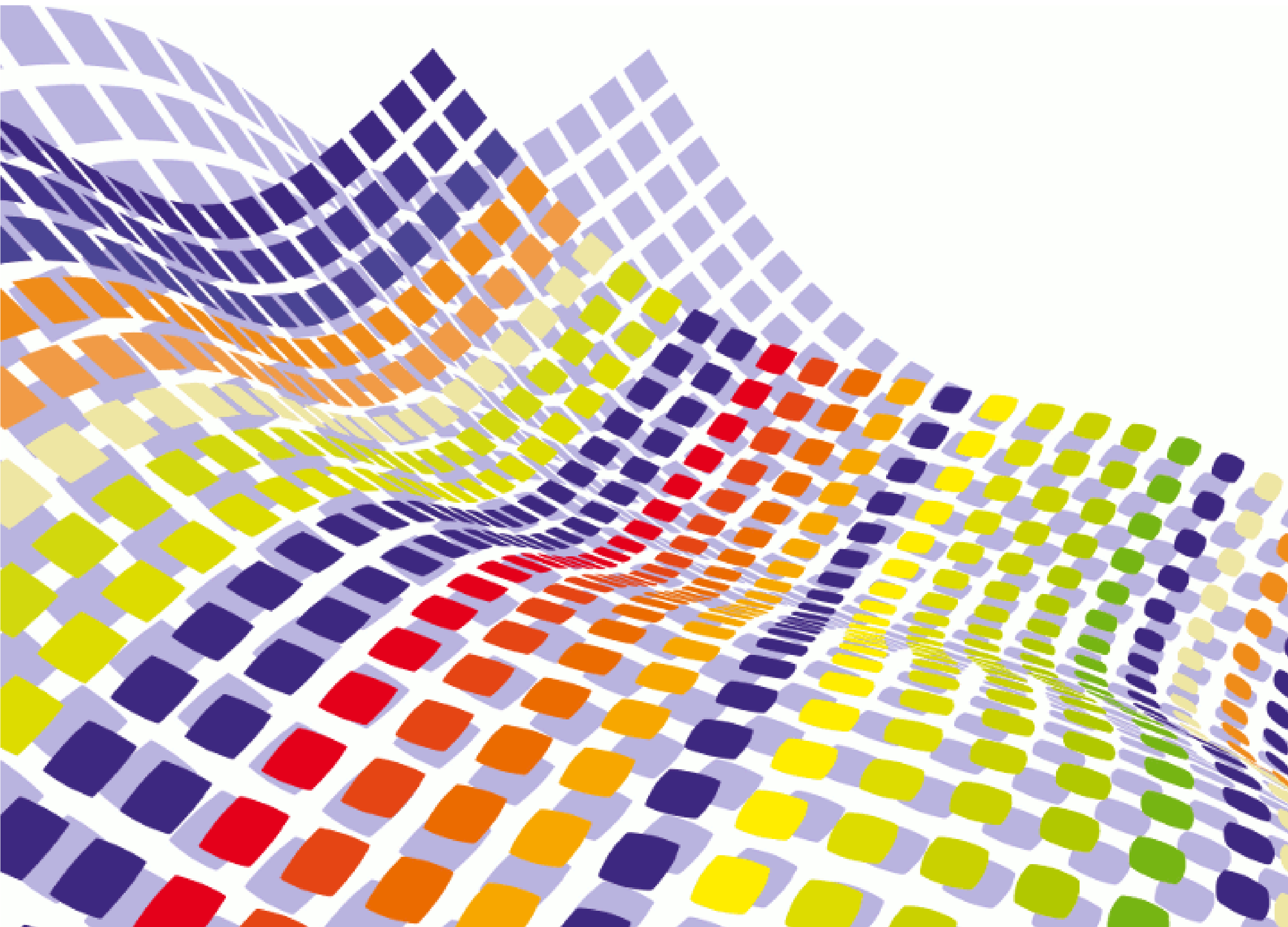
HMPP Directives

HMPP Workbench 3.0



Headquarters – France
Immeuble CAP Nord
4A Allée Marie Berhaut
35000 Rennes
France

Tel.: +33 (0)2 22 51 16 00
Fax: +33 (0)2 23 20 16 43

info@caps-entreprise.com

N° d'agrément formation :
53 35 08397 35

CAPS – USA
4701 Patrick Drive Bldg 12
Santa Clara
CA 95054

Tel.: +1 408 550 2887 x70

usa@caps-entreprise.com

CAPS – CHINA
Suite E2, 30/F
JuneYao International Plaza
789, Zhaojiabang Road,
Shanghai 200032

Tel.: +86 21 3363 0057
Fax: +86 21 3363 0067

apac@caps-entreprise.com

Visit our website: http://www.caps-entreprise.com

SUMMARY

Revisions history

Version	Date	Writer	Modified pages	Revision object
V2.4.0	15/11/2010	CAPS entreprise	All §4.5.1, §4.7.4, §4.7.4 §0	Restructuration of the documentation (from version 2.3.5) Addition of the automatic data transfer mode for codelet's argument Automatic detection of Inputs and Outputs in HMPP region
V2.4.1	24/12/2010	CAPS entreprise	§3.4.1 §4.1.2	Correction on Listing 8 Clarification on parameter passing convention
V2.4.2	28/01/2011	CAPS entreprise		Addition of Intel FORTRAN Compiler for Windows
V2.4.3	04/03/2011	CAPS entreprise		CAPS entreprise
V2.4.4	24/03/2011	CAPS entreprise		Addition of Absoft's Pro Fortran V11 Compiler for Windows
V2.5.0	16/06/2011	CAPS entreprise	§4.7.4 §4.11, §7.2.5, §7.2.6 §7.2.9	Additional information on automatic data transfer Addition of external and native functions __HMPP predefined macro
V2.5.1	07/07/2011	CAPS entreprise		Typography corrections
V2.5.2	06/06/2011	CAPS entreprise	§4.6.2 §4.6.3	Add asynchronous clause for delegatedstore directive Addition of asynchronous data transfers
V3.0.0	16/12/2011	CAPS entreprise	§4.5.5 §4.5.8 §4.8.4 §4.9 §4.7	New HMPP directives: • acquire • free • Addition of data mirroring • Multiple devices management Data transfer policies
V3.0.1	12/01/2012	CAPS entreprise	§5.1.4	Supported atomic functions
V3.0.2	20/01/2012	CAPS entreprise		Typography corrections
V3.0.3	25/01/2012	CAPS entreprise		Link reference corrections

1. Introduction

Warning:

HMPP 3.0.0 only supports the CUDA target for codelet generation. The OpenCL target will be supported current Q1 2012 (currently available in HMPP 2.5.x).

For convenience, the text referencing the two targets was left in state. In HMPP 3.0.0, only the CUDA target is to be considered.

The Hybrid Multicore Parallel Programming workbench (HMPP) provides developers with a set of tools dedicated to build parallel hybrid applications running on manycore systems. These architectures combine general purpose cores with hardware accelerators (HWAs) such as GPUs or SIMD computing units.

HMPP allows the programmer to write hardware independent applications where hardware specific codes are dissociated from the legacy code as additional software plug-ins. Contrary to applications that have been explicitly written for a target architecture, HMPP produces applications that execute on various hardware platform configurations, whether a HWA is present or not. Hardware-accelerated versions of functions are executed if the accelerator is present and available, otherwise their native versions are run.

The present document introduces the main HMPP concepts and describes the HMPP directives. This document comes in addition to the following manuals:

- `HMPP Basics` (`[R1]`). This document introduces the main HMPP concepts.
- `HMPP Codelet Generator Directives, Reference Manual ([R3])`. This manual describes how to enhance your codelet generation by using HMPPCG directives. An HMPP preprocessor allowing users to factorize HMPP directives is also described;
- `HMPP Linux Manual ([R5])`. This manual describes how to compile and run your application on Linux platforms. It also introduces the compilers and Operating Systems supported;
- `HMPP License Installation Guide ([R6])`. This manual presents the procedure to set the HMPP license on your system.

The remainder of this document is organized as follow:

- Chapter 3 presents the main concepts of HMPP,
- Chapter 4 introduces the HMPP directives,
- Chapter 5 describes the supported languages,
- Chapter 6 presents the HMPP codelets Generators
- Chapter 7 is dedicated to the compilation flow process.

A glossary can be found at the end of the document.

It should be noted that most of the examples provided in this document are based on the CUDA backend generator for historical reasons. The functionalities offered by HMPP are the same for all the backend generators marketed by CAPS entreprise.

When necessary, CAPS will specify in the text if a feature is dependent of a given material.

2. HMPP Overview

2.1. HMPP Development Workbench Overview

Based on a set of directives, the HMPP Workbench contains C and FORTRAN compiler drivers, target code generators (CUDA, OPENCL) and a runtime for the execution of parallel hybrid applications.

To accelerate the execution of your application with HMPP, the first step is to identify the parts of the application source code to speed up. Those will become functions called “HMPP codelets” (see section 3.1) using the HMPP directives. The hardware-accelerated versions of the codelet are defined in their specific language i.e. C or FORTRAN and using the same programming model. They are hand-written by the user or automatically produced by the HMPP codelet generators and compiled with the compilers of the HWA vendor.

The HMPP annotated source code is parsed by the HMPP preprocessor to extract the codelets and to translate the HMPP directives into calls to the HMPP runtime. The preprocessed code is then compiled and linked with the HMPP runtime using the host compiler. The HMPP runtime is in charge of managing the concurrent execution of the codelets.

When no HWA implementation of a codelet is found or if the chosen HWA is not available, the HMPP runtime executes the native version instead. So, the execution of an HMPP user’s application is still possible.

Figure 1 shows the general workflow of an HMPP application. The left flow path shows how an annotated codelet is compiled for a given HWA. The right flow path shows the compilation of the rest of the application compiled and run using the C or FORTRAN compiler on the host.

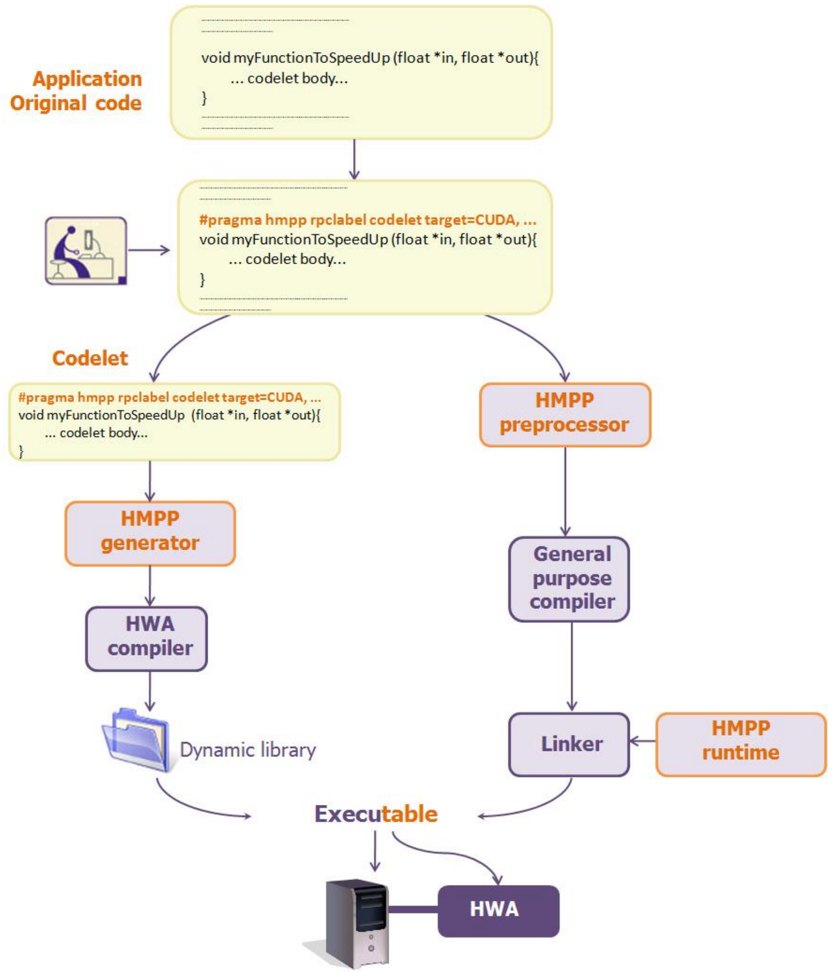

Application Original code
void myFunctionToSpeedUp (float *in, float *out){
... codelet body...
}
#pragma hmpp rpclabel codelet target=CUDA, ...
void myFunctionToSpeedUp (float *in, float *out){
... codelet body...
}
Codelet
#pragma hmpp rpclabel codelet target=CUDA, ...
void myFunctionToSpeedUp (float *in, float *out){
... codelet body...
}
HMPP preprocessor
HMPP generator
General purpose compiler
HWA compiler
Dynamic library
Linker
HMPP runtime
Executable
HWA

Figure 1 - Workflow overview of the HMPP workbench

2.2. HMPP Runtime Overview

The HMPP runtime is the dynamic library in charge of the execution of the remote procedure calls to the HWA. Linked to the application, this library initializes the HWA, allocates memory, relays communications between the host and the HWA and manages the execution of codelets.

2.3. HMPP generators

HMPP workbench provides users with back-end code generators. These code generators are specifically designed to extract the most of data parallelism from your C and FORTRAN kernels and translate them into NVIDIA CUDA or OPENCL (Open Computing Language) allowing to run your applications on various systems.

The code generators marketed by CAPS entreprise are:

- CUDA for NVIDIA GPU systems;
- OPENCL for NVIDIA and AMD ATI Stream GPU systems.

It should be noted that hardware constructors do not offer the same level of functionalities with the OpenCL framework. For the execution of their applications, end-users will pay attention to get the most recently drivers for their HWA in order to take advantage of the state-of-the-art of hardware constructor’s development.

3. HMPP Concept

HMPP is based on the concept of codelets, functions that can be remotely executed on HWAs. The HMPP runtime library is in charge of calling the remote procedure (RPCs) as well as managing resources.

In version 2.0, HMPP introduced the facility of defining a group of codelets allowing the programmer to share data between codelets that are distinct and may run at very different times on the HWA.

As usual with directive-based programming environments, it is important to note that the HMPP development workbench does not check for incorrect usage of the directives. Misuse of the HMPP directives may lead to erroneous results.

3.1. The HMPP Codelet Concept

A codelet is a computational part of a program located in a function inside the application. It takes several scalars and array parameters, performs a computation on these data and returns. The result of the computation is passed by some parameters given by reference (INPUT(inout) in FORTRAN and pointers and arrays in C). The function does not support any return code (it is like a subroutine procedure in FORTRAN and void functions in C). The execution of a codelet is considered as atomic: the execution does not have an identified intermediate state or data. The execution has no side effects.

The transfer of codelet parameters is performed via the HMPP Runtime protocol. The size of all parameters must be known[1] before the transfer of any parameter, and obviously before the codelet execution.

A codelet has the following properties:

1. It is a pure function.
   a. It does not contain static or volatile variable declarations nor refer to any global variables except if these have been declared by a HMPP directive "`resident`" (see chapter 4.8.3 for more details on this subject).
   b. It does not contain any function calls with an invisible body (that cannot be inlined). This includes the use of libraries and system functions such as malloc, printf, ...
   c. Every function call must refer to a static pure function (no function pointers).
2. It does not return any value (`void` function in C or a subroutine in FORTRAN).
3. The number of arguments should be fixed (i.e. no variable number of arguments like `vararg` in C).
4. It is not recursive.
5. Its parameters are assumed to be non-aliased.
6. It does not contain `callsite` directives (i.e. RPC to another codelet) or other HMPP directives.

These properties ensure that a codelet RPC can be remotely executed by a HWA. This RPC and its associated data transfers can be asynchronous.

By default, all the parameters are uploaded to the HWA just before the RPC and downloaded just after its execution has completed.

[1] The scalar arguments and arrays which size is constant and statically evaluable by HMPP do not require the user to specify their size.

Below is an example of a correct codelet:

```
#pragma hmpp testlabel1 codelet, target=CUDA, args[v1].io=out
static void codeletOk(int n, float v1[n], float v2[n], float v3[n]) {
  int i;
  for (i = 0 ; i < n ; i++) {
    v1[i] = v2[i] + v3[i];
  }
}
```

Listing 1 - Codelet definition

The following examples are incorrect codelet definitions or uses:

- ***Use of a global variable in a codelet body:*** since the memory between the HWA and the CPU is not shared, a global variable cannot be used in a codelet.

```
......
float globalVar[SIZE];
......
#pragma hmpp testlabel1 codelet, target=CUDA, args[v1].io=out
static void codeletNotOk(int n, float v1[n], float v2[n], float v3[n]) {
  int i;
  for (i = 0 ; i < n ; i++) {
    v1[i] = v2[i] + v3[i]*globalVar[i];
  }
}
```

Listing 2 - Wrong codelet definition due to the use of a global variable

To fix the error, the global variable needs to be passed as a parameter to the codelet or to be declared as a "`resident`" variable (see chapter 4.8.3 for more details).

- ***Aliasing between parameters***: the following code produces an erroneous result due to the aliasing between v1 and v2 that point to the same caller parameters (see line 18, at the "`callsite`" level). On the device, the parameters are in independent data structures.

```
1       /* Legal codelet declaration */
2       #pragma hmpp testlabel1 codelet, target=CUDA, args[v1].io=inout
3         static void codeletNotOk(int n,
4                                  float v1[n],
5                                  float v2[n],
6                                  float v3[n]) {
7         int i;
8         for (i = 1 ; i < n ; i++) {
9           v1[i] = v2[i-1] + v3[i];
10        }
11      }
12
13      int main(int argc, char **argv) {
14        ............
15        /* wrong codelet use: the first two vectors are the same array */
16
17      #pragma hmpp testlabel1 callsite
18        codeletNotOk(n, t1, t1, t3);
19        ............
20      }
```

Listing 3 - Wrong codelet definition due to aliasing between parameters

3.1.1. Execution Error with Synchronous Codelet RPCs

In the case of a synchronous codelet RPC (default), when an error occurs during the hardware allocation, memory loading, or during a codelet call, the runtime API reverts back to the native codelet to resume the execution. This is illustrated in Figure 2.

3.1.2. Execution Error with Asynchronous Codelet RPCs

In asynchronous mode, if a codelet execution fails, the application stops with an error code. In that mode it is not possible to restore the program state, as the effect of executed instructions between the codelet call site and the synchronization barrier are not known and cannot be cancelled. This is illustrated in Figure 2. However, if an error occurs during the memory allocation or during data transfer (the most common cases) the execution of the codelet is cancelled and the HMPP native codelet is executed.

Asynchronous data transfer or asynchronous codelet execution are hardware accelerator dependent.

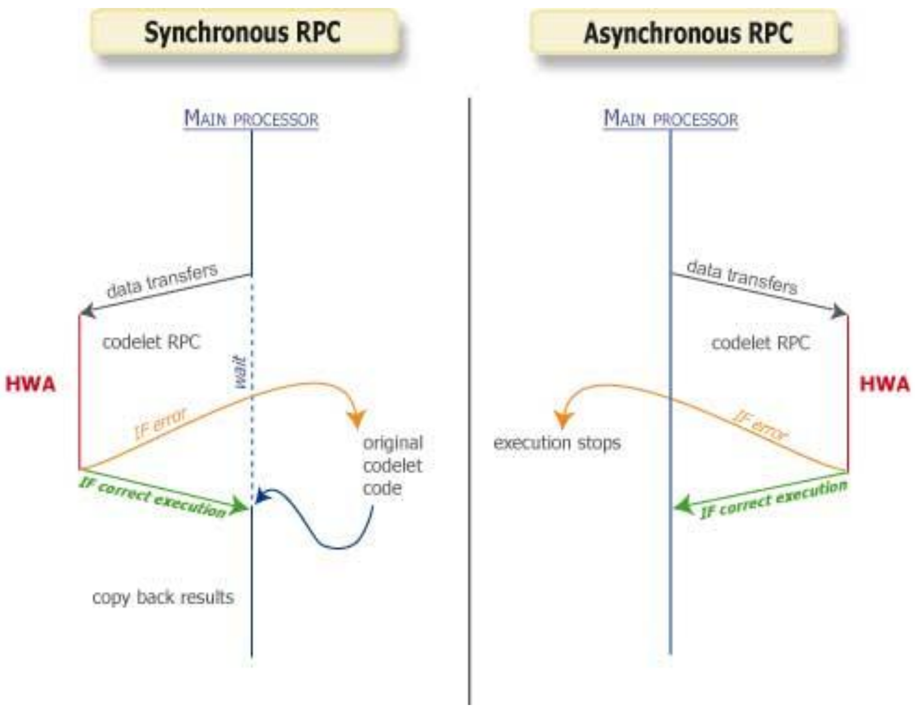


Figure 2 - Synchronous versus asynchronous RPC

3.2. HMPP Runtime

The HMPP runtime is in charge of carrying out the concurrent execution of the native and HWA implementations of the codelets.

At execution, the HMPP runtime detects the available HWAs. When a codelet or a group of codelets is specified to run on a HWA, if a device is available and if the corresponding group of codelets or the codelet implementation is present, the HMPP runtime loads it as a software plug-in. It is not necessary to build a machine-specific version of the host application. The HMPP runtime is able to manage simultaneously multiple and various HWAs.

If an improved version of a codelet is available, the HMPP runtime loads that in place of the previous codelet implementation without any recompilation of the application.

3.3. HMPP Memory Model

In the current version of HMPP, the memory address managed at the host level and at the HWA level are different (see Figure 3). The "Application" and the HMPP runtime have their own private memory. HMPP deals with this in a transparent way for the user. HMPP can be seen as programming glue between target-specific programming environments and general purpose programming.

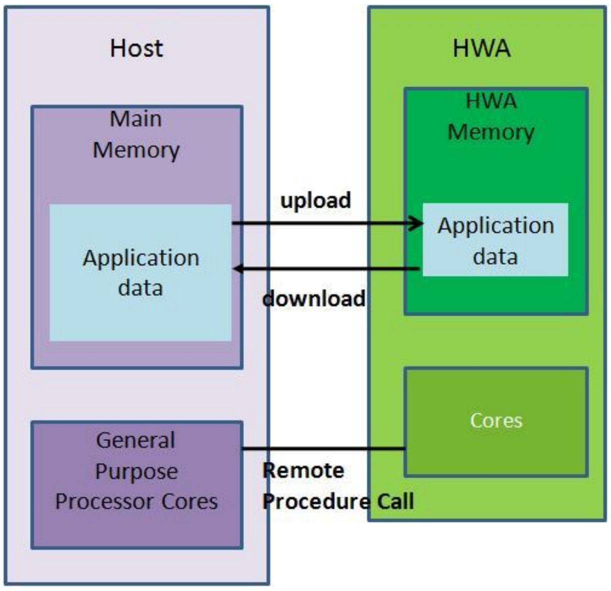


Figure 3 - HMPP memory model

4. HMPP Directives

4.1. Introduction

The HMPP directives may be seen as “meta-information” added in the application source code. They are safe meta-information i.e. they do not change the original code. They address the remote execution (RPC) of a function as well as the transfers of data to/from the HWA memory.

The simplest use case of HMPP directives is composed of two directives made of a codelet declaration and callsite marker. They are identified by a unique label indicated in each directive. The scope of the label is the whole application. For instance, in the listing below the directive at line 2, with label `testlabel`, declares a CUDA codelet implementation to be run on a NVIDIA GPU. The call to this codelet is marked line 31.

```
1   ....
2   #pragma hmpp testlabel codelet, target=CUDA, args[vout].io=inout
3   static void kernel(unsigned int N, unsigned int M,
4                          float vout[N][M], float vin[N][M]){
5     int i, j;
6     for(i = 2; i < (N-2); i++) {
7       for(j = 2; j < (M-2); j++) {
8         float temp;
9         temp = vin[i][j]
10          + 0.3f *(vin[i-1][j-1] + vin[i+1][j+1])
11          - 0.506f *(vin[i-2][j-2] + vin[i+2][j+2]);
12        vout[i][j] = temp * (vout[i][j]);
13      }
14    }
15  }
16  int main(int argc, char **argv){
17    unsigned int n = 100;
18    unsigned int m = 20;
19    int i, j;
20    float resultat = 0.0f;
21    float out[n][m];
22    float in[n][m];
23     …
24    // init
25    for(i = 0 ; i < n ; i++){
26      for(j = 0 ; j < m ; j++){
27        in[i][j] = (COEFF) * (-1.0f);
28        out[i][j] = (COEFF) + (j * 0.01f) ;
29      }
30    }
31  #pragma hmpp testlabel callsite
32     kernel(n,m,out,in);
33     ....
34     printf("result : %f\n",resultat);
35  }
```

Listing 4 - HMPP codelet source code example

The Table 1 below introduces the HMPP directives. HMPP directives address different needs: some of them are dedicated to declarations and others are dedicated to the management of the execution.

	Control flow instructions	Directives for data management
Declarations	▪ codelet ▪ group ▪ function	▪ resident ▪ map ▪ mapbyname
Operational Directives	▪ callsite ▪ synchronize ▪ region ▪ parallel	▪ allocate ▪ free ▪ acquire ▪ release ▪ advancedload ▪ delegatedstore ▪ disregard

Table 1 - HMPP Directives

4.2. Concept of set of directives

One of the fundamental points of the HMPP approach is the concept of directives and their associated labels which makes it possible to recreate a coherent structure on a whole set of directives disseminated in an application.

We distinguish two kinds of labels:

- One associated to a codelet. In general, the directives carrying this kind of labels are limited to the management of only one codelet (called stand-alone codelet in the remainder of the document to distinguish it from the group of codelets).
- One associated to a group of codelets. These labels are noted as follow: "**<***LabelOfGroup***>**", where "*LabelOfGroup*" is a name specified by the user. In general, the directives which have a label of this type relate to the whole group.

The concept of group is reserved to a class of problems which requires a specific management of the data throughout the application to obtain performance.

In the following, for each directive, we will present the both notations for:

- A stand-alone codelet context: it means that only one set of directives associated to one codelet is defined. Note that in an application, several separate set of directives can be defined.
- A group of codelets: means that the set of directives deals with the definition of several codelets in the same group.

The HMPP directives with different labels do not see each other, i.e. a directive of a given label does not interfere with a directive using a different label.

Please note that:

- Inside a set, directives can only interfere (between them) by sharing data;
- No data can be shared between two distinct sets of directives.

4.3. Syntax of the HMPP directives

In order to simplify the notations, regular expressions will be used to describe the syntax of the HMPP directives. Below is a short summary of the main notations used.

- "?" The question mark indicates there is no preceding item or one preceding item.
- "*" The asterisk indicates there are zero or more the preceding items.
- "+" The plus sign indicates that there is one or more the preceding items.

Furthermore, to keep the notation as simple as possible, we separately present the notation used in stand-alone codelet context of the one used with group of codelets. The main difference between the two syntaxes lies in an additional label dedicated to the management of the groups.

We also introduced a color convention for the description of syntax directives:

- Reserved HMPP keywords are **in blue**;
- Elements of grammar which can be declined in HMPP keywords are **in red**;
- User's variables remain in black.

In stand-alone codelet context, the general syntax of the HMPP directives is:

- For C language:

```
#pragma hmpp codelet_label directive_type [, directive_parameters]* [&]
```

- For FORTRAN language:

```
!$hmpp codelet_label directive_type [, directive_parameters]* [&]
```

In a group of codelets context, the general syntax of the HMPP directives is:

- For C language:

```
#pragma hmpp <grp_label> [codelet_label]? directive_type [,directive_parameters]* [&]
```

- For FORTRAN language:

```
!$hmpp <grp_label> [codelet_label]? directive_type [, directive_parameters]* [&]
```

Where:

- `<grp_label>: is` a unique identifier naming a group of codelets. In cases where no groups are defined in the application, this label can simply miss. Legal label name must follow this grammar: `[a-z,A-Z,_][a-z,A-Z,0-9,_]*`. Note that the "< >" characters belong to the syntax and are mandatory for this kind of label.
- `codelet_label:` is a unique identifier naming a codelet. Legal label name must follow this grammar: `[a-z,A-Z,_][a-z,A-Z,0-9,_]*`
- `directive_type:` is the type of the directive;
- `directive_parameters`: designates some parameters associated to the directive_type. These parameters may be of different kinds and specify either some arguments given to the directive either a mode of execution (asynchronous versus synchronous for example);
- `[&]`: is a character used to continue the directive on the next line (same for C and FORTRAN).

This is illustrated below:

- example of a simple codelet declaration with no group definition:

```
#pragma hmpp codelet_label codelet, &
#pragma hmpp &             directive_parameter &
#pragma hmpp &           [, directive_parameter]*
```

- example of a codelet declaration inside a group:

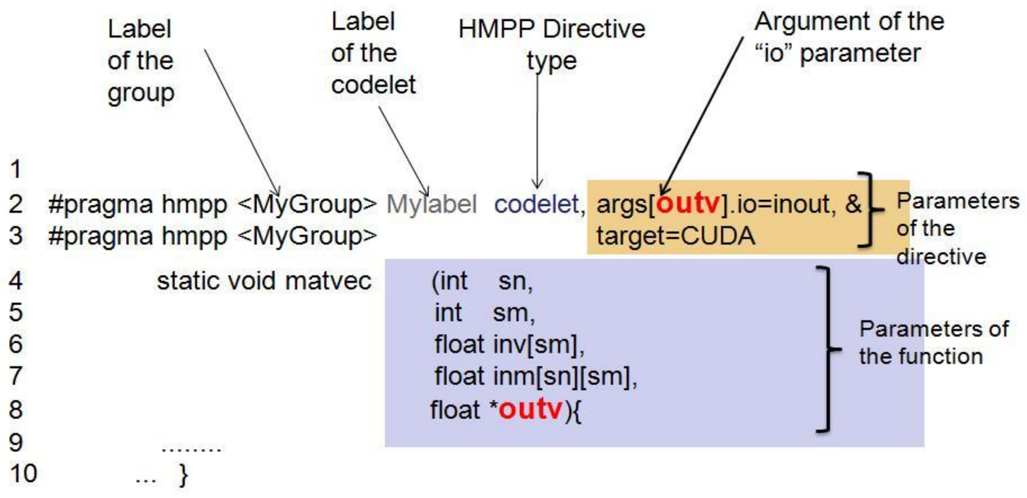
```
#pragma hmpp <grp_label> codelet_label codelet, &
#pragma hmpp &                         directive_parameter &
#pragma hmpp &                       [, directive_parameter]*
```

Furthermore, the directive's parameters may themselves accept some arguments. In most cases, these arguments apply to the parameters of the function.

In the remainder of this document, we will distinguish these two notions by speaking of:

- *Parameters:* directives' parameters,
- *Arguments*: directives parameters' arguments.

The Figure 4 illustrates this with an example. Note that in this example, "`outv`" indicated as a value of the directive parameter points the user's function arguments.



```
#pragma hmpp <MyGroup> Mylabel  codelet, args[outv].io=inout, &
#pragma hmpp <MyGroup>                   target=CUDA
          static void matvec  (int   sn,
                               int   sm,
                               float inv[sm],
                               float inm[sn][sm],
                              float *outv){
          ........
       ...  }
```

Label of the group
Label of the codelet
HMPP Directive type
Argument of the "io" parameter
Parameters of the directive
Parameters of the function

Figure 4 - Description of parameters and arguments in HMPP directives

Values of the directives' parameters can be specified by either:

- Their formal name;
- Or their order in the function definition;
- Or under the form of a range (in case several arguments need to be provided to the directives).

Example:

```
#pragma hmpp <grp_label> directive_type, args[arg_items].xxx
```

Where "`args[arg_items].xxx`" represents the directive's parameter with:

```
arg_items:          arg_item [ ';'  arg_item ]*
arg_item:           IDENTIFIER | NUMBER | arg_range | param_with_ident
arg_range:          NUMBER '-'  NUMBER
param_with_ident:   ident '::' [* | IDENTIFIER]
ident:              codelet_label | *
```

Where:

- `IDENTIFIER:` is the name of a parameter in the codelet prototype;
- `NUMBER` is the numerical position of a function's argument taken in order starting from 0 in the codelet prototype.

Listing 5 provides an example where:

- `args[0-1]`respectively points out `sn` and `sm`,
- `args[inv]` of course designates `inv`,
- `args[3]` designates `inm`,
- etc.

```
#pragma hmpp simple1 codelet, args[0-1;inv].io=in, &
#pragma hmpp &                args[3].io=in,       &
#pragma hmpp &                args[outv].io=inout, &
#pragma hmpp &                target=CUDA
static void matvec(int sn, int sm,
                   float inv[sm], float inm[sn][sm],
                   float *outv){
  ........
}
```

Listing 5 - Directive's parameter and arguments (case of stand-alone codelet notation)

The following constructions are also legal:

```
#pragma hmpp <MyGroup> delegatedstore, args[*::var_b]
```

The "delegatedstore" directive is applied on all the variables "var_b" defined in the group "MyGroup" (codelet's parameters and resident variables if any).

Example:

```
#pragma hmpp <MyGroup> delegatedstore, args[::MyResidentVarData;cod1::var_a;*::var_b]
```

The "delegatedstore" directive is applied on the group "MyGroup" on the following variables:

- the resident data "MyResidentVarData";
- the "var_a" argument of the codelet "cod1";
- all the arguments called "var_b" defined in the group "MyGroup"

Please note that when many parameters of <u>a same codelet</u> are referenced, the following notation is also supported:

```
#pragma hmpp <MyGroup> delegatedstore, args[cod1::var_a;cod1::var_b]
```

is equivalent to:

```
#pragma hmpp <MyGroup> cod1 delegatedstore, args[var_a;var_b]
```

The codelet label "cod1" has been moved at the beginning of the directive and has been removed from the variable declarations in order to shorten the writing.

Table 2 summarizes the different way to access to the arguments:

	By name	By rank (start from 0)	By range	All
Implicit current scope	MyArgument	3	0-5	*
Explicit codelet scope	MyCodelet::MyArgument	MyCodelet::3	MyCodelet::0-7	MyCodelet::*
Explicit resident scope	::MyResidentVariable			::*
Global scope	*::MyVariable			*::*

Table 2 - Access to HMPP arguments according to their scope

In the remainder of this document, most examples of directives will be given in C. FORTRAN directives only differ by their prefix.

In FORTRAN and C languages, directives are case insensitive.

4.4. Factorizing directive arguments: the `with` directive

When using e.g. the multi-device allocation and execution capabilities of HMPP, several directive parameters end up getting duplicated.

The `with` directive allows to add some parameters to the directives of a given scope.

The syntax of the directive is:

```
#pragma hmpp with [,asynchronous]*
                  [,exclusive]*
                  [,device="device_num"]*
                  [,elementsize="expr"]*
                  [,size={dimsize[,dimsize]*}]*
                  [,args[arg_items].section={[subscript_triplet,]+}]*
```

Where:

- `asynchronous`[2]: indicates that the transfer can be performed asynchronously, meaning that it is a non-blocking transfer.
- `exclusive`: specifies that the HWA should be locked to the given codelet or grouplet until it is unlocked with the `release` directive. When locked, the HWA will not be available for use by other codelets or grouplets, as well as to other thread or processes.
- `device="device_number"`: gives the number of the device on which all data should be allocated. This is mainly useful when dispatching computations over multiple devices[3].
- `elementsize="expr"`: specifies the element size (for data mirrors allocation mainly).
- `args[arg_items].size={dimsize[,dimsize]*}`: specifies the size of a non-scalar parameter (an array). Each `dimsize` provides the size for one dimension. `dimsize` must be a simple expression depending only of the scalar arguments of the codelets.
- `args[arg_items].section={[subscript_triplet,]+}*`: indicates that only an array section will be transferred to the device. See section 4.6.4 - Array section in HMPP on page 34 for further details.

The example below give an example of the utilization of the `with` directive:

```
#pragma hmpp with size={100}, elementsize="sizeof(float)", device="i%2"
  for (i = 0; i < 4; ++i) {
    // Declaration, then allocation of data mirrors on alternative devices
    #pragma hmpp f allocate, data["&x[i][0]"]
    #pragma hmpp f allocate, data["&y[i][0]"]
    // upload of data based on the address
    #pragma hmpp f advancedload, data["&x[i][0]","&y[i][0]"]
  }
```

Figure 5 – An example of the utilization of the `with` directive: using the with directive in the mirror allocation loop of Listing 34 on page 52

[2] Target dependent. For more details, see section 4.6.3-Asynchronous transfers on page 35.

[3] See section 4.9 - Parallel directive (Using multiple HWA devices on page 52 for more details

4.5. Directives for Implementing the Remote Procedure Call on a HWA

Using a HWA consists in a remote procedure call. A set of directives controls the implementation of the RPC[4]:

1. The `codelet` directive marks a function as a codelet with the properties of its parameters (inputs and outputs).
2. The `callsite` directive declares the call to the codelet that is remotely executed.

4.5.1. *`codelet` directive*

A codelet directive specifies that a version of the function following must be optimized for a given hardware. Its label must be unique in the application.

For the codelet directive:

- The codelet label is mandatory
- The group label is not required if no group is defined.

The codelet directive must be inserted immediately before the function declaration or definition in C, immediately before the subroutine definition in FORTRAN.

The syntax of the directive is:

For a stand-alone codelet:

```
#pragma hmpp codelet_label codelet [, args[arg_items].io=[in|out|inout|none]]*
                           [, args[arg_items].size={dimsize[,dimsize]*}]*
                           [, args[arg_items].transfer=[atcall|atfirstcall|manual|auto]] *
                           [, cond = "expr"]
                           [, target=target_name[:target_name]*]
```

For a group of codelets:

```
#pragma hmpp <grp_label> codelet_label codelet [, args[arg_items].io=[in|out|inout|none]]*
                             [, args[arg_items].size={dimsize[,dimsize]*}]*
                             [, args[arg_items].transfer=[atcall|atfirstcall|manual|auto]]*
                             [, cond = "expr"]
                             [, target=target_name[:target_name]*]
```

Where:

- `<grp_label>`: is a unique identifier associated with all the directives that belong to the group (definition and use).
- `codelet_label`: is a unique identifier associated with all the directives that belong to the same codelet execution (definition and use).

[4] Further details about HMPP's RPC can be found in the reference document [R1].

- `args[arg_items].size={dimsize[,dimsize]*}`: specifies the size of a non-scalar parameter (an array). Each `dimsize` provides the size for one dimension. `dimsize` must be a simple expression depending only of the scalar arguments of the codelets.

`args[arg_items].transfer=[atcall|atfirstcall|manual|auto]`: indicates which transfer policy should be used for transfer policy should be used for each argument. Though section 0 - Listing 16 - Array section in advancedload directive - Transfer of 1 row (FORTRAN)

- Transfer policies on page 36 has more details, each valid value is briefly described below :
  - `atcall`: indicates that HMPP should systematically upload/download the argument right before and after the callsite ;
  - `atfirstcall`: indicates that the argument is to be uploaded only once;
  - `manual`: indicates that the argument will not be uploaded/downloaded unless an explicit transfer (`advancedload/delegatedstore`) is requested
  - `auto`: indicates that HMPP should automatically and cleverly upload/download the argument right before and after the callsite
- `args[arg_items`[5]`].io=[in|out|inout|none]`: indicates that the specified function arguments are either input, output, both (`inout`) or unused (`none`). By default, unqualified arguments of codelets, region and resident are INOUT.
  The specification of this parameter drives the data transfers between the host and the HWA. Furthermore, it allows some additional checks about the use of the data in HMPP applications.

In FORTRAN, the "***.io***" parameter can be omitted when an 'INTENT' attribute is explicitly specified in the code source.

Table 3 describes the policy applied when both the FORTRAN INTENT and the HMPP parameters are specified.

In C, a scalar argument is passed by value, so its HMPP input/output property cannot be OUT or INOUT. Pointer argument with a `const` attribute has the same restriction (see Table 4).

INTENT / HMPP IO	Default	IN	OUT	INOUT
Unset	IN	IN	OUT	INOUT
IN	IN	IN	Error	Warning
OUT	OUT	Error	OUT	Warning
INOUT	INOUT	Error	Error	INOUT

Table 3 - Intent in FORTRAN language versus HMPP Input/Output parameter policy

[5] See section 4.3 for the syntax of `arg_items`.

C Parameters / HMPP IO	By Value	By Const address	By address
Unset	IN	IN	INOUT
IN	IN	IN	IN
OUT	Error	Error	OUT
INOUT	Error	Error	INOUT

Table 4 - C language parameter versus HMPP Input/Output parameter policy

- `cond="expr"`: specifies an execution condition as a boolean C or Fortran logical expression that needs to be true in order to start the execution of the codelet. The expression must be correct and evaluable in all operational directive contexts (see Table 1 - HMPP Directives). `cond` is useful to control the flow of directive execution. All directives are normally executed but since they are invisible to the host compiler (they are treated as comments in FORTRAN for example) they will still be executed by HMPP even if, for example, a `goto` statement in the host code implicitly skips a HMPP directive. The host code is required to set up the expression "expr" so that if it wants to skip an HMPP directive "expr" evaluates to FALSE.
- `target=target_name[:target_name]*`: specifies one or more targets for which the codelet must be generated. It means that according to the target specified, if the corresponding hardware is available AND the codelet implementation for this hardware is also available, this one will be executed. Otherwise, the next target specified in the list will be tried. The values of the targets can be one of the following:
  - `CUDA`: for NVIDIA platforms.
  - `OPENCL`: for NVIDIA and AMD ATI Stream Computing platforms.

  For more information on the targets, please refer to section 6.

The examples below give example of codelet declaration:

```
#pragma hmpp simple1 codelet, args[outv].io=inout, target=CUDA
static void matvec(int sn, int sm,
                   float inv[sm], float inm[sn][sm], float *outv){
  int i, j;
  for (i = 0 ; i < sm ; i++) {
    float temp = outv[i];
    for (j = 0 ; j < sn ; j++) {
      temp += inv[j] * inm[i][ j];
  }
  outv[i] = temp;
}

int main(int argc, char **argv) {
  int n;
  ........
 #pragma hmpp simple1 callsite, args[outv].size={n}
  matvec(n, m, myinc, inm, myoutv);
  ........
}
```

Listing 6 - Simple codelet declaration

```
#pragma hmpp <myGroup> simple1 codelet, args[outv].io=inout, target=CUDA
static void matvec(int sn, int sm,
                   float inv[sm], float inm[sn][sm], float *outv){
  int i, j;
  for (i = 0 ; i < sm ; i++) {
    float temp = outv[i];
    for (j = 0 ; j < sn ; j++) {
      temp += inv[j] * inm[i][ j];
  }
  outv[i] = temp;
}

int main(int argc, char **argv) {
  int n;
  ........
  #pragma hmpp <myGroup> simple1 callsite, args[outv].size={n}
  matvec(n, m, myinc, inm, myoutv);
  ........
}
```

Listing 7 - Codelet declaration inside a group

More than one codelet directive can be added to a function in order to specify different uses or different execution contexts. However, there can be only one codelet directive for a given call site label. An example is given below:

```
#pragma hmpp simple1 codelet, args[outv].io=inout, &
#pragma hmpp &                cond ="n==1024", target=CUDA
#pragma hmpp simple2 codelet, args[outv].io=inout, &
#pragma hmpp &                cond ="n==40", target=OPENCL
static void matvec(int sn, int sm,
                   float inv[sm], float inm[sn][sm], float *outv){
  int i, j;
  for (i = 0 ; i < sm ; i++) {
     float temp = outv[i];
    for (j = 0 ; j < sn ; j++) {
      temp += inv[j] * inm[i][ j];
    }
    outv[i] = temp;
  }
}
int main(int argc, char **argv) {
  int n;
  ........
#pragma hmpp simple1 callsite, args[outv].size={n}
#pragma hmpp simple2 callsite, args[outv].size={n}
  matvec(n, m, myinc, inm, myoutv);
  ........
#pragma hmpp simple1 release
#pragma hmpp simple2 release
}
```

Listing 8 - Multiple codelet declarations (stand-alone codelet context)

Note that if more than one `callsite` directive precedes a function call, only one of them can initiate an RPC call. The execution policy is based on the order of the `callsite` directives: the directives are evaluated one after the other sequentially. Thus, a `callsite` can be launched if and only if the condition of all previous `callsite` directives failed and the condition of the current directive is true and the HWA is available. Subsequent directives will be ignored once one has been executed.

The target codelet can either be produced using one of the appropriate HMPP codelet generator or hand-written using HWA vendor programming language (i.e. CUDA for NVIDIA targets or OPENCL).

4.5.2. `group` directive

The group directive allows the declaration of a group of codelets. The parameters defined in this directive are applied to all codelets belonging to the group.

The syntax of the directive is:

```
#pragma hmpp <grp_label> group ,[target= target_name[:target_name]*]]? &
#pragma hmpp &                 ,[cond  = “expr”]?
```

Where the directive parameters are:

- `<grp_label>:` a unique identifier associated with all the directives that belong to the group (definition and use). Thus, this label will have to be re-used to be able to run any codelet within a group.
- `cond = "expr"`: specifies an execution condition as a boolean C or Fortran logical expression that needs to be true in order to start the execution of the group of codelets. If a condition is specified at this level for a group, this one will overwrites all the codelet’s conditions of the same group. See comments above under codelet directive for alternate applications of this `cond` parameter.
- `target=target_name[:target_name]*`: specifies which targets to use and their order. Means that according to the target specified, if the corresponding hardware is available AND that all the codelet implementations for this hardware are also available, this one will be executed. Otherwise, the next target specified in the list will be checked. For more information on targets, please refer to chapter 6, HMPP Codelet Generators.

4.5.3. `callsite` directive

The `callsite` directive specifies the use of a codelet at a given point in the program. Related data transfers and synchronization points that are inserted elsewhere in the application have to use the same label.

For the callsite directive:

- The codelet label is always mandatory
- The group label is required if the codelet belongs to a group.

The callsite directive must be inserted immediately before the function call.

The syntax of the directive is:

In stand-alone codelet context:

```
#pragma hmpp codelet_label callsite [, asynchronous]?
```

In group of codelets context:

```
#pragma hmpp <grp_label> codelet_label callsite [, asynchronous]?
```

Where the directive parameters are:

- `<grp_label>`: is a unique identifier associated with all the directives that belong to the group (definition and use).
- `codelet_label`: is a unique identifier associated with all the directives that belong to the same codelet execution (definition and use).
- `asynchronous`: specifies that the codelet execution is not blocking (default is synchronous). In asynchronous mode, all the output parameters have to be downloaded using `delegatedstore` directive.
  A `synchronize` directive is mandatory before the first `delegatedstore` directive to insure that the codelet execution is completed.
  When an asynchronous codelet is declared, the `release` directive is also mandatory.

Usage examples of the `callsite` directive are given in Listing 8. If the condition of the directive is not evaluated as `true`, or if no resources are available on the HWA, the native codelet code is used instead.

4.5.4. `synchronize` directive

The `synchronize` directive specifies to wait until the completion of an asynchronous callsite execution.

For the synchronize directive:

- The codelet label is always mandatory
- The group label is required if the codelet belongs to a group.

The syntax of the directive is:

In stand-alone codelet context:

```
#pragma hmpp codelet_label synchronize
```

In group of codelets context:

```
#pragma hmpp <grp_label> codelet_label synchronize
```

Where the directive parameters are:

- `<grp_label>`: a unique identifier associated to all the directives that belong to the group (definition and use).
- `codelet_label`: a unique identifier associated to all the directives that belong to the same codelet execution (definition and use).

Note that the `synchronize` directive is only a synchronization barrier.

4.5.5. `acquire` directive

An HWA may need some time to be allocated or initialized before being used by a directive set. Thus, before the RPC call or any data uploading, an anticipated allocation of the hardware would improve the execution time of the RPC. This anticipated allocation can be done using the `acquire` directive.

When an `acquire` directive is used, it should be placed so that it is executed before all other instructions of the directive set. If another directive is reached before that `acquire` directive, then the HMPP's runtime will implicitly acquire the default HWA (i.e. like if the `device="device_number"` clause was ignored).

The syntax of the directive is:

```
#pragma hmpp codelet_label acquire [device="device_number"], [exclusive]
```

In group of codelets context:

```
#pragma hmpp <grp_label> acquire [device="device_number"], [exclusive]
```

Where:

- `device="device_number"`: gives the number of the device on which all data should be allocated. This is mainly useful when dispatching computations over multiple devices[6].
- `exclusive`: specifies that the HWA should be locked to the given codelet or grouplet until it is unlocked with the `release` directive. When locked, the HWA will not be available for use by other codelets or grouplets as well as to other threads or processes.

4.5.6. `release` directive

The `release` directive specifies when to release the HWA for a group or a stand-alone codelet (this directive is generally used in association with the "`acquire`" directive (see the section 4.5.5 `acquire` directive above). The `release` directive does not physically free the HWA but marks it for re-allocation.

If no `release` directive is specified, by default, HWA is released at program exit.

The syntax of the directive is the following:

In stand-alone codelet context:

```
#pragma hmpp codelet_label release [device="device_number]
```

In group of codelets context:

[6] See section 4.9 - Parallel directive (Using multiple HWA devices on page 52 for more details

```
#pragma hmpp <grp_label> release [device="device_number"]
```

Where the directive parameters are:

- `<grp_label>`: a unique identifier associated to all the directives that belong to the group (definition and use).
- `codelet_label`: a unique identifier associated to all the directives that belong to the same codelet execution (definition and use).
- `device="device_number"`: gives the specific number of the device to release[7].

Listing 9 shows a usage of the `release` directive. The allocated HWA of the `testlabel1` call site is released after the while loop.

```
  ......
   while (j){
     for (k = 0 ; k < iter ; k++) {
 #pragma hmpp testlabel1 callsite
         simplefunc1(n, &(t1[k*n]), &(t2[k*n]), &(t3[k*n]));
       }
     j--;
   }
 #pragma hmpp testlabel1 release
  ......
```

Listing 9 - `release` directive example (case of stand-alone codelet notation)

4.5.7. *`allocate` directive*

To allocate the codelet's arguments memory on the HWA, HMPP evaluates the sizes of the non-scalar parameters during the execution either from the codelet's signature, or directly from an expression given by the user in the `callsite` (which is not recommended as it is deprecated) (see parameter `size` of the HMPP `callsite` directive, chapter 4.5.2).

This directive can also be used to allocate data mirrors (see section 4.8.4, Data mirroring directives)

Note that once the size has been evaluated, it cannot be changed during any execution of the codelet up to the next `free` directive.

The syntax of the directive is:

In stand-alone codelet context:

```
#pragma hmpp codelet_label allocate [,(args|data)[arg_items].size={dimsize[,dimsize]*}]*
                                    [,(args|data)[arg_items].elementsize="expr"]*
                                    [,(args|data)[arg_items].device="device_number"]*
                                    [,(args|data)[arg_items].hostdata="var_addr"]*
```

In group of codelets context:

```
#pragma hmpp <grp_label> allocate [,(args|data)[arg_items].size={dimsize[,dimsize]*}]*
                                  [,(args|data)[arg_items].elementsize="expr"]*
                                  [,(args|data)[arg_items].device="device_number"]*
                                  [,(args|data)[arg_items].hostdata="var_addr"]*
```

[7] See section 4.9 - Parallel directive (Using multiple HWA devices on page 52 for more details

Where the directive parameters are:

- `<grp_label>`: a unique identifier associated to all the directives that belong to the group (definition and use).
- `codelet_label`: a unique identifier associated to all the directives that belong to the same codelet execution (definition and use).
- `(args|data)[arg_items`[8]`].size={dimsize[,dimsize]*}`: gives an alternate way to evaluate the size of non-scalar codelet arguments or data mirrors. Each `dimsize` provides the size for one dimension. "`dimsize`" is an expression evaluable at the location of the directive (can be a variable, a value, an expression to evaluate, etc.).
- `(args|data)[arg_items`[9]`].elementsize="expr"`: specifies the element size of the allocated memory (for data mirrors mainly).
- `(args|data)[arg_items].device="device_number"`: gives the number of the device on which the data should be allocated. This is mainly useful when dispatching computations over multiple devices[10].
- `args[arg_items].hostdata="expr"`: `expr` is an expression that gives the host address of the data to upload.

This directive is used when the `callsite` specifies a size that is not known in the `advancedload` directive used. The size must be specified for each dimension of the argument. Listing 10 illustrates the `size` declaration for two n-by-m matrices "`inm`" and "`outv`".

Please, note that once a "`.size`" parameter is specified for an argument in an `allocate` directive, this value cannot be changed in an `advancedload` or `delegatedstore` directives.

```
#pragma hmpp matvec allocate, args[inm;outv].size={n,m}
  ....
  while (...){
#pragma hmpp matvec callsite, asynchronous
    matvec(n, m, (inc+(k*n)), inm, (outv+(k*m)));
    ....
#pragma hmpp matvec synchronize
#pragma hmpp matvec delegatedstore, args[outv]
  }/* endwhile */
#pragma hmpp matvec release
```

Listing 10 - `allocate` directive example (case of stand-alone codelet notation)

4.5.8. `free` directive

Data mirrors can be dynamically created with the `Listing 9` - `release` directive example (case of stand-alone codelet notation)

`allocate` directive, so it is also logical to allow to destroy them dynamically. This directive allows that.

[8] See section 4.3 for the syntax or `arg_items`.

[9] See section 4.3 for the syntax or `arg_items`.

[10] See section 4.9 - Parallel directive (Using multiple HWA devices on page 52 for more details

```
#pragma hmpp (<grp_label>|codelet_label|) free [,(args|data)[arg_items]
```

Where the directive parameters are:

- `<grp_label>:` a unique identifier associated to all the directives that belong to the group (definition and use).
- `codelet_label`: a unique identifier associated to all the directives that belong to the same codelet execution (definition and use).
- `(args|data)[arg_items`[11]`]:` gives the name of the codelet argument or the base address of the mirror to de-allocate from the HWA.

4.6. Controlling Data Transfer between the Host CPU and the HWA

When using a HWA, an important bottleneck is often the data transfer between the HWA memory and the host memory. To limit the communication overhead, the programmer can try to overlap data transfers with successive executions of the same codelets by using the asynchronous property of the HWA. Two directives can be used for that purpose:

1. The `advancedload` directive loads data before the remote execution of the codelet.
2. The `delegatedstore` directive delays the fetching of the result.

These directives are detailed in the next sections.

4.6.1. *advancedload directive*

Data can be uploaded before the execution of the codelet by using the `advancedload` directive. The syntax is:

In stand-alone codelet context:

```
#pragma hmpp codelet_label advancedload
                           ,args[arg_items]
                          [,args[arg_items].size={dimsize[,dimsize]*}]*
                          [,args[arg_items].addr="expr"]*
                          [,args[arg_items].hostdata="expr"]*
                          [,args[arg_items].section={[subscript_triplet,]+}]*
                          [,asynchronous]
```

In group of codelets context:

```
#pragma hmpp <grp_label> [codelet_label]? advancedload
                         ,args[arg_items]
                         [,args[arg_items].size={dimsize[,dimsize]*}]*
                         [,args[arg_items].addr="expr"]*
                         [,args[arg_items].hosdata="expr"]*
                         [,args[arg_items].section={[subscript_triplet,]+}]*
                         [,asynchronous]
```

Where the directive parameters are:

- `<grp_label>:` a unique identifier associated with all the directives that belong to the group (definition and use).

[11] See section 4.3 for the syntax or `arg_items`.

- `codelet_label`: a unique identifier associated with all the directives that belong to the same codelet execution (definition and use).
- `args[arg_items]`: the name or rank (caller program) of the argument to be loaded.
- `args[arg_items].size={dimsize[,dimsize]*}`: gives an alternate way to evaluate the size of non scalar codelet arguments. Each `dimsize` provides the size for one dimension. This parameter may be used when the `callsite` specifies a size that is not known in the `advancedload` directive used.
  Note: This parameter is deprecated since the size should preferably be specified though an `allocate` directive.
- `args[arg_items].addr="expr"`: `expr` is an expression that gives the host address of the data to upload.
  Note: This parameter is deprecated since it lead to some users to believe that it allowed to manipulate the base address on the HWA's side. Users should specify the base address with the `.hostdata` parameter.
- `args[arg_items].hostdata="expr"`: `expr` is an expression that gives the host address of the data to upload.
- `args[arg_items].section={[subscript_triplet,]+}*`: indicates that only an array section will be transferred to the device. See section 4.6.4 - Array section in HMPP on page 34 for further details.
- `asynchronous`[12]: indicates that the transfer can be performed asynchronously, meaning that it is a non-blocking transfer.

The `advancedload` directive is used on data whose the "`intent`" status is "`in`" or "`inout`". An error message is generated otherwise.

```
1   #pragma hmpp matvec codelet, args[n;m; inc].transfer=atcall, args[inm;outv].transfer=manual
2   void matvec(int n, int m, float *inc, float *inm, float *outv);
3
4   #pragma hmpp matvec advancedload, args[inm], args[inm].size={n,m}
5     ....
6     while (...){
7   #pragma hmpp matvec callsite, args[inm].size={n+1,m+1}, &
8   #pragma hmpp &                 asynchronous
9      matvec(n, m, (inc+(k*n)), inm, (outv+(k*m)));
10      ....
11  #pragma hmpp matvec synchronize
12  #pragma hmpp matvec delegatedstore, args[outv]
13      if (...) {
14        for (i=0; i<m; i++) {
15          inm[...] = 0.1;
16        } /* endfor */
17  #pragma hmpp matvec advancedload, args[inm]
18      } /* endif */
19    } /* endwhile */
```

Listing 11 - `advancedload` directive example *(case of stand-alone codelet notation)*

An example of the `advancedload` directive is given in Listing 11. The `advancedload` directive at line 17 loads the `inm` matrix after it has been modified and before the next call to the codelet

[12] Target dependent. For more details, see section 4.6.3-Asynchronous transfers on page 35.

Warning:

The expression used to specify the size and address of the arguments can be evaluated only when the `advancedload` is used. However, most inconsistencies are likely to be detected at compile time. Listing 12 shows an illegal use of the `advancedload` directive where an error message will be issued by the compiler.

```
void foo_xxx(int* N, float* CA, float* CX, float* CY) {
  ...
  /* Illegal preloading of the "table" input data because
     table is declared below (“table” designated here as args [0]) */
#pragma hmpp callfoo advancedload, args[0], &
#pragma hmpp &                          asynchronous
   ...
  /* Call the codelet */
   {
      float table[2];
      table[0] = 3.14159265357;
      table[1] = 2.718281;
  #pragma hmpp callfoo callsite, asynchronous
      foo_hmpp(table, CX, CY, SY_out);
    }
    ...
  #pragma hmpp callfoo synchronize
    /* Starting from there, the codelet execution has complete */
    ...
  #pragma hmpp callfoo delegatedstore, args[SY_out]
    /* Starting from there, the value of SY_out has been updated */
    ...
  #pragma hmpp callfoo release
    /* Starting from there, the hardware can be reallocated
       to another codelet */
  }
```

Listing 12 - Illegal use of the `advancedload` directive - (the actual arguments of the codelet is not in the scope of the `advancedload` directive).

When the execution reaches an `advancedload` program point, the HWA, if available, is locked by the HMPP runtime. When an asynchronous `advancedload` directive is used, the argument must not be modified between that directive and the call of the codelet.

4.6.2. `delegatedstore` directive

The `delegatedstore` directive is the opposite of the `advancedload` directive in the sense that it downloads output data from the HWA to the host. The program execution is pause until all transfers are completed. The syntax is:

In stand-alone codelet context:

```
#pragma hmpp codelet_label delegatedstore
                      ,args[arg_items]
                      [,args[arg_items].addr="expr"]*
                      [,args[arg_items].hostdata="expr"]*
                      [,args[arg_items].section={[subscript_triplet,]+}]*
                      [,asynchronous]
```

In group of codelets context:

```
#pragma hmpp <grp_label> [codelet_label]? delegatedstore
                      ,args[arg_items]
                     [,args[arg_items].addr="expr"]*
                     [,args[arg_items].hostdata="expr"]*
                     [,args[arg_items].section={[subscript_triplet,]+}]*
                     [,asynchronous]
```

Where the directive parameters are:

- `<grp_label>`: a unique identifier associated with all the directives that belong to the group (definition and use).
- `codelet_label`: the unique identifier associated with all the directives that belong to the same codelet execution (definition and use);
- `args[arg_items`[13]`]`: the name (caller program) or rank of the codelet arguments to download.
- `args[arg_items].addr="expr"`: `expr` is an expression that gives the host address of the data to store.
  Note: This parameter is deprecated since it lead to some users to believe that it allowed to manipulate the base address on the HWA's side. Users should specify the base address with the `.hostdata` parameter of directives that support it.
- `args[arg_items].hostdata="expr"`: `expr` is an expression that gives the host address of the data to upload.
- `args[arg_items].section={[subscript_triplet,]+}*`: indicates that only an array section will be transferred from the device. See section 4.6.4 - Array section in HMPP on page 34 for further details.
- `asynchronous`[14]: indicates that the transfer can be performed asynchronously, meaning that it is a non-blocking transfer.

An example of the `delegatedstore` directive is given in Listing 13. In this example, the simple function is called twice. Only the first call is a candidate for remote execution, so only that call is offloaded to an accelerator or a worker thread. The value of `myoutv1` is downloaded after the second call.

Note that for an asynchronous `callsite` a `delegatedstore` directive must be preceded by a `synchronize` directive.

The `delegatedstore` directive is used on data whose the "`intent`" status is "`inout`" or "`out`". An error message is generated otherwise.

```
#pragma hmpp simple callsite, asynchronous
  simple(n, m, myinc1,inm, myoutv1);
  simple(n, m, myinc2,inm, myoutv2);
#pragma hmpp simple synchronize
#pragma hmpp simple delegatedstore, args[outv]
#pragma hmpp simple release
```

Listing 13 - `delegatedstore` directive example

[13] See section 4.3 for the syntax or `arg_items`.

[14] Target dependent. For more details, see section 4.6.3-Asynchronous transfers on page 35.

Warnings:

You have to ensure that the argument expression stays valid in the context of the `delegatedstore` *use.*

This directive is mandatory in the context of asynchronous `callsite`*.*

4.6.3. Asynchronous transfers

Like asynchronous `callsites`, asynchronous transfers are useful to overlap operations on the HWA with operations on the host.

In OpenCL, no specific allocation functions are required to ensure asynchronous operations. Asynchronous behavior depends on how the OpenCL library has been implemented by the vendor.

Asynchronous Loads

To enable asynchronous loads, the asynchronous clause of the `advancedload` directive must be used.

The semantic of the operation is defined as follows:

- The load operation starts when the directive is reached. As a consequence, transferred arguments must not be modified during the whole operation.
- The HMPP runtime automatically waits for pending transfers when a callsite directive is reached.
- The new `waitload` directive can[15] be used to wait for one or more load operations to complete:

```
#pragma hmpp waitload [,args[arg_items]]*
```

The asynchronous clause does not guarantee real non-blocking and overlapping transfers. The current implementation is subject to the target limitations.

Asynchronous Stores

To enable asynchronous stores, the asynchronous clause of the `delegatedstore` directive must be used.

The semantic of the operation is defined as follows:

- The store operation starts when the directive is reached. As a consequence, transferred arguments must not be modified or read during the whole operation.
- The HMPP runtime automatically waits for pending transfers when a release directive is reached.
- The `waitstore` directive should[16] be used to wait for one or more store operations to complete:

```
#pragma hmpp waitstore [,args[arg_items]]*
```

The asynchronous clause does not guarantee real non-blocking and overlapping transfers. The current implementation is subject to the target limitations and constraints.

[15] In the absence of `waitload` directive, the next callsite affected by this load will implicitly wait for it to complete if needed.

[16] In the absence of `waitstore`, your code may yield non reproducible results since the content of the downloaded variable may or may not have arrived on the host when you read it.

4.6.4. Array section in HMPP

An array section is a selected portion of an array. It designates a set of elements from an array.

The array sections can be used in order to optimize data transfers between the host and the HWA in some cases where it is not necessary to transfer the whole array.

This parameter can be used with both the `advancedload` and the `delegatedstore` directives (see respectively chapter 4.6.1 and 4.6.2).

The syntax of this parameter is of the form:

```
args[arg_item].section={[subscript_triplet,]+}*
```

Where

- `arg_item` designates an array;
- `subscript_triplet` consists of two subscripts and a stride and defines a sequence of numbers corresponding to array element positions along a single dimension. The notation for the subscript_triplet is: “`start:end:stride`” where:
    - `start, end`: are subscripts which designate the first and last values of a dimension.
    - `stride`: is a scalar integer expression that specifies how many subscript positions to count to reach the next selected element. If the stride is omitted, it has a value of 1. The stride must be positive.

The `subscript_triplet` must be specified for each dimension of the array.

Warnings:

Array sections must be used carefully in HMPP applications. Indeed, the use of a stride greater than 1 may results to a slowdown of the application when lots of data are transferred. In such cases, the transfer of the whole array still remains the best solution.

To get performance, users should not forget the constraints inherent in data layout:

- They should favor the transfer of contiguous data;

- They should favor data locality in array section (means for example to transfer data by column for FORTRAN and by row for C language instead of the opposite).

Case of not normalized arrays

By default HMPP makes the assumption that the arrays are normalized, meaning that all the dimensions of the arrays:

- Start from 0, in C language;
- Start from 1, in FORTRAN language;

In cases where at least one of an array’s dimensions is not normalized, the shape must be specified using the following notation:

```
args[arg_item].section={[subscript_triplet,]+}* of {[shape_couple,]+}
```

Where `shape_couple`: designates the first and the last values in the sequence of indices for a dimension.

Listing 14 illustrates the approach. In the `delegatedStore` directive, the array section requests the transfer of the contiguous data “`u[0:1024]`” of a one dimension array “`u`” declared with the “`(-1024:1024)`” array shape.

```
…
INTEGER, PARAMETER :: M=4
INTEGER, PARAMETER :: Ns=-1024
INTEGER, PARAMETER :: Ne=+1024
REAL  :: u(Ns:Ne) , v(Ns:Ne)
…
!- Transfer of the whole array
!$HMPP <conv> advancedload, args[f1::A]

!- callsite
!$HMPP <conv> f1 callsite
call doubleconv1d(Ne-Ns,M,u,v,coef)
…
!- callsite
!$HMPP <conv> f2 callsite
call conv1d(Ne-Ns,M,u,coef)
…

!- get only the modified data on the host
!$HMPP <conv> delegatedstore, args[f1::A],args[f1::A].section={0:Ne} of { Ns:Ne }
.
.
.


!------------
! Codelet declaration
!--------------
!$HMPP <conv> f1 codelet
  SUBROUTINE doubleconv1d(n,iter,A,B,C)
.
.
.
```

Listing 14 - Array section specified with a shape (extract) (FORTRAN)

Use of array sections in HMPP, examples

Below are a few examples provided to illustrate the use of the “`.section`” parameter.

```
INTEGER, PARAMETER :: size = 3661
INTEGER*4, dimension(size,size) :: tab
…
  !$hmpp <Mygroup> get_col advancedload, args[tab], args[tab].section={1:size,1:1}
…
  !$hmpp <group> get_col callsite
  call put(size, tab)
…
!$hmpp <Mygroup> get_col codelet, args[tab].transfer=atcall
SUBROUTINE put(size, tab)
…
END SUBROUTINE put
```

Listing 15 - Array section in advancedload directive - Transfer of 1 column (FORTRAN)

On Listing 15, through the use of an `advancedload` directive, the user transfers the first column, and on Listing 16 the first row of the array “`tab`”. The `advancedload` parameter is set to true at the `callsite` level to notify that the transfer of the data has already been done.

```
INTEGER, PARAMETER :: size = 3661
INTEGER*4, dimension(size,size) :: tab
…
  !$hmpp <Mygroup> get_col advancedload, args[tab], args[tab].section={1:1,1:size}
…
  !$hmpp <group> get_col callsite
  call put(size, tab)
…
!$hmpp <Mygroup> get_col codelet, args[tab].transfer=atcall
SUBROUTINE put(size, tab)
…
END SUBROUTINE put
```

Listing 16 - Array section in advancedload directive - Transfer of 1 row (FORTRAN)

4.7. Transfer policies

By default, all data transfers are being done implicitly by HMPP. This has the great advantage that a code can be offloaded to a GPU with the blink of an eye since all you need is to put two directives on your code, namely the `codelet/callsite` directives.

However, once you want to finely control transfers, be it to remove redundant transfers, or to leave some GPU data untouched, you need to add directives options to explicitly disable implicit transfers and add explicit transfers where needed.

This policy, thereafter referred as the legacy policy, not only makes it harder than necessary to change the default behavior, but also leads to code that may be hard to review, as implicit and explicit operations are mixed.

With HMPP3, several policies are available to simplify the transition from a basic, and less efficient, usage of HMPP directives, to a more efficient, advanced usage.

4.7.1. `atcall` transfer policy

This transfer policy is the easiest to understand, and the easiest to use. All transfers are implicitly performed at the callsite.

At the callsite are performed the following operations:

- Update of the pointer to the host data according to the argument passed to the callsite.
- Transfer of the data zone from the host to the accelerator if the parameter has IN or INOUT intent.
- Execution of the codelet on the accelerator
- Transfer of the data zone from the accelerator to the host if the parameter has OUT or INOUT intent.

In the example below, the `atcall` transfer policy is used instead of the usual, “`legacy`” transfer policy.

```
#pragma hmpp foo codelet, target=CUDA, args[a].io=inout, args[*].transfer=atcall
void foo(int a[10], const int b[10]) {
  for (int i = 0; i < 10; ++i)
    a[i] *= b[i];
}

int A[10], B[10];

int main(void) {
  #pragma hmpp foo allocate

  for (int j = 0; j < 2; ++j) {
    #pragma hmpp foo callsite
    foo(A, B);
  }

  #pragma hmpp foo release
  return 0;
}
```

Listing 17 - atcall transfer policy example

The following transfers will be performed at the execution:

- callsite: A➔a, B➔b, A🡐a
- callsite: A➔a, B➔b, A🡐a

Note that this pattern of transfers corresponds to the pattern which would be obtained with the "`legacy`" policy.

The use of the wildcard notation "`*`" means that all the parameters are concerned by this property. In some situations, only some parameters can be specified.

This policy forbids the following directive options to be used:

- `.addr`
- `.advancedload`

Note that these directives are forbidden on arguments that use the `atcall` transfer policy since that policy tells HMPP to always transfer all arguments. Therefore, using these directive options doesn't make sense.

What's more, this policy should not[17] be used with any `advancedload` or `delegatedstore` directives.

If they are used anyway, the pattern of transfer when the callsite is reached will remain the same, and additional `advancedload` or `delegatedstore` directives will just generate extraneous transfers.

4.7.2. *`atfirstcall`* *transfer policy*

This transfer policy allows transfers savings by leaving constant codelet arguments on the HWA's memory. This policy should be used instead of the `.const` directive attribute, that has been deprecated since HMPP-3.0.

[17] Whether the HMPP compiler will accept `advancedload` or `delegatedstore` directives on parameters that use the `atcall` policy is subject to changes.

```
#pragma hmpp foo codelet, target=CUDA, args[a].io=inout, args[b].transfer=atfirstcall, &
#pragma hmpp & args[a].transfer=atcall
void foo(int a[10], const int b[10]) {
  for (int i = 0; i < 10; ++i)
    a[i] *= b[i];
}

int A[10], B[10];

int main(void) {
  #pragma hmpp foo allocate

  for (int j = 0; j < 2; ++j) {
    #pragma hmpp foo callsite
    foo(A, B);
  }

  #pragma hmpp foo release
  return 0;
}
```

Listing 18 - `atfirstcall` transfer policy example

4.7.3. `manual` transfer policy

This transfer policy is meant to be used for advanced users, to finely control transfers. No transfer is automatically performed at the callsite.

At the callsite are performed the following operations:

- Update of the pointer to the host data according to the argument passed to the callsite.
- No transfer from the host to the accelerator.
- Execution of the codelet on the accelerator.
- No transfer from the accelerator to the host.

Transfers will only be performed if `advancedload` or `delegatedstore` directives are used before or after the `callsite`.

If an `advancedload` is reached before the host pointer is known, this will trigger a runtime error.

To let the host pointer be known, the directive option `.hostdata` must be used.

In the example below, the `manual` transfer policy is used to optimize transfers.

```
#pragma hmpp foo codelet, target=CUDA, args[a].io=inout, args[*].transfer=manual
void foo(int a[10], const int b[10]) {
  for (int i = 0; i < 10; ++i)
    a[i] *= b[i];
}

int A[10], B[10];

int main(void) {
#pragma hmpp foo allocate, args[a].hostdata="A", args[b].hostdata="B"

  #pragma hmpp foo advancedload, args[a, b]

  for (int j = 0; j < 2; ++j) {
    #pragma hmpp foo callsite
    foo(A, B);
  }

  #pragma hmpp foo delegatedstore, args[a]

  #pragma hmpp foo release
  return 0;
}
```

Listing 19 - manual transfer policy example

The following transfers will be performed at the execution:

- `advancedload: A`➔`a, B`➔`b`
- `callsite:`
- `callsite:`
- `delegatedstore: A`🡐`a`
- `release`

4.7.4. `automatic` transfer policy

Transfer clause (codelet and region directives)

The automatic management of data transfers policy is intended to improve the basic performance of HMPP applications.

Quite often, data used in the codelet may not need to be synchronized between the host and the GPU, typically:

- If the host doesn't use the output of the codelet (either to read it or to modify it) until after the loop
- If some of the arguments are never modified (constant arrays, array bounds, …)

For example:

```
#pragma hmpp cod1 codelet, target=CUDA, args[c].io=inout, &
#pragma hmpp &                          args[*].transfer=auto
void k( int n,
        float alpha, float beta,
        const float a[n*n], const float b[n*n],
        float c[n*n] ) {
…
}
```

Listing 20 - automatic transfer clause in codelet definition

When "`auto`" transfer mode is activated, read and write accesses to the codelet parameters are instrumented around the callsite to trigger an automatic transfer[18]. This is done only when a modified version of an argument is needed on the host or on the GPU.

It is still possible to use `advancedLoad` and `delegatedStore` pragmas to force transfers.

Currently a parameter is automatically updated in any of the following cases:

- A parameter is written on the host.
- A function call is performed near the callsite, inside the loop.
- At the beginning or the end of the current function containing the codelet.

Automatic transfer cannot be used with asynchronous codelet execution.

Implications of automatic transfers on codelet's parameters

Since automatic transfers work by instrumenting the program statements around the callsite, codelet argument's size are also inferred from the callsite's context (NB: without automatic transfers, codelet arguments's size is inferred from the codelet's signature).

In practice, that means that in the following FORTRAN program, it is required to specify the size of the codelet argument named "`t`" since the array passed at the callsite, "`tab`", is a FORTRAN allocatable array, which size is not specified at the declaration of this array.

[18] See section 2748416.0.1073774592 Implications of automatic transfers on codelet's parameters for noteworthy implications of using automatic transfers on your HMPP program.

```
!$ hmpp <g> group, target=cuda

program tr
  integer, dimension(:), allocatable :: array
  integer :: size
  allocate(array(size))

  !$hmpp <g> allocate, args[s::t].size={size}

  !$ hmpp <g> s callsite
  call sub(size, array)

contains

  !$ hmpp <g> s codelet, args[*].transfer=auto
  subroutine sub(n, t)
    implicit none
    integer :: n
    integer, dimension(n) :: t
    !!
  end subroutine sub

end program tr

If the allocate .size option was omitted, the following error would be issued:
hmpp: [Error HP0946] tr.f90:8: Cannot deduce 'addr' for the parameter 'n' at rank #0 in codelet
's' of directive set 'g'
hmpp: [Error HP0944] tr.f90:8: Cannot deduce 'size' for the parameter 't' at rank #1 in codelet
's' of directive set 'g'
```

Likewise, in following program in C language, it is required to specify the size of the codelet argument named "`t`" since the array passed at the callsite, "`pt`", is a C pointer, which pointed memory region size is not specified at the declaration of this pointer.

```
#include <stdlib.h>
#pragma hmpp <g> group, target=CUDA
#pragma hmpp <g> s codelet, args[*].transfer=auto
void sub(int n, int t[n]) {
  /* */
}

int main (void) {
  const int size = 10;
  int *pt = NULL;
  pt = calloc(size, sizeof(int));

#pragma hmpp <g> allocate, args[s::t].size={size}

#pragma hmpp <g> s callsite
  sub(size, pt);
  return 0;
}

If the allocate .size option was omitted, the following error would be issued:
hmpp: [Error HP0946] tr.c:13: Cannot deduce 'addr' for the parameter 'n' at rank #0 in codelet
's' of directive set 'g'
hmpp: [Error HP0944] tr.c:13: Cannot deduce 'size' for the parameter 't' at rank #1 in codelet
's' of directive set 'g'
```

4.7.5. `disregard` directive

If a callsite is surrounded by functions that the user knows they have no side effects on the codelet parameters (printer or timing functions for instance), automatic transfers are limited. Modification of codelet's parameters by functions is not detected by HMPP.

This limitation can be removed with the `disregard` directive associated to the function calls. The syntax is:

```
#pragma hmpp [<group_label>]? [codelet_label] disregard args[arg_items]
```

For block of statements (case of FORTRAN for example), the following directives are also available:

To mark the beginning of the block to ignore:

```
#pragma hmpp [<group_label>]? [codelet_label] begindisregard args[arg_items]
```

To mark the end of the block to ignore:

```
#pragma hmpp [<group_label>]? [codelet_label] enddisregard args[arg_items]
```

In FORTRAN language, the equivalent of the `disregard` directive for a block of statements is:

```
!$hmpp [<group_label>]? [codelet_label] begindisregard


FORTRAN STATEMENTS


!$hmpp [<group_label>]? [codelet_label] enddisregard
```

This directive allows to inhibit data transfer from or to the GPU.

So for example, if we consider the following codelet definition where only matrix `c` has an `inout` status

```
  !$HMPP <myGRP> sgemm codelet, target=CUDA, args[c].io=inout
  SUBROUTINE sgemm(n,alpha,a,b,beta,c)
    IMPLICIT NONE
    INTEGER, INTENT(IN)    :: n
    REAL,    INTENT(IN)    :: alpha,beta
    REAL,    INTENT(IN)    :: b(n,n),a(n,n)
    REAL,    INTENT(INOUT) :: c(n,n)
    …
```

Listing 21 - disregard directive example - codelet definition (extract)

And if we consider the following code (we assume here that there are no other HMPP directives in the application):

```
  !$HMPP <MyGrp> allocate, args[sgemm::a;sgemm::b;sgemm::c].size={N,N}

  DO i=1,2
     !$HMPP <MyGrp> sgemm callsite
     call sgemm(N,alpha,a,b,beta,c_hmpp)
  END DO

  !$HMPP <MyGrp> release
```

Listing 22 - disregard directive example - callsite level

This leads to the following output[19]:

```
...group "myGRP", codelet "sgemm": Allocating input: arg[n].size=0(scalar) arg[n].const=0
...group "myGRP", codelet "sgemm": Allocating input: arg[alpha].size=0(scalar) arg[alpha].const=0
...group "myGRP", codelet "sgemm": Allocating input: arg[a].size=[128][128] arg[a].const=0
...group "myGRP", codelet "sgemm": Allocating input: arg[b].size=[128][128] arg[b].const=0
...group "myGRP", codelet "sgemm": Allocating input: arg[beta].size=0(scalar) arg[beta].const=0
...group "myGRP", codelet "sgemm": Allocating inout: arg[c].size=[128][128] arg[c].const=0
...group "myGRP", codelet "sgemm": Writing data to HWA: arg[n].size=0(scalar) arg[n].async=0
...group "myGRP", codelet "sgemm": Writing data to HWA: arg[alpha].size=0(scalar)
arg[alpha].async=0
...group "myGRP", codelet "sgemm": Writing data to HWA: arg[a].size=[128][128] arg[a].async=0
...group "myGRP", codelet "sgemm": Writing data to HWA: arg[b].size=[128][128] arg[b].async=0
...group "myGRP", codelet "sgemm": Writing data to HWA: arg[beta].size=0(scalar)
arg[beta].async=0
...group "myGRP", codelet "sgemm": Writing data to HWA: arg[c].size=[128][128] arg[c].async=0
...group "myGRP", codelet "sgemm": Starting codelet: async=0
...group "myGRP", codelet "sgemm": Reading data from HWA: arg[c].size=[128][128] arg[c].async=0
...group "myGRP", codelet "sgemm": Writing data to HWA: arg[n].size=0(scalar) arg[n].async=0
...group "myGRP", codelet "sgemm": Writing data to HWA: arg[alpha].size=0(scalar)
arg[alpha].async=0
...group "myGRP", codelet "sgemm": Writing data to HWA: arg[a].size=[128][128] arg[a].async=0
...group "myGRP", codelet "sgemm": Writing data to HWA: arg[b].size=[128][128] arg[b].async=0
...group "myGRP", codelet "sgemm": Writing data to HWA: arg[beta].size=0(scalar)
arg[beta].async=0
...group "myGRP", codelet "sgemm": Writing data to HWA: arg[c].size=[128][128] arg[c].async=0
...group "myGRP", codelet "sgemm": Starting codelet: async=0
...group "myGRP", codelet "sgemm": Reading data from HWA: arg[c].size=[128][128] arg[c].async=0
```

Figure 6 - HMPP output execution with all the transfers

On Figure 6[20], it should be noticed that at each iteration of the loop:

- All the input arguments are transferred from the CPU to the GPU (data in green);
- All the output arguments are transferred from the GPU to the CPU (data in blue).

If we modify now Listing 21, by adding the "`auto`" transfer clause as shown on Listing 23.

```
 !$HMPP <MyGrp> sgemm codelet, target=CUDA, args[c].io=inout,args[*].transfer=auto
 SUBROUTINE sgemm(n,alpha,a,b,beta,c)
   IMPLIcIT NONE
   INTEGER, INTENT(IN)    :: n
   REAL,    INTENT(IN)    :: alpha,beta
   REAL,    INTENT(IN)    :: b(n,n),a(n,n)
   REAL,    INTENT(INOUT) :: c(n,n)
    …
```

Listing 23 - Codelet definition with automatic data transfer validated

The addition of this clause leads now to the following HMPP execution:

[19] Got by setting `HMPP_VERBOSITY=9`

[20] Some details of the HMPP messages have been removed in order to improve the readability of the example.

```
...group "mygrp": Allocated 'NVIDIA GPU (CUDA Runtime 3.1)'.
...group "mygrp", codelet "sgemm": Allocating input: arg[n].size=0(scalar) arg[n].const=0
...group "mygrp", codelet "sgemm": Allocating input: arg[alpha].size=0(scalar) arg[alpha].const=0
...group "mygrp", codelet "sgemm": Allocating input: arg[a].size=[128][128] arg[a].const=0
...group "mygrp", codelet "sgemm": Allocating input: arg[b].size=[128][128] arg[b].const=0
...group "mygrp", codelet "sgemm": Allocating input: arg[beta].size=0(scalar) arg[beta].const=0
...group "mygrp", codelet "sgemm": Allocating inout: arg[c].size=[128][128] arg[c].const=0
...group "mygrp", codelet "sgemm": Writing data to HWA: arg[n].size=0(scalar) arg[n].async=0
...group "mygrp", codelet "sgemm": Writing data to HWA: arg[alpha].size=0(scalar)
arg[alpha].async=0
...group "mygrp", codelet "sgemm": Writing data to HWA: arg[a].size=[128][128] arg[a].async=0
...group "mygrp", codelet "sgemm": Writing data to HWA: arg[b].size=[128][128] arg[b].async=0
...group "mygrp", codelet "sgemm": Writing data to HWA: arg[beta].size=0(scalar)
arg[beta].async=0
...group "mygrp", codelet "sgemm": Writing data to HWA: arg[c].size=[128][128] arg[c].async=0
...group "mygrp", codelet "sgemm": Starting codelet: async=0
...group "mygrp", codelet "sgemm": Starting codelet: async=0
...group "mygrp", codelet "sgemm": All inputs may not be transfered to HW.
...group "mygrp", codelet "sgemm": Reading data from HWA: arg[c].size=[128][128] arg[c].async=0
```

Figure 7 - HMPP output execution – effect of the automatic clause on codelet's arguments

Figure 7 shows that all the useless intermediate data transfers have been removed between the two executions of the codelet.

Now if we consider the following piece of code, with the introduction of two user's function calls (`gettime`).

```
DO i=1,2
     start_time = getTime()

     !$HMPP <MyGrp> sgemm callsite
     caLL sgemm(N,alpha,a,b,beta,c_hmpp)

     stop_time = getTime()

  END DO
```

Listing 24 - disregard directive example - callsite level, introduction of two function calls

These functions may have some effects on the arguments. Without any additional information, HMPP will do this assumption. So, in this context, the execution will be identical to those illustrated Figure 6.

If you want to specify that these functions have no side effect on the codelet's arguments you can for example:

- Add a `disregard` directive before each function call in order to indicate to HMPP that these function calls must not be took into account in the data flow computation. Listing 25 illustrates a such approach

```
  DO i=1,2
     !$HMPP <MyGrp> sgemm disregard, args[*]
     start_time = getTime()

     !$HMPP <MyGrp> sgemm callsite
     caLL sgemm(N,alpha,a,b,beta,c_hmpp)

     !$HMPP <MyGrp> sgemm disregard, args[*]
     stop_time = getTime()

  END DO
```

Listing 25 - disregard directive applied on statements

Or you can also add `begindisregard` and `enddisregard` directives as Listing 26. These directives indicate to HMPP that all the statements include between the `begin` and the `end` directives must be ignored by the analysis.

```
!$HMPP <MyGrp> sgemm begindisregard, args[*]

DO i=1,2
   start_time = getTime()

   !$HMPP <MyGrp> sgemm callsite
   caLL sgemm(N,alpha,a,b,beta,c_hmpp)

   stop_time = getTime()

END DO

!$HMPP <MyGrp> sgemm enddisregard
```

Listing 26 - disregard directive applied on a block of statements

In both cases (Listing 25 and Listing 26), the execution of their code leads to the transfers shown on Figure 7.

4.8. HMPP data declaration

4.8.1. *map directive*

In a group, arguments from different codelets may share resources on the device: for instance if they refer to the same table or if one uses the result of another one. In these cases, HMPP can take advantage of using the same memory space on the device for all these arguments.

The map directive provides this feature: it maps several arguments on the device.

The notation is the following:

```
#pragma hmpp <grp_label>  map, args[arg_items]
```

The Listing 27 au-dessous illustrates the use of the map directive (in same color the "mapped" variables):

- Line 2: is the definition of a group of codelets;
- Line 3: illustrates the mapping of respectively two variables named "v1" defined in two different codelets names "init" and "dotSum".
- Line 4: illustrates the mapping of respectively two variables named "lxp" and "v2" defined in two different codelets names "init" and "dotSum".

From HMPP point of view, the introduction of these two "map" directives means that:

- The two variables "v1" will be seen as the same on the device;
- The two variables "lxp" and "v2" will be seen as the same;

Warning:

The IO status may be still different for each directive because they each refer to different particular callsite: this will determine the transfer requirements. However the union set of IO directives will define the way the map memory will be allocated!

Example: in a map: a, b
- If "a" is 'in' in codelet F1
- If "b" is 'out' in codelet F2
- Then the memory allocation will be inout (only one for both).
- "a" will be loaded before F1
- " b" will be downloaded after F2

```
…
#pragma hmpp <myGroup> group, target=CUDA // definition of the group
#pragma hmpp <myGroup> map, args[init::v1;dotSum::v1]
#pragma hmpp <myGroup> map, args[init::lxp;dotSum::v2]

#pragma hmpp <myGroup> init codelet, args[v1].io=out
void init(int n, float v1[n], float initval, float lxp[n]) {
 int j;
  for (j = 0 ; j < n ; j++)
    v1[j] = initval + lxp[j];
    …
}

#pragma hmpp <myGroup> dotSum codelet, args[v1].io=inout
void dotSum(int n, float v1[n], float v2[n])
{
  int j;
  for (j = 0 ; j < n ; j++)
    v1[j] += v2[j];
}
```

Listing 27 - map directive example

To be able to be mapped, the variables must:

- have the same dimensions;
- have the same type.

The example given below shows an illegal map association between two array variables and a scalar. In such situations HMPP will generate an error message.

```
…
#pragma hmpp <myGroup> group, target=CUDA
#pragma hmpp <myGroup> map, args[dotSum::v1;init::n]

#pragma hmpp <myGroup> init codelet, args[v1].io=out
void init(int n, float v1[n]) {
  int j;
  float val = 0.0;
  for (j = 0 ; j < n ; j++)
    v1[j] = val++;
}

#pragma hmpp <myGroup> dotSum codelet, args[v1].io=inout
void dotSum(int n, float v1[n], float v2[n])
{
 int j;
  …
}
```

Listing 28 - Illegal map directive usage

4.8.2. `mapbyname` directive

This directive is quite similar as the "`map`" directive except that the arguments to be mapped are directly specified by their name. So, the notation is the following:

```
#pragma hmpp <grp_label>  mapbyname [,variableName]+
```

To be able to be mapped, the same constraints as for the map directive are applied, the variables must have:

- the same dimensions;
- the same type.

Listing 29 shows a use of this directive. In the group "<fxx_myGroup>" all the variables called:

- "xmin" will be mapped together;
- "xmax" will be mapped together;
- etc.

```
!$hmpp <fxx_myGroup> mapbyname, xmin,xmax,ymin,ymax,zmin,zmax
```

Listing 29 - mapbyname directive example

The "mapbyname" directive is equivalent to multiple "map" directives.

```
!$hmpp <fxx_myGroup> mapbyname, xmin, xmax
```

Is equal to:

```
!$hmpp <fxx_myGroup> map, args[*::xmin]
!$hmpp <fxx_myGroup> map, args[*::xmax]
```

4.8.3. resident directive

The resident directive declares some variables as global within a group. Those variables can then be directly accessed from any codelet belonging to the group. In practice, it means that those variables will reside in the HWA memory. So they can be seen as "resident" on the HWA for the considered group.

This directive applies to the declaration statement just following it in the source code.

The syntax of this directive is:

```
#pragma hmpp <grp_label> resident
       [, args[::var_name].io=[in|out|inout|none]]*
       [, args[::var_name].size={dimsize[,dimsize]*}]*
```

Where the directive parameters are:

- <grp_label>: a unique identifier associated to all the directives that belong to the group (definition and use).
- args[::var_name].io=in|out|inout|none: indicates that the specified variables are either input, output, both or unused. By default, unqualified variables are INOUT. The specification of this parameter drives the data transfers between the host and the HWA. Furthermore, it allows some additional checks about the use of the data in HMPP applications (see chapter 4.5.1 for more details about the management of this property).
- args[::var_name].size={dimsize[,dimsize]*}: specifies the size of a non scalar parameter (an array). Each dimsize provides the size for one dimension. The set is evaluated at runtime by an allocate directive, or by all callsite and advancedload directives within the group.

The notation "::var_name" with the prefix "::", indicates an application's variable declared as resident.

Note that, unlike input or output codelet arguments, resident variables are never implicitly transferred to and from the HWA. Explicit advancedload and delegatedstore directives are required when necessary.

The Listing 30 illustrates the use of this directive. The corresponding results are presented on Listing 31.

```
#include <stdio.h>
#define SIZE 10240

// group declaration. The group label is “myGroup”
#pragma hmpp <myGroup> group, target=CUDA

// resident data declaration inside the group "MyGroup"
#pragma hmpp <myGroup> resident, args[::tab_init_on_hwa].io=out &
#pragma hmpp &                 , args[::tab_init_on_host].io=in
float tab_init_on_hwa [SIZE], tab_init_on_host[SIZE];

// declaration of the codelet "init" inside the group "MyGroup"
#pragma hmpp <myGroup> init codelet
void init(int n) {
  int j;
  float val = 0.0;
  for (j = 0 ; j < n ; j++) tab_init_on_hwa[j] = val++  ;
}

// declaration of the codelet "dotSum" inside the group "MyGroup"
#pragma hmpp <myGroup> dotSum codelet
void dotSum(int n)
{
  int j;
  for (j = 0 ; j < n ; j++) tab_init_on_hwa[j] += tab_init_on_host[j];
}

int main(int argc, char **argv)
{
  int i, m=SIZE;
  float val = 0.0;

  for (i = 0 ; i < m ; i++)  tab_init_on_host[i] = val++*2;

#pragma hmpp <myGroup> allocate // allocation of the group on the HWA

// transfer onto the HWA of the variable tab_init_on_host
#pragma hmpp <myGroup> advancedload, args[::tab_init_on_host]

#pragma hmpp <myGroup> init callsite    // call to the "init" codelet
  init(m);

#pragma hmpp <myGroup> dotSum callsite   // call to the "dotSum" codelet
  dotSum(m);

 //transfer of the data from the HWA to the CPU
#pragma hmpp <myGroup> delegatedstore, args[::tab_init_on_hwa]


#pragma hmpp <myGroup> release // release of the HWA

  // short display of the results
  for (i = 0 ; i < m ; i=i+2) {
    if ((i <= 5) || (i >=  m-5))
     printf ("tab_init_on_hwa[%d]= %4.2f \t\t tab_init_on_hwa[%d]= %4.2f \n",
             i, tab_init_on_hwa[i], i+1, tab_init_on_hwa[i+1]);
  }

  return 0;}
```

Listing 30 - resident directive example

```
$ hmpp gcc MyProgramWithResident.c -o MyProgramWithResident.hmpp
$ ./MyProgramWithResident.hmpp

tab_init_on_hwa[0]= 0.00            tab_init_on_hwa[1]= 3.00
tab_init_on_hwa[2]= 6.00            tab_init_on_hwa[3]= 9.00
tab_init_on_hwa[4]= 12.00           tab_init_on_hwa[5]= 15.00
tab_init_on_hwa[10236]= 30708.00  tab_init_on_hwa[10237]= 30711.00
tab_init_on_hwa[10238]= 30714.00  tab_init_on_hwa[10239]= 30717.00


$ gcc MyProgramWithResident.c -o MyProgramWithResident.gcc
$ ./MyProgramWithResident.gcc
tab_init_on_hwa[0]= 0.00            tab_init_on_hwa[1]= 3.00
tab_init_on_hwa[2]= 6.00            tab_init_on_hwa[3]= 9.00
tab_init_on_hwa[4]= 12.00           tab_init_on_hwa[5]= 15.00
tab_init_on_hwa[10236]= 30708.00  tab_init_on_hwa[10237]= 30711.00
tab_init_on_hwa[10238]= 30714.00  tab_init_on_hwa[10239]= 30717.00
```

Listing 31 - Results of the application described Listing 30 (with hmpp and usual compiler like gcc)

4.8.4. Data mirroring directives

HMPP currently defaults to "buffer" memory mode. Its main purpose is to save HWA memory, as its memory is typically much more limited than main memory's. There are a few downsides to "buffer" memory mode, such as:

- the fact that codelet name arguments must be used in all directives arguments (including at `callsite`),
- it makes data sharing between `callsites` very verbose and error prone (see the `map` directive and `Listing` 28 - Illegal map directive usage
- `mapbyname` directive)

Thanks to data mirroring, it is possible to refer to arguments with their host address, which allows to get rid of the two above mentioned disadvantages of "buffer" memory mode.

Data mirroring however requires data mirrors to be declared and allocated before being used.

The following example shows how, thanks to mirroring, it is now possible to decouple the pre-loading of data on the GPU with the call of the offloaded routine.

```
// mirror clause specifies that arguments will be manipulated as mirrored data
#pragma hmpp f codelet, target=CUDA, args[*].mirror, args[*].transfer=manual
void f(float a[100], float b[100]) {
  int i;
  for (i = 0; i < 100; ++i) {
    a[i] = a[i] + (b[i]);
  }
}

int main(void) {
  float x[3][100];
  int i, j;
#pragma hmpp f allocate

  for (i = 0; i < 3; ++i) {
    // Declaration, then allocation of data mirrors
    #pragma hmpp f new, data["x[i]"]
    #pragma hmpp f allocate, data["x[i]"], data["x[i]"].size={100}, &
    #pragma hmpp &           data["x[i]"].elementsize="sizeof(float)"
    // upload of data based on the address
    #pragma hmpp f advancedload, data["x[i]"]
  }
  for (i = 0; i < 3; ++i) {
    for (j = 0; j < 3; ++j) {
      if (i != j) {
        #pragma hmpp f callsite
        f(x[i], x[j]);
      }
    }
  }
  for (i = 0; i < 3; ++i) {
    // download of data based on the address
    #pragma hmpp f delegatedstore, data["x[i]"]
    #pragma hmpp f free, data["x[i]"]   // deallocation of data mirror
    #pragma hmpp f delete, data["x[i]"] // mirror descriptor release
  }
#pragma hmpp f release
  return 0;
}
```

Listing 32 - An example of data mirroring in C

As you can see, using mirrors requires the declaration and allocation of mirrors prior to being used. The first loop allocates several mirrors on the HWA from different lines of the array "x", and then uploads the mirrored data to the HWA with an `advancedload` directive.

Arguments are now referred with their host address such as `data["x[i]"]` rather than their name in the codelet they belong to.

HMPP's runtime ensures that the offloaded function uses the right data simply by comparing the address of callsite arguments with previously declared mirrors.

The verbose log that corresponds to the execution of the above C example is listed hereinafter.

```
Starting HMPPRT logging...
[     0.214183] ( 1) INFO : --> allocate <f> at mirror.c:12
[     0.214838] ( 1) INFO :     - Acquisition of grouplet 'f' (1 CUDA devices)
[     0.217071] ( 1) INFO :       0) Tesla T20 Processor
[     0.217169] ( 1) INFO : <-- allocate <f> at mirror.c:12
[     0.217339] ( 2) INFO : --> new, data <f> at mirror.c:16
[     0.217451] ( 2) INFO : <-- new, data <f> at mirror.c:16

[... more allocations]

[     0.299332] ( 1) INFO : --> advancedload, data <f> at mirror.c:20
[     0.299432] ( 1) INFO :     - Upload mirror 0x7fff60d964a0 (on device 0)
[     0.299600] ( 1) INFO : <-- advancedload, data <f> at mirror.c:20

[... more uploads ...]

[     0.299732] ( 3) INFO : --> callsite <f> at mirror.c:25
[     0.299790] ( 3) INFO :     - Call codelet 'f' (on device 0)
[     0.391331] ( 3) INFO : <-- callsite <f> at mirror.c:25
[     0.391681] ( 1) INFO : --> callsite <f> at mirror.c:25
[     0.391770] ( 1) INFO :     - Call codelet 'f' (on device 0)
[     0.391971] ( 1) INFO : <-- callsite <f> at mirror.c:25

[... more callsites ...]

[     0.393407] ( 4) INFO : --> delegatedstore, data <f> at mirror.c:32
[     0.393458] ( 4) INFO :     - Download mirror 0x7fff60d96180 (on device 0)
[     0.393579] ( 4) INFO : <-- delegatedstore, data <f> at mirror.c:32
[     0.393772] ( 1) INFO : --> free, data <f> at mirror.c:33
[     0.393861] ( 1) INFO :     - Free mirror 0x7fff60d96180 (on device 0)
[     0.394057] ( 1) INFO : <-- free, data <f> at mirror.c:33
[     0.394199] ( 3) INFO : --> delete, data <f> at mirror.c:34
[     0.394259] ( 3) INFO : <-- delete, data <f> at mirror.c:34

[... more de-allocations ...]

[     0.396549] ( 1) INFO : --> release <f> at mirror.c:36
[     0.396605] ( 1) INFO :     - Release of grouplet 'f'
[     0.397114] ( 1) INFO : <-- release <f> at mirror.c:36
```

Listing 33 - The runtime log obtained from the execution of the code from Listing 32 - An example of data mirroring in C

As you can see, thanks to data mirroring, we are able to quite simply ensure that once the callsite loop is reached, all data is already present on the HWA.

4.9. Parallel directive (Using multiple HWA devices)

HMPP can dispatch computations on multiple HWA, provided that the user has allocated the memory needed for each computation on the devices, following the “`owner compute rule`”.

The owner computes rule reads that the HWA that ends up "owning" the data (because it has been allocated on it) is the one that will carry out the computations.

In order to allocate data on a given device, all that is needed is to use the `.device=”expression”` option on an `allocate` directive of mirror allocations to let HMPP’s runtime allocate the data on the device which number equals the value of the expression. This implies that using multi-devices with HMPP requires the utilization of data mirrors (see chapter 4.8.4).

Then, you can use HMPP’s `parallel` directive on a loop scope to let HMPP dispatch the computations in parallel on the allocated HWA (note: if that directive is not used, the computations will just occur on each device alternatively, without even being used simultaneously).

The syntax of the `parallel` directive is the following:

```
!$hmpp parallel [, device="device_num"]
```

Where the directive parameters are:

- `device="device_num"`: gives the number of the device on which the data should be executed. It should not be necessary to specify it here as one should have allocated the data used within the parallel execution with the data mirroring's "`device=`" directive parameter.

The following complete example shows how dispatch computations on several devices once the data has been allocated in round-robin on several HWA.

```
#pragma hmpp f codelet, target=CUDA, args[*].mirror, args[*].transfer=manual
void f(float a[100], float b[100]) {
  int i;
  for (i = 0; i < 100; ++i) {
    a[i] = a[i] + (b[i]);
  }
}
int main(void) {
  float x[4][100];
  float y[4][100];
  int i;
#pragma hmpp f allocate

  for (i = 0; i < 4; ++i) {
    // Declaration, then allocation of data mirrors on alternative devices
    #pragma hmpp f allocate, data["&x[i][0]"], size={100}, &
    #pragma hmpp &   elementsize="sizeof(float)", device="i%2"
    #pragma hmpp f allocate, data["&y[i][0]"], size={100}, &
    #pragma hmpp &   elementsize="sizeof(float)", device="i%2"
    // upload of data based on the address
    #pragma hmpp f advancedload, data["&x[i][0]","&y[i][0]"]
  }
  #pragma hmpp parallel
  for (i = 0; i < 4; ++i) {
    #pragma hmpp f callsite
    f(&x[i][0], &y[i][0]);
  }
  for (i = 0; i < 4; ++i) {
    #pragma hmpp f delegatedstore, data["&x[i][0]"]
    #pragma hmpp f free, data["&x[i][0]","&y[i][0]"]
  }
#pragma hmpp f release
  return 0;
}
```

Listing 34 - An example of the utilization of the `parallel` directive

4.10. Regions[21] in HMPP

This section presents a set of HMPP directives to allow expressing computation for GPU as regions of code. The goal is to avoid code restructuration to build the codelet.

A region is a merge of the codelet/callsite directives. Therefore, all the attributes available for codelet or callsite directives can be used on regions directives.

[21] Be careful; do not confuse HMPP section, which refers to an array section (see chapter 4.6.4, Array section in HMPP) with HMPP region, which refers to a block of statements.

In C, the region directive must be inserted immediately before a block.

In FORTRAN, the region and the corresponding endregion directives must be inserted around a part of executable code.

The constraints for writing regions are the same as for codelets (see chapter 3.1 for more details). In addition, the control flow must remain inside the region; that is, there must not be any:

- "`return`" (in C) and "`stop`" (in FORTRAN);
- no "`break`" and "`continue`" (in C), "`cycle`" and "`exit`" (in FORTRAN) to a loop enclosing the region;
- "`goto`" to jump inside or outside the region.

We distinguish two parts in the declaration of a region: one dedicated to the codelet parameters, the other dedicated to the callsite parameters. So, the syntax for the definition of a region is the following:

In C language:

```
#pragma hmpp [<MyGroup>] [label] region
        Codelet parameters      [, args[arg_items].io=[in|out|inout|none]]*
                                [, cond = "expr"]
                                [, args[arg_items].transfer=[atcall|atfirstcall|manual|auto]]*
                                [, target=target_name[:target_name]*]
        Callsite parameters     [, args[arg_items].size={dimsize[,dimsize]*}]*
                                 [, args[arg_items].addr="expr"]*
                                [, asynchronous]?
                                [, private=[arg_items]]*
    {

C BLOCK STATEMENTS

     }
```

In FORTRAN language:

```
!$hmpp [<MyGroup>] [label] region
          Codelet parameters    [, args[arg_items].io=[in|out|inout|none]]*
                                [, cond = "expr"]
                                [, args[arg_items].transfer=[atcall|atfirstcall|manual|auto]]*
                                [, target=target_name[:target_name]*]
          Callsite parameters   [, args[arg_items].size={dimsize[,dimsize]*}]*
                                 [, args[arg_items].addr="expr"]*
                                [, asynchronous]?
                                [, private=[arg_items]]*


FORTRAN STATEMENTS


!$hmpp [<MyGroup>] [label] endregion
```

Where the directive parameters are:

- All the codelet parameters refer to parameters available for the codelet directive (see chapter 4.5.1, `codelet` directive)
- All the callsite parameters refer to parameters available for the callsite directive (see chapter 4.5.3, `callsite` directive);

- `private:` specifies the variables that should be re-declared to be only used in the region. Typically, this parameter applies for loop induction variables. The HMPP `private` keyword usage is identical to the OpenMP `private` keyword.

Since HMPP 2.4, HMPP provides users with an automatic detection of the input and output data. So, by default, variables that are only read are seen as input (`IN` intent) while those that are written are seen are both input and output (`INOUT` intent). To avoid useless transfer, users can override intents determined by HMPP using the `.io` attributes.

HMPP offers the "`--io-report`" option to display the intents detected by HMPP.

For instance, with the following region definition:

```
#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;
}
```

The "`--io-report`" option provides the output below:

```
$ 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 INOUT
      Parameter 'a' has intent IN
      Parameter 'b' has intent INOUT
      Parameter 'r' has intent INOUT
```

As can be seen, `r` is detected as both an input and output.

Since we know that `r` is only written, its intent property can be force to output only and thus avoiding a needless transfer from the host to the GPU, as follows:

```
#pragma hmpp <g> foo region, args[r].io=out
{
   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;
}
```

The following restrictions apply:

- Regions cannot be nested;
- Asynchronous region must have at least a label;
- Only "`hmppcg`" directives are allowed inside the region.

Warning:

In FORTRAN, all variables accessed in a region must have their declarations in the same compilation unit. That is, at the present time, you cannot create a region where a variable is defined in an external module.

4.11. External and native[22] functions

Warning:

HMPP 3.0.0 only supports external functions.

For convenience, the text referencing external and native functions was left in state. In HMPP 3.0.0, only the external function is to be considered.

Automatic inlining of functions called within codelets was already supported by HMPP. HMPP 2.5 introduces new mechanisms to support direct calls to functions in codelets.

Functions that can be called from codelets are either hand-written CUDA/OPENCL native functions or external C/FORTRAN functions. In codelets generated by HMPP, these functions can be seen as CUDA `__device__` functions called in CUDA kernels. ***External and native functions are not CUDA kernels or library functions such as CUBLAS.***

An "`external`" function is a function defined in the source code (C or FORTRAN), not necessarily in the same file, and called within a codelet or a region. In this context HMPP automatically generates its CUDA or OpenCL version in an XML file.

External functions can be compiled separately from the files containing codelets or regions that call it. This avoids code duplication when a function is used in several HMPP codelets or regions.

External functions are declared using the following HMPP directives, placed just before the function definition:

In C:

```
#pragma hmpp function, target=list_of_targets
```

In FORTRAN:

```
!$hmpp function, target=list_of_target
```

The use of native and external functions requires HMPPCG directives. So complete description of external and native functions is detailed in [R3].

[22] Native functions are not available in HMPP 3.0.0. These one can be used with HMPP 2.5.x. This feature will be restored in a future version.

5. Supported Languages

The HMPP codelet generators do not handle full languages for C and FORTRAN. This restriction aims at ensuring portability of the code on most HWAs (for instance, allowing pointer arithmetic in C language would forbid generation of code for many hardware platforms) and also performance.

Moreover, it should be noted that in addition to the restrictions bring by HMPP, hardware constructors do not offer for all targets a full support of the language. End-users should pay attention to the current limitations of the hardware accelerators that they want to use by consulting hardware constructor's website.

5.1. Input C Code

As mentioned above, the HMPP codelet generators do not handle the full C language. The HMPP codelet generators take C99 input code so the array size can be specified in the parameter declaration. The remainder of this section is organized as follow.

- Section 5.1.1 describes the valid C constructs for HMPP;
- Section 5.1.2 shows how codelet parameter data sizes are addressed by the HMPP codelet generator.

5.1.1. Supported C Language Constructs

In this section we describe the language constructs which are supported by the HMPP codelet generators. The codelet prototype is preferably in C99 style in which all array sizes are specified in the declaration (see Section 5.1.2). Typically a codelet code looks like:

```
void simplefunc(int n, float s1[1], float v2[n], float v3[n]){
  int i;
  float r = s1[0];
  for (i = 0 ; i < n ; i++) {
    r += v2[i] * v3[i];
  }
  s1[0] = r;
}
```

Listing 35 - C codelet code example

Below are the language constructs supported by the HMPP codelet generators. If a construct is not supported, the HMPP codelet generator issues an error message and no codelet implementation is produced.

1. Atomic data types
   a. `char`, `unsigned char`, `short`, `unsigned short`, `integer`, `long`, `long long`, `unsigned integer`, `unsigned long`, `unsigned long long;`
   b. `float`, `double, complex`
2. Data structures
3. Language constructs
   a. All arithmetic, shift and comparison operations.
   b. `for` loops with simple induction variables. The following styles of `for` loops are supported:

```
for (i=lowbound ; i<highbound ; i++){...}
for (i=lowbound ; i<=highbound ; i++){...}
for (i=lowbound ; i<=highbound ; i = i+s){...}
```

Where `lowbound` and `highbound` are invariant in the loop. The step value `s` is an integer constant. Furthermore, the induction variable `i` cannot be modified in the loop body.

4. Conditional statements `if() ... else ....`
5. Calls to intrinsic (see Section **Erreur ! Source du renvoi introuvable.** for the list of supported intrinsic) and functions.

The following constructs are not supported in a codelet:

1. `switch` and `case` statements.
2. Function pointers.

Warning:Initialization of structure using C99 style is not supported.

5.1.2. Parameter Passing Convention for C Codelets

To implement the communications between the host and HWAs, it is necessary to provide the HMPP API runtime with the size of the data to be transferred to/from the HWAs. Listing 36 illustrates this.

Warning:By default, HMPP assumes that no aliasing exists between codelet parameters.

```
/* C99 syntax */
#pragma hmpp csmain codelet, args[a].io=in, &
#pragma hmpp &               args[b].io=in, &
#pragma hmpp &               args[r].io=out
void csmain(unsigned int S, float r[S], float a[S], float b[S]) {
  unsigned i;
  for (i=0 ; i<S ; i++){
    r[i] = b[i] / sqrt(a[i]);
  }
}
```

Listing 36 - Parameter data size passing using C99 for codelets

5.1.3. Inlined functions

HMPP supports the inlining of functions with the following restrictions:

- The definition of the inlined function must be available in the compilation scope of the codelet;
- The inlined function must not have any HMPP directives;
- The inlined function must not be recursive;
- The inlined function must not access global variables

5.1.4. Atomic intrinsic functions

HMPP supports the following atomic functions inherited from gcc and icc compilers.

Name	Type
`type __sync_fetch_and_add(type *ptr, type value)`	`int or unsigned int`
`type __sync_fetch_and_sub(type *ptr, type value)`	`int or unsigned int`
`type __sync_fetch_and_or(type *ptr, type value)`	`int or unsigned int`
`type __sync_fetch_and_and(type *ptr, type value)`	`int or unsigned int`
`type __sync_fetch_and_xor(type *ptr, type value)`	`int or unsigned int`

Table 5 - Supported atomic intrinsic functions

These built-ins functions perform operation atomically according to their definition and return the value that had previously been in memory.

5.2. Input FORTRAN Code

The HMPP codelet generators do not support the full FORTRAN language. The subset taken into account is similar to the C subset described in Chapter 5.1. The remainder of this section is organized as follow:

- Section 5.2.1 describes the supported FORTRAN language constructs.
- Section 5.2.2 indicates how codelet parameter data sizes are addressed by the HMPP codelet generators.

5.2.1. Supported FORTRAN Language Constructs

In this section we describe the language constructs that are supported by the HMPP codelet generators. Typically a codelet code looks like:

```
!$hmpp simple codelet, target=CUDA
SUBROUTINE simple(n,m,inv,inm,outv)
  IMPLICIT NONE
  INTEGER, INTENT(IN) :: n,m
  REAL, INTENT(IN) :: inv(n)
  REAL, INTENT(IN) :: inm(m,n)
  REAL, INTENT(OUT) :: outv(m,n)
  INTEGER :: i,j


   DO j = 1,n
     DO i = 1,m
       outv(i,j) = inv(j) * inm(i,j)
     ENDDO
   ENDDO

END SUBROUTINE simple
```

Listing 37 - FORTRAN codelet code example

The language constructs presented below are the ones supported by the Fortran HMPP codelet generators. If a construct is not supported, the code generator issues an error and no codelet is produced.

Explicit declaration in codelet

The "`IMPLICIT NONE`" statement is required in FORTRAN codelet. All variables must be explicitly declared in FORTRAN codelets.

Supported Data Types

The table below summarizes the scalar data types that are supported within the codelets and shows how they are interpreted.

Name	Type	Semantic
ABS(x)	REAL*n or INTEGER*n	Absolute value
LOG(n)	REAL*n	Natural logarithmic
LOG10(n)	REAL*n	Base-10 logarithmic function
SQRT(n)	REAL*n	Square root
MIN(a,b,...)	REAL*n or INTEGER*n	Minimum
MAX(a,b,...)	REAL*n or INTEGER*n	Maximum
MOD(a,b)	INTEGER*n	a modulo b
EXP(a)	REAL*n	Base-E exponential
COS(a)	REAL*n	Cosine
SIN(a)	REAL*n	Sine
TAN(a)	REAL*n	Tangent
ACOS(a)	REAL*n	Arc-Cosine
ASIN(a)	REAL*n	Arc-Sine
ATAN(a)	REAL*n	Arc-Tangent
COSH(a)	REAL*n	Hyperbolic Cosine
SINH(a)	REAL*n	Hyperbolic Sine
TANH(a)	REAL*n	Hyperbolic Tangent
ACOSH(a)	REAL*n	Inverse Hyperbolic Cosine
ASINH(a)	REAL*n	Inverse Hyperbolic Sine
ATANH(a)	REAL*n	Inverse Hyperbolic Tangent
IAND(a,b)	INTEGER*n	Bitwise AND
IOR(a,b)	INTEGER*n	Bitwise OR
IEOR(a,b)	INTEGER*n	Bitwise Exclusive-OR
NOT(a)	INTEGER*n	Bitwise NOT
REAL(a)		Convert a to REAL
DBLE(a)		Convert a to DOUBLE PRECISION (i.e. REAL(8))
INT(a)		Convert a to INTEGER
INT1(a)		Convert a to INTEGER(1)
INT2(a)		Convert a to INTEGER(2)
INT4(a)		Convert a to INTEGER(4)
INT8(a)		Convert a to INTEGER(8)

Table 6 - Supported FORTRAN data types

Current restrictions:

- The `KIND` of all types is hard-coded to the values used by most FORTRAN compilers. In the future, they will be configurable for each FORTRAN compiler,
- User defined types via the `TYPE` statements are allowed with several restrictions,
- The `CHARACTER` type and the character constants are only allowed for `LEN=1`. Virtually no operation except comparison is allowed on characters so they are of limited usage except when passed as arguments to the codelet.

Using Fortran's User defined types (UDT) in Codelets

The Alignment and Size of All UDT Members Must Be Known at Compile Time

Scalar members are always allowed:

```
TYPE Sample
      INTEGER :: x,y,z
      REAL :: a,b,c
END TYPE Sample
```

Array members are allowed assuming that their dimensions are known by the HMPP compiler. In practice, this means that the INTEGER expressions used to specify the array member shapes should only contain literal constants and scalar PARAMETERs known by HMPP:

```
       INTEGER, PARAMETER :: N=100, P=200
       TYPE Sample
              INTEGER :: x(4,N+1)
              REAL :: y(N:P)
       END TYPE Sample
```

All basic FORTRAN types are supported (`INTEGER`, `LOGICAL`, `REAL`, `COMPLEX` and `CHARACTER` of any kinds).

Restriction:

The `CHARACTER` type is only supported with a `LEN` of `1`.

Members can also be of another user-defined type assuming, of course, that a type does not attempt to include itself directly or indirectly.

```
       TYPE T1
              INTEGER :: a,b,c
       END TYPE T1
       TYPE T2
              TYPE(T1) :: x,y
       END TYPE T2
       TYPE T3
              TYPE(T1) :: x,y
              TYPE(T3) : z           !!!!! ILLEGAL
       END TYPE T3
```

Members with the POINTER or the ALLOCATABLE Attribute Are Not Allowed

`POINTER` and `ALLOCATABLE` imply a reference to a memory area which, in the general case, is not managed by HMPP (the host and the HMPP target typically have distinct memory spaces) except in the situation where the structure is defined as a codelet's parameter and the data are not accessed in the codelet.

In the case where the pointer is defined as a local variable of the codelet, this construction is not supported by HMPP.

Local ALLOCATABLE arrays are supported in HMPP 3.x.

UDT Can Be Imported From Modules

The limitations are the same than for PARAMETER values imported from modules: the Fortran file defining the module must have been previously compiled with HMPP.

```
       MODULE MyTypes
          TYPE Point
              REAL :: x,y,z
          END TYPE Point
       END MODULE MyTypes
       ...
       USE MyTypes
       ...
       !$hmpp project codelet, target=CUDA
       SUBROUTINE project(points)
        TYPE(Point), INTENT(IN) :: points
        ...
       END SUBROUTINE project(points)
```

Known Limitations to the support of user-defined-types

Intel Compiler and SEQUENCE

A UDT declaration may start by a SEQUENCE statement to indicate that the compiler is not allowed to reorder the members. The FORTRAN standard does not clearly specify the semantic of the SEQUENCE statement (or of its absence). In practice, it does seem to have any effect in most compilers with the exception of the Intel compiler (ifort) where SEQUENCE removes all padding normally inserted to meet the alignment constraints of the members.

Let's consider, for example, the following UDT declaration:

```
TYPE Data
       SEQUENCE
       INTEGER(1) :: a      ! 8 bit integer at byte offset 1
       REAL(4) :: b         ! 32bit real at byte offset 2
END TYPE Data
```

The overall size of this UDT is 5 bytes. Without the SEQUENCE statement, a padding of 3 bytes would be inserted between the members `a` and `b`, and the overall size would be 8 bytes.

Accessing misaligned data-types is slower but legal on Intel processors. This is not the case on most of the HMPP target (especially GPUs where misaligned accesses are illegal).

For that reason, the SEQUENCE statement is not supported in HMPP codelets when using the Intel FORTRAN compiler.

Alignment of COMPLEX Data in CUDA

In the current version of HMPP, the COMPLEX FORTRAN types are implemented using the CUDA native types `float2` and `double2` that respectively represent a pair of 'float' and a pair of 'double'.

Unfortunately, `float2` and `double2` have different alignment constraints than their FORTRAN counterparts (e.g. float2 are aligned to multiples of 8 bytes while `COMPLEX(4)` are typically aligned like `REAL(4)` on multiples of 4 bytes). The specific alignment constraints of `float2` and `double2` can increase the performance of regular arrays of complex elements but as a side effect, they also break the compatibility between the implementation of UDT on the host and on the CUDA target.

Consider, for example, the following type:

```
TYPE Value
       REAL :: x
       COMPLEX :: y
       REAL  :: z
END TYPE Value
```

On most host FORTRAN compilers, that structure is implemented by:

- a 4 bytes REAL at offset 0
- a 8 bytes COMPLEX at offset 4
- a 4 bytes REAL at offset 12

The current implementation in HMPP CUDA is different:

- A 4 bytes 'float' at offset 0
- A 4 bytes padding at offset 4
- A 8 bytes 'float2' at offset 8
- A 4 bytes 'float' at offset 16
- A 4 bytes padding at offset 20 (to make the structure size a multiple of 8, the float2 alignment size)

In practice, UDT containing COMPLEX members are still possible in HMPP/CUDA if and only if those members are aligned on the host to a multiple of the COMPLEX type.

For instance, the previous UDT can be made compatible with CUDA by inserting some dummy padding members before the misaligned complex and at the end of the type.

```
TYPE Value
       REAL :: x
       INTEGER(4) :: padding1
       COMPLEX :: y
       REAL  :: z
       INTEGER(4) :: padding2
END TYPE Value
```

In future version, this manual padding will not be strictly required but may be recommended to improve performances.

Declarations

Declarations can be provided using the old F77 or the new F90 form:

```
INTEGER    a,b    ! F77 form
INTEGER :: c,d    ! F90 form
```

The attribute DIMENSION can also be used to specify array shapes:

```
INTEGER               :: A(10)
INTEGER,DIMENSION(10) :: B
```

Parameters

`PARAMETER` statements and attributes are supported for scalar objects only.

```
INTEGER, PARAMETER :: N=42
INTEGER M
PARAMETER ( M = 42 )
```

Inlined functions

HMPP supports the inlining of functions with the same restrictions as for C language (see chapter 0).

Intrinsic functions

Intrinsic functions used in codelets must have been declared through the use of the INTRINSIC FORTRAN statement. The example below illustrates the use of intrinsic functions in FORTRAN codelets.

```
…
REAL(8),DIMENSION(N) :: V
real(8),dimension(N,N) :: Loc
INTEGER :: J
INTRINSIC :: LOG, COS, SIN
…
```

Other Type Attributes and Declarations

Most type attributes introduced by Fortran90 are currently not supported in codelets (`POINTER`, `VOLATILE`, `TARGET`, ...). A noticeable exception is `INTENT` which is in fact recommended for all codelet arguments.

`COMMON`, `EQUIVALENCE`, `BLOCKDATA` and all declaration statements that may create aliasing between variables are not allowed in codelets.

Arrays

Arrays as codelet arguments

Array bounds of arguments should be fully specified using constants or other scalar integer arguments of the codelet.

Current restrictions:

- Scalar integer arguments used to specify an array bound shall not be modified within the codelet. Ideally, they should have the `INTENT(IN)` attribute,
- Scalar integer arguments used to specify an array bound must appear before that array in the argument list,

Below is a typical example:

```
SUBROUTINE codelet(m,n,A,B,C)
  INTEGER, INTENT(IN)    :: m,n
  INTEGER, INTENT(INOUT) :: A(100), B(m,n), C(0:m*n-1)
  ...
END SUBROUTINE
```

Listing 38 - FORTRAN array declaration in codelet

The following forms of arrays are not allowed:

- Assumed-size array arguments as in `A(*)` or `B(100,*)`
- Assumed-shape and deferred-shape array arguments as in `A(:)` or `B(3:) since the upper bound is not specified`
- `Arrays with an ALLOCATABLE or POINTER attribute`

Remark: Array arguments of the form `A(:m)` are allowed since their lower bound are by default equal to one.

Arrays as local variables in codelet

Two kinds of local arrays are allowed in codelets:

- Arrays whose shape is entirely specified using constants or integer arguments of the codelet.
- Arrays with the ALLOCATABLE attribute

Below is a typical example:

```
SUBROUTINE codelet(m,n,A)
  INTEGER, INTENT(IN)    :: m,n
  ...
  REAL :: TMP1(m,0:n+1)
  REAL, ALLOCATABLE :: TMP2(:,:)
  ...
END SUBROUTINE
```

Listing 39 - Local FORTRAN arrays in codelet

The `ALLOCATE` and `DEALLOCATE` statements are allowed within codelet to manage arrays with the `ALLOCATABLE` attribute. However, they can only take place outside the gridified loops.

The most frequently used implicit functions are supported:

- LBOUND
- UBOUND
- SIZE

`IF` statements

The following forms of `IF` statements are supported:

1. `IF .. ENDIF` constructs optionally with `ELSE IF` and `ELSE`:

```
IF (A>B) THEN
  C = 1
ELSE IF (A<B) THEN
  C = -1
ELSE
  C = 0
ENDIF
```

2. Logical `IF` statements:

```
IF (A==B) C=0
```

Current restrictions:

- `SELECT CASE` constructs are currently not supported.
- `GOTO`s are not supported as well as arithmetic `IF` statements that are in fact disguised `GOTO`s.

Loops

The following forms of loops are supported:

1. `DO` statements with index, start, end and an optional step. The index and all 3 expressions shall be of type integer.
2. `DO WHILE` statements;
3. Standalone `DO` – so a potentially infinite loop.

A `DO` construct must be terminated by an `ENDDO` statement. The old F77 form using a termination label is not allowed. `EXIT` and `CYCLE` statements are allowed within `DO` constructs.

Current restrictions:

- The step, if any, must be a simple constant (such as 1 or -2).
- No loop name shall be specified to an `EXIT` or `CYCLE` statement. They are applied to the first outer loop.
- The computation of the number of iterations in a loop of the form (a) is assumed not to overflow when computed using the type of the index. In practice, e.g. for `INTEGER*4`, the number of iterations shall not be greater than `2*31-1 = 2147483647`.

Modules

HMPP brings a preliminary support of FORTRAN modules. The objective is to provide users with the most frequently constructions used in FORTRAN applications. Thus, scalar `PARAMETER` variables of types INTEGER, LOGICAL, REAL and COMPLEX defined in modules can be directly used in HMPP codelets.

However, this first implementation mainly focuses on `INTEGER` parameters. Thus, the following operations are supported on `INTEGER` type only:

- Constant definitions. Evaluation of expressions is supported for the usual INTEGER arithmetic operators "`+, -, *, /`".

```
MODULE foo
  INTEGER, PARAMETER :: N=24, M=5
  INTEGER, PARAMETER :: P= ((N+1)*(M-5))/(M+N)
END MODULE foo
```

- INTEGER comparison and LOGICAL operators (.OR., .AND., .EQ., …)

```
MODULE foo
  INTEGER, PARAMETER :: M = 34
  INTEGER, PARAMETER :: N = 22
  LOGICAL, PARAMETER :: M_IS_BIGGER = M>N
  LOGICAL, PARAMETER :: M_EQUALS_N = M .EQ. N
  LOGICAL, PARAMETER :: DEBUG = .TRUE.
  LOGICAL, PARAMETER :: M_IS_SMALLER = .NOT. (M_IS_BIGGER .OR. M_EQUAL_N)
END MODULE foo
```

- Intrinsic functions to query type kind information (`SELECTED_INT_KIND`, `SELECTED_REAL_KIND` and `KIND`)

```
MODULE foo
  INTEGER, PARAMETER :: INT4 = SELECTED_INT_KIND(4)
  INTEGER, PARAMETER :: INT10 = SELECTED_INT_KIND(10)
  INTEGER, PARAMETER :: INT14 = SELECTED_INT_KIND(14)

  INTEGER, PARAMETER :: FLOAT_4_7 = SELECTED_REAL_KIND(4,7)
  INTEGER, PARAMETER :: FLOAT_P10 = SELECTED_REAL_KIND(P=10)
  INTEGER, PARAMETER :: FLOAT_R20 = SELECTED_REAL_KIND(R=40)

  INTEGER, PARAMETER :: FLOAT  = KIND(1.0E0)
  INTEGER, PARAMETER :: DOUBLE = KIND(1.0D0)
END MODULE foo
```

Because of the difficulty to ensure consistent rounding in floating point arithmetic, operations on `REAL` or `COMPLEX` data types are not yet supported. It is however possible to define parameters of REAL or COMPLEX types as long as their expressions only contain:

- `REAL` constant (e.g. `1.2, 1.2D0, 1.2_4, 1.2_INT4`)
- `COMPLEX` constant;
- Unary operator `"-"`;
- Parenthesis;
- References to other parameters of the same type.

REAL conversions whether they are implicit or explicit are not supported. In practice that means that the expression must be of the exact same type than the parameter. For instance, the example below is correct if we assume that the default `REAL` kind is 4:

```
REAL(4), PARAMETER :: X1 = 3.1415
REAL   , PARAMETER :: X2 = 3.1415_4
```

However, the following equivalent declarations containing an implicit and an explicit cast to `REAL(8)` will not be able to be evaluated:

```
REAL(8), PARAMETER :: Y1 = 3.1415
REAL(8), PARAMETER :: Y2 = REAL(3.1415,kind=8)
```

In practice, one could write the declaration which is similar even though it is not semantically equivalent:

```
  REAL(8), PARAMETER :: Y =3.1415_8
```

Note: FORTRAN module support will be improved in future releases, so some of these limitations should be removed.

Operations

Arithmetic operations are currently limited to scalars. Support for arrays should be available in future releases.

All native operators are supported:

- Arithmetic: `+ - / * **;`
- Comparison: `> < >= < == /=` (and their 'dotted' forms: `.GT. .LT.` etc.);
- Logical: `.NOT. .AND. .OR. .EQV. .NEQV.`

Function Calls

Only calls to intrinsic functions listed below are supported. All arguments should be of scalar type.

Name	**Type**	**Semantic**
ABS(x)	REAL*n or INTEGER*n	Absolute value
LOG(n)	REAL*n	Natural logarithmic
LOG10(n)	REAL*n	Base-10 logarithmic function
SQRT(n)	REAL*n	Square root
MIN(a,b,...)	REAL*n or INTEGER*n	Minimum
MAX(a,b,...)	REAL*n or INTEGER*n	Maximum
MOD(a,b)	INTEGER*n	a modulo b
EXP(a)	REAL*n	Base-E exponential
COS(a)	REAL*n	Cosine
SIN(a)	REAL*n	Sine
TAN(a)	REAL*n	Tangent
ACOS(a)	REAL*n	Arc-Cosine
ASIN(a)	REAL*n	Arc-Sine
ATAN(a)	REAL*n	Arc-Tangent
COSH(a)	REAL*n	Hyperbolic Cosine
SINH(a)	REAL*n	Hyperbolic Sine
TANH(a)	REAL*n	Hyperbolic Tangent
ACOSH(a)	REAL*n	Inverse Hyperbolic Cosine
ASINH(a)	REAL*n	Inverse Hyperbolic Sine
ATANH(a)	REAL*n	Inverse Hyperbolic Tangent
IAND(a,b)	INTEGER*n	Bitwise AND
IOR(a,b)	INTEGER*n	Bitwise OR
IEOR(a,b)	INTEGER*n	Bitwise Exclusive-OR
NOT(a)	INTEGER*n	Bitwise NOT
REAL(a)		Convert a to REAL
DBLE(a)		Convert a to DOUBLE PRECISION (i.e. REAL(8))
INT(a)		Convert a to INTEGER
INT1(a)		Convert a to INTEGER(1)
INT2(a)		Convert a to INTEGER(2)
INT4(a)		Convert a to INTEGER(4)
INT8(a)		Convert a to INTEGER(8)

Table 7 - Supported Intrinsic functions

Warning

In FORTRAN, local variables can be stored in global memory and be initialized at startup. Then they keep their value between function calls. This is not the case in codelets where variable declared locally are assumed to be strictly local (as in C).

5.2.2. Unsupported statements in codelet

The following statements are not supported in HMPP Fortran codelets:

- WHERE, SELECT, FORALL, GOTO, CONTAINS, INCLUDE;
- I/O statements: OPEN, CLOSE, …
- Memory statements: =>,
- Arithmetic if

5.2.3. Parameter Passing Convention for FORTRAN codelets

To implement the communication between the host and the HWAs, it is necessary to provide the HMPP runtime with the size of the data to be transferred to/from the HWAs. This is performed using the FORTRAN syntax with the array bound specified as an expression of the codelet parameters as shown in the example presented in Section 5.2.1. In other words, a parameter declaration such as `A(*)` is not supported. The `INTENT(IN|INOUT|OUT)` clause is mandatory.

5.2.4. Known limitations

HMPP FORTRAN parser should accept most of the syntaxes described in the F2003 norm.

However, the following F2003 syntaxes are known to be unsupported even outside codelets:

CLASS, EXTENDS, PASS and other TYPE-related features introduced in F2003

- The ENUM construct.
- The SELECT TYPE construct.
- The ASSOCIATE construct.

6. HMPP Codelet Generators

The HMPP codelet generators are used by the HMPP compilers to generate HWA implementations of the codelets.

HMPP includes different generators according to the considered architecture:

- CUDA for NVIDIA architecture;
- OPENCL for NVIDIA or AMD ATI Stream architecture.

It should be noted that HMPP codelet generators are based on the state-of-the-art of SDK and drivers marketed by hardware constructors. Thus HMPP inherits same limitations.

Warning:

HMPP 3.0.0 only supports the CUDA target for codelet generation. The OpenCL target will be supported current Q1 2012 (currently available in HMPP 2.5.x).

For convenience, the text referencing the two targets was left in state. In HMPP 3.0.0, only the CUDA target is to be considered.

6.1. CUDA Generator

This generator produces CUDA implementation of HMPP codelets to be executed on NVIDIA GPUs.

This generator is used when the target `CUDA` is specified as shown in the following codelet declaration:

```
C:       #pragma hmpp mycodelet codelet, args[vout].io=inout, target=CUDA
FORTRAN: !hmpp mycodelet codelet, args[vout].io=inout, target=CUDA
```

6.2. OpenCL Generator

This generator produces OpenCL implementation of HMPP codelets to be executed on NVIDIA GPUs as well as AMD ATI Stream GPU supporting OpenCL framework.

This generator is used when the target `OPENCL` is specified as shown in the following codelet declaration:

```
C:       #pragma hmpp mycodelet codelet, args[vout].io=inout, target=OPENCL
FORTRAN: !hmpp mycodelet codelet, args[vout].io=inout, target=OPENCL
```

6.3. Naming Convention

To be correctly handled by the different tools of the HMPP workbench, codelet files (either hand-written or automatically generated) must respect a specific naming convention. The general name format of HMPP codelets and libraries are described below.

6.3.1. CUDA Codelet Generator

With the CUDA keyword specified, generated file names follow the rules below.

Generated target source code

The generated file has the following name:

```
[label]_[target].[ext].[target_ext]
```

Where:

- `[label]` is:
  - The label of the group in the case of a group,
  - The label of the codelet in the case of a single codelet,
  - The name of the function in the case where no label is defined.
- `[target]` is “`cuda`” in the case of CUDA.
- `[ext]` is the file extension. Value is :
  - “`hmg`”: in the case of a group;
  - “`hmc`”: in case of codelet only
  - “`hmf`”: in case of codelet without any labels.
- `[target_ext]: is “cu”` in the case of CUDA.

Generated library

Following the generation of the source code, a dynamic library is generated to be loaded by the HMPP runtime.

The generated dynamic library has the following name:

```
[label]_[target].[ext]
```

Where:

- `[label]` is:
  - The label of the group in the case of a group,
  - The label of the codelet in the case of a single codelet,
  - The name of the function in the case where no label is defined.
- `[target]` is “`cuda`” in the case of CUDA.
- `[ext]` is the file extension. Value is :
  - “`hmg`”: in the case of a group;
  - “`hmc`”: in case of codelet only
  - “`hmf`”: in case of codelet without any labels.

So with the previous rules, for the following code:

```
#pragma hmpp rpclabel codelet target=CUDA, ...
void myFunctionToSpeedup(float *in,float *out){
.... codelet body...
}
```

The source codelet file will be `rpclabel_cuda.hmc.cu` and the corresponding library file will be `rpclabel_cuda.hmc`

6.3.2. OPENCL Codelet Generator

Unlike the CUDA target, two files are generated for the OPENCL target:

- One dedicated to the initialization of the openCL context and which may be seen as a wrapper for the launch of the opencl kernel. In case of group, this file is named:

```
[grp_label]_[target].[ext]
```

- The other corresponding to the kernel to execute on the HWA. In case of group, this file is named:

```
[grp_label]_[target].cc-kernels.[ext]
```

So, the preceding rules still apply but with the following modifications:

- `[target]` is opencl,
- `[ext]` is “cc” for the wrapper file and “.cl” for the kernel.

So for the following code:

```
#pragma hmpp rpclabel codelet target=CUDA, ...
void myFunctionToSpeedup(float *in,float *out){
.... codelet body...
}
```

The generated files will be:

- The source codelet file will be `rpclabel_opencl.cc;`
- The kernel file will be `rpclabel_opencl.cc-kernel.cl;`
- And the corresponding library file will be rpclabel_opencl.so.

7. Compiling HMPP Applications

The HMPP development workbench provides developers with HMPP compilers in order to easily build HMPP applications. HMPP compilers are available for C and FORTRAN[23]. They are used:

- to preprocess HMPP annotated applications,
- to extract and to generate HWA codelets,
- and finally to compile and link the HMPP application.

To know the list of supported Operating System and compilers:

- For Linux platform, please refer to[R5];

Warning:

We introduce in this section the main concepts concerning the compilation of HMPP application. For some historical reasons and to keep readability, examples are mainly given for Linux platforms.

HMPP 3.0.0 only supports Linux platform. Windows OS will be supported current Q2 2012 (currently available in HMPP 2.5.x).

7.1. Overview

In terms of use, the HMPP compiler workflow is really close to traditional compilers. However, as illustrated in Figure 8, we can distinguish two main paths:

- The left one (in Figure 8) is dedicated to the compilation of the *main* application which will be executed on the host processor (as in traditional compilers). In this case, we will designate the compiler used under the name *host compiler*,
- The right one (in Figure 8) is dedicated to the codelet generation and compilation. The codelet are generated under the form of shared libraries in order to be loaded by the HMPP runtime during the execution of the application. In this case, we will designate the compilers under the name of "*HMPP Codelet Generator"* for the generation of the codelets and the "*hardware vendor compilers"* for the compilation of the codelet using the provided hardware vendor tools.

[23] FORTRAN compilers are available on Linux platforms only.

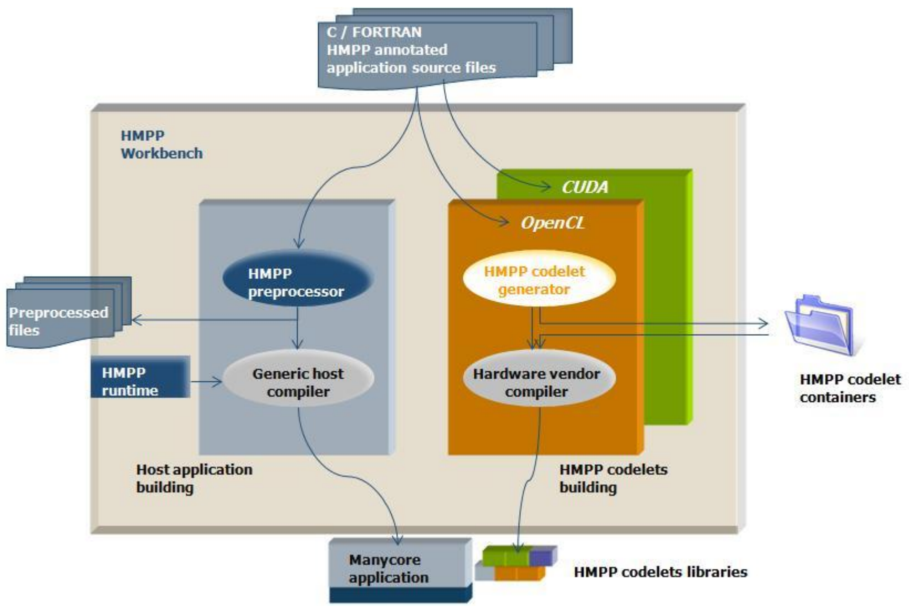


Figure 8 - HMPP compiler workflow

Compiling a HMPP program is done by using the `hmpp` command followed by the appropriate compiler depending on the considered language (C or FORTRAN):

```
$ hmpp gcc program.c -o program.exe
```

Or:

```
$ hmpp ifort  program.f90 -o program.exe
```

Or

```
$ hmpp cl  program.c -o program.exe
```

Like with usual compilers, the default output file name is `a.out`.

The `hmpp` commands successively runs the HMPP preprocessor to process the directives by inserting calls to the HMPP runtime and then invoke the user's specified native compiler to produce the application executable.

`hmpp` extracts the marked codelets from the application sources and generates their hardware accelerated implementation as shared libraries with the appropriate HMPP codelet generator.

7.2. Common Command Line Parameters

The HMPP compiler runs as follow:

```
$ hmpp [HMPP_OPTIONS] HOST_COMPILER [HOST_COMPILER_OPTIONS] files
```

Here is the list of the available command line options for HMPP compiler.

7.2.1. General Options

General options are:

- `-t, --temp DIRNAME`: sets the temporary directory (default is /tmp),

- `-k, --keep`: does not remove temporary files,
- `-d[x]*, --debug`: set HMPP verbosity. A numerical value can be specified to increase the level of the verbosity of the messages displayed.
- The command line below illustrates the use of the –d option with a high value of verbosity (level `3`).

```
$ hmpp -d3  icc myHMPPApplication.c
```

7.2.2. Host compiler options

Most of the standard compiler options are supported by HMPP. These one are directly given on the command line and follow the specification of the compiler.

The "`-d`" HMPP's option can be notified on the command line to increases the level of verbosity of HMPP during the compilation stage. Thus all the commands executed will be displayed allowing the user to check that the right options are given to the compiler.

```
$ hmpp -d ifort -O3 myHMPPApplication.f90
```

Note `that "-E" option` runs preprocessor only. With this option, only the preprocessing of the file is done, resulting in source files where the HMPP directives have been translated into calls to the HMPP runtime. The preprocessed files can then be compiled with the usual general purpose compiler

Compiler options that would change the semantic of the code should not be used. Typical example for FORTRAN compilers are the following:

- `-fall-intrinsics`
- `-fd-lines-as-code`, `-fd-lines-as-comments`
- `-fdefault-double-8, -fdefault-integer-8, -fdefault-real-8`
- `-fmodule-private`
- `-fbackslash`
- `-fcray-pointer`
- `-fdollar-ok`
- …

These options are mainly to support FORTRAN dialects.

7.2.3. Report option

This option provides users with some results of analysis done by HMPP. Currently, HMPP offers:

- --"--io-report": option to display the intents detected by HMPP

7.2.4. HMPP codelet generation options

These options can be used to modify the default behavior of the HMPP command. These options are the following:

- `-f, --force`: forces codelet file overwrite,
- `--codelet-off`: does not generate codelets,
- `--codelet-build`: generates and compiles codelets only,
- `--codelet-generate`: generates codelet only,
- `--codelet-compile codelet_filename`: compiles codelet only. The specified file must be a codelet source file.

Please refer to "Appendix A - HMPP Compilation examples" for detailed use examples.

7.2.5. HMPP native function compilation[24]

Warning:

HMPP 3.0.0 only supports external functions.

For convenience, the text referencing external and native functions was left in state. In HMPP 3.0.0, only the external function is to be considered.

In order to compile a file that uses HMPP native functions in codelets, the XML file that describes them needs to be passed to the HMPP compiler using the `--native` option:

```
--native=[PATH]/my_xml_file.xml [, [PATH]/my_other_xml_file.xml]*
```

For example:

```
hmpp --native=my_native_function.xml gcc sum.c -o sum.exe
```

When HMPP detects the use of a native function, the following `HMPP DPL0716` message appears during the compilation:

```
hmppcg: [Message DPL0716] sum.c:21: Using function 'my_function_name' provided at line 2 of
"my_xml_file.xml"
```

This message indicates that a native function, called in a codelet, has been found in the provided XML file and that this function is going to be used in the generated code.

The generated codelet file is then compiled with the target compiler. In case of programming errors in the definition of the native function in the XML file (code syntax, wrong prototype, wrong number or type of parameters…), the target compiler should report them (note that with OpenCL, the compilation of the kernel is done at the execution time).

7.2.6. HMPP external function compilation[25]

The files defining external functions need to be compiled before the ones that call them. So, the compilation process has two phases:

- First compile the files defining external functions. This generates their XML description file.
- Then compile the files that use external functions.

When an external function is generated HMPP emits the following message:

```
hmpp: [Info] Generated XML filename is "hmppcg_functions.xml"
```

[24] See documentation [R3] to have a complete description of HMPP native functions.

[25] See documentation [R3] to have a complete description of HMPP external functions.

Compilation of Files Defining External Functions

The following command create or update in the [PATH] directory the file named "myFunctions.xml" that contains all the target versions (CUDA/OpenCL) indicated in the declaration.

```
$ hmpp --function=[PATH]/myFunctions.xml gcc -c sum.c -o sum.o
```

Compilation of files that call external functions

By using the following command, HMPP will look for the definition of the external functions in the "myFunctions.xml" file located in the [PATH] directory (current if empty):

```
$ hmpp --function=[PATH]/myFunctions.xml gcc -c extern.c -o extern.o
```

7.2.7. HMPP codelet compilation: proprietary compiler options

In HMPP, the final codelet code is generated with the proprietary hardware accelerator compiler. In some context, it may be useful to forward some specific options to this compiler.

For NVIDIA architecture, options can be passed to the nvcc compiler (NVIDIA CUDA Compiler driver) by using the options "- -nvcc-options".

For example the following command line will forward the options "ptxas=-v, -arch,sm_13" to the nvcc compiler:

```
$ hmpp -d --nvcc-options -Xptxas=-v,-arch,sm_13 ifort main.f90 -o main.exe
…
hmpp: [Info] Running command: nvcc --cudafe-options --no_warning saxpy_cuda.cu -shared -Xptxas=-v
-arch sm_13 -o saxpy_cuda.so --compiler-options -fPIC
ptxas info    : Compiling entry function '_Z13hmppcg_loop0_ILj32ELj4EEvifPfS0_'
ptxas info    : Used 3 registers, 48+48 bytes smem, 2000 bytes cmem[0], 8 bytes cmem[1]
…
```

Another possible approach is to use an environment variable as for example the NVCCFLAGS.

```
NVCCFLAGS='-O3 -use_fast_math' hmpp gfortran -O3  sgemm1.f90   -o sgemm1.exe
```

7.2.8. HMPP miscellaneous options

Various others options can be used with HMPP:

- --hmpp-version: displays HMPP version number,
- --hmpp-full-version: displays HMPP full version message,
- -h, --help: displays an help message and exit,
- --licenses: displays information about HMPP licenses found in the system and exit.

7.2.9. __HMPP predefined macro

During compilation the __HMPP macro is set by default. Its value is equal to the current HMPP version. For instance -D__HMPP=20500 for HMPP 2.5.0 or -D__HMPP=30000 for HMPP 3.0

7.2.10. HMPP Environment Variables

The HMPP environment variables available for Linux and Microsoft Windows platforms are respectively described into:

- Document [R5] for Linux platforms;

8. Running HMPP Applications

The execution of the application is based on the HMPP runtime library that manages the correct execution of the HMPP application according to the user's environment.

For further details concerning the execution of HMPP program, readers will refer to:

- [R5] for Linux platforms;

9. Supported Platforms and Compilers

For a complete knowledge of:

- The operating systems on which you can run HMPP;
- The Software Development Kit (SDK) provided by constructors and supported by HMPP;
- The different compilers that you can use with HMPP

Please refer to:

- [R5] for Linux platforms.

10. HMPP Installation

Instructions to set HMPP on your system are respectively described into:

- Document [R5] for Linux platforms.

11. Annexes

Annex 1. Glossary

callsite	In HMPP context, designates a codelet call in the application
Codelet	A routine to be remotely executed in a HWA. A codelet is a pure function. It is a small self-contained subset section of executable code whose dynamic execution consumes a significant amount of time
CUDA	Programming language for the NVIDIA CUDA compatible hardware
Device	A particular HWA device
General purpose compiler	The usual compiler for general purpose cores (i.e. gcc, icc, ifort, ...),
Guards	Predicates expressed using HMPP directives to define runtime conditions to execute a codelet RPC in a HWA
Hardware Accelerators (HWA)	Devices used to speedup applications' codes. Considered HWA considered are GPUs, FPGAs, or streaming units (SSE, ...). The HWA is not assumed to share memory with the main processor
HMPP	A short name for HMPP development workbench
HMPP codelet	Contains a pure function that can be executed in a HWA using HMPP. The HMPP codelet also contains the HMPP runtime callbacks
HMPP Group of codelets	A group of codelets designates the execution of several codelets based on a same hardware allocation and with the possibility to share data.
HMPP codelet container	HMPP codelet container is a file containing the HMPP runtime callbacks and the HMPP target codelet
HMPP codelet generator	Code generator that takes a C codelet as input and translates it into the HWA input code
HMPP compiler	The HMPP compiler drives all the HMPP passes to build a hybrid application from host application compilation to codelet generation and compilation.
HMPP Codelet Compiler	Compiler used for the compilation of the HMPP codelets as opposed to the HMPP Host Compiler that is used to produce the binary host application.
HMMP Host Compiler	Compiler used to produce the binary host application as opposed to the HMPP Codelet Compiler which designates the compiler used to

	compile the codelets.
HMPP development workbench	A set of tools to help developers programming application that make use of HWAs
HMPP directives	Set of directives to program the use of HWAs in application source
HMPP native codelet	HMPP native codelet is the original function that is annotated using the HMPP directives
HMPP native function	Hand-written CUDA or OPENCL functions provided by end-user and called from HMPP codelet
HMPP external function	Function defined in the source code (C or FORTRAN) and called within an HMPP codelet or region. HMPP automatically generates its CUDA or OpenCL version in an XML file.
HMPP preprocessor	The HMPP preprocessor translates the HMPP directives into calls to the HMPP runtime library
HMPP program	A C or Fortran program that contains HMPP directives
HMPP region	Defines a set of contiguous statements to be executed on the HWA.
HMPP runtime	Runtime library linked with the HMPP program to manage the execution of the HMPP codelet.
HMPP runtime callbacks	API that provides the HMPP runtime with all the necessary services to execute a target codelet
HMPP target codelet	HMPP target codelet is the hardware dedicated implementation of the codelet
HMPP template generator	HMPP template generator creates an empty HMPP codelet container
Label	A label identifying a group of directives defining the declaration and execution of a codelet.
main thread	Process that executes the original code
Remote Procedure Call (RPC)	In HMPP, a RPC denotes the remote execution of a codelet in a HWA

Annex 2. Bibliography

[R1]	HMPPWorkbench-3.0_Basics.pdf, CAPS entreprise
[R2]	HMPPWorkbench-3.0_HMPP_Directives_ReferenceManual.pdf, CAPS entreprise.
[R3]	HMPPWorkbench-3.0_HMPPCG_Directives_ReferenceManual.pdf, CAPS entreprise
[R4]	HMPPWorkbench-3.0_Windows_Manual.pdf, CAPS entreprise
[R5]	HMPPWorkbench-3.0_Linux_Manual.pdf, CAPS entreprise
[R6]	HMPPWorkbench-3.0_LicenseInstallationGuide.pdf, CAPS entreprise

Annex 3. List of Figures

Annex 4. List of Listings

Annex 5. List of Tables