OPENMP* GPU OFFLOAD BASICS

Objectives

- To learn the basic OpenMP* offload constructs to deploy OpenMP application for execution on GPUs

- Knowledge of using OpenMP with Fortran, C or C++ on CPUs

Agenda

- OneAPI and OpenMP* Offload
- OpenMP on CPUs Review
- Introduction to OpenMP Offload
- Constructs to Manage Device Data
- Constructs to Leverage Parallelism
- Case Study
- Summary

ONEAPI AND OPENMP* OFFLOAD

PROGRAMMING CHALLENGES
FOR MULTIPLE ARCHITECTURES

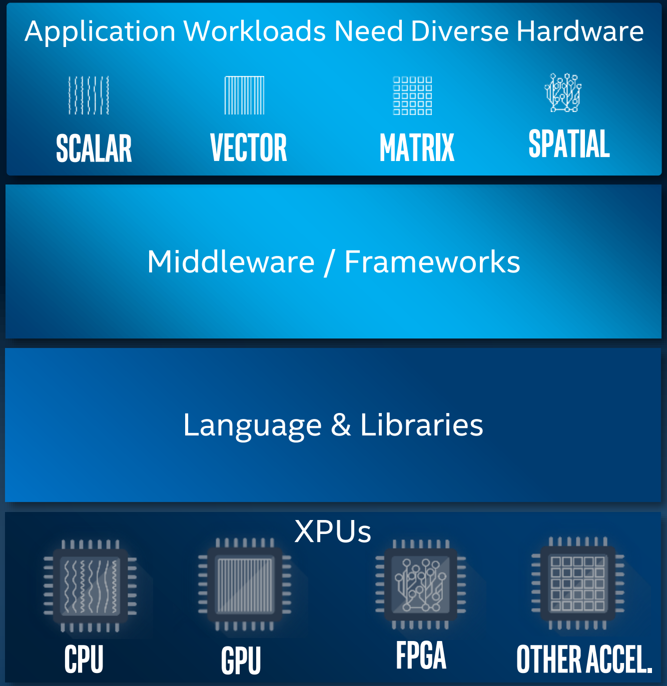
Growth in specialized workloads

No common programming language or APIs

Inconsistent tool support across platforms

Each platform requires unique software investment

Diverse set of data-centric hardware required



Refer to software.intel.com/articles/optimization-notice for more information regarding performance & optimization choices in Intel software products.

INTRODUCING ONEAPI

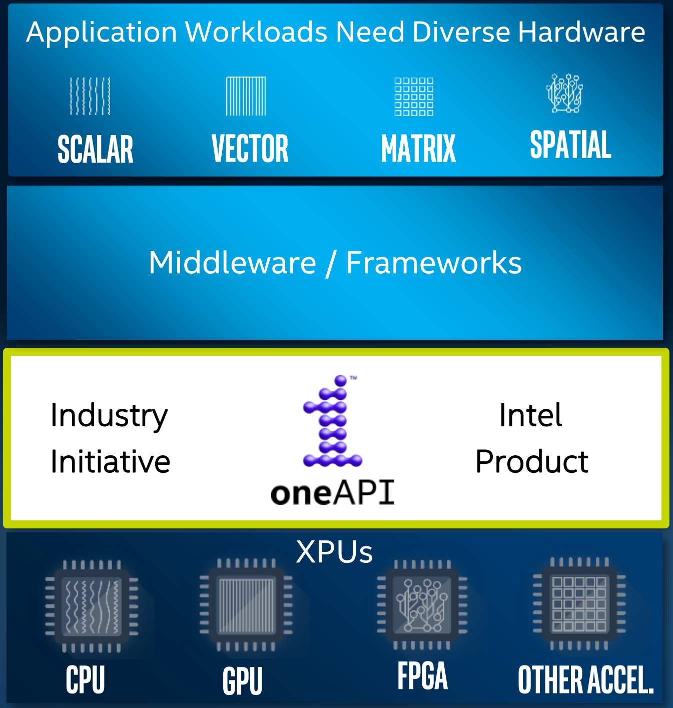
Unified programming model to simplify development across diverse architectures

Unified and simplified language and libraries for expressing parallelism

Uncompromised native high-level language performance

Based on industry standards and open specifications

Interoperable with existing HPC programming models



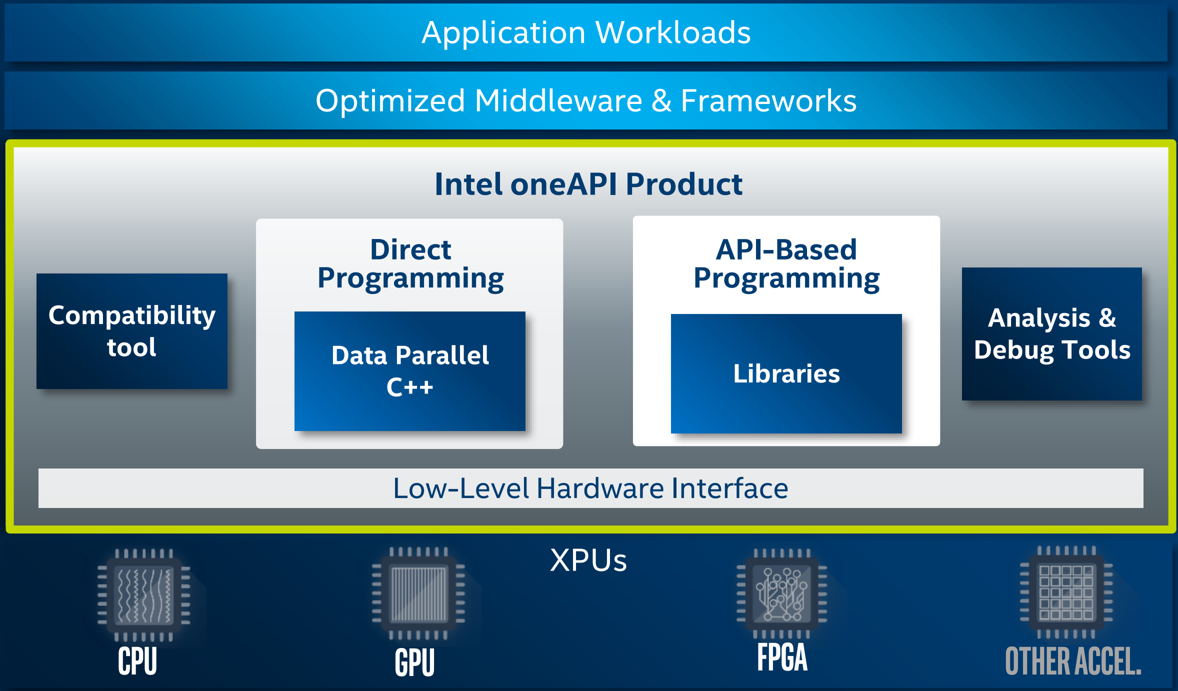
Refer to software.intel.com/articles/optimization-notice for more information regarding performance & optimization choices in Intel software products.

INTEL® ONEAPI PRODUCTS(BETA)

Distributed through a core toolkit and a complementary set of add-on domain-specific toolkits

Includes Data Parallel C++ (DPC++) compatibility tool for code migration along with advanced performance analysis and debug tools

Beta Available Now



Visit software.intel.com/oneapi for more details

Some capabilities may differ per architecture and custom-tuning will still be required. Other accelerators to be supported in the future.

Refer to software.intel.com/articles/optimization-notice for more information regarding performance & optimization choices in Intel software products.

INTEL® ONEAPI TOOLKITS(BETA)

TOOLKITS TAILORED TO YOUR NEEDS

Domain-specific sets of tools to get your job done quickly.

Intel® oneAPI Base Toolkit

A core set of high-performance tools for building Data Parallel C++ applications and oneAPI library based applications

Learn More

Intel® oneAPI HPC Toolkit

Everything HPC developers need to deliver fast C++, Fortran, & OpenMP* applications that scale

Learn More

Intel® oneAPI IoT Toolkit

Tools for building high-performing, efficient, reliable solutions that run at the network's edge

Learn More

Intel® oneAPI DL Framework Developer Toolkit

Tools for developers & researchers who build deep learning frameworks or customize existing ones so applications run faster

Learn More

Intel® oneAPI Rendering Toolkit

Powerful rendering libraries to create high-performance, high-fidelity visualization applications

Learn More

Toolkits Powered by oneAPI

Intel® System Bring-Up Toolkit

Tools to debug & tune power & performance in pre- & post-silicon development

Learn More

Intel® Distribution of OpenVINO™ Toolkit

Tools to build high performance deep learning inference & computer vision applications (production-level tool)

Learn More

Intel® AI Analytics Toolkit

Tools to build applications that leverage machine learning & deep learning models

Learn More

Refer to software.intel.com/articles/optimization-notice for more information regarding performance & optimization choices in Intel software products.

INTEL® ONEAPI HPC TOOLKIT (BETA)

A toolkit that makes it easier to build, analyze, optimize & scale HPC applications for Intel® Xeon® Scalable, Intel® Core™ processors & Intel® Accelerators.

Who Uses It?

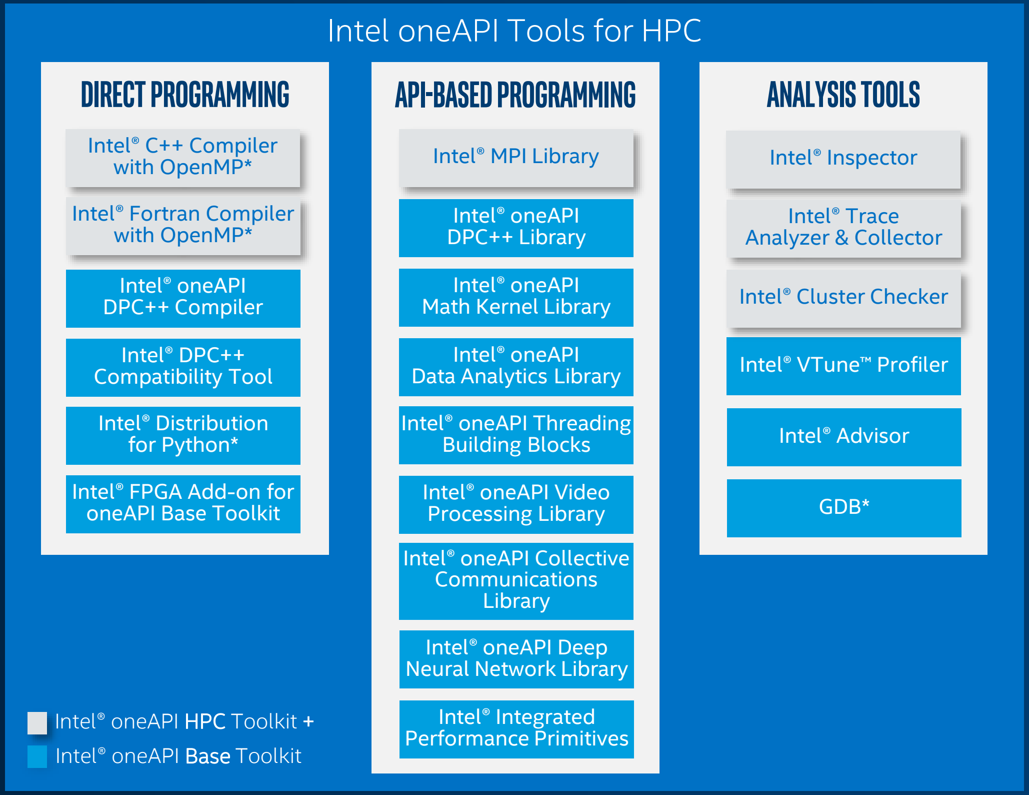
C/C++, Fortran, OpenMP* & MPI application developers

Top Features/Benefits

Optimized compilers & performance libraries for Intel® architectures

Powerful analysis tools to identify optimization opportunities for threading, memory & offloading

Standards-driven to scale forward & preserve development investment



Intel oneAPI Tools for HPC

DIRECT PROGRAMMING
- Intel® C++ Compiler with OpenMP*
- Intel® Fortran Compiler with OpenMP*
- Intel® oneAPI DPC++ Compiler
- Intel® DPC++ Compatibility Tool
- Intel® Distribution for Python*
- Intel® FPGA Add-on for oneAPI Base Toolkit

API-BASED PROGRAMMING
- Intel® MPI Library
- Intel® oneAPI DPC++ Library
- Intel® oneAPI Math Kernel Library
- Intel® oneAPI Data Analytics Library
- Intel® oneAPI Threading Building Blocks
- Intel® oneAPI Video Processing Library
- Intel® oneAPI Collective Communications Library
- Intel® oneAPI Deep Neural Network Library
- Intel® Integrated Performance Primitives

ANALYSIS TOOLS
- Intel® Inspector
- Intel® Trace Analyzer & Collector
- Intel® Cluster Checker
- Intel® VTune™ Profiler
- Intel® Advisor
- GDB*

⬜ Intel® oneAPI **HPC** Toolkit +

🟦 Intel® oneAPI **Base** Toolkit

Refer to software.intel.com/articles/optimization-notice for more information regarding performance & optimization choices in Intel software products.

INTEL® C++ & FORTRAN COMPILERS(BETA)

HIGH PERFORMANCE COMPILERS WITH OPENMP*

Deliver Industry-leading C/C++ & Fortran code performance with OpenMP*, unleash the power of the latest Intel® platforms

Develop optimized & vectorized code for Intel® architectures, including Intel® Xeon® processors

Leverage latest language & OpenMP standards, & compatibility with leading compilers & IDEs

Beta support for Intel GEN9 GPU with OpenMP offload

Refer to software.intel.com/articles/optimization-notice for more information regarding performance & optimization choices in Intel software products.

OPENMP* ON CPUS

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

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

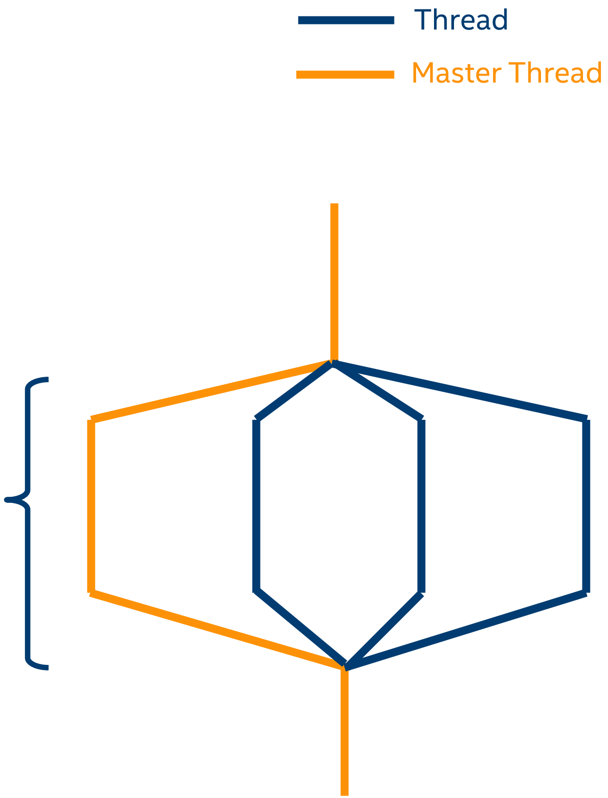
Create threads with the **parallel** construct

```
#include <omp.h>

void saxpy()
{
   float a, x[ARRAY_SZ], y[ARRAY_SZ];
#pragma omp parallel
{
   int id=omp_get_thread_num();
   int nthrs=omp_get_num_threads();
   for (int i=id; i < ARRAY_SZ; i+=nthrs) {
      y[i] = a * x[i] + y[i];
   }
}
}
```

Parallel Region.

Team of **threads** created.

Each thread executes the same code redundantly



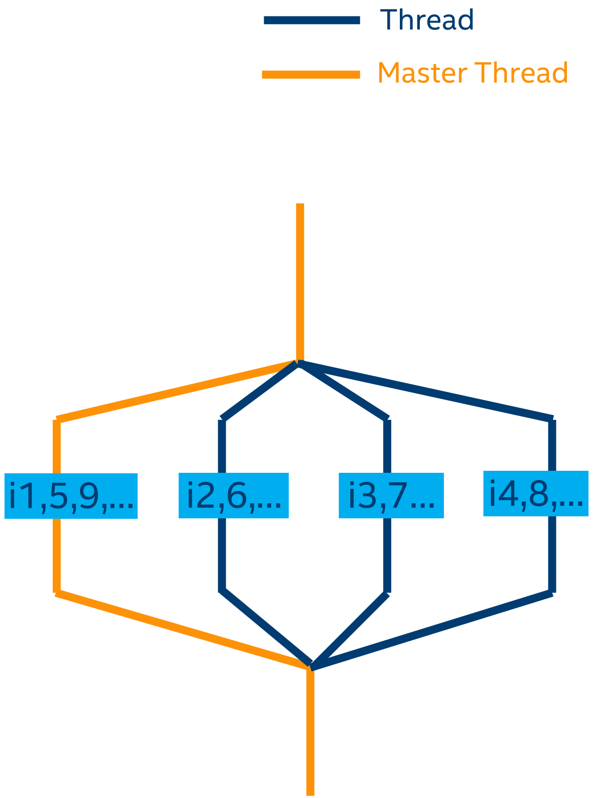
Loops

Use For/Do Loop Directive to Workshare

```
#include <omp.h>

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

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
   use omp_lib
   …
   !$omp parallel do reduction (+:total)
   do i=0,ARRAY_SZ
      total = total + x(i)
   end do
   !$omp end parallel do
…
end program main
```

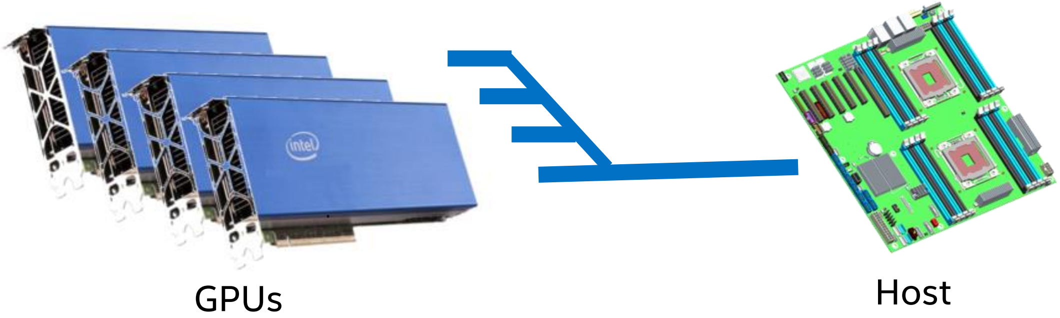
Other Notable OpenMP* Constructs

- 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

- Intel® C++ Compiler

```
icx –fiopenmp –fopenmp-targets=spir64 <source>.c
```

```
icpx –fiopenmp –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 transfer clauses control direction of data flow
- Array notation is used to describe array length

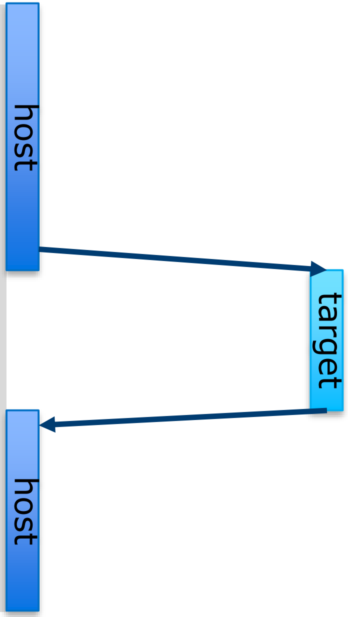
Target Region Example: saxpy

Sequential Host Code

Target Region

Sequential Host Code

```
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
```

Device Clause

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

```
#pragma 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

MANAGING DEVICE DATA

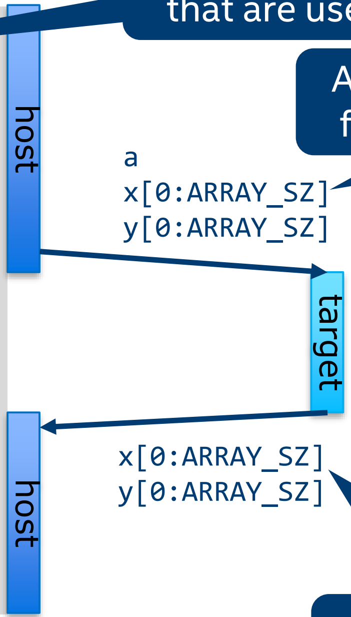
Offload Data

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

```
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);
}
```



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

All accessed arrays are copied from host to device and back

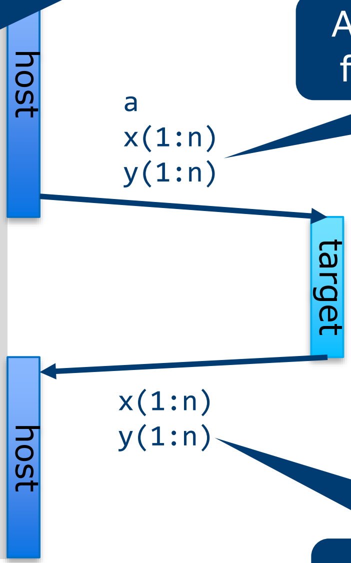
Copying x back is not necessary: it was not changed.

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

Example: saxpy

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

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

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



All accessed arrays are copied from host to device and back

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

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

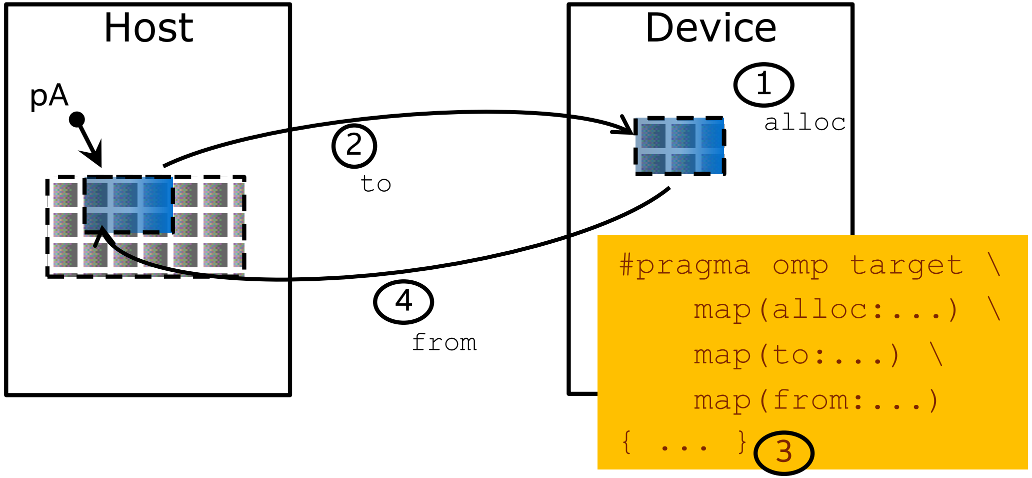
Map Clause

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

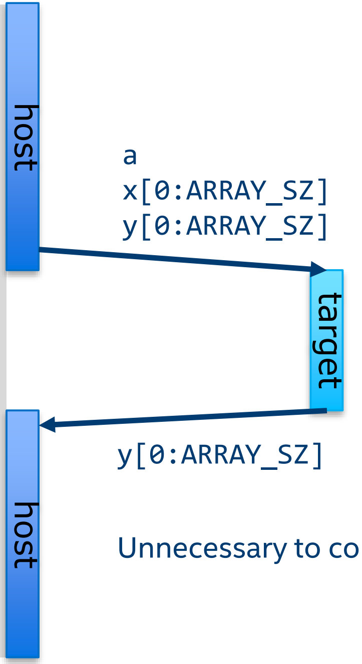
Use **map** clause to manually determine how an original variable in a data environment is mapped to a corresponding variable in a device data environment



```
#pragma omp target \
    map(alloc:...) \
    map(to:...) \
    map(from:...)
{ ... }
```

Example: saxpy

```
void saxpy() {
    double a, x[ARRAY_SZ], y[ARRAY_SZ];
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
#pragma omp target map(to:x) \
                   map(tofrom:y)
    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);
}
```



host

a
x[0:ARRAY_SZ]
y[0:ARRAY_SZ]

target

y[0:ARRAY_SZ]

host

Unnecessary to copy x back to the host

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

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])
```

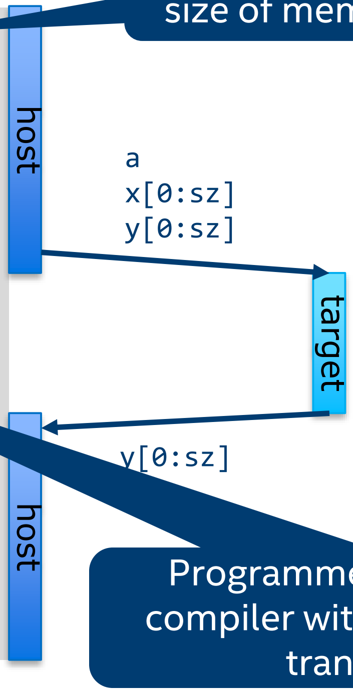
```
!$omp target map(to:array(start:end))
```

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

Example: saxpy

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

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



Programmers have to help the compiler with the size of the data transfer needed.

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

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

```
#pragma omp target data map(tofrom: x)
{
     #pragma omp target map(to: y)
     {
          ...//1st target region, device operations on x and y
     }
     host_update(y);
     #pragma omp target map(to: y)
     {
          ...//2nd target region, device operations on x and y
     }
}
```

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)
    if(scalar-expr)
```

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

```
#pragma omp target enter data map(to: y) map(alloc: x)
#pragma omp target
{    ...//1st target region, device operations on x and y
}
#pragma omp target update from(y)
host_update(y);
#pragma omp target update to(y)

#pragma omp target
{    ...//2nd target region, device operations on x and y
}
#pragma omp target exit data map(from:x)
```

Unstructured mapping, data environment can span multiple functions

y must be mapped at each target region 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
```

PARALLELISM

Creating Parallelism on the Target Device

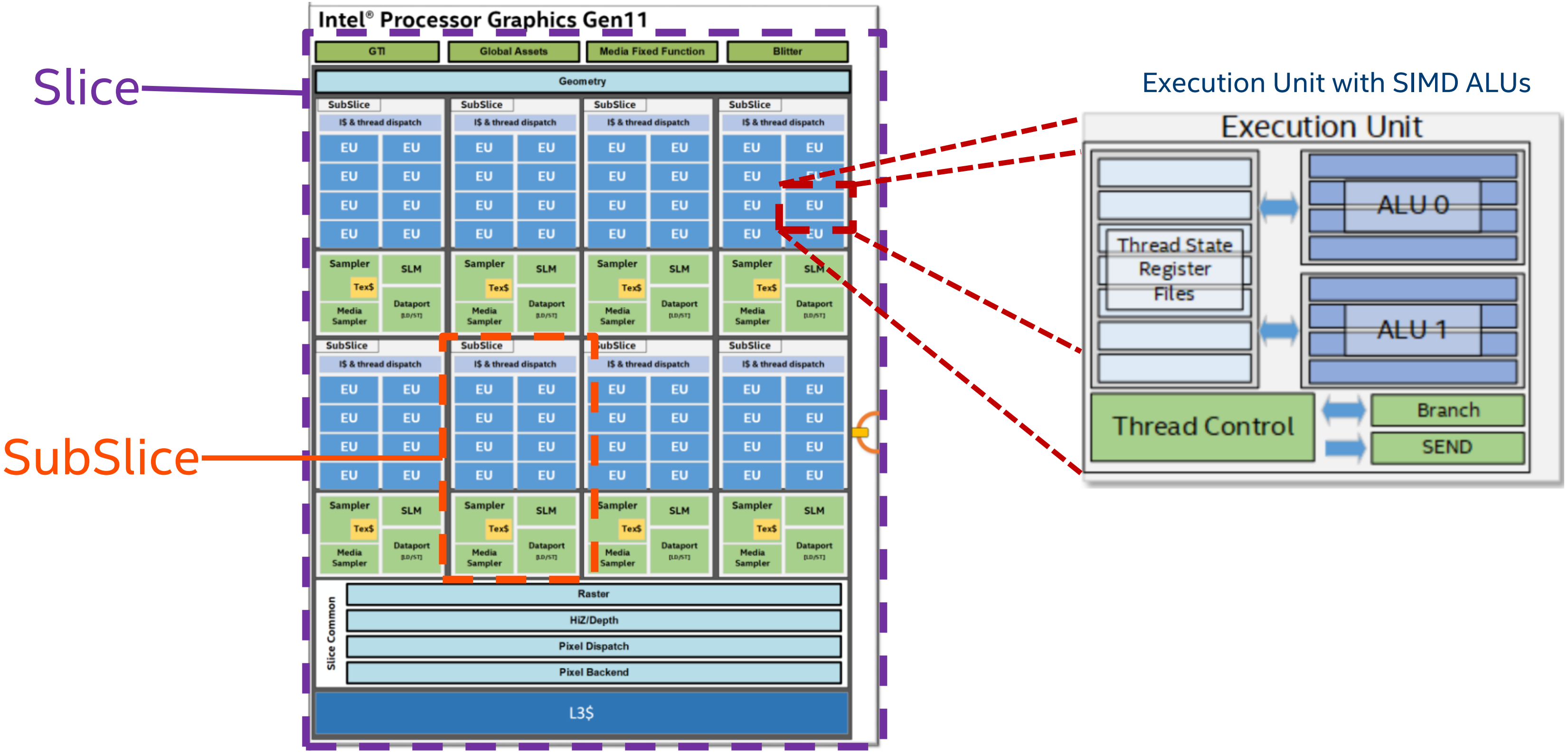
The **`target construct`** transfers the control flow to the target device

- 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



Slice

SubSlice

Execution Unit with SIMD ALUs

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

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

host

target

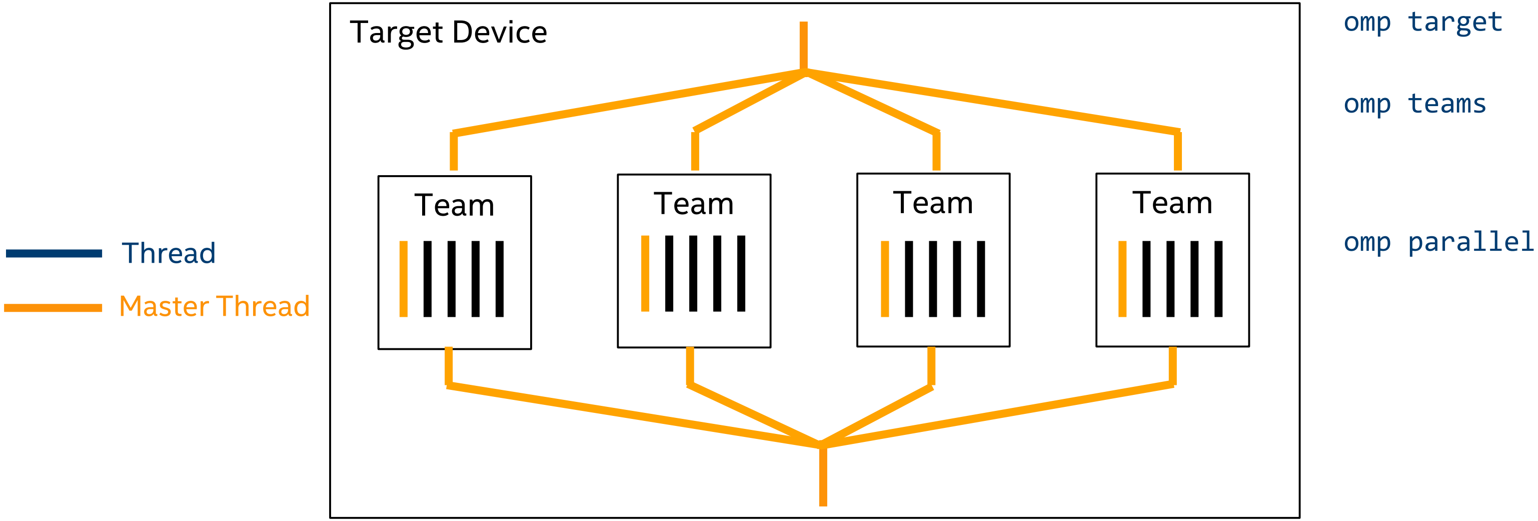
GPUs are multi-level devices: SIMD, threads, thread blocks

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

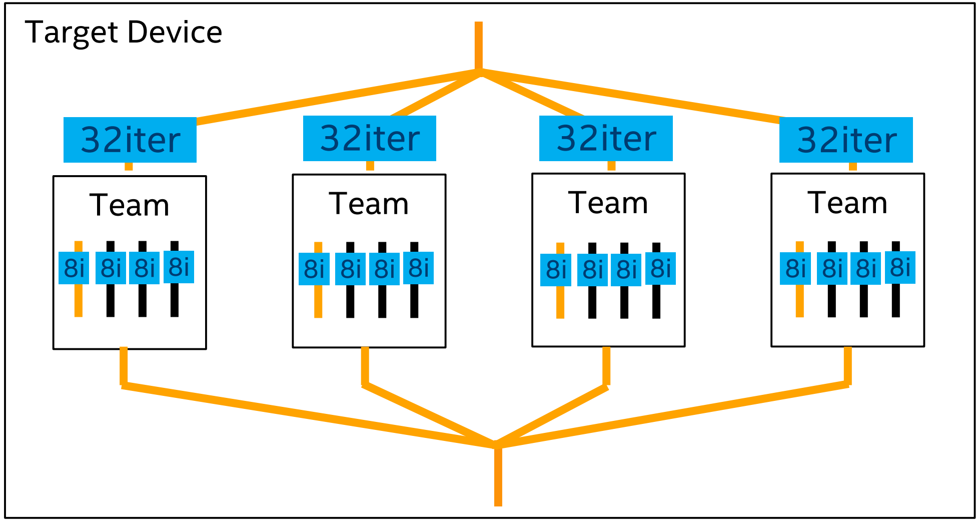
```
num_teams(integer-expression), thread_limit(integer-expression)
default(shared | firstprivate | private none)
private(list), firstprivate(list), shared(list), reduction(operator:list)
```

Distribute Construct

`distribute` construct distributes iterations of a loop across the different teams

- Worksharing within a league
- Nested in side 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



━━ Thread

━━ Master Thread

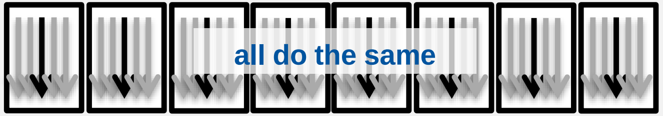
```
omp target

omp teams
omp distribute

omp parallel
omp for/do

omp simd
```

Multi-level Parallel saxpy

```
void saxpy(float a, float* x, float* y, int sz) {
    #pragma omp target map(to:x[0:sz]) map(tofrom(y[0:sz])
    {

        {




            for (ib = 0; ib < sz; ib += num_blocks) {




                for (int i = ib; i < ib + num_blocks; i++) {



                    y[i] = a * x[i] + y[i];
}   }   }   }   }
```

Multi-level Parallel saxpy

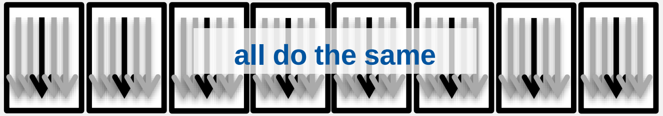
```
void saxpy(float a, float* x, float* y, int sz) {
    #pragma omp target map(to:x[0:sz]) map(tofrom(y[0:sz])
    {
        #pragma omp teams num_teams(num_blocks)
        {
```



```
            for (ib = 0; ib < sz; ib += num_blocks) {


                for (int i = ib; i < ib + num_blocks; i++) {


                    y[i] = a * x[i] + y[i];
}   }   }   }   }
```

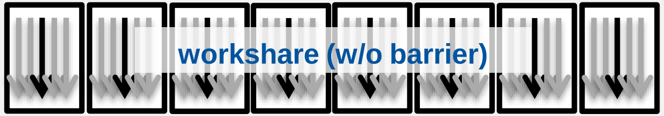
Multi-level Parallel saxpy

```
void saxpy(float a, float* x, float* y, int sz) {
    #pragma omp target map(to:x[0:sz]) map(tofrom(y[0:sz])
    {
        #pragma omp teams num_teams(num_blocks)
        {
```



all do the same

```
            #pragma omp distribute
            for (ib = 0; ib < sz; ib += num_blocks) {
```

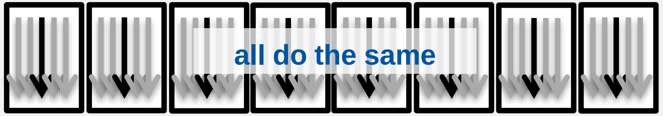


workshare (w/o barrier)

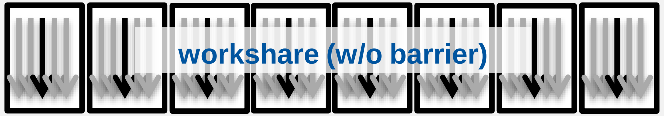
```
                for (int i = ib; i < ib + num_blocks; i++) {



                    y[i] = a * x[i] + y[i];
}   }   }   }   }
```

Multi-level Parallel saxpy

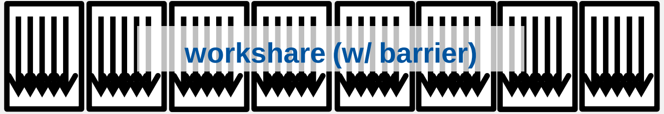
```
void saxpy(float a, float* x, float* y, int sz) {
    #pragma omp target map(to:x[0:sz]) map(tofrom(y[0:sz])
    {
        #pragma omp teams num_teams(num_blocks)
        {
```



```
            #pragma omp distribute
            for (ib = 0; ib < sz; ib += num_blocks) {
```



```
                #pragma omp parallel for simd
                for (int i = ib; i < ib + num_blocks; i++) {
```



```
                    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
```

Complete Saxpy Example

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

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

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

CASE STUDY: NWCHEM TCE CCSD(T)

TCE: Tensor Contraction Engine
CCSD(T): Coupled Cluster w/ Singles Doubles, Triples approximation

NWChem

Computational chemistry software package

- Quantum chemistry
- Molecular dynamics

Designed for large-scale supercomputers

Developed at the EMSL at PNNL

- EMSL: Environmental Molecular Sciences Laboratory
- PNNL: Pacific Northern National Lab

URL: http://www.nwchem-sw.org

Finding Offload Candidates

Requirements for offload candidates

- Compute-intensive code regions (kernels)
- Highly parallel
- Compute scaling stronger than data transfer, e.g., compute $O(n^3)$ vs. data size $O(n^2)$

Intel® Advisor: Offload Advisor can be used to identify candidates

Example Kernel (1 of 27 in total)

```
      subroutine offl_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,
     1                 h7d,triplesx,t2sub,v2sub)
c     Declarations omitted.
      double precision triplesx(h3d*h2d,h1d,p6d,p5d,p4d)
      double precision t2sub(h7d,p4d,p5d,h1d)
      double precision v2sub(h3d*h2d,p6d,h7d)
!$omp target
!$omp teams distribute parallel do private(p4,p5,p6,h2,h3,h1,h7)
      do p4=1,p4d
      do p5=1,p5d
      do p6=1,p6d
      do h1=1,h1d
      do h7=1,h7d
      do h2h3=1,h3d*h2d
       triplesx(h2h3,h1,p6,p5,p4)=triplesx(h2h3,h1,p6,p5,p4)
     1   - t2sub(h7,p4,p5,h1)*v2sub(h2h3,p6,h7)
      end do
      end do
      end do
      end do
      end do
      end do
!$omp end teams distribute parallel do
!$omp end target
      end subroutine
```

1.5GB data transferred (host to device)

1.5GB data transferred (device to host)

All kernels expose the same structure

7 perfectly nested loops

Some kernels contain inner product loop (then, 6 perfectly nested loops)

Trip count per loop is equal to "tile size" (20-30 in production)

Naïve data allocation (tile size 24)

- Per-array transfer for each `target` construct

triplesx:	1458 MB
t2sub, v2sub:	2.5 MB

Invoking the Kernels / Data Management

Simplified pseudo-code of the actual

```
!$omp target enter data alloc(triplesx(1:tr_size))
c     for all tiles
      do ...
        call zero_triplesx(triplesx)
        do ...
          call comm_and_sort(t2sub, v2sub)
!$omp target data map(to:t2sub(t2_size)) map(to:v2sub(v2_size))
          if (...)
            call sd_t_d1_1(h3d,h2d,h1d,p6d,p5d,p4d,h7,triplesx,t2sub,v2sub)
          end if
c         same for sd_t_d1_2 until sd_t_d1_9
!$omp target end data
        end do
        do ...
c         Similar structure for sd_t_d2_1 until sd_t_d2_9, incl. target data
        end do
        call sum_energy(energy, triplesx)
      end do
!$omp target exit data release(triplesx(1:size))
```

Allocate 1.5GB data once, stays on device.

Update 4MB of data for (potentially) multiple kernels.

Reduced data transfers:

- triplesx:
  - allocated once
  - always kept on the target
- t2sub, v2sub:
  - allocated after comm.
  - kept for (multiple) kernel invocations

CONCLUSION

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

Summary

- OpenMP* offload supported by the Intel® C++ Compiler and Intel® Fortran Compiler as part of the Intel® oneAPI HPC Toolkit
- Use the **`target`** directive to offload
- Use the **`map`** clause with **`target, target data, target enter/exit data`** directives to improve data transfer efficiency
- Use the **`teams/distribute`** directives fully utilize multiple GPU subslices
- Use the **`parallel/for/do`** directive to use the threads within a GPU subslice
- Use the **`simd`** directive for optimal simd execution on GPU execution units

Other Topics of Interest

- Using the Intel® Advisor : Offload Advisor to identify areas of code that are advantageous to offload
  - Provides performance speedup projection on accelerators
- Using the Intel® Advisor: Roofline Analysis to visualize hardware-imposed performance ceilings for the CPU and GPU.
  - Provides insights on bottlenecks and optimization steps

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.

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

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®

Software