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
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

1 Why Loop Optimizations in the Compiler?

2 The Good

3 The Bad

4 The Ugly

5 The Solution (?)
Table of Contents

1 Why Loop Optimizations in the Compiler?

2 The Good

3 The Bad

4 The Ugly

5 The Solution (?)
Why Loop Optimizations in the Compiler?

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, …)
Why Loop Optimizations in the Compiler?

Partial Unrolling

```c
#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
Why Loop Optimizations in the Compiler?

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 nodepch

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
1. Why Loop Optimizations in the Compiler?

2. The Good
   - Available Loop Transformations
   - Available Pragmas
   - Available Infrastructure

3. The Bad

4. The Ugly

5. The Solution (?)
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
The Bad

Table of Contents

1 Why Loop Optimizations in the Compiler?

2 The Good

3 The Bad
   - Disabled Loop Passes
   - Pipeline Inflexibility
   - Loop Structure Preservation
   - Scalar Code Movement
   - Writing a Loop Pass is Hard

4 The Ugly

5 The Solution (?)
The Bad

Clang/LLVM/Polly Compiler Pipeline

```c
void f() {
    for (int i=...) {
        ...
    }
}
```

Clang

- Lexer
- Preprocessor
- Parser
- Semantic Analyzer
- IR Generation

LLVM

- Canonicalization passes
- Loop optimization passes
- Polly
- LoopVectorize
- Late Mid-End passes
- Target Backend
- Assembly

source.c
Unavailable Loop Passes

- 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

```c
#ifdef __builtin_unroll
    #pragma unroll 2
    for (int i = 0; i < 128; i+=1) {
        Stmt(i);
    }
#endif
```

```c
#ifdef __builtin_unroll
    #pragma unroll 2
    for (int i = 127; i >= 0; i-=1) {
        Stmt(i);
    }
#endif
```

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

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

- [Link to the review on LLVM](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
The Bad  →  Scalar Code Movement

Scalar/Loop Pass Interaction
Loop Nest Baking-In

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

LICM  (Register Promotion)
```

```c
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

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

GVN  (LoadPRE)
```

```c
for (int j=0; j<m; j+=1) {
    tmp = B[j];
    for (int i=0; i<n; i+=1)
        A[i] += i*tmp;
}
```
The Bad → Writing a Loop Pass is Hard

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, ...)

The Bad → Writing a Loop Pass is Hard
The Bad → Writing a Loop Pass is Hard

ISL Schedule Tree Transformation
Loop Distribution

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

```c
for (int i = 0; i < n; i+=1) {
    StmtA(i);
    StmtB(i);
}
```
The Bad → Writing a Loop Pass is Hard

Polly Code for Loop Distribution
Transformation-Specific Code

```c
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](https://reviews.llvm.org/D52116))
- No equivalent for LoopIDs
- Difference between PHI and select irrelevant for high-level purposes
Table of Contents

1 Why Loop Optimizations in the Compiler?

2 The Good

3 The Bad

4 The Ugly
   ■ Independent Loop Pass Profitability
   ■ Code Version Explosion

5 The Solution (?)
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
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
for (int i = 0; i < n; i+=1)
    for (int j = 0; j < m; j+=1)
        Stmt(i,j);
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);
}
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);
    }
}
The Ugly → Code Version Explosion

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 {
        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);}}}
    }
}
```
The Solution (?)

Table of Contents

1 Why Loop Optimizations in the Compiler?

2 The Good

3 The Bad

4 The Ugly

5 The Solution (?)
   - Integrated Loop Pass
   - Combined Profitability Heuristic
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
The Solution → Combined Profitability Heuristic

**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;
}
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)
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);
    }
}
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://github.com/dotnet/roslyn/blob/master/src/Compilers/Core/Portable/Syntax/GreenNode.cs
Red-Green Tree
The Green DAG
The Solution (?) → Combined Profitability Heuristic

Red-Green Tree

The Red Tree

Root
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
Closed-Form Expressions

ScalarEvolution \(-01\)

PredicatedScalarEvolution \(-02\)

PolyhedralValueAnalysis \(-03\)
Access Analysis

One-dimensional

One-dimensional, allow additional assumptions

Multi-dimensional, allow additional assumptions
The Solution (?) → Combined Profitability Heuristic

**Dependency Analysis**

- Control-flow insensitive
- SCEV-based
- Polyhedral
- Approximative LP solver
- Exact LP solver

-01
-02
-03 / -027
 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:
    \[ \rightarrow \text{Loop} \rightarrow \text{Sequential} \rightarrow \text{Loop} \rightarrow \text{Sequential} \rightarrow \ldots \]
  - 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)
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);
}
The Solution → Combined Profitability Heuristic

Non-Cyclic Control Flow
Sequential, but Conditional

```c
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);
}
```

Control dependency

- Necessary condition: 1
- Sufficient condition: a
- Necessary condition: b
- Sufficient condition: a && b

42 / 45
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

}
```
The Solution → Combined Profitability Heuristic

Non-Cyclic Control Flow

Loop Distribution

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

for (int i = 0; i < n; i += 1) {
    StmtD(i);
    if (condition) {
        StmtC(i);
        StmtB(i);
    }
}
```

Necessary condition: 1
Sufficient condition: a

Necessary condition: b
Sufficient condition: a && b
Code Generation

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

```c
for (int i=0; i<128; i++)
    for (int j=0; j<64; j++)
        for (int k=0; k<256; k++)
            A[i][j] = ...
            B[i][k] = ...
            j*sin(…)
            k*cos(…)
            2*PI*i/128
```
The Solution (?) → Combined Profitability Heuristic

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
Conclusion

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!
##Bonus

###LLVM Loop Passes

Excluding Normalization Passes

<table>
<thead>
<tr>
<th>LLVM Pass</th>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Simple-)LoopUnswitch</td>
<td>none</td>
</tr>
<tr>
<td>LoopIdiom</td>
<td>none</td>
</tr>
<tr>
<td>LoopDeletion</td>
<td>none</td>
</tr>
<tr>
<td>LoopInterchange*</td>
<td>none</td>
</tr>
<tr>
<td>SimpleLoopUnroll</td>
<td>llvm.loop.unroll.*</td>
</tr>
<tr>
<td>LoopReroll*</td>
<td>none</td>
</tr>
<tr>
<td>LoopVersioningLICM++</td>
<td>llvm.loop.licm_versioning.disable</td>
</tr>
<tr>
<td>LoopDistribute+</td>
<td>llvm.loop.distribute.enable</td>
</tr>
<tr>
<td>LoopVectorize+</td>
<td>llvm.loop.vectorize.*</td>
</tr>
<tr>
<td></td>
<td>llvm.loop.interleave.count</td>
</tr>
<tr>
<td></td>
<td>llvm.loop.isvectorized</td>
</tr>
<tr>
<td>LoopLoadElimination+</td>
<td>none</td>
</tr>
<tr>
<td>LoopUnrollAndJam*</td>
<td>llvm.loop.unroll_and_jam.*</td>
</tr>
<tr>
<td>LoopUnroll</td>
<td>llvm.loop.unroll.*</td>
</tr>
</tbody>
</table>
The Polyhedral Model

for (int i=1; i<5; i++)
    for (int j=1; i+j<6; j++)
        S(i,j);
The Polyhedral Model

\[
\{ S(i,j) \mid 0 < i,j \land i + j < 6 \} \quad S(1,1), S(1,2), S(1,3), S(1,4), S(2,1), S(2,2), S(2,3), S(3,1), S(3,2), S(4,1)
\]

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 \land 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 \((\text{int } j=1; j<5; j++)\)

\[
\text{for (int } i=1; i+j<6; i++)
\]

\[ S(i, j); \]
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)

\[ S(i,j) \mapsto (i, j/2, j \mod 2) \]

\[
\begin{align*}
\text{for (int } i=1; i<5; i++) \\
&\text{ for (int } t=1; i+t<6; t+=2) \\
&\quad \text{ for (int } j=t; j<t+2 && i+j<6; j++) \\
&\quad \quad \text{ S}(i,j);
\end{align*}
\]
The Polyhedral Model

Tiling

\[ S(i,j) \mapsto (i/2, j/2, i \mod 2, j \mod 2) \]

\[
\begin{align*}
\text{for (int } s=1; s<5; s+=2) \\
\text{for (int } t=1; s+t<6; t+=2) \\
\text{for (int } i=s; i<s+2 && i<5; i++) \\
\text{for (int } j=t; j<t+2 && i+j<6; j++) \\
S(i,j);
\end{align*}
\]
$S(i,j) \mapsto (i/2,j, i \mod 2)$

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

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

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);
}
The Polyhedral Model

Loop Distribution

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

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

\[ S(5-i,6-j); \]
Bonus

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. mayThrow 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
Bonus

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;
}
```
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;
}
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];
}

Matrix-Multiplication
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

<table>
<thead>
<tr>
<th>Method</th>
<th>Execution Time (s)</th>
<th>Speed Relative to Theoretical Peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>theoretical peak</td>
<td>0.53s</td>
<td>100%</td>
</tr>
<tr>
<td>Intel MKL 2018.3</td>
<td>0.59s (89%)</td>
<td></td>
</tr>
<tr>
<td>OpenBLAS</td>
<td>0.64s (83%)</td>
<td></td>
</tr>
<tr>
<td>ATLAS</td>
<td>0.9s (60%)</td>
<td></td>
</tr>
<tr>
<td>Polly MatMul</td>
<td>1.25s (42%)</td>
<td></td>
</tr>
<tr>
<td>OpenBLAS*</td>
<td>1.27s (42%)</td>
<td></td>
</tr>
<tr>
<td>#pragma clang loop</td>
<td>2.2s (24%)</td>
<td></td>
</tr>
<tr>
<td>ATLAS*</td>
<td>2.2s (24%)</td>
<td></td>
</tr>
<tr>
<td>manual replication</td>
<td>3.9s (14%)</td>
<td></td>
</tr>
<tr>
<td>Netlib CBLAS*</td>
<td>33.5s (1.6%)</td>
<td></td>
</tr>
<tr>
<td>-O3 -march=native</td>
<td>74.9s (0.7%)</td>
<td></td>
</tr>
</tbody>
</table>

*Pre-compiled from Ubuntu repository*