Some implementation details from the smecc compiler

Ronan Keryell (Ronan.Keryell@silkan.com)

December 10, 2012

Contents

1 Introduction **1**

2 OpenMP support **1**

3 Mapping on accelerators **2**
3.1 Mapping on OpenCL 2
3.2 Mapping on STHORM MCA API 6
3.2.1 Addressing model 6
3.2.2 Memory model 7
3.2.3 Runtime execution 7
3.2.4 Code transformation 8

4 SMECY low level hardware API **10**

5 Compilation **12**
5.1 Example of OpenCL mapping 12
5.2 Example of EdkDSP mapping 12
5.3 OpenMP support 12
5.4 Doxygen qualifier 12
5.5 C qualifier 12
5.6 Generating communications from pragma 13
5.7 Compilation for streaming computing 14
5.8 Geometric inference 16

1 Introduction

The shorter the better.

2 OpenMP support

SME-C is a single program model (SPMD) based on OpenMP as its core model with the OpenMP program being the controller of the whole application driving some miscellaneous accelerators.

In our implementation, our compiler keep the OpenMP pragma in the output main program.

The OpenMP pragma inside some piece of code mapped to accelerators may be discarded if it does not make sense, for example if it is translated to an OpenCL kernel or if it is replaced by a piece of hardware (FPGA...).

3 Mapping on accelerators

The mapping of a function call on a specific piece of hardware can be specified with pragma describing where the function is to be run and optionally what arguments have to be transferred before execution and what has to be retrieved after execution:

```
#pragma smecy map(hardware[, unit]*)
#pragma smecy arg(arg_id, arg_clause[, arg_clause]...)
  some_function_call(...);
```

- *hardware* is a symbol representing a hardware component of a given target such as `CPU`, `GPP`, `GPU`, `PE`... They are target specific.
- *unit* entries are optional hierarchical instance number for a specific hardware part. This is typically an integer starting a 0. This hardware number can be an expression of the environment to be able to have a loop managing different accelerators.

3.1 Mapping on OpenCL

The OpenCL can be used to target accelerators like GPU, STHORM or other multicores.

For example the SME-C program

```
1  #define N 1000
   #include <stdio.h>

   void init_array(int a[N][N]) {
     /* The OpenMP part is understood by smecc/Par4All to generate 2D OpenCL
6       workitems */
   #pragma omp parallel for
     for (int i = 0; i < N; i++)
   #pragma omp parallel for
       for (int j = 0; j < N; j++)
11       a[i][j] = 2*i + 3*j;
   }

   int main() {
     int a[N][N];
16
   #pragma smecy map(OpenCL) arg(a, out)
     init_array(a);

     printf("a[27][42] = %d\n", a[27][42]);
21
     return 0;
   }
```

is analyzed by the `smecc` compiler to generate an XML description file to explain to the Par4All compiler that the init_array() function is parallel and has to be transformed to an OpenCL kernel, with some memory transfers.

Then Par4All compiles the previous file to into 2 files, one for the host program running on the main CPU:

```
   /* Use the Par4All accelerator run time: */
   #include <p4a_accel.h>
   P4A_wrapper_proto(p4a_wrapper_init_array, j, i, a);
   /*
5   * file for init_a.c
    */
   #include <stdio.h>
```

```
    void init_array(int a[1000][1000]);
10

    int main();
    //PIPS generated variable
    void p4a_launcher_init_array(int *a);
15  //PIPS generated variable;
    //PIPS generated variable;
    //PIPS generated variable
    void P4A_accel_malloc(void **address, size_t size);
    //PIPS generated variable
20  void P4A_copy_from_accel_2d(size_t element_size, size_t d1_size, size_t d2_size, size_t d1_
    //PIPS generated variable
    void P4A_accel_free(void *address);
    void p4a_launcher_init_array(int *a)
    {
25     //Opencl wrapper declaration
       //PIPS generated variable
       int i;
          int j;
       P4A_call_accel_kernel_2d(p4a_wrapper_init_array, 1000, 1000, j, i, a);
30  }
    void init_array(int a[1000][1000])
    {
       {
          //PIPS generated variable
35        int (*p4a_var_a0)[1000][1000] = (int (*)[1000][1000]) 0;
          P4A_accel_malloc((void **) &p4a_var_a0, sizeof(int)*1000000);

          p4a_launcher_init_array((int *) *p4a_var_a0);
          P4A_copy_from_accel_2d(sizeof(int), 1000, 1000, 1000, 1000, 0, 0, &a[0][0], *p4a_var_a
40        P4A_accel_free(p4a_var_a0);
       }
    }
    int main()
    {
45     P4A_init_accel;
       int a[1000][1000];

    #pragma smecy map(OpenCL) arg(a, out)
       init_array(a);
50

       printf("a[27][42] = %d\n", a[27][42]);

       return 0;
    }
```

and an OpenCL file describing the kernel to be run on the accelerator:

```
1  /** @file

       API of Par4All C to OpenCL for the kernel wrapper and kernel.

       Funded by the FREIA (French ANR), TransMedi\@ (French Pôle de
6      Compétitivité Images and Network) and SCALOPES (Artemis European
       Project project), with the support of the MODENA project (French
       Pôle de Compétitivité Mer Bretagne)

       "mailto:Stephanie.Even@enstb.org"
```

```
11        "mailto:Ronan.Keryell@hpc-project.com"

          This work is done under MIT license.
     */

16   #ifndef P4A_ACCEL_WRAPPER_CL_H
     #define P4A_ACCEL_WRAPPER_CL_H

     /** @defgroup P4A_qualifiers Kernels and arguments qualifiers

21        @{
     */

     /** A declaration attribute of a hardware-accelerated kernel in CL
          called from the GPU it-self
26
          This is the return type of the kernel.
          The type is here undefined and must be locally defined.
     */
     /* change this define to void:
31   #define P4A_accel_kernel inline
     */
     #define P4A_accel_kernel void

     /** A declaration attribute of a hardware-accelerated kernel called from
36        the host in CL

          This is the return type of the kernel wrapper.
          It must be a void function.
          Type used in the protoizer.
41   */
     #define P4A_accel_kernel_wrapper __kernel void

     /** The address space visible for all functions.
          Allocation in the global memory pool.
46    */
     #define P4A_accel_global_address __global

     /** The address space in the global memory pool but in read-only mode.
      */
51   #define P4A_accel_constant_address __constant

     /** The address space visible by all work-items in a work group.
          This is the <<shared>> memory in the CUDA architecture.
          Can't be initialized :
56        * __local float a = 1; is not allowed
          * __local float a;
                     a = 1;        is allowed.
      */
     #define P4A_accel_local_address __local
61

     /** Get the coordinate of the virtual processor in X (first) dimension in
          CL
     */
66   #define P4A_vp_0 get_global_id(0)

     /** Get the coordinate of the virtual processor in Y (second) dimension in
```

```
    CL
*/
#define P4A_vp_1 get_global_id(1)

/** Get the coordinate of the virtual processor in Z (second) dimension in
    CL
*/
#define P4A_vp_2 get_global_id(2)

/*
The OpenCL extension cl_khr_byte_addressable_store removes certain
restrictions on built-in types char, uchar, char2, uchar2, short, and
half. An application that wants to be able to write to elements of a
pointer (or struct) that are of type char, uchar, char2, uchar2,
short, ushort, and half will need to include the #pragma OPENCL
EXTENSION cl_khr_byte_addressable_store : enable directive before any
code that performs writes that may not be supported.
*/
#ifdef cl_khr_byte_addressable_store
#pragma OPENCL EXTENSION cl_khr_byte_addressable_store : enable
#endif

/*
Pragma to support double floating point precision
* */

#ifdef cl_khr_fp64
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
#elif defined(cl_amd_fp64)
#pragma OPENCL EXTENSION cl_amd_fp64 : enable
#else
#warning "Your OpenCL device doesn't support double precision"
#endif

// Required for loop unrolling
#define MOD(x,n) ((x)%(n))
#define MAX0(x,n) max(x,n)



/**
   @}
*/

#endif //P4A_ACCEL_WRAPPER_CL_H
/*
 * file for p4a_kernel_init_array.c
 */
//PIPS generated variable
P4A_accel_kernel p4a_kernel_init_array(int j, int i, P4A_accel_global_address int
*a);
//
//      This module was automatically generated by PIPS
//
;
//PIPS generated variable
P4A_accel_kernel_wrapper p4a_wrapper_init_array(int j, int i, P4A_accel_global_address int
*a);
```

```
    P4A_accel_kernel_wrapper p4a_wrapper_init_array(int j,  int i, P4A_accel_global_address int
    *a)
126 {
       // Index has been replaced by P4A_vp_1:
       i = P4A_vp_1;
       // Index has been replaced by P4A_vp_0:
       j = P4A_vp_0;
131    p4a_kernel_init_array(j, i, a);
    }
    P4A_accel_kernel p4a_kernel_init_array(int j,  int i, P4A_accel_global_address int
    *a)
    {
       if (i<=999&&j<=999)
136       *(a+(1000*i+j)) = 2*i+3*j;
    }
```

The OpenCL is indeed hidden in some higher-level macros beginning with `P4A_` to have terser code. For example `P4A_call_accel_kernel_2d` call the OpenCL API to compile the kernel, stacking the call parameters and launching the kernel with the correct NDRange. This allows to redirect more easily the compilation to CUDA for example by changing the macro definitions.

Implementation limitations:

- an execution flow already mapped on a GPU kernel cannot launch another kernel because of the current OpenCL restriction[1];
- all the program has to be written in C, not C++.

3.2 Mapping on STHORM MCA API

The STHORM platform is a MP-SoC with a 2-core ARM processor Cortex-A9 running Linux and an accelerator fabric with a 2D array of clusters, each with 16 processing elements (PE).

3.2.1 Addressing model

The MCA API contain the MCAPI message passing interface for embedded system devices using an hierarchical addressing model made as a triplet $< domain, node, port >$

The addressing model chosen by CEA for STHORM is to use the domain number to select the clusters or the ARM host and use the node numbers to select a PE inside the cluster selected by the domain number.

The domain numbering chosen by CEA for a cluster $(x, y) \in (\mathbb{N}/n\mathbb{N}) \times (\mathbb{N}/m\mathbb{N})$ inside a $n \times m$ cluster machine is a plain 2D linearization:

$$d = y \times n + x$$

and the ARM host processor has the domain number $n \times m + 1$ with node number 0.

An example for the 2×2 cluster simulator we use is shown on Figure 1.

Since the SME-C programming model is a SPMD model, by default all the code runs on the host AMD processor, which is for example on domain 5 node 0, perhaps with several OpenMP threads.

By using pragma such as

```
#pragma smecy map(STHORM, 1, 3)
```

the execution of a function can be synchronously executed in the STHORM fabric on the PE 3 of cluster 1.

Since for an execution on a PE the ARM processor has to launch a function on it and there are many PEs, the function to be executed has to be large grain.

[1]But this could be done in CUDA 5 with some recent K20 GPU.

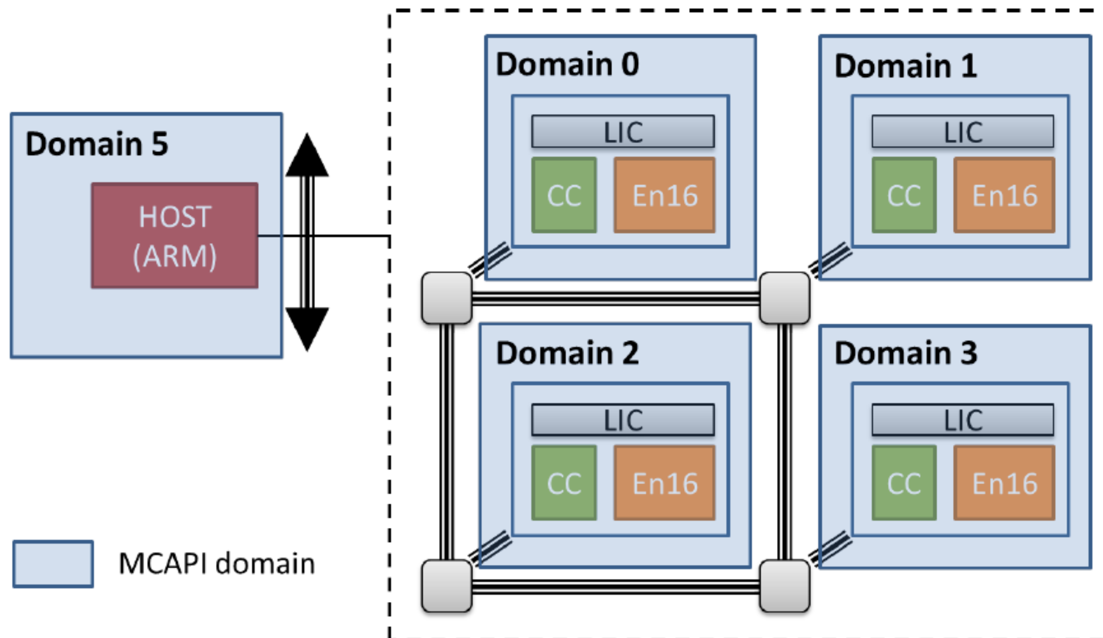

Figure 1: MCAPI addressing model for a STHORM platform with 2×2 clusters.

3.2.2 Memory model

The memory model on STHORM is hierarchical and partially shared between the ARM host processor and the fabric.

The global DDR memory, called the L3 memory, is shared in a weakly coherent way by the host processor and the fabric PEs, through a global L2 cache.

This means that a multithread program running on the various cores and with the right synchronization could run in parallel just by sharing information though the L3 memory. Unfortunately, for power efficiency reasons, there is no system like on a GPU to deal with coalescing multiple accesses to the memory into a big one to improve efficiencyof the memory access. This means that the accesses will be slow.

To improve the efficiency, the local memory shared by the PEs of a cluster or the private memory to a PEhave to be used by placing the data in the right place.

RK→JM: does the L2 cache mitigates this?

RK: double check it it makes sense.

3.2.3 Runtime execution

RK: express the following in some way with pragmas, comm

Memory allocation

Our implementation is based on ST STHORM runtime. During nodes initialization and channels connection we call ST memory allocators. User may want to use directly these memory allocators. If it is the case user can refer to ST STHORM Runtime documentation [6]. Extract of [6] which lists all the memory allocators accessible from CC :

```
void * CC_l3Malloc (int size)
void CC_l3Free (void *ptr)
void * CC_enMalloc (int size)
void CC_enFree (void *ptr)
void * CC_malloc (int size)
void CC_free (void *ptr)
```

3.2.4 Code transformation

The basic compilation scheme to generate MCAPI from the SME-C original code is to generate several files:

- 1 host file running the host code on the Cortex-A9, an OpenMP C or C++ program with some calls to MCAPI functions to interact with the PEs;
- as many source codes there are PEs, with the dual MCAPI function calls to get some data and to respond to the remote procedure calls from the hosts.

In this current implementation, since the host can chose at run-time the PE on which a function will be executed, all the mapped functions are loaded on the PEs and the real function to be executed is selected from a previous MCAPI communication.

For example the `examples/simple_map.c`

```
#include <stdio.h>

#define N 10
#define M 5

void init(int* array, int size, int scale) {
  for (int i = 0; i < size; i++)
    array[i] = i*scale;
}

int main() {
  int tab[N][M];
  /* The schedule(static, 1) enforces each iteration to be executed in a
     different thread, whatever the number of CPU is: */
#pragma omp parallel for schedule(static, 1)
  for (int i = 0; i < N; i++) {
    // Map on STHORM cluster 0 PE i:
#pragma smecy map(STHORM, 0, i)          \
              arg(1,out,[N][M],/[i][])  \
              arg(2,in) \
              arg(3,in)
    init(&tab[i][0], M, i+1);
  }

  for (int i = 0; i < N; i++) {
    printf("Line %d: ", i);
    for (int j = 0; j < M; j++)
      printf("%d ", tab[i][j]);
    puts("");
  }
  return 0;
}
```

is compiled to 2 files, a `STHORM_rose_simple_map.c` file which is to be compiled for the Cortex A9 ARM host with this content:

```
#include <stdio.h>
#define N 10
#define M 5
#include "smecy.h"

int main()
{
  int tab[10UL][5UL];
```

```
/* The schedule(static, 1) enforces each iteration to be executed in a
     different thread, whatever the number of CPU is: */

#pragma omp parallel for schedule ( static, 1 )
13   for (int i = 0; i < 10; i++) {
       SMECY_set(init,STHORM,0,i);
       SMECY_prepare_get_arg_vector(init,1,int,(tab[i] + 0),5,STHORM,0,i);
       SMECY_send_arg(init,2,int,5,STHORM,0,i);
       SMECY_send_arg(init,3,int,(i + 1),STHORM,0,i);
18     SMECY_launch(init,3,STHORM,0,i);
       SMECY_get_arg_vector(init,1,int,(tab[i] + 0),5,STHORM,0,i);
     }
     for (int i = 0; i < 10; i++) {
       printf("Line %d :",i);
23     for (int j = 0; j < 5; j++)
         printf("%d ",tab[i][j]);
       puts("");
     }
     return 0;
28 }
```

the effect of `SMECY_set(init,STHORM,0,i)` is to prepare the selection of `init()` on the STHORM PE of coordinate $(0, i)$ by sending a first MCAPI packet to prepare the execution of the following.

The `STHORM_fabric.c` to be executed on the processors elements of the STHORM fabric that will launch all the threads is constructed by outlining the function calls to be executed to the fabric and only then adding the MCAPI macros. This way, we can have a function object to be given to the STHORM run-time that contains the MCAPI macros:

RK: rewrite the following by using a dispatch loop according to the SMECY_set()

```
#include "smecy.h"
2
// Provide extern call awaited by STHORM MCAPI implementation
void mcapi_domain_entry()
{

7    SMECY_accelerator_loop_begin(init,STHORM,0,i);
     SMECY_set(init,STHORM,0,i);
     SMECY_prepare_get_arg_vector(init,1,int,(tab[i] + 0),5,STHORM,0,i);
     SMECY_send_arg(init,2,int,5,STHORM,0,i);
     SMECY_send_arg(init,3,int,(i + 1),STHORM,0,i);
12   SMECY_launch(init,3,STHORM,0,i);
     SMECY_get_arg_vector(init,1,int,(tab[i] + 0),5,STHORM,0,i);
```

and for each MCAPI kernel to be launched on a PE, a file like `STHORM_PE_simple_map_init.c` with the code for the `init` function in this example with:

```
#include <stdio.h>
2 #define N 10
#define M 5
#include "smecy.h"

void init(int *array, int size, int scale)
7 {
    for (int i = 0; i < size; i++)
      array[i] = (i * scale);
  }

12 void STHORM_PE_simple_map_init()
  {
    SMECY_accelerator_loop_begin(init,STHORM,0,i);
    SMECY_set(init,STHORM,0,i);
```

```
    SMECY_prepare_get_arg_vector(init,1,int,(tab[i] + 0),5,STHORM,0,i);
17  SMECY_send_arg(init,2,int,5,STHORM,0,i);
    SMECY_send_arg(init,3,int,(i + 1),STHORM,0,i);
    SMECY_launch(init,3,STHORM,0,i);
    SMECY_get_arg_vector(init,1,int,(tab[i] + 0),5,STHORM,0,i);
    SMECY_accelerator_loop_end(init,STHORM,0,i);
22 }
```

All the `SMECY_` have different meanings in the 3 different use cases, so that mainly only one code is interpreted differently for the right purpose.

4 SMECY low level hardware API

To call real hardware accelerators, few C macros are needed to interface a program running on some processor to a function running on another processor or in some hardware accelerator.

Since the implementation may depend also on the processor calling the macros (not the same IO bus will be used from an $x86$ or a DSP to call the same operator), a global preprocessing symbol must be defined by the compiler before using these macros, such as

```
#define SMECY_LOCAL_PROC x86
```

or

```
#define SMECY_LOCAL_PROC DSP
```

Few macros are necessary:

RK: to update with an automatic extraction

```
/* Prepare a processing element to execute a function

   @param pe is the symbol of a processing element, such as GPP, DSP, PE...
4  @param[in] instance is the instance number of the processor element to use
   @param func is the function to load on the processor element

   pe is not a string because it is easy to make a string from it but not
   the opposite. The same for func
9 */
#define SMECY_set(pe, instance, func) ...

/* Send a scalar argument to a function on a processing element

14 @param pe is the symbol of a processing element, such as GPP, DSP, PE...
   @param[in] instance is the instance number of the processor element to use
   @param func is the function running on the processor element
   @param[in] arg is the argument instance to set
   @param type is the type of the scalar argument to send
19 @param[in] val is the value of the argument to send
*/
#define SMECY_send_arg(pe, instance, func, arg, type, val) ...

/* Send a vector argument to a function on a processing element
24
   @param pe is the symbol of a processing element, such as GPP, DSP, PE...
   @param[in] instance is the instance number of the processor element to use
   @param func is the function to load on the processor element
   @param[in] arg is the argument instance to set
29 @param type is the type of the vector element to send
   @param[in] addr is the starting address of the vector to read from
              caller memory
   @param[in] size is the length of the vector
*/
```

```
34  #define SMECY_send_arg_vector(pe, instance, func, arg, type, addr, size) ...

    /* Launch the hardware function or remote program using previously loaded
       arguments

39     @param pe is the symbol of a processing element, such as GPP, DSP, PE...
       @param[in] instance is the instance number of the processor element to use
       @param func is the function to load on the processor element

       A kernel can be launched several times without having to set/reset its function.
44  */
    #define SMECY_launch(pe, instance, func) ...

    /* Get the return value of a function on a processing element

49     @param pe is the symbol of a processing element, such as GPP, DSP, PE...
       @param[in] instance is the instance number of the processor element to use
       @param func is the function to load on the processor element
       @param type is the type of the scalar argument to send
       @return the value computed by the function
54  */
    #define SMECY_get_return(pe, instance, func, type) ...

    /* Get a vector value computed by a function on a processing element

59     @param pe is the symbol of a processing element, such as GPP, DSP, PE...
       @param[in] instance is the instance number of the processor element to use
       @param func is the function to load on the processor element
       @param[in] arg is the argument instance to retrieve
       @param type is the type of the vector element
64     @param[out] addr is the starting address of the vector to write in
                   caller memory
       @param[in] size is the length of the vector
    */
    #define SMECY_get_arg_vector(pe, instance, func, arg, type, addr, size) ...
69
    /* Prepare the retrieving of a vector value that will be computed by a
       function on a processing element.

       Typically, it can be used to allocate software or hardware resources
74     for the later SMECY_get_arg_vector().

       @param pe is the symbol of a processing element, such as GPP, DSP, PE...
       @param[in] instance is the instance number of the processor element to use
       @param func is the function to load on the processor element
79     @param[in] arg is the argument instance to retrieve
       @param type is the type of the vector element
       @param[out] addr is the starting address of the vector to write in
                   caller memory
       @param[in] size is the length of the vector
84  */
    #define SMECY_future_get_arg_vector(pe, instance, func, arg, type, addr, size) ...

    /* Reset a processing element to execute a function

89     @param pe is the symbol of a processing element, such as GPP, DSP, PE...
       @param[in] instance is the instance number of the processor element to use
       @param func is the function to unload from the processor element
```

```
    This is used for example to remove consuming resources to decrease
94  power. Giving here the function name may be useful for weird case to
    avoid having short-circuit between CLB in a FPGA during unconfiguring stuff
*/
#define SMECY_reset(pe, instance, func) ...
```

There is also a macro for an asynchronous call and to wait for completion.

5 Compilation

5.1 Example of OpenCL mapping

A typical mapping of the SMECY low level hardware API shown on § 4 would be organized as follows:

- `SMECY_set(PE, proc & 7, invert_vector)`:
  `clCreateProgramFromSource()` + `clBuildProgramExecutable()` + `clCreateKernel()`
- `SMECY_send_arg(PE, proc & 7, invert_vector, 1, int, LINE_SIZE)`:
  `clSetKernelArg()`
- `SMECY_send_arg_vector(PE, proc & 7, invert_vector, 2, int,...)`:
  `clCreateBuffer()` + `clSetKernelArg()` (+ `clEnqueueWriteBuffer()`)
- `SMECY_launch(PE, 0, invert_vector)`:
  `clExecuteKernel()`
- `SMECY_get_arg_vector(PE, proc & 7, invert_vector, 3, ...)`:
  `clEnqueueReadBuffer()`

5.2 Example of EdkDSP mapping

5.3 OpenMP support

From the programmer point of view it may be equivalent to have

- an OpenMP compiler generating SMECY API code such as the parallelism between SMECY target accelerators is run by OpenMP threads dealing each one with an hardware resource sequentially in a synchronous way;
- or a SMECY compiler understanding the OpenMP syntax and generating directly some parallel execution of SMECY accelerators in an asynchronous way.

5.4 Doxygen qualifier

In the Doxygen documentation mark-up language used in comments to detail various entities of a program, there are information such as `in`, `out` or `inout` qualifier on function parameters that can be used to generate the right communication with some hardware accelerators.

5.5 C qualifier

Qualifiers such as `const` attribute qualifier, address space names, register names, accumulator, saturation, etc. are used to generate the right target function or instruction.

5.6 Generating communications from pragma

Of course the pragma information is heavily used in the compilation process.

For example to generate correct communication, the mapping information is used, and from the used/defined information plus optional remapping information, a real communication with an API is used.

```
void
invert_vector(int line_size,
              int input_line[line_size],
              int output_line[line_size]) {
  for(int i = 0; i < line_size; i++)
    output_line[i] = 500 - input_line[i];
}
  [...]
  int image[HEIGHT][WIDTH];
  [...]
#pragma smecy map(PE, proc & 7)           \
              arg(2, in, [1][LINE_SIZE])     \
              arg(3, out, [1][LINE_SIZE])
    // Invert an horizontal line:
    invert_vector(LINE_SIZE,
          &image[HEIGHT - 20 - proc][WIDTH/2 + 2*proc],
          &image[HEIGHT - 20 - proc][WIDTH/2 + 2*proc]);
```

can be compiled into

```
  int image[HEIGHT][WIDTH];
  /* First prepare the PE #(proc & 7) hardware to execute invert_vector.
     That may be used to load some program or microcode, reconfigure a
     FPGA, load/compile an OpenCL kernel... */
  SMECY_set(PE, proc & 7, invert_vector);
  /* Send the WIDTH integer as arg 1 on invert_vector hardware function on
     PE #(proc & 7): */
  SMECY_send_arg(PE, proc & 7, invert_vector, 1, int, LINE_SIZE);
  /* Send a vector of int of size LINE_SIZE as arg 2 on invert_vector
     hardware function on PE #(proc & 7): */
  SMECY_send_arg_vector(PE, proc & 7, invert_vector, 2, int,
                        &image[HEIGHT - 20 - proc][WIDTH/2 + 2*proc], LINE_SIZE);
  // Launch the hardware function or remote program:
  SMECY_launch(PE, 0, invert_vector);
  /* Get a vector of int of size LINE_SIZE as arg 3 on invert_vector
     hardware function on PE #(proc & 7): */
  SMECY_get_arg_vector(PE, proc & 7, invert_vector, 3, &image[HEIGHT - 20 - proc]
                                                              [WIDTH/2 + 2*proc],
                       LINE_SIZE);
```

with low level macros described in § 4.

invert_vector () is either an already implemented function in a hardware library, or it is compiled by a target specific compiler with some callable interface.

For more complex calls needing remapping, such as:

```
    int input_line[LINE_SIZE];
    int output_line[LINE_SIZE];
    /* We need to remap data in the good shape. The compiler should use
       the remapping information to generate DMA transfer for example and
       remove input_line array */
    SMECY_remap_int2D_to_int1D(HEIGHT, WIDTH, HEIGHT/3, 30 + 20*proc,
                               LINE_SIZE, 1, image,
                               LINE_SIZE, input_line);
    // Each iteration is on a different PE in parallel:
```

```
10 #pragma smecy map(PE, proc) arg(2, in, [LINE_SIZE]) arg(3, out, [LINE_SIZE])
       invert_vector(LINE_SIZE, input_line, output_line);
       SMECY_remap_int1D_to_int2D(LINE_SIZE, output_line,
                            HEIGHT, WIDTH, HEIGHT/3, 30 + 20*proc,
                            LINE_SIZE, 1, image);
```

is compiled by using other hardware interfaces involving more complex DMA:

```
1   // May not be useful if this function is already set:
    SMECY_set(PE, proc & 7, invert_vector);
    /* Send the WIDTH integer as arg 1 on invert_vector hardware function on
       PE #(proc & 7): */
    SMECY_send_arg(PE, proc & 7, invert_vector, 1, int, LINE_SIZE);
6   /* Send a vector of int of size LINE_SIZE as arg 2 on invert_vector
       hardware function on PE #(proc & 7) but read as a part of a 2D array: */
    smecy_send_arg_int_DMA_2D_PE_0_invert_vector(3, &image[HEIGHT - 20 - proc]
                                                       [WIDTH/2 + 2*proc],
                                                 HEIGHT, WIDTH, HEIGHT/3, 30 + 20*proc,
11                                               LINE_SIZE, 1);
    SMECY_send_arg_DMA_2D_to_1D(PE, proc & 7, invert_vector, 2, int,
                                &image[HEIGHT - 20 - proc][WIDTH/2 + 2*proc],
                                HEIGHT, WIDTH, HEIGHT/3, 30 + 20*proc,
                                LINE_SIZE, 1);
16  // Launch the hardware function or remote program:
    SMECY_launch(PE, 0, invert_vector);
    SMECY_get_arg_DMA_1D_to_2D(PE, proc & 7, invert_vector, 3, int,
                               &image[HEIGHT - 20 - proc][WIDTH/2 + 2*proc],
                               HEIGHT, WIDTH, HEIGHT/3, 30 + 20*proc,
21                             LINE_SIZE, 1);
```

5.7 Compilation for streaming computing

If we come back on the example shown in § **??** on page **??**, this program is given to the source-to-source stream compiler that may generate the following output (only partially shown, the type definitions are missing):

RK: not longer implemented this way

```
/* Definition of procedures */
int main(void)
{
4     typ_7 data_buffer;
      typ_21 data12Link;
      data12Link = pth_CreateDbLink(512);
      pth_CreateProcess((&__Node1), data12Link);
      pth_CreateProcess((&__Node2), data12Link);
9     pause();
      return 0;
}

static void __outlinedproc2(typ_9 data_buffer)
14 {
      Consume(((typ_4)data_buffer));
}

static void __outlinedproc1(typ_9 data_buffer)
19 {
      Produce(((typ_4)data_buffer));
}

static void __Node1(typ_21 data12Link)
```

```
24 {
       typ_9 data_bufferOut;
       typ_4 tmp0;
       typ_4 tmp1;
       tmp0 = DbLinkGetInitBuff(data12Link);
29     data_bufferOut = ((typ_9)tmp0);
       while(1){
           __outlinedproc1(data_bufferOut);
           tmp1 = DbLinkPutData(data12Link);
           data_bufferOut = ((typ_9)tmp1);
34     }
   }

   static void __Node2(typ_21 data12Link)
   {
39     typ_9 data_bufferIn;
       typ_4 tmp0;
       while(1){
           tmp0 = DbLinkGetData(data12Link);
           data_bufferIn = ((typ_9)tmp0);
44         __outlinedproc2(data_bufferIn);
       }
   }
```

The last two functions are `__Node1()` and `__Node2()`. These are the functions that run in the two processes. These two functions are the wrappers around the user code, which is embedded in the two `__oulined...` functions. These wrappers take care of the data communications. For every communication channel, they manage a double-buffered scheme that minimizes the amount of copying.

In more realistic settings than this simple send/receive scheme, the wrapper function become quickly more complex and non-trivial to maintain manually.

The generated code contains calls to the following communication library:

```
   /*
    * Create a double buffered communication link,
    * with each buffer the given size in bytes */
 4 DbLink createDbLink( int size ) ;
   /*
    * Get a pointer to a free output buffer from the
    * link. The pointer must be given back in the
    * call to DbLinkPutData */
 9 void *DbLinkGetInitBuf( DbLink outputLink ) ;
   /*
    * Get a pointer to data out of the link. This
    * is a read action of the link. The data pointed
    * to is valid until the next read. */
14 void *DbLinkGetData( DbLink inputLink ) ;
   /*
    * Output the data pointed to over the link. The
    * pointer must have been previously obtained from
    * the link. */
19 data_bufferOut = DbLinkPutData( data_bufferOut ) ;
```

And finally there is also the process creation call:

```
1 int pth_CreateProcess( int (*f)(), ... );
```

This library is not intended to become part of the SMECY-API described for example in § **??** or 4. Instead, the library is intended for easy porting to SMECY-API, while taking advantage of

the most efficient communication implementation for the specific target architecture. Currently, a `pthreads`-based implementation exists.

This is just an example of a compilation path and it should be easy to generate a C++ TBB (Thread Building Block) target code using for example the `tbb::pipeline`, a more low-level OpenMP runtime based on `parallel sections` or `parallel task` pragma, or even an MPI or MCA API for distributed memory execution.

5.8 Geometric inference

In the OpenMP SMP model, there is a global memory (well, with a weak coherence model) that may not exist in the execution model of a given target. For example, even if 2 hardware accelerators exchange information through memory according to the high-level programming model, in the real world we may have 2 processors communicating through message passing with MCAPI on a NoC or 2 hardware accelerators connected through a pipeline.

To solve this issue, we use the OpenMP global memory like a scoreboard memory that is used to symbolically relate all the data flows and generates various communication schemes.

Since the memory dependencies are expresses by hyperparallelepipedes, we can do some intersection analysis to compute if a communication is needed or not between 2 devices of the target.