Deep Copy and Unified Memory in OpenMP
Why both?
Or: If everyone has unified memory why map?

1. Mapping provides more information to both the compiler and the runtime

2. Not all hardware has unified memory

3. Not all unified memory is the same
   1. Can all memory be accessed with the same performance from everywhere?
   2. Do atomics work across the full system?
   3. Are flushes required for coherence? How expensive are they?
Specifying unified memory in OpenMP

• OpenMP does not require unified memory
  • Or even a unified address space
• This is not going to change
How do you make non-portable features portable?

- Specify what they provide when they are present
- Give the user a way to assert that they are required
- Give implementers a way to react to that assertion
One solution: Extension declarations

#pragma omp extension [extension clauses…]

<table>
<thead>
<tr>
<th>Extension name</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>unified_address</td>
<td>Guarantee that device pointers are unique across all devices, is_device_ptr is not required</td>
</tr>
<tr>
<td>unified_memory</td>
<td>Host pointers are valid device pointers and considered present by all implicit maps, implies unified_address, memory is synchronized at target task sync</td>
</tr>
<tr>
<td>system_atomics</td>
<td>OpenMP atomic operations work across devices, may require extra flags</td>
</tr>
<tr>
<td>system_coherence</td>
<td>Combined with unified_memory, explicit flushes and synchronization are not required, even across devices</td>
</tr>
</tbody>
</table>
OpenMP Extension example #1

```c
int * arr = new int[50];
#pragma omp target teams distribute parallel for
for (int i=0; i<50; ++i){
    arr[i] = i;
}
```
OpenMP Extension example #1

```c
int * arr = new int[50];
#pragma omp target teams distribute parallel for
for (int i=0; i<50; ++i){
    arr[i] = i;
}
```
OpenMP Extension example #1

```c
#pragma omp extension unified_memory
int * arr = new int[50];
#pragma omp target teams distribute parallel for
for (int i=0; i<50; ++i){
    arr[i] = i;
}
```
OpenMP Extension example #1

```c
#pragma omp extension unified_memory
int * arr = new int[50];
#pragma omp target teams distribute parallel for
for (int i=0; i<50; i++)
  arr[i] = i;
```
int * arr = omp_target_alloc(sizeof(int)*50, omp_get_default_device());
#pragma omp target teams distribute parallel for
for (int i=0; i<50; ++i){
    arr[i] = i;
}
OpenMP Extension example #2

```c
int * arr = omp_target_alloc(sizeof(int)*50, omp_get_default_device());
#pragma omp target teams distribute parallel for
for (int i=0; i<50; i++)
    arr[i] = i;
```
OpenMP Extension example #2

```c
#pragma omp extension unified_address
int * arr = omp_target_alloc(sizeof(int)*50,
omp_get_default_device());
#pragma omp target teams distribute parallel for
for (int i=0; i<50; ++i){
    arr[i] = i;
}
```
OpenMP Extension example #2

```c
#pragma omp extension unified_address
int * arr = omp_target_alloc((sizeof(int)*50, omp_get_default_device()));
#pragma omp target teams distribute parallel for
for (int i=0; i<50; i++)
    arr[i] = i;
```
Deep copy today

• It is possible to use deep copy in OpenMP today

• Manual deep copy works by *pointer attachment*
typedef struct myvec {
    size_t  len;
    double *data;
} myvec_t;
myvec_t v = init_myvec();
#pragma omp target map(v, v.data[:,:v.len])
{  
do_something_with_v(&v);
}
typedef struct myvec {
   size_t len;
   double *data;
} myvec_t;

myvec_t v = init_myvec();
#pragma omp target map(v, v.data[:v.len])
{
   do_something_with_v(&v);
}
typedef struct myvec {
  size_t len;
  double *data;
} myvec_t;

myvec_t v = init_myvec();
#pragma omp target map(v, v.data[:v.len])
{
  do_something_with_v(&v);
}
What’s the downside?

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

size_t num = 50;
myvec_t *v = alloc_array_of_myvec(num);
#pragma omp target map(v[:50], ??????)
{
    do_something_with_v(&v);
}
typedef struct myvec {
    size_t len;
    double *data;
} myvec_t;

size_t num = 50;
myvec_t *v = alloc_array_of_myvec(num);
#pragma omp target map(v[:50], ????)
{
    do_something_with_v(&v);
}
The future of deep copy: User-defined mappers

```
#pragma omp declare mapper(<type> <var>)
    [name(<name>)]
    [use_by_default]
    [map(<list-items>…)]
```

- Allow users to define mappers in terms of normal map clauses
- Offer extension mechanisms to pack or unpack data that can’t be bitwise copied, or expressed as flat maps
typedef struct myvec {
    size_t len;
    double *data;
} myvec_t;

#pragma omp declare mapper(myvec_t v)
    use_by_default map(v, v.data[:v.len])

size_t num = 50;
myvec_t *v = alloc_array_of_myvec(num);
#pragma omp target map(v[:50])
{
    do_something_with_v(&v);
}
Our array example

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

#pragma omp declare mapper(myvec_t v)
use_by_default map(v, v.data[:v.len])

size_t num = 50;
myvec_t *v = alloc_array_of_myvec(num);
#pragma omp target map(v[:50])
{
    do_something_with_v(&v);
}
Composition of mappers

typedef struct myvec {
    size_t len;
    double *data;
} myvec_t;
#pragma omp declare mapper(myvec_t v)
    use_by_default map(v, v.data[:v.len])
typedef struct mypoints {
    struct myvec *x;
    struct myvec scratch;
    double useless_data[500000];
} mypoints_t;
#pragma omp declare mapper(mypoints_t p) \
    use_by_default \ 
    map( /* self only partially mapped, useless_data can be ignored */\ 
        p.x, p.x[:1]) /* map and attach x */ \ 
    map(alloc:p.scratch) /* never update scratch, including its internal maps */

mypoints_t p = new_mypoints_t();
#pragma omp target
{
    do_something_with_p(&v);
}
Composition of mappers

typedef struct myvec {
    size_t len;
    double *data;
} myvec_t;
#pragma omp declare mapper(myvec_t v)\
    use_by_default map(v, v.data[:v.len])
typedef struct mypoints {
    struct myvec *x;
    struct myvec scratch;
    double useless_data[500000];
} mypoints_t;
#pragma omp declare mapper(mypoints_t p)\
    use_by_default \
    map(/* self only partially mapped, useless_data can be ignored */\
     p.x, p.x[1]) /* map and attach x */ \
    map(alloc:p.scratch) /* never update scratch, including its internal maps */

mypoints_t p = new_mypoints_t();
#pragma omp target
{
    do_something_with_p(&v);
}

Pick and choose what to map
Composition of mappers

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

#pragma omp declare mapper(myvec_t v)
    use_by_default map(v, v.data[:v.len])

typedef struct mypoints {
    struct myvec * x;
    struct myvec scratch;
    double useless_data[500000];
} mypoints_t;

#pragma omp declare mapper(mypoints_t p) \
    use_by_default \ 
    map(/* self only partially mapped, useless_data can be ignored */\ 
       p.x, p.x[:1]) /* map and attach x */ \ 
    map(alloc:p.scratch) /* never update scratch, including its internal maps */

mypoints_t p = new_mypoints_t();
#pragma omp target
{
    do_something_with_p(&v);
}
```

Pick and choose what to map

Re-use the myvec_t mapper
Composition of mappers

typedef struct myvec {
    size_t len;
    double *data;
} myvec_t;
#pragma omp declare mapper(myvec_t v)
    use_by_default map(v, v.data[:v.len])

typedef struct mypoints {
    struct myvec * x;
    struct myvec scratch;
    double useless_data[500000];
} mypoints_t;
#pragma omp declare mapper(mypoints_t p) 
    use_by_default 
    map(/* self only partially mapped, useless_data can be ignored */
        p.x, p.x[1]) /* map and attach x */ 
    map(alloc:p.scratch) /* never update scratch, including its internal maps */

mypoints_t p = new_mypoints_t();
#pragma omp target
{
    do_something_with_p(&v);
}

Re-use the myvec_t mapper

Pick and choose what to map

No explicit map required!
Defining mappers from explicit serialization and deserialization

- Declare mappers by stages, all are replaceable
  - alloc
  - pack_to
  - unpack_to
  - pack_from
  - unpack_from
  - release

- Any arbitrary data can be mapped
Conclusions

• *OpenMP needs* more refined control over memory visibility

• Unified memory is becoming common, but not all unified memory is created equal

• Deep copy with composition can make mapping less painful