A Proposal for A Framework for More Effective Loop Optimization

LLVM Developer’s Meeting 2020

Michael Kruse, Hal Finkel

Argonne Leadership Computing Facility
Argonne National Laboratory

2020-10-06
Not an universal solution for everyone’s use case

However, there is overlapping functionality

- LLVM-IR, Machine-IR, VPLAN, MLIR, …
Motivation

The Good, The Bad, and The Ugly

https://www.youtube.com/watch?v=QpvZt9w-Jik
# Complexity of Writing a New Loop Pass

<table>
<thead>
<tr>
<th>Loop Pass</th>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>LoopDistribute</td>
<td>1063</td>
</tr>
<tr>
<td>LoopInterchange</td>
<td>1529</td>
</tr>
<tr>
<td>LoopUnroll</td>
<td>2025</td>
</tr>
<tr>
<td>LoopIdiom</td>
<td>1794</td>
</tr>
</tbody>
</table>

## High-Level Difficulties
- Ensure legality (incl. Dependencies: LoopAccessInfo, Memory Dependence Analysis, Memory SSA, …)
- Machine profitability model

## Low-Level Troubles
- Preserve control flow
- Preserve (LC-)SSA
- Preserve passes (LoopInfo, Dominator Tree, Scalar Evolution, …)
**Motivation** → **Code Versioning**

**Loop Version Explosion**

**Original Source**

```c
for (int i = 0; i < n; i+=1)
    for (int j = 0; j < m; j+=1)
        Stmt(i,j);
```

4 / 23
if (rtc1) {
  for (int i = 0; i < n; i+=1) /* 1x transformed */
    for (int j = 0; j < m; j+=1)
      Stmt(i,j);
} else {
  for (int i = 0; i < n; i+=1) /* fallback */
    for (int j = 0; j < m; j+=1)
      Stmt(i,j);
}
Loop Version Explosion
Strip-Mine Outer Loop (2 transformations so far)

```c
if (rtc1) {
    if (rtc2) {
        for (int i1 = 0; i1 < n; i1+=4) /* 2x transformed */
            for (int j = 0; j < m; j+=1)
                for (int i2 = 0; i2 < 4; i2+=1) /* new loop */
                    Stmt(i1+i2,j);
    } else {
        for (int i = 0; i < n; i+=1) /* 1x transformed */
            for (int j = 0; j < m; j+=1)
                Stmt(i,j);
    }
} else {
    if (rtc3) {
        for (int i1 = 0; i1 < n; i1+=4) /* 1x transformed */
            for (int j = 0; j < m; j+=1)
                for (int i2 = 0; i2 < 4; i2+=1) /* new loop */
                    Stmt(i1+i2,j);
    } else {
        for (int i = 0; i < n; i+=1) /* fallback-fallback */
            for (int j = 0; j < m; j+=1)
                Stmt(i,j);
    }
}
```
Motivation → Code Versioning

Loop Version Explosion

Optimize Inner Loop (3 transformations so far)

```c
if (rtc1) {
    if (rtc2) {
        for (int i1 = 0; i1 < n; i1+=4)
            for (int j = 0; j < m; j+=1) {
                if (rtc4) {
                    for (int i2 = 0; i2 < 4; i2+=1)
                        Stmt(i1+i2,j);
                } else {
                    for (int i2 = 0; i2 < 4; i2+=1) /* fallback */
                        Stmt(i1+i2,j);
                }
            }
    } else {
        for (int i = 0; i < n; i+=1) {
            if (rtc5) {
                for (int j = 0; j < m; j+=1)
                    Stmt(i,j);
            } else {
                for (int j = 0; j < m; j+=1) /* fallback-fallback */
                    Stmt(i,j);
            }
        }
    }
} else {
    if (rtc3) {
        for (int i1 = 0; i1 < n; i1+=4)
            for (int j = 0; j < m; j+=1) {
                if (rtc6) {
                    for (int i2 = 0; i2 < 4; i2+=1)
                        Stmt(i1+i2,j);
                } else {
                    for (int i2 = 0; i2 < 4; i2+=1) /* fallback-fallback */
                        Stmt(i1+i2,j);
                }
            }
    } else {
        for (int i = 0; i < n; i+=1) {
            if (rtc7) {
                for (int j = 0; j < m; j+=1)
                    Stmt(i,j);
            } else {
                for (int j = 0; j < m; j+=1) /* fallback-fallback-fallback */
                    Stmt(i,j);
            }
        }
    }
}
```
Fixed transformation order

May conflict with user directives:

```c
#pragma distribute
#pragma interchange

for (int i = 1; i < n; i+=1)
    for (int j = 0; j < m; j+=1) {
        A[i][j] = i + j;
        B[i][j] = A[i-1][j];
    }
```
Motivation → Conflicting Passes

Scalar/Loop Optimization Interference

for (int i=0; i<n; i+=1)
  for (int j=0; j<m; j+=1)
    A[i] += i*B[j];

LICM (Register Promotion)

for (int i=0; i<n; i+=1) {
  tmp = A[i];
  for (int j=0; j<m; j+=1)
    tmp += i*B[j];
  A[i] = tmp;
}

Loop Interchange

for (int j=0; j<m; j+=1)
  for (int i=0; i<n; i+=1)
    A[i] += i*B[j];

Loop Rotation

for (int j=0; j<m; j+=1) {
  tmp = B[j];
  for (int i=0; i<n; i+=1)
    A[i] += i*tmp;
}

GVN (LoadPRE)

Conflicting Normal Forms

- LCSSA vs. InstCombine
- LoopSimplify vs. SimplifyCFG (LoopSimplifyCFG)
- Loop metadata drop (e.g. llvm.org/PR27974)

Pessimizing Normal Forms

- LoopRotation
- IndVarSimplify
What If... Copying IR Was Cheap?

```
for (int i = c-2; i < c+255; i+=2)
```

Normalization

```
for (int i = 0; i < 128; i+=1)
```

Pragma clang loop vectorize(enable)

```
#pragma clang loop vectorize(enable)
for (int i = 0; i < 128; i+=1)
```

Pragma clang loop distribute(enable)

```
#pragma clang loop distribute(enable)
for (int i = 0; i < 128; i+=1)
```

Pragma omp target parallel for

```
#pragma omp target parallel for
for (int i = 0; i < 128; i+=1)
```

---

**Advantage**

- Generic legality & profitability analysis on transformed code
  - Passes don’t need to implement themselves
  - Cheap heuristics can still be applied beforehand
What If... Copying IR Was Cheap?

Advantage

- Generic legality & profitability analysis on transformed code
- Passes don’t need to implement themselves
- Cheap heuristics can still be applied beforehand
What If... Copying IR Was Cheap?

```
for (int i = c-2; i < c+255; i+=2)
```

normalization

```
for (int i = 0; i < 128; i+=1)
```

```
#pragma clang loop vectorize(enable)
for (int i = 0; i < 128; i+=1)
```

```
#pragma clang loop distribute(enable)
for (int i = 0; i < 128; i+=1)
```

```
#pragma clang loop vectorize(enable)
#pragma clang loop distribute(enable)
for (int i = 0; i < 128; i+=1)
```

```
#pragma omp target parallel for
for (int i = 0; i < 128; i+=1)
```

---

**Advantage**

- Generic legality & profitability analysis on transformed code
- Passes don’t need to implement themselves
- Cheap heuristics can still be applied beforehand
What If... Copying IR Was Cheap?

```
for (int i = c-2; i < c+255; i+=2)
```

```
for (int i = 0; i < 128; i+=1)
```

```
#pragma clang loop vectorize(enable)
for (int i = 0; i < 128; i+=1)
```

```
#pragma clang loop distribute(enable)
for (int i = 0; i < 128; i+=1)
```

```
#pragma omp target parallel for
for (int i = 0; i < 128; i+=1)
```

```
#pragma clang loop vectorize(enable)
#pragma clang loop distribute(enable)
for (int i = 0; i < 128; i+=1)
```

```
#pragma omp target parallel for
```

```
#pragma omp target parallel for
```

---

**Advantage**

- Generic legality & profitability analysis on transformed code
  - Passes don’t need to implement themselves
  - Cheap heuristics can still be applied beforehand
What If... Copying IR Was Cheap?

```
for (int i = c-2; i < c+255; i+=2)
```

```
for (int i = 0; i < 128; i+=1)
```

```
#pragma clang loop vectorize(enable)
for (int i = 0; i < 128; i+=1)
```

```
#pragma clang loop distribute(enable)
for (int i = 0; i < 128; i+=1)
```

```
#pragma omp target parallel for
for (int i = 0; i < 128; i+=1)
```

**Advantage**
- Generic legality & profitability analysis on transformed code
- Passes don't need to implement themselves
- Cheap heuristics can still be applied beforehand
Legality Check
Comparison between known-good (original) and transformed loop tree

- All statement instances executed
- No additional instances
- No dependency violations
- If statements are changed, require explicit mapping
Profitability Check

- Infrastructure to enable possibilities...

Optimization Library

- Hard-coded best practices
## Profitability Check

- Infrastructure to enable possibilities...

### Optimization Library

- Hard-coded best practices

### Execution Time Machine Model

- Estimate cycles of straight-line code
  - `llvm-mca`
  - Memory access latency
- Estimate trip count
  - Constant ("100")
  - "infinity" (only innermost kernel counts)
  - From user annotations (#pragma loop count(n))
  - From PGO / previous JIT stage
Profitability Check

- Infrastructure to enable possibilities...

### Optimization Library
- Hard-coded best practices

### Execution Time Machine Model
- Estimate cycles of straight-line code
  - `llvm-mca`
  - Memory access latency
- Estimate trip count
  - Constant ("100")
  - "infinity" (only innermost kernel counts)
  - From user annotations (`#pragma loop count(n)`)
  - From PGO / previous JIT stage

### User-Directed
- Apply user-annotations (pragmas)
- Applications come with optimization plugins
## Profitability Check

- Infrastructure to enable possibilities...

### Optimization Library

- Hard-coded best practices

### Execution Time Machine Model

- Estimate cycles of straight-line code
  - `llvm-mca`
  - Memory access latency
- Estimate trip count
  - Constant ("100")
  - "infinity" (only innermost kernel counts)
  - From user annotations (#pragma loop count(n))
  - From PGO / previous JIT stage

### User-Directed

- Apply user-annotations (pragmas)
- Applications come with optimization plugins

### Autotuning

- Select most-promising not-yet-evaluated
- or
- Select know-fastest
Profitability Check

- Infrastructure to enable possibilities...

**Optimization Library**
- Hard-coded best practices

**Execution Time Machine Model**
- Estimate cycles of straight-line code
  - `llvm-mca`
  - Memory access latency
- Estimate trip count
  - Constant ("100")
  - "infinity" (only innermost kernel counts)
  - From user annotations (`#pragma loop count(n)`)
  - From PGO / previous JIT stage

**User-Directed**
- Apply user-annotations (pragmas)
- Applications come with optimization plugins

**Autotuning**
- Select most-promising not-yet-evaluated
  - or -
- Select know-fastest

**Machine Learning**
- Apply a per-architecture pre-trained model
CPU Compute Hierarchy

Loop structure

```c
for (int i = 0; i < 1024; i+=1) {
    Body(i, ...)
}
```

Compute hierarchy

- Cluster nodes
- Processors
- Cores
- SMT
- SIMD
- Superscalar
- Sequential

- MPI/PGAS
- thread-parallelization
- vectorization
- unrolling+instruction scheduling
Loop Hierarchies → Profitability

**GPU Hierarchy Mapping**

---

**Loop structure**

```c
for (int i = 0; i < 1024; i+=1)
```

**Compute hierarchy**

- Cluster nodes
- Accelerator cards
- Multiprocessors (Grid)
- CUDA Core (Block)
- SIMT (Warp)
- SIMD
- Sequential

**Body**

```c
Body(i, ...)
```
## GPU Hierarchy Mapping

<table>
<thead>
<tr>
<th>Loop structure</th>
<th>Compute hierarchy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop hierarchy</td>
<td>Cluster nodes</td>
</tr>
<tr>
<td></td>
<td>Accelerator cards</td>
</tr>
<tr>
<td>for (int i = 0; i &lt; 1024; i+=1)</td>
<td>Multiprocessors (Grid)</td>
</tr>
<tr>
<td></td>
<td>CUDA Core (Block)</td>
</tr>
<tr>
<td></td>
<td>SIMT (Warp)</td>
</tr>
<tr>
<td></td>
<td>SIMD</td>
</tr>
<tr>
<td></td>
<td>Sequential</td>
</tr>
</tbody>
</table>

Body(i, ...)

---

Loop Hierarchies → Profitability
GPU Hierarchy Mapping

<table>
<thead>
<tr>
<th>Loop structure</th>
<th>Compute hierarchy</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (int i = 0; i &lt; 1024; i+=1)</td>
<td>Cluster nodes</td>
</tr>
<tr>
<td></td>
<td>Accelerator cards</td>
</tr>
<tr>
<td></td>
<td>Multiprocessors (Grid)</td>
</tr>
<tr>
<td></td>
<td>CUDA Core (Block)</td>
</tr>
<tr>
<td></td>
<td>SIMT (Warp)</td>
</tr>
<tr>
<td></td>
<td>SIMD</td>
</tr>
<tr>
<td></td>
<td>Sequential</td>
</tr>
</tbody>
</table>

Body(i, ...)

Loop Hierarchies → Profitability
## GPU Hierarchy Mapping

<table>
<thead>
<tr>
<th>Loop structure</th>
<th>Compute hierarchy</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>for (int i = 0; i &lt; 1024; i+=1)</code></td>
<td>Cluster nodes</td>
</tr>
<tr>
<td><code>for (int k = 0; k &lt; K; k+=1)</code></td>
<td>Accelerator cards</td>
</tr>
<tr>
<td><code>for (int l = 0; l &lt; L; l+=1)</code></td>
<td>Multiprocessors (Grid)</td>
</tr>
<tr>
<td><code>for (int m = 0; m &lt; M; m+=1)</code></td>
<td>CUDA Core (Block)</td>
</tr>
<tr>
<td><code>for (int n = 0; n &lt; N; n+=1)</code></td>
<td>SIMT (Warp)</td>
</tr>
<tr>
<td><code>for (int o = 0; o &lt; O; o+=1)</code></td>
<td>SIMD</td>
</tr>
<tr>
<td><code>Body(i, ...)</code></td>
<td>Sequential</td>
</tr>
</tbody>
</table>
Loop Hierarchies → Profitability

GPU Hierarchy Mapping

<table>
<thead>
<tr>
<th>Loop structure</th>
<th>Compute hierarchy</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (int floor = 0; floor &lt; 1024; floor+=128)</td>
<td>Cluster nodes</td>
</tr>
<tr>
<td>for (int tile = floor; tile &lt; floor+128; tile+=1)</td>
<td>Accelerator cards</td>
</tr>
<tr>
<td></td>
<td>Multiprocessors (Grid)</td>
</tr>
<tr>
<td></td>
<td>CUDA Core (Block)</td>
</tr>
<tr>
<td></td>
<td>SIMT (Warp)</td>
</tr>
<tr>
<td></td>
<td>SIMD</td>
</tr>
<tr>
<td></td>
<td>Sequential</td>
</tr>
</tbody>
</table>

Body(tile, ...)
## GPU Hierarchy Mapping

<table>
<thead>
<tr>
<th>Loop structure</th>
<th>Compute hierarchy</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (int floor = 0; floor &lt; 1024; floor+=128)</td>
<td>Cluster nodes</td>
</tr>
<tr>
<td>for (int t1 = floor; t1 &lt; floor+128; t1+=32)</td>
<td>Accelerator cards</td>
</tr>
<tr>
<td>for (int t2 = t1; t2 &lt; t1+32; t2+=1)</td>
<td>Multiprocessors (Grid)</td>
</tr>
<tr>
<td>Body(t2, ...)</td>
<td>CUDA Core (Block)</td>
</tr>
<tr>
<td></td>
<td>SIMT (Warp)</td>
</tr>
<tr>
<td></td>
<td>SIMD</td>
</tr>
<tr>
<td></td>
<td>Sequential</td>
</tr>
</tbody>
</table>
Loop Hierarchies → The Concept

Loop Tree
An Old Idea

- Open64 LNO (Loop Nest Optimizer)
- xlf ASTI (Analyzer Scalarizer Transformer Inliner; -qhot)
- ISL Schedule Trees
- MLIR Dialects
Loop Hierarchy DAG

```c
void Function(int s) {
    for (int i = 0; i < 128; i+=1) {
        for (int j = s; j < 64; j+=1) A[i][j] = j*sin(2*PI*i/128);
        for (int k = s; k < 256; k+=1) B[i][k] = k*cos(2*PI*i/128);
    }
}
```

- **Function**
- **Root**
- **Loops**
- **Statements**
- **Expressions**
Loop Hierarchy DAG

```c
void Function(int s) {
    for (int i = 0; i < 128; i+=1) {
        for (int j = s ; j < 64; j+=1) A[i][j] = j*sin(2*PI*i/128);
        for (int k = 255; k >= s ; k-=1) B[i][k] = k*cos(2*PI*i/128);
    }
}
```

Assumption: s ≠ INT_MIN
Green/Red/Blue Tree

Tree Types

1 Green tree: Source of truth
Loop Hierarchies → Green/Red//RedRef Trees

Green/Red/Blue Tree

Tree Types

1. Green tree: Source of truth
2. Red tree: On demand
Green/Red/Blue Tree

**Tree Types**

1. **Green tree**: Source of truth
2. **Red tree**: On demand
Green/Red/Blue Tree

Tree Types

1. Green tree: Source of truth
2. Red tree: On demand
3. RedRef tree: Recursive visitor
Loop Hierarchies → Green/Red//RedRef Trees

Green/Red/Blue Tree

Tree Types

1. Green tree: Source of truth
2. Red tree: On demand
3. RedRef tree: Recursive visitor
Green/Red/Blue Tree

Tree Types

1. Green tree: Source of truth
2. Red tree: On demand
3. RedRef tree: Recursive visitor
Green/Red/Blue Tree

- **Root**

**Tree Types**

1. Green tree: Source of truth
2. Red tree: On demand
3. RedRef tree: Recursive visitor
Green/Red/Blue Tree

Tree Types

1. Green tree: Source of truth
2. Red tree: On demand
3. RedRef tree: Recursive visitor
RedRef Visitor

class Search : RecursiveRedRefVisitor {
    void visit(const RedRef &Node) {
        ... Node.getParent() ...

        for (RedRef Child : node.children())
            visit(Child);
    }
}
## Node Properties

### Loops/Sequences
- Children
- Execution condition
- Repeat condition/trip count
- Loop-carried scalars/array-regions
- Private scalars/array-regions
- Assumptions
- Statement summary
  - Read/(Over-)Written scalars
  - Read/(Over-)Written array regions
  - Unaccounted side-effects
  - Original IR region
  - Origin node

### Side-effect Statements
- Operation kind
- Execution condition
- Assigned scalars
- Assumptions
- Statement summary
  - Read/(Over-)Written scalars
  - Read/(Over-)Written array regions
  - Unaccounted side-effects
  - Original IR region
  - Origin node

### Expressions
- Operation kind
- Scalar arguments
## Operation Lifting

<table>
<thead>
<tr>
<th>IR Construct</th>
<th>Raised to</th>
<th>Assumptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLVM instruction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MLIR operation</td>
<td>Generic <code>lof::Operation</code></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Operation Lifting

<table>
<thead>
<tr>
<th>IR Construct</th>
<th>Raised to</th>
<th>Assumptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLVM instruction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MLIR operation</td>
<td>Generic <code>lof::Operation</code></td>
<td></td>
</tr>
<tr>
<td>Two’s complement arithmetic</td>
<td>Infinite precision arithmetic</td>
<td>No integer overflow/wrap</td>
</tr>
<tr>
<td>MLIR index</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Array subscripts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loop-carried dependency of associative operation Reduction</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Operation Lifting

<table>
<thead>
<tr>
<th>IR Construct</th>
<th>Raised to</th>
<th>Assumptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLVM instruction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MLIR operation</td>
<td>Generic <code>lof::Operation</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Two's complement arithmetic</td>
<td>Infinite precision arithmetic</td>
<td>No integer overflow/wrap</td>
</tr>
<tr>
<td>MLIR index</td>
<td></td>
<td></td>
</tr>
<tr>
<td>select</td>
<td><code>lof::Operation::nop</code></td>
<td>(assignment+condition encoded in nodes)</td>
</tr>
<tr>
<td>PHI node</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MLIR BB parameter</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Operation Lifting

<table>
<thead>
<tr>
<th>IR Construct</th>
<th>Raised to</th>
<th>Assumptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLVM instruction</td>
<td>Generic <code>lof::Operation</code></td>
<td></td>
</tr>
<tr>
<td>MLIR operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Two’s complement arithmetic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MLIR index</td>
<td>Infinite precision arithmetic</td>
<td>No integer overflow/wrap</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>select</td>
<td><code>lof::Operation::nop</code></td>
<td>(assignment + condition encoded in nodes)</td>
</tr>
<tr>
<td>PHI node</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MLIR BB parameter</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>llvm::LoadInst</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MLIR memref store</td>
<td>Array subscripts</td>
<td>No aliasing of memory range</td>
</tr>
<tr>
<td>memcpy</td>
<td></td>
<td>Subscripts within shape</td>
</tr>
</tbody>
</table>
## Operation Lifting

<table>
<thead>
<tr>
<th>IR Construct</th>
<th>Raised to</th>
<th>Assumptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLVM instruction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MLIR operation</td>
<td>Generic <code>lof::Operation</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Two's complement arithmetic</td>
<td>Infinite precision arithmetic</td>
<td>No integer overflow/wrap</td>
</tr>
<tr>
<td>MLIR index</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Infinite precision arithmetic</td>
<td>No integer overflow/wrap</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>select</code></td>
<td><code>lof::Operation::nop</code></td>
<td>(assignment+condition encoded in nodes)</td>
</tr>
<tr>
<td><code>PHI node</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>MLIR BB parameter</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>llvm::LoadInst</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>MLIR memref store memcpy</code></td>
<td>Array subscripts</td>
<td>No aliasing of memory range</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Subscripts within shape</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loop-carried dependency of as-</td>
<td>Reduction</td>
<td></td>
</tr>
<tr>
<td>sociative operation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Analyses

### Arithmetic Evaluator

- Expression simplification
- Tautology/Unsatisfiability (Approximative)
## Analyses

### Arithmetic Evaluator
- Expression simplification
- Tautology/Unsatisfiability (Approximative)

### Closed-Form Expressions
- Expressions based only on invariants and loop counters
- Like ScalarEvolution / MLIR Affine expressions / isl_pw_aff
## Analyses

### Arithmetic Evaluator
- Expression simplification
- Tautology/Unsatisfiability (Approximative)

### Closed-Form Expressions
- Expressions based only on invariants and loop counters
- Like ScalarEvolution / MLIR Affine expressions / isl_pw_aff

### Array Detection
- Identify non-aliasing address ranges/base pointers (AliasSetTracker/Assumption)
- Derive array subscripts (GetElementPtr/MLIR MemRef/Delinearization)
## Analyses

### Arithmetic Evaluator
- Expression simplification
- Tautology/Unsatisfiability (Approximative)

### Closed-Form Expressions
- Expressions based only on invariants and loop counters
- Like ScalarEvolution / MLIR Affine expressions / isl_pw_aff

### Array Detection
- Identify non-aliasing address ranges/base pointers (AliasSetTracker/Assumption)
- Derive array subscripts (GetElementPtr/MLIR MemRef/Delinearization)

### Dependency Analysis
- Each identify statements that do NOT use a resource
- Data-flow sweep over statements for avoid pairwise comparison
void func(int I, int J, int A[], int B) {
  #pragma unroll_and_jam
  for (int i = 0; i < I; i+=1) {
    int sum = 0;
    for (int j = 0; j < J; j+=1)
      sum += B[j];
    A[i] = sum;
  }
}

void func(int I, int J, int A[], int B[]) {
  for (int i = 0; i < I; i+=2) {
    int sum1 = 0;
    int sum2 = 0;
    for (int j = 0; j < J; j+=1)
      sum1 += B[j];
      sum2 += B[j];
    A[i] = sum1;
    A[i+1] = sum2;
  }
}
Sequence 'func' ScalarReads=[I, J, A, B] ScalarWrites=[]

Loop 'for.outer' ScalarReads=[B, A] ScalarWrites=[] ScalarRecurrences=[i]
\[ i = \text{isfirst} \, ? \, 0 : (i + 1) \]

Loop 'for.inner' ScalarReads=[B] ScalarWrites=[sum] ScalarRecurrences=[j]
\[ j = \text{isfirst} \, ? \, 0 : (j + 1) \]
\[ ld = \text{load i32} \ldots \]
\[ \text{getelementptr inbounds} \, \text{i32, i32* B, i32 j} \]
\[ \text{sum} = 0 + \text{ReduceAdd} \, ld \]
\[ \text{store i32 sum, i32*} \ldots \]
\[ \text{getelementptr inbounds} \, \text{i32, i32* A, i32 i} \]
Illustration
Scalar Dependencies

Sequence 'func' ScalarReads=[I, J, A, B] ScalarWrites=[]

Loop 'for.outer' ScalarReads=[B, A] ScalarWrites=[] ScalarRecurrences=[i]
\[ i = \text{isfirst} \ ? \ 0 : (i + 1) \]

Loop 'for.inner' ScalarReads=[B] ScalarWrites=[sum] ScalarRecurrences=[j]
\[ j = \text{isfirst} \ ? \ 0 : (j +1) \]

\[ ld = \text{load i32 } \ldots \]
\[ \text{getelementptr inbounds i32, i32* } B, \text{i32 } j \]
\[ \text{sum} = 0 + \text{ReduceAdd } ld \]

\[ \text{store i32 sum, i32* } \ldots \]
\[ \text{getelementptr inbounds i32, i32* } A, \text{i32 } i \]
Scalar Dependencies

Sequence 'func' ScalarReads=[I, J, A, B] ScalarWrites=[]

Loop 'for.outer' ScalarReads=[B, A] ScalarWrites=[] ScalarRecurrences=[i]
  i = isfirst ? 0 : (i + 1)

Loop 'for.inner' ScalarReads=[B] ScalarWrites=[sum] ScalarRecurrences=[j]
  j = isfirst ? 0 : (j + 1)
  ld = load i32 ...
  getelementptr inbounds i32, i32* B, i32 j
  sum = 0 + ReduceAdd ld
  store i32 sum, i32* ...
  getelementptr inbounds i32, i32* A, i32 i

i = i' + 1
  i = i' j = j' + 1
  i = i' j = j'
Array Dependencies

Sequence 'func' ScalarReads=[I, J, A, B] ScalarWrites=[]

Loop 'for.outer' ScalarReads=[B, A] ScalarWrites=[] ScalarRecurrences=[i]
\[ i = \text{isfirst} ? 0 : (i + 1) \]

Loop 'for.inner' ScalarReads=[B] ScalarWrites=[sum] ScalarRecurrences=[j]
\[ j = \text{isfirst} ? 0 : (j + 1) \]

\[ \text{ld} = \text{load i32} \ldots \]
\[ \text{getelementptr inbounds i32, i32* B, i32 j} \]
\[ \text{sum} = 0 + \text{ReduceAdd ld} \]

\[ \text{store i32 sum, i32*} \ldots \]
\[ \text{getelementptr inbounds i32, i32* A, i32 i} \]
Sequence 'func' ScalarReads=[I, J, A, B] ScalarWrites=[]

Loop 'for.outer' ScalarReads=[B, A] ScalarWrites=[] ScalarRecurrences=[i]

\[ i = \text{isfirst} \ ? 0 : (i + 1) \]

Loop 'for.inner' ScalarReads=[B] ScalarWrites=[sum] ScalarRecurrences=[j]

\[ j = \text{isfirst} \ ? 0 : (j +1) \]

\[ ld = \text{load i32} \ldots \]

\[ \text{getelementptr inbounds i32, i32* B, i32 j} \]

\[ \text{sum} = 0 + \text{ReduceAdd} \ ld \]

\[ \text{store i32 sum, i32*} \ldots \]

\[ \text{getelementptr inbounds i32, i32* A, i32 i} \]

Output
Sequence 'func' ScalarReads=[I, J, A, B] ScalarWrites=[]

Loop 'for.outer' ScalarReads=[B, A] ScalarWrites=[] ScalarRecurrences=[i]
  \( i = \text{isfirst} \ ? \ 0 \ : \ (i + 1) \)

Loop 'for.inner' ScalarReads=[B] ScalarWrites=[sum] ScalarRecurrences=[j]
  \( j = \text{isfirst} \ ? \ 0 \ : \ (j + 1) \)
  \( \text{ld} = \text{load} \ i32 \ldots \)
  \( \text{getelementptr inbounds} \ i32, \ i32* \ B, \ i32 \ j \)
  \( \text{sum} = 0 + \text{ReduceAdd} \ \text{ld} \)

store i32 sum, i32* \ldots \)
  \( \text{getelementptr inbounds} \ i32, \ i32* \ A, \ i32 \ i \)
Sequence 'unroll-and-jammed' ScalarReads: [I, J, A, B] ScalarWrites: []

Loop ScalarReads: [B, A] ScalarWrites: [] ScalarRecurrences: []
- \( i = \text{isfirst} ? 0 : (i + 2) \)

Loop ScalarReads: [B] ScalarWrites: [sum1, sum2] ScalarRecurrences: []
- \( j = \text{isfirst} ? 0 : (j + 1) \)

Sequence ScalarReads: [B, j] ScalarWrites: [ld1]
- \( ld1 = ld \)

Sequence ScalarReads: [B, j] ScalarWrites: [ld2]
- \( ld2 = ld \)

Sequence ScalarReads: [A, i, sum1] ScalarWrites: []
- \( i = ii \)
- \( sum = sum1 \)

Sequence ScalarReads: [A, i, sum2] ScalarWrites: []
- \( i = ii + 1 \)
- \( sum = sum2 \)

Sequence 'func' ScalarReads: [I, J, A, B] ScalarWrites: []

Loop 'for.outer' ScalarReads: [B, A] ScalarWrites: [] ScalarRecurrences: [i]
- \( i = \text{isfirst} ? 0 : (i + 1) \)

Loop 'for.inner' ScalarReads: [B] ScalarWrites: [sum] ScalarRecurrences: [j]
- \( j = \text{isfirst} ? 0 : (j + 1) \)

\( i = \text{load i32} \ldots \)
\( \text{getelementptr inbounds i32, i32* B, i32 j} \)

\( \text{sum} = 0 + \text{ReduceAdd} \ldots \)
\( \text{store i32 sum, i32*} \ldots \)
\( \text{getelementptr inbounds i32, i32* A, i32 i} \)

\( \text{for (int ii = 0; ii < I; ii+=2)} \)
- \( i = \{\} \)
- \( i = \{\} \)
- \( \{\} + 1 \)
- \( ii \)
- \( \text{Body(i)} \)
## Object Count

<table>
<thead>
<tr>
<th>LLVM-IR</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Blocks</td>
<td>Instructions</td>
</tr>
<tr>
<td>7</td>
<td>23</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Green/Red Tree</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Green Nodes</td>
<td>Red Nodes</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
</tr>
</tbody>
</table>
## Object Count

### LLVM-IR

<table>
<thead>
<tr>
<th>Basic Blocks</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>1015</td>
</tr>
</tbody>
</table>

### Green/Red Tree

<table>
<thead>
<tr>
<th>Green Nodes</th>
<th>Red Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1003</td>
<td>4</td>
</tr>
</tbody>
</table>
# Object Count

## LLVM-IR

<table>
<thead>
<tr>
<th>Basic Blocks</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>1015</td>
</tr>
</tbody>
</table>

16x Unroll(-And-Jam)

<table>
<thead>
<tr>
<th>Basic Blocks</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>16015</td>
</tr>
</tbody>
</table>

## Green/Red Tree

<table>
<thead>
<tr>
<th>Loop Tree</th>
<th>IR-Gen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Green Nodes</td>
<td>Red Nodes</td>
</tr>
<tr>
<td>1003</td>
<td>4</td>
</tr>
</tbody>
</table>

16x Unroll(-And-Jam)

<table>
<thead>
<tr>
<th>Loop Tree</th>
<th>IR-Gen</th>
</tr>
</thead>
<tbody>
<tr>
<td>1040</td>
<td>20</td>
</tr>
</tbody>
</table>
# Object Count

## LLVM-IR

<table>
<thead>
<tr>
<th></th>
<th>Basic Blocks</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unroll-And-Jam Illustration</td>
<td>7</td>
<td>1015</td>
</tr>
<tr>
<td>Speculative Copy</td>
<td>10</td>
<td>2023</td>
</tr>
<tr>
<td>4x Versioning</td>
<td>68</td>
<td>256183</td>
</tr>
</tbody>
</table>

## Green/Red Tree

<table>
<thead>
<tr>
<th></th>
<th>Loop Tree</th>
<th>IR-Gen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Green Nodes</td>
<td>BBs</td>
<td>Insts</td>
</tr>
<tr>
<td>Speculative Copy</td>
<td>1003</td>
<td>4</td>
</tr>
<tr>
<td>4x Transformations</td>
<td>1003</td>
<td>4 15 2017</td>
</tr>
</tbody>
</table>
Central Goals & Ideas

- Representation raising
- Cheap Copies
  - Generic legality and profitability analyses
- Loop-centric rather than instruction-centric
  - Decoupled from base IR (LLVM-IR or MLIR)
  - Treat scalars and array elements as similar as possible (e.g.: no SSA)
  - No difference between PHI and select
- Avoid dependencies
  - Reduction operations
  - No anti/output dependencies from scalars
- Predicates instead acyclic control-flow
- Sequence is a loop with exactly iteration

Status

- Experimenting with tree representation
- Working round-trip
- Currently making dependence analysis work

Got Interested?

Interested in collaborating? Contact me!
mkruse@anl.gov
That's all Folks!
This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, in particular its subproject PROTEAS-TUNE.

This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

This material was based in part upon funding from the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.