HMPP Directive Set

Hybridize your applications

Agenda

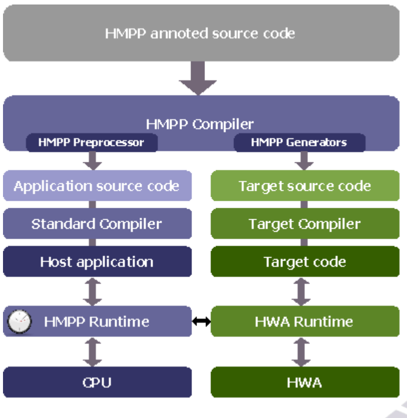
- HMPP Concepts and Overview
- Starting with HMPP
- HMPP Runtime
- HMPP Toolchain
- RPC Sequence
- Advanced Transfer Policy
- Data Storage Policy
- Regions

CAPS

HMPP
Concepts and Overview

What does HMPP offer?

CAPS

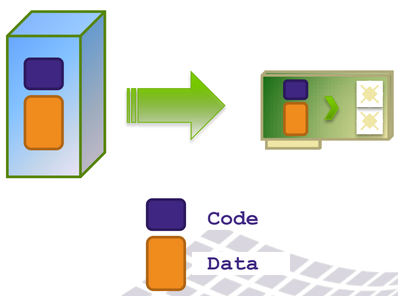


- HMPP is a glue between general purpose programming and hardware accelerators
  - Abstract the architecture and keep the application portable
  - Manage data transfers
  - Ensure application interoperability & resource management
  - Adapt to platform configuration for easy deployment

Offload Computations

CAPS

- HMPP enables to launch computations on hardware accelerators
  - Data need to be transferred
  - Code is executed on the accelerator



How do I do that?

CAPS

- HMPP 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
  - Choose when/how you want to start data transfers

- Just use HMPP directives!
  - The candidate function has to be **insulated** and **offloaded** on the remote hardware accelerator
  - HMPP does it for you!

Basic Concepts

- Pure function calls are offloaded on the accelerator: codelets
  - 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 (C)

- A single line HMPP directive is:

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

- Long directives can be continued on multiple lines

```
#pragma hmpp myLabel command …     &
#pragma hmpp     &              …  &
#pragma hmpp     &              …
…
```

General Syntax of the Directives (Fortran)

CAPS

- Any form of comment (F77, F90,...) starting with $hmpp

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

- Long directives can be continued on multiple lines

```
!$hmpp myLabel command …    &
!$hmpp    &                …    &
!$hmpp    &                …
…
```


General Syntax of the Directives

CAPS

- 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 HMPP directive

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

- Directives are case-insensitive

CAPS

Starting with HMPP

How to execute code on the accelerator?

CAPS

- Think codelet
  - The candidate function has to be **insulated** and **offloaded** on the remote hardware accelerator -> this is the **codelet**
  - The HMPP directives provide the user with codelet control
- There are two essential HMPP directives to know:
  - **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

- The CODELET clause is placed just before the function

```
#pragma hmpp myLabel codelet …
void myFunc(…){
    …
}
```

- The CALLSITE clause is placed just before the call statement

```
…
#pragma hmpp myLabel callsite
myFunc(…);
…
```


Ground Rules

- An HMPP 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/CALLSITE Example

CAPS

- Multiple CALLSITEs:

```
!$hmpp myCall codelet, target=CUDA, ...
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)
    ...
    !$hmpp myCall callsite, ...
    CALL myFunc(10000,X,Y)
    ...
    !$hmpp myCall callsite, ...
    CALL myFunc(10000,Y,Z)
    ...
END PROGRAM
```


How do I choose the Accelerator?

CAPS

- HMPP can address several HWAs:
  - CUDA
  - OpenCL
- But how can I specialize my codelet?
  - See the TARGET attribute

Codelet Directive: the TARGET Attribute

CAPS

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

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


Referencing Arguments in the Directives

CAPS

- HMPP 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[*]
    ```

Basic Transfer Policy

- HMPP, let you apply an incremental method in your porting process
  - Use the attribute transfer to specify the policy of your arguments

```
#pragma hmpp myLabel codelet, target=CUDA, &
#pragma hmpp & args[*].transfer=atcall
```

  - ATCALL means that the arguments will be transferred automatically at every callsite
  - By default:
    - Scalar will be transferred only in input
    - Arrays will be transferred as input and output
- Be careful, if you forget to put the transfer policy :
  - HMPP will use, by default, the HMPP2 policy (Deprecated)
  - To keep for the retro-compatibility with HMPP-2

Referencing Arguments in the Directives

- Example:

```
#pragma hmpp myLabel codelet, args[0-1;2;v].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;
}
```

- Which is equivalent to:

```
#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;
}
```

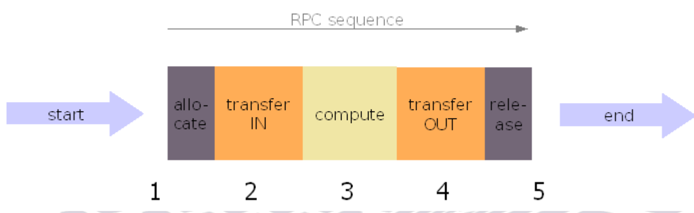
Referencing Arguments in the Directives

- Attention! Arguments are referred upon their names in the CODELET's definition, even in the execution context
  - Arguments' properties are global, not restricted to where they are written
- If you are far from the CODELET's definition, you'd prefer to refer arguments by their ranks to prevent confusion

HMPP Runtime

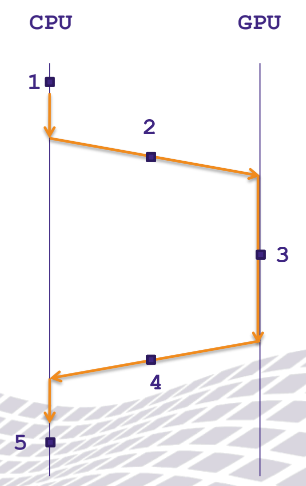
CALLSITE: Full Remote Procedure Call sequence

- By default, a CALLSITE directive implements the whole RPC sequence
  - RPC = Remote Procedure Call
- An RPC sequence consists in 5 steps:
  - (1) Allocate the HWA and the memory
  - (2) Transfer the input data: CPU => HWA
  - (3) Compute
  - (4) Transfer the output data: HWA => CPU
  - (5) Release the HWA and the memory



Full RPC sequence

- At this time, RPC steps are considered as synchronous
  - Each step is blocking
  - Application wait for the step completion



HMPP Fallback

- What HMPP does if a codelet file is not present at run time?
  - If the hardware accelerator is unavailable?
  - If data allocation fails?
  - If the data transfer fails?
  - If the computation raises an error?
- The native CPU version is used as a fallback
  - This behavior can be inhibited with HMPPRT_NO_FALLBACK
- Fallback is only possible until step 3 (computation) in synchronous mode
  - In asynchronous mode, a computation failure leads to the end of the application

HMPP Environment

- Setting up environment variables
  - Prevent HMPP from default fallback behavior

```
$ export HMPPRT_NO_FALLBACK=<errorCode>
```

  - Force permanent verbosity

```
$ export HMPPRT_LOG_LEVEL=INFO
```

  - Add a new codelet repository

```
$ export HMPPRT_PATH=<pathThatContainsHmgAndHmc>:$HMPPRT_PATH
```

CODELET Directive: multiple TARGETs

- Specify for which HWA a specialized version must be generated
  - No target mean no specialized code to generate

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

- Multiple targets are possible (selected in order)
  - Then the following is the fallback of the previous one

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

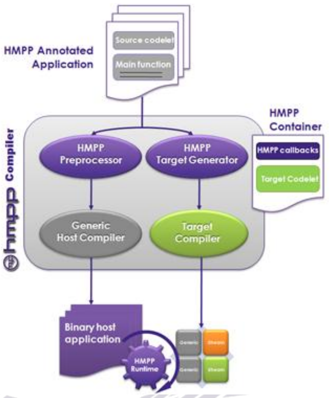

HMPP Toolchain

HMPP Compilation Process

- HMPP applications consist of:
  - The host application (binary)
    - Use your common compiler (gcc, icc, ifort…)
  - The codelets
    - Let the HMPP Code generator do it for you

HMPP Complete Application Compilation

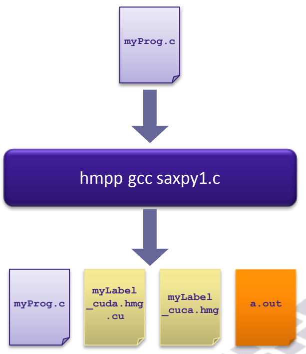
- HMPP drives all compilation passes
  - Host application compilation
    - HMPP Runtime is linked to the host part of the application
  - Codelet production
    - Target code is produced
    - A dynamic library is built

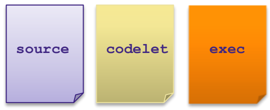



```
$ hmpp gcc myProgram.c
```

HMPP Generated Files

- Compiling generated codelet files
- All files are available for lecture/modification
    - Codelets and main source files





<codeletName>_<target>.hmc.<ext>

mysaxpy1_cuda.hmc.cu

sgemm1_opencl.hmc

...

HMPP Compilation Passes

- How do I split the building process of my application?
    - Use HMPP toolchain to generate separately
        - Host application
        - Codelets
- The compilation process can be split into:
    - Host application compilation
    - HMPP Codelet compilation (generate + compile)
    - HMPP Codelet compilation (generate only)
    - HMPP Codelet compilation (compile only)

RPC sequence

HMPP Directives Overview

- CODELET : Specialize a subroutine
- CALLSITE : Specialize a call statement

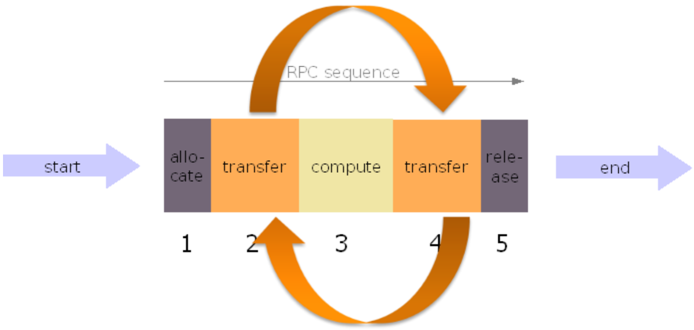
HMPP Directives Overview

- CODELET : Specialize a subroutine
- CALLSITE : Specialize a call statement
- SYNCHRONIZE : Wait for completion of the callsite
- ACQUIRE : Explicit the HWA acquisition
- RELEASE : Release HWA and its memory
- ALLOCATE : Allocate memory for the grouplet arguments
- FREE : Free memory for the grouplet arguments
- ADVANCEDLOAD : Explicit data transfer CPU -> HWA
- DELEGATEDSTORE: Explicit data transfer HWA -> CPU
- GROUP : Define a group of codelets
- RESIDENT : Declare a resident (global) variable
- MAP : Map arguments together

  » Directives in green are declarative
  » Directives in orange are operational

Reminder

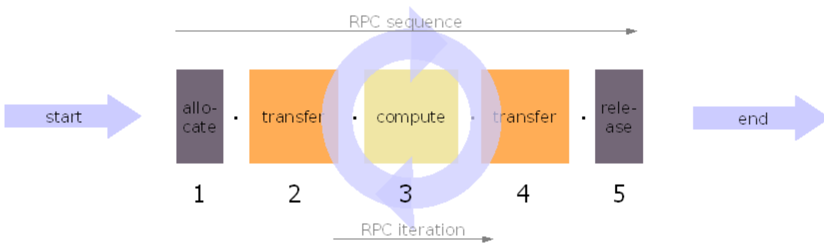
- A standalone CALLSITE directive implements the whole RPC sequence



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

Splitting the RPC sequence

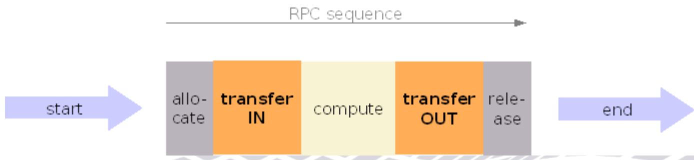
- To optimize HWA usage according to your application algorithm, this RPC sequence can be split into several parts



Advanced Transfer Policy

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)



Transfer Policy Rules

- The default IO rules are
  - 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
    - So we recommand to use the fortran INTENT
- Selecting the right IO is important to reduce data transfer costs

```
#pragma hmpp myLabel codelet, target=CUDA, &
#pragma hmpp & args[*].transfer=atcall, &
#pragma hmpp & args[inputVar].io=in, args[outputVar].io=out
Void myfunc (int size, float* inputVar, float* outputVar)
```

Transfer Policy

- The transfer policy specifies when the data must be transferred to and from the HWA

- HMPP2 (LEGACY)
  - The original (deprecated) HMPP 2.x policy
- MANUAL
  - A simple policy in which the data transfers are explicitly specified by the user via dedicated directives
- ATCALL
  - A simple policy in which the data are systematically transferred at CALLSITE
  - ATFIRSTCALL
    - The data is transferred at the first CALLSITE occurrence

Transfer Policy (1)

- How to avoid transfers with ATCALL policy
  - By default an array has ATCALL transfer policy
  - But in C, an array declared as *const* is just transferred as input

```
#pragma hmpp myLabel codelet, target=CUDA, &
#pragma hmpp & args[*].transfer=atcall, &
#pragma hmpp & args[outputVar].io=out
Void myfunc (int size, const float* inputVar, float* outputVar)
```

  - Here, *inputVar* will be only transfered as an input data

- To transfer data only once at the first callsite
  - Use the transfer policy ATFIRSTCALL
  - Useful for constant data (coefficient, constant sizes, …)

```
#pragma hmpp myLabel codelet, target=CUDA, &
#pragma hmpp & args[1,2].transfer=atcall, &
#pragma hmpp & args[0].transfer=atfirstcall, &
#pragma hmpp & args[outputVar].io=out
Void myfunc (int size, const float* inputVar, float* outputVar)
```

Transfer Policy (2)

CAPS

- To improve the transfer performance
  - MANUAL transfer policy
  - Manually manage the data transfer

```
#pragma hmpp myLabel codelet, target=CUDA, &
#pragma hmpp & args[1,2].transfer=manual, &
#pragma hmpp & args[0].transfer=atfirstcall
Void myfunc (int size, float* inputVar, float* outputVar)
```

- To transfer data from/to the GPU only when the application needs it
  - ADVANCEDLOAD directives to upload data to the GPU
  - DELEGATEDSTORE directives to download data from the GPU

Transfer Policy: ATCALL

CAPS

- The argument is ALWAYS transferred at CALLSITE
  - At input or output depending of its I/O status
- A very safe policy
  - But potentially inefficient
  - Arguments may be transferred to / back to GPU even if it is useless
- ADVANCEDLOAD or DELEGATEDSTORE are possible
  - But maybe useless if already done at the call

Transfer Policy: MANUAL

- No implicit transfers at CALLSITE
  - Never
- All transfers must be specified by the user
  - With ADVANCEDLOAD or DELEGATEDSTORE
- The resulting code is potentially incorrect
  - If the user forgets a transfer
  - But cheaper: you get exactly what you ask for
    - No more! No less!
    - Unlike HMPP2 policy (legacy)

Manually Managing Data

ACQUIRE Directive

- Step #1 in RPC sequence
- Set the HWA for the application
- Syntax:

```
#pragma hmpp myLabel acquire[ , … ]

!$hmpp myLabel acquire[ , … ]
```

- If not present, executed implicitly before
  - The ALLOCATE directive
- Possible to select a logical device using the device attribute

```
#pragma hmpp myLabel acquire, device="expr"
```


RELEASE directive (1)

- Release the HWA and free the used memory
- Syntax:

```
#pragma hmpp myLabel release

!$hmpp myLabel release
```

- Should be the last directive executed for a given label
  - This directive is implicitly called at the end of application

ALLOCATE directive

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

```
#pragma hmpp myLabel allocate [ , … ]

!$hmpp myLabel allocate [ , … ]
```

- Possible to allocate specified arguments

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

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

ALLOCATE directive: the SIZE attribute (1)

- Allocation of array arguments requires to evaluate their dimensions
- If ATCALL policy, their value is obtained by combining the declaration in the CODELET with the argument values in the CALLSITE
- Example:

```
#pragma hmpp myFunc1 codelet, ...
void myFunc( int n, int A[n*2]) {       // n*2
    ...
}

...
#pragma hmpp myFunc1 callsite
myFunc(sz+1,X) ;                        // n==sz+1
```

» The expression (sz+1)*2 is evaluated by the ALLOCATE

ALLOCATE directive: the SIZE attribute (2)

- If independent ALLOCATE directive, the size expression does not exist in the allocate context
  - Use an attribute size to specify an alternative expression
- Syntax:

```
#pragma hmpp Label allocate, args[…], size={ expr , … }

!$hmpp       Label allocate, args[…], size={ expr , … }
```

- Specify one expression per dimension
  - The value evaluated by the SIZE attribute is supposed to be equal to the dimension at the callsite

ALLOCATE directive: the SIZE attribute (3)

- A full example:

```
#pragma hmpp myFunc1 codelet, ...
void myFunc( int n, int A[n*2]) {
  ...
}

...
sz = NB+4 ;
#pragma hmpp myFunc1 allocate, args[A], size={(sz+1)*2}
...
#pragma hmpp myFunc1 callsite
myFunc(sz+1,X) ;
```

- Warning: the size expression is inserted in the instrumented code “as is”
  - Expression is not preprocessed (no macros)

FREE Directive

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

```
#pragma hmpp myLabel free[ , … ]

!$hmpp myLabel free[ , … ]
```

- If not present, executed implicitly before
  - RELEASE directive

- 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

ADVANCEDLOAD directive (1)

- Transfer arguments from CPU to HWA
- Syntax:

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

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

- The standalone attribute ARGS specifies the arguments to be transferred
- It can be performed safely several times on the same arguments

ADVANCEDLOAD: the HOSTDATA attribute (2)

- Makes the correspondance between the arguments and the data on the host
- Syntax:

```
#pragma hmpp label advancedload, args[A], hostdata="expression"
```

- Warning: the expression is used verbatim in the code and is not preprocessed (no macros)
- NB: used for buffer storage policy

ADVANCEDLOAD directive (2)

- Example:

```
#pragma hmpp myFunc codelet, target=CUDA, args[*].transfer=manual
void myFunc( int n, int A[n], int B[n], int C[n])
{
  ...
}

main(){

...

#pragma hmpp myFunc advancedload, args[n;A;B;C], &
#pragma hmpp &            args[n]="size", &
#pragma hmpp &            args[A]="X"
#pragma hmpp &            args[B]="Y"
#pragma hmpp &            args[C]="Z"
...

#pragma hmpp myFunc callsite
myFunc(size,X,Y,Z) ;

...
}
```

ADVANCEDLOAD: ASYNCHRONOUS attribute (1)

- By default, the ADVANCEDLOAD directive is blocking until all specified arguments are transferred
- The ASYNCHRONOUS attribute makes the transfer non-blocking
  - The transferred variable should remain alive (and unchanged) until the CALLSITE
- Some targets may not implement asynchronous transfers
  - The transfer could be implemented in a blocking way

ADVANCEDLOAD: ASYNCHRONOUS attribute (2)

- Example:

```
#pragma hmpp myFunc codelet, target=CUDA, args[*].transfer=manual
void myFunc( int n, int A[n], int B[n], int C[n])
{
  ...
}

main(){

...

#pragma hmpp myFunc advancedload, args[n;A;B;C], …, asynchronous

...
// The transfer of n, A and B is currently being performed
...
#pragma hmpp myFunc callsite
myFunc(size,X,Y,Z) ;

...
}
```

The DELEGATEDSTORE directive

- Transfer arguments from HWA to HOST
  - RPC sequence step #4
- Syntax:

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

- The standalone attribute “args[…]” specifies the arguments to be transferred
- Arguments with a DELEGATEDSTORE are not implicitly transferred in the CALLSITE
- May need the HOSTDATA attribute if the arguments is only OUT

Asynchronous CALLSITEs

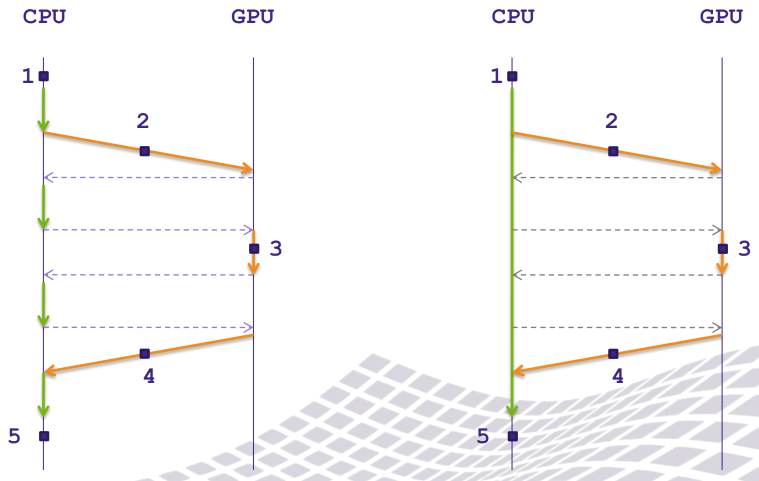
- The ASYNCHRONOUS attribute make the CALLSITE non-blocking
- A SYNCHRONIZE directive is mandatory to wait for the CALLSITE completion
- All the OUT/INOUT arguments with a ATCALL policy will tbe transferred at the SYNCHRONIZE directive
- Syntax:

```
#pragma hmpp label callsite, asynchronous, ...
...
#pragma hmpp label synchronize
```

Split RPC sequence

CAPS

- Synchronous
- Asynchronous



CAPS

Data Management: Storage Policy

Introduction

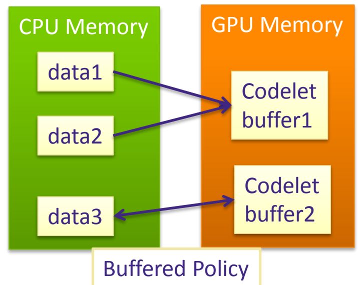
- The Storage and Transfer policies are two property of each Codelet argument
  - They specify how they are managed by HMPP and by the user
- The Storage Policy
  - Specifies *where* and *how* the data must be allocated on the HWA
  - Two types: Buffer or Mirror
- The Transfer Policy
  - Specifies *when* and *how* the data must be transfered from or onto the HWA
  - ATCALL, MANUAL, AUTO, ATFIRSTCALL or HMPP2

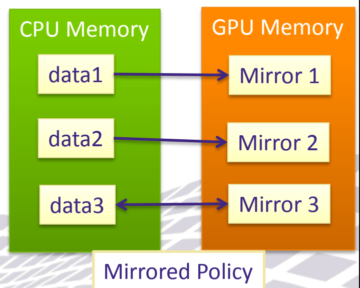
Storage Policy

- Buffered Argument or simply Buffer
  - A single static buffer is associated to each codelet argument
  - The buffer is pre-allocated by the ALLOCATE directive
  - The buffer is identified by the argument (and codelet) name
  - This is the default policy
- Mirrored Data or simply Mirror
  - An area of memory on the host is mirrored on the accelerator
  - The mirror is identified by its address on the host

Storage Policy (2)

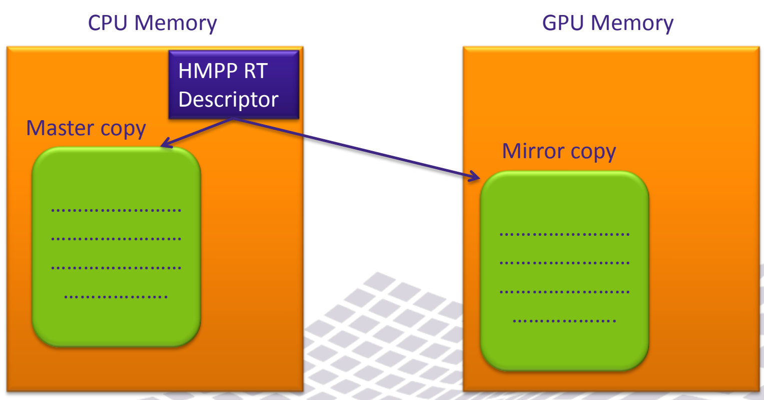
- Allocate a buffer on GPU for each codelet argument (HMPP 2.0 & 3.0)
  - Many CPU memory zones to one GPU buffer
  - Save GPU space
  - May increase the numbers of CPU-GPU transfers
  - Static management == user control == many directives
- Use of mirroring (HMPP 3.0)
  - One CPU memory zone for one GPU buffer
  - More device memory used
  - A bit more runtime overheads
  - Simplifies data managements == less directives
- Using one mode or the other depends on the data accesses





Data Mirroring

- Mirroring duplicate a CPU memory block in GPU memory
  - Mirror identifier is CPU memory block address
  - Only one mirror per CPU block
  - Users ensure consistency of copies via **`advancedload`** and **`delegatedstore`** directives



Data Management: Buffering vs Mirroring

CAPS

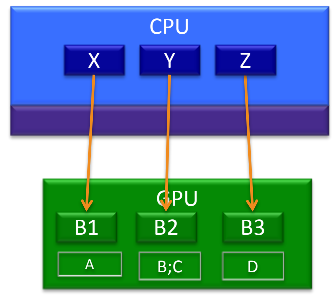
Buffering

Declaration

```
#pragma hmpp <g> group, …
#pragma hmpp <g> map args(f1::B;f2::C]

hmpp <g> f1 codelet, …
void C1(float A[n], float B[n]){}

#pragma hmpp <g> f2 codelet, …
void C2(float C[n], float D[n]){}
```

Execution

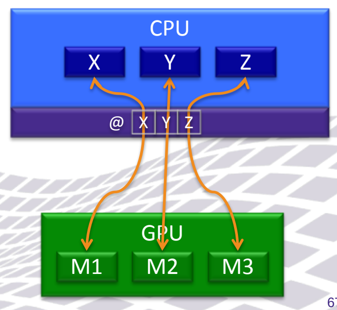
```
#pragma hmpp <g> f1 callsite, …
call C1(X, Y)

#pragma hmpp <g> f2 callsite, …
void C2(Y, Z)
```



Mirroring

Declaration

```
#pragma hmpp <g> group, …
#pragma hmpp <g> f1 codelet, args[*].mirror, …
void C1(float A[n], float B[n]){}

#pragma hmpp <g> f2 codelet, args[*].mirror, …
void C2(float C[n], float D[n]){}
```



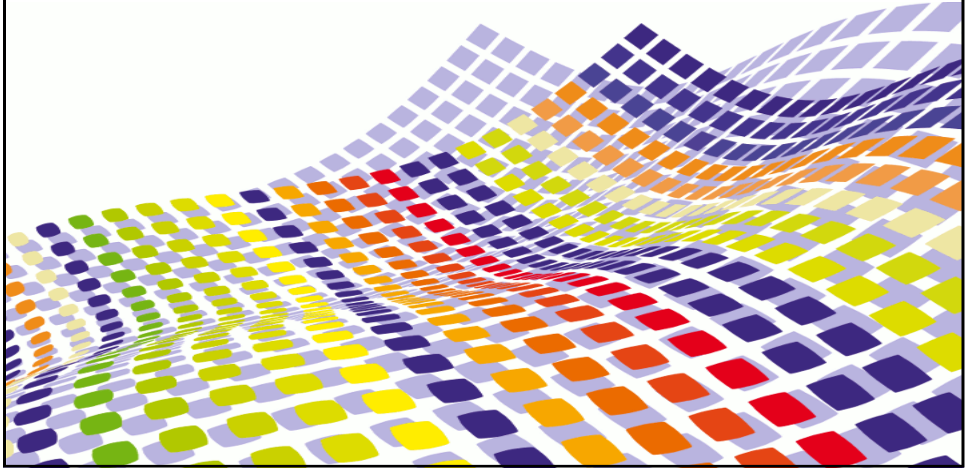
Independance of Storage and Transfer Policies

CAPS

- Both policies are independent
  - You can have a MANUAL & Buffer, ATCALL & Mirror, …
  - This is possible because the sole purpose of the Storage Policy is to find the source and destination of the transfer
    - Transfer Policy decides when they occur

CAPS

The Mirrored Storage Policy



CODELET Directive: the MIRROR clause

CAPS

- Use the MIRROR clause to change the storage policy of one or more arguments from Buffered to Mirrored
- Added to the CODELET directive

```
!$hmpp foobar codelet, args[A,B].mirror
SUBROUTINE (n,A,B)
  INTEGER , INTENT(IN)    :: n
  REAL    , INTENT(INOUT) :: A,B
  ...
END SUBROUTINE foobar
```

Mirroring: Directives ALLOCATE and FREE

CAPS

- Use ALLOCATE to allocate HWA memory for the mirror
  - A SIZE clause must be specified
- Use FREE to de-allocate HWA memory
  - A mirror can be temporarily released to save memory

```
int  A[100][1000] ;
...
for(i=0;i<100;i++) {
  #pragma hmpp foo allocate, data["A[i]"], data["A[i]"].size={1000}
  #pragma hmpp foo advancedload, data["A[i]"]
  ...
  #pragma hmpp foo free, data["A[i]"]
}
…
```

- The data[...] field refers to one or more mirrors
  - The expression between [ ] should be a list of
    - Addresses in C
    - Expressions with an address in Fortran
      - beware of temporary sub-arrays
  - Anything more complex than a single identifier should be "quoted"

Mirrors: CALLSITE

CAPS

- Just ensure that
  - The MIRROR clause is specified on the CODELET
  - And pass an address for which a mirror is currently allocated

```
#pragma hmpp foo codelet, args[X].mirror
void foo(int n, float *X)
{
 ...
}

int  A[1000][1000] ;
#pragma hmpp foo allocate, data[A], size={1000,1000}
...
#pragma hmpp foo callsite
foo(1000,A) ;
...
```

  - The transfer policy rules are not changed
    - e.g. with ATCALL, the whole mirror may be transferred IN or OUT

CAPS

HMPP Regions

CAPS

How to offload portions of code ?

- Sometimes, not a function to offload but a region of code

```
int main()
{

//some initialisations

  int i;
  for( i = 0; i < n1; ++i )
     r[i] = a[i]*2.0f;

  for( i = 0; i < n2; ++i )
     b[i] = b[i]*2.0f;

//other computations
}
```

- In this case, just want to execute on the accelerator the 2 loops

HMPP Regions

- Syntax :
  - In C :

```
#pragma hmpp region
{
  //Block to offload
}
```

  - In Fortran

```
!$hmpp region

  !!Block to offload

!$hmpp endregion
```

- Like codelets, no IO functions inside the region
  - No GOTO to jump inside or outside the region

HMPP Regions

- Same attributes as codelet and callsite :
  - Group and label
  - Target
  - IO, transfer policy, storage policy
  - Size of arguments
  - …
- Attribute private to make private a variable to each thread
- Same way to manage the RPC sequence as codelets
  - Acquire, Release
  - Allocate, Free
  - Advanceload, Delegatedstore
  - Synchronize (asynchronous region)

HMPP Regions in C

- C example

```
int main()
{

//some initialisations and computations

#pragma hmpp region, target=CUDA
{
  int i;
  for( i = 0; i < n1; ++i )
     r[i] = a[i]*2.0f;

  for( i = 0; i < n2; ++i )
     b[i] = b[i]*2.0f;
}

//other computations
}
```


HMPP Regions in Fortran

- Fortran example

```
PROGRAM test
    ! some initialisations

!$hmpp mylabel region, target=CUDA
    DO i=1,n1
        r(i)=a(i)*2.0
    ENDDO
    DO i=1,n2
        b(i)=b(i)*2.0
    ENDDO
!hmpp mylabel endregion

    !other computations
END PROGRAM
```

HMPP Region : constraints

CAPS

- Some constraints on the data and loops
  - No IO
  - (C) No return, break or continue in the region
  - (Fortran) stop, cycle or exit in the region
  - No goto to jump inside or outside the region
- Also some restrictions on the region
  - Regions cannot be nested;
  - Asynchronous region must have at least a label;
  - No HMPP directives are allowed inside the region, only HMPPCG

HMPP Region : IO Report

CAPS

- HMPP provides an option, "`--io-report`" that returns an IO report at compilation time
- Example

```
#pragma hmpp <group> foo region
{
   int i;
   for( i = 0; i < n; ++i )
      r[i] = a[i]*2.0f;

   for( i = 0; i < n; ++i )
      b[i] = b[i]*2.0f;
}
```

- Compilation

```
$ hmpp --io-report gcc simple_region-000.c -o test.exe
In GROUP 'group'
REGION 'foo' at simple_region-000.c:25, function
'__hmpp_region__group__foo'
      Parameter 'n' has intent IN
      Parameter 'a' has intent INOUT
      Parameter 'b' has intent INOUT
      Parameter 'r' has intent INOUT
```

HMPP advanced features

CAPS

- Multiple devices management
  - Data collection / map operation
- Library integration directives
  - Needed for a “single source many-core code” approach
- Loop transformations directives for kernel tuning
  - Tuning is very target machine dependent
- Open performance APIs
  - Tracing
  - Auto-tuning (H2 2012)
- And many more features
  - Native functions, buffer mode, codelets, …

What’s next in HMPP 3 ?

CAPS

- Input code
  - C
  - Fortran 90
  - C++ (API)
- Targeted accelerators
  - Nvidia
  - AMD
  - Intel Xeon Phi
- Targeted Operating Systems
  - Linux
  - Windows

Accelerator Programming Model

Parallelization

CAPS

Directive-based programming

GPGPU

Manycore programming

Hybrid Manycore Programming

HPC community

Petaflops

Parallel computing

HPC open standard

Multicore programming

Exaflops

NVIDIA Cuda

Code speedup

Hardware accelerators programming

High Performance Computing

Parallel programming interface

Massively parallel

Open CL

CAPS

open hmpp

http://www.caps-entreprise.com
http://twitter.com/CAPSentreprise

http://www.openhmpp.org