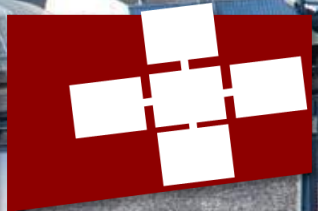


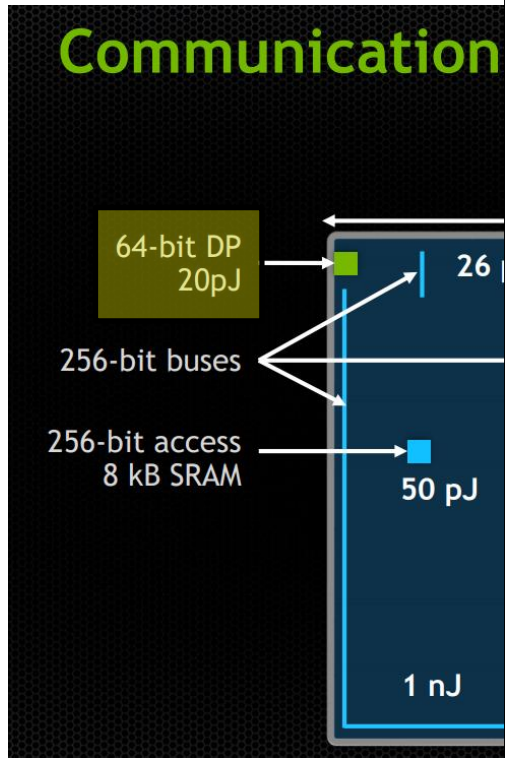
Alexandru Calotoiu, Tal Ben-Nun, Grzegorz Kwasniewski, Johannes de Fine Licht,  
Timo Schneider, Philipp Schaad, Torsten Hoefler

## Lifting C Semantics for Dataflow Optimization



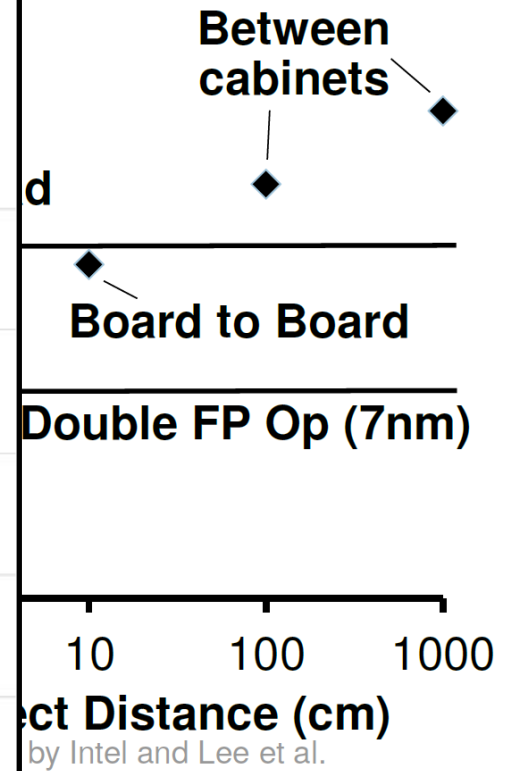
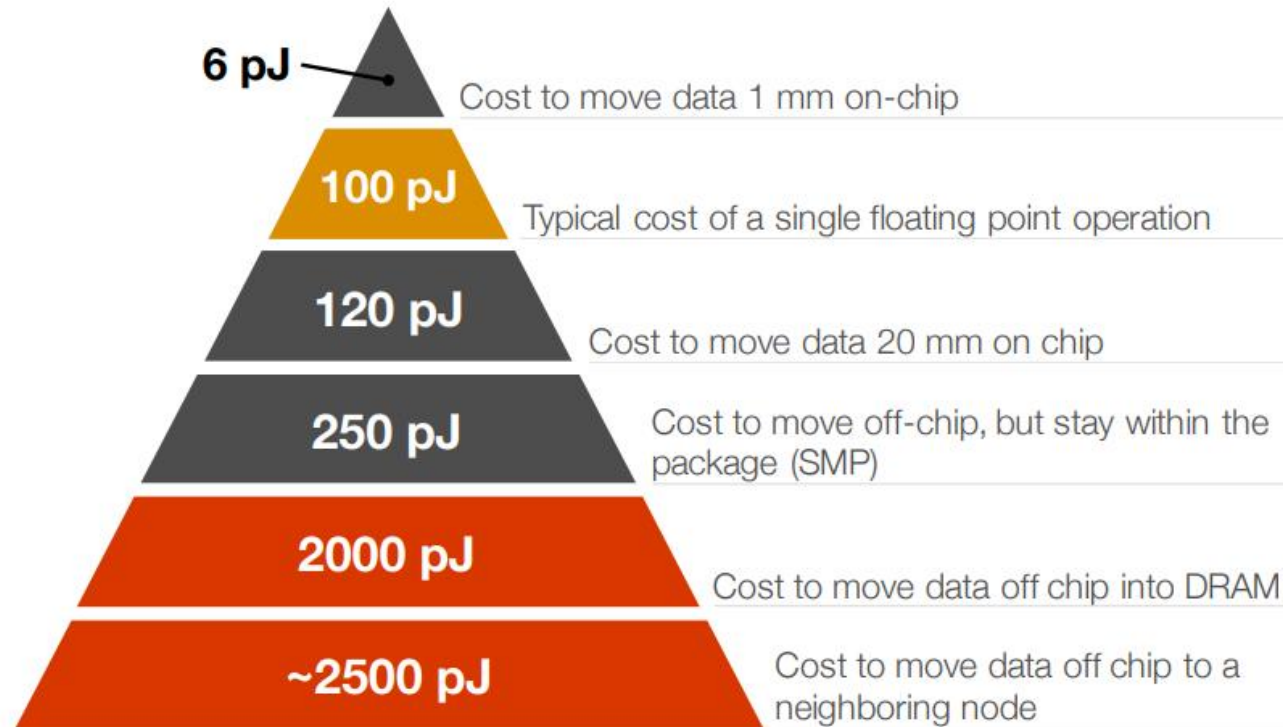


# Data movement dominates energy costs!



## Hierarchical Power Costs

Data Movement is the Dominant Power Cost



Slide

Source: Fatollahi-Fard et al.

# Efficient data movement is hard!

Halide: Decoupling  
Schedules from

## Polyhedral-Based Data Reuse Optimization for Configurable Computing

Louis-Noël Pouchet,<sup>1</sup> Peng  
<sup>1</sup> University of California, Los Angeles  
<sup>2</sup> Ohio State University

## MAPS: Optimizing Massively Parallel Applications Using Device-Level Memory Abstraction

EDUARDIN ELY LEVY, AMNON BARAK, and TAL BEN-NUN,  
University of Jerusalem



## LIFT: A Functional Data-Parallel IR for High-Performance GPU Code Generation

Michel Steuwer Toomas Rimmelg Christophe Dubach  
University of Edinburgh, United Kingdom  
{michel.steuwer, toomas.rimmelg, christophe.dubach}@ed.ac.uk

### Abstract

Parallel patterns (*e.g.*, map, reduce) have gained traction as an abstraction for targeting parallel accelerators and are a promising answer to the performance portability problem. However, compiling high-level programs into efficient low-level parallel code is challenging. Current approaches start from a high-level parallel IR and proceed to emit GPU code directly in one big step. Fixed strategies are used to optimize and map parallelism exploiting properties of a particular GPU generation leading to performance portability issues.

We introduce the LIFT IR, a new data-parallel IR which encodes OpenCL-specific constructs as functional patterns. Our prior work has shown that this functional nature simplifies the exploration of optimizations and mapping of parallelism from portable high-level programs using rewrite-rules.

This paper describes how LIFT IR programs are compiled into efficient OpenCL code. This is non-trivial as many performance sensitive details such as memory allocation, array

particular implementation which is *the* key for achieving performance portability across parallel architectures.

From the compiler point of view, the semantic information associated with parallel patterns offers a unique opportunity for optimization. These abstractions make it is easier to reason about parallelism and apply optimizations without the need for complex analysis. However, designing an IR (Internal Representation) that preserves this semantic information throughout the compilation pipeline is difficult. Most existing approaches either lower the parallel primitives into loop-based code, loosing high-level semantic information, or directly produce GPU code using fixed optimization strategies. This inevitably results in missed opportunities for optimizations or performance portability issues.

In this paper, we advocate the use of a functional data-parallel IR which expresses OpenCL-specific constructs. Our functional IR is built on top of lambda-calculus and can express a whole computational kernel as a series of nested and

## Spatial: A Language and Compiler for Application Accelerators

David Koeplinger<sup>†</sup> Matthew Feldman<sup>†</sup> Raghu Prabhakar<sup>†</sup> Yaqi Zhang<sup>†</sup>  
Hadjis<sup>†</sup> Ruben Fiszels<sup>†</sup> Tian Zhao<sup>†</sup> Luigi Nardi<sup>†</sup> Ardavan Pedram<sup>†</sup>  
Christos Kozyrakis<sup>†</sup> Kunle Olukotun<sup>†</sup>  
<sup>†</sup> Stanford University, USA

increasingly important role in high-performance computing. While developing naive code is optimizing massively parallel applications requires deep understanding of the underlying developer must struggle with complex index calculations and manual memory transfers. identifies memory access patterns used in most parallel algorithms, based on Berkeley's Parthen proposes the MAPS framework, a device-level memory abstraction that facilitates GPUs, alleviating complex indexing using on-device containers and iterators. This article presentation of MAPS and shows that its performance is comparable to carefully optimized of real-world applications.

Subject Descriptors: C.1.4 [Parallel Architectures]: GPU Memory Abstraction

Keywords: Parallelism, Abstraction, Performance

Terms and Phrases: GPGPU, memory abstraction, heterogeneous computing architectures, patterns

### Format:

Levy, Amnon Barak, and Tal Ben-Nun. 2014. MAPS: Optimizing massively parallel application-level memory abstraction. ACM Trans. Architec. Code Optim. 11, 4, Article 44 (December



# Efficient data movement is hard!

**Multi-target IR: LLVM, HPVM**



**Polyhedral compilers: Pluto, Polly**

**General compilers: GCC, Clang, ICC**

Efficient data movement is hard!

Indirect accesses

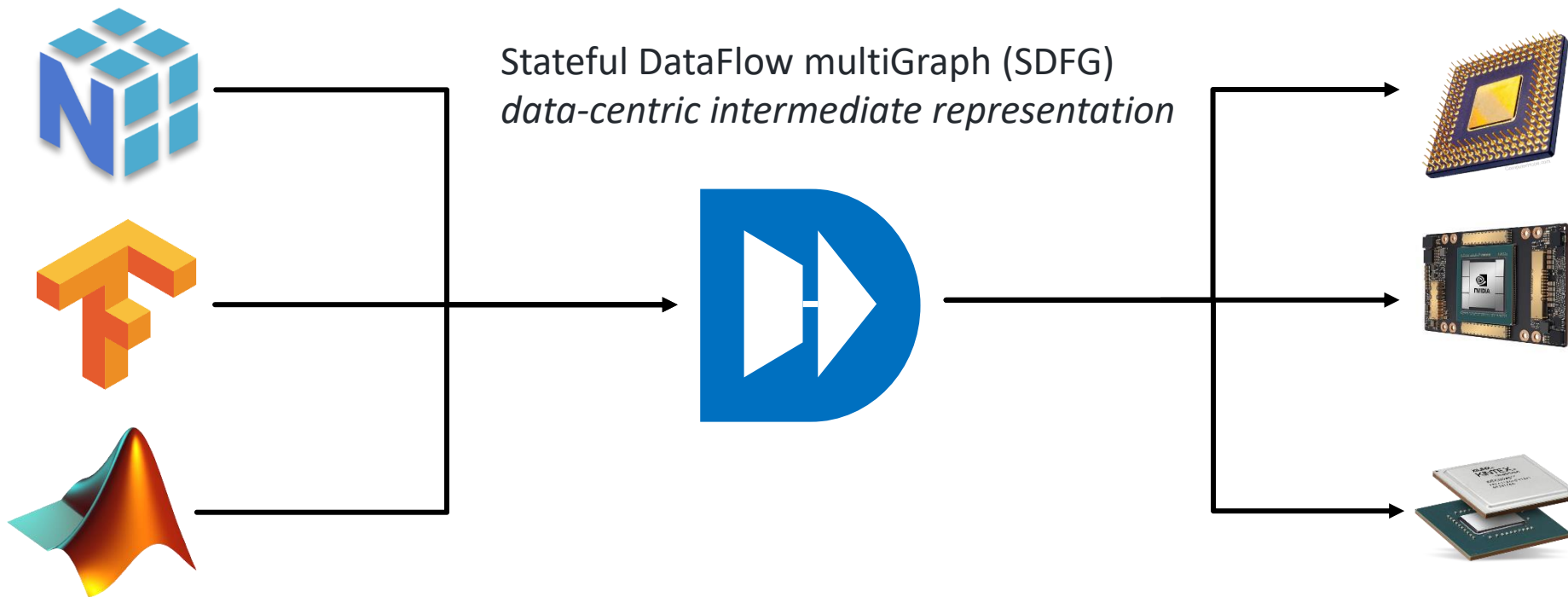
Pointers and aliasing

Difficult & expensive whole program analysis

General compilers: GCC, Clang, ICC

Polyhedral compilers: Pluto, Polly

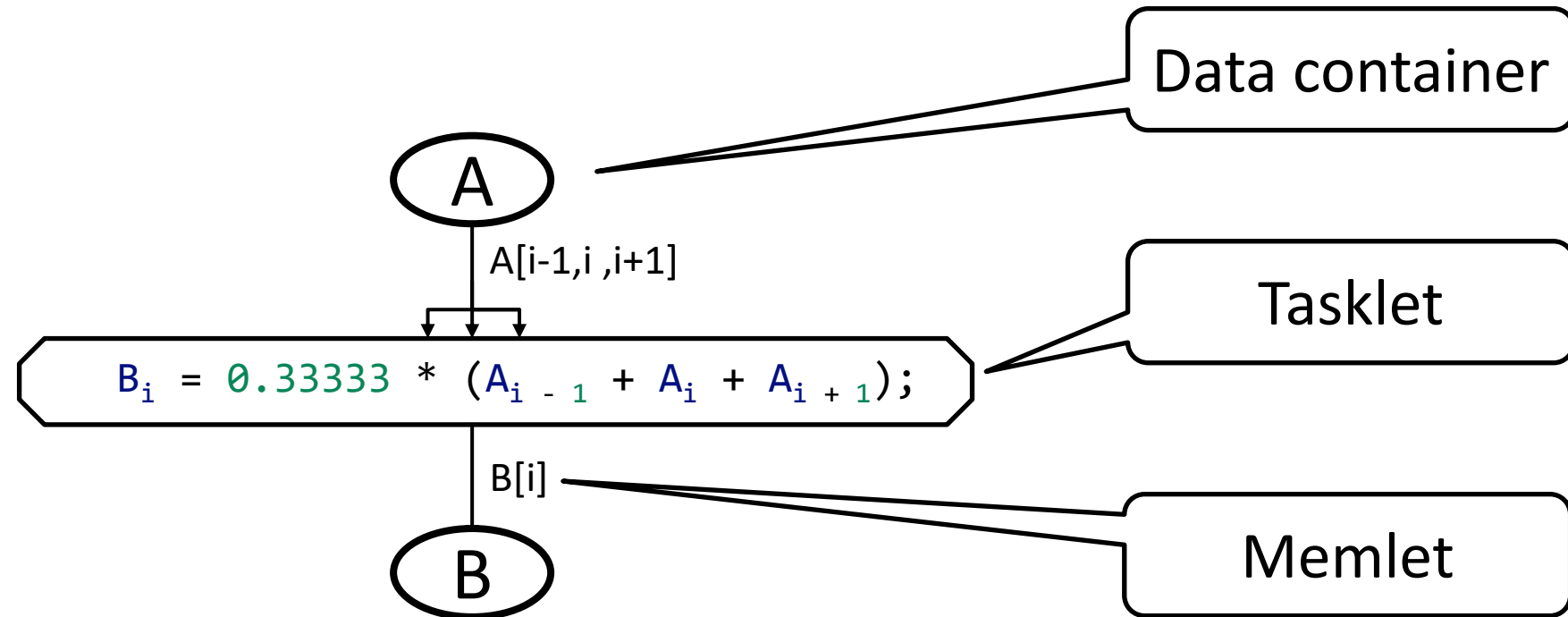
# Data Centric Parallel Programming



# What is the SDFG IR?

```
B[i] = 0.33333 * (A[i - 1] + A[i] + A[i + 1]);
```

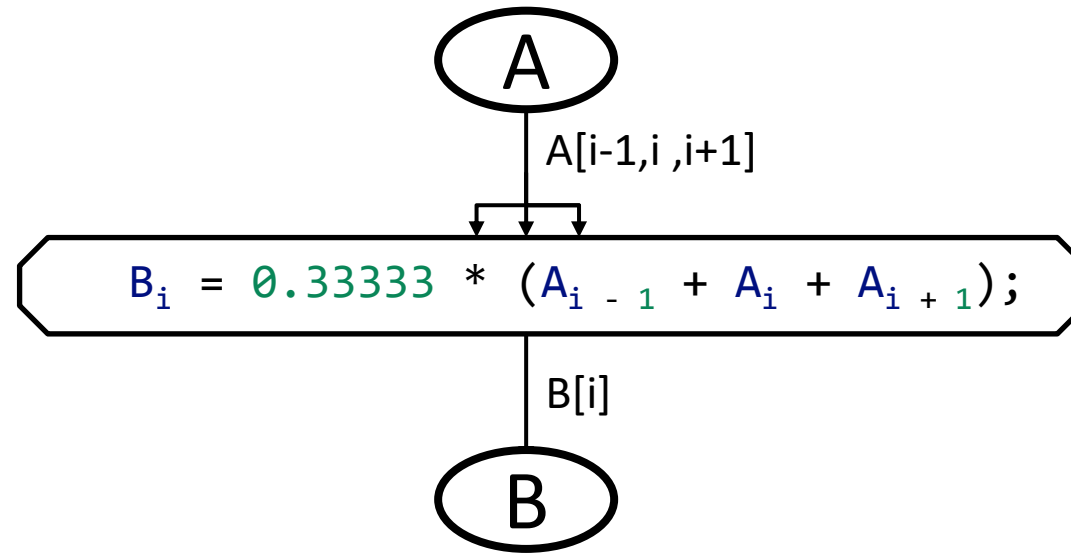
# What is the SDFG IR?





# What is the SDFG IR?

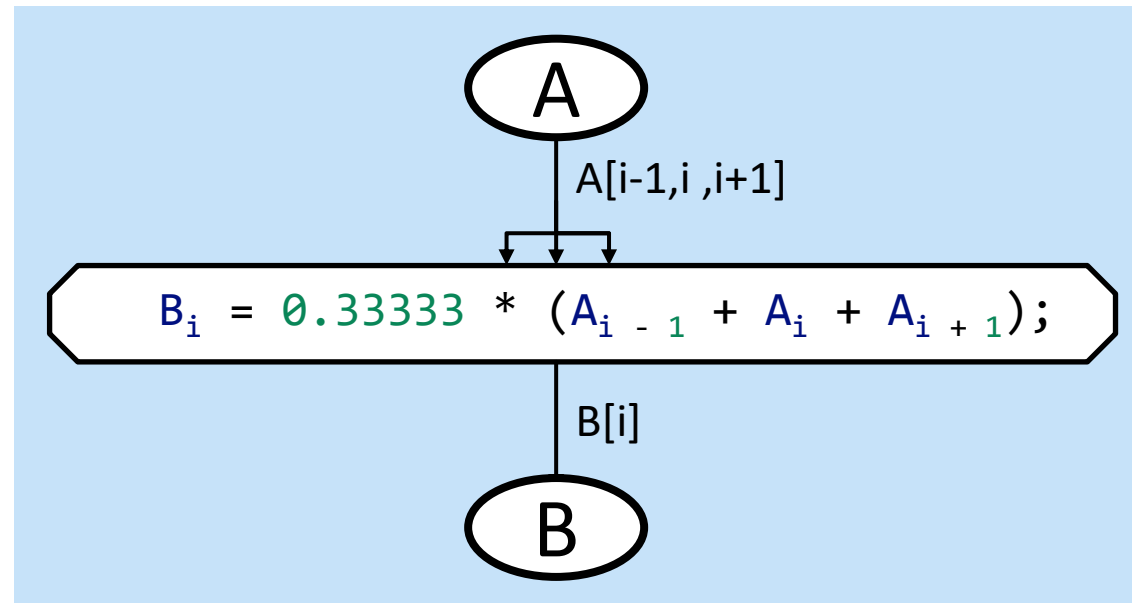
```
for (i = 1; i < n - 1; i++)
{
```



```
}
```

# What is the SDFG IR?

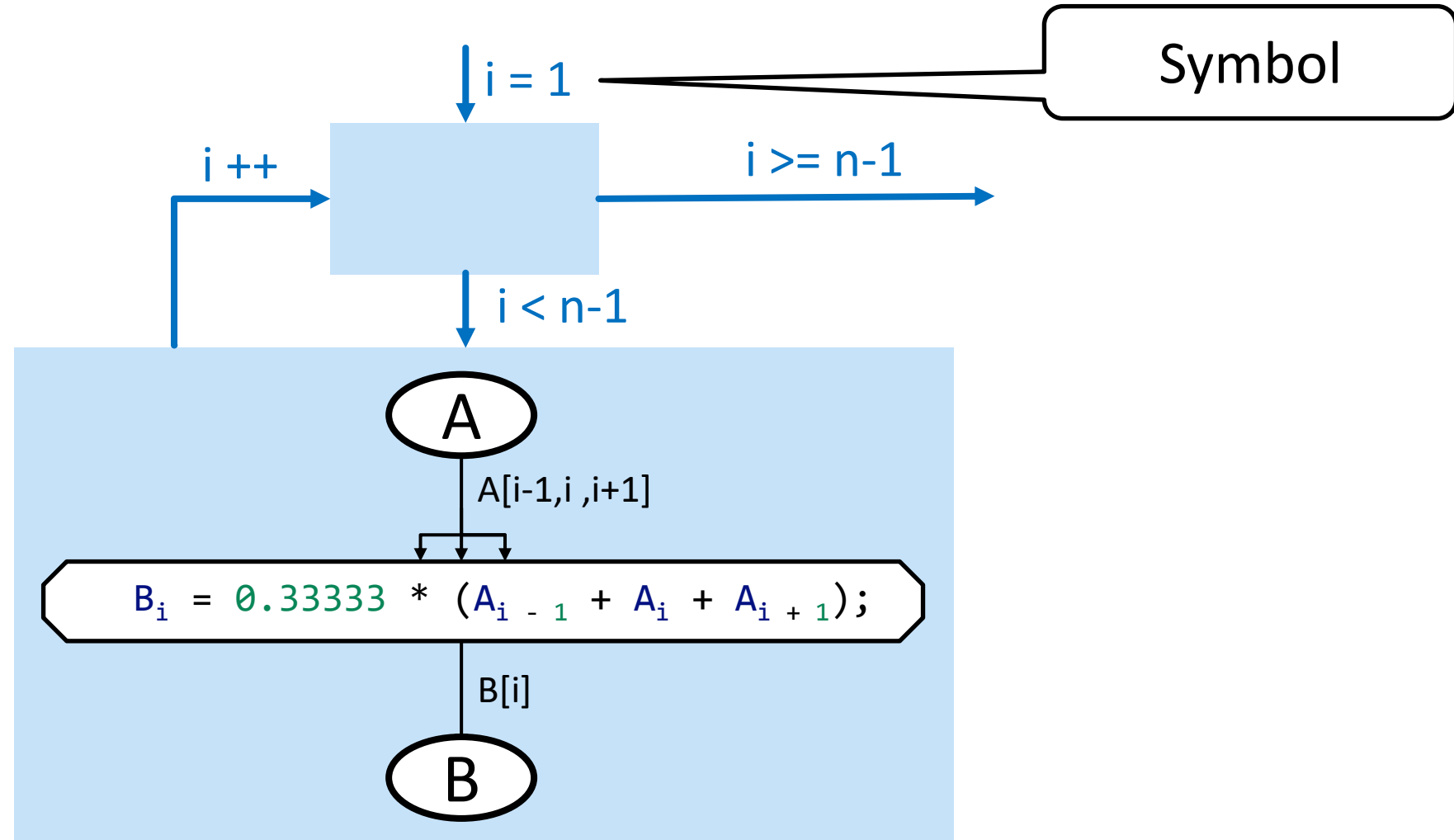
```
for (i = 1; i < n - 1; i++)  
{
```



State

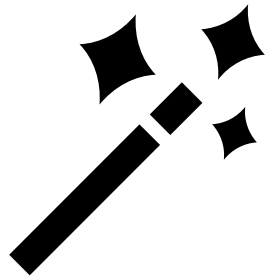
```
}
```

# What is the SDFG IR?





# Why translate C to DaCe?



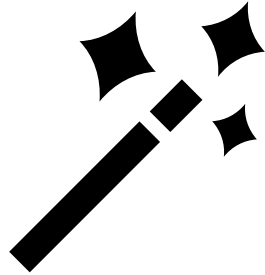
- Generate GPU code ✓
- Generate FPGA code ✓
- Improve data movement ✓
  - Apply tiling ✓
  - Reorder loops ✓

```
static void kernel_jacobi_1d(int tsteps,
                             int n,
                             double A[2000],
                             double B[2000])
{
    int t, i;

    for (t = 0; t < tsteps; t++)
    {
        for (i = 1; i < n - 1; i++)
        {
            B[i] = 0.33333 * (A[i - 1] + A[i] + A[i + 1]);
        }

        for (i = 1; i < n - 1; i++)
        {
            A[i] = 0.33333 * (B[i - 1] + B[i] + B[i + 1]);
        }
    }
}
```

# Why translate C to DaCe?



- Autoparallelization
- Generate GPU code ✓
- Generate FPGA code ✓
- Improve data movement ✓
  - Apply tiling ✓
  - Reorder loops ✓

```
static void kernel_jacobi_1d(int tsteps,
                             int n,
                             double A[2000],
                             double B[2000])
{
    int t, i;

    for (t = 0; t < tsteps; t++)
    {
        #pragma omp parallel for
        for (i = 1; i < n - 1; i++)
        {
            B[i] = 0.33333 * (A[i - 1] + A[i] + A[i + 1]);
        }
        #pragma omp parallel for
        for (i = 1; i < n - 1; i++)
        {
            A[i] = 0.33333 * (B[i - 1] + B[i] + B[i + 1]);
        }
    }
}
```

# C to DaCe: SpMV

## 1. AST Transformations

```
for (int i = 0; i < N ; i++)
```

```
    for (int j = row_ptr[i]; j < row_ptr[i+1]; j++)
```

```
        y[i] += A[j] * x[col_idx[j]];
```



# C to DaCe: SpMV

## 1. AST Transformations

- Make basic blocks explicit

```
for (int i = 0; i < N ; i++){  
    for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){  
  
        y[i] += A[j] * x[col_idx[j]];  
    }  
}
```

## C to DaCe: SpMV

### 1. AST Transformations

- Make basic blocks explicit
- Extract array indices\*
- And many others...

```
for (int i = 0; i < N ; i++){  
    for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){  
        int idx = col_idx[j];  
        y[i] += A[j] * x[idx];  
    }  
}
```

AST Transformations *canonicalize* the program representation and allows the translation to SDFG to make simplifying assumptions.

\*All array indices are extracted, including **i**, **i+1**, **j**. They are omitted here for simplicity and space

## C to DaCe: SpMV

### 1. AST Transformations

- Make basic blocks explicit
- Extract array indices\*
- And many others...

```
for (int i = 0; i < N ; i++){  
    for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){  
        int idx = col_idx[j];  
        y[i] += A[j] * x[idx];  
    }  
}
```

## Indirect accesses

AST Transformations *canonicalize* the program representation and allows the translation to SDFG to make simplifying assumptions.

\*All array indices are extracted, including  $i$ ,  $i+1$ ,  $j$ . They are omitted here for simplicity and space



## C to DaCe: SpMV

1. AST Transformations
2. Translation from C to SDFG

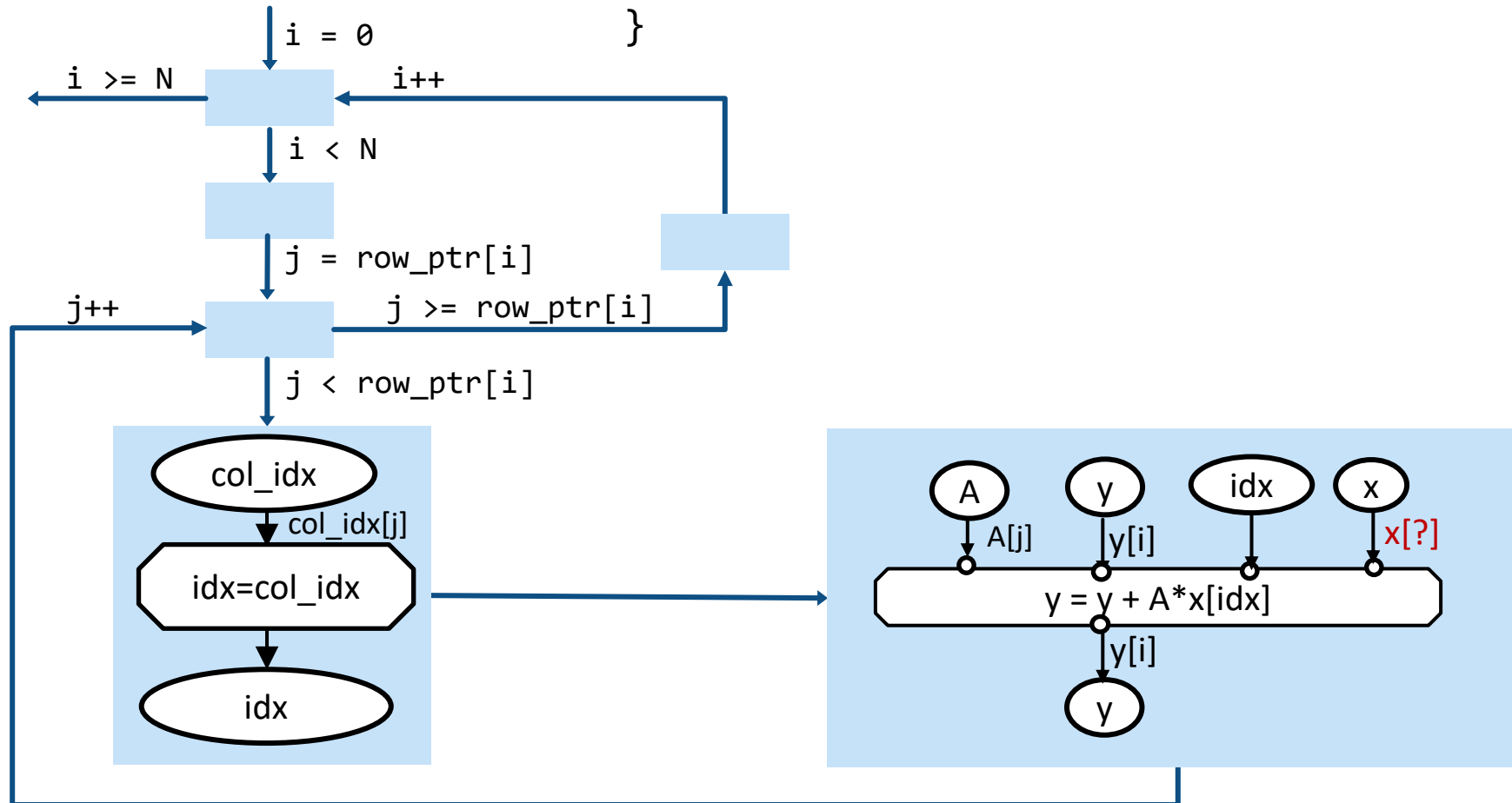
```
for (int i = 0; i < N ; i++){  
    for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){  
        int idx = col_idx[j];  
        y[i] += A[j] * x[idx];  
    }  
}
```

# C to DaCe: SpMV

1. AST Transformations
2. Translation from C to SDFG

```

for (int i = 0; i < N ; i++){
    for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){
        int idx = col_idx[j];
        y[i] += A[j] * x[idx];
    }
}
    
```

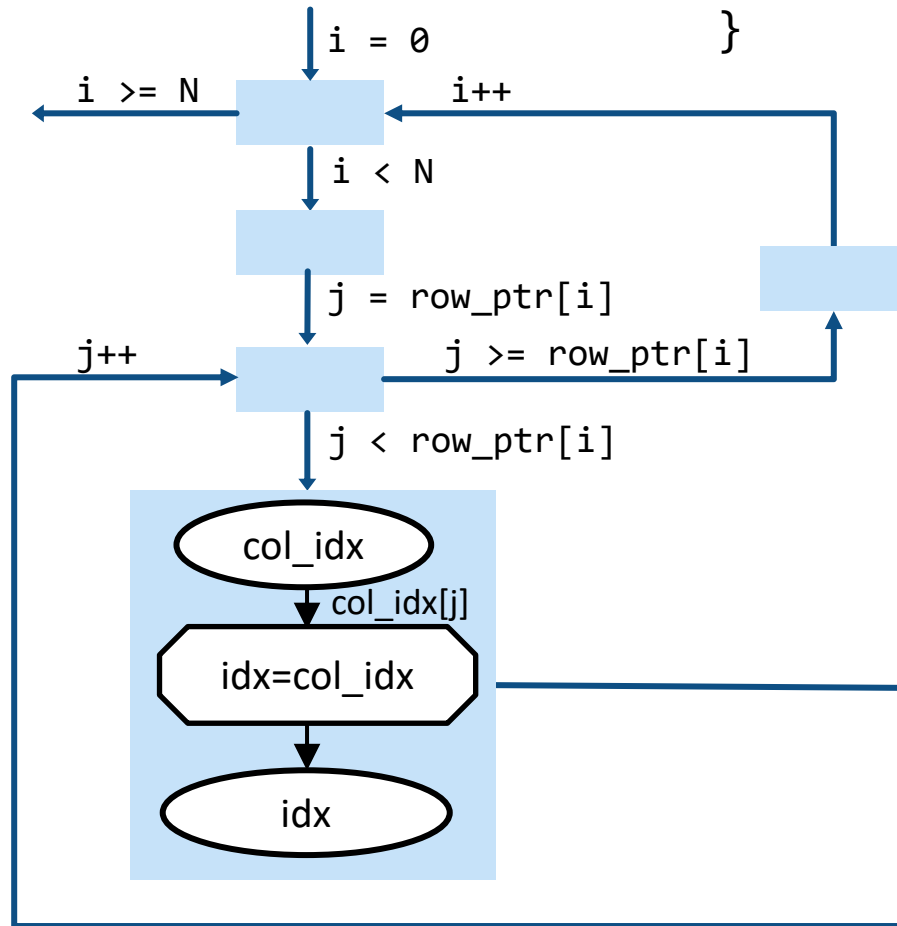


# C to DaCe: SpMV

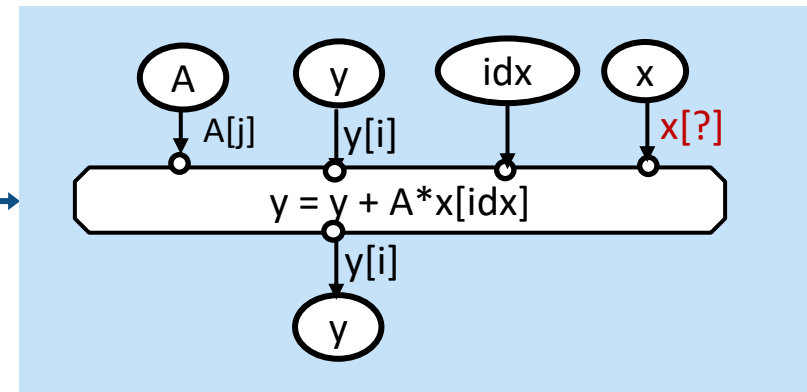
1. AST Transformations
2. Translation from C to SDFG

```

for (int i = 0; i < N ; i++){
    for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){
        int idx = col_idx[j];
        y[i] += A[j] * x[idx];
    }
}
    
```



The SDFG representation is at this point a direct representation of the *control-flow centric* C code. **Generates valid results!**



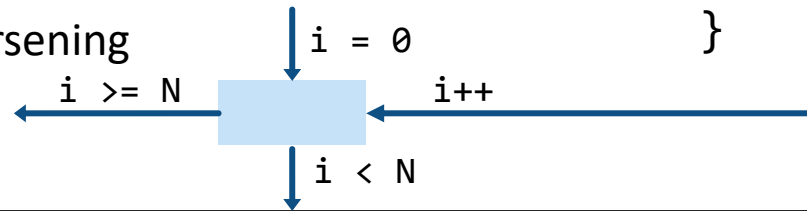


# C to DaCe: SpMV

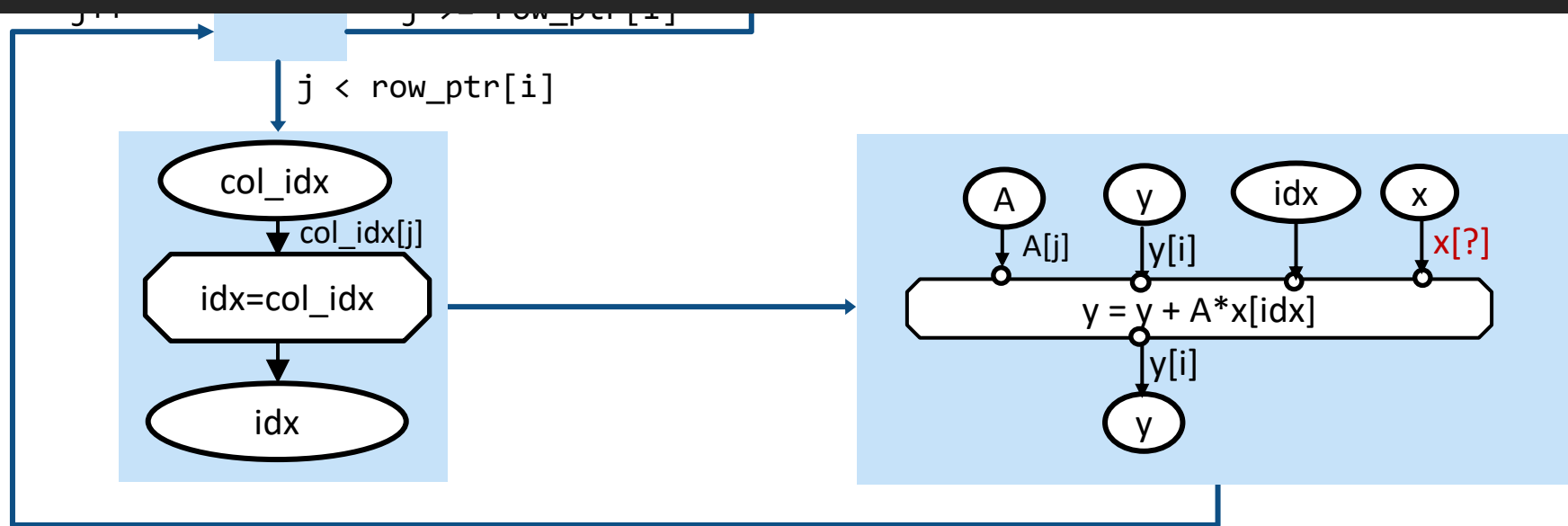
1. AST Transformations
2. Translation from C to SDFG
3. Dataflow coarsening

```

for (int i = 0; i < N ; i++){
    for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){
        int idx = col_idx[j];
        y[i] += A[j] * x[idx];
    }
}
    
```



## Indirect accesses

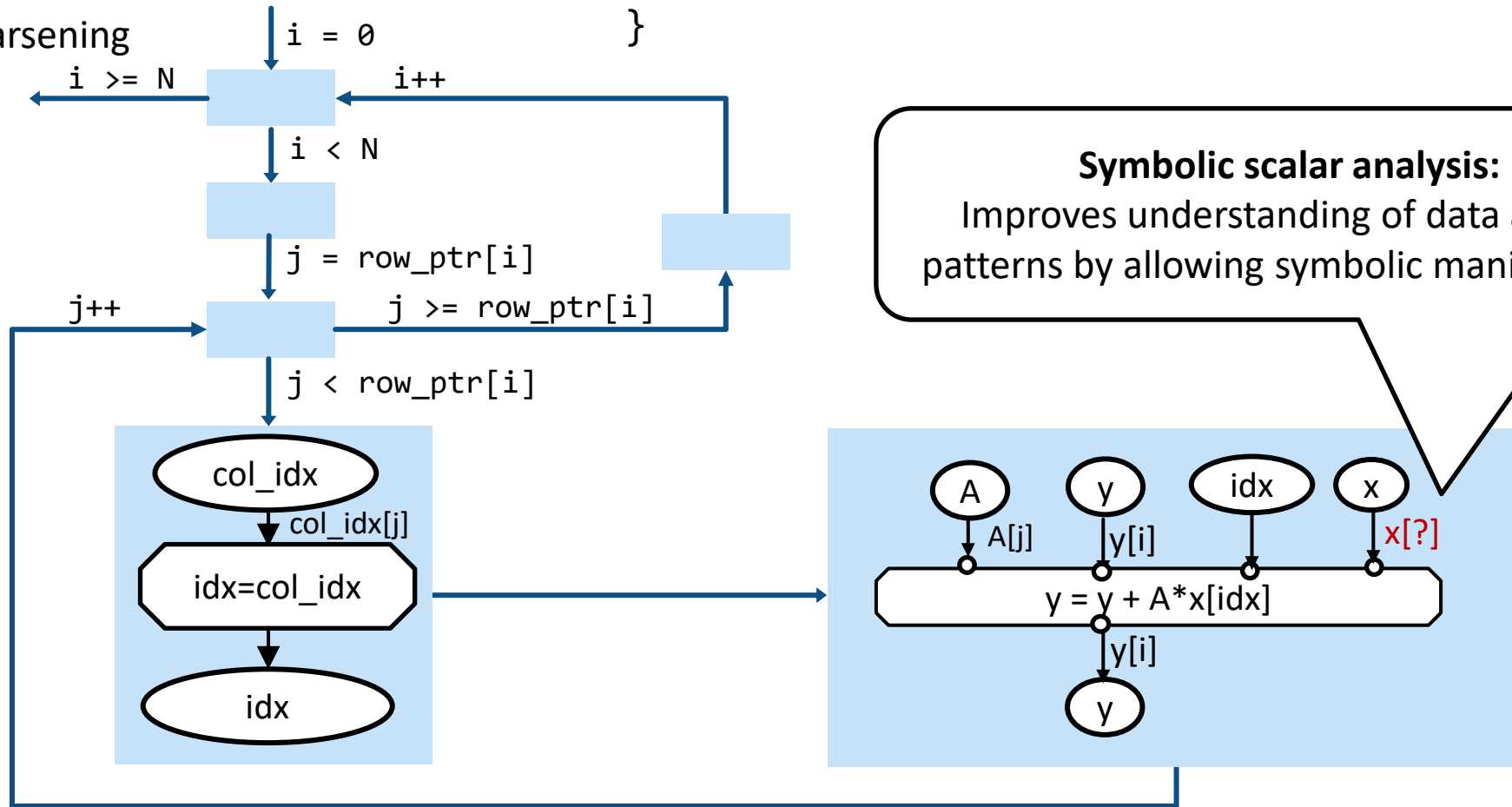


# C to DaCe: SpMV

1. AST Transformations
2. Translation from C to SDFG
3. Dataflow coarsening

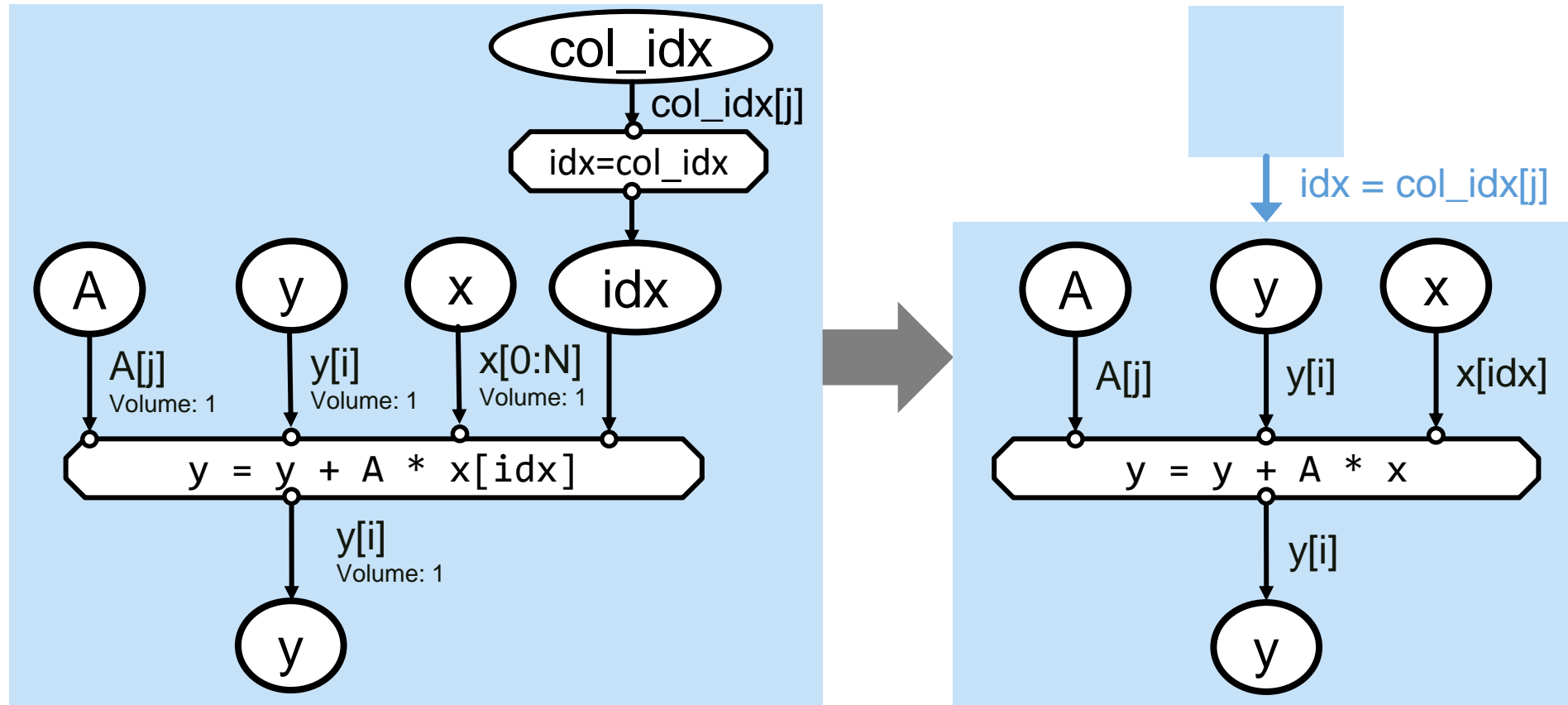
```

for (int i = 0; i < N ; i++){
    for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){
        int idx = col_idx[j];
        y[i] += A[j] * x[idx];
    }
}
    
```

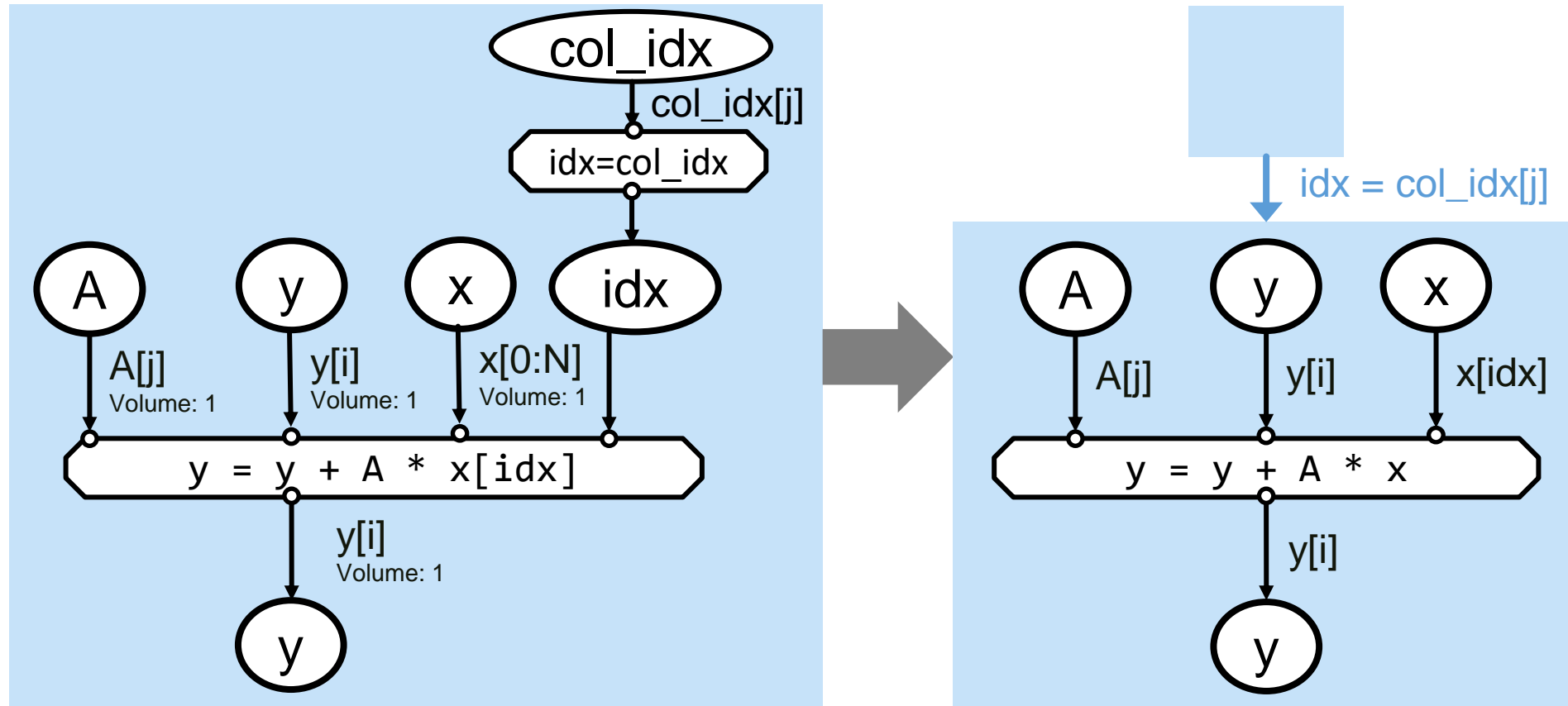


**Symbolic scalar analysis:**  
 Improves understanding of data access patterns by allowing symbolic manipulation

# Symbolic scalar analysis



# Symbolic scalar analysis



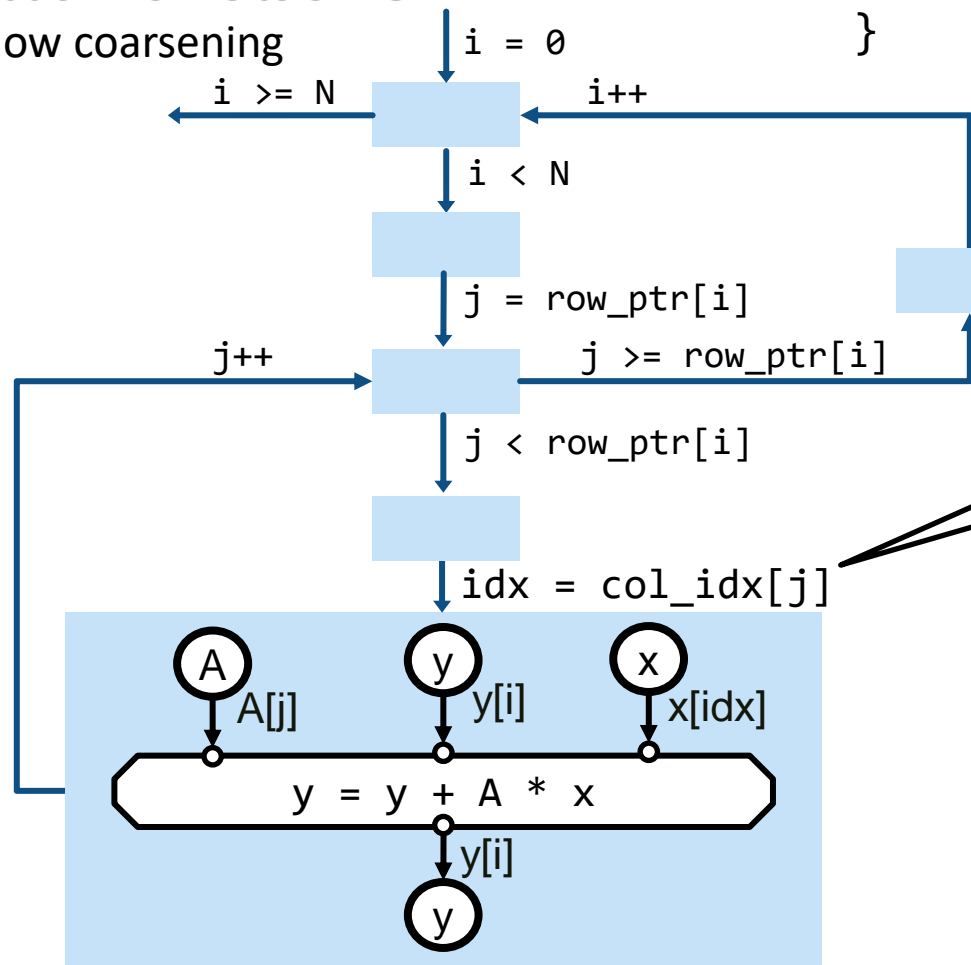
Indirect accesses

# C to DaCe: SpMV

1. AST Transformations
2. Translation from C to SDFG
3. Dataflow coarsening

```

for (int i = 0; i < N ; i++){
    for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){
        int idx = col_idx[j];
        y[i] += A[j] * x[idx];
    }
}
    
```



**Symbolic scalar analysis:**  
 Improves understanding of data access patterns by allowing symbolic manipulation

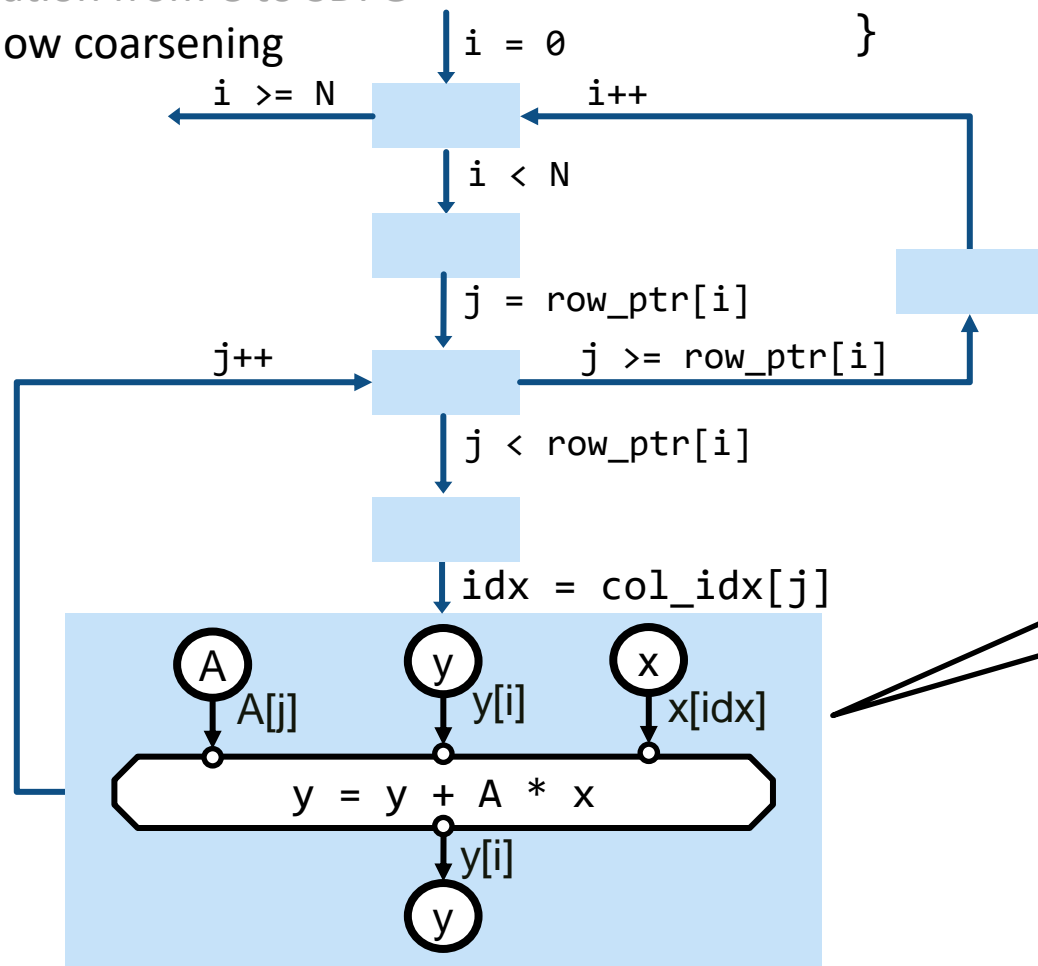


# C to DaCe: SpMV

1. AST Transformations
2. Translation from C to SDFG
3. Dataflow coarsening

```

for (int i = 0; i < N ; i++){
    for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){
        int idx = col_idx[j];
        y[i] += A[j] * x[idx];
    }
}
    
```



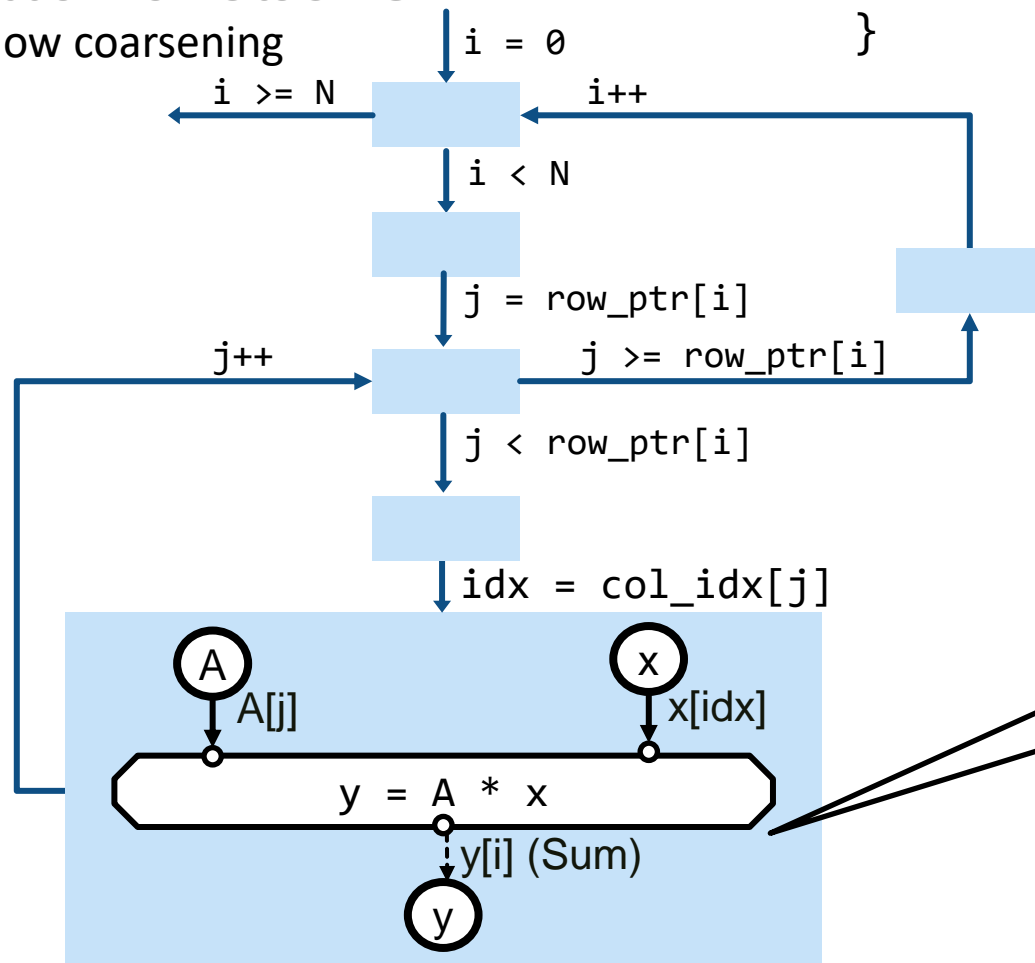
**Update detection**

# C to DaCe: SpMV

1. AST Transformations
2. Translation from C to SDFG
3. Dataflow coarsening

```

for (int i = 0; i < N ; i++){
    for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){
        int idx = col_idx[j];
        y[i] += A[j] * x[idx];
    }
}
    
```



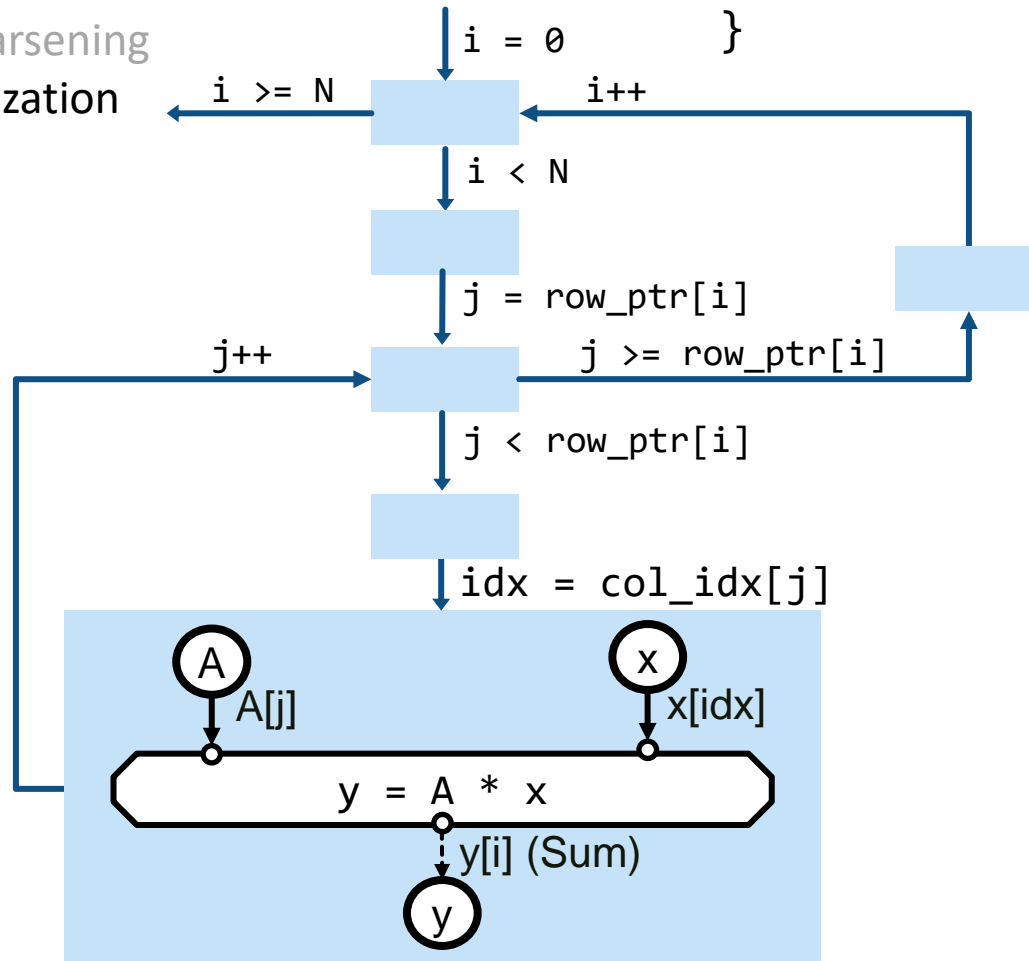
**Update detection:**  
 Allows more parallelization by creating the specialized *update* type of assignment, supporting conflict resolution for multiple operations

# C to DaCe: SpMV

1. AST Transformations
2. Translation from C to SDFG
3. Dataflow coarsening
4. Autoparallelization

```

for (int i = 0; i < N ; i++){
    for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){
        int idx = col_idx[j];
        y[i] += A[j] * x[idx];
    }
}
    
```

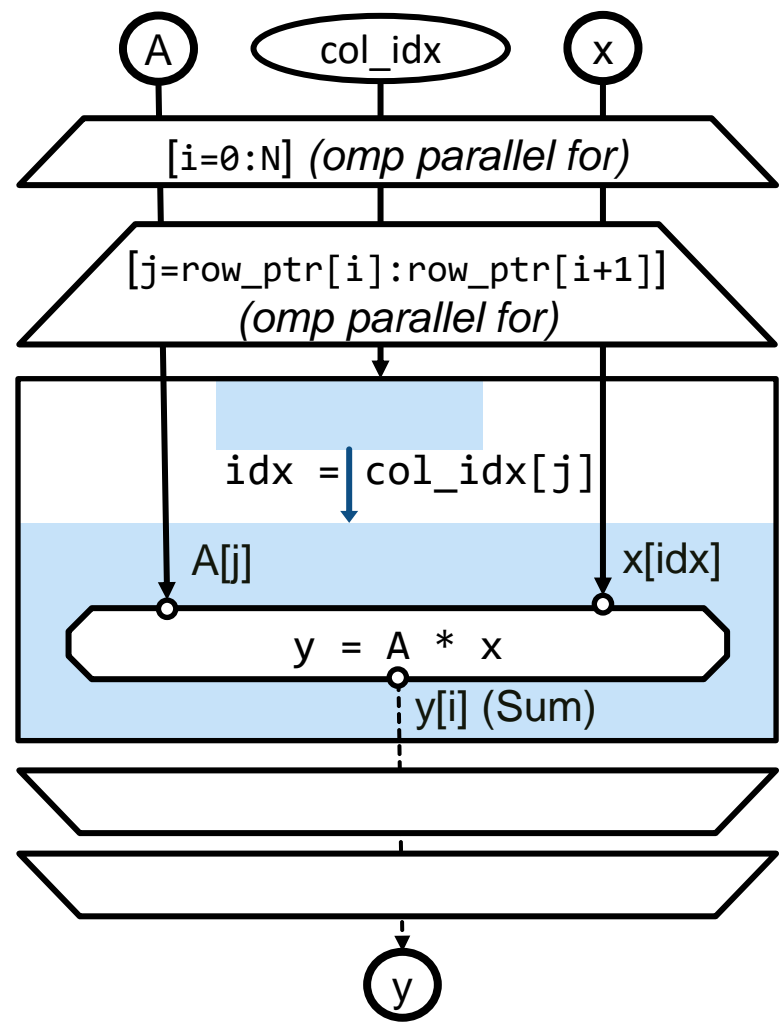
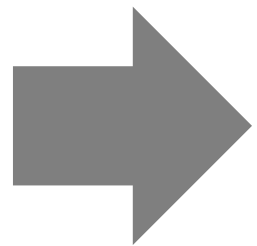
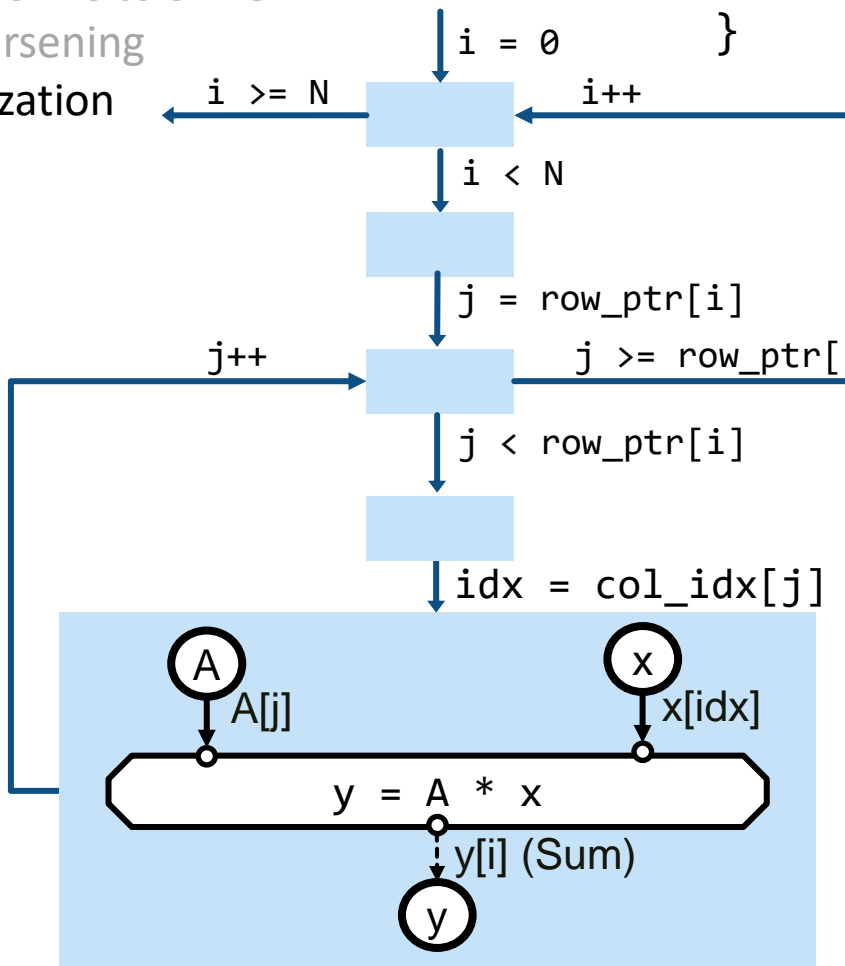


# C to DaCe: SpMV

```

for (int i = 0; i < N ; i++){
    for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){
        int idx = col_idx[j];
        y[i] += A[j] * x[idx];
    }
}
    
```

1. AST Transformations
2. Translation from C to SDFG
3. Dataflow coarsening
4. Autoparallelization

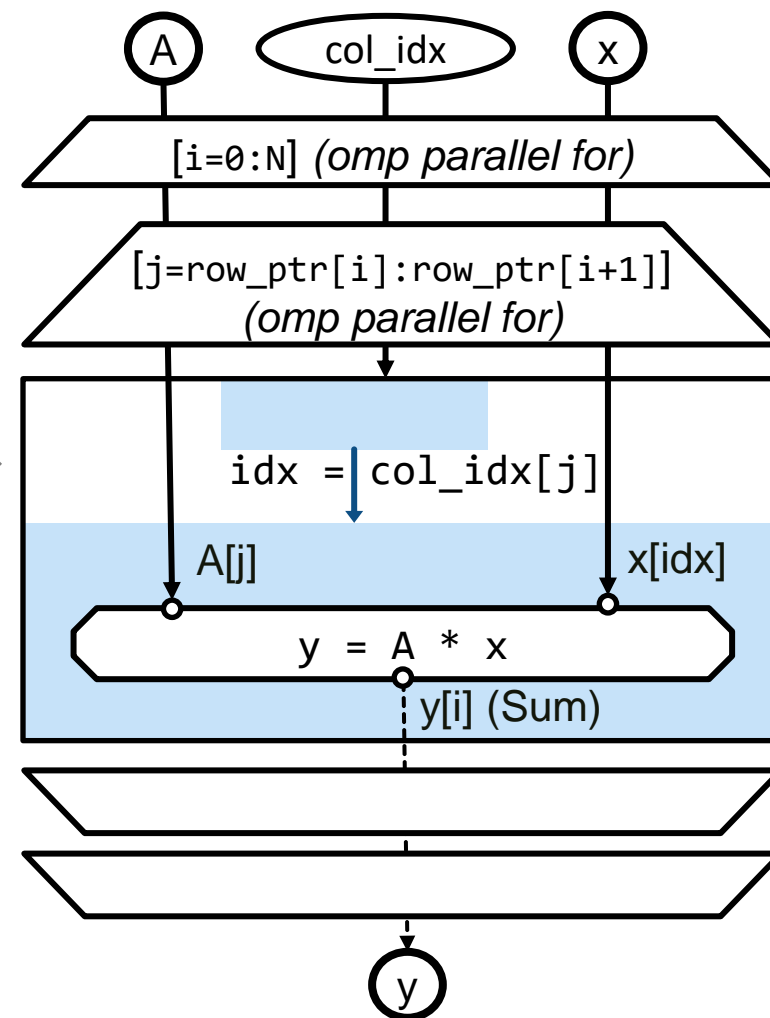
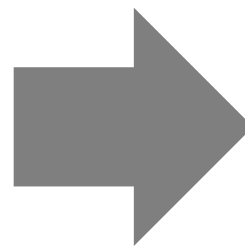


# C to DaCe: SpMV

1. AST Transformations
2. Translation from C to SDFG
3. Dataflow coarsening
4. Autoparallelization

```

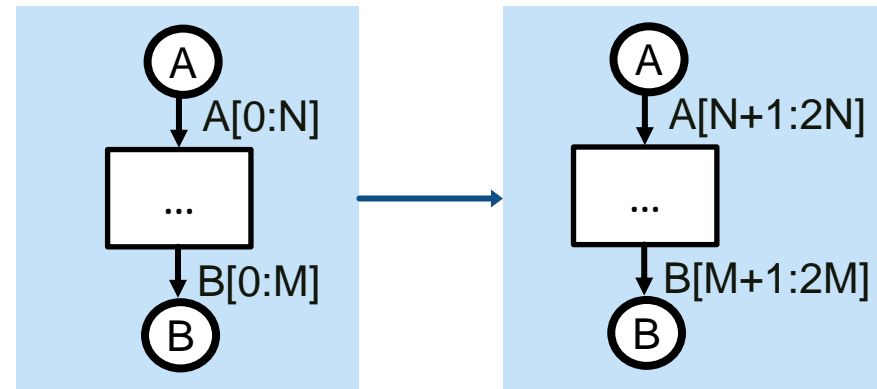
for (int i = 0; i < N ; i++){
    for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){
        int idx = col_idx[j];
        y[i] += A[j] * x[idx];
    }
}
    
```



DaCe	polly	pluto	icc - parallel
✓	✗	✗✗✗	✓

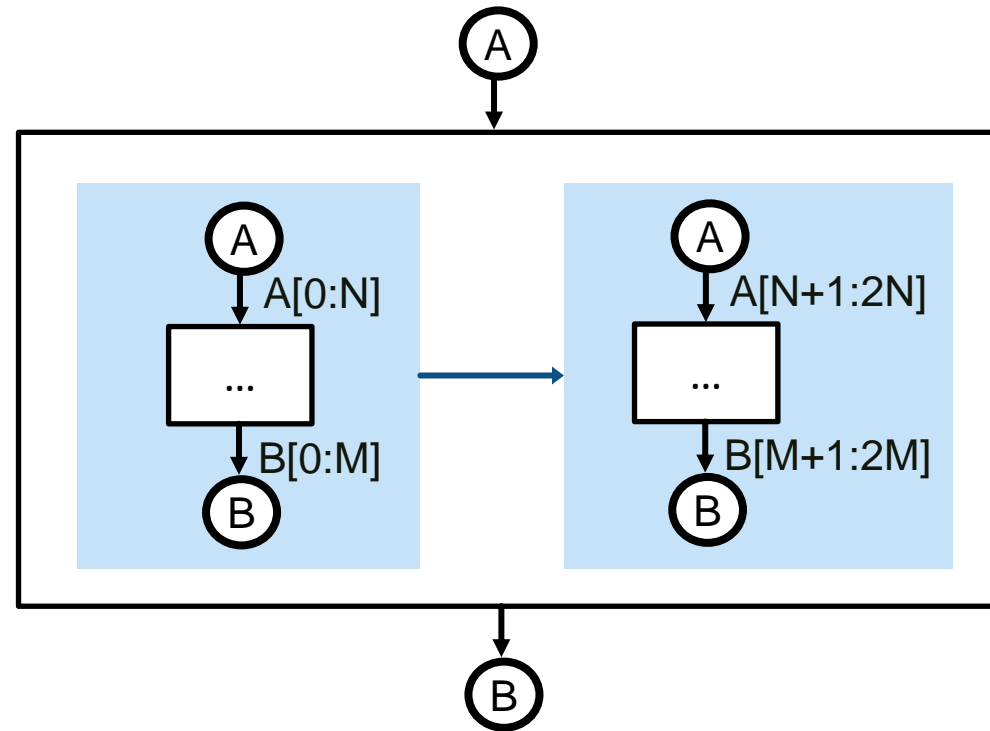


# Access pattern propagation



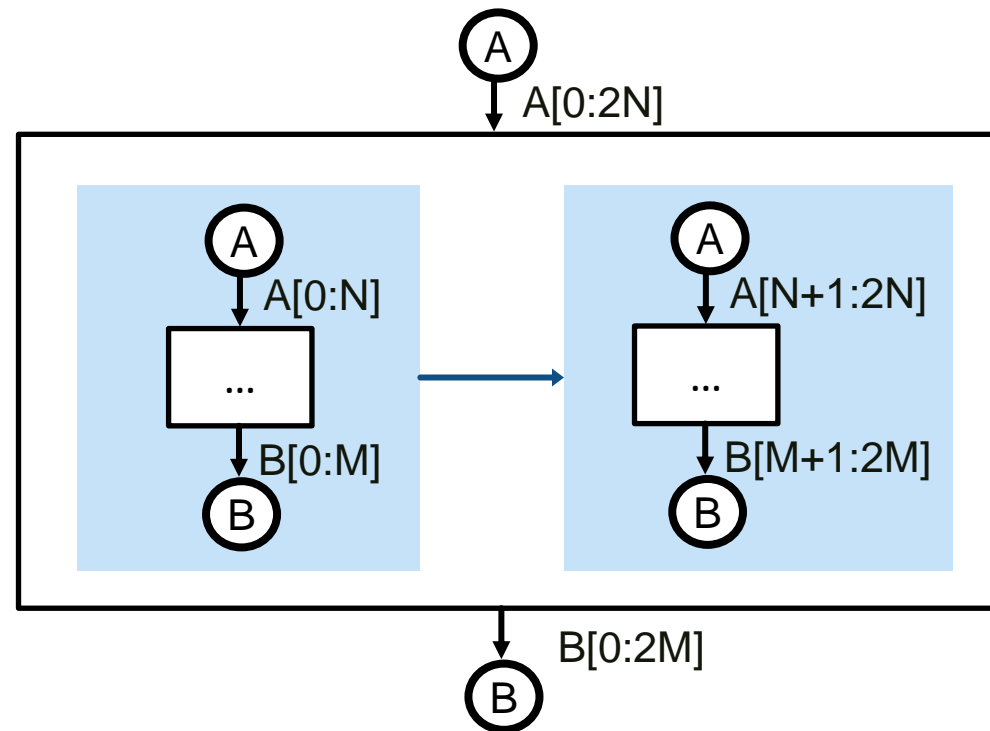
Difficult & expensive whole program analysis

# Access pattern propagation



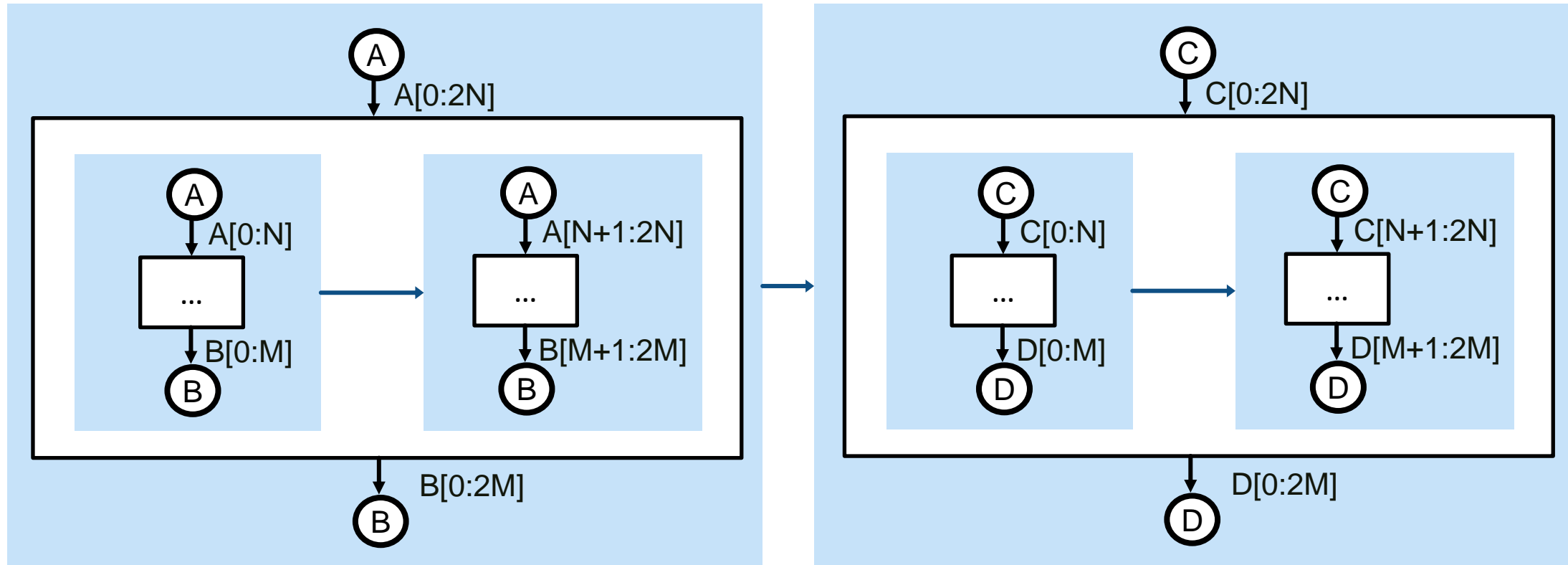
Difficult & expensive whole program analysis

# Access pattern propagation

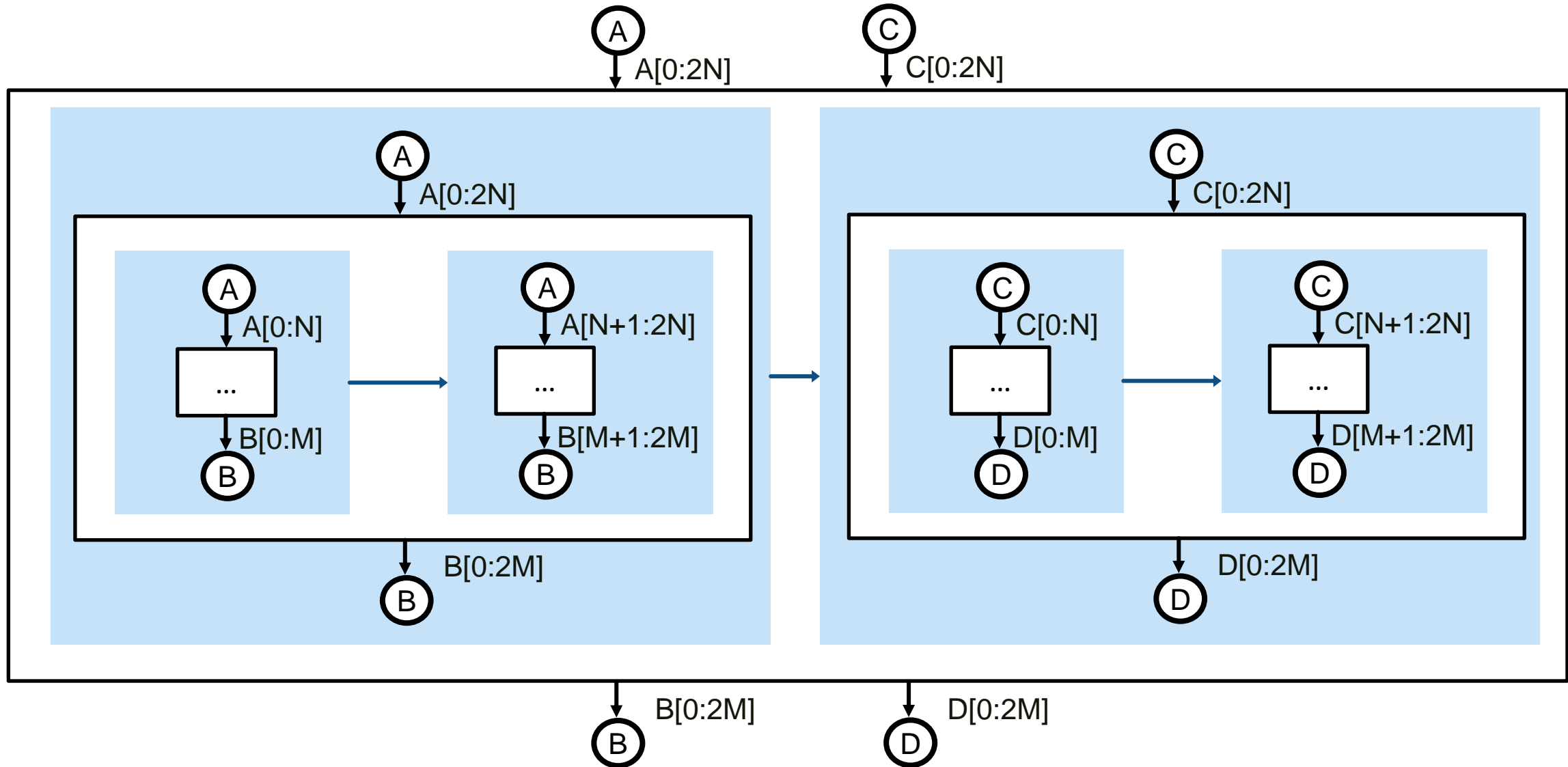


Difficult & expensive whole program analysis

# Access pattern propagation

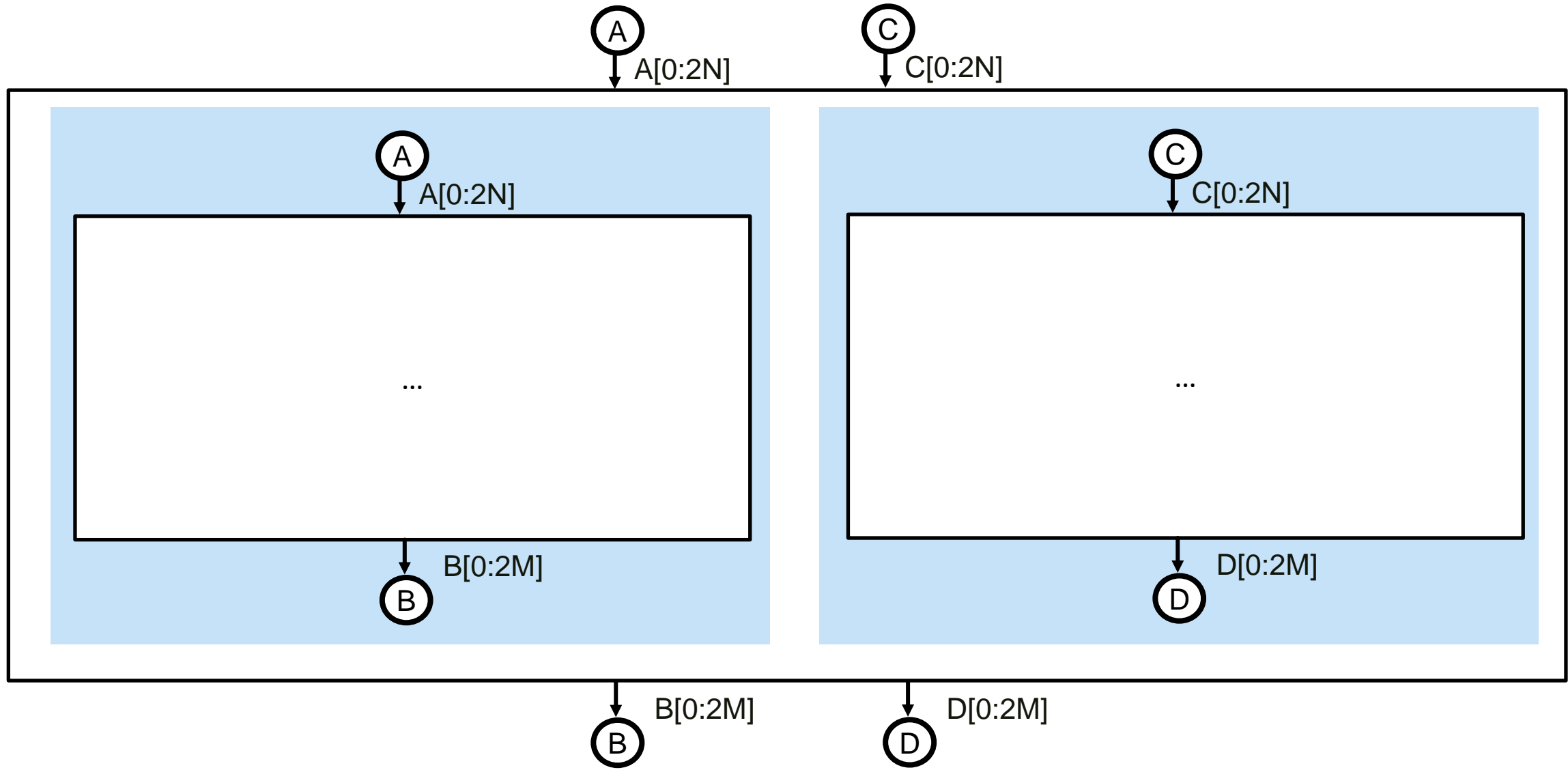


# Access pattern propagation





# Access pattern propagation



# Access pattern propagation



Difficult & expensive whole program analysis



# Access pattern propagation

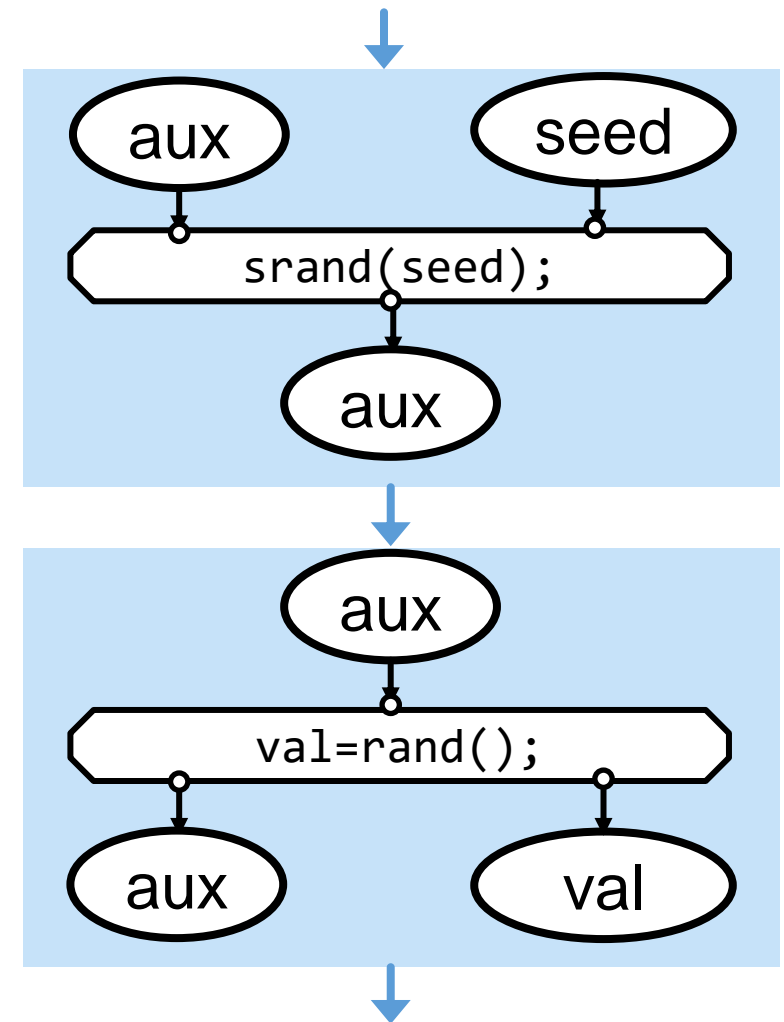


Powerful & efficient whole program analysis



# Enforcing order for data-less dependencies

- Global dictionary of states
- By default, all calls within a library should share a state
- Lists of stateless functions/libraries avoid needless complication of the SDFG



# Polybench

- 30 benchmarks
- Provided as tests for polyhedral compilers
- Small kernels (tens – hundreds LOC)

## Pointers and aliasing

No manual tuning

Full code analysis (No SCOP)

**\*No polyhedral analysis (yet)**

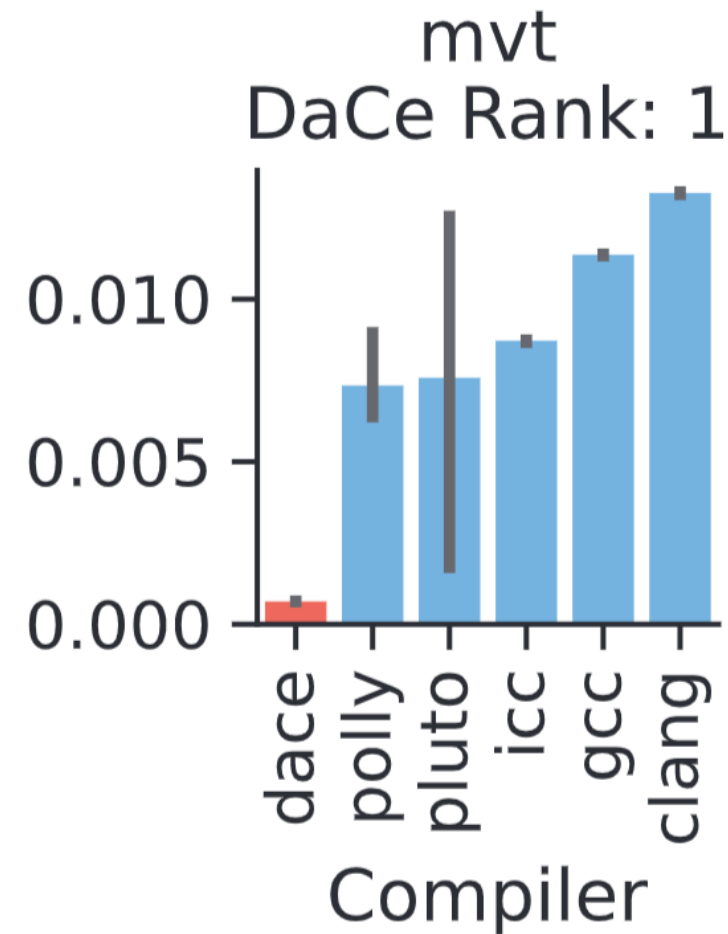
# Polybench - MVT

```

for (i = 0; i < n; i++)
{
    for (j = 0; j < n; j++)
    {
        x1[i] = x1[i] + A[i][j] * y_1[j];
    }
}
for (i = 0; i < n; i++)
{
    for (j = 0; j < n; j++)
    {
        x2[i] = x2[i] + A[j][i] * y_2[j];
    }
}

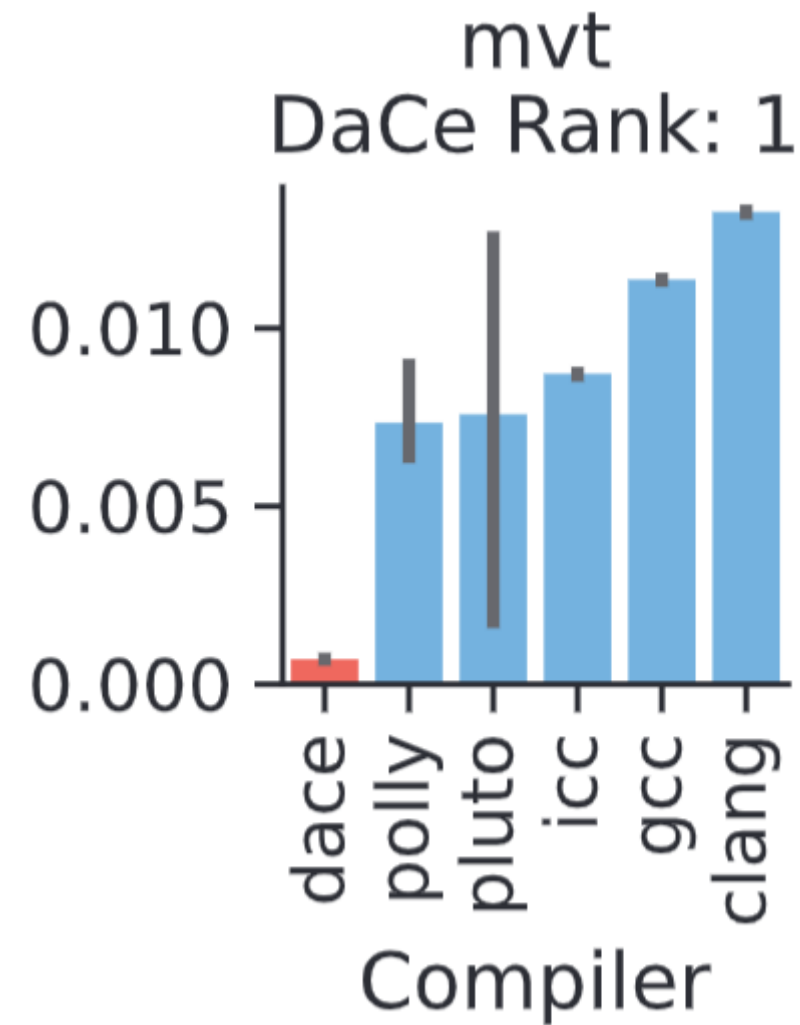
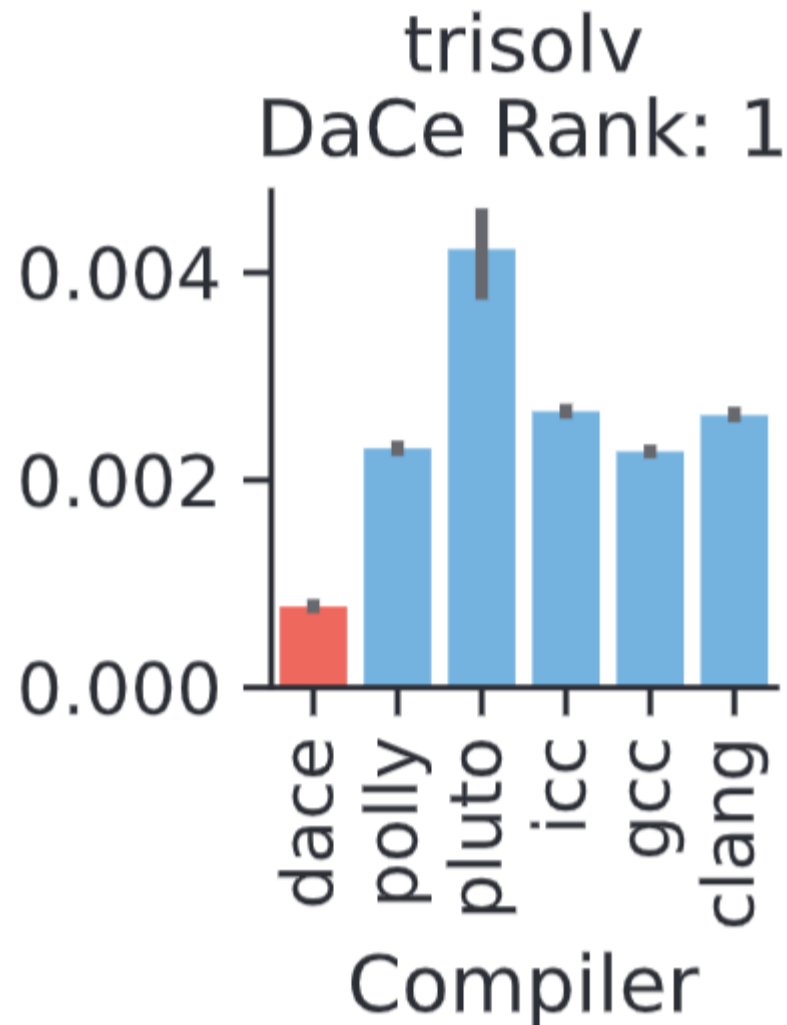
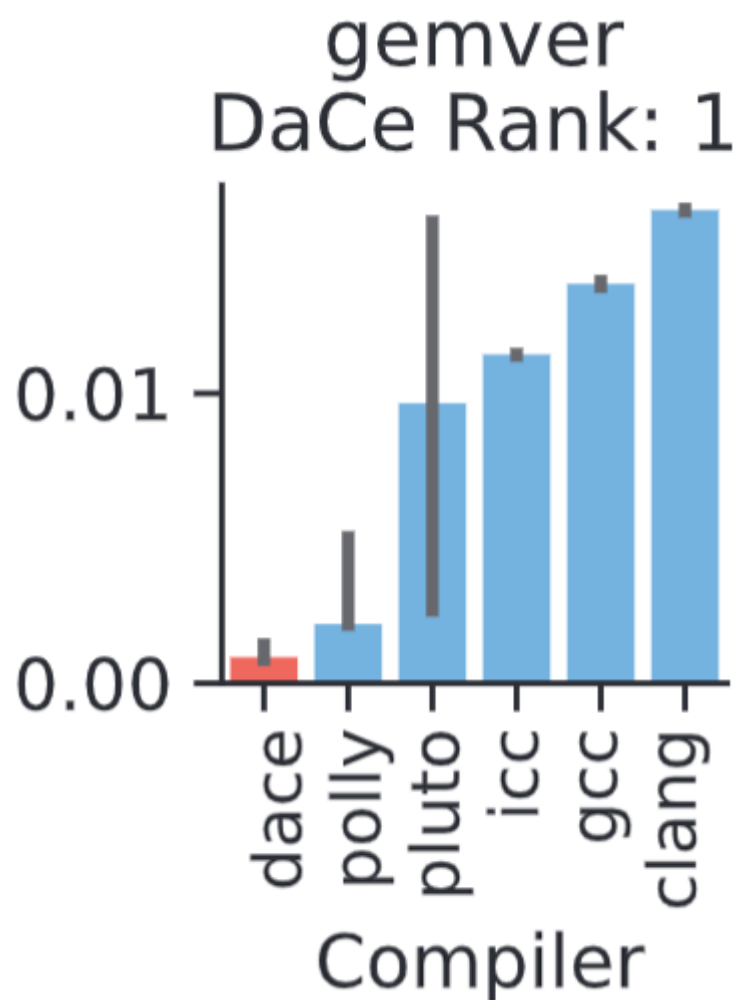
```

Update detection





# Polybench

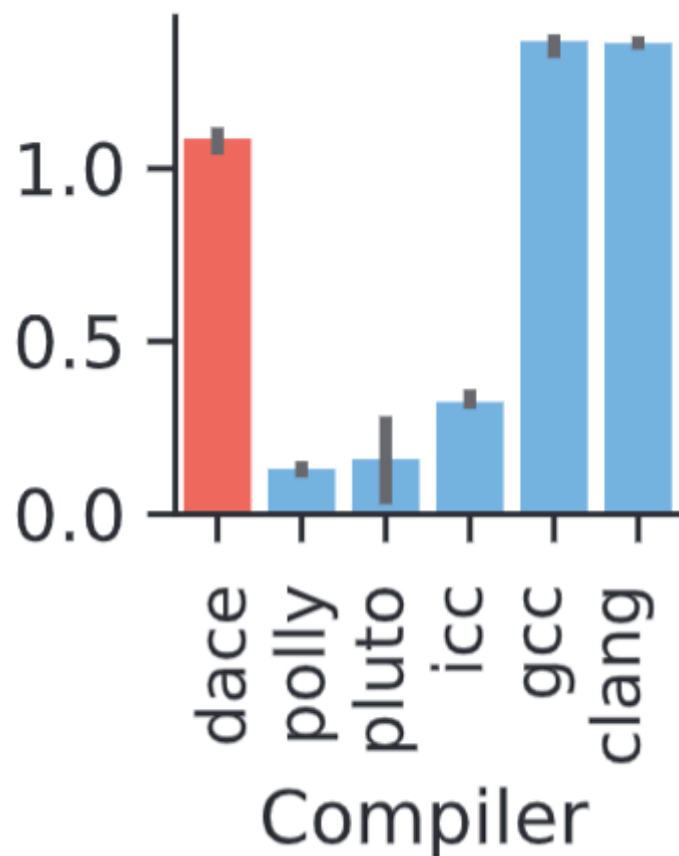


# Polybench

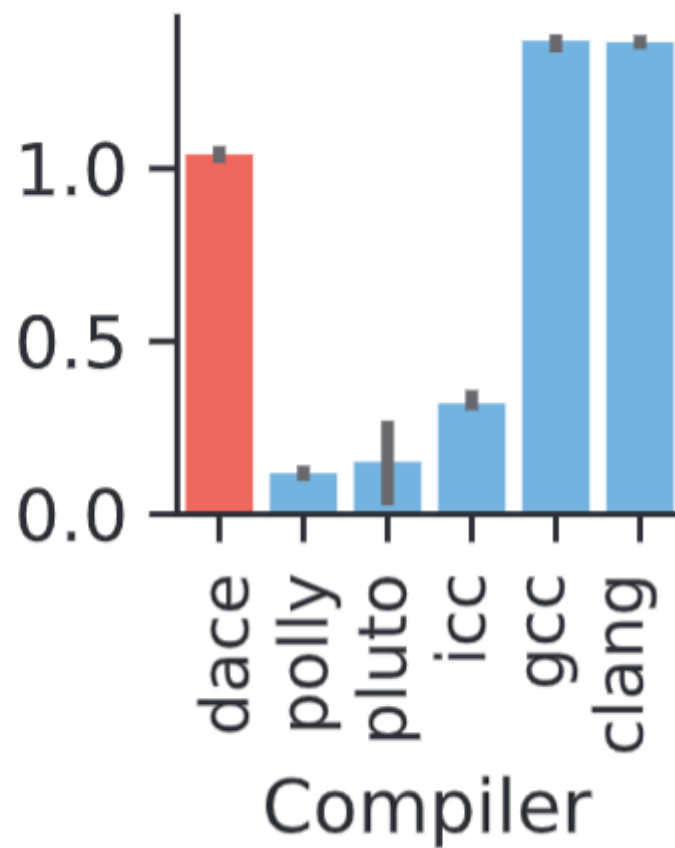
No polyhedral analysis

Auto-parallelization opportunity missed

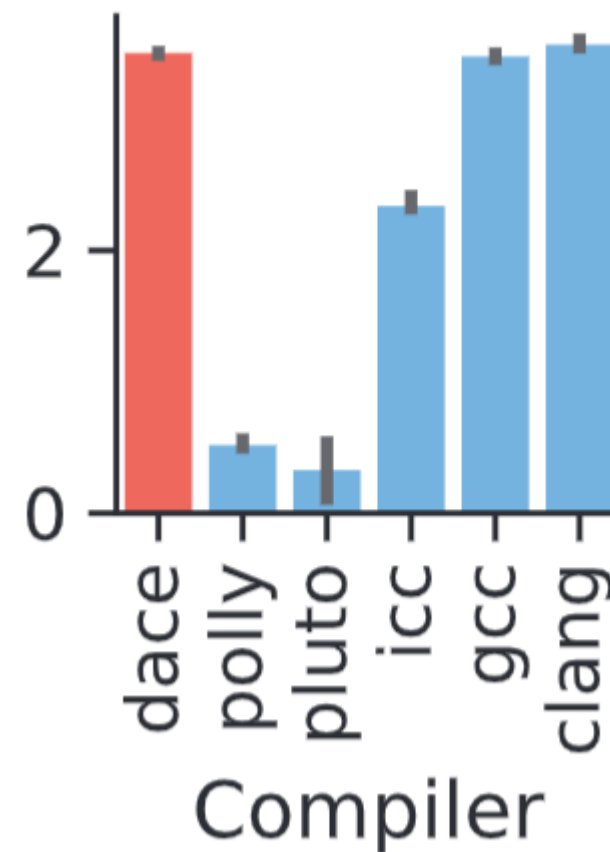
correlation  
DaCe Rank: 4



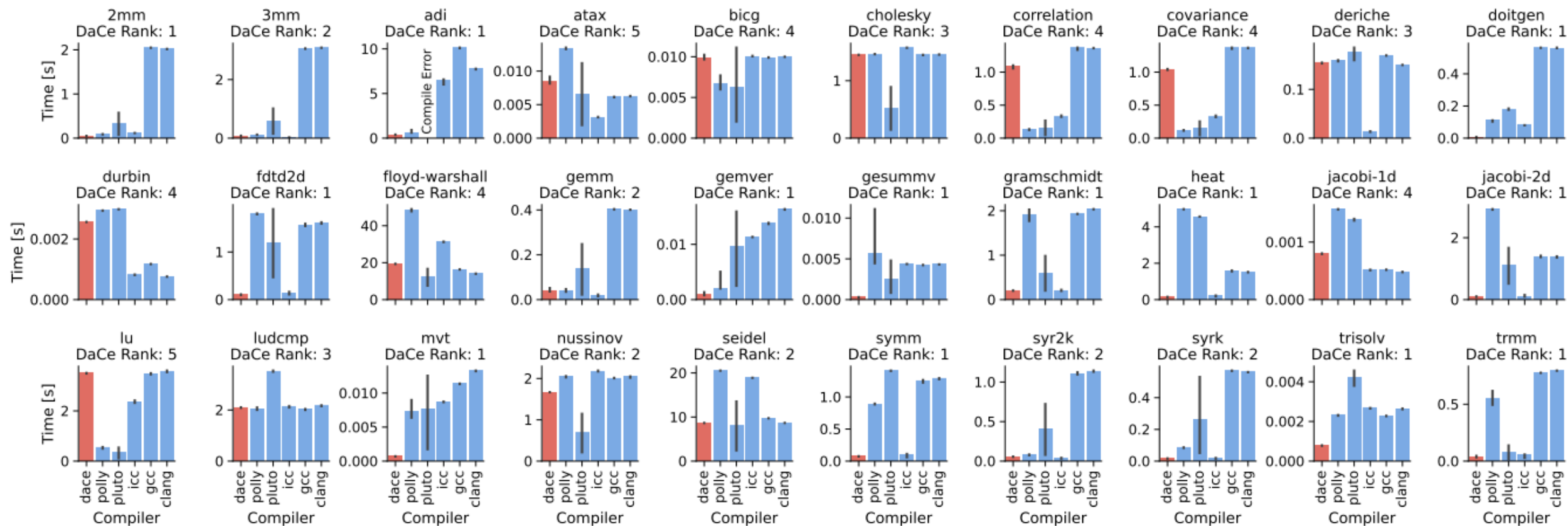
covariance  
DaCe Rank: 4



lu  
DaCe Rank: 5



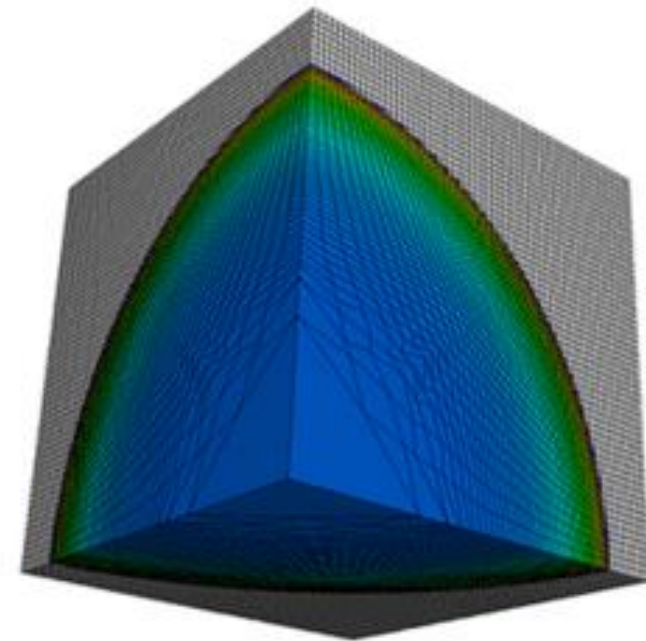
# Polybench Overview



# LULESH

- Exascale proxy app from LLNL
- *Unstructured* hydrodynamics solver

```
elemX[0] = domx[nd0i];  
elemX[1] = domx[nd1i];  
elemX[2] = domx[nd2i];  
elemX[3] = domx[nd3i];  
elemX[4] = domx[nd4i];  
elemX[5] = domx[nd5i];  
elemX[6] = domx[nd6i];  
elemX[7] = domx[nd7i];
```



Indirect accesses

# LULESH

- Developers provide manually tuned OpenMP implementation

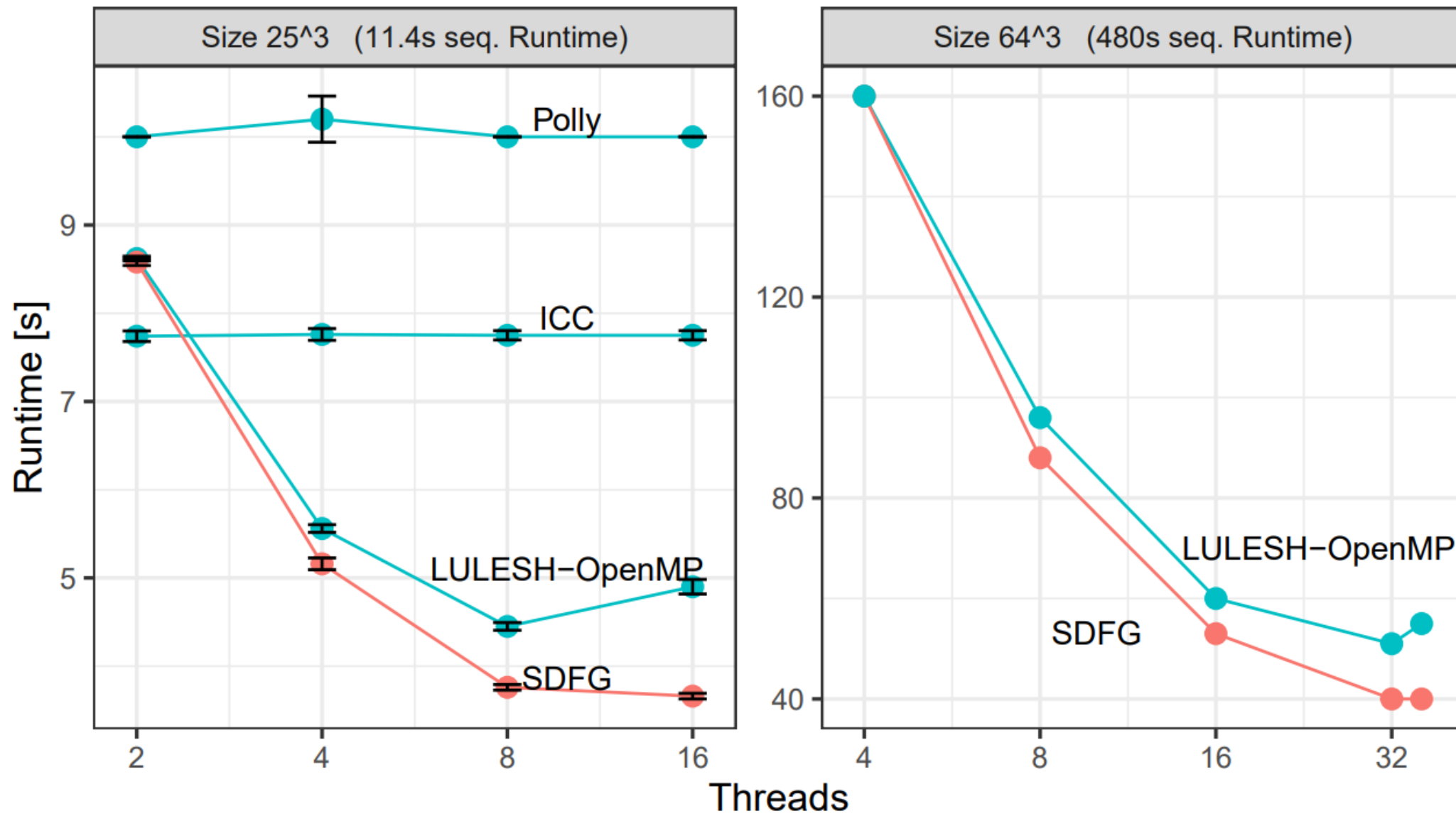
Serial result accumulation:

```
for( Index_t lnode=0 ; lnode<8 ; ++lnode ) {
    Index_t gnode = elemToNode[lnode];
    domain.fx(gnode) += fx_local[lnode];
    domain.fy(gnode) += fy_local[lnode];
    domain.fz(gnode) += fz_local[lnode];
}
```

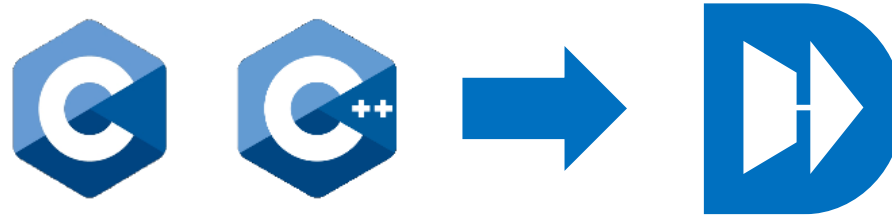
Parallel result accumulation:

```
#pragma omp parallel for firstprivate(numNode)
for( Index_t gnode=0 ; gnode<numNode ; ++gnode )
{
    Index_t count = domain.nodeElemCount(gnode) ;
    Index_t *cornerList = domain.nodeElemCornerList(gnode) ;
    Real_t fx_tmp = Real_t(0.0) ;
    Real_t fy_tmp = Real_t(0.0) ;
    Real_t fz_tmp = Real_t(0.0) ;
    for (Index_t i=0 ; i < count ; ++i) {
        Index_t ielem = cornerList[i] ;
        fx_tmp += fx_elem[ielem] ;
        fy_tmp += fy_elem[ielem] ;
        fz_tmp += fz_elem[ielem] ;
    }
    domain.fx(gnode) = fx_tmp ;
    domain.fy(gnode) = fy_tmp ;
    domain.fz(gnode) = fz_tmp ;
}
```

# LULESH



# Conclusion



Symbolic analysis is needed  
to understand data  
movement

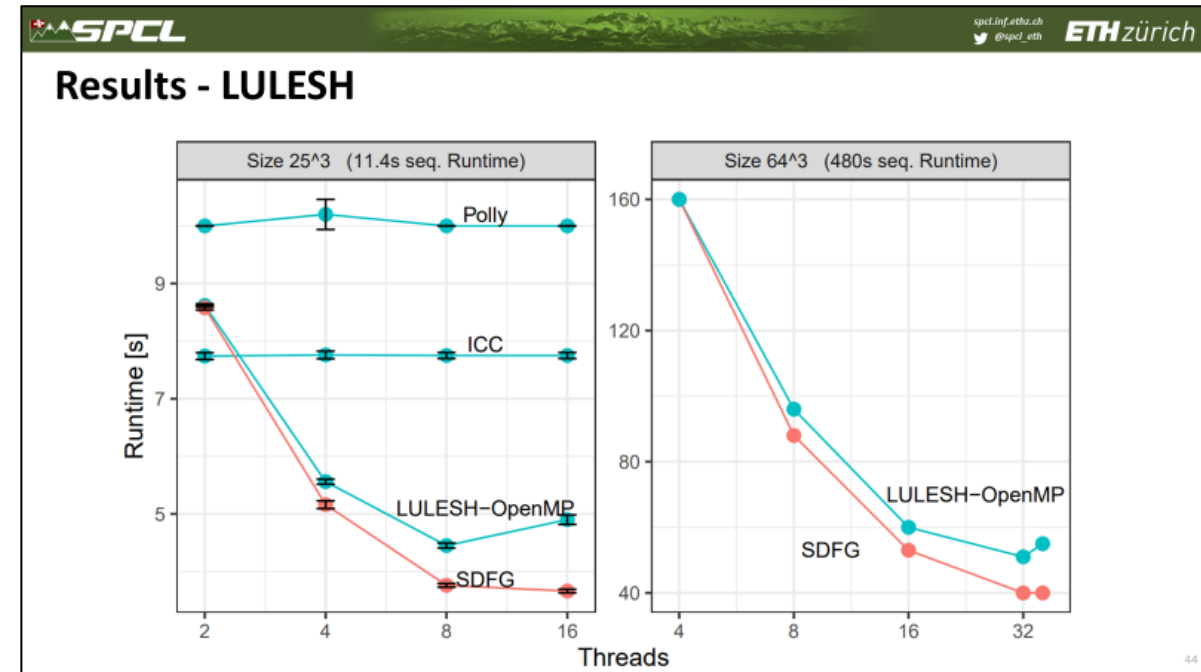
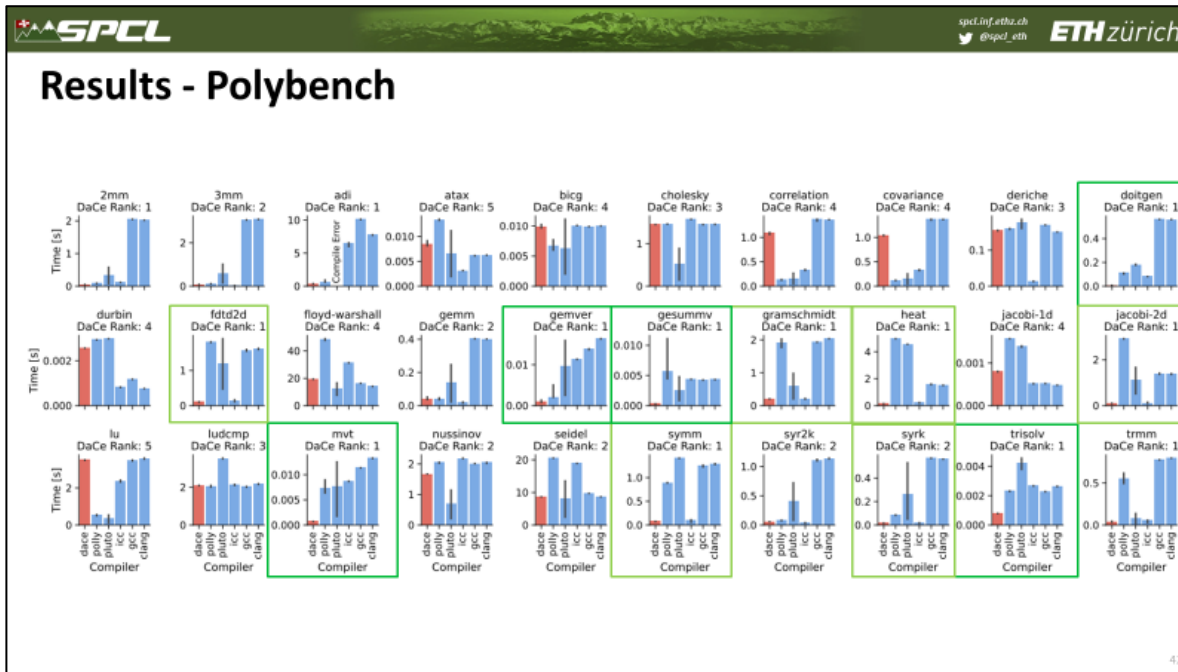
Update detection opens  
parallelization  
opportunities

Access pattern propagation  
allows efficient whole  
program analysis

**Understanding data movement is key to performance & portability**



# Thank you!



Paper: <https://arxiv.org/abs/2112.11879>

Code prototype: <https://github.com/spcl/c2dace> DaCe: <https://github.com/spcl/dace>