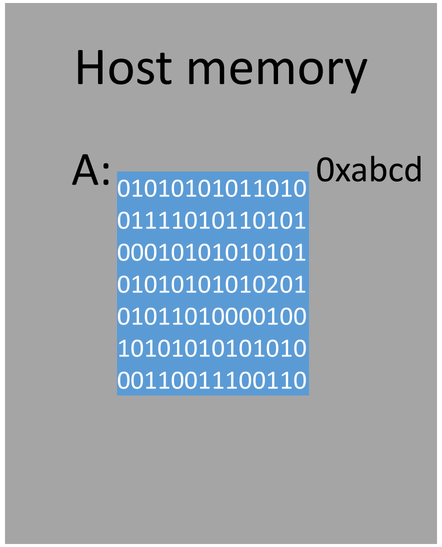
OpenMP Offloading

Data Management and Asynchronous Execution

Dr.-Ing. Michael Klemm
Chief Executive Officer
OpenMP Architecture Review Board

OpenMP Execution Model for Devices

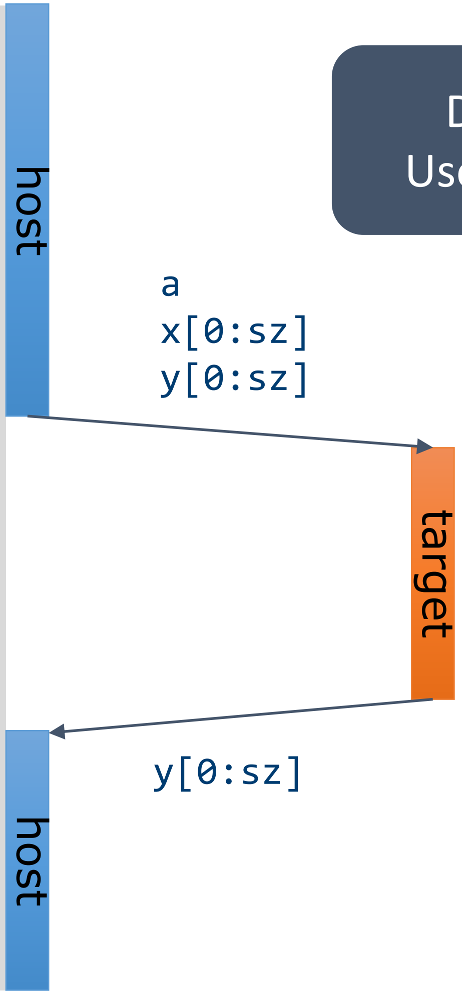
- Offload region and its data environment are bound to the lexical scope of the construct
  - Data environment is created at the opening curly brace
  - Data environment is automatically destroyed at the closing curly brace
  - Data transfers (if needed) are done at the curly braces, too:
    - Upload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

Example: saxpy

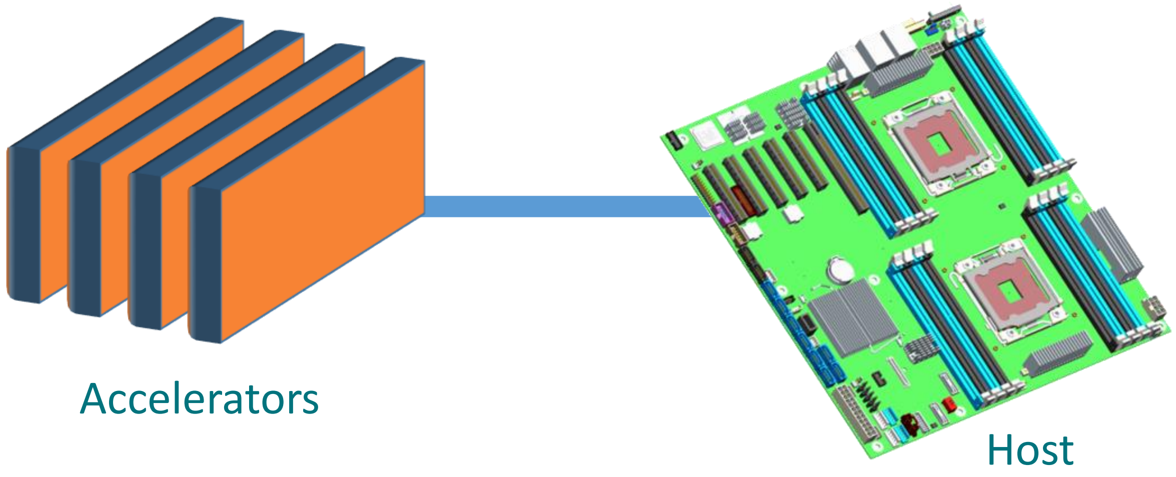
```c
void saxpy(float a, float* x, float* y,
           int sz) {
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
#pragma omp target teams \
            distribute parallel for \
            map(to:x[0:sz]) \
            map(tofrom:y[0:sz])
    for (int i = 0; i < sz; i++) {
       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



Don’t do this at home!
Use a BLAS library instead!

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.

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];
}
```

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

```
!$omp target enter data alloc(triplesx(1:tr_size))
c     for all tiles
      do ...
        call zero_triplesx(triplesx)
        do ...
          call comm_and_sort(t2sub, v2sub)
!$omp target data map(to:t2sub(t2_size)) map(to:v2sub(v2_size))
          if (...)
            call sd_t_d1_1(h3d,h2d,h1d,p6d,p5 ... 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 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,p5  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(h2h3
     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.

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)

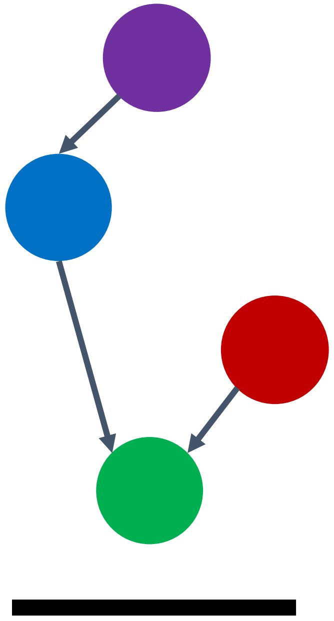
```
#pragma omp task                                                depend(out:a)
    init_data(a);

#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:z[:N])  nowait      depend(out:y)
    compute_3(b, z, N);

#pragma omp target map(to:y[:N]) map(to:z[:N])    nowait      depend(in:x) depend(in:y)
    compute_4(z, x, y, N);

#pragma omp taskwait
```



OpenMP®

Enabling HPC since 1997

Visit www.openmp.org for more information