OpenMP Offloading Outline

- *HPC GPU Computing*
  - GPU Architectures
- Offload basics
  - Fundamentals
  - OpenMP Target Offload Directive
- Data Management (motion)
  - Mapping variables
  - Mapping pointers and pointee data, alloc map-type
  - Functions and Static Variable in external Compile Units
  - Persistence/Updating
  - Direct Allocation on Devices
- **Advanced Features and more on teams distribute parallel**
  - **Reductions, Mapper**
  - **Async Offloading**
  - **function variants**
  - **target teams distribute parallel for|do**

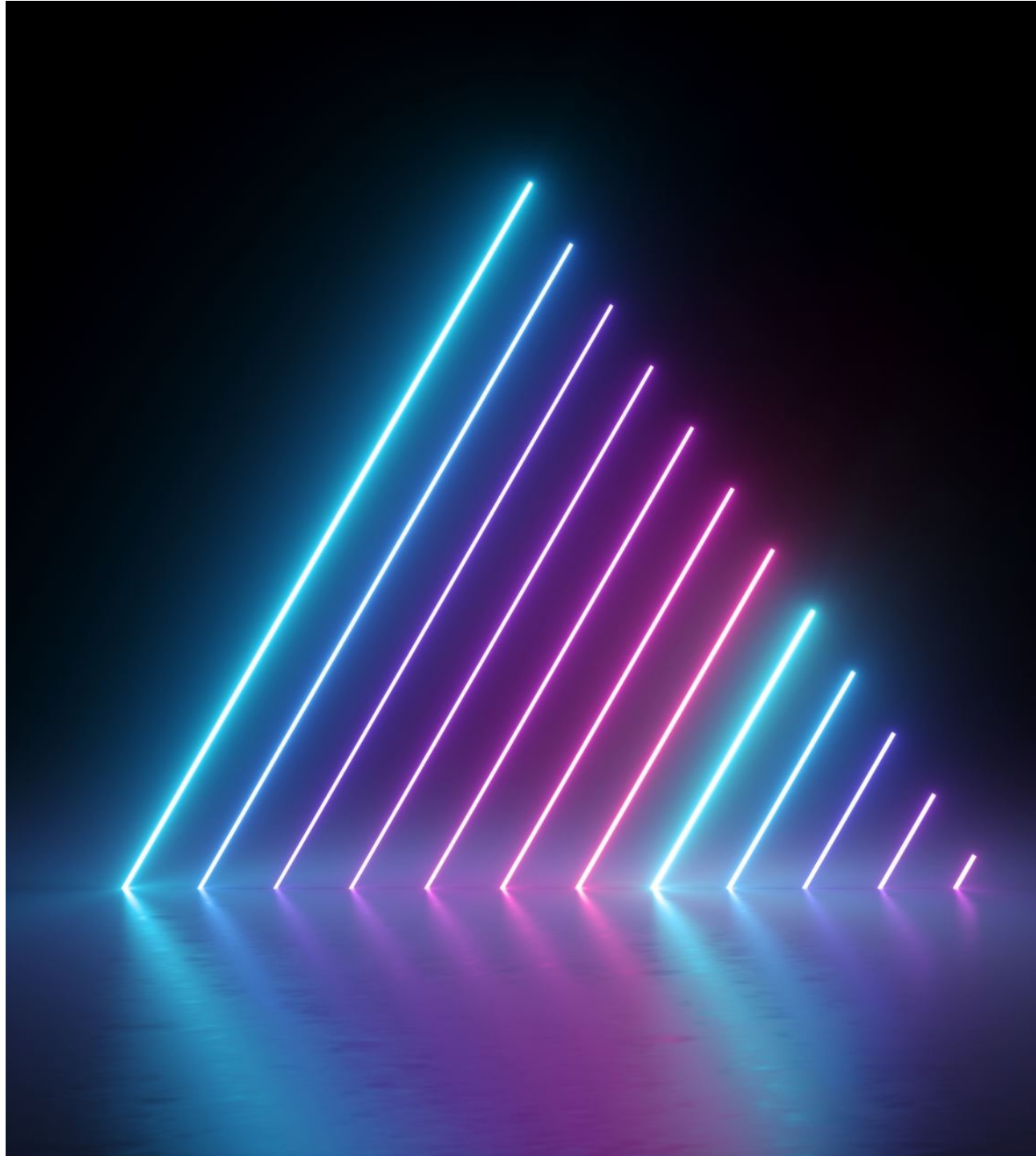
Advanced OpenMP Topics and GPU Programming with OpenMP

Advanced Offload

by

Kent Milfeld (TACC)

Emanuele Vitali (CSC)



Section Objective

- Reduction in target regions
- How to predefine mappers for structures & derived types
- How to asynchronously move data while processing on host
- How to prescribe ordering target constructs.
- Learn more details on teams distribute parallel for | do
- How to specify function calls that match a system's library (nvidia/amd/intel/...)

Mappers

The declare **mapper** directive prescribes a map for later use.

Restricted use: structs, unions, classes (C or C++) & derived-types (F)

It is useful for pre-defining partitioned and nested structure elements.

declare mapper syntax

Syntax:

C/C++

```
declare mapper([mapper_id:] type var) map() [,clauses])
```

F90

```
declare mapper([mapper_id:]type::var) map() [,clauses])
```

mapper_id	base-language identifier or ***default***
type	valid type in scope
var	valid base-langue identifier
clauses	5.0 map(alloc|to|tofrom|from) + always and close modifiers

The **type** must be of struct, union or class type in C and C++ or a non-intrinsic type in Fortran.

declare mapper example

```
#define N 100;

typedef struct myvec{ size_t len; double *data;} myvec_t;

#pragma omp declare mapper(myvec_t v) \
                    map(v, v.data[0:v.len])  //map dyn. storage too
void init(myvec_t *s);

int main(){
   myvec_t s;

   s.data = (double *)calloc(N,sizeof(double));
   s.len  = N;

   #pragma omp target
   init(&s);
}
```

declare mapper example description

- Any variable of type **myvec_t** will have the default map-type specified by the **map** clause.
- The variable **v** is used to reference elements within the map clause.

```
typedef struct myvec{ size_t len; double *data;} myvec_t

#pragma omp declare mapper(myvec_t v) \
                    map(v, v.data[0:v.len])
.
   myvec_t s;

   #pragma omp target  // user-define map implicitly used here
   {
     // use s here
   }
```

declare mapper example with mapper-identifier

```
#define N 100

typedef struct dzmat{double r_m[N][N];double i_m[N][N];} dzmat_t;

  #pragma omp declare mapper( top_id: dzmat_t v) \
                      map(v.r_m[0:N/2][0:N],     \
                          v.i_m[0:N/2][0:N]      )

  void dzmat_init(dzmat_t *z, int is, int ie, int n);
int main(){
  dzmat_t a;
  int is=0, ie=N/2-1;

  #pragma omp target map(mapper(top_id), tofrom: a)
  dzmat_init(&a,is,ie,N);   //init top N/2 rows on device
}
```

Device Reductions

Reductions on devices

Each team of a league has a different contention group, meaning the threads of one team act independently of threads of another team.

To provide reductions across the teams it is necessary to include a reduction clause for a ***target teams*** construct. This is in addition to providing the reduction clause on a (nested) ***parallel for I do***. -- the same reduction behavior occurs across teams.**

** After the end of the region, the original list item is updated with the values of the private copies using the combiner associated with the reduction-identifier.

reduction syntax (refresh)

Syntax:

```
reduction([modifier,] identifier : list)
```

identifier, list items

- Directives that support reduction: parallel, teams, sections, for | do, taskloop, loo
- Identifier applies operation (+,*, ...) or user defined reduction id as operation
  local private copies of variables (list item) are created for each thread
  initialization value depends upon operation
- Variable needs to be shared in enclosing region
  at barrier local copies (partials) are combined with the original shared variable

modifier

task	for reduction across tasks
default	See Spec
inscan	for scan computation

teams reduction example

C/C++

```
float dotprod(float B[], float C[], int n) {
  float sum = 0.0f; int i;
  #pragma omp target teams map(to: B[0:n], C[0:n]) \
                           map(tofrom:sum) reduction(+:sum)
  #pragma omp distribute parallel for reduction(+:sum)
  for (i=0; i<n; i++){ sum += B[i] * C[i]; }
  return sum;
}
```

F90

```
function dotprod(B,C,N) result(sum)
   real    :: B(N), C(N), sum=0.0e0
   integer :: N, i;
   !$omp target teams map(to: B, C) reduction(+:sum) &
   !$omp&           map(tofrom:sum)
   !$omp distribute  parallel  do   reduction(+:sum)
      do i = 1,N; sum=sum + B(i) * C(i); end do
   !$omp end target teams
end function
```

function variant

variants motivation

C/C++

```
#pragma omp parallel
vxv(v1,v2,v3,N);

#pragma omp target teams map(to: v1[:N],v2[:N]) map(from: v3[:N])
vxv(v1,v2,v3,N);

vxv(v1,v2,v3,N);
```

One may want a *base* function executed by different OpenMP directives and the functions may need specific OpenMP-appropriate operations (or even compute statements) in the function that *match* the directive used. (Of course #ifdefs could be used.) It makes sense to provide a mechanism to overload a *base* function. The ***declare variant*** directive serves this purpose.

variants scoring

For now (until 6.0), use variant matching that provides an exclusive outcome (only one is selected) and has a clear winner, or that has a set which is a superset of all others (the winner).

The matching score algorithm for construct contexts will probably be updated in 6.0 because of complications with nested constructs and the definition of of set context-matching constructs, but allowing all constructs in the scoring algorithm.

declare variant directive

Syntax:

C/C++ `declare variant(` ***`variant-name`*** `)` ***`clause`***

F90 `declare variant(`*`[base-name:]`*`variant-name)` ***`clause`***

variant-name

name of a function/procedure* variant (base language identifier; or C++ template-id).

clause

match(*context-selector-specification***)**

C/C++ declare before base function

F90 declare in specification of part of procedure (unless base-name function not specified)

variant C/C++ example

C/C++

```
void p_vxv(int *v1,int *v2,int *v3,int n);
void t_vxv(int *v1,int *v2,int *v3,int n);

#pragma omp declare variant( p_vxv ) match( construct={parallel} )
#pragma omp declare variant( t_vxv ) match( construct={target}   )
void vxv(int *v1,int *v2,int *v3,int n)                    // base function
     { for (int i= 0; i< n; i++)  v3[i] = v1[i] * v2[i]; }

void p_vxv(int *v1,int *v2,int *v3,int n)                  // func. variant
     { #pragma omp for
       for (int i= 0; i< n; i++)  v3[i] = v1[i] * v2[i];}

#pragma omp begin declare target
void t_vxv(int *v1,int *v2,int *v3,int n)                  // func. variant
     { #pragma omp distribute parallel for simd
       for (int i= 0; i< n; i++)  v3[i] = v1[i] * v2[i]; }
#pragma omp end declare target
```

variant F90 example

F90

```
subroutine vxv(v1, v2, v3, n)                    !! base function
   integer :: v1(:),v2(:), v3:), i, n
   !$omp  declare variant( p_vxv ) match( construct={parallel} )
   !$omp  declare variant( t_vxv ) match( construct={target}   )
   do i = 1,n; v3(i) = v1(i) * v2(i); enddo
end subroutine

subroutine p_vxv(v1, v2, v3, n)                   !! function variant
   integer :: v1(:),v2(:), v3:), i, n
   !$omp do
   do i = 1,n; v3(i) = v1(i) * v2(i); enddo
end subroutine

subroutine t_vxv(v1, v2, v3, n)                   !! function variant
   integer :: v1(:),v2(:), v3:), i, n
   !$omp declare target
   !$omp distribute parallel for simd
   do i = 1,n; v3(i) = v1(i) * v2(i) ; enddo
end subroutine
```

variant F90 example

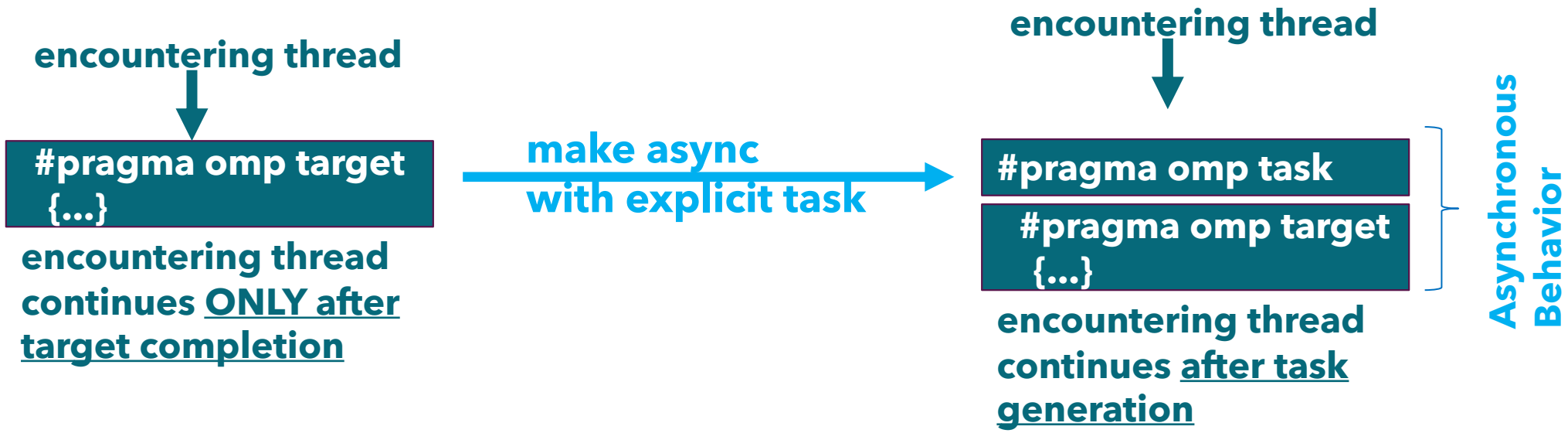
F90

```
program main
...
  !$omp parallel
     call vxv(v1,v2,v3, n)                          // will use variant p_vxv
  !$omp end parallel


  !$omp target teams map(to: v1,v2) map(from: v3)
     call vxv(v1,v2,v3, n)                          // will use variant t_vxv
  !$omp end target teams


     call vxv(v1,v2,v3, n)                          // will use base vxv
...
end program main
```

Asynchronous offloading

Asynchronous target execution -- early years

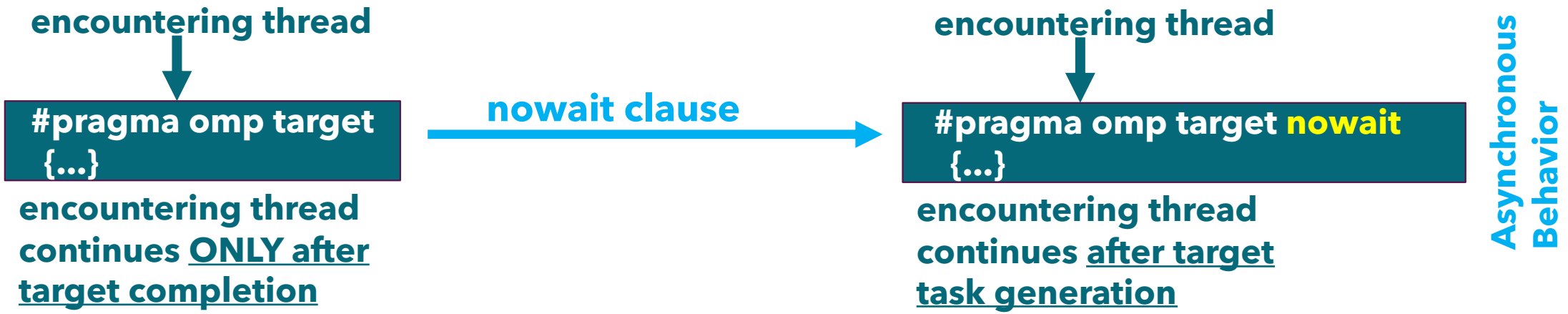
Version (4.0) of OpenMP offloading provided no way, other than using explicit tasks,* for offloading to be asynchronous.



* Can set up dependences with **depend** clause.

Asynchronous target execution -- early years

After (4.0) OpenMP offloading specifies that a *target task* is generated and performs the offload – the initial task/thread waits ("blocks") unless the **nowait** clause is present (for async behavior).



depend clause available on target construct

Asynchronous target: wait and depend clauses

```
#pragma omp parallel num_threads(three_or_greater)
{
  #pragma omp masked //master
  {
    #pragma omp target nowait depend(out:v1) map(from:v1)
    for(int i=0;i<n;i++) v1[i]=1;

    #pragma omp target nowait depend(out:v2) map(from:v2)
    for(int i=0;i<n;i++) v2[i]=1;

    #pragma omp target nowait depend(in:v1,v2) depend(out:p) \
                                    map(to:v1,v2)   map(from:p)
    for(int i=0;i<n;i++) p[i]=v1[i]*v2[i];
  }
  host_independent_parallel_work(); //inside fun. schedule(dynamic,#)

  #pragma omp taskwait
  #pragma omp masked
  for(int i=0; i<n;   i++) assert(y[i]==3); //Validate
}
```

Asynchronous target: wait and depend clauses

```
!$omp parallel num_threads(3)
 !$omp master !or masked
    !$omp target nowait depend(out:v1) map(from:v1)
    do i =1,N; v1(i)=1; enddo
    !$omp end target

    !$omp target nowait depend(out:v2) map(from:v1)
    do i =1,N; v2(i)=2; enddo
    !$omp end target

    !$omp target nowait depend(in:v1,v2) depend(out:p) &
    !$omp&                    map(to:v1,v2)   map(from:p)
    do i=1,N; p(i) = v1(i) * v2(i); enddo
    !$omp end target
!$omp end master
 call host_independent_parallel_work()
!$omp taskwait
!$omp master !or masked
do i=1,N; if( p(i) /= 2 ) stop ("p != 2"); enddo
!$omp end master
!$omp end parallel
```

multiple devices

```
ndevs    = omp_get_num_devices();
blk_size = N/ndevs;

#pragma omp parallel for num_threads(ndevs) private(start,end)
for (int idev=0; idev<ndevs; idev++)
{
    start=blk_size*idev;  end=start+blk_size;

    #pragma omp target device(idev) map(tofrom: x[start:blk_size]) \
                                    map(to:     y[start:blk_size])
    for( int i=start; i<end; i++){ x[i] = x[i] + a*y[i]; }
}
```