SMECY Internal Representation

—

C + #pragma

—

SMECY-C or SME-C?

Michel BARRÉTEAU[2] Rémi BARRÈRE[2] Ronan KERYELL[1]

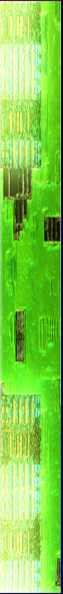
[1]HPC Project
[2]THALES Research & Technology

2011/05/25

—

SMECY General Assembly
Delft

SMECY C programming environment



- Focus classical programming with legacy applications
- Close to classical sequential C with C unified memory model
- Add some `#pragma` to specify SMECY details
- Use cases
  - Direct high-level programming
  - System high-level synthesis
    1. Plain C99 or Matlab or SPEAR-DE or Fortran or DSL or Ptolemy II or...
    2. Tool: analyze and parallelize the code by adding automatically parallel and mapping pragma
    3. SMECY C
  - Hardware high-level synthesis
    1. C99 program with SMECY pragma
    2. SMECY compiler with target description + target API
    3. Executable on SMECY target with host and accelerator parts

Pragma tour (I)

- High Performance Fortran HPF (data parallelism, memory distribution, data remapping...)
- OpenMP 3.1 (data and task parallelism...)
- BlueBee (parallelism & hardware mapping)
- hArtes
- Many others
  - Lot in hardware synthesis world
  - ⚠ Bibliography to finish... Help! Already done by a SMECY partner?

Sequential equivalence

Same semantics

Sequential $\equiv$ Parallel $\equiv$ SMECY

- Do not perturb the programmer... ⚠
  - Sequential execution gives same results as any SMECY target implementation
  - Functional simulator for free! ☺ (think as SystemC...)
  - Easy debug of applications (do not even need of SMECY tools or hardware)
  - Easy debug of SMECY tools too... ☺
  - Can test the concepts before building tools!!!
    $\leadsto$ Do not sequentialize the project! ☺
- OpenMP execution
  - Parallelized version of functional simulator
  - Debug parallelized version of code
  - Only need OpenMP compiler + SMP machine
  - Can run Hellgrind or other execution verifiers ☺

Sequential equivalence

Same semantics

Sequential ≡ Parallel ≡ SMECY

- Do not perturb the programmer...
  - ▶ Sequential execution gives same results as any SMECY target implementation
  - ▶ Functional simulator for free! ☺ (think as SystemC...)
  - ▶ Easy debug of applications (do not even need of SMECY tools or hardware)
  - ▶ Easy debug of SMECY tools too... ☺
  - ▶ Can test the concepts before building tools!!!

    ⇝ Do not sequentialize the project! ☺
- OpenMP execution
  - ▶ Parallelized version of functional simulator
  - ▶ Debug parallelized version of code
  - ▶ Only need OpenMP compiler + SMP machine
  - ▶ Can run Hellgrind or other execution verifiers ☺

SMECY programming model

- Sequential C programming model
- Unified classical C shared memory model
- OpenMP possible on SMP host and accelerators if available in SMECY target
- Some functions can be executed on some hardware accelerators or other processing elements
- No explicit communication between different target memory spaces
- Help compiler with pragma or API to deal with
  - Parallel execution
  - Mapping to specific hardware or processing elements
  - Consumed and produced data
  - Data remapping to cope with hardware constraints
  - Asynchronism & synchronization on hardware resources

Use specific naming space to avoid conflicts with other existing pragma

```
#pragma smecy ...
```

Parallel execution (I)

- Use OpenMP 3.1 `#pragma` syntax & API
- ∃ OpenMP API reference implementation for sequential execution ☺
  - For example `omp_get_num_procs()` return always 1

```
#pragma omp parallel for
  for (int i = 0; i < size; i++)
    out [i] = in [i] + 1;

#pragma omp parallel sections
  {
    {
      Add(200*2, (int *) tab, (int *) tab);
    }
#pragma omp section
    {
      Add(200*2, &tab[2][0], &tab[2][0]);
    }
#pragma omp section
    {
      Add(200*2, &tab[4][0], &tab[4][0]);
```

Parallel execution (II)

```
    }
  }

#pragma omp parallel
{
#pragma omp task
  this_may_be_in_another_task();
}
```

Hardware mapping

- Specify where a function is executed
- Use target-specific identifiers

```
#pragma smecy map(GPP, 0) ...
  bool result = Test(200*6, (int *) tab);

#pragma smecy map(PE, 4) ...
      Add(200*2, &tab[4][0], &tab[4][0]);
```

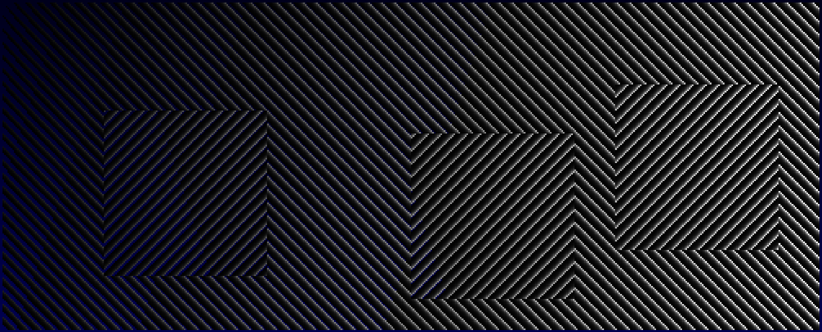
Data flow information *(I)*

- A SMECY compiler needs to add communications around accelerator calls
- Difficult in the general case for a compiler to track information flow
- ⇝ Use annotation to describe memory use-def

```
void Gen(int *out, int size) {
  for (int i = 0; i < size; i++)
    out [i] = 0;
}
    [...]
#pragma smecy map(GPP, 0) arg(1, [6][200], out)
  Gen((int *) tab, 200*6);
    [...]
void Add(int size, int in[size], int out[size]) {
  for (int i = 0; i < size; i++)
    out [i] = in [i] + 1;
}
    [...]
#pragma smecy map(PE, 4) arg(2, [2][200], in) arg(3, [2][200], out)
      Add(200*2, &tab[4][0], &tab[4][0]);
```

Data flow information *(II)*

- Sparse sub-array access

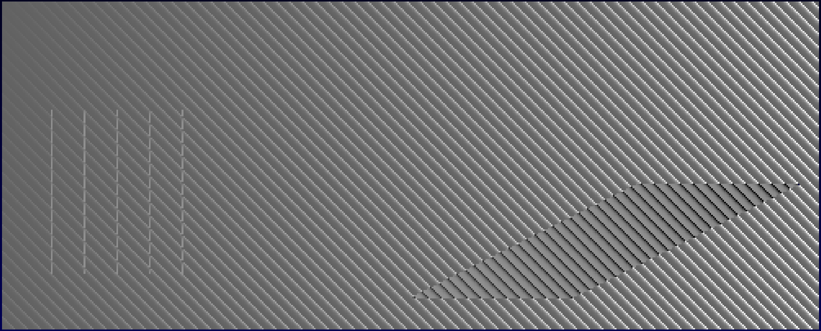


```
#pragma smecy map(PE, 0) arg(3, inout, [HEIGHT][WIDTH]                    \
                            /[HEIGHT/3:HEIGHT/3 + HEIGHT/2 - 1]
                            [WIDTH/8:WIDTH/8 + HEIGHT/2 - 1])
    square_symmetry(WIDTH, HEIGHT, image, HEIGHT/2, WIDTH/8, HEIGHT/3)
```

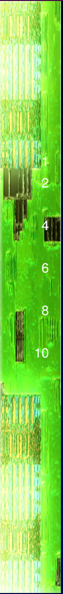
Data remapping (I)

- Some (hardware) functions need to access memory in a specific pattern
  - Vector operation on a part of a 2D or 3D array...
- Need to adapt memory layout between use and function requirements
- Because of sequential equivalence, even the sequential code has this issue
  - ⇝ Use an API instead of a `#pragma`
  - ⇝ Provide SMECY API for non-SMECY target (sequential, OpenMP)
- Example
  - `invert_vector()` operates on continuous memory
  - Can be applied on continuous memory (horizontal line in an image)
  - ... or on discontinuous memory ☹ (vertical line in an image)

Data remapping *(II)*



Data remapping (III)



```
  // 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]);
```

Data remapping (IV)

```
/* 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);
  }
```

Synchronization

- By default, synchronous function calls to accelerators
- Asynchronous execution needs OpenMP threads around accelerator calls
    - May be overkill if a lot of fine grain accelerator calls to do pipelining...
- ⇝ Introduce asynchronous function calls

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

- Rely on synchronization #pragma

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

- Syntax/concept still to finalize with an example of pipelined application...

Compilation

- Lot of information in `#pragma` and API
- Simple use-def analysis
- Simple geometrical array analysis to generate communications
- No need for polyhedral model
- Recycle some concepts from:
  Corinne ANCOURT, Fabien COELHO, François IRIGOIN and Ronan KERYELL. « A Linear Algebra Framework for Static HPF Code Distribution. » in *CPC'93 : Fourth Workshop on Compilers for Parallel Computers.* **Delft, Netherlands, December 1993.** ☺

Conclusion

- Classical C programming and other languages
- `#pragma`tic approach
- Simple `#pragma` & API instead of specific DSL to learn
- No need to define explicit communications
- Can be used to program SMECY applications
- Usable as *a part of the* Internal Representation between SMECY tools
- Should be easy to compile
- Sequential equivalence semantics for easy programming and debugging of applications, tools, with or *without* SMECY compilers and targets
  - ⇝ Few small already examples available and run in sequential and with OpenMP
  - Need to port use-case applications or to generate SMECY C with automatic tools
- Syntax detail of `#pragma` & API still to tweak and contribute!
- Do we need higher level `#pragma` (*pipeline this loop*...)? Different levels? Different tools?

Table of content