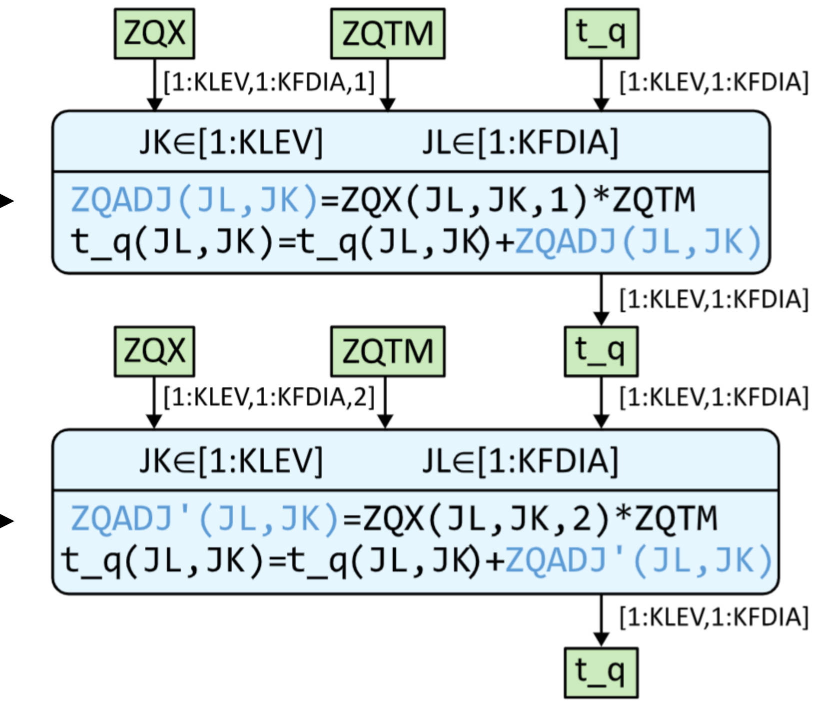
DaFlEx – Workshop Recap

Recap – Scalar fission

```
do JK=1,KLEV
   do JL=1,KFDIA
     ZQADJ=ZQX(JL,JK,1)*ZQTM
     t_q(JL,JK)=t_q(JL,JK)+ZQADJ
   enddo
enddo

do JK=1,KLEV
   do JL=1,KFDIA
     ZQADJ=ZQX(JL,JK,2)*ZQTM
     t_q(JL,JK)=t_q(JL,JK)+ZQADJ
   enddo
enddo
```



Recap – CloudSC to GPU

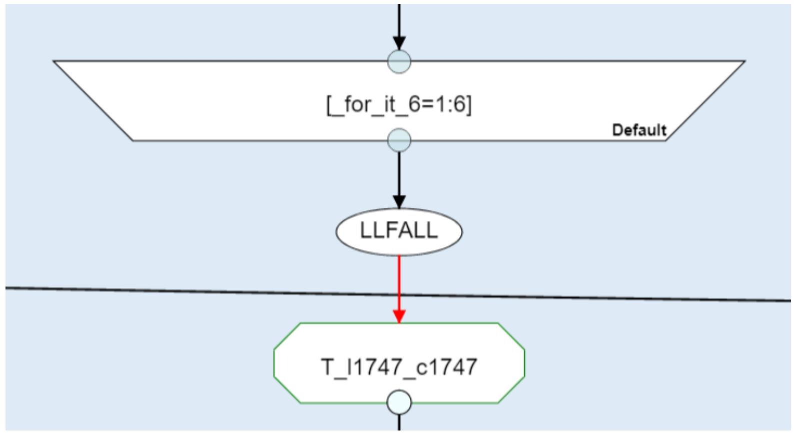
- **Fixed To_GPU transformation**
  - CloudSC becomes a single (global) map on GPU.
  - All sequential code has been hidden by using parallelism across NBLOCKS.
  - Dedicated transformations to simplify inter-state edges depending on constant variables and arrays.
  - Minimizing the size of temporary arrays

- **Unused Cloudsc_to_GPU**
  - Should form the basis for big_application_to_gpu
  - Support transferring data between host and device “on-demand”.
  - Optimize data transfers for CPU loops (hoist out copies).
  - Allow to run sequential loops and tasklets on CPU.

Recap – Dependency edges: Lex

- **Implemented as state fusion variation**

```
LLFALL(:)=.FALSE.
DO JM=1,NCLV
  IF (ZVQX(JM)>0.0) LLFALL(JM)=.TRUE.
ENDDO
LLFALL(NCLDQI)=.FALSE.
```

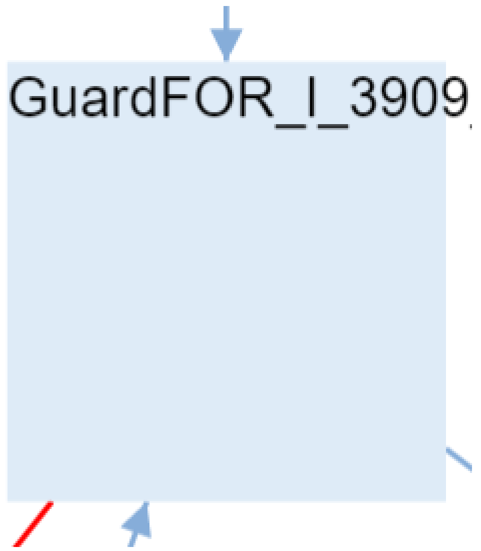


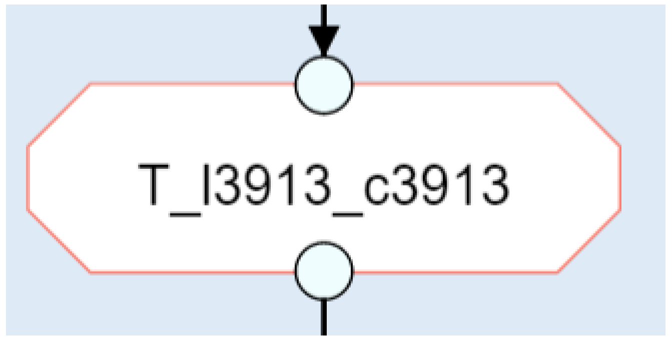
How does it work?

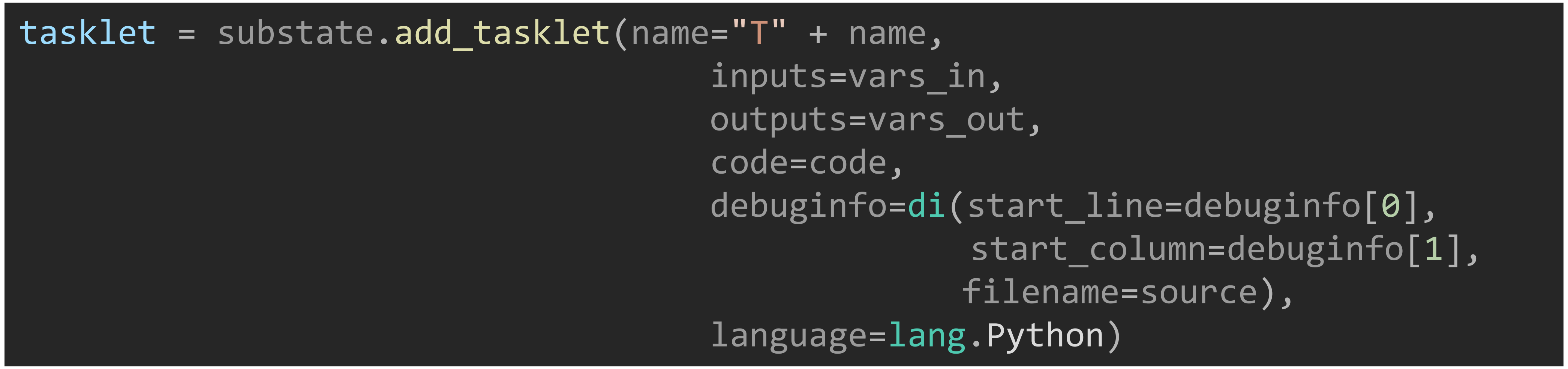
- Checks for all conditions state fusion does, but stores instances of WAW or WAR rather than breaking
- If all other conditions are met, fuses the states, and for each instance of WAW or WAR stored, connects the “bottom” of the subgraph containing the last access in the first state, with the “top” of the subgraph containing the first access in the second state.
- The connection is an empty memlet – so in effect a happens-before dependency edge.

How does it help? It allows more state fusion, so other subgraphs can be better optimized.

Persistent debug information – Marcin

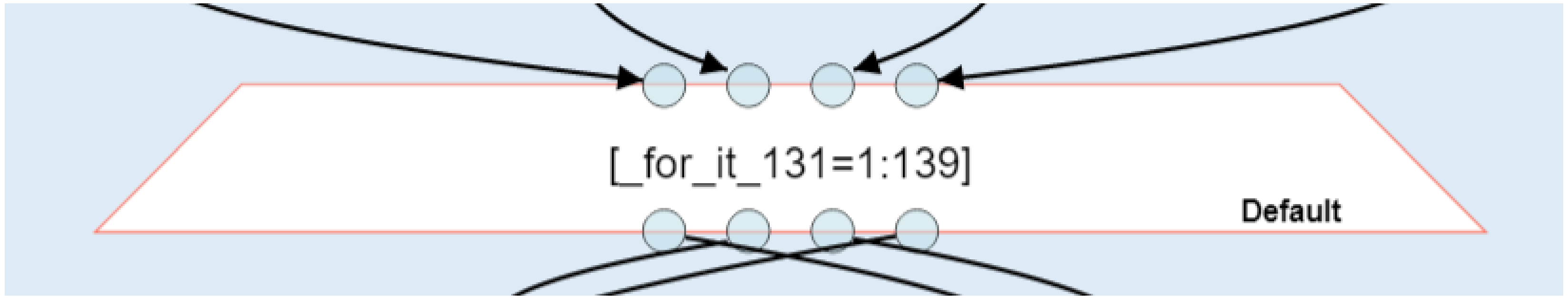






```
tasklet = substate.add_tasklet(name="T" + name,
                               inputs=vars_in,
                               outputs=vars_out,
                               code=code,
                               debuginfo=di(start_line=debuginfo[0],
                                            start_column=debuginfo[1],
                                           filename=source),
                               language=lang.Python)
```

Persistent debug information – Marcin



Some transformations remove debug information!

Proposed rework: search for first instance of debug information available in the transformed subgraph – might not be ideal in all cases, but better than nothing!

Removing offsets from DaCe – Lex

- **Proposal: Have frontends handle offsets if they exist by normalizing every array and access to each array as though they would be 0-based.**
- **Long-term: Remove offset support from DaCe entirely (as we rework transformations and components.)**

- **Why?**
  - Memlets are not linked to a specific array.
  - When doing range/subset operations, the offset is necessary.
  - Many transformations, including subset operations implicitly assume 0-based arrays
  - Having partial support is confusing to developers.

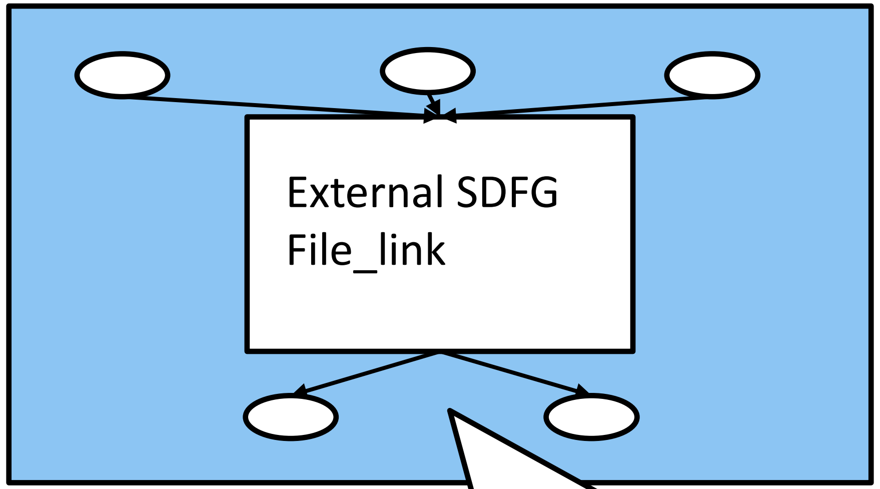
External nested SDFGs – Lex

```
if (direction_switch) then
     !x-direction first
     call semi_discrete_step( state , state     , state_tmp , dt3 , DIR_X , flux , tend ,hy_dens_cell,hy_dens_theta_cell,hy_dens_int,hy_dens_theta_int,hy_pressure_int)
     call semi_discrete_step( state , state_tmp , state_tmp , dt2 , DIR_X , flux , tend,hy_dens_cell,hy_dens_theta_cell ,hy_dens_int,hy_dens_theta_int,hy_pressure_int)
     call semi_discrete_step( state , state_tmp , state     , dt1 , DIR_X , flux , tend ,hy_dens_cell,hy_dens_theta_cell,hy_dens_int,hy_dens_theta_int,hy_pressure_int)
     !z-direction second
     call semi_discrete_step( state , state     , state_tmp , dt3 , DIR_Z , flux , tend,hy_dens_cell,hy_dens_theta_cell,hy_dens_int,hy_dens_theta_int,hy_pressure_int )
     call semi_discrete_step( state , state_tmp , state_tmp , dt2 , DIR_Z , flux , tend,hy_dens_cell,hy_dens_theta_cell,hy_dens_int,hy_dens_theta_int,hy_pressure_int )
     call semi_discrete_step( state , state_tmp , state     , dt1 , DIR_Z , flux , tend ,hy_dens_cell,hy_dens_theta_cell,hy_dens_int,hy_dens_theta_int,hy_pressure_int)
   else
     !z-direction second
     call semi_discrete_step( state , state     , state_tmp , dt3 , DIR_Z , flux , tend,hy_dens_cell,hy_dens_theta_cell,hy_dens_int,hy_dens_theta_int,hy_pressure_int )
     call semi_discrete_step( state , state_tmp , state_tmp , dt2 , DIR_Z , flux , tend,hy_dens_cell,hy_dens_theta_cell ,hy_dens_int,hy_dens_theta_int,hy_pressure_int)
     call semi_discrete_step( state , state_tmp , state     , dt1 , DIR_Z , flux , tend,hy_dens_cell,hy_dens_theta_cell,hy_dens_int,hy_dens_theta_int,hy_pressure_int )
     !x-direction first
     call semi_discrete_step( state , state     , state_tmp , dt3 , DIR_X , flux , tend,hy_dens_cell,hy_dens_theta_cell,hy_dens_int,hy_dens_theta_int,hy_pressure_int )
     call semi_discrete_step( state , state_tmp , state_tmp , dt2 , DIR_X , flux , tend,hy_dens_cell,hy_dens_theta_cell,hy_dens_int,hy_dens_theta_int,hy_pressure_int )
     call semi_discrete_step( state , state_tmp , state     , dt1 , DIR_X , flux , tend,hy_dens_cell,hy_dens_theta_cell,hy_dens_int,hy_dens_theta_int,hy_pressure_int )
   endif
```

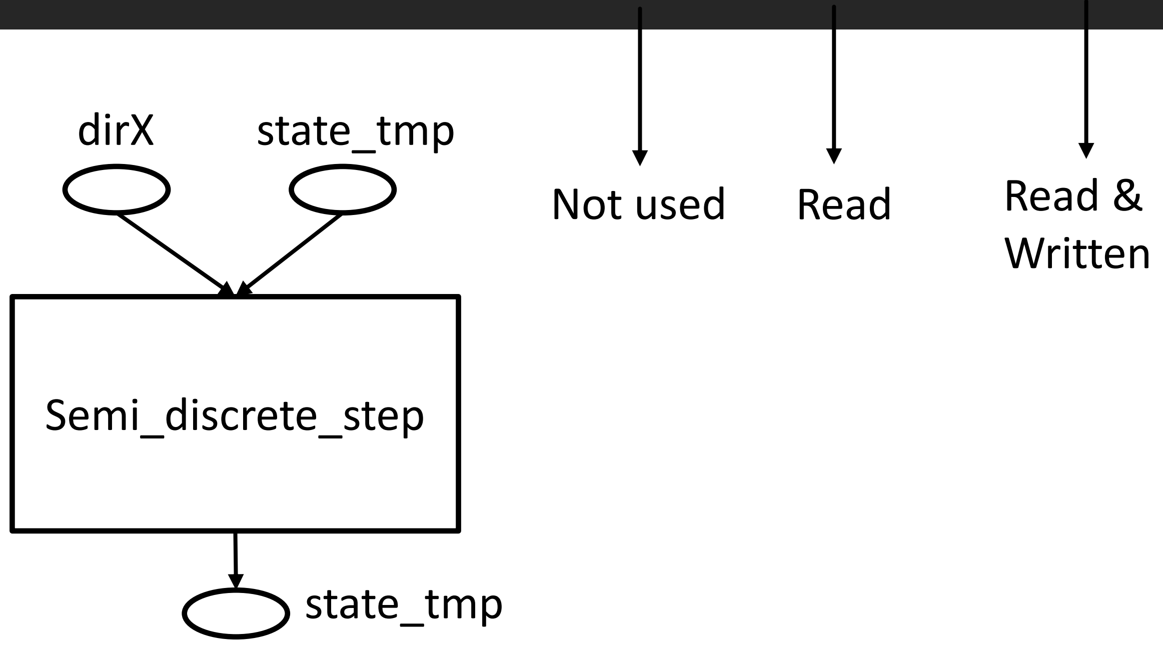
These each call 4 more functions

External nested SDFGs – Lex

- **Goal – New workflow:**
  - create SDFGs for each function first
  - do local optimization
  - load them together (potentially hierarchically)
  - do global optimization



```
call semi_discrete_step( state , dirX , state_tmp)
```



"Slotting" the external SDFG in is not necessarily trivial: What if the NestedSDFG was simplified and no longer uses all arguments? We can leverage the lessons of the frontends!