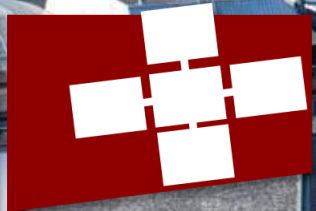


# Data-Centric Parallel Programming

Torsten Hoefler, Keynote at PARCO 2019, Prague, Czech Republic

Tal Ben-Nun, Alexandros Ziogas, Guillermo Indalecio, Timo Schneider, Mathieu Luisier, and Johannes de Fine Licht and the whole DAPP team @ SPCL

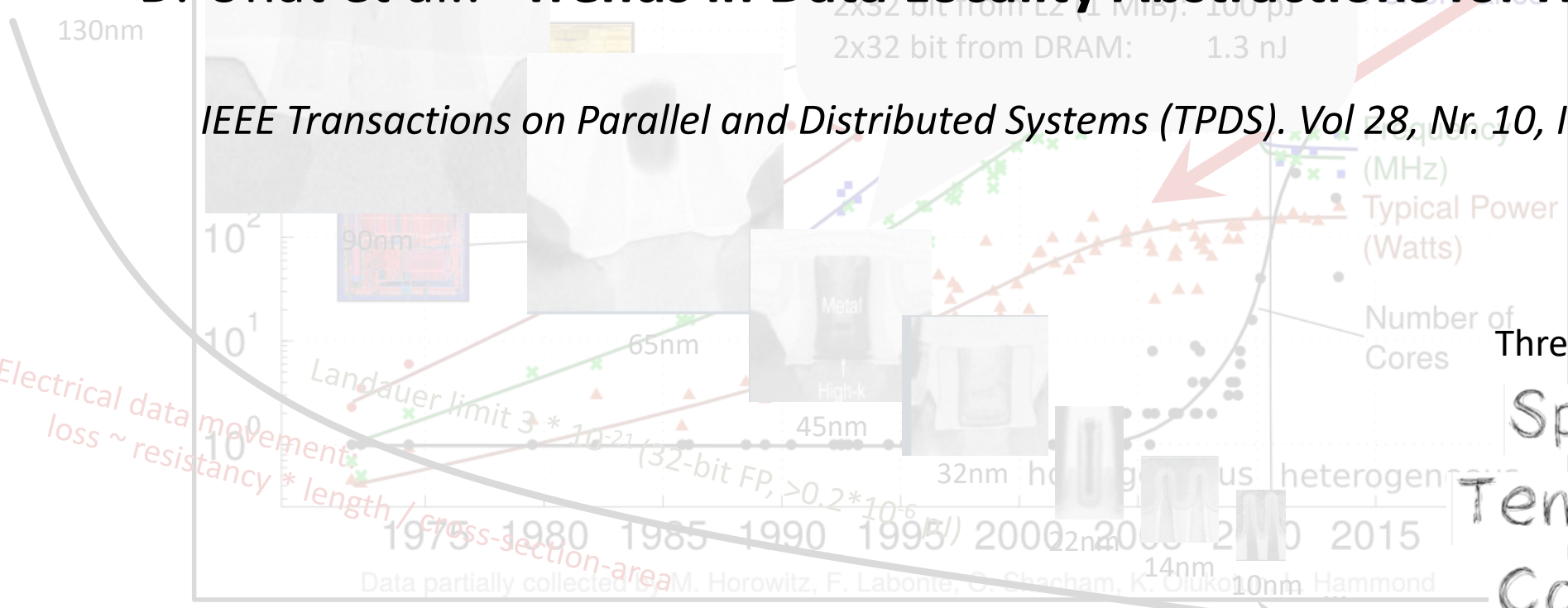




# Changing hardware constraints and the physics of computing

How to address locality challenges on standard architectures and programming?  
 D. Unat et al.: **“Trends in Data Locality Abstractions for HPC Systems”**

*IEEE Transactions on Parallel and Distributed Systems (TPDS). Vol 28, Nr. 10, IEEE, Oct. 2017*



Three Ls of modern computing:  
 Spatial Locality  
 Temporal Locality  
 Control Locality

[1]: Marc Horowitz, Computing's Energy Problem (and what we can do about it), ISSC 2014, plenary  
 [2]: Moore: Landauer Limit Demonstrated, IEEE Spectrum 2012

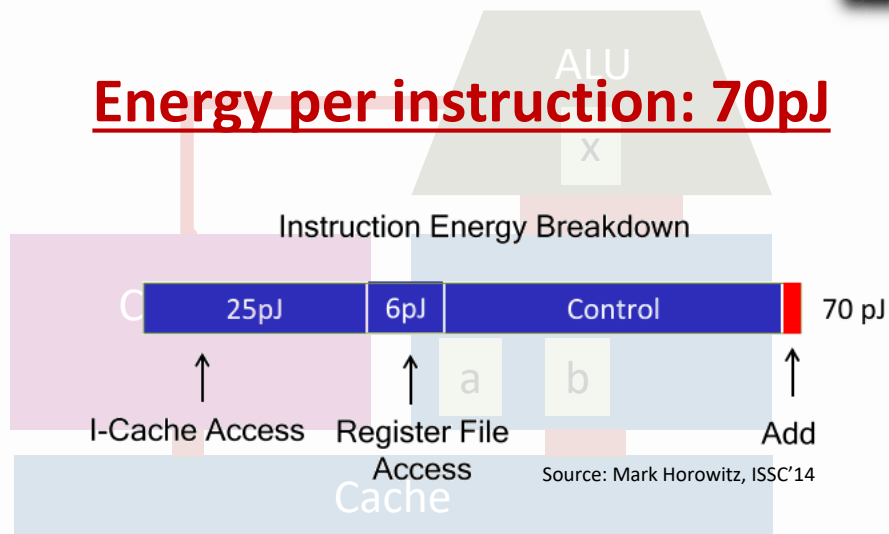
# Control in Load-store vs. Dataflow

Turing Award 1977 (Backus): "Surely there must be a less primitive way of making big changes in the store than pushing vast numbers of words back and forth through the von Neumann bottleneck."

## Load-store ("von Neumann")



$$x = a + b$$

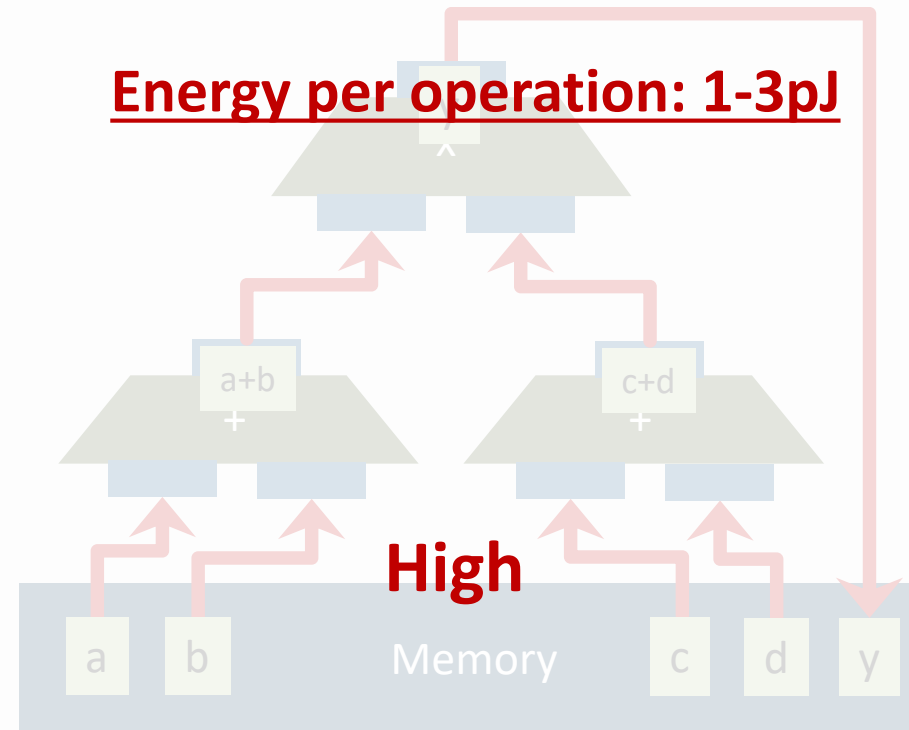


**Very Low**

## Static Dataflow ("non von Neumann")

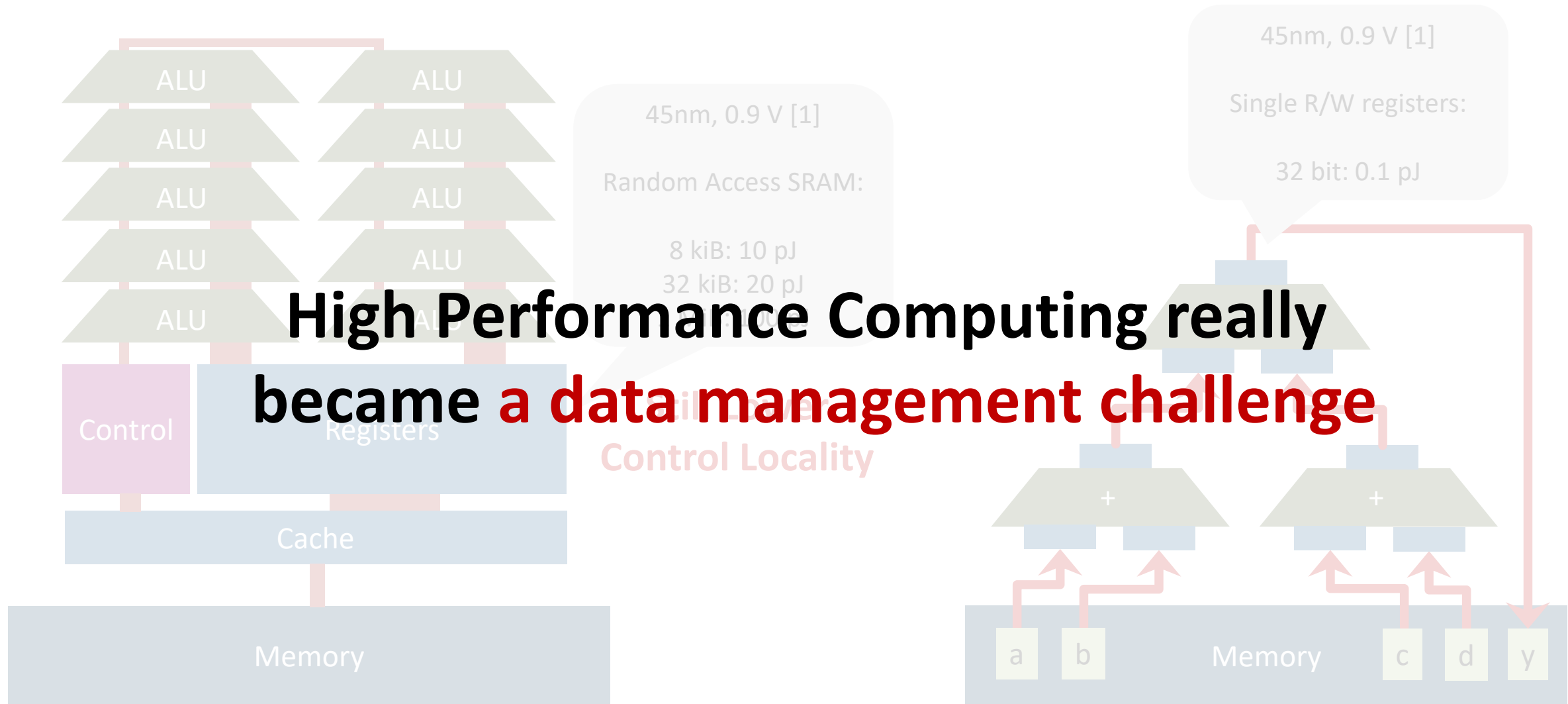


$$y = (a + b) * (c + d)$$



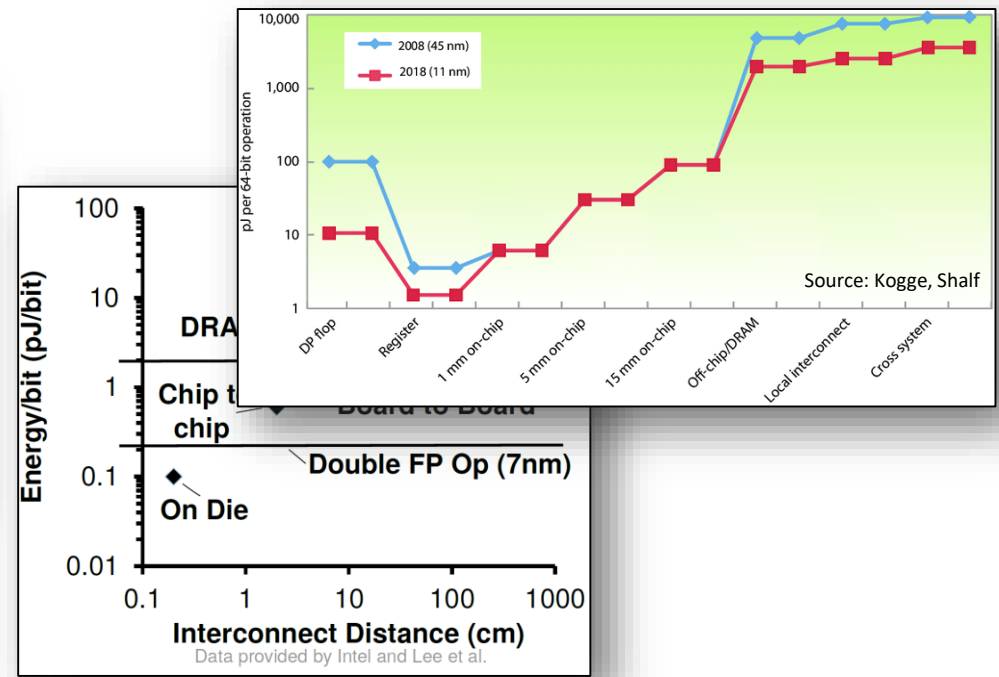
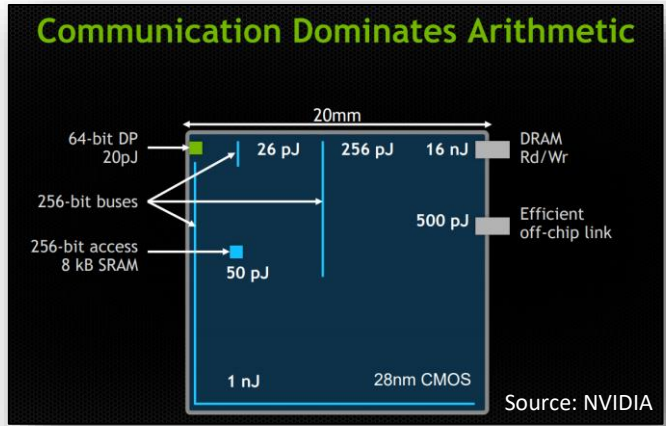
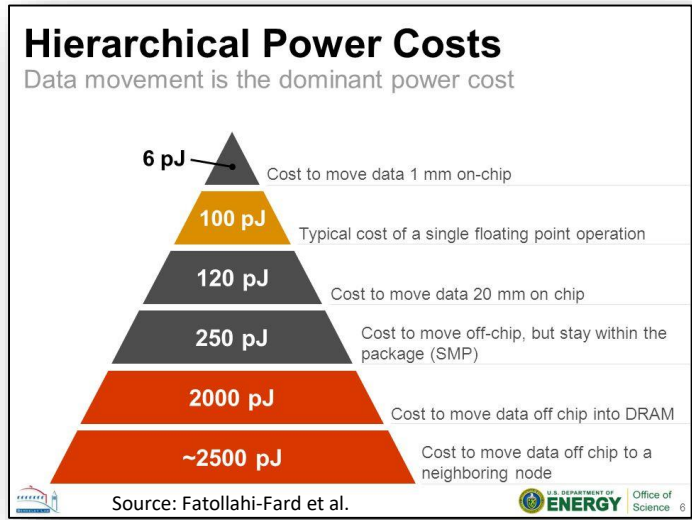
Control Locality

# Single Instruction Multiple Data/Threads (SIMD - Vector CPU, SIMT - GPU)



[1]: Marc Horowitz, Computing's Energy Problem (and what we can do about it), ISSC 2014, plenary

# Data movement will dominate everything!



DOI:10.1145/1941487.1941507

**Energy efficiency is the new fundamental limiter of processor performance, way beyond numbers of processors.**

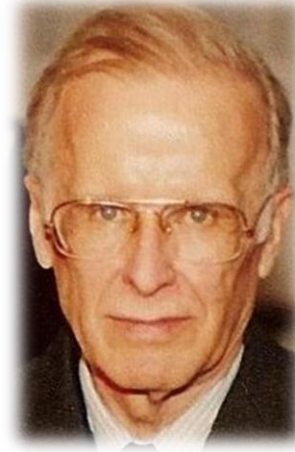
BY SHEKHAR BORKAR AND ANDREW A. CHIEN

# The Future of Microprocessors

- “In future microprocessors, the energy expended for data movement will have a critical effect on achievable performance.”
- “... movement consumes almost 58 watts with hardly any energy budget left for computation.”
- “...the cost of data movement starts to dominate.”
- “...data movement over these networks must be limited to conserve energy...”
- the phrase “data movement” appears 18 times on 11 pages (usually in concerning contexts)!
- “Efficient data orchestration will increasingly be critical, evolving to more efficient memory hierarchies and new types of interconnect tailored for locality and that **depend on sophisticated software to place computation and data so as to minimize data movement.**”

# “Sophisticated software”: How do we program today?

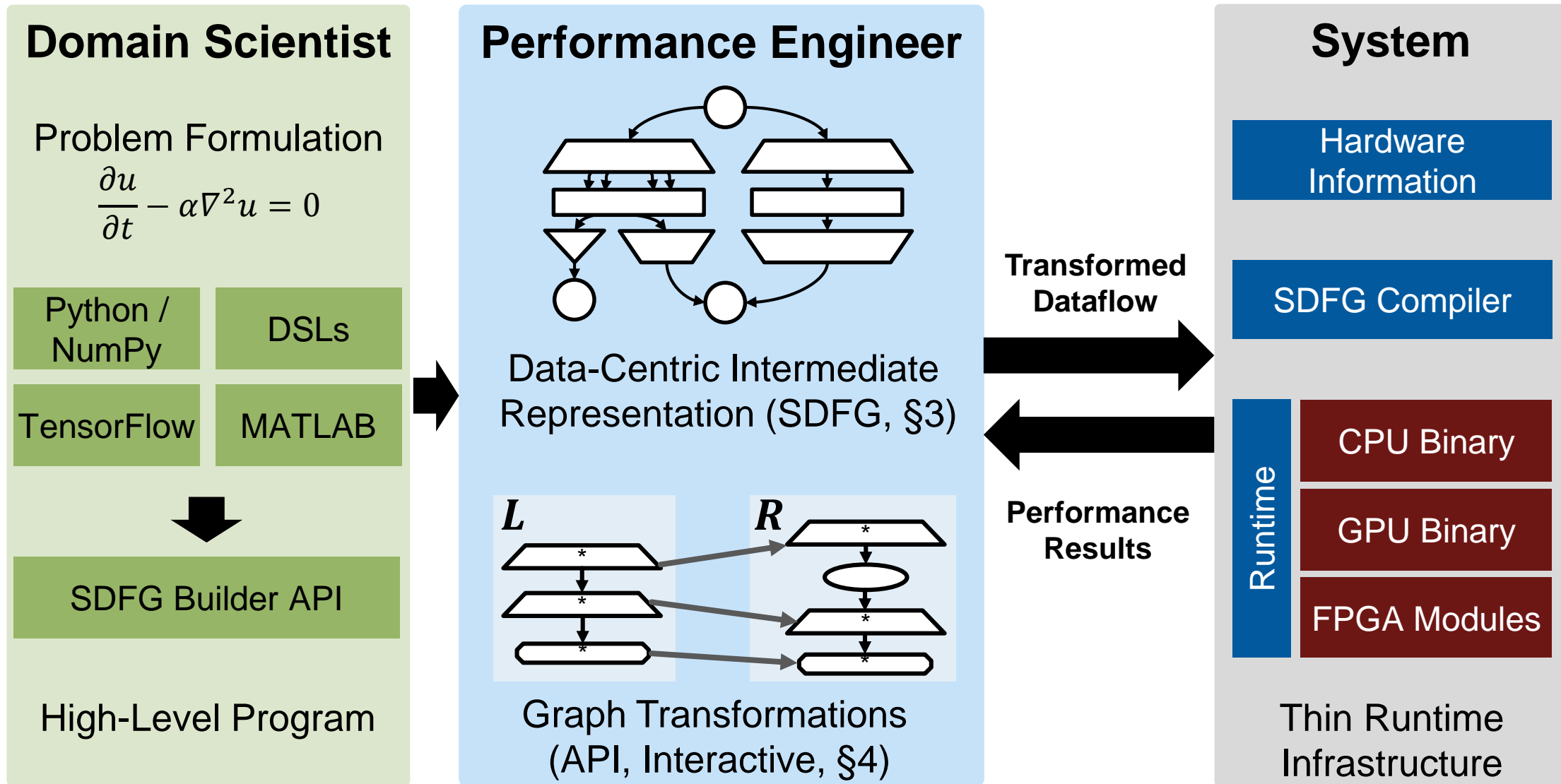
- **Well, to a good approximation how we programmed yesterday**
  - Or last year?
  - Or four decades ago?
- **Control-centric programming**
  - Worry about operation counts (flop/s is **the metric**, isn't it?)
  - Data movement is at best implicit (or invisible/ignored)
- **Legion [1] is taking a good direction towards data-centric**
  - Tasking relies on data placement but not really dependencies (not visible to tool-chain)
  - But it is still control-centric in the tasks – not (performance) portable between devices!
- **Let's go a step further towards an explicitly data-centric viewpoint**
  - For performance engineers at least!



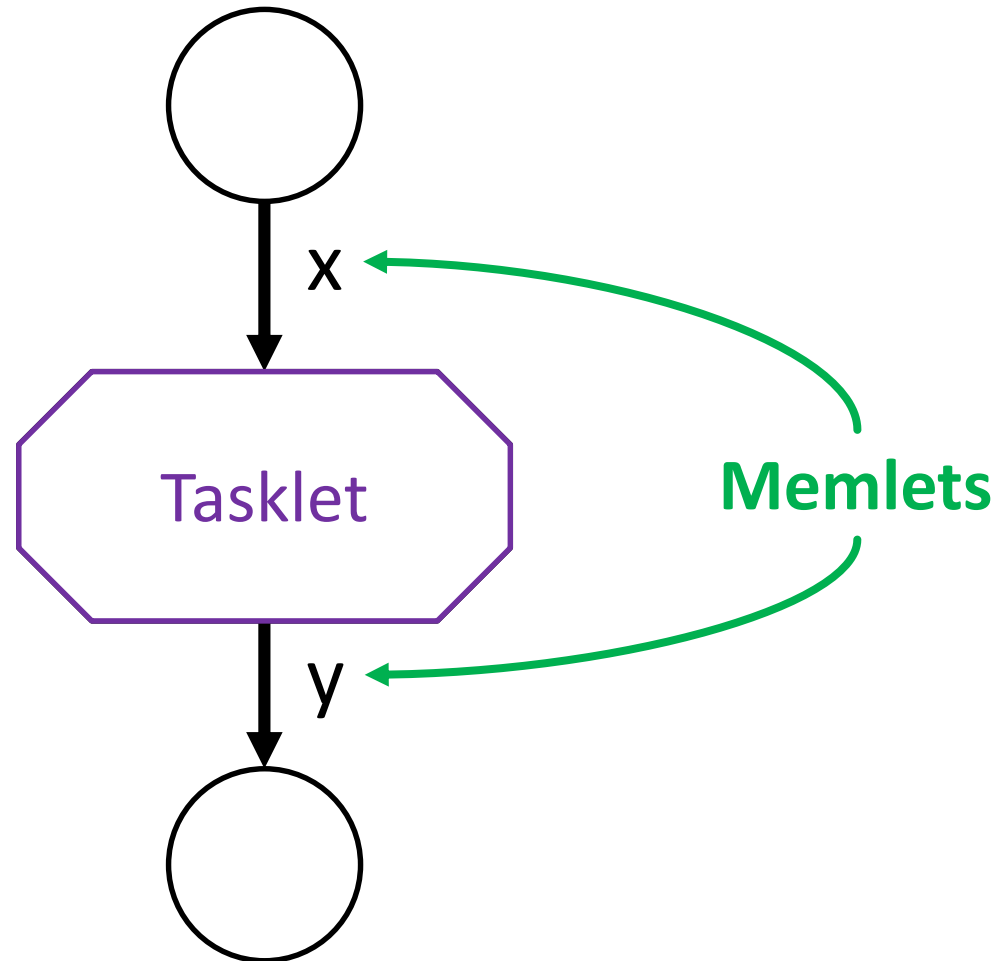
Backus '77: *“The assignment statement is the von Neumann bottleneck of programming languages and keeps us thinking in word-at-a-time terms in much the same way the computer's bottleneck does.”*



# Performance Portability with DataCentric (DaCe) Parallel Programming

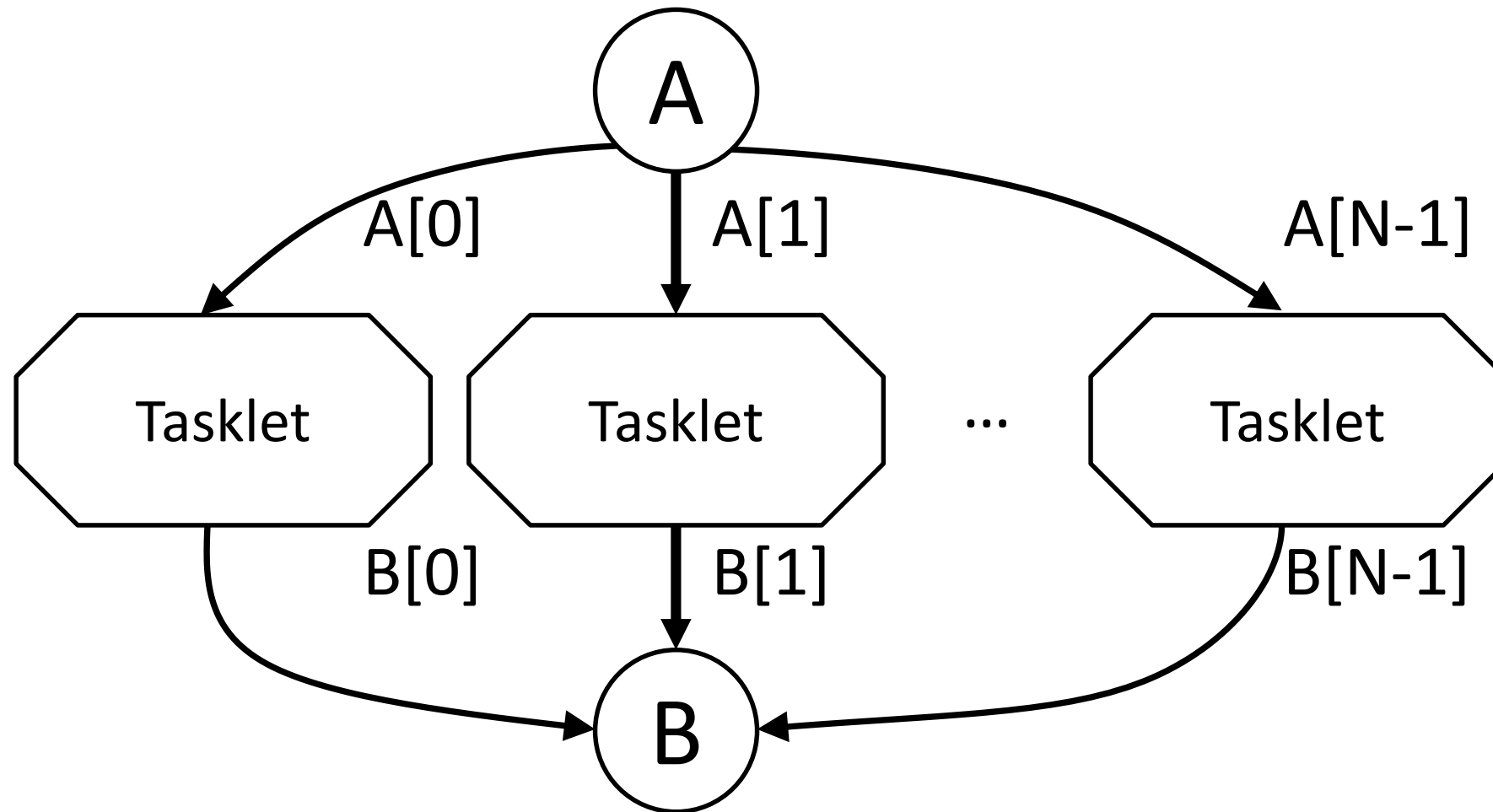


# Dataflow Programming

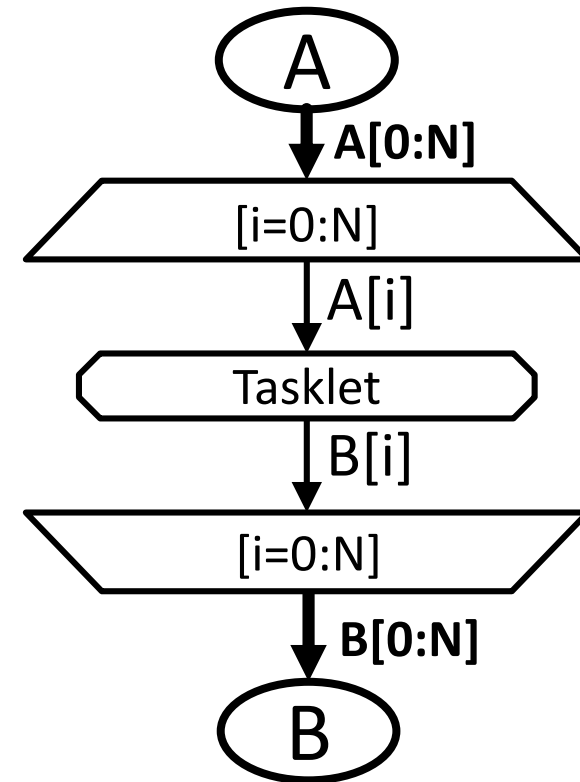
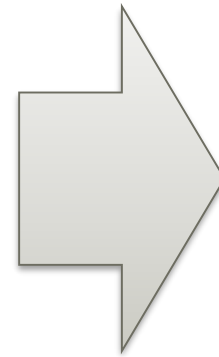
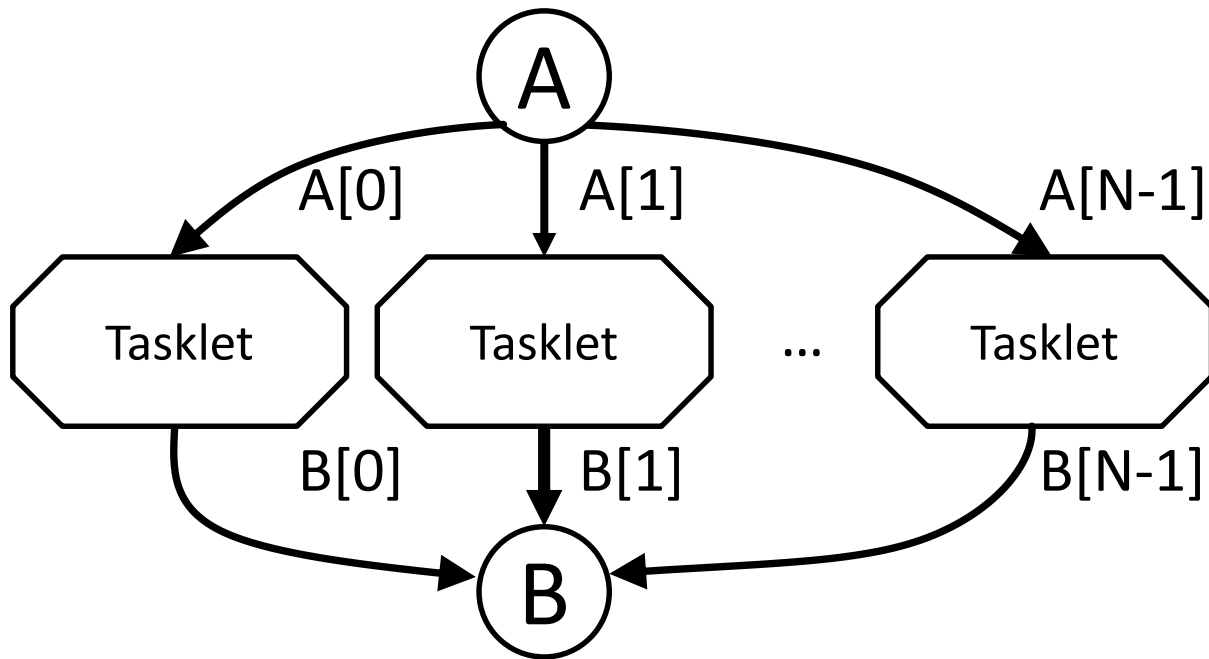




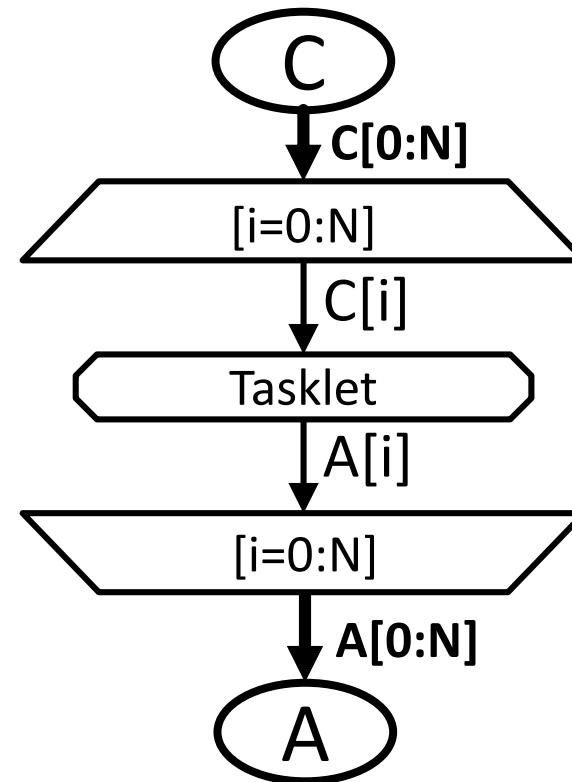
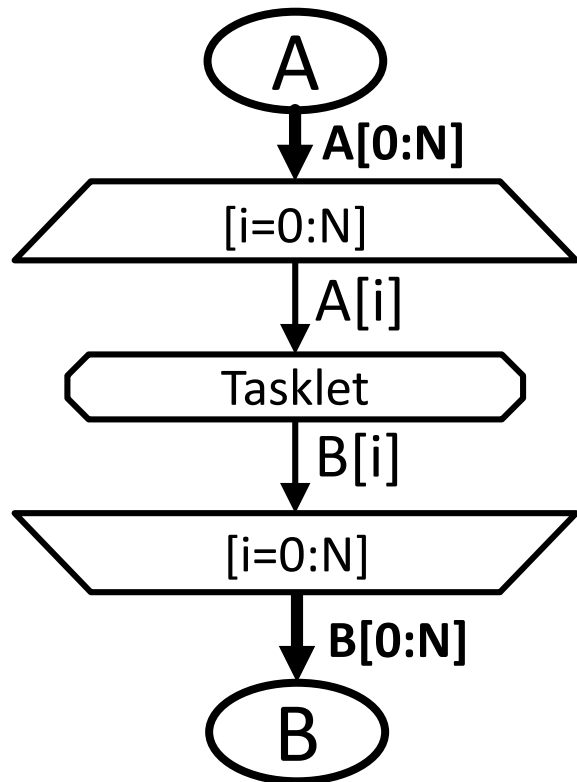
# Parallel Dataflow Programming



# Parallel Dataflow Programming

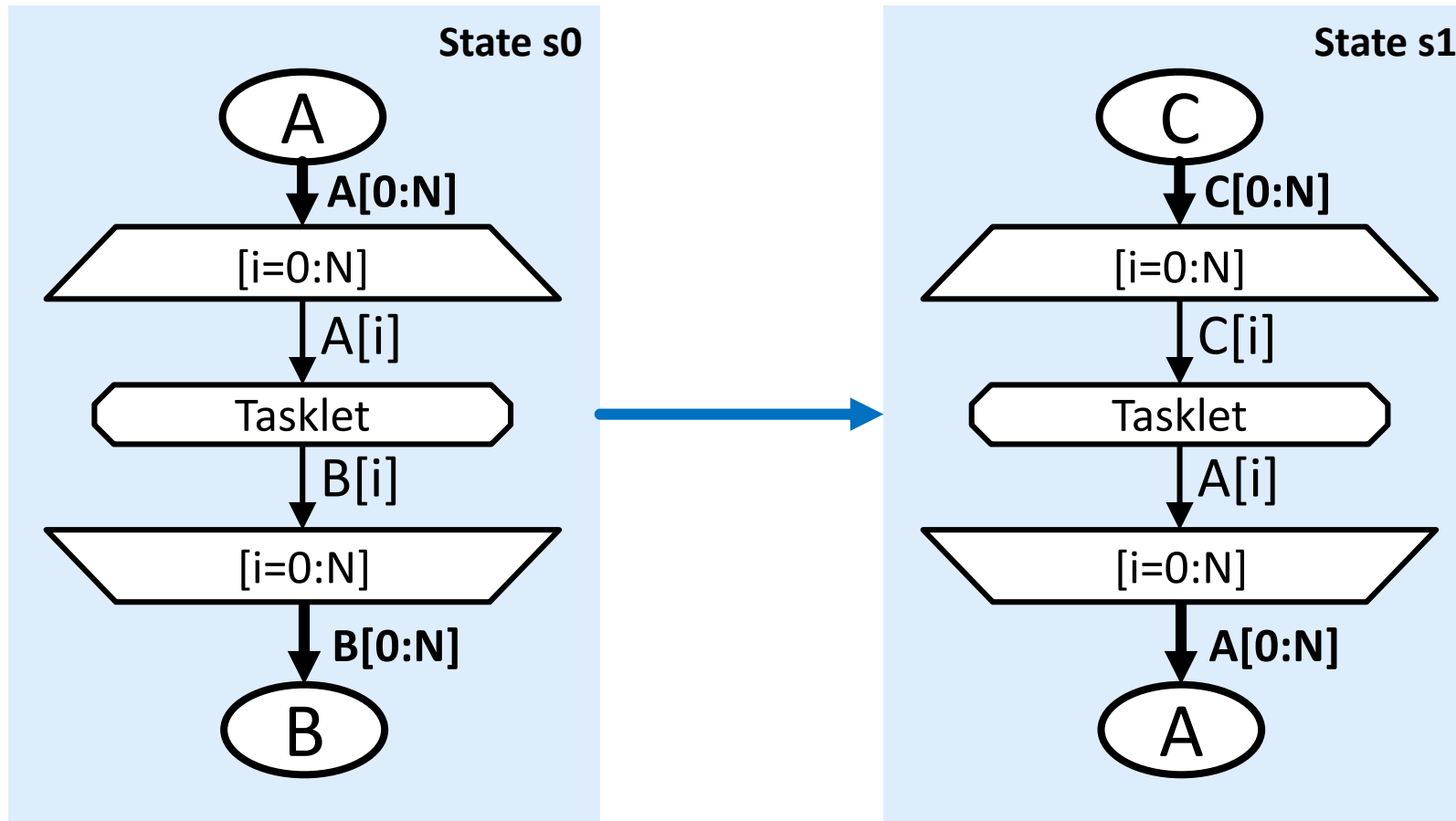


# Stateful Parallel Dataflow Programming

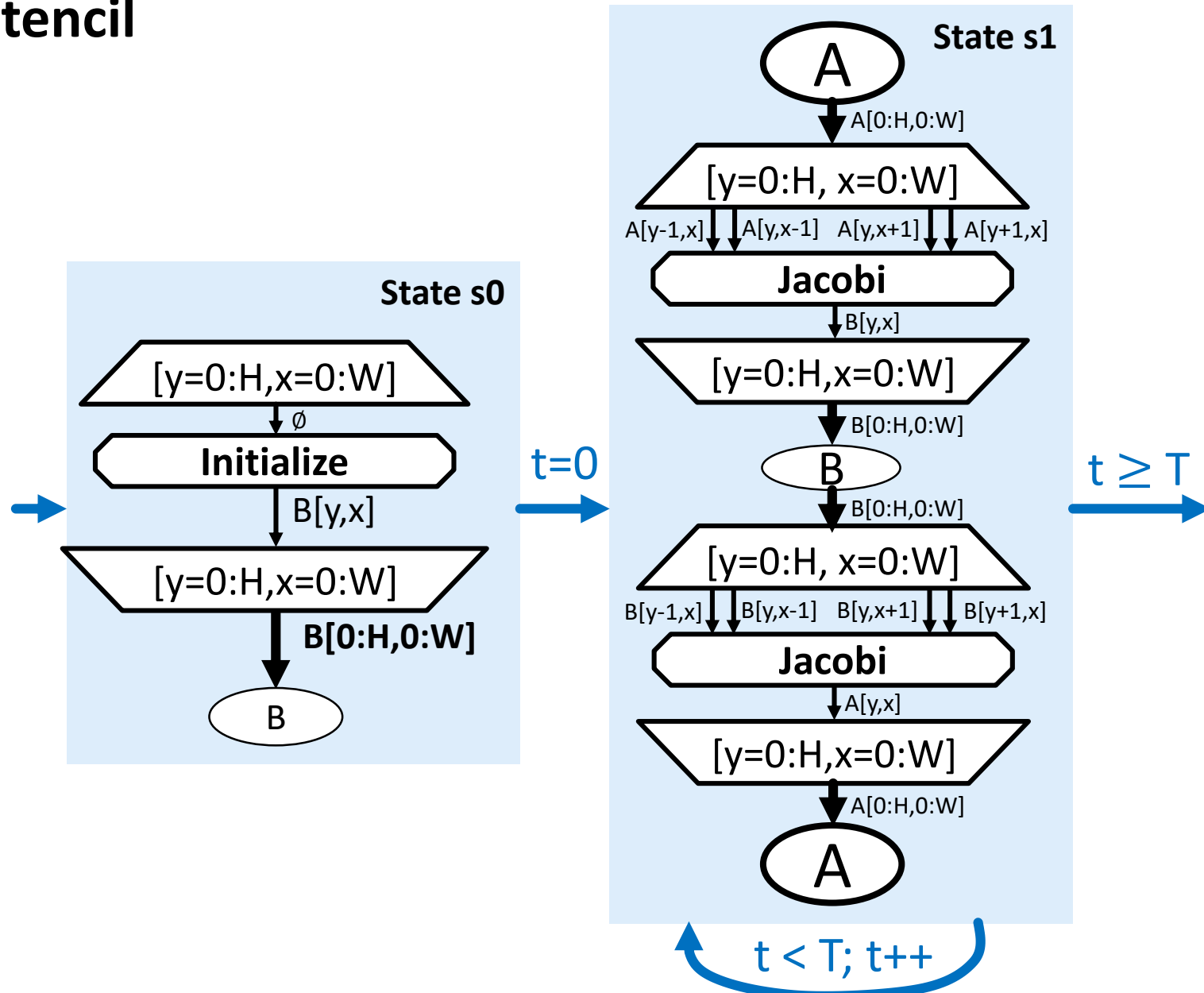




# Stateful Parallel Dataflow Programming



# Example: 2D Stencil



# Transformations

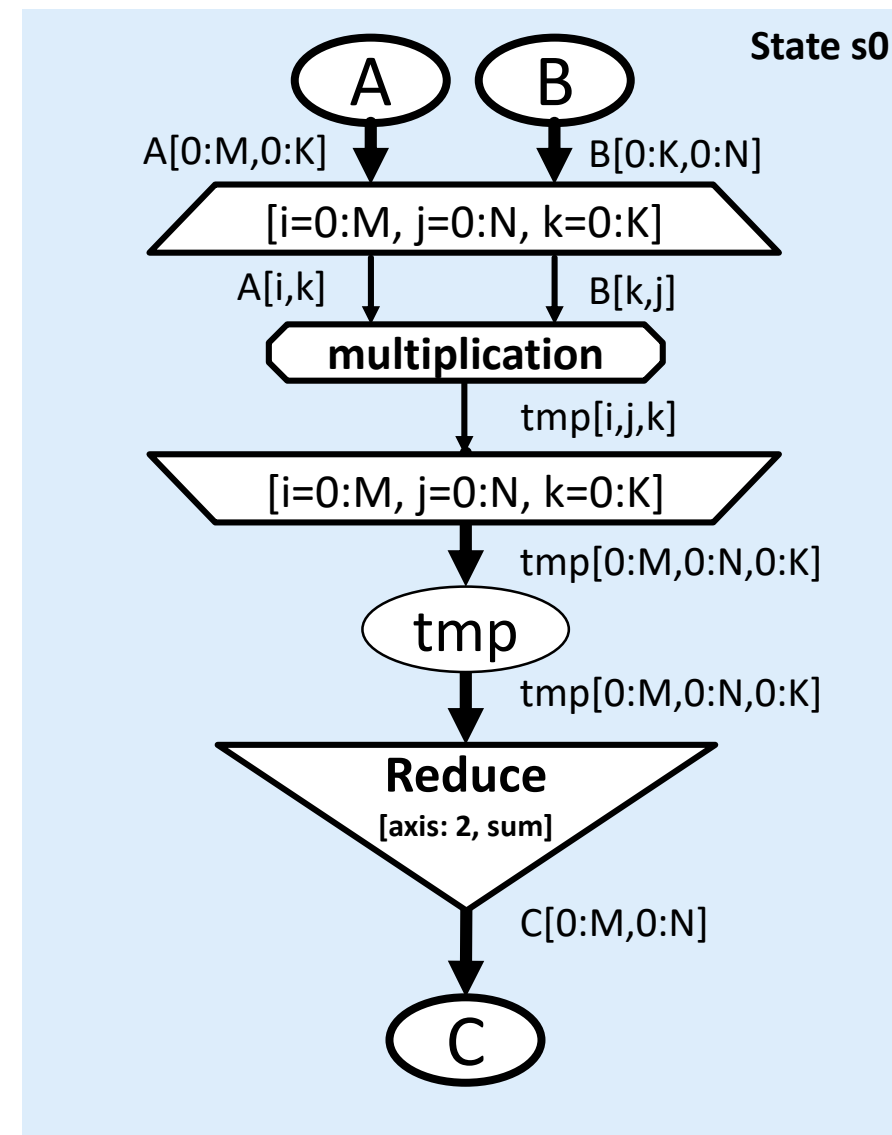
```

@dace.program
def gemm(A, B, C):
    # Transient variable
    tmp = dace.define_local([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)
    
```





# Transformations

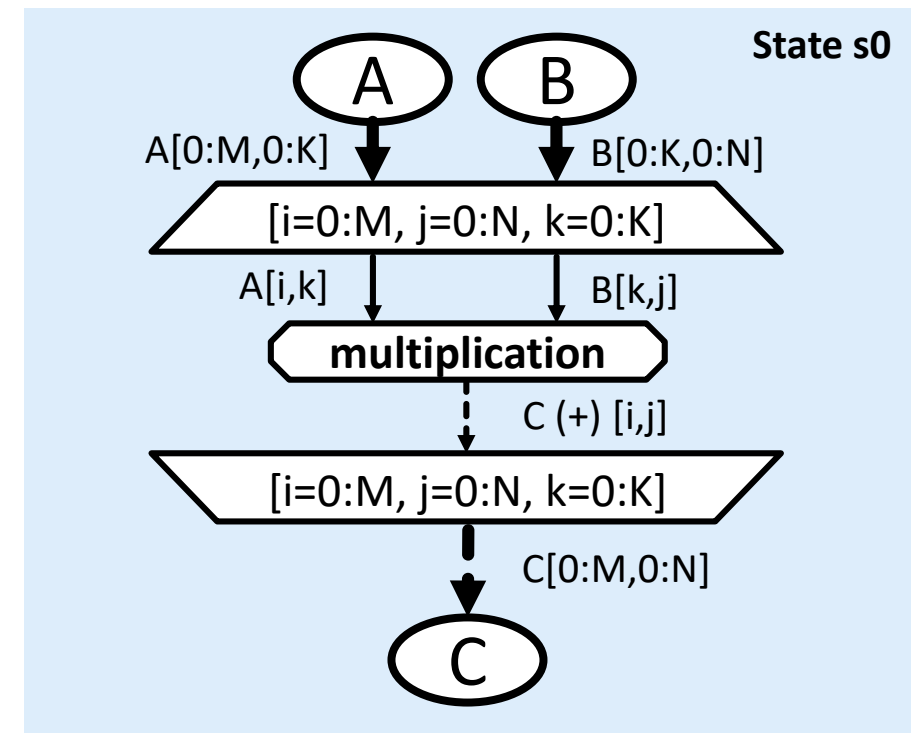
```

@dace.program
def gemm(A, B, C):
    # Transient variable
    tmp = dace.define_local([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

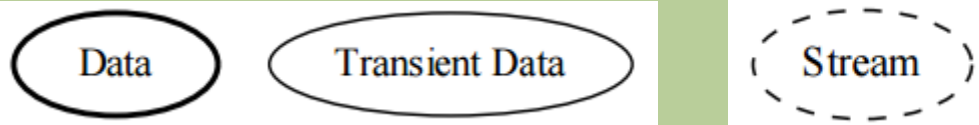
    dace.reduce(lambda a, b: a + b, tmp, C, axis=2)
    
```



# DAPP – Data Centric Programming Concepts

## Data Containers

- Store volatile (buffers, queues, RAM) and nonvolatile (files, I/O) information
- Can be sources or sinks of data



## Computation

- Stateless functions that perform computations at any granularity
- Data access only through ports



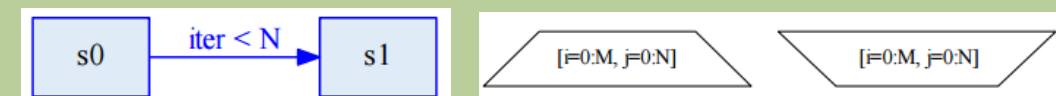
## Data Movement / Dependencies

- Data flowing between containers and tasklets/ports
- Implemented as access, copies, streaming, ...



## Parallelism and States

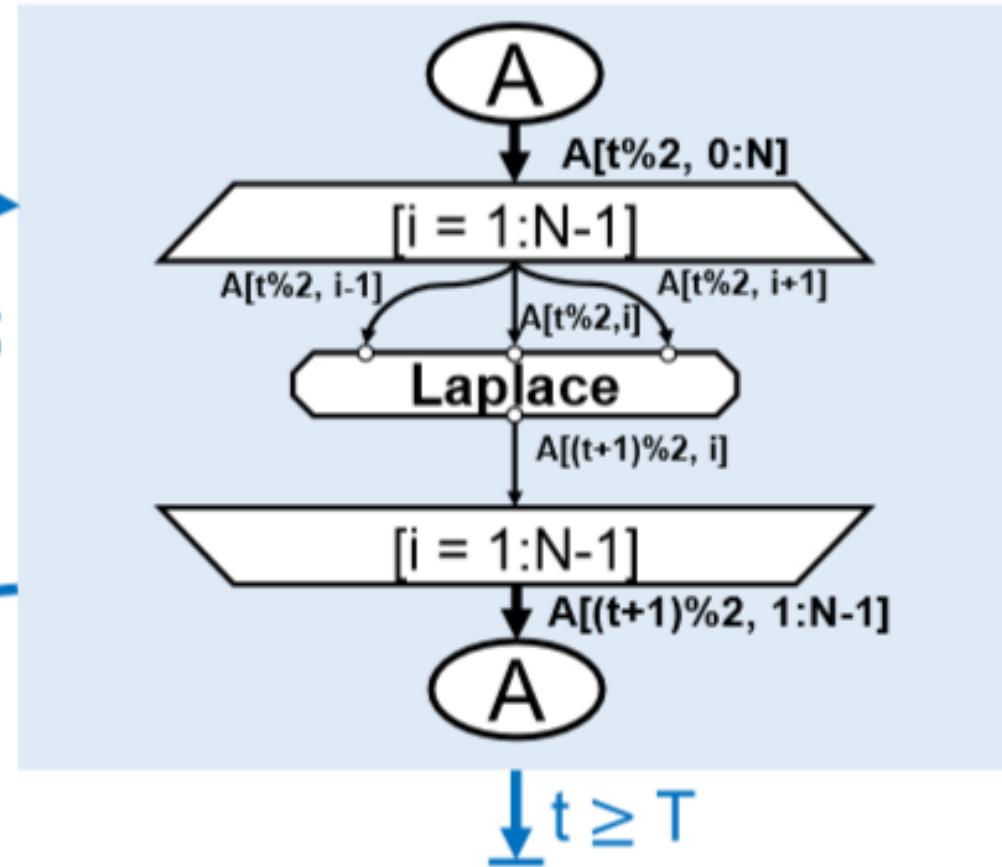
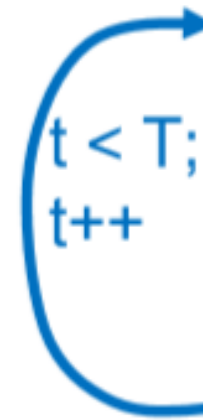
- Map scopes provide parallelism
- States constrain parallelism outside of dataflow



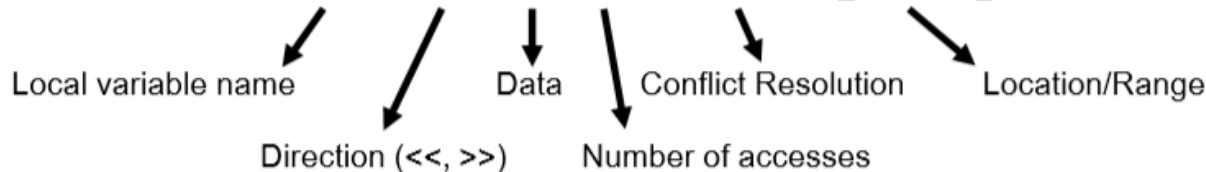
# A first example in DaCe Python

```

@dace.program
def Laplace(A: dace.float64[2,N],
            T: dace.uint32):
    for t in range(T):
        for i in dace.map[1:N-1]:
            # Data dependencies
            in_l << A[t%2, i-1]
            in_c << A[t%2, i]
            in_r << A[t%2, i+1]
            out >> A[(t+1)%2, i]
            # Computation
            out = in_l - 2*in_c + in_r
    
```



var << A(1, WCR)[0:N]





# DIODE User Interface

DIODE: Data-centric Integrated Optimization Development Environment

File Edit View Help

Optimizer Transformation Editor

```

15 @dapp.program(dapp.float64[M,N], dapp.float64[N,K], da
16 def gemm(A, B, C):
17     # Transient variable
18     tmp = dapp.define_local([M, K, N], dtype=A.dtype)
19
20     @dapp.map(_[0:M, 0:K, 0:N])
21     def multiplication(i, j, k):
22         # ...
23         out >> tmp[i,j,k]
24
25     out = in_A * in_B
26
27     @dapp.reduce(tmp, C, axis=2, identity=0)
28     def sum(a,b):
29         return a+b
30
                
```

## Source Code

## Transformations

## SDFG (malleable)

```

12 #pragma omp parallel for
13 for (auto i = 0; i < M; i += 1) {
14     for (auto j = 0; j < K; j += 1) {
15         for (auto k = 0; k < N; k += 1) {
16             {
17                 auto __in_A = dapp::ArrayView<double, 0, 1, 1> (A + i*N + k);
18                 dapp::vec<double, 1> in_A = __in_A;
19                 auto __in_B = dapp::ArrayView<double, 0, 1, 1> (B + k*K + j);
20                 dapp::vec<double, 1> in_B = __in_B;
21
22                 auto __out = dapp::ArrayView<double, 0, 1, 1> (C + i*K + j);
23                 dapp::vec<double, 1> out;
24
25                 // Tasklet code (multiplication)
26                 out = (in_A * in_B);
27
                
```

## Generated Code

Variant [order of exec.]	Performance [s]
0	0.095
1	0.015
2	0.015
3	0.010
4	0.010

## Performance

Properties:

async  OFF

name multiplication

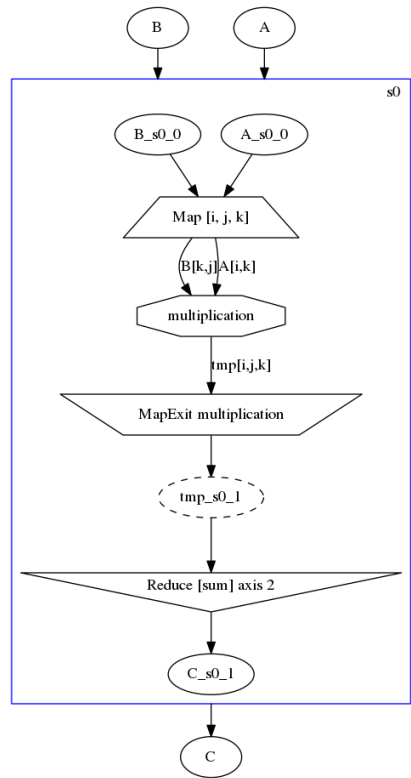
order (i, j, k)

schedule ScheduleType.Multicore

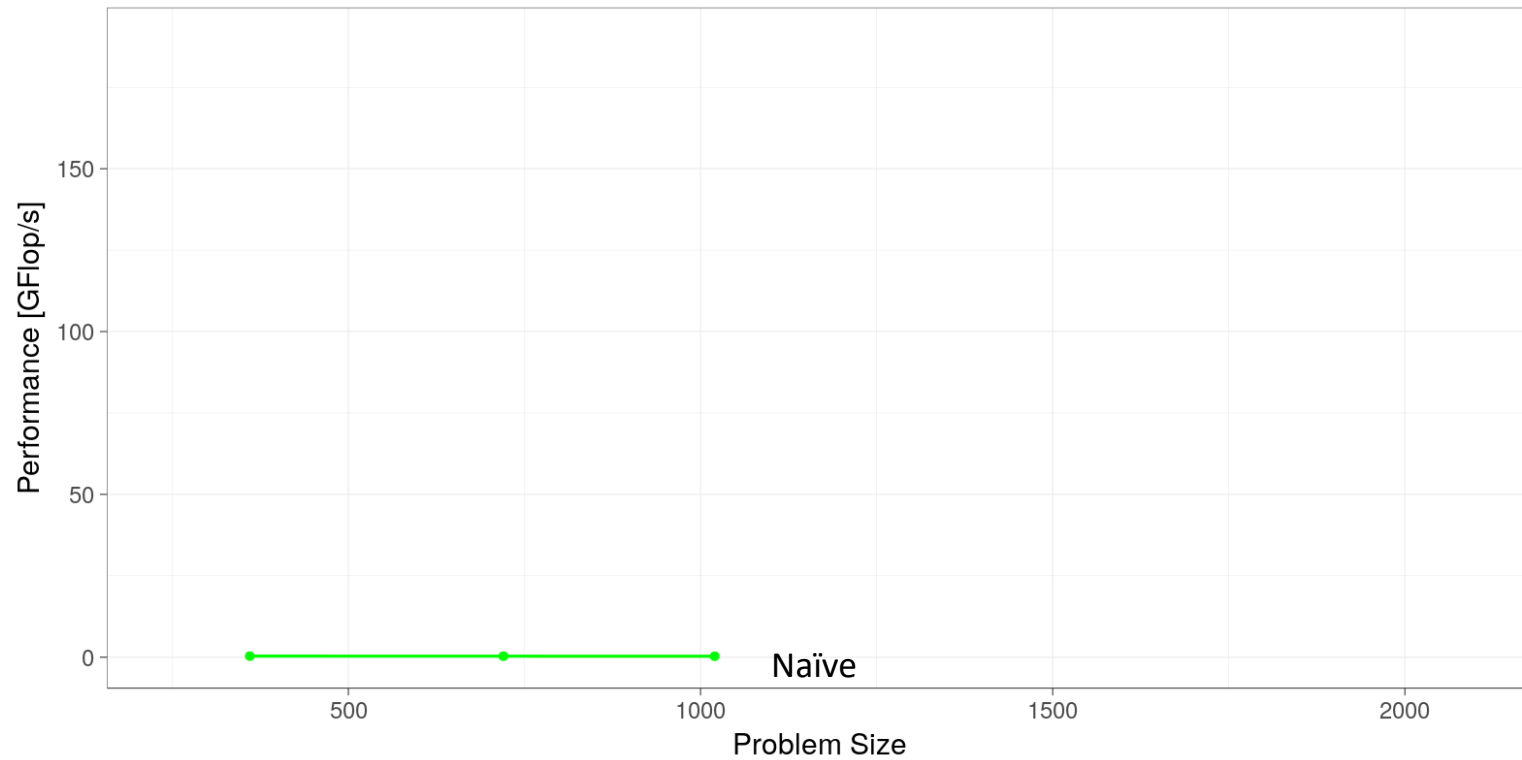
unroll  OFF

## SDFG

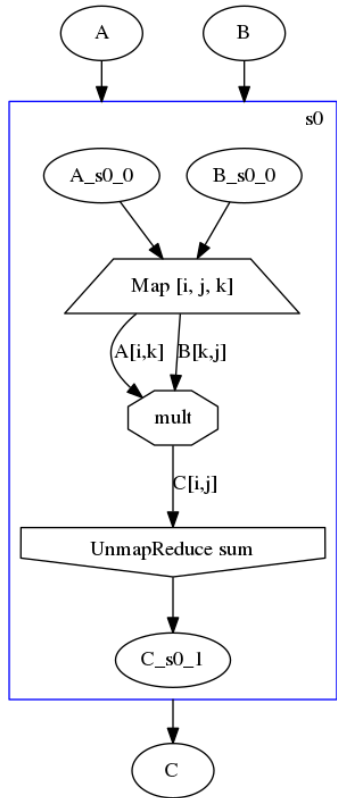
# Performance for matrix multiplication on x86



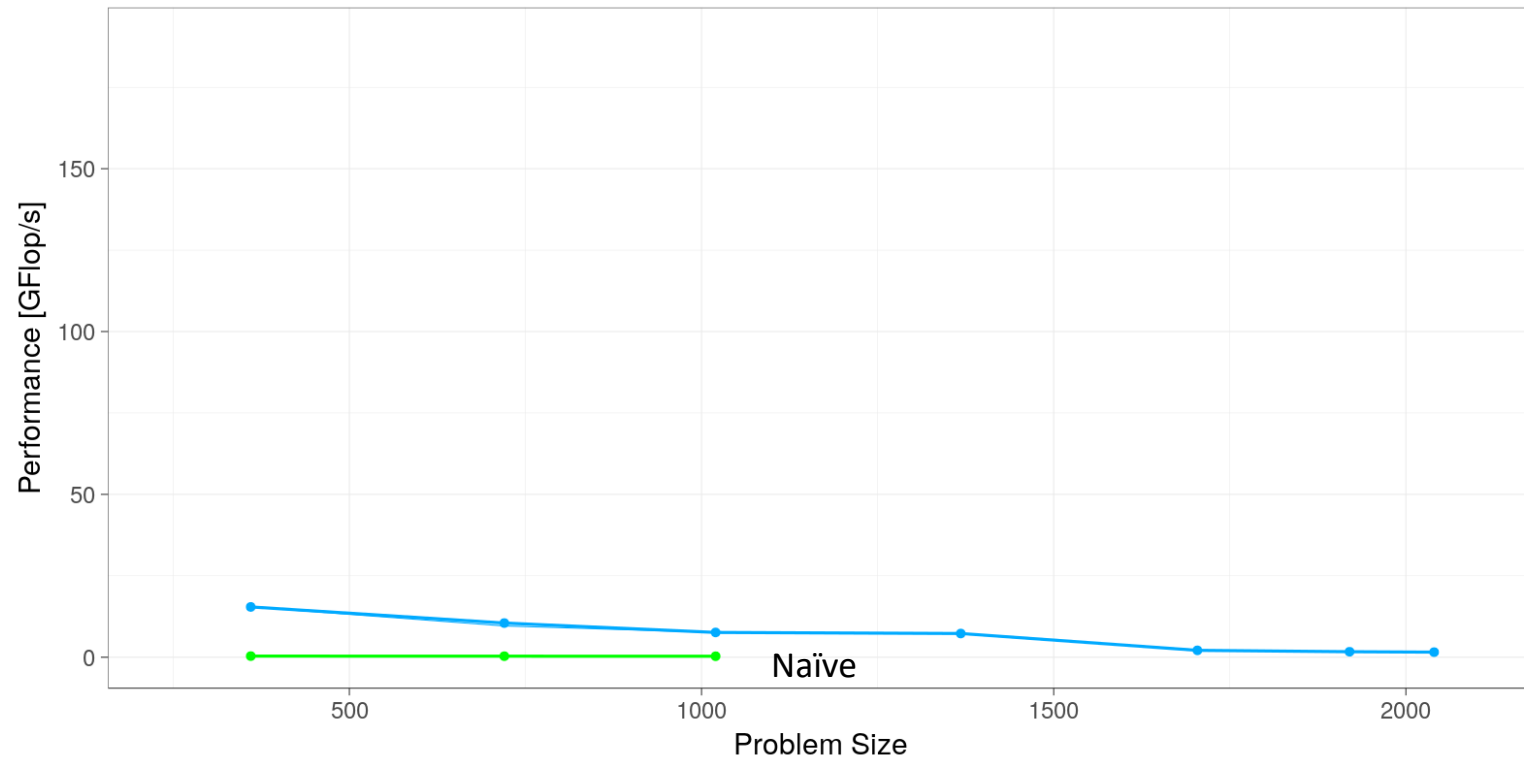
SDFG



# Performance for matrix multiplication on x86



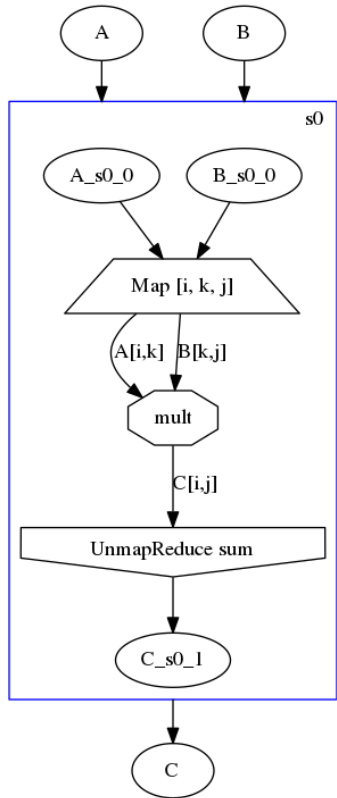
SDFG



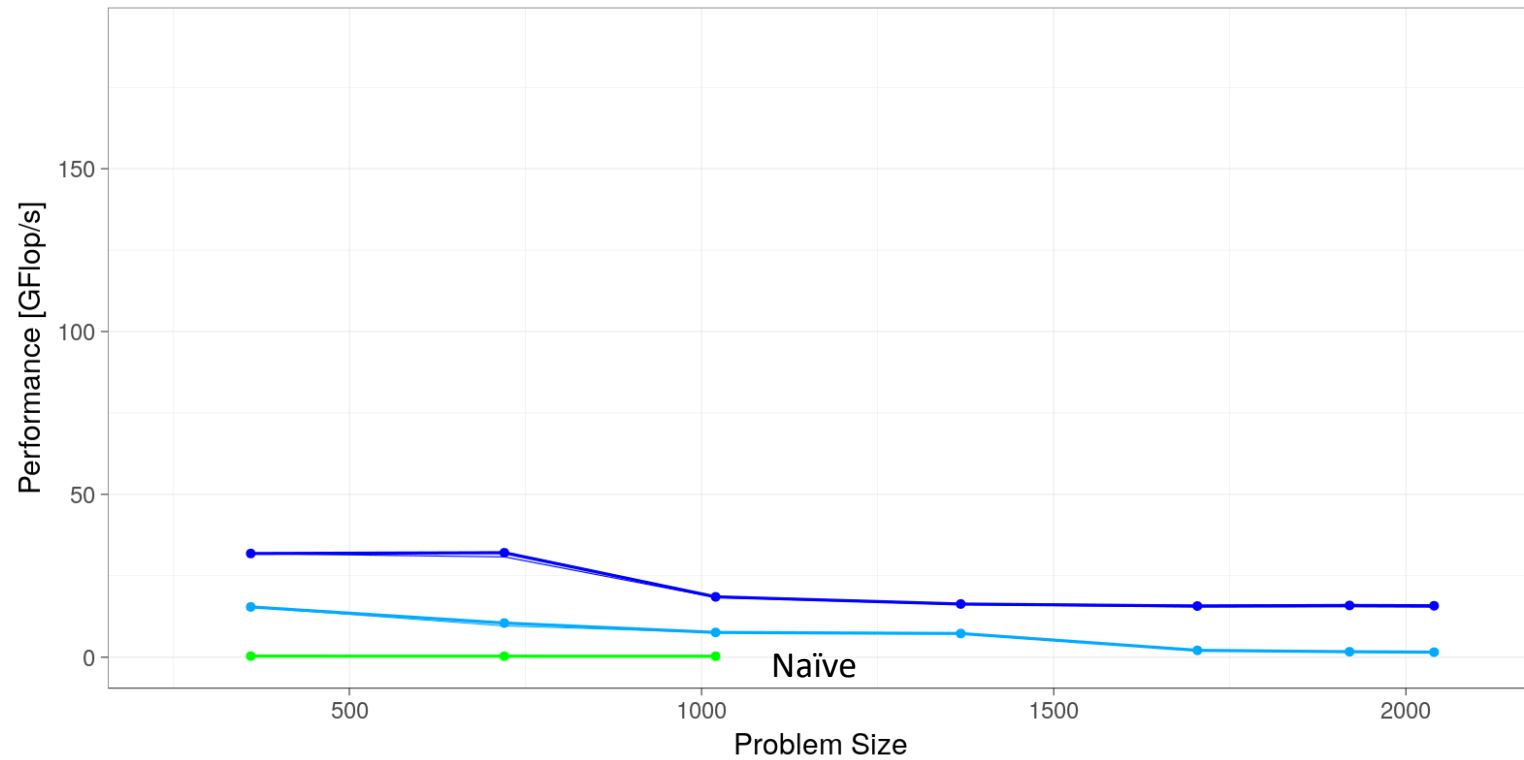
MapReduceFusion



# Performance for matrix multiplication on x86



SDFG

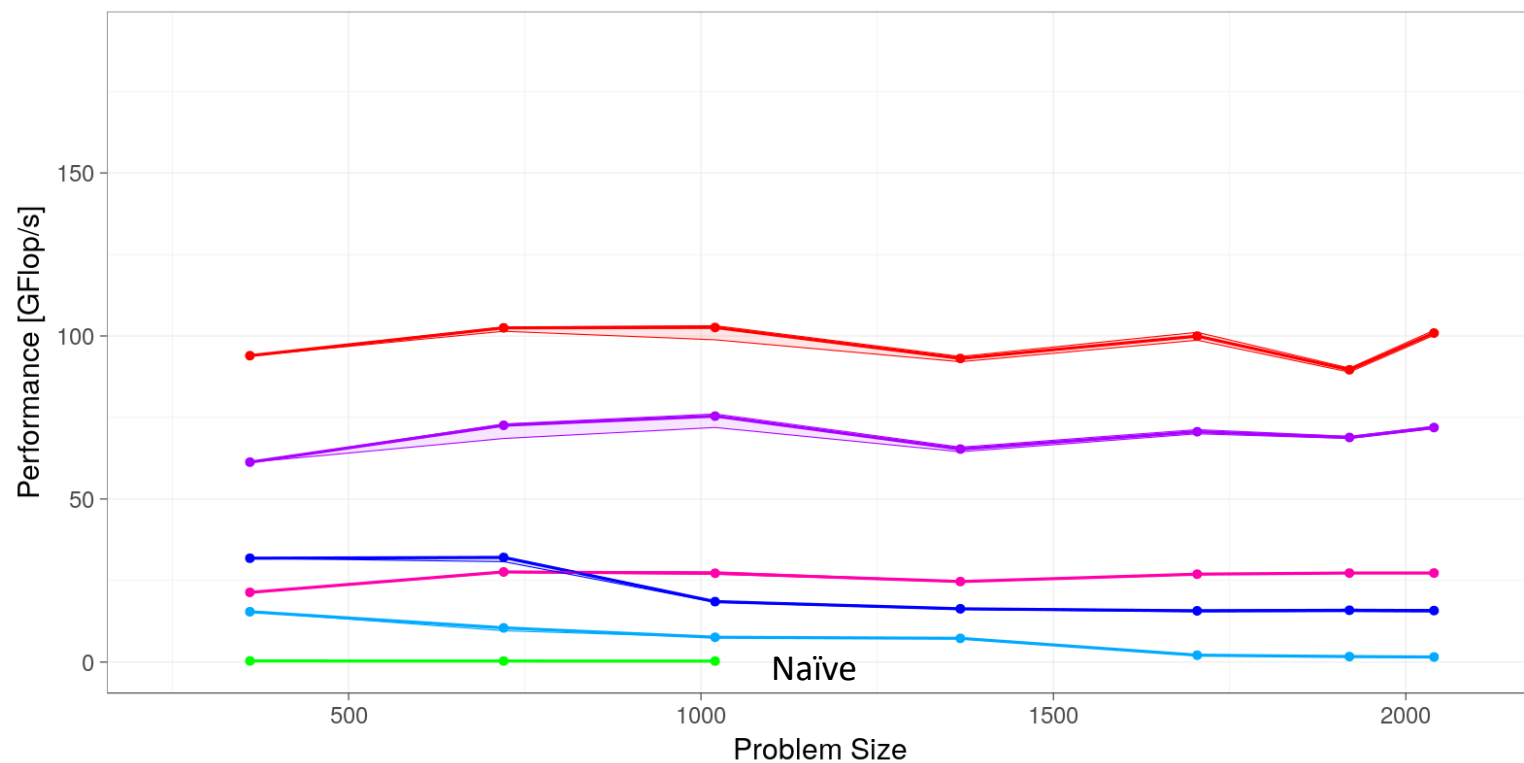
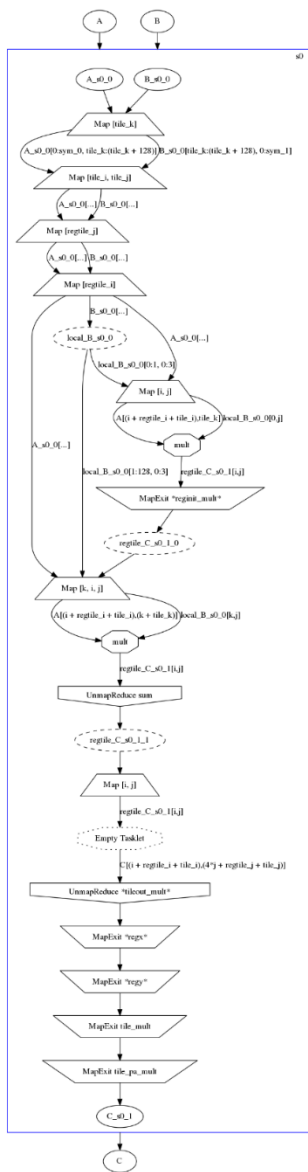


LoopReorder  
MapReduceFusion





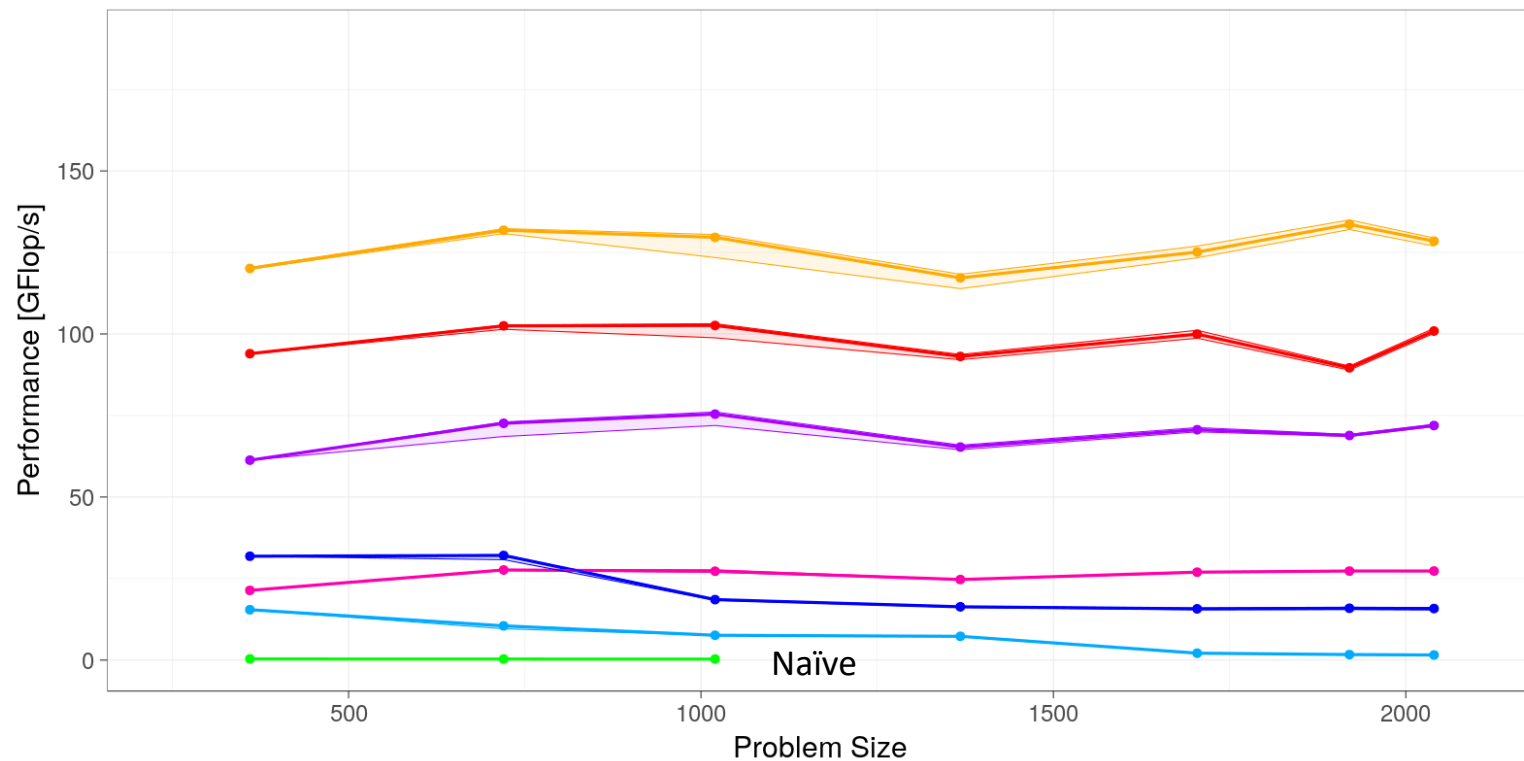
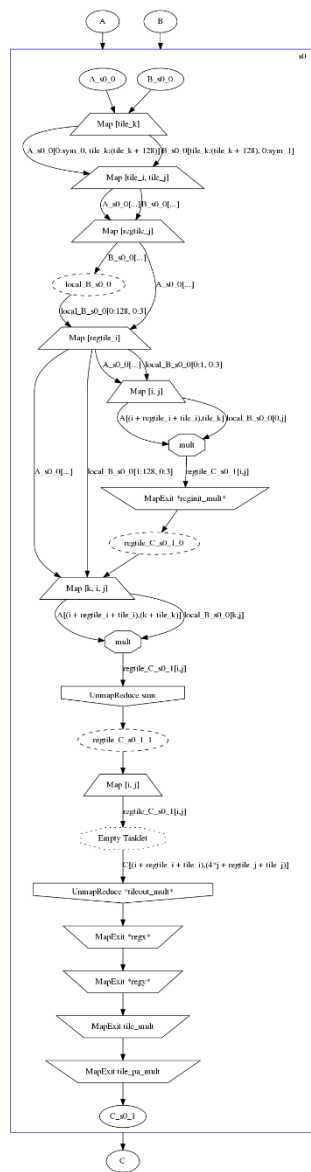
# Performance for matrix multiplication on x86



LocalStorage  
 RegisterTiling  
 BlockTiling  
 LoopReorder  
 MapReduceFusion



# Performance for matrix multiplication on x86



PromoteTransient

LocalStorage

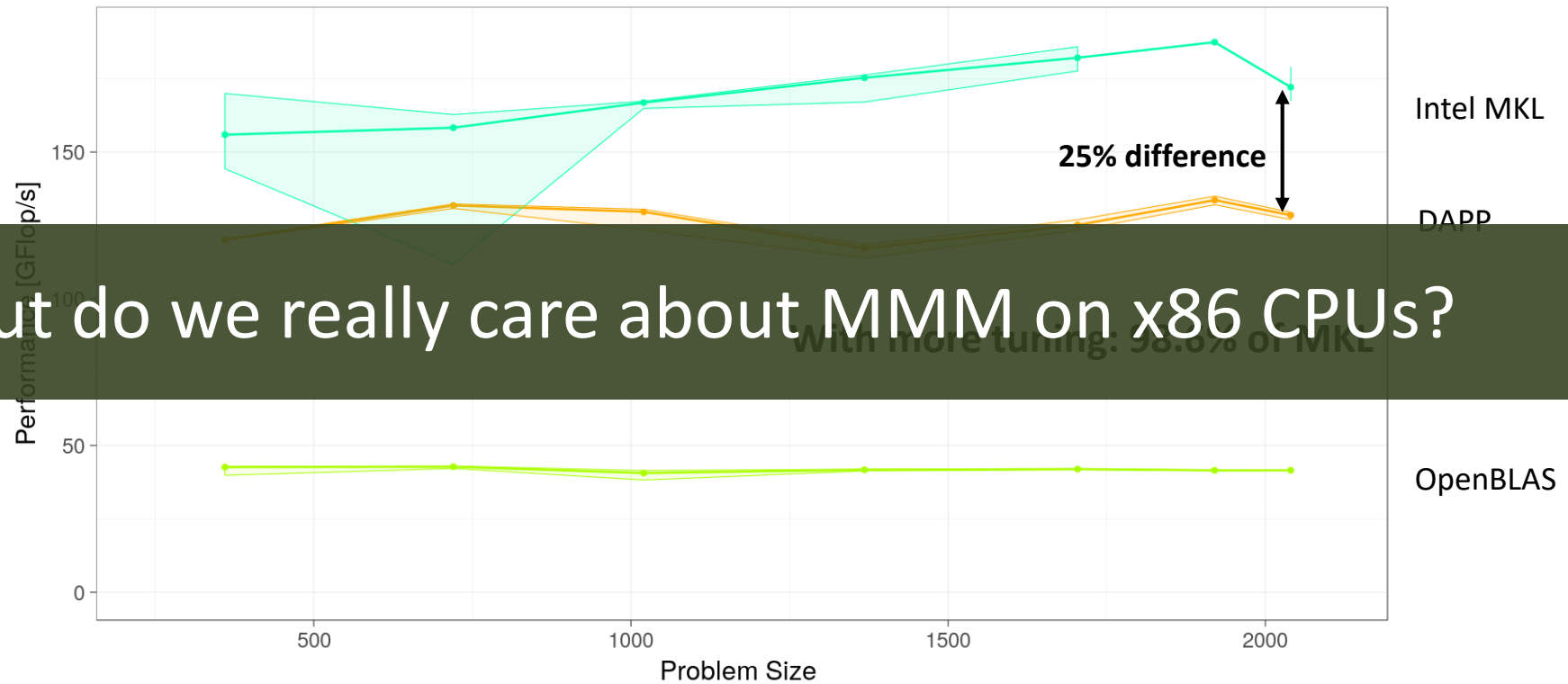
RegisterTiling

BlockTiling

LoopReorder

MapReduceFusion

# Performance for matrix multiplication on x86



But do we really care about MMM on x86 CPUs?

With more tuning: 98.6% of MKL

# 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
  - Memory access validation on compilation
  - Automatic CPU SDFG to GPU transformation
  
- **Tasklet code immutable**

```

void _program_gemm(int sym_0, int sym_1, int sym_2, double * __re
// State s0
for (int tile_k = 0; tile_k < sym_2; tile_k += 128) {
    #pragma omp parallel for
    for (int tile_i = 0; tile_i < sym_0; tile_i += 64) {
        for (int tile_j = 0; tile_j < sym_1; tile_j += 240) {
            for (int regtile_j = 0; regtile_j < (min(240, sym

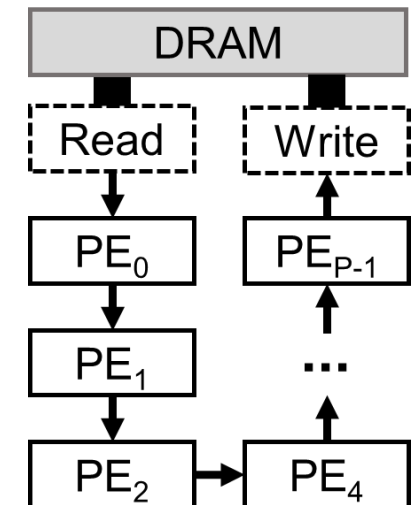
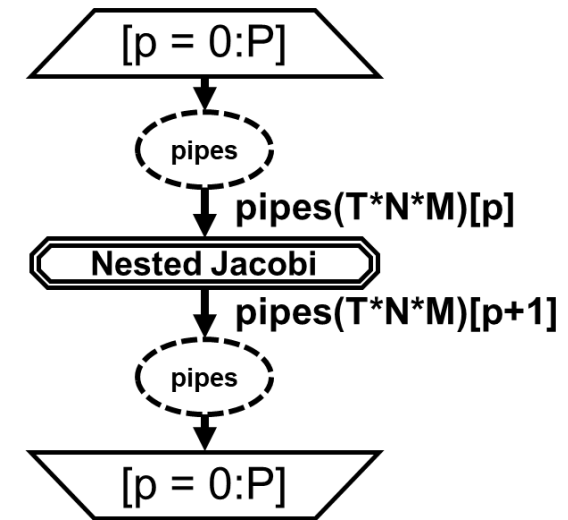
vec<double, 4> local_B_s0_0[128 * 3];
Global2Stack_2D_FixedWidth<double, 4, 3>(&B[t
local

for (int regtile_i = 0; regtile_i < (min(64,
vec<double, 4> regtile_C_s0_1[4 * 3];
for (int i = 0; i < 4; i += 1) {
    for (int j = 0; j < 3; j += 1) {
        double in_A = A[(i + regtile_i +
vec<double, 4> in_B = local_B_s0_0
// Tasklet code (mult)
auto out = (in_A * in_B);
regtile_C_s0_1[i*3 + j] = out;
    }
}
for (int k = 1; k < (min(128, sym_2 - til
// ...

```

# Hardware Mapping: Pipelined Architectures

- **Module generation with HDL and HLS**
  - Integration with Xilinx SDAccel
  - Nested SDFGs become FPGA state machines
  
- **Parallelism**
  - Exploiting temporal locality: Pipelines
  - Exploiting spatial locality: Vectorization, replication
  
- **Replication**
  - Enables parametric systolic array generation
  
- **Memory access**
  - Burst memory access, vectorization
  - Streams for inter-PE communication



# Performance (Portability) Evaluation

- **Three platforms:**

- Intel Xeon E5-2650 v4 CPU (2.20 GHz, no HT)
- Tesla P100 GPU
- Xilinx VCU1525 hosting an XCVU9P FPGA

- **Compilers and frameworks:**

- Compilers:

*GCC 8.2.0*

*Clang 6.0*

*icc 18.0.3*

- Polyhedral optimizing compilers:

*Polly 6.0*

*Pluto 0.11.4*

*PPCG 0.8*

- GPU and FPGA compilers:

*CUDA nvcc 9.2*

*Xilinx SDAccel 2018.2*

- Frameworks and optimized libraries:

*HPX*

*Halide*

*Intel MKL*

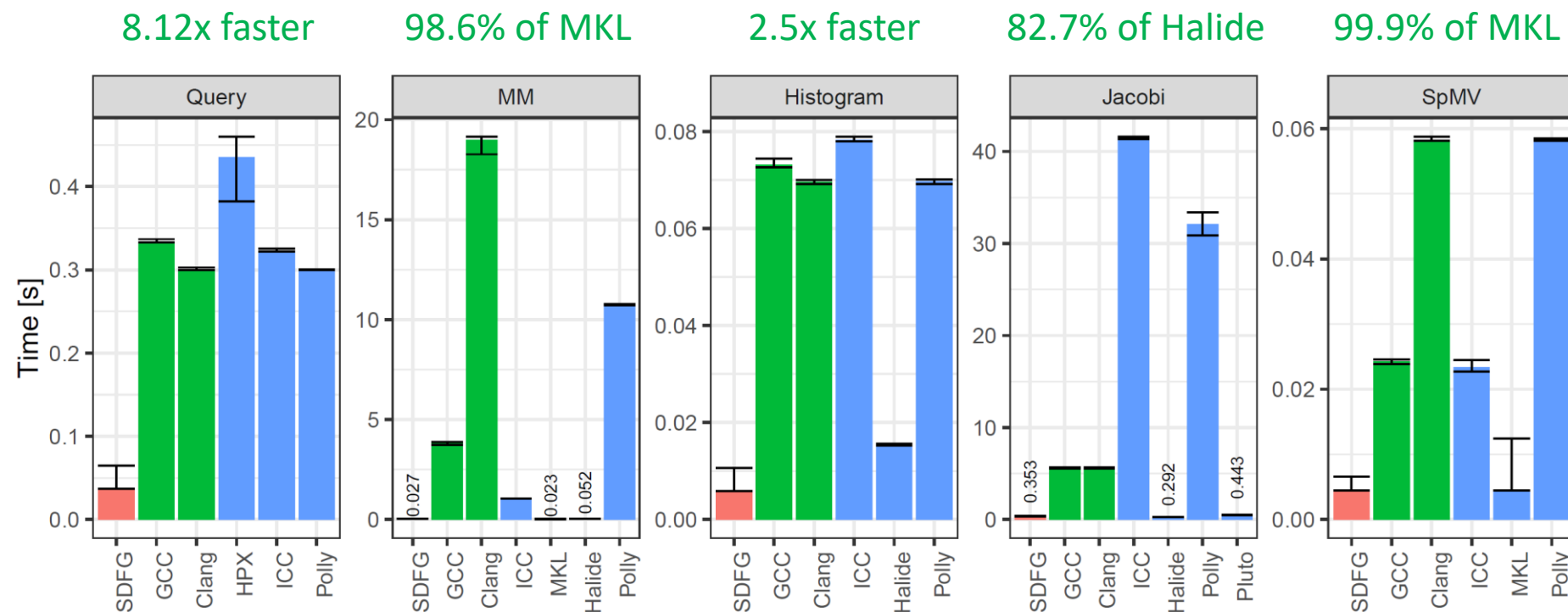
*NVIDIA CUBLAS, CUSPARSE, CUTLASS*

*NVIDIA CUB*



# Performance Evaluation: Fundamental Kernels (CPU)

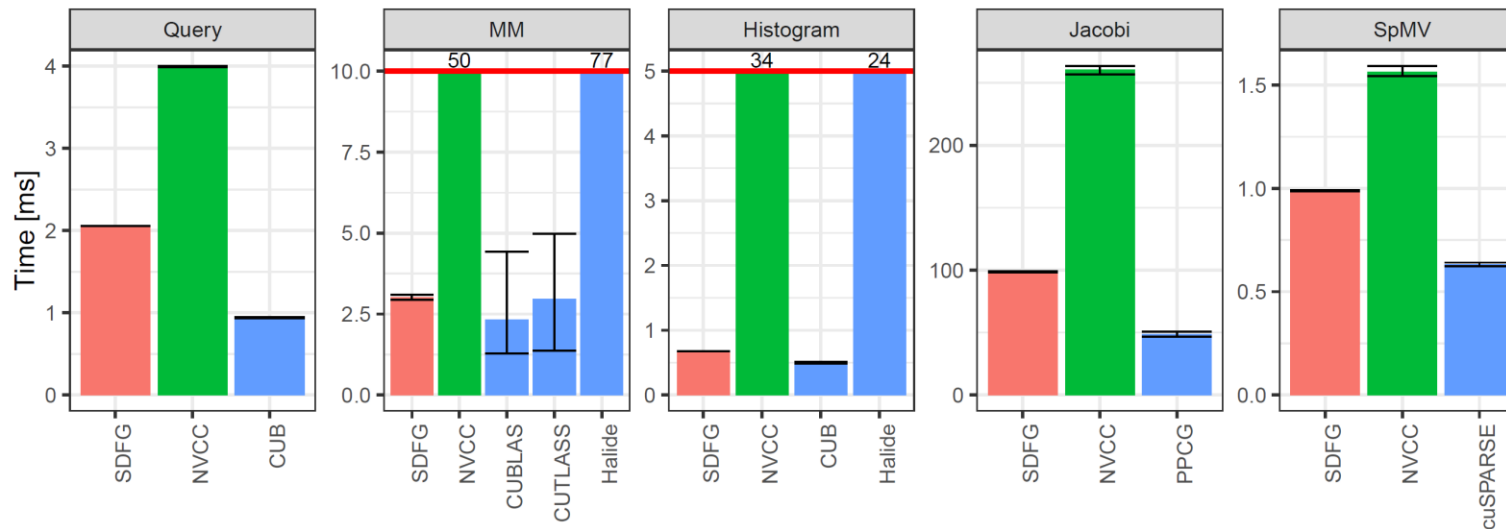
- **Database Query:** roughly 50% of a 67,108,864 column
- **Matrix Multiplication (MM):** 2048x2048x2048
- **Histogram:** 8192x8192
- **Jacobi stencil:** 2048x2048 for T=1024
- **Sparse Matrix-Vector Multiplication (SpMV):** 8192x8192 CSR matrix (nnz=33,554,432)



# Performance Evaluation: Fundamental Kernels (GPU, FPGA)

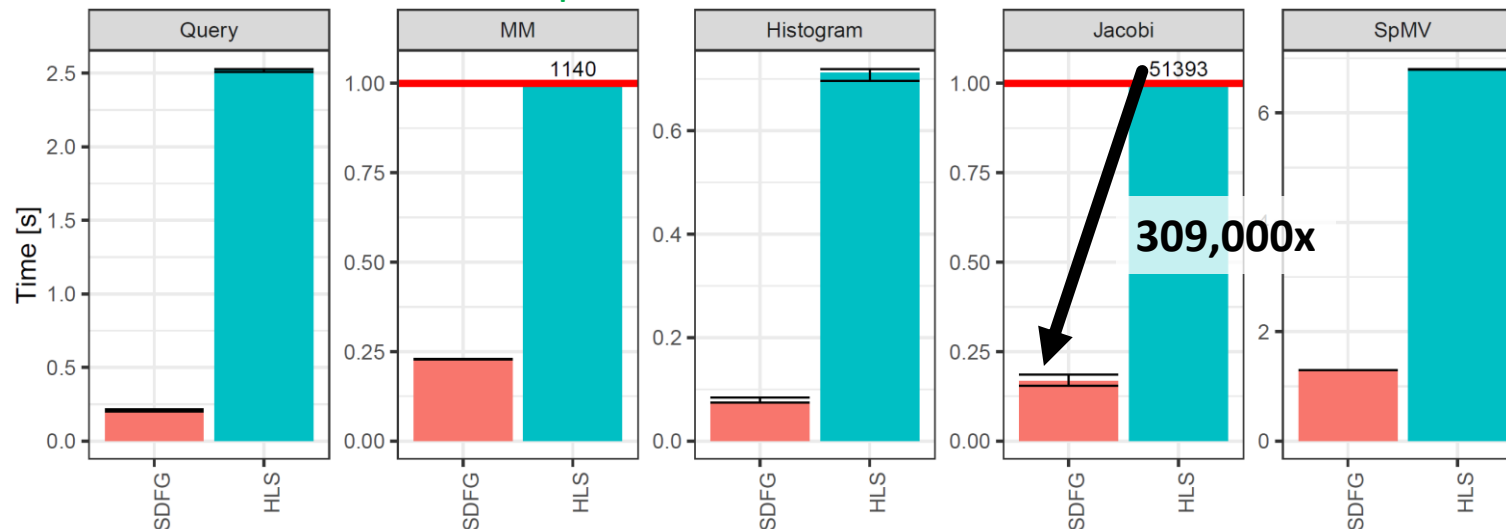
90% of CUTLASS

GPU



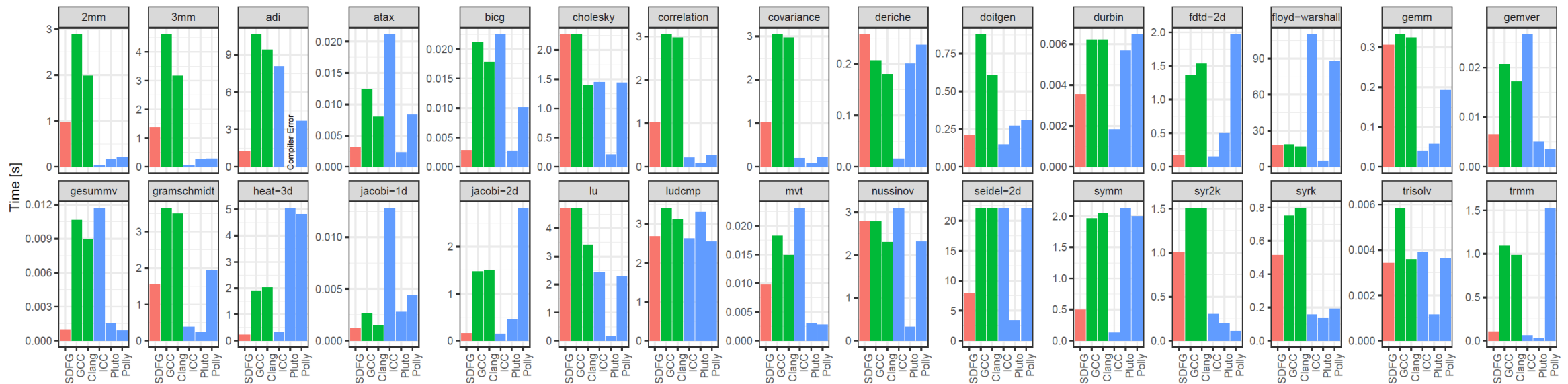
19.5x of Spatial

FPGA



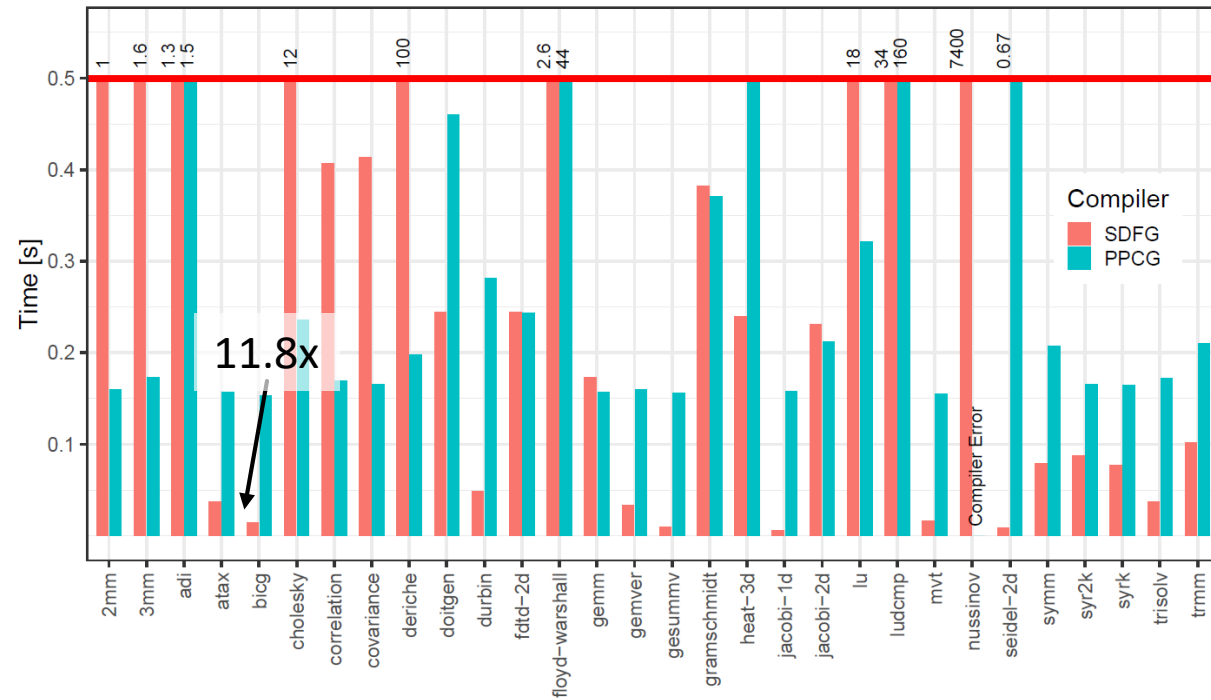
# Performance Evaluation: Polybench (CPU)

- Polyhedral benchmark with 30 applications
- Without any transformations, achieves 1.43x (geometric mean) over general-purpose compilers



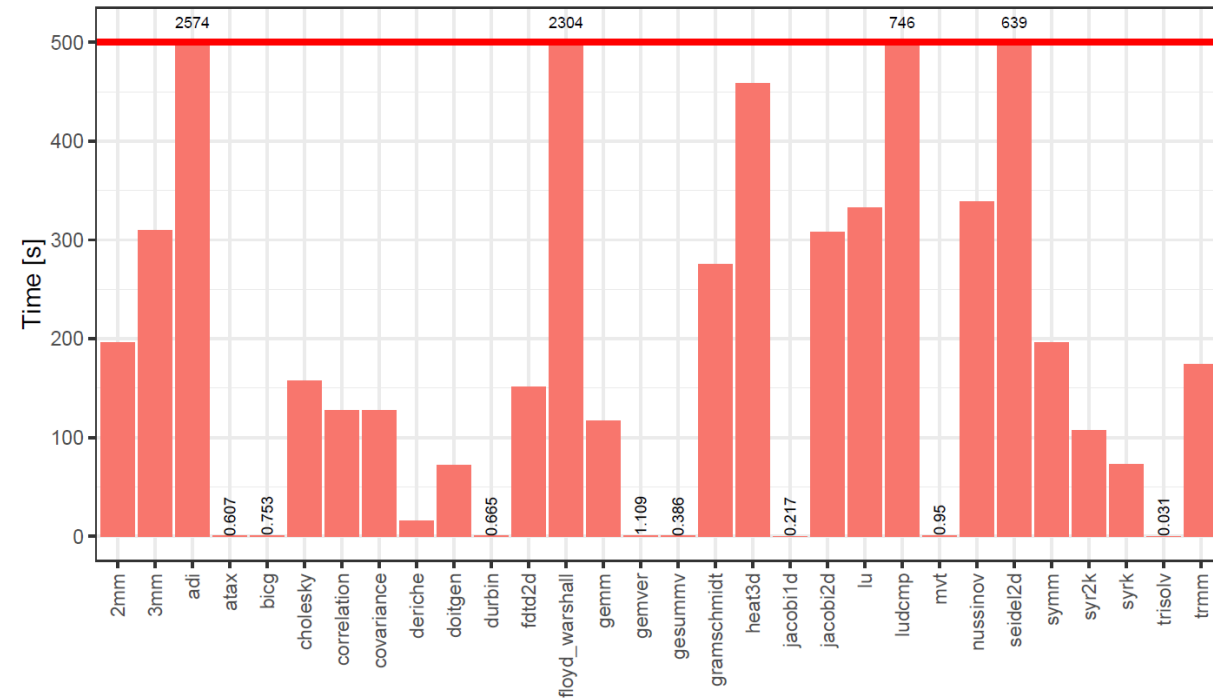
# Performance Evaluation: Polybench (GPU, FPGA)

- Automatically transformed from CPU code



**GPU**

(1.12x geomean speedup)

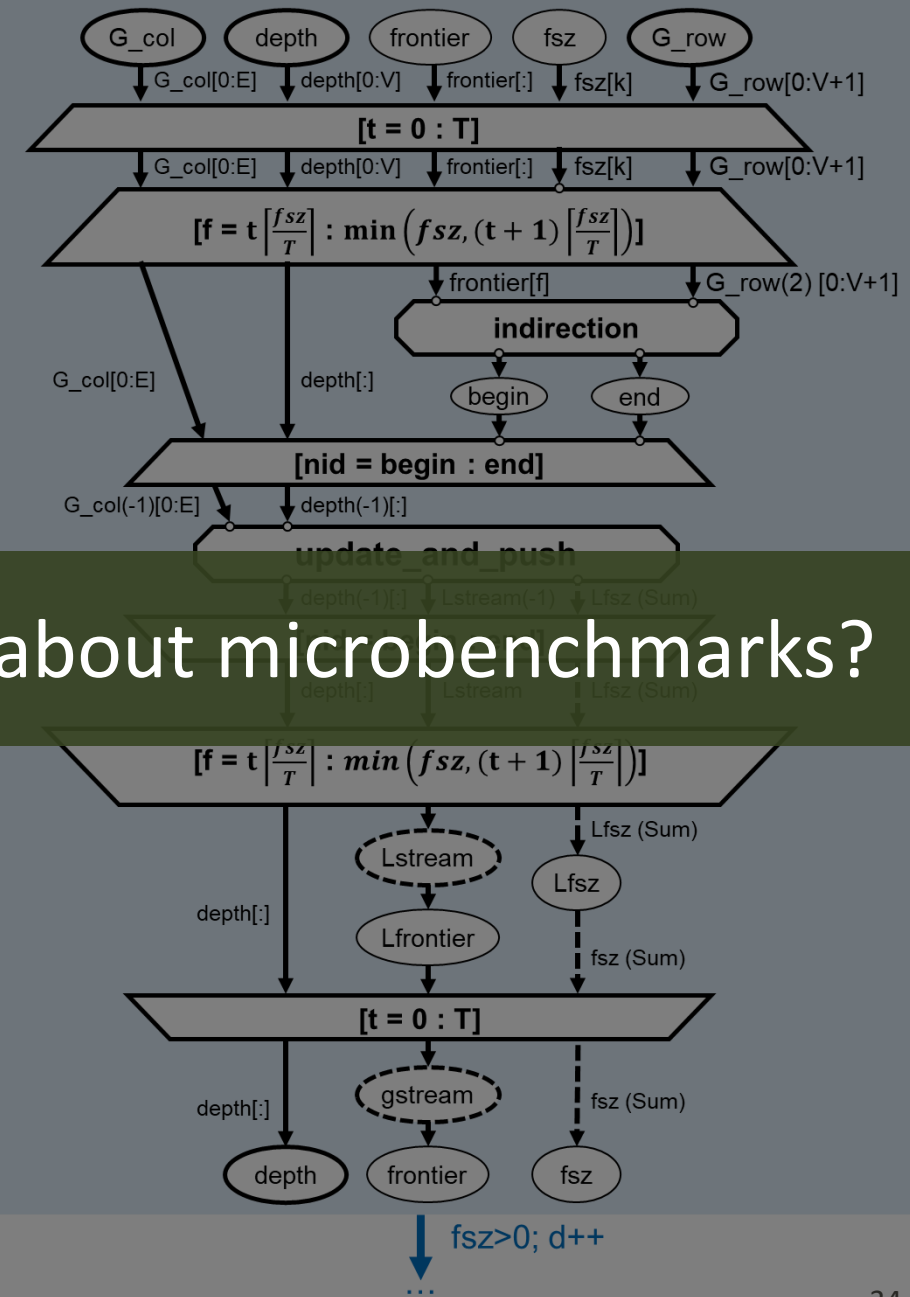


**FPGA**

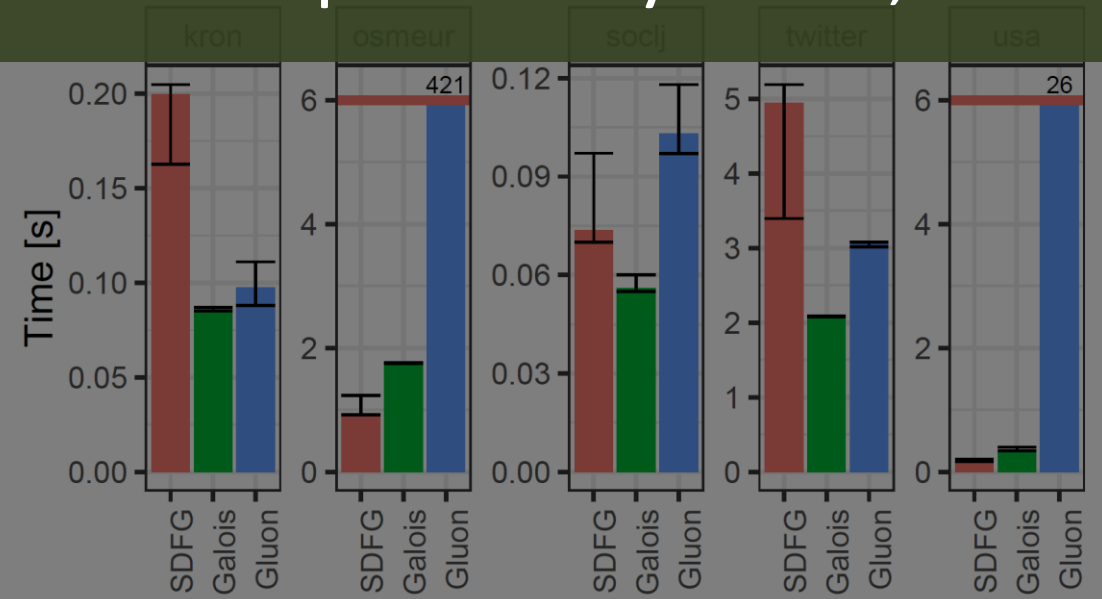
The **first** full set of placed-and-routed Polybench

# Case Study: Parallel Breadth-First Search

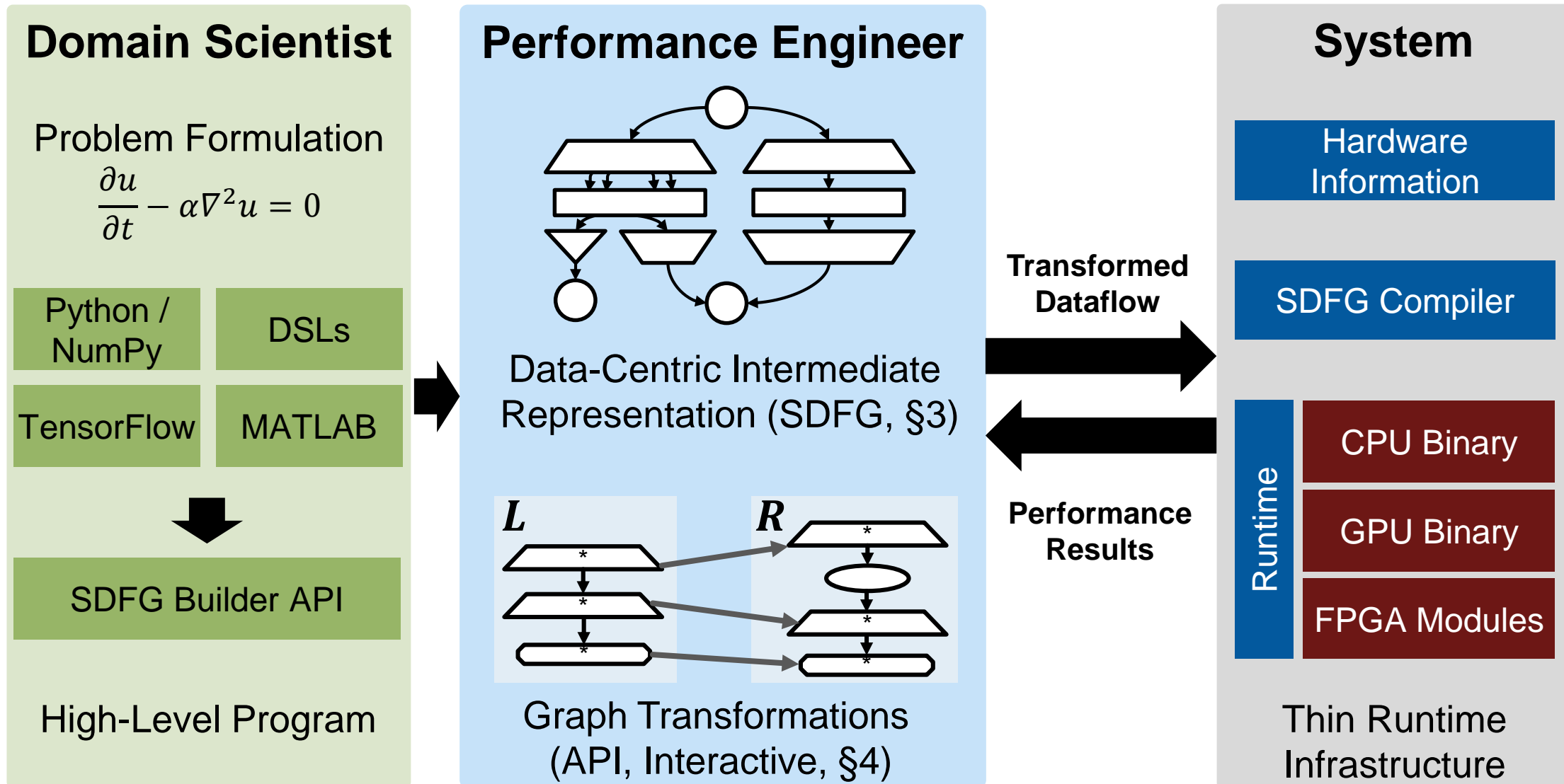
- Compared with Galois and Gluon
  - State-of-the-art graph processing frameworks on CPU
- Graphs:
  - Road maps: USA, OSM-Europe
  - Social networks: Twitter, LiveJournal



Performance portability – fine, but who cares about microbenchmarks?

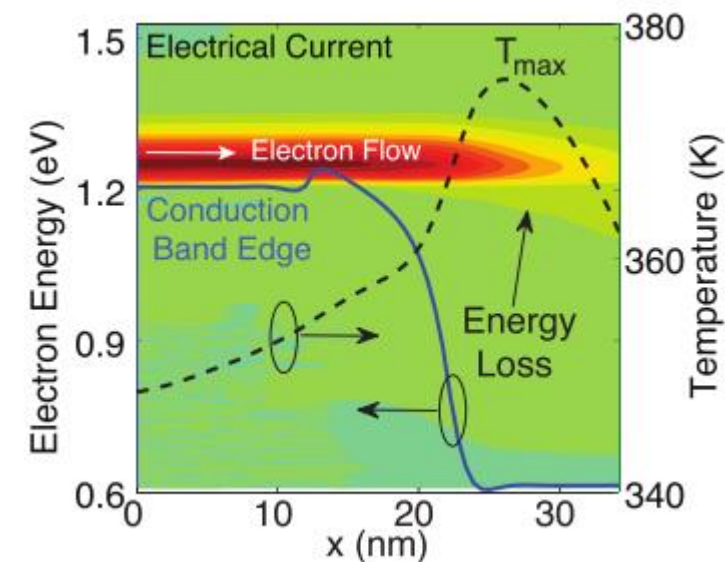
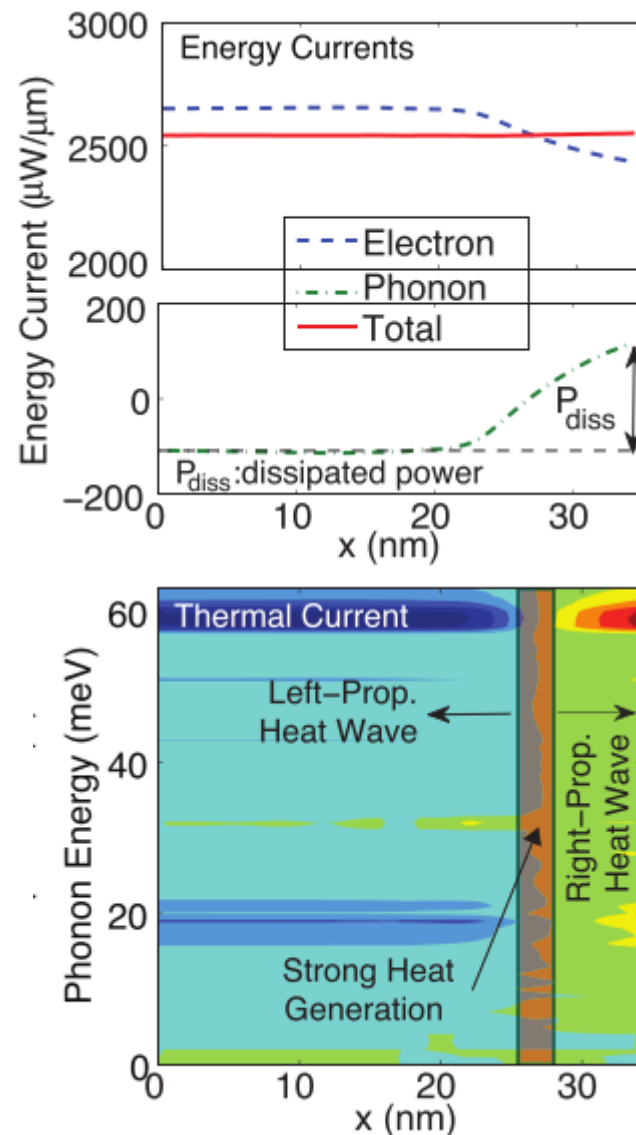
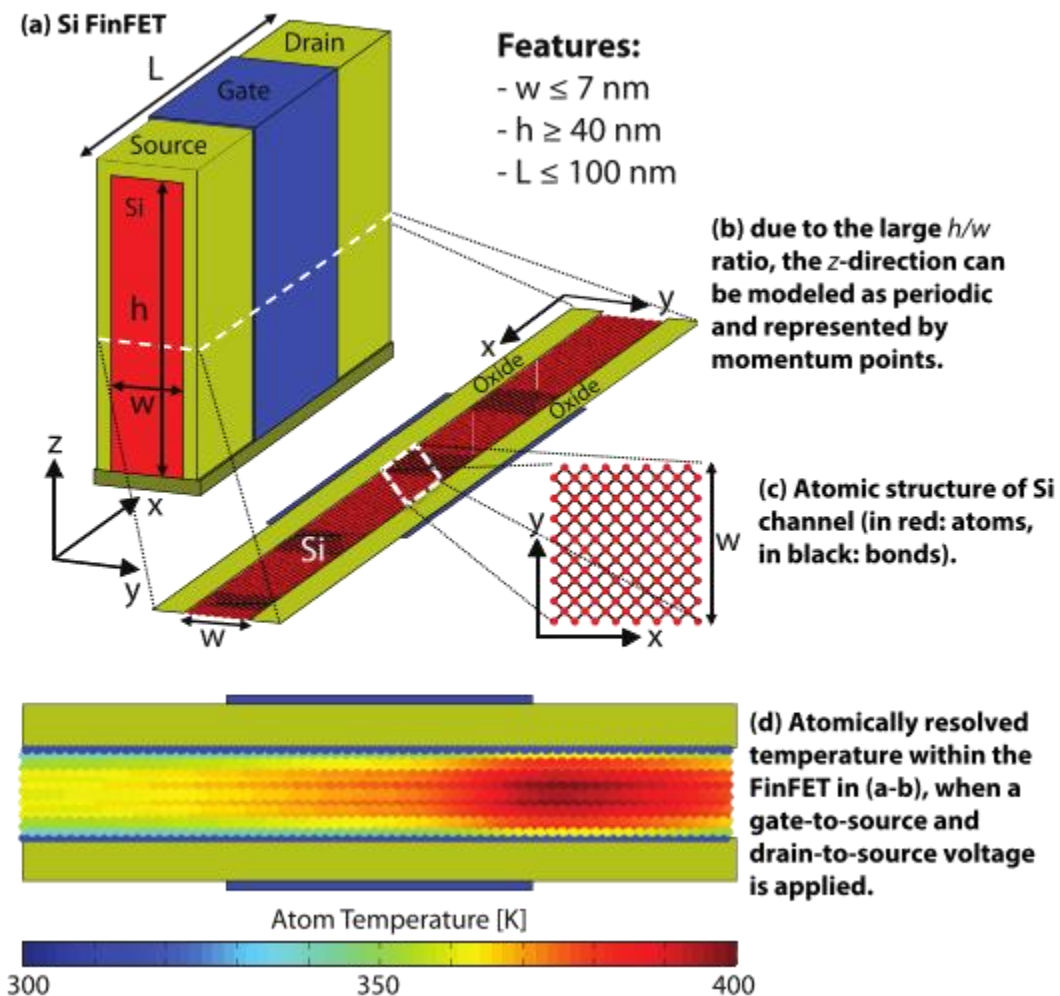


# Remember the promise of DAPP – on to a real application!



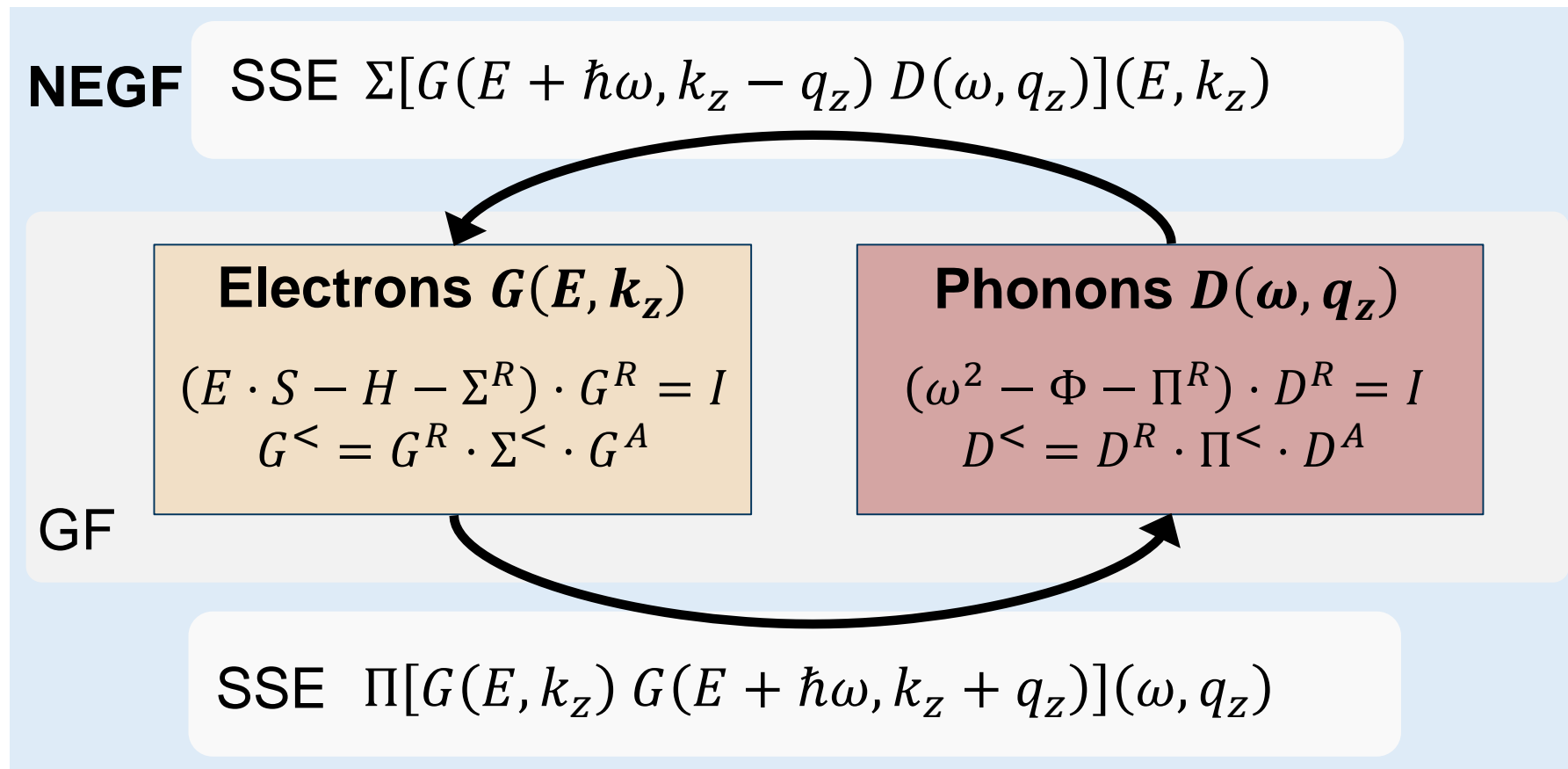


# Next-Generation Transistors need to be cooler – addressing self-heating

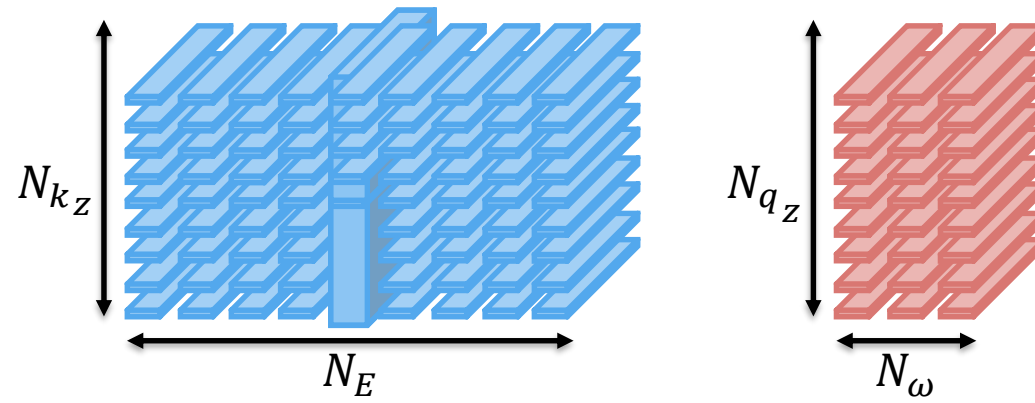


# Quantum Transport Simulations with OMEN

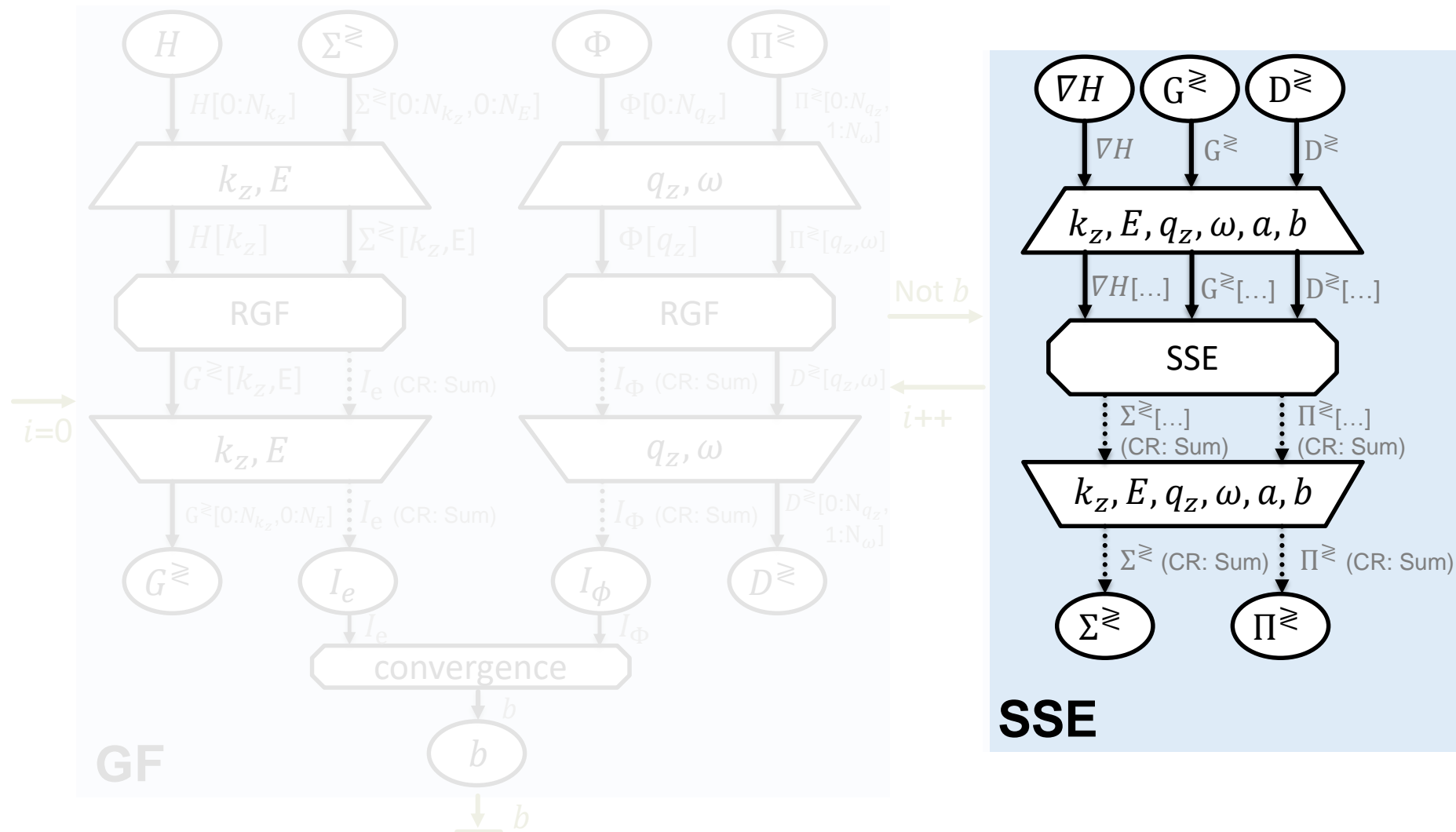
- OMEN Code (Luisier et al., **Gordon Bell award finalist 2011 and 2015**)
  - 90k SLOC, C, C++, CUDA, MPI, OpenMP, ...



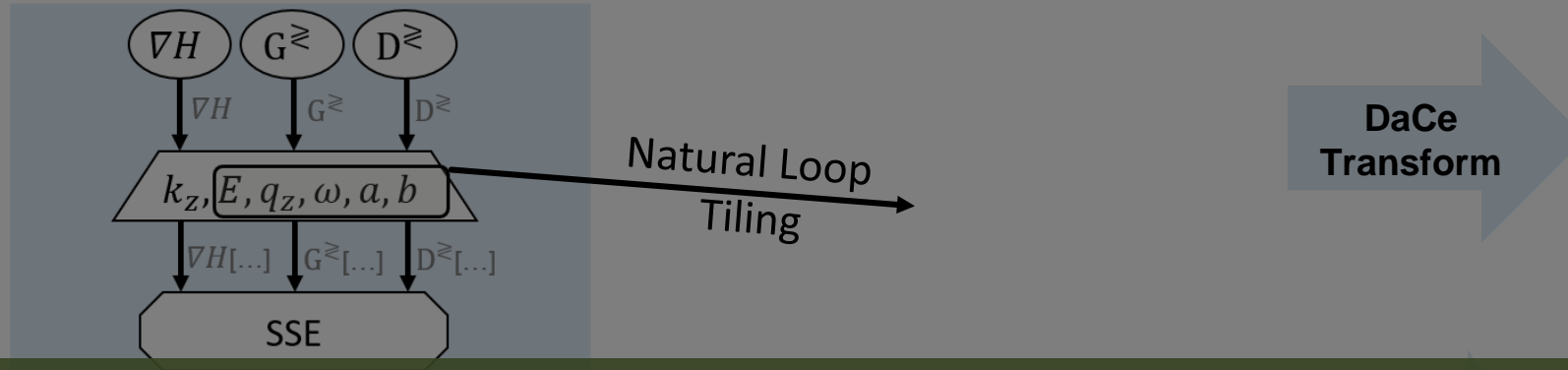
# OMEN



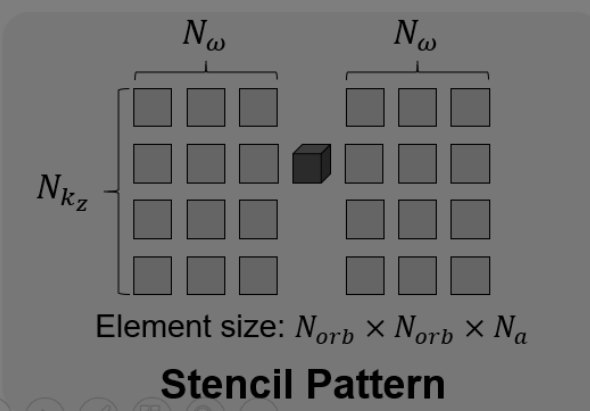
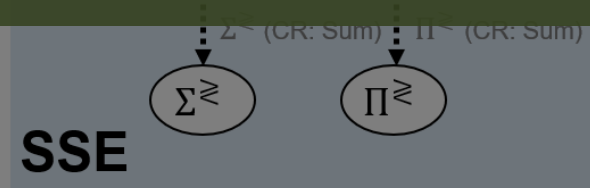
# All of OMEN (90k SLOC) in a single SDFG – (collapsed) tasklets contain more SDFGs



# Zooming into SSE (large share of the runtime)



Between 100-250x less communication at scale! (from PB to TB)



# Additional interesting performance insights

Python is slow! Ok, we knew that – but compiled can be fast!

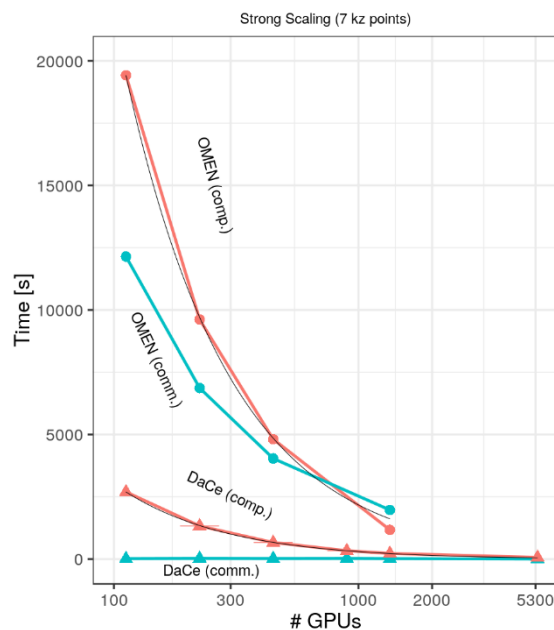
Variant	Phase					
	GF			SSE		
	Tflop	Time [s]	% Peak	Tflop	Time [s]	% Peak
OMEN	174.0	144.14	23.2%	63.6	965.45	1.3%
Python	174.0	1,342.77	2.5%	63.6	30,560.13	0.2%
DaCe	174.0	<b>111.25</b>	<b>30.1%</b>	31.8	<b>29.93</b>	<b>20.4%</b>

Piz Daint single node (P100)

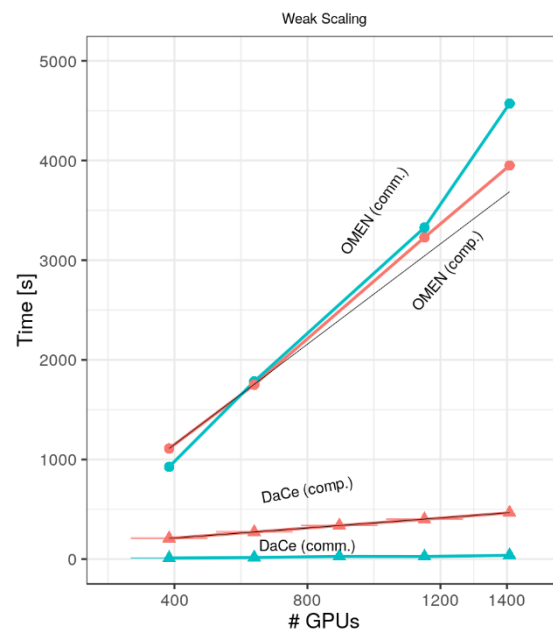
cuBLAS can be very inefficient (well, unless you floptimize)

GPU	cuBLAS			DaCe (SBSMM)		
	Gflop	Time	% Peak (Useful)	Gflop	Time	% Peak
P100	27.42	6.73 ms	86.6% (6.1%)	1.92	4.03 ms	10.1%
V100	27.42	4.62 ms	84.8% (5.9%)	1.92	0.97 ms	28.3%

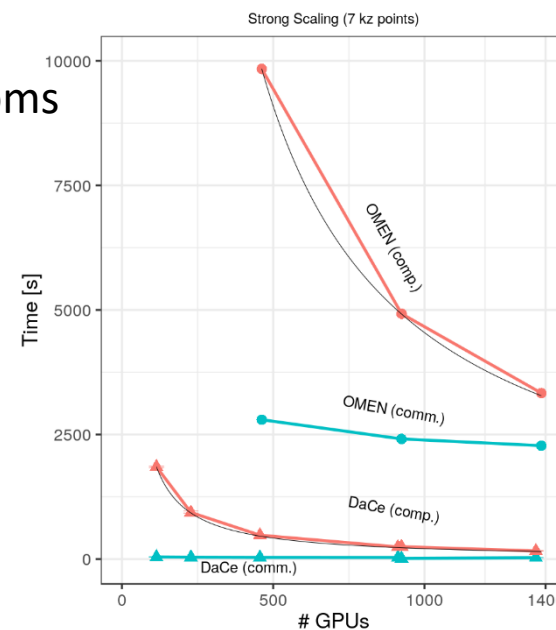
Basic operation in SSE (many very small MMMs)



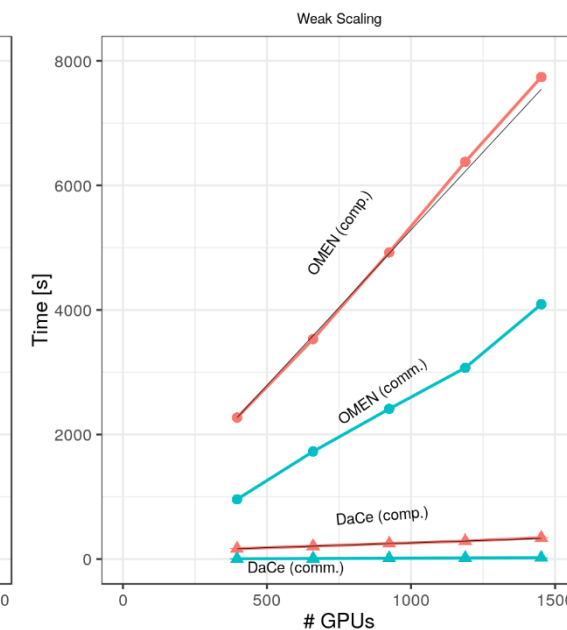
Piz Daint



5k atoms



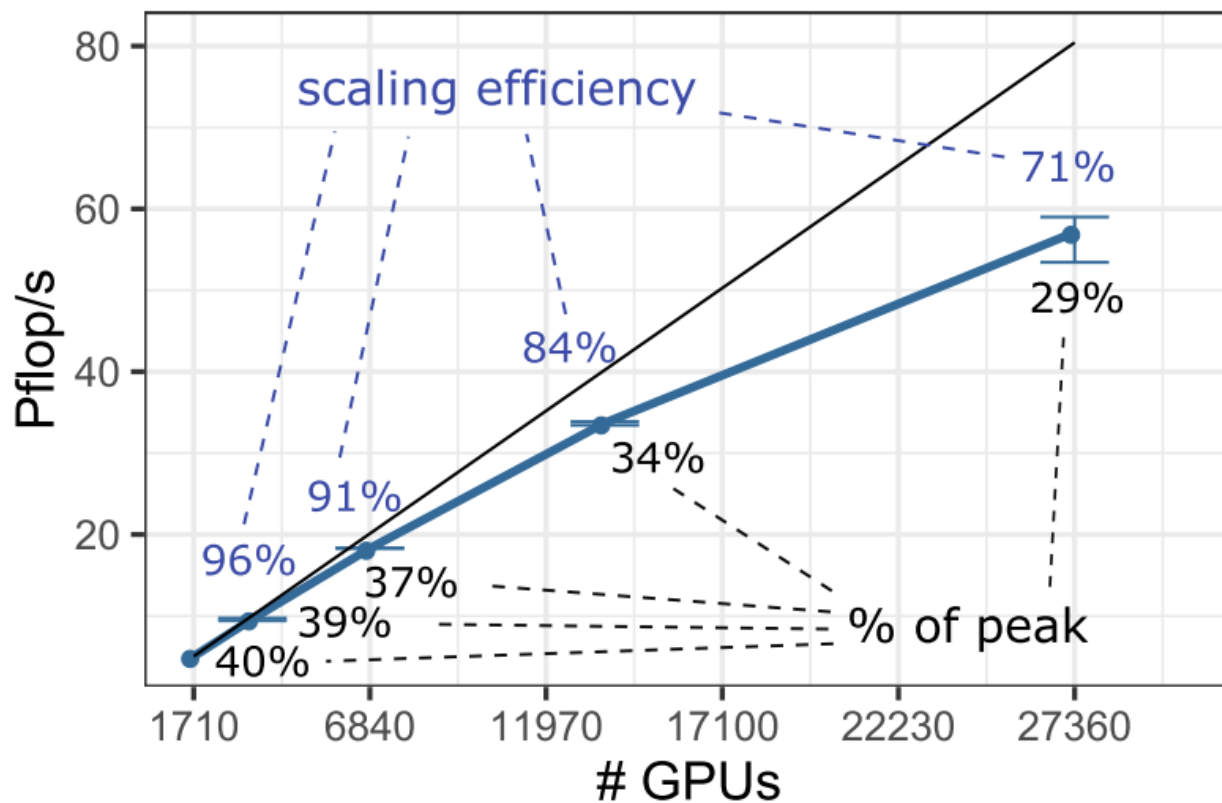
Summit





# 10,240 atoms on 27,360 V100 GPUs (full-scale Summit)

- 56 Pflop/s with I/O (28% peak)



Variant	$N_a$	Time [s]	Time/Atom [s]	Speedup
OMEN	1,064	4695.70	4.413	1.0x
DaCe	10,240	489.83	0.048	92.3x

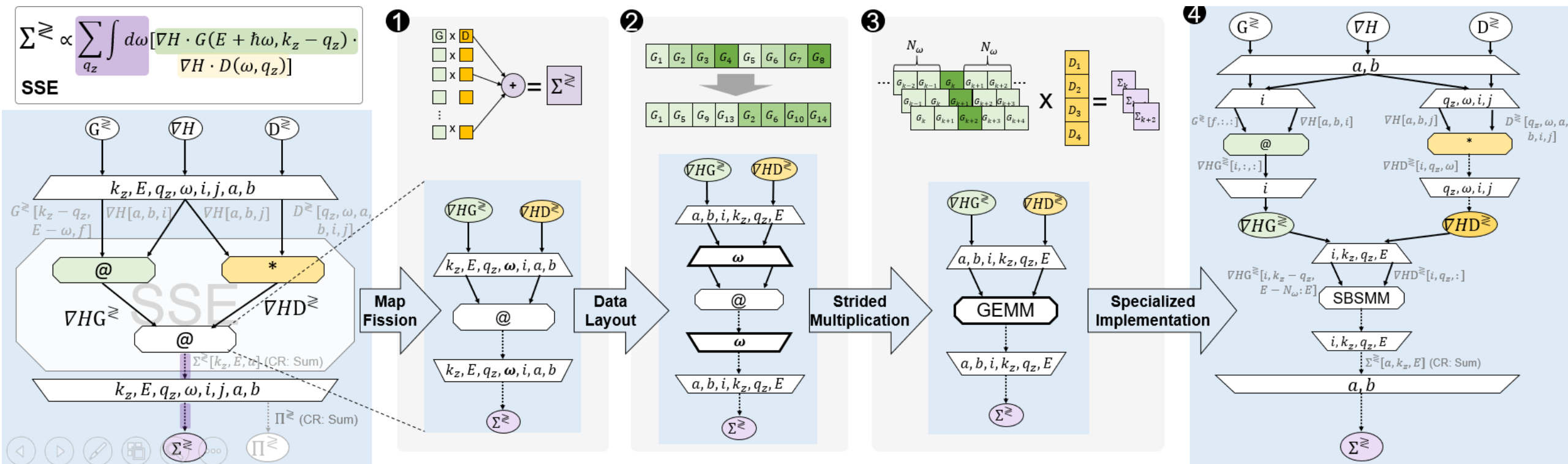
$$P = 6,840, N_b = 34, N_{orb} = 12, N_E = 1, N_\omega = 70.$$

Already ~100x speedup on 25% of Summit – the original OMEN does not scale further!

Communication time reduced by 417x on Piz Daint!

Volume on full-scale Summit from 12 PB/iter → 87 TB/iter

# An example of fine-grained data-centric optimization (i.e., how to vectorize)



# Overview and wrap-up

### Single Instruction Multiple Data/Threads (SIMD - Vector CPU, SIMT - GPU)

High Performance Computing really became a data management challenge

### Performance Portability with DataCentric (DaCe) Parallel Programming

Domain Scientist: Problem Formulation  $\frac{\partial u}{\partial t} - \alpha \nabla^2 u = 0$ , Python / NumPy, TensorFlow, DSLs, MATLAB, SDFG Builder API, High-Level Program

Performance Engineer: Data-Centric Intermediate Representation (SDFG, §3), Graph Transformations (API, Interactive, §4)

System: Hardware Information, SDFG Compiler, Runtime (CPU Binary, GPU Binary, FPGA Modules), Thin Runtime Infrastructure

### DIODE User Interface

### Next-Generation Transistors need to be cooler – addressing self-heating

Zooming into SSE (large share of the runtime)

### Zooming into SSE (large share of the runtime)

OMEN Decomposition  $\rightarrow$  Natural Loop Tiling  $\rightarrow$  Data-Centric Decomposition  $\xrightarrow{\text{DaCe Transform}}$

Between 100-250x less communication at scale! (from PB to TB)

Method: Broadcast, Reduce, P2P  
Volume:  $\mathcal{O}(N_x N_y N_z N_{nb} N_{nb}^2)$   
MPI Invocations:  $9 N_x N_y$

Method: Collective Alltoall  
Volume:  $\mathcal{O}(N_x (N_x + N_y + N_z) N_{nb}^2)$   
MPI Invocations: 4

### 10,240 atoms on 27,360 V100 GPUs (full-scale Summit)

- 56 Pfplop/s with I/O (28% peak)

Variant	$N_a$	Time [s]	Time/Atom [s]	Speedup
OMEN	1,064	4695.70	4.413	1.0x
DaCe	10,240	489.83	0.048	92.3x

$P = 6,840, N_b = 34, N_{orb} = 12, N_E = 1,220, N_{cov} = 70$

Already ~100x speedup on 25% of Summit – the original OMEN does not scale further!

Communication time reduced by 417x on Piz Daint!

Volume on full-scale Summit from 12 PB/iter  $\rightarrow$  87 TB/iter