# Implementation notes for *nlchains*

Lorenzo Pistone

May 24, 2019

## Contents

# 1 Introduction

The purpose of this document is to give an overview of the source code structure of *nlchains*, to help the user modify it to his or her needs. The code is written in C++14 and CUDA, and a good knowledge of both languages is expected to follow these notes. The end-user aspects of *nlchains* (command line arguments, mathematical details) are detailed in the source code README and in the Software X article, and they are not described here.

The overall structure and coding idioms of *nlchains* is reviewed in Sections 2 and 3; then in Section 4 the general structure of a *nlchains* subprogram is introduced, together with a detailed presentation of a number of utilities; the GPU kernel implementations are commented in Section 5; finally the CPU-only code is commented in Section 6.

# 2 Source code structure and build system

The source code structure of *nlchains* is presented in Figure 1. The code specific to the models is put into separate folders, while the code common to all models is put in the `common/` folder. Within each specific model folder, there is a folder `cpu-implementation/` where the time marching algorithm is implemented on CPU rather than GPU. The `cub/` folder is populated with the required Git submodule CUB (https://nvlabs.github.io/cub/), which provides some low-level GPU primitives.

Many files have the same base name and just a different extension. As a rule of thumb, the `.cu` sources and `.cuh` header contain only code that is specific to the GPU implementation, and it is to be compiled by the CUDA compiler `nvcc`. Files with extension `.cpp` and `.hpp` is to be compiled with the host compiler. This is done because while the CUDA toolchain should in theory be compatible with any standard C++ source file, this is in practice not true, because the internal C++ compiler of `nvcc` may have quirks different to that of the host compiler. For example, it turns out that Boost.MPI headers version 1.66 cause bogus errors when compiling under `nvcc` version 10. Ensuring the maximum separation between host code and GPU kernels minimize the risk of compatibility issues across systems.

## 2.1 The CMake project file

The Cmake project file gathers the required libraries (MPI, Boost.MPI, Boost.Program_options, Armadillo, fftw3), sets up the compiler flags for them, configures the variable `optimized_chain_length`

```
nlchains
├── CMakeLists.txt
├── cub/
├── common/
│   ├── utilities.hpp
│   ├── utilities_cuda.cuh
│   ├── utilities_cuda.cpp
│   ├── utilities_cuda.cu
│   ├── mpi_environment.hpp
│   ├── configuration.hpp
│   ├── loop_control.hpp
│   ├── symplectic.hpp
│   ├── results.hpp
│   ├── results.cpp
│   ├── results_linenergies.cpp
│   ├── wisdom_sync.hpp
│   └── main.cpp
├── DNKG_FPUT_Toda/
│   ├── dispersion.cpp
│   ├── DNKG_FPUT_Toda.cpp
│   ├── DNKG_FPUT_Toda.cu
│   ├── DNKG_FPUT_Toda.hpp
│   └── cpu-implementation/
│       └── DNKG_FPUT_Toda.cpp
├── dDNKG/
│   ├── eigensystem.cpp
│   ├── dDNKG.cpp
│   ├── dDNKG.cu
│   ├── dDNKG.hpp
│   └── cpu-implementation/
│       └── dDNKG.cpp
└── DNLS/
    ├── dispersion.cpp
    ├── DNLS.cpp
    ├── DNLS.cu
    ├── DNLS.hpp
    └── cpu-implementation/
        └── DNLS.cpp
```

Figure 1: The source code tree of *nlchains*.

| Header file in `common/` | Defined functionalities |
|---|---|
| `utilities.hpp` | `ginit = lambda`<br>`destructor(lambda)`<br>`openmp_simd`, `make_simd_clones`, `simd_allocator<T>` |
| `utilities_cuda.cuh` | `assertcu`, `assertcufft`, `assertcublas`<br>`cudalist<T, host>`<br>`completion`, `add_cuda_callback`<br>`cuda_ctx`, `make_kernel_info`<br>`plane2split` |
| `mpi_environment.hpp` | all about MPI: `mpi_global`, `assertmpi`, `quit_requested`, ... |
| `configuration.hpp` | `programs()`<br>`gconf`, `gres` |
| `loop_control.hpp` | `loop_control_gpu`, `loop_control` |
| `symplectic.hpp` | symplectic coefficients: `symplectic_c`, `symplectic_d` |
| `results.hpp` | `results` |
| `wisdom_sync.hpp` | `wisdom_sync` |

Table 1: The location of the defined functionality in the headers of the `common/` folder.

for the relevant GPU implementations, and defines two targets: *nlchains* for both the GPU and CPU implementations, and *nlchains-cpu* for the CPU-only implementation. Additionally, when the project is built in Release mode (`-DCMAKE_BUILD_TYPE=Release`) it probes support for OpenMPI 4.x and uses it, together with the GCC/Clang flag `-ffast-math`, on the source files that contain CPU-intensive operations; these are `common/results_linenergies.cpp` and all the CPU-only implementation of the models. This hints the compiler to vectorize the performance critical loops (more information later in Section 6).

# 3 Coding environment and utilities

In the following we review the coding idioms and the most important utilities that are used throughout the code of *nlchains*. These can be reused to implement new models, and so their semantics are described in details. The location of the functionality explained in this and the following section is show in Table 1.

## 3.1 Error handling

Exceptions are used throughout all the sources as the primary method to handle with errors, and the RAII (Resource Acquisition is Initialization) pattern is used for all resources in *nlchains*. A few utilities are provided to handle with the interface of the libraries that use C-style return code error checking.

When a resource is dynamically allocated, and a cleanup routine should be called to free it, there may be a memory leak if an exception is thrown during execution of following code. One option in this case is to wrap each of this type of resources in a C++ struct with a destructor to properly call the freeing routine. In *nlchains* a different approach is used. It is possible to use the `destructor(lambda)` macro. This macro creates an object whose destructor will call the user defined `lambda` function. For example, this is ensures that a fftw plan is freed when an exception is thrown or the execution completes:

```
auto plan = fftw_plan_dft_1d(...);
destructor([=] { fftw_destroy_plan(plan); });
```

The CUDA runtime, MPI and the cuFFT and cuBLAS libraries employ error code checking as error handling mechanism. To avoid writing a large quantity of boilerplate error checking code, *nlchains* uses a mechanism to translate error codes to exceptions. For each of the aforementioned libraries, four macros are provided: `assertcu`, `assertmpi`, `assertcufft`, and `assertcublas`. These macros have similar usage and have similar functionality, that is they translate an error code in an exception that carries a meaningful error string, including the source code location where the error occurred. These macro must be appended after the relevant library call with the `&&` operator. For example, this is a typical usage of `assertcu` when initializing the first available GPU:

```
cudaSetDevice(0) && assertcu;
```

This allows to pinpoint easily where and why the exception occurred. For example, the following message is printed on terminal when launching *nlchains* on a system where the CUDA kernel modules are not loaded: `exception cuda_error, @ /nlchains/common/utilities_cuda.cpp:14: CUDA driver version is insufficient for CUDA runtime version`.

### 3.2 The MPI environment

The MPI environment is accessed, where possible, through Boost.MPI[1], which is a modern C++ thin layer on top of MPI. Refer to the Boost.MPI documentation for a description of its interface. As a short hand to common MPI operations, the following constant global variables are defined: `mpi_global` is a `boost::mpi::communicator` representing the `MPI_COMM_WORLD` communicator, and `mpi_global_size` is its size; `mpi_global_coord` is the process MPI coordinate within the `MPI_COMM_WORLD` communicator; `mpi_node_coord` is the coordinate across a communicator created from `MPI_COMM_WORLD` with the splitting `MPI_COMM_TYPE_SHARED` (useful to establish a rank among MPI processes on the same host, see the next Section). The MPI environment is initialised at global initialization, that is before the `main()` function of *nlchains*.

[1] `https://www.boost.org/doc/libs/1_70_0/doc/html/mpi.html`

### 3.3 CUDA context handling and utilities

The CUDA context is initialized in *nlchains* by using the following idiom:

```
auto ctx = cuda_ctx.activate(mpi_node_coord);
```

`ctx` is an opaque token that serves the purpose of acquiring the CUDA context in a RAII fashion. `cuda_ctx` is a global variable that holds information about the current context and GPU in the field `cuda_ctx::dev_props` (type `cudaDeviceProp`). The argument to the method `activate()` indicates which GPU to initialize: by using the value of `mpi_node_coord`, each MPI process within a single host will target a different GPU. When the CUDA context is initialized through `cuda_ctx`, argument parsing must already be performed (see Section 4.1), because acquiring the context automatically allocates the GPU buffers according to the simulation parameters.

A few utilities are mainly used in *nlchains* for managing the CUDA kernels. While not necessary for the implementation of new algorithms, it is useful to comment on them for a better understanding of the existing implementation.

#### 3.3.1 Buffer management

GPU buffers are allocated with the template `cudalist<Type, host>`. A `cudalist` is a memory buffer of a number of elements of type `Type`, and the boolean template argument `host` switches between GPU memory or pinned host memory. A `cudalist` can be constructed as follows:

```
//15 elements of type Type, on gpu
cudalist<Type> on_gpu(15);
//15 elements of type Type, on host pinned memory
cudalist<Type, true> on_host(15, wc);
```

The boolean argument `wc` in the constructor for host pinned memory is a boolean that when true turns on the allocation flag `cudaHostAllocWriteCombined`. A `cudalist` has a cast operator overload for the type `Type*`, and can be accessed with the indexing `operator[]`, hence a `cudalist` object can be transparently used on host and GPU kernels as a normal array.

#### 3.3.2 Stream dependency and synchronization

Synchronization between CUDA streams and the host is simplified by using the `completion` struct. A `completion` object is a thin wrapper around `cudaEvent_t` that represent an execution point in a CUDA stream. This execution point can be used to synchronize with other CUDA streams, by using the method `completion::blocks(cudaStream_t s2)`, which instructs the second stream `s2` to wait for the completion of the execution point before launching kernels enqueued subsequently. It can also be used to wait on the host for

the completion on the GPU of the specified event, with the method `completion::wait()`. The constructor of `completion` takes as argument the `cudaStream_t` where the execution point is immediately created. However, a `completion` object does not necessarily refer to the same execution point (the same `cudaEvent_t` object) for its whole lifetime: it can be reassigned to a new execution point using the method `completion::record(cudaStream_t s)`. It can also be default-constructed, and in this case it does not represent any execution point: in this state the `blocks()` and `wait()` methods can still be called, and they are no-ops. A sample usage of a `completion` object is as follows:

```
cudaStream_t s1, s2;
//[...]
a_kernel<<<blocks, threads, 0, s1>>>(...);
completion sync(s1);  //completes after a_kernel
sync.blocks(s2);
//a_dependent_kernel is guaranteed to execute after a_kernel
a_dependent_kernel<<<blocks, threads, 0, s2>>>(...);
//wait on the host for the completion of a_dependent_kernel
sync.record(s2).wait();
```

### 3.3.3 Kernel grid and block size optimization

A non trivial problem in programming with CUDA is to choose correctly the block size in order to maximize occupancy, that is the number of concurrent blocks on the device. There are various limits on the number of blocks that can be launched, their size, the total shared memory, etc., and they are all dependent on the specific GPU in use. The block size then cannot be decided once for all ensemble sizes, and for all hardware. *nlchains* provides a simple utility to find an optimal block and grid size, based on some heuristics. To access this utility, one first creates a `kernel_info` structure by using the macro `make_kernel_info(kernel)`. The returned object provides a method, `linear_configuration(size_t elements)`, which calculates the optimal block and grid size, assuming that each kernel thread accesses a single item out of a total of `elements` items to be worked on. The result is returned in a struct with `int` fields `blocks` and `threads`, that can be passed directly to a kernel invocation. Here is a typical usage of `kernel_info` for a kernel `some_kernel` where each thread applies a transformation on each element of an array of 10000 `double` values:

```
cudalist<double> buffer(10000);
//use static keyword to calculate only once
static auto info = make_kernel_info(some_kernel);
auto launch = info.linear_configuration(10000);
//info.k == some_kernel
info.k<<<launch.blocks, launch.threads>>>(buffer);
```

The heuristic employed in `linear_configuration` uses the CUDA utility `cudaOccupancyMaxPotentialBlockSize()`, and takes into account that the number of blocks should be at least equal to the number of streaming multiprocessors on the GPU in order to utilize them all. The returned block size is always a multiple of 32 (a warp). If the argument passed to `linear_configuration` is not a multiple of the final block size, the grid size is increased by one in order to make sure that all elements are processed: in the kernel it must be ensured that threads do not access out-of-bounds elements. Executing `linear_configuration` prints some debug information on the console about the kernel if the verbose flag is specified on command line, including the register usage, the local memory size, and the chosen block and grid dimensions.

### 3.3.4 Stream host callbacks

Streams provide a way to enqueue callback functions, to be executed when the stream finishes previous work. This functionality can be used to immediately consume the results of the GPU kernels when they are available. The callback is called in the context of a CUDA system thread, and several limitations are imposed on what the callback function can do. In particular, exception may not be thrown. There is also no guarantee on how many callbacks can be executed concurrently, and it turns out that the current implementation uses only one thread to execute all the callbacks sequentially. Care must be taken to avoid deadlocks. In *nlchains*, callbacks are used to calculate the entropy and to write the simulation results to disk in an asynchronous way (see Section 5.3), in order not to halt the main computational task. To simplify error management within a callback, the utility `add_cuda_callback(cudaStream_t s, std::exception_ptr &callback_err, lambda)` is provided. The callback function `lambda` is a functor that takes a single argument, a `cudaError_t`, as passed by the CUDA implementation (see the documentation for `cudaStreamAddCallback()`). The functor `lambda` is wrapped so that exception thrown from it are caught and stored into the `callback_err`, to be later inspected and possibly rethrown by the main thread (see Section 4.3). This is a sample usage of the `add_cuda_callback()` utility:

```
std::exception_ptr cb_exception;
cudaStream_t s;
//[...]
do_work<<<blocks, threads, 0 s>>>(...);
add_cuda_callback(s, cb_exception, [](cudaError_t e){
  if (e) throw std::runtime_error("some error!");
  std::cout<<"No error."<<std::endl;
});
completion(s).wait();
if (cb_exception) std::rethrow_exception(cb_exception);
```

# 4 Initialization and utilities common to all models

Here we outline the initial execution flow of *nlchains*. Each subprogram registers its entry routine (with signature `int(int args, char** argv)`) with a name in the map returned by `programs()`. This must be done before the `main()` function begins execution, that is within a global constructor. The idiomatic way to perform this operation is the following:

```
int subprogram_main(int argc, char *argv[]) { [...] }
ginit = [] {
        ::programs()["subprogram"] = subprogram_main;
};
```

The expression `ginit = function;` schedules a call to `function()` at the time of global initialization. The `main()` function of *nlchains* then essentially checks the first user argument, that is the subprogram name, and calls the appropriate subroutine. At this time, the MPI environment is already initialized, but the CUDA context is not.

The subprogram function now controls all aspects of the execution of the program. However, the workflow for all the implemented model is essentially the same:

1. parse command line arguments
2. initialize resources
3. run the simulation, collecting results.

The source code in the `common/` folder implements some common functions and structures that are of use for all the models. We review in the following the interface for these utilities that cover the following areas:

- parsing of common command line arguments
- allocation of common resources such as host and GPU buffers
- collection of simulation results (ensemble dumps, linear modes energy, entropy)
- synchronization among MPI processes (termination due to user request or entropy limit reached).

## 4.1 Command line arguments and common resources

Some command line arguments, such as the chain length, ensemble size and time step are common to all models. *nlchains* provides the struct `parse_cmdline` to handle all the standard arguments, and additional arguments can be defined. `parse_cmdline` uses Boost.Program_options[2] as the underlying engine for argument parsing. This is a typical usage of the struct:

[2] `https://www.boost.org/doc/libs/1_70_0/doc/html/program_options.html`

| Argument | Populated field and type |
|---|---|
| `--verbose -v` | `bool gconf.verbose` |
| `--initial -i` | `double2* gres.shard_host` |
| `--chain_length -n` | `uint16_t gconf.chain_length` |
| `--copies -c` | `uint16_t gconf.shard_copies` |
| `--dt` | `double gconf.dt` |
| `--kernel_batching -b` | `uint32_t gconf.kernel_batching` |
| `--steps -s` | `uint64_t gconf.steps` |
| `--time_offset -o` | `uint64_t gconf.time_offset` |
| `--entropy -e` | `double gconf.entropy_limit` |
| `--WTlimit` | `enum {NONE, INFORMATION, WT} gconf.entropy_limit_type` |
| `--entropymask` | `std::vector<uint16_t> gconf.entropy_modes_indices` |
| `--prefix -p` | `std::string gconf.dump_prefix` |
| `--dump_interval` | `uint32_t gconf.dump_interval` |

Table 2: The standard arguments of *nlchains* are the corresponding fields of structs `gconf` and `gres`.

1. create an instance of `parse_cmdline` (this sets up the standard arguments)
2. add any custom argument using `parse_cmdline::options`, which is of type `boost::program_options::options_description` (refer to the Boost documentation for a tutorial on how to add new arguments)
3. call the `run(int argc, char *argv[])` method on the instance, forwarding the values that have been passed to the subprogram routine (see previous Section).

In the `run()` method, all the standard arguments are validated, and the appropriate resources are loaded. The results are saved in two separate global structures, `gconf` ("global configuration") for configuration values and `gres` ("global resources") for memory resources. This is a typical usage of the `parse_cmdline` struct, requiring the user to pass a parameter `-f <value>` on the command line:

```
using namespace boost::program_options;
double f;
parse_cmdline parser("Options for my submodel");
parser.options.add_options()
        (",f", value(&f)->required(), "some parameter");
//initialize gconf, gres and f
parser.run(argc, argv);
```

The standard arguments and the corresponding global variables and their types are listed in Table 2. The ensemble of chains is automatically distributed across all the MPI

processes, and each part of the whole ensemble is referred to as "shard". The execution of all the processes is then essentially independent, except for the collection of the dumps and the linear modes energy and entropies.

After the arguments are parsed with `parse_cmdline`, the `gres` global variable is populated with the shard of the ensemble in `gres.shard_host`. The field `gres.linenergies_host` holds a pointer to a buffer `double[gconf.chain_length]`, which will be populated with the linear modes energy of the shard (possibly to be normalized with some implementation-specific constant). When the CUDA context is initialized (see Section 3.3), two additional fields are initialized: `gres.shard_gpu` and `gres.linenergies_gpu`, which are the GPU version of the host fields, and the ensemble shard is copied from the host to the GPU buffer.

## 4.2 Collection of results

An object of type `results` provides the utilities to save the results of a simulation. A `results` objects is used to dump the shards into an single file, and to reduce the per-shard linear modes energy into a single average, over which the entropy is calculated. The entropy is calculated over all the modes (possibly except the mode zero which may have a zero pulsation), but possibly also only on the modes specified by the `--entropymask` argument: in fact, a `results` object uses the mode indices recorded in `gconf.entropy_modes_indices` if the vector is not empty.

The constructor of `results` takes a single boolean argument: if true, the zeroth linear mode is excluded in the calculation of the entropy (to use in the case that the zeroth mode as a null frequency). At construction, a file `<prefix>-entropy` is opened, where `<prefix>` is `gconf.dump_prefix`. Every `gconf.kernel_batching` time steps, the models calculate the linear modes energy per every shard in some model-specific way, and then call the method `calc_linenergies(double norm_factor)` which reduces `gres.linenergies_host` across all the MPI processes and multiplies it for `norm_factor`, which may be used to example to normalize the result of a Fourier transform. The result is accessible to all processes in the field `results::linenergies` (a `std::vector<double>`); the result is dumped to the file `<prefix>-linenergies-<timestep>` by calling the method `results::write_linenergies (uint64_t timestep)`. After `results::linenergies` is populated, the entropy can be calculated with the method `results::calc_entropies()`, which populates the fields `results::WTentropy` and `results::INFentropy` (type `double`). The entropies are dumped to the file `<prefix>-entropy` by calling the method `results::write_entropies( uint64_t timestep)`. The ensemble dump file `<prefix>-<timestep>` is created reading from the `gres.shard_host` buffer by calling the method `results::write_shard(uint64_t timestep)`. Finally, the entropy limit is checked against the user-provided value by calling the method `results::check_entropy()`: if the limit is reached, the global variable `quit_requested` is set (see Section 4.3). All the methods of `results` return a reference to the instance itself, so the calls can be chained, and all methods are expected to be called by all the MPI

processes (they are "collective" calls in the MPI language). Note that this implies that the entropy is calculated locally on all the MPI processes, and they all set their variable `quit_requested` when needed. A sample usage of the `results` struct is as follows:

```
results res(false);
uint64_t current_timestep;
while (1) {     //main simulation loop
  //populate gres.linenergies_host, current_timestep
  //[...]
  res.calc_linenergies(0.5).calc_entropies().check_entropy();
  res.write_entropy(current_timestep);
  if (current_timestep % gconf.dump_interval == 0) {
    res.write_linenergies(current_timestep);
    res.write_shard(current_timestep);
  }
}
```

### 4.3 Main loop control and termination

The main simulation loop may be interrupted with a signal sent from the user (e.g. SIGINT from pressing CTRL+C in the terminal), or by reaching the entropy limit. When a termination condition is encountered, the individual MPI processes cannot exit right away. In fact, different MPI processes maybe be processing different time steps when such condition is met, as the implementation of *nlchains* is fully asynchronous in the launch of GPU kernels and calculation of results. This is the reason why a signal or a entropy limit do not make the processes quit immediately, but rather just set the global flag `quit_requested`, and the actual termination is deferred. The `loop_control_gpu` struct is useful to actually terminate the execution.

A `loop_control_gpu` object can be used for the following purposes:

- keep track of the local current time step
- synchronize the last time step to be calculated
- throttle the launch of asynchronous GPU kernels
- signal the main simulation loop thread that it should quit, either because the simulation has terminated or because an error occurred outside the main thread.

Throttling of kernel launches is needed because otherwise an excessive backlog of kernels may be present on the GPU at the time that the termination conditions are met, and there is no way in the CUDA programming model to cancel a kernel that has already been launched. A `loop_control_gpu` object can treated as a time step counter, as in it can be casted to a

`uint64_t` and it supports a `+=` expression with an integer argument, to increment the time step counter (usually with the value of `gconf.kernel_batching`). The constructor takes as arguments the CUDA stream where the simulation kernels are launched and an integer parameter `throttle_period` (default 2), for the purpose of throttling kernel enqueues. Each time the `+=` operator is called, a `completion` object is created on the stream specified at construction, and put in an internal list: before returning from the `+=` operator, previous `completion` objects are waited for and discarded, until there are only `throttle_period` left in the list. This scheme should ensure that not too may kernels are enqueued on the device, yet new kernels are enqueued while previous kernels are still running, so the GPU is always busy. The method `loop_control_gpu::break_now()` returns `true` if the simulation loop should quit. Internally, the flag `quit_requested` is checked, and if true, the implementation determines among all the MPI processes which one is the most advanced in terms of timestep calculation, and sets that as the final limit of the simulation, by overwriting the value of `gconf.steps`. Here is a typical usage of the `loop_control_gpu` struct:

```
cudaStream_t s1;
//[...]
//create loop_control_gpu object with throttle_period = 2
loop_control_gpu loop_ctl(s1);
while (1) {     //main simulation loop
  if (loop_ctl.break_now()) break;
  //[...] advance gconf.kernel_batching steps on stream s1
  //mark advancement, possibly wait for throttling
  loop_ctl += gconf.kernel_batching;
}
```

The `loop_control_gpu` also provides a field `callback_err` of type `std::exception_ptr`, to be used in conjunction with `add_cuda_callback()` (see Section 3.3.4). The method `break_now()` also checks if the exception pointer is populated, and in such case it immediately rethrows the exception in the main thread.

## 4.4 Planar and split representation of the ensemble

In general the GPU kernel implementations work on the "planar" representation of the ensemble of chains, that is the chain elements are arranged in memory like the dump format, `double2[gconf.shard_copies][gconf.chain_length]`. In some cases however it is useful to work with a different representation, called "split" representation, where the coordinates and momenta are locate in different buffers in memory, and the major index is the chain element index rather than the ensemble element index, that is the coordinates and momenta are two buffers of type `double[gconf.chain_length][gconf.shard_copies]`. The transposition between these two representations is implemented in the `plane2split` struct.

A `plane2split` object is default-constructed, and it reads from `gres.shard_gpu`, the planar representation, and writes the split representation to its fields `coords_transposed` and `momenta_transposed`. The method `plane2split::split(cudaStream_t producer, cudaStream_t consumer)` launches a conversion from planar to split format, waiting for the completion of the `producer` stream to access the `gres.shard_gpu` buffer, and enqueues the kernels in the `consumer` stream. Similarly, the method `plane2split::plane(cudaStream_t producer, cudaStream_t consumer)` starts the inverse transformation, waiting for the `producer` stream to complete before accessing the transposed buffers and enqueuing kernels in the `consumer` stream. The transposition operation is implemented with a call to `cublasZgeam()` of the library cuBLAS.

### 4.5 Symplectic integration coefficients

The coefficient for the symplectic integration of the chains are hard-coded in the arrays `symplectic_c` and `symplectic_d` (following the notation of Yoshida, "Construction of higher order symplectic integrators, Physics letters A 150.5-7 (1990): 262-268), corresponding to the coefficient of the time evolution of the momenta and coordinates. The hard-coded order of the integrator is six. It is easy to turn *nlchains* into an implementation of a higher order integrator: one simply modifies the the arrays in `common/symplectic.hpp` with the new coefficients, and adapts the number of iterations in the loops of the kernels to use all of them. This has not been made selectable at runtime through a command line option, because the GPU kernels need the size of `symplectic_c` and `symplectic_d` to be known at compile time in order to be compiled optimally (see Section 5).

Note that the integration algorithm has a symmetric form, that is the evolution operator is in the form (e.g. in the case of the sixth order)

$$e^{c[0]\delta t M} e^{d[0]\delta t C} e^{c[1]\delta t M} e^{d[1]\delta t C} \ldots e^{c[6]\delta t C} e^{d[6]\delta t M} e^{c[7]\delta t C}$$

where $c =$`symplectic_c`, $d =$`symplectic_d`, $M$ is the momenta evolution operator and $C$ the coordinates evolution operator, and $c[0] = c[7]$, $c[1] = c[6]$ etc. and $d[0] = d[6]$, $d[1] = d[5]$ etc. This symmetry is common to all higher order of the Yoshida algorithm. Due to this symmetry, it is always possible to merge the evolution $e^{c[7]\delta t M}$ of a time step with the evolution $e^{c[0]\delta t M}$ of the next time step, of course taking care not to perform this optimization when results are going to be collected (hence the ensemble must not be in state which is half-way through a time step).

## 5 The GPU models implementations

Here we document some implementation details of the GPU kernels for the modesl that are readily available in *nlchains*. The general structure of the models main host function is the one outlined in Section 4, plus trivial details such as additional resource allocations. Here we comment only on the actual GPU kernel implementation.

## 5.1 The real models

We detail in particular the four different implementations of the real models, that is dDNKG, DNKG, FPUT and Toda, and the meaning of the `--split_kernel` command line argument. These four models are very similar to each other. The DNKG, FPUT and Toda models are implemented in a single source directory (`DNKG_FPUT_Toda/`), as they share most of the implementation. The integration algorithm is implemented as a template, to be instantiated with different right-hand side equations. The dDNKG model is implemented in a separate folder (`dDNKG/`) despite being very similar to the DNKG model. This is because the calculation of the linear modes energy is very different from that of the DNKG model (an explicit projection on the eigenvectors is needed), and because the presence of site-specific coefficients in the calculation of the potentials would require an unnecessary generalization of the code in the `DNKG_FPUT_Toda/` folder. The discussion that follows however, on the implementation of the kernels for different chain length size, applies to all four models.

### 5.1.1 Chain updated in a single warp

The most performant implementation of these models loads a chain in GPU registers and updates it, without storing the result in global memory until `gconf.kernel_batching` steps have been run. This is possible because register space is more abundant in GPU than CPU. However, the code cannot be generic in the size of the chain, because dynamic addressing of arrays in register memory is not possible. This is the reason behind the `optimized_chain_length` build-type parameter: by storing the chain length as a compile-time constant, the compiler can unroll all the loops that access the chain elements sequentially, and assign them to specific registers and act directly on them without indexing. With this technique, the performance is close to the theoretical limit for operations on `double` data. The kernel `move_chain_in_warp`, implemented in `DNKG_FPUT_Toda/DNKG_FPUT_Toda.cu` and `dDNKG/dDNKG.cu` employs this technique.

A single warp (32 threads) loads and operates on a single chain, so that no block synchronization is needed. When the number of chain elements is greater than 32 (the size of the warp), some threads operate on more than a single chain element. The elements of a chain are distributed in order to minimize the numbers of predicated operations when the chain length is not a multiple of 32: the last `(31*gconf.chain_length)%32` lanes in a warp contain one element less than the other lanes. The elements that a thread operates on are always contiguous, in order to simplify the logic to communicate the elements boundary elements to the neighbour thread. For example, for a chain length of 62, the first 30 thread load in memory two elements: the thread index 0 loads elements 0 and 1, thread index 1 loads elements 2 and 3, etc., while thread index 30 and 31 load elements 62 and 63 respectively. To evaluate the nearest neighbour interaction, a thread with index `t` that operates on the elements from index `e_min` to `e_max` (inclusive) communicates the element `e_max` to the thread `(t+1)%32` and the element `e_min` to the thread `(t+31)%32`. The

communication of adjacent elements is efficient as war shuffle operations are used (via the CUB library). This raises the minimum compute capability to 3.0.

This kernel model is obviously limited by the number of available register. It is not possible to give a definite maximum value of `optimized_chain_length` as it largely depends on the compiler version and the target architecture. As a rule of thumb, since two registers are needed to hold a `double` value, and since the maximum number of registers in a thread is 255, the theoretical limit for the chain length is 2048, however the compiler may chose to start spilling variables to local memory much earlier than that: it is advised to check that the kernel does not actually use local memory by launching *nlchains* with the verbose (`-v`) flags (see Section 3.3.3).

### 5.1.2 Chain updated in a single thread

When the number of elements in a chain is less than 32, a warp would not be fully utilized in the previous scheme. However, it becomes viable to hold an entire chain in a thread, and have the threads work independently of each other. This is the scheme implemented in the kernels `move_chain_in_thread`. The kernel is a template in the length of the chain, and it is instantiated recursively for all chain lengths from 2 to 31 (see helper struct `thread_kernel_resolver` in `DNKG_FPUT_Toda/DNKG_FPUT_Toda.cu` and `dDNKG/dDNKG.cu`): this has the purpose of making the chain length a compile-time constant, for the same reasons explained in the previous section about the variable `optimized_chain_length`. Since a single thread works on a whole chain, the code is much more similar to a traditional CPU implementation, and all the threads in a warp operate on the same chain element index at the same time.

### 5.1.3 Chain updated in split format

When *nlchains* has not been compiled with a value of `optimized_chain_length` matching the chain length required at runtime, or when the `--split_kernel` argument is specified, *nlchains* uses a generic, less efficient kernel implementation, where each element is loaded from global memory, updated, and then stored back in global memory. The kernel function name is `move_split`. As the name suggest, this kernel uses the split representation of the ensemble, in order to maximize the efficiency of memory loads. In fact, similarly to the `move_chain_in_thread` kernel, each thread operates on a single chain, and all threads operate on the same chain element index at the same time: in the planar format this would result in strided load operations, whereas in the GPU architecture contiguous loads from all the threads in a warp are maximally efficient.

In order to minimize the number of memory loads and stores, this kernel iterates over the chain element indexes `0<=i<gconf.chain_length`, and it updates the coordinate `i` and the momentum `i-1`, keeping in register memory the elements necessary for the computation of the next index. The main drawback of this scheme (besides the large number of memory

operations) is that the compiler cannot unroll the iterations of this loop, as there are cross-iteration dependencies. This implementation should be considered a last resort and *nlchains* should always be recompiled with the correct value for `optimized_chain_length`.

### 5.1.4 Caching of nearest-neighbour interactions

In all the real nearest-neighbour models, the nearest-neighbour interaction is calculated in physical space. The equation of motion of a particle includes two force terms, due to the interaction with the particle to the left and to the right. For the coordinate $q_i$ the interaction force is $F(q_{i+1} - q_i) - F(q_i - q_{i-1})$, while for the particle $q_{i+1}$ the force is $F(q_{i+2} - q_{i+1}) - F(q_{i+1} - q_i)$. It is evident that the value of $F(q_{i+1} - q_i)$ is used twice with an inverted sign across neighbour particles, and as such can be cached and reused once. This optimization is particularly important in the Toda model, as the exponential is an expensive operation. The models right-hand sides are implemented in `DNKG_FPUT_Toda/DNKG_FPUT_Toda.cu` as a template struct, `RHS`: the constructor takes the zeroth particle and the last particle, in order to calculate and cache the left interaction of the zeroth particle. Every time the right-hand side of the equation is calculated through this struct, the cached value is used for the interaction to the left with an inverted sign, then the interaction to the right is calculated, stored in the cache, and summed to the other interaction terms.

### 5.1.5 The projection operation for the dDNKG model

While on all the implemented models the calculation of the linear modes energy implies a Discrete Fourier Transform, for the dDNKG model an explicit projection over the eigenvectors. The operation can be mapped to a matrix multiplication, for which CUDA provides the library cuBLAS. The projection must be performed both for the coordinates and the momenta, separately. Given a chain length $N$ and a size of the ensemble $C$, the eigenvector matrix has dimensions $N \times N$ where each eigenvector is a row (in the row-major ordering of C++), while in the split format the coordinates and momenta are matrices with dimension $N \times C$. However, cuBLAS interprets matrices in column-major order, consequently the eigenvector matrix is seen effectively transposed by cuBLAS, so the first index is the element index and the second is the eigenvector index, while the ensemble in split format is interpreted as two $C \times N$ matrices, the second index being the element index within a chain. The `cublasDgemm` function is used to multiply the ensemble matrix by the eigenvector matrix, $(C \times N) \times (N \times N)$, which results in a (column-major) $C \times N$ matrix, where the second index is the eigenvector index. This projection matrix is then reduced by an additional kernel in order to obtain the average linear modes energy, that is a vector of length $N$ which is stored into `gres.linenergies_gpu`.

### 5.2 The DNLS model

The DNLS model implementation essentially consist in a large number of Discrete Fourier Transforms implemented with the library cuFFT. The evolution of the linear and nonlinear operators are local, hence the implementation is straightforward. The only room for optimization is to use the advanced features of cuFFT in order to save some round trips of the data from and to global memory. cuFFT supports "callbacks", that is a way to attach a function as a prelude or epilogue to a Fourier transform. By implementing the evolution of the chains in these callbacks one can save an additional load and store operation of the whole ensemble. Function pointers however carry an implicit performance penalty: a function call requires a number of registry shuffle operations and local memory loads and stores in order to pass arguments. Worse than that, the number of registers available to a function is very limited. This can cause the nonlinear evolution operator to cause spilling on local memory, as the nonlinear evolution operator includes an exponentiation, which requires a moderate number of registers. Both callback and kernel versions of the evolution operator are implemented in `DNLS/DNLS.cu`, and the user can disable selectively the usage of callbacks with the arguments `--no_linear_callback` and `--no_nonlinear_callback`.

### 5.3 Asynchronous collection of results

As previously mentioned, the implementation of the models and the utilities allows for a fully asynchronous execution of the GPU kernels. Kernel launches are asynchronous with respect to the host, but also the collection of results, that is shard dumps, linear modes energy and entropy calculation are asynchronous operations with respect to each other. The specific details of the kernel launching scheme depend on the specific model. However, the structure is common to all the implemented models.

There is a minimum of four CUDA streams: one for updating the ensemble (`s_move`), one transferring the shard from GPU memory to host memory (`s_dump`), one for calculating the linear modes energy (`s_linenergies`), and one for collecting the results from the host (`s_results`). In the main loop, ensemble updates are enqueued in the (`s_move`) stream. After `gconf.kernel_batching` steps, `s_linenergies` is synchronized with `s_move` and then the kernel or kernels for the calculation of the energies is enqueued. In general the calculation of the energies is composed from a number of kernel invocation, of which only the first one access the working buffer `gres.shard_gpu`. For this reason, `s_move` is synchronized with `s_linenergies` right after the kernels that actually used the buffer, in order to resume the advancement of the simulation as soon as possible. A similar procedure is used to enqueue a copy of the shard from GPU to host memory (`gres.shard_gpu` to `gres.shard_host`) on the `s_dump` stream, if `gconf.dump_interval` steps have passed. Then, `s_results` is synchronized on `s_dump` and `s_linenergies`, and a stream callback is added to it, with the logic to consume the fresh host buffers. Finally, `s_dump` and `s_linenergies` synchronize on `s_results` to avoid filling the host buffers during the new

iteration of the above, in the case that `s_results` takes a long time to complete (which may be the case as dumps are actually written to disk). A simpler model would be to have different callbacks for collecting the results of `s_dump` and `s_linenergies`, and eliminating the `s_results` streams, which has no kernel enqueues but only stream callbacks: however, as previously mentioned (see Section 3.3.4), there is no guarantee in the order of callbacks among different streams, and no guarantee that different callbacks can be called concurrently (e.g. they are independent from one another). Since in practice only one thread executes the callbacks sequentially, having different callbacks in different streams can cause a deadlock, and that is why results are collected from the host in a single callback.

# 6 The CPU-only implementations

The CPU-only implementations are placed in the `cpu-implementation` folder within each model-specific source folder. They employ the same time marching algorithms of the GPU implementation. The structure of the code is also very similar, in the sense that argument parsing, resources setup and the main loop logic is the same. In the place of `loop_control_gpu` (see Section 4.3), the struct `loop_control` (default constructed) is used. This is because there is no need to throttle kernel enqueuing in a CPU-only implementation. The main structural difference between the GPU and CPU implementation is that a single chain is advanced for all `gconf.chain_length` steps at a time. This is done to maximize cache locality.

## 6.1 Automatic vectorization

On modern CPUs it is possible to take advantage of SIMD instructions, that is to operate on multiple data in single operation (vectorization). SIMD programming, if done explicitly, tends to be cumbersome as it is necessary to take into account low-level details in the implementation, such as the width of the SIMD registers, memory alignment requirements, different generations of SIMD instruction sets, etc. The approach of *nlchains* is to let the compiler automatically vectorize the code, using modern features of GCC to hint the compiler on where and how vectorization is possible. In this way, the code remains readable and close to a non-SIMD implementation, yet most of the benefits of vectorization can be reaped. The vectorization is performed within a single chain, hence the implicit SIMD vectors contain consecutive elements of a chain. This is effectively transparent to the programmer, as the code essentially looks like normal scalar code.

SIMD instruction sets have developed over time, yet on an Intel architecture by default the compiler needs to emit instructions that are compatible with the standard x86_64 instruction set that dates back to 2003. Recent GCC versions however allow function multiversioning[3]. The same function can be compiled automatically for multiple target architec-

[3] `https://lwn.net/Articles/691932/`

tures, and the most advanced version compatible with the host CPU is selected at runtime. This feature is available with the function attribute `[[gnu::target_clones("x")]]`, where `"x"` is a list of different instruction sets to compile for. This feature is accessed in *nlchains* through the macro `make_simd_clones(x)`, which expands to `[[gnu::target_clones("x")]]` if the compiler supports it, and in *nlchains* the target architectures string is `"default,avx,avx512f"`. The default (x86_64 with SSE), AVX and AVX512 architectures are chosen because they correspond to the introduction of the XMM, YMM and ZMM registers, respectively 128, 256 and 512-bits long, so they enable SIMD operations on 2, 4, and 8 `double` variables. Other architectures may provide different SIMD capabilities: the user is advised to modify the set of clone targets if *nlchains* is to be recompiled for an architecture different than x86_64.

Automatic vectorization in *nlchains* is relevant in the tight loops that implement the time marching algorithm. Specific compiler optimization flags must be passed in order to activate automatic optimization, and the compiler must be hinted in various ways, described below, that vectorization is safe to perform. All the CPU-only source files and `common/results_linenergies.cpp` are compiled in release mode with `-Ofast` and with `-fopenmp` if OpenMP version 4.x is available. OpenMP is used to activate vectorization in GCC version earlier than 6. All loops that are expected to be vectorized are prepended with the macro `openmp_simd`, which expands to `_Pragma("omp simd")` if available.

SIMD operations are the most efficient when they operate on aligned data, that is data whose address is a multiple of the size of the SIMD vector. The compiler cannot assume that the alignment is correct, hence it creates sub-optimal code even if a loop is vectorized. In order to allocate aligned memory, and to inform the compiler that it is safe to assume that a memory pointer is aligned, the Boost.Align library is used. To cover all existing SIMD widths, the template struct `simd_allocator<T>` can be used as an allocator in STL containers to obtain 64-bytes aligned memory. Then, the pointer `p` to the buffer must be passed to the macro `BOOST_ALIGN_ASSUME_ALIGNED(p, 64)`, in order to instruct the compiler that the pointer is aligned. Here is a sample usage for a vector of `double` values:

```
std::vector<double, simd_allocator<double>> data_buffer(123);
double *__restrict__ data = data_buffer.data();
BOOST_ALIGN_ASSUME_ALIGNED(data, 64);
//operations on data[] here are likely to be vectorized
```

Automatic vectorization works best when the body of the loop that operates on the data does not make calls to functions. If function calls are used, it should be made sure that the function body is inlined: in *nlchains* the function attribute `[[gnu::always_inline]]` is used to hint the compiler that inlining is desired. Some mathematical function cannot however be inlined, such as `exp()` and trigonometric functions. Fortunately, SIMD-ready versions are present from GLIBC version 2.22, and recent versions of GCC can automatically generate calls to them from an automatically vectorized code.

### 6.2 FFTW3 wisdom synchronization

In all the CPU-only models except dDNKG, a Discrete Fourier Transform (DFT) is needed to either advance the integration (DNLS), or to collect results. The implementation is provided by FFTW3. FFTW3 has a mechanism to tune at runtime the algorithm for the DFT to the details of the architecture, and the parameters of the transformation. This mechanism is called "wisdom", as the result of this tuning is stored in a structure called wisdom. This tuning can have different outcomes at different times and on different machines, hence it can cause inconsistencies among MPI processes, but also across different *nlchains* runs in the numerical results. In general, these differences amount to tiny discrepancies (a relative error of $10^{-15}$, that is in the order of the machine epsilon for the type `double`). However, the sophisticated user may want to control the reproducibility of the runs.

The struct `wisdom_sync` is provided to synchronize the FFTW3 wisdom across MPI processes and runs. The constructor takes a string argument, which can be either `"none"`, `"sync"`, and anything else is interpreted as a filename. The meaning of these modes is the following:

- `"none"`: do not attempt to synchronize wisdom
- `"sync"`: calculate the wisdom once in one process and distribute it to all processes
- filename: use the wisdom from this filename (to be generated with the utility `fftw-wisdom` or by copying the output of a run of *nlchains* with the verbose flag `-v`).

The `wisdom_sync` object must be allocated before any FFTW3 plan is created, and a call to the methods `wisdom_sync::gather()` and `wisdom_sync::scatter()` must be the prolog and epilog respectively to all FFTW3 plan allocations. The method `wisdom_sync::add_options(boost::program_options::options_description &options)` can be used to make the wisdom synchronization mode available to the user from command line. In order to force FFTW3 to use the synchronized wisdom, the flag `FFTW_WISDOM_ONLY` must be specified when the plan is created. The `int` field `wisdom_sync::fftw_flags` is set to this flag when needed (that is, when the synchronization mode is not `"none"`). The correct usage of the `wisdom_sync` struct is as following:

```
wisdom_sync wsync("none");  //default mode "none"
parse_cmdline parser("Options for my submodel");
wsync.add_options(parser.options);
parser.run(argc, argv);
//[...]
wsync.gather();
//create FFTW3 plans here
auto plan1 = fftw_plan_dft_1d(...,
                 FFTW_EXHAUSTIVE | wsync.fftw_flags);
```

```
auto plan2 = fftw_plan_dft_1d(...,
                 FFTW_EXHAUSTIVE | wsync.fftw_flags);
//[...]
wsync.scatter();
```