Loop Optimizations in LLVM: The Good, The Bad, and The Ugly

Michael Kruse, Hal Finkel

Argonne Leadership Computing Facility
Argonne National Laboratory

18th October 2018

Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative.

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.

Table of Contents

Table of Contents

Loop Transformations in the Compiler?

Approaches

- Compiler-based
    - Automatic (Polly, ...)
    - Language extensions (OpenMP, OpenACC, ...)
        - Prescriptive
        - Descriptive
    - New languages (Chapel, X10, Fortress, UPC, ...)
- Source-to-Source (PLuTo, ROSE, PPCG, ...)
- Library-based
    - Hand-optimized (MKL, OpenBLAS, ...)
    - Templates (RAJA, Kokkos, HPX, Halide, ...)
    - Embedded DSL (Tensor Comprehensions, ...)
- Domain-Specific Languages and Compilers (QIRAL, SPIRAL, LIFT, SQL, ...)

Partial Unrolling

```
#pragma unroll 4
for (int i = 0; i < n; i += 1)
  Stmt(i);
```

```
if (n > 0) {
  for (int i = 0; i+3 < n; i += 4) {
    Stmt(i);
    Stmt(i + 1);
    Stmt(i + 2);
    Stmt(i + 3);
  }
  switch (n % 4) {
  case 3:
    Stmt(n - 3);
  case 2:
    Stmt(n - 2);
  case 1:
    Stmt(n - 1);
  }
}
```

- Why?
  - Compiler pragmas
    https://arxiv.org/abs/1805.03374
  - Optimization heuristics
  - Loop Autotuning
    https://github.com/kavon/atJIT

Compiler-Supported Pragmas

Compiler Loop Transformations are Here to Stay

Clang

```
#pragma unroll
#pragma clang loop unroll(enable)
#pragma unroll_and_jam
#pragma clang loop distribute(enable)
#pragma clang loop vectorize(enable)
#pragma clang loop interleave(enable)
```

gcc

```
#pragma GCC unroll
#pragma GCC ivdep
```

msvc

```
#pragma loop(hint_parallel(0))
#pragma loop(no_vector)
#pragma loop(ivdep)
```

Cray

```
#pragma _CRI unroll
#pragma _CRI fusion
#pragma _CRI nofission
#pragma _CRI blockingsize
#pragma _CRI interchange
#pragma _CRI collapse
```

OpenMP

```
#pragma omp simd
#pragma omp for
#pragma omp target
```

PGI

```
#pragma concur
#pragma vector
#pragma ivdep
#pragma nodepchk
```

xlc

```
#pragma unrollandfuse
#pragma stream_unroll
#pragma block_loop
#pragma loopid
```

SGI/Open64

```
#pragma fuse
#pragma fission
#pragma blocking size
#pragma altcode
#pragma noinvarif
#pragma mem prefetch
#pragma interchange
#pragma ivdep
```

OpenACC

```
#pragma acc kernels
```

icc

```
#pragma parallel
#pragma offload
#pragma unroll_and_jam
#pragma nofusion
#pragma distribute_point
#pragma simd
#pragma vector
#pragma swp
#pragma ivdep
#pragma loop_count(n)
```

Oracle Developer Studio

```
#pragma pipeloop
#pragma nomemorydepend
```

HP

```
#pragma UNROLL_FACTOR
#pragma IF_CONVERT
#pragma IVDEP
#pragma NODEPCHK
```

Table of Contents

Supported Loop Transformations

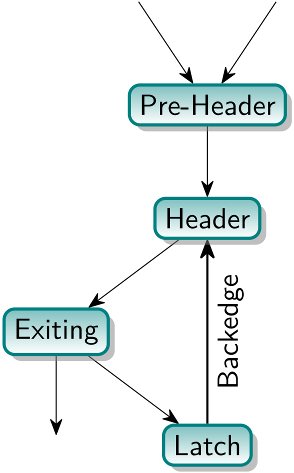
- Available passes:
    - Loop Unroll (-and-Jam)
    - Loop Unswitching
    - Loop Interchange
    - Detection of `memcpy`, `memset` idioms
    - Delete side-effect free loops
    - Loop Distribution
    - Loop Vectorization
- Modular: Can switch passes on and off independently

Supported Pragmas

- `#pragma clang loop unroll` / `#pragma unroll`
- `#pragma unrollandjam`
- `#pragma clang loop vectorize(enable)` / `#pragma omp simd`
- `#pragma clang loop interleave(enable)`
- `#pragma clang loop distribute(enable)`

Canonical Loop Form

- Loop-rotated form (at least one iteration)
  - Can hoist invariant loads
- Loop-Closed SSA



Available Infrastructure

Analysis passes:

- LoopInfo
- ScalarEvolution / PredicatedScalarEvolution

Preparation passes:

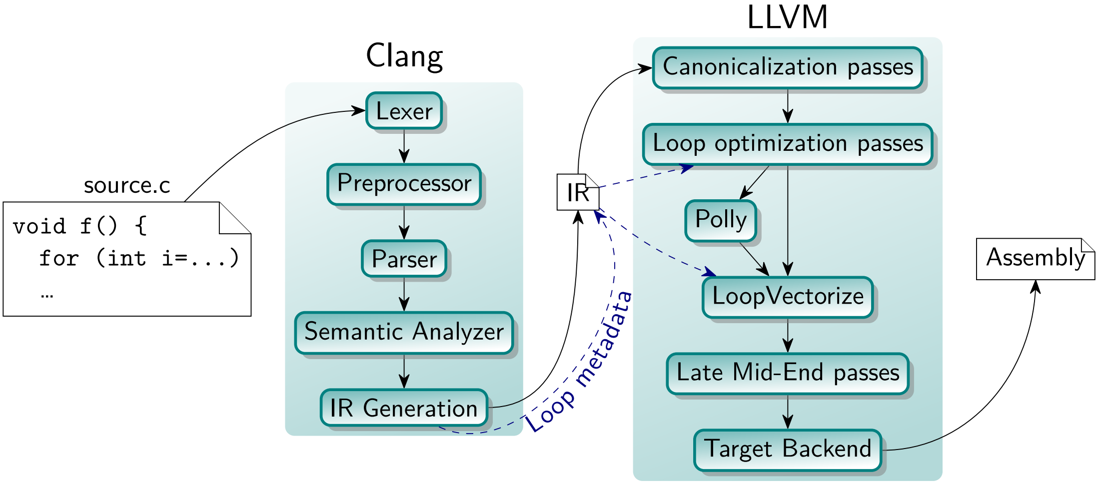
- LoopRotate
- LoopSimplify
- IndVarSimplify

Transformations:

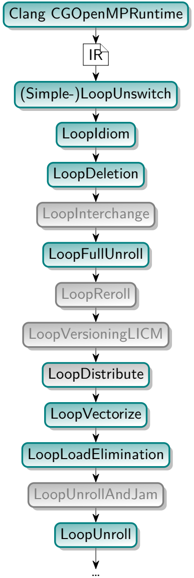
- LoopVersioning

Table of Contents

Clang/LLVM/Polly Compiler Pipeline



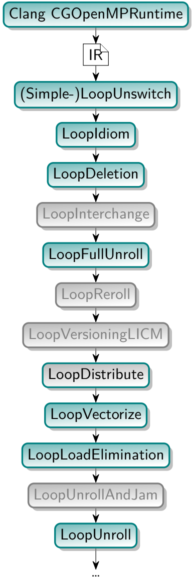
Unavailable Loop Passes



Clang CGOpenMPRuntime
IR
(Simple-)LoopUnswitch
LoopIdiom
LoopDeletion
LoopInterchange
LoopFullUnroll
LoopReroll
LoopVersioningLICM
LoopDistribute
LoopVectorize
LoopLoadElimination
LoopUnrollAndJam
LoopUnroll
...

- Many transformations disabled by default
    - Experimental / not yet matured

Static Loop Pipeline



- Fixed transformation order
  - OpenMP outlining happens first
    - Difficult to optimize afterwards
  - May conflict with source 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];
  }
```

  - OpenMP proposal: `https://arxiv.org/abs/1805.03374`

Composition of Transformations

```
#pragma unroll 2
#pragma reverse
for (int i = 0; i < 128; i+=1)
  Stmt(i);
```

```
#pragma unroll 2
for (int i = 127; i >= 0; i-=1)
  Stmt(i);
```

```
for (int i = 127; i >= 0; i-=1) {
  Stmt(i);
  Stmt(i-1);
}
```

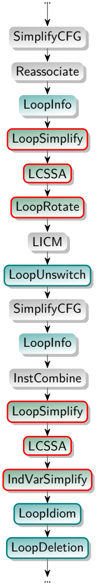
```
#pragma reverse
#pragma unroll 2
for (int i = 0; i < 128; i+=1)
  Stmt(i);
```

```
#pragma reverse
for (int i = 0; i < 128; i+=2) {
  Stmt(i);
  Stmt(i+1);
}
```

```
for (int i = 126; i >= 0; i-=2) {
  Stmt(i);
  Stmt(i+1);
}
```

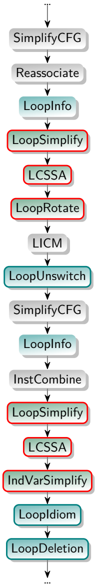
`https://reviews.llvm.org/D49281`

Non-Loop Passes Between Loop Passes



- Non-loop passes may destroy canonical loop structure
  - SimplifyCFG removes empty loop headers
    - keeps a list of loop headers
    - LoopSimplifyCFG only merges blocks within loop
    - Fixed in r343816
  - JumpThreading skips exiting blocks
    - has an integrated loop header detection
    - makes ScalarEvolution not recognize the loop
    - Fixed in r312664(?)
  - Bit-operations created by InstCombine must be understood by ScalarEvolution
- Analysis invalidation / Extra work in non-loop passes

Instruction Movement vs. Loop Transformations



- Scalar transformations making loop optimizations harder
  - Loop-Invariant Code Motion
  - Global Value Numbering
  - Loop-Closed SSA

Scalar/Loop Pass Interaction

Loop Nest Bakin-In

```
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;
}
```

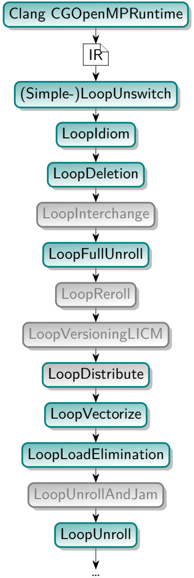
Loop
Interchange

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

GVN
(LoadPRE)

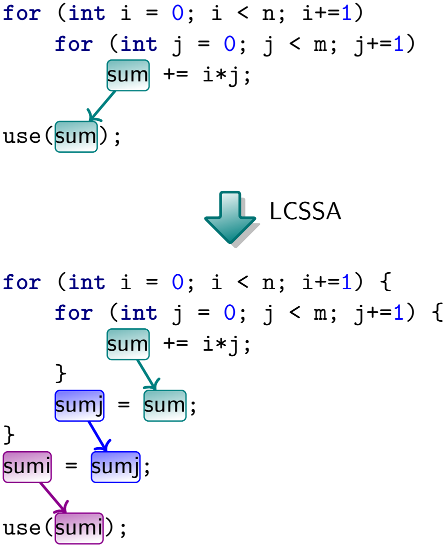
```
for (int j=0; j<m; j+=1) {
  tmp = B[j];
  for (int i=0; i<n; i+=1)
    A[i] += i*tmp;
}
```

Non-Shared Infrastructure



- Dependence analysis (not passes that can be preserved!):
  - LoopAccessInfo (LoopDistribute, LoopVectorize, LoopLoadElimination)
  - LoopInterchangeLegality (LoopInterchange)
  - MemoryDependenceAnalysis (LoopIdiom)
  - MemorySSA (LICM, LoopInstSimplify)
  - PolyhedralInfo
- Profitability:
  - LoopInterchangeProfitability
  - LoopVectorizationCostModel
  - UnrolledInstAnalyzer
- Code transformation

Loop-Closed SSA Form



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

use(sum);
```

LCSSA

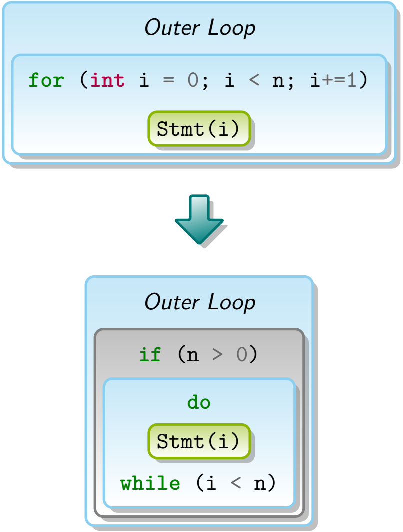
```
for (int i = 0; i < n; i+=1) {
    for (int j = 0; j < m; j+=1) {
        sum += i*j;
    }
    sumj = sum;
}
sumi = sumj;

use(sumi);
```

- Allows referencing the loop's exit value
    - Otherwise need to pass the loop every time
- Adds spurious dependencies
- Makes some (non-innermost) loop transformations more complicated

Loop-Rotated Normal Form in Tree Hierarchies

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

```
int i = 0;
if (n > 0) {
   do {
     Stmt(i);
     i+=1;
   } while (i < n);
}
```



Loop Pass Boilerplate

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

Low-level complexity:

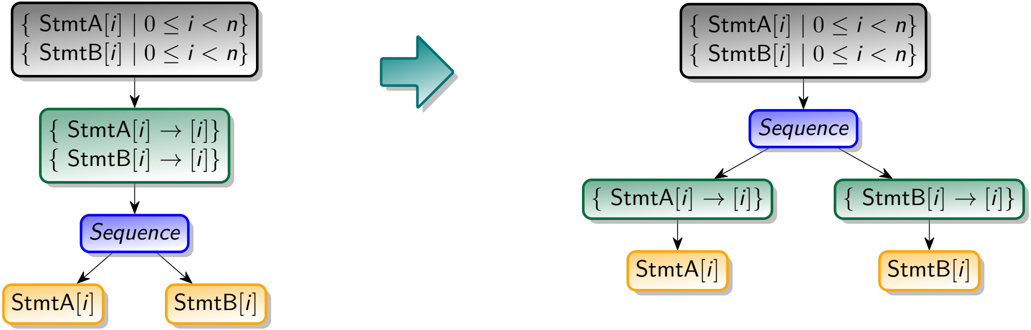
- Repair control flow
- Repair (LC-)SSA
- Preserve passes (LoopInfo, DominatorTree, ScalarEvolution, ...)

ISL Schedule Tree Transformation

Loop Distribution

```
for (int i = 0; i < n; i+=1) {
  StmtA(i);
  StmtB(i);
}
```

```
for (int i = 0; i < n; i+=1)
  StmtA(i);
for (int i = 0; i < n; i+=1)
  StmtB(i);
```



$\{\ \text{StmtA}[i] \mid 0 \leq i < n\}$
$\{\ \text{StmtB}[i] \mid 0 \leq i < n\}$

$\{\ \text{StmtA}[i] \to [i]\}$
$\{\ \text{StmtB}[i] \to [i]\}$

Sequence

StmtA[i]

StmtB[i]

$\{\ \text{StmtA}[i] \mid 0 \leq i < n\}$
$\{\ \text{StmtB}[i] \mid 0 \leq i < n\}$

Sequence

$\{\ \text{StmtA}[i] \to [i]\}$

$\{\ \text{StmtB}[i] \to [i]\}$

StmtA[i]

StmtB[i]

Polly Code for Loop Distribution

Transformation-Specific Code

```
isl::schedule_node distributeBand(isl::schedule_node Band, const Dependences &D) {
  auto Partial = isl::manage(isl_schedule_node_band_get_partial_schedule(Band.get()));
  auto n = Seq.n_children();

  // Transformation
  auto Seq = isl::manage(isl_schedule_node_delete(Band.release()));
  for (int i = 0; i < n; i+=1)
      Seq = Seq.get_child(i).insert_partial_schedule(Partial).parent();

  // Legality check
  if (!D.isValidSchedule(Seq.get_schedule()))
    return {};

  return Seq;
}
```

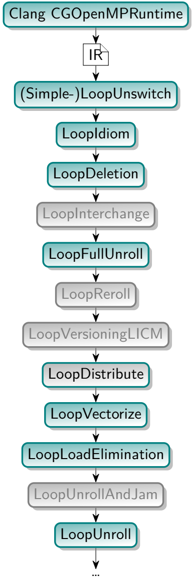
- Dependences *unchanged*
- LLVM LoopDistribute: 1529 lines

Miscellaneous

- Forced promotion of induction variable to 64 bits
  - Multiple induction variables not coalesced
- SCEVExpander strength-reduces everything
- LoopIDs are not identifying loops (`https://reviews.llvm.org/D52116`)
- No equivalent for LoopIDs
- Difference between PHI and `select` irrelevant for high-level purposes

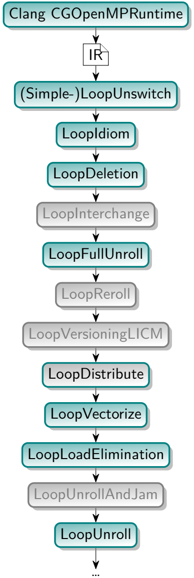
Table of Contents

Loop Profitability



- Profitability determined independently
- Transformations might only be profitable in combination
  - Strip-mining alone only adds overhead
  - Loop distribution/fusion vs. loop vectorizer
    - Loop distribute targets vectorizability, but does not know whether vectorization is profitable
    - Inverse problem for loop fusion
  - Loop Unroll vs. Unroll-And-Jam
    - If unroll is “forced”, then unroll, do not unroll-and-jam
    - If unroll-and-jam is “forced”, then unroll-and-jam
    - If unroll-and-jam is profitable, then unroll-and-jam
    - If unroll is profitable, then unroll

Loop Versioning



Clang CGOpenMPRuntime

IR

(Simple-)LoopUnswitch

LoopIdiom

LoopDeletion

LoopInterchange

LoopFullUnroll

LoopReroll

LoopVersioningLICM

LoopDistribute

LoopVectorize

LoopLoadElimination

LoopUnrollAndJam

LoopUnroll

...

- Multiple passes do code versioning
    - LoopVersioningLICM
    - LoopDistribute
    - LoopVectorize
    - LoopLoadElimination
- → up to $2^4 = 16$ copies of the same (innermost) loop
- Outer loop transformation fallbacks include inner loops

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);
  }
}
```

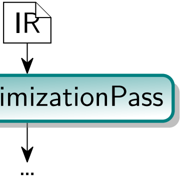
Loop Version Explosion

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);
      }
    }
  }
}
```

Table of Contents

Single Integrated Loop Pass



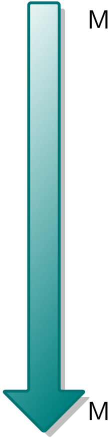
- Single pass in the pass pipeline
    - No interaction with scalar passes
    - No loop analysis invalidation
- Similar "passes" in LLVM:
    - VPlan
    - Machine pass manager

---

https://lists.llvm.org/pipermail/llvm-dev/2017-October/118125.html

Straightforward Optimization Heuristic

```
RedLoop optimizeLoop(RedLoop L) {
  if (L.hasPragma())
    return applyPragmas(L);

  if (L.isGEMM())
    return createCallToLibBLAS(L);

  if (L.canUnrollAndJam())
    L = L.unrollAndJam(TTI.getUnrollFactor());
  else
    L = L.unroll(TTI.getUnrollFactor());

  if (L.isParallelizable() && L.isProfitable())
    L = L.parallelize();

  return L;
}
```



More specific

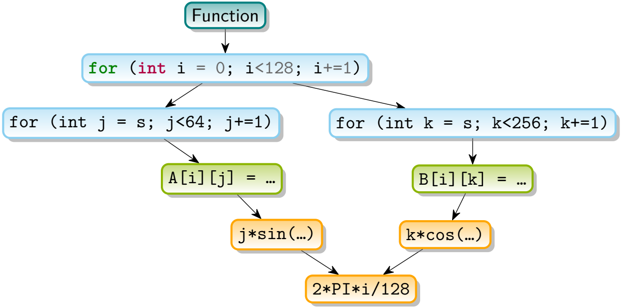
More general

Loop Structure DAG

- Use loop tree intermediate representation
  - Easily modifiable
  - Hierarchical
  - No bail-out (irreducible loops, exceptions, ...)
    - Irreducible loops can be converted to reducible loop by some code duplication
    - For other difficult constructs, loop can be marked as non-regular
- Three types of nodes
  - Loops (repeat something)
  - Statements (with side-effects)
  - Expressions (floating)

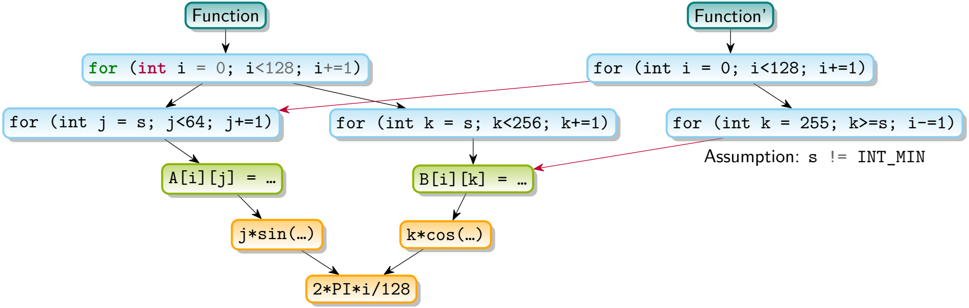
Loop Structure DAG

```
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 Structure 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);
    }
}
```



Red-Green Tree

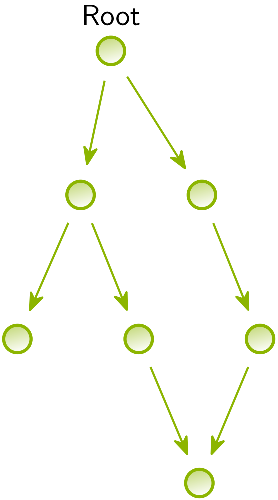
- Used by Roslyn's C# compiler
  - Immutable subtrees
  - Easy modification
  - Cheap copy
  - Create multiple variant, and chose most profitable

---

https://blogs.msdn.microsoft.com/ericlippert/2012/06/08/persistence-facades-and-roslyns-red-green-trees/
https://github.com/dotnet/roslyn/blob/master/src/Compilers/Core/Portable/Syntax/GreenNode.cs

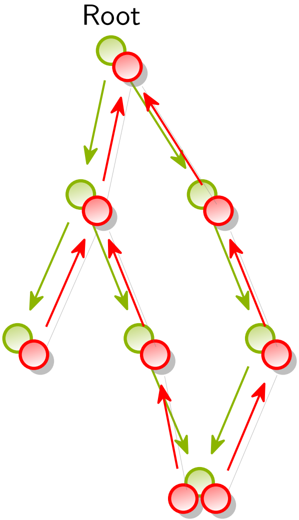
Red-Green Tree

The Green DAG



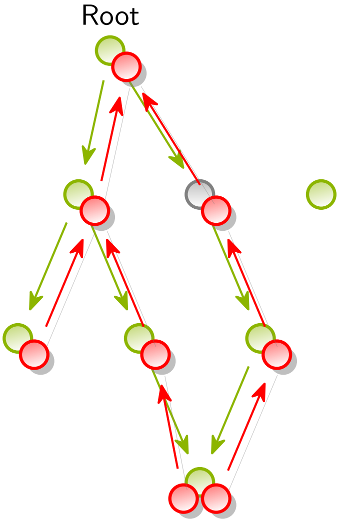
Red-Green Tree

The Red Tree



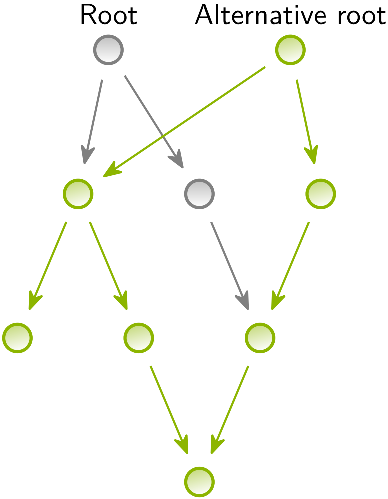
Red-Green Tree

Modify a Node



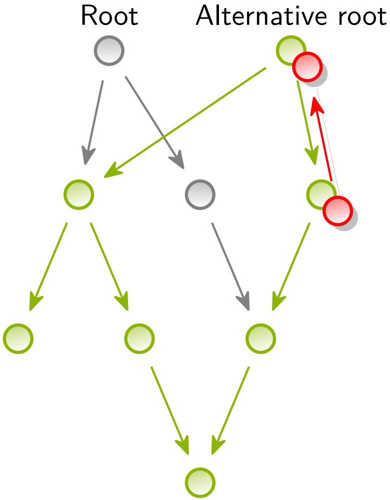
Red-Green Tree

Rebuild Green Tree Reusing Nodes



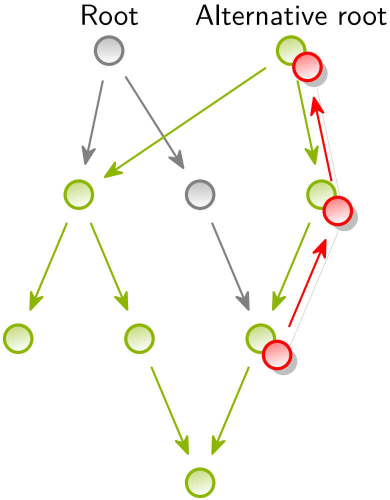
Red-Green Tree

Recreate Red Nodes on Demand



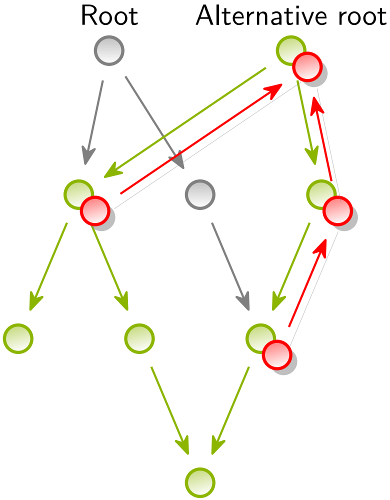
Red-Green Tree

Recreate Red Nodes on Demand

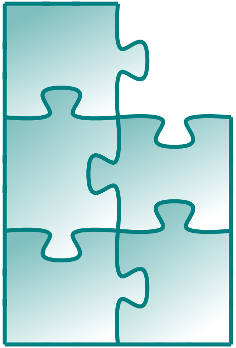


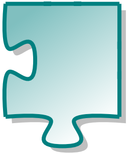
Red-Green Tree

Recreate Red Nodes on Demand

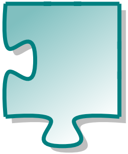


Closed-Form Expressions

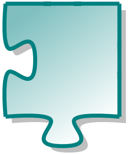




ScalarEvolution −01

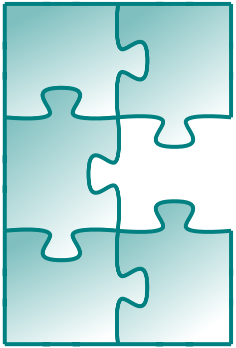


PredicatedScalarEvolution −02



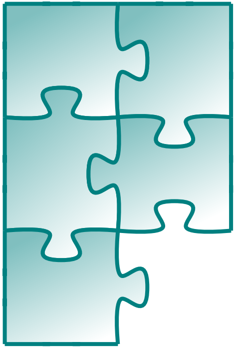
PolyhedralValueAnalysis −03

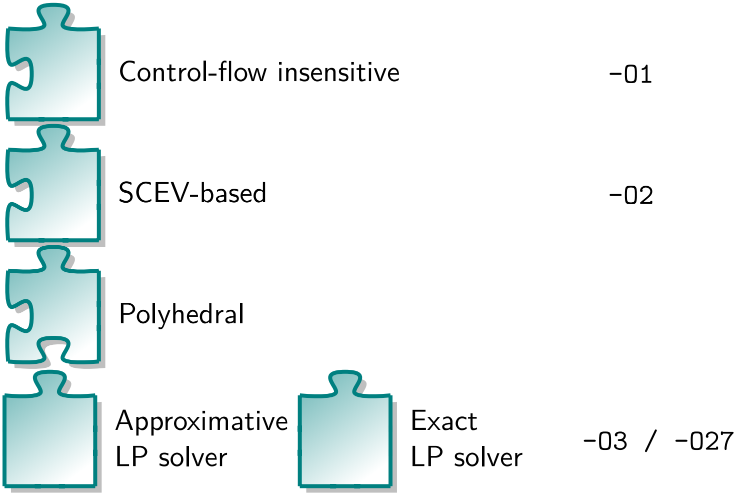
Access Analysis



One-dimensional	–01
One-dimensional, allow additional assumptions	–02
Multi-dimensional, allow additional assumptions	–03

Dependency Analysis





Control-flow insensitive `-O1`

SCEV-based `-O2`

Polyhedral

Approximative
LP solver

Exact
LP solver `-O3 / -O27`

Dependency Analysis

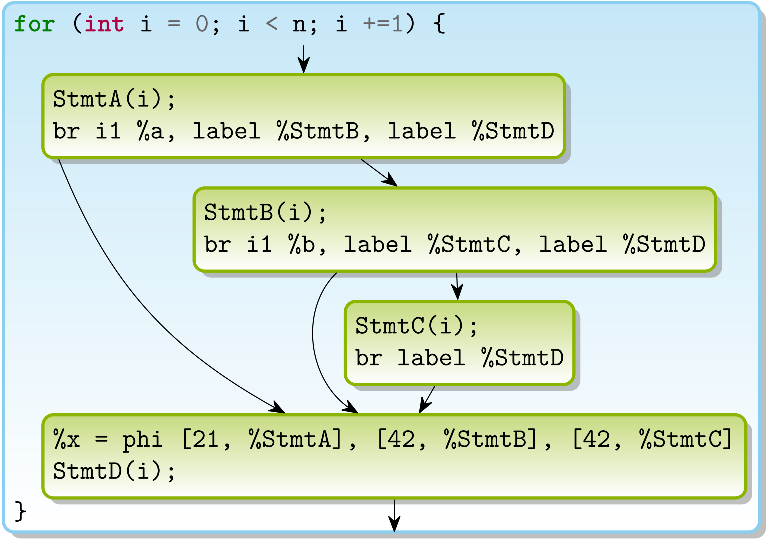
- Special purpose dependency types
  - Flow-, Anti-dependencies
    - No need for output-dependencies when anti-dependencies to a virtual return node
  - Memory clobber
  - Register dependencies (due to SSA)
  - Control dependencies (execute on if/on else flags)
- Register/Control dependencies may be backed by array storage if necessary
  - For instance, loop distribution crossing a def-use chain
  - Optimizer responsible for ensuring memory usage remains reasonable

Non-Cyclic Control Flow

- Predicated preferred
  - Simpler to handle: Sequential Root: →Loop→Sequential→Loop→Sequential→...
  - Corresponds SIMT model
  - Statements have execution conditions
    - Must execute conditions
    - May execute conditions (allow speculative execution)
  - Can be converted back to branching control flow
  - Makes PHI and `select` instructions the same
  - Difficulty: Branch out of loop to multiple targets (**`break`**, **`return`**)

Non-Cyclic Control Flow

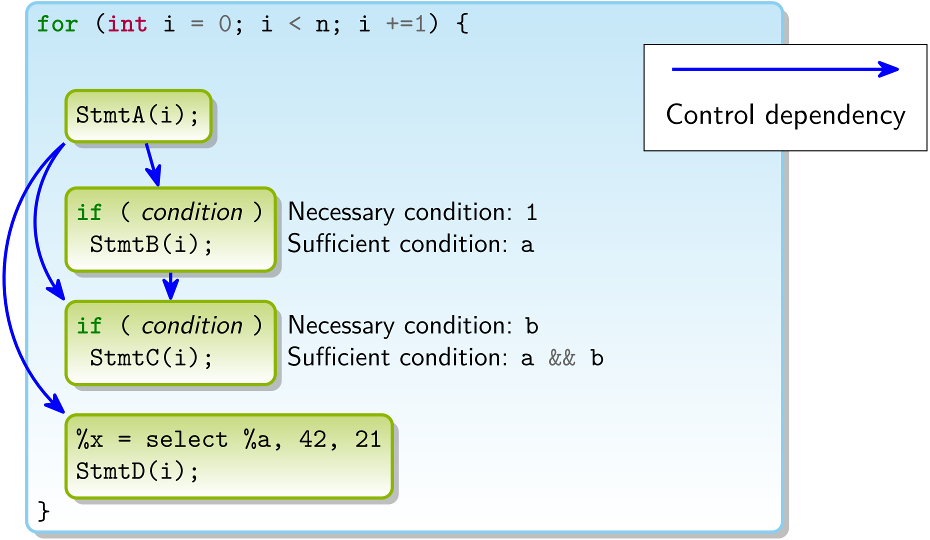
CFG Inside Loops



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

StmtA(i);
br i1 %a, label %StmtB, label %StmtD

StmtB(i);
br i1 %b, label %StmtC, label %StmtD

StmtC(i);
br label %StmtD

%x = phi [21, %StmtA], [42, %StmtB], [42, %StmtC]
StmtD(i);
}
```

Non-Cyclic Control Flow

Sequential, but Conditional



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

  StmtA(i);

  if ( condition )
   StmtB(i);

  if ( condition )
   StmtC(i);

  %x = select %a, 42, 21
  StmtD(i);
}
```

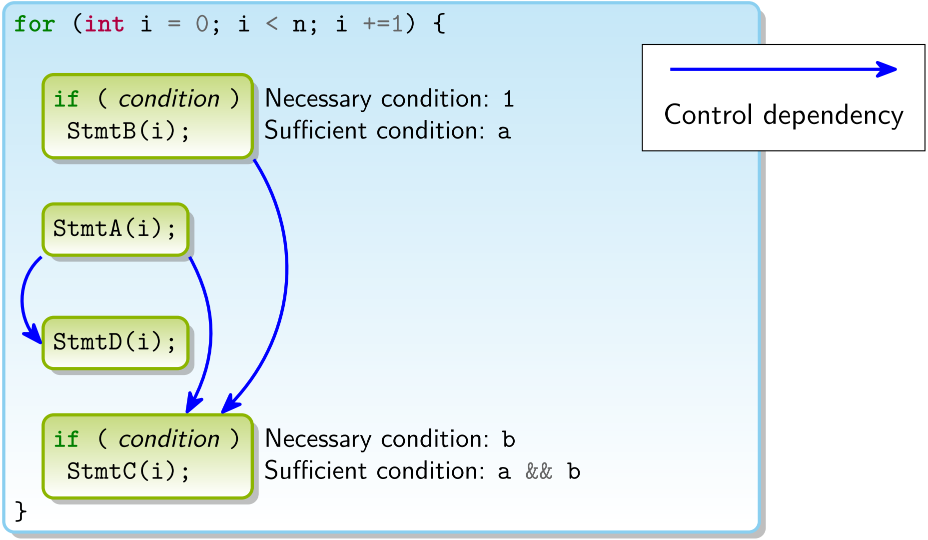
Necessary condition: 1
Sufficient condition: a

Necessary condition: b
Sufficient condition: a && b

Control dependency

Non-Cyclic Control Flow

Statement Reordering



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

    if ( condition )
     StmtB(i);

    StmtA(i);

    StmtD(i);

    if ( condition )
     StmtC(i);
}
```

Necessary condition: 1
Sufficient condition: a

Necessary condition: b
Sufficient condition: a && b

Control dependency

Non-Cyclic Control Flow

Loop Distribution

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

    if ( condition )
     StmtB(i);


    StmtA(i);
}
```

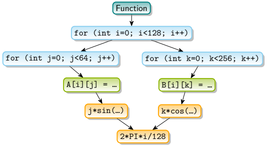
Necessary condition: 1
Sufficient condition: a

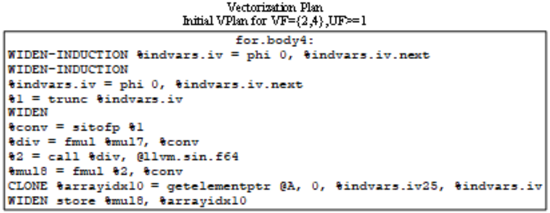
```
for (int i = 0; i < n; i +=1) {
    StmtD(i);


    if ( condition )
     StmtC(i);
}
```

Necessary condition: b
Sufficient condition: a && b

Code Generation

- Only emit modified subtrees
- Collect assumptions for runtime checks
- Recover non-cyclic control flow





Vectorization Plan
Initial VPlan for VF={2,4},UF>=1

```
for.body4:
WIDEN-INDUCTION %indvars.iv = phi 0, %indvars.iv.next
WIDEN-INDUCTION
%indvars.iv = phi 0, %indvars.iv.next
%1 = trunc %indvars.iv
WIDEN
%conv = sitofp %1
%div = fmul %mul7, %conv
%2 = call %div, @llvm.sin.f64
%mul8 = fmul %2, %conv
CLONE %arrayidx10 = getelementptr @A, 0, %indvars.iv25, %indvars.iv
WIDEN store %mul8, %arrayidx10
```

```
for.body4:
  %indvars.iv = phi i64 [ 127, %for.cond1.preheader ], [ %indvars.iv.next, %for.body4 ]
  %1 = trunc i64 %indvars.iv to i32
  %conv = sitofp i32 %1 to double
  %div = fmul fast double %mul7, %conv
  %2 = tail call fast double @llvm.cos.f64(double %div)
  %mul8 = fmul fast double %2, %conv
  %arrayidx10 = getelementptr inbounds [128 x double]* @B, i64 0, i64 %indvars.iv24, i64 %indvars.iv
  store double %mul8, double* %arrayidx10, align 8, !tbaa !5
  %indvars.iv.next = add nsw i64 %indvars.iv, -1
  %cmp2 = icmp eq i64 %indvars.iv, 0
  br i1 %cmp2, label %for.cond.cleanup3, label %for.body4, !llvm.loop !9
```

Pipeline

1. Create DAG from IR (lazy expansion)
2. Canonicalization
3. Analysis
    - Closed-form expressions
    - Array accesses
    - Dependencies
    - Idiom recognition
4. Transform
    - User-directives `#pragma`
    - Optimization heuristics
    - Using MINLP solver (polyhedral)
5. Cost model: Choose green tree root
6. Code Generation
    - To LLVM-IR
    - To VPlan

Summary

- LLVM not designed with loop optimizations in mind
  - Pass pipeline design
  - Normalized IR form
  - Non-shared infrastructure
  - Separate profitability analysis
  - Code version explosion
- Proposed solution:
  - Single integrated pass
  - Shared infrastructure
  - Loop hierarchy DAG
  - Red-Green Tree
  - If-converted normal from
  - Generate to LLVM-IR or VPlan
- Similar work
  - Every optimizing compiler with loop transformations
  - Silicon Graphics: *Loop Nest Optimization* (LNO)
    - Source available as part of Open64
  - IBM: *ASTI* and *Loop Structure Graph* (LSG) for xlf `https://www.doi.org/10.1147/rd.413.0233`
  - Intel: *VPlan* for LLVM
  - isl's *Schedule Trees* `https://hal.inria.fr/hal-00911894`

- Kit Barton (IBM), 3pm: "Revisiting Loop Fusion, and its place in the loop transformation framework"

That's all Folks!

LLVM Loop Passes

Excluding Normalization Passes

LLVM Pass	Metadata
(Simple-)LoopUnswitch	*none*
LoopIdiom	*none*
LoopDeletion	*none*
LoopInterchange*	*none*
SimpleLoopUnroll	`llvm.loop.unroll.*`
LoopReroll*	*none*
LoopVersioningLICM^{+*}	`llvm.loop.licm_versioning.disable`
LoopDistribute^{+}	`llvm.loop.distribute.enable`

LoopVectorize^{+}	`llvm.loop.vectorize.*`
	`llvm.loop.interleave.count`
	`llvm.loop.isvectorized`

LoopLoadElimination^{+}	*none*
LoopUnrollAndJam*	`llvm.loop.unroll_and_jam.*`
LoopUnroll	`llvm.loop.unroll.*`

The Polyhedral Model

```
for (int i=1; i<5; i++)
  for (int j=1; i+j<6; j++)
    S(i,j);
```

The Polyhedral Model

$\{S(i,j) \mid 0 < i,j \wedge i + j < 6\}$

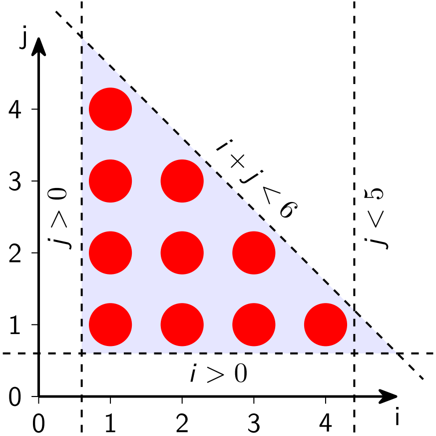
$S(1,1)$, $S(1,2)$, $S(1,3)$, $S(1,4)$, $S(2,1)$, $S(2,2)$, $S(2,3)$, $S(3,1)$, $S(3,2)$, $S(4,1)$

```
for (int i=1; i<5; i++)
  for (int j=1; i+j<6; j++)
    S(i,j);
```

The Polyhedral Model

$$\{S(i,j) \mid 0 < i,j \wedge i+j < 6\}$$

$S(1,1)$, $S(1,2)$, $S(1,3)$, $S(1,4)$, $S(2,1)$, $S(2,2)$, $S(2,3)$, $S(3,1)$, $S(3,2)$, $S(4,1)$

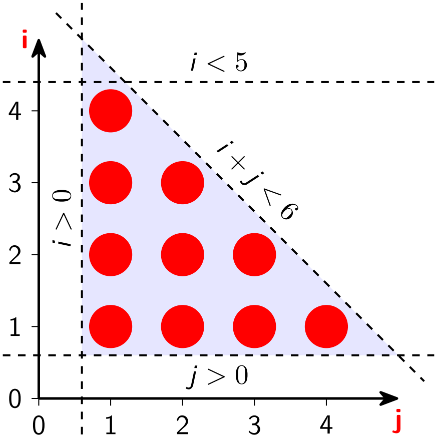


```
for (int i=1; i<5; i++)
  for (int j=1; i+j<6; j++)
    S(i,j);
```

The Polyhedral Model

Loop Interchange

$$S(i, j) \mapsto (j, i)$$



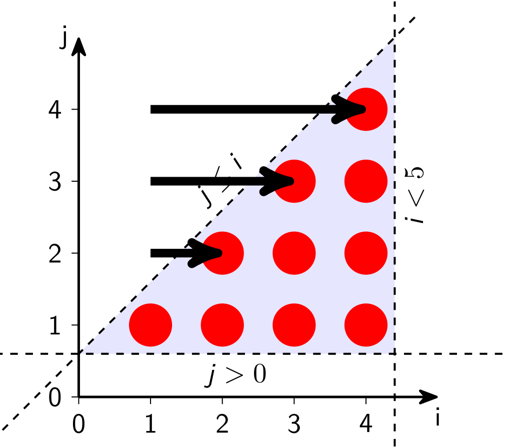
```
for (int j=1; j<5; j++)
  for (int i=1; i+j<6; i++)
    S(i,j);
```

Bonus

The Polyhedral Model

Skewing (Wavefronting)

$$S(i,j) \mapsto (i, i+j-1)$$

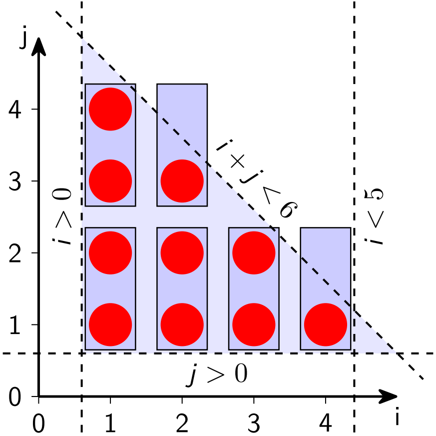


```
for (int i=1; i<5; i++)
  for (int j=i; j<5; j++)
    S(i,j-i+1);
```

The Polyhedral Model

Strip Mining (Vectorization)

$$S(i,j) \mapsto (i, j/2, j \textbf{ mod } 2)$$

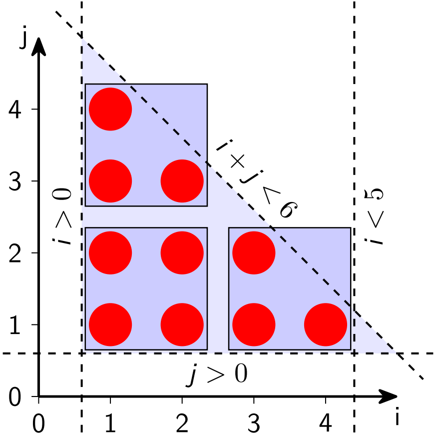


```
for (int i=1; i<5; i++)
  for (int t=1; i+t<6; t+=2)
    for (int j=t; j<t+2 && i+j<6; j++)
      S(i,j);
```

The Polyhedral Model

Tiling

$$S(i,j) \mapsto (i/2, j/2, i \ \mathbf{mod}\ 2, j \ \mathbf{mod}\ 2)$$

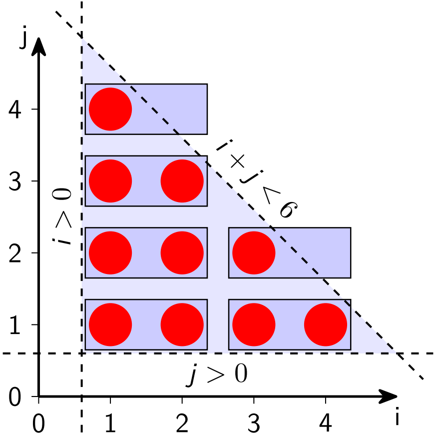


```
for (int s=1; s<5; s+=2)
  for (int t=1; s+t<6; t+=2)
    for (int i=s; i<s+2 && i<5; i++)
      for (int j=t; j<t+2 && i+j<6; j++)
        S(i,j);
```

The Polyhedral Model

Strip Mining (Outer Loop Vectorization)

$$S(i,j) \mapsto (i/2, j, i \textbf{ mod } 2)$$

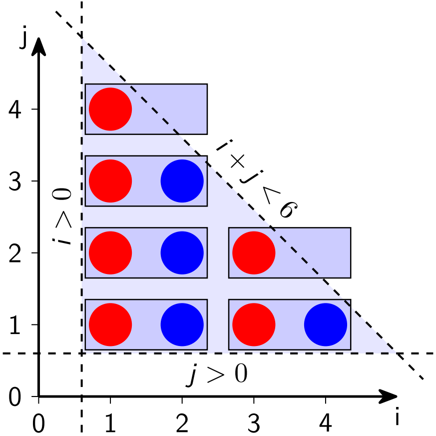


```
for (int t=1; t<5; t+=2)
  for (int j=1; i+t<6; j++)
    for (int i=t; i<t+2 && j+i<6; i++)
      S(i,j);
```

The Polyhedral Model

Unroll-and-Jam

$$S(i,j) \mapsto \begin{cases} (i/2,j,0) & \text{if } i \textbf{ mod } 2 = 0 \\ (i/2,j,1) & \text{if } i \textbf{ mod } 2 = 1 \end{cases}$$



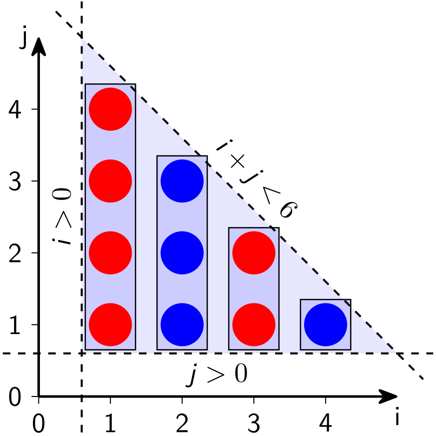
```
for (int i=1; i<5; i+=2)
  for (int j=1; i+j<6; j++) {
    S(i,j);
    if (i+j+1<6)
      S(i+1,j);
  }
```

Bonus

The Polyhedral Model

Loop Distribution

$$S(i,j) \mapsto \begin{cases} (i/2,0,j) & \text{if } i \bmod 2 = 0 \\ (i/2,1,j) & \text{if } i \bmod 2 = 1 \end{cases}$$

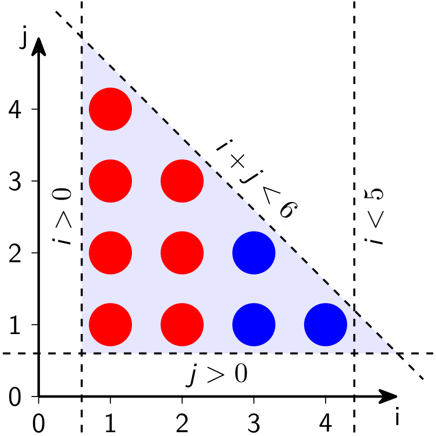


```
for (int i=1; i<5; i++) {
  for (int j=1; i+j<6; j+=2)
    S(i,j);
  for (int j=2; i+j<6; j+=2)
    S(i,j);
}
```

The Polyhedral Model

Index Set Splitting

$$S(i,j) \mapsto \begin{cases} (0, i, j) & \text{if } i < 3 \\ (1, i, j) & \text{if } i \geq 3 \end{cases}$$

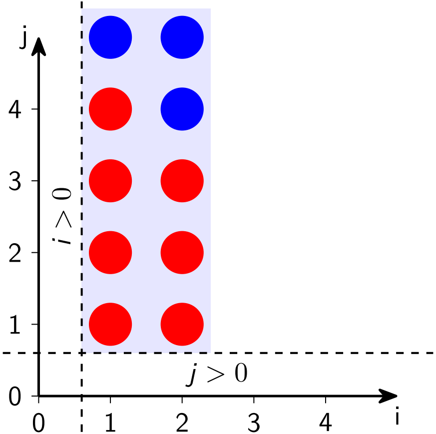


```
for (int i=1; i<3; i++)
  for (int j=1; i+j<6; j++)
    S(i,j);
for (int i=3; i<5; i++)
  for (int j=1; i+j<6; j++)
    S(i,j);
```

The Polyhedral Model

“Loop Fusion”

$$S(i,j) \mapsto \begin{cases} (i,j) & \text{if } i < 3 \\ (5-i, 6-j) & \text{if } i \geq 3 \end{cases}$$



```
for (int i=1; i<3; i++)
  for (int j=1; j<6; j++)
    if (i+j<6)
      S(i,j);
    else
      S(5-i,6-j);
```

Polly Solution to Everything?

- Scalar Dependencies
- Only Single-Entry-Single-Exit regions
- Non-affine loop bounds
- Non-affine control flow is atomic
- Statically infinite loops
- No exceptions (incl. mayThow and invoke)
- No VLAs inside loops
- Complexity limits
- Checkable aliasing
- Profitability heuristics always apply
- Always detect and codegen the max compatible regions
- Unpredictable loop bodies

When do Loop Optimization?

- After inlining
- Before parallel outlining (OpenMP)
- Before vectorization
- Before LICM, LoadPRE
- Before LoopRotate

Polly Code for Loop Reversal

From OpenMP Prototype Implementation

```
isl::schedule applyLoopReversal(isl::schedule_node BandToReverse) {
  auto PartialSched = isl::manage(
    isl_schedule_node_band_get_partial_schedule(BandToReverse.get()));
  auto MPA = PartialSched.get_union_pw_aff(0);
  auto Neg = MPA.neg();
  auto Node = isl::manage(isl_schedule_node_delete(BandToReverse.copy()));
  Node = Node.insert_partial_schedule(Neg);

  return Node;
}
```

From OpenMP Prototype Implementation

```
isl::schedule_node interchangeBands(isl::schedule_node Band, ArrayRef<LoopIdentification> NewOrder) {
  auto NumBands = NewOrder.size();
  Band = moveToBandMark(Band);
  SmallVector<isl::schedule_node, 4> OldBands;

  // Scan loops
  int NumRemoved = 0;
  int NodesToRemove = 0;
  auto BandIt = Band;
  while (true) {
    if (NumRemoved >= NumBands)
      break;

    if (isl_schedule_node_get_type(BandIt.get()) == isl_schedule_node_band) {
      OldBands.push_back(BandIt);
      NumRemoved += 1;
    }
    BandIt = BandIt.get_child(0);
    NodesToRemove += 1;
  }

  // Remove old order
  for (int i = 0; i < NodesToRemove; i += 1)
    Band = isl::manage(isl_schedule_node_delete(Band.release()));

  // Rebuild loop nest bottom-up according to new order.
  for (auto &NewBandId : reverse(NewOrder)) {
    auto OldBand = findBand(OldBands, NewBandId);
    auto OldMarker = LoopIdentification::createFromBand(OldBand);
    auto TheOldBand = ignoreMarkChild(OldBand);
    auto TheOldSchedule = isl::manage(
        isl_schedule_node_band_get_partial_schedule(TheOldBand.get()));

    Band = Band.insert_partial_schedule(TheOldSchedule);
    Band = Band.insert_mark(OldMarker.getIslId());
  }

  return Band;
}
```

Matrix-Multiplication

```
void matmul(int M, int N, int K,
            double C[const restrict static M][N],
            double A[const restrict static M][K],
            double B[const restrict static K][N]) {
  #pragma clang loop(j2) pack array(A)
  #pragma clang loop(i1) pack array(B)
  #pragma clang loop(i1,j1,k1,i2,j2) interchange \
                                     permutation(j1,k1,i1,j2,i2)
  #pragma clang loop(i,j,k) tile sizes(96,2048,256) \
                            pit_ids(i1,j1,k1) tile_ids(i2,j2,k2)
  #pragma clang loop id(i)
  for (int i = 0; i < M; i += 1)
    #pragma clang loop id(j)
    for (int j = 0; j < N; j += 1)
      #pragma clang loop id(k)
      for (int k = 0; k < K; k += 1)
        C[i][j] += A[i][k] * B[k][j];
}
```

Bonus

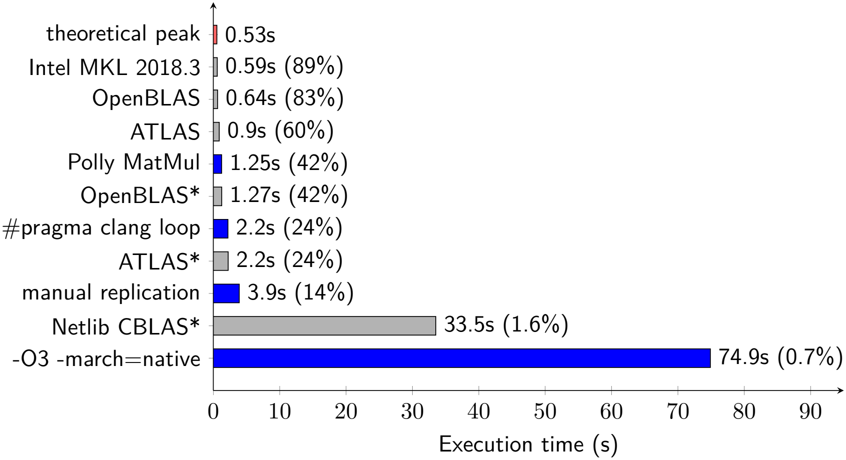
Matrix-Multiplication

After Transformation

```
double Packed_B[256][2048];
double Packed_A[96][256];
if (runtime check) {
  if (M >= 1)
    for (int c0 = 0; c0 <= floord(N - 1, 2048); c0 += 1)     // Loop j1
      for (int c1 = 0; c1 <= floord(K - 1, 256); c1 += 1) { // Loop k1

        // Copy-in: B -> Packed_B
        for (int c4 = 0; c4 <= min(2047, N - 2048 * c0 - 1); c4 += 1)
          for (int c5 = 0; c5 <= min(255, K - 256 * c1 - 1); c5 += 1)
            Packed_B[c4][c5] = B[256 * c1 + c5][2048 * c0 + c4];

        for (int c2 = 0; c2 <= floord(M - 1, 96); c2 += 1) { // Loop i1

          // Copy-in: A -> Packed_A
          for (int c6 = 0; c6 <= min(95, M - 96 * c2 - 1); c6 += 1)
            for (int c7 = 0; c7 <= min(255, K - 256 * c1 - 1); c7 += 1)
              Packed_A[c6][c7] = A[96 * c2 + c6][256 * c1 + c7];

          for (int c3 = 0; c3 <= min(2047, N - 2048 * c0 - 1); c3 += 1)   // Loop j2
            for (int c4 = 0; c4 <= min(95, M - 96 * c2 - 1); c4 += 1)      // Loop i2
              for (int c5 = 0; c5 <= min(255, K - 256 * c1 - 1); c5 += 1) // Loop k2
                C[96 * c2 + c4][2048 * c0 + c3] += Packed_A[c4][c5] * Packed_B[c3][c5];
        }
      }
} else {
  /* original code */
}
```

Matrix-Multiplication

Execution Speed



* Pre-compiled from Ubuntu repository