OpenMP® Target Offloading for AMD Instinct™ GPUs and APUs

Dr.-Ing. Michael Klemm
Principal Member of Technical Staff
Compilers, Languages, Runtimes & Tools
Machine Learning & Software Engineering

Chief Executive Officer
OpenMP Architecture Review Board

Credits: AMD ROCm OpenMP Team

AMD
together we advance_

Agenda

1. OpenMP® Target Offload
2. OpenMP® Unified Shared Memory
3. Unified Shared Memory via Zero Copy
4. Outlook: OpenMP® 6.0 Self Maps

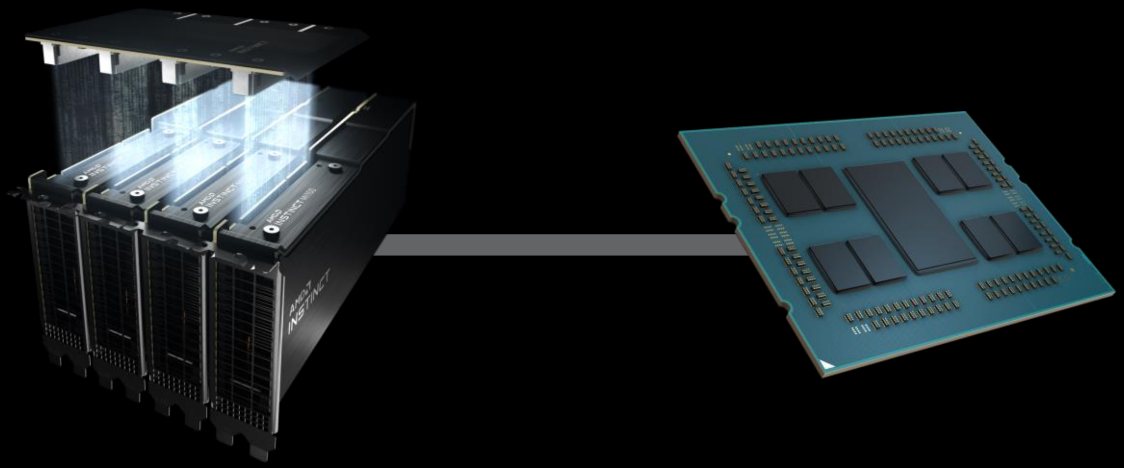
OpenMP® Target Offload

OpenMP® Target Offload

- OpenMP® language-feature to use an accelerator (e.g., a GPU) for parts of your program
- Enables standardized way for offloading to an accelerator as opposed to CUDA® or HIP
- Expressed in terms of **`target`** regions
- Target regions have a data environment that is maintained/manipulated via **`map`** clauses
- Target regions execute one or multiple teams of threads on a device
- And much more …

OpenMP® Device Model

- As of version 4.0 the OpenMP® API supports accelerators/coprocessors.
- Device model:
  - One host for “traditional” multi-threading
  - Multiple accelerators/coprocessors of the same kind for offloading
  - Devices are accessible though a device ID (from 0 to *n*-1 for *n* devices)
- OpenMP device model is agnostic of actual technology. In theory, devices only need to
  - be able to receive data from the host and send data back and
  - perform computation upon request.



AMD
together we advance_

OpenMP® Target Offload Code Example

```
int main(int argc, char **argv) {
  int *vals = new int[1024];




  for(int i = 0; i < 1024; ++i) {
    vals[i] = 1;
  }


  for(const auto vi : vals)
    std::cout << vi << ‘\n’;
  return 0;
}
```

OpenMP® Target Offload Code Example

```
int main(int argc, char **argv) {
  int *vals = new int[1024];


  #pragma omp target teams distribute parallel for
  for(int i = 0; i < 1024; ++i) {
    vals[i] = 1;
  }


  for(const auto vi : vals)
    std::cout << vi << '\n';
  return 0;
}
```


AMD
together we advance_

OpenMP® Target Offload Code Example – Code Transformation

```
int main(int argc, char **argv) {
  int *vals = new int[1024];



  #pragma omp target teams distribute parallel for
  for(int i = 0; i < 1024; ++i) {
    vals[i] = 1;
  }



  for(const auto vi : val
    std::cout << vi << '\
  return 0;
}
```

```
#pragma omp target teams
{
  // compute bs depending on number of teams on GPU
  #pragma omp distribute
  for (int ii = 0; ii < n; ii += bs) {
    #pragma omp parallel for
    for (int i = ii; i < min(i + bs, 1024); ++i) {
      vals[i] = 1;
} } }
```

OpenMP® Target Offload Code Example – Fortran

```fortran
program offload
  integer, dimension(:), allocatable :: vals

  allocate(vals(1024))

  !$omp target teams workdistribute  ! OpenMP 6.0 feature
  vals = 1
  !omp end target teams workdistribute

  print 100, vals
  100 format (1X,1I0)
end program offload
```

OpenMP® Target Offload Code Example – Fortran

```
program offload
  integer, dimension(:), allocatable :: vals

  allocate(vals(1024))

  !$omp target teams workdistribute  ! OpenMP 6.0 feature
  vals = 1
  !omp end target teams workdistribute

  print 100, vals
  100 format (1X,1I0)
end program offload
```

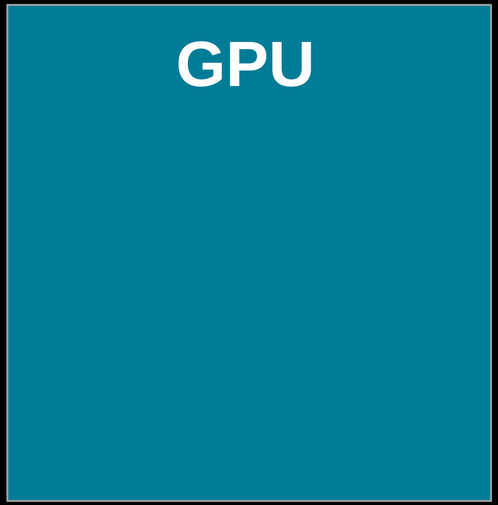
```
!$omp target teams distribute &
      parallel do
do i = 1, n
    vals(i) = 1
end do
```

Data Environments and Memory Transfer

```
int main(int argc, char **argv) {
  int *vals = new int[1024];



  #pragma omp target teams distribute parallel for map(tofrom: vals[0:1024])
  for(int i = 0; i < 1024; ++i) {
    vals[i] = 1;
  }



  for(const auto vi : vals)
    std::cout << vi << '\n';
  return 0;
}
```

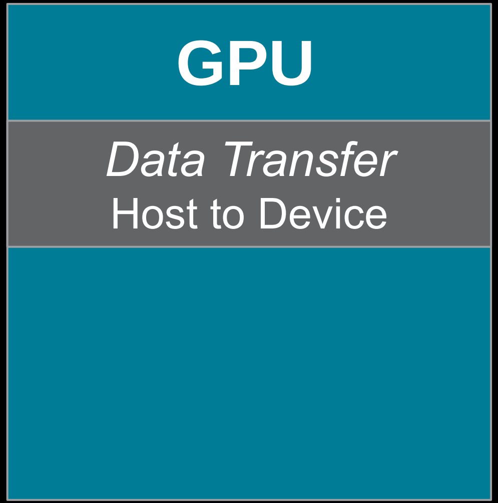
Data Environments and Memory Transfer

```
int main(int argc, char **argv) {
  int *vals = new int[1024];


  #pragma omp target teams distribute parallel for map(tofrom:
  for(int i = 0; i < 1024; ++i) {
    vals[i] = 1;
  }


  for(const auto vi : vals)
    std::cout << vi << '\n';
  return 0;
}
```



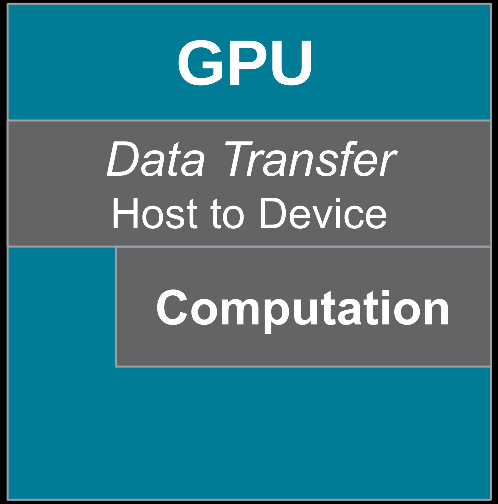
Data Environments and Memory Transfer

```
int main(int argc, char **argv) {
  int *vals = new int[1024];

  #pragma omp target teams distribute parallel for map(tofrom:
  for(int i = 0; i < 1024; ++i) {
    vals[i] = 1;
  }

  for(const auto vi : vals)
    std::cout << vi << ‘\n’;
  return 0;
}
```



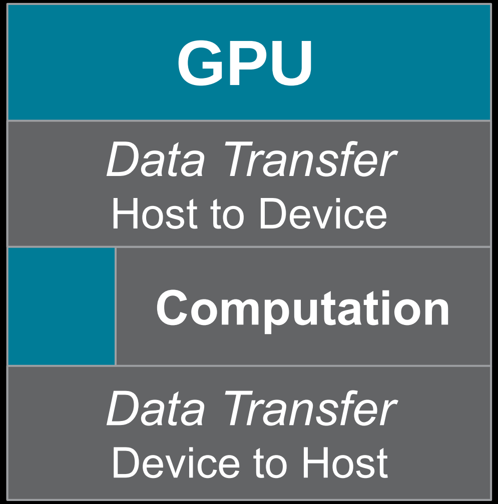
Data Environments and Memory Transfer

```
int main(int argc, char **argv) {
  int *vals = new int[1024];

  #pragma omp target teams distribute parallel for map(tofrom:
  for(int i = 0; i < 1024; ++i) {
    vals[i] = 1;
  }

  for(const auto vi : vals)
    std::cout << vi << ‘\n’;
  return 0;
}
```



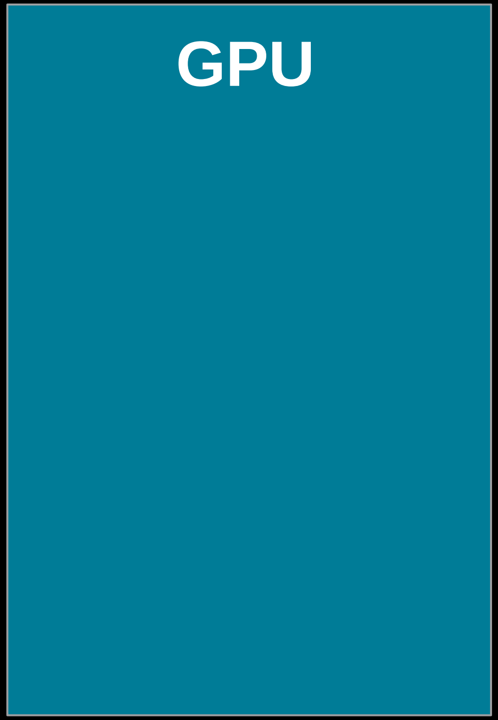
Data Environments and Memory Transfer

```
int main(int argc, char **argv) {
  int *vals = new int[1024];

#pragma omp target teams distribute parallel for map(tofrom:
for(int i = 0; i < 1024; ++i) {
  vals[i] = 1;
}

  for(const auto vi : vals)
    std::cout << vi << ‘\n’;
  return 0;
}
```



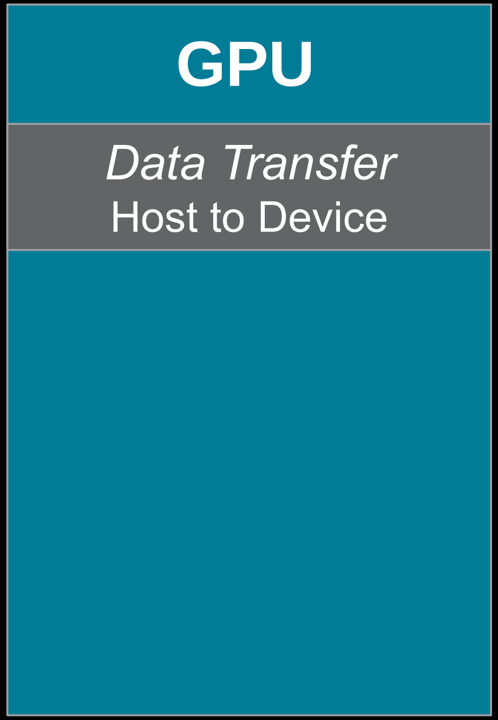
Data Environments and Memory Transfer

```
int main(int argc, char **argv) {
  int *vals = new int[1024];

  #pragma omp target enter data map(to:vals[0:1024])

  #pragma omp target teams distribute parallel for map(tofrom:
  for(int i = 0; i < 1024; ++i) {
    vals[i] = 1;
  }

  #pragma omp target exit data map(from:vals[0:1024])

  for(const auto vi : vals)
    std::cout << vi << ‘\n’;
  return 0;
}
```



Data Environments and Memory Transfer

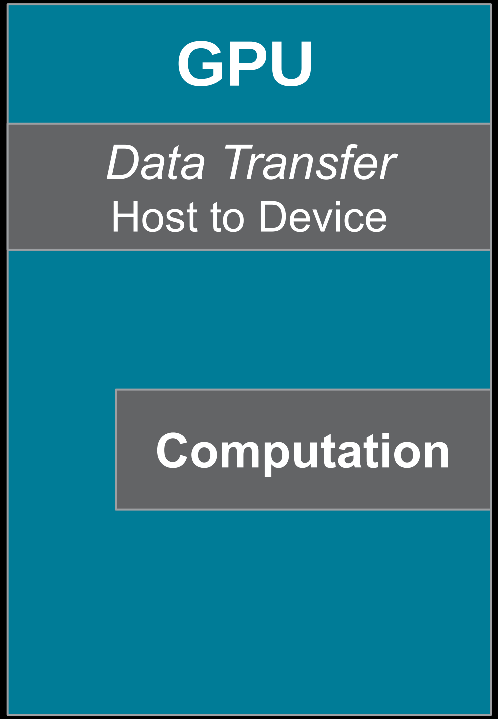
```
int main(int argc, char **argv) {
  int *vals = new int[1024];

  #pragma omp target enter data map(to:vals[0:1024])

  #pragma omp target teams distribute parallel for map(tofrom:
  for(int i = 0; i < 1024; ++i) {
    vals[i] = 1;
  }

  #pragma omp target exit data map(from:vals[0:1024])

  for(const auto vi : vals)
    std::cout << vi << '\n';
  return 0;
}
```



AMD
together we advance_

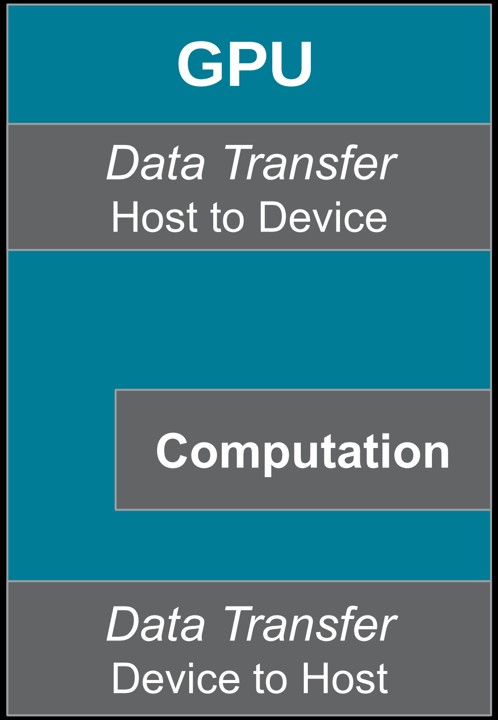
Data Environments and Memory Transfer

```
int main(int argc, char **argv) {
  int *vals = new int[1024];

  #pragma omp target enter data map(to:vals[0:1024])

  #pragma omp target teams distribute parallel for map(tofrom:
  for(int i = 0; i < 1024; ++i) {
    vals[i] = 1;
  }

  #pragma omp target exit data map(from:vals[0:1024])

  for(const auto vi : vals)
    std::cout << vi << '\n';
  return 0;
}
```



Data Environments and Memory Transfer

```
int main(int argc, char **argv) {
  int *vals = new int[1024];

  #pragma omp target enter data map(to:vals[0:1024])

  #pragma omp target teams distribute parallel for map(tofrom:
  for(int i = 0; i < 1024; ++i) {
    vals[i] = 1;
  }

  #pragma omp target exit data map(from:vals[0:1024])

  for(const auto vi : vals)
    std::cout << vi << ‘\n’;
  return 0;
}
```



AMD
together we advance_

Asynchronous Offloads

- OpenMP `target` constructs are synchronous by default
  - The encountering host thread awaits the end of the `target` region before continuing

```
#pragma omp task
    init_data(a);

#pragma omp target map(to:a[:N]) map(from:x[:N])
    compute_1(a, x, N);

#pragma omp target map(to:b[:N]) map(from:y[:N])
    compute_2(b, y, N);

#pragma omp target map(to:x[:N],y[:N]) map(from:z[:N])
    compute_3(x, y, z, N);
```

Asynchronous Offloads

- OpenMP `target` constructs are synchronous by default
  - The encountering host thread awaits the end of the `target` region before continuing
  - The `nowait` clause makes the target constructs asynchronous
    (in OpenMP lingo: they become an OpenMP task)

```
#pragma omp task
    init_data(a);

#pragma omp target map(to:a[:N]) map(from:x[:N])       nowait
    compute_1(a, x, N);

#pragma omp target map(to:b[:N]) map(from:y[:N])       nowait
    compute_2(b, y, N);

#pragma omp target map(to:x[:N],y[:N]) map(from:z[:N]) nowait
    compute_3(x, y, z, N);
```

Asynchronous Offloads

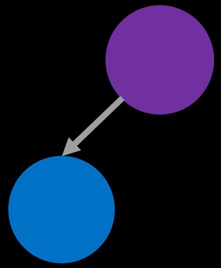
- OpenMP `target` constructs are synchronous by default
  - The encountering host thread awaits the end of the `target` region before continuing
  - The `nowait` clause makes the target constructs asynchronous
    (in OpenMP lingo: they become an OpenMP task)

```
#pragma omp task                                             depend(out:a)
    init_data(a);

#pragma omp target map(to:a[:N]) map(from:x[:N])       nowait
    compute_1(a, x, N);

#pragma omp target map(to:b[:N]) map(from:y[:N])       nowait
    compute_2(b, y, N);

#pragma omp target map(to:x[:N],y[:N]) map(from:z[:N]) nowait
    compute_3(x, y, z, N);
```

Asynchronous Offloads

- OpenMP `target` constructs are synchronous by default
  - The encountering host thread awaits the end of the `target` region before continuing
  - The `nowait` clause makes the target constructs asynchronous
    (in OpenMP lingo: they become an OpenMP task)

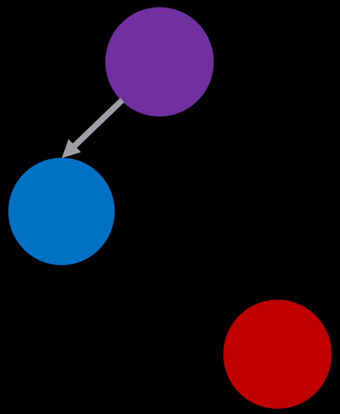
```
#pragma omp task                                                         depend(out:a)
    init_data(a);

#pragma omp target map(to:a[:N]) map(from:x[:N])       nowait   depend(in:a) depend(out:x)
    compute_1(a, x, N);

#pragma omp target map(to:b[:N]) map(from:y[:N])       nowait
    compute_2(b, y, N);

#pragma omp target map(to:x[:N],y[:N]) map(from:z[:N]) nowait
    compute_3(x, y, z, N);
```



Asynchronous Offloads

- OpenMP `target` constructs are synchronous by default
  - The encountering host thread awaits the end of the `target` region before continuing
  - The `nowait` clause makes the target constructs asynchronous
    (in OpenMP lingo: they become an OpenMP task)

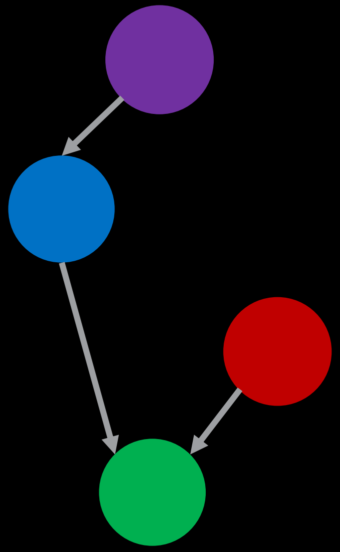
```
#pragma omp task                                                    depend(out:a)
    init_data(a);

#pragma omp target map(to:a[:N]) map(from:x[:N])       nowait   depend(in:a) depend(out:x)
    compute_1(a, x, N);

#pragma omp target map(to:b[:N]) map(from:y[:N])       nowait   depend(in:b) depend(out:y)
    compute_2(b, y, N);

#pragma omp target map(to:x[:N],y[:N]) map(from:z[:N]) nowait
    compute_3(x, y, z, N);
```



Asynchronous Offloads

- OpenMP `target` constructs are synchronous by default
  - The encountering host thread awaits the end of the `target` region before continuing
  - The `nowait` clause makes the target constructs asynchronous
    (in OpenMP lingo: they become an OpenMP task)

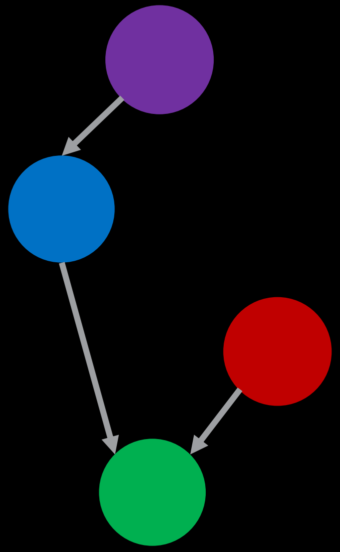
```
#pragma omp task                                                   depend(out:a)
    init_data(a);

#pragma omp target map(to:a[:N]) map(from:x[:N])        nowait   depend(in:a) depend(out:x)
    compute_1(a, x, N);

#pragma omp target map(to:b[:N]) map(from:y[:N])        nowait   depend(in:b) depend(out:y)
    compute_2(b, y, N);

#pragma omp target map(to:x[:N],y[:N]) map(from:z[:N]) nowait   depend(in:x) depend(in:y)
    compute_3(x, y, z, N);
```



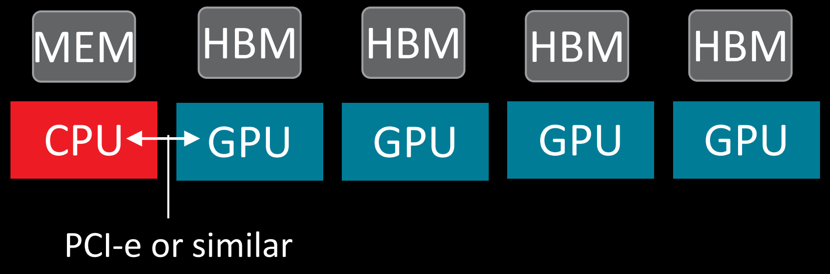
Asynchronous Offloads

- OpenMP `target` constructs are synchronous by default
  - The encountering host thread awaits the end of the `target` region before continuing
  - The `nowait` clause makes the target constructs asynchronous
    (in OpenMP lingo: they become an OpenMP task)

```
#pragma omp task                                                   depend(out:a)
    init_data(a);

#pragma omp target map(to:a[:N]) map(from:x[:N])       nowait   depend(in:a) depend(out:x)
    compute_1(a, x, N);

#pragma omp target map(to:b[:N]) map(from:y[:N])       nowait   depend(in:b) depend(out:y)
    compute_2(b, y, N);

#pragma omp target map(to:x[:N],y[:N]) map(from:z[:N]) nowait   depend(in:x) depend(in:y)
    compute_3(x, y, z, N);

#pragma omp taskwait
```



OpenMP® Target Offload on Discrete GPUs (Default Mode)

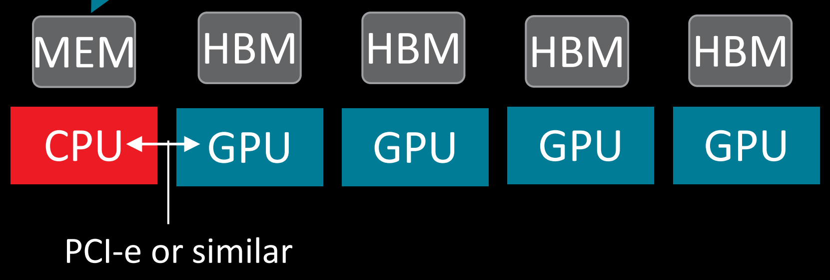



```
int main() {
  int *vals = new int[1024];

  #pragma omp target map(tofrom: vals[0:1024])
  {
    for(int i = 0; i < 1024; i++)
      vals[i] = i;
  }
}
```

For best performance, programmers minimize transfers between host and device by placing map clauses at the beginning and ending of an application

OpenMP® Target Offload on Discrete GPUs (Default Mode)

OS allocators allocate host memory

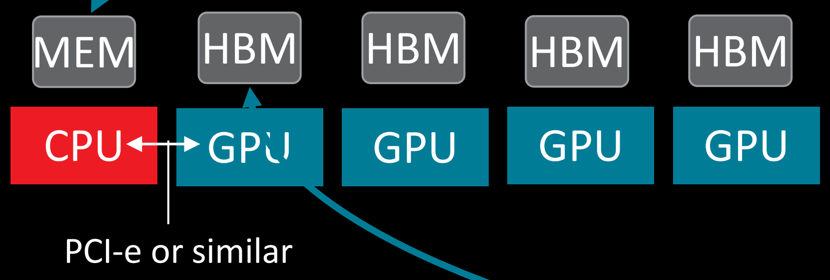



```
int main() {
  int *vals = new int[1024];

  #pragma omp target map(tofrom: vals[0:1024])
  {
    for(int i = 0; i < 1024; i++)
      vals[i] = i;
  }
}
```

For best performance, programmers minimize transfers between host and device by placing map clauses at the beginning and ending of an application

OpenMP® Target Offload on Discrete GPUs (Default Mode)

OS allocators allocate host memory



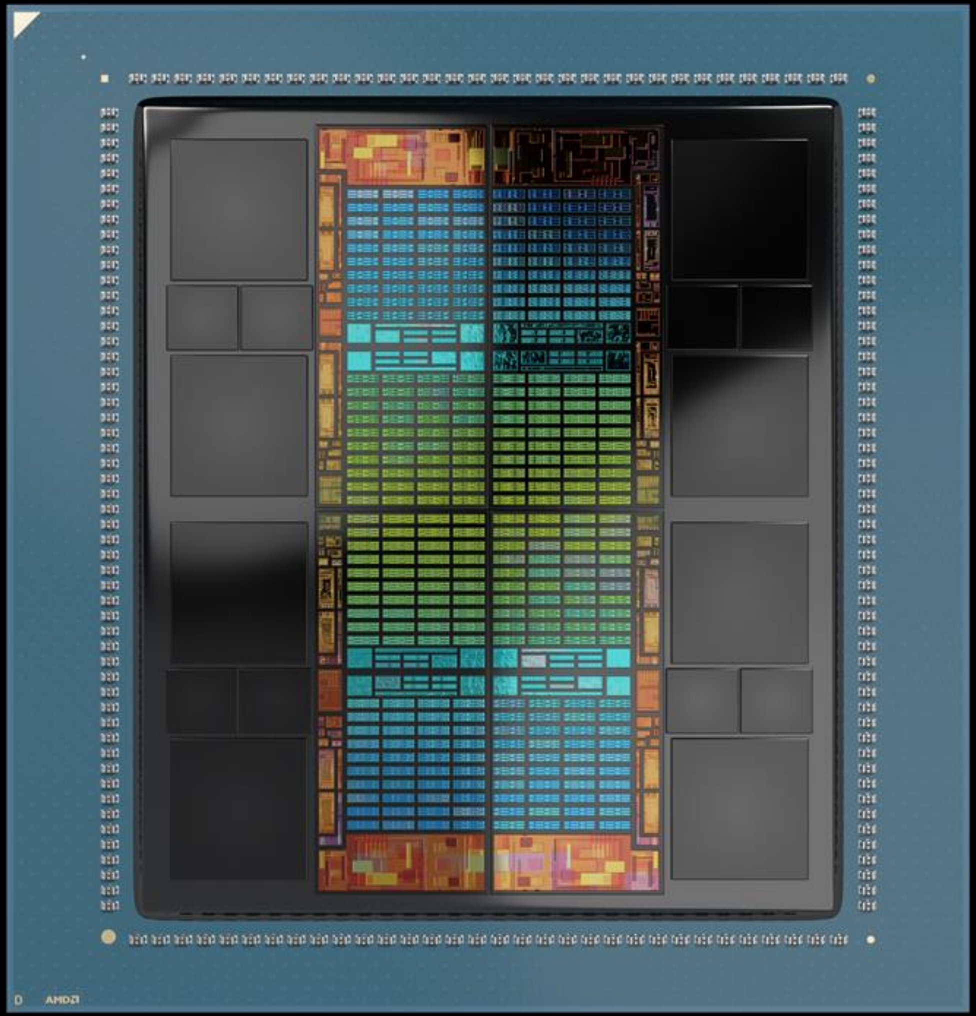

```
int main() {
  int *vals = new int[1024];

  #pragma omp target map(tofrom: vals[0:1024])
  {
    for(int i = 0; i < 1024; i++)
      vals[i] = i;
  }
}
```

- Allocate “d_vals” 1024 integers size on GPU HBM
- Before kernel launch, (DMA) copy `vals` into “d_vals”
- After kernel completion, (DMA) copy “d_vals” back to `vals`

For best performance, programmers minimize transfers between host and device by placing map clauses at the beginning and ending of an application

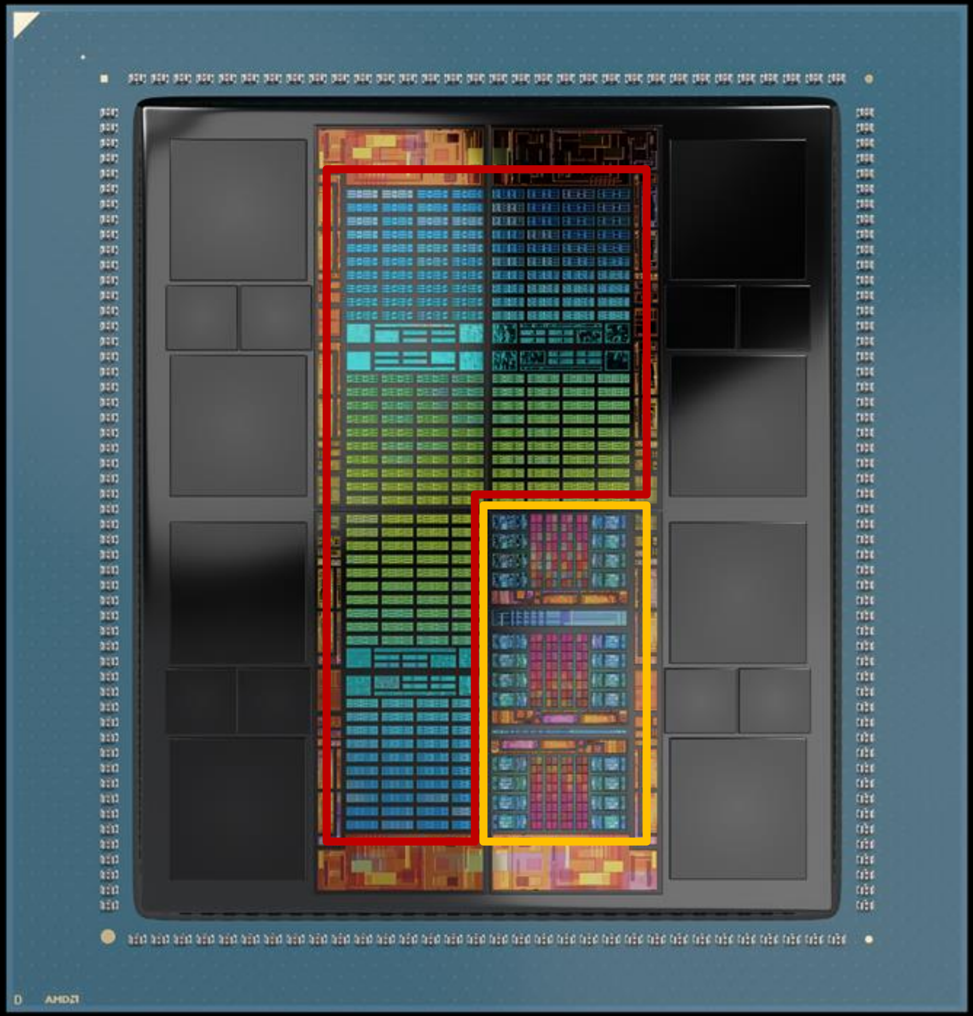
OpenMP® Unified Shared Memory

AMD CDNA™ 3 Architecture – AMD Instinct™ MI300X



- Discrete GPU
- 3D chiplet packaging
- 304 AMD CDNA™ 3 compute units
- 192 GB HBM3
- Discrete GPU architecture, OAM
  - AMD EPYC™ Processor as the host system
  - Use host system for memory capacity beyond GPU HBM capacity
  - Connects via
    - PCIe® Gen 5
    - AMD Infinity Fabric™ Links

AMD CDNA™ 3 Architecture – AMD Instinct™ MI300A



- APU with unified shared memory
- 3D chiplet packaging
- 24 cores, “Zen 4” architecture
- *228* AMD CDNA™ 3 compute units
- 128 GB HBM3
- Unified memory architecture
  - no discrete CPU and GPU memory
  - CPU and GPU access same physical memory

AMD
together we advance_

Unified Shared Memory (USM)

CPU CODE

```
double* in = (double*)malloc(Msize);
double* out = (double*)malloc(Msize);

 for (int i=0; i<M; i++)
   in[i] = …;



 for (int i=0; i<M; i++)
   out[i] = ... in[i] ...;





for (int i=0; i<M; i++)
  … = out[i];
```

Unified Shared Memory (USM)

CPU CODE

```
double* in = (double*)malloc(Msize);
double* out = (double*)malloc(Msize);

 for (int i=0; i<M; i++)
   in[i] = …;



 for (int i=0; i<M; i++)
   out[i] = ... in[i] ...;




for (int i=0; i<M; i++)
  … = out[i];
```

W/O UNIFIED SHARED MEMORY

```
double* in = (double*)malloc(Msize);
double* out = (double*)malloc(Msize);

for (int i=0; i<M; i++)
   in[i] = …;

#pragma omp target teams distribute \
            parallel for \
            map(to:in[0:Msize]) \
            map(from:out[0:Msize])
  for (int i=0; i<M; i++)
    out[i] = ... in[i] ...;



for (int i=0; i<M; i++)
  … = out[i];
```

Unified Shared Memory (USM)

CPU CODE

```
double* in = (double*)malloc(Msize);
double* out = (double*)malloc(Msize);

 for (int i=0; i<M; i++)
   in[i] = …;

 for (int i=0; i<M; i++)
   out[i] = ... in[i] ...;

for (int i=0; i<M; i++)
  … = out[i];
```

W/O UNIFIED SHARED MEMORY

```
double* in = (double*)malloc(Msize);
double* out = (double*)malloc(Msize);

for (int i=0; i<M; i++)
   in[i] = …;

#pragma omp target teams distribute \
            parallel for \
            map(to:in[0:Msize]) \
            map(from:out[0:Msize])
  for (int i=0; i<M; i++)
    out[i] = ... in[i] ...;

for (int i=0; i<M; i++)
  … = out[i];
```

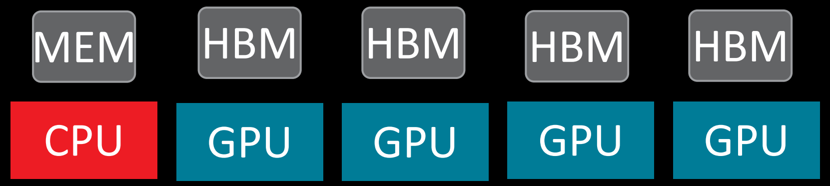
UNIFIED SHARED MEMORY

```
#pragma omp require unified_shared_memory

double* in = (double*)malloc(Msize);
double* out = (double*)malloc(Msize);

for (int i=0; i<M; i++)
   in[i] = …; //writes GPU mem directly

#pragma omp target teams distribute \
            parallel for
  for (int i=0; i<M; i++)
    out[i] = ... in[i] ...;

for (int i=0; i<M; i++)
  … = out[i]; //reads GPU mem directly
```


AMD
together we advance_

USM via Zero Copy

OpenMP® Target Offload on Discrete GPUs (MI300X, USM Mode)

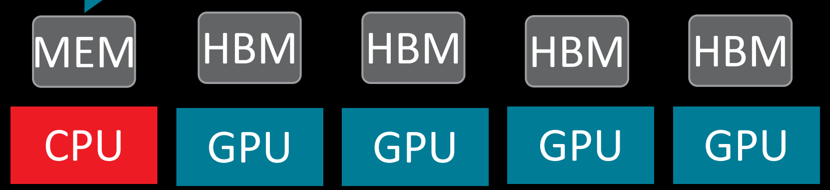



```
#pragma omp requires unified_shared_memory
int main() {
  int *vals = new int[1024];

  #pragma omp target map(tofrom: vals[0:1024])
  {
    for(int i = 0; i < 1024; i++)
      vals[i] = i;
  }
}
```

Driver handles page migrations. Migration depends on allocator being used on host.

OpenMP® Target Offload on Discrete GPUs (MI300X, USM Mode)

OS allocators allocate host memory

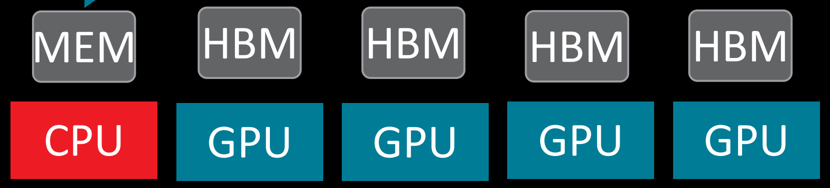



```
#pragma omp requires unified_shared_memory
int main() {
  int *vals = new int[1024];

  #pragma omp target map(tofrom: vals[0:1024])
  {
    for(int i = 0; i < 1024; i++)
      vals[i] = i;
  }
}
```

Driver handles page migrations. Migration depends on allocator being used on host.

OpenMP® Target Offload on Discrete GPUs (MI300X, USM Mode)

OS allocators allocate host memory



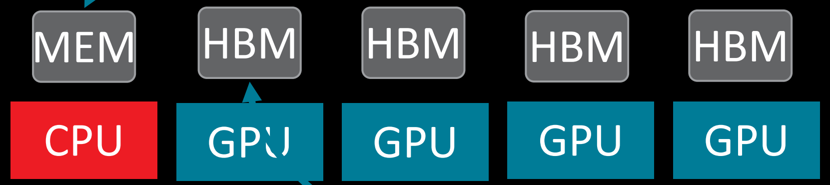

```
#pragma omp requires unified_shared_memory
int main() {
  int *vals = new int[1024];

  #pragma omp target map(tofrom: vals[0:1024])
  {
    for(int i = 0; i < 1024; i++)
      vals[i] = i;
  }
}
```

map: just pass host pointer to kernel

Driver handles page migrations. Migration depends on allocator being used on host.

OpenMP® Target Offload on Discrete GPUs (MI300X, USM Mode)

OS allocators allocate host memory



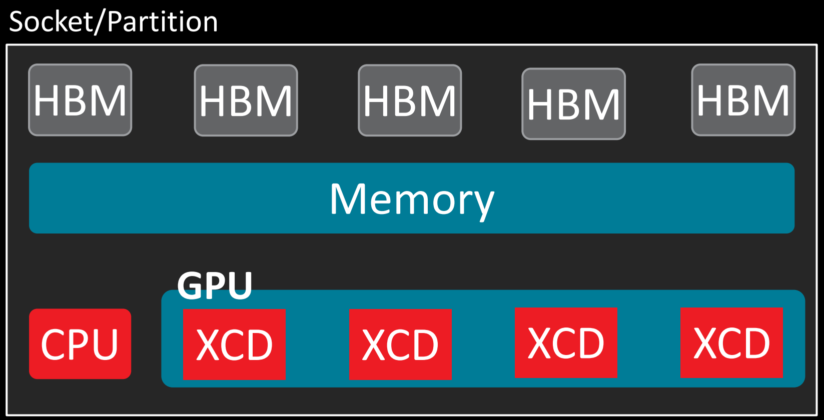

```
#pragma omp requires unified_shared_memory
int main() {
  int *vals = new int[1024];

  #pragma omp target map(tofrom: vals[0:1024])
  {
    for(int i = 0; i < 1024; i++)
      vals[i] = i;
  }
}
```

map: just pass host pointer to kernel

Accessing a page address located on system memory provokes page migration to HBM

Driver handles page migrations. Migration depends on allocator being used on host.

OpenMP® on APUs (MI300A): Zero-Copy Mode

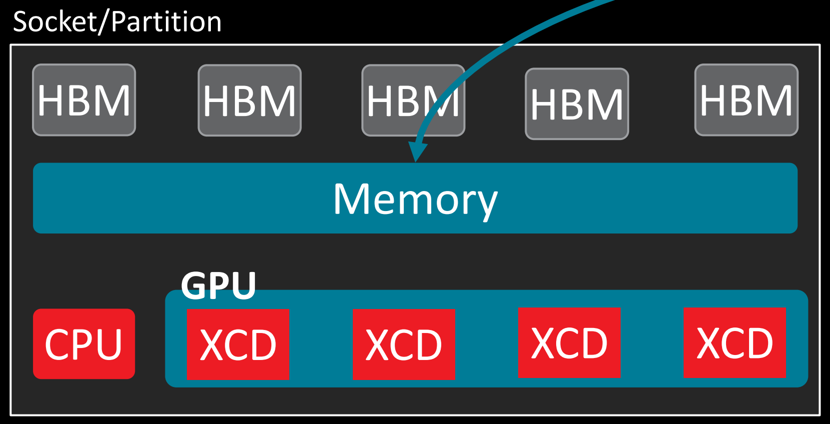



```
int main() {
  int *vals = new int[1024];

  #pragma omp target map(tofrom: vals[0:1024])
  {
    for(int i = 0; i < 1024; i++)
      vals[i] = i;
  }
}
```

* Interleaved allocation across the HBMs on the socket

OpenMP® on APUs (MI300A): Zero-Copy Mode

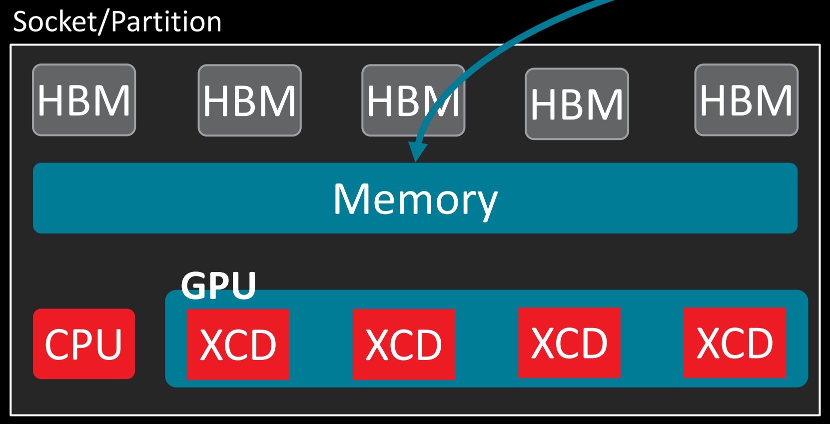
OS allocators allocate unified memory*




```
int main() {
  int *vals = new int[1024];

  #pragma omp target map(tofrom: vals[0:1024])
  {
    for(int i = 0; i < 1024; i++)
      vals[i] = i;
  }
}
```

* Interleaved allocation across the HBMs on the socket

OpenMP® on APUs (MI300A): Zero-Copy Mode



OS allocators allocate unified memory*

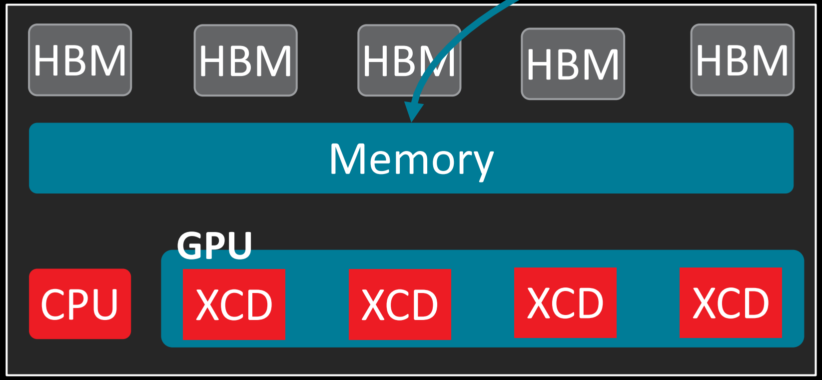
map: just pass pointer to kernel

```
int main() {
  int *vals = new int[1024];

  #pragma omp target map(tofrom: vals[0:1024])
  {
    for(int i = 0; i < 1024; i++)
      vals[i] = i;
  }
}
```

* Interleaved allocation across the HBMs on the socket

OpenMP® on APUs (MI300A): Zero-Copy Mode

Socket/Partition



OS allocators allocate unified memory*

```
int main() {
  int *vals = new int[1024];

  #pragma omp target map(tofrom: vals[0:1024])
  {
    for(int i = 0; i < 1024; i++)
      vals[i] = i;
  }
}
```

map: just pass pointer to kernel

No page migration necessary upon page touch
(in this single socket example)

* Interleaved allocation across the HBMs on the socket

Side-to-side Compiler Code

```
#include <omp.h>



int main() {
    int* data = (int*) malloc(sizeof(int) * 10);

    #pragma omp target teams loop map(tofrom:data[:10])
    for(int i = 0; i < 10; i++) {
        vals[i] += 1;
    }

    // Print the updated array
    return 0;
}
```

```
#include <omp.h>

#pragma omp requires unified_shared_memory

int main() {
    int* vals = (int*) malloc(sizeof(int) * 10);

    #pragma omp target teams loop
    for(int i = 0; i < 10; i++) {
        vals[i] += 1;
    }

    // Print the updated array
    return 0;
}
```

Side-to-side Compiler Code – Host Binary

without USM

```
struct.ident_t = type { i32, i32, i32, i32, ptr }
%struct.__tgt_offload_entry = type { ptr, ptr, i64, i32, i32 }
%struct.__tgt_kernel_arguments = type { i32, i32, ptr, ptr, ptr, ptr, ptr, ptr, i64, i64, [3 x i32], [3 x i32], i32 }

@0 = private unnamed_addr constant [23 x i8] c";unknown;unknown;0;0;;\00", align 1
@1 = private unnamed_addr constant %struct.ident_t { i32 0, i32 2050, i32 0, i32 22, ptr @0 }, align 8
@2 = private unnamed_addr constant %struct.ident_t { i32 0, i32 514, i32 0, i32 22, ptr @0 }, align 8
@3 = private unnamed_addr constant %struct.ident_t { i32 0, i32 2, i32 0, i32 22, ptr @0 }, align 8
@.__omp_offloading_811_d56211f_main_l16.region_id = weak constant i8 0
@.offload_sizes = private unnamed_addr constant [1 x i64] [i64 40]
@.offload_maptypes = private unnamed_addr constant [1 x i64] [i64 35]
@.offloading.entry_name = internal unnamed_addr constant [38 x i8] c"__omp_offloading_811_d56211f_main_l16\00"
@.offloading.entry.__omp_offloading_811_d56211f_main_l16 = weak constant %struct.__tgt_offload_entry { ptr @.__omp_offloading_811_d56211f_main_l16.region_id,[…]
```

with USM

```
%struct.ident_t = type { i32, i32, i32, i32, ptr }
%struct.__tgt_offload_entry = type { ptr, ptr, i64, i32, i32 }
%struct.__tgt_kernel_arguments = type { i32, i32, ptr, ptr, ptr, ptr, ptr, ptr, i64, i64, [3 x i32], [3 x i32], i32 }

@0 = private unnamed_addr constant [23 x i8] c";unknown;unknown;0;0;;\00", align 1
@1 = private unnamed_addr constant %struct.ident_t { i32 0, i32 2050, i32 0, i32 22, ptr @0 }, align 8
@2 = private unnamed_addr constant %struct.ident_t { i32 0, i32 514, i32 0, i32 22, ptr @0 }, align 8
@3 = private unnamed_addr constant %struct.ident_t { i32 0, i32 2, i32 0, i32 22, ptr @0 }, align 8
@.__omp_offloading_811_d56211f_main_l16.region_id = weak constant i8 0
@.offload_sizes = private unnamed_addr constant [1 x i64] [i64 40]
@.offload_maptypes = private unnamed_addr constant [1 x i64] [i64 35]
@.offloading.entry_name = internal unnamed_addr constant [38 x i8] c"__omp_offloading_811_d56211f_main_l16\00"
@.offloading.entry.__omp_offloading_811_d56211f_main_l16 = weak constant %struct.__tgt_offload_entry { ptr @.__omp_offloading_811_d56211f_main_l16.region_id,[…]
@.offloading.entry_name.1 = internal unnamed_addr constant [1 x i8] zeroinitializer
@.offloading.entry. = weak constant %struct.__tgt_offload_entry { ptr null, ptr @.offloading.entry_name.1, i64 0, i32 16, i32 8 }, section "omp_offloading_entries", align 1
```

Side-to-side Compiler Code – Target Binary

Compiled with #pragma omp requires unified_shared_memory

```
; Function Attrs: alwaysinline mustprogress norecurse nounwind
define weak_odr protected amdgpu_kernel void @__omp_offloading_811_d56211f_main_l16(
ptr noalias noundef %dyn_ptr,
ptr noundef %vals) local_unnamed_addr #0 {
entry:
  %vals.global1 = addrspacecast ptr %vals to ptr addrspace(1)
  %0 = tail call i32 @__kmpc_get_hardware_thread_id_in_block() #1
  %nvptx_num_threads = tail call i32 @__kmpc_get_hardware_num_threads_in_block() #1
  %gpu_block_id = tail call i32 @llvm.amdgcn.workgroup.id.x()
  %1 = mul i32 %nvptx_num_threads, %gpu_block_id
  %2 = add i32 %1, %0
  %cmp6 = icmp slt i32 %2, 10
  br i1 %cmp6, label %for.body, label %for.cond.cleanup

for.cond.cleanup:                                 ; preds = %for.body, %entry
  ret void

for.body:                                         ; preds = %entry, %for.body
  %.omp.iv.07 = phi i32 [ %6, %for.body ], [ %2, %entry ]
  %idxprom = sext i32 %.omp.iv.07 to i64
  %arrayidx = getelementptr inbounds i32, ptr addrspace(1) %vals.global1, i64 %idxprom
  %3 = load i32, ptr addrspace(1) %arrayidx, align 4, !tbaa !9
  %add1 = add nsw i32 %3, 1
  store i32 %add1, ptr addrspace(1) %arrayidx, align 4, !tbaa !9
  %nvptx_num_threads2 = tail call i32 @__kmpc_get_hardware_num_threads_in_block() #1
  %4 = tail call i32 @__kmpc_get_hardware_num_blocks() #1
  %5 = mul i32 %4, %nvptx_num_threads2
  %6 = add i32 %5, %.omp.iv.07
  %cmp = icmp slt i32 %6, 10
  br i1 %cmp, label %for.body, label %for.cond.cleanup, !llvm.loop !13
```

Compiled without #pragma omp requires unified_shared_memory

```
; Function Attrs: alwaysinline mustprogress norecurse nounwind
define weak_odr protected amdgpu_kernel void @__omp_offloading_811_d56211f_main_l16(
ptr noalias noundef %dyn_ptr,
ptr noundef %vals) local_unnamed_addr #0 {
entry:
  %vals.global1 = addrspacecast ptr %vals to ptr addrspace(1)
  %0 = tail call i32 @__kmpc_get_hardware_thread_id_in_block() #1
  %nvptx_num_threads = tail call i32 @__kmpc_get_hardware_num_threads_in_block() #1
  %gpu_block_id = tail call i32 @llvm.amdgcn.workgroup.id.x()
  %1 = mul i32 %nvptx_num_threads, %gpu_block_id
  %2 = add i32 %1, %0
  %cmp6 = icmp slt i32 %2, 10
  br i1 %cmp6, label %for.body, label %for.cond.cleanup

for.cond.cleanup:                                 ; preds = %for.body, %entry
  ret void

for.body:                                         ; preds = %entry, %for.body
  %.omp.iv.07 = phi i32 [ %6, %for.body ], [ %2, %entry ]
  %idxprom = sext i32 %.omp.iv.07 to i64
  %arrayidx = getelementptr inbounds i32, ptr addrspace(1) %vals.global1, i64 %idxprom
  %3 = load i32, ptr addrspace(1) %arrayidx, align 4, !tbaa !9
  %add1 = add nsw i32 %3, 1
  store i32 %add1, ptr addrspace(1) %arrayidx, align 4, !tbaa !9
  %nvptx_num_threads2 = tail call i32 @__kmpc_get_hardware_num_threads_in_block() #1
  %4 = tail call i32 @__kmpc_get_hardware_num_blocks() #1
  %5 = mul i32 %4, %nvptx_num_threads2
  %6 = add i32 %5, %.omp.iv.07
  %cmp = icmp slt i32 %6, 10
  br i1 %cmp, label %for.body, label %for.cond.cleanup, !llvm.loop !13
```

Side-to-side Compiler Code – Target Binary

Compiled with #pragma omp requires unified_shared_memory

```
; Function Attrs: alwaysinline mustprogress norecurse nounwind
define weak_odr protected amdgpu_kernel void @__omp_offloading_811_d56211f_main_l16(
ptr noalias noundef %dyn_ptr,
ptr noundef %vals) local_unnamed_addr #0 {
entry:
  %vals.global1 = addrspacecast ptr %vals to ptr addrspace(1)
  %0 = tail call i32 @__kmpc_get_hardware_thread_id_in_block() #1
  %nvptx_num_threads = tail call i32 @__kmpc_get_hardware_num_threads_in_block() [illegible]
  %gpu_block_id = tail call i32 @llvm.amdgcn.workgroup.id.x()
  %1 = mul i32 %nvptx_num_threads, %gpu_block_id
  %2 = add i32 %1, %0
  %cmp6 = icmp slt i32 %2, 10
  br i1 %cmp6, label %for.body, label [illegible]

for.cond.cleanup:                                 ; [illegible]
  ret void

for.body:                                         ; preds [illegible]
  %.omp.iv.07 = phi i32 [ %6, %for.body ], [illegible]
  %idxprom = sext i32 %.omp.iv.07 to i64
  %arrayidx = getelementptr inbounds i32, p[illegible]
  %3 = load i32, ptr addrspace(1) %arrayidx, [illegible]
  %add1 = add nsw i32 %3, 1
  store i32 %add1, ptr addrspace(1) %arrayid[illegible] [illegible]aa !9
  %nvptx_num_threads2 = tail call i32 @__km[illegible]_get_hardware_num_threads_in_block() #1
  %4 = tail call i32 @__kmpc_get_hardware_num_blocks() #1
  %5 = mul i32 %4, %nvptx_num_threads2
  %6 = add i32 %5, %.omp.iv.07
  %cmp = icmp slt i32 %6, 10
  br i1 %cmp, label %for.body, label %for.cond.cleanup, !llvm.loop !13
```

Diff between the two LLVM IR snippets:

Compiled without #pragma omp requires unified_shared_memory

```
; Function Attrs: alwaysinline mustprogress norecurse nounwind
define weak_odr protected amdgpu_kernel [illegible]__omp_offloading_811_d56211f_main_l16(
ptr noalias noundef %dyn_ptr,
ptr noundef %vals) loc[illegible]
entry:
  [illegible]space(1)
  [illegible]_in_block() #1
  [illegible]rdware_num_threads_in_block() #1
  [illegible]p.id.x()
  [illegible]
  [illegible]or.body, %entry
  [illegible]
  [illegible]                                     ; preds = %entry, %for.body
  [illegible]mp.iv.07 = phi i32 [ %6, %for.body ], [ %2, %entry ]
  %idxprom = sext i32 %.omp.iv.07 to i64
  %arrayidx = getelementptr inbounds i32, ptr addrspace(1) %vals.global1, i64 %idxprom
  %3 = load i32, ptr addrspace(1) %arrayidx, align 4, !tbaa !9
  %add1 = add nsw i32 %3, 1
  store i32 %add1, ptr addrspace(1) %arrayidx, align 4, !tbaa !9
  %nvptx_num_threads2 = tail call i32 @__kmpc_get_hardware_num_threads_in_block() #1
  %4 = tail call i32 @__kmpc_get_hardware_num_blocks() #1
  %5 = mul i32 %4, %nvptx_num_threads2
  %6 = add i32 %5, %.omp.iv.07
  %cmp = icmp slt i32 %6, 10
  br i1 %cmp, label %for.body, label %for.cond.cleanup, !llvm.loop !13
```

AMD
together we advance_

Difference in runtime behaviour

… without using zero copy

Allocate memory on the device

omptarget --> Looking up mapping(HstPtrBegin=0x000000000238b850, Size=40)...
TARGET AMDGPU RTL --> MemoryManagerTy::allocate: size 40 with host pointer 0x000000000238b850.
TARGET AMDGPU RTL --> findBucket: Size 40 is floored to 32.
TARGET AMDGPU RTL --> Cannot find a node in the FreeLists. Allocate on device.
TARGET AMDGPU RTL --> Node address 0x00000000023af530, target pointer 0x00007fe5a6220000, size 40
omptarget --> Creating new map entry with HstPtrBase=0x000000000238b850, HstPtrBegin=0x000000000238b850, TgtAllocBegin=0x00007fe5a6220000, TgtPtrBegin=0x00007fe5a6220000, Size=40, DynRefCount=1, HoldRefCount=0, Name=unknown
omptarget --> Notifying about new mapping: HstPtr=0x000000000238b850, Size=40

Shipping the data to the device

omptarget --> Moving 40 bytes (hst:0x000000000238b850) -> (tgt:0x00007fe5a6220000)
omptarget --> There are 40 bytes allocated at target address 0x00007fe5a6220000 - is new

Pointer Translation

omptarget --> Looking up mapping(HstPtrBegin=0x000000000238b850, Size=40)...
omptarget --> Mapping exists with HstPtrBegin=0x000000000238b850, TgtPtrBegin=0x00007fe5a6220000, Size=40, DynRefCount=1 […]

omptarget --> Launching target execution __omp_offloading_811_d562480_main_l15 with pointer 0x0000000002371820 (index=0).
PluginInterface --> Launching kernel __omp_offloading_811_d562480_main_l15 with 2 blocks and 8 threads in SPMD-Big-Jump-Loop mode

… using zero copy

omptarget --> Looking up mapping(HstPtrBegin=0x0000000000c60850, Size=40)...
omptarget --> Memory pages for HstPtrBegin 0x0000000000c60850 Size=40 switched to coarse grain
omptarget --> Return HstPtrBegin 0x0000000000c60850 Size=40 for unified shared memory
omptarget --> There are 40 bytes allocated at host address 0x0000000000c60850 - is not new
omptarget --> Looking up mapping(HstPtrBegin=0x0000000000c60850, Size=40)...
omptarget --> Get HstPtrBegin 0x0000000000c60850 Size=40 for unified shared memory
omptarget --> Obtained target argument 0x0000000000c60850 from host pointer 0x0000000000c60850
omptarget --> Launching target execution __omp_offloading_811_d56211f_main_l15 with pointer 0x0000000000c46820 (index=0).
PluginInterface --> Launching kernel __omp_offloading_811_d56211f_main_l15 with 2 blocks and 8 threads in SPMD-Big-Jump-Loop mode

Difference in runtime behaviour

… without using zero copy

Allocate memory on the device

```
omptarget --> Looking up mapping(HstPtrBegin=0x000000000238b850, Size=40)...
TARGET AMDGPU RTL --> MemoryManagerTy::allocate: size 40 with host pointer 0x000000000238b850.
TARGET AMDGPU RTL --> findBucket: Size 40 is floored to 32.
TARGET AMDGPU RTL --> Cannot find a node in the FreeLists. Allocate on device.
TARGET AMDGPU RTL --> Node address 0x00000000023af530, target pointer 0x00007fe5a6220000, size 40
omptarget --> Creating new map entry with HstPtrBase=0x000000000238b850, HstPtrBegin=0x000000000238b850, TgtAllocBegin=0x00007fe5a6220000,
TgtPtrBegin=0x00007fe5a6220000, Size=40, DynRefCount=1, HoldRefCount=0, Name=unknown
omptarget --> Notifying about new mapping: HstPtr=0x000000000238b850, Size=40
```

Shipping the data to the device

```
omptarget --> Moving 40 bytes (hst:0x000000000238b850) -> (tgt:0x00007fe5a6220000)
omptarget --> There are 40 bytes allocated at target address 0x00007fe5a6220000 - is new
```

Pointer Translation

```
omptarget --> Looking up mapping(HstPtrBegin=0x000000000238b850, Size=40)...
omptarget --> Mapping exists with HstPtrBegin=0x000000000238b850, TgtPtrBegin=0x00007fe5a6220000, Size=40, DynRefCount=1 […]
omptarget --> Launching target execution __omp_offloading_811_d562480_main_l15 with pointer 0x0000000002371820 (index=0).
PluginInterface --> Launching kernel __omp_offloading_811_d562480_main_l15 with 2 blocks and 8 threads in SPMD-Big-Jump-Loop mode
```

… using zero copy

```
omptarget --> Looking up mapping(HstPtrBegin=0x0000000000c60850, Size=40)...
omptarget --> Memory pages for HstPtrBegin 0x0000000000c60850 Size=40 switched to coarse grain
omptarget --> Return HstPtrBegin 0x0000000000c60850 Size=40 for unified shared memory
omptarget --> There are 40 bytes allocated at host address 0x0000000000c60850 - is not new
omptarget --> Looking up mapping(HstPtrBegin=0x0000000000c60850, Size=40)...
omptarget --> Get HstPtrBegin 0x0000000000c60850 Size=40 for unified shared memory
omptarget --> Obtained target argument 0x0000000000c60850 from host pointer 0x0000000000c60850
omptarget --> Launching target execution __omp_offloading_811_d56211f_main_l15 with pointer 0x0000000000c46820 (index=0).
PluginInterface --> Launching kernel __omp_offloading_811_d56211f_main_l15 with 2 blocks and 8 threads in SPMD-Big-Jump-Loop mode
```

Host-pointer is returned. No memory allocation or copy operation is necessary.

Dealing with File-scope Variables – non-USM

```
#pragma omp declare target
int x;
#pragma omp end declare target
void foo() {
  int *a = malloc(N*sizeof(int));
  x = 12345;
  #pragma omp target teams loop  \
            map(tofrom:a[0:N]) \
            map(always,to:x)
  for(...) {
    a[i] = i + x;
} }
```

The `map(always,to:x)` is required in non `unified_shared_memory` and copies the content of x on host to x on the device.

Dealing with File-scope Variables – non-USM

```
#pragma omp declare target
int x;
#pragma omp end declare target
void foo() {
  int *a = malloc(N*sizeof(int));
  x = 12345;
  #pragma omp target teams loop  \
              map(tofrom:a[0:N]) \
              map(always,to:x)
  for(...) {
    a[i] = i + x;
} }
```

Host Binary

```
@x = ... global i32 0 ..

store i32 12345, ptr @x
```

Device Binary

```
@x = ...global i32 0 ...

%1 = load i32, ptr addrspace(1) @x
```

The `map(always,to:x)` is required in non `unified_shared_memory` and copies the content of x on host to x on the device.

Dealing with File-scope Variables – non-USM

```
#pragma omp declare target
int x;
#pragma omp end declare target
void foo() {
  int *a = malloc(N*sizeof(int));
  x = 12345;
  #pragma omp target teams loop  \
            map(tofrom:a[0:N]) \
            map(always,to:x)
  for(...) {
    a[i] = i + x;
} }
```

Host Binary

```
@x = ... global i32 0 ..

store i32 12345, ptr @x
```

Device Binary

```
@x = ...global i32 0 ...

%1 = load i32, ptr addrspace(1) @x
```

The `map(always,to:x)` is required in non `unified_shared_memory` and copies the content of x on host to x on the device.

Dealing with File-scope Variables – non-USM

```
#pragma omp declare target
int x;
#pragma omp end declare target
void foo() {
  int *a = malloc(N*sizeof(int));
  x = 12345;
  #pragma omp target teams loop  \
              map(tofrom:a[0:N]) \
              map(always,to:x)
  for(...) {
    a[i] = i + x;
} }
```

Host Binary

```
@x = ... global i32 0 ..

store i32 12345, ptr @x
```

Device Binary

```
@x = ...global i32 0 ...

%1 = load i32, ptr addrspace(1) @x
```

The `map(always,to:x)` is required in non `unified_shared_memory` and copies the content of x on host to x on the device.

Dealing with File-scope Variables – USM

```
#pragma omp requires unified_shared_memory
#pragma omp declare target
int x;
#pragma omp end declare target
void foo() {
  int *a = malloc(N*sizeof(int));
  x = 12345;
  #pragma omp target teams loop  \
              map(tofrom:a[0:N]) \
              map(always,to:x)
  for(...) {
    a[i] = i + x;
} }
```

At code object initialization, the OpenMP runtime writes the address of x in the host binary into `x_ref_ptr` in the device binary. Then, `map(always,to:x)` is a no-op, even if it is present.

Dealing with File-scope Variables – USM

```
#pragma omp requires unified_shared_memory
#pragma omp declare target
int x;
#pragma omp end declare target
void foo() {
  int *a = malloc(N*sizeof(int));
  x = 12345;
  #pragma omp target teams loop  \
            map(tofrom:a[0:N]) \
            map(always,to:x)
  for(...) {
    a[i] = i + x;
} }
```

Host Binary

```
@x = ... global i32 0 ..

store i32 12345, ptr @x
```

Device Binary

```
@x_decl_tgt_ref_ptr = weak global ptr null

%1 = load ptr, ptr @x_decl_tgt_ref_ptr
%2 = load i32, ptr %1, align 4
```

At code object initialization, the OpenMP runtime writes the address of x in the host binary into `x_ref_ptr` in the device binary. Then, `map(always,to:x)` is a no-op, even if it is present.

Dealing with File-scope Variables – USM

```
#pragma omp requires unified_shared_memory
#pragma omp declare target
int x;
#pragma omp end declare target
void foo() {
  int *a = malloc(N*sizeof(int));
  x = 12345;
  #pragma omp target teams loop  \
            map(tofrom:a[0:N]) \
            map(always,to:x)
  for(...) {
    a[i] = i + x;
} }
```

Host Binary

```
@x = ... global i32 0 ..

store i32 12345, ptr @x
```

Device Binary

```
@x_decl_tgt_ref_ptr = weak global ptr null

%1 = load ptr, ptr @x_decl_tgt_ref_ptr
%2 = load i32, ptr %1, align 4
```

At code object initialization, the OpenMP runtime writes the address of x in the host binary into `x_ref_ptr` in the device binary. Then, `map(always,to:x)` is a no-op, even if it is present.

Dealing with File-scope Variables – USM

```
#pragma omp requires unified_shared_memory
#pragma omp declare target
int x;
#pragma omp end declare target
void foo() {
  int *a = malloc(N*sizeof(int));
  x = 12345;
  #pragma omp target teams loop  \
              map(tofrom:a[0:N]) \
              map(always,to:x)
  for(...) {
    a[i] = i + x;
} }
```

Host Binary

```
@x = ... global i32 0 ..

store i32 12345, ptr @x
```

Device Binary

```
@x_decl_tgt_ref_ptr = weak global ptr null

%1 = load ptr, ptr @x_decl_tgt_ref_ptr
%2 = load i32, ptr %1, align 4
```

At code object initialization, the OpenMP runtime writes the address of x in the host binary into `x_ref_ptr` in the device binary. Then, `map(always,to:x)` is a no-op, even if it is present.

AMD
together we advance_

Dealing with File-scope Variables – USM

```
#pragma omp requires unified_shared_memory
#pragma omp declare target
int x;
#pragma omp end declare target
void foo() {
  int *a = malloc(N*sizeof(int));
  x = 12345;
  #pragma omp target teams loop  \
              map(tofrom:a[0:N]) \
              map(always,to:x)
  for(...) {
    a[i] = i + x;
} }
```

Host Binary

```
@x = ... global i32 0 ..

store i32 12345, ptr @x
```

Device Binary

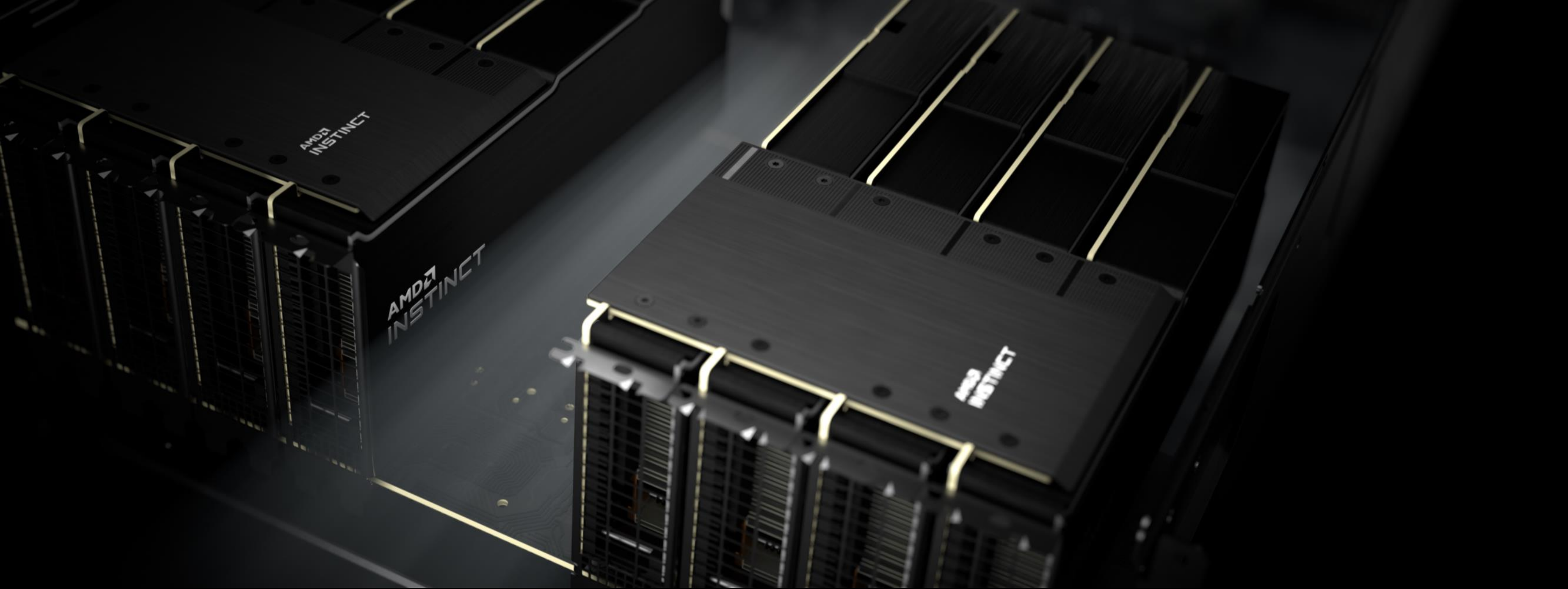
```
@x_decl_tgt_ref_ptr = weak global ptr null

%1 = load ptr, ptr @x_decl_tgt_ref_ptr
%2 = load i32, ptr %1, align 4
```

At code object initialization, the OpenMP runtime writes the address of x in the host binary into `x_ref_ptr` in the device binary. Then, `map(always,to:x)` is a no-op, even if it is present.

AMD Compiler Behavior

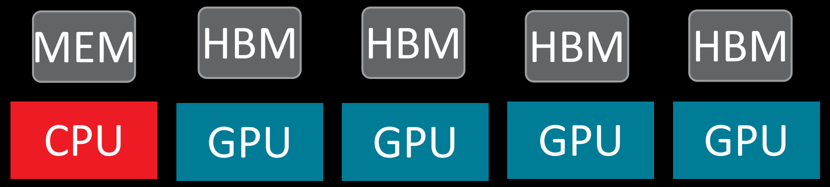
Compiler Flag: gfx942	Default (non-unified_shared_memory)			unified_shared_memory		
	xnack-	xnack-any	xnack+	xnack-	xnack-any	xnack+
XNACK-Enabled	Mismatch	Zero-copy	Zero-copy	Mismatch	Zero-copy	Zero-copy
XNACK-Disabled	Copy	Copy	Mismatch	Zero-copy (RT-Warn)	Zero-copy (RT-Warn)	Mismatch

Mismatch: code object requirement does not match available hardware capability (it's as if you built for a different GPU target)



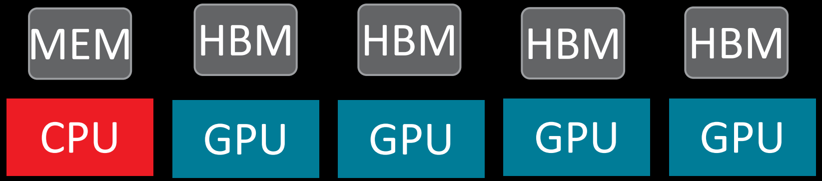
OpenMP® 6.0 Self Maps

OpenMP® 6.0 Self Maps




```
#pragma omp requires self_maps
int main() {
  int *vals = new int[1024];

  #pragma omp target \
          map(“self”,tofrom: vals[0:1024])
  {
    for(int i = 0; i < 1024; i++)
      vals[i] = i;
  }
}
```

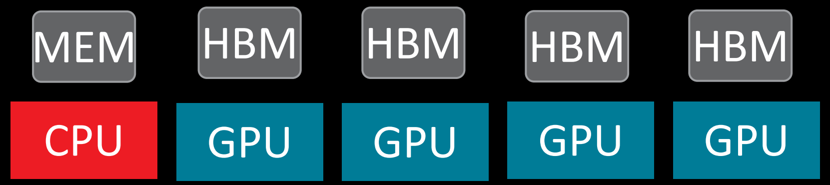
OpenMP® 6.0 Self Maps




```
#pragma omp requires self_maps
int main() {
  int *vals = new int[1024];

  #pragma omp target \
          map("self",tofrom: vals[0:1024])
  {
    for(int i = 0; i < 1024; i++)
      vals[i] = i;
  }
}
```

Compiler/runtime guarantees that no copy was made.

OpenMP® 6.0 Self Maps



USM + “self” for all map( ) clauses

```
#pragma omp requires self_maps
int main() {
  int *vals = new int[1024];

  #pragma omp target \
          map(“self”,tofrom: vals[0:1024])
  {
    for(int i = 0; i < 1024; i++)
      vals[i] = i;
  }
}
```

Compiler/runtime guarantees that no copy was made.

OpenMP® 6.0 Self Maps

```
#pragma omp requires self_maps
int main() {
  int *vals = new int[1024];

  #pragma omp target \
          map("self",tofrom:vals[0:1024])
  {
    for(int i = 0; i < 1024; i++)
      vals[i] = i;
  }
}
```

OpenMP® 6.0 Self Maps

OMPSelfMapsClause(OMPC_self_maps)

```
#pragma omp requires self_maps
int main() {
  int *vals = new int[1024];

  #pragma omp target \
          map("self",tofrom:vals[0:1024])
  {
    for(int i = 0; i < 1024; i++)
      vals[i] = i;
  }
}
```

OpenMP® 6.0 Self Maps

```
#pragma omp requires self_maps
int main() {
  int *vals = new int[1024];

  #pragma omp target \
          map("self",tofrom:vals[0:1024])
  {
    for(int i = 0; i < 1024; i++)
      vals[i] = i;
  }
}
```

OMPSelfMapsClause(OMPC_self_maps)

Record bit OMP_SELF_MAPS=0x020 in OpenMPOffloadingRequiresDirFlags for the current translation unit.

OpenMP® 6.0 Self Maps

OMPSelfMapsClause(OMPC_self_maps)

```
#pragma omp requires self_maps
int main() {
  int *vals = new int[1024];

  #pragma omp target \
          map("self",tofrom:vals[0:1024])
  {
    for(int i = 0; i < 1024; i++)
      vals[i] = i;
  }
}
```

Record bit OMP_SELF_MAPS=0x020 in
OpenMPOffloadingRequiresDirFlags
for the current translation unit.

```
@.offloading.entry_name.1 = internal unnamed_addr constant [1 x i8] zeroinitializer
@.offloading.entry. = weak constant %struct.__tgt_offload_entry
   { ptr null, ptr @.offloading.entry_name.1, i64 0, i32 16, i32 32 },
   section "omp_offloading_entries", align 1
```

Summary

- Zero Copy is a ROCm™ OpenMP® offloading-runtime feature
- Enables execution of OpenMP® programs without explicit data copies*
- Code generation is unaffected
  - OpenMP® program uses explicit map clauses
- Requires hardware/driver support and may not work across all existing devices
- Enabled via environment variable on supported devices

*Except for a specific case of global variables

- Unified Shared Memory is a concept in the OpenMP® standard
- Eliminates the need for data environments via explicit map clauses
- Unified Shared Memory implies code generation that assumes host memory can be accessed
- Requires hardware/driver support and may not work across all existing devices
- Enabled via
  - #pragma omp requires unified_shared_memory
  - -fopenmp-force-usm (future ROCm™ release)

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.

AMD