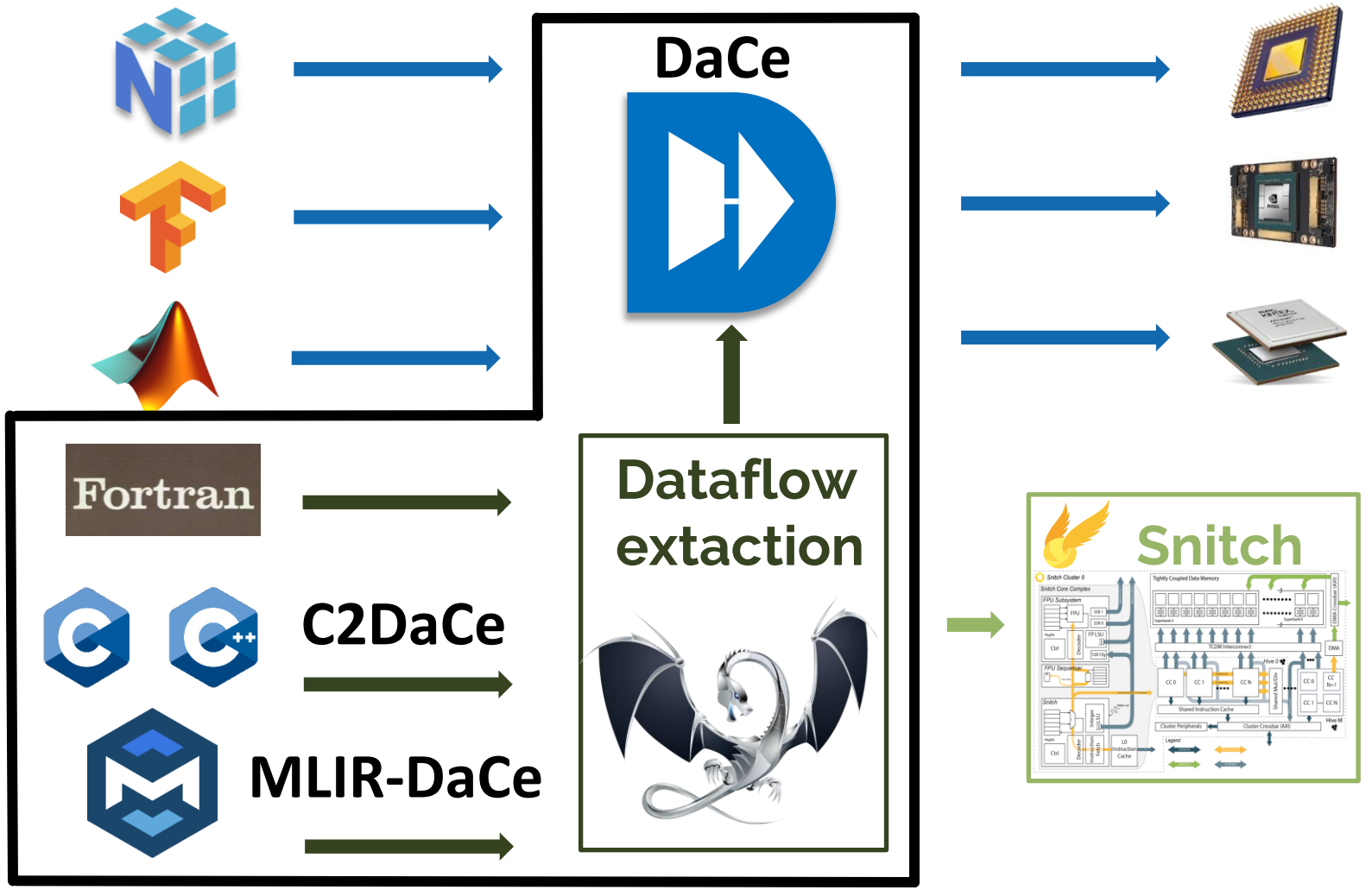
Alexandru Calotoiu,

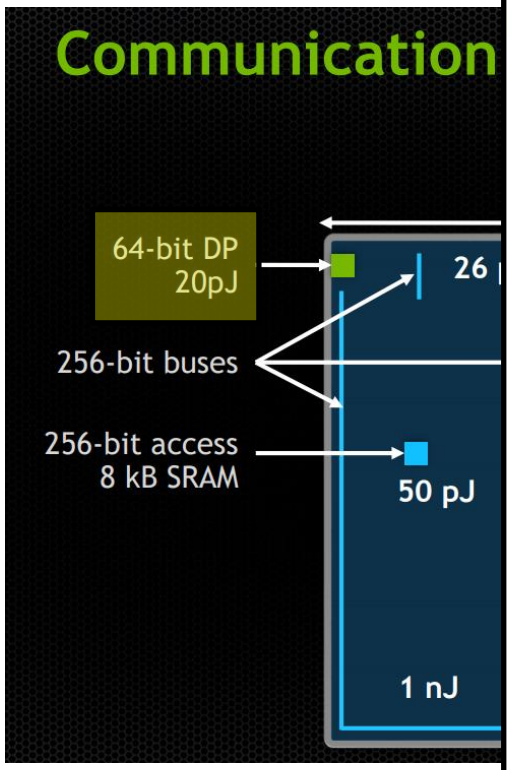
With contributions from Tal Ben-Nun, Berke Ates, Grzegorz Kwasniewski, Johannes de Fine Licht, Timo Schneider, Philipp Schaad, Alexandros Nikolas Ziogas, Marcin Copik, Torsten Hoefler

From Control-Centric Programming to Data-Centric Optimization

DaFlEx project summary



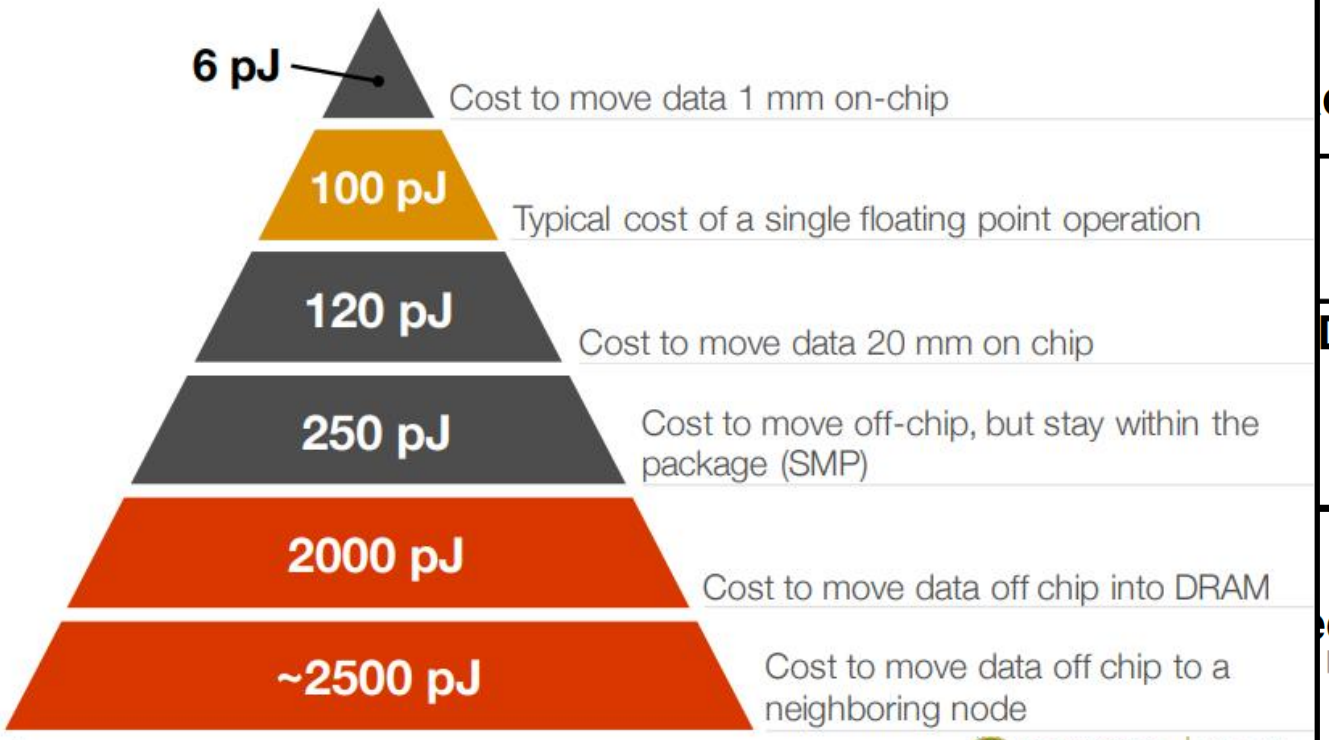
Data movement dominates energy costs!



Slide

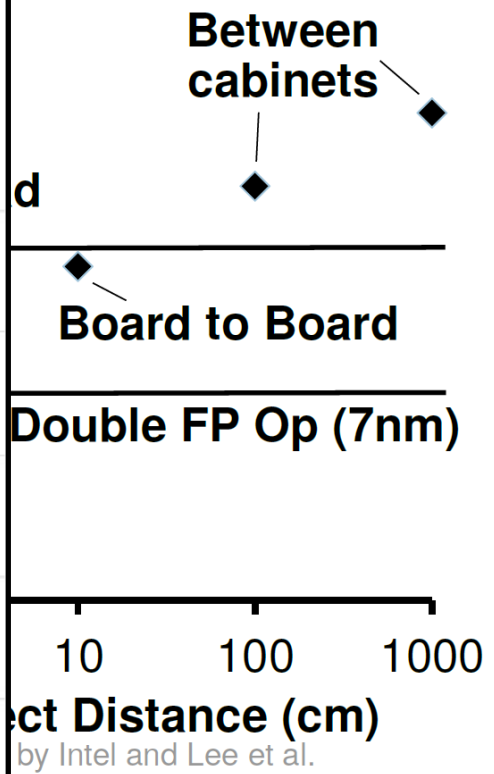
Hierarchical Power Costs

Data Movement is the Dominant Power Cost



Source: Fatollahi-Fard et al.

U.S. DEPARTMENT OF ENERGY | Office of Science | 10



by Intel and Lee et al.

Efficient data movement is hard!

Halide: Decoupli
Schedules fo
I

[redacted]

Polyhedral-Based Data Reuse Optimization for Configurable Computing

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

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

MLIR: A Compiler I
Mo

Chris La
Goog

Jacqu
G

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

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

reasingly important role in high-performance computing. While developing naive code is
ptimizing 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

First year recap – from C to DaCe

```
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];
```

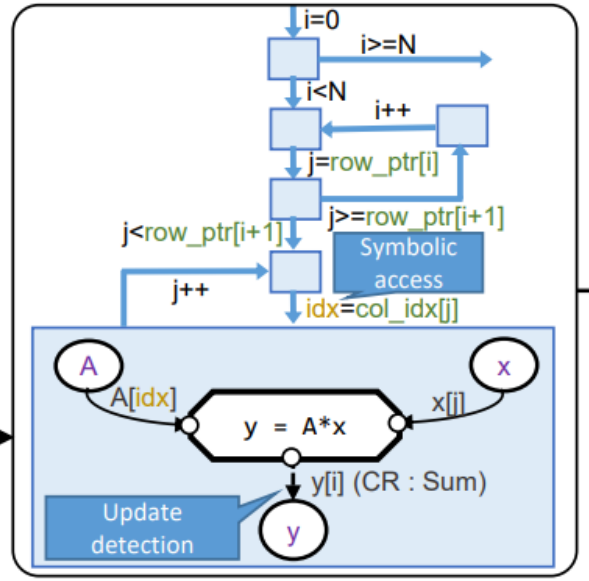
AST transformation (§2)

```
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];
    }
```

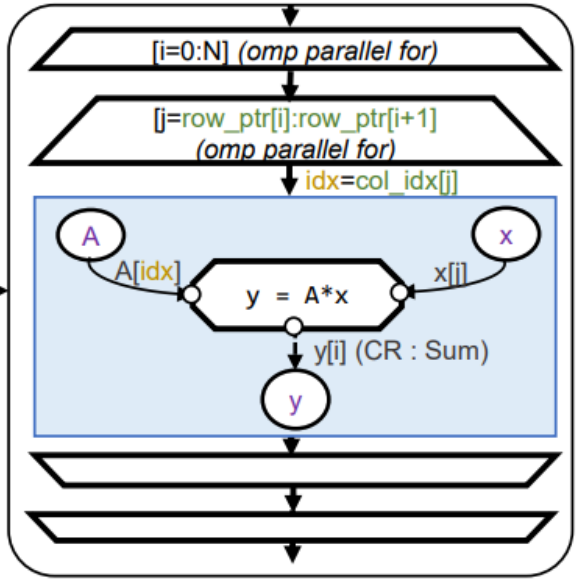
Index extraction

C-to-DaCe translation (§2)

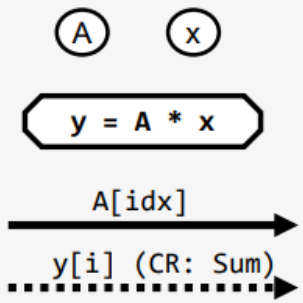
Dataflow coarsening (§3)



Dataflow optimization (§4)



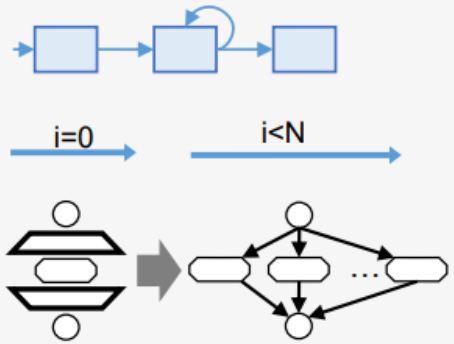
SDFG components



Data: Array containers

Tasklet: Fine-grained computation

Memlet: Data movement unit, with parallel write conflict resolution (CR) options



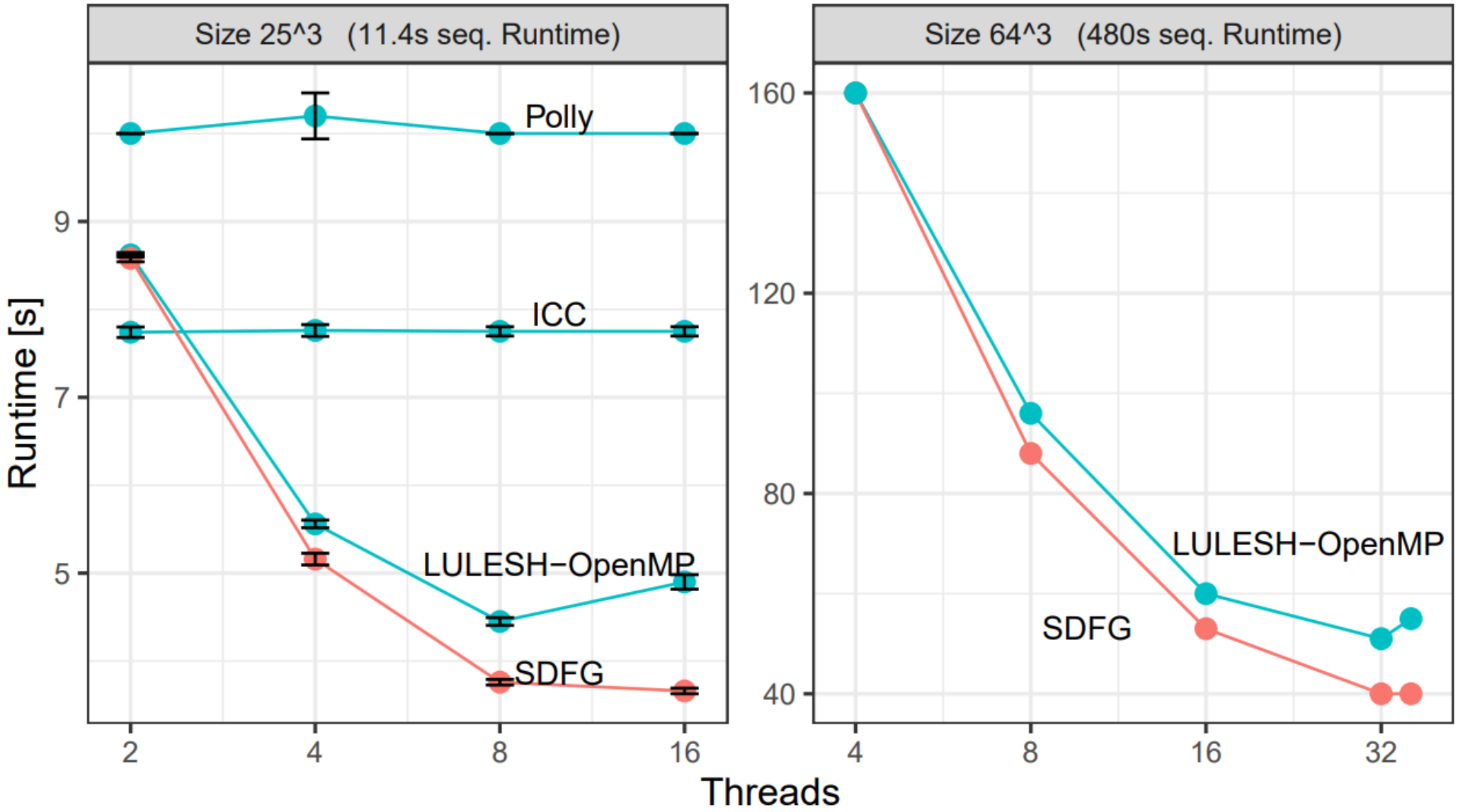
States: Control dependencies

Interstate edges: symbolic conditions and assignments dependencies

Map: Parametric parallelism scope

*Calotoiu et al: Lifting C semantics for dataflow optimization. (ICS '22) https://doi.org/10.1145/3524059.3532389

LULESH


Size 25^3 (11.4s seq. Runtime)
Size 64^3 (480s seq. Runtime)
Runtime [s]
Threads
Polly
ICC
LULESH−OpenMP
SDFG
9
7
5
160
120
80
40
2
4
8
16
32

Second year

Improving pointer analysis capabilities

Bridging control- and data-centric optimization

Performance portability without code changes

Still growing!

Advanced pointer analysis

C representation

```
p = a;
for (int i=0; i<n; i++) {
  p[0] = i + 1;
  p++;
}
p = a;
```

twin transformation

```
p = a;
for (int i=0; i<n; i++) {
  p[p_twin + 0] = i + 1;
  p_twin++;
}
p = a;
```

ASM representation

```
.L4:
  add     eax, 1      ; i++
  add     rdx, 4      ; p++
  mov     DWORD PTR [rdx-4], eax
  cmp     eax, r14d   ; i < n
  jne     .L4
```

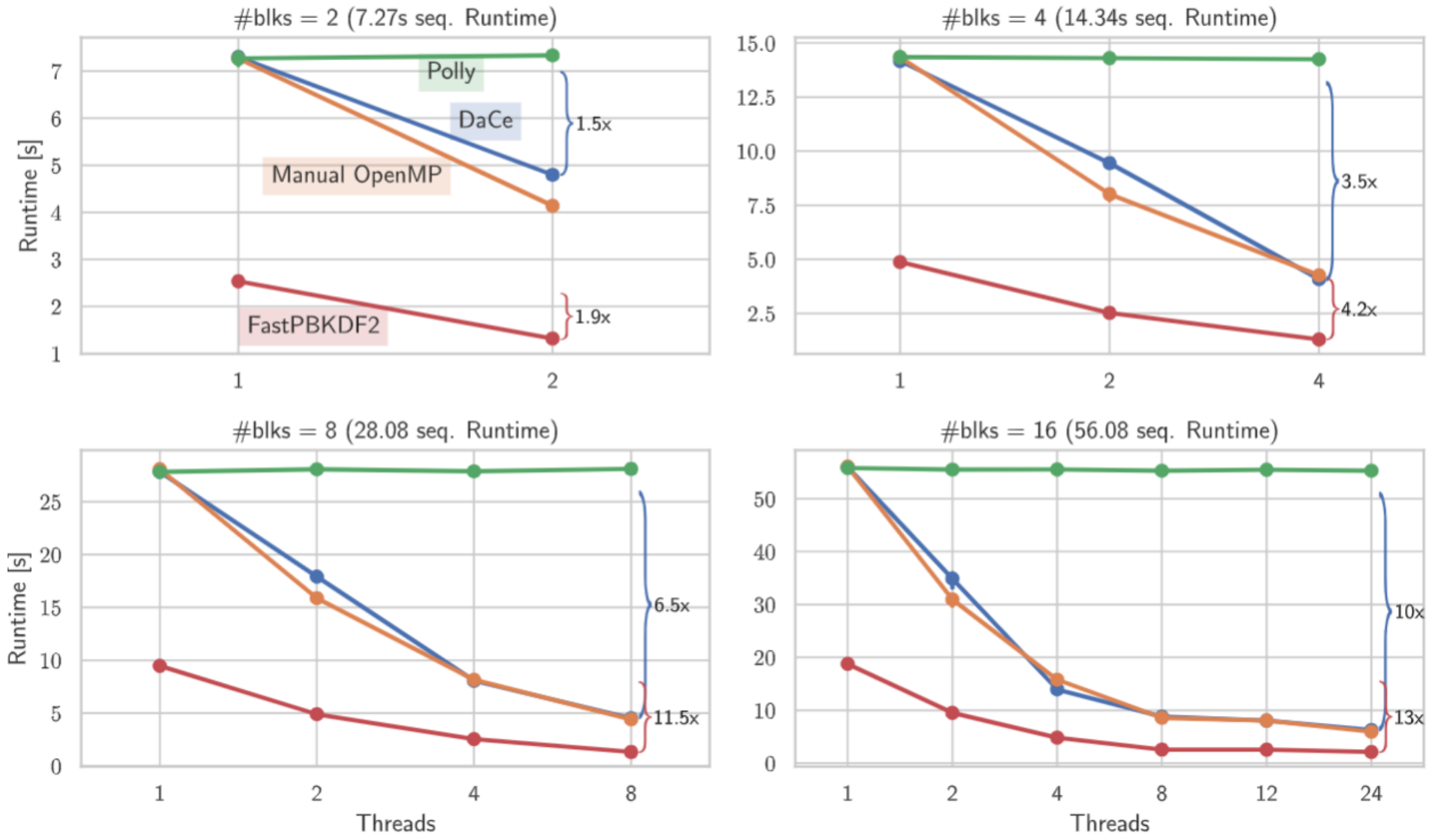
loop iterator → rax
pointer iterator → rdx
n value → r14d

No pointer increment
⇒ one less instruction

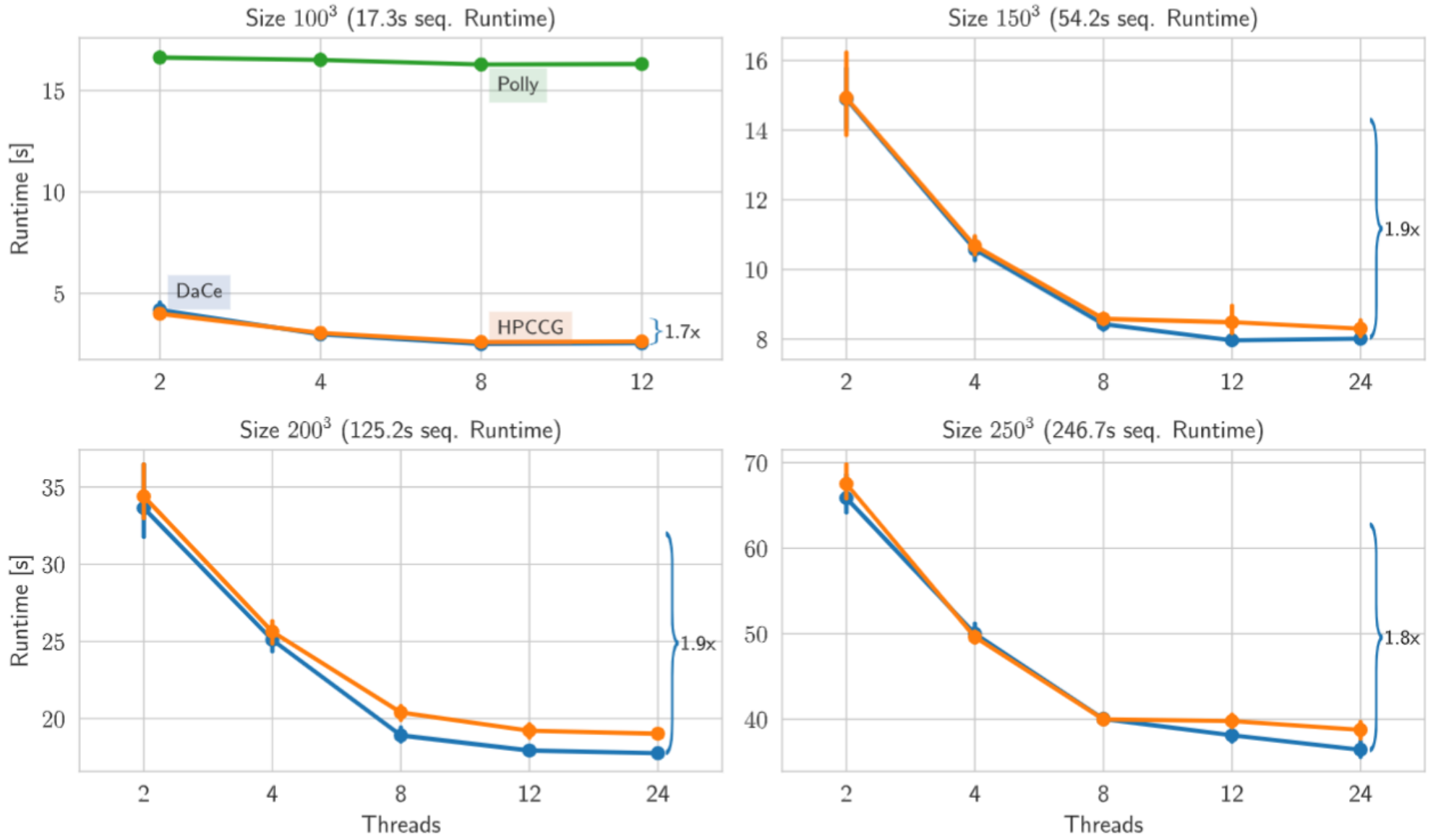
```
.L4:
  mov     DWORD PTR [r12-4+rax*4], eax
  add     rax, 1      ; i++
  cmp     rdx, rax    ; i < n
  jne     .L4
```

loop iterator → rax
pointer iterator ↛
container base address → r12
n value → rdx

PBKDF2


#blks = 2 (7.27s seq. Runtime)
Runtime [s]
Polly
DaCe
Manual OpenMP
FastPBKDF2
1.5x
1.9x
#blks = 4 (14.34s seq. Runtime)
3.5x
4.2x
#blks = 8 (28.08 seq. Runtime)
6.5x
11.5x
#blks = 16 (56.08 seq. Runtime)
10x
13x
Threads

HPCCG


Size 100^3 (17.3s seq. Runtime)
Size 150^3 (54.2s seq. Runtime)
Size 200^3 (125.2s seq. Runtime)
Size 250^3 (246.7s seq. Runtime)
Runtime [s]
Threads
Polly
DaCe
HPCCG
1.7x
1.9x
1.9x
1.8x

A new direction!

```c
int example() {
  int *A = (int *)malloc(100000 * sizeof(int));
  int *B = (int *)malloc(100000 * sizeof(int));
  for (int i = 0; i < 100000; ++i) {
    A[i] = 5;
    for (int j = 0; j < 100000; ++j)
      B[j] = A[i];
    for (int j = 0; j < 10000; ++j)
      A[j] = A[i];
  }
  int res = B[0];
  free(A);
  free(B);
  return res;
}
```

LICM

SIMD

Copy

```
int example() {
  int *A = (int *)malloc(100000 * sizeof(int));
  int *B = (int *)malloc(100000 * sizeof(int));
  for (int i = 0; i < 100000; ++i) {
    A[i] = 5;
    for (int j = 0; j < 100000; ++j)
      B[j] = A[i];
    for (int j = 0; j < 10000; ++j)
      A[j] = A[i];
  }
  int res = B[0];
  free(A);
  free(B);
  return res;
}
```

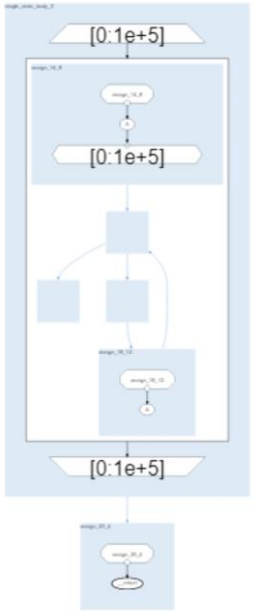
clang 15.0.0

Runtime: 1.54 s

```
8:
  %9 = phi i64 [ 0, %5 ], [ %34, %8 ], !dbg !253
  %10 = getelementptr inbounds i32, ptr %1, i64 %9, !dbg !255
  store <8 x i32> <i32 5, i32 5, i32 5, i32 5, i32 5, i32 5, i32 5, i32 5>, ptr %10, align 4, !dbg !256, !tbaa !242
...
  %33 = getelementptr inbounds i32, ptr %30, i64 24, !dbg !256
  store <8 x i32> <i32 5, i32 5, i32 5, i32 5, i32 5, i32 5, i32 5, i32 5>, ptr %33, align 4, !dbg !256, !tbaa !242
  %34 = add nuw nsw i64 %9, 160, !dbg !253
  %35 = icmp eq i64 %34, 100000, !dbg !253
  br i1 %35, label %39, label %8, !dbg !253, !llvm.loop !257
```

```
  %68 = load i32, ptr %7, align 4, !dbg !266, !tbaa !242
  %69 = getelementptr inbounds i32, ptr %1, i64 %67, !dbg !268
  store i32 %68, ptr %69, align 4, !dbg !269, !tbaa !242
  %70 = add nuw nsw i64 %40, 10, !dbg !270
```

LLVM IR



SDFG (DaCe IR)

```
int example() {
  int *A = (int *)malloc(100000 * sizeof(int));
  int *B = (int *)malloc(100000 * sizeof(int));
  for (int i = 0; i < 100000; ++i) {
    A[i] = 5;
    for (int j = 0; j < 100000; ++j)
      B[j] = A[i];
    for (int j = 0; j < 10000; ++j)
      A[j] = A[i];
  }
  int res = B[0];
  free(A);
  free(B);
  return res;
}
```

DaCe 0.14
+ C2DaCe

Runtime: 378 ms

Control-Centric

```
int example() {
  return 5;
}
```

Data-Centric

1.54 s

LICM, CSE, DCE

MCR, DME, DDE

0.38 s

```
for (int i = 0; i < 100000; ++i) {
  for (int j = 0; j < 100000/8; ++j)
    B[8*j..8*j+8] = 5;
  for (int j = 0; j < 10000; ++j)
    A[j] = A[i];
}
```

```
for (int i = 0; i < 100000; ++i) {
  A[i] = 5;
  for (int j = 0; j < 100000; ++j)
    B[j] = A[i];
}
```

Control-Centric

```
int example() {
  return 5;
}
```

Data-Centric

1.54 s

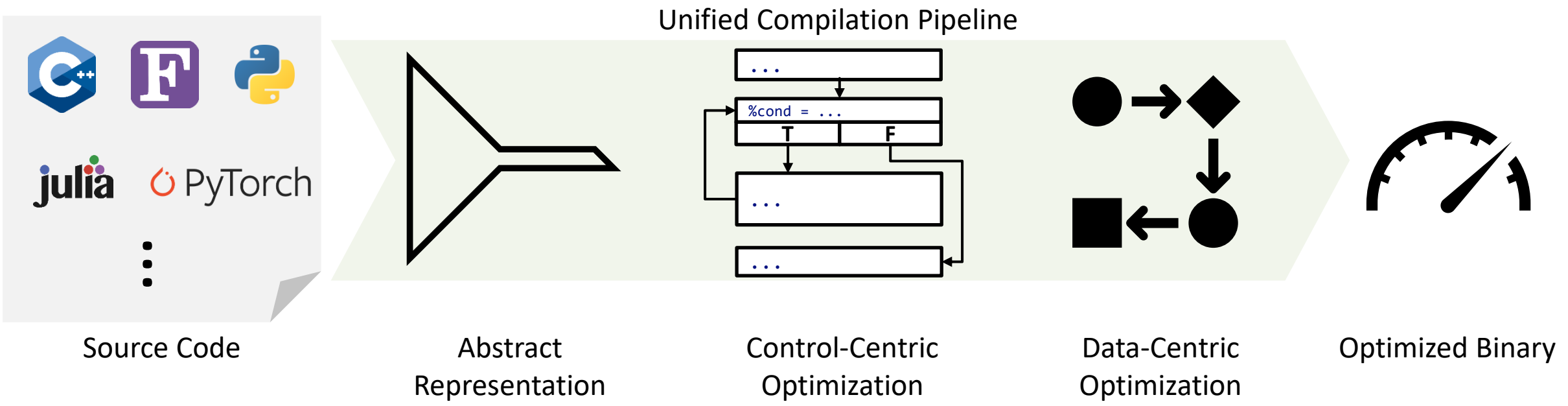
0.38 s

```
for (int i = 0; i < 100000; ++i) {
  for (int j = 0; j < 100000/8; ++j)
    B[8*j..8*j+8] = 5;
  for (int j = 0; j < 10000; ++j)
    A[j] = A[i];
}
```

```
for (int i = 0; i < 100000; ++i) {
  A[i] = 5;
  for (int j = 0; j < 100000; ++j)
    B[j] = A[i];
}
```

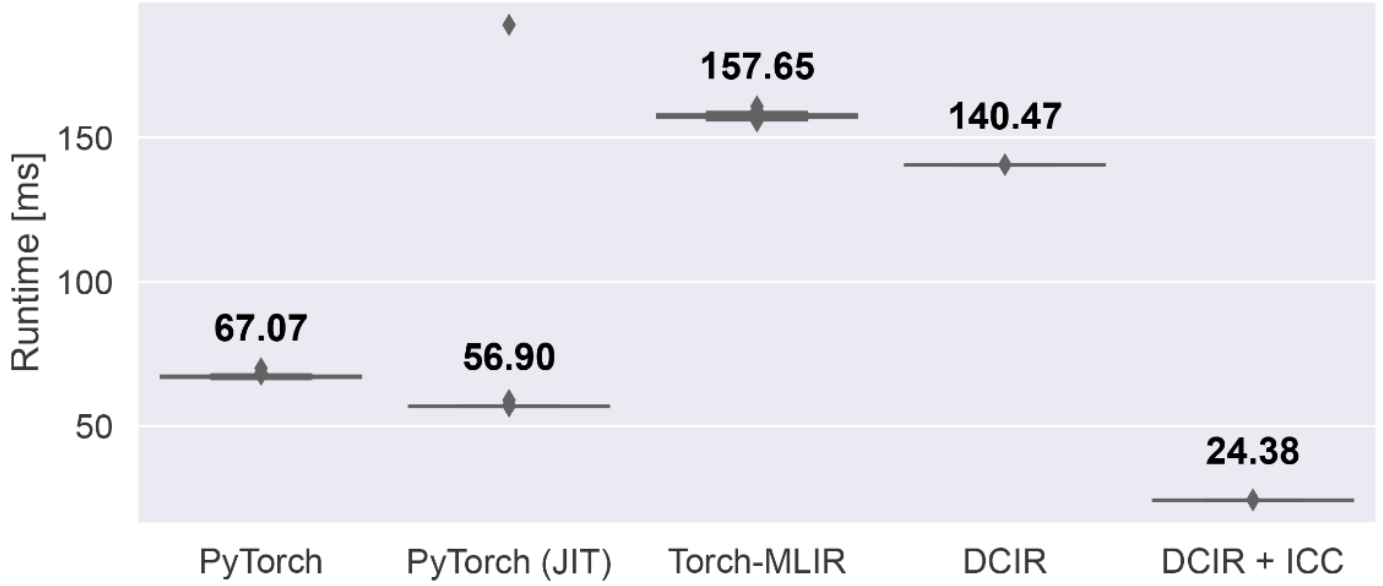
A single compiler pipeline that encapsulates both optimization classes

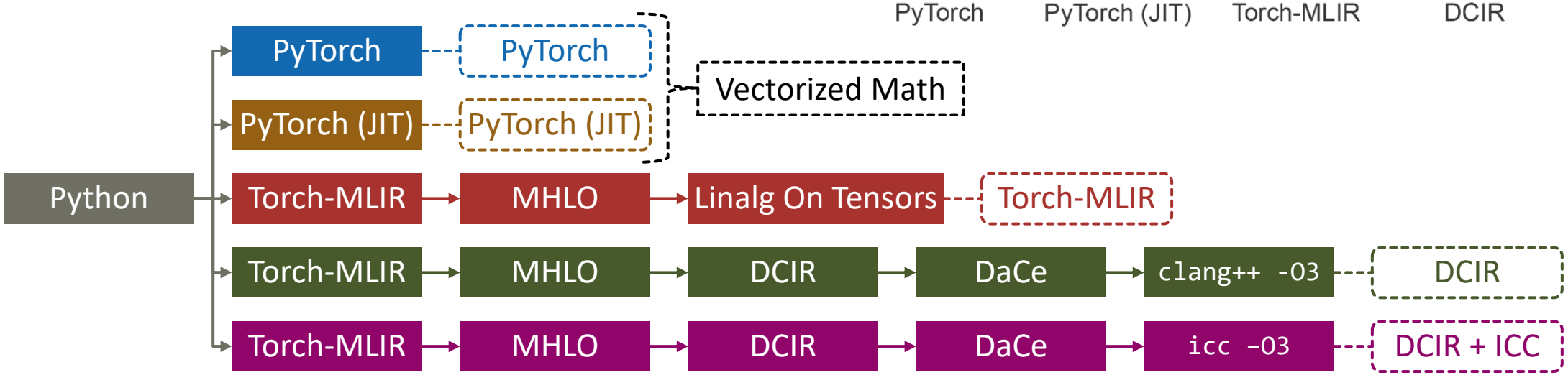
Overview


Unified Compilation Pipeline
julia
PyTorch
%cond = ...
T
F
Source Code
Abstract Representation
Control-Centric Optimization
Data-Centric Optimization
Optimized Binary

Mish Activation Function

```
class Mish(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        x = torch.log(1 + torch.exp(x))
        return x
```

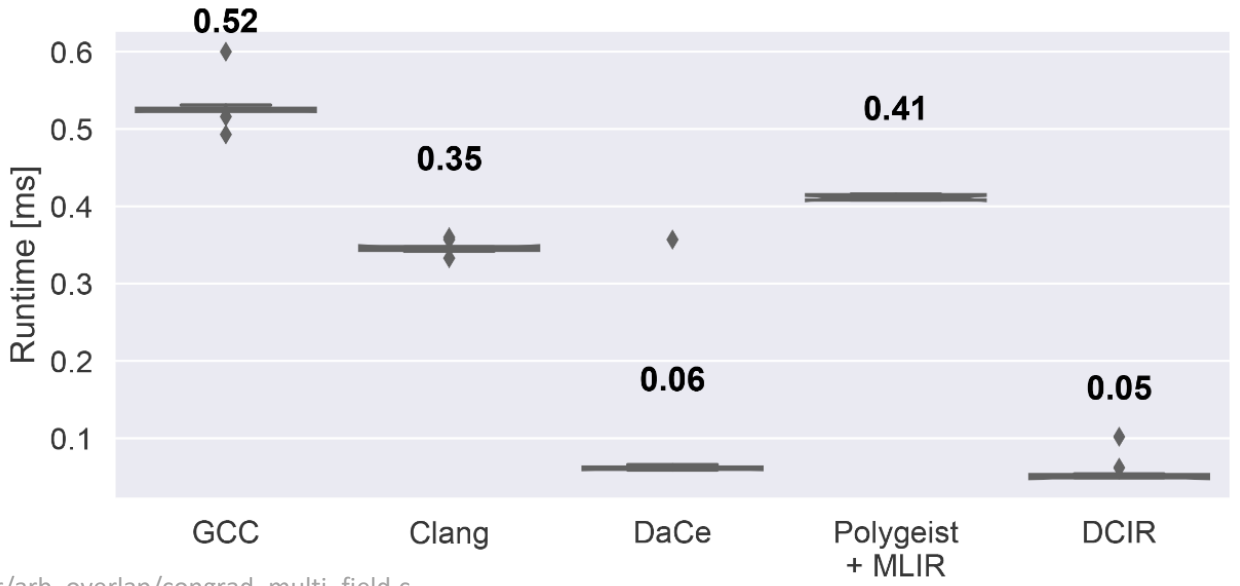




Quantum Chromodynamics (MILC)

Multi-Mass Conjugate Gradient Algorithm

```
[...]
  for (j = 1; j < Norder; j++) {
    if (converged[j] == 0) {
      zeta_ip1[j] = zeta_i[j] * zeta_im1[j] * beta_im1[0];
      c1 = beta_i[0] * alpha[0] * (zeta_im1[j] - zeta_i[j]);
      c2 = zeta_im1[j] * beta_im1[0] * (1.0 - (shift[j] - shift[0]) * beta_i[0]);
      zeta_ip1[j] /= c1 + c2;
      beta_i[j] = beta_i[0] * zeta_ip1[j] / zeta_i[j];
    }
  }
[...]
```



https://github.com/milc-qcd/milc_qcd/blob/master/arb_overlap/congrad_multi_field.c

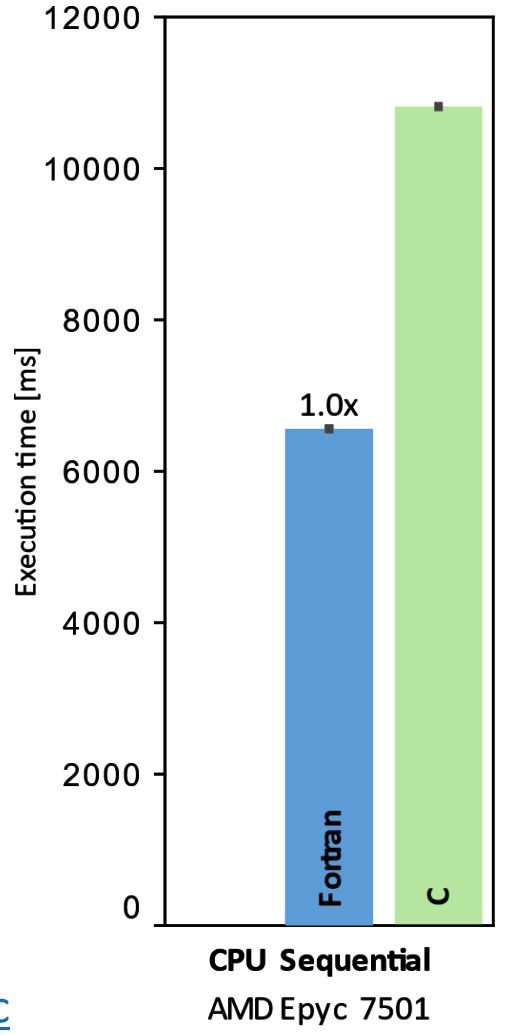
The Fortran frontend is here – and still growing!

Let's get to the real example … ECMWF's CLOUDSC

```
 9
10  SUBROUTINE CLOUDSC &
11   !---input
12   & (KIDIA,    KFDIA,    KLON,    KLEV, &
13   & PTSPHY,&
14   & PT, PQ, tendency_cml,tendency_tmp,tendency_loc, &
15   & PVFA, PVFL, PVFI, PDYNA, PDYNL, PDYNI, &
16   & PHRSW,    PHRLW,&
17   & PVERVEL,  PAP,      PAPH,&
18   & PLSM,     LDCUM,    KTYPE, &
19   & PLU,      PLUDE,    PSNDE,    PMFU,     PMFD,&
20   !---prognostic fields
21   & PA,&
22   & PCLV,  &
23   & PSUPSAT,&
24  !-- arrays for aerosol-cloud interactions
25  !!! & PQAER,    KAER, &
26   & PLCRIT_AER,PICRIT_AER,&
27   & PRE_ICE,&
28   & PCCN,     PNICE,&
29   !---diagnostic output
30   & PCOVPTOT, PRAINFRAC_TOPRFZ,&
31   !---resulting fluxes
32   & PFSQLF,   PFSQIF ,  PFCQNNG,  PFCQLNG,&
33   & PFSQRF,   PFSQSF ,  PFCQRNG,  PFCQSNG,&
34   & PFSQLTUR, PFSQITUR , &
35   & PFPLSL,   PFPLSN,   PFHPSL,   PFHPSN, KFLDX, &
36   & YDCST, YDTHF, YDECLDP)
```

… variable setup/initialization until line 500 ;-)

- **Cloud Microphysics of IFS**
  - Resolve sub-grid features
  - Original 2,525 SLOC of Fortran 95
- **Rewritten for performance portability benchmarking (optimization took months!)**
  - 2,635 SLOC C
  - 2,610 SLOC C++/CUDA



https://github.com/ecmwf-ifs/dwarf-p-cloudsc

A first simple loop from CLOUDSC*

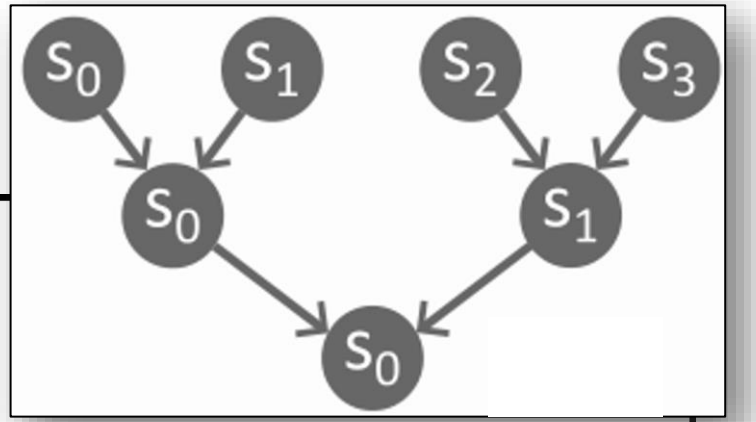
Data Parallelism	✓ ..... `do JK=1,KLEV`
	✓ ..... `do JL=1,KFDIA`
	`ZQSM(JL,JK)=ZQSM(JL,JK)/(1.0-RE*ZQSM(JL,JK))`
	`enddo`
	`enddo`

Work	KLEV * KFDIA
Depth	**1**
Average Parallelism	KLEV * KFDIA

Fully data parallel

* examples are simplified for presentation purposes

A second more complex loop from CLOUDSC



Data Parallelism	✗ ····· do JN=1, NSTEP-1 ✓ ····· do JL=1, KFDIA ZQXN(JL, NSTEP) = ZQXN(JL, NSTEP)+ZQXN(JL, JN) enddo enddo	(array) accumulation prevents parallelization ☹
Work	(NSTEP-1) * KFDIA	(NSTEP-1) * KFDIA
Depth	(NSTEP-1) * KFDIA	$\log_2$(NSTEP-1)
Average Parallelism	1	(NSTEP-1) * KFDIA / $\log_2$(NSTEP-1)

Now multiple realistic CLOUDSC loops

Data Parallelism

L1

```
X ..... do JM=1,4
X ..... do JK=1,KLEV
X ..... do JL=1,KFDIA
1   if ZQX(JL,JK,JM)<RLMIN) then
2     ZQADJ=ZQX(JL,JK,JM)*ZQTMST
3     tend_q(JL,JK)=tend_q(JL,JK)+ZQADJ
4     tend_T(JL,JK)=tend_T(JL,JK)-RAL*ZQADJ
5     ZQX(JL,JK,JM)=0.0
```

reuse of temporary variable prevents parallelization

L2

```
✓ ..... do JK=1,KLEV
✓ ..... do JL=1,KFDIA
6     ZQSM(JL,JK)=ZQSM(JL,JK)/(1.0-RE*ZQSM(JL,JK))
```

L3

```
✓ ..... do JK=1,KLEV
✓ ..... do JL=1,KFDIA
7     ZA(JL,JK)=MAX(0.0,MIN(1.0,ZA(JL,JK)))
8     ZLI(JL,JK)=ZQX(JL,JK,1)+ZQX(JL,JK,2)
9     if (ZLI(JL,JK)>RLMIN) then
10      ZLFRAC(JL,JK)=ZQX(JL,JK,1)/ZLI(JL,JK)
      else
11      ZLFRAC(JL,JK)=0.0
```

Order Constraints

Happens-before (→)

- 1 → 2 Control
- 2 → 3 RAW
- 2 → 4 RAW
- 2 → 5 WAR

No order constraint: L2 || L1

- 5 → 8 RAW
- 8 → 9 RAW
- 9 → 10 Control
- 9 → 11 Control

L3 → L1

	L1	L2	L3	L1	L2	L3
Work	4 * KLEV * KFDIA * (1+4)	KLEV * KFDIA	KLEV * KFDIA * 4	KLEV * KFDIA * 25		
Depth	log2(4) * 1 *1 * (1+2)	1	1 * 1 * (2+1)	8		
Average Parallelism	KLEV * KFDIA * 10/3	KLEV * KFDIA	KLEV * KFDIA * 4/3	KLEV * KFDIA * 25/8		

Performance Metaprogramming for Performance Portability

10s of SLOC

Domain Scientist

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

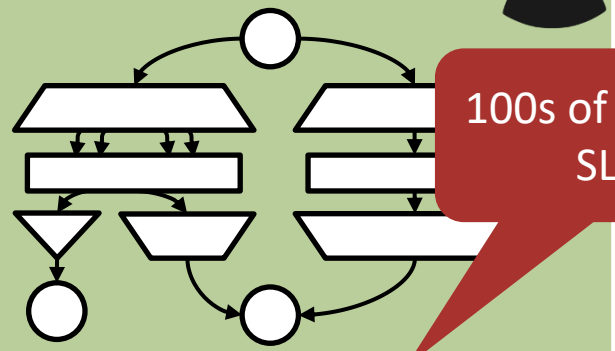
NumPy DSLs TensorFlow PyTorch MATLAB

Applied Scientist

translate DSL into parametric dataflow graphs

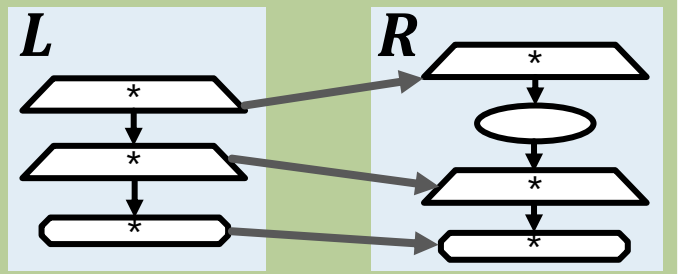
SDFG Builder API
Multi-Level Library Nodes

Performance Engineer



100s of reusable SLOC

Parametric Dataflow Graphs (SDFG)



Graph Transformations (API, Interactive)

Transformed Dataflow

Performance Results

1000s of auto-generated SLOC

Specialized Code Generation

Runtime

CPU Code

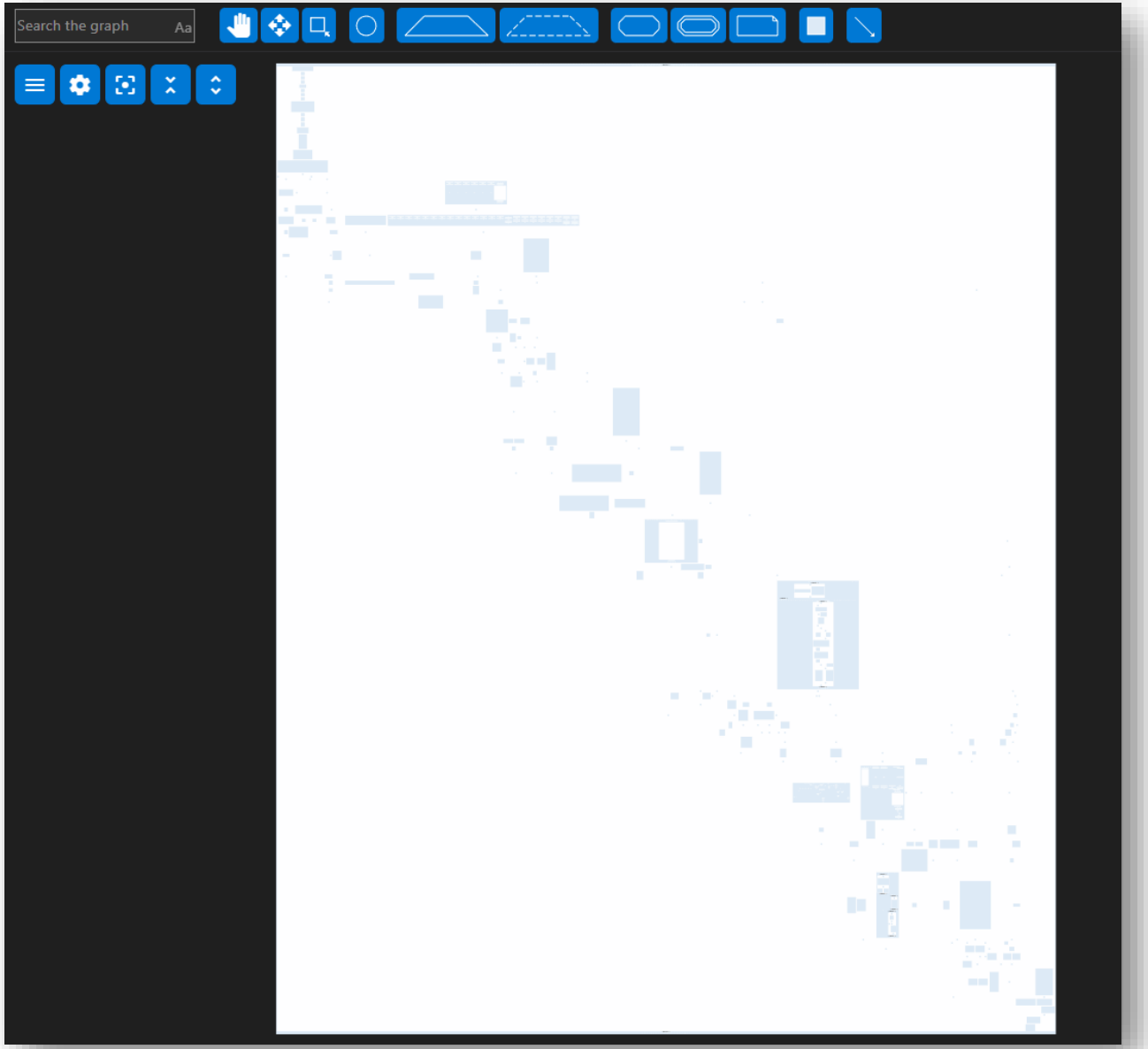
GPU Code

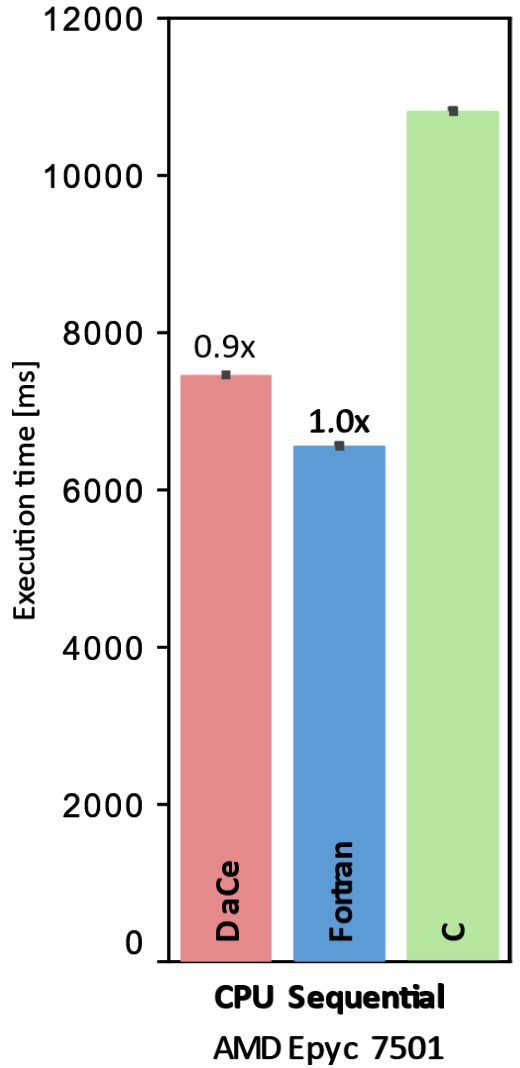
FPGA Code

C++ code generation/runtime

Ben-Nun, de Fine Licht, Ziogas, TH: Stateful Dataflow Multigraphs: A Data-Centric Model for High-Performance Parallel Programs, SC19

SDFG view of CLOUDSC – investigate dataflow





Transformations in DaCe – Performance Metaprogramming!

Expose parallelism

- Data management transformations:
  - Changing data container lifetime
  - Versioning data containers

1

- Loop Parallelization

2

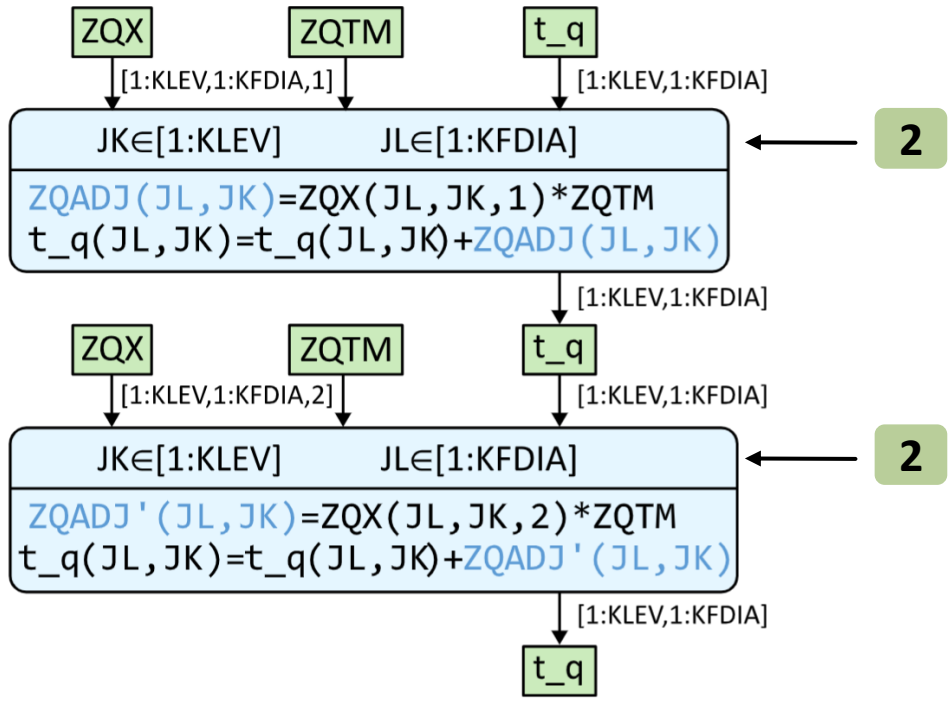
```
do JK=1,KLEV
  do JL=1,KFDIA
    ZQADJ=ZQX(JL,JK,1)*ZQTM
    t_q(JL,JK)=t_q(JL,JK)+ZQADJ
  enddo
enddo

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

1

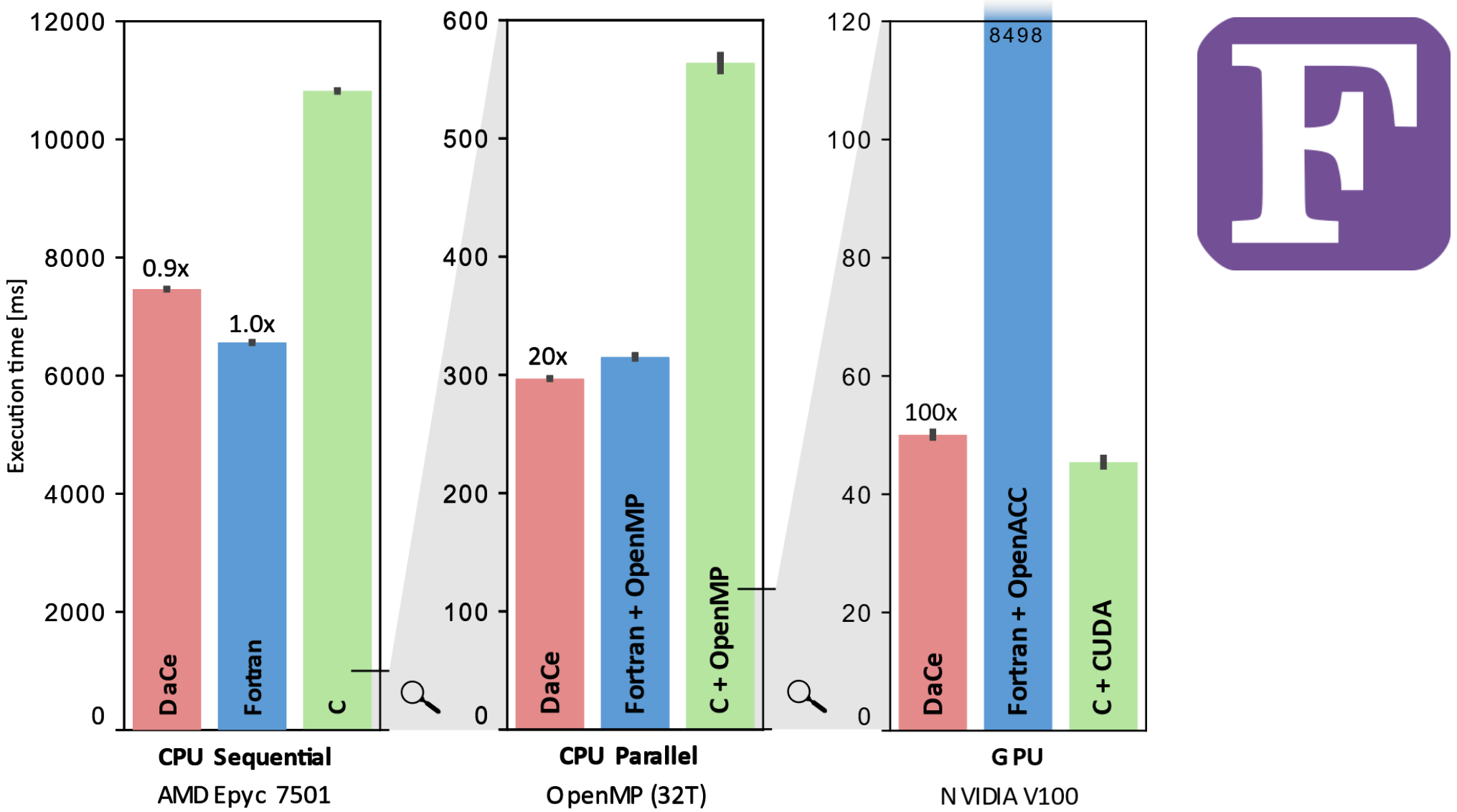
1

Improve performance

- Specializing numerical values
- Changing the data layout



Performance portability – all from the original, unchanged CLOUDSC code!



Conclusion

Understanding data movement is key to performance & portability