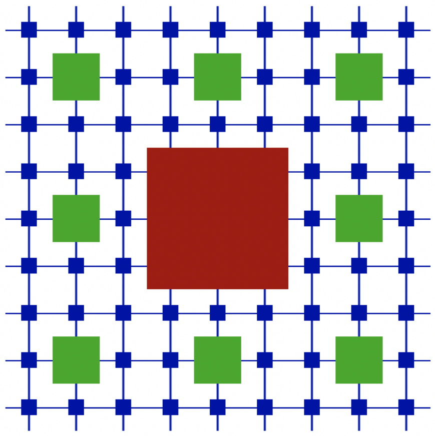
CarpetX

Erik Schnetter <eriks@email.com>
Lucas Timotheo Sanches <lucas.t.s.carneiro@gmail.com>
Steve Brandt <steves@email.com>

October 11, 2023



Abstract

TODO: Maybe change this description

`CarpetX` is a `Cactus` driver based on `AMReX`, a software framework for block-structured AMR (adaptive mesh refinement). `CarpetX` is intended for the `Einstein Toolkit`.

1 Introduction

TODO: This document should cover the topics listed here.
TODO: We should have some words explaining what `CarpetX` is.

1. `loop.hxx/where_t/ghosts_inclusive`
2. What loop functions should be used? `loop_device` or the other ones?
3. Picture of grid regions

2 Building and using standard images

3 Loops over grid elements

Contrary to `Carpet`, in `CarpetX` , loops over grid elements are not written explicitly. Operations that are to be executed for every grid element (cells, edges or points) are specified via `C++` *lambda functions*, also known as closures or anonymous functions.

These objects behave like regular `C++` functions, but can be defined *inline*, that is, on the body of a function or as an argument to another function.

An important concept to grasp with lambda function is *captures*. If a lambda (let us call this the child function) is defined in the body of an outer function (let us call this the parent function), the child can access variables define in the parent function, provided that these variables are *captured*. The two most relevant modes of capture while using `CarpetX` are *capture by reference* (denoted with the `&` sign in the square brackets denoting the start of the lambda) and *capture by value* (denoted by an `=` sign inside the square brackets of the lambda declaration).

When running on GPUs, capture by values are *required*. This is because data must be copied from host (CPU side) memory to device (GPU side) memory in order to be executed.

The API for writting loops in `CarpetX` is provided by the `Loop` thorn. To use it, one must add

```
REQUIRES Loop
```

to the thorn's `configuration.ccl` file and

```
INHERITS: CarpetX
USES INCLUDE HEADER: loop.hxx         # For using CarpetX on CPUs
USES INCLUDE HEADER: loop_device.hxx # For using CarpetX on GPUs
```

to the thorn's `interface.ccl` file. Furthermore, one must include the `Loop` API header file in all source files where the API is needed by adding

```
#include <loop_device.hxx>
```

to the beginning of the source file.

To understand how to utilize the Loop API within Cactus scheduled functions, let us consider the following excerpt from the schedule.ccl file of the WaveToyX thorn, included in the CarpetX repository:

```
    SCHEDULE WaveToyX_RHS IN ODESolvers_RHS
    {
        LANG: C
        READS: state(everywhere)
        WRITES: rhs(interior)
        SYNC: rhs
    } "Calculate␣scalar␣wave␣RHS"
```

This schedule block declares to Cactus that a C++ function (with C linkage) called WaveToyX_RHS should be executed on the ODESolvers_RHS schedule bin (for further information on ODESolvers, see Sec. 6).

The first few lines of C++ source for WaveToyX_RHS read

```
    extern "C" void WaveToyX_RHS(CCTK_ARGUMENTS) {
        DECLARE_CCTK_ARGUMENTSX_WaveToyX_RHS;
        DECLARE_CCTK_PARAMETERS;
        .
        .
        .
    }
```

The macros DECLARE_CCTK_ARGUMENTSX_WaveToyX_RHS, CCTK_ARGUMENTS and DECLARE_CCTK_PARAMETERS allow the thorn writer to access parameters and grid functions declared in the thorn's .ccl files. Note that Cactus now supports the DECLARE_CCTK_ARGUMENTSX_FUNC_NAME macro, where FUNC_NAME is the name of a function declared in the schedule.ccl file. These macros restrict the access of a function to it's schedule-declared grid functions. More importantly, it provides a variable called grid which can be used to access the functionalities of the Loop API.

3.1 Loop regions

Before actually writing any code that iterates over grid elements, one must choose *which* elements are to be iterated over. We shall refer to the set of points in the grid hierarchy will be iterated over when a loop is executed as a *Loop region*. The following regions are defined in the Loop API:

1. All: This region refers to all points contained in the grid. Denoted in code by the all suffix.
2. Interior: This region refers to the interior of the grid. Denoted in code by the int prefix.
3. Outermost interior: This region refers to the outermost "boundary" points in the interior. They correspond to points that are shifted inwards by = cctk_nghostzones[3] from those that CarpetX identifies as boundary points. From the perspective of CarpetX (or AMReX), these do not belong in the outer boundary, but rather the interior. This excludes ghost faces, but includes ghost edges/corners on non-ghost faces. Loop over faces first, then edges, then corners. Modified from loop_bnd_device. Denoted in code by the outermost_int suffix.

TODO: Picture of grid regions

3.2 Loop methods

Remember that the macro `DECLARE_CCTK_ARGUMENTSX_FUNC_NAME` provides a variable called `grid`, which is an instance of either `GridDescBase` or `GridDescBaseDevice` classes which contain functions for looping over grid elements on the CPU or GPU, respectively. The name of each looping method is formed according to

`loop_` + <loop region>+ [`_device`]

For example, to loop over boundaries using the CPU one would call

```
    grid.loop_bnd<...>(...);
```

To obtain a GPU equivalent version, one would simply append `_device` to the function name. Thus, for example, to loop over the interior using a GPU, one would call

```
    grid.loop_int_device<...>(...);
```

Let us now look at the required parameter of loop methods.The typical signature is as follows

```
  template <int CI, int CJ, int CK, ..., typename F>
  void loop_REG_PU(const vect<int, dim> &group_nghostzones, const F &f);
```

The template parameters meanings are as follows:

1. `CI`: Centering index for the first grid direction. Must be set explicitly.
2. `CJ`: Centering index for the second grid direction. Must be set explicitly.
3. `CK`: Centering index for the third grid direction. Must be set explicitly.
4. `F`: The type signature of the lambda function passed to the loop. It is not required to be set explicitly and is automatically deduced by the compiler.

Function parameter meanings are as follows:

1. `group_nghostzones`: The number of ghost zones in each direction of the grid. This can be obtained by calling `grid.nghostzones`.
2. `f`: The `C++` lambda to be executed on each step of the loop.

3.3 Loop Lambdas

We shall now discuss the syntax and the available elements of the lambda functions that are to be fed to the Loop methods described in Section 3.2.

To start, let us be reminded of the general syntax of a lambda function in `C++`:

```
// append ; if assigning to a variable
[capture_parameter] (argument_list) -> return_type { function_body }
```

When running on GPUs, the capture_parameter field used should always be `=` , indicating pass by value (copy) rather than `&`, indicating pass by reference. The `argument_list` of the lambda should receive only one element of type `PointDesc` (which will be described on Sec. 3.4) and the lambda must return no value, which means that `return_type` can be omitted altogether.

This means that a typical lambda passed to a loop method will have the form

```
[=] (const Loop::PointDesc &p) {
  // loop body
}
```

3.4 The `PointDesc` type and loop lambda body

The `PointDesc` type provides a complete description of the current grid element in the loop. The following members are the ones that are expected to be used more often:

1. `I`: A 3-element array containing the grid point indices.
2. `DI`: A 3-element array containing the direction unit vectors from the current grind point.
3. `X`: A 3-element array containing the point's coordinates.
4. DX: A 3-element array containing the point's grid spacings.
5. iter: The current loop iteration.

In the body of a loop lambda, grid functions declared in the thorn's `schedule.ccl` file are available as `GF3D2` objects, which are `C++` wrappers around native `Cactus` grid functions. These objects are accessible by directly calling them as functions taking arrays of grid indices as input. Such indices, in turn can be obtained by directly accessing `PointDesc` members.

3.5 Example: Computing a RHS with finite differences

Let us now combine the elements describe thus far into a single example. Let us suppose that the following system of PDEs is implemented in `Cactus`:

$$\partial_t u = \rho \tag{1}$$

$$\partial_t \rho = \partial_x^2 u + \partial_y^2 u + \partial_z^2 u \tag{2}$$

Let us suppose that the grid functions `u` and `rho` where made available, while grid functions `u_rhs` and `rho_rhs` are their corresponding RHS storage variables. The function that computes the RHS of Eqs. (1)-(2) can be written as

```
extern "C" void LoopExample_RHS(CCTK_ARGUMENTS) {
  DECLARE_CCTK_ARGUMENTS_LoopExample_RHS;
  DECLARE_CCTK_PARAMETERS;

  // The grid variable is implicitly defined via the CCTK macros
  // A 0/1 in template parameters indicate that a grid is vertex/cell
     centered
  grid.loop_int<0, 0, 0>(
    grid.nghostzones,

    // The loop lambda
    [=] (const Loop::PointDesc &p) {
      using std::pow;

      const CCTK_REAL hx = p.DX[0] * p.dX[0];
      const CCTK_REAL hy = p.DX[1] * p.dX[1];
      const CCTK_REAL hz = p.DX[2] * p.dX[2];

      const CCTK_REAL dudx = u(p.I - p.DI[0]) - 2 * u(p.I)
        + u(p.I + p.DI[0])/hx;

      const CCTK_REAL dudy = u(p.I - p.DI[1]) - 2 * u(p.I)
        + u(p.I + p.DI[1])/hy;

      const CCTK_REAL dudz = u(p.I - p.DI[2]) - 2 * u(p.I)
        + u(p.I + p.DI[2])/hz;

      u_rhs(p.I) = rho(p.I);
      rho_rhs(p.I) = ddu;

    } // Ending of the loop lambda
  ); // Ending of the loop_int call
}
```

4 SIMD Vectorization of loops

5 Using flux

6 Time integration using `ODESolvers`

7 Implementing boundary conditions

8 Writing CCL files

9 Adding and controlling AMR

10 Analyzing data

10.1 OpenPMD

10.2 SILO

11 Acknowledgements

References