INTRODUCTION TO OPENMP

BRENT LEBACK, MEMBER OF THE NVIDIA HPC SDK TEAM

OPENMP RESOURCES

openmp.org

The OpenMP API specification for parallel programming

OpenMP 5.2 Released with Improvements and Refinements

Syntax is refined and several API features are improved

Latest News

OpenMP ARB
@OpenMP_ARB

Do you want to get up to speed on #OpenMP programming? The following

OPENMP HAS BEEN AROUND A LONG TIME, BUT...

Back to our Day 1 discussion on differences between CPU and GPU

OpenMP

Fork-join model

Will your legacy OpenMP code perform well on the GPU?

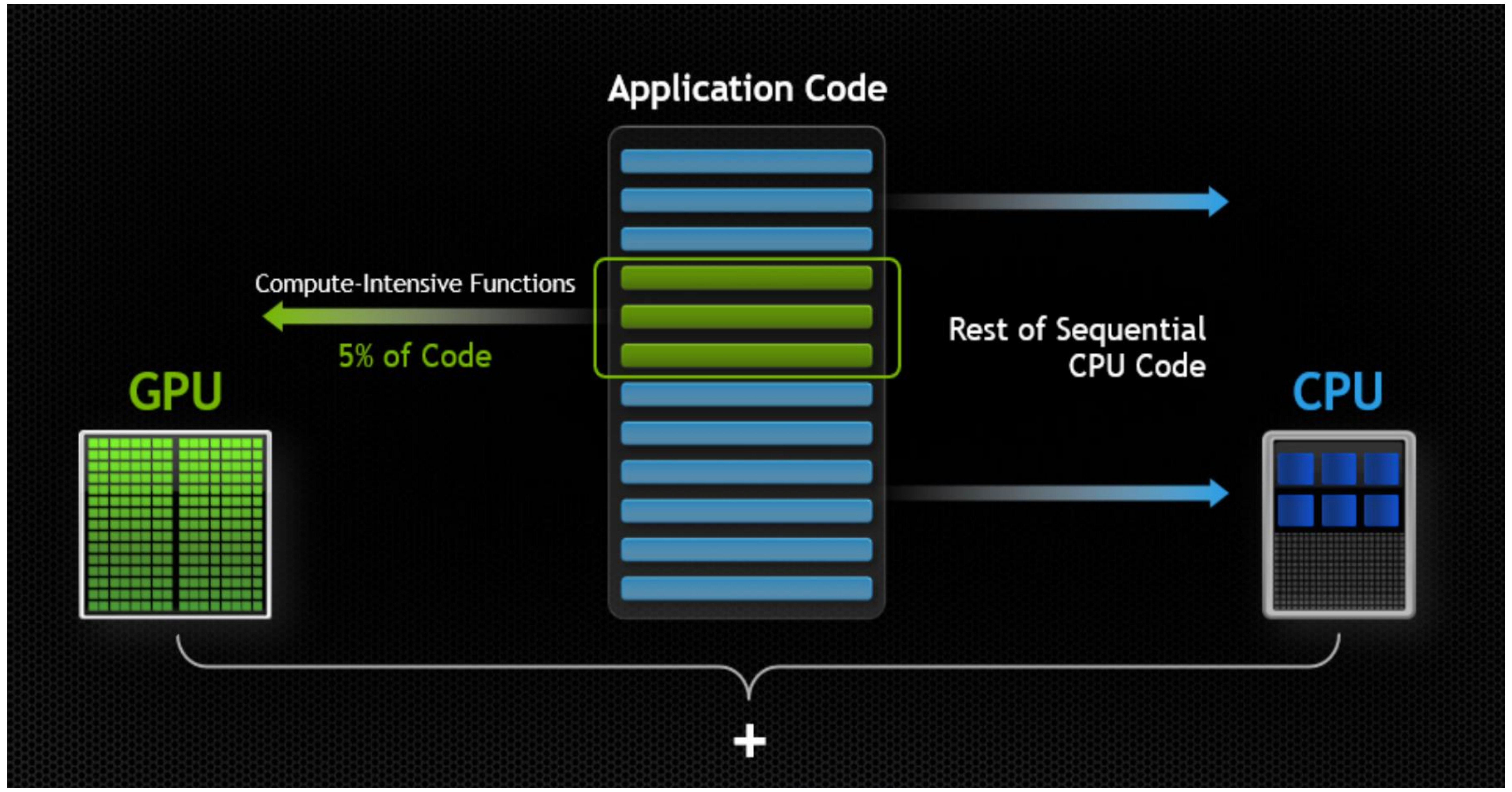
NO

```
#pragma omp parallel for
for (int i = 0; i < N; ++i) {
    compute();
}
```

BASIC SYNTACTIC CONCEPTS

Directive-Based API Designed for Parallel Programming

- C/C++ OpenMP pragma syntax
  - `#pragma omp` *`directive [clause]... eol`*
  - `_Pragma("`*`omp directive …`*`")`
  - continue to next line with backslash
- Fortran OpenMP directive syntax
  - `!$omp` *`directive [clause]...`*
  - `&` continuation
  - Fortran-77 syntax rules
    - `!$omp` or `C$omp` or `*$omp` in columns 1-5
    - continuation with nonblank in column 6
- Target, Teams, and Distribute Constructs are the major additions for GPU acceleration
  - Target starts the offload, maps variables to the device, and executes the construct on the device
  - Teams creates the teams for execution
  - Distribute says to workshare amongst the teams
  - Parallel creates the (CUDA) threads within the team
  - do/for says to workshare amongst the threads



OPENMP TARGET+ CONSTRUCTS

The first attempt might be to just add “target teams distribute” where you already have OpenMP directives:

```
#pragma omp target teams distribute parallel for reduction(max:error)
        for( int j = 1; j < SZ-1; j++) {
            for( int i = 1; i < SZ-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]));
            }
        }
```

```
 66, #omp target teams distribute parallel for
        66, Generating "nvkernel_main_F1L66_3" GPU kernel
            Loop parallelized across teams and threads(128), schedule(static)
            Generating reduction(max:error)
```

OPENMP TARGET+ CONSTRUCTS

To obtain more parallelism (and get coalesced accesses) you can break up the directive, and use “parallel for” on the inner loop:

```
#pragma omp target teams distribute reduction(max:error)
        for( int j = 1; j < SZ-1; j++) {
#pragma omp parallel for reduction(max:error)
            for( int i = 1; i < SZ-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]));
            }
        }
```

```
66, #omp target teams distribute
        66, Generating "nvkernel_main_F1L66_3" GPU kernel
            Generating reduction(max:error)
            Loop parallelized across teams, schedule(static)
        69, Team private (j) located in CUDA shared memory
            #omp parallel
          69, Generating reduction(max:.error1714p)
              Loop parallelized across threads, schedule(static)
```

OPENMP TARGET+ CONSTRUCTS

Unfortunately, this is not what you want to do on the CPU. On the CPU, we generate 1 team by default; no parallelization across the outer loop.

```
#pragma omp target teams distribute reduction(max:error)
        for( int j = 1; j < SZ-1; j++) {
#pragma omp parallel for reduction(max:error)
            for( int i = 1; i < SZ-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]));
            }
        }
```

```
 66, #omp target teams distribute
        66, Generating reduction(max:error)
            Loop parallelized across teams, schedule(static)
```

OPENMP SOLVING THE PORTABILITY ISSUE

One method in OpenMP is to use a meta-directive. We recommend you use the target_device option. This is not available until our 22.2 release.

```
#pragma omp metadirective \
    when(target_device={kind(gpu)} : target teams distribute reduction(max:error)) \
        default( parallel for reduction(max:error))
    for( int j = 1; j < SZ-1; j++) {
#pragma omp metadirective \
    when(target_device={kind(gpu)} : parallel for reduction(max:error))
        for( int i = 1; i < SZ-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]));
        }
    }
```

```
68, #omp target teams distribute  // compiled with –mp=gpu
        68, Generating reduction(max:error)
            Loop parallelized across teams, schedule(static)
68, #omp parallel                // compiled with –mp=multicore
        68, Generating reduction(max:error)
```

OPENMP LOOP DIRECTIVE

We provide a more descriptive OpenMP loop construct. It has some restrictions on what can be within a "loop", but states the iterations may execute concurrently, more like OpenACC.

```
#pragma omp target teams loop
  for (int i=0; i<N; i++) {
    y[i] = a*x[i] + y[i];
  }

<alternatively>

#pragma omp target teams
{
  #pragma omp loop
  for (int i=0; i<N; i++) {
    y[i] = a*x[i] + y[i];
  }
}  //end target teams in Fortran
```

1 parallel kernel

Similar to OpenACC, compiler translates the parallel region into a kernel that runs in parallel on the GPU

OPENMP TARGET TEAMS LOOP CONSTRUCT

No need for metadirectives or ifdef macros. Provides target-specific parallelism in the same executable.

```
#pragma omp target teams loop reduction(max:error)
        for( int j = 1; j < SZ-1; j++) {
#pragma omp loop reduction(max:error)
            for( int i = 1; i < SZ-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]));
            }
        }
```

```
 66, Generating NVIDIA GPU code
          66, Loop parallelized across teams /* blockIdx.x */
          69, Loop parallelized across threads(128) /* threadIdx.x */
          66, Generating reduction(max:error)
 66, Generating Multicore code
          66, Loop parallelized across threads
```

BIND (TEAMS | PARALLEL | THREAD) CLAUSES ON OMP LOOP

The developer can instruct the compiler which levels of parallelism to use on given loops by adding clauses:

- teams – Mark this loop for work-shared execution by all the teams
- parallel – Mark this loop for execution by all the threads in a team
- thread – Mark this loop for execution by a single thread

There is only an implicit barrier between bind(parallel) loops

```
#pragma omp loop bind(teams)
for( i = 0; i < size; i++ )
  #pragma omp loop bind(parallel)
  for( j = 0; j < size; j++ )
    #pragma omp loop bind(thread)
    for( k = 0; k < size; k++ )
      c[i][j] += a[i][k] * b[k][j];
```

ADJUSTING THE KERNEL LAUNCH PARAMETERS IN OPENMP

Useful when you know more about the loop bounds than the compiler

The compiler will choose a number of teams and threads, for each kernel, for you, but you can change it with clauses.

Our reductions may use atomic operations, and you might find limiting the number of teams gives better performance.

num_teams(N) – Generate N teams for this target region

thread_limit(Q) – Generate only Q threads per team

```
 . . .
  int b[10][32];
 . . .
#pragma omp target teams loop thread_limit(32) \
                         num_teams(N) map(tofrom: b)
  for (i = 0; i < N; ++i)
#pragma omp loop bind(parallel)
      for (j = 0; j < 32; ++j)
          b[i][j] = i;

Minfo:
 10, Generating "nvkernel_main_F1L10_1" GPU kernel
     Generating NVIDIA GPU code
     10, Loop parallelized across teams(N) /* blockIdx.x */
     12, Loop parallelized across threads(32) /* threadIdx.x */
```

OPENMP COLLAPSE CLAUSE

The same as OpenACC

Collapse(n): Applies the associated directive to the following *n* tightly nested loops

Useful when loop extents are short, or there are more loops than levels (teams, parallel) available

```
#pragma omp target teams
#pragma omp loop collapse(2)
for (int i=0; i<N; i++)
  for (int j=0; j<N; j++)
    ...
```

$\Rightarrow$

```
#pragma omp target teams
#pragma omp loop
for (int ij=0; ij<N*N; ij++)
    i = ij / N;
    j = ij % N;
    ...
```

CALLING USER ROUTINES IN DEVICE CODE

OpenACC is more explicit than OpenMP

```
! OpenACC
 real function fs(a)
  !$acc routine seq
  fs = a + 1.0
  end function

 subroutine fv(a,j,n)
  !$acc routine vector
  real :: a(n,n)
  !$acc loop vector
  do i = 1, n
   a(i,j) = fs(a(i,j))
  enddo
 end subroutine

 subroutine fg(a,n)
  !$acc routine gang
  real :: a(n,n)
  !$acc loop gang
  do j = 1, n
   call fv(a,j,n)
  enddo
 end subroutine

 !$acc parallel num_gangs(100) vector_length(32)
   call fg(a,n)
 !$acc end parallel
```

```
! OpenMP
 real function fs(a)
  !$omp declare target
  fs = a + 1.0
  end function

 subroutine fv(a,j,n)
  !$omp declare target
  real :: a(n,n)
  do i = 1, n
  !$omp parallel do
   a(i,j) = fs(a(i,j))
  enddo
 end subroutine

 subroutine fg(a,n)
  !$omp declare target
  real :: a(n,n)
  do j = 1, n
   call fv(a,j,n)
  enddo
 end subroutine
```

```
NVFORTRAN-F-1196-OpenMP - Standalone 'omp parallel' in a 'declare target'
routine is not supported yet.
NVFORTRAN/x86-64 Linux 21.11-0: compilation aborted
```

REDUCTION CLAUSE

OpenACC borrowed heavily from OpenMP, which didn’t change

The **reduction** clause takes many values and “reduces” them to a single value, such as in a sum or maximum

Each thread calculates its part

Reductions can be over all teams in the kernel, or within a team

The compiler will perform a final reduction to produce a **single result** using the specified operation

```
for( i = 0; i < size; i++ )
  for( j = 0; j < size; j++ )
    for( k = 0; k < size; k++ )
      c[i][j] += a[i][k] * b[k][j];
```

```
#pragma omp target teams loop
for( i = 0; i < size; i++ )
  for( j = 0; j < size; j++ )
    double tmp = 0.0f;
    #pragma omp loop reduction(+:tmp)
    for( k = 0; k < size; k++ )
      tmp += a[i][k] * b[k][j];
    c[i][j] = tmp;
```

ATOMIC OPERATIONS

Also very similar between OpenACC and OpenMP

The **atomic** construct ensures that a specific storage location is accessed and/or updated atomically, preventing simultaneous (and indeterminate) reading and writing by threads.

The “atomic update” is most commonly used.

Hackathons have shown the need for double complex atomic updates, but for most purposes it is okay to do the real and imaginary parts separately.

```
!$omp target teams loop private(j)
 do i = 1, n
   j = mod(f(i),m)
   !$omp atomic update
     r(j+1) = r(j+1) + s(i)
   !$omp end atomic
 enddo
```

“DESCRIPTIVE” OPENMP LOOP VS. “PRESCRIPTIVE” OPENMP

Why we encourage users to try the loop construct

OpenMP loop more-directly leverages years of OpenACC scheduling/kernel generation

We don’t allow many parallelism-limiting directives in OpenMP loop:

- master, single, barrier, etc.
- OpenMP API calls

We don’t need to insert our device-side OpenMP runtime support into the generated kernels with OpenMP loop.

The CUDA toolchain can do a good job or removing or at least minimizing the overhead, but we have not gained a lot of experience yet with complicated kernels.

DEFINING DATA REGIONS

All OpenMP data directives are very similar in form and function to OpenACC

The **target data** construct defines a region of code in which GPU arrays remain on the GPU and are shared among all kernels in that region.

```
#pragma omp target data
{
  #pragma omp target teams loop
  ...

  #pragma omp target teams loop
  ...
}
```

Arrays used within the data region will remain on the GPU until the end of the data region.

DATA CLAUSES

Again, similar to OpenACC, with some additions

map (tofrom: *list* **)**	Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.
map (to: *list* **)**	Allocates memory on GPU and copies data from host to GPU when entering region.
map (from: *list* **)**	Allocates memory on GPU and copies data to the host when exiting region.
map (alloc: *list* **)**	Allocates memory on GPU but does not copy.
map (always, to: *list***)**	Allocates memory on the GPU if it is not there. Always copy data from the host to the GPU.
map (always, from: *list***)**	Allocates memory on the GPU if it is not there. Always copy data to the host when exiting region.

UNSTRUCTURED DATA DIRECTIVES

Basic Example

enter data: Defines the start of an unstructured data region

clauses: map(to: list), map(alloc: list)

exit data: Defines the end of an unstructured data region

clauses: map(from: list), map(delete: list), map(release: list)

```
#pragma omp target enter data map(to:a[0:N],b[0:N]) map(alloc:c[0:N])

  #pragma omp target teams distribute parallel loop
  for(int i = 0; i < N; i++){
    c[i] = a[i] + b[i];
  }

#pragma omp target exit data map(from: c[0:N]) map(delete: a,b)
```

OPENMP TARGET UPDATE DIRECTIVE

Update: Explicitly transfers data between the host and the device

Always updates, not a “present_or” operation

Useful when you want to update data in the middle of a data region

Clauses:

target update to(): copies from the host to the device

target update from(): copies data from the device to the host

```
#pragma omp target update from(x[0:count])
MPI_Send(x,count,datatype,dest,tag,comm);
```

ARRAY SHAPING

When the compiler fails to properly determine the size of arrays

Sometimes the compiler cannot determine the size of array

Examine the –Minfo output!

Developers must specify sizes explicitly using data clauses and array “shape”

C

```
#pragma omp target data map(to:a[0:size]), map(from:b[s/4:3*s/4])
```

Numbers in brackets are starting-element : number-of-elements

Fortran

```
!$omp target data map(to:a(1:end)), map(from:b(s/4+1:3*s/4))
```

Numbers in parenthesis are starting-element : ending-element

SUMMARY: BASIC USE OF DATA DIRECTIVES IN OPENACC AND OPENMP

more similar than different

```
! OpenACC
!$acc data <clause>  ! Starts a structured data region

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

  copyin(list) Allocates memory on the 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.

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

!$acc enter data <clause>  ! Starts unstructured data region.
  clause can be copyin or create

!$acc exit data <clause>  ! Ends unstructured data region.
  clause can be copyout or delete

!$acc update [host|self|device](list)
```

```
! OpenMP
!$omp target data<clause>  ! Starts a structured data region

  map(tofrom:list) Allocates memory on the GPU and copies data
from host to GPU when entering region and copies data to the host
when exiting region.

  map(to:list) Allocates memory on the GPU and copies data from
host to GPU when entering region

  map(from:list) Allocates memory on GPU and copies data to the
host when exiting region.

  map(alloc:list)  Allocates memory on GPU but does not copy.

!$omp target enter data <clause>  ! Starts unstructured data
region.
  clause can be map(to:) or map(alloc:)

!$omp target exit data <clause>  ! Ends unstructured data region.
  clause can be map(from:) or map(delete:)

!$omp target update [to|from](list)
```

ASYNCHRONOUS BEHAVIOR, QUEUES, DEPENDENCIES, STREAMS

1-1 correspondence between OpenACC async numbers and streams. OpenMP is WIP.

```
! OpenACC
 !$acc data create(a, b, c)

  ierr = cufftPlan2D(iplan1,n,m,CUFFT_C2C)
  ierr = cufftSetStream(iplan1,acc_get_cuda_stream(10))

  !$acc update device(a) async(10)

  !$acc host_data use_device(a,b,c)
  ierr = ierr + cufftExecC2C(iplan1,a,b,CUFFT_FORWARD)
  ierr = ierr + cufftExecC2C(iplan1,b,c,CUFFT_INVERSE)
  !$acc end host_data

  ! scale c
  !$acc kernels async(10)
  c = c / (m*n)
  !$acc end kernels

  !$acc update host(c) async(10)
  !$acc wait(10)

  ! Check inverse answer
  write(*,*) 'Max error C2C INV: ', maxval(abs(a-c))

  !$acc end data
```

```
! OpenMP
 !$omp target enter data map(alloc:a,b,c)

  ierr = cufftPlan2D(iplan1,n,m,CUFFT_C2C)
  nstream = omp_get_cuda_stream(omp_get_default_device(), .true.)
  ierr = cufftSetStream(iplan1,nstream)

  !$omp target update to(a) depend(out:nstream) nowait

  !$omp target data use_device_ptr(a,b,c)
  ierr = ierr + cufftExecC2C(iplan1,a,b,CUFFT_FORWARD)
  ierr = ierr + cufftExecC2C(iplan1,b,c,CUFFT_INVERSE)
  !$omp end target data

  ! scale c
  !$omp target teams distribute depend(inout:nstream) nowait
  do j = 1, n
    !$omp parallel do
    do i = 1, m
      c(i,j) = c(i,j) / (m*n)
    end do
  end do

  !$omp target update from(c) depend(in:nstream) nowait
  !$omp taskwait

  !$omp target exit data map(delete:a,b,c)
```

PASSING DEVICE POINTERS TO CUDA LIBRARIES IN OPENACC AND OPENMP

Getting the compiler to pass the device pointer within a data region

```
! OpenACC
 use curand
 integer, parameter :: N=10000000, HN=10000
 integer            :: a(N), h(HN), i
 type(curandGenerator) :: g

 istat = curandCreateGenerator(g,CURAND_RNG_PSEUDO_XORWOW)

 !$acc data create(a)

 !$acc host_data use_device(a)
 istat = curandGenerate(g, a, N)
 !$acc end host_data
```

```
! OpenMP
  use curand
  integer, parameter :: N=10000000, HN=10000
  integer            :: a(N), h(HN), i
  type(curandGenerator) :: g

  istat = curandCreateGenerator(g,CURAND_RNG_PSEUDO_XORWOW)

  !$omp target data map(alloc:a)

  !$omp target data use_device_ptr(a)
  istat = curandGenerate(g, a, N)
  !$omp end target data
```

FORTRAN ARRAY SYNTAX IN DEVICE CODE

Currently not available in our OpenMP compiler, would require support for workshare in target regions

```
! OpenACC
  use curand
  integer, parameter :: N=10000000, HN=10000
  integer            :: a(N), h(HN), i
  type(curandGenerator) :: g

  istat = curandCreateGenerator(g,CURAND_RNG_PSEUDO_XORWOW)

  !$acc data create(a) copyout(h)

  !$acc host_data use_device(a)
  istat = curandGenerate(g, a, N)
  !$acc end host_data

  !$acc kernels
  a = mod(abs(a),HN) + 1
  !$acc end kernels

  !$acc kernels
  h(:) = 0
  !$acc end kernels
```

```
! OpenMP
  use curand
  integer, parameter :: N=10000000, HN=10000
  integer            :: a(N), h(HN), i
  type(curandGenerator) :: g

  istat = curandCreateGenerator(g,CURAND_RNG_PSEUDO_XORWOW)

 !$omp target data map(alloc:a) map(from:h)

  !$omp target data use_device_ptr(a)
  istat = curandGenerate(g, a, N)
  !$omp end target data

  !$omp target teams loop
  do idum=1,1
  a = mod(abs(a),HN) + 1
  end do

  !$omp target teams loop
  do idum = 1, size(h)
    h(i) = 0
  end do
```

NVIDIA HPC SDK OPENMP WORK IN PROGRESS

Things we are working on for our 22.2 release

Array reductions (in both OpenMP and OpenACC)

Target/task/nowait (and how that maps to CUDA streams)

Support for Orphaned Parallel (A parallel loop in a user function called from a kernel)

Metadirectives

Performance, performance, performance

NVIDIA®