



Advanced OpenMP

Presenter : Giacomo Capodaglio
AMD @ CASTIEL HPC
Oct 28-30, 2025

AMD 
together we advance_

Review of the compute directive

Directive/Clause	Meaning
target	offloads the enclosed code to the device (GPU)
teams	creates a “league of teams” with the initial thread of each executing the code region with all of the data on each thread
distribute	loop iterations are distributed out across the teams and executed on the main thread
parallel	Create multiple threads (of a team)
for/do	spread out different portions of work over threads
simd	vectorization (use SIMD instructions), but not used by most compilers, including amdclang
loop	effectively replaces "distribute parallel for simd" In OpenMP 6.0, also "teams distribute parallel for simd"

SIMD – Single Instruction Multiple Data is a term from Flynn’s Taxonomy that categorizes types of computer architectures. In this architecture, a single instruction is applied to multiple data. One of the examples of this type is a vector unit where one instruction is applied to multiple lanes. The GPU is very much like a wide vector unit and is also a SIMD architecture. It is often described as SIMT, Single Instruction Multiple Thread, a subcategory of SIMD with some important distinctions.

Breaking down the compute directive – C/C++ and Fortran

Directive/Clause	C/C++	Fortran
target	<code>#pragma omp target {structured code block}</code>	<code>!\$omp target <code block> !\$omp end target</code>
teams	<code>#pragma omp target teams {structured code block}</code>	<code>!\$omp target teams <code block> !\$omp end target teams</code>
distribute	<code>#pragma omp target teams distribute {structured code block}</code>	<code>!\$omp target teams distribute <code block> !\$omp end target teams distribute</code>
parallel for/do simd	<code>#pragma omp target teams distribute parallel for simd {structured code block}</code>	<code>!\$omp target teams distribute parallel do simd <code block> !\$omp end target teams distribute parallel do simd</code>
loop	<code>#pragma omp target teams loop {structured code block}</code>	<code>!\$omp target teams loop <code block> !\$omp end target teams loop</code>
loop (OpenMP 6.0 standard alternate)	<code>#pragma omp target loop* {structured code block}</code> * both forms valid	<code>!\$omp target loop <code block> !\$omp end target loop</code>

Exploring subsets of the full compute pragma

Replacing the pragma with subsets of the full pragma
See each of the following slides

```
void saxpy(float a, float *x, float *y, int N) {  
#pragma omp target teams distribute parallel for simd  
    for (int i = 0; i < N; i++) {  
        y[i] += a * x[i];  
    }  
  
    printf("check output:\n");  
    printf("y[0] %1f\n",y[0]);  
    printf("y[N-1] %1f\n",y[N-1]);  
}
```

```
amdclang -fopenmp --offload-arch=$GPU_ARCH ...
```

Requires "export HSA_XNACK=1"
or map clause

Querying what implementation does with each compute directive

- Add export LIBOMPTARGET_KERNEL_TRACE=1

```
#pragma omp target teams distribute parallel for simd
```

```
DEVID: 0 SGN:5 ConstWGSize:256 args: 5 teamsXthrds:( 416X 256) reqd:( 0X 0)  
lds_usage:0B sgpr_count:24 vgpr_count:8 sgpr_spill_count:0 vgpr_spill_count:0  
tripcount:10000000 rpc:0 n:__omp_offloading_34_8975356_saxpy_18  
Time of kernel: 0.082906
```

- The next slide shows four different directives and the results in the same quad chart layout as the earlier slide

Comparing subsets of GPU parallel compute directive

```
#pragma omp target
```

```
DEVID:  0 SGN:3 ConstWGSize:257  args: 5
teamsXthrds:(  1X 256) reqd:(  0X  0)
lds_usage:16B sgpr_count:16 vgpr_count:3
sgpr_spill_count:0 vgpr_spill_count:0
tripcount:0 rpc:0
n:__omp_offloading_34_5c4ed40a_saxpy_18
Time of kernel: 5.407085
```

```
#pragma omp target teams
```

```
DEVID:  0 SGN:3 ConstWGSize:257  args: 5
teamsXthrds:( 624X 256) reqd:(  0X  0)
lds_usage:16B sgpr_count:12 vgpr_count:3
sgpr_spill_count:0 vgpr_spill_count:0
tripcount:0 rpc:0
n:__omp_offloading_34_5c4ed40b_saxpy_18
Time of kernel: 11.166301
```

```
#pragma omp target teams distribute
```

```
DEVID:  0 SGN:3 ConstWGSize:257  args: 5
teamsXthrds:( 624X 256) reqd:(  0X  0)
lds_usage:16B sgpr_count:24 vgpr_count:3
sgpr_spill_count:0 vgpr_spill_count:0
tripcount:10000000 rpc:0
n:__omp_offloading_34_5c4ed40c_saxpy_18
Time of kernel: 0.149113
```

```
#pragma omp target parallel for
```

```
DEVID:  0 SGN:2 ConstWGSize:256  args: 5
teamsXthrds:(  1X 256) reqd:(  0X  0)
lds_usage:32B sgpr_count:25
vgpr_count:17 sgpr_spill_count:0
vgpr_spill_count:0 tripcount:0 rpc:0
n:__omp_offloading_34_5c4ed416_saxpy_18
Time of kernel: 0.126748
```

For reference; #pragma omp target teams distribute parallel for from previous slide Time of kernel: 0.082906

Breaking up the OpenMP[®] Compute Constructs

- The single-line directives can be split apart into separate directives. We've been using the single line compute construct something like the following
 - `#pragma omp target teams distribute parallel for simd`
- But we are not limited to just a single line. We can break up the compute directive into multiple lines. The simplest multi-line directives are equivalent to the single line form.
- Compute on the GPU with all teams (workgroups) and data partitioned, but only the main thread
 - `#pragma omp target teams distribute`
- Parallelize the following for loop (use all the threads in a workgroup)
 - `#pragma omp parallel for simd`

Proper nomenclature is that alone or first on a line, it is a directive. When it follows a directive, it is a modifier or a clause.

Multi-level Parallel saxpy

- We can split the directives across an outer loop and an inner loop to have more control. Usually, it is best to let the compiler do this as it generally does a better job. But there are special cases where the application developer may have information about something like typical sizes of loops. Note that this is somewhat different than the previous saxpy example in that it is built around a 2D data structure.

```
void saxpy(float a, float **x, float **y, int M, int N) {
    double tb, te;

    tb = omp_get_wtime();
    #pragma omp target teams distribute
    for (int j = 0; j < N; j++) {
        #pragma omp parallel for simd
        for (int i = 0; i < M; i++) {
            y[j][i] += a * x[j][i];
        }
    }
    te = omp_get_wtime();

    printf("Time of kernel: %lf\n", te - tb);

    printf("check output:\n");
    printf("y[0][0] %lf\n",y[0][0]);
    printf("y[N-1][M-1] %lf\n",y[N-1][M-1]);
}
```

Notes

- While this example shows split level pragmas that might be useful in special cases:
 - We do not recommend doing this in simple cases – let the compiler decide how to do the parallelism
 - Add a collapse clause instead – it increases the parallel work
 - Generally, it is not a good idea to use split level to force performance optimization, but only to address special cases
- Special cases
 - Small sizes of the outer or inner loop (maybe even unroll a loop?)
 - Something special being done on the inner loop where the synchronization benefits the required work

Other compute clauses – tile and more

- **tile** – block the loops into small tiles rather than a standard loop traversal of all x and then y.
- **num_teams(x)** – launch the kernel with the specified number of thread blocks
- **num_threads(x)** – defines the number of threads for the parallel construct
- **thread_limit(x)** – defines the maximum number of threads per team: can cause the compiler to generate code with a maximum number of threads, reducing register pressure (some compilers are still adding this optimization)
- **nowait** – do not wait at end of compute kernel. Default is to wait. This is one of the optimization options, but it can lead to race conditions and incorrect results
- **reduction(op: x)** – special case where multiple iterations write to common location(2). This might be a sum, min, max or similar type of operation

The `thread_limit` clause: for better compiler optimization

- The most commonly used of these compute clauses is `thread_limit`
- The `thread_limit` clause specifies the maximum workgroup size for the compiler generated GPU code
 - It frees up some additional memory resources in the kernel code such as registers that can make the code more efficient
 - It needs to follow a “teams” construct

```
!$omp target teams distribute thread_limit(256)
do j=1,n
  !$omp parallel do
    do i=1,m
      y(i,j) = y(i,j) + a * x(i,j)
    end do
  end do
end do
```

- `num_threads` specifies the threads for the code generated for this specific case (but not all situations)
 - It needs to follow a “parallel” construct

```
!$omp target teams distribute
do j=1,n
  !$omp parallel do num_threads(256)
    do i=1,m
      y(i,j) = y(i,j) + a * x(i,j)
    end do
  end do
end do
```

Understanding hardware options

- rocminfo
 - 110 CUs
 - Wavefront of size 64
 - 4 SIMDs per CU

```

Node: 11
Device Type: GPU
Cache Info:
  L1: 16(0x10) KB
  L2: 8192(0x2000) KB
Chip ID: 29704(0x7408)
Cacheline Size: 64(0x40)
Max Clock Freq. (MHz): 1700
BDFID: 56832
Internal Node ID: 11
Compute Unit: 110
SIMDs per CU: 4
Shader Engines: 8
Shader Arrs. per Eng.: 1
WatchPts on Addr. Ranges:4
Features: KERNEL_DISPATCH
Fast F16 Operation: TRUE
Wavefront Size: 64(0x40)
Workgroup Max Size: 1024(0x400)
Workgroup Max Size per Dimension:
  x 1024(0x400)
  y 1024(0x400)
  z 1024(0x400)
Max Waves Per CU: 32(0x20)
Max Work-item Per CU: 2048(0x800)

```



`#pragma omp target teams distribute parallel for simd`

Options for `#pragma omp target teams`:

- `num_teams(220)`: it is good practice to set the number of workgroups as a multiple of the CUs (which is 110 in this case)
- `thread_limit(256)`: the number of threads per workgroup should be a multiple of 64

The total number of threads is:

`num_teams*thread_limit` which should evenly divide the trip count of a loop

OpenMP® heavily relies on the concept of *region*

- What are regions?
 - A part of the code where a pragma applies
 - Default is the normal “block” of code following the directive
 - Can be specified by curly brackets { } in C/C++ or an end directive in Fortran
 - What kinds of regions are there?
 - Data regions – data may also exist on the GPU in this code region
 - Target regions – code in region is executed on the GPU
 - Parallel regions – code in region is executed in parallel
 - Original OpenMP specification only had **structured data regions**
 - How to handle Object-oriented code and other patterns?
- Later version of the standard added **unstructured data region** concept

Structured vs Unstructured Data regions example

Structured data region

```
#pragma omp target data map(tofrom: x[0:n])  
{  
#pragma omp target teams distribute parallel for simd  
  for (int i = 0; i < n; n++){  
    x[i] = 0.0;  
  }  
}
```

Unstructured data region

```
class myclass (int n) {  
  myclass(){  
    x=new double[n];  
    #pragma omp target enter data map(alloc: x[0:n])  
  }  
  
  ~myclass(){  
    #pragma omp target exit data map(delete: x[0:n])  
    delete [] x;  
  }  
}
```

While object exists

Map clause example

```

void saxpy(float a, float* x, float* y,
          int n) {
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
    #pragma omp target map(to:x[0:n]) \
                      map(tofrom:y[0:n])
    for (int i = 0; i < n; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf\n", t);
}

```

The compiler cannot determine the size of memory behind the pointer.

host

a
x[0:n]
y[0:n]

target

y[0:n]

host

Programmers have to help the compiler with the amount of data to transfer.

```
amdclang -fopenmp --offload-arch=$GPU_ARCH ...
```

Effect of **map** types and **target update** directive clauses

OpenMP syntax	Allocates/deletes device memory	Modifies reference counter	Copies data
Map to/from ¹	Yes	Yes	Yes
Map always ²	Yes	Yes	Yes
Map alloc/delete ³	Yes ⁴	Yes	No
Map release	If reference counter 0, delete	Decrements	No
Update to/from	No	No	Yes

Notes:

1. "Map to" checks if the memory is already allocated for the device.
 - a. If not allocated, the device memory is allocated and the reference counter is set to one, and the data is copied to the device
 - b. If allocated, the size is checked, and the reference counter is incremented

Similar for "map from"

2. Same as to/from, but always copies the memory over even if it already exists
3. "Map alloc" checks if the memory is already allocated for the device.
 - a. if not allocated, the device memory is allocated, and the reference counter is set to one
 - b. if allocated, the size is checked, and the reference counter is incremented.

"Map delete" will delete the memory and set the reference counter to zero

4. More generally, to cover single memory spaces, the data must be available in the memory space.

OpenMP[®] Device Constructs: target data

- Create scoped data environment and transfer data from the host to the device and back

- Syntax (C/C++)

```
#pragma omp target data [clause[[,] clause],...]  
structured-block
```

- Syntax (Fortran)

```
!$omp target data [clause[[,] clause],...]  
structured-block  
!$omp end target data
```

- Clauses

```
device(scalar-integer-expression)  
map([{alloc | to | from | tofrom | release | delete}:] list)  
if(scalar-expr)
```

Optimize Data Transfers

- Reduce the amount of time spent transferring data
 - Use map clauses to enforce direction of data transfer.
 - Use target data to keep data on the device.
 - Note, anything that is mapped tofrom will be copied back to the host and data modified by the host inside the target data region **will be overwritten**
 - There is an implicit delete at the end of the target data

No map clauses! Presence checks will find data via the pointer.

```
void example() {
    float tmp[N], a[N], b[N], c[N];
    #pragma omp target data map(alloc:tmp[:N]) \
        map(to:a[:N],b[:N]) \
        map(tofrom:c[:N])
    {
        zeros(tmp, N);
        compute_kernel_1(tmp, a, N); // uses target
        saxpy(2.0f, tmp, b, N);
        compute_kernel_2(tmp, b, N); // uses target
        saxpy(2.0f, c, tmp, N);
    }
}
```

Create data environment.

```
void zeros(float* a, int n) {
    #pragma omp target teams distribute parallel for
    for (int i = 0; i < n; i++)
        a[i] = 0.0f;
}

void saxpy(float a, float* y, float* x, int n) {
    #pragma omp target teams distribute parallel for
    for (int i = 0; i < n; i++)
        y[i] = a * x[i] + y[i];
}
```

Unstructured Data Regions

- **Unstructured data regions** with the target enter/exit data were added to the standard to handle cases such as C++ classes
- Often, the target enter data will be right after allocation. And the target exit data will be right before the free.
- from and tofrom not allowed in : `#pragma omp target enter data map`
- To enforce copy back from the device to host: `#pragma omp target exit data map(from:`

```
float *tmp, *a, *b, *c;
int main(int argc, char *argv[]) {
    int N = 100;
    tmp = (float *)malloc(N*sizeof(float));
    a = (float *)malloc(N*sizeof(float));
    b = (float *)malloc(N*sizeof(float));
    c = (float *)malloc(N*sizeof(float));
    #pragma omp target enter data \
        map(alloc:tmp[:N], a[:N], b[:N], c[:N])
        zeros(tmp, N);
        compute_kernel_1(tmp, a, N);
        saxpy(2.0f, tmp, b, N);

        compute_kernel_2(tmp, b, N);
        saxpy(2.0f, c, tmp, N);

    #pragma omp target exit data map(delete:tmp, a, b, c)
    free(tmp, a, b, c);
}
```

```
void zeros(float* a, int n) {
    #pragma omp target teams distribute parallel for
    for (int i = 0; i < n; i++)
        a[i] = 0.0f;
}
```

```
void saxpy(float a, float* y, float* x, int n) {
    #pragma omp target teams distribute parallel for
    for (int i = 0; i < n; i++)
        y[i] = a * x[i] + y[i];
}
```

OpenMP[®] Device Constructs: target **update**

- Issue data transfers to or from existing data device environment

- Syntax (C/C++)

```
#pragma omp target update [clause[[,] clause],...]
```

- Syntax (Fortran)

```
!$omp target update [clause[[,] clause],...]
```

- Clauses

```
device(scalar-integer-expression)
```

```
to(list)
```

```
from(list)
```

```
if(scalar-expr)
```

Example: target **data** and target **update**

```

#pragma omp target data map(alloc:tmp[:N]) map(to:input[:N]) map(from:res)
{
#pragma omp target
#pragma omp teams distribute parallel for simd
    for (int i=0; i<N; i++)
        tmp[i] = some_computation(input[i], i);

    update_input_array_on_the_host(input);

#pragma omp target update to(input[:N])

#pragma omp target
#pragma omp teams parallel for simd reduction(+:res)
    for (int i=0; i<N; i++)
        res += final_computation(input[i], tmp[i], i);
}

```



Review of Data Movement/ Management pragmas and clauses

`map(to:var[0:n]), map(from:var[0:n]), map(tofrom:var[0:n])` can be added to any compute directive

- Structured data regions – a block of code in a functional unit that will have data mapped over at the start and mapped back at the end

```
#pragma omp target data map(to:x[0:n]) map(from:y[0:n])
```

- Unstructured data regions are more flexible and can be done after array creation on the host and just before freeing the memory

```
#pragma omp target enter data map(alloc:var[0:n])
```

```
#pragma omp target exit data map(release:var[0:n])
```

- Updating data from host to device or vice-versa

```
#pragma omp target update to(var) -- host to device
```

```
#pragma omp target update from(var) -- device to host
```

Complex cases

- Usually, the difficulty in porting a code to OpenMP® is due to complex combinations of code
- Watch out for structs and classes or any data type that doesn't map to simple arrays
- Allocatables, pointers or implicit allocation/reallocation are tricky to get right, both for the programmer and the compiler developer
- The declare pragma can be tricky for both subroutines and data. Some variation in implementations also exist across compilers
- Deep copies (structs or classes that contain pointers and data that won't be copied over to the device)
- Anything that has been added more recently to the OpenMP® standard may have portability issues

Dynamic arrays dependent on program input

- Original code has a statically sized array.

```
#pragma omp target data
double atmos_temp[nsize];
#pragma omp end target data
```

- We want to change it to a dynamic size based on input

```
#pragma omp target data
double *atmos_temp;
#pragma omp end target data
```

```
atmos_temp = (double *) malloc(nsize_input * sizeof(double));
// use in device computation loop
```

- The malloc is going to assign a new pointer location to atmos_temp. Instead use **enter data** after allocation and preferably set values in a device loop.

```
#pragma omp target enter data map(alloc:atmos_temp[0:nsize_input])
```

Harder to catch in Fortran and C++

Dynamic array as part of a struct

```
#pragma omp target data
```

```
struct {  
    int n;  
    double *x;  
}
```

```
#pragma omp end target data
```

```
x = (double *)malloc(nsize*sizeof(double));
```

```
#pragma omp target enter data map(alloc:x[0:nsize]))
```

```
/// lots of code
```

```
#pragma omp target exit data map(release:x) – don't use delete? (open question)
```

- There is some variation in current compilers on how they handle release vs delete
- OpenMP[®] release has same behavior as OpenACC delete

Some examples with runtime debugging output

- It can be difficult to understand what the compiler does with the memory management directives
- A simple way to expose what is happening is to use the LIBOMPTARGET_INFO environment variable
- To get all the information:
 - `export LIBOMPTARGET_INFO=-1`
- There are subsets of the information that can be requested (to reduce the verbosity)
 - Print data arguments at entry to an OpenMP device kernel: 0x01
 - Report if a mapped address already exists in the device mapping table: 0x02
 - Dump the device pointer map at kernel exit: 0x04
 - Report changes in the device mapping table: 0x08
 - Print OpenMP kernel information from device plugins: 0x10
 - Report when data is copied to and from the device: 0x20
- Set one or two of the flags
 - `export LIBOMPTARGET_INFO=$((0x01 | 0x10))`

Set LIBOMPTARGET_INFO flag at runtime

- By setting this LIBOMPTARGET_INFO flag at runtime, you can see the detailed information **at a particular point in your code**. LLVM™ has the `__tgt_set_info_flag()` routine for this purpose.

```
extern "C" void __tgt_set_info_flag(uint32_t);
```

```
<...>
```

```
__tgt_set_info_flag(-1);  
#pragma omp target teams distribute parallel for simd map(to: x[0:n], y[0:n]) map(from: z[0:n])  
for (int i = 0; i < n; i++){  
    z[i] = a*x[i] + y[i];  
}  
__tgt_set_info_flag(0);
```

Basic OpenMP[®] daxpy code: we'll show variants of this

```

62 double sum_time = 0.0;
63 double max_time = -1.0e10;
64 double min_time = 1.0e10;
65 for (int iter=0; iter<num_iteration; iter++) {
66     sum_time += timers[iter];
67     max_time = max(max_time,timers[iter]);
68     min_time = min(min_time,timers[iter]);
69 }
70
71 double avg_time = sum_time / (double)num_iteration;
72
73 cout << "-Timing in Seconds: min=" << fixed << setprecision(6) << min_time << ", max=" << max_time << ", avg=" << avg_time << endl;
74
75 main_timer = omp_get_wtime()-main_start;
76 cout << "-Overall time is " << main_timer << endl;
77
78 cout << "Last Value: z[" << n-1 << "]=" << z[n-1] << endl;
79
80 delete [] x;
81 delete [] y;
82 delete [] z;
83
84 return 0;
85 }
86
87 void daxpy(int n, double a, double *__restrict__ x, double *__restrict__ y, double *__restrict__ z)
88 {
89     #pragma omp target teams distribute parallel for simd map(to: x[0:n], y[0:n]) map(from: z[0:n])
90     for (int i = 0; i < n; i++)
91         z[i] = a*x[i] + y[i];
92 }

```

To get code:

```
git clone https://github.com/amd/HPCTrainingExamples.git
```

then

```
cd HPCTrainingExamples/Pragma_Examples/OpenMP/CXX/memory_pragmas
```

multiple versions are available, we will consider only a few of them here

Mem1.cc version

Map clause on pragma line just before computational loop

```
#pragma omp target teams distribute parallel for simd map(to: x[0:n], y[0:n]) map(from: z[0:n])
```

Running this with LIBOMPTARGET_INFO=-1, we can see the memory operations. All occur from the pragma at line 89.

```
omptarget device 0 info: Entering OpenMP kernel at mem1.cc:89:1 with 5 arguments:
omptarget device 0 info: firstprivate(n)[4] (implicit)
omptarget device 0 info: from(z[0:n])[800000]
omptarget device 0 info: firstprivate(a)[8] (implicit)
omptarget device 0 info: to(x[0:n])[800000]
omptarget device 0 info: to(y[0:n])[800000]
omptarget device 0 info: Creating new map entry with HstPtrBase=0x000000001eac540, HstPtrBegin=0x000000001eac540, TgtAllocBegin=0x00007f0030e20000, TgtPtrBegin=0x00007f0030e20000, Size=800000, DynRefCount=1, HoldRefCount=0, Name=z[0:n]
omptarget device 0 info: Creating new map entry with HstPtrBase=0x000000001d25b20, HstPtrBegin=0x000000001d25b20, TgtAllocBegin=0x00007f0030ee4000, TgtPtrBegin=0x00007f0030ee4000, Size=800000, DynRefCount=1, HoldRefCount=0, Name=x[0:n]
omptarget device 0 info: Copying data from host to device, HstPtr=0x000000001d25b20, TgtPtr=0x00007f0030ee4000, Size=800000, Name=x[0:n]
omptarget device 0 info: Creating new map entry with HstPtrBase=0x000000001de9030, HstPtrBegin=0x000000001de9030, TgtAllocBegin=0x00007efffc200000, TgtPtrBegin=0x00007efffc200000, Size=800000, DynRefCount=1, HoldRefCount=0, Name=y[0:n]
omptarget device 0 info: Copying data from host to device, HstPtr=0x000000001de9030, TgtPtr=0x00007efffc200000, Size=800000, Name=y[0:n]
omptarget device 0 info: Mapping exists with HstPtrBegin=0x000000001eac540, TgtPtrBegin=0x00007f0030e20000, Size=800000, DynRefCount=1 (update suppressed), HoldRefCount=0
omptarget device 0 info: Mapping exists with HstPtrBegin=0x000000001d25b20, TgtPtrBegin=0x00007f0030ee4000, Size=800000, DynRefCount=1 (update suppressed), HoldRefCount=0
omptarget device 0 info: Mapping exists with HstPtrBegin=0x000000001de9030, TgtPtrBegin=0x00007efffc200000, Size=800000, DynRefCount=1 (update suppressed), HoldRefCount=0
"PluginInterface" device 0 info: Launching kernel __omp_offloading_42_82257ae0_Z5daxpyidPdS_S_l89 with 391 blocks and 256 threads in SPMD-Big-Jump-Loop mode
DEVID: 0 SGN:5 ConstWGSz:256 args: 6 teamsXthrds:( 391X 256) reqd:( 0X 0) lds_usage:0B sgpr_count:24 vgpr_count:12 sgpr_spill_count:0 vgpr_spill_count:0 tripcount:100000 rpc:0 n:__omp_offloading_42_82257ae0_Z5daxpyidPdS_S_l89
AMDGPU device 0 info: #Args: 6 Teams x Thrds: 391x 256 (MaxFlatWorkGroupSize: 256) LDS Usage: 0B #SGPRs/VGPRs: 24/12 #SGPR/VGPR Spills: 0/0 Tripcount: 100000
omptarget device 0 info: Mapping exists with HstPtrBegin=0x000000001de9030, TgtPtrBegin=0x00007efffc200000, Size=800000, DynRefCount=0 (decremented, delayed deletion), HoldRefCount=0
omptarget device 0 info: Mapping exists with HstPtrBegin=0x000000001d25b20, TgtPtrBegin=0x00007f0030ee4000, Size=800000, DynRefCount=0 (decremented, delayed deletion), HoldRefCount=0
omptarget device 0 info: Mapping exists with HstPtrBegin=0x000000001eac540, TgtPtrBegin=0x00007f0030e20000, Size=800000, DynRefCount=0 (decremented, delayed deletion), HoldRefCount=0
omptarget device 0 info: Copying data from device to host, TgtPtr=0x000000001eac540, HstPtr=0x00007f0030e20000, Size=800000, Name=z[0:n]
omptarget device 0 info: Removing map entry with HstPtrBegin=0x000000001de9030, TgtPtrBegin=0x00007efffc200000, Size=800000, Name=y[0:n]
omptarget device 0 info: Removing map entry with HstPtrBegin=0x000000001d25b20, TgtPtrBegin=0x00007f0030ee4000, Size=800000, Name=x[0:n]
omptarget device 0 info: Removing map entry with HstPtrBegin=0x000000001eac540, TgtPtrBegin=0x00007f0030e20000, Size=800000, Name=z[0:n]
```

Note implicit firstprivate for scalar arguments

Device memory allocated and reference count incremented

A new map entry is created for x, y and z

Data copied

Device array deleted

Mem2.cc version -- Add enter/exit data alloc/delete when memory is created/freed

After new:

```
#pragma omp target enter data map(alloc: x[0:n], y[0:n], z[0:n])
```

Keep map on computational loop. The maps see that the memory is allocated, but still do the copies to and from thanks to the presence of the **always** clause. This will increment the Reference Counter and decrement it at end of loop.

```
#pragma omp target teams distribute parallel for simd map(always to: x[0:n], y[0:n]) map(always from: z[0:n])
```

Before delete:

```
#pragma omp target exit data map(delete: x[0:n], y[0:n], z[0:n])
```

After enter data map:

```
omptarget device 0 info: Creating new map entry with HstPtrBase=0x000000000a27b30, HstPtrBegin=0x000000000a27b30, TgtAllocBegin=0x00007f2540c20000, TgtPtrBegin=0x00007f2540c20000, Size=800000, DynRefCount=1, HoldRefCount=0, Name=x[0:n]
```

Computational Loop:

← The mapping already exists so no new map is created

```
omptarget device 0 info: Mapping exists with HstPtrBegin=0x000000000bae550, TgtPtrBegin=0x00007f2509200000, Size=800000, DynRefCount=2 (incremented), HoldRefCount=0, Name=z[0:n]
```

```
omptarget device 0 info: Mapping exists with HstPtrBegin=0x000000000a27b30, TgtPtrBegin=0x00007f2540c20000, Size=800000, DynRefCount=2 (incremented), HoldRefCount=0, Name=x[0:n]
```

```
omptarget device 0 info: Copying data from host to device, HstPtr=0x000000000a27b30, TgtPtr=0x00007f2540c20000, Size=800000, Name=x[0:n]
```

```
omptarget device 0 info: Mapping exists with HstPtrBegin=0x000000000aeb040, TgtPtrBegin=0x00007f2540ce4000, Size=800000, DynRefCount=2 (incremented), HoldRefCount=0, Name=y[0:n]
```

After exit data map:

```
omptarget device 0 info: Removing map entry with HstPtrBegin=0x000000000bae550, TgtPtrBegin=0x00007f2509200000, Size=800000, Name=z[0:n]
```

```
omptarget device 0 info: Removing map entry with HstPtrBegin=0x000000000aeb040, TgtPtrBegin=0x00007f2540ce4000, Size=800000, Name=y[0:n]
```

```
omptarget device 0 info: Removing map entry with HstPtrBegin=0x000000000a27b30, TgtPtrBegin=0x00007f2540c20000, Size=800000, Name=x[0:n]
```

Mem3.cc version – replace map on computation loop with updates

```
void daxpy(int n, double a, double *__restrict__ x, double *__restrict__ y, double *__restrict__ z)
{
#pragma omp target update to (x[0:n], y[0:n])
#pragma omp target teams distribute parallel for simd
    for (int i = 0; i < n; i++)
        z[i] = a*x[i] + y[i];
#pragma omp target update from (z[0:n])
}
```

This is the target enter data

```
omptarget device 0 info: Creating new map entry with HstPtrBase=0x000000000550b30, HstPtrBegin=0x000000000550b30, TgtAllocBegin=0x00007f0d98220000, TgtPtrBe
gin=0x00007f0d98220000, Size=800000, DynRefCount=1, HoldRefCount=0, Name=x[0:n]
omptarget device 0 info: Creating new map entry with HstPtrBase=0x000000000614040, HstPtrBegin=0x000000000614040, TgtAllocBegin=0x00007f0d982e4000, TgtPtrBe
gin=0x00007f0d982e4000, Size=800000, DynRefCount=1, HoldRefCount=0, Name=y[0:n]
omptarget device 0 info: Creating new map entry with HstPtrBase=0x0000000006d7550, HstPtrBegin=0x0000000006d7550, TgtAllocBegin=0x00007f0c90200000, TgtPtrBe
gin=0x00007f0c90200000, Size=800000, DynRefCount=1, HoldRefCount=0, Name=z[0:n]
omptarget device 0 info: OpenMP Host-Device pointer mappings after block at mem3.cc:52:1:
omptarget device 0 info: Host Ptr      Target Ptr      Size (B)  DynRefCount  HoldRefCount  Declaration
omptarget device 0 info: 0x000000000550b30 0x00007f0d98220000 800000    1             0             x[0:n] at mem3.cc:43:12
omptarget device 0 info: 0x000000000614040 0x00007f0d982e4000 800000    1             0             y[0:n] at mem3.cc:44:12
omptarget device 0 info: 0x0000000006d7550 0x00007f0c90200000 800000    1             0             z[0:n] at mem3.cc:45:12
omptarget device 0 info: Updating data within the OpenMP data region with update_mapper at mem3.cc:92:1 with 2 arguments:
omptarget device 0 info: to(x[0:n])[800000]
omptarget device 0 info: to(y[0:n])[800000]
omptarget device 0 info: Mapping exists with HstPtrBegin=0x000000000550b30, TgtPtrBegin=0x00007f0d98220000, Size=800000, DynRefCount=1 (update suppressed), H
oldRefCount=0
omptarget device 0 info: Copying data from host to device, HstPtr=0x000000000550b30, TgtPtr=0x00007f0d98220000, Size=800000, Name=x[0:n]
omptarget device 0 info: Mapping exists with HstPtrBegin=0x000000000614040, TgtPtrBegin=0x00007f0d982e4000, Size=800000, DynRefCount=1 (update suppressed), H
oldRefCount=0
omptarget device 0 info: Copying data from host to device, HstPtr=0x000000000614040, TgtPtr=0x00007f0d982e4000, Size=800000, Name=y[0:n]
omptarget device 0 info: OpenMP Host-Device pointer mappings after block at mem3.cc:92:1:
omptarget device 0 info: Host Ptr      Target Ptr      Size (B)  DynRefCount  HoldRefCount  Declaration
omptarget device 0 info: 0x000000000550b30 0x00007f0d98220000 800000    1             0             x[0:n] at mem3.cc:43:12
omptarget device 0 info: 0x000000000614040 0x00007f0d982e4000 800000    1             0             y[0:n] at mem3.cc:44:12
omptarget device 0 info: 0x0000000006d7550 0x00007f0c90200000 800000    1             0             z[0:n] at mem3.cc:45:12
```

This is the target update

Note that the reference counter has not changed

Mem4.cc version – modify Mem2 to replace delete with release

LIBOMPTARGET_INFO Report is the same as for mem2.cc

Reference counter is decremented to zero and device array is deleted

When should I use delete and when should I use release?

In many situations, either could be used. If the pragmas are at the allocation and deletion, the delete clause would seem more sensible. The one place to be careful is when the memory is part of a structure. Deleting the memory can cause the map for the structure to be removed.

Reductions to scalar or arrays

OpenMP® Offloading Example: Reduction

```

#include <stdio.h>
#include <stdlib.h>
#define N 5000000
int main(){
    double *a, *b;
    a = (double*)malloc(sizeof(double) * N);
    b = (double*)malloc(sizeof(double) * N);
    for(int i = 0; i < N; i++){
        a[i] = 1.0;
        b[i] = 1.0;
    }

    double sum = 0.0;
    #pragma omp target data map(to:a[0:N], b[0:N])
    ///#pragma omp target teams distribute parallel for private(i) map(tofrom:sum) reduction(+:sum)
    #pragma omp target teams distribute parallel for reduction(+:sum)
    for(int i = 0; i < N; i++)
        sum += a[i] * b[i];

    printf("SUM = %f\n", sum);
    free(a);
    free(b);
    return 0;
}

```

Sum is automatically set to zero for sum reductions. It does not need to be set for OpenMP, but it is needed for serial code compiled without OpenMP.

Note: Scalars are implicitly firstprivate in target constructs (as of OpenMP 4.5)

Data directive to move data to device (GPU)

Compute loop on GPU, copy sum to and from GPU and do a sum reduction on sum variable

`map` clause should not be included if it is a reduction variable. Each reduction variable is initialized based on the type of reduction and then the host version of the reduction variable is updated with the final result. `private(i)` is also not needed – index variables are automatically private.

Real-world cases

It is pretty common in physics applications that there is a long computational loop and at the end there is a reduction into an **array** variable

- * Weather/Climate codes where energy contributions are summed into ocean or atmospheric levels
- * Reaction energies are summed into a particle array

OpenMP® Offloading Example: Reduction to Array

```

#include <stdio.h>
#include <stdlib.h>
#define N 5000000
int main(){
    double **a, **b;
    // allocate 2D arrays a and b

    double sum[n];
    #pragma omp target data map(to:a[0:M][0:N], b[0:M][0:N])
    #pragma omp target teams distribute parallel for reduction(+:sum[0:n])
    for(int j = 0; j < N; j++){
        for(int i = 0; i < N; i++){
            sum[j] += a[j][i] * b[j][i];
        }
    }

    for(int j = 0; j < N; j++){
        printf("SUM = %f\n", sum);
    }

    // free 2D arrays

    return 0;
}

```

Data directive to move data to device (GPU)

Compute loop on GPU, copy sum to and from GPU and do a sum reduction on sum variable

It is common to call functions from inside target regions (loops, for instance)

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

If the subroutine is compiled for the architecture of the host, then it may not be able to run on the device architecture. In some compilers this would result in a linking error. To resolve this, we must indicate that a device compatible version of the subroutine should be generated

Calling a subroutine from a target region

To call a subroutine from a target region, it must have the pragma `declare target` added to the specification block and if not visible, to the prototype

C
/
C++

```
#pragma omp begin declare target
void *compute(){
    ...
};
#pragma omp end declare target
```

```
#pragma omp declare target
void *compute(){
    ...
};
```

```
#pragma omp declare target(compute)
...
void *compute(){
    ...
};
```

F
o
r
t
r
a
n

```
subroutine compute()
    use some_module
    implicit none
    !$omp declare target
    integer :: variable
    ...
end subroutine
```

```
!$omp declare target(compute)
...
subroutine compute()
    ...
end subroutine
```

Not all compilers recognize device routines or different forms of the directives. They can have difficulties with global variables that are used in device routines.

Data can also be declared to be on the target

C
/
C++

```
#pragma omp begin declare target
double constants[10] = { ... }
#pragma omp end declare target
```

```
#pragma omp declare target
double constants[10] = { ...
}
```

```
#pragma omp declare target(constants)
...
double constants[10] = { ... }
```

F
o
r
t
r
a
n

```
!$omp declare target(constants)
...
integer :: constants(10)
```

declare target available clauses

- **device_type**: specifies where the specified version of the procedure (or variable) should be made available: possible keywords are `host` | `nohost` | `any`
 - If `device_type(nohost)` is used on the `declare target` directive for global data, variable data must be in the “heap” and not a “stack” based variable (module variable, variable with `save` attribute)
- **enter**: data mapping attribute clause
- **indirect**: informs the compiler that the function can potentially be used via function pointers and to use device version of the same within the target regions; the functions must be specified in an `enter` clause, thus must be mappable (this feature limits the number of functions that can be used by function pointers in target region)
- **link**: supports compilation of device procedures that refer to variable with static storage duration that appear as list item in the clause – data is not mapped at the `declare target` directive, but is mapped according to OpenMP data mapping rules
- **local**: each list item has the device-local attribute, so it’s allocated on the memory of a specific device

Examples

- It is useful to look at a few examples in C, Fortran, and C++ to see how these basic concepts of declaring subroutines and data as being on the target actually work in practice
- The examples are in the following repository:
git clone <https://github.com/amd/HPCTrainingExamples>
- They are in the following directories in the repository
 - HPCTrainingExamples/Pragma_Examples/OpenMP/C/5_device_routines
 - HPCTrainingExamples/Pragma_Examples/OpenMP/Fortran/5_device_routines
 - HPCTrainingExamples/Pragma_Examples/OpenMP/CXX/5_device_routines

We start with C as the most straightforward examples. Then Fortran has some additional complexities with interfaces and modules. Finally, we look at C++ classes and the complexities with them.

C Language device subroutine examples – part 1

- The first example calls a subroutine from within a for loop. The subroutine is defined in another file.
- To port the loop to the GPU device, compute directives are added before the for loop.

```

10 int main(int argc, char *argv[]){
11     int N=1000;
12     double *x = (double *)malloc(N*sizeof(double));
13     for (int k = 0; k < N; k++){
14         x[k] = 0.0;
15     }
16
17     for (int k = 0; k < N; k++){
18         compute(&x[k]);
19     }
20
21     double sum = 0.0;
22     for (int k = 0; k < N; k++){
23         sum += x[k];
24     }
25
26     printf("Result: sum of x is %lf\n",sum);
27
28     free(x);
29 }

```

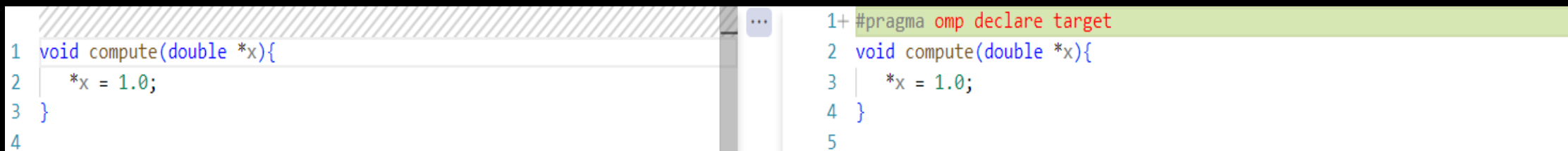
```

10 int main(int argc, char *argv[]){
11     int N=1000;
12     double *x = (double *)malloc(N*sizeof(double));
13+ #pragma omp target enter data map (alloc:x[0:N])
14+ #pragma omp target teams distribute parallel for
15     for (int k = 0; k < N; k++){
16         x[k] = 0.0;
17     }
18
19+ #pragma omp target teams distribute parallel for
20     for (int k = 0; k < N; k++){
21         compute(&x[k]);
22     }
23
24     double sum = 0.0;
25+ #pragma omp target teams distribute parallel for reduction(+:sum)
26     for (int k = 0; k < N; k++){
27         sum += x[k];
28     }
29
30     printf("Result: sum of x is %lf\n",sum);
31
32+ #pragma omp target exit data map (release:x[0:N])
33     free(x);
34 }

```

C Language device subroutine examples – part 2

- The compiler will now complain about a missing `compute` routine. We need to add the `declare target` directive around the subroutine code block



```

1 void compute(double *x){
2     *x = 1.0;
3 }
4
1+ #pragma omp declare target
2 void compute(double *x){
3     *x = 1.0;
4 }
5

```

- Experiment with the different ways to add the `declare target` directive from the earlier slide. Currently, the first two methods work with the `amdclang` compiler, and the `begin` keyword is not required. If the end directive is not present, the compiler will generate a warning to include it:

```
warning: expected '#pragma omp end declare target' at end of file to match '#pragma
omp declare target' [-Wsource-uses-openmp]
```

C Language device subroutine examples – with global data

- For global data, the constants variable is initialized in another file. An extern declaration is added to the compute.c file to make it accessible and global scope for initialization. We have to add the #pragma omp declare target directive

<pre> 1 2 extern double constants[10]; 3 4 void compute(int cindex, double *x){ 5 *x = 1.0 + constants[cindex]; 6 } </pre>	→	<pre> 1 2+ #pragma omp declare target 3 extern double constants[10]; 4+ #pragma omp end declare target 5 6+ #pragma omp declare target 7 void compute(int cindex, double *x){ 8 *x = 1.0 + constants[cindex]; 9 } 10+ #pragma omp end declare target </pre>
--	---	---

- Then in the global_data.c file where the constants are set, we also need to add the declare target directive.

<pre> 1 double constants[10] = {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0}; </pre>	...	<pre> 1+ #pragma omp declare target 2 double constants[10] = {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0}; 3+ #pragma omp end declare target </pre>
---	-----	---

Fortran Language device subroutine example – part 1

- We'll first look at the target enter data map explicit mapping in 05_device_routine_enter_data directory. In the device_routine.f90 file, these are the changes need to move the computation to the GPU device. Note that the array syntax has to be expanded to an explicit for loop and the pragma added¹. But with these changes, you are still not done because of the call to the compute subroutine.

<pre> 39 !---initialisation 40 x = -1.0_rk 41 !--- call a device subroutine in kernel 42 do k=1,N 43 call compute(x(k)) 44 45 46 47 48 49 50 51 sum = sum + x(k) 52 53 54 55 Write(*,'(A,F0.12)') "Result: sum of x is ",sum </pre>	→	<pre> 39+ !\$omp target enter data map(alloc:x(1:N)) 40 !---initialisation 41+ !\$omp target teams distribute parallel do 42+ do k=1,N 43+ x(k) = -1.0_rk 44+ 45 46+ !\$omp target teams distribute parallel do 47 do k=1,N 48 call compute(x(k)) 49+ !x(k) = 1.0_rk 50 51 52 53 54 55 56+ !\$omp target teams distribute parallel do reduction(+:sum) 57 do k=1,N 58 sum = sum + x(k) 59 60 61 62 Write(*,'(A,F0.12)') "Result: sum of x is ",sum 63+ 64+ !\$omp target exit data map(delete:x) </pre>
---	---	---

Fortran Language device subroutine example – part 2

- For the `compute.f90` file, we have to mark the subroutine to be compiled for the GPU device. Note that for Fortran, the `!$omp declare target` directive goes inside the subroutine rather than outside as is done for the C subroutine.

<pre> 6 subroutine compute(x) 7 implicit none 8 /-----/ 9 !example routine called from kernel 10 !--- variables 11 integer,parameter :: rk=8 12 real(kind=rk),intent(inout) :: x 13 !x a value (from array) </pre>	<pre> 6 subroutine compute(x) 7 implicit none 8+ !\$omp declare target 9 /-----/ 10 !example routine called from kernel 11 !--- variables 12 integer,parameter :: rk=8 13 real(kind=rk),intent(inout) :: x 14 !x a value (from array) </pre>
---	--

C++ Language device subroutine example – member function part 1

- The first example is a straightforward example with the main program called `bigscience.cc` and the class `Science` defined in `Science.hh`. To port the loop with the call to the compute subroutine in the `Science` class, we add a `pragma` around the for loop. We can uncomment the `requires unified_shared_memory` for the unified shared memory version. Otherwise, we might want to add an explicit `map` clause to the `target teams loop` directive.

```
7 #include "Science.hh"
8
9 using namespace std;
10
11 int main(int argc, char *argv[]){
12     Science myscienceclass;
13
14     int N=10000;
15     double *x = new double[N];
16
17
18     for (int k = 0; k < N; k++){
19         myscienceclass.compute(&x[k], N);
20     }
```



```
7 #include "Science.hh"
8
9 using namespace std;
10+
11+ //#pragma omp requires unified_shared_memory
12
13 int main(int argc, char *argv[]){
14     Science myscienceclass;
15
16     int N=10000;
17     double *x = new double[N];
18
19
20+ #pragma omp target teams loop
21     for (int k = 0; k < N; k++){
22         myscienceclass.compute(&x[k], N);
23     }
```

C++ Language device subroutine example – member function part 2

- Now what do we need to do for the `Science` class in `Science.hh`?

```
1  class Science
2  {
3
4  public:
5      double init_value;
6
7  void compute(double *x, int N){
8      |   *x = 1.0;
9      |   }
10 };
```

- Actually, we don't have to do anything! Why? In general, methods that are defined in the `Class` file are in-lined into the calling site. This is somewhat dependent on the compiler and the size of the method.
- In this case, the `compute` method is in-lined into the `bigscience` loop and the compiler will properly generate the device code there.
- The in-lining is key for performance even on the CPU because it removes the overhead of calling the method.

C++ Language device subroutine example – external member function

- Now let's look at the case where the method is defined in another file.

```

1  class Science
2  {
3
4  public:
5      double init_value;
6
7      void compute(double *x, int N);
8  };

```

Before

```

1  #include "Science.hh"
2
3  void Science::compute(double *x, int N){
4      *x = 1.0;
5  }

```

- In this case we have to tell the compiler to generate the method for the GPU because it is generally not inlined. We add the directive in the `Science_member_functions.cc`. No change is needed in the `Science.hh` file.

```

1  #include "Science.hh"
2
3  #pragma omp declare target
4  void Science::compute(double *x, int N){
5      *x = 1.0;
6  }
7  #pragma omp end declare target

```

After



C++ Language device subroutine example – virtual methods part 1

- Let's up the challenge a little bit. How about if we are using class inheritance and redefining our method in the new class. We have added two more files for this example:
 - HotScience.hh
 - HotScience_member_functions.cc
- We override the compute member function definition in HotScience_member_functions.cc
- We also change the bigscience.cc code to use the HotScience class instead of the Science class

```
7 #include "HotScience.hh"
8
9 using namespace std;
10
11 int main(int argc, char *argv[]){
12     HotScience myscienceclass;
13
14     int N=10000;
15     double *x = new double[N];
16
17
18     for (int k = 0; k < N; k++){
19         myscienceclass.compute(&x[k], N);
20     }
```

```
7 #include "HotScience.hh"
8
9 using namespace std;
10+
11+ //pragma omp requires unified_shared_memory
12
13 int main(int argc, char *argv[]){
14     HotScience myscienceclass;
15
16     int N=10000;
17     double *x = new double[N];
18
19
20+ #pragma omp target teams loop
21     for (int k = 0; k < N; k++){
22         myscienceclass.compute(&x[k], N);
23     }
```

C++ Language device subroutine example – virtual methods part 1

- Since it is the derived class member function that is called from the target region, that is the one we need to put the directives around. So in `HotScience_member_functions` we add the pragmas as shown in the comparison of the original file and the ported version for the GPU device.

```
1 #include "HotScience.hh"
2
3 void HotScience::compute(double *x, int N){
4     *x = 5.0;
5 }
```

```
1 #include "HotScience.hh"
2
3+ #pragma omp declare target
4 void HotScience::compute(double *x, int N){
5     *x = 5.0;
6 }
7+ #pragma omp end declare target
```

Portability and testing

- The compiling of a device function and data is a relatively new addition to the OpenMP standard
- Implementing this in a compiler is complex
- Use the latest version of compilers that you possibly can
- It is suggested that you test the pattern that you would like to use in some simple examples such as these to find out what works in the various compilers you want to use
- If possible, keep things relatively simple in any application code that you plan to port to the GPU: complexity can challenge compilers to generate proper code

ROCm Note:

- ROCm and llvm trunk are able to build device routines marked as `declare target` to be used from within target regions.
- The OpenMP 5.0 specification introduced "implicit declare target" where the following code in the same file (compilation unit) will produce the desired effect:

```
int foo(int x) {
    return x+1;
}

int main() {
    int *a = new int[N];

    #pragma omp target teams loop
    for(size_t i = 0; i < N; i++)
        a[i] = foo(i);
}
```

- foo will be implicitly made "declare target" by the compiler because at the call site in the target region, the *definition* of foo is available.
- Conversely, if the definition of foo is in a separate file, then implicit declare target will not be able to build it for the device, and unless the programmer adds a "declare target" around it, it will result in a linker error.
- The second point is that global variables are accessible from within declare target functions when using ROCm.
- As a shortcut, the clang compiler provides the flag `-fopen-force-usm` to add `#pragma requires unified_shared_memory` to every file

OpenMP optimizations

Oct 28-30, 2025

AMD @ CASTIEL HPC

1. Reduce data transfer between host and device

Data transfers between the host and device are one of the largest impacts on performance. There are several ways to reduce this performance bottleneck

- Making data regions on device larger with multiple compute operations
- Only moving data that is needed to the device or back to the host
- Allocating data on the device instead of moving it there

If data must be moved

- Use pinned memory buffers
- Place CPU and GPU close together
 - Same NUMA memory region
 - Fewer data links for transfers
- Overlap data communication and computation

Reducing data transfers is important for discrete GPUs. It is far less important for APUs like the MI300A.

2. Moving memory allocations and frees out of inner loops

- Memory allocations and frees are performed on the host. They
 - can cause synchronizations
 - are serial operations
- Moving memory allocations out of the inner loops and the target regions can substantially improve performance
 - Allocate maximum memory needed for loop iterations
- Moving memory allocations out of kernels is even more important
- To explore this problem, we'll look at the following examples. First set up the environment. Try the language of your choice by selecting the language in the directory path for the examples.

```
git clone https://github.com/amd/HPCTrainingExamples  
cd HPCTrainingExamples/Pragma_Examples/OpenMP/[C|CXX|Fortran]/optimizations/Allocations/  
module load [amdclang|amdflang-new]  
export HSA_XNACK=1
```

2. Moving memory allocations and frees out of inner loops

The 1_alloc_problem example has the allocation in the inner loop

```
cd 1_alloc_problem
make
./alloc_prob
```

- The 2_opt_allocation directory moves the allocation outside the loop

```
cd 2_opt_allocation
make
./opt_allocation
```

- The speedup from moving the allocation out of the loop will be 3-6x times faster

```
for (int iter = 0; iter < Niter; iter++){
    // Allocate memory for each vector
    double *a = (double *) malloc(n*sizeof(double));
    double *b = (double *) malloc(n*sizeof(double));
    double *c = (double *) malloc(n*sizeof(double));
```

```
    free(a);
    free(b);
    free(c);
}
```

3. Use memory pools

- Memory pools can help with frequent memory allocations and memory fragmentations
- Recommended if you cannot move allocations outside the inner computational loops
- In this example, we use the Umpire memory management package from LLNL
- Going to the example with the changes to use the memory pool allocator.

```
cd 3_memorypool
```

- Install umpire memory pool

```
./umpire_setup.sh
```

```
export UMPIRE_PATH=${PWD}/Umpire_install
```

- Build the example

```
make
```

```
./memorypool
```

- Runtime is about the same or slightly slower then when we moved the memory allocation outside the inner loop.

Reducing memory allocation costs is particularly important on APUs like the MI300A.

4. Large page sizes

Default page mapping:

16384 TLB entries * 4 KiB page size = 64MiB addressable memory. Fine for Word, but not for scientific applications

Our applications need much more memory, so they spend a lot of time on table walks. Increasing page size (large pages) to 2 MiB can yield > 10% speedup.

For random access patterns (sparse) the miss rate can be as high as 40% of memory accesses.

How do I get large page sizes?

It varies a bit by the Operating System and Software Stack

HPE (Cray) usually has modules with a selection of different page sizes. 2 MiB is sufficient to address most of memory

On many Linux distributions

Inspect `/sys/kernel/mm/transparent_hugepage/enabled`

Set value to always with

```
echo always > /sys/kernel/mm/transparent_hugepage/enabled
```

Set the Linux kernel parameter `transparent_hugepage` to always in the `.cfg` file for your system

Allocating memory on the GPU with `hipMalloc` will get 2MiB page sizes

5. Kernel Tuning

- There are a couple of common clauses used to tune kernel performance
 - `num_threads(##)` – use this size work group size for the kernel call
 - `thread_limit(##)` – the kernel will never be called with more than this work group size
- These can help with optimizing kernel memory and register usage
- Also, it may help to add the `collapse(#)` and `tile(#)` clauses
- Hoisting invariants out of the kernel loop and reducing the integer calculations can be helpful
- By setting `LIBOMPTARGET_KERNEL_TRACE=1` or `2`, we can see what the OpenMP® runtime does behind the scenes.
 - Setting to `1` shows the name of every kernel, number of teams, threads, and register usage.
 - Setting to `2` prints timing and data transfer information
- `LIBOMPTARGET_DEBUG=1` will show more information about data transfer operations and kernel launch
- HPE/Cray
 - `CRAY_ACC_DEBUG=[1,2,3]`
 - `-hlist=aimd` at compile time

5. Kernel Tuning

- These examples are at <https://github.com/AMD/HPCTrainingExamples> in the HPCTrainingExamples/Pragma_Examples/OpenMP/CXX/kernel_pragmas directory
- We'll experiment with different optimizations with pragmas. By setting LIBOMPTARGET_KERNEL_TRACE=1 or 2, we can see what the OpenMP® runtime does behind the scenes.
 - Setting to 1 shows the name of every kernel, number of teams, threads, and register usage.
 - Setting to 2 prints timing and data transfer information
- LIBOMPTARGET_DEBUG=1 will show more information about data transfer operations and kernel launch
- HPE/Cray
 - CRAY_ACC_DEBUG=[1,2,3]
 - -hlist=aimd at compile time

Kernel1.cc

```
module load amdclang
export HSA_XNACK=1
export LIBOMPTARGET_KERNEL_TRACE=1
mkdir build && cd build
CXX=amdclang++ cmake ..
make
./kernel1
```

LIBOMPTARGET_KERNEL_TRACE Report

```
DEVID: 0 SGN:2 ConstWGSize:256 args: 3 teamsXthrs:( 391X 256) reqd:( 0X 0) lds_usage:9784B sgpr_count:106 vgpr_count:58 sgpr_spill_count:39 vgpr_spill_count:0
tripcount:100000 rpc:1 n:__omp_offloading_3d_1a2def2_main_I52
DEVID: 0 SGN:2 ConstWGSize:256 args: 5 teamsXthrs:( 391X 256) reqd:( 0X 0) lds_usage:9784B sgpr_count:106 vgpr_count:56 sgpr_spill_count:47 vgpr_spill_count:0
tripcount:100000 rpc:1 n:__omp_offloading_3d_1a2def2_Z5daxpyidPdS_S_I97
```

Adding thread clauses

kernel2.cc

- Change number of threads – add `num_threads(64)`

LIBOMPTARGET_KERNEL_TRACE Report

DEVID: 0 SGN:2 ConstWGSize:64 args: 3 teamsXthrds:(416X 64) reqd:(0X 64) lds_usage:9784B sgpr_count:106 vgpr_count:59 sgpr_spill_count:42 vgpr_spill_count:0
tripcount:100000 rpc:1 n:__omp_offloading_3d_1a2def6_main_I52

DEVID: 0 SGN:2 ConstWGSize:64 args: 5 teamsXthrds:(416X 64) reqd:(0X 64) lds_usage:9784B sgpr_count:106 vgpr_count:57 sgpr_spill_count:48 vgpr_spill_count:0
tripcount:100000 rpc:1 n:__omp_offloading_3d_1a2def6_Z5daxpyidPdS_S_I97

kernel3.cc

- Add thread limit for kernel – `num_threads(64) thread_limit(64)`

LIBOMPTARGET_KERNEL_TRACE Report

DEVID: 0 SGN:2 ConstWGSize:64 args: 3 teamsXthrds:(416X 64) reqd:(0X 64) lds_usage:9784B sgpr_count:106 vgpr_count:55 sgpr_spill_count:37 vgpr_spill_count:0
tripcount:100000 rpc:1 n:__omp_offloading_3d_1a2def8_main_I52

DEVID: 0 SGN:2 ConstWGSize:64 args: 5 teamsXthrds:(416X 64) reqd:(0X 64) lds_usage:9784B sgpr_count:106 vgpr_count:53 sgpr_spill_count:45 vgpr_spill_count:0
tripcount:100000 rpc:1 n:__omp_offloading_3d_1a2def8_Z5daxpyidPdS_S_I97

Disclaimer

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

THIS INFORMATION IS PROVIDED 'AS IS.' AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

© 2025 Advanced Micro Devices, Inc and OpenMP® Architecture Review Board. All rights reserved.

AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

LLVM is a trademark of LLVM Foundation

The OpenMP name and the OpenMP logo are registered trademarks of the OpenMP Architecture Review Board

AMD 