Introduction to GPU porting with OpenACC

Dominik Zobel, Claudia Frauen

DKRZ

Goal of this session

Learn basics of GPU offloading with OpenACC directives

- Differences between GPU to CPU
- Offloading possibilites
- Knowledge about basic OpenACC directives (`parallel`, `kernels` and `data` construct)

 |

References

The following sources were used as cited on the respective pages:

Reference	Description and links
OpenACC	OpenACC slides from 2019-2021 as referenced here: https://github.com/OpenACC/openacc-training-materials
Jacob	Marek Jacob’s presentation *“Introduction to GPU computing FORTRAN and OpenACC (Part 1)”* from 2023-06 at DKRZ (not publicly available)
PRACE	PRACE Training Course: Directive-based GPU programming with OpenACC from 2021-11 in Jülich: https://juser.fz-juelich.de/record/902543/

General GPU introduction

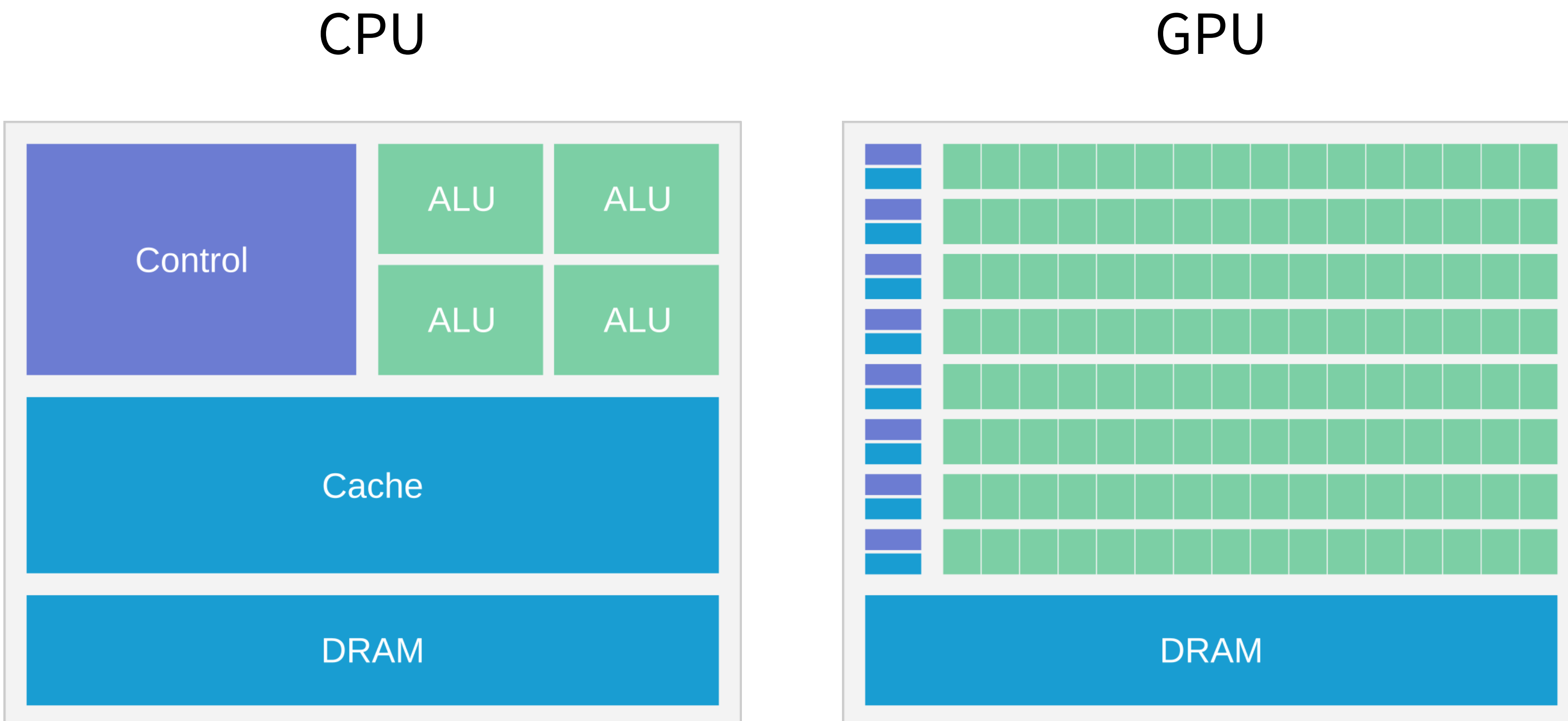
Accelerator Computing

- GPU = Graphical Processing Unit
- GPU computing: use of a GPU to offload (intensive) parts of an application, while the remainder is computed on the CPU
- GPUs have thousands of compute cores: need to express fine-grain parallelism
- GPU and CPU have (currently) separate physical memory
  - requires specific data management
  - data transfer may be a performance issue (slow transfer via PCI bus)

Source: Jacob M1_4

DKRZ

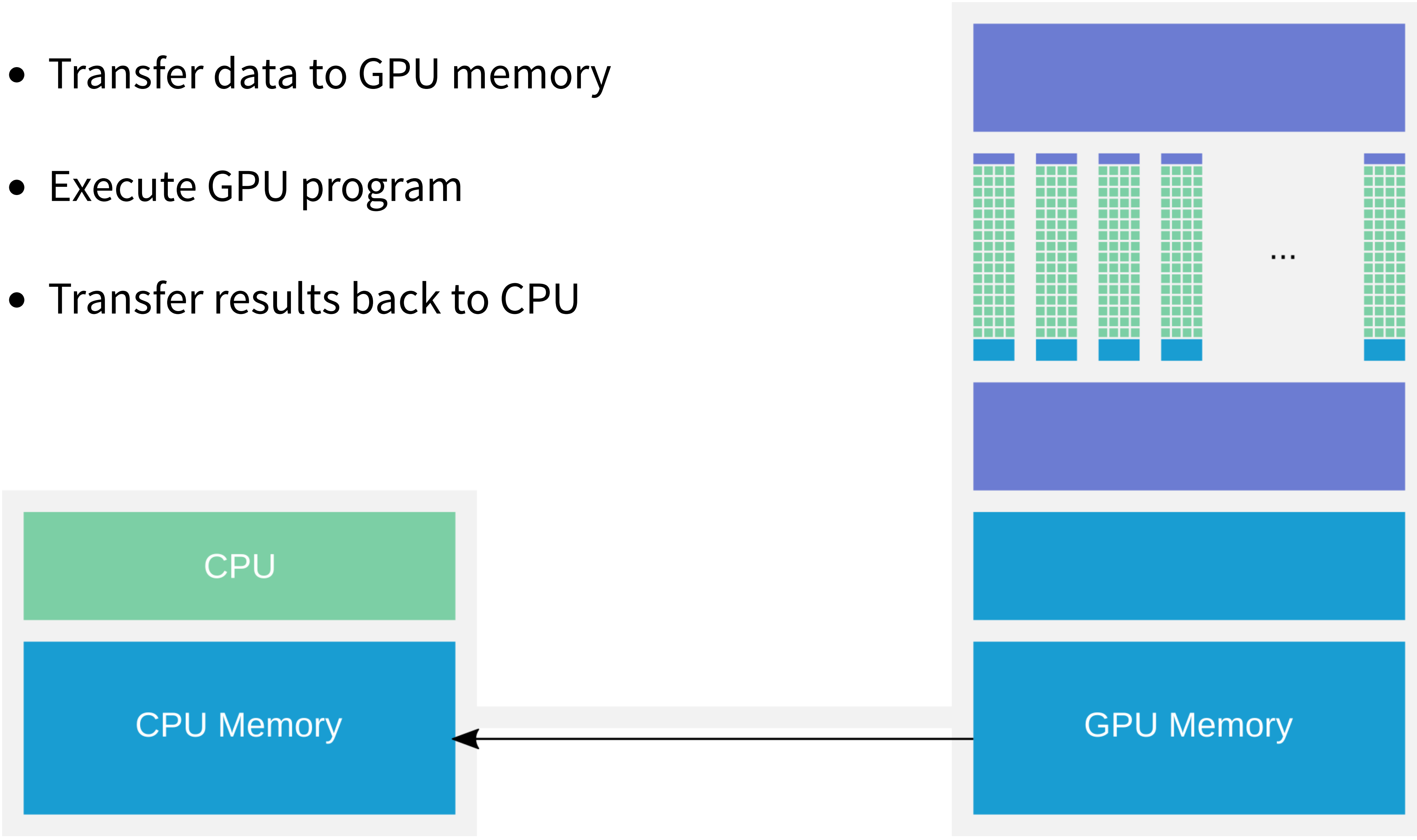
Differences CPU and GPU computing



Source: PRACE P1_10

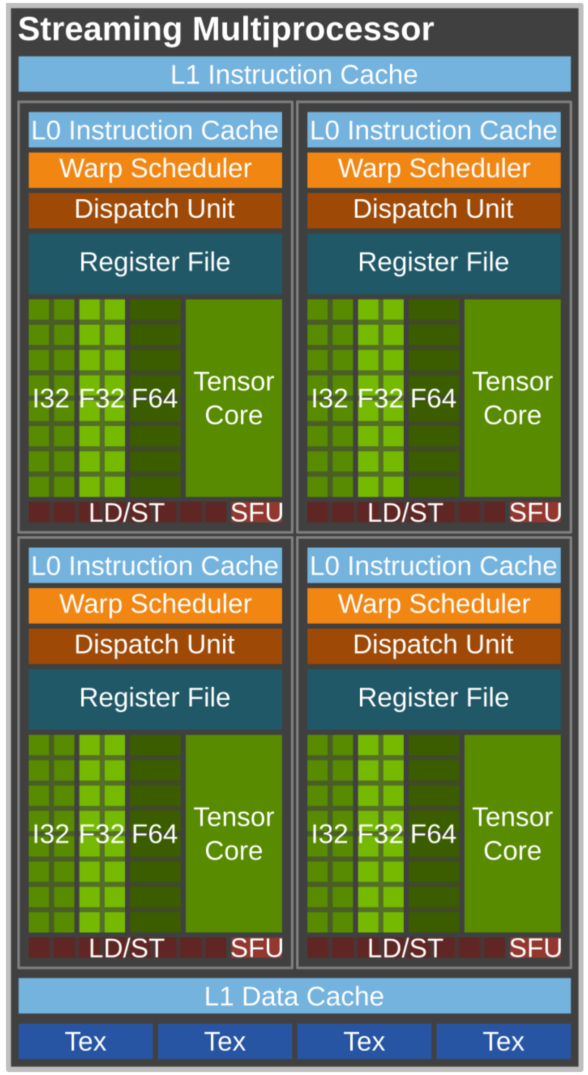
Processing Flow

- Transfer data to GPU memory
- Execute GPU program
- Transfer results back to CPU



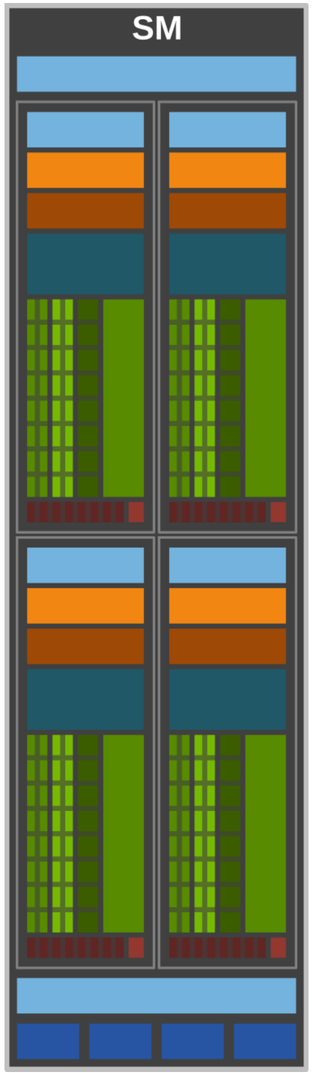
Source: PRACE P1_13

Processing on GPU hardware



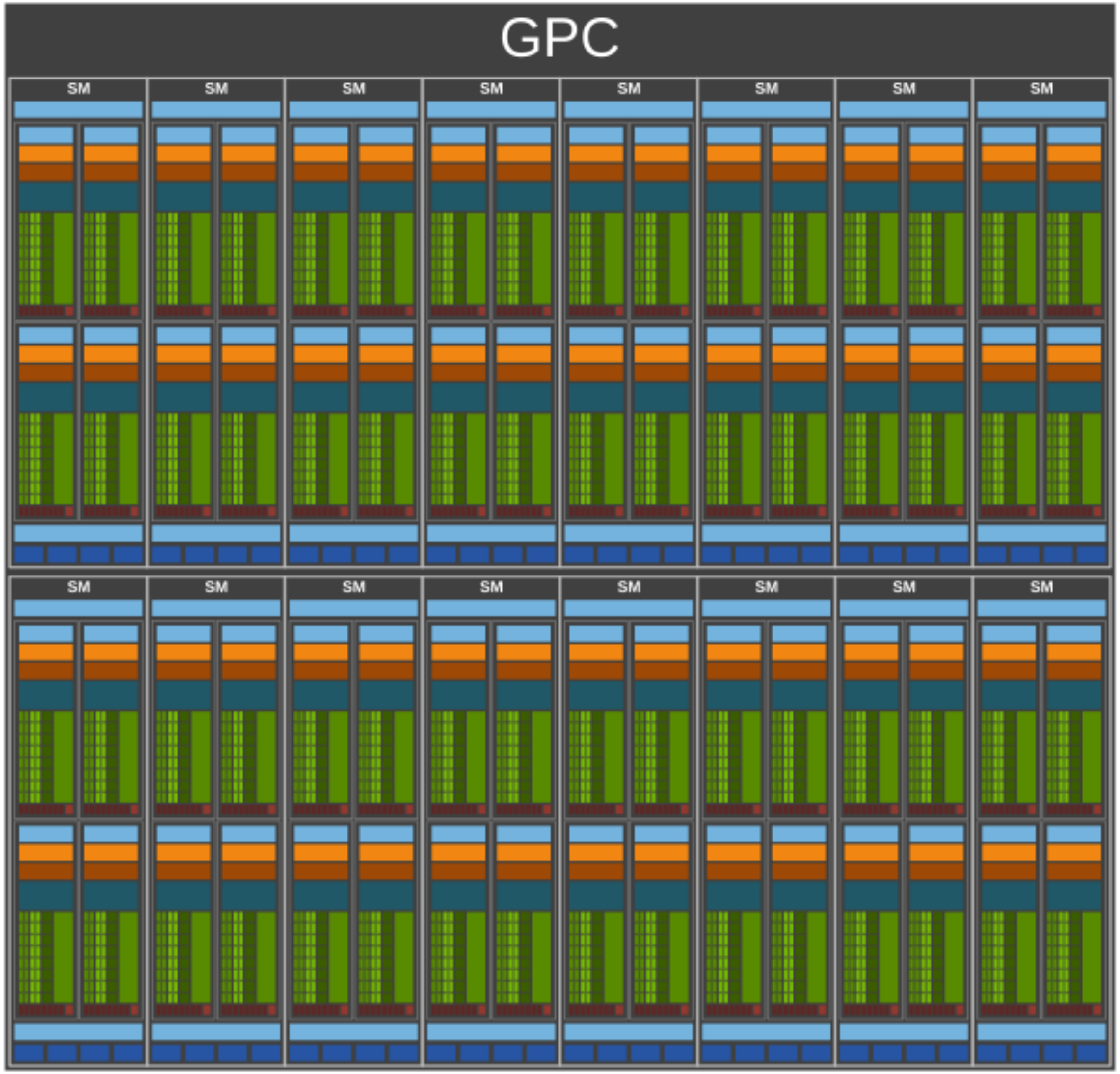
Source: PRACE P1_17

Processing on GPU hardware



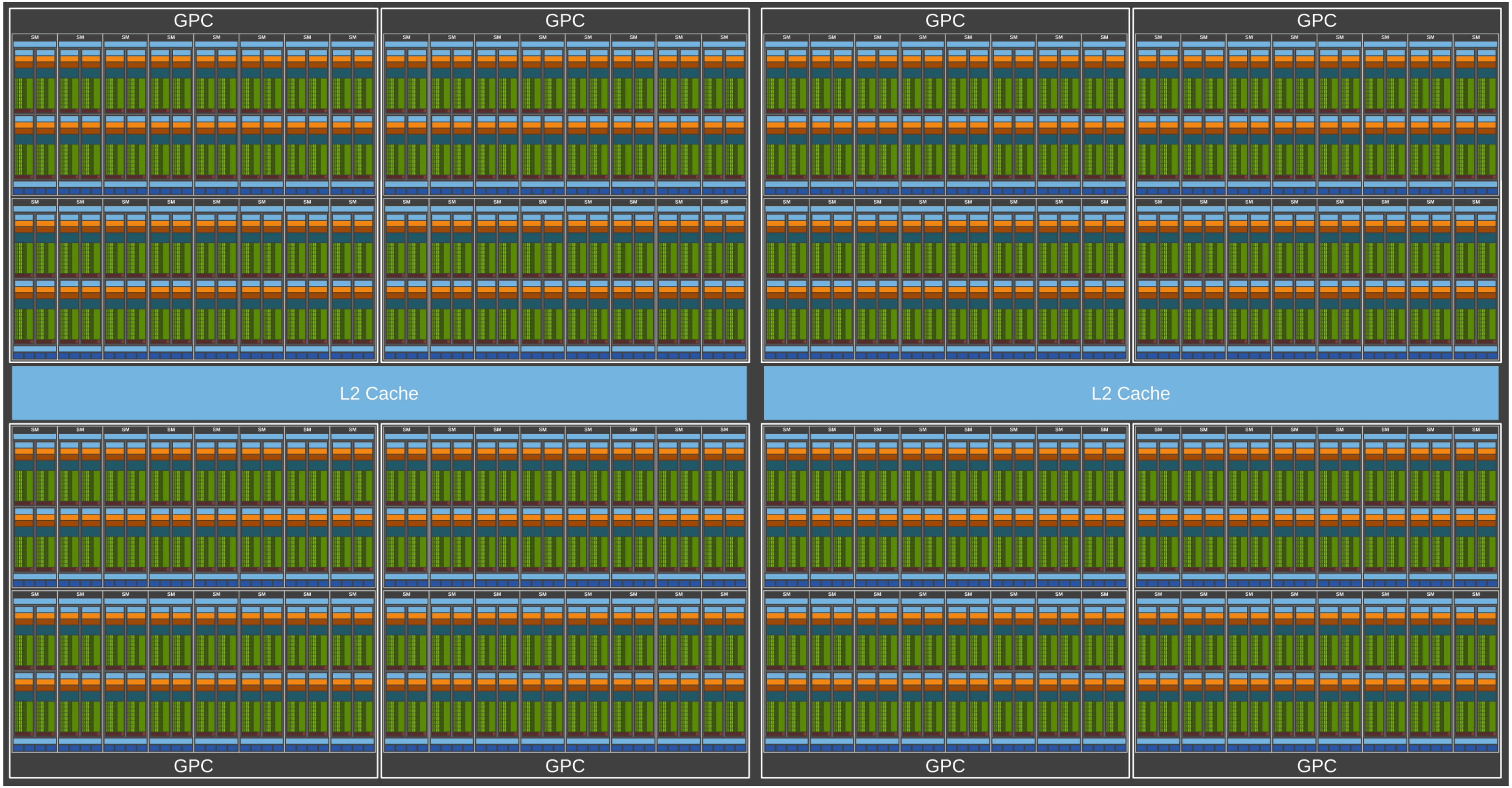
Source: PRACE P1_17

Processing on GPU hardware



Source: PRACE P1_17

Processing on GPU hardware



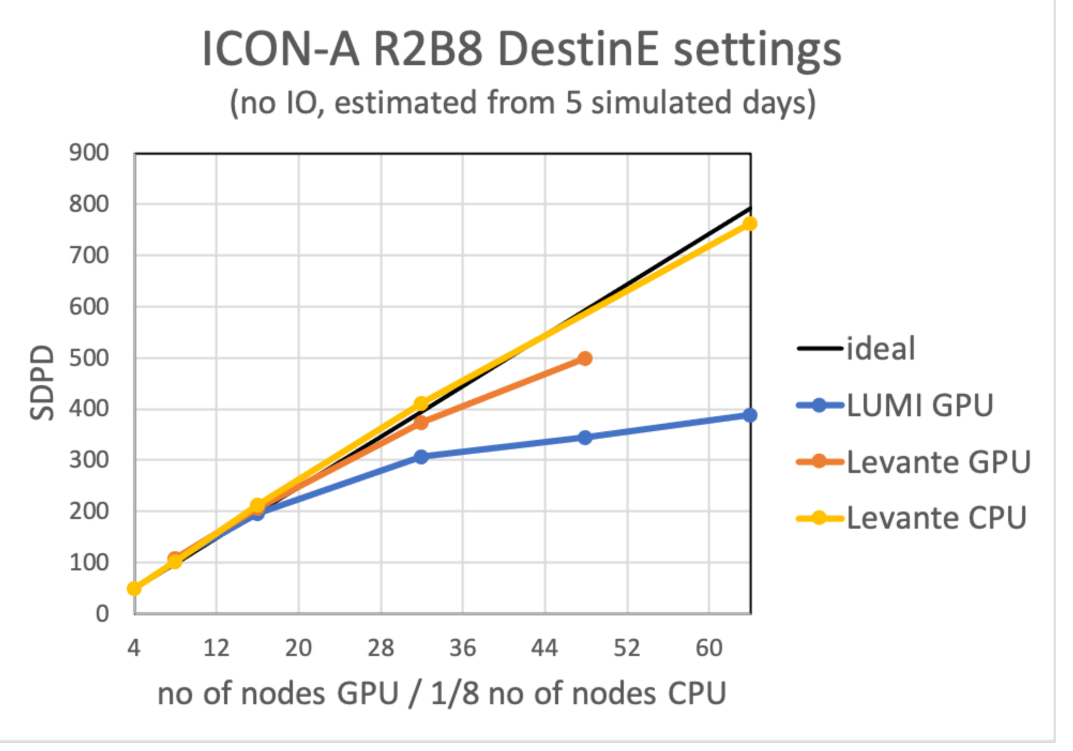
Source: PRACE P1_17

DKRZ

GPUs need massively parallel applications

ICON-A 10km setup: 5242880 horizontal grid points

- On 8 Levante GPU nodes (4 GPUs per node) 2 blocks with nproma=83300
- On 16 Levante GPU nodes 1 block with nproma=83884
- On 48 Levante GPU nodes 1 block with nproma=28454
- LUMI has 8 GPUs per node and thus the blocks are only half the size of Levante



DKRZ

CPU vs. GPU

CPU

Optimized for **low latency**

+ Large main memory
+ Fast clock rate
+ Large caches
+ Branch prediction
+ Powerful ALU

- Relatively low memory bandwidth
- Cache misses costly
- Low performance per watt

GPU

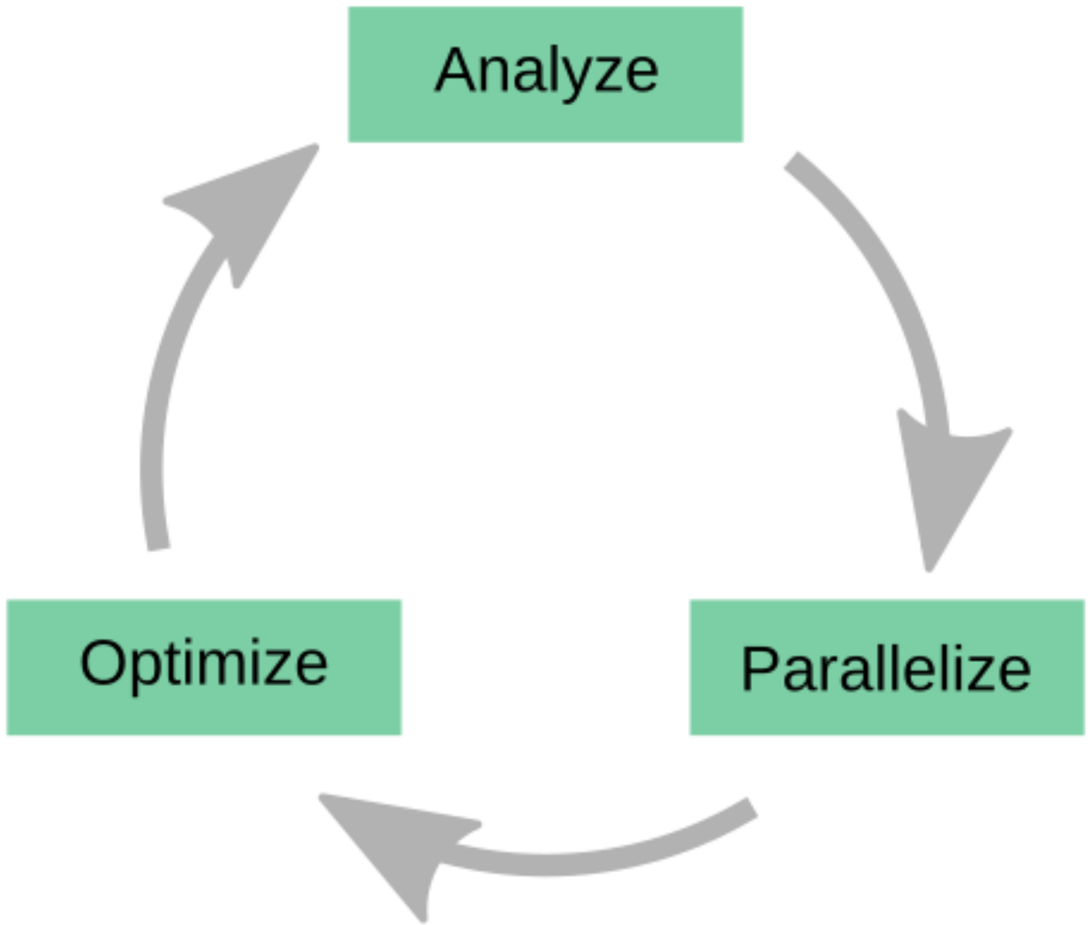
Optimized for **high throughput**

+ High bandwidth main memory
+ Latency tolerant (parallelism)
+ More compute resources
+ High performance per watt

- Limited memory capacity
- Low per-thread performance
- Extension card

Source: PRACE P1_19

Porting workflow cycle

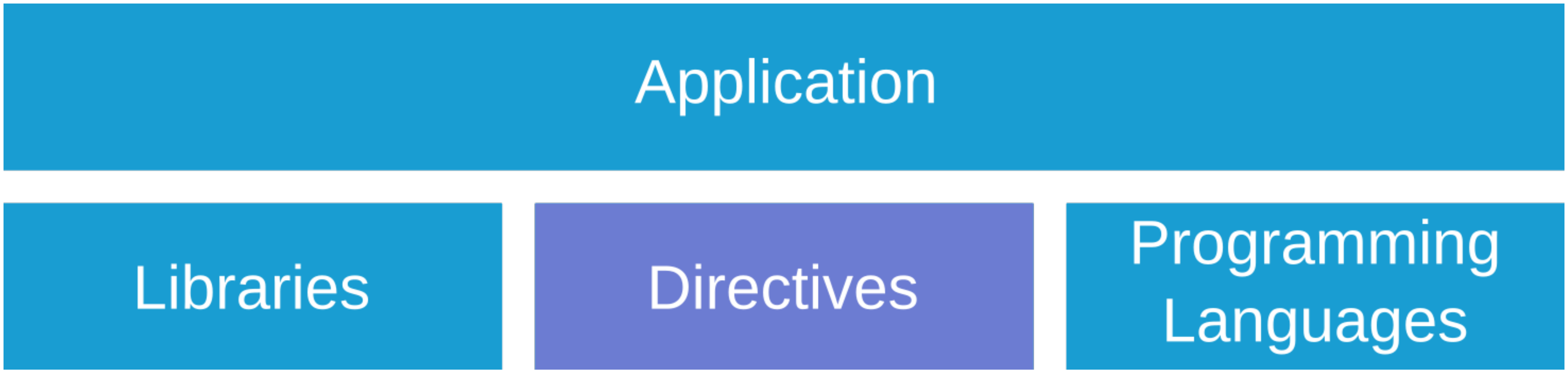


This workshop mostly focuses on parallelization

Source: OpenACC O2_12

 |

Offloading possibilities



Drop-In Acceleration

Easy Acceleration

Flexible Acceleration

Source: PRACE P1_21

DKRZ

OpenACC introduction

Concept of directives in source code

- Compiler directives state intent to compiler

```c
#pragma acc kernels
for (int i = 0; i < 1023; i++) {
    // ...
}
```

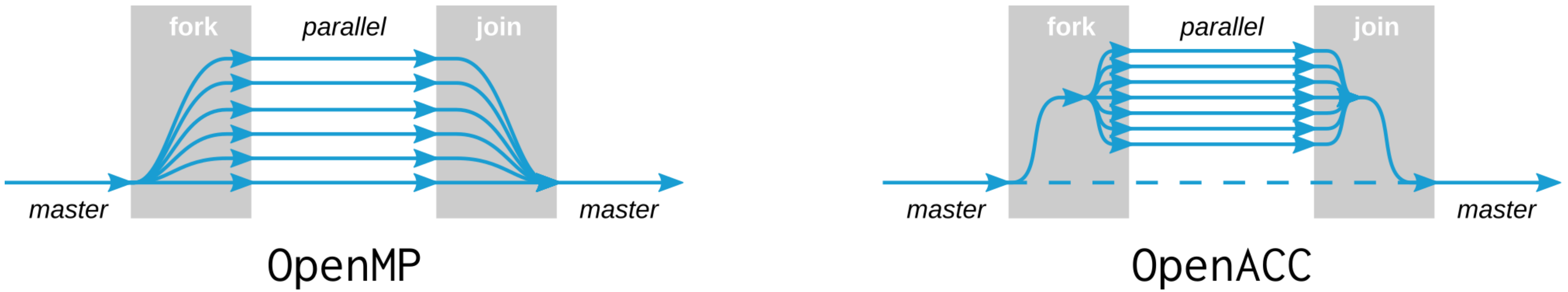
```fortran
!$acc kernels
do i = 1, 1024
    ! ...
end do
!$acc end kernels
```

- Ignored by compiler which does not understand OpenACC
- OpenACC: Compiler directives, library routines, environment variables
- Portable across host systems and accelerator architectures (NVIDIA)

Source: PRACE P2_7

OpenMP and OpenACC

- OpenACC modeled after OpenMP, but specific for accelerators
- OpenMP: Offloading; compiler support improving (Clang, GCC, NVHPC, …)
- Same basic principle: Fork/join model



Source: PRACE P2_4

DKRZ

OpenACC overview

OpenACC is a specification for high-level compiler directives to express parallelism for accelerators in Fortran, C and C++

- Originally aimed to be performance portable to a wide range of accelerator devices
- Multiple Vendors, Multiple Devices, One Specification

The OpenACC specification was first released in November 2011

- Compilers: Nvidia, Cray, GCC
- ICON follows OpenACC 2.6, released in November 2017
- Hint: 3.3 standard is easier to read, only details have changed

Official web site: https://www.openacc.org

Source: Jacob M1_5

 |

Offloading with the parallel construct

`!$acc parallel` Start parallel execution of the following section until `!$acc end parallel`

`!$acc loop` to specify the type of parallelism for the immediately following loop. Possible clauses are:

- `vector`, `worker`, or `gang`: set the specific type of accelerator parallelism to execute the iterations of the loop.
- `seq`: execute the loop sequentially
- [no clause]: Choice of parallelism type left to the compiler

Recommendation: Add `!$acc Loop` to each loop within a parallel region.

Source: Jacob M1_8

DKRZ

Parallel construct

```c
#pragma acc parallel loop
for (int i=0; i<N; i++) {
    x[i] = 1.0;
    y[i] = 2.0;
}

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

```fortran
!$acc parallel loop
do i = 1, N
    x(i) = 1.0
    y(i) = 2.0
end do
!$acc end parallel loop

!$acc parallel loop
do i = 1, N
    y(i) = i*x(i)+y(i);
end do
!$acc end parallel loop
```

Arrays x and y are automatically copied to the GPU memory before kernel execution and back to the CPU memory afterwards.

(Scalars are automatically copied to the GPU)

Source: PRACE P2_12, Jacob M1_9

Levels of parallelism

`vector` Threads that work in SIMT (SIMD) fashion

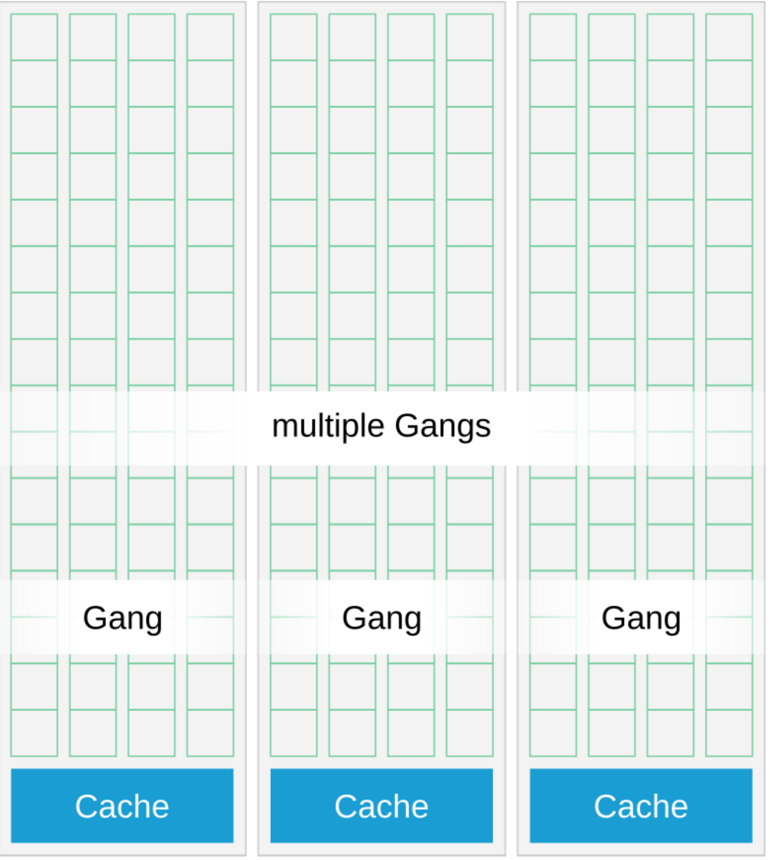
Operate on same cache

`worker` Group of threads that can operate vector instructions

`gang` Independent parallelism

Threads in a gang can operate on same memory

`seq` No parallelism



The optimal choice of parallelism can be hardware dependent.

Source: Jacob M1_31

Compiler flags for offloading

OpenACC compiler support: activate with compile flag

NVHPC `nvfortran -acc`

Compiler flag	Effect
`-acc=gpu|multicore`	Target GPU or CPU
`-acc=gpu -gpu=cc80`	Generate Ampere-compatible code
`-gpu=cc80,lineinfo`	Add source code correlation into binary
`-gpu=managed`	Use unified memory
`-Minfo=accel`	Print acceleration info

Source: PRACE P2_9

 |

Loop dependencies

Not all loops are parallel

```fortran
program fwd_dep
   implicit none
   integer, parameter :: nel = 10
   integer, dimension(nel) :: vec
   integer :: i

   vec = [(i, i = 1, nel)]
   do i = 2, nel
      vec(i) = vec(i) + vec(i-1)
   end do
   write(*, '(9(i2,", "),i2)') vec
end program fwd_dep
```

Sequential result:

```
 1,  3,  6, 10, 15, 21, 28, 36, 45, 55
```

⇒ Forward loop dependency prevents parallelization

Source: OpenACC 01_36

Reductions

- The reduction clause is used to calculate a single value based on multiple ones (e.g. summation)
- Each thread will have their own private copy of the reduction variable (partial reduction)
- Afterwards, the global result will be calculated with a final reduction

```fortran
do k = 1, size
   do j = 1, size
      do i = 1, size
         c(i, j) = c(i, j) + a(i, k) * b
      end do
   end do
end do
```

```fortran
!$acc parallel loop collapse(2)
do j = 1, size
   do i = 1, size
      tmp = c(i, j)
      !$acc loop reduction(+: tmp)
      do k = 1, size
         tmp = tmp + a(i, k) * b(k, j)
      end do
      c(i, j) = tmp
   end do
end do
!$acc end parallel loop
```

Source: OpenACC 03_29

 |

Kernel construct

```
!$acc kernels
a(:) = 0.0
!$acc end kernels
```

```
!$acc kernels
do i = 1, n
   a(i) = 2.0 * b(i)
end do
!$acc end kernels
```

- The `kernels` directive instructs the compiler to search for parallel loops in the code
- The compiler will analyze the loops and parallelize those it finds safe and profitable to do so ⇒ more freedom for compiler
- Rest: Same as for parallel

Source: PRACE P2_35; OpenACC O3_36; Jacob M1_29

DKRZ

Kernel vs. parallel

Both approaches valid, may perform equally well

`!$acc kernels`

- Compiler performs parallel analysis
- Can cover large area of code with single directive
- Gives compiler additional leeway (possibly not optimal performance)

`!$acc parallel`

- Requires parallel analysis by programmer
- Will also parallelize what compiler may miss (or ignore loop dependencies)
- More explicit
- Similar to OpenMP

Source: PRACE P2_37

Example (1/6)

```fortran
program p_example
  implicit none
  integer, parameter :: dp = selected_real_
  integer, parameter :: nel = 1000
  real(dp), dimension(nel, nel) :: a
  integer :: i, j



  do i = 1, nel

    do j = 1, nel
      a(j, i) = 0.5_dp*(i*nel+j)
    end do
  end do


  write(*, *) a(10, 10)
end program p_example
```

How to port the do-loops?

Example (2/6)

```
program p_example
  implicit none
  integer, parameter :: dp = selected_real_
  integer, parameter :: nel = 1000
  real(dp), dimension(nel, nel) :: a
  integer :: i, j


  !$acc parallel
  do i = 1, nel

    do j = 1, nel
      a(j, i) = 0.5_dp*(i*nel+j)
    end do
  end do
  !$acc end parallel

  write(*, *) a(10, 10)
end program p_example
```

```
NVCOMPILER_ACC_TIME=1 ./p_example
```

LOG

⇒ `parallel` alone does not parallelize

DKRZ

Example (3/6)

```
program p_example
  implicit none
  integer, parameter :: dp = selected_real_
  integer, parameter :: nel = 1000
  real(dp), dimension(nel, nel) :: a
  integer :: i, j


  !$acc parallel loop
  do i = 1, nel

    do j = 1, nel
      a(j, i) = 0.5_dp*(i*nel+j)
    end do
  end do
  !$acc end parallel loop

  write(*, *) a(10, 10)
end program p_example
```

```
NVCOMPILER_ACC_TIME=1 ./p_example
```

LOG

⇒ Works, but we can be more specific about distribution

Example (4/6)

```fortran
program p_example
  implicit none
  integer, parameter :: dp = selected_real_
  integer, parameter :: nel = 1000
  real(dp), dimension(nel, nel) :: a
  integer :: i, j

  !$acc parallel
  !$acc loop gang
  do i = 1, nel
    !$acc loop vector
    do j = 1, nel
      a(j, i) = 0.5_dp*(i*nel+j)
    end do
  end do
  !$acc end parallel

  write(*, *) a(10, 10)
end program p_example
```

```
NVCOMPILER_ACC_TIME=1 ./p_example
```

LOG

$\Rightarrow$ All relevant do-loops annotated

Example (5/6)

```fortran
program p_example
  implicit none
  integer, parameter :: dp = selected_real_
  integer, parameter :: nel = 1000
  real(dp), dimension(nel, nel) :: a
  integer :: i, j


  !$acc parallel loop gang vector
  do i = 1, nel

    do j = 1, nel
      a(j, i) = 0.5_dp*(i*nel+j)
    end do
  end do
  !$acc end parallel loop

  write(*, *) a(10, 10)
end program p_example
```

```
NVCOMPILER_ACC_TIME=1 ./p_example
```

LOG

⇒ Works, but we can also specify type for each loop

Example (6/6)

```fortran
program p_example
  implicit none
  integer, parameter :: dp = selected_real_
  integer, parameter :: nel = 1000
  real(dp), dimension(nel, nel) :: a
  integer :: i, j


  !$acc parallel loop gang vector collapse(
  do i = 1, nel

    do j = 1, nel
      a(j, i) = 0.5_dp*(i*nel+j)
    end do
  end do
  !$acc end parallel loop

  write(*, *) a(10, 10)
end program p_example
```

```
NVCOMPILER_ACC_TIME=1 ./p_example
```

LOG

⇒ Collapse if possible

DKRZ

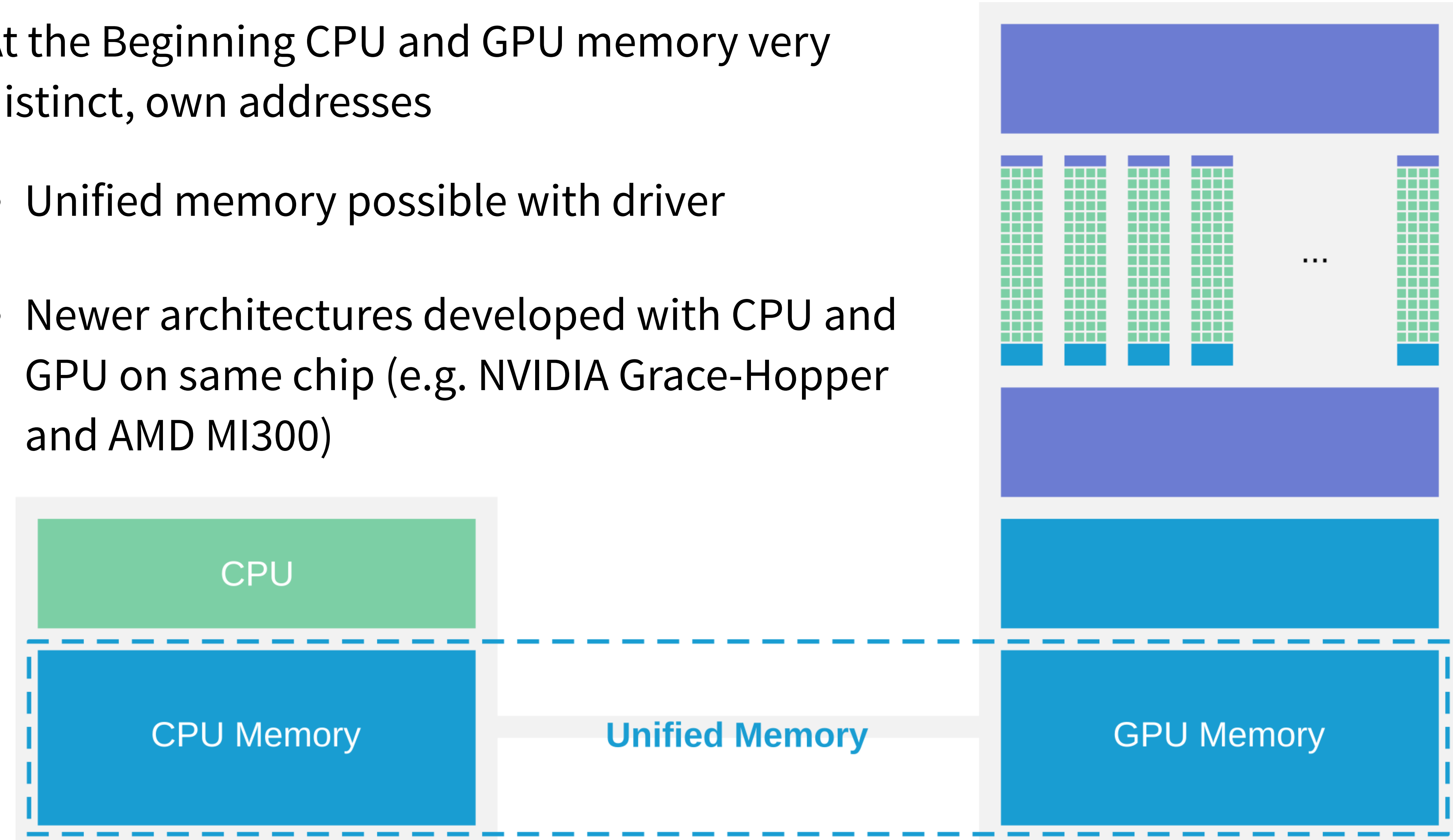
OpenACC data movement

DKRZ

CPU and GPU memory

At the Beginning CPU and GPU memory very distinct, own addresses

- Unified memory possible with driver
- Newer architectures developed with CPU and GPU on same chip (e.g. NVIDIA Grace-Hopper and AMD MI300)

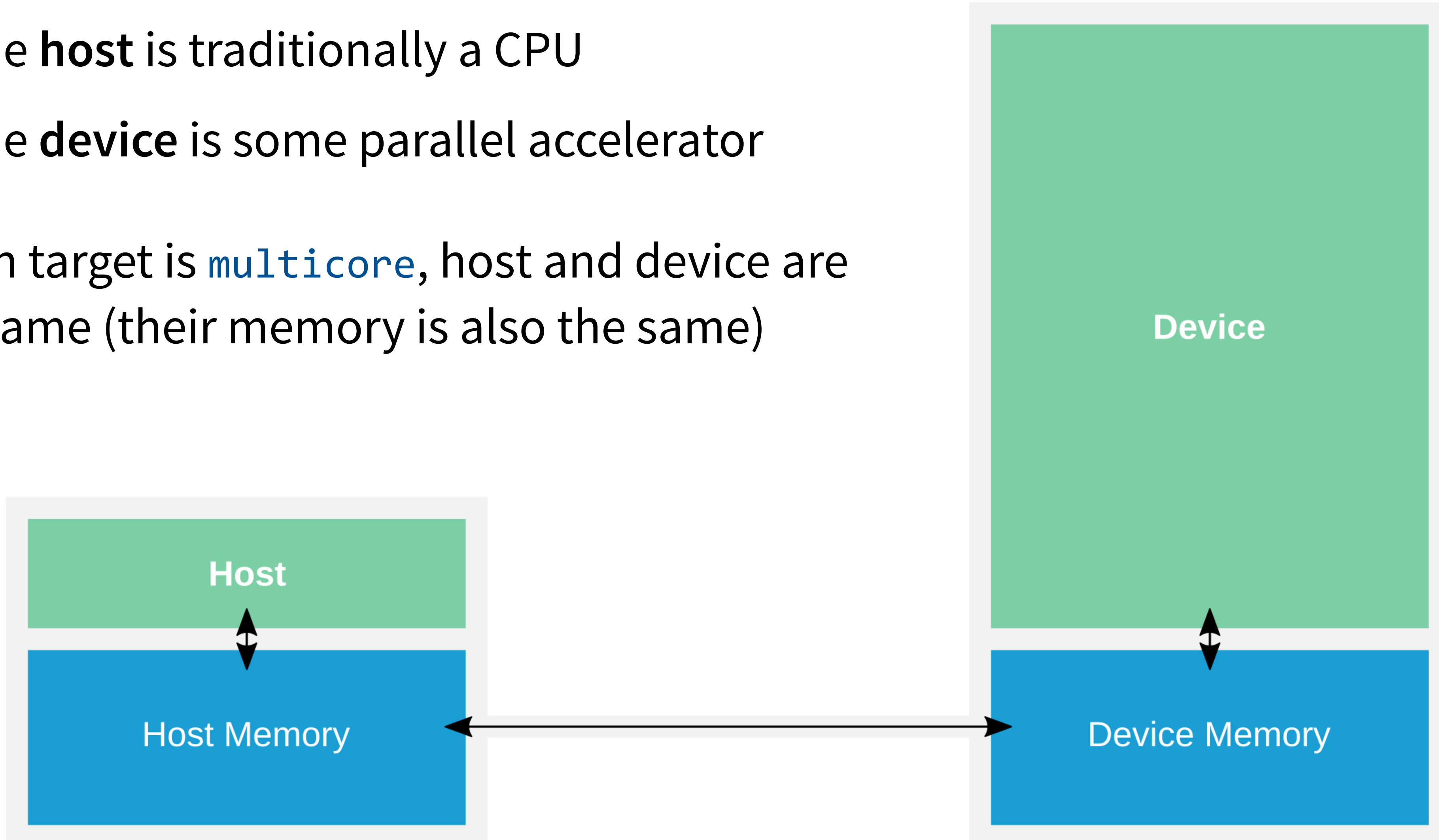


Source: PRACE P2_44

Data management overview

- The **host** is traditionally a CPU
- The **device** is some parallel accelerator

When target is `multicore`, host and device are the same (their memory is also the same)



Source: OpenACC O4_9

Data statements

```
!$acc data [clause [, clause]... ]

! ...

!$acc end data
```

- Defines region of code in which data remains on device
- Data is shared among all kernels in region
- Explicit data transfers
- Transferring between these two memories can be a very time consuming process
- Clauses to augment the data regions

Source: PRACE P2_55, OpenACC O5_5

Data clauses

`copy( <list> )`

- Allocates memory on GPU
- Copies data from host to GPU when entering region
- Copies data to the host when exiting region

Variants:

- `copyin( <list> )`: No copying to host at the end
- `copyout( <list> )`: No copying to GPU at the beginning
- `create( <list> )`: Only memory allocation, no copying

Note: Scalars are first private by default, they rarely appear in a data clause.

Source: OpenACC O4_22; Jacob M1_16

Update clauses

Note: The clauses from the previous slide have no effect if the data is already on the device! Within data regions host and device memory can be updated with the `update` *clause.*

`update device( <list> )`

- Copy data from host to device

`update self( <list> )` or `update host( <list> )`

- Copy data from device to host

Source: Jacob M1_17

Data movement (1/4)

```
! ...
iter = 0

do iter = 1, iter_max
  error = 0.0_dp


  do j = 2, m-1
    do i = 2, n-1
      Anew(i, j) = 0.25_dp * (A(i+1, j) &
        + A(i-1, j) + A(i, j-1) + A(i, j+1)
      error = max(error, abs(Anew(i, j)-A(i
    end do
  end do


  A = Anew

  if (error <= tol) exit
end do

! ...
```

Source: PRACE P2_60

Example code with

- outer convergence loop
- Stencil operation within nested loop
- Array assignment

Data movement (2/4)

```fortran
! ...
iter = 0

do iter = 1, iter_max
  error = 0.0_dp
  !$acc parallel loop &
  !$acc reduction(max:error)
  do j = 2, m-1
    do i = 2, n-1
      Anew(i, j) = 0.25_dp * (A(i+1, j) &
        + A(i-1, j) + A(i, j-1) + A(i, j+1)
      error = max(error, abs(Anew(i, j)-A(i
    end do
  end do
  !$acc end parallel loop
  !$acc kernels
  A = Anew
  !$acc end kernels
  if (error <= tol) exit
end do

! ...
```

Source: PRACE P2_60

Porting

- stencil with `parallel` construct
- Array assignment with `kernels` construct

Data movement (3/4)

```
! ...
iter = 0

do iter = 1, iter_max
  error = 0.0_dp
  !$acc parallel loop copyin(A) copy(Anew)
  !$acc reduction(max:error)
  do j = 2, m-1
    do i = 2, n-1
      Anew(i, j) = 0.25_dp * (A(i+1, j) &
        + A(i-1, j) + A(i, j-1) + A(i, j+1)
      error = max(error, abs(Anew(i, j)-A(i
    end do
  end do
  !$acc end parallel loop
  !$acc kernels copyin(Anew) copy(A)
  A = Anew
  !$acc end kernels
  if (error <= tol) exit
end do

! ...
```

Source: PRACE P2_60

Data movement

- for each kernel with `copyin` and `copy` clauses
- states intent but does not improve performance

Data movement (4/4)

```fortran
! ...
iter = 0
!$acc data copy(A, Anew)
do iter = 1, iter_max
  error = 0.0_dp
  !$acc parallel loop default(present) &
  !$acc reduction(max:error)
  do j = 2, m-1
    do i = 2, n-1
      Anew(i, j) = 0.25_dp * (A(i+1, j) &
        + A(i-1, j) + A(i, j-1) + A(i, j+1)
      error = max(error, abs(Anew(i, j)-A(i
    end do
  end do
  !$acc end parallel loop
  !$acc kernels default(present)
  A = Anew
  !$acc end kernels
  if (error <= tol) exit
end do
!$acc end data
! ...
```

Source: PRACE P2_60

Data movement

- combined movement of all non-scalar variables
- explicitly outside the loop

Unstructured data clauses

`enter data` and `exit data` directives can be used to manage unstructured data regions, e.g. `!$acc enter data create(a)`

Additional data clause `delete`, e.g. `!$acc exit data delete(var)` to deallocate `var` on GPU

Source: Jacob M1_25

DKRZ

Alternative OpenMP

 |

OpenACC - OpenMP

Most OpenACC directives can be directly translated to OpenMP 5.x `target` directives:

OpenACC	OpenMP
`!$acc parallel`	`!$omp target teams distribute`
`!$acc parallel loop`	`!$omp target teams distribute parallel for`
`!$acc data`	`!$omp target data`
`!$acc data copyin`	`!$omp target data map(to:)`
`!$acc update host`	`!$omp target update from()`

 |

Further training

DKRZ

GPU hackathons

- A great way to learn GPU porting techniques and at the same time get started with porting your application are GPU hackathons
- For example there are annual hackathons hosted by Helmholtz or CSCS in Lugano
  - Helmholtz GPU Hackathon 2024
  - CSCS EuroHack 2024

 |

Training courses

- JSC online training course "Directive-based GPU programming with OpenACC" 29.10.-01.11.2024
- natESM Technical Training "Performance analysis and GPU programming (CUDA, OpenACC, OpenMP, kokkos)" 05.-06.11.2024 at JSC

DKRZ