Programming Hardware Accelerators with Directives: OpenHMPP & OpenACC

CAPS Compilers

Florent Lebeau

Toulon
Dec. 20, 2012

About CAPS

CAPS Mission and Offer

Mission
Help you break the parallel wall and
harness the power of manycore computing

- Software programming tools
  - CAPS Manycore Compilers
  - CAPS Workbench: set of development tools
- CAPS engineering services
  - Recommend machine configurations
  - Diagnose application parallelism
  - Port applications to manycore systems
  - Fine tune parallel applications
- Training sessions
  - OpenACC, CUDA, OpenCL, …

Based in Rennes, France
Created in 2002
More than 10 years of expertise
About 30 employees

Many-Core Technology Insights

Various Many-Core Paths

CAPS

- Large number of small cores
- Data parallelism is key
- PCIe to CPU connection

AMD Discrete GPU

- 50+ number of x86 cores
- Support conventional programming
- Vectorization is key
- Run as an accelerator or standalone

INTEL Many Integrated Cores

- Large number of small cores
- Data parallelism is key
- Support nested and dynamic parallelism
- PCIe to host CPU or low power ARM CPU (CARMA)

NVIDIA GPU

- Integrated CPU+GPU cores
- Target power efficient devices at this stage
- Shared memory system with partitions

AMD APU

Many-core Programming Models

CAPS

Applications

Libraries	Directives (OpenACC, OpenHMPP)	Programming languages (CUDA, OpenCL,,…)
Just use as is	Standard and high level	Maximum performance and maintenance cost

CAPS Compilers

Portability and Performance

Nvidia Architecture Overview
GT200 / GF100 / GK110

Directive-based Programming

- Consortium created last year by CAPS, CRAY, NVIDIA and PGI
  - New members in 2012: Allinea, Georgia Tech, ORNL, TU-Dresden, University of Houston, Rogue Wave
  - Momentum with broad adoption
  - 2nd specification to be announced

OpenACC
DIRECTIVES FOR ACCELERATORS

- Created by CAPS in 2007 and adopted by PathScale
  - Advanced features
    - Multi devices
    - Native and accelerated libraries with directives and proxy
    - Tuning directives for loop optimizations, use of device features, …
    - Ground for innovation in CAPS
  - Interoperate with OpenACC

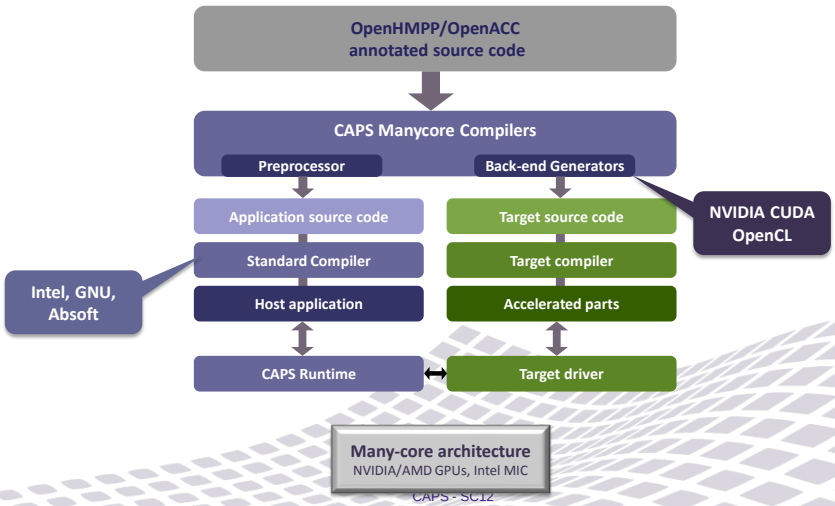
open hmpp

- First technical report proposing directives for accelerators
  - Not a specification yet
  - Aim is to get early feedback

OpenMP

CAPS Compilers

- Source to source compilers
  - Use your preferred native x86 and hardware vendor compilers



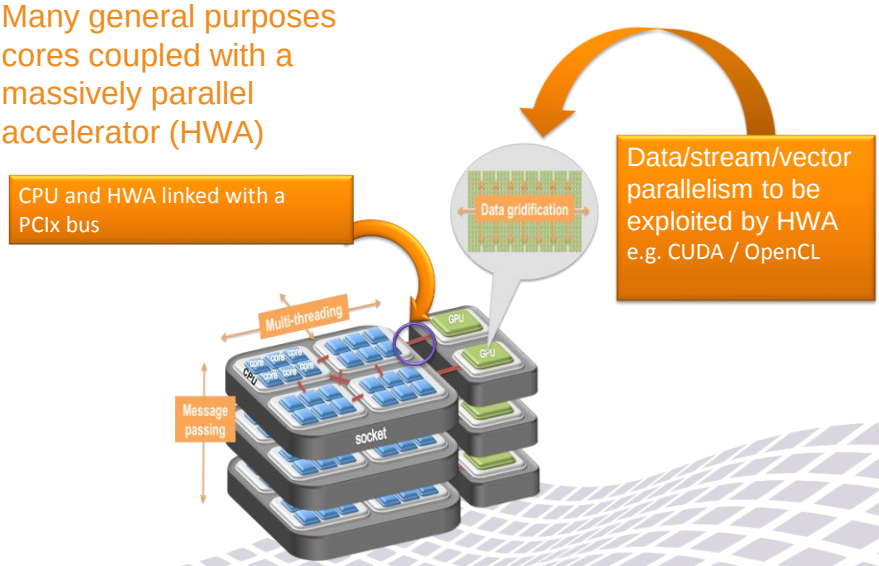
OpenACC and OpenHMPP Directive-based Programming

Directive-based Approaches

- Supplement an existing serial language with directives to express parallelism and data management
  - Preserves code basis (e.g. C, Fortran) and serial semantic
  - Competitive with code written in the device dialect (e.g. CUDA)
  - Incremental approach to many-core programming
  - Mainly targets legacy codes

Heterogeneous Many-Cores

- Many general purposes cores coupled with a massively parallel accelerator (HWA)



Remote Procedure Call Basic Operations

- A RPC sequence consists in 5 basic steps:
  (1) Allocate the HWA and the memory
  (2) Transfer the input data: CPU => HWA
  (3) Compute
  (4) Transfer the output data: CPU <= HWA
  (5) Release the HWA and the memory

How Does Directive Based Approaches Differ from CUDA or OpenCL?

- OpenHMPP/OpenACC parallel programming model is **parallel loop centric**
- CUDA and OpenCL parallel programming models are **thread centric**

```
void saxpy(int n, float alpha,
           float *x, float *y){
#pragma acc kernels loop independent
for(int i = 0; i<n; ++i)
    y[i] = alpha*x[i] + y[i];
}
```

```
__global__
void saxpy_cuda(int n,
                float alpha,
                float *x,
                float *y) {
int i = blockIdx.x*blockDim.x
        +threadIdx.x;
if(i<n)
  y[i] = alpha*x[i]+y[i];
}

int nblocks = (n + 255)/256;
saxpy_cuda <<<nblocks,256>>>
           ( n, 2.0, x, y);
```

Two Styles of Directives in CAPS Compilers

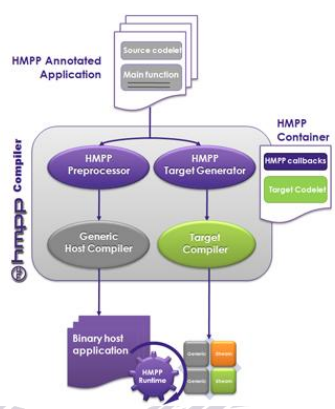
- Functions based, i.e. codelet (OpenHMPP)
- Code regions based (OpenACC)

```
#pragma hmpp myfunc codelet, …
void saxpy(int n, float alpha, float x[n], float
y[n]){
  for(int i = 0; i<n; ++i)
    y[i] = alpha*x[i] + y[i];
}
```

```
#pragma acc kernels …
{
  for(int i = 0; i<n; ++i)
    y[i] = alpha*x[i] + y[i];
}
```


Compiling with CAPS Compilers

- CAPS Compilers drives all compilation passes
  - Host application compilation
    - CAPS Runtime is linked to the host part of the application
  - HWA code production
    - Target code is produced
    - A dynamic library is built




```
$ capsmc gcc myProgram.c
```

OpenHMPP

OpenHMPP directives concepts

- OpenHMPP identifies codelets functions
  - Corresponds to hotspots of the application that can be accelerated
  - A codelet function has to be **insulated** and **offloaded** on the remote hardware accelerator
- OpenHMPP provides the user with codelet control
  - Choose when/how you want to allocate/release the hardware
  - Choose when/how you want to launch the computations (ie: codelets)
  - Choose when/how you want to start data transfers

Codelet Basic Concepts

- Pure function calls are offloaded on the accelerator
  - These functions must fit some common constraints
    - No I/O on data
    - No access to global/volatile variables
    - Fixed number of arguments
- Controlled by directives
  - Pragmas in C
  - Special comments in Fortran
- A unique label associates a set of directives

General Syntax of the Directives

- In C, a single line OpenHMPP directive is:

```
#pragma hmpp myLabel command [, attribute]
```

- In FORTRAN, any form of comment (F77, F90,...) starting with $hmpp

```
!$hmpp myLabel command [, attribute]
```

General Syntax of the Directives

- Directives are either standalone or precede the statement they are related to
- Directive comments cannot end a statement line
- The following Fortran comment is not a valid OpenHMPP directive

```
PRINT *, "Hello"  !$hmpp myLabel command
```

- Directives are case-insensitive

CODELET/CALLSITE

- Two essential OpenHMPP directives
  - **CODELET**
    - Identify the native function to offload on a specific target
    - Order HMPP Codelet Generators to produce target code
  - **CALLSITE**
    - Explicit a call to this specialized function in the application

CODELET/CALLSITE example

C

```c
#pragma hmpp myCall codelet, ...
void myFunc( int n, int A[n], int B[n])
{
    int i;
    for (i=0 ; i<n ; ++i)
        B[i] = A[i] + 1;
}



void main(void)
{
    int X[10000], Y[10000], Z[10000];
    …
    #pragma hmpp myCall callsite
    myFunc(10000, X, Y);
    ...
    myFunc(1000, Y, Z);
    …
}
```

FORTRAN

```fortran
!$hmpp myCall codelet, ...
SUBROUTINE myFunc(n,A,B)
    INTEGER, INTENT(IN)  :: n, A(n)
    INTEGER, INTENT(OUT) :: B(n)
    INTEGER :: i
    DO i=1,n
        B(i) = A(i) + 1
    ENDDO
END SUBROUTINE

PROGRAM test
    INTEGER :: X(10000), Y(10000),
Z(10000)
    ...
    !$hmpp myCall callsite
    CALL myFunc(10000,X,Y)
    ...
    CALL myFunc(10000,Y,Z)
    ...
END PROGRAM
```


Ground Rules

- An OpenHMPP program contains at least a pair of CODELET/CALLSITE directives
  - A CODELET is a specialization of a subroutine
  - A CALLSITE is the specialization of a call statement
- Each CODELET may correspond to multiple CALLSITE
  - But each CALLSITE belongs to a single CODELET
- A CALLSITE is identified by a unique label

Codelet Directive: the TARGET Attribute

- The target attribute tells CAPS Compilers for which HWA the specialized version must be generated:

```
#pragma hmpp myLabel codelet, target=CUDA
```

- CAPS Compilers can address several HWAs:
  - CUDA
  - OpenCL

Referencing Arguments in the Directives

- OpenHMPP directives are always related to a subroutine, whose arguments can be referenced by:
  - their codelet name

    ```
    args[x;y;z]
    ```

  - their numeric rank starting from 0

    ```
    args[0;4;5;6]
    ```

  - intervals of their ranks

    ```
    args[0-2;3-6]
    ```

  - a combination of the previous methods

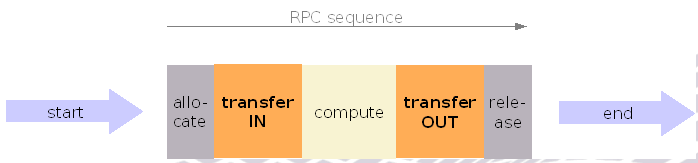
    ```
    args[x;y;4-6]
    ```

  - The wildcard can also be used to select all the codelet arguments

    ```
    args[*]
    ```

CODELET Directive: the IO Attribute (1)

- Attached to the CODELET directive and used by CALLSITE to realize the right steps in the RPC sequence
- This attribute allows to specify if an argument is
  - IN for an input (data needed on the HWA)
  - OUT for an output (result to be sent back from the HWA)
  - INOUT for both
- In the RPC sequence
  - IN and INOUT arguments are transferred in step (2)
  - OUT and INOUT arguments are transferred in step (4)



IO Default Rules

- In C
  - scalars and const arrays or pointers are IN
  - Non-const arrays or pointers are INOUT
- In Fortran
  - use INOUT by default
  - Or follow the Fortran INTENT() specification

```
#pragma hmpp myFunc codelet, args[1,2].io=in, args[C].io=out
void myFunc( int n, int A[n], int B[n], int C[n]) {
 for(int i=0; i < n; i++)
  C[i] = A[i]+B[i];
}

int main()
{
 ...
 #pragma hmpp myFunc callsite
 myFunc(100,X,Y,Z) ;
 ...
}
```

Basic Transfer Policy

- OpenHMPP has different transfer policies in order incrementally port the application
  - Use the TRANSFER attribute to specify the policy of your arguments
  - ATCALL transfer policy is the simplest: the arguments will be transferred automatically at every callsite

```
#pragma hmpp myLabel codelet, args[*].transfer=atcall, &
#pragma hmpp     &              target=CUDA
void myFunc( int n, float a, float b, float v[n])
{
    for( int i=0 ; i<n ; ++i)
        v[i] = v[i] * a + b;
}
```

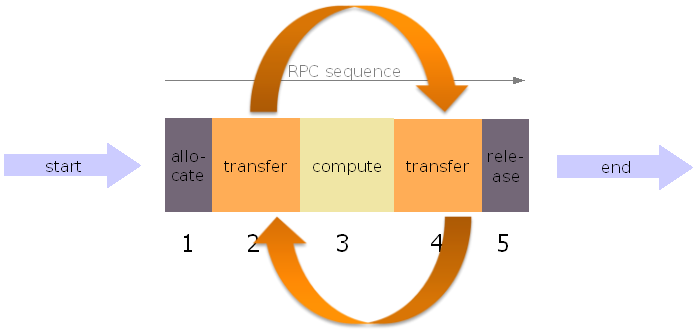

CAPS Runtime Fallback Mode

- With OpenHMPP, what happens if
  - The hardware accelerator is unavailable?
  - A data allocation fails?
  - A data transfer fails?
  - The computation raises an error?

➔ The native CPU version is used as a fallback
  - This behavior can be inhibited with HMPPRT_NO_FALLBACK different than 0

Reminder

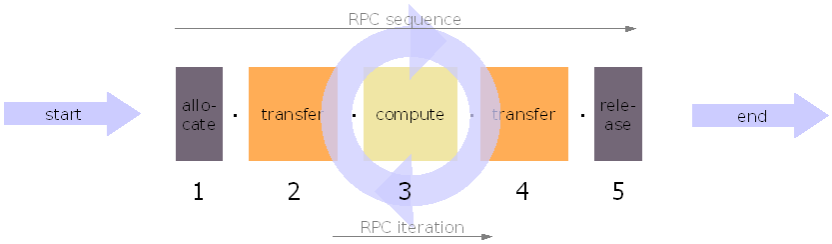
- A standalone CALLSITE directive implements the whole RPC sequence



- With ATCALL transfer policy, if you iterate on callsite all data are transferred at each call to the callsite

Splitting the RPC sequence

- To optimize HWA usage according to your application algorithm, this RPC sequence can be split



ALLOCATE directive

- Step #1 in RPC sequence
- Allocate memory **for all** codelet arguments, by default

```
#pragma hmpp myLabel allocate
```

- Possible to also allocate only specified arguments
  - This will only allocate a, b and c for mylabel

```
#pragma hmpp myLabel allocate, args[a;b;c]
```

- SIZE attribute enables to specify the size of the arrays to allocate

```
#pragma hmpp myLabel allocate, args[A], size={512,512}
```


FREE Directive

- Step #5 in RPC sequence
- Free memory **for all** codelet arguments, by default

```
#pragma hmpp myLabel free
```

- Possible to also free only specified arguments

```
#pragma hmpp myLabel free, args[a;b;c]
```

  - This will only free a, b and c for mylabel

Manual Transfer Policy

- To improve the transfer performance
  - Manually manage data transfers to transfer data from/to the GPU only when the application needs it

```
#pragma hmpp myLabel codelet, target=CUDA, &
#pragma hmpp & args[0].transfer=atcall, &
#pragma hmpp & args[1-3].transfer=manual
```

- ADVANCEDLOAD directives enables to upload data to the GPU
- DELEGATEDSTORE directives enables to download data from the GPU

ADVANCEDLOAD directive (1)

- Step #2 in RPC sequence
- Transfer arguments from CPU to HWA

```
#pragma hmpp myLabel advancedload, args[…]

!$hmpp myLabel advancedload, args[…]
```

- The standalone attribute ARGS specifies the arguments to be transferred
- The HOSTDATA attribute enable to specify the name of the data on the host

```
#pragma hmpp label advancedload, args[buffer_A], hostdata="host_A"
```

The DELEGATEDSTORE directive

- Step #4 in RPC sequence
- Transfer arguments from HWA to CPU

```
#pragma hmpp label delegatedstore, args[…]
```

- The HOSTDATA attribute can also be used

Manual Transfer Policy Example

```
void work(int n, float *alpha, float *X, float *Y, float *Z)
{
 int i,j ;
 #pragma hmpp advancedload, args[A,B,C], &
 #pragma hmpp & args[A].hostdata="X", &
 #pragma hmpp & args[A].hostdata="Y", &
 #pragma hmpp & args[A].hostdata="Z"
 for (j=1;j<n;j++)
 {
  #pragma hmpp toto callsite
  fun(n,X,Y,Z) ;
  if (j % 10 == 0) {
   for (i=0;i<n;i++) {
    X[i] += alpha ;
   }
  #pragma hmpp advancedload, args[A]
  }
 }
 #pragma hmpp advancedload, args[C]
 return ;
}
```

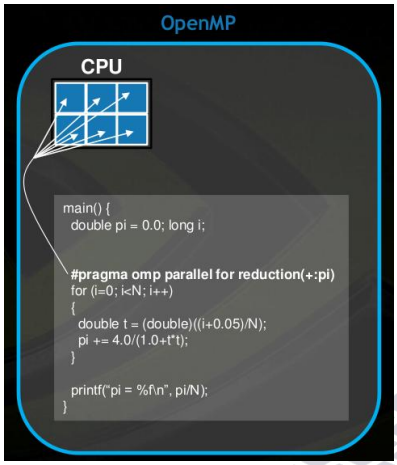
```
#pragma hmpp toto codelet, target=CUDA, &
#pragma hmpp &   args[A,B,C].transfer=manual,&
#pragma hmpp &   args[n].transfer=atcall
void
fun(int n, float A[n], float B[n], float C[n])
{
  for (int i=0;i<n;i++)
   C[i] = C[i] + A[i]*B[i] ;
}
```

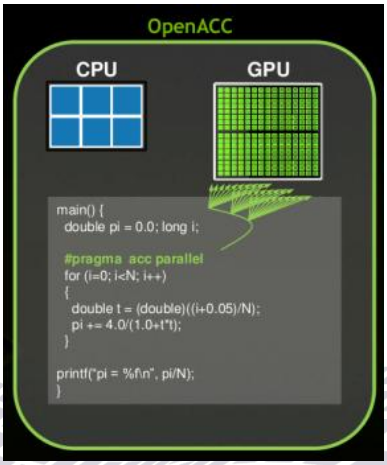
OpenACC

Execution Model

- Among a bulk of computations executed by the CPU, some regions can be offloaded to hardware accelerators
- OpenACC defines parallel regions to be executed by the accelerator:
  - Use work-sharing directives
  - Specify level of parallelization
- Host is responsible for:
  - Allocating memory space on accelerator
  - Initiating data transfers
  - Lauching computations
  - Waiting for completion
  - Deallocating memory space

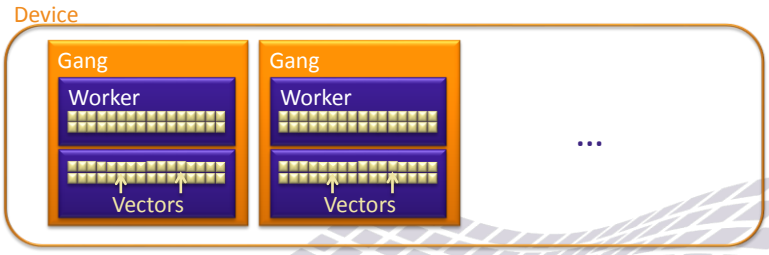
OpenACC Programming Overview





OpenACC Execution Model

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



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
```


Work Management: 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

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

Code executed on the hardware accelerator

Parallel Construct: Gangs and Workers

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

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

- 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

If Clause

- Available on *parallel* and *kernels* constructs
- The compiler generates two copies:
  - One to be executed on the host
  - One to be executed on the accelerator
- 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

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

```
$!acc kernels if(cond)

  DO i=1,n
    …
  END DO
  …
$!acc end kernels
```


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

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

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

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

    A(i) = A(i-1)
END DO
```


Data Independence

- 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

```
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

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

Iterations Mapping

```
#pragma acc loop gang(2)
for (i=0; i<n; i++){
#pragma acc loop worker(2)
   for (j=0;j<m;j++){
     iter(j,j);
   }
}
```

Gang 0

```
for (i=0;
i<n/2; i++){
   for
(j=0;j<m;j++){
iter(i,j);
}
}
```

Gang 1

```
for (i=n/2+1;
i<n; i++){
   for
(j=0;j<m;j++){
     iter(i,j);
}
}
```

Gang 0, Worker 0

```
for (i=0;
i<n/2; i++){
   for
(j=0;j<m;j+=2){
     iter(i,j);
}
}
```

Gang 0, Worker 1

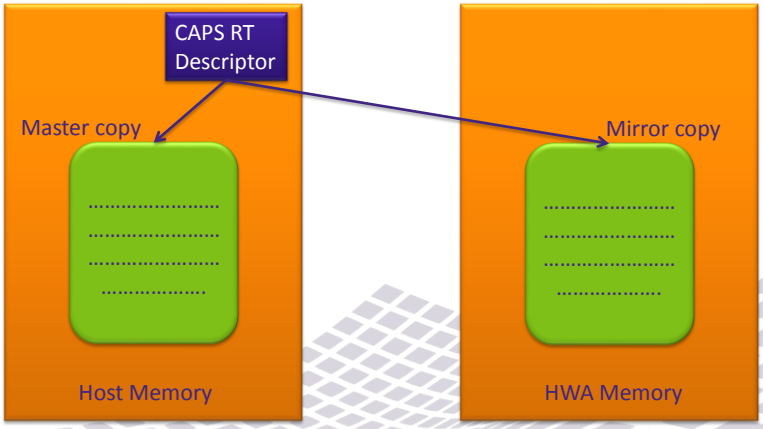
```
for (i=0;
i<n/2; i++){
   for
(j=1;j<m;j+=2){
     iter(i,j);
}
}
```

. . . . . .

Distribution scheme is compiler dependant (here simplified scheme)

Data Storage

- 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

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

Data constructs define data behavior for the duration of the code region

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

```
INTEGER::d_val;

!$acc data, create (d_val)
  …
  !$acc kernels
  d_val = 1;
  …
  !$acc end kernels
  …
  !$acc kernels
  d_val = 5;
  …
  !$acc end kernels
!$acc end data
```

Allocation of val on the device

Deallocation of val on the device

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

```
float A[m][n];
init(A);

#pragma acc data, copy (A[0:n*m])
{
  …
  #pragma acc kernels
  for(i=0; i < n; i++) {
    for(j=0; j < m; j++) {
      A[j][i] = A[j][i] + 1.0;
      …
    }
  }
  …
}
```

Allocation of array A on the device and transfer of the data from the host to the device

Transfer of the data of array A from the device to the host and deallocation on the device

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
- CAPS Runtime will find and use the data on device

```
#pragma acc data, copy (A[0:n*m])
{
   …
   #pragma acc data, present (A)
   {
      …
   }
}
```

```
$!acc data, copy (A(1:n,1:m))

   …
   $!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)
{
  …
  for(i=0; i < n; i++) {
    for(j=0; j < m; j++) {
      A[j][i] = A[j][i] + 1.0;
      …
    }
  }
  …
}
```

A is already allocated	A is not allocated
Finds the device address of A	Allocate A
	Release A

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

- *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)
  {
    …
  }
}
```

Data Management: Default Behavior

- CAPS Compilers are 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

Conclusion

Conclusion

- CAPS Compilers support two sets of directives to use accelerators:
  - OpenHMPP
  - OpenACC
- 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 Compilers support:
    - Nvidia Tesla GPUs
    - AMD
    - X86 Intel Phi

CAPS Compilers Advanced Features

- Library integration directives
  - Needed for a “single source many-core code” approach
- Tracing Open performance APIs
  - Allows to use tracing third party tools
- Tuning directives
  - Loops nest transformations
- External and native functions
  - Allows to include CUDA code into kernels
- Multiple devices management
  - Data collection / map operation
- And many more features
  - Loop transformations directives for kernel tuning
  - buffer mode, UVA support, …

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