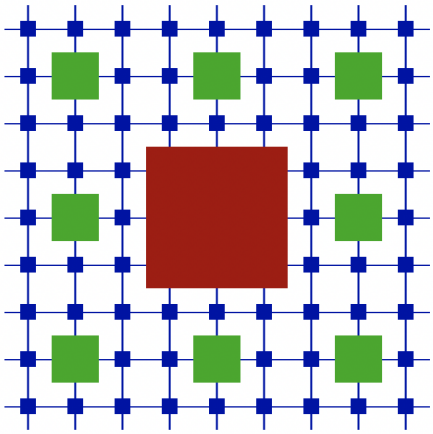
CarpetX

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 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. **??**).

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.

2.1 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

2.2 Non-vectorized loops

Non vectorized loops iterate over grid elements one element at a time without taking advantage of hardware vectorization capabilities (SIMD).

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 function 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<...>(...);
```

Alternatively, it is also possible to execute loops by calling the `loop` method of the `grid` variable and providing an instance of the enumeration `where_t` either as a template parameter (thus specifying the loop location at compile time) or as a regular function parameter (specifying the loop region during runtime). To specify a location using `loop` with template arguments, one would call

```
grid.loop<..., REG>(...);
```

or

```
grid.loop<...>(REG, ...);
```

where `REG` is one of the following

1. `everywhere`.
2. `interior`.
3. `boundary`.
4. `ghosts`

Note that these functions do not have an explicit optional `device` suffix attached to them to inddicate that the loop is to be done on a GPU. Control of the processing unit in which the code is executed in this case is controlled by the type of the `grid` variable and therefore, if one wishes to run on the GPU instead of the CPU, one must be sure to include `loop_device.hxx` instead of simply `loop.hxx`.

Let us now look at the required parameter of loop functions. For loop functions that specify a grid region `REG` and processing unit `PU` in the function's name, the typical signature is as follows

```
template <int CI, int CJ, int CK, int VS = 1, int N = 1, typename F>
void loop_REG_PU(const vect<int, dim> &group_nghostzones, const F &f);
```

The template parameters caen be understood 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. `VS`: Vector size. Its value must be set to 1 when running on GPUs, when running on CPUs that have no SIMD instructions available or when one wishes to disable vectorization altogether. It is not required to be set explicitly and defaults to 1.
5. `N`: TODO: What is this?
6. `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.

```
template <int CI, int CJ, int CK, where_t where, typename F>
loop(const vect<int, dim> &group_nghostzones, const F &f);
```

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

2.3 SIMD Vectorized loops

3 Method of Lines via `ODESolvers`

4 Acknowledgements

Appendix A `C++` lambda function syntax primer

References