DaCe

Release 0.10.0a

Feb 10, 2022

Contents

DaCe: Data-Centric Parallel Programming framework.

CHAPTER 1

dace

1.1 dace package

1.1.1 Subpackages

dace.codegen package

Subpackages

dace.codegen.instrumentation package

Submodules

dace.codegen.instrumentation.gpu_events module

class dace.codegen.instrumentation.gpu_events.**GPUEventProvider**

Bases: *dace.codegen.instrumentation.provider.InstrumentationProvider*

Timing instrumentation that reports GPU/copy time using CUDA/HIP events.

on_node_begin(*sdfg*, *state*, *node*, *outer_stream*, *inner_stream*, *global_stream*)

Event called at the beginning of generating a node. :param sdfg: The generated SDFG object. :param state: The generated SDFGState object. :param node: The generated node. :param outer_stream: Code generator for the internal code before

the scope is opened.

Parameters

- **inner_stream** – Code generator for the internal code within the scope (at the beginning).
- **global_stream** – Code generator for global (external) code.

on_node_end(*sdfg*, *state*, *node*, *outer_stream*, *inner_stream*, *global_stream*)
Event called at the end of generating a node. :param sdfg: The generated SDFG object. :param state: The generated SDFGState object. :param node: The generated node. :param outer_stream: Code generator for the internal code after

the scope is closed.

Parameters

- **inner_stream** – Code generator for the internal code within the scope (at the end).
- **global_stream** – Code generator for global (external) code.

on_scope_entry(*sdfg*, *state*, *node*, *outer_stream*, *inner_stream*, *global_stream*)
Event called at the beginning of a scope (on generating an EntryNode). :param sdfg: The generated SDFG object. :param state: The generated SDFGState object. :param node: The EntryNode object from which code is generated. :param outer_stream: Code generator for the internal code before

the scope is opened.

Parameters

- **inner_stream** – Code generator for the internal code within the scope (at the beginning).
- **global_stream** – Code generator for global (external) code.

on_scope_exit(*sdfg*, *state*, *node*, *outer_stream*, *inner_stream*, *global_stream*)
Event called at the end of a scope (on generating an ExitNode). :param sdfg: The generated SDFG object. :param state: The generated SDFGState object. :param node: The ExitNode object from which code is generated. :param outer_stream: Code generator for the internal code after

the scope is closed.

Parameters

- **inner_stream** – Code generator for the internal code within the scope (at the end).
- **global_stream** – Code generator for global (external) code.

on_sdfg_begin(*sdfg*, *local_stream*, *global_stream*)
Event called at the beginning of SDFG code generation. :param sdfg: The generated SDFG object. :param local_stream: Code generator for the in-function code. :param global_stream: Code generator for global (external) code.

on_state_begin(*sdfg*, *state*, *local_stream*, *global_stream*)
Event called at the beginning of SDFG state code generation. :param sdfg: The generated SDFG object. :param state: The generated SDFGState object. :param local_stream: Code generator for the in-function code. :param global_stream: Code generator for global (external) code.

on_state_end(*sdfg*, *state*, *local_stream*, *global_stream*)
Event called at the end of SDFG state code generation. :param sdfg: The generated SDFG object. :param state: The generated SDFGState object. :param local_stream: Code generator for the in-function code. :param global_stream: Code generator for global (external) code.

dace.codegen.instrumentation.papi module

Implements the PAPI counter performance instrumentation provider. Used for collecting CPU performance counters.

class dace.codegen.instrumentation.papi.**PAPIInstrumentation**
Bases: *dace.codegen.instrumentation.provider.InstrumentationProvider*

Instrumentation provider that reports CPU performance counters using the PAPI library.

configure_papi()

get_unique_number()

static has_surrounding_perfcounters(*node*, *dfg: dace.sdfg.state.StateGraphView*)
Returns true if there is a possibility that this node is part of a section that is profiled.

on_consume_entry(*sdfg*, *state*, *node*, *outer_stream*, *inner_stream*)

on_copy_begin(*sdfg*, *state*, *src_node*, *dst_node*, *edge*, *local_stream*, *global_stream*, *copy_shape*, *src_strides*, *dst_strides*)
Event called at the beginning of generating a copy operation. :param sdfg: The generated SDFG object. :param state: The generated SDFGState object. :param src_node: The source node of the copy. :param dst_node: The destination node of the copy. :param edge: An edge in the memlet path of the copy. :param local_stream: Code generator for the internal code. :param global_stream: Code generator for global (external) code. :param copy_shape: Tuple representing the shape of the copy. :param src_strides: Element-skipping strides for each dimension of the copied source. :param dst_strides: Element-skipping strides for each dimension of the copied destination.

on_copy_end(*sdfg*, *state*, *src_node*, *dst_node*, *edge*, *local_stream*, *global_stream*)
Event called at the end of generating a copy operation. :param sdfg: The generated SDFG object. :param state: The generated SDFGState object. :param src_node: The source node of the copy. :param dst_node: The destination node of the copy. :param edge: An edge in the memlet path of the copy. :param local_stream: Code generator for the internal code. :param global_stream: Code generator for global (external) code.

on_map_entry(*sdfg*, *state*, *node*, *outer_stream*, *inner_stream*)

on_node_begin(*sdfg*, *state*, *node*, *outer_stream*, *inner_stream*, *global_stream*)
Event called at the beginning of generating a node. :param sdfg: The generated SDFG object. :param state: The generated SDFGState object. :param node: The generated node. :param outer_stream: Code generator for the internal code before

the scope is opened.

Parameters

- **inner_stream** – Code generator for the internal code within the scope (at the beginning).
- **global_stream** – Code generator for global (external) code.

on_node_end(*sdfg*, *state*, *node*, *outer_stream*, *inner_stream*, *global_stream*)
Event called at the end of generating a node. :param sdfg: The generated SDFG object. :param state: The generated SDFGState object. :param node: The generated node. :param outer_stream: Code generator for the internal code after

the scope is closed.

Parameters

- **inner_stream** – Code generator for the internal code within the scope (at the end).
- **global_stream** – Code generator for global (external) code.

on_scope_entry(*sdfg*, *state*, *node*, *outer_stream*, *inner_stream*, *global_stream*)
Event called at the beginning of a scope (on generating an EntryNode). :param sdfg: The generated SDFG object. :param state: The generated SDFGState object. :param node: The EntryNode object from which code is generated. :param outer_stream: Code generator for the internal code before

the scope is opened.

Parameters

- **inner_stream** – Code generator for the internal code within the scope (at the beginning).
- **global_stream** – Code generator for global (external) code.

on_scope_exit(*sdfg*, *state*, *node*, *outer_stream*, *inner_stream*, *global_stream*)
Event called at the end of a scope (on generating an ExitNode). :param sdfg: The generated SDFG object. :param state: The generated SDFGState object. :param node: The ExitNode object from which code is generated. :param outer_stream: Code generator for the internal code after

the scope is closed.

Parameters

- **inner_stream** – Code generator for the internal code within the scope (at the end).
- **global_stream** – Code generator for global (external) code.

on_sdfg_begin(*sdfg*, *local_stream*, *global_stream*)
Event called at the beginning of SDFG code generation. :param sdfg: The generated SDFG object. :param local_stream: Code generator for the in-function code. :param global_stream: Code generator for global (external) code.

on_sdfg_end(*sdfg*, *local_stream*, *global_stream*)
Event called at the end of SDFG code generation. :param sdfg: The generated SDFG object. :param local_stream: Code generator for the in-function code. :param global_stream: Code generator for global (external) code.

on_state_begin(*sdfg*, *state*, *local_stream*, *global_stream*)
Event called at the beginning of SDFG state code generation. :param sdfg: The generated SDFG object. :param state: The generated SDFGState object. :param local_stream: Code generator for the in-function code. :param global_stream: Code generator for global (external) code.

static perf_counter_end_measurement_string(*unified_id*)

perf_counter_start_measurement_string(*unified_id: int*, *iteration: str*, *core_str: str = 'PAPI_thread_id()'*)

perf_counter_string()
Creates a performance counter template string.

static perf_counter_string_from_string_list(*counterlist: [<class 'str'>]*)
Creates a performance counter typename string.

static perf_get_supersection_start_string(*node*, *dfg*, *unified_id*)

static perf_section_start_string(*unified_id: int*, *size: str*, *in_size: str*, *core_str: str = 'PAPI_thread_id()'*)

static perf_supersection_start_string(*unified_id*)

perf_whitelist_schedules = [<ScheduleType.CPU_Multicore: 4>, <ScheduleType.Sequential

static should_instrument_entry(*map_entry: dace.sdfg.nodes.EntryNode*) → bool
Returns True if this entry node should be instrumented.

class dace.codegen.instrumentation.papi.**PAPIUtils**
Bases: object

General-purpose utilities for working with PAPI.

static accumulate_byte_movement(*outermost_node*, *node*, *dfg: dace.sdfg.state.StateGraphView*, *sdfg*, *state_id*)

static all_maps(*map_entry: dace.sdfg.nodes.EntryNode*, *dfg: dace.sdfg.graph.SubgraphView*) → List[dace.sdfg.nodes.EntryNode]
Returns all scope entry nodes within a scope entry.

static available_counters() → Dict[str, int]
Returns the available PAPI counters on this machine. Only works on *nix based systems with `grep` and `papi-tools` installed. :return: A set of available PAPI counters in the form of a dictionary

mapping from counter name to the number of native hardware events.

static get_iteration_count(*map_entry: dace.sdfg.nodes.MapEntry*, *mapvars: dict*)
Get the number of iterations for this map, allowing other variables as bounds.

static get_memlet_byte_size(*sdfg: dace.sdfg.sdfg.SDFG*, *memlet: dace.memlet.Memlet*)
Returns the memlet size in bytes, depending on its data type. :param sdfg: The SDFG in which the memlet resides. :param memlet: Memlet to return size in bytes. :return: The size as a symbolic expression.

static get_memory_input_size(*node*, *sdfg*, *state_id*) → str

static get_out_memlet_costs(*sdfg: dace.sdfg.sdfg.SDFG*, *state_id: int*, *node: dace.sdfg.nodes.Node*, *dfg: dace.sdfg.state.StateGraphView*)

static get_parents(*outermost_node: dace.sdfg.nodes.Node*, *node: dace.sdfg.nodes.Node*, *sdfg: dace.sdfg.sdfg.SDFG*, *state_id: int*) → List[dace.sdfg.nodes.Node]

static get_tasklet_byte_accesses(*tasklet: dace.sdfg.nodes.CodeNode*, *dfg: dace.sdfg.state.StateGraphView*, *sdfg: dace.sdfg.sdfg.SDFG*, *state_id: int*) → str
Get the amount of bytes processed by *tasklet*. The formula is sum(inedges * size) + sum(outedges * size)

static is_papi_used(*sdfg: dace.sdfg.sdfg.SDFG*) → bool
Returns True if any of the SDFG elements includes PAPI counter instrumentation.

static reduce_iteration_count(*begin*, *end*, *step*, *rparams: dict*)

dace.codegen.instrumentation.perfdb module

dace.codegen.instrumentation.provider module

class dace.codegen.instrumentation.provider.**InstrumentationProvider**
Bases: object

Instrumentation provider for SDFGs, states, scopes, and memlets. Emits code on event.

extensions()

static get_provider_mapping() → Dict[dace.dtypes.InstrumentationType, Type[dace.codegen.instrumentation.provider.InstrumentationProvider]]
Returns a dictionary that maps instrumentation types to provider class types, given the currently-registered extensions of this class.

on_copy_begin(*sdfg*, *state*, *src_node*, *dst_node*, *edge*, *local_stream*, *global_stream*, *copy_shape*, *src_strides*, *dst_strides*)
Event called at the beginning of generating a copy operation. :param sdfg: The generated SDFG object. :param state: The generated SDFGState object. :param src_node: The source node of the copy. :param dst_node: The destination node of the copy. :param edge: An edge in the memlet path of the copy. :param local_stream: Code generator for the internal code. :param global_stream: Code generator for global (external) code. :param copy_shape: Tuple representing the shape of the copy. :param src_strides: Element-skipping strides for each dimension of the copied source. :param dst_strides: Element-skipping strides for each dimension of the copied destination.

on_copy_end(*sdfg*, *state*, *src_node*, *dst_node*, *edge*, *local_stream*, *global_stream*)
Event called at the end of generating a copy operation. :param sdfg: The generated SDFG object. :param state: The generated SDFGState object. :param src_node: The source node of the copy. :param dst_node: The destination node of the copy. :param edge: An edge in the memlet path of the copy. :param local_stream: Code generator for the internal code. :param global_stream: Code generator for global (external) code.

on_node_begin(*sdfg*, *state*, *node*, *outer_stream*, *inner_stream*, *global_stream*)
Event called at the beginning of generating a node. :param sdfg: The generated SDFG object. :param state: The generated SDFGState object. :param node: The generated node. :param outer_stream: Code generator for the internal code before

the scope is opened.

Parameters

- **inner_stream** – Code generator for the internal code within the scope (at the beginning).
- **global_stream** – Code generator for global (external) code.

on_node_end(*sdfg*, *state*, *node*, *outer_stream*, *inner_stream*, *global_stream*)
Event called at the end of generating a node. :param sdfg: The generated SDFG object. :param state: The generated SDFGState object. :param node: The generated node. :param outer_stream: Code generator for the internal code after

the scope is closed.

Parameters

- **inner_stream** – Code generator for the internal code within the scope (at the end).
- **global_stream** – Code generator for global (external) code.

on_scope_entry(*sdfg*, *state*, *node*, *outer_stream*, *inner_stream*, *global_stream*)
Event called at the beginning of a scope (on generating an EntryNode). :param sdfg: The generated SDFG object. :param state: The generated SDFGState object. :param node: The EntryNode object from which code is generated. :param outer_stream: Code generator for the internal code before

the scope is opened.

Parameters

- **inner_stream** – Code generator for the internal code within the scope (at the beginning).
- **global_stream** – Code generator for global (external) code.

on_scope_exit(*sdfg*, *state*, *node*, *outer_stream*, *inner_stream*, *global_stream*)
Event called at the end of a scope (on generating an ExitNode). :param sdfg: The generated SDFG object. :param state: The generated SDFGState object. :param node: The ExitNode object from which code is generated. :param outer_stream: Code generator for the internal code after

the scope is closed.

Parameters

- **inner_stream** – Code generator for the internal code within the scope (at the end).
- **global_stream** – Code generator for global (external) code.

on_sdfg_begin(*sdfg*, *local_stream*, *global_stream*)
Event called at the beginning of SDFG code generation. :param sdfg: The generated SDFG object. :param local_stream: Code generator for the in-function code. :param global_stream: Code generator for global (external) code.

on_sdfg_end(*sdfg*, *local_stream*, *global_stream*)
Event called at the end of SDFG code generation. :param sdfg: The generated SDFG object. :param local_stream: Code generator for the in-function code. :param global_stream: Code generator for global (external) code.

on_state_begin(*sdfg*, *state*, *local_stream*, *global_stream*)
Event called at the beginning of SDFG state code generation. :param sdfg: The generated SDFG object. :param state: The generated SDFGState object. :param local_stream: Code generator for the in-function code. :param global_stream: Code generator for global (external) code.

on_state_end(*sdfg*, *state*, *local_stream*, *global_stream*)
Event called at the end of SDFG state code generation. :param sdfg: The generated SDFG object. :param state: The generated SDFGState object. :param local_stream: Code generator for the in-function code. :param global_stream: Code generator for global (external) code.

register(***kwargs*)

unregister()

dace.codegen.instrumentation.timer module

class dace.codegen.instrumentation.timer.**TimerProvider**
Bases: *dace.codegen.instrumentation.provider.InstrumentationProvider*

Timing instrumentation that reports wall-clock time directly after timed execution is complete.

on_node_begin(*sdfg*, *state*, *node*, *outer_stream*, *inner_stream*, *global_stream*)
Event called at the beginning of generating a node. :param sdfg: The generated SDFG object. :param state: The generated SDFGState object. :param node: The generated node. :param outer_stream: Code generator for the internal code before

the scope is opened.

Parameters

- **inner_stream** – Code generator for the internal code within the scope (at the beginning).
- **global_stream** – Code generator for global (external) code.

on_node_end(*sdfg*, *state*, *node*, *outer_stream*, *inner_stream*, *global_stream*)
Event called at the end of generating a node. :param sdfg: The generated SDFG object. :param state: The generated SDFGState object. :param node: The generated node. :param outer_stream: Code generator for the internal code after

the scope is closed.

Parameters

- **inner_stream** – Code generator for the internal code within the scope (at the end).
- **global_stream** – Code generator for global (external) code.

on_scope_entry(*sdfg*, *state*, *node*, *outer_stream*, *inner_stream*, *global_stream*)
Event called at the beginning of a scope (on generating an EntryNode). :param sdfg: The generated SDFG object. :param state: The generated SDFGState object. :param node: The EntryNode object from which code is generated. :param outer_stream: Code generator for the internal code before

the scope is opened.

Parameters

- **inner_stream** – Code generator for the internal code within the scope (at the beginning).
- **global_stream** – Code generator for global (external) code.

on_scope_exit(*sdfg*, *state*, *node*, *outer_stream*, *inner_stream*, *global_stream*)
Event called at the end of a scope (on generating an ExitNode). :param sdfg: The generated SDFG object. :param state: The generated SDFGState object. :param node: The ExitNode object from which code is generated. :param outer_stream: Code generator for the internal code after

the scope is closed.

Parameters

- **inner_stream** – Code generator for the internal code within the scope (at the end).
- **global_stream** – Code generator for global (external) code.

on_sdfg_begin(*sdfg*, *local_stream*, *global_stream*)
Event called at the beginning of SDFG code generation. :param sdfg: The generated SDFG object. :param local_stream: Code generator for the in-function code. :param global_stream: Code generator for global (external) code.

on_sdfg_end(*sdfg*, *local_stream*, *global_stream*)
Event called at the end of SDFG code generation. :param sdfg: The generated SDFG object. :param local_stream: Code generator for the in-function code. :param global_stream: Code generator for global (external) code.

on_state_begin(*sdfg*, *state*, *local_stream*, *global_stream*)
Event called at the beginning of SDFG state code generation. :param sdfg: The generated SDFG object. :param state: The generated SDFGState object. :param local_stream: Code generator for the in-function code. :param global_stream: Code generator for global (external) code.

on_state_end(*sdfg*, *state*, *local_stream*, *global_stream*)
Event called at the end of SDFG state code generation. :param sdfg: The generated SDFG object. :param state: The generated SDFGState object. :param local_stream: Code generator for the in-function code. :param global_stream: Code generator for global (external) code.

on_tbegin(*stream: dace.codegen.prettycode.CodeIOStream*, *sdfg=None*, *state=None*, *node=None*)

on_tend(*timer_name: str*, *stream: dace.codegen.prettycode.CodeIOStream*, *sdfg=None*, *state=None*, *node=None*)

Module contents

dace.codegen.targets package

Submodules

dace.codegen.targets.cpu module

class dace.codegen.targets.cpu.**CPUCodeGen**(*frame_codegen*, *sdfg*)
Bases: *dace.codegen.targets.target.TargetCodeGenerator*

SDFG CPU code generator.

allocate_array(*sdfg*, *dfg*, *state_id*, *node*, *nodedesc*, *function_stream*, *declaration_stream*, *allocation_stream*)
Generates code for allocating an array, outputting to the given code streams. :param sdfg: The SDFG to generate code from. :param dfg: The SDFG state to generate code from. :param state_id: The node ID of the state in the given SDFG. :param node: The data node to generate allocation for. :param nodedesc: The data descriptor to allocate. :param global_stream: A *CodeIOStream* object that will be

generated outside the calling code, for use when generating global functions.

Parameters

- **declaration_stream** – A *CodeIOStream* object that points to the point of array declaration.
- **allocation_stream** – A *CodeIOStream* object that points to the call-site of array allocation.

allocate_view(*sdfg: dace.sdfg.sdfg.SDFG*, *dfg: dace.sdfg.state.SDFGState*, *state_id: int*, *node: dace.sdfg.nodes.AccessNode*, *global_stream: dace.codegen.prettycode.CodeIOStream*, *declaration_stream: dace.codegen.prettycode.CodeIOStream*, *allocation_stream: dace.codegen.prettycode.CodeIOStream*)
Allocates (creates pointer and refers to original) a view of an existing array, scalar, or view.

static cmake_options()

copy_memory(*sdfg*, *dfg*, *state_id*, *src_node*, *dst_node*, *edge*, *function_stream*, *callsite_stream*)
Generates code for copying memory, either from a data access node (array/stream) to another, a code node (tasklet/nested SDFG) to another, or a combination of the two. :param sdfg: The SDFG to generate code from. :param dfg: The SDFG state to generate code from. :param state_id: The node ID of the state in the given SDFG. :param src_node: The source node to generate copy code for. :param dst_node: The destination node to generate copy code for. :param edge: The edge representing the copy (in the innermost

scope, adjacent to either the source or destination node).

Parameters

- **function_stream** – A *CodeIOStream* object that will be generated outside the calling code, for use when generating global functions.

- **callsite_stream** – A *CodeIOStream* object that points to the current location (call-site) in the code.

deallocate_array(*sdfg*, *dfg*, *state_id*, *node*, *nodedesc*, *function_stream*, *callsite_stream*)
Generates code for deallocating an array, outputting to the given code streams. :param sdfg: The SDFG to generate code from. :param dfg: The SDFG state to generate code from. :param state_id: The node ID of the state in the given SDFG. :param node: The data node to generate deallocation for. :param nodedesc: The data descriptor to deallocate. :param function_stream: A *CodeIOStream* object that will be

generated outside the calling code, for use when generating global functions.

Parameters **callsite_stream** – A *CodeIOStream* object that points to the current location (call-site) in the code.

declare_array(*sdfg*, *dfg*, *state_id*, *node*, *nodedesc*, *function_stream*, *declaration_stream*)
Generates code for declaring an array without allocating it, outputting to the given code streams. :param sdfg: The SDFG to generate code from. :param dfg: The SDFG state to generate code from. :param state_id: The node ID of the state in the given SDFG. :param node: The data node to generate allocation for. :param nodedesc: The data descriptor to allocate. :param global_stream: A *CodeIOStream* object that will be

generated outside the calling code, for use when generating global functions.

Parameters **declaration_stream** – A *CodeIOStream* object that points to the point of array declaration.

define_out_memlet(*sdfg*, *state_dfg*, *state_id*, *src_node*, *dst_node*, *edge*, *function_stream*, *callsite_stream*)

generate_node(*sdfg*, *dfg*, *state_id*, *node*, *function_stream*, *callsite_stream*)
Generates code for a single node, outputting it to the given code streams. :param sdfg: The SDFG to generate code from. :param dfg: The SDFG state to generate code from. :param state_id: The node ID of the state in the given SDFG. :param node: The node to generate code from. :param function_stream: A *CodeIOStream* object that will be

generated outside the calling code, for use when generating global functions.

Parameters **callsite_stream** – A *CodeIOStream* object that points to the current location (call-site) in the code.

generate_nsdfg_arguments(*sdfg*, *dfg*, *state*, *node*)

generate_nsdfg_call(*sdfg*, *state*, *node*, *memlet_references*, *sdfg_label*, *state_struct=True*)

generate_nsdfg_header(*sdfg*, *state*, *state_id*, *node*, *memlet_references*, *sdfg_label*, *state_struct=True*)

generate_scope(*sdfg: dace.sdfg.sdfg.SDFG*, *dfg_scope: dace.sdfg.scope.ScopeSubgraphView*, *state_id*, *function_stream*, *callsite_stream*)
Generates code for an SDFG state scope (from a scope-entry node to its corresponding scope-exit node), outputting it to the given code streams. :param sdfg: The SDFG to generate code from. :param dfg_scope: The *ScopeSubgraphView* to generate code from. :param state_id: The node ID of the state in the given SDFG. :param function_stream: A *CodeIOStream* object that will be

generated outside the calling code, for use when generating global functions.

Parameters **callsite_stream** – A *CodeIOStream* object that points to the current location (call-site) in the code.

generate_scope_postamble(*sdfg*, *dfg_scope*, *state_id*, *function_stream*, *outer_stream*, *inner_stream*)

Generates code for the end of an SDFG scope, outputting it to the given code streams. :param sdfg: The SDFG to generate code from. :param dfg_scope: The *ScopeSubgraphView* to generate code from. :param state_id: The node ID of the state in the given SDFG. :param function_stream: A *CodeIOStream* object that will be

generated outside the calling code, for use when generating global functions.

Parameters

- **outer_stream** – A *CodeIOStream* object that points to the code after the scope (e.g., after for-loop closing braces or kernel invocations).
- **inner_stream** – A *CodeIOStream* object that points to the end of the inner scope code (e.g., before for-loop closing braces or end of kernel).

generate_scope_preamble(*sdfg*, *dfg_scope*, *state_id*, *function_stream*, *outer_stream*, *inner_stream*)

Generates code for the beginning of an SDFG scope, outputting it to the given code streams. :param sdfg: The SDFG to generate code from. :param dfg_scope: The *ScopeSubgraphView* to generate code from. :param state_id: The node ID of the state in the given SDFG. :param function_stream: A *CodeIOStream* object that will be

generated outside the calling code, for use when generating global functions.

Parameters

- **outer_stream** – A *CodeIOStream* object that points to the code before the scope generation (e.g., before for-loops or kernel invocations).
- **inner_stream** – A *CodeIOStream* object that points to the beginning of the scope code (e.g., inside for-loops or beginning of kernel).

generate_tasklet_postamble(*sdfg*, *dfg_scope*, *state_id*, *node*, *function_stream*, *before_memlets_stream*, *after_memlets_stream*)

Generates code for the end of a tasklet. This method is intended to be overloaded by subclasses. :param sdfg: The SDFG to generate code from. :param dfg_scope: The *ScopeSubgraphView* to generate code from. :param state_id: The node ID of the state in the given SDFG. :param node: The tasklet node in the state. :param function_stream: A *CodeIOStream* object that will be

generated outside the calling code, for use when generating global functions.

Parameters

- **before_memlets_stream** – A *CodeIOStream* object that will emit code before output memlets are generated.
- **after_memlets_stream** – A *CodeIOStream* object that will emit code after output memlets are generated.

generate_tasklet_preamble(*sdfg*, *dfg_scope*, *state_id*, *node*, *function_stream*, *before_memlets_stream*, *after_memlets_stream*)

Generates code for the beginning of a tasklet. This method is intended to be overloaded by subclasses. :param sdfg: The SDFG to generate code from. :param dfg_scope: The *ScopeSubgraphView* to generate code from. :param state_id: The node ID of the state in the given SDFG. :param node: The tasklet node in the state. :param function_stream: A *CodeIOStream* object that will be

generated outside the calling code, for use when generating global functions.

Parameters

- **before_memlets_stream** – A *CodeIOStream* object that will emit code before input memlets are generated.
- **after_memlets_stream** – A *CodeIOStream* object that will emit code after input memlets are generated.

get_generated_codeobjects()
Returns a list of generated *CodeObject* classes corresponding to files with generated code. If an empty list is returned (default) then this code generator does not create new files. @see: CodeObject

has_finalizer
Returns True if the target generates a *__dace_exit_<TARGET>* function that should be called on finalization.

has_initializer
Returns True if the target generates a *__dace_init_<TARGET>* function that should be called on initialization.

language = 'cpp'

make_ptr_assignment(*src_expr*, *src_dtype*, *dst_expr*, *dst_dtype*, *codegen=None*)
Write source to destination, where the source is a scalar, and the destination is a pointer. :return: String of C++ performing the write.

make_ptr_vector_cast(**args*, ***kwargs*)

memlet_ctor(*sdfg*, *memlet*, *dtype*, *is_output*)

memlet_definition(*sdfg: dace.sdfg.sdfg.SDFG*, *memlet: dace.memlet.Memlet*, *output: bool*, *local_name: str*, *conntype: Union[dace.data.Data, dace.dtypes.typeclass] = None*, *allow_shadowing=False*, *codegen=None*)

memlet_stream_ctor(*sdfg*, *memlet*)

memlet_view_ctor(*sdfg*, *memlet*, *dtype*, *is_output*)

process_out_memlets(*sdfg*, *state_id*, *node*, *dfg*, *dispatcher*, *result*, *locals_defined*, *function_stream*, *skip_wcr=False*, *codegen=None*)

target_name = 'cpu'

title = 'CPU'

unparse_tasklet(*sdfg*, *state_id*, *dfg*, *node*, *function_stream*, *inner_stream*, *locals*, *ldepth*, *toplevel_schedule*)

write_and_resolve_expr(*sdfg*, *memlet*, *nc*, *outname*, *inname*, *indices=None*, *dtype=None*)
Emits a conflict resolution call from a memlet.

dace.codegen.targets.cuda module

class dace.codegen.targets.cuda.**CUDACodeGen**(*frame_codegen*, *sdfg: dace.sdfg.sdfg.SDFG*)
Bases: *dace.codegen.targets.target.TargetCodeGenerator*

GPU (CUDA/HIP) code generator.

allocate_array(*sdfg*, *dfg*, *state_id*, *node*, *nodedesc*, *function_stream*, *declaration_stream*, *allocation_stream*)
Generates code for allocating an array, outputting to the given code streams. :param sdfg: The SDFG to generate code from. :param dfg: The SDFG state to generate code from. :param state_id: The node ID of

the state in the given SDFG. :param node: The data node to generate allocation for. :param nodedesc: The data descriptor to allocate. :param global_stream: A *CodeIOStream* object that will be

generated outside the calling code, for use when generating global functions.

Parameters

- **declaration_stream** – A *CodeIOStream* object that points to the point of array declaration.
- **allocation_stream** – A *CodeIOStream* object that points to the call-site of array allocation.

allocate_stream(*sdfg*, *dfg*, *state_id*, *node*, *nodedesc*, *function_stream*, *declaration_stream*, *allocation_stream*)

static cmake_options()

copy_memory(*sdfg*, *dfg*, *state_id*, *src_node*, *dst_node*, *memlet*, *function_stream*, *callsite_stream*)

Generates code for copying memory, either from a data access node (array/stream) to another, a code node (tasklet/nested SDFG) to another, or a combination of the two. :param sdfg: The SDFG to generate code from. :param dfg: The SDFG state to generate code from. :param state_id: The node ID of the state in the given SDFG. :param src_node: The source node to generate copy code for. :param dst_node: The destination node to generate copy code for. :param edge: The edge representing the copy (in the innermost

scope, adjacent to either the source or destination node).

Parameters

- **function_stream** – A *CodeIOStream* object that will be generated outside the calling code, for use when generating global functions.
- **callsite_stream** – A *CodeIOStream* object that points to the current location (call-site) in the code.

deallocate_array(*sdfg*, *dfg*, *state_id*, *node*, *nodedesc*, *function_stream*, *callsite_stream*)

Generates code for deallocating an array, outputting to the given code streams. :param sdfg: The SDFG to generate code from. :param dfg: The SDFG state to generate code from. :param state_id: The node ID of the state in the given SDFG. :param node: The data node to generate deallocation for. :param nodedesc: The data descriptor to deallocate. :param function_stream: A *CodeIOStream* object that will be

generated outside the calling code, for use when generating global functions.

Parameters **callsite_stream** – A *CodeIOStream* object that points to the current location (call-site) in the code.

deallocate_stream(*sdfg*, *dfg*, *state_id*, *node*, *nodedesc*, *function_stream*, *callsite_stream*)

declare_array(*sdfg*, *dfg*, *state_id*, *node*, *nodedesc*, *function_stream*, *declaration_stream*)

Generates code for declaring an array without allocating it, outputting to the given code streams. :param sdfg: The SDFG to generate code from. :param dfg: The SDFG state to generate code from. :param state_id: The node ID of the state in the given SDFG. :param node: The data node to generate allocation for. :param nodedesc: The data descriptor to allocate. :param global_stream: A *CodeIOStream* object that will be

generated outside the calling code, for use when generating global functions.

Parameters **declaration_stream** – A *CodeIOStream* object that points to the point of array declaration.

define_out_memlet(*sdfg*, *state_dfg*, *state_id*, *src_node*, *dst_node*, *edge*, *function_stream*, *callsite_stream*)

generate_devicelevel_scope(*sdfg*, *dfg_scope*, *state_id*, *function_stream*, *callsite_stream*)

generate_devicelevel_state(*sdfg*, *state*, *function_stream*, *callsite_stream*)

generate_kernel_scope(*sdfg: dace.sdfg.sdfg.SDFG*, *dfg_scope: dace.sdfg.scope.ScopeSubgraphView*, *state_id: int*, *kernel_map: dace.sdfg.nodes.Map*, *kernel_name: str*, *grid_dims: list*, *block_dims: list*, *has_tbmap: bool*, *has_dtbmap: bool*, *kernel_params: list*, *function_stream: dace.codegen.prettycode.CodeIOStream*, *kernel_stream: dace.codegen.prettycode.CodeIOStream*)

generate_node(*sdfg*, *dfg*, *state_id*, *node*, *function_stream*, *callsite_stream*)

Generates code for a single node, outputting it to the given code streams. :param sdfg: The SDFG to generate code from. :param dfg: The SDFG state to generate code from. :param state_id: The node ID of the state in the given SDFG. :param node: The node to generate code from. :param function_stream: A *CodeIOStream* object that will be

generated outside the calling code, for use when generating global functions.

Parameters **callsite_stream** – A *CodeIOStream* object that points to the current location (call-site) in the code.

generate_nsdfg_arguments(*sdfg*, *dfg*, *state*, *node*)

generate_nsdfg_call(*sdfg*, *state*, *node*, *memlet_references*, *sdfg_label*)

generate_nsdfg_header(*sdfg*, *state*, *state_id*, *node*, *memlet_references*, *sdfg_label*)

generate_scope(*sdfg*, *dfg_scope*, *state_id*, *function_stream*, *callsite_stream*)

Generates code for an SDFG state scope (from a scope-entry node to its corresponding scope-exit node), outputting it to the given code streams. :param sdfg: The SDFG to generate code from. :param dfg_scope: The *ScopeSubgraphView* to generate code from. :param state_id: The node ID of the state in the given SDFG. :param function_stream: A *CodeIOStream* object that will be

generated outside the calling code, for use when generating global functions.

Parameters **callsite_stream** – A *CodeIOStream* object that points to the current location (call-site) in the code.

generate_state(*sdfg*, *state*, *function_stream*, *callsite_stream*)

Generates code for an SDFG state, outputting it to the given code streams. :param sdfg: The SDFG to generate code from. :param state: The SDFGState to generate code from. :param function_stream: A *CodeIOStream* object that will be

generated outside the calling code, for use when generating global functions.

Parameters **callsite_stream** – A *CodeIOStream* object that points to the current location (call-site) in the code.

get_generated_codeobjects()

Returns a list of generated *CodeObject* classes corresponding to files with generated code. If an empty list is returned (default) then this code generator does not create new files. @see: CodeObject

get_kernel_dimensions(*dfg_scope*)

Determines a GPU kernel's grid/block dimensions from map scopes.

Ruleset for kernel dimensions:

1. If only one map (device-level) exists, of an integer set S, the block size is 32x1x1 and grid size is ceil(|S|/32) in 1st dimension.
2. If nested thread-block maps exist (T_1,...,T_n), grid size is |S| and block size is max(|T_1|,...,|T_nl) with block specialization.
3. **If block size can be overapproximated, it is (for** dynamically-sized blocks that are bounded by a predefined size).

@note: Kernel dimensions are separate from the map variables, and they should be treated as such.

@note: To make use of the grid/block 3D registers, we use multi- dimensional kernels up to 3 dimensions, and flatten the rest into the third dimension.

get_next_scope_entries(*dfg*, *scope_entry*)

get_tb_maps_recursive(*subgraph*)

has_finalizer
Returns True if the target generates a *__dace_exit_<TARGET>* function that should be called on finalization.

has_initializer
Returns True if the target generates a *__dace_init_<TARGET>* function that should be called on initialization.

make_ptr_vector_cast(**args*, ***kwargs*)

node_dispatch_predicate(*sdfg*, *state*, *node*)

on_target_used() → None
Called before generating frame code (headers / footers) on this target if it was dispatched for any reason. Can be used to set up state struct fields.

process_out_memlets(**args*, ***kwargs*)

state_dispatch_predicate(*sdfg*, *state*)

target_name = 'cuda'

title = 'CUDA'

dace.codegen.targets.cuda.**cpu_to_gpu_cpred**(*sdfg*, *state*, *src_node*, *dst_node*)
Copy predicate from CPU to GPU that determines when a copy is illegal. Returns True if copy is illegal, False otherwise.

dace.codegen.targets.cuda.**prod**(*iterable*)

dace.codegen.targets.framecode module

class dace.codegen.targets.framecode.**DaCeCodeGenerator**(**args*, ***kwargs*)
Bases: object

DaCe code generator class that writes the generated code for SDFG state machines, and uses a dispatcher to generate code for individual states based on the target.

allocate_arrays_in_scope(*sdfg: dace.sdfg.sdfg.SDFG*, *scope: Union[dace.sdfg.nodes.EntryNode, dace.sdfg.state.SDFGState, dace.sdfg.sdfg.SDFG]*, *function_stream: dace.codegen.prettycode.CodeIOStream*, *callsite_stream: dace.codegen.prettycode.CodeIOStream*)
Dispatches allocation of all arrays in the given scope.

deallocate_arrays_in_scope(*sdfg: dace.sdfg.sdfg.SDFG, scope: Union[dace.sdfg.nodes.EntryNode, dace.sdfg.state.SDFGState, dace.sdfg.sdfg.SDFG], function_stream: dace.codegen.prettycode.CodeIOStream, callsite_stream: dace.codegen.prettycode.CodeIOStream*)
Dispatches deallocation of all arrays in the given scope.

determine_allocation_lifetime(*top_sdfg: dace.sdfg.sdfg.SDFG*)
Determines where (at which scope/state/SDFG) each data descriptor will be allocated/deallocated. :param top_sdfg: The top-level SDFG to determine for.

dispatcher

generate_code(*sdfg: dace.sdfg.sdfg.SDFG, schedule: Optional[dace.dtypes.ScheduleType], sdfg_id: str = ''*) → Tuple[str, str, Set[dace.codegen.targets.target.TargetCodeGenerator], Set[str]]
Generate frame code for a given SDFG, calling registered targets' code generation callbacks for them to generate their own code. :param sdfg: The SDFG to generate code for. :param schedule: The schedule the SDFG is currently located, or

None if the SDFG is top-level.

Parameters **sdfg_id** – An optional string id given to the SDFG label

Returns A tuple of the generated global frame code, local frame code, and a set of targets that have been used in the generation of this SDFG.

generate_constants(*sdfg: dace.sdfg.sdfg.SDFG, callsite_stream: dace.codegen.prettycode.CodeIOStream*)

generate_fileheader(*sdfg: dace.sdfg.sdfg.SDFG, global_stream: dace.codegen.prettycode.CodeIOStream, backend: str = 'frame'*)
Generate a header in every output file that includes custom types and constants. :param sdfg: The input SDFG. :param global_stream: Stream to write to (global). :param backend: Whose backend this header belongs to.

generate_footer(*sdfg: dace.sdfg.sdfg.SDFG, global_stream: dace.codegen.prettycode.CodeIOStream, callsite_stream: dace.codegen.prettycode.CodeIOStream*)
Generate the footer of the frame-code. Code exists in a separate function for overriding purposes. :param sdfg: The input SDFG. :param global_stream: Stream to write to (global). :param callsite_stream: Stream to write to (at call site).

generate_header(*sdfg: dace.sdfg.sdfg.SDFG, global_stream: dace.codegen.prettycode.CodeIOStream, callsite_stream: dace.codegen.prettycode.CodeIOStream*)
Generate the header of the frame-code. Code exists in a separate function for overriding purposes. :param sdfg: The input SDFG. :param global_stream: Stream to write to (global). :param callsite_stream: Stream to write to (at call site).

generate_state(*sdfg*, *state*, *global_stream*, *callsite_stream*, *generate_state_footer=True*)

generate_states(*sdfg*, *global_stream*, *callsite_stream*)

dace.codegen.targets.mpi module

class dace.codegen.targets.mpi.**MPICodeGen**(*frame_codegen*, *sdfg: dace.sdfg.sdfg.SDFG*)
Bases: *dace.codegen.targets.target.TargetCodeGenerator*

An MPI code generator.

static cmake_options()

generate_scope(*sdfg*, *dfg_scope*, *state_id*, *function_stream*, *callsite_stream*)
Generates code for an SDFG state scope (from a scope-entry node to its corresponding scope-exit node), outputting it to the given code streams. :param sdfg: The SDFG to generate code from. :param dfg_scope: The *ScopeSubgraphView* to generate code from. :param state_id: The node ID of the state in the given SDFG. :param function_stream: A *CodeIOStream* object that will be

generated outside the calling code, for use when generating global functions.

Parameters **callsite_stream** – A *CodeIOStream* object that points to the current location (call-site) in the code.

get_generated_codeobjects()
Returns a list of generated *CodeObject* classes corresponding to files with generated code. If an empty list is returned (default) then this code generator does not create new files. @see: CodeObject

has_finalizer
Returns True if the target generates a *__dace_exit_<TARGET>* function that should be called on finalization.

has_initializer
Returns True if the target generates a *__dace_init_<TARGET>* function that should be called on initialization.

language = 'cpp'

target_name = 'mpi'

title = 'MPI'

dace.codegen.targets.target module

class dace.codegen.targets.target.**IllegalCopy**
Bases: *dace.codegen.targets.target.TargetCodeGenerator*

A code generator that is triggered when invalid copies are specified by the SDFG. Only raises an exception on failure.

copy_memory(*sdfg*, *dfg*, *state_id*, *src_node*, *dst_node*, *edge*, *function_stream*, *callsite_stream*)
Generates code for copying memory, either from a data access node (array/stream) to another, a code node (tasklet/nested SDFG) to another, or a combination of the two. :param sdfg: The SDFG to generate code from. :param dfg: The SDFG state to generate code from. :param state_id: The node ID of the state in the given SDFG. :param src_node: The source node to generate copy code for. :param dst_node: The destination node to generate copy code for. :param edge: The edge representing the copy (in the innermost

scope, adjacent to either the source or destination node).

Parameters

- **function_stream** – A *CodeIOStream* object that will be generated outside the calling code, for use when generating global functions.
- **callsite_stream** – A *CodeIOStream* object that points to the current location (call-site) in the code.

class dace.codegen.targets.target.**TargetCodeGenerator**
Bases: object

Interface dictating functions that generate code for: * Array allocation/deallocation/initialization/copying * Scope (map, consume) code generation

allocate_array(*sdfg: dace.sdfg.sdfg.SDFG*, *dfg: dace.sdfg.state.SDFGState*, *state_id: int*, *node: dace.sdfg.nodes.Node*, *nodedesc: dace.data.Data*, *global_stream: dace.codegen.prettycode.CodeIOStream*, *declaration_stream: dace.codegen.prettycode.CodeIOStream*, *allocation_stream: dace.codegen.prettycode.CodeIOStream*) → None

Generates code for allocating an array, outputting to the given code streams. :param sdfg: The SDFG to generate code from. :param dfg: The SDFG state to generate code from. :param state_id: The node ID of the state in the given SDFG. :param node: The data node to generate allocation for. :param nodedesc: The data descriptor to allocate. :param global_stream: A *CodeIOStream* object that will be

generated outside the calling code, for use when generating global functions.

Parameters

- **declaration_stream** – A *CodeIOStream* object that points to the point of array declaration.
- **allocation_stream** – A *CodeIOStream* object that points to the call-site of array allocation.

copy_memory(*sdfg: dace.sdfg.sdfg.SDFG, dfg: dace.sdfg.state.SDFGState, state_id: int, src_node: dace.sdfg.nodes.Node, dst_node: dace.sdfg.nodes.Node, edge: dace.sdfg.graph.MultiConnectorEdge[dace.memlet.Memlet][dace.memlet.Memlet], function_stream: dace.codegen.prettycode.CodeIOStream, callsite_stream: dace.codegen.prettycode.CodeIOStream*) → None

Generates code for copying memory, either from a data access node (array/stream) to another, a code node (tasklet/nested SDFG) to another, or a combination of the two. :param sdfg: The SDFG to generate code from. :param dfg: The SDFG state to generate code from. :param state_id: The node ID of the state in the given SDFG. :param src_node: The source node to generate copy code for. :param dst_node: The destination node to generate copy code for. :param edge: The edge representing the copy (in the innermost

scope, adjacent to either the source or destination node).

Parameters

- **function_stream** – A *CodeIOStream* object that will be generated outside the calling code, for use when generating global functions.
- **callsite_stream** – A *CodeIOStream* object that points to the current location (call-site) in the code.

deallocate_array(*sdfg: dace.sdfg.sdfg.SDFG*, *dfg: dace.sdfg.state.SDFGState*, *state_id: int*, *node: dace.sdfg.nodes.Node*, *nodedesc: dace.data.Data*, *function_stream: dace.codegen.prettycode.CodeIOStream*, *callsite_stream: dace.codegen.prettycode.CodeIOStream*) → None

Generates code for deallocating an array, outputting to the given code streams. :param sdfg: The SDFG to generate code from. :param dfg: The SDFG state to generate code from. :param state_id: The node ID of the state in the given SDFG. :param node: The data node to generate deallocation for. :param nodedesc: The data descriptor to deallocate. :param function_stream: A *CodeIOStream* object that will be

generated outside the calling code, for use when generating global functions.

Parameters **callsite_stream** – A *CodeIOStream* object that points to the current location (call-site) in the code.

declare_array(*sdfg: dace.sdfg.sdfg.SDFG*, *dfg: dace.sdfg.state.SDFGState*, *state_id: int*, *node: dace.sdfg.nodes.Node*, *nodedesc: dace.data.Data*, *global_stream: dace.codegen.prettycode.CodeIOStream*, *declaration_stream: dace.codegen.prettycode.CodeIOStream*) → None

Generates code for declaring an array without allocating it, outputting to the given code streams. :param sdfg: The SDFG to generate code from. :param dfg: The SDFG state to generate code from. :param state_id: The node ID of the state in the given SDFG. :param node: The data node to generate allocation for. :param nodedesc: The data descriptor to allocate. :param global_stream: A *CodeIOStream* object that will be

generated outside the calling code, for use when generating global functions.

Parameters **declaration_stream** – A *CodeIOStream* object that points to the point of array declaration.

extensions()

generate_node(*sdfg: dace.sdfg.sdfg.SDFG*, *dfg: dace.sdfg.state.SDFGState*, *state_id: int*, *node: dace.sdfg.nodes.Node*, *function_stream: dace.codegen.prettycode.CodeIOStream*, *callsite_stream: dace.codegen.prettycode.CodeIOStream*) → None

Generates code for a single node, outputting it to the given code streams. :param sdfg: The SDFG to generate code from. :param dfg: The SDFG state to generate code from. :param state_id: The node ID of the state in the given SDFG. :param node: The node to generate code from. :param function_stream: A *CodeIOStream* object that will be

generated outside the calling code, for use when generating global functions.

Parameters **callsite_stream** – A *CodeIOStream* object that points to the current location (call-site) in the code.

generate_scope(*sdfg: dace.sdfg.sdfg.SDFG*, *dfg_scope: dace.sdfg.scope.ScopeSubgraphView*, *state_id: int*, *function_stream: dace.codegen.prettycode.CodeIOStream*, *callsite_stream: dace.codegen.prettycode.CodeIOStream*) → None

Generates code for an SDFG state scope (from a scope-entry node to its corresponding scope-exit node), outputting it to the given code streams. :param sdfg: The SDFG to generate code from. :param dfg_scope: The *ScopeSubgraphView* to generate code from. :param state_id: The node ID of the state in the given SDFG. :param function_stream: A *CodeIOStream* object that will be

generated outside the calling code, for use when generating global functions.

Parameters **callsite_stream** – A *CodeIOStream* object that points to the current location (call-site) in the code.

generate_state(*sdfg: dace.sdfg.sdfg.SDFG*, *state: dace.sdfg.state.SDFGState*, *function_stream: dace.codegen.prettycode.CodeIOStream*, *callsite_stream: dace.codegen.prettycode.CodeIOStream*) → None

Generates code for an SDFG state, outputting it to the given code streams. :param sdfg: The SDFG to generate code from. :param state: The SDFGState to generate code from. :param function_stream: A *CodeIOStream* object that will be

generated outside the calling code, for use when generating global functions.

Parameters **callsite_stream** – A *CodeIOStream* object that points to the current location (call-site) in the code.

get_generated_codeobjects() → List[dace.codegen.codeobject.CodeObject]
Returns a list of generated *CodeObject* classes corresponding to files with generated code. If an empty list is returned (default) then this code generator does not create new files. @see: CodeObject

has_finalizer
Returns True if the target generates a *__dace_exit_<TARGET>* function that should be called on finalization.

has_initializer
Returns True if the target generates a *__dace_init_<TARGET>* function that should be called on initialization.

on_target_used() → None
Called before generating frame code (headers / footers) on this target if it was dispatched for any reason. Can be used to set up state struct fields.

register(***kwargs*)

unregister()

dace.codegen.targets.target.**make_absolute**(*path: str*) → str
Finds an executable and returns an absolute path out of it. Used when finding compiler executables. :param path: Executable name, relative path, or absolute path. :return: Absolute path pointing to the same file as path.

dace.codegen.targets.xilinx module

class dace.codegen.targets.xilinx.**XilinxCodeGen**(**args*, ***kwargs*)
Bases: dace.codegen.targets.fpga.FPGACodeGen

Xilinx FPGA code generator.

allocate_view(*sdfg: dace.sdfg.sdfg.SDFG*, *dfg: dace.sdfg.state.SDFGState*, *state_id: int*, *node: dace.sdfg.nodes.AccessNode*, *global_stream: dace.codegen.prettycode.CodeIOStream*, *declaration_stream: dace.codegen.prettycode.CodeIOStream*, *allocation_stream: dace.codegen.prettycode.CodeIOStream*)

static cmake_options()

define_local_array(*var_name*, *desc*, *array_size*, *function_stream*, *kernel_stream*, *sdfg*, *state_id*, *node*)

define_shift_register(***kwargs*)

static define_stream(*dtype*, *buffer_size*, *var_name*, *array_size*, *function_stream*, *kernel_stream*, *sdfg*)
Defines a stream :return: a tuple containing the type of the created variable, and boolean indicating

whether this is a global variable or not

generate_converter(***kwargs*)

static generate_flatten_loop_post(*kernel_stream*, *sdfg*, *state_id*, *node*)

static generate_flatten_loop_pre(*kernel_stream*, *sdfg*, *state_id*, *node*)

generate_host_function_body(*sdfg: dace.sdfg.sdfg.SDFG*, *state: dace.sdfg.state.SDFGState*, *kernel_name: str*, *predecessors: list*, *parameters: list*, *rtl_tasklet_names: list*, *kernel_stream: dace.codegen.prettycode.CodeIOStream*, *instrumentation_stream: dace.codegen.prettycode.CodeIOStream*)
Generate the host-specific code for spawning and synchronizing the given kernel. :param sdfg: :param

state: :param predecessors: list containing all the name of kernels that must be finished before starting this one :param parameters: list containing the kernel parameters (of all kernels in this state) :param rtl_tasklet_names :param kernel_stream: Device-specific code stream :param instrumentation_stream: Code for profiling kernel execution time.

generate_host_header(*sdfg*, *kernel_function_name*, *parameters*, *host_code_stream*)

static generate_kernel_boilerplate_post(*kernel_stream*, *sdfg*, *state_id*)

generate_kernel_boilerplate_pre(*sdfg*, *state_id*, *kernel_name*, *parameters*, *bank_assignments*, *module_stream*, *kernel_stream*, *external_streams*)

generate_kernel_internal(*sdfg: dace.sdfg.sdfg.SDFG*, *state: dace.sdfg.state.SDFGState*, *kernel_name: str*, *predecessors: list*, *subgraphs: list*, *kernel_stream: dace.codegen.prettycode.CodeIOStream*, *state_host_header_stream: dace.codegen.prettycode.CodeIOStream*, *state_host_body_stream: dace.codegen.prettycode.CodeIOStream*, *instrumentation_stream: dace.codegen.prettycode.CodeIOStream*, *function_stream: dace.codegen.prettycode.CodeIOStream*, *callsite_stream: dace.codegen.prettycode.CodeIOStream*, *state_parameters: list*)

Generates Kernel code, both device and host side. :param sdfg: :param state: :param kernel_name: :param predecessors: list containing all the name of kernels from which this one depends :param subgraphs: :param kernel_stream: Device code stream, contains the kernel code :param state_host_header_stream: Device-specific code stream: contains the host code

for the state global declarations.

Parameters

- **state_host_body_stream** – Device-specific code stream: contains all the code related to this state, for creating transient buffers, spawning kernels, and synchronizing them.
- **instrumentation_stream** – Code for profiling kernel execution time.
- **function_stream** – CPU code stream.
- **callsite_stream** – CPU code stream.
- **state_parameters** – list of state parameters. The kernel-specific parameters will be appended to it.

generate_memlet_definition(*sdfg*, *dfg*, *state_id*, *src_node*, *dst_node*, *edge*, *callsite_stream*)

generate_module(*sdfg*, *state*, *kernel_name*, *name*, *subgraph*, *parameters*, *module_stream*, *entry_stream*, *host_stream*, *instrumentation_stream*)

Generates a module that will run as a dataflow function in the FPGA kernel.

generate_no_dependence_post(*kernel_stream*, *sdfg: dace.sdfg.sdfg.SDFG*, *state_id: int*, *node: dace.sdfg.nodes.Node*, *var_name: str*, *accessed_subset: Union[int, dace.subsets.Subset] = None*)

Adds post loop pragma for ignoring loop carried dependencies on a given variable

static generate_no_dependence_pre(*kernel_stream*, *sdfg*, *state_id*, *node*, *var_name=None*)

generate_nsdfg_arguments(*sdfg*, *dfg*, *state*, *node*)

generate_nsdfg_header(*sdfg*, *state*, *state_id*, *node*, *memlet_references*, *sdfg_label*)

static generate_pipeline_loop_post(*kernel_stream*, *sdfg*, *state_id*, *node*)

static generate_pipeline_loop_pre(*kernel_stream*, *sdfg*, *state_id*, *node*)

static generate_unroll_loop_post(*kernel_stream*, *factor*, *sdfg*, *state_id*, *node*)

generate_unroll_loop_pre(*kernel_stream*, *factor*, *sdfg*, *state_id*, *node*)

get_generated_codeobjects()
Returns a list of generated *CodeObject* classes corresponding to files with generated code. If an empty list is returned (default) then this code generator does not create new files. @see: CodeObject

language = 'hls'

static make_kernel_argument(*data: dace.data.Data, var_name: str, subset_info: Union[int, dace.subsets.Subset], sdfg: dace.sdfg.sdfg.SDFG, is_output: bool, with_vectorization: bool, interface_id: Union[int, List[int]] = None*)

make_ptr_assignment(*src_expr*, *src_dtype*, *dst_expr*, *dst_dtype*)
Write source to destination, where the source is a scalar, and the destination is a pointer. :return: String of C++ performing the write.

static make_read(*defined_type*, *dtype*, *var_name*, *expr*, *index*, *is_pack*, *packing_factor*)

make_shift_register_write(*defined_type*, *dtype*, *var_name*, *write_expr*, *index*, *read_expr*, *wcr*, *is_unpack*, *packing_factor*, *sdfg*)

static make_vector_type(*dtype*, *is_const*)

static make_write(*defined_type*, *dtype*, *var_name*, *write_expr*, *index*, *read_expr*, *wcr*, *is_unpack*, *packing_factor*)

rtl_tasklet_name(*node: dace.sdfg.nodes.RTLTasklet*, *state*, *sdfg*)

target_name = 'xilinx'

title = 'Xilinx'

unparse_tasklet(**args*, ***kwargs*)

write_and_resolve_expr(*sdfg*, *memlet*, *nc*, *outname*, *inname*, *indices=None*, *dtype=None*)
Emits a conflict resolution call from a memlet.

Module contents

Submodules

dace.codegen.codegen module

dace.codegen.codegen.**generate_code**(*sdfg*, *validate=True*) → List[dace.codegen.codeobject.CodeObject]
Generates code as a list of code objects for a given SDFG. :param sdfg: The SDFG to use :param validate: If True, validates the SDFG before generating the code. :return: List of code objects that correspond to files to compile.

dace.codegen.codegen.**generate_dummy**(*sdfg: dace.sdfg.sdfg.SDFG*) → str
Generates a C program calling this SDFG. Since we do not know the purpose/semantics of the program, we allocate the right types and and guess values for scalars.

dace.codegen.codegen.**generate_headers**(*sdfg: dace.sdfg.sdfg.SDFG*) → str
Generate a header file for the SDFG

dace.codegen.codeobject module

class dace.codegen.codeobject.**CodeObject**(**args*, ***kwargs*)
Bases: object

clean_code

code
The code attached to this object

environments
Environments required by CMake to build and run this code node.

extra_compiler_kwargs
Additional compiler argument variables to add to template

language
Language used for this code (same as its file extension)

linkable
Should this file participate in overall linkage?

name
Filename to use

properties()

target
Target to use for compilation

target_type
Sub-target within target (e.g., host or device code)

title
Title of code for GUI

dace.codegen.compiler module

Handles compilation of code objects. Creates the proper folder structure, compiles each target separately, links all targets to one binary, and returns the corresponding CompiledSDFG object.

dace.codegen.compiler.**configure_and_compile**(*program_folder*, *program_name=None*, *output_stream=None*)
Configures and compiles a DaCe program in the specified folder into a shared library file.

Parameters

- **program_folder** – Folder containing all files necessary to build, equivalent to what was passed to *generate_program_folder*.
- **output_stream** – Additional output stream to write to (used for DIODE client).

Returns Path to the compiled shared library file.

dace.codegen.compiler.**generate_program_folder**(*sdfg*, *code_objects: List[dace.codegen.codeobject.CodeObject]*, *out_path: str*, *config=None*)
Writes all files required to configure and compile the DaCe program into the specified folder.

Parameters

- **sdfg** – The SDFG to generate the program folder for.

- **code_objects** – List of generated code objects.
- **out_path** – The folder in which the build files should be written.

Returns Path to the program folder.

dace.codegen.compiler.**get_binary_name**(*object_folder*, *object_name*, *lib_extension='so'*)

dace.codegen.compiler.**get_environment_flags**(*environments*) → Tuple[List[str], Set[str]]
Returns the CMake environment and linkage flags associated with the given input environments/libraries. :param environments: A list of `@dace.library.environment`-decorated

classes.

Returns A 2-tuple of (environment CMake flags, linkage CMake flags)

dace.codegen.compiler.**get_program_handle**(*library_path*, *sdfg*)

dace.codegen.compiler.**identical_file_exists**(*filename: str*, *file_contents: str*)

dace.codegen.compiler.**load_from_file**(*sdfg*, *binary_filename*)

dace.codegen.compiler.**unique_flags**(*flags*)

dace.codegen.cppunparse module

class dace.codegen.cppunparse.**CPPLocals**
Bases: *dace.codegen.cppunparse.LocalScheme*

clear_scope(*from_indentation*)
Clears all locals defined in indentation 'from_indentation' and deeper

define(*local_name*, *lineno*, *depth*, *dtype=None*)

get_name_type_associations()

is_defined(*local_name*, *current_depth*)

class dace.codegen.cppunparse.**CPPUnparser**(*tree*, *depth*, *locals*, *file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*, *indent_output=True*, *expr_semicolon=True*, *indent_offset=0*, *type_inference=False*, *defined_symbols=None*, *language=<Language.CPP: 2>*)
Bases: object

Methods in this class recursively traverse an AST and output C++ source code for the abstract syntax; original formatting is disregarded.

binop = {'Add': '+', 'BitAnd': '&', 'BitOr': '|', 'BitXor': '^', 'Div': '/', 'LS

boolops = {<class '_ast.And'>: '&&', <class '_ast.Or'>: '||'}

cmpops = {'Eq': '==', 'Gt': '>', 'GtE': '>=', 'Is': '==', 'IsNot': '!=', 'Lt': '<

dispatch(*tree*)
Dispatcher function, dispatching tree type T to method _T.

dispatch_lhs_tuple(*targets*)

enter()
Print '{', and increase the indentation.

fill(*text=''*)
Indent a piece of text, according to the current indentation level

format_conversions = {97: 'a', 114: 'r', 115: 's'}

funcops = {'FloorDiv': (' /', 'dace::math::ifloor'), 'MatMult': (',', 'dace::gemm')}

leave()
Decrease the indentation and print '}'.

unop = {'Invert': '~', 'Not': '!', 'UAdd': '+', 'USub': '-'}

write(*text*)
Append a piece of text to the current line

class dace.codegen.cppunparse.**LocalScheme**
Bases: object

clear_scope(*from_indentation*)

define(*local_name*, *lineno*, *depth*)

is_defined(*local_name*, *current_depth*)

dace.codegen.cppunparse.**cppunparse**(*node*, *expr_semicolon=True*, *locals=None*, *defined_symbols=None*)

dace.codegen.cppunparse.**interleave**(*inter*, *f*, *seq*, ***kwargs*)
Call f on each item in seq, calling inter() in between. f can accept optional arguments (kwargs)

dace.codegen.cppunparse.**py2cpp**(*code*, *expr_semicolon=True*, *defined_symbols=None*)

dace.codegen.cppunparse.**pyexpr2cpp**(*expr*)

dace.codegen.prettycode module

Code I/O stream that automates indentation and mapping of code to SDFG nodes.

class dace.codegen.prettycode.**CodeIOStream**(*base_indentation=0*)
Bases: _io.StringIO

Code I/O stream that automates indentation and mapping of code to SDFG nodes.

write(*contents*, *sdfg=None*, *state_id=None*, *node_id=None*)
Write string to file.

Returns the number of characters written, which is always equal to the length of the string.

Module contents

dace.frontend package

Subpackages

dace.frontend.common package

Submodules

dace.frontend.common.op_repository module

class dace.frontend.common.op_repository.**Replacements**
Bases: object

A management singleton for functions that replace existing function calls with either an SDFG subgraph. Used in the Python frontend to replace functions such as *numpy.ndarray* and operators such as *Array.__add__*.

static get(*name: str*)
Returns an implementation of a function.

static get_attribute(*classname: str*, *attr_name: str*)

static get_method(*classname: str*, *method_name: str*)

static get_ufunc(*ufunc_method: str = None*)
Returns the implementation for NumPy universal functions.

static getop(*classname: str*, *optype: str*, *otherclass: str = None*)
Returns an implementation of an operator.

dace.frontend.common.op_repository.**replaces**(*func: Callable[[...], Tuple[str]], name: str*)
Registers a replacement sub-SDFG generator for a function. :param func: A function that receives an SDFG, SDFGState, and the original function

arguments, returning a tuple of array names to connect to the outputs.

Parameters **name** – Full name (pydoc-compliant, including package) of function to replace.

dace.frontend.common.op_repository.**replaces_attribute**(*func: Callable[[...], Tuple[str]], classname: str, attr_name: str*)
Registers a replacement sub-SDFG generator for object attributes. :param func: A function that receives an SDFG, SDFGState, and the original

function arguments, returning a tuple of array names to connect to the outputs.

Parameters

- **classname** – Full name (pydoc-compliant, including package) of the object class.
- **attr_name** – Name of the attribute.

dace.frontend.common.op_repository.**replaces_method**(*func: Callable[[...], Tuple[str]], classname: str, method_name: str*)
Registers a replacement sub-SDFG generator for methods on objects. :param func: A function that receives an SDFG, SDFGState, and the original

function arguments, returning a tuple of array names to connect to the outputs.

Parameters

- **classname** – Full name (pydoc-compliant, including package) of the object class.
- **method_name** – Name of the invoked method.

dace.frontend.common.op_repository.**replaces_operator**(*func: Callable[[Any, Any, str, str], Tuple[str]], classname: str, optype: str, otherclass: str = None*)

Registers a replacement sub-SDFG generator for an operator. :param func: A function that receives an SDFG, SDFGState, and the two operand array names,

returning a tuple of array names to connect to the outputs.

Parameters

- **classname** – The name of the class to implement the operator for (extends dace.Data).
- **optype** – The type (as string) of the operator to replace (extends ast.operator).
- **otherclass** – Optional argument defining operators for a second class that differs from the first.

dace.frontend.common.op_repository.**replaces_ufunc**(*func: Callable[[...], Tuple[str]], name: str*)

Registers a replacement sub-SDFG generator for NumPy universal functions and methods.

Parameters

- **func** – A function that receives a ProgramVisitor, AST call node, SDFG, SDFGState, ufunc name, and the original function positional and keyword arguments, returning a tuple of array names to connect to the outputs.
- **name** – 'ufunc' for NumPy ufunc or ufunc method name for replacing the NumPy ufunc methods.

Module contents

dace.frontend.octave package

Submodules

dace.frontend.octave.ast_arrayaccess module

class dace.frontend.octave.ast_arrayaccess.**AST_ArrayAccess**(*context, arrayname, accdims*)

Bases: *dace.frontend.octave.ast_node.AST_Node*

generate_code(*sdfg*, *state*)

get_basetype()

get_children()

get_dims()

is_data_dependent_access()

make_range_from_accdims()

replace_child(*old*, *new*)

dace.frontend.octave.ast_assign module

class dace.frontend.octave.ast_assign.**AST_Assign**(*context*, *lhs*, *rhs*, *op*)
Bases: *dace.frontend.octave.ast_node.AST_Node*

defined_variables()

generate_code(*sdfg*, *state*)

get_children()

print_nodes(*state*)

provide_parents(*parent*)

replace_child(*old*, *new*)

dace.frontend.octave.ast_expression module

class dace.frontend.octave.ast_expression.**AST_BinExpression**(*context*, *lhs*, *rhs*, *op*)
Bases: *dace.frontend.octave.ast_node.AST_Node*

generate_code(*sdfg*, *state*)

get_basetype()

get_children()

get_dims()

matrix2d_matrix2d_mult(*sdfg*, *state*)

matrix2d_matrix2d_plus_or_minus(*sdfg*, *state*, *op*)

matrix2d_scalar(*sdfg*, *state*, *op*)

provide_parents(*parent*)

replace_child(*old*, *new*)

scalar_scalar(*sdfg*, *state*, *op*)

vec_mult_vect(*sdfg*, *state*, *op*)

class dace.frontend.octave.ast_expression.**AST_UnaryExpression**(*context*, *arg*, *op*, *order*)
Bases: *dace.frontend.octave.ast_node.AST_Node*

get_children()

replace_child(*old*, *new*)

specialize()
Some nodes can be simplified after parsing the complete AST and before actually generating code, i.e., AST_FunCall nodes could be function calls or array accesses, and we don't really know unless we know the context of the call.

This function traverses the AST and tries to specialize nodes after completing the AST. It should be called on the top-level AST_Statements node, and a node that wants to be specialized should return its new instance. If no specialzation should take place, it should return None.

dace.frontend.octave.ast_function module

class dace.frontend.octave.ast_function.**AST_Argument**(*context*, *name*, *default=None*)
Bases: *dace.frontend.octave.ast_node.AST_Node*

get_children()

class dace.frontend.octave.ast_function.**AST_BuiltInFunCall**(*context*, *funname*, *args*)
Bases: *dace.frontend.octave.ast_node.AST_Node*

generate_code(*sdfg*, *state*)

get_basetype()

get_children()

get_dims()

replace_child(*old*, *new*)

specialize()
Some nodes can be simplified after parsing the complete AST and before actually generating code, i.e., AST_FunCall nodes could be function calls or array accesses, and we don't really know unless we know the context of the call.

This function traverses the AST and tries to specialize nodes after completing the AST. It should be called on the top-level AST_Statements node, and a node that wants to be specialized should return its new instance. If no specialzation should take place, it should return None.

class dace.frontend.octave.ast_function.**AST_EndFunc**(*context*)
Bases: *dace.frontend.octave.ast_node.AST_Node*

generate_code(*sdfg*, *state*)

get_children()

replace_child(*old*, *new*)

class dace.frontend.octave.ast_function.**AST_FunCall**(*context*, *funname*, *args*)
Bases: *dace.frontend.octave.ast_node.AST_Node*

get_children()

replace_child(*old*, *new*)

specialize()
Some nodes can be simplified after parsing the complete AST and before actually generating code, i.e., AST_FunCall nodes could be function calls or array accesses, and we don't really know unless we know the context of the call.

This function traverses the AST and tries to specialize nodes after completing the AST. It should be called on the top-level AST_Statements node, and a node that wants to be specialized should return its new instance. If no specialzation should take place, it should return None.

class dace.frontend.octave.ast_function.**AST_Function**(*context*, *name*, *args*, *retvals*)
Bases: *dace.frontend.octave.ast_node.AST_Node*

generate_code(*sdfg*, *state*)

get_children()

replace_child(*old*, *new*)

set_statements(*stmtlist*)

dace.frontend.octave.ast_loop module

class dace.frontend.octave.ast_loop.**AST_ForLoop**(*context*, *var*, *initializer*, *stmts*)
Bases: *dace.frontend.octave.ast_node.AST_Node*

generate_code(*sdfg*, *state*)

generate_code_proper(*sdfg*, *state*)

get_children()

replace_child(*old*, *new*)

dace.frontend.octave.ast_matrix module

class dace.frontend.octave.ast_matrix.**AST_Matrix**(*context*, *rows*)
Bases: *dace.frontend.octave.ast_node.AST_Node*

generate_code(*sdfg*, *state*)

get_basetype()

get_children()

get_dims()

get_values_row_major()

is_constant()

provide_parents(*parent*)

replace_child(*old*, *new*)

class dace.frontend.octave.ast_matrix.**AST_Matrix_Row**(*context*, *elements*)
Bases: *dace.frontend.octave.ast_node.AST_Node*

get_children()

get_dims()

is_constant()

provide_parents(*parent*)

replace_child(*old*, *new*)

class dace.frontend.octave.ast_matrix.**AST_Transpose**(*context*, *arg*, *op*)
Bases: *dace.frontend.octave.ast_node.AST_Node*

generate_code(*sdfg*, *state*)

get_basetype()

get_children()

get_dims()

replace_child(*old*, *new*)

dace.frontend.octave.ast_node module

class dace.frontend.octave.ast_node.**AST_Node**(*context*)
Bases: object

defined_variables()

find_data_node_in_sdfg_state(*sdfg*, *state*, *nodename=None*)

generate_code(**args*)

get_children()

get_datanode(*sdfg*, *state*)

get_initializers(*sdfg*)

get_name_in_sdfg(*sdfg*)
If this node has no name assigned yet, create a new one of the form *__tmp_X* where *X* is an integer, such that this node does not yet exist in the given SDFG. @note: We assume that we create exactly one SDFG from each AST,

otherwise we need to store the hash of the SDFG the name was created for (would be easy but seems useless at this point).

get_new_tmpvar(*sdfg*)

get_parent()

print_as_tree()

provide_parents(*parent*)

replace_child(*old*, *new*)

replace_parent(*newparent*)

search_vardef_in_scope(*name*)

shortdesc()

specialize()
Some nodes can be simplified after parsing the complete AST and before actually generating code, i.e., AST_FunCall nodes could be function calls or array accesses, and we don't really know unless we know the context of the call.

This function traverses the AST and tries to specialize nodes after completing the AST. It should be called on the top-level AST_Statements node, and a node that wants to be specialized should return its new instance. If no specialzation should take place, it should return None.

class dace.frontend.octave.ast_node.**AST_Statements**(*context*, *stmts*)
Bases: *dace.frontend.octave.ast_node.AST_Node*

append_statement(*stmt*)

generate_code(*sdfg=None*, *state=None*)

get_children()

provide_parents(*parent=None*)

replace_child(*old*, *new*)

specialize()
Some nodes can be simplified after parsing the complete AST and before actually generating code, i.e.,

AST_FunCall nodes could be function calls or array accesses, and we don't really know unless we know the context of the call.

This function traverses the AST and tries to specialize nodes after completing the AST. It should be called on the top-level AST_Statements node, and a node that wants to be specialized should return its new instance. If no specialzation should take place, it should return None.

dace.frontend.octave.ast_nullstmt module

class dace.frontend.octave.ast_nullstmt.**AST_Comment**(*context*, *text*)
Bases: *dace.frontend.octave.ast_node.AST_Node*

generate_code(*sdfg*, *state*)

get_children()

replace_child(*old*, *new*)

class dace.frontend.octave.ast_nullstmt.**AST_EndStmt**(*context*)
Bases: *dace.frontend.octave.ast_node.AST_Node*

get_children()

replace_child(*old*, *new*)

class dace.frontend.octave.ast_nullstmt.**AST_NullStmt**(*context*)
Bases: *dace.frontend.octave.ast_node.AST_Node*

generate_code(*sdfg*, *state*)

get_children()

replace_child(*old*, *new*)

dace.frontend.octave.ast_range module

class dace.frontend.octave.ast_range.**AST_RangeExpression**(*context*, *lhs*, *rhs*)
Bases: *dace.frontend.octave.ast_node.AST_Node*

generate_code(*sdfg*, *state*)

get_basetype()

get_children()

get_dims()

replace_child(*old*, *new*)

specialize()
Some nodes can be simplified after parsing the complete AST and before actually generating code, i.e., AST_FunCall nodes could be function calls or array accesses, and we don't really know unless we know the context of the call.

This function traverses the AST and tries to specialize nodes after completing the AST. It should be called on the top-level AST_Statements node, and a node that wants to be specialized should return its new instance. If no specialzation should take place, it should return None.

dace.frontend.octave.ast_values module

class dace.frontend.octave.ast_values.**AST_Constant**(*context*, *value*)
Bases: *dace.frontend.octave.ast_node.AST_Node*

generate_code(*sdfg*, *state*)

get_basetype()

get_children()

get_dims()

get_value()

is_constant()

replace_child(*old*, *new*)

class dace.frontend.octave.ast_values.**AST_Ident**(*context*, *value*)
Bases: *dace.frontend.octave.ast_node.AST_Node*

generate_code(*sdfg*, *state*)

get_basetype()
Check in the scope if this is defined and return the basetype of the corresponding SDFG access node this currently maps to.

get_children()

get_dims()

get_name()

get_name_in_sdfg(*sdfg*)
If this node has no name assigned yet, create a new one of the form *__tmp_X* where *X* is an integer, such that this node does not yet exist in the given SDFG. @note: We assume that we create exactly one SDFG from each AST,

otherwise we need to store the hash of the SDFG the name was created for (would be easy but seems useless at this point).

get_propagated_value()

is_constant()

replace_child(*old*, *new*)

specialize()
Some nodes can be simplified after parsing the complete AST and before actually generating code, i.e., AST_FunCall nodes could be function calls or array accesses, and we don't really know unless we know the context of the call.

This function traverses the AST and tries to specialize nodes after completing the AST. It should be called on the top-level AST_Statements node, and a node that wants to be specialized should return its new instance. If no specialzation should take place, it should return None.

dace.frontend.octave.lexer module

dace.frontend.octave.lexer.**main**()

dace.frontend.octave.lexer.**new**()

dace.frontend.octave.lexer.**raise_exception**(*error_type*, *message*, *my_lexer*)

dace.frontend.octave.parse module

dace.frontend.octave.parse.**p_arg1**(*p*)
arg1 : IDENT

dace.frontend.octave.parse.**p_arg2**(*p*)

arg1 [NUMBER]

STRING

dace.frontend.octave.parse.**p_arg_list**(*p*)

arg_list [ident_init_opt]

arg_list COMMA ident_init_opt

dace.frontend.octave.parse.**p_args**(*p*)

args [arg1]

args arg1

dace.frontend.octave.parse.**p_break_stmt**(*p*)
break_stmt : BREAK SEMI

dace.frontend.octave.parse.**p_case_list**(*p*)

case_list :

CASE expr sep stmt_list_opt case_list
CASE expr error stmt_list_opt case_list
OTHERWISE stmt_list

dace.frontend.octave.parse.**p_cellarray**(*p*)

cellarray [LBRACE RBRACE]

LBRACE matrix_row RBRACE
LBRACE matrix_row SEMI RBRACE

dace.frontend.octave.parse.**p_cellarray_2**(*p*)
cellarray : LBRACE expr_list RBRACE

dace.frontend.octave.parse.**p_cellarrayref**(*p*)
expr : expr LBRACE expr_list RBRACE | expr LBRACE RBRACE

dace.frontend.octave.parse.**p_command**(*p*)
command : ident args SEMI

dace.frontend.octave.parse.**p_comment_stmt**(*p*)
comment_stmt : COMMENT

dace.frontend.octave.parse.**p_concat_list1**(*p*)
matrix_row : expr_list SEMI expr_list

dace.frontend.octave.parse.**p_concat_list2**(*p*)
matrix_row : matrix_row SEMI expr_list

dace.frontend.octave.parse.**p_continue_stmt**(*p*)
continue_stmt : CONTINUE SEMI

dace.frontend.octave.parse.**p_elseif_stmt**(*p*)

elseif_stmt :

ELSE stmt_list_opt

ELSEIF expr sep stmt_list_opt elseif_stmt

ELSEIF LPAREN expr RPAREN stmt_list_opt elseif_stmt

dace.frontend.octave.parse.**p_end**(*p*)
top : top END_STMT

dace.frontend.octave.parse.**p_end_function**(*p*)
top : top END_FUNCTION

dace.frontend.octave.parse.**p_error**(*p*)

dace.frontend.octave.parse.**p_error_stmt**(*p*)
error_stmt : ERROR_STMT SEMI

dace.frontend.octave.parse.**p_expr**(*p*)
expr : ident | end | number | string | colon | NEG | matrix | cellarray | expr2 | expr1 | lambda_expr

dace.frontend.octave.parse.**p_expr1**(*p*)
expr1 : MINUS expr %prec UMINUS | PLUS expr %prec UMINUS | NEG expr | HANDLE ident | PLUSPLUS ident | MINUSMINUS ident

dace.frontend.octave.parse.**p_expr2**(*p*)
expr2 : expr AND expr | expr ANDAND expr | expr BACKSLASH expr | expr COLON expr | expr DIV expr | expr DOT expr | expr DOTDIV expr | expr DOTDIVEQ expr | expr DOTEXP expr | expr DOTMUL expr | expr DOTMULEQ expr | expr EQEQ expr | expr POW expr | expr EXP expr | expr EXPEQ expr | expr GE expr | expr GT expr | expr LE expr | expr LT expr | expr MINUS expr | expr MUL expr | expr NE expr | expr OR expr | expr OROR expr | expr PLUS expr | expr EQ expr | expr MULEQ expr | expr DIVEQ expr | expr MINUSEQ expr | expr PLUSEQ expr | expr OREQ expr | expr ANDEQ expr

dace.frontend.octave.parse.**p_expr_2**(*p*)
expr : expr PLUSPLUS | expr MINUSMINUS

dace.frontend.octave.parse.**p_expr_colon**(*p*)
colon : COLON

dace.frontend.octave.parse.**p_expr_end**(*p*)
end : END_EXPR

dace.frontend.octave.parse.**p_expr_ident**(*p*)
ident : IDENT

dace.frontend.octave.parse.**p_expr_list**(*p*)

expr_list [exprs]

exprs COMMA

dace.frontend.octave.parse.**p_expr_number**(*p*)
number : NUMBER

dace.frontend.octave.parse.**p_expr_stmt**(*p*)
expr_stmt : expr_list SEMI

dace.frontend.octave.parse.**p_expr_string**(*p*)
string : STRING

dace.frontend.octave.parse.**p_exprs**(*p*)

exprs [expr]

exprs COMMA expr

dace.frontend.octave.parse.**p_field_expr**(*p*)
expr : expr FIELD

dace.frontend.octave.parse.**p_foo_stmt**(*p*)
foo_stmt : expr OROR expr SEMI

dace.frontend.octave.parse.**p_for_stmt**(*p*)

for_stmt [FOR ident EQ expr SEMI stmt_list END_STMT]

FOR LPAREN ident EQ expr RPAREN SEMI stmt_list END_STMT
FOR matrix EQ expr SEMI stmt_list END_STMT

dace.frontend.octave.parse.**p_func_stmt**(*p*)
func_stmt : FUNCTION ident lambda_args SEMI | FUNCTION ret EQ ident lambda_args SEMI

dace.frontend.octave.parse.**p_funcall_expr**(*p*)
expr : expr LPAREN expr_list RPAREN | expr LPAREN RPAREN

dace.frontend.octave.parse.**p_global**(*p*)
arg1 : GLOBAL

dace.frontend.octave.parse.**p_global_list**(*p*)
global_list : ident | global_list ident

dace.frontend.octave.parse.**p_global_stmt**(*p*)

global_stmt [GLOBAL global_list SEMI]

GLOBAL ident EQ expr SEMI

dace.frontend.octave.parse.**p_ident_init_opt**(*p*)

ident_init_opt [NEG]

ident
ident EQ expr

dace.frontend.octave.parse.**p_if_stmt**(*p*)

if_stmt [IF expr sep stmt_list_opt elseif_stmt END_STMT]

IF LPAREN expr RPAREN stmt_list_opt elseif_stmt END_STMT

dace.frontend.octave.parse.**p_lambda_args**(*p*)
lambda_args : LPAREN RPAREN | LPAREN arg_list RPAREN

dace.frontend.octave.parse.**p_lambda_expr**(*p*)
lambda_expr : HANDLE lambda_args expr

dace.frontend.octave.parse.**p_matrix**(*p*)
matrix : LBRACKET RBRACKET | LBRACKET matrix_row RBRACKET | LBRACKET matrix_row SEMI RBRACKET

dace.frontend.octave.parse.**p_matrix_2**(*p*)
matrix : LBRACKET expr_list RBRACKET | LBRACKET expr_list SEMI RBRACKET

dace.frontend.octave.parse.**p_null_stmt**(*p*)

null_stmt [SEMI]

COMMA

dace.frontend.octave.parse.**p_parens_expr**(*p*)
expr : LPAREN expr RPAREN

dace.frontend.octave.parse.**p_persistent_stmt**(*p*)

persistent_stmt [PERSISTENT global_list SEMI]

PERSISTENT ident EQ expr SEMI

dace.frontend.octave.parse.**p_ret**(*p*)

ret [ident]

LBRACKET RBRACKET
LBRACKET expr_list RBRACKET

dace.frontend.octave.parse.**p_return_stmt**(*p*)
return_stmt : RETURN SEMI

dace.frontend.octave.parse.**p_semi_opt**(*p*)

semi_opt :

semi_opt SEMI
semi_opt COMMA

dace.frontend.octave.parse.**p_separator**(*p*)

sep [COMMA]

SEMI

dace.frontend.octave.parse.**p_stmt**(*p*)

stmt [continue_stmt]

comment_stmt
func_stmt
break_stmt
expr_stmt
global_stmt
persistent_stmt
error_stmt
command
for_stmt
if_stmt
null_stmt
return_stmt
switch_stmt
try_catch
while_stmt
foo_stmt
unwind

dace.frontend.octave.parse.**p_stmt_list**(*p*)

stmt_list [stmt]

stmt_list stmt

dace.frontend.octave.parse.**p_stmt_list_opt**(*p*)

stmt_list_opt :

stmt_list

dace.frontend.octave.parse.**p_switch_stmt**(*p*)
switch_stmt : SWITCH expr semi_opt case_list END_STMT

dace.frontend.octave.parse.**p_top**(*p*)

top :

top stmt

dace.frontend.octave.parse.**p_transpose_expr**(*p*)
expr : expr TRANSPOSE

dace.frontend.octave.parse.**p_try_catch**(*p*)
try_catch : TRY stmt_list CATCH stmt_list END_STMT

dace.frontend.octave.parse.**p_unwind**(*p*)
unwind : UNWIND_PROTECT stmt_list UNWIND_PROTECT_CLEANUP stmt_list END_UNWIND_PROTECT

dace.frontend.octave.parse.**p_while_stmt**(*p*)
while_stmt : WHILE expr SEMI stmt_list END_STMT

dace.frontend.octave.parse.**parse**(*buf*, *debug=False*)

dace.frontend.octave.parsetab module

Module contents

dace.frontend.python package

Submodules

dace.frontend.python.astnodes module

dace.frontend.python.astutils module

Various AST parsing utilities for DaCe.

class dace.frontend.python.astutils.**ASTFindReplace**(*repldict: Dict[str, str]*)
Bases: ast.NodeTransformer

visit_Name(*node: _ast.Name*)

visit_keyword(*node: _ast.keyword*)

class dace.frontend.python.astutils.**ExtNodeTransformer**
Bases: ast.NodeTransformer

A *NodeTransformer* subclass that walks the abstract syntax tree and allows modification of nodes. As opposed to *NodeTransformer*, this class is capable of traversing over top-level expressions in bodies in order to discern DaCe statements from others.

generic_visit(*node*)
Called if no explicit visitor function exists for a node.

visit_TopLevel(*node*)

class dace.frontend.python.astutils.**ExtNodeVisitor**
Bases: ast.NodeVisitor

A *NodeVisitor* subclass that walks the abstract syntax tree. As opposed to *NodeVisitor*, this class is capable of traversing over top-level expressions in bodies in order to discern DaCe statements from others.

generic_visit(*node*)
Called if no explicit visitor function exists for a node.

visit_TopLevel(*node*)

class dace.frontend.python.astutils.**RemoveSubscripts**(*keywords: Set[str]*)
Bases: ast.NodeTransformer

visit_Subscript(*node: _ast.Subscript*)

class dace.frontend.python.astutils.**TaskletFreeSymbolVisitor**(*defined_syms*)
Bases: ast.NodeVisitor

Simple Python AST visitor to find free symbols in a code, not including attributes and function calls.

visit_AnnAssign(*node*)

visit_Attribute(*node*)

visit_Call(*node: _ast.Call*)

visit_Name(*node*)

dace.frontend.python.astutils.**astrange_to_symrange**(*astrange*, *arrays*, *arrname=None*)
Converts an AST range (array, [(start, end, skip)]) to a symbolic math range, using the obtained array sizes and resolved symbols.

dace.frontend.python.astutils.**evalnode**(*node: _ast.AST, gvars: Dict[str, Any]*) → Any
Tries to evaluate an AST node given only global variables. :param node: The AST node/subtree to evaluate. :param gvars: A dictionary mapping names to variables. :return: The result of evaluation, or raises SyntaxError on any

failure to evaluate.

dace.frontend.python.astutils.**function_to_ast**(*f*)
Obtain the source code of a Python function and create an AST. :param f: Python function. :return: A 4-tuple of (AST, function filename, function line-number,

source code as string).

dace.frontend.python.astutils.**negate_expr**(*node*)
Negates an AST expression by adding a *Not* AST node in front of it.

dace.frontend.python.astutils.**rname**(*node*)
Obtains names from different types of AST nodes.

dace.frontend.python.astutils.**slice_to_subscript**(*arrname*, *range*)
Converts a name and subset to a Python AST Subscript object.

dace.frontend.python.astutils.**subscript_to_ast_slice**(*node*, *without_array=False*)
Converts an AST subscript to slice on the form (<name>, [<3-tuples of AST nodes>]). If an ast.Name is passed, returns (name, None), implying the full range. :param node: The AST node to convert. :param without_array: If True, returns only the slice. Otherwise,

returns a 2-tuple of (array, range).

dace.frontend.python.astutils.**subscript_to_ast_slice_recursive**(*node*)
Converts an AST subscript to a slice in a recursive manner into nested subscripts. @see: subscript_to_ast_slice

dace.frontend.python.astutils.**subscript_to_slice**(*node*, *arrays*, *without_array=False*)
Converts an AST subscript to slice on the form (<name>, [<3-tuples of indices>]). If an ast.Name is passed, return (name, None), implying the full range.

dace.frontend.python.astutils.**unparse**(*node*)
Unparses an AST node to a Python string, chomping trailing newline.

dace.frontend.python.decorators module

dace.frontend.python.ndloop module

A single generator that creates an N-dimensional for loop in Python.

dace.frontend.python.ndloop.**NDLoop**(*ndslice*, *internal_function*, **args*, ***kwargs*)
Wrapped generator that calls an internal function in an N-dimensional for-loop in Python. :param ndslice: Slice or list of slices (*slice* objects) to loop over. :param internal_function: Function to call in loop. :param *args: Arguments to *internal_function*. :param **kwargs: Keyword arguments to *internal_function*. :return: N-dimensional loop index generator.

dace.frontend.python.ndloop.**ndrange**(*slice_list: Union[Tuple[slice], slice]*)
Generator that creates an N-dimensional for loop in Python. :param slice_list: Slice or list of slices (as tuples or 'slice's)

to loop over.

Returns N-dimensional loop index generator.

dace.frontend.python.ndloop.**slicetoxrange**(*s*)
Helper function that turns a slice into a range (for iteration).

dace.frontend.python.newast module

class dace.frontend.python.newast.**AddTransientMethods**
Bases: object

A management singleton for methods that add transient data to SDFGs.

static get(*datatype*)
Returns a method.

class dace.frontend.python.newast.**ProgramVisitor**(*name: str, filename: str, line_offset: int, col_offset: int, global_vars: Dict[str, Any], constants: Dict[str, Any], scope_arrays: Dict[str, dace.data.Data], scope_vars: Dict[str, str], map_symbols: Set[Union[str, dace.symbolic.symbol]] = None, other_sdfgs: Dict[str, Union[dace.sdfg.sdfg.SDFG, DaceProgram]] = None, nested: bool = False, tmp_idx: int = 0, strict: Optional[bool] = None*)
Bases: *dace.frontend.python.astutils.ExtNodeVisitor*

A visitor that traverses a data-centric Python program AST and constructs an SDFG.

defined

make_slice(*arrname: str*, *rng: dace.subsets.Range*)

parse_program(*program: _ast.FunctionDef*, *is_tasklet: bool = False*)
Parses a DaCe program or tasklet

Arguments: program {ast.FunctionDef} – DaCe program or tasklet

Keyword Arguments: is_tasklet {bool} – True, if program is tasklet (default: {False})

Returns: Tuple[SDFG, Dict, Dict] – Parsed SDFG, its inputs and outputs

visit(*node: _ast.AST*)
Visit a node.

visit_AnnAssign(*node: _ast.AnnAssign*)

visit_Assign(*node: _ast.Assign*)

visit_AsyncWith(*node*)

visit_Attribute(*node: _ast.Attribute*)

visit_AugAssign(*node: _ast.AugAssign*)

visit_BinOp(*node: _ast.BinOp*)

visit_BoolOp(*node: _ast.BoolOp*)

visit_Break(*node: _ast.Break*)

visit_Call(*node: _ast.Call*)

visit_Compare(*node: _ast.Compare*)

visit_Constant(*node: _ast.Constant*)

visit_Continue(*node: _ast.Continue*)

visit_ExtSlice(*node: _ast.ExtSlice*) → Any

visit_For(*node: _ast.For*)

visit_FunctionDef(*node: _ast.FunctionDef*)

visit_If(*node: _ast.If*)

visit_Index(*node: _ast.Index*) → Any

visit_Lambda(*node: _ast.Lambda*)

visit_List(*node: _ast.List*)

visit_Name(*node: _ast.Name*)

visit_NameConstant(*node: _ast.NameConstant*)

visit_NamedExpr(*node*)

visit_Num(*node: _ast.Num*)

visit_Return(*node: _ast.Return*)

visit_Str(*node: _ast.Str*)

visit_Subscript(*node: _ast.Subscript*)

visit_TopLevelExpr(*node: _ast.Expr*)

visit_Tuple(*node: _ast.Tuple*)

visit_UnaryOp(*node: _ast.UnaryOp*)

visit_While(*node: _ast.While*)

visit_With(*node*, *is_async=False*)

exception dace.frontend.python.newast.**SkipCall**
Bases: Exception

Exception used to skip calls to functions that cannot be parsed.

class dace.frontend.python.newast.**TaskletTransformer**(*defined, sdfg: dace.sdfg.sdfg.SDFG, state: dace.sdfg.state.SDFGState, filename: str, lang=None, location: dict = {}, nested: bool = False, scope_arrays: Dict[str, dace.data.Data] = {}, scope_vars: Dict[str, str] = {}, variables: Dict[str, str] = {}, accesses: Dict[Tuple[str, dace.subsets.Subset, str], str] = {}, symbols: Dict[str, dace.symbol] = {}*)
Bases: *dace.frontend.python.astutils.ExtNodeTransformer*

A visitor that traverses a data-centric tasklet, removes memlet annotations and returns input and output memlets.

parse_tasklet(*tasklet_ast: Union[_ast.FunctionDef, _ast.With, _ast.For], name: Optional[str] = None*)
Parses the AST of a tasklet and returns the tasklet node, as well as input and output memlets. :param tasklet_ast: The Tasklet's Python AST to parse. :param name: Optional name to use as prefix for tasklet. :return: 3-tuple of (Tasklet node, input memlets, output memlets). @rtype: Tuple[Tasklet, Dict[str, Memlet], Dict[str, Memlet]]

visit_Name(*node: _ast.Name*)

visit_TopLevelExpr(*node*)

visit_TopLevelStr(*node: _ast.Str*)

dace.frontend.python.newast.**add_indirection_subgraph**(*sdfg: dace.sdfg.sdfg.SDFG, graph: dace.sdfg.state.SDFGState, src: dace.sdfg.nodes.Node, dst: dace.sdfg.nodes.Node, memlet: dace.memlet.Memlet, local_name: str, pvisitor: dace.frontend.python.newast.ProgramVisitor, output: bool = False, with_wcr: bool = False*)
Replaces the specified edge in the specified graph with a subgraph that implements indirection without nested memlet subsets.

dace.frontend.python.newast.**parse_dace_program**(*name: str, preprocessed_ast: _ast.AST, argtypes: Dict[str, dace.data.Data], constants: Dict[str, Any], closure: dace.frontend.python.common.SDFGClosure, strict: Optional[bool] = None, save=True*) → dace.sdfg.sdfg.SDFG

Parses a *@dace.program* function into an SDFG. :param src_ast: The AST of the Python program to parse. :param visitor: A ProgramVisitor object returned from

`preprocess_dace_program`.

Parameters

- **`closure`** – An object that contains the @dace.program closure.
- **`strict`** – If True, strict transformations will be performed.
- **`save`** – If True, saves source mapping data for this SDFG.

Returns A 2-tuple of SDFG and its reduced (used) closure.

`dace.frontend.python.newast.`**`specifies_datatype`**(*func: Callable[[Any, dace.data.Data, Any], Tuple[str, dace.data.Data]], datatype=None*)

`dace.frontend.python.newast.`**`until`**(*val*, *substr*)

Helper function that returns the substring of a string until a certain pattern.

dace.frontend.python.parser module

DaCe Python parsing functionality and entry point to Python frontend.

`class` `dace.frontend.python.parser.`**`DaceProgram`**(*f*, *args*, *kwargs*, *auto_optimize*, *device*, *constant_functions=False*, *method=False*)

Bases: `dace.frontend.python.common.SDFGConvertible`

A data-centric program object, obtained by decorating a function with `@dace.program`.

`closure_resolver`(*constant_args*, *parent_closure=None*)

Returns an SDFGClosure object representing the closure of the object to be converted to an SDFG. :param constant_args: Arguments whose values are already resolved to

compile-time values.

Parameters **`parent_closure`** – The parent SDFGClosure object (used for, e.g., recursion detection).

Returns New SDFG closure object representing the convertible object.

`compile`(**args*, *strict=None*, *save=False*, ***kwargs*)

Convenience function that parses and compiles a DaCe program.

`load_precompiled_sdfg`(*path: str*, **args*, ***kwargs*) → None

Loads an external compiled SDFG object that will be invoked when the function is called. :param path: Path to SDFG build folder (e.g., ".dacecache/program").

Path has to include `program.sdfg` and the binary shared object under the `build` folder.

Parameters

- **`args`** – Optional compile-time arguments.
- **`kwargs`** – Optional compile-time keyword arguments.

load_sdfg(*path: str*, **args*, ***kwargs*) → None
Loads an external SDFG that will be used when the function is called. :param path: Path to SDFG file. :param args: Optional compile-time arguments. :param kwargs: Optional compile-time keyword arguments.

methodobj

to_sdfg(**args*, *strict=None*, *save=False*, *validate=False*, ***kwargs*) → dace.sdfg.sdfg.SDFG
Parses the DaCe function into an SDFG.

validate = None
Whether to validate on code generation

dace.frontend.python.parser.**infer_symbols_from_datadescriptor**(*sdfg: dace.sdfg.sdfg.SDFG*, *args: Dict[str, Any]*, *exclude: Optional[Set[str]] = None*) → Dict[str, Any]
Infers the values of SDFG symbols (not given as arguments) from the shapes and strides of input arguments (e.g., arrays). :param sdfg: The SDFG that is being called. :param args: A dictionary mapping from current argument names to their

values. This may also include symbols.

Parameters **exclude** – An optional set of symbols to ignore on inference.

Returns A dictionary mapping from symbol names that are not in `args` to their inferred values.

Raises **ValueError** – If symbol values are ambiguous.

dace.frontend.python.simulator module

dace.frontend.python.wrappers module

Types and wrappers used in DaCe's Python frontend.

dace.frontend.python.wrappers.**define_local**(*dimensions*, *dtype=float*)
Defines a transient array in a DaCe program.

dace.frontend.python.wrappers.**define_local_scalar**(*dtype=float*)
Defines a transient scalar (array of size 1) in a DaCe program.

dace.frontend.python.wrappers.**define_stream**(*dtype=float*, *buffer_size=1*)
Defines a local stream in a DaCe program.

dace.frontend.python.wrappers.**define_streamarray**(*dimensions*, *dtype=float*, *buffer_size=1*)
Defines a local stream array in a DaCe program.

dace.frontend.python.wrappers.**ndarray**(*shape*, *dtype=<class 'numpy.float64'>*, **args*, ***kwargs*)
Returns a numpy ndarray where all symbols have been evaluated to numbers and types are converted to numpy types.

dace.frontend.python.wrappers.**scalar**(*dtype=float*)
Convenience function that defines a scalar (array of size 1).

class dace.frontend.python.wrappers.**stream_array**(*dtype*, *shape*)
Bases: typing.Generic

Stream array object in Python.

shape

Module contents

dace.frontend.tensorflow package

Submodules

dace.frontend.tensorflow.tensorflow module

dace.frontend.tensorflow.winograd module

dace.frontend.tensorflow.winograd.**add_cublas_cusolver**(*sdfg: dace.sdfg.sdfg.SDFG*)
Add CUBLAS and CUSOLVER handles to the SDFG.

dace.frontend.tensorflow.winograd.**mm**(*state*, *A_node*, *B_node*, *C_node*, *A_mode: str = 'N'*, *B_mode: str = 'N'*, *label: str = None*, *A_subset=None*, *B_subset=None*, *C_subset=None*, *A_memlet=None*, *B_memlet=None*, *C_memlet=None*, *map_entry=None*, *map_exit=None*, *shadow_a=False*, *shadow_b=False*, *buffer_a=False*, *buffer_c=False*)

dace.frontend.tensorflow.winograd.**mm_small**(*state*, *A_node*, *B_node*, *C_node*, *A_subset=None*, *B_subset=None*, *C_subset=None*, *A_memlet=None*, *B_memlet=None*, *C_memlet=None*, *map_entry=None*, *map_exit=None*, *A_direct=True*, *B_direct=True*)

dace.frontend.tensorflow.winograd.**printer**(**inp*)

dace.frontend.tensorflow.winograd.**string_builder**(*string*)
To match DaCe variable naming conventions, replaces all undesired characters with "_".

dace.frontend.tensorflow.winograd.**winograd_convolution**(*dace_session*, *tf_node*)

Module contents

Submodules

dace.frontend.operations module

dace.frontend.operations.**detect_reduction_type**(*wcr_str*, *openmp=False*)
Inspects a lambda function and tries to determine if it's one of the built-in reductions that frameworks such as MPI can provide.

Parameters

- **wcr_str** – A Python string representation of the lambda function.

- **openmp** – Detect additional OpenMP reduction types.

Returns dtypes.ReductionType if detected, dtypes.ReductionType.Custom if not detected, or None if no reduction is found.

dace.frontend.operations.**elementwise**(*func*, *in_array*, *out_array=None*)
Applies a function to each element of the array :param in_array: array to apply to. :param out_array: output array to write the result to. If *None*, a new array will be returned :param func: lambda function to apply to each element. :return: new array with the lambda applied to each element

dace.frontend.operations.**is_op_associative**(*wcr_str*)
Inspects a custom lambda function and tries to determine whether it is symbolically associative (disregarding data type). :param wcr_str: A string in Python representing a lambda function. :return: True if associative, False if not, None if cannot be

determined.

dace.frontend.operations.**is_op_commutative**(*wcr_str*)
Inspects a custom lambda function and tries to determine whether it is symbolically commutative (disregarding data type). :param wcr_str: A string in Python representing a lambda function. :return: True if commutative, False if not, None if cannot be

determined.

dace.frontend.operations.**reduce**(*op*, *in_array*, *out_array=None*, *axis=None*, *identity=None*)
Reduces an array according to a binary operation *op*, starting with initial value *identity*, over the given axis (or axes if axis is a list), to *out_array*.

Requires *out_array* with *len(axis)* dimensions less than *in_array*, or a scalar if *axis* is None.

Parameters

- **op** – binary operation to use for reduction.
- **in_array** – array to reduce.
- **out_array** – output array to write the result to. If *None*, a new array will be returned.
- **axis** – the axis or axes to reduce over. If *None*, all axes will be reduced.
- **identity** – intial value for the reduction. If *None*, uses value stored in output.

Returns *None* if out_array is given, or the newly created *out_array* if *out_array* is *None*.

dace.frontend.operations.**timethis**(*sdfg*, *title*, *flop_count*, *f*, **args*, ***kwargs*)
Runs a function multiple (*DACE_treps*) times, logs the running times to a file, and prints the median time (with FLOPs if given). :param sdfg: The SDFG belonging to the measurement. :param title: A title of the measurement. :param flop_count: Number of floating point operations in *program*.

If greater than zero, produces a median FLOPS report.

Parameters

- **f** – The function to measure.
- **args** – Arguments to invoke the function with.
- **kwargs** – Keyword arguments to invoke the function with.

Returns Latest return value of the function.

Module contents

dace.graph package

Submodules

dace.graph.graph module

dace.graph.nodes module

Module contents

dace.sdfg package

Submodules

dace.sdfg.propagation module

Functionality relating to Memlet propagation (deducing external memlets from internal memory accesses and scope ranges).

class dace.sdfg.propagation.**AffineSMemlet**
Bases: *dace.sdfg.propagation.SeparableMemletPattern*

Separable memlet pattern that matches affine expressions, i.e., of the form *a * {index} + b*.

can_be_applied(*dim_exprs*, *variable_context*, *node_range*, *orig_edges*, *dim_index*, *total_dims*)

propagate(*array*, *dim_exprs*, *node_range*)

class dace.sdfg.propagation.**ConstantRangeMemlet**
Bases: *dace.sdfg.propagation.MemletPattern*

Memlet pattern that matches arbitrary expressions with constant range.

can_be_applied(*expressions*, *variable_context*, *node_range*, *orig_edges*)

propagate(*array*, *expressions*, *node_range*)

class dace.sdfg.propagation.**ConstantSMemlet**
Bases: *dace.sdfg.propagation.SeparableMemletPattern*

Separable memlet pattern that matches constant (i.e., unrelated to current scope) expressions.

can_be_applied(*dim_exprs*, *variable_context*, *node_range*, *orig_edges*, *dim_index*, *total_dims*)

propagate(*array*, *dim_exprs*, *node_range*)

class dace.sdfg.propagation.**GenericSMemlet**
Bases: *dace.sdfg.propagation.SeparableMemletPattern*

Separable memlet pattern that detects any expression, and propagates interval bounds. Used as a last resort.

can_be_applied(*dim_exprs*, *variable_context*, *node_range*, *orig_edges*, *dim_index*, *total_dims*)

propagate(*array*, *dim_exprs*, *node_range*)

class dace.sdfg.propagation.**MemletPattern**
Bases: object

A pattern match on a memlet subset that can be used for propagation.

can_be_applied(*expressions*, *variable_context*, *node_range*, *orig_edges*)

extensions()

propagate(*array*, *expressions*, *node_range*)

register(***kwargs*)

unregister()

class dace.sdfg.propagation.**ModuloSMemlet**
Bases: *dace.sdfg.propagation.SeparableMemletPattern*

Separable memlet pattern that matches modulo expressions, i.e., of the form *f(x) % N*.

Acts as a meta-pattern: Finds the underlying pattern for *f(x)*.

can_be_applied(*dim_exprs*, *variable_context*, *node_range*, *orig_edges*, *dim_index*, *total_dims*)

propagate(*array*, *dim_exprs*, *node_range*)

class dace.sdfg.propagation.**SeparableMemlet**
Bases: *dace.sdfg.propagation.MemletPattern*

Meta-memlet pattern that applies all separable memlet patterns.

can_be_applied(*expressions*, *variable_context*, *node_range*, *orig_edges*)

propagate(*array*, *expressions*, *node_range*)

class dace.sdfg.propagation.**SeparableMemletPattern**
Bases: object

Memlet pattern that can be applied to each of the dimensions separately.

can_be_applied(*dim_exprs*, *variable_context*, *node_range*, *orig_edges*, *dim_index*, *total_dims*)

extensions()

propagate(*array*, *dim_exprs*, *node_range*)

register(***kwargs*)

unregister()

dace.sdfg.propagation.**propagate_memlet**(*dfg_state*, *memlet: dace.memlet.Memlet*, *scope_node: dace.sdfg.nodes.EntryNode*, *union_inner_edges: bool*, *arr=None*, *connector=None*)

Tries to propagate a memlet through a scope (computes the image of the memlet function applied on an integer set of, e.g., a map range) and returns a new memlet object. :param dfg_state: An SDFGState object representing the graph. :param memlet: The memlet adjacent to the scope node from the inside. :param scope_node: A scope entry or exit node. :param union_inner_edges: True if the propagation should take other

neighboring internal memlets within the same scope into account.

dace.sdfg.propagation.**propagate_memlets_nested_sdfg**(*parent_sdfg*, *parent_state*, *nsdfg_node*)

Propagate memlets out of a nested sdfg.

Parameters

- **parent_sdfg** – The parent SDFG this nested SDFG is in.
- **parent_state** – The state containing this nested SDFG.

- **nsdfg_node** – The NSDFG node containing this nested SDFG.

Note This operates in-place on the parent SDFG.

dace.sdfg.propagation.**propagate_memlets_scope**(*sdfg*, *state*, *scopes*)
Propagate memlets from the given scopes outwards. :param sdfg: The SDFG in which the scopes reside. :param state: The SDFG state in which the scopes reside. :param scopes: The ScopeTree object or a list thereof to start from. :note: This operation is performed in-place on the given SDFG.

dace.sdfg.propagation.**propagate_memlets_sdfg**(*sdfg*)
Propagates memlets throughout an entire given SDFG. :note: This is an in-place operation on the SDFG.

dace.sdfg.propagation.**propagate_memlets_state**(*sdfg*, *state*)
Propagates memlets throughout one SDFG state. :param sdfg: The SDFG in which the state is situated. :param state: The state to propagate in. :note: This is an in-place operation on the SDFG state.

dace.sdfg.propagation.**propagate_states**(*sdfg*) → None
Annotate the states of an SDFG with the number of executions.

Algorithm: 1. Clean up the state machine by splitting condition and assignment edges

into separate edes with a dummy state in between.

2. Detect and annotate any for-loop constructs with their corresponding loop variable ranges.
3. Start traversing the state machine from the start state (start state gets executed once by default). At every state, check the following:
   a) The state was already visited -> in this case it can either be the guard of a loop we're returning to - in which case the number of executions is additively combined - or it is a state that can be reached through multiple paths (e.g. if/else branches), in which case the number of executions is equal to the maximum number of executions for each incoming path (in case this fully merges a previously branched out tree again, the number of executions isn't dynamic anymore). In both cases we override the calculated number of executions if we're propagating dynamic unbounded. This DFS traversal is complete and we continue with the next unvisited state.
   b) We're propagating dynamic unbounded -> this overrides every calculated number of executions, so this gets unconditionally propagated to all child states.
   c) **None of the above, the next regular traversal step is executed:**
      3.1: If there is no further outgoing edge, this DFS traversal is done and we continue with the next unvisited state.
      3.2: If there is one outgoing edge, we continue propagating the same number of executions to the child state. If the transition to the child state is conditional, the current state might be an implicit exit state, in which case we mark the next state as dynamic to signal that it's an upper bound.
      3.3: If there is more than one outgoing edge we:
         3.3.1: Check if it's an annotated loop guard with a range. If so, we calculate the number of executions for the loop and propagate this down the loop.
         3.3.2: Check if it's a loop that hasn't been unannotated, which means it's unbounded. In this case we propagate dynamic unbounded down the loop.
         3.3.3: Otherwise this must be a conditional branch, so this state's number of executions is given to all child states as an upper bound.
4. The traversal ends when all reachable states have been visited at least once.

Parameters **sdfg** – The SDFG to annotate.

Note This operates on the SDFG in-place.

dace.sdfg.propagation.**propagate_subset**(*memlets: List[dace.memlet.Memlet], arr: dace.data.Data, params: List[str], rng: dace.subsets.Subset, defined_variables: Set[Union[sympy.core.basic.Basic, dace.symbolic.SymExpr]] = None, use_dst: bool = False*) → dace.memlet.Memlet

Tries to propagate a list of memlets through a range (computes the image of the memlet function applied on an integer set of, e.g., a map range) and returns a new memlet object. :param memlets: The memlets to propagate. :param arr: Array descriptor for memlet (used for obtaining extents). :param params: A list of variable names. :param rng: A subset with dimensionality len(params) that contains the

range to propagate with.

Parameters

- **defined_variables** – A set of symbols defined that will remain the same throughout propagation. If None, assumes that all symbols outside of *params* have been defined.
- **use_dst** – Whether to propagate the memlets' dst subset or use the src instead, depending on propagation direction.

Returns Memlet with propagated subset and volume.

dace.sdfg.propagation.**reset_state_annotations**(*sdfg*)

Resets the state (loop-related) annotations of an SDFG. :note: This operation is shallow (does not go into nested SDFGs).

dace.sdfg.scope module

class dace.sdfg.scope.**ScopeSubgraphView**(*graph*, *subgraph_nodes*, *entry_node*)

Bases: dace.sdfg.state.StateSubgraphView

An extension to SubgraphView that enables the creation of scope dictionaries in subgraphs and free symbols.

parent

top_level_transients()

Iterate over top-level transients of this subgraph.

class dace.sdfg.scope.**ScopeTree**(*entrynode: dace.sdfg.nodes.EntryNode, exitnode: dace.sdfg.nodes.ExitNode*)

Bases: object

A class defining a scope, its parent and children scopes, and scope entry/exit nodes.

dace.sdfg.scope.**common_parent_scope**(*sdict: Dict[dace.sdfg.nodes.Node, List[dace.sdfg.nodes.Node]], scope_a: dace.sdfg.nodes.Node, scope_b: dace.sdfg.nodes.Node*) → dace.sdfg.nodes.Node

Finds a common parent scope for both input scopes, or None if the scopes are in different connected components. :param sdict: Scope parent dictionary. :param scope_a: First scope. :param scope_b: Second scope. :return: Scope node or None for top-level scope.

dace.sdfg.scope.**devicelevel_block_size**(*sdfg: dace.sdfg.SDFG, state: dace.sdfg.SDFGState, node: dace.sdfg.nodes.Node*) → Tuple[dace.symbolic.SymExpr]

Returns the current thread-block size if the given node is enclosed in a GPU kernel, or None otherwise. :param

sdfg: The SDFG in which the node resides. :param state: The SDFG state in which the node resides. :param node: The node in question :return: A tuple of sizes or None if the node is not in device-level

code.

dace.sdfg.scope.**is_devicelevel_fpga**(*sdfg: dace.sdfg.SDFG*, *state: dace.sdfg.SDFGState*, *node: dace.sdfg.nodes.Node*) → bool

Tests whether a node in an SDFG is contained within FPGA device-level code. :param sdfg: The SDFG in which the node resides. :param state: The SDFG state in which the node resides. :param node: The node in question :return: True if node is in device-level code, False otherwise.

dace.sdfg.scope.**is_devicelevel_gpu**(*sdfg: dace.sdfg.SDFG*, *state: dace.sdfg.SDFGState*, *node: dace.sdfg.nodes.Node*, *with_gpu_default: bool = False*) → bool

Tests whether a node in an SDFG is contained within GPU device-level code. :param sdfg: The SDFG in which the node resides. :param state: The SDFG state in which the node resides. :param node: The node in question :return: True if node is in device-level code, False otherwise.

dace.sdfg.scope.**is_in_scope**(*sdfg: dace.sdfg.SDFG, state: dace.sdfg.SDFGState, node: dace.sdfg.nodes.Node, schedules: List[dace.dtypes.ScheduleType]*) → bool

Tests whether a node in an SDFG is contained within a certain set of scope schedules. :param sdfg: The SDFG in which the node resides. :param state: The SDFG state in which the node resides. :param node: The node in question :return: True if node is in device-level code, False otherwise.

dace.sdfg.scope.**scope_contains_scope**(*sdict: Dict[dace.sdfg.nodes.Node, List[dace.sdfg.nodes.Node]], node: dace.sdfg.nodes.Node, other_node: dace.sdfg.nodes.Node*) → bool

Returns true iff scope of *node* contains the scope of *other_node*.

dace.sdfg.sdfg module

class dace.sdfg.sdfg.**InterstateEdge**(**args*, ***kwargs*)

Bases: object

An SDFG state machine edge. These edges can contain a condition (which may include data accesses for data-dependent decisions) and zero or more assignments of values to inter-state variables (e.g., loop iterates).

assignments
Assignments to perform upon transition (e.g., 'x=x+1; y = 0')

condition
Transition condition

condition_sympy()

free_symbols
Returns a set of symbols used in this edge's properties.

static from_json(*json_obj*, *context=None*)

is_unconditional()
Returns True if the state transition is unconditional.

label

new_symbols(*sdfg*, *symbols*) → Dict[str, dace.dtypes.typeclass]
Returns a mapping between symbols defined by this edge (i.e., assignments) to their type.

properties()

replace(*name: str*, *new_name: str*, *replace_keys=True*) → None
Replaces all occurrences of `name` with `new_name`. :param name: The source name. :param new_name: The replacement name. :param replace_keys: If False, skips replacing assignment keys.

to_json(*parent=None*)

class dace.sdfg.sdfg.**SDFG**(**args*, ***kwargs*)
Bases: dace.sdfg.graph.OrderedDiGraph

The main intermediate representation of code in DaCe.

A Stateful DataFlow multiGraph (SDFG) is a directed graph of directed acyclic multigraphs (i.e., where two nodes can be connected by more than one edge). The top-level directed graph represents a state machine, where edges can contain state transition conditions and assignments (see the *InterstateEdge* class documentation). The nested acyclic multigraphs represent dataflow, where nodes may represent data regions in memory, tasklets, or parametric graph scopes (see *dace.sdfg.nodes* for a full list of available node types); edges in the multigraph represent data movement using memlets, as described in the *Memlet* class documentation.

add_array(*name: str*, *shape*, *dtype*, *storage=<StorageType.Default: 1>*, *location=None*, *transient=False*, *strides=None*, *offset=None*, *lifetime=<AllocationLifetime.Scope: 1>*, *debuginfo=None*, *allow_conflicts=False*, *total_size=None*, *find_new_name=False*, *alignment=0*, *may_alias=False*) → Tuple[str, dace.data.Array]
Adds an array to the SDFG data descriptor store.

add_constant(*name: str*, *value: Any*, *dtype: dace.data.Data = None*)
Adds/updates a new compile-time constant to this SDFG. A constant may either be a scalar or a numpy ndarray thereof. :param name: The name of the constant. :param value: The constant value. :param dtype: Optional data type of the symbol, or None to deduce

automatically.

add_datadesc(*name: str*, *datadesc: dace.data.Data*, *find_new_name=False*) → str
Adds an existing data descriptor to the SDFG array store. :param name: Name to use. :param datadesc: Data descriptor to add. :param find_new_name: If True and data descriptor with this name

exists, finds a new name to add.

Returns Name of the new data descriptor

add_edge(*u*, *v*, *edge*)
Adds a new edge to the SDFG. Must be an InterstateEdge or a subclass thereof. :param u: Source node. :param v: Destination node. :param edge: The edge to add.

add_loop(*before_state*, *loop_state*, *after_state*, *loop_var: str*, *initialize_expr: str*, *condition_expr: str*, *increment_expr: str*, *loop_end_state=None*)
Helper function that adds a looping state machine around a given state (or sequence of states). :param before_state: The state after which the loop should

begin, or None if the loop is the first state (creates an empty state).

Parameters

- **loop_state** – The state that begins the loop. See also *loop_end_state* if the loop is multi-state.
- **after_state** – The state that should be invoked after the loop ends, or None if the program should terminate (creates an empty state).
- **loop_var** – A name of an inter-state variable to use for the loop. If None, *initialize_expr* and *increment_expr* must be None.

- **initialize_expr** – A string expression that is assigned to *loop_var* before the loop begins. If None, does not define an expression.
- **condition_expr** – A string condition that occurs every loop iteration. If None, loops forever (undefined behavior).
- **increment_expr** – A string expression that is assigned to *loop_var* after every loop iteration.

  If None, does not define an expression.
- **loop_end_state** – If the loop wraps multiple states, the state where the loop iteration ends. If None, sets the end state to *loop_state* as well.

Returns A 3-tuple of (*before_state*, generated loop guard state, *after_state*).

add_node(*node*, *is_start_state=False*)
Adds a new node to the SDFG. Must be an SDFGState or a subclass thereof. :param node: The node to add. :param is_start_state: If True, sets this node as the starting

state.

add_scalar(*name: str*, *dtype*, *storage=<StorageType.Default: 1>*, *transient=False*, *lifetime=<AllocationLifetime.Scope: 1>*, *debuginfo=None*, *find_new_name=False*) → Tuple[str, dace.data.Scalar]
Adds a scalar to the SDFG data descriptor store.

add_state(*label=None*, *is_start_state=False*) → dace.sdfg.state.SDFGState
Adds a new SDFG state to this graph and returns it. :param label: State label. :param is_start_state: If True, resets SDFG starting state to this

state.

Returns A new SDFGState object.

add_state_after(*state: dace.sdfg.state.SDFGState*, *label=None*, *is_start_state=False*) → dace.sdfg.state.SDFGState
Adds a new SDFG state after an existing state, reconnecting it to the successors instead. :param state: The state to append the new state after. :param label: State label. :param is_start_state: If True, resets SDFG starting state to this

state.

Returns A new SDFGState object.

add_state_before(*state: dace.sdfg.state.SDFGState*, *label=None*, *is_start_state=False*) → dace.sdfg.state.SDFGState
Adds a new SDFG state before an existing state, reconnecting predecessors to it instead. :param state: The state to prepend the new state before. :param label: State label. :param is_start_state: If True, resets SDFG starting state to this

state.

Returns A new SDFGState object.

add_stream(*name: str*, *dtype*, *buffer_size=1*, *shape=(1,)*, *storage=<StorageType.Default: 1>*, *transient=False*, *offset=None*, *lifetime=<AllocationLifetime.Scope: 1>*, *debuginfo=None*, *find_new_name=False*) → Tuple[str, dace.data.Stream]
Adds a stream to the SDFG data descriptor store.

add_symbol(*name*, *stype*)
Adds a symbol to the SDFG. :param name: Symbol name. :param stype: Symbol type.

add_temp_transient(*shape*, *dtype*, *storage=<StorageType.Default: 1>*, *location=None*, *strides=None*, *offset=None*, *lifetime=<AllocationLifetime.Scope: 1>*, *debuginfo=None*, *allow_conflicts=False*, *total_size=None*, *alignment=0*, *may_alias=False*)
Convenience function to add a transient array with a temporary name to the data descriptor store.

add_temp_transient_like(*desc: dace.data.Array*, *dtype=None*, *debuginfo=None*)
Convenience function to add a transient array with a temporary name to the data descriptor store.

add_transient(*name*, *shape*, *dtype*, *storage=<StorageType.Default: 1>*, *location=None*, *strides=None*, *offset=None*, *lifetime=<AllocationLifetime.Scope: 1>*, *debuginfo=None*, *allow_conflicts=False*, *total_size=None*, *find_new_name=False*, *alignment=0*, *may_alias=False*) → Tuple[str, dace.data.Array]
Convenience function to add a transient array to the data descriptor store.

add_view(*name: str*, *shape*, *dtype*, *storage=<StorageType.Default: 1>*, *strides=None*, *offset=None*, *debuginfo=None*, *allow_conflicts=False*, *total_size=None*, *find_new_name=False*, *alignment=0*, *may_alias=False*) → Tuple[str, dace.data.View]
Adds a view to the SDFG data descriptor store.

all_edges_recursive()
Iterate over all edges in this SDFG, including state edges, inter-state edges, and recursively edges within nested SDFGs, returning tuples on the form (edge, parent), where the parent is either the SDFG (for states) or a DFG (nodes).

all_nodes_recursive() → Iterator[Tuple[dace.sdfg.nodes.Node, Union[dace.sdfg.sdfg.SDFG, dace.sdfg.state.SDFGState]]]
Iterate over all nodes in this SDFG, including states, nodes in states, and recursive states and nodes within nested SDFGs, returning tuples on the form (node, parent), where the parent is either the SDFG (for states) or a DFG (nodes).

all_sdfgs_recursive()
Iterate over this and all nested SDFGs.

append_exit_code(*cpp_code: str*, *location: str = 'frame'*)
Appends C++ code that will be generated in the __dace_exit_* functions on one of the generated code files. :param cpp_code: The code to append. :param location: The file/backend in which to generate the code.

Options are None (all files), "frame", "openmp", "cuda", "xilinx", "intel_fpga", or any code generator name.

append_global_code(*cpp_code: str*, *location: str = 'frame'*)
Appends C++ code that will be generated in a global scope on one of the generated code files. :param cpp_code: The code to set. :param location: The file/backend in which to generate the code.

Options are None (all files), "frame", "openmp", "cuda", "xilinx", "intel_fpga", or any code generator name.

append_init_code(*cpp_code: str*, *location: str = 'frame'*)
Appends C++ code that will be generated in the __dace_init_* functions on one of the generated code files. :param cpp_code: The code to append. :param location: The file/backend in which to generate the code.

Options are None (all files), "frame", "openmp", "cuda", "xilinx", "intel_fpga", or any code generator name.

append_transformation(*transformation*)
Appends a transformation to the treansformation history of this SDFG. If this is the first transformation

being applied, it also saves the initial state of the SDFG to return to and play back the history. :param transformation: The transformation to append.

apply_fpga_transformations(*states=None*, *validate=True*, *validate_all=False*, *strict=True*)
Applies a series of transformations on the SDFG for it to generate FPGA code.

Note This is an in-place operation on the SDFG.

apply_gpu_transformations(*states=None*, *validate=True*, *validate_all=False*, *strict=True*)
Applies a series of transformations on the SDFG for it to generate GPU code. :note: It is recommended to apply redundant array removal transformation after this transformation. Alternatively, you can apply_strict_transformations() after this transformation. :note: This is an in-place operation on the SDFG.

apply_strict_transformations(*validate=True*, *validate_all=False*)
Applies safe transformations (that will surely increase the performance) on the SDFG. For example, this fuses redundant states (safely) and removes redundant arrays.

B{Note:} This is an in-place operation on the SDFG.

apply_transformations(*xforms: Union[Type[CT_co], List[Type[CT_co]]], options: Union[Dict[str, Any], List[Dict[str, Any]], None] = None, validate: bool = True, validate_all: bool = False, strict: bool = False, states: Optional[List[Any]] = None, print_report: Optional[bool] = None*) → int
This function applies a transformation or a sequence thereof consecutively. Operates in-place. :param xforms: A Transformation class or a sequence. :param options: An optional dictionary (or sequence of dictionaries)

to modify transformation parameters.

Parameters

- **validate** – If True, validates after all transformations.
- **validate_all** – If True, validates after every transformation.
- **strict** – If True, operates in strict transformation mode.
- **states** – If not None, specifies a subset of states to apply transformations on.
- **print_report** – Whether to show debug prints or not (None if the DaCe config option 'debugprint' should apply)

Returns Number of transformations applied.

Examples:

```
# Applies MapTiling, then MapFusion, followed by
# GPUTransformSDFG, specifying parameters only for the
# first transformation.
sdfg.apply_transformations(
  [MapTiling, MapFusion, GPUTransformSDFG],
  options=[{'tile_size': 16}, {}, {}])
```

apply_transformations_repeated(*xforms: Union[Type[CT_co], List[Type[CT_co]]], options: Union[Dict[str, Any], List[Dict[str, Any]], None] = None, validate: bool = True, validate_all: bool = False, strict: bool = False, states: Optional[List[Any]] = None, print_report: Optional[bool] = None, order_by_transformation: bool = True*) → int
This function repeatedly applies a transformation or a set of (unique) transformations until none can be

found. Operates in-place. :param xforms: A Transformation class or a set thereof. :param options: An optional dictionary (or sequence of dictionaries)

to modify transformation parameters.

Parameters

- **validate** – If True, validates after all transformations.
- **validate_all** – If True, validates after every transformation.
- **strict** – If True, operates in strict transformation mode.
- **states** – If not None, specifies a subset of states to apply transformations on.
- **print_report** – Whether to show debug prints or not (None if the DaCe config option 'debugprint' should apply).
- **order_by_transformation** – Try to apply transformations ordered by class rather than SDFG.

Returns Number of transformations applied.

Examples:

```
# Applies InlineSDFG until no more subgraphs can be inlined
sdfg.apply_transformations_repeated(InlineSDFG)
```

arg_names
Ordered argument names (used for calling conventions).

arglist(*scalars_only=False*) → Dict[str, dace.data.Data]
Returns an ordered dictionary of arguments (names and types) required to invoke this SDFG.

The arguments follow the following order: <sorted data arguments>, <sorted scalar arguments>. Data arguments are all the non-transient data containers in the SDFG; and scalar arguments are all the non-transient scalar data containers and free symbols (see `SDFG.free_symbols`). This structure will create a sorted list of pointers followed by a sorted list of PoDs and structs.

Returns An ordered dictionary of (name, data descriptor type) of all the arguments, sorted as defined here.

argument_typecheck(*args*, *kwargs*, *types_only=False*)
Checks if arguments and keyword arguments match the SDFG types. Raises RuntimeError otherwise.

Raises

- **RuntimeError** – Argument count mismatch.
- **TypeError** – Argument type mismatch.
- **NotImplementedError** – Unsupported argument type.

arrays
Returns a dictionary of data descriptors (*Data* objects) used in this SDFG, with an extra *None* entry for empty memlets.

arrays_recursive()
Iterate over all arrays in this SDFG, including arrays within nested SDFGs. Yields 3-tuples of (sdfg, array name, array).

build_folder
Returns a relative path to the build cache folder for this SDFG.

clear_instrumentation_reports()
Clears the instrumentation report folder of this SDFG.

compile(*output_file=None*, *validate=True*) → dace.codegen.compiler.CompiledSDFG
Compiles a runnable binary from this SDFG. :param output_file: If not None, copies the output library file to

the specified path.

Parameters **validate** – If True, validates the SDFG prior to generating code.

Returns A callable CompiledSDFG object.

constants
A dictionary of compile-time constants defined in this SDFG.

constants_prop
Compile-time constants

data(*dataname: str*)
Looks up a data descriptor from its name, which can be an array, stream, or scalar symbol.

exit_code
Code generated in the *__dace_exit* function.

expand_library_nodes(*recursive=True*)
Recursively expand all unexpanded library nodes in the SDFG, resulting in a "pure" SDFG that the code generator can handle. :param recursive: If True, expands all library nodes recursively,

including library nodes that expand to library nodes.

fill_scope_connectors()
Fills missing scope connectors (i.e., "IN_#"/"OUT_#" on entry/exit nodes) according to data on the memlets.

find_new_constant(*name: str*)
Tries to find a new constant name by adding an underscore and a number.

find_new_symbol(*name: str*)
Tries to find a new symbol name by adding an underscore and a number.

find_state(*state_id_or_label*)
Finds a state according to its ID (if integer is provided) or label (if string is provided).

Parameters **state_id_or_label** – State ID (if int) or label (if str).

Returns An SDFGState object.

free_symbols
Returns a set of symbol names that are used by the SDFG, but not defined within it. This property is used to determine the symbolic parameters of the SDFG and verify that `SDFG.symbols` is complete. :note: Assumes that the graph is valid (i.e., without undefined or

overlapping symbols).

static from_file(*filename: str*) → dace.sdfg.sdfg.SDFG
Constructs an SDFG from a file. :param filename: File name to load SDFG from. :return: An SDFG.

classmethod from_json(*json_obj*, *context_info=None*)

generate_code()
Generates code from this SDFG and returns it. :return: A list of *CodeObject* objects containing the generated

code of different files and languages.

get_instrumentation_reports() → List[dace.codegen.instrumentation.report.InstrumentationReport]
Returns a list of instrumentation reports from previous runs of this SDFG. :return: A List of timestamped InstrumentationReport objects.

get_latest_report() → Optional[dace.codegen.instrumentation.report.InstrumentationReport]
Returns an instrumentation report from the latest run of this SDFG, or None if the file does not exist. :return: A timestamped InstrumentationReport object, or None if does

not exist.

global_code
Code generated in a global scope on the output files.

hash_sdfg(*jsondict: Optional[Dict[str, Any]] = None*) → str
Returns a hash of the current SDFG, without considering IDs and attribute names. :param jsondict: If not None, uses given JSON dictionary as input. :return: The hash (in SHA-256 format).

init_code
Code generated in the *__dace_init* function.

input_arrays()
Returns a list of input arrays that need to be fed into the SDFG.

instrument
Measure execution statistics with given method

is_instrumented() → bool
Returns True if the SDFG has performance instrumentation enabled on it or any of its elements.

is_loaded() → bool
Returns True if the SDFG binary is already loaded in the current process.

is_valid() → bool
Returns True if the SDFG is verified correctly (using *validate*).

label
The name of this SDFG.

make_array_memlet(*array: str*)
Convenience method to generate a Memlet that transfers a full array.

Parameters **array** – the name of the array

Returns a Memlet that fully transfers array

name
The name of this SDFG.

openmp_sections
Whether to generate OpenMP sections in code

optimize(*optimizer=None*) → dace.sdfg.sdfg.SDFG
Optimize an SDFG using the CLI or external hooks. :param optimizer: If defines a valid class name, it will be called

during compilation to transform the SDFG as necessary. If None, uses configuration setting.

Returns An SDFG (returns self if optimizer is in place)

orig_sdfg
Object property of type SDFGReferenceProperty

output_arrays()
Returns a list of output arrays that need to be returned from the SDFG.

parent
Returns the parent SDFG state of this SDFG, if exists.

parent_nsdfg_node
Returns the parent NestedSDFG node of this SDFG, if exists.

parent_sdfg
Returns the parent SDFG of this SDFG, if exists.

predecessor_state_transitions(*state*)
Yields paths (lists of edges) that the SDFG can pass through before computing the given state.

predecessor_states(*state*)
Returns a list of unique states that the SDFG can pass through before computing the given state.

prepend_exit_code(*cpp_code: str*, *location: str = 'frame'*)
Prepends C++ code that will be generated in the __dace_exit_* functions on one of the generated code files. :param cpp_code: The code to prepend. :param location: The file/backend in which to generate the code.

Options are None (all files), "frame", "openmp", "cuda", "xilinx", "intel_fpga", or any code generator name.

propagate

properties()

read_and_write_sets() → Tuple[Set[AnyStr], Set[AnyStr]]
Determines what data containers are read and written in this SDFG. Does not include reads to subsets of containers that have previously been written within the same state. :return: A two-tuple of sets of things denoting

({data read}, {data written}).

remove_data(*name*, *validate=True*)
Removes a data descriptor from the SDFG. :param name: The name of the data descriptor to remove. :param validate: If True, verifies that there are no access

nodes that are using this data descriptor prior to removing it.

remove_symbol(*name*)
Removes a symbol from the SDFG. :param name: Symbol name.

replace(*name: str*, *new_name: str*)
Finds and replaces all occurrences of a symbol or array name in SDFG. :param name: Name to find. :param new_name: Name to replace. :raise FileExistsError: If name and new_name already exist as data descriptors or symbols.

replace_dict(*repldict: Dict[str, str]*) → None
Replaces all occurrences of keys in the given dictionary with the mapped values. :param repldict: The replacement dictionary. :param replace_keys: If False, skips replacing assignment keys.

reset_sdfg_list()

save(*filename: str*, *use_pickle=False*, *hash=None*, *exception=None*) → Optional[str]
Save this SDFG to a file. :param filename: File name to save to. :param use_pickle: Use Python pickle as the SDFG format (default:

JSON).

Parameters

- **hash** – By default, saves the hash if SDFG is JSON-serialized. Otherwise, if True, saves the hash along with the SDFG.
- **exception** – If not None, stores error information along with SDFG.

Returns The hash of the SDFG, or None if failed/not requested.

sdfg_id
Returns the unique index of the current SDFG within the current tree of SDFGs (top-level SDFG is 0, nested SDFGs are greater).

sdfg_list

set_exit_code(*cpp_code: str*, *location: str = 'frame'*)
Sets C++ code that will be generated in the __dace_exit_* functions on one of the generated code files. :param cpp_code: The code to set. :param location: The file/backend in which to generate the code.

Options are None (all files), "frame", "openmp", "cuda", "xilinx", "intel_fpga", or any code generator name.

set_global_code(*cpp_code: str*, *location: str = 'frame'*)
Sets C++ code that will be generated in a global scope on one of the generated code files. :param cpp_code: The code to set. :param location: The file/backend in which to generate the code.

Options are None (all files), "frame", "openmp", "cuda", "xilinx", "intel_fpga", or any code generator name.

set_init_code(*cpp_code: str*, *location: str = 'frame'*)
Sets C++ code that will be generated in the __dace_init_* functions on one of the generated code files. :param cpp_code: The code to set. :param location: The file/backend in which to generate the code.

Options are None (all files), "frame", "openmp", "cuda", "xilinx", "intel_fpga", or any code generator name.

set_sourcecode(*code: str*, *lang=None*)
Set the source code of this SDFG (for IDE purposes). :param code: A string of source code. :param lang: A string representing the language of the source code,

for syntax highlighting and completion.

shared_transients(*check_toplevel=True*) → List[str]
Returns a list of transient data that appears in more than one state.

signature(*with_types=True*, *for_call=False*, *with_arrays=True*) → str
Returns a C/C++ signature of this SDFG, used when generating code. :param with_types: If True, includes argument types (can be used

for a function prototype). If False, only include argument names (can be used for function calls).

Parameters

- **for_call** – If True, returns arguments that can be used when calling the SDFG.
- **with_arrays** – If True, includes arrays, otherwise, only symbols and scalars are included.

signature_arglist(*with_types=True*, *for_call=False*, *with_arrays=True*) → List[str]
Returns a list of arguments necessary to call this SDFG, formatted as a list of C definitions. :param with_types: If True, includes argument types in the result. :param for_call: If True, returns arguments that can be used when

calling the SDFG.

Parameters **with_arrays** – If True, includes arrays, otherwise, only symbols and scalars are included.

Returns A list of strings. For example: *['float *A', 'int b']*.

specialize(*symbols: Dict[str, Any]*)
Sets symbolic values in this SDFG to constants. :param symbols: Values to specialize.

start_state
Returns the starting state of this SDFG.

states()
Alias that returns the nodes (states) in this SDFG.

symbols
Global symbols for this SDFG

temp_data_name()
Returns a temporary data descriptor name that can be used in this SDFG.

to_json(*hash=False*)
Serializes this object to JSON format. :return: A string representing the JSON-serialized SDFG.

transformation_hist
Object property of type list

transients()
Returns a dictionary mapping transient data descriptors to their parent scope entry node, or None if top-level (i.e., exists in multiple scopes).

update_sdfg_list(*sdfg_list*)

validate() → None

view(*filename=None*)
View this sdfg in the system's HTML viewer :param filename: the filename to write the HTML to. If *None*, a temporary file will be created.

dace.sdfg.utils module

Various utility functions to create, traverse, and modify SDFGs.

dace.sdfg.utils.**change_edge_dest**(*graph: dace.sdfg.graph.OrderedDiGraph, node_a: Union[dace.sdfg.nodes.Node, dace.sdfg.graph.OrderedMultiDiConnectorGraph], node_b: Union[dace.sdfg.nodes.Node, dace.sdfg.graph.OrderedMultiDiConnectorGraph]*)
Changes the destination of edges from node A to node B.

The function finds all edges in the graph that have node A as their destination. It then creates a new edge for each one found, using the same source nodes and data, but node B as the destination. Afterwards, it deletes the edges found and inserts the new ones into the graph.

Parameters

- **graph** – The graph upon which the edge transformations will be applied.
- **node_a** – The original destination of the edges.
- **node_b** – The new destination of the edges to be transformed.

dace.sdfg.utils.**change_edge_src**(*graph: dace.sdfg.graph.OrderedDiGraph, node_a: Union[dace.sdfg.nodes.Node, dace.sdfg.graph.OrderedMultiDiConnectorGraph], node_b: Union[dace.sdfg.nodes.Node, dace.sdfg.graph.OrderedMultiDiConnectorGraph]*)

Changes the sources of edges from node A to node B.

The function finds all edges in the graph that have node A as their source. It then creates a new edge for each one found, using the same destination nodes and data, but node B as the source. Afterwards, it deletes the edges found and inserts the new ones into the graph.

Parameters

- **graph** – The graph upon which the edge transformations will be applied.
- **node_a** – The original source of the edges to be transformed.
- **node_b** – The new source of the edges to be transformed.

dace.sdfg.utils.**concurrent_subgraphs**(*graph*)

Finds subgraphs of an SDFGState or ScopeSubgraphView that can run concurrently.

dace.sdfg.utils.**consolidate_edges**(*sdfg: dace.sdfg.sdfg.SDFG*, *starting_scope=None*) → int

Union scope-entering memlets relating to the same data node in all states. This effectively reduces the number of connectors and allows more transformations to be performed, at the cost of losing the individual per-tasklet memlets. :param sdfg: The SDFG to consolidate. :return: Number of edges removed.

dace.sdfg.utils.**consolidate_edges_scope**(*state: dace.sdfg.state.SDFGState, scope_node: Union[dace.sdfg.nodes.EntryNode, dace.sdfg.nodes.ExitNode]*) → int

Union scope-entering memlets relating to the same data node in a scope. This effectively reduces the number of connectors and allows more transformations to be performed, at the cost of losing the individual per-tasklet memlets. :param state: The SDFG state in which the scope to consolidate resides. :param scope_node: The scope node whose edges will be consolidated. :return: Number of edges removed.

dace.sdfg.utils.**depth_limited_dfs_iter**(*source*, *depth*)

Produce nodes in a Depth-Limited DFS.

dace.sdfg.utils.**depth_limited_search**(*source*, *depth*)

Return best node and its value using a limited-depth Search (depth- limited DFS).

dace.sdfg.utils.**dfs_conditional**(*G*, *sources=None*, *condition=None*)

Produce nodes in a depth-first ordering.

Parameters

- **G** – An input DiGraph (assumed acyclic).
- **sources** – (optional) node or list of nodes that specify starting point(s) for depth-first search and return edges in the component reachable from source.

Returns A generator of edges in the lastvisit depth-first-search.

@note: Based on http://www.ics.uci.edu/~eppstein/PADS/DFS.py by D. Eppstein, July 2004.

@note: If a source is not specified then a source is chosen arbitrarily and repeatedly until all components in the graph are searched.

dace.sdfg.utils.**dfs_topological_sort**(*G*, *sources=None*, *condition=None*)

Produce nodes in a depth-first topological ordering.

The function produces nodes in a depth-first topological ordering (DFS to make sure maps are visited properly), with the condition that each node visited had all its predecessors visited. Applies for DAGs only, but works on any directed graph.

Parameters

- **G** – An input DiGraph (assumed acyclic).
- **sources** – (optional) node or list of nodes that specify starting point(s) for depth-first search and return edges in the component reachable from source.

Returns A generator of nodes in the lastvisit depth-first-search.

@note: Based on http://www.ics.uci.edu/~eppstein/PADS/DFS.py by D. Eppstein, July 2004.

@note: If a source is not specified then a source is chosen arbitrarily and repeatedly until all components in the graph are searched.

dace.sdfg.utils.**dynamic_map_inputs**(*state: dace.sdfg.state.SDFGState*, *map_entry: dace.sdfg.nodes.MapEntry*) → List[dace.sdfg.graph.MultiConnectorEdge]

For a given map entry node, returns a list of dynamic-range input edges. :param state: The state in which the map entry node resides. :param map_entry: The given node. :return: A list of edges in state whose destination is map entry and denote

dynamic-range input memlets.

dace.sdfg.utils.**find_input_arraynode**(*graph*, *edge*)

dace.sdfg.utils.**find_output_arraynode**(*graph*, *edge*)

dace.sdfg.utils.**find_sink_nodes**(*graph*)

Finds the sink nodes of a graph.

The function finds the sink nodes of a graph, i.e. the nodes with zero out-degree.

Parameters **graph** – The graph whose sink nodes are being searched for.

Returns A list of the sink nodes found.

dace.sdfg.utils.**find_source_nodes**(*graph*)

Finds the source nodes of a graph.

The function finds the source nodes of a graph, i.e. the nodes with zero in-degree.

Parameters **graph** – The graph whose source nodes are being searched for.

Returns A list of the source nodes found.

dace.sdfg.utils.**fuse_states**(*sdfg: dace.sdfg.sdfg.SDFG*, *strict: bool = True*, *progress: bool = False*) → int

Fuses all possible states of an SDFG (and all sub-SDFGs) using an optimized routine that uses the structure of the StateFusion transformation. :param sdfg: The SDFG to transform. :param strict: If True (default), operates in strict mode. :param progress: If True, prints out a progress bar of fusion (may be

inaccurate, requires `tqdm`)

Returns The total number of states fused.

dace.sdfg.utils.**get_last_view_node**(*state: dace.sdfg.state.SDFGState*, *view: dace.sdfg.nodes.AccessNode*) → dace.sdfg.nodes.AccessNode

Given a view access node, returns the last viewed access node if existent, else None

dace.sdfg.utils.**get_next_nonempty_states**(*sdfg: dace.sdfg.sdfg.SDFG*, *state: dace.sdfg.state.SDFGState*) → Set[dace.sdfg.state.SDFGState]

From the given state, return the next set of states that are reachable in the SDFG, skipping empty states. Traversal stops at the non-empty state. This function is used to determine whether synchronization should happen at the

end of a GPU state. :param sdfg: The SDFG that contains the state. :param state: The state to start from. :return: A set of reachable non-empty states.

dace.sdfg.utils.**get_view_edge**(*state: dace.sdfg.state.SDFGState*, *view: dace.sdfg.nodes.AccessNode*) → dace.sdfg.graph.MultiConnectorEdge[dace.memlet.Memlet][dace.memlet.Memlet]

Given a view access node, returns the incoming/outgoing edge which points to the viewed access node. See the ruleset in the documentation of `dace.data.View`.

Parameters

- **state** – The state in which the view resides.
- **view** – The view access node.

Returns An edge pointing to the viewed data or None if view is invalid.

See `dace.data.View`

dace.sdfg.utils.**get_view_node**(*state: dace.sdfg.state.SDFGState*, *view: dace.sdfg.nodes.AccessNode*) → dace.sdfg.nodes.AccessNode

Given a view access node, returns the viewed access node if existent, else None

dace.sdfg.utils.**has_dynamic_map_inputs**(*state: dace.sdfg.state.SDFGState*, *map_entry: dace.sdfg.nodes.MapEntry*) → bool

Returns True if a map entry node has dynamic-range inputs. :param state: The state in which the map entry node resides. :param map_entry: The given node. :return: True if there are dynamic-range input memlets, False otherwise.

dace.sdfg.utils.**inline_sdfgs**(*sdfg: dace.sdfg.sdfg.SDFG*, *strict: bool = True*, *progress: bool = False*) → int

Inlines all possible nested SDFGs (or sub-SDFGs) using an optimized routine that uses the structure of the SDFG hierarchy. :param sdfg: The SDFG to transform. :param strict: If True (default), operates in strict mode. :param progress: If True, prints out a progress bar of inlining (may be

inaccurate, requires `tqdm`)

Returns The total number of SDFGs inlined.

dace.sdfg.utils.**is_array_stream_view**(*sdfg: dace.sdfg.sdfg.SDFG*, *dfg: dace.sdfg.state.SDFGState*, *node: dace.sdfg.nodes.AccessNode*)

Test whether a stream is directly connected to an array.

dace.sdfg.utils.**is_nonfree_sym_dependent**(*node: dace.sdfg.nodes.AccessNode, desc: dace.data.Data, state: dace.sdfg.state.SDFGState, fsymbols: Set[str]*) → bool

Checks whether the Array or View descriptor is non-free symbol dependent. An Array is non-free symbol dependent when its attributes (e.g., shape) depend on non-free symbols. A View is non-free symbol dependent when either its adjacent edges or its viewed node depend on non-free symbols. :param node: the access node to check :param desc: the data descriptor to check :param state: the state that contains the node :param fsymbols: the free symbols to check against

dace.sdfg.utils.**is_parallel**(*state: dace.sdfg.state.SDFGState*, *node: Optional[dace.sdfg.nodes.Node] = None*) → bool

Returns True if a node or state are contained within a parallel section. :param state: The state to test. :param node: An optional node in the state to test. If None, only checks

state.

Returns True if the state or node are located within a map scope that is scheduled to run in parallel, False otherwise.

dace.sdfg.utils.**load_precompiled_sdfg**(*folder: str*)

Loads a pre-compiled SDFG from an output folder (e.g. ".dacecache/program"). Folder must contain a file called "program.sdfg" and a subfolder called "build" with the shared object.

Parameters **folder** – Path to SDFG output folder.

Returns A callable CompiledSDFG object.

dace.sdfg.utils.**local_transients**(*sdfg*, *dfg*, *entry_node*, *include_nested=False*)

Returns transients local to the scope defined by the specified entry node in the dataflow graph. :param entry_node: The entry node that opens the scope. If *None*, the

top-level scope is used.

Parameters **include_nested** – Include transients defined in nested scopes.

dace.sdfg.utils.**merge_maps**(*graph: dace.sdfg.state.SDFGState, outer_map_entry: dace.sdfg.nodes.MapEntry, outer_map_exit: dace.sdfg.nodes.MapExit, inner_map_entry: dace.sdfg.nodes.MapEntry, inner_map_exit: dace.sdfg.nodes.MapExit, param_merge: Callable[[List[dace.symbolic.symbol], List[dace.symbolic.symbol]], List[dace.symbolic.symbol]] = <function <lambda>>, range_merge: Callable[[List[dace.subsets.Subset], List[dace.subsets.Subset]], List[dace.subsets.Subset]] = <function <lambda>>) -> (<class 'dace.sdfg.nodes.MapEntry'>, <class 'dace.sdfg.nodes.MapExit'>*)

Merges two maps (their entries and exits). It is assumed that the operation is valid.

dace.sdfg.utils.**node_path_graph**(**args*)

Generates a path graph passing through the input nodes.

The function generates a graph using as nodes the input arguments. Subsequently, it creates a path passing through all the nodes, in the same order as they were given in the function input.

Parameters ***args** – Variable number of nodes or a list of nodes.

Returns A directed graph based on the input arguments.

@rtype: gr.OrderedDiGraph

dace.sdfg.utils.**remove_edge_and_dangling_path**(*state: dace.sdfg.state.SDFGState*, *edge: dace.sdfg.graph.MultiConnectorEdge*)

Removes an edge and all of its parent edges in a memlet path, cleaning dangling connectors and isolated nodes resulting from the removal. :param state: The state in which the edge exists. :param edge: The edge to remove.

dace.sdfg.utils.**separate_maps**(*state*, *dfg*, *schedule*)

Separates the given ScopeSubgraphView into subgraphs with and without maps of the given schedule type. The function assumes that the given ScopeSubgraph view does not contain any concurrent segments (i.e. pass it through concurrent_subgraphs first). Only top level maps will be accounted for, if the desired schedule occurs in another (undesired) map, it will be ignored.

Returns a list with the subgraph views in order of the original DFG. ScopeSubgraphViews for the parts with maps, StateSubgraphViews for the parts without maps.

dace.sdfg.utils.**trace_nested_access**(*node: dace.sdfg.nodes.AccessNode*, *state: dace.sdfg.state.SDFGState*, *sdfg: dace.sdfg.sdfg.SDFG*) → List[Tuple[dace.sdfg.nodes.AccessNode, dace.sdfg.state.SDFGState, dace.sdfg.sdfg.SDFG]]

Given an AccessNode in a nested SDFG, trace the accessed memory back to the outermost scope in which it is

defined.

Parameters

- **node** – An access node.
- **state** – State in which the access node is located.
- **sdfg** – SDFG in which the access node is located.

Returns A list of scopes ((input_node, output_node), (memlet_read, memlet_write), state, sdfg) in which the given data is accessed, from outermost scope to innermost scope.

dace.sdfg.utils.**unique_node_repr**(*graph: Union[dace.sdfg.state.SDFGState, dace.sdfg.scope.ScopeSubgraphView]*, *node: dace.sdfg.nodes.Node*) → str

Returns unique string representation of the given node, considering its placement into the SDFG graph. Useful for hashing, or building node-based dictionaries. :param graph: the state/subgraph that contains the node :param node: node to represent :return: the unique representation

dace.sdfg.utils.**weakly_connected_component**(*dfg*, *node_in_component: dace.sdfg.nodes.Node*) → dace.sdfg.state.StateSubgraphView

Returns a subgraph of all nodes that form the weakly connected component in *dfg* that contains *node_in_component*.

dace.sdfg.validation module

Exception classes and methods for validation of SDFGs.

exception dace.sdfg.validation.**InvalidSDFGEdgeError**(*message: str*, *sdfg*, *state_id*, *edge_id*)

Bases: *dace.sdfg.validation.InvalidSDFGError*

Exceptions of invalid edges in an SDFG state.

to_json()

exception dace.sdfg.validation.**InvalidSDFGError**(*message: str*, *sdfg*, *state_id*)

Bases: Exception

A class of exceptions thrown when SDFG validation fails.

to_json()

exception dace.sdfg.validation.**InvalidSDFGInterstateEdgeError**(*message: str*, *sdfg*, *edge_id*)

Bases: *dace.sdfg.validation.InvalidSDFGError*

Exceptions of invalid inter-state edges in an SDFG.

to_json()

exception dace.sdfg.validation.**InvalidSDFGNodeError**(*message: str*, *sdfg*, *state_id*, *node_id*)

Bases: *dace.sdfg.validation.InvalidSDFGError*

Exceptions of invalid nodes in an SDFG state.

to_json()

exception dace.sdfg.validation.**NodeNotExpandedError**(*sdfg: dace.sdfg.SDFG*, *state_id: int*, *node_id: int*)

Bases: *dace.sdfg.validation.InvalidSDFGNodeError*

Exception that is raised whenever a library node was not expanded before code generation.

dace.sdfg.validation.**validate**(*graph: dace.sdfg.graph.SubgraphView*)

dace.sdfg.validation.**validate_sdfg**(*sdfg: dace.sdfg.SDFG*)
Verifies the correctness of an SDFG by applying multiple tests. :param sdfg: The SDFG to verify.

Raises an InvalidSDFGError with the erroneous node/edge on failure.

dace.sdfg.validation.**validate_state**(*state: dace.sdfg.SDFGState*, *state_id: int = None*, *sdfg: dace.sdfg.SDFG = None*, *symbols: Dict[str, dace.dtypes.typeclass] = None*)
Verifies the correctness of an SDFG state by applying multiple tests. Raises an InvalidSDFGError with the erroneous node on failure.

Module contents

dace.transformation package

Subpackages

dace.transformation.dataflow package

Submodules

dace.transformation.dataflow.copy_to_device module

Contains classes and functions that implement copying a nested SDFG and its dependencies to a given device.

class dace.transformation.dataflow.copy_to_device.**CopyToDevice**(**args*, ***kwargs*)
Bases: *dace.transformation.transformation.Transformation*

Implements the copy-to-device transformation, which copies a nested SDFG and its dependencies to a given device.

The transformation changes all data storage types of a nested SDFG to the given *storage* property, and creates new arrays and copies around the nested SDFG to that storage.

static annotates_memlets()
Indicates whether the transformation annotates the edges it creates or modifies with the appropriate memlets. This determines whether to apply memlet propagation after the transformation.

apply(*sdfg*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

properties()

storage
Nested SDFG storage

dace.transformation.dataflow.copy_to_device.**change_storage**(*sdfg*, *storage*)

dace.transformation.dataflow.double_buffering module

Contains classes that implement the double buffering pattern.

class dace.transformation.dataflow.double_buffering.**DoubleBuffering**(**args*, ***kwargs*)
Bases: *dace.transformation.transformation.Transformation*

Implements the double buffering pattern, which pipelines reading and processing data by creating a second copy of the memory. In particular, the transformation takes a 1D map and all internal (directly connected) transients, adds an additional dimension of size 2, and turns the map into a for loop that processes and reads the data in a double-buffered manner. Other memlets will not be transformed.

apply(*sdfg: dace.sdfg.sdfg.SDFG*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

dace.transformation.dataflow.gpu_transform module

Contains the GPU Transform Map transformation.

class dace.transformation.dataflow.gpu_transform.**GPUTransformMap**(**args*, ***kwargs*)
Bases: *dace.transformation.transformation.Transformation*

Implements the GPUTransformMap transformation.

Converts a single map to a GPU-scheduled map and creates GPU arrays outside it, generating CPU<->GPU memory copies automatically.

apply(*sdfg*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

fullcopy
Copy whole arrays rather than used subset

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

properties()

register_trans
Make all transients inside GPU maps registers

sequential_innermaps
Make all internal maps Sequential

stdlib = <module 'dace.libraries.standard' from '/home/docs/checkouts/readthedocs.org/

toplevel_trans
Make all GPU transients top-level

dace.transformation.dataflow.gpu_transform_local_storage module

Contains classes and functions that implement the GPU transformation (with local storage).

class dace.transformation.dataflow.gpu_transform_local_storage.**GPUTransformLocalStorage**(*ar **k

Bases: *dace.transformation.transformation.Transformation*

Implements the GPUTransformLocalStorage transformation.

Similar to GPUTransformMap, but takes multiple maps leading from the same data node into account, creating a local storage for each range.

@see: GPUTransformMap

apply(*sdfg*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

fullcopy
Copy whole arrays rather than used subset

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

nested_seq
Makes nested code semantically-equivalent to single-core code,transforming nested maps and memory into sequential and local memory respectively.

properties()

stdlib = <module 'dace.libraries.standard' from '/home/docs/checkouts/readthedocs.org/

dace.transformation.dataflow.gpu_transform_local_storage.**in_path**(*path*, *edge*, *node-type*, *for-ward=True*)

dace.transformation.dataflow.gpu_transform_local_storage.**in_scope**(*graph*, *node*, *parent*)

Returns True if *node* is in the scope of *parent*.

dace.transformation.dataflow.local_storage module

Contains classes that implement transformations relating to streams and transient nodes.

class dace.transformation.dataflow.local_storage.**InLocalStorage**(**args*, ***kwargs*)

Bases: *dace.transformation.dataflow.local_storage.LocalStorage*

Implements the InLocalStorage transformation, which adds a transient data node between two scope entry nodes.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)

Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

properties()

class dace.transformation.dataflow.local_storage.**LocalStorage**(**args*, ***kwargs*)

Bases: *dace.transformation.transformation.Transformation*, abc.ABC

Implements the Local Storage prototype transformation, which adds a transient data node between two nodes.

static annotates_memlets()

Indicates whether the transformation annotates the edges it creates or modifies with the appropriate memlets. This determines whether to apply memlet propagation after the transformation.

apply(*sdfg*)

Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

array

Array to create local storage for (if empty, first available)

create_array
if false, it does not create a new array.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

node_a = <dace.transformation.transformation.PatternNode object>

node_b = <dace.transformation.transformation.PatternNode object>

prefix
Prefix for new data node

properties()

class dace.transformation.dataflow.local_storage.**OutLocalStorage**(**args*, ***kwargs*)
Bases: *dace.transformation.dataflow.local_storage.LocalStorage*

Implements the OutLocalStorage transformation, which adds a transient data node between two scope exit nodes.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

properties()

dace.transformation.dataflow.map_collapse module

Contains classes that implement the map-collapse transformation.

class dace.transformation.dataflow.map_collapse.**MapCollapse**(**args*, ***kwargs*)
Bases: *dace.transformation.transformation.Transformation*

Implements the Map Collapse pattern.

Map-collapse takes two nested maps with M and N dimensions respectively, and collapses them to a single M+N dimensional map.

apply(*sdfg*) → Tuple[dace.sdfg.nodes.MapEntry, dace.sdfg.nodes.MapExit]
Collapses two maps into one. :param sdfg: The SDFG to apply the transformation to. :return: A 2-tuple of the new map entry and exit nodes.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

properties()

dace.transformation.dataflow.map_expansion module

Contains classes that implement the map-expansion transformation.

class dace.transformation.dataflow.map_expansion.**MapExpansion**(**args*, ***kwargs*)
Bases: *dace.transformation.transformation.Transformation*

Implements the map-expansion pattern.

Map-expansion takes an N-dimensional map and expands it to N unidimensional maps.

New edges abide by the following rules:

1. If there are no edges coming from the outside, use empty memlets
2. Edges with IN_* connectors replicate along the maps
3. Edges for dynamic map ranges replicate until reaching range(s)

apply(*sdfg: dace.sdfg.sdfg.SDFG*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph: dace.sdfg.state.SDFGState*, *candidate: Dict[dace.transformation.transformation.PatternNode, int]*, *expr_index: int*, *sdfg: dace.sdfg.sdfg.SDFG*, *strict: bool = False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

map_entry = <dace.transformation.transformation.PatternNode object>

static match_to_str(*graph: dace.sdfg.state.SDFGState*, *candidate: Dict[dace.transformation.transformation.PatternNode, int]*) → str
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

dace.transformation.dataflow.map_fission module

Map Fission transformation.

class dace.transformation.dataflow.map_fission.**MapFission**(**args*, ***kwargs*)
Bases: *dace.transformation.transformation.Transformation*

Implements the MapFission transformation. Map fission refers to subsuming a map scope into its internal subgraph, essentially replicating the map into maps in all of its internal components. This also extends the dimensions of "border" transient arrays (i.e., those between the maps), in order to retain program semantics after fission.

There are two cases that match map fission: 1. A map with an arbitrary subgraph with more than one computational

(i.e., non-access) node. The use of arrays connecting the computational nodes must be limited to the subgraph, and non transient arrays may not be used as "border" arrays.

2. A map with one internal node that is a nested SDFG, in which each state matches the conditions of case (1).

If a map has nested SDFGs in its subgraph, they are not considered in the case (1) above, and MapFission must be invoked again on the maps with the nested SDFGs in question.

static annotates_memlets()
Indicates whether the transformation annotates the edges it creates or modifies with the appropriate memlets. This determines whether to apply memlet propagation after the transformation.

apply(*sdfg: dace.sdfg.sdfg.SDFG*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

dace.transformation.dataflow.map_for_loop module

This module contains classes that implement a map->for loop transformation.

class dace.transformation.dataflow.map_for_loop.**MapToForLoop**(**args*, ***kwargs*)
Bases: *dace.transformation.transformation.Transformation*

Implements the Map to for-loop transformation.

Takes a map and enforces a sequential schedule by transforming it into a state-machine of a for-loop. Creates a nested SDFG, if necessary.

static annotates_memlets()
Indicates whether the transformation annotates the edges it creates or modifies with the appropriate memlets. This determines whether to apply memlet propagation after the transformation.

apply(*sdfg*) → Tuple[dace.sdfg.nodes.NestedSDFG, dace.sdfg.state.SDFGState]
Applies the transformation and returns a tuple with the new nested SDFG node and the main state in the for-loop.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

dace.transformation.dataflow.map_fusion module

This module contains classes that implement the map fusion transformation.

class dace.transformation.dataflow.map_fusion.**MapFusion**(**args*, ***kwargs*)
Bases: *dace.transformation.transformation.Transformation*

Implements the MapFusion transformation. It wil check for all patterns MapExit -> AccessNode -> MapEntry, and based on the following rules, fuse them and remove the transient in between. There are several possibilities of what it does to this transient in between.

Essentially, if there is some other place in the sdfg where it is required, or if it is not a transient, then it will not be removed. In such a case, it will be linked to the MapExit node of the new fused map.

Rules for fusing maps:

0. The map range of the second map should be a permutation of the first map range.
1. Each of the access nodes that are adjacent to the first map exit should have an edge to the second map entry. If it doesn't, then the second map entry should not be reachable from this access node.
2. Any node that has a wcr from the first map exit should not be adjacent to the second map entry.
3. Access pattern for the access nodes in the second map should be the same permutation of the map parameters as the map ranges of the two maps. Alternatively, this access node should not be adjacent to the first map entry.

static annotates_memlets()
Indicates whether the transformation annotates the edges it creates or modifies with the appropriate memlets. This determines whether to apply memlet propagation after the transformation.

apply(*sdfg*)
This method applies the mapfusion transformation. Other than the removal of the second map entry node (SME), and the first map exit (FME) node, it has the following side effects:

1. Any transient adjacent to both FME and SME with degree = 2 will be removed. The tasklets that use/produce it shall be connected directly with a scalar/new transient (if the dataflow is more than a single scalar)
2. If this transient is adjacent to FME and SME and has other uses, it will be adjacent to the new map exit post fusion. Tasklet-> Tasklet edges will ALSO be added as mentioned above.
3. If an access node is adjacent to FME but not SME, it will be adjacent to new map exit post fusion.
4. If an access node is adjacent to SME but not FME, it will be adjacent to the new map entry node post fusion.

array = <dace.transformation.transformation.PatternNode object>

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static find_permutation(*first_map: dace.sdfg.nodes.Map*, *second_map: dace.sdfg.nodes.Map*) → Optional[List[int]]
Find permutation between two map ranges. :param first_map: First map. :param second_map: Second map. :return: None if no such permutation exists, otherwise a list of

indices L such that L[x]'th parameter of second map has the same range as x'th parameter of the first map.

first_map_exit = <dace.transformation.transformation.PatternNode object>

fuse_nodes(*sdfg*, *graph*, *edge*, *new_dst*, *new_dst_conn*, *other_edges=None*)
Fuses two nodes via memlets and possibly transient arrays.

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

second_map_entry = <dace.transformation.transformation.PatternNode object>

dace.transformation.dataflow.map_interchange module

Implements the map interchange transformation.

class dace.transformation.dataflow.map_interchange.**MapInterchange**(**args*, ***kwargs*)
Bases: *dace.transformation.transformation.Transformation*

Implements the map-interchange transformation.

Map-interchange takes two nested maps and interchanges their position.

static annotates_memlets()
Indicates whether the transformation annotates the edges it creates or modifies with the appropriate memlets. This determines whether to apply memlet propagation after the transformation.

apply(*sdfg: dace.sdfg.sdfg.SDFG*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

inner_map_entry = <dace.transformation.transformation.PatternNode object>

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

outer_map_entry = <dace.transformation.transformation.PatternNode object>

properties()

dace.transformation.dataflow.mapreduce module

Contains classes and functions that implement the map-reduce-fusion transformation.

class dace.transformation.dataflow.mapreduce.**MapReduceFusion**(**args*, ***kwargs*)
Bases: *dace.transformation.transformation.Transformation*

Implements the map-reduce-fusion transformation. Fuses a map with an immediately following reduction, where the array between the map and the reduction is not used anywhere else.

apply(*sdfg: dace.sdfg.sdfg.SDFG*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

no_init
If enabled, does not create initialization states for reduce nodes with identity

properties()

stdlib = <module 'dace.libraries.standard' from '/home/docs/checkouts/readthedocs.org/

class dace.transformation.dataflow.mapreduce.**MapWCRFusion**(**args*, ***kwargs*)
Bases: *dace.transformation.transformation.Transformation*

Implements the map expanded-reduce fusion transformation. Fuses a map with an immediately following reduction, where the array between the map and the reduction is not used anywhere else, and the reduction is divided to two maps with a WCR, denoting partial reduction.

apply(*sdfg*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

dace.transformation.dataflow.matrix_product_transpose module

Implements the matrix-matrix product transpose transformation.

class dace.transformation.dataflow.matrix_product_transpose.**MatrixProductTranspose**(**args*, ***kwargs*)
Bases: *dace.transformation.transformation.Transformation*

Implements the matrix-matrix product transpose transformation.

T(A) @ T(B) = T(B @ A)

apply(*sdfg*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

properties()

dace.transformation.dataflow.merge_arrays module

class dace.transformation.dataflow.merge_arrays.**InMergeArrays**(**args*, ***kwargs*)
Bases: *dace.transformation.transformation.Transformation*

Merge duplicate arrays connected to the same scope entry.

apply(*sdfg*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.

- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

class dace.transformation.dataflow.merge_arrays.**MergeSourceSinkArrays**(**args*, ***kwargs*)

Bases: *dace.transformation.transformation.Transformation*

Merge duplicate arrays that are source/sink nodes.

apply(*sdfg*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

class dace.transformation.dataflow.merge_arrays.**OutMergeArrays**(**args*, ***kwargs*)

Bases: *dace.transformation.transformation.Transformation*

Merge duplicate arrays connected to the same scope entry.

apply(*sdfg*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

dace.transformation.dataflow.mpi module

Contains the MPITransformMap transformation.

class dace.transformation.dataflow.mpi.**MPITransformMap**(**args*, ***kwargs*)
Bases: *dace.transformation.transformation.Transformation*

Implements the MPI parallelization pattern.

Takes a map and makes it an MPI-scheduled map, introduces transients that keep locally accessed data.

‘‘‘ Input1 - Output1

/

Input2 — MapEntry – Arbitrary R – MapExit – Output2 / InputN - OutputN

‘‘‘

Nothing in R may access other inputs/outputs that are not defined in R itself and do not go through MapEntry/MapExit Map must be a one-dimensional map for now. The range of the map must be a Range object.

- Add transients for the accessed parts
- The schedule property of Map is set to MPI
- The range of Map is changed to var = startexpr + p * chunksize … startexpr + p + 1 * chunksize where p is the current rank and P is the total number of ranks, and chunksize is defined as (endexpr - startexpr) / P, adding the remaining K iterations to the first K procs.
- For each input InputI, create a new transient transInputI, which has an attribute that specifies that it needs to be filled with (possibly) remote data

- Collect all accesses to InputI within R, assume their convex hull is InputI[rs ... re]
- The transInputI transient will contain InputI[rs ... re]
- Change all accesses to InputI within R to accesses to transInputI

static annotates_memlets()
Indicates whether the transformation annotates the edges it creates or modifies with the appropriate memlets. This determines whether to apply memlet propagation after the transformation.

apply(*sdfg*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

properties()

dace.transformation.dataflow.redundant_array module

Contains classes that implement a redundant array removal transformation.

class dace.transformation.dataflow.redundant_array.**RedundantArray**(**args*, ***kwargs*)
Bases: *dace.transformation.transformation.Transformation*

Implements the redundant array removal transformation, applied when a transient array is copied to and from (to another array), but never used anywhere else.

apply(*sdfg*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

in_array = <dace.transformation.transformation.PatternNode object>

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

out_array = <dace.transformation.transformation.PatternNode object>

class dace.transformation.dataflow.redundant_array.**RedundantReadSlice**(**args*, ***kwargs*)

Bases: *dace.transformation.transformation.Transformation*

Detects patterns of the form Array -> View(Array) and removes the View if it is a slice.

apply(*sdfg*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

in_array = <dace.transformation.transformation.PatternNode object>

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

out_array = <dace.transformation.transformation.PatternNode object>

class dace.transformation.dataflow.redundant_array.**RedundantSecondArray**(**args*, ***kwargs*)

Bases: *dace.transformation.transformation.Transformation*

Implements the redundant array removal transformation, applied when a transient array is copied from and to (from another array), but never used anywhere else. This transformation removes the second array.

apply(*sdfg*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

class dace.transformation.dataflow.redundant_array.**RedundantWriteSlice**(**args*, ***kwargs*)

Bases: *dace.transformation.transformation.Transformation*

Detects patterns of the form View(Array) -> Array and removes the View if it is a slice.

apply(*sdfg*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

in_array = <dace.transformation.transformation.PatternNode object>

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

out_array = <dace.transformation.transformation.PatternNode object>

class dace.transformation.dataflow.redundant_array.**SqueezeViewRemove**(**args*, ***kwargs*)
Bases: *dace.transformation.transformation.Transformation*

apply(*sdfg: dace.sdfg.sdfg.SDFG*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

can_be_applied(*state: dace.sdfg.state.SDFGState*, *candidate*, *expr_index: int*, *sdfg: dace.sdfg.sdfg.SDFG*, *strict: bool = False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

in_array = <dace.transformation.transformation.PatternNode object>

out_array = <dace.transformation.transformation.PatternNode object>

class dace.transformation.dataflow.redundant_array.**UnsqueezeViewRemove**(**args*, ***kwargs*)

Bases: *dace.transformation.transformation.Transformation*

apply(*sdfg: dace.sdfg.sdfg.SDFG*)

Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

can_be_applied(*state: dace.sdfg.state.SDFGState*, *candidate*, *expr_index: int*, *sdfg: dace.sdfg.sdfg.SDFG*, *strict: bool = False*)

Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()

Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

in_array = <dace.transformation.transformation.PatternNode object>

out_array = <dace.transformation.transformation.PatternNode object>

dace.transformation.dataflow.redundant_array.**compose_and_push_back**(*first*, *second*, *dims=None*, *popped=None*)

dace.transformation.dataflow.redundant_array.**find_dims_to_pop**(*a_size*, *b_size*)

dace.transformation.dataflow.redundant_array.**pop_dims**(*subset*, *dims*)

dace.transformation.dataflow.redundant_array_copying module

Contains redundant array removal transformations.

class dace.transformation.dataflow.redundant_array_copying.**RedundantArrayCopying**(**args*, ***kwargs*)

Bases: *dace.transformation.transformation.Transformation*

Implements the redundant array removal transformation. Removes the last access node in pattern A -> B -> A, and the second (if possible)

apply(*sdfg*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

class dace.transformation.dataflow.redundant_array_copying.**RedundantArrayCopying2**(**args*, ***kwargs*)

Bases: *dace.transformation.transformation.Transformation*

Implements the redundant array removal transformation. Removes multiples of array B in pattern A -> B.

apply(*sdfg*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

class dace.transformation.dataflow.redundant_array_copying.**RedundantArrayCopying3**(**args*, ***kwargs*)

Bases: *dace.transformation.transformation.Transformation*

Implements the redundant array removal transformation. Removes multiples of array B in pattern MapEntry -> B.

apply(*sdfg*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

class dace.transformation.dataflow.redundant_array_copying.**RedundantArrayCopyingIn**(**args*, ***kwargs*)

Bases: *dace.transformation.transformation.Transformation*

Implements the redundant array removal transformation. Removes the first and second access nodeds in pattern A -> B -> A

apply(*sdfg*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

dace.transformation.dataflow.stream_transient module

Contains classes that implement transformations relating to streams and transient nodes.

class dace.transformation.dataflow.stream_transient.**AccumulateTransient**(**args*, ***kwargs*)
Bases: *dace.transformation.transformation.Transformation*

Implements the AccumulateTransient transformation, which adds transient stream and data nodes between nested maps that lead to a stream. The transient data nodes then act as a local accumulator.

apply(*sdfg: dace.sdfg.sdfg.SDFG*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

array
Array to create local storage for (if empty, first available)

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.

- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

identity
Identity value to set

map_exit = <dace.transformation.transformation.PatternNode object>

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

outer_map_exit = <dace.transformation.transformation.PatternNode object>

properties()

class dace.transformation.dataflow.stream_transient.**StreamTransient**(**args*, ***kwargs*)
Bases: *dace.transformation.transformation.Transformation*

Implements the StreamTransient transformation, which adds a transient and stream nodes between nested maps that lead to a stream. The transient then acts as a local buffer.

apply(*sdfg: dace.sdfg.sdfg.SDFG*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

map_exit = <dace.transformation.transformation.PatternNode object>

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

outer_map_exit = <dace.transformation.transformation.PatternNode object>

properties()

tasklet = <dace.transformation.transformation.PatternNode object>

with_buffer
Use an intermediate buffer for accumulation

dace.transformation.dataflow.stream_transient.**calc_set_image**(*map_idx*, *map_set*, *array_set*)

dace.transformation.dataflow.stream_transient.**calc_set_image_index**(*map_idx*, *map_set*, *array_idx*)

dace.transformation.dataflow.stream_transient.**calc_set_image_range**(*map_idx*, *map_set*, *array_range*)

dace.transformation.dataflow.strip_mining module

This module contains classes and functions that implement the strip-mining transformation.

class dace.transformation.dataflow.strip_mining.**StripMining**(**args*, ***kwargs*)
Bases: *dace.transformation.transformation.Transformation*

Implements the strip-mining transformation.

Strip-mining takes as input a map dimension and splits it into two dimensions. The new dimension iterates over the range of the original one with a parameterizable step, called the tile size. The original dimension is changed to iterates over the range of the tile size, with the same step as before.

static annotates_memlets()
Indicates whether the transformation annotates the edges it creates or modifies with the appropriate memlets. This determines whether to apply memlet propagation after the transformation.

apply(*sdfg: dace.sdfg.sdfg.SDFG*) → dace.sdfg.nodes.Map
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

dim_idx
Index of dimension to be strip-mined

divides_evenly
Tile size divides dimension range evenly?

entry

exit

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

new_dim_prefix
Prefix for new dimension name

print_match_pattern(*candidate*)

properties()

skew
If True, offsets inner tile back such that it starts with zero

strided
Continuous (false) or strided (true) elements in tile

tasklet

tile_offset
Tile stride offset (negative)

tile_size
Tile size of strip-mined dimension, or number of tiles if tiling_type=number_of_tiles

tile_stride
Stride between two tiles of the strip-mined dimension. If zero, it is set equal to the tile size.

tiling_type
normal: the outerloop increments with tile_size, ceilrange: uses ceiling(N/tile_size) in outer range, number_of_tiles: tiles the map into the number of provided tiles, provide the number of tiles over tile_size

dace.transformation.dataflow.strip_mining.**calc_set_image**(*map_idx*, *map_set*, *array_set*)

dace.transformation.dataflow.strip_mining.**calc_set_image_index**(*map_idx*, *map_set*, *array_idx*)

dace.transformation.dataflow.strip_mining.**calc_set_image_range**(*map_idx*, *map_set*, *array_range*)

dace.transformation.dataflow.strip_mining.**calc_set_union**(*set_a*, *set_b*)

dace.transformation.dataflow.tiling module

This module contains classes and functions that implement the orthogonal tiling transformation.

class dace.transformation.dataflow.tiling.**MapTiling**(**args*, ***kwargs*)
Bases: *dace.transformation.transformation.Transformation*

Implements the orthogonal tiling transformation.

Orthogonal tiling is a type of nested map fission that creates tiles in every dimension of the matched Map.

static annotates_memlets()
Indicates whether the transformation annotates the edges it creates or modifies with the appropriate memlets. This determines whether to apply memlet propagation after the transformation.

apply(*sdfg*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

divides_evenly
Tile size divides dimension length evenly

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

map_entry = <dace.transformation.transformation.PatternNode object>

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

prefix
Prefix for new range symbols

properties()

strides
Tile stride (enables overlapping tiles). If empty, matches tile

tile_offset
Negative Stride offset per dimension

tile_sizes
Tile size per dimension

tile_trivial
Tiles even if tile_size is 1

dace.transformation.dataflow.vectorization module

Contains classes that implement the vectorization transformation.

class dace.transformation.dataflow.vectorization.**Vectorization**(**args*, ***kwargs*)

Bases: *dace.transformation.transformation.Transformation*

Implements the vectorization transformation.

Vectorization matches when all the input and output memlets of a tasklet inside a map access the inner-most loop variable in their last dimension. The transformation changes the step of the inner-most loop to be equal to the length of the vector and vectorizes the memlets.

apply(*sdfg: dace.sdfg.sdfg.SDFG*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

can_be_applied(*graph: dace.sdfg.state.SDFGState*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

postamble
Force creation or skipping a postamble map without vectors

preamble
Force creation or skipping a preamble map without vectors

propagate_parent
Propagate vector length through parent SDFGs

properties()

strided_map
Use strided map range (jump by vector length) instead of modifying memlets

vector_len
Vector length

Module contents

This module initializes the dataflow transformations package.

dace.transformation.interstate package

Submodules

dace.transformation.interstate.fpga_transform_sdfg module

Contains inter-state transformations of an SDFG to run on an FPGA.

class dace.transformation.interstate.fpga_transform_sdfg.**FPGATransformSDFG**(**args*, ***kwargs*)

Bases: *dace.transformation.transformation.Transformation*

Implements the FPGATransformSDFG transformation, which takes an entire SDFG and transforms it into an FPGA-capable SDFG.

static annotates_memlets()
Indicates whether the transformation annotates the edges it creates or modifies with the appropriate memlets. This determines whether to apply memlet propagation after the transformation.

apply(*sdfg*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

promote_global_trans
If True, transient arrays that are fully internal are pulled out so that they can be allocated on the host.

properties()

dace.transformation.interstate.fpga_transform_state module

Contains inter-state transformations of an SDFG to run on an FPGA.

class dace.transformation.interstate.fpga_transform_state.**FPGATransformState**(**args*, ***kwargs*)
Bases: *dace.transformation.transformation.Transformation*

Implements the FPGATransformState transformation.

apply(*sdfg*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

dace.transformation.interstate.fpga_transform_state.**fpga_update**(*sdfg*, *state*, *depth*)

dace.transformation.interstate.gpu_transform_sdfg module

Contains inter-state transformations of an SDFG to run on the GPU.

class dace.transformation.interstate.gpu_transform_sdfg.**GPUTransformSDFG**(**args*, ***kwargs*)

Bases: *dace.transformation.transformation.Transformation*

Implements the GPUTransformSDFG transformation.

Transforms a whole SDFG to run on the GPU: Steps of the full GPU transform

0. Acquire metadata about SDFG and arrays
1. Replace all non-transients with their GPU counterparts
2. Copy-in state from host to GPU
3. Copy-out state from GPU to host
4. Re-store Default-top/CPU_Heap transients as GPU_Global
5. Global tasklets are wrapped with a map of size 1
6. Global Maps are re-scheduled to use the GPU
7. Make data ready for interstate edges that use them
8. Re-apply strict transformations to get rid of extra states and transients

static annotates_memlets()
Indicates whether the transformation annotates the edges it creates or modifies with the appropriate memlets. This determines whether to apply memlet propagation after the transformation.

apply(*sdfg: dace.sdfg.sdfg.SDFG*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

exclude_copyin
Exclude these arrays from being copied into the device (comma-separated)

exclude_copyout
Exclude these arrays from being copied out of the device (comma-separated)

exclude_tasklets
Exclude these tasklets from being processed as CPU tasklets (comma-separated)

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

properties()

register_trans
Make all transients inside GPU maps registers

sequential_innermaps
Make all internal maps Sequential

skip_scalar_tasklets
If True, does not transform tasklets that manipulate (Default-stored) scalars

strict_transform
Reapply strict transformations after modifying graph

toplevel_trans
Make all GPU transients top-level

dace.transformation.interstate.loop_detection module

Loop detection transformation

class dace.transformation.interstate.loop_detection.**DetectLoop**(**args*, ***kwargs*)
Bases: *dace.transformation.transformation.Transformation*

Detects a for-loop construct from an SDFG.

apply(*sdfg*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

dace.transformation.interstate.loop_detection.**find_for_loop**(*sdfg: dace.sdfg.sdfg.SDFG*, *guard: dace.sdfg.state.SDFGState*, *entry: dace.sdfg.state.SDFGState*, *itervar: Optional[str] = None*) → Optional[Tuple[AnyStr, Tuple[Union[sympy.core.basic.Basic, dace.symbolic.SymExpr], Union[sympy.core.basic.Basic, dace.symbolic.SymExpr], Union[sympy.core.basic.Basic, dace.symbolic.SymExpr]], Tuple[List[dace.sdfg.state.SDFGState], dace.sdfg.state.SDFGState]]]
Finds loop range from state machine. :param guard: State from which the outgoing edges detect whether to exit

the loop or not.

Parameters **entry** – First state in the loop "body".

Returns

(iteration variable, (start, end, stride), (start_states[], last_loop_state)), or None if proper

for-loop was not detected. `end` is inclusive.

dace.transformation.interstate.loop_peeling module

Loop unroll transformation

class dace.transformation.interstate.loop_peeling.**LoopPeeling**(**args*, ***kwargs*)
Bases: *dace.transformation.interstate.loop_unroll.LoopUnroll*

Splits the first *count* iterations of a state machine for-loop into multiple, separate states.

apply(*sdfg: dace.sdfg.sdfg.SDFG*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

begin
If True, peels loop from beginning (first *count* iterations), otherwise peels last *count* iterations.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

properties()

dace.transformation.interstate.loop_unroll module

Loop unroll transformation

class dace.transformation.interstate.loop_unroll.**LoopUnroll**(**args*, ***kwargs*)
Bases: *dace.transformation.interstate.loop_detection.DetectLoop*

Unrolls a state machine for-loop into multiple states

apply(*sdfg*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

count
Number of iterations to unroll, or zero for all iterations (loop must be constant-sized for 0)

instantiate_loop(*sdfg: dace.sdfg.sdfg.SDFG, loop_states: List[dace.sdfg.state.SDFGState], loop_subgraph: dace.sdfg.graph.SubgraphView, itervar: str, value: Union[sympy.core.basic.Basic, dace.symbolic.SymExpr], state_suffix=None*)

properties()

dace.transformation.interstate.sdfg_nesting module

SDFG nesting transformation.

class dace.transformation.interstate.sdfg_nesting.**ASTRefiner**(*to_refine: str*, *refine_subset: dace.subsets.Subset*, *sdfg: dace.sdfg.sdfg.SDFG*, *indices: Set[int] = None*)

Bases: ast.NodeTransformer

Python AST transformer used in RefineNestedAccess to reduce (refine) the subscript ranges based on the specification given in the transformation.

visit_Subscript(*node: _ast.Subscript*) → _ast.Subscript

class dace.transformation.interstate.sdfg_nesting.**InlineSDFG**(**args*, ***kwargs*)

Bases: *dace.transformation.transformation.Transformation*

Inlines a single-state nested SDFG into a top-level SDFG.

In particular, the steps taken are:

1. All transient arrays become transients of the parent
2. If a source/sink node is one of the inputs/outputs:
   a. Remove it
   b. Reconnect through external edges (map/accessnode)
   c. Replace and reoffset memlets with external data descriptor
3. If other nodes carry the names of inputs/outputs:
   a. Replace data with external data descriptor
   b. Replace and reoffset memlets with external data descriptor
4. If source/sink node is not connected to a source/destination, and the nested SDFG is in a scope, connect to scope with empty memlets
5. Remove all unused external inputs/output memlet paths
6. Remove isolated nodes resulting from previous step

static annotates_memlets()

Indicates whether the transformation annotates the edges it creates or modifies with the appropriate memlets. This determines whether to apply memlet propagation after the transformation.

apply(*sdfg: dace.sdfg.sdfg.SDFG*)

Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph: dace.sdfg.state.SDFGState*, *candidate*, *expr_index*, *sdfg*, *strict=False*)

Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

properties()

class dace.transformation.interstate.sdfg_nesting.**InlineTransients**(**args*, ***kwargs*)

Bases: *dace.transformation.transformation.Transformation*

Inlines all transient arrays that are not used anywhere else into a nested SDFG.

static annotates_memlets()
Indicates whether the transformation annotates the edges it creates or modifies with the appropriate memlets. This determines whether to apply memlet propagation after the transformation.

apply(*sdfg*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph: dace.sdfg.state.SDFGState*, *candidate: Dict[dace.transformation.transformation.PatternNode, int]*, *expr_index: int*, *sdfg: dace.sdfg.sdfg.SDFG*, *strict: bool = False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

nsdfg = <dace.transformation.transformation.PatternNode object>

properties()

class dace.transformation.interstate.sdfg_nesting.**NestSDFG**(**args*, ***kwargs*)
Bases: *dace.transformation.transformation.Transformation*

Implements SDFG Nesting, taking an SDFG as an input and creating a nested SDFG node from it.

static annotates_memlets()
Indicates whether the transformation annotates the edges it creates or modifies with the appropriate memlets. This determines whether to apply memlet propagation after the transformation.

apply(*sdfg*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

promote_global_trans
Promotes transients to be allocated once

properties()

class dace.transformation.interstate.sdfg_nesting.**RefineNestedAccess**(**args*, ***kwargs*)
Bases: *dace.transformation.transformation.Transformation*

Reduces memlet shape when a memlet is connected to a nested SDFG, but not using all of the contents. Makes the outer memlet smaller in shape and ensures that the offsets in the nested SDFG start with zero. This helps with subsequent transformations on the outer SDFGs.

For example, in the following program:

```
@dace.program
def func_a(y):
    return y[1:5] + 1

@dace.program
def main(x: dace.float32[N]):
    return func_a(x)
```

The memlet pointing to `func_a` will contain all of `x` (`x[0:N]`), and it is offset to `y[1:5]` in the function, with `y`'s size being `N`. After the transformation, the memlet connected to the nested SDFG of `func_a` would contain `x[1:5]` directly and the internal `y` array would have a size of 4, accessed as `y[0:4]`.

static annotates_memlets()
Indicates whether the transformation annotates the edges it creates or modifies with the appropriate memlets. This determines whether to apply memlet propagation after the transformation.

apply(*sdfg*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph: dace.sdfg.state.SDFGState, candidate: Dict[dace.transformation.transformation.PatternNode, int], expr_index: int, sdfg: dace.sdfg.sdfg.SDFG, strict: bool = False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

nsdfg = <dace.transformation.transformation.PatternNode object>

properties()

dace.transformation.interstate.state_elimination module

State elimination transformations

class dace.transformation.interstate.state_elimination.**EndStateElimination**(**args*, ***kwargs*)

Bases: *dace.transformation.transformation.Transformation*

End-state elimination removes a redundant state that has one incoming edge and no contents.

apply(*sdfg*)

Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)

Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()

Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)

Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

class dace.transformation.interstate.state_elimination.**HoistState**(**args*, ***kwargs*)

Bases: *dace.transformation.transformation.Transformation*

Move a state out of a nested SDFG

apply(*sdfg: dace.sdfg.sdfg.SDFG*)

Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph: dace.sdfg.state.SDFGState*, *candidate*, *expr_index*, *sdfg*, *strict=False*)

Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

nsdfg = <dace.transformation.transformation.PatternNode object>

class dace.transformation.interstate.state_elimination.**StartStateElimination**(**args*, ***kwargs*)

Bases: *dace.transformation.transformation.Transformation*

Start-state elimination removes a redundant state that has one outgoing edge and no contents. This transformation applies only to nested SDFGs.

apply(*sdfg*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

start_state = SDFGState (None)

class dace.transformation.interstate.state_elimination.**StateAssignElimination**(**args*, ***kwargs*)

Bases: *dace.transformation.transformation.Transformation*

State assign elimination removes all assignments into the final state and subsumes the assigned value into its contents.

apply(*sdfg*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

class dace.transformation.interstate.state_elimination.**SymbolAliasPromotion**(**args*, ***kwargs*)
Bases: *dace.transformation.transformation.Transformation*

SymbolAliasPromotion moves inter-state assignments that create symbolic aliases to the previous inter-state edge according to the topological order. The purpose of this transformation is to iteratively move symbolic aliases together, so that true duplicates can be easily removed.

apply(*sdfg*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.

- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

dace.transformation.interstate.state_fusion module

State fusion transformation

class dace.transformation.interstate.state_fusion.**CCDesc**(*first_inputs: Set[str], first_outputs: Set[str], first_output_nodes: Set[dace.sdfg.nodes.AccessNode], second_inputs: Set[str], second_outputs: Set[str], second_input_nodes: Set[dace.sdfg.nodes.AccessNode]*)
Bases: object

class dace.transformation.interstate.state_fusion.**StateFusion**(**args*, ***kwargs*)
Bases: *dace.transformation.transformation.Transformation*

Implements the state-fusion transformation.

State-fusion takes two states that are connected through a single edge, and fuses them into one state. If strict, only applies if no memory access hazards are created.

static annotates_memlets()
Indicates whether the transformation annotates the edges it creates or modifies with the appropriate memlets. This determines whether to apply memlet propagation after the transformation.

apply(*sdfg*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static find_fused_components(*first_cc_input*, *first_cc_output*, *second_cc_input*, *second_cc_output*) → List[dace.transformation.interstate.state_fusion.CCDesc]

first_state = <dace.transformation.transformation.PatternNode object>

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

static memlets_intersect(*graph_a: dace.sdfg.state.SDFGState*, *group_a: List[dace.sdfg.nodes.AccessNode]*, *inputs_a: bool*, *graph_b: dace.sdfg.state.SDFGState*, *group_b: List[dace.sdfg.nodes.AccessNode]*, *inputs_b: bool*) → bool
Performs an all-pairs check for subset intersection on two groups of nodes. If group intersects or result is indeterminate, returns True as a precaution. :param graph_a: The graph in which the first set of nodes reside. :param group_a: The first set of nodes to check. :param inputs_a: If True, checks inputs of the first group. :param graph_b: The graph in which the second set of nodes reside. :param group_b: The second set of nodes to check. :param inputs_b: If True, checks inputs of the second group. :returns True if subsets intersect or result is indeterminate.

second_state = <dace.transformation.transformation.PatternNode object>

dace.transformation.interstate.state_fusion.**top_level_nodes**(*state: dace.sdfg.state.SDFGState*)

dace.transformation.interstate.transient_reuse module

class dace.transformation.interstate.transient_reuse.**TransientReuse**(**args*, ***kwargs*)

Bases: *dace.transformation.transformation.Transformation*

Implements the TransientReuse transformation. Finds all possible reuses of arrays, decides for a valid combination and changes sdfg accordingly.

apply(*sdfg*)
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)
Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.

- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

expansion()

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

properties()

Module contents

This module initializes the inter-state transformations package.

dace.transformation.subgraph package

Submodules

dace.transformation.subgraph.expansion module

This module contains classes that implement the expansion transformation.

class dace.transformation.subgraph.expansion.**MultiExpansion**(**args*, ***kwargs*)
Bases: *dace.transformation.transformation.SubgraphTransformation*

Implements the MultiExpansion transformation. Takes all the lowest scope maps in a given subgraph, for each of these maps splits it into an outer and inner map, where the outer map contains the common ranges of all maps, and the inner map the rest. Map access variables and memlets are changed accordingly

allow_offset
Offset ranges to zero

apply(*sdfg*, *map_base_variables=None*)
Applies the transformation on the given subgraph. :param sdfg: The SDFG that includes the subgraph.

can_be_applied(*sdfg: dace.sdfg.sdfg.SDFG*, *subgraph: dace.sdfg.graph.SubgraphView*) → bool
Tries to match the transformation on a given subgraph, returning True if this transformation can be applied. :param sdfg: The SDFG that includes the subgraph. :param subgraph: The SDFG or state subgraph to try to apply the

transformation on.

Returns True if the subgraph can be transformed, or False otherwise.

check_contiguity
Don't allow expansion if last (contiguous)dimension is partially split

debug
Debug Mode

expand(*sdfg*, *graph*, *map_entries*, *map_base_variables=None*)
Expansion into outer and inner maps for each map in a specified set. The resulting outer maps all have same range and indices, corresponding variables and memlets get changed accordingly. The inner map contains the leftover dimensions :param sdfg: Underlying SDFG :param graph: Graph in which we expand :param map_entries: List of Map Entries(Type MapEntry) that we want to expand :param map_base_variables: Optional parameter. List of strings

If None, then expand() searches for the maximal amount of equal map ranges and pushes those and their corresponding loop variables into the outer loop. If specified, then expand() pushes the ranges belonging to the loop iteration variables specified into the outer loop (For instance map_base_variables = ['i','j'] assumes that all maps have common iteration indices i and j with corresponding correct ranges)

permutation_only
Only allow permutations without inner splits

properties()

sequential_innermaps
Make all inner maps that arecreated during expansion sequential

dace.transformation.subgraph.expansion.**offset_map**(*state*, *map_entry*)

dace.transformation.subgraph.gpu_persistent_fusion module

class dace.transformation.subgraph.gpu_persistent_fusion.**GPUPersistentKernel**(**args*, ***kwargs*)
Bases: *dace.transformation.transformation.SubgraphTransformation*

This transformation takes a given subgraph of an SDFG and fuses the given states into a single persistent GPU kernel. Before this transformation can be applied the SDFG needs to be transformed to run on the GPU (e.g. with the GPUTransformSDFG transformation).

If applicable the transform removes the selected states from the original SDFG and places a *launch* state in its place. The removed states will be added to a nested SDFG in the launch state. If necessary guard states will be added in the nested SDFG, in order to make sure global assignments on Interstate edges will be performed in the kernel (this can be disabled with the *include_in_assignment* property).

The given subgraph needs to fulfill the following properties to be fused:

- **All states in the selected subgraph need to fulfill the following:**
  - access only GPU accessible memory
  - all concurrent DFGs inside the state are either sequential or inside a GPU_Device map.
- the selected subgraph has a single point of entry in the form of a single InterstateEdge entering the subgraph (i.e. there is at most one state (not part of the subgraph) from which the kernel is entered and exactly one state inside the subgraph from which the kernel starts execution)
- the selected subgraph has a single point of exit in the form of a single state that is entered after the selected subgraph is left (There can be multiple states from which the kernel can be left, but all will leave to the same state outside the subgraph)

apply(*sdfg: dace.sdfg.sdfg.SDFG*)
Applies the transformation on the given subgraph. :param sdfg: The SDFG that includes the subgraph.

static can_be_applied(*sdfg: dace.sdfg.sdfg.SDFG*, *subgraph: dace.sdfg.graph.SubgraphView*)
Tries to match the transformation on a given subgraph, returning True if this transformation can be applied. :param sdfg: The SDFG that includes the subgraph. :param subgraph: The SDFG or state subgraph to try to apply the

transformation on.

Returns True if the subgraph can be transformed, or False otherwise.

static get_entry_states(*sdfg: dace.sdfg.sdfg.SDFG*, *subgraph*)

static get_exit_states(*sdfg: dace.sdfg.sdfg.SDFG*, *subgraph*)

include_in_assignment
Wether to include global variable assignments of the edge going into the kernel inside the kernel or have it happen on the outside. If the assignment is needed in the kernel, it needs to be included.

static is_gpu_state(*sdfg: dace.sdfg.sdfg.SDFG*, *state: dace.sdfg.state.SDFGState*) → bool

kernel_prefix
Name of the kernel. If no value is given the kerenl will be refrenced as *kernel*, if a value is given the kernel will be named <*kernel_prefix*>*_kernel*. This is useful if multiple kernels are created.

properties()

validate
Validate the sdfg and the nested sdfg

dace.transformation.subgraph.helpers module

Subgraph Transformation Helper API

dace.transformation.subgraph.helpers.**common_map_base_ranges**(*ranges: List[dace.subsets.Range]*) → List[dace.subsets.Range]
Finds a maximal set of ranges that can be found in every instance of the ranges in the given list

dace.transformation.subgraph.helpers.**find_reassignment**(*maps: List[dace.sdfg.nodes.Map], common_ranges, offset=False*) → Dict[dace.sdfg.nodes.Map, List[T]]
Provided a list of maps and their common base ranges (found via common_map_base_ranges()), for each map greedily assign each loop to an index so that a base range has the same index in every loop. If a loop range of a certain map does not correspond to a common base range, no index is assigned (=-1)

Parameters

- **maps** – List of maps
- **common_ranges** – Common ranges extracted via common_map_base_ranges()
- **offset** – If true, offsets each range to 0 before checking

Returns Dict that maps each map to a vector with the same length as number of map loops. The vector contains, in order, an index for each map loop that maps it to a common base range or '-1' if it does not.

dace.transformation.subgraph.helpers.**get_outermost_scope_maps**(*sdfg*, *graph*, *subgraph=None*, *scope_dict=None*)
Returns all Map Entries inside of a given subgraph that have the outermost scope. If the underlying subgraph is not connected, there might be multiple locally outermost scopes. In this ambiguous case, the method returns an empty list. If subgraph == None, the whole graph is taken for analysis.

dace.transformation.subgraph.helpers.**outermost_scope_from_maps**(*graph*, *maps*, *scope_dict=None*)

Returns the outermost scope of a set of given maps. If the underlying maps are not topologically connected to each other, there might be several scopes that are locally outermost. In this case it throws an Exception

dace.transformation.subgraph.helpers.**outermost_scope_from_subgraph**(*graph*, *subgraph*, *scope_dict=None*)

Returns the outermost scope of a subgraph. If the subgraph is not connected, there might be several scopes that are locally outermost. In this case, it throws an Exception.

dace.transformation.subgraph.helpers.**subgraph_from_maps**(*sdfg*, *graph*, *map_entries*, *scope_children=None*)

Given a list of map entries in a single graph, return a subgraph view that includes all nodes inside these maps as well as map entries and exits as well as adjacent nodes.

dace.transformation.subgraph.reduce_expansion module

This module contains classes that implement the reduce-map transformation.

class dace.transformation.subgraph.reduce_expansion.**ReduceExpansion**(**args*, ***kwargs*)

Bases: *dace.transformation.transformation.Transformation*

Implements the ReduceExpansion transformation. Expands a Reduce node into inner and outer map components, where the outer map consists of the axes not being reduced. A new reduce node is created inside the inner map. Special cases where e.g reduction identities are not defined and arrays being reduced to already exist are handled on the fly.

apply(*sdfg: dace.sdfg.sdfg.SDFG*, *strict=False*)

Splits the data dimension into an inner and outer dimension, where the inner dimension are the reduction axes and the outer axes the complement. Pushes the reduce inside a new map consisting of the complement axes.

static can_be_applied(*graph*, *candidate*, *expr_index*, *sdfg*, *strict=False*)

Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

create_in_transient

Create local in-transientin registers

create_out_transient

Create local out-transientin registers

debug

Debug Info

expand(*sdfg*, *graph*, *reduce_node*)
Splits the data dimension into an inner and outer dimension, where the inner dimension are the reduction axes and the outer axes the complement. Pushes the reduce inside a new map consisting of the complement axes.

static expressions()
Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static match_to_str(*graph*, *candidate*)
Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

properties()

reduce_implementation
Reduce implementation of inner reduce. If specified,overrides any existing implementations

reduction_type_identity = {<ReductionType.Sum: 4>: 0, <ReductionType.Product: 5>:

reduction_type_update = {<ReductionType.Max: 3>: 'out = max(reduction_in, array_in)'

dace.transformation.subgraph.subgraph_fusion module

This module contains classes that implement subgraph fusion

class dace.transformation.subgraph.subgraph_fusion.**SubgraphFusion**(**args*, ***kwargs*)
Bases: *dace.transformation.transformation.SubgraphTransformation*

Implements the SubgraphFusion transformation. Fuses together the maps contained in the subgraph and pushes inner nodes into a global outer map, creating transients and new connections where necessary.

SubgraphFusion requires all lowest scope level maps in the subgraph to have the same indices and parameter range in every dimension. This can be achieved using the MultiExpansion transformation first. Reductions can also be expanded using ReduceExpansion as a preprocessing step.

adjust_arrays_nsdfg(*sdfg: dace.sdfg.sdfg.SDFG*, *nsdfg: dace.sdfg.nodes.NestedSDFG*, *name: str*, *nname: str*, *memlet: dace.memlet.Memlet*)
DFS to replace strides and volumes of data that exhibits nested SDFGs adjacent to its corresponding access nodes, applied during post-processing of a fused graph. Operates in-place. :param sdfg: SDFG :param nsdfg: The Nested SDFG of interest :param name: Name of the array in the SDFG :param nname: Name of the array in the nested SDFG :param memlet: Memlet adjacent to the nested SDFG that leads to the

access node with the corresponding data name

apply(*sdfg*, *do_not_override=None*, ***kwargs*)
Apply the SubgraphFusion Transformation. See @fuse for more details

can_be_applied(*sdfg: dace.sdfg.sdfg.SDFG*, *subgraph: dace.sdfg.graph.SubgraphView*) → bool
Fusible if 1. Maps have the same access sets and ranges in order 2. Any nodes in between two maps are AccessNodes only, without WCR

There is at most one AccessNode only on a path between two maps, no other nodes are allowed

3. The exiting memlets' subsets to an intermediate edge must cover the respective incoming memlets' subset into the next map. Also, as a limitation, the union of all exiting memlets' subsets must be contiguous.
4. Check for any disjoint accesses of arrays.

static check_topo_feasibility(*sdfg*, *graph*, *map_entries*, *intermediate_nodes*, *out_nodes*)
Checks whether given outermost scoped map entries have topological structure apt for fusion :param sdfg: SDFG :param graph: State :param map_entries: List of outermost scoped map entries induced by subgraph :param intermediate_nodes: List of intermediate access nodes :param out_nodes: List of outgoing access nodes :return: Boolean value indicating fusibility

clone_intermediate_nodes(*sdfg: dace.sdfg.sdfg.SDFG, graph: dace.sdfg.state.SDFGState, intermediate_nodes: List[dace.sdfg.nodes.AccessNode], out_nodes: List[dace.sdfg.nodes.AccessNode], map_entries: List[dace.sdfg.nodes.MapEntry], map_exits: List[dace.sdfg.nodes.MapExit]*)
Creates cloned access nodes and data arrays for nodes that are both in intermediate nodes and out nodes, redirecting output from the original node to the cloned node. Operates in-place. :param sdfg: SDFG :param state: State of interest :param intermediate_nodes: List of intermediate nodes appearing in a fusible subgraph :param out_nodes: List of out nodes appearing in a fusible subgraph :param map_entries: List of outermost scoped map entries in the subgraph :param map_exits: List of map exits corresponding to map_entries in order :return: A dict that maps each intermediate node that also functions as an out node

to the respective cloned transient node

consolidate
Consolidate edges that enter and exit the fused map.

copy_edge(*graph*, *edge*, *new_src=None*, *new_src_conn=None*, *new_dst=None*, *new_dst_conn=None*, *new_data=None*, *remove_old=False*)
Copies an edge going from source to dst. If no destination is specified, the edge is copied with the same destination and port as the original edge, else the edge is copied with the new destination and the new port. If no source is specified, the edge is copied with the same source and port as the original edge, else the edge is copied with the new source and the new port If remove_old is specified, the old edge is removed immediately If new_data is specified, inserts new_data as a memlet, else else makes a deepcopy of the current edges memlet

debug
Show debug info

static determine_compressible_nodes(*sdfg: dace.sdfg.sdfg.SDFG, graph: dace.sdfg.state.SDFGState, intermediate_nodes: List[dace.sdfg.nodes.AccessNode], map_entries: List[dace.sdfg.nodes.MapEntry], map_exits: List[dace.sdfg.nodes.MapExit], do_not_override: List[str] = []*)
Checks for all intermediate nodes whether they appear only within the induced fusible subgraph my map_entries and map_exits. This is returned as a dict that contains a boolean value for each intermediate node as a key. :param sdfg: SDFG :param state: State of interest :param intermediate_nodes: List of intermediate nodes appearing in a fusible subgraph :param map_entries: List of outermost scoped map entries in the subgraph :param map_exits: List of map exits corresponding to map_entries in order :param do_not_override: List of data array names not to be compressed :param return: A dictionary indicating for each data string whether its array can be compressed

determine_invariant_dimensions(*sdfg: dace.sdfg.sdfg.SDFG, graph: dace.sdfg.state.SDFGState, intermediate_nodes: List[dace.sdfg.nodes.AccessNode], map_entries: List[dace.sdfg.nodes.MapEntry], map_exits: List[dace.sdfg.nodes.MapExit]*)
Determines the invariant dimensions for each node – dimensions in which the access set of the memlets propagated through map entries and exits does not change. :param sdfg: SDFG :param state: State of interest :param intermediate_nodes: List of intermediate nodes appearing in a fusible subgraph :param map_entries: List of outermost scoped map entries in the subgraph :param map_exits: List

of map exits corresponding to map_entries in order :return: A dict mapping each intermediate node (nodes.AccessNode) to a list of integer dimensions

disjoint_subsets
Check for disjoint subsets in can_be_applied. If multipleaccess nodes pointing to the same data appear within a subgraphto be fused, this check confirms that their access sets areindependent per iteration space to avoid race conditions.

fuse(*sdfg: dace.sdfg.sdfg.SDFG, graph: dace.sdfg.state.SDFGState, map_entries: List[dace.sdfg.nodes.MapEntry], do_not_override=None, **kwargs*)
takes the map_entries specified and tries to fuse maps.

all maps have to be extended into outer and inner map (use MapExpansion as a pre-pass)

Arrays that don't exist outside the subgraph get pushed into the map and their data dimension gets cropped. Otherwise the original array is taken.

For every output respective connections are crated automatically.

Parameters

- **sdfg** – SDFG
- **graph** – State
- **map_entries** – Map Entries (class MapEntry) of the outer maps which we want to fuse
- **do_not_override** – List of data names whose corresponding nodes are fully contained within the subgraph but should not be compressed nevertheless.

static get_adjacent_nodes(*sdfg*, *graph*, *map_entries*) → Tuple[List[dace.sdfg.nodes.AccessNode], List[dace.sdfg.nodes.AccessNode], List[dace.sdfg.nodes.AccessNode]]
For given map entries, finds a set of in, out and intermediate nodes as defined below :param sdfg: SDFG :param graph: State of interest :param map_entries: List of all outermost scoped maps that induce the subgraph :return: Tuple of (in_nodes, intermediate_nodes, out_nodes)

- In_nodes are nodes that serve as pure input nodes for the map entries
- Out nodes are nodes that serve as pure output nodes for the map entries
- Interemdiate nodes are nodes that serve as buffer storage between outermost scoped map entries and exits of the induced subgraph

-> in_nodes are trivially disjoint from the other two types of access nodes -> Intermediate_nodes and out_nodes are not necessarily disjoint

get_invariant_dimensions(*sdfg: dace.sdfg.sdfg.SDFG, graph: dace.sdfg.state.SDFGState, map_entries: List[dace.sdfg.nodes.MapEntry], map_exits: List[dace.sdfg.nodes.MapExit], node: dace.sdfg.nodes.AccessNode*)
For a given intermediate access node, return a set of indices that correspond to array / subset dimensions in which no change is observed upon propagation through the corresponding map nodes in map_entries / map_exits. :param map_entries: List of outermost scoped map entries :param map_exits: List of corresponding exit nodes to map_entries, in order :param node: Intermediate access node of interest :return: Set of invariant integer dimensions

keep_global
A list of array names to treat as non-transients and not compress

prepare_intermediate_nodes(*sdfg: dace.sdfg.sdfg.SDFG, graph: dace.sdfg.state.SDFGState, in_nodes: List[dace.sdfg.nodes.AccessNode], out_nodes: List[dace.sdfg.nodes.AccessNode], intermediate_nodes: List[dace.sdfg.nodes.AccessNode], map_entries: List[dace.sdfg.nodes.MapEntry], map_exits: List[dace.sdfg.nodes.MapExit], do_not_override: List[str] = []*)

Helper function that computes the following information: 1. Determine whether intermediate nodes only appear within the induced fusible subgraph. This is equivalent to checking for compresssibility. 2. Determine whether any intermediate transients are also out nodes, if so they have to be cloned 3. Determine invariant dimensions for any intermediate transients (that are compressible). :return: A tuple (subgraph_contains_data, transients_created, invariant_dimensions)

of dictionaries containing the necessary information

propagate

Propagate memlets of edges that enter and exit the fused map.Disable if this causes problems (e.g., if memlet propagation doesnot work correctly).

properties()

schedule_innermaps

Schedule of inner maps. If none, keeps schedule.

transient_allocation

Storage Location to push transients to that are fully contained within the subgraph.

Module contents

This module initializes the subgraph transformations package.

Submodules

dace.transformation.transformation module

Contains classes that represent data-centric transformations.

There are three general types of transformations:

- Pattern-matching Transformations (extending Transformation): Transformations that require a certain subgraph structure to match.
- Subgraph Transformations (extending SubgraphTransformation): Transformations that can operate on arbitrary subgraphs.
- Library node expansions (extending ExpandTransformation): An internal class used for tracking how library nodes were expanded.

class dace.transformation.transformation.**ExpandTransformation**(**args*, ***kwargs*)

Bases: *dace.transformation.transformation.Transformation*

Base class for transformations that simply expand a node into a subgraph, and thus needs only simple matching and replacement functionality. Subclasses only need to implement the method “expansion”.

This is an internal interface used to track the expansion of library nodes.

apply(*sdfg*, **args*, ***kwargs*)

Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

static can_be_applied(*graph: dace.sdfg.graph.OrderedMultiDiConnectorGraph, candidate: Dict[dace.sdfg.nodes.Node, int], expr_index: int, sdfg, strict: bool = False*)

Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

static expansion(*node*)

classmethod expressions()

Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

static from_json(*json_obj: Dict[str, Any], context: Dict[str, Any] = None*) → dace.transformation.transformation.ExpandTransformation

classmethod match_to_str(*graph: dace.sdfg.graph.OrderedMultiDiConnectorGraph, candidate: Dict[dace.sdfg.nodes.Node, int]*)

Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

static postprocessing(*sdfg*, *state*, *expansion*)

properties()

to_json(*parent=None*) → Dict[str, Any]

class dace.transformation.transformation.**PatternNode**(*nodeclass: Type[Union[dace.sdfg.nodes.Node, dace.sdfg.state.SDFGState]]*)

Bases: object

Static field wrapper of a node or an SDFG state that designates it as part of a subgraph pattern. These objects are used in subclasses of *Transformation* to represent the subgraph patterns.

Example use: ‘‘‘ @registry.autoregister_params(singlestate=True) class MyTransformation(Transformation):

some_map_node = PatternNode(nodes.MapEntry) array = PatternNode(nodes.AccessNode)

‘‘‘

The two nodes can then be used in the transformation static methods (e.g., *expressions*, *can_be_applied*) to represent the nodes, and in the instance methods to point to the nodes in the parent SDFG.

class dace.transformation.transformation.**SubgraphTransformation**(**args, **kwargs*)

Bases: *dace.transformation.transformation.TransformationBase*

Base class for transformations that apply on arbitrary subgraphs, rather than matching a specific pattern.

Subclasses need to implement the *can_be_applied* and *apply* operations, as well as registered with the subclass registry. See the *Transformation* class docstring for more information.

apply(*sdfg: dace.sdfg.sdfg.SDFG*)
Applies the transformation on the given subgraph. :param sdfg: The SDFG that includes the subgraph.

classmethod apply_to(*sdfg: dace.sdfg.sdfg.SDFG*, **where*, *verify: bool = True*, ***options*)
Applies this transformation to a given subgraph, defined by a set of nodes. Raises an error if arguments are invalid or transformation is not applicable.

To apply the transformation on a specific subgraph, the *where* parameter can be used either on a subgraph object (*SubgraphView*), or on directly on a list of subgraph nodes, given as *Node* or *SDFGState* objects. Transformation properties can then be given as keyword arguments. For example, applying *SubgraphFusion* on a subgraph of three nodes can be called in one of two ways: ‘‘‘ # Subgraph SubgraphFusion.apply_to(

sdfg, SubgraphView(state, [node_a, node_b, node_c]))

Simplified API: list of nodes SubgraphFusion.apply_to(sdfg, node_a, node_b, node_c) ‘‘‘

Parameters

- **sdfg** – The SDFG to apply the transformation to.
- **where** – A set of nodes in the SDFG/state, or a subgraph thereof.
- **verify** – Check that *can_be_applied* returns True before applying.
- **options** – A set of parameters to use for applying the transformation.

can_be_applied(*sdfg: dace.sdfg.sdfg.SDFG*, *subgraph: dace.sdfg.graph.SubgraphView*) → bool
Tries to match the transformation on a given subgraph, returning True if this transformation can be applied. :param sdfg: The SDFG that includes the subgraph. :param subgraph: The SDFG or state subgraph to try to apply the

transformation on.

Returns True if the subgraph can be transformed, or False otherwise.

extensions()

static from_json(*json_obj: Dict[str, Any]*, *context: Dict[str, Any] = None*) → dace.transformation.transformation.SubgraphTransformation

properties()

register(***kwargs*)

sdfg_id
ID of SDFG to transform

state_id
ID of state to transform subgraph within, or -1 to transform the SDFG

subgraph
Subgraph in transformation instance

subgraph_view(*sdfg: dace.sdfg.sdfg.SDFG*) → dace.sdfg.graph.SubgraphView

to_json(*parent=None*)

unregister()

class dace.transformation.transformation.**Transformation**(**args*, ***kwargs*)
Bases: *dace.transformation.transformation.TransformationBase*

Base class for pattern-matching transformations, as well as a static registry of transformations, where new transformations can be added in a decentralized manner. An instance of a Transformation represents a match of the transformation on an SDFG, complete with a subgraph candidate and properties.

New transformations that extend this class must contain static *PatternNode* fields that represent the nodes in the pattern graph, and use them to implement at least three methods:

- ***expressions*: A method that returns a list of graph** patterns (SDFG or SDFGState objects) that match this transformation.
- ***can_be_applied*: A method that, given a subgraph candidate,** checks for additional conditions whether it can be transformed.
- ***apply*: A method that applies the transformation** on the given SDFG.

For more information and optimization opportunities, see the respective methods' documentation.

In order to be included in lists and apply through the *sdfg.apply_transformations* API, each transformation shouls be registered with `Transformation.register` (or, more commonly, the `@dace.registry.autoregister_params` class decorator) with two optional boolean keyword arguments: `singlestate` (default: False) and `strict` (default: False). If `singlestate` is True, the transformation is matched on subgraphs inside an SDFGState; otherwise, subgraphs of the SDFG state machine are matched. If `strict` is True, this transformation will be considered strict (i.e., always beneficial to perform) and will be performed automatically as part of SDFG strict transformations.

annotates_memlets() → bool
Indicates whether the transformation annotates the edges it creates or modifies with the appropriate memlets. This determines whether to apply memlet propagation after the transformation.

apply(*sdfg: dace.sdfg.sdfg.SDFG*) → Optional[Any]
Applies this transformation instance on the matched pattern graph. :param sdfg: The SDFG to apply the transformation to. :return: A transformation-defined return value, which could be used

to pass analysis data out, or nothing.

apply_pattern(*sdfg: dace.sdfg.sdfg.SDFG*, *append: bool = True*, *annotate: bool = True*) → Optional[Any]
Applies this transformation on the given SDFG, using the transformation instance to find the right SDFG object (based on SDFG ID), and applying memlet propagation as necessary. :param sdfg: The SDFG (or an SDFG in the same hierarchy) to apply the

transformation to.

Parameters **append** – If True, appends the transformation to the SDFG transformation history.

Returns A transformation-defined return value, which could be used to pass analysis data out, or nothing.

classmethod apply_to(*sdfg: dace.sdfg.sdfg.SDFG*, *options: Optional[Dict[str, Any]] = None*, *expr_index: int = 0*, *verify: bool = True*, *annotate: bool = True*, *strict: bool = False*, *save: bool = True*, ***where*)
Applies this transformation to a given subgraph, defined by a set of nodes. Raises an error if arguments are invalid or transformation is not applicable.

The subgraph is defined by the *where* dictionary, where each key is taken from the *PatternNode* fields of the transformation. For example, applying *MapCollapse* on two maps can pe performed as follows:

```
` MapCollapse.apply_to(sdfg, outer_map_entry=map_a,
inner_map_entry=map_b) `
```

Parameters

- **sdfg** – The SDFG to apply the transformation to.
- **options** – A set of parameters to use for applying the transformation.
- **expr_index** – The pattern expression index to try to match with.
- **verify** – Check that *can_be_applied* returns True before applying.
- **annotate** – Run memlet propagation after application if necessary.
- **strict** – Apply transformation in strict mode.
- **save** – Save transformation as part of the SDFG file. Set to False if composing transformations.
- **where** – A dictionary of node names (from the transformation) to nodes in the SDFG or a single state.

can_be_applied(*graph: Union[dace.sdfg.sdfg.SDFG, dace.sdfg.state.SDFGState], candidate: Dict[PatternNode, int], expr_index: int, sdfg: dace.sdfg.sdfg.SDFG, strict: bool = False*) → bool

Returns True if this transformation can be applied on the candidate matched subgraph. :param graph: SDFGState object if this Transformation is

single-state, or SDFG object otherwise.

Parameters

- **candidate** – A mapping between node IDs returned from *Transformation.expressions* and the nodes in *graph*.
- **expr_index** – The list index from *Transformation.expressions* that was matched.
- **sdfg** – If *graph* is an SDFGState, its parent SDFG. Otherwise should be equal to *graph*.
- **strict** – Whether transformation should run in strict mode.

Returns True if the transformation can be applied.

expr_index

Object property of type int

expressions() → List[dace.sdfg.graph.SubgraphView]

Returns a list of Graph objects that will be matched in the subgraph isomorphism phase. Used as a pre-pass before calling *can_be_applied*. :see: Transformation.can_be_applied

extensions()

static from_json(*json_obj: Dict[str, Any], context: Dict[str, Any] = None*) → dace.transformation.transformation.Transformation

match_to_str(*graph: Union[dace.sdfg.sdfg.SDFG, dace.sdfg.state.SDFGState], candidate: Dict[PatternNode, int]*) → str

Returns a string representation of the pattern match on the candidate subgraph. Used when identifying matches in the console UI.

print_match(*sdfg: dace.sdfg.sdfg.SDFG*) → str

Returns a string representation of the pattern match on the given SDFG. Used for printing matches in the console UI.

properties()

register(***kwargs*)

sdfg_id
Object property of type int

state_id
Object property of type int

subgraph

to_json(*parent=None*) → Dict[str, Any]

unregister()

class dace.transformation.transformation.**TransformationBase**
Bases: object

Base class for data-centric transformations.

dace.transformation.transformation.**strict_transformations**() → List[Type[dace.transformation.transformation.Tr

Returns List of all registered strict transformations.

dace.transformation.helpers module

Transformation helper API.

dace.transformation.helpers.**are_subsets_contiguous**(*subset_a: dace.subsets.Subset*, *subset_b: dace.subsets.Subset*, *dim: int = None*) → bool

dace.transformation.helpers.**constant_symbols**(*sdfg: dace.sdfg.sdfg.SDFG*) → Set[str]
Returns a set of symbols that will never change values throughout the course of the given SDFG. Specifically, these are the input symbols (i.e., not defined in a particular scope) that are never set by interstate edges. :param sdfg: The input SDFG. :return: A set of symbol names that remain constant throughout the SDFG.

dace.transformation.helpers.**contained_in**(*state: dace.sdfg.state.SDFGState*, *node: dace.sdfg.nodes.Node*, *scope: dace.sdfg.nodes.EntryNode*) → bool
Returns true if the specified node is contained within the scope opened by the given entry node (including through nested SDFGs).

dace.transformation.helpers.**extract_map_dims**(*sdfg: dace.sdfg.sdfg.SDFG*, *map_entry: dace.sdfg.nodes.MapEntry*, *dims: List[int]*) → Tuple[dace.sdfg.nodes.MapEntry, dace.sdfg.nodes.MapEntry]
Helper function that extracts specific map dimensions into an outer map. :param sdfg: The SDFG where the map resides. :param map_entry: Map entry node to extract. :param dims: A list of dimension indices to extract. :return: A 2-tuple containing the extracted map and the remainder map.

dace.transformation.helpers.**find_contiguous_subsets**(*subset_list: List[dace.subsets.Subset]*, *dim: int = None*) → Set[dace.subsets.Subset]
Finds the set of largest contiguous subsets in a list of subsets. :param subsets: Iterable of subset objects. :param dim: Check for contiguity only for the specified dimension. :return: A list of contiguous subsets.

dace.transformation.helpers.**get_internal_scopes**(*state: dace.sdfg.state.SDFGState*, *entry: dace.sdfg.nodes.EntryNode*, *immediate: bool = False*) → List[Tuple[dace.sdfg.state.SDFGState, dace.sdfg.nodes.EntryNode]]

Returns all internal scopes within a given scope, including if they reside in nested SDFGs. :param state: State in which entry node resides. :param entry: The entry node to start from. :param immediate: If True, only returns the scopes that are immediately

nested in the map.

dace.transformation.helpers.**get_parent_map**(*state: dace.sdfg.state.SDFGState*, *node: Optional[dace.sdfg.nodes.Node] = None*) → Optional[Tuple[dace.sdfg.nodes.EntryNode, dace.sdfg.state.SDFGState]]

Returns the map in which the state (and node) are contained in, or None if it is free. :param state: The state to test or parent of the node to test. :param node: The node to test (optional). :return: A tuple of (entry node, state) or None.

dace.transformation.helpers.**gpu_map_has_explicit_threadblocks**(*state: dace.sdfg.state.SDFGState*, *entry: dace.sdfg.nodes.EntryNode*) → bool

Returns True if GPU_Device map has explicit thread-block maps nested within.

dace.transformation.helpers.**is_symbol_unused**(*sdfg: dace.sdfg.sdfg.SDFG*, *sym: str*) → bool

Checks for uses of symbol in an SDFG, and if there are none returns False. :param sdfg: The SDFG to search. :param sym: The symbol to test. :return: True if the symbol can be removed, False otherwise.

dace.transformation.helpers.**nest_state_subgraph**(*sdfg: dace.sdfg.sdfg.SDFG*, *state: dace.sdfg.state.SDFGState*, *subgraph: dace.sdfg.graph.SubgraphView*, *name: Optional[str] = None*, *full_data: bool = False*) → dace.sdfg.nodes.NestedSDFG

Turns a state subgraph into a nested SDFG. Operates in-place. :param sdfg: The SDFG containing the state subgraph. :param state: The state containing the subgraph. :param subgraph: Subgraph to nest. :param name: An optional name for the nested SDFG. :param full_data: If True, nests entire input/output data. :return: The nested SDFG node. :raise KeyError: Some or all nodes in the subgraph are not located in

this state, or the state does not belong to the given SDFG.

Raises **ValueError** – The subgraph is contained in more than one scope.

dace.transformation.helpers.**offset_map**(*sdfg: dace.sdfg.sdfg.SDFG, state: dace.sdfg.state.SDFGState, entry: dace.sdfg.nodes.MapEntry, dim: int, offset: Union[sympy.core.basic.Basic, dace.symbolic.SymExpr], negative: bool = True*)

Offsets a map parameter and its contents by a value. :param sdfg: The SDFG in which the map resides. :param state: The state in which the map resides. :param entry: The map entry node. :param dim: The map dimension to offset. :param offset: The value to offset by. :param negative: If True, offsets by `-offset`.

dace.transformation.helpers.**permute_map**(*map_entry: dace.sdfg.nodes.MapEntry, perm: List[int]*)

Permutes indices of a map according to a given list of integers.

dace.transformation.helpers.**reconnect_edge_through_map**(*state: dace.sdfg.state.SDFGState, edge: dace.sdfg.graph.MultiConnectorEdge[dace.memlet.M new_node: Union[dace.sdfg.nodes.EntryNode, dace.sdfg.nodes.ExitNode], keep_src: bool*) → Tuple[dace.sdfg.graph.MultiConnectorEdge[dace.meml dace.sdfg.graph.MultiConnectorEdge[dace.memlet.M

Reconnects an edge through a map scope, removes old edge, and returns the two new edges. :param state: The state in which the edge and map reside. :param edge: The edge to reconnect and remove. :param new_node: The scope (map) entry or exit to reconnect through. :param keep_src: If True, keeps the source of the edge intact, otherwise

keeps destination of edge.

Returns A 2-tuple of (incoming edge, outgoing edge).

dace.transformation.helpers.**redirect_edge**(*state: dace.sdfg.state.SDFGState, edge: dace.sdfg.graph.MultiConnectorEdge[dace.memlet.Memlet][dace.memle new_src: Optional[dace.sdfg.nodes.Node] = None, new_dst: Optional[dace.sdfg.nodes.Node] = None, new_src_conn: Optional[str] = None, new_dst_conn: Optional[str] = None, new_data: Optional[str] = None, new_memlet: Optional[dace.memlet.Memlet] = None*) → dace.sdfg.graph.MultiConnectorEdge[dace.memlet.Memlet][dace.memle

Redirects an edge in a state. Choose which elements to override by setting the keyword arguments. :param state: The SDFG state in which the edge resides. :param edge: The edge to redirect. :param new_src: If provided, redirects the source of the new edge. :param new_dst: If provided, redirects the destination of the new edge. :param new_src_conn: If provided, renames the source connector of the edge. :param new_dst_conn: If provided, renames the destination connector of the

edge.

Parameters

- **new_data** – If provided, changes the data on the memlet of the edge, and the entire associated memlet tree.
- **new_memlet** – If provided, changes only the memlet of the new edge.

Returns The new, redirected edge.

Note `new_data` and `new_memlet` cannot be used at the same time.

dace.transformation.helpers.**replicate_scope**(*sdfg: dace.sdfg.sdfg.SDFG, state: dace.sdfg.state.SDFGState, scope: dace.sdfg.scope.ScopeSubgraphView*) → dace.sdfg.scope.ScopeSubgraphView

Replicates a scope subgraph view within a state, reconnecting all external edges to the same nodes. :param sdfg: The SDFG in which the subgraph scope resides. :param state: The SDFG state in which the subgraph scope resides. :param scope: The scope subgraph to replicate. :return: A reconnected replica of the scope.

dace.transformation.helpers.**scope_tree_recursive**(*state: dace.sdfg.state.SDFGState*, *entry: Optional[dace.sdfg.nodes.EntryNode] = None*) → dace.sdfg.scope.ScopeTree

Returns a scope tree that includes scopes from nested SDFGs. :param state: The state that contains the root of the scope tree. :param entry: A scope entry node to set as root, otherwise the state is

the root if None is given.

dace.transformation.helpers.**simplify_state**(*state: dace.sdfg.state.SDFGState*, *remove_views: bool = False*) → networkx.classes.multidigraph.MultiDiGraph

Returns a networkx MultiDiGraph object that contains all the access nodes and corresponding edges of an SDFG state. The removed code nodes and map scopes are replaced by edges that connect their ancestor and succesor access nodes. :param state: The input SDFG state. :return: The MultiDiGraph object.

dace.transformation.helpers.**split_interstate_edges**(*sdfg: dace.sdfg.sdfg.SDFG*) → None

Splits all inter-state edges into edges with conditions and edges with assignments. This procedure helps in nested loop detection. :param sdfg: The SDFG to split :note: Operates in-place on the SDFG.

dace.transformation.helpers.**state_fission**(*sdfg: dace.sdfg.sdfg.SDFG*, *subgraph: dace.sdfg.graph.SubgraphView*) → dace.sdfg.state.SDFGState

Given a subgraph, adds a new SDFG state before the state that contains it, removes the subgraph from the original state, and connects the two states. :param subgraph: the subgraph to remove. :return: the newly created SDFG state.

dace.transformation.helpers.**tile**(*sdfg: dace.sdfg.sdfg.SDFG*, *map_entry: dace.sdfg.nodes.MapEntry*, *divides_evenly: bool*, *skew: bool*, ***tile_sizes*)

Helper function that tiles a Map scope by the given sizes, in the given order. :param sdfg: The SDFG where the map resides. :param map_entry: The map entry node to tile. :param divides_evenly: If True, skips pre/postamble for cases

where the map dimension is not a multiplier of the tile size.

Parameters

- **skew** – If True, skews the tiled map to start from zero. Helps compilers improve performance in certain cases.
- **tile_sizes** – An ordered dictionary of the map parameter names to tile and their respective tile size (which can be symbolic expressions).

dace.transformation.helpers.**unsqueeze_memlet**(*internal_memlet: dace.memlet.Memlet*, *external_memlet: dace.memlet.Memlet*, *preserve_minima: bool = False*, *use_src_subset: bool = False*, *use_dst_subset: bool = False*) → dace.memlet.Memlet

Unsqueezes and offsets a memlet, as per the semantics of nested SDFGs. :param internal_memlet: The internal memlet (inside nested SDFG)

before modification.

Parameters

- **external_memlet** – The external memlet before modification.
- **preserve_minima** – Do not change the subset's minimum elements.

- **use_src_subset** – If both sides of the memlet refer to same array, prefer source subset.
- **use_dst_subset** – If both sides of the memlet refer to same array, prefer destination subset.

Returns Offset Memlet to set on the resulting graph.

dace.transformation.pattern_matching module

Contains functions related to pattern matching in transformations.

dace.transformation.pattern_matching.**collapse_multigraph_to_nx**(*graph: Union[dace.sdfg.graph.MultiDiGraph, dace.sdfg.graph.OrderedMultiDiGraph]*) → networkx.classes.digraph.DiGraph

Collapses a directed multigraph into a networkx directed graph.

In the output directed graph, each node is a number, which contains itself as node_data['node'], while each edge contains a list of the data from the original edges as its attribute (edge_data[0...N]).

Parameters **graph** – Directed multigraph object to be collapsed.

Returns Collapsed directed graph object.

dace.transformation.pattern_matching.**enumerate_matches**(*sdfg: dace.sdfg.sdfg.SDFG*, *pattern: dace.sdfg.graph.Graph*, *node_match=<function type_or_class_match>*, *edge_match=None*) → Iterator[dace.sdfg.graph.SubgraphView]

Returns a generator of subgraphs that match the given subgraph pattern. :param sdfg: The SDFG to search in. :param pattern: A subgraph to look for. :param node_match: An optional function to use for matching nodes. :param node_match: An optional function to use for matching edges. :return: Yields SDFG subgraph view objects.

dace.transformation.pattern_matching.**get_transformation_metadata**(*patterns: List[Type[dace.transformation.transfo*, *options: Optional[List[Dict[str, Any]]]] = None*) → Tuple[List[Tuple[Type[dace.transformati int, networkx.classes.digraph.DiGraph, Callable, Dict[str, Any]]], List[Tuple[Type[dace.transformation.t int, networkx.classes.digraph.DiGraph, Callable, Dict[str, Any]]]]

Collect all transformation expressions and metadata once, for use when applying transformations repeatedly. :param patterns: Transformation type (or list thereof) to compute. :param options: An optional list of transformation parameter dictionaries. :return: A tuple of inter-state and single-state pattern matching

transformations.

dace.transformation.pattern_matching.**match_patterns**(*sdfg: dace.sdfg.sdfg.SDFG, patterns: Union[Type[dace.transformation.transformation.Transfo List[Type[dace.transformation.transformation.Transforma node_match: Callable[[Any, Any], bool] = <function type_match>, edge_match: Optional[Callable[[Any, Any], bool]] = None, strict: bool = False, metadata: Optional[Tuple[List[Tuple[Type[dace.transformation.transfo int, networkx.classes.digraph.DiGraph, Callable, Dict[str, Any]]], List[Tuple[Type[dace.transformation.transformation.Tra int, networkx.classes.digraph.DiGraph, Callable, Dict[str, Any]]]]] = None, states: Optional[List[dace.sdfg.state.SDFGState]] = None, options: Optional[List[Dict[str, Any]]] = None*)

Returns a generator of Transformations that match the input SDFG. Ordered by SDFG ID. :param sdfg: The SDFG to match in. :param patterns: Transformation type (or list thereof) to match. :param node_match: Function for checking whether two nodes match. :param edge_match: Function for checking whether two edges match. :param strict: Only match transformation if strict (i.e., can only

improve the performance/reduce complexity of the SDFG).

Parameters

- **metadata** – Transformation metadata that can be reused.
- **states** – If given, only tries to match single-state transformations on this list.
- **options** – An optional iterable of transformation parameter dictionaries.

Returns A list of Transformation objects that match.

dace.transformation.pattern_matching.**type_match**(*graph_node*, *pattern_node*)

Checks whether the node types of the inputs match. :param graph_node: First node (in matched graph). :param pattern_node: Second node (in pattern subgraph). :return: True if the object types of the nodes match, False otherwise. :raise TypeError: When at least one of the inputs is not a dictionary

or does not have a 'node' attribute.

Raises **KeyError** – When at least one of the inputs is a dictionary, but does not have a 'node' key.

dace.transformation.pattern_matching.**type_or_class_match**(*node_a*, *node_b*)

Checks whether *node_a* is an instance of the same type as *node_b*, or if either *node_a*/*node_b* is a type and the

other is an instance of that type. This is used in subgraph matching to allow the subgraph pattern to be either a graph of instantiated nodes, or node types.

Parameters

- **node_a** – First node.
- **node_b** – Second node.

Returns True if the object types of the nodes match according to the description, False otherwise.

Raises

- **TypeError** – When at least one of the inputs is not a dictionary or does not have a 'node' attribute.
- **KeyError** – When at least one of the inputs is a dictionary, but does not have a 'node' key.

See enumerate_matches

dace.transformation.optimizer module

Contains classes and functions related to optimization of the stateful dataflow graph representation.

class dace.transformation.optimizer.**Optimizer**(*sdfg*, *inplace=False*)
Bases: object

Implements methods for optimizing a DaCe program stateful dataflow graph representation, by matching patterns and applying transformations on it.

get_pattern_matches(*strict=False*, *states=None*, *patterns=None*, *sdfg=None*, *options=None*) → Iterator[dace.transformation.transformation.Transformation]
Returns all possible transformations for the current SDFG. :param strict: Only consider strict transformations (i.e., ones

that surely increase performance or enhance readability)

Parameters

- **states** – An iterable of SDFG states to consider when pattern matching. If None, considers all.
- **patterns** – An iterable of transformation classes to consider when matching. If None, considers all registered transformations in Transformation.
- **sdfg** – If not None, searches for patterns on given SDFG.
- **options** – An optional iterable of transformation parameters.

Returns List of matching Transformation objects.

See Transformation.

optimization_space()
Returns the optimization space of the current SDFG

optimize()

set_transformation_metadata(*patterns: List[Type[dace.transformation.transformation.Transformation]]*, *options: Optional[List[Dict[str, Any]]] = None*)
Caches transformation metadata for a certain set of patterns to match.

class dace.transformation.optimizer.**SDFGOptimizer**(*sdfg*, *inplace=False*)
Bases: *dace.transformation.optimizer.Optimizer*

optimize()
A command-line UI for applying patterns on the SDFG. :return: An optimized SDFG object

dace.transformation.testing module

class dace.transformation.testing.**TransformationTester**(*sdfg: dace.sdfg.sdfg.SDFG*, *depth=1*, *validate=True*, *generate_code=True*, *compile=False*, *print_exception=True*, *halt_on_exception=False*)
Bases: *dace.transformation.optimizer.Optimizer*

An SDFG optimizer that consecutively applies available transformations up to a fixed depth.

optimize()

Module contents

1.1.2 Submodules

1.1.3 dace.config module

class dace.config.**Config**
Bases: object

Interface to the DaCe hierarchical configuration file.

static append(**key_hierarchy*, *value=None*, *autosave=False*)
Appends to the current value of a given configuration entry and sets it. Example usage: *Config.append('compiler', 'cpu', 'args', value='-fPIC')* :param key_hierarchy: A tuple of strings leading to the

configuration entry. For example: ('a', 'b', 'c') would be configuration entry c which is in the path a->b.

Parameters

- **value** – The value to append.
- **autosave** – If True, saves the configuration to the file after modification.

Returns Current configuration entry value.

static cfg_filename()
Returns the current configuration file path.

static get(**key_hierarchy*)
Returns the current value of a given configuration entry. :param key_hierarchy: A tuple of strings leading to the

configuration entry. For example: ('a', 'b', 'c') would be configuration entry c which is in the path a->b.

Returns Configuration entry value.

static get_bool(**key_hierarchy*)
Returns the current value of a given boolean configuration entry. This specialization allows more string types to be converted to boolean, e.g., due to environment variable overrides. :param key_hierarchy: A tuple of strings leading to the

configuration entry. For example: ('a', 'b', 'c') would be configuration entry c which is in the path a->b.

Returns Configuration entry value (as a boolean).

static get_default(**key_hierarchy*)
Returns the default value of a given configuration entry. Takes into accound current operating system. :param key_hierarchy: A tuple of strings leading to the

configuration entry. For example: ('a', 'b', 'c') would be configuration entry c which is in the path a->b.

Returns Default configuration value.

static get_metadata(**key_hierarchy*)
Returns the configuration specification of a given entry from the schema. :param key_hierarchy: A tuple of strings leading to the

configuration entry. For example: ('a', 'b', 'c') would be configuration entry c which is in the path a->b.

Returns Configuration specification as a dictionary.

static initialize()
Initializes configuration.

B{Note:} This function runs automatically when the module is loaded.

static load(*filename=None*)
Loads a configuration from an existing file. :param filename: The file to load. If unspecified,

uses default configuration file.

static load_schema(*filename=None*)
Loads a configuration schema from an existing file. :param filename: The file to load. If unspecified,

uses default schema file.

static save(*path=None*)
Saves the current configuration to a file. :param path: The file to save to. If unspecified,

uses default configuration file.

static set(**key_hierarchy*, *value=None*, *autosave=False*)
Sets the current value of a given configuration entry. Example usage: *Config.set('profiling', value=True)* :param key_hierarchy: A tuple of strings leading to the

configuration entry. For example: ('a', 'b', 'c') would be configuration entry c which is in the path a->b.

Parameters

- **value** – The value to set.
- **autosave** – If True, saves the configuration to the file after modification.

dace.config.**set_temporary**(**path*, *value*)
Temporarily set configuration value at `path` to value, and reset it after the context manager exits.

Example print(Config.get("compiler", "build_type") with set_temporary("compiler", "build_type", value="Debug"):

print(Config.get("compiler", "build_type")

print(Config.get("compiler", "build_type")

dace.config.**temporary_config**()
Creates a context where all configuration options changed will be reset when the context exits.

with temporary_config(): Config.set("testing", "serialization", value=True) Config.set("optimizer", "autooptimize", value=True) foo()

1.1.4 dace.data module

class dace.data.**Array**(**args*, ***kwargs*)
Bases: *dace.data.Data*

Array/constant descriptor (dimensions, type and other properties).

alignment
Allocation alignment in bytes (0 uses compiler-default)

allow_conflicts
If enabled, allows more than one memlet to write to the same memory location without conflict resolution.

as_arg(*with_types=True*, *for_call=False*, *name=None*)
Returns a string for a C++ function signature (e.g., *int *A*).

clone()

covers_range(*rng*)

free_symbols
Returns a set of undefined symbols in this data descriptor.

classmethod from_json(*json_obj*, *context=None*)

is_equivalent(*other*)
Check for equivalence (shape and type) of two data descriptors.

may_alias
This pointer may alias with other pointers in the same function

offset
Initial offset to translate all indices by.

properties()

sizes()

strides
For each dimension, the number of elements to skip in order to obtain the next element in that dimension.

to_json()

total_size
The total allocated size of the array. Can be used for padding.

validate()
Validate the correctness of this object. Raises an exception on error.

class dace.data.**Data**(**args*, ***kwargs*)
Bases: object

Data type descriptors that can be used as references to memory. Examples: Arrays, Streams, custom arrays (e.g., sparse matrices).

as_arg(*with_types=True*, *for_call=False*, *name=None*)
Returns a string for a C++ function signature (e.g., *int *A*).

copy()

ctype

debuginfo
Object property of type DebugInfo

dtype
Object property of type typeclass

free_symbols
Returns a set of undefined symbols in this data descriptor.

is_equivalent(*other*)
Check for equivalence (shape and type) of two data descriptors.

lifetime
Data allocation span

location
Full storage location identifier (e.g., rank, GPU ID)

properties()

set_strides_from_layout(**dimensions*, *alignment: Union[sympy.core.basic.Basic, dace.symbolic.SymExpr] = 1*, *only_first_aligned: bool = False*)
Sets the absolute strides and total size of this data descriptor, according to the given dimension ordering and alignment. :param dimensions: A sequence of integers representing a permutation

of the descriptor's dimensions.

Parameters

- **alignment** – Padding (in elements) at the end, ensuring stride is a multiple of this number. 1 (default) means no padding.
- **only_first_aligned** – If True, only the first dimension is padded with `alignment`. Otherwise all dimensions are.

shape
Object property of type tuple

storage
Storage location

strides_from_layout(**dimensions*, *alignment: Union[sympy.core.basic.Basic, dace.symbolic.SymExpr] = 1*, *only_first_aligned: bool = False*) → Tuple[Tuple[Union[sympy.core.basic.Basic, dace.symbolic.SymExpr]], Union[sympy.core.basic.Basic, dace.symbolic.SymExpr]]
Returns the absolute strides and total size of this data descriptor, according to the given dimension ordering and alignment. :param dimensions: A sequence of integers representing a permutation

of the descriptor's dimensions.

Parameters

- **alignment** – Padding (in elements) at the end, ensuring stride is a multiple of this number. 1 (default) means no padding.
- **only_first_aligned** – If True, only the first dimension is padded with `alignment`. Otherwise all dimensions are.

Returns A 2-tuple of (tuple of strides, total size).

to_json()

toplevel

transient
Object property of type bool

validate()
Validate the correctness of this object. Raises an exception on error.

veclen

class dace.data.**Scalar**(**args*, ***kwargs*)
Bases: *dace.data.Data*

Data descriptor of a scalar value.

allow_conflicts
Object property of type bool

as_arg(*with_types=True*, *for_call=False*, *name=None*)
Returns a string for a C++ function signature (e.g., *int *A*).

clone()

covers_range(*rng*)

static from_json(*json_obj*, *context=None*)

is_equivalent(*other*)
Check for equivalence (shape and type) of two data descriptors.

offset

properties()

sizes()

strides

total_size

class dace.data.**Stream**(**args*, ***kwargs*)
Bases: *dace.data.Data*

Stream (or stream array) data descriptor.

as_arg(*with_types=True*, *for_call=False*, *name=None*)
Returns a string for a C++ function signature (e.g., *int *A*).

buffer_size
Size of internal buffer.

clone()

covers_range(*rng*)

free_symbols
Returns a set of undefined symbols in this data descriptor.

classmethod from_json(*json_obj*, *context=None*)

is_equivalent(*other*)
Check for equivalence (shape and type) of two data descriptors.

is_stream_array()

offset
Object property of type list

properties()

size_string()

sizes()

strides

to_json()

total_size

class dace.data.**View**(**args*, ***kwargs*)
Bases: *dace.data.Array*

Data descriptor that acts as a reference (or view) of another array. Can be used to reshape or reinterpret existing data without copying it.

To use a View, it needs to be referenced in an access node that is directly connected to another access node. The rules for deciding which access node is viewed are:

- If there is one edge (in/out) that leads (via memlet path) to an access node, and the other side (out/in) has a different number of edges.
- If there is one incoming and one outgoing edge, and one leads to a code node, the one that leads to an access node is the viewed data.
- If both sides lead to access nodes, if one memlet's data points to the view it cannot point to the viewed node.
- If both memlets' data are the respective access nodes, the access node at the highest scope is the one that is viewed.
- If both access nodes reside in the same scope, the input data is viewed.

Other cases are ambiguous and will fail SDFG validation.

In the Python frontend, `numpy.reshape` and `numpy.ndarray.view` both generate Views.

as_array()

properties()

validate()
Validate the correctness of this object. Raises an exception on error.

dace.data.**create_datadescriptor**(*obj*)
Creates a data descriptor from various types of objects. @see: dace.data.Data

dace.data.**find_new_name**(*name: str, existing_names: Sequence[str]*) → str
Returns a name that matches the given `name` as a prefix, but does not already exist in the given existing name set. The behavior is typically to append an underscore followed by a unique (increasing) number. If the name does

not already exist in the set, it is returned as-is. :param name: The given name to find. :param existing_names: The set of existing names. :return: A new name that is not in existing_names.

1.1.5 dace.dtypes module

A module that contains various DaCe type definitions.

class dace.dtypes.**AccessType**(**args*, ***kwds*)
Bases: aenum.AutoNumberEnum

Types of access to an *AccessNode*.

ReadOnly = 1

ReadWrite = 3

Undefined = 4

WriteOnly = 2

class dace.dtypes.**AllocationLifetime**(**args*, ***kwds*)
Bases: aenum.AutoNumberEnum

Options for allocation span (when to allocate/deallocate) of data.

Global = 4
Allocated throughout the entire program (outer SDFG)

Persistent = 5
Allocated throughout multiple invocations (init/exit)

SDFG = 3
Allocated throughout the innermost SDFG (possibly nested)

Scope = 1
Allocated/Deallocated on innermost scope start/end

State = 2
Allocated throughout the containing state

Undefined = 6

register(**args*)

class dace.dtypes.**DebugInfo**(*start_line*, *start_column=0*, *end_line=-1*, *end_column=0*, *filename=None*)
Bases: object

Source code location identifier of a node/edge in an SDFG. Used for IDE and debugging purposes.

static from_json(*json_obj*, *context=None*)

to_json()

class dace.dtypes.**DeviceType**(**args*, ***kwds*)
Bases: aenum.AutoNumberEnum

CPU = 1
Multi-core CPU

FPGA = 3
FPGA (Intel or Xilinx)

GPU = 2
GPU (AMD or NVIDIA)

Undefined = 4

register(**args*)

class dace.dtypes.**InstrumentationType**(**args*, ***kwds*)
Bases: aenum.AutoNumberEnum

Types of instrumentation providers. @note: Might be determined automatically in future versions.

FPGA = 5

GPU_Events = 4

No_Instrumentation = 1

PAPI_Counters = 3

Timer = 2

Undefined = 6

register(**args*)

class dace.dtypes.**Language**(**args*, ***kwds*)
Bases: aenum.AutoNumberEnum

Available programming languages for SDFG tasklets.

CPP = 2

MLIR = 5

OpenCL = 3

Python = 1

SystemVerilog = 4

Undefined = 6

register(**args*)

class dace.dtypes.**ReductionType**(**args*, ***kwds*)
Bases: aenum.AutoNumberEnum

Reduction types natively supported by the SDFG compiler.

Bitwise_And = 7
Bitwise AND (&)

Bitwise_Or = 9
Bitwise OR (|)

Bitwise_Xor = 11
Bitwise XOR (^)

Custom = 1
Defined by an arbitrary lambda function

Div = 16
Division (only supported in OpenMP)

Exchange = 14
Set new value, return old value

Logical_And = 6
Logical AND (&&)

Logical_Or = 8
Logical OR (||)

Logical_Xor = 10
Logical XOR (!=)

Max = 3
Maximum value

Max_Location = 13
Maximum value and its location

Min = 2
Minimum value

Min_Location = 12
Minimum value and its location

Product = 5
Product

Sub = 15
Subtraction (only supported in OpenMP)

Sum = 4
Sum

Undefined = 17

class dace.dtypes.**ScheduleType**(**args*, ***kwds*)
Bases: aenum.AutoNumberEnum

Available map schedule types in the SDFG.

CPU_Multicore = 4
OpenMP

Default = 1
Scope-default parallel schedule

FPGA_Device = 12

GPU_Default = 7
Default scope schedule for GPU code. Specializes to schedule GPU_Device and GPU_Global during inference.

GPU_Device = 8
Kernel

GPU_Persistent = 11

GPU_ThreadBlock = 9
Thread-block code

GPU_ThreadBlock_Dynamic = 10
Allows rescheduling work within a block

MPI = 3
MPI processes

SVE_Map = 6
Arm SVE

Sequential = 2
Sequential code (single-thread)

Undefined = 13

Unrolled = 5
Unrolled code

register(**args*)

class dace.dtypes.**StorageType**(**args*, ***kwds*)
Bases: aenum.AutoNumberEnum

Available data storage types in the SDFG.

CPU_Heap = 4
Host memory allocated on heap

CPU_Pinned = 3
Host memory that can be DMA-accessed from accelerators

CPU_ThreadLocal = 5
Thread-local host memory

Default = 1
Scope-default storage location

FPGA_Global = 8
Off-chip global memory (DRAM)

FPGA_Local = 9
On-chip memory (bulk storage)

FPGA_Registers = 10
On-chip memory (fully partitioned registers)

FPGA_ShiftRegister = 11
Only accessible at constant indices

GPU_Global = 6
Global memory

GPU_Shared = 7
Shared memory

Register = 2
Local data on registers, stack, or equivalent memory

Undefined = 12

register(**args*)

class dace.dtypes.**TilingType**(**args*, ***kwds*)
Bases: aenum.AutoNumberEnum

Available tiling types in a *StripMining* transformation.

CeilRange = 2

Normal = 1

NumberOfTiles = 3

Undefined = 4

register(**args*)

class dace.dtypes.**Typeclasses**(**args*, ***kwds*)
Bases: aenum.AutoNumberEnum

Undefined = 16

bool = 1

bool_ = 2

complex128 = 15

complex64 = 14

float16 = 11

float32 = 12

float64 = 13

int16 = 4

int32 = 5

int64 = 6

int8 = 3

register(**args*)

uint16 = 8

uint32 = 9

uint64 = 10

uint8 = 7

class dace.dtypes.**callback**(*return_type*, **variadic_args*)
Bases: *dace.dtypes.typeclass*

Looks like dace.callback([None, <some_native_type>], *types)

as_arg(*name*)

as_ctypes()
Returns the ctypes version of the typeclass.

as_numpy_dtype()

static from_json(*json_obj*, *context=None*)

get_trampoline(*pyfunc*, *other_arguments*)

to_json()

dace.dtypes.**can_access**(*schedule: dace.dtypes.ScheduleType*, *storage: dace.dtypes.StorageType*)
Identifies whether a container of a storage type can be accessed in a specific schedule.

dace.dtypes.**can_allocate**(*storage: dace.dtypes.StorageType*, *schedule: dace.dtypes.ScheduleType*)
Identifies whether a container of a storage type can be allocated in a specific schedule. Used to determine arguments to subgraphs by the innermost scope that a container can be allocated in. For example, FPGA_Global memory cannot be allocated from within the FPGA scope, or GPU shared memory cannot be allocated outside of device-level code.

Parameters

- **storage** – The storage type of the data container to allocate.
- **schedule** – The scope schedule to query.

Returns True if the container can be allocated, False otherwise.

class dace.dtypes.**constant**
Bases: object

Data descriptor type hint signalling that argument evaluation is deferred to call time.

Example usage:

```
@dace.program
def example(A: dace.float64[20], constant: dace.constant):
    if constant == 0:
        return A + 1
    else:
        return A + 2
```

In the above code, constant will be replaced with its value at call time during parsing.

dace.dtypes.**deduplicate**(*iterable*)
Removes duplicates in the passed iterable.

dace.dtypes.**is_array**(*obj: Any*) → bool
Returns True if an object implements the data_ptr(), __array_interface__ or __cuda_array_interface__ standards (supported by NumPy, Numba, CuPy, PyTorch, etc.). If the interface is supported, pointers can be directly obtained using the _array_interface_ptr function.

Parameters **obj** – The given object.

Returns True iff the object implements the array interface.

dace.dtypes.**isallowed**(*var*, *allow_recursive=False*)
Returns True if a given object is allowed in a DaCe program.

Parameters **allow_recursive** – whether to allow dicts or lists containing constants.

dace.dtypes.**isconstant**(*var*)
Returns True if a variable is designated a constant (i.e., that can be directly generated in code).

dace.dtypes.**ismodule**(*var*)
Returns True if a given object is a module.

dace.dtypes.**ismodule_and_allowed**(*var*)
Returns True if a given object is a module and is one of the allowed modules in DaCe programs.

dace.dtypes.**ismoduleallowed**(*var*)
Helper function to determine the source module of an object, and whether it is allowed in DaCe programs.

dace.dtypes.**json_to_typeclass**(*obj*, *context=None*)

dace.dtypes.**max_value**(*dtype: dace.dtypes.typeclass*)
Get a max value literal for *dtype*.

dace.dtypes.**min_value**(*dtype: dace.dtypes.typeclass*)
Get a min value literal for *dtype*.

class dace.dtypes.**opaque**(*typename*)
Bases: *dace.dtypes.typeclass*

A data type for an opaque object, useful for C bindings/libnodes, i.e., MPI_Request.

as_ctypes()
Returns the ctypes version of the typeclass.

as_numpy_dtype()

static from_json(*json_obj*, *context=None*)

to_json()

dace.dtypes.**paramdec**(*dec*)
Parameterized decorator meta-decorator. Enables using @*decorator*, @*decorator()*, and @*decorator(...)* with the same function.

class dace.dtypes.**pointer**(*wrapped_typeclass*)
Bases: *dace.dtypes.typeclass*

A data type for a pointer to an existing typeclass.

Example use: *dace.pointer(dace.struct(x=dace.float32, y=dace.float32))*.

as_ctypes()
Returns the ctypes version of the typeclass.

as_numpy_dtype()

base_type

static from_json(*json_obj*, *context=None*)

ocltype

to_json()

dace.dtypes.**ptrtonumpy**(*ptr*, *inner_ctype*, *shape*)

dace.dtypes.**reduction_identity**(*dtype: dace.dtypes.typeclass*, *red: dace.dtypes.ReductionType*) → Any
Returns known identity values (which we can safely reset transients to) for built-in reduction types. :param dtype: Input type. :param red: Reduction type. :return: Identity value in input type, or None if not found.

dace.dtypes.**result_type_of**(*lhs*, **rhs*)
Returns the largest between two or more types (dace.types.typeclass) according to C semantics.

class dace.dtypes.**struct**(*name*, ***fields_and_types*)
Bases: *dace.dtypes.typeclass*

A data type for a struct of existing typeclasses.

Example use: *dace.struct(a=dace.int32, b=dace.float64)*.

as_ctypes()
Returns the ctypes version of the typeclass.

as_numpy_dtype()

emit_definition()

fields

static from_json(*json_obj*, *context=None*)

to_json()

class dace.dtypes.**typeclass**(*wrapped_type*)
Bases: object

An extension of types that enables their use in DaCe.

These types are defined for three reasons:

1. Controlling DaCe types
2. Enabling declaration syntax: *dace.float32[M,N]*
3. Enabling extensions such as *dace.struct* and *dace.vector*

as_arg(*name*)

as_ctypes()
Returns the ctypes version of the typeclass.

as_numpy_dtype()

base_type

static from_json(*json_obj*, *context=None*)

is_complex()

ocltype

to_json()

to_string()
A Numpy-like string-representation of the underlying data type.

veclen

dace.dtypes.**validate_name**(*name*)

class dace.dtypes.**vector**(*dtype: dace.dtypes.typeclass*, *vector_length: int*)
Bases: *dace.dtypes.typeclass*

A data type for a vector-type of an existing typeclass.

Example use: *dace.vector(dace.float32, 4)* becomes float4.

as_ctypes()
Returns the ctypes version of the typeclass.

as_numpy_dtype()

base_type

ctype

ctype_unaligned

static from_json(*json_obj*, *context=None*)

ocltype

to_json()

veclen

1.1.6 dace.jupyter module

Jupyter Notebook support for DaCe.

dace.jupyter.**enable**()

dace.jupyter.**isnotebook**()

dace.jupyter.**preamble**()

1.1.7 dace.memlet module

class dace.memlet.**Memlet**(**args*, ***kwargs*)
Bases: object

Data movement object. Represents the data, the subset moved, and the manner it is reindexed (*other_subset*) into the destination. If there are multiple conflicting writes, this object also specifies how they are resolved with a lambda function.

allow_oob
Bypass out-of-bounds validation

bounding_box_size()
Returns a per-dimension upper bound on the maximum number of elements in each dimension.

This bound will be tight in the case of Range.

data
Data descriptor attached to this memlet

debuginfo
Line information to track source and generated code

dst_subset

dynamic
Is the number of elements moved determined at runtime (e.g., data dependent)

free_symbols
Returns a set of symbols used in this edge's properties.

static from_array(*dataname*, *datadesc*, *wcr=None*)
Constructs a Memlet that transfers an entire array's contents. :param dataname: The name of the data descriptor in the SDFG. :param datadesc: The data descriptor object. :param wcr: The conflict resolution lambda. :type datadesc: Data

static from_json(*json_obj*, *context=None*)

get_dst_subset(*edge: dace.sdfg.graph.MultiConnectorEdge*, *state: dace.sdfg.state.SDFGState*)

get_src_subset(*edge: dace.sdfg.graph.MultiConnectorEdge*, *state: dace.sdfg.state.SDFGState*)

is_empty() → bool
Returns True if this memlet carries no data. Memlets without data are primarily used for connecting nodes to scopes without transferring data to them.

num_accesses
Returns the total memory movement volume (in elements) of this memlet.

num_elements()
Returns the number of elements in the Memlet subset.

other_subset
Subset of elements after reindexing to the data not attached to this edge (e.g., for offsets and reshaping).

properties()

replace(*repl_dict*)
Substitute a given set of symbols with a different set of symbols. :param repl_dict: A dict of string symbol names to symbols with

which to replace them.

static simple(*data*, *subset_str*, *wcr_str=None*, *other_subset_str=None*, *wcr_conflict=True*, *num_accesses=None*, *debuginfo=None*, *dynamic=False*)

DEPRECATED: Constructs a Memlet from string-based expressions. :param data: The data object or name to access. :type data: Either a string of the data descriptor name or an

AccessNode.

Parameters

- **subset_str** – The subset of *data* that is going to be accessed in string format. Example: '0:N'.
- **wcr_str** – A lambda function (as a string) specifying how write-conflicts are resolved. The syntax of the lambda function receives two elements: *current* value and *new* value, and returns the value after resolution. For example, summation is *'lambda cur, new: cur + new'*.
- **other_subset_str** – The reindexing of *subset* on the other connected data (as a string).
- **wcr_conflict** – If False, forces non-locked conflict resolution when generating code. The default is to let the code generator infer this information from the SDFG.
- **num_accesses** – The number of times that the moved data will be subsequently accessed. If -1, designates that the number of accesses is unknown at compile time.
- **debuginfo** – Source-code information (e.g., line, file) used for debugging.
- **dynamic** – If True, the number of elements moved in this memlet is defined dynamically at runtime.

src_subset

subset

Subset of elements to move from the data attached to this edge.

to_json()

try_initialize(*sdfg: dace.sdfg.sdfg.SDFG*, *state: dace.sdfg.state.SDFGState*, *edge: dace.sdfg.graph.MultiConnectorEdge*)

Tries to initialize the internal fields of the memlet (e.g., src/dst subset) once it is added to an SDFG as an edge.

validate(*sdfg*, *state*)

volume

The exact number of elements moved using this memlet, or the maximum number if dynamic=True (with 0 as unbounded)

wcr

If set, defines a write-conflict resolution lambda function. The syntax of the lambda function receives two elements: *current* value and *new* value, and returns the value after resolution

wcr_nonatomic

If True, always generates non-conflicting (non-atomic) writes in resulting code

class dace.memlet.**MemletTree**(*edge*, *parent=None*, *children=None*)

Bases: object

A tree of memlet edges.

Since memlets can form paths through scope nodes, and since these paths can split in "OUT_*" connectors, a memlet edge can be extended to a memlet tree. The tree is always rooted at the outermost-scope node, which

can mean that it forms a tree of directed edges going forward (in the case where memlets go through scope-entry nodes) or backward (through scope-exit nodes).

Memlet trees can be used to obtain all edges pertaining to a single memlet using the *memlet_tree* function in SDFGState. This collects all siblings of the same edge and their children, for instance if multiple inputs from the same access node are used.

root() → dace.memlet.MemletTree

traverse_children(*include_self=False*)

1.1.8 dace.properties module

class dace.properties.**CodeBlock**(*code: Union[str, List[_ast.AST], CodeBlock], language: dace.dtypes.Language = <Language.Python: 1>*)

Bases: object

Helper class that represents code blocks with language. Used in *CodeProperty*, implemented as a list of AST statements if language is Python, or a string otherwise.

as_string

static from_json(*tmp*, *sdfg=None*)

get_free_symbols(*defined_syms: Set[str] = None*) → Set[str]

Returns the set of free symbol names in this code block, excluding the given symbol names.

to_json()

class dace.properties.**CodeProperty**(*getter=None*, *setter=None*, *dtype=None*, *default=None*, *from_string=None*, *to_string=None*, *from_json=None*, *to_json=None*, *meta_to_json=None*, *choices=None*, *unmapped=False*, *allow_none=False*, *indirected=False*, *category='General'*, *desc=''*)

Bases: *dace.properties.Property*

Custom Property type that accepts code in various languages.

dtype

from_json(*tmp*, *sdfg=None*)

static from_string(*string*, *language=None*)

to_json(*obj*)

static to_string(*obj*)

class dace.properties.**DataProperty**(*desc=''*, *default=None*, ***kwargs*)

Bases: *dace.properties.Property*

Custom Property type that represents a link to a data descriptor. Needs the SDFG to be passed as an argument to *from_string* and *choices*.

static choices(*sdfg=None*)

from_json(*s*, *context=None*)

static from_string(*s*, *sdfg=None*)

to_json(*obj*)

static to_string(*obj*)

typestring()

class dace.properties.**DataclassProperty**(*getter=None*, *setter=None*, *dtype=None*, *default=None*, *from_string=None*, *to_string=None*, *from_json=None*, *to_json=None*, *meta_to_json=None*, *choices=None*, *unmapped=False*, *allow_none=False*, *indirected=False*, *category='General'*, *desc=''*)

Bases: *dace.properties.Property*

Property that stores pydantic models or dataclasses.

from_json(*d*, *sdfg=None*)

static from_string(*s*)

to_json(*obj*)

static to_string(*obj*)

class dace.properties.**DebugInfoProperty**(***kwargs*)

Bases: *dace.properties.Property*

Custom Property type for DebugInfo members.

allow_none

dtype

static from_string(*s*)

static to_string(*di*)

class dace.properties.**DictProperty**(*key_type*, *value_type*, **args*, ***kwargs*)

Bases: *dace.properties.Property*

Property type for dictionaries.

from_json(*data*, *sdfg=None*)

static from_string(*s*)

to_json(*d*)

static to_string(*d*)

class dace.properties.**EnumProperty**(*dtype*, **args*, ***kwargs*)

Bases: *dace.properties.Property*

class dace.properties.**LambdaProperty**(*getter=None*, *setter=None*, *dtype=None*, *default=None*, *from_string=None*, *to_string=None*, *from_json=None*, *to_json=None*, *meta_to_json=None*, *choices=None*, *unmapped=False*, *allow_none=False*, *indirected=False*, *category='General'*, *desc=''*)

Bases: *dace.properties.Property*

Custom Property type that accepts a lambda function, with conversions to and from strings.

dtype

from_json(*s*, *sdfg=None*)

static from_string(*s*)

to_json(*obj*)

static to_string(*obj*)

class dace.properties.**LibraryImplementationProperty**(*getter=None*, *setter=None*, *dtype=None*, *default=None*, *from_string=None*, *to_string=None*, *from_json=None*, *to_json=None*, *meta_to_json=None*, *choices=None*, *unmapped=False*, *allow_none=False*, *indirected=False*, *category='General'*, *desc=''*)

Bases: *dace.properties.Property*

Property for choosing an implementation type for a library node. On the Python side it is a standard property, but can expand into a combo-box in DIODE.

typestring()

class dace.properties.**ListProperty**(*element_type*, **args*, ***kwargs*)

Bases: *dace.properties.Property*

Property type for lists.

from_json(*data*, *sdfg=None*)

from_string(*s*)

to_json(*l*)

static to_string(*l*)

class dace.properties.**OrderedDictProperty**(*getter=None*, *setter=None*, *dtype=None*, *default=None*, *from_string=None*, *to_string=None*, *from_json=None*, *to_json=None*, *meta_to_json=None*, *choices=None*, *unmapped=False*, *allow_none=False*, *indirected=False*, *category='General'*, *desc=''*)

Bases: *dace.properties.Property*

Property type for ordered dicts

static from_json(*obj*, *sdfg=None*)

to_json(*d*)

class dace.properties.**Property**(*getter=None*, *setter=None*, *dtype=None*, *default=None*, *from_string=None*, *to_string=None*, *from_json=None*, *to_json=None*, *meta_to_json=None*, *choices=None*, *unmapped=False*, *allow_none=False*, *indirected=False*, *category='General'*, *desc=''*)

Bases: object

Class implementing properties of DaCe objects that conform to strong typing, and allow conversion to and from strings to be edited.

static add_none_pair(*dict_in*)

allow_none

category

choices

default

desc

dtype

from_json

from_string

static get_property_element(*object_with_properties*, *name*)

getter

indirected

meta_to_json
Returns a function to export meta information (type, description, default value).

setter

to_json

to_string

typestring()

unmapped

exception dace.properties.**PropertyError**
Bases: Exception

Exception type for errors related to internal functionality of these properties.

class dace.properties.**RangeProperty**(*getter=None*, *setter=None*, *dtype=None*, *default=None*, *from_string=None*, *to_string=None*, *from_json=None*, *to_json=None*, *meta_to_json=None*, *choices=None*, *unmapped=False*, *allow_none=False*, *indirected=False*, *category='General'*, *desc=''*)
Bases: *dace.properties.Property*

Custom Property type for *dace.subsets.Range* members.

dtype

static from_string(*s*)

static to_string(*obj*)

class dace.properties.**ReferenceProperty**(*getter=None*, *setter=None*, *dtype=None*, *default=None*, *from_string=None*, *to_string=None*, *from_json=None*, *to_json=None*, *meta_to_json=None*, *choices=None*, *unmapped=False*, *allow_none=False*, *indirected=False*, *category='General'*, *desc=''*)
Bases: *dace.properties.Property*

Custom Property type that represents a link to another SDFG object. Needs the SDFG to be passed as an argument to *from_string*.

static from_string(*s*, *sdfg=None*)

static to_string(*obj*)

class dace.properties.**SDFGReferenceProperty**(*getter=None*, *setter=None*, *dtype=None*, *default=None*, *from_string=None*, *to_string=None*, *from_json=None*, *to_json=None*, *meta_to_json=None*, *choices=None*, *unmapped=False*, *allow_none=False*, *indirected=False*, *category='General'*, *desc=''*)

Bases: *dace.properties.Property*

from_json(*obj*, *context=None*)

to_json(*obj*)

class dace.properties.**SetProperty**(*element_type*, *getter=None*, *setter=None*, *default=None*, *from_string=None*, *to_string=None*, *from_json=None*, *to_json=None*, *unmapped=False*, *allow_none=False*, *desc=''*, ***kwargs*)

Bases: *dace.properties.Property*

Property for a set of elements of one type, e.g., connectors.

dtype

from_json(*l*, *sdfg=None*)

static from_string(*s*)

to_json(*l*)

static to_string(*l*)

class dace.properties.**ShapeProperty**(*getter=None*, *setter=None*, *dtype=None*, *default=None*, *from_string=None*, *to_string=None*, *from_json=None*, *to_json=None*, *meta_to_json=None*, *choices=None*, *unmapped=False*, *allow_none=False*, *indirected=False*, *category='General'*, *desc=''*)

Bases: *dace.properties.Property*

Custom Property type that defines a shape.

dtype

from_json(*d*, *sdfg=None*)

static from_string(*s*)

to_json(*obj*)

static to_string(*obj*)

class dace.properties.**SubsetProperty**(*getter=None*, *setter=None*, *dtype=None*, *default=None*, *from_string=None*, *to_string=None*, *from_json=None*, *to_json=None*, *meta_to_json=None*, *choices=None*, *unmapped=False*, *allow_none=False*, *indirected=False*, *category='General'*, *desc=''*)

Bases: *dace.properties.Property*

Custom Property type that accepts any form of subset, and enables parsing strings into multiple types of subsets.

allow_none

dtype

from_json(*val*, *sdfg=None*)

static from_string(*s*)

to_json(*val*)

static to_string(*val*)

class dace.properties.**SymbolicProperty**(*getter=None*, *setter=None*, *dtype=None*, *default=None*, *from_string=None*, *to_string=None*, *from_json=None*, *to_json=None*, *meta_to_json=None*, *choices=None*, *unmapped=False*, *allow_none=False*, *indirected=False*, *category='General'*, *desc=''*)

Bases: *dace.properties.Property*

Custom Property type that accepts integers or Sympy expressions.

dtype

static from_string(*s*)

static to_string(*obj*)

class dace.properties.**TransformationHistProperty**(**args*, ***kwargs*)

Bases: *dace.properties.Property*

Property type for transformation histories.

from_json(*data*, *sdfg=None*)

to_json(*hist*)

class dace.properties.**TypeClassProperty**(*getter=None*, *setter=None*, *dtype=None*, *default=None*, *from_string=None*, *to_string=None*, *from_json=None*, *to_json=None*, *meta_to_json=None*, *choices=None*, *unmapped=False*, *allow_none=False*, *indirected=False*, *category='General'*, *desc=''*)

Bases: *dace.properties.Property*

Custom property type for memory as defined in dace.types, e.g. *dace.float32*.

dtype

static from_json(*obj*, *context=None*)

static from_string(*s*)

to_json(*obj*)

static to_string(*obj*)

class dace.properties.**TypeProperty**(*getter=None*, *setter=None*, *dtype=None*, *default=None*, *from_string=None*, *to_string=None*, *from_json=None*, *to_json=None*, *meta_to_json=None*, *choices=None*, *unmapped=False*, *allow_none=False*, *indirected=False*, *category='General'*, *desc=''*)

Bases: *dace.properties.Property*

Custom Property type that finds a type according to the input string.

dtype

static from_json(*obj*, *context=None*)

static from_string(*s*)

dace.properties.**indirect_properties**(*indirect_class*, *indirect_function*, *override=False*)
A decorator for objects that provides indirect properties defined in another class.

dace.properties.**indirect_property**(*cls*, *f*, *prop*, *override*)

dace.properties.**make_properties**(*cls*)
A decorator for objects that adds support and checks for strongly-typed properties (which use the Property class).

dace.properties.**set_property_from_string**(*prop*, *obj*, *string*, *sdfg=None*, *from_json=False*)
Interface function that guarantees that a property will always be correctly set, if possible, by accepting all possible input arguments to from_string.

1.1.9 dace.serialize module

class dace.serialize.**NumpySerializer**
Bases: object

Helper class to load/store numpy arrays from JSON.

static from_json(*json_obj*, *context=None*)

static to_json(*obj*)

class dace.serialize.**SerializableObject**(*json_obj={}*, *typename=None*)
Bases: object

static from_json(*json_obj*, *context=None*, *typename=None*)

json_obj = {}

to_json()

typename = None

dace.serialize.**all_properties_to_json**(*object_with_properties*)

dace.serialize.**dumps**(**args*, ***kwargs*)

dace.serialize.**from_json**(*obj*, *context=None*, *known_type=None*)

dace.serialize.**get_serializer**(*type_name*)

dace.serialize.**loads**(**args*, *context=None*, ***kwargs*)

dace.serialize.**serializable**(*cls*)

dace.serialize.**set_properties_from_json**(*object_with_properties*, *json_obj*, *context=None*, *ignore_properties=None*)

dace.serialize.**to_json**(*obj*)

1.1.10 dace.sdfg module

1.1.11 dace.subsets module

class dace.subsets.**Indices**(*indices*)
Bases: *dace.subsets.Subset*

A subset of one element representing a single index in an N-dimensional data descriptor.

absolute_strides(*global_shape*)

at(*i*, *strides*)
Returns the absolute index (1D memory layout) of this subset at the given index tuple. For example, the range [2:10::2] at index 2 would return 6 (2+2*2). :param i: A tuple of the same dimensionality as subset.dims(). :param strides: The strides of the array we are subsetting. :return: Absolute 1D index at coordinate i.

bounding_box_size()

compose(*other*)

coord_at(*i*)
Returns the offseted coordinates of this subset at the given index tuple. For example, the range [2:10:2] at index 2 would return 6 (2+2*2). :param i: A tuple of the same dimensionality as subset.dims(). :return: Absolute coordinates for index i.

data_dims()

dims()

free_symbols
Returns a set of undefined symbols in this subset.

static from_json(*obj*, *context=None*)

static from_string(*s*)

intersection(*other: dace.subsets.Indices*)

intersects(*other: dace.subsets.Indices*)

max_element()

max_element_approx()

min_element()

min_element_approx()

ndrange()

num_elements()

num_elements_exact()

offset(*other*, *negative*, *indices=None*)

offset_new(*other*, *negative*, *indices=None*)

pop(*dimensions*)

pystr()

reorder(*order*)
Re-orders the dimensions in-place according to a permutation list. :param order: List or tuple of integers from 0 to self.dims() - 1,

indicating the desired order of the dimensions.

replace(*repl_dict*)

size()

size_exact()

squeeze(*ignore_indices=None*)

strides()

to_json()

unsqueeze(*axes: Sequence[int]*) → List[int]
Adds zeroes to the subset, in the indices contained in axes.

The method is mostly used to restore subsets that had their zero-indices removed (i.e., squeezed subsets). Hence, the method is called 'unsqueeze'.

Examples (initial subset, axes -> result subset, output): - [i], [0] -> [0, i], [0] - [i], [0, 1] -> [0, 0, i], [0, 1] - [i], [0, 2] -> [0, i, 0], [0, 2] - [i], [0, 1, 2, 3] -> [0, 0, 0, 0, i], [0, 1, 2, 3] - [i], [0, 2, 3, 4] -> [0, i, 0, 0, 0], [0, 2, 3, 4] - [i], [0, 1, 1] -> [0, 0, 0, i], [0, 1, 2]

Parameters **axes** – The axes where the zero-indices should be added.

Returns A list of the actual axes where the zero-indices were added.

class dace.subsets.**Range**(*ranges*)
Bases: *dace.subsets.Subset*

Subset defined in terms of a fixed range.

absolute_strides(*global_shape*)
Returns a list of strides for advancing one element in each dimension. Size of the list is equal to *data_dims()*, which may be larger than *dims()* depending on tile sizes.

at(*i*, *strides*)
Returns the absolute index (1D memory layout) of this subset at the given index tuple.

For example, the range [2:10:2] at index 2 would return 6 (2+2*2).

Parameters

- **i** – A tuple of the same dimensionality as subset.dims() or subset.data_dims().
- **strides** – The strides of the array we are subsetting.

Returns Absolute 1D index at coordinate i.

bounding_box_size()
Returns the size of a bounding box around this range.

compose(*other*)

coord_at(*i*)
Returns the offseted coordinates of this subset at the given index tuple.

For example, the range [2:10:2] at index 2 would return 6 (2+2*2).

Parameters **i** – A tuple of the same dimensionality as subset.dims() or subset.data_dims().

Returns Absolute coordinates for index i (length equal to *data_dims()*, may be larger than *dims()*).

data_dims()

static dim_to_string(*d*, *t=1*)

dims()

free_symbols
Returns a set of undefined symbols in this subset.

static from_array(*array: dace.data.Data*)
Constructs a range that covers the full array given as input.

static from_indices(*indices: dace.subsets.Indices*)

static from_json(*obj*, *context=None*)

static from_string(*string*)

intersects(*other: dace.subsets.Range*)

max_element()

max_element_approx()

min_element()

min_element_approx()

ndrange()

static ndslice_to_string(*slice*, *tile_sizes=None*)

static ndslice_to_string_list(*slice*, *tile_sizes=None*)

num_elements()

num_elements_exact()

offset(*other*, *negative*, *indices=None*)

offset_new(*other*, *negative*, *indices=None*)

pop(*dimensions*)

pystr()

reorder(*order*)
Re-orders the dimensions in-place according to a permutation list. :param order: List or tuple of integers from 0 to self.dims() - 1,

indicating the desired order of the dimensions.

replace(*repl_dict*)

size(*for_codegen=False*)
Returns the number of elements in each dimension.

size_exact()
Returns the number of elements in each dimension.

squeeze(*ignore_indices=None*, *offset=True*)

strides()

string_list()

to_json()

unsqueeze(*axes: Sequence[int]*) → List[int]
Adds 0:1 ranges to the subset, in the indices contained in axes.

The method is mostly used to restore subsets that had their length-1 ranges removed (i.e., squeezed subsets). Hence, the method is called 'unsqueeze'.

Examples (initial subset, axes -> result subset, output): - [i:i+10], [0] -> [0:1, i], [0] - [i:i+10], [0, 1] -> [0:1, 0:1, i:i+10], [0, 1] - [i:i+10], [0, 2] -> [0:1, i:i+10, 0:1], [0, 2] - [i:i+10], [0, 1, 2, 3] -> [0:1, 0:1, 0:1, 0:1, i:i+10], [0, 1, 2, 3] - [i:i+10], [0, 2, 3, 4] -> [0:1, i:i+10, 0:1, 0:1, 0:1], [0, 2, 3, 4] - [i:i+10], [0, 1, 1] -> [0:1, 0:1, 0:1, i:i+10], [0:1, 1, 2]

Parameters **axes** – The axes where the 0:1 ranges should be added.

Returns A list of the actual axes where the 0:1 ranges were added.

class dace.subsets.**Subset**

Bases: object

Defines a subset of a data descriptor.

at(*i*, *strides*)

Returns the absolute index (1D memory layout) of this subset at the given index tuple.

For example, the range [2:10:2] at index 2 would return 6 (2+2*2).

Parameters

- **i** – A tuple of the same dimensionality as subset.dims() or subset.data_dims().
- **strides** – The strides of the array we are subsetting.

Returns Absolute 1D index at coordinate i.

coord_at(*i*)

Returns the offseted coordinates of this subset at the given index tuple.

For example, the range [2:10:2] at index 2 would return 6 (2+2*2).

Parameters **i** – A tuple of the same dimensionality as subset.dims() or subset.data_dims().

Returns Absolute coordinates for index i (length equal to *data_dims()*, may be larger than *dims()*).

covers(*other*)

Returns True if this subset covers (using a bounding box) another subset.

free_symbols

Returns a set of undefined symbols in this subset.

offset(*other*, *negative*, *indices=None*)

offset_new(*other*, *negative*, *indices=None*)

dace.subsets.**bounding_box_union**(*subset_a: dace.subsets.Subset*, *subset_b: dace.subsets.Subset*) → dace.subsets.Range

Perform union by creating a bounding-box of two subsets.

dace.subsets.**intersects**(*subset_a: dace.subsets.Subset*, *subset_b: dace.subsets.Subset*) → Optional[bool]

Returns True if two subsets intersect, False if they do not, or None if the answer cannot be determined.

Parameters

- **subset_a** – The first subset.
- **subset_b** – The second subset.

Returns True if subsets intersect, False if not, None if indeterminate.

dace.subsets.**union**(*subset_a: dace.subsets.Subset*, *subset_b: dace.subsets.Subset*) → dace.subsets.Subset

Compute the union of two Subset objects. If the subsets are not of the same type, degenerates to bounding-box union. :param subset_a: The first subset. :param subset_b: The second subset. :return: A Subset object whose size is at least the union of the two

inputs. If union failed, returns None.

1.1.12 dace.symbolic module

class dace.symbolic.**DaceSympyPrinter**(*arrays*, **args*, ***kwargs*)
Bases: sympy.printing.str.StrPrinter

Several notational corrections for integer math and C++ translation that sympy.printing.cxxcode does not provide.

class dace.symbolic.**SymExpr**(*main_expr: Union[str, SymExpr]*, *approx_expr: Union[str, SymExpr, None] = None*)
Bases: object

Symbolic expressions with support for an overapproximation expression.

approx

expr

match(**args*, ***kwargs*)

subs(*repldict*)

class dace.symbolic.**SympyAwarePickler**
Bases: _pickle.Pickler

Custom Pickler class that safely saves SymPy expressions with function definitions in expressions (e.g., int_ceil).

persistent_id(*obj*)

class dace.symbolic.**SympyAwareUnpickler**
Bases: _pickle.Unpickler

Custom Unpickler class that safely restores SymPy expressions with function definitions in expressions (e.g., int_ceil).

persistent_load(*pid*)

class dace.symbolic.**SympyBooleanConverter**
Bases: ast.NodeTransformer

Replaces boolean operations with the appropriate SymPy functions to avoid non-symbolic evaluation.

visit_BoolOp(*node*)

visit_Compare(*node: _ast.Compare*)

visit_Constant(*node*)

visit_NameConstant(*node*)

visit_UnaryOp(*node*)

dace.symbolic.**contains_sympy_functions**(*expr*)
Returns True if expression contains Sympy functions.

dace.symbolic.**equalize_symbol**(*sym: sympy.core.expr.Expr*) → sympy.core.expr.Expr
If a symbol or symbolic expressions has multiple symbols with the same name, it substitutes them with the last symbol (as they appear in s.free_symbols).

dace.symbolic.**equalize_symbols**(*a: sympy.core.expr.Expr*, *b: sympy.core.expr.Expr*) → Tuple[sympy.core.expr.Expr, sympy.core.expr.Expr]
If the 2 input expressions use different symbols but with the same name, it substitutes the symbols of the second expressions with those of the first expression.

dace.symbolic.**evaluate**(*expr: Union[sympy.core.basic.Basic, int, float], symbols: Dict[Union[dace.symbolic.symbol, str], Union[int, float]]*) → Union[int, float, numpy.number]

Evaluates an expression to a constant based on a mapping from symbols to values. :param expr: The expression to evaluate. :param symbols: A mapping of symbols to their values. :return: A constant value based on `expr` and `symbols`.

dace.symbolic.**free_symbols_and_functions**(*expr: Union[sympy.core.basic.Basic, dace.symbolic.SymExpr, str]*) → Set[str]

dace.symbolic.**inequal_symbols**(*a: Union[sympy.core.expr.Expr, Any], b: Union[sympy.core.expr.Expr, Any]*) → bool

Compares 2 symbolic expressions and returns True if they are not equal.

dace.symbolic.**is_sympy_userfunction**(*expr*)

Returns True if the expression is a SymPy function.

dace.symbolic.**issymbolic**(*value*, *constants=None*)

Returns True if an expression is symbolic with respect to its contents and a given dictionary of constant values.

dace.symbolic.**overapproximate**(*expr*)

Takes a sympy expression and returns its maximal possible value in specific cases.

dace.symbolic.**pystr_to_symbolic**

Takes a Python string and converts it into a symbolic expression.

dace.symbolic.**resolve_symbol_to_constant**(*symb*, *start_sdfg*)

Tries to resolve a symbol to constant, by looking up into SDFG's constants, following nested SDFGs hierarchy if necessary. :param symb: symbol to resolve to constant :param start_sdfg: starting SDFG :return: the constant value if the symbol is resolved, None otherwise

dace.symbolic.**safe_replace**(*mapping: Dict[Union[sympy.core.basic.Basic, dace.symbolic.SymExpr, str], Union[sympy.core.basic.Basic, dace.symbolic.SymExpr, str]], replace_callback: Callable[[Dict[str, str]], None], value_as_string: bool = False*) → None

Safely replaces symbolic expressions that may clash with each other via a two-step replacement. For example, the mapping `{M: N, N: M}` would be translated to replacing `{N, M} -> __dacesym_{N, M}` followed by `__dacesym{N, M} -> {M, N}`. :param mapping: The replacement dictionary. :param replace_callback: A callable function that receives a replacement

dictionary and performs the replacement (can be unsafe).

Parameters **value_as_string** – Replacement values are replaced as strings rather than symbols.

dace.symbolic.**simplify**

dace.symbolic.**simplify_ext**(*expr*)

An extended version of simplification with expression fixes for sympy. :param expr: A sympy expression. :return: Simplified version of the expression.

dace.symbolic.**swalk**(*expr*, *enter_functions=False*)

Walk over a symbolic expression tree (similar to *ast.walk*). Returns an iterator that yields the values and recurses into functions, if specified.

class dace.symbolic.**symbol**

Bases: `sympy.core.symbol.Symbol`

Defines a symbolic expression. Extends SymPy symbols with DaCe-related information.

add_constraints(*constraint_list*)

check_constraints(*value*)

constraints

default_assumptions = {}

get()

get_or_return(*uninitialized_ret*)

is_initialized()

s_currentsymbol = 0

set(*value*)

set_constraints(*constraint_list*)

dace.symbolic.**symbol_name_or_value**(*val*)
Returns the symbol name if symbol, otherwise the value as a string.

dace.symbolic.**symbols_in_ast**(*tree*)
Walks an AST and finds all names, excluding function names.

dace.symbolic.**symlist**(*values*)
Finds symbol dependencies of expressions.

dace.symbolic.**sympy_divide_fix**(*expr*)
Fix SymPy printouts where integer division such as "tid/2" turns into ".5*tid".

dace.symbolic.**sympy_intdiv_fix**(*expr*)
Fix for SymPy printing out reciprocal values when they should be integral in "ceiling/floor" sympy functions.

dace.symbolic.**sympy_numeric_fix**(*expr*)
Fix for printing out integers as floats with ".00000000". Converts the float constants in a given expression to integers.

dace.symbolic.**sympy_to_dace**(*exprs*, *symbol_map=None*)
Convert all *sympy.Symbol's to DaCe symbols, according to 'symbol_map*.

dace.symbolic.**symstr**(*sym*, *arrayexprs: Optional[Set[str]] = None*) → str
Convert a symbolic expression to a C++ compilable expression. :param sym: Symbolic expression to convert. :param arrayexprs: Set of names of arrays, used to convert SymPy

user-functions back to array expressions.

Returns C++-compilable expression.

dace.symbolic.**symtype**(*expr*)
Returns the inferred symbol type from a symbolic expression.

dace.symbolic.**symvalue**(*val*)
Returns the symbol value if it is a symbol.

1.1.13 Module contents

class dace.**DaceModule**
Bases: module

1.2 diode package

1.2.1 Subpackages

diode.db_scripts package

Submodules

diode.db_scripts.db_setup module

diode.db_scripts.sql_to_json module

diode.db_scripts.sql_to_json_test module

Module contents

1.2.2 Submodules

1.2.3 diode.DaceState module

State of DaCe program in DIODE.

class diode.DaceState.**DaceState**(*dace_code*, *fake_fname*, *source_code=None*, *sdfg=None*, *remote=False*)

Bases: object

This class abstracts the DaCe implementation from the GUI. It accepts a string of DaCe code and compiles it, giving access to the SDFG and the generated code, as well as the matching transformations.

DaCe requires the code to be in a file (for code inspection), but while the user types in the GUI we do not have the data available in a file. Thus we create a temporary directory and save it there. However, the user might check for the filename in the code, thus we provide the original file name in argv[0].

compile()

get_arg_initializers()

get_call_args()

get_dace_code()

get_dace_fake_fname()
Returns the original filename of the DaCe program, i.e., the name of the file he stored to, before performing modifications in the editor

get_dace_generated_files()
Writes the generated code to a temporary file and returns the file name. Compiles the code if not already compiled.

get_dace_tmpfile()
Returns the current temporary path to the generated code files.

get_generated_code()

get_sdfg()

get_sdfgs()
Returns the current set of SDFGs in the workspace. @rtype: Tuples of (name, SDFG).

set_is_compiled(*state*)

set_sdfg(*sdfg*, *name='Main SDFG'*)

1.2.4 diode.abstract_sdfg module

1.2.5 diode.adjust_settings module

1.2.6 diode.config_ui module

1.2.7 diode.diode_client module

DIODE client using a command line interface.

1.2.8 diode.diode_server module

class diode.diode_server.**ConfigCopy**(*config_values*)
Bases: object

Copied Config for passing by-value

get(**key_hierarchy*)

get_bool(**key_hierarchy*)

save(*path=None*)
Nonstatic version of Config::save()

set(**key_hierarchy*, *value=None*, *autosave=False*)

class diode.diode_server.**ExecutorServer**
Bases: object

Implements a server scheduling execution of dace programs

addCommand(*cmd*)

addRun(*client_id*, *compilation_output_tuple*, *more_options*)

consume()

consume_programs()

executorLoop()

getExecutionOutput(*client_id*)

static getPerfdataDir(*client_id*)

lock()

loop()

run(*cot*, *options*)

stop()

unlock()

waitForCommand(*ticket*)

diode.diode_server.**applyOptPath**(*sdfg*, *optpath*, *useGlobalSuffix=True*, *sdfg_props=None*)

diode.diode_server.**applySDFGProperties**(*sdfg*, *properties*, *step=None*)

diode.diode_server.**applySDFGProperty**(*sdfg*, *property_element*, *step=None*)

diode.diode_server.**collect_all_SDFG_nodes**(*sdfg*)

diode.diode_server.**compile**(*language*)

POST-Parameters:

sdfg: ser. sdfg: Contains the root SDFG, serialized in JSON-string. If set, options *code* and *sdfg_props* are taken fron
Can be a list of SDFGs. NOTE: If specified, *code*, *sdfg_prop*, and *language* (in URL) are ignored.

code: string/list. Contains all necessary input code files [opt] optpath: list of dicts, as { <sdfg_name/str>: { name: <str>, params: <dict> }}. Contains the current optimization path/tree.

This optpath is applied to the provided code before compilation

[opt] sdfg_props: list of dicts, as { <sdfg_name/str>: { state_id: <str>, node_id: <str>, params: <dict>, step: <opt int
The step element of the dicts is optional. If it is provided, it specifies the number of optpath elements that preceed it. E.g. a step value of 0 means that the property is applied before the first optimization. If it is omitted, the property is applied after all optimization steps, i.e. to the resulting SDFG

[opt] perf_mode: string. Providing "null" has the same effect as omission. If specified, enables performance instrume
provided in the DaCe settings. If null (or omitted), no instrumentation is enabled.

client_id: <string>: For later identification. May be unique across all runs, must be unique across clients

Returns: sdfg: object. Contains a serialization of the resulting SDFGs. generated_code: string. Contains the output code sdfg_props: object. Contains a dict of all properties for

every existing node of the sdfgs returned in the sdfg field

diode.diode_server.**compileProgram**(*request*, *language*, *perfopts=None*)

diode.diode_server.**create_DaceState**(*code*, *sdfg_dict*, *errors*)

diode.diode_server.**diode_settings**(*operation*)

diode.diode_server.**execution_queue_query**(*op*)

diode.diode_server.**expand_node_or_sdfg**()
Performs expansion of a single library node or an entire SDFG. Fields: sdfg (required): SDFG as JSON nodeid (not required): A list of: [SDFG ID, state ID, node ID]

diode.diode_server.**getEnum**(*name*)
Helper function to enumerate available values for *ScheduleType*.

Returns: enum: List of string-representations of the values in the enum

diode.diode_server.**getPubSSH**()

diode.diode_server.**get_available_ace_editor_themes**()

diode.diode_server.**get_library_implementations**(*name*)
Helper function to enumerate available implementations for a given library node.

Returns: enum: List of string-representations of implementations

diode.diode_server.**get_run_status**()

diode.diode_server.**get_settings**(*client_id*, *name=''*, *cv=None*, *config_path=''*)

diode.diode_server.**get_transformations**(*sdfgs*)

diode.diode_server.**index**(*path*)
This is an http server (on the same port as the REST API). It serves the files from the 'webclient'-directory to user agents. Note: This is NOT intended for production environments and security is disregarded!

diode.diode_server.**main**()

diode.diode_server.**optimize**()
Returns a list of possible optimizations (transformations) and their properties.

POST-Parameters: input_code: list. Contains all necessary input code files optpath: list of dicts, as { name: <str>, params: <dict> }. Contains the current optimization path/tree.

This optpath is applied to the provided code before evaluating possible pattern matches.

client_id: For identification. May be unique across all runs, must be unique across clients

Returns matching_opts: list of dicts, as { opt_name: <str>, opt_params: <dict>, affects: <list>, children: <recurse> }. Contains the matching transformations. *affects* is a list of affected node ids, which must be unique in the current program.

diode.diode_server.**properties_to_json_list**(*props*)

diode.diode_server.**redirect_base**()

diode.diode_server.**run**()
This function is equivalent to the old DIODE "Run"-Button.

POST-Parameters: (Same as for compile(), language defaults to 'dace') perfmodes: list including every queried mode corecounts: list of core counts (one run for every number of cores)

diode.diode_server.**set_settings**(*settings_array*, *client_id*)

diode.diode_server.**split_nodeid_in_state_and_nodeid**(*nodeid*)

diode.diode_server.**status**()

1.2.9 diode.remote_execution module

class diode.remote_execution.**AsyncExecutor**(*remote*)
Bases: object

Asynchronous remote execution.

add_async_task(*task*)

append_run_async(*dace_state*, *fail_on_nonzero=False*)

callMethod(*obj*, *name*, **args*)

execute_task(*task*)

join(*timeout=None*)

run()

run_async(*dace_state*, *fail_on_nonzero=False*)

run_sync(*func*)

class diode.remote_execution.**Executor**(*remote*, *async_host=None*)
Bases: object

DaCe program execution management class for DIODE.

config_get(**key_hierarchy*)

copy_file_from_remote(*src*, *dst*)

copy_file_to_remote(*src*, *dst*)

copy_folder_from_remote(*src: str*, *dst: str*)

copy_folder_to_remote(*src*, *dst*)

create_remote_directory(*path*)
Creates a path on a remote node.

@note: We use *mkdir -p* for now, which is not portable.

delete_local_folder(*path*)

exec_cmd_and_show_output(*cmd*, *fail_on_nonzero=True*)

remote_compile(*rem_path*, *dace_progname*)

remote_delete_dir(*deldir*)

remote_delete_file(*delfile*)

remote_exec_dace(*remote_workdir*, *dace_file*, *use_mpi=True*, *fail_on_nonzero=False*, *omp_num_threads=None*, *additional_options_dict=None*, *repetitions=None*)

run(*dace_state*, *fail_on_nonzero=False*)

run_local(*sdfg: dace.sdfg.sdfg.SDFG*, *driver_file: str*)

run_remote(*sdfg: dace.sdfg.sdfg.SDFG*, *dace_state*, *fail_on_nonzero: bool*)

set_config(*config*)

set_exit_on_error(*do_exit*)

show_output(*outstr*)
Displays output of any ongoing compilation or computation.

class diode.remote_execution.**FunctionStreamWrapper**(**funcs*)
Bases: object

Class that wraps around a function with a stream-like API (write).

flush()

write(**args*, ***kwargs*)

1.2.10 diode.sdfv module

1.2.11 Module contents

CHAPTER 2

Reference

- genindex
- modindex

Python Module Index

d

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X