How to add OpenACC in an OpenSHMEM application?

OpenSHMEM Workshop
March 4th 2014
Agenda

- OpenACC Overview and Compilers
  - Lab Session 1: Using CAPS Compilers
- Programming Model
  - Lab Session 2: Offloading Computations
- Managing Data
  - Lab Session 3: Optimizing Data Transfers
- Specifying Parallelization
  - Lab Session 4: Optimizing Compute Kernels
- Asynchronism
- Runtime API
OpenACC Overview and Compilers
Three ways of programming GPGPU applications:

- Libraries
  - Ready-to-use Acceleration

- Directives
  - Quickly Accelerate Existing Applications

- Programming Languages
  - Maximum Performance
Directive-based Programming (2)

```
main() {
    double pi = 0.0; long i;

    #pragma omp parallel for reduction(:pi)
    for (i=0; i<N; i++)
    {
        double t = (double)((i+0.05)/N);
        pi += 4.0/(1.0+t*t);
    }

    printf("pi = %f\n", pi/N);
}
```

```
main() {
    double pi = 0.0; long i;

    #pragma acc parallel
    for (i=0; i<N; i++)
    {
        double t = (double)((i+0.05)/N);
        pi += 4.0/(1.0+t*t);
    }

    printf("pi = %f\n", pi/N);
}
```
Advantages of Directive-based Programming

- Simple and fast development of accelerated applications

- Non-intrusive

- Helps to keep a unique version of code
  - To preserve code assets
  - To reduce maintenance cost
  - To be portable on several accelerators

- Incremental approach

- Enables "portable" performance
A CAPS, CRAY, Nvidia and PGI initiative
Open Standard
A directive-based approach for programming heterogeneous many-core hardware for C and FORTRAN applications

http://www.openacc-standard.com
OpenACC Compilers (1)

CAPS Compilers:
- Source-to-source compilers
- Support Intel Xeon Phi, NVIDIA GPUs, AMD GPUs and APUs

PGI Accelerator
- Extension of x86 PGI compiler
- Support Intel Xeon Phi, NVIDIA GPUs, AMD GPUs and APUs

Cray Compiler:
- Provided with Cray systems only
Are source-to-source compilers, composed of 3 parts:

- **The directives** (OpenACC or OpenHMPP)
  - Define parts of code to be accelerated
  - Indicate resource allocation and communication
  - Ensure portability
- **The toolchain**
  - Helps building manycore applications
  - Includes compilers and target code generators
  - Insulates hardware specific computations
  - Uses hardware vendor SDK
- **The runtime**
  - Helps to adapt to platform configuration
  - Manages hardware resource availability
CAPS Compilers (3)

- Take the original application as input and generate another application source code as output
  - Automatically turn the OpenACC source code into a accelerator-specific source code (CUDA, OpenCL)
- Compile the entire hybrid application
- Just prefix the original compilation line with `capsmc` to produce a hybrid application

```bash
$ capsmc gcc myprogram.c
$ capsmc gfortran myprogram.f90
```
- CAPS Compilers drives all compilation passes
- Host application compilation
  - Calls traditional CPU compilers
  - CAPS Runtime is linked to the host part of the application
- Device code production
  - According to the specified target
  - A dynamic library is built

[Caps-Enteprise.com](www.caps-entreprise.com)
CAPS Compilers Options

- **Usage:**
  
  ```
  $ capsmc [CAPSMC_FLAGS] <host_compiler> [HOST_COMPILER_FLAGS] <source_files>
  ```

- **To display the compilation process**
  
  ```
  $ capsmc -d -c gcc myprogram.c
  ```

- **To specify accelerator-specific code**
  
  ```
  $ capsmc --openacc-target CUDA gcc myprogram.c #(default)
  $$ capsmc --openacc-target OPENCL gcc myprogram.c #(AMD and Phi)
  ```
Lab Session 1: Using CAPS Compilers
Lab 1: Using CAPS Compilers
Programming Model
Programming Model

- Express data and computations to be executed on an accelerator
  - Using marked code regions

- Main OpenACC constructs
  - Parallel and kernel regions
  - Parallel loops
  - Data regions
  - Runtime API

Data/stream/vector parallelism to be exploited by HWA e.g. CUDA / OpenCL

CPU and HWA linked with a PCIx bus
How to address these systems?

- OpenACC and OpenSHMEM

- OpenSHMEM will manage
  - Inter-node communications
  - Intra-node communications

- What about the accelerator? OpenACC!
  - Communication between accelerator and host
  - Computation on the accelerator
OpenACC Execution Model

- Among a bulk of computations executed by the CPU, some regions can be offloaded to hardware accelerators
  - Parallel regions
  - Kernels regions

- Host is responsible for:
  - Allocating memory space on accelerator
  - Initiating data transfers
  - Launching computations
  - Waiting for completion
  - Deallocating memory space

- Accelerators execute parallel regions:
  - Use work-sharing directives
  - Specify level of parallelization
OpenACC Execution Model

- Host-controlled execution
- Based on three parallelism levels
  - Gangs – coarse grain
  - Workers – fine grain
  - Vectors – finest grain
In CAPS Compilers, gangs, workers and vectors correspond to the following in a CUDA grid:

- **gridDim.x** = number of **gangs**
- **blockDim.y** = number of **workers**
- **blockDim.x** = number of **vectors**
- **gridDim.y** = 1

**Beware:** this implementation is compiler-dependent.
Directive Syntax

- **C**
  ```
  #pragma acc directive-name [clause [, clause] ...]
  {
    code to offload
  }
  ```

- **Fortran**
  ```
  !$acc directive-name [clause [, clause] ...]
  code to offload
  !$acc end directive-name
  ```
Parallel Construct

- Starts parallel execution on the accelerator
- Creates gangs and workers
- The number of gangs and workers remains constant for the parallel region
- One worker in each gang begins executing the code in the region

```c
#pragma acc parallel [...]
{
    ...
    for(i=0; i < n; i++) {
        for(j=0; j < n; j++) {
            ...
        }
    }
    ...
}
```

Code executed on the hardware accelerator
Kernels Construct

- Defines a region of code to be compiled into a sequence of accelerator kernels
  - Typically, each loop nest will be a distinct kernel
- The number of gangs and workers can be different for each kernel

```c
#pragma acc kernels [...] 
{
  for(i=0; i < n; i++) {
    ...
  }
  ...
  for(j=0; j < n; j++) {
    ...
  }
}
```

```c
$!acc kernels [...] 
  
  DO i=1,n
  ...
  END DO
  ...
  DO j=1,n
  ...
  END DO
$!acc end kernels
```

1st Kernel

2nd Kernel
Lab Session 2: Offloading Computations
Lab 2: Offloading Computations
Managing Data
What is the problem using discrete accelerators?

- PCIe transfers have huge latencies

- In *kernels* and *parallel* regions, data are implicitly managed
  - Data are automatically transferred to and from the device
  - Implies possible useless communications

- Avoiding transfers leads to a better performance

- OpenACC offers a solution to control transfers
Device Memory Reuse

- In this example:
  - A and B are allocated and transferred for the first *kernels* region
  - A and C are allocated and transferred for the second *kernels* region

- How to reuse A between the two *kernels* regions?
  - And save transfer and allocation time

```c
float A[n];

#pragma acc kernels
{
    for(i=0; i < n; i++) {
        A[i] = B[n - i];
    }
}

... shmem_float_get(C, ...)
...
#pragma acc kernels
{
    for(i=0; i < n; i++) {
        C[i] += A[i] * alpha;
    }
}
```
How to tell the compiler: “that data has already been allocated”?

The *present* clause declares data that are already present on the device
  - Thanks to data region that contains this region of code

CAPS Runtime will find and use the data on device
Data Construct: Create and Present Clause

float A[n];
#pragma acc data create(A)
{
    #pragma acc kernels present(A)
    {
        for(i=0; i < n; i++) {
            A[i] = B[n – i];
        }
    }
    ...
    shmem_float_get(C, ...)
    ...
    #pragma acc kernels present(A)
    {
        for(i=0; i < n; i++) {
            C[i] += A[i] * alpha;
        }
    }
}

Allocation of A of size n on the device

Reuse of A already allocated on the device

Reuse of A already allocated on the device

Deallocation of A on the device
How is the data stored in a data region?

A data construct defines a section of code where data are mirrored between host and device.

Mirroring duplicates a CPU memory block into the HWA memory:
- The mirror identifier is a CPU memory block address
- Only one mirror per CPU block
- Users ensure consistency of copies via directives
Arrays and Subarrays (1)

- In C and C++, specified with start and length

  ```
  #pragma acc data create a[0:n]
  OR
  #pragma acc data create a[:n]
  ```

  - Allocation of an array `a` of size `n`

  ```
  #pragma acc data create a[2:n/2]
  ```

  - Allocation of a subarray of `a` of size `n/2`

- Static arrays can be allocated without having to specify their size: it’s known at compile time

- Length of dynamically allocated arrays must be explicitly specified
Arrays and Subarrays Example

```c
#pragma acc data create(A[:n])
{
    #pragma acc kernels present(A[:n])
    {
        for(i=0; i < n; i++) {
            A[i] = B[n - i];
        }
    }
}

#pragma acc data create(A(1:n))

#pragma acc kernels present(A(1:n))
{
    do i=1,n
        A(i) = B(n - i)
    end do
}

#pragma acc kernels present(A(1:n))
{
    do i=1,n
        C(i) = A(i) * alpha + C(i)
    end do
}

#pragma acc end data
```
Redundant Transfers

In this example:
- **A** is allocated for the data section
  - No data transfer of A between host and device
- **B** is allocated and transferred for the first *kernels* region
  - Input transfer
  - Output transfer
- **C** is allocated and transferred for the second *kernels* region
  - Input transfer
  - Output transfer

**How to avoid useless data transfers for B and C?**

```c
#pragma acc data create(A[:n])
{
    #pragma acc kernels present(A[:n])
    {
        for(i=0; i < n; i++) {
            A[i] = B[n - i];
        }
    }

    ...

    #pragma acc kernels present(A[:n])
    {
        for(i=0; i < n; i++) {
            C[i] = A[i] * alpha;
        }
    }
}
```
Output Transfers: Copyin and Copyout Clause

- Copyin declares data that need only to be copied **from the host to the device** when entering data section
  - Performs input transfers only

- Copyout declares data that need only to be copied **from the device to the host** when exiting data section
  - Performs output transfers only

- It defines scalars, arrays and subarrays to be allocated on the device memory for the duration of the data region
If we change the example, how to express that input and output transfers of C are required?

Use `copy` clause to:
- Declare data that need to be copied from the host to the device when entering the `data` section
- Assign values on the device that need to be copied back to the host when exiting the `data` section
- Allocate scalars, arrays and subarrays on the device memory for the duration of the `data` region

It corresponds to the default behavior of data regions.
Present_or_* Clauses

- Combines two behaviors

- Declares data that may be present
  - If data is already present, use value in the device memory
  - If not, allocate data on device when entering region and deallocate when exiting and transfer data if needed

- Syntax example: `present_or_create`
  - May be shortened to `pcreate`
Present_or_* clauses are generally safer
CAPS Compilers is able to detect the variables required on the device for the *kernels* and *parallel* constructs.

According to the specification, depending on the type of the variables, they follow the following policies:

- Tables: *present_or_copy* behavior
- Scalar
  - On a *parallel* construct: *firstprivate* behavior by default
  - On a *kernels* construct: *copy* behavior by default
The default behavior, especially on *kernels* is very error-prone:
- Write-only variables that could be privatized have a copy default policy

The default policy leads to needless copies:
- Loop bounds are usually constant, yet copied in and out
- Write-only arrays don’t need to be initialized, yet copied in and out
- Etc...

The *default*(none) policy, on *kernels* or *parallel* will issue a compilation error unless all variables have their policy specified explicitly: can be time consuming, but allows to fix bugs easily in the process.
Conducts and Directives

- OpenACC defines two ways of managing accelerator allocations and transfers
  - With *data* constructs followed by allocation or transfer clauses
  - Or standalone directives for allocations or transfers

- Data constructs are declarative
  - They define properties for a code regions and variables

- Imperative directives are standalone statements
Declare Directive: use the variable’s scope as lifetime on device

- In Fortran: used in the declaration section of a subroutine, function, or module
- In C/C++: follow a variable declaration
- Specifies variables or arrays to be allocated on the device memory for the duration of the function, subroutine or program
- Specifies the kind of transfer to realize (create, copy, copyin, etc)

```c
float A[n];
#pragma acc data create(A)
{
    #pragma acc kernels present(A)
    {
        for(i=0; i < n; i++) {
            A[i] = B[n - i];
        }
    }
    ...
}
```

```c
float A[n];
#pragma acc declare create(A)
#pragma acc kernels present(A)
{
    for(i=0; i < n; i++) {
        A[i] = B[n - i];
    }
}
```
Update Directive: make explicit transfers

- Used within explicit or implicit data region
- Updates all or part of host memory arrays with values from the device when used with *host* clause
- Updates all or part of device memory arrays with values from the host when used with *device* clause

```c
!$acc kernels copyout(A(1:n)) \ 
   copyin (B(1:n))
   do i=1,n
     A(i) = B(n - i)
   end do
!$acc end kernels

!$acc data create( A(1:n), \ 
                  B(1:n) )

!$acc update device (B(1:n))
!$acc kernels
do i=1,n
  A(i) = B(n - i)
end do
!$acc end kernels

!$acc update host (A(1:n))
!$acc end kernels
```
Lab session 3:
Data Management
Lab 3: Data Management
Specifying Parallelization
In parallel constructs, the number of gangs, workers and vectors is the same for the entire section.

The clauses:
- `num_gangs`
- `num_workers`
- `vector_length`

Enable to specify the number of gangs, workers and vectors in the corresponding parallel section.

```c
#pragma acc parallel, num_gangs(128) \ num_workers(256)
{
...
for(i=0; i < n; i++) {
    for(j=0; j < m; j++) {
        ...
    }
}
...
```
A `Loop` directive applies to a loop that immediately follow the directive.

The parallelism to use is described by one of the following clause:
- Gang for coarse-grain parallelism
- Worker for middle-grain parallelism
- Vector for fine-grain parallelism
Gangs (1)

- **Gang clause:**
  - The iterations of the following loop are executed in parallel
  - Iterations are distributed among the gangs available
  - In a *parallel* construct, no argument is allowed

```c
#pragma acc parallel, num_gangs(128) \ numWorkers(192)
{
  ...
  #pragma acc loop gang
  for (i=0; i < n; i++) {
    for (j=0; j < m; j++) {
      ...
    }
  }
  ...
}
```

192 128
#pragma parallel num_gang(2)
{
    #pragma acc loop gang
    for(i = 0; i < n; i ++)
    {
    }
}

if(i = 0; i < n/2; i ++)
{
}

if(i = n/2; i < n; i ++)
{
}
Workers

- **Worker clause:**
  - The iterations of the following loop are executed in parallel
  - Iterations are distributed among the multiple workers within a single gang
  - Loop iterations must be data independent, unless it performs a reduction operation
  - In a parallel construct, no argument is allowed
• **Vector clause**
  o The iterations of the following loop are executed in SIMD mode
  o Iterations are distributed among the multiple workers within a single gang
  o In a *parallel* construct, no argument is allowed
The parallelism description is the same as in parallel sections.

However, these clauses accept an argument to specify the number of gangs, workers or vectors to use.

Every loop can have a different number of gangs, workers or vectors in the same kernels region.
Data Independency

- In kernels sections, the clause *independent* specifies that iterations of the loop are data-independent.

- The user does not have to think about gangs, workers or vector parameters.

- Allows the compiler to generate code to execute the iterations in parallel with no synchronization.

```
A[0] = 0;
#pragma acc loop independent
for(i=1; i<n; i++)
{
    A[i] = A[i]-1;
}
```

```
A(1) = 0
$!acc loop independent
DO i=2, n
    A(i) = A(i-1)
END DO
```

**Programming error**
Extra OpenACC loop clauses

- Sequential execution
- Automatic parallelization
  - Compiler should parallelize a loop if proved parallel
- Loop collapsing
- Loop tiling
- Privatize variable
- Reduction
- Hardware specific
- Atomics
By default, if a function is called from a kernel/parallel section, it is inlined
  - Requires to have the body of the function in the same file as where it’s called
  - Leads to code bloat
Marking a function with a “routine” pragma allows to generate a HWA for later use from a kernel/parallel section

```
#pragma acc routine [gang, worker, seq...]
int do_stuff(int n, float *t) {
}
```

As the caller’s context can be different (gang, worker, vector) it may be necessary to specify which context will be used to enable work-sharing in the routine.
Routine (2): nested parallelism

- Before OpenACC 2.0, a kernel/parallel section was not allowed to call a function that contains other kernels/parallel section
- In theory, this should get rid of some call overhead, and needless device ⇔ host ⇔ device copies
  - In practice, current generation of HWA do not allow improve performance in a noticeable way.
Lab Session 4: Compute Kernels
Lab 4: Compute Kernels
Asynchronism & Runtime API
Asynchronism

- By default, the code on the accelerator is synchronous
  - The host waits for completion of the parallel or kernels region

- The `async` clause enables to use the device while the host process continues with the code following the region

- Can be used on `parallel` and `kernels` regions and `update` directives
Wait Directive (1)

- Causes the program to wait for an asynchronous activity
  - Parallel, kernels regions or update directives
- An identifier can be added to the async clause and wait directive:
  - Host thread will wait for the asynchronous activities with the same ID
- Without any identifier, the host process waits for all asynchronous activities

```c
#pragma acc kernels, async
{
    ...
}
#pragma acc kernels, async
{
    ...
}
#pragma acc wait
```
```c
#!acc kernels, async (1)
    ...
#!acc end kernels
    ...
#!acc kernels, async (2)
    ...
#!acc end kernels
    ...
#!acc wait (1)
```
Compute regions have an implicit *data* section.
- Most HWA do not offer asynchronous transfer from non-pinned memory
- Dependence between kernels is bound by their data dependence

In practice, an explicit *data* section is required

```acc
!acc data create (t1,t2,t3)
!acc kernels, async (1)
  ... write t1()
!acc end kernels
  ...
!acc kernels, async (2)
  ... write t2()
!acc end kernels
  ...
!acc wait(1,2), async(3)

!acc kernels, async(3)
  ... write t3(), read t1(),t2()
!acc end kernels
  ...
!acc end data
```
OpenACC sections defines the behavior of the accelerator
  - What happens if there is no accelerator?
  - What if the OpenACC code should also be executed on the host?

The *if* clause enables to generate two copies of the OpenACC code:
  - One to be executed on the host
  - One to be executed on the accelerator
### If Clause

- Available on *parallel, kernels* or *data constructs* and *update* directive

- When clause evaluation corresponds to:
  - Zero in C or C++ or `.false.` in Fortran, the host copy is executed
  - Nonzero in C or C++ or `.true.` in Fortran, the accelerator copy is executed

```c
#pragma acc kernels if(cond) {
    for(i=0; i < n; i++) {
        ...
    }
    ...
}
```

```fortran
$!acc kernels if(cond)
    DO i=1,n
        ...
    END DO
    ...
$!acc end kernels
```
May limit portability of the code
- Conditional compilation using _OPENACC preprocessor variable is available

Enables to:
- Query available HWA,
- Retrieve environment information,
- Program in OpenACC without having to use directives
- Do things that aren’t possible with directives
  - Ex: do a partial update of a structure field
Runtime API

- Initialize the runtime for a device type
- Disconnect a device from a program
- Get the number of device
- Set/get the type of device
- Allocate/free memory
- Transfer data
Check number of CUDA devices available on the system
• If 0 is return, no CUDA device is available

```c
int dev;
Dev = acc_get_num_device(acc_device_cuda);

#pragma acc data copy(A[0:N]) if (dev)
{
  #pragma acc kernels if (dev)
  ...

  #pragma acc kernels if (dev)
  for (int i = 0+t*N/2; i < (1+t)*N/2; ++i) {
  }
  ...
}
```
If no device is available, the host code is executed
Two CPU threads are created with OpenMP:
- thread #0 will manage device #0
- thread #1 will manage device #1

Data set is split in two:
- each set will be processed by one device

```c
#pragma omp parallel for
for (int t = ; t < 2; ++t) {
    acc_set_device_num(t, acc_device_default);

    #pragma acc kernels copy(A[0+t*N/2:(1+t)*N/2])
    {
        #pragma acc loop independent
        for (int i = 0+t*N/2; i < (1+t)*N/2; ++i) {
        }
        ...
    }

    acc_shutdown(acc_device_default)
}
```
Conclusion
Conclusion

- **OpenSHMEM and OpenACC are fully compatible!**
  - For now, OpenACC regions cannot contain OpenSHMEM communications

- **OpenACC allows you to address the accelerator**
  - Fast development of high-level heterogeneous applications
  - Beware of compiler-dependent behaviors
  - Explicit the calls to an accelerator
  - Whatever the target
    - Nvidia GPU
    - AMD GPU and APU
    - Intel Xeon Phi
Accelerator Programming model
Directive-based programming
Parallel Computing
OpenHMPP
OpenACC
GPGPU
Many-Core programming
Parallelization
HPC
NVIDIA Cuda
Code speedup
High Performance Computing
OpenCL
CAPS Compilers
Portability
Many-Core programming
Performance
CAPS Workbench
CAPS Compilers
Visit CAPS Website:
www.caps-entreprise.com