Advanced OpenMP Tutorial

Christian Terboven — NHR4CES | RWTH AACHEN UNIVERSITY

Michael Klemm — OpenMP

Bronis R. de Supinski — LAWRENCE LIVERMORE NATIONAL LABORATORY

Members of the OpenMP Language Committee

ISC High Performance
The HPC Event.

Agenda – 09:00 through 13:00 ISC Time ☺

- OpenMP Overview (~20 min.)
- Techniques to Obtain High Performance with OpenMP: Memory Access (~30 min.)
- Techniques to Obtain High Performance with OpenMP: Loops (~30 min.)
- Techniques to Obtain High Performance with OpenMP: Vectorization (~20 min.)
- OpenMP for Attached Compute Accelerators (~90 min.)
- Future OpenMP Directions (~20 min.)

Break:
- Coffee: 11:00 – 11:30

Updated slides

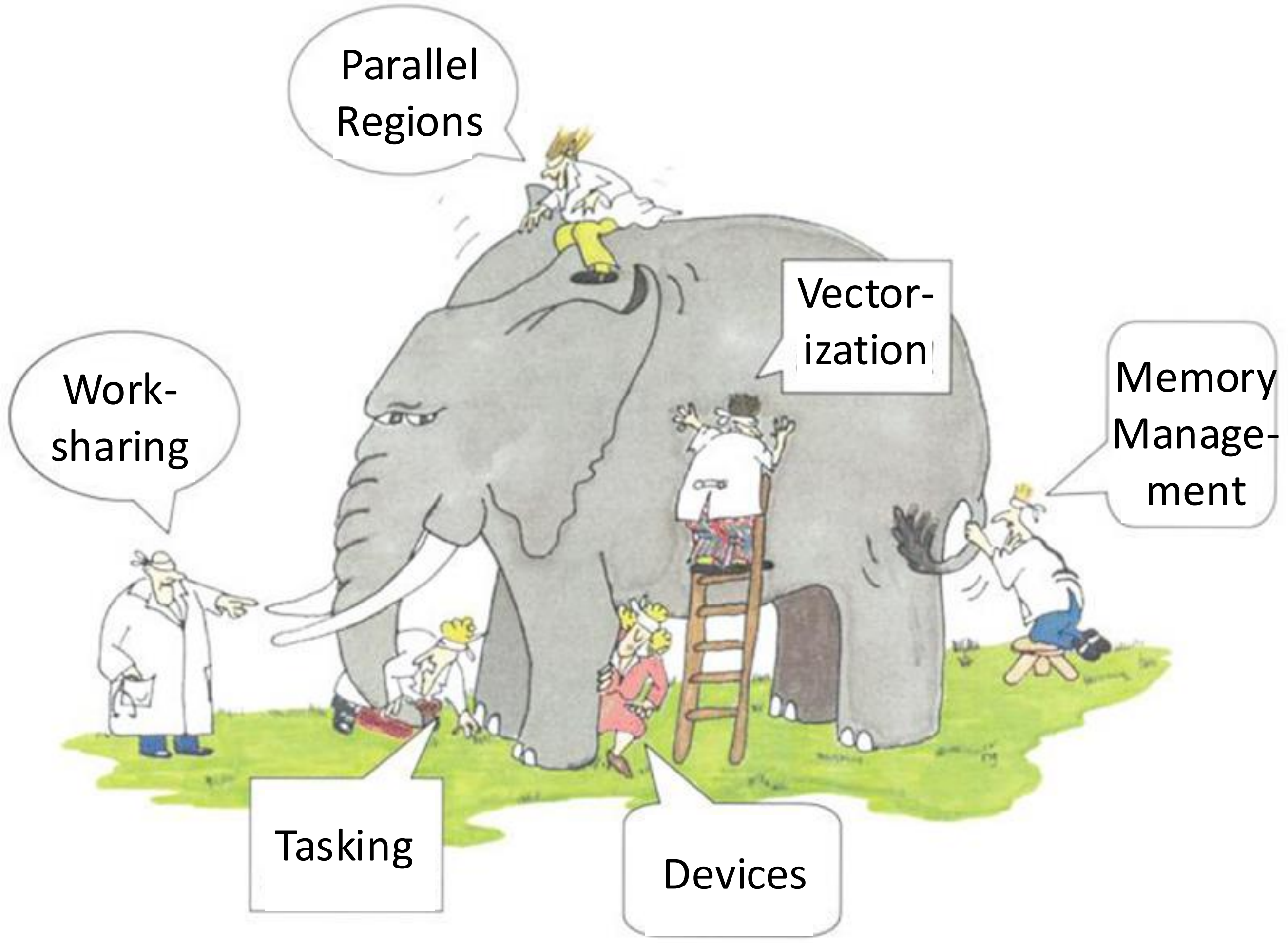
- Slides are never perfect …
- … but we offer a free update service :-)

https://bit.ly/isc25-adv-omp

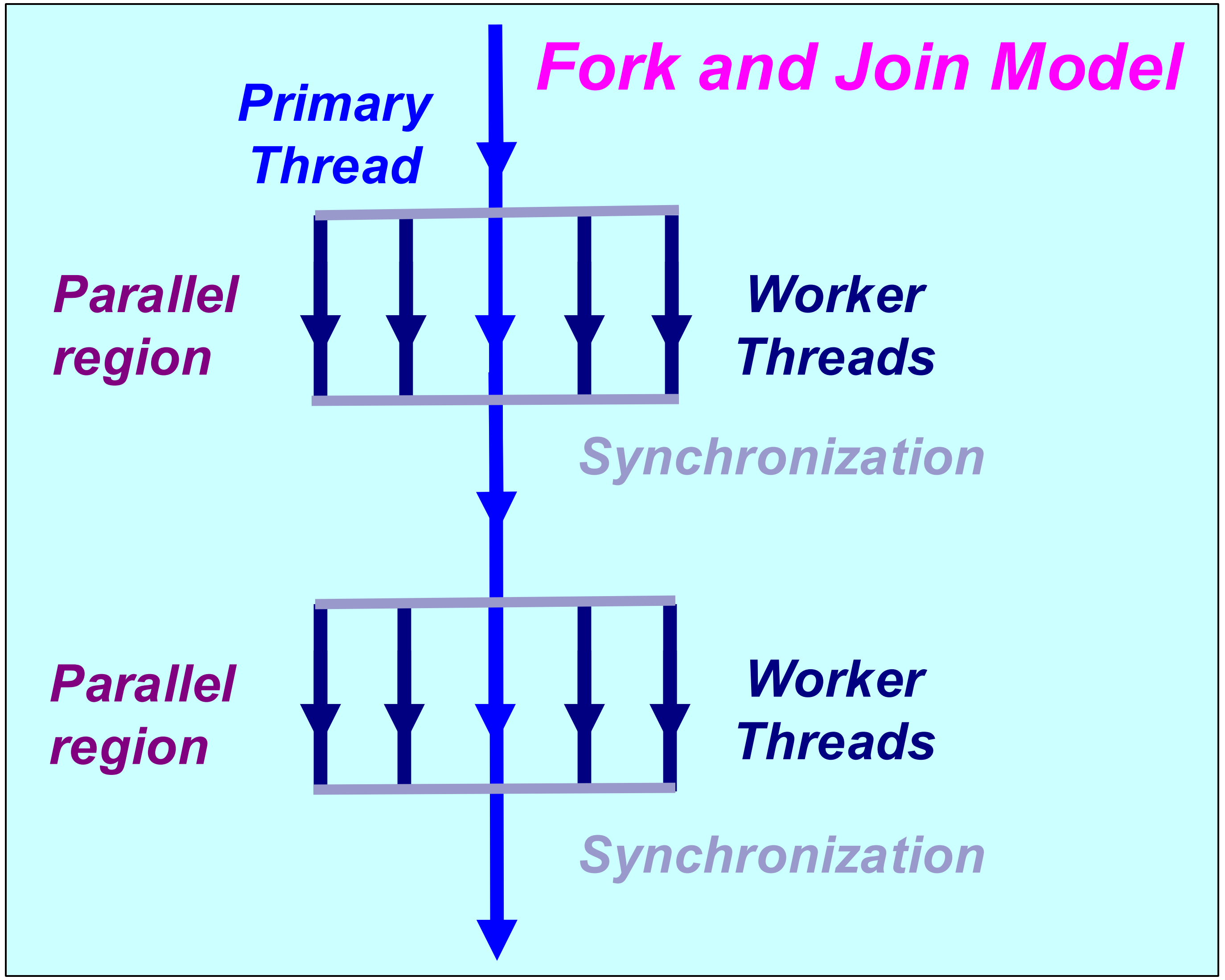
Core Concepts: Worksharing and Tasking

What is OpenMP?

- De-facto standard Application Programming Interface (API) to write shared memory parallel applications in C, C++, and Fortran
- Consists of compiler directives, runtime routines and environment variables
- Version 5.0 was released at SC18
- Version 5.2 was released at SC21
- Version 6.0 was released at SC24



The OpenMP Execution Model



```
#pragma omp parallel
{
    ....
}
```

```
#pragma omp parallel
{
    ....
}
```

The Worksharing Constructs

- *The work is distributed over the threads*
- *Must be enclosed in a parallel region*
- *Must be encountered by all threads in the team, or none at all*
- *No implied barrier on entry*
- *Implied barrier on exit (unless the nowait clause is specified)*
- *A work-sharing construct does not launch any new threads*

```
#pragma omp for
{
    ....
}
```

```
#pragma omp sections
{
       ....
}
```

```
#pragma omp single
{
       ....
}
```

The task construct

- Deferring (or not) a unit of work (executable for any member of the team)

```
#pragma omp task [clause[[,] clause]...]
{structured-block}
```

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

- Where clause is one of:

→ private(list) → firstprivate(list) → shared(list) → default(shared | none) → in_reduction(r-id: list)	**Data Environment**
→ allocate([allocator:] list) → detach(event-handler)	**Miscellaneous**
→ if(scalar-expression) → mergeable → final(scalar-expression)	**Cutoff Strategies**
→ depend(dep-type: list)	**Synchronization**
→ untied → priority(priority-value) → affinity(list)	**Task Scheduling**

Tasking execution model

- Supports unstructured parallelism
  - → unbounded loops

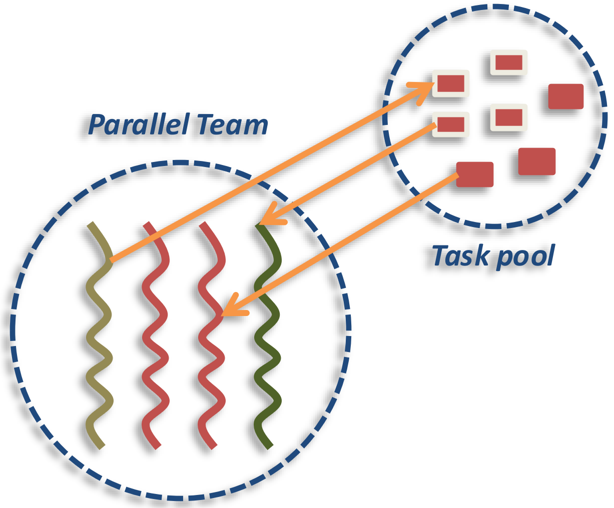
```
while ( <expr> ) {
    ...
}
```

  - → recursive functions

```
void myfunc( <args> )
{
    ...; myfunc( <newargs> ); ...;
}
```

- Example (unstructured parallelism)

```
#pragma omp parallel
#pragma omp single
while (elem != NULL) {
   #pragma omp task
      compute(elem);
   elem = elem->next;
}
```



- Several scenarios are possible:
  - → single creator, multiple creators, nested tasks (tasks & WS)
- All threads in the team are candidates to execute tasks

Single ~~and Master~~ and Masked / 1

- Single: only one thread in the team executes the code enclosed

```
#pragma omp single [private][firstprivate] \
                   [copyprivate][nowait]
{
    <code-block>
}
```

There is no implied barrier on entry or exit !

- Masked: rule-based selection of threads for region execution

```
#pragma omp masked [filter(integer-expression)]
{<code-block>}
```

Single ~~and Master~~ and Masked / 2

- Single: only one thread in the team executes the code enclosed

```
#pragma omp single [private][firstprivate] \
                   [copyprivate][nowait]
{
    <code-block>
}
```

- Masked: rule-based selection of threads for region execution

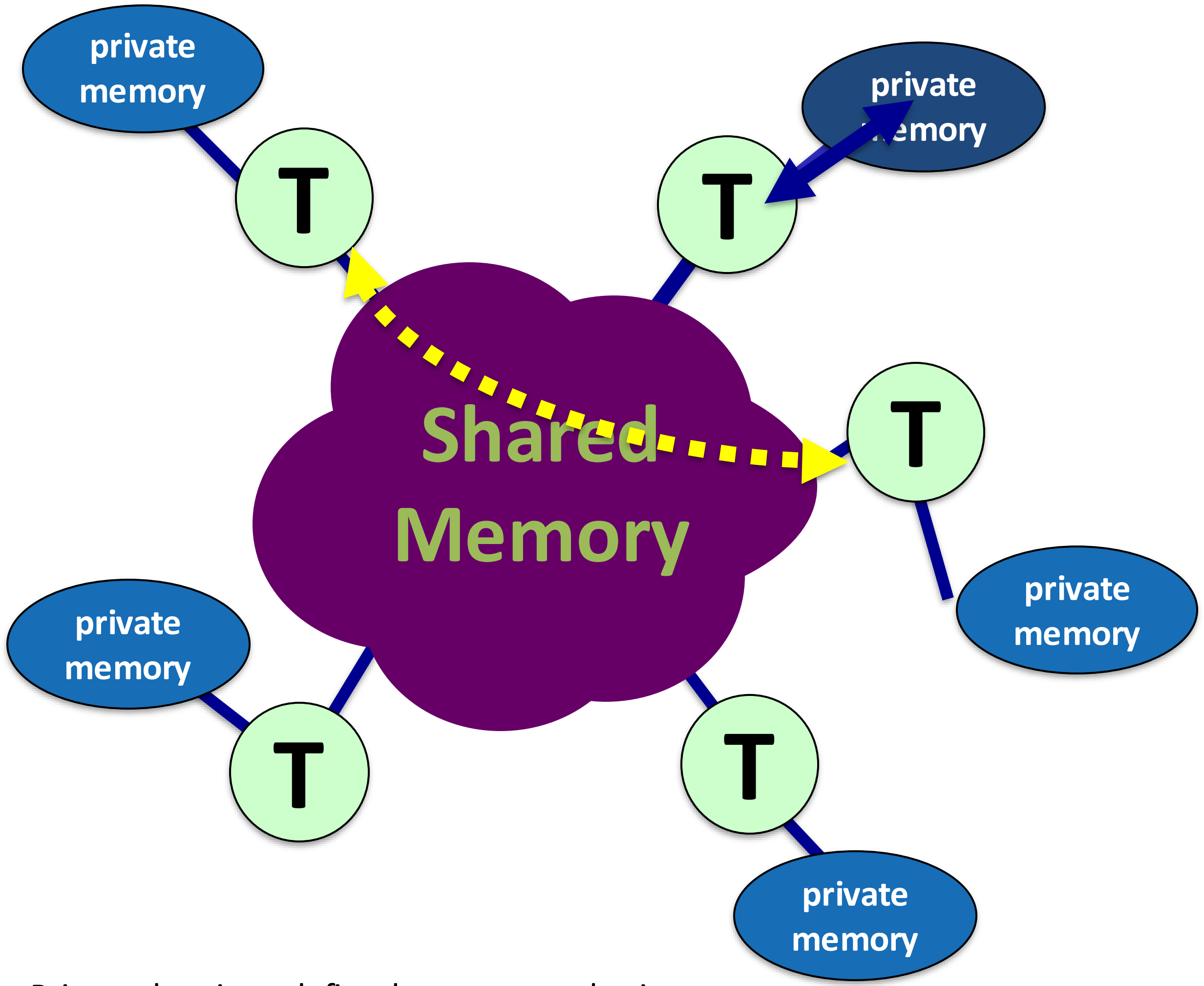
```
#pragma omp masked [filter(integer-expression)]
{<code-block>}
```

→Replacement of `master` construct:

```
#pragma omp masked [filter(0)]
{<code-block>}
```

Synchronization

The OpenMP Memory Model



Private data is undefined on entry and exit

- Can use firstprivate and lastprivate to address this

- ***All threads have access to the same, globally shared memory***
- ***Data in private memory is only accessible by the thread owning this memory***
- ***No other thread sees the change(s) in private memory***
- ***Data transfer is through shared memory and is 100% transparent to the application***

Barrier and Taskwait Constructs

- OpenMP `barrier` (implicit or explicit)
  - →All tasks created by any thread of the current *Team* are guaranteed to be completed at barrier exit

```
C/C++

#pragma omp barrier
```

- Task barrier: `taskwait`
  - →Encountering task is suspended until child tasks are complete
    - →Applies only to direct childs, not descendants!

```
C/C++

#pragma omp taskwait
```

The nowait Clause

- To minimize synchronization, some directives support the optional nowait clause
    - →If present, threads do not synchronize/wait at the end of that particular construct

- In C, it is one of the clauses on the pragma
- In Fortran, it is appended at the closing part of the construct

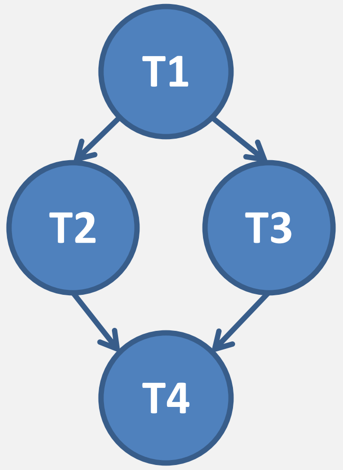
```
#pragma omp for nowait
{
        :
}
```

```
!$omp do
        :
        :
!$omp end do nowait
```

Task depend clause

- With task dependencies, a task cannot be executed until all its predecessor tasks are completed

```
int x = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(inout: x) //T1
  { ... }

  #pragma omp task depend(in: x)    //T2
  { ... }

  #pragma omp task depend(in: x)    //T3
  { ... }

  #pragma omp task depend(inout: x) //T4
  { ... }
}
```



The taskgroup Construct

C/C++

```
#pragma omp taskgroup
... structured block ...
```

Fortran

```
!$omp taskgroup
... structured block ...
!$omp end taskgroup
```

- Specifies a wait on completion of child tasks and their descendent tasks
  - →„deeper“ sychronization than `taskwait`, but
  - →with the option to restrict to a subset of all tasks (as opposed to a `barrier`)

User Defined Reductions

User Defined Reductions (UDRs) expand OpenMP's usability

- Use `declare reduction` directive to define operators
- Operators used in reduction clause like predefined ops

```
#pragma omp declare reduction (reduction-identifier :
typename-list : combiner) [initializer(initializer-expr)]
```

- `reduction-identifier` gives a name to the operator
  - → Can be overloaded for different types
  - → Can be redefined in inner scopes
- `typename-list` is a list of types to which it applies
- `combiner` expression specifies how to combine values
- `initializer` specifies the operator's identity value
  - → `initializer-expression` is an expression or a function

A simple UDR example

- Declare the reduction operator

```
#pragma omp declare reduction (mindex : index_struct:
   (omp_in.value < omp_out.value) ? omp_in : omp_out)
   initializer(omp_priv = {.value = MAX_INT, .index = 0})
```

- Use the reduction operator in a `reduction` clause

```
index_struct min_value = (.value = MAX_INT, .index = 0);
#pragma omp parallel for reduction (mindex : min_value)
  for (i = 0; I < NUM_ELEMENTS; i++)
    if ( a[i] < min_value.value) {
      min_value.value = a[i]; min_value.index = i;}
```

- Private copies created for a reduction are initialized to the identity that was specified for the operator and type
  - → Default identity defined if `identity` clause not present
- Compiler uses `combiner` to combine private copies
  - → `omp_out` refers to private copy that holds combined value
  - → `omp_in` refers to the other private copy

Atomics

The atomic construct supports efficient parallel accesses

- Use `atomic` construct for mutually exclusive access to a single memory location

```
#pragma omp atomic [read|write|update] [capture] [compare|weak] [fail|seq_cst]
  expression-stmt
```

- `expression-stmt` restricted based on type of atomic
- `update`, the default behavior, reads and writes the single memory location atomically
- `read` reads location atomically
- `write` writes location atomically
- `capture` updates or writes location and captures its value (before or after update) into a private variable

OpenMP supports several atomic operations

- Early versions did not support atomic capture

```
int schedule (int upper) {
   static int iter = 0; int ret;
   ret = iter;
   #pragma omp atomic
      iter++;
   if (ret <= upper) { return ret; }
   else { return -1; }  //no more iters
}
```

- Atomic capture provides the needed functionality

```
int schedule (int upper) {
   static int iter = 0; int ret;
   #pragma omp atomic capture
      ret = iter++;     // atomic capture
   if (ret <= upper) { return ret; }
   else { return -1; }  // no more iters
}
```

User-level synchronization supported by memory ordering clauses

- Naive attempt to write user-level critical section
  - →Assume `shared_*` are all shared variables
  - →Assume only two threads access `shared_lock`

```
int local, do_not_have_lock = 1;

while (do_not_have_lock) {
   #pragma omp atomic capture
   do_not_have_lock = shared_lock++;
}

local =  shared_a;
shared_a = shared_b;
shared_b = local;

#pragma omp atomic write
shared_lock = 0;
```

- What’s wrong with this code?

User-level synchronization must ensure that memory is consistent

- Correct user-level critical section must include flushes
  - →Assume `shared_*` are all shared variables
  - →Assume only two threads access `shared_lock`

```
int local, do_not_have_lock = 1;

while (do_not_have_lock) {
   #pragma omp atomic capture seq_cst
   do_not_have_lock = shared_lock++;
}

local =  shared_a;
shared_a = shared_b;
shared_b = local;

#pragma omp atomic write seq_cst
shared_lock = 0;
```

- Alternatively, must add several flushes (more than 2)

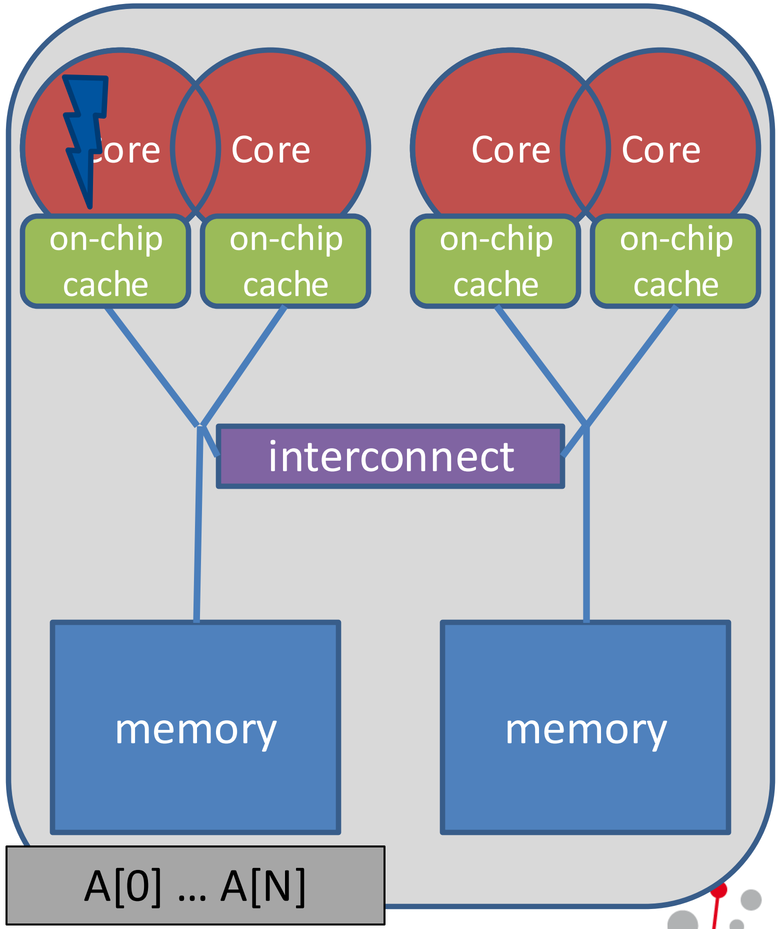
Understanding Memory Access

Memory Affinity

Non-uniform Memory

- Serial code: all array elements are allocated in the memory of the NUMA node closest to the core executing the initializer thread (first touch)

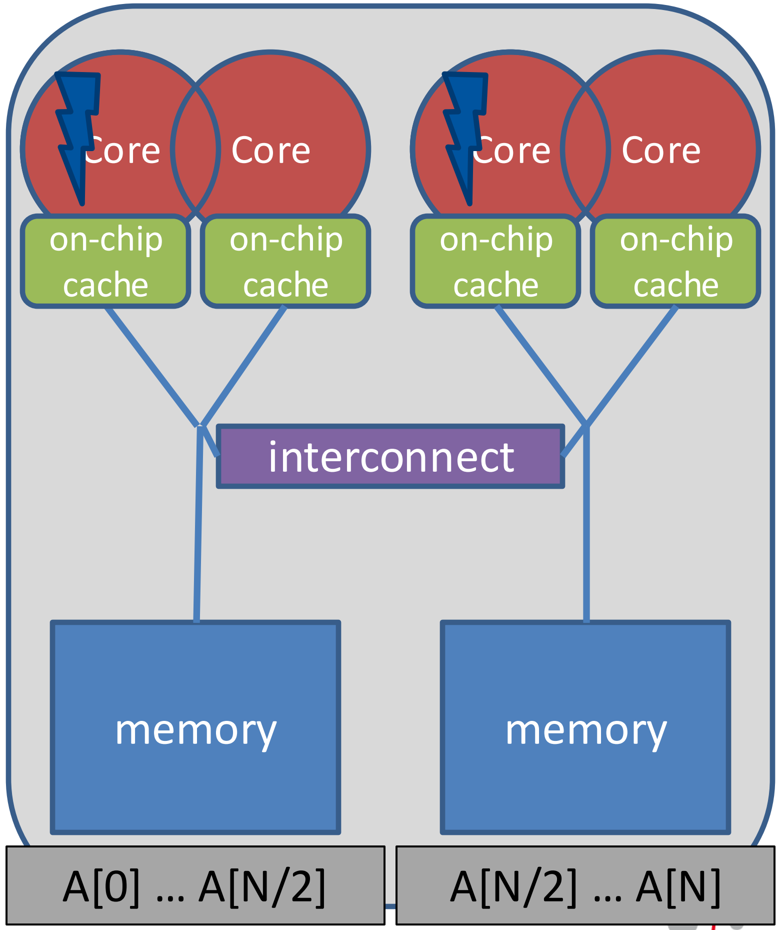
```
double* A;
A = (double*)
    malloc(N * sizeof(double));



for (int i = 0; i < N; i++) {
   A[i] = 0.0;
}
```



First Touch Memory Placement

- First Touch w/ parallel code: all array elements are allocated in the memory of the NUMA node that contains the core that executes the thread that initializes the partition

```
double* A;
A = (double*)
    malloc(N * sizeof(double));

omp_set_num_threads(2);

#pragma omp parallel for proc_bind(spread)
for (int i = 0; i < N; i++) {
   A[i] = 0.0;
}
```

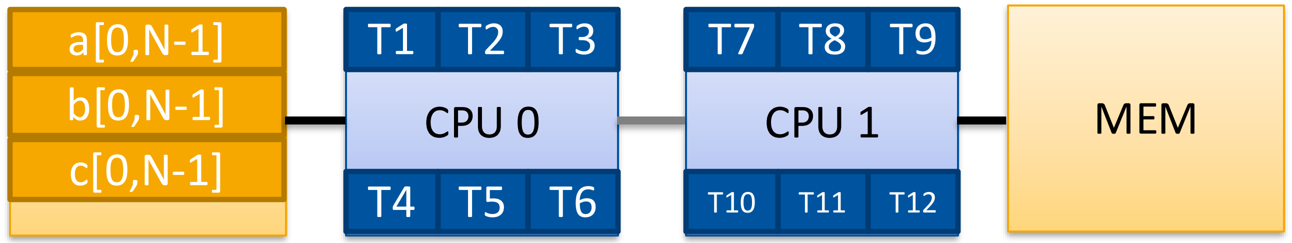


Serial vs. Parallel Initialization

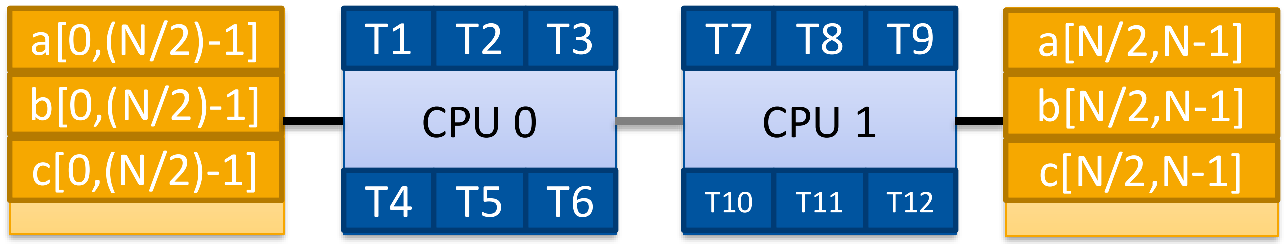
- Stream example with and without parallel initialization.
  - → 2 socket sytem with Xeon X5675 processors, 12 OpenMP threads

	copy	scale	add	triad
ser_init	18.8 GB/s	18.5 GB/s	18.1 GB/s	18.2 GB/s
par_init	41.3 GB/s	39.3 GB/s	40.3 GB/s	40.4 GB/s

ser_init:



par_init:



Thread Binding and Memory Placement

Get Info on the System Topology

- Before you design a strategy for thread binding, you should have a basic understanding of the system topology:
  - → Intel MPI‘s `cpuinfo` tool
    - → `module switch openmpi intelmpi`
    - → `cpuinfo`
    - → Delivers information about the number of sockets (= packages) and the mapping of processor IDs to CPU cores used by the OS
  - → hwlocs‘ `hwloc-ls` tool
    - → `hwloc-ls`
    - → Displays a representation of the system topology, separated into NUMA nodes, along with the mapping of processor IDs to CPU cores used by the OS and additional information on caches

Decide for Binding Strategy

- Selecting the „right“ binding strategy depends not only on the topology, but also on the characteristics of your application.
  - → Putting threads far apart, i.e., on different sockets
    - → May improve the aggregated memory bandwidth available to your application
    - → May improve the combined cache size available to your application
    - → May decrease performance of synchronization constructs
  - → Putting threads close together, i.e., on two adjacent cores that possibly share some caches
    - → May improve performance of synchronization constructs
    - → May decrease the available memory bandwidth and cache size

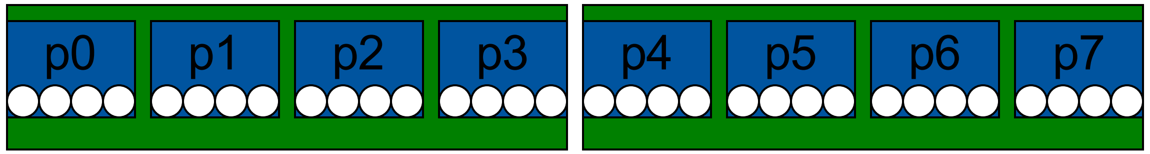
- If you are unsure, just try a few options and then select the best one.

Since OpenMP 4.0: Places + Policies

- Define OpenMP places
  - → set of OpenMP threads running on one or more processors
  - → can be defined by the user, i.e., `OMP_PLACES=cores`
- Define a set of OpenMP thread affinity policies
  - → SPREAD: spread OpenMP threads evenly among the places, partition the place list
  - → CLOSE: pack OpenMP threads near primary thread
  - → PRIMARY: collocate OpenMP thread with primary thread
- Goals
  - → user has a way to specify where to execute OpenMP threads for locality between OpenMP threads / less false sharing / memory bandwidth

OMP_PLACES env. variable

- Assume the following machine:



  → 2 sockets, 4 cores per socket, 4 hyper-threads per core

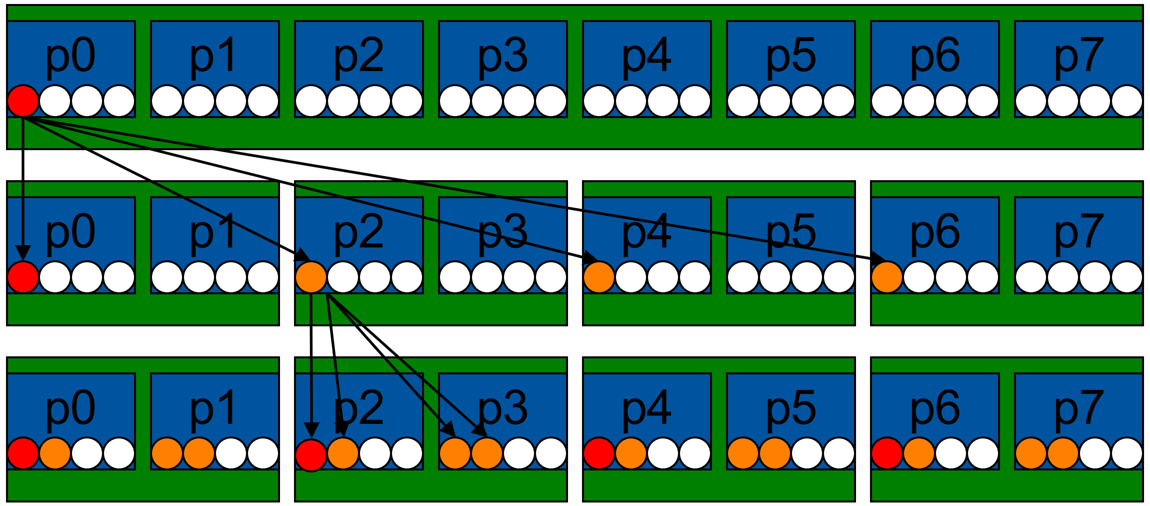
- Abstract names for OMP_PLACES:
  → threads: Each place corresponds to a single hardware thread.
  → cores: Each place corresponds to a single core (having one or more hardware threads).
  → sockets: Each place corresponds to a single socket (consisting of one or more cores).
  → ll_caches (5.1): Each place corresponds to a set of cores that share the last level cache.
  → numa_domains (5.1): Each places corresponds to a set of cores for which their closest memory is: the same memory; and at a similar distance from the cores.

OpenMP 4.0: Places + Policies

- Example‘s Objective:
  - → separate cores for outer loop and near cores for inner loop
- Outer Parallel Region: proc_bind(spread), Inner: proc_bind(close)
  - → spread creates partition, compact binds threads within respective partition

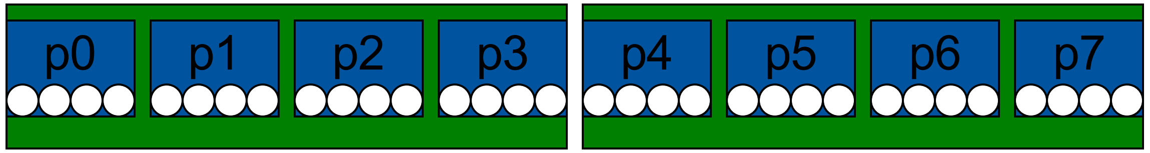
```
OMP_PLACES=(0,1,2,3), (4,5,6,7), ... = (0-4):4:8    = cores
#pragma omp parallel proc_bind(spread) num_threads(4)
#pragma omp parallel proc_bind(close) num_threads(4)
```

- Example
  - → initial
  - → spread 4
  - → close 4



More Examples (1/3)

- Assume the following machine:



  → 2 sockets, 4 cores per socket, 4 hyper-threads per core

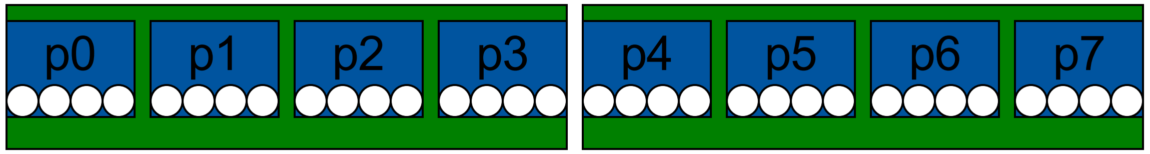
- Parallel Region with two threads, one per socket

  → `OMP_PLACES=sockets`

  → `#pragma omp parallel num_threads(2) proc_bind(spread)`

More Examples (2/3)

- Assume the following machine:



  → 2 sockets, 4 cores per socket, 4 hyper-threads per core

- Parallel Region with four threads, one per core, but only on the first socket

  → `OMP_PLACES=cores`

  → `#pragma omp parallel num_threads(4) proc_bind(close)`

More Examples (3/3)

- Spread a nested loop first across two sockets, then among the cores within each socket, only one thread per core
  - → `OMP_PLACES=cores`
  - → `#pragma omp parallel num_threads(2) proc_bind(spread)`
  - → `#pragma omp parallel num_threads(4) proc_bind(close)`

- Places API routines allow to
  - → query information about binding…
  - → query information about the place partition…

Places API: Example

- Simple routine printing the processor ids of the place the calling thread is bound to:

```
void print_binding_info() {
     int my_place = omp_get_place_num();
     int place_num_procs = omp_get_place_num_procs(my_place);

     printf("Place consists of %d processors: ", place_num_procs);

     int *place_processors = malloc(sizeof(int) * place_num_procs);
     omp_get_place_proc_ids(my_place, place_processors)

     for (int i = 0; i < place_num_procs - 1; i++) {
             printf("%d ", place_processors[i]);
     }
     printf("\n");

     free(place_processors);
}
```

OpenMP 5.x way to do this

- Set `OMP_DISPLAY_AFFINITY=TRUE`
  - →Instructs the runtime to display formatted affinity information
  - →Example output for two threads on two physical cores:

```
nesting_level=   1,    thread_num=   0,    thread_affinity=   0,1
nesting_level=   1,    thread_num=   1,    thread_affinity=   2,3
```

  - →Output c
    correspon routine
  - →Formatted affinity information can be printed with
    `omp_display_affinity(const char* format)`

Affinity format specification

t	omp_get_team_num()
T	omp_get_num_teams()
L	omp_get_level()
n	omp_get_thread_num()
N	omp_get_num_threads()

a	omp_get_ancestor_thread_num() at level-1
H	hostname
P	process identifier
i	native thread identifier
A	thread affinity: list of processors (cores)

- Example:

```
OMP_AFFINITY_FORMAT="Affinity: %0.3L %.8n %.15{A} %.12H"
```

→Possible output:

```
Affinity: 001        0       0-1,16-17      host003
Affinity: 001        1       2-3,18-19      host003
```

Fine-grained control of Memory Affinity

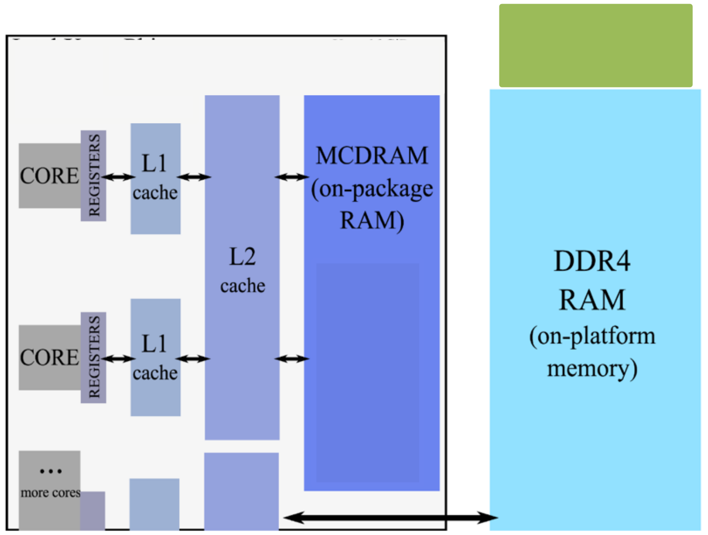
- Explicit NUMA-aware memory allocation:
  - →By carefully touching data by the thread which later uses it
  - →By changing the default memory allocation strategy
    - →Linux: `numactl` command
  - →By explicit migration of memory pages
    - →Linux: `move_pages()`

- Example: using numactl to distribute pages round-robin:
  - →`numactl –interleave=all ./a.out`

Memory Management

Different kinds of memory

- Traditional DDR-based memory
- High-bandwidth memory
- Non-volatile memory
- …



Cascade Lake (Leonide at INRIA)

```
CPU: Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz
Freq Govenor: performance
---------------------
available: 4 nodes (0-3)
node 0 cpus: 0 2 4 6 8 10 12 14 16 18
             20 22 24 26 28 30 32 34 36 38
node 0 size: 191936 MB
node 0 free: 178709 MB
node 1 cpus: 1 3 5 7 9 11 13 15 17 19 21 23
             25 27 29 31 33 35 37 39
node 1 size: 192016 MB
node 1 free: 179268 MB
node 2 cpus:
node 2 size: 759808 MB
node 2 free: 759794 MB
node 3 cpus:
node 3 size: 761856 MB
node 3 free: 761851 MB
node distances:
node   0   1   2   3
  0:  10  21  17  28
  1:  21  10  28  17
  2:  17  28  10  28
  3:  28  17  28  10
```

DRAM + Optane

Memory Management

- Allocator := an OpenMP object that fulfills requests to allocate and deallocate storage for program variables

- OpenMP allocators are of type `omp_allocator_handle_t`

- Default allocator for host
  - →via `OMP_ALLOCATOR` env. var. or corresponding API

- OpenMP 5.0 supports
  a set of memory allocators

OpenMP allocators

- Selection of a certain kind of memory

Allocator name	Storage selection intent
omp_default_mem_alloc	use default storage
omp_large_cap_mem_alloc	use storage with large capacity
omp_const_mem_alloc	use storage optimized for read-only variables
omp_high_bw_mem_alloc	use storage with high bandwidth
omp_low_lat_mem_alloc	use storage with low latency
omp_cgroup_mem_alloc	use storage close to all threads in the contention group of the thread requesting the allocation
omp_pteam_mem_alloc	use storage that is close to all threads in the same parallel region of the thread requesting the allocation
omp_thread_local_mem_alloc	use storage that is close to the thread requesting the allocation

Using OpenMP allocators

- New clause on all constructs with data sharing clauses:
  - → `allocate( [allocator:] list )`
- Allocation:
  - → `omp_alloc(size_t size, omp_allocator_handle_t allocator)`
- Deallocation:
  - → `omp_free(void *ptr, const omp_allocator_handle_t allocator)`

- `allocate` directive: standalone directive for allocation, or declaration of allocation stmt.

OpenMP Allocator Traits / 1

sync_hint	contended, uncontended, serialized, private	default: contended
alignment	positive integer value that is a power of two	default: 1 byte
access	all, memspace, device, cgroup, pteam, thread	default: memspace
pool_size	positive integer value	
fallback	default_mem_fb, null_fb, abort_fb, allocator_fb	default: default_mem_fb
fb_data	an allocator handle	
pinned	true, false	default: false
partition	environment, nearest, blocked, interleaved	default: environment
pin_device	conforming device number	
preferred_device	conforming device number	
target access	single, multiple	default: single
atomic_scope	all, device	default: device
part_size	positive integer value	
partitioner	a memory partitioner handle	
partitioner_arg	an integer value	0

OpenMP Allocator Traits / 2

- `fallback`: describes the behavior if the allocation cannot be fulfilled
  - → `default_mem_fb`: return system's default memory
  - → Other options: null, abort, or use different allocator

- `pinned`: request pinned memory, i.e. for GPUs,
  - → device may be specified

OpenMP Allocator Traits / 3

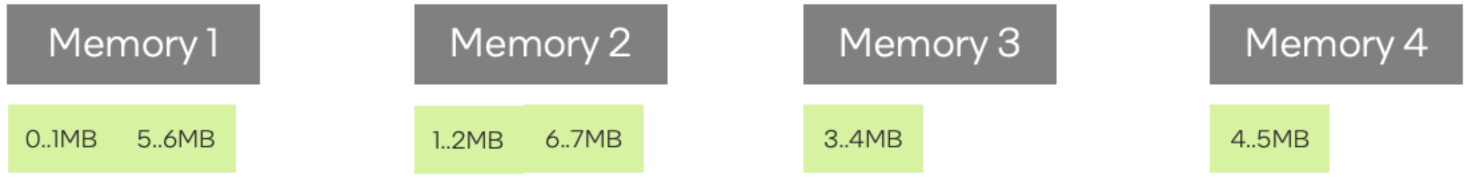
- `partition`: partitioning of allocated memory of physical storage resources (think of NUMA)
  - → `environment`: use system's default behavior
  - → `nearest`: most closest memory
  - → `blocked`: partitioning into approx. same size with at most one block per storage resource
  - → `interleaved`: partitioning in a round-robin fashion across the storage resources, in which `part_size` specifies the size of individual partitions
  - → `partitioner`: definition of memory parts and distribution across storage are defined by a memory partitioner

OpenMP Allocator Traits / 4

- Example code:

```
const omp_alloctrait_t traits[] ={{omp_atk_partition,
                                    omp_atv_interleaved},
                                 {omp_atk_part_size, 1024*1024} };

omp_allocator_handle_t numa_dev_alloc =
           omp_init_allocator(omp_default_mem_space, 2, traits);
int * a = omp_alloc(numa_dev_alloc, 6*1024*1024);
```

→Distributes chunks of memory:



Example created by Alex Duran (Intel)

OpenMP Allocator Traits / 5

sync_hint	contended, uncontended, serialized, private	default: contended
alignment	positive integer value that is a power of two	default: 1 byte
access	all, memspace, device, cgroup, pteam, thread	default: memspace
pool_size	positive integer value	
fallback	default_mem_fb, null_fb, abort_fb, allocator_fb	default: default_mem_fb
fb_data	an allocator handle	
pinned	true, false	default: false
partition	environment, nearest, blocked, interleaved	default: environment
pin_device	conforming device number	
preferred_device	conforming device number	
target access	single, multiple	default: single
atomic_scope	all, device	default: device
part_size	positive integer value	
partitioner	a memory partitioner handle	
partitioner_arg	an integer value	0

OpenMP Allocator Traits / 6

- `partition`: partitioning of allocated memory of physical storage resources (think of NUMA)
  - → `environment`: use system's default behavior
  - → `nearest`: most closest memory
  - → `blocked`: partitioning into approx. same size with at most one block per storage resource
  - → `interleaved`: partitioning in a round-robin fashion across the storage resources, in which `part_size` specifies the size of individual partitions
  - → `partitioner`: definition of memory parts and distribution across storage are defined by a memory partitioner

OpenMP Memory Partitioner

- Memory Partitioner := an OpenMP object that represents mechanisms to create and destroy memory partitions
  - →Memory Partition := a definition how an allocator divides memory into parts
    - →Memory Part := a storage block in a single storage resource within a memory space

- `omp_init_mempartitioner` routine: initializes a partitioner that …
  - →… can be used with an OpenMP allocator
  - →... takes the argument `compute_proc` to determine the number of memory parts and their distribution across the storage resources
  - →+ further management and cleanup routines

- Memory Space Retrieving Routines: return memory space handles

Using OpenMP allocator traits

- Construction of allocators with traits via
  - →`omp_allocator_handle_t   omp_init_allocator(`
    `omp_memspace_handle_t memspace,`
    `int ntraits, const omp_alloctrait_t traits[]);`
  - →Selection of memory space mandatory
  - →Empty traits set: use defaults

- Allocators have to be destroyed with `*_destroy_*`

- Custom allocator can be made default with
  `omp_set_default_allocator(omp_allocator_handle_t allocator)`

Memory Management Status (status: 11/2024)

- **LLVM OpenMP runtime internally already uses libmemkind (libnuma, numactl)**
  - → Support for various kinds of memory: DDR, HBW and Persistent Memory (Optane)
  - → Library loaded at initialization (checks for availability)
  - → If requested memory space for allocator is not available → fallback to DDR

- **Memory Management implementation in LLVM still not complete**
  - → Some allocator traits not implemented yet
  - → Some `partition` values not implemented yet (environment, interleaved, nearest, blocked)
  - → Semantics of `omp_high_bw_mem_space and omp_large_cap_mem_space` unclear. Which memory should be used?
    - → Explicitly target HBM → currently implemented in LLVM

- **LLVM has custom implementation of aligned memory allocation**
  - → Allocation covers → {Allocator Information + Requested Size + Buffer based on alignment}

Loop Transformations

Loop Unrolling

- Loop unrolling is a standard tuning practice to reduce loop overhead and increase potential for pipeline.

```
subroutine loop()
    do i = 1, 4
        call body(i)
    end do
end subroutine loop
```

$\downarrow$

```
subroutine loop()
    call body(i + 0)
    call body(i + 1)
    call body(i + 2)
    call body(i + 3)
end subroutine loop
```

```
subroutine loop()
    !$omp unroll full
    do i = 1, 4
        call body(i)
    end do
end subroutine loop
```

- "`full`" completely unrolls the loop
  - → Needs a compile-time constant upper bound.
  - → Loop is no longer present after unrolling took place.

Loop Unrolling

- Loop unrolling is a standard tuning practice to reduce loop overhead and increase potential for pipeline.

```
subroutine loop()
    do i = 1, n
        call body(i)
    end do
end subroutine loop
```

↓

```
subroutine loop()
    do i = 1, n, 4
        call body(i + 0)
        call body(i + 1)
        call body(i + 2)
        call body(i + 3)
    end do
end subroutine loop
```

```
subroutine loop()
    !$omp unroll partial(4)
    do i = 1, n
        call body(i)
    end do
end subroutine loop
```

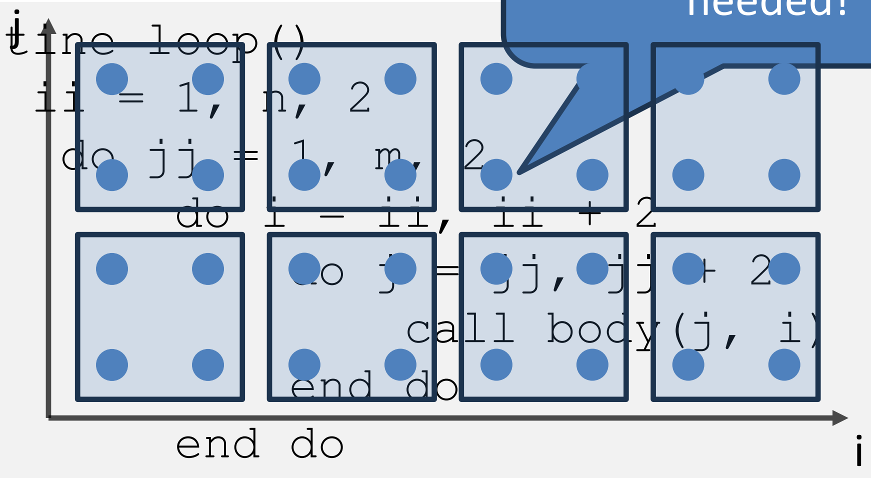
- "`partial(`*`f`*`)`" unrolls the loop with unroll factor *f*
  - → Upper bound can now be a runtime value
  - → Compiler will introduce remainder loops as necessary

Tiling

- Tiling is a useful to optimize a loop nest for the cache hierarchy and exploiting temporal/spatial locality

```
subroutine loop()
    !$omp tile sizes(2,2)
    do i = 1, n
        do j = 1, m
            call body(j, i)
        end do
    end do
end subroutine loop
```

```
subroutine loop()
    do ii = 1, n, 2
        do jj = 1, m, 2
            do i = ii, ii + 2
                do j = jj, jj + 2
                    call body(j, i)
                end do
            end do
        end do
    end do
end subroutine loop
```



Handling of partial tiles needed!

Tiling

- Tiling is a useful to optimize a loop nest for the cache hierarchy and exploiting temporal/spatial locality

```
subroutine loop()
    !$omp tile sizes(2,2)
    do i = 1, n
        do j = 1, m
            call body(j, i)
        end do
    end do
end subroutine loop
```

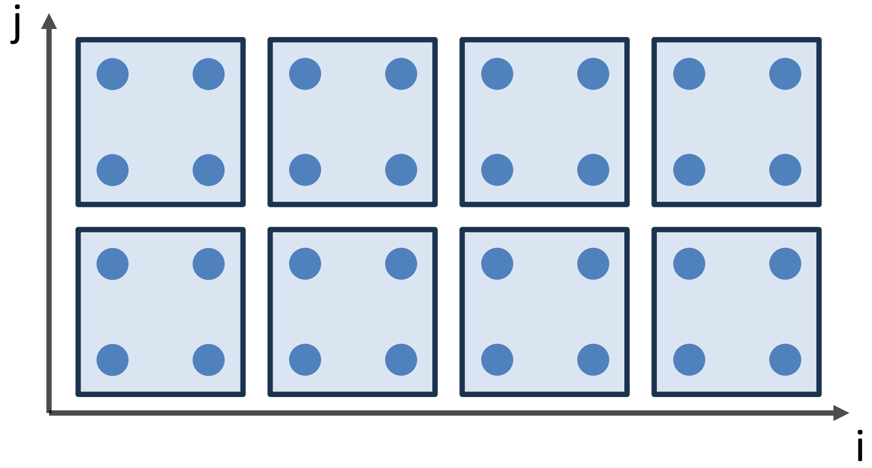
```
subroutine loop()
    do ii = 1, n, 2
        do jj = 1, m, 2
            do i = ii, ii + 2
                do j = jj, jj + 2
                    call body(j, i)
                end do
            end do
        end do
    end do
end subroutine loop
```

Handling of partial tiles needed!

Tiling

- Tiling is a useful to optimize a loop nest for the cache hierarchy and exploiting temporal/spatial locality

```
subroutine loop()
    !$omp tile sizes(2,2)
    do i = 1, n
        do j = 1, m
            call body(j, i)
        end do
    end do
end subroutine loop
```

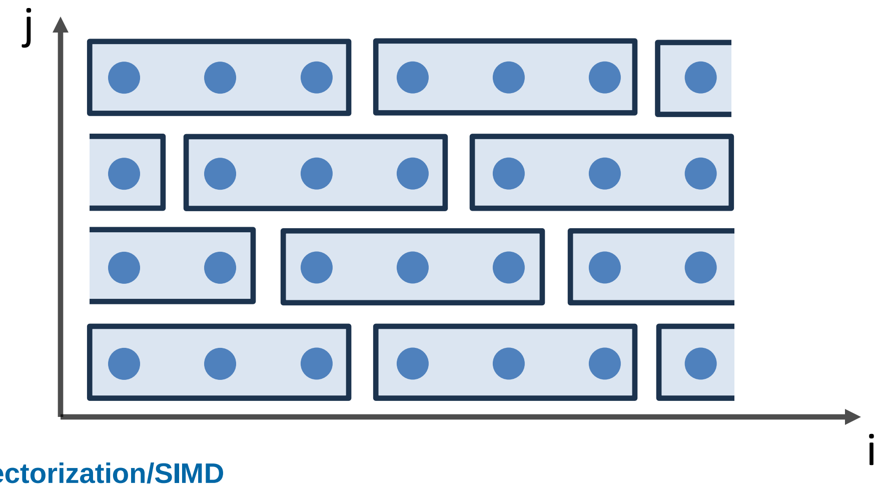


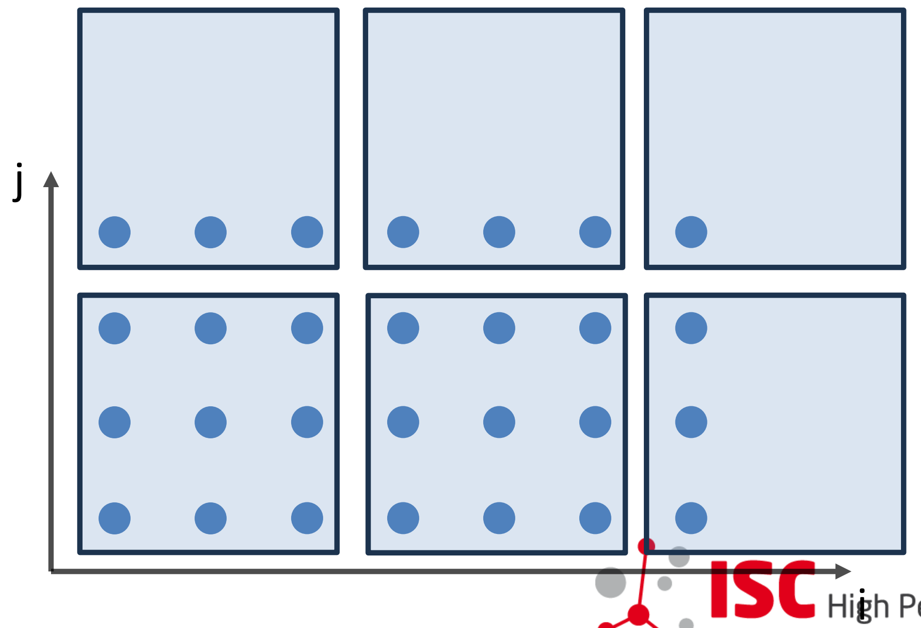
Tiling and Chunking

- One can think of tiling as “multi-dimensional” chunking:

```
!$omp for schedule(static, 3) &
             collapse(2)
do i = 1, n
     do j = 1, m
          call body(j, i)
     end do
end do
```

```
!$omp tile sizes(3,3)
do i = 1, n
     do j = 1, m
          call body(j, i)
     end do
end do
```





Other Loop Transformations /1

- Loop Interchange

```
!$omp interchange permutation(3,1,2)
do i = 1, n
    do j = 1, m
        do k = 1, l
            call body(j, i, k)
        end do
    end do
end do
```

→

```
do k = 1, l
    do i = 1, n
        do j = 1, m
            call body(j, i, k)
        end do
    end do
end do
```

- Loop Reversal

```
!$omp reverse
do i = 1, n
    call body(i)
end do
```

→

```
do i = 1, n
    call body(n - (i - 1))
end do
```

Other Loop Transformations /2

- Loop Fusion

```
!$omp fuse
do i = 1, n
    call body1(i)
end do
do i = 1, n
    call body2)(i)
end do
!$omp end fuse
```

$\rightarrow$

```
do i = 1, n
    call body1(i)
    call body2(i)
end do
```

- Loop Reversal

```
!$omp reverse
do i = 1, n
    call body(i)
end do
```

$\rightarrow$

```
do i = 1, n
    call body(n - (i - 1))
end do
```

Other Loop Transformations /3

- Loop Index Splitting

```
!$omp split counts(k, omp_fill)
do i = 1, n
    call body(i)
end do
```

→

```
do i = 1, k
    call body(i)
end do
do i = k, n
    call body(i)
end do
```

- All these transformations can be useful:
  - →Fusion: reduce loop overhead and get more work per loop iteration
  - →Reversal: create forward memory references
  - →Index splitting: peel off loop iterations, e.g., for better SIMD/memory alignment

Composing Loop Transformations

- Loop transformations can be composed, e.g., tiling and unrolling:

```
!$omp tile sizes(2,2)  &
      apply(intratile:unroll full, &
                      unroll full)
do i = 1, n
    do j = 1, m
        call body(j, i)
    end do
end do
```

```
do ii = 1, n, 2
    do jj = 1, m, 2
        i = ii; j = jj
        call body(j + 0, i + 0)
        call body(j + 1, i + 0)
        call body(j + 0, i + 1)
        call body(j + 1, i + 1)
    end do
end do
```

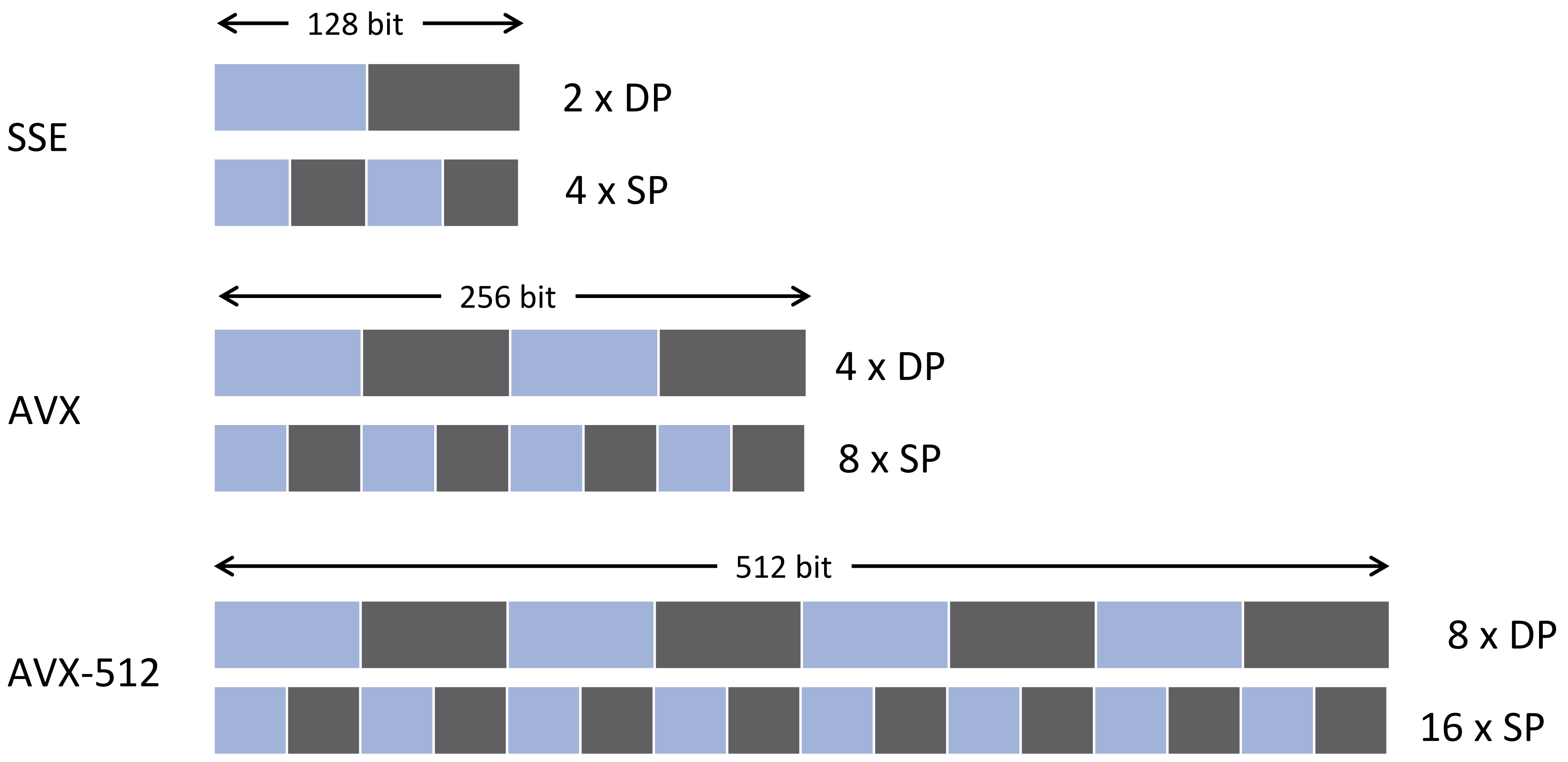
Vectorization

Topics

- Exploiting SIMD parallelism with OpenMP
- Using SIMD directives with loops
- Creating SIMD functions

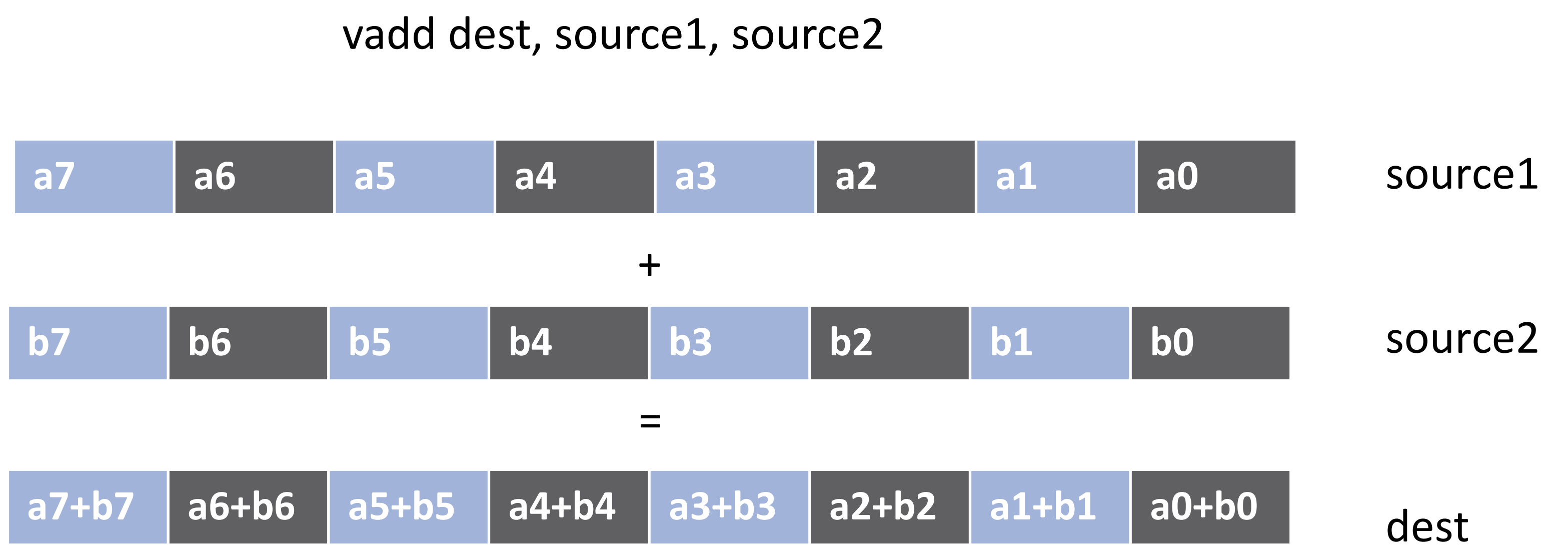
SIMD on x86 Architectures

- Width of SIMD registers has been growing in the past:



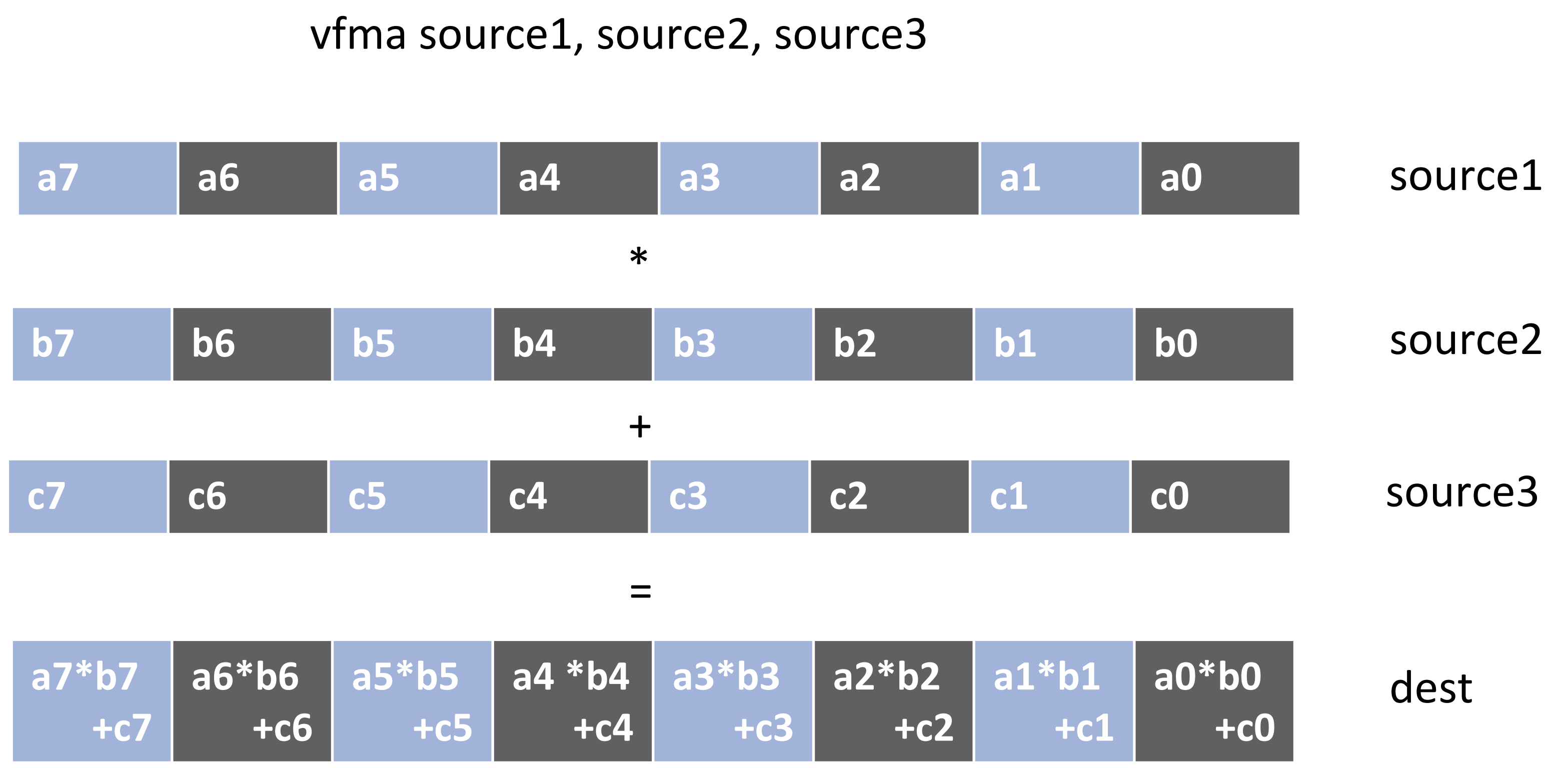
More Powerful SIMD Units

- SIMD instructions become more powerful



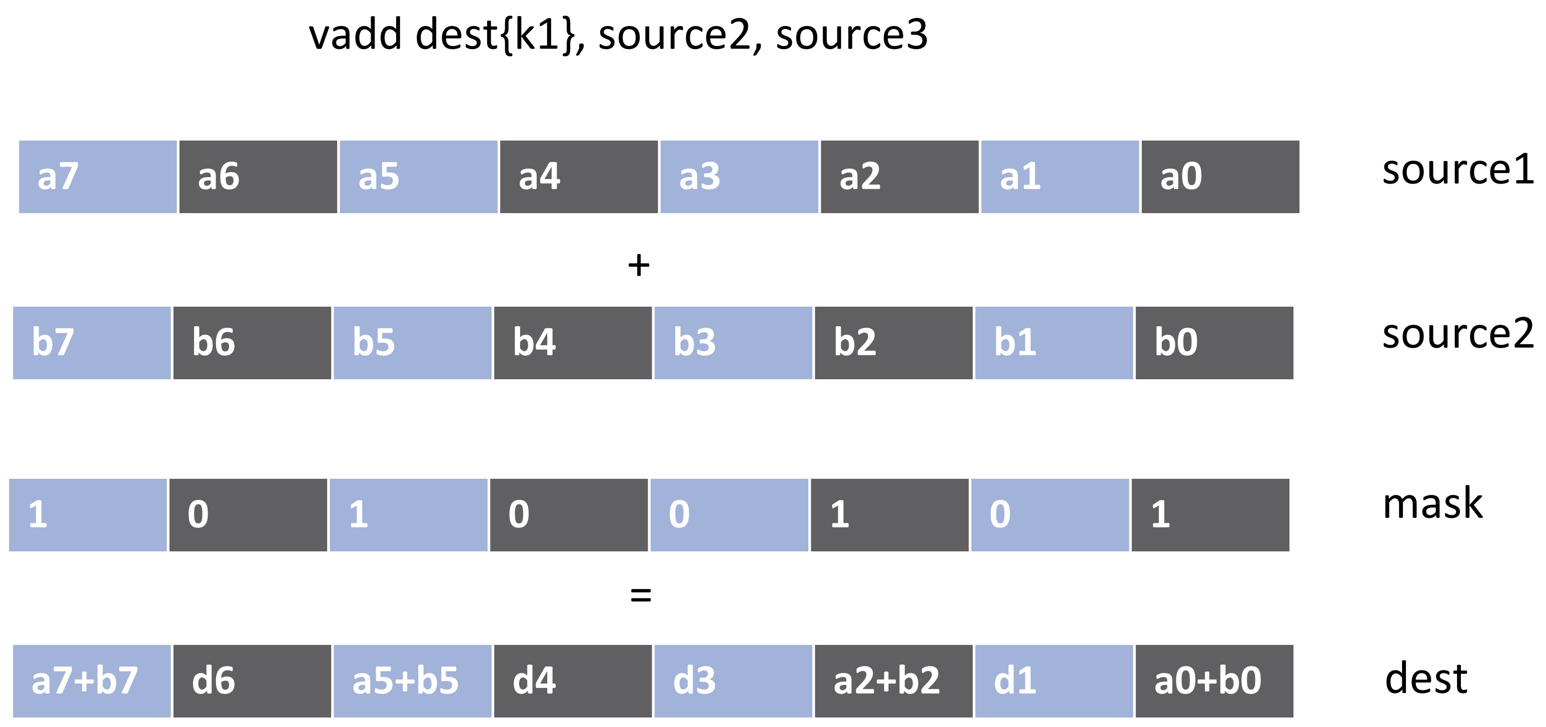
More Powerful SIMD Units

- SIMD instructions become more powerful



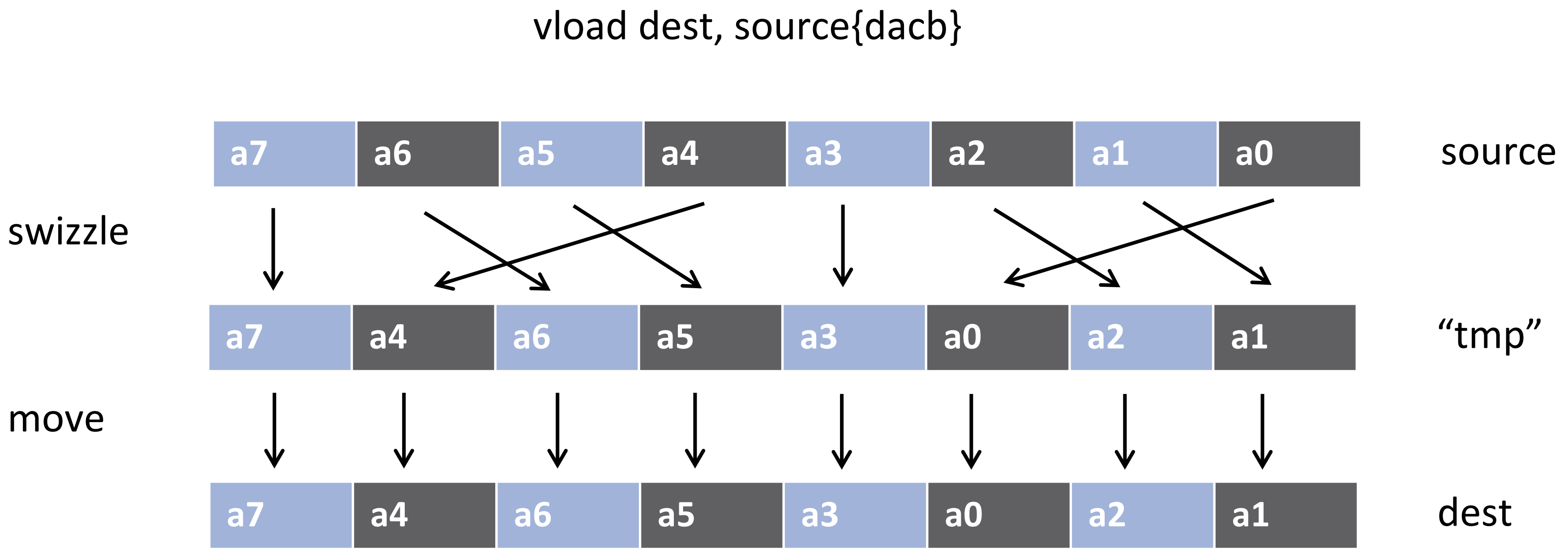
More Powerful SIMD Units

- SIMD instructions become more powerful



More Powerful SIMD Units

- SIMD instructions become more powerful



Auto-vectorization

- Compilers offer auto-vectorization as an optimization pass
  - →Usually, part of the general loop optimization passes
  - →Code analysis detects code properties that inhibit SIMD vectorization ?
  - →Heuristics determine if SIMD execution might be beneficial
  - →If all goes well, the compiler will generate SIMD instructions

- Example:

clang/LLVM	GCC	Intel Compiler
→-fvectorize	-ftree-vectorize	-vec (enabled w/ -O2)
→-Rpass=loop-.*	-ftree-loop-vectorize	-qopt-report=vec
→-mprefer-vector-width=<*width*>	-fopt-info-vec-all	

Why Auto-vectorizers Fail

- Data dependencies
- Other potential reasons
  - →Alignment
  - →Function calls in loop block
  - →Complex control flow / conditional branches
  - →Loop not "countable"
    - →e.g., upper bound not a runtime constant
  - →Mixed data types
  - →Non-unit stride between elements
  - →Loop body too complex (register pressure)
  - →Vectorization seems inefficient
- Many more … but less likely to occur

Data Dependencies

- Suppose two statements S1 and S2
- S2 depends on S1, iff S1 must execute before S2
  - →Control-flow dependence
  - →Data dependence
  - →Dependencies can be carried over between loop iterations
- Important flavors of data dependencies

FLOW

```
s1: a = 40

    b = 21
s2: c = a + 2
```

ANTI

```
    b = 40

s1: a = b + 1
s2: b = 21
```

Loop-Carried Dependencies

- Dependencies may occur across loop iterations
  →Loop-carried dependency
- The following code contains such a dependency:

```
void lcd_ex(float* a, float* b, size_t n, float c1, float c2)
{
    size_t i;
    for (i = 0; i < n; i++) {
        a[i] = c1 * a[i + 17] + c2 * b[i];
    }
}
```

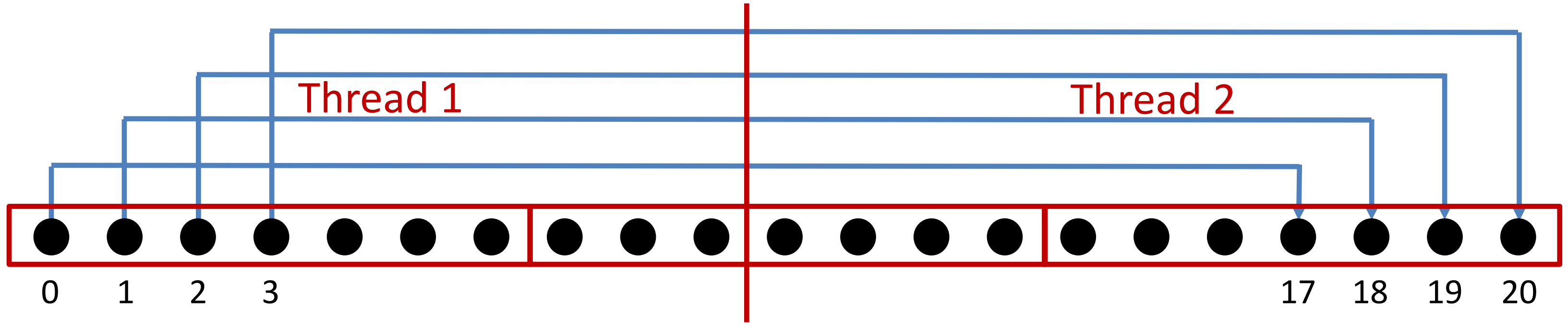
Loop-carried dependency for a[i] and a[i+17]; distance is 17.

- Some iterations of the loop have to complete before the next iteration can run
  →Simple trick: Can you reverse the loop w/o getting wrong results?

Loop-carried Dependencies

- Can we parallelize or vectorize the loop?

```
void lcd_ex(float* a, float* b, size_t n, float c1, float c2) {
    for (int i = 0; i < n; i++) {
        a[i] = c1 * a[i + 17] + c2 * b[i];
}   }
```



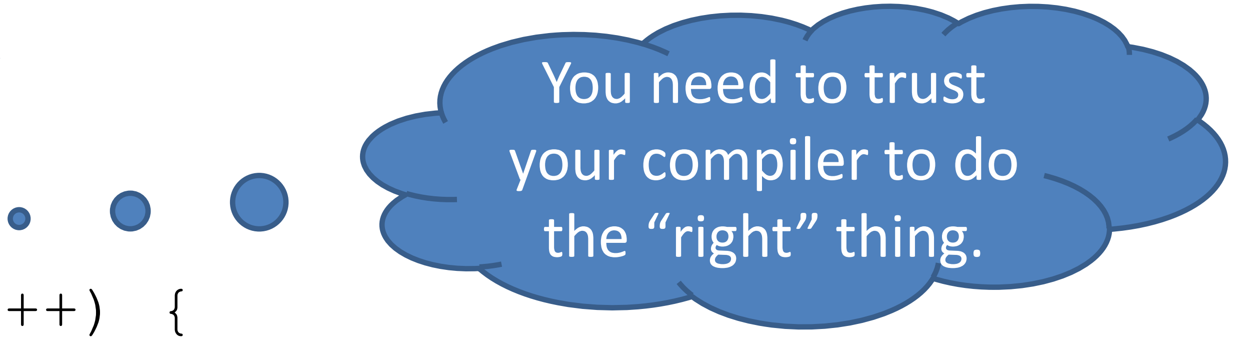
→ Parallelization: no
(except for very specific loop schedules)

→ Vectorization: yes
(iff vector length is shorter than any distance of any dependency)

In a Time Before OpenMP 4.0

- Support required vendor-specific extensions
  - →Programming models (e.g., Intel® Cilk Plus)
  - →Compiler pragmas (e.g., `#pragma vector`)
  - →Low-level constructs (e.g., `_mm_add_pd()`)

```
#pragma omp parallel for
#pragma vector always
#pragma ivdep
for (int i = 0; i < N; i++) {
    a[i] = b[i] + ...;
}
```



SIMD Loop Construct

- Vectorize a loop nest
  - →Cut loop into chunks that fit a SIMD vector register
  - →No parallelization of the loop body

- Syntax (C/C++)

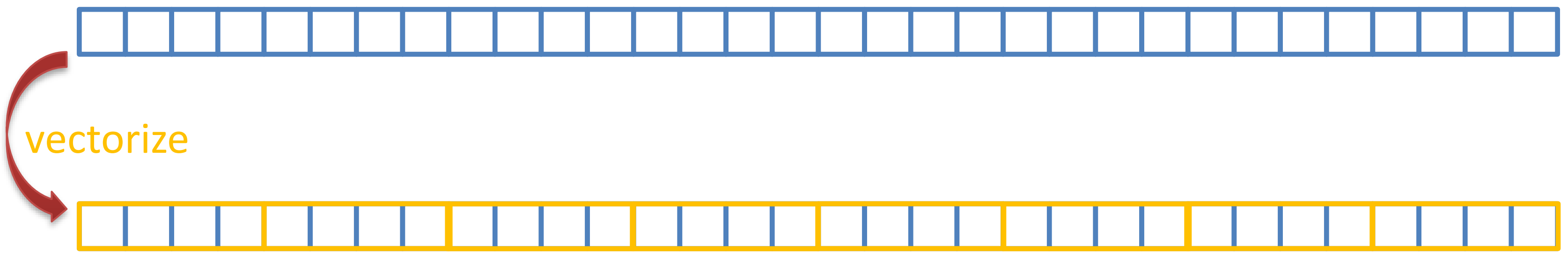
```
#pragma omp simd [clause[[,] clause],…]
for-loops
```

- Syntax (Fortran)

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

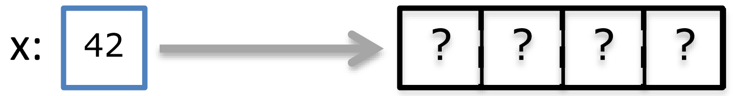
Example

```
float sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp simd reduction(+:sum)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```

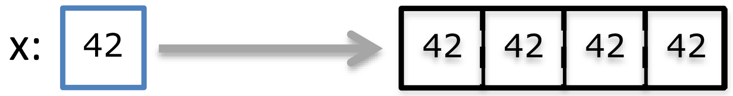


Data Sharing Clauses

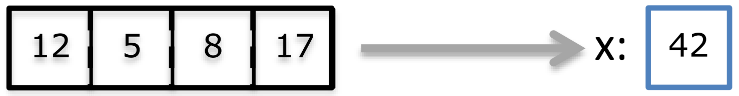
- `private(var-list)`:
  Uninitialized vectors for variables in *var-list*



- `firstprivate(var-list)`:
  Initialized vectors for variables in *var-list*



- `reduction(op:var-list)`:
  Create private variables for *var-list* and apply reduction operator *op* at the end of the construct



SIMD Loop Clauses

- `safelen (`*`length`*`)`
  - →Maximum number of iterations that can run concurrently without breaking a dependence
  - →In practice, maximum vector length
- `linear (`*`list`*`[:`*`linear-step`*`])`
  - →The variable's value is in relationship with the iteration number
    - →$x_i = x_{orig} + i * \text{linear-step}$
- `aligned (`*`list`*`[:`*`alignment`*`])`
  - →Specifies that the list items have a given alignment
  - →Default is alignment for the architecture
- `collapse (`*`n`*`)`

SIMD Worksharing Construct

- Parallelize and vectorize a loop nest
  - →Distribute a loop's iteration space across a thread team
  - →Subdivide loop chunks to fit a SIMD vector register

- Syntax (C/C++)

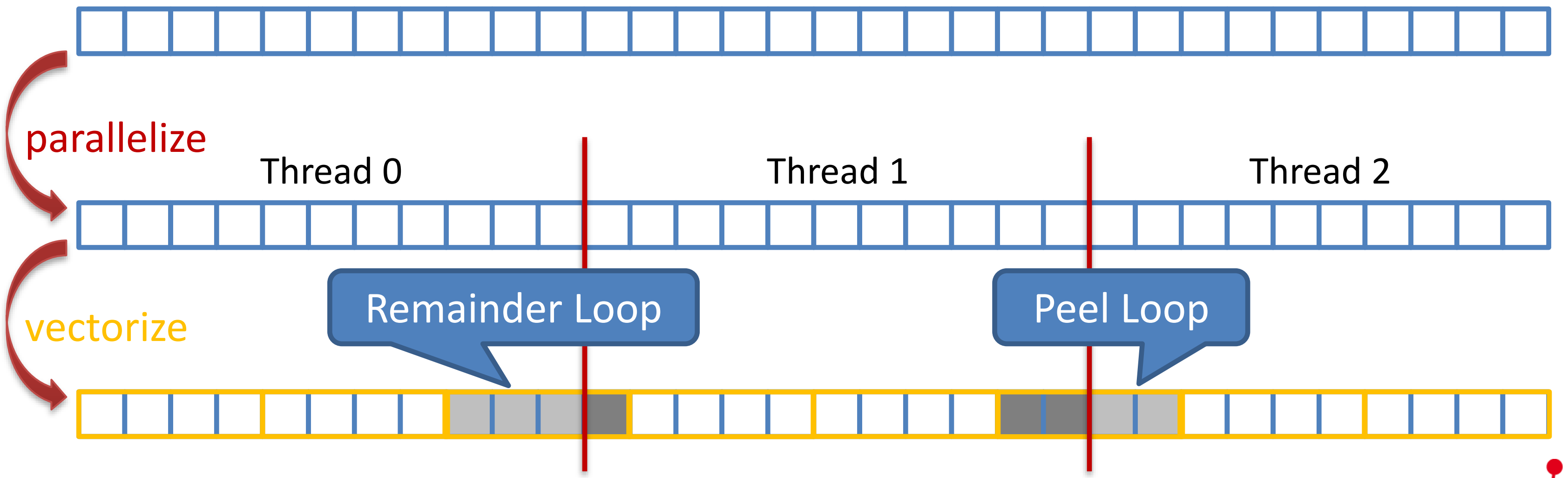
```
#pragma omp for simd [clause[[,] clause],…]
for-loops
```

- Syntax (Fortran)

```
!$omp do simd [clause[[,] clause],…]
do-loops
[!$omp end do simd [nowait]]
```

Example

```
float sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp for simd reduction(+:sum)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```



Be Careful What You Wish For…

```
float sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp for simd reduction(+:sum) \
                     schedule(static, 5)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```

- You should choose chunk sizes that are multiples of the SIMD length
  - → Remainder loops are not triggered
  - → Likely better performance
- In the above example …
  - → and AVX2, the code will only execute the remainder loop!
  - → and SSE, the code will have one iteration in the SIMD loop plus one in the remainder loop!

OpenMP 4.5 Simplifies SIMD Chunks

```
float sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp for simd reduction(+:sum) \
                     schedule(simd: static, 5)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```

- Chooses chunk sizes that are multiples of the SIMD length
  - →First and last chunk may be slightly different to fix alignment and to handle loops that are not exact multiples of SIMD width
  - →Remainder loops are not triggered
  - →Likely better performance

SIMD Function Vectorization

```
float min(float a, float b) {
    return a < b ? a : b;
}


float distsq(float x, float y) {
    return (x - y) * (x - y);
}

void example() {
#pragma omp parallel for simd
    for (i=0; i<N; i++) {
        d[i] = min(distsq(a[i], b[i]), c[i]);
}   }
```

SIMD Function Vectorization

- Declare one or more functions to be compiled for calls from a SIMD-parallel loop

- Syntax (C/C++):

```
#pragma omp declare simd [clause[[,] clause],…]
[#pragma omp declare simd [clause[[,] clause],…]]
[…]
function-definition-or-declaration
```

- Syntax (Fortran):

```
!$omp declare simd (proc-name-list)
```

SIMD Function Vectorization

```
#pragma omp declare simd
float min(float a, float b) {
    return a < b ? a : b;
}

#pragma omp declare simd
float distsq(float x, float y) {
    return (x - y) * (x - y);
}

void example() {
#pragma omp parallel for simd
    for (i=0; i<N; i++) {
        d[i] = min(distsq(a[i], b[i]), c[i]);
}   }
```

```
_ZGVZN16vv_min(%zmm0, %zmm1):
    vminps %zmm1, %zmm0, %zmm0
    ret
```

```
_ZGVZN16vv_distsq(%zmm0, %zmm1):
    vsubps %zmm0, %zmm1, %zmm2
    vmulps %zmm2, %zmm2, %zmm0
    ret
```

```
vmovups (%r14,%r12,4), %zmm0
vmovups (%r13,%r12,4), %zmm1
call _ZGVZN16vv_distsq
vmovups (%rbx,%r12,4), %zmm1
call _ZGVZN16vv_min
```

SIMD Function Vectorization

- `simdlen (`*`length`*`)`
  - → generate function to support a given vector length
- `uniform (`*`argument-list`*`)`
  - → argument has a constant value between the iterations of a given loop
- `inbranch`
  - → function always called from inside an if statement
- `notinbranch`
  - → function never called from inside an if statement
- `linear (`*`argument-list[:linear-step]`*`)`
- `aligned (`*`argument-list[:alignment]`*`)`

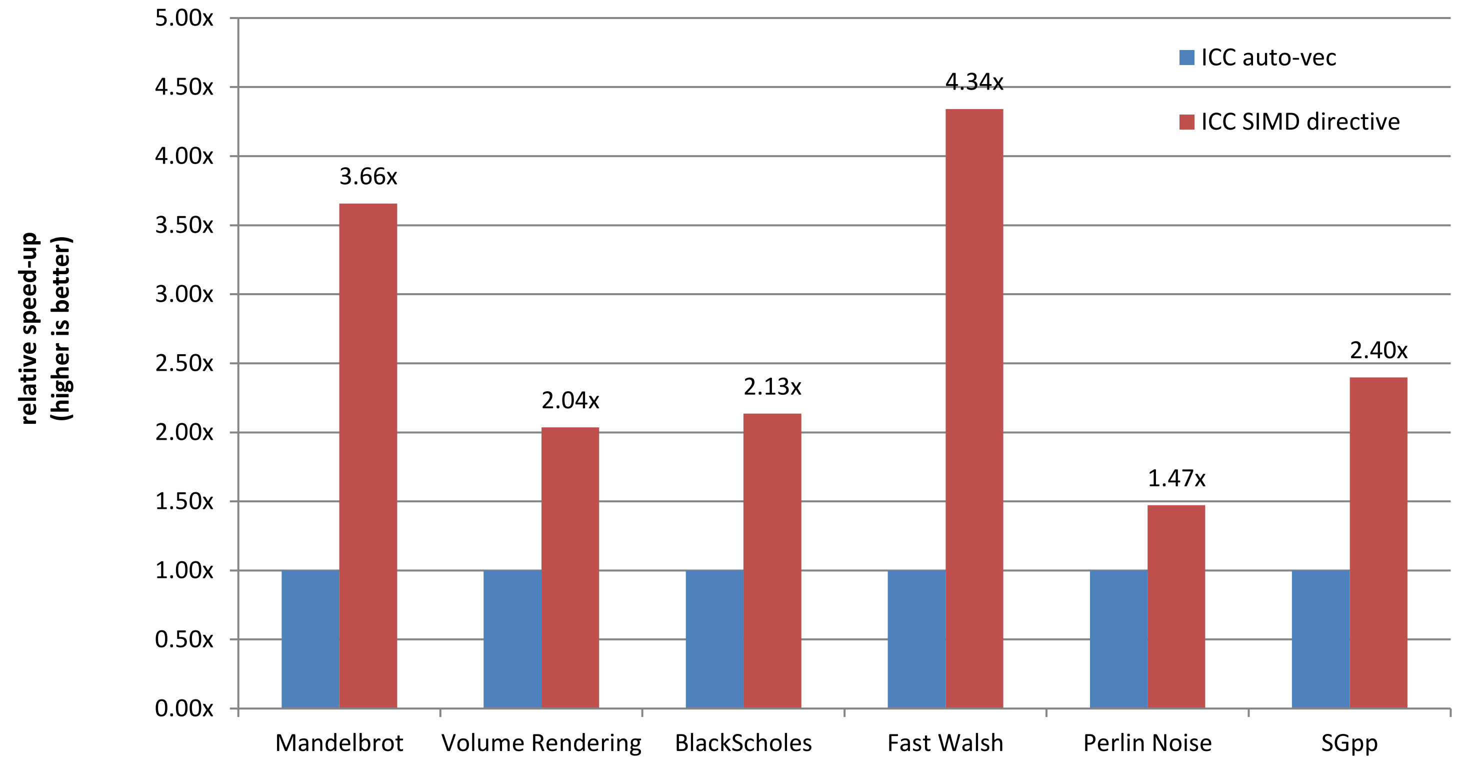
inbranch & notinbranch

```
#pragma omp declare simd inbranch
float do_stuff(float x) {
    /* do something */
    return x * 2.0;
}

void example() {
#pragma omp simd
    for (int i = 0; i < N; i++)
        if (a[i] < 0.0)
            b[i] = do_stuff(a[i]);
}
```

```
vec8 do_stuff_v(vec8 x, mask m) {
    /* do something */
    vmulpd x{m}, 2.0, tmp
    return tmp;
}
```

```
for (int i = 0; i < N; i+=8) {
    vcmp_lt &a[i], 0.0, mask
    b[i] = do_stuff_v(&a[i], mask);
}
```

SIMD Constructs & Performance



M.Klemm, A.Duran, X.Tian, H.Saito, D.Caballero, and X.Martorell. Extending OpenMP with Vector Constructs for Modern Multicore SIMD Architectures. In Proc. of the Intl. Workshop on OpenMP, pages 59-72, Rome, Italy, June 2012. LNCS 7312.

OpenMP Offload Programming

Topics

- OpenMP device and execution model
- Offload basics and exploiting parallelism
- Asynchronous offloading
- Hybrid OpenMP and HIP
- Advanced Task Synchronization
- Case Study: NWChem TCE CCSD(T)

Introduction to OpenMP Offload Features

Running Example for this Presentation: saxpy

```
void saxpy() {
    float a, x[SZ], y[SZ];
    // left out initialization
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
#pragma omp parallel for firstprivate(a)
    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);
}
```

Timing code (not needed, just to have a bit more code to show ☺)

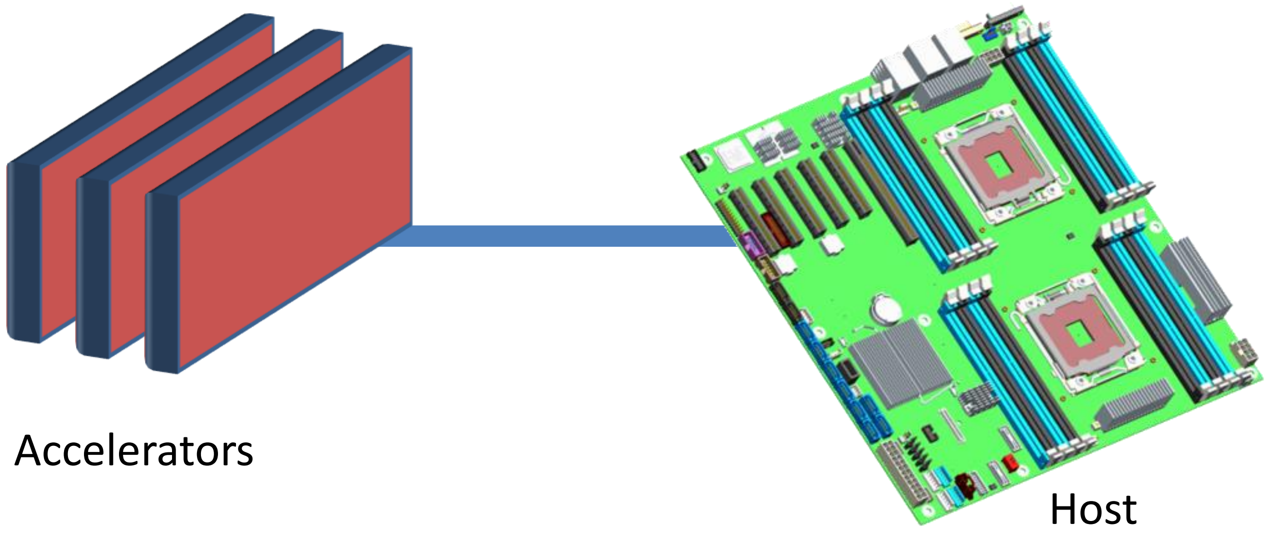
This is the code we want to execute on a target device (i.e., GPU)

Timing code (not needed, just to have a bit more code to show ☺)

Don't do this at home!
Use a BLAS library for this!

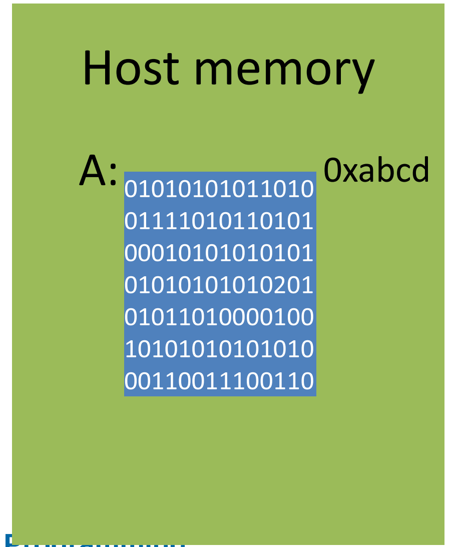
Device Model

- As of version 4.0 the OpenMP API supports accelerators/coprocessors
- Device model:
  - →One host for “traditional” multi-threading
  - →Multiple accelerators/coprocessors of the same kind for offloading



OpenMP Execution Model for Devices

- Offload region and its data environment are bound to the lexical scope of the construct
  - → Data environment is created at the opening curly brace
  - → Data environment is automatically destroyed at the closing curly brace
  - → Data transfers (if needed) are done at the curly braces, too:
    - → Upload data from the host to the target device at the opening curly brace.
    - → Download data from the target device at the closing curly brace.



```
!$omp target         &
!$omp   map(alloc:A) &
!$omp   map(to:A)    &
!$omp   map(from:A)  &
   call compute(A)
!$omp end target
```

OpenMP for Devices - Constructs

- Transfer control and data from the host to the device
- Syntax (C/C++)

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

- Syntax (Fortran)

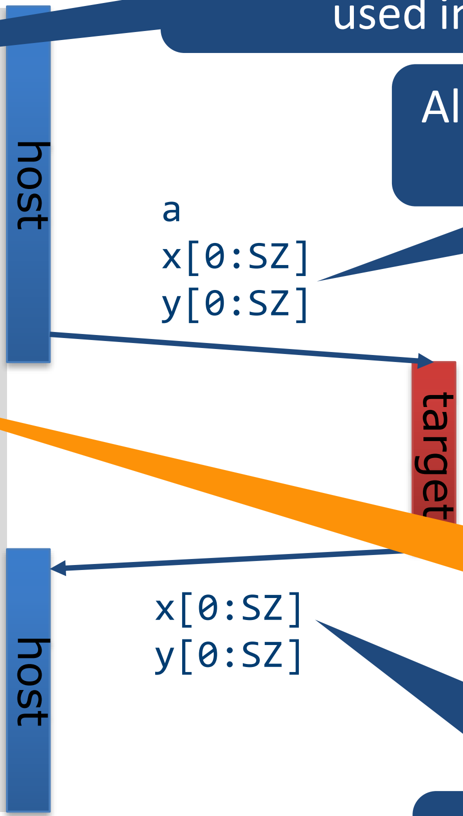
```
!$omp target [clause[[,] clause],…]
structured-block
!$omp end target
```

- Clauses

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

Example: saxpy

```
void saxpy() {
    float a, x[SZ], y[SZ];
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
#pragma omp target “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);
}
```



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

All accessed arrays are copied from host to device and back

Presence check: only transfer if not yet allocated on the device.

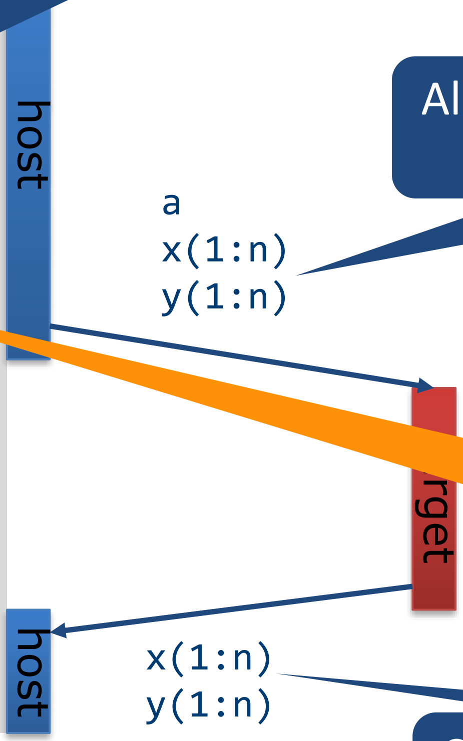
Copying x back is not necessary: it was not changed.

```
clang -fopenmp --offload-arch=gfx90a ...
```

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

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



All accessed arrays are copied from host to device and back

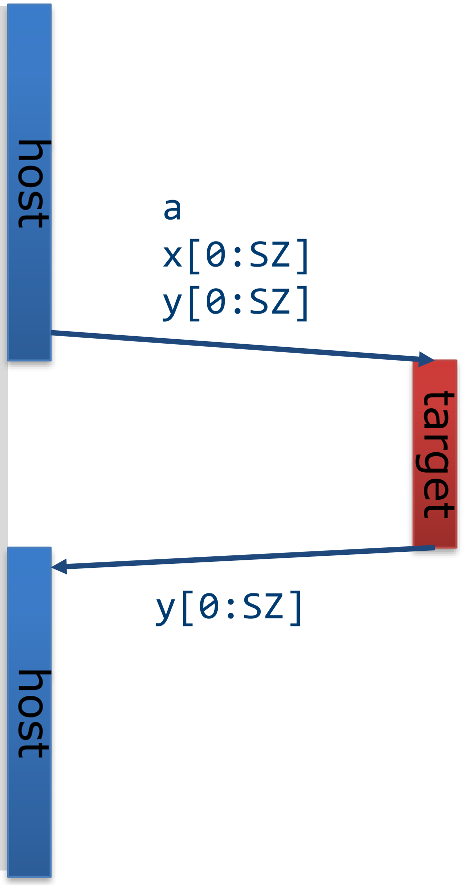
Presence check: only transfer if not yet allocated on the device.

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

```
flang -fopenmp --offload-arch=gfx90a ...
```

Example: saxpy

```
void saxpy() {
    double a, x[SZ], y[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);
}
```

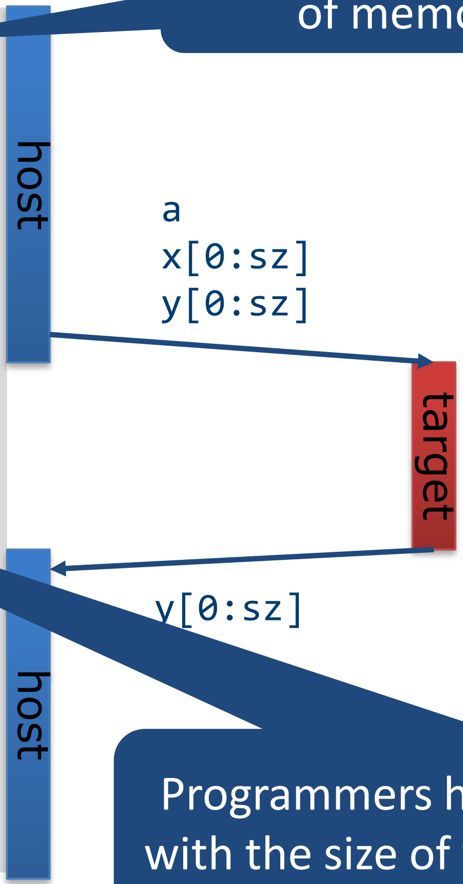


```
clang -fopenmp --offload-arch=gfx90a ...
```

Example: saxpy

```
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);
}
```

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



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

```
clang -fopenmp --offload-arch=gfx90a
```

Exploiting (Multilevel) 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
  - →This is intentional!

- 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 features for a target device such as a GPU, e.g., no I/O, limited use of base language features.

Example: 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 parallel for simd
    for (int i = 0; i < sz; i++) {
        y[i] = a * x[i] + y[i];
    }
}
```

host

target

host

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

Create a team of threads to execute the loop in parallel using SIMD instructions.

```
clang -fopenmp --offload-arch=gfx90a
```

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)
```

Multi-level Parallel saxpy

- Manual code transformation
  - → Tile the loop into an outer loop and an inner loop.
  - → Assign the outer loop to “teams”.
  - → Assign the inner loop to the “threads”.
  - → (Assign the inner loop to SIMD units.)

```
void saxpy(float a, float* x, float* y, int sz) {
    #pragma omp target teams map(to:x[0:sz]) map(tofrom:y[0:sz]) num_teams(nteams)
    {
        int bs = n / omp_get_num_teams();   // n assumed to be multiple of #teams
        #pragma omp distribute
        for (int i = 0; i < sz; i += bs) {
            #pragma omp parallel for simd firstprivate(i,bs)
            for (int ii = i; ii < i + bs; ii++) {
                y[ii] = a * x[ii] + y[ii];
}   }   }   }
```

Multi-level Parallel saxpy

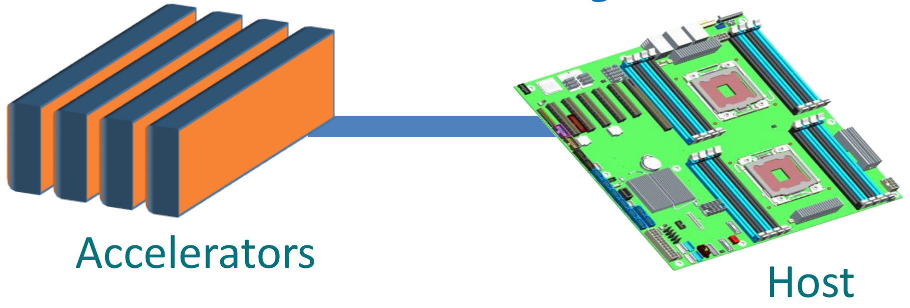
- For convenience, OpenMP defines composite constructs to implement the required code transformations

```
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
```

Optimizing Data Transfers

Optimizing Data Transfers is Key to Performance



- Connections between host and accelerator are typically lower-bandwidth, higher-latency interconnects

→Bandwidth host memory:	hundreds of GB/sec
→Bandwidth accelerator memory:	TB/sec
→PCIe Gen 4 bandwidth (16x):	tens of GB/sec

- Unnecessary data transfers must be avoided, by
  - →only transferring what is actually needed for the computation, and
  - →making the lifetime of the data on the target device as long as possible.

Role of the Presence Check

- If `map` clauses are not added to `target` constructs, presence checks determine if data is already available in the device data environment:

```
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
```

“present?(y)” “present?(x)”

– OpenMP maintains a mapping table that records what memory pointers have been mapped.
– That table also maintains the translation between host memory and device memory.
– Constructs with no `map` clause for a data item then determine if data has been mapped and if not, a `map(tofrom:…)` is added for that data item.

Optimize Data Transfers

- Reduce the amount of time spent transferring data:
  - →Use `map` clauses to enforce direction of data transfer.
  - →Use `target data`, `target enter data`, `target exit data` constructs to keep data environment on the target device.

```
subroutine caller
    ! Declarations omitted

!$omp target data map(to:x) &
                  map(tofrom:y)
    call saxpy(a, x, y, n)
!$omp end target
end subroutine
```

```
subroutine saxpy(a, x, y, n)
    ! Declarations omitted

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

Optimize Data Transfers

- Reduce the amount of time spent transferring data:
  - →Use `map` clauses to enforce direction of data transfer.
  - →Use `target data`, `target enter data`, `target exit data` constructs to keep data environment on the target device.

```
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, N); // uses target
        saxpy(2.0f, tmp, b, N);
        compute_kernel_2(tmp, b, N); // uses target
        saxpy(2.0f, c, tmp, N);
}   }
```

```
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];
}
```

`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 update` Construct Syntax

- Issue data transfers to or from existing data device environment
- Syntax (C/C++)

```
#pragma omp target update [clause[[,] clause],…]
```

- Syntax (Fortran)

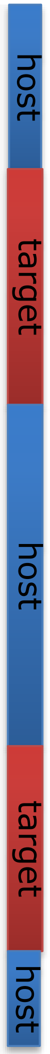
```
!$omp target update [clause[[,] clause],…]
```

- Clauses

```
device(scalar-integer-expression)
to(list)
from(list)
if(scalar-expr)
```

Example: target data and target update

```
#pragma omp target data device(0) map(alloc:tmp[:N]) map(to:input[:N)) map(from:res)
  {
#pragma omp target device(0)
#pragma omp parallel for
    for (i=0; i<N; i++)
      tmp[i] = some_computation(input[i], i);

    update_input_array_on_the_host(input);

#pragma omp target update device(0) to(input[:N])

#pragma omp target device(0)
#pragma omp parallel for reduction(+:res)
    for (i=0; i<N; i++)
      res += final_computation(input[i], tmp[i], i)
  }
```

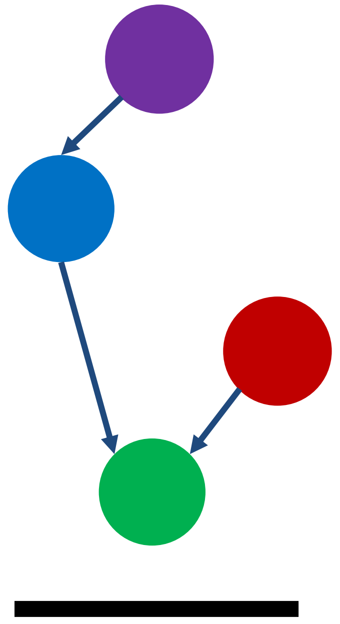


Asynchronous Offloading

Asynchronous Offloads

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

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

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

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

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

#pragma omp taskwait
```



Hybrid Programming

Hybrid Programming

- Hybrid programming here stands for the interaction of OpenMP with a lower-level programming model, e.g.
  - →OpenCL
  - →CUDA
  - →HIP

- OpenMP supports these interactions
  - →Calling low-level kernels from OpenMP application code
  - →Calling OpenMP kernels from low-level application code

Example: Calling saxpy

```
void example() {
    float a = 2.0;
    float * x;
    float * y;

    // allocate the device memory
    #pragma omp target data map(to:x[0:count]) map(tofrom:y[0:count])
    {
        compute_1(n, x);
        compute_2(n, y);
        saxpy(n, a, x, y)
        compute_3(n, y);
    }
}
```

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

Let’s assume that we want to implement the saxpy() function in a low-level language.

HIP Kernel for saxpy()

- Assume a HIP version of the SAXPY kernel:

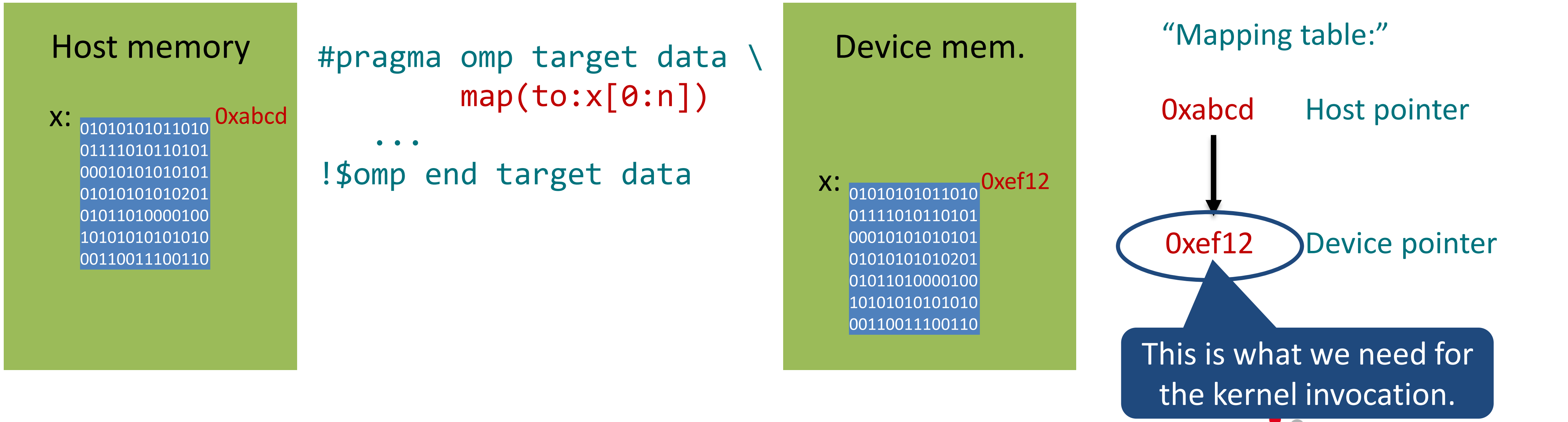
```
__global__ void saxpy_kernel(size_t n, float a, float * x, float * y) {
    size_t i = threadIdx.x + blockIdx.x * blockDim.x;
    y[i] = a * x[i] + y[i];
}

void saxpy_hip(size_t n, float a, float * x, float * y) {
    assert(n % 256 == 0);
    saxpy_kernel<<<n/256,256,0,NULL>>>(n, a, x, y);
}
```

These are device pointers!

- We need a way to translate the host pointer that was mapped by OpenMP directives and retrieve the associated device pointer.

Pointer Translation /1

- When creating the device data environment, OpenMP creates a mapping between
  - → the (virtual) memory pointer on the host and
  - → the (virtual) memory pointer on the target device.
- This mapping is established through the data-mapping directives and their clauses.



```
#pragma omp target data \
        map(to:x[0:n])
   ...
!$omp end target data
```

Pointer Translation /2

- The target data construct defines the `use_device_addr` clause to perform pointer translation.
  - → The OpenMP implementation searches for the host pointer in its internal mapping tables.
  - → The associated device pointer is then returned.

```
type * x = 0xabcd;
#pragma omp target data use_device_addr(x[:0])
{
    example_func(x);   // x == 0xef12
}
```

- Note: the pointer variable shadowed within the `target data` construct for the translation.

Putting it Together…

```
void example() {
    float a = 2.0;
    float * x = ...;   // assume: x = 0xabcd
    float * y = ...;

    // allocate the device memory
    #pragma omp target data map(to:x[0:count]) map(tofrom:y[0:count])
    {
        compute_1(n, x);  // mapping table: x:[0xabcd,0xef12], x = 0xabcd
        compute_2(n, y);
        #pragma omp target data use_device_addr(x[:0],y[:0])
        {
            saxpy_hip(n, a, x, y) // mapping table: x:[0xabcd,0xef12], x = 0xef12
        }
        compute_3(n, y);
    }
}
```

Advanced Task Synchronization

Asynchronous API Interaction

- Some APIs are based on asynchronous operations
  - → MPI asynchronous send and receive
  - → Asynchronous I/O
  - → HIP, CUDA and OpenCL stream-based offloading
  - → In general: any other API/model that executes asynchronously with OpenMP (tasks)
- Example: HIP memory transfers

```
do_something();
hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
do_something_else();
hipStreamSynchronize(stream);
do_other_important_stuff(dst);
```

- Programmers need a mechanism to marry asynchronous APIs with the parallel task model of OpenMP
  - → How to synchronize completions events with task execution?

Try 1: Use just OpenMP Tasks

```
void hip_example() {
#pragma omp task     // task A
    {
        do_something();
        hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
    }
    #pragma omp task // task B
    {
        do_something_else();
    }
    #pragma omp task // task C
    {
        hipStreamSynchronize(stream);
        do_other_important_stuff(dst);
    }
}
```

Race condition between the tasks A & C, task C may start execution before task A enqueues memory transfer.

- This solution does not work!

Try 2: Use just OpenMP Tasks Dependences

```
void hip_example() {
#pragma omp task depend(out:stream)    // task A
    {
        do_something();
        hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
    }
    #pragma omp task                   // task B
    {
        do_something_else();
    }
    #pragma omp task depend(in:stream) // task C
    {
        hipStreamSynchronize(stream);
        do_other_important_stuff(dst);
    }
}
```

Synchronize execution of tasks through dependence. May work, but task C will be blocked waiting for the data transfer to finish

- This solution may work, but
  - → takes a thread away from execution while the system is handling the data transfer.
  - → may be problematic if called interface is not thread-safe

Detachable

- OpenMP 5.0 introduces the concept of a detachable task
  - →Task can detach from executing thread without being “completed”
  - →Regular task synchronization mechanisms can be applied to await completion of a detached task
  - →Runtime API to complete a task

- Detached task events: `omp_event_handle_t` datatype
- Detached task clause: `detach(event)`
- Runtime API:
  `void omp_fulfill_event(omp_event_handle_t *event)`

Detaching Tasks

```
omp_event_handle_t  *event;
void detach_example() {
#pragma omp task detach(event)
    {
        important_code();
    }①

    #pragma omp taskwait ②④
}
```

Some other thread/task:

```
omp_fulfill_event(event); ③
```

1. Task detaches
2. `taskwait` construct cannot complete
3. Signal event for completion
4. Task completes and `taskwait` can continue

Putting It All Together

```
void callback(hipStream_t stream, hipError_t status, void *cb_dat) {
 ③omp_fulfill_event(* (omp_event_handle_t *) cb_data);
}
void hip_example() {
    omp_event_handle_t hip_event;
#pragma omp task detach(hip_event) // task A
    {
        do_something();
        hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
        hipStreamAddCallback(stream, callback, &hip_event, 0);
  ①}
#pragma omp task                  // task B
        do_something_else();

#pragma omp taskwait②④
#pragma omp task                  // task C
    {
        do_other_important_stuff(dst);
}   }
```

1. Task A detaches
2. taskwait does not continue
3. When memory transfer completes, callback is invoked to signal the event for task completion
4. taskwait continues, task C executes

Removing the `taskwait` Construct

```
void callback(hipStream_t stream, hipError_t status, void *cb_dat) {
  ② omp_fulfill_event(* (omp_event_handle_t *) cb_data);
}
void hip_example() {
    omp_event_handle_t hip_event;
#pragma omp task depend(out:dst) detach(hip_event) // task A
    {
        do_something();
        hipMemcpyAsync(dst, src, nbytes, hipMemcpyDeviceToHost, stream);
  ①     hipStreamAddCallback(stream, callback, &hip_event, 0);
    }
#pragma omp task                     // task B
        do_something_else();

#pragma omp task depend(in:dst)    ③ // task C
    {
        do_other_important_stuff(dst);
}   }
```

1. Task A detaches and task C will not execute because of its unfulfilled dependency on A
2. When memory transfer completes, callback is invoked to signal the event for task completion
3. Task A completes and C's dependency is fulfilled

Case Study: NWChem TCE CCSD(T)

- TCE: Tensor Contraction Engine
  CCSD(T): Coupled-Cluster with Single, Double,
  and perturbative Triple replacements

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 Northwest 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 $O(n^2)$

Example Kernel (1 of 27 in total)

```
      subroutine sd_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)
!$omp target „presence?(triplesx,t2sub,v2sub)"
!$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 have 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 each

Invoking the Kernels / Data Management

- Simplified pseudo-code

```
!$omp target enter data map(alloc:triplesx(1:tr_size))
c     for all tiles
      do ...
        call zero_triplesx(triplesx)
        do ...
          call comm_and_sort(t2sub, v2sub)
!$omp target data map(to:t2sub(t2_size)) map(to:v2sub(v2_size))
          if (...)
            call sd_t_d1_1(h3d,h2d,h1d,p6d,p[illegible]4d,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 map(release:triplesx(1:size))
```

Allocate 1.5GB data once, stays on device.

Update 2x2.5MB 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

Invoking the Kernels / Data Management

- Simplified pseudo-code

```
!$omp target enter data map(alloc:triplesx(1:tr_size))
c     for all tiles
      do ...
        call zero_triplesx(triplesx)
        do ...
          call comm_and_sort(t2sub, v2sub)
!$omp target data map(to:t2sub(t2_size)) map(to:v2sub(v2_size)
          if (...)
            call sd_t_d1_1(h3d,h2d,h1d,p6d,p    4d,h7,triplesx
          end if
c         same for sd_t_d1_2 until sd_t_d1_9
!$omp target end data
        end do
        do ...
c         Similar structure for sd_t_d2_1 until sd_t_d2_9, inc
        end do
        call sum_energy(energy, triplesx)
      end do
!$omp target exit data map(release:triplesx(1:size))
```

Allocate 1.5G
stays on

Update 2x2.5
(potentially) r

```
      subroutine sd_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)
!$omp target „presence?(triplesx,t2sub,v2sub)"
!$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
       triplesx(h2h
     1   - t2sub(h7
      end do
      end do
      end do
      end do
      end do
      end do
!$omp end teams distribute parallel do
!$omp end target
      end subroutine
```

Presence check determines that arrays have been allocated in the device data environment already.

Future OpenMP Directions

OpenMP 6.0 includes many major new features

- Officially released on November 14, 2024
  - → Reflects three years of work since release of OpenMP 5.2
  - → Includes 416 enacted issues, covering a wide range of content and complexity
- Free-agent threads significantly change execution model, implementations
- New concept for task dependences: transparent tasks
  - → Enables asynchronous `target data` (also enables other future extensions)
- User-defined induction and `induction` clause expand parallelism support
- Many significant device support improvements (e.g., `workdistribute`)
- Several additional (sequential) loop transforming directives
- Supported compound constructs are now defined based on a grammar
- Significant improvements to usability and correctness of specification

OpenMP tasking advances have pervasive impact

- Other major additions to 6.0 include:
  - → Support for dependences and affinity of tasks generated by `taskloop` directives
  - → A new `taskgraph` directive that enables optimized task generation
- Task-generating constructs are fundamental to OpenMP offload model
  - → Most device constructs (e.g., `target` and `target_update` directives) already generate them
  - → Another major change: `target_data` is now a dependence sequence of three tasks
    - → Middle task is transparent by default
    - → The construct now is also a `taskgroup` region by default
    - → Can specify `no_wait` and `no_group` to rely only on dependences for ordering
- Other constructs (e.g., `parallel` and `teams`) are composed of implicit tasks
  - → While not adopted for 6.0, expect to add `transparent` clause to many of them eventually
  - → Will enable `no_wait` to be supported for `parallel` construct

Topics

- Current OpenMP Language Committee Activities
- OpenMP Organizational Overview
- Final Review of OpenMP 5.0, 5.1, 5.2 and 6.0 (included for reference)

OpenMP Language Committee Current Activities: TR14 and OpenMP 6.1

OpenMP 6.1 will refine and amend OpenMP 6.0

- Significant progress has already been made
  - → 18 issues have been adopted, mostly covering small updates to 6.0 additions
  - → Language committee face-to-face meeting week after next will result in many more issues moving forward
- Targeting some significant improvements for device support
  - → Support for dynamic groupprivate memory (e.g., small, optimized GPU memory pool) (done!)
  - → Support for explicit control of pointer attachment (done!)
  - → Improved support for implicit `declare target` in Fortran
  - → Beginning work on “kernel language”, which will provide more low-level device control
- Expect continued refinement in many other areas
  - → More loop transformations, refinements of other ones
  - → Working on mechanism to control OpenMP defaults used for a translation unit
  - → Considering additional extensions that build on transparent tasks (e.g., `parallel nowait`)
  - → Many other small changes, particularly related to tasking and tool support, are likely

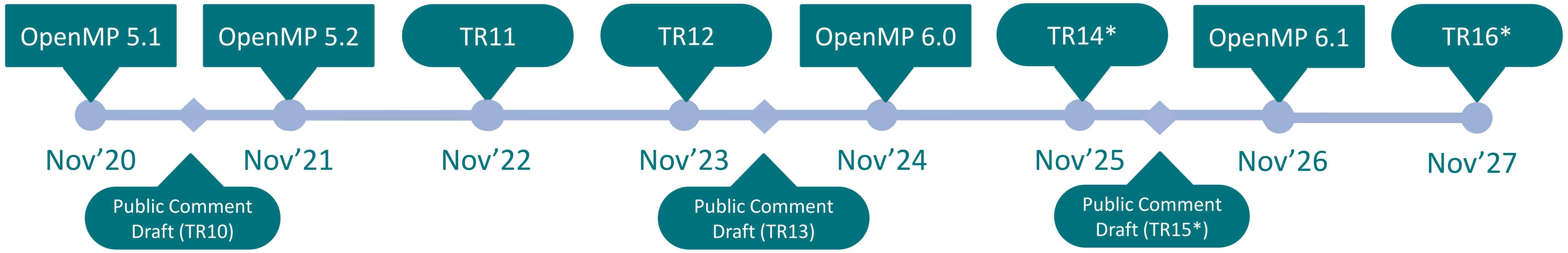
Things likely to be deferred to beyond 6.1

- True support for using multiple devices
  - →Device-to-device scoping support for atomic and other memory operations
  - →Support for bulk launch
  - →Support to update data on multiple devices (broadcast/multicast, other collectives)
  - →Support for work distribution across devices
  - →Considering relaxing restrictions on nested `target` regions
- Support for pipelining, data-flow, other parallelization models
- Support for event-based parallelism
- Characterizing loop-based work distribution constructs as transformations
- Efficient use of multiple compilation units (i.e., support for efficient IPO)

OpenMP Organizational Overview

OpenMP Roadmap

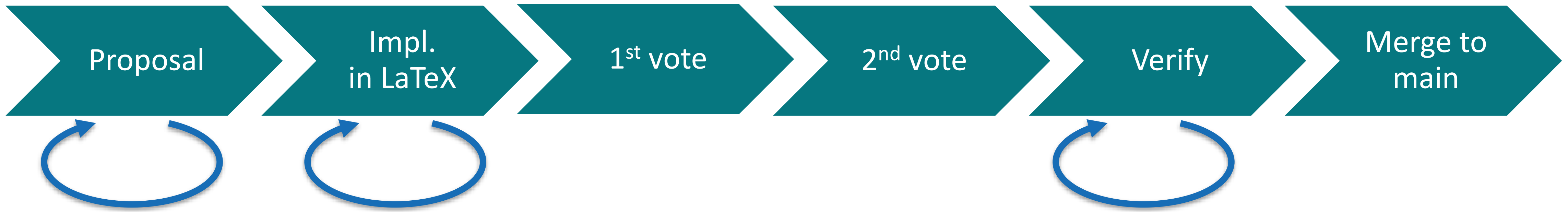
- OpenMP has a well-defined roadmap:
  - →5-year cadence for major releases
  - →One minor release in between
  - →OpenMP 5.2 was added as a second minor release before OpenMP version 6.0
  - →(At least) one Technical Report (TR) with feature previews in every year



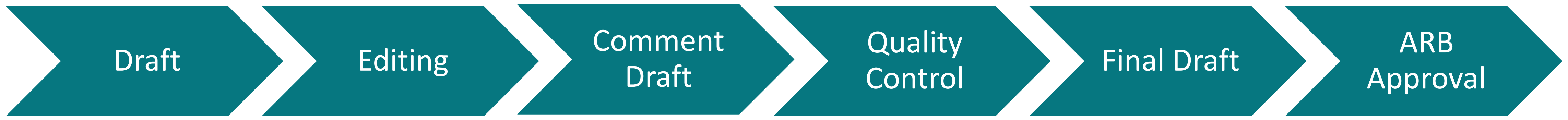
* Numbers assigned to TRs may change if additional TRs are released.

Development Process of the Specification

- Modifications to the OpenMP specification follow a (strict) process:



- Release process for specifications:



User Outreach & Education

Check out openmp.org/news/events-calendar/

Help Us Shape the Future of OpenMP

- OpenMP continues to grow
  - →32 members currently
- You can contribute to our annual releases
- Attend IWOMP, understand and shape research directions
- OpenMP membership types now include less expensive memberships
  - →Please let us know if you would be interested

Final Review of OpenMP 5.0, 5.1, 5.2 and 6.0 Included for Reference

Ratified OpenMP 5.0 in November 2018, Ratified OpenMP 5.1 in November 2020

- OpenMP 5.0
  - →Addressed several major open issues for OpenMP
  - →Included 293 passed tickets
- OpenMP 5.1
  - →Includes many refinements to 5.0 additions
  - →Included 254 passed GitHub issues
- OpenMP 5.2
  - →Mostly address quality of specification issues but also refines 5.0 and 5.1 additions
  - →Included 131 passed GitHub issues

Major new features in OpenMP 5.0

- Significant extensions to improve usability
  - →OpenMP contexts, `metadirective` and `declare variant`
  - →Addition of `requires` directive, including support for unified shared memory
  - →Memory allocators and support for deep memory hierarchies
  - →Descriptive `loop` construct
  - →Ability to quiesce OpenMP threads
  - →Support to print/inspect affinity state
  - →Release/acquire semantics added to memory model
  - →Support for C/C++ array shaping
- First (OMPT) and third (OMPD) party tool support

Major new features in OpenMP 5.0

- Some significant extensions to existing functionality
  - → Verbosity reducing changes such as implicit `declare target` directives
  - → User defined mappers provide deep copy support for map clauses
  - → Support for reverse offload
  - → Support for task reductions , including on `taskloop` construct, task affinity, new dependence types, depend objects and detachable tasks
  - → Allows `teams` construct outside of `target` construct (i.e., on host)
  - → Supports collapse of non-rectangular loops
  - → Scan extension of reductions
- Major advances for base language normative references
  - → Completed support for Fortran 2003
  - → Added initial support of Fortran 2008, C11, C++11, C++14 and C++17

OpenMP 5.0 clarifications and enhancements

- Supports collapse of imperfectly nested loops
- Supports `!=` on C/C++ loops
- Adds `conditional` modifier to `lastprivate`
- Support use of any C/C++ *lvalue* in `depend` clauses
- Permits `declare target` on C++ classes with virtual members
- Clarification of `declare target` C++ initializations
- Adds `task` modifier on many `reduction` clauses
- Adds `depend` clause to `taskwait` construct

OpenMP 5.1 refines existing functionality

- Adds full support for C11, C++11, C++14, C++17, C++20 and Fortran 2008 and partial support for Fortran 2018
- Extends directive syntax to C++ attribute specifiers
- The `scope` construct supports reductions within parallel regions
  - →Christian discussed this enhancement in another session
- Extends `atomic` construct to support compare-and-swap, min and max
  - →Detailed these enhancements in another session
- Adds many clauses and clause modifiers including:
  - →`nowait` to `taskwait` construct
  - →`strict` modifier to clauses on the `taskloop` construct

OpenMP 5.1 refines existing functionality

- Support for mapping (translated) function pointers
- Device-specific environment variables to control their ICVs
- `nothing` directive supports `metadirective` clarity and completeness
- Several new runtime routines, including more memory allocation flavors
- Deprecations include:
  - →The `master` affinity policy and `master` construct
  - →Cray pointers
  - →Many enum values, most related to OMPT (first-party tool interface)

OpenMP 5.1 adds some significant extensions

- The `interop` construct
  - →Improves native device support (e.g., CUDA streams)
  - →Also supports interoperability with CPU-based libraries (e.g., TBB)
- The new `dispatch` construct, improved `declare variant` directive
  - →Enable use of variants with device-specific arguments
  - →Elision of “unrecognized” code

OpenMP 5.1 adds some significant extensions

- The `assume` directive
  - → Supports optimization hints based on invariants
  - → Supports promise to limit OpenMP usage to (optimizable) subsets
- Loop transformation directives: The `tile` and `unroll` directives
  - → Control use of traditional sequential optimizations
  - → Ensure that they are applied when, where appropriate relative to parallelization

The `error` directive supports user-defined warnings and errors

- Use `error` directive to interact with the compiler

```
#pragma omp error [at(compilation|execution)] [severity(fatal|warning)] \
                  [message(msg-string)]
  structured-block
```

- Compiler displays `msg-string` as part of implementation-defined message
- The `at` clause determines when the effect of the directive occurs
  - → `compilation`: If encountered during compilation in a declarative context (useful along with `metadirective`) or is reachable at runtime
  - → `execution`: If the code location is encountered during execution (similar to `assert()`)
- The `severity` clause determines compiler action
  - → `warning`: Print message only (default)
  - → `fatal`: Stop compilation or execution

The `masked` construct supports filtering execution per thread

- Use `masked` construct to limit parallel execution (low cost: no data environ.)

```
#pragma omp masked [filter(integer-expression)]
  structured-block
```

- Encountering thread executes if `filter` clause matches its thread number
- Default (i.e., no clause) is equivalent to deprecated `master` construct
- Future (i.e., OpenMP 6.0) enhancements planned
  - →Define concept of thread groups, a subset of the threads in a team
  - →Extend `masked` to `filter` based on thread groups or booleans (via clause modifier)
  - →`filter` clause added to other constructs, relying on thread group concept

OpenMP 5.2 improves quality of the specification

- Large portions of specification now generated from JSON-based database
  - → Section headers and directive and clause format
  - → Cross references, index entries, hyperlinks and many other document details
  - → Long-term plan will capture sufficient information in database to generate much more, including grammar, quick reference guide, and header and runtime library routine stub files
- Improves specification of OpenMP syntax
  - → Ensuring syntax of directives and clauses is well-specified and consistent
  - → Ensuring restrictions are consistent and not just implied by syntax
  - → Deprecating one-off syntax choices, many other inconsistencies (12 new deprecation entries)
  - → Makes C++ attribute syntax a first-class citizen
- Many other minor improvements

OpenMP 6.0 includes many major new features

- Free-agent threads significantly change execution model, implementations
- New concept for task dependences: transparent tasks
  - →Enables asynchronous `target data` (also enables other future extensions)
- The `target_data` directive is now a dependence sequence of three task
- Support for dependences and affinity of tasks generated by `taskloop`
- The `taskgraph` directive enables optimized task generation
- User-defined induction and `induction` clause expand parallelism support
- Many significant device support improvements (e.g., `workdistribute`)
- Several additional (sequential) loop transforming directives
- Supported compound constructs are now defined based on a grammar
- Significant improvements to usability and correctness of specification

Induction: Parallelization despite dependences

```
xi = x0;
result = 0.0;
#pragma omp parallel for reduction(+: result) induction(step(x), *: xi)
for (I = 0; I < N; i++) {
  result += c[i] * xi;
  xi *= x;
}
```

- Simple inductions are similar to reductions, particulary with use of `inscan`
  - →Avoids complexity needed to avoid serialization for parallel scan computations
- User-defined induction greatly expands expressible loop parallelism
  - →Can define complex functions to perform computations with dependences
  - →Can use `collector` clause to specify closed form function to enable starting at arbitrary iterations (typically used for start of chunks but can allow arbitrarily)

What is the effect of the following code?

```
// assume in main with initialization omitted
// assume no OpenMP directives omitted

TS = 4096;
#pragma omp taskloop grainsize(TS)
for (i = 0; i < SIZE; i++) {
  A[i] = A[i] * B[i] * s;
}
```

- Pre-6.0 need `parallel masked` directive so multiple threads execute tasks

```
// assume in main with initialization omitted
// assume no OpenMP directives omitted

TS = 4096;
#pragma omp parallel masked
#pragma omp taskloop grainsize(TS)
for (i = 0; i < SIZE; i++) {
  A[i] = A[i] * B[i] * s;
}
```

6.0 evolves execution model significantly

```
// assume in main with initialization omitted
// assume no OpenMP directives omitted

TS = 4096;
#pragma omp taskloop grainsize(TS) threadset(omp_pool)
for (i = 0; i < SIZE; i++) {
  A[i] = A[i] * B[i] * s;
}
```

- OpenMP 6.0 defines OpenMP threads as members of logical thread pool
  - →Pool size can be specified by `OMP_THREAD_LIMIT` environment variable
- OpenMP 6.0 also adds the concept of free-agent threads
  - →Do not need `parallel masked` directive
  - →Instead `threadset` clause can specify that unassigned threads may execute tasks

Task dependences constrain modularity

```
// assume library must ensure fine-grain dependences are honored
int my_func(double *M, double *v) {
  int i, j, k;

  for (i = 0; i < N_ROWS; i += ROWS_PER_TASK) {
    #pragma omp task depend(inout:M[i*N_COLS])
    for (j = 0; j < ROWS_PER_TASK; j++) {
      for (k = 0; k < N_COLS; k++) {
        M[(i+j)*N_COLS + k] = M[(i+j)*N_COLS + k] * v[k]; } } }
  return 0;
}
```

- Successive calls to `my_func` with the same `M` are ordered correctly in OpenMP 5.2 and earlier if they are issued in the same task
  - →Ensures all uses of `task` construct will not deadlock
  - →Other synchronization can alleviate constraint by eliminating concurrency of tasks from different calls so this solution does not provide the desired result

Transparency supports rich dependence graphs

```
// assume my_func as in previous example
double M[N_ROWS*NCOLS], v[NUM_VS][N_COLS];
int i;

// code to initialize M and v omitted for brevity

for (i = 0; i < NUM_VS; i++) {
  #pragma omp task depend(inout:i) transparent(omp_impex)
  my_func(M, &v[i*N_COLS]);
}
```

- The calls to `my_func` are ordered because of the dependence shown
- These tasks are transparent importing and exporting ("`omp_impex`") tasks
  - →Dependences expressed in the calls are now imported and exported
  - →Deadlock freedom is still guaranteed

Extended `parallel` directive to support complete user control of number of threads

- The `parallel` directive will accept a new modifier and two “new” clauses

```
#pragma omp parallel [num_threads(prescriptiveness: nthreads)] \
                     [severity(fatal|warning)][message(msg-string)]
  structured-block
```

- Using `strict` *prescriptiveness* requires `nthreads` to be provided
- Clauses, previously available on `error` directive, effective with `strict` if cannot provide `nthreads`
  - →Display `msg-string` as part of implementation-defined message
  - →If `severity` is `fatal` execution is terminated
  - →If `severity` is `warning` message is displayed but execution continues
- Also now allowed to provide a list for `nthreads` to support nested parallelism