OpenACC Standard

Directives for Accelerators

Credits

- http://www.openacc.org/
  - V1.0: November 2011 Specification
- OpenACC, Directives for Accelerators, Nvidia Slideware
- CAPS OpenACC Compiler, HMPP Workbench 3.1.x, CAPS entreprise

Agenda

- OpenACC Overview and Compilers
- Programming Model
- Managing Data
- Loops
- Asynchronism
- Runtime API

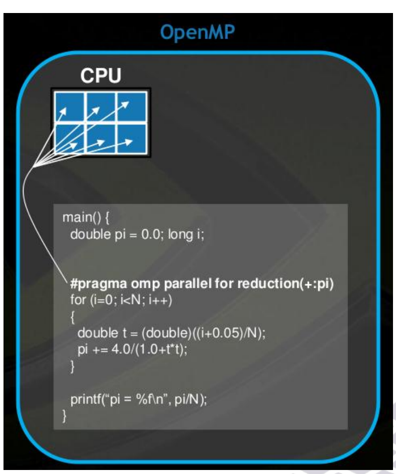
OpenACC Overview and Compilers

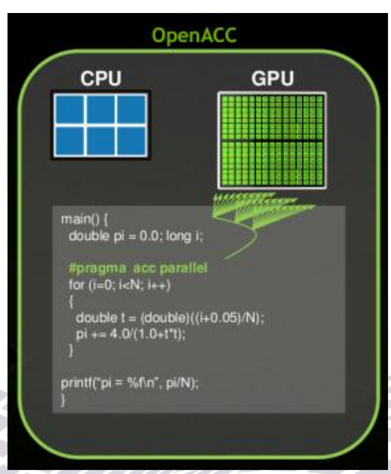
Directive-based Programming

- Three ways of programming GPGPU applications:

Libraries	Directives	Programming Languages
Ready-to-use Acceleration	*Quickly Accelerate Existing Applications*	*Maximum Performance*

Directive-based Programming





Introduction to Directive-based Programming

CAPS

- Keeping a unique version of codes, preferably mono-language
  - Reduces maintenance cost
  - Preserves code assets
  - Is less sensitive to fast moving hardware targets
    - Codes last several generations of hardware architecture
- Help to get "portable" performance
  - Multiple forms of parallelism cohabiting
    - Multiple devices (e.g. GPUs) with their own address space
    - Multiple threads inside a device
    - Vector/SIMD parallelism inside a thread
  - Dealing with massive parallelism
- OpenACC is a promising approach

www.caps-entreprise.com 7

OpenACC Initiative

CAPS

OpenACC. DIRECTIVES FOR ACCELERATORS

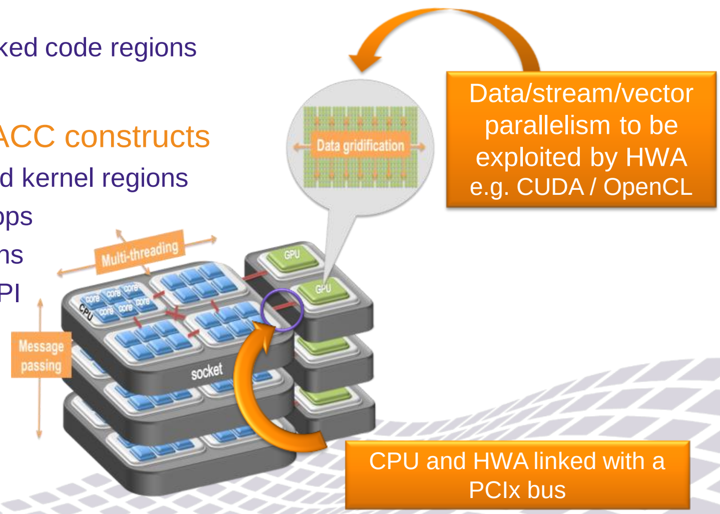
CAPS CRAY NVIDIA PGI

- A CAPS, CRAY, Nvidia and PGI initiative
- Open Standard
- A directive-based approach for programming heterogeneous many-core hardware for C and FORTRAN applications
- Available for implementation
  - As CRAY's, PGI's…
  - CAPS OpenACC Compiler ➔ released in April 2012 with HMPP 3.1
    - Satisfies the OpenACC Test Suite provided by University of Houston
- Visit http://www.openacc-standard.com for more information

www.caps-entreprise.com 8

OpenACC Initiative

- 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

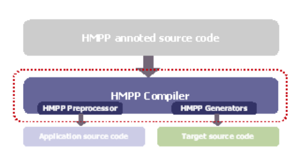


HMPP Compiler

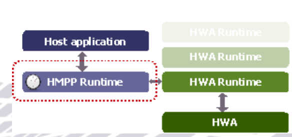
Composed of 3 parts:

- A set of directives to program hardware accelerators
  - Drive your HWAs, launch computations, manage transfers

- A complete toolchain to build manycore applications
  - Build your hybrid application



- A runtime to adapt to platform configuration



HMPP Compiler

CAPS

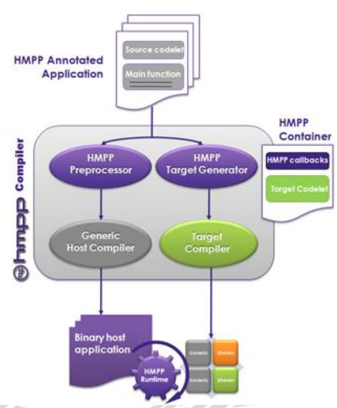
- The directives
  - Define hardware implementations of native functions (codelets)
  - Indicate resource allocation and communication
  - Ensure portability (future-proof) and default execution (no exit cost)
- The toolchain
  - Helps building manycore applications
  - Includes compilers and target code generators
  - Insulates hardware specific computations
  - Uses hardware vendor SDK
- The runtime
  - Helps to adapt to platform configuration
  - Manages hardware resource availability

www.caps-entreprise.com 11

HMPP Compiler

CAPS

- HMPP drives all compilation passes
  - Host application compilation
    - Calls traditional CPU compilers
    - HMPP Runtime is linked to the host part of the application
  - Device code production
    - According to the specified target
    - A dynamic library is built




```
$ hmpp gcc myprogram.c
$ hmpp gfortran myprogram.f90
```

www.caps-entreprise.com 12

Programming Model

Execution Model

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

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

- Accelerators execute parallel regions:
  - Use work-sharing directive
  - Specify level of parallelization

Levels of Parallelism

CAPS

- Host-controlled execution
- Based on three parallelism levels
  - Gangs – coarse grain
  - Workers – fine grain
  - Vectors – finest grain

Directive Syntax

CAPS

- C

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

- Fortran

```
!$acc directive-name [clause [, clause] …]
  code to offload
!$acc end directive-name
```

Work Management: Parallel Construct

CAPS

- 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

```
#pragma acc parallel […]
{
  …
}
```

```
$!acc parallel […]
  …
$!acc end parallel
```


Parallel Construct: Gangs and Workers

CAPS

- The clauses:
  - *num_gangs*
  - *num_workers*

Enables to specify the number of gangs and workers in the corresponding *parallel* section

```
#pragma acc parallel, num_gangs[32], num_workers[256]
{
  …
  for(i=0; i < n; i++) {
    for(j=0; j < n; j++) {
      …
    }
  }
  …
}
```

Work distribution over 32 gangs and 256 workers

Work Management: Kernels Construct

CAPS

- 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

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

```
$!acc kernels […]

  DO i=1,n
    …
  END DO
  …
  DO j=1,n
    …
  END DO

$!acc end kernels
```

1st Kernel

2nd Kernel

www.caps-entreprise.com 19

CAPS

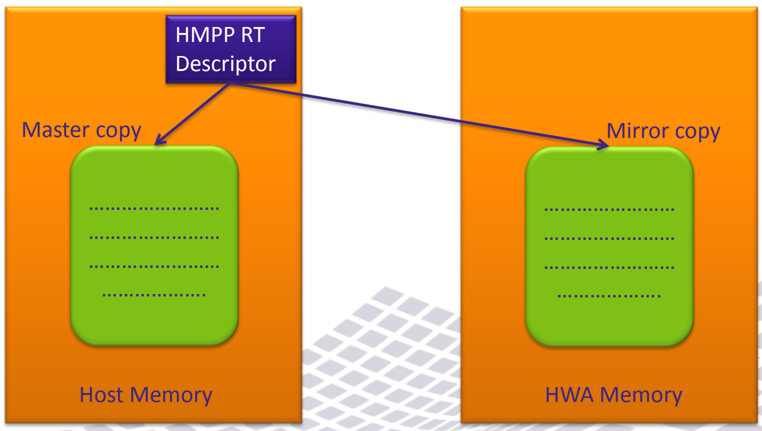
Managing Data

11/10/2012www.caps-entreprise.com 20

Data Storage

CAPS

- Mirroring duplicates a CPU memory block into the HWA memory
  - Mirror identifier is a CPU memory block address
  - Only one mirror per CPU block
  - Users ensure consistency of copies via directives



Data Management: Data Constructs

CAPS

- Defines scalars, arrays and subarrays to be allocated on the device memory for the duration of the region
  - Data can be copied from the host to the device when entering region
  - Data can be copied from the device to the host when exiting region
- *if* clause can be used

```
#pragma acc data […]
{
   …
}
```

```
$!acc data […]
   …
$!acc end data
```

Data Allocation: Create Clause

- Declares variable, arrays or subarrays to be allocated in the device memory
- No data specified in this clause will be copied between host and device

```
#pragma acc data, create (A)
{
   …
}
```

```
$!acc data, create (A)
   …
$!acc end data
```


Subarrays

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

```
a[2:n]
```

ie: elements a[2], a[3], …, a[2+n-1]

  - If the lower bound is missing, zero is used
  - If the length is missing, the difference between the lower bound and the declared size of the array is used

- In Fortran, specified with a list of range specifications

```
a(1:3,5:6)
```

ie: elements a(1,5), a(2,5), a(3,5), a(1,6), a(2,6), a(3,6)

- Any Array or subarray must be a contiguous block of memory

Transfers: Copy Clause

- Declares data that need to be copied from the host to the device when entering the data section
- These data are assigned values on the device that need to be copied back to the host when exiting the data section

```
#pragma acc data, copy (A)
{
   ...
}
```

```
$!acc data, copy (A)
   ...
$!acc end data
```


Transfers: Copyin/Copyout Clause

- *copyin*
  - Declares data that need to be copied from the host to the device when entering the data section
- *copyout*
  - Declares data that need to be copied from the device to the host when exiting data section

```
#pragma acc data, copyin (A)
{
   ...
}
```

```
$!acc data, copyout (A)
   ...
$!acc end data
```

Present Clause

- Declares data that are already present on the device
  - Thanks to data region that contains this region of code
- HMPP Runtime will find and use the data on device

```
#pragma acc data, copy (A)
{
  …
  #pragma acc data, present (A)
  {
    …
  }
}
```

```
$!acc data, copy (A)
  …
  $!acc data, present (A)
  …
  $!acc end data
$!acc end data
```


Data Allocation: Present_or_create Clause

- 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
- May be shortened to *pcreate*

```
#pragma acc data, pcreate (A)
{
  …
}
```

```
$!acc data, pcreate (A)
  …
$!acc end data
```

Transfers: Present_or_copy Clause

- If data is already present, use value in the device memory
- If not:
  - Allocates data on device and copies the value from the host at region entry
  - Copies the value from the device to the host and deallocate memory at region exit
- May be shortened to *pcopy*

```
#pragma acc data, pcopy (A)
{
   …
}
```

```
$!acc data, pcopy (A)
   …
$!acc end data
```


Transfers: Present_or_copyin / Present_or_copyout Clause

- If data is already present, use value in the device memory
- If not:
  - Both *present_or_copyin*/*present_or_copyout* allocate memory on device at region entry
  - *present_or_copyin* copies the value from the host at region entry
  - *present_or_copyout* copies the value from the device to the host at region exit
  - Both *present_or_copyin*/*present_or_copyout* deallocate memory at region exit
- May be shortened to *pcopyin* and *pcopyout*

```
#pragma acc data, pcopyin (A)
{
   …
}
```

```
$!acc data, pcopyout (A)
   …
$!acc end data
```

Kernels, Parallel Contructs and Data Clauses

CAPS

- *Kernels* and *parallel* constructs implicitly define data regions
- Data clauses also apply to these structures
- *Kernels* and *parallel* constructs cannot contain other *kernels* or *parallel* regions
- Data inside *kernels* or *parallel* regions data can be managed by a data construct at an higher level

data.c

```
int A[n]
…
#pragma acc data, copyin (A)
{
   …
   function(A)
   …
}
```

kernels.c

```
function(float A[n])
{
  #pragma acc kernels, \
                pcopyin (A)
  {
     …
  }
}
```

11/10/2012 www.caps-entreprise.com 31

Data Management: Default Behavior

CAPS

- HMPP compiler is able to detect the variables required on the device for the *kernels* and *parallel* constructs.
- Depending on their type, they follow the following policies
  - Tables: *present_or_copy* behavior
  - Scalar
    - if not live in or live out variable: *private* behavior
    - *copy* behavior otherwise

www.caps-entreprise.com 32

Loop Constructs

Kernel Optimization: Loop Construct

- *Loop* directive applies to a loop that immediately follow the directive
- Describes what kind of parallelism to use

```
#pragma acc loop […]
for(i=0; i<n; i++)
{
  …
}
```

```
$!acc loop […]
DO i=1,n
    …
END DO
```

Sequential Execution

CAPS

- The *seq* clause specifies that the associated loop should be executed sequentially
- This is the default behavior in a *parallel* region

```
#pragma acc loop seq
for(i=0; i<n; i++)
{
  …
}
```

```
$!acc loop seq
DO i=1,n
    …
END DO
```


Data Independence

CAPS

- The clause *independent* specifies that iterations of the loop are data-independent
- Allowed on loop directives in kernels regions
- Allows the compiler to generate code to execute the iterations in parallel with no synchronisation

```
#pragma acc loop independent
for(i=0; i<n; i++)
{
  for(j=0; j<m; j++)
  {
    A(j,i*3+MOD(i,2)) = i*j;
  }
}
```

```
#pragma acc loop independent
for(i=0; i<n; i++)
{
  for(j=0; j<m; j++)
  {
    A(j,i*3+MOD(i,2)) =
                   i*A(i,j-1);
  }
}
```

Programming error

Gangs

- *Gang* clause:
  - The iterations of the following loop are executed in parallel
  - In a parallel construct:
    - Iterations are distributed among the gangs created by the *parallel* contruct
    - No argument is allowed
  - In a kernels construct
    - Iterations are distributed among the gangs created by the kernel created by a loop
    - An argument can specify the number of gangs to use for this loop

Workers

- *Worker* clause:
  - The iterations of the following loop are executed in parallel
  - In a parallel construct:
    - Iterations are distributed among the multiple workers withing a single gang
    - No argument is allowed
    - Loop iterations must be data independent, unless it performs a reduction operation
  - In a kernels construct:
    - Iterations are distributed among the workers within the gangs created by the kernel within a loop
    - An argument can specify the number of workers to use for this loop

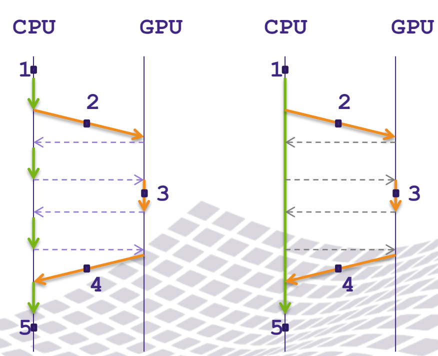
CAPS

Asynchronism

Asynchronism

CAPS

- 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

CAPS

- 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

```
#pragma acc kernels, async
{
   …
}
#pragma acc kernels, async
{
   …
}
#pragma acc wait
```

```
$!acc kernels, async 1
   …
$!acc end kernels
   …
$!acc kernels, async 2
   …
$!acc end kernels
   …
$!acc wait 1
```


CAPS

Runtime API

Runtime Library Definition

- For C:
  - Header file: openacc.h
- For Fortran:
  - Interface declaration in: openacc_lib.h in a Fortran module called openacc
- acc_device_t: type of accelerator device
  - acc_device_none
  - acc_device_default
  - acc_device_host
  - acc_device_not_host
  - …

www.caps-entreprise.com 43

Runtime API

- int *acc_get_num_device* (acc_device_t) (C)
- integer function *acc_get_num_device* (devicetype) (Fortran)
  - Returns the number of accelerator devices of the given type attached to the host
- int *acc_set_device_type* (acc_device_t) (C)
- subroutine *acc_set_device_type* (devicetype) (Fortran)
  - Tells the runtime which type of device to use
- acc_device_type *acc_get_device_type* (void) (C)
- function *acc_get_device_type* () (Fortran)
  - Tells the program what type of device will be used

www.caps-entreprise.com 44

Runtime API

- void *acc_set_device_num* (int, acc_device_t) (C)
- subroutine *acc_set_device_num* (devicenum, devicetype) (Fortran)
  - Tells the runtime which device to use
- int *acc_get_device_num* (acc_device_t) (C)
- Integer function *acc_get_device_num* (devicetype) (Fortran)
  - Return the device number of the specified device type that will be used

Runtime API

- void acc_init (acc_device_t) (C)
- Subroutine acc_init (devicetype) (Fortran)
  - Initialize the runtime for the given type
- void acc_shutdown (acc_device_t) (C)
- Subroutine acc_shutdown (devicetype) (Fortran)
  - Disconnect the program from the accelerator device
- void* acc_malloc (size_t) (C)
  - Allocates memory on accelerator device
  - Pointers assigned to this function may be reused
- void* acc_free (size_t) (C)
  - Deallocates memory on accelerator device

Conclusion

CAPS

- Beware of compiler-dependent behaviors
- Fast development of high-level heterogenous applications
  - For C and FORTRAN code
- Explicit the calls to a hardware accelerator in your code
  - Whatever the target
  - CAPS OpenACC compiler supports:
    - Nvidia Tesla GPUs
    - AMD
    - X86 Intel Phi

11/10/2012 www.caps-entreprise.com 47

Accelerator Programming Model Parallelization CAPS
Directive-based programming GPGPU Manycore programming
Hybrid Manycore Programming HPC community OpenACC
Petaflops Parallel computing HPC open standard
Multicore programming Exaflops NVIDIA Cuda
Code speedup Hardware accelerators programming
High Performance Computing OpenHMPP
Parallel programming interface
Massively parallel
Open CL

CAPS OpenACC DIRECTIVES FOR ACCELERATORS open hmpp

http://www.caps-entreprise.com
http://twitter.com/CAPSentreprise
http://www.openacc-standard.org/
http://www.openhmpp.org