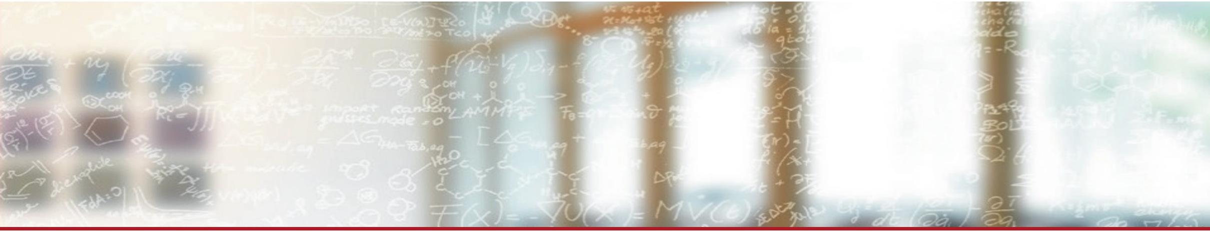
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



GPU programming using OpenACC

CSCS-USI Summer School 2021

Andreas Jocksch, Vasileios Karakasis, CSCS

July 19–30, 2021

Goals of the course

- Part I
  - Basic concepts
    - Execution and memory model
    - Basic directives
    - Hands-on sessions
  - Advanced topics
    - Asynchronous execution and wait queues
    - Interoperability with CUDA and MPI
    - Deep copy
    - Hands-on sessions
- Part II
  - Port the miniapp to GPU using OpenACC
  - Walk away ready to start hacking your own code

Introduction to OpenACC

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: 3.1
- Similarities to classic OpenMP for multicores

Why to use OpenACC?

Because . . .

- I don’t care about all the little hardware details, I want my science done.
- I want to run on accelerators, but I still need a fast and readable code.
- I need portability across different accelerator vendors, but also to be able to run on the multicore.
- I inhereted a large legacy monolithic codebase, which I don’t dare to refactor completely, but I need to get my results faster.
- My code is in Fortran.

OpenACC is not a silver bullet

- A high-level representation is not a panacea.
  - You still need to understand and adapt to the programming model.

- Does not substitute hand-tuning, but can serve as a very good starting point.
  - Some low-level CUDA constructs are not exposed (shared memory, groups etc.)

- User base not yet as large as of classic OpenMP for multicores, but it is expanding.
  - You may run into compiler bugs or specification ambiguities.

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*

Format of directives

OpenMP looks like this ...

```
#pragma omp parallel for
for (auto i = 0; i < m; ++i) {
    // work on A[p] array
}
```

.. while OpenACC looks like this ...

```
#pragma acc parallel loop
for (auto i = 0; i < m; ++i) {
    // work on A[p] array
}
```

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 parallelism
  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 parallelism
  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

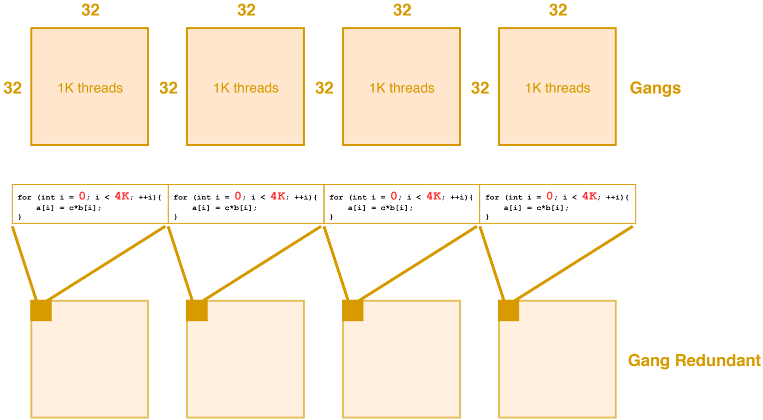
Modes of execution

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

Execution model

Modes of execution

```
for (int i = 0; i < 4096; ++i) {
    a[i] = c*b[i];
}
```



```
for (int i = 0; i < 4K; ++i){
    a[i] = c*b[i];
}
```

```
for (int i = 0; i < 4K; ++i){
    a[i] = c*b[i];
}
```

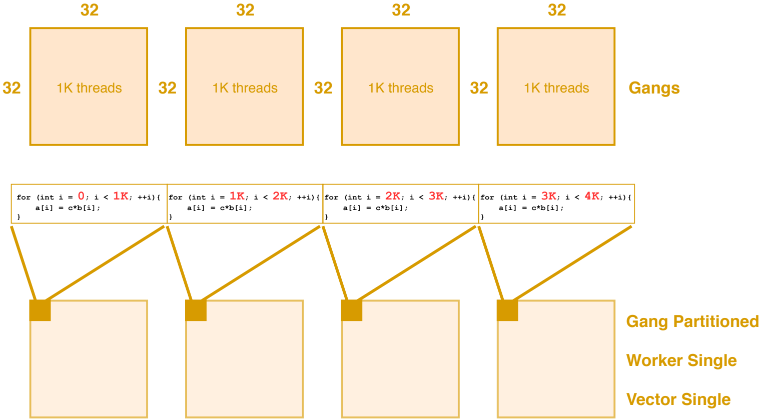
```
for (int i = 0; i < 4K; ++i){
    a[i] = c*b[i];
}
```

```
for (int i = 0; i < 4K; ++i){
    a[i] = c*b[i];
}
```

Execution model

Modes of execution

```
for (int i = 0; i < 4096; ++i) {
    a[i] = c*b[i];
}
```



```
for (int i = 0; i < 1K; ++i){
    a[i] = c*b[i];
}
```

```
for (int i = 1K; i < 2K; ++i){
    a[i] = c*b[i];
}
```

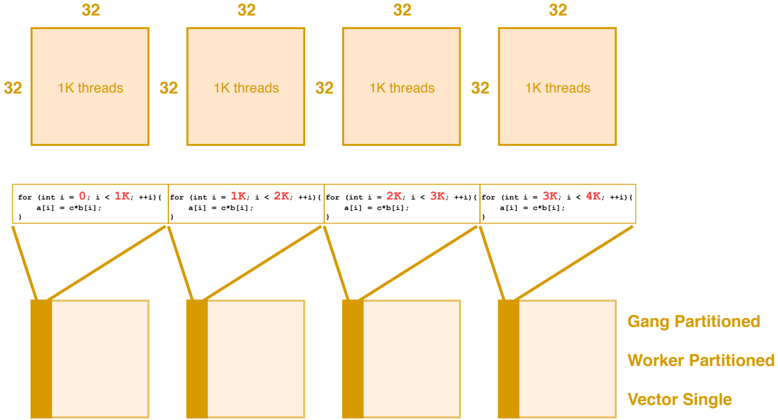
```
for (int i = 2K; i < 3K; ++i){
    a[i] = c*b[i];
}
```

```
for (int i = 3K; i < 4K; ++i){
    a[i] = c*b[i];
}
```

Execution model

Modes of execution

```
for (int i = 0; i < 4096; ++i) {
    a[i] = c*b[i];
}
```



```
for (int i = 0; i < 1K; ++i){
    a[i] = c*b[i];
}
```

```
for (int i = 1K; i < 2K; ++i){
    a[i] = c*b[i];
}
```

```
for (int i = 2K; i < 3K; ++i){
    a[i] = c*b[i];
}
```

```
for (int i = 3K; i < 4K; ++i){
    a[i] = c*b[i];
}
```

Gang Partitioned

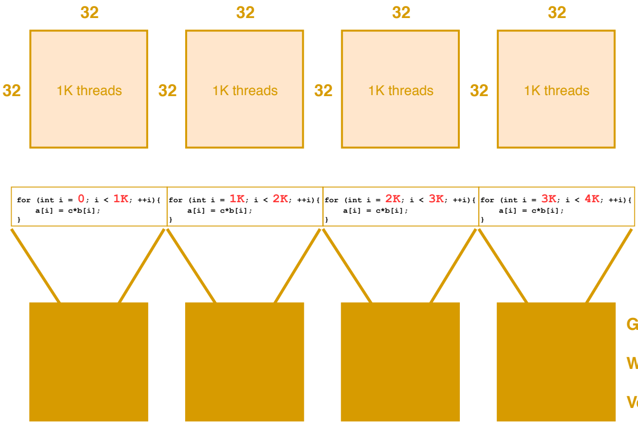
Worker Partitioned

Vector Single

Execution model

Modes of execution

```
for (int i = 0; i < 4096; ++i) {
    a[i] = c*b[i];
}
```



```
for (int i = 0; i < 1K; ++i){
    a[i] = c*b[i];
}
```

```
for (int i = 1K; i < 2K; ++i){
    a[i] = c*b[i];
}
```

```
for (int i = 2K; i < 3K; ++i){
    a[i] = c*b[i];
}
```

```
for (int i = 3K; i < 4K; ++i){
    a[i] = c*b[i];
}
```

Gang Partitioned

Worker Partitioned

Vector Partitioned

Execution model

The `kernels` construct

Multiple loops inside kernels construct

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

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

Execution model

The parallel construct

Parallel construct

```
!$acc parallel
    y(1) = y(1) + x(1)  !GR model
!$acc end parallel
!$acc parallel
    do i = 1, N  !loop executed in GR mode
        y(i) = y(i) + a*x(i)
    enddo
!$acc end parallel
!$acc parallel
    !$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

```fortran
!$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, worker, 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
- `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
- `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) Scalar variables are firstprivate to the *thread* that executes the loop, array variables are shared in global memory (different to OpenMP!)
- (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
  - Internally the address spaces are linked with a so-called present table

Memory model

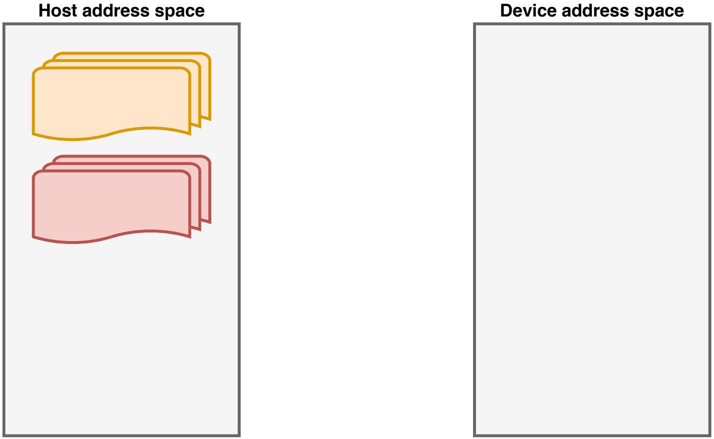
Where is my data?

- 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
  - Internally the address spaces are linked with a so-called present table

- 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

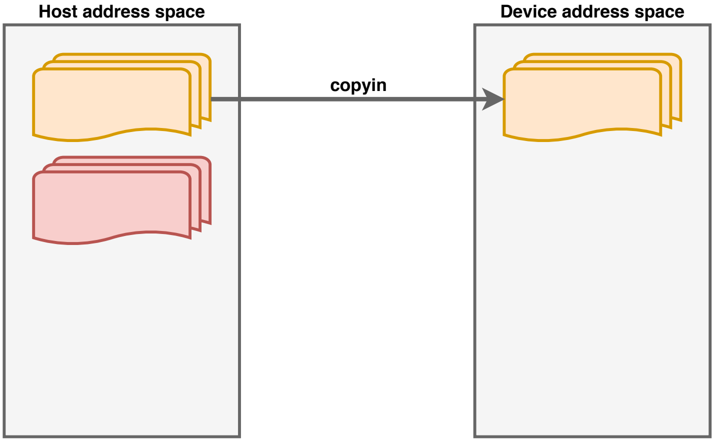
Memory model

Separate address spaces



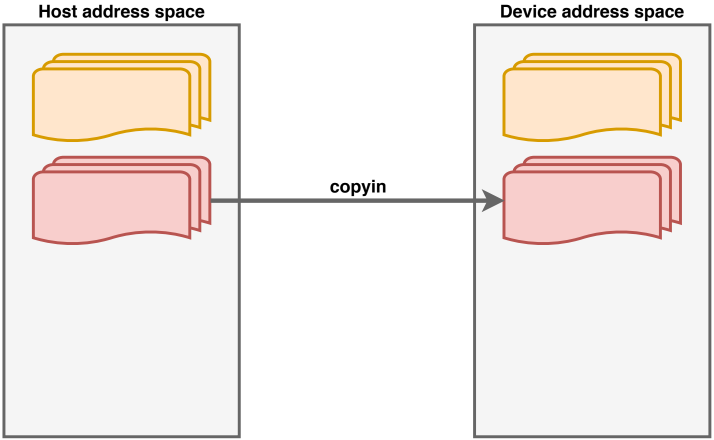
Memory model

Separate address spaces



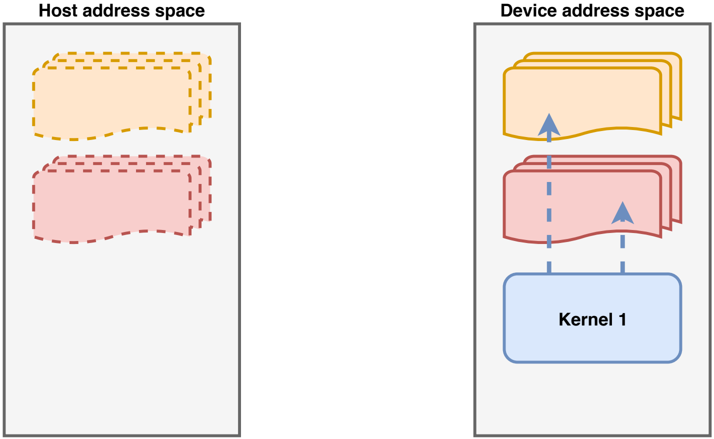
Memory model

Separate address spaces



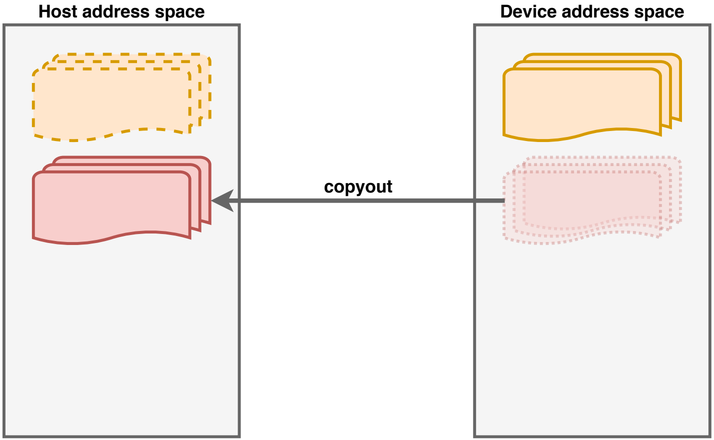
Memory model

Separate address spaces



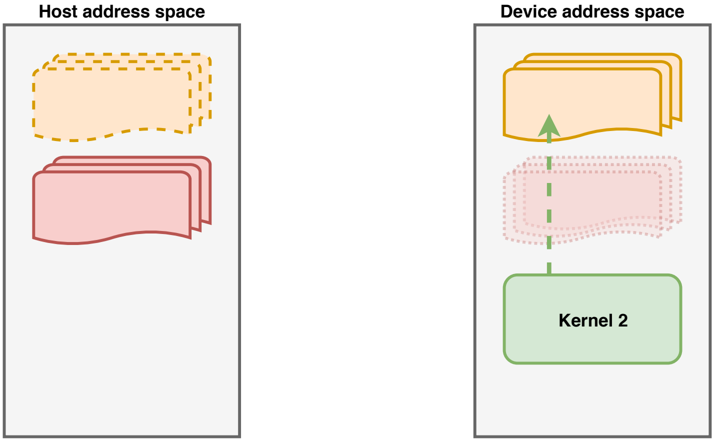
Memory model

Separate address spaces



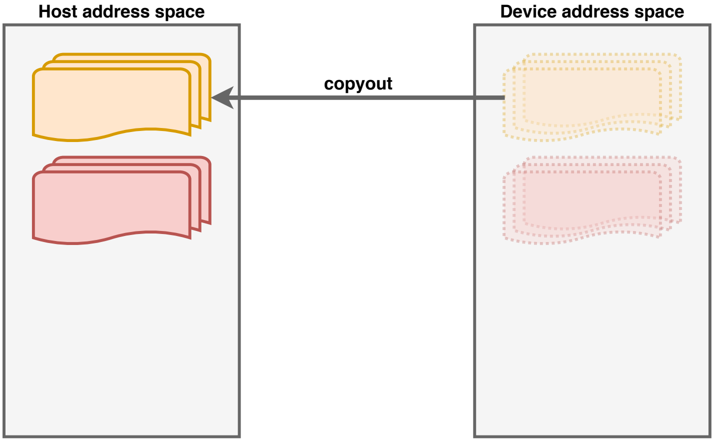
Memory model

Separate address spaces



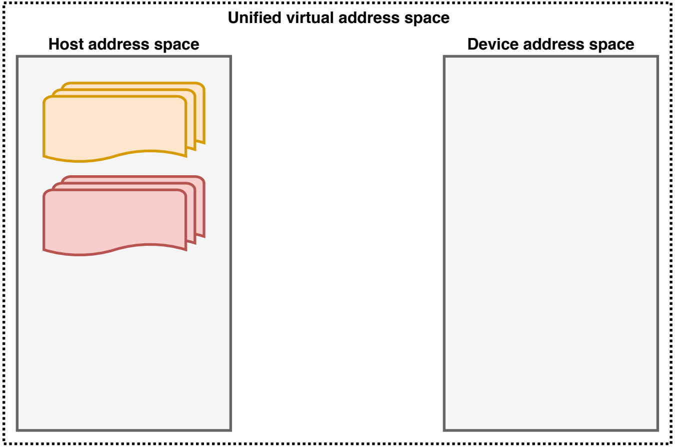
Memory model

Separate address spaces



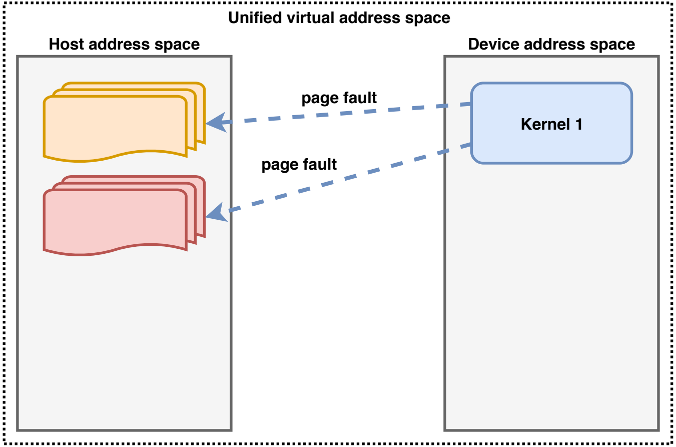
Memory model

Unified memory address space



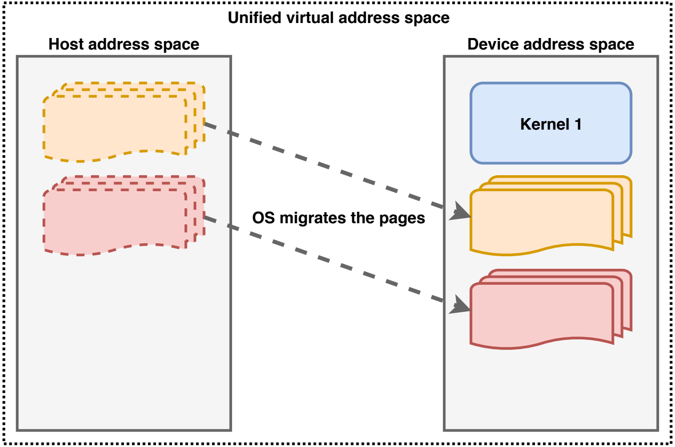
Memory model

Unified memory address space



Memory model

Unified memory address space



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]*

- The variables are not initalized

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
    - `!$acc data copyin(a)`
- Array subranges
  - C/C++: start and length
    - `#pragma acc data copyin(a[2:n-2])`
  - Fortran: first index and last index
    - `!$acc data copyin(a[3:n])`

Synchronization directives

- 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()`

Leverage the 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

Leverage the 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 !
- Supported by the PGI compiler using the `-ta=tesla:managed` option

Hands-on exercises

General information

- Base directory for the OpenACC exercises is `topics/openacc`:
- `practicals/`: The hands-on exercises
- `solutions/`: Where the solutions will appear
- `ci/`: Continuous integration tests for the exercises (ask me offline if interested)

Hands-on exercises

General information

- `grep TODO *.{cpp,f90,f03}`
- PGI compiler is recommended, GNU and Cray compiler are present in `make` files
- `module load craype-accel-nvidia60` for loading CUDA and set the target architecture to the GPU
- `make`

Hands-on

Exercise 1 – AXPY

- `practicals/axpy/axpy_openacc.{cpp,f90}`
- Run as:

  ```
  srun --reserv=summer_school -Cgpu ./axpy.openacc [ARRAY_SIZE]
  ```

  - `ARRAY_SIZE` is power of 2, default is 16
- Try with different sizes. Does the GPU outperform the CPU version?

Hands-on

Exercise 2 – Dot product

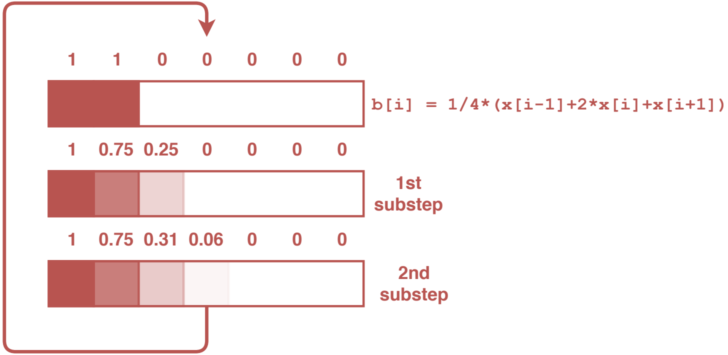
- `practicals/basics/dot_openacc.{cpp,f90}`
- Run as:

  ```
  srun --reserv=summer_school -Cgpu ./dot.openacc [ARRAY_SIZE]
  ```

  - `ARRAY_SIZE` is power of 2, default is 2
- Try with different sizes. Does the GPU outperform the CPU version?

Hands-on

Exercise 3 – 1D blur kernel



Hands-on

Exercise 3 – 1D blur kernel

- `practicals/basics/blur_openacc.{cpp,f90}`
- Run as:

```
srun --reserv=summer_school -Cgpu ./blur.openacc [ARRAY_SIZE]
```

  - `ARRAY_SIZE` is multiple of 2, default is 20
- Offload to GPU the loops of the naive kernel; why is it so slow?

Hands-on

Exercise 3 – 1D blur kernel

- `practicals/basics/blur_openacc.{cpp,f90}`
- Run as:

  ```
  srun --reserv=summer_school -Cgpu ./blur.openacc [ARRAY_SIZE]
  ```

  - `ARRAY_SIZE` is multiple of 2, default is 20
- Offload to GPU the loops of the naive kernel; why is it so slow?
- Moving data to and from the device is slow (≈7–8 GB/s per direction)
- Avoid unnecessary data movement in the `nocopies` kernel
  - Move the necessary 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

Exercise 4 – Experiment with the unified memory

- Remove all the data directives and data clauses
- Compile the `blur_twice_naive` kernel with `-ta=tesla:managed`
- How does it compare to the manual data management in terms of performance?
- Can you explain the performance difference?

Asynchronous execution and wait queues

By default, all OpenACC directives are blocking.

- The calling CPU thread must wait for the OpenACC operation (data transfer, kernel etc.) to complete
- All OpenACC operations are enqueued in a single *activity queue* (CUDA stream)
- All items in an activity queue are executed synchronously, but activity queues are independent from each other

Asynchronous execution and wait queues

OpenACC allows you to enqueue operations on different activity queues using the `async` clause and wait for them using the `wait` directive/clause.

- `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
  - Wait for all queues to finish if used without an argument

Asynchronous execution and wait queues

Example of operations pipelining

Operations are executed sequentially

```
#pragma acc data ...
for (auto p = 0; p < n; ++p) {
    #pragma acc update device(A[p][0:m])
    #pragma acc parallel loop
    for (auto i = 0; i < m; ++i) {
        // work on A[p] array
    }

    #pragma acc update host(A[p][0:m])
}
```

Asynchronous execution and wait queues

Example of operations pipelining

Operations are pipelined

```
#pragma acc data ...
for (auto p = 0; p < n; ++p) {
    #pragma acc update device(A[p][0:m]) async(p)
    #pragma acc parallel loop async(p)
    for (auto i = 0; i < m; ++i) {
        // work on A[p] array
    }

    #pragma acc update host(A[p][0:m]) async(p)
}
#pragma acc wait
```

This concept is useful for overlapping computation and data transfers to the device.

Interoperability with CUDA

- Can I use a CUDA pointer inside OpenACC device context?
- Can I call a CUDA function from OpenACC host context?

Short answer is *yes*.

Interoperability with CUDA

Use CUDA pointers inside OpenACC device context

A scenario:

- Have a CUDA code that needs to call a function that uses OpenACC.
- This function may accept an array that has been allocated already on the GPU by CUDA.

The problem?

- OpenACC only knows of pointers that it is managing itself; the `present` clause won't work. No idea what this pointer is; never seen it before!

Interoperability with CUDA

Use CUDA pointers inside OpenACC device context

Solution:

- We need to instruct the OpenACC runtime to trust this pointer and that it is a valid device pointer.
- OpenACC runtime will just treat that pointer as known, but it won't check its shape.
- Use the `deviceptr(<ptrlist>)` clause with `parallel`, `kernels` and `data` directives

Interoperability with CUDA

Use CUDA pointers inside OpenACC device context – Example

```
void copy(double *dst, const double *src, size_t n) {
  #pragma acc parallel loop deviceptr(dst, src)
  for (size_t i = 0; i < n; ++i) {
    dst[i] = src[i];
  }
}

int main() {
  double *a, *b;
  cudaMalloc(&a, 1024);
  cudaMalloc(&b, 1024);
  ...
  copy(b, a, 1024);
  return 0;
}
```

Interoperability with CUDA

Register CUDA pointers in the present table with runtime library routine

Another scenario:

- You want to call OpenACC code without `deviceptr(<ptrlist>)` clause, e.g., from OpenACC and CUDA.

The Solution:

- You register the CUDA pointer with `acc_map_data` in the present table.

```
int main() {
  double *a_host, *a_device;
  a_host = new double[1024];
  cudaMalloc(&a_device, 1024);
  acc_map_data(a_host, a_device, sizeof(double)*1024);
  ...
  acc_unmap_data(a_host);
  ...
  return 0;
}
```

Interoperability with CUDA

Call a CUDA function from OpenACC host context

Scenario:

- My code is in OpenACC, but I need to call an optimized library written in CUDA, which accepts device pointers, e.g., cuBLAS.

Interoperability with CUDA

Call a CUDA function from OpenACC host context

Scenario:

- My code is in OpenACC, but I need to call an optimized library written in CUDA, which accepts device pointers, e.g., cuBLAS.

Problem:

- I only “see” device pointers while in a parallel region, but I want to get a device pointer, while executing on the host.

Interoperability with CUDA

Call a CUDA function from OpenACC host context

Scenario:

- My code is in OpenACC, but I need to call an optimized library written in CUDA, which accepts device pointers, e.g., cuBLAS.

Problem:

- I only “see” device pointers while in a parallel region, but I want to get a device pointer, while executing on the host.

Solution:

- Use a `host_data` region
  - `#pragma acc host_data use_device(<varlist>)`

Interoperability with CUDA

The host_data directive

- C/C++: `#pragma acc host_data use_device(<varlist>)`
  - In the next block of code the compiler will make available the device address of any variable in `<varlist>`.
- Fortran: `!$acc host_data use_device(<varlist>)`
  - The compiler will make available the device address of any variable in `<varlist>` until a matching `!$acc end host_data` is found.
- Optional clauses:
  - `if(`*condition*`)`: Use the device pointer if *condition* is true.
  - `if_present`: Use the device pointer if variables in `<varlist>` are present on the device.

Heads-up: Remember this directive if you want to combine OpenACC and MPI.

Hands-on

Exercise 5 – Calling cuBLAS methods

Source code:

- `practicals/gemm/gemm.cpp`
- Run as:

```
srun --reserv=summer_school -Cgpu ./axpy.openacc [ARRAY_SIZE]
```

  - `ARRAY_SIZE` is power of 2, default is 8

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
4. Compare the performance of the different versions

Hands-on

Exercise 6.1 – 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
  - Run as:

```
srun --reserv=summer_school -Cgpu ./diffusion2d.openacc [ARRAY_SIZ
```

    - `ARRAY_SIZE` is power of 2, default is 16
  - Fill in the parts where `OPENACC_DATA` is defined.

Hands-on

Exercise 6.2 – 2D diffusion example using CUDA data management

Source code:

- `diffusion2d_openacc.{cpp,f90}`
  - Single node OpenACC version
  - Run as:
    `srun --reserv=summer_school -Cgpu ./diffusion2d.openacc.cuda [ARRA`
    - `ARRAY_SIZE` is power of 2, default is 16
  - Fill in the parts where `OPENACC_DATA` is undefined.

Deep Copy

The concept

True deep copy (ideal)

```
struct foo {
    int *arr;
    size_t len;
};
// ...
for (auto i = 0; i < 3; ++i) {
    f[i].len = 10;
    f[i].arr = new int[f[i].len];
}

#pragma acc enter data copyin(f[0:3])
```

- Where will `f[i].arr` refer to?

Deep Copy

The concept

True deep copy (ideal)

```
struct foo {
    int *arr;
    size_t len;
};
// ...
for (auto i = 0; i < 3; ++i) {
    f[i].len = 10;
    f[i].arr = new int[f[i].len];
}

#pragma acc enter data copyin(f[0:3])
```

- Where will `f[i].arr` refer to? → They will be host pointers!
- Ideally, we would like everything to be magically copied.
  - Not so easy, especially for C/C++.

Deep Copy

The manual solution – OpenACC 2.6

Manual deep copy (top-down approach)

```
#pragma acc enter data copyin(f[0:3])
for (auto i = 0; i < 3; ++i) {
    #pragma acc enter data copyin(f[i].arr[0:f[i].len])
}
// do stuff on the device
for (auto i = 0; i < 3; ++i) {
    #pragma acc exit data copyout(f[i].arr[0:f[i].len])
}
#pragma acc exit data copyout(f[0:3])
```

- The runtime will attach the `f[i].arr` pointer to the device copy of the data.
- This happens implicitly if the `f[i].arr` pointer is present on the device.

Deep Copy

The manual solution – OpenACC 2.6

Manual deep copy (bottom-up approach)

```
for (auto i = 0; i < 3; ++i) {
    #pragma acc enter data copyin(f[i].arr[0:f[i].len])
}

#pragma acc enter data copyin(f[0:3])
for (auto i = 0; i < 3; ++i) {
    acc_attach((void **) &f[i].arr);
}
// do stuff on the device
```

- At the time when `f[i].arr` is copied to the device, the pointer is not already present on the device.
- If we copy the `struct` later, we need to manually attach the pointer to the device copy of the data.

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
- Compiler support for GPU targets
  - Cray
  - IBM XL
  - GCC (needs to be compiled specially)
  - Clang (under development)

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
  - From CCE 9.0 onward support will be dropped
- GCC
  - Support of OpenACC 2.0a from GCC 5.1 onward
  - Support of OpenACC 2.5 in development branch

More information and events

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

- GPU Hackathons
  - One week+ of intensive development for porting your code to the GPUs
  - 3 developers + 2 mentors per team
  - Several virtual events scheduled for 2021
  - Find the one that fits you and apply!

Porting the miniapp to GPUs using OpenACC

General information

- C++ only
- OpenACC version resides in `miniapp/openacc`
- MPI+OpenACC version resides in `miniapp/mpi/openacc`
- Plotting script inside the `scripts/` folder
  - Use `plot.sh` in order not to mess with your environment.
- Interesting files
  - `main.cpp`: the solver
  - `data.h`: domain types
  - `linalg.cpp`: linear algebra kernels
  - `operators.cpp`: the diffusion kernel + communication routines

Hints

Differences to CUDA version

- No need for a separate `kernels` namespace
- No need for keeping a a copy of the device pointer
  - OpenACC runtime does just that for you
- No need for grid and block dimensions calculations
- The calculations for the interior points and the boundaries are based on the serial version

Hints

- There are two C++ abstractions that complicate things:

  1. Domain data is encapsulated inside the `Field` class
     - Allocated and initialized inside the constructor
     - Deallocated inside the destructor

Hints

- There are two C++ abstractions that complicate things:

  1. Domain data is encapsulated inside the `Field` class
     - Allocated and initialized inside the constructor
     - Deallocated inside the destructor

Hint: *OpenACC provides the* `enter data` *and* `exit data` *directives for unscoped data management*

Hints

- There are two C++ abstractions that complicate things:

  1. Domain data is encapsulated inside the `Field` class
     - Allocated and initialized inside the constructor
     - Deallocated inside the destructor

  Hint: *OpenACC provides the* `enter data` *and* `exit data` *directives for unscoped data management*

  2. Operators for accessing the domain data

Hints

- There are two C++ abstractions that complicate things:

1. Domain data is encapsulated inside the `Field` class
   - Allocated and initialized inside the constructor
   - Deallocated inside the destructor

Hint: *OpenACC provides the* `enter data` *and* `exit data` *directives for unscoped data management*

2. Operators for accessing the domain data

Hint: *Operators are just another kind of functions;* `acc routine` *directive is just for that*

Performance tips

- All OpenACC operations are synchronous. Do we really want that?
  - You may just push your operations to an activity queue (note: cuBLAS is pushing to queue 0), but . . .
  - . . . you will need to synchronize sometimes!

- Fine tune your parallelism

- Try dropping the C++ operator abstractions
  - Code will become ugly (and error-prone), but you may get some bits of performance; judge if it's worth it.