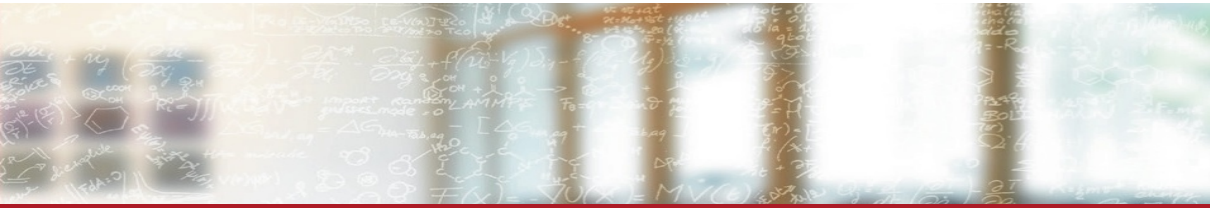
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



Introduction to OpenACC

Summer School 2017 – Effective High Performance Computing

Vasileios Karakasis, CSCS

July 26, 2017

Goals of the course

- Part I
  - Overview of OpenACC
  - Deeper understanding of the concepts through hands-on examples
- Part II
  - Port the miniapp to GPU using OpenACC
  - Walk away ready to start hacking your own code

What is OpenACC?

- Collection of compiler directives for specifying loops and regions to be offloaded from a host CPU to an attached accelerator device
- Host + Accelerator programming model
- High-level representation
- Current specification version: 2.5
  - 2.6 is scheduled soon

When to use OpenACC?

In any of the following cases:

- I program in Fortran
- I need portability across different accelerator vendors
- I don't care about the details, I want my science done
- I want to run on accelerators, but I still need a readable code
- I inhereted a large legacy monolithic codebase, which I don't dare to refactor completely, but I need results faster

OpenACC is not a silver bullet

- User base is still relatively small but expanding
  - You may run into compiler bugs or specification ambiguities
- A high-level representation is not a panacea
  - You need to adapt to the programming model
- Be ware of avoiding a `#pragma`-clutter
  - Rethink and refactor
- Does not substitute hand-tuning

Format of directives

- C/C++
  - `#pragma acc` *directive-name [clause-list] new-line*
  - Scope is the following block of code
- Fortran
  - `!$acc` *directive-name [clause-list] new-line*
  - Scope is until `!$acc end` *directive-name*

Programming model

- Host-directed execution
- Compute intensive regions are offloaded to attached accelerator devices
- Host orchestrates the execution on the device
  - Allocations on the device
  - Data transfers
  - Kernel launches
  - Wait for events
  - Etc. . .

Execution model

- The device executes *parallel* or *kernel regions*
- Parallel region
  - Work-sharing loops
- Kernel region
  - Multiple loops to be executed as multiple kernels
- Levels of parallelim
  1. *Gang*
  2. *Worker*
  3. *Vector*

  - Parallelism levels are decided by the compiler but can be fine-tuned by the user

Execution model

- The device executes *parallel* or *kernel regions*
- Parallel region
  - Work-sharing loops
- Kernel region
  - Multiple loops to be executed as multiple kernels
- Levels of parallelim
  1. *Gang → CUDA block*
  2. *Worker → CUDA warp or second dimension of a block*
  3. *Vector → CUDA threads*

  - Parallelism levels are decided by the compiler but can be fine-tuned by the user
  - Mapping to CUDA blocks/warps/threads is implementation defined

Execution model

Modes of execution

- Gang
  - Gang-redundant (GR)
  - Gang-partioned (GP)
- Worker
  - Worker-single (WS)
  - Worker-partitioned (WP)
- Vector
  - Vector-single (VS)
  - Vector-partitioned (VP)

Execution model

The `kernels` construct

Multiple loops inside kernels construct

```
!$acc kernels
    !GR mode
    do i = 1, N
        !compiler decides on the partitioning (GP/WP/VP modes)
        y(i) = y(i) + a*x(i)
    enddo
    do i = 1, N
        !compiler decides on the partitioning (GP/WP/VP modes)
        y(i) = b*y(i) + a*x(i)
    enddo
!$acc end kernels
```

- Compiler will try to deduce parallelism
- Loops are launched as different GPU kernels

Execution model

The `parallel` construct

Parallel construct

```
!$acc parallel
    do i = 1, N
        ! loop executed in GR mode
        y(i) = y(i) + a*x(i)
    enddo
    !$acc loop
    do i = 1, N
        !compiler decides on the partitioning (GP/WP/VP modes)
        y(i) = b*y(i) + a*x(i)
    enddo
!$acc end parallel
```

- No automatic parallelism deduction → parallel loops must be specified explicitly
- Implicit gang barrier at the end of `parallel`

Execution model

Work-sharing loops

- C/C++: `#pragma acc loop`
  - Applies to the immediately following `for` loop
- Fortran: `!$acc loop`
  - Applies to the immediately following `do` loop
- Loop will be automatically striped and assigned to different threads
  - Use the `independent` clause to force striping
- Convenience syntax combines `parallel`/`kernels` and `loop` constructs
  - `#pragma acc parallel loop`
  - `#pragma acc kernels loop`
  - `!$acc parallel loop`
  - `!$acc kernels loop`

Execution model

Work-sharing loops – the `collapse` clause

Collapse loops

```
!$acc loop collapse(2)
do i = 1,N
    do j = 1,N
        A(i,j) = coeff*B(i,j)
    enddo
enddo
```

- OpenACC vs. OpenMP
  - OpenACC: apply the `loop` directive to the following N loops and possibly collapse their iteration spaces if independent
  - OpenMP: Collapse the iteration spaces of the following N loops

Execution model

Controlling parallelism

- Amount of parallelism at the `kernels` and `parallel` level
  - `num_gangs(...)`, `num_workers(...)`, `vector_length(...)`
- At the `loop` level
  - `gang`, `worker`, `vector`

100 thread blocks with 128 threads each

```
!$acc parallel num_gangs(100), vector_length(128)
    !$acc loop gang, vector
    do i = 1, n
        y(i) = y(i) + a*x(i)
    enddo
!$acc end parallel
```

Execution model

Variable scoping

- Allowed in the `parallel` directive only
- By default, if outside of a code block, variables are shared in global memory
- `private`: A copy of the variable is placed in each *gang* (CUDA block)
- `firstprivate`: Same as `private` but initialized from the host value

Execution model

Variable scoping

- Allowed in the `parallel` directive only
- By default, if outside of a code block, variables are shared in global memory
- `private`: A copy of the variable is placed in each *gang* (CUDA block)
- `firstprivate`: Same as `private` but initialized from the host value

Implicit scoping:

- (C/C++/Fortran) Loop variables are private to the *thread* that executes the loop
- (C/C++ only) Scope of variables declared inside a parallel block depends on the current execution mode:
  - *Vector-partitioned* mode → private to the thread
  - *Worker-partitioned, Vector-single* mode → private to the worker
  - *Worker-single* mode → private to the gang

Execution model

Reduction operations

- `#pragma acc parallel reduction(<op>:<var>)`
  - e.g., `#pragma acc parallel reduction(+:sum)`
- `#pragma acc loop reduction(<op>:<var>)`
- `var` must be scalar
- `var` is copied and default initialized within each gang
- Intermediate results from each gang are combined and made available outside the parallel region
- Complex numbers are also supported
- Operators: `+`, `*`, `max`, `min`, `&`, `|`, `%`, `&&`, `||`

Execution model

Calling functions from parallel regions

- `#pragma acc routine {gang | worker | vector | seq}`
  - Just before the function declaration or definition
- `!$acc routine {gang | worker | vector | seq}`
  - In the specification part of the subroutine
- Parallelism level of the routine
  - `gang`: must be called from GR context
  - `worker`: must be called from WS context
  - `vector`: must be called from VS context
  - `seq`: must be called from sequential context

Memory model

Where is my data?

- The host and the device have separate address spaces
  - Data management between the host and the device is the programmer's responsibility
  - You must make sure that all the necessary data for a computation is available on the accelerator before entering the compute region
  - You must make sure to transfer the processed data back to the host if needed

Memory model

Where is my data?

- The host and the device have separate address spaces
  - Data management between the host and the device is the programmer's responsibility
  - You must make sure that all the necessary data for a computation is available on the accelerator before entering the compute region
  - You must make sure to transfer the processed data back to the host if needed
- But there can be some exceptions:
  - The "device" might be the multicore → no need for data management
  - Some compilers may infer automatically the necessary data transfers
  - Nvidia Pascal GPUs provide efficient support for a unified memory view between the host and the accelerator

Memory model

Directives accepting data clauses

Data clauses may appear in the following directives:

- Compute directives:
  - `#pragma acc kernels`
  - `#pragma acc parallel`
- Data directives:
  - `#pragma acc data`
  - `#pragma acc enter data`
  - `#pragma acc exit data`
  - `#pragma acc declare`
  - `#pragma acc update`

Memory model

Data clauses

- `create(a[0:n])`: Allocate array `a` on device
- `copyin(a[0:n])`: Copy array `a` to device
- `copyout(a[0:n])`: Copy array `a` from device
- `copy(a[0:n])`: Copy array `a` to and from device
- `present(a)`: Inform OpenACC runtime that array `a` is on device
- `delete(a)`: Deallocate array `a` from device (`exit data` only)

Not for the `acc update` directive

Memory model

The `acc data` directive

- Defines a scoped data region
  - Data will be copied in at entry of the region and copied out at exit
  - A *structural reference count* is associated with each memory region that appears in the data clauses

- C/C++: `#pragma acc data` *`[data clauses]`*
  - The next block of code is a data region

- Fortran: `!$acc data` *`[data clauses]`*
  - Defines a data region until `!$acc end data` is encountered

Memory model

The `acc enter/exit data` directives

- Defines an unscoped data region
  - Data will be resident on the device until a corresponding `exit data` directive is found
  - Useful for managing data on the device across compilation units
  - A *dynamic reference count* is associated with each memory region that appears in the data clauses
- C/C++:
  - `#pragma acc enter data [data clauses]`
  - `#pragma acc exit data [data clauses]`
- Fortran:
  - `!$acc enter data [data clauses]`
  - `!$acc exit data [data clauses]`

Memory model

The `acc declare` directive

- Functions, subroutines and programs define *implicit data regions*
- The `acc declare` directive is used in variable declarations for making them available on the device during the lifetime of the implicit data region
- Useful for copying global variables to the device

- C/C++: `#pragma acc declare [data clauses]`
- Fortran: `!$acc declare [data clauses]`

Memory model

The `acc update` directive

- May be used during the lifetime of device data for updating the copies on either host or the device

- `#pragma acc update host(<var-list>)`
  - Update host copy with corresponding data from the device

- `#pragma acc update device(<var-list>)`
  - Update device copy with corresponding data from the host

Memory model

Array ranges

Data clauses may accept as arguments

- Whole arrays
  - C/C++: You *must* specify bounds for dynamically allocated arrays
    - `#pragma acc data copyin(a[0:n])`
    - But `#pragma acc data present(a)` is acceptable: `a`'s bounds can be inferred by the runtime
  - Fortran: array shape information is already embedded in the data type
    - `!$acc data copyin(a)`
- Array subranges
  - `#pragma acc data copyin(a[2:n-2])`

Advanced topics

Synchronization primitives

- Atomic operations
  - `#pragma acc atomic` [atomic-clause]
  - `!$acc atomic` [atomic-clause]
  - Atomic clauses: `read`, `write`, `update` and `capture`
  - Example of “capturing” a value:
    - `v = x++;`
- No global barriers → cannot be implemented due to hardware restrictions
- No equivalent of `__syncthreads()`

Advanced topics

Activity queues

- Activity queues are the equivalent of CUDA event queues or streams
- Data copies and kernels are launched *synchronously* inside the activity queues
- Additional clauses in compute or data directives control the activity queues:
  - `async(<qno>)`: push operations to activity queue `qno` and continue execution on the host
  - `wait(<qno>)`: wait for pending operations in activity queue `qno` to finish before launching next operation on the device
- `#pragma acc wait(<qno>)`: Wait for all events in activity queue `qno` to finish before continuing execution on the host

Advanced topics

Activity queues example

Launch multiple kernels asynchronously on the GPU

```
// Launch kernel on GPU and continue on CPU
#pragma acc parallel loop async(1) present(a)
for(i = 0; i < N; ++i) {
  a[i] = // ... compute on GPU
}
// Launch another kernel on GPU and continue on CPU
#pragma acc parallel loop async(2) present(b)
for(j = 0; j < N; ++j) {
  b[j] = // ... compute on GPU
}
// Wait for all kernels to finish
#pragma acc wait
```

- Especially useful for overlapping data transfers and execution

Advanced topics

Unified memory

- Virtual address space shared between CPU and GPU
- The CUDA driver and the hardware take care of the page migration
- Introduced with the Kepler architecture and CUDA 6, but is significantly improved with Pascal

Advanced topics

Unified memory

- Virtual address space shared between CPU and GPU
- The CUDA driver and the hardware take care of the page migration
- Introduced with the Kepler architecture and CUDA 6, but is significantly improved with Pascal

- You could completely omit the data management in OpenACC !
- Currently, supported by the PGI compiler using the `-ta=tesla:unified` option

Advanced topics

Deep copy

Deep copy example

```
struct foo {
    int *array;
    size_t len;
};
foo a[10];
for (int i = 0; i < 10; ++i) {
    a.len = 100;
    a.array = new int[a.len];
}
#pragma acc enter data copyin(a[0:10])
```

- What will be copied over to the device?

Advanced topics

Deep copy

Deep copy example

```
struct foo {
    int *array;
    size_t len;
};
foo a[10];
for (int i = 0; i < 10; ++i) {
    a.len = 100;
    a.array = new int[a.len];
}
#pragma acc enter data copyin(a[0:10])
```

- What will be copied over to the device? → just `a` with dangling `array` pointers :-(

Advanced topics

Deep copy

Deep copy example

```
struct foo {
    int *array;
    size_t len;
};
foo a[10];
for (int i = 0; i < 10; ++i) {
    a.len = 100;
    a.array = new int[a.len];
}
#pragma acc enter data copyin(a[0:10])
```

- What will be copied over to the device? → just `a` with dangling `array` pointers :-(
- What you would like to be copied? → everything, you must wait for OpenACC 3.0
  - Cray compiler supports deep copy of derived types in Fortran only
  - PGI compiler introduced support for manual deep copy

Combining it all

Data movement/Activity queues/Parallel loops

```
// prepare array a on host
#pragma acc enter data async(1) copyin(a[0:N])
// prepare array b on host
#pragma acc enter data async(2) copyin(b[0:N])
#pragma acc parallel loop async(1) present(a[0:N])
for (i = 0; i < N; ++i)
    foo(a[i])

#pragma acc exit data copyout(a[0:N]) async(1)
#pragma acc parallel loop async(2) present(b[0:N])
for (i = 0; i < N; ++i)
    bar(b[i])
#pragma acc exit data copyout(b[0:N]) async(2)
// some more stuff on the host and then wait for all streams to finish
#pragma acc wait
```

Profiling

- NVIDIA tools (`nvprof`, `nvpp`)
  - `$ nvprof <openacc-executable>`
- CrayPAT
  - `$ module load daint-gpu`
  - `$ module load perftools-cscs/645openacc`
  - Recompile and run
  - Report in `.rpt` file

Hands-on exercises

General information

- `grep TODO *.{cpp,f90,f03}`
- Both Cray/PGI compilers are supported, unless otherwise stated
- `source <ssprefix>/scripts/setup.sh` → will make available PGI 17.4
- `module load craype-accel-nvidia60`
- `make` or `make VERBOSE=1` to get compiler information about offloaded regions

Hands-on

The basics

- Vector scale:
  - `exercises/openacc/shared/axpy_openacc.{cpp,f90}`
  - Run as:
    `srun --reserv=summer -Cgpu ./axpy.openacc [ARRAY_SIZE]`
  - `ARRAY_SIZE` is power of 2, default is 16

- Dot product:
  - `exercises/openacc/shared/dot_openacc.{cpp,f90}`

Data management

- Moving data to and from the device is slow (≈7–8 GB/s per direction)
- Avoid unnecessary data movement
  - Move needed data to GPU early enough and keep it there as long as possible
  - Update host copies using `#pragma acc update` directive if needed

Hands-on

Blur kernel

Naive implementation

```
for (auto istep = 0; istep < nsteps; ++istep) {
    int i;

    #pragma acc parallel loop copyin(in[0:n]) copyout(buffer[0:n])
    for(i=1; i<n-1; ++i) {
        buffer[i] = blur(i, in);
    }
    #pragma acc parallel loop copyin(buffer[0:n]) copy(out[0:n])
    for(i=2; i<n-2; ++i) {
        out[i] = blur(i, buffer);
    }

    std::swap(in, out);
}
```

Interoperability with MPI and CUDA

1. Call an optimised library function that expects data on the device, e.g., cuBLAS
2. Let optimised MPI implementations do RDMA between remote devices' memory
3. Manual data management with CUDA, but parallelisation with OpenACC
   - The safest way to manipulate pointers on the device

Interoperability with MPI and CUDA

1. Call an optimised library function that expects data on the device, e.g., cuBLAS
2. Let optimised MPI implementations do RDMA between remote devices' memory
3. Manual data management with CUDA, but parallelisation with OpenACC
   - The safest way to manipulate pointers on the device

Scenarios (1) and (2)

```
#pragma acc host_data use_device(<varlist>)
```

Interoperability with MPI and CUDA

1. Call an optimised library function that expects data on the device, e.g., cuBLAS
2. Let optimised MPI implementations do RDMA between remote devices' memory
3. Manual data management with CUDA, but parallelisation with OpenACC
   - The safest way to manipulate pointers on the device

Scenarios (1) and (2)

```
#pragma acc host_data use_device(<varlist>)
```

Scenario (3)

Use the `deviceptr(<ptrlist>)` clause with `parallel`, `kernels` and `data`

Hands-on

2D diffusion example

Source code:

- `diffusion2d_omp.{cpp,f90}`: our baseline code
  - Single node OpenMP version for the CPU
- `diffusion2d_openacc.{cpp,f90}`
  - Single node OpenACC version
- `diffusion2d_openacc_mpi.{cpp,f90}`
  - MPI+OpenACC version
  - If `OPENACC_DATA` is undefined, data management is performed by CUDA

Hands-on

Calling cuBLAS methods

Source code:

- `topics/openacc/practicals/gemm/gemm.cpp`

Steps:

1. Compile with '`make CPPFLAGS=`' to get also the naive implementation → too slow!
2. Offload the GEMM method to the GPU using OpenACC
3. Make use of cuBLAS GEMM through OpenACC

Outlook

OpenACC 2.6 is due end of the year

- Manual deep copy
- Standardize behavior of Fortran optional arguments
- Fortran bindings for all API routines
- `acc serial` directive
- Device query routines
- Improvements in error handling

OpenACC 3.0

- Not scheduled yet
- The big feature should be the true deep copy

OpenACC vs. OpenMP

- OpenMP 4.0 introduced directives for offloading computation to accelerators
- Similar concepts to OpenACC but OpenMP is a more prescriptive standard
- There is no OpenMP-OpenACC merger envisioned right now

OpenACC and compiler support

- PGI
  - Latest spec support; drives the OpenACC development
  - Twice per year a community release
- Cray
  - Support up to OpenACC 2.0; no new features or later spec support
  - Bug fixes and support for the current implementation only
- GCC
  - Support of OpenACC 2.0a from GCC 5.1 onward
  - Support of OpenACC 2.5 perhaps in GCC 8.0

More information and events

- `http://www.openacc.org`
  - Specification and related documents
  - Tutorials
  - Events

- OpenACC Hackathons
  - One week of intensive development for porting your code to the GPUs
  - 3 developers + 2 mentors per team
  - 3× in USA + 2× in Europe in 2017
  - Find the one that fits you and apply!

Porting the miniapp to GPUs using OpenACC

General info

- Fortran 90 version
  - `miniapp/openacc/fortran/`
- C++11 version
  - `miniapp/openacc/cxx/`
  - Compile with PGI 17.4
    - `source <ss-prefix>/scripts/setup.sh`
    - `module switch pgi/16.9.0 pgi/17.4`
- Interesting files
  - `main.{cpp,f90}`: the solver
  - `data.{h,f90}`: domain types
  - `linalg.{cpp,f90}`: linear algebra kernels
  - `operators.{cpp,f90}`: the diffusion kernel

Notes for the C++ version

- There are two C++-isms that complicate things:
  1. Domain data is encapsulated inside the `Field` class
     - Allocated and initialised inside the constructor
     - Deallocated inside the destructor
  2. Operators for accessing the domain data

- OpenACC provides the `enter data` and `exit data` directives for unscoped data management
- Operators are just another kind of functions
  - `acc routine` directive is just for that
- Remember to copy the object itself (`this` pointer)