# A Proposal for A Framework for More Effective Loop Optimization

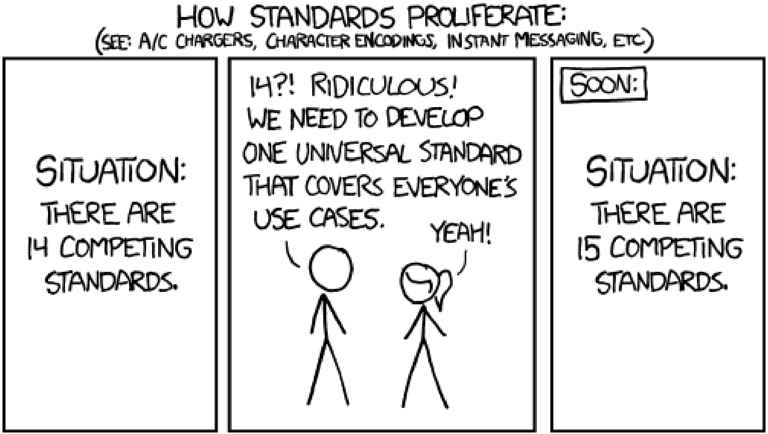
## LLVM Developer's Meeting 2020

Michael Kruse, Hal Finkel

Argonne Leadership Computing Facility
Argonne National Laboratory

2020-10-06

# Enough Justification?




Randall Munroe: https://xkcd.com/927/

- NOT an universal solution for everyone's use case
- However, there is overlapping functionality
  - LLVM-IR, Machine-IR, VPLAN, MLIR, ...

# The Good, The Bad, and The Ugly


https://www.youtube.com/watch?v=QpvZt9w-Jik

# Complexity of Writing a New Loop Pass

- LoopDistribute: 1063 lines
- LoopInterchange: 1529 lines
- LoopUnroll: 2025 lines
- LoopIdiom: 1794 lines

## High-Level Difficulties

- Ensure legality (incl. Dependencies: LoopAccessInfo, MemoryDependenceAnalysis, MemorySSA, ...)
- Machine profitability model

## Low-Level Troubles

- Preserve control flow
- Preserve (LC-)SSA
- Preserve passes (LoopInfo, DominatorTree, ScalarEvolution, ...)

# Loop Version Explosion

## Original Source

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

# Loop Version Explosion

## Optimize Outer Loop (1 transformation so far)

```
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)

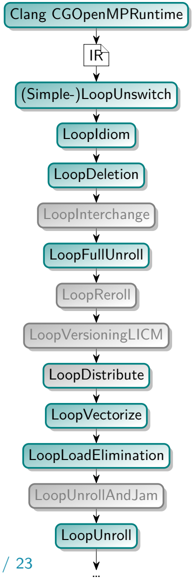
```
if (rtc1) {
  if (rtc2) {
    for (int i1 = 0; i1 < n; i1+=4) /* 2x transformed */
      for (int j = 0; j < m; j+=1)
        for (int i2 = 0; i2 < 4; i2+=1) /* new loop */
          Stmt(i1+i2,j);
  } else {
    for (int i = 0; i < n; i+=1) /* 1x transformed */
      for (int j = 0; j < m; j+=1)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

# Loop Version Explosion

## Optimize Inner Loop (3 transformations so far)

```
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);
      }
    }
  }
}
```

# Static Loop Pipeline




- Fixed transformation order
- May conflict with user directives:

```
#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];
  }
```

# 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];
```

Loop Interchange

```
for (int j=0; j<m; j+=1)
  for (int i=0; i<n; i+=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;
}
```

GVN (LoadPRE)

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

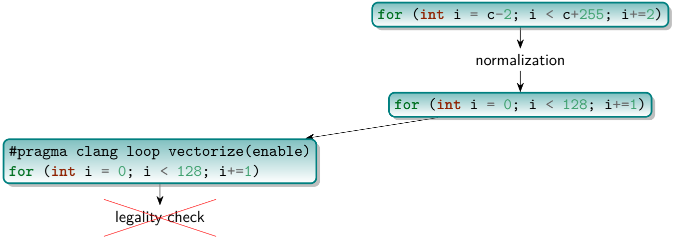
## Pessimizing Normal Forms

- LoopRotation
- IndVarSimplify

## Conflicting Normal Forms

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

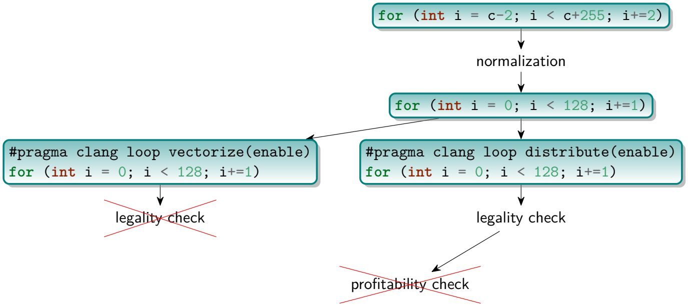
# What If... Copying IR Was Cheap?




## Advantage

- Generic legality & profitability analysis on transformed code
  - Passes don't need to implement themselves
  - Cheap heuristics can still be applied beforehand

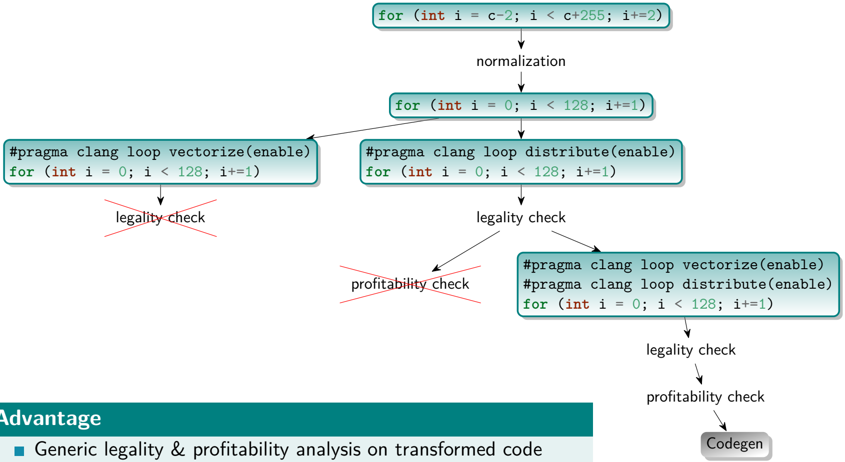
# What If... Copying IR Was Cheap?




## Advantage

- Generic legality & profitability analysis on transformed code
  - Passes don't need to implement themselves
  - Cheap heuristics can still be applied beforehand

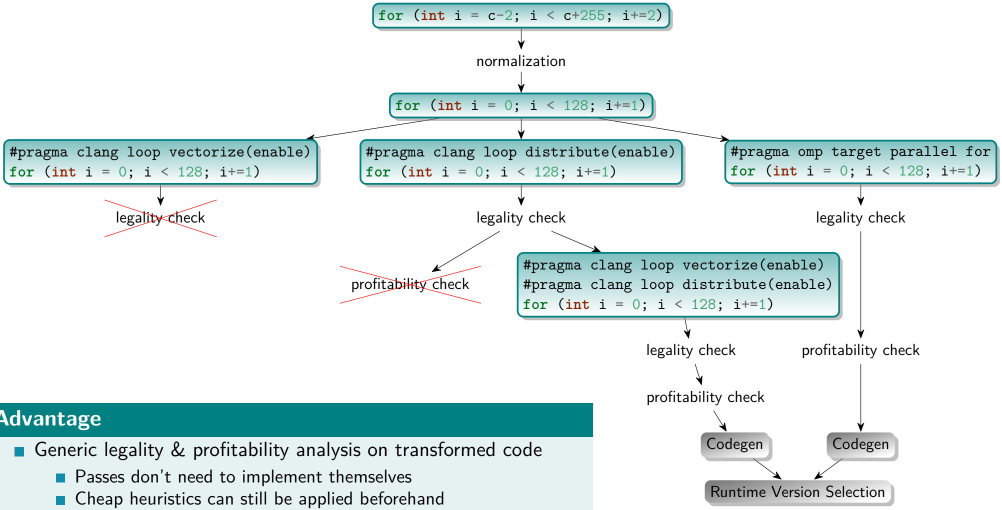
# What If... Copying IR Was Cheap?




## Advantage

- Generic legality & profitability analysis on transformed code
  - Passes don't need to implement themselves
  - Cheap heuristics can still be applied beforehand

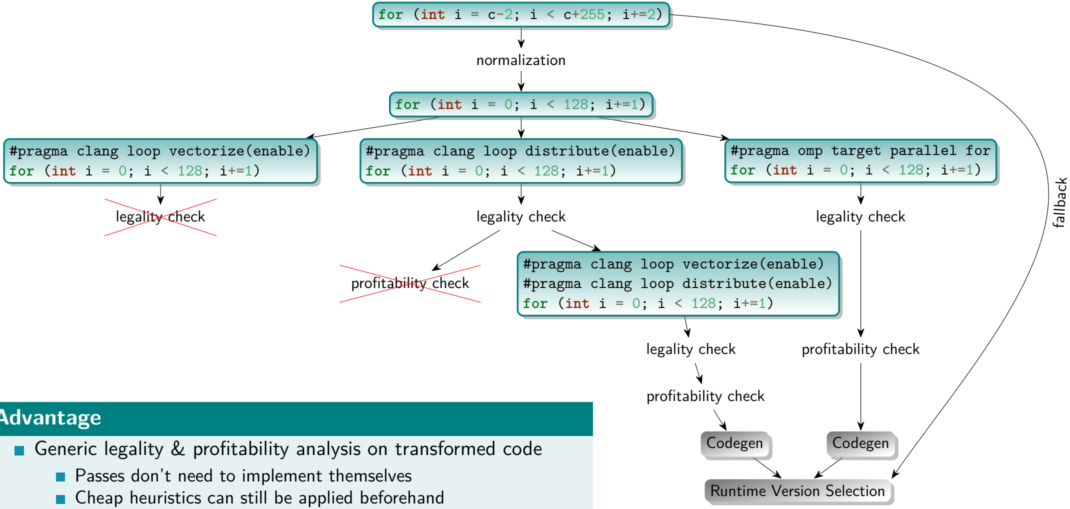
# What If... Copying IR Was Cheap?




## 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

# 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

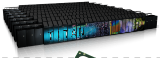
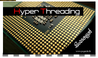
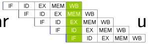
## Autotuning

- Select most-promising not-yet-evaluated
  – or –
- Select know-fastest

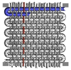
## Machine Learning

- Apply a per-architecture pre-trained model

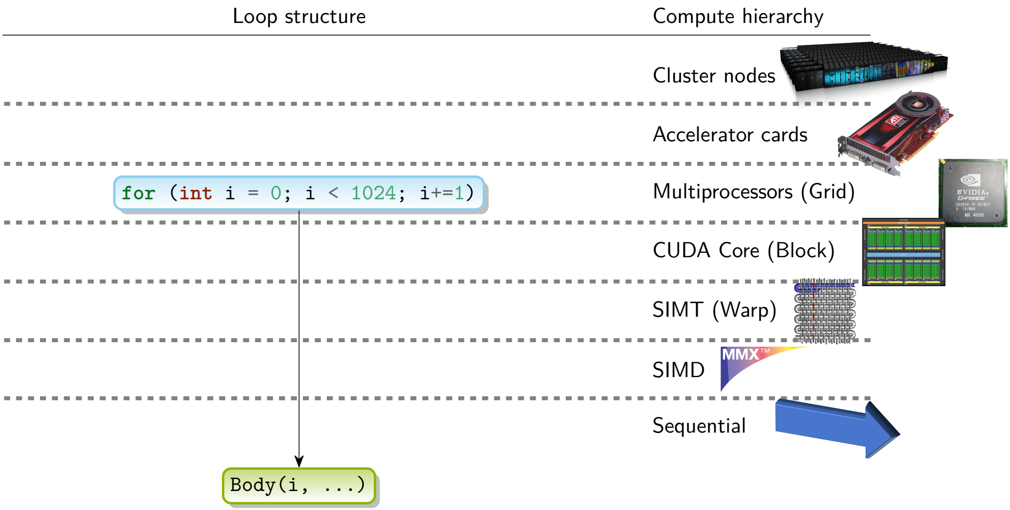
# CPU Compute Hierarchy

| Loop structure | Compute hierarchy | | |
|---|---|---|---|
| | Cluster nodes |  | MPI/PGAS |
| | Processors |  | thread-parallelization |
| | Cores |  | |
| | SMT |  | |
| `for (int i = 0; i < 1024; i+=1)` ? | SIMD | MMX™ | vectorization |
| | Superscalar |  | unrolling+instruction scheduling |
| `Body(i, ...)` | Sequential | | |

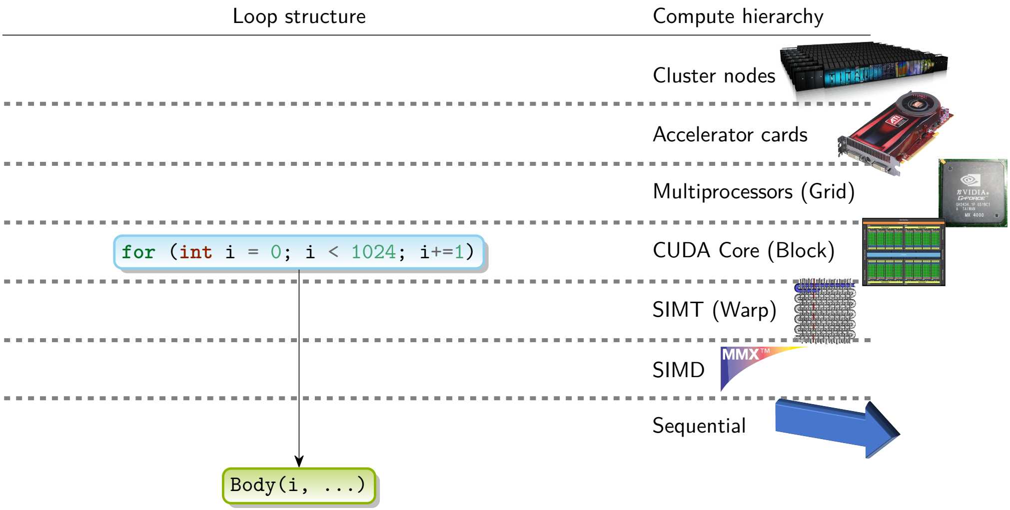
# GPU Hierarchy Mapping

| Loop structure | Compute hierarchy |
|---|---|
| | Cluster nodes  |
| | Accelerator cards  |
| | Multiprocessors (Grid)  |
| `for (int i = 0; i < 1024; i+=1)` ? | CUDA Core (Block)  |
| ↓ | SIMT (Warp)  |
| | SIMD |
| `Body(i, ...)` | Sequential |

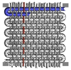
# GPU Hierarchy Mapping



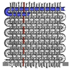
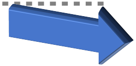
# GPU Hierarchy Mapping



# GPU Hierarchy Mapping

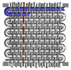
| Loop structure | Compute hierarchy |
|---|---|
| | Cluster nodes  |
| | Accelerator cards  |
| | Multiprocessors (Grid)  |
| | CUDA Core (Block)  |
| `for (int i = 0; i < 1024; i+=1)` | SIMT (Warp)  |
| | SIMD |
| `Body(i, ...)` | Sequential |

# GPU Hierarchy Mapping

| Loop structure | Compute hierarchy |
| --- | --- |
| | Cluster nodes  |
| | Accelerator cards  |
| `for (int floor = 0; floor < 1024; floor+=128)` | Multiprocessors (Grid)  |
| `for (int tile = floor; tile < floor+128; tile+=1)` | CUDA Core (Block)  |
| | SIMT (Warp)  |
| | SIMD |
| | Sequential  |

```
Body(tile, ...)
```

# GPU Hierarchy Mapping

| Loop structure | Compute hierarchy |
|---|---|
| | Cluster nodes  |
| | Accelerator cards  |
| `for (int floor = 0; floor < 1024; floor+=128)` | Multiprocessors (Grid)  |
| `for (int t1 = floor; t1 < floor+128; t1+=32)` | CUDA Core (Block)  |
| `for (int t2 = t1; t2 < t1+32; t2+=1)` | SIMT (Warp)  |
| | SIMD |
| | Sequential |
| `Body(t2, ...)` | |

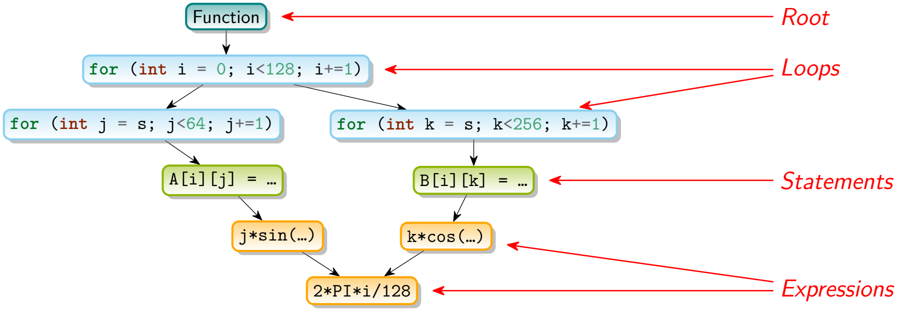
# Loop Tree

## An Old Idea

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

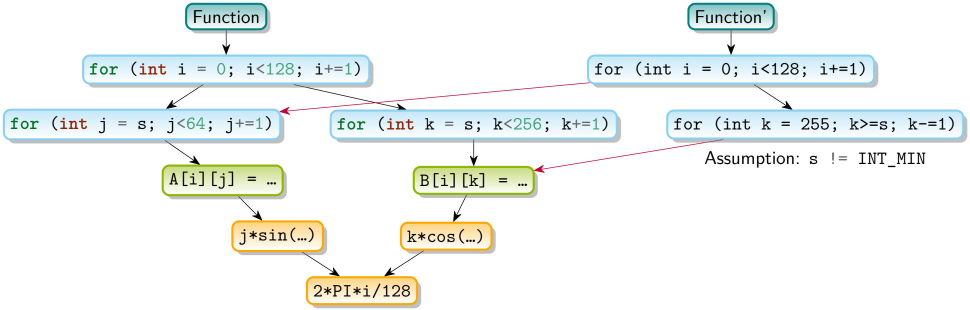
# Loop Hierarchy DAG

```
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);
    }
}
```

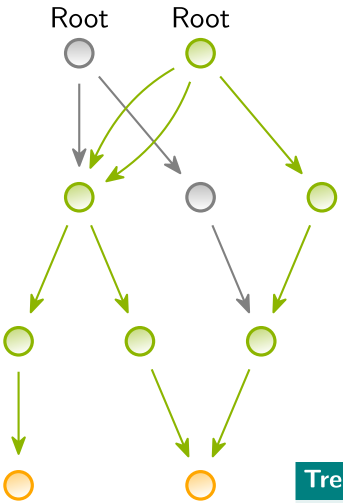


# Loop Hierarchy DAG

```
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);
    }
}
```



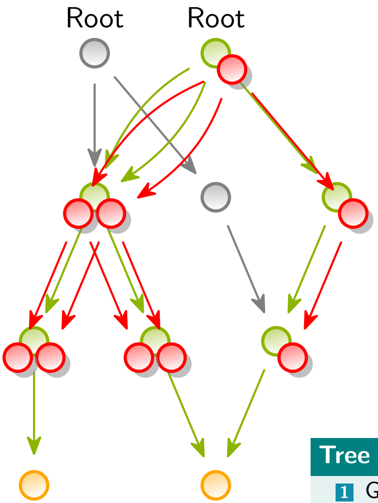
# Green/Red/Blue Tree




**Tree Types**

1. **Green tree: Source of truth**

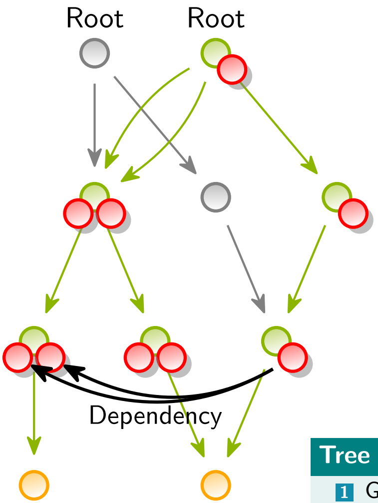
# Green/Red/Blue Tree




**Tree Types**

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

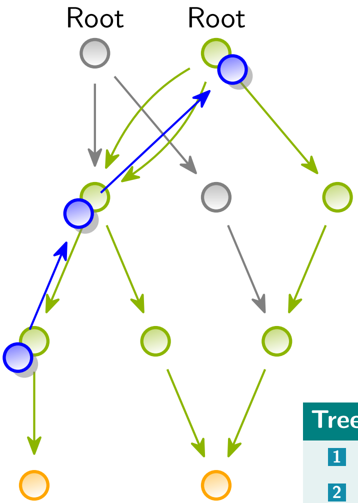
# Green/Red/Blue Tree




**Tree Types**

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

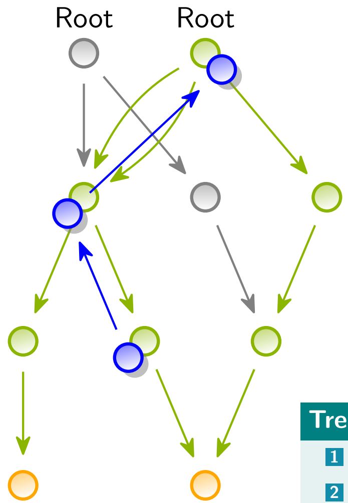
# Green/Red/Blue Tree




**Tree Types**

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

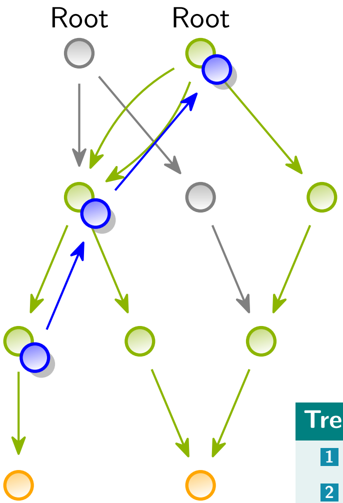
# Green/Red/Blue Tree




**Tree Types**

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

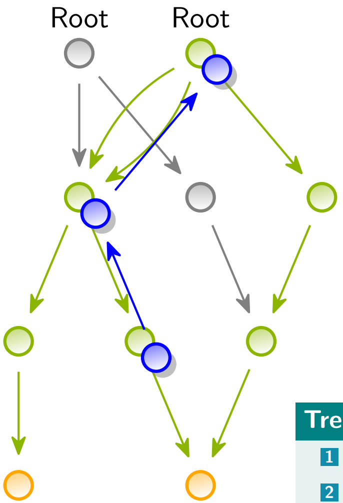
# Green/Red/Blue Tree




**Tree Types**

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

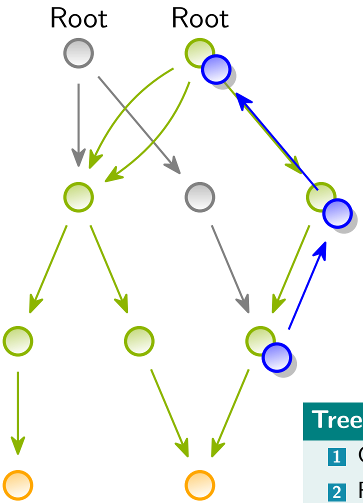
# Green/Red/Blue Tree




**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

| IR Construct | Raised to | Assumptions |
|---|---|---|
| LLVM instruction | | |
| MLIR operation | Generic `lof::Operation` | |
| ... | | |

# Operation Lifting

| IR Construct | Raised to | Assumptions |
|---|---|---|
| LLVM instruction<br>MLIR operation<br>... | Generic `lof::Operation` | |
| Two's complement arithmetic<br>MLIR index<br>... | Infinite precision arithmetic | No integer overflow/wrap |

# Operation Lifting

| IR Construct | Raised to | Assumptions |
|---|---|---|
| LLVM instruction<br>MLIR operation<br>... | Generic `lof::Operation` | |
| Two's complement arithmetic<br>MLIR index<br>... | Infinite precision arithmetic | No integer overflow/wrap |
| `select`<br>PHI node<br>MLIR BB parameter | `lof::Operation::nop`<br>(assignment+condition encoded in nodes) | |

# Operation Lifting

| IR Construct | Raised to | Assumptions |
|---|---|---|
| LLVM instruction<br>MLIR operation<br>... | Generic `lof::Operation` | |
| Two's complement arithmetic<br>MLIR index<br>... | Infinite precision arithmetic | No integer overflow/wrap |
| `select`<br>PHI node<br>MLIR BB parameter | `lof::Operation::nop`<br>(assignment+condition encoded in nodes) | |
| `llvm::LoadInst`<br>MLIR memref `store`<br>`memcpy`<br>... | Array subscripts | No aliasing of memory range<br>Subscripts within shape |

# Operation Lifting

| IR Construct | Raised to | Assumptions |
| --- | --- | --- |
| LLVM instruction<br>MLIR operation<br>... | Generic `lof::Operation` | |
| Two's complement arithmetic<br>MLIR index<br>... | Infinite precision arithmetic | No integer overflow/wrap |
| `select`<br>PHI node<br>MLIR BB parameter | `lof::Operation::nop`<br>(assignment+condition encoded in nodes) | |
| `llvm::LoadInst`<br>MLIR memref `store`<br>`memcpy`<br>... | Array subscripts | No aliasing of memory range<br>Subscripts within shape |
| Loop-carried dependency of associative operation | Reduction | |

# 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

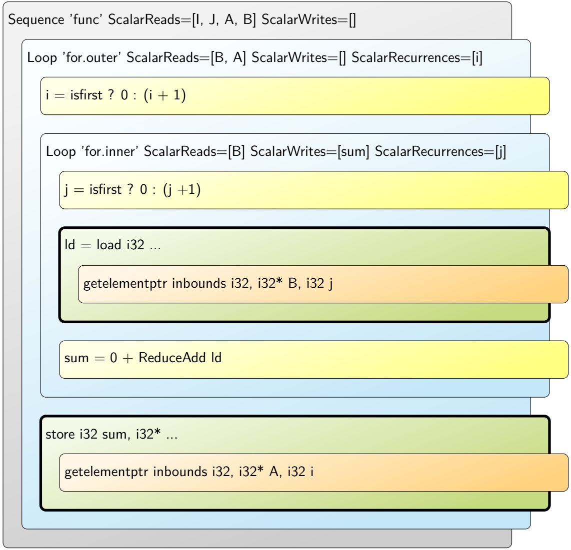
# Unroll-And-Jam Example

```
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; k < 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; k < J; j+=1) {
      sum1 += B[j];
      sum2 += B[j] ;
    }
    A[i] = sum1;
    A[i+1] = sum2;
  }
}
```

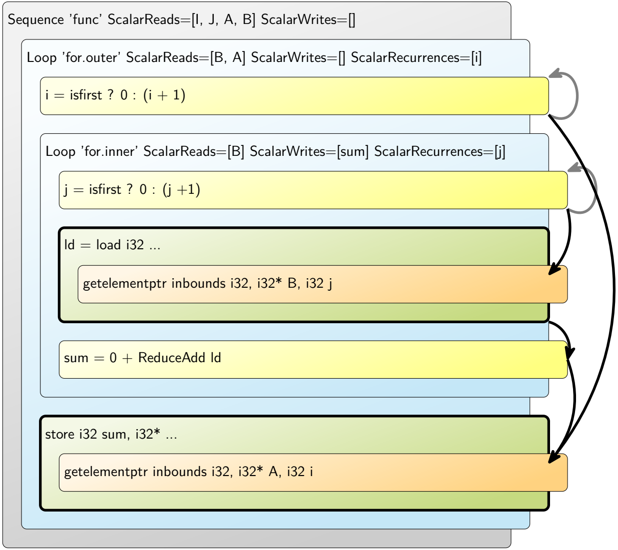
# Illustration

## Loop Hierarchy



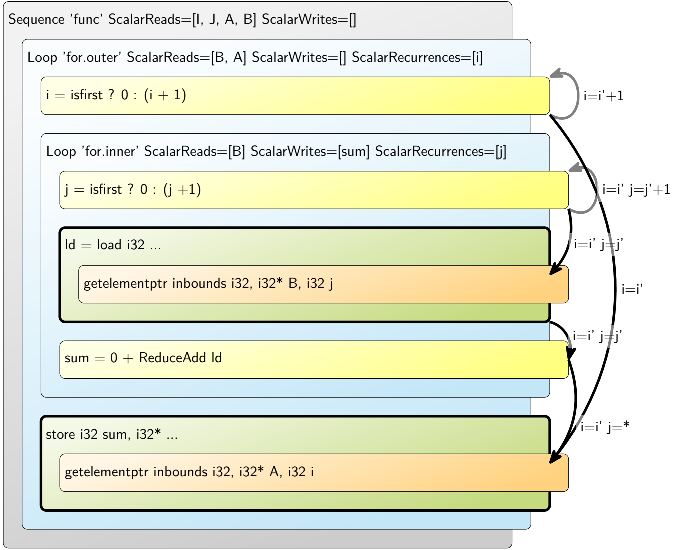
# Illustration

## Scalar Dependencies



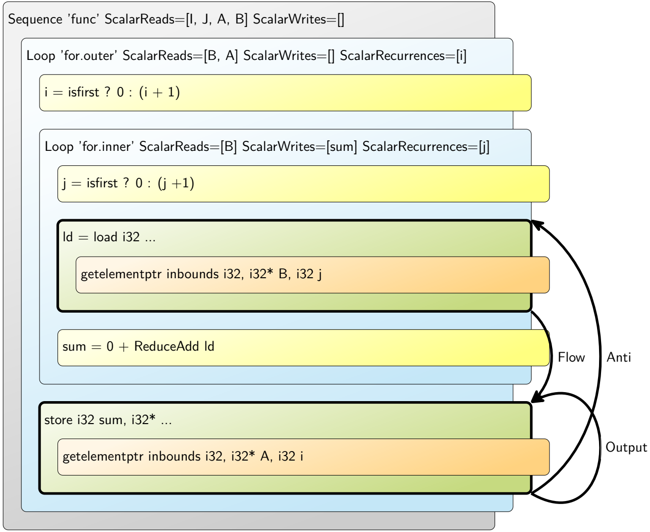
# Illustration

## Scalar Dependencies



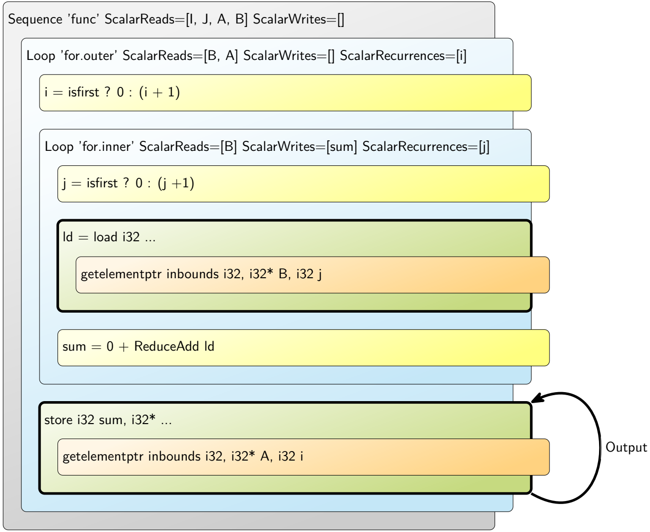
# Illustration

## Array Dependencies



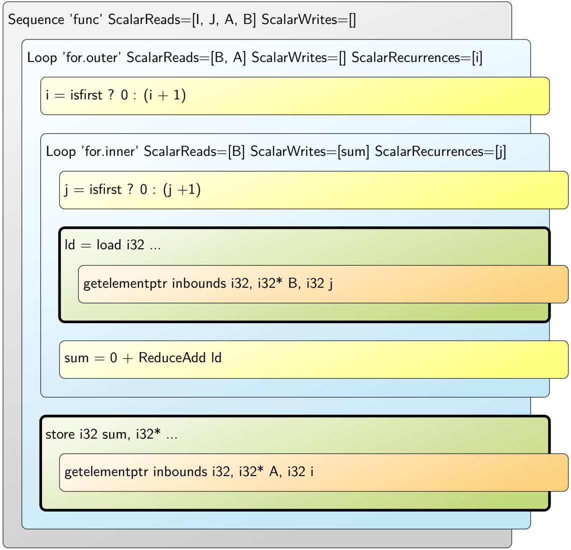
# Illustration

## Array Dependencies

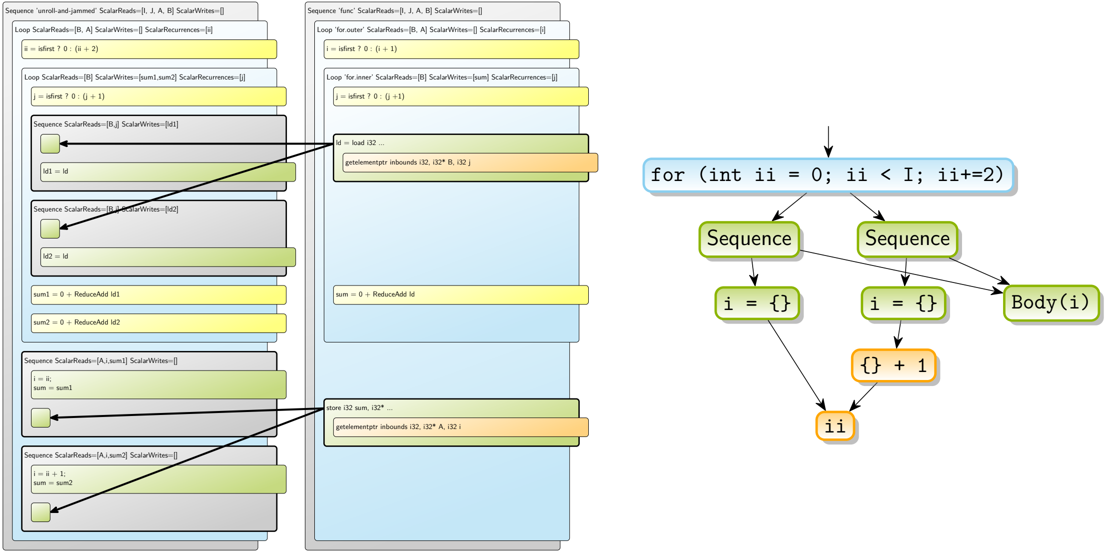


# Illustration

## Array Dependencies



# After Unroll-And-Jam



# Object Count

## LLVM-IR

| Basic Blocks | Instructions |
|---|---|
| 7 | 23 |

## Green/Red Tree

| Green Nodes | Red Nodes |
|---|---|
| 11 | 4 |

# Object Count

## LLVM-IR

| Basic Blocks | Instructions |
|---|---|
| 7 | 1015 |

## Green/Red Tree

| Green Nodes | Red Nodes |
|---|---|
| 1003 | 4 |

# Object Count

## LLVM-IR

| | Basic Blocks | Instructions |
|---|---|---|
| | 7 | 1015 |
| 16x Unroll(-And-Jam) | 7 | 16015 |

## Green/Red Tree

| | Loop Tree | | IR-Gen | |
|---|---|---|---|---|
| | Green Nodes | Red Nodes | BBs | Insts |
| | 1003 | 4 | | |
| 16x Unroll(-And-Jam) | 1040 | 20 | 7 | 16015 |

# Object Count

## LLVM-IR

| | Basic Blocks | Instructions |
|---|---|---|
| | 7 | 1015 |
| Speculative Copy | 10 | 2023 |
| 4x Versioning | 68 | 256183 |

## Green/Red Tree

| | Loop Tree | | IR-Gen | |
|---|---|---|---|---|
| | Green Nodes | Red Nodes | BBs | Insts |
| | 1003 | 4 | | |
| Speculative Copy | 1004 | 8 | | |
| 4x Transformations | 1003 | 4 | 15 | 2017 |

# 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!

# Acknowledgments

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.