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

Source: US DoE
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
- MATLAB
- Scientific Frontend

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

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

**Graph Transformations**
- L
- R

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

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

\[
\begin{align*}
B[0] & \rightarrow B[1] & \rightarrow B[N-1]
\end{align*}
\]

\[\text{Tasklet} \quad \rightarrow \quad \text{Tasklet} \quad \rightarrow \quad \text{Tasklet} \]

\[
\begin{align*}
B[i] & \rightarrow B[i] & \rightarrow B[i]
\end{align*}
\]

\[\text{Scope} \quad \rightarrow \quad \text{Scope} \]

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

State s0

A

\[ A[0:N] \]

\[ i=0:N \]

A[i]

Tasklet

B[i]

\[ i=0:N \]

B[0:N]

B

State s1

C

\[ C[0:N] \]

\[ i=0:N \]

C[i]

Tasklet

A[i]

\[ i=0:N \]

A[0:N]

A

\[ A[0:N] \]
Example: 2D Stencil

**State s0**

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

**State s1**

- **Jacobi**
  - $[y=0:H, x=0:W]$

- **A**

- **B**
  - $B[y,x]$

- **B**
  - $B[y-1,x] ightarrow B[y,x-1] ightarrow B[y,x+1] ightarrow B[y+1,x]$

- **A**
  - $A[y+1,x]$

- **B**
  - $B[y,x]$

- **A**

- **B**
  - $B[y-1,x] ightarrow B[y,x-1] ightarrow B[y,x+1] ightarrow B[y+1,x]$

**Transitions:**

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

**State**
- State machine element
- Fine-grained computational block
- N-dimensional data container
- Parametric graph abstraction for parallelism
- Streaming data container
- Dynamic mapping of computations on streams
- Defines behavior during conflicting writes

**Tasklet**
- State machine element
- Fine-grained computational block

**Array**
- N-dimensional data container

**Map**
- Parametric graph abstraction for parallelism

**Exit**
- Dynamic mapping of computations on streams

**Stream**
- Streaming data container

**Consume**
- Dynamic mapping of computations on streams

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

State machine element

Interactive Semantics

Tasklet

Fine-grained computational block

Array

N-dimensional data container

Stream

Streaming data container

Consume

Exit

Dynamic mapping of computations on streams

Defines behavior during conflicting writes

Map

Exit

Parametric graph abstraction for parallelism

Internal Processing

In each step, we take one element $i$ from a set of states $S$ and update the state $s$ at step $i$. For each input connector $c$, we use the state $s$ to determine the output connector $c(i)$.

If $i$ is a data node, update its internal memory for an input connector $c(i)$, as well as output connector $c(i)$ generate an update $u(i)$ which updates $p(i) = s(i)$ for each output connector $c(i)$ of $s$ with the contents of the appropriate variable defined in $c(i)$. Execute the consumption of $(p(i), p(i), \ldots, p(i))$.

If $i$ is a message node with source $A$ and sink $B$, treat $A[i]$ as the identifier and process the message accordingly. For each input connector $c$, replace $A[i]$ with the appropriate variable defined in $c(i)$ and execute the corresponding code.

If $i$ is a consume-entry node defined by (conn, cons, ctx, cont), replace $u$ with a message and do the same for the corresponding consume exit node. Then we create a new SDGC node which contains the contents of the original consume scope(s). The new contains one or more state $s$, and connects state transition to the same state with the condition $cond$ defined by $cond($true$)$. Finally, we replace the scope(s) in current with an invoke node for new and reconnect the appropriate edges between the entry and exit nodes.

If $i$ is a transition node defined by the tuple (src, act, cont), we create a message node $m$ with the same name, a message node $r$, and a taskset $t = n$. We also store these to the node set $N$.
Hierarchical Parallelism and Heterogeneity

- Maps have schedules, arrays have storage locations

```
out = in_A * in_A
```
Hierarchical Parallelism and Heterogeneity

- Maps have schedules, arrays have storage locations

```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;
    }
}
```
Hierarchical Parallelism and Heterogeneity

\[
A[i=0:N:TN] \rightarrow A[i:i+TN] \rightarrow tA \rightarrow tA[0:TN] \rightarrow \text{Core} \rightarrow tA[ti] \rightarrow \text{out} = \text{in}_A \times \text{in}_A
\]
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;
}
```
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

```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)
```

Diagram:
- **State s0**
- **A** \([0:M, 0:K]\)
- **B** \([0:K, 0:N]\)
- **C** \([0:M, 0:N]\)
- **multiplication**
  - **[i=0:M, j=0:N, k=0:K]**
  - **A[i,k]** \(\rightarrow\) \(B[k,j]\)
  - **tmp** \([0:M, 0:N, 0:K]\)
  - **Reduce**
    - **[axis: 2, sum]**
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
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

![Image of SDFG and line graph showing performance comparison between Naive and MapReduceFusion]
Performance

SDFG

LoopReorder
MapReduceFusion

Naïve
Performance

![Graph showing performance comparison between different methods: BlockTiling, LoopReorder, MapReduceFusion, and Naive. The graph plots Performance [GFlop/s] against Problem Size.]
Performance

SDFG

RegisterTiling
BlockTiling
LoopReorder
MapReduceFusion

Performance [GFlop/s]

Problem Size

Naïve
Performance

SDFG

LocalStorage
RegisterTiling
BlockTiling
LoopReorder
MapReduceFusion

Naive
Performance

- PromoteTransient
-LocalStorage
-RegisterTiling
-BlockTiling
-LoopReorder
-MapReduceFusion

SDFG
Performance

With tuning: 98.6% of MKL

25% difference
Intel Xeon E5-2650 v4

NVIDIA Tesla P100

Xilinx VU9P

SDFG

Polyhedral Optimizers

Polly 6, Pluto 0.11.4, PPCG 0.8

Frameworks & Libraries

HPX, Halide, Intel MKL, CUBLAS, CUSPARSE, CUTLASS, CUB

General Compilers

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

19.5x of Spatial

**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

![Image of a diagram showing a comparison of algorithms with time bars and a parallel breadth-first search algorithm flowchart.]
This project has received funding from the European Research Council (ERC) under grant agreement "DAPP (PI: T. Hoefler)".

Conclusions

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

pip install dace