SME-C v0.3

—

C99 with pragma and API for parallel execution, streaming, processor mapping and communication generation

—

SMECY

Rémi BARRÈRE (Remi.Barrere@thalesgroup.com)
Marcel BEEMSTER (marcel@ace.nl)
Ronan KERYELL (Ronan.Keryell@hpc-project.com)

October 25, 2011

Contents

1 Introduction

In the SMECY project we want to use C source code as a portable intermediate representation (IR) between tools from high-level tools down to lower-level tools because of its good trade-off between expressiveness and readability, without compromising portability.

The targets envisioned in SMECY are heterogeneous multicore systems with shared memory or not, with various hardware accelerator, such as ASIC, ASIP, FPGA, GPU, specialized vector processors, partially reconfigurable accelerators...

Unfortunately, since it is undecidable to get high-level properties from such programs, we use decorations to help tools to understand program behaviour and generate codes for some hardware targets. We try to keep clear decorations, easy to understand, so that SMECY C can also used as a programming language.

A SMECY program contains various functions that may be executed on various processors, accelerators, GPU... that may consume and produce data from different physical memory spaces.

Since we want to express also performance on given platforms, we keep the opportunity to have platform-specific pragma and API or specialized intrinsic types and functions, for example to express use of special hardware accelerator functions or operations.

The hardware specific pragma and intrinsics are to be defined between software and hardware partners involved in various use cases. But it may not possible to address all the programming models and platforms envisioned in the project.

1.1 SMECY programming model

The programming model is based on C processes, with a virtual shared memory and threads *à la* OpenMP. Since we may have quite asynchronous processes in a real application description or at the back-end level in the execution model, we can cope indeed with different C processes communicating with an API.

We add mapping information stating on which hardware part a function is to be placed and run.

The programming model exposed in the following is based on an OpenMP SMP model because of its (rather) simple readability, elegance, old background and wide acceptance. We make the hypothesis that a SMECY program is a correct OpenMP program that can be executed in sequential with a C OpenMP-free compiler (just by ignoring OpenMP `#pragma`) and in parallel on a SMP target (such an $x86$ machine) by using an OpenMP compiler *with the same semantics*. Since we use C (by opposition to a DSL) as an internal representation, we choose this behaviour to stick to standard behaviour as much as possible. This is known as the sequential equivalence. Since we can cope different results for performance reasons from executing in parallel non associative floating point operations, we deal with only *weak* sequential equivalence instead of *strong* sequential equivalence.

Of course, this model is incompatible with a real hardware target envisioned in SMECY, so we need to add hints in the code explaining memory dependencies at the function call levels. Since it is quite difficult to describe general dependencies, we approximate memory dependencies with rectangles (and more generally hyperparallelepipede in any dimension) that can be read, written or both. We think these abstractions are good trade-offs between expressiveness (and what a programmer can endure...) and hardware capabilities. Even if there is some similitude with HPF (High Performance Fortran) we do not deal with strides[1].

With this information the tools can guess the communication to generate between the different functions and memory spaces to emulate the global OpenMP memory semantics.

The neat side effect is that we have the same global program executed on all the platforms (sequential, real OpenMP and SMECY) with the same semantics and we can see the sequential version as a functional simulator of the SMECY application and the real OpenMP version as a parallelized quicker version of this simulator.

It is also easy to debug the application, but also all the SMECY tools used or developed in the project.

To be able to address real hardware from the C level with special needs:

- to specify hardware register names;
- define input/output routines specifically but in portable way;
- define fixed point computations;
- specific data size;
- accumulator register (DSP...);
- different memory spaces that can be chosen specifically to optimize storage and speed (DSP, hardware accelerators with scratch pad memory...);
- saturated arithmetic.

For all these we rely on the TR 18037 Embedded C standard, supported for example by ACE tools.

To describe different processes communicating together in an asynchronous way, we do not have anymore a sequential equivalence and then do not use pragma to express this. So we use a simple communication, synchronization and threading API. Since we target embedded systems with a

[1] Indeed, by using some higher dimensional arrays than the arrays used in the application, you can express them... So may be we can express them in the syntax?

light efficient implementation, we can rely on such standard API as the ones of the MultiCore Association: MCAPI (communication), MRAPI (synchronization) and MTAPI (threading).

As modern programs need a clear documentation, such as with Doxygen marking style providing meta information on different elements of the program, we can use this (hopefully correct) information to help the compiling process itself.

Since tools are to be oriented for a specific target, taking into account various hardware and compilation parameters that are to be kept orthogonally to application sources, those parameters are kept aside in some description files that flow between tools. These files may be represented with an XML syntax (using a SMECY naming space) or by even simpler format (JASON, YAML...).

1.2 SME-C Streaming Model

This section describes the state of the newly developed Streaming Annotation to C for use in SMECY. This is a work in progress, so many imperfections still need to be fixed and many extensions are both desireable and possible.

1.2.1 Background

In several of the SMECY applications, is appears to be a suitable parallel computation model to exploit application parallelism. This is the case in particular for two of the Use-Case-A applications, the M5 protocol analyser and the OFDM application.

Streaming has the advantage that it fits well to parallelizing applications that process data in a pipelined fashion, while there may still be strong data dependencies that require data to be processed sequentially at specific points in the pipeline. This is opposed to a data-parallel model, where such dependencies do not exist.

Streaming can exploit both a coarse-grained level of parallelism and parallelism at a fine-grained level. At the fine-grained level, the overhead of passing data between processing nodes in the pipeline must of course be minimized. For fine-grained parallelism, this may require hardware support. To achieve good load balancing, it is important that the processing nodes have a comparable grain size. If one node required much more processing than the others, it becomes the bottleneck and no parallelism can be exploited.

Streaming has the advantage of data locality. Data is passed around in the distributed point-to-point network. Such networks can be implemented far more efficiently than, for example, a shared memory connection between the processing nodes.

1.2.2 Goals

1. The SME-C streaming model is an annotation on top of a valid sequential program. Thus, the program should also work when the streaming annotations are ingored. This eases program development because it allows the validation of the program in an sequential environment.

2. In principle, a streaming program can also be written using OpenMP, but it requires the explicit coding of the communication between nodes. In the SME-C streaming model, the communication is derived from the program source and generated by the compiler. This is very important for the parallel performance tuning of the application because it allows to experiment with different load partitioning without having to re-program process communication.

3. The SME-C streaming model is mapped onto a few basic parallel machine primitives for process creation and communication. The eases the task of retargeting to different target architectures, with shared or distributed memory between the nodes.

1.2.3 Status as of June 2011

ACE has currently implemented a source to source compiler that accepts C programs with the streaming annotation and produces a partitioned program with separate processes (nodes) for each of the nodes in the stream. In addition, it implements the communication links between the nodes.

The generated code includes library calls to implement the low-level tasks of process creation and synchronization. This small library of about 5 calls is currently implemented on top of (shared memory) POSIX pthreads. It is not hard to retarget this library to different underlying run-time systems.

To be used, the Streaming model requires a part of the target application to be rewritten into a particular form, using a while loop and the SME-C stream annotations. Only this part needs to be passed through the source-to-source compiler. Hence, large parts of the application remain unmodified and do not need to be processed by the stream compiler.

Stream termination is currently not handled well. Stream termination has to be mapped from a sequential to a distributed decision process and the design and implementation of that is still to be done.

1.2.4 Future Extensions (not Implementation Related)

The highest priority for extension is to provide a mechanism for stream termination. The challenge here is to make the mechanism such that it still allows a natural programming style under sequential program interpretation.

To facilitate additional parallel performance tuning, a mechanism must be design to allow node replication. It would allow for a single node (that turns out to be a bottleneck) to be replicated into multiple nodes that run on multiple processors. Obviously this complicates the generated communication primitives.

Instead of generating communication primitives, the compiler can also limit itself to only partitioning and generate a communication graph for subsequent processing by tools such as SPEAR and BIPS.

An extension is needed to pass (fixed) parameters and initial values into the streaming nodes that do not turn into communication.

For TVN's H264 application, certain compute intensive loops rely on array processing. Although there is parallelism in these loops, they cannot be easily mapped to a fully data-parallel implementation because of data dependences. Given the nature of these data dependences, it seems to be possible to transform them into a stream-processing model.

1.3 Reference documents

Besides the SMECY documentation, the reader should be knowledgeable of some work of the ISO/IEC JTC1/SC22/WG14 committee on the C language standard:

- ISO/IEC 9899 - Programming languages - C (Technical Corrigendum 3 Cor. 3:2007(E)) `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf`
- TR 18037: Embedded C `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1169.pdf`
- Future C1X standard `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1548.pdf`

and other standards such as

- MCA (MultiCore Association) API: MCAPI, MRAPI & MTAPI, `www.multicore-association.org`
- HPF (High Performance Fortran) `http://hpff.rice.edu/versions/hpf2`

2 Intermediate representation use cases

2.1 Direct programming

TODO: insert here the SMECY use-cases to develop generic use-cases

A programmer can program her application directly in SMECY C with OpenMP and SMECY-specific pragma and API to target a SMECY platform.

It can be at rather high-level, by using only high-level pragma, or rather at a lower level, by using different communicating processes with the API and even specialized API and pragma for specific hardware.

A process written in C code with pragma and API can express a global host controlling process of an application or a local program in a specialized processor. And we may have many of such processes to express different producer and consumer Kahn processes interacting through a NoC in an asynchronous way.

2.2 System high-level synthesis

A compiler can take a sequential plain C or Matlab or another language code, analyze and parallelize the code by adding automatically parallel and mapping pragma. This can be seen as a high-level synthesis at system level.

A tool such as Par4All can do these kinds of transformation.

2.3 Hardware high-level synthesis

A compiler can take a program with SMECY pragma and compile any call with a mapping of a given kind into some hardware configuration or program to be executed instead of the function and an API call to use this hardware part from the host program.

Since the pragma are designed to be concretely compilable, such a tool should be easy to do with a simple compilation framework, such as ROSE Compiler.

The SMECY API and intrinsics are chosen to be mapped quite straightforwardly to real hardware functions by the back-end.

3 Exemples

3.1 Program with contiguous memory transfers

During C memory transfer, if we work on arrays with the last dimension taken as a whole, the memory is contiguous and the programs often work even there are some aliasing such as using a 2D array zone as a linearized 1D vector.

The following program exposes this kind of code where some work sharing is done by contiguous memory blocks.

```
/* To compile this program on Linux, try:

   make CFLAGS='-std=c99 -Wall' pragma_example

   To run:
   ./pragma_example; echo $?
   It should print 0 if OK.

   You can even compile it to run on multicore SMP for free with

   make CFLAGS='-std=c99 -fopenmp -Wall' pragma_example

   To verify there are really some clone() system calls that create the threads:
   strace -f ./pragma_example ; echo $?
```

```
   You can notice that the #pragma smecy are ignored (the project is
   on-going :-) ) but that the program produces already correct results in
   sequential execution and parallel OpenMP execution.

   Enjoy!

   Remi.Barrere@thalesgroup.com
   Ronan.Keryell@hpc-project.com
   for ARTEMIS SMECY European project.
*/

#include <stdbool.h>

/* function Gen

   Example of old C89 array use-case where the size is unknown. Note that
   this implies some nasty access linearization with array with more than
   1 dimension.
 */
void Gen(int *out, int size) {
  // Can be executed in parallel
#pragma omp parallel for
  for (int i = 0; i < size; i++)
    out [i] = 0;
}


/* function Add

   Nice C99 array with dynamic size definition. Note this implies having
   array size given first
*/
void Add(int size, int in[size], int out[size]) {
  // Can be executed in parallel
#pragma omp parallel for
  for (int i = 0; i < size; i++)
    out [i] = in [i] + 1;
}


/* function Test */
bool Test(int size, int in[size]) {
  bool ok = true;
  /* Can be executed in parallel, ok is initialized from global value and
     at loop exit ok is the && operation between all the local ok
     instances: */
#pragma omp parallel for reduction(&&:ok)
  for (int i = 0; i < size; i++)
    /* We cannot have this simple code here:
       if (in [i] != 2)
         exit(-1) ;
       because a loop or a fonction with exit() cannot be executed in parallel.

       Proof: there is a parallel execution interleaving that may execute
       some computations in some threads with a greater i that the one
       executing the exit() done on another thread. So the causality is
       not respected.
```

```
      Anyway, in an heterogenous execution, just think about how to
      implement the exit() operating system call from an
      accelerator... No hope. :-)

      So use a reduction instead and return the status for later
      inspection:
    */
    ok &= (in[i] == 2);

  // Return false if at least one in[i] is not equal to 2:
  return ok;
}


/* main */
int main(int argc, char* argv[]) {
  int tab[6][200];
  // Gen is mapped on GPP 0, it produced (out) an array written to arg 1:
#pragma smecy map(GPP, 0) arg(1, [6][200], out)
  /* Note there is an array linearization here, since we give a 2D array
     to Gen() that uses it . This is bad programming style, but it is just
     to show it can be handled in the model :-) */
  Gen((int *) tab, 200*6);

  // Launch different things in parallel:
#pragma omp parallel sections
  {
    // Do one thing in parallel...
#pragma omp section
    {
      /* Map this "Add" call to PE 0, arg 2 is communicated as input as an
         array of "int [3][200]", and after execution arg 3 is
         communicated out as an array of "int [3][200]"

         Note the aliasing of the 2 last arguments. Just to show we can
         handle it. :*/
#pragma smecy map(PE, 0) arg(2, [3][200], in) arg(3, [3][200], out)
      Add(200*3, (int *) tab, (int *) tab);
    }
    // ...with another thing
#pragma omp section
    {
      /* Map this "Add" call to PE 1, arg 2 is communicated as input as an
         array of "int [3][200]" from address tab[3][0], that is the
         second half of tab, and after execution arg 3 is communicated out
         as an array of "int [3][200]", that is the second half of tab

         Note the aliasing of the 2 last arguments. Just to show we can
         handle it. :*/
#pragma smecy map(PE, 1) arg(2, [3][200], in) \
                         arg(3, [3][200], out)
      Add(200*3, &tab[3][0], &tab[3][0]);
    }
  }

  // Launch different things in parallel:
#pragma omp parallel sections
  {
```

```
#pragma omp section
    {
#pragma smecy map(PE, 2) arg(2, [2][200], in) arg(3, [2][200], out)
      Add(200*2, (int *) tab, (int *) tab);
    }
#pragma omp section
    {
#pragma smecy map(PE, 3) arg(2, [2][200], in) arg(3, [2][200], out)
      Add(200*2, &tab[2][0], &tab[2][0]);
    }
#pragma omp section
    {
#pragma smecy map(PE, 4) arg(2, [2][200], in) arg(3, [2][200], out)
      Add(200*2, &tab[4][0], &tab[4][0]);
    }
  }
  // An example where arg 2 is just used as a whole implicitly:
#pragma smecy map(GPP, 0) arg(2, in)
  bool result = Test(200*6, (int *) tab);
  // Return non 0 if the computation went wrong:
  return !result;
}
```

3.2 Program with non-contiguous memory transfers

In the following example, we apply different computations on square pieces of the image, that do not have contiguous representation in memory. That is why we need to express restrictions on the use of the whole array.

```
/* To compile this program on Linux, try:

   make CFLAGS='-std=c99 -Wall' example_2D

   To run:
   ./example_2D; echo $?
   It should print 0 if OK.

   You can even compile it to run on multicore SMP for free with

   make CFLAGS='-std=c99 -fopenmp -Wall' example_2D

   To verify there are really some clone() system calls that create the threads:
   strace -f ./example_2D ; echo $?

   You can notice that the #pragma smecy are ignored (the project is
   on-going :-) ) but that the program produces already correct results in
   sequential execution and parallel OpenMP execution.

   Enjoy!

   Ronan.Keryell@hpc-project.com
   for ARTEMIS SMECY European project.
*/

#include <stdlib.h>
#include "example_helper.h"
```

```
// Problem size
enum { WIDTH = 500, HEIGHT = 200 };


/* The main host program controlling and representing the whole
   application */
int main(int argc, char* argv[]) {
  int image[HEIGHT][WIDTH];
  unsigned char output[HEIGHT][WIDTH];

  // Initialize with some values
  init_image(WIDTH, HEIGHT, image);

#pragma omp parallel sections
  {
    // On one processor
    // We rewrite a small part of image:
#pragma smecy map(PE, 0) arg(3, inout, [HEIGHT][WIDTH]                    \
                             /[HEIGHT/3:HEIGHT/3 + HEIGHT/2 - 1]          \
                             [WIDTH/8:WIDTH/8 + HEIGHT/2 - 1])
    square_symmetry(WIDTH, HEIGHT, image, HEIGHT/2, WIDTH/8, HEIGHT/3);

    // On another processor
#pragma omp section
    // Here let the compiler to guess the array size
#pragma smecy map(PE, 1) arg(3, inout, /[HEIGHT/4:HEIGHT/4 + HEIGHT/2 - 1] \
                             [3*WIDTH/8:3*WIDTH/8 + HEIGHT/2 - 1])
    square_symmetry(WIDTH, HEIGHT, image, HEIGHT/2, 3*WIDTH/4, HEIGHT/4);

    // On another processor
#pragma omp section
    // Here let the compiler to guess the array size
#pragma smecy map(PE, 1) arg(3, inout, /[2*HEIGHT/5:2*HEIGHT/5 + HEIGHT/2 - 1]   \
                             [WIDTH/2:WIDTH/2 + HEIGHT/2 - 1])
    square_symmetry(WIDTH, HEIGHT, image, HEIGHT/2, WIDTH/2, 2*HEIGHT/5);
  }
  // Here there is a synchronization because of the parallel part end

  // Since there
  normalize_to_char(WIDTH, HEIGHT, image, output);

  write_pgm_image("2D_example-output.pgm", WIDTH, HEIGHT, output);

  return EXIT_SUCCESS;
}
```

3.3 Pipelined example

The example shows a produced/consumer program. At every communication step, a full array of 128 integers is passed from the 'Producer' to the 'Consumer'. The streaming compiler finds out the sizes of the communication buffers from the program.

3.4 Remapping example

Some information can be in a given layout but needed in another layout to be used by a specific hardware accelerator.

```
#include <stdlib.h>
```

```c
#include <stdlib.h>
#include <stdio.h>

#define buffer_length 128
typedef int data_buff[buffer_length];

/* Produce a random buffer

   @param[out] data_buffer is an array initialized with random numbers
*/
void Produce(data_buff data_buffer) {
  for(int i = 0; i < buffer_length; i++)
    /* Note that rand() is not thread-safe but it is OK for this
       example */
    data_buffer[i] = rand();
}

/* Compute the average value of an array and display it

   @param[in] data_buffer is the array to analyze
*/
void Consume(data_buff data_buffer) {
  double average = 0;
  for(int i = 0; i < buffer_length; i++)
    /* Note that rand() is not thread-safe but it is OK for this
       example */
    average += data_buffer[i];
  // Normalize:
  average /= RAND_MAX;
  average /= buffer_length;
  printf("Average = %f\n", average);
}


int main() {
  data_buff data_buffer;
  /* This while-loop is indeed to be executed in a pipelined way according
     to the following pragma: */
#pragma smecy stream_loop
  while(1) {
    // This pragma is optional indeed:
#pragma smecy stage
    Produce(data_buffer);
#pragma smecy stage
    Consume(data_buffer);
  }
  return 0;
}
```

Figure 1: Example of pipelined streamed loop.

```
#include "example_helper.h"

// Problem size
enum { WIDTH = 500, HEIGHT = 200, LINE_SIZE = 100 };

/* Apply some pixel value inversion in a 1D array
 */
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];
}


/* The main host program controlling and representing the whole
   application */
int main(int argc, char* argv[]) {
  int image[HEIGHT][WIDTH];
  unsigned char output[HEIGHT][WIDTH];

  // Initialize with some values
  init_image(WIDTH, HEIGHT, image);

  // Draw 70 horizontal lines and map operation on 8 PEs:
#pragma omp parallel for num_threads(8)
  for(int proc = 0; proc < 70; proc++)
    // Each iteration is on a different PE in parallel:
#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]);

  /* Here we guess we have 5 hardware accelerators and we launch
     operations on them: */
#pragma omp parallel for num_threads(5)
  for(int proc = 0; proc < 5; proc++) {
    /* This is need to express the fact that our accelerator only accept
       continuous data but we want apply them on non contiguous data in
       the array */
    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:
#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);
```

```
  }

  // Convert int image to char image:
  normalize_to_char(WIDTH, HEIGHT, image, output);

  write_pgm_image("remapping_example-output.pgm", WIDTH, HEIGHT, output);

  return EXIT_SUCCESS;
}
```

4 Description of high level process structure

Here should be described the metadata on the process organization.

It should be able to describe process instance, interconnection... TODO

5 SMECY embedded C language

We take as input C99 (ISO/IEC 9899:2007) language with extensions for embedded systems (TR 18037).

Refer to these documents for more information. TODO: comment the following

```
__thread
Thread-Local Storage
_Thread_local
Doc. No.:   WG14/N1364
Date:   2008-11-11
Reply to:   Clark Nelson

http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1364.htm

C++
ISO/IEC JTC1 SC22 WG21 N2659 = 08-0169 - 2008-06-11 proposal
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2659.htm



http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1351.pdf
WG14 N1351
C Language support for multiprocessor application
environments.
Walter Banks
Byte Craft Limited
Canada
February 2009


ISO/IEC JTC1 SC22 WG14 N1275
Date: 2007-10-20
Reference number of document:
ISO/IEC TR 18037
Committee identification: ISO/IEC JTC1 SC22 WG14
SC22 Secretariat: ANSI
http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1275.pdf
```

```
WG14 N1386
Additions to ISO/IEC TR 18037 to support named execution
space.
Walter Banks
Byte Craft Limited
Canada
April 2009
Named execution addition to IEC/ISO 18037
http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1386.pdf
```

6 Description of the SMECY directives

The generic format of SMECY code decorations are language dependent, because if here we describe a SMECY IR implementation based on C, it is indeed more general.

- In C/C++:

  `#pragma smecy` *`clause[[,]clause]... newline`*

  We can use \ at the end of line for continuation information.

- In Fortran:

  `!$smecy` *`clause[[,]clause]... newline`*

  Use `&` at the end of line for continuation information.

- In other languages: use `#pragma` equivalent, if not available, use comments *à la* Fortran. For example in Python:

  `#$smecy` *`clause[[,]clause]... newline`*

  Use also `&` at the end of line for continuation information

In implementations that support a preprocessor, the `_SMECY` macro name is defined to have the decimal value *yyyymm* where *yyyy* and *mm* are the year and month designations of the version of the SMECY API that the implementation supports. If this macro is the subject of a `#define` or a `#undef` preprocessing directive, the behavior is unspecified.

6.1 OpenMP support

SMECY is based on OpenMP. It is not clear yet what level of OpenMP is supported. Since a SMECY platform is made at least form a (SMP) control processor, any OpenMP compliant program can run on it anyway.

But the SMECY tools can use more or less information from available OpenMP decorations available in the code.

6.2 Mapping on hardware

The mapping of a function call on a specific piece of hardware can be specified with

```
#pragma smecy map(hardware[, unit])
  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* is an optional 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.

We can add an `if(`*scalar-expression*`)` to predicate hardware launching according some run-time expression to choose between hardware or local software execution, as in OpenMP with the same syntax. The idea is to be able to do a software execution if the data to process is too small compared to the latency of an hardware accelerator.

Recursion is not supported on hardware-mapped functions. If there are functions called from hardware-mapped functions, they will be automatically inlined (so no recursion allowed in them either). If a function is mapped to a more programmable hardware (GPP), recursions in these called functions should be allowed.

By default, call to hardware accelerators are synchronous, so you may launch the call into an OpenMP thread or into another MCA API process. But simpler way is to use the `async` keyword to launch in an asynchronous way, such as:

```
#pragma smecy map(...) async
```

But the it may be useful to way later for the production of an accelerator to be ready. This is done with the `wait` pragma such as in:

```
#pragma smecy wait(PE,2)
```

6.3 Producer/consumer information

To generate hardware communications where there is only a function call, the compiler need to figure out what is the memory zone used to execute the function and then what memory zone in written by the function *and* that will be used later in the program[2]. From these information, copy-in and copy-out operations can be generates.

6.3.1 Function arguments

```
#pragma smecy arg(arg_id, arg_clause[, arg_clause]...)
  some_function_call(...);
```

Direction directive defines how the data flows between the caller and the function:

- `in` the argument is used by the function;
- `out` the argument is used by the function;
- `inout` the argument is used and produced by the function;
- `unused` the argument is not used and produced by the function.

Argument layout specifies how the argument is used in the function with:

- an optional *array_size_descriptor* such as `[n][m]` expressing that the data is used from the callee point of view as such an array starting at the address given in parameter. If not specified, all the caller argument is used;
- an optional `/`*array_range_descriptor*restriction such as `/[n:m][2][3:7]` expressing that the data is used from the callee point of view as an array with only this element ranges used. If not specified, all the array is used according to its size specified or not. If only some ranges are lacking, all the matching dimension is used. For example `/[4][]` matches the column 4 of an array.

[2]If a function produces something not used later, it is useless to get it back.

The more precise this description is and the less data transfers occur.

If the argument usage is independent from the call site they can be specified only at the function definition level instead of at each call site..

6.3.2 Global variables

Right now we do not deal with sharing information through global variables, because it is more difficult to track. Only function parameters are used to exchange information.

But we can imagine to map global variables with this clause:

```
#pragma smecy global_var(var, arg_clause[, arg_clause]...)
  some_function_call(...);
```

6.4 Stream programming

It is possible to stream a `while` loop in several pipeline stages that execute in parallel and pass information between stages/

The two pragmas used here are:

`#pragma smecy stream_loop`: this indicated the following `while` loop must be turned into a stream of processes;

`#pragma smecy stage`: this acts as a separator between groups of statements and define the boundary of pipeline stages. Only data passing over these separators is turned into communication.[3] Note that the first stage pragma can be eluded since the begin of the loop body define a stage up to the next stage pragma.

You can refere to example shown on figure 1.

6.5 Labelling statement

Since ACE mentioned an interest to name part of a program from other external tools, such as to do some fine mapping, a labeling pragma has been added to name[4] statements:

```
#pragma smecy label(name)
  any_statement;
```

6.6 Remapping specification

TO FINISH

Since we always want a sequential equivalence, that means that the sequential code representing the computation on an accelerator really consume The easy

HPF

6.7 Hardware specific pragma

To be defined in collaboration with the various hardware suppliers of the project (P2012, EdkDSP...).

7 SMECY high-level API

7.1 OpenMP

Since we support OpenMP pragma, we also support OpenMP API that allows for example:

- getting/setting the number of threads;

[3] It used to be `#pragma smecy stream_node(n)` as an instruction to define both a stage node and a way to name it for another later use by ACE. But after København meeting, it appears it is better to use another way to label things only when needed. See § 6.5.

[4] Should it be an atom, a string, a number?

- getting the number of available processors on the current domain;
- manipulating locks.

7.2 MultiCore Association API

There is a reference implementation based on Linux *pthreads* that can be used as an example to port to the various available hardware.

This can be used to express communicating process, asynchronous communications, synchronization, etc.

8 MultiCore Association APIs

Since in this project we deal with communicating processes, asynchronous communications, synchronization, etc. we need an API to be able to express them from inside the processes. We rely on a standard API instead of reinventing on again. The MultiCore Association designed some APIs specially targeted at low resource embedded systems, so We rely on the MCAPI, MTAPI and MRAPI for low memory footprint light-weight communications, threading and synchronization.

There is a free implementation based on POSIX so we can have a working version on any decent operating system for testing the SME-C programs. Since this is a reference implementation based on Linux *pthreads*, it can also be used as an example to port to the various available hardware.

Right now there is 3 different specification we can use in the project.

8.1 Multicore Communication API (MCAPI)

This is the main and first produced API by the MCA[5] dealing with communications between processes.

This standard define basically 3 kinds of communication channels:

- messages: connection-less datagrams, similar to UDP datagrams in IP networking;
- packet channels: connection-oriented, unidirectional, FIFO packet streams;
- scalar channel: connection-oriented, single-word, unidirectional, FIFO scalar streams.

A communication entity in MCAPI is a node, that can be gathered in some domains to add hierarchy, and can represent a process on a processor or a hardware accelerator for example. TODO

Please refer to the "Multicore Communications API (MCAPI) Specification V2.015" document for more information.

8.1.1 MTAPI

8.1.2 MRAPI

8.2 NPL API

NPL is the API defined to program ST P2012 in a native way. Since it is rather at the same level of MCAPI/MRAPI/MTAPI, it should be easy to implement one above the other. Since the MCA APIs are standard, we thing that it is commercially interesting to provide a MCA APIs over NPL or other to widen P2012 usage.

Indeed, in 10/2011 this API seems to have been phased out by ST. So this part should be removed soon... To be kept in the history part.

8.3 EdkDSP API

Define here what is useful in the project.

8.4 OpenCL

Since ST P2012 can be programmed in OpenCL which is also a programming API, a C process can use OpenCL orthogonally with other API. A kernel launching is done by defining the kernel source, the memory zone to use and to transfer and the different parameters of the kernel.

Refer to OpenCL documentation for more information.

9 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:

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

   @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
*/
#define SMECY_set(pe, instance, func) ...

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

   @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
   @param type is the type of the scalar argument to send
   @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

   @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
   @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
*/
#define SMECY_send_arg_vector(pe, instance, func, arg, type, addr, size) ...
```

[5]Note that this MCAPI should not be confused with the MCA APIs, that are the APIs in general designed by the MultiCore Association, even if the WWW site and the documents of this association are not always very clear... This was confusing during the København meeting.

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

   @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.
*/
#define SMECY_launch(pe, instance, func) ...

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

   @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
*/
#define SMECY_get_return(pe, instance, func, type) ...

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

   @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
   @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) ...

/* Reset a processing element to execute a function

   @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
   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.

10 Compilation

10.1 Example of OpenCL mapping

A typical mapping of the SMECY low level hardware API shown on § 9 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()`

10.2 Example of EdkDSP mapping

10.3 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.

10.4 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.

10.5 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.

10.6 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 § 9.

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:
#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:

```
// 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);
/* 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,
                                             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);
  // 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,
                             LINE_SIZE, 1);
```

10.7 Compilation for streaming computing

If we come back on the example shown in § 3.3 on page 10, 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):

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

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

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

static void __Node1(typ_21 data12Link)
{
    typ_9 data_bufferOut;
    typ_4 tmp0;
    typ_4 tmp1;
    tmp0 = DbLinkGetInitBuff(data12Link);
    data_bufferOut = ((typ_9)tmp0);
    while(1){
        __outlinedproc1(data_bufferOut);
        tmp1 = DbLinkPutData(data12Link);
        data_bufferOut = ((typ_9)tmp1);
    }
}

static void __Node2(typ_21 data12Link)
{
    typ_9 data_bufferIn;
    typ_4 tmp0;
    while(1){
        tmp0 = DbLinkGetData(data12Link);
        data_bufferIn = ((typ_9)tmp0);
        __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 */
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 */
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. */
void *DbLinkGetData( DbLink inputLink );
/*
 * Output the data pointed to over the link. The
 * pointer must have been previously obtained from
 * the link. */
data_bufferOut = DbLinkPutData( data_bufferOut );
```

And finally there is also the process creation call:

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

This library is not intended to become part of the SMECY-API described for example in § 7 or 9. 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.

10.8 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.