OpenMP Tutorial

Performance: Vectorization

Christian Terboven

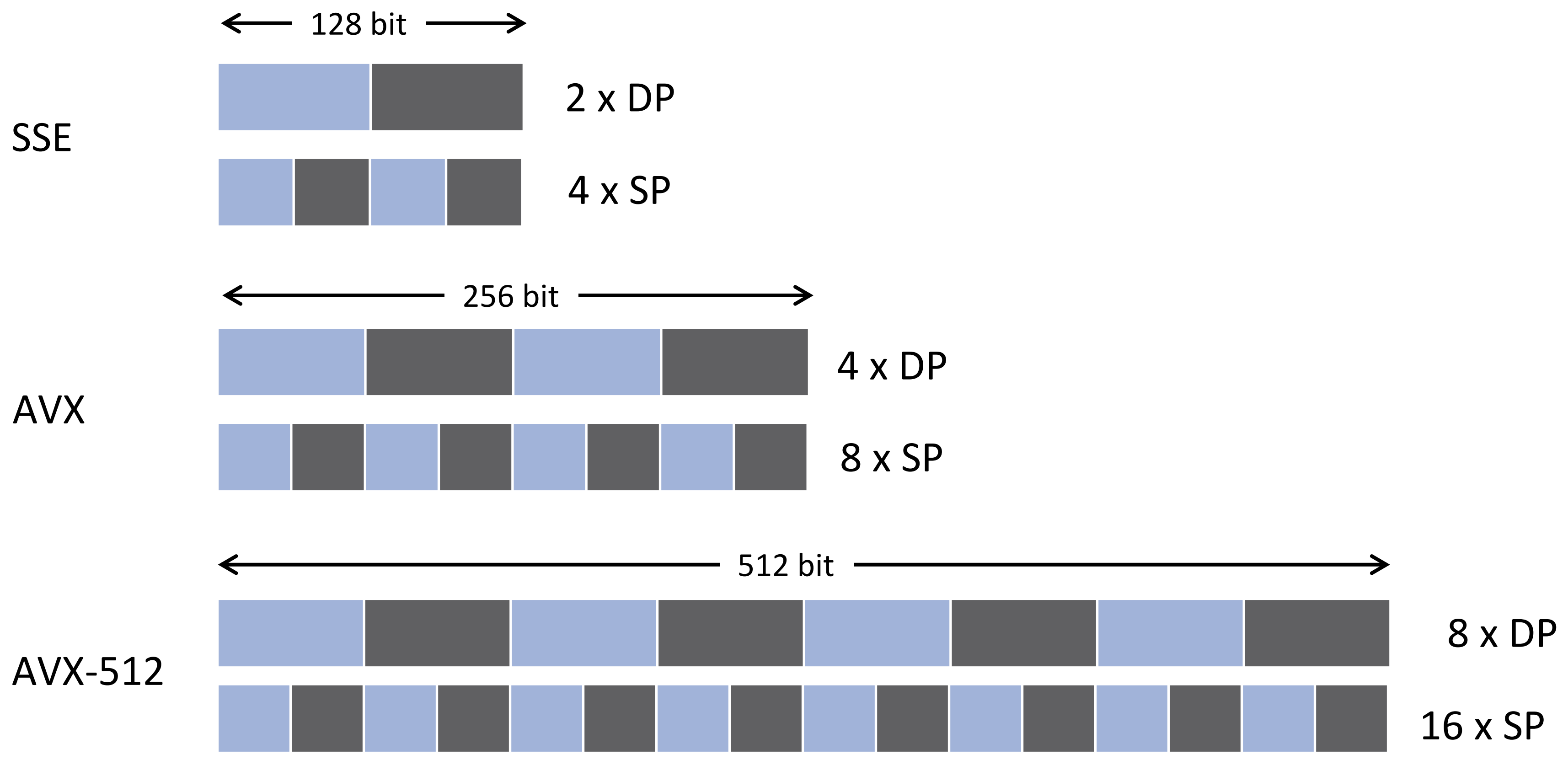
Michael Klemm

Topics

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

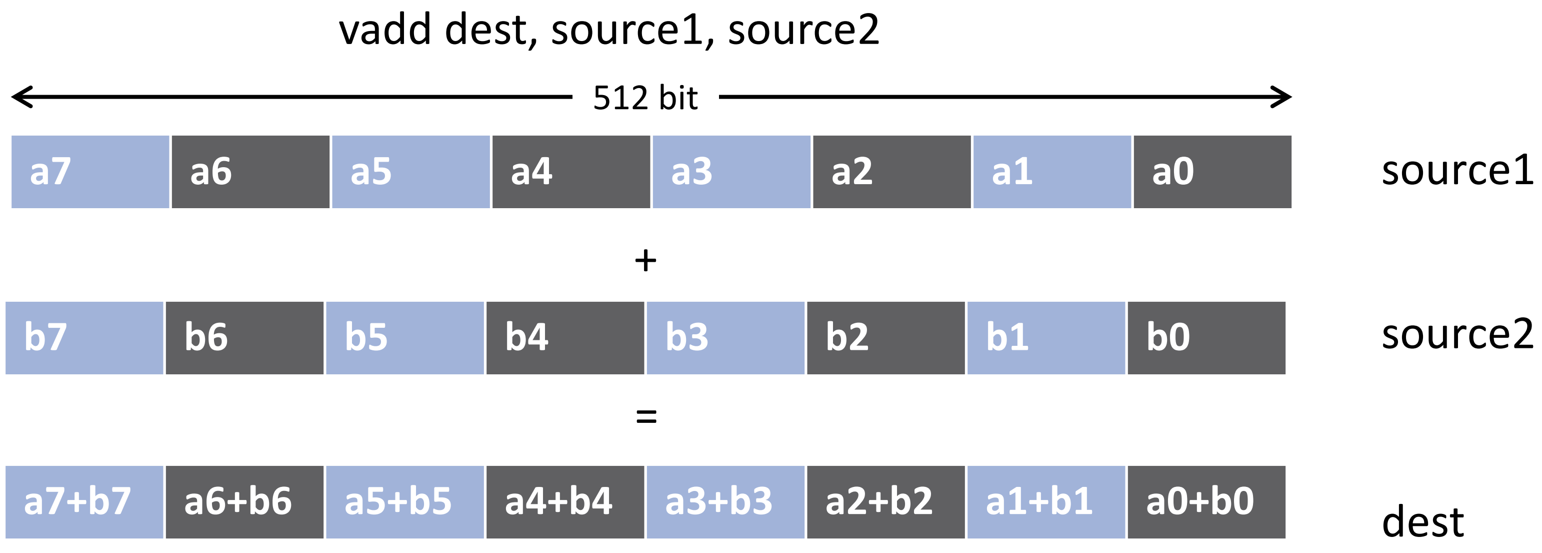
SIMD on x86 Architectures

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



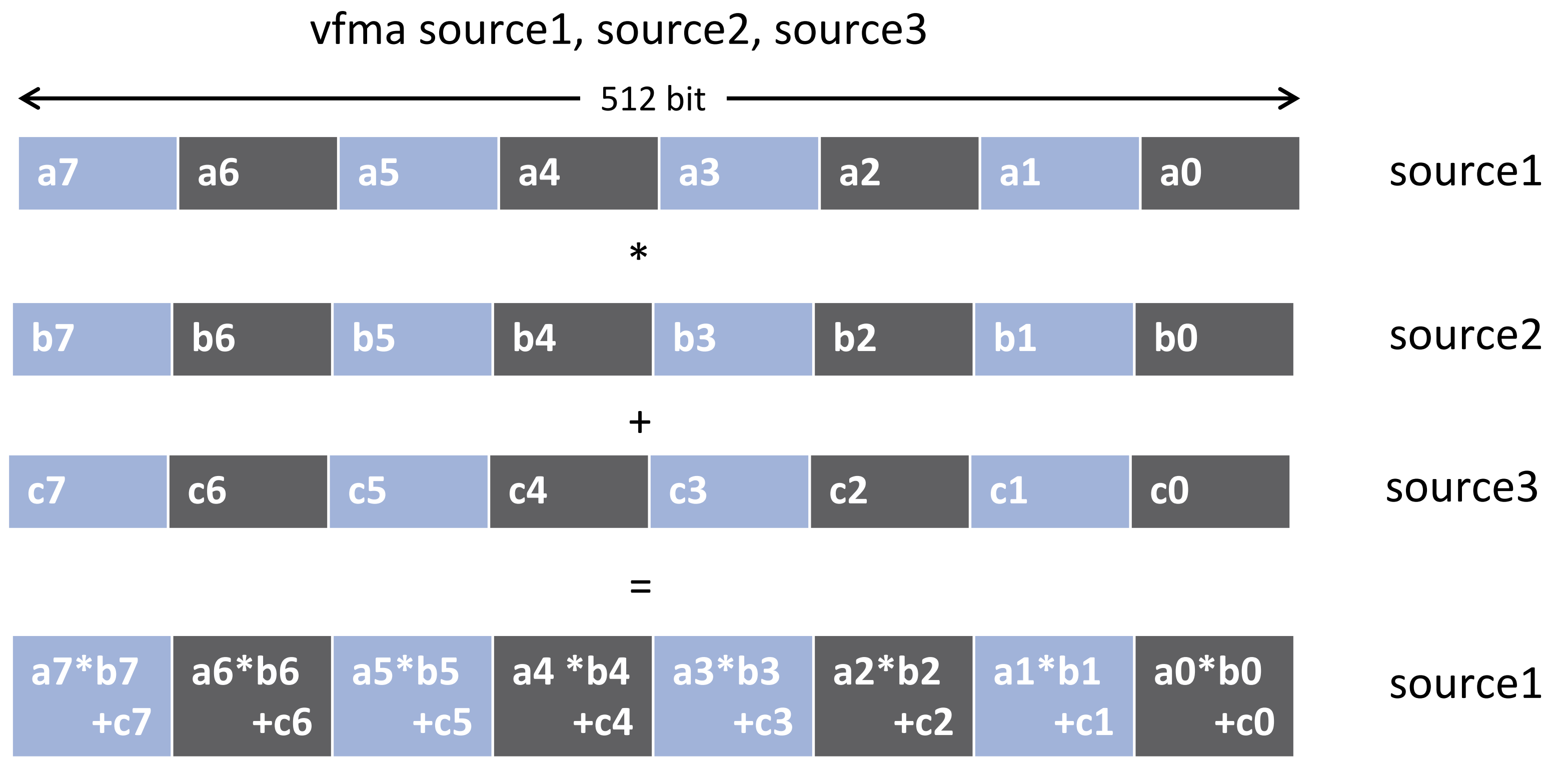
More Powerful SIMD Units

- SIMD instructions become more powerful



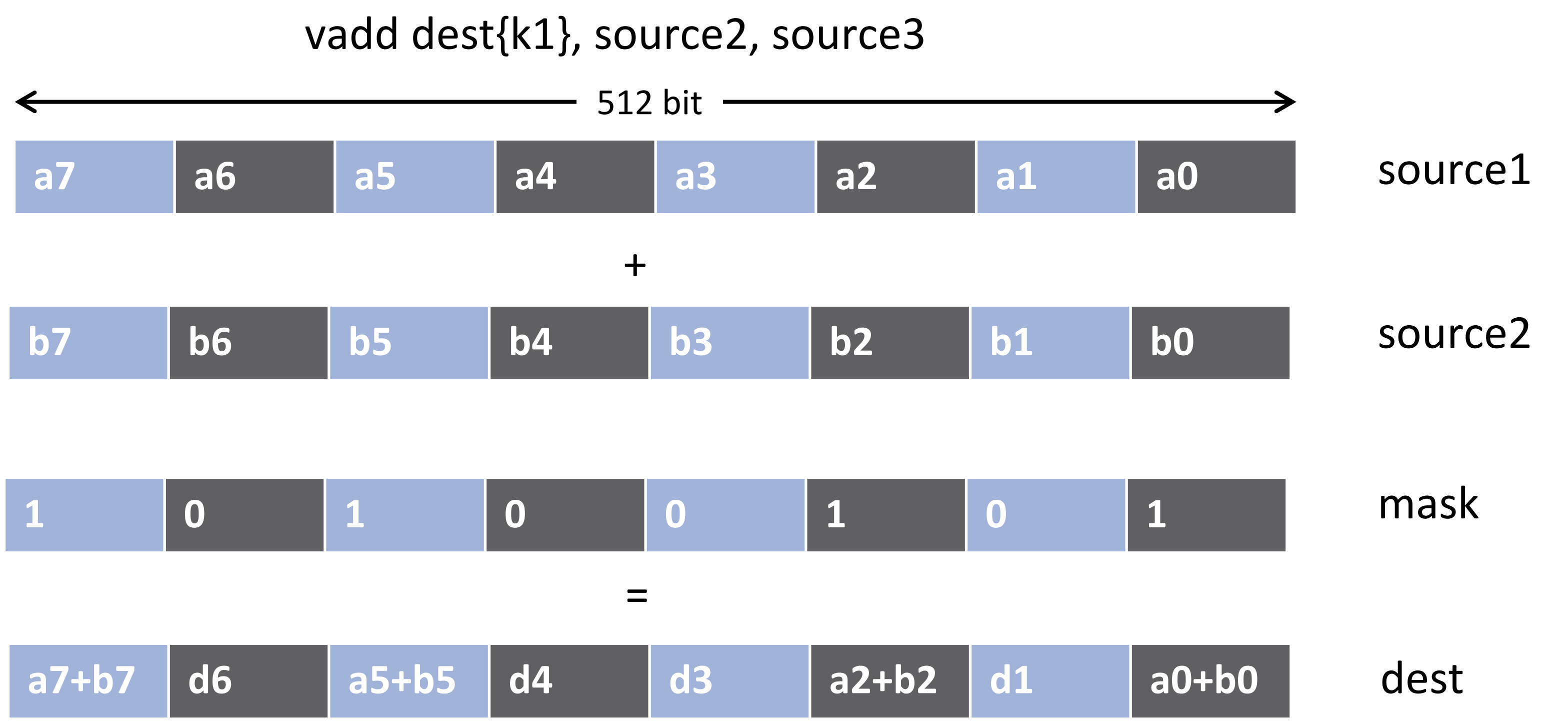
More Powerful SIMD Units

- SIMD instructions become more powerful



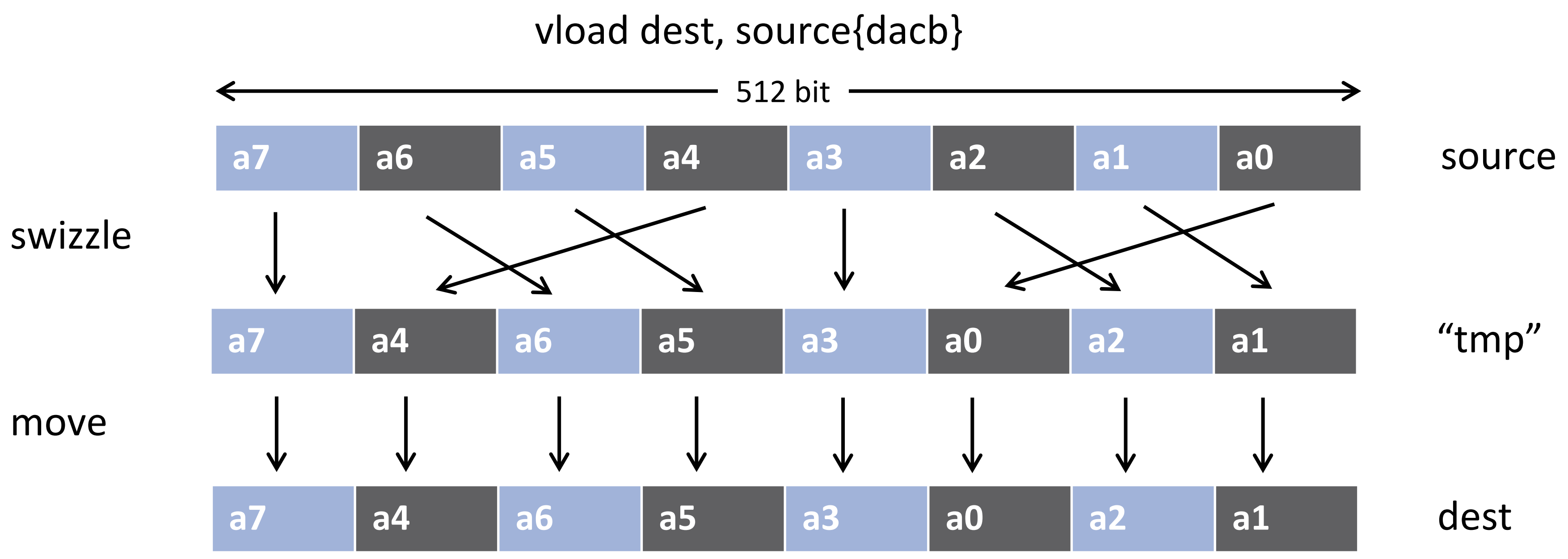
More Powerful SIMD Units

- 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];
    }
}
```

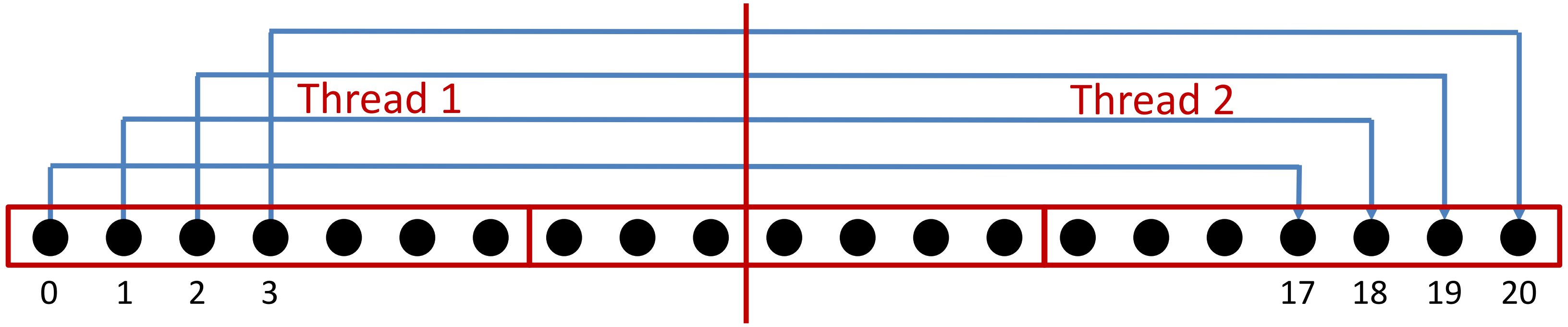
Loop-carried dependency for a[i] and a[i+17]; distance is 17.

- 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] + ...;
}
```

You need to trust your compiler to do the “right” thing.

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++)

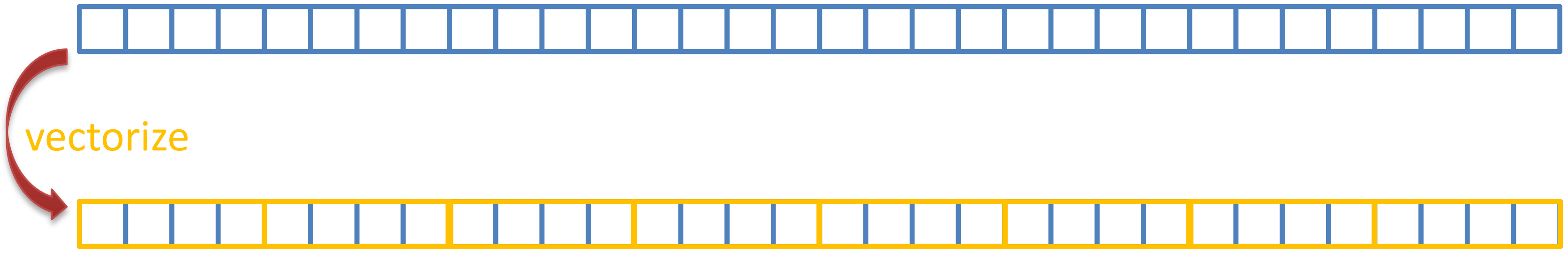
```
#pragma omp simd [clause[[,] clause],…]
for-loops
```

- Syntax (Fortran)

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

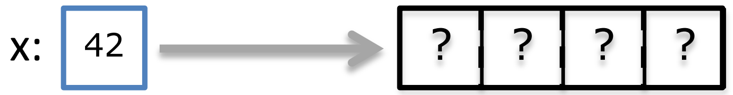
Example

```
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;
}
```

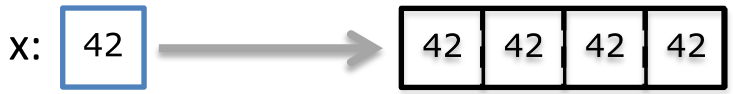


Data Sharing Clauses

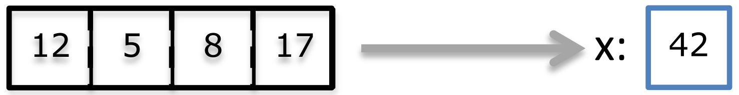
- `private(`*`var-list`*`):`
  Uninitialized vectors for variables in *var-list*



- `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++)

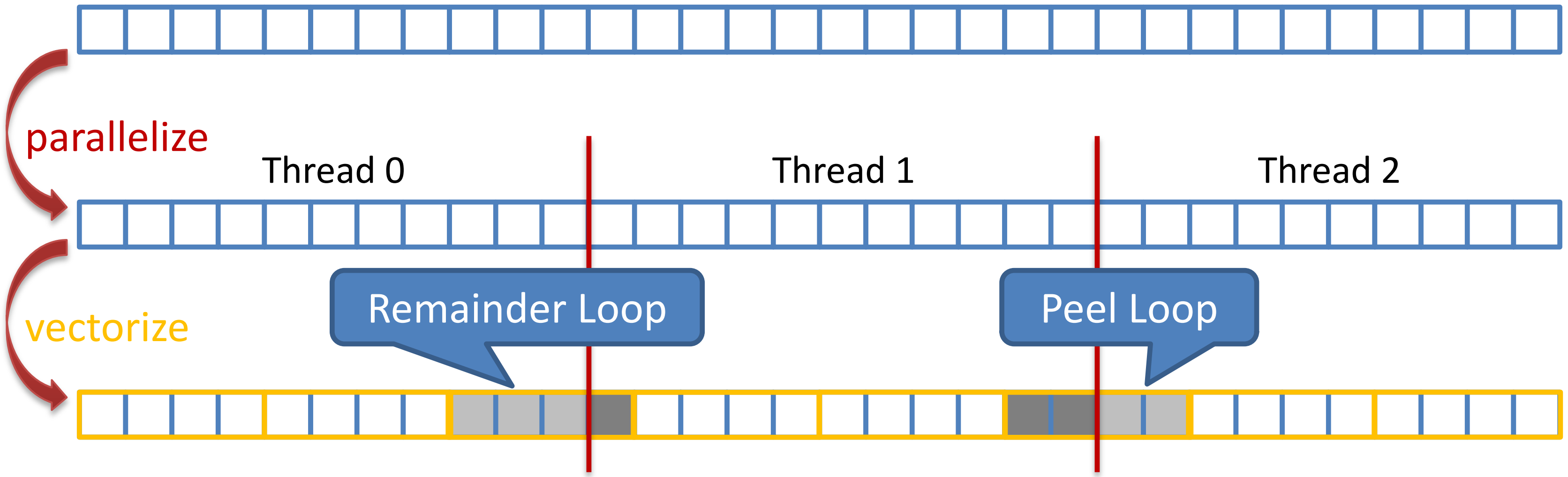
```
#pragma omp for simd [clause[[,] clause],…]
for-loops
```

- 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]`*`)`

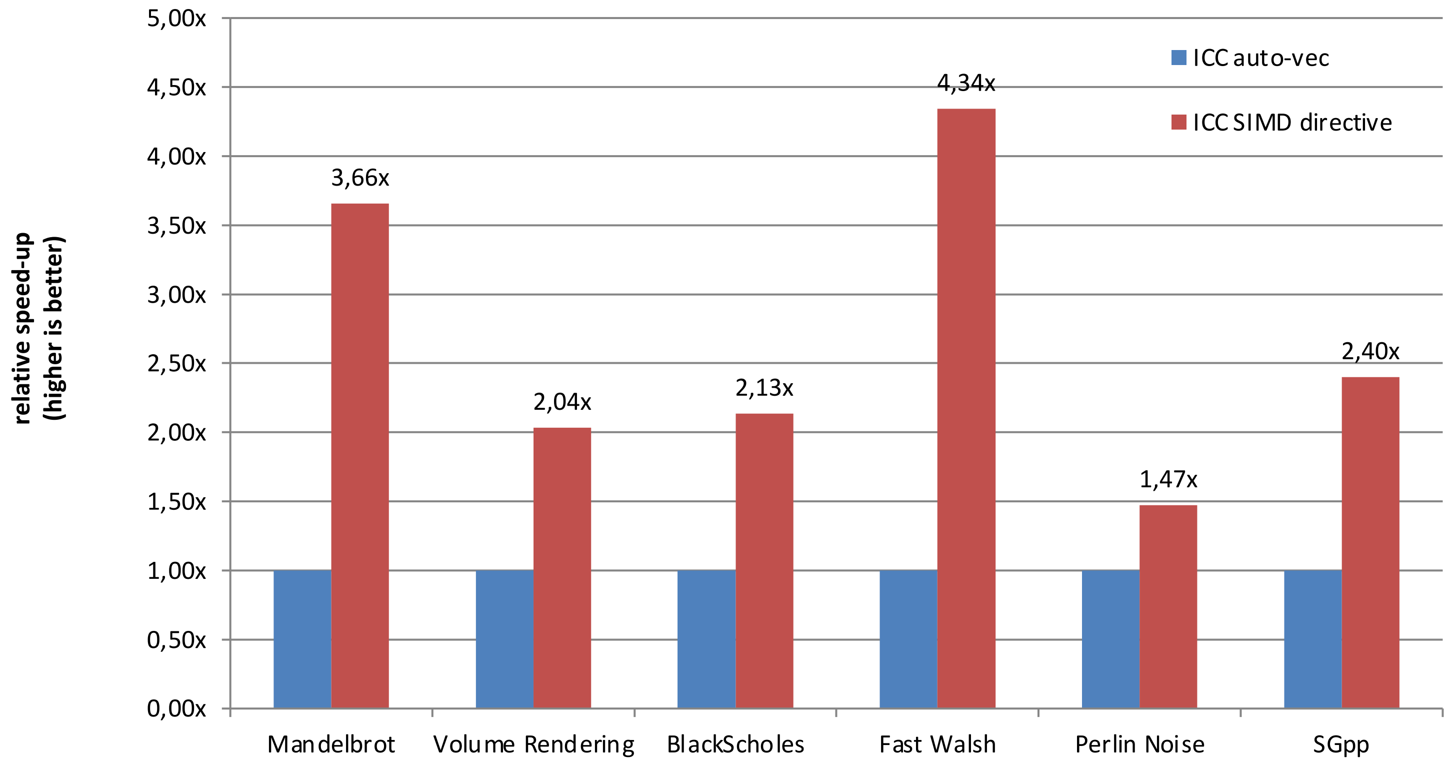
inbranch & notinbranch

```
#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

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 ☺)

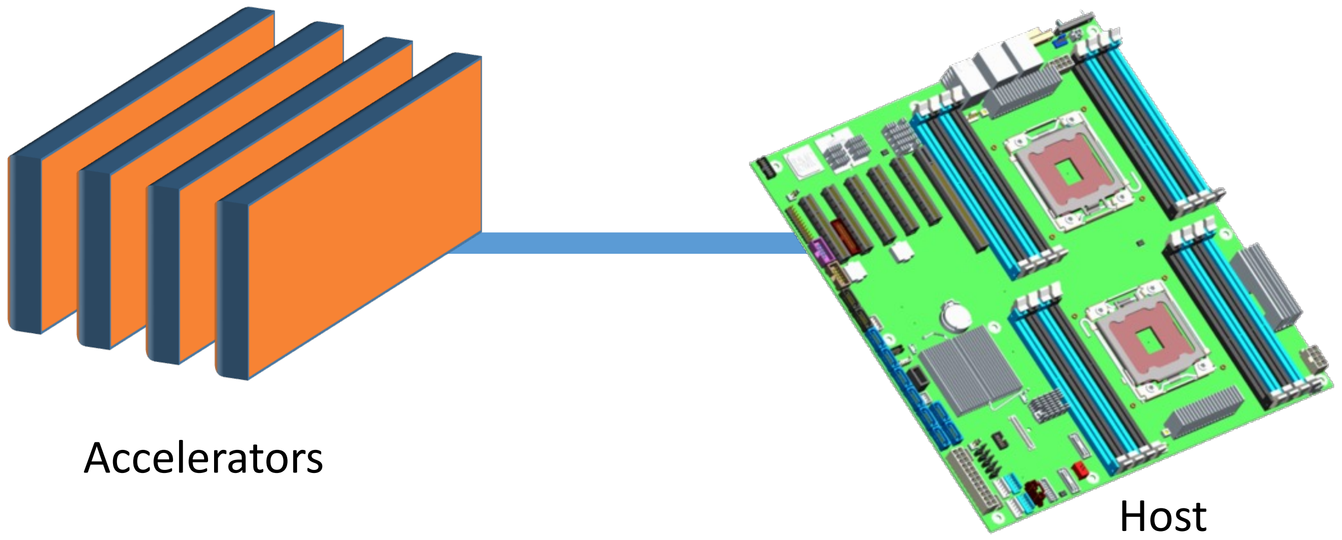
This is the code we want to execute on a target device (i.e., GPU)

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!

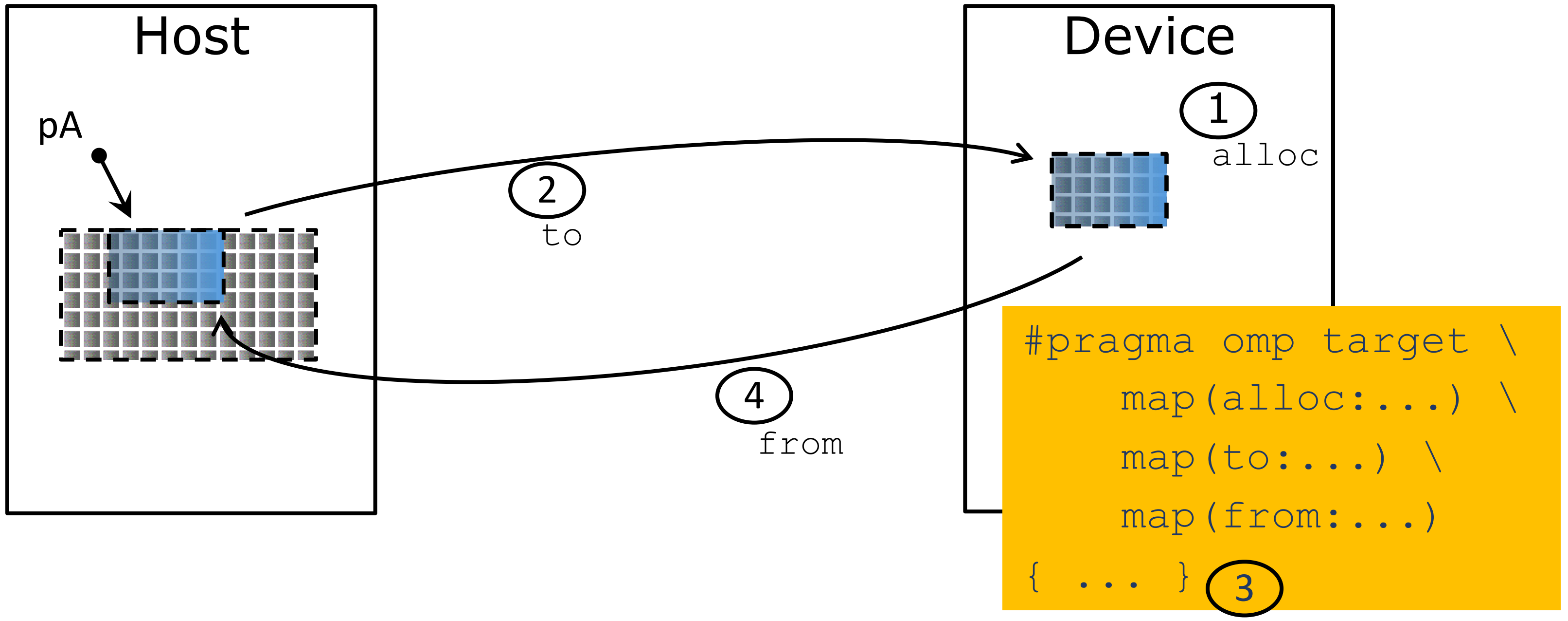
Device Model

- 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



Execution Model

- Offload region and data environment is lexically scoped
  - Data environment is destroyed at closing curly brace
  - Allocated buffers/data are automatically released



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

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

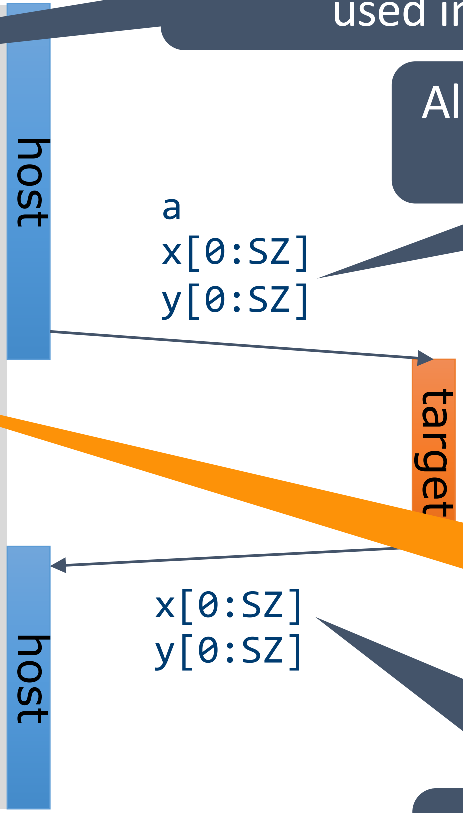
```
device(scalar-integer-expression)
map([{alloc | to | from | tofrom}:] list)
if(scalar-expr)
```

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.

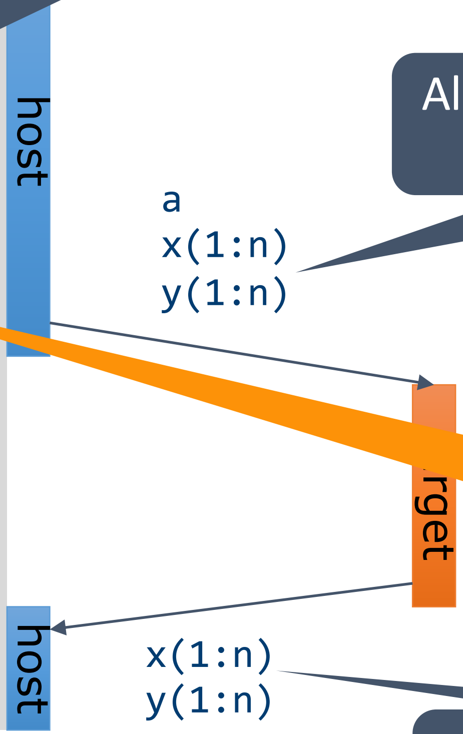
Copying x back is not necessary: it was not changed.

```
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

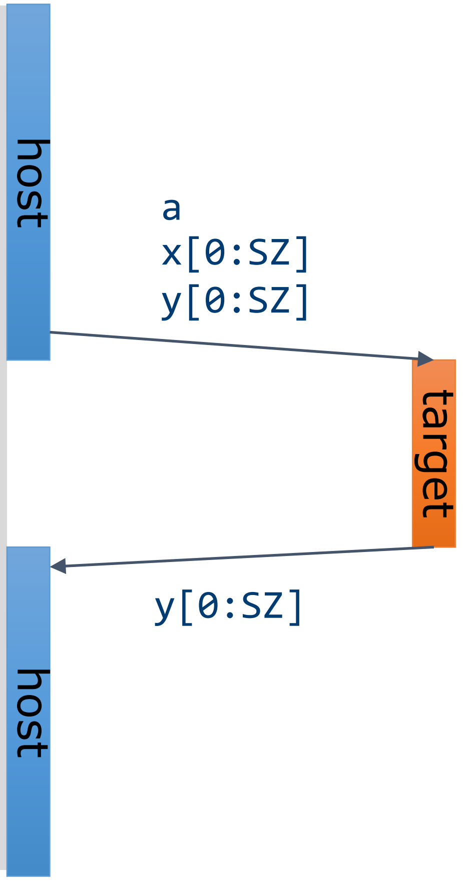
Presence check: only transfer if not yet allocated on the device.

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);
}
```

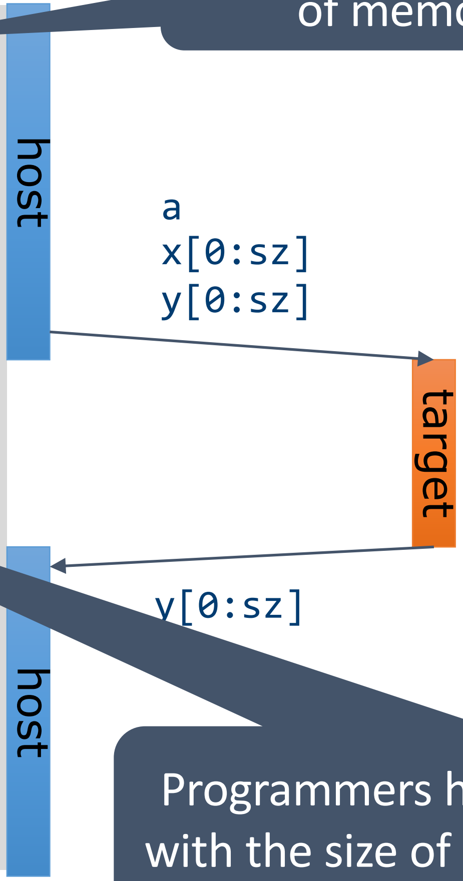


```
clang -fopenmp --offload-arch=gfx90a ...
```

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 ...
```

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 loops into an outer loop and an inner loop
  - Assign the outer loop to “teams” (OpenCL: work groups)
  - Assign the inner loop to the “threads” (OpenCL: work items)

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 n) {
    #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 < n; 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
```

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], a_in[N], b[N], c[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)

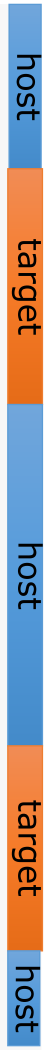
```
!$omp target update [clause[[,] clause],…]
```

- Clauses

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

Example: `target data` and `target update`

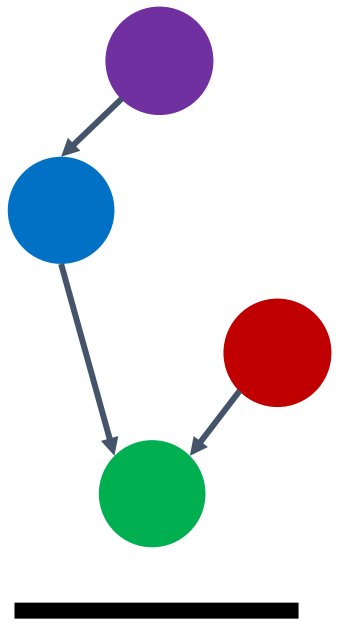
```
#pragma omp target data device(0) map(alloc:tmp[:N]) map(to:input[:N)) map(from:res)
  {
#pragma omp target device(0)
#pragma omp parallel for
    for (i=0; i<N; i++)
      tmp[i] = some_computation(input[i], i);

    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 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:y[:N]) map(to:x[:N])    nowait      depend(in:x) depend(in:y)
    compute_3(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:

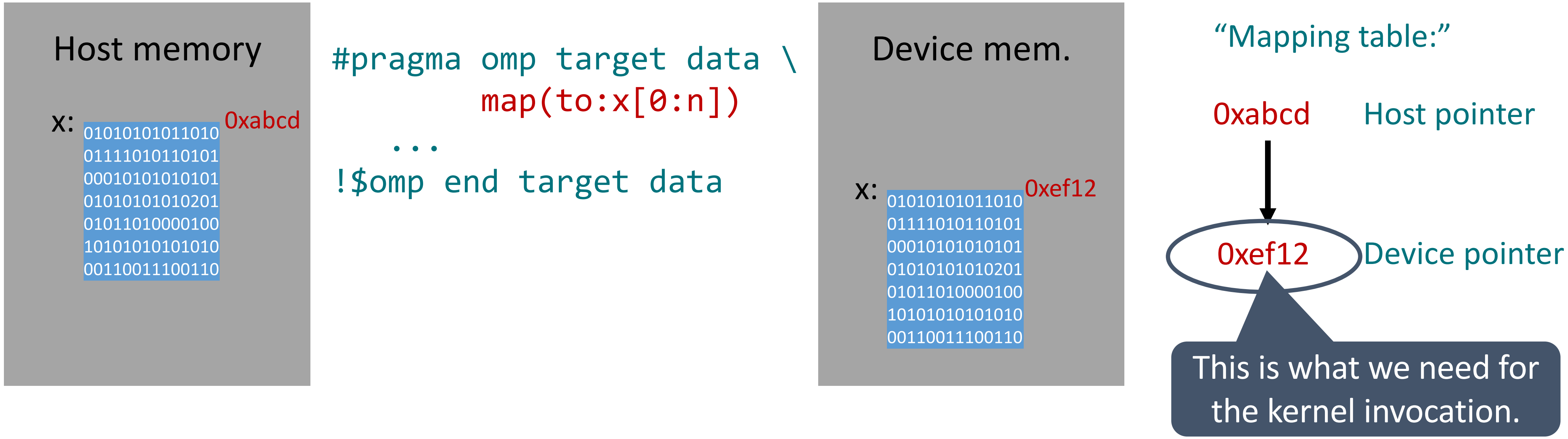
```
__global__ void saxpy_kernel(size_t n, float a, float * x, float * y) {
    size_t i = threadIdx.x + blockIdx.x * blockDim.x;
    y[i] = a * x[i] + y[i];
}

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[0:0])
{
    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[0:0],y[0:0])
        {
            saxpy_hip(n, a, x, y) // mapping table: x:[0xabcd,0xef12], x = 0xef12
        }
        compute_3(n, y);
    }
}
```

Summary

- OpenMP API is ready to use Intel discrete GPUs for offloading compute
    - Mature offload model w/ support for asynchronous offload/transfer
    - Tightly integrates with OpenMP multi-threading on the host
- More, advanced features (not covered here)
    - Memory management API
    - Interoperability with native data management
    - Interoperability with native streaming interfaces
    - Unified shared memory support

OpenMP®

Enabling HPC since 1997

Visit www.openmp.org for more information

Programming OpenMP

GPU: expressing parallelism

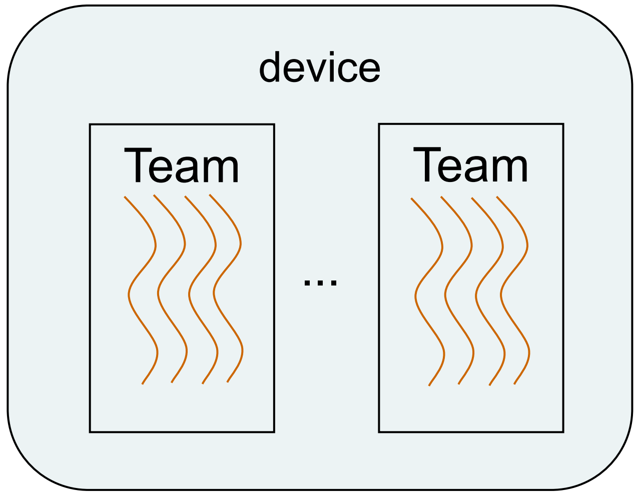
Christian Terboven
Michael Klemm

`teams` and `distribute` constructs

Many slides are taken from the lecture High-Performance Computing at RWTH Aachen University
Authors include: Sandra Wienke, Julian Miller

Terminology

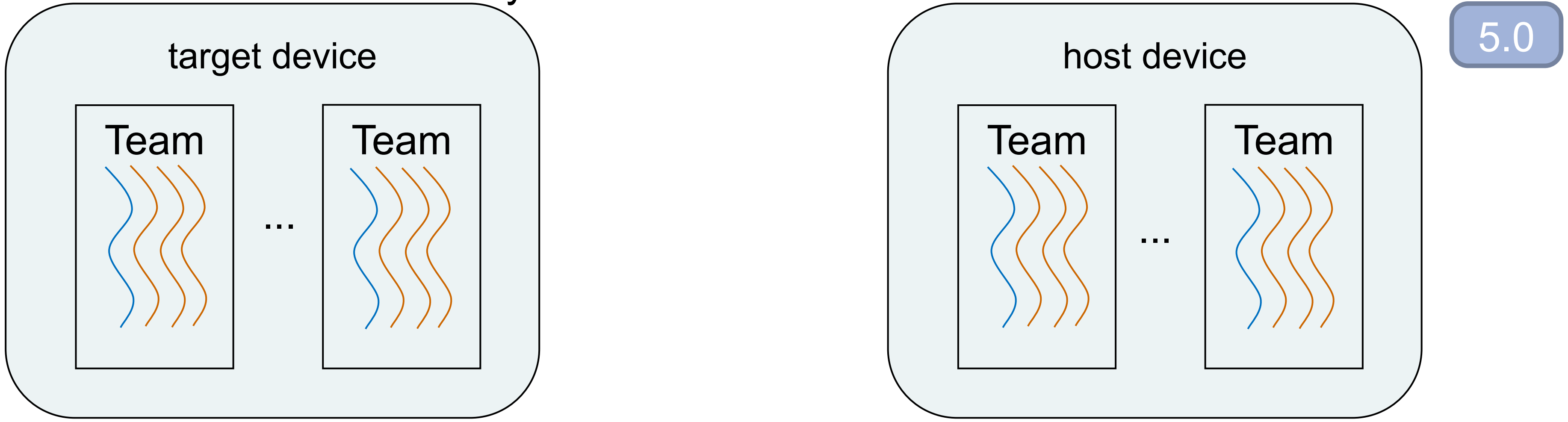
- League:
  the set of threads teams created by a `teams` construct
- Contention group:
  threads of a team in a league and their descendant threads



`teams` Construct

The **`teams`** construct creates a *league* of thread teams

- The master thread of each team executes the **`teams`** region
- The number of teams is specified by the **`num_teams`** clause
- Each team executes with **`thread_limit`** threads
- Threads in different teams cannot synchronize with each other



Only special OpenMP constructs or routines can be strictly nested inside a **`teams`** construct:

- **`distribute [simd]`**, distribute [parallel] worksharing-loop [SIMD]
- **`parallel`** regions (**`parallel for/do`**, **`parallel sections`**)
- **`omp_get_num_teams()`** and **`omp_get_team_num()`**

`distribute` Construct

- work sharing among the teams regions
  - Distribute the iterations of the associated loops across the master threads of each team executing the region

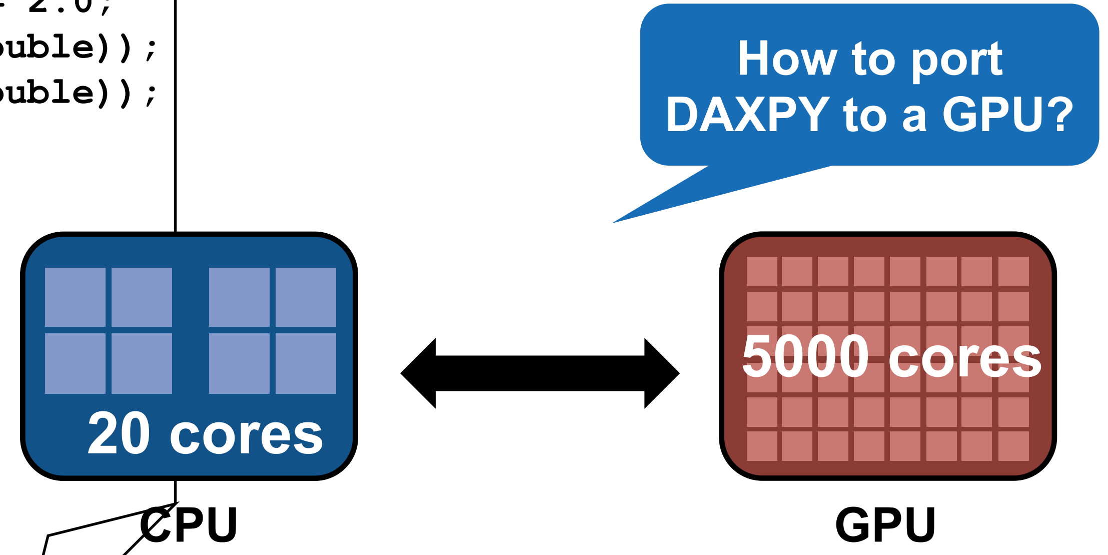
- Strictly nested inside a teams region

- No implicit barrier at the end of the construct

- **`dist_schedule(`***`kind[, chunk_size]`***`)`**
  - The scheduling kind must be **static**
  - Chunks are distributed in round-robin fashion of chunks with size *`chunk_size`*
  - If no chunk size specified, chunks are of (almost) equal size; each team receives at most one chunk

Example DAXPY: How to Port to GPU?

```
void daxpy(int n, double a, double *x, double *y) {
  for (int i = 0; i < n; ++i)
    y[i] = a * x[i] + y[i];
}

int main(int argc, const char* argv[]) {
  static int n = 100000000; static double a = 2.0;
  double *x = (double *) malloc(n * sizeof(double));
  double *y = (double *) malloc(n * sizeof(double));

  // Initialize x, y
  for(int i = 0; i < n; ++i){
    x[i] = 1.0;
    y[i] = 2.0;
  }
  daxpy(n, a, x, y); // Invoke daxpy kernel
  // Check if all values are 4.0

  free(x); free(y);
  return 0;
}
```



Kernel Directives

- Offload kernel code
  - **`target`**: offload work
  - **`teams, parallel`**: create in parallely running threads
  - **`distribute, do, for, simd`**: worksharing across parallel units

- Worksharing
  - **`for`**: offload work
  - **`collapse`**: collapse two or more nested loops to increase parallelism

Compilation

```
clang -fopenmp -Xopenmp-target -fopenmp-targets=nvptx64-nvidia-cuda -march=sm_70
--cuda-path=$CUDA_TOOLKIT_ROOT_DIR daxpy.c
```

`clang`	A recent clang compiler with OpenMP target support
`-fopenmp`	Enables general OpenMP support
`-Xopenmp-target`	Enables OpenMP target support
`-fopenmp-targets=nvptx64-nvidia-cuda`	Specifies the target architecture → here: NVIDIA GPUs
`-march=sm_70`	Optional. Specifies the target compute architecture
`--cuda-path=$CUDA_TOOLKIT_ROOT_DIR`	Optional. Specifies the CUDA path

Example: DAXPY

```
void daxpy(int n, double a, double *x, double *y) {
  #pragma omp target
  for (int i = 0; i < n; ++i)
    y[i] = a * x[i] + y[i];
}

int main(int argc, const char* argv[]) {
  static int n = 100000000; static double a = 2.0;
  double *x = (double *) malloc(n * sizeof(double));
  double *y = (double *) malloc(n * sizeof(double));

  // Initialize x, y
  for(int i = 0; i < n; ++i){
    x[i] = 1.0;
    y[i] = 2.0;
  }
  daxpy(n, a, x, y); // Invoke daxpy kernel
  // Check if all values are 4.0

  free(x); free(y);
  return 0;
}
```

Output:

```
$ $CC $FLAGS_OFFLOAD_OPENMP  daxpy.c
$ a.out
Libomptarget fatal error 1: failure of target
construct while offloading is mandatory
```

Example DAXPY: Debugging

- No compiler error but cryptic runtime error
- NVIDIA Profiler

```
$ nvprof daxpy.exe
==40419== NVPROF is profiling process 40419, command: daxpy.exe
==40419== Profiling application: daxpy.exe
==40419== Profiling result:
No kernels were profiled.

==40419== API calls:
No API activities were profiled.
```

- Cuda-memcheck

```
$ cuda-memcheck daxpy.exe
========= CUDA-MEMCHECK
========= Invalid __global__ read of size 8
=========     at 0x00000d10 in __omp_offloading_4b_f850d140_daxpy_l3
=========     by thread (32,0,0) in block (0,0,0)
=========     Address 0x00000000 is out of bounds
```

Example DAXPY: Data Management

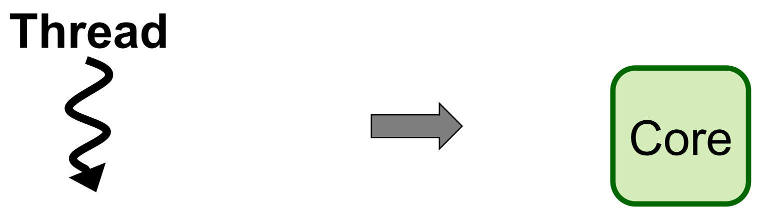
```c
void daxpy(int n, double a, double *x, double *y) {
  #pragma omp target map(tofrom:y[0:n]) map(to:a,x[0:n])
  for (int i = 0; i < n; ++i)
    y[i] = a * x[i] + y[i];
}

int main(int argc, const char* argv[]) {
  static int n = 100000000; static double a = 2.0;
  double *x = (double *) malloc(n * sizeof(double));
  double *y = (double *) malloc(n * sizeof(double));

  // Initialize x, y
  for(int i = 0; i < n; ++i){
    x[i] = 1.0;
    y[i] = 2.0;
  }
  daxpy(n, a, x, y); // Invoke daxpy kernel
  // Check if all values are 4.0

  free(x); free(y);
  return 0;
}
```

For comparison: ~0.12s on a single CPU core

Output:

```
$ $CC $FLAGS_OFFLOAD_OPENMP  daxpy.c
$ a.out
Max error: 0.00000
Total runtime: 102.50s
```

Mapping to Hardware



- Each thread is executed by a core

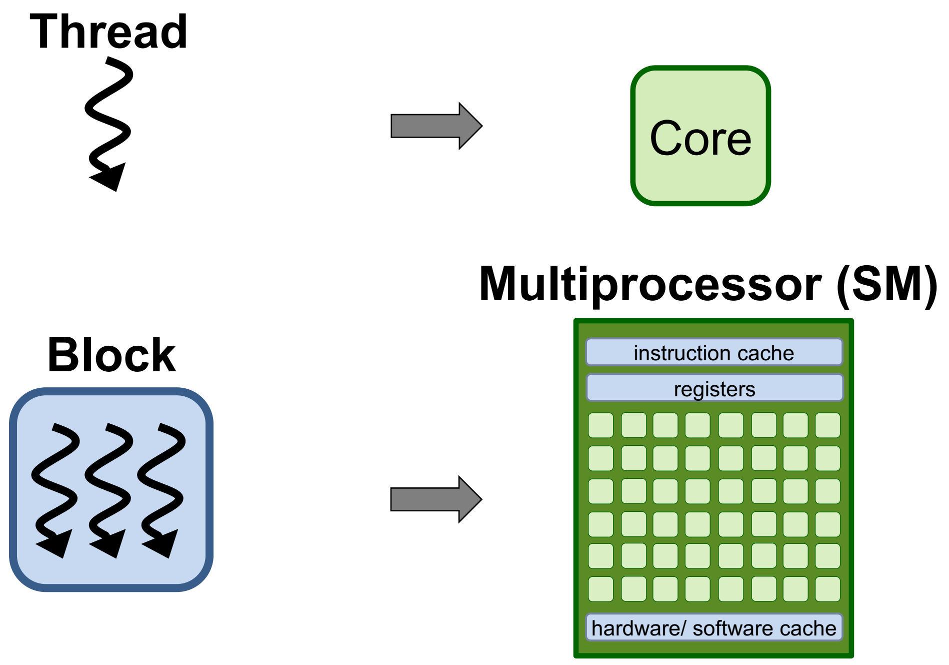
Example DAXPY: Thread Parallelism

```
void daxpy(int n, double a, double *x, double *y) {
  #pragma omp target parallel for map(tofrom:y[0:n]) map(to:a,x[0:n])
  for (int i = 0; i < n; ++i)
    y[i] = a * x[i] + y[i];
}

int main(int argc, const char* argv[]) {
  static int n = 100000000; static double a = 2.0;
  double *x = (double *) malloc(n * sizeof(double));
  double *y = (double *) malloc(n * sizeof(double));

  // Initialize x, y
  for(int i = 0; i < n; ++i){
    x[i] = 1.0;
    y[i] = 2.0;
  }
  daxpy(n, a, x, y); // Invoke daxpy kernel
  // Check if all values are 4.0

  free(x); free(y);
  return 0;
}
```

Output:

```
$ $CC $FLAGS_OFFLOAD_OPENMP  daxpy.c
$ a.out
Max error: 0.00000
Total runtime: 9.65s
```

Mapping to Hardware



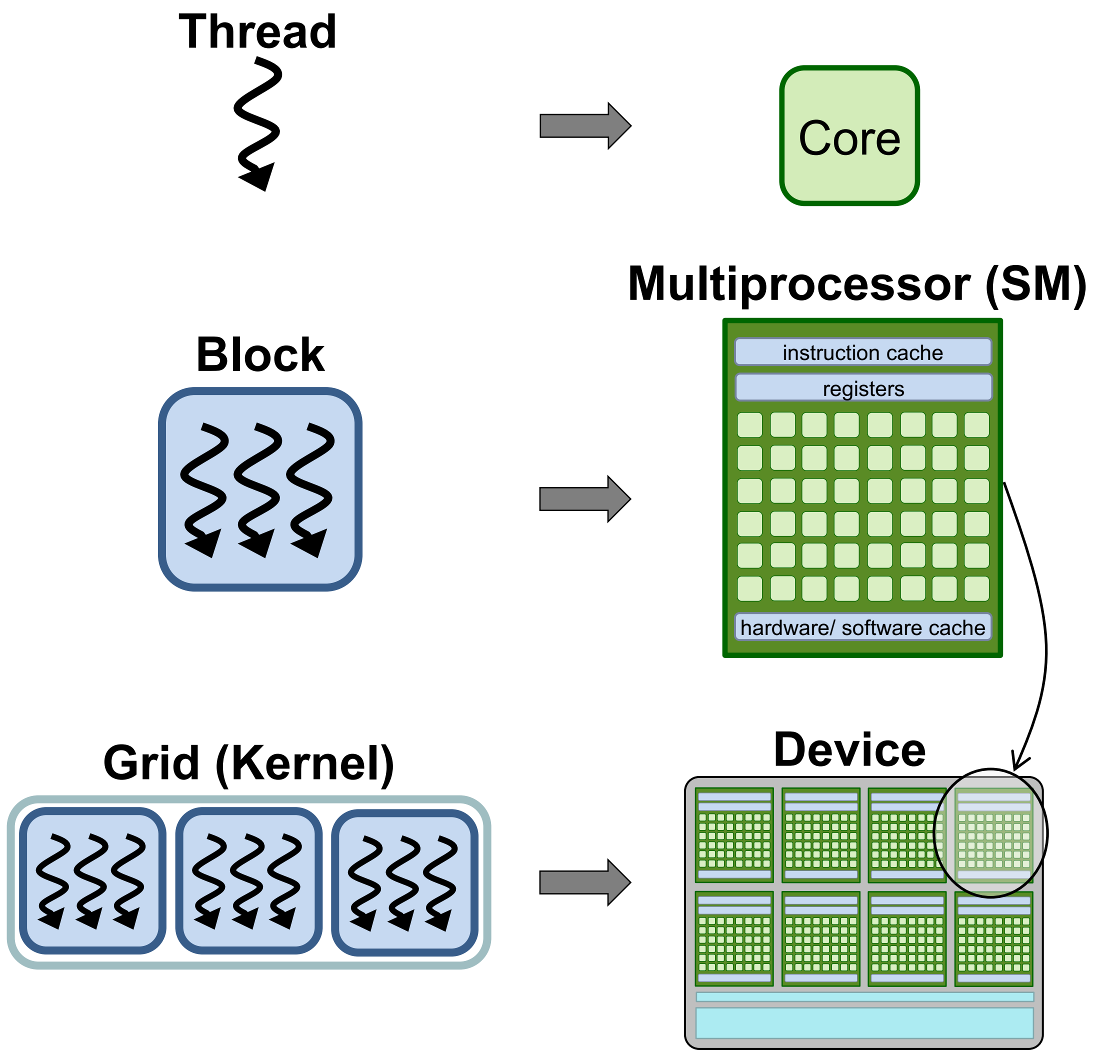
- Each thread is executed by a core
- Each block is executed on a SM
- Several concurrent blocks can reside on a SM depending on shared resources

Example DAXPY: Thread Parallelism

```
void daxpy(int n, double a, double *x, double *y) {
  #pragma omp target teams distribute parallel for map(tofrom:y[0:n]) map(to:a,x[0:n])
for (int i = 0; i < n; ++i)
    y[i] = a * x[i] + y[i];
}

int main(int argc, const char* argv[]) {
  static int n = 100000000; static double a = 2.0;
  double *x = (double *) malloc(n * sizeof(double));
  double *y = (double *) malloc(n * sizeof(double));

  // Initialize x, y
  for(int i = 0; i < n; ++i){
    x[i] = 1.0;
    y[i] = 2.0;
  }
  daxpy(n, a, x, y); // Invoke daxpy kernel
  // Check if all values are 4.0

  free(x); free(y);
  return 0;
}
```

```
Output:
$ $CC $FLAGS_OFFLOAD_OPENMP  daxpy.c
$ a.out
Max error: 0.00000
Total runtime: 0.80s
```

Mapping to Hardware



- Each thread is executed by a core
- Each block is executed on a SM
- Several concurrent blocks can reside on a SM depending on shared resources
- Each kernel is executed on a device

`teams` Construct

- 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 | none)
```
OR
```
default(shared|private|firstprivate|none)
private(list)
firstprivate(list)
shared(list)
reduction([default,]reduction-identifier : list)
allocate([allocator:]list)
```
5.0

`distribute` Construct

- Syntax (C/C++):

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

- Syntax (Fortran):

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

- Clauses

```
private(list)
firstprivate(list)
lastprivate(list)
collapse(n)
dist_schedule(kind[, chunk_size])
allocate([allocator:]list)
```

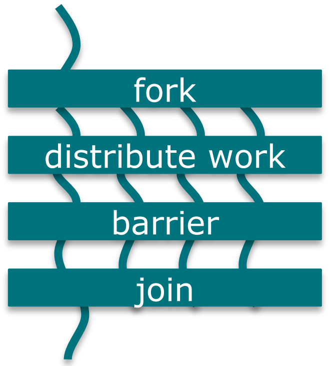
5.0

OpenMP Parallel Loops

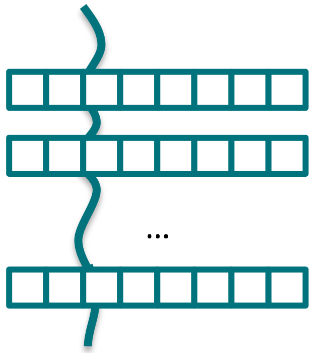
`loop` Construct

- Existing loop constructs are tightly bound to execution model:

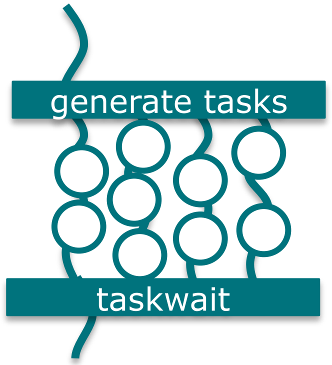
```
#pragma omp parallel for
for (i=0; i<N;++i) {…}
```



```
#pragma omp simd
for (i=0; i<N;++i) {…}
```



```
#pragma omp taskloop
for (i=0; i<N;++i) {…}
```



- The **`loop`** construct is meant to tell OpenMP about truly parallel semantics of a loop.

OpenMP Fully Parallel Loops

```
int main(int argc, const char* argv[]) {
    float *x = (float*) malloc(n * sizeof(float));
    float *y = (float*) malloc(n * sizeof(float));
    // Define scalars n, a, b & initialize x, y

#pragma omp parallel
#pragma omp loop
    for (int i = 0; i < n; ++i){
      y[i] = a*x[i] + y[i];
    }
  }
}
```

`loop` Constructs, Syntax

- Syntax (C/C++)

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

- Syntax (Fortran)

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

`loop` Constructs, Clauses

- `bind(`*`binding`*`)`
  - → Binding region the loop construct should bind to
  - → One of: `teams`, `parallel`, `thread`

- `order(concurrent)`
  - → Tell the OpenMP compiler that the loop can be executed in any order.
  - → Default!

- `collapse(`*`n`*`)`
- `private(`*`list`*`)`
- `lastprivate(`*`list`*`)`
- `reduction(`*`reduction-id:list`*`)`

Extensions to Existing Constructs

- Existing loop constructs have been extended to also have truly parallel semantics.

- C/C++ Worksharing:

```
#pragma omp [for|simd] order(concurrent) \
                       [clause[[,] clause],…]
for-loops
```

- Fortran Worksharing:

```
!$omp [do|simd] order(concurrent) &
                [clause[[,] clause],…]
do-loops
[!$omp end [do|simd}]
```

DOACROSS Loops

DOACROSS Loops

- “DOACROSS” loops are loops with special loop schedules
  - → Restricted form of loop-carried dependencies
  - → Require fine-grained synchronization protocol for parallelism

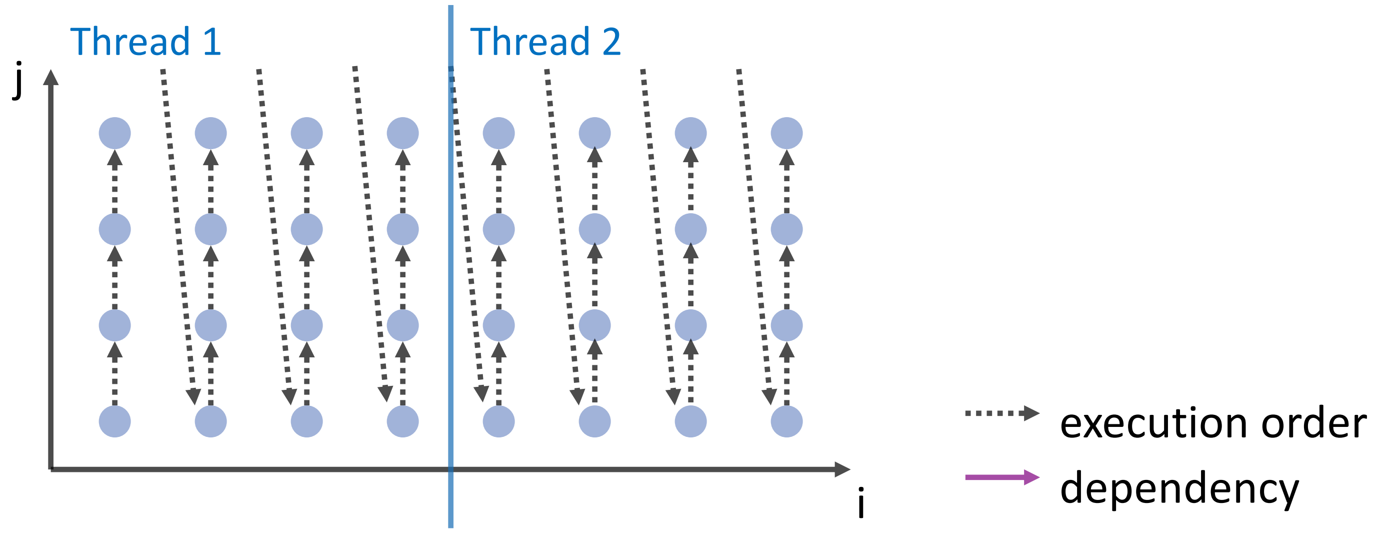
- Loop-carried dependency:
  - → Loop iterations depend on each other
  - → Source of dependency must scheduled before sink of the dependency

- DOACROSS loop:
  - → Data dependency is an invariant for the execution of the whole loop nest

Parallelizable Loops

- A parallel loop cannot not have any loop-carried dependencies (simplified just a little bit!)

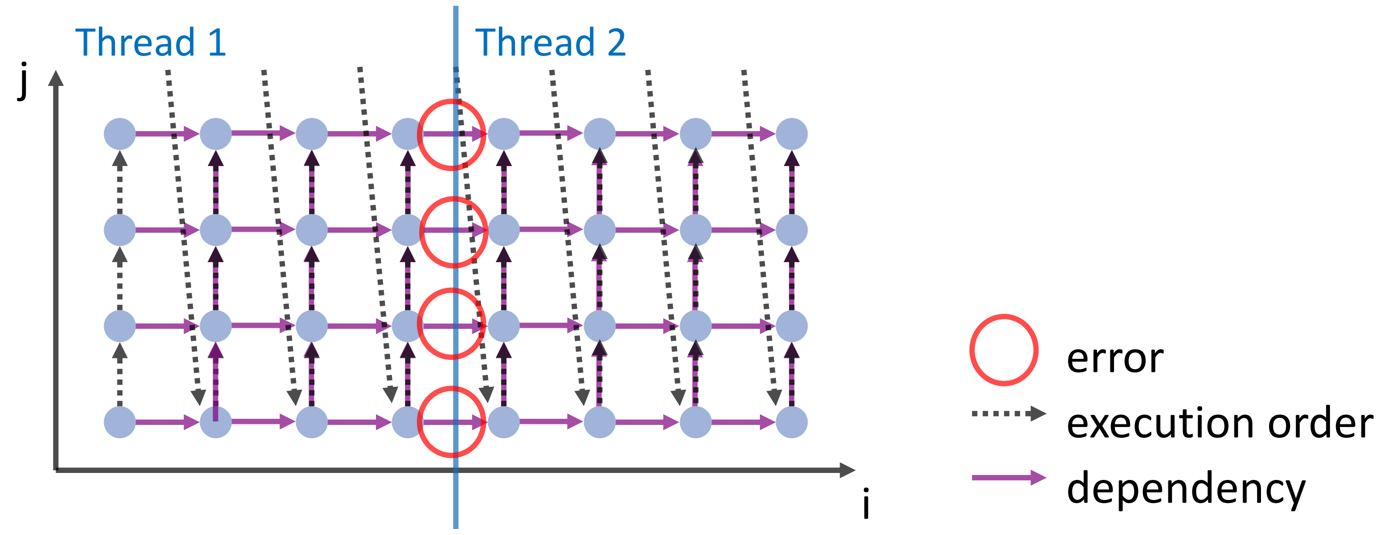
```
for (int i = 1; i < N; ++i) {
    for (int j = 1; j < M; ++j) {
        b[i][j] = f(b[i][j],
                    b[i][j], a[i][j]);
    }
}
```



Non-parallelizable Loops

- If there is a loop-carried dependency, a loop cannot be parallelized anymore ("easily" that is)

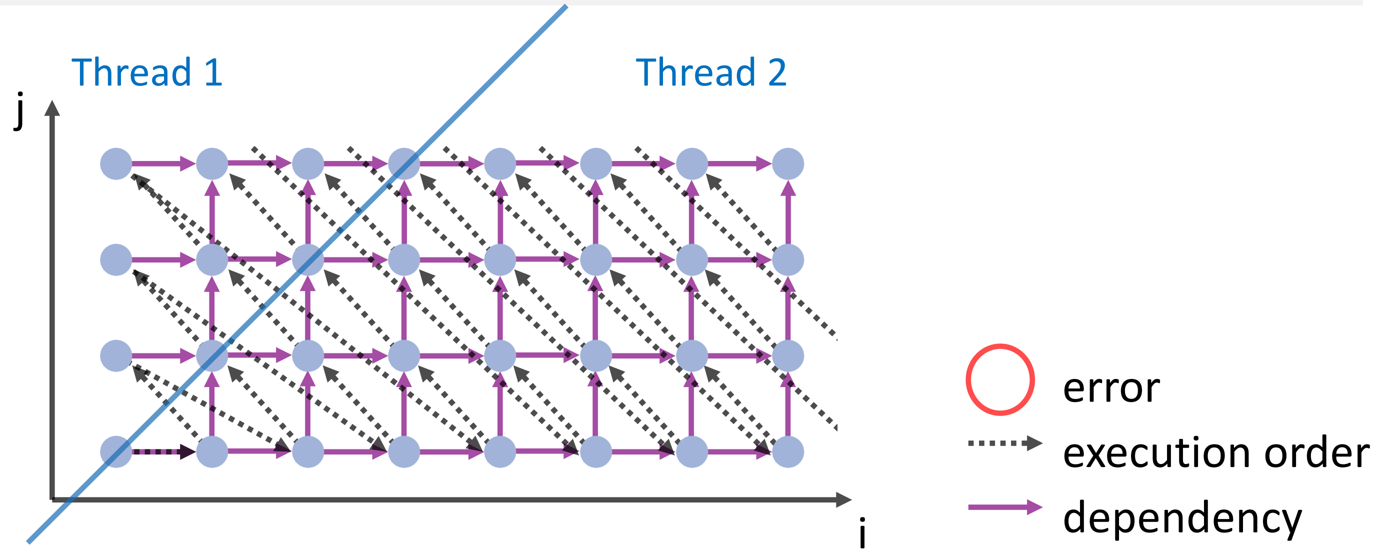
```
for (int i = 1; i < N; ++i) {
    for (int j = 1; j < M; ++j) {
        b[i][j] = f(b[i-1][j],
                    b[i][j-1], a[i][j]);
    }
}
```



Wavefront-Parallel Loops

- If the data dependency is invariant, then skewing the loop helps remove the data dependency

```
for (int i = 1; i < N; ++i) {
    for (int j = i+1; j < i+N; ++j) {
        b[i][j-i] = f(b[i-1][j-i],
                      b[i][j-i-1], a[i][j]);
    }
}
```



DOACROSS Loops with OpenMP

Deprecated in v5.2

- OpenMP 4.5 extends the notion of the ordered construct to describe loop-carried dependencies
- Syntax (C/C++):

```
#pragma omp for ordered(d) [clause[[,] clause],…]
for-loops
```

and

```
#pragma omp ordered [clause[[,] clause],…]
```

where clause is one of the following:

```
  depend(source)
  depend(sink:vector)
```

- Syntax (Fortran):

```
!$omp do ordered(d) [clause[[,] clause],…]
do-loops
!$omp ordered [clause[[,] clause],…]
```

Example

Deprecated in v5.2

- The ordered clause tells the compiler about loop-carried dependencies and their distances

```
#pragma omp parallel for ordered(2)
for (int i = 1; i < N; ++i) {
    for (int j = 1; j < M; ++j) {
#pragma omp ordered depend(sink:i-1,j) depend(sink:i,j-1)
        b[i][j] = f(b[i-1][j],
                    b[i][j-1], a[i][j]);
    }
#pragma omp ordered depend(source)
}
```

Example: 3D Gauss-Seidel

Deprecated in v5.2

```
#pragma omp for ordered(2) private(j,k)
for (i = 1; i < N-1; ++i) {
  for (j = 1; j < N-1; ++j)   {
#pragma omp ordered depend(sink: i-1,j-1) depend(sink: i-1,j) \
                    depend(sink: i-1,j+1) depend(sink: i,j-1)
    for (k = 1; k < N-1; ++k) {
      double tmp1 = (p[i-1][j-1][k-1] + p[i-1][j-1][k] + p[i-1][j-1][k+1]
                    + p[i-1][j][k-1] + p[i-1][j][k] + p[i-1][j][k+1]
                    + p[i-1][j+1][k-1] + p[i-1][j+1][k] + p[i-1][j+1][k+1]);
      double tmp2 = (p[i][j-1][k-1] + p[i][j-1][k] + p[i][j-1][k+1]
                    + p[i][j][k-1] + p[i][j][k] + p[i][j][k+1]
                    + p[i][j+1][k-1] + p[i][j+1][k] + p[i][j+1][k+1]);
      double tmp3 = (p[i+1][j-1][k-1] + p[i+1][j-1][k] + p[i+1][j-1][k+1]
                    + p[i+1][j][k-1] + p[i+1][j][k] + p[i+1][j][k+1]
                    + p[i+1][j+1][k-1] + p[i+1][j+1][k] + p[i+1][j+1][k+1]);
      p[i][j][k] = (tmp1 + tmp2 + tmp3) / 27.0;
    }
#pragma omp ordered depend(source)
  }
}
```

DOACROSS Loops with OpenMP

- OpenMP 4.5 extends the notion of the ordered construct to describe loop-carried dependencies
- Syntax (C/C++):

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

  and

```
#pragma omp ordered [clause[[,] clause],…]
```

  where clause is one of the following:

  `doaccross(source:vector)`, vector can be `omp_cur_iteration`
  `doaccross(sink:vector)`

- Syntax (Fortran):

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

Example

- The ordered clause tells the compiler about loop-carried dependencies and their distances

```
#pragma omp parallel for ordered
for (int i = 1; i < N; ++i) {
    for (int j = 1; j < M; ++j) {
#pragma omp ordered doacross(sink:i-1,j) doacross(sink:i,j-1)
        b[i][j] = f(b[i-1][j],
                    b[i][j-1], a[i][j]);
    }
#pragma omp ordered doacross(source:omp_cur_iteration)
}
```

Example: 3D Gauss-Seidel

```
#pragma omp for ordered private(j,k)
for (i = 1; i < N-1; ++i) {
  for (j = 1; j < N-1; ++j)   {
#pragma omp ordered doacross(sink: i-1,j-1) doacross(sink: i-1,j) \
                    doacross(sink: i-1,j+1) doacross(sink: i,j-1)
    for (k = 1; k < N-1; ++k) {
      double tmp1 = (p[i-1][j-1][k-1] + p[i-1][j-1][k] + p[i-1][j-1][k+1]
                   + p[i-1][j][k-1] + p[i-1][j][k] + p[i-1][j][k+1]
                   + p[i-1][j+1][k-1] + p[i-1][j+1][k] + p[i-1][j+1][k+1]);
      double tmp2 = (p[i][j-1][k-1] + p[i][j-1][k] + p[i][j-1][k+1]
                   + p[i][j][k-1] + p[i][j][k] + p[i][j][k+1]
                   + p[i][j+1][k-1] + p[i][j+1][k] + p[i][j+1][k+1]);
      double tmp3 = (p[i+1][j-1][k-1] + p[i+1][j-1][k] + p[i+1][j-1][k+1]
                   + p[i+1][j][k-1] + p[i+1][j][k] + p[i+1][j][k+1]
                   + p[i+1][j+1][k-1] + p[i+1][j+1][k] + p[i+1][j+1][k+1]);
      p[i][j][k] = (tmp1 + tmp2 + tmp3) / 27.0;
    }
#pragma omp ordered depend(source:omp_cur_iteration)
  }
}
```

OpenMP Meta-Programming

The `metadirective` Directive

- Construct OpenMP directives for different OpenMP contexts
- Limited form of meta-programming for OpenMP directives and clauses

```
#pragma omp target map(to:v1,v2) map(from:v3)
#pragma omp metadirective \
            when( device={arch(nvptx)}: teams loop ) \
            default( parallel loop )
for (i = lb; i < ub; i++)
   v3[i] = v1[i] * v2[i];
```

```
!$omp begin metadirective &
            when( implementation={unified_shared_memory}: target ) &
            default( target map(mapper(vec_map),tofrom: vec) )
!$omp teams distribute simd
do i=1, vec%size()
   call vec(i)%work()
end do
!$omp end teams distribute simd
!$omp end metadirective
```

Nothing Directive

The nothing Directive

- The `nothing` directive makes meta programming a bit clearer and more flexible.
- If a certain criterion matches, the nothing directive can stand to indicate that no (other) OpenMP directive should be used.
  - → The `nothing` directive is implicitly added if no condition matches

```
!$omp begin metadirective &
            when( implementation={unified_shared_memory}: &
                  target teams distribute parallel do simd) &
            default( nothing )
do i=1, vec%size()
   call vec(i)%work()
end do
!$omp end metadirective
```

Error Directive

`Error` Directive Syntax

- Syntax (C/C++)
  `#pragma omp error` *`[clause[[,] clause],…]`*
  `for-loops`

- Syntax (Fortran)
  `!$omp error` *`[clause[[,] clause],…]`*
  `do-loops`
  `[!$omp end loop]`

- Clauses
  one of: `at(compilation)`, `at(runtime)`
  one of: `severity(fatal)`, `severity(warning)`
  `message(`*`msg-string`*`)`

Error Directive

- Can be used to issue a warning or an error at compile time and runtime.
- Consider this a "directive version" of `assert()`, but with a bit more flexibility.

```
#pragma omp parallel
{
    if (omp_get_num_threads() % 2) {
#pragma omp error at(runtime) severity(warning) \
                  message("Running on odd number of threads\n");
    }
    do_stuff_that_works_best_with_even_thread_count();
}
```

Error Directive

- Can be used to issue a warning or an error at compile time and runtime.
- Consider this a “directive version” of `assert()`, but with a bit more flexibility.
- More useful in combination with OpenMP metadirective

```
!$omp begin metadirective &
            when( arch={fancy_processor}: parallel ) &
            default( error severity(fatal) at(compilation) &
                           message(“No implementation available” )
    call fancy_impl_for_fancy_processor()
!$omp end metadirective
```

Programming OpenMP

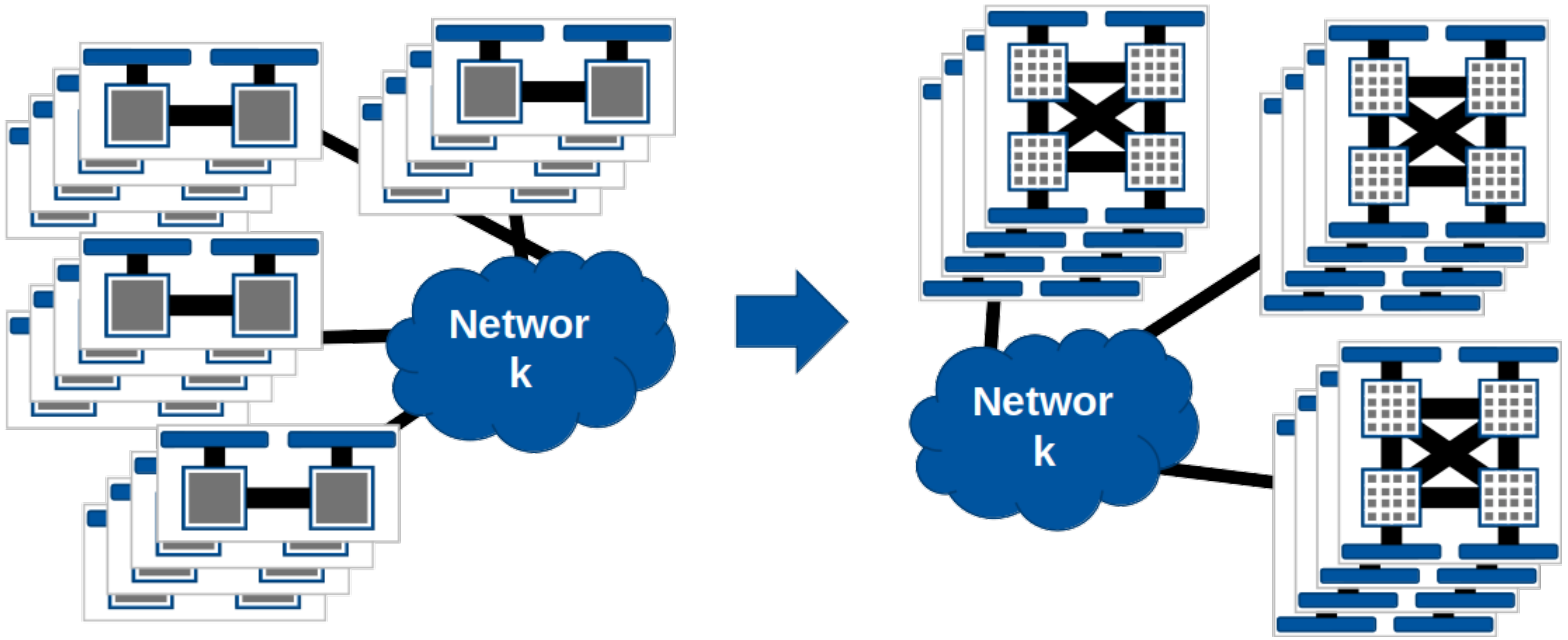
OpenMP and MPI

Christian Terboven
Michael Klemm

Motivation

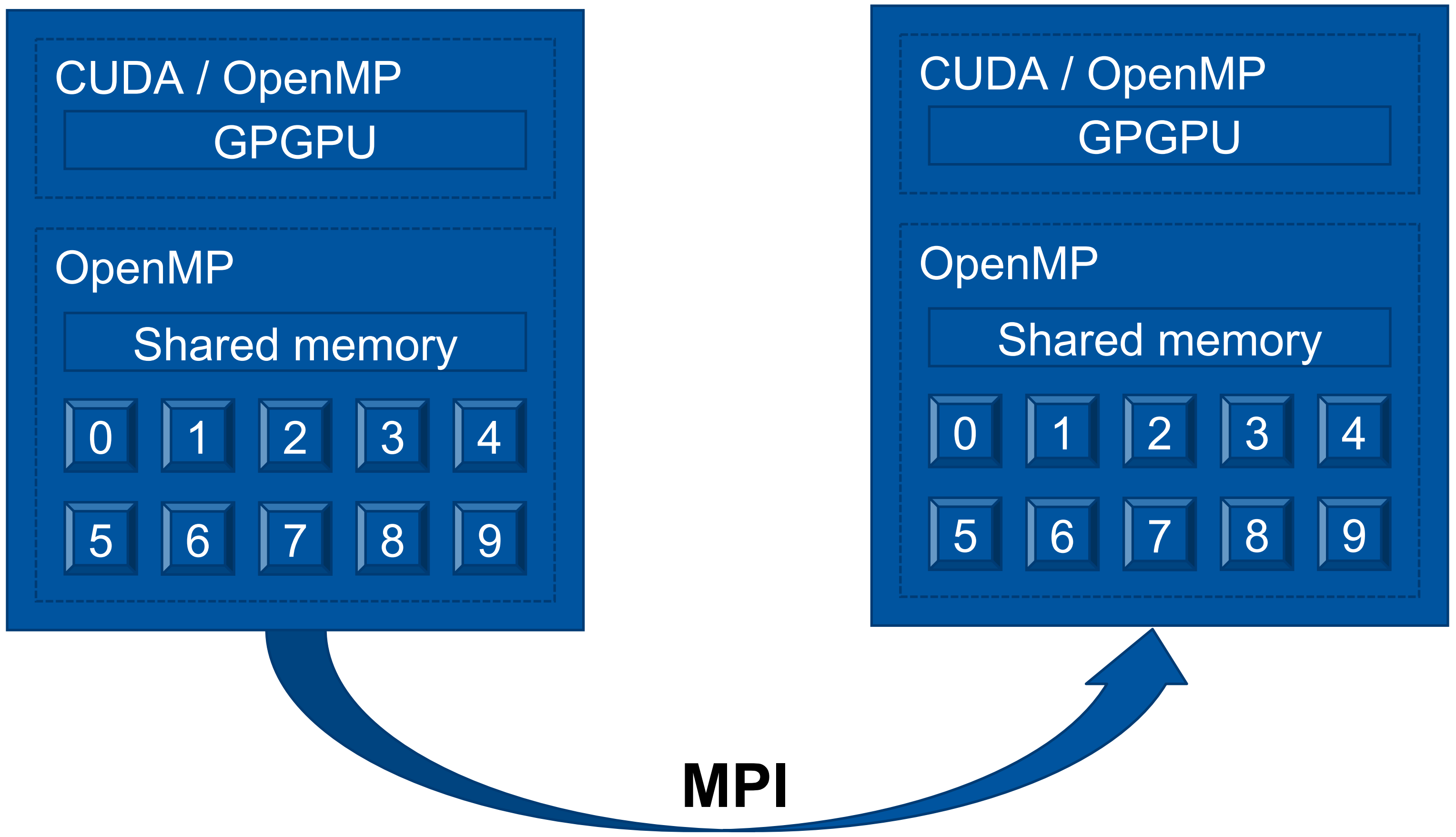
Motivation for hybrid programming

- Increasing number of cores per node



Hybrid programming

- (Hierarchical) mixing of different programming paradigms

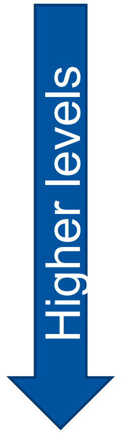


MPI and OpenMP

MPI – threads interaction

- MPI needs special initialization in a threaded environment
  - Use `MPI_Init_thread` to communicate thread support level

- Four levels of threading support



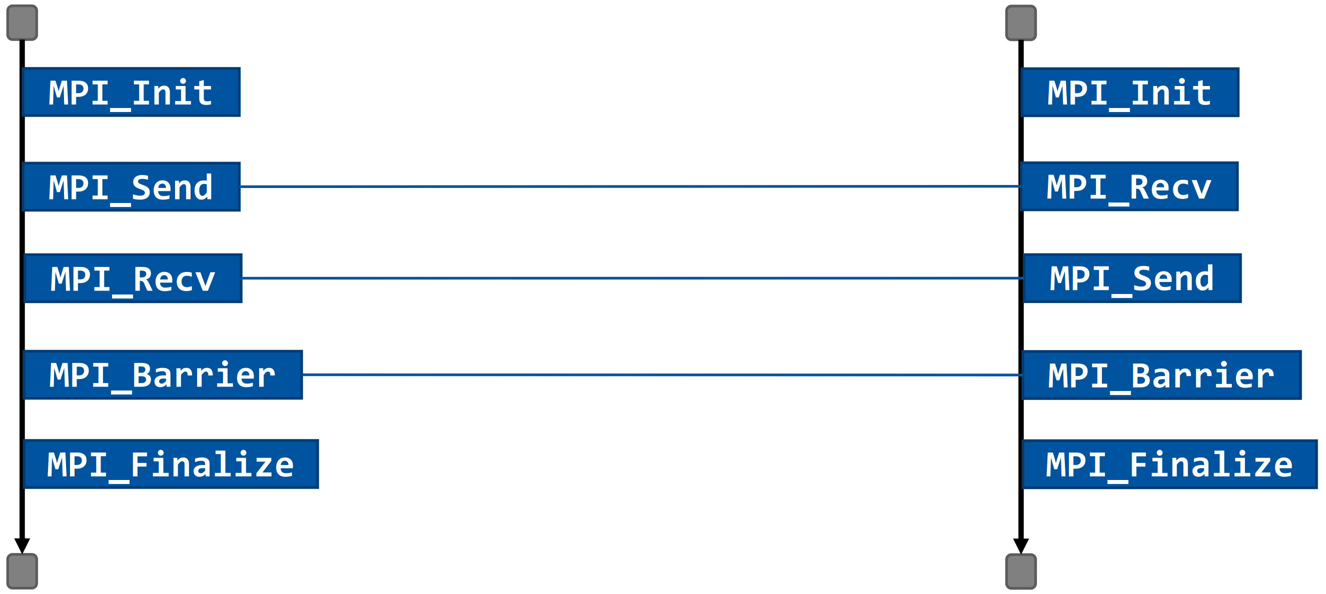
Level identifier	Description
`MPI_THREAD_SINGLE`	Only one thread may execute
`MPI_THREAD_FUNNELED`	Only the main thread may make MPI calls
`MPI_THREAD_SERIALIZED`	Any one thread may make MPI calls at a time
`MPI_THREAD_MULTIPLE`	Multiple threads may call MPI concurrently with no restrictions

- `MPI_THREAD_MULTIPLE` may incur significant overhead inside an MPI implementation

MPI – Threading support levels

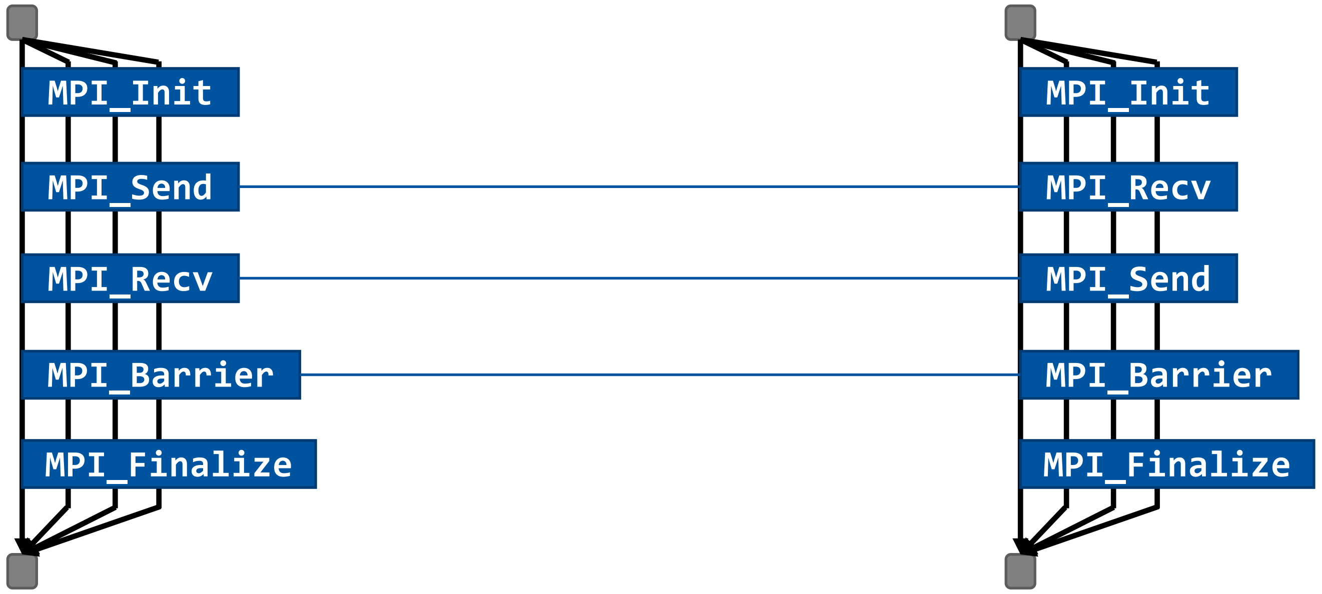
- MPI_THREAD_SINGLE
  - Only one thread per MPI rank

— MPI Communication
— Thread Synchronization



MPI – Threading support levels

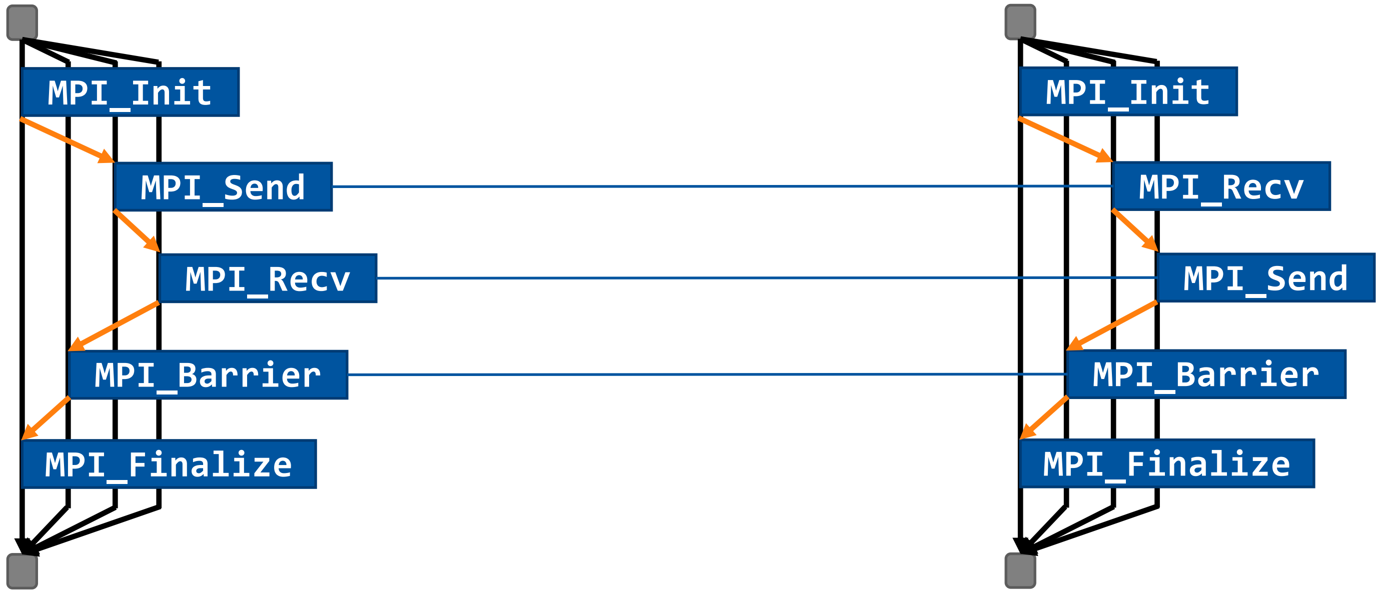
- MPI_THREAD_FUNNELED
  - Only one thread communicates



MPI – Threading support levels

- MPI_THREAD_SERIALIZED
  - Only one thread communicates at a time

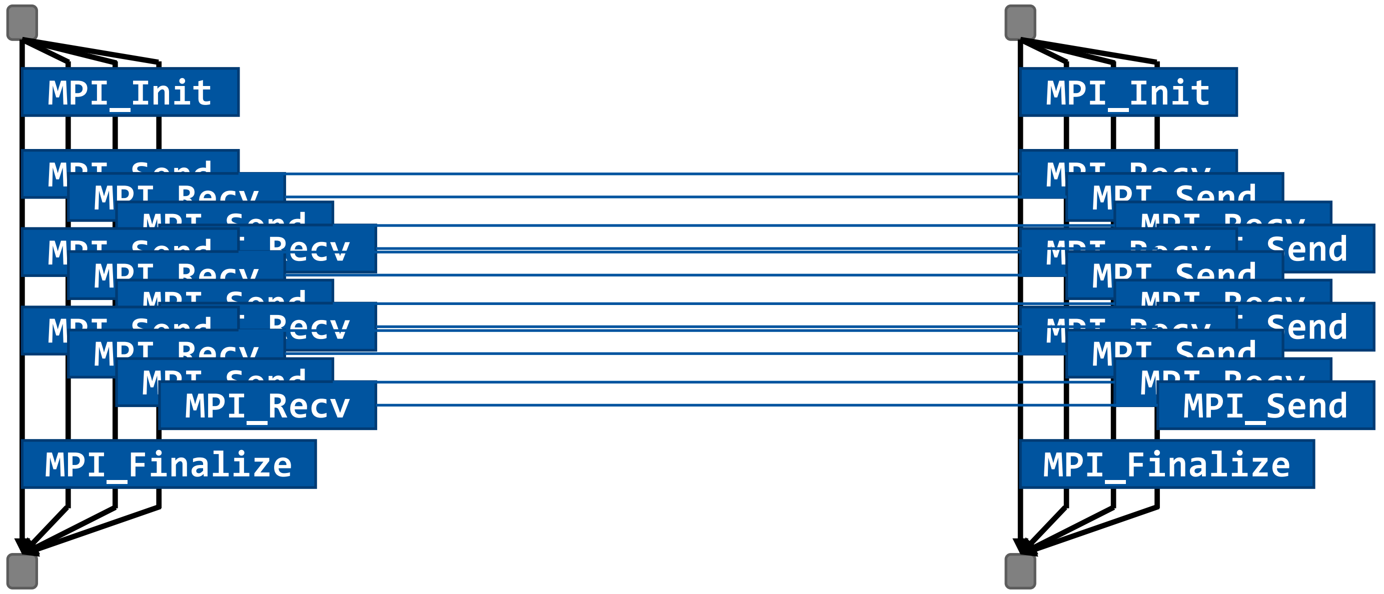
─	MPI Communication
─	Thread Synchronization



MPI – Threading support levels

- MPI_THREAD_MULTIPLE
  - All threads communicate concurrently without synchronizatio

MPI Communication
Thread Synchronization


MPI_Init
MPI_Send
MPI_Recv
MPI_Send
MPI_Recv
MPI_Finalize
MPI_Init
MPI_Recv
MPI_Send
MPI_Recv
MPI_Send
MPI_Finalize