Outline

OpenACC Overview
  GPU Acceleration Possibilities
  History
  OpenMP
  Modus Operandi
  OpenACC’s Models

OpenACC Directives
  Parallelize Loops
    parallel
    loops
    kernels
  Data Transfers
    Clause: copy
data
    enter data
  More Directives
    Clause: gang
    Clause: routine
    Directive: host_data use_device

OpenACC Infrastructure
  Software
  Using OpenACC on JUWELS
  OpenMP
  Conclusions
  List of Tasks
GPU Acceleration Possibilities

Application

Libraries

Abstractions

Programming Languages

Drop-in Acceleration

Easy Acceleration

Flexible Acceleration
GPU Acceleration Possibilities

- **Libraries**
  - *Drop-in Acceleration*

- **OpenACC**

- **Programming Languages**
  - *Easy Acceleration*
  - *Flexible Acceleration*
OpenACC Mission Statement

[...] OpenACC [is] for writing parallel programs in C, C++, and Fortran that run identified regions in parallel on multicore CPUs or attached accelerators.

[...] a model for parallel programming that is portable across operating systems and various types of multicore CPUs and accelerators.

– OpenACC API Documentation, openacc.org
OpenACC History

2011  OpenACC 1.0 specification is released at SC11
      *NVIDIA, Cray, PGI, CAPS*

2013  OpenACC 2.0: More functionality, portability

2015  OpenACC 2.5: Enhancements, clarifications

2017  OpenACC 2.6: Deep copy, …

2018  OpenACC 2.7: More host, reductions, …

2019  OpenACC 3.0: Newer C++, more lambdas, …
      - Run as a non-profit organization, OpenACC.org
      - Members from industry and academia

→ https://www.openacc.org/ (see also: Best practice guide)

OpenACC-enabled Applications

- ANSYS Fluent
- Gaussian
- VASP
- COSMO
- GTC
- SOMA
- …
OpenMP ↔ 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*
OpenACC Overview

Modus Operandi
OpenACC Acceleration Workflow
Three-step program

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

Compiler directives state intent 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

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Targets</th>
<th>Languages</th>
<th>OSS</th>
<th>Free</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>PGI</td>
<td>NVIDIA GPU, CPU</td>
<td>C, C++, Fortran</td>
<td>No</td>
<td>Yes</td>
<td>Best performance</td>
</tr>
<tr>
<td>GCC</td>
<td>NVIDIA GPU, AMD GPU</td>
<td>C, C++, Fortran</td>
<td>Yes</td>
<td>Yes</td>
<td>AMD support coming up</td>
</tr>
<tr>
<td>Cray</td>
<td>NVIDIA GPU</td>
<td>C, C++</td>
<td>No</td>
<td>No</td>
<td>???</td>
</tr>
<tr>
<td>Clang/LLVM</td>
<td>CPU, NVIDIA GPU</td>
<td>C, C++</td>
<td>Yes</td>
<td>Yes</td>
<td>Via Clang OpenMP backend; very fresh!</td>
</tr>
</tbody>
</table>
Compiler

Flags and options

OpenACC compiler support: activate with compile flag

**PGI**  `pgcc -acc`
- `-ta=tesla|-ta=multicore` Target GPU or CPU
- `-ta=tesla:cc70` Generate Volta-compatible code
- `-ta=tesla:lineinfo` Add source code correlation into binary
- `-ta=tesla: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
Iteration is key

- Serial to parallel: fast
- Serial to fast parallel: more time needed
- Start simple $\rightarrow$ refine
- Expose more and more parallelism

$\Rightarrow$ Productivity

- Because of generality: Sometimes not last bit of hardware performance accessible
- But: Use OpenACC together with other accelerator-targeting techniques (CUDA, libraries, …)
OpenACC Accelerator Model
For computation and memory spaces

- Main program executes on host
- Device code is transferred to accelerator
- Execution on accelerator is started
- Host waits until return (except: async)

- Two separate memory spaces; data transfers back and forth
  - Transfers hidden from programmer
  - Memories not coherent!
  - Compiler helps; GPU runtime helps
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];
    }
}

!$acc data copy(x(1:N),y(1:N))
!$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
```
OpenACC Directives
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

#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**

```
!$acc parallel [clause, [, clause] ...]
!$acc end parallel
```
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: 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] ...] newline
{structured block}
```
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];
}

sum = 0.0
#pragma acc parallel loop
do i = 1, N
    x(i) = 1.0
    y(i) = 2.0
end do
#pragma acc end parallel loop
!

Sum = 0.0
#pragma acc parallel loop
do i = 1, N
    y(i) = i*x(i)+y(i)
    sum+=y(i)
end do
#pragma acc end parallel loop
Compilation Result

PGI

```
$ pgcc -acc -ta=tesla:cc70 -Minfo=accel -o reduce-parallel.exe reduce-parallel.c
...
$ srun nsys profile --stats=true ./reduce-parallel.exe  # N = 2000000
...
```

CUDA Kernel Statistics (nanoseconds)

<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Total Time</th>
<th>Instances</th>
<th>Average</th>
<th>Minimum</th>
<th>Maximum</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>43.1</td>
<td>37216</td>
<td>1</td>
<td>37216.0</td>
<td>37216</td>
<td>37216</td>
<td>main_13_gpu</td>
</tr>
<tr>
<td>32.6</td>
<td>28160</td>
<td>1</td>
<td>28160.0</td>
<td>28160</td>
<td>28160</td>
<td>main_8_gpu</td>
</tr>
<tr>
<td>24.2</td>
<td>20928</td>
<td>1</td>
<td>20928.0</td>
<td>20928</td>
<td>20928</td>
<td>main_13_gpu__red</td>
</tr>
</tbody>
</table>

CUDA Memory Operation Statistics (nanoseconds)

<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Total Time</th>
<th>Instances</th>
<th>Average</th>
<th>Minimum</th>
<th>Maximum</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>58.6</td>
<td>1842336</td>
<td>4</td>
<td>460584.0</td>
<td>1856</td>
<td>616032</td>
<td>[CUDA memcpyDtoH]</td>
</tr>
<tr>
<td>41.3</td>
<td>1298912</td>
<td>2</td>
<td>649456.0</td>
<td>649152</td>
<td>649760</td>
<td>[CUDA memcpyHtoD]</td>
</tr>
<tr>
<td>0.1</td>
<td>1888</td>
<td>1</td>
<td>1888.0</td>
<td>1888</td>
<td>1888</td>
<td>[CUDA memset]</td>
</tr>
</tbody>
</table>
Compilation Result

GCC

```bash
$ gcc -fopenacc -fopt-info-omp -o reduce-parallel.exe reduce-parallel.c
...
$ srun nsys profile --stats=true ./reduce-parallel.exe # N = 2000000
...
```

**CUDA Kernel Statistics (nanoseconds)**

<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Total Time</th>
<th>Instances</th>
<th>Average</th>
<th>Minimum</th>
<th>Maximum</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>99.4</td>
<td>5669158</td>
<td>1</td>
<td>5669158.0</td>
<td>5669158</td>
<td>5669158</td>
<td>main$_omp_fn$1</td>
</tr>
<tr>
<td>0.6</td>
<td>36256</td>
<td>1</td>
<td>36256.0</td>
<td>36256</td>
<td>36256</td>
<td>main$_omp_fn$0</td>
</tr>
</tbody>
</table>

**CUDA Memory Operation Statistics (nanoseconds)**

<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Total Time</th>
<th>Instances</th>
<th>Average</th>
<th>Minimum</th>
<th>Maximum</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>54.7</td>
<td>7373929</td>
<td>5</td>
<td>1474785.8</td>
<td>2112</td>
<td>2627491</td>
<td>[CUDA memcpyDtoH]</td>
</tr>
<tr>
<td>45.3</td>
<td>6104103</td>
<td>7</td>
<td>872014.7</td>
<td>1472</td>
<td>1608162</td>
<td>[CUDA memcpyHtoD]</td>
</tr>
</tbody>
</table>
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

```c
#pragma acc kernels [clause, [, clause] ...]
```
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];
    }
}

$ pgcc -acc -ta=tesla:cc70 -Minfo=accel -o reduce-kernels.exe reduce-kernels.c
main:
    8, Generating implicit copyout(y[:],x[:])
    10, Loop is parallelizable
        Generating Tesla code
        10, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
    14, Loop is parallelizable
        Generating Tesla code
        14, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
    16, Generating implicit reduction(+:sum)
$ gcc -fopenacc -fopt-info-omp -o reduce-kernels.exe reduce-kernels.c
reduce-kernels.c:8:10: optimized: assigned OpenACC seq loop parallelism
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 Transfers

- Automated transfers: `-ta=tesla:managed! (PGI)`
  - Driver automatically determines needed memory
  - Page-faulting mechanism
  - Data correctness

- Manual transfers
  - More fine-grained control
  - Increased portability
  - Can perform better in many cases
Copy Clause

- Clause to parallel or kernels region
- Which variable to copy to/from device

```
OpenACC: copy

#pragma acc parallel copy(A[start:length])
Also: copyin(B[s:l]) copyout(C[s:l]) present(D[s:l]) create(E[s:l])
```
Data Regions
To manually specify data locations

- Defines **region** of code in which data remains on device
- Data is shared among all kernels in region
- Explicit data transfers
- Clauses like before: copy, copyin, ...

#### OpenACC: data

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

$ pgcc -acc -ta=tesla:cc70 -Minfo=accel -o reduce-data.exe reduce-data.c
main:
     8, Generating create(x[:])
     Generating copyout(y[:])
    10, Generating Tesla code
        11, #pragma acc loop gang, vector(128) /*
         blockIdx.x threadIdx.x */
    15, Generating Tesla code
        16, #pragma acc loop gang, vector(128) /*
         blockIdx.x threadIdx.x */
             Generating reduction(+:sum)
    15, Generating implicit copy(sum)
Data Regions II
Explicit copies: enter data directive

- Define data regions, but not for structured block
- Clauses executed at the very position the directive encountered
- Closest to cudaMemcpy()
- Still, explicit data transfers

OpenACC: enter data

```
#pragma acc enter data [clause, [, clause] ...]
#pragma acc exit data [clause, [, clause] ...]
```
OpenACC Directives

More Directives
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

```plaintext
#pragma acc parallel loop gang worker vector
Size: num_gangs(n), num_workers(n), vector_length(n)
```
Accelerated Routines

- Enable functions/sub-routines for acceleration
- Make routine callable from device (CUDA: `__device__`)
- Needed for binding, refactoring, modular designs, ...

**OpenACC: routine**

```c
#pragma acc routine (name) [clause, [, clause] ...]
```
Interfacing to Other GPU Functions

- OpenACC hides handling of CPU and GPU memory pointer from user
- OpenACC uses appropriate pointer in accelerated kernel
- Interface to external GPU-accelerated routines: Use device-version of data pointer for scope of structured block
  → Interoperate with GPU libraries (cuBLAS, cuFFT, CUDA, …)
- Also: `deviceptr(ptr)`, when externally-allocated memory is passed to OpenACC

```c
#pragma acc host_data use_device(ptr)
```

Member of the Helmholtz Association
4 February 2020
Slide 31/39
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
- **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
OpenACC Infrastructure
GPU Tools

- OpenACC only interface
- Backend: CUDA (for GPUs)
- Use all CUDA tools
- Profiler:
  - Legacy `nvprof`, Visual Profiler *(to be deprecated soon)*
  - New Nsight Systems, Nsight Compute
- Debugger: `cuda-gdb`, `cuda-memcheck`
- Also: PGI environment variables
  - `PGI_ACC_TIME` Lightweight command-line profiler
  - `PGI_ACC_NOTIFY` Print GPU-related events
GPU Tools

- OpenACC
- Backend: CUDA (for GPUs)
  → Use all CUDA tools
  - Profiler:
    - Legacy nvprof, Visual Profiler (to be deprecated soon)
    - New Nsight Systems, Nsight Compute
  - Debugger:
    - cuda-gdb, cuda-memcheck
  - Also: PGI environment variables
    - PGI_ACC_TIME
      - Lightweight command-line profiler
    - PGI_ACC_NOTIFY
      - Print GPU-related events

Member of the Helmholtz Association
GPU Tools

- OpenACC only interface
- Backend: CUDA (for GPUs)
- Use all CUDA tools

$ PGI_ACC_NOTIFY=3 srun ./poisson2d
upload CUDA data file=/p/project/cjsc/aherten/MSA-Seminar-OpenACC/poisson2d.c
  function=main line=104 device=0 threadid=1 variable=A bytes=16777216
...
launch CUDA kernel file=/p/project/cjsc/aherten/MSA-Seminar-OpenACC/poisson2d.c
  function=main line=110 device=0 threadid=1 num_gangs=2046 num_workers=1 vector_length=128
  grid=2046 block=128 shared memory=2048
...

PGI_ACC_NOTIFY Print GPU-related events
- OpenACC only interface
- Backend: CUDA (for GPUs)

→ Use all CUDA-aware MPIs with host_data use_device

```c
#pragma acc host_data use_device(A)
MPI_Sendrecv(A+i_start, nx, MPI_DOUBLE, top, 0, A+i_end, nx, MPI_DOUBLE, bottom, 0, ,);
```
Using OpenACC on JUWELS

**PGI** module load PGI/19.3-GCC-8.3.0

**MPI** module load MVAPICH2/2.3.3-GDR

**Tools** module load CUDA/10.1.105

**GCC+OpenACC** module use $PROJECT_cjsc/herten1/modulefiles/module load gcc-openacc/9.2.0

*(Still testing!)*
OpenMP GPU Offloading

- OpenMP very similar to OpenACC
- Compiler support: Clang, XL, GCC, AMD, Cray
- Quality of generated GPU code improving...
- Directives: Target, Teams, Distribute

OpenMP

```c
!$omp target teams distribute parallel do simd
    private(t) map(pi) reduction(+:pi)
    do i = 0, N-1
        t = (i + 0.5) / N
        pi = pi + 4.0 / (1.0 + t*t)
    end do
!$omp end target teams distribute parallel do simd
```

OpenACC

```c
!$acc parallel loop reduction(+:pi)
    do i = 0, N-1
        t = (i + 0.5) / N
        pi = pi + 4.0 / (1.0 + t*t)
    end do
!$acc end parallel
```
Conclusions
Conclusions

- OpenACC: High-level GPU programming model
- Compiler 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
- Interface to other GPU programming models, libraries
- JSC OpenACC course: October 2020

Thank you for your attention!

a.herten@fz-juelich.de
Appendix
Appendix

Glossary
Glossary I

**AMD**  Manufacturer of CPUs and GPUs. 11, 43

**CUDA**  Computing platform for GPUs from NVIDIA. Provides, among others, CUDA C/C++. 13, 38, 39, 40, 41

**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. 11

**MPI**  The Message Passing Interface, a API definition for multi-node computing. 41

**NVIDIA**  US technology company creating GPUs. 6, 11, 48, 49
Glossary II

OpenACC  Directive-based programming, primarily for many-core machines. 2, 6, 7, 8, 9, 10, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 28, 29, 31, 32, 33, 34, 35, 37, 38, 39, 40, 41, 42, 43, 45

OpenMP  Directive-based programming, primarily for multi-threaded machines. 2, 7, 11, 26, 43

PGI  Compiler creators. Formerly The Portland Group, Inc.; since 2013 part of NVIDIA. 12, 38, 39, 40

Volta  GPU architecture from NVIDIA (announced 2017). 12

CPU  Central Processing Unit. 11, 35, 48

GPU  Graphics Processing Unit. 11, 14, 35, 38, 39, 40, 41, 43, 45, 48, 49
References: Images, Graphics