GPU BOOTCAMP
OPENACC

OPENACC

What to expect?

- Basic introduction to OpenACC directives
- HPC SDK Usage
- Portability across Multicore and GPU

OpenACC is...

a directives-based

parallel programming model

designed for

performance and **portability.**

Add Simple Compiler Directive

```
main()
 {
  <serial code>
  #pragma acc kernels
  {
    <parallel code>
  }
}
```

OpenACC Directives

Manage Data Movement

Initiate Parallel Execution

Optimize Loop Mappings

```
#pragma acc data copyin(a,b) copyout(c)
{
  ...
  #pragma acc parallel
  {
  #pragma acc loop gang vector
      for (i = 0; i < n; ++i) {
          c[i] = a[i] + b[i];
          ...
      }
  }
  ...
}
```

- Incremental
- Single source
- Interoperable
- Performance portable
- CPU, GPU, Manycore

OpenACC
Directives for Accelerators

OPENACC SYNTAX

Syntax for using OpenACC directives in code

C/C++

```
#pragma acc directive clauses
<code>
```

A *pragma* in C/C++ gives instructions to the compiler on how to compile the code. Compilers that do not understand a particular pragma can freely ignore it.

A *directive* in Fortran is a specially formatted comment that likewise instructions the compiler in it compilation of the code and can be freely ignored.

“*acc*” informs the compiler that what will come is an OpenACC directive

Directives are commands in OpenACC for altering our code.

Clauses are specifiers or additions to directives.

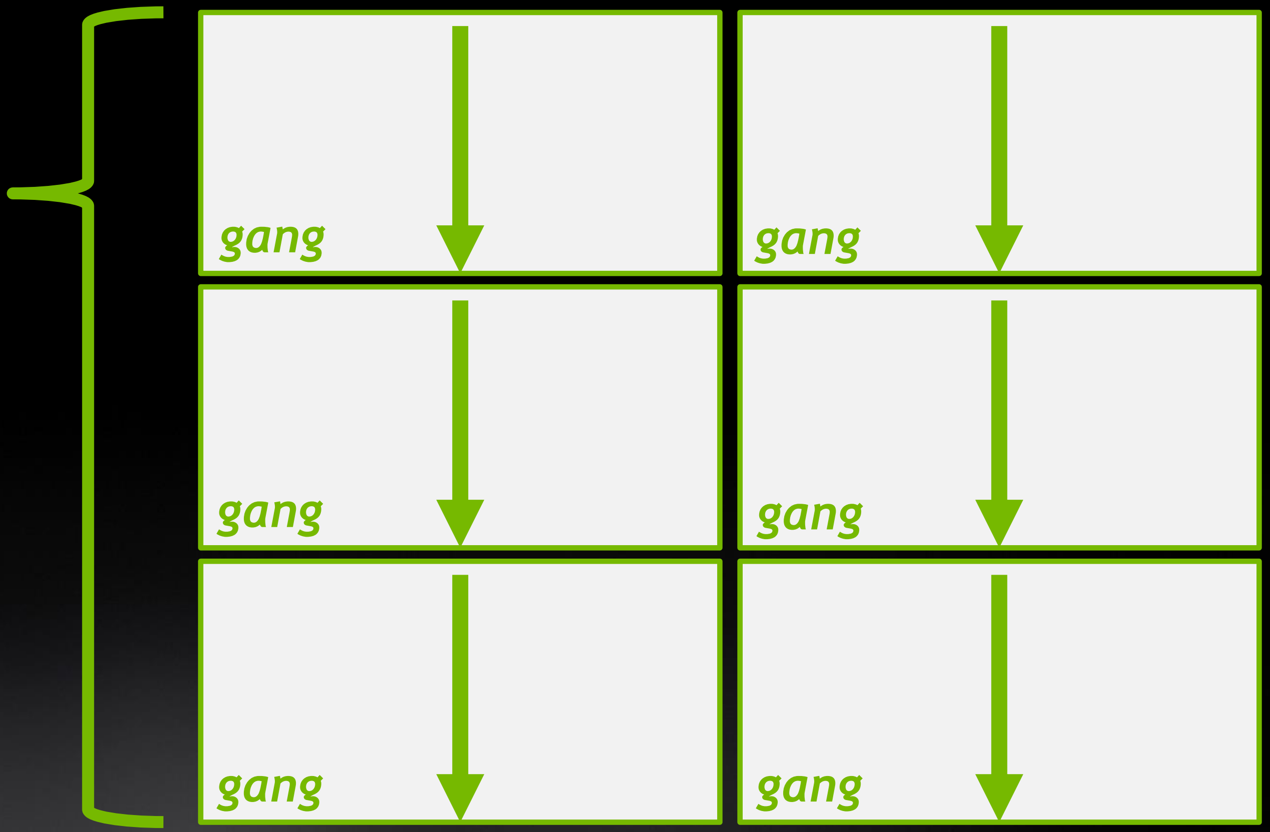
OPENACC PARALLEL DIRECTIVE

Expressing parallelism

```
#pragma acc parallel
{
```

When encountering the *parallel* directive, the compiler will generate *1 or more parallel* ***gangs***, which execute redundantly.

```
}
```



OPENACC PARALLEL DIRECTIVE

Expressing parallelism

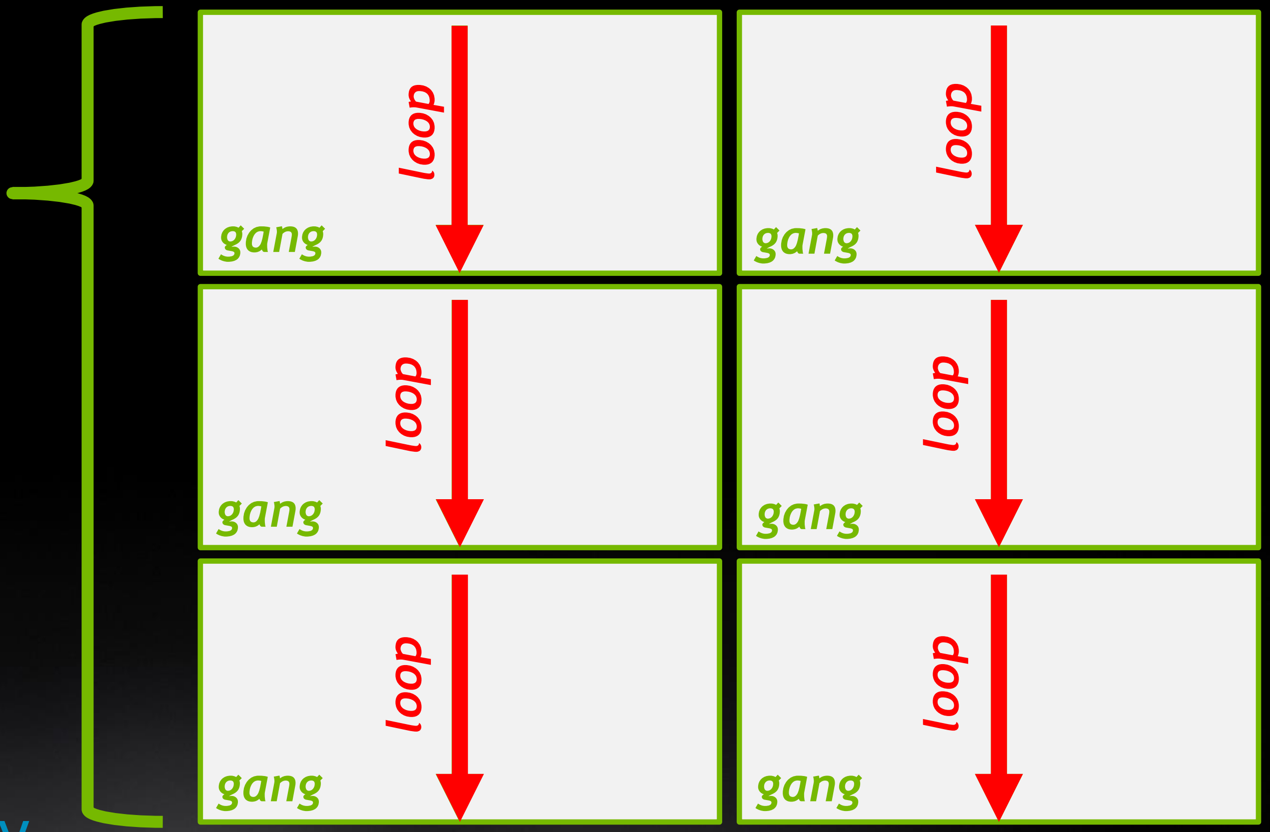
```
#pragma acc parallel
{

    for(int i = 0; i < N; i++)
    {
        // Do Something
    }

}
```

loop

This loop will be executed redundantly on each gang



OPENACC PARALLEL DIRECTIVE

Expressing parallelism

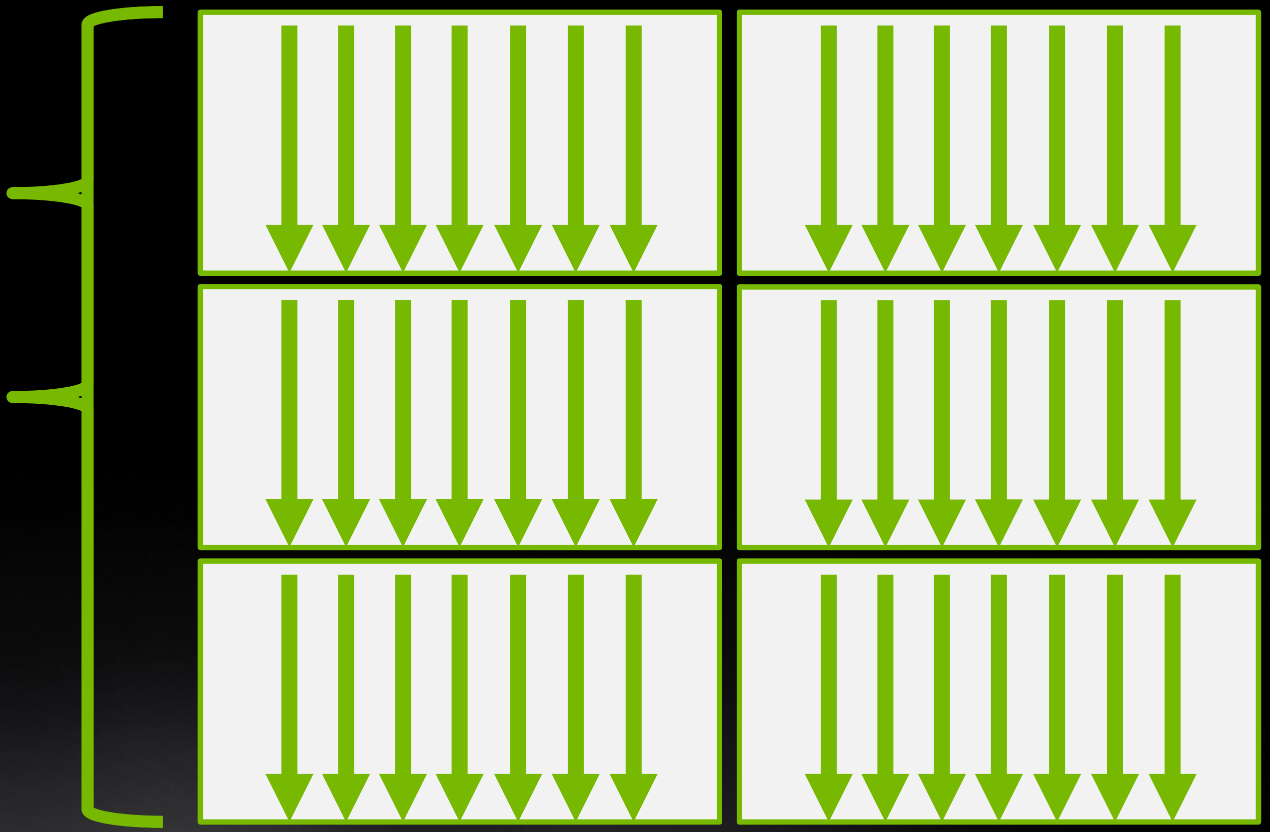
```
#pragma acc parallel
{


    #pragma acc loop
    for(int i = 0; i < N; i++)
    {
        // Do Something
    }



}
```

The *loop* directive informs the compiler which loops to parallelize.



OPENACC PARALLEL DIRECTIVE

Parallelizing a single loop

C/C++

```
#pragma acc parallel
{
  #pragma acc loop
  for(int i = 0; j < N; i++)
    a[i] = 0;
}
```

Use a **parallel** directive to mark a region of code where you want parallel execution to occur

This parallel region is marked by curly braces in C/C++ or a start and <end directive in Fortran

The **loop** directive is used to instruct the compiler to parallelize the iterations of the next loop to run across the parallel gangs

OPENACC PARALLEL DIRECTIVE

Parallelizing a single loop

C/C++

```
#pragma acc parallel loop
for(int i = 0; j < N; i++)
  a[i] = 0;
```

This pattern is so common that you can do all of this in a single line of code

In this example, the parallel loop directive applies to the next loop

This directive both marks the region for parallel execution and distributes the iterations of the loop.

When applied to a loop with a data dependency, parallel loop may produce incorrect results

BUILD AND RUN THE CODE

NVIDIA HPC SDK

- Comprehensive suite of compilers, libraries, and tools used to GPU accelerate HPC modeling and simulation application
- The NVIDIA HPC SDK includes the new NVIDIA HPC compiler supporting OpenACC C and Fortran
  - The command to compile C code is ‘nvcc’
  - The command to compile C++ code is ‘nvc++’

```
nvcc –fast –Minfo=accel -ta=tesla:managed main.c
```

BUILDING THE CODE

-Minfo shows more details

```
$ nvcc -fast -ta=multicore -Minfo=accel laplace2d_uvm.c
main:
     63, Generating Multicore code
         64, #pragma acc loop gang
     64, Accelerator restriction: size of the GPU copy of Anew,A is unknown
         Generating reduction(max:error)
     66, Loop is parallelizable
```

```
$ nvcc -fast -ta=tesla:managed -Minfo=accel rdf.c
main:
     63, Accelerator kernel generated
         Generating Tesla code
         64, #pragma acc loop gang /* blockIdx.x */
             Generating reduction(max:error)
         66, #pragma acc loop vector(128) /* threadIdx.x */
     63, Generating implicit copyin(A[:])
Generating implicit copy(error)
     66, Loop is parallelizable
```

RDF
Pseudo Code

```
for (int frame=0;frame<nconf;frame++){

        for(int id1=0;id1<numatm;id1++)
        {
                for(int id2=0;id2<numatm;id2++)
                {
                        dx=d_x[]-d_x[];
                        dy=d_y[]-d_y[];
                        dz=d_z[]-d_z[];
                        r=sqrtf(dx*dx+dy*dy+dz*dz);

                        if (r<cut) {
                                ig2=(int)(r/del);
                                d_g2[ig2] = d_g2[ig2] +1 ;
                        }
                }
        }
}
```

- Across Frames
- Find Distance
- Reduction

RDF
Pseudo Code

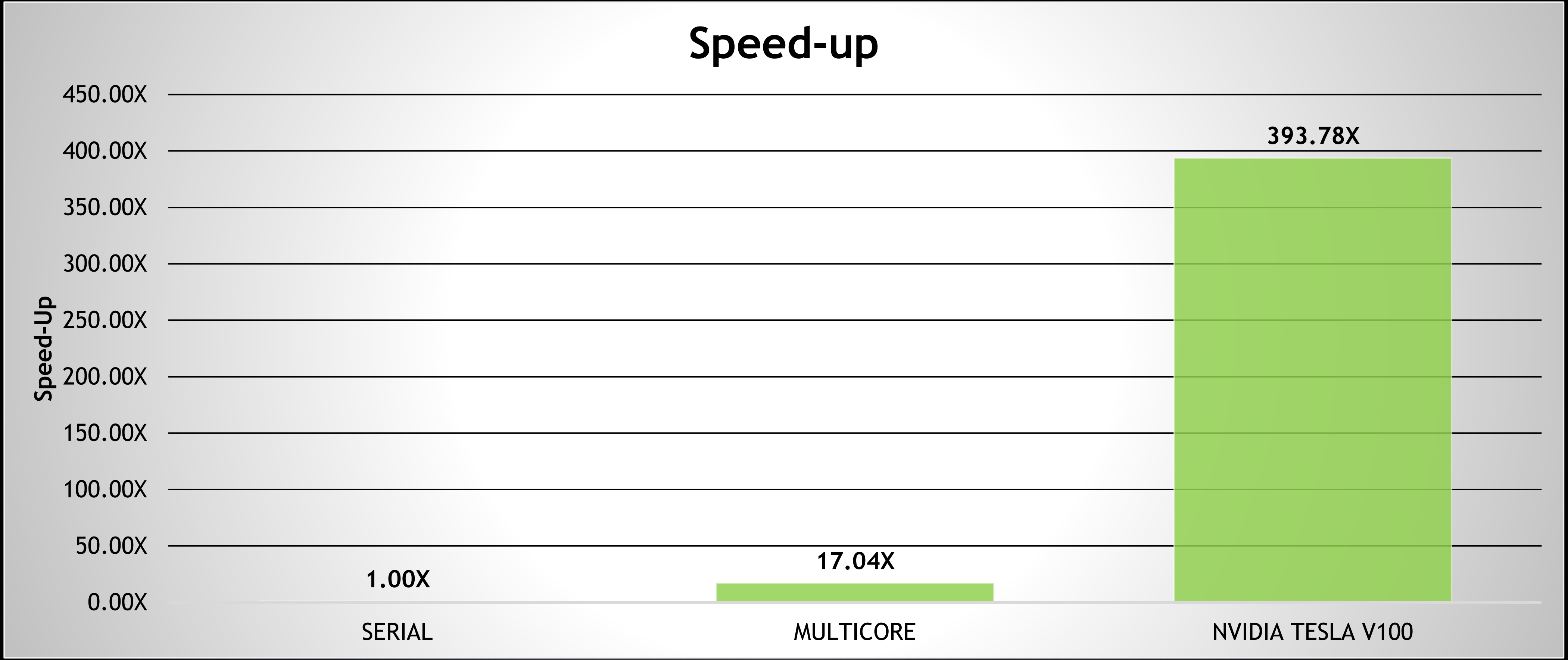
```
#pragma acc parallel loop
    for (int frame=0;frame<nconf;frame++){

        for(int id1=0;id1<numatm;id1++)
        {
            for(int id2=0;id2<numatm;id2++)
            {
                dx=d_x[]-d_x[];
                dy=d_y[]-d_y[];
                dz=d_z[]-d_z[];
                r=sqrtf(dx*dx+dy*dy+dz*dz);

                if (r<cut) {
                    ig2=(int)(r/del);
                    #pragma acc atomic
                    d_g2[ig2] = d_g2[ig2] +1 ;
                }
            }
        }
```

- Parallel Loop construct
- Atomic Construct

OPENACC SPEEDUP



HPC SDK 20.11, NVIDIA Tesla V100, DGX1

REFERENCES

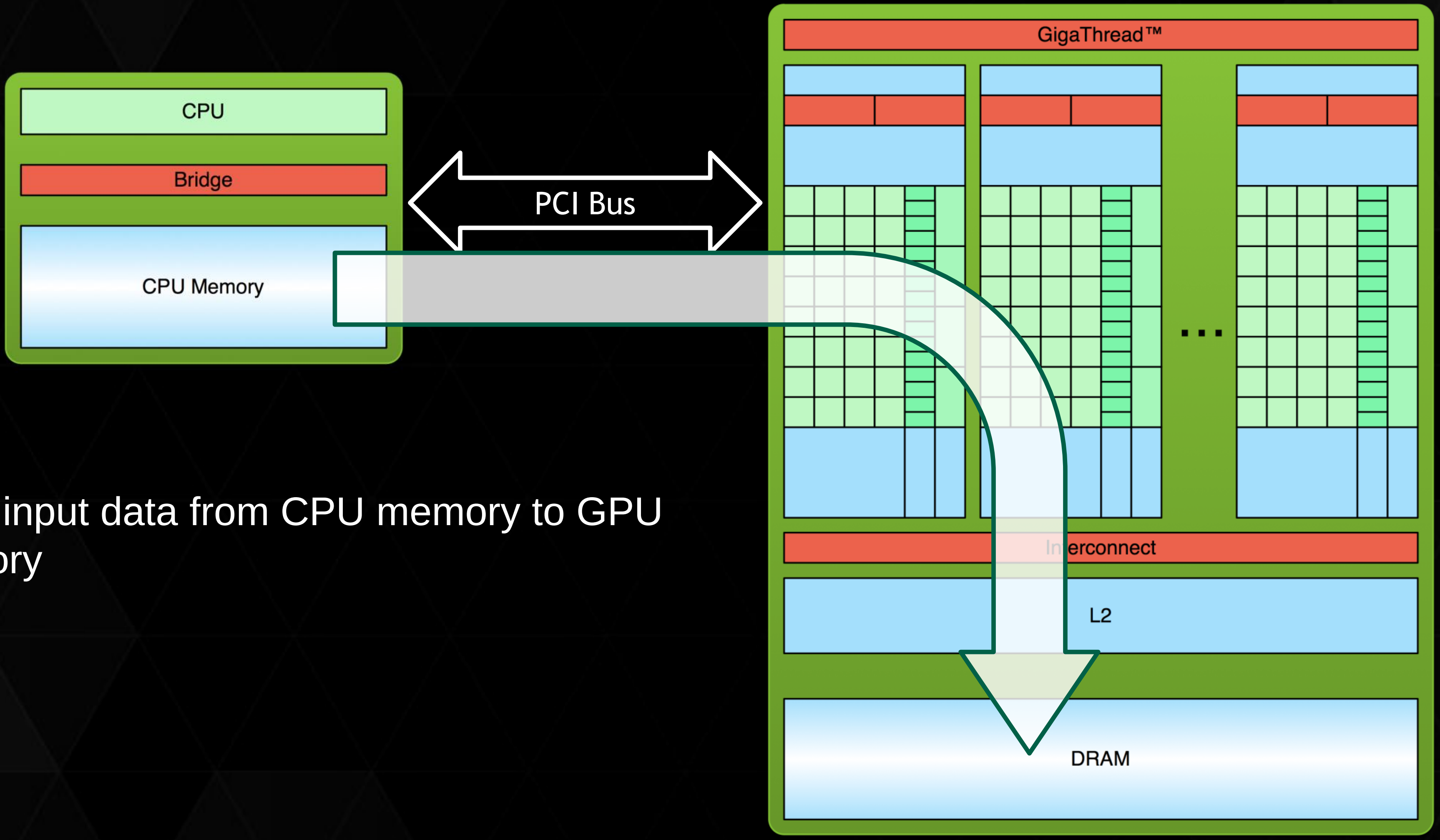
https://www.openacc.org/get-started

https://developer.nvidia.com/hpc-sdk

THANK YOU

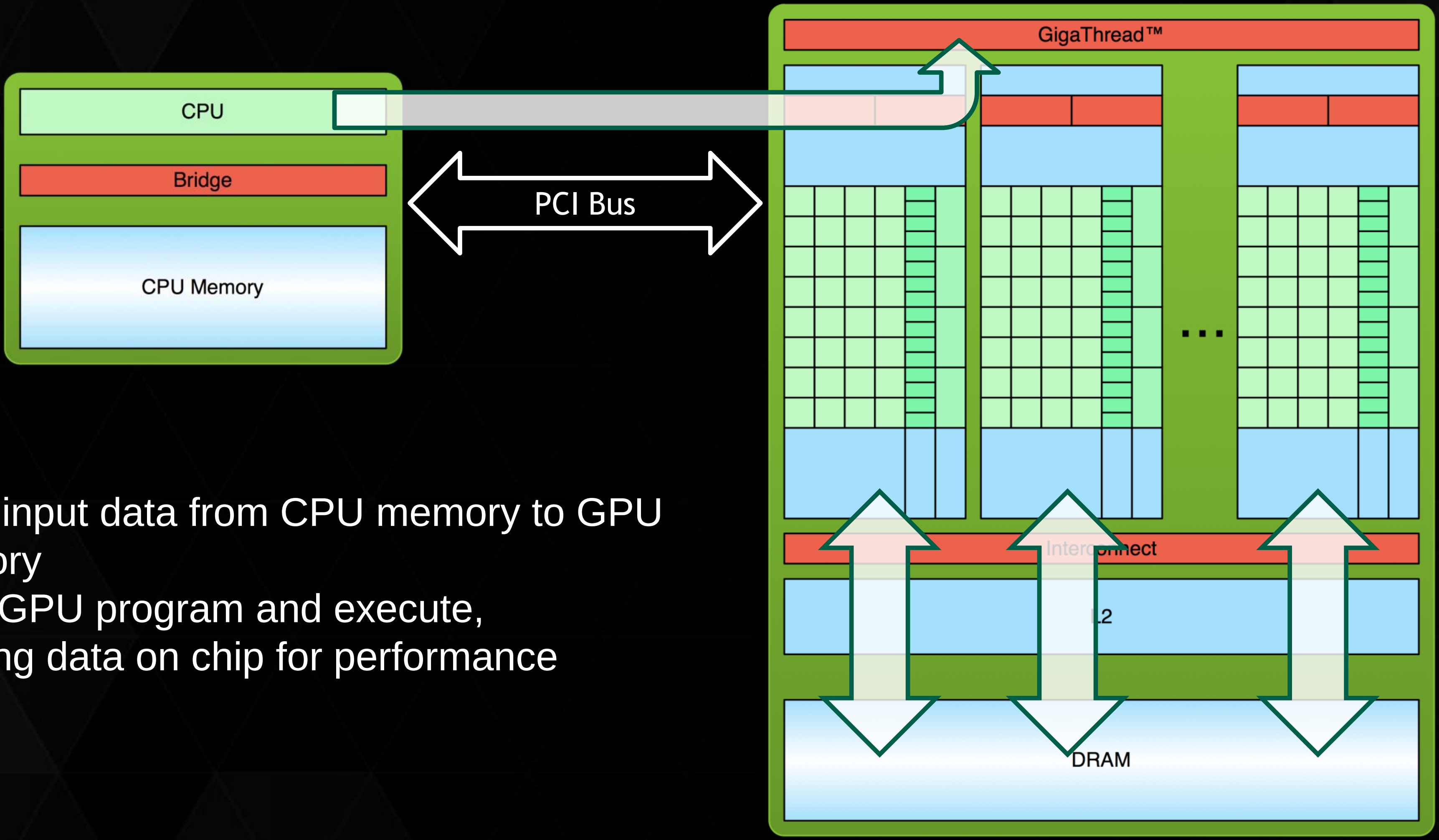
NVIDIA

PROCESSING FLOW – STEP 1



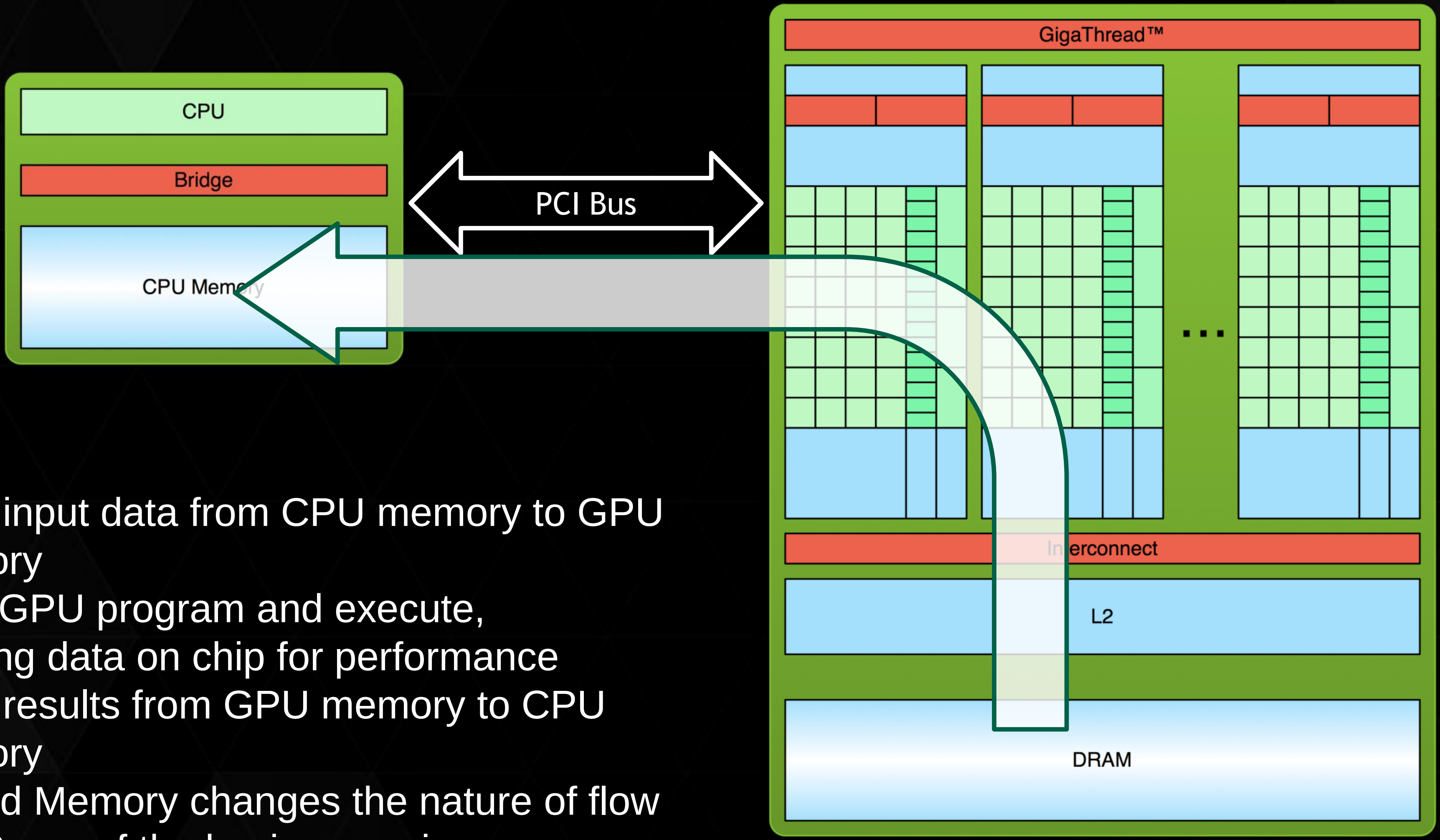
1. Copy input data from CPU memory to GPU memory

PROCESSING FLOW – STEP 2



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

PROCESSING FLOW – STEP 3

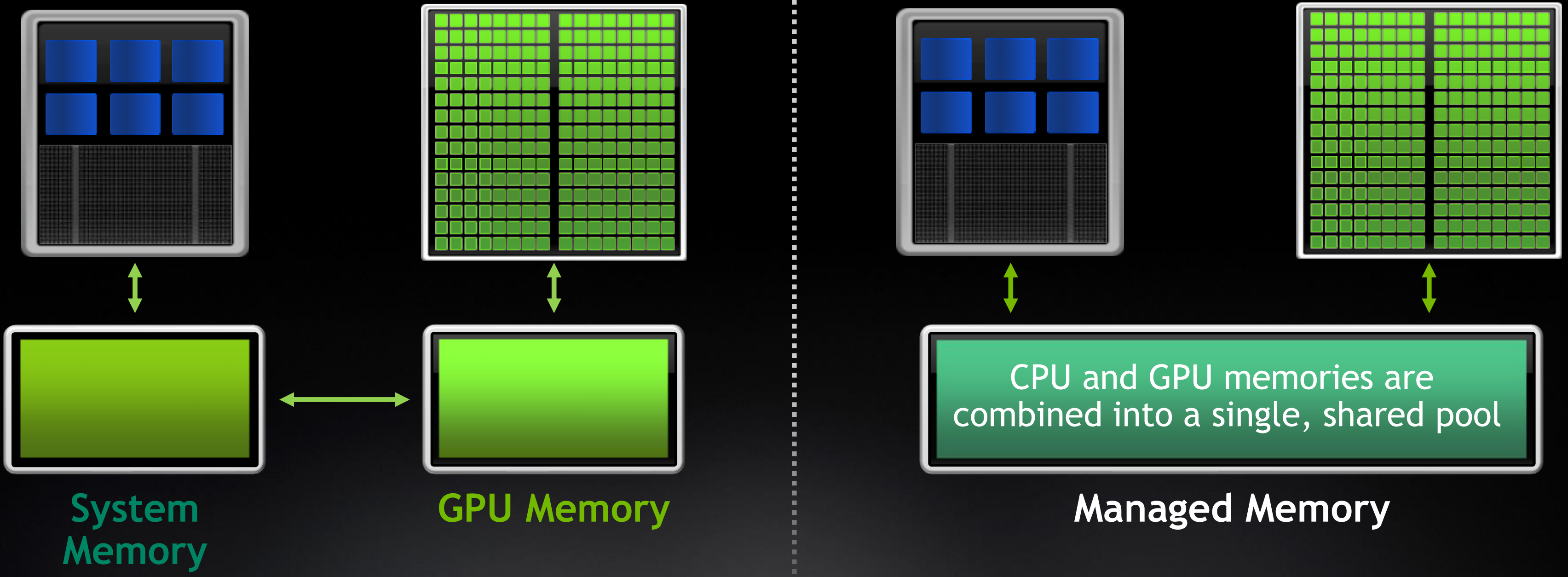


1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory
4. Unified Memory changes the nature of flow
   - Some of the basics remains same

CUDA UNIFIED MEMORY

Simplified Developer Effort

Commonly referred to as
"managed memory."

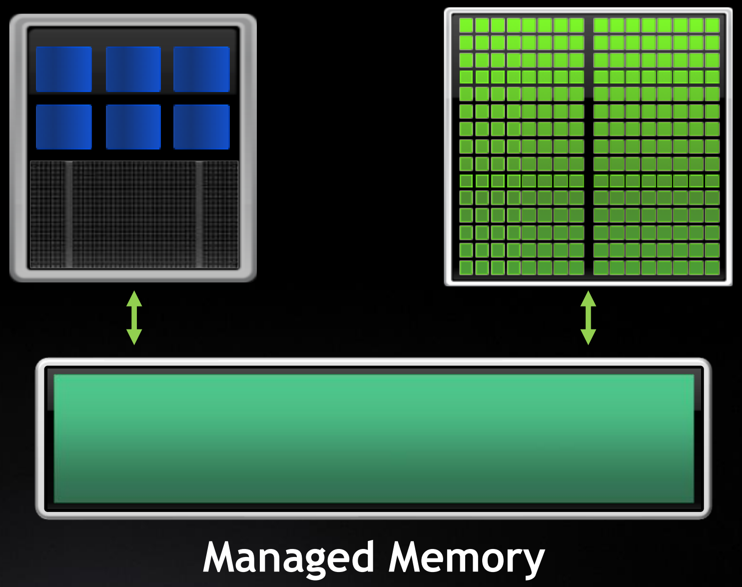


MANAGED MEMORY

Limitations

- The programmer will almost always be able to get better performance by manually handling data transfers
- Memory allocation/deallocation takes longer with managed memory
- Cannot transfer data asynchronously
- Currently only available from PGI on NVIDIA GPUs.

With Managed Memory



DATA CLAUSES

`copy( list )` **Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.**

Principal use: For many important data structures in your code, this is a logical default to input, modify and return the data.

`copyin( list )` **Allocates memory on GPU and copies data from host to GPU when entering region.**

Principal use: Think of this like an array that you would use as just an input to a subroutine.

`copyout( list )` **Allocates memory on GPU and copies data to the host when exiting region.**

Principal use: A result that isn't overwriting the input data structure.

`create( list )` **Allocates memory on GPU but does not copy.**

Principal use: Temporary arrays.

ARRAY SHAPING

Sometimes the compiler needs help understanding the *shape* of an array

The first number is the start index of the array

In C/C++, the second number is how much data is to be transferred

In Fortran, the second number is the ending index

```
copy(array[starting_index:length])
```

C/C++

```
copy(array(starting_index:ending_index))
```

Fortran

ARRAY SHAPING (CONT.)

Multi-dimensional Array shaping

```
copy(array[0:N][0:M])
```

C/C++

Both of these examples copy a 2D array to the device

```
copy(array(1:N, 1:M))
```

Fortran

OPENACC DATA DIRECTIVE

Definition

The data directive defines a lifetime for data on the device beyond individual loops

During the region data is essentially “owned by” the accelerator

Data clauses express shape and data movement for the region

```
#pragma acc data clauses
{

  < Sequential and/or Parallel code >

}
```

```
!$acc data clauses

  < Sequential and/or Parallel code >

!$acc end data
```

STRUCTURED DATA DIRECTIVE

Example

```
#pragma acc data copyin(a[0:N],b[0:N]) copyout(c[0:N])
{
  #pragma acc parallel loop
  for(int i = 0; i < N; i++){
    c[i] = a[i] + b[i];
  }
}
```

Action

Host Memory

A | B | C’

Device memory

A | B | C’