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

Communication Dominates Arithmetic

64-bit DP 20pJ
256-bit buses
256-bit access 8 KB SRAM

1 nJ 256 pJ 16 nJ
28nm CMOS

20mm

26 pJ

Efficient off-chip link

500 pJ

Slide courtesy of NVIDIA
Computational Scientist

Source: US DoE
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
  - DSLs
  - MATLAB

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

**System**
- Hardware Information
- Compiler
- CPU Binary
- GPU Binary
- FPGA Modules

**Runtime**
- Transformed Dataflow
- Performance Results
Dataflow Programming in DaCe

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

Diagram showing a dataflow graph with tasks labeled A[0], A[1], ..., A[N-1], B[0], B[1], ..., B[N-1] connected by directed edges.
Parallel Dataflow Programming

A\[0\:N\]

Tasklet

B\[0\:N\]

Scope

\[i=0:N\]
Tasklet
B\[i\]

}\ [i=0:N]
Stateful Parallel Dataflow Programming

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

\[
\begin{align*}
C[i] & \rightarrow \text{Tasklet} \\
A[i] & \rightarrow \text{Tasklet} \\
A[0:N] & \rightarrow B[0:N]
\end{align*}
\]
Stateful Parallel Dataflow Programming

State s0

\[
\begin{align*}
A & \rightarrow A[0:N] \\
& \left[i=0:N\right] \\
& \downarrow A[i] \\
\text{Tasklet} & \\
& \downarrow B[i] \\
& \left[i=0:N\right] \\
& \downarrow B[0:N] \\
B &
\end{align*}
\]

State s1

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

State \( s_0 \)

\[
[y=0:H, x=0:W] \\
\text{Initialize} \\
[y=0:H, x=0:W] \\
B[y,x] \\
B[0:H,0:W]
\]

\[
t=0 \\
B[0:H,0:W] \\
A[y-1,x] \\
A[y,x-1] \\
A[y,x+1] \\
A[y+1,x]
\]

\[
t < T; t++
\]

State \( s_1 \)

\[
[y=0:H, x=0:W] \\
\text{Jacobi} \\
B[y,x] \\
B[0:H,0:W] \\
A[y-1,x] \\
A[y,x-1] \\
A[y,x+1] \\
A[y+1,x]
\]

\[
t \geq T \\
A[0:H,0:W] \\
B[0:H,0:W] \\
B[0:H,0:W] \\
B[0:H,0:W] \\
A[0:H,0:W]
\]
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**

- **Conflict Resolution**
  - Defines behavior during conflicting writes
Meet the Nodes

State machine elements

- State: state machine element
- Tasklet: fine-grained computational block
- Array: N-dimensional data container
- Stream: streaming data container
- Consume: exit
- Dynamic mapping of computations on streams

Elemental Processing: In each step, we take one element of a stream as a state machine element, or a node of current, for which all input connections have valid data. If x is a data node, update its internal memory for an input connection a, i.e., a(b) = A[i]. If x is a tasklet, generate a path that directs local variables for all input connectors: a0 ⇒ b0 ⇒ an ⇒ bn. As an output connector, b generates an output tuple, which updates b(x) = b(x) + 1 for each input connector a of b with the contents of the appropriate variable (delayed in P). Delete the consumption of (P1, P2, e0) for each (P, P1, e) in state. If x is a memory node with size x. If a is the identifier and size = size + 1. Assume a stream space, P as the corresponding input node form. For each element x in E, replace x by x in state.

State s0

A

[i=0:N]

A[i]

Filter

Bsize(+) S

[i=0:N]

S

S

B

Bsize(+) B[0:N]

Conflict Resolution

Defines behavior during conflicting writes

Static Dataflow Multigraph

- Atom symbol • operation: Once a state finishes execution, all unapplied state transitions of that state are evaluated in an arbitrary order, and the destruction of the last transition whose condition is true is the next action, which will be executed. If no transition evaluates to 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.

A.2 Operational Semantics

A.2.1 Initiation: Notation: We denote collections (note lists) as capital letters and their members with the corresponding lowercase letters and a subscript i, i.e., in S(0, T), a is the set of states a as a[i]. Without loss of generality we assume a0 to be the start state. We denote the value stored in memory location e at state s by M(e) and assume all basic types are size-one elements to simplify address calculations.

The state of execution is denoted by p. Within the state we can observe state transitions, which may update data nodes and transition to new state addresses, or which may apply symbol names (identifiers) to state transition values, and in which, maps connect the data node to that connection in the current state of execution.

We define a function (map), which returns the product of all dimensions of the data node or element (using e to render symbolic values). Furthermore, all returns the same number of a data or transition node, and offset the offset of a data element relative to the start of the memory region it is stored in. The function apply returns a copy of the element given as argument. When we modify the copy, the original object remains the same.

1. When an SODC G is called with the data arguments A = A[i] = A[i]: jn, as an identifier, jn is an address pointer), and unapplied arguments f_j = [f_i = f_i], as an integer) we initialize the configuration to.

A.2.2 Propagating Data in a State: Execution of a state entails propagating data along edges, governed by the rules defined below.
Hierarchical Parallelism and Heterogeneity

- Maps have schedules, arrays have storage locations

```
[i=0:N:TN]
A[i:i+TN]
A[0:N]

CPU

[t=0:N]
Core

A[i+ti]
A[i:ti+TN]

out = in_A * in_A
```

...
Hierarchical Parallelism and Heterogeneity

- Maps have schedules, arrays have storage locations

```cpp
// ...
#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

\[ \begin{align*} &A[i=0:N:TN] \\
&\text{CPU} \\
&\downarrow \quad A[i:i+TN] \\
&tA \\
&\downarrow \quad tA[0:TN] \\
&\text{Core} \\
&\downarrow \quad tA[ti] \\
&\text{out} = \text{in}_A * \text{in}_A \\
&\ldots \end{align*} \]
Hierarchical Parallelism and Heterogeneity

```cpp
__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_numpy(A, B):
    B[:] = np.transpose(A)
```

```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
```
Matrix Multiplication SDFG

```python
@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

```
A[$ar]
* 
A[:]
   
A[::]
  
my_tasklet

B[$br]
  
my_tasklet

B[$ar]

A

B

X

arr

arr
```
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

MapReduce Fusion

Naive
Performance

SDFG

Naive

LoopReorder MapReduceFusion

Problem Size

Performance [GFlop/s]
Performance

![SPCL Diagram]

BlockTiling
LoopReorder
MapReduceFusion

SDFG
Performance

RegisterTiling
BlockTiling
LoopReorder
MapReduceFusion

Naive

Problem Size
Performance [GFlop/s]
Performance

SDFG

LocalStorage
RegisterTiling
BlockTiling
LoopReorder
MapReduceFusion

Performance [GFlop/s]

Problem Size
Performance

SDFG

Promote Transient
LocalStorage
Register Tiling
Block Tiling
Loop Reorder
Map Reduce Fusion

Naïve

Performance [GFlop/s]

Problem Size
Performance

![Graph showing performance comparison between Intel MKL, DaCe, and OpenBLAS. The graph indicates a 25% difference in performance between Intel MKL and DaCe with tuning, and that DaCe achieves 98.6% of MKL's performance.]
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)

8.12x faster  98.6% of MKL  2.5x faster  82.7% of Halide  99.9% of MKL
Performance Evaluation: Fundamental Kernels (GPU, FPGA)

GPU

90% of CUTLASS

FPGA

19.5x of Spatial
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

![Chart showing comparison between kron, osmeur, socij, twitter, and usa with Time [s] on the y-axis and SDFG, Galois, and Gluon on the x-axis]
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)".