MetaFork: the parametric CUDA code generator

Xiaohui Chen Marc Moreno Maza Ning Xie

Department of Computer Science
University of Western Ontario, Canada

February 11, 2016

Overview (1/2)

- The MetaFork-to-CUDA code generator allows the generation of kernels depending on one parameter (or several parameters, in certain cases).
- Typically, these parameters specify thread-block dimension sizes
- The input of our code generator is a meta_schedule statement (from the MetaFork language) as defined in http://dl.acm.org/citation.cfm?id=2886456.
- This code MetaFork-to-CUDA code generator modifies and extends the PPCG code generator whose original code is available at https://www.openhub.net/p/ppcg.
- To be precise, our MetaFork-to-CUDA code generator is based on the version 0.04 of PPCG.

Overview (2/2)

- The code generation follows two algorithms, depending on whether one intends to use shared memory or not in the kernel code.
- In the sequel, we refer to these algorithms as the *shared memory* and *global memory* modes of the code generator.
- These two algorithms are available to the user as two different targets of the `Makefile` for compiling our code generator, see the instructions and examples on the `htttp://www.metafork.org` web site.
- The remaining slides explain the syntax and constraints of the input `meta_schedule` statement.

meta_schedule statement

`meta_schedule { ... }`

- It indicates its body will be launched to hardware accelerators, i.e. NVIDIA GPUs.
- It also transfers the data from CPU to GPU before launching kernels, and transfers the data back from GPU to CPU after executing kernels.
- Note that data transfer between CPU and GPU is automatically detected by this statement.

Supported statements within its body: a sequence of nested for loops

- Each nested `for` loops consist of `parallel for` loops (only 2 or 4) and/or serial `for` loops, and will be translated into a kernel call.
- In the case of `parallel for` loops, it is identified with the '`meta_for`' keyword (see next slide).

meta_for statement

`meta_for (initialize; condition; increment) { ... }`

- `initialize`: 0
- `condition`: < (a variable)
- `increment`: ++ or += 1
- Example: `meta_for (int i = 0; i < upper_bound; i++) { ... }`

The upper bound in the `condition` indicates either the number of threads per thread-block or the number of thread-blocks per kernel.

- Thus, for launching a one-dimension kernel, it requires one outer `meta_for` loop specifying the grid size and one inner `meta_for` loop specifying the block size.
- For a two-dimension kernel, two (immediately) consecutive outer `meta_for` loops are used to specify the grid sizes, and two (immediately) consecutive inner `meta_for` loops are used to specify the block sizes. Moreover, the iterators in the first and second outer (resp. inner) `meta_for` loops correspond to `blockIdx.y` and `blockIdx.x` (resp. `threadIdx.y` and `threadIdx.x`), respectively, in the generated kernel code.

meta_schedule example

```
meta_schedule {
  // only for loops are supported here
  meta_for (int i = 0; i < gridDim.x; i++)
    // only for loops are supported here
    meta_for (int j = 0; j < blockDim.x; j++) {
        ... // nested for-loop body
    }

  // only for loops are supported here
  meta_for (int u = 0; u < gridDim.y; u++)
    meta_for (int i = 0; i < gridDim.x; i++)
      // only for loops are supported here
      meta_for (int v = 0; v < blockDim.y; v++)
        meta_for (int j = 0; j < blockDim.x; j++) {
            ... // nested for-loop body
        }
}
```

Constraints on the input `meta_schedule` statement

Constraints on serial for loops

`for (initialize; condition; increment) { ... }`

- `condition`: the upper or lower bound is a linear expression.
- `increment`: it is increased or decreased by a constant

Constraints in the global memory mode

`array[expression]` or `array[expression][expression]`

- `expression`: one can only use linear expressions as the indices of 1D or 2D arrays.
- However, one can hide non-linear expressions by using a separate statement. For instance, `int p = i * B + j * s + k; array[p] = ...`
  Note that non-linear expressions cannot be analyzed by PPCG, such that whether `array[p]` is reused or coalesced accessed is unknown to PPCG.

Constraints in the shared memory mode (1/2)

`array[expression]` or `array[expression][expression]`

- `expression`: due to the lack of analysis of non-linear expressions, we make the following assumptions.
  - If `expression` is a linear expression, all its variables must refer to the variable counters in the serial `for` loops of the current loop nest.
    In this case, we rely on `PPCG` to analyze the access patterns of the corresponding array.
  - If `expression` contains one and only one non-linear term, that is, one variable multiplying by another, say `i * B`, then one variable must refer to the current block id and the other must refer to corresponding block size.
    One shall hide this non-linear expression by using a separate statement, while this statement shall add a third variable referring to the thread id. For instance, `int p = i * B + j; array[p] = ...`
    Moreover, adding a constant to the above format of the index of `array[]` is supported, say `array[p] = array[p+1]`.
    - For a 2D array, the first (resp. second) `expression` refers to the first (resp. second) dimension of grid and blocks, defined by the first (resp. second) outer and inner `meta_for` loops, respectively.
  - No other forms of non-linear expressions are accepted in `expression`.

Constraints in the shared memory mode (2/2)

In the shared memory mode, not all arrays occurring in the source `MetaFork` code will necessarily have shared memory counterparts in the generated `CUDA` code. In fact, in the addition to the syntax constraints described before, one of the following conditions must also hold:

- If `array[]` (resp. `array[][]`) is written more than once and threads access it in a coalesced fashion, then a shared memory counterpart of `array[]` (resp. `array[][]`) will be generated.
- If `array[]` (resp. `array[][]`) is read and threads access it in a coalesced fashion, then a shared memory counterpart of `array[]` (resp. `array[][]`) will be generated.

When none of those conditions is satisfied, them shared memory counterpart of `array[]` (resp. `array[][]`) is not generated.

How to use the global memory and shared memory modes

- Note that, by default, the compilation of `MetaFork` code, with our extended version of `PPCG`, uses global memory only. This mode is compiled by running `make`) at the root of the source tree of `PPCG`.
- To enable the use of shared memory, one should compile the code generator by issuing `make mem=mlocal`.