An OpenCL code generator for Lustre + 1-synchronous clocks and underspecification

Guillaume Iooss, Dumitru Potop, Marc Pouzet

ENS - PARKAS

November 28, 2019

Part 1 - Introduction

- Performance helps to respect timing requirements
    - Ex: synchronous applications using many FFT/convolutions

$\Rightarrow$ Possible solution: offloading code to an accelerator.

- We focus on the code generation.

Part 1 - Introduction

- Performance helps to respect timing requirements
  - Ex: synchronous applications using many FFT/convolutions

$\Rightarrow$ Possible solution: offloading code to an accelerator.

- We focus on the code generation.
- **Goal:** Show that offloading computation using OpenCL can be done:
  - With few modifications to Lustre code generator
  - With a small Lustre language extension
  - To generate efficient code

Part 1 - Introduction

- Performance helps to respect timing requirements
  - Ex: synchronous applications using many FFT/convolutions

$\Rightarrow$ Possible solution: offloading code to an accelerator.

- We focus on the code generation.
- **Goal:** Show that offloading computation using OpenCL can be done:
  - With few modifications to Lustre code generator
  - With a small Lustre language extension
  - To generate efficient code
- **Structure of this part of the presentation:**
  1. Generating sequential offloaded code
  2. Parallel offloaded code

Basic OpenCL notions

- **Host:** call the OpenCL API
- **Device:** run the OpenCL kernels (accelerator)
  - We assume 1 Host and 1 Device
  - For OpenCL: no communication directly from a device to another

- **Kernel:** Computation to be executed on a Device
- **Buffer:** Memory object, channel between Host and Device

- **Command queue:** enqueue commands to be run on a Device
  - Several command queue can be associated to 1 Device
  - Similar to threads
  - In our case: impose in-order execution

A quick reminder on classical Lustre code generation

- For each Lustre node, generate:
  - *Step function*: computation for a tick
  - *Reset function*: initialize/reset the internal memory

A quick reminder on classical Lustre code generation

- For each Lustre node, generate:
  - *Step function*: computation for a tick
  - *Reset function*: initialize/reset the internal memory
- Running it: main *infinite while loop*
  - 1 iteration = 1 tick of the global clock
  - Acquire the inputs
  - Calls the top Lustre node
  - Return the outputs

Structure of the generated code

- In order to offload a function, we need to:
  1. Build the OpenCL objects (before the while loop)
  2. Use the OpenCL objects to execute a kernel (step function)
  - No reset function needed (no data kept on device)

Structure of the generated code

- In order to offload a function, we need to:
    1. Build the OpenCL objects (before the while loop)
    2. Use the OpenCL objects to execute a kernel (step function)
    - No reset function needed (no data kept on device)

- Main function: Initialize objects (command queue, buffer, kernel)
    - Need to be communicated to the step function

  $\Rightarrow$ Use a global data structure to transmit them.

Structure of the main function

In the main function, OpenCL prelude (before the while loop):

1. Obtain the information about the architecture
   - Create the command queue associated to the device
   - This part of the code is fixed
2. Load and build the kernels
   - One kernel per instance of an offloaded function
   - *Name of the ".cl" file and name of the kernel needed*
   - *Dimension of the kernel needed (dim of thread id)*
3. Create the buffers (one per input/output of every kernels)
   - *Size of the data needed here*
4. Associate the buffers to their kernel
   - Local memory initialization done here
5. Save data inside the global data structure

Structure of the step function

When we generate the code for an offloaded function call:

1. Write the inputs in their buffers
2. Enqueue the computation of the kernel
   - *Need the total number of threads to be used*
   - *Need the number of threads per workgroup*
3. Wait for the completion of the computation
4. Retrieve the outputs in their buffers

Remark: Sequential (Host wait for the kernel on the Device)

Language extension

Information needed by the Code Generator:

- *Buffer-related:*
  - Type of the data transmitted (for the size)
  - Global (i.e., visible input or output) or local memory
- *Kernel-related:* file where its code is, name and dimension.
- *Computation-related:*
  - Total number of threads used by an instance of a kernel
  - Number of threads per workgroup

Language extension

Information needed by the Code Generator:

- *Buffer-related:*
  - Type of the data transmitted (for the size)
  - Global (i.e., visible input or output) or local memory
- *Kernel-related:* file where its code is, name and dimension.
- *Computation-related:*
  - Total number of threads used by an instance of a kernel
  - Number of threads per workgroup

Position of these infos in the language extension:

- Buffer/kernel-related $\rightarrow$ signature of an OpenCL function
- Computation-related $\rightarrow$ at the level of the equation

Language extension - example

Example of program using the OpenCL extension:

```
(* Offloaded computation *)
__clkernel node vector_add(a : int^1024; b : int^1024) returns (c : int^1024)
    __clsource "sum_vector.cl"
    __cldim 1;

(* Main node *)
node main(i1 : int^1024, i2 : int^1024) returns (o : int^1024)
let
    o = __clglobal_worksize 1024 __cllocal_worksize 32
        vector_add(i1, i2) ;
tel
```

OpenCL parallel code generation for Lustre

- **What do we want?**
  - Several threads
  - Several command queues (associated to same Device)
  - Shared memory between all threads, statically allocated

OpenCL parallel code generation for Lustre

- **What do we want?**
  - Several threads
  - Several command queues (associated to same Device)
  - Shared memory between all threads, statically allocated
- **Parallel schedule:** provided by external tool
  - Get a table of scheduling (of equations of main node)
  - Start/end dates are only used as ordering (no deadlock)

(Running) example of scheduling table

Host 1	Host 2	Dev 1	Dev 2
f1			
f3	f2		
	f4	ker1	
	f4		ker2
f5	f6		
f7			

Offloading management as a preprocessing

- OpenCL is thread-safe:
    - Offloading can be done from any thread
- No direct communication between 2 kernels
    - In graph of dependence, add a task (just a copy)
- Two parts:
    - Launch from thread where the oldest input was produced
    - Recover on first thread finishing a task after completion

Synchronization placement as a preprocessing

- Data is associated with the thread producing it.

$\Rightarrow$ Synchronization if consumer is on a different thread.

Synchronization placement as a preprocessing

- Data is associated with the thread producing it.

⇒ Synchronization if consumer is on a different thread.

- **Memory organization:** (shared memory)
  - All memory is allocated from the start
  - Data structure containing addresses (args to all threads)

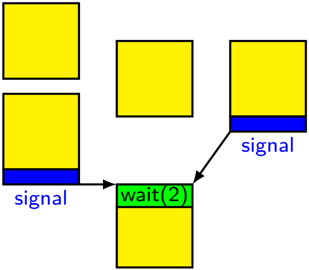
Synchronization placement as a preprocessing

- Data is associated with the thread producing it.

$\Rightarrow$ Synchronization if consumer is on a different thread.

- **Memory organization:** (shared memory)
  - All memory is allocated from the start
  - Data structure containing addresses (args to all threads)

- **Memory transfer/synchronization placement:**



Structure of the generated code

- Main function: prelude for thread/synch init
- One kernel function per column of the scheduling table
  - For one core, no thread creation (code is in the while loop)
- Step function generation: follow the scheduling table (slice)
  - Normal function call
  - Synchronization (transmission or reception)
  - Offloading (enqueue or completion)
- Global barrier in while loop to synchronize all threads together

Part 1 - Conclusion

- Sequential, then parallel OpenCL code generator for Lustre
- Potential improvements:
  - Many options are disabled by default in our code generator
  - Architecture is fixed. Extending to several devices should be simple

Part 1 - Conclusion

- Sequential, then parallel OpenCL code generator for Lustre
- Potential improvements:
  - Many options are disabled by default in our code generator
  - Architecture is fixed. Extending to several devices should be simple
- Implementation: extension to Heptagon
  - Sequential case done
  - Parallel case in progress

Part 1 - Conclusion

- Sequential, then parallel OpenCL code generator for Lustre
- Potential improvements:
  - Many options are disabled by default in our code generator
  - Architecture is fixed. Extending to several devices should be simple
- Implementation: extension to Heptagon
  - Sequential case done
  - Parallel case in progress
- My experience:
  - Using OpenCL = filling "administrative forms"
  - Goes surprisingly well with Lustre code generation scheme

Part 1 finished

Now would be a good time for questions. . .

Part 1 finished

Now would be a good time for questions...

......And now, for a (almost) completely different topic!

Part 2 - Previously - 1-synchronous clock

- Consider **integration program**:
  Top-level node, orchestrating all tasks of an application
  - Multiple harmonic periods (ex: 5 ms / 10 ms / 20 ms / ...)
  - Tasks are present only once per period
- **1-synchronous clocks:** "$(0^k 1 0^{n-k-1})$" (or "$0^k (1 0^{n-1})$")
  with $0 \leq k < n$, $n =$ period and $k =$ phase

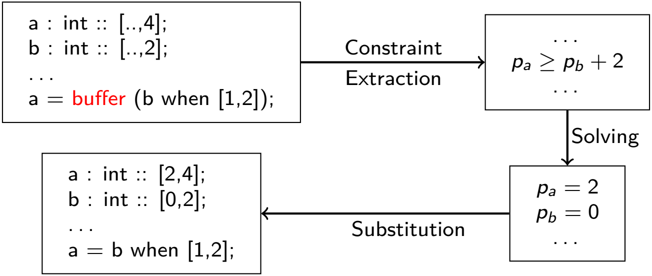
Part 2 - Previously - 1-synchronous clock

- Consider **integration program**:
  Top-level node, orchestrating all tasks of an application
  - Multiple harmonic periods (ex: 5 ms / 10 ms / 20 ms / ...)
  - Tasks are present only once per period
- **1-synchronous clocks:** "$(0^k 10^{n-k-1})$" (or "$0^k(10^{n-1})$")
  with $0 \leq k < n$, $n =$ period and $k =$ phase

- **Last year presentation:**
  Three successive extension to the Lustre language:
  1) Nodes restricted to 1-synchronous clocks
     - Operators: delay(k), specialized when, specialized current
     - Clocking rules
     - Issue: hard to write

Previously - unspecified phase

2) Unspecified phase for 1-synchronous clocks
   - Phases are linear expression of clock variable
   - Buffer operator + various constraints on phases

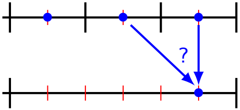


(Note: Implementation available as a Heptagon branch)

Previously - underspecified computation

3) Underspecified computation:
   - Which instance of a value is taken? $\rightsquigarrow$ unspecified by user
   - Compiler decides which value to take

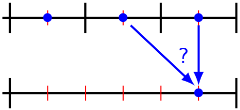
How to use this to relax constraints on phases?
(latency constraints might prevent too much relaxation)



Previously - underspecified computation

3) Underspecified computation:
   - Which instance of a value is taken? $\rightsquigarrow$ unspecified by user
   - Compiler decides which value to take

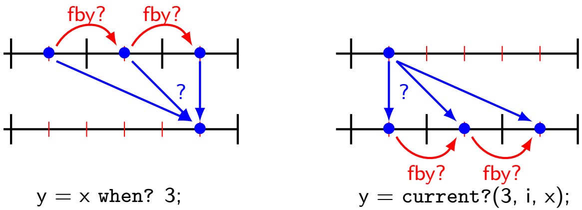
   How to use this to relax constraints on phases?
   (latency constraints might prevent too much relaxation)



- **Operator:** `i fby?`^{n} `expr`.
  - Value: `i fby`^{d} `expr` (with $0 \leq d \leq n$)
  - Determinization: find a value of d for every fby? operator.
  - (Remy Wyss [Asplas12]: "don't care" (`dc`) operator)

Multi-periodic underspecified operators

- `when?` and `current?` operator:
  - Sampled value is underspecified (only the ratio is provided)
  - Can be obtained from `fby?` with syntactic sugar.
  - Determinization: which value is [sub/over]sampled?



⇒ What are their corresponding clocking rule?

Clocking rule for the when? and current?

- `fby?`: Same clocking rule than `fby`
- `when?`: expr must be after the selected (d-th) instance

$$\frac{H \vdash x :: [p, m] \quad m.r = n \quad \begin{array}{c} 0 \leq d < r \\ p + d.m \leq q \end{array}}{H \vdash \mathrm{x}\ \texttt{when?}\ \mathrm{r} :: [q, n]}$$

Clocking rule for the when? and current?

- `fby?`: Same clocking rule than `fby`
- `when?`: expr must be after the selected (d-th) instance

$$\frac{H \vdash x :: [p, m] \quad m.r = n \quad \begin{array}{c} 0 \leq d < r \\ p + d.m \leq q \end{array}}{H \vdash \mathrm{x}\ \texttt{when?}\ \mathrm{r} :: [q, n]}$$

- `current?`: expr must be after the selected (d-th) update

$$\frac{H \vdash i :: [p, n] \quad H \vdash x :: [p, n] \quad m.r = n \quad \begin{array}{c} 0 \leq d < r \\ p - d.n \leq q \end{array}}{H \vdash \texttt{current?}(\mathrm{r}, \mathrm{i}, \mathrm{x}) :: [q, m]}$$

Adding constraint for causality analysis

- **Easy solution:** consider `fby?` as a potential copy
  Example of rejected program:

$$a = 0 \text{ fby? } b;$$
$$b = 0 \text{ fby? } a;$$

Adding constraint for causality analysis

- **Easy solution:** consider `fby?` as a potential copy
  Example of rejected program:

$$a = 0 \text{ fby? } b;$$
$$b = 0 \text{ fby? } a;$$

- **Better solution:**
  - Remy Wyss [Asplas12]: Monoperiodic case (bool constraints)
  - In our case: encode it with linear constraints (using d)
    - Find the cycles of dependence with no `fby`
    - Log the `fby?`
    - Constraint form: $1 \leq d_1 + d_2 + \ldots$
    - Example: $1 \leq d_1 + d_2$.

Part 2 - Conclusion

- Underspecified computation for 1-synchronous computation
  - How to take advantage of them for phase inference?
  - Causality analysis with these operators

Part 2 - Conclusion

- Underspecified computation for 1-synchronous computation
  - How to take advantage of them for phase inference?
  - Causality analysis with these operators

- Do you have any other questions?

Bonus slide - offloading management as a preprocessing

- Example of why we need to add a task on a communication between 2 Devices
- Comms between 2 Devices must go through a Host

$\Rightarrow$ Need a dedicated thread.