MapFusion

Content

- The cache
- Bugfixes
- New cases
- Struct data flow

Cache

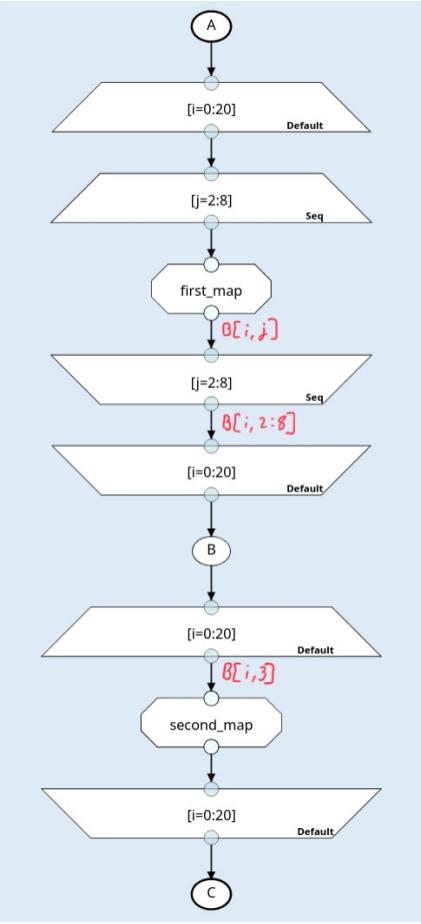
- In order to determine if an intermediate can be removed the whole SDFG needs to be scanned.
- The old transformation scanned the full SDFG for every intermediate on every fusing operation.
- The new one scans the SDFG once, computes all data that could not be deleted and stores it.
- The cache is never invalidated over the lifetime of the fusion object.
- This is safe, as long as no new names are added, which is the case for its use during auto optimize.
- However, it is a little bit different than the old version, so we could:
  - Writing it into the change log and life with it.
  - Making caching an opt-in behaviour (it should be activated inside auto opt).

Bugfixes

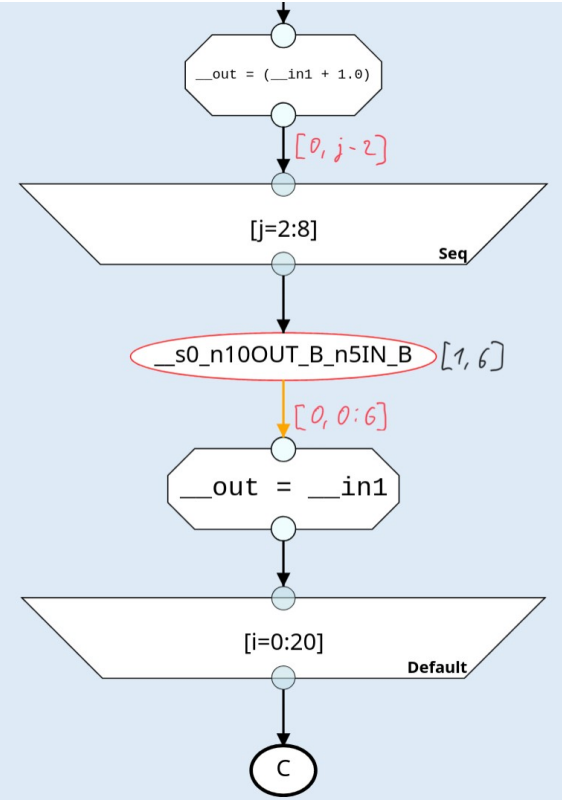
- Now a list of fixes

`mapfusion_test.py::test_offset_correction_scalar_read`

Current version



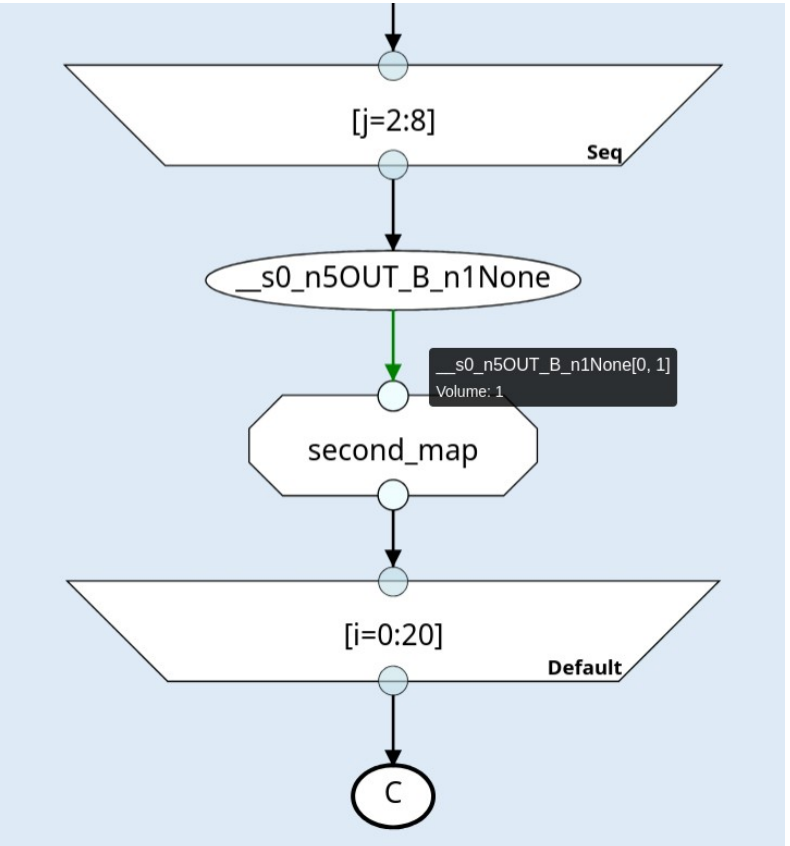
Where it fails



The top Map contains a a nested Map that only partially writes into `B`, i.e. `B[:, 2:8]`. The second Map only accesses `B[i, 3]`. The current implementation applies, passes validation, but fails to compile. For some reason it wants to pass the full array (pointer) into the Tasklet that only accepts a scalar.

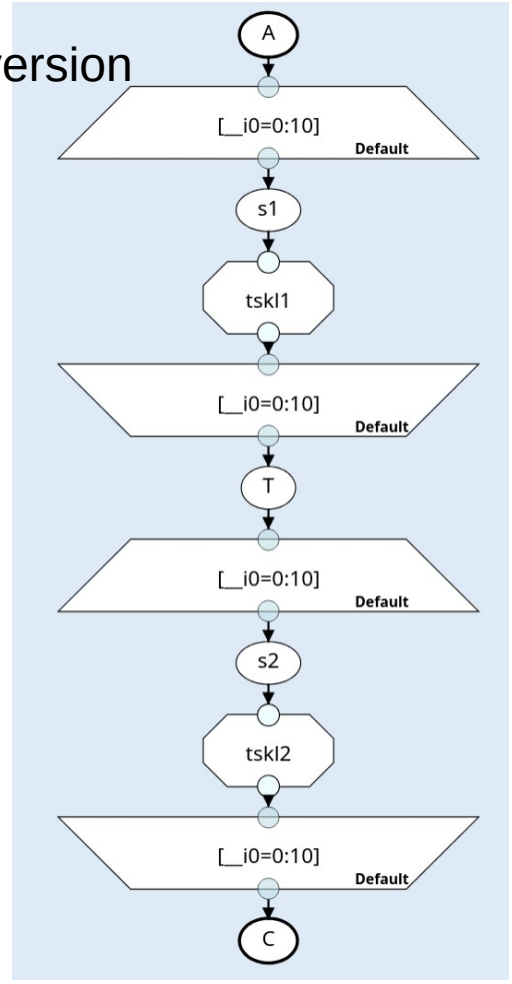
mapfusion_test.py::test_offset_correction_scalar_read

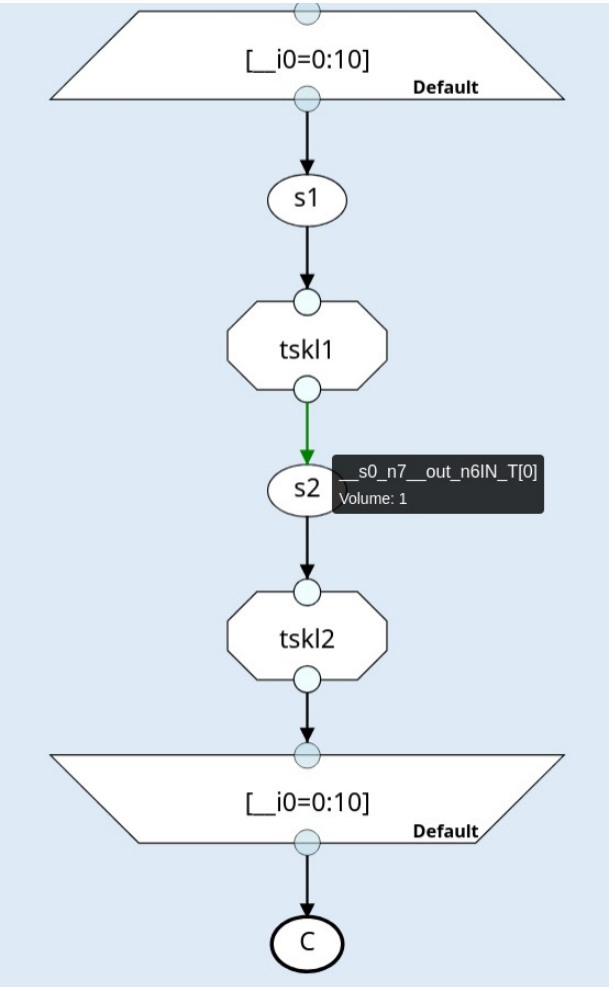
The new version:
As you can see the read to the intermediate is now correct.
Although it still has a size of 6.



mapfusion_test.py::test_inner_map_dependency_resolved

Current version

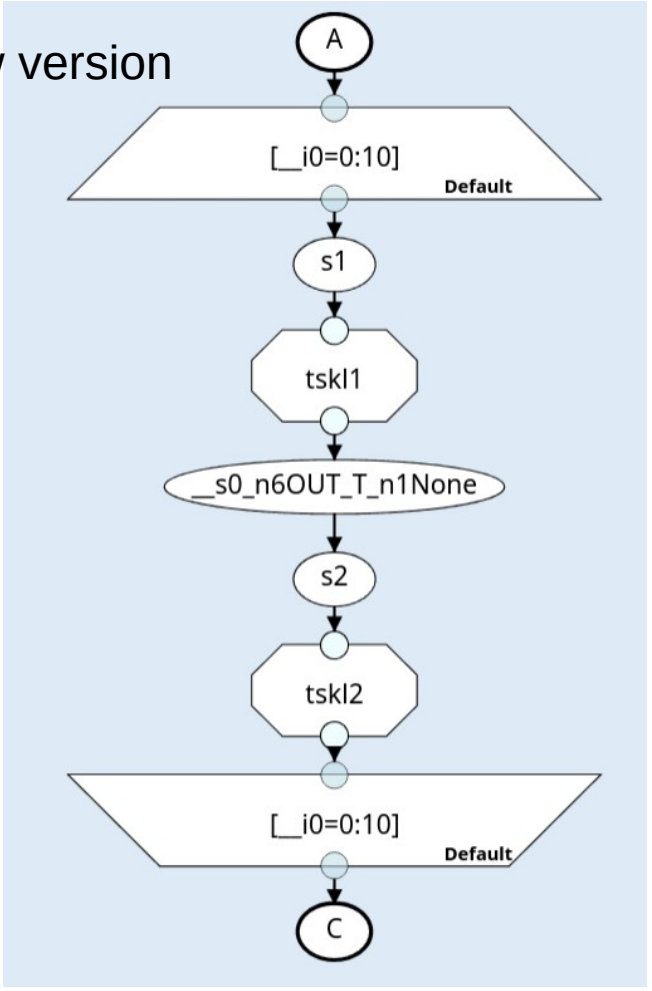




Where it fails:
The transformation applies, it creates an Memlet and associates it with the intermediate, it created, but it does not create a corresponding AccessNode.
Note that here it is safe to fuse, because the two scalars, that are used inside the Map scopes are different. But the transformation would also apply, if both would refer to the same data and produce the same error.

mapfusion_test.py::test_inner_map_dependency_resolved

New version

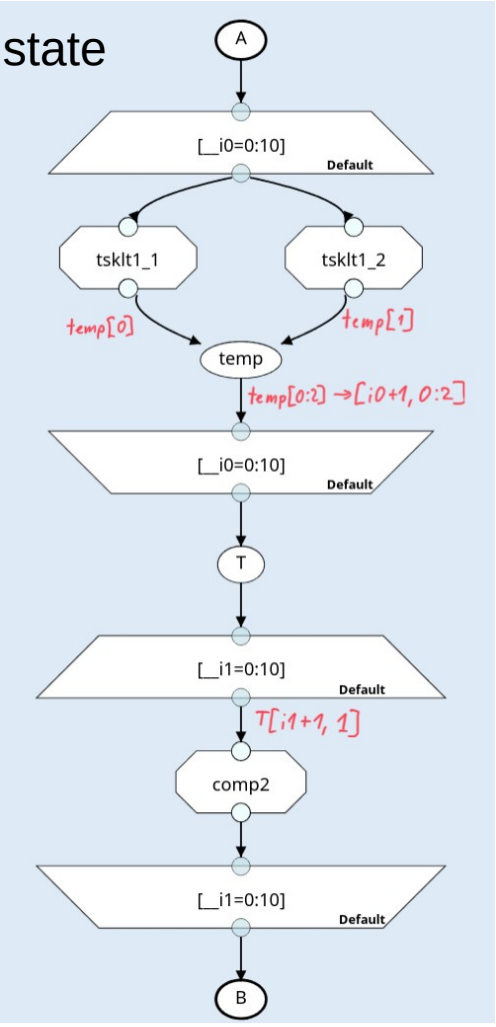


The new transformation applies correctly.

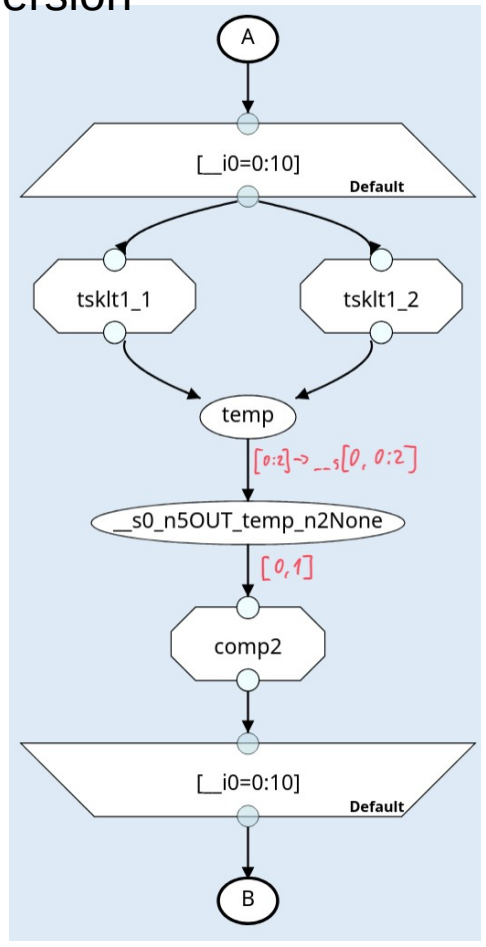
In case both Map scopes would refer to the same data, it would not apply. Note: In this situation fusion would be safe, but it might not be in general, so the transformation does nothing.

mapfusion_test.py::test_fusion_intermediate_different_access

Initial state



New version



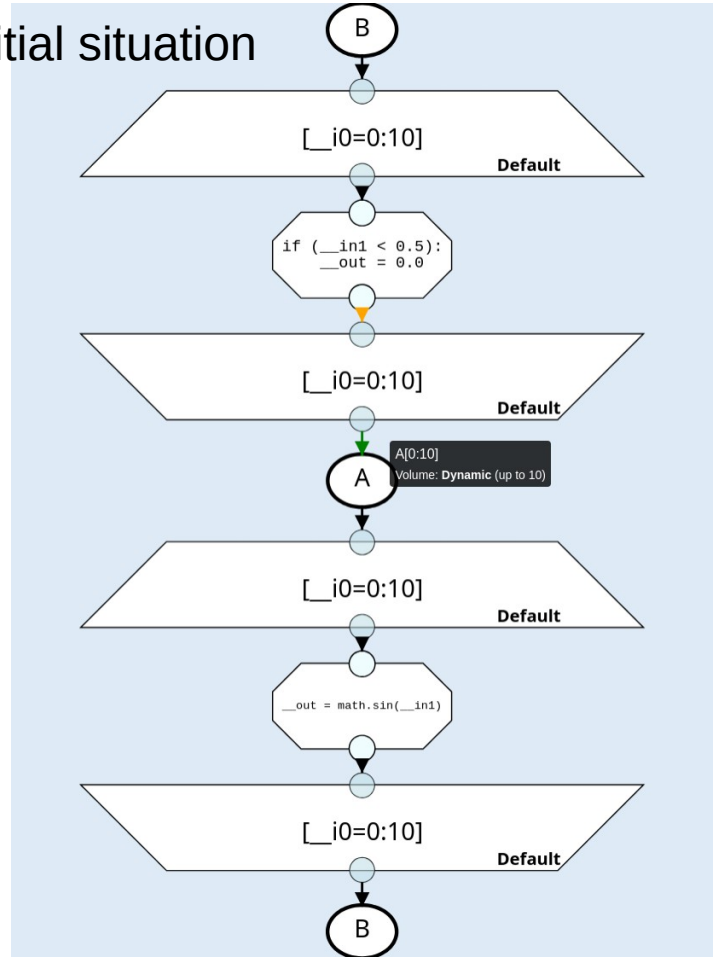
Similar to the previous case but the intermediate is explicitly there. Important: The Memlet that connects `temp` and the first MapExit is associated to `temp`. The current transformation fails to apply. If we look why, we see that it picks up the wrong subset (direction issue) and then the data dependency can not be met. If we "fix" that (i.e. remove a special case) it still fails. Now it complains that `temp` has only one dimension.

If also fix that, i.e. add a dummy dimension to it, then it complains that the second Tasklet does not have a `.data` attribute.

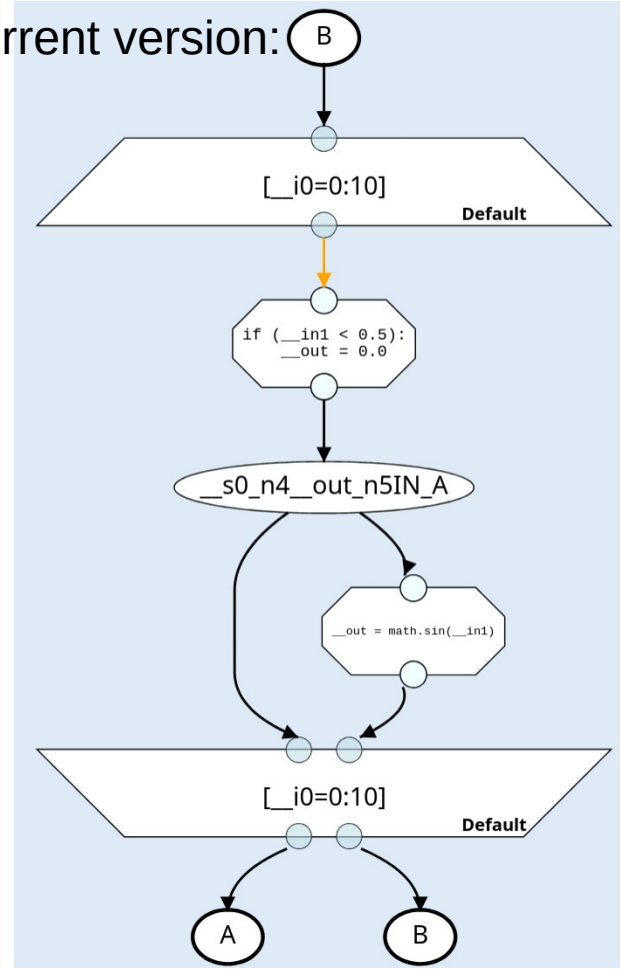
Thus it fails to handle this case.

mapfusion_test.py::test_fusion_dynamic_producer

Initial situation



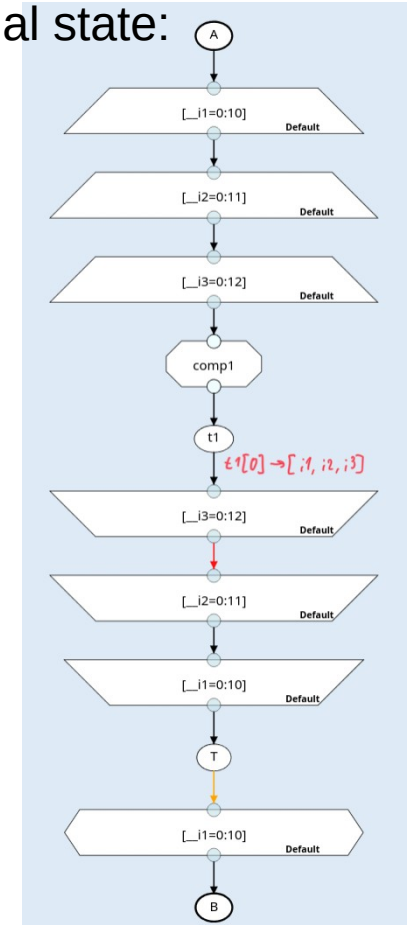
Current version:



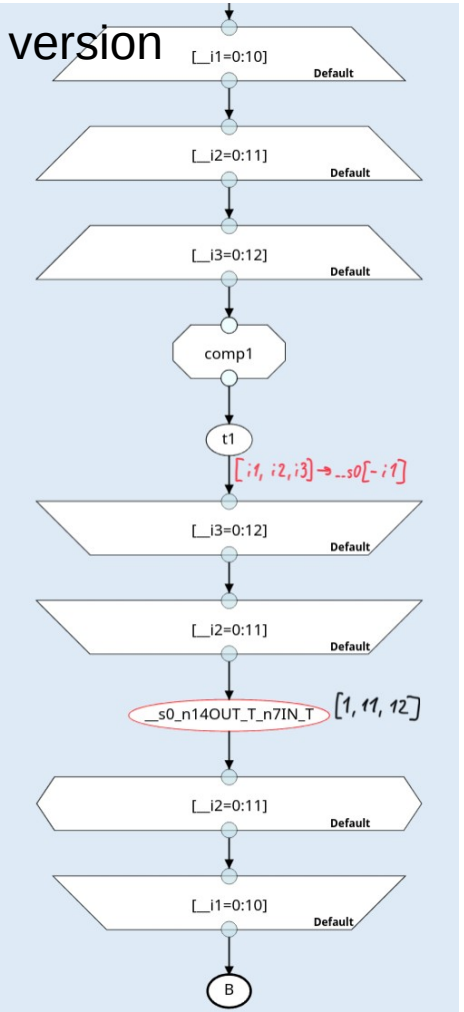
The current transformation does not respect the dynamic Memlets, of producers, it ignores them and simply merges the Maps together. The correct behaviour is to not fuse. In the generated code, the `__out` in the top Tasklet will be replaced with the intermediate (`__s0...`), thus it will only have a value if `__in1 < 0.5` holds otherwise it will have an undefined value.

mapfusion_test.py::test_fusion_intrinsic_memlet_direction

Initial state:



Current version

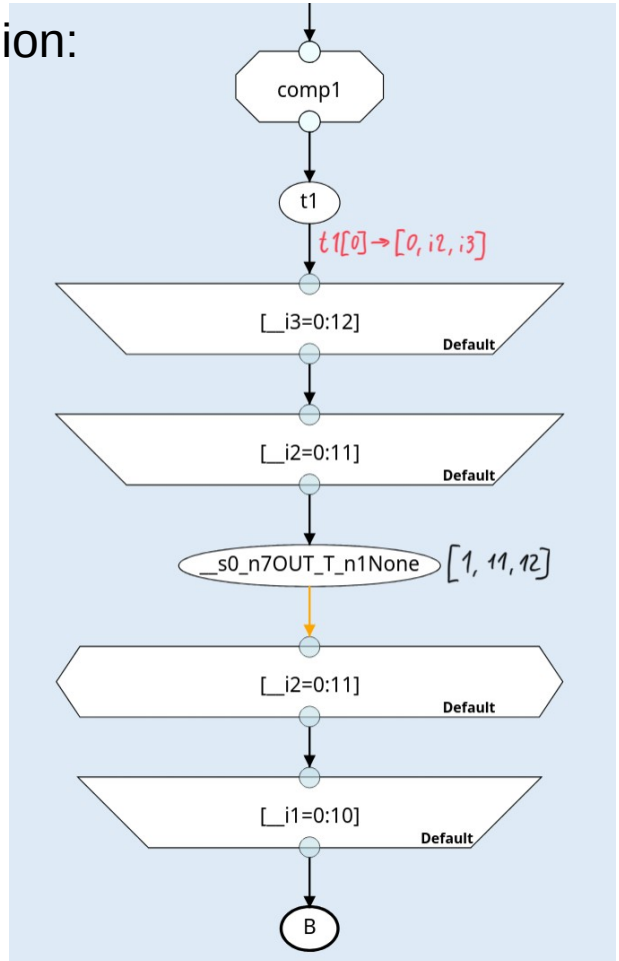


The main difficulty here is that the Memlet, that goes from `t1` into the MapExit of the most inner Map, does not refer to `T`, the intermediate, but to `t1`. The current transformation applies and modifies this inner Memlet such that it refers to the new intermediate data and sets `.subset` to `[-__i1]` and `.other_subset` to `[__i1, __i2, __i3]`. This causes a validation error.

A similar situation is also present the the second Map, which is collapsed for visibility.

mapfusion_test.py::test_fusion_intrinsic_memlet_direction

New version:

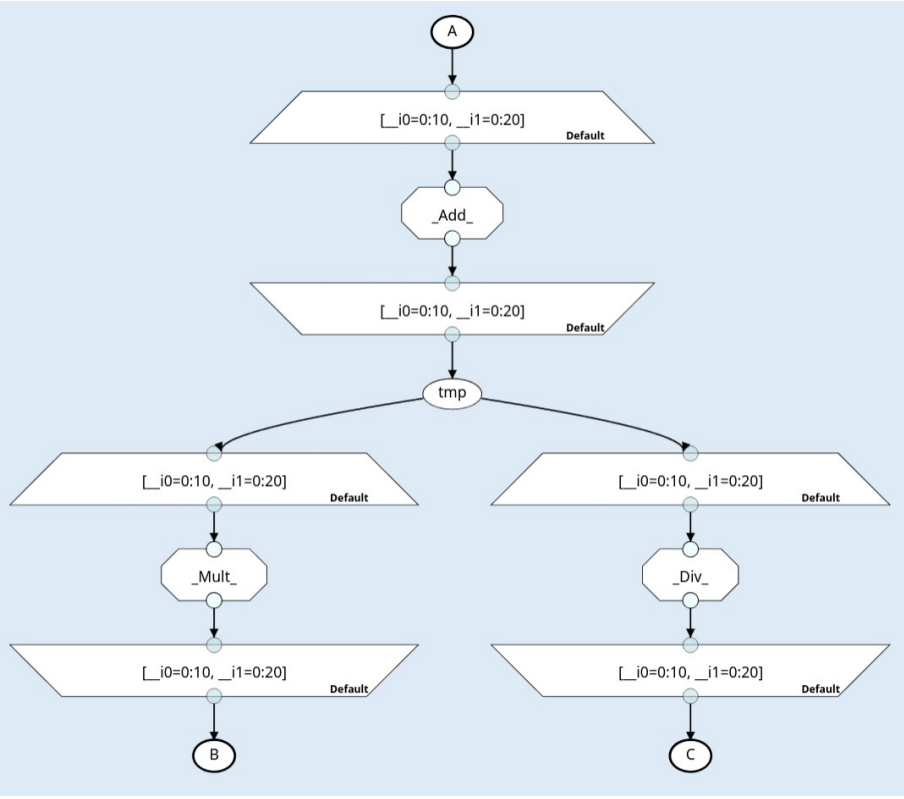


New Cases That Are Handled

- These are cases that were not handled before. Thus the transformation did not apply.
- They are mostly due to a better handling of intermediates that are needed somewhere else, i.e. these can not simply be removed.
- NOTE: The original MapFusion can potentially handle such cases as well.

mapfusion_test.py::test_fusion_shared

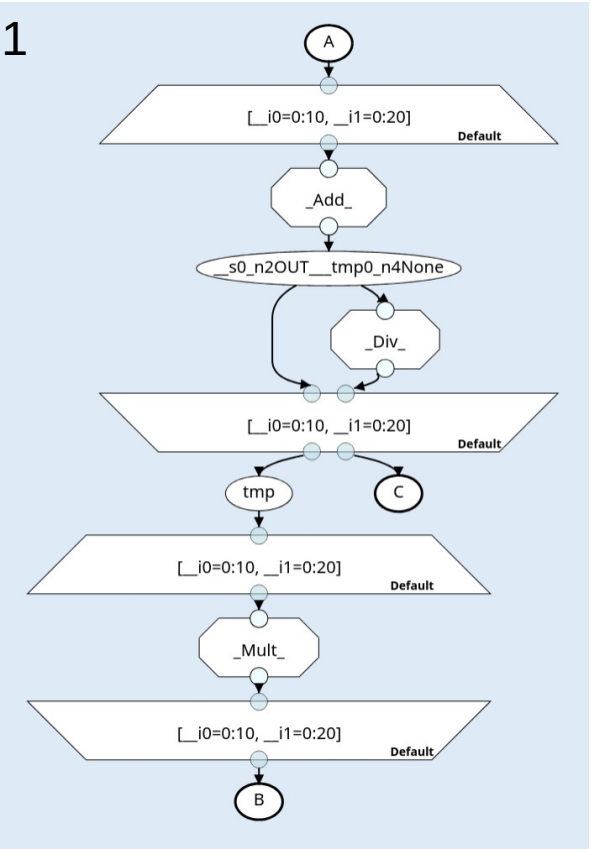
Initial state



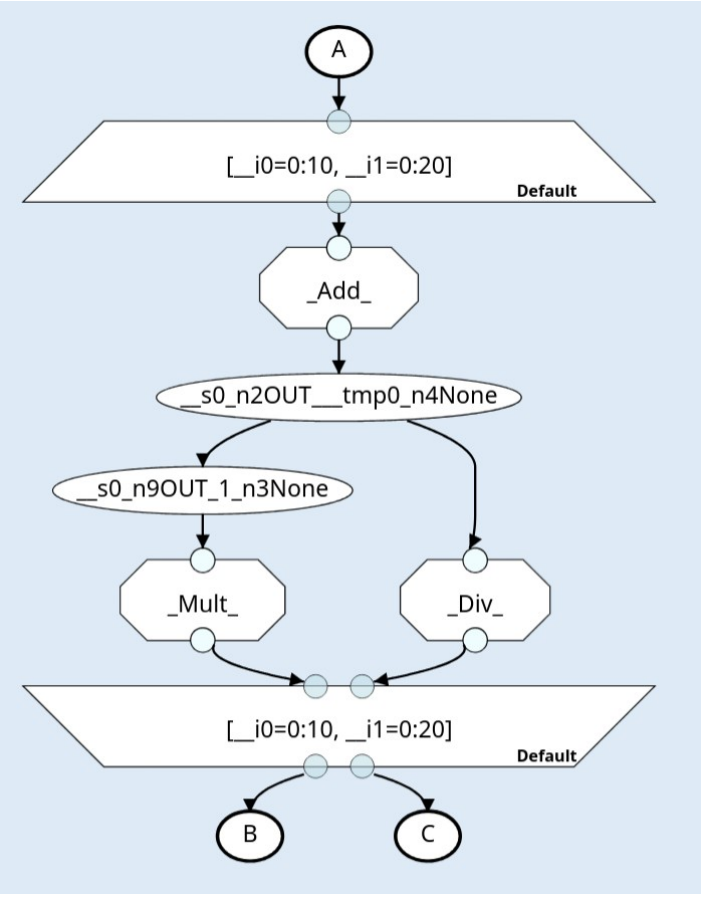
The new transformation applies here, it needs however two steps, see nest slide.

mapfusion_test.py::test_fusion_shared

Step 1

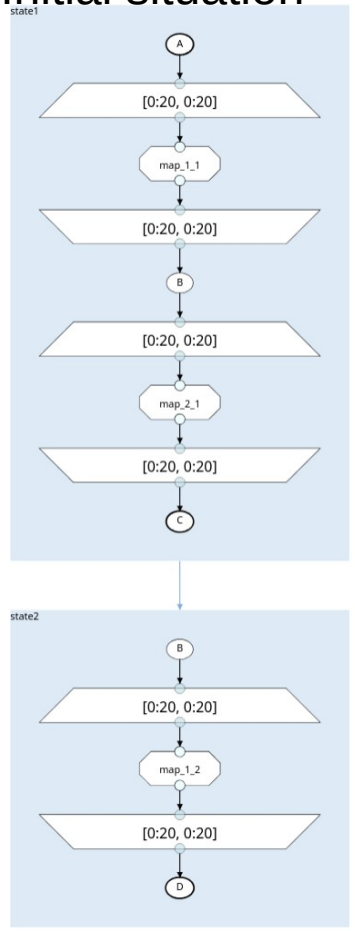


Step 2

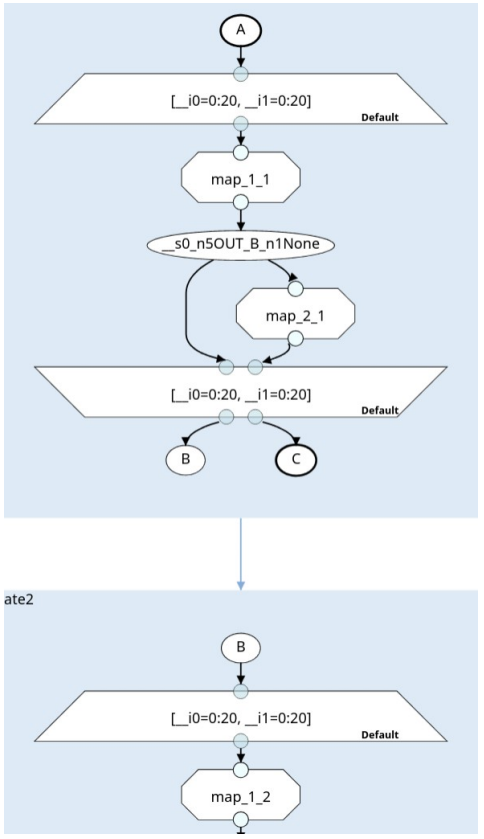


`mapfusion_test.py::test_interstate_fusion`

Initial situation



New version

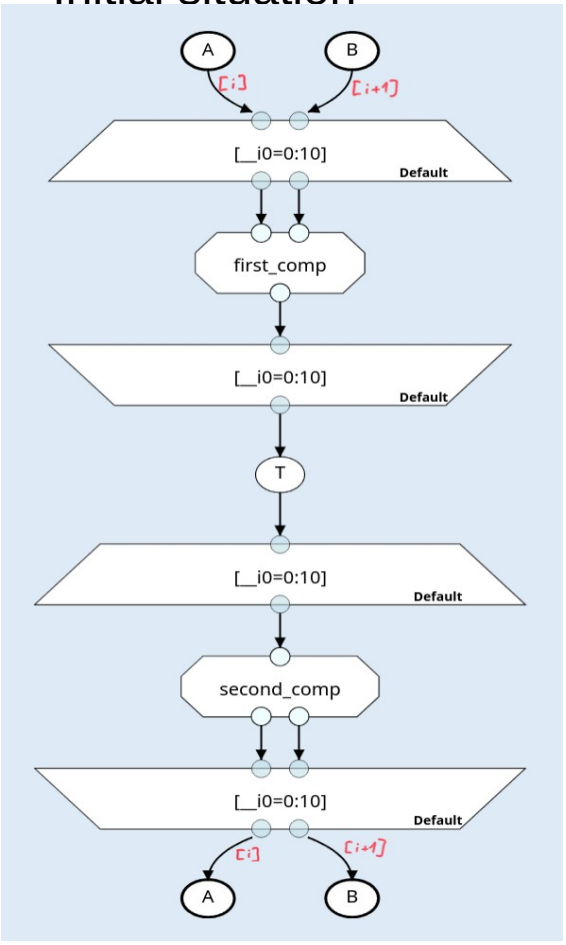


The current transformation did not perform the merge, because it picked up that `B` was used in another state.

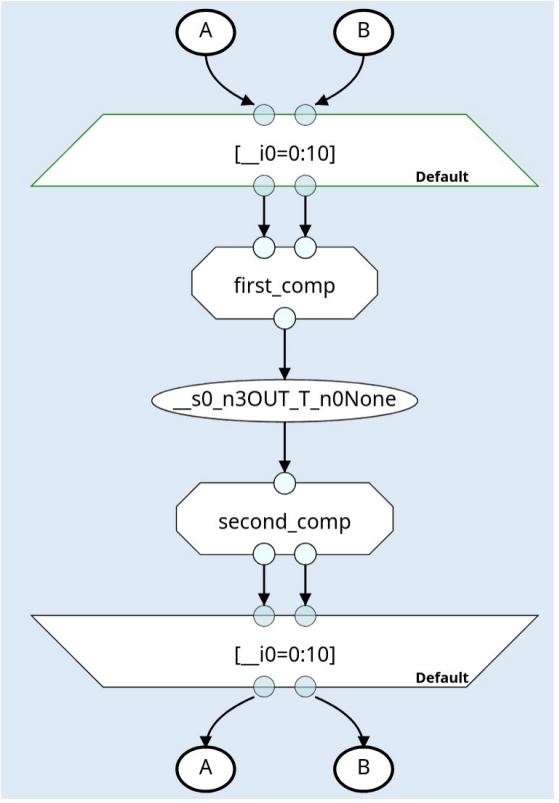
The new transformation will apply, because it recreates `B` as a new output of the fused Map.

mapfusion_test.py::test_fusion_different_global_accesses

Initial situation



New version

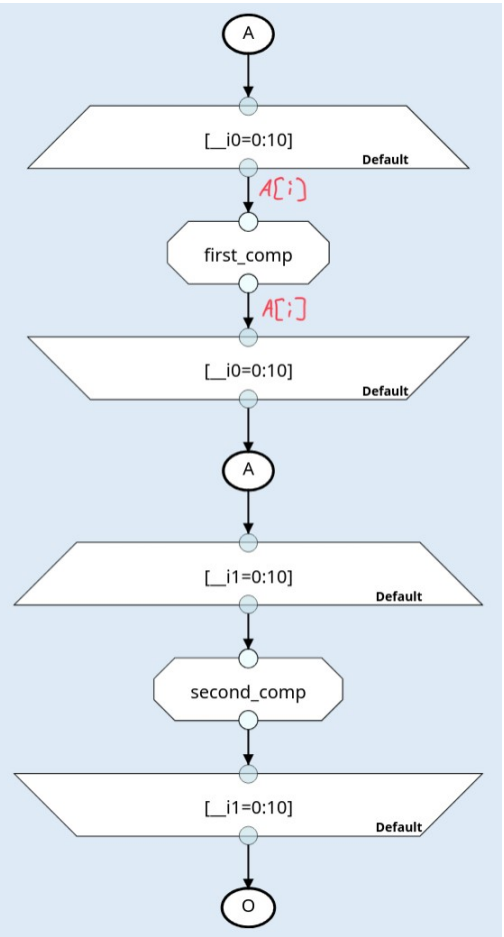


`A` and `B` are both used as input and output. The important thing is that `A` is only accessed as `A[i]` while `B` is accessed as `B[i+1]`.
The current transformation does not apply here, while the new does.

Note: If `B` would be accessed as `B[i]` (or more generally in the same style as `A`) then the current transformation would apply as well.

mapfusion_test.py::test_fusion_dataflow_intermediate_2

Initial situation

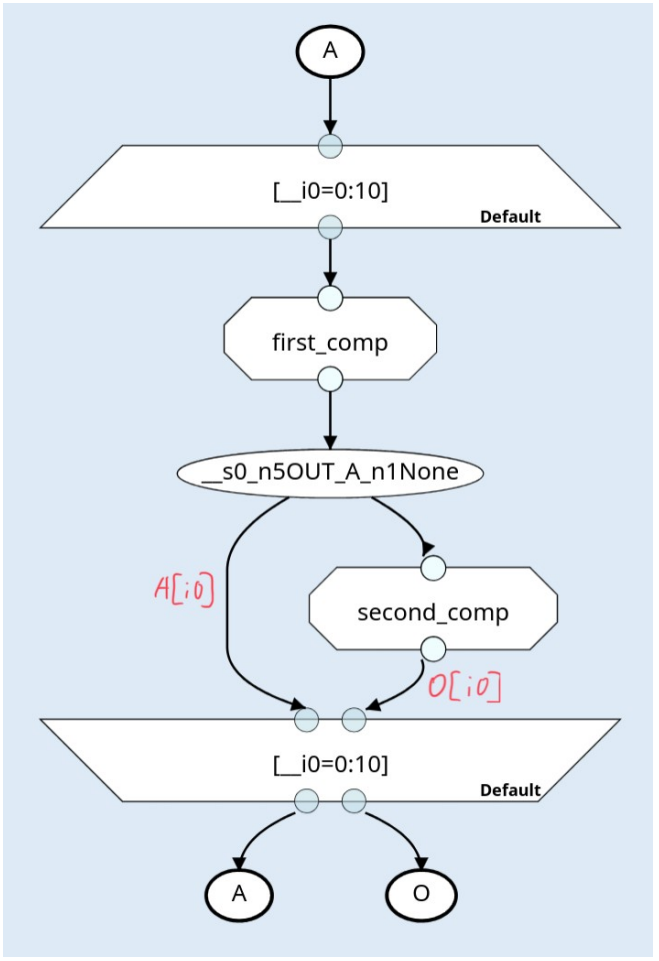


As it can be seen, `A` is both used as input and output to the first Map and serves as intermediate, i.e. is also input to the second Map. The old transformation does not apply, but the new one does.
However, besides the normal requirement of MapFusion, i.e. one iteration of the first Map must produce everything that an iteration of the second Map needs, the following must hold: The access of `A`, in the first Map must be pointwise, i.e. every position that is read must also be written to by an iteration. So if the first Map would read `A[9 - i]` instead, then this is not satisfied anymore and the transformation would not apply. Note that the constraint on the intermediate would still be satisfied, as the first Map writes `A[i]` and the second would still read `A[i]`.

I am not sure, but I am pretty sure that such a Map is invalid and is only valid in particular situations.

mapfusion_test.py::test_fusion_dataflow_intermediate_2

New version:



Strict Dataflow

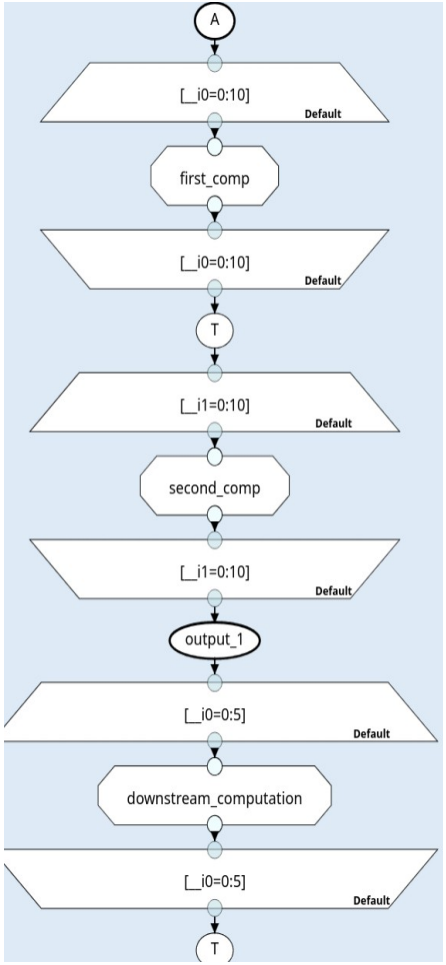
The most controversial change is most likely the introduction of the "strict dataflow" mode, which is on my default.
It started as a compatibility flag to work around some behaviour in DaCe. They are mostly related to "shared intermediate nodes", i.e. intermediates that can not be removed from the SDFG, because they are used somewhere else. See `issue#1642`.

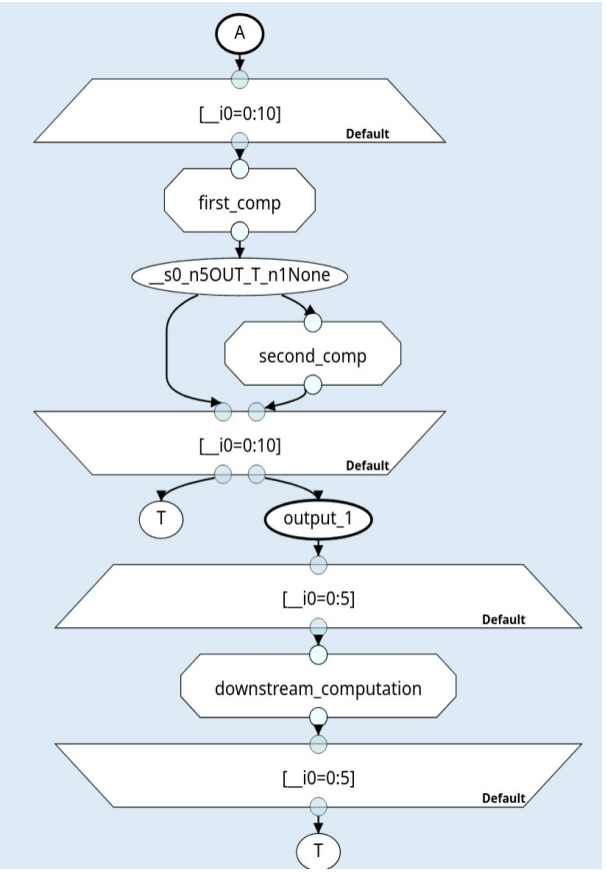
It believe that it is related that some transformations do not carefully enough check some data dependencies, for this let's look at an example (nest slide).

Strict data flow means that if an intermediate, is classified as a shared intermediate and there is another AccessNode in the dataflow graph, that is reachable from the intermediate itself, then the fusion is rejected.
Note, that if the usage is in another state, upstream the data flow or in a concurrent dataflow graph (most likely invalid anyway), then this restriction does not apply.

mapfusion_test.py::test_fusion_dataflow_intermediate_downstream

Initial state





There is another state where `T` is stored into global data.

The first `T` is the intermediate, but there is another `T`, that is reachable from the first one. If strict dataflow is enabled then the transformation does not apply. If it is disabled then it applies and generates the SDFG on the right.

You see that `T` has become a sink node of the first Map. But, it still appears, downstream the dataflow graph as output of the second Map.

This is (at least I think that) the prototypical pattern that causes problems in some transformations. See also `issue#1642`.