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

This project has received funding from the European Research Council (ERC) under grant agreement "DAPP (PI: T. Hoefler)".
Motivation
Computational Scientist
Domain Scientist  Performance Engineer
Optimization Techniques

▪ Multi-core CPU
  ▪ Tiling for complex cache hierarchies
  ▪ Register optimizations
  ▪ Vectorization

▪ Many-core GPU
  ▪ Coalesced memory access
  ▪ Warp divergence minimization, register tiling
  ▪ Task fusion

▪ FPGA
  ▪ Maximize resource utilization (logic units, DSPs)
  ▪ Streaming optimizations, pipelining
  ▪ Explicit buffering (FIFO) and wiring
## DaCe Overview

### Domain Scientist
- Problem Formulation
  \[
  \frac{\partial u}{\partial t} - \alpha \nabla^2 u = 0
  \]
- Python
- TensorFlow
- Scientific Frontend

### Performance Engineer
- Data-Centric Intermediate Representation (SDFG)
- Graph Transformations

### System
- Hardware Information
- Compiler
- CPU Binary
- GPU Binary
- FPGA Modules
Dataflow Programming in DaCe

\[ y = x^2 + \sin(x \pi) \]
Parallel Dataflow Programming

A


Tasklet

B[0]  B[1]  ...  B[N-1]

Tasklet

B
Parallel Dataflow Programming

![Diagram showing parallel dataflow programming with tasklets and scopes.](image-url)
Stateful Parallel Dataflow Programming
Stateful Parallel Dataflow Programming

State s0

\[
\begin{align*}
A & \quad [i=0:N] \\
\quad & \quad A[i] \\
\quad & \quad \text{Tasklet} \\
\quad & \quad B[i] \\
\quad & \quad [i=0:N] \\
B & \quad \quad [i=0:N] \\
\end{align*}
\]

State s1

\[
\begin{align*}
C & \quad [i=0:N] \\
\quad & \quad C[i] \\
\quad & \quad \text{Tasklet} \\
\quad & \quad A[i] \\
\quad & \quad [i=0:N] \\
A & \quad \quad [i=0:N] \\
\end{align*}
\]
Example: 2D Stencil

State s0

$[y=0: H, x=0: W]$  
Initialize  
$B[y, x]$  
$[y=0: H, x=0: W]$  
$B[0: H, 0: W]$  

State s1

$A[0: H, 0: W]$  
Jacobi  
$B[y, x]$  
$[y=0: H, x=0: W]$  
Jacobi  
$B[y-1, x] B[y, x-1] B[y, x+1] B[y+1, x]$  
$A[y, x]$  
$[y=0: H, x=0: W]$  
$A[0: H, 0: W]$  

$t = 0$  
$t < T; t++$  
$t \geq T$
Meet the Nodes

State
- State machine element

Tasklet
- Fine-grained computational block

Array
- N-dimensional data container

Map
- Parametric graph abstraction for parallelism

Stream
- Streaming data container

Consume
- Dynamic mapping of computations on streams

Exit
- Defines behavior during conflicting writes

Conflict Resolution
Meet the Nodes

Conflict Resolution

A2. Operational Semantics

2.1. Introductions. Notations. We denote collections (note lists) as capital letters and their members with the corresponding lowercase letters followed by a subscript. I.e., \( S(DG) = (S_1, \ldots, S_N) \) the set of states \( S = s_0 \), with \( i \in \{0, 1, \ldots, N\} \). Without loss of generality we assume \( s_0 \) to be the start state. We denote the value stored at memory location \( e \) as \( M(e) \), and assume all basic types are size-one elements to simplify address calculations.

The state of execution is denoted by \( s \). Within the state we carry several sets, i.e., the memory, a set of data nodes and transients in memory addresses, \( S \), which maps symbolic names (identifiers) to their current value, and \( T \), which maps transients to the data node at that connection in the current state of execution. We define a helper function \( \phi(e) \), which returns the product of all dimensions of the data node or element (using \( e \) to resolve symbolic vs values). Furthermore, \( \phi(e) \) returns the same shape of the data or data transient, and (1) the offset of a data element relative to the start of the memory region it is stored in.

The function \( \phi \) returns a copy of the given object as an argument. When we modify the copy, the original object remains the same.

A2.2. Propagating Data in a State. Execution of a state entails propagating data along edges governed by the rules defined below.

State Machine: Operations. Once a state reaches execution, all unpopulated state transitions of that state are evaluated in an arbitrary order, and the destination of the first transition whose condition is true is the next state which will be evaluated. If no transition evaluates true, the program terminates. Before starting the execution of the next state, all assignments are performed, and the left-hand side of assignments become symbols.

A2. Operational Semantics

2.1. Introductions. Notations. We denote collections (note lists) as capital letters and their members with the corresponding lowercase letters followed by a subscript. I.e., \( S(DG) = (S_1, \ldots, S_N) \) the set of states \( S = s_0 \), with \( i \in \{0, 1, \ldots, N\} \). Without loss of generality we assume \( s_0 \) to be the start state. We denote the value stored at memory location \( e \) as \( M(e) \), and assume all basic types are size-one elements to simplify address calculations.

The state of execution is denoted by \( s \). Within the state we carry several sets, i.e., the memory, a set of data nodes and transients in memory addresses, \( S \), which maps symbolic names (identifiers) to their current value, and \( T \), which maps transients to the data node at that connection in the current state of execution. We define a helper function \( \phi(e) \), which returns the product of all dimensions of the data node or element (using \( e \) to resolve symbolic vs values). Furthermore, \( \phi(e) \) returns the same shape of the data or data transient, and (1) the offset of a data element relative to the start of the memory region it is stored in.

The function \( \phi \) returns a copy of the given object as an argument. When we modify the copy, the original object remains the same.

A2.2. Propagating Data in a State. Execution of a state entails propagating data along edges governed by the rules defined below.

State Machine: Operations. Once a state reaches execution, all unpopulated state transitions of that state are evaluated in an arbitrary order, and the destination of the first transition whose condition is true is the next state which will be evaluated. If no transition evaluates true, the program terminates. Before starting the execution of the next state, all assignments are performed, and the left-hand side of assignments become symbols.
Hierarchical Parallelism and Heterogeneity

- Maps have schedules, arrays have storage locations

\[
\begin{align*}
\text{out} &= \text{in}_A \times \text{in}_A
\end{align*}
\]
Hierarchical Parallelism and Heterogeneity

- Maps have schedules, arrays have storage locations

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

    for (int ti = 0; ti < TN; ti += 1) {
        vec<double, 4> in_A = tA[ti];
        auto out = (in_A * in_A);
        tC[ti] = out;
    }
```

...
Hierarchical Parallelism and Heterogeneity

\[ \text{out} = \text{in}_A \times \text{in}_A \]
Hierarchical Parallelism and Heterogeneity

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

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

    vec<double, 2> in_A = tA[ti];
    auto out = (in_A * in_A);
    tC[ti] = out;
}
Hardware Mapping: Load/Store Architectures

- Recursive code generation (C++, CUDA)
  Control flow: Construct detection and gotos

- Parallelism
  Multi-core CPU: OpenMP, atomics, and threads
  GPU: CUDA kernels and streams
  Connected components run concurrently

- Memory and interaction with accelerators
  Array-array edges create intra-/inter-device copies

```c++
#pragma omp parallel for
for (int i = 0; i < N; i += TN) {
    vec<double, 4> tA[TN];
    Global2Stack_1D<double, 4, 1> (
        &A[i], min(N - i, TN), tA);
    for (int ti = 0; ti < TN; ti += 1) {
        vec<double, 4> in_A = tA[ti];
        auto out = (in_A * in_A);
        tC[ti] = out;
    }
}
```
Mapping to Reconfigurable Hardware

- Module generation with HDL and HLS
  - Xilinx SDAccel
  - Intel FPGA (experimental)

- Parallelism
  - Exploiting temporal locality: pipelines
  - Exploiting spatial locality: vectorization, replication

- Replication
  - Enables parametric systolic array generation
Data-centric Parallel Programming for Python

- **Programs** are integrated within existing codes
  - In Python, integrated functions in existing code
  - In MATLAB, separate .m files
  - In TensorFlow, takes existing graph

- In **Python**: Implicit and Explicit Dataflow
  - Implicit: numpy syntax
  - Explicit: Enforce memory access decoupling from computation

- **Output compatible with existing programs**
  - C-compatible SO/DLL file with autogenerated include file

```python
@dace.program
def program_explicit(A, B):
    @dace.map
def transpose(i: [0:N], j: [0:M]):
        a << A[i,j]
        b >> B[j,i]
        b = a

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

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

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

out = in_A * in_B

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

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

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

        out = in_A * in_B

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

my_tasklet

$A[\text{ar}]$

$
\ast$

$A[:,]$

$A[\cdot]$

\text{$\text{arr}$}

\text{$\text{arr}$}

\text{$\text{arr}$}

$\text{$\text{my\_tasklet}$}$

$\text{$\text{my\_tasklet}$}$

$B[\text{ar}]$

$\ast$

$B[\text{br}]$

$\text{B}$

\text{$\text{arr}$}

\text{$\text{arr}$}

\text{$\text{arr}$}

$B$

$B$
Programming Model Challenges

Indirect memory access

Nested state machines
DIODE (or: Data-centric Integrated Optimization Development Environment)
DIODE (or: Data-centric Integrated Optimization Development Environment)
Performance

SDFG
Performance

SDFG

MapReduceFusion

Naïve

Problem Size

Performance [GFlop/s]
Performance

SDFG

LoopReorder
MapReduceFusion

Naive
Performance

SDFG

BlockTiling
LoopReorder
MapReduceFusion
Performance

SDFG

RegisterTiling
BlockTiling
LoopReorder
MapReduceFusion

Performance [GFlop/s]

Problem Size

Naive
Performance

- LocalStorage
- RegisterTiling
- BlockTiling
- LoopReorder
- MapReduceFusion

SDFG

Graph representation of an SDFG.
Performance

- Promote Transient
- Local Storage
- Register Tiling
- Block Tiling
- Loop Reorder
- MapReduce Fusion

SDFG

![Graphical representation of performance metrics vs. problem size]

Performance [GFlop/s] vs. Problem Size

- Promote Transient
- Local Storage
- Register Tiling
- Block Tiling
- Loop Reorder
- MapReduce Fusion

Naïve
Performance

- Intel MKL
- DaCe
- OpenBLAS

25% difference

With tuning: 98.6% of MKL
Intel Xeon E5-2650 v4

NVIDIA Tesla P100

Xilinx VU9P

SDFG

General Compilers

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

Polyhedral Optimizers

Polly 6, Pluto 0.11.4, PPCG 0.8

Frameworks & Libraries

HPX, Halide, Intel MKL, CUBLAS, CUSPARSE, CUTLASS, CUB
Performance Evaluation: Fundamental Kernels (CPU)

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

**Matrix Multiplication (MM):** 2048x2048x2048

**Histogram:** 8192x8192

**Jacobi stencil:** 2048x2048 for T=1024

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

![Graph showing the performance comparison of different kernels.](image)
Performance Evaluation: Fundamental Kernels (GPU, FPGA)

90% of CUTLASS

19.5x of Spatial

GPU

FPGA
Performance Evaluation: Fundamental Kernels (GPU, FPGA)
Performance Evaluation: Polybench (CPU)

- Polyhedral benchmark with 30 applications

- Without any transformations, achieves 1.43x (geometric mean) over general-purpose compilers
Performance Evaluation: Polybench (GPU, FPGA)

- Automatically transformed from CPU code

**GPU**

(1.12x geometric speedup)

**FPGA**

The first full set of placed-and-routed Polybench
Case Study: Parallel Breadth-First Search

- Compared with Galois and Gluon

- Graphs:
  - Road maps: USA, OSM-Europe
  - Social networks: Twitter, LiveJournal
  - Synthetic: Kronecker Graphs

![Graphs](image-url)
Conclusions

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

pip install dace

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