Tal Ben-Nun, Johannes de Fine Licht, Alexandros-Nikolaos Ziogas, Timo Schneider, Torsten Hoefler

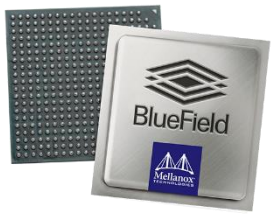
Stateful Dataflow Multigraphs: A Data-Centric Model for Performance Portability on Heterogeneous Architectures

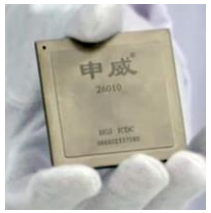
This project has received funding from the European Research Council (ERC) under grant agreement "DAPP (PI: T. Hoefler)".

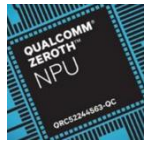
Motivation


intel
Stratix® 10


KINTEX
UltraSCALE


BlueField
Mellanox


申威


QUALCOMM
ZEROTH
NPU

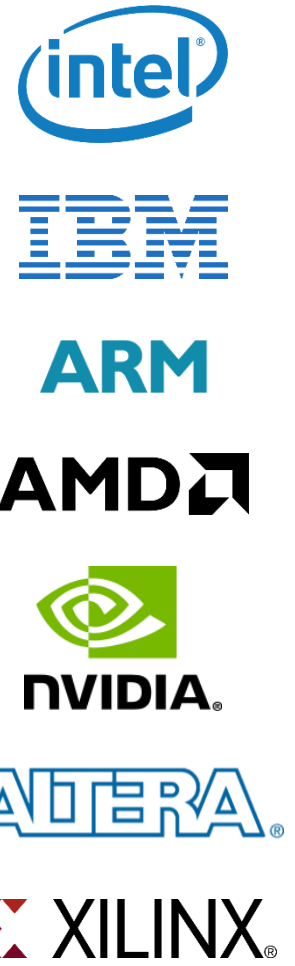
Computational Scientist

Domain Scientist

Performance Engineer

Optimization Techniques

- **Multi-core CPU**
  - Tiling for complex cache hierarchies
  - Register optimizations
  - Vectorization
- **Many-core GPU**
  - Coalesced memory access
  - Warp divergence minimization, register tiling
  - Task fusion
- **FPGA**
  - Maximize resource utilization (logic units, DSPs)
  - Streaming optimizations, pipelining
  - Explicit buffering (FIFO) and wiring



DaCe Overview

Domain Scientist

Problem Formulation

$$\frac{\partial u}{\partial t} - \alpha \nabla^2 u = 0$$

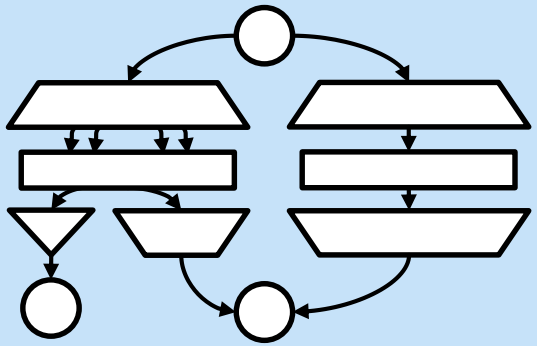
Python | DSLs

TensorFlow | MATLAB

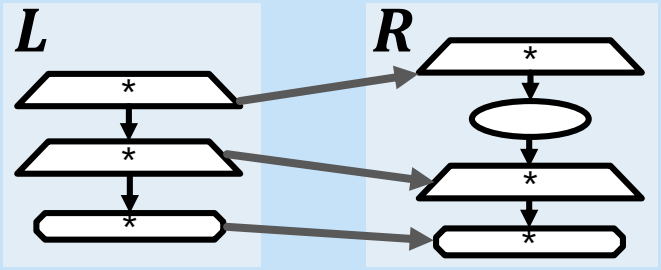
...

Scientific Frontend

Performance Engineer



Data-Centric Intermediate Representation (SDFG)



Graph Transformations

Transformed Dataflow

Performance Results

System

Hardware Information

Compiler

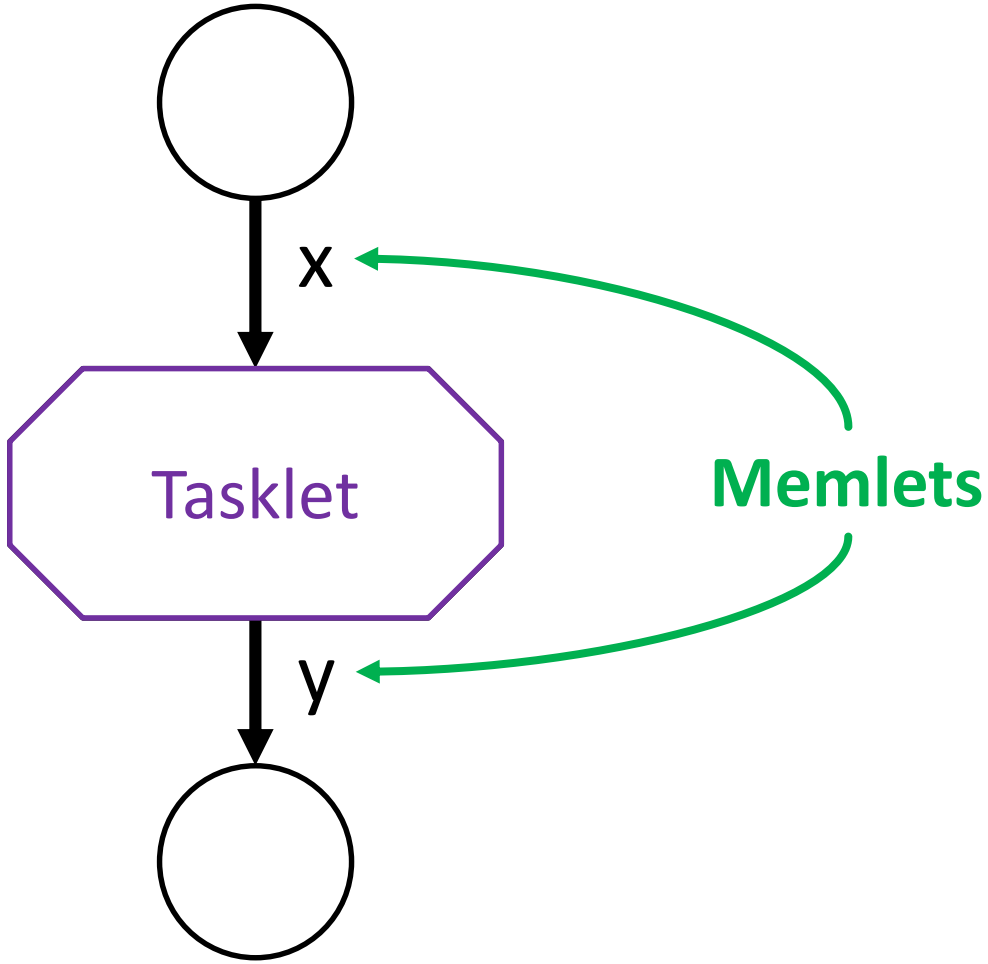
Runtime

CPU Binary

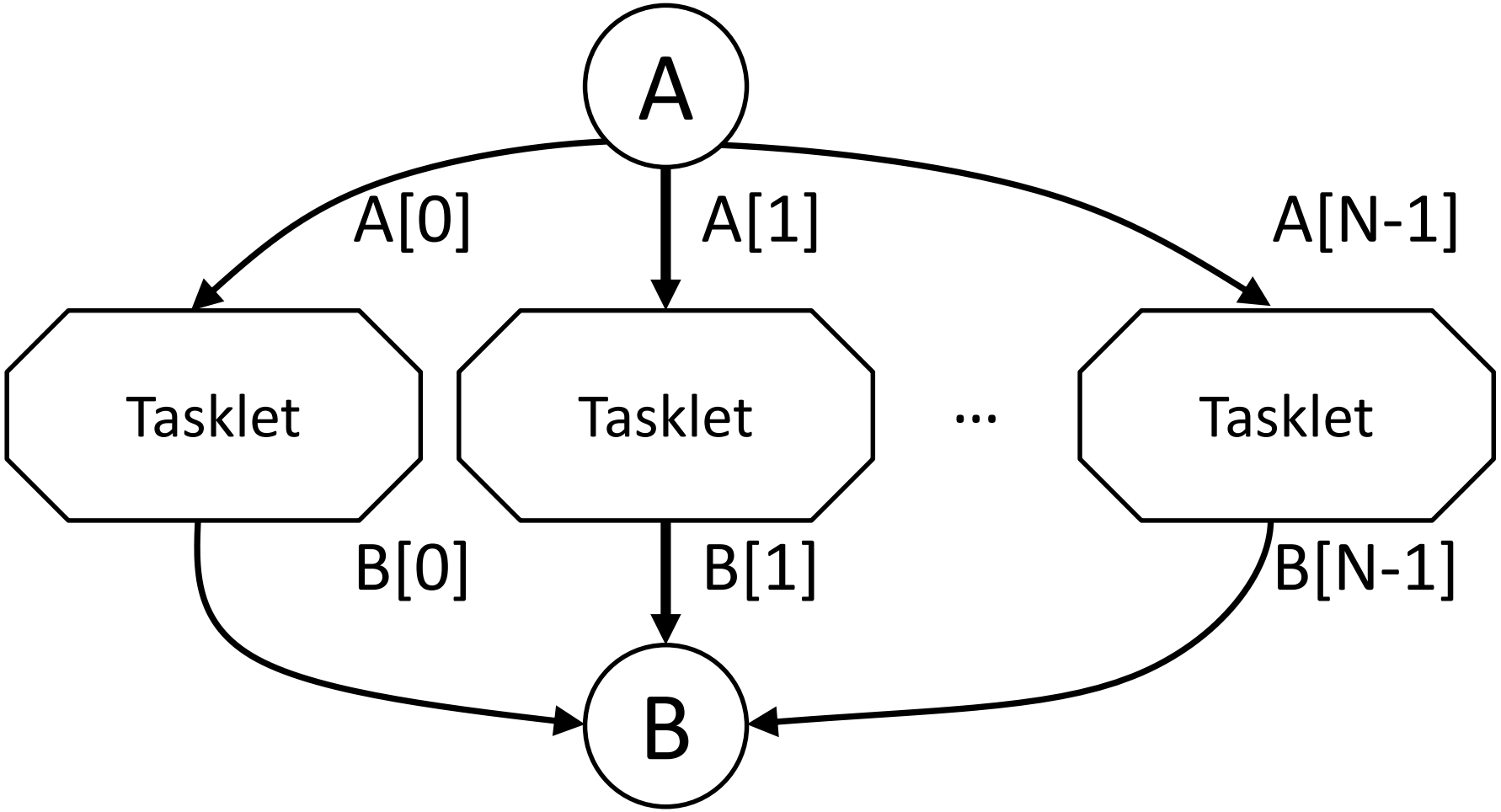
GPU Binary

FPGA Modules

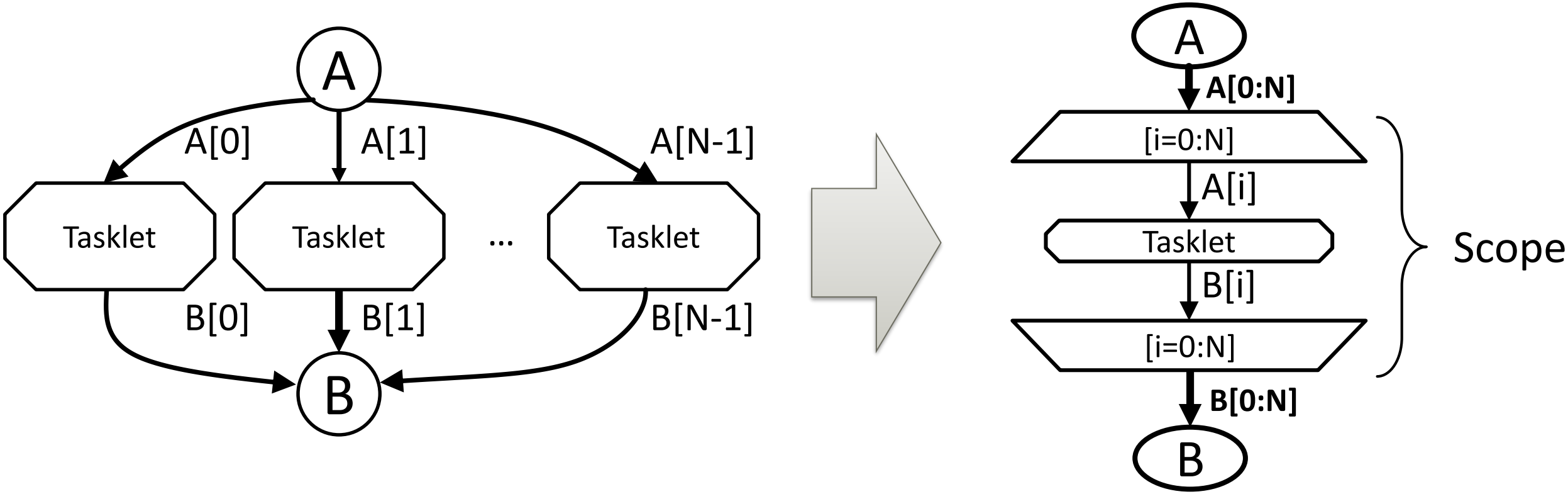
Dataflow Programming in DaCe



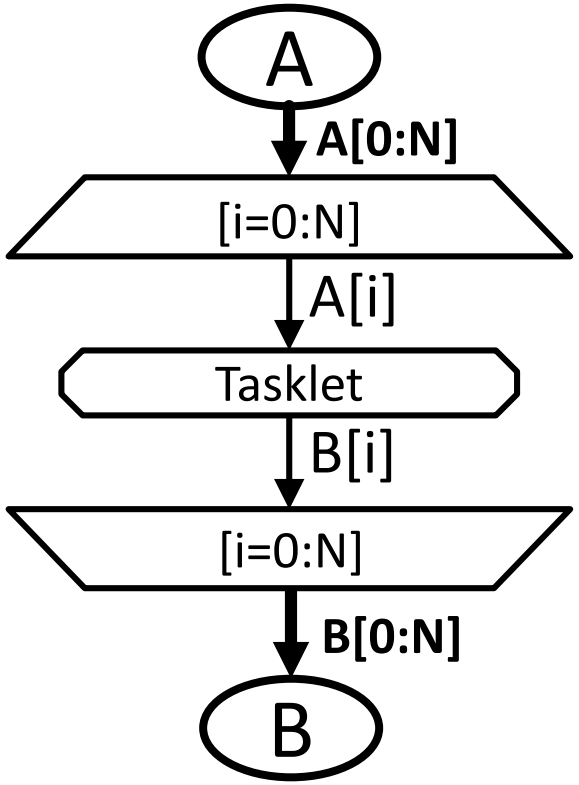
Parallel Dataflow Programming

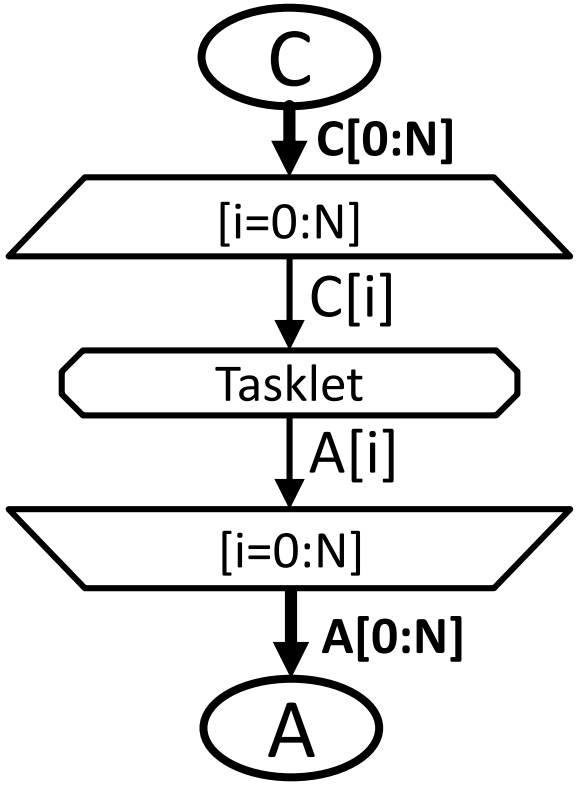


Parallel Dataflow Programming

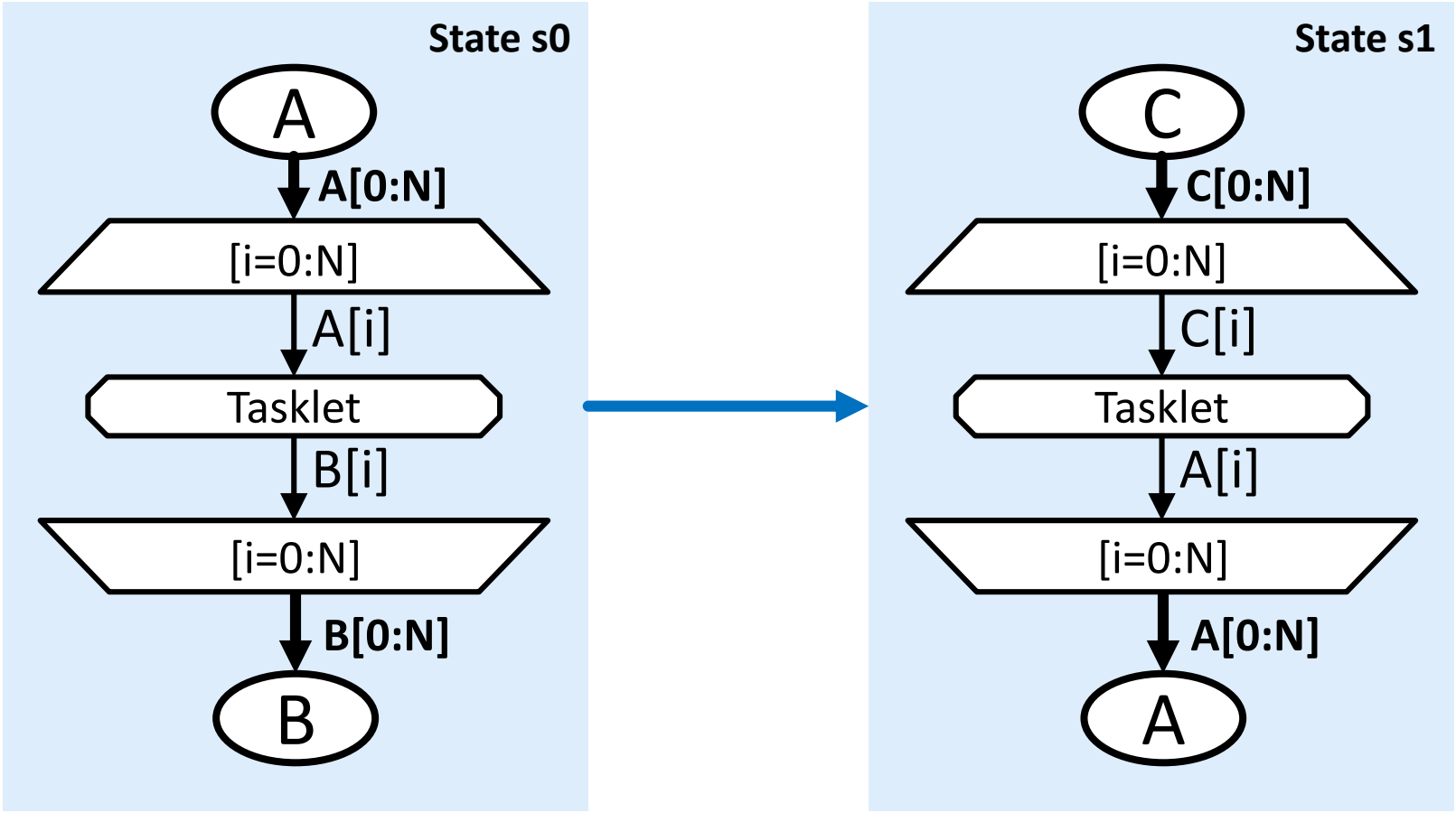


Stateful Parallel Dataflow Programming

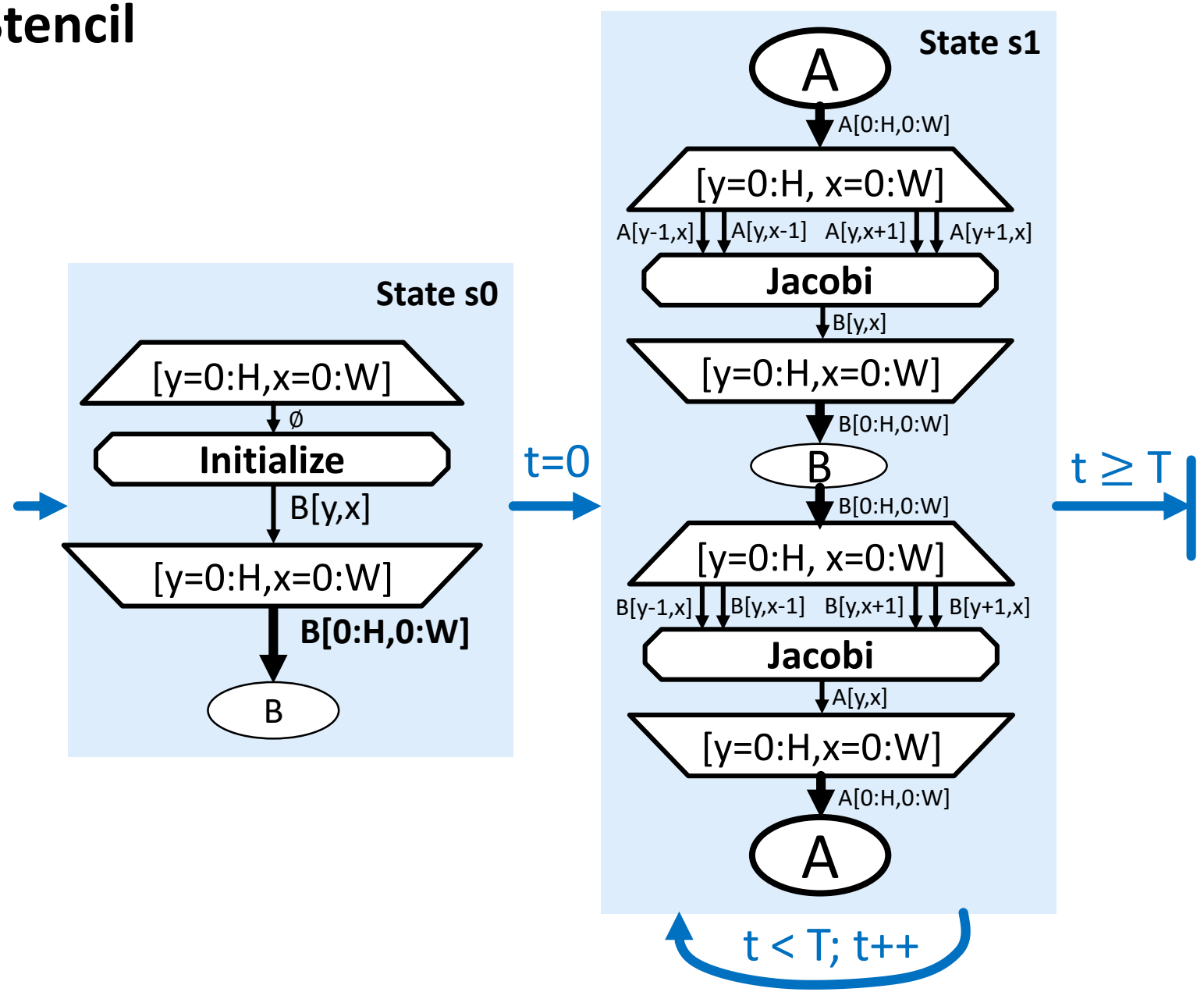




Stateful Parallel Dataflow Programming



Example: 2D Stencil

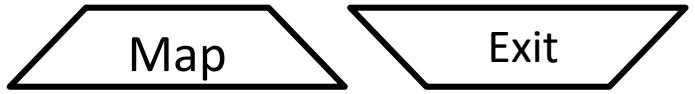


Meet the Nodes

State — State machine element

Fine-grained computational block

N-dimensional data container

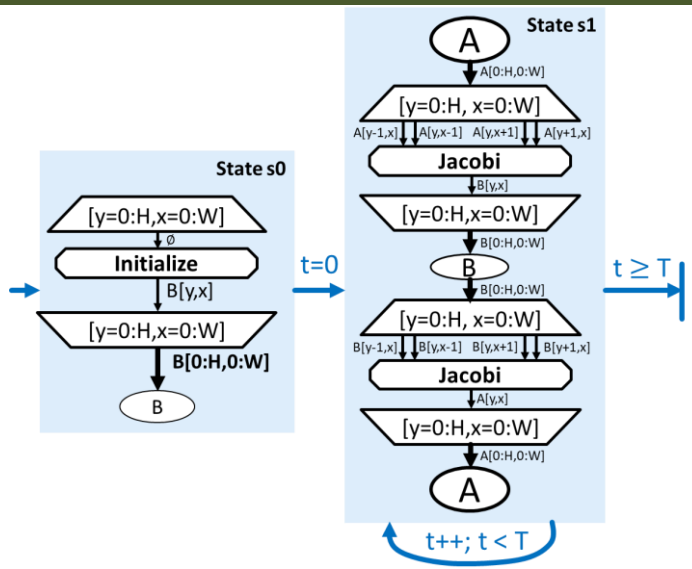


Parametric graph abstraction for parallelism

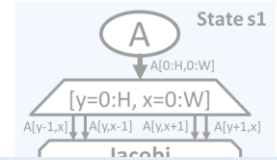
Streaming data container

Dynamic mapping of computations on streams

Defines behavior during conflicting writes



Meet the Nodes



form *symbol = expression*. Once a state finishes execution, all outgoing state transitions of that state are evaluated in an arbitrary order, and the destination of the first transition whose condition is true is the next state which will be executed. If no transition evaluates to true, the program terminates. Before starting the execution of the next state, all assignments are performed, and the left-hand side of assignments become symbols.

A.2 Operational Semantics

A.2.1 Initialization. ***Notation*** We denote collections (sets/lists) as capital letters and their members with the corresponding lowercase letter and a subscript, i.e., in an SDFG $G \equiv (S, T, s_0)$ the set of states S as s_i, with $0 \leq i < |S|$. Without loss of generality we assume s_0 to be the start state. We denote the value stored at memory location a as $M[a]$, and assume all basic types are size-one elements to simplify address calculations.

The state of execution is denoted by ρ. Within the state we carry several sets: *loc*, which maps names of data nodes and transients to memory addresses; *sym*, which maps symbol names (identifiers) to their current value; and *vis*, which maps connectors to the data visible at that connector in the current state of execution.

We define a helper function *size*(), which returns the product of all dimensions of the data node or element given as argument (using ρ to resolve symbolic values). Furthermore, *id*() returns the *name* property of a data or transient node, and *offs*() the offset of a data element relative to the start of the memory region it is stored in. The function *copy*() creates a copy of the object given as argument, i.e., when we modify the copy, the original object remains the same.

Invocation When an SDFG G is called with the data arguments $A \equiv [a_i = p_i]$ (a_i is an identifier, p_i is an address/pointer) and symbol arguments $Z \equiv [z_i = v_i]$ (z_i is an identifier, v_i an integer) we initialize the configuration ρ:

(1) For all symbols z_i in Z: $sym[z_i] \leftarrow v_i$.
(2) For all data and stream nodes $d_i \in G$ without incoming edges s.t. $id(d_i) = a_i$: $loc(d_i) \leftarrow p_i, vis(d_i.data) \leftarrow M[p_i, ..., p_i + size(d_i)]$.
(3) Set *current* to a copy of the start state of G, s_0.
(4) Set *state* to $id(s_0)$.
(5) Set $qsize[f_i]$ to zero for all stream nodes $f_i \in G$.

This can be expressed as the following rule:

$$\frac{\begin{array}{l} G = (S, T), \\ start\_state(G) \mapsto s_0, \\ D : \text{data nodes in } s_0, \\ F : \text{stream nodes in } s_0 \end{array}}{\begin{array}{ll} (call(G, A, Z), \rho) \rightarrow \rho[& state \mapsto id(s_0), \\ & current \mapsto copy(s_0), \\ & \forall a_i = p_i \in A : loc[a_i \mapsto p_i], \\ & \forall z_i = v_i \in Z : sym[z_i] \mapsto v_i, \\ & \forall d_i \in D : vis[d_i.data] \mapsto M[p_i, ..., p_i + size(d_i)], \\ & \forall f_i \in G.S : qsize[id(f_i)] \mapsto 0] \end{array}}$$

A.2.2 Propagating Data in a State. Execution of a state entails propagating data along edges, governed by the rules defined below.

Element Processing In each step, we take one element q (either a memlet or a node) of *current*, for which all input connectors have visible data, then:

If q is a **memlet** *(src, dst, subset, reindex, wcr)*, update $vis[dst]$ to $wcr(reindex(subset(vis[src])))$:

$$\frac{q = memlet(src, dst, subset, reindex, wcr), \quad (vis[src], \rho) \neq \emptyset, \quad (wcr(reindex(subset(vis[src]))), \rho) \rightarrow [d_0, ..., d_N]}{(q, \rho) \rightarrow \rho[vis[dst] \mapsto [d_0, ..., d_N]]}$$

If q is a **data** node, update its referenced memory for an input connector c_i,
$M[loc(id(q)), .., loc(id(q)) + size(vis[q.data])] = vis[q.data]$:

$$\frac{\begin{array}{l} q = data(id, dims, lit, transient), \\ (\forall x, q.c_i \in current : vis[q.c_i], \rho) \neq \emptyset, \\ (\forall x. q.c_i \in current : vis[q.c_i], \rho) \rightarrow [d_0^i, d_1^i, .., d_{k_i}^i], \\ \forall d_j^i : loc(id(q)) + offs(d_j^i) = l_j^i \end{array}}{(q, \rho) \rightarrow \rho[\forall i, k_i : M[l_j^i, .., l_j^i + size(d_j^i)] = d_j^i]}$$

If q is a **tasklet**, generate a prologue that allocates local variables for all input connectors c_i of q, initialized to $vis[c_i]$ (P_1), as well as output connectors (P_2). Generate an epilogue Ep which updates $\rho[vis[c_i] \mapsto v_i]$ for each output connector c_i of q with the contents of the appropriate variable (declared in P_2). Execute the concatenation of $(P_1; P_2; code; Ep;)$.

$$\frac{\begin{array}{l} q = tasklet(Cin, Cout, code), \\ (\forall x, q.c_i \in current : vis[q.c_i], \rho) \neq \emptyset, \\ (P_1 = [\forall c_i \in Cin : type(c_i) id(c_i) = vis[c_i]], \rho), \\ P_2 = [\forall c_i \in Cout : type(c_i) id(c_i)], \\ (Ep = [\forall c_i \in Cout : &vis(c_i) = id(c_i)], \rho), \end{array}}{(q, \rho) \rightarrow \rho[exec(P_1; P_2; code; Ep;)]}$$

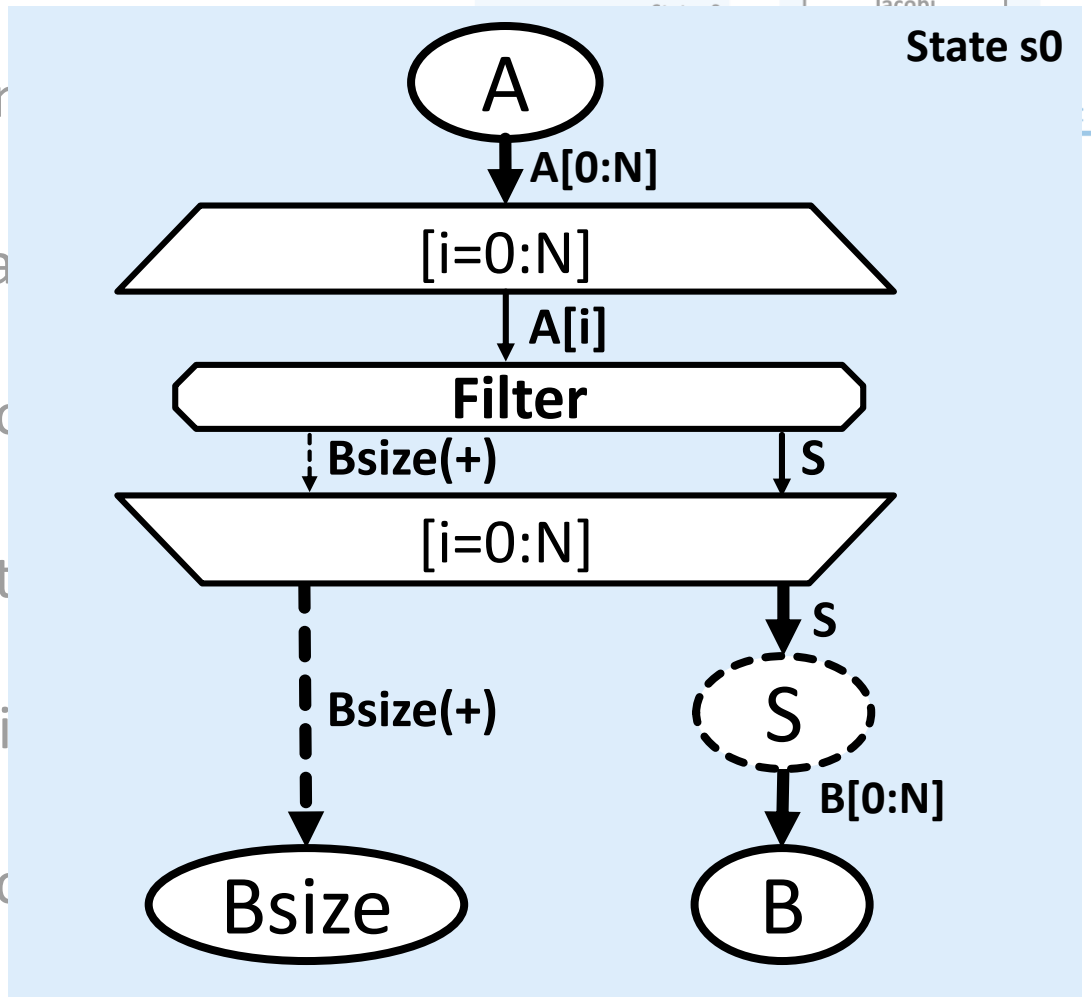
If q is a **mapentry** node with range $y = R$ (y is the identifier) and scope $o \subset V$: Remove o from *current*. Remove q and the corresponding map exit node from o. For each element in $r_i \in R$, replicate o, resolve any occurrence of y to r_i, connect incoming connectors of q and p in *state*.

$$\frac{\begin{array}{l} q = mapentry(Cin, Cout, R), \\ \forall cin_i : vis[cin_i] \neq \emptyset, \\ o = scope(q), o' = scope(q) \setminus \{q, mexit(q)\}, \\ NewSyms = [cin_i : \exists (cin_i, y)] \end{array}}{\begin{array}{ll} (q, \rho) \rightarrow \rho[& current \mapsto o' \cup [\forall r_i : reasym(copy(o'), r_i)], \\ & \forall n_i \in NewSym : sym[n_i] \mapsto vis[cin_i]] \end{array}}$$

If q is a **consume-entry** node, defined by *(range, cond, cin, cout)*, replace q with a mapentry and do the same for the corresponding consume exit node. Then we create a new SDFG *new*, which contains the contents of the original consume scope $scope(q)$. *new* consists of one state s_0, and a single state transition to the same state with the condition *cond*, defined by $(s_0, s_0, cond, [])$. Finally, we replace $scope(q)$ in *current* with an invoke node for *new* and reconnect the appropriate edges between the entry and exit nodes.

$$\frac{\begin{array}{l} q = consume - entry(range, cond, cin, cout) \\ newsdfg = SDFG(scope(q) \setminus \{q, cexit(q)\}, (s_0, s_0, cond, [])) \\ iv = invoke(newsdfg) \\ men = mapentry(range, cin, cout) \\ mex = mapexit(cexit(q).cin, cexit(q).cout) \end{array}}{(q, \rho) \rightarrow \rho[current \mapsto (current \setminus q, cexit(q)) \cup (iv, men, mex)]}$$

If q is a **reduce** node defined by the tuple *(cin, cout, range)*, we create a mapentry node *men* with the same range, a mapexit node *mex*, and a tasklet `o = i`. We add these nodes to the node set of

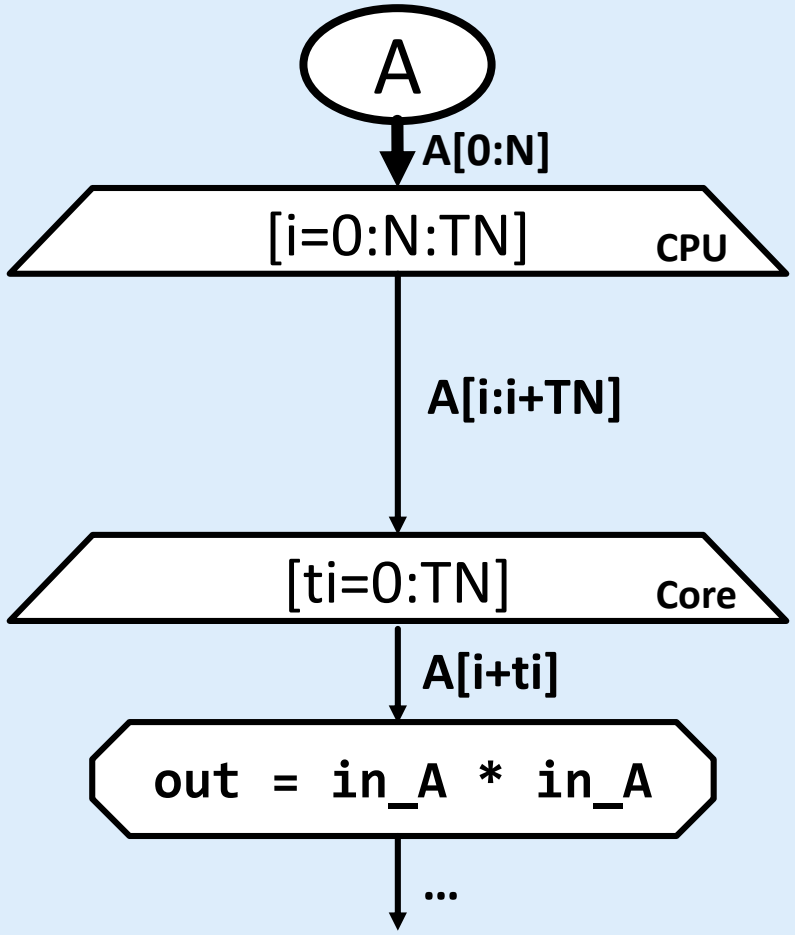


Conflict Resolution

Defines behavior during conflicting writes

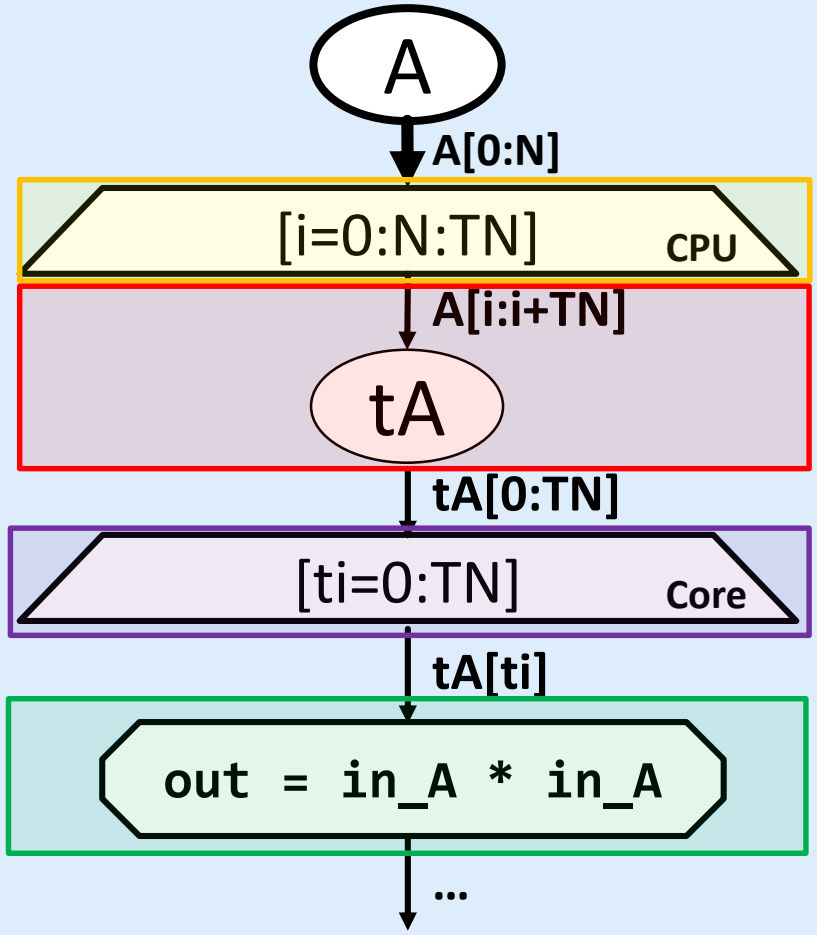
Hierarchical Parallelism and Heterogeneity

- Maps have schedules, arrays have storage locations



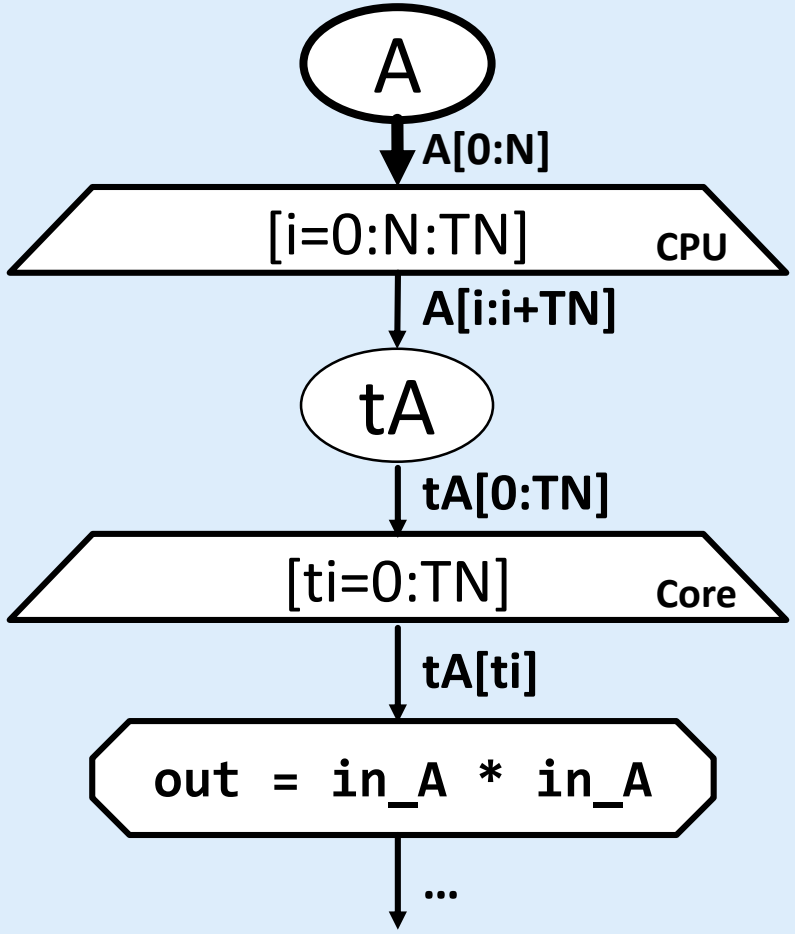
Hierarchical Parallelism and Heterogeneity

- Maps have schedules, arrays have storage locations

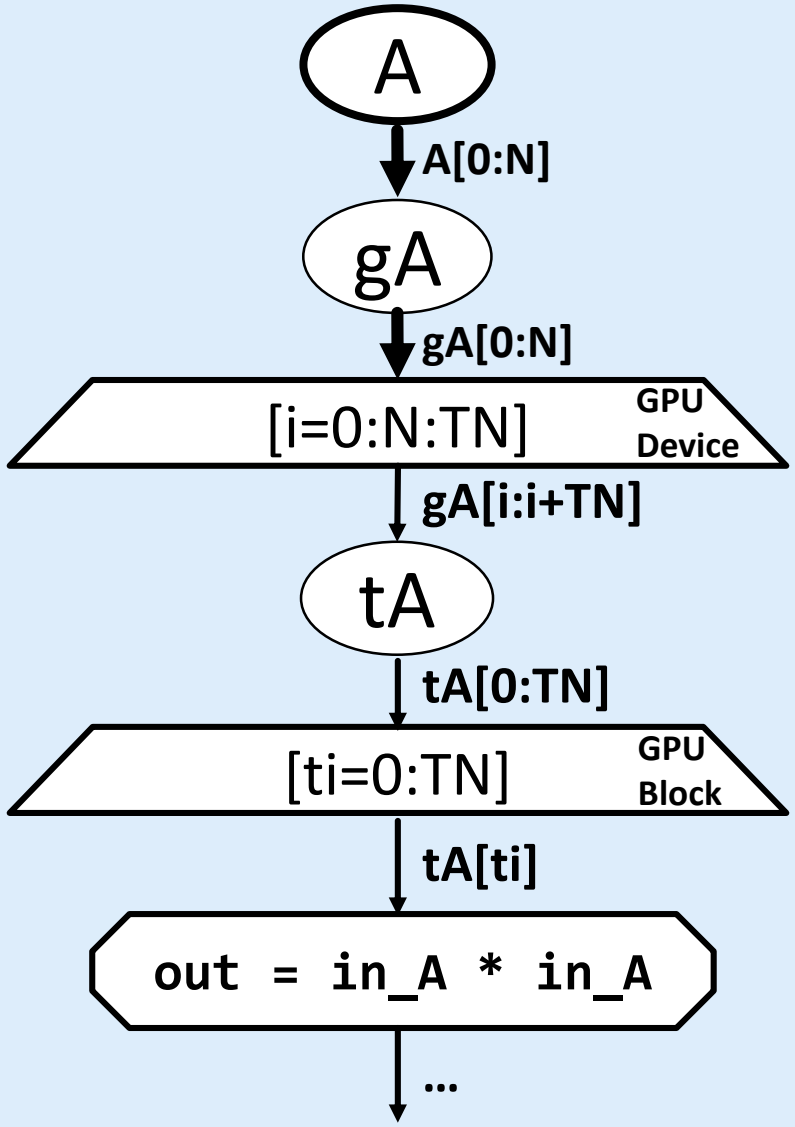



```
// ...
#pragma omp parallel for
for (int i = 0; i < N; i += TN) {
    vec<double, 4> tA[TN];
    Global2Stack_1D<double, 4, 1> (
        &A[i], min(N - i, TN), tA);

    for (int ti = 0; ti < TN; ti += 1) {

        vec<double, 4> in_A = tA[ti];
        auto out = (in_A * in_A);
        tC[ti] = out;
    }
```

Hierarchical Parallelism and Heterogeneity



Hierarchical Parallelism and Heterogeneity




```
__global__ void multiplication_1(...) {
    int i = blockIdx.x * TN;
    int ti = threadIdx.y + 0;
    if (i+ti >= N)  return;

    __shared__ vec<double, 2> tA[TN];
    GlobalToShared1D<double, 2, TN, 1, 1, false>(gA, tA);

    vec<double, 2> in_A = tA[ti];
    auto out = (in_A * in_A);
    tC[ti] = out;
}
```

Hardware Mapping: Load/Store Architectures

- **Recursive code generation (C++, CUDA)**

  Control flow: Construct detection and gotos

- **Parallelism**

  Multi-core CPU: OpenMP, atomics, and threads

  GPU: CUDA kernels and streams

  Connected components run concurrently

- **Memory and interaction with accelerators**

  Array-array edges create intra-/inter-device copies

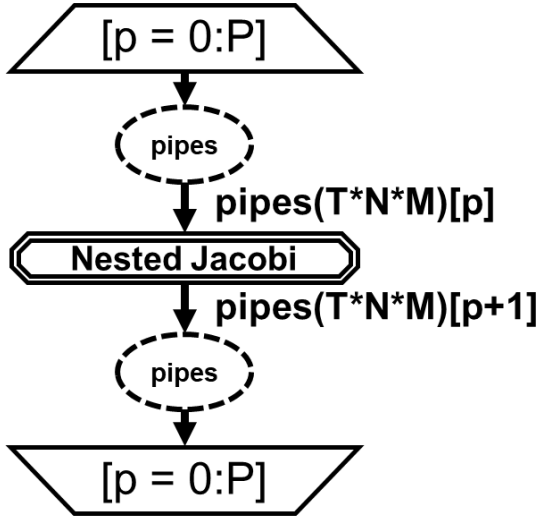
```
// ...
#pragma omp parallel for
for (int i = 0; i < N; i += TN) {
    vec<double, 4> tA[TN];
    Global2Stack_1D<double, 4, 1> (
        &A[i], min(N - i, TN), tA);

    for (int ti = 0; ti < TN; ti += 1) {

        vec<double, 4> in_A = tA[ti];
        auto out = (in_A * in_A);
        tC[ti] = out;
    }
```

Mapping to Reconfigurable Hardware

- **Module generation with HDL and HLS**

  Xilinx SDAccel

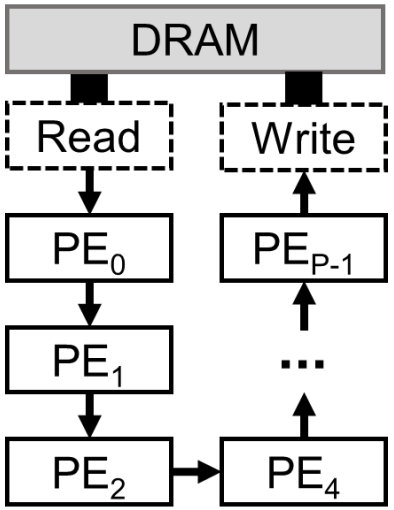
  Intel FPGA (experimental)



- **Parallelism**

  Exploiting **temporal** locality: pipelines

  Exploiting **spatial** locality: vectorization, replication



- **Replication**

  Enables parametric systolic array generation

Data-centric Parallel Programming for Python

- **Programs are integrated within existing codes**

  In Python, integrated functions in existing code

  In MATLAB, separate .m files

  In TensorFlow, takes existing graph

```
@dace.program
def program_numpy(A, B):
  B[:] = np.transpose(A)
```

- **In Python: Implicit and Explicit Dataflow**

  Implicit: numpy syntax

  Explicit: Enforce memory access decoupling from computation

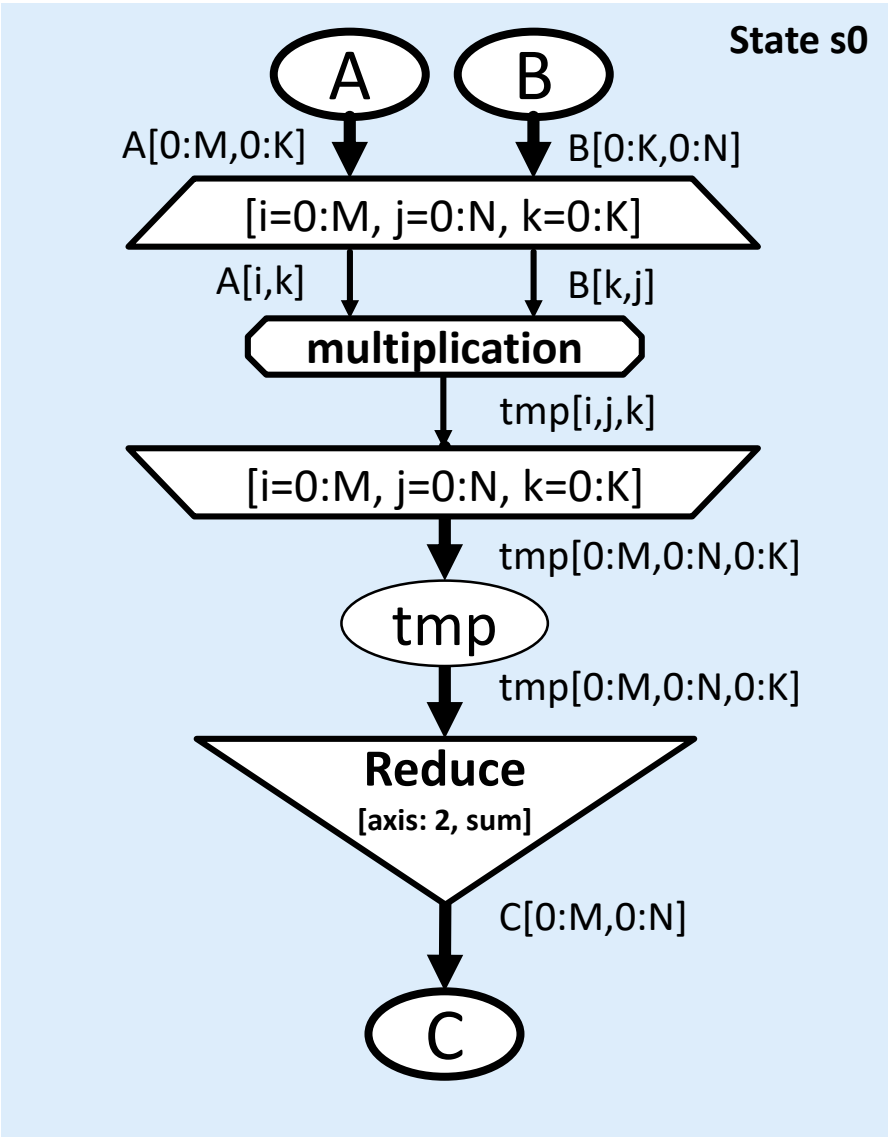
```
@dace.program
def program_explicit(A, B):

  @dace.map
  def transpose(i: _[0:N],
                j: _[0:M]):
    a << A[i,j]
    b >> B[j,i]

    b = a
```

- **Output compatible with existing programs**

  C-compatible SO/DLL file with autogenerated include file

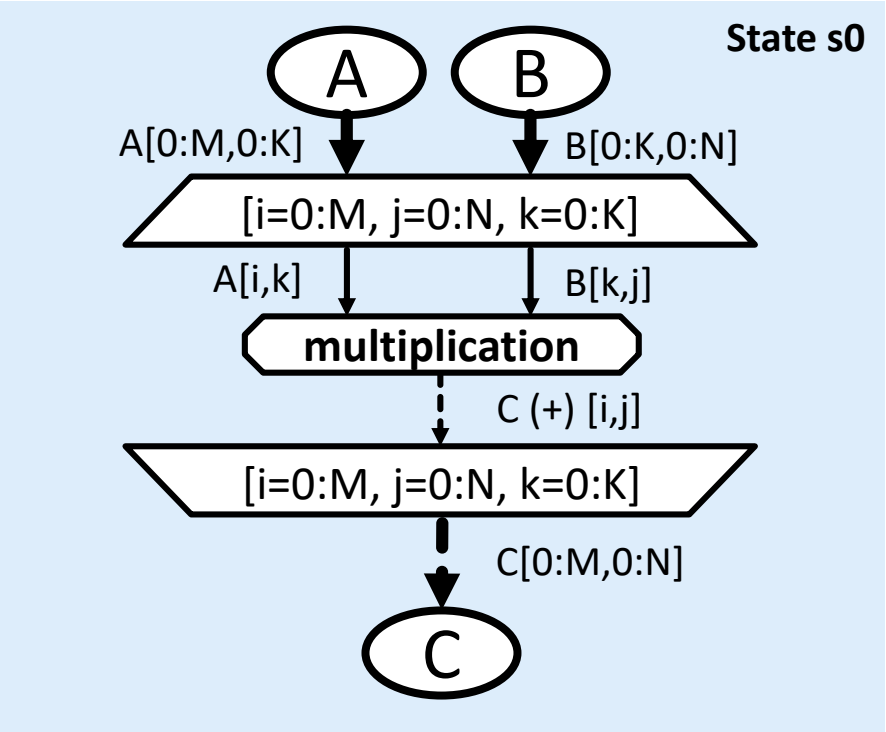
Matrix Multiplication SDFG

```
@dace.program
def gemm(A: dace.float64[M, K], B: dace.float64[K, N],
         C: dace.float64[M, N]):
    # Transient variable
    tmp = np.ndarray([M, N, K], dtype=A.dtype)

    @dace.map
    def multiplication(i: _[0:M], j: _[0:N], k: _[0:K]):
        in_A << A[i,k]
        in_B << B[k,j]
        out  >> tmp[i,j,k]

        out = in_A * in_B

    dace.reduce(lambda a, b: a + b, tmp, C, axis=2)
```

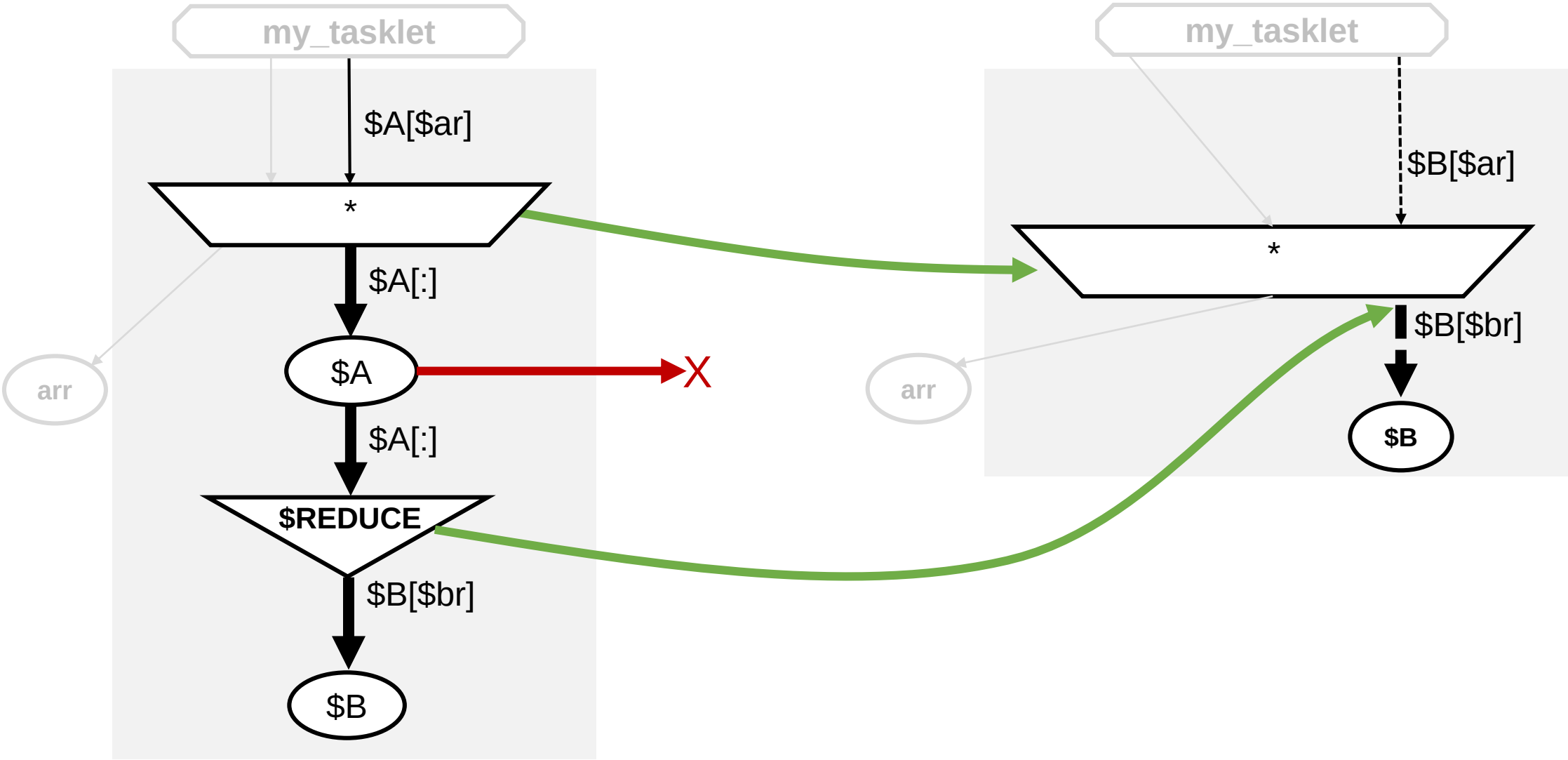


Matrix Multiplication SDFG

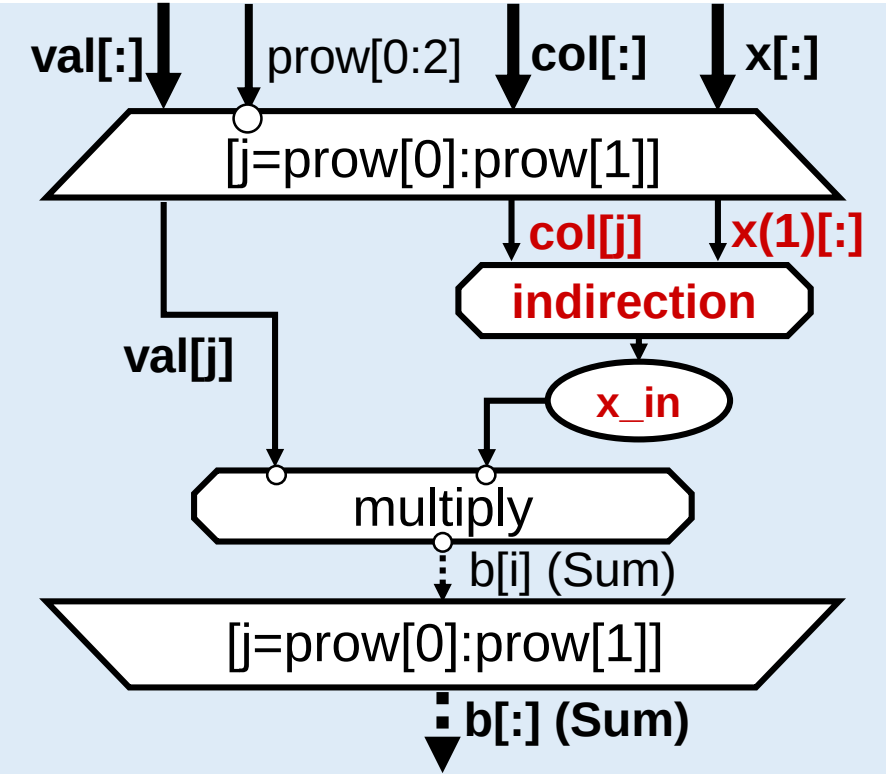
```
@dace.program
def gemm(A: dace.float64[M, K], B: dace.float64[K, N],
         C: dace.float64[M, N]):
    # Transient variable
    tmp = np.ndarray([M, N, K], dtype=A.dtype)

    @dace.map
    def multiplication(i: _[0:M], j: _[0:N], k: _[0:K]):
        in_A << A[i,k]
        in_B << B[k,j]
        out  >> tmp[i,j,k]

        out = in_A * in_B

    dace.reduce(lambda a, b: a + b, tmp, C, axis=2)
```



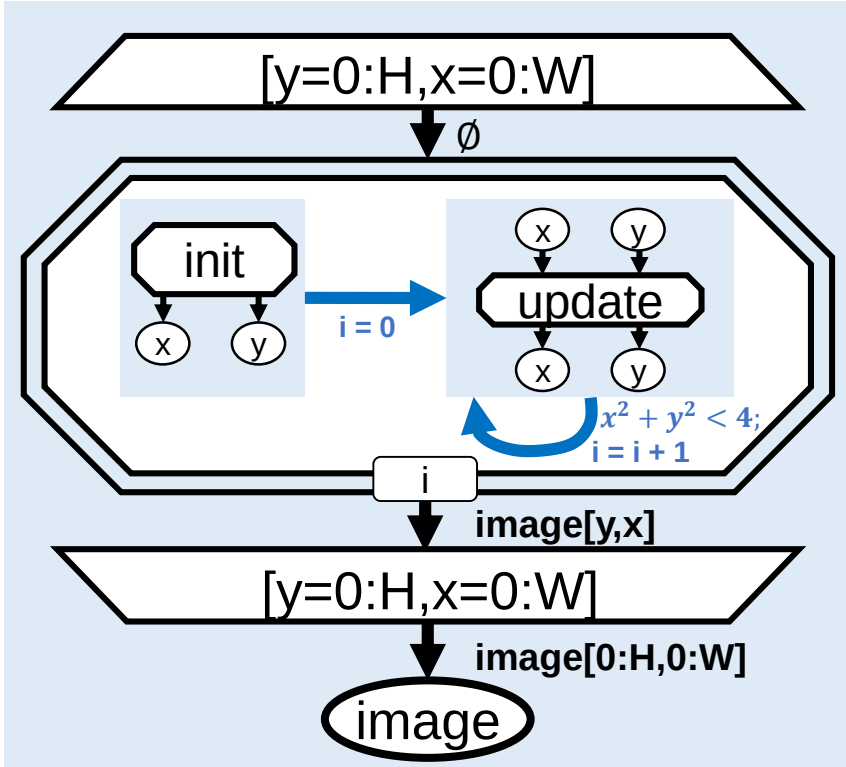
MapReduceFusion Transformation



Programming Model Challenges

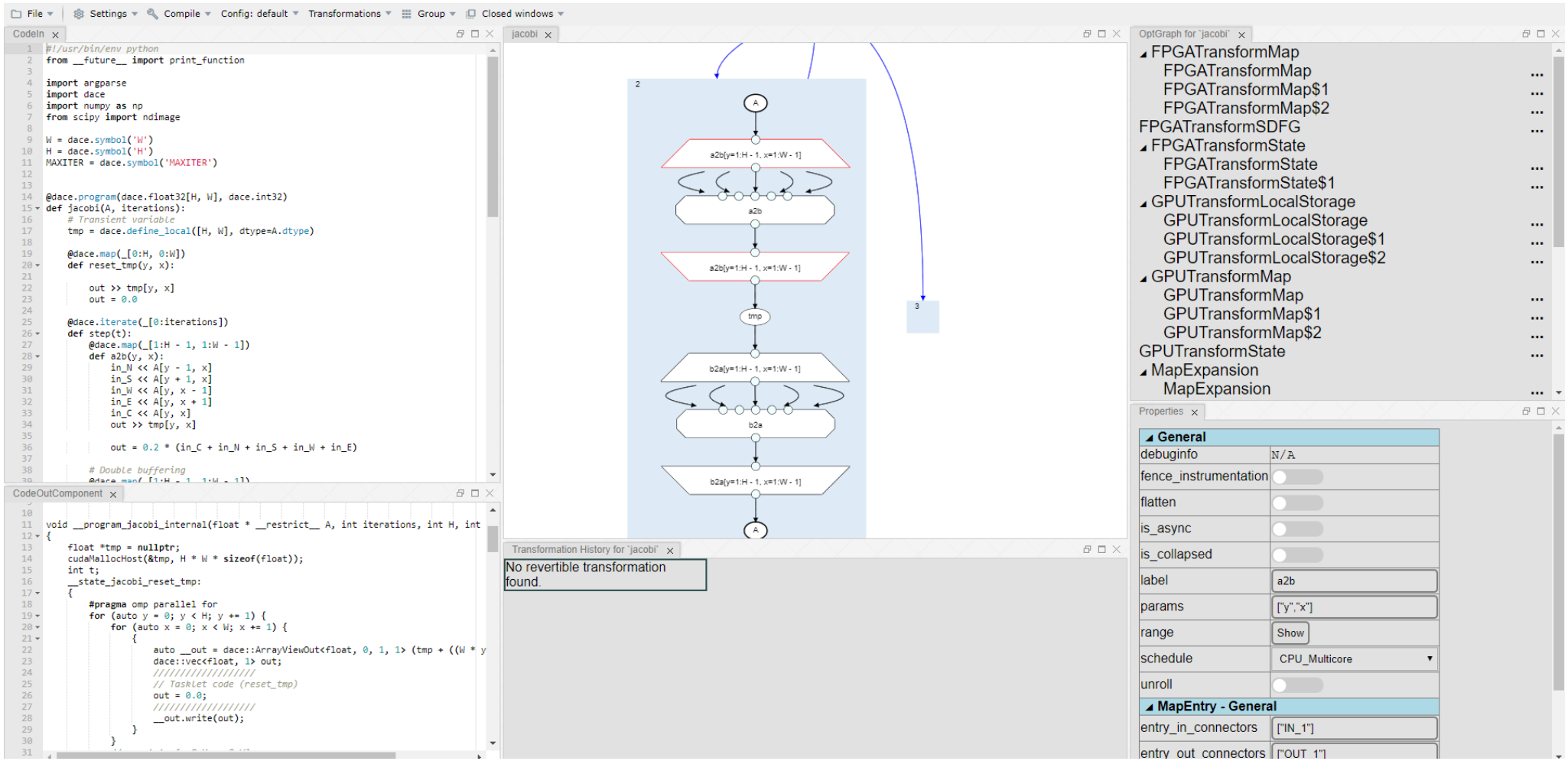


Indirect memory access

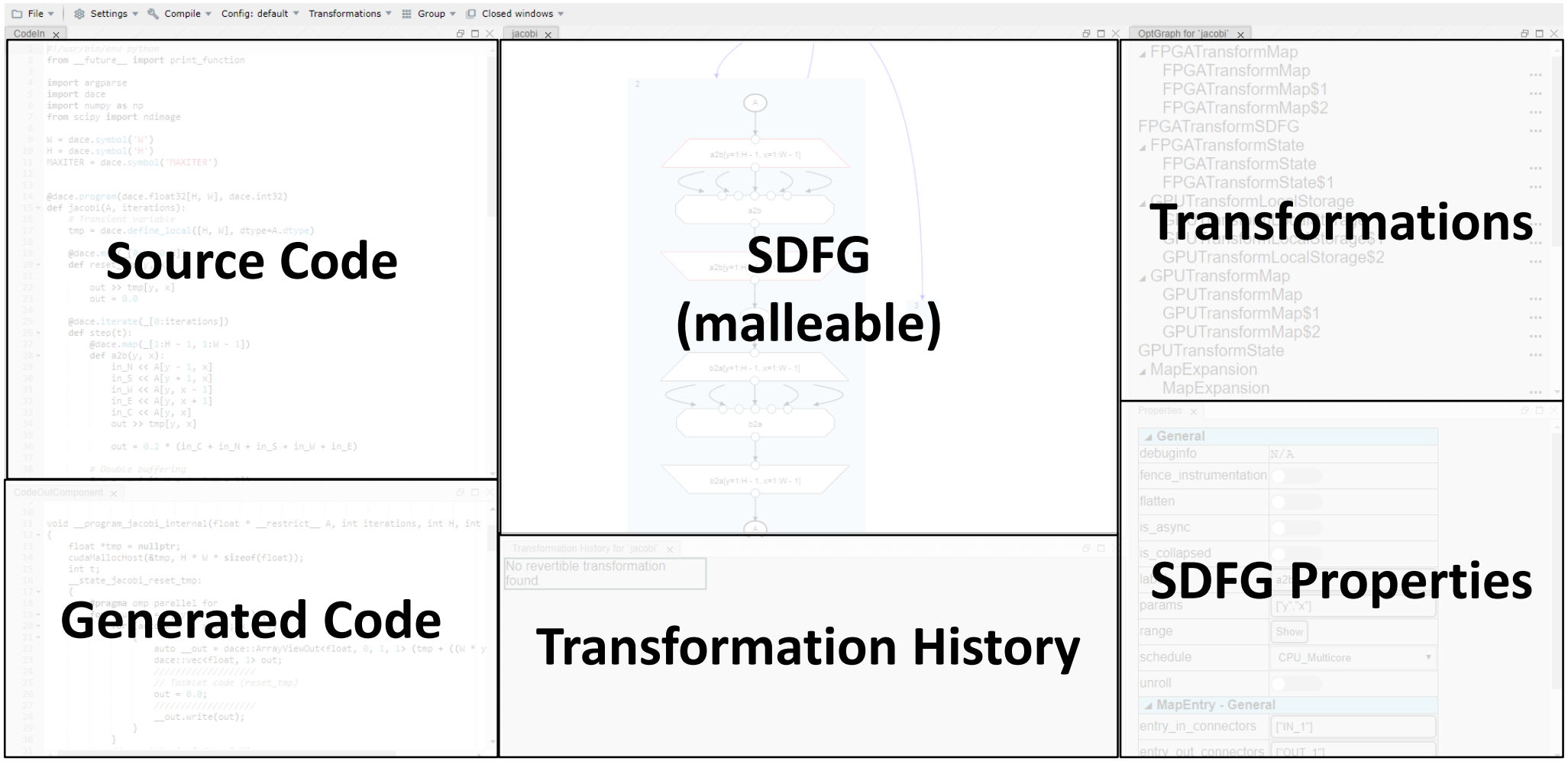


Nested state machines

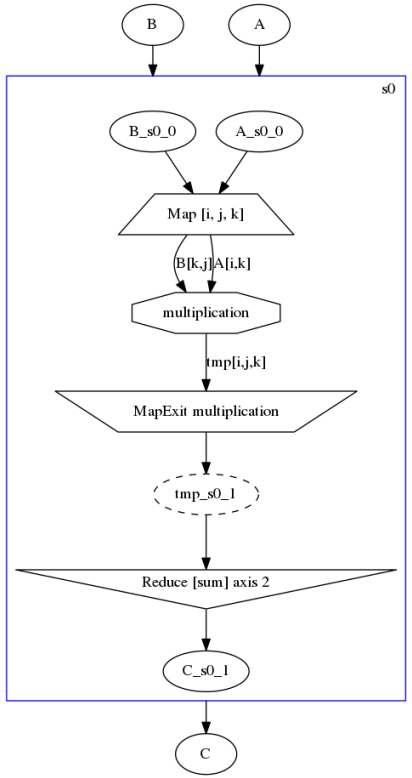
DIODE (or: Data-centric Integrated Optimization Development Environment)

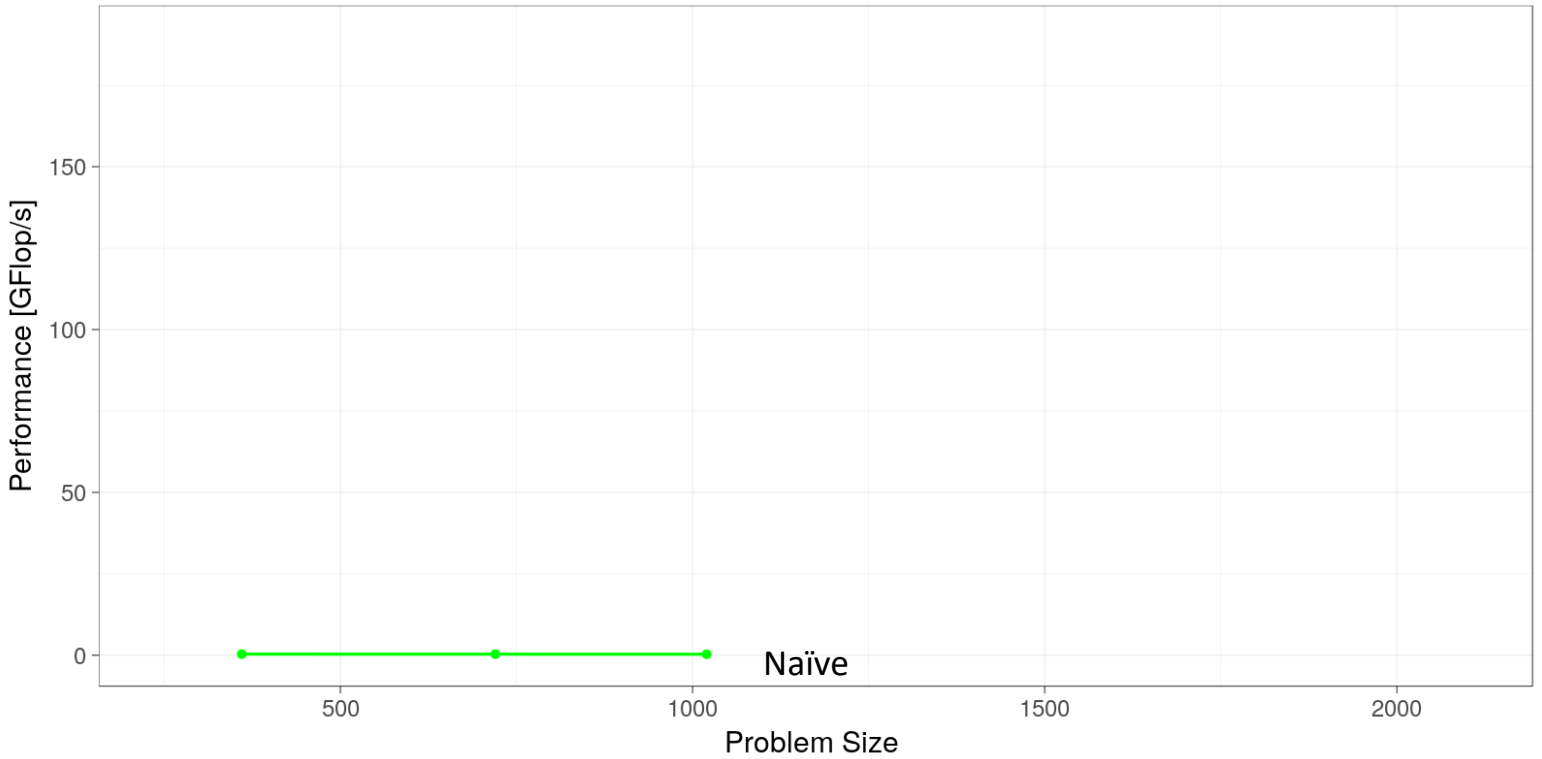


DIODE (or: Data-centric Integrated Optimization Development Environment)



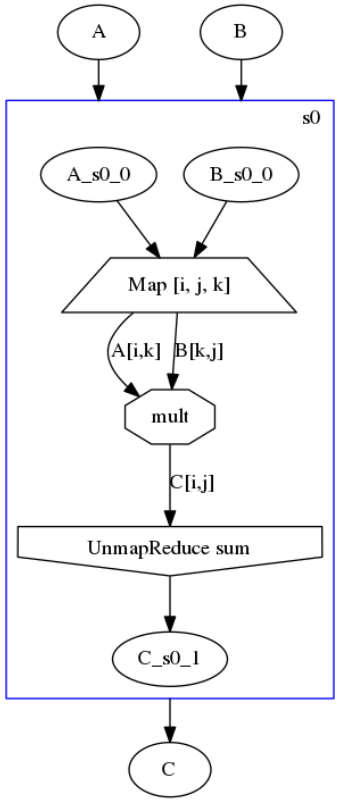
Performance



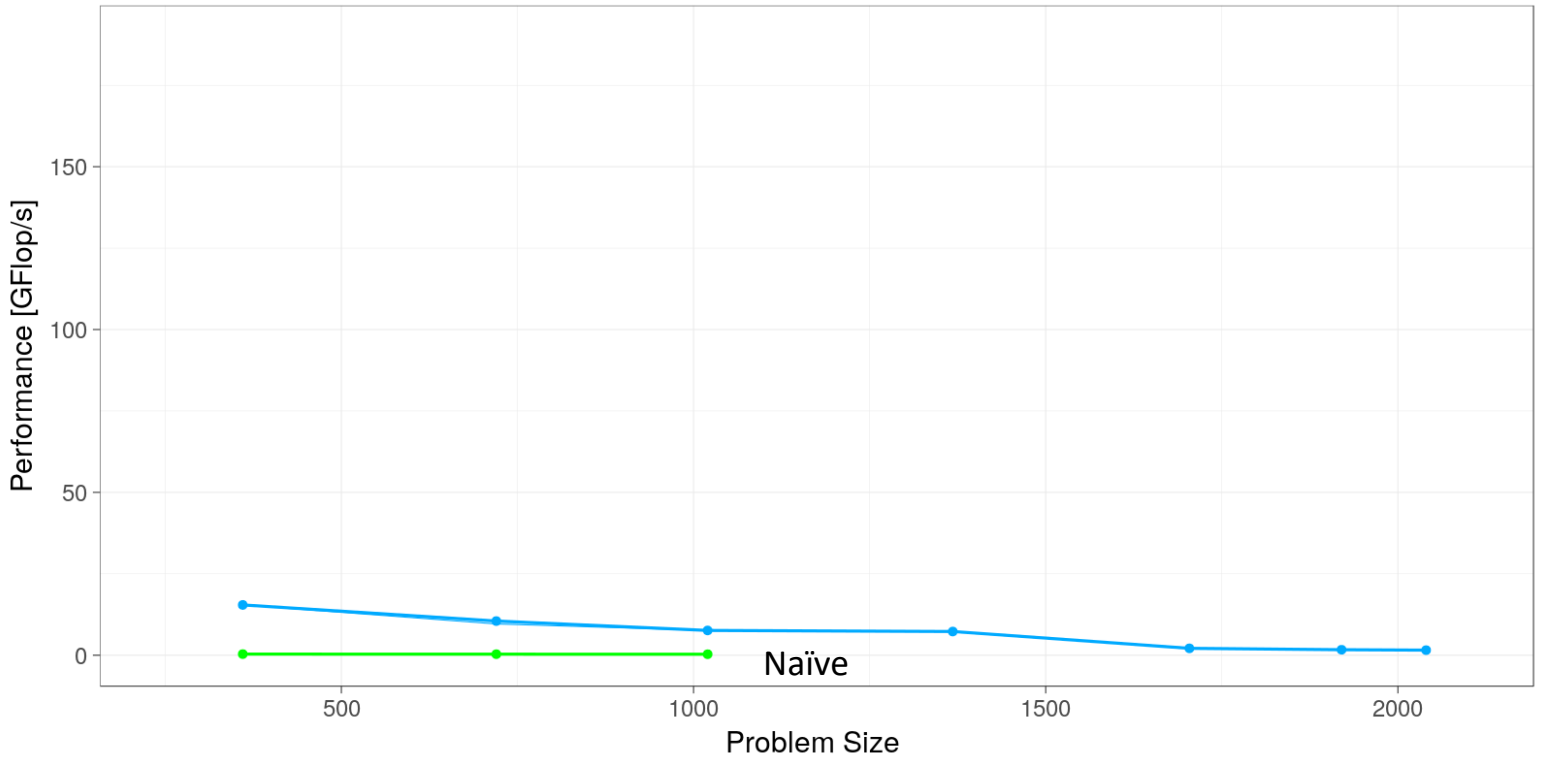


SDFG

Performance

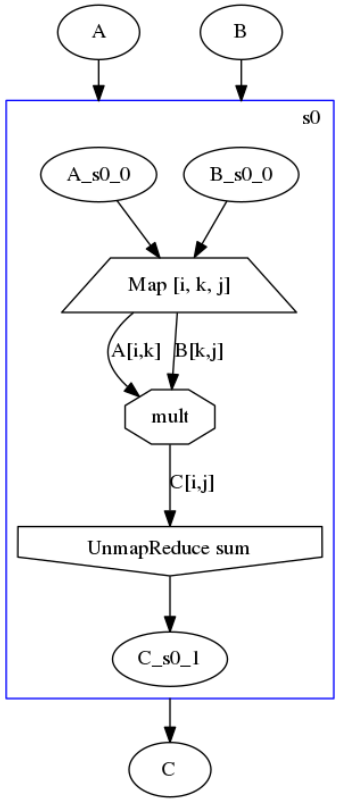


SDFG

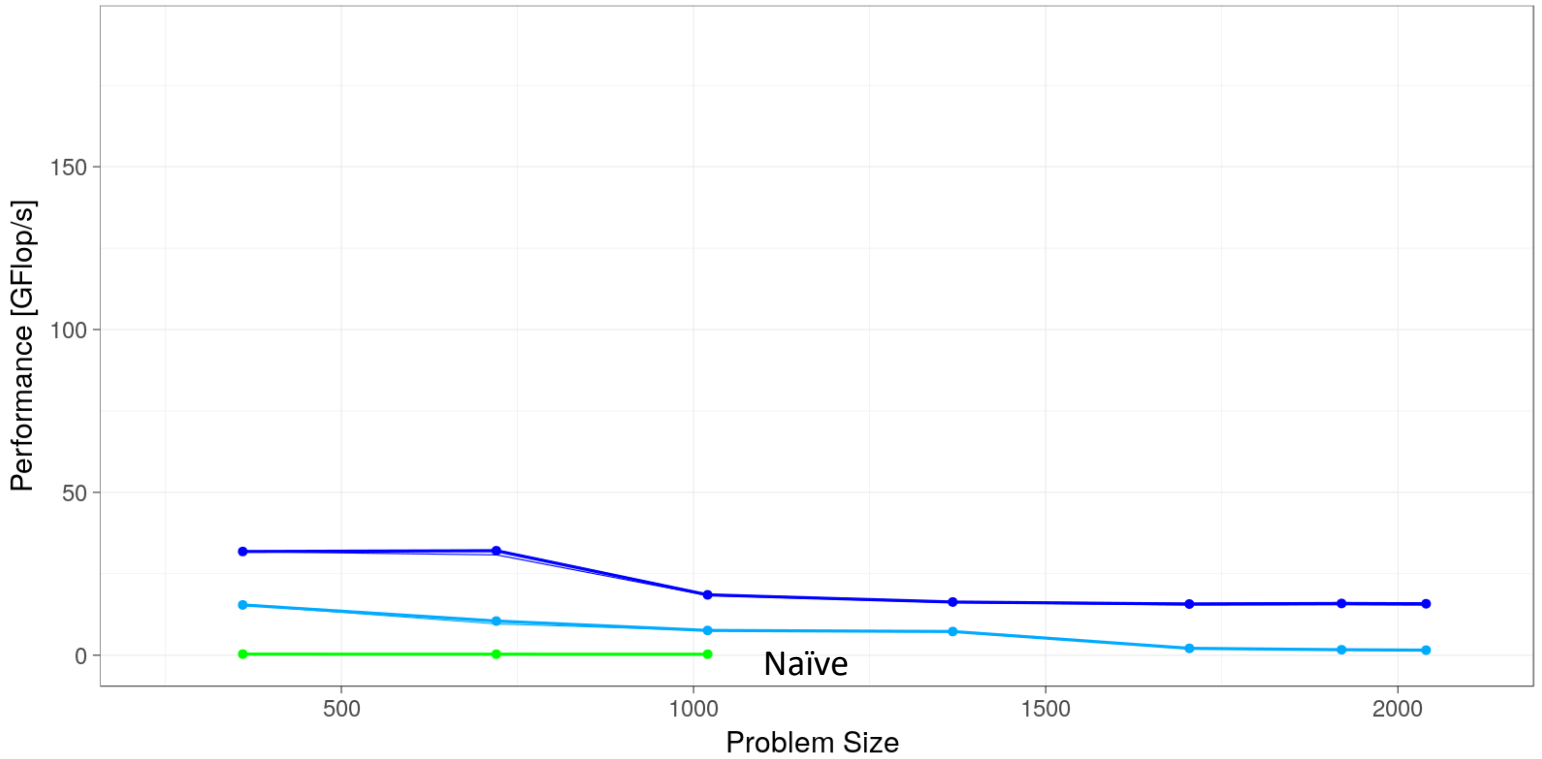


MapReduceFusion

Performance

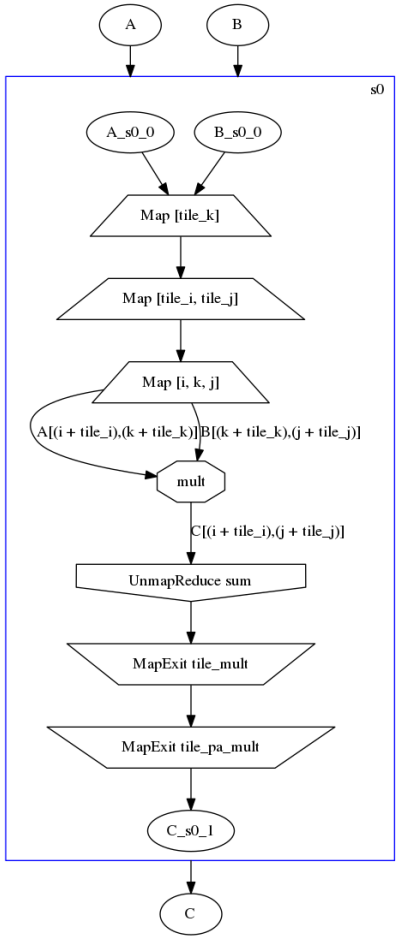


SDFG

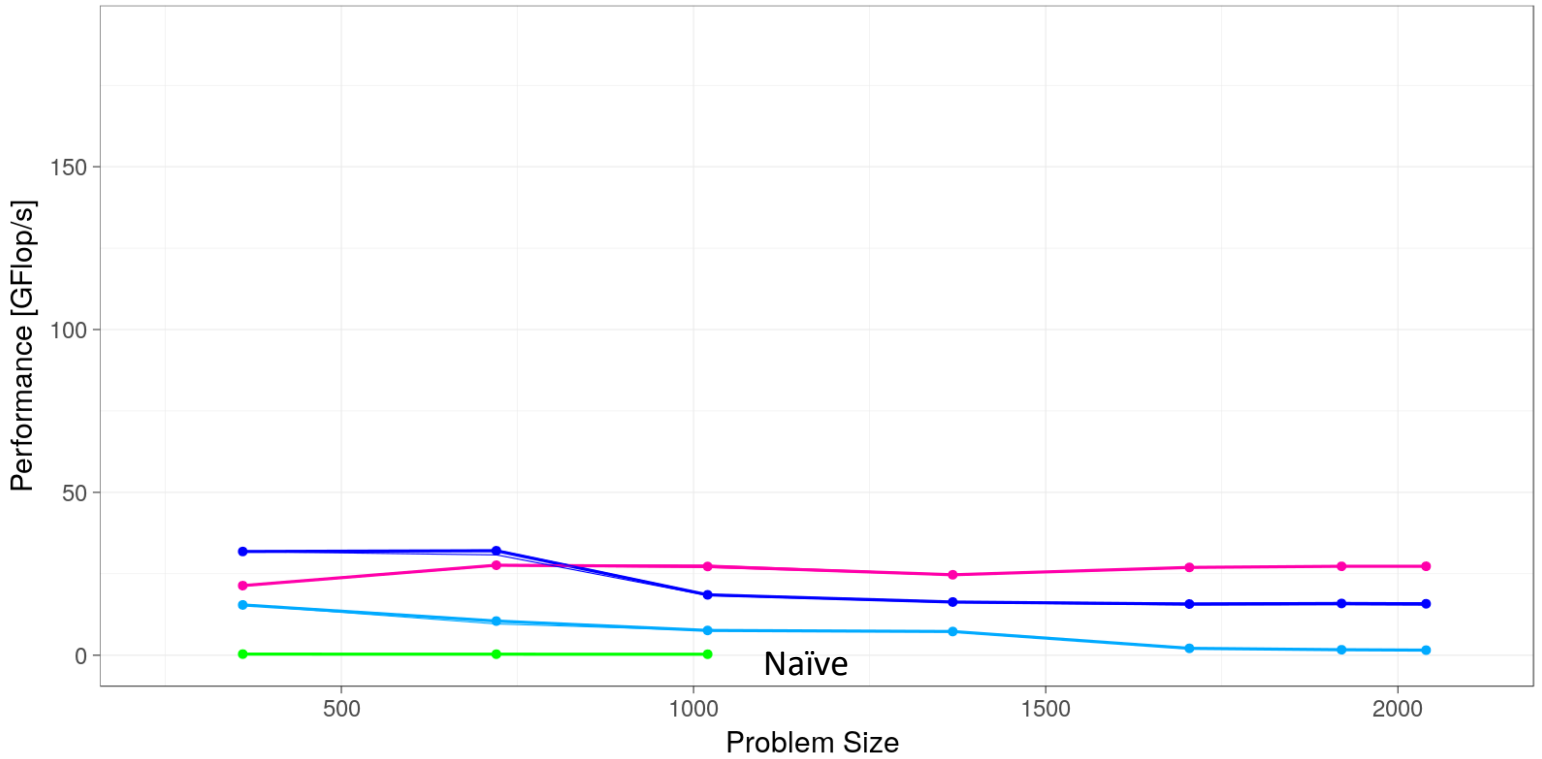


LoopReorder
MapReduceFusion

Performance

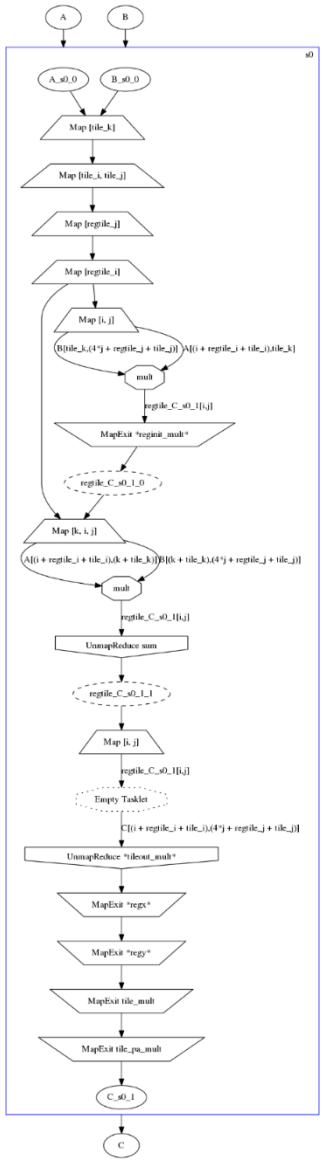


SDFG

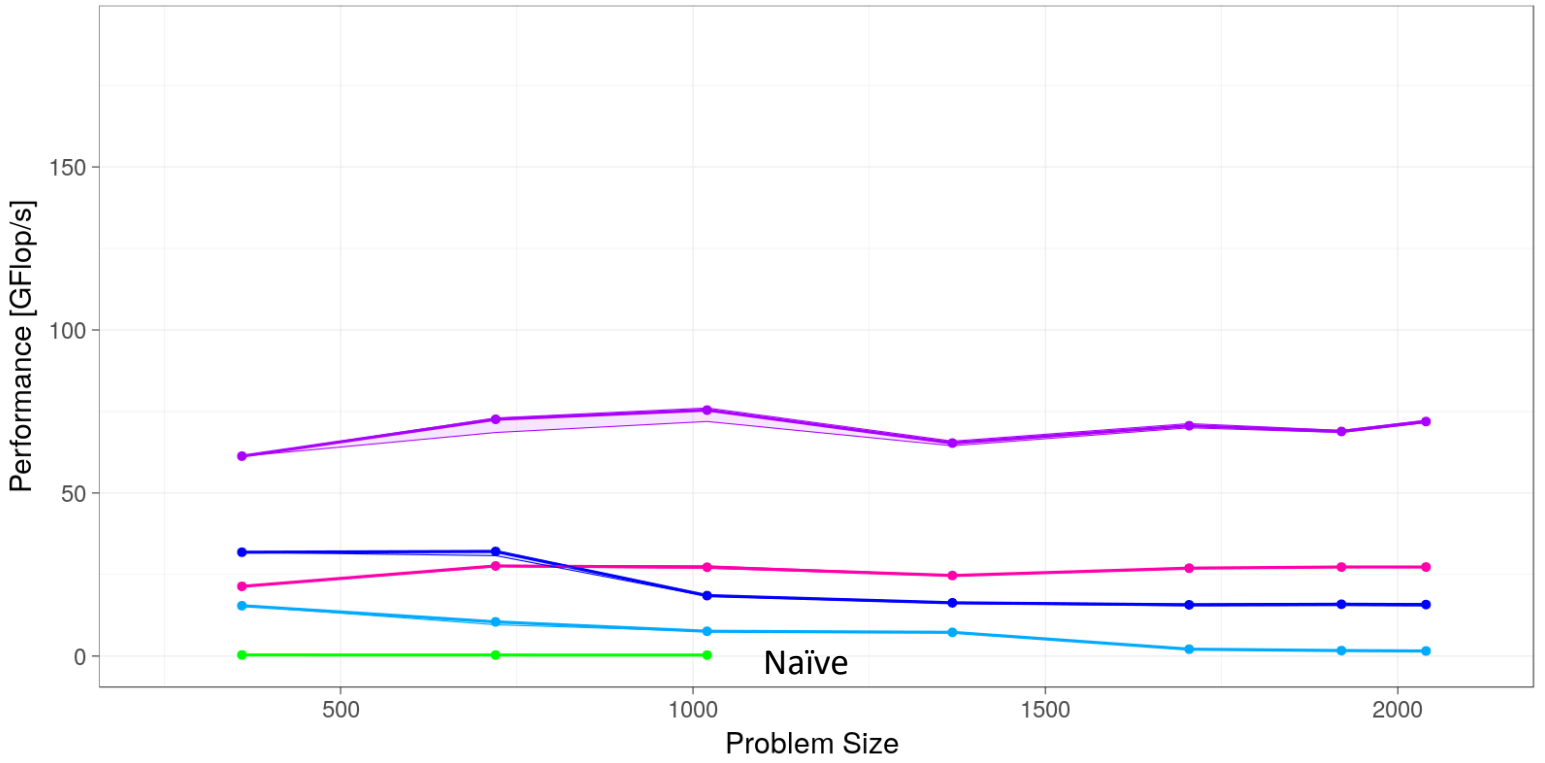


BlockTiling
LoopReorder
MapReduceFusion

Performance



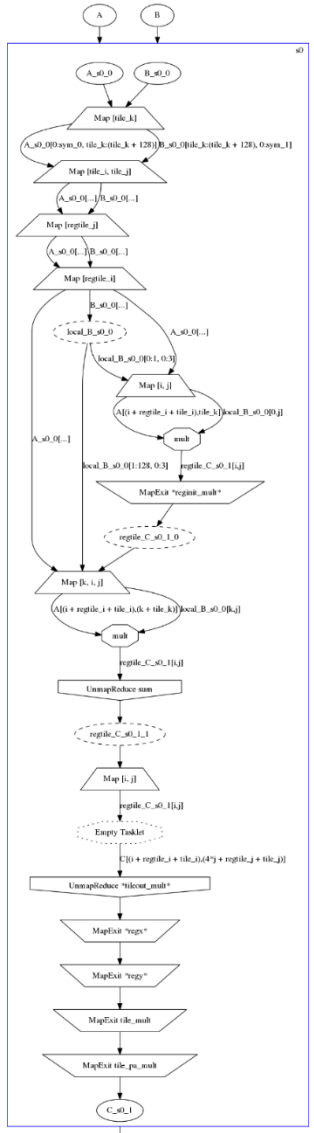
SDFG



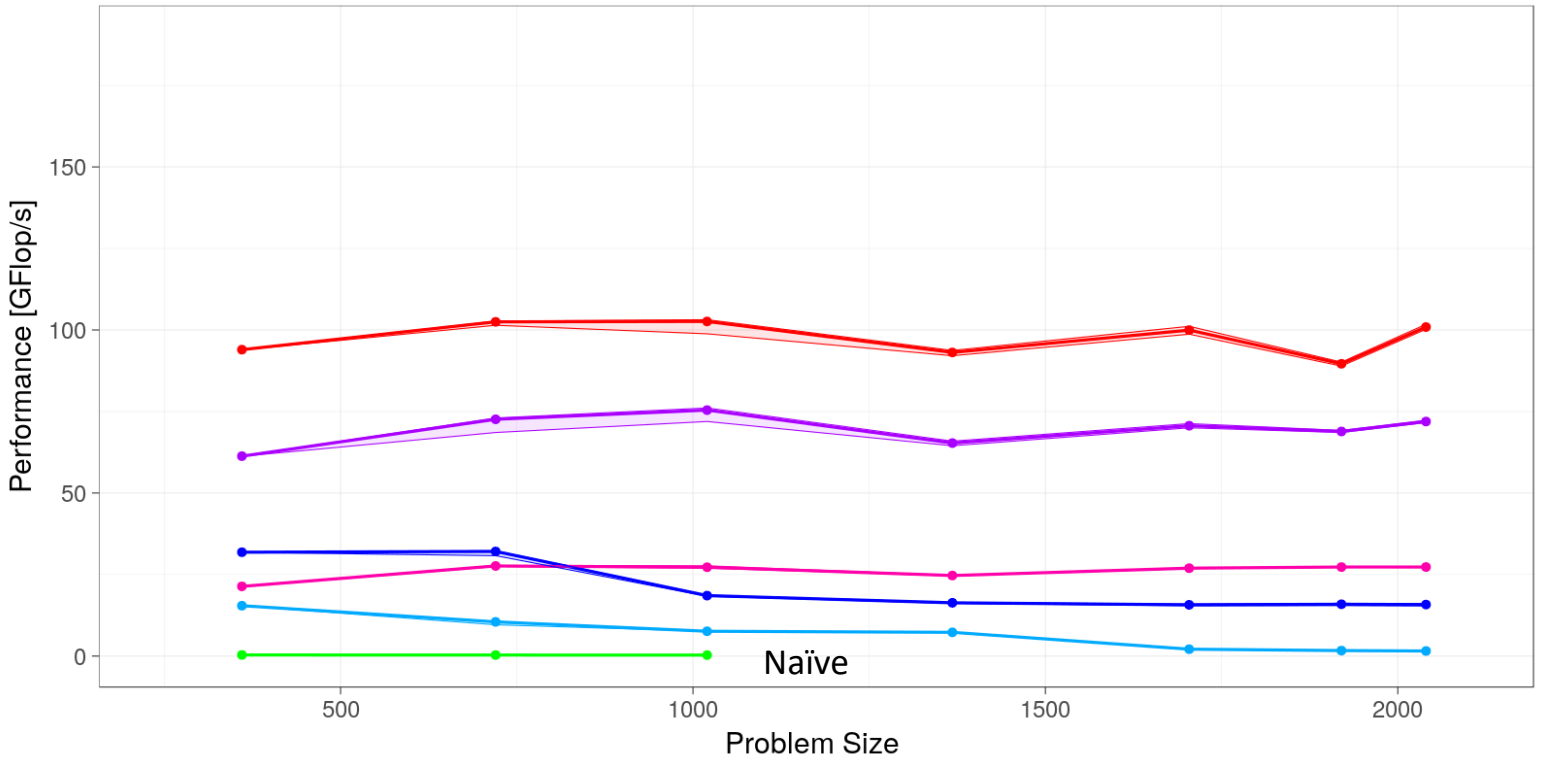
RegisterTiling

BlockTiling
LoopReorder
MapReduceFusion

Performance



SDFG

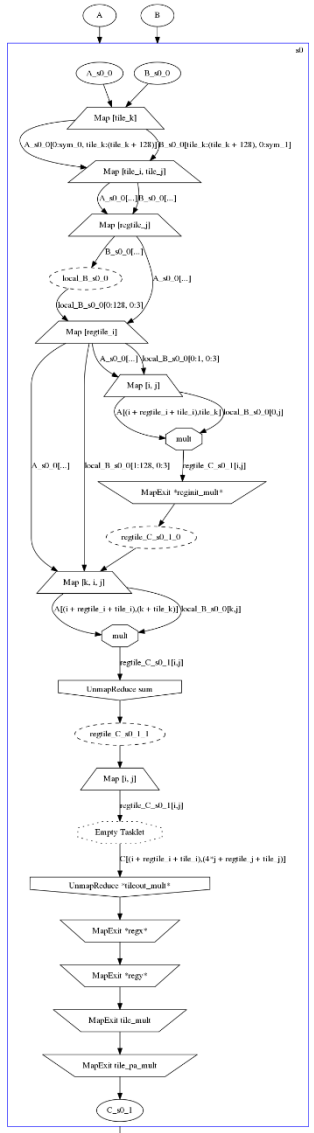


LocalStorage

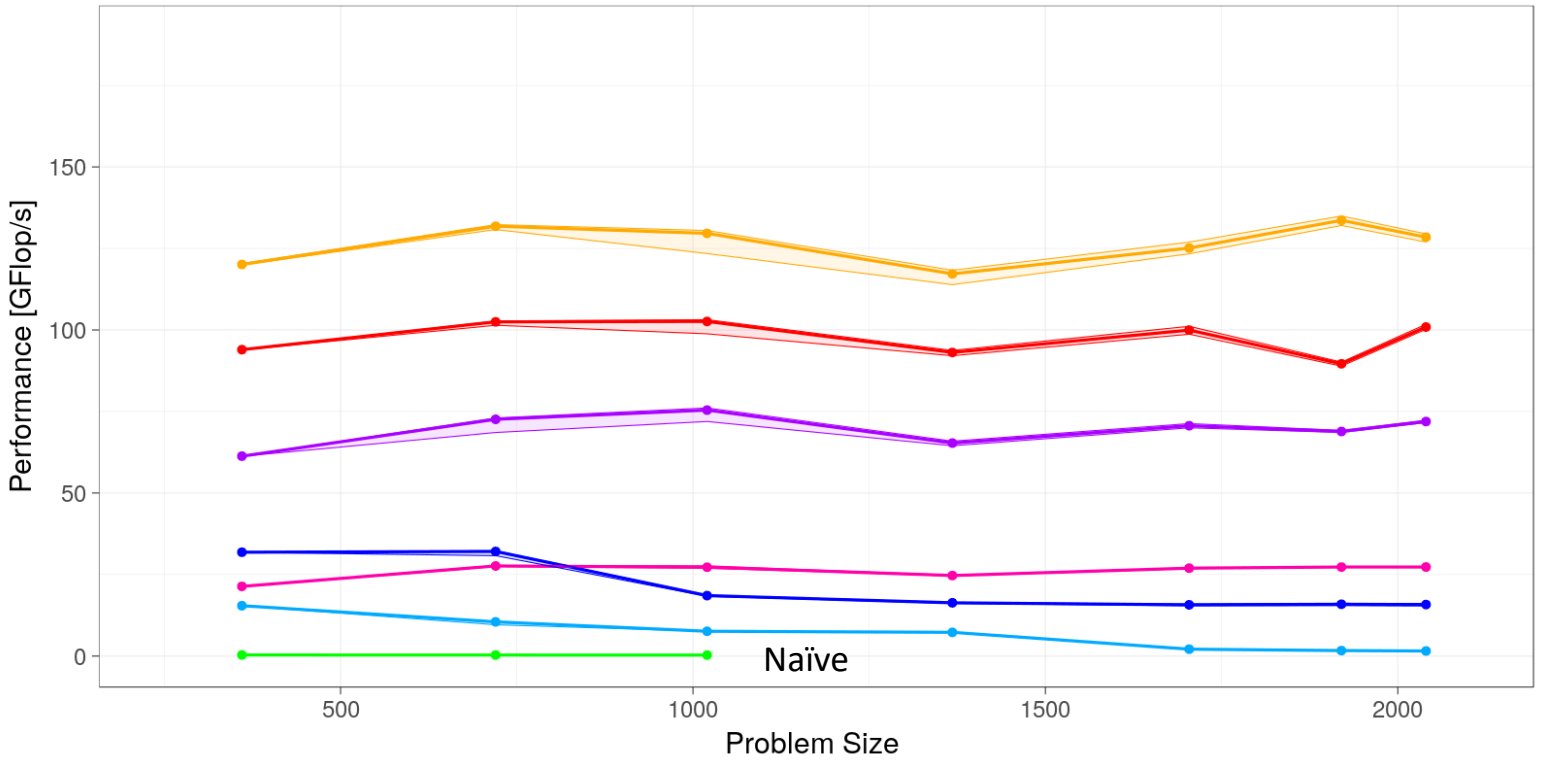
RegisterTiling

BlockTiling
LoopReorder
MapReduceFusion

Performance



SDFG



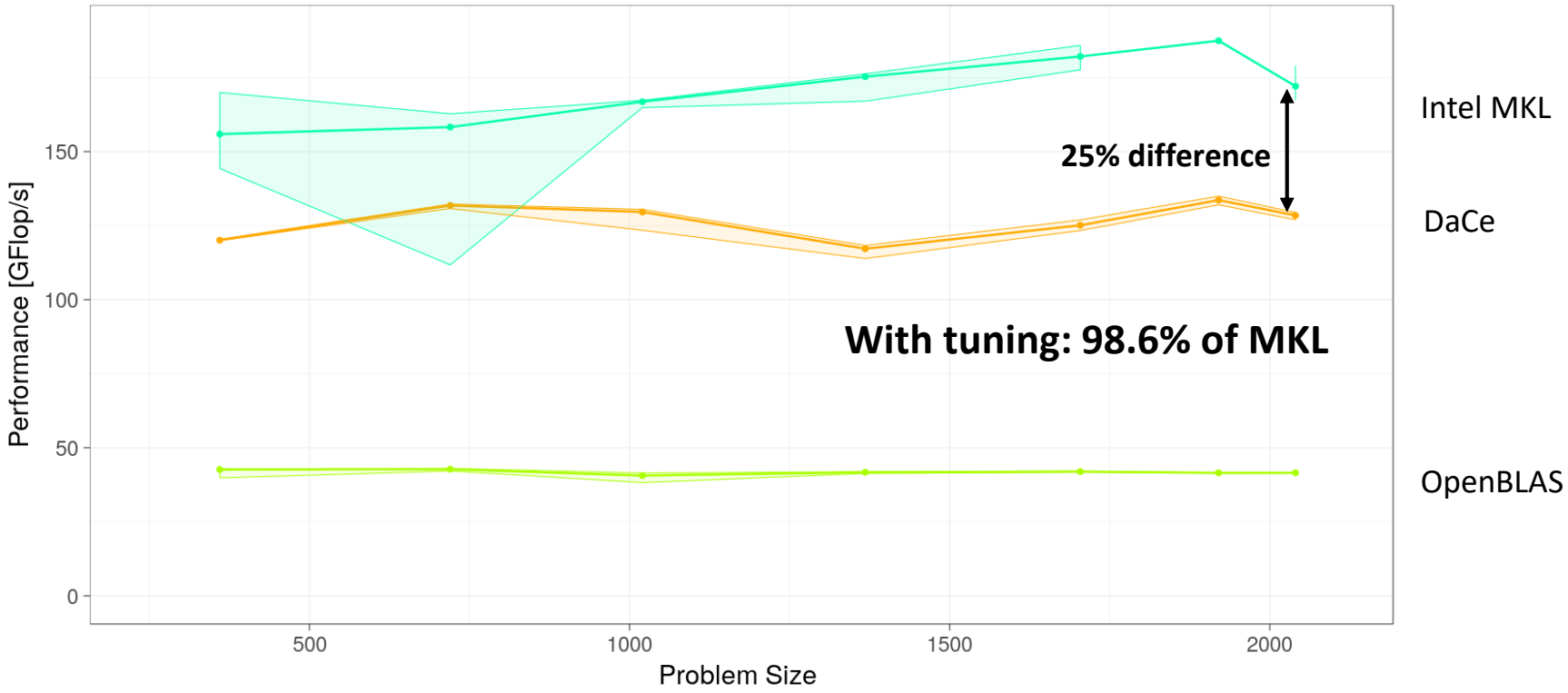
PromoteTransient

LocalStorage

RegisterTiling

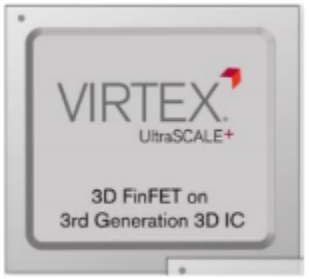
BlockTiling
LoopReorder
MapReduceFusion

Performance



Intel Xeon E5-2650 v4

NVIDIA Tesla P100



Xilinx VU9P

SDFG

General Compilers

GCC 8, Clang 6, icc 18,
NVCC 9.2, SDAccel

Polyhedral Optimizers

Polly 6, Pluto 0.11.4, PPCG 0.8

Frameworks & Libraries

HPX, Halide, Intel MKL, CUBLAS,
CUSPARSE, CUTLASS, CUB

Performance Evaluation: Fundamental Kernels (CPU)

Database Query: roughly 50% of a 67,108,864 column

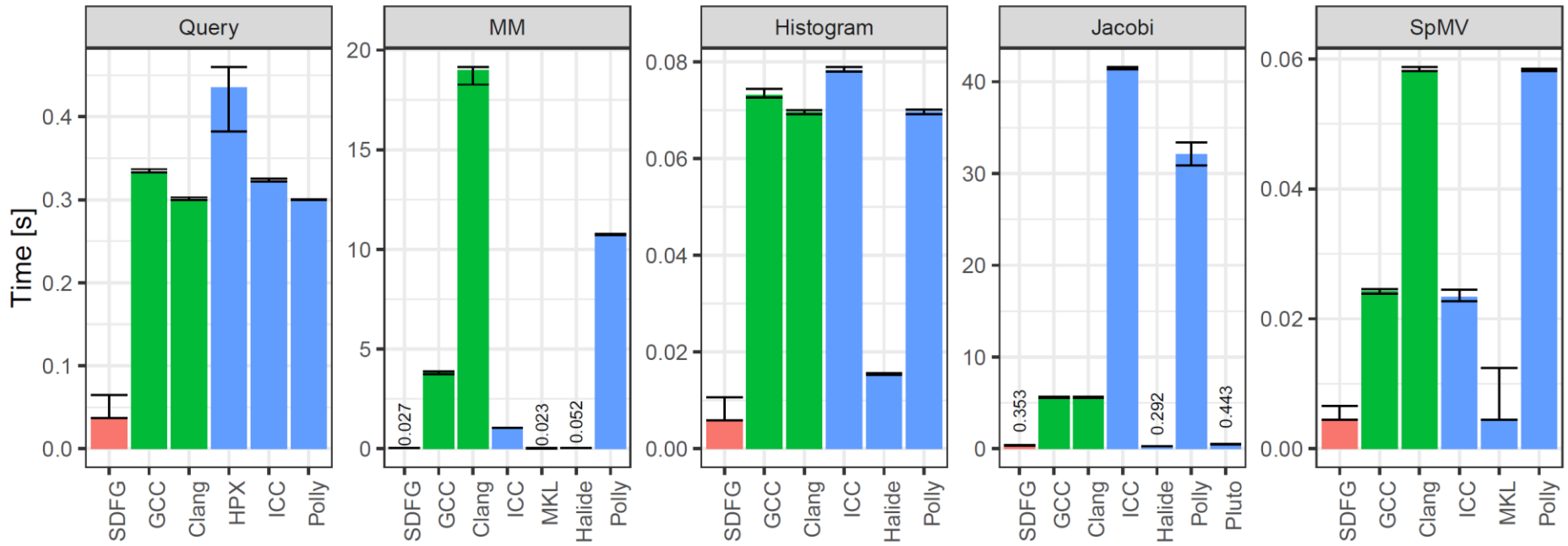
Matrix Multiplication (MM): 2048x2048x2048

Histogram: 8192x8192

Jacobi stencil: 2048x2048 for T=1024

Sparse Matrix-Vector Multiplication (SpMV): 8192x8192 CSR matrix (nnz=33,554,432)

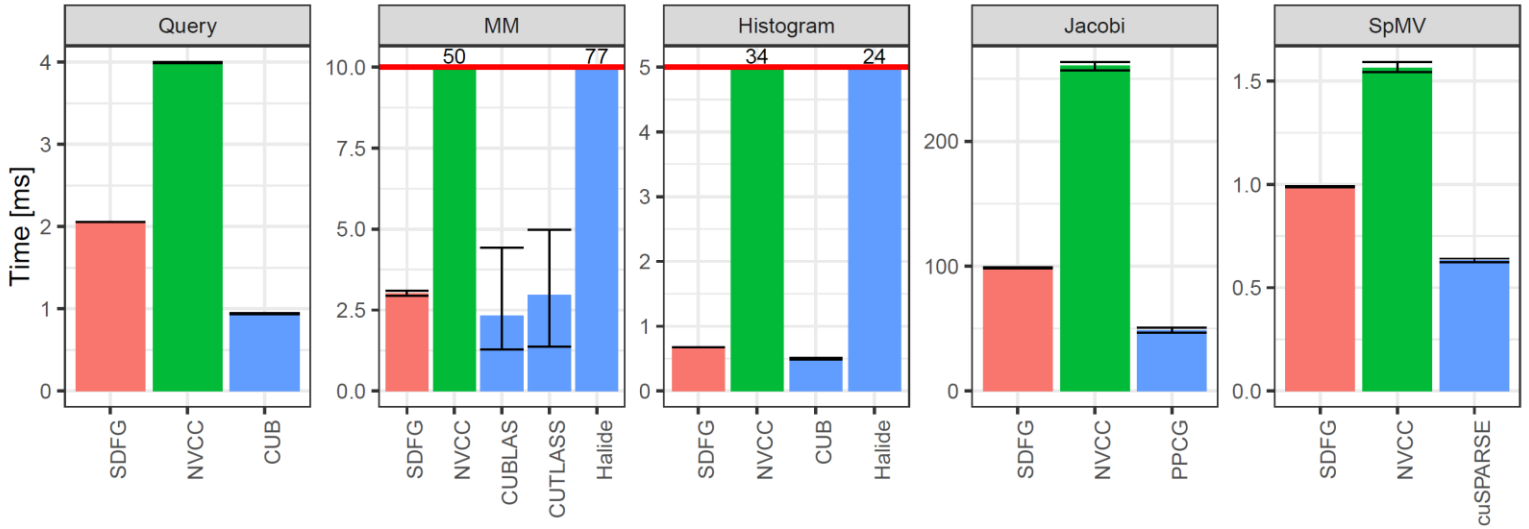
8.12x faster | 98.6% of MKL | 2.5x faster | 82.7% of Halide | 99.9% of MKL



Performance Evaluation: Fundamental Kernels (GPU, FPGA)

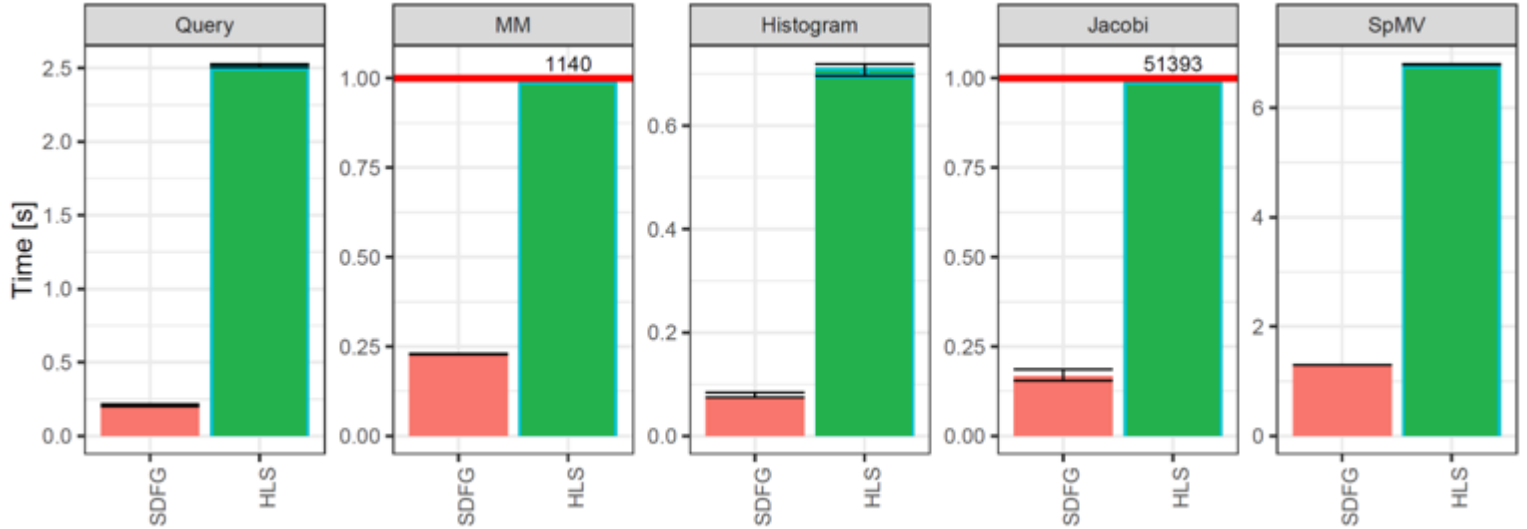
GPU

90% of CUTLASS



FPGA

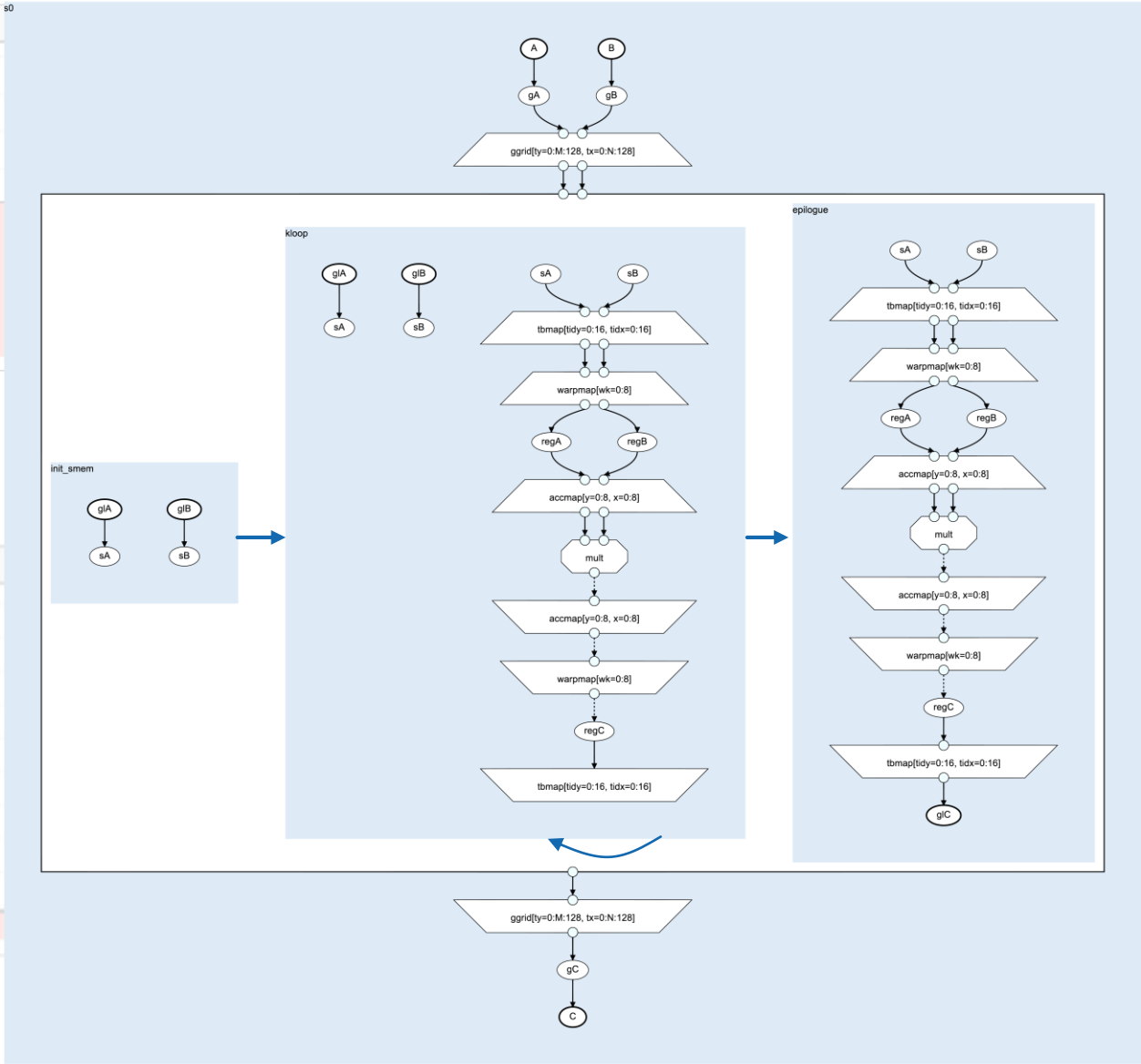
19.5x of Spatial



Performance Evaluation: Fundamental Kernels (GPU, FPGA)

GPU

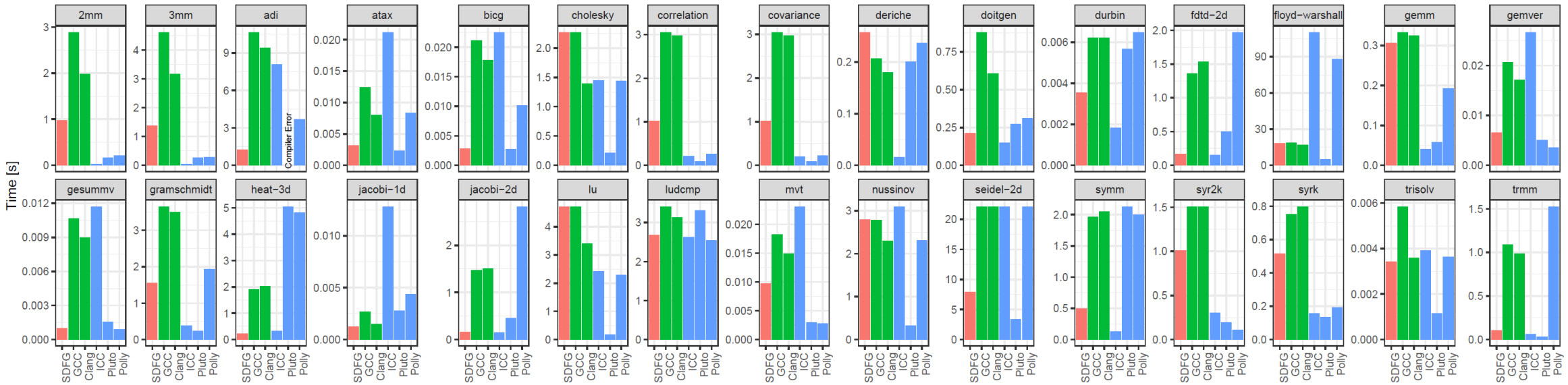
FPGA



Performance Evaluation: Polybench (CPU)

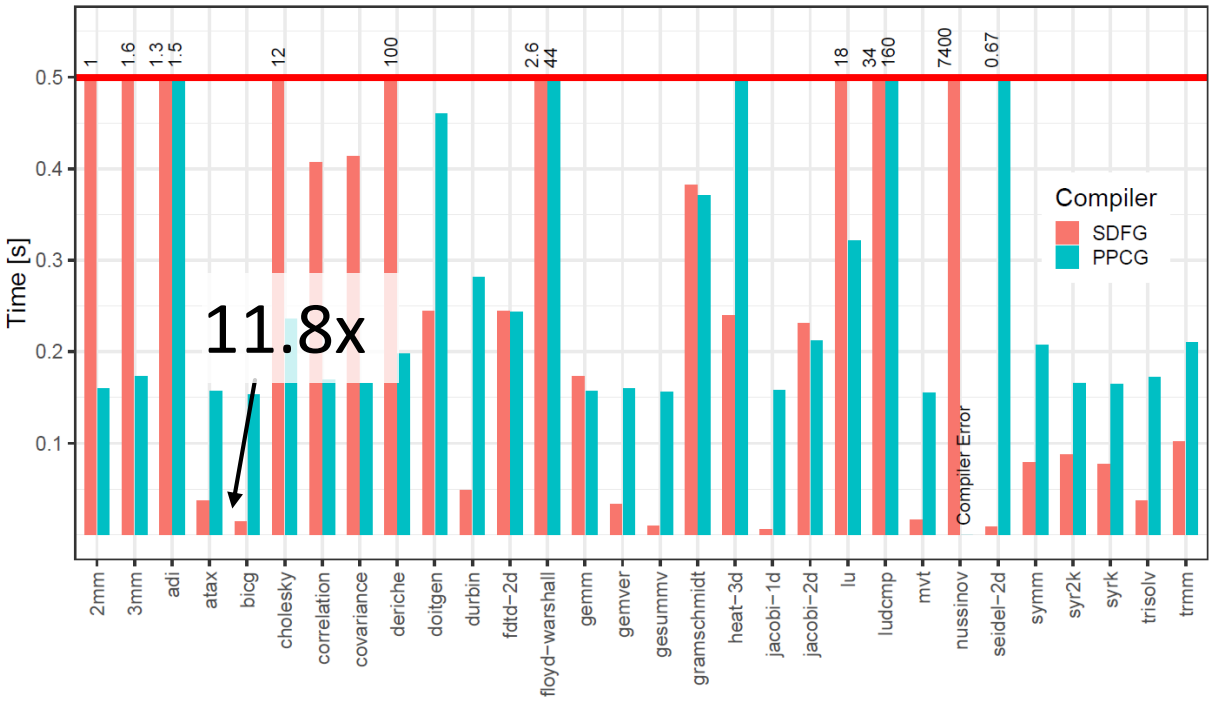
- Polyhedral benchmark with 30 applications

- Without any transformations, achieves 1.43x (geometric mean) over general-purpose compilers



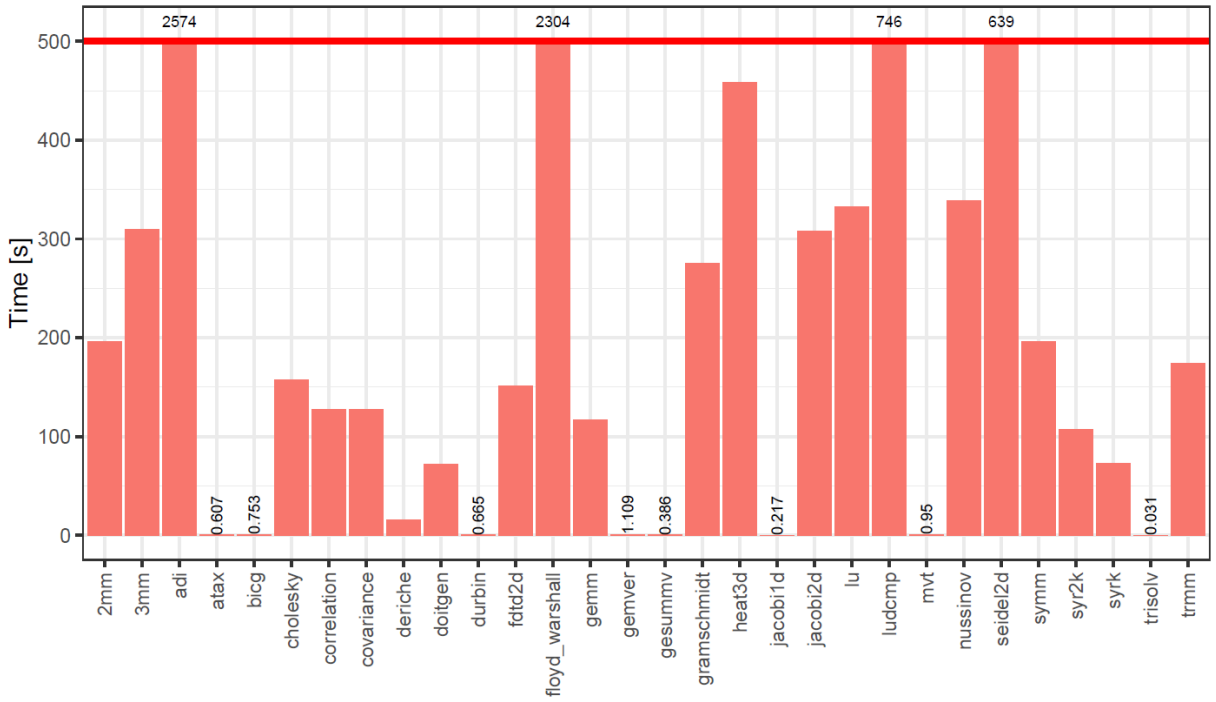
Performance Evaluation: Polybench (GPU, FPGA)

- **Automatically transformed from CPU code**



GPU

(1.12x geomean speedup)



FPGA

The **first** full set of placed-and-routed Polybench

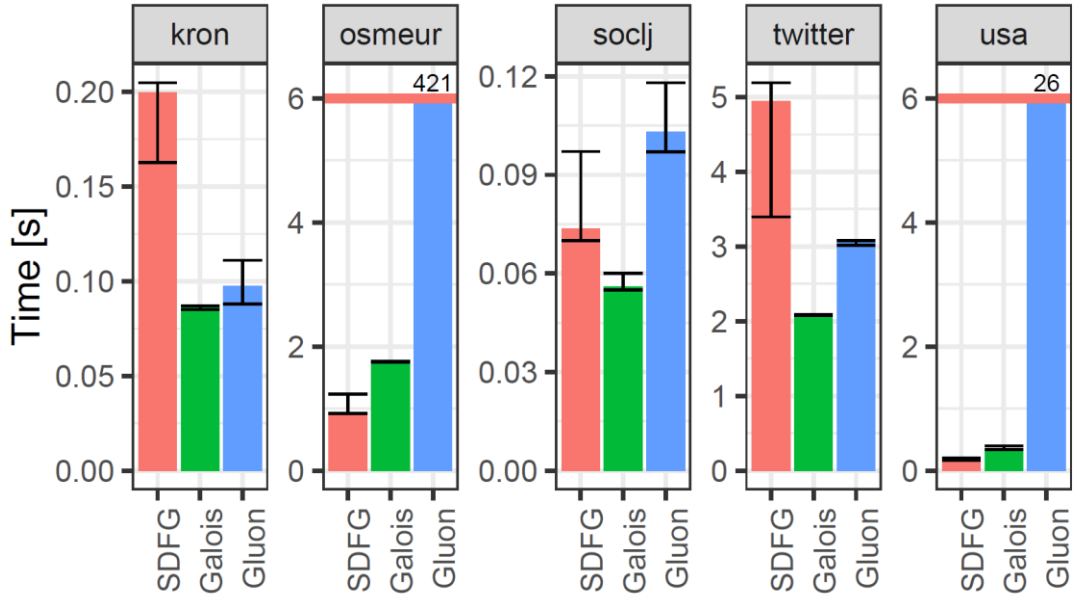
Case Study: Parallel Breadth-First Search

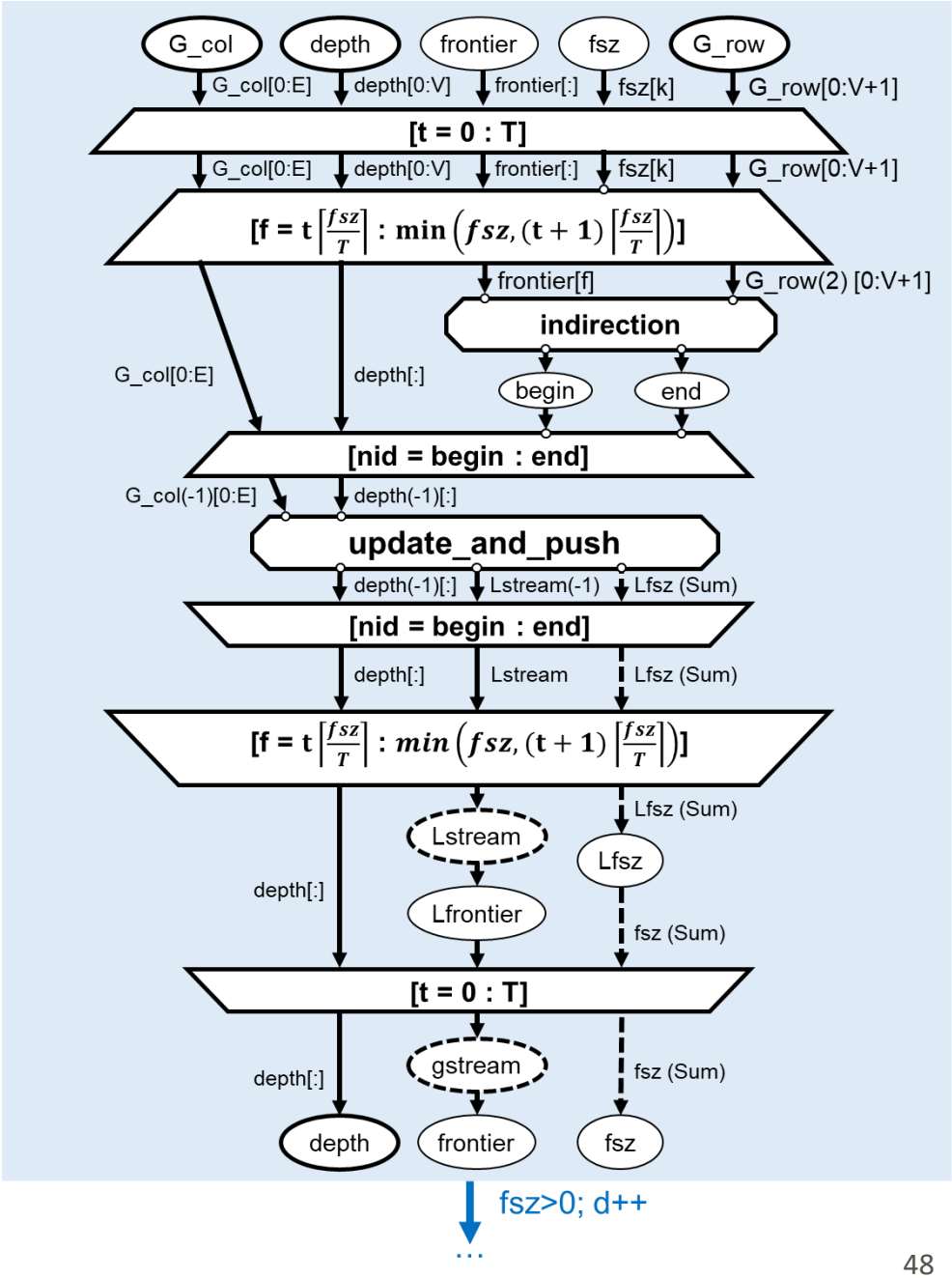
- **Compared with Galois and Gluon**
- **Graphs:**

  Road maps: USA, OSM-Europe

  Social networks: Twitter, LiveJournal

  Synthetic: Kronecker Graphs

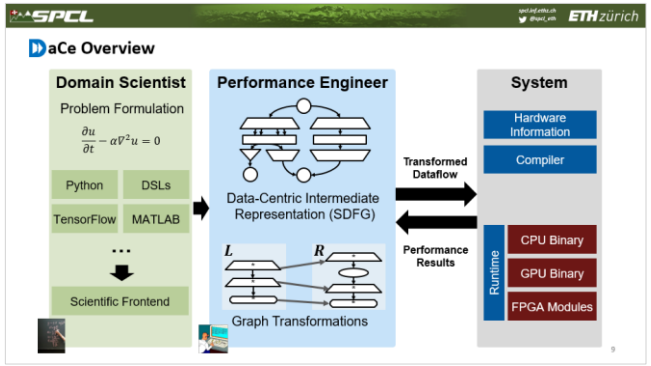


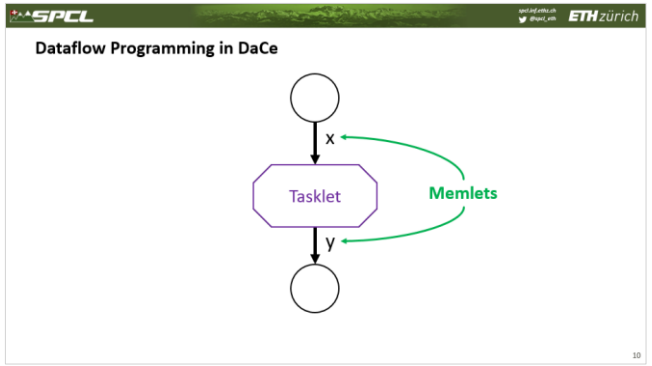


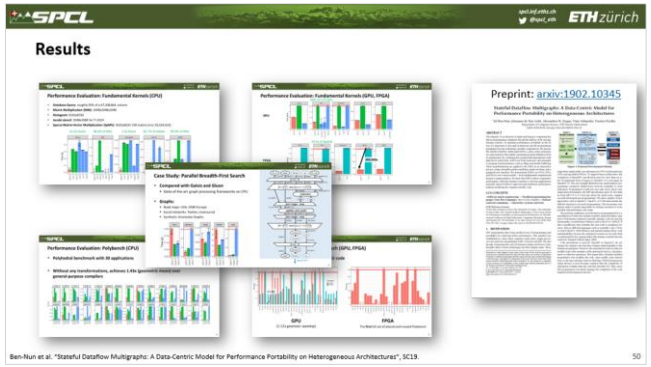
Conclusions

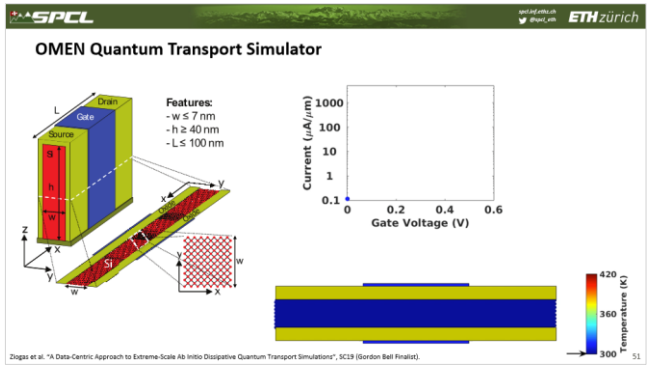
https://www.github.com/spcl/dace

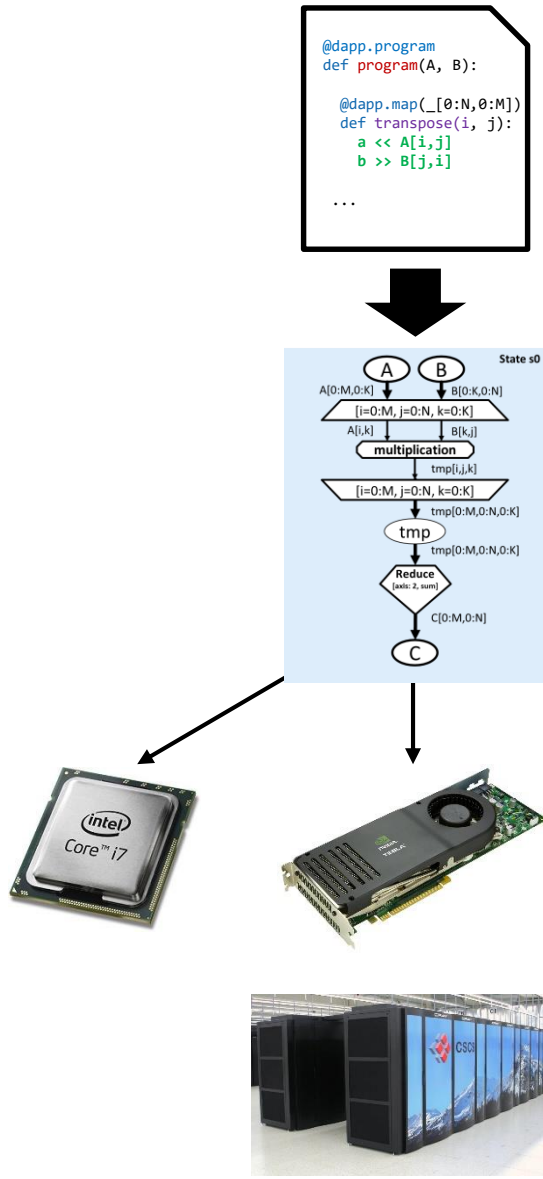
```
pip install dace
```











This project has received funding from the European Research Council (ERC) under grant agreement "DAPP (PI: T. Hoefler)".