A Graphical Dataflow Programming Approach To High Performance Computing

Somashekaracharya G. Bhaskaracharya
National Instruments
Bangalore
Outline

• Graphical Dataflow Programming

• LabVIEW – Introduction and Demo

• LabVIEW Compiler (under the hood)

• Multicore Programming in LabVIEW

• Polyhedral Compilation of Graphical Dataflow Programs
Evolution of Programming Languages

- Binary
- Text Based: Fortran, Pascal
- C, C++, C#
- Java, Python, Ruby
- LabVIEW
Graphical Dataflow v/s Imperative Programs

**Imperative Programming**

- Computation specified as sequence of statements
- Each statement changes the program state

```c
// s = ut + 0.5at²
double displacement_in_time_t(double time,
                               double initial_velocity,
                               double acceleration) {
    double displacement = initial_velocity * time;
    displacement += 0.5 * acceleration * time * time;
    return displacement;
}
```
Graphical Dataflow v/s Imperative Programs

**Imperative Programming**

- Computation specified as sequence of statements
- Each statement changes the program state

```java
// s = ut + 0.5at^2
double displacement_in_time_t(double time,
                             double initial_velocity,
                             double acceleration) {
    double displacement = initial_velocity * time;
    displacement += 0.5 * acceleration * time * time;
    return displacement;
}
```

**Graphical dataflow programming**

- No notion of statements
- No fixed relative execution order
- Referential transparency
Dataflow Execution Semantics

- Interconnected set of nodes that represent specific computations
- Nodes consume input data to produce output data
- Nodes ready to **fired** as soon as data is available on all inputs

![Diagrame of dataflow execution semantics](image-url)
Inherent Parallelism Of Dataflow Programs

Partially ordered program specification

Possible orderings of node execution:

- **Strictly Sequential**
  - Multiply < Square < TernaryMultiply < Add
  - Square < TernaryMultiply < Multiply < Add
  - Square < Multiply < TernaryMultiply < Add

- Sequentiality enforced through data dependences
Inherent Parallelism Of Dataflow Programs

Partially ordered program specification

Possible orderings of node execution:

**Strictly Sequential**
- Multiply < Square < TernaryMultiply < Add
- Square < TernaryMultiply < Multiply < Add
- Square < Multiply < TernaryMultiply < Add

**Exploiting inherent parallelism**
- (Multiply || Square) < TernaryMultiply < Add
- (Multiply || (Square < TernaryMultiply)) < Add
- Square < (Multiply || TernaryMultiply) < Add

- Sequentiality enforced through data dependences
- Compiler determines the granularity of parallelism
Memory Allocation in Graphical Dataflow

- Valid to substitute expression with its value
  - at any point in program execution

Programmer’s perspective of memory allocation

Each new output value in a new memory location
Memory Allocation in Graphical Dataflow

- Valid to substitute expression with its value
  - at any point in program execution

  Programmer’s perspective of memory allocation
  Each new output value in a new memory location

- Copy avoidance strategies to reduce memory overhead
  - Output data is *inplace* to input data wherever possible

  After copy-avoidance, only 3 memory allocations are needed
Copy-avoidance and Execution Schedule

- **TernaryMultiply** < Multiply
  - Destructive update of MEM2
  - Pending read of MEM2
  - Cannot exploit parallelism
Copy-avoidance and Execution Schedule

- TernaryMultiply < Multiply
  - Destructive update of MEM2
  - Pending read of MEM2
  - Cannot exploit parallelism

- No destructive update of MEM2
  - TernaryMultiply < Multiply
  - TernaryMultiply | | Multiply
  - TernaryMultiply > Multiply

Strong interplay between copy-avoidance, clumping and scheduling
Outline

• Graphical Dataflow Programming

• LabVIEW – Introduction and Demo

• LabVIEW Compiler (under the hood)

• Multicore Programming in LabVIEW

• Polyhedral Compilation of Graphical Dataflow Programs
LabVIEW

• Platform for graphical dataflow programming
  • Owned by National Instruments
  • G dataflow programming language
  • Editor, compiler, runtime and debugger
  • Supported on Windows, Linux, Mac
  • Power PC, Intel architectures, FPGA

Measurement Control I/O  Deployable Math and Analysis  User Interface  Technology Integration

ni.com
Scalable: From Kindergarten to Rocket Science
LabVIEW Program

- LabVIEW program
  - Front Panel + Block Diagram
G Programming Language

- Data types
  - Built-in types: integer and floating point types, Boolean, string etc
  - Aggregate types: arrays, clusters, classes

- Data manipulation through built-in collection of primitives
  - Numeric palette (add, multiply, divide, subtract etc)
  - Array palette (Build array, Index array, concatenate array, decimate array etc)
G Programming Language – Control Constructs

• Case Structure

- One or more diagrams (cases)
- Value wired to selector terminal for switching
  - Boolean, string, integer, enumerated type
G Programming Language – Control Constructs

Loop structures
- While loop
- Timed loop
- **For loop**
  - LoopMax and LoopIndex boundary nodes
  - Loop carried data through shift registers
  - Tunnels (with optional indexing)

Unindexed tunnels propagate same data every iteration

Indexed tunnels
- Array auto-indexing
- Auto- accumulate iteration outputs

Shift registers to propagate data across iterations

ni.com
Outline

• Graphical Dataflow Programming

• LabVIEW – Introduction and Demo

• LabVIEW Compiler (under the hood)

• Multicore Programming in LabVIEW

• Polyhedral Compilation of Graphical Dataflow Programs
LabVIEW Compiler

Compiler
LabVIEW Compiler

- Abstracts the complexities of programming
  - Memory management
  - Thread allocation
  - Language syntax
- Edit-time semantic analysis
- Compile on Load/Run/Save
Optimizing the LabVIEW Compiler

DataFlow Intermediate Representation (DFIR)
• High-level graph-based representation
• Preserves execution semantics, dataflow, parallelism, and structure hierarchy
• Developed internally at NI
Optimizing the LabVIEW Compiler

**DataFlow Intermediate Representation (DFIR)**
- High-level graph-based representation
- Preserves execution semantics, dataflow, parallelism, and structure hierarchy
- Developed internally at NI

**Low-Level Virtual Machine (LLVM)**
- Low-level sequential representation
- Knowledge of target machine characteristics
- 3rd party, Open Source
What does DFIR look like?
DFIR Decomposition Transforms

• Lowering high-level nodes and constructs
  • equivalent lower-level nodes

Feedback Node Decomposition
DFIR Optimization Transforms

Common Sub-expression Elimination
DFIR Optimization Transforms

Common Sub-expression Elimination
DFIR Optimization Transforms

Common Sub-expression Elimination

Unreachable Code Elimination
DFIR Optimization Transforms

Loop Invariant Code Motion
DFIR Optimization Transforms

Loop Invariant Code Motion
DFIR Optimization Transforms

Loop Invariant Code Motion

Constant folding
DFIR Optimization Transforms

Loop Invariant Code Motion

Dead Code Elimination

Constant folding
Outline

• Graphical Dataflow Programming

• LabVIEW – Introduction and Demo

• LabVIEW Compiler (under the hood)

• Multicore Programming in LabVIEW

• Polyhedral Compilation of Graphical Dataflow Programs
Task Parallelism

- Divide application into independent tasks
  - Tasks mapped to separate processors
Task Parallelism

- Divide application into independent tasks
  - Tasks mapped to separate processors

- Traditional text-based languages have sequential syntax
  - Difficult to visualize and organize in parallel form

- Parallelism is more evident in graphical dataflow programs
  - Tasks as parallel sections of code on LabVIEW block diagram
  - No need to manage threads or their synchronization
Task Parallelism – An Example

- Independent data acquisition tasks
- Can be executed concurrently on multicore processor
Task Parallelism – An Example With Pitfalls

- Independent data acquisition tasks
- Can be executed concurrently on multicore processor
- Tasks not truly parallel
- Digital task depends on analog task

To maximize task parallelism, avoid unnecessary resource sharing
Multi-threaded LabVIEW Execution Environment

• LabVIEW compiler identifies *clumps*
  • Parallel sections of code on block diagram
Multi-threaded LabVIEW Execution Environment

- LabVIEW compiler identifies *clumps*
  - Parallel sections of code on block diagram

- LabVIEW runtime maintains pool of execution threads
  - Pool size at least as much as number of cores
  - During sequential run, some threads are asleep
  - Idle threads get woken up as degree of parallelism increases
Multi-threaded LabVIEW Execution Environment

- LabVIEW compiler identifies **clumps**
  - Parallel sections of code on block diagram

- LabVIEW runtime maintains pool of execution threads
  - Pool size at least as much as number of cores
  - During sequential run, some threads are asleep
  - Idle threads get woken up as degree of parallelism increases

- Thread co-operatively multitasks across clumps
  - Clumps yield periodically to scheduler
  - Waiting clumps get chance to run
Data Parallelism

- Split large dataset into smaller chunks
  - Operate on smaller chunks in parallel
  - Individual results are combined to obtain final result
Data Parallelism

- Split large dataset into smaller chunks
  - Operate on smaller chunks in parallel
  - Individual results are combined to obtain final result

- No data parallelism
- Inefficient use of resources
Data Parallelism

• Split large dataset into smaller chunks
  • Operate on smaller chunks in parallel
  • Individual results are combined to obtain final result

• No data parallelism
  • Inefficient use of resources

• Large dataset broken up into 4 subsets
  • Each core is engaged
  • Improved execution speed
Data Parallelism in LabVIEW

- Standard matmul operation in LabVIEW
- No data parallelism being exploited
- Long execution time for large datasets
Data Parallelism in LabVIEW

- Standard matmul operation in LabVIEW
- No data parallelism being exploited
- Long execution time for large datasets

- Data parallel matmul
- Matrix 1 divided into two halves
- Concurrent matmul with each half
- Individual results combined
Data Parallelism in LabVIEW

- Standard matmul operation in LabVIEW
- No data parallelism being exploited
- Long execution time for large datasets

- Data parallel matmul
- Matrix 1 divided into two halves
- Concurrent matmul with each half
- Individual results combined

<table>
<thead>
<tr>
<th>Data Parallelism</th>
<th>Execution Time on Single Core Processor</th>
<th>Execution Time on Dual Core Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Multiplication</td>
<td>1.195 seconds</td>
<td>1.159 seconds</td>
</tr>
<tr>
<td>without Data Parallelism</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td>1.224 seconds</td>
<td>0.629 seconds</td>
</tr>
<tr>
<td>with Data Parallelism</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Data Parallelism in the Real World

- Matrix-vector in real-time HPC application e.g. control system
- Sensor measurements as vector input on per-loop basis
- Matrix-vector result to control actuators
- Matrix-vector computation on 8 cores
Data Parallelism in the Real World

- Matrix-vector in real-time HPC application e.g. control system
- Sensor measurements as vector input on per-loop basis
- Matrix-vector result to control actuators
- Matrix-vector computation on 8 cores

LabVIEW program for **plasma control in ASDEX tokamak**
- Germany’s most advanced nuclear fusion platform
- Compute-intensive matrix operations on octal core server
- Real-time constraint of maintaining a 1ms control loop

“in first design stage...with LabVIEW, we obtained a **20X processing speedup on an octal core processor machine over a single-core processor**, while reaching our 1 ms control loop requirement” -- Louis Giannone, lead researcher
Structured Grids

Near-neighbor dependences in time-iterated stencil computations

```
for(t = 1; t < T; ++t)
  for(i = 1; i < N; ++i)
    for(j = 1; j < N; ++j)
      grid[t][i][j] = f(grid[t-1][i-1][j],
                        grid[t-1][i+1][j],
                        grid[t-1][i][j-1],
                        grid[t-1][i][j+1]);
```
Structured Grids

Near-neighbor dependences in time-iterated stencil computations

\[
\text{grid}[t][i][j] = f(\text{grid}[t-1][i-1][j], \text{grid}[t-1][i+1][j], \text{grid}[t-1][i][j-1], \text{grid}[t-1][i][j+1]);
\]

- Split into sub-grids
- Compute them independently
Structured Grids

Near-neighbor dependences in time-iterated stencil computations

\[
\begin{align*}
\text{for}(t = 1; t < T; ++t) \\
& \quad \text{for}(i = 1; i < N; ++i) \\
& \quad \quad \text{for}(j = 1; j < N; ++j) \\
& \quad \quad \quad \text{grid}[t][i][j] = \text{f(grid}[t-1][i-1][j],} \\
& \quad \quad \quad \text{grid}[t-1][i+1][j],} \\
& \quad \quad \quad \text{grid}[t-1][i][j-1],} \\
& \quad \quad \quad \text{grid}[t-1][i][j+1]);}
\end{align*}
\]

- Split into sub-grids
- Compute them independently
- Each icon mapped to separate core
- Feedback nodes represent data exchange
Pipelining

- Divide inherently serial task into concrete stages
- Execute stages in assembly-line fashion

Build Frame  ➔  Install Parts  ➔  Paint  = 1 Car / 3 Hours

- No pipelining
- Poor throughput
Pipelining

- Divide inherently serial task into concrete stages
- Execute stages in assembly-line fashion

- No pipelining
- Poor throughput

- Pipelined execution
- Improved throughput
Pipelining in LabVIEW

- Sequential task in a loop, with 4 stages
- Typical of streaming applications
  - FFTs manipulated one step at a time
Pipelining in LabVIEW

- Sequential task in a loop, with 4 stages
- Typical of streaming applications
  - FFTs manipulated one step at a time
- Feedback nodes to separate pipeline stages
Pipelining in LabVIEW

- Sequential task in a loop, with 4 stages
- Typical of streaming applications
  - FFTs manipulated one step at a time
- Feedback nodes to separate pipeline stages
- Pipelined execution through shift registers
- Each stage can be mapped to a separate core
Pipelining – Important Concerns

Non-Pipelined (total time = 4s):

- Stage 1 (3s)
- Stage 2 (1s)

Pipelined (total time = 3s):

- Stage 1 (3s)
- Stage 2 (1s)

Note: Performance increase = 1.33X (not an ideal case for pipelining)

Pipeline stages must be well-balanced

LabVIEW built-in timing primitives for benchmarking
Pipelining – Important Concerns

Pipeline stages must be well-balanced

LabVIEW built-in timing primitives for benchmarking

Avoid large data transfer between stages, across cores

- Cores may not share cache
- Data size could exceed cache size

Non-Pipelined (total time = 4s):

- Stage 1 (3s)
- Stage 2 (1s)

Pipelined (total time = 3s):

- Stage 1 (3s)
- Stage 2 (1s)

Note: Performance increase = 1.33X (not an ideal case for pipelining)
Parallel For Loop for Iteration Parallelism

- Concurrent execution iterations of a for loop in multiple threads
  - Greater CPU utilization

Auto-parallelization of for loop
Parallel For Loop for Iteration Parallelism

- Concurrent execution iterations of a for loop in multiple threads
  - Greater CPU utilization

Auto-parallelization of for loop
Parallel For Loop for Iteration Parallelism

- Concurrent execution iterations of a for loop in multiple threads
  - Greater CPU utilization

Auto-parallelization of for loop

- Compiler generate multiple parallel loop instances
- Each parallel loop instance represents independently schedulable clump
Configuring Iteration Parallelism
Configuring Iteration Parallelism

Automatic iteration partitioning

• Initial chunks of iterations are large (reduces scheduling overhead)

• Chunk size gradually decreases (better load balancing)
Configuring Iteration Parallelism

Automatic iteration partitioning
• Initial chunks of iterations are large (reduces scheduling overhead)
• Chunk size gradually decreases (better load balancing)

Customized iteration partitioning
• Wire in chunk size or array of chunk sizes to the C terminal
Iteration Parallelism – When to Use?

Loop must produce same result regardless of order of execution of iterations

Data carried across iterations through shift registers
Iteration Parallelism – When to Use?

Loop must produce same result regardless of order of execution of iterations

```
for (int i = 1; i < N; ++i)
    for (int j = 1; j < N; ++j)
        a[i][j] = a[i-1][j] + 1;
```

Can any loop be parallelized here?

Data carried across iterations through shift registers
Iteration Parallelism – When to Use?

Loop must produce same result regardless of order of execution of iterations

```
for (int i = 1; i < N; ++i)
    for (int j = 1; j < N; ++i)
        a[i][j] = a[i-1][j] + 1;
```

Can any loop be parallelized here?

Data carried across iterations through shift registers
Iteration Parallelism – When to Use?

Loop must produce same result regardless of order of execution of iterations

```
for (int i = 1; i < N; ++i)
    for (int j = 1; j < N; ++i)
        a[i][j] = a[i-1][j] + 1;
```

Can any loop be parallelized here?

Data carried across iterations through shift registers

LabVIEW automatically does **cross-iteration dependence analysis**
- VI breaks if dependences are violated

One iteration should not depend on results of another
- Writing A[i-1] in iteration i-1
- Reading A[i-1] in iteration (i)
Outline

• Graphical Dataflow Programming

• LabVIEW – Introduction and Demo

• LabVIEW Compiler (under the hood)

• Multicore Programming in LabVIEW

• Polyhedral Compilation of Graphical Dataflow Programs
Parallel For Loop Limitations

None of these loops can be parallelized

```plaintext
for(i=1;i<=N;i++)
   for(j=1;j<=N;j++){
      // neither of the two loops are parallel
      a[i][j] = a[i][j-1] + a[i-1][j]
   }
```

Loop-nest is inner parallel
None of these loops can be parallelized

for(i=1;i<=N;i++)
  for(j=1;j<=N;j++){
    // neither of the two loops are parallel
    a[i][j] = a[i][j-1] + a[i-1][j]
  }

Loop-nest is inner parallel
None of these loops can be parallelized

```
for(i=1; i<=N; i++)
  for(j=1; j<=N; j++)
    // neither of the two loops are parallel
    a[i][j] = a[i][j-1] + a[i-1][j]
```
Parallel For Loop Limitations

None of these loops can be parallelized

Loop skewing exposes the hidden parallelism

Loop-nest is inner parallel
Polyhedral Model - A Short Overview

- Abstract mathematical representation
  - Convenient to reason about complex program transformations
- Static Control Parts (SCoP), typically affine loop-nests
  - e.g. stencil computations, linear algebra kernels

```c
for(i=0; i<=n-1; i++) // loop bounds are affine
    for(j=2i; j<=2i+n-1; j++)
        for(k=2i-j; k<=2i-j+n-1; k++)
            a[i][j][k] = a[i+j][i+j+k][2i-3j+k+n-1] + 1; // accesses are affine
```
Polyhedral Model - A Short Overview

• Abstract mathematical representation
  • Convenient to reason about complex program transformations
• Static Control Parts (SCoP), typically affine loop-nests
  • e.g. stencil computations, linear algebra kernels

```plaintext
for(i=0; i<=n-1; i++) // loop bounds are affine
    for(j=2i; j<=2i+n-1; j++)
        for(k=2i-j; k<=2i-j+n-1; k++)
            a[i][j][k] = a[i+j][i+j+k][2i-3j+k+n-1] + 1; // accesses are affine
```
Polyhedral Model - A Short Overview

- **Dynamic instances** of a statement
  - Integer points inside a polyhedron
  - **Iteration domain** as conjunction of affine inequalities involving surrounding loop iterators and global parameters

```plaintext
for(i=1;i<=n;i++)
for(j=1;j<=n;j++)
if(i<=n-j+2)
S1;
```

*Figure*. Polyhedral representation of a loop-nest in geometrical and linear algebraic form
A multi-dimensional affine schedule

- Specifies order in which the integer points need to be scanned
- Maps each integer point to multi-dimensional logical timestamp (think...hours, minutes, seconds)

Schedule of the statement instances is given by $\theta(i, j) = (i, j)$
Polyhedral model - a brief overview

- Array access information also encoded, must be affine
- Polyhedral optimizer/parallelizer
  - Analyzes the dependences
  - Pick schedule without violating dependences using a cost model
  - **PLuTo**: minimize dependence distances in transformed space
    - Optimizes parallelism and locality simultaneously
Polyhedral model - a brief overview

- Array access information also encoded, must be affine
- Polyhedral optimizer/parallelizer
  - Analyzes the dependences
  - Pick schedule without violating dependences using a cost model
  - PLuTo: minimize dependence distances in transformed space
    - Optimizes parallelism and locality simultaneously

```c
for(i=1;i<=N;i++)
    for(j=1;j<=N;j++){
        // neither of the two loops are parallel
        a[i][j] = a[i][j-1] + a[i-1][j]
    }
```

Schedule of the statement instances is given by

\[ \text{theta}(i, j) = (i, j) \]
Polyhedral model - a brief overview

- Array access information also encoded, must be affine
- Polyhedral optimizer/parallelizer
  - Analyzes the dependences
  - Pick schedule without violating dependences using a cost model
- PLuTo: minimize dependence distances in transformed space
  - Optimizes parallelism and locality simultaneously

Schedule of the statement instances is given by
\[ \theta(i, j) = (i, j) \]

New schedule is
\[ \theta(i, j) = (i+j, j) \]
Polyhedral compilation - some related work

Polyhedral compilation of imperative programs
• Extract polyhedral representation e.g. Clan (Cedric Bastoul et al)
• Polyhedral transformation - PLuTo (Uday Bondhugula et al)
• Generated transformed code e.g. CLooG (Cedric Bastoul et al)
• Polyhedral compilation in production compilers e.g. IBM-XL, RSTREAM

Polyhedral compilation of graphical dataflow programs?
• Polyhedral extraction from dataflow programs
• Synthesizing dataflow programs from polyhedral representation
Extracting Polyhedral Representation

• Identifying statement analogues

• Relating array accesses to a particular array allocation

• Execution schedule depends on the actual inplaceness strategy
Static Control Dataflow Diagram (SCoD)

- Canonical form of dataflow program
- Inplaceness patterns that facilitate polyhedral extraction
  - no new memory allocation for array data inside the SCoD

- Similarities with SCoP
  - All computations nodes are functional
  - Maximal dataflow diagram with countable loop constructs
  - Loop bounds and conditional depend on parameters that are invariant for the diagram
SCoD – Destructive Updates

• At most one destructive update of array data
Compute-dags as Statement Analogues

• Schedule of nodes exists such that no array copy is needed
  • hint: schedule all array reads ahead of the array write

• SCoD as sequence of computations that over-write incoming array data

• Compute-dags can be identified to serve as statement analogues
Compute-dags as Statement Analogues

- A path exists from all nodes in the compute-dag to the root
Iteration Domain of Statement Analogues

# Number of statements
1

# ---------------------------------------- Statement 0
# ---------------------------------------- Domain
# Iterator Domain is provided
1
# Iterator Domain
# In matrix form
# There are 3 surrounding loops
# -t2 + N0 -1 >= 0
# t2 >= 0
# -t1 + N0 -1 >= 0
# t1 >= 0
# -t0 + N0 -1 >= 0
# t0 >= 0
6 6
 1 1 0 0 0 0
 1 -1 0 0 1 -1
 1 0 1 0 0 0
 1 0 -1 0 1 -1
 1 0 0 1 0 0
 1 0 0 -1 1 -1
Determining Schedule of Statement Analogues

# Scattering function is provided

# Scattering function
# ReplaceArrayNode is scheduled at 0, t0, 0, t1, 0, t2, 0,
# In matrix form

\[
\begin{bmatrix}
7 & 6 \\
0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]
Analyzing Accesses of Statement Analogues

# Access informations are provided
1
# ArrayIndexNode::Terminal(3) accesses t1
# ArrayIndexNode::Terminal(2) accesses t0
# ReplaceArrayNode::Terminal(4) accesses t1
# ReplaceArrayNode::Terminal(3) accesses t0
# ArrayIndexNode::Terminal(3) accesses t1
# ArrayIndexNode::Terminal(2) accesses t0
# ArrayIndexNode::Terminal(3) accesses t2
# ArrayIndexNode::Terminal(2) accesses t2
# ArrayIndexNode::Terminal(2) accesses t0
# In matrix form
# Read access information
6 6
1 1 0 0 0 0
0 0 1 0 0 0
2 0 0 1 0 0
0 0 1 0 0 0
3 1 0 0 0 0
0 0 0 1 0 0
# Write access information
2 6
1 1 0 0 0 0
0 0 1 0 0 0
The PolyGLoT framework

```c
for(t0=0;t0<=N0-1;t0++){
    for(t1=0;t1<=N0-1;t1++){
        for(t2=0;t2<=N0-1;t2++){
            // This is just a representative statement of the form
            // <Statement-id>[0] = <waccess> * <sum of racceses>
            // S0[0]=A1[t0][t1]*A1[t0][t1]+A2[t2][t1]+A3[t0][t2];
        }
    }
}
```

A high-level overview of PolyGLoT:

```c
if (N0 >= 1) {
    lbp=0;
    ubp=floor(N0-1,32);
    #pragma omp parallel for private(lbv,ubv)
    for (t1=0;t1<=floor(N0-1,32);t1++) {
        for (t2=0;t2<=floor(N0-1,32);t2++) {
            for (t3=0;t3<=floor(N0-1,32);t3++) {
                for (t4=32*t1;t4<=min(N0-1,32*t1+31);t4++) {
                    for (t5=32*t2;t5<=min(N0-1,32*t2+31);t5++) {
                        for (t6=32*t3;t6<=min(N0-1,32*t3+31);t6++) {
                            S0[0]=A1[t4][t5]*A1[t4][t5]+A2[t6][t5]+A3[t4][t6];
                        }
                    }
                }
            }
        }
    }
}```
Experimental evaluation

- Implemented benchmarks in Polybench suite in LabVIEW
- PolyGLoT as a separate transform pass in LV desktop compiler
  - uses Pluto as the polyhedral optimizer (locality transformations + parallelization)

- Dual-socket Intel(R) Xeon(R) CPU E5606 (2.13GHz) machine with 8 cores, 24GB RAM, 8MB L3 cache
Experimental evaluation

- **lv-parallel** - LabVIEW production compiler, with parallelization
- **pg-par** - LabVIEW compiler with PolyGLOt enabled for auto-parallelization
- **pg-loc-par** - LabVIEW compiler with PolyGLOt enabled for auto-parallelization + locality optimization
- Mean speed-up of 2.30× with **pg-loc-par** over **lv-parallel**

<table>
<thead>
<tr>
<th></th>
<th>lv-par (s)</th>
<th>pg-par (s)</th>
<th>pg-loc-par (s)</th>
<th>Speedup (pg-par over lv-par)</th>
<th>Speedup (pg-loc-par over lv-par)</th>
</tr>
</thead>
<tbody>
<tr>
<td>atax</td>
<td>0.707</td>
<td>0.642</td>
<td>0.167</td>
<td>1.101</td>
<td>4.234</td>
</tr>
<tr>
<td>bicg</td>
<td>0.409</td>
<td>0.22</td>
<td>0.093</td>
<td>1.859</td>
<td>4.398</td>
</tr>
<tr>
<td>doitgen</td>
<td>0.976</td>
<td>0.999</td>
<td>0.934</td>
<td>0.977</td>
<td>1.045</td>
</tr>
<tr>
<td>floyd-war</td>
<td>82.76</td>
<td>13.64</td>
<td>4.909</td>
<td>6.067</td>
<td>16.859</td>
</tr>
<tr>
<td>gemm</td>
<td>7.026</td>
<td>5.473</td>
<td>3.628</td>
<td>1.284</td>
<td>1.937</td>
</tr>
<tr>
<td>gesummv</td>
<td>0.078</td>
<td>0.069</td>
<td>0.074</td>
<td>1.130</td>
<td>1.054</td>
</tr>
<tr>
<td>matmul</td>
<td>89.49</td>
<td>94.7</td>
<td>27.44</td>
<td>0.945</td>
<td>3.261</td>
</tr>
<tr>
<td>mvt</td>
<td>0.195</td>
<td>0.334</td>
<td>0.105</td>
<td>0.584</td>
<td>1.857</td>
</tr>
<tr>
<td>seidel</td>
<td>45.03</td>
<td>9.797</td>
<td>8.364</td>
<td>4.596</td>
<td>5.384</td>
</tr>
<tr>
<td>ssymm</td>
<td>15.03</td>
<td>55.45</td>
<td>23.85</td>
<td>0.271</td>
<td>0.630</td>
</tr>
<tr>
<td>spr2k</td>
<td>4.19</td>
<td>4.423</td>
<td>4.223</td>
<td>0.947</td>
<td>0.992</td>
</tr>
<tr>
<td>syrk</td>
<td>2.974</td>
<td>3.118</td>
<td>2.793</td>
<td>0.954</td>
<td>1.065</td>
</tr>
<tr>
<td>trmm</td>
<td>41.29</td>
<td>39.94</td>
<td>11.42</td>
<td>1.034</td>
<td>3.616</td>
</tr>
</tbody>
</table>
Summary

- Graphical dataflow programming
  - Simple, intuitive and accessible to novice programmers
  - Well-suited for exploiting and expressing parallelism
  - Used by scientists and engineers in various domains

- Optimizing and parallelizing LabVIEW compiler
  - Clumps of independently schedulable sections of code
  - Task parallelism, data parallelism, pipelining etc

- Parallel for loop for cross-iteration parallelism

- Polyhedral model for complex program transformations
Thanks!

Questions?