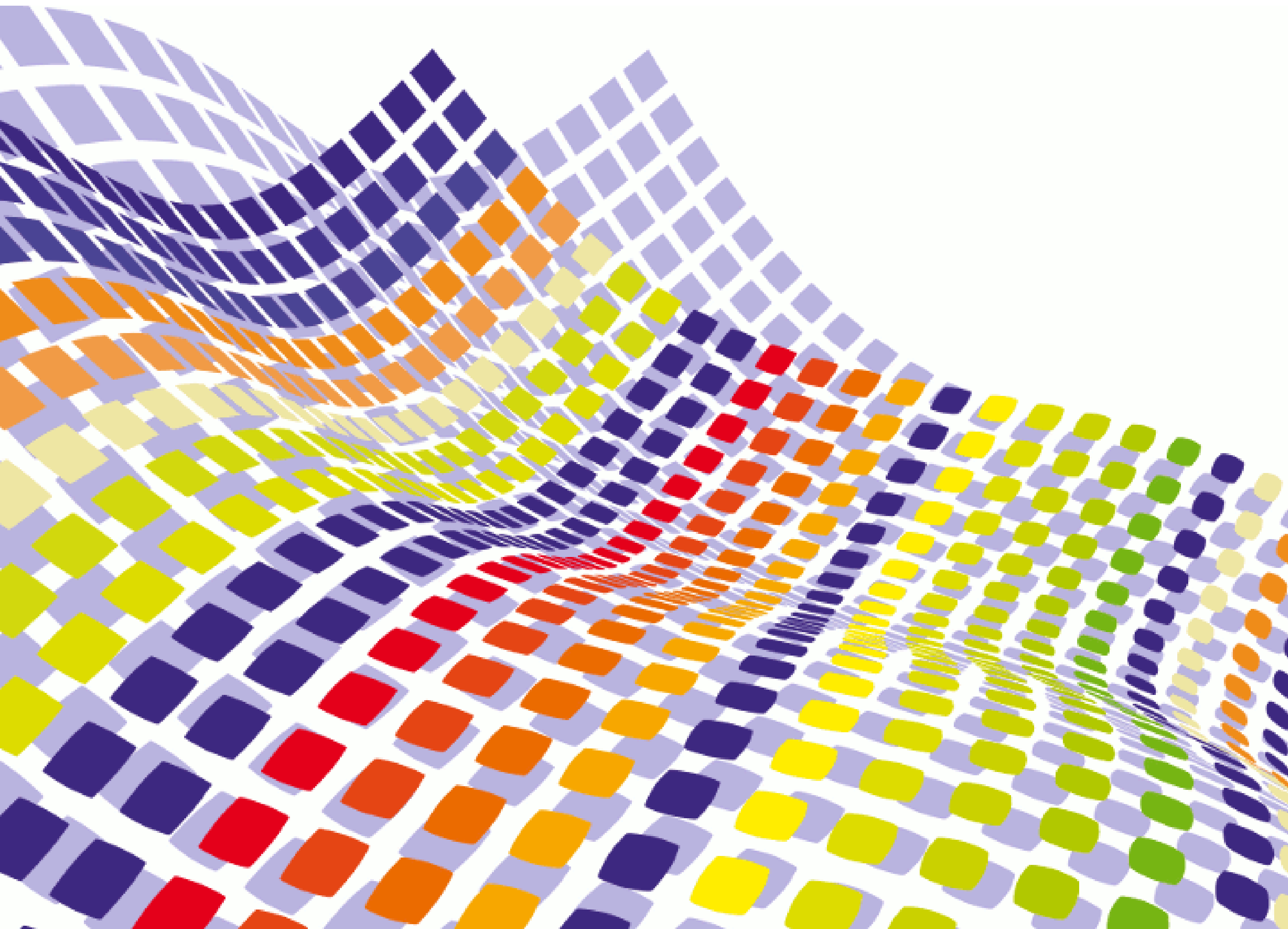
HMPP Codelet Generator Directives

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

1. Introduction

1.1. Revisions history

Version	Date	Writer	Modified pages	Revision object
V2.4.0	12/11/2010	CAPS entreprise	All § 3.3.1 § 3.3.2	Creation from HMPP 2.3.5 User Guide Add new clauses to parallel directive Introduction of the gridify directive
V2.4.4	25/03/2011	CAPS entreprise	§ 3.3.6	Reference to bibliography updated
V2.5.0	16/06/2011	CAPS entreprise	§4.1, §4.2	Introduction of native and external functions
V3.0.0	12/2011	CAPS entreprise	§3.3.1 §3.3.2	Clause parallel deprecated Complement to gridify directive

1.2. Introduction

Warning:

In this User Guide we assume that the reader has a background in parallel programming techniques and in hardware accelerators (HWAs) such as Graphics Processing Units (GPUs).

We also assume that the reader knows the HMPP concepts introduced in the HMPP Directives reference manual.

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.

The present document introduces:

- A set of directives dedicated to the improvement of the code performance by allowing some automatic transformations done on the user's source code. Unlike the HMPP directives, these directives are prefixed by the "**`HMPPCG`**" keyword for "**HMPP C**odelet **G**enerator" according that they act at codelet generation level (chapter 3)
- A HMPP preprocessor allowing users to factorize HMPP directive definition (see chapter 4).

This document comes in addition to the following manuals:

- `HMPP Basics` ([R9]). This document introduces the main HMPP concepts.
- `HMPP Directives, Reference Manual ([R10])`. This manual introduces the main HMPP concepts and describes the HMPP directives. This document is the main document for people who want to discover HMPP;
- `HMPP Linux Manual ([R12])`. 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 ([R13])`. This manual presents the procedure to set the HMPP license on your system.

1.3. Content of this document

The remainder of this document is organized as follow:

- Chapter 3 provides some hints on advanced usage of HMPP and loop transformations;
- Chapter 4 details the use of HMPP native and external functions;

- Chapter 5 introduces a preprocessor of HMPP directives.

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 offers 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 document [R10]) 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 API. The preprocessed code is then compiled and linked with the HMPP runtime API using the host compiler. The HMPP runtime API 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.

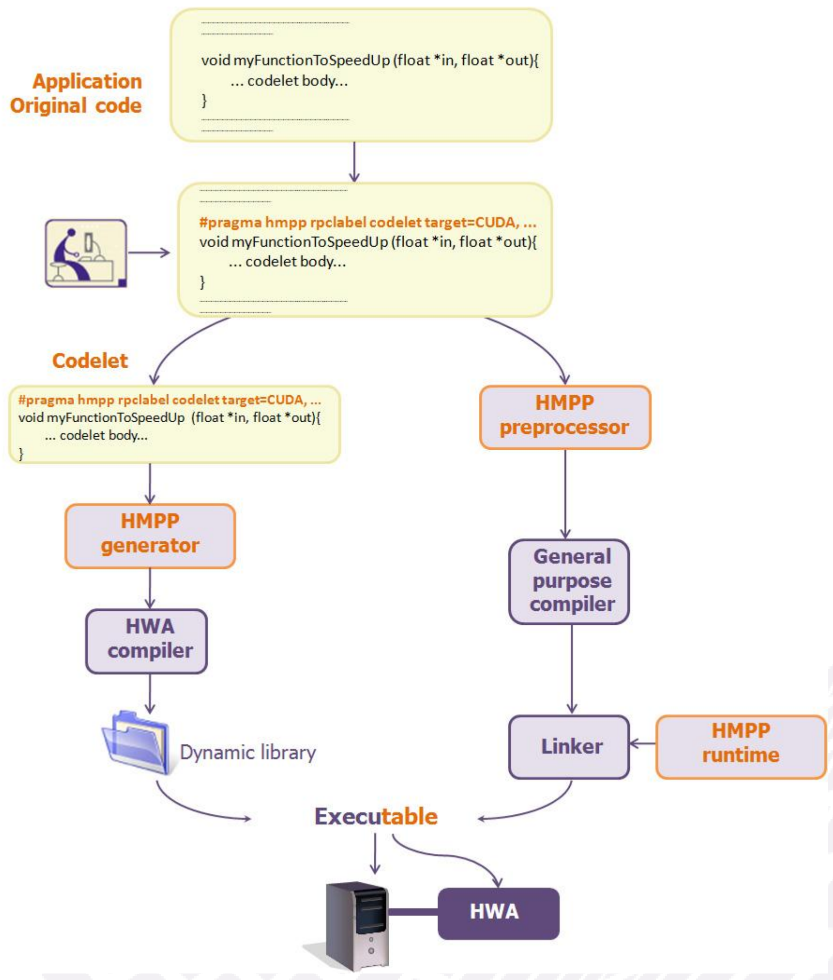

Application Original code
void myFunctionToSpeedUp (float *in, float *out){
... codelet body...
}
#pragma hmpp rpclabel codelet target=CUDA, ...
void myFunctionToSpeedUp (float *in, float *out){
... codelet body...
}
Codelet
#pragma hmpp rpclabel codelet target=CUDA, ...
void myFunctionToSpeedUp (float *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 API is the dynamic library in charge of the execution of the remote procedure calls to the HWA. Linked to the application, this library allocates and initializes the HWA in order to allow the execution of the codelets. It relays communications between the host and the HWA and manages the asynchronous 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. Going further: improving code performance by enhancing the code generation

A set of directives is provided to optimize the generation and mapping of input codelets into the target code (CUDA [R2] or OpenCL [R6] for example).

Most of the transformations described in this part apply on loops. A loop is a syntactic language construction expressing the repetition of some statements.

In HMPP, a transformation can be applied on a loop if this one can be normalized by HMPP. Such a loop must have:

- A unique induction variable;
- Its number of iterations must be computable at run-time before entering the loop.

"**for**" loops in C language and "**DO**" loops in FORTRAN are supported.

To optimize the code generated by HMPP, two main types of directives are used:

- Some specifying loop properties;
- Others mentioning transformations to be applied on the loops.

More directives will be provided in next versions of HMPP.

Please note that HMPP does not check for the incorrect usage of the directives. Be aware that misuse of the HMPPCG directives may lead to erroneous results.

3.1. HMPPCG Directives syntax

The general syntax of the directives is (respectively for C and FORTRAN) the following:

C language:

```
#pragma hmppcg [(target)]? directive_type [clause] [, clause]*
```

FORTRAN language:

```
!$hmppcg[(target)]? directive_type [clause] [, clause]*
```

Where:

- `target`: allows to restrict the execution of the directive to a specific target.

For example, on Listing 1 the hmppcg permute transformation will be applied regardless of the considered hardware accelerator.

Listing 1 - hmppcg directive (basic example)

```
#pragma hmppcg permute (j, i)
  for (i = 1; i < M - 1; ++i) // 0
    {
      for (j = 1; j < N - 1; ++j) // 1
        {
          B[i][j] = c11 * A[i - 1][j - 1] + c12 * A[i][j] ;
        }
    }
```

On Listing 2, the `hmppcg permute` transformation will be applied only for target OPENCL or CUDA.

Listing 2 - hmppcg target dependent execution

```
#pragma hmppcg (OPENCL:CUDA) permute (j, i)
  for (i = 1; i < M - 1; ++i) // 0
    {
      for (j = 1; j < N - 1; ++j) // 1
        {
          B[i][j] = c11 * A[i - 1][j - 1] + c12 * A[i][j] ;
        }
    }
```

Warning

Note that all the directives described in this part are introduced by using the hmppcg keyword and do not contain any labels. They are dedicated to codelet generation and apply only on the codelet's source code that they just precede. They can only be used in codelets or regions.

hmppcg is an abbreviation of HMPP Codelet Generator.

3.2. Interpretation order of the HMPPCG directives

With HMPP, several transformations can be applied, one after the other, on a loop nest. Two modes or directives scheduling are provided:

- One based on lexical order in the source code (default mode);
- One base on the evaluation of an "`order`" clause.

In the first case, the order in which the transformations are executed follows these steps:

- Step 1: search the first HMPPCG directive;
- Step 2: apply the source code transformation given by the directive (or ignore it if the target does not match);
- Step 3: go back to step 1 with the resulting code until no transformation remain to be applied.

Users must be careful about the order in which the directives are applied. The directives are successively evaluated and their execution is performed on the code resulting from the previous transformation.

Table 1 below illustrates this mode of operation with the following assumptions:

- "`dx`" means here directives to apply;
- A symbolic notation is used.

Table 1 – interpretation order of hmppcg directive: lexical order

```
d1  permute k, i, j
    loop i        // i index
d2  unroll 2
     loop j       // j index
d3  unroll 3
       loop k     // k index
         s1
         s2
```
	```
	d3  unroll 3
	    loop k          // k index
	      loop i        // i index
	d2   unroll 2
	        loop j      // j index
	         s1
	         s2
	```

1 – Initial code. First the directive `d1` is applied	2 - The execution of `d1` leads now to have `d3` in first position. So, the directive `d3` will be the next directive applied.

`   loop k    // loop k is unrolled` `    loop i         // i index` `d2   unroll 2` `     loop j        // j index` `         s1` `         s2`	`loop k   // loop k is unrolled` `  loop i   // i index` `   loop j  // loop j  is unrolled` `        s1` `        s2`
3 - Then d3 is applied. The execution of d3 does not change the order of the directive. Loop k is unrolled. The directive d2 will then be the next directive applied.	4 - d2 is now applied. There is no more directives to be applied

Otherwise, another mode is possible through the use of the "order" clause available in certain directives. In this mode, the "order" clause forces the execution of the directives in the increasing order of "order" attributes.

If several directives have the same order value, then they are executed in lexical order. Table 2 illustrates the use of this clause (with the same assumptions as previously).

Table 2 - interpretation order of hmppcg directive: use of the order clause

`d1 permute j,i` `   loop i` `d2 unroll 2,order=1` `     loop j` `d3 unroll 3,order=0` `       loop k` `         s1` `         s2`	`d1 permute j,i` `   loop i` `d2 unroll 2,order=1` `    loop j` `     loop k  // loop k is unrolled` `         s1` `         s2` `…`
1 – Initial code. Directive d3 is first applied since the order directive has the smallest value.	2 - The directive d3 has been applied. The directive d2 will be the next directive applied
`d1 permute j,i` `   loop i` `    loop j   // loop j is unrolled` `     loop k  // loop k is unrolled` `         s1` `         s2` `…`	`loop j   // loop j is unrolled` `  loop i` `    loop k  // loop k is unrolled` `         s1` `         s2` `…`
The directive d2 has been applied. Then the last directive to execute is d1	4 – Loops I and J have been permuted. All the directives have been applied.

Currently, HMPP does not apply more than 5 transformations consecutively on a same loop nest.

3.3. HMPPCG: loop properties

The directives described in this part allow specifying some properties on loops. These properties are then used by the HMPP generator in order to optimize the generated code.

3.3.1. *HMPPCG parallel Directive (deprecated)*

In HMPP 3, this directive is now deprecated. Please consider to use the "hmppcg gridify" directive instead (see chapter 3.3.2 below).

This directive has to be used when the codelet generator is not able to compute the parallel properties of complicated loops.

A parallel loop is declared using the following directive:

```
#pragma hmppcg parallel
               [, none    (*) ]*
               [, global  (var [, var]* ) ]*
               [, private (var [, var]* ) ]*
               [, reduce  (operator:var [, operator:var]* ) ]*
```

Where:

- `none (*)`: by default no scope is specified for any variable. So if this clause is specified and if a loop nest variable is not mentioned in one of the `global` or `private` clause, this will trigger an error.
- `global`**:** this clause declares the variables as shared between the threads.
- `private`: specifies that each loop iteration should have its own instance of the variable. A private variable is not initialized and the value is not maintained for use outside of the loop.
- `reduce`*:* specifies that in the considered loop, a reduction operation is performed (see chapter 0.0.1073790976 below);

This directive applies on the loop it precedes.

3.3.2. HMPPCG gridify Directive

This directive can be used to guide the gridification of the loop nest. The loop nest gridification process converts parallel loop nests into a grid of GPU threads.

HMPP implements a 2-dimension gridification process which is in most of the cases applied automatically. However, in some situations, users may want to specify how to gridify the loops.

The syntax of the directive is:

```
#pragma hmppcg gridify [ (inductionVariable [,inductionVariable]*)]?
               [, none    (*) ]*
               [, global  (var [, var]* ) ]*
               [, private (var [, var]* ) ]*
               [, reduce  (operator:var [, operator:var]* ) ]*
               [, blocksize (N x M) ]
                [, const   (var, [, var]*) ]*
                [, constaddr (var, [, var]* ]*
                [, disable_auto_const ]
                [, sharedinit (var, [, var]* ) ]*
                [, shared(var, [, var]* ) ]*
                [, unguarded]?
```

Where:

- `inductionVariable`: designates the loop indexes to “`gridify`”. By default, if no induction variable is specified, the clause applies only on the loop which follows the directive. It should be noted that the order of the induction variable is particularly important. Indeed, the order of the specified induction variables is taken into account by the gridify directive and permutation of the loops is automatically done in accordance of the specified induction variable order.
- `none (*)`: by default no scope is specified for any variable. So if this clause is specified and if a loop nest variable is not mentioned in one of the clause `global` or `private` clause, this will trigger an error.
- `global`**:** this clause declares the variables as shared between the threads.
- `private`: specifies that each loop iteration should have its own instance of the variable. A private variable is not initialized and the value is not maintained for use outside of the loop.

- `reduce`*:* specifies that in the considered loop, a reduction operation is performed (see chapter *"reduce clause"* below);
- `blocksize`: specifies the number of threads in a block of the gridification.
- `const`: specifies which variables must be placed in constant memory in the gridification
- `constaddr`: similar to clause `const` but just place the address of the parameter on constant memory. This clause has no effect on scalars and structures passed by value.
- `disable_auto_const`: disable auto promotion of scalar to constant memory in the gridification
- `sharedinit`: specifies each variables must be preload in shared memory.
- `shared`: specifies each variable must be placed in shared memory. Initialization must be node explicitly by the user.
- `unguarded`: removes the guard normally used to 'kill' the unneeded threads in the last blocks of each dimension of gridification.

This directive applies on the loops it precedes.

Scope of the variables inside the gridify pragma

The three clauses: `global, private, none`, allow to specify the scope of the variables present in a loop nest according to the gridification mechanism implied by the gridify directive. The global and private clauses accept a list of arguments or the "`*`" wildcard as a shortcut to specify any variables. The `none` clause applied to all variables of the loop nest.

- `global (var [, var]* )`: this clause declares the variables as shared between the threads[1].
- `private`: specifies that each loop iteration should have its own instance of the variable. A private variable is not initialized and the value is not maintained for use outside of the loop[2].
- `none` **(*)**: by default no scope is specified for any variable. So if this clause is specified and if a loop nest variable is not mentioned in one of the clause `global` or `private`, this will trigger an error. This allows to guarantee that all the variables present in the loop nest are mentioned in one of the `global` or `private` clauses.

The variables specified in this directive must be declared outside of the loop nest and used in the loop nest.

So, for example, Listing 3 declares the variable "`mylocalvar`" as private to the threads, "`v1, v2, alpha`" as global for all the threads.

```
#pragma hmppcg gridify, private (mylocalvar), global (v1, v2, alpha)
  for( i = 0 ; i < n ; i++ ) {
    v1[i] = alpha * v2[i] + v1[i];
    mylocalvar= v1[i];
  }
```

Listing 3 - hmppcg gridify directive with private and global clauses

In the case of any of these clauses are specified, HMPP automatically privatizes the necessary variables to gridify the loop.

[1] This concept is similar to the shared clause of the OpenMP parallel directive.

[2] This concept is similar to the private clause of the OpenMP parallel directive.

The reduce clause

The `reduce` clause allows the user to indicate that one or several reductions are done in the loop. Indeed, without this clause, the parallel execution of a loop with such an operation could lead to a wrong result.

- `operator`: specifies a reduction operator (see Table 3)
- `var`: is the name of a scalar variable referenced in the loop

The table below presents the list of allowed reduction operators in the `reduce` clause.

Table 3 - List of reduction operators defined in HMPP

Operators	Meaning
`+`	Addition
`*`	Multiplication
`min`	Minimum
`max`	Maximum

`and`
`.and.`
`&&`
	Logical and

`or`
`.or.`
`||`
	Logical or

`ixor`
`ieor`
`^`
	bitwise exclusive or

`ior` `|`	bitwise inclusive or
`iand` `&`	bitwise and

Listing 4 illustrates the use of the hmppcg parallel directive with two addition (i.e. +) reduction operations.

Listing 4 - hmppcg reduce clause with reduction operations

```
#pragma hmppcg gridify, reduce (+:ssx,+:ssy)
 for ( i = 0; i < NK; i++)
 {
   if (qqprim2[i])
     {
       qq[qqprim[i]] += 1.0;
       ssx = ssx + qqprim3[i];                       /* sum of Xi */
       ssy = ssy + qqprim4[i];                       /* sum of Yi */
    }
}
```

Note that the use of this directive forces HMPP to consider the loop parallel independently of any analysis carried out by HMPP. In some cases, some conflicts may arise between the directive specified and the HMPP loop analysis. The table below summarizes such situations:

Results of HMPP loop-kinds analysis	HMPPCG pragma used	Results
Parallel	`None`	Loop is computed on hardware accelerator
	`Parallel / gridify`	Loop is computed on hardware accelerator
	`Noparallel`	Loop is computed on the CPU
	`Parallel / gridify reduce`	Loop is computed on hardware accelerator
Sequential	`None`	Loop is computed on the CPU
	`Parallel / gridify`	Loop is computed on the hardware accelerator. A warning message mentions that this execution could lead to erroneous result
	`Noparallel`	Loop is computed on the CPU
	`Parallel / gridify, reduce`	Loop is computed on hardware accelerator
Parallel with reduction	`None`	Loop is computed on the CPU
	`Parallel / gridify`	Loop is computed on the hardware accelerator. A warning message mentions that this execution could lead to erroneous result
	`Noparallel`	Loop is computed on the CPU
	`Parallel / gridify, reduce`	Loop is computed on hardware accelerator (with a warning message if a reduction variable is not mentioned in the reduce clause)

3.3.3. *Inhibiting Vectorization or Parallelization*

A non-parallel loop (i.e. sequential) is declared using the following directive:

```
#pragma hmppcg noParallel
```

The following example shows a loop nest where the use of the HMPP directives allows guiding the code generation.

Listing 5 – `noParallel` and `parallel` directives

```
#pragma hmppcg noParallel
for (i=0; i < n; i++) {
  A[i][n] = B[i+1];
#pragma hmppcg parallel
  for (j=0; j < n; j++) {
    D[i][j] = A[i][j] * E[3][j];
  }
}
```

This directive proved to be useful to control the gridification process of loops on targets such as CUDA. However, we recommend to use the `gridify` directive (§ 3.3.2)

Note that this directive forces HMPP to consider the loop as sequential independently of any analysis carried out by HMPP. Such a loop will be executed on the CPU.

This directive applies on the loop it precedes.

3.3.4. HMPPCG blocksize directive

This directive controls the number of threads in a block for the "gridification" of a loop nest.

This directive only applies on the loop nest it precedes.

This pragma applies only on the loop nest it precedes. If no pragma is supplied, the default value is used (`"32x4")`.

The syntax is:

```
#pragma hmppcg blocksize  "nxm"
```

Where:

- `n` and `m` are the new dimensions of blocks sizes within the grid.

For example, for NVIDIA architecture, typical values are: `"16x16", "32x8", "64x2", "32x4".`

Note that the optimal value of the block size is dependent on the loop nest and on the targeted hardware.

After having been used on a loop nest, if the value of the block size needs to be change for a following loop nest, a new `"hmppcg grid blocksize"` directive must be added in the codelet otherwise the default value will be applied (see example below).

```
//Loops will be gridified with the default value
for (i=0; i < n; i++) {
   for (j=0; j < n; j++) {
     …
  }
}

…
#pragma hmppcg blocksize 8x8
//Loops will be gridified with 8x8 threads in a block
for (i=0; i < n; i++) {
   for (j=0; j < n; j++) {
     …
  }
}

…
//Loops will be gridified with the default value
for (i=0; i < n; i++) {
   for (j=0; j < n; j++) {
     …
  }
}
```

Listing 6 - hmppcg blocksize directive. Example of use

3.3.5. *HMPPCG accelerated context queries*

Within a HMPP codelet or region, the "hmppcg set" directive provides a way to obtain information about the current accelerated context.

The general syntax of the directive is:

C language syntax:

```
#pragma hmppcg set <varname> = <query>(<arguments>)
```

FORTRAN language syntax:

```
!$hmppcg set <varname> = <query>(<arguments>)
```

Where:

- <varname> is a scalar integer variable
- <query> is one of the supported HMPPCG query intrinsics
- <arguments> is a comma separated list of arguments (if the query intrinsic needs any).

Alternatively, the query intrinsic can be replaced by a single default integer constant

```
#pragma hmppcg set <varname> = <constant>
```

The semantic of the "hmppcg set" directive is that of a standard assignment of the specified variable for all the specified HMPP targets. It is however ignored in the fallback implementation.

This behavior allows detecting dynamically whether the fallback is executed or not as shown by Listing 7.

```
PROGRAM test
  integer :: x

  !$hmpp foo callsite
  CALL foo(x)

  IF (x==0) THEN
     PRINT *, "The fallback was executed"
  ELSE
     PRINT *, "The CUDA target was executed"
  END IF

CONTAINS

  !$hmpp foo codelet, target=CUDA
  SUBROUTINE foo(status)
    IMPLICIT NONE
    INTEGER, INTENT(OUT) :: status
    status = 0
    !$hmppcg set status = 1
  END SUBROUTINE foo

END PROGRAM test
```

Listing 7 - Illustration of the hmppcg set directive

Combined with the ability to restrict any HMPPCG directive to a specific target, the "`set`" directive allows detecting dynamically which target is currently executed (see Listing 8).

```
PROGRAM test
  integer :: x

  !$hmpp foo callsite
  CALL foo(x)

  IF (x==0) THEN
     PRINT *, "The fallback was executed"
  ELSE IF (x==1) THEN
     PRINT *, "The CUDA target was executed"
  ELSE IF (x==2) THEN
     PRINT *, "The C target was executed"
  END IF

CONTAINS

  !$hmpp foo codelet, target=CUDA:C
  SUBROUTINE foo(status)
    IMPLICIT NONE
    INTEGER, INTENT(OUT) :: status
    status = 0
    !$hmppcg(CUDA) set status = 1
    !$hmppcg(C)    set status = 2
  END SUBROUTINE foo

END PROGRAM test
```

Listing 8 - Example of the hmppcg set directive used to detect which target is executed

The GridSupport() query

The `GridSupport()` intrinsic returns `1` if the current HMPP target supports the concept of loop gridification (targets `CUDA`, `OPENCL`,...) and `0` otherwise (target `C`).

C language syntax:

```
#pragma hmppcg set <varname> = GridSupport()
```

FORTRAN language syntax:

```
!hmppcg set <varname> = GridSupport()
```

This query is typically used to detect whether an implementation using shared memory is possible in a codelet.

```
 !$hmpp jacobi codelet, target=CUDA:OPENCL:C
 SUBROUTINE jacobi(n,A,B)
   IMPLICIT NONE
   INTEGER, INTENT(IN) :: n
   INTEGER, INTENT(INOUT) :: A(n,n), B(n,n)
   INTEGER :: i,j
   INTEGER :: grid_support
   grid_support = 0
   !$hmppcg set grid_support = GridSupport()
   IF (grid_support==1) THEN
      ! Implement here a version using shared memory
      ...
   ELSE
      ! Implement here a version without shared memory
      ...
   ENDIF
 END SUBROUTINE jacobi
```

Listing 9 - GridSupport query example

The gridification queries

A set of query intrinsics are provided to provide information about the current gridified loop nest. Due to their nature, these queries should only be used within a gridified loop nest. They are not strictly forbidden outside such loops but their result would then be inconsistent.

Each of the gridification query intrinsics exists in 3 forms:

- The `'X'` and `'Y'` forms respectively refer to the internal and external gridified loop.
- The `'XY'` refers to a linearized view of the gridification.
- The last form takes reference to the gridified loop through their index variable which is given as argument.

The following gridification queries are currently supported:

- `BlockSizeX()`, `BlockSizeY()`, `BlockSizeXY()` and `BlockSize(index)` provide the block size as specified by the "`hmppcg grid blocksize`" directive.
- `RankInBlockX()`, `RankInBlockY()`, `RankInBlockXY()` and `RankInBlock(index)` provide the ranks of the current thread within the current block. Numbering starts from `0`.
- `RankInGridX()`, `RankInGridY()`, `RankInGridXY()` and `RankInGrid(index)` provide the rank of the current thread in the complete gridification. Numbering also starts from `0`.
- `BlockIdX()`, `BlockIdY()`, `BlockIdXY()` and `BlockId(index)` provide the rank of the block in the gridification. Numbering also starts from `0`.
- `BlockCountX()`, `BlockCountY()`, `BlockCountXY()` and `BlockCount(index)` provide the number of blocks in the gridification.

The `XY` linearization is performed according to the following formulas:

```
BlockSizeXY()   =  BlockSizeX() * BlockSizeY()
RankInBlockXY() =  BlockSizeX() * RankInBlockY() + RankInBlockX()
BlockIdXY()     =  BlockCountX() * BlockIdY()    + BlockIdX()
BlockCountXY()  =  BlockCountX() * BlockCountY()
RankInGridXY()  =  BlockIdXY() * BlockSizeXY() + RankInBlockXY()
```

Remark: Those intrinsics are all computed using 32bits integers. This is sufficient given the limitations of the current GPUs architectures. The only exception is `RankInGridXY()` which may overflow in 32bit integers for large problem sizes (e.g. a typical CUDA GPU may accept up to 64K*64K blocks of up to 512 threads).

3.3.6. HMPPCG gridification support (deprecated)

In HMPP 3, all the following directives are now deprecated. Please consider to use the "`hmppcg gridify`" directive instead (see chapter 3.3.2 below).

The following directives provide a set of functionalities related to the gridification process:

- "`hmppcg shared variableName, [copyin]?`": declares that a local scalar or array variable must be allocated in shared memory (i.e. all threads in the current gridified block have access to it). For arrays, their dimensions must be constant and known at compile time. The directive must be located within the gridified loop while the shared object must be declared outside that loop. In addition the clause "`copyin`" can be used to initialize the variable using the values of the original global memory variable. The following code illustrates such approach where "`table`" is declared in shared memory and directly initialized thanks to the "`copyin`" clause.

```
#pragma hmpp asin codelet, target=CUDA, args[A].io=out, args[B].io=in
void codelet(int n, int m, float A[m][n][n], float B[m][n][n],
             float table[512][2])
{
  // Use the lookup table to compute A = asin(B)

#pragma hmppcg blocksize 16x16
  for (int i = 0; i < m; i++)
  {
    for (int j = 0; j < n; j++)
    {
      #pragma hmppcg shared table, copyin
      for (int k = 0; k < n; k++)
      {
        int index = (int)(B[i][k][j] * 1023);
        A[i][k][j] = table[index / 2][index % 2];
      }
    }
  }
}
```

- "`hmppcg barrier`": introduces a synchronization barrier between all threads of the current gridified block. This is typically needed to avoid race conditions when accessing objects placed in shared memory. It is important to notice that most targets require ALL threads in the current block to honour the barrier. As a consequence, barriers should never be placed inside divergent conditional statements (i.e. not executed identically by all threads) and use of the "`hmppcg grid unguarded`" directive may be necessary to ensure that all threads of the block are alive.
- "`hmppcg unguarded`": removes the guard normally used to 'kill' the unneeded threads in the last blocks of each dimension of gridification. Consider for example a 1D gridified loop of 1000 iterations and a block size of 64. Without a "`hmppcg unguarded`" directive, the last blocks should only execute the loop body for (1000 modulo 64) = 40 out of its 64 threads. The remaining 24 threads must do nothing and so are considered as dead. For an unguarded gridification those dead threads would be executed thus increasing the effective number of iterations from 1000 to 1024. Using an unguarded gridification is like increasing manually the loop upper bound such that the number of iterations becomes a multiple of the block size. Unguarded gridification is usually needed when using the "`hmppcg grid barrier`" directive that requires all threads of the block to be alive. It should be noted that in most cases some guards must be manually reinserted to insure that the loop indexes remains in the expected ranges.

A typical gridified loop using barriers and shared memory looks like this:

```
!$hmppcg blocksize 64x1
!$hmppcg unguarded
DO i=1,n
  IF (i<=n) THEN
     ... write to shared memory
  ENDIF
  !$hmppcg barrier
  IF (i<=n) THEN
     ... read from shared memory
  ENDIF
  !$hmppcg barrier
ENDDO
```

To get further details or examples about the use of the shared memory, see document [R7], section 4.6 "Exploiting the Shared Memory".

3.3.7. HMPPCG constantmemory directive (deprecated)

In HMPP 3, all the following directives are now deprecated. Please consider to use the "`hmppcg gridify`" directive instead (see chapter 3.3.2 below).

CUDA devices use several memory spaces, which have different characteristics that reflect their distinct usages in CUDA applications. These memory spaces include global, local, shared, texture, and registers (see [R3] and [R4] for more details).

The directive described here helps to improve the performance by allowing the use of the constant memory available on NVIDIA architecture.

Access to this memory space from a HMPP application is possible by the introduction of the following directive in the codelet definition:

C language syntax:

```
#pragma hmppcg constantmemory <param> [, <size>]?
```

FORTRAN language syntax:

```
!$hmppcg constantmemory <param> [, <size>]?
```

With:

- `<param>` : the codelet's parameter (array or scalar)
- `<size>`: the size of the array (number of elements). When scalar variable are defined, the size is optional or must be equal to 1.

It should be noted that by default, scalar variables are automatically placed in constant memory. There is no more than 2Ko of possible allocation.

In the current version of HMPP, limitations are as follow:

- If the size is not specified in the directive, the array must be defined with a constant size;
- If the array is defined with a non constant size, then the size must be specified in the directive;
- If the array is defined with a constant size AND the size of the array is also specified in the directive, the directive's parameter will be chosen.

This directive applies on codelet's parameters. In FORTRAN application, it must be introduced before executable statements.

The limitations expressed here will be removed in next versions of HMPP.

3.4. HMPPCG: loop transformations

Unlike the directives described earlier, those described in this part specify some transformation to apply on a loop. These transformations are applied before the final code generation. Their application can provide better performance by improving computation scheduling or data locality.

3.5. Permute transformation

The loop permutation is a common transformation which is usually used to improve data accesses locality but can also be used to create coarse-grain or fine-grain parallelization.

This directive provides a way to permute nested loops. It may be very useful to reorder the loops according to the code that will be executed on CPU or on hardware accelerator. The order of loops may impact the coalescing of memory accesses.

The syntax is:

```
#pragma hmppcg permute <var>, <var> [, <var>]*
                      [, order = <order_value> ]?
```

Where:

- <var>: identify one of the loops, based on the name of its induction variable.
- <order_value>: is a positive number (starting at zero)

The application of this transformation leads to reorganize the loop control structures according to the new order specified by the directive. Example:

Before	After

```
!$hmppcg permute k, i, j
DO I = 1, 8
  DO J = 1, 8
    DO K = 1, 8
      A(I, J, K) = B(I, J, K)*1.2
    ENDDO
  ENDDO
ENDDO
```
	```
	DO K = 1, N
	    DO I = 1, N
	      DO J = 1, N
	        A(I, J, K) = B(I, J, K)*1.2
	      ENDDO
	    ENDDO
	  ENDDO
	```

	The loops follow now the order k, I, j as specified in the directive.

3.5.1. *Distribute transformation*

In some situations, loops may be too complex to be automatically parallelized:

- It may contain statements which prevent the parallelization
- The generated loop can use too many registers which prevent effective execution.

The distribute transformation splits the initial loop into several separate loops. This directive has two parts:

- The first part identifies the loop on which the transformation will be applied
- The second part identifies where the loop shall be cut off.

The syntax is:

```
#pragma hmppcg distribute
                         [, addtoall {<dir> [; <dir>]*} ]*
                         [, order = <order_value> ]?


#pragma hmppcg cut  [, add      {<dir> [; <dir>]*} ]*
```

Where:

- `<dir>` is a HMPP Codelet Generator directive. In this context, the hmppcg directive is written without the “language directive prefix”. This one is automatically add by HMPP.

For example:

```
!$hmppcg distribute, addtoall {unroll 2, jam}
```

Add an hmppcg unroll directive to the loops resulting from the application of the distribute clause.

- `<order_value>` is a positive number (starting at zero).
- The “`addtoall`” clause allows user to add new directives to the resulting loops created by the transformation.

The distribute directive is attached to the loop to be divided. This loop must contain at least one cut directive.

Listing 10 and Listing 11 illustrate the use of this transformation.

Listing 10 - Original code

```
DO I = 1, SIZE_1
!$hmppcg distribute
 DO J = 1, SIZE_2   ! original loop
 T(I, J) = 0
!$hmppcg cut
   DO K = 1, SIZE_2
     T(I, J) = A(I, K) * B(I, J)
   ENDDO
!$hmppcg cut
 C(I, J) = C(I, J) + T(I, J)
  ENDDO
 ENDDO
```

Listing 11 - Code after having applied the distribute transformation[3]

```
for (i_2 = 0, hmppcg_end = (*size_1) - 1; i_2 <= hmppcg_end; i_2 += 1)
 {
  for (j_21 = 0, hmppcg_end = (*size_2) - 1; j_21 <= hmppcg_end; j_21 += 1)
   {
    t[j_21][i_2] = 0;
   } // end loop j_21

  for (j_2 = 0, hmppcg_end = (*size_2) - 1; j_2 <= hmppcg_end; j_2 += 1)
    {
      for (k_2 = 0, hmppcg_end = (*size_2) - 1; k_2 <= hmppcg_end; k_2 += 1)
        {
          t[j_2][i_2] = (a[k_2][i_2]) * (b[j_2][i_2]);
        } // end loop k_2
    } // end loop j_2

  for (j_22 = 0, hmppcg_end = (*size_2) - 1; j_22 <= hmppcg_end; j_22 += 1)
    {
       c[j_22][i_2] = (c[j_22][i_2]) + (t[j_22][i_2]);
    } // end loop j_22
    } // end loop i_2
```

3.5.2. *Fuse transformation*

This transformation is just the opposite of the previous one. If the granularity of a loop, or the work performed by a loop, is small, the performance gain from its parallelization may be insignificant. This is because the overhead of parallel loop start-up is too high compared to the loop workload. In such situations, the hmppcg fuse transformation can be used to combine several loops into a single one, and thus increase the granularity of the loop.

To apply this transformation, the loops must have the same iteration space and must not be separated by any non-loops statements.

The syntax is:

```
#pragma hmppcg fuse <offset>
                                [, add {<dir> [; <dir>]*} ]*
                                [, order = <order_value> ]
```

Where:

- `<offset>` : identifies the loops to consider. Value "`0`" designates the current loop (where the directive is set). So "`-1`" designates the first previous, "`+1`" the first next, "`+2`" the two next loops and so on
- `<dir>` : is a HMPP Codelet Generator directive
- `<order_value>` : is a positive number starting at zero.

The "`add`" clause allows user to add new directives to the resulting loop created by the transformation.

Listing 12 to Listing 15 illustrate the use of this transformation.

[3] The real result differs from this presentation given her for educational purpose.

Listing 12 - Original code

```
 !$hmppcg fuse 1
  DO I = 1, N
     A(I) = B(I) - C(I)
  ENDDO

  DO J = 1, N
     IF(A(J) .LT. 0) A(J) = B(J)*B(J)
  ENDDO
```

Listing 13 - Code after having applied the fuse transformation[4]

```
 for (i_2 = 0, __hmppcg_end = (*n) - 1; i_2 <= __hmppcg_end; i_2 += 1)
  {
    a[i_2] = b[i_2] - c[i_2];
    if (a[i_2] < 0)
    {
      a[i_2] = b[i_2] * b[i_2];
    }
  } // end loop i_2
```

Listing 14 - Original code –with negative fuse index-

```
   DO I = 1, N
     A(I) = B(I) - C(I)
  ENDDO
  !$hmppcg fuse -1
  DO J = 1, N
     IF(A(J) .LT. 0) A(J) = B(J)*B(J)
  ENDDO
```

Listing 15 - Original code –with negative fuse index-

```
  for (i_2 = 0, hmppcg_end = (*n) - 1; i_2 <= hmppcg_end; i_2 += 1)
  {
    a[i_2] = b[i_2] - c[i_2];
    if (a[i_2] < 0)
    {
      a[i_2] = b[i_2] * b[i_2];
    }
  } // end loop i_2
```

3.5.3. Unroll directive transformation

The loop `unroll` transformation is intended to increase register exploitation and decrease memory loads and stores per operation within an iteration of a nested loop. Improved register usage decreases the need for main memory accesses and allows better exploitation of some machine instructions.

[4] The real result differs from this presentation given her for educational purpose.

This transformation can be automatically applied by using the following directive:

```
#pragma hmppcg unroll { <var>:<factor> [, <var>:<factor>]* | <factor> [, <factor>]* }
                      [, remainder|noremainder|guarded [(<var> [, <var>]*)] ]*
                      [, contiguous|split|changestep   [(<var> [, <var>]*)] ]*
                      [, scalartemp|arraytemp]?
                      [, jam                           [(<var> [, <var>]*)] ]*
                      [, addtounrolled  [(<var> [, <var>]*)] {<dir> [; <dir>]*} ]*
                      [, addtoremainder [(<var> [, <var>]*)] {<dir> [; <dir>]*} ]*
                      [, order = <order_value> ]
```

Where:

- <var> : identify one of the loops, based on the name of its induction variable;
- <factor> : is an unroll factor, strictly greater than zero (1 means no unroll performed, but the associated clauses are still executed.).
- The "addtounrolled" and "addtoremainder" clauses allow user to add new directives to the resulting loops created by the transformation.

Then the other clauses drives the loop unroll algorithm.

Dealing with the unroll strategy

Different schemas of unrolling can be used in HMPP. These ones are controlled thank to the following options:

- contiguous: which is the default behavior: the end bound is divided by <factor> and arrays are accessed by a sequence of contiguous indexes.

Table 4 - unroll directive with contiguous option

Initial code

```
#pragma hmppcg unroll i:4, contiguous
for( i = 0 ; i < n ; i++ ) {
v1[i] = alpha * v2[i] + v1[i];
}
```

Extract of generated code (the remainder loop is not represented)

```
for (i_1 = 0, __hmppcg_end = (n / 4) - 1; i_1 <= __hmppcg_end; i_1 += 1)
 {
  v1[4 * i_1] = (alpha * (v2[4 * i_1])) + (v1[4 * i_1]);
  v1[(4 * i_1) + 1] = (alpha * (v2[(4 * i_1) + 1])) + (v1[(4 * i_1) + 1]);
  v1[(4 * i_1) + 2] = (alpha * (v2[(4 * i_1) + 2])) + (v1[(4 * i_1) + 2]);
  v1[(4 * i_1) + 3] = (alpha * (v2[(4 * i_1) + 3])) + (v1[(4 * i_1) + 3]);
}
```

- split: array accesses are distributed along the iteration space.

Table 5 - unroll directive with split option

Initial code

```
#pragma hmppcg unroll i:4, split
  for( i = 0 ; i < n ; i++ ) {
       v1[i] = alpha * v2[i] + v1[i];
 }
```

Extract of generated code (the remainder loop is not represented)

```
for (i_1 = 0, __hmppcg_end = (n / 4) - 1; i_1 <= __hmppcg_end; i_1 += 1)
 {
  v1[i_1] = (alpha * (v2[i_1])) + (v1[i_1]);
  v1[i_1 + (n / 4)] = (alpha * (v2[i_1 + (n / 4)])) +
                      (v1[i_1 + (n / 4)]);
  v1[i_1 + ((n / 4) * 2)] = (alpha * (v2[i_1 + ((n / 4) * 2)])) +
                             (v1[i_1 + ((n / 4) * 2)]);
  v1[i_1 + ((n / 4) * 3)] = (alpha * (v2[i_1 + ((n / 4) * 3)])) +
                             (v1[i_1 + ((n / 4) * 3)]);
 }
```

- `changestep`: similar to contiguous, but the stride of the loop is multiplied by <factor> instead of recomputing accesses from the body of the loop. This strategy requires that the loop has no inter-iteration dependencies.

Table 6- unroll directive with changestep option

Initial code

```
#pragma hmppcg unroll i:4, changestep
  for( i = 0 ; i < n ; i++ ) {
       v1[i] = alpha * v2[i] + v1[i];
}
```

Extract of generated code (the remainder loop is not represented)

```
for (i_1 = 0, __hmppcg_end = ((n / 4) * 4) - 1; i_1 <= __hmppcg_end; i_1 += 4)
    {
      v1[i_1] = (alpha * (v2[i_1])) + (v1[i_1]);
      v1[i_1 + 1] = (alpha * (v2[i_1 + 1])) + (v1[i_1 + 1]);
      v1[i_1 + 2] = (alpha * (v2[i_1 + 2])) + (v1[i_1 + 2]);
      v1[i_1 + 3] = (alpha * (v2[i_1 + 3])) + (v1[i_1 + 3]);
    }
```

Dealing with the remainder loop:

Like the unroll strategy, there are different ways to handle the remainder loop. The following keywords are provided:

- `remainder` is the default behavior. A remainder loop is generated when the number of iterations is unknown or if it is not modulo of the unrolling `<factor>`.
- `noremainder` can be used to prevent the generation of a remainder loop. This option must be used carrefuly. It forces HMPP not to generate a remainder loop (even when the number of iterations is not modulo of the unrolling factor).
- `guarded` is an alternate way to avoid the execution of a remainder loop by inserting guards inside the body of the loop unrolled.

Dealing with scalar variables

When applying a loop unroll and jam transformation, scalar variables can be handled in two ways:

- `scalartemp`: which is the default, temporary variables remain untouched. For example, for the loop nest containing the following statements and unrolled with a factor of two:

```
        tmp += 1;
        out[i1][i2] = in[i1][i2] + 1;
```

it will be transformed into:

```
    tmp__0 = tmp__0 + 1;
    tmp__1 = tmp__1 + 1;
    out[2 * i1_1][i2_1__0] = (in[2 * i1_1][i2_1__0]) + 1;
    out[(2 * i1_1) + 1][i2_1__0] = (in[(2 * i1_1) + 1][i2_1__0]) + 1;
```

- `arraytemp`: private variables accesses are transformed into an array. So in this context, a loop nest containing the following statements and unrolled on the first index with a factor of two and a jam:

```
   tmp+=1;
   out[i1][i2]=in[i1][i2]+1;
```

will be transformed into:

```
      tmp[0] = (tmp[0]) + 1;
      tmp[1] = (tmp[1]) + 1;
      out[2 * i1_1][i2_11] = (in[2 * i1_1][i2_11]) + 1;
      out[(2 * i1_1) + 1][i2_11] = (in[(2 * i1_1) + 1][i2_11]) + 1;
```

Jam clause

Finally, you can control the way duplicated statements are fused together:

- `jam` [(<var> [, <var>]*)]]*   : enable the merge of duplicated child statements inside the specified loop
- The jam's argument designates a loop induction variable. The jam's argument is optional.
- By default, without any arguments, the jam clause applies to the most internal loop of the loop nest. If an argument is specified, this one specifies the loop in which the jam is applied.
- The following examples –given under the form of a pseudo-code to preserve the readability - illustrate the behavior of the jam clause:
- Table 7: illustrates the default behavior of the jam clause. The loop is unrolled according to the loop induction variable, and then the structure of the loop nest is jammed.
- Table 8: illustrates the use of the jam clause with an argument. The loop nest is not completely jammed according to the jam argument which specified that the jam must only be applied at the i_loop level (so only on the j_loop).

Before	After transformation

`#unroll i:2, jam`
`loop i`
`  loop j`
`    loop k`
`     a(i,j,k)`
`    EndLoop k`
`  EndLoop j`
`EndLoop i`
	`loop i`
	`  loop j`
	`    loop k`
	`      a(i,j,k)`
	`      a(i+1,j,k)`
	`     EndLoop k`
	`   EndLoop j`
	`EndLoop i`

Table 7 - Illustration of the jam clause with no argument

Before	Intermediate state	After transformation

`#unroll i:2, jam(i)`
`loop i`
`  loop j`
`    loop k`
`      a(i,j,k)`
`    EndLoop k`
`  EndLoop j`
`EndLoop i`
	`loop i`
	`  loop j`
	`    loop k`
	`      a(i,j,k)`
	`    EndLoop k`
	`  EndLoop j`
	`  loop j’`
	`    loop k’`
	`      a(i+1,j,k)`
	`    EndLoop k’`
	`  EndLoop j’`
	`EndLoop i`
		`loop i`
		`  loop j`
		`    loop k`
		`      a(i,j,k)`
		`     EndLoop k`
		`    loop k`
		`      a(i+1,j,k)`
		`    EndLoop k`
		`   EndLoop j`
		`EndLoop i`

Table 8 - Illustration of the jam clause with argument (the k_loop is not jammed)

Thus, on the original code below:

Listing 16 - Unroll and Jam transformation - Original code

```
#pragma hmppcg unroll i1:2, scalartemp, jam
  for(i1=0; i1<n1; i1++)
    {
      int tmp = 0;
      for(i2=0; i2<n2; i2++)
        {
          tmp+=1;
          out[i1][i2]=in[i1][i2]+1;
        }
    }
```

Listing 17 shows the results of the unroll transformation without the jam clause. The structure of the loop is duplicated two times.

From the same initial code, Listing 18 shows the result obtained with the jam clause. Both loop control structures have been merged into a single one and the statements have been grouped together.

Listing 17 - Unroll transformation with no jam clause[5][6]

```
for (i1_1 = 0, _hmppcg_end = (n1 / 2) - 1; i1_1 <= _hmppcg_end; i1_1 += 1)
{
  tmp_0 = 0;
  {
   for (i2_1_0 = 0, _hmppcg_end = n2 - 1; i2_1_0 <= _hmppcg_end; i2_1_0 += 1)
    {
      tmp_0 = tmp_0 + 1;
      out[2 * i1_1][i2_1_0] = (in[2 * i1_1][i2_1_0]) + 1;
    }
  }
   tmp_1 = 0;
    {
    for (i2_1_1 = 0, _hmppcg_end = n2 - 1; i2_1_1 <= _hmppcg_end; i2_1_1 += 1)
     {
       tmp_1 = tmp_1 + 1;
       out[(2 * i1_1) + 1][i2_1_1] = (in[(2 * i1_1) + 1][i2_1_1]) + 1;
     }
   }
 }
…
```

Listing 18 - Unroll transformation with jam clause applied

```
for (i1_1 = 0, _hmppcg_end = (n1 / 2) - 1; i1_1 <= _hmppcg_end; i1_1 += 1)
{
  int32_t tmp_0;
  int32_t tmp_1;
  tmp_0 = 0;
  tmp_1 = 0;
  {
    int32_t _hmppcg_end, i2_1_0;
    for (i2_1_0 = 0, _hmppcg_end = n2 - 1; i2_1_0 <= _hmppcg_end; i2_1_0 += 1)
    {
      tmp_0 = tmp_0 + 1;
      tmp_1 = tmp_1 + 1;
      out[2 * i1_1][i2_1_0] = (in[2 * i1_1][i2_1_0]) + 1;
      out[(2 * i1_1) + 1][i2_1_0] = (in[(2 * i1_1) + 1][i2_1_0]) + 1;
    }
  }
}
```

3.5.4. Full unroll transformation

This directive is used to fully unroll a loop and its nested loops. Fully unrolling a loop means that the loop is unrolled by its number of iterations and finally replaced by its body.

Of course, this directive can be applied provided that the number of iterations of all loops can be determined at compile-time (otherwise a transformation failure is issued).

The syntax is:

[5] The real result differs from this presentation given here for educational purpose

[6] The remainder loop is not presented on this example.

```
#pragma hmppcg fullunroll [<var>]
                          [, order = <order_value> ]
```

Where:

- <var> is the induction variable of the deepest nested loop which will be fully unrolled.
- <order> is a positive number starting at zero.

Listing 19 - fullunroll directive - original code

```
#pragma hmppcg fullunroll i1
  for(i1=0; i1<13; i1++)
    {
      #pragma hmppcg fullunroll i2
      for(i2=0; i2<10; i2++)
       {
         out[i1][i2]=in[i1][i2]+1;
       }
    }
```

Listing 20 – Code after applying the fullunroll transformation

```
{
  out(0, 0) = in(0, 0) + 1;
  out(0, 1) = in(0, 1) + 1;
  ...
  out(0, 9) = in(0, 9) + 1;
  out(1, 0) = in(1, 0) + 1;
  out(1, 1) = in(1, 1) + 1;
  out(1, 2) = in(1, 2) + 1;
  ...
  out(11, 9) = in(11, 9) + 1;
  out(12, 0) = in(12, 0) + 1;
  out(12, 1) = in(12, 1) + 1;
  out(12, 2) = in(12, 2) + 1;
  out(12, 3) = in(12, 3) + 1;
  out(12, 4) = in(12, 4) + 1;
  out(12, 5) = in(12, 5) + 1;
  out(12, 6) = in(12, 6) + 1;
  out(12, 7) = in(12, 7) + 1;
  out(12, 8) = in(12, 8) + 1;
  out(12, 9) = in(12, 9) + 1;
}
```

3.5.5. *Tile transformation*

This directive is used to divide the iteration space of perfectly nested loops into blocks. This transformation can improve the use of the memory hierarchy through the reuse of variables.

For each of the loops to tile:

- its iteration space is reduced to the wanted size;
- a new loop is created around to iterate between blocks

Applied to a set of loops, every newly created loops are placed outside the original set of loops. Original loops are not destroyed nor replaced. The table below sums up the transformation done:

Before	After having applied the transformation

```
   #pragma hmppcg tile i:2
0  loop i
     loop j
       s1[i]
       s2[i]
```
	```
	   loop ii by 2
	0   loop i [1:2]
	     loop j
	       s1[i]
	       s2[i]
	```

The syntax is:

```
#pragma hmppcg tile { <var>:<size> [, <var>:<size>]*
                    | <size>       [, <size>]* }
       [, addtoouter [(<var> [, <var>]*)] {<dir> [; <dir>]*} ]*
       [, addtotiled [(<var> [, <var>]*)] {<dir> [; <dir>]*} ]*
       [, order = <order_value> ]
```

Where:

- <size> is the new value of one of the dimension of the iteration space of the loop nest
- <var> identifies a loop (based on its induction variable name.)
- <dir> is a HMPP Codelet Generator directive
- <order_value> is a positive number starting at zero.

Listing 21 and Listing 22 illustrate on a simple example the use of this transformation.

Listing 21 - HMPPCG Tile transformation

```
#pragma hmppcg tile i:8
  for( i = 0 ; i < n ; i++ ) {
       v1[i] = alpha * v2[i] + v1[i];
  }
```

Listing 22 - code after having applied the HMPPCG Tile transformation

```
hmppcg_end_outer = (n - 1) / 8;
for (outer_i_2 = 0; outer_i_2 <= hmppcg_end_outer; outer_i_2 += 1)
{
  hmppcg_end_i_2 = ((((outer_i_2 * 8) + 7)>(n - 1)?
                         (n - 1) :
                         ((outer_i_2 * 8) + 7))) - (outer_i_2 * 8);
    for (i_2 = 0 ; i_2 <= hmppcg_end_i_2; i_2 += 1)
     {
      v1[i_2 + (outer_i_2 * 8)] = (alpha * (v2[i_2 + (outer_i_2 * 8)])) +
                                  (v1[i_2 + (outer_i_2 * 8)]);
     }
   }
}
```

4. Native and external functions

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.

The two next sections detail how to declare and call in HMPP codelets both native and external functions.

4.1. Native functions

A native function is a pure CUDA/OpenCL function provided by the user. They are hand-written and can use functions provided in the vendor SDK such as the CUDA warp vote functions: `__all()`, `__any()` and `__ballot()`.

Native functions are called in HMPP codelets or regions. They are defined in a XML file that is passed to the HMPP compiler.

4.1.1. Native function prerequisites

Native functions support scalars variables, pointers and arrays of atomic types. In FORTRAN, variables may be declared as `in`, `out` or `inout` variables. Native functions cannot be recursive (same as for CUDA or OpenCL). Structured types are not supported.

4.1.2. Declaration of the Use of Native Functions

The use of a native function in HMPP codelets or regions is declared using the following directive:

```
#pragma hmppcg native myFunc [, myFunc]*
```

Where:

- `myFunc`: identifies the native function to call in replacement of the existing one.

This directive is inserted in the scope of the codelet or region. It is important to notice that this directive is not attached to a statement but affects the whole codelet or region in which it is placed. The following examples show how to use it in C and FORTRAN:

C example:

```
float sum( float x, float y ) {
  return x+y;
}

int main(int argc, char **argv) {
  int i, N=100;
  float A[N], B[N];

  for(i=0;i<N;i++){
    A[i]=1;
    B[i]=2;
  }


  #pragma hmpp cdlt region, args[B].io=inout, target=CUDA, private=[i]
  {
    #pragma hmppcg native, sum
    for( int i = 0 ; i < N ; i++ )
      B[i] = sum( A[i], B[i] );
  }
}
```

Listing 1 - Use of HMPPCG native pragma in a C code

FORTRAN example:

```
program test
  integer, parameter :: n=100
  real ,allocatable :: A(:), B(:)
  integer i

  allocate(A(n))
  allocate(B(n))

!$hmpp cdlt region, target=CUDA, private=[i]
  !$hmppcg native, sum
  do i=1,n
     B(i) = sum(A(i), B(i))
  enddo
!$hmpp cdlt endregion


CONTAINS


  function sum(a, b) result(res)
    implicit none
    real, intent(IN) :: a
    real, intent(IN) :: b
    real res

    res = a + b

  end function sum

end program test
```

Listing 2 - Use of HMPPCG native pragma in a FORTRAN code

4.1.3. XML Description of Native Functions

The definition of native functions is described in a XML file. For each native function, the description contains:

- The source code language
- The signature of the native function in the indicated source code language

- The definition of the function in each supported target language (CUDA/OpenCL).

Below is the skeleton of the XML file:

```
<hmppcg>
    <function name="myFunction">
        <signature language="mySourceCodeLanguage">
             //Signature of my function
        </signature>


        <definition target="myTarget">
        <![CDATA[

         //my function in myTarget language

        ]]>
       </definition>
     </function>
</hmppcg>
```

Listing 3 - XML skeleton for native function declaration

The values given in the target and language attributes are not case sensitive. However, the name attribute of the function is case sensitive. If the language of the source code is FORTRAN, the value of the name attribute must be in lower-case.

It is important to notice that in the case of native functions called in FORTRAN codelets:

- All parameters of native functions are passed by address.
- All arrays are normalized: a FORTRAN array declared as `A(xmin : xmax, ymin : ymax)` must be used as a C array declared as `A[(xmax – xmin + 1) * (ymax – ymin +1)]` where the first element is `A[0]` and the last element is `A[(xmax-xmin) + (ymax-ymin) * (xmax-xmin +1)]`.

C example:

```
<hmppcg>
    <function name="sum">
        <signature language="c">
             float sum(float x, float y);
        </signature>


        <definition target="cuda">
        <![CDATA[
__device__ float sum(float x, float y)
{
  return x + y;
}
         ]]>
         </definition>

        <definition target="opencl">
        <![CDATA[
float sum(float x, float y)
{
  return x + y;
}
         ]]>
         </definition>
     </function>
</hmppcg>
```

Listing 4 - Definition of the native function SUM in the context of a C signature

FORTRAN example:

```
<hmppcg>
    <function name="sum">
        <signature language="fortran">
             function sum(x, y) result(res)
                real, intent(in):: x
                real, intent(in):: y
                real res
        </signature>


        <definition target="cuda">
        <![CDATA[
//by addresses
__device__ float sum(float *x, float *y)
{
  float res = *x + *y;
  return res;
}
         ]]>
         </definition>

        <definition target="opencl">
        <![CDATA[
//by addresses
float sum(__global float *x, __global float *y)
{
  float res = *x + *y;
  return res;
}
         ]]>
         </definition>
     </function>
</hmppcg>
```

Listing 5 - Definition of the native SUM function in the context of a FORTRAN signature (parameters referenced by addresses)

4.1.4. Compilation

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).

4.2. External functions

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 region. 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.

4.2.1. External function prerequisites

External functions support scalars variables, pointers and arrays of atomic types. In FORTRAN, variables may be declared as `in`, `out` or `inout` variables. External functions cannot be recursive (same as for CUDA or OpenCL). Structured types are not supported.

4.2.2. Declaration of HMPP external functions

External functions are declared using the following HMPP directives, placed just before the function definition (beware that in this context, to control the generation of the function, an HMPP directive is used):

In C:

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

In FORTRAN:

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

Where:

- `target`: designates the target language to be used for the generation of the function, `CUDA` or `OPENCL`.

A version of each indicated target (`CUDA`/`OPENCL`) will be generated by HMPP in a XML file named by default `"hmppcg_functions.xml"`. This file uses the same skeleton (see Listing 3) as for native functions with two additional attributes in the function element:

- `extern` sets to the value "`true`" to specify that we are in the case of an external function;
- and `location` sets to the value "`file.extension:line_number`"

```
<function extern="true" location="file.extension:line_number" name="myFunction">
```

It should be noted that contrary to the native functions, the XML file is automatically generated by HMPP and does not need to be edited.

Below are C and FORTRAN examples of the declaration of external functions:

C example:

```
#include "sum.h"

#pragma hmpp function, target=CUDA
float sum( float x, float y ) {
  return x+y;
}
```

Listing 6 - External function definition (C language)

With `sum.h`:

```
#ifndef SUM_H
#define SUM_H
float sum( float x, float y );
#endif /* SUM_H */
```

Listing 7 - Function prototype declaration (C language)

FORTRAN example:

```
!$hmpp function, target=CUDA
function sum(a, b) result(res)
  implicit none
  real, intent(IN) :: a
  real, intent(IN) :: b
  real res

  res = a + b

end function sum
```

Listing 8 - External function definition (FORTRAN language)

4.2.3. Declaration of the use of external functions

The use of external functions in HMPP codelets or regions is indicated with the following HMPP directive:

```
#pragma hmppcg extern myFunc [, myFunc_x]*
```

Where:

- `myFunc`: identifies the external function to call in replacement of the existing one.
- As for native functions, this directive is inserted in the scope of the codelet or region. It is important to notice that this directive is not attached to a statement but affects the whole codelet or region in which it is placed. The following are two C and FORTRAN examples that show how to use this functionality:

C example:

```
#include "sum.h" // contains the prototype of the sum function (see Listing 7)

int main(int argc, char **argv) {
  int i, N = 10;
  float A[N], B[N];

  for(i=0;i<N;i++){
    A[i]=1;
    B[i]=2;
  }


// HMPP region declaration
#pragma hmpp cdlt region, args[B].io=inout, target=CUDA, private=[i]
  {
    #pragma hmppcg extern, sum   // reference of an external function in the
                                 // scope of the region
    for(i = 0 ; i < N ; i++ )
      B[i] = sum( A[i], B[i] );
  }

}
```

Listing 9 - External function use (C language)

FORTRAN example:

```
program test

INTERFACE
   FUNCTION SUM(a,b) result(res)
    real, intent(IN) :: a
    real, intent(IN) :: b
    real  res
   END FUNCTION SUM
END INTERFACE


  integer, parameter :: n=10
  real ,allocatable :: A(:), B(:)
  integer i

  allocate(A(n))
  allocate(B(n))

  B = 2
  A = 1

!!! HMPP region declaration
!$hmpp myregion region, target=CUDA
  !$hmppcg extern, sum   ! Reference of an external function in the
                         ! scope of the region
  do i=1,n
     B(i) = sum(A(i), B(i))
  enddo
!$hmpp myregion endregion

end program test
```

Listing 10 - External function use (FORTRAN language)

4.2.4. Compilation

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.

These two steps are described below.

Compilation of Files Defining External Functions

By default all external functions are described in a XML file named "`hmppcg_functions.xml`" generated by HMPP in the current directory.

The name of that file can be changed using the "`—function`" option of the HMPP compiler (see below). If the XML file already exists, it is updated. Otherwise, it is created.

For example, if an external function is defined in a file named "`sum.c`":

- The following command create or update in the current directory the default "`hmppcg_functions.xml`" file that contains all the target versions (CUDA/OpenCL) indicated in the declaration.

```
$ hmpp gcc –c sum.c –o sum.o
```

- 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
```

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

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

Compilation of files that call external functions

By default, HMPP looks for the definition of external functions in the "`hmppcg_functions.xml`" file located in the current directory. To specify another file, use the `--function` option of HMPP.

For example, if an external function is used in a file called "`extern.c`":

- Using the following command, HMPP will look for the definition of external functions in the default "`hmppcg_functions.xml`" file located in the current directory:

```
$ hmpp gcc -c extern.c -o extern.o
```

- While 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
```

When an external function call is detected, HMPP emits the following `DPL0713` message:

```
hmppcg: [Message DPL0713] sum.f90:27: Using function 'sum' found at line 3 of "sum_function.f90"
```

5. Going further: factorization of the HMPP directives

HMPP provides a preprocessor which allows the programmer to factorize the declarations of HMPP directives.

The main purposes of having a preprocessor are:

- to simplify the writing of HMPP directives;
- to allow HMPP directives to be configured via compilation options.

The HMPP preprocessor will be run before the native language preprocessor, if any. In practice, it means that using the preprocessor features within included files (e.g. by a Fortran INCLUDE statement or a C #include directive) will not be possible.

The HMPP preprocessor is mostly inspired from the standard C preprocessor

5.1. General Rules for Preprocessor Commands

Preprocessor commands are directives similar to the HMPP directives. All preprocessor commands will start with character '#', to distinguish them from other HMPP directives.

The general syntax for the HMPP preprocessor commands is in FORTRAN:

```
!$hmpp #KEYWORD [ARGUMENTS...]
```

and in C:

```
#pragma hmpp #KEYWORD [ARGUMENTS...]
```

5.1.1. Display Commands

The display commands simply print their arguments.

Syntax:

```
!$hmpp #echo args
```

```
!$hmpp #error args
```

```
!$hmpp #warning args
```

Potential macros in arguments are expanded.

Arguments of `#echo` are printed to the standard output stream. Arguments of `#error` and `#warning` are printed to the standard error stream, prefixed with the location of the command.

A `#error` immediately stops the preprocessing and produces an error code, a `#warning` does not.

Note that the `#echo` command is mostly intended for debug and should not appear in release code.

5.1.2. #PRINT Command

The #print command allows printing the arguments into the output source file.

Syntax:

```
!$hmpp #print args
```

Potential macros in arguments are expanded.

5.1.3. #DEFINE Command without Argument

The #define command associates an arbitrary value to a symbolic name.

Syntax:

```
!$hmpp #define name value
```

The name can be any valid identifier. In the resulting code, the `#define` command is expanded to a single empty line.

After a `#define` command, each occurrence of `\name` or `\{name}` in a HMPP or HMPPCG directive is replaced by the specified value.

The following rules are applied during the definition of a macro:

- The first blank character (space or TAB) after "`name`" is not part of the value.
- The trailing newline character is not part of the value.
- In all directives, the characters \ can be escaped by doubling them as in \\.
- No expansion is performed on the value before affecting the macro.
- A '##' sequence indicates that the text on the left and on the right must be concatenated ignoring all neighboring spaces (same semantic as in CPP).

Example 1: A simple macro usage

```
!$hmpp #define NB 4

!$hmppcg unroll(\NB), noremainder
```

Becomes:

```


!$hmppcg unroll(4), noremainder
```

Example 2: In this example, the `\X1` and `\X2` are both extended to `B` because the `\ARG` in their value is expanded during the `'callsite'` and not during the `#define` statements.

```
!$hmpp #define ARG A
!$hmpp #define X1 \ARG
!$hmpp #define ARG B
!$hmpp #define X2 \ARG
...
!$hmpp Foo callsite , args[\X1;\X2].noupdate
```

Becomes:

```
...
!$hmppc Foo callsite , args[B;B].noupdate
```

5.1.4. #DEFINE Command with Arguments

A macro can be specified with a list of arguments as follow:

Syntax:

```
!$hmpp #define name(arg [, arg ,...]) value
```

For each of the specified arguments, macros of that same can be expanded in the value.

The expansion follows the following rules:

- The argument hides any macro of the same that may exist in the expansion context.
- The argument is only visible during the first level of expansion of value (see the example below)

- The arguments are expanded before the macro;
- Commas ',' and closing parenthesis ')' characters are not allowed in the arguments before their expansion.

Example 1:

```
!$hmpp #define FOO(a,b) From \a to \b

!$hmpp #echo FOO(100,200)
```

Becomes:

```
From 100 to 200
```

5.1.5. #BLOCK and #INSERT without Arguments

The #block command marks the start of a named block of text. The block ends with the corresponding #endblock command.

A block defined can be later inserted using a #insert command

Syntax:

```
!$hmpp #block name
  body
!$hmpp #endblock name
  ...
!$hmpp #insert name
```

The body of the block is arbitrary. It is not interpreted in any ways when the block is defined.

When the #insert directive is encountered, the lines forming the body are inserted and processed according to the usual rules.

Example: A block can be inserted in multiple places

```
!$hmpp #block MyLoad
!$hmpp <MyGroup> MyCodelet1 advancedload, args[A;B;C]
!$hmpp <MyGroup> MyCodelet2 advancedload, args[A;B;C]
!$hmpp #endblock MyLoad

IF (debug) THEN
  !$hmpp #insert MyLoad
ELSE
  PRINT *,'Begin Load'
  !$hmpp #insert MyLoad
  PRINT *,'End Load'
ENDIF
```

Becomes:

```
IF (debug) THEN
  !$hmpp <MyGroup> MyCodelet1 advancedload, args[A;B;C]
  !$hmpp <MyGroup> MyCodelet2 advancedload, args[A;B;C]
ELSE
  PRINT *,'Begin Load'
  !$hmpp <MyGroup> MyCodelet1 advancedload, args[A;B;C]
  !$hmpp <MyGroup> MyCodelet2 advancedload, args[A;B;C]
  PRINT *,'End Load'
ENDIF
```

5.1.6. #BLOCK and #INSERT with Arguments

Blocks can be defined with arguments.

Syntax:

```
!$hmpp #block name(arg1 [, arg2 ,...])
  body
!$hmpp #endblock name
...
!$hmpp #insert name(val1 [, val2 ,...])
```

The `arg`$_1$ `..` `arg`$_n$ are identifiers.

The `val`$_1$ `..` `val`$_2$ are arbitrary expressions with the following restrictions:

- They cannot contain commas `','`
- They cannot contain closing parenthesis `')'`

The rules for processing the arguments in a `#insert` directive are:

- A macro is defined for each `arg`$_1$ `...` `arg`$_n$ using the corresponding val$_1$, ... , val$_n$.
- A macro expansion is applied to val$_1$ ... val$_n$ before affecting arg$_1$, ... , arg$_n$.
- After that expansion val$_1$ ... val$_n$ are allowed to contain commas and closing parenthesis.
- The definition of arg$_1$ .. arg$_n$ is valid for the whole inserted body.
- The macros arg$_1$ ... arg$_n$ are restored to their original value after the `#insert`.
- `#define` or `#undef` to arg$_1$ ... arg$_n$ are only valid within the `#insert` (according to rule just before)
- `#define` or `#undef` applied to any other macros remain valid after the #insert

Example: A simple `#block` and `#insert` with arguments

```
!$hmpp #block myBlock(A,B)
!$hmpp #echo I say \B \A
!$hmpp #echo Hoops! I say \A \B
!$hmpp #endblock myBlock

!$hmpp #insert myBlock(Hello,World)
```

Becomes:

```
I say World Hello
Hoops! I say Hello World
```

6. 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 API	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]	AMD Stream computing. http://ati.amd.com/technology/streamcomputing/index.html
[R2]	NVIDIA developers site, http://developer.nvidia.com/object/cuda.html
[R3]	NVIDIA_CUDA_Programming_Guide_4.0.pdf
[R4]	NVIDIA_CUDA_BestPracticesGuide_4.0.pdf
[R5]	HMPP – NVIDIA GPU, FORTRAN and C Cookbook, CAPS entreprise, 2009.
[R6]	OpenCL, the Khronos consortium. http://www.khronos.org/
[R7]	HMPP HMPP – NVIDIA GPU FORTRAN and C Cookbook, Version 2.0, May 2010
[R8]	App_Note-Running_ATI_Stream_Apps_Remotely.pdf, KB19 - Running ATI Stream Applications Remotely (Knowledge Base, http://developer.amd.com/support/KnowledgeBase/Lists/KnowledgeBase/AllItems.aspx)
[R9]	HMPPWorkbench-3.0_Basics.pdf, CAPS entreprise
[R10]	HMPPWorkbench-3.0_HMPP_Directives_ReferenceManual.pdf, CAPS entreprise.
[R11]	HMPPWorkbench-3.0_Windows_Manual.pdf, CAPS entreprise
[R12]	HMPPWorkbench-3.0_Linux_Manual.pdf, CAPS entreprise
[R13]	HMPPWorkbench-3.0_License_InstallationGuide.pdf, CAPS entreprise
[R14]	HMPPWorkbench-3.0_Basics.pdf, CAPS entreprise