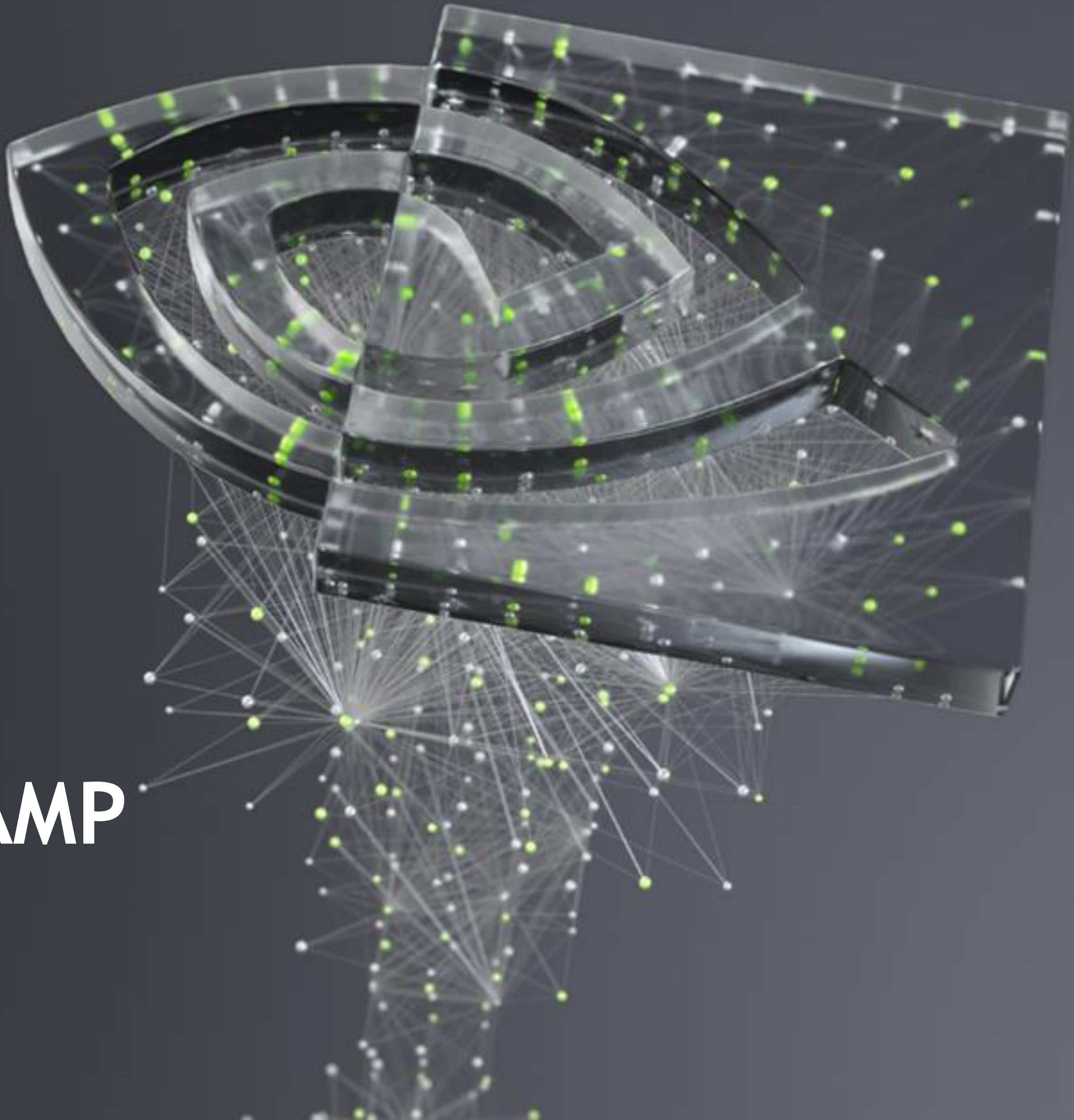
NVIDIA



N-WAYS GPU BOOTCAMP

OPENMP TARGET OFFLOAD

OPENMP

What to expect?

- OpenMP basic
- OpenMP target offload constructs for accelerated computing
- Portability between multicore and GPU

OPENMP

A Brief History

- 1996 - Architecture Review Board (ARB) formed by several vendors implementing their own directives for Shared Memory Parallelism (SMP).
- 1997 - 1.0 was released for C/C++ and Fortran with support for parallelizing loops across threads.
- 2000, 2002 – Version 2.0 of Fortran, C/C++ specifications released.
- 2005 – Version 2.5 released, combining both specs into one.
- 2008 – Version 3.0 released, added support for tasking
- 2011 – Version 3.1 release, improved support for tasking
- 2013 – Version 4.0 released, added support for offloading (and more)
- 2015 – Version 4.5 released, improved support for offloading targets (and more)

OPENMP ON CPU

OPENMP

Syntax

```
#pragma omp directive
```

```
!$ omp directive
```

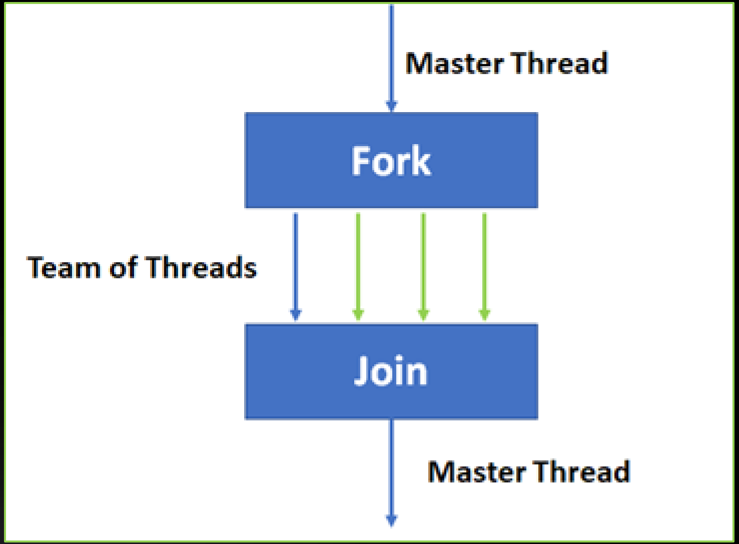
- #pragma in C/C++ is what's known as a "compiler hint."
- omp is an addition to our pragma, it is known as the “sentinel”. It specifies that this is an OpenMP pragma. Any non-OpenMP compiler will ignore this pragma.
- directives are commands in OpenMP that will tell the compiler to do some action. For now, we will only use directives that allow the compiler to parallelize our code

OPENMP

Fork Join Model

Fork Join Model

- OpenMP uses the fork-join model of parallel execution. All OpenMP programs begin as a single process: the master thread. The master thread executes sequentially until the first parallel region construct is encountered.
- FORK: the master thread then creates a team of parallel threads.The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads.
- JOIN: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.

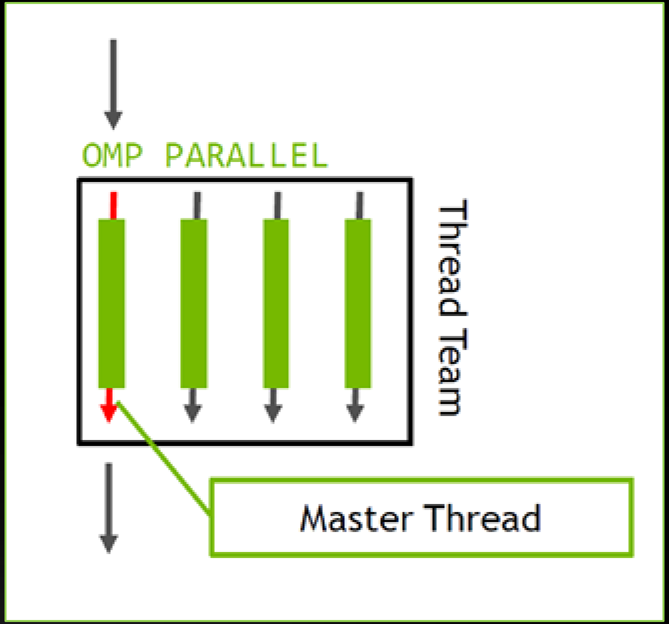


OPENMP

Parallel Region

PARALLEL Directive

- Spawns a team of threads
- Execution continues redundantly on all threads of the team.
- All threads join at the end and the master thread continues execution.



OpenMP Parallel Region

C - Syntax

```
//Include the header file
#include <omp.h>

 main(int argc, char *argv[]) {

 int nthreads;

 /* Fork a team of threads*/
 #pragma omp parallel
   {

   /* Obtain and print thread id */
   printf("Hello World from thread = %d\n", omp_get_thread_num());

   /* Only master thread does this */
   if (omp_get_thread_num() == 0)
     {
     nthreads = omp_get_num_threads();
     printf("Number of threads = %d\n", nthreads);
     }

   }  /* All threads join master thread and terminate */

 }
```

Include Header File

- Spawns parallel region

- Get Thread Id

OpenMP Parallel Region

Fortran - Syntax

```
program hello
  integer :: omp_rank

  !$omp parallel private(omp_rank)

  omp_rank = omp_get_thread_num()
  print *, 'Hello world! by thread ', omp_rank

  !$omp end parallel

end program hello
```

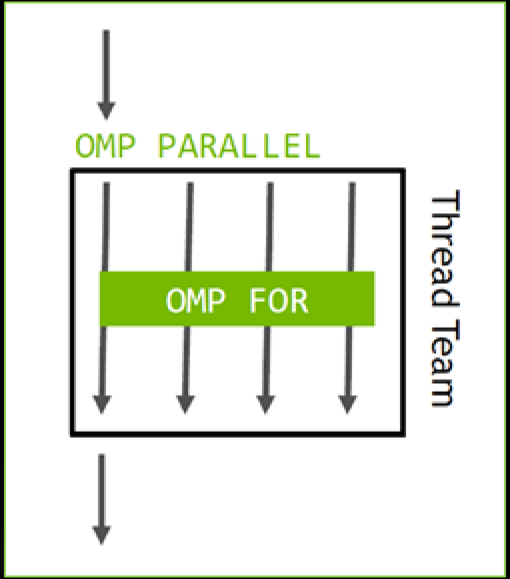
- Spawns parallel region
- Get Thread Id

OPENMP

Worksharing

FOR/DO (Loop) Directive

- Divides ("workshares") the iterations of the next loop across the threads in the team
- How the iterations are divided is determined by a schedule.



C/C++

```
//Create a team of threads
#pragma omp parallel
{
//workshare this loop across those threads.
   #pragma omp for
   for (i=0; i < N; i++)
     c[i] = a[i] + b[i];

   }   /* end of parallel region */
```

Fortran

```
!Create a team of threads
!$omp parallel
!workshare this loop across those threads.
   !$omp for
    do i=1,N
        < loop code >
    end do
!$omp end parallel
```

TARGETING THE GPU

OPENMP

Target Offloading

TARGET Directive

- Offloads execution and associated data from the CPU to the GPU
- The target device owns the data, accesses by the CPU during the execution of the target region are forbidden.
- Data used within the region may be implicitly or explicitly mapped to the device.
- All of OpenMP is allowed within target regions, but only a subset will run well on GPUs.

C/C++

```
#pragma omp target
{
                #pragma omp parallel for reduction(max:error)
                for( int j = 1; j < n-1; j++) {
…
                }
 }}
```

Fortran

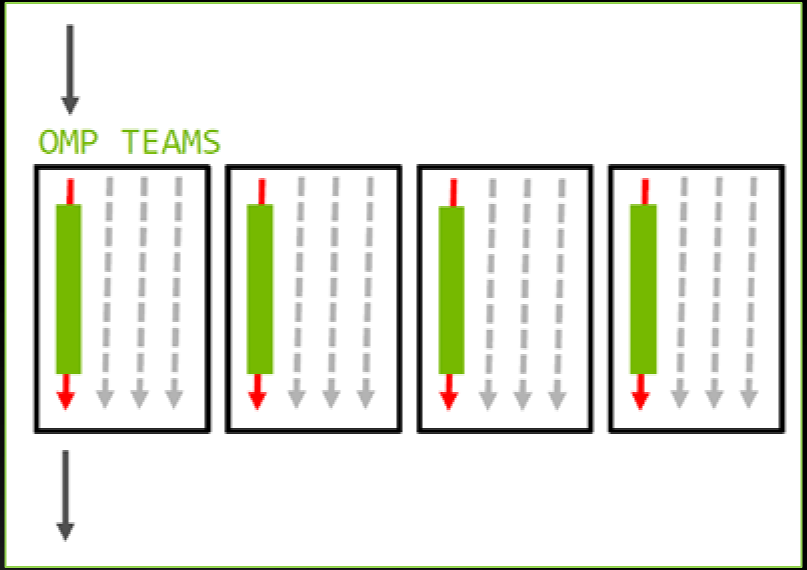
```
!Moves this region of code to the GPU and implicitly maps data.
  !$omp target
     !$omp parallel for
     do i=2,N-1
        ANew(i) = A (i-1) + A(i+1)
     end do
  !$omp end target
```

OPENMP

Teams

Teams Directive

- To better utilize the GPU resources, use many thread teams via the TEAMS directive.
- Spawns 1 or more thread teams with the same number of threads
- Execution continues on the master threads of each team (redundantly)
- No synchronization between teams

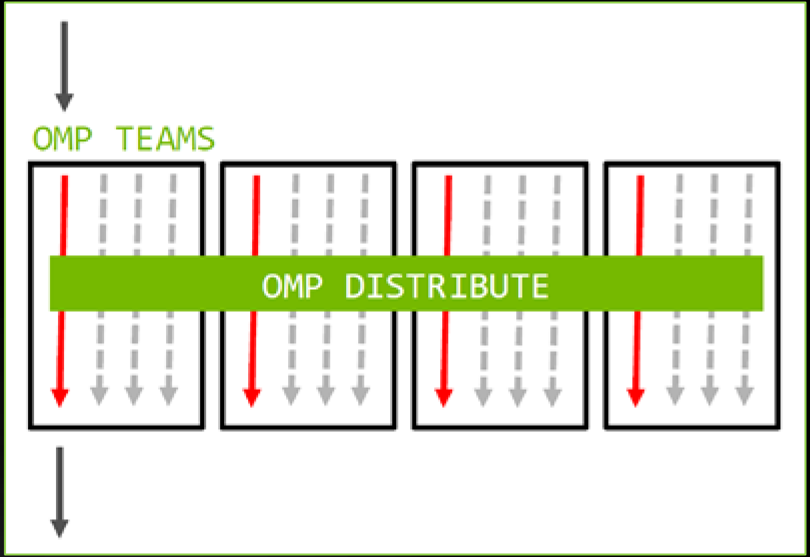


OPENMP

Teams

Distribute Directive

- Distributes the iterations of the next loop to the master threads of the teams.
- Iterations are distributed statically.
- There’s no guarantees about the order teams will execute.
- No guarantee that all teams will execute simultaneously
- Does not generate parallelism/worksharing within the thread teams.



OPENMP

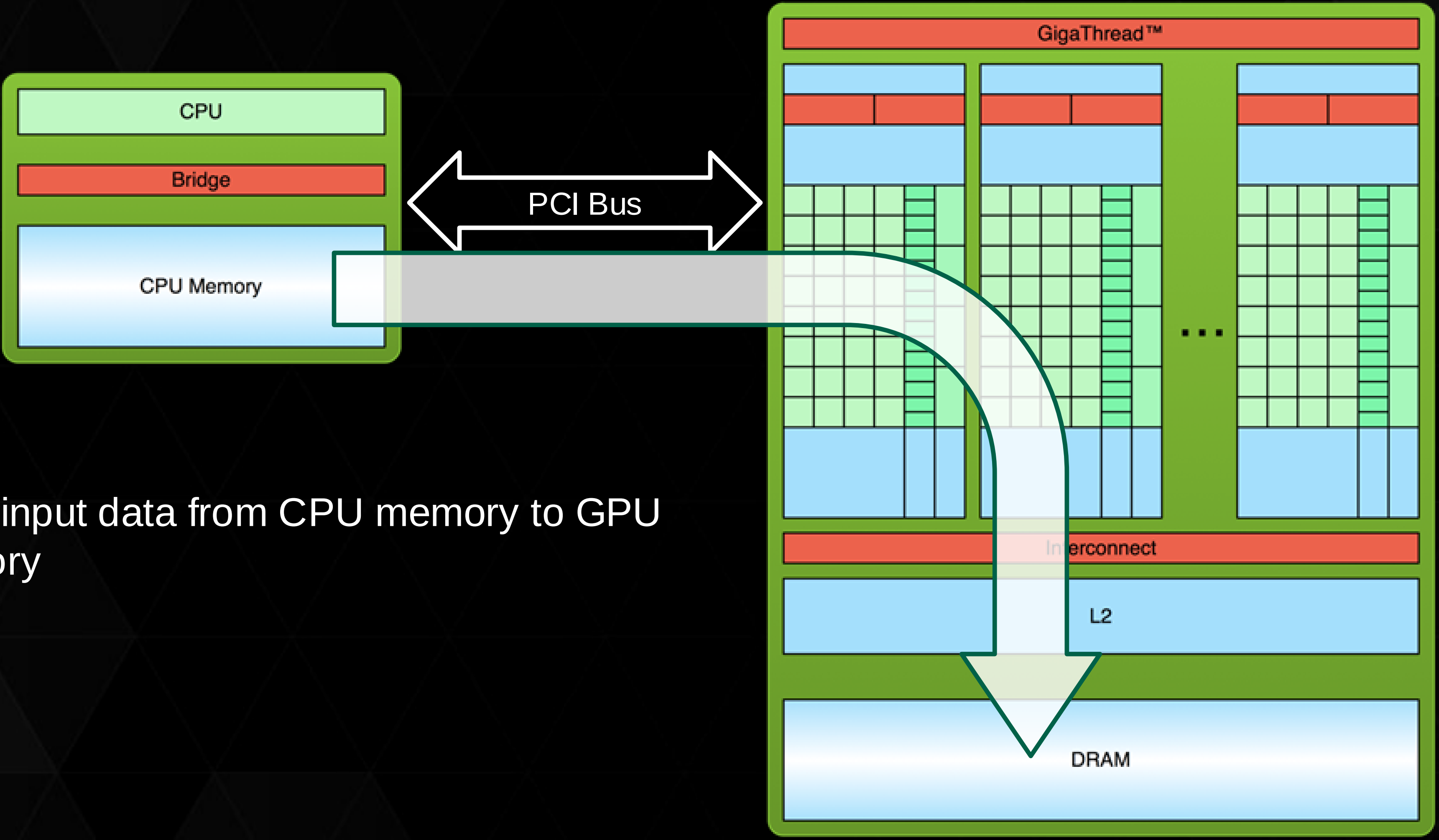
Teams

Loop Directive

- Expose more parallelism in a program is to allow a compiler to do the mapping onto the target architectures
- Similar to OpenACC, compiler translates the parallel region into a kernel that runs in parallel on the GPU
- The programmer specifies the loop regions to be parallelized by the compiler and the compilers parallelize loop across teams and threads using “teams loop” construct

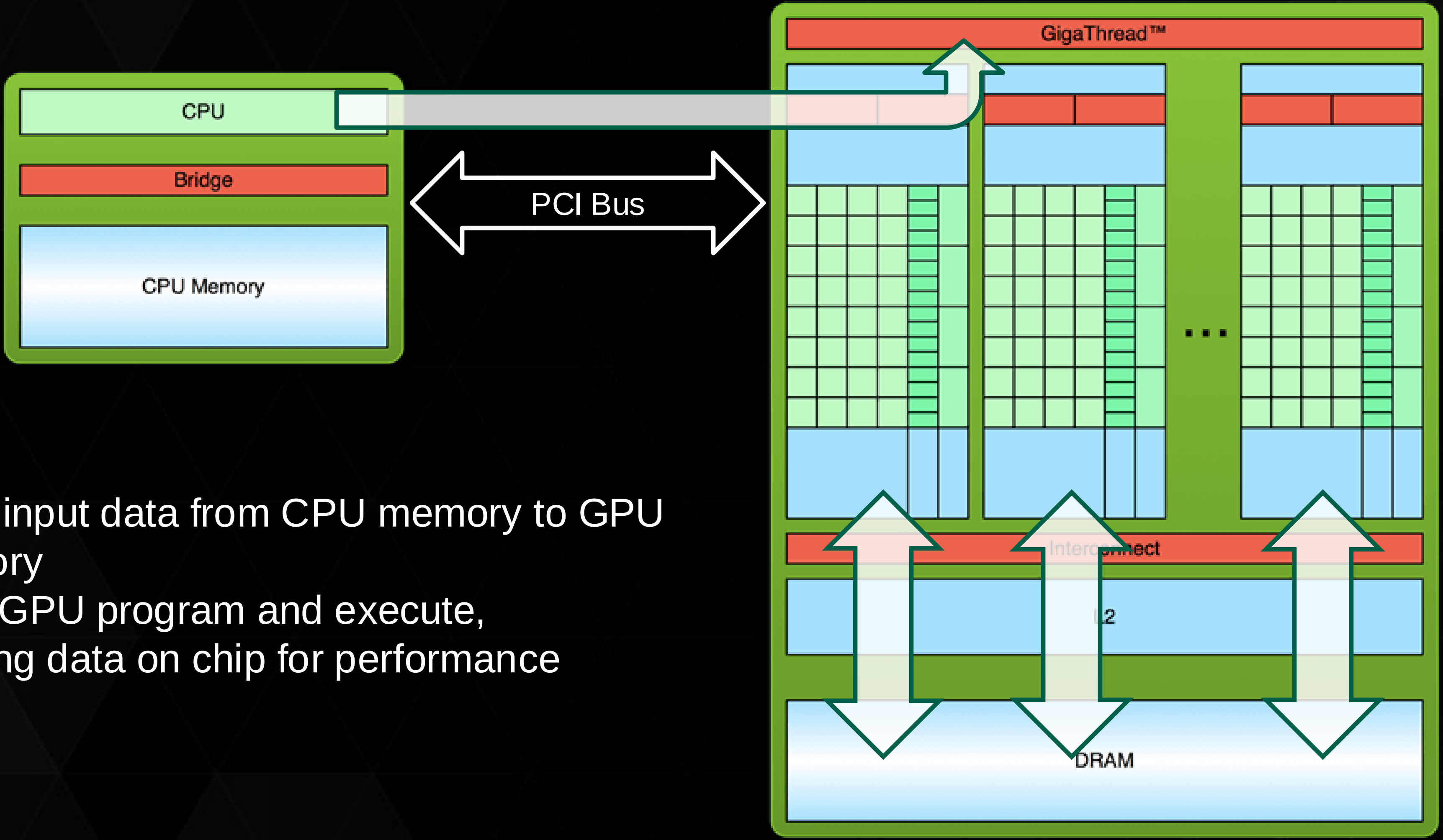
```
#pragma omp target teams loop reduction(max:error)
for( int j = 1; j < n-1; j++) {
  #pragma omp loop reduction(max:error)
  for( int i = 1; i < m-1; i++ ) {
      Anew[j][i] = 0.25f * ( A[j][i+1] + A[j][i-1]
                           + A[j-1][i] + A[j+1][i]);
      error = fmaxf( error, fabsf(Anew[j][i]-
A[j][i]));
  }
}
```

PROCESSING FLOW – STEP 1



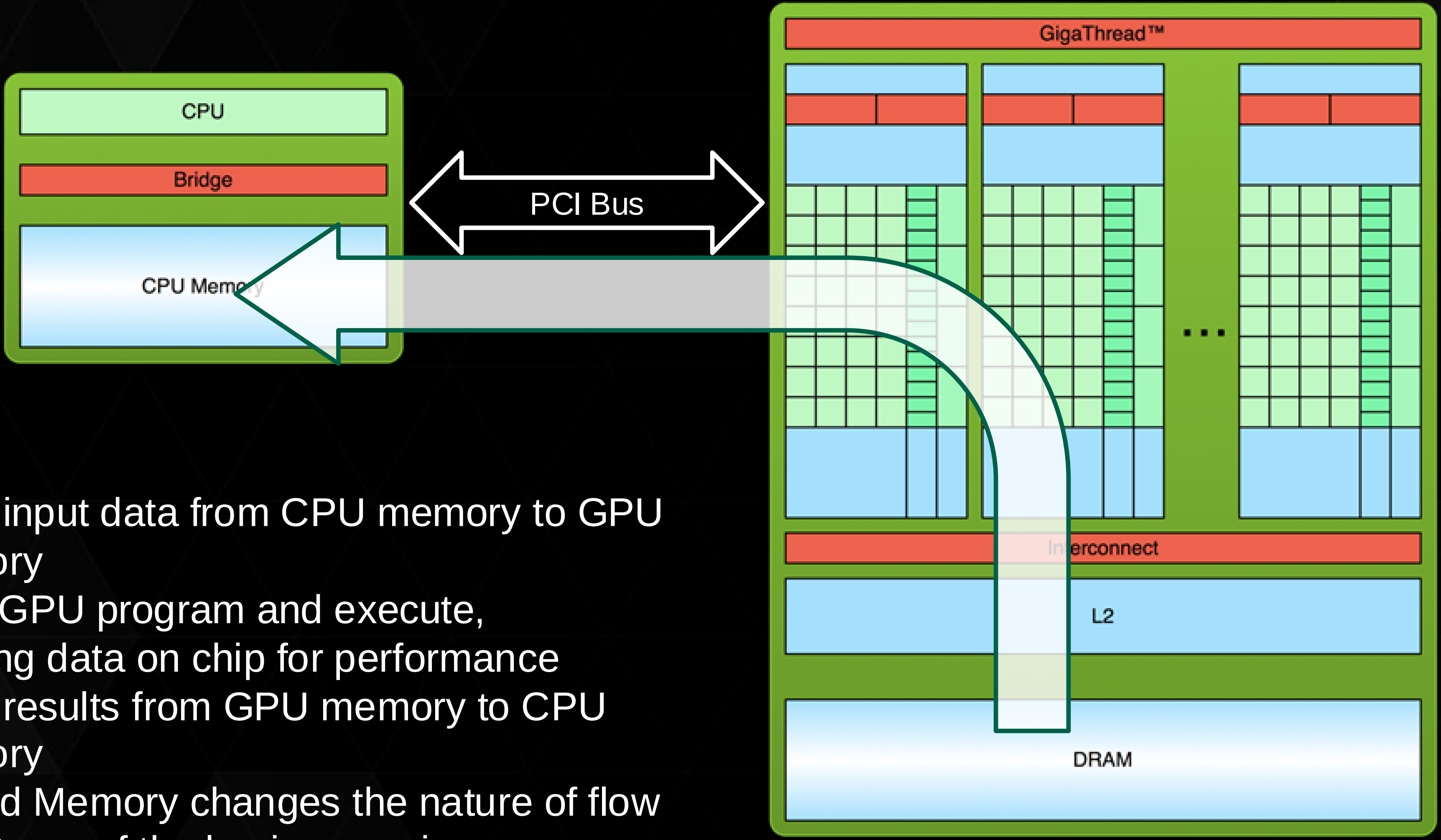
1. Copy input data from CPU memory to GPU memory

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

OPENMP

Data Offloading

TARGET Data Directive

- Offloads data from the CPU to the GPU, but not execution
- The target device owns the data, accesses by the CPU during the execution of contained target regions are forbidden.
- Useful for sharing data between TARGET regions

```
#pragma omp target data map(to:A[:n]) map(from:ANew[:n])
  {
     #pragma omp parallel for
     for( int j = 1; j < n-1; j++) {
        ANew[j] = A [j-1] + A[j+1];
     }
  }
```

```
!$omp target data map(to:A(:)) map(from:ANew(:))
      !$omp parallel for
      do j=2,N-1
         ANew(j) = A (j-1) + A(j+1)
      end do
!$omp end target data
```

Summary

START Offloading "OMP LOOP" Three ways

- `omp target teams loop`
  - Recommended way
  - You can use num_teams and thread_limit clauses
- `omp target loop`
  - Fully automatic
  - You cannot use num_teams / thread_limit
- `omp target parallel loop`
  - Uses only threads, and doesn't use teams
  - Might be useful for light kernels

GPU Porting advice for OpenMP Programmers

- Re-order loops or transpose arrays to enable SIMD/SIMT accesses in outermost loops
- Use collapse(N) directives on loops to increase parallelism
- Replace critical sections with atomics
- Remove all I/O statements
- Remove memory allocation
- Use compiler feedback to identify and factor out unsupported or non-scalable OpenMP constructs and API calls

- Parallelism, Parallelism, Parallelism

Best Practices for OpenMP on GPUs

Use the `teams` and `distribute` directive to expose all available parallelism

Use the `loop` directive when the mapping to hardware isn't obvious

Aggressively `collapse` loops to increase available parallelism

Use the `target data` directive and `map` clauses to reduce data movement between CPU and GPU

…or just skip the `target data` directive and use managed memory

Use OpenMP tasks to go asynchronous and better utilize the whole system

Use host fallback (`if` clause) to generate host and device code

Use accelerated libraries whenever possible

Less is more with the NVIDIA compiler. Being pedantic can reduce performance.

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 OpenMP Target Offload onto GPU
    - The command to compile C code is ‘nvc’
    - The command to compile C++ code is ‘nvc++’
    - The command to compile fortran code is ‘nvfortran’

NVIDIA HPC SDK

- The NVIDIA HPC SDK includes the new NVIDIA HPC compiler supporting OpenMP C and Fortran
  - -mp: compiler switch to enable processing of OpenMP directives and pragmas
  - gpu: OpenMP directives are compiled for GPU execution plus multicore CPU fallback; this Beta feature is supported on Linux/x86 for NVIDIA V100 or later GPUs.
  - multicore: OpenMP directives are compiled for multicore CPU execution only; this sub-option is the default.

```
nvc –mp=gpu main.c
```

```
nvfortran –Minfo=mp –mp=gpu main.f90
```

BUILDING THE CODE

-Minfo shows more details

```
Use of loop in Fortran:

!$omp target teams loop
do n1loc_blk = 1, n1loc_blksize
  do igp = 1, ngpown
    do ig_blk = 1, ig_blksize
      do ig = ig_blk, ncouls, ig_blksize
        do n1_loc = n1loc_blk, ntband_dist, n1loc_blksize
          !expensive computation codes
        enddo
      enddo
    enddo
  enddo
enddo
```

```
$ nvfortran test.f90 -mp=gpu -Minfo=mp
42, !$omp target teams loop
  42, Generating "nvkernel_MAIN__F1L42_1" GPU kernel
      Generating Tesla code
    43, Loop parallelized across teams ! blockidx%x
    44, Loop run sequentially
    45, Loop run sequentially
    46, Loop run sequentially
    47, Loop parallelized across threads(128) !
threadidx%x
  42, Generating Multicore code
    43, Loop parallelized across threads
```

RDF

Pseudo Code - C

```
          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);
```

- Across Frames
- Find Distance
- Reduction

RDF

Pseudo Code -C

```
#pragma omp target data map(d_x[0:nconf*numatm],....)
        for (int frame=0;frame<nconf;frame++){

                #pragma omp target teams distribute parallel for
                for(int id1=0;id1<numatm;id1++)
                {
                        for(int id2=0;id2<numatm;id2++)
                        {
                                …
                                r=sqrtf(dx*dx+dy*dy+dz*dz);

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

- Target Offload construct
- Map data to GPU
- Distribute Inner Loop

- Atomic Construct

RDF

Pseudo Code – Fortran

```
!$omp target data map(x(:,:), y (:,:), z (:,:), g (:))
do iconf=1,nframes
      if (mod(iconf,1).eq.0) print*,iconf
      !$omp target teams distribute parallel do
private(dx,dy,dz,r,ind)
      do i=1,natoms
        do j=1,natoms
          dx=x(iconf,i)-x(iconf,j)
          dy=y(iconf,i)-y(iconf,j)
          dz=z(iconf,i)-z(iconf,j)
                                  ....
                         if(r<cut)then
            !$omp atomic
            g(ind)=g(ind)+1.0d0
          endif
        enddo
      enddo
    enddo
```

- Map data to GPU
- Target Offload construct
- Distribute Inner Loop
- Atomic Construct

PRIVATE CLAUSE

In the C/C++ language it is possible to declare variables inside a lexical scope ; roughly: inside curly braces.

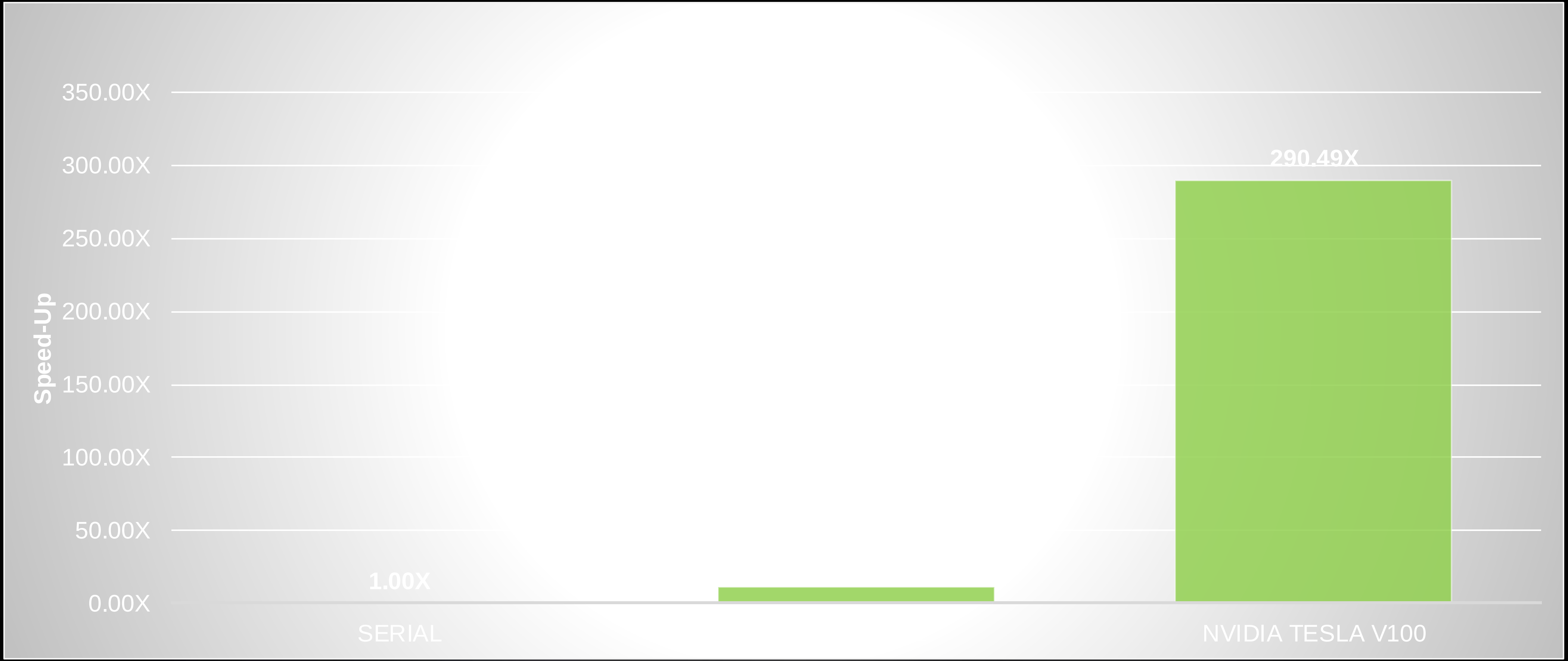
```
int x = 5;
#pragma omp parallel
  {
    int x; x = 3;
    printf("local: x is %d\n", x);
  }
```

This concept extends to OpenMP parallel regions and directives: any variable declared in a block following an OpenMP directive will be local to the executing thread

```
int x = 5;

#pragma omp parallel private(x)
{
    x = x+1; // dangerous
    printf("private: x is %d\n",x);
}
  printf("after: x is %d\n",x); // also
dangerous
```

OPENMP SPEEDUP



HPC SDK 20.11, NVIDIA Tesla V100, DGX1

KNOWN LIMITATIONS

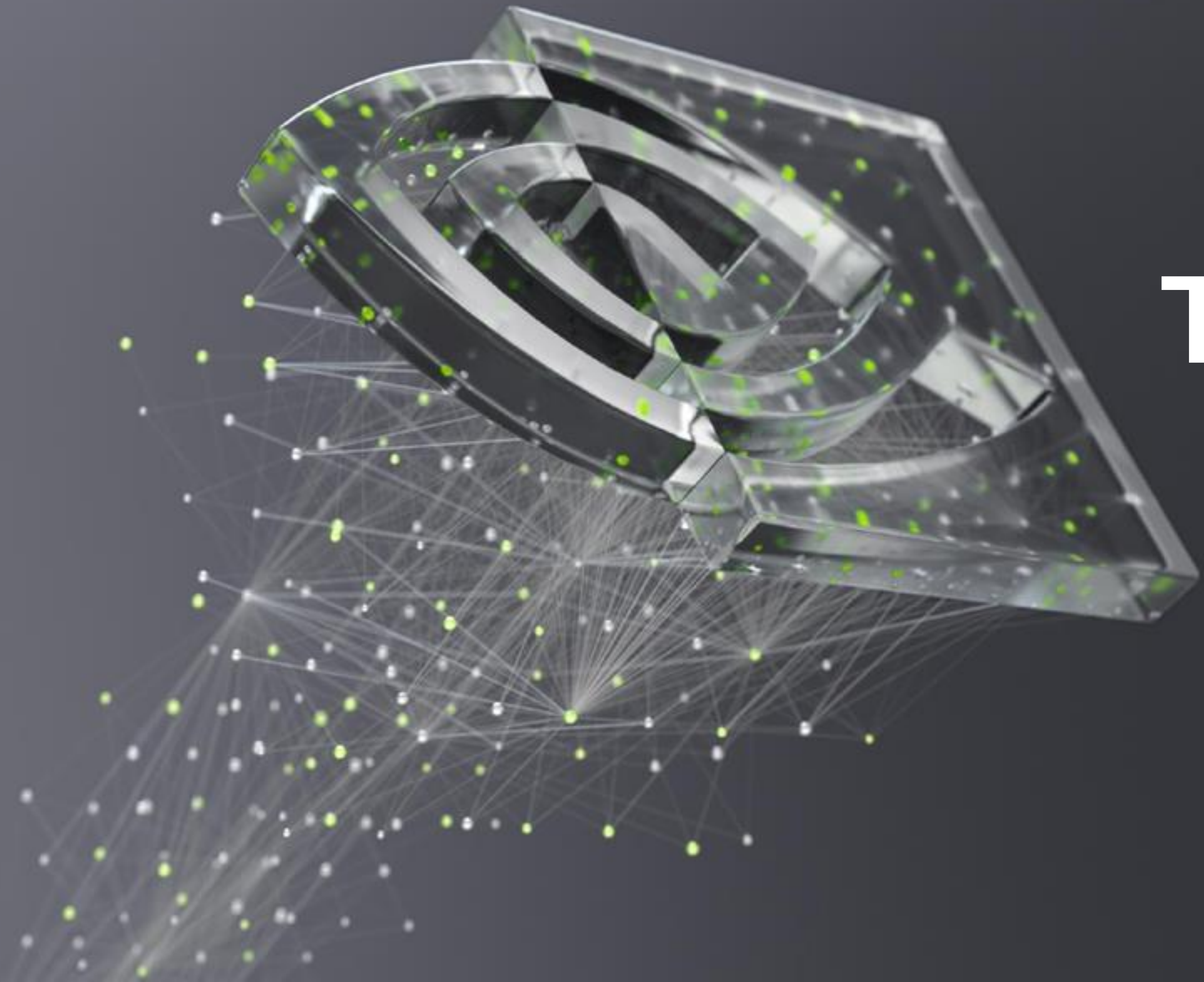
HPC SDK LIMITATION

- Not all functionality associated with loop is supported in the Beta release of OpenMP target offload.
- The compilers support loop regions containing procedure calls as long as the callee does not contain OpenMP directives.

REFERENCES

https://on-demand.gputechconf.com/gtc/2016/presentation/s6510-jeff-larkin-targeting-gpus-openmp.pdf

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



THANK YOU

NVIDIA.