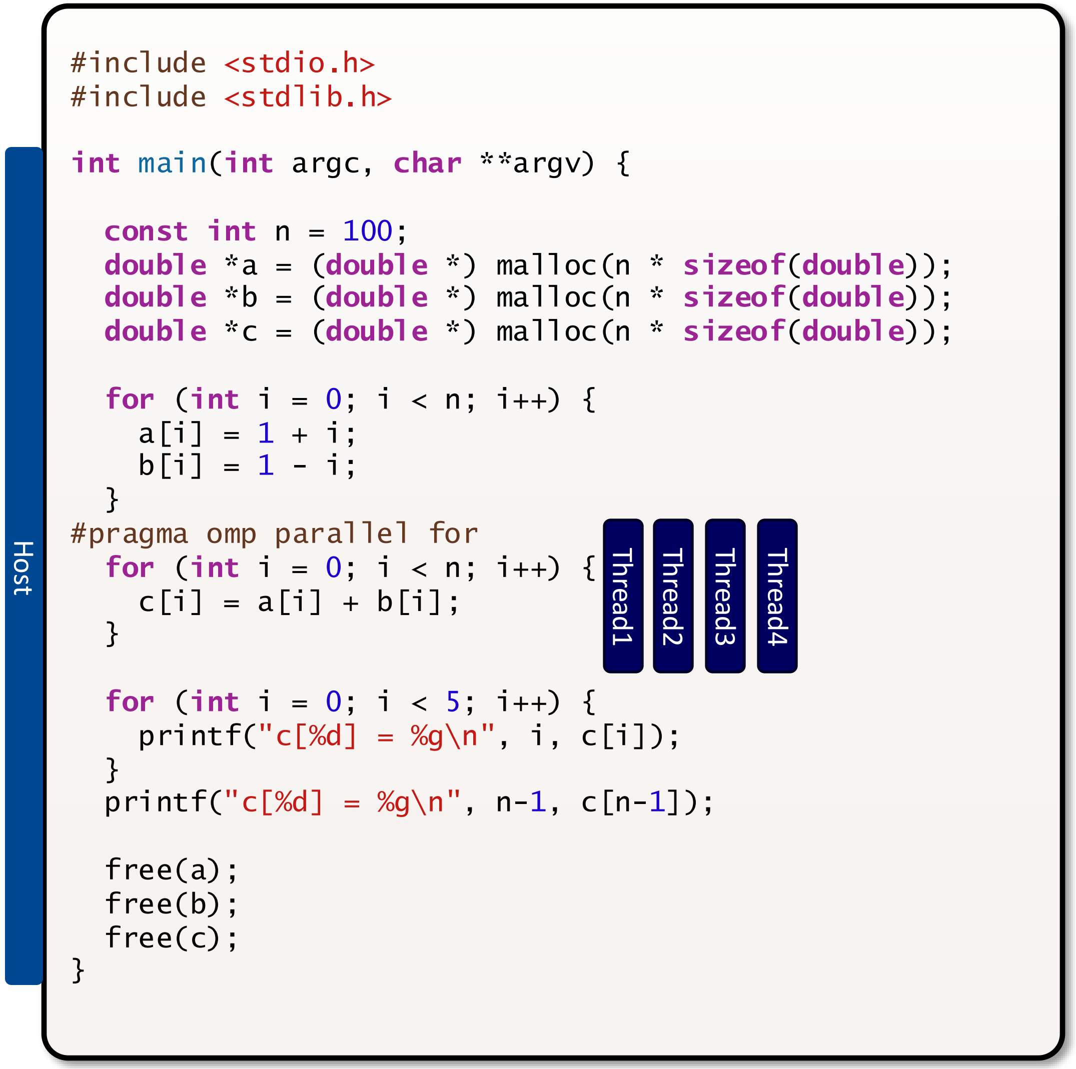
Accelerator Programming with OpenMP

Introduction

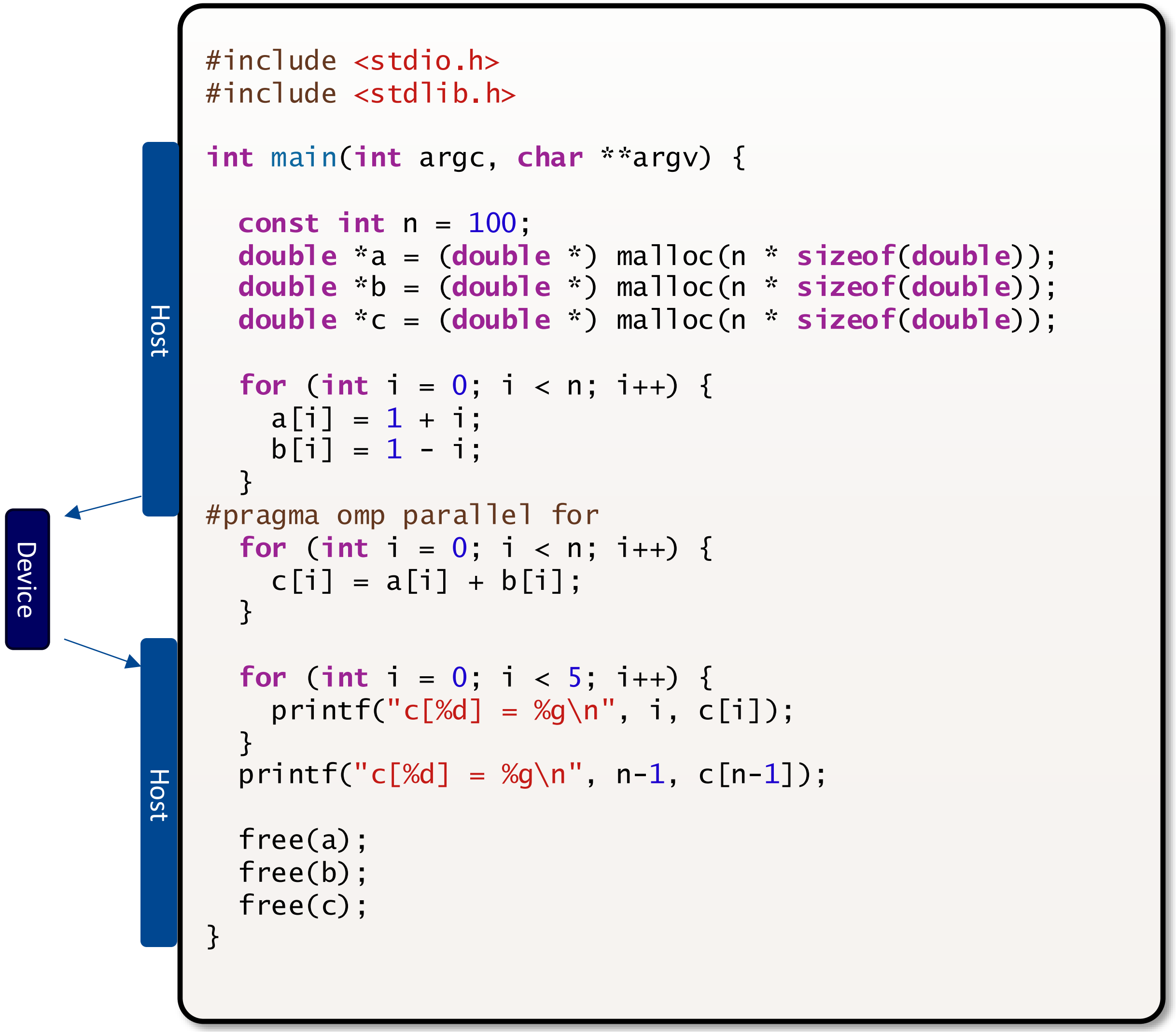
- In a perfect world we would like the OpenMP work-sharing directives to automatically run on an accelerator



```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {

  const int n = 100;
  double *a = (double *) malloc(n * sizeof(double));
  double *b = (double *) malloc(n * sizeof(double));
  double *c = (double *) malloc(n * sizeof(double));

  for (int i = 0; i < n; i++) {
    a[i] = 1 + i;
    b[i] = 1 - i;
  }
#pragma omp parallel for
  for (int i = 0; i < n; i++) {
    c[i] = a[i] + b[i];
  }

  for (int i = 0; i < 5; i++) {
    printf("c[%d] = %g\n", i, c[i]);
  }
  printf("c[%d] = %g\n", n-1, c[n-1]);

  free(a);
  free(b);
  free(c);
}
```

Introduction

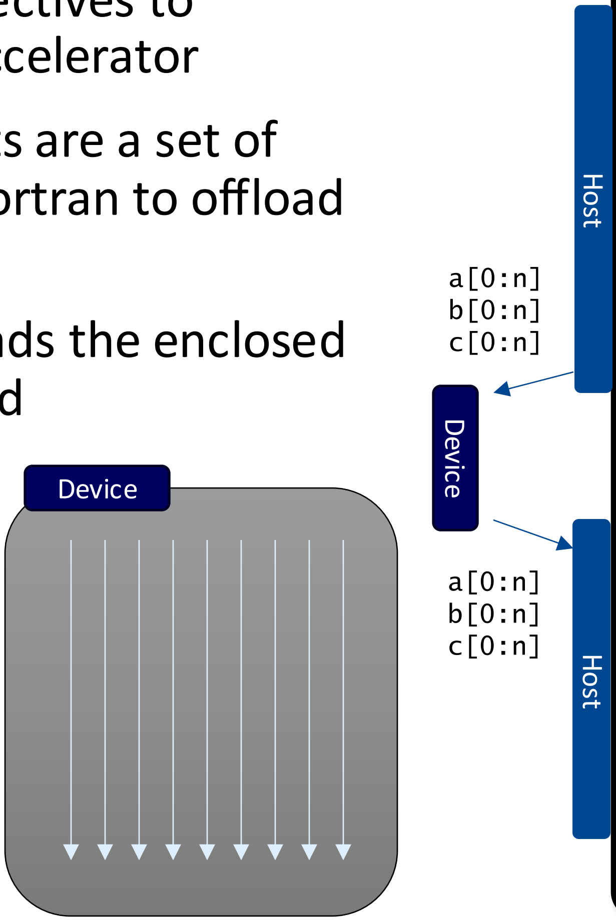
- In a perfect world we would like the OpenMP work-sharing directives to automatically run on an accelerator



```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {

  const int n = 100;
  double *a = (double *) malloc(n * sizeof(double));
  double *b = (double *) malloc(n * sizeof(double));
  double *c = (double *) malloc(n * sizeof(double));

  for (int i = 0; i < n; i++) {
    a[i] = 1 + i;
    b[i] = 1 - i;
  }
#pragma omp parallel for
  for (int i = 0; i < n; i++) {
    c[i] = a[i] + b[i];
  }

  for (int i = 0; i < 5; i++) {
    printf("c[%d] = %g\n", i, c[i]);
  }
  printf("c[%d] = %g\n", n-1, c[n-1]);

  free(a);
  free(b);
  free(c);
}
```

Introduction

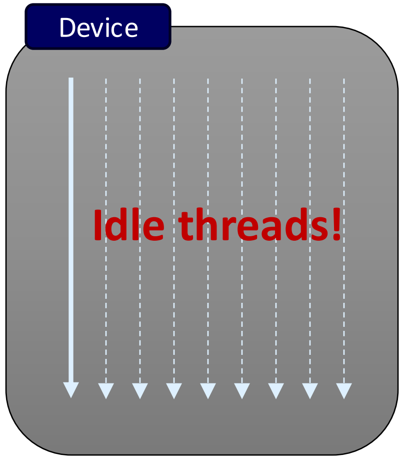
- In a perfect world we would like the OpenMP work-sharing directives to automatically run on an accelerator
- OpenMP offload constructs are a set of directives for C/C++ and Fortran to offload work to an accelerator
- The **target** construct offloads the enclosed code to the accelerator and **map** transfers data
- But how is it executed?

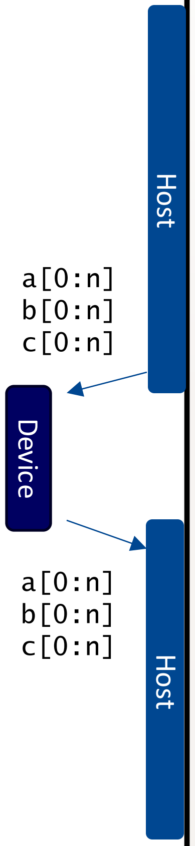


```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {

  const int n = 100;
  double *a = (double *) malloc(n * sizeof(double));
  double *b = (double *) malloc(n * sizeof(double));
  double *c = (double *) malloc(n * sizeof(double));

  for (int i = 0; i < n; i++) {
    a[i] = 1 + i;
    b[i] = 1 - i;
  }
#pragma omp target map(a[0:n],b[0:n],c[0:n])
  for (int i = 0; i < n; i++) {
    c[i] = a[i] + b[i];
  }

  for (int i = 0; i < 5; i++) {
    printf("c[%d] = %g\n", i, c[i]);
  }
  printf("c[%d] = %g\n", n-1, c[n-1]);

  free(a);
  free(b);
  free(c);
}
```

Introduction

- In a perfect world we would like the OpenMP work-sharing directives to automatically run on an accelerator
- OpenMP offload constructs are a set of directives for C/C++ and Fortran to offload work to an accelerator
- The **target** construct offloads the enclosed code to the accelerator and **map** transfers data
- But how is it executed?
- Without additional directives expressing parallelism the kernel will run in **serial!**



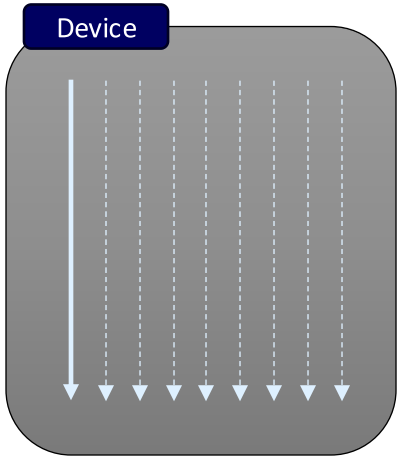


```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {

  const int n = 100;
  double *a = (double *) malloc(n * sizeof(double));
  double *b = (double *) malloc(n * sizeof(double));
  double *c = (double *) malloc(n * sizeof(double));

  for (int i = 0; i < n; i++) {
    a[i] = 1 + i;
    b[i] = 1 - i;
  }
#pragma omp target map(a[0:n],b[0:n],c[0:n])
  for (int i = 0; i < n; i++) {
    c[i] = a[i] + b[i];
  }

  for (int i = 0; i < 5; i++) {
    printf("c[%d] = %g\n", i, c[i]);
  }
  printf("c[%d] = %g\n", n-1, c[n-1]);

  free(a);
  free(b);
  free(c);
}
```

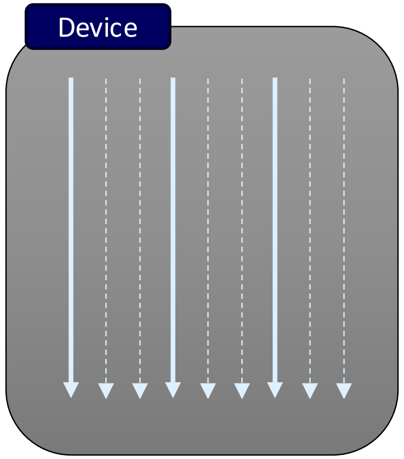
OpenMP offload target parallelism

Several different combinations of device execution directives (we will only cover some of them)

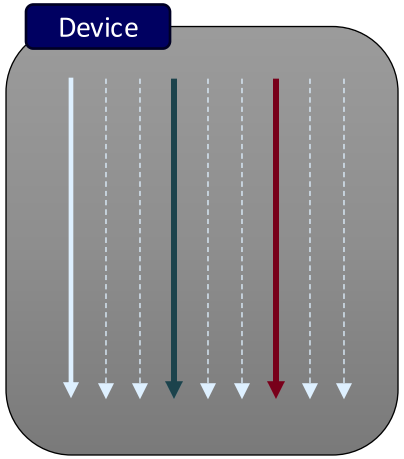
`omp target`



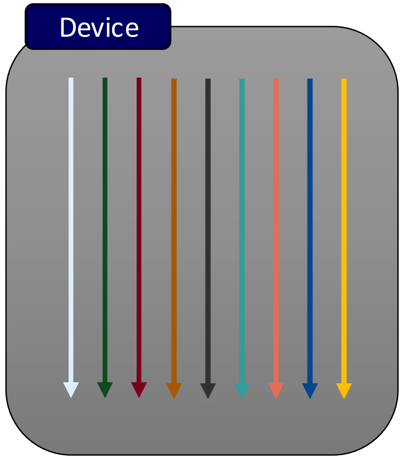
`omp target teams`



`omp target teams distribute`



`omp target teams distribute parallel for/do`



OpenMP offload target parallelism

- The **target** construct offloads the code to the accelerator
- The **teams** construct creates a set of teams (but still no work-sharing within the teams)
- With the **distribute** construct work can be distributed between teams (e.g. **target teams distribute**), but still no work-sharing within the teams
- Using a **parallel for/do** construct, work can be distributed within each team (e.g. **target teams distribute parallel for/do**)
- Or use the modern (OpenMP 5) construct **omp target teams loop** to use the full GPU (similar to the **target teams distribute parallel for/do** construct)

OpenMP device memory

Controlling and minimising data movement to/from the device is the most important step when optimising an OpenMP offloading program

- The **map** clause can be used to control data movement
- Various form of the clause:
  - `map(to:list)` copy data to the device when entering the region
  - `map(from:list)` copy data back to the host when leaving the region
  - `map(tofrom:list)` combination of the above to and from clause
  - `map(alloc:list)` allocate data when entering the region

Note: In Fortran nothing special needs to be done to move dynamically allocated memory, while in C/C++ the dimension has to be specified e.g. `map(to:a[0:n])`

Hint: Vendors often provide tools to show data movement, on a HPE Cray set the environment variable CRAY_ACC_DEBUG=`1, 2 or 3`

OpenMP device memory

The **target data** construct creates a data region, to keep the data on the device between different target offload regions (use **map** to control data movement)

```
!$omp target data map(a,b) <- move data to device

  !$omp target
   do a lot of work on the device with a,b

  do something on the host

  !$omp target
    do a lot of work on the device with a,b

!$omp end target data <- move data back to host
```

Update data to/from a device using the **target update to(list)/from(list)** clause

References

OpenMP specifications:

http://www.openmp.org/specifications/

OpenMP API syntax reference guide

https://www.openmp.org/wp-content/uploads/OpenMPRefGuide-5.2-Web-2024.pdf