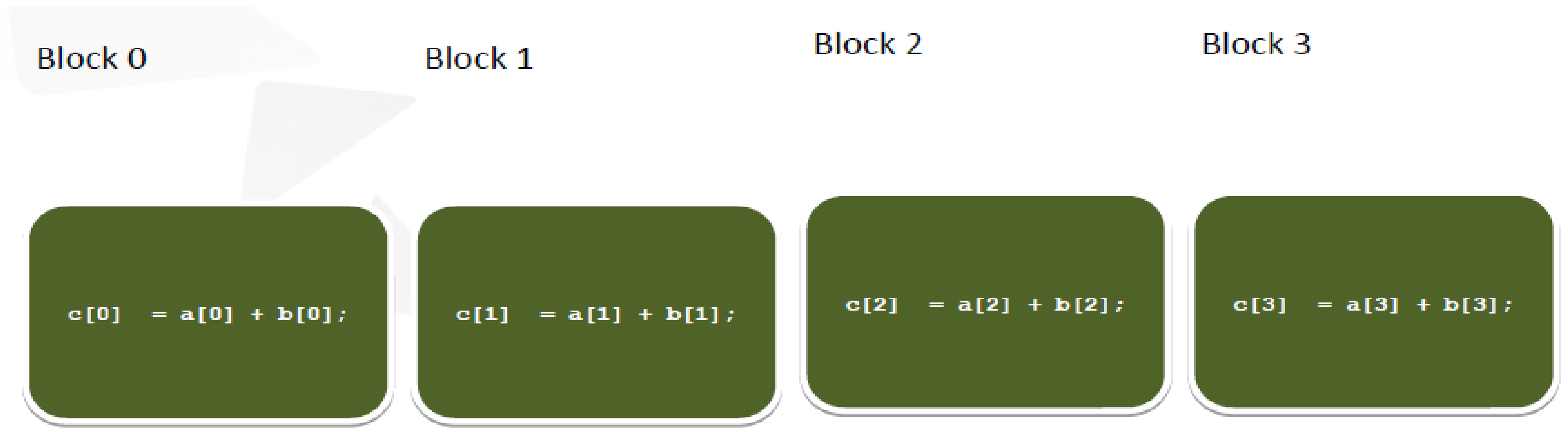
- With `add()` running in parallel we can do vector addition
- Terminology: each parallel invocation of `add()` is referred to as a block
  - The set of blocks is referred to as a grid
  - Each invocation can refer to its block index using blockIdx.x

```
__global__ void add(int *a, int *b, int *c) {
  c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
  }
```

- By using blockIdx.x to index into the array, each block handles a different index

Vector Addition on the Device

```
__global__ void add(int *a, int *b, int *c) {
c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```



Vector Addition on the Device: complete program (1)

```
#define N 512
int main(void) {
int *a, *b, *c; // host copies of a, b, c
int *d_a, *d_b, *d_c; // device copies of a, b, c
int size = N * sizeof(int);
// Alloc space for device copies of a, b, c
cudaMalloc((void **)&d_a, size);
cudaMalloc((void **)&d_b, size);
cudaMalloc((void **)&d_c, size);
// Alloc space for host copies of a, b, c and setup input values
a = (int *)malloc(size); random_ints(a, N);
b = (int *)malloc(size); random_ints(b, N);
c = (int *)malloc(size);
```

Vector Addition on the Device: complete program (2)

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
// Launch add() kernel on GPU with N blocks
add<<<N,1>>>(d_a, d_b, d_c);
// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

Review so far:

- Difference between host and device
  - Host CPU
  - Device GPU
- Using `__global__` to declare a function as device code
  - Executes on the device
  - Called from the host
- Passing parameters from host code to a device function

Review so far:

- Basic device memory management
  - `cudaMalloc()`
  - `cudaMemcpy()`
  - `cudaFree()`
- Launching parallel kernels
  - Launch N copies of add() with add<<<N,1>>>(…);
  - Use blockIdx.x to access block index

CUDA threads

- Terminology: a block can be split into parallel threads
- Let's change add() to use parallel threads instead of parallel blocks
- We use `threadIdx.x` instead of `blockIdx.x`
- Need to make one change in main()…

```
__global__ void add(int *a, int *b, int *c) {
c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

Vector Addition on the Device: using threads

```
#define N 512
int main(void) {
int *a, *b, *c; // host copies of a, b, c
int *d_a, *d_b, *d_c; // device copies of a, b, c
int size = N * sizeof(int);
// Alloc space for device copies of a, b, c
cudaMalloc((void **)&d_a, size);
cudaMalloc((void **)&d_b, size);
cudaMalloc((void **)&d_c, size);
// Alloc space for host copies of a, b, c and setup input values
a = (int *)malloc(size); random_ints(a, N);
b = (int *)malloc(size); random_ints(b, N);
c = (int *)malloc(size);
```

Vector Addition on the Device:using threads

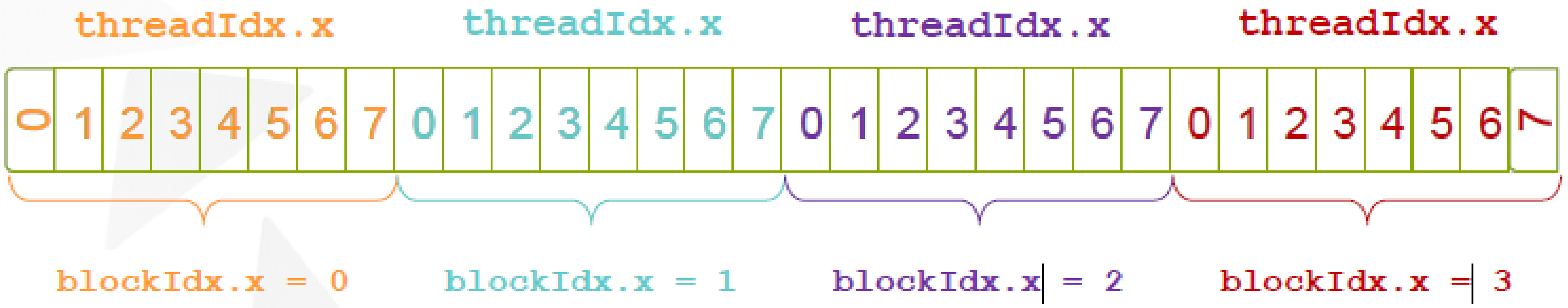
```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
// Launch add() kernel on GPU with N threads
add<<<1,N>>>(d_a, d_b, d_c);
// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

Combing blocks and threads

- We’ve seen parallel vector addition using:
  - Many blocks with one thread each
  - One block with many threads
- We want to adapt vector addition to use both blocks and threads
- First let’s discuss data indexing…

Indexing Arrays with Blocks and Threads

- No longer as simple as using blockIdx.x and threadIdx.x
- Consider indexing an array with one element per thread (8 threads/block)

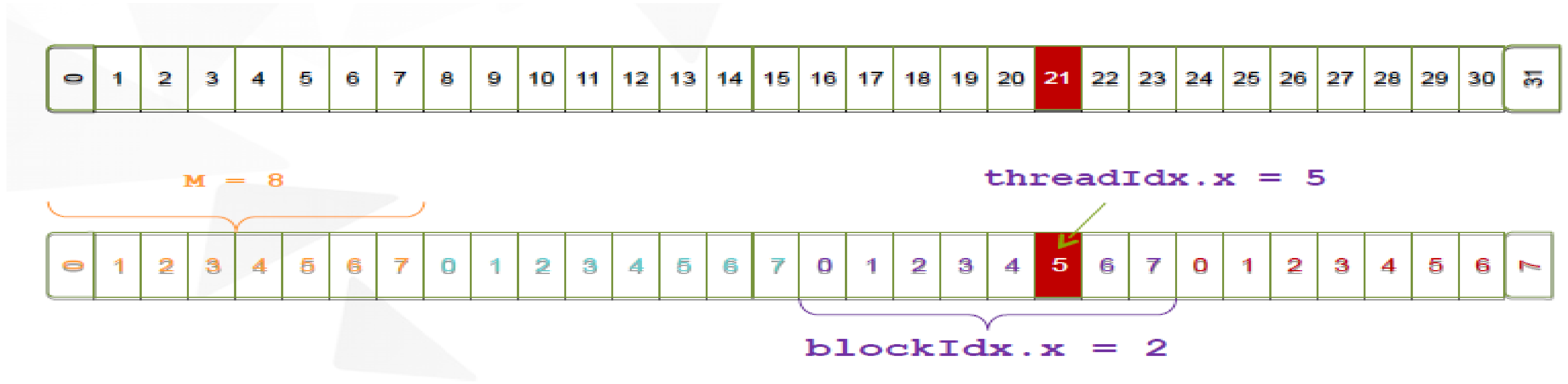


- With M threads/block a unique index for each thread is given by:

```
int index = threadIdx.x + blockIdx.x * M;
```

Indexing Arrays: Example

- Which thread will operate on the red element?



```
int index = threadIdx.x + blockIdx.x * M;
          = 5 + 2 * 8;
          = 21;
```

Addition with Blocks and Threads

- Use the built-in variable blockDim.x for threads per block

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

- Combined version of add() to use parallel threads and parallel blocks

```
__global__ void add(int *a, int *b, int *c) {
int index = threadIdx.x + blockIdx.x * blockDim.x;
c[index] = a[index] + b[index];
}
```

Vector Addition using threads and blocks

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
int *a, *b, *c; // host copies of a, b, c
int *d_a, *d_b, *d_c; // device copies of a, b, c
int size = N * sizeof(int);
// Alloc space for device copies of a, b, c
cudaMalloc((void **)&d_a, size);
cudaMalloc((void **)&d_b, size);
cudaMalloc((void **)&d_c, size);
// Alloc space for host copies of a, b, c and setup input values
a = (int *)malloc(size); random_ints(a, N);
b = (int *)malloc(size); random_ints(b, N);
c = (int *)malloc(size);
```

Vector Addition on the Device:using threads

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
// Launch add() kernel on GPU with N threads
add<<<N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(d_a, d_b, d_c);
// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of blockDim.x
- Avoid accessing beyond the end of the arrays:

```
__global__ void add(int *a, int *b, int *c, int n) {
int index = threadIdx.x + blockIdx.x * blockDim.x;
if (index < n)
c[index] = a[index] + b[index];
}
```

- Update the kernel launch:

```
add<<<(N + M-1)/M,M>>>(d_a, d_b, d_c, N);
```

Review

- Launching parallel kernels
  - Launch N copies of `add()` with `add<<<N/M,M>>>(…);`
  - Use `blockIdx.x` to access block index
  - Use `threadIdx.x` to access thread index within block
- Allocate elements to threads:

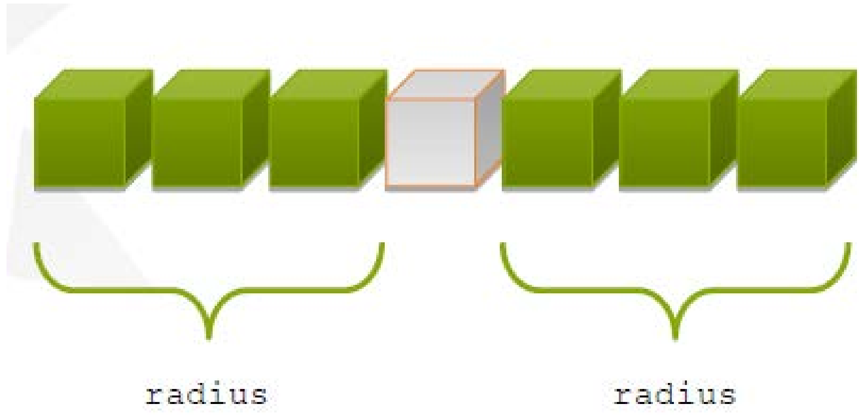
```
  int index = threadIdx.x + blockIdx.x
* blockDim.x;
```

Why bother with threads

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?
- Unlike parallel blocks, threads have mechanisms to:
  - Communicate
  - Synchronize
- To look closer, we need a new example…

1D stencil

- Consider applying a 1D stencil to a 1D array of elements
  - Each output element is the sum of input elements within a radius
- If radius is 3, then each output element is the sum of 7 input elements:



Implement within a block

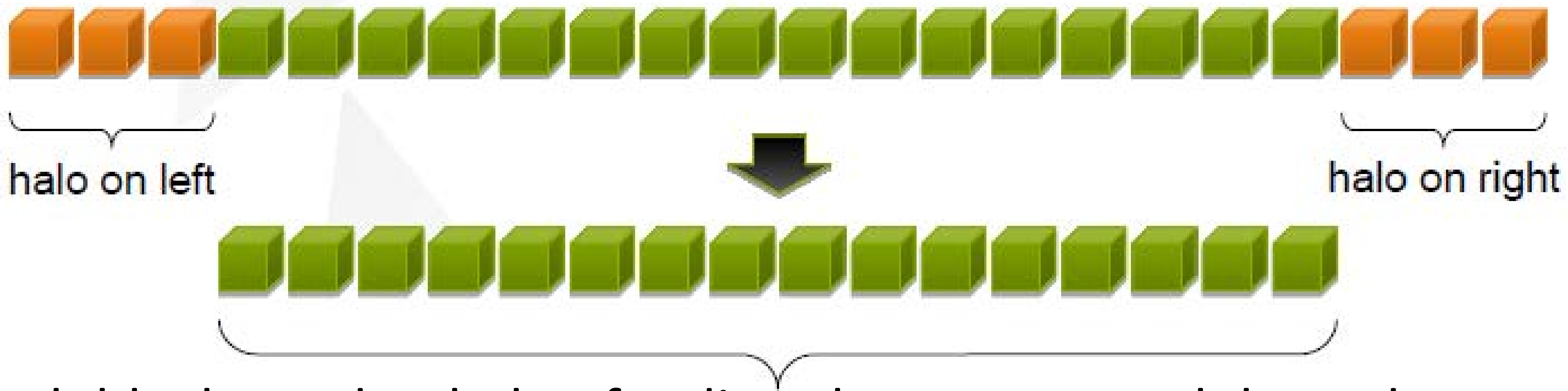
- Each thread processes one output element
  - blockDim.x elements per block
- Input elements are read several times
  - With radius 3, each input element is read seven times

Sharing Data between Threads

- Terminology: within a block, threads share data via shared memory
  - Extremely fast on-chip memory, user-managed
- Declare using `__shared__`, allocated per block
- Data is not visible to threads in other blocks

Implementing With Shared Memory

- Cache data in shared memory
  - Read (blockDim.x + 2 * radius) input elements from global memory to shared memory
  - Compute blockDim.x output elements
  - Write blockDim.x output elements to global memory



- Each block needs a halo of radius elements at each boundary

Stencil kernel

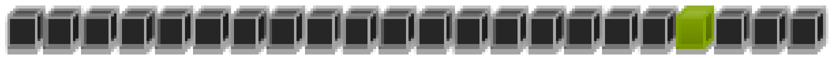
```
__global__ void stencil_1d(int *in, int *out) {

__shared__ int temp[BLOCK_SIZE + 2 * RADIUS];

int gindex = threadIdx.x + blockIdx.x * blockDim.x;
int lindex = threadIdx.x + RADIUS;

// Read input elements into shared memory
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
temp[lindex - RADIUS] = in[gindex - RADIUS];
temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}
```

Data race !

- The stencil example will not work…
- Suppose thread 15 reads the halo before thread 0 has fetched it…

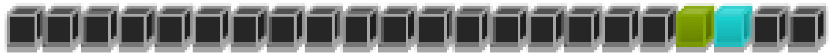
```
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
  temp[lindex - RADIUS = in[gindex - RADIUS];
  temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}

int result = 0;
result += temp[lindex + 1];
```

Store at temp[18]



Skipped, threadIdx > RADIUS

Load from temp[19]



_syncthreads()

```
void __syncthreads();
```

- Synchronizes all threads within a block
  - Used to prevent RAW / WAR / WAW hazards
- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block

Stencil kernel

```
__global__ void stencil_1d(int *in, int *out) {

__shared__ int temp[BLOCK_SIZE + 2 * RADIUS];

int gindex = threadIdx.x + blockIdx.x * blockDim.x;
int lindex = threadIdx.x + RADIUS;

// Read input elements into shared memory
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
temp[lindex - RADIUS] = in[gindex - RADIUS];
temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}
// Synchronize (ensure all the data is available)
__syncthreads();
```

Apply the stencil

```
// Apply the stencil

int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
result += temp[lindex + offset];

// Store the result

out[gindex] = result;
}
```

Coordinating host and device

- Kernel launches are asynchronous
  - Control returns to the CPU immediately
- CPU needs to synchronize before consuming the results
- A few ways to do this:

`cudaMemcpy()` Blocks the CPU until the copy is complete. Copy begins when all preceding CUDA calls have completed

`cudaMemcpyAsync()` Asynchronous, does not block the CPU

`cudaDeviceSynchronize()` Blocks the CPU until all preceding CUDA calls have completed

What to do now:

- CUDA
  - CUDA101:
    - Simple commands/program to interact with GPU cards on ORFEO
  - Vector:
    - Exercises on vector
  - 1DStencil
    - Exercise on 1D-stencil
- GEMM:
  - Compare CPU vs GPU performance on Matrix-Matrix multiplication