# INTRODUCTION TO OPENACC
## NATESM TRAINING WORKSHOP

6 November 2024 | Andreas Herten, Kaveh Haghighi Mood | Forschungszentrum Jülich



JÜLICH Forschungszentrum

# Outline

# Open{MP↔ACC}

## Everything's connected

- OpenACC modeled after OpenMP …
- … but specific for accelerators
- OpenMP 4.0/4.5: Offloading; compiler support improving (Clang, XL, GCC, …)
- OpenACC more descriptive, OpenMP more prescriptive
- OpenMP 5.0: Descriptive directive `loop`
- Same basic principle: Fork/join model

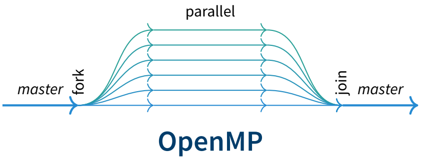
  *Master thread launches parallel child threads; merge after execution*

# Open{MP↔ACC}

## Everything's connected

- OpenACC modeled after OpenMP …
- … but specific for accelerators
- OpenMP 4.0/4.5: Offloading; compiler support improving (Clang, XL, GCC, …)
- OpenACC more descriptive, OpenMP more prescriptive
- OpenMP 5.0: Descriptive directive `loop`
- Same basic principle: Fork/join model

  *Master thread launches parallel child threads; merge after execution*

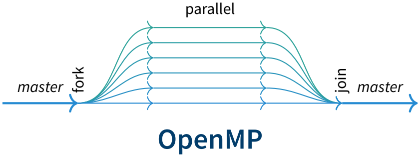


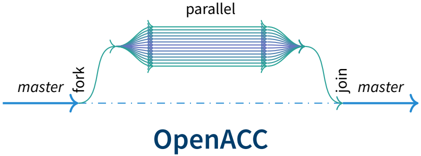
# Open{MP↔ACC}

## Everything's connected

- OpenACC modeled after OpenMP …
- … but specific for accelerators
- OpenMP 4.0/4.5: Offloading; compiler support improving (Clang, XL, GCC, …)
- OpenACC more descriptive, OpenMP more prescriptive
- OpenMP 5.0: Descriptive directive `loop`
- Same basic principle: Fork/join model

  *Master thread launches parallel child threads; merge after execution*



# Introduction
## Modus Operandi

# OpenACC Acceleration Workflow

## Three-step program

1. Annotate code with directives, indicating parallelism
2. OpenACC-capable compiler generates accelerator-specific code
3. $uccess

# 1 Directives

## pragmatic

- Compiler directives state intend to compiler

### C/C++

```c
#pragma acc kernels
for (int i = 0; i < 23; i++)
// ...
```

### Fortran

```fortran
!$acc kernels
do i = 1, 24
! ...
!$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

# 2 Compiler

## Simple and abstracted

- Trust compiler to generate intended parallelism; always check status output!
- No need to know details of accelerator; leave it to expert compiler engineers *Tuning possible*
- One code can target different accelerators: GPUs, CPUs → **Portability**

# 2 Compiler

## Simple and abstracted

- Trust compiler to generate intended parallelism; always check status output!
- No need to know details of accelerator; leave it to expert compiler engineers *Tuning possible*
- One code can target different accelerators: GPUs, CPUs → **Portability**

| Compiler | Targets | Languages | OSS | Free | Comment |
|---|---|---|---|---|---|
| NVIDIA HPC SDK | NVIDIA GPU, CPU | C, C++, Fortran | No | Yes | Best performance |
| GCC | NVIDIA GPU, AMD GPU | C, C++, Fortran | Yes | Yes | |
| HPE Cray | NVIDIA GPU | Fortran | No | No | *???* |
| Clang/LLVM | CPU, NVIDIA GPU | C, C++. *Fortran* | Yes | Yes | Via Clang OpenMP backend |

# 2 Compiler

## Simple and abstracted

- Trust compiler to generate intended parallelism; always check status output!
- No need to know details of accelerator; leave it to expert compiler engineers *Tuning possible*
- One code can target different accelerators: GPUs, CPUs → **Portability**

| Compiler | Targets | Languages | OSS | Free | Comment |
|---|---|---|---|---|---|
| NVHPC | NVIDIA GPU, CPU | C, C++, Fortran | No | Yes | Best performance |
| GCC | NVIDIA GPU, AMD GPU | C, C++, Fortran | Yes | Yes | |
| HPE Cray | NVIDIA GPU | Fortran | No | No | *???* |
| Clang/LLVM | CPU, NVIDIA GPU | C, *C++. Fortran* | Yes | Yes | Via Clang OpenMP backend |

# 2 Compiler

## Flags and options

OpenACC compiler support: activate with compile flag

NVHPC `nvc -acc`

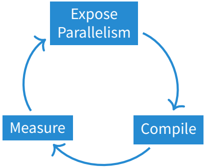
| | |
|---|---|
| `-acc=gpu\|multicore` | Target GPU or CPU |
| `-acc=gpu -gpu=cc80` | Generate Ampere-compatible code |
| `-gpu=cc80,lineinfo` | Add source code correlation into binary |
| `-gpu=managed` | Use unified memory |
| `-Minfo=accel` | Print acceleration info |

GCC `gcc -fopenacc`

| | |
|---|---|
| `-fopenacc-dim=geom` | Use *geom* configuration for threads |
| `-foffload="-lm -O3"` | Provide flags to offload compiler |
| `-fopt-info-omp` | Print acceleration info |

# 3 $uccess

## Iteration is key




- Serial to parallel: fast
- Serial to fast parallel: more time needed
- Start simple → refine
- Expose more and more parallelism

⇒ **Productivity**

- Because of *generality*: Sometimes not last bit of hardware performance accessible
- But: Use OpenACC together with other accelerator-targeting techniques (CUDA, libraries, …)

# A Glimpse of OpenACC

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

```fortran
!$acc data copy(x(1:N),y(1:N))
!$acc parallel loop

    do i = 1, N
        x(i) = 1.0
        y(i) = 2.0
    end do
    do i = 1, N
        y(i) = i*x(i)+y(i);
    end do

!$acc end parallel loop
!$acc end data
```

# Parallel Loops: Parallel

## An important directive

- Programmer identifies block containing parallelism
  $\rightarrow$ compiler generates offload code
- Program launch creates *gangs* of parallel threads on parallel device
- Implicit barrier at end of parallel region
- Each gang executes same code sequentially

**OpenACC: `parallel`** C

```
#pragma acc parallel [clause, [, clause] ...] newline
{structured block}
```

# Parallel Loops: Parallel

## An important directive

- Programmer identifies block containing parallelism
  → compiler generates offload code
- Program launch creates *gangs* of parallel threads on parallel device
- Implicit barrier at end of parallel region
- Each gang executes same code sequentially

**OpenACC: `parallel`** F

```
!$acc parallel [clause, [, clause] ...]
!$acc end parallel
```

# Parallel Loops: Parallel

## An important directive

- Programmer identifies block containing parallelism
  $\rightarrow$ compiler generates offload code
- Program launch creates *gangs* of parallel threads on parallel device
- Implicit barrier at end of parallel region
- Each gang executes same code sequentially

**OpenACC: `parallel`** F

```
!$acc parallel [clause, [, clause] ...]
!$acc end parallel
```

# Parallel Loops: Parallel

## Clauses

Diverse clauses to augment the parallel region

| | |
| --- | --- |
| `private(var)` | A copy of variables `var` is made for each gang |
| `firstprivate(var)` | Same as `private`, except `var` will initialized with value from host |
| `if(cond)` | Parallel region will execute on accelerator only if `cond` is true |
| `reduction(op:var)` | Reduction is performed on variable `var` with operation `op`; supported: `+ * max min …` |
| `async[(int)]` | No implicit barrier at end of parallel region |



JÜLICH Forschungszentrum

# Parallel Loops: Loops

## Also an important directive

- Programmer identifies loop eligible for parallelization
- Directive must be directly before loop
- Optional: Describe type of parallelism

**OpenACC: `loop`** C

```
#pragma acc loop [clause, [, clause] ...] newline
{structured block}
```

# Parallel Loops: Loops

## Also an important directive

- Programmer identifies loop eligible for parallelization
- Directive must be directly before loop
- Optional: Describe type of parallelism

**OpenACC: `loop`** F

```
!$acc loop [clause, [, clause] ...]
!$acc end loop
```

# Parallel Loops: Loops

## Clauses

| | |
|---|---|
| `independent` | Iterations of loop are data-independent (implied if in `parallel` region (and no `seq` or `auto`)) |
| `collapse(int)` | Collapse `int` tightly-nested loops |
| `seq` | This loop is to be executed sequentially (not parallel) |
| `tile(int[,int])` | Split loops into loops over tiles of the full size |
| `auto` | Compiler decides what to do |

# Parallel Loops: Parallel Loops

## Maybe the most important directive

- Combined directive: shortcut
  *Because its used so often*
- Any clause that is allowed on `parallel` or `loop` allowed
- Restriction: May not appear in body of another parallel region

**OpenACC: `parallel loop`** C

```
#pragma acc parallel loop [clause, [, clause] ...] newline
{structured block}
```

# Parallel Loops: Parallel Loops

## Maybe the most important directive

- Combined directive: shortcut
  *Because its used so often*
- Any clause that is allowed on `parallel` or `loop` allowed
- Restriction: May not appear in body of another parallel region

**OpenACC: `parallel loop`** F

```
!$acc parallel loop [clause, [, clause] ...]
!$acc end parallel loop
```

# Parallel Loops: Parallel Loops

## Maybe the most important directive

- Combined directive: shortcut
  *Because its used so often*
- Any clause that is allowed on `parallel` or `loop` allowed
- Restriction: May not appear in body of another parallel region

**OpenACC: `parallel loop`**

```
#pragma acc parallel loop [clause, [, clause] ...]
```

# Parallel Loops Example

```c
double sum = 0.0;
#pragma acc parallel loop
for (int i=0; i<N; i++) {
    x[i] = 1.0;
    y[i] = 2.0;
}

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

```fortran
sum = 0.0
!$acc parallel loop
do i = 1, N
    x(i) = 1.0
    y(i) = 2.0
end do
!$acc end parallel loop
!$acc parallel loop reduction(+:sum)
do i = 1, N
    y(i) = i*x(i)+y(i)
    sum+=y(i)
end do
!$acc end parallel loop
```

# Parallel Loops Example

```c
double sum = 0.0;
#pragma acc parallel loop
for (int i=0; i<N; i++) {
    x[i] = 1.0;
    y[i] = 2.0;
}

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

```fortran
sum = 0.0
!$acc parallel loop
do i = 1, N
    x(i) = 1.0
    y(i) = 2.0
end do
!$acc end parallel loop
!$acc parallel loop reduction(+:sum)
do i = 1, N
    y(i) = i*x(i)+y(i)
    sum+=y(i)
end do
!$acc end parallel loop
```

Kernel 1

Kernel 2

# More Parallelism: Kernels

## More freedom for compiler

- Kernels directive: second way to expose parallelism
- Region may contain parallelism
- Compiler determines parallelization opportunities

→ More freedom for compiler

- Rest: Same as for `parallel`

**OpenACC: `kernels`**

```
#pragma acc kernels [clause, [, clause] ...]
```

# Kernels Example

```
double sum = 0.0;
#pragma acc kernels
{
for (int i=0; i<N; i++) {
    x[i] = 1.0;
    y[i] = 2.0;
}
for (int i=0; i<N; i++) {
    y[i] = i*x[i]+y[i];
    sum+=y[i];
}
}
```

Kernels created here

# `kernels` vs. `parallel`

- Both approaches equally valid; can perform equally well

# `kernels` vs. `parallel`

- Both approaches equally valid; can perform equally well
- `kernels`
  - Compiler performs parallel analysis
  - Can cover large area of code with single directive
  - Gives compiler additional leeway
- `parallel`
  - Requires parallel analysis by programmer
  - Will also parallelize what compiler may miss
  - More explicit
  - Similar to OpenMP

# `kernels` vs. `parallel`

- Both approaches equally valid; can perform equally well
- `kernels`
  - Compiler performs parallel analysis
  - Can cover large area of code with single directive
  - Gives compiler additional leeway
- `parallel`
  - Requires parallel analysis by programmer
  - Will also parallelize what compiler may miss
  - More explicit
  - Similar to OpenMP
- Both regions may not contain other `kernels`/`parallel` regions
- No branching into or out
- Program must not depend on order of evaluation of clauses
- At most: One `if` clause

# Data Regions

## Structured Data Regions

- Defines region of code in which data remains on device
- Data is shared among all kernels in region
- Explicit data transfers

**OpenACC: `data`**

```
#pragma acc data [clause, [, clause] ...]
```

# Data Regions

## Clauses

Clauses to augment the data regions

| | |
|---|---|
| `copy(var)` | Allocates memory of `var` on GPU, copies data to GPU at beginning of region, copies data to host at end of region<br>Specifies size of `var`: `var[`*`lowerBound`*`:`*`size`*`]` |
| `copyin(var)` | Allocates memory of `var` on GPU, copies data to GPU at beginning of region |
| `copyout(var)` | Allocates memory of `var` on GPU, copies data to host at end of region |
| `create(var)` | Allocates memory of `var` on GPU |
| `present(var)` | Data of `var` is not copies automatically to GPU but considered present |

# Data Region Example

```c
#pragma acc data copyout(y[0:N]) create(x[0:N])
{
double sum = 0.0;
#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 data copyout(y(1:N)) create(x(1,N))

sum = 0.0;
!$acc parallel loop
do i = 1, N
    x(i) = 1.0
    y(i) = 2.0
end do
!$acc end parallel loop
!$acc parallel loop
do i = 1, N
    y(i) = i*x(i)+y(i)
end do
!$acc end parallel loop
!$acc end data
```

# Further Keywords

## Directives

| | |
|---|---|
| `serial` | Serial GPU Region |
| `wait` | Wait for any async operation |
| `atomic` | Atomically access data (no interference of concurrent accesses) |
| `cache` | Fetch data to GPU caches |
| `declare` | Make data live on GPU for implicit region directly after variable declaration |
| `update` | Update device data |
| `shutdown` | Shutdown connection to GPU |

# Further Keywords

## Directives

`serial` Serial GPU Region

`wait` Wait for any async operation

`atomic` Atomically access data (no interference of concurrent accesses)

`cache` Fetch data to GPU caches

`declare` Make data live on GPU for implicit region directly after variable declaration

`update` Update device data

`shutdown` Shutdown connection to GPU

## Clauses

`gang worker vector` Type of parallelism

`collapse` Combine tightly-nested loops

`tile` Split loop into two loops

`(first)private` Create thread-private data (and init)

`attach` Reference counting for data pointers

`async` Schedule operation asynchronously

# Further Keywords

## Directives

`serial` Serial GPU Region

`wait` Wait for any async operation

`atomic` Atomically access data (no interference of concurrent accesses)

`cache` Fetch data to GPU caches

`declare` Make data live on GPU for implicit region directly after variable declaration

`update` Update device data

`shutdown` Shutdown connection to GPU

## Clauses

**`gang worker vector`** Type of parallelism

`collapse` Combine tightly-nested loops

`tile` Split loop into two loops

`(first)private` Create thread-private data (and init)

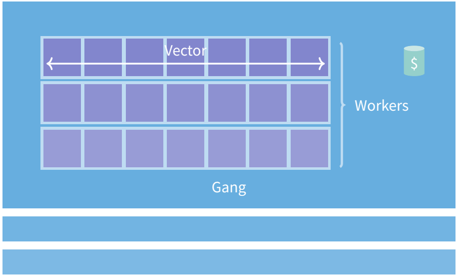
`attach` Reference counting for data pointers

`async` Schedule operation asynchronously

# Launch Configuration

## Specify number of threads and blocks

- 3 **clauses** for changing distribution of group of threads (clauses of parallel region (`parallel`, `kernels`))
- Presence of keyword: Distribute using this level
- Optional size: Control size of parallel entity




**OpenACC: `gang worker vector`**

```
#pragma acc parallel loop gang worker vector
```

*Size:* `num_gangs(n)`, `num_workers(n)`, `vector_length(n)`

# Exercise

Task

- See `$HOME/natESM/GPU-Course/OpenACC`
- Read instructions!
- Solutions given; you tinker as long as you want, then ask or check solutions
- Timeline reminder
  - CUDA until coffee break; solutions after break
  - OpenACC until lunch, solutions before/after?
  - Kokkos in afternoon

# Conclusions

# Conclusions

- OpenACC directives and clauses
  `#pragma acc parallel loop copyin(A[0:N]) reduction(max:err) vector`
- Start easy, optimize from there; express as much parallelism as possible
- Optimize data for locality, prevent unnecessary movements
- OpenACC is interoperable to other GPU programming models

# Conclusions

- OpenACC directives and clauses
  `#pragma acc parallel loop copyin(A[0:N]) reduction(max:err) vector`
- Start easy, optimize from there; express as much parallelism as possible
- Optimize data for locality, prevent unnecessary movements
- OpenACC is interoperable to other GPU programming models

*Thank you for your attention!*

*a.herten@fz-juelich.de*

# Appendix

# List of Tasks

# Glossary I

**AMD** Manufacturer of CPUs and GPUs. 9, 10, 11

**Ampere** GPU architecture from NVIDIA (announced 2019). 12

**CUDA** Computing platform for GPUs from NVIDIA. Provides, among others, CUDA C/C++. 13

**GCC** The GNU Compiler Collection, the collection of open source compilers, among others for C and Fortran. 12

**LLVM** An open Source compiler infrastructure, providing, among others, Clang for C. 9, 10, 11

**NVHPC** NVIDIA HPC SDK; Collection of GPU-capable compilers and libraries. Formerly known as PGI.. 12

# Glossary II

**NVIDIA** US technology company creating GPUs. 9, 10, 11, 45, 46

**OpenACC** Directive-based programming, primarily for many-core machines. 3, 4, 5, 7, 8, 12, 13, 14, 15, 16, 17, 19, 20, 22, 23, 24, 27, 32, 38, 41, 42

**OpenMP** Directive-based programming, primarily for multi-threaded machines. 3, 4, 5, 9, 10, 11, 29, 30, 31

**PGI** Compiler creators. Formerly *The Portland Group, Inc.*; since 2013 part of NVIDIA. 45

**POWER** CPU architecture from IBM, earlier: PowerPC. See also POWER8. 46

**POWER8** Version 8 of IBM's POWER processor, available also within the OpenPOWER Foundation. 46

**CPU** Central Processing Unit. 9, 10, 11, 45, 46

**GPU** Graphics Processing Unit. 9, 10, 11, 33, 41, 42, 45, 46

# References I

# References: Images, Graphics