Generation of Fast and Parallel Code in LLVM

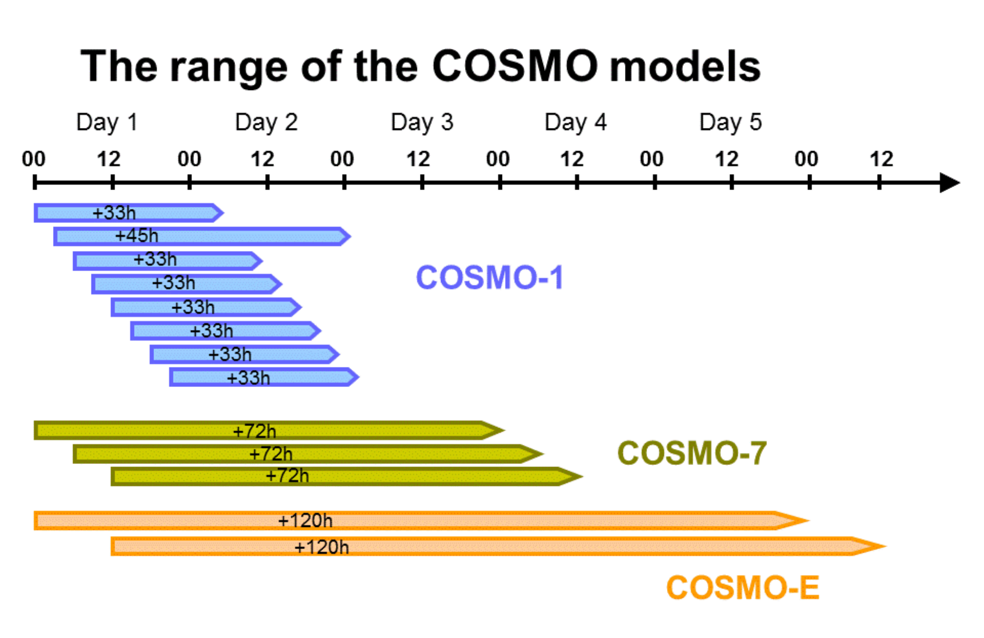
Tobias Grosser

LLVM and Clang Summer School
Paris, June 2017

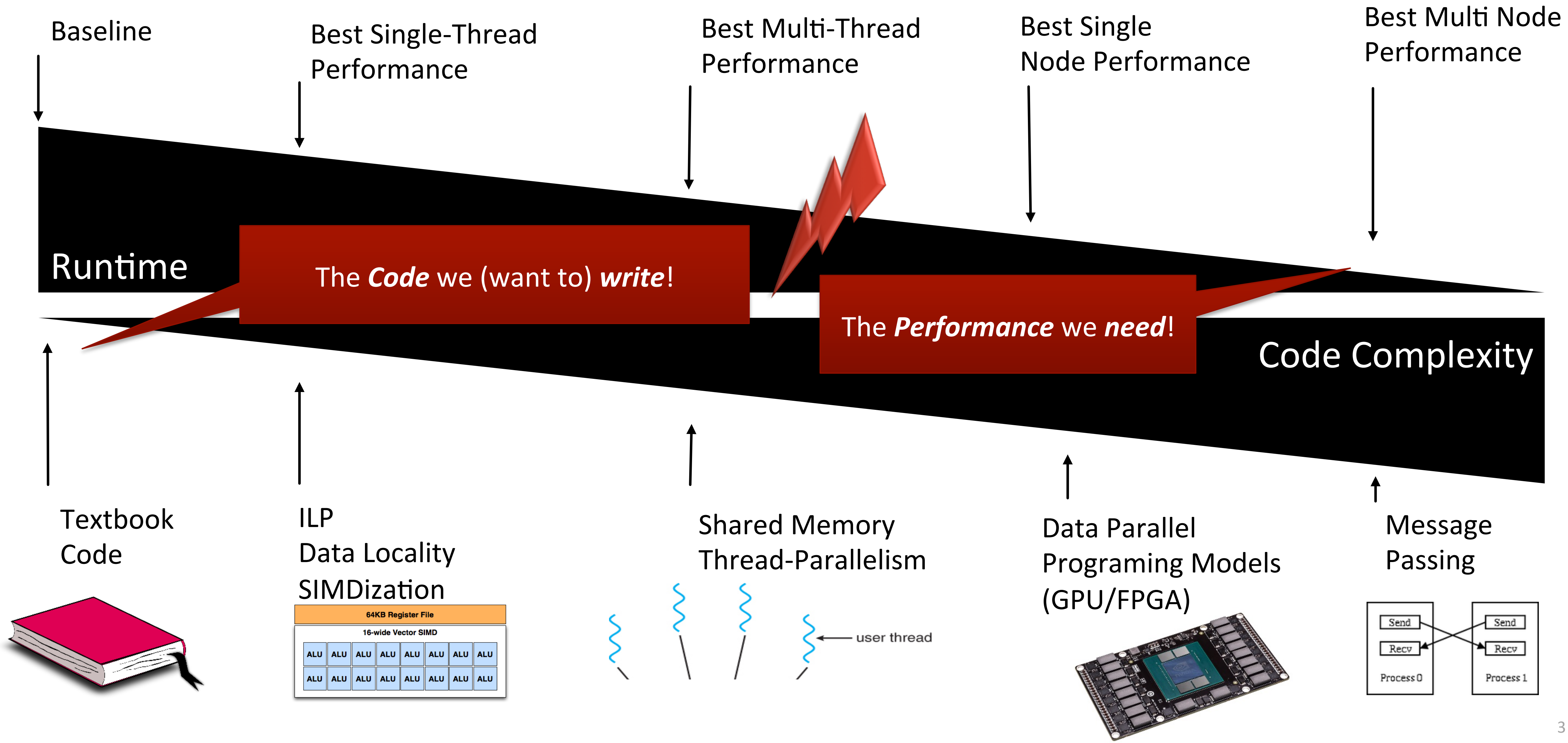
COSMO: Weather Prediction in Switzerland

Running Large Programs in Parallel is Challenging

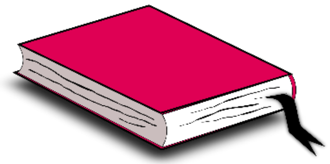
- > 500,000 Lines Code
- > 15,000 Loops
- 12 nodes + 192 GPUs @CSCS Lugano



Performance vs. Code Complexity



GEMM: Generalized Matrix Multiplication

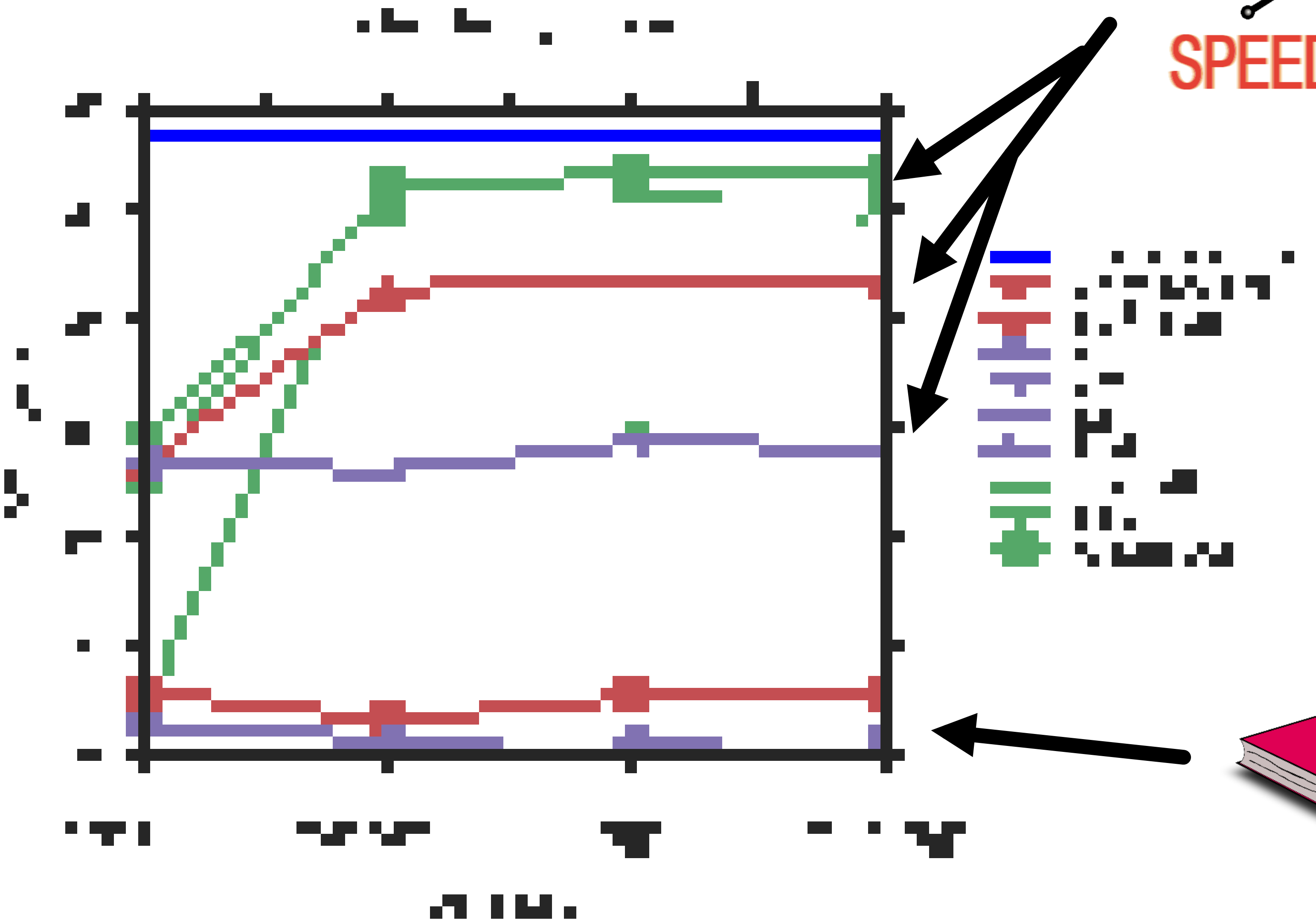


The Simple Textbook Version

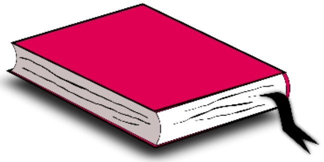
$$C = A \times B$$

```
void gemm(int N, int M, int K,
          double A[N][K], double B[K][M], double C[N][M]) {

  for (i = 0; i < N; i++)
    for (j = 0; j < M; j++)
      for (k = 0; k < K; k++)
        C[i][j] += A[i][k] * B[k][j];
}
```

GEMM: Performance Single Threaded



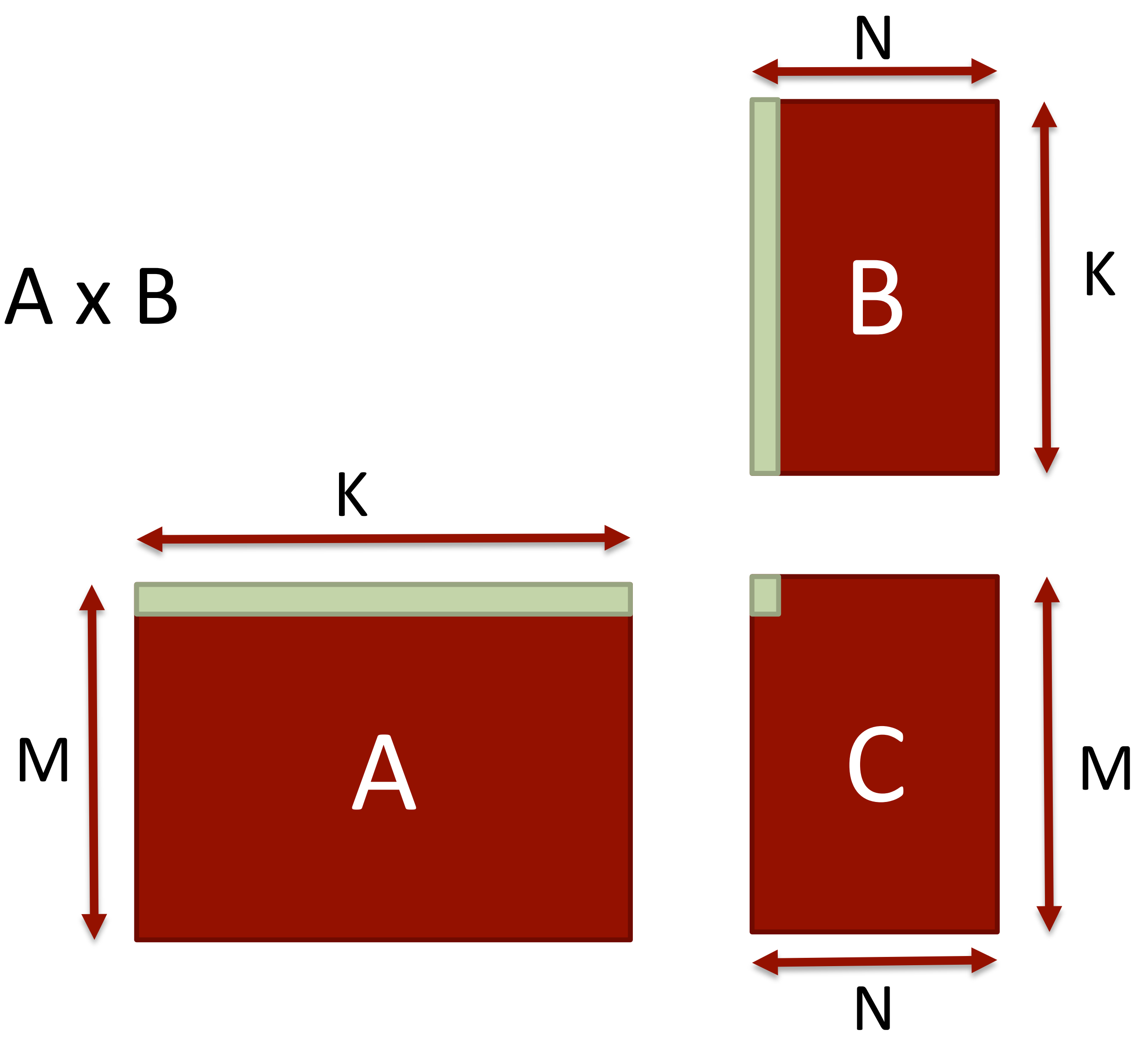
Optimized Codes



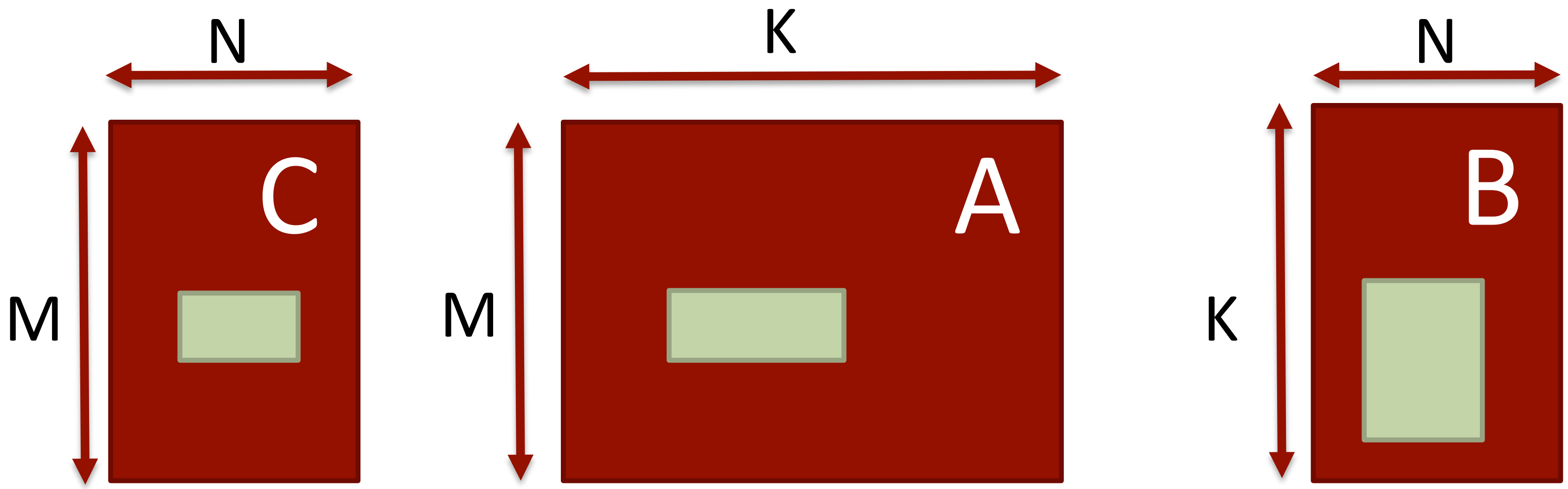
The Simple Textbook Version

GEMM: Computing on Micro Panels

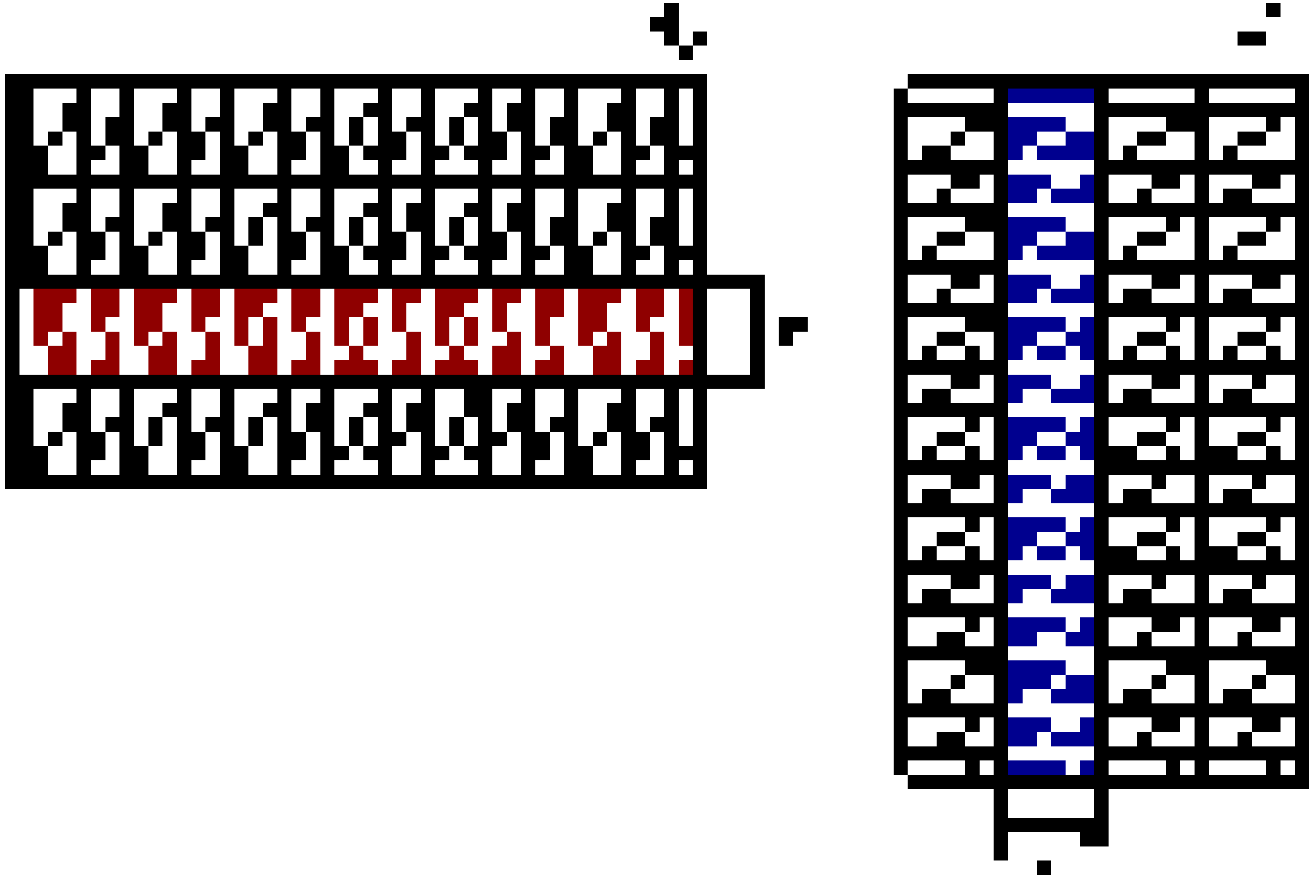
$C = A \times B$



GEMM: Computing on Micro Panels



GEMM: Repack Micro Panels



BLIS: A Framework for Rapidly Instantiating BLAS Functionality
FIELD G. VAN ZEE and ROBERT A. VAN DE GEIJN

GEMM: The BLIS Kernel Structure

Data Layout Transformation

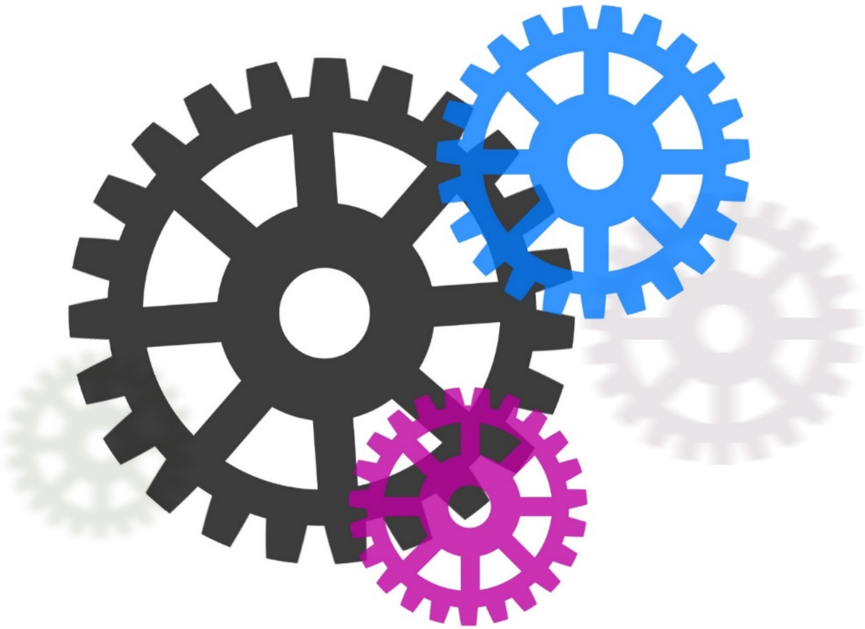
```
L1: for jc = 0,...,n−1 in steps of nc
L2:   for pc = 0,...,k−1 in steps of kc
            B(pc : pc + kc −1,jc : jc + nc −1) → Bc // Pack into Bc
L3:     for ic = 0,...,m−1 in steps of mc
              A(ic : ic + mc −1,pc : pc + kc −1) → Ac // Pack into Ac
L4:       for jr = 0,...,nc −1 in steps of nr // Macro-kernel
L5:         for ir = 0,...,mc −1 in steps of mr
L6:           for pr = 0,...,kc −1 in steps of 1 // Micro-kernel
                    Cc(ir : ir + mr −1,jr : jr + nr −1) +=
                      Ac(ir : ir + mr −1,pr) · Bc(pr,jr : jr + nr −1)
```

Loop Blocking

SIMD Instructions

BLIS: A Framework for Rapidly Instantiating BLAS Functionality

FIELD G. VAN ZEE and ROBERT A. VAN DE GEIJN



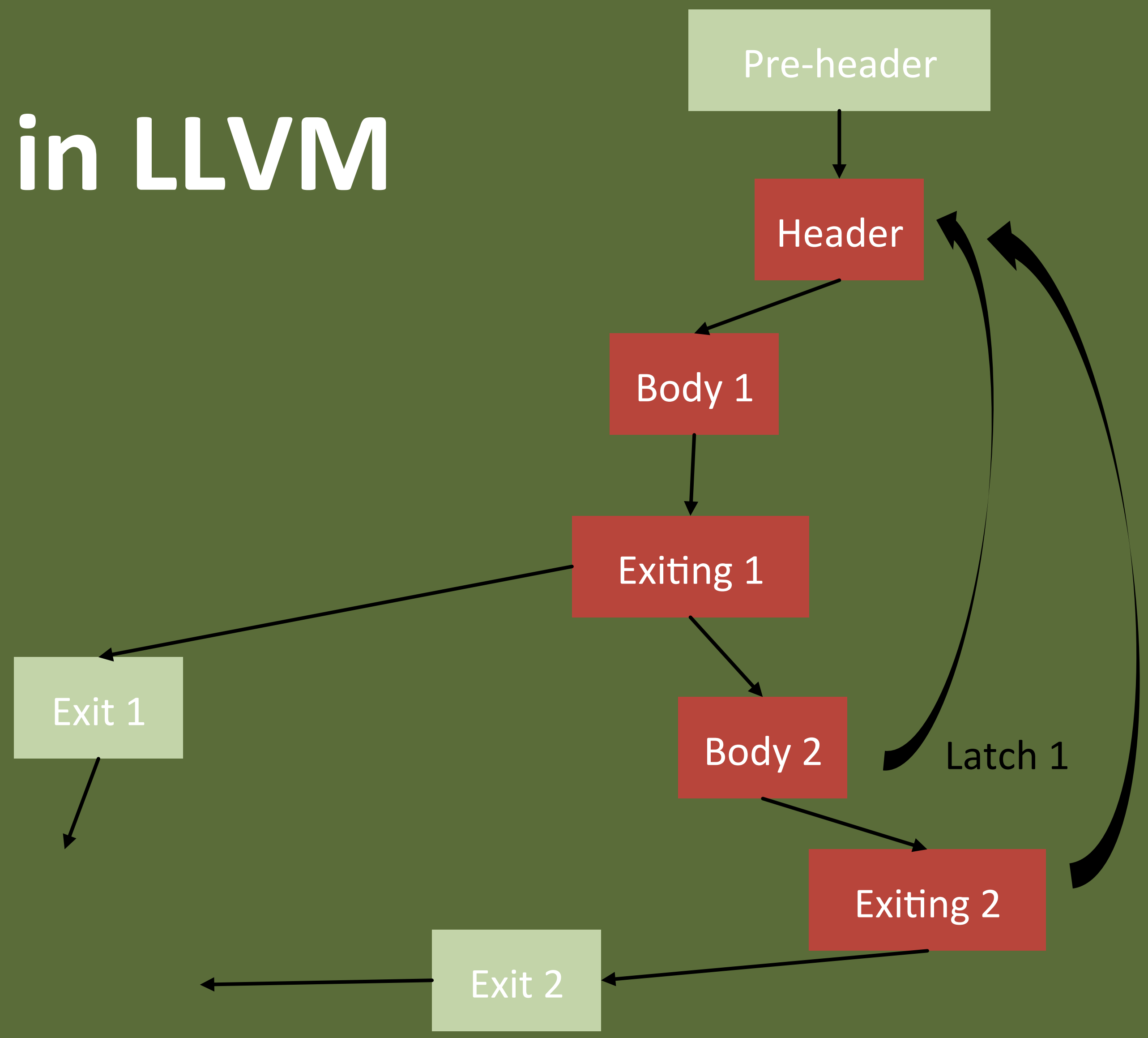
Parallel Code Generation

Which facilities does LLVM provide?

What we learn today (and tomorrow):

- LLVM Analysis Passes
- Automatic SIMDization
- Modeling of Computational Loops with Presburger Sets
- Detection of Parallel Loops

Analysis Passes in LLVM



LLVM IR: Modeling high-level knowledge in LLVM-IR

Metadata

- Information **cannot be derived** from IR directly

+ No need to recompute
- Must be kept consistent

Analysis

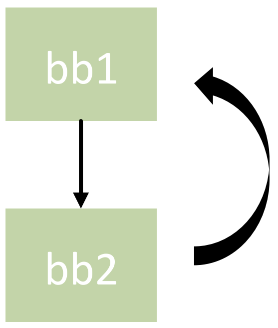
- Information **can be derived** from IR directly

+ Must be recomputed
- Never outdated

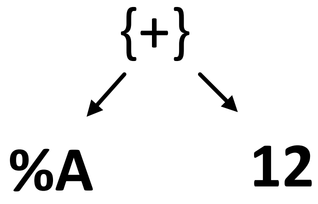
Preferred!

Analysis Passes in LLVM

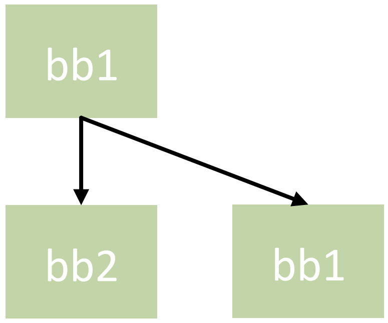
Loops



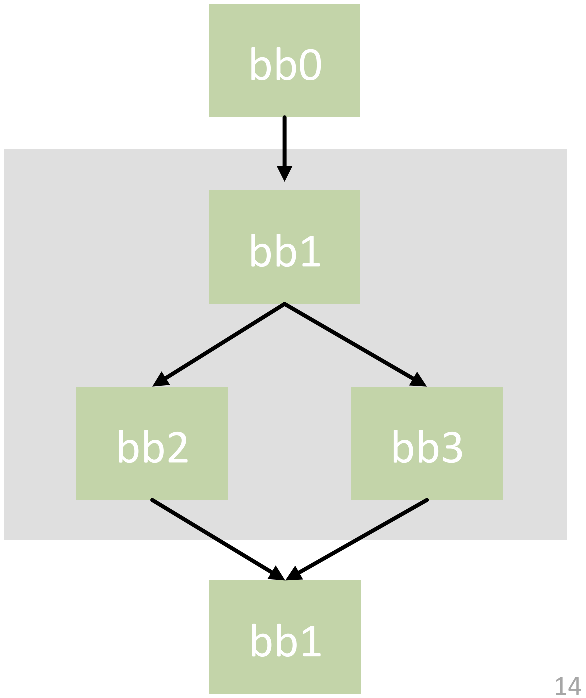
Scalar Evolution



(Post) Dominance

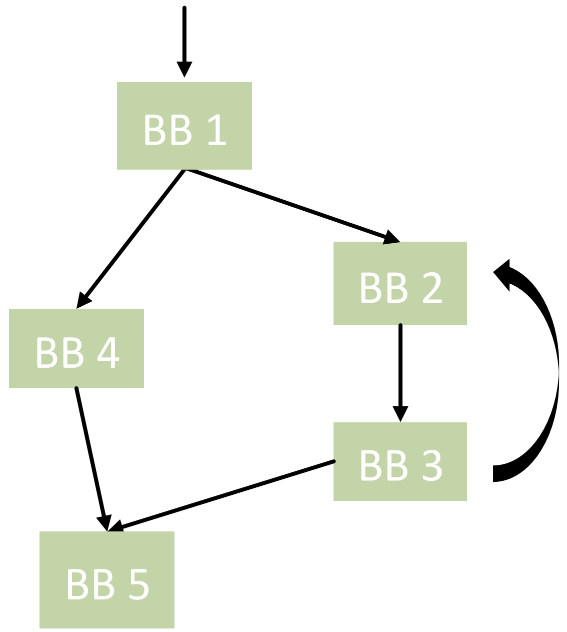


Regions

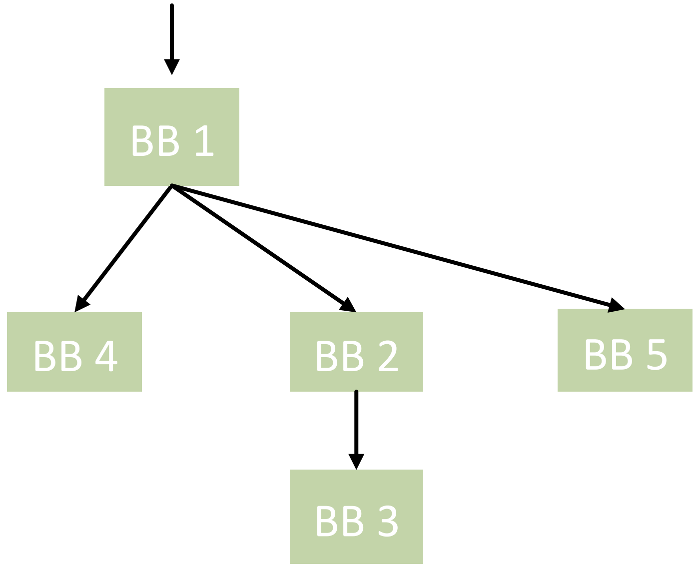


Dominance

CFG



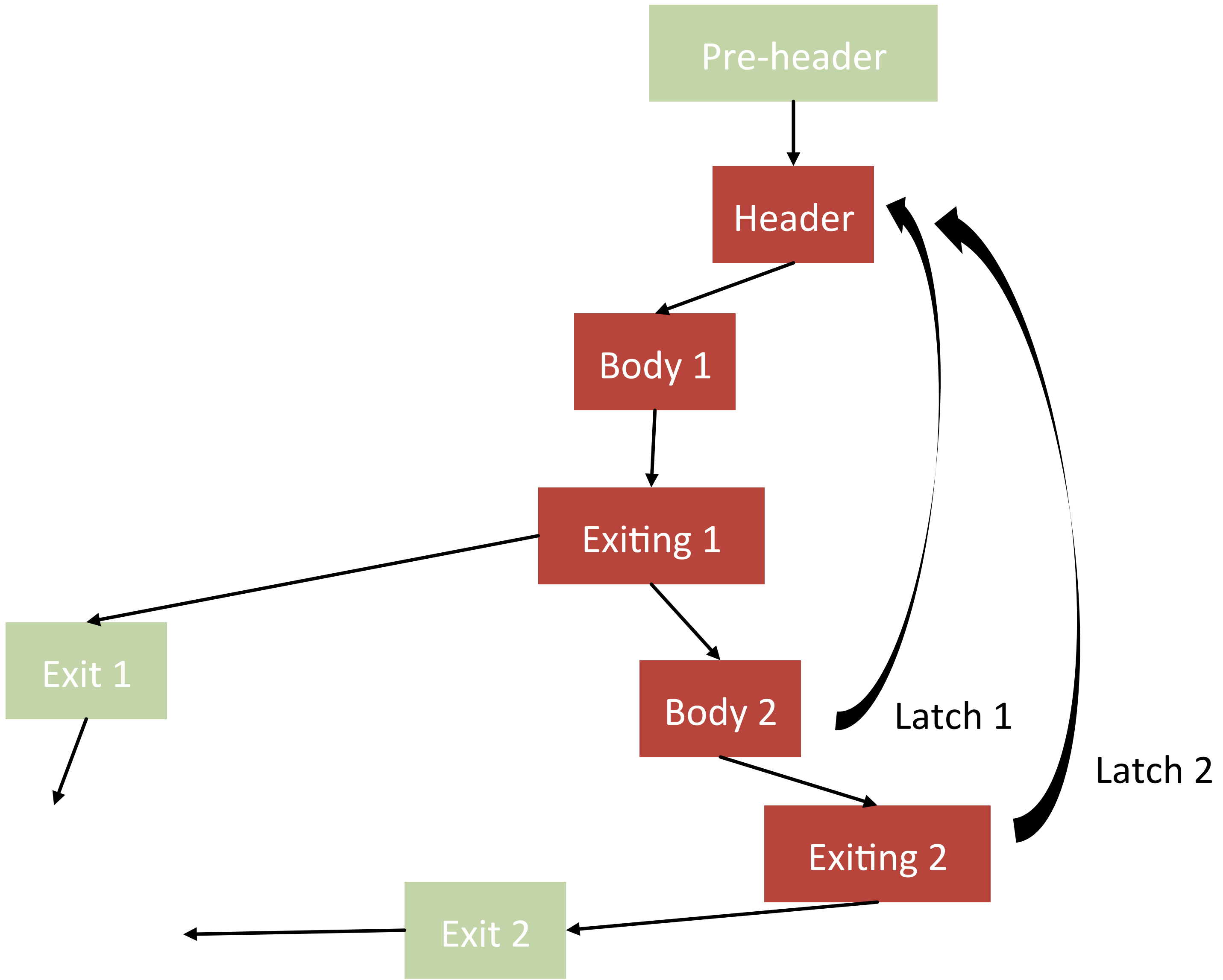
Dominator Tree



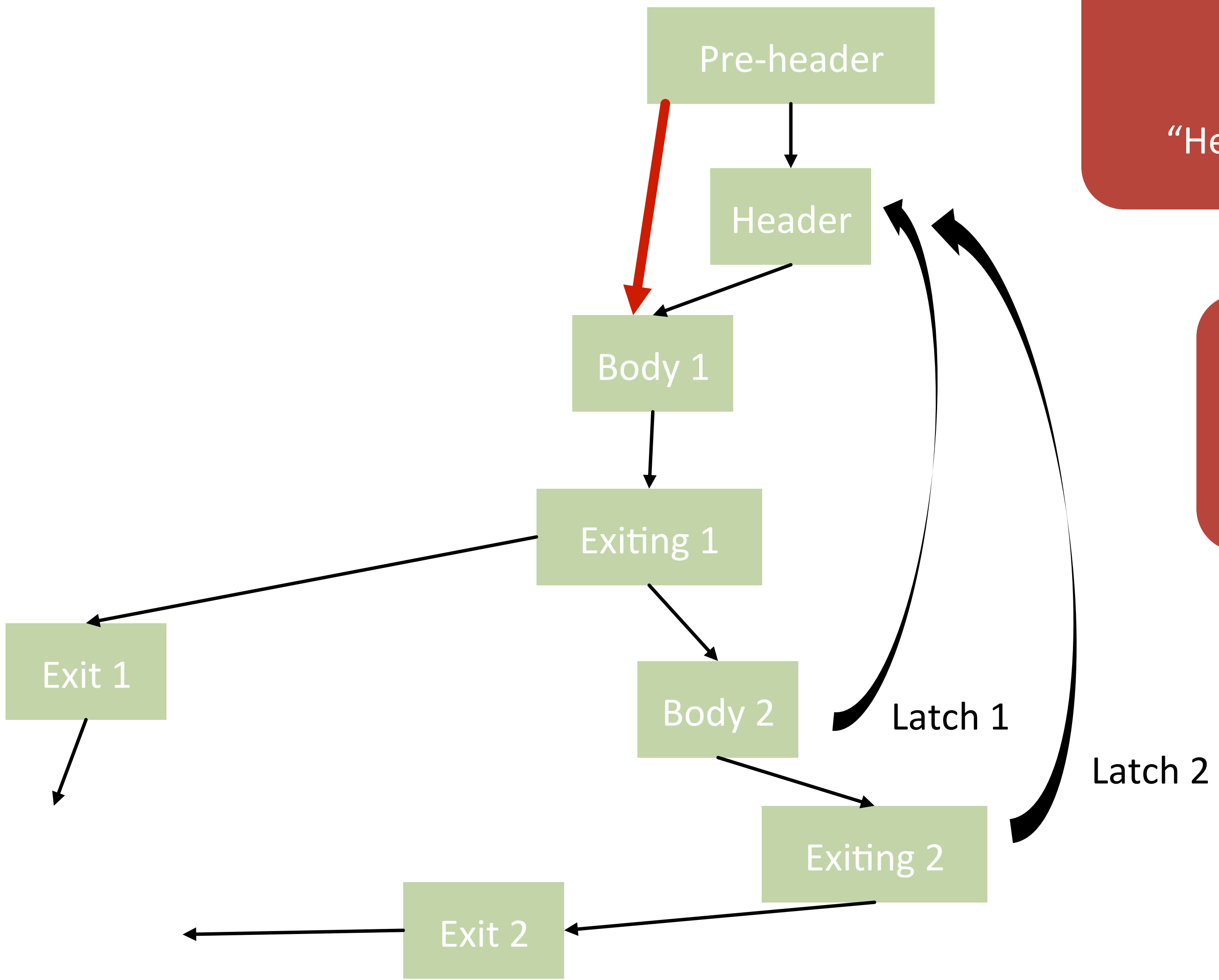
A ***dominates*** B, if each path from the entry to B contains A.

A ***post-dominates*** B if each path from B to the exit contains A.

Loop Info: Detect Natural Loops



Loop Info: Detect Natural Loops

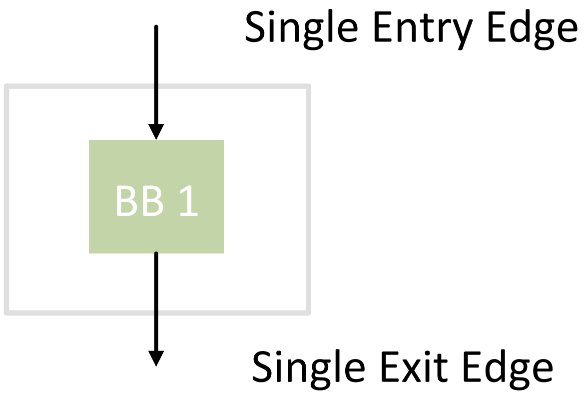


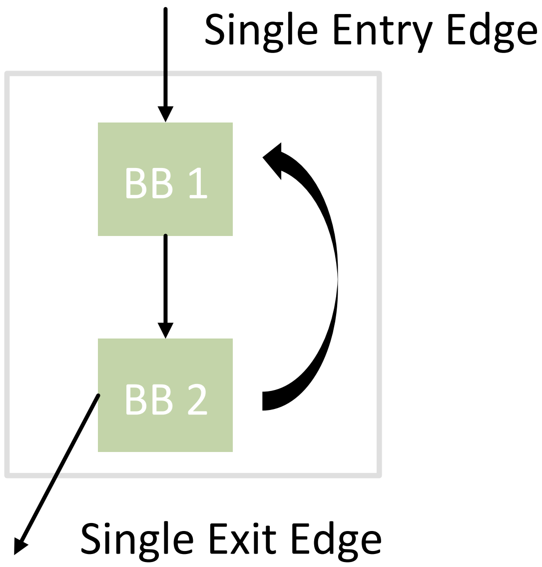
No Natural Loop!

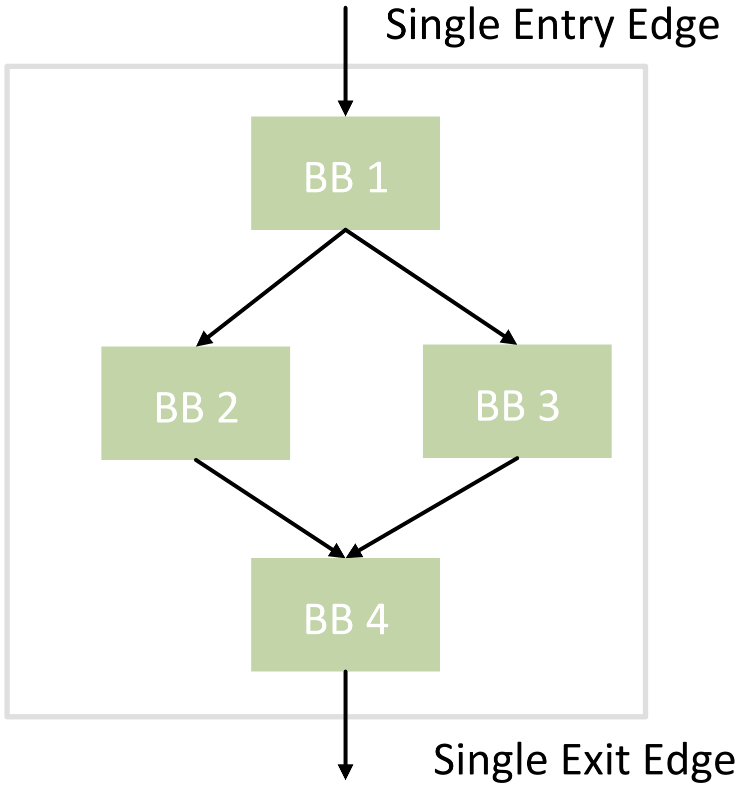
"Header" does not dominate latches.

LLVM ***does not*** model this loop!

Region Info: Single Entry Single Exit Regions

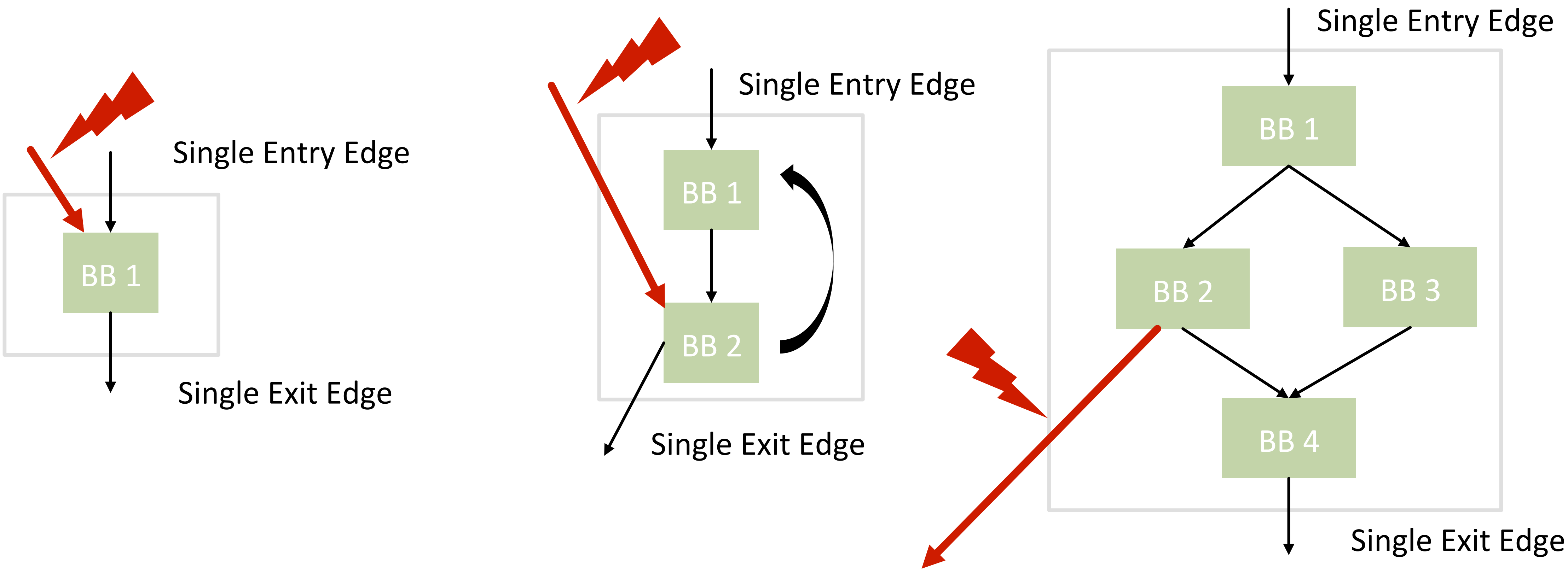






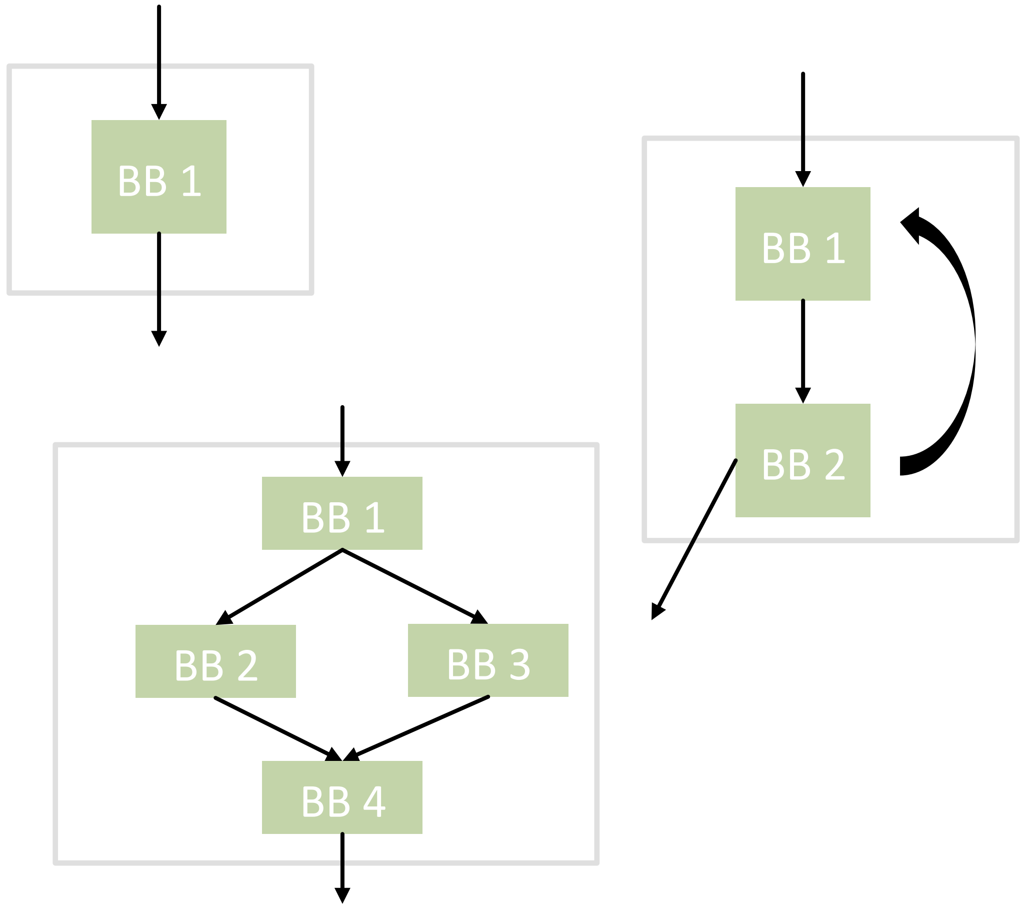
A ***simple region*** is a subgraph of the CFG with a single entry and a single exit edge.

Region Info: No Regions

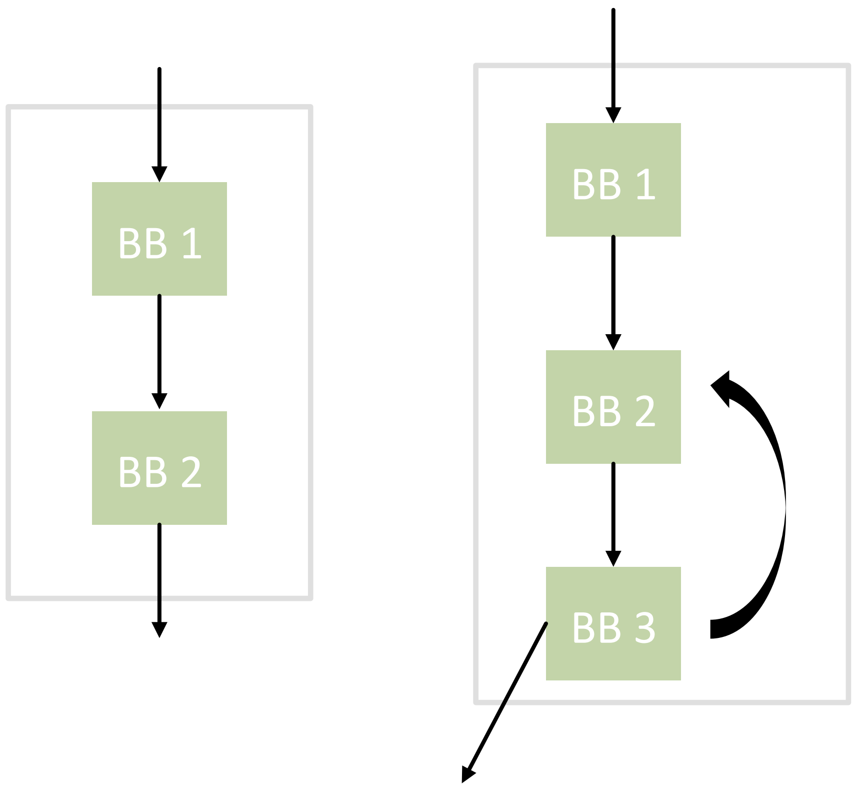


Region Info: Single Entry Single Exit Regions

A region is ***canonical*** if it cannot be split into a sequence of smaller regions.

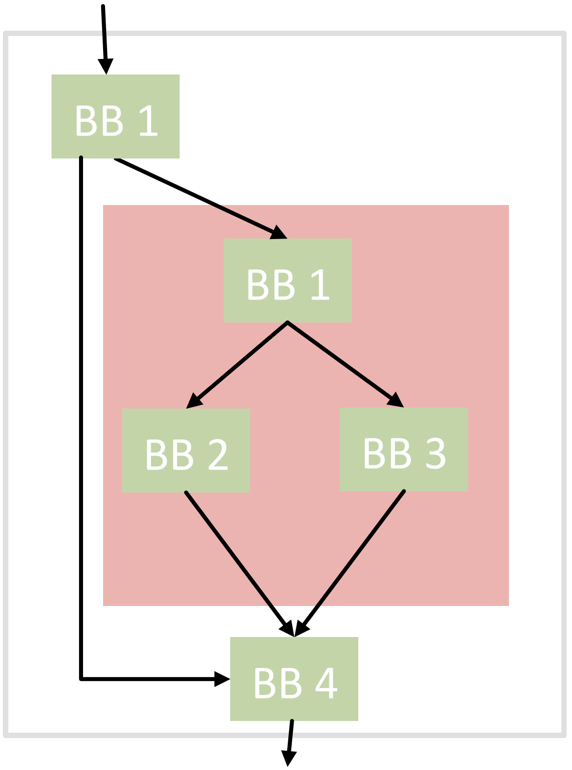


Canonical Regions

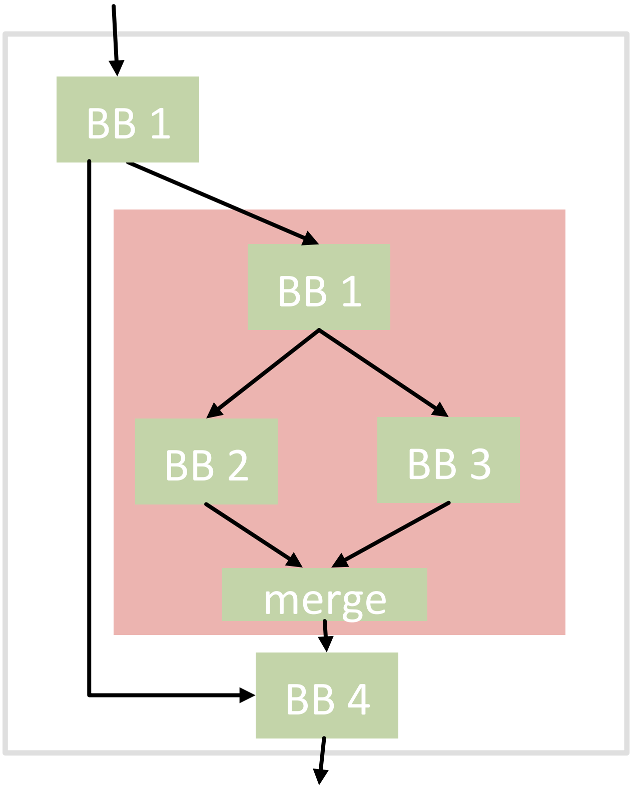


Non-Canonical

Region Info: Refined Regions



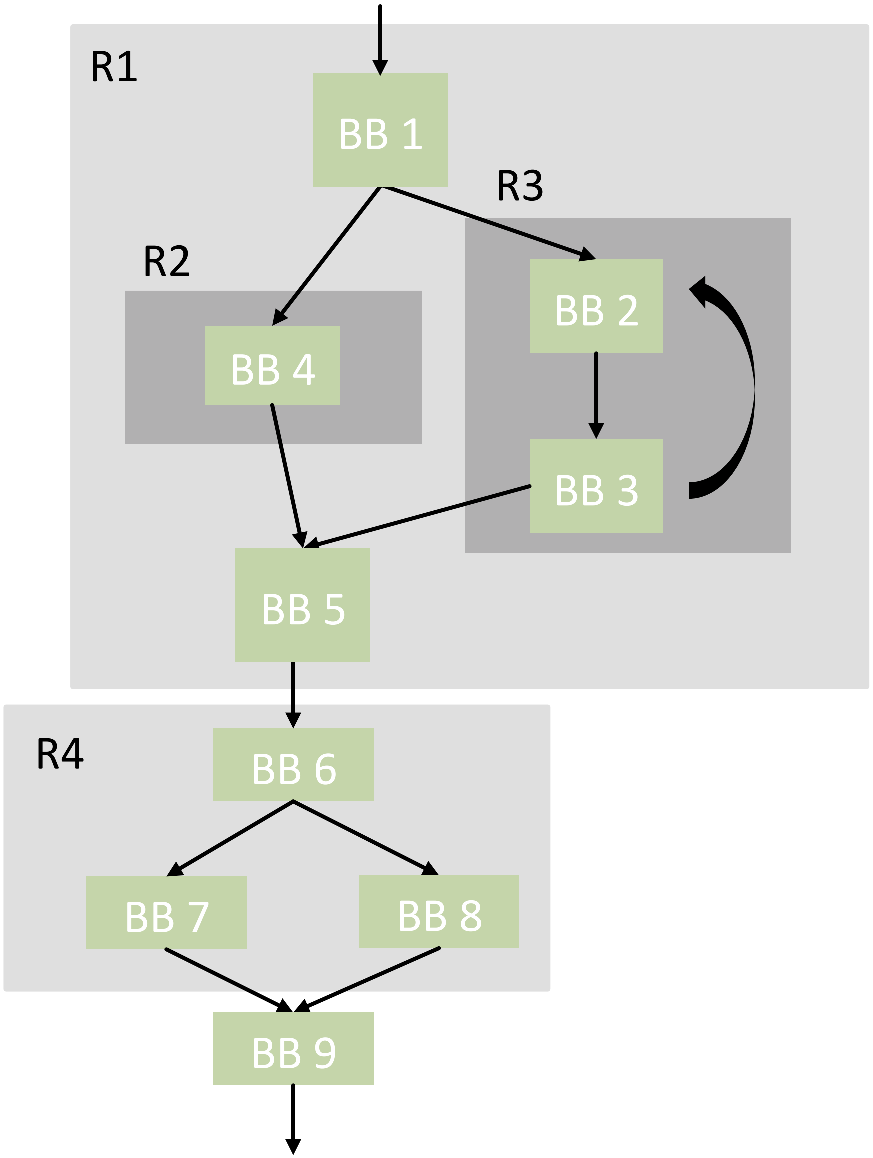
Refined Region

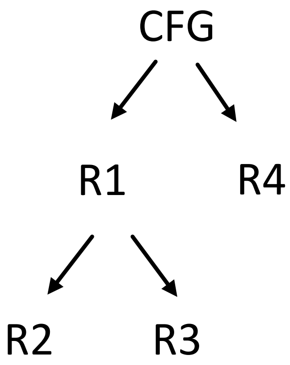


Simplified Region

A ***refined region*** can be transformed into a ***simple region*** by interesting a single merge block.

The (Refine) Region Tree





(Refined) regions form a tree.
This tree is unique!

Scalar Evolution

```
define void @foo(i64 %a, i64 %b, i64 %c) {
  %t0 = add i64 %b, %a
  %t1 = add i64 %t0, 7
  %t2 = add i64 %t1,
  %c ret i64 %t2
}
```

Provides closed form expressions for scalar variables!

SCEV: (7 + %a + %b +%c)

History: Scalar Evolution

- **Bachmann 1994:** "Chains of recurrences - A method to expedit the evaluation of closed-form functions"
- **Engelen 2000: "**Chains of recurrences for loop optimization"
- **Pop 2003:** "Analysis of induction variables using chains of recurrences"

- Introduced in Compilers:
  - **GCC:** 20 June, 2004 by Sebastian Pop
  - **LLVM:** 2 April, 2004 by Chris Lattner

ScalarEvolution: Components

- Arithmetic Operations
  - Addition (SCEVAdd)
  - Multiplication (SCEVMul)
  - Signed Division (SCEVSDiv)
  - SignExension (SCEVSExt)
  - ZeroExtension (SCEVZExt)
  - Truncation (SCEVTrunc)
  - Signed Maximum (SCEVSMax)
  - Unsigned Maximum (SCEVUMax)

Many heuristics to recover these common pattern

- Special Values
  - Reference to LLVM Value (SCEVUnknown)
  - Integer Constant (SCEVConstant)
    - *Symbolic Type Size*
    - *Symbolic Alignment*
    - *Symbolic Field Offset*
  - Add Recurrences (SCEVAddRec)

Two Dimensional Array – No Loops

```
double *bar(double a[10][10], long b, long c) {
  return &a[b * 3 + 7][c + 5];
}
define double* @bar([10 x double]* %a, i64 %b, i64 %c) {
  %bx3 = mul i64 %b, 3
  %bx3a7 = add i64 %bx3, 7
  %ca5 = add i64 %c, 5
  %z = getelementptr [10 x double]* %a, i64 %bx3a7,
                                        i64 %ca5
  ret double* %z
}
```

SCEV (no TargetData): `(((75 + %c + (30 * %b)) * sizeof(double)) + %a)`

SCEV (with TargetData): `(600 + (8 * %c) + (240 * %b) + %a)`

Add-Recurrences

Template of an Add Recurrence: {base, +,stride}_<loop>

Value:
base + <virtual_iv> * stride

```
void foo(long n, double *p) {
  for (long i = 0; i < n; ++i)
    double *ptr = &p[i];
}
```

%for.body: reference to header of loop in which expression evolves!

SCEV (no TargetData): %ptr = {%p, +, sizeof(double)}_<%for.body>

SCEV (with TargetData): %ptr = {%p, +, 8}_<%for.body>

Using Scalar Evolution

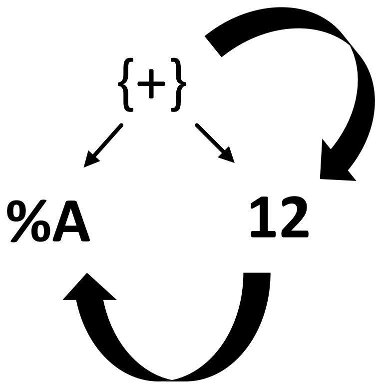
```
void YourPass::getAnalysisUsage(AnalysisUsage &AU) const {
  AU.setPreservesAll(); AU.addRequired();
}

bool YourPass::runOnFunction(Function &F) {
  ScalarEvolution &SE = getAnalysis();

  // Get SVEV for the first instruction of the function.
  Instruction *FirstInstruction = (*F.begin())->begin();
  const SCEV *evolution = SE->getSCEV(FirstInstruction);

  if (isa<SCEVConstant>(evolution))
    errs() << "The first instruction is a constant SCEV";
}
```

Analyzing and Modifying Scalar Evolutions

1. Analyse
   - ScalarEvolution

LLVM-IR → {%A, +, 12}_<L1>

2. Transform
   - SCEVVisitor
   - SCEVTraversal



3. Code Generation
   - SCEVExpander

{%A, +, 12}_<L1> → **LLVM-IR**

Scalar Evolution: nsw / nuw

- Scalar Evolution allows integer wrapping

  a + b < a is possible

- Flags: no-signed-wrap (nsw) and no-unsigned-wrap (nuw)

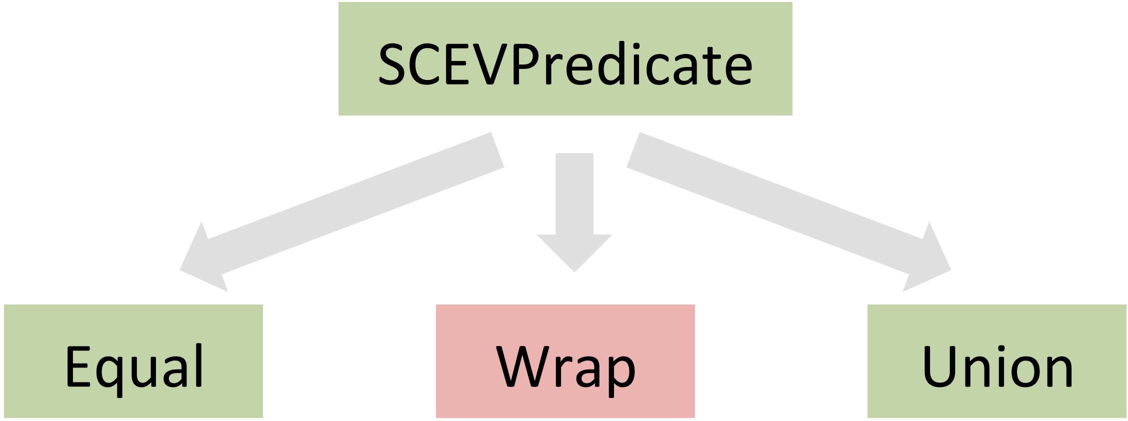
  If present, one can assume no (un)signed wrapping to happen

- Information is derived from LLVM-IR nsw, nuw flags

Predicated Scalar Evolution

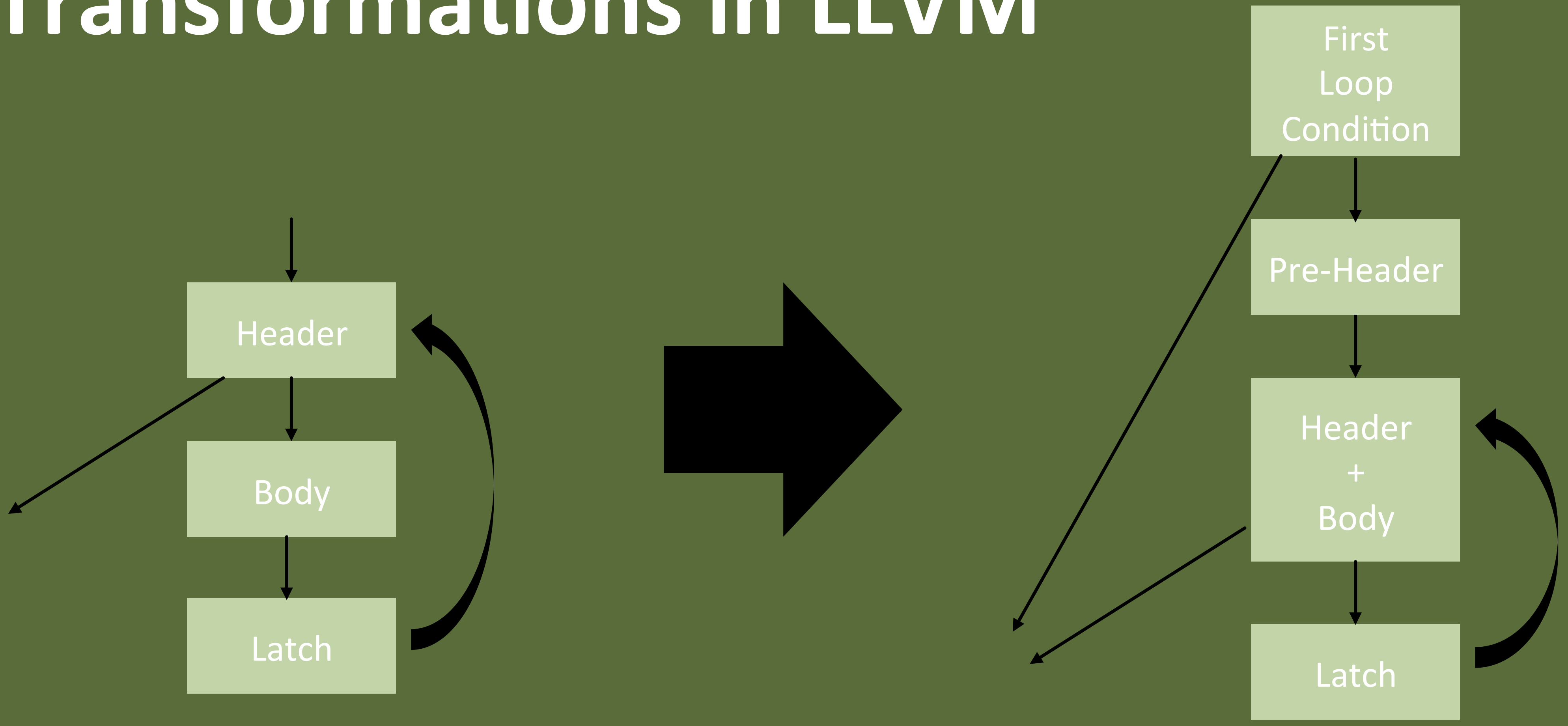
```
for (unsigned i = p; i <= n + m; i++)
  …
```

```
const SCEV *getPredicatedBackedgeTakenCount(
                          const Loop *L,
                          SCEVUnionPredicate &Predicates);
```



Count: n + m – p
Predicate: Assuming n + m – p does not wrap

Loop Transformations in LLVM



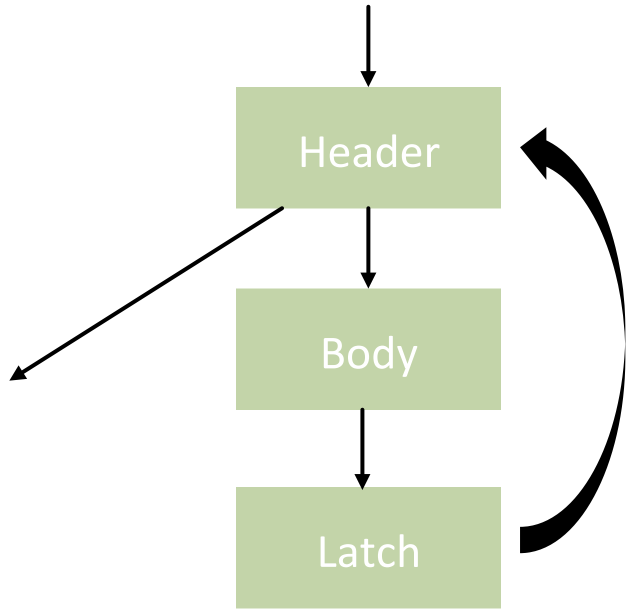
Loop Optimizations in LLVM (ignoring Polly)

-loop-deletion	Deletion of dead loops
-loop-distribute	Split loops (e.g., to expose SIMDization opportunities)
-loop-idiom	Recognize loop idioms (e.g., memcpy)
-loop-interchange	Improve data-locality by interchanging loops
-loop-reduce	Loop Strength reduction
-loop-reroll	Reroll loops
-loop-rotate	**Rotate loops**
-loop-simplify	**Canonicalize natural loops (e.g., insert preheader)**
-loop-unroll	Unroll loops (also done by the vectorizer)
-loop-unswitch	Unswitch loops
-indvars	**Induction Variable Simplification**

Uses Hal's BB Vectorizer

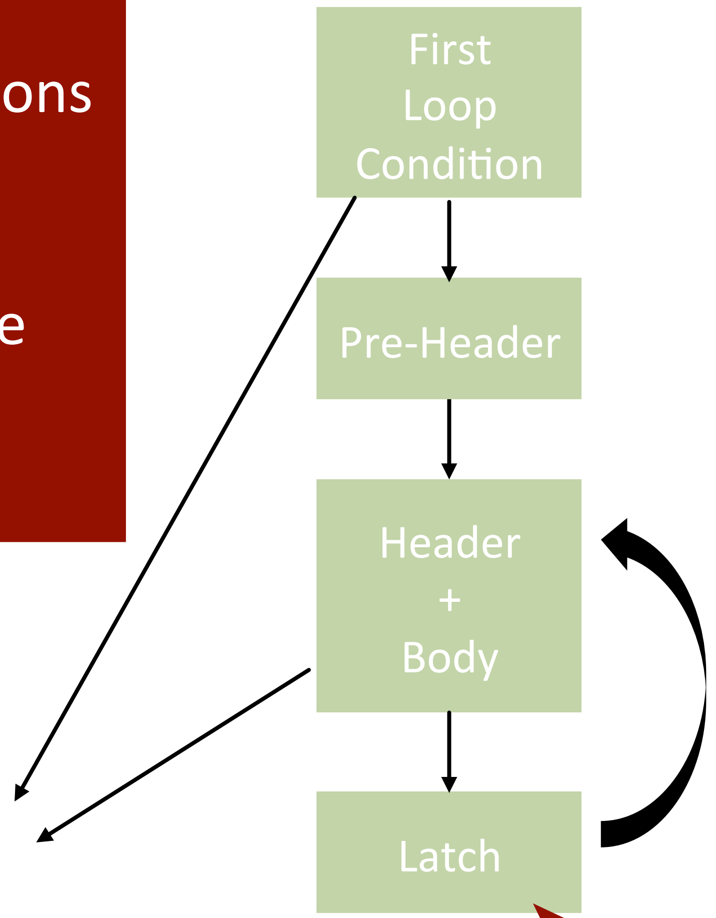
Very Conservative

Loop Rotation



Benefits:

- Invariant instructions can be hoisted to pre-header
- All instructions are executed equally often.

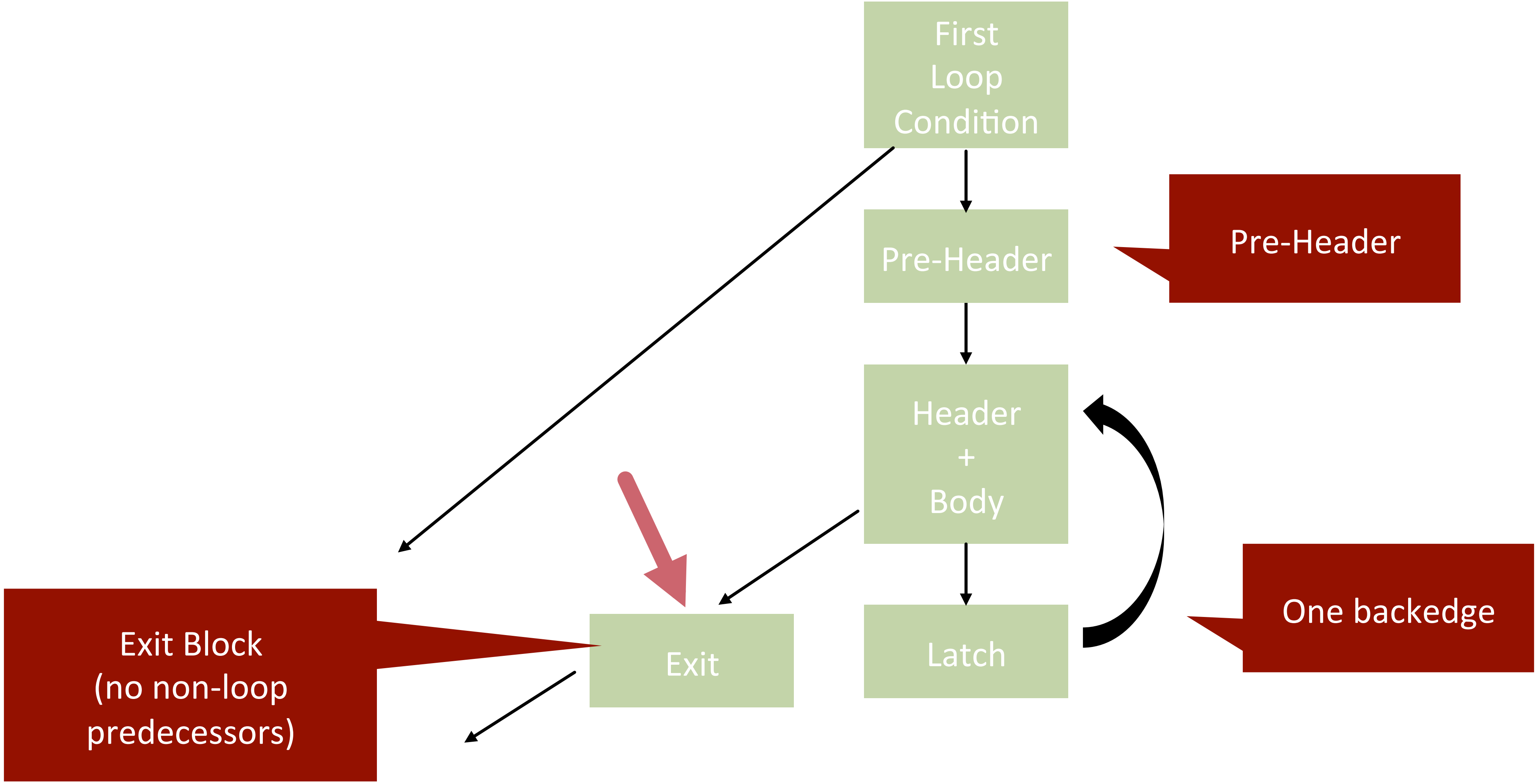


Standard For-Loop

Rotated Loop

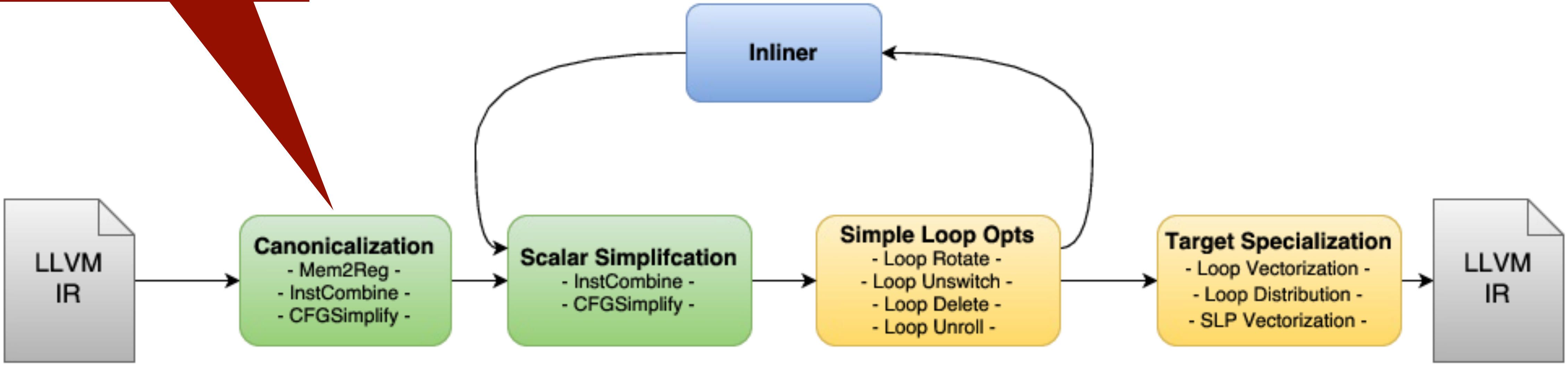
No computational code in Latch!

Loop Simplify Form

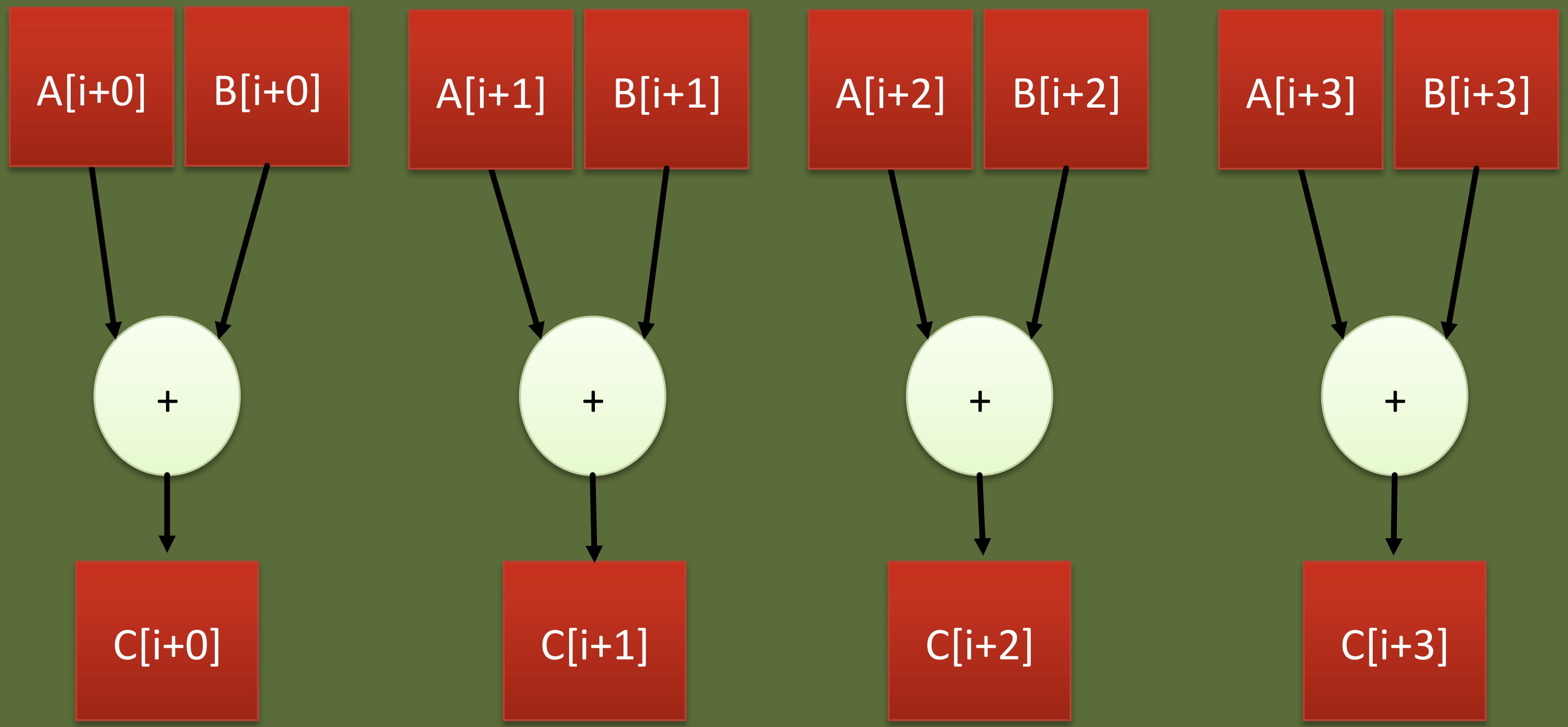


The LLVM Pass Pipeline

Canonicalization is essential for analysis to work!



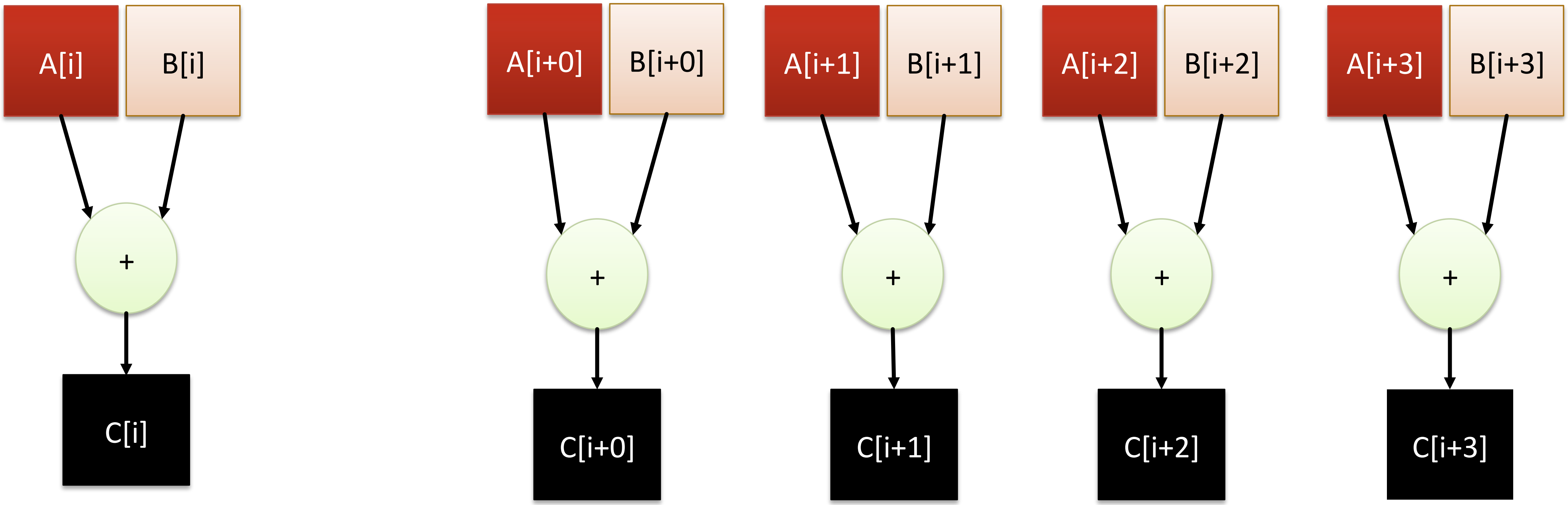
Automatic Vectorization (SIMDization)



Recap: Vectorization (SIMDization)

C[i] = A[i] + B[i]

C[i:i+3] = A[i:i+3] + B[i:i+3]



Scalar Execution

SIMD (Single Instruction Multiple Data) Execution

Intel SIMD Extensions

Vector width grows constantly!

SIMDization as solution for higher-performance at constant frequency

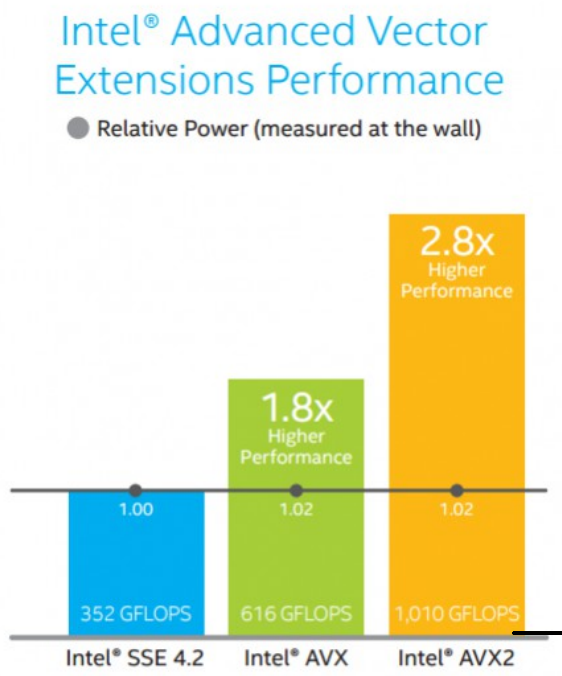


Figure 1. Measuring performance on the same processor using Linpack* benchmarks shows substantial increases from Intel® Streaming SIMD Extensions 4.2 (Intel® SSE 4.2) to Intel® Advanced Vector Extensions (Intel® AVX) and from Intel AVX to Intel® AVX2, with up to 2.8x the GFLOPS throughput when comparing Intel SSE 4.2 to Intel AVX2.[2]

Wider Vectors drive performance growth!

State-of-the-art SIMD Instruction Set Extensions

Property	Intel / AMD	ARM / ARM64	ARM HPC	PowerPC
Name	AVX 512	NEON	SVE	AltiVEC
Vector Size [Bits]	512	128	128 – 2048	128
Vector Size [floats]	16	4	4 – 64	4
Vector Size [doubles]	8	2	2 – 32	2

Introduced predicated loads/stores into LLVM

Requires LLVM-IR changes (not yet implemented)

How to write SIMD Code

Option 1: Manually Write SIMD Code

- Use intrinsic or write assembly code
- **Pro**
  - Maximal control
- **Con**
  - Complex
  - Not portable
  - Not available in Java

Option 2: Auto-generated SIMD Code

- Automatic Loop Vectorization techniques to introduce SIMD instructions
- **Pro**
  - Automatic
  - Portable
- **Con**
  - Not always statically provable
  - Java compilers not good at it

LLVM IR Vector Instructions

```
define i32 @foo(<4 x i32>* %P0, <4 x i32>* %P1, <6 x i32>* %P2) {
  %V0 = load <4 x i32>, <4 x i32>* %P0
  %V1 = load <4 x i32>, <4 x i32>* %P1

  %V2 = add <4 x i32> %V0, %V1

  %V3 = shufflevector <4 x i32> %V2, <4 x i32> <i32 1, i32 1, i32 1, i32 1>,
        <6 x i32> <i32 0, i32 4, i32 1, i32 2, i32 3, i32 5>

  %VX =  insertelement <6 x i32> %V3, i32 42, i32 1
  %val = extractelement <6 x i32> %VX, i32 0

  store <6 x i32> %VX, <6 x i32>* %P2
  ret i32 %val
}
```

SIMD Type

SIMD Load

SIMD Arithmetic

SIMD Shuffle

SIMD Per-element Access

SIMD Store

http://llvm.org/docs/LangRef.html

C/C++ Vector Extension

```
typedef int int4 __attribute__((__vector_size__(16)));
typedef int int6 __attribute__((__vector_size__(24)));

int foo(int4* P0, int4* P1, int6* P2) {
  int4 V0 = *P0;
  int4 V1 = *P1;
  int4 V2 = V0 + V1;
  int4 Constants = {1, 1, 1, 1};
  int6 V3 = __builtin_shufflevector(V2, Constants, 0, 4, 1, 2, 3, 5);
  V3[1] = 42;
  *P2 = V3;
  return V3[0];
}
```

Operations on Vector Extensions

Operator	OpenCL	AltiVec	GCC	NEON
[]	Yes	Yes	Yes	-
Unary +, -	Yes	Yes	Yes	-
++, --	Yes	Yes	Yes	-
+, -, *, /, %	Yes	Yes	Yes	-
Bitwise &, |, ^, ~	Yes	Yes	Yes	-
>>, <<	Yes	Yes	Yes	-
!, &&, ||	Yes	-	-	-
==, !=, >, <, >=, <=	Yes	Yes	-	-
=	Yes	Yes	Yes	Yes
?:	Yes	-	-	-
sizeof	Yes	Yes	Yes	Yes

avxintrin.h: Vector addition

Intrinsics work on generic vector types

```
typedef float __m256 __attribute__ ((__vector_size__ (32)));

/// \brief Subtracts two 256-bit vectors of [8 x float].
///
/// This intrinsic corresponds to the <c> VSUBPS </c> instruction.
///
/// \param __a  A 256-bit vector of [8 x float] containing the minuend.
/// \param __b  A 256-bit vector of [8 x float] containing the subtrahend.
/// \returns    A 256-bit vector of [8 x float] containing the differences between
///    both operands.
static __inline __m256 __DEFAULT_FN_ATTRS
_mm256_sub_ps(__m256 __a, __m256 __b)
{
  return (__m256)((__v8sf)__a - (__v8sf)__b);
}
```

avxintr.h: Implementation of VMOVSHDUP

```
/// \brief Moves and duplicates high-order (odd-indexed) values from a 256-bit
///    vector of [8 x float] to float values in a 256-bit vector of [8 x float].
///
/// \param a
///    A 256-bit vector of [8 x float]. \n
///    Bits [255:224] of a are written to bits [255:224] and [223:192] of return value.
///    Bits [191:160] of a are written to bits [191:160] and [159:128] of return value.
///    Bits [127: 96] of a are written to bits [127: 96] and [ 95: 64] of return value.
///    Bits [ 63: 32] of a are written to bits [ 63: 32] and [ 31:  0] of return value.
/// \returns A 256-bit vector of [8 x float] containing the moved and duplicated values.
static __inline __m256 __DEFAULT_FN_ATTRS
_mm256_movehdup_ps(__m256 a)
{
  return __builtin_shufflevector((__v8sf)__a, (__v8sf)__a, 1, 1, 3, 3, 5, 5, 7, 7);
}
```

Most operations are lowered to generic vector builtins!

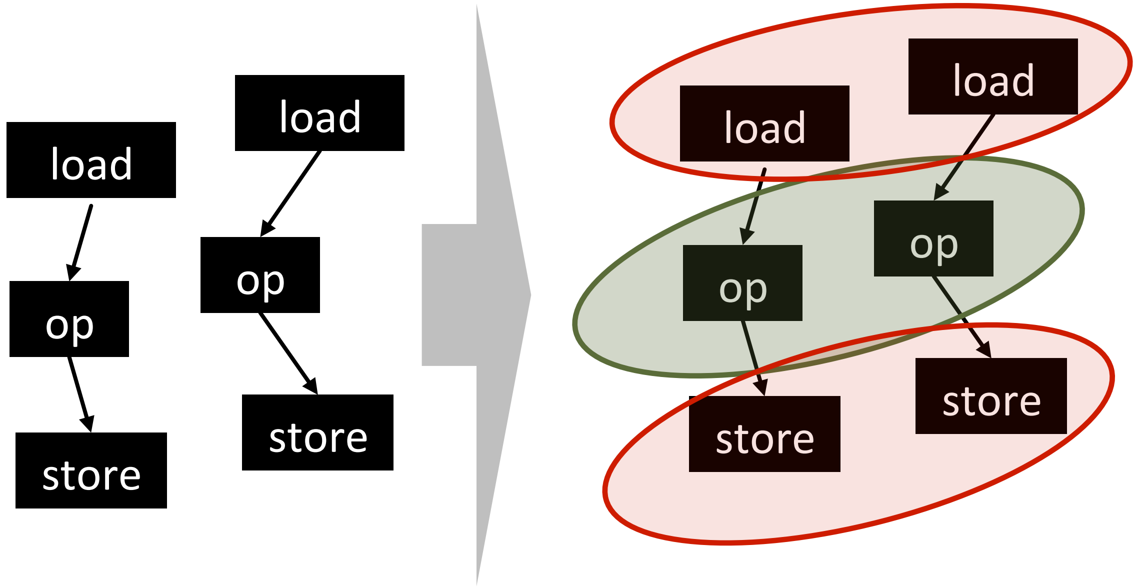
Different Kinds of Automatic SIMDization

Loop Vectorization

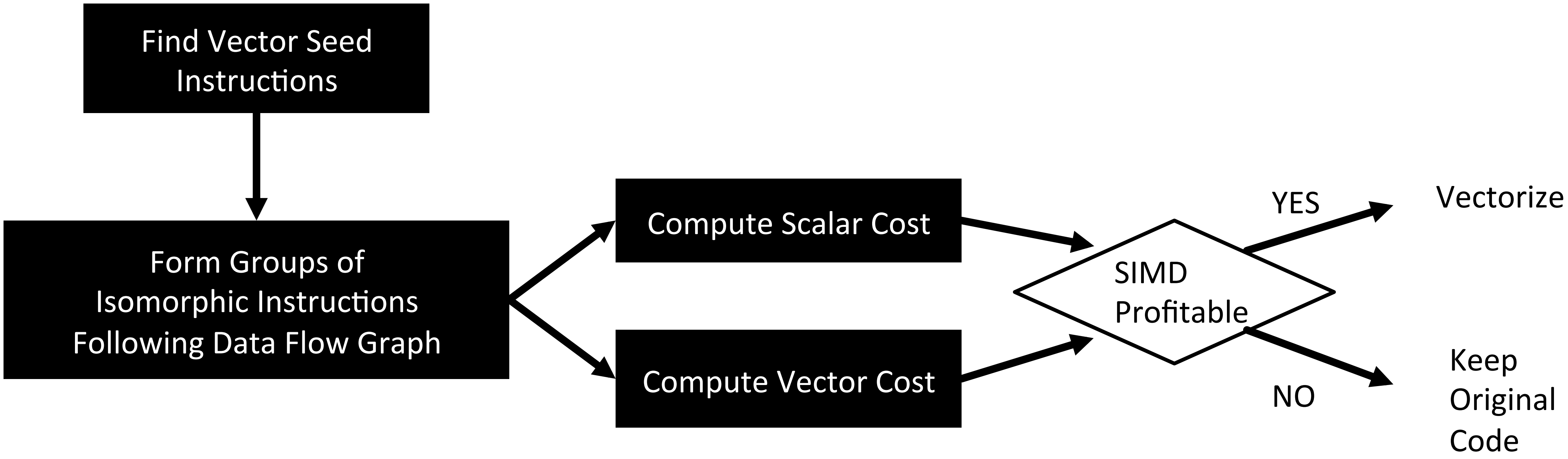
```
for (i = 0; i < n; i++)
  A[i] = …
```

```
for (i = 0; i < n; i+=X)
  A[i:i+X] = …
```

Superworld Level Parallelism (SLP) SIMDization



SLP Vectorization



SLP Vectorization - Example

```
b = a[i+0]
c = 5
d = b + c

e = a[i+1]
f = 6
g = e + f

h = a[i+2]
j = 7
k = h + j
```

```
c = 5
d = b + c

f = 6
g = e + f

j = 7
k = h + j
```

```
b = a[i+0]
e = a[i+1]

e = a[i+1]
h = a[i+2]
```

```
c = 5
f = 6
j = 7
```

```
b = a[i+0]
e = a[i+1]

e = a[i+1]
h = a[i+2]

d = b + c
g = e + f

g = e + f
k = h + j
```

Exploiting Superword Level Parallelism with Multimedia Instruction Sets, PLDI 2000, Samuel Larsen and Saman Amarasinghe

SLP Vectorization - Example

```
c = 5
f = 6
j = 7
```

```
b = a[i+0]
e = a[i+1]

e = a[i+1]
h = a[i+2]

d = b + c
g = e + f

g = e + f
k = h + j
```

```
b = a[i+0]
e = a[i+1]

e = a[i+1]
h = a[i+2]

d = b + c
g = e + f

g = e + f
k = h + j

c = 5
f = 6

f = 6
j = 7
```

```
b = a[i+0]
e = a[i+1]
h = a[i+2]

d = b + c
g = e + f
k = h + j

c = 5
f = 6
j = 7
```

Exploiting Superword Level Parallelism with Multimedia Instruction Sets, PLDI 2000, Samuel Larsen and Saman Amarasinghe

Cost Evaluation

b = a[i+0]	1
c = 5	1
d = b + c	1
e = a[i+1]	1
f = 6	1
g = e + f	1
h = a[i+2]	1
j = 7	1
k = h + j	1

9

b = a[i+0] e = a[i+1] h = a[i+2]	2
d = b + c g = e + f k = h + j	1
c = 5 f = 6 j = 7	2

5 ✓

SLP Vectorization – Seed Instructions

- Instructions that access neighboring memory locations
- Today, GCC and LLVM **start from store instructions**
- In general, any two independent instructions are valid seed instructions

Inner Loop Vectorization

```
for (int i = 0; i < 1024; i++)
    B[i] += A[i];
```

↓

```
for (int i = 0; i < 1024; i+=4)
    B[i:i+3] += A[i:i+3];
```

Automatic (Inner) Loop Vectorization

- Validity
  - Innermost loop must be parallel (or behave after vectorization as if it was)
  - No aliasing between different arrays
- Profitability
  - Memory accesses must be “stride-one”

    or
  - Computational cost must dominate the loop

Can these loops be vectorized?

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

YES, the arrays are different objects

```
for (int i = 0; i <= n; i++)
  A[i] += A[i];
```

YES, there is no dependence to any previous iteration

Can these loops be vectorized?

```
for (int i = 1; i <= n; i++)
  A[i] += A[i] + A[i – 1];
```

NO, iteration i depends on iteration i – 1

Can these loops be vectorized: pointer-to-pointer arrays

```
int[ ][ ] A = new int[N][M];
int[ ][ ] B = new int[N][M];

for (int i = 0; i <= N; i++)
  for (int j = 0; i <= M; i++)
     A[i][j] += B[i][j];
```

YES, in C/C++/Fortran array dimensions are independent

We now assume multi-dimensional arrays in the mathematical sense

Can these loops be vectorized?

```
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    for (k = 0; k < K; k++)
      C[i][j] += A[i][k] * B[k][j];
```

NO, the inner loop has data-dependences between subsequent iterations

Can these loops be vectorized?

```
for (i = 0; i < N; i++)
  for (k = 0; k < K; k++)
    for (j = 0; j < M; j++)
      C[i][j] += A[i][k] * B[k][j];
```

Interchange

YES, the inner loop has no data-dependences between subsequent iterations

Advanced Support for SIMDization

Target Transform Info [include/llvm/Analysis/TargetTransformInfo.h]

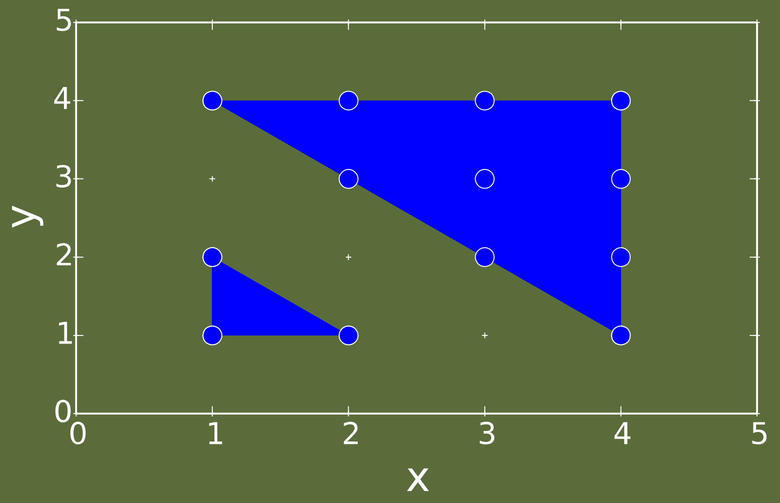
Target Specific Cost Estimates Without Instruction Selection

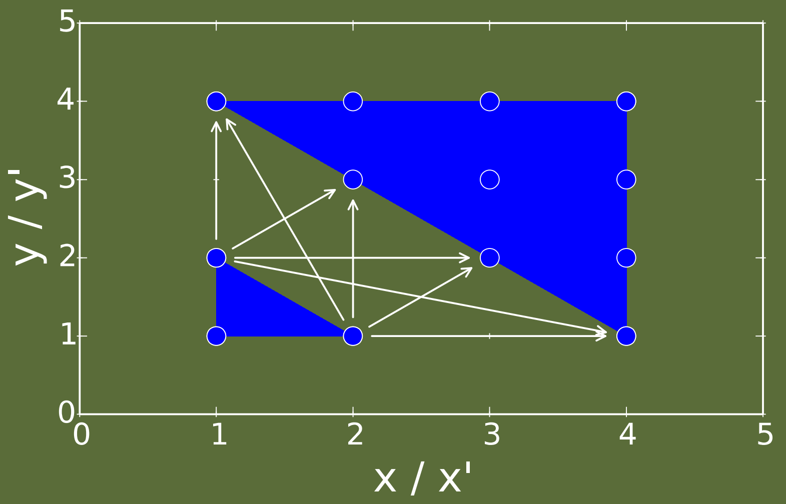
```
/// \return The expected cost of arithmetic ops, such as mul, xor, fsub, etc.
/// \p Args is an optional argument which holds the instruction operands
/// values so the TTI can analyze those values searching for special
/// cases\optimizations based on those values.
int getArithmeticInstrCost(
    unsigned Opcode, Type *Ty, OperandValueKind Opd1Info = OK_AnyValue,
    OperandValueKind Opd2Info = OK_AnyValue,
    OperandValueProperties Opd1PropInfo = OP_None,
    OperandValueProperties Opd2PropInfo = OP_None,
    ArrayRef<const Value *> Args = ArrayRef<const Value *>()) const;

/// \return The cost of a shuffle instruction of kind Kind and of type Tp.
/// The index and subtype parameters are used by the subvector insertion and
/// extraction shuffle kinds.
int getShuffleCost(ShuffleKind Kind, Type *Tp, int Index = 0,
                   Type *SubTp = nullptr) const;
```

LoopAccessAnalysis

- Analyze Innermost Loops
- Check data-dependences and legality of SIMDization
- Generates run-time Alias Checks
- Analysis the Stride of Memory Accesses

Presburger Sets and Relations





Quasi-Affine Expression

- Base
  - Constants (c_i)
  - Parameters (p_i)
  - Variables (v_i)
- Operations
  - Negation ($-e$)
  - Addition ($e_0 + e_1$)
  - Multiplication by constant ($c * e$)
  - Division by constant (c/e)
  - Remainder of constant division ($e \bmod c$)

```
void foo (int n, int m) {

  for (int i = 0; …; …)
       int tmp = …
       for (int j = 0; …; …) {





  }
}
```

Presburger Formula

- Base
  - Boolean Constants (⊤, ⊥)

- Operations
  - Comparisons of quasi-affine expressions

    $e_0 \oplus e_1 , \oplus \{<, \leq, =, \neq, \geq, >\}$
  - Boolean Operations between Presburger Formula

    $p_0 \otimes p_1 , \otimes \{ \wedge, \vee, \neg, \Rightarrow, \Leftarrow, \Leftrightarrow \}$
  - Quantified Variables

    $\exists x | p(x, \ldots)$

    $\forall x | p(x, \ldots)$

Presburger Sets and Relations

Presburger Set

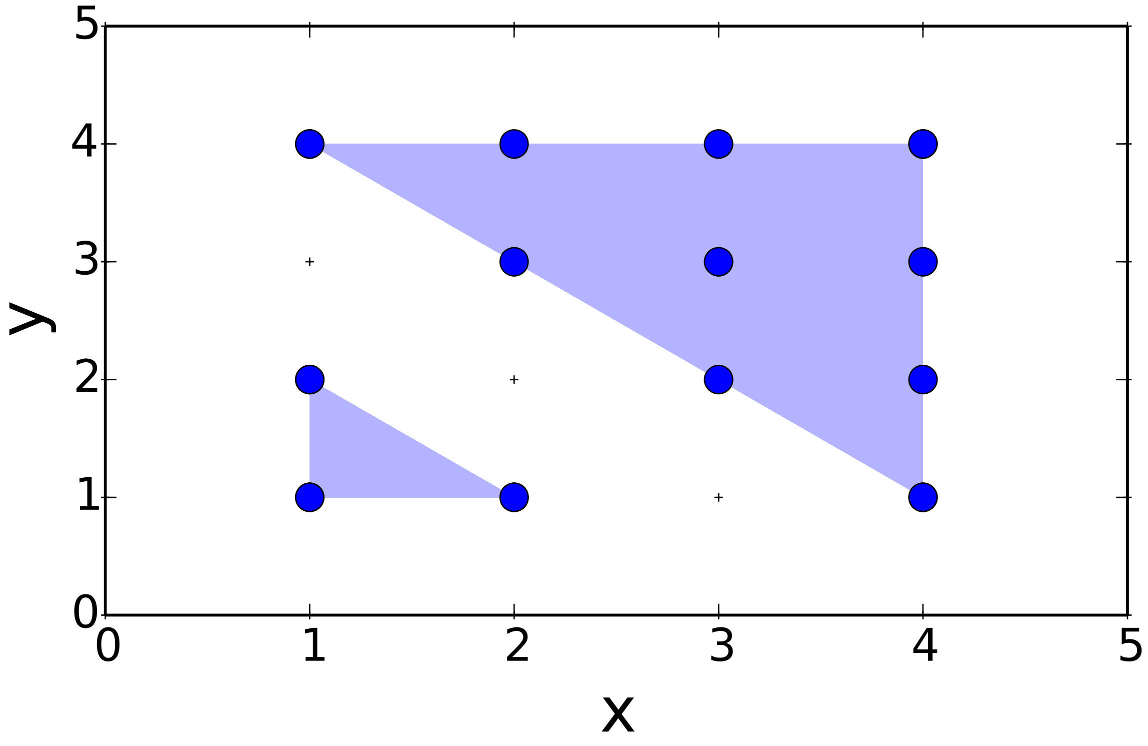
$$S = p \rightarrow \{ v \mid v \in \mathbb{Z}^{n} : p(v, p) \}$$

Presburger Relation

$$R = p \rightarrow \{ v_{0} \rightarrow v_{1} \mid v_{0} \in \mathbb{Z}^{n}, v_{1} \in \mathbb{Z}^{m} : p(v_{0}, v_{1}, p) \}$$

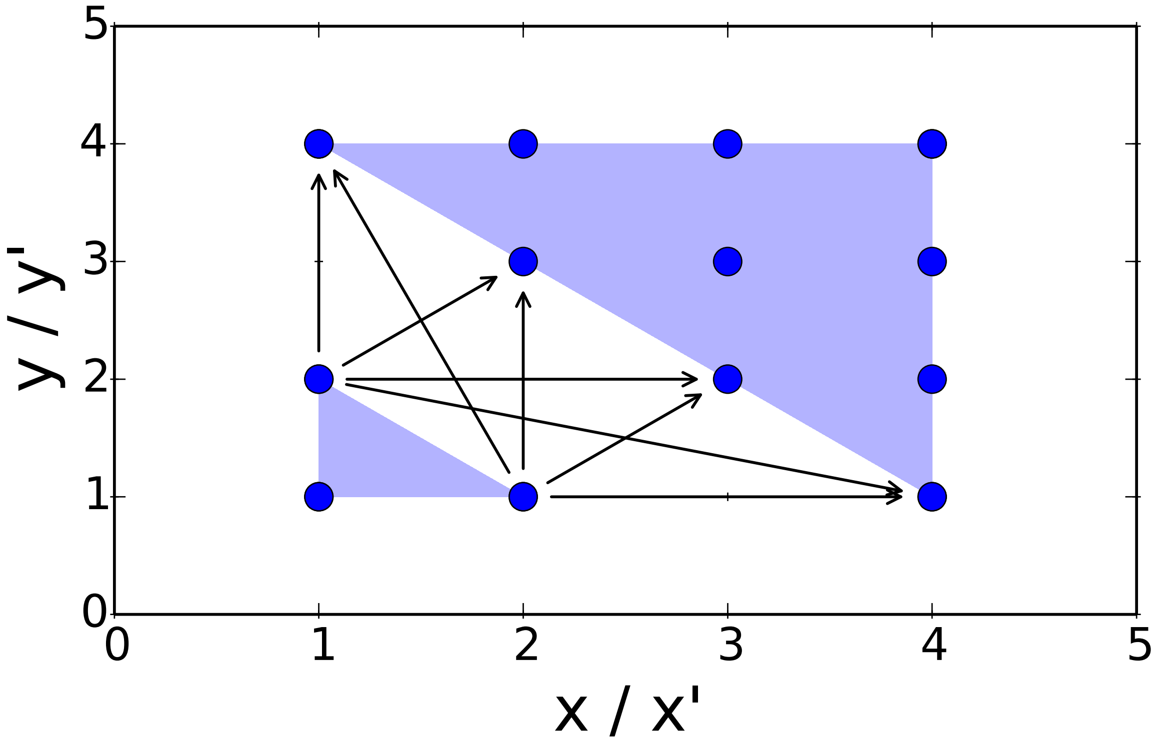
Example: Presburger Set

S = $(x,y) 1 \le x, y \le 4 \wedge (x+y \le 3 \vee x+y \ge 5)$



Example: Presburger Map

$$R = \{ (x,y) \to (x', y') \mid x+y=3 \wedge x'+y'=5 \}$$



Presburger Arithmetic

- Benefits
  - Decidable
  - Closed under common operations

    $\cap$, $\cup$, $\setminus$, proj, $\circ$, not transitive hull

  Þ Precise results

- Computational Complexity
  - Some operations double-exponential (in dimensions)
  - Often lower complexity for bounded dimension

Can we solve more complex Diophantine equations?

- Does $x^3 + y^3 = z^3$ with x,y,z∈ℤ have a solution?

  No, Fermat's last theorem! Answered in 1994, after year 357 years!

- Does $x^3 + y^3 + z^3 = 29$ have a solutior
- Does $x^3 + y^3 + z^3 = 33$ have a solutior ’!

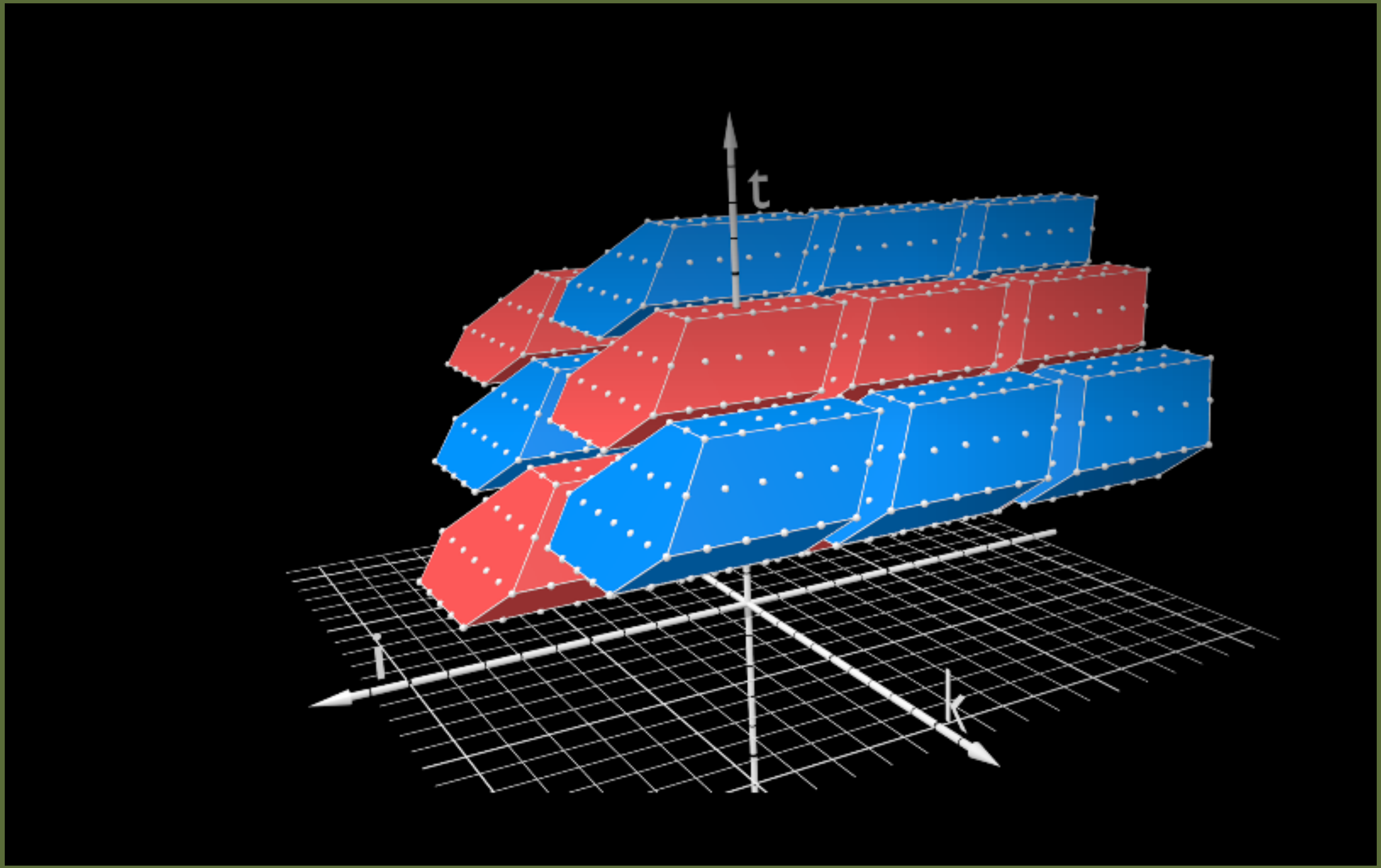
Note: No general algorithm for solving polynomial equations over integers exists!
(Hilbert's 10^{th} problem)

Proof is interesting: encodes Turing machine in Diophantine equations

Undecidability in number theory, Bjorn Poonen, Notices Amer. Math. Soc. 55 (2008)

Demo: Presburger Sets

Modeling Loop Programs with Presburger Sets



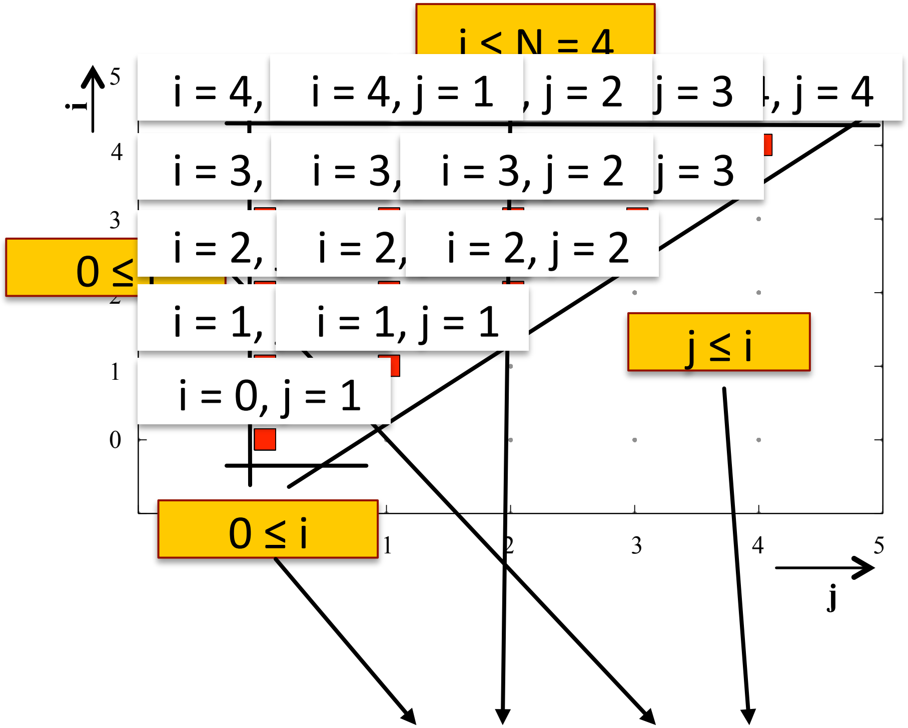
Polyhedral Loop Modeling

Program Code

```
for (i = 0; i <= N; i++)
  for (j = 0; j <= i; j++)
    S(i,j);
```

N = 4

Iteration Space



D = { (i,j) | 0 ≤ i ≤ N ⋀ 0 ≤ j ≤ i }

Static Control Parts - SCoPs

- Structured Control
  - IF-conditions
  - Counted FOR-loops (Fortran style)
- Multi-dimensional array accesses (and scalars)
- Loop-conditions and IF-conditions are Presburger Formula
- Loop increments are constant (non-parametric)
- Array subscript expressions are piecewise-affine

Þ Can be modeled precisely with Presburger Sets

Polyhedral Model of Static Control Part

```
for (i = 0; i <= N; i++)
  for (j = 0; j <= i; j++)
S:  B[i][j] = A[i][j] + A[i][j+1];
```

- **Iteration Space (Domain)**

$I\downarrow S = S(i,j) 0 \leq i \leq N \wedge 0 \leq j \leq i$

- **Schedule**

$\theta\downarrow S = \{\ S(i,j) \to (i,j)\ \}$

- **Access Relation**
  - Reads: $\{S(i,j) \to A(i,j); S(i,j) \to A(i, j+1)\}$
  - Writes: $\{S(i,j) \to B(i,j)\}$

Polyhedral Schedule: Original

Model

$$I↓S = S(i,j) 0 \leq i \leq n \wedge 0 \leq j \leq i$$

$$\theta↓S = \{S\,(i,j) \rightarrow (i,j)\} \rightarrow (\lfloor i/4 \rfloor, j, i \bmod 4)$$

Code

```
for (i = 0; i <= n; i++)
  for (j = 0; j <= i; j++)
    S(i, j);
```

Polyhedral Schedule: Original

Model

$$I_S = S(i,j) \; 0 \leq i \leq n \wedge 0 \leq j \leq i$$

$$\theta_S = \{S\,(i,j) \rightarrow (i,j)\} \quad \rightarrow (\lfloor i/4 \rfloor, j, i \; mod \; 4)$$

Code

```
for (c0 = 0; c0 <= n; c0++)
  for (c1 = 0; c1 <= c0; c1++)
    S(c0, c1);
```

Polyhedral Schedule: Interchanged

Model

$$I_S = S(i,j)\ 0 \le i \le n \wedge 0 \le j \le i$$

$$\theta_S = \{S\,(i,j) \rightarrow (j,\, i)\} \rightarrow (\lfloor i/4 \rfloor, j, i \bmod 4)$$

Code

```
for (c0 = 0; c0 <= n; c0++)
  for (c1 = c0; c1 <= n; c1++)
    S(c1, c0);
```

Polyhedral Schedule: Strip-mined

Model

$$I_S = S(i,j) \mid 0 \le i \le n \wedge 0 \le j \le i$$

$$\theta_S = \{S(i,j) \to (\lfloor i/4 \rfloor, j, i \bmod 4)\}$$

Code

```
for (c0 = 0; c0 <= floord(n, 4); c0++)
  for (c1 = 0; c1 <= min(n, 4 * c0 + 3); c1++)
    for (c2 = max(0, -4 * c0 + c1);
         c1 <= min(3, n - 4 * c0); c2++)
      S(4 * c0 + c2, c1);
```

Polyhedral Schedule: Blocked

Model

$$I_S = S(i,j) \mid 0 \le i \le n \wedge 0 \le j \le i$$

$$\theta_S = \{S(i,j) \to (\lfloor i/4 \rfloor, \lfloor j/4 \rfloor, i \bmod 4, j \bmod 4)\}$$

Code

```
for (c0 = 0; c0 <= floord(n, 4); c0++)
  for (c1 = 0; c1 <= c0; c1++)
    for (c2 = 0; c2 <= min(3, n - 4 * c0); c2++)
      for (c3 = 0; c3 <= min(3, 4 * c0 – 4 * c1 + c2); c3++)
        S(4 * c0 + c2, 4 * c1 + c3);
```

How to derive a good schedule

Stepwise Improvement	Construct "perfect" Schedule

• Interchange
• Fusion
• Distribution
• Skewing
• Tiling
• Unroll-and-Jam
	• ***Feautrier Scheduler***
	  • Resolve data-dependences at outer levels
	  • Maximize inner parallelism
	
	• ***Pluto Scheduler***
	  • Resolve data-dependences at inner levels
	  • Maximize outer parallelism
	  • Fusion model to minimize dependence distances

Classical Loop Transformations – Loop Reversal

```
// Original Loop
for (i = 0; i <= n; i+=1)
  S(i);
```

```
// Transformed Loop
for (i = n; i >= 0; i-=1)
  S(i);
```

$$D_I = S(i) 0 \le i \le n$$
$$S_{Orig} = \{ S(i) \rightarrow (i) \}$$
$$S_T = \{ S(i) \rightarrow (n - i) \}$$

Classical Loop Transformations – Loop Interchange

```
// Original Loop
for (i = 0; i <= n; i+=1)
  for (j = 0; j <= n; j+=1)
    S(i,j);
```

```
// Transformed Loop
for (j = 0; j <= n; j+=1)
  for (i = 0; i <= n; i+=1)
    S(i,j);
```

$D_I = S(i,j) 0 \leq i,j \leq n$

$S_{Orig} = \{ S(i,j) \rightarrow (i,j) \}$

$S_T = \{ S(i,j) \rightarrow (j,i) \}$

Classical Loop Transformations – Fusion

```
// Original Loop
for (i = 0; i <= n; i+=1)
  S1(i);
for (i = 0; i <= n; i+=1)
  S2(i);
```

```
// Transformed Loop
for (i = 0; i <= n; i+=1) {
  S1(i);
  S2(i);
}
```

$$D{\downarrow}I = S(i) 0 \le i \le n T(i) | 0 \le j \le n$$
$$S{\downarrow}Orig = \{ S(i) \to (0,i); T(i) \to (1,i) \}$$
$$S{\downarrow}T = \{ S(i) \to (i, 0); T(i) \to (i,1) \}$$

Classical Loop Transformations – Fission (also called Distribution)

```
// Original Loop
for (i = 0; i <= n; i+=1) {
  S1(i);
  S2(i);
}
```

```
// Transformed Loop
for (i = 0; i <= n; i+=1)
  S1(i);
for (i = 0; i <= n; i+=1)
  S2(i);
```

$$D_I = S(i) 0 \le i \le n T(i) | 0 \le j \le n$$
$$S_{Orig} = \{ S(i) \to (i,0); T(i) \to (i,1) \}$$
$$S_T = \{ S(i) \to (0,1); T(i) \to (1,i) \}$$

Classical Loop Transformations – Skewing

```
// Original Loop
for (i = 0; i <= n; i+=1)
  for (j = 0; j <= n; j+=1)
    S(i,j);
```

```
// Transformed Loop
for (i = 0; i <= n; i+=1)
  for (j = i+1; j <=n+i; j+=1)
    S(i,j);
```

$$D_I = S(i,j) 0 \le i,j \le n$$
$$S_{Orig} = \{ S(i,j) \rightarrow (i,j) \}$$
$$S_T = \{ S(i,j) \rightarrow (i,i+j)$$

Classical Loop Transformations – Strip-Mining

```
// Original Loop
for (i = 0; i <= 1024; i+=1)
  S(i);
```

```
// Transformed Loop
for (i = 0; i <= 1024; i+=4)
  for (ii = i; ii <= i+3; ii+=1)
    S(ii);
```

$$D_I = S(i) 0 \leq i \leq n$$
$$S_{Orig} = \{ S(i) \rightarrow (i) \}$$
$$S_T = \{ S(i) \rightarrow (\lfloor i/4 \rfloor, i) \}$$

Classical Loop Transformations – Blocking (Tiling)

```
// Original Loop
for (i = 0; i <= 1024; i+=1)
  for (j = 0; j <= 1024; j+=1)
    S(i,j);
```

```
// Transformed Loop
for (i = 0; i <= 1024; i+=8)
  for (j = 0; j <= 1024; j+=8)
    for (ii = i; ii <= i+8; ii+=1)
     for (jj = j; jj <= j+8; j+=1)
       S(ii, jj);
```

$$D{\downarrow}I = S(i,j) 0 \le i,j \le n$$
$$S{\downarrow}Orig = \{ S(i,j) \rightarrow (i,j) \}$$
$$S{\downarrow}T = \{ S(i) \rightarrow (\lfloor i/4 \rfloor, \lfloor j/4 \rfloor, i, j) \}$$

Legality of Loop Transformations

1. **Conflicting Accesses**
   Two statement instance access the same memory location
2. **Execution**
   Each statement instance is known to be executed
3. **At least one write access**
   Two memory reads do not conflict

The direction of the data dependency is defined through the schedule.

Conditions for Data Dependence

1. **Conflicting Accesses**
   Two statement instance access the same memory location
2. **Execution**
   Each statement instance is known to be executed
3. **At least one write access**
   Two memory reads do not conflict

The direction of the data dependency is defined through the schedule.

Data Dependence Types

- **Read-After-Write (true)**
  - Flow (subset of RAW-dependences that carries data)
- **Write-After-Read (anti)**
- **Write-After-Write (output)**
- **Read-After-Read**

False dependences: Write-After-Read + Write-After-Write

Precision of Data Dependences

```
Example: for I = 0..N
           for J = 0..N
             for K = 0..N
               A(I+1, J, K-1) = A(I, J, K)
```

- **Direction Vectors**

  Dependences are tuples over: +, -, =

  D(+, =, -)

- **Distance Vectors**

  Dependences are given through their integer distance

  D(1, 0, -1)

- **Presburger Sets**

  Dependences are described as Presburger Relations

$$\{(I,J,K) \to (I+1, J, K-1) \mid 0 \leq I, J, K \leq N\}$$

Invariants on Dependences

- **The first non-zero component must be positive**

  Otherwise, the dependence goes backwards in time

Validity of a Schedule

A schedule θ_S is valid for an iteration space I_S and a set of dependences D_S, iff $\forall (s,d) \in D_S : \theta_S(s) \prec \theta_S(d)$.

Loop Carried Dependences

- A data dependence **D** is carried by a loop **L** that corresponds to the first non-zero dimension of the dependence vector

```
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    for (k = 0; k < K; k++)
      C[i][j] += …
```

D(0, 0, +1)

```
for (i = 0; i < N; i++)
  for (k = 0; k < K; k++)
    for (j = 0; j < M; j++)
      C[i][j] += …
```

D(0, +1, 0)

Parallel Loops

- **A loop is parallel if it does not carry any data dependences**

```
parfor (i = 0; i < N; i++)
  parfor (j = 0; j < M; j++)
    for (k = 0; k < K; k++)
      C[i][j] += …
```

D(0, 0, +1)

```
parfor (i = 0; i < N; i++)
  for (k = 0; k < K; k++)
    parfor (j = 0; j < M; j++)
      C[i][j] += …
```

D(0, +1, 0)

Elimination of Scalar Dependences: Static Array Expansion

```
for (i = 0; i < 100; i++) {
  tmp = A[i];
  A[i] = B[i];
  B[i] = tmp;
}
```

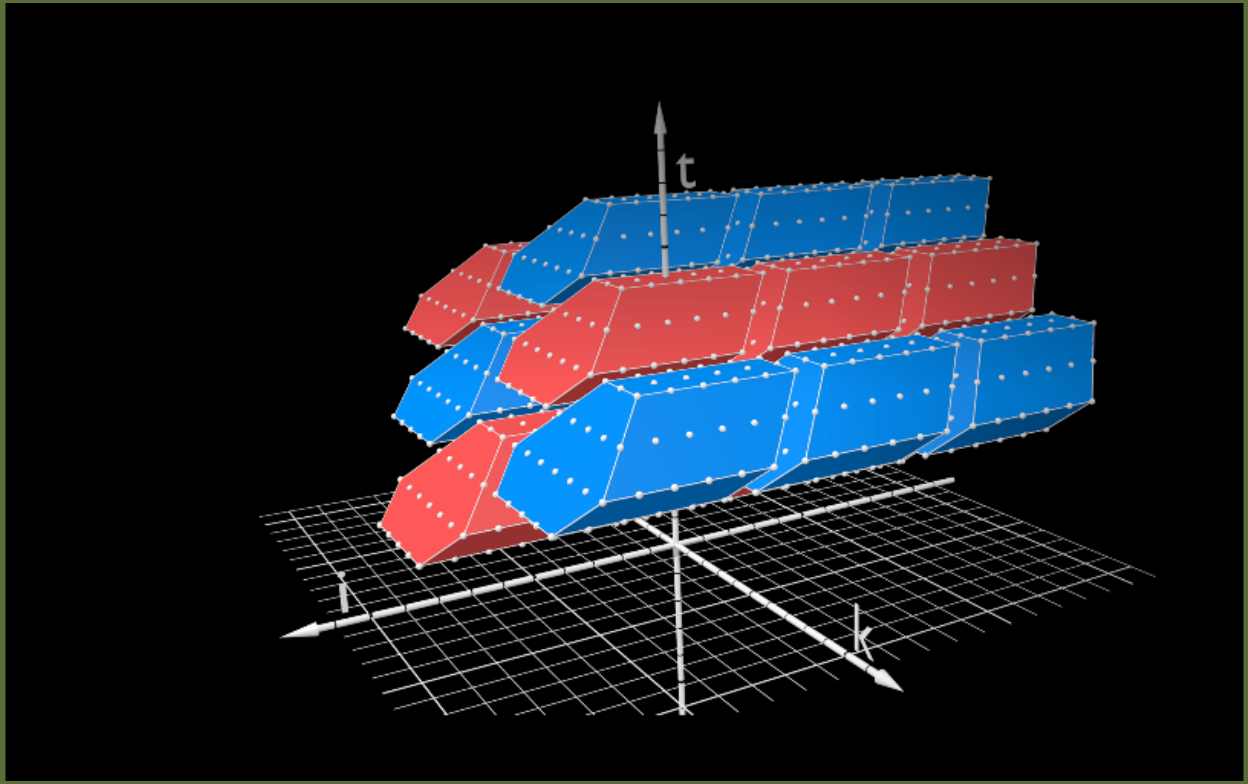
```
for (i = 0; i < 100; i++) {
  TMP[i] = A[i];
  A[i] = B[i];
  B[i] = TMP[i];
}
```

A loop carried write-after-read (anti) dependence prevents parallel execution.

Transform scalar **tmp** into an array **TMP** that contains for each loop iteration private storage.

Demo: Loop Modeling with Presburger Sets

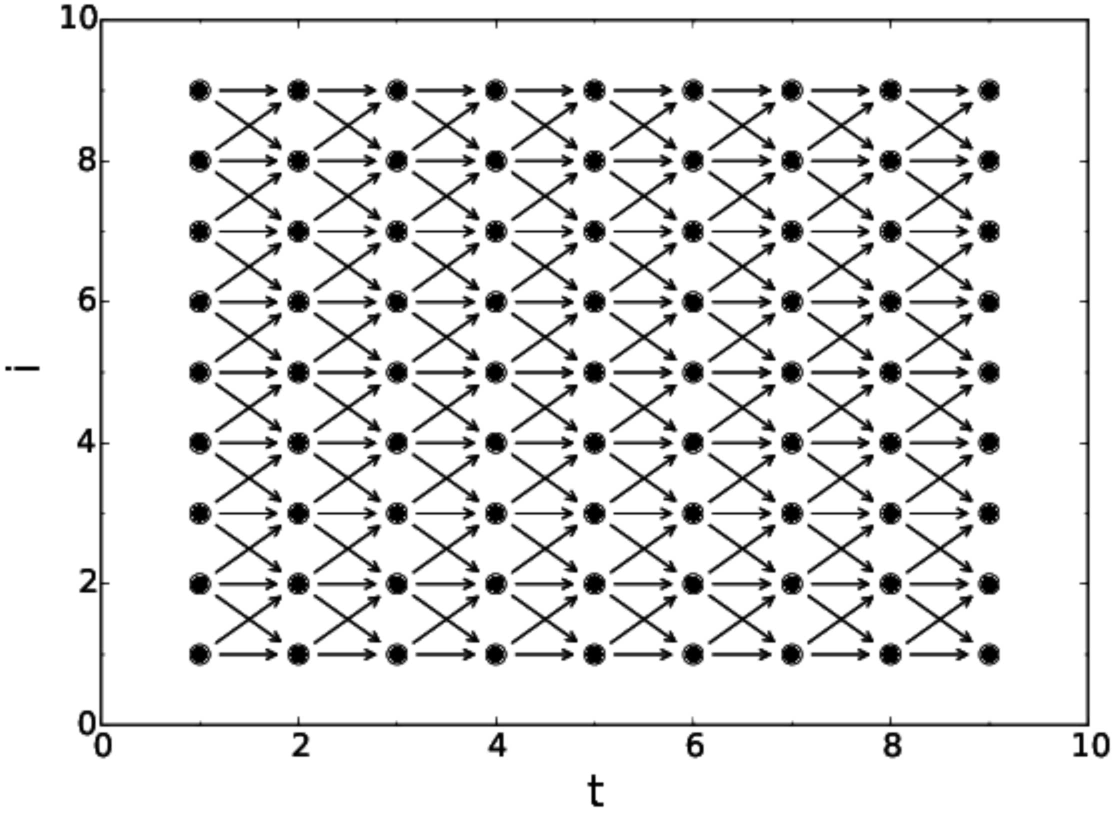
Tiling for Data-Locality and Parallelism



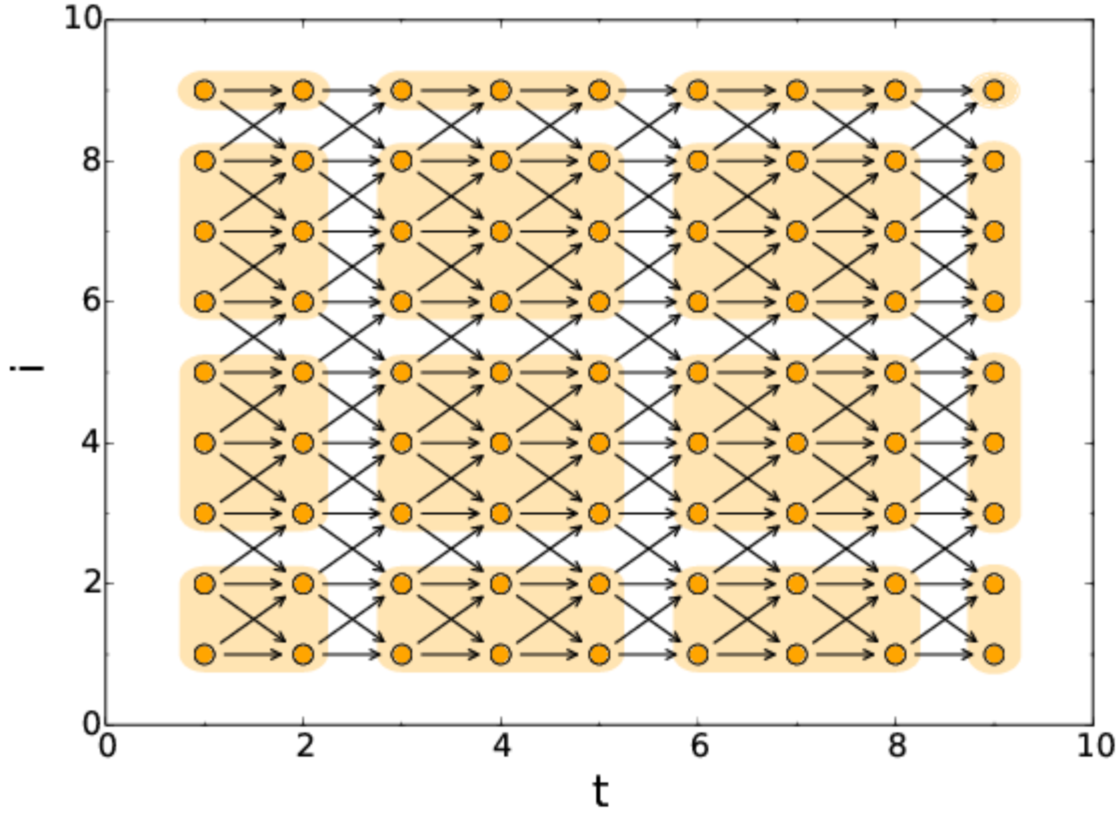
Tiling of a 1D Stencil

```
for (int t = 0; t < T; t++)
  for (int i = 0; i < N; i++)
    A[t+1][i] = A[t][i] + A[t][i-1] + A[t][i+1];
```

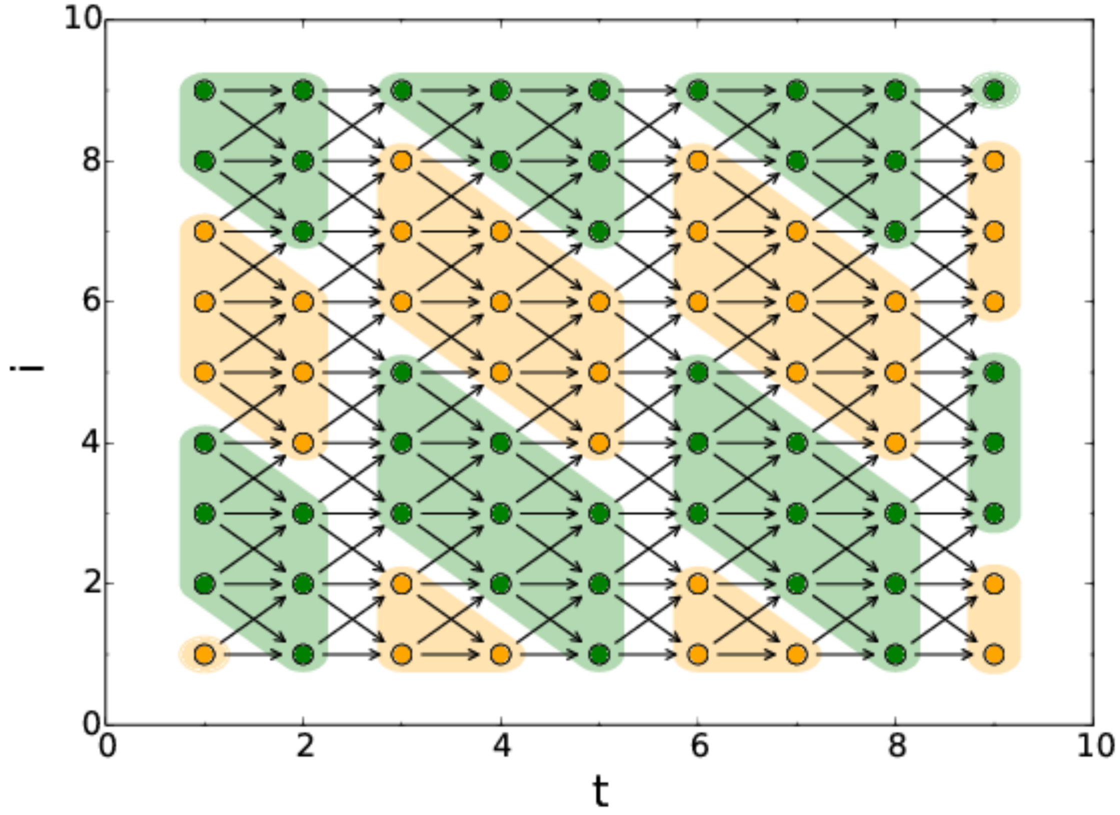
Jacobi Stencil with 1D Space + Time



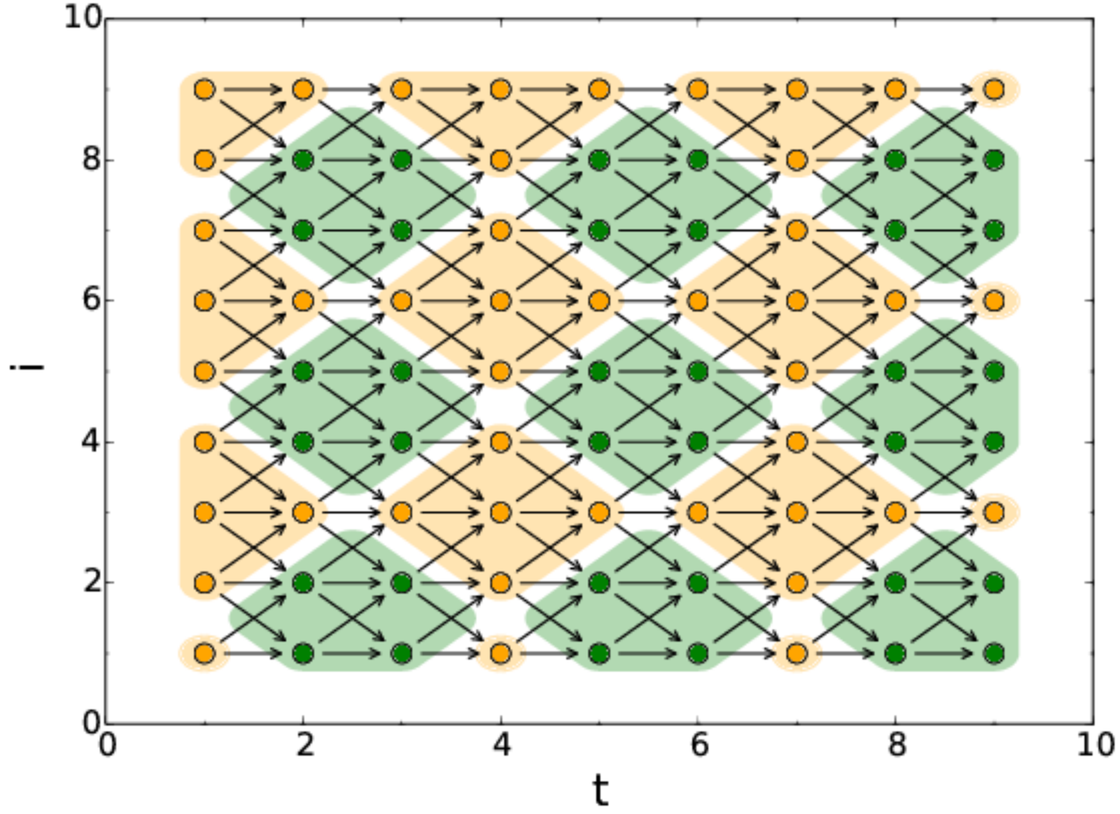
Jacobi Stencil with 1D Space + Time: Rectangular Tiles



Jacobi Stencil with 1D Space + Time: Skewed and Tiled



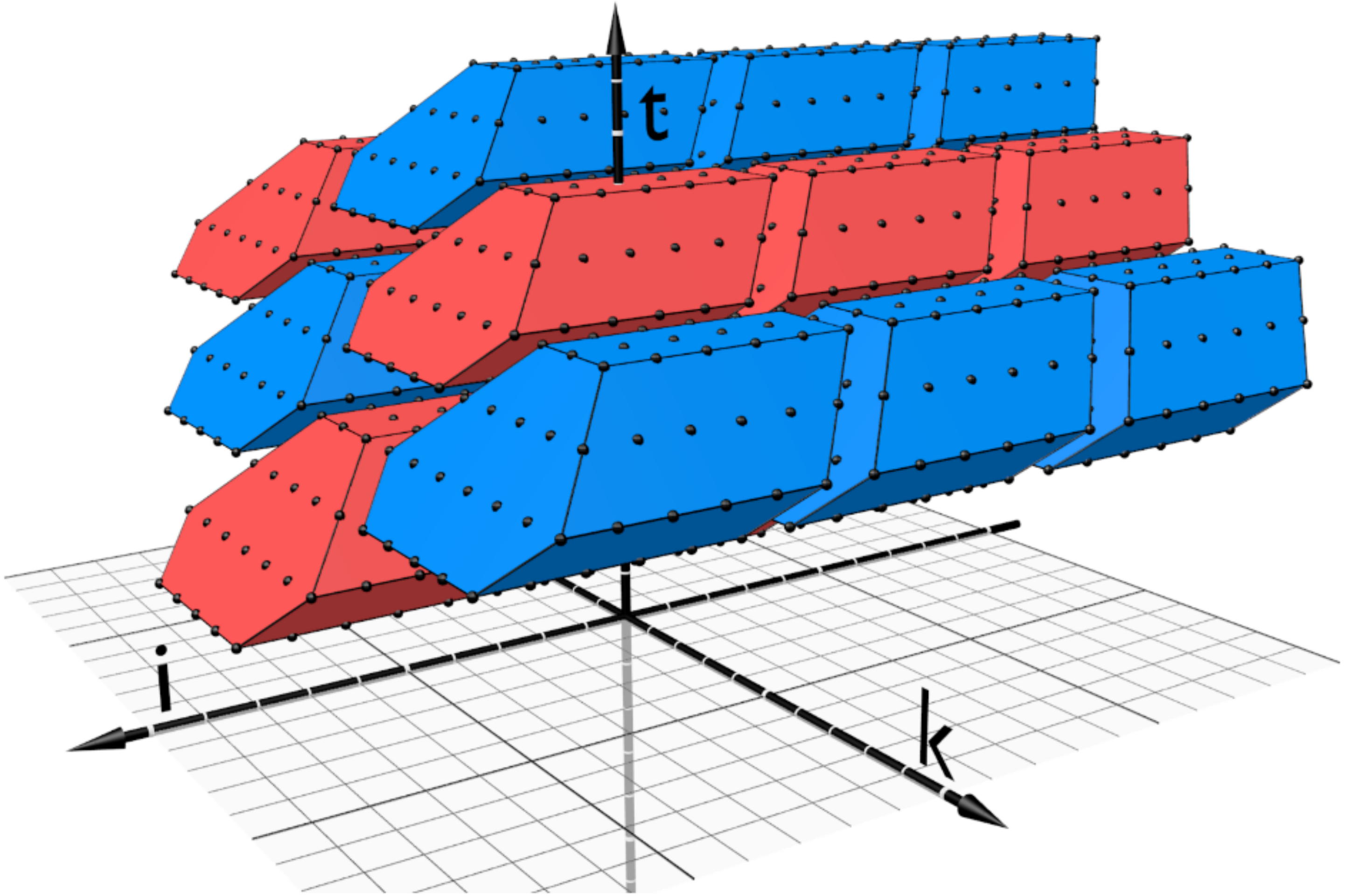
Jacobi Stencil with 1D Space + Time: Diamond Tiling



Advanced Tiling: A 2D Stencil

```
for (int t = 0; t < T; t++)
  for (int i = 0; i < N; i++)
    A[t+1][i][j] = A[t][i][j]
                 + A[t][i-1][j-1] + A[t][i-1][j+1]
                 + A[t][i+1][j-1] + A[t][i+1][j+1];
```

Hybrid Hexagonal/Parallelogram Tiling



AST Expression Generation

Piecewise Affine Expr.	AST Expression
$(i) \to (\lfloor i/4 \rfloor)$	$\to$ `floordiv(i, 4)`
$(i) \to (i \bmod 4)$	$\to$ `i - 4 * floordiv(i, 4)`

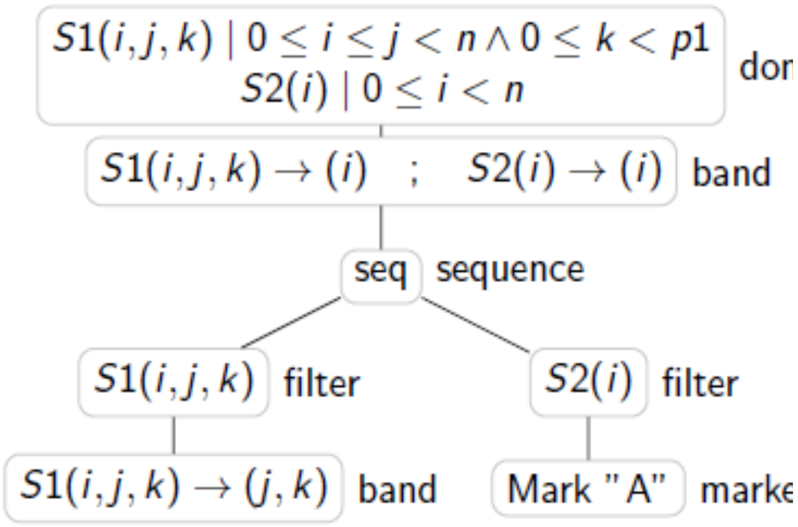
C implementation

```
#define floordiv(n, d) \
        (((n)<0) ? -((-(n)+(d)-1)/(d)) : (n)/(d))
```

Pw. Aff. Expr.	Context	AST Expression
$(i) \to (\lfloor i/4 \rfloor)$	$i \geq 0$	$\to$ `i / 4`
	$i \leq 0$	$\to$ `-((-i + 3) / 4)`
	$i \bmod 4 = 0$	$\to$ `i / 4`
$(i) \to (i \bmod 4)$	$i \geq 0$	$\to$ `i % 4`
	$i \leq 0$	$\to$ `-((-i + 3) % 4) + 3`

Schedule Trees

```
  for (i = 0; i < n; i++) {
    for (j = i; j < n; j++)
      for (k = 0; k < p1 ; k++)
S1:       A[i][j] = k * B[i]

    // Mark "A"
S2: A[i][i] = A[i][i] / B[i];
  }
```



Schedule Tree – Original Code

$S1(i,j,k) \mid 0 \leq i \leq j < n \wedge 0 \leq k < p1$ domain

$S1(i,j,k) \rightarrow (i,j,k)$ band

```
for (i = 0; i < n; i++)
  for (j = i; j < n; j++)
    for (k = 0; k < n ; k++)
S1:   S(i,j,k)
```

Schedule Tree – Tiled

$S1(i,j,k) \mid 0 \leq i \leq j < n \wedge 0 \leq k < p1$ domain

$S1(i,j,k) \rightarrow (\lfloor i/128 \rfloor, \lfloor j/128 \rfloor, \lfloor k/128 \rfloor)$ band

$S1(i,j,k) \rightarrow (i\%128, j\%128, k\%128)$ band

```
for (c0 = 0; c0 < n; c0 += 128)
 for (c1 = 0; c1 < n; c1 += 128)
  for (c2 = 0; c2 < n; c2 += 128)
   for (c3 = 0;
        c3 <= min(127, n - c0 - 1);
        c3 += 1)
    for (c4 = 0;
         c4 <= min(127, n - c1 - 1);
         c4 += 1)
     for (c5 = 0;
          c5 <= min(127, n - c2 - 1);
          c5 += 1)
       S1(c0 + c3, c1 + c4, c2 + c5)
```

Schedule Tree – Split Band

$S1(i,j,k) \mid 0 \leq i \leq j < n \wedge 0 \leq k < p1$ domain

$S1(i,j,k) \rightarrow (\lfloor i/128 \rfloor, \lfloor j/128 \rfloor, \lfloor k/128 \rfloor)$ band

$S1(i,j,k) \rightarrow (i\%128)$ band

$S1(i,j,k) \rightarrow (j\%128)$ band

$S1(i,j,k) \rightarrow (k\%128)$ band

```
for (c0 = 0; c0 < n; c0 += 128)
 for (c1 = 0; c1 < n; c1 += 128)
  for (c2 = 0; c2 < n; c2 += 128)
   for (c3 = 0;
        c3 <= min(127, n - c0 - 1);
        c3 += 1)
    for (c4 = 0;
         c4 <= min(127, n - c1 - 1);
         c4 += 1)
     for (c5 = 0;
          c5 <= min(127, n - c2 - 1);
          c5 += 1)
        S1(c0 + c3, c1 + c4, c2 + c5)
```

Schedule Tree – Strip-mine and Interchange

$S1(i,j,k) \mid 0 \leq i \leq j < n \wedge 0 \leq k < p1$ domain

$S1(i,j,k) \rightarrow (\lfloor i/128 \rfloor, \lfloor j/128 \rfloor, \lfloor k/128 \rfloor)$ band

$S1(i,j,k) \rightarrow (i\%128)$ band

$S1(i,j,k) \rightarrow (\lfloor (j\%128)/8 \rfloor)$ band

$S1(i,j,k) \rightarrow| (k\%128)$ band

$S1(i,j,k) \rightarrow (j\%8)$ band

```
[...]
  for (c3 = 0;
       c3 <= min(127, n - c0 - 1);
       c3 += 1)
   for (c4 = 0;
        c4 <= min(127, n - c1 - 1);
        c4 += 1)
    for (c5 = 0;
         c5 <= min(127, n - c2 - 1);
         c5 += 1)
     // SIMD Parallel Loop
     // at most 8 iterations
     for (c6 = 0;
          c6 <= min(7, n - c1 - c4 - 1);
          c6 += 1)
       S1(c0 + c3, c1 + c4 + c6, c2 + c5
```

Schedule Tree – Isolate

$S1(i,j,k) \mid 0 \leq i \leq j < n \wedge 0 \leq k < p1$ domain

$S1(i,j,k) \rightarrow (\lfloor i/128 \rfloor, \lfloor j/128 \rfloor, \lfloor k/128 \rfloor)$ band

$S1(i,j,k) \rightarrow (i\%128)$ band

$S1(i,j,k) \rightarrow (\lfloor (j\%128)/8 \rfloor)$
$\{isolate[[a,b,c,d] \rightarrow [e]] : b < \lfloor n/128 \rfloor\}$ band

$S1(i,j,k) \rightarrow (k\%128)$ band

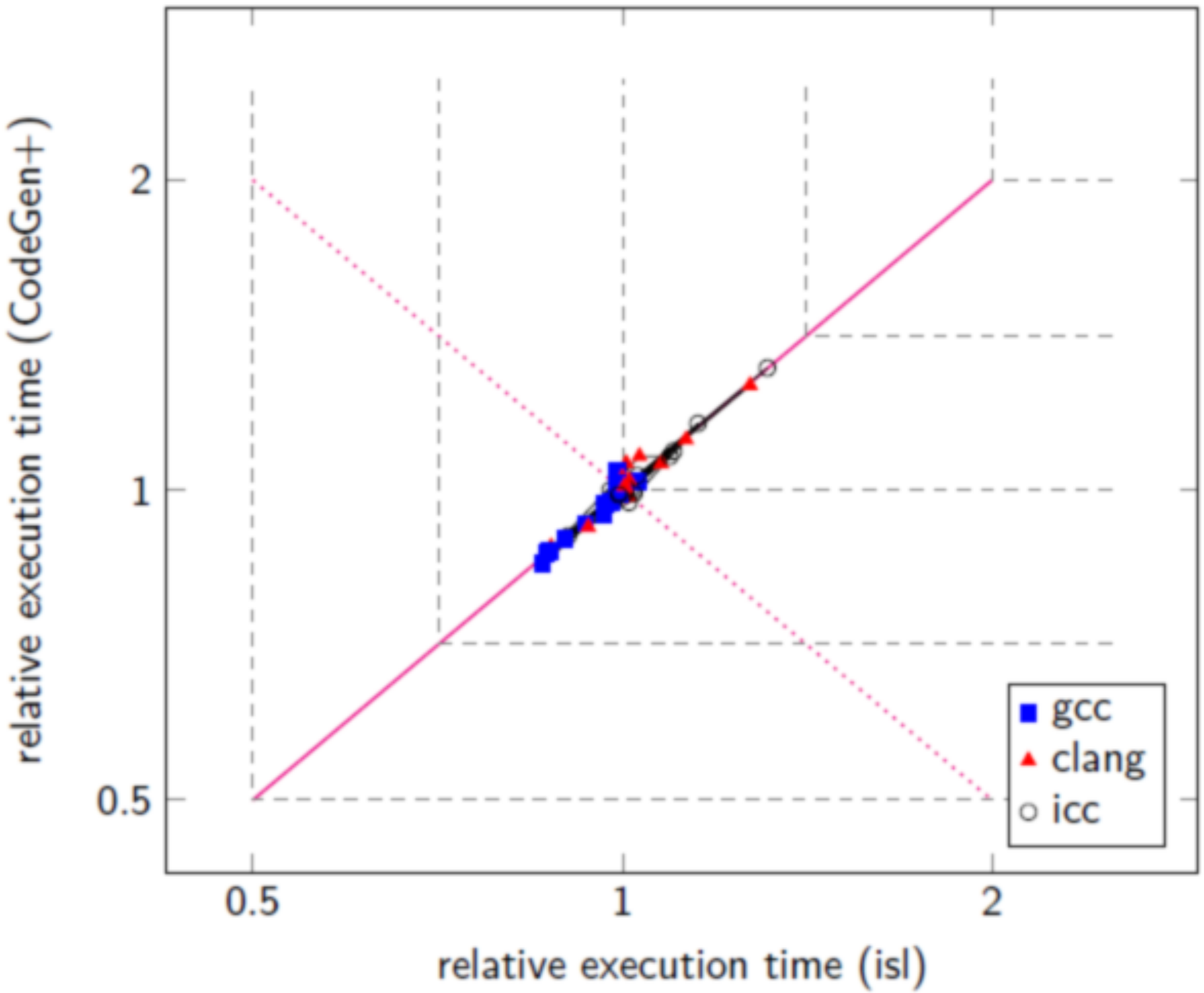
$S1(i,j,k) \rightarrow (j\%8)$ band

```
[...]
for (c3 = 0;
     c3 <= min(127, n - c0 - 1);
     c3 += 1)
 if (n >= 128 * c1 + 128) {
  for (c4 = 0; c4 <= 127; c4 += 8)
   for (c5 = 0;
        c5 <= min(127, n - c2 - 1); c5 +=

     // SIMD Parallel Loop
     // Exactly 8 Iterations
     for (c6 = 0; c6 <= 7; c6 += 1)
      S1(c0 + c3, c1 + c4 + c6, c2 + c5);

 } else {
   // Handle remainder
```

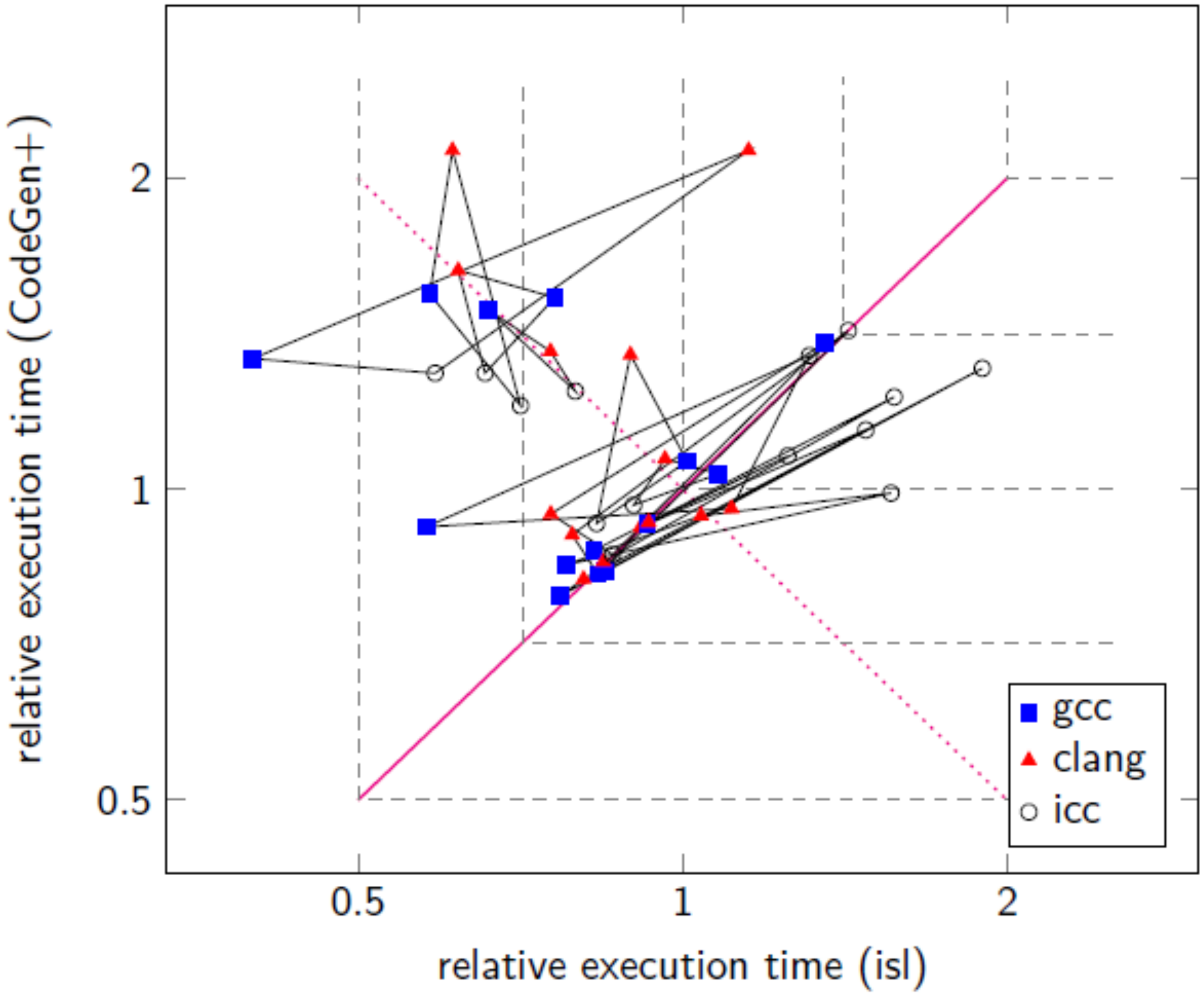
Evaluation

AST Generation

Generated Code Performance - Consistent



Generated Code Performance – Differing

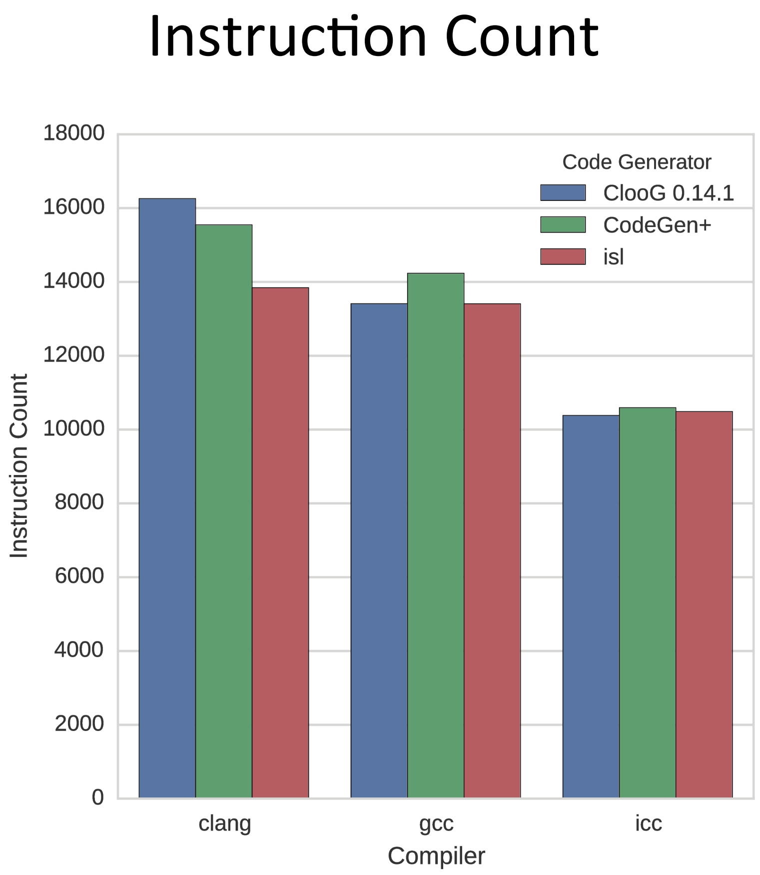


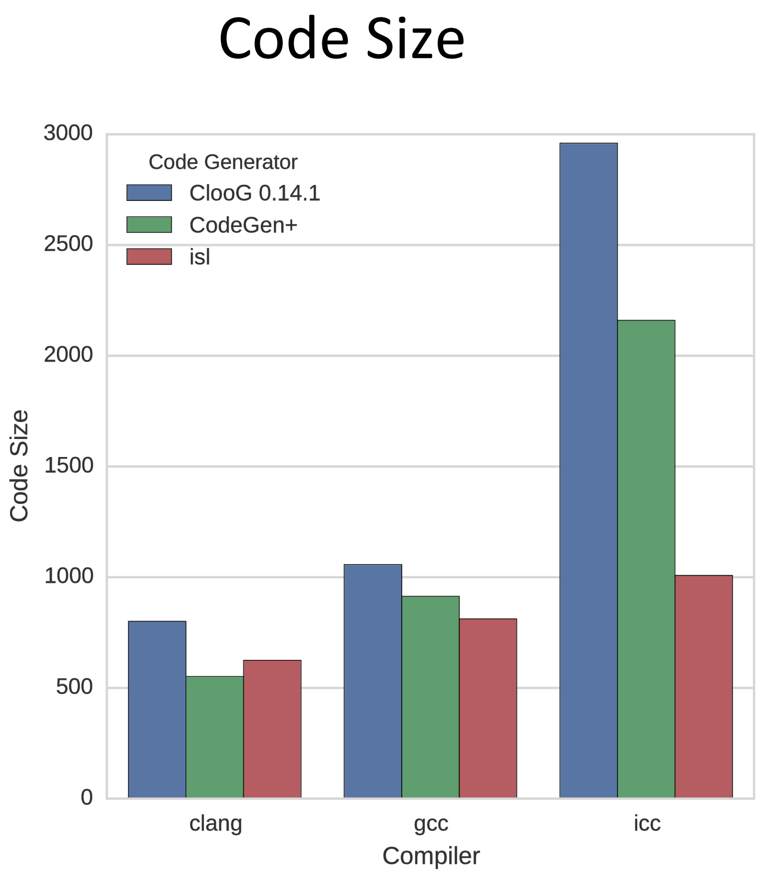
Code Quality: youcefn [Bastoul 2004]

CLooG 0.14.1

```
for(i=1; i<=n-2; i++) {
  S0(i,i);
  S1(i,i);
  for(j=i+1; j<=n-1; j++)
    S1(i,j);
  S1(i,n);
  S2(i,n);
}
S0(n-1,n-1);
S1(n-1,n-1);
S1(n-1,n);
S2(n-1,n);
S0(n,n);
S1(n,n);
S2(n,n);
for (i=n+1; i <= m; i++)
  S3(i,j);
```

Code Quality: youcefn [Bastoul 2004]





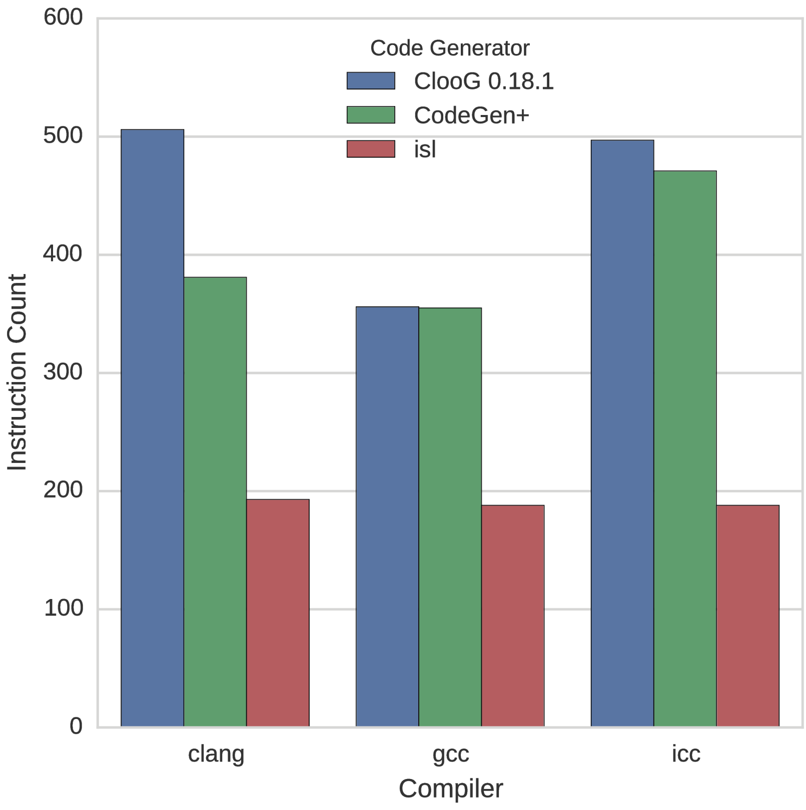
Code Quality: [Chen 2012] - Figure 8(b)

CLooG 0.18.1

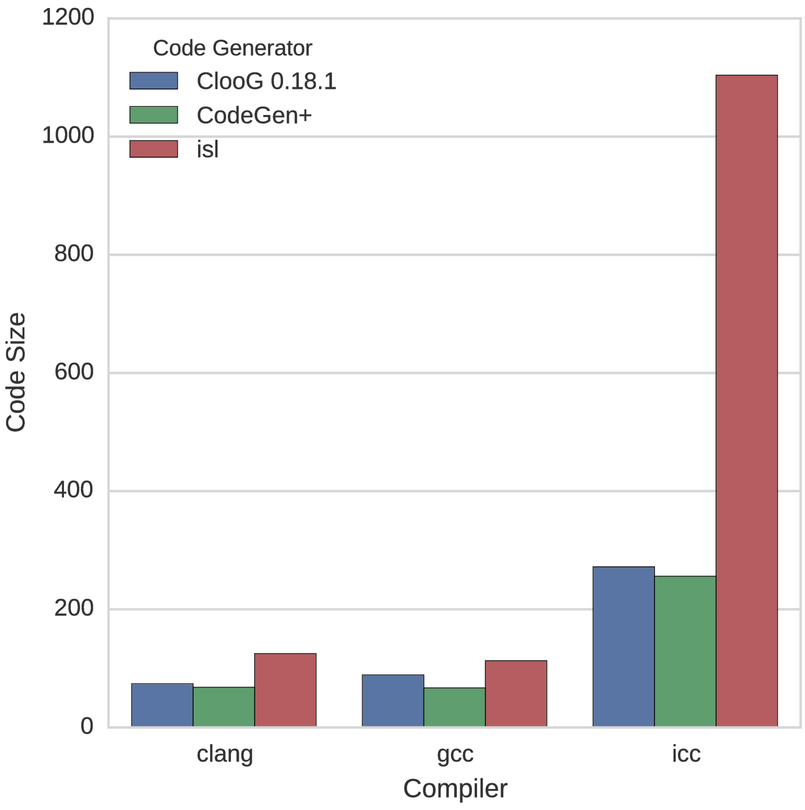
```
if (n >= 2)
  for (i = 2; i <= n; i += 2) {
    if (i%4 == 0)
      S0(i);
    if ((i+2)%4 == 0)
      S1(i);
  }
```

Code Quality: [Chen 2012] - Figure 8(b)

Instruction Count

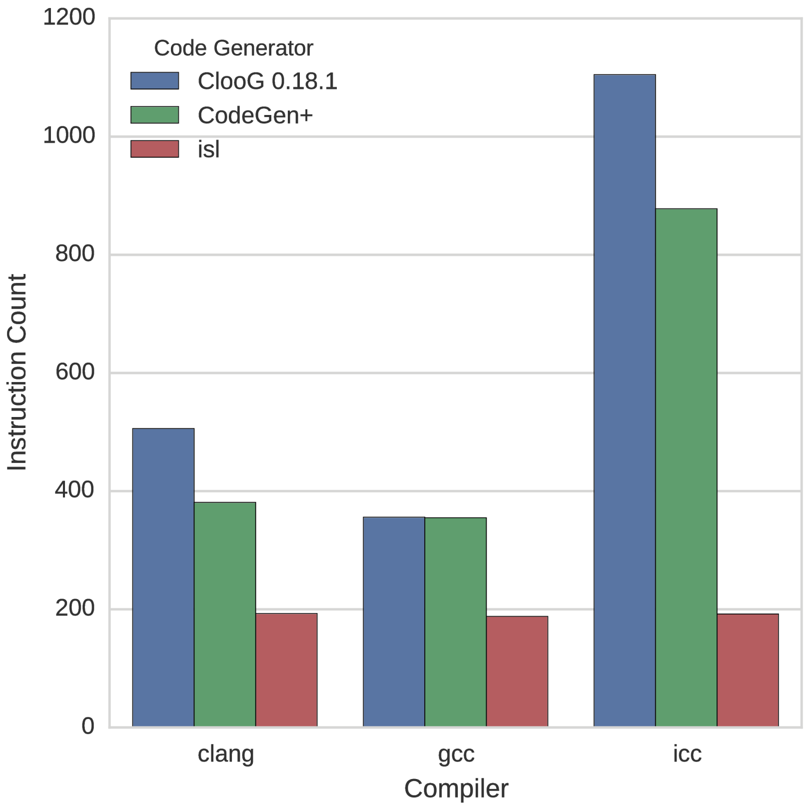


Code Size

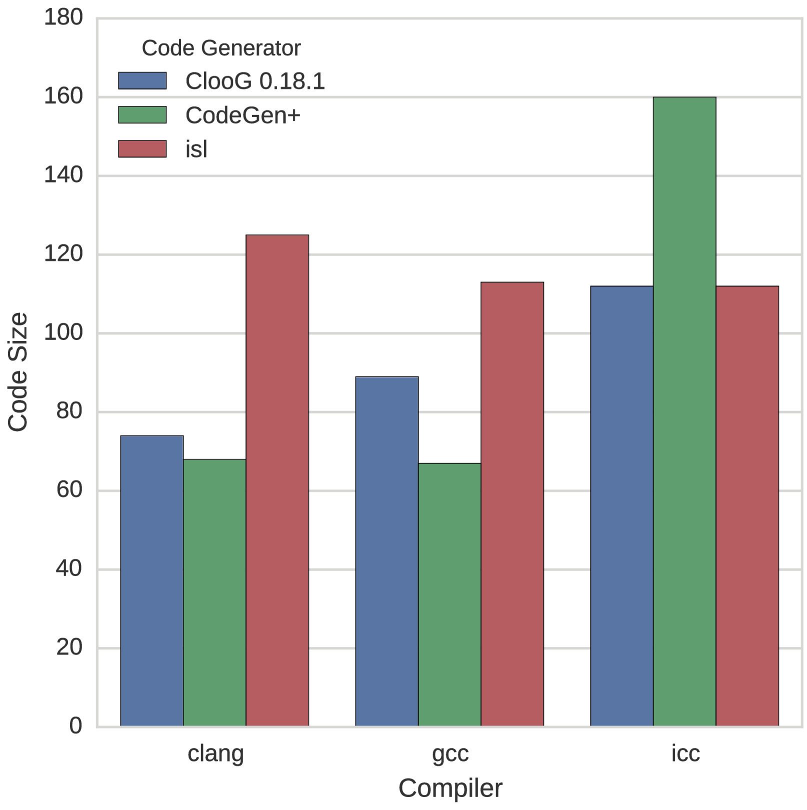


Code Quality: [Chen 2012] - Figure 8(b) novec/unroll

Instruction Count



Code Size

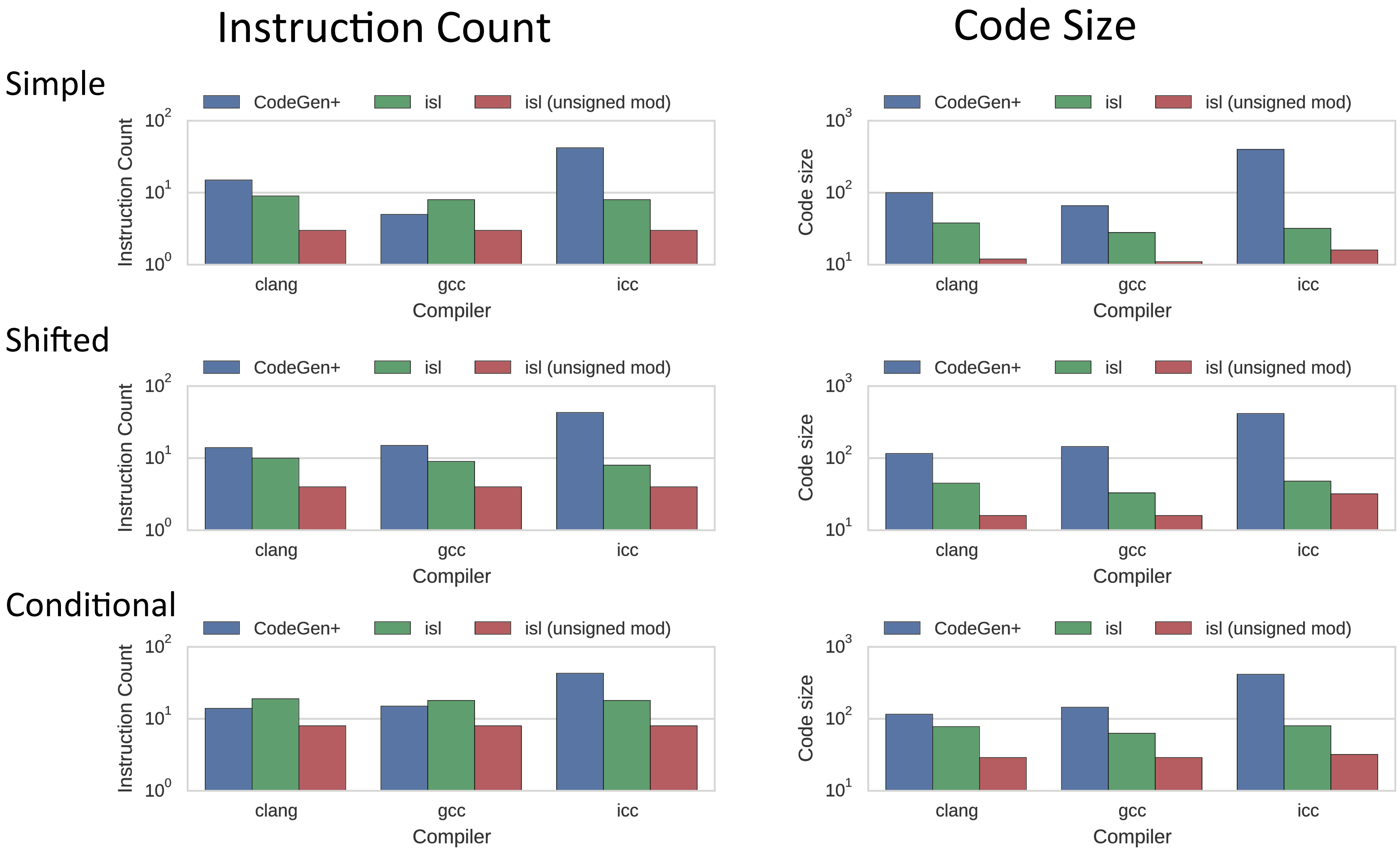


Modulo and Existentially Quantified Variables

CodeGen+

```
// Simple
for(i = intMod(n,128); i <= 127; i += 128)
  S(i);

// Shifted
for(i = 7+intMod(t1-7,128); i <= 134; i += 128)
  S(i);

// Conditional
for(i = 7+intMod(t1-7,128); i <= 130; i += 128)
  S(i);
```

Modulo and Existentially Quantified Variables



Polyhedral Unrolling

Normal loop code

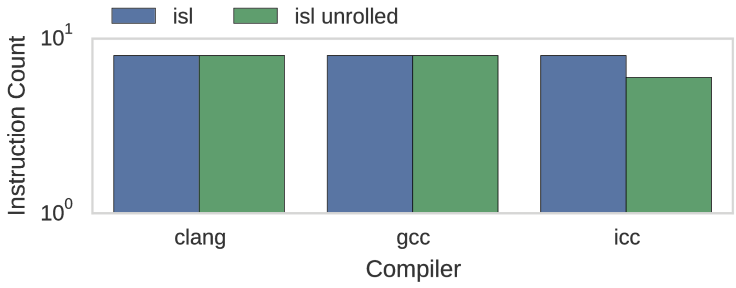
```
// Two e.q. variables
for (c0 = 0; c0 <= 7; c0 += 1)
  if (2 * (2 * c0 / 3) >= c0)
    S(c0);

// Multiple bounds
for (c0 = 0; c0 <= 1; c0 += 1)
  for (c1 = max(t1 - 384, t2 - 514);
       c1 < t1 - 255; c1 += 1)
    if (c1 + 256 == t1 ||
        (t1 >= 126 && t2 <= 255 &&
         c1 + 384 == t1) ||
        (t2 == 256 && c1 + 384 == t1))
      S(c0, c1);
```

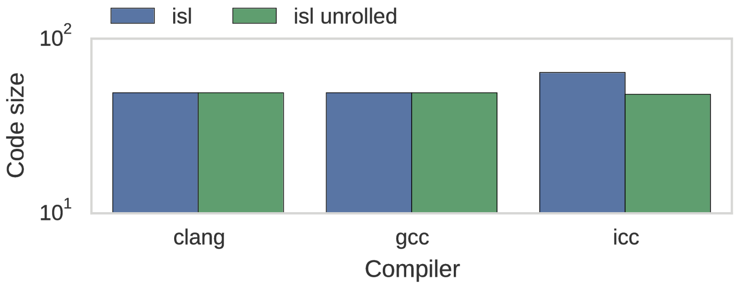
Polyhedral Unrolling

Instruction Count

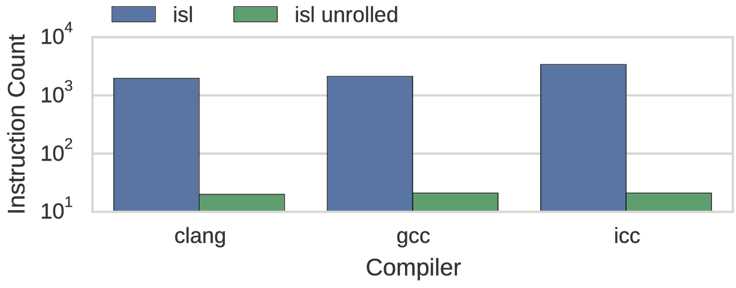
Code Size

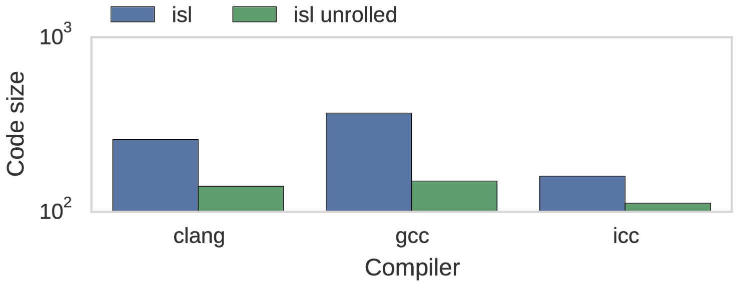
Two variables



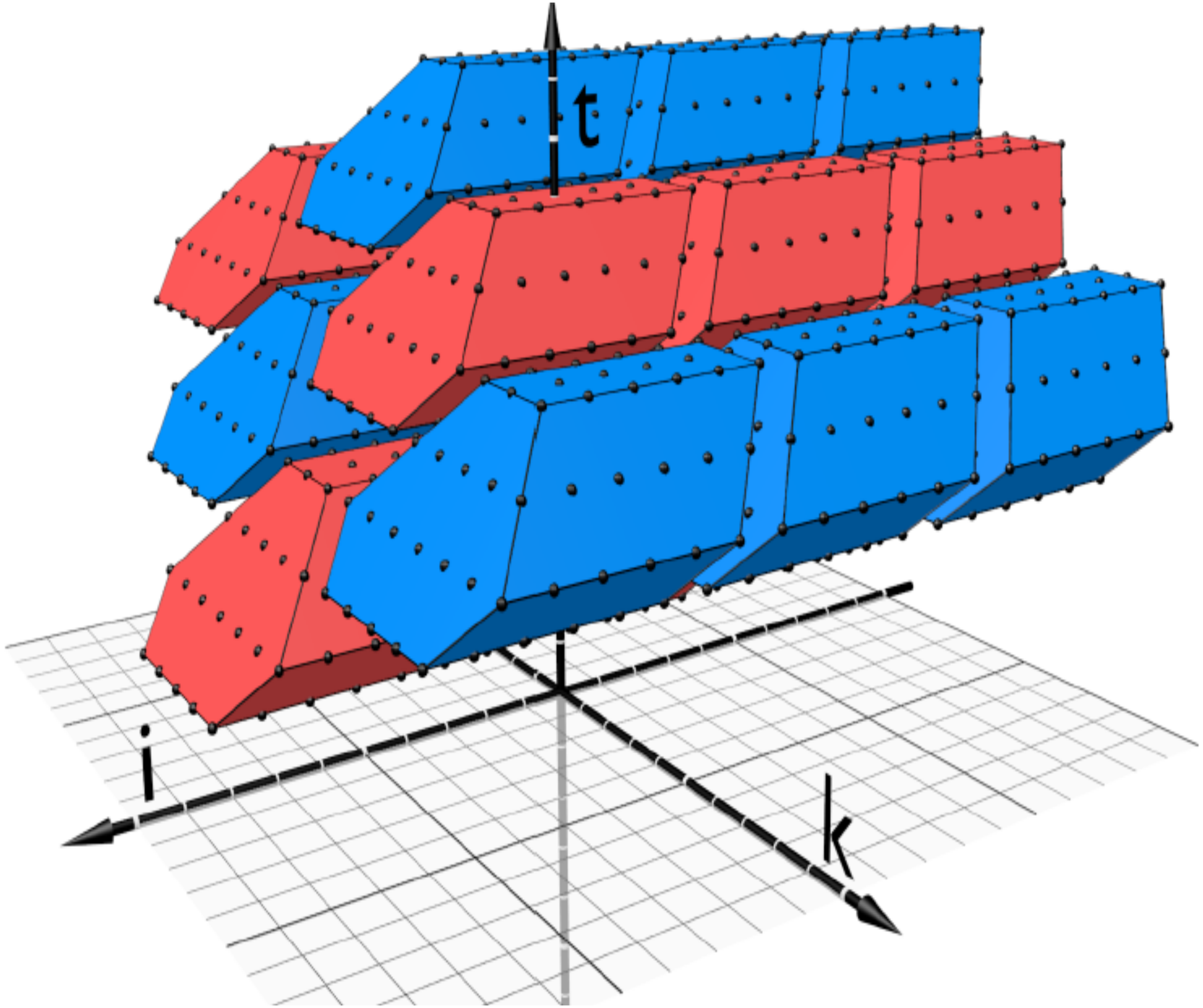


Multi Bound

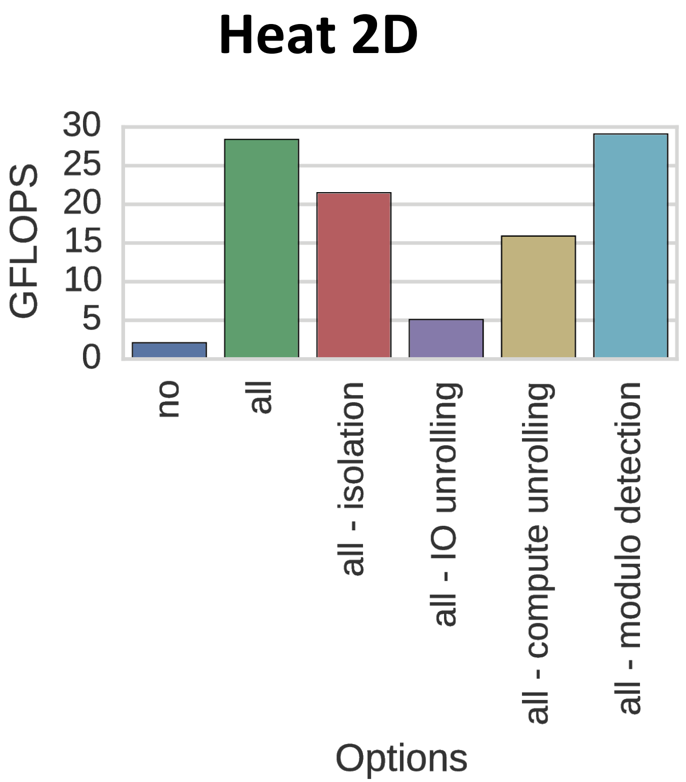


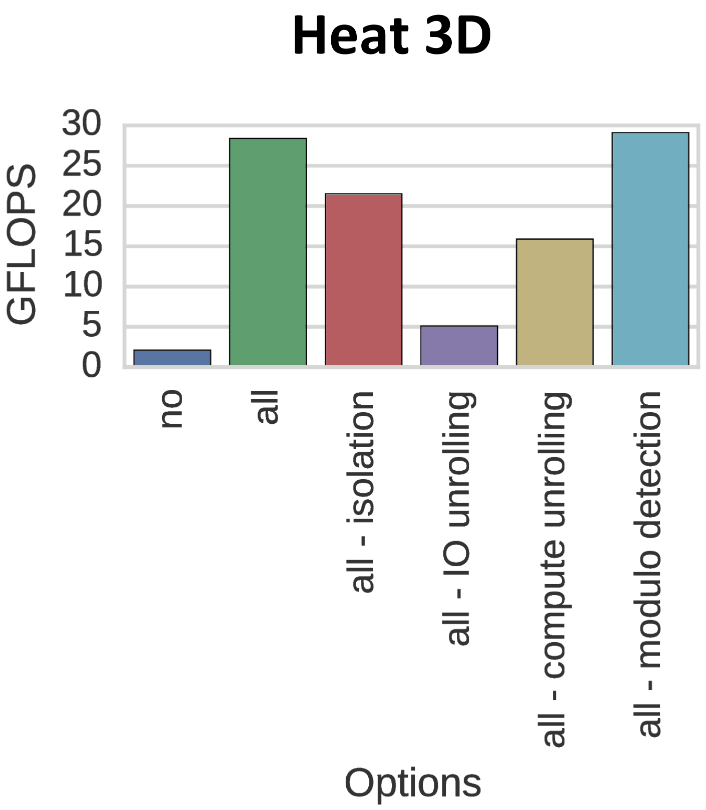


Hybrid Hexagonal Tiling for Stencil Programs



AST Generation Strategies for Hybrid-Hexagonal Tiling





Clearly beneficial loop interchange

```
void oddEvenCopy(int N, int M, float A[][M]) {
  for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++)
      A[2 * j][i] = A[2 * j + 1][i];
}
```

$\Rightarrow$ 15s

Assumption: Fixed size arrays do not overflow

```
void arrayOverflow(int N, float A[][20000]) {
  for (int i = 1; i < N; i++)
    for (int j = 1; j < M; j++) {
S1:    A[i][j-1] = ...;
S2:    A[i][j ] = ...;
S3:    A[i][j+1] = ...;
    }
}
```

Simplify Assumptions

- $A_{S1} := \forall i,j : 1 \leq i < N \wedge 1 \leq j < M \implies 0 \leq j - 1 < 20000$
- $A_{S2} := \forall i,j : 1 \leq i < N \wedge 1 \leq j < M \implies 0 \leq j < 20000$
- $A_{S3} := \forall i,j : 1 \leq i < N \wedge 1 \leq j < M \implies 0 \leq j + 1 < 20000$

Run-time check generation

- Set of constraints → AST expression
- Arbitrary Presburger Formula
- Implemented in a polyhedral code generator (as part of isl)

```
void arrayOverflow(int N, float A[][20000]) {
  if (M <= 19999) {
    // optimized code
  } else {
    // original code
  }
}
```

Optimistic Delinearization

```
void copyOddEven(int N, float *Ptr) {

#define A(x, y) Ptr[(x) * N + (y)]
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      A(2 * j, i) = A(2 * j + 1, i);
}
```

Tobias Grosser, Sebastian Pop, Louis-Noël Pouchet, P. Sadayappan, and Sebastian Pop
Optimistic delinearization of parametrically sized arrays, ICS 2015

Optimistic Exceptions Elimination

```
void copy(float A[][100], float B[][100],
  int DebugLevel, int N) {
  for (int i = 0; i < N; i++)
    for (int j = 0; j < 100; j++)
S1:    A[j][i] = B[j][i];
    if (DebugLevel > 5)
S2:    printf("Column %d copied\n", i);
  }
}
```

Optimistic Loop Invariant Code Motion

```
void copy(struct Array A) {

int tmp0, tmp1;
tmp0 = size0(A);

if (tmp0 > 0)
  tmp1 = size1(A);

for (int i = 0; i < tmp0; i++)
  for (int j = 0; j < tmp1; j++)
    S1:      access(A,j,i) += ..;
}
```

Integer Overflow

```
void overflow(unsigned n, unsigned m, float A[]) {
    for (unsigned i = 0; i < n; i++) {
      A[i] = 0;
    }
    for (unsigned i = 0; i < n + m; i++) {
      A[i] = 1;
    }


}
```

CGO'17: Optimistic Loop Optimization
(with Johannes Doerfert und Sebastian Hack)