DIRECTIVE-BASED GPU PROGRAMMING WITH OPENACC

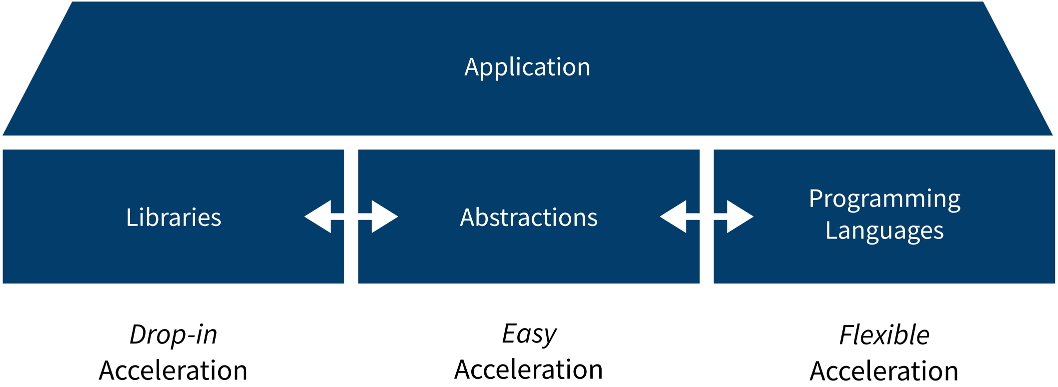
JSC MSA:GPU SEMINAR

4 February 2020 | Andreas Herten | Forschungszentrum Jülich

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Outline

GPU Acceleration Possibilities



GPU Acceleration Possibilities

Drop-in Acceleration

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
- …

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

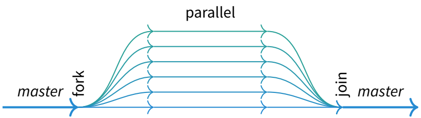
  Master thread launches parallel child threads; merge after execution

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



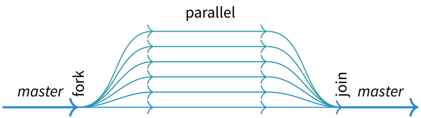
OpenMP

Open{MP↔ACC}

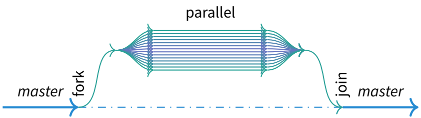
Everything's connected

- 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



OpenMP



OpenACC

JÜLICH SUPERCOMPUTING CENTRE

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. $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 $\rightarrow$ **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
PGI	NVIDIA GPU, CPU	C, C++, Fortran	No	Yes	Best performance
GCC	NVIDIA GPU, *AMD GPU*	C, C++, Fortran	Yes	Yes	AMD support coming up
Cray	NVIDIA GPU	C, C++	No	No	*???*
Clang/LLVM	CPU, *NVIDIA GPU*	C, *C++*	Yes	Yes	Via Clang OpenMP backend; very fresh!

2 Compiler

Flags and options

OpenACC compiler support: activate with compile flag

PGI `pgcc -acc`

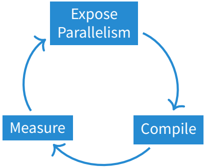
`-ta=tesla|-ta=multicore`	Target GPU or CPU
`-ta=tesla:cc70`	Generate Volta-compatible code
`-ta=tesla:lineinfo`	Add source code correlation into binary
`-ta=tesla:managed`	Use unified memory
`-Minfo=accel`	Print acceleration info

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

3 $uccess

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**

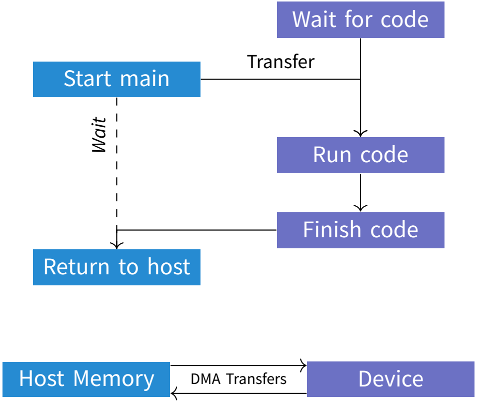
- Because of *generality*: Sometimes not last bit of hardware performance accessible
- But: Use OpenACC together with other accelerator-targeting techniques (CUDA, libraries, …)

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];
    }
}
```

```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
```

OpenACC Directives

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
  $\rightarrow$ 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

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: 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` C

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

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

Compilation Result

PGI

```
$ pgcc  -acc -ta=tesla:cc70 -Minfo=accel -o reduce-parallel.exe reduce-parallel.c
main:
      8, Generating Tesla code
          9, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
      8, Generating implicit copyout(y[:],x[:])
     13, Generating Tesla code
         14, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
             Generating reduction(+:sum)
     13, Generating implicit copy(sum,y[:])
         Generating implicit copyin(x[:])
```

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)
Time(%)   Total Time   Instances     Average   Minimum   Maximum               Name
-------  -----------  ----------  ----------  --------  --------  ------------------
   43.1        37216           1     37216.0     37216     37216   main_13_gpu
   32.6        28160           1     28160.0     28160     28160   main_8_gpu
   24.2        20928           1     20928.0     20928     20928   main_13_gpu__red
CUDA Memory Operation Statistics (nanoseconds)
Time(%)   Total Time   Instances     Average   Minimum   Maximum               Name
-------  -----------  ----------  ----------  --------  --------  ------------------
   58.6      1842336           4    460584.0      1856    616032  [CUDA memcpy DtoH]
   41.3      1298912           2    649456.0    649152    649760  [CUDA memcpy HtoD]
    0.1         1888           1      1888.0      1888      1888  [CUDA memset]
```

Compilation Result

GCC

```
$ gcc  -fopenacc -fopt-info-omp -o reduce-parallel.exe reduce-parallel.c
reduce-parallel.c:8:10: optimized: assigned OpenACC gang vector loop parallelism
reduce-parallel.c:13:10: optimized: assigned OpenACC gang vector loop parallelism
```

Compilation Result

GCC

```
$ 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)
Time(%)   Total Time   Instances     Average   Minimum   Maximum                Name
-------  -----------  ----------  ----------  --------  --------  ------------------
   99.4      5669158           1   5669158.0   5669158   5669158       main$_omp_fn$1
    0.6        36256           1     36256.0     36256     36256       main$_omp_fn$0
CUDA Memory Operation Statistics (nanoseconds)
Time(%)   Total Time   Instances     Average   Minimum   Maximum                Name
-------  -----------  ----------  ----------  --------  --------  ------------------
   54.7      7373929           5   1474785.8      2112   2627491  [CUDA memcpy DtoH]
   45.3      6104103           7    872014.7      1472   1608162  [CUDA memcpy HtoD]
```

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

```c
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 Example

```c
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` 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 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

JÜLICH Forschungszentrum JÜLICH SUPERCOMPUTING CENTRE

Copy Clause

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

OpenACC: `copy` C

```
#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] ...]
```

Data Region Example

```c
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] ...]
```

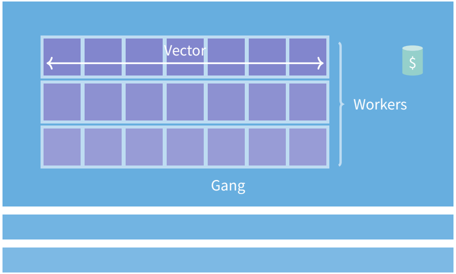
OpenACC Directives

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`

```
#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`

```
#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

OpenACC: `host_data use_device`

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

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

`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 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 Too

- OpenA
- Backen

→ Use all

- Profiler

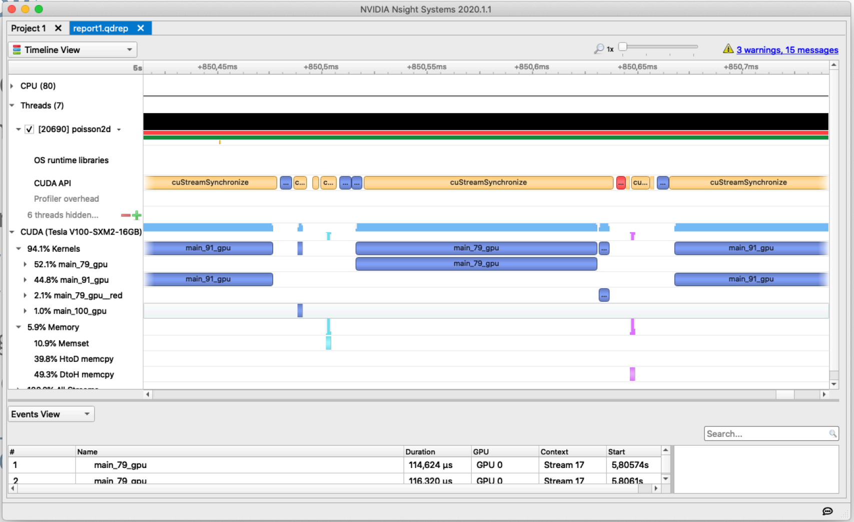
  Legacy

  New

- Debugg
- Also: P

  PGI_

  PGI_A



GPU Tools

- OpenACC only interface
- Backend: CUDA (for GPUs)

```
$ 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

MPI

- OpenACC only interface
- Backend: CUDA (for GPUs)

→ Use all CUDA-aware MPIs with `host_data use_device`

```
#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 GPU Offloading

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

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

OpenMP

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

OpenACC

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

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. 13, 14, 55, 56

CUDA Computing platform for GPUs from NVIDIA. Provides, among others, CUDA C/C++. 16, 49, 50, 51, 52, 53

GCC The GNU Compiler Collection, the collection of open source compilers, among others for C and Fortran. 15

LLVM An open Source compiler infrastructure, providing, among others, Clang for C. 13, 14

MPI The Message Passing Interface, a API definition for multi-node computing. 53

NVIDIA US technology company creating GPUs. 6, 13, 14, 62, 63

Glossary II

OpenACC Directive-based programming, primarily for many-core machines. 2, 6, 7, 8, 9, 10, 11, 12, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 31, 38, 39, 41, 42, 43, 44, 45, 48, 49, 50, 51, 52, 53, 54, 55, 56, 58, 59

OpenMP Directive-based programming, primarily for multi-threaded machines. 2, 7, 8, 9, 13, 14, 34, 35, 36, 55, 56

PGI Compiler creators. Formerly *The Portland Group, Inc.*; since 2013 part of NVIDIA. 15, 49, 50, 51, 52

Volta GPU architecture from NVIDIA (announced 2017). 15

CPU Central Processing Unit. 13, 14, 45, 62

GPU Graphics Processing Unit. 13, 14, 17, 45, 49, 50, 51, 52, 53, 55, 56, 58, 59, 62, 63

References: Images, Graphics

[1] Bill Jelen. *SpaceX Falcon Heavy Launch*. Freely available at Unsplash. URL: `https://unsplash.com/photos/lDEMa5dPcNo`.

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE