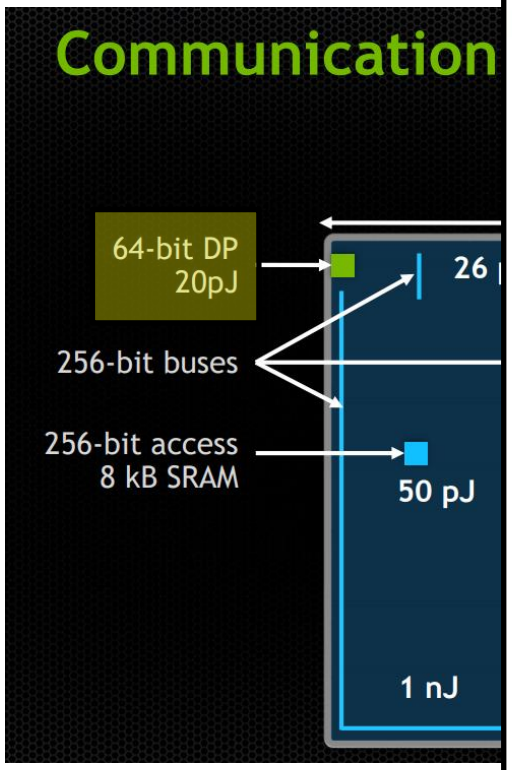
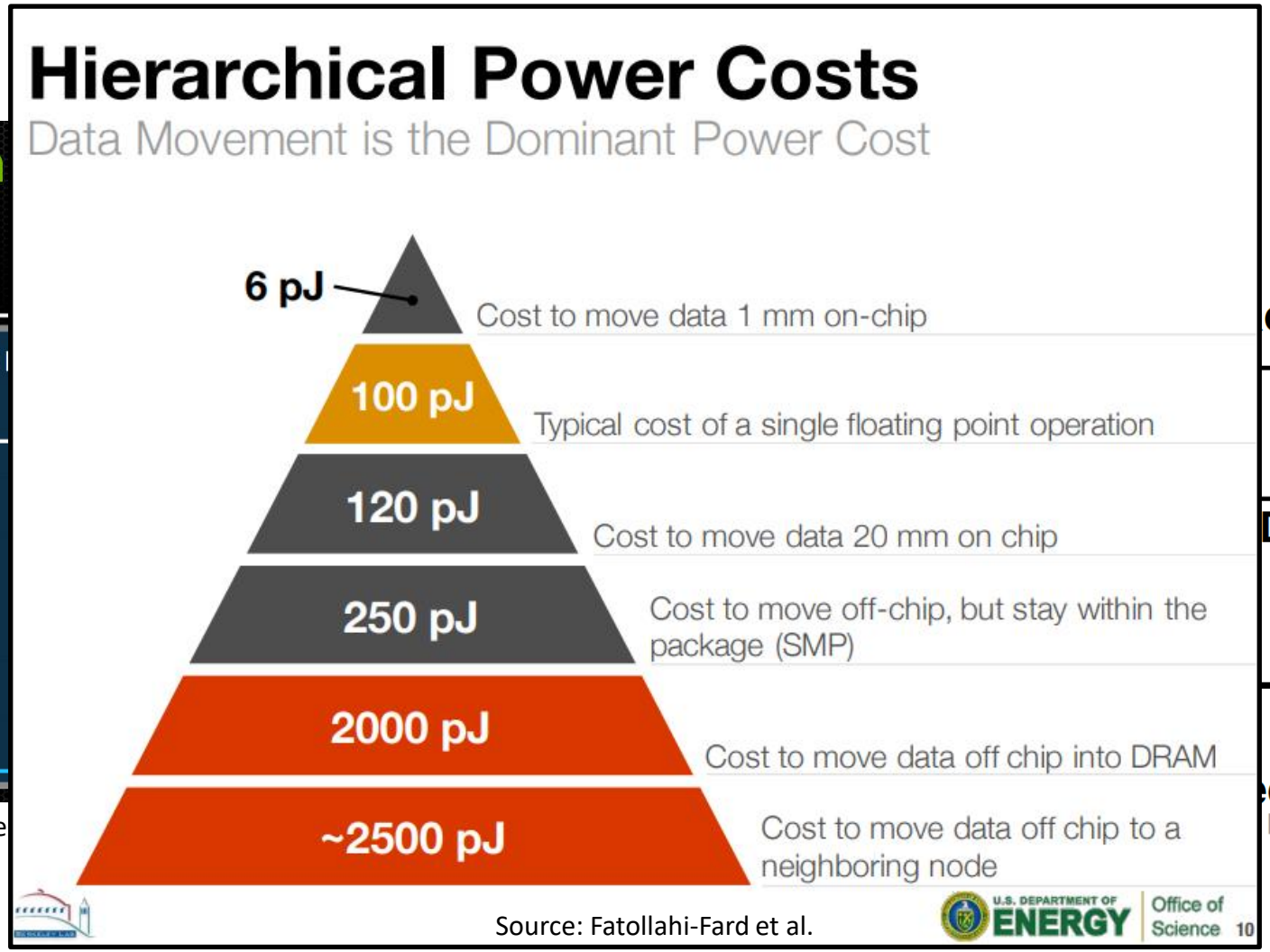
Alexandru Calotoiu, Tal Ben-Nun, Grzegorz Kwasniewski, Johannes de Fine Licht,
Timo Schneider, Philipp Schaad, Torsten Hoefler

Lifting C Semantics for Dataflow Optimization

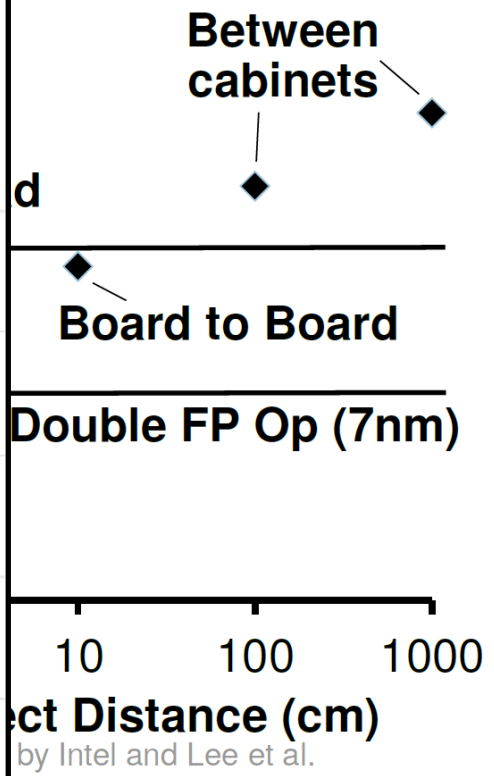
Data movement dominates energy costs!



Slide



Source: Fatollahi-Fard et al.



by Intel and Lee et al.

Efficient data movement is hard!

Halide: Decoupli
Schedules fo
I

Spatial: A Language and Compiler for Application Accelerators

id Koeplinger† Matthew Feldman† Raghu Prabhakar† Yaqi Zhang†
Hadjis† Ruben Fiszel‡ Tian Zhao† Luigi Nardi† Ardavan Pedram†
Christos Kozyrakis† Kunle Olukotun†
† Stanford University, USA

Polyhedral-Based Data Reuse Optimization for Configurable Computing

Louis-Noël Pouchet,[1] Peng
[1] University of California, Los Ange
[2] Ohio State University

MLIR: A Compiler I
Mo

Chris La
Goog

Jacqu
G

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

ERI RUBIN, ELY LEVY, AMNON BARAK, and TAL BEN-NUN,
ersity of Jerusalem

reasingly 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.
ifies memory access patterns used in most parallel algorithms, based on Berkeley's Par-
then proposes the MAPS framework, a device-level memory abstraction that facilitates
GPUs, alleviating complex indexing using on-device containers and iterators. This article
mentation of MAPS and shows that its performance is comparable to carefully optimized
of real-world applications.

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

arallelism, Abstraction, Performance

ords and Phrases: GPGPU, memory abstraction, heterogeneous computing architectures,
tterns

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

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

Michel Steuwer Toomas Remmelg Christophe Dubach
University of Edinburgh, United Kingdom
{michel.steuwer, toomas.remmelg, 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.

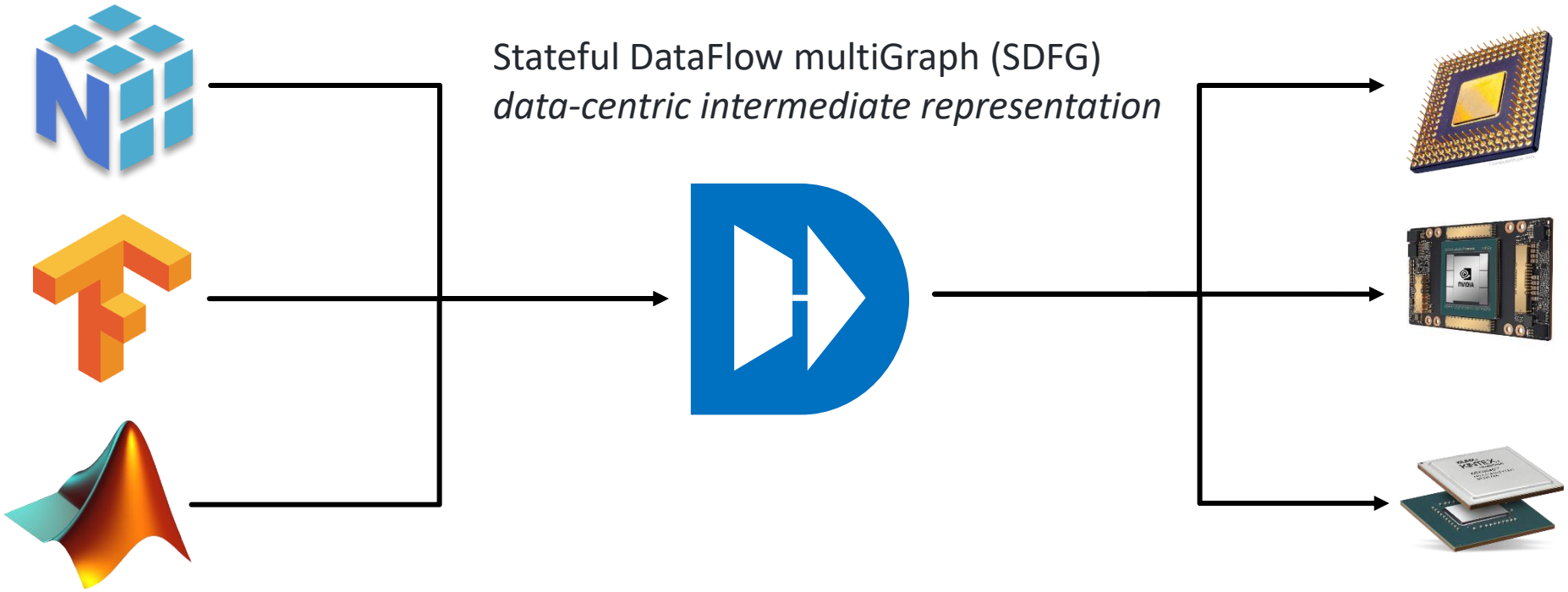
This paper describes how LIFT IR programs are compiled into efficient OpenCL code. This is non-trivial as many performance sensitive details such as memory allocation, array

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

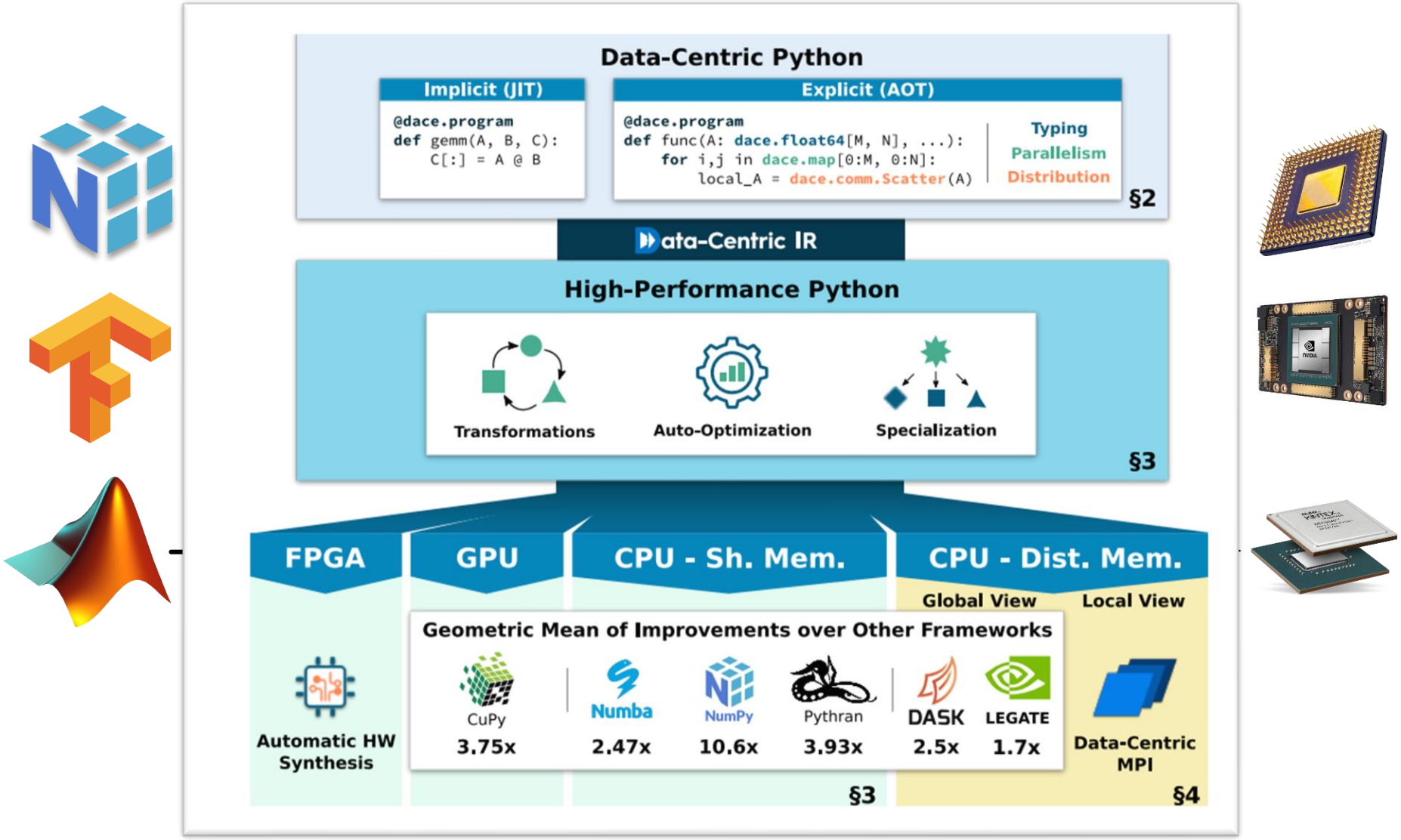
From the compiler point of view, the semantic information associated with parallel patterns offers a unique opportunity for optimization. These abstractions make it is easier to reason about parallelism and apply optimizations without the need for complex analysis. However, designing an IR (Internal 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

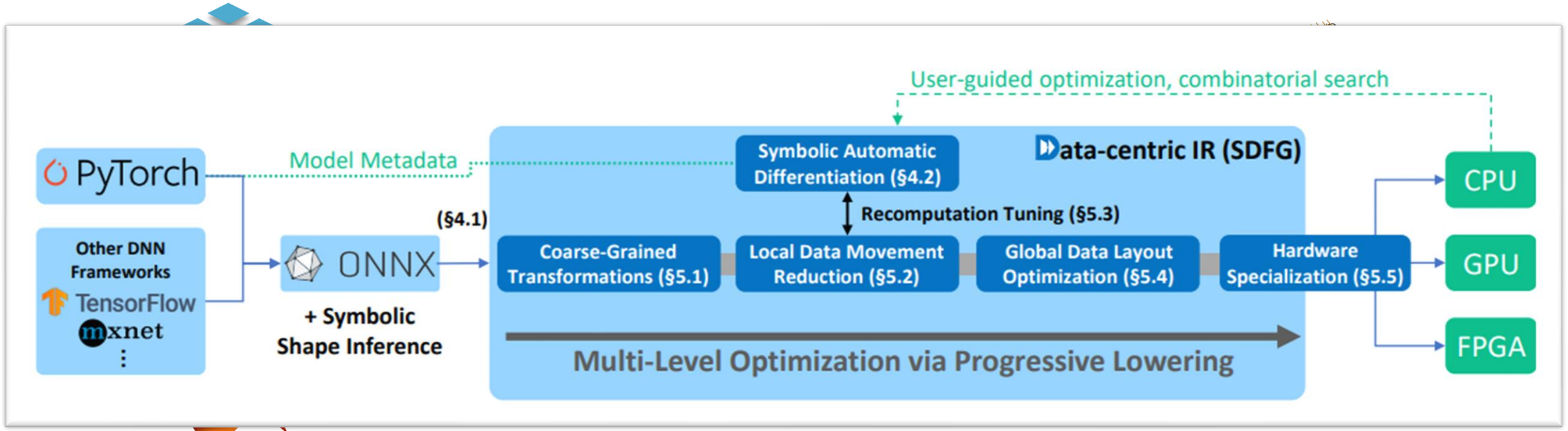
Data Centric Parallel Programming

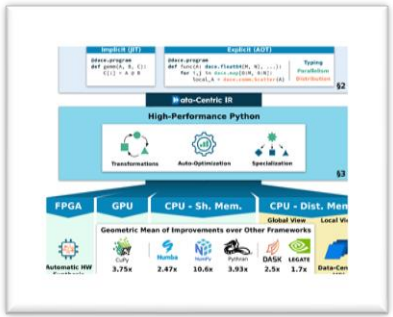


Data Centric Parallel Programming

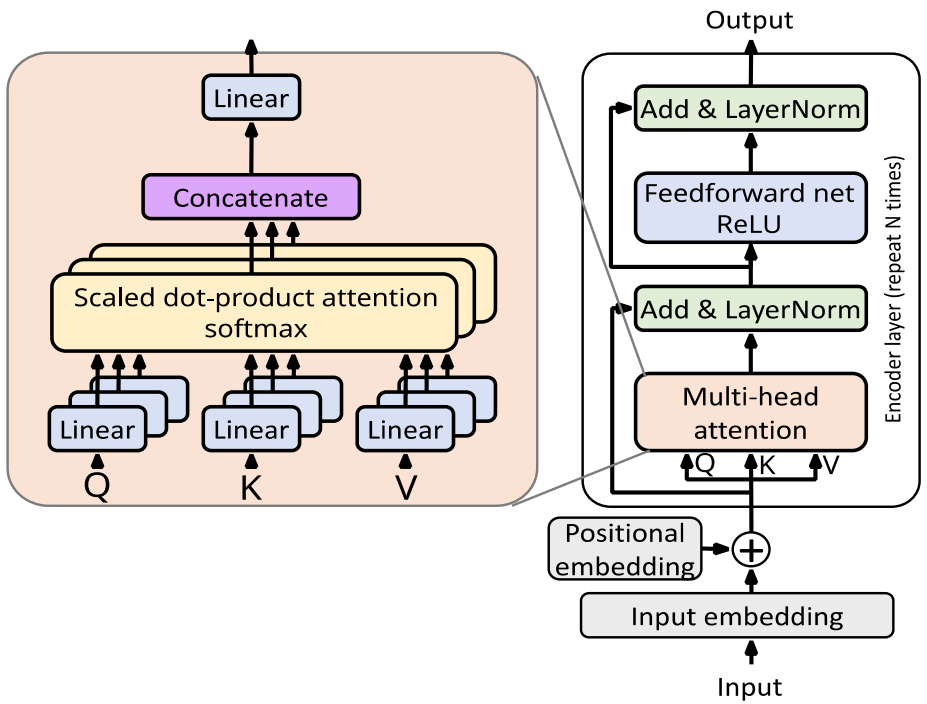


Data Centric Parallel Programming





Data Centric Parallel Programming



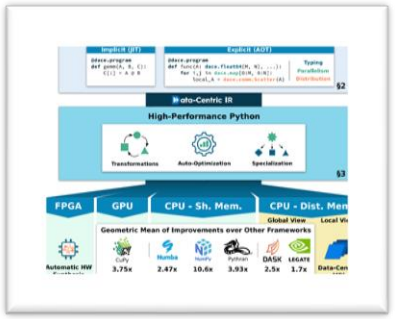
GPT-3 has a training cost of $12M

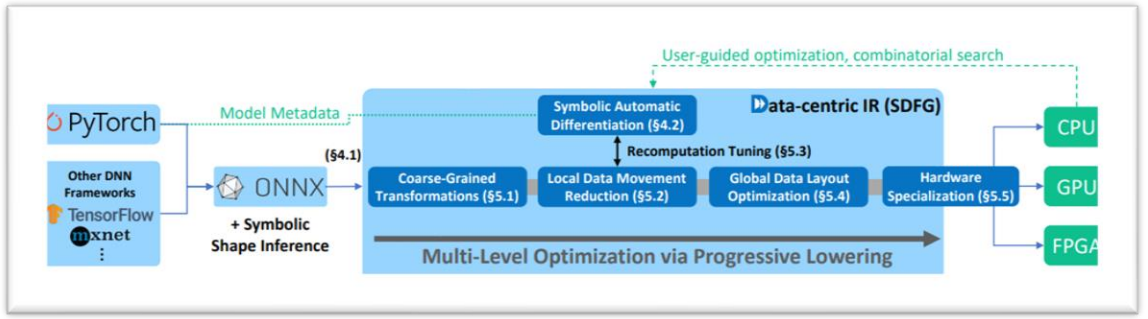
Our performance improvement for BERT-large

- 30% over PyTorch
- 20% over Tensorflow + XLA
- 8% over DeepSpeed

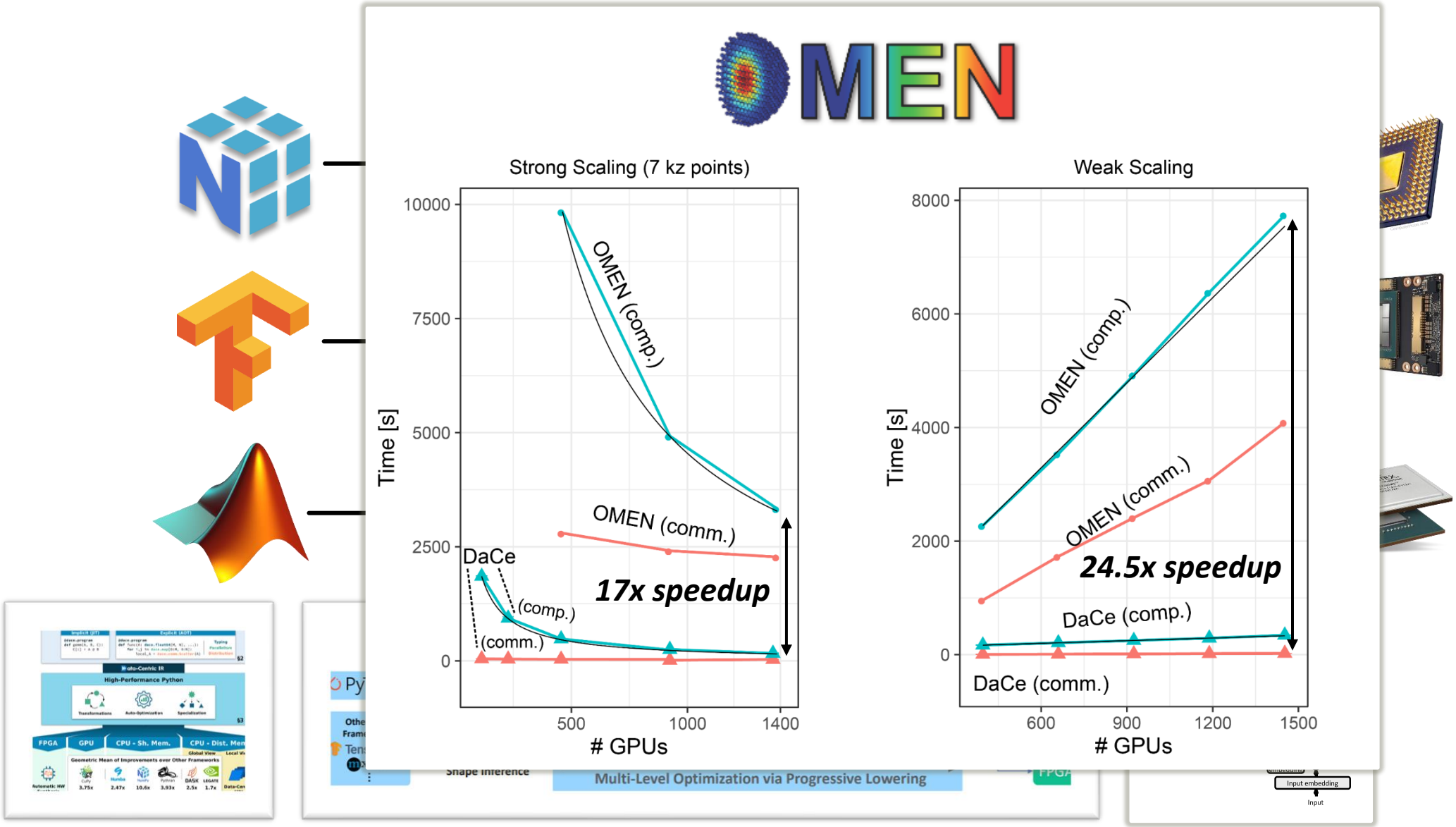
Estimated savings

- $85,000 for BERT on AWS using PyTorch
- $3.6M and 120 MWh for GPT-3

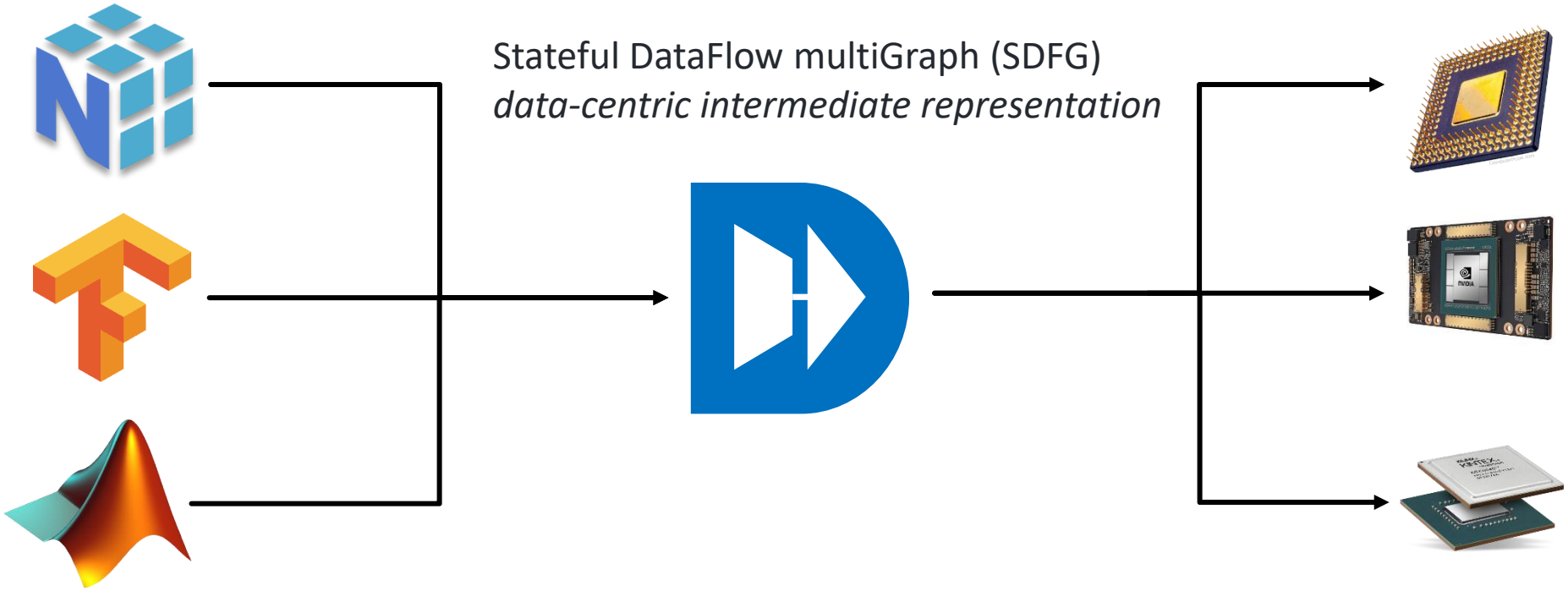


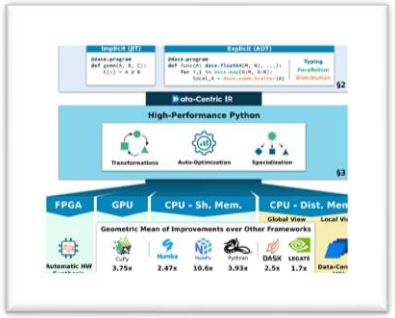


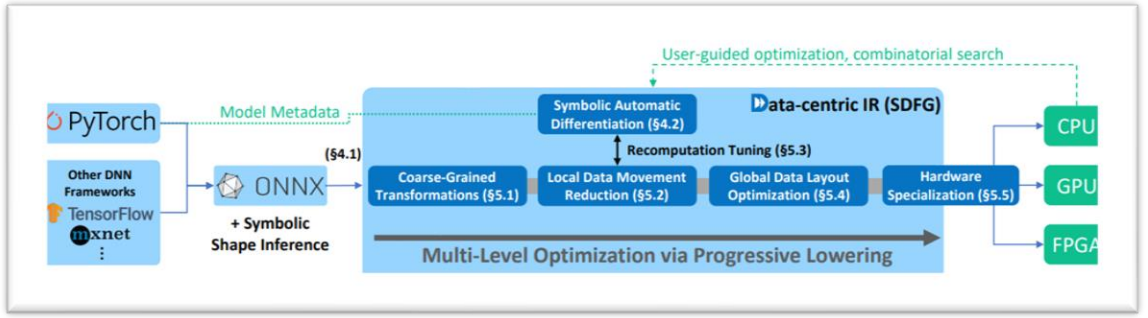
Data Centric Parallel Programming

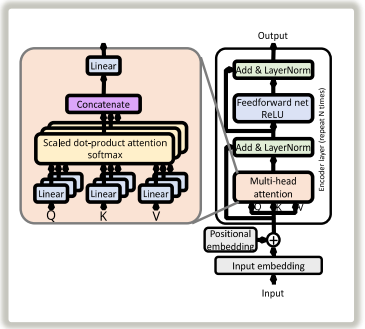


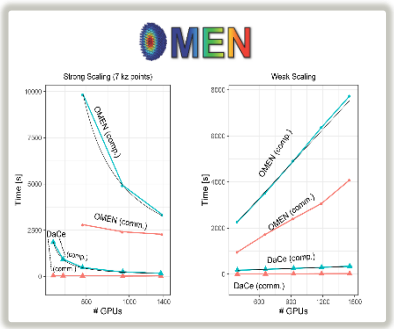
Data Centric Parallel Programming





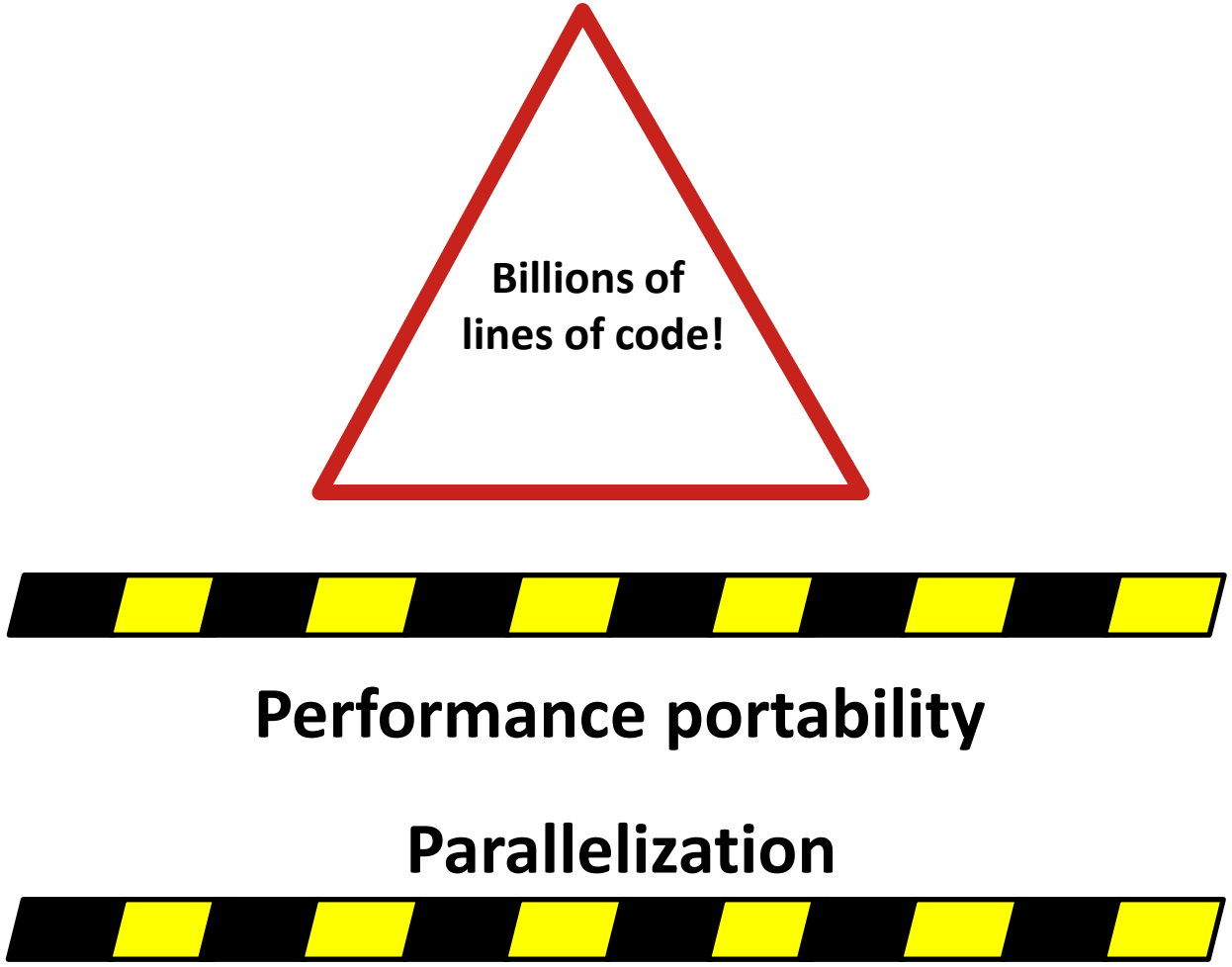




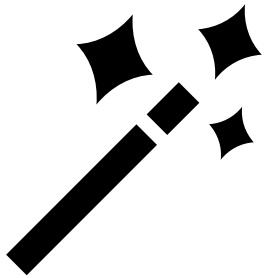


What about C?

TIOBE Index 2022	#2	#4
TIOBE Index 2021	#1	#4
TIOBE Index 2020	#1	#4
TIOBE Index 2019	#2	#4
TIOBE Index 2018	#2	#3
TIOBE Index 2017	#2	#3
TIOBE Index 2012	#2	#3
TIOBE Index 2007	#2	#3
TIOBE Index 2002	#2	#3
TIOBE Index 1997	#1	#2
TIOBE Index 1992	#1	#2
TIOBE Index 1987	#1	#5

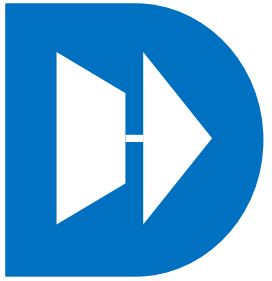


What about C?

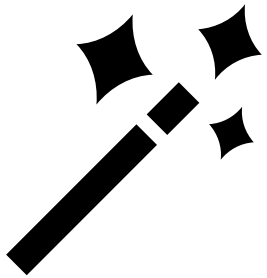


- Autoparallelization
- Generate GPU code
- Generate FPGA code
- Improve data movement
  - Apply tiling
  - Reorder loops

```c
static void kernel_jacobi_1d(int tsteps,
                             int n,
                             double A[2000 + 0],
                             double B[2000 + 0])
{
  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]);
    }
  }
}
```

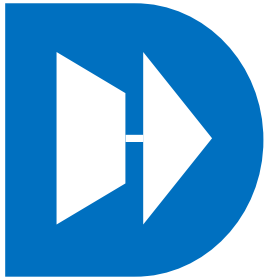


What about C?



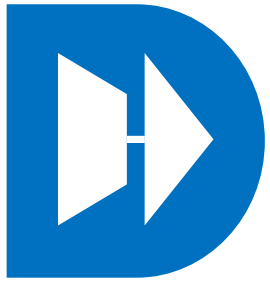
- Autoparallelization
- Generate GPU code ✓
- Generate FPGA code ✓
- Improve data movement ✓
  - Apply tiling ✓
  - Reorder loops ✓

```
static void kernel_jacobi_1d(int tsteps,
                             int n,
                             double A[2000 + 0],
                             double B[2000 + 0])
{
  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]);
    }
  }
}
```



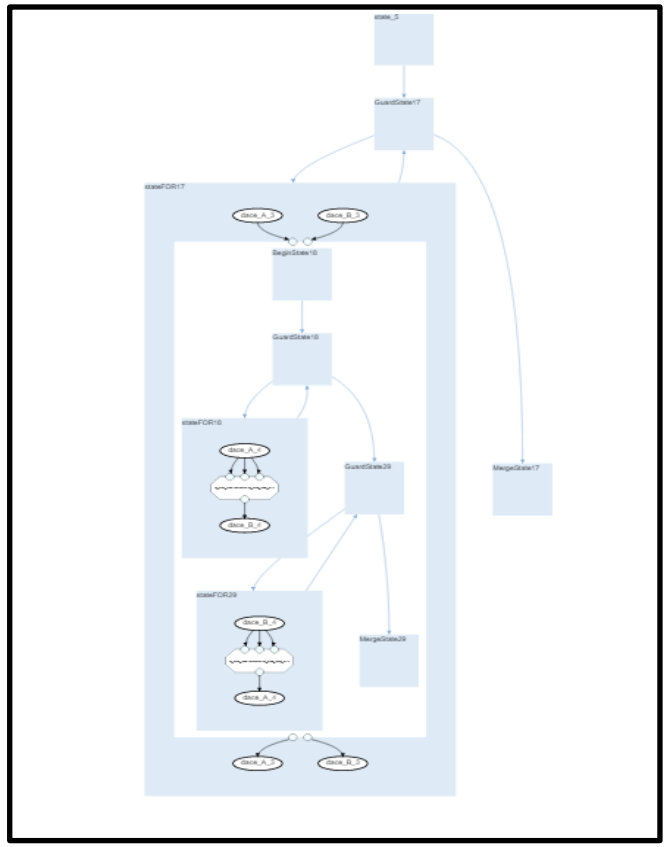
What about C?

C to DaCe



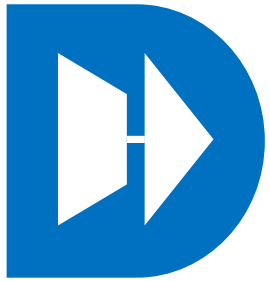
*Autoparallelization

```
static void kernel_jacobi_1d(int tsteps,
                             int n,
                             double A[2000 + 0],
                             double B[2000 + 0])
{
  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]);
    }
  }
}
```



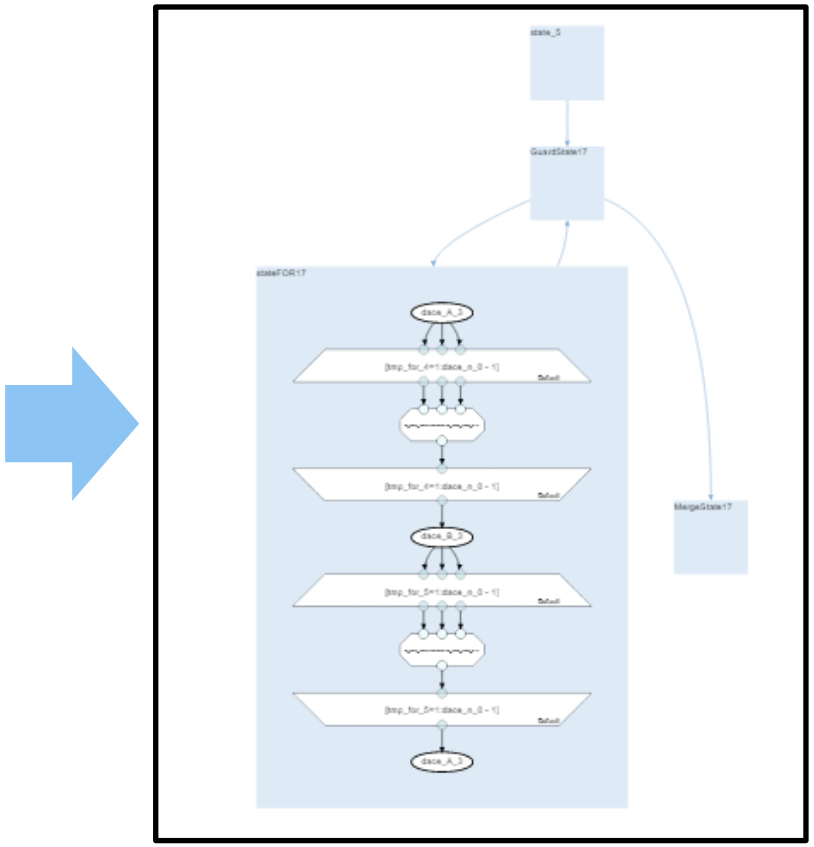
What about C?

C to DaCe



*Autoparallelization

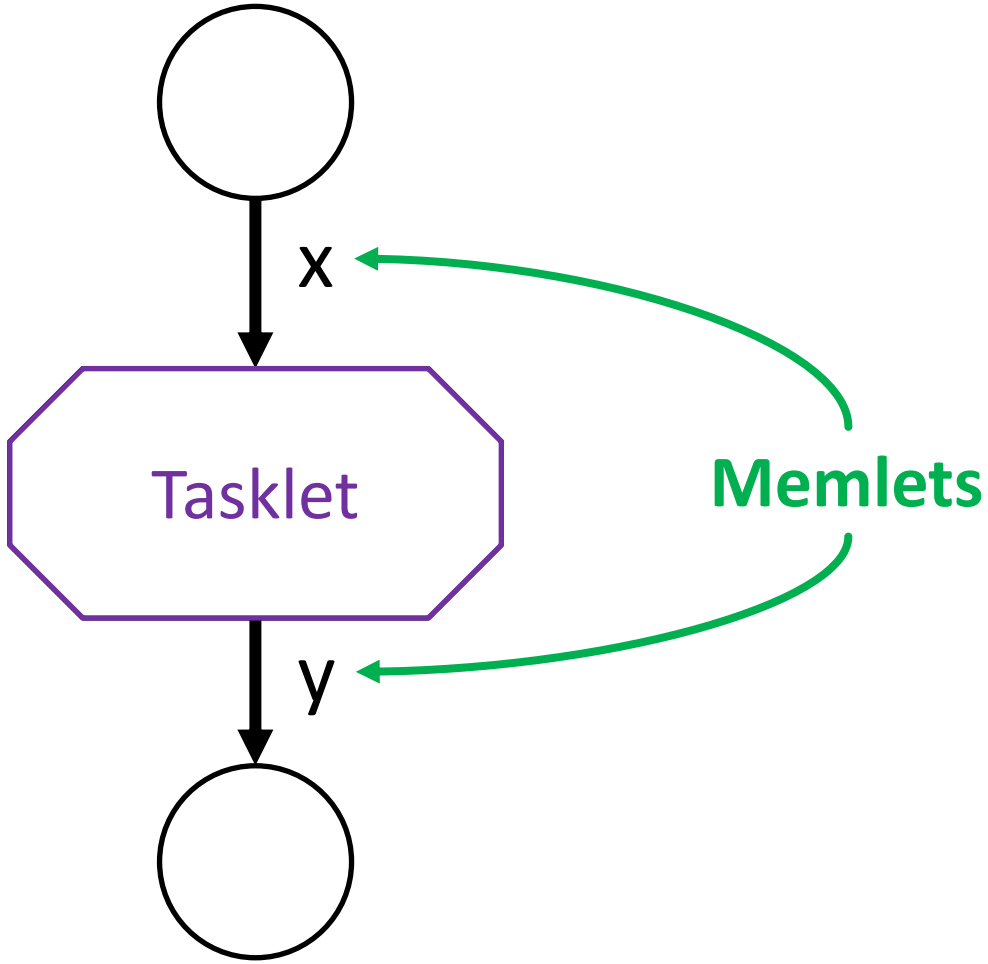
```
static void kernel_jacobi_1d(int tsteps,
                             int n,
                             double A[2000 + 0],
                             double B[2000 + 0])
{
  int t, i;

  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]);
    }
  }
}
```



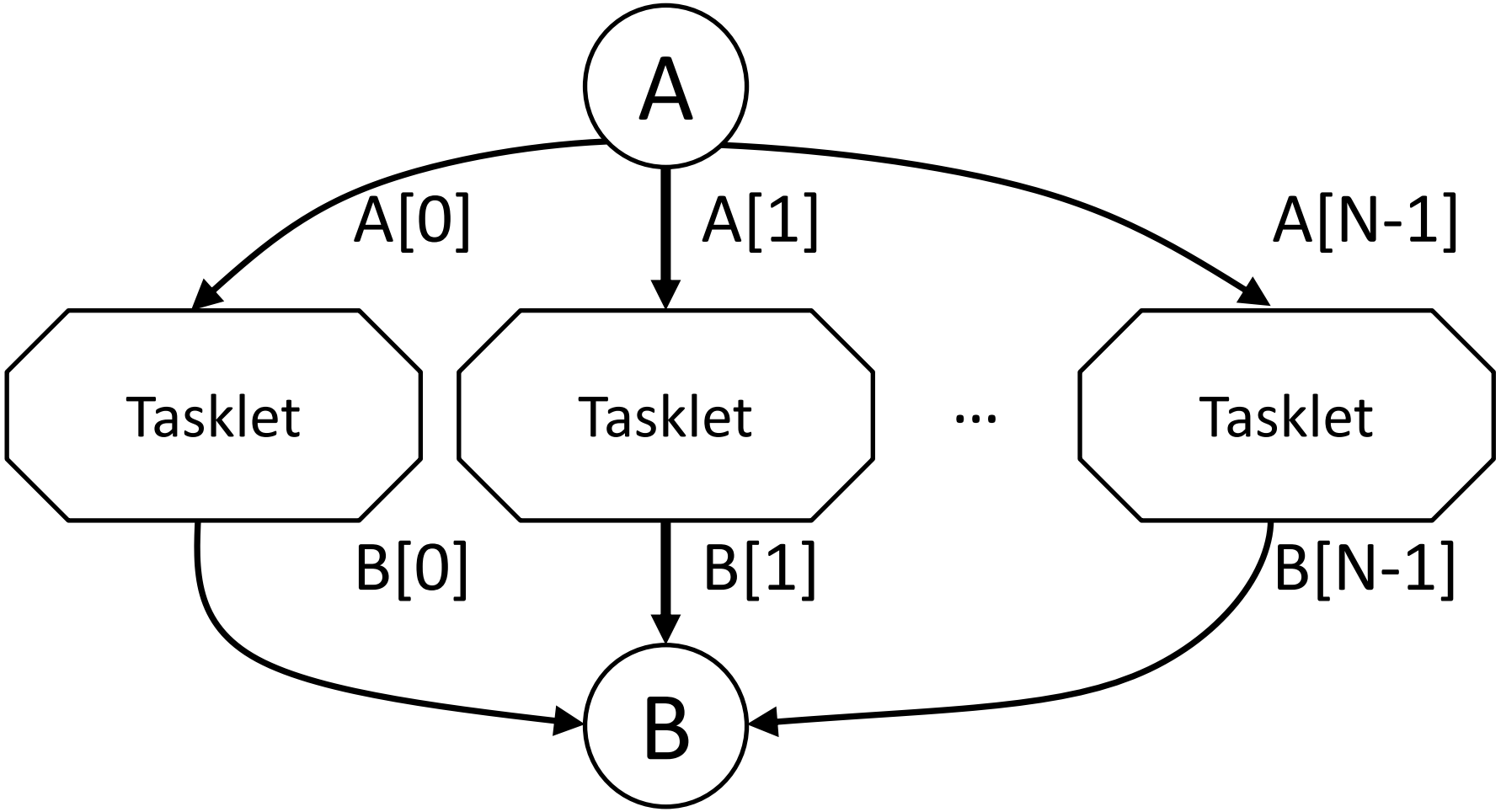
Dataflow Programming in DaCe

$$y = x^2 + \sin\frac{x}{\pi}$$

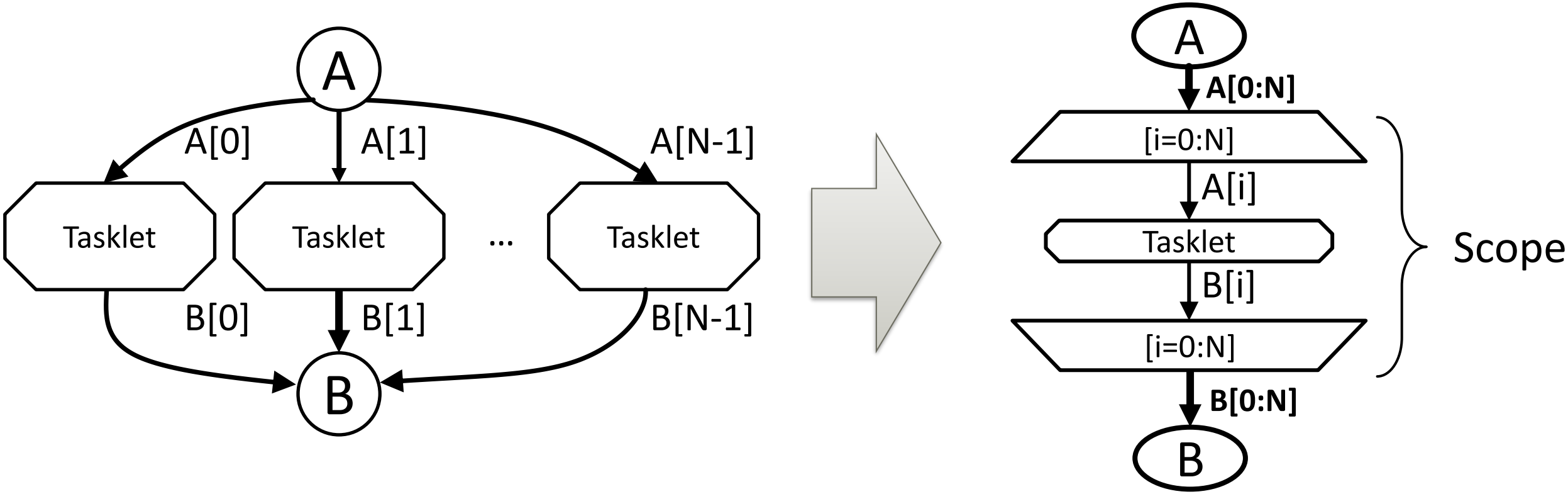
Dataflow Programming in DaCe



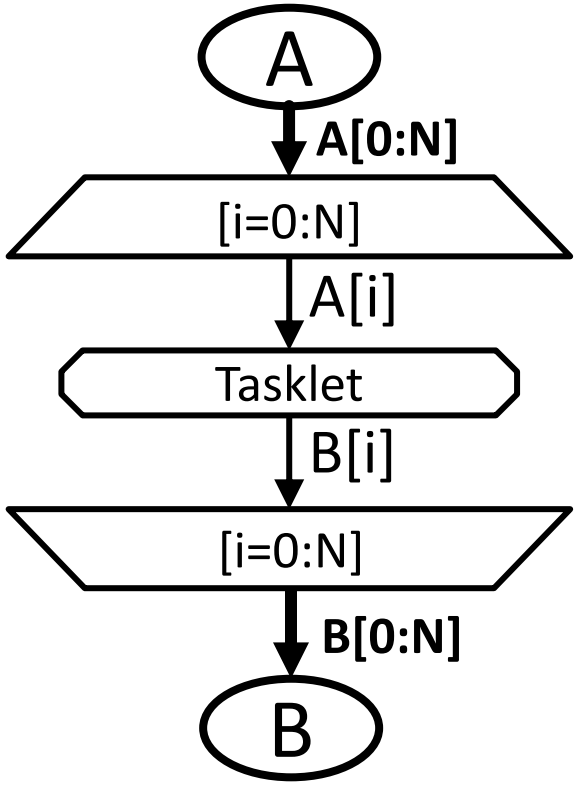
Parallel Dataflow Programming

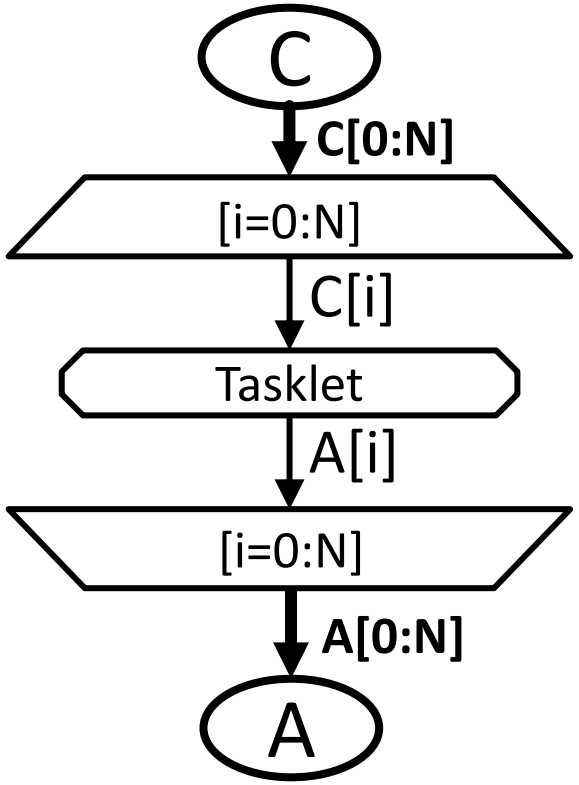


Parallel Dataflow Programming

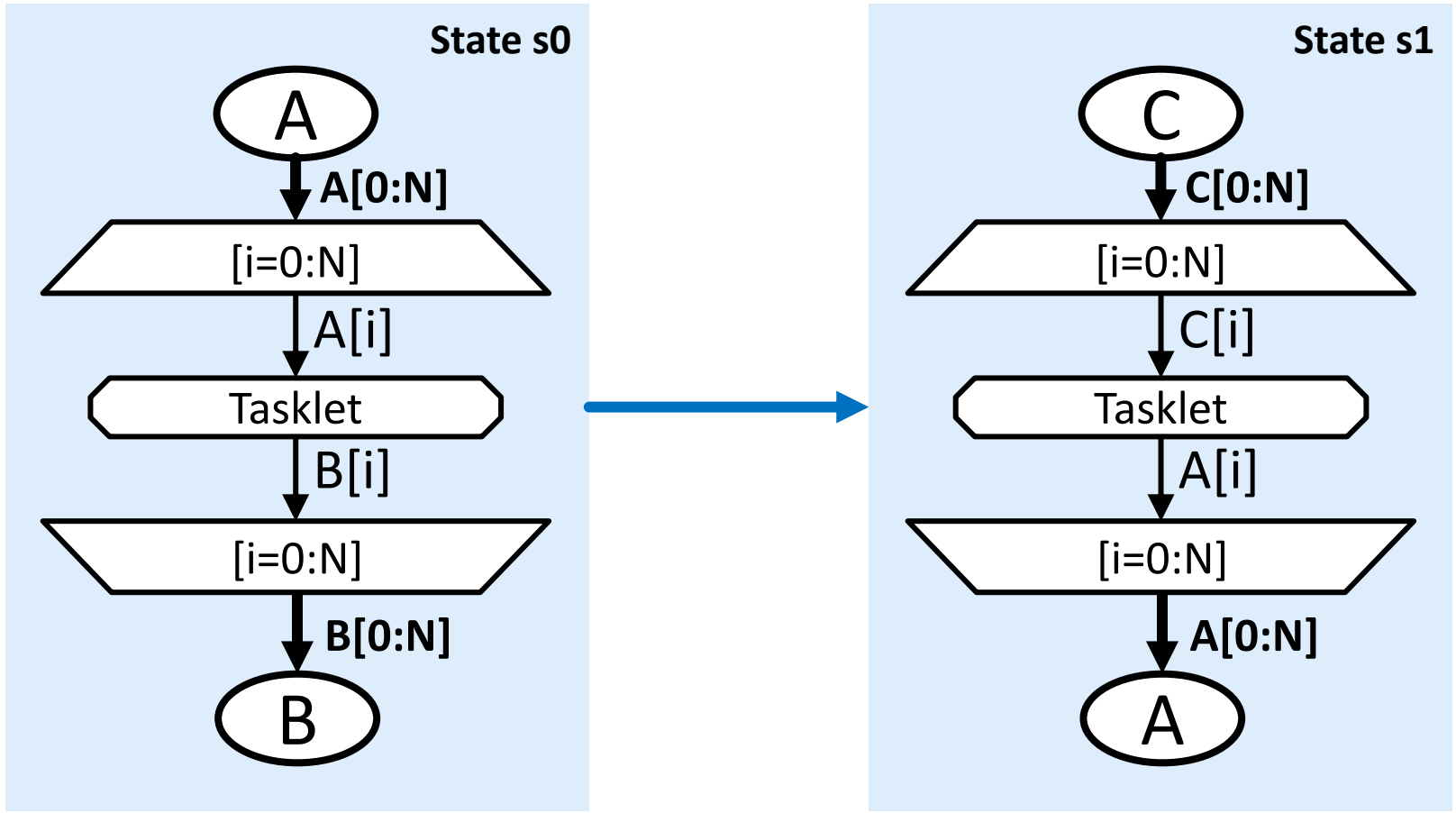


Stateful Parallel Dataflow Programming

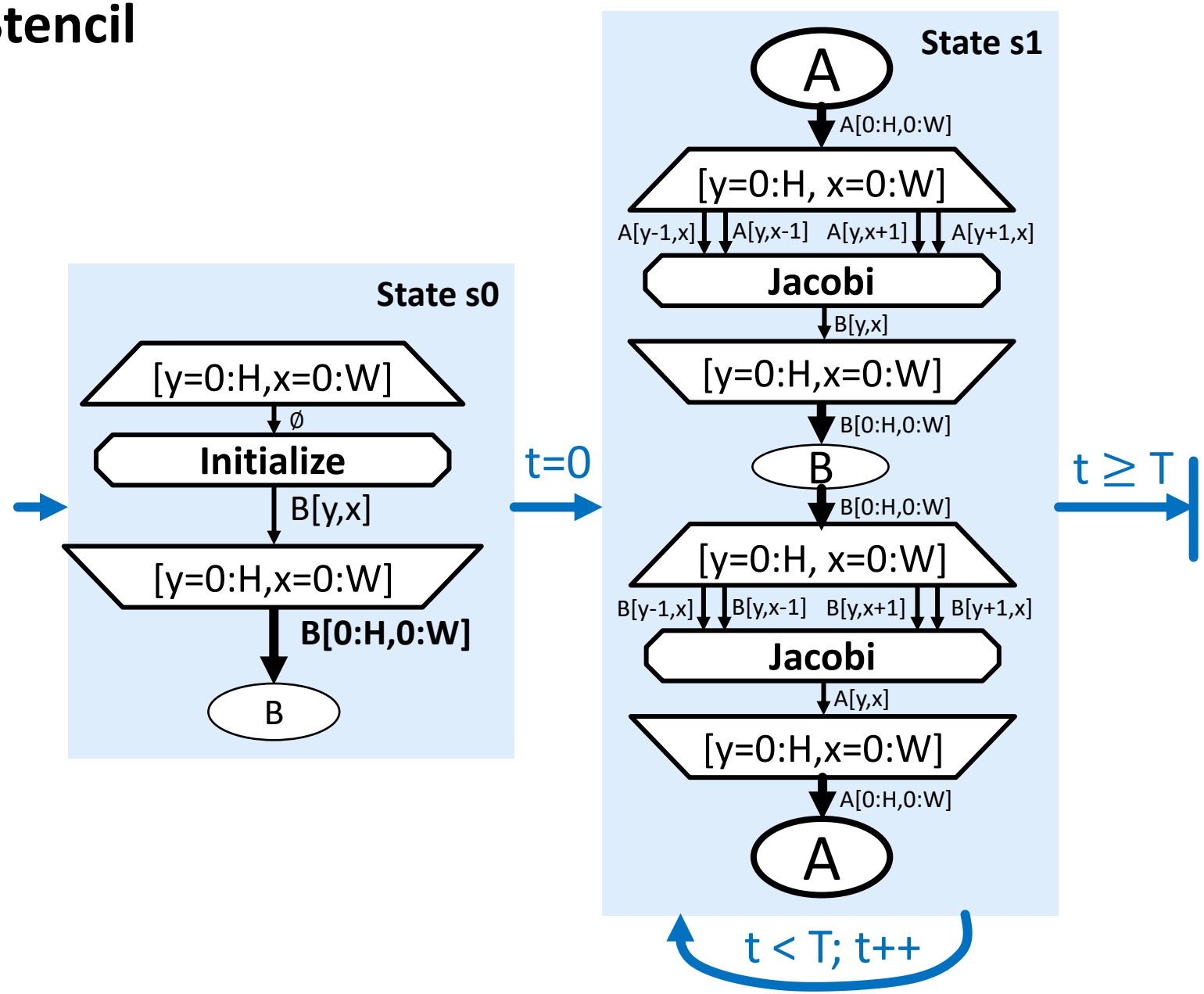




Stateful Parallel Dataflow Programming



Example: 2D Stencil



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[col_idx[j]] * x[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[col_idx[j]] * x[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[   idx    ] * x[j];
 }
}
```

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[  idx   ] * x[j];
 }
}
```

```
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[  idx   ] * x[j];
 }


}
```

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[  idx   ] * x[j];
 }
}
```

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

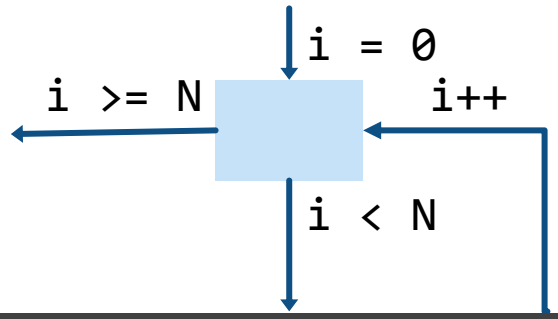
```
 for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){
      int idx =  col_idx[j];
     y[i] +=  A[  idx   ] * x[j];
 }
```

```
}
```

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[  idx   ] * x[j];
 }
}
```

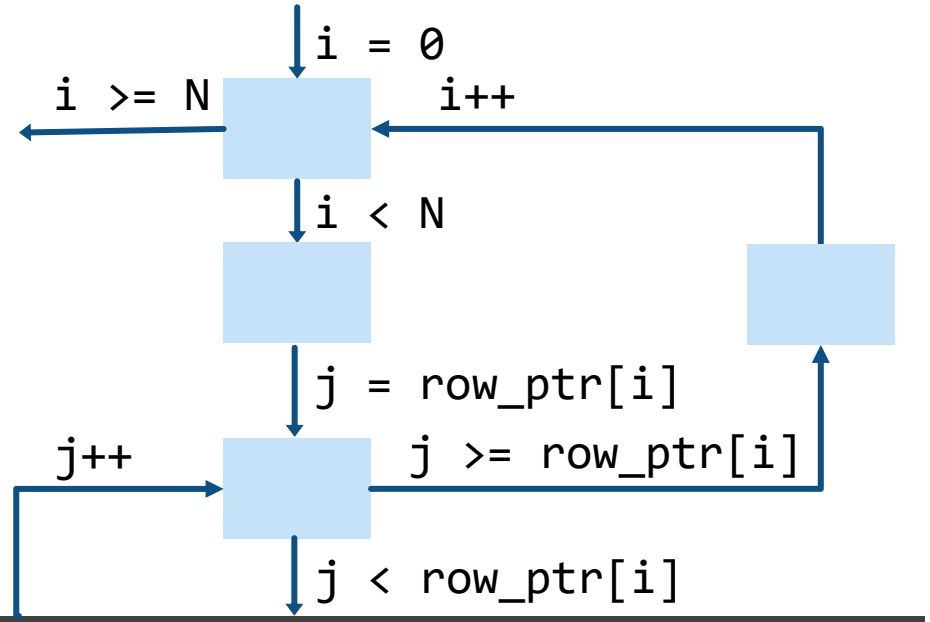



```
for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){
     int idx =  col_idx[j];
    y[i] +=  A[  idx   ] * x[j];
}
```

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[  idx   ] * x[j];
 }
}
```

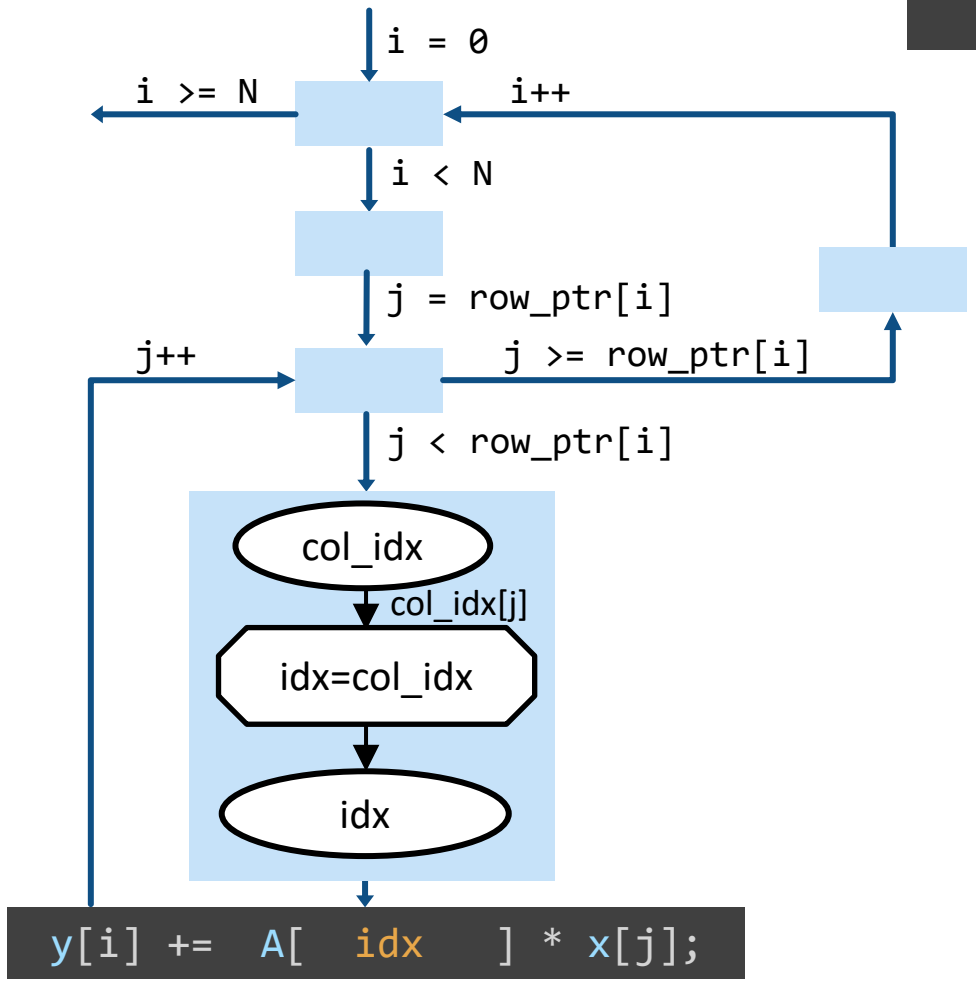



```
int idx =  col_idx[j];
y[i] +=  A[  idx   ] * x[j];
```

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[  idx   ] * x[j];
 }
}
```

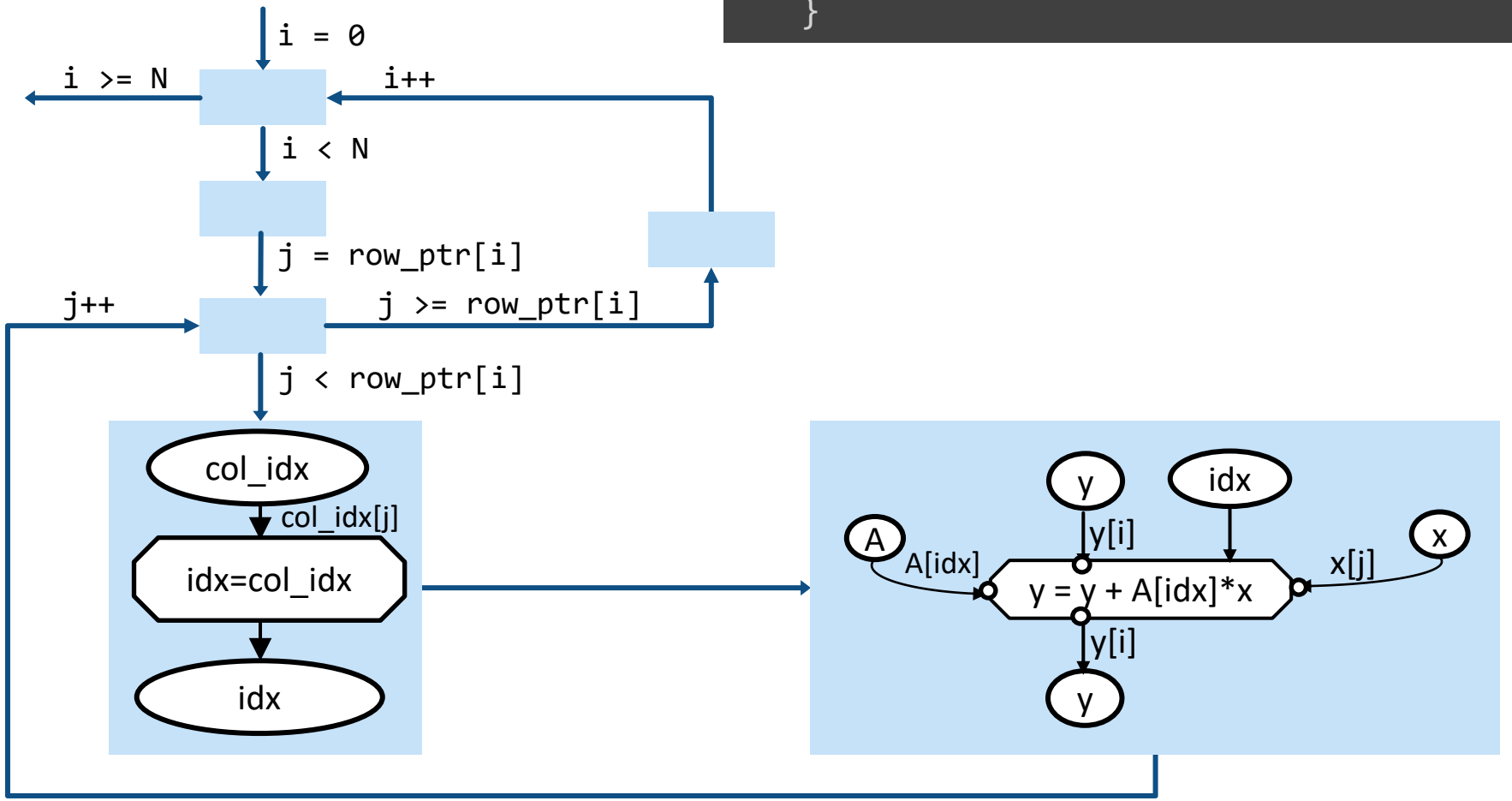
1. AST Transformations
2. Translation from C to SDFG



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[  idx   ] * x[j];
 }
}
```

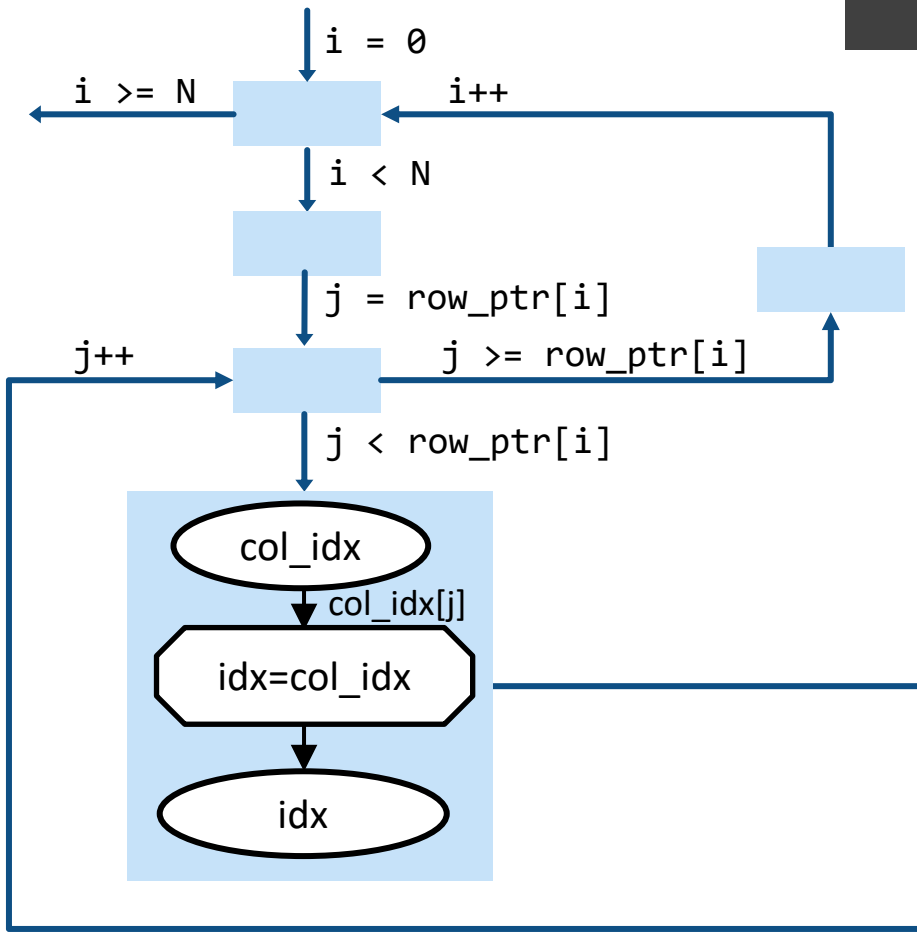
1. AST Transformations
2. Translation from C to SDFG

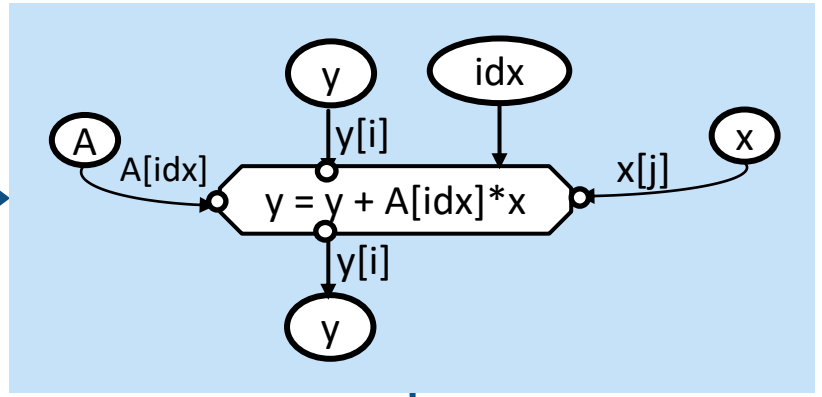


C to DaCe: SpMV

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[  idx   ] * x[j];
 }
}
```



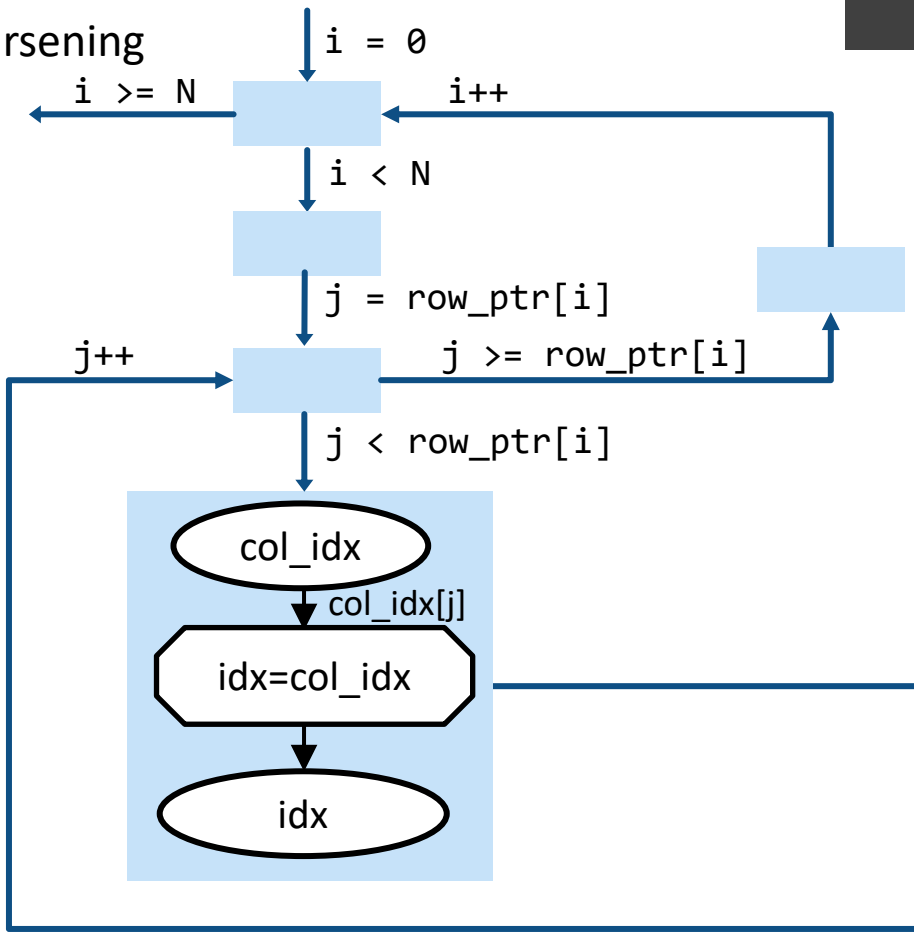


The SDFG representation is at this point a direct representation of the *control-flow centric* C code.

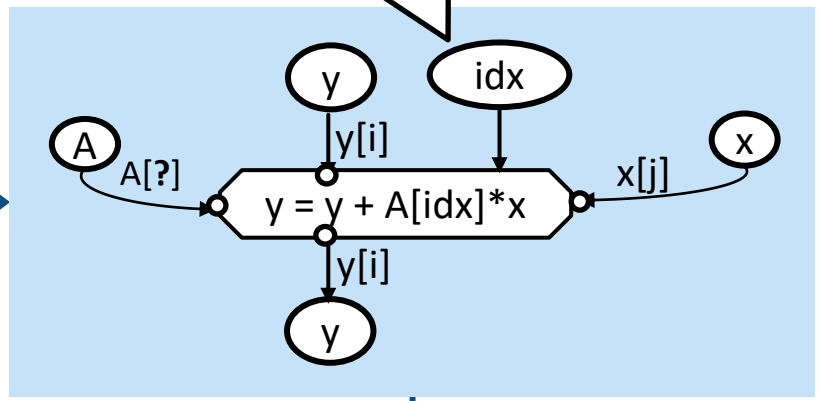
C to DaCe: SpMV

```
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[  idx   ] * x[j];
 }
}
```

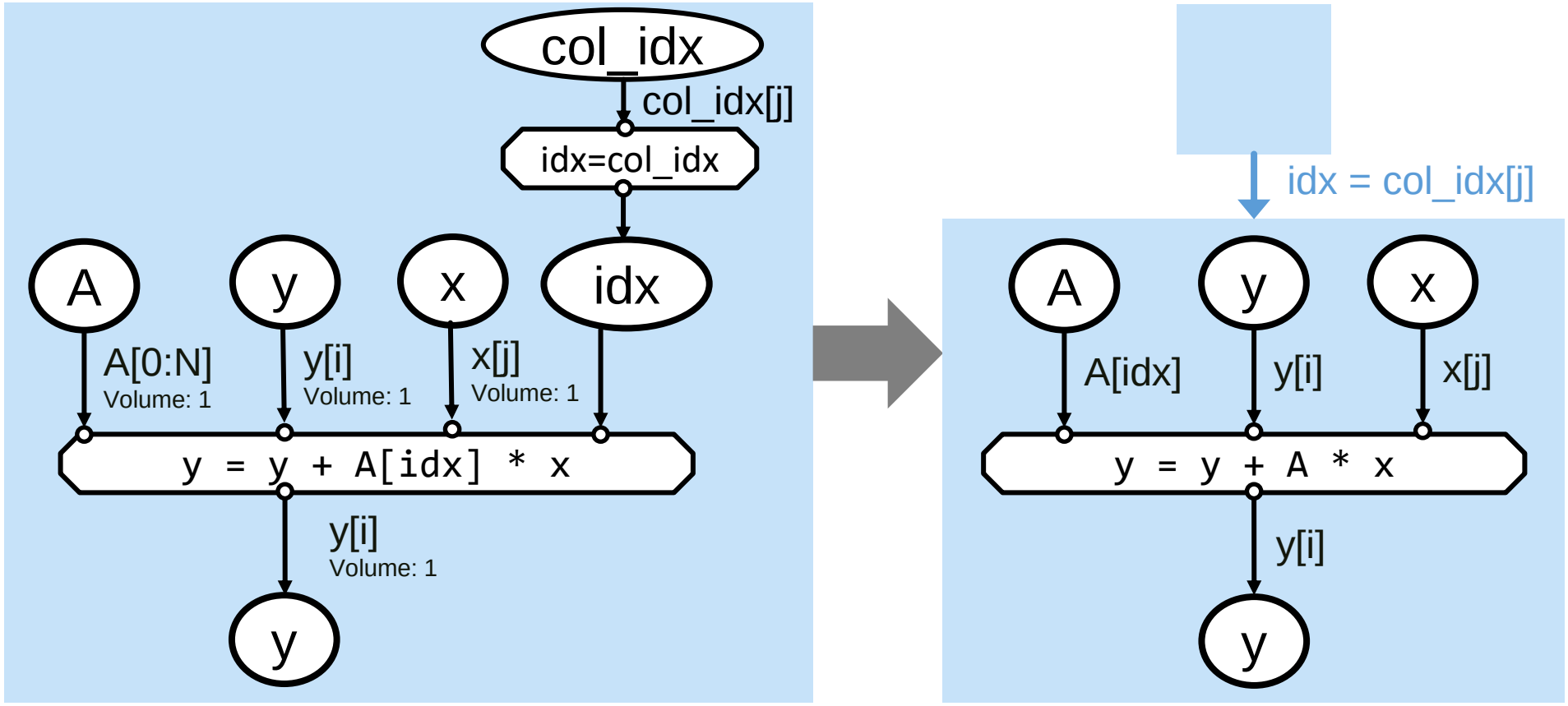
1. AST Transformations
2. Translation from C to SDFG
3. Dataflow coarsening



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



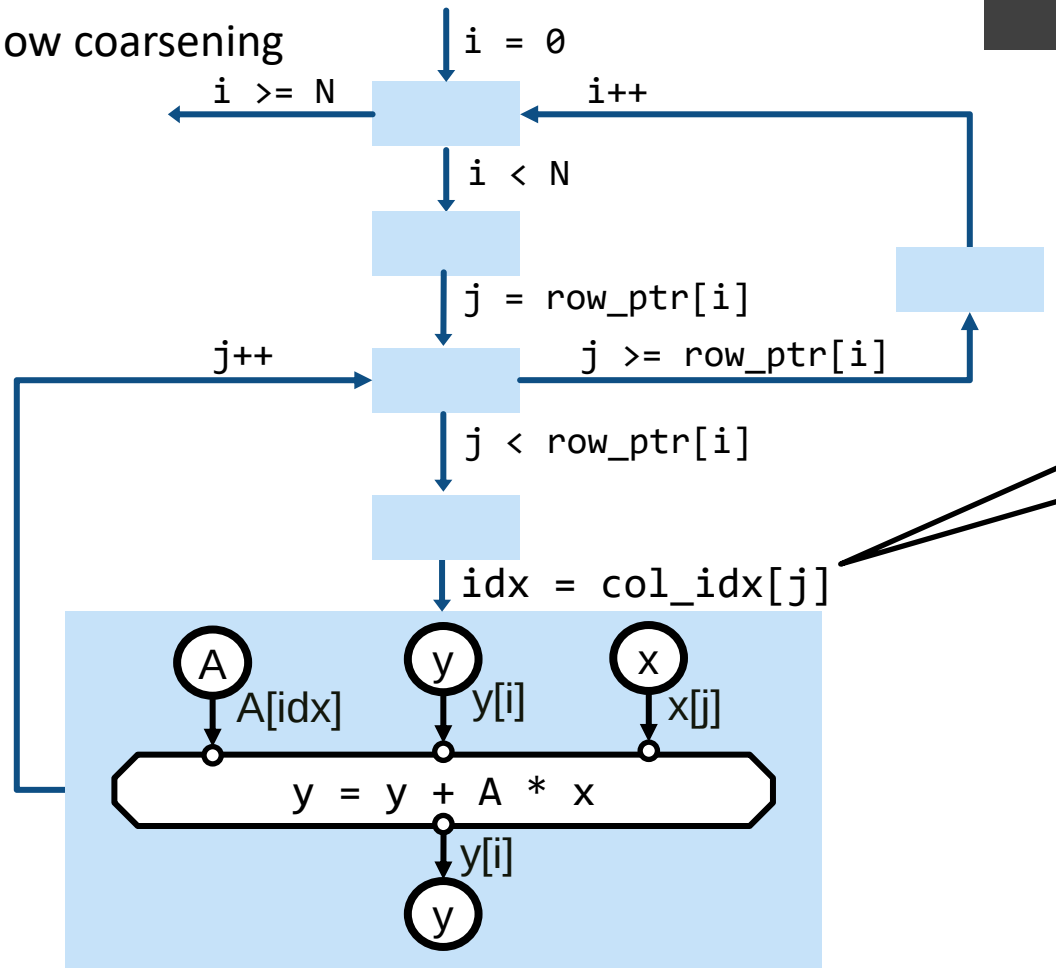
Symbolic scalar analysis



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[  idx   ] * x[j];
 }
}
```

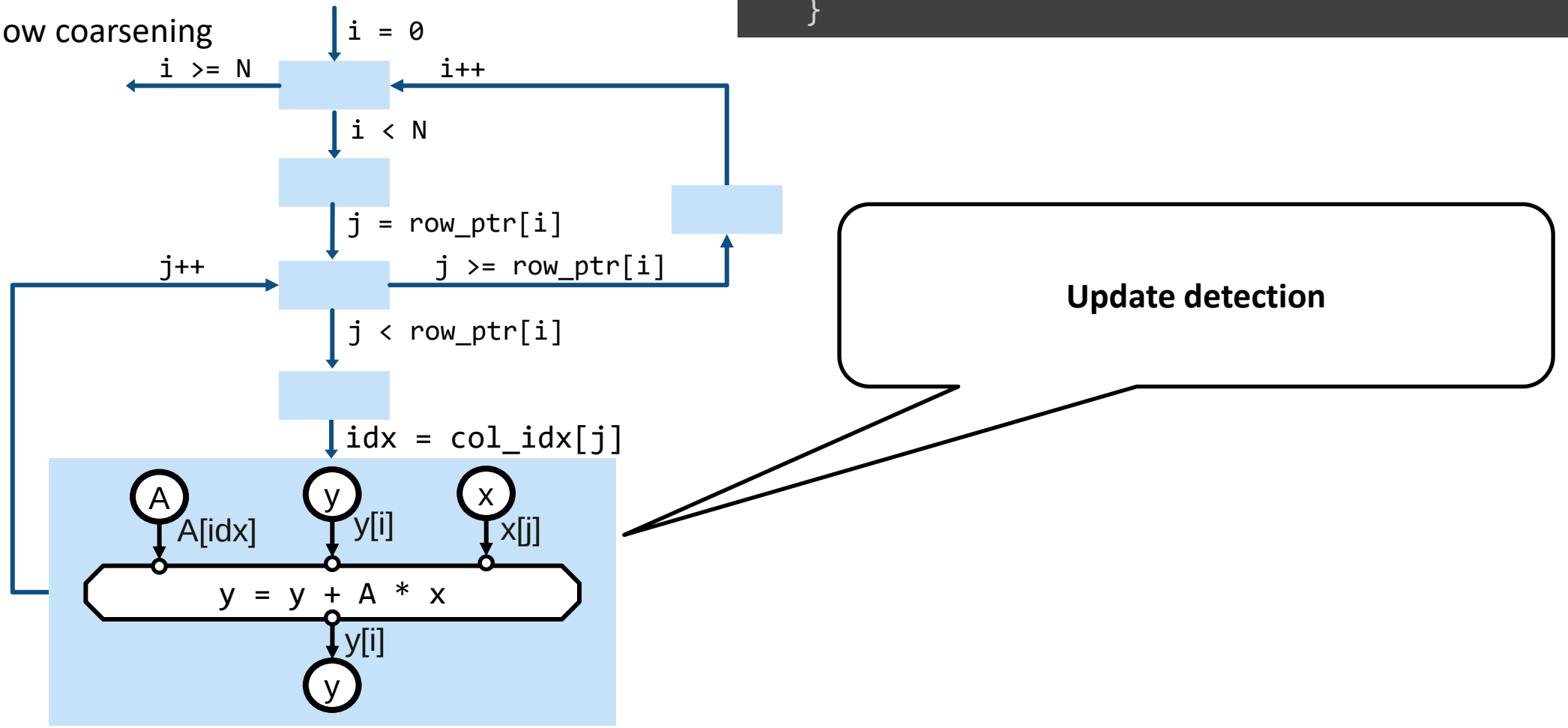


Symbolic scalar analysis:
Improves understanding of data access patterns by allowing symbolic manipulation

C to DaCe: SpMV

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

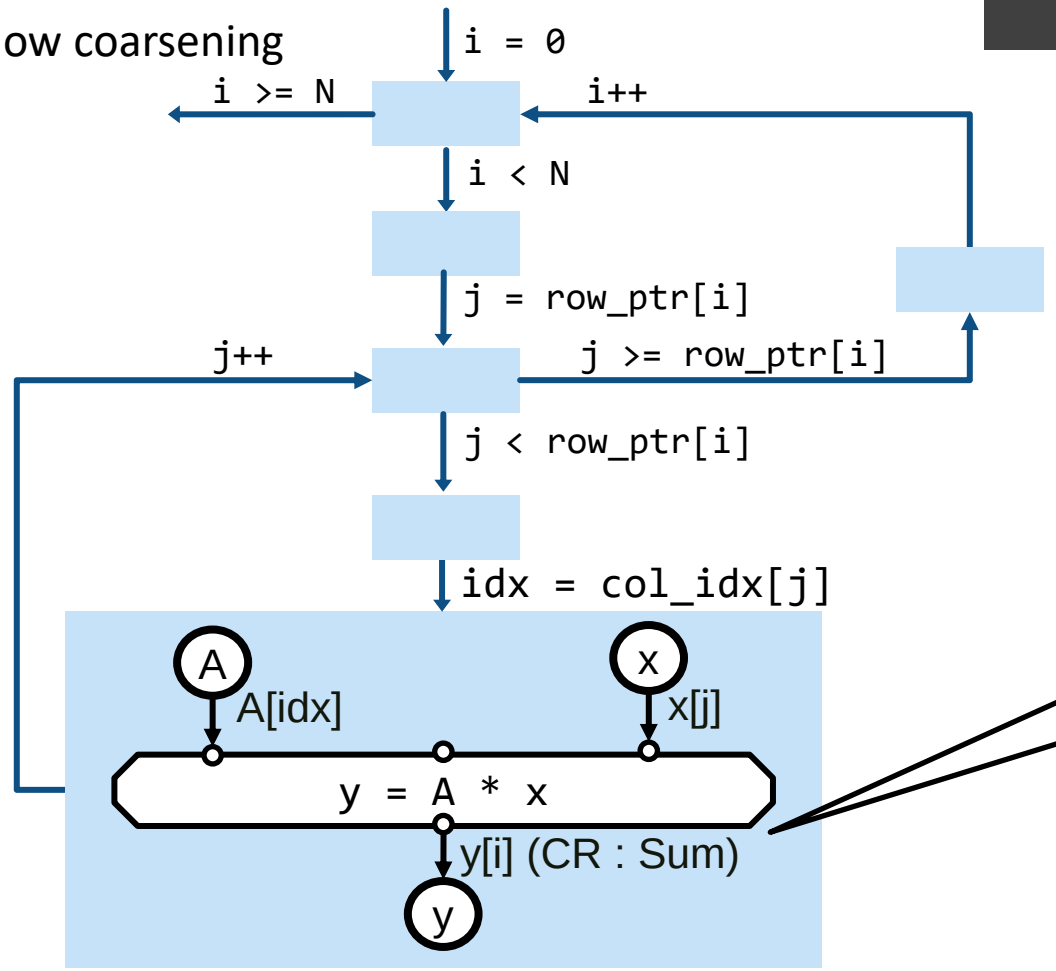
```
for (int i = 0; i < N ; i++){
 for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){
      int idx =  col_idx[j];
     y[i] +=  A[  idx   ] * x[j];
 }
}
```



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[  idx   ] * x[j];
 }
}
```

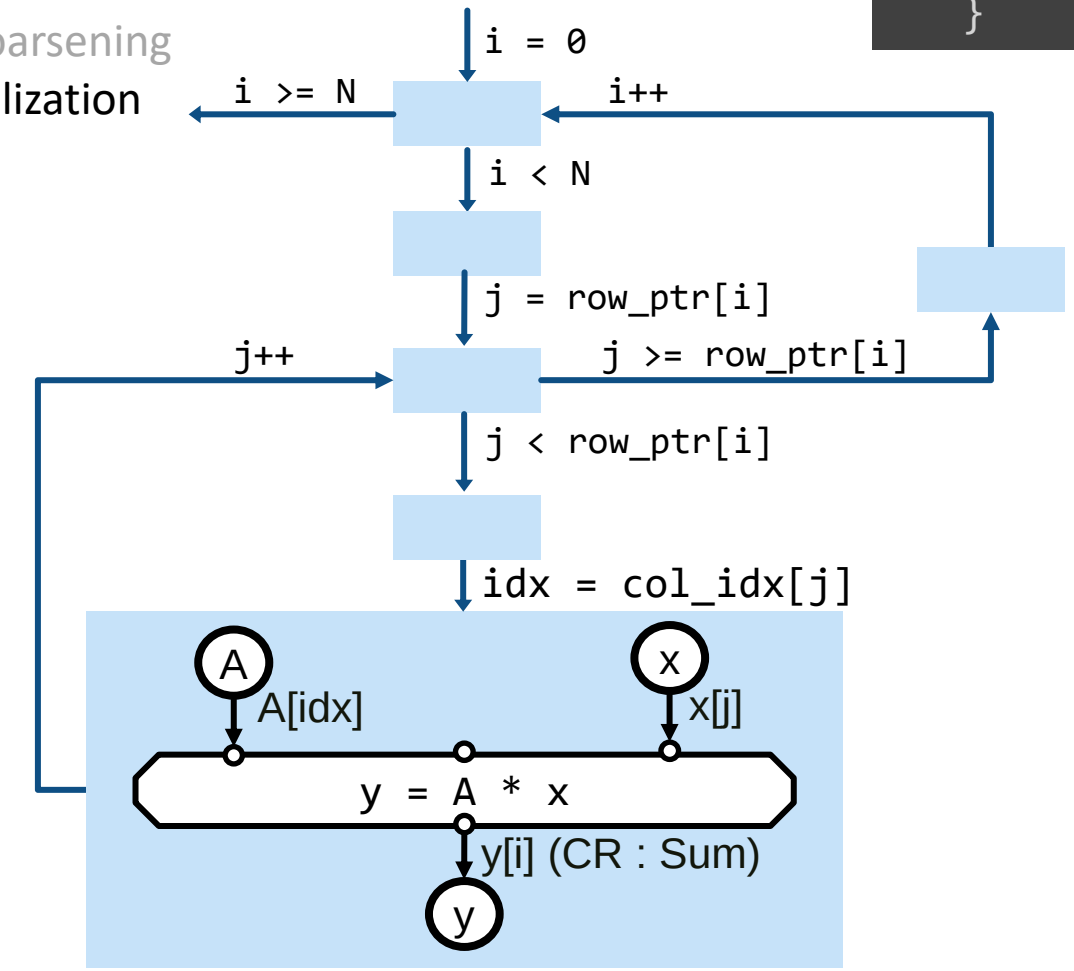


Update detection:
Allows more parallelization by creating the specialized *update* type of assignment, supporting conflict resolution for multiple operations

C to DaCe: SpMV

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

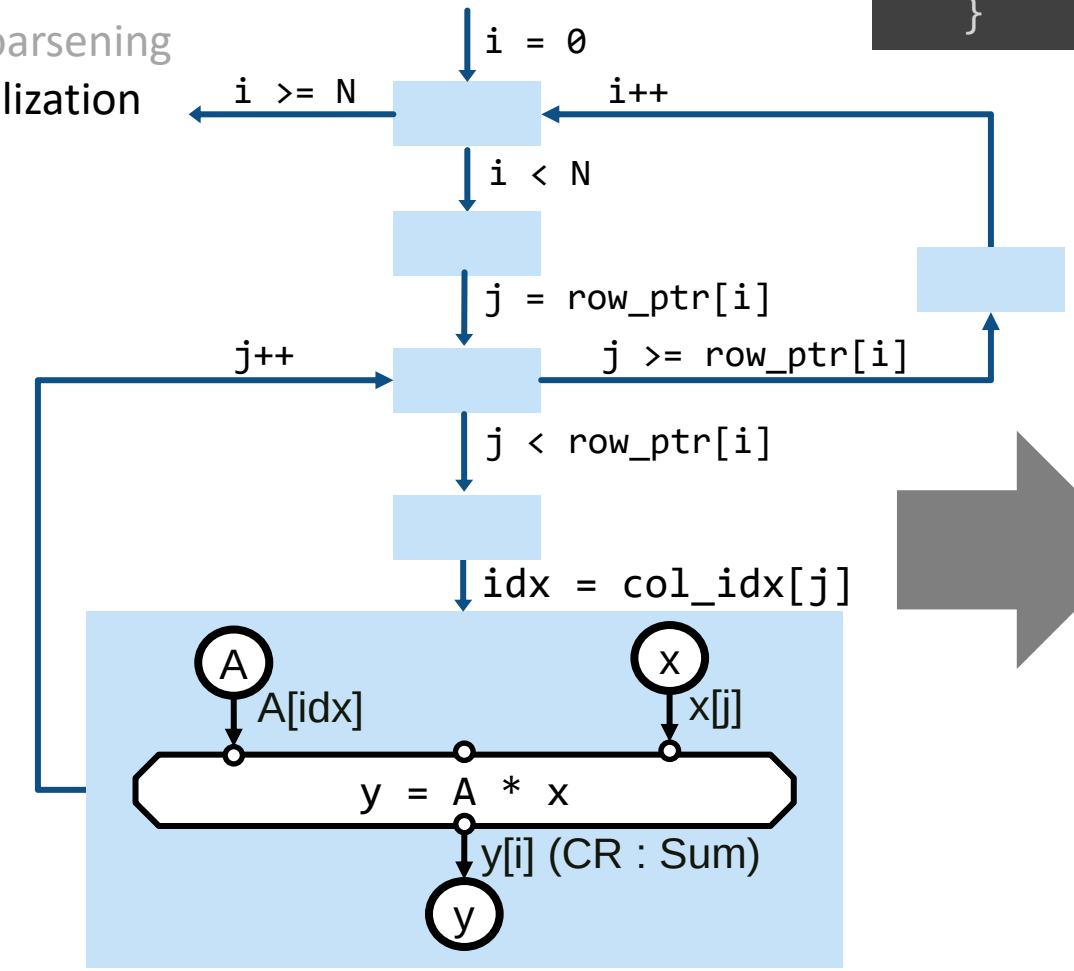
```
for (int i = 0; i < N ; i++){
 for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){
      int idx =  col_idx[j];
     y[i] +=  A[  idx   ] * x[j];
 }
}
```

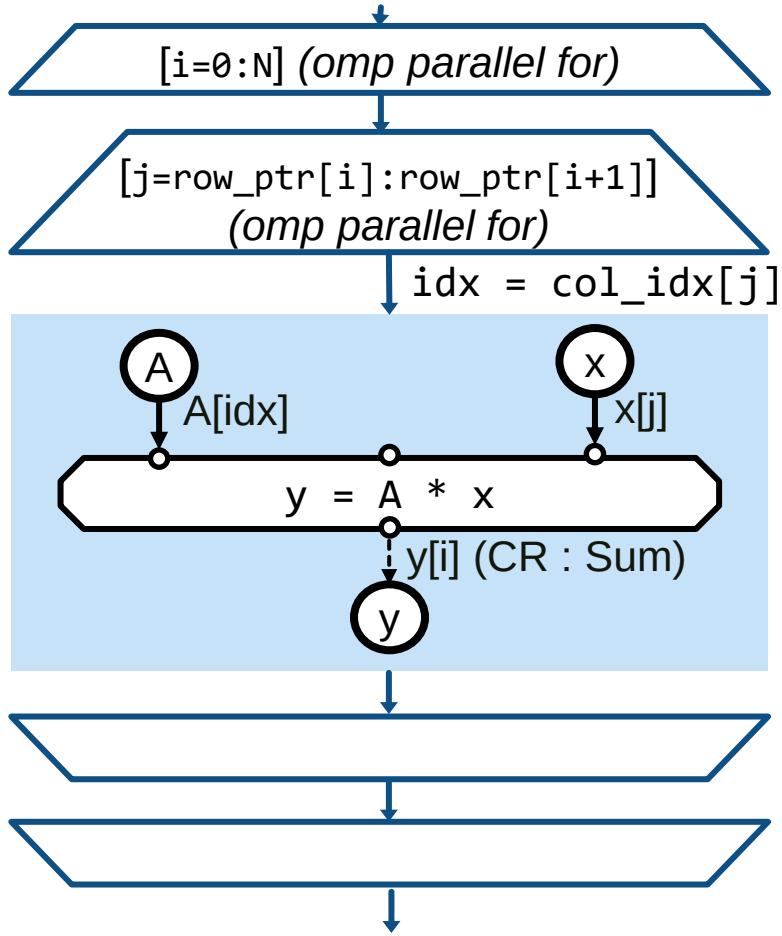


C to DaCe: SpMV

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[  idx   ] * x[j];
 }
}
```

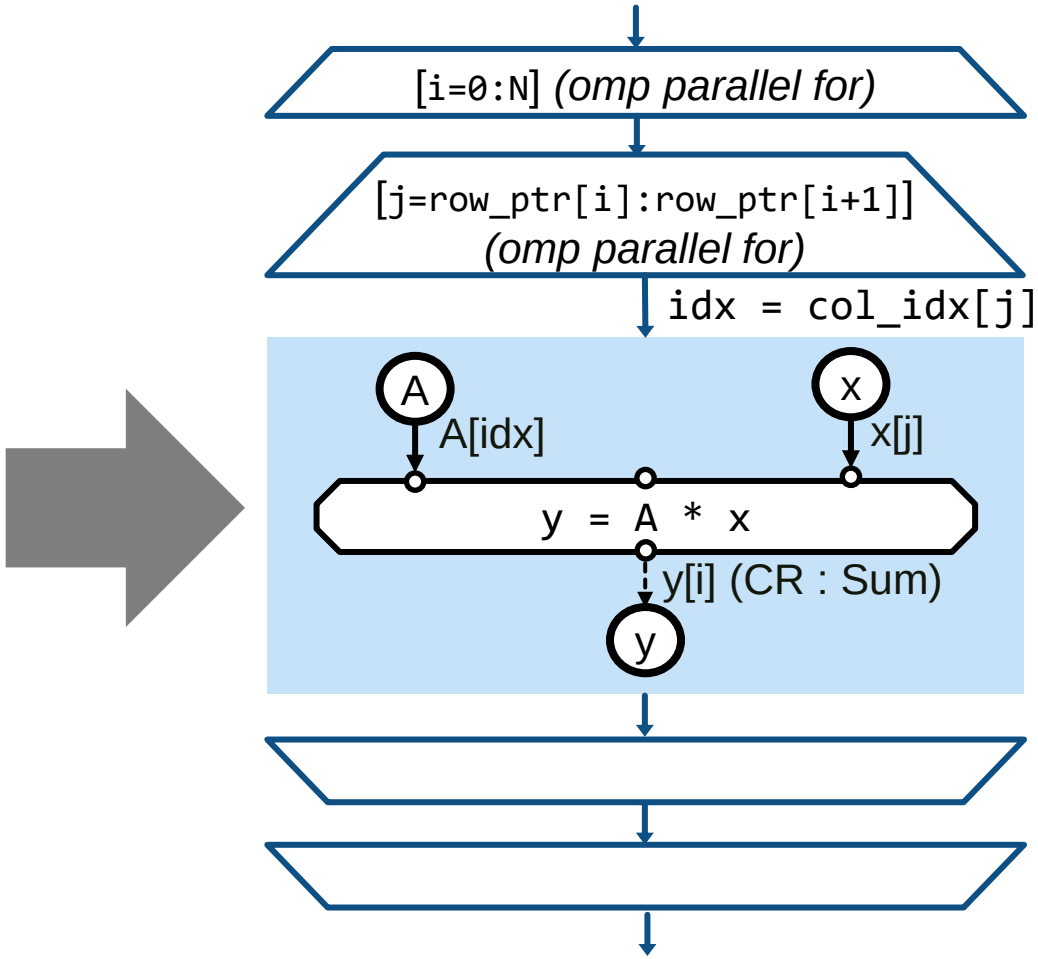




C to DaCe: SpMV

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

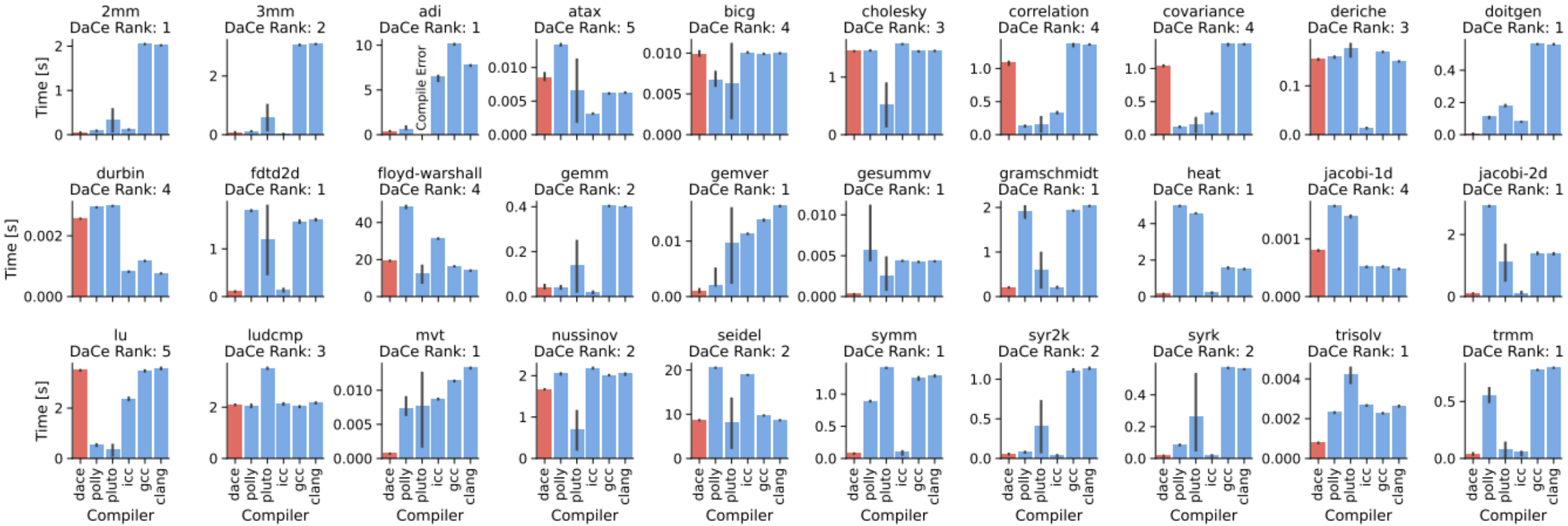
```
for (int i = 0; i < N ; i++){
 for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){
      int idx =  col_idx[j];
     y[i] +=  A[  idx   ] * x[j];
 }
}
```



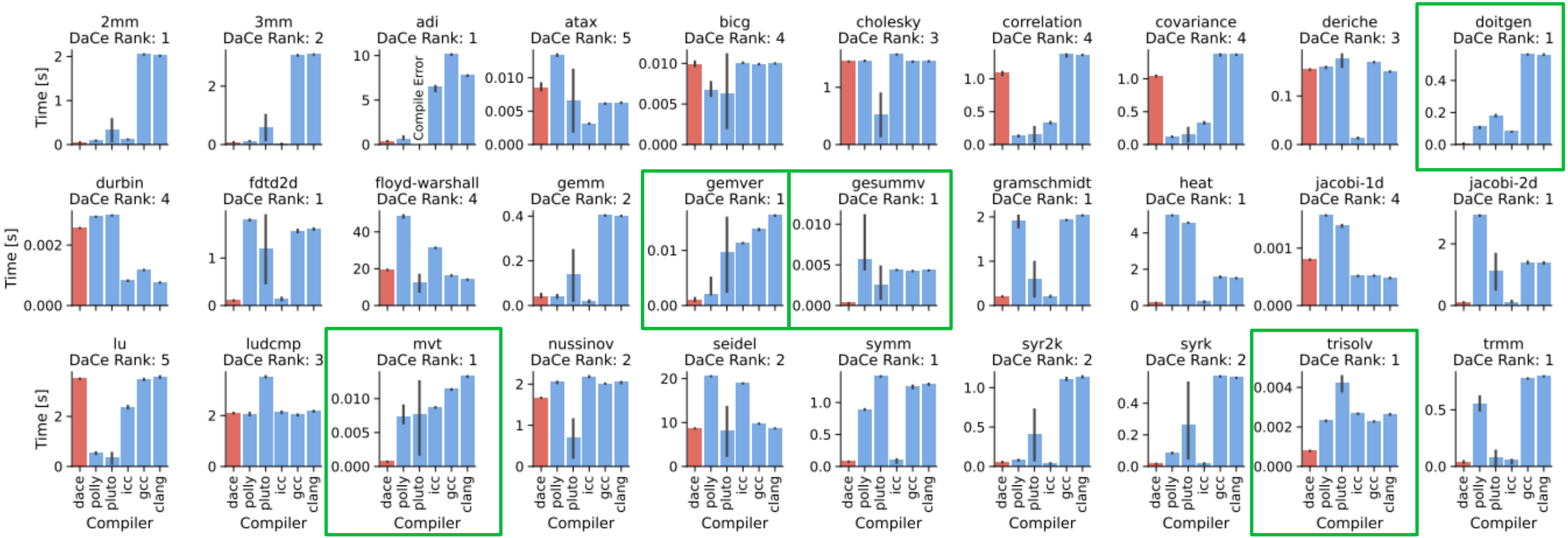
Performance on up to 36 cores

DaCe	polly	pluto	icc - parallel	icc	gcc	clang
0.54s	3.95s	failed	0.45s	4.98s	3.55s	4.19s

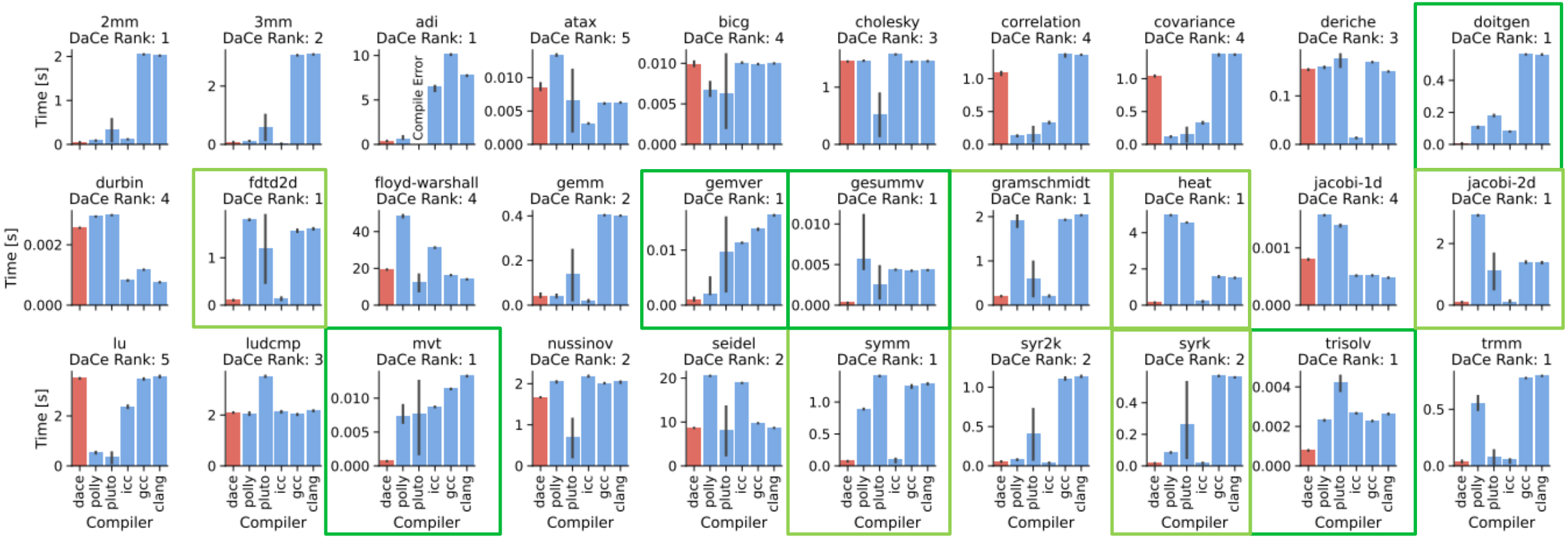
Results - Polybench



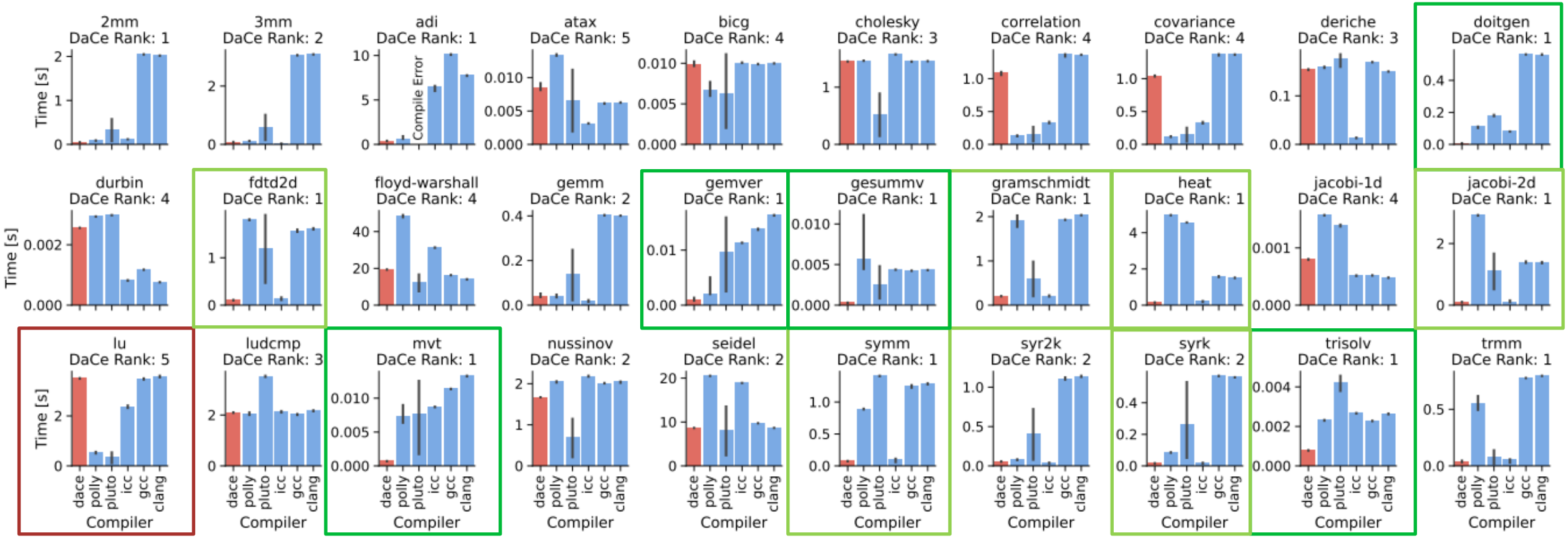
Results - Polybench



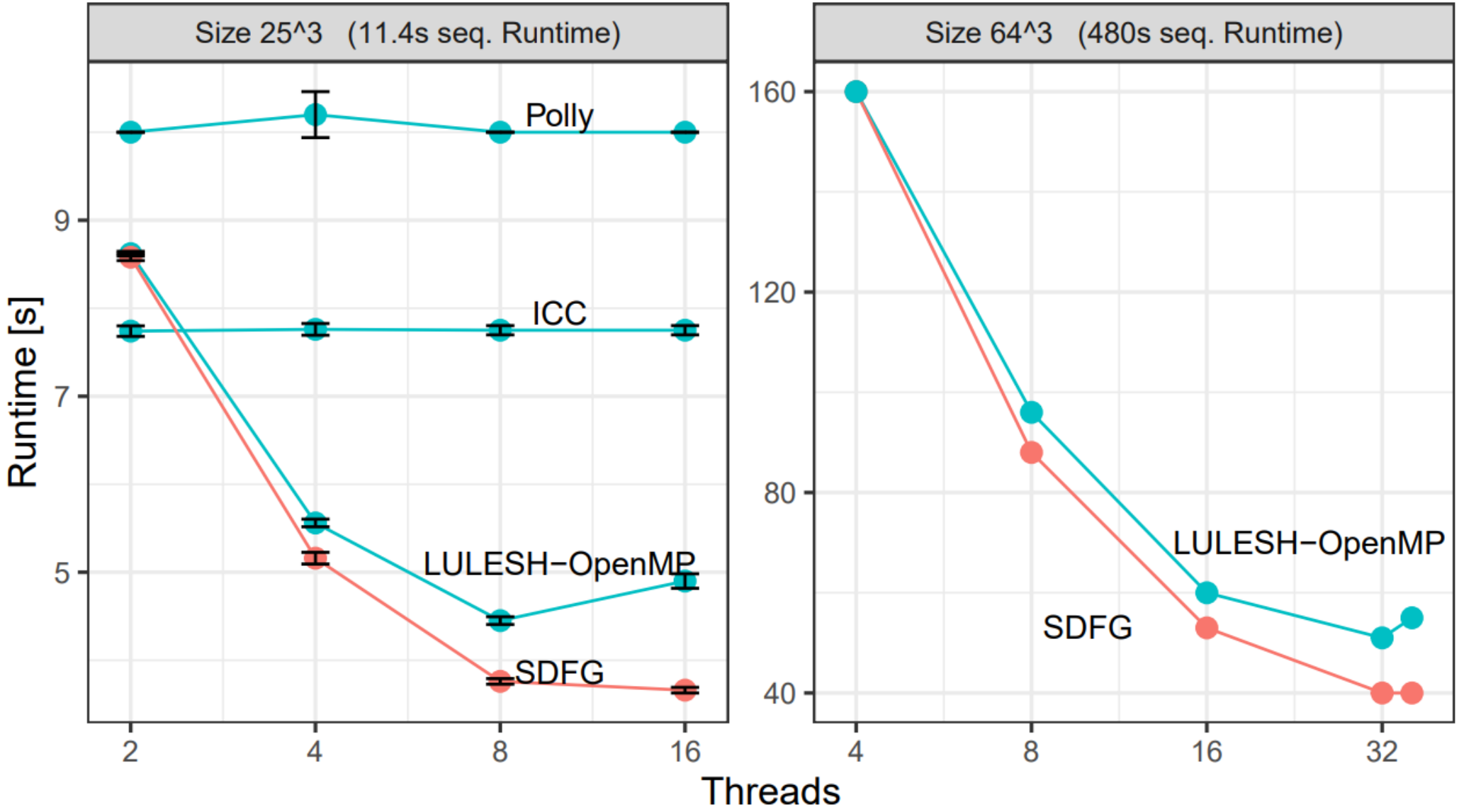
Results - Polybench



Results - Polybench

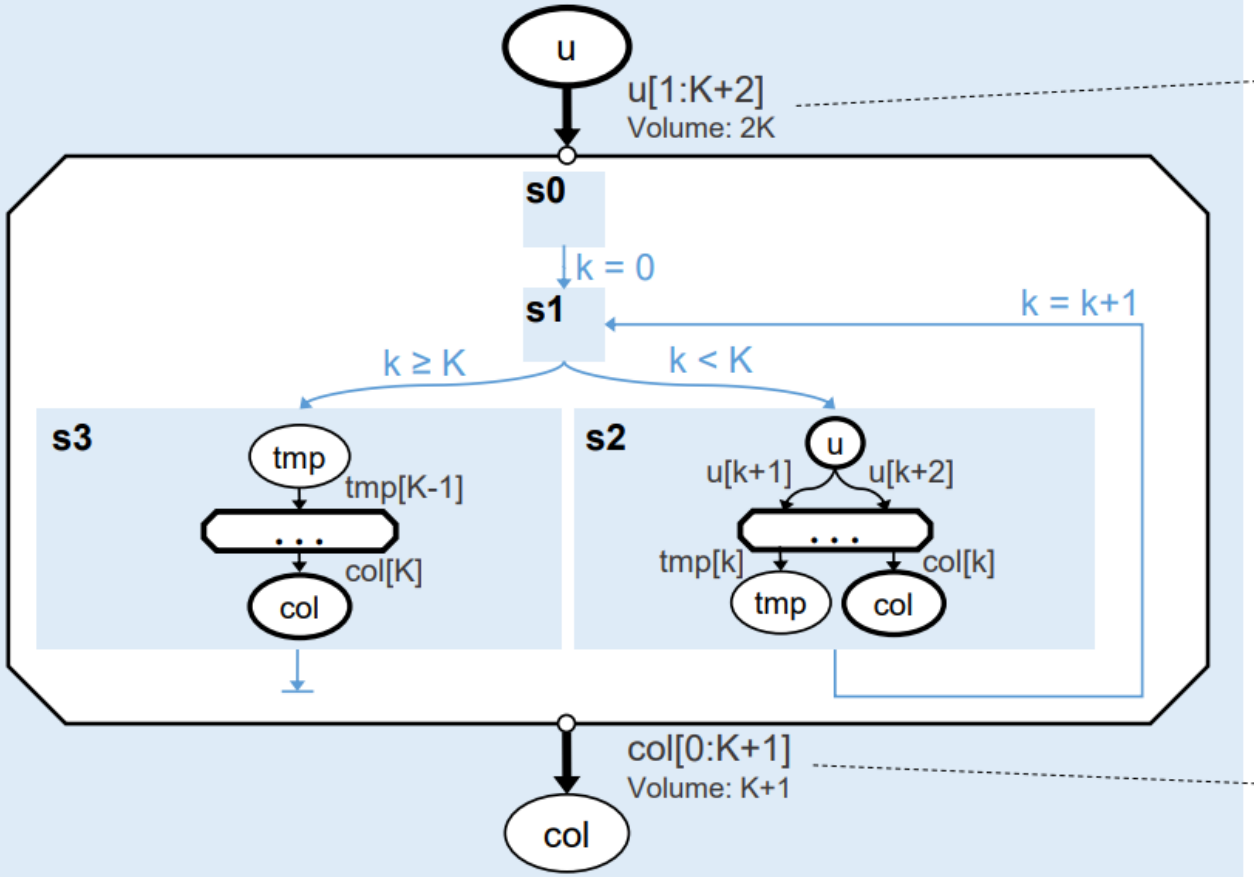


Results - LULESH



Thank you!

Access pattern propagation



Subset: $\bigcup_{k=0}^{K}\{k+1, k+2\}$
Volume: $2 \cdot exec(s2)$

s0:
Executions = 1
Symbol Ranges = ∅

s3:
Executions = 1
Symbol Ranges = {k: K}

s1, s2:
Executions = K
Symbol Ranges = {k: 0:K}

Subset: $\bigcup_{k=0}^{K}\{k\} \cup \{K\}$
Volume: $1 \cdot exec(s2) + 1 \cdot exec(s3)$

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