OpenMP* GPU Offload Basics

Soner Steiner

Intel oneAPI Certified Instructor

VSC

December 2023

intel®

Intel oneAPI HPC Workshop- Agenda
Online 19th-20th September 2023

DAY 2 – THEME: PROGRAMMING WITH ONEAPI

Time	Session name / description	Presenter
09:30	Welcome	*Soner/Claudia*
09:40	Porting CUDA code to SYCL using the Compatibility Tool	*Georg*
10:30	HANDS-ON WITH COMPATIBILITY TOOL A hands-on lab session where you can try porting a CUDA code to oneAPI with the help of the Compatibility Tool	*Georg*
11:00	Coffee	
11:15	OFFLOADING WITH C/C++ and FORTRAN Offloading using OpenMP mainly in C/C++ Offloading using OpenMP in FORTRAN Automatic offloading using DO CONCURRENT	*Soner*
12:00	Lunch	
13:00	LAB3: HANDS-ON OFFLOADING WITH OPENMP	*Soner*
14:30	Coffee	
14:45	LAB4: HANDS-ON VTUNE A hands-on lab session where you can use the Vtune and Advisor profilers to assess the performance of some example codes.	*Soner*
16:00	End of Day 2	

Agenda

- oneAPI and OpenMP* Offload
- OpenMP on CPUs Review
- Introduction to OpenMP Offload
- Constructs to Manage Device Data
- Constructs to Leverage Parallelism
- Summary
- Calling oneMKL OpenMP Offload functions

oneAPI and OpenMP* Offload

*Other names and brands may be claimed as the property of others.

Programming Challenges

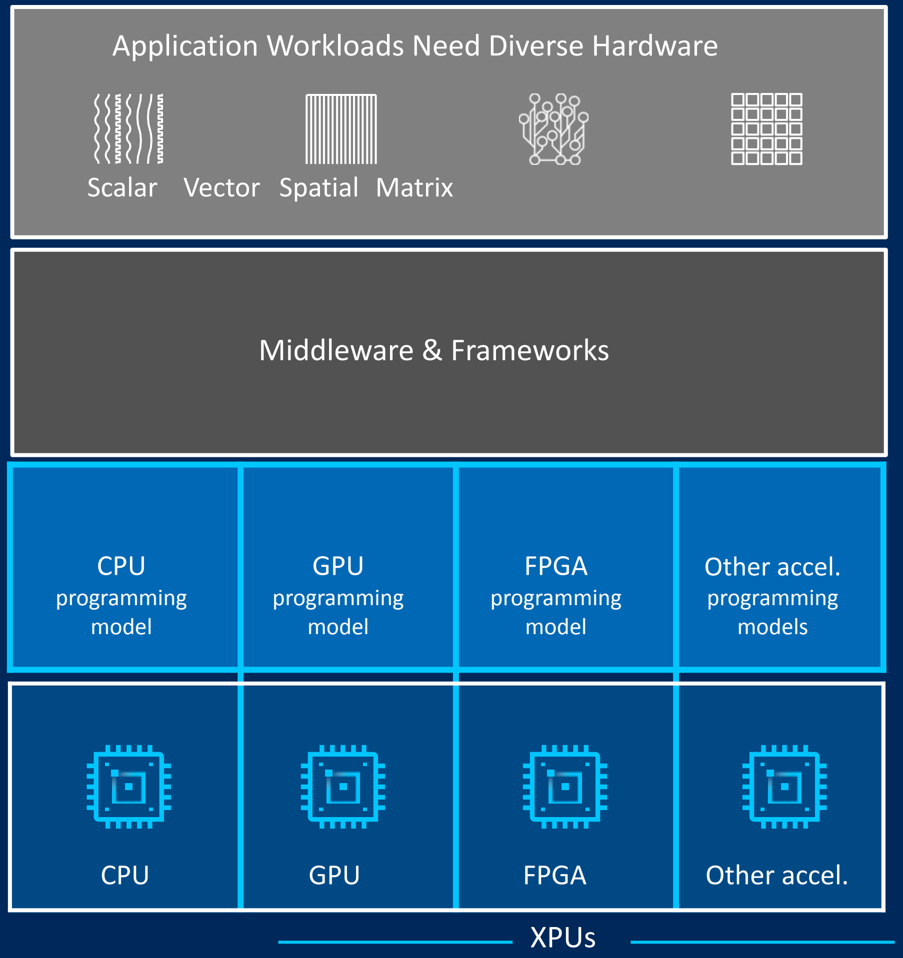
for Multiple Architectures

Growth in specialized workloads

Variety of data-centric hardware required

Separate programming models and toolchains for each architecture are required today

Software development complexity limits freedom of architectural choice



OpenMP* on CPUs

OpenMP* Overview

- Cross-platform standard supporting shared-memory-multi-processing programming in C, C++ and Fortran
  - API for writing multithreaded applications
  - Set of compiler directives and library routines for parallel application programmers
  - Greatly simplifies writing multi-threaded programs in Fortran, C and C++
  - Portable across vendors and platforms
  - Supports various types of parallelism

OpenMP* History

- 1997: Version 1.0 for Fortran
- 1998: Version 1.0 for C/C++
- 2002-2005: Versions 2.0-2.5, Merger of Fortran and C/C++ specifications
- 2008: Version 3.0, Incorporates Task Parallelism
- 2013: Version 4.0, Support for Accelerators, SIMD support
- 2018: Version 5.0, C11/C++17/Fortran 2008 support

OpenMP* Threads

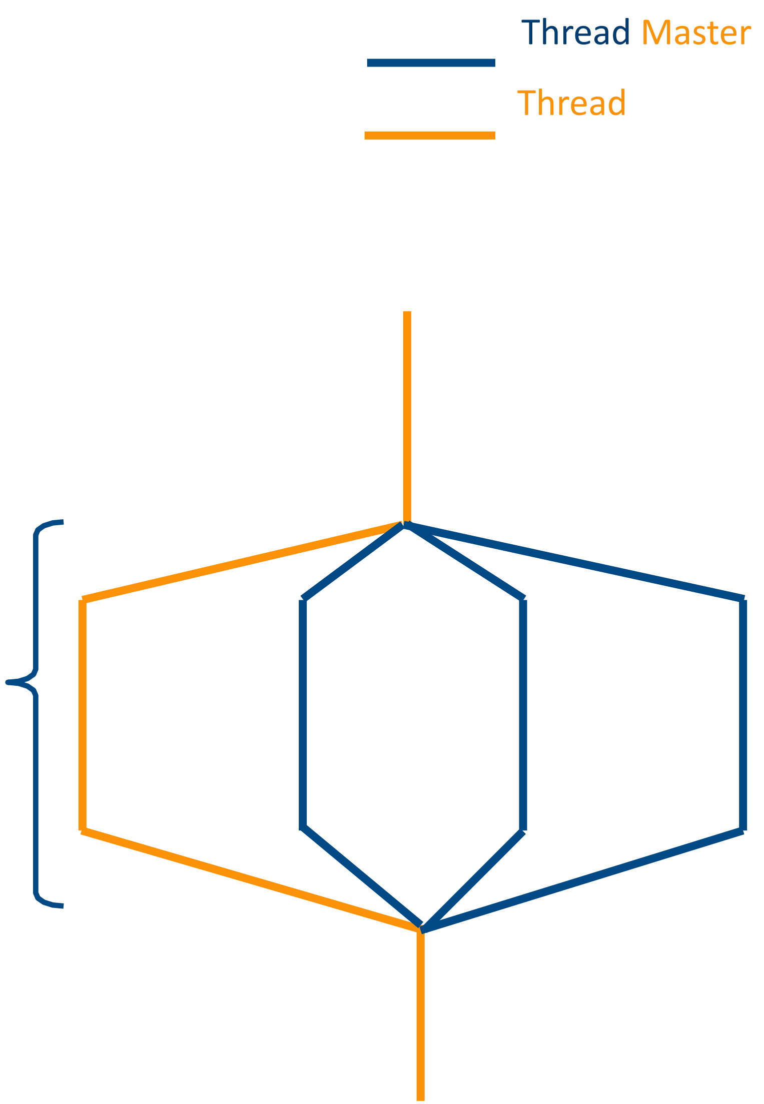
- Create threads with the **`parallel`** construct

```
subroutine saxpy(a, x, y,
    n) use iso_fortran_env
    integer      :: n, i
    real(real32) :: a, x(n), y(n)

!$omp parallel
    do i=1,n
        y(i) = a * x(i) +
    y(i) end do
!$omp end parallel

end subroutine
```

Parallel Region.

Team of threads created.

Each thread executes the same code redundantly



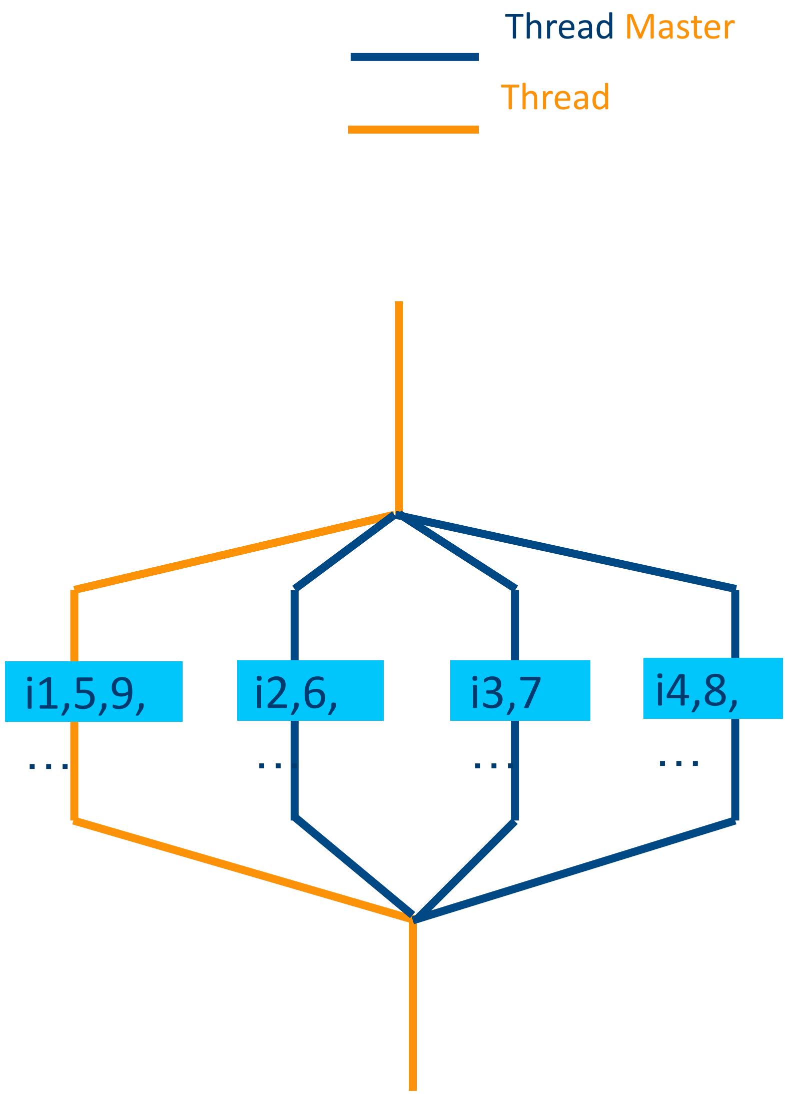
Loops

- Use For/Do Loop Directive to Workshare

```
subroutine saxpy(a, x, y,
    n) use iso_fortran_env
    integer       :: n, i
    real(real32) :: a, x(n), y(n)

!$omp parallel
    !$omp do
    do i=1,n
        y(i) = a * x(i) +
    y(i) end do
!$omp end parallel

end subroutine
```

Workshare:

Distributes the execution of loop iterations across the threads



Basic Examples

C/C++

```
#include <omp.h>


…
#pragma omp parallel for reduction (+:sum)
{
   for (int i=0; i<ARRAY_SZ; i++) {
      sum += x[i];
   }
}
…
```

Fortran

```
program main
…
   !$omp parallel do reduction (+:total)
   do i=1,ARRAY_SZ
      total = total + x(i)
   end do
   !$omp end parallel do
…
end program main
```

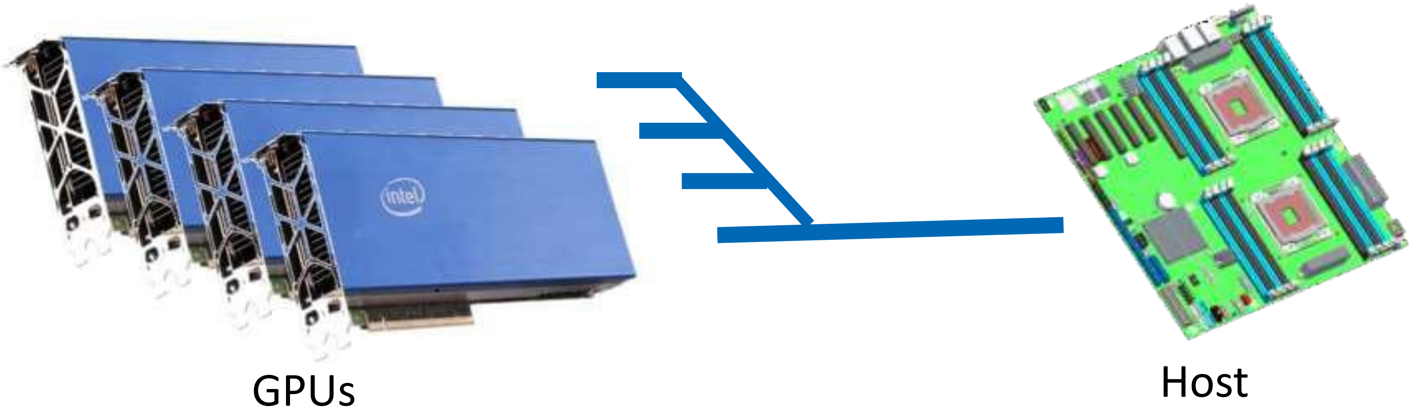
Other Notable OpenMP* Constructs

- Sections/Section
  - Distribute blocks of code (sections) among existing threads
- Task
  - Create independent units of work (including code, data, and internal control variables) for execution on a thread
- SIMD
  - Specifies iterations of a given loop can be executed concurrently with SIMD instructions
    - i.e. compiler can ignore vector dependencies

Introduction: OpenMP* Offload

OpenMP* Device Model

- OpenMP 4.0+ supports accelerators/coprocessors (devices)
  - Not GPU-specific
- Device model:
  - One host
  - Multiple accelerators/coprocessors of the same kind



OpenMP* Offload Compiler Support

- OpenMP Offload Supported in the Intel® oneAPI HPC Toolkit
  - Need to enable OpenMP* => 4.5 support (-fiopenmp) and OpenMP* => 4.5 offloading support (-fopenmp-targets=spir64)
  - Intel® oneAPI C++ Compiler

```
icx –fiopenmp –fopenmp-version=51 –fopenmp-
targets=spir64 <source>.c
icpx –fiopenmp –fopenmp-version=51 –fopenmp-
targets=spir64 <source>.cpp
```

  - Intel® Fortran Compiler

```
ifx –fiopenmp –fopenmp-targets=spir64 <source>.f90
```

OpenMP* 4.0 for Devices - Constructs

- target construct transfer control and data from the host to the device
- Syntax (C/C++)

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

- Syntax (Fortran)

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

- Clauses

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

Execution Model

- The `target` construct transfers the control flow to the target device
  - Transfer of control is sequential and synchronous
  - The map clause controls direction of data flow
  - Array notation is used to describe array length

Target Region Example: saxpy

Sequential Host Code

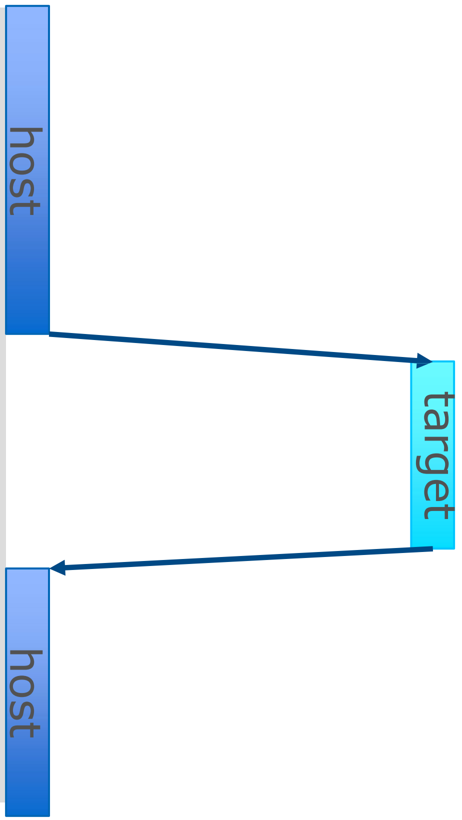
```
subroutine saxpy(a, x, y,
    n) use iso_fortran_env
    integer :: n, i
    real(real32) :: a, x(n), y(n)
```

Target Region

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

Sequential Host Code

```
end subroutine
```




```
ifx -c -fiopenmp -fopenmp-targets=spir64 -o saxpy saxpy.f90
```

Device Clause

- Specify which device to offload to in a multi-device environment

```
!$omp target device(i)
```

  - Device number an integer
    - Assignment is implementation-specific
    - Usually start at 0 and sequentially increments
  - Works with **`target, target data, target enter/exit data, target update`** directives

Calling Functions Inside Target Area

- **`declare target`** construct compiles a version of the function/subroutine for the target device
  - Function compiled for both host execution and target execution by default

```
#pragma omp declare target
int devicefunc(){
…
}
#pragma omp end declare target

#pragma omp target
{
         result = devicefunc();
}
```

```
subroutine devicefunc()
!$omp declare target device_type(device)
…
end subroutine

program main
!$omp target
         call devicefunc()
!$omp end target
end program
```

Optional device_type specifies host and/or device execution

if device is specified, it needs to be always available

Select Target Device with Environment Variable

- Use OMP_TARGET_OFFLOAD to specify where the target region code should run.
  - Useful for debugging
  - OMP_TARGET_OFFLOAD={“MANDATORY” | “DISABLED” | “DEFAULT”}

Type	Description
MANDATORY	The target region code runs on GPU or other accelerator.
DISABLED	The target region code runs on CPU.
DEFAULT	The target region code runs on a GPU if the device is available, will fall back to the CPU

Asynchronous Offloading

- OpenMP target constructs are synchronous by default
  - Host thread awaits the end of the target region before continuing
- The `nowait` clause makes the target constructs asynchronous
  - Target region becomes an OpenMP task (use task synchronization)

```
 !$omp task              depend(out:in1)
           init_data(in1);

   !$omp target map(to:in1) map(from:out1)           depend(in:in1)
                  nowait                             depend(out:out1)
          compute_1(in1, out1, N);
   !$omp target map(to:in2) map(from:out2)           depend(out:out2)
                  nowait
          compute_2(in2, out2, N);
   !$omp target map(to:out1) map(to:out2)            depend(in:out1)
                 nowait                              depend(in:out2)
          compute_3(out1, out2, N);
!$omp taskwait
```

Managing Device Data

Offload Data

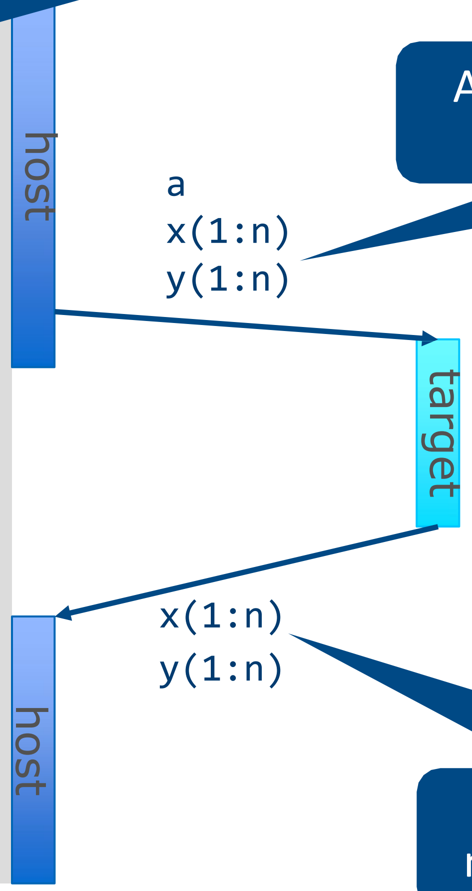
- Host and devices have separate memory spaces
  - Data needs to be mapped to the target device in order to be accessed inside the target region
  - Default for variables accessed inside the target region:
    - Scalars: treated as firstprivate
    - Static arrays: copied to and from the device on entry and exit
  - Data environment is lexically scoped
    - Data environment is destroyed at closing curly brace
    - Allocated buffers/data are automatically released

Example: saxpy

The compiler identifies variables that are used in the `target` region.

```
subroutine saxpy(a, x, y, n)
    use iso_fortran_env
    use omp_lib
    integer :: n, i real(real32)
    :: a, x(n), y(n)
    real(real64) :: t, tb, te
t=0._real64
tb=omp_get_wtime()
!$omp target
    do i=1,n
       y(i) = a * x(i) +
    y(i) end do
!$omp end target
te=omp_get_wtime(
) t=tb-te
write(*,*) "Time of kernel:",
t end subroutine
```

```
ifx -fiopenmp -fopenmp-targets=spir64 -c -o saxpy
saxpy.f90
```



All accessed arrays are copied from host to device and back

Copying x back is not necessary: it was not changed.

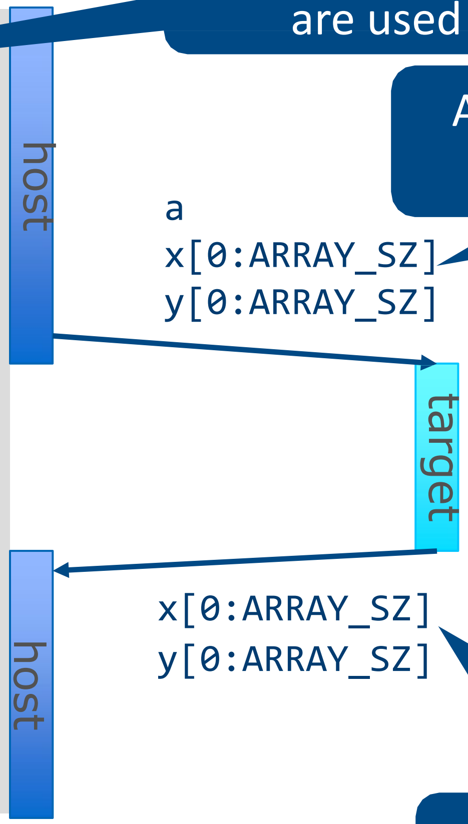
Example: saxpy

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

```
icx -fiopenmp -fopenmp-targets=spir64 -o saxpy saxpy.c
```

The compiler identifies variables that are used in the `target` region.

All accessed arrays are copied from host to device and back



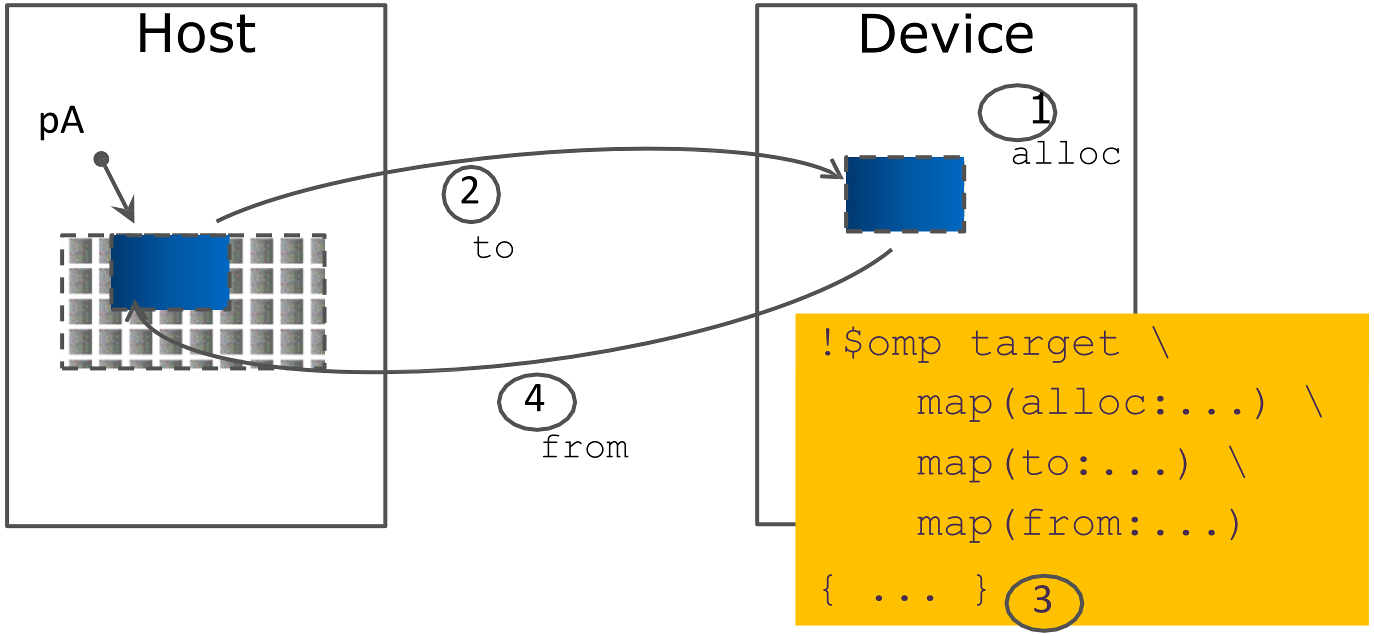
Copying x back is not necessary: it was not changed.

Map Clause

- Use **`map`** clause to manually determine how an original variable in a data environment is mapped to a corresponding variable in a device data environment
  - `omp target map (`*`map-type: list`*`)`
  - Available map-type
    - alloc : allocate storage for variable on target device (values not copied)
    - to : alloc and assign value of original variable on target region entry
    - from : alloc and assign value to original variable on target region exit
    - tofrom: default, both to and from

Map Clause

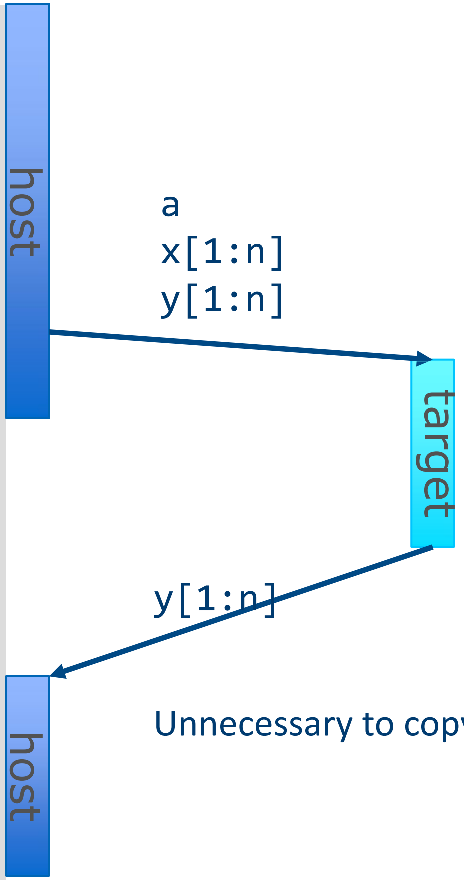
- Use **`map`** clause to manually determine how an original variable in a data environment is mapped to a corresponding variable in a device data environment



Example: saxpy

```
subroutine saxpy(a, x, y,
n)
    use iso_fortran_env
    integer       :: n, i
    real(real32) :: a, x(n), y(n)
    real(real64) :: t, tb, te
t=0._real64
tb=omp_get_wtime()
!$omp target map(to:x)
    map(tofrom:y) do i=1,n
        y(i) = a * x(i) +
    y(i) end do
!$omp end target
te=omp_get_wtime(
) t=tb-te
write(*,*) "Time of kernel:",
t end subroutine
```

```
ifx -fiopenmp -fopenmp-targets=spir64 -o saxpy saxpy.f90
```



Unnecessary to copy x back to the host

Mapping Dynamically Allocated Data

- When pointers are dynamically allocated, number of elements to be mapped must be explicitly specified

```
#pragma omp target map(to:array[start:length])
!$omp target map(to:array(start:end))
```

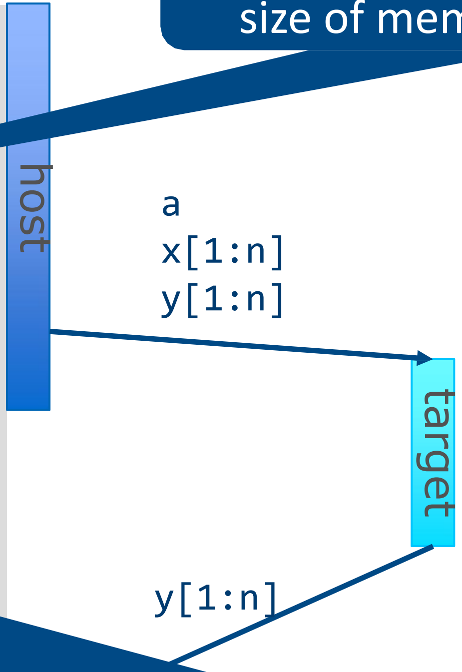
  - Partial array may be specified
  - Note: syntax in C/C++ (uses *length*) is different from Fortran (uses *end*)

Example: saxpy

```
subroutine saxpy(a, x, y, n)
    use iso_fortran_env
    use omp_lib
    integer :: n, i real(real32)
    :: a, x(*), y(*)
    real(real64) :: t, tb, te
t=0._real64
tb=omp_get_wtime()
!$omp target map(to:x(1:n))
    map(tofrom:y(1:n)) do i=1,n
        y(i) = a * x(i) +
    y(i) end do
!$omp end target
te=omp_get_wtime(
) t=tb-te
write(*,*) "Time of kernel:",
t end subroutine
```

```
ifx -fiopenmp -fopenmp-targets=spir64 -o saxpy saxpy.f90
```

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



Programmers must help the compiler with the size of the data transfer

Minimize Data Copy Across Target Regions

- Use `target data`, `target enter data`, and `target exit data` to form target data region and optimize sharing of data between host and device
  - Maps variables, code execution not offloaded
  - Variables remain on device for duration of the target data region
  - **`target update`** construct can copy values between host and device

`target data` Construct Syntax

- 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)
```

Target Data Example

- Use target data construct to create target data environment

```
!$omp target data map(tofrom: x)

          !$omp target map(to: y)
          1st target region, device operations on x and y
          !$omp end target
          host_update(y);
          !$omp target map(to: y)
          2nd target region, device operations on x and y
          !$omp end target
```

Device data environment created, array x is mapped

y must be mapped at each target region since it's updated by the host here

`target update` Construct Syntax

- 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*`)`

Target Enter/Exit Data and Update Example

- Use **`target enter/exit data`** to map to/from target data environment
- Use **`target update`** to maintain consistency between host and device

```
!$omp target enter data map(to: y) map(alloc: x)
!$omp target
        ...//1st target region, device operations on x and y
!$omp end target
!$omp target update from(y)
host_update(y)
!$omp target update to(y)

!$omp target
        ...//2nd target region, device operations on x and y
!$omp end target
!$omp target exit data map(from:x)
```

Unstructured mapping, data environment can span multiple functions

y must be updated from and to the device since it's updated by the host here

Map Global Variable to Device

- Use **`declare target`** construct for to map variables to the device for the duration of the program

```
#pragma omp declare target
int a[N]
#pragma omp end declare target
…
init(a);
#pragma omp target update to(a)
…
#pragma omp target teams\
distribute parallel for
for (int i=0; i<N; i++){
          result[i] = process(a[i]);
}
```

```
module my_arrays
!$omp declare target (a)
integer :: a(N)
end module
…
use my_arrays
integer :: i
call init(a);
!$omp target update to(a)
…
!$omp target teams distribute &
!$omp& parallel do
do i=1,N
          result(i) =
process(a(i));
end do
```

Unified Shared Memory

- Single address space for CPU and GPU
- Data migration among CPU and GPUs transparent to the application
  - Explicit mapping of data not required

Type	Location	Accessible From	Allocation Routine
Host	Host	Host or Device	omp_target_alloc_host(size, device_num)
Device	Device	Device	omp_target_alloc_device(size, device_num)
Shared	Host or Device	Host or Device	omp_target_alloc_shared(size, device_num)

- Use Shared or Host memory for implicit data movement to achieve ease of coding
- Use Device memory for explicit data movement to achieve maximum performance

USM Example (Fortran)

```
program main
use omp_lib
integer, parameter :: N=16
integer :: i, dev
integer, allocatable :: x(:)

dev = omp_get_default_device()
!$omp allocate allocator(omp_target_shared_mem_alloc)
allocate(x(N))

do i=1,N
   x(i) = i
end do

!$omp target has_device_addr(x)
!$omp teams distribute parallel do
do i=1,N
   x(i) = x(i) * 2
end do
!$omp end target
…
deallocate(x)
…
end program main
```

`omp_target_host_mem_alloc` and `omp_target_device_mem_alloc` allocation types also available

USM support via managed memory allocator

Unified Shared Memory (Implicit) Example

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define SIZE 1024
#pragma omp requires unified_shared_memory
int main() {
  int deviceId = (omp_get_num_devices() > 0) ?
       omp_get_default_device() : omp_get_initial_device();
  int *a = (int *)omp_target_alloc_shared(SIZE * sizeof(int) , deviceId);
  int *b = (int *)omp_target_alloc_shared(SIZE * sizeof(int) , deviceId);
  for (int i = 0; i < SIZE; i++) {
    a[i] = i; b[i] = SIZE - i;
  }
#pragma omp target teams distribute parallel for
  for (int i = 0; i < SIZE; i++) {
    a[i] += b[i];
  }
  for (int i = 0; i < SIZE; i++) {
    if (a[i] != SIZE) {
      printf("%s failed\n", __func__);
      return EXIT_FAILURE;
    }
  }
  omp_target_free(a, deviceId);
  omp_target_free(b, deviceId);
  printf("%s passed\n", __func__);
  return EXIT_SUCCESS;
}
```

USM support via managed memory allocator

Unified Shared Memory (Explicit) Example

```
…
int main() {
  int deviceId = (omp_get_num_devices() > 0) ? omp_get_default_device() : omp_get_initial_device();
  int *a = (int *)malloc(SIZE * sizeof(int));     int *b = (int *)malloc(SIZE * sizeof(int));
  for (int i = 0; i < SIZE; i++) {
    a[i] = i;  b[i] = SIZE - i;
  }
 int *a_dev = (int *)omp_target_alloc_device(SIZE * sizeof(int) , deviceId);
 int *b_dev = (int *)omp_target_alloc_device(SIZE * sizeof(int) , deviceId);
 int error=omp_target_memcpy(a_dev, a, SIZE*sizeof(int), 0, 0, deviceId, 0);
 error=omp_target_memcpy(b_dev, b, SIZE*sizeof(int), 0, 0, deviceId, 0);
 #pragma omp target teams distribute parallel for
  for (int i = 0; i < SIZE; i++) {
    a_dev[i] += b_dev[i];
  }
 error=omp_target_memcpy(a, a_dev, SIZE*sizeof(int), 0, 0, 0, deviceId);
 error=omp_target_memcpy(b, b_dev, SIZE*sizeof(int), 0, 0, 0, deviceId);

  for (int i = 0; i < SIZE; i++) {
    if (a[i] != SIZE) { printf("%s failed\n", __func__);   return EXIT_FAILURE; }}
 omp_target_free(a_dev, deviceId);
 omp_target_free(b_dev, deviceId);
  free(a);     free(b);
  printf("%s passed\n", __func__);
  return EXIT_SUCCESS;
}
```

Explicit Data Movement from Host to Device

Explicit Data Movement from Device to Host

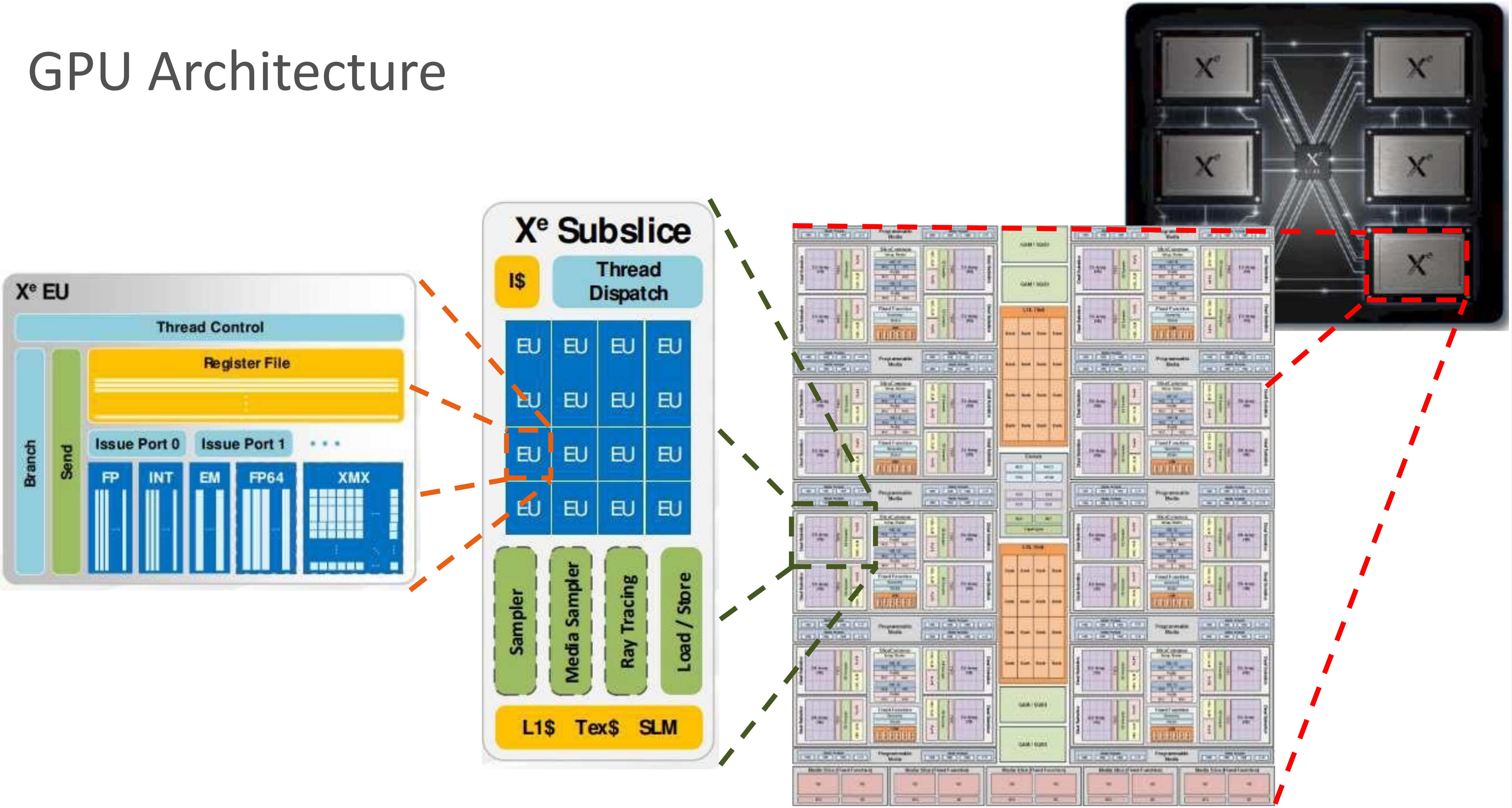
Parallelism

Creating Parallelism on the Target Device

- The `target construct` transfers the control flow to the target device
  - Transfer of control is sequential and synchronous

- OpenMP* separates offload and parallelism
  - Programmers need to explicitly create parallel regions on the target device
  - In theory, this can be combined with any OpenMP construct
  - In practice, there is only a useful subset of OpenMP for a target device (more later)

GPU Architecture



OpenMP* GPU Offload and OpenMP Constructs

- OpenMP GPU offload support all “normal” OpenMP constructs
  - E.g. parallel, for/do, barrier, sections, tasks, etc.
  - Not every construct will be useful
- Full threading model outside of a single GPU subslice not supported
  - No synchronization among subslices
  - No coherence and memory fence between among subslice L1 caches

Example: saxpy

- On the device, the **`parallel`** construct creates a team of threads to be executed on one subslice or stream multiprocessor

```
subroutine saxpy(a, x, y, n)
...
!$omp target map(to:x(1:n))
map(tofrom:y(1:n))
!$omp parallel do simd
    do i=1,n
        y(i) = a * x(i) + y(i)
    end do
!$omp end target
end subroutine
```

host

target

host

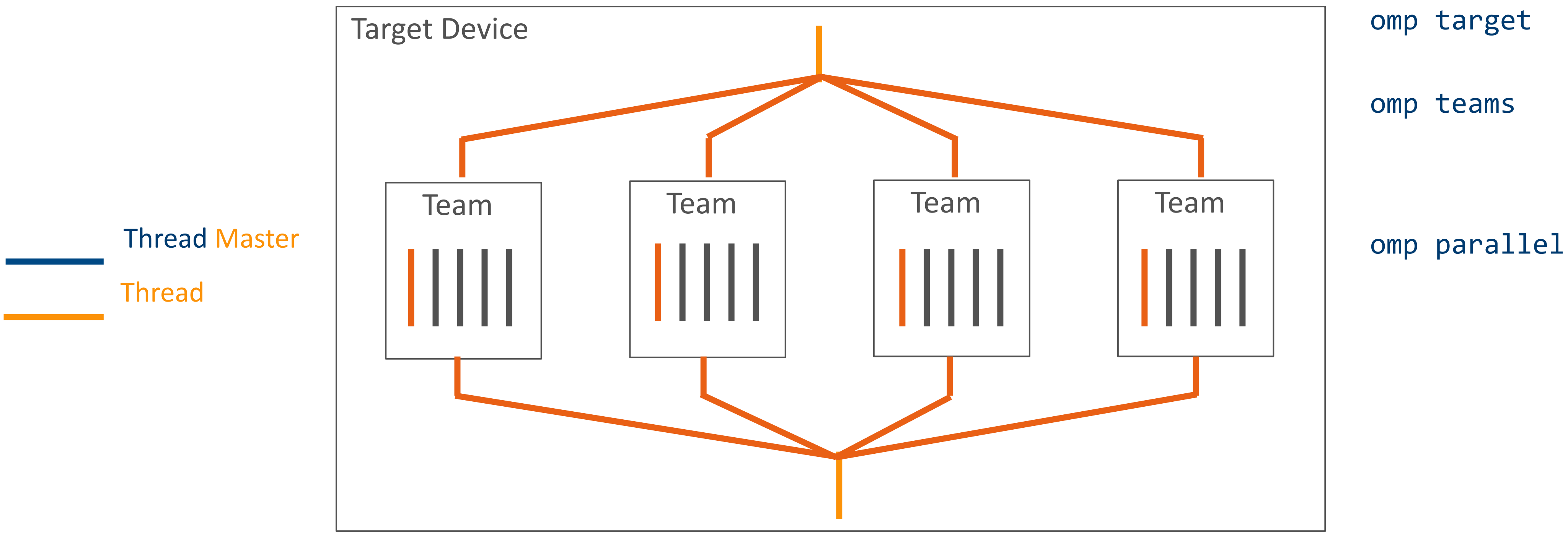
GPUs are multi-level devices:
SIMD, threads, thread blocks

Create a team of threads to execute the loop in parallel and SIMDify.
Only one GPU subslice utilized, GPU significantly underutilized

```
icx -fiopenmp -fopenmp-targets=spir64 -o saxpy saxpy.c
```

Teams Construct

- Creates multiple master threads, effectively creates a set of thread teams (league)
- Synchronization does not apply across teams.



Teams Construct

- Support multi-level parallel devices
- Syntax (C/C++):

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

- Syntax (Fortran):

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

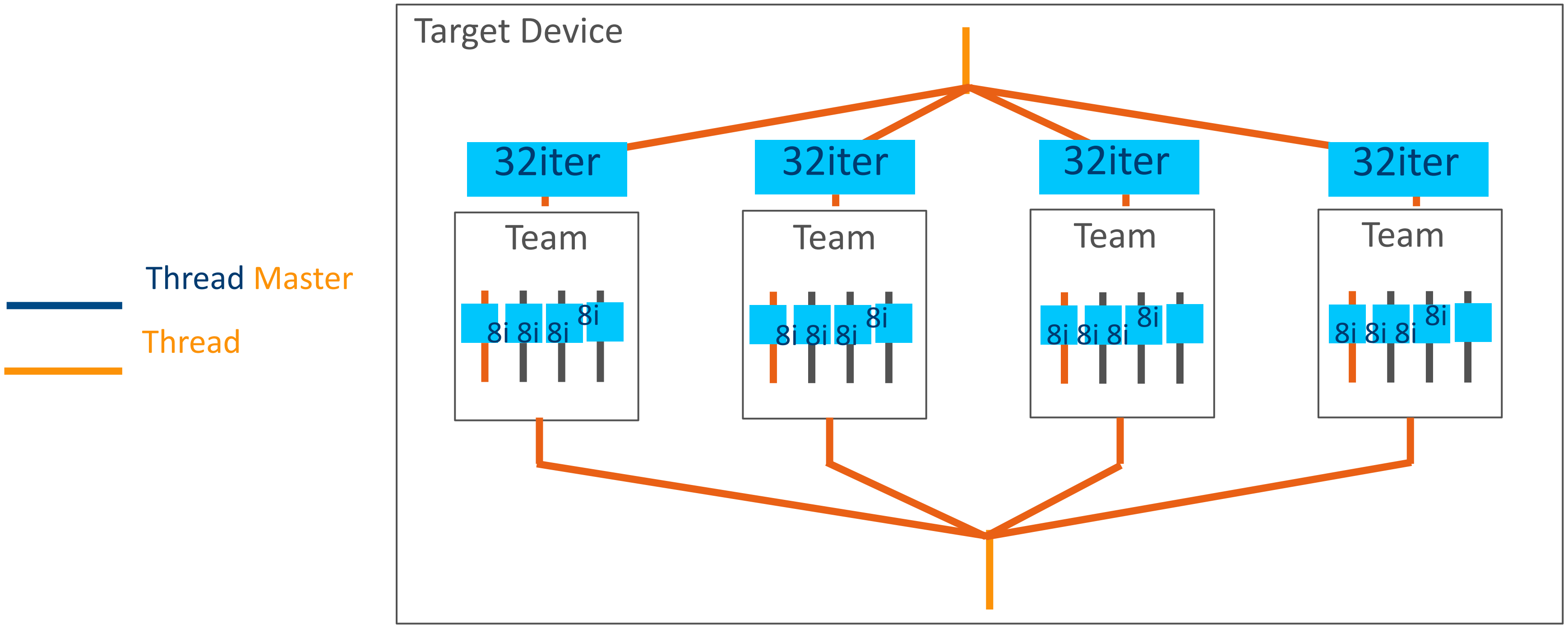
- Clauses

```
num_teams(integer-expression), thread_limit(integer-expression)
default(shared | firstprivate | private none)
private(list), firstprivate(list), shared(list), reduction(operator:list)
```

Distribute Construct

- **`distribute`** construct distributes iterations of a loop across the different teams
  - Worksharing within a league
  - Nested inside a **`teams`** region
  - Can specify distribution schedule
  - Similar to for/do construct for parallel regions
  - Syntax
    - `#pragma omp distribute` *[clause[[,] clause]…]*
    - `!$omp distribute` *[clause[[,] clause]…]*

Distribute Diagram



omp target

omp teams
omp distribute

omp parallel
omp for/do

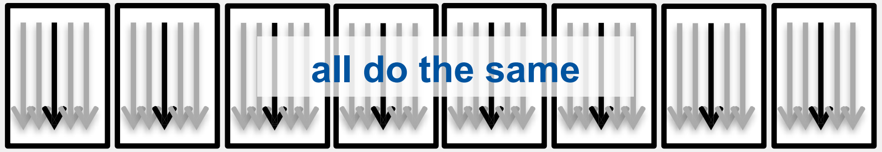
omp simd

Multi-level Parallel saxpy

```
subroutine saxpy(a, x, y, n)
    !$omp target map(to:x[1:n]) map(tofrom(y[1:n])









            do ib=1,sz,num_blocks


                do i=ib,ib+num_blocks-1



                    y(i) = a * x(i) + y(i)
```

Multi-level Parallel saxpy

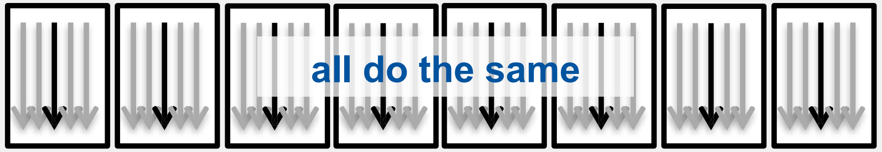
```
subroutine saxpy(a, x, y, n)
    !$omp target map(to:x[1:n]) map(tofrom(y[1:n])

        !$omp teams num_teams(num_blocks)
```



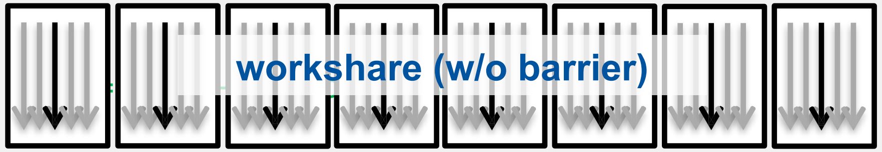

```
            do ib=1,sz,num_blocks

                do i=ib,ib+num_blocks-1

                    y(i) = a * x(i) + y(i)
```

Multi-level Parallel saxpy

```
subroutine saxpy(a, x, y, n)
    !$omp target map(to:x[1:n]) map(tofrom(y[1:n])

        !$omp teams num_teams(num_blocks)
```



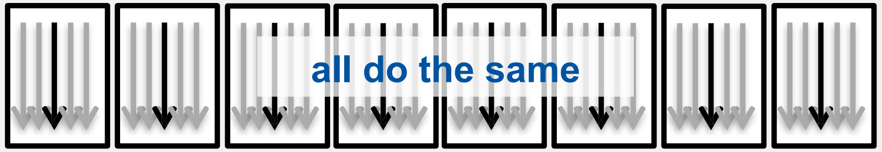

```
            !$omp distribute
            do ib=1,sz,num_blocks
```



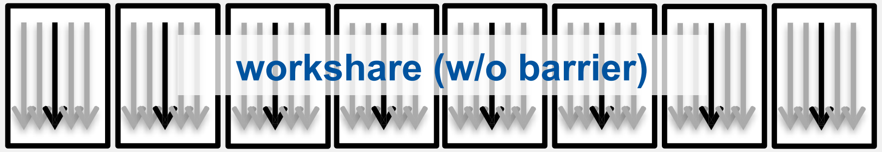

```
                y(i) = a * x(i) + y(i)
```

Multi-level Parallel saxpy

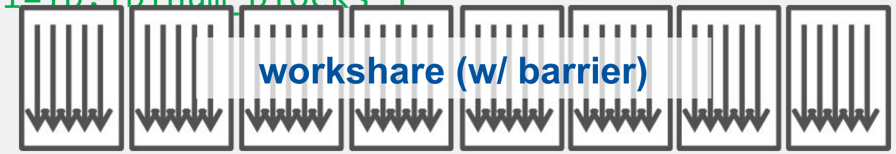
```
subroutine saxpy(a, x, y, n)
    !$omp target map(to:x[1:n]) map(tofrom(y[1:n])

        !$omp teams num_teams(num_blocks)
```




```
            !$omp distribute
            do ib=1,sz,num_blocks
```




```
                !$omp parallel for simd
                 do i=ib,ib+num_blocks-1
```




```
                    y(i) = a * x(i) + y(i)
```

Multi-level Parallel saxpy

- For convenience, OpenMP* defines composite construct to implement the required code transformation

```
void saxpy(float a, float* x, float* y, int sz) {
    #pragma omp target teams distribute parallel for simd \
            num_teams(num_blocks) map(to:x[0:sz]) map(tofrom(y[0:sz])
    for (int i = 0; i < sz; i++) {
        y[i] = a * x[i] + y[i];
    }
}
```

```
subroutine saxpy(a, x, y, n)
    ! Declarations omitted
!$omp omp target teams distribute parallel do simd &
!$omp&   num_teams(num_blocks) map(to:x) map(tofrom(y)
    do i=1,n
        y(i) = a * x(i) + y(i)
    end do
!$omp end target teams distribute parallel do simd
end subroutine
```

Conclusion

Calling MKL OpenMP offload (OpenMP => 5.1)

- Compile:
  icx -fiopenmp -fopenmp-version=51 -fopenmp-targets=spir64 -qmkl \
     -o dgemm_sample.o -c dgemm_sample.c

- Link:
  icx -fiopenmp -fopenmp-version=51 -fopenmp-targets=spir64 -qmkl
     -o dgemm_sample dgemm_sample.o

- Many more options (64/32-bit integers, static/dynamic linking, Linux*/Windows*, etc.) are available through the:

[Intel® oneAPI Math Kernel Library Link Line Advisor]

Notices & Disclaimers

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at intel.com, or from the OEM or retailer.

The benchmark results reported herein may need to be revised as additional testing is conducted. The results depend on the specific platform configurations and workloads utilized in the testing, and may not be applicable to any particular user's components, computer system or workloads. The results are not necessarily representative of other benchmarks and other benchmark results may show greater or lesser impact from mitigations.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.
Notice revision #20110804

intel®