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!

Hierarchical Power Costs
Data Movement is the Dominant Power Cost

- 6 pJ: Cost to move data 1 mm on-chip
- 100 pJ: Typical cost of a single floating point operation
- 120 pJ: Cost to move data 20 mm on chip
- 250 pJ: Cost to move off-chip, but stay within the package (SMP)
- 2000 pJ: Cost to move data off chip into DRAM
- ~2500 pJ: Cost to move data off chip to a neighboring node

Source: Fatollahi-Fard et al.
Efficient data movement is hard!
Efficient data movement is hard!

**Multi-target IR:** LLVM, HPVM

**Polyhedral compilers:** Pluto, Polly

**General compilers:** GCC, Clang, ICC
Efficient data movement is hard!

Indirect accesses

Pointers and aliasing

Difficult & expensive whole program analysis
Data Centric Parallel Programming

Stateful DataFlow multiGraph (SDFG)  
*data-centric intermediate representation*
What is the SDFG IR?

\[ B[i] = 0.33333 \times (A[i - 1] + A[i] + A[i + 1]); \]
What is the SDFG IR?

\[ B_1 = 0.33333 \times (A_{i-1} + A_i + A_{i+1}) \]
What is the SDFG IR?

\[
\text{for } (i = 1; i < n - 1; i++) \\
\{
\]

\[
B_i = 0.33333 \times (A_{i-1} + A_i + A_{i+1});
\]

\[
A_{i-1,i,i+1}
\]

\[
B_i
\]
What is the SDFG IR?

```c
for (i = 1; i < n - 1; i++)
{
}
```
What is the SDFG IR?

\[ B_i = 0.33333 \times (A_{i-1} + A_i + A_{i+1}) \]

Symbol

\[ i = 1 \]

\[ i >= n-1 \]

\[ i < n-1 \]

\[ A[i-1,i,i+1] \]

\[ B_i \]
Why translate C to DaCe?

- 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],
    double B[2000])
{
    int t, i;

    for (t = 0; t < tsteps; t++)
    {
        for (i = 1; i < n - 1; i++)
        {
        }
        for (i = 1; i < n - 1; i++)
        {
            A[i] = 0.33333 * (B[i - 1] + B[i] + B[i + 1]);
        }
    }
}
```
Why translate C to DaCe?

- 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],
    double B[2000])
{
    int t, i;

    for (t = 0; t < tsteps; t++)
    {
        #pragma omp parallel for
        for (i = 1; i < n - 1; i++)
        {
        }
        #pragma omp parallel for
        for (i = 1; i < n - 1; i++)
        {
            A[i] = 0.33333 * (B[i - 1] + B[i] + B[i + 1]);
        }
    }
}
```
1. AST Transformations

```c
for (int i = 0; i < N; i++)
    for (int j = row_ptr[i]; j < row_ptr[i+1]; j++)
```
1. AST Transformations
   • Make basic blocks explicit

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

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
   • 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[j] * x[idx];
       }
   }

Indirect accesses

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

```c
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[j] * x[idx];
    }
}
```
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[j] * x[idx];
    }
}
C to DaCe: SpMV

1. AST Transformations
2. Translation from C to SDFG

The SDFG representation is at this point a direct representation of the control-flow centric C code. **Generates valid results!**
C to DaCe: SpMV

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

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

Indirect accesses
C to DaCe: SpMV

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

\[ y = y + A \times x[\text{idx}] \]
Symbolic scalar analysis

\[ y = y + A \times x[idx] \]

Indirect accesses
C to DaCe: SpMV

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

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

\[
y = y + A \ast x
\]
C to DaCe: SpMV

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

\[
j = \text{row_ptr}[i]
\]

\[
i = 0
\]

\[
\begin{array}{c}
\text{for (int } i = 0; i < N; i++)
\end{array}
\]

\[
\text{for (int } j = \text{row_ptr}[i]; j < \text{row_ptr}[i+1]; j++)
\]

\[
\begin{array}{c}
\text{int } idx = \text{col_idx}[j];
\end{array}
\]

\[
y[i] += A[j] * x[idx];
\]

**Update detection:** Allows more parallelization by creating the specialized \textit{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

```c
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[j] * x[idx];
```
C to DaCe: SpMV

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

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

```c
y = A * x
```

```
y[i] (Sum)
```
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[j] * x[idx];
```

<table>
<thead>
<tr>
<th>DaCe</th>
<th>polly</th>
<th>pluto</th>
<th>icc - parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
</tr>
</tbody>
</table>
Access pattern propagation

Difficult & expensive whole program analysis
Access pattern propagation

Difficult & expensive whole program analysis
Access pattern propagation

Difficult & expensive whole program analysis
Access pattern propagation
Access pattern propagation
Access pattern propagation

A[0:2N] → A
B[0:2M] → B
C[0:2N] → C
D[0:2M] → D
Access pattern propagation

Difficult & expensive whole program analysis

A $\rightarrow$ A[0:2N]
B $\rightarrow$ B[0:2M]
C $\rightarrow$ C[0:2N]
D $\rightarrow$ D[0:2M]
Access pattern propagation

Powerful & efficient whole program analysis
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
Polybench

- 30 benchmarks
- Provided as tests for polyhedral compilers
- Small kernels (tens – hundreds LOC)

Pointers and aliasing

- No manual tuning
- Full code analysis (No SCOP)

*No polyhedral analysis (yet)
Polybench - MVT

```c
for (i = 0; i < n; i++)
{
    for (j = 0; j < n; j++)
    {
        x1[i] = x1[i] + A[i][j] * y_1[j];
    }
}
for (i = 0; i < n; i++)
{
    for (j = 0; j < n; j++)
    {
        x2[i] = x2[i] + A[j][i] * y_2[j];
    }
}
```

Update detection
Polybench

- **gemver**: DaCe Rank: 1
- **trisolv**: DaCe Rank: 1
- **mvt**: DaCe Rank: 1
Polybench

No polyhedral analysis

Auto-parallelization opportunity missed

correlation
DaCe Rank: 4

covariance
DaCe Rank: 4

lu
DaCe Rank: 5

Compiler
Polybench Overview
LULESH

- Exascale proxy app from LLNL
- *Unstructured* hydrodynamics solver

\[
\begin{align*}
\text{elemX}[0] &= \text{domx}[\text{nd0i}] ; \\
\text{elemX}[1] &= \text{domx}[\text{nd1i}] ; \\
\text{elemX}[2] &= \text{domx}[\text{nd2i}] ; \\
\text{elemX}[3] &= \text{domx}[\text{nd3i}] ; \\
\text{elemX}[4] &= \text{domx}[\text{nd4i}] ; \\
\text{elemX}[5] &= \text{domx}[\text{nd5i}] ; \\
\text{elemX}[6] &= \text{domx}[\text{nd6i}] ; \\
\text{elemX}[7] &= \text{domx}[\text{nd7i}] ;
\end{align*}
\]
Developers provide manually tuned OpenMP implementation

Serial result accumulation:

```c
for ( Index_t lnode=0 ; lnode<8 ; ++lnode ) {
    Index_t gnode = elemToNode[lnode];
    domain.fx(gnode) += fx_local[lnode];
    domain.fy(gnode) += fy_local[lnode];
    domain.fz(gnode) += fz_local[lnode];
}
```

Parallel result accumulation:

```c
#pragma omp parallel for firstprivate(numNode)
for( Index_t gnode=0 ; gnode<numNode ; ++gnode )
{
    Index_t count = domain.nodeElemCount(gnode);
    Index_t *cornerList = domain.nodeElemCornerList(gnode);
    Real_t fx_tmp = Real_t(0.0);
    Real_t fy_tmp = Real_t(0.0);
    Real_t fz_tmp = Real_t(0.0);
    for (Index_t i=0 ; i < count ; ++i) {
        Index_t ielem = cornerList[i];
        fx_tmp += fx_elem[ielem];
        fy_tmp += fy_elem[ielem];
        fz_tmp += fz_elem[ielem];
    }
    domain.fx(gnode) = fx_tmp;
    domain.fy(gnode) = fy_tmp;
    domain.fz(gnode) = fz_tmp;
}
```
LULESH

Size 25^3 (11.4s seq. Runtime)

- Polly
- ICC
- LULESH-OpenMP
- SDFG

Size 64^3 (480s seq. Runtime)

- LULESH-OpenMP
- SDFG

Runtime [s] vs. Threads
Conclusion

Symbolic analysis is needed to understand data movement

Update detection opens parallelization opportunities

Access pattern propagation allows efficient whole program analysis

Understanding data movement is key to performance & portability
Thank you!

Results - Polybench

Results - LULESH