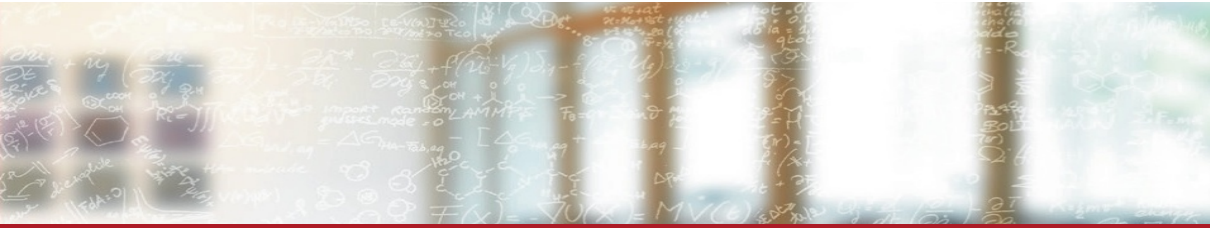
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



Introduction to OpenACC

Summer School 2016 – Effective High Performance Computing

Vasileios Karakasis, CSCS

July 27, 2016

Goals of this course

- Part I
  - Quick 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
  - 3.0 is scheduled this fall

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*
  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 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

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

Execution model

Synchronization & Activity queues

- Atomics
  - `#pragma acc atomic` [atomic-clause]
  - `!$acc atomic` [atomic-clause]
  - Atomic clauses: `read`, `write`, `update` and `capture`
  - Example of “capturing” a value:
    - `v = x++;`
- Activity queues (CUDA event queues)
  - Data copies and kernels are launched synchronously inside the activity queues
  - `async` clause → pushes operations to an activity queue and host continues execution
  - `wait` clause → wait for *pending* operations to finish in an activity queue
  - `#pragma acc wait`
- No `__syncthreads()`

Execution model

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

Memory model

Where is my data?

- `#pragma acc data` *[data-clause]*
  - Scope and lifetime is the immediately following block of code
- `#pragma acc enter data` *[data-clause]*
- `#pragma acc exit data` *[data-clause]*
- Common clauses:
  - `create(a)`: Allocate array `a` on device (`data` and `enter data` only)
  - `copyin(a)`: Copy array `a` to device (`data` and `enter data` only)
  - `copyout(a)`: Copy array `a` from device (`data` and `exit data` only)
  - `copy(a)`: Copy array `a` to and from device (`data` only)
  - `present(a)`: Inform OpenACC runtime that array `a` is on device (`data` only)
  - `delete(a)`: Deallocate array `a` from device (`exit data` only)
  - `wait`, `async`: `enter data` and `exit data` only

Memory model

Array ranges and shared memory

- Whole arrays
  - C/C++: You *must* specify bounds for dynamically allocated arrays
    - `#pragma acc data copyin(a[0:n])`
  - 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])`
- Hint that a subarray should reside in the shared memory of the device
  - `#pragma acc cache(<varlist>)`

Memory model

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?

Memory model

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 :-(

Memory model

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

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 perftools-cscs/630openacc`
  - Recompile and run
  - Report in `.rpt` file

Hands-on

axpy

- `exercises/openacc/axpy/axpy_openacc.{cpp,f90}`
- `grep TODO *.{cpp,f90,f03}`
- `module load craype-accel-nvidia35`
- `module switch cce/8.3.12 cce/8.4.6`
- `module switch pgi/15.3.0 pgi/15.9.0`
- `make VERBOSE=1 PGI=1` or `CRAY=1`

Hands-on

Reduction

- `#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 and/or thread
- Intermediate results from each gang are combined and made available outside the parallel region
- `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

The diffusion kernel

- `exercises/openacc/diffusion/diffusion_omp.cpp`
- `exercises/openacc/diffusion/diffusion_openacc.{cpp,f90}`
- `exercises/openacc/diffusion/diffusion_openacc_mpi.{cpp,f90}`
- `-DOPENACC_DATA` → data management by OpenACC

Future prospects

- OpenACC 3.0 is coming
  - Deep copy of data structures
  - Better semantics for the `reduction` clause
  - Math function intrinsics
  - Several other smaller scale improvements
- OpenACC vs. OpenMP 4.0 and 4.5
  - There is no merger envisioned right now
  - PGI and NVIDIA are actively supporting OpenACC
  - Cray abandons OpenACC development, but will provide support up to OpenACC 2.5
  - GCC 5 supports OpenACC 2.0a
  - Pathscale supports OpenACC 2.0
  - Intel and OpenACC? – No such thoughts

More information and events

- `http://www.openacc.org`
- OpenACC Hackathons
  - One week of intensive development for porting your code to the GPUs
  - 5 developers + 2 mentors per team
  - 2× in USA + 2× in Europe per year
  - 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 15.9
- 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)