

Programming OpenMP

Introduction to GPU Offloading

Christian Terboven
Michael Klemm



Q&A

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 😊)

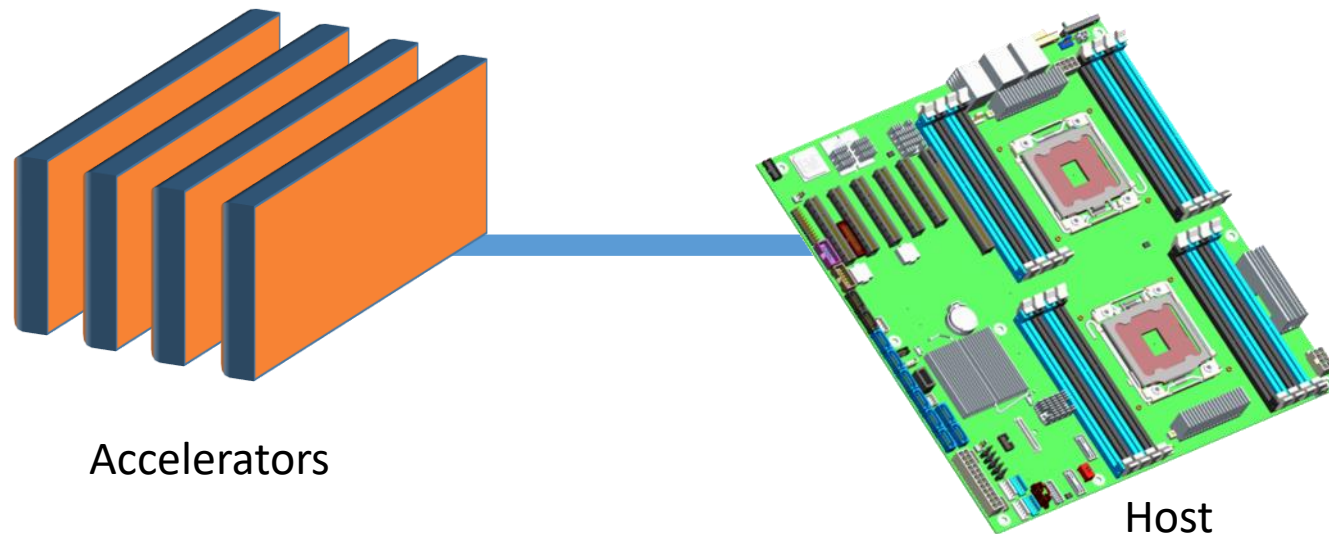
This is the code we want to execute on a target device (i.e., GPU)

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!

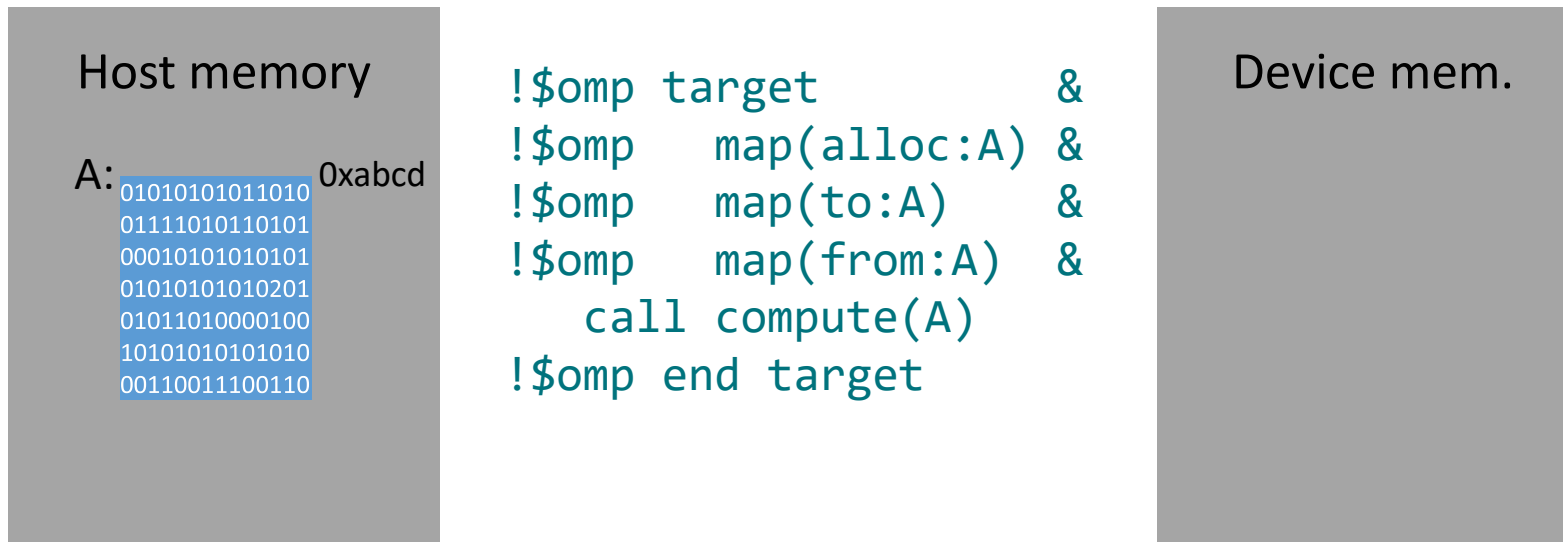
Device Model

- 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.



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

a
x[0:SZ]
y[0:SZ]

target

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

x[0:SZ]
y[0:SZ]

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

clang -fopenmp -fopenmp-targets=amdgc-n-amd-amdhsa -Xopenmp-target=amdgc-n-amd-amdhsa -march=gfx908

Example: saxpy

```
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
```

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

All accessed arrays are copied from host to device and back

a
x(1:n)
y(1:n)

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

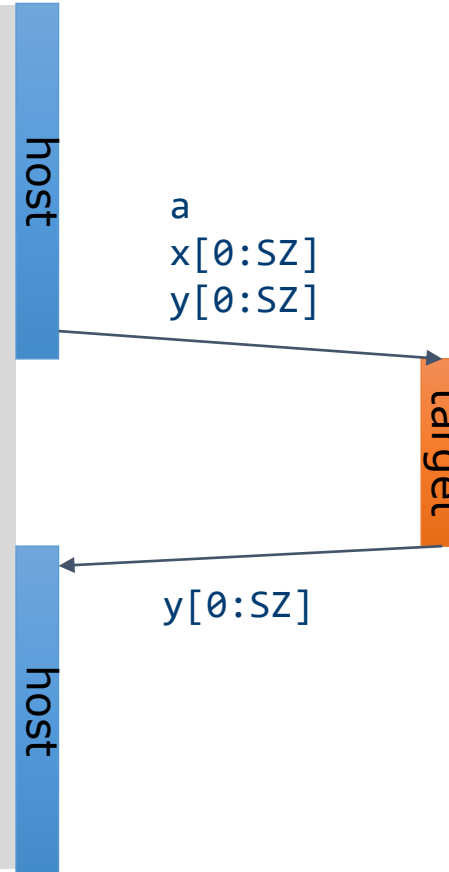
x(1:n)
y(1:n)

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

flang -fopenmp -fopenmp-targets=amdgc-n-amd-amdhsa -Xopenmp-target=amdgc-n-amd-amdhsa -march=gfx908

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 -fopenmp-targets=amdgc-n-amd-amdhsa -Xopenmp-target=amdgc-n-amd-amdhsa -march=gfx908
```

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.

host

a
x[0:sz]
y[0:sz]

target

y[0:sz]

host

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

```
clang -fopenmp -fopenmp-targets=amdgc-n-amd-amdhsa -Xopenmp-target=amdgc-n-amd-amdhsa -march=gfx908
```

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

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

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

```
clang -fopenmp -fopenmp-targets=amdgc-n-amd-amdhsa -Xopenmp-target=amdgc-n-amd-amdhsa -march=gfx908
```

Programming OpenMP

GPU: expressing parallelism

Christian Terboven

Michael Klemm

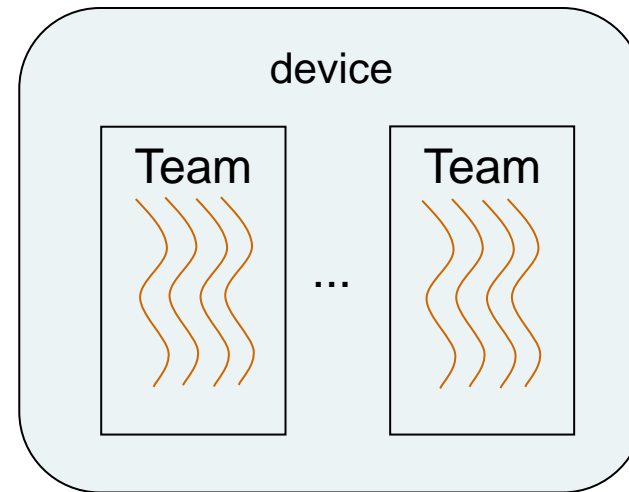


teams and distribute constructs

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

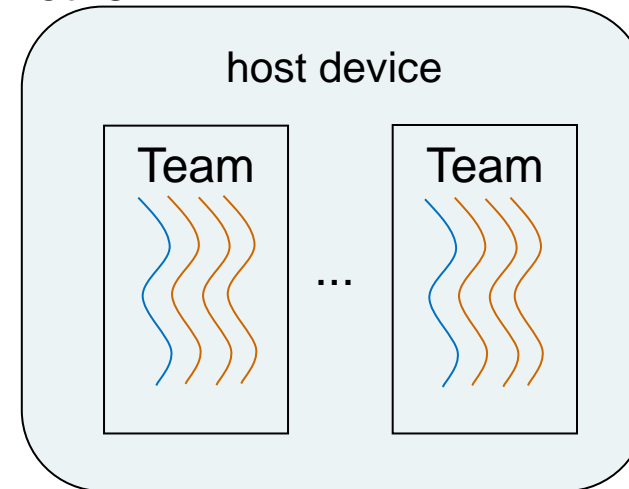
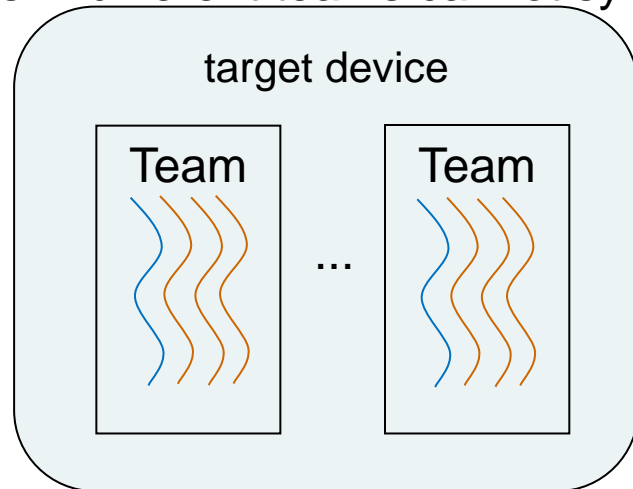
Terminology

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



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



5.0

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

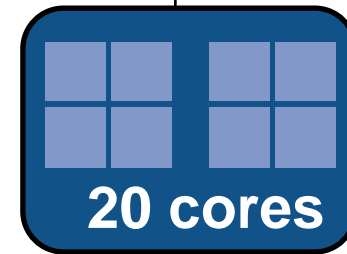
- **distribute** [**simd**], **distribute** [**parallel**] worksharing-loop [**SIMD**]
- **parallel** regions (**parallel for/do, parallel sections**)
- **omp_get_num_teams()** and **omp_get_teams_num()**

distribute Construct

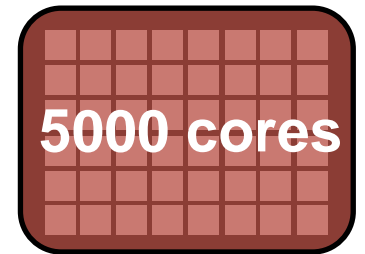
- work sharing among the teams regions
 - Distribute the iterations of the associated loops across the master threads of each team executing the region
- Strictly nested inside a teams region
- No implicit barrier at the end of the construct
- **dist_schedule**(*kind*[, *chunk_size*])
 - The scheduling 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;  
}
```



CPU



GPU

How to port
DAXPY to a GPU?

Kernel Directives

- Offload kernel code
 - `target`: offload work
 - `teams, parallel`: create in parallelly running threads
 - `distribute, do, for, simd`: worksharing across parallel units
- Worksharing
 - `for`: offload work
 - `collapse`: collapse two or more nested loops to increase parallelism

```
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

```
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

Thread




Core


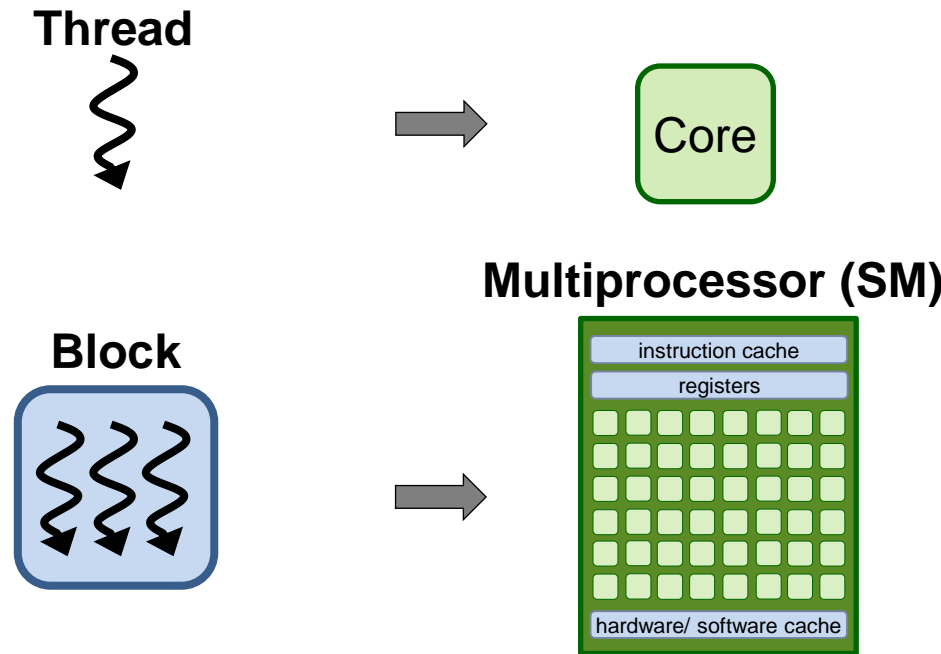
- Each thread is executed by a core

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:

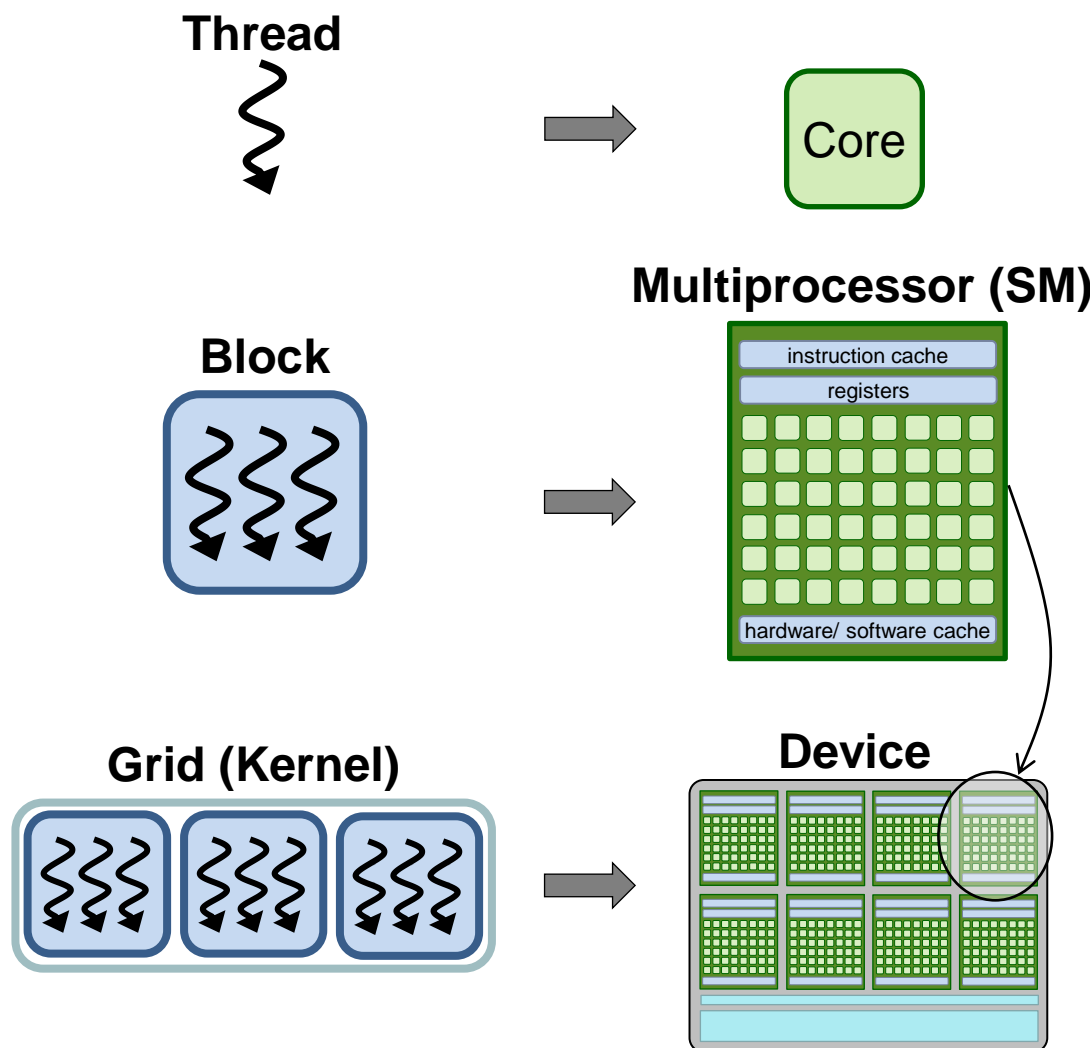
\$ \$CC \$FLAGS_OFFLOAD_OPENMP daxpy.c

\$ 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

loop constructs

Motivation

- Sometimes, it might be reasonable to shift some burden to the compiler + runtime
 - Discussion: prescriptive vs. descriptive OpenMP
 - OpenACC decided to go the other way
- But: OpenMP has to maintain backwards compatibility
- Loop construct: (IMHO) the first step to introduce descriptivity in OpenMP
 - `loop`: specifies that the iterations may be executed concurrently
 - Enables (= permits) the compiler to generate threaded / accelerated code

loop construct

- Syntax (C/C++):
`#pragma omp loop [clause[[,] clause]...]`
for-loops
- Syntax (Fortran):
`!$omp teams [clause[[,] clause]...]`
do-loops
- Clauses
`bind`: **either** `teams`, `parallel` or `thread`: determines parallel execution entity
`collapse(n)` : explained above
`ordered(concurrent)` : (for future extensions: concurrent is currently def.)
`private(list)` : explained above
`firstprivate(list)` : explained above
`reduction([default,]reduction-identifier:list)` : explained above

Programming OpenMP

Hands-on Exercises: Jacobi on GPU

Christian Terboven
Michael Klemm



STREAM...

The first hands-on is to port the infamous STREAM benchmark to GPU.

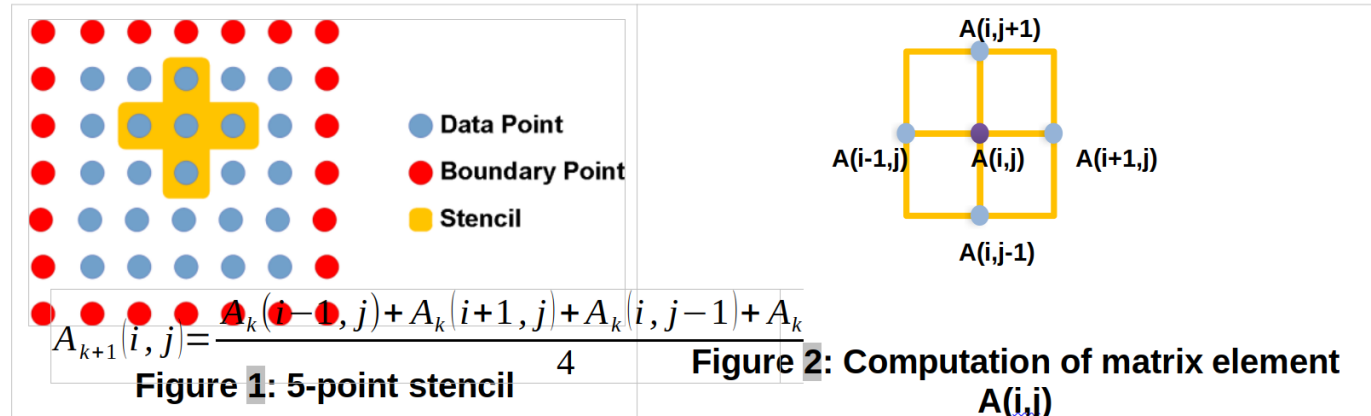
The code already contains function that have “GPU” in their name. Add the proper target directives and data-mapping clauses.

Note: the reported bandwidth will be horrendously low. This is intended and will lead to the next webinar’s topic.

During the following exercises, you will port a Jacobi solver to OpenMP. This **Jacobi** example solves a finite difference discretization (5-point-stencil) of the Laplace equation (2D):

$$\nabla^2 A(x, y) = 0$$

using the Jacobi iterative method. To this end, the Jacobi method starts with an approximation of the objective function $f(x, y)$ and reuses formerly-computed matrix elements to solve the current one. It iterates only about the inner elements of the 2D-grid so that the boundary elements are only used within the stencil. The solving process is aborted if either a certain number of iterations is achieved (see `iter_max`) or the computed approximation is probably close to the solution. In this code, the latter is evaluated by checking whether the biggest change on any matrix element (see array `err` and variable `err`) is smaller than a given tolerance value, in the current iteration.



Jacobi on GPU / 2

- Task 0: You might want to acquire reference measurements on the host (w/ GPU)...
- Task 1: Get it to the GPU: Parallelize only the one most compute-intensive loop
- Task 2: Improve the data management and the amount of parallelism on the GPU
- Task 3: Optimize that scheduling of iterations for the GPU