

Generation of Fast and Parallel Code in LLVM

Tobias Grosser

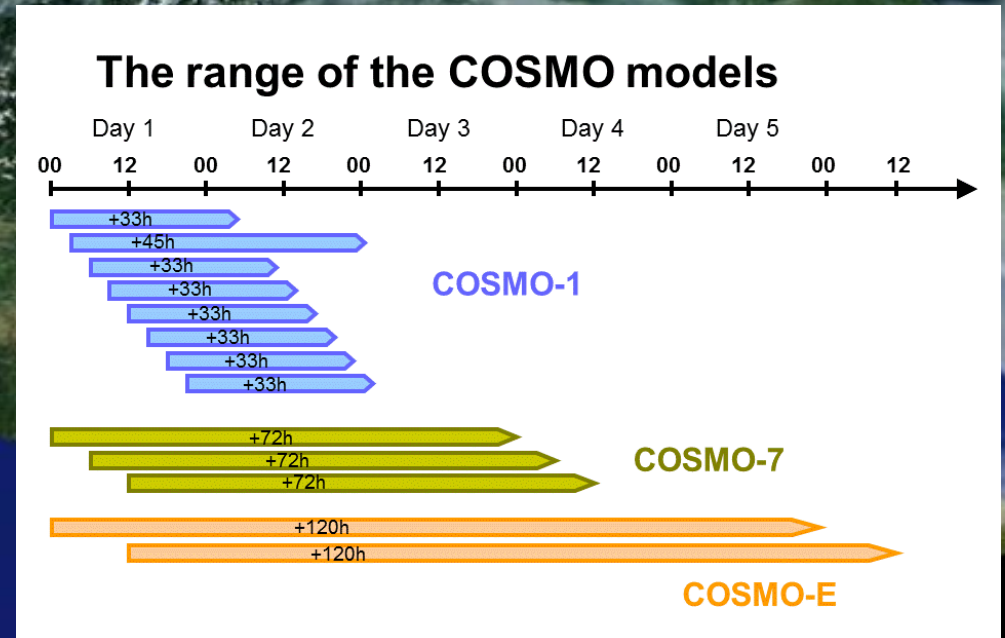


**LLVM and Clang Summer School
Paris, June 2017**

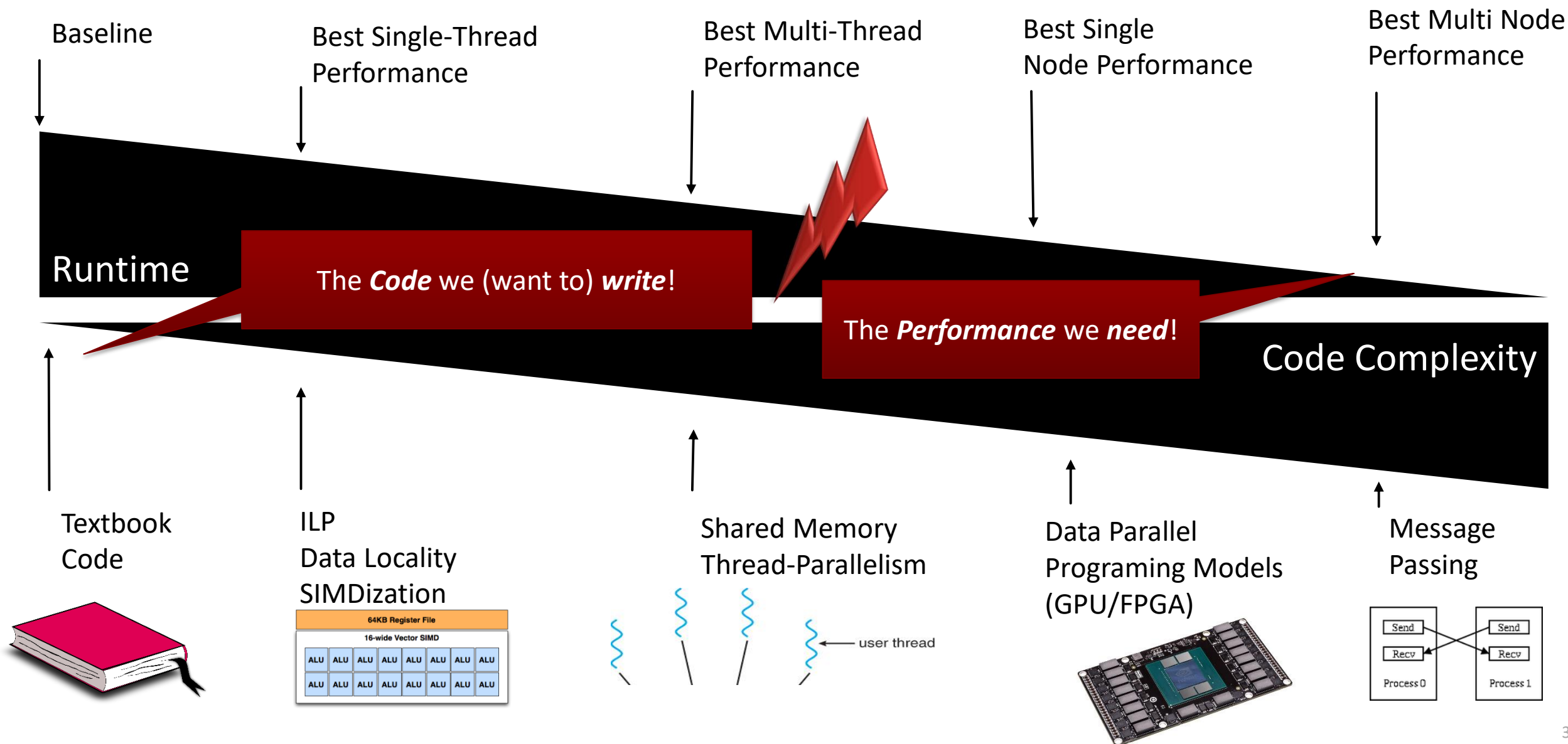
COSMO: Weather Prediction in Switzerland

Running Large Programs
in Parallel is Challenging

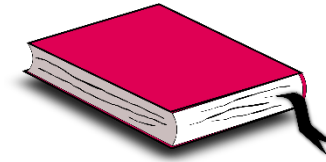
- > 500,000 Lines Code
- > 15,000 Loops
- 12 nodes + 192 GPUs
@CSCS Lugano



Performance vs. Code Complexity



GEMM: Generalized Matrix Multiplication

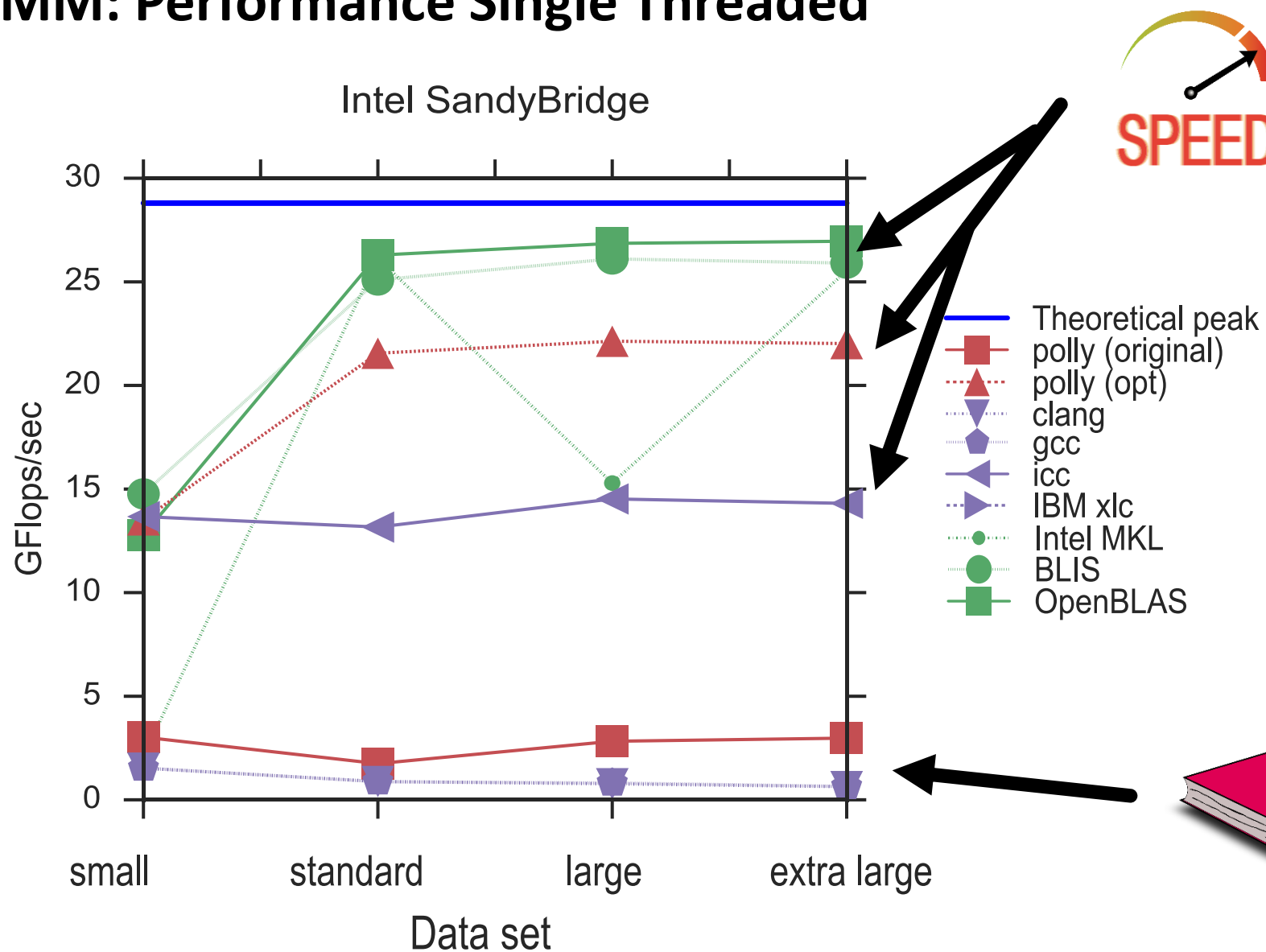


*The Simple
Textbook Version*

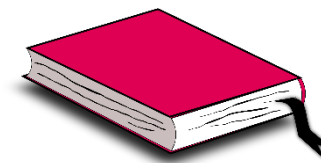
$$C = A \times B$$

```
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



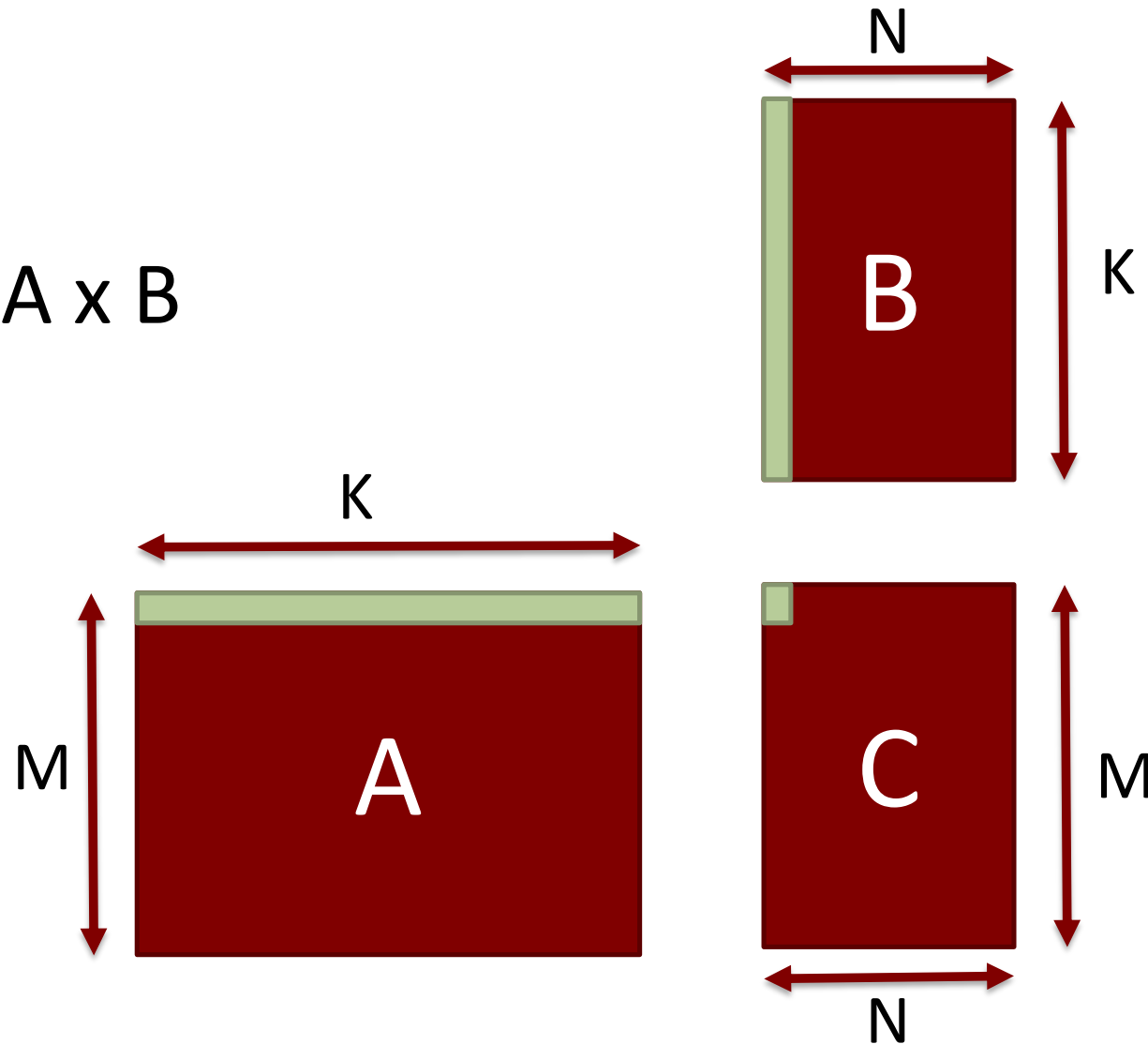
Optimized Codes



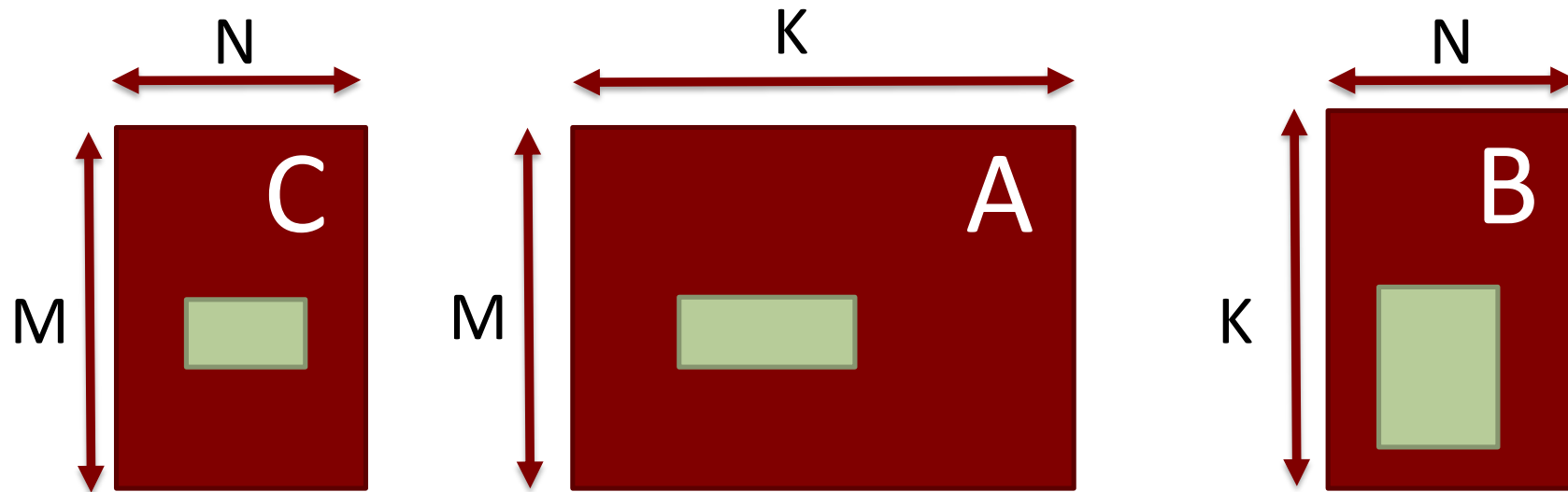
The Simple Textbook Version

GEMM: Computing on Micro Panels

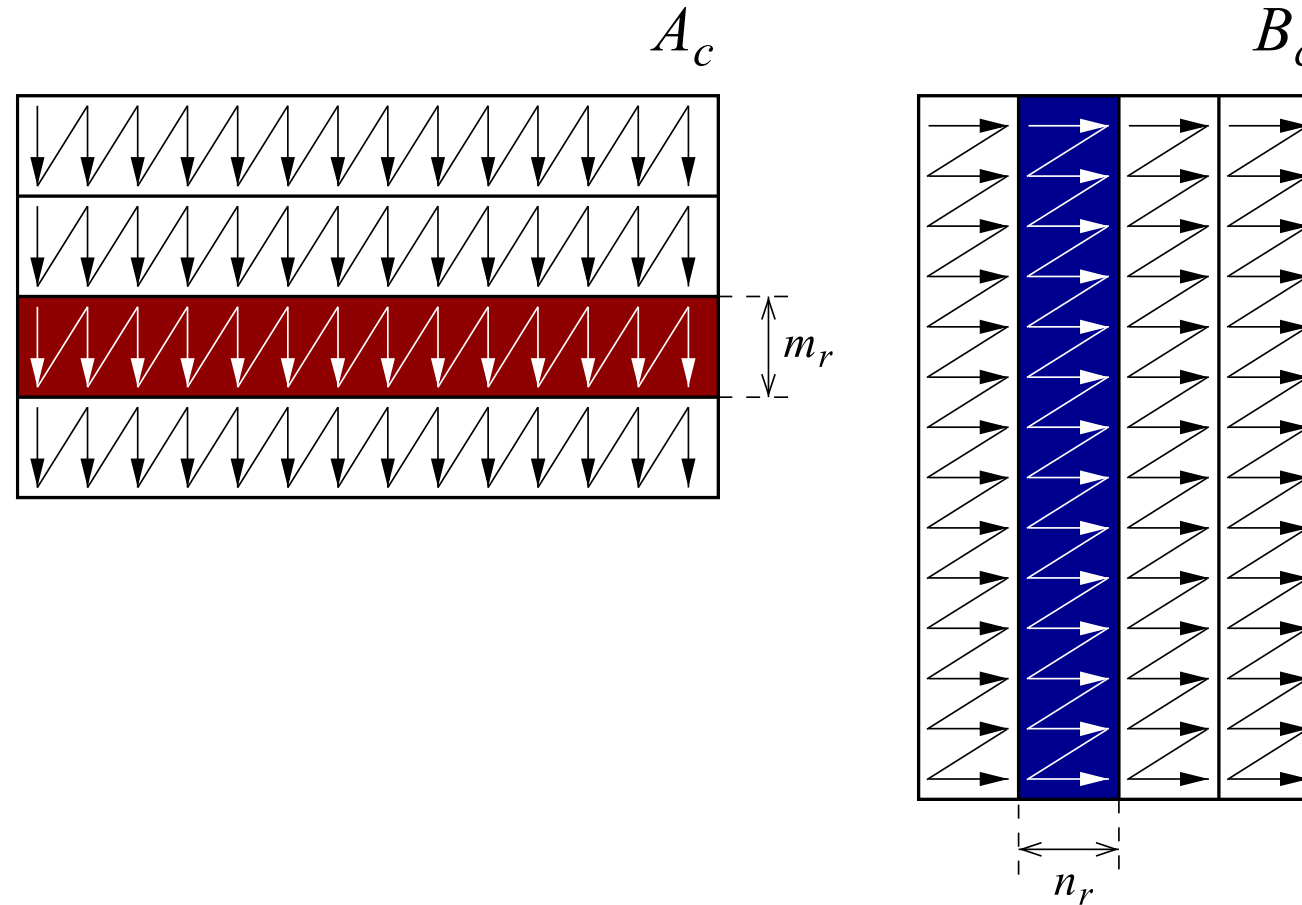
$$C = A \times B$$



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

```

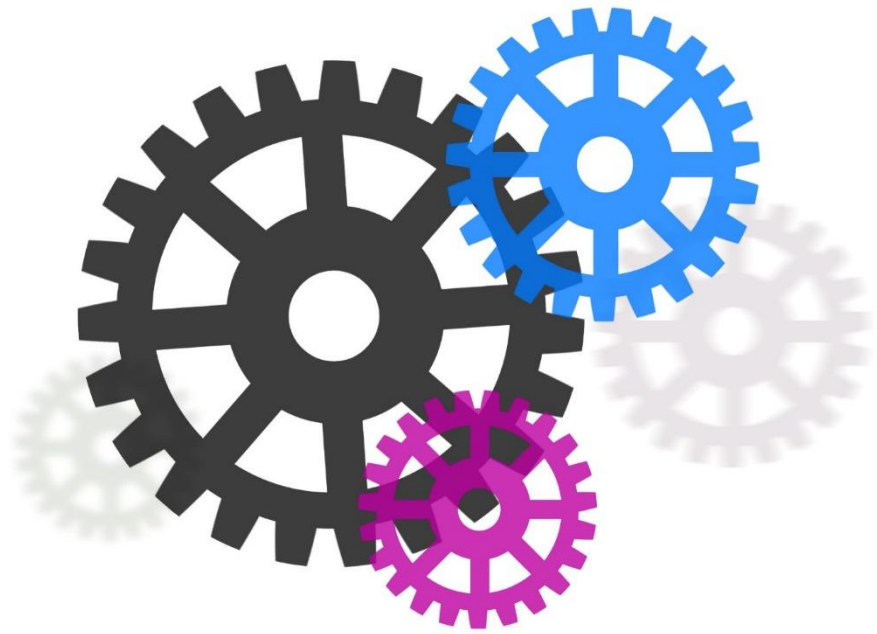
L1: for jc = 0,...,n-1 in steps of nc
L2:   for pc = 0,...,k-1 in steps of kc
      B(pc : pc + kc -1,jc : jc + nc -1) → Bc // Pack into Bc
L3:   for ic = 0,...,m-1 in steps of mc
      A(ic : ic + mc -1,pc : pc + kc -1) → Ac // Pack into Ac
L4:   for jr = 0,...,nc -1 in steps of nr // Macro-kernel
L5: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

Data Layout
Transformation

Loop
Blocking

SIMD Instructions

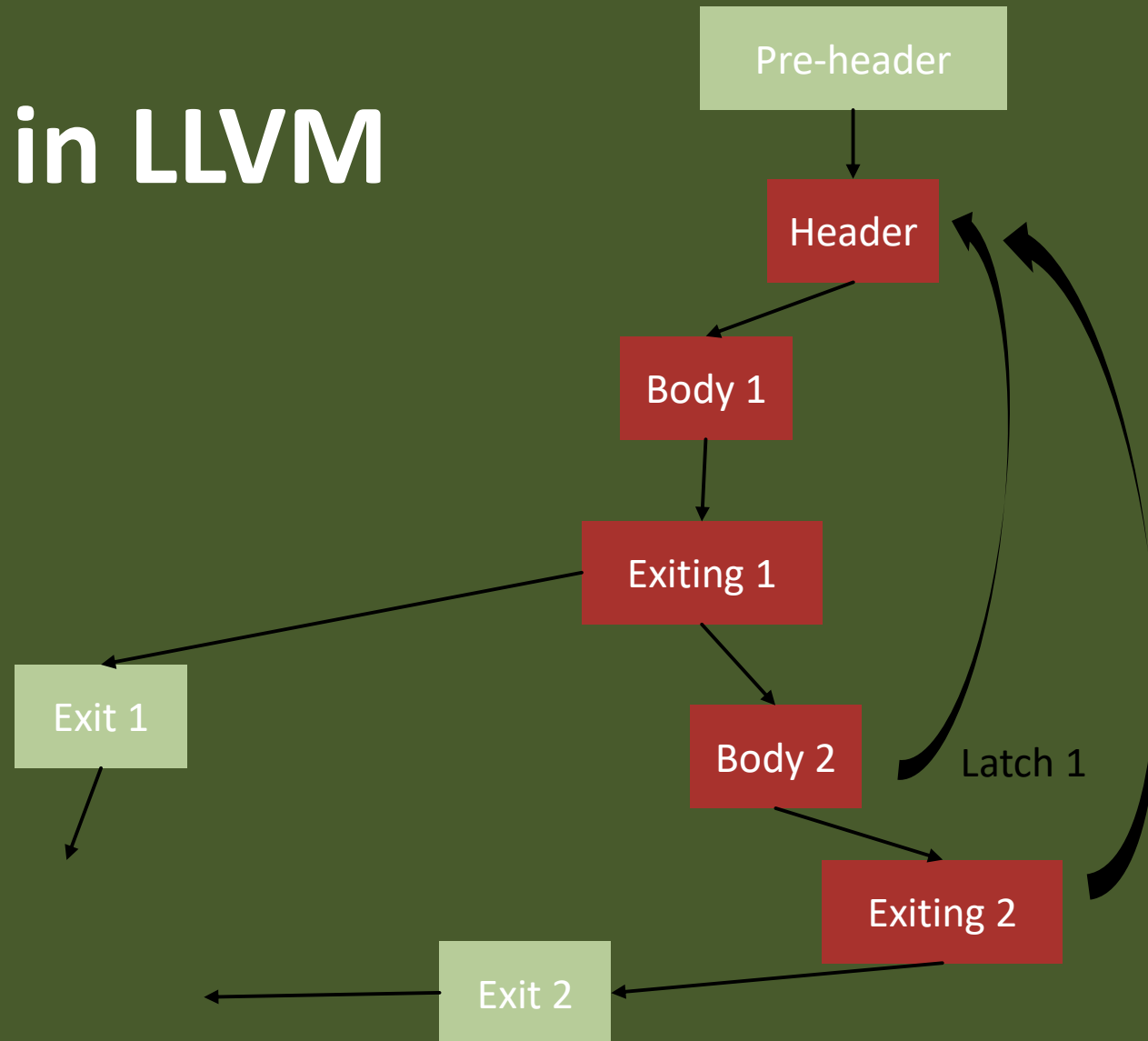
BLIS: A Framework for Rapidly Instantiating BLAS Functionality
 FIELD G. VAN ZEE and ROBERT A. VAN DE GEIJN



Parallel Code Generation

Which facilities does LLVM provide?

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

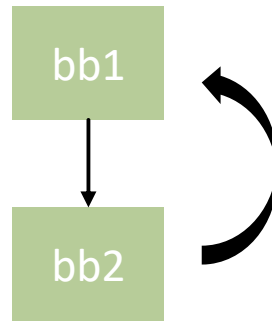
Analysis

- Information **can be derived** from IR directly
- + Must be recomputed
- Never outdated

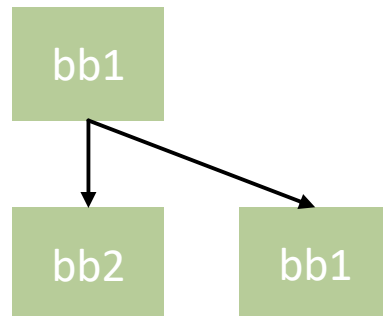
Preferred!

Analysis Passes in LLVM

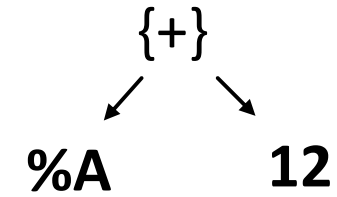
Loops



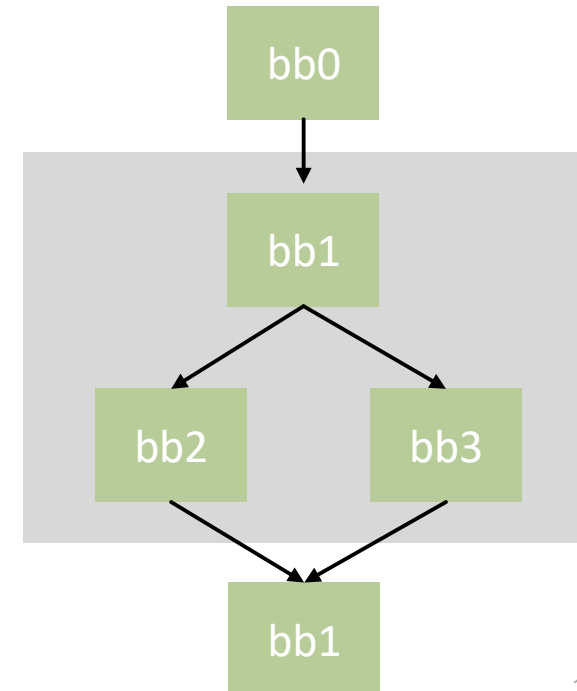
(Post) Dominance



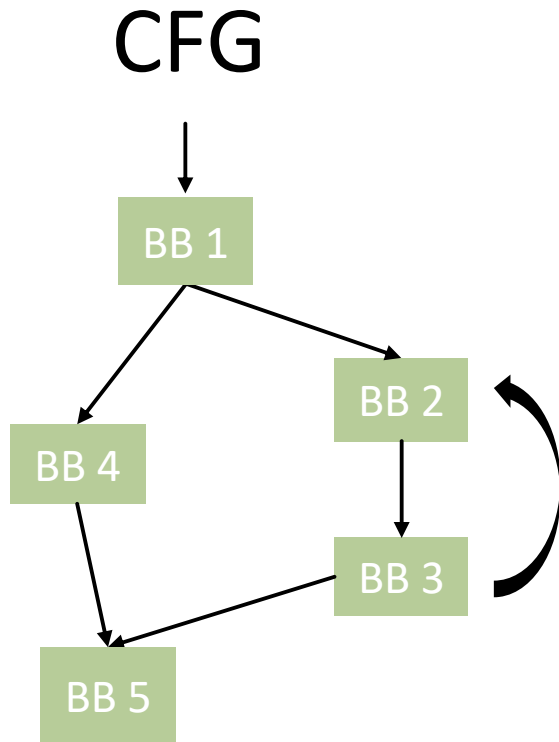
Scalar Evolution



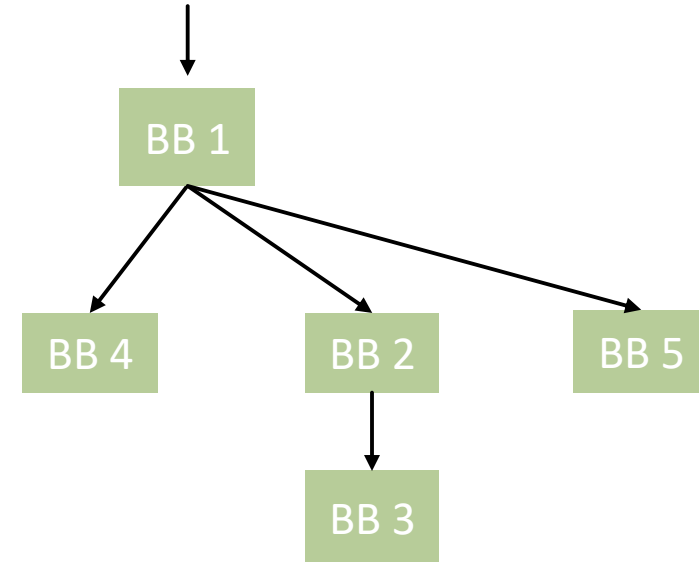
Regions



Dominance



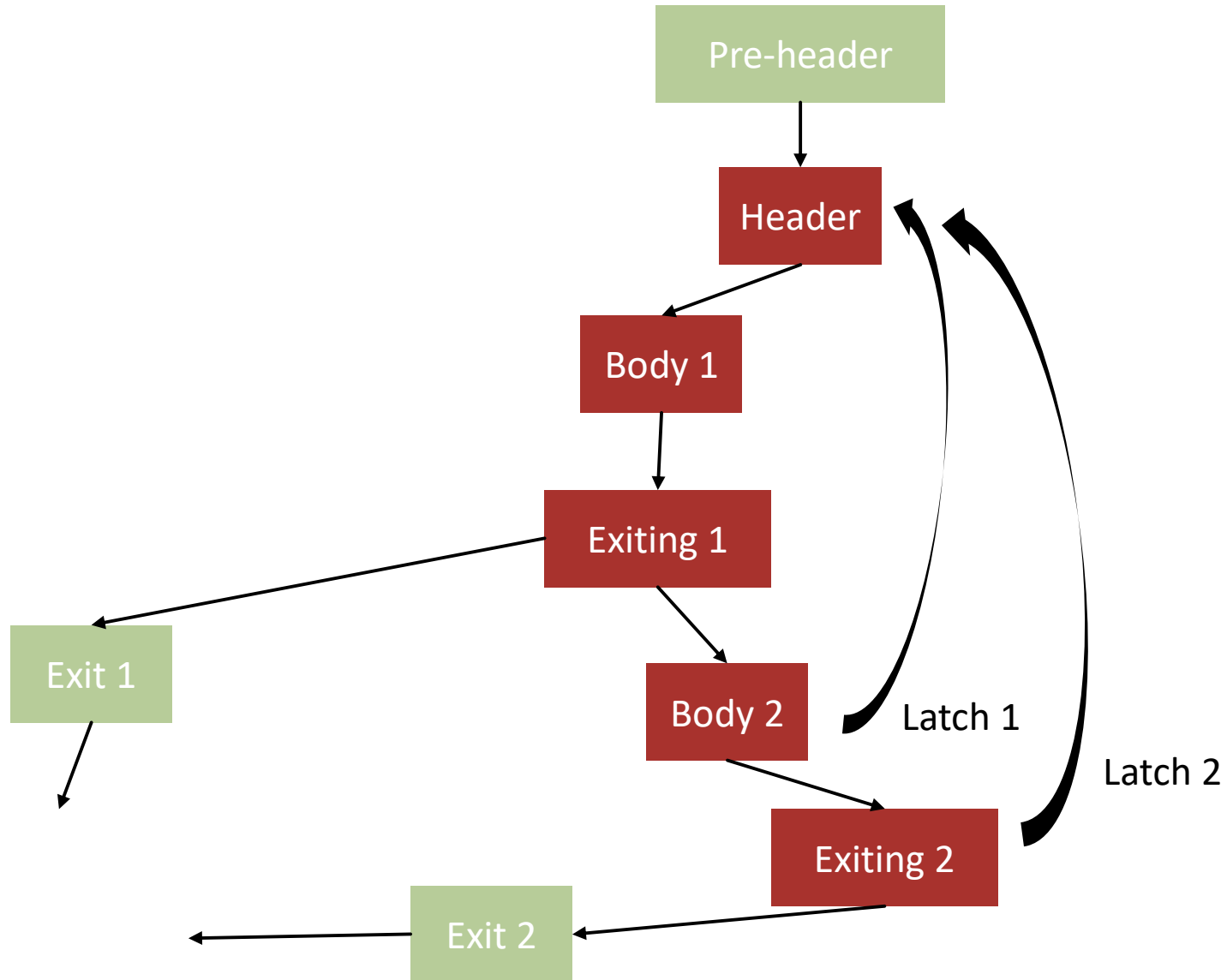
Dominator Tree



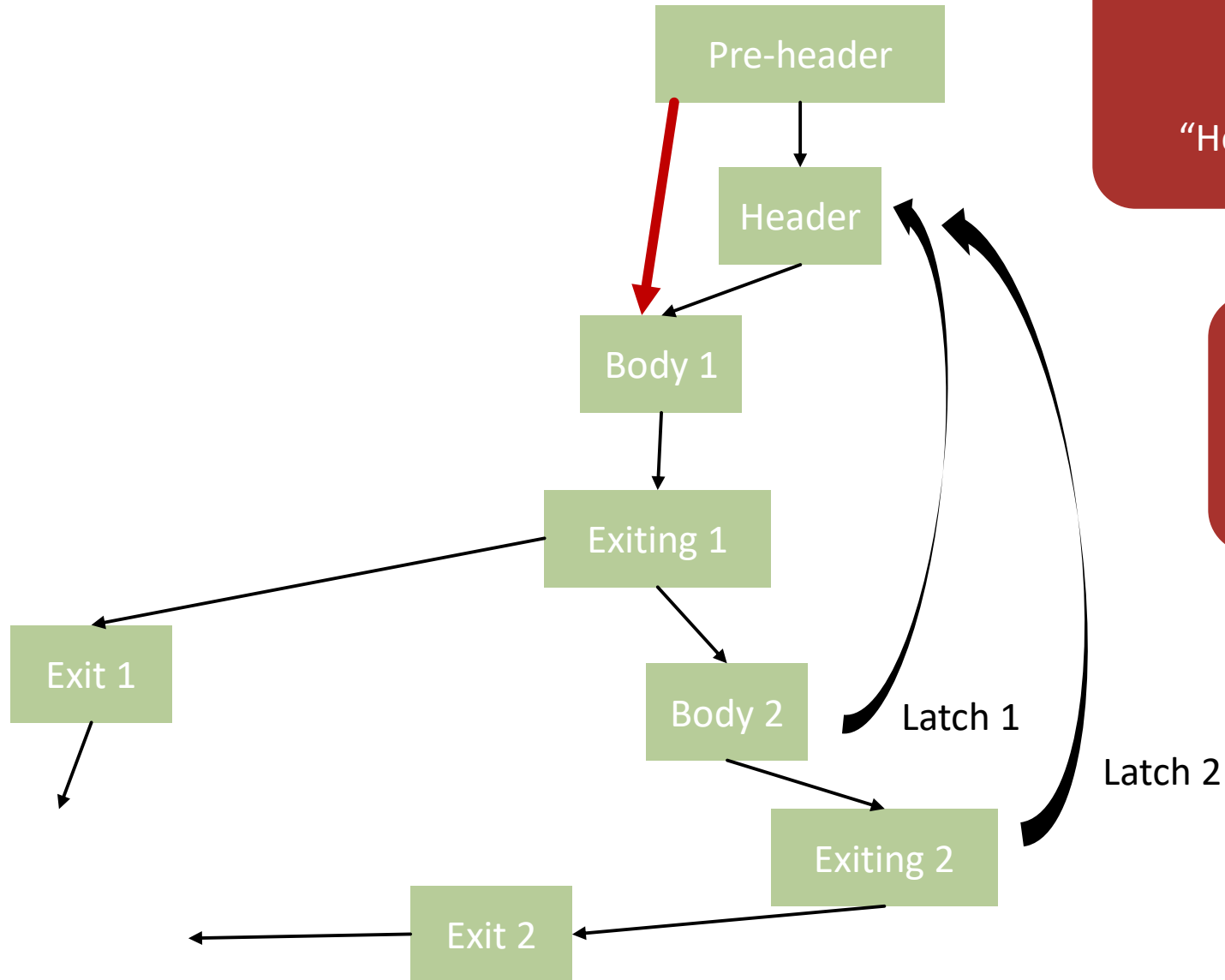
A **dominates** B, if each path from the entry to B contains A.

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

Loop Info: Detect Natural Loops



Loop Info: Detect Natural Loops

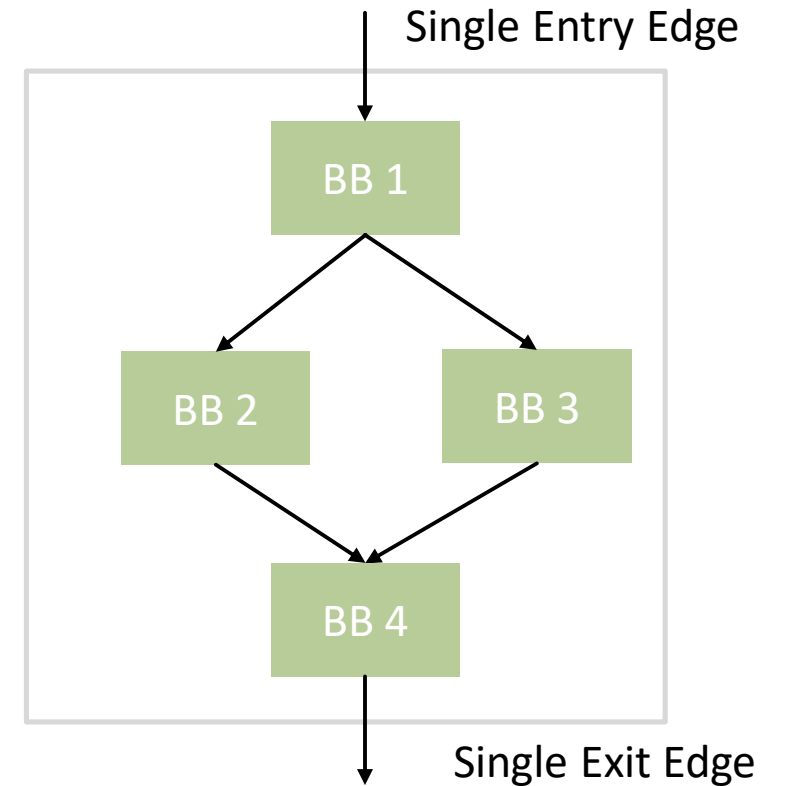
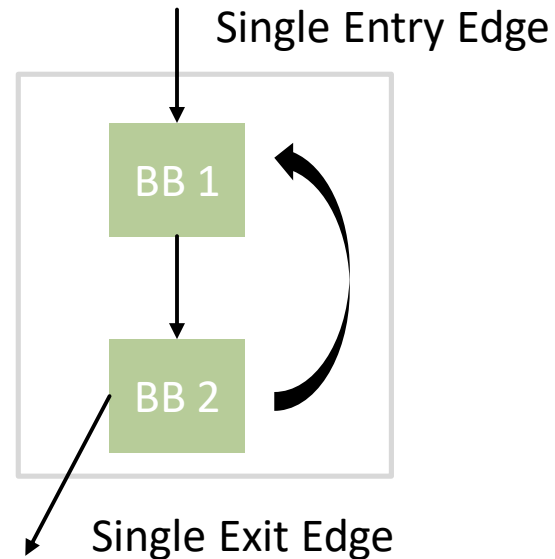
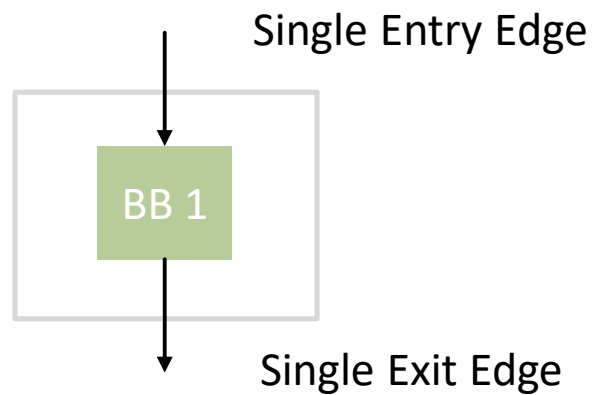


No Natural Loop!

“Header” does not dominate latches.

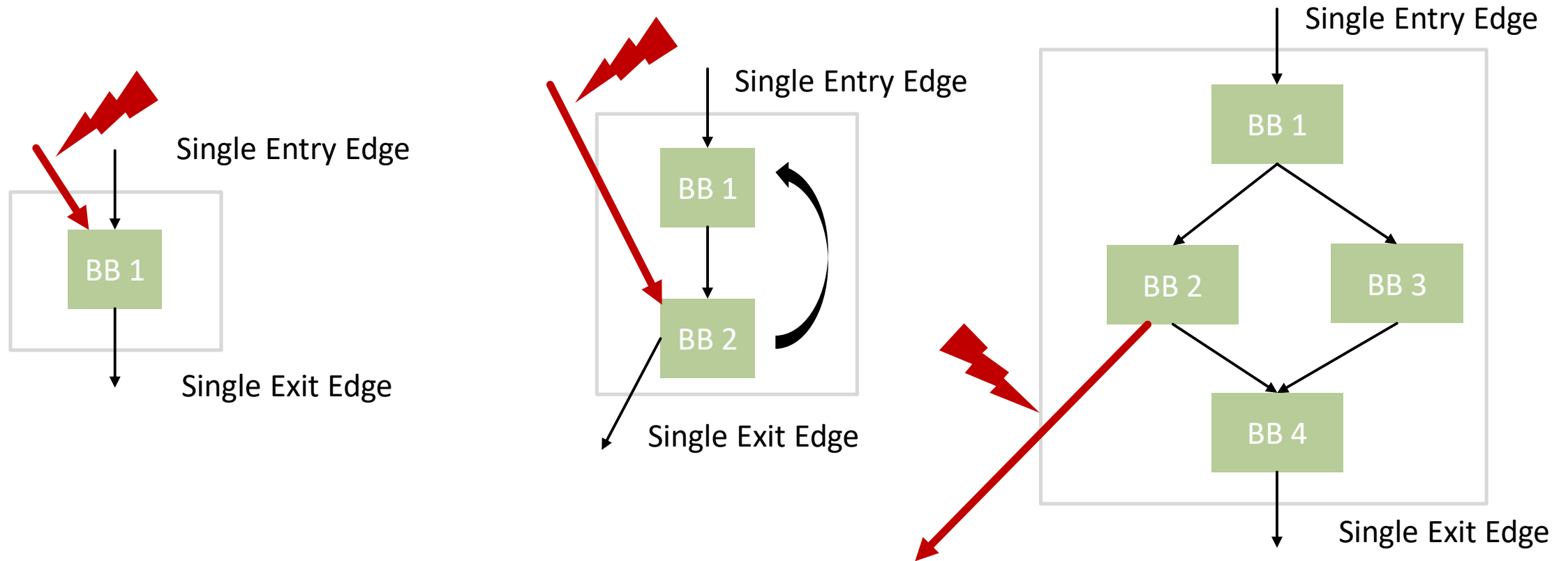
LLVM *does not* model this loop!

Region Info: Single Entry Single Exit Regions



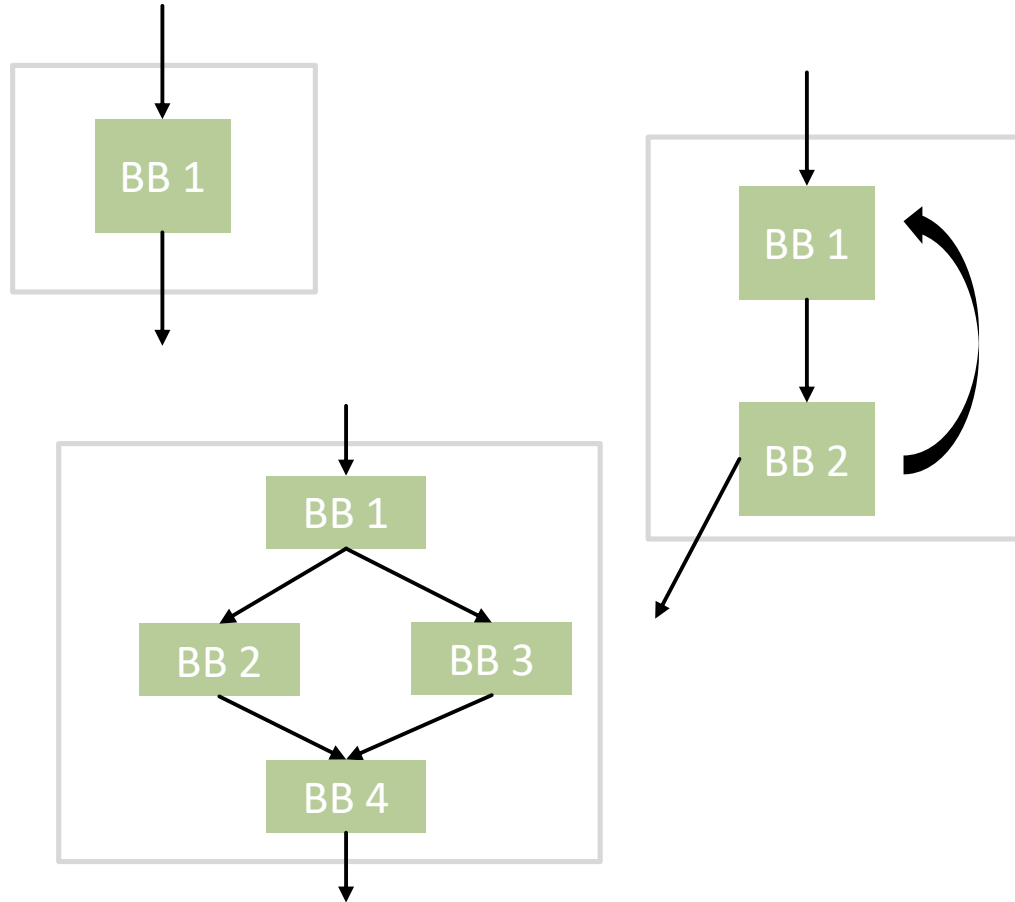
A **simple region** is a subgraph of the CFG with a single entry and a single exit edge.

Region Info: No Regions

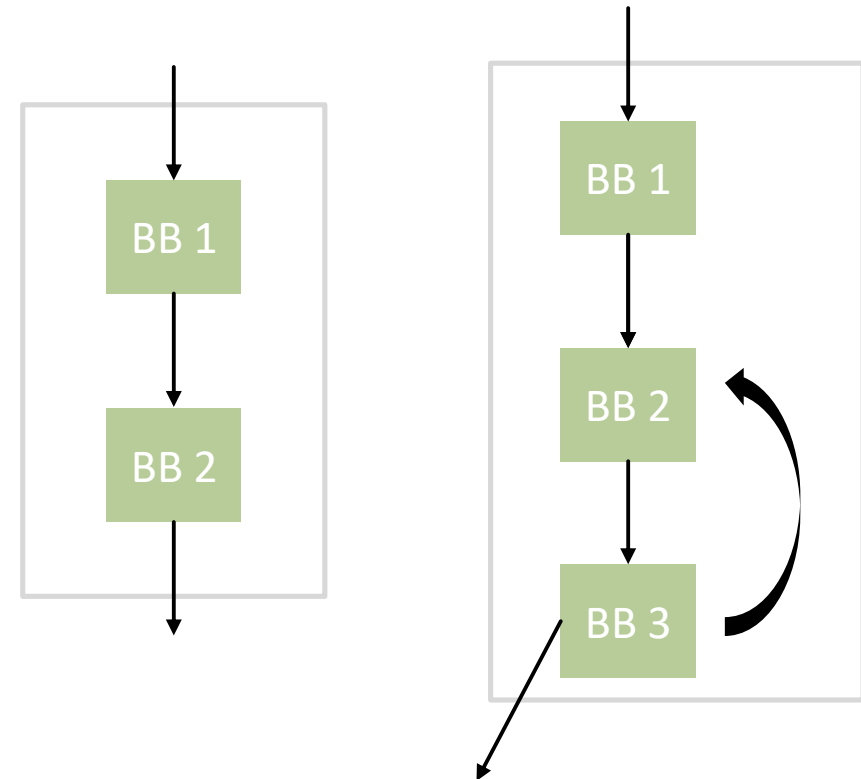


Region Info: Single Entry Single Exit Regions

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

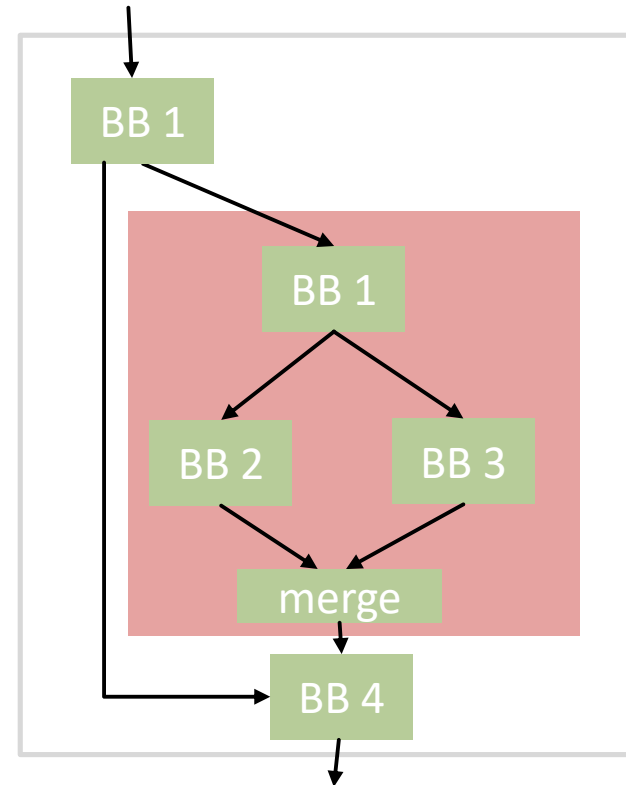
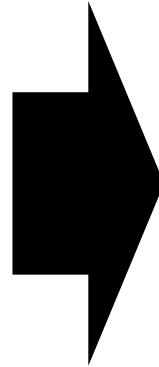
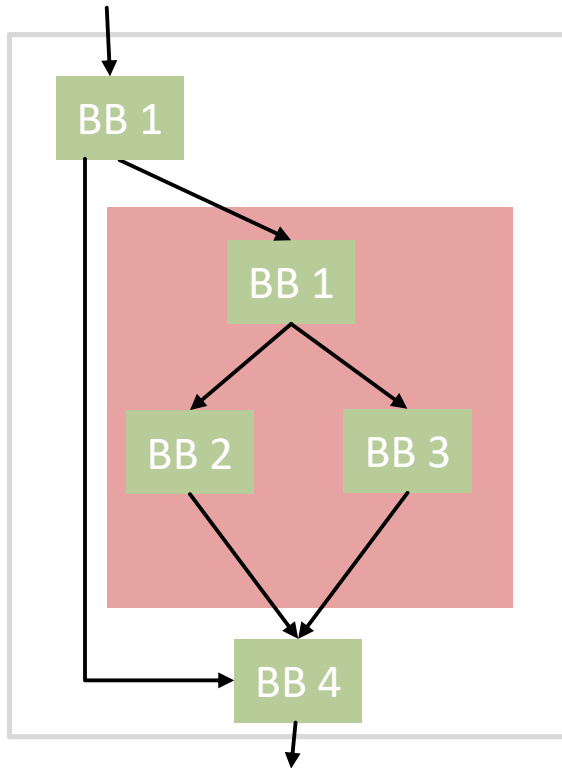


Canonical Regions



Non-Canonical

Region Info: Refined Regions

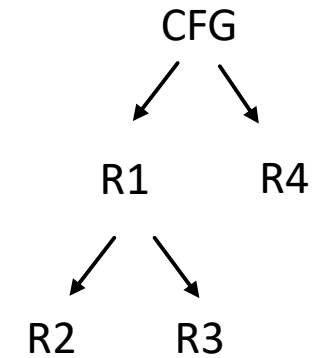
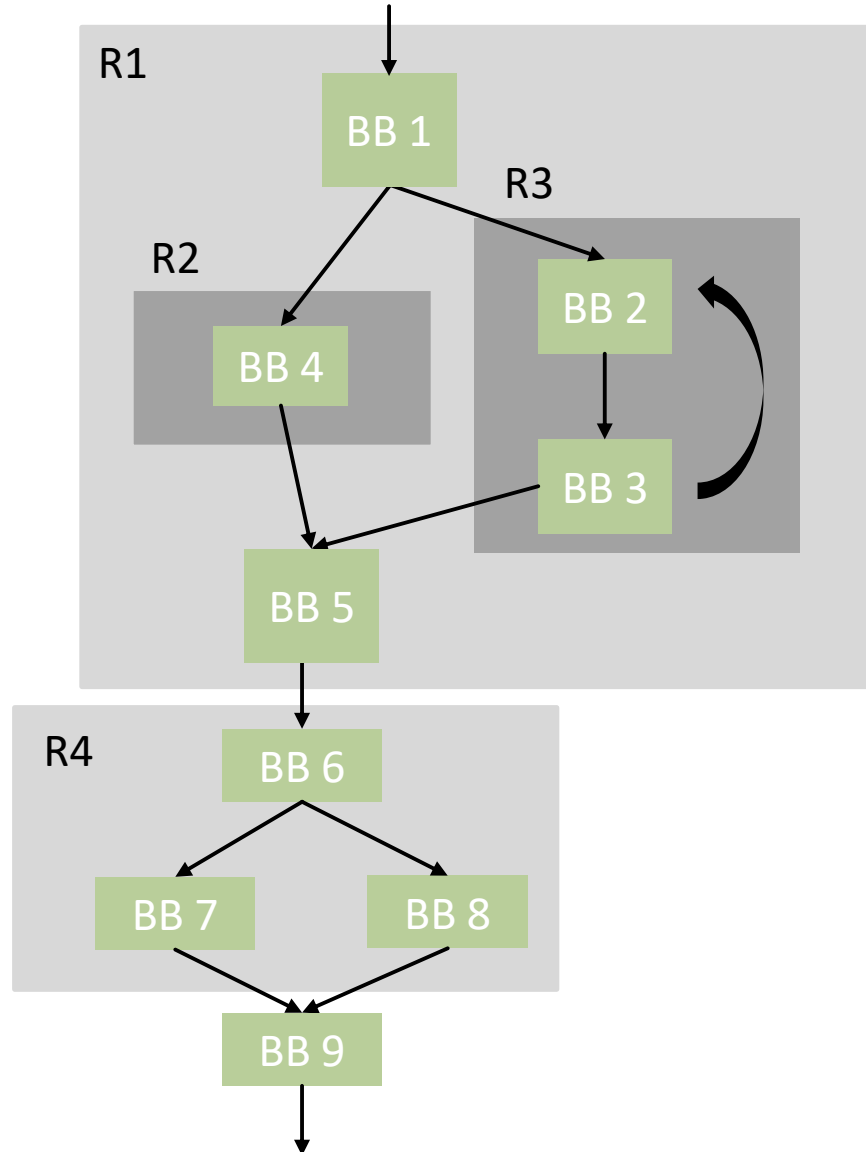


A *refined region* can be transformed into a *simple region* by interesting a single merge block.

Refined Region

Simplified Region

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 expedite 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)
- SignExtension (SCEVSExt)
- ZeroExtension (SCEVZExt)
- Truncation (SCEVTrunc)
- Signed Maximum (SCEVSMMax)
- Unsigned Maximum (SCEVUMMax)

■ Special Values

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

Many heuristics to recover
these common pattern

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): $((75 + \%c + (30 * \%b)) * \text{sizeof}(\text{double})) + \%a$

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

Add-Recurrences

Template of an Add Recurrence:

`{base, +, stride}_<loop>`

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

Value:

`base + <virtual_iv> * stride`

%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

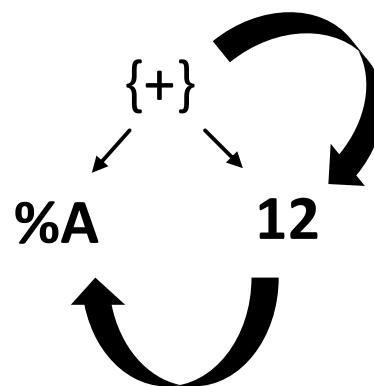
LLVM-IR



{%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++)  
    ...
```

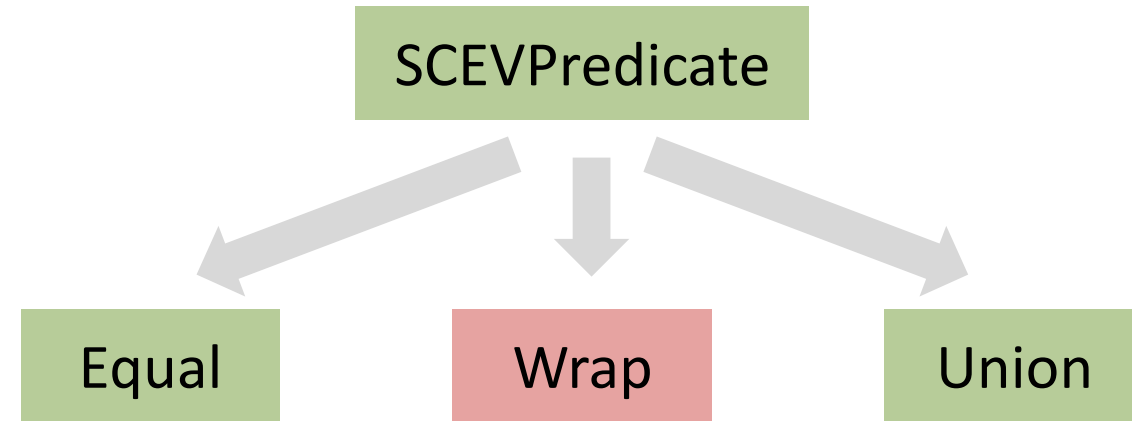


```
const SCEV *getPredicatedBackedgeTakenCount(  
    const Loop *L,  
    SCEVUnionPredicate &Predicates);
```

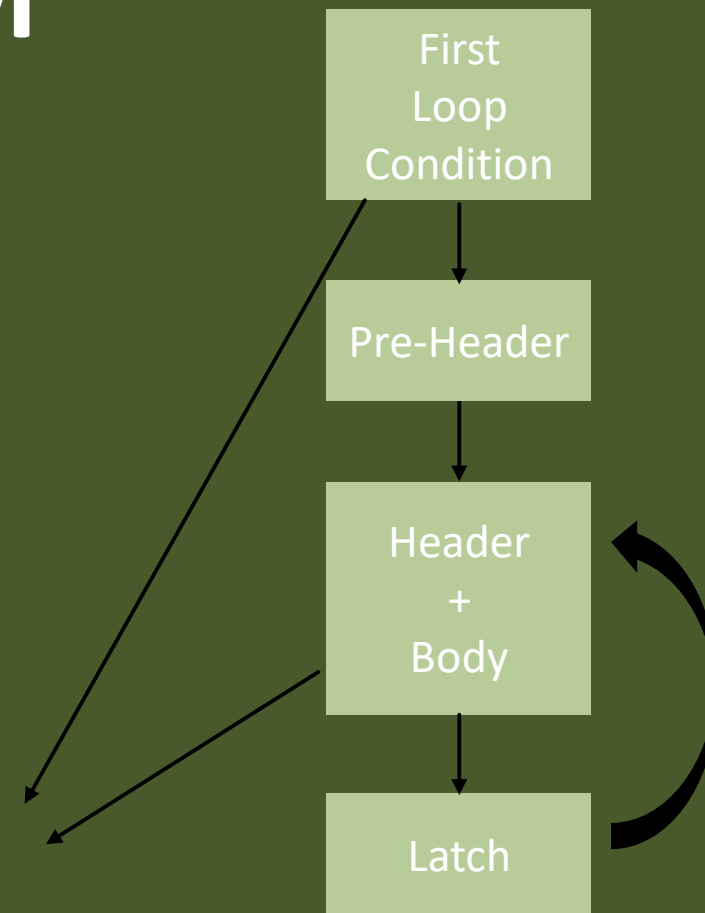
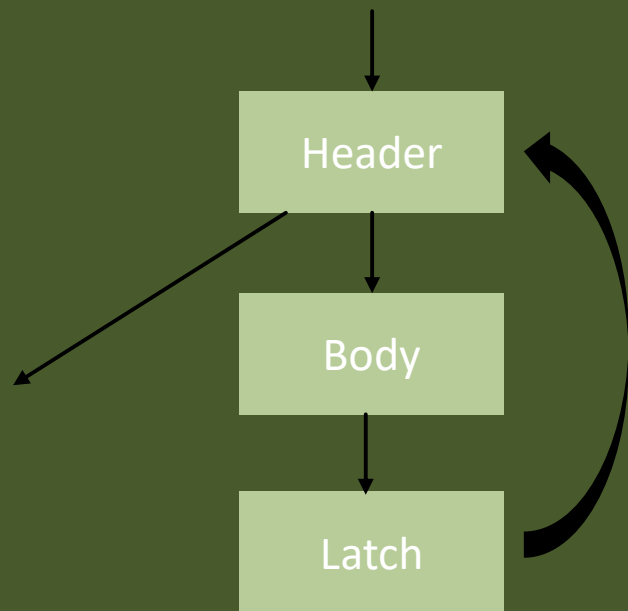


Count: $n + m - p$

Predicate: Assuming $n + m - p$ does not wrap



Loop Transformations in LLVM



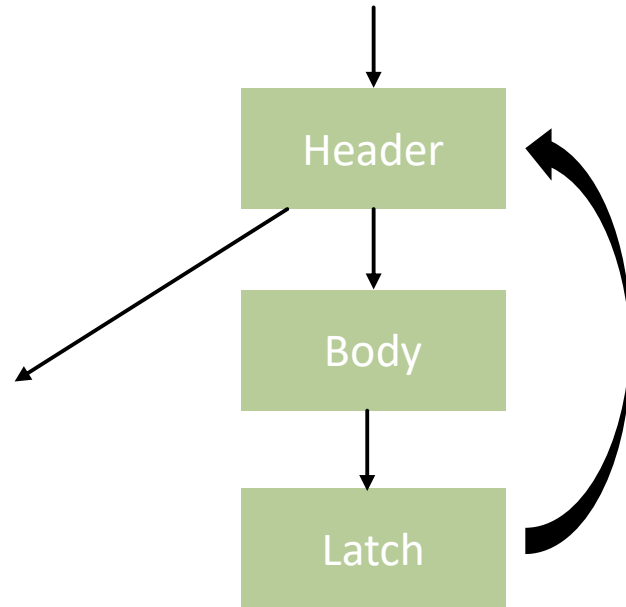
Loop Optimizations in LLVM (ignoring Polly)

- -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

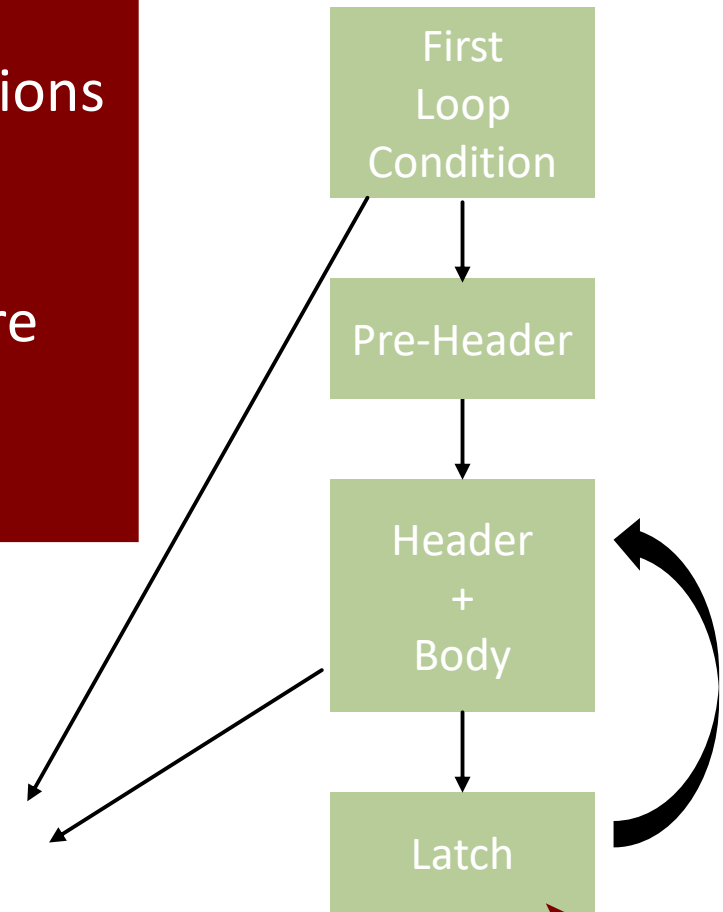
Loop Rotation



Standard For-Loop

Benefits:

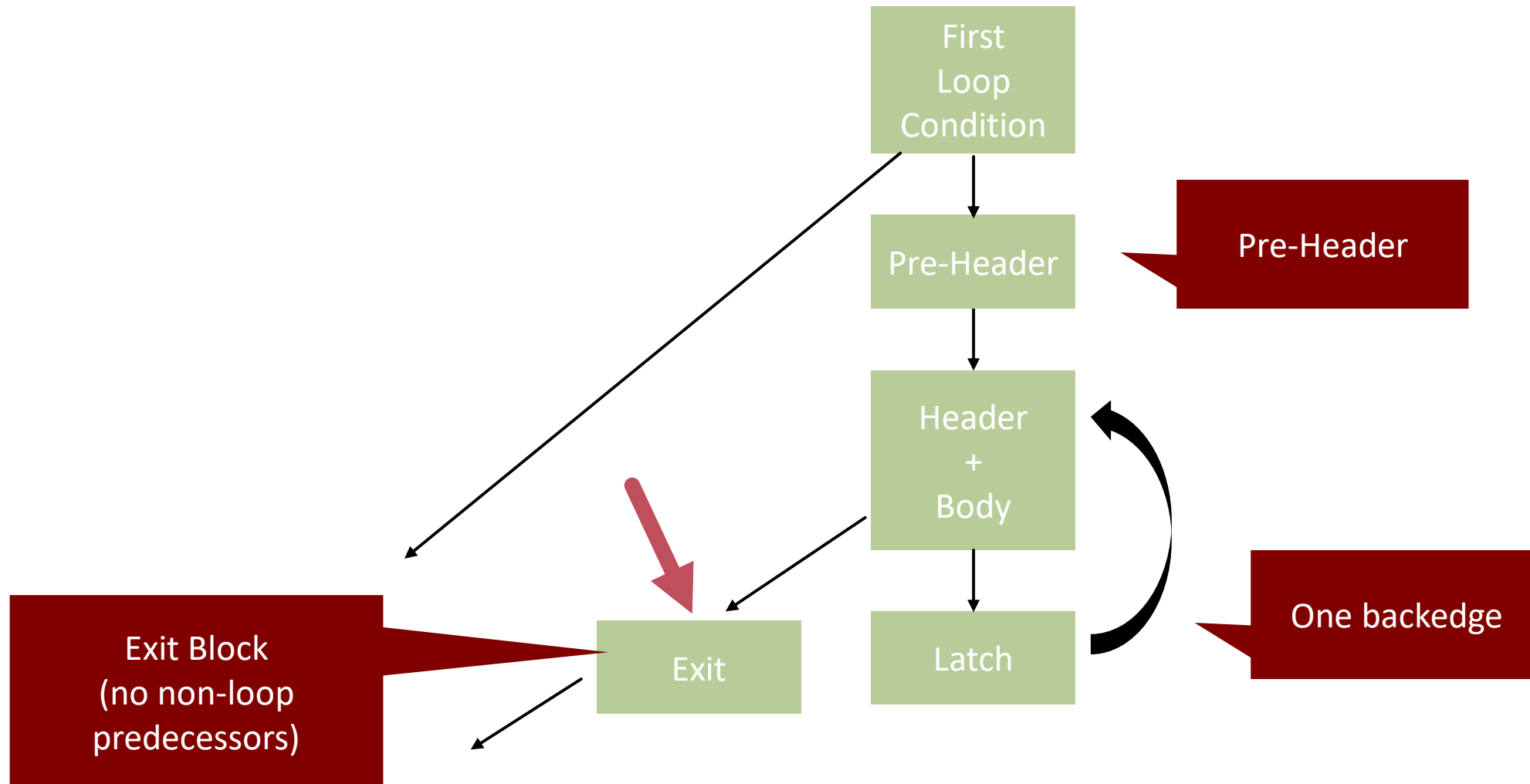
- Invariant instructions can be hoisted to pre-header
- All instructions are executed equally often.



Rotated Loop

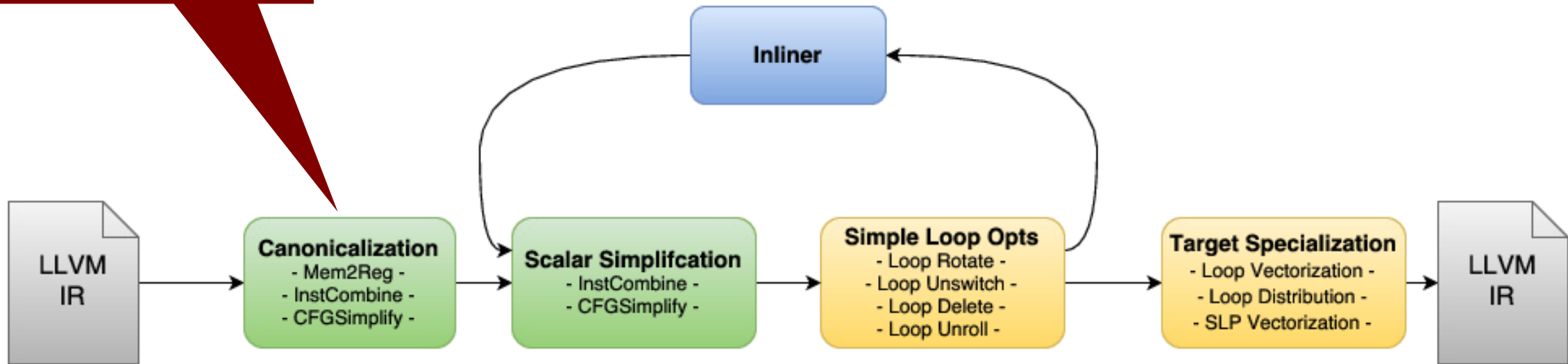
No computational code in Latch!

Loop Simplify Form

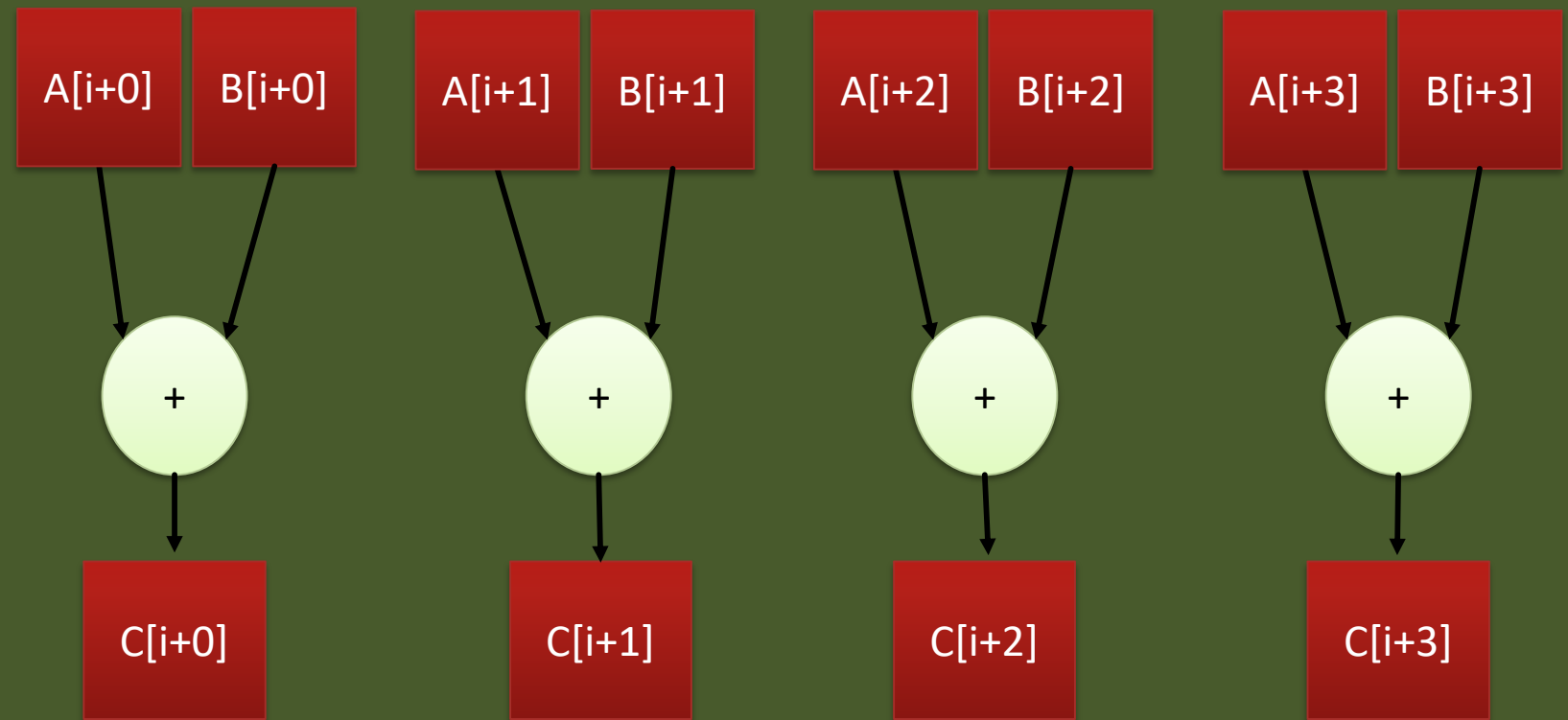


The LLVM Pass Pipeline

Canonicalization is essential for analysis to work!

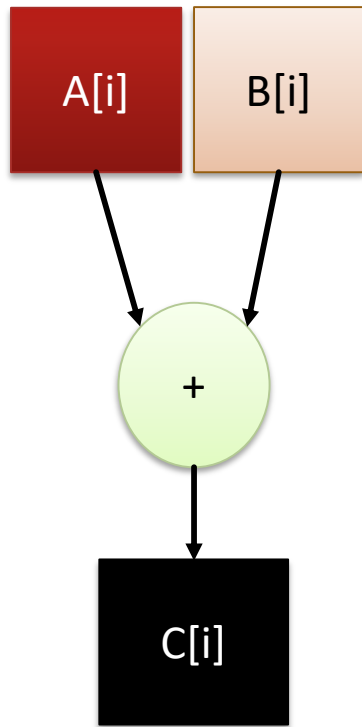


Automatic Vectorization (SIMDization)



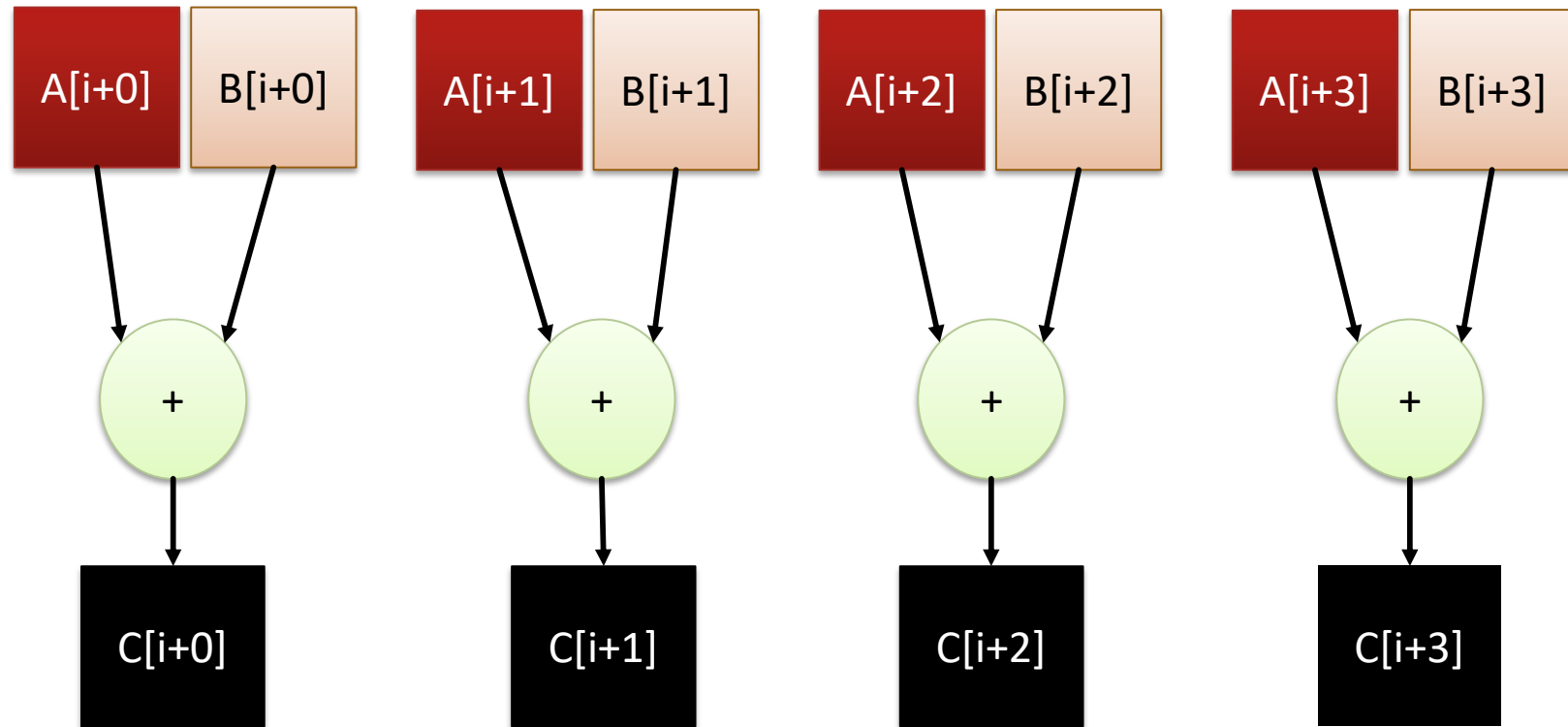
Recap: Vectorization (SIMDization)

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



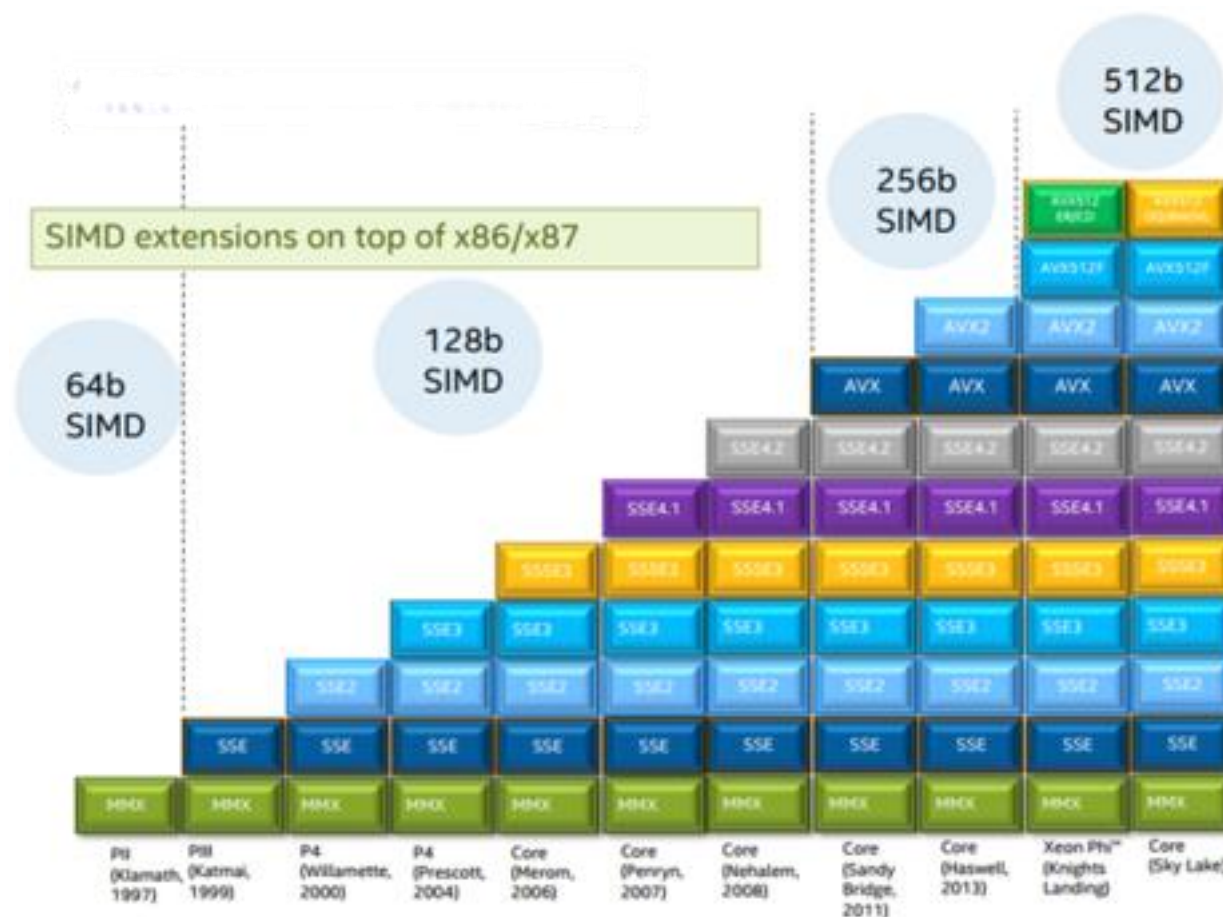
Scalar Execution

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



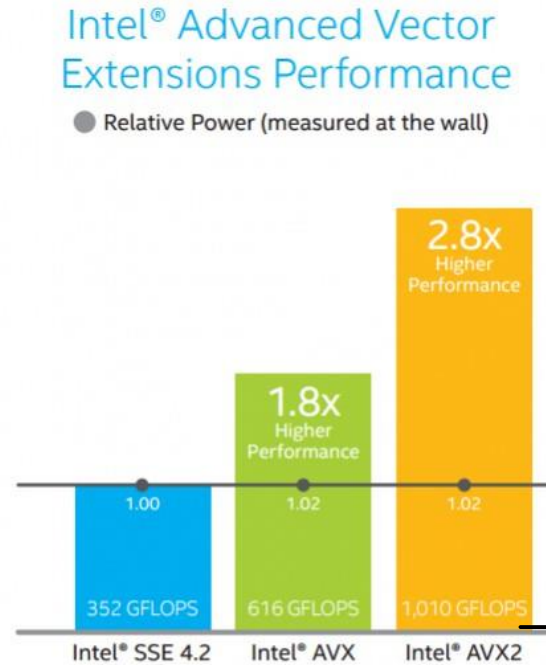
SIMD (Single Instruction Multiple Data) Execution

Intel SIMD Extensions



Vector width
grows
constantly!

SIMDization as solution for higher-performance at constant frequency



Wider Vectors
drive
performance
growth!

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



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!**

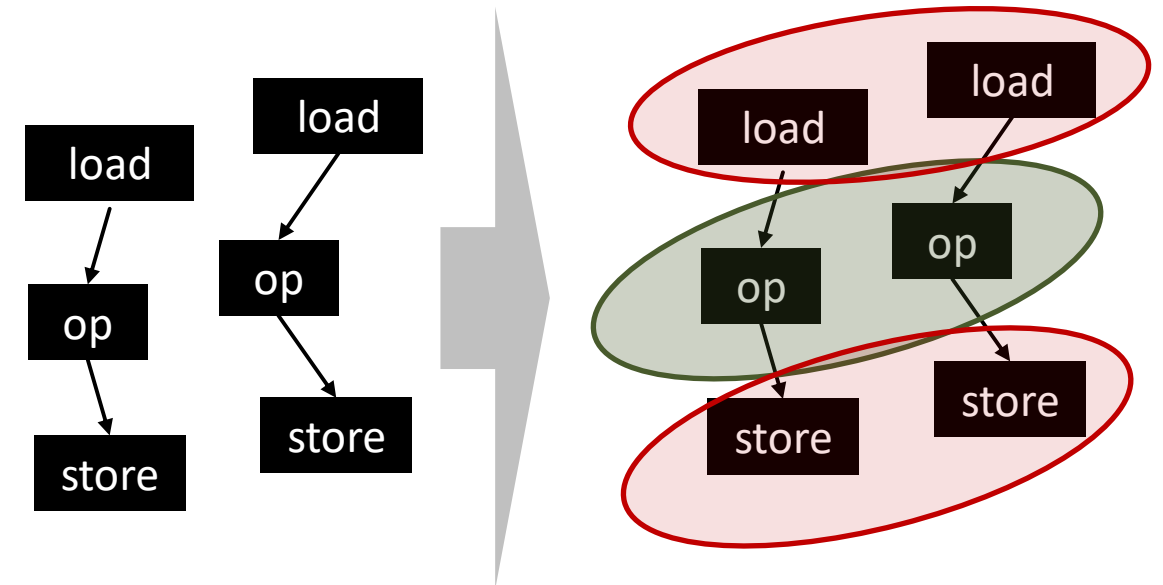
Different Kinds of Automatic SIMDization

Loop Vectorization

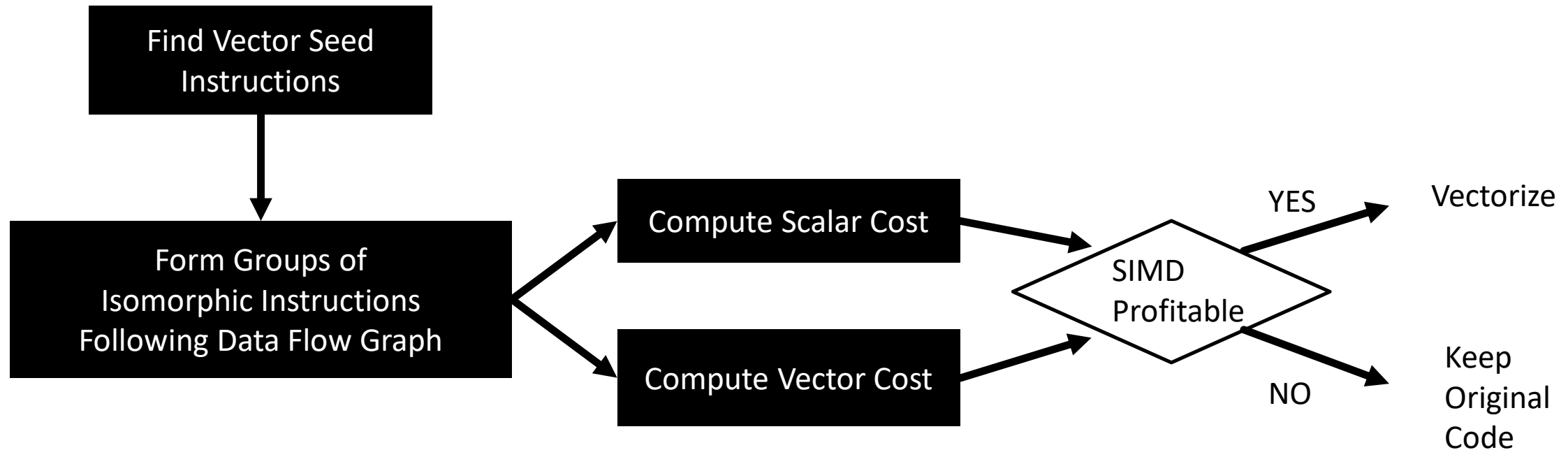
```
for (i = 0; i < n; i++)  
    A[i] = ...
```

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

Superword Level Parallelism (SLP) SIMDization



SLP Vectorization



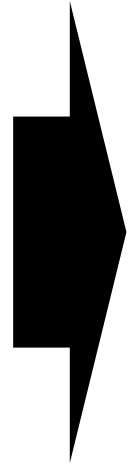
SLP Vectorization - Example

```

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
    
```

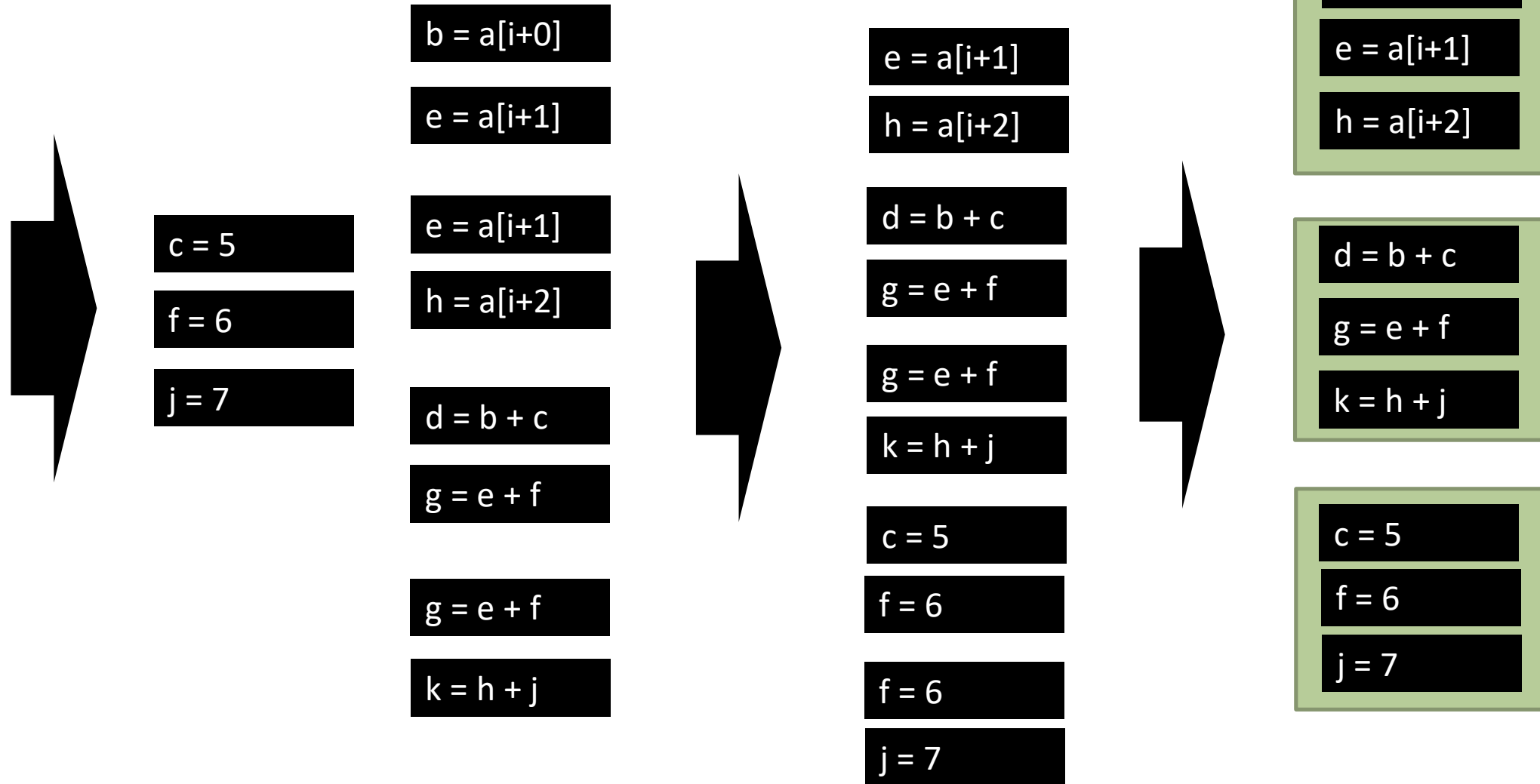
```

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

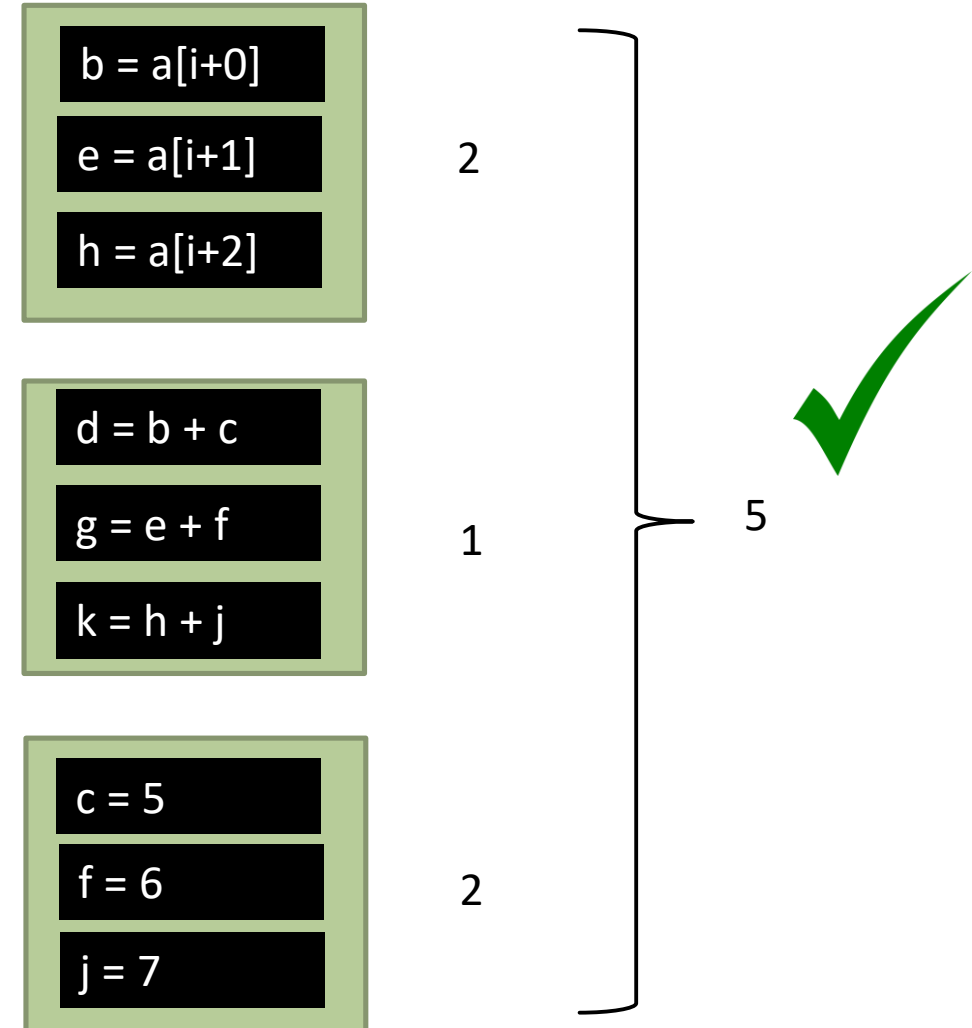
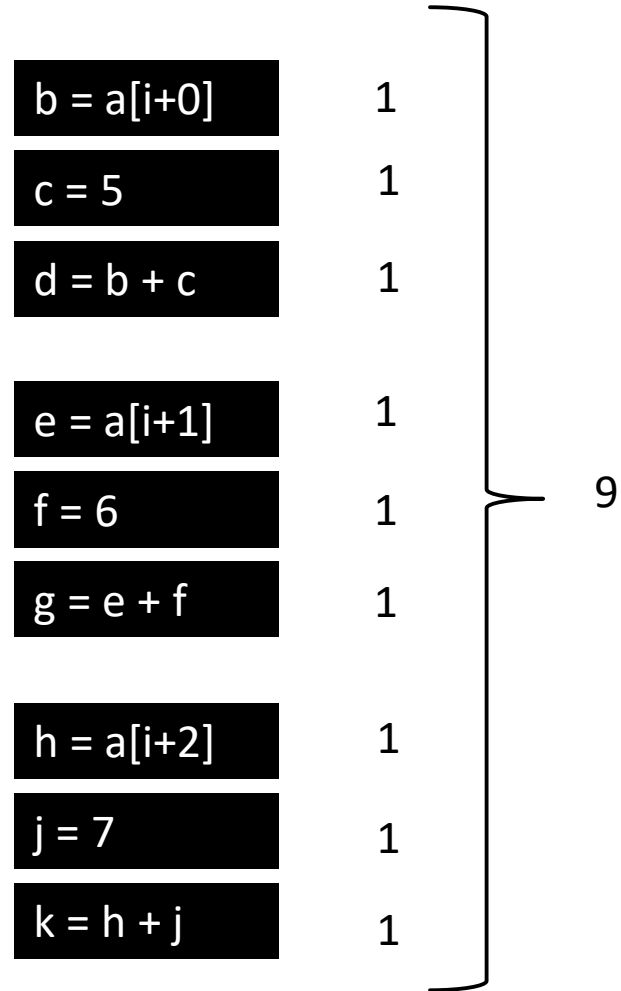
d = b + c
g = e + f

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

SLP Vectorization - Example



Cost Evaluation

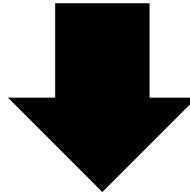


SLP Vectorization – Seed Instructions

- 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; j <= M; j++)
        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

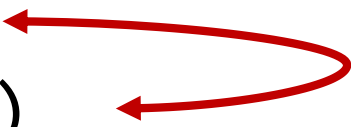
Can these loops be vectorized?

```
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];
```



A red curved arrow points from the 'k' loop to the 'j' loop, with the word 'Interchange' written in red next to it.

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;
```