brian2cuda Documentation

Release 1.0.0

brian2cuda authors

Jul 09, 2023

CONTENTS

CHAPTER

ONE

INTRODUCTION

Brian2CUDA is a Python package for simulating spiking neural networks on graphics processing units (GPUs). It is an extension of the spiking neural network simulator Brian2, which allows flexible model definitions in Python. Brian2CUDA uses the code generation system from Brian2 to generate simulation code in C++/CUDA, which is then executed on NVIDIA GPUs.

To use Brian2CUDA, add the following two lines of code to your Brian2 imports. This will execute your simulations on a GPU:

```
from brian2 import *
import brian2cuda
set_device("cuda_standalone")
```

For more details on the code generation process and settings, read the Brian2 standalone device documentation.

1.1 Getting help and reporting bugs

If you need help with Brian2CUDA, please use the Brian2 discourse forum. If you think you found a bug, please report it in our issue tracker on GitHub.

1.2 Citing Brian2CUDA

If you use Brian2CUDA in your work, please cite:

> Alevi, D., Stimberg, M., Sprekeler, H., Obermayer, K., & Augustin, M. (2022). Brian2CUDA: flexible and efficient simulation of spiking neural network models on GPUs. Frontiers in Neuroinformatics. https://doi.org/10.3389/fninf.2022.883700

CHAPTER

TWO

INSTALLATION

2.1 Requirements

- Linux operating system (support for Windows is planned, see https://github.com/brian-team/brian2cuda/issues/225)
- NVIDIA CUDA GPU with compute capability 3.5 or larger
- CUDA Toolkit with `nvcc` compiler
- Python version 3.6 or larger
- Brian2: Each Brian2CUDA version is compatible with a specific Brian2 version. The correct Brian2 version is installed during the Brian2CUDA installation.

We recommend installing Brian2CUDA in a separate Python environment, either using a "virtual environment" or a "conda environment". If you are unfamiliar with that, check out the Brian2 installation instructions.

2.2 Standard install

To install Brian2CUDA with a compatible Brian2 version, use `pip`:

```
python -m pip install brian2cuda
```

2.3 Updating an existing installation

Use the install command together with the `--upgrade` option:

```
python -m pip install --upgrade brian2cuda
```

This will also update the installed Brian2 version if required.

2.4 Development install

When you encounter a problem in BrianCUDA, we will sometimes ask you to install Brian2CUDA's latest development version, which includes changes that were included after its last release.

We regularly upload the latest development version of Brian2CUDA to PyPI's test server. You can install it via:

```
python -m pip install --upgrade --pre -i https://test.pypi.org/simple/ brian2cuda
```

Note that this requires that you already have a compatible Brian2 version and all of its dependencies installed.

If you have `git` installed, you can also install directly from github:

```
python -m pip install git+https://github.com/brian-team/brian2cuda.git
```

If you want to either contribute to Brian's development or regularly test its latest development version, you can directly clone the git repository at github (https://github.com/brian-team/brian2cuda) and then run `pip install -e /path/to/brian2cuda`, to install Brian2CUDA in "development mode". As long as the compatible Brian2 version doesn't change, updating the git repository is in general enough to keep up with changes in the code, i.e. it is not necessary to install it again. If the compatible Brian2 versions changes though, you need to manually update Brian2.

2.5 Testing your installation

Brian2CUDA tries to automatically detect your CUDA toolkit installation and choose the newest GPU on your system to run simulations. To test if this detection and your installation were successful, you can run this test simulation:

```
import brian2cuda
brian2cuda.example_run()
```

If the automatic CUDA and GPU detection fails or you want to manually change it, read *Configuring the CUDA backend*.

2.6 Running the Brian2CUDA test suit

If you have the pytest testing utility installed, you can run Brian2CUDA's test suite:

```
import brian2cuda
brian2cuda.test()
```

This runs all standalone-comatible tests from the Brian2 test suite and additional Brian2CUDA tests (see the Brian2 developer documentation on testing for more details) and can take 1-2 hours, depending on your hardware. The test suite should end with "OK", showing a number of skipped tests but no errors or failures. If you want to run individual tests instead of the entire test suite (e.g. during development), check out the Brian2CUDA tools directory.

CHAPTER

THREE

CONFIGURING THE CUDA BACKEND

Brian2CUDA tries to detect your CUDA installation and uses the GPU with highest compute capability by default. To query information about available GPUs, `nvidia-smi` (installed alongside NVIDIA display drivers) is used. For older driver versions (< `510.39.01`), `nvidia-smi` doesn't support querying the GPU compute capabilities and some additional setup might be required.

This section explains how you can manually set which CUDA installation or GPU to use, how to cross-compile Brian2CUDA projects on systems without GPU access (e.g. during remote development) and what to do when the compute capability detection fails.

3.1 Manually specifying the CUDA installation

If you installed the CUDA toolkit in a non-standard location or if you have a system with multiple CUDA installations, you may need to manually specify the installation directory.

Brian2CUDA tries to detect your CUDA installation in the following order:

1. Use Brian2CUDA preference *devices.cuda_standalone.cuda_backend.cuda_path*
2. Use `CUDA_PATH` environment variable
3. Use location of `nvcc` to detect CUDA installation folder (needs `nvcc` binary in `PATH`)
4. Use standard location `/usr/local/cuda`
5. Use standard location `/opt/cuda`

If you set the path manually via the 1. or 2. option, specify the parent path to the `nvcc` binary (e.g. `/usr/local/cuda` if `nvcc` is in `/usr/local/cuda/bin/nvcc`).

Depending on your system configuration, you may also need to set the `LD_LIBRARY_PATH` environment variable to `$CUDA_PATH/lib64`.

3.2 Manually selecting a GPU to use

On systems with multiple GPUs, Brian2CUDA uses the first GPU with highest compute capability as returned by `nvidia-smi`. If you want to manually choose a GPU you can do so via Brian2CUDA preference *devices.cuda_standalone.cuda_backend.gpu_id*.

Note: You can limit the visibility of NVIDIA GPUs by setting the environment variable `CUDA_VISIBLE_DEVICES`. This also limits the GPUs visible to Brian2CUDA. That means Brian2CUDA's *devices.cuda_standalone.cuda_backend.gpu_id* preference will index only those GPUs that are visible. E.g. if you run a Brian2CUDA script with `prefs.devices.cuda_standalone.cuda_backend.gpu_id = 0` on a system with two GPUs via `CUDA_VISIBLE_DEVICES=1 python your-brian2cuda-script.py`, the simulation would run on the second GPU (with ID 1, visible to Brian2CUDA as ID `0`).

3.3 Cross-compiling on systems without GPU access

On systems without GPU, Brian2CUDA will fail before code generation by default (since it tries to detect the compute capability of the available GPUs and the CUDA runtime version). If you want to compile your code on a system without GPUs, you can disable automatic GPU detection and manually set the compute capability and runtime version. To do so, set the following preferences:

```
prefs.devices.cuda_standalone.cuda_backend.detect_gpus = False
prefs.devices.cuda_standalone.cuda_backend.compute_capability = <compute_capability>
prefs.devices.cuda_standalone.cuda_backend.runtime_version = <runtime_version>
```

See *devices.cuda_standalone.cuda_backend.detect_gpus*, *devices.cuda_standalone.cuda_backend.compute_capability* and *devices.cuda_standalone.cuda_backend.cuda_runtime_version*.

3.4 Detecting GPU compute capability on systems with outdated NVIDIA drivers

We use `nvidia-smi` to query the compute capability of GPUs during automatic GPU selection. On older driver versions (`< 510.39.01`, these are driver versions shipped with CUDA toolkit `< 11.6`), this was not supported. For those versions, we use the `deviceQuery` tool from the CUDA samples, which is by default installed with the CUDA Toolkit under `extras/demo_suite/deviceQuery` in the CUDA installation directory. For some custom CUDA installations, the CUDA samples are not included, in which case Brian2CUDA's GPU detection fails. In that case, you have three options. Do one of the following:

1. Update your NVIDIA driver
2. Download the CUDA samples to a folder of your choice and compile `deviceQuery` manually:

   ```
   git clone https://github.com/NVIDIA/cuda-samples.git
   cd cuda-samples/Samples/1_Utilities/deviceQuery
   make
   # Run deviceQuery to test it
   ./deviceQuery
   ```

   Now set Brian2CUDA preference *devices.cuda_standalone.cuda_backend.device_query_path* to point to your `deviceQuery` binary.

3. Disable automatic GPU detection and manually provide the GPU ID and compute capability (you can find the compute capability of your GPU on https://developer.nvidia.com/cuda-gpus):

```
prefs.devices.cuda_standalone.cuda_backend.detect_gpus = False
prefs.devices.cuda_standalone.cuda_backend.compute_capability = <compute_capability>
```

See *devices.cuda_standalone.cuda_backend.detect_gpus* and *devices.cuda_standalone.cuda_backend.compute_capability*.

CHAPTER

FOUR

KNOWN ISSUES

In addition to the issues noted below, you can refer to our bug tracker on GitHub.

List of known issues:

4.1 Known issues when using multiple `run` calls

4.1.1 Changing the integration time step of `Synapses` with delays between `run` calls

Changing the integration time step of `Synapses` objects with transmission delays between successive `run` calls currently leads to the loss of spikes. This is the case for spikes that are queued for effect application but haven't been applied yet when the first `run` call terminates. See Brian2CUDA issue #136 for progress on this issue.

4.1.2 Changing delays between `run` calls

Changing the delay of `Synapses` objects between `run` calls currently leads to the loss of spikes. This is the case when changing homogenenous delays or when switching between homogeneous and heterogeneous delays (e.g. `Synapses.delay = 'j*ms'` before the first `run` call and `Synapses.delay = '1*ms'` after the first `run` call). Changing heterogenenous delays between `run` calls is not effected from this bug and should work as expected (e.g. from `synapses.delay = 'j*ms'` to `synapses.delay = '2*j*ms'`). See Brian2CUDA issue #302 for progress on this issue.

4.2 Using a different integration time for Synapses and its source NeuronGroup

There is currently a bug when using `Synapses` with homogeneous delays and choosing a different integration time step (`dt`) for any of its `SynapticPathway` and its associated source `NeuronGroup`. This bug does not occur when the delays are heterogenenous or when only the target `NeuronGroup` has a different clock. See Brian2CUDA issue #222 for progress on the issue. Any of the following examples has this bug:

```
from brian2 import *

group_different_dt = NeuronGroup(1, 'v:1', threshold='True', dt=2*defaultclock.dt)
group_same_dt = NeuronGroup(1, 'v:1', threshold='True', dt=defaultclock.dt)

# Bug: Source of pre->post synaptic pathway uses different dt than synapses
#      and synapses have homogeneous delays
synapses = Synapses(
    group_different_dt,
    group_same_dt,
    on_pre='v+=1',
    delay=1*ms,
    dt=defaultclock.dt
)

# No bug: Synapses have no delays
synapses = Synapses(
    group_different_dt,
    group_same_dt,
    on_pre='v+=1',
    dt=defaultclock.dt
)

# No bug: Synapses have heterogeneous delays
synapses = Synapses(
    group_different_dt,
    group_same_dt,
    on_pre='v+=1',
    dt=defaultclock.dt
)
synapses.delay = 'j*ms'

# No bug: Source of pre->post synaptic pathway uses the same dt as synapses
synapses = Synapses(
    group_same_dt,
    group_different_dt,
    on_post='v+=1',
    delay=1*ms,
    dt=defaultclock.dt
)
```

4.3 `SpikeMonitor` and `EventMonitor` data is not sorted by indices

In all Brian2 devices, `SpikeMonitor` and `EventMonitor` data is first sorted by time and then by neuron index. In Brian2CUDA, the data is only sorted by time but not always by index given a fixed time point. See Brian2CUDA issue #46 for progress on this issue.

4.4 Single precision mode fails when using variable names with double digit and dot or scientific notations in name

In single precision mode (set via `prefs.core.default_float_dtype`), Brian2CUDA replaces floating point literals like `.2`, `1.` or `.4` in generated code with single precision versions `1.2f`, `1.f` and `.4f`. Under some circumstances, the search/replace algorithm fails and performs a wrong string replacement. This is the case e.g. for variable name with double digit and a dot in their name, such as `variable12.attribute` or when variable names have a substring that can be interpreted as a scientific number, e.g. `variable28e2`, which has `28e2` as substring. If such a wrong replacement occurs, compilation typically fails due to not declared variables. See Brian2CUDA issue #254 for progress on the issue.

CHAPTER

FIVE

BRIAN2CUDA SPECIFIC PREFERENCES

For information on the Brian2 preference system, read Brian2 preference documentation. The following Brian2CUDA preferences are used in the same way.

5.1 List of preferences

Brian2CUDA preferences

devices.cuda_standalone.SM_multiplier = 1
The number of blocks per SM. By default, this value is set to 1.

devices.cuda_standalone.bundle_threads_warp_multiple = False
Whether to round the number of threads used per synapse bundle during effect application (see *devices.cuda_standalone.threads_per_synapse_bundle*) to a multiple of the warp size. Round to next multiple if preference is `'up'`, round to previous multiple if `'low'` and don't round at all if `False` (default). If rounding down results in `0` threads, `1` thread is used instead.

devices.cuda_standalone.calc_occupancy = True
Wether or not to use cuda occupancy api to choose num_threads and num_blocks.

devices.cuda_standalone.default_functions_integral_convertion = float64
The floating point precision to which integral types will be converted when passed as arguments to default functions that have no integral type overload in device code (sin, cos, tan, sinh, cosh, tanh, exp, log, log10, sqrt, ceil, floor, arcsin, arccos, arctan)." NOTE: Convertion from 32bit and 64bit integral types to single precision (32bit) floating-point types is not type safe. And convertion from 64bit integral types to double precision (64bit) floating-point types neither. In those cases the closest higher or lower (implementation defined) representable value will be selected.

devices.cuda_standalone.extra_threshold_kernel = True
Wether or not to use a extra threshold kernel for resetting.

devices.cuda_standalone.launch_bounds = False
Wether or not to use `__launch_bounds__` to optimise register usage in kernels.

devices.cuda_standalone.no_post_references = False
Set this preference if you don't need access to `j` in any synaptic code string and no Synapses object applies effects to postsynaptic variables. This preference is for memory optimization until unnecassary device memory allocations in synapse creation are fixed, it is only relevant if your network uses close to all memory.

devices.cuda_standalone.no_pre_references = False
Set this preference if you don't need access to `i` in any synaptic code string and no Synapses object applies effects to presynaptic variables. This preference is for memory optimization until unnecassary device memory allocations in synapse creation are fixed, it is only relevant if your network uses close to all memory.

devices.cuda_standalone.parallel_blocks = 1
The total number of parallel blocks to use. If `None`, the number of parallel blocks equals the number streaming multiprocessors on the GPU.

devices.cuda_standalone.profile_statemonitor_copy_to_host = None
Profile the final device to host copy of StateMonitor data. This preference is used for benchmarking and assumes that there is only one active StateMonitor in the network. The parameter of this preference is the recorded variable for which the device to host copy is recorded (e.g. 'v').

devices.cuda_standalone.push_synapse_bundles = True
If True, synaptic events are propagated by pushing bundles of synapse IDs with same delays into the corresponding delay queue. If False, each synapse of a spiking neuron is pushed in the corresponding queue individually. For very small bundle sizes (number of synapses with same delay, connected to a single neuron), pushing single Synapses can be faster. This option only has effect for `Synapses` objects with heterogenous delays.

devices.cuda_standalone.random_number_generator_ordering = False
The ordering parameter (str) used to choose how the results of cuRAND random number generation are ordered in global memory. See cuRAND documentation for more details on generator types and orderings.

devices.cuda_standalone.random_number_generator_type = 'CURAND_RNG_PSEUDO_DEFAULT'
Generator type (str) that cuRAND uses for random number generation. Setting the generator type automatically resets the generator ordering (prefs.devices.cuda_standalone.random_number_generator_ordering) to its default value. See cuRAND documentation for more details on generator types and orderings.

devices.cuda_standalone.syn_launch_bounds = False
Wether or not to use `__launch_bounds__` in synapses and synapses_push to optimise register usage in kernels.

devices.cuda_standalone.threads_per_synapse_bundle = '{max}'
The number of threads used per synapses bundle during effect application. This has to be a string, which can be passed to Python's `eval` function. The string can can use `{mean}`, `{std}`, `{max}` and `{min}` expressions, which refer to the statistics across all bundles, and the function 'ceil'. The result of this expression will be converted to the next lower `int` (e.g. `1.9` will be cast to `1.0`). Examples: `'{mean} + 2 * {std}'` will use the mean bunde size + 2 times the standard deviation over bundle sizes and round it to the next lower integer. If you want to round up instead, use `'ceil({mean} + 2 * {std})'`.

devices.cuda_standalone.use_atomics = True
Weather to try to use atomic operations for synaptic effect application. Since this avoids race conditions, effect application can be parallelised.

Preferences for the CUDA backend in Brian2CUDA

devices.cuda_standalone.cuda_backend.compute_capability = None
Manually set the compute capability for which CUDA code will be compiled. Has to be a float (e.g. `6.1`) or None. If None, compute capability is chosen depending on GPU in use.

devices.cuda_standalone.cuda_backend.cuda_path = None
The path to the CUDA installation. If set, this preferences takes precedence over environment variable `CUDA_PATH`.

devices.cuda_standalone.cuda_backend.cuda_runtime_version = None
The CUDA runtime version.

devices.cuda_standalone.cuda_backend.detect_cuda = True
Whether to try to detect CUDA installation paths and version. Disable this if you want to generae CUDA standalone code on a system without CUDA installed.

devices.cuda_standalone.cuda_backend.detect_gpus = True
Whether to detect names and compute capabilities of all available GPUs. This needs access to `nvidia-smi` and `deviceQuery` binaries.

devices.cuda_standalone.cuda_backend.device_query_path = None
Path to CUDA's deviceQuery binary. Used to detect a GPUs compute capability

devices.cuda_standalone.cuda_backend.extra_compile_args_nvcc = ['-w', '-use_fast_math']
Extra compile arguments (a list of strings) to pass to the nvcc compiler.

devices.cuda_standalone.cuda_backend.gpu_heap_size = 128
Size of the heap (in MB) used by malloc() and free() device system calls, which are used in the `cudaVector` implementation. `cudaVectors` are used to dynamically allocate device memory for `SpikeMonitors` and the synapse queues in the `CudaSpikeQueue` implementation for networks with heterogeneously distributed delays.

devices.cuda_standalone.cuda_backend.gpu_id = None
The ID of the GPU that should be used for code execution. Default value is `None`, in which case the GPU with the highest compute capability and lowest ID is used.

If environment variable `CUDA_VISIBLE_DEVICES` is set, this preference will be interpreted as ID from the visible devices (e.g. with `CUDA_VISIBLE_DEVICES=2` and `gpu_id=0` preference, the GPU 2 will be used).

CHAPTER

SIX

PERFORMANCE CONSIDERATIONS

Check out our Brian2CUDA paper for performance benchmarks and discussions.

If you have performance questions or want to share your experience with Brian2CUDA performance, feel free to post on the Brian2 discourse forum.

CHAPTER

SEVEN

BRIAN2CUDA PACKAGE

Package implementing the CUDA "standalone" `Device` and `CodeObject`.

Functions

example_run([device_name, directory])	Run a simple example simulation to test whether Brian2CUDA is correctly set up.

7.1 example_run function

(*Shortest import*: `from brian2cuda.__init__ import example_run`)

`brian2cuda.__init__.`**`example_run`**(*device_name='cuda_standalone'*, *directory=None*, ***build_options*)

Run a simple example simulation to test whether Brian2CUDA is correctly set up.

Parameters

device_name : str

What device to use (default: "cuda_standalone").

directory : str ,optional

The output directory to write the project to, any existing files will be overwritten. If the given directory name is `None` (default for this example run), then a temporary directory will be used.

build_options : dict, optional

Additional options that will be forwarded to the `device.build` call,

7.2 `_version` module

7.3 `binomial` module

CUDA implementation of `BinomialFunction`

7.4 codeobject module

Module implementing the CUDA "standalone" `CodeObject`. Brian2CUDA implements two different code objects. *CUDAStandaloneCodeObject* is the standard implementation, which does not use atomic operations but serialized synaptic effect application if race conditions are possible. *CUDAStandaloneAtomicsCodeObject* uses atomic operations which allows parallel effect applications even when race conditions are possible.

Exported members: *CUDAStandaloneCodeObject*, *CUDAStandaloneAtomicsCodeObject*

Classes

CUDAStandaloneAtomicsCodeObject(*args, **kw)	CUDA standalone code object which uses atomic operations for parallel execution

7.4.1 CUDAStandaloneAtomicsCodeObject class

(*Shortest import*: `from brian2cuda.codeobject import CUDAStandaloneAtomicsCodeObject`)

class brian2cuda.codeobject.**CUDAStandaloneAtomicsCodeObject**(**args*, ***kw*)

CUDA standalone code object which uses atomic operations for parallel execution

The `code` should be a `MultiTemplate` object with two macros defined, `main` (for the main loop code) and `support_code` for any support code (e.g. function definitions).

CUDAStandaloneCodeObject(*args, **kw)	CUDA standalone code object

7.4.2 CUDAStandaloneCodeObject class

(*Shortest import*: `from brian2cuda.codeobject import CUDAStandaloneCodeObject`)

class brian2cuda.codeobject.**CUDAStandaloneCodeObject**(**args*, ***kw*)

CUDA standalone code object

The `code` should be a `MultiTemplate` object with two macros defined, `main` (for the main loop code) and `support_code` for any support code (e.g. function definitions).

Methods

__call__(**kwds)	Call self as a function.
compile_block(block)	
run_block(block)	

Details

__call__(***kwds*)
Call self as a function.

compile_block(*block*)

run_block(*block*)

7.5 cuda_generator module

Exported members: *CUDACodeGenerator*, *CUDAAtomicsCodeGenerator*, c_data_type

Classes

CUDAAtomicsCodeGenerator(*args, **kwds)	

7.5.1 CUDAAtomicsCodeGenerator class

(*Shortest import*: from brian2cuda.cuda_generator import CUDAAtomicsCodeGenerator)

class brian2cuda.cuda_generator.**CUDAAtomicsCodeGenerator**(**args*, ***kwds*)

CUDACodeGenerator(*args, **kwds)	C++ language with CUDA library

7.5.2 CUDACodeGenerator class

(*Shortest import*: from brian2cuda.cuda_generator import CUDACodeGenerator)

class brian2cuda.cuda_generator.**CUDACodeGenerator**(**args*, ***kwds*)
C++ language with CUDA library

CUDA code templates should provide Jinja2 macros with the following names:

main
The main loop.

support_code
The support code (function definitions, etc.), compiled in a separate file.

For user-defined functions, there are two keys to provide:

support_code
The function definition which will be added to the support code.

hashdefine_code
The #define code added to the main loop.

See TimedArray for an example of these keys.

Attributes

flush_denormals	
restrict	
universal_support_code	

Methods

atomics_parallelisation(statement, ...)	
conditional_write(line, statement, ...)	
denormals_to_zero_code()	
determine_keywords()	A dictionary of values that is made available to the templated.
get_array_name(var[, access_data, prefix])	Return a globally unique name for `var()`.
parallelise_code(statements)	
translate_expression(expr)	Translate the given expression string into a string in the target language, returns a string.
translate_one_statement_sequence(statements)	
translate_statement(statement)	Translate a single line `Statement` into the target language, returns a string.
translate_to_declarations(read, write, indices)	
translate_to_read_arrays(read, write, indices)	
translate_to_statements(statements, ...)	
translate_to_write_arrays(write)	

Details

flush_denormals

restrict

universal_support_code = None

atomics_parallelisation(*statement*, *conditional_write_vars*, *used_variables*)

conditional_write(*line*, *statement*, *conditional_write_vars*)

denormals_to_zero_code()

determine_keywords()

A dictionary of values that is made available to the templated. This is used for example by the CPPCodeGenerator to set up all the supporting code

static get_array_name(*var*, *access_data=True*, *prefix=None*)

Return a globally unique name for var(). See CUDAStandaloneDevice.get_array_name for parameters.

Here, prefix defaults to '_ptr' when access_data=True.

prefix='_ptr' is used since the CUDACodeGenerator generates the scalar_code and vector_code snippets.

parallelise_code(*statements*)

translate_expression(*expr*)

Translate the given expression string into a string in the target language, returns a string.

translate_one_statement_sequence(*statements*, *scalar=False*)

translate_statement(*statement*)

Translate a single line Statement into the target language, returns a string.

translate_to_declarations(*read*, *write*, *indices*)

translate_to_read_arrays(*read*, *write*, *indices*)

translate_to_statements(*statements*, *conditional_write_vars*)

translate_to_write_arrays(*write*)

ParallelisationError

7.5.3 ParallelisationError class

(*Shortest import*: from brian2cuda.cuda_generator import ParallelisationError)

class brian2cuda.cuda_generator.**ParallelisationError**

7.6 cuda_prefs module

Preferences that relate to the brian2cuda interface.

Functions

validate_bundle_size_expression(string)

7.6.1 validate_bundle_size_expression function

(*Shortest import*: `from brian2cuda.cuda_prefs import validate_bundle_size_expression`)

`brian2cuda.cuda_prefs.`**`validate_bundle_size_expression`**`(`*`string`*`)`

7.7 device module

Module implementing the CUDA "standalone" device.

Classes

CUDAStandaloneDevice()	The `Device` used for CUDA standalone simulations.

7.7.1 CUDAStandaloneDevice class

(*Shortest import*: `from brian2cuda.device import CUDAStandaloneDevice`)

`class` `brian2cuda.device.`**`CUDAStandaloneDevice`**

The `Device` used for CUDA standalone simulations.

Methods

build([directory, compile, run, debug, ...])	Build the project
check_openmp_compatible(nb_threads)	
code_object(owner, name, abstract_code, ...)	
code_object_class([codeobj_class, fallback_pref])	Return `CodeObject` class (either `CUDAStandaloneCodeObject` class or input)
copy_source_files(writer, directory)	
fill_with_array(var, *args, **kwargs)	Fill an array with the values given in another array.
generate_codeobj_source(writer)	
generate_main_source(writer)	
generate_makefile(writer, cpp_compiler, ...)	
generate_network_source(writer)	
generate_objects_source(writer, ...)	
generate_rand_source(writer)	
generate_run_source(writer)	
generate_synapses_classes_source(writer)	
get_array_name(var[, access_data, prefix])	Return a globally unique name for `var()`.
get_array_read_write(abstract_code, variables)	
network_restore(net, *args, **kwds)	
network_run(net, duration[, report, ...])	
network_store(net, *args, **kwds)	
variableview_set_with_index_array(...)	

Details

build(*directory='output'*, *compile=True*, *run=True*, *debug=False*, *clean=False*, *with_output=True*, *disable_asserts=False*, *additional_source_files=None*, *run_args=None*, *direct_call=True*, ***kwds*)

Build the project

TODO: more details

Parameters

directory : str, optional

The output directory to write the project to, any existing files will be overwritten. If the given directory name is `None`, then a temporary directory will be used (used in the test

suite to avoid problems when running several tests in parallel). Defaults to `'output'`.

compile : bool, optional

Whether or not to attempt to compile the project. Defaults to `True`.

run : bool, optional

Whether or not to attempt to run the built project if it successfully builds. Defaults to `True`.

debug : bool, optional

Whether to compile in debug mode. Defaults to `False`.

with_output : bool, optional

Whether or not to show the `stdout` of the built program when run. Output will be shown in case of compilation or runtime error. Defaults to `True`.

clean : bool, optional

Whether or not to clean the project before building. Defaults to `False`.

additional_source_files : list of str, optional

A list of additional `.cu` files to include in the build.

direct_call : bool, optional

Whether this function was called directly. Is used internally to distinguish an automatic build due to the `build_on_run` option from a manual `device.build` call.

`check_openmp_compatible`(*nb_threads*)

`code_object`(*owner*, *name*, *abstract_code*, *variables*, *template_name*, *variable_indices*, *codeobj_class=None*, *template_kwds=None*, *override_conditional_write=None*, *compiler_kwds=None*)

`code_object_class`(*codeobj_class=None*, *fallback_pref=None*)

Return `CodeObject` class (either `CUDAStandaloneCodeObject` class or input)

Parameters

codeobj_class : a `CodeObject` class, optional

If this is keyword is set to None or no arguments are given, this method will return the default (`CUDAStandaloneCodeObject` class).

fallback_pref : str, optional

For the cuda_standalone device this option is ignored.

Returns

codeobj_class : class

The `CodeObject` class that should be used

`copy_source_files`(*writer*, *directory*)

`fill_with_array`(*var*, **args*, ***kwargs*)

Fill an array with the values given in another array.

Parameters

var : `ArrayVariable`

The array to fill.

arr : ndarray

The array values that should be copied to `var()`.

generate_codeobj_source(*writer*)

generate_main_source(*writer*)

generate_makefile(*writer*, *cpp_compiler*, *cpp_compiler_flags*, *cpp_linker_flags*, *debug*, *disable_asserts*)

generate_network_source(*writer*)

generate_objects_source(*writer*, *arange_arrays*, *synapses*, *static_array_specs*, *networks*)

generate_rand_source(*writer*)

generate_run_source(*writer*)

generate_synapses_classes_source(*writer*)

get_array_name(*var*, *access_data=True*, *prefix=None*)

Return a globally unique name for `var()`.

Parameters

access_data : bool, optional

For `DynamicArrayVariable` objects, specifying `True` here means the name for the underlying data is returned. If specifying `False`, the name of object itself is returned (e.g. to allow resizing).

prefix: {'_ptr', 'dev', 'd'}, optional :

Prefix for array name. Host pointers to device memory are prefixed with `dev`, device pointers to device memory are prefixed with `d` and pointers used in `scalar_code` and `vector_code` are prefixed with `_ptr` (independent of whether they are used in host or device code). The `_ptr` variables are declared as parameters in the kernel definition (KERNEL_PARAMETERS).

get_array_read_write(*abstract_code*, *variables*)

network_restore(*net*, **args*, ***kwds*)

network_run(*net*, *duration*, *report=None*, *report_period=10. * second*, *namespace=None*, *profile=False*, *level=0*, ***kwds*)

network_store(*net*, **args*, ***kwds*)

variableview_set_with_index_array(*variableview*, **args*, ***kwargs*)

CUDAWriter(project_dir)

Methods

7.7.2 CUDAWriter class

(*Shortest import*: `from brian2cuda.device import CUDAWriter`)

`class brian2cuda.device.CUDAWriter(project_dir)`

Methods

write(filename, contents)	

Details

`write(filename, contents)`

Functions

prepare_codeobj_code_for_rng(codeobj)	Prepare a CodeObject for random number generation (RNG).

7.7.3 prepare_codeobj_code_for_rng function

(*Shortest import*: `from brian2cuda.device import prepare_codeobj_code_for_rng`)

`brian2cuda.device.prepare_codeobj_code_for_rng(codeobj)`

Prepare a CodeObject for random number generation (RNG).

There are two different ways that random numbers are generated in CUDA:

1) Using a buffer system which is refilled from host code in regular intervals using the cuRAND host API. This is used for `rand()`, `randn()` and `poisson(lambda)` when `lambda` is a scalar. The buffer system is implemented in the `rand.cu` template.
2) Using on-the-fly RNG from device code using the cuRAND device API. This is used for `binomial` and `poisson(lambda)` when `lambda` is a vectorized variable (different across neurons/synapses). This needs initilization of cuRAND random states, which is also happening in the `rand.cu` template.

This function counts the number of `rand()`, `randn()` and `poisson(<lambda>)` appearances in `codeobj.code.cu_file` and stores this number in the `codeobj.rng_calls` dictionary (with keys `"rand"`, `"randn"` and `"poisson_<idx>"` ,one `<idx>` per `poisson()` call). If the codeobject uses the curand device API for RNG (for binomial of poisson with variable lambda), this function sets `codeobj.needs_curand_states = True`.

For RNG functions that use the buffer system, this function replaces the function arguments in the generated code such that a pointer to the random number buffer and the correct index are passed as function arguments.

For RNG functions that use on-the-fly RNG, the functions are not replaced since no pointer or index has to be passed.

For the `poisson` RNG, the RNG type depends on the `lambda` value. For scalar `lambda`, we use the buffer system which is most efficient and most robust in the RNG. For vectorized `lambda` values, the host API is inefficient and instead the simple device API is used, which is the most efficient but least robust. For the two RNG systems to work, we overload the CUDA implementation of `_poisson` with `_poisson(double _lambda, ...)` and `_poisson(unsigned int* _poisson_buffer, ...)`. When the buffer system is used, we replace the `_poisson(<lambda>, ...)` calls with `_poisson(<int_pointer>, ...)` calls.

For `poisson` with `lambda <= 0`, the returned random numbers are always `0`. This function makes sure that the `lambda` is replaced with a double literal for our overloaded `_poisson` function to work correctly.

Parameters

codeobj: CodeObjects :

Codeobject with generated CUDA code in `codeobj.code.cu_file`.

Objects

cuda_standalone_device	The `Device` used for CUDA standalone simulations.

7.7.4 cuda_standalone_device object

(*Shortest import*: `from brian2cuda.device import cuda_standalone_device`)

`brian2cuda.device.`**`cuda_standalone_device = <brian2cuda.device.CUDAStandaloneDevice object>`**

The `Device` used for CUDA standalone simulations.

7.8 `timedarray` module

CUDA implementation of `TimedArray`

7.9 Subpackages

7.9.1 utils package

Utility functions for Brian2CUDA

`gputools` module

Tools to get information about available GPUs.

Functions

get_available_gpus()	Return list of names of available GPUs, sorted by GPU ID as reported in `nvidia-smi`

get_available_gpus function

(*Shortest import*: `from brian2cuda.utils.gputools import get_available_gpus`)

`brian2cuda.utils.gputools.`**`get_available_gpus`**`()`

Return list of names of available GPUs, sorted by GPU ID as reported in `nvidia-smi`

get_best_gpu()	Get the "best" GPU available.

get_best_gpu function

(*Shortest import*: `from brian2cuda.utils.gputools import get_best_gpu`)

`brian2cuda.utils.gputools.`**`get_best_gpu`**`()`

Get the "best" GPU available. This currently chooses the GPU with highest compute capability and lowest GPU ID (as reported by `nvidia-smi`)

get_compute_capability(gpu_id)	Get compute capability of GPU with ID `gpu_id`.

get_compute_capability function

(*Shortest import*: `from brian2cuda.utils.gputools import get_compute_capability`)

`brian2cuda.utils.gputools.`**`get_compute_capability`**`(`*`gpu_id`*`)`

Get compute capability of GPU with ID `gpu_id`. Returns a float (e.g. `6.1`).

get_cuda_installation()	Return new dictionary of cuda installation variables

get_cuda_installation function

(*Shortest import*: `from brian2cuda.utils.gputools import get_cuda_installation`)

`brian2cuda.utils.gputools.`**`get_cuda_installation`**`()`

Return new dictionary of cuda installation variables

get_cuda_path()	Detect the path to the CUDA installation (e.g.

get_cuda_path function

(*Shortest import*: `from brian2cuda.utils.gputools import get_cuda_path`)

`brian2cuda.utils.gputools.`**`get_cuda_path`**`()`

Detect the path to the CUDA installation (e.g. '/usr/local/cuda'). This takes into account user defined environmental variable `CUDA_PATH` and preference `prefs.devices.cuda_standalone.cuda_backend.cuda_path`.

get_cuda_runtime_version()	Return CUDA runtime version (as float, e.g.

get_cuda_runtime_version function

(*Shortest import*: `from brian2cuda.utils.gputools import get_cuda_runtime_version`)

brian2cuda.utils.gputools.**get_cuda_runtime_version**()

Return CUDA runtime version (as float, e.g. `11.2`)

get_gpu_selection()	Return dictionary of selected gpu variable

get_gpu_selection function

(*Shortest import*: `from brian2cuda.utils.gputools import get_gpu_selection`)

brian2cuda.utils.gputools.**get_gpu_selection**()

Return dictionary of selected gpu variable

get_nvcc_path()	Return the path to the `nvcc` binary.

get_nvcc_path function

(*Shortest import*: `from brian2cuda.utils.gputools import get_nvcc_path`)

brian2cuda.utils.gputools.**get_nvcc_path**()

Return the path to the `nvcc` binary.

reset_cuda_installation()	Reset detected CUDA installation.

reset_cuda_installation function

(*Shortest import*: `from brian2cuda.utils.gputools import reset_cuda_installation`)

brian2cuda.utils.gputools.**reset_cuda_installation**()

Reset detected CUDA installation. This will detect the CUDA installation again when it is needed.

reset_gpu_selection()	Reset selected GPU.

reset_gpu_selection function

(*Shortest import*: `from brian2cuda.utils.gputools import reset_gpu_selection`)

brian2cuda.utils.gputools.**reset_gpu_selection**()

Reset selected GPU. This will select a new GPU the next time it is needed.

restore_cuda_installation(cuda_installation)	Set global cuda installation dictionary to `cuda_installation`

restore_cuda_installation function

(*Shortest import*: `from brian2cuda.utils.gputools import restore_cuda_installation`)

brian2cuda.utils.gputools.**restore_cuda_installation**(*cuda_installation*)

Set global cuda installation dictionary to `cuda_installation`

restore_gpu_selection(gpu_selection)	Set global gpu selection dictionary to `gpu_selection`

restore_gpu_selection function

(*Shortest import*: `from brian2cuda.utils.gputools import restore_gpu_selection`)

brian2cuda.utils.gputools.**restore_gpu_selection**(*gpu_selection*)

Set global gpu selection dictionary to `gpu_selection`

select_gpu()	Select GPU for simulation, based on user preference `prefs.devices.cuda_standalone.cuda_backend.gpu_id` or (if not provided) pick the GPU with highest compute capability.

select_gpu function

(*Shortest import*: `from brian2cuda.utils.gputools import select_gpu`)

brian2cuda.utils.gputools.**select_gpu**()

Select GPU for simulation, based on user preference `prefs.devices.cuda_standalone.cuda_backend.gpu_id` or (if not provided) pick the GPU with highest compute capability. Returns tuple of (gpu_id, compute_capability) of type (int, float).

logger module

Brian2CUDA's logging system extensions

Exported members: suppress_brian2_logs

Functions

suppress_brian2_logs()	Suppress all logs coming from brian2.

suppress_brian2_logs function

(*Shortest import*: `from brian2cuda.utils import suppress_brian2_logs`)

brian2cuda.utils.logger.**suppress_brian2_logs**()

Suppress all logs coming from brian2.

stringtools module

Brian2CUDA regex functions.

Functions

append_f(match)	Append `'f'` to the string in `match` if it doesn't end with `'f'`.

append_f function

(*Shortest import*: `from brian2cuda.utils import append_f`)

brian2cuda.utils.stringtools.**append_f**(*match*)

Append `'f'` to the string in `match` if it doesn't end with `'f'`. Used in `replace_floating_point_literals`.

Parameters

match : re.MatchObject

The return type of e.g. `re.match` or `re.search`.

Returns

str :

The string returned from `match.group()` if it end with `'f'`, else the string with `'f'` appended.

replace_floating_point_literals(code)	Replace double-precision floating-point literals in `code` by single-precision literals.

replace_floating_point_literals function

(*Shortest import*: `from brian2cuda.utils import replace_floating_point_literals`)

`brian2cuda.utils.stringtools.`**`replace_floating_point_literals`**`(`*`code`*`)`

Replace double-precision floating-point literals in `code` by single-precision literals.

Parameters

code : str

A string to replace the literals in. C++ syntax is assumed, s.t. e.g. `a1.b` would not be replaced.

Returns

str :

A copy of `code`, with double-precision floating point literals replaced by single-precision flaoting-point literals (with an `f` appended).

Examples

```
>>> replace_floating_point_literals('1.|.2=3.14:5e6>.5E-2<7.e-8==a1.b')
1.f|.2f=3.14f:5e6f>.5E-2f<7.e-8f==a1.b
```

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

U

V

W