OpenMP Offloading

Giacomo Rossi

Intel SCG/SCALE/TCE/XCSS

TREX Hackathon
March 7th 2023

Agenda

- oneAPI
- OpenMP Offload
- oneMKL

oneAPI

Strategy

What is oneAPI?

Open-source software stack

Built with industry standard components
(CLANG, LLVM, SPIR-V)

Intel LLVM | oneAPI Open-Source Projects

An open community

github/oneAPI-TAB

Technical Advisory Boards	Implementations	
SYCL	Intel	
oneMKL	NVidia	AMD
oneDNN	Xilinx	ARM

An open specification

oneAPI
oneAPI.io

Building on other open standards
(SYCL 2020, OpenMP, BLAS, ...)

creative commons

Intel Products

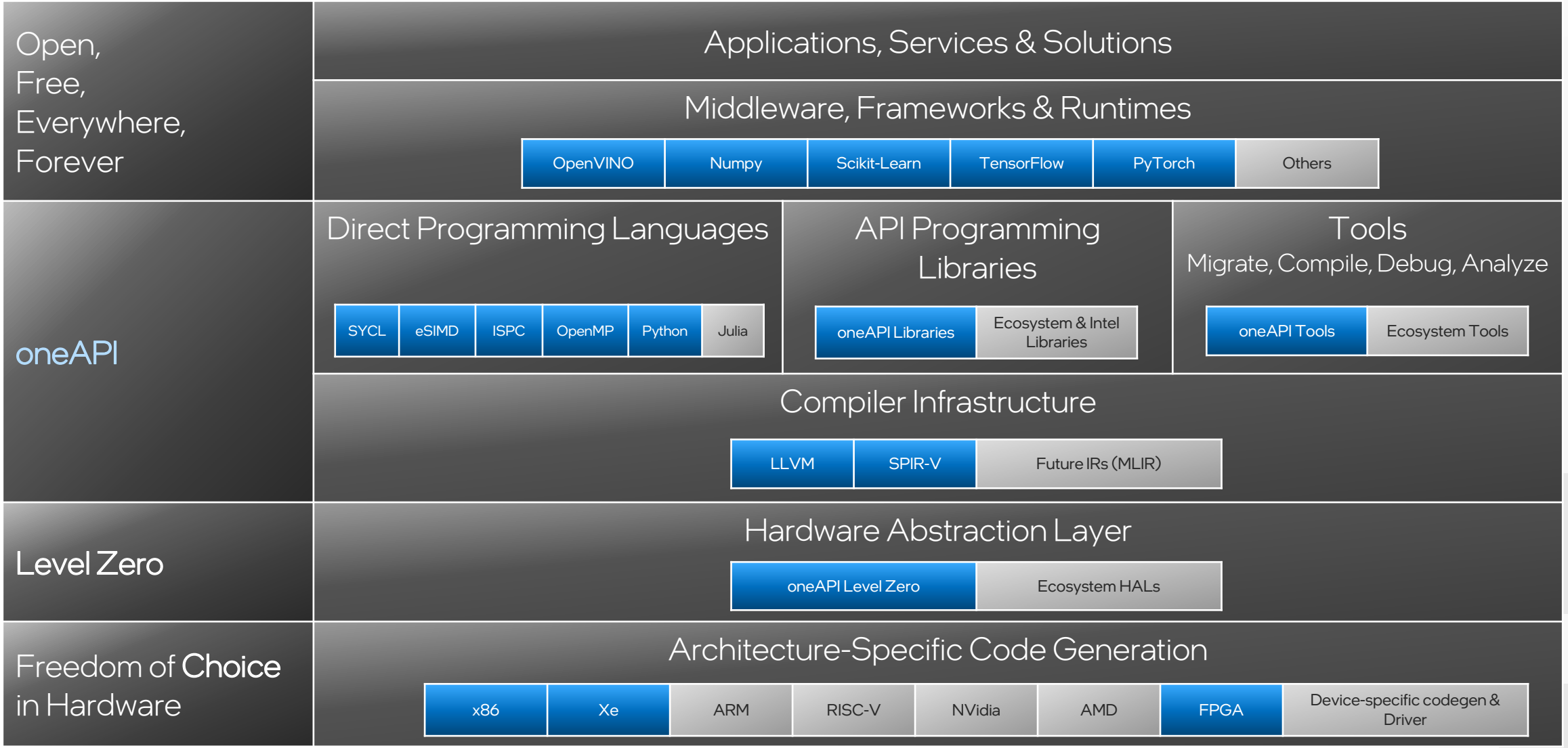
oneAPI
software.intel.com/oneapi

Intel® oneAPI Toolkits
Release 2023.1

Available on

apt | GitHub | CentOS yum/dnf | intel | openSUSE zypper | nuget | CONDA | Maven | Spack

oneAPI: the Big Picture



OpenMP Offload


CSC


OMPcommunity


epcc


FUJITSU


Hewlett Packard
Enterprise


IBM


intel


Los Alamos
NATIONAL LABORATORY


lrz


DSRC
MAUI HIGH PERFORMANCE COMPUTING CENTER
DOD SUPERCOMPUTING RESOURCE CENTER


Micron


NASA


NEC


NVIDIA


OAK
RIDGE
National Laboratory


Pawsey


RWTH AACHEN
UNIVERSITY


Sandia
National
Laboratories


SIEMENS


Stony Brook
University


SUSE
We adapt. You succeed.


TACC


University
of Basel


UNIVERSITY OF
DELAWARE

OpenMP for Accelerators

- Starting from OpenMP specifications 4.0, OpenMP introduced the “target” terminology: the programmer can offload portion of code to device other than the CPU (coprocessors, FPGAs, GPUs…)

- Version 4.5 of OpenMP specifications introduced device memory routines and a couple of constructs in order to control device data mapping

- OpenMP 5.0 specifications extended the device memory routines and improved the device support adding the declare variant construct

- OpenMP 5.1 specifications introduced Fortran interfaces to device memory routines and mainly focused on OpenMP usability on accelerators.

- OpenMP 5.2 is the last version of the specifications available.

Introduction

```
subroutine vec_mult(p, v1, v2, n)
  double precision, intent(inout) :: p(:)
  double precision, intent(in)    :: v1(:), v2(:)
  integer, intent(in)             :: n
  integer                         :: i

  !$omp target teams distribute parallel do simd map(to: v1(1:n), v2(1:n)) map(from: p(1:n))
  do i=1, n
    p(i) = v1(i) * v2(i)
  enddo
endsubroutine
```

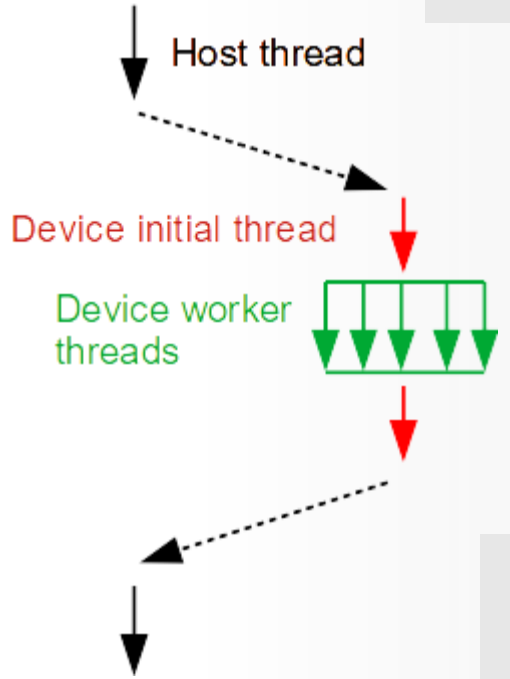
Controls data transfer between host and device

Distributes iterations to the threads and each thread use SIMD parallelism

Identifies the portion of the program that should be run in parallel on the device

Execution model

- OpenMP heterogeneous execution model is host-centric
- The host thread that encounters a target region does not execute the target region; by defaults it waits for the execution of the target region
- A new initial thread is generated on the device
- The **`target teams`** construct starts a league of teams executing in parallel the subsequent code
- When the **`parallel`** construct is encountered by a league, each initial thread becomes the master of a new team of threads
- Each team is a contention group, so are restricted in how they can synchronize with each other



Heterogeneous Memory Model

- OpenMP supports heterogeneous architectures by mapping variables from the host to a device
- The accelerator(s) has a device data environment that contains the set of the variables currently accessible by threads running on that device
- An original variable in host thread's data environment is mapped to a corresponding variable in the accelerator's data environment

Device Execution Control – TARGET TEAMS Construct

`target teams` is a combined construct: it specifies that the subsequent code block should run in parallel

```
!$omp target teams[clause[[,]clause]…]
          structured block
!$omp end target teams
```

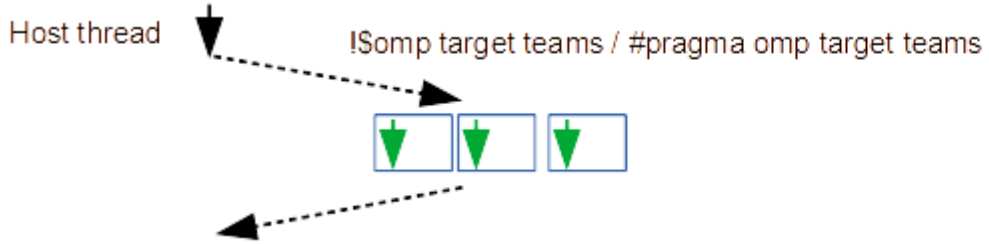
```
#pragma omp target teams[clause[[,]clause]…]
          structured block
```

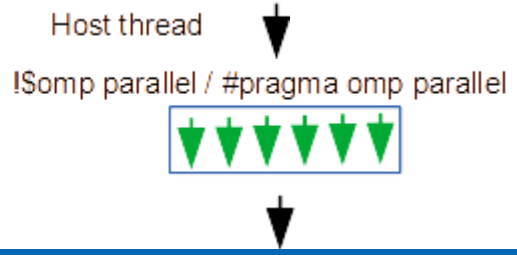
target teams

- This construct starts a league of initial threads where each thread is its own team, and in its own contention group. Each initial thread executes the teams region in parallel
- Threads in different contention group cannot synchronize with each other

parallel

- This construct creates a single team of threads, where each thread in the team executes the parallel region
- Threads can synchronize with each other



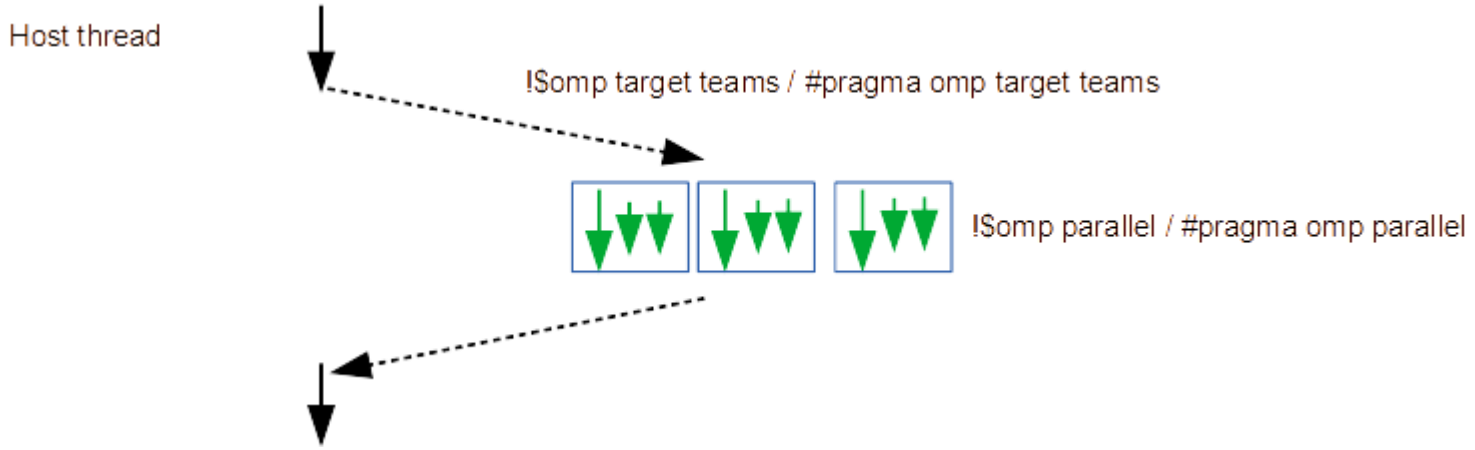


Device Execution Control – TARGET TEAMS Construct

```
!$omp target teams[clause[[,]clause]…]
!$omp parallel
   structured block
!$omp end parallel
!$omp end target teams
```

```
#pragma omp target teams[clause[[,]clause]…]
#pragma omp parallel
          structured block
```

- When a parallel construct is encountered by a league, each thread in the league becomes the master of a new team of threads
- Each team of threads concurrently executes the parallel region



Loop Related Directives: DISTRIBUTE

```
!$omp distribute[clause[[,]clause]…]
   do loops
!$omp end distribute
```

```
#pragma omp distribute[clause[[,]clause]…]
   do loops
```

- Distribute construct has the potential for better performance because of the restrictions on where it can be used and what other OpenMP constructs can appear inside the distribute region
- The do/for construct are more versatile but not perform as well

```
!$omp target teams num_teams(4)
!$omp distribute
do j=1, n, n/2
   !$omp parallel do
   do i=j, j+n/2
      y(i) = x(i)
   enddo
   !$omp end parallel do
Enddo
!$omp end distribute
```

Creates a league of 4 teams

Distributes the loop iterations across the master threads of each team

Activates the threads in the teams and distributes the loop iterations to the threads

Loop Related Directives: LOOP

loop specifies that the logical iterations of the associated loops may execute concurrently

```
!$omp loop[clause[[,]clause]…]
   do loops
!$omp end loop
```

```
#pragma omp loop[clause[[,]clause]…]
   do loops
```

- Loop construct is a worksharing construct if its binding region is the innermost enclosing parallel region
- The directive asserts that the iterations of the associated loops may execute in any order, including concurrently. Each logical iteration is executed once per instance of the **loop** region that is encountered by exactly one thread that is a member of the binding thread set.

```
!$omp target teams num_teams(4)
!$omp loop collapse(2)
do j=1, n
   y(i) = x(i)
!$omp end parallel do
enddo
!$omp end distribute
```

Creates a league of 4 teams

Distributes the loop iterations across the master threads of each team

Activates the threads in the teams and distributes the loop iterations to the threads

Combine and Composite Accelerated Worksharing Constructs

The combined constructs has the same execution behavior of separate constructs, but in some instances, depending on the compiler, the combined constructs may achieve better performance than individual constructs and in such way is possible to distribute loop iterations across multiple levels of parallelism without needing multiple loop nests

```
!$omp target teams distribute parallel do [clause[[,] clause]…]
      do-loops
!$omp end target teams distribute parallel do
```

```
!$omp target teams distribute simd [clause[[,] clause]…]
      do-loops
!$omp end target teams distribute simd
```

```
!$omp target teams distribute parallel do simd [clause[[,] clause]…]
      do-loops
!$omp end target teams distribute parallel do simd
```

```
!$omp target teams loop [clause[[,] clause]…]
      do-loops
!$omp end target teams loop
```

Data Mapping

- The accelerator has its own data environment which contains all set of all variables that are available to the threads executing on that accelerator
- When a host variable is mapped to an accelerator, a corresponding variable is allocated in the accelerator's device data environment
- Host and device variables may share the same storage location → synchronization and memory consistency are required to avoid data races
- Host and device may not share the same storage location → copy operations are required in order to make original and corresponding variable consistent

map Clause

`map([[map-type-modifier[,] [map-type-modifier[,] ...]] map-type: ] locator-list)`

Map type:

- `to`
- `from`
- `tofrom`
- `alloc`
- `release`
- `delete`

Map type modifiers:

- `always`
- `close`
- `mapper`
- `present`
- `iterator`

Map clause on a target constructs:

- `target`
- `target data`
- `target enter data`
- `target exit data`

- Defaults:
- Arrays are `tofrom`
- Scalars are `firstprivate`
- Be careful about pointers: their memory address may not exist on the device!

BE AWARE OF FORTRAN DERIVED TYPES / C STRUCTS!!!

always map modifier

This modifier forces OpenMP runtime to map data even if such data is already present in the device memory space:

- OpenMP runtime performs presence check and if data is present, mapping clause is most likely translated in a target update

```
!$omp target teams
distribute parallel do
map(always,to:a)
```

```
!$omp target update to(a)
!$omp target teams
distribute parallel do
```

```
!$omp target data
map(always,from:a)
...
!$omp end target data
```

```
...
!$omp target update from(a)
```

Mapping in `target` construct

- `map` clause on a target construct:

A. map variables for a single target region

B. enclosed region executes on device and maps data

```
!$omp target map ([[map-type-modifier[,]] map-type:] list)
...
!$omp end target map
```

```
#pragma omp target map ([[map-type-modifier[,]] map-type:] list)
...
```

Incremental porting and data mapping

```
subroutine foo(x)
implicit none

integer, intent(inout) :: x(:)
integer :: n, i

   do i=1, n
      x(i) = x(i) + i
   enddo
   ...
endsubroutine foo
```

Original subroutine, no offload

Modified subroutine with OpenMP offload

```
subroutine foo(x)
implicit none

integer, intent(inout) :: x(:)
integer :: n, i

   !$omp target teams distribute parallel do map(tofrom:x)
   do i=1, n
      x(i) = x(i) + i
   enddo

endsubroutine foo
```

Incremental porting and data mapping (continued)

```
subroutine foo_caller(x,a)
implicit none

integer, intent(inout) :: x(:), a(:)
integer :: n, i

   do i=1, n
      a(i) = x(i) + a(i)
   enddo
   call foo_second(x) !CPU routine that
                      !MODIFIES x ONLY on the
                      !CPU!!!
   call foo(x)
   ...
endsubroutine foo_caller
```

Modified subroutine with OpenMP offload: your program will give wrong results (or will crash)

Original subroutine, no offload: main program works also when subroutine foo is offloaded

```
subroutine foo_caller(x,a)
implicit none

integer, intent(inout) :: x(:), a(:)
integer :: n, i

   !$omp target data map(to:x) map(tofrom:a)

   !$omp target teams distribute parallel do
   do i=1, n
      a(i) = x(i) + a(i)
   enddo

   call foo_second(x) !CPU routine that MODIFIES x
                      !ONLY on the CPU!!!

   call foo(x)

   !$omp end target data
   ...
endsubroutine foo_caller
```

`defaultmap` clause

```
!$omp target defaultmap(implicit-behavior[:variable-category])
...
!$omp end target
```

```
#pragma omp target defaultmap(implicit-behavior[:variable-category])
...
```

- explicitly determines the data-mapping attributes of variables referenced in the region

Implicit behavior
- `alloc`
- `to, from, tofrom`
- `firstprivate`
- `none`
- `default`
- `present`

Variable category
- `scalar`
- `aggregate`
- `pointer`
- `allocatable` (Fortran only)
- `all`

Implicit vs Explicit Mapping

Default map for arrays is `tofrom`

```
double precision :: x(1:n), y(1:n)
integer :: i, n

!$omp target
do i=1, n
  y(i) = i+n
enddo

do i=1, n
  x(i) = y(i)
end do
!$omp end target
```

4 data transfers

```
double precision :: x(1:n), y(1:n)
integer :: i, n

!$omp target map(alloc:y) map(from:x)
do i=1, n
  y(i) = i+n
enddo

do i=1, n
  x(i) = y(i)
end do
!$omp end target
```

1 data transfer

Mapping in `target data` construct

`target data`:

- map variables across multiple target regions in a structured block
- enclosed region does not execute on the device, only maps data from host to device

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

```
integer :: i, n
double precision :: v1(1:n), v2(1:n)
!$omp target map(to: v1, v2) map(from: p)
!$omp parallel do
do i=1, n
  p(i) = v1(i) * v2(i)
enddo
!$omp end parallel do
!$omp end target

do other stuffs on v1 and v2

!$omp target map(to: v1, v2) map(tofrom: p)
!$omp parallel do
do i=1, n
  p(i) = p(i) + v1(i) * v2(i)
enddo
!$omp end parallel do
!$omp end target
```

```
integer :: i, n
double precision :: v1(1:n), v2(1:n)
!$omp target data map (from:p)
!$omp target map(to: v1, v2)
!$omp parallel do
do i=1, n
  p(i) = v1(i) * v2(i)
enddo
!$omp end parallel do
!$omp end target

do other stuffs on v1 and v2

!$omp target map(to: v1, v2)
!$omp parallel do
do i=1, n
  p(i) = p(i) + v1(i) * v2(i)
enddo
!$omp end parallel do
!$omp end target
!$omp end target data
```

`declare target` construct

`declare target`:

- allows global variables to be mapped on an accelerator's device data environment for the whole execution of the program
- - if a function is called from a target region, then the name of the function must appear in a declare target directive

```
!$omp declare target (extended list)
or
!$omp declare target [clause[[,] clause]…]
```

```
#pragma omp declare target (extended list)
or
#pragma omp declare target [clause[[,] clause]…]
```

BE AWARE OF FORTRAN ALLOCATABLES / C POINTERS!!!

declare target construct (II)

```
module my_arrays
!$omp declare target (N, p, v1, v2)
integer, parameter :: N=1000
real :: p(N), v1(N), v2(N)
end module

subroutine vec_mult()
use my_arrays
integer :: I
call init(v1, v2, N);
!$omp target update to(v1, v2)
!$omp target
!$omp parallel do
do i = 1,N
    p(i) = v1(i) * v2(i)
end do
!$omp end target
!$omp target update from (p)
call output(p, N)
end subroutine
```

target enter / target exit data constructs

- map variables in stand alone clauses, not associated with a statement or a structured block

```
!$omp target enter data [clause[[,] clause]…]
or
!$omp target exit data [clause[[,] clause]…]
```

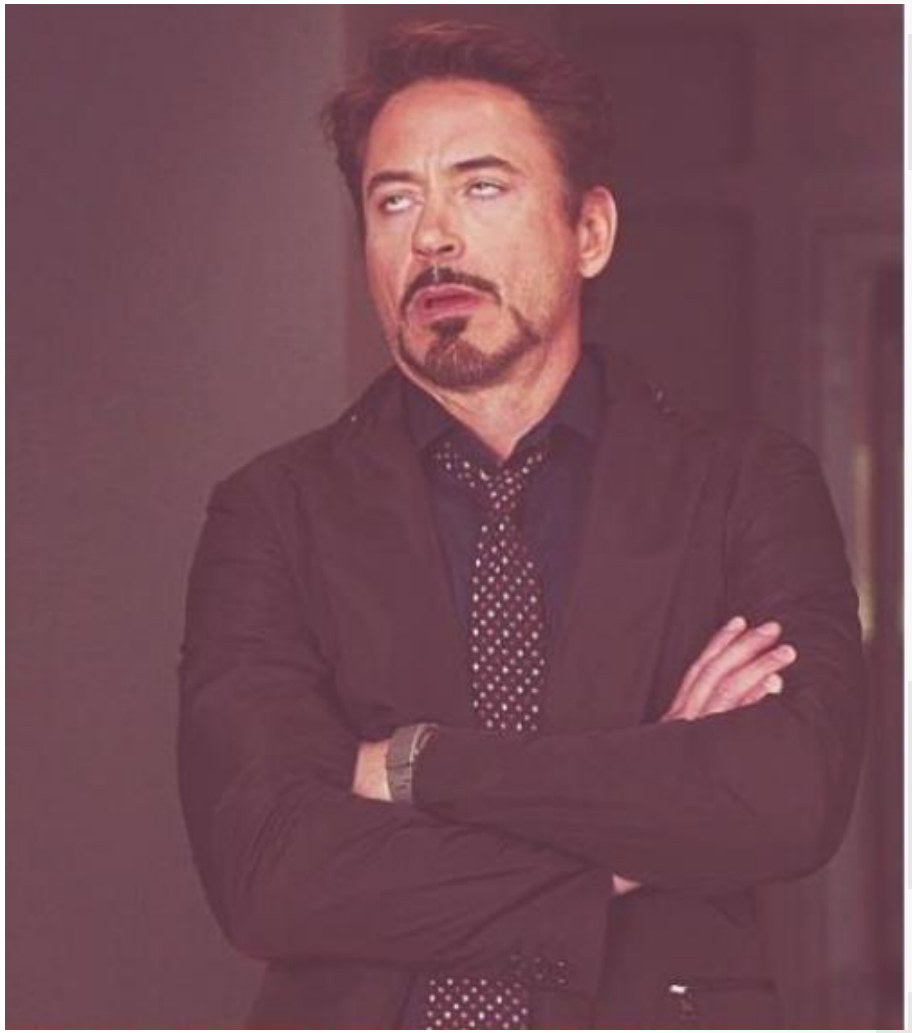
```
module example
real(8), allocatable :: A(:), B(:)
contains

subroutine initialize(N) integer :: N
allocate(A(N)); allocate(B(N))
!$omp target enter data map(alloc:A,B)
end subroutine initialize

subroutine finalize()
!$omp target exit data map(delete:A) map(from:B)
deallocate(A, B)
end subroutine finalize
end module example
```

Avoid Unnecessary Data Transfers

- Don’t rely on implicit mapping! Use map clause to specify when a variable needs to be copied to or from the device.
- Use target data regions around structured blocks to avoid mapping variables unnecessarily
- Use target enter/exit data and target declare data to manage data transfer more explicitly



Data-Sharing Attribute Clauses

- These clauses allow the user to control data-sharing attributes of variables referenced in a construct.
- Concerning offload, four clauses are available:

```
is_device_ptr

use_device_ptr

has_device_addr

use_device_addr
```

is_device_ptr clause

- Can be used in **target** or **dispatch** directives
- It indicates that the list items are device pointers: so each list item is privatized inside the construct and the new list item is initialized to the device address to which the original list item refers.
- In Fortran, each list item should be of type C_PTR

```
...
  arr_host = (int *) malloc(N * sizeof(int));
  arr_device = (int *) omp_target_alloc(N * sizeof(int), omp_get_default_device());

  #pragma omp target is_device_ptr(arr_device) map(from: arr_host[0:N])
  {
    for (int i = 0; i < N; ++i) {
      arr_device[i] = i;
      arr_host[i] = arr_device[i];
    }
  }
...
```

`use_device_ptr` clause

- Can be use in **`target data`** directive
- It indicates that each list item is a pointer to an object that has corresponding storage on the device or is accessible on the device.
- In Fortran, each list item should be of type C_PTR

```
...
  A = (double *) omp_target_alloc(bytes, device_id);
  if (A == NULL){
     printf(" ERROR: Cannot allocate space for A using omp_target_alloc_device().\n");
     exit(1);
  }

  B = (double *) omp_target_alloc(bytes, device_id);
  if (B == NULL){
     printf(" ERROR: Cannot allocate space for B using omp_target_alloc_device().\n");
     exit(1);
  }

  #pragma omp target data use_device_ptr(A,B)
  {
      #pragma omp target teams distribute parallel for
      for (size_t i=0; i<length; i++) {
          A[i] = 2.0;
          B[i] = 2.0;
  }
...
```

has_device_addr clause

- Can be use in **target** directive
- It indicates that the list items already have valid device addresses, and therefore may be directly accessed from the device

```
...
  real(kind=REAL64), allocatable ::  A(:)
  real(kind=REAL64), allocatable ::  B(:)
  !
  ! Allocate arrays in device memory

  !$omp allocate allocator(omp_target_device_mem_alloc) !Intel Extension
  allocate(A(length))

  !$omp allocate allocator(omp_target_device_mem_alloc)
  allocate(B(length))

  !
  ! Initialize the arrays

  !$omp target teams distribute parallel do has_device_addr(A, B)
  do i = 1, length
      A(i) = 2.0
      B(i) = 2.0
  end do
...
```

use_device_addr clause

- Can be use in **target data** directive
- It indicates that the list items already have valid device addresses, and therefore may be directly accessed from the device

```
...
  real(kind=real64), parameter :: aval = real(42, real64)
  real(kind=real64), allocatable :: array_d(:), array_h(:)
  integer :: i,err

  ! Allocate host data
  allocate(array_h(N1))

  !$omp target data map (from:array_h(1:N1)) map(alloc:array_d(1:N1))
  !$omp target data use_device_addr(array_d)
  !$omp target
      do i=1, N1
        array_d(i) = aval
        array_h(i) = array_d(i)
     end do
  !$omp end target
  !$omp end target data
  !$omp end target data
...
```

`declare variant` directive

- Declare base function / routines to have the specified function variant
- The context selector in the `match` clause is associated with the variant

```
! In oneMKL OpenMP offload interface (file mkl_blas_omp_offload_lp64.f90)
module subroutine mkl_blas_dgemm_omp_offload_ilp64 ( transa, transb, m, n, k, alpha, &
   &a, lda, b, ldb, beta, c, ldc ) BIND(C)

   character*1,intent(in) :: transa, transb
   integer,intent(in) :: m, n, k, lda, ldb, ldc
   double precision,intent(in) :: alpha, beta
   double precision,intent(in) :: a( lda, * ), b( ldb, * )
   double precision,intent(inout) :: c( ldc, * )
end subroutine mkl_blas_dgemm_omp_offload_ilp64

subroutine dgemm ( transa, transb, m, n, k, alpha, a, lda, &
   &b, ldb, beta, c, ldc ) BIND(C)

   character*1,intent(in) :: transa, transb
   integer,intent(in) :: m, n, k, lda, ldb, ldc
   double precision,intent(in) :: alpha, beta
   double precision,intent(in) :: a( lda, * ), b( ldb, * )
   double precision,intent(inout) :: c( ldc, * )

   !$omp declare variant( dgemm:mkl_blas_dgemm_omp_offload_ilp64 ) match( construct={dispatch}, device={arch(gen)} ) &
   !$omp& append_args(interop(targetsync)) adjust_args(need_device_ptr:a,b,c)

end subroutine dgemm
```

dispatch construct

- Controls whether variant substitution occurs for target call in the associated function dispatch structured block

```
include 'mkl_omp_offload.f90'
SUBROUTINE laxlib_cdiaghg_gpu( n, m, h, s, ldh, e, v, me_bgrp, root_bgrp,
intra_bgrp_comm )
!------------------------------------------------------------------------- !
USE laxlib_parallel_include
#if defined(MKL_ILP64)
   USE onemkl_lapack_omp_offload_ilp64
#else
   USE onemkl_lapack_omp_offload_lp64
#endif
IMPLICIT NONE
...
   !$omp target data map(to:h, s) map(from:m, e, v, lwork, rwork, ifail, info)
   !$omp dispatch is_device_ptr(h, s, m, e, v, lwork, rwork, ifail, info)
   CALL ZHEGVX( 1, 'V', 'I', 'U', n, h, ldh, s, ldh, &
                0.D0, 0.D0, 1, m, abstol, mm, e, v, ldh, &
                work, lwork, rwork, iwork, ifail, info )
   !$omp end target data
...
END SUBROUTINE laxlib_cdiaghg_gpu
```

Device Memory Routines

OpenMP 4.5 introduced device memory routines for C/C++, to support pointers allocation/deallocation and management in the data environment of target devices:

```
void* omp_target_alloc(size_t size, int device_num);

void omp_target_free(void *device_ptr, int device_num);

int omp_target_is_present(void *ptr, int device_num);

int omp_target_memcpy(void *dst, void *src, size_t length, size_t dst_offset, size_t src_offset, int
dst_device_num, int src_device_num);

int omp_target_memcpy_rect( void *dst, void *src, size_t element_size, int num_dims, const size_t*
volume, const size_t* dst_offsets, const size_t* src_offsets, const size_t* dst_dimensions, const
size_t* src_dimensions, int dst_device_num, int src_device_num);

int omp_target_associate_ptr(void *host_ptr, void *device_ptr, size_t size, size_t device_offset,
int device_num);

int omp_target_disassociate_ptr(void *ptr, int device_num);
```

Device Memory Routines (II)

OpenMP 5.1 added:

```
int omp_target_is_accessible( const void *ptr, size_t size, int
device_num);

int omp_target_memcpy_async( void *dst, const void *src, size_t length,
size_t dst_offset, size_t src_offset, int dst_device_num, int
src_device_num, int depobj_count, omp_depend_t *depobj_list );

int omp_target_memcpy_rect_async( void *dst, const void *src, size_t
element_size, int num_dims, const size_t *volume, const size_t
*dst_offsets, const size_t *src_offsets, const size_t *dst_dimensions,
const size_t *src_dimensions, int dst_device_num, int src_device_num, int
depobj_count, omp_depend_t *depobj_list );
```

OpenMP 5.1 added also Fortran interfaces to all device memory routines

Runtime Routines and Environment Variables

- set the default device:

```
subroutine omp_set_default_device(device_num)
integer :: device_num
```

- get the default device:

```
integer function omp_get_default_device()
```

```
int omp_get_default_device(void);
```

- get the number of target devices:

```
integer function omp_get_num_devices()
```

```
int omp_get_num_devices(void);
```

- find out if we are on the host:

```
logical function omp_is_initial_device()
```

```
int omp_is_initial_device(void);
```

- find out the device number of the host:

```
integer function omp_get_initial_device()
```

```
int omp_get_initial_device(void);
```

Runtime Routines and Environment Variables (II)

- get the total number of teams:

```
integer function omp_get_num_teams()
```

```
int omp_get_num_teams(void);
```

- get the team number:

```
integer function omp_get_team_num()
```

```
int omp_get_team_num(void);
```

OMP_DEFAULT_DEVICE → Set the default device

OMP_TARGET_OFFLOAD={"mandatory" | "disabled" | "default" } →

Intel Offload Runtime Environment Variables

- **`LIBOMPTARGET_PLUGIN=<Name>:`** Designates offload plugin name to use. Offload runtime does not try to load other RTLs if this option is used.

```
<Name> := LEVEL0 | OPENCL | CUDA | X86_64 | NIOS2 |
          level0 | opencl | cuda | x86_64 | nios2
```

- **`LIBOMPTARGET_DEBUG:`** Control whether or not debugging information will be displayed
  - 1 → basic information (device detection, kernel compilation, memory copy operations, kernel invocations, and other plugin-dependent actions)
  - 2 → additionally displays which GPU runtime API functions are invoked with which arguments/parameters
- **`LIBOMPTARGET_INFO:`** Allows the user to request different types of runtime information from **`libomptarget`**
- **`LIBOMPTARGET_PLUGIN_PROFILE=<Enable>[,<Unit>]:`** Enables basic plugin profiling and displays the result when program finishes. Microsecond is the default unit if ``<Unit>`` is not specified.
- **`LIBOMPTARGET_DEVICETYPE=<Type>:`** Decides which device type is used. Only OpenCL plugin supports "CPU" device type.

```
<Type> := GPU | gpu | CPU | cpu
```

LIBOMPTARGET_PLUGIN_PROFILE

```
================================================================================================================================
LIBOMPTARGET_PLUGIN_PROFILE(LEVEL0) for OMP DEVICE(0) Intel(R) Data Center GPU Max 1550, Thread 0
--------------------------------------------------------------------------------------------------------------------------------
Kernel 0 : __omp_offloading_802_202e1e_fft_helper_subroutines_mp_fftx_c2psi_gamma_omp__l1457
Kernel 1 : __omp_offloading_802_202e1e_fft_helper_subroutines_mp_fftx_c2psi_gamma_omp__l1463
Kernel 2 : __omp_offloading_802_202e1e_fft_helper_subroutines_mp_fftx_c2psi_gamma_omp__l1469
Kernel 3 : __omp_offloading_802_202e1e_fft_helper_subroutines_mp_fftx_psi2c_gamma_omp__l1948
Kernel 4 : __omp_offloading_802_202e1e_fft_helper_subroutines_mp_fftx_psi2c_gamma_omp__l1956
Kernel 5 : __omp_offloading_802_262342_qe_drivers_lda_lsda_mp_xc_lda__l95
Kernel 6 : __omp_offloading_802_822655_v_xc__l1474
Kernel 7 : __omp_offloading_802_822655_v_xc__l1488
Kernel 8 : __omp_offloading_802_885338_vloc_psi_gamma__l120
Kernel 9 : __omp_offloading_802_885338_vloc_psi_gamma__l134
--------------------------------------------------------------------------------------------------------------------------------
: Host Time (msec) Device Time (msec)
Name : Total Average Min Max Total Average Min Max Count
--------------------------------------------------------------------------------------------------------------------------------
Compiling : 193.28 193.28 193.28 193.28 0.00 0.00 0.00 0.00 1.00
DataAlloc : 7.12 0.00 0.00 2.16 0.00 0.00 0.00 0.00 3453.00
DataRead (Device to Host) : 42.46 0.02 0.01 0.30 3.03 0.00 0.00 0.03 2772.00
DataWrite (Host to Device): 75.48 0.02 0.01 0.52 8.62 0.00 0.00 0.05 4164.00
Kernel 0 : 2.50 0.04 0.02 0.21 0.71 0.01 0.01 0.08 56.00
Kernel 1 : 1.20 0.03 0.03 0.07 0.40 0.01 0.01 0.01 40.00
Kernel 2 : 0.54 0.03 0.03 0.09 0.18 0.01 0.01 0.01 16.00
Kernel 3 : 1.26 0.03 0.03 0.08 0.44 0.01 0.01 0.03 40.00
Kernel 4 : 0.51 0.03 0.02 0.08 0.15 0.01 0.01 0.03 16.00
Kernel 5 : 2.09 0.42 0.05 1.86 0.12 0.02 0.02 0.02 5.00
Kernel 6 : 4.00 0.80 0.11 3.55 0.21 0.04 0.03 0.06 5.00
Kernel 7 : 1.01 0.20 0.16 0.36 0.65 0.13 0.13 0.13 5.00
Kernel 8 : 1.77 0.03 0.02 0.10 0.50 0.01 0.01 0.01 56.00
Kernel 9 : 1.49 0.03 0.02 0.06 0.44 0.01 0.01 0.01 56.00
Linking : 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 1.00
OffloadEntriesInit : 8.96 8.96 8.96 8.96 0.00 0.00 0.00 0.00 1.00
================================================================================================================================
```

LIBOMPTARGET_DEBUG

```
Libomptarget --> Launching target execution __omp_offloading_802_262342_qe_drivers_lda_lsda_mp_xc_lda__l95 with pointer 0x000000000bf358a0 (index=38).
Target LEVEL0 RTL --> Executing a kernel 0x000000000bf358a0...
Target LEVEL0 RTL --> Assumed kernel SIMD width is 32
Target LEVEL0 RTL --> Preferred team size is multiple of 64
Target LEVEL0 RTL --> Loop 0: lower bound = 0, upper bound = 91124, Stride = 1
Target LEVEL0 RTL --> Team sizes = {64, 1, 1}
Target LEVEL0 RTL --> Number of teams = {1424, 1, 1}
Target LEVEL0 RTL --> Kernel Pointer argument 0 (value: 0xff00ffffff400000) was set successfully for device 0.
Target LEVEL0 RTL --> Kernel Scalar argument 7 (value: 0x00000000000163f5) was set successfully for device 0.
...
Target LEVEL0 RTL --> Kernel Scalar argument 9 (value: 0x00000000000163f5) was set successfully for device 0.
Target LEVEL0 RTL --> Kernel Scalar argument 26 (value: 0x00000000000163f5) was set successfully for device 0.
Target LEVEL0 RTL --> Kernel Scalar argument 27 (value: 0x00000000000163f5) was set successfully for device 0.
Target LEVEL0 RTL --> Submitted kernel 0x000000000c4fc420 to device 0
Target LEVEL0 RTL --> Executed kernel entry 0x000000000bf358a0 on device 0
...
Libomptarget --> Looking up mapping(HstPtrBegin=0x0000153919c6e900, Size=729000)...
Libomptarget --> Mapping exists with HstPtrBegin=0x0000153919c6e900, TgtPtrBegin=0xff00ffffff400000, Size=729000, DynRefCount=1 (decremented), HoldRefCount=0
Libomptarget --> There are 729000 bytes allocated at target address 0xff00ffffff400000 - is not last
Libomptarget --> Entering target region with entry point 0x0000000000ef14bc and device Id 0
Libomptarget --> Call to omp_get_num_devices returning 1
Libomptarget --> Call to omp_get_num_devices returning 1
Libomptarget --> Call to omp_get_initial_device returning 1
Libomptarget --> Checking whether device 0 is ready.
Libomptarget --> Is the device 0 (local ID 0) initialized? 1
Libomptarget --> Device 0 is ready to use.
Libomptarget --> Entry 0: Base=0x0000000007f790b0, Begin=0x0000000007f790b0, Size=8, Type=0x223, Name=v_xc_$ETXC
...
Libomptarget --> Entry 6: Base=0x00007ffdbac90900, Begin=0x00007ffdbac90908, Size=88, Type=0x5000000000001, Name=v_xc_$VX_dv_len
...
Libomptarget --> Entry 54: Base=0x0000000000000000, Begin=0x0000000000000000, Size=0, Type=0x120, Name=unknown
Libomptarget --> Looking up mapping(HstPtrBegin=0x0000000007f790b0, Size=8)...
Target LEVEL0 RTL --> Ptr 0x0000000007f790b0 requires mapping
Libomptarget --> Creating new map entry with HstPtrBegin=0x0000000007f790b0, TgtPtrBegin=0xff00ffffffffa0800, Size=8, DynRefCount=1, HoldRefCount=0, Name=v_xc_$ETXC
Libomptarget --> Moving 8 bytes (hst:0x0000000007f790b0) -> (tgt:0xff00ffffffffa0800)
Target LEVEL0 RTL --> Copied 8 bytes (hst:0x0000000007f790b0) -> (tgt:0xff00ffffffffa0800)
Libomptarget --> There are 8 bytes allocated at target address 0xff00ffffffffa0800 - is new
Libomptarget --> Looking up mapping(HstPtrBegin=0x0000000007f7aed0, Size=8)...
Libomptarget --> Moving 8 bytes (hst:0x0000000007f7aed0) -> (tgt:0xff00ffffffffa02c0)
Target LEVEL0 RTL --> Copied 8 bytes (hst:0x0000000007f7aed0) -> (tgt:0xff00ffffffffa02c0)
Libomptarget --> Mapping exists (implicit) with HstPtrBegin=0x0000153919d22980, TgtPtrBegin=0xff00ffffff700000, Size=729000, DynRefCount=2 (incremented), HoldRefCount=0, Name=v_xc_$V
Libomptarget --> There are 729000 bytes allocated at target address 0xff00ffffff700000 - is not new
Libomptarget --> Looking up mapping(HstPtrBegin=0x00007ffdbac90900, Size=96)...
Libomptarget --> Mapping exists with HstPtrBegin=0x00007ffdbac90900, TgtPtrBegin=0xff00ffffffff80500, Size=96, DynRefCount=2 (incremented), HoldRefCount=0, Name=v_xc_$VX
Libomptarget --> There are 96 bytes allocated at target address 0xff00ffffffff80500 - is not new
```

OpenMP 5.x: *standard ready to compete with OpenACC*

Hierarchical parallelism

```
#pragma omp target teams distribute
for (...) {
    #pragma omp parallel for
    for ( ...) {
        for (...) {
            #pragma omp simd
            for (...) {
                for (...) {

                }
            }
        }
    }
}
```

Unified Shared Memory

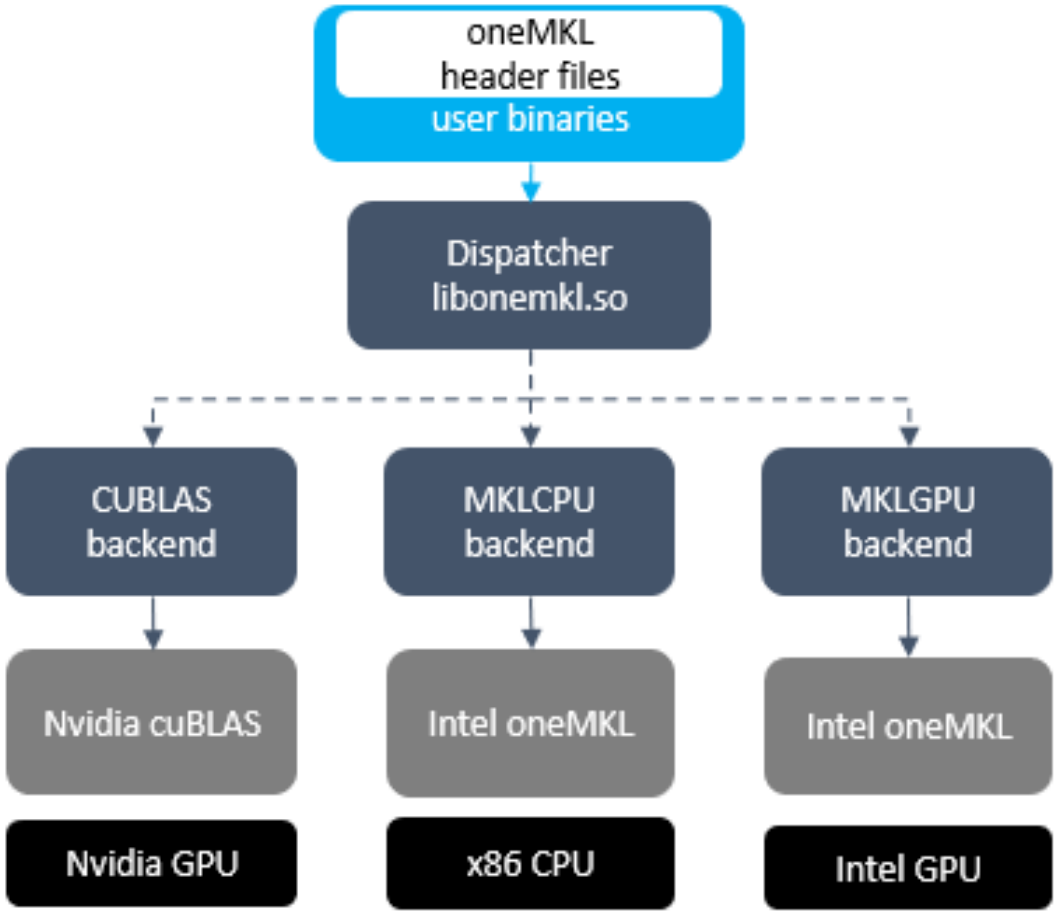
```
#pragma omp requires unified_address
 A = omp_target_alloc_shared(...);
```

Or explicit control of data movement

```
int *arr_host = malloc(...);
int *arr_device = omp_target_alloc_device(...);
#pragma omp target is_device_ptr(arr_device)
#pragma omp target map(tofrom: arr_host[0:N])
```

OneMKL

oneMKL defines common interface to multiple targets




```
// C := alpha*(AxB)+ beta*C

gemm(q,
     transA, transB,
     m, n, k, alpha,
     A, ldA, B, ldB,
     beta, C, ldC);
```

GEMM is called from the host

Call is routed to appropriate library by the queue

Dispatch can be dynamic or determined statically

Thank You


intel