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/

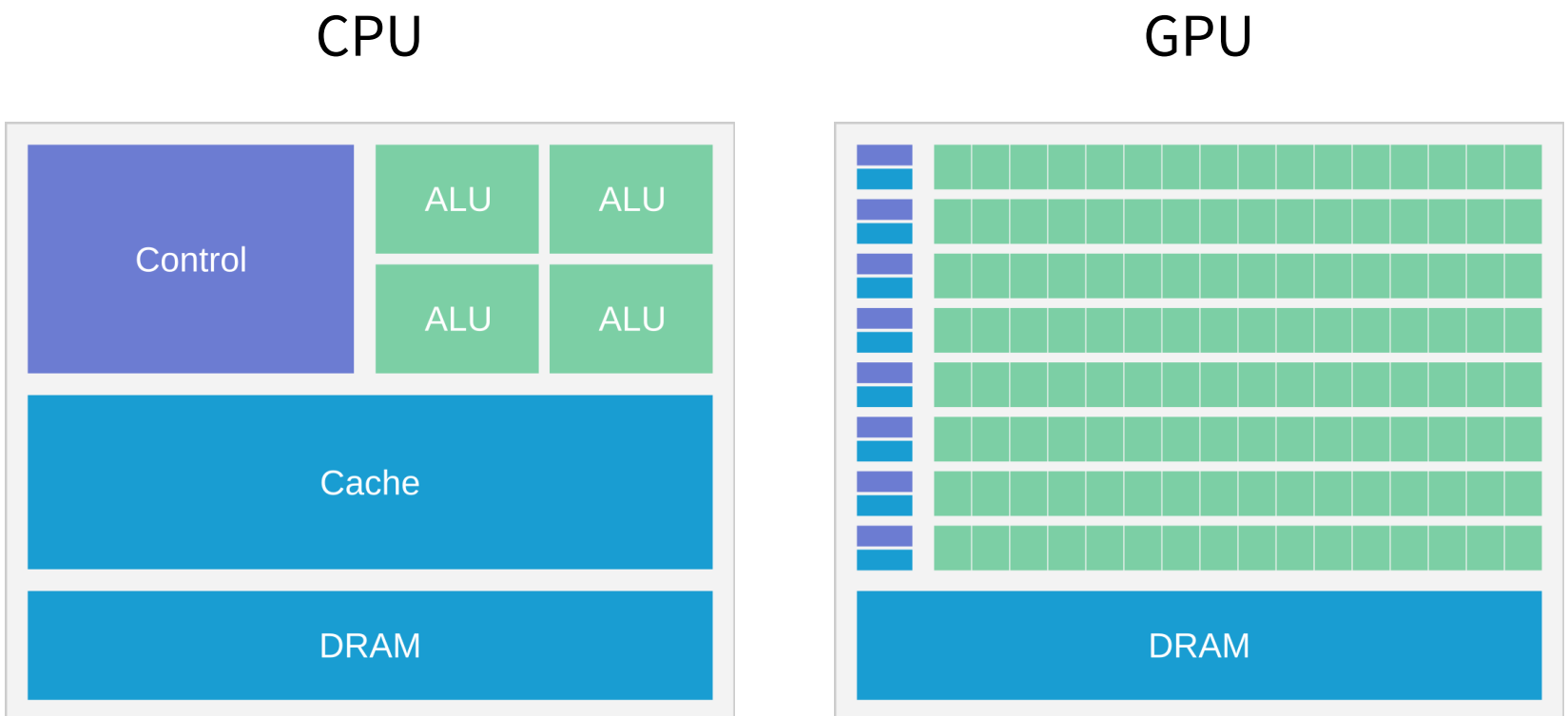
General GPU introduction

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

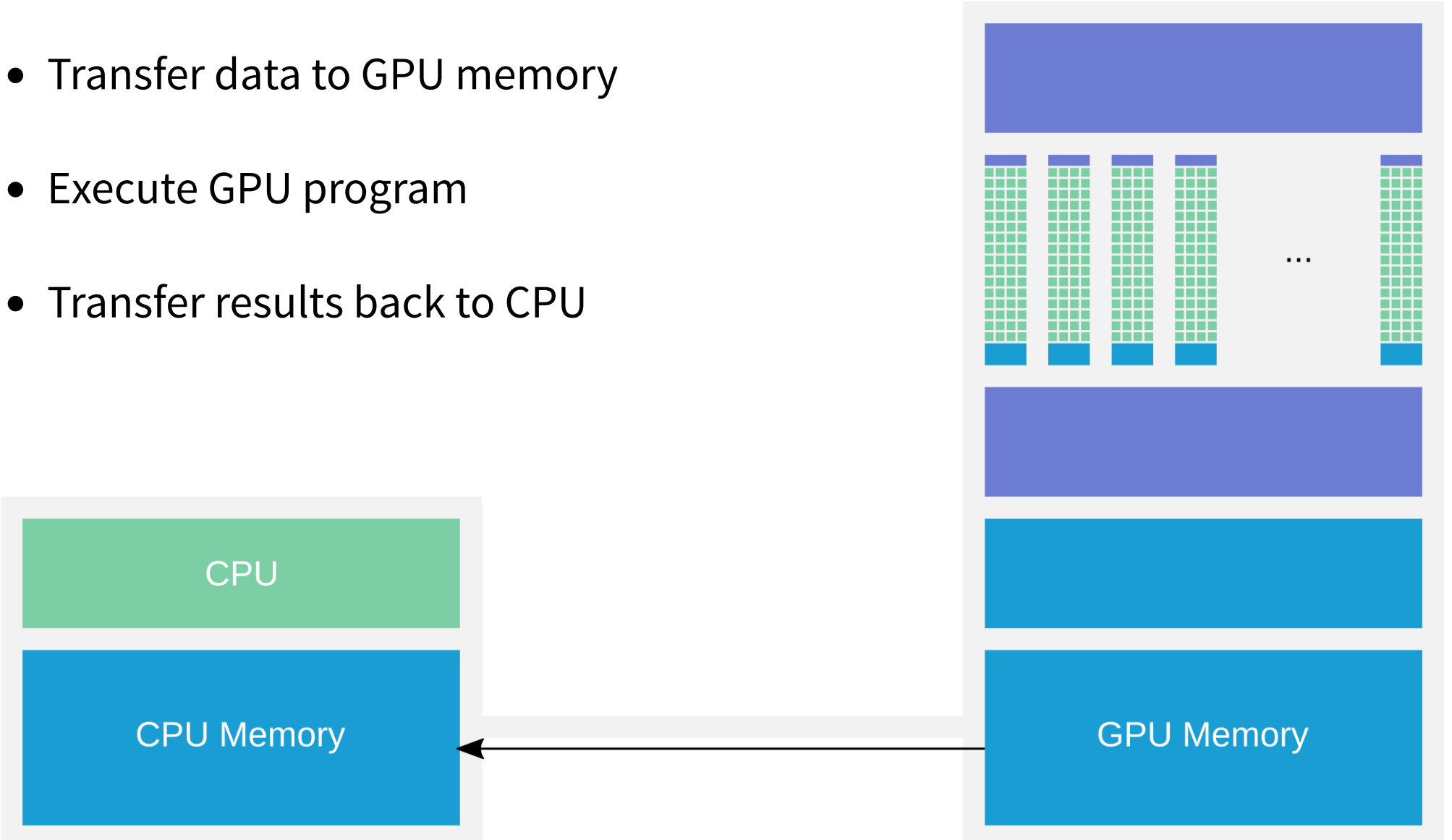
Differences CPU and GPU computing



Source: PRACE P1_10

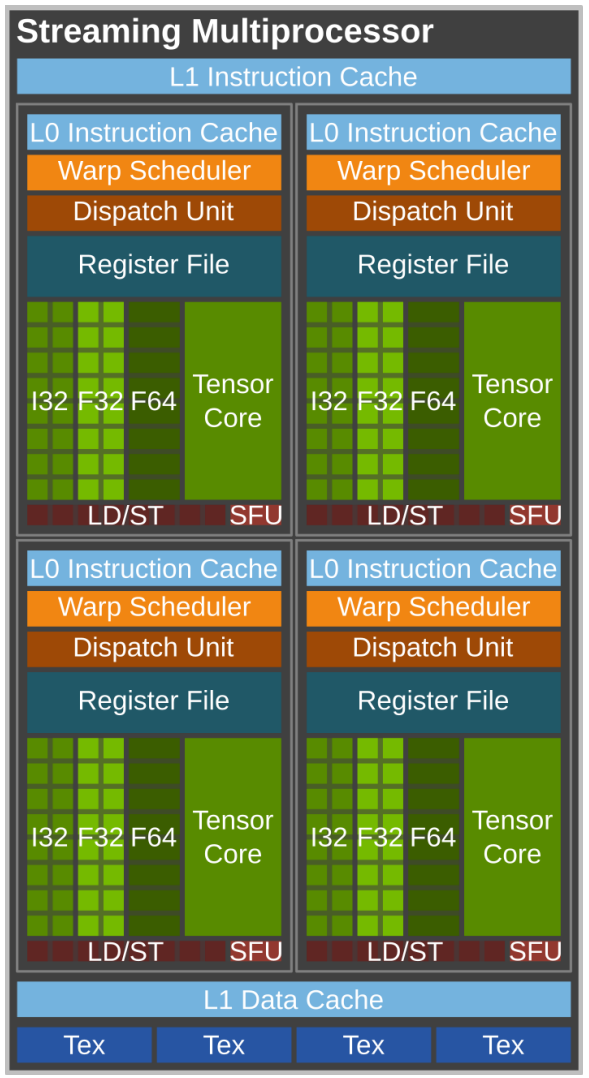
Processing Flow

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



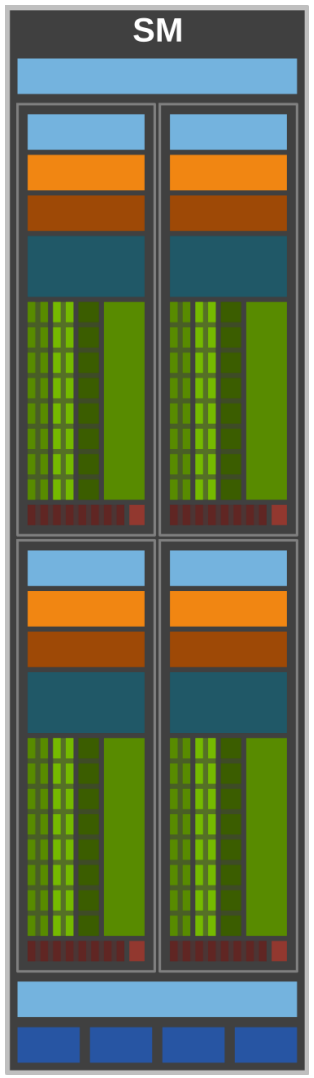
Source: PRACE P1_13

Processing on GPU hardware



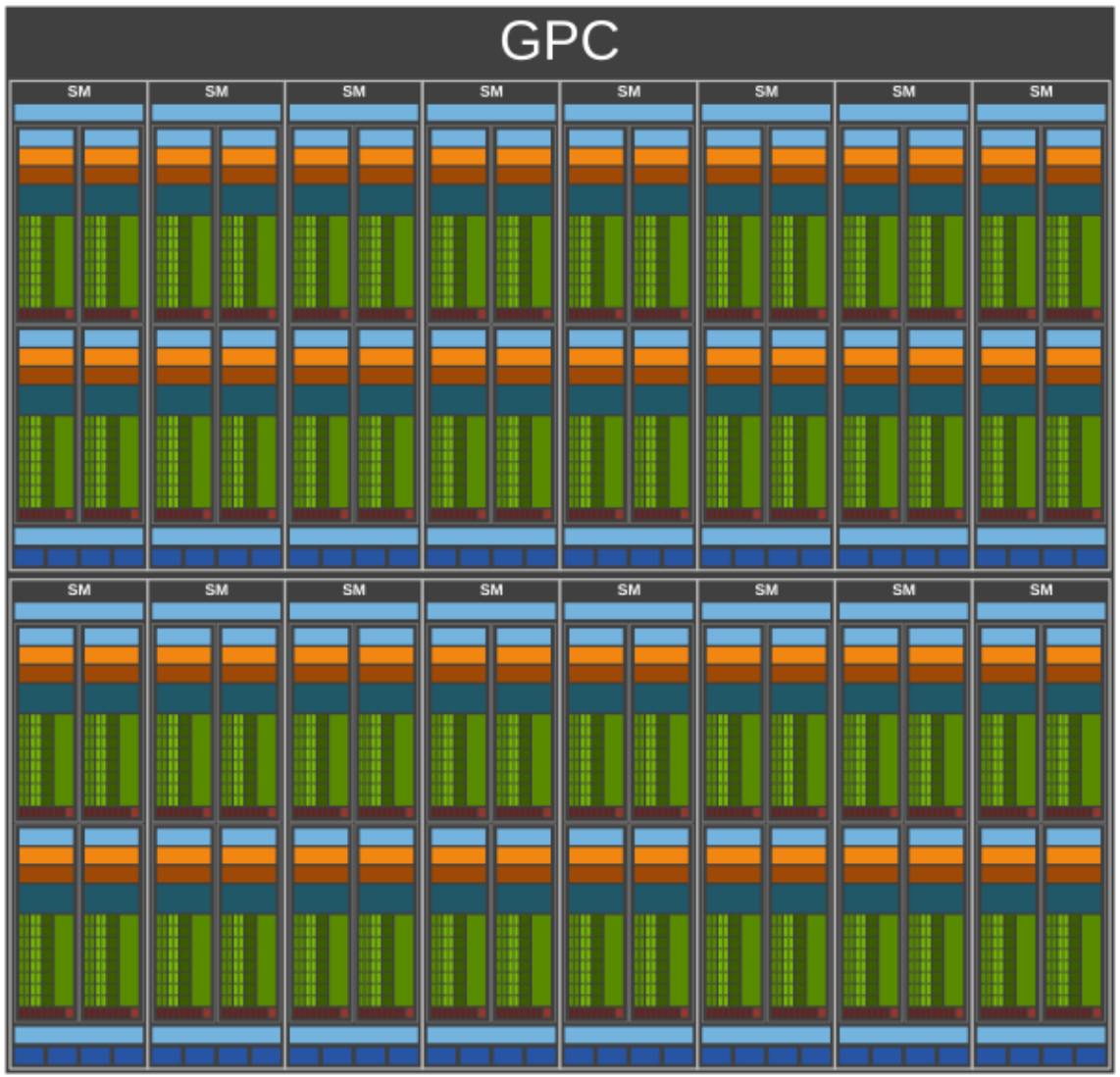
Source: PRACE P1_17

Processing on GPU hardware



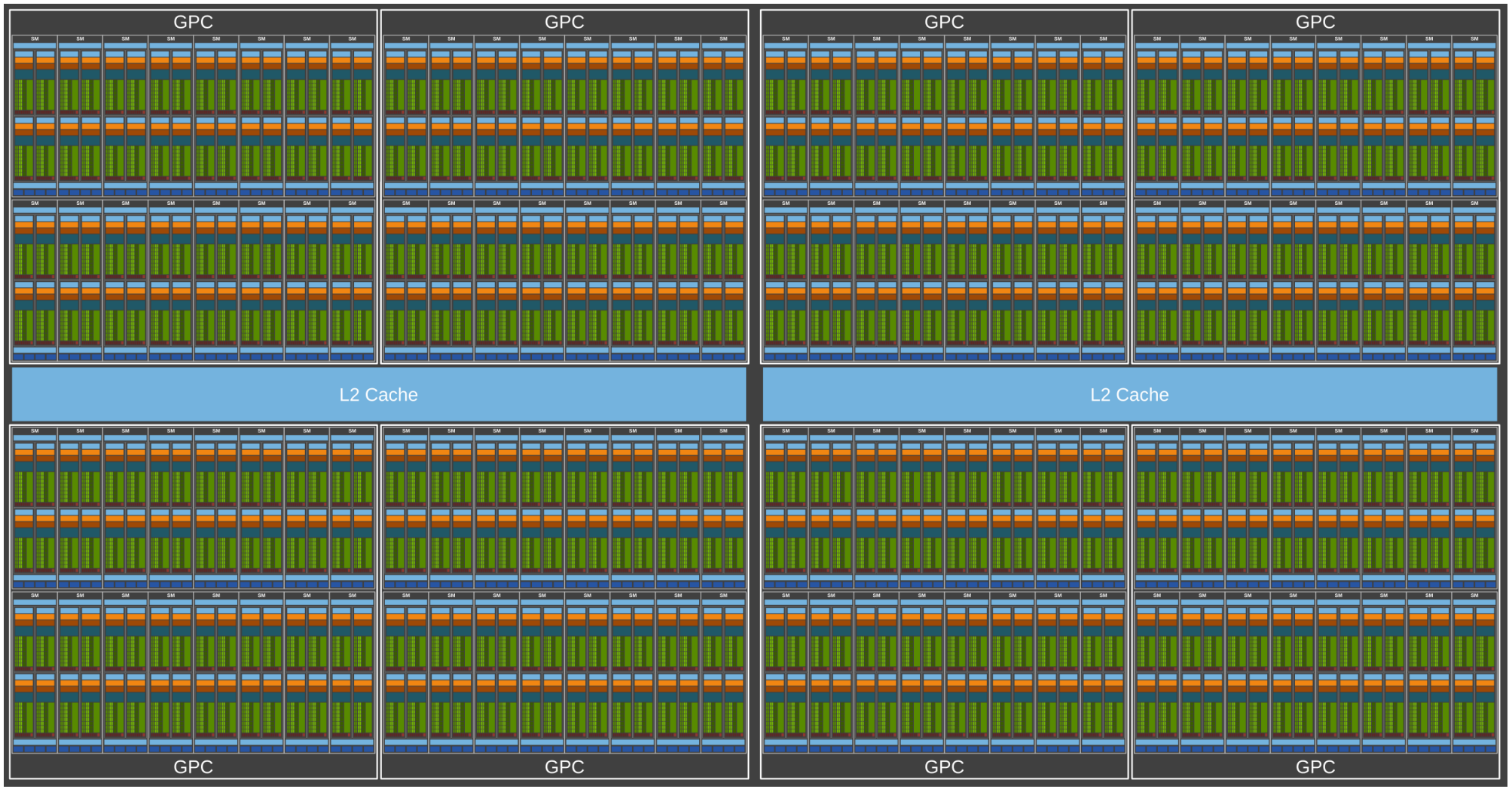
Source: PRACE P1_17

Processing on GPU hardware



Source: PRACE P1_17

Processing on GPU hardware

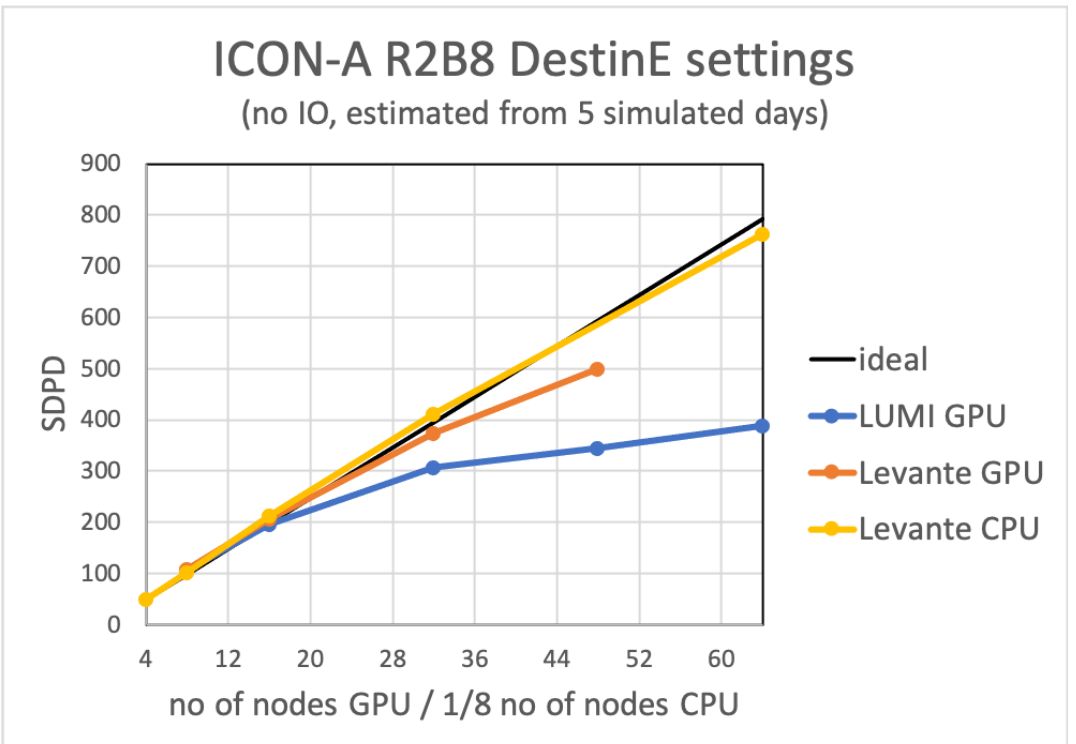


Source: PRACE P1_17

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

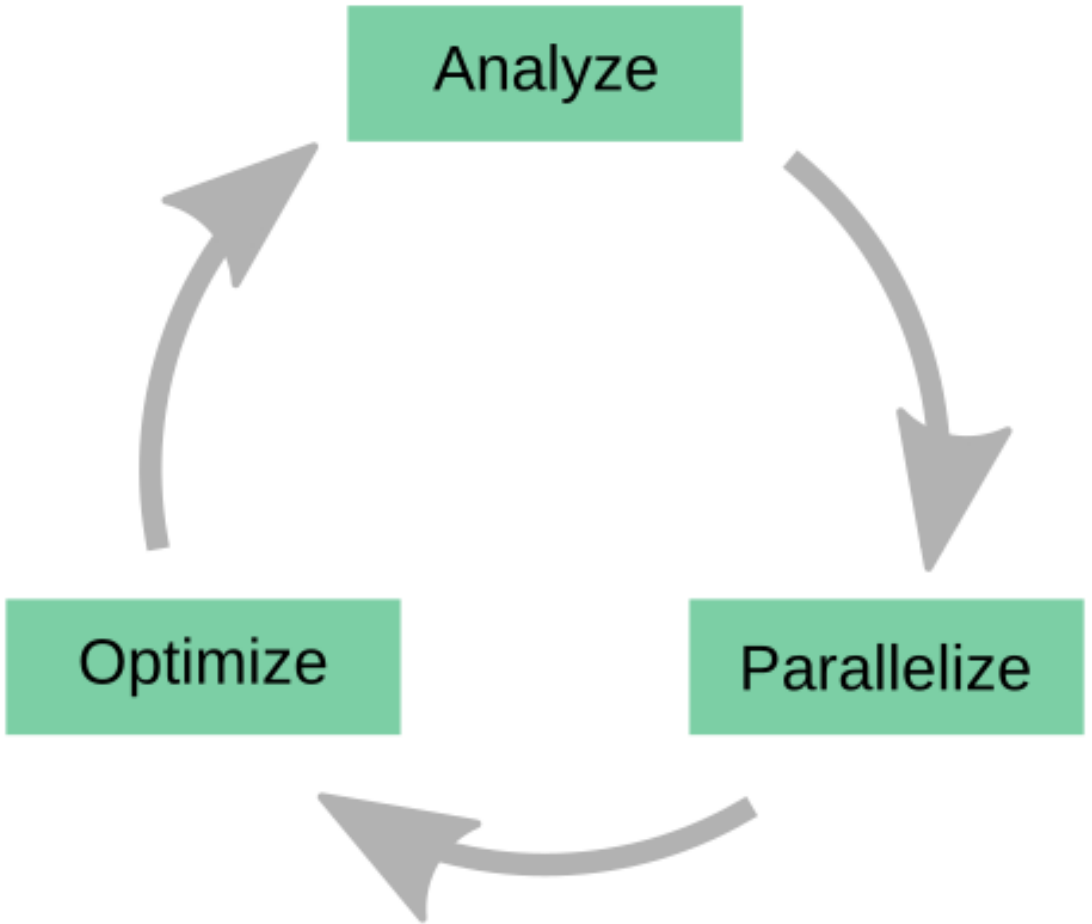
Optimized for **high throughput**

+ 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

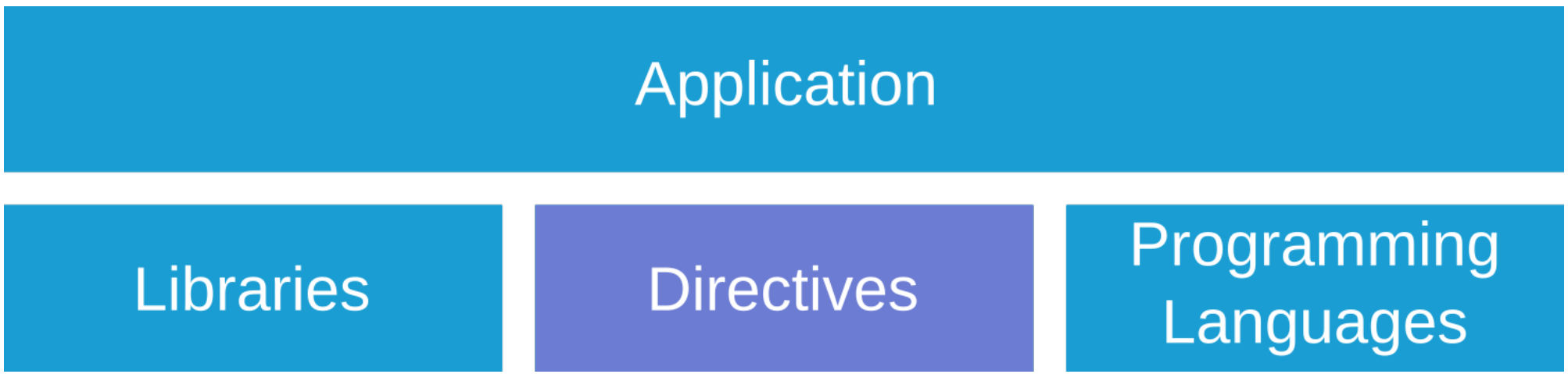
Porting workflow cycle



This workshop mostly focuses on parallelization

Source: OpenACC O2_12

Offloading possibilities



Drop-In Acceleration

Easy Acceleration

Flexible Acceleration

Source: PRACE P1_21

OpenACC introduction

Concept of directives in source code

- Compiler directives state intent to compiler

```
1  #pragma acc kernels
2  for (int i = 0; i < 1023; i++) {
3     // ...
4  }
```

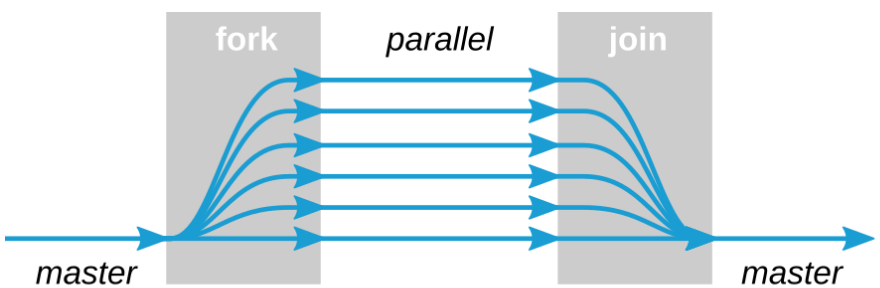
```
1  !$acc kernels
2  do i = 1, 1024
3     ! ...
4  end do
5  !$acc end kernels
```

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

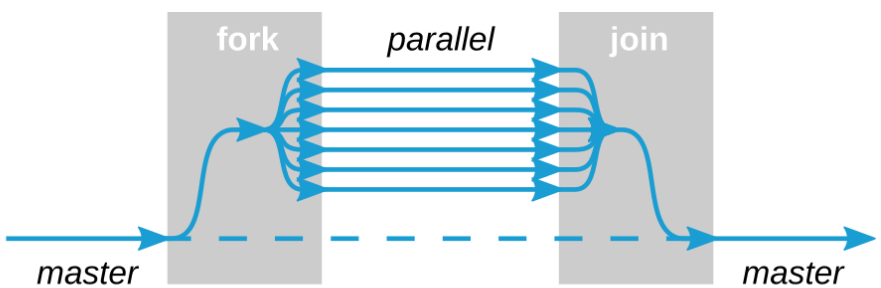
Source: PRACE P2_7

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



OpenMP



OpenACC

Source: PRACE P2_4

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

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

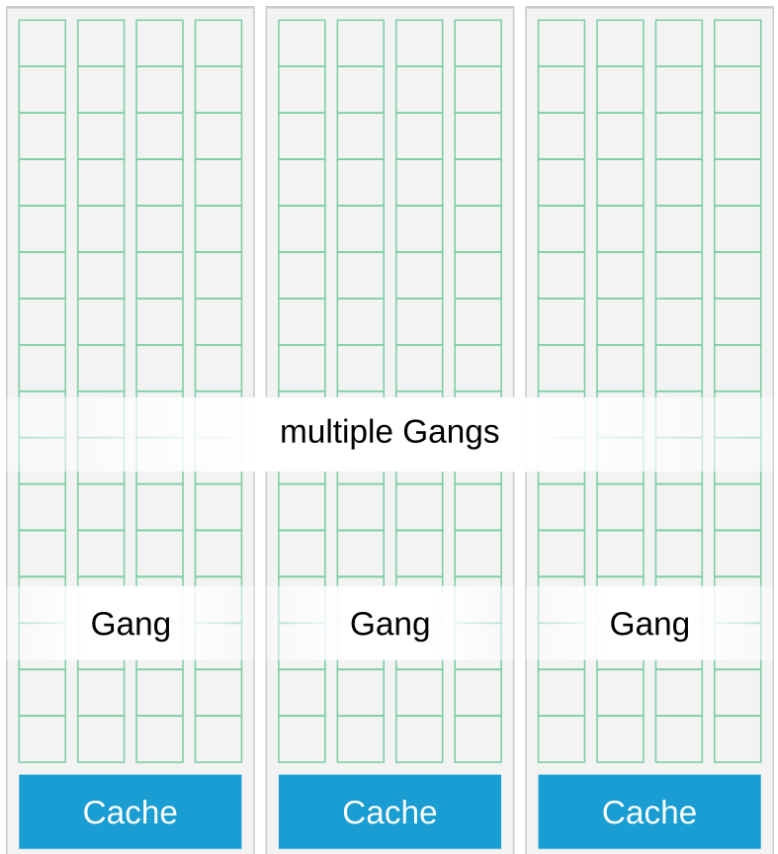
Operate on same cache

`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

```
 1  program fwd_dep
 2     implicit none
 3     integer, parameter :: nel = 10
 4     integer, dimension(nel) :: vec
 5     integer :: i
 6
 7     vec = [(i, i = 1, nel)]
 8     do i = 2, nel
 9        vec(i) = vec(i) + vec(i-1)
10     end do
11     write(*, '(9(i2,", "),i2)') vec
12  end program fwd_dep
```

Sequential result:

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

⇒ Forward loop dependency prevents parallelization

Source: OpenACC O1_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

```
1  do k = 1, size
2     do j = 1, size
3        do i = 1, size
4           c(i, j) = c(i, j) + a(i, k) * b
5        end do
6     end do
7  end do
```

```
1  !$acc parallel loop collapse(2)
2  do j = 1, size
3     do i = 1, size
4        tmp = c(i, j)
5        !$acc loop reduction(+: tmp)
6        do k = 1, size
7           tmp = tmp + a(i, k) * b(k, j)
8        end do
9        c(i, j) = tmp
10    end do
11 end do
12 !$acc end parallel loop
```

Source: OpenACC O3_29

Kernel construct

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

```
1  !$acc kernels
2  do i = 1, n
3     a(i) = 2.0 * b(i)
4  end do
5  !$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

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)

```
 1 program p_example
 2   implicit none
 3   integer, parameter :: dp = selected_real_
 4   integer, parameter :: nel = 1000
 5   real(dp), dimension(nel, nel) :: a
 6   integer :: i, j
 7
 8
 9
10   do i = 1, nel
11
12     do j = 1, nel
13       a(j, i) = 0.5_dp*(i*nel+j)
14     end do
15   end do
16
17
18   write(*, *) a(10, 10)
19 end program p_example
```

How to port the do-loops?

Example (2/6)

```
1  program p_example
2    implicit none
3    integer, parameter :: dp = selected_real_
4    integer, parameter :: nel = 1000
5    real(dp), dimension(nel, nel) :: a
6    integer :: i, j
7
8
9    !$acc parallel
10   do i = 1, nel
11
12     do j = 1, nel
13       a(j, i) = 0.5_dp*(i*nel+j)
14     end do
15   end do
16   !$acc end parallel
17
18   write(*, *) a(10, 10)
19 end program p_example
```

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

LOG

$\Rightarrow$ `parallel` alone does not parallelize

Example (3/6)

```
1  program p_example
2    implicit none
3    integer, parameter :: dp = selected_real_
4    integer, parameter :: nel = 1000
5    real(dp), dimension(nel, nel) :: a
6    integer :: i, j
7
8
9    !$acc parallel loop
10   do i = 1, nel
11
12     do j = 1, nel
13       a(j, i) = 0.5_dp*(i*nel+j)
14     end do
15   end do
16   !$acc end parallel loop
17
18   write(*, *) a(10, 10)
19 end program p_example
```

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

LOG

⇒ Works, but we can be more specific about distribution