What's new in VPlan

Florian Hahn
CPU and Accelerator Compilers

Context: Vectorization Plan in the Loop Vectorizer

Ongoing incremental effort to upgrade the Loop Vectorizer's infrastructure and extend its capabilities

Extending LoopVectorizer towards supporting OpenMP4.5 SIMD and outer loop auto-vectorization
2016 US LLVM Developers' Meeting, *H. Saito,* https://www.youtube.com/watch?v=XXAvdUwO7kQ

Introducing VPlan to the Loop Vectorizer
2017 Euro LLVM Developers' Meeting, *G. Rapaport,* https://www.youtube.com/watch?v=IqzJRs6tb7Y

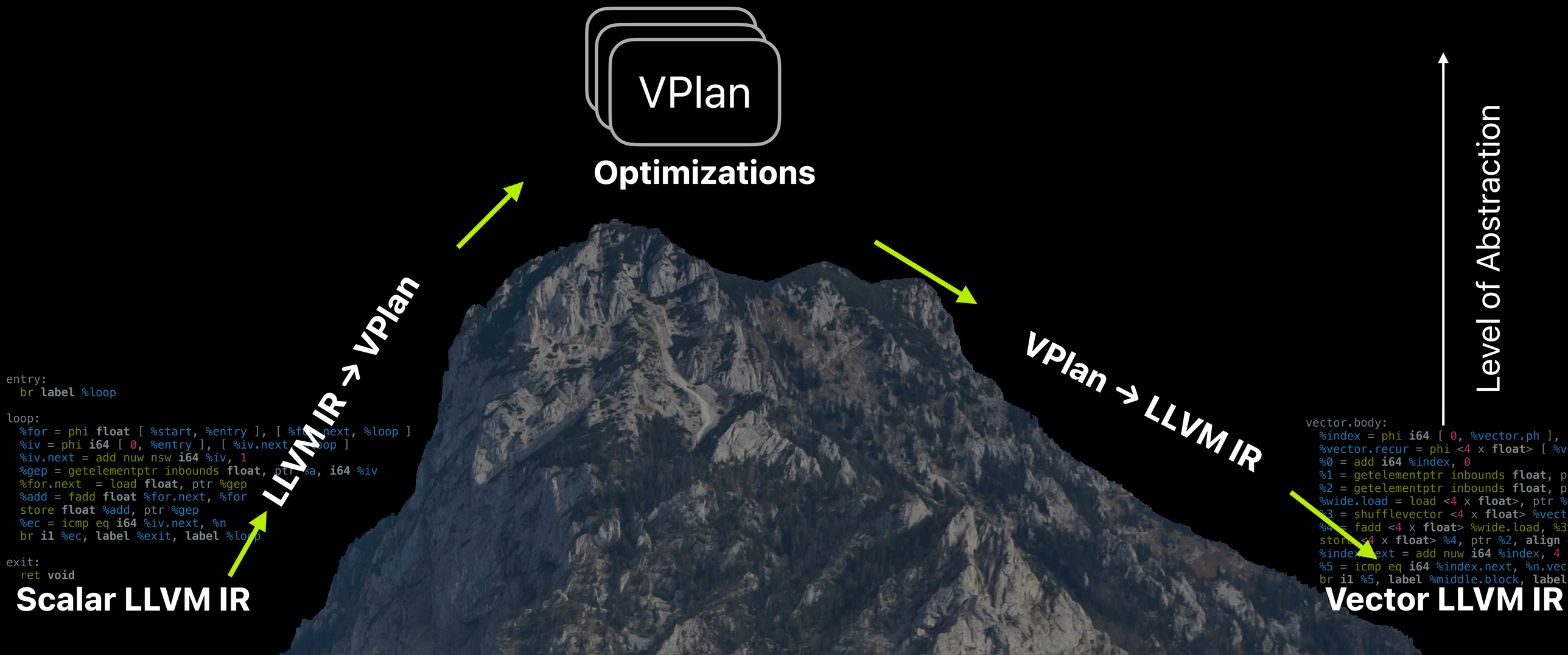
Vectorizing Loops with VPlan – Current State and Next Steps
2017 US LLVM Developers' Meeting, *A. Zaks,* https://www.youtube.com/watch?v=BjBSJFzYDVk

Extending LoopVectorize to Support Outer Loop Vectorization Using VPlan
2018 Euro LLVM Developers' Meeting, *D. Caballero, S. Guggilla,* https://www.youtube.com/watch?v=z6NeHLRNVok

VPlan: Status Update and Future Roadmap
2023 US LLVM Developers' Meeting, *F. Hahn, A. Zaks,* https://www.youtube.com/watch?v=SzGP4PgMuLE

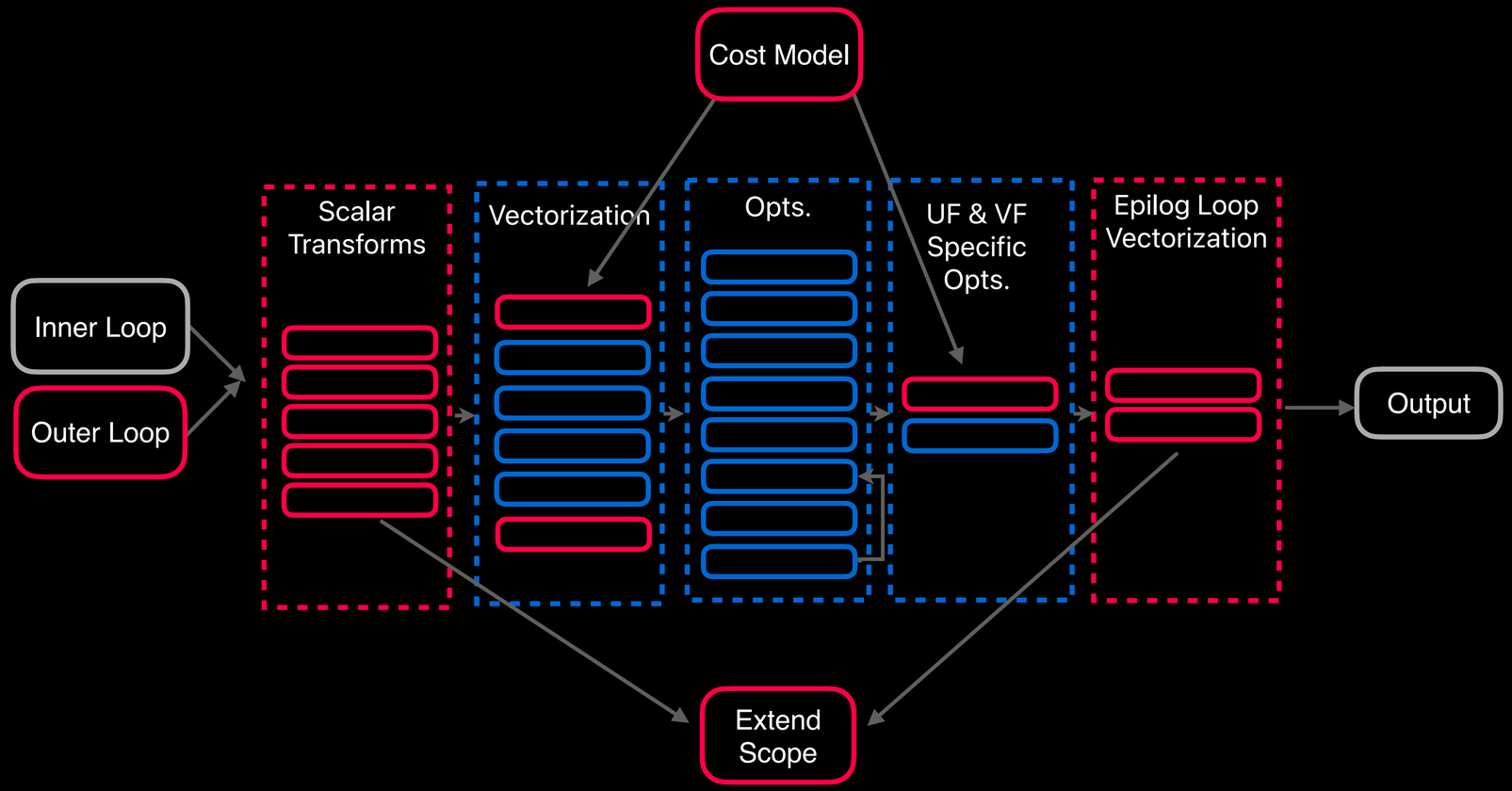
Context: Vectorization Plan in the Loop Vectorizer

Vectorization Plan is an explicit model for describing vectorization candidates



Roadmap Outlined in 2023

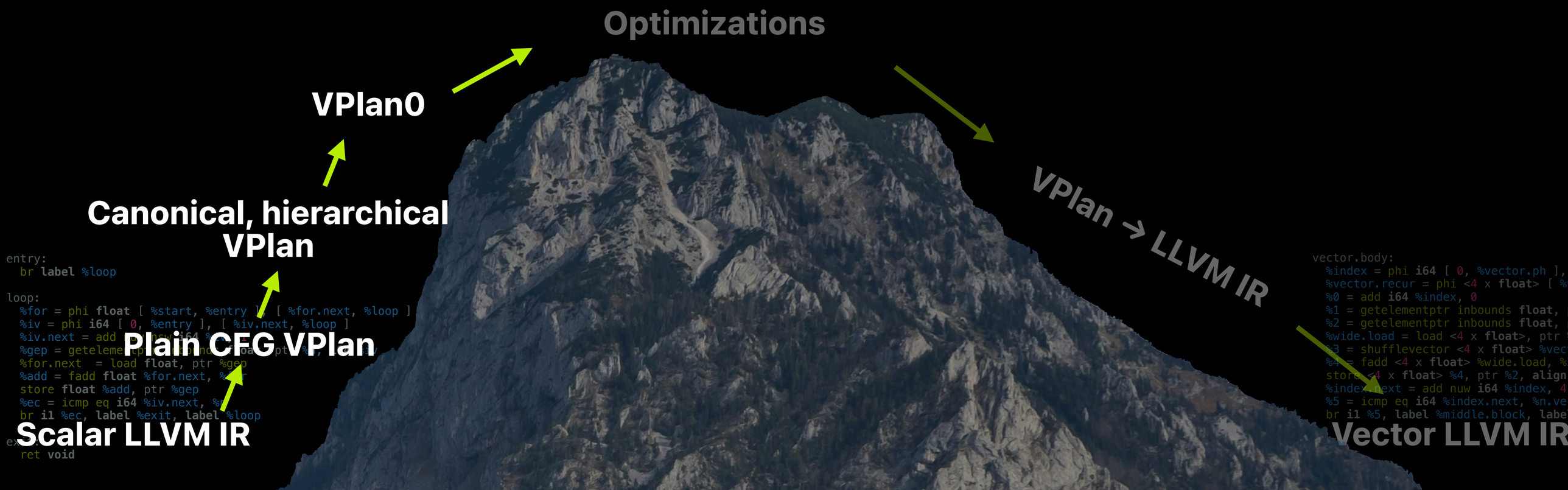
- VPlan is ready for transformations now!
- Multiple advancements in recent years
- Multiple directions forward:



Refactor Initial VPlan Creation (LLVM IR → VPlan)

Goal: Simplify VPlan creation by splitting into set of modular transforms

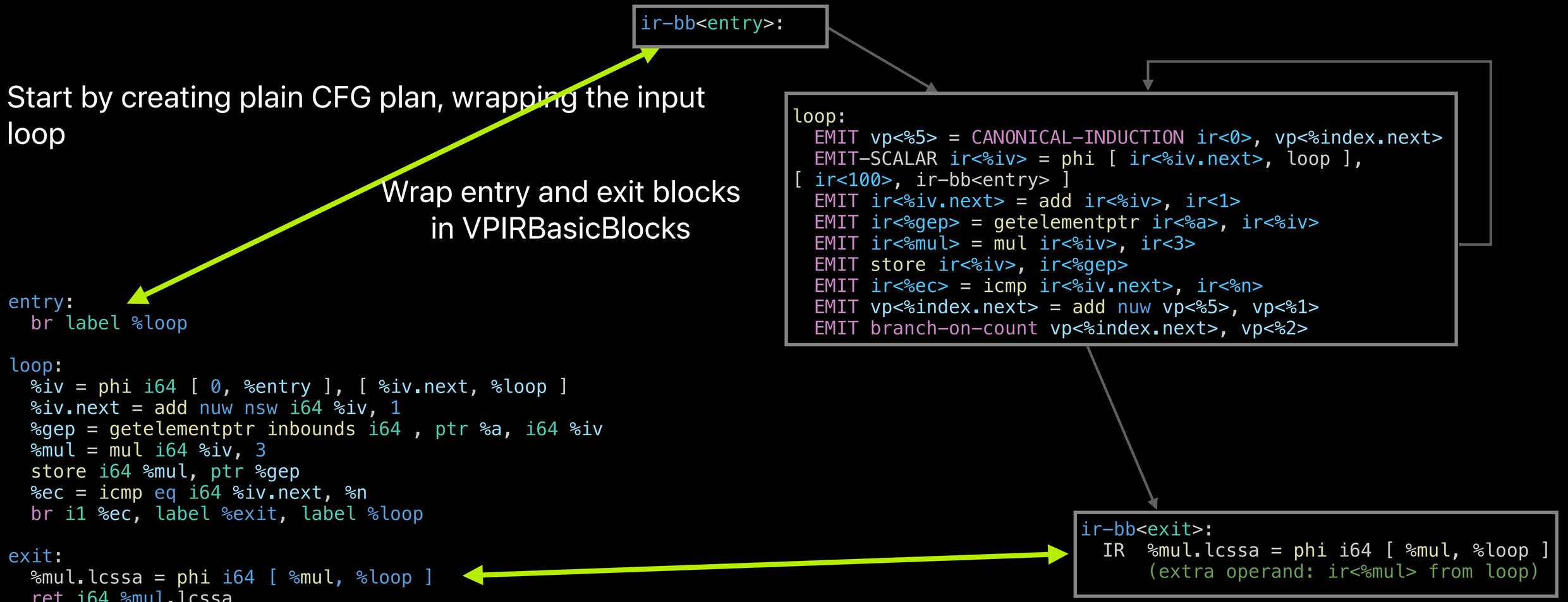
Create initial VPlan0 at the start of the vectorization process



Refactor Initial VPlan Creation (LLVM IR → VPlan)

1. Start by creating plain CFG plan, wrapping the input loop

Wrap entry and exit blocks in VPIRBasicBlocks

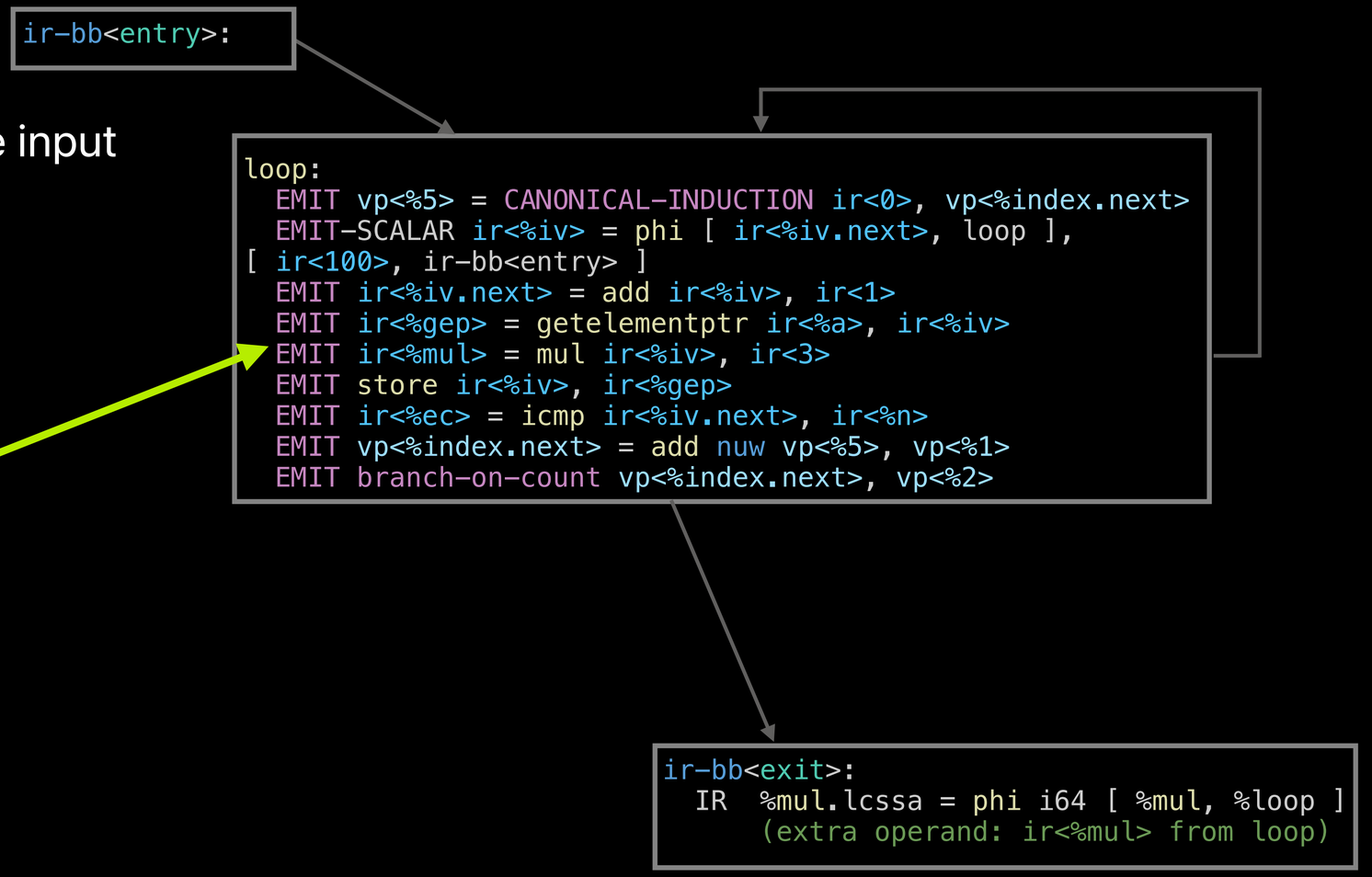


Refactor Initial VPlan Creation (LLVM IR → VPlan)

1. Start by creating plain CFG plan, wrapping the input loop

```
entry:
  br label %loop

loop:
  %iv = phi i64 [ 0, %entry ], [ %iv.next, %loop ]
  %iv.next = add nuw nsw i64 %iv, 1
  %gep = getelementptr inbounds i64 , ptr %a, i64 %iv
  %mul = mul i64 %iv, 3
  store i64 %mul, ptr %gep
  %ec = icmp eq i64 %iv.next, %n
  br i1 %ec, label %exit, label %loop

exit:
  %mul.lcssa = phi i64 [ %mul, %loop ]
  ret i64 %mul.lcssa
```

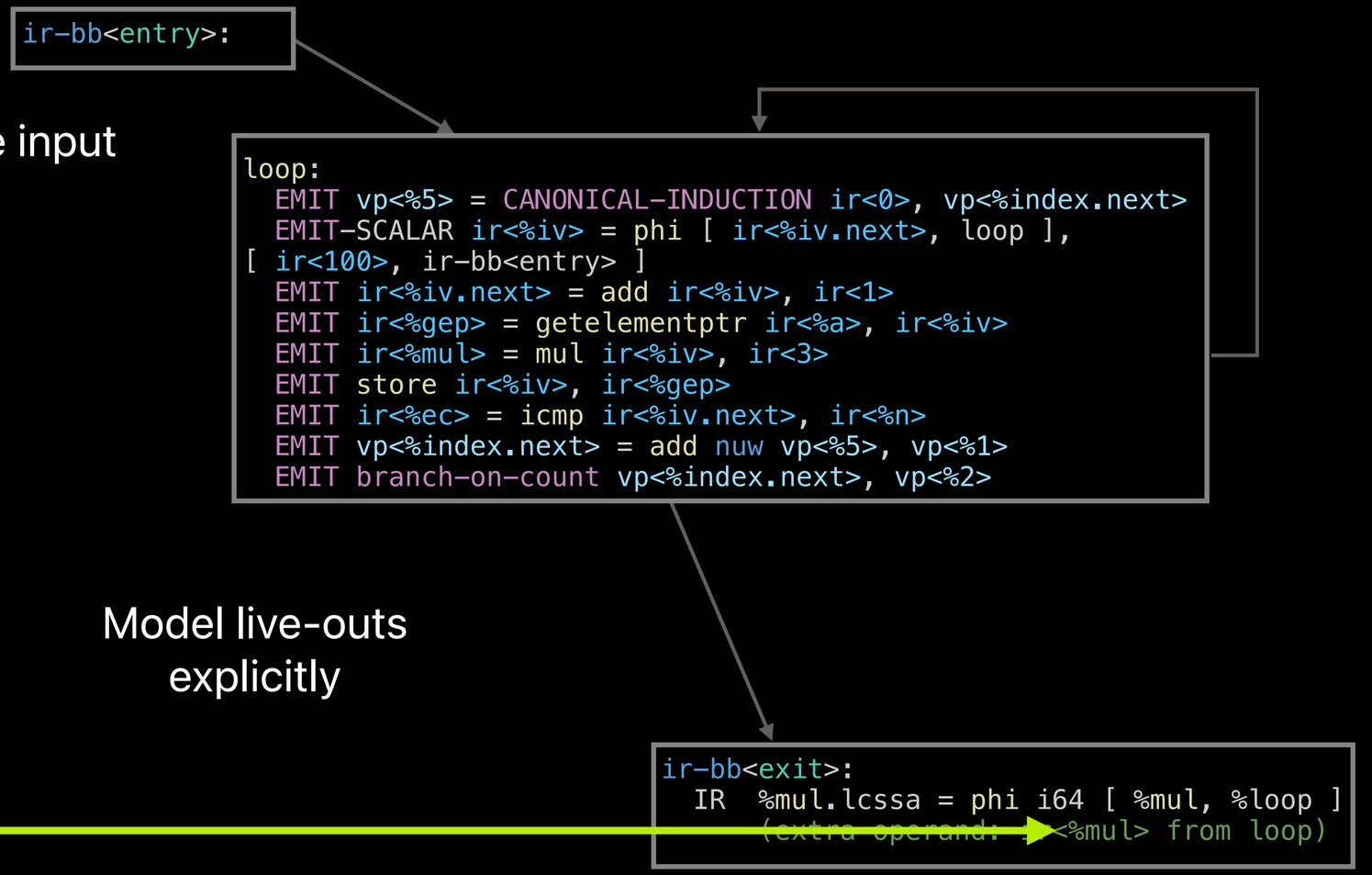
Wrap instructions in loop in VPInstructions



Refactor Initial VPlan Creation (LLVM IR → VPlan)

1. Start by creating plain CFG plan, wrapping the input loop

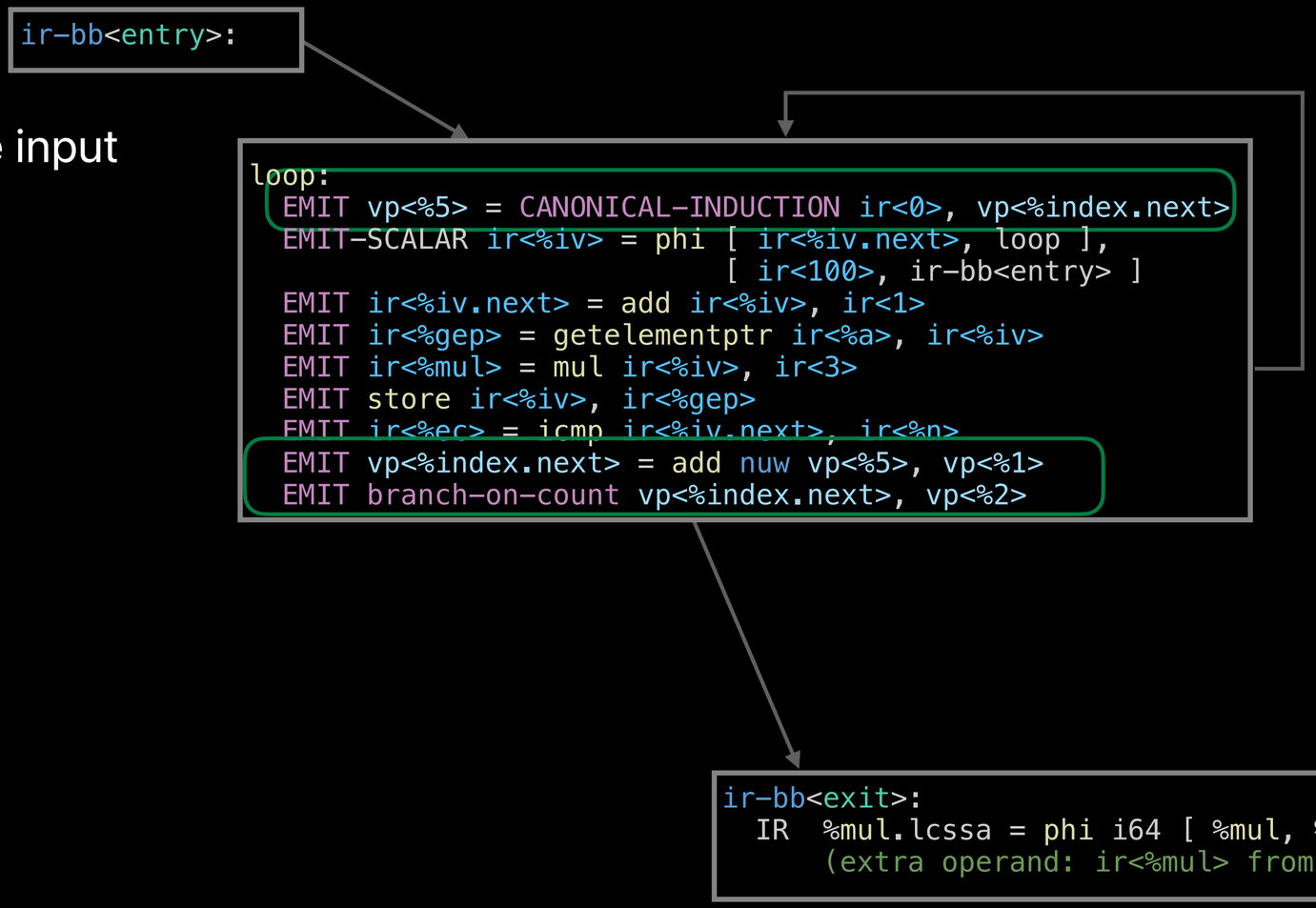
```
entry:
  br label %loop

loop:
  %iv = phi i64 [ 0, %entry ], [ %iv.next, %loop ]
  %iv.next = add nuw nsw i64 %iv, 1
  %gep = getelementptr inbounds i64 , ptr %a, i64 %iv
  %mul = mul i64 %iv, 3
  store i64 %mul, ptr %gep
  %ec = icmp eq i64 %iv.next, %n
  br i1 %ec, label %exit, label %loop

exit:
  %mul.lcssa = phi i64 [ %mul, %loop ]
  ret i64 %mul.lcssa
```



Model live-outs explicitly

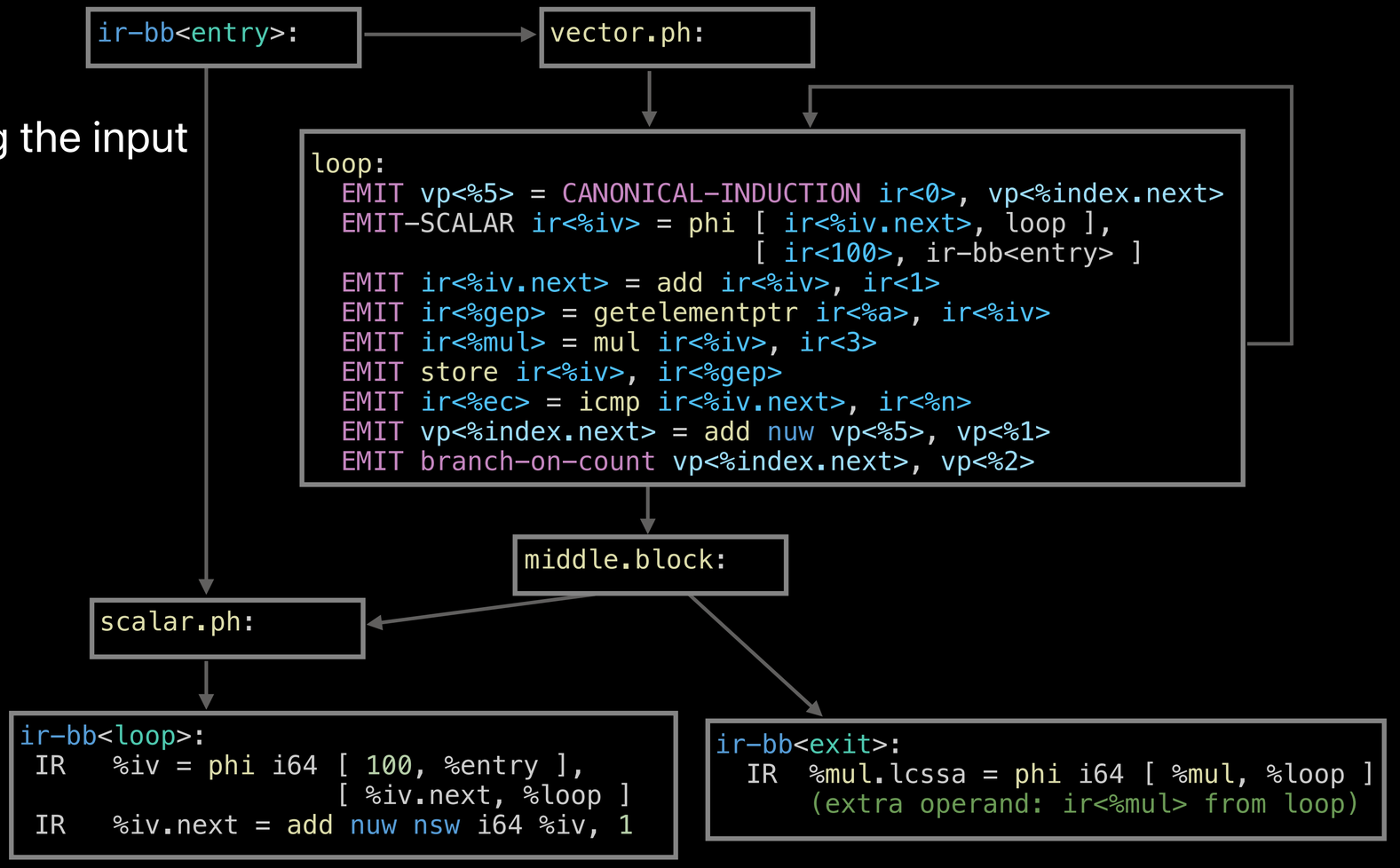
Refactor Initial VPlan Creation (LLVM IR → VPlan)

1. Start by creating plain CFG plan, wrapping the input loop
2. Model Canonical Induction (raising abstraction)



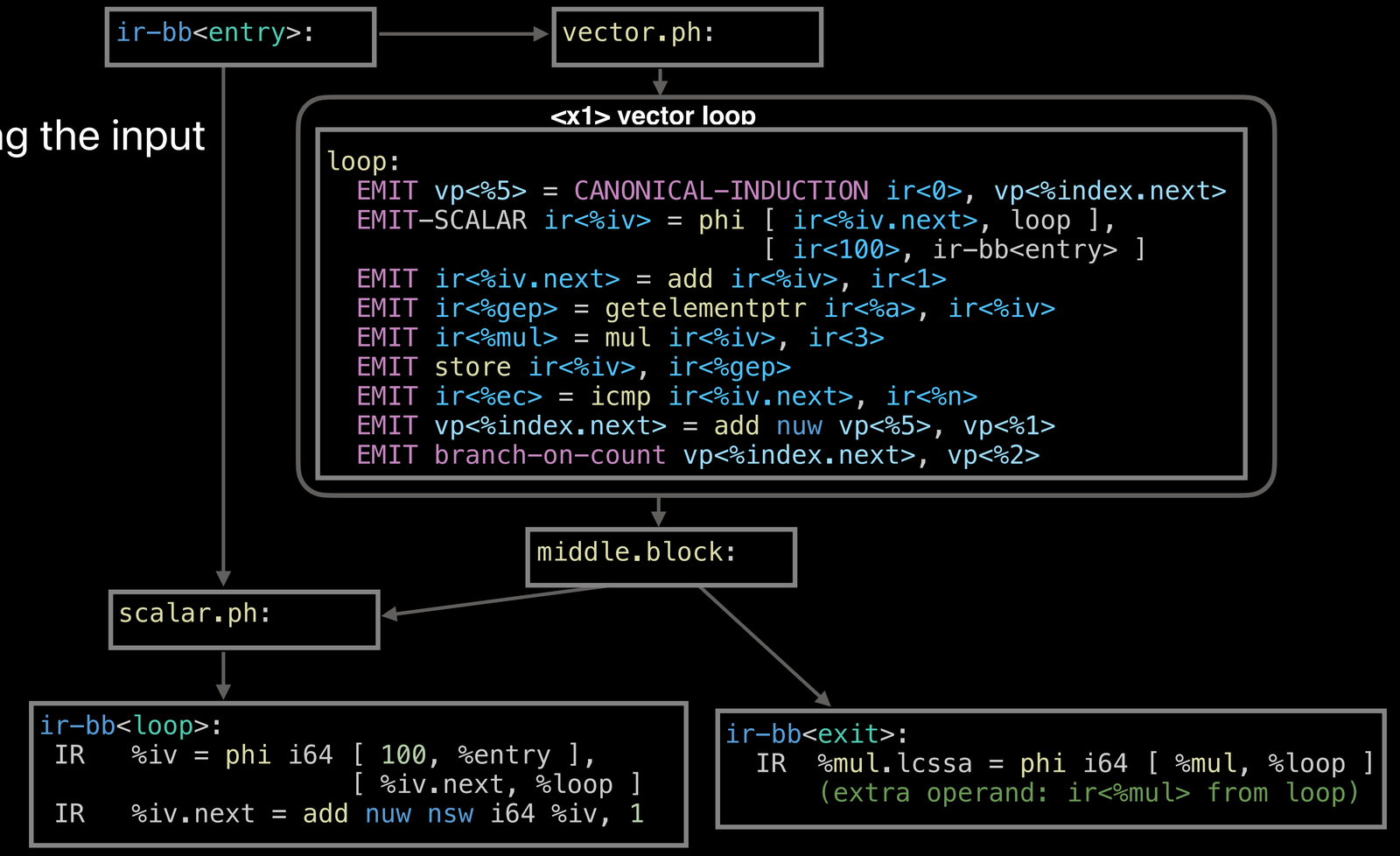
Refactor Initial VPlan Creation (LLVM IR → VPlan)

1. Start by creating plain CFG plan, wrapping the input loop
2. Model Canonical Induction (raising abstraction)
3. Complete Initial Skeleton



Refactor Initial VPlan Creation (LLVM IR → VPlan)

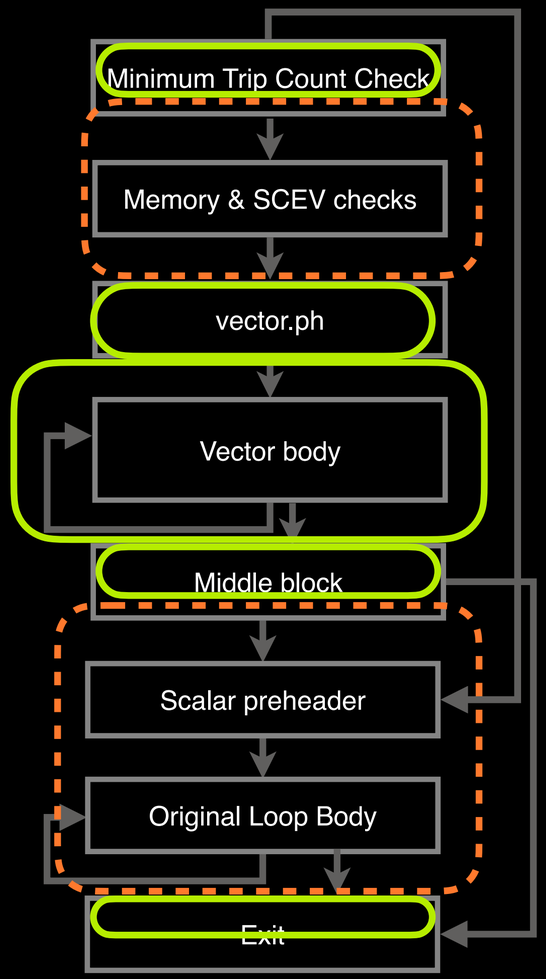
1. Start by creating plain CFG plan, wrapping the input loop
2. Model Canonical Induction (raising abstraction)
3. Complete Initial Skeleton
4. Convert to Vector Loop regions (raising abstraction)



VPlan Scope in 2023

Yet to model in VPlan '23

Output LLVM-IR



VPlan for VF={2}.UF>=1

Live-in vp<%0> = vector-trip-count

```
entry:
  EMIT vp<%1> = EXPAND SCEV (1000 umin %k)
```

vector.ph

<x1> vector loop

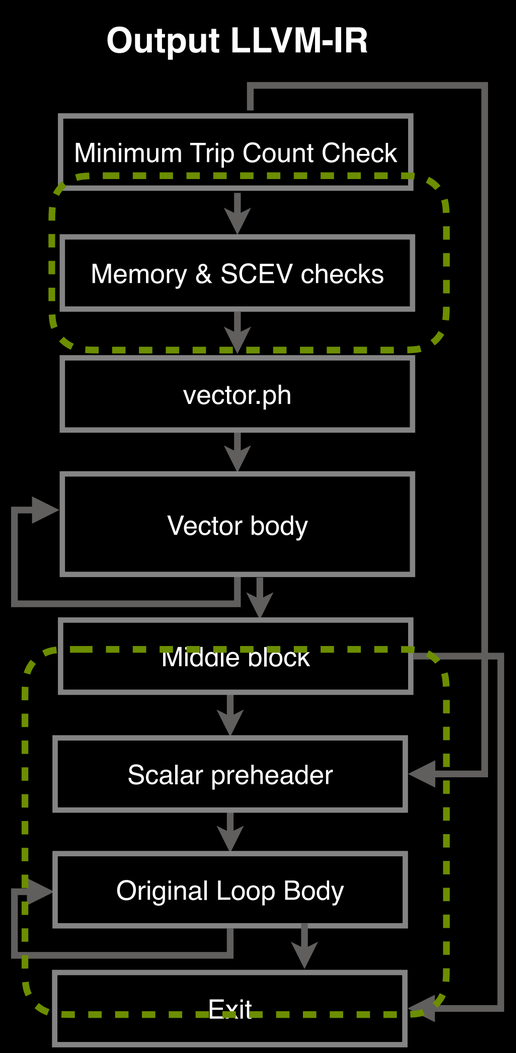
```
vector.body:
  EMIT vp<%2> = CANONICAL-INDUCTION
  WIDEN-INDUCTION %iv = phi 0, %iv.next
  vp<%4> = SCALAR-STEPS vp<%2>, ir<1>
  CLONE ir<%gep.a> = getelementptr ir<%a>,
                                   vp<%4>
  WIDEN ir<%mul> = mul ir<%iv>, ir<3>
  WIDEN store ir<%gep.a>, ir<%mul>
  EMIT vp<%7> = VF * UF + nuw vp<%2>
  EMIT branch-on-count vp<%7>, vp<%0>
```

middle.block

Live-out i32 %lcssa = ir<%mul>

VPlan in 2025

Now modeled in VPlan '25




```
ir-bb<entry>:
  EMIT vp<%1> = EXPAND SCEV (1000 umin %k)
```

```
vector.ph:
  vp<%4> = DERIVED-IV ir<100> + vp<%2> * ir<1>
```

<x1> vector loop

```
vector.body:
  EMIT vp<%2> = CANONICAL-INDUCTION
  WIDEN-INDUCTION %iv = phi 0, %iv.next
  vp<%4> = SCALAR-STEPS vp<%2>, ir<1>
  CLONE ir<%gep.a> = getelementptr ir<%a>,
                                   vp<%4>
  WIDEN ir<%mul> = mul ir<%iv>, ir<3>
  WIDEN store ir<%gep.a>, ir<%mul>
  EMIT vp<%7> = VF * UF + nuw vp<%2>
  EMIT branch-on-count vp<%7>, vp<%0>
```

```
middle.block:
  EMIT vp<%10> = extract-last-element ir<%mul>
  EMIT vp<%cmp.n> = icmp eq vp<%3>, vp<%2>
  EMIT branch-on-cond vp<%cmp.n>
```

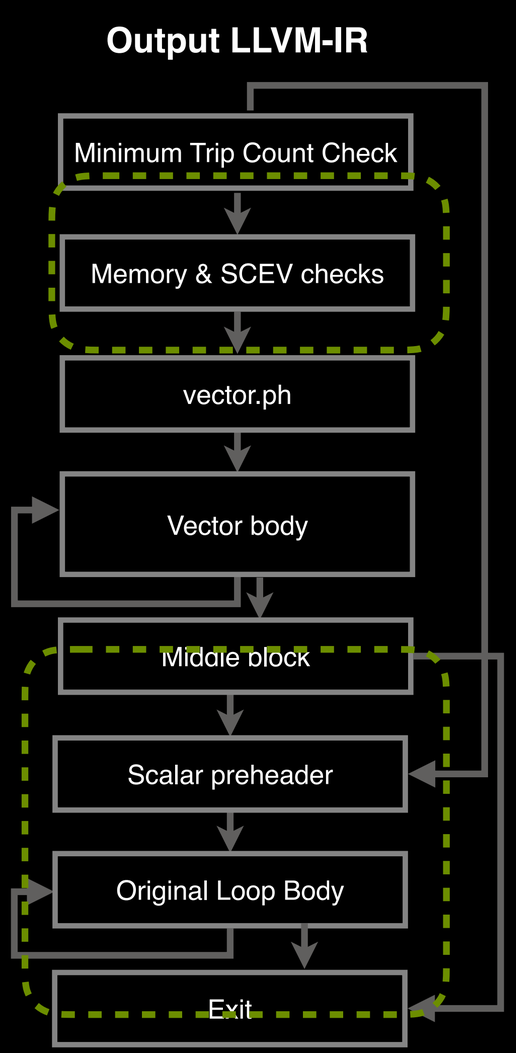
```
scalar.ph:
  EMIT-SCALAR vp<%bc> = phi
                [ vp<%4>, middle.block ],
                [ ir<100>, ir-bb<entry> ]
```

```
ir-bb<loop>:
 IR %iv = phi i64 [ 100, %entry ],
                  [ %iv.next, %loop ]
    (extra operand: vp<%bc> from scalar.ph)
```

```
ir-bb<exit>::
 IR %mul.lcssa = phi i64 [ %mul, %loop ]
(extra operand: vp<%10> from middle.block)
```

VPlan in 2025

Now modeled in VPlan '25



Entry/Exit blocks wrapped in VPIRBasicBlocks

```
ir-bb<entry>:
  EMIT vp<%1> = EXPAND SCEV (1000 umin %k)
```

```
vector.ph:
  vp<%4> = DERIVED-IV ir<100> + vp<%2> * ir<1>
```

<x1> vector loop

```
vector.body:
  EMIT vp<%2> = CANONICAL-INDUCTION
  WIDEN-INDUCTION %iv = phi 0, %iv.next
  vp<%4> = SCALAR-STEPS vp<%2>, ir<1>
  CLONE ir<%gep.a> = getelementptr ir<%a>,
                                   vp<%4>
  WIDEN ir<%mul> = mul ir<%iv>, ir<3>
  WIDEN store ir<%gep.a>, ir<%mul>
  EMIT vp<%7> = VF * UF + nuw vp<%2>
  EMIT branch-on-count vp<%7>, vp<%0>
```

```
middle.block:
  EMIT vp<%10> = extract-last-element ir<%mul>
  EMIT vp<%cmp.n> = icmp eq vp<%3>, vp<%2>
  EMIT branch-on-cond vp<%cmp.n>
```

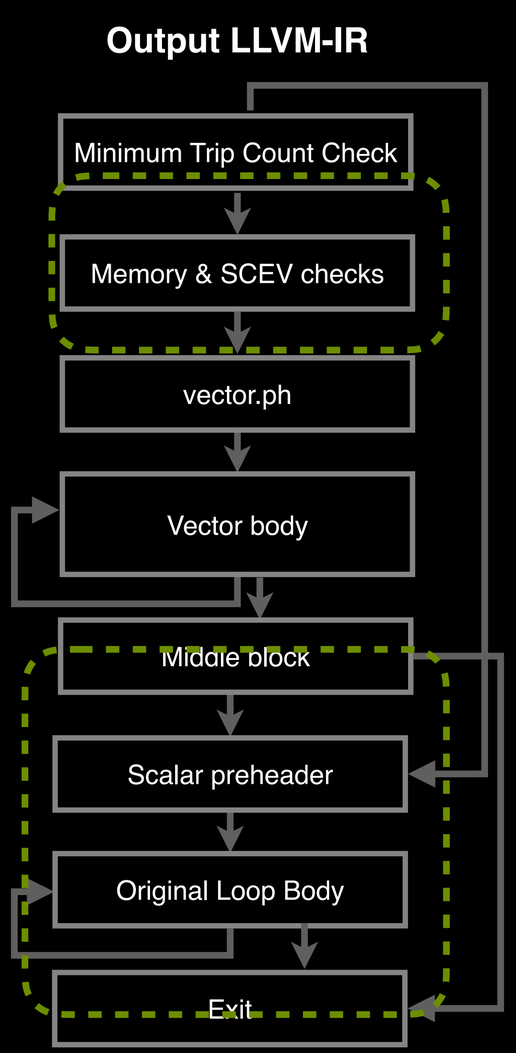
```
scalar.ph:
  EMIT-SCALAR vp<%bc> = phi
              [ vp<%4>, middle.block ],
              [ ir<100>, ir-bb<entry> ]
```

```
ir-bb<loop>:
 IR %iv = phi i64 [ 100, %entry ],
                  [ %iv.next, %loop ]
    (extra operand: vp<%bc> from scalar.ph)
```

```
ir-bb<exit>::
 IR %mul.lcssa = phi i64 [ %mul, %loop ]
(extra operand: vp<%10> from middle.block)
```

VPlan in 2025

Now modeled in VPlan '25




```
ir-bb<entry>:
  EMIT vp<%1> = EXPAND SCEV (1000 umin %k)
```

```
vector.ph:
  vp<%4> = DERIVED-IV ir<100> + vp<%2> * ir<1>
```

<x1> vector loop

```
vector.body:
  EMIT vp<%2> = CANONICAL-INDUCTION
  WIDEN-INDUCTION %iv = phi 0, %iv.next
  vp<%4> = SCALAR-STEPS vp<%2>, ir<1>
  CLONE ir<%gep.a> = getelementptr ir<%a>,
                                   vp<%4>
  WIDEN ir<%mul> = mul ir<%iv>, ir<3>
  WIDEN store ir<%gep.a>, ir<%mul>
  EMIT vp<%7> = VF * UF + nuw vp<%2>
  EMIT branch-on-count vp<%7>, vp<%0>
```

Resume & Exit values modeled explicitly

```
middle.block:
  EMIT vp<%10> = extract-last-element ir<%mul>
  EMIT vp<%cmp.n> = icmp eq vp<%3>, vp<%2>
  EMIT branch-on-cond vp<%cmp.n>
```

```
scalar.ph:
  EMIT-SCALAR vp<%bc> = phi
              [ vp<%4>, middle.block ],
              [ ir<100>, ir-bb<entry> ]
```

```
ir-bb<loop>:
 IR %iv = phi i64 [ 100, %entry ],
                  [ %iv.next, %loop ]
    (extra operand: vp<%bc> from scalar.ph)
```

```
ir-bb<exit>::
 IR %mul.lcssa = phi i64 [ %mul, %loop ]
(extra operand: vp<%10> from middle.block)
```

Increasing VPlan Scope: Retire Legacy Code

Removes legacy runtime check handling

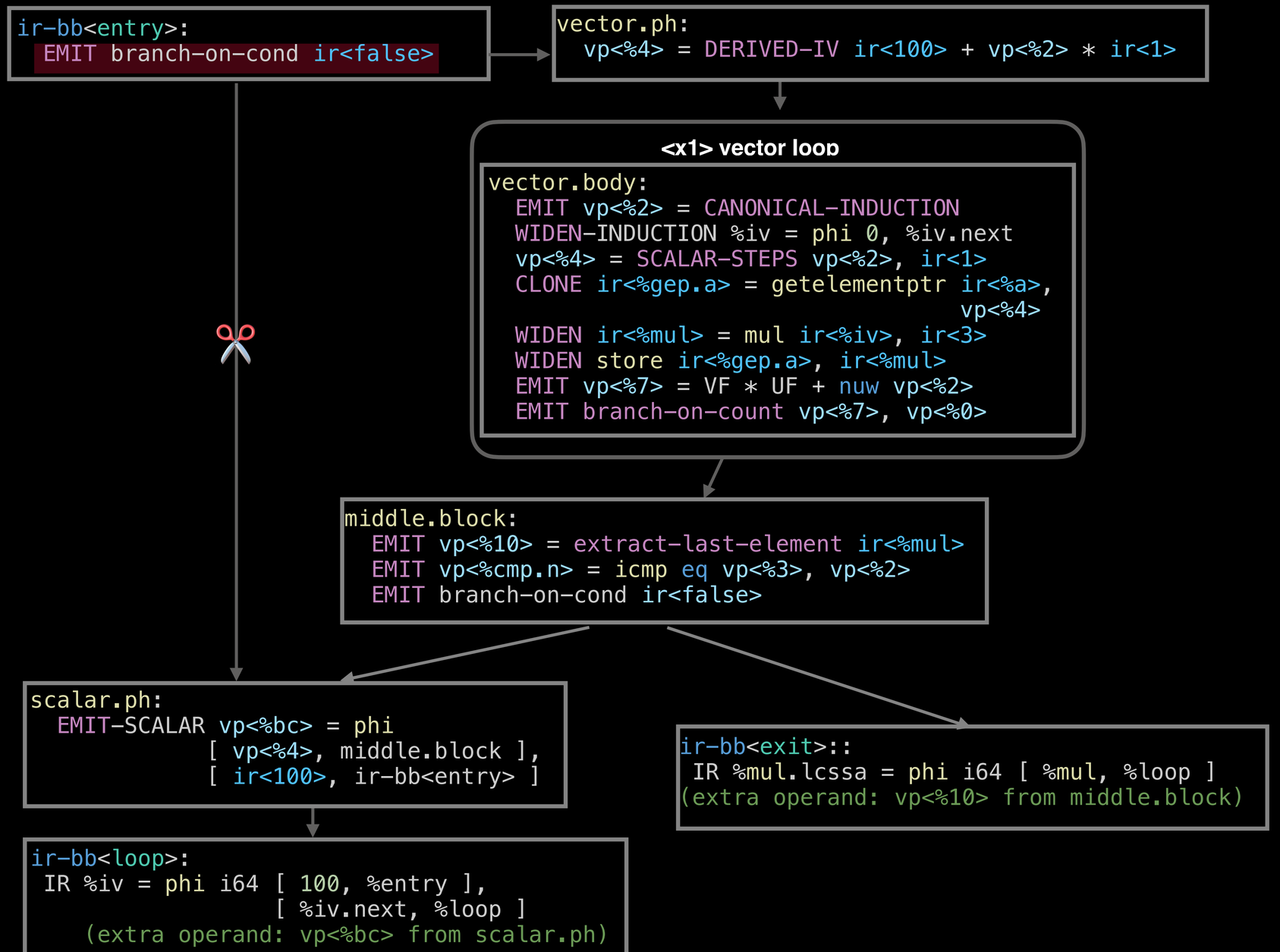
```
class InnerLoopVectorizer {
public:
...
-  // Create a check to see if the vector loop should be executed
-  Value *createIterationCountCheck(ElementCount VF, unsigned UF)
const;
-
-  /// Emit a bypass check to see if the vector trip count is zero, including if
-  /// it overflows.
-  void emitIterationCountCheck(BasicBlock *Bypass);
-
-  /// Emit basic blocks (prefixed with \p Prefix) for the iteration check,
-  /// vector loop preheader, middle block and scalar preheader.
-  void createVectorLoopSkeleton(StringRef Prefix);
+  /// Create a new IR basic block for the scalar preheader whose name is
+  /// prefixed with \p Prefix.
+  void createScalarPreheader(StringRef Prefix);
...
/// vector elements.
ElementCount VF;

-  ElementCount MinProfitableTripCount;
-
```

Moves epilogue trip count check logic to InnerLoopAndEpilogueVectorizer

```
class InnerLoopAndEpilogueVectorizer : public InnerLoopVectorizer {
public:
...
GeneratedRTChecks &Checks, VPlan &Plan, ElementCount VecWidth,
ElementCount MinProfitableTripCount, unsigned UnrollFactor)
  : InnerLoopVectorizer(OrigLoop, PSE, LI, DT, TTI, AC, VecWidth,
-                        MinProfitableTripCount, UnrollFactor, CM, BFI,
-                        Checks, Plan),
-        EPI(EPI) {}
-
-  // Override this function to handle the more complex control flow a
-  // three loops.
-  BasicBlock *createVectorizedLoopSkeleton() final {
-    return createEpilogueVectorizedLoopSkeleton();
-  }
-
-  /// The interface for creating a vectorized skeleton using one of two
-  /// different strategies, each corresponding to one execution of the vplan
-  /// as described above.
-  virtual BasicBlock *createEpilogueVectorizedLoopSkeleton() = 0;
+                               UnrollFactor, CM, BFI, PSI, Checks, Plan)
+        EPI(EPI), MinProfitableTripCount(MinProfitableTripCount) {}
```

Increasing VPlan Scope: Additional Simplifications

1. Simplify conditional branches



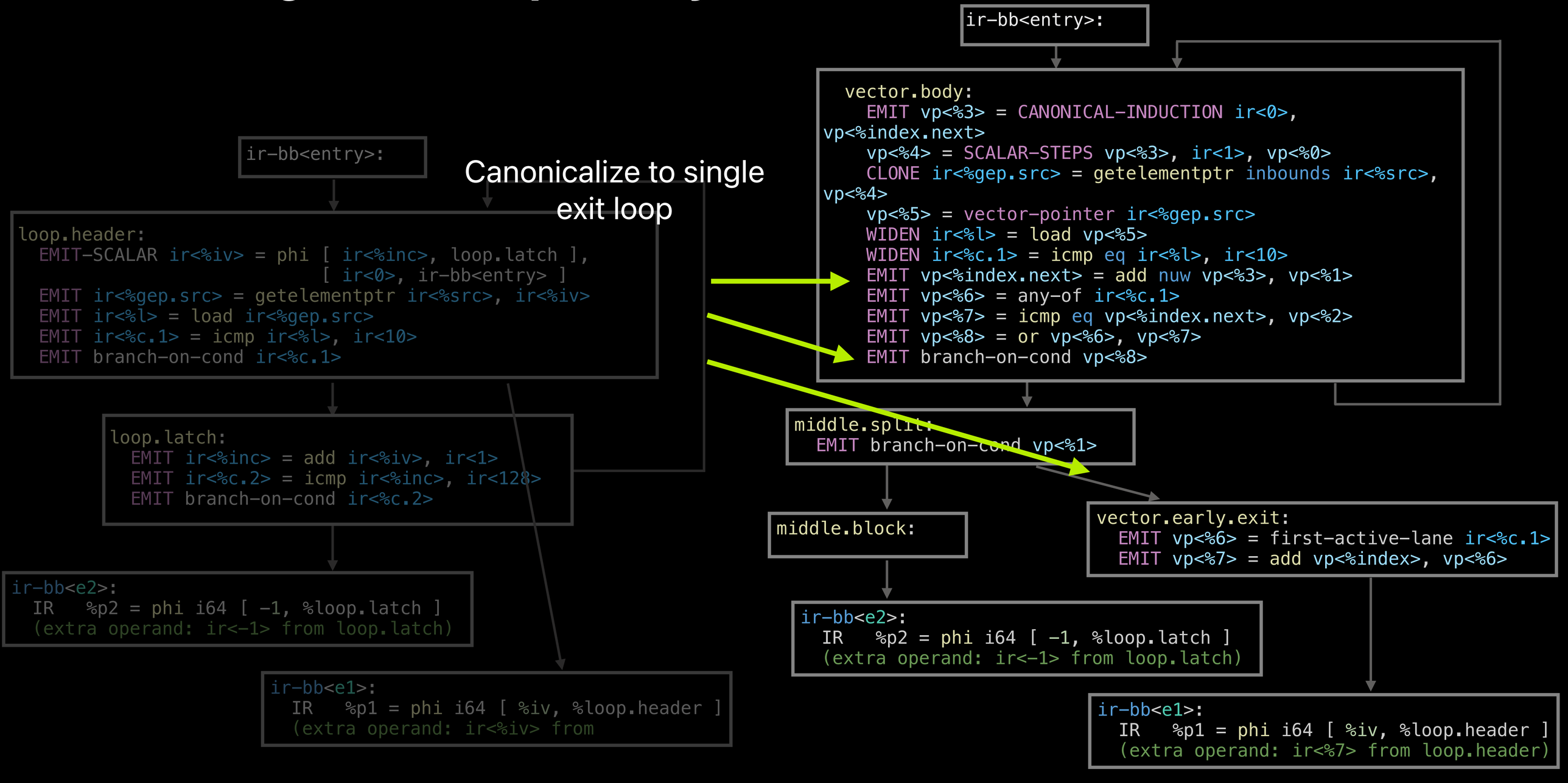
Increasing VPlan Scope: Additional Simplifications

1. Simplify conditional branches

Increasing VPlan Scope: Additional Simplifications

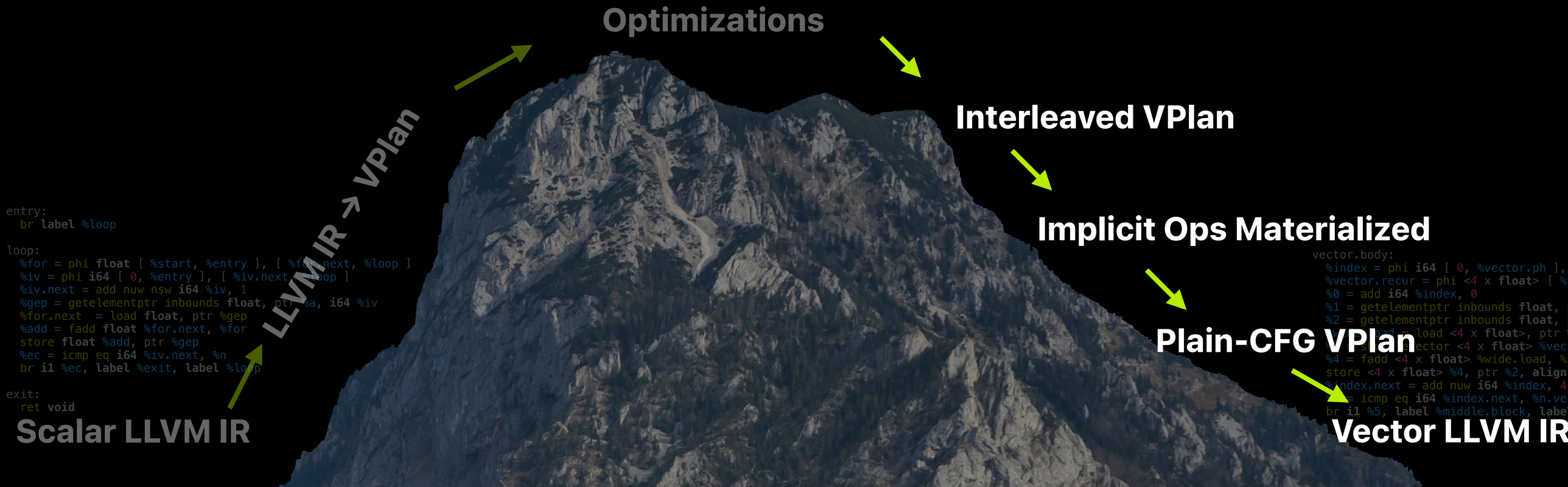
1. Simplify conditional branches
2. Remove unreachable blocks

Increasing VPlan Scope: Early-Exit Vectorization



Simplify VPlan Execution (VPlan → LLVM IR)

Goal: Simplify VPlan ::execution by gradually lowering recipes to simpler ones



Interleave Vector Iterations

Perform interleaving explicitly as VPlan transform

Plan valid for any interleave count

<x1> vector loop

```
vector.body:
  EMIT vp<%3> = CANONICAL-INDUCTION ir<0>,
                                    vp<%index.next>
  ir<%iv> = WIDEN-INDUCTION  ir<0>, ir<1>, vp<%0>
  vp<%4> = SCALAR-STEPS vp<%3>, ir<1>, vp<%0>
  CLONE ir<%gep> = getelementptr inbounds ir<%a>,
                                          vp<%4>
  WIDEN ir<%mul> = mul ir<%iv>, ir<3>
  vp<%5> = vector-pointer ir<%gep>
  WIDEN store vp<%5>, ir<%mul>
  EMIT vp<%index.next> = add nuw vp<%3>, vp<%1>
  EMIT branch-on-count vp<%index.next>, vp<%2>
```

Clone recipes *IC* times

Plan only valid for interleave count = 2

<x1> vector loop

```
vector.body:
  EMIT vp<%6> = CANONICAL-INDUCTION ir<0>,
                                    vp<%index.next>
  ir<%iv> = WIDEN-INDUCTION  ir<0>, ir<1>, vp<%0>,
                             vp<%5>, vp<%step.add>
  EMIT vp<%step.add> = add ir<%iv>, vp<%5>
  vp<%7> = SCALAR-STEPS vp<%6>, ir<1>, vp<%0>
  CLONE ir<%gep> = getelementptr inbounds ir<%a>,
                                          vp<%7>
  WIDEN ir<%mul> = mul ir<%iv>, ir<3>
  WIDEN ir<%mul>.1 = mul vp<%step.add>, ir<3>
  vp<%8> = vector-pointer ir<%gep>
  vp<%9> = vector-pointer ir<%gep>, ir<1>
  WIDEN store vp<%8>, ir<%mul>
  WIDEN store vp<%9>, ir<%mul>.1
  EMIT vp<%index.next> = add nuw vp<%6>, vp<%1>
  EMIT branch-on-count vp<%index.next>, vp<%2>
```

Interleave Vector Iterations

Perform interleaving explicitly as VPlan transform

Plan valid for any interleave count

<x1> vector loop

```
vector.body:
  EMIT vp<%3> = CANONICAL-INDUCTION ir<0>,
                                    vp<%index.next>
  ir<%iv> = WIDEN-INDUCTION  ir<0>, ir<1>, vp<%0>
  vp<%4> = SCALAR-STEPS vp<%3>, ir<1>, vp<%0>
  CLONE ir<%gep> = getelementptr inbounds ir<%a>,
                                          vp<%4>
  WIDEN ir<%mul> = mul ir<%iv>, ir<3>
  vp<%5> = vector-pointer ir<%gep>
  WIDEN store vp<%5>, ir<%mul>
  EMIT vp<%index.next> = add nuw vp<%3>, vp<%1>
  EMIT branch-on-count vp<%index.next>, vp<%2>
```

Plan only valid for interleave count = 2

<x1> vector loop

```
vector.body:
  EMIT vp<%6> = CANONICAL-INDUCTION ir<0>,
                                    vp<%index.next>
  ir<%iv> = WIDEN-INDUCTION  ir<0>, ir<1>, vp<%0>,
                             vp<%5>, vp<%step.add>
  EMIT vp<%step.add> = add ir<%iv>, vp<%5>
  vp<%7> = SCALAR-STEPS vp<%6>, ir<1>, vp<%0>
  CLONE ir<%gep> = getelementptr inbounds ir<%a>,
                                          vp<%7>
  WIDEN ir<%mul> = mul ir<%iv>, ir<3>
  WIDEN ir<%mul>.1 = mul vp<%step.add>, ir<3>
  vp<%8> = vector-pointer ir<%gep>
  vp<%9> = vector-pointer ir<%gep>, ir<1>
  WIDEN store vp<%8>, ir<%mul>
  WIDEN store vp<%9>, ir<%mul>.1
  EMIT vp<%index.next> = add nuw vp<%6>, vp<%1>
  EMIT branch-on-count vp<%index.next>, vp<%2>
```

Interleave Vector Iterations

Simplifies ::execute for all recipes

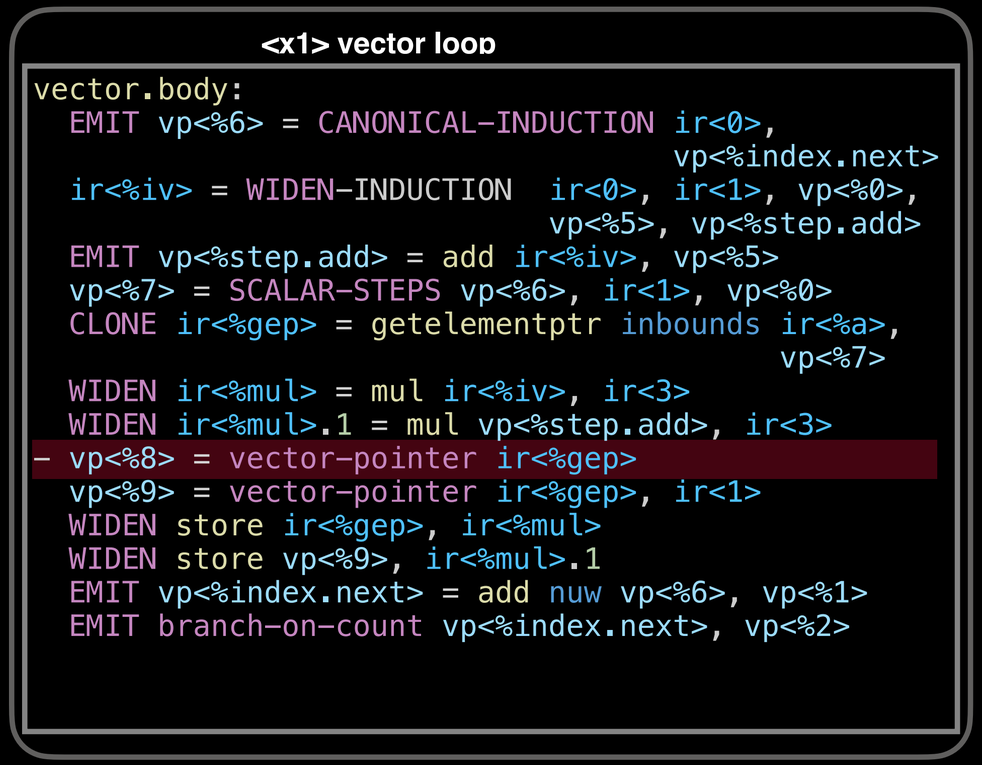
```
- for (unsigned Part = 0, UF = State.UF; Part < UF; ++Part) {
  // Generate IR for each part.
- }
```

Simplifies VPTransformState

```
struct DataState {
-    /// A type for vectorized values in the new loop. Each value from the
-    /// original loop, when vectorized, is represented by UF vector values in
-    /// the new unrolled loop, where UF is the unroll factor.
-    typedef SmallVector<Value *, 2> PerPartValuesTy;
-
-    DenseMap<VPValue *, PerPartValuesTy> PerPartOutput;
+    // Each value from the original loop, when vectorized, is represented by a
+    // vector value in the map.
+    DenseMap<VPValue *, Value *> VPV2Vector;
```

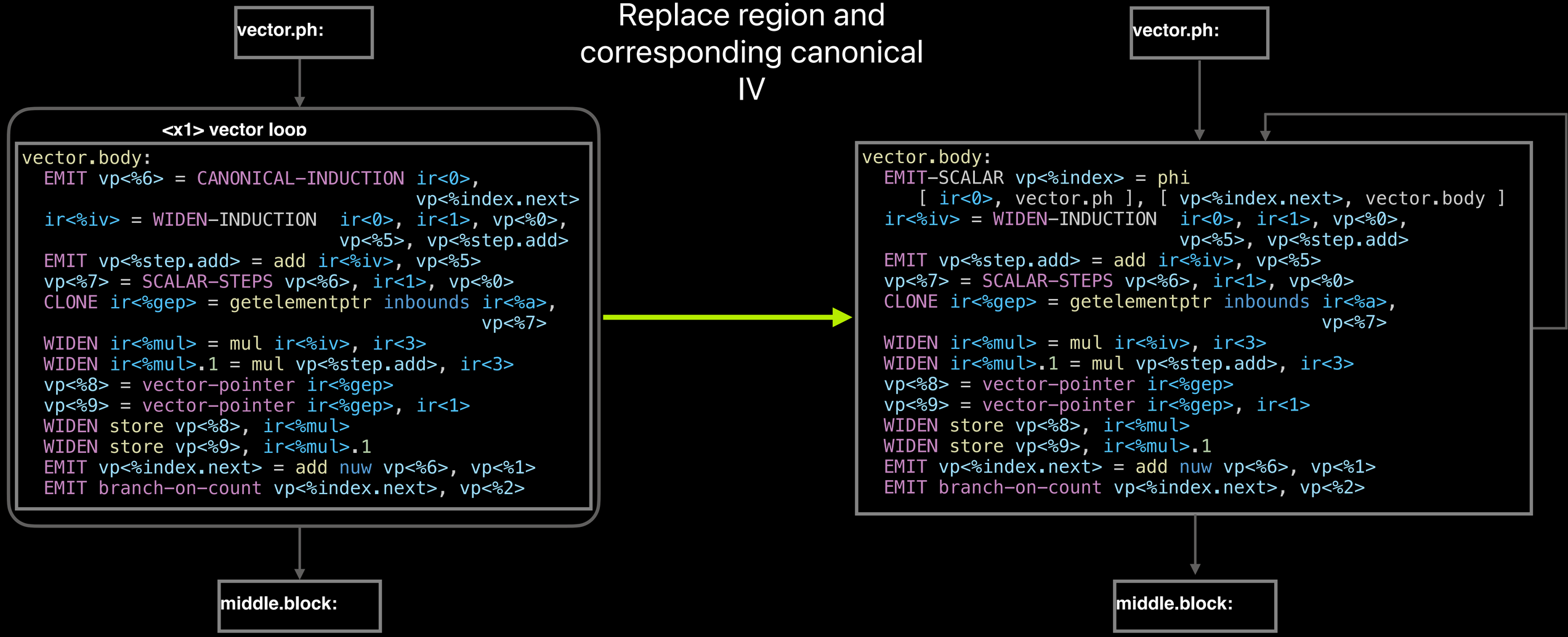
Interleave Vector Iterations

Enables additional simplifications



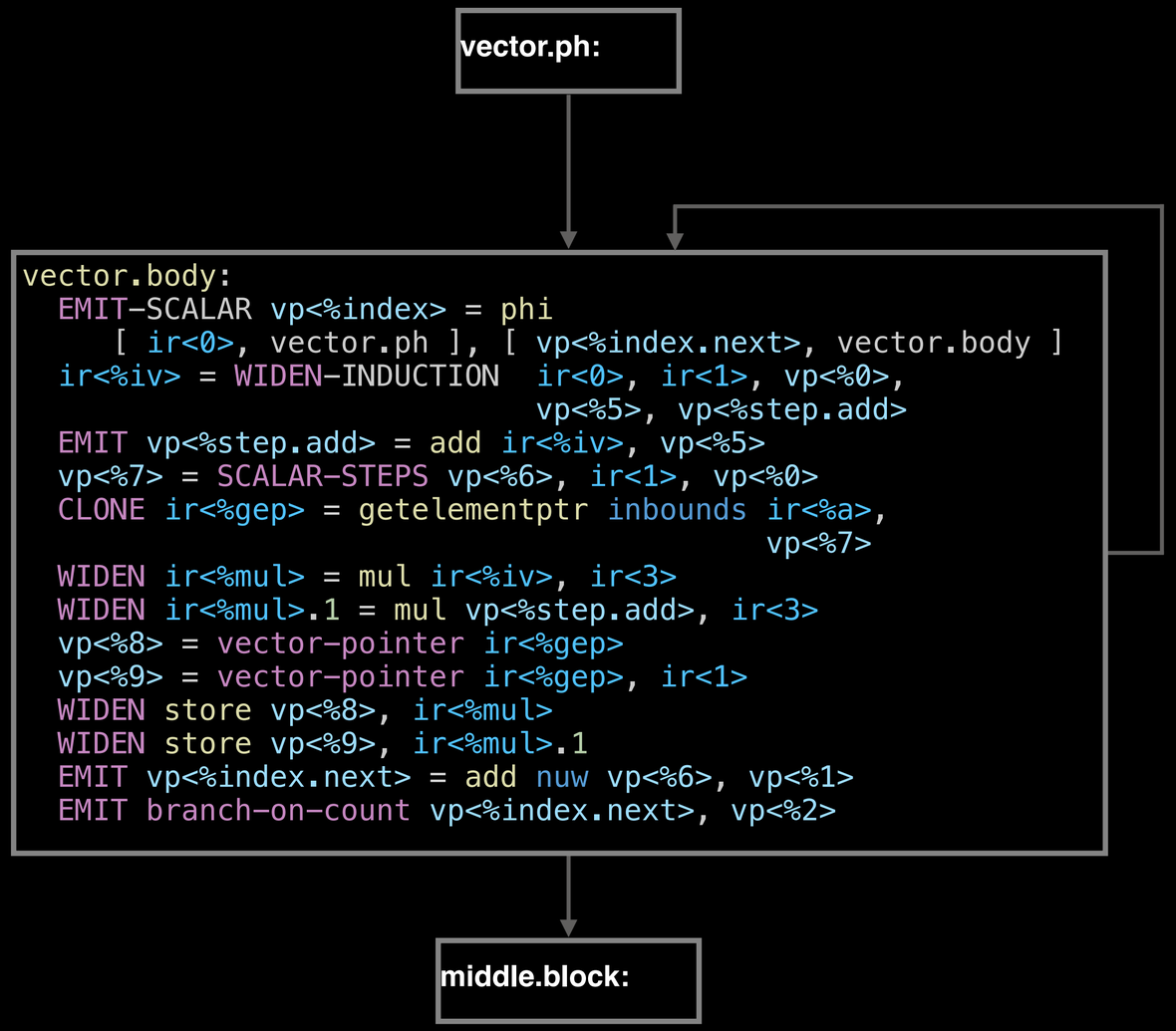
Some simplifications can
only be applied after interleaving

Dissolve Loop Regions



Dissolve Loop Regions

- Allows removing canonical IV if unused
- Allows replacing canonical IV, e.g. by a value based on the explicit vector length on RISCV



Teach VPlan to Compute and use Cost

Goal: Compute cost of VPlan after construction, instead of making cost-based decisions up-front.

Recipes compute their cost via ::computeCost()

```
/// Return the cost of this VPInstruction.
InstructionCost computeCost(ElementCount VF,
                            VPCostContext &Ctx) const override;
```

Blocks compute their cost via ::cost()

```
/// Return the cost of this VPBasicBlock.
InstructionCost cost(ElementCount VF, VPCostContext &Ctx) override;
```

Teach VPlan to Compute and use Cost

Enables accurate cost computations in presence of VPlan transforms

Simplifications

<x1> vector loop

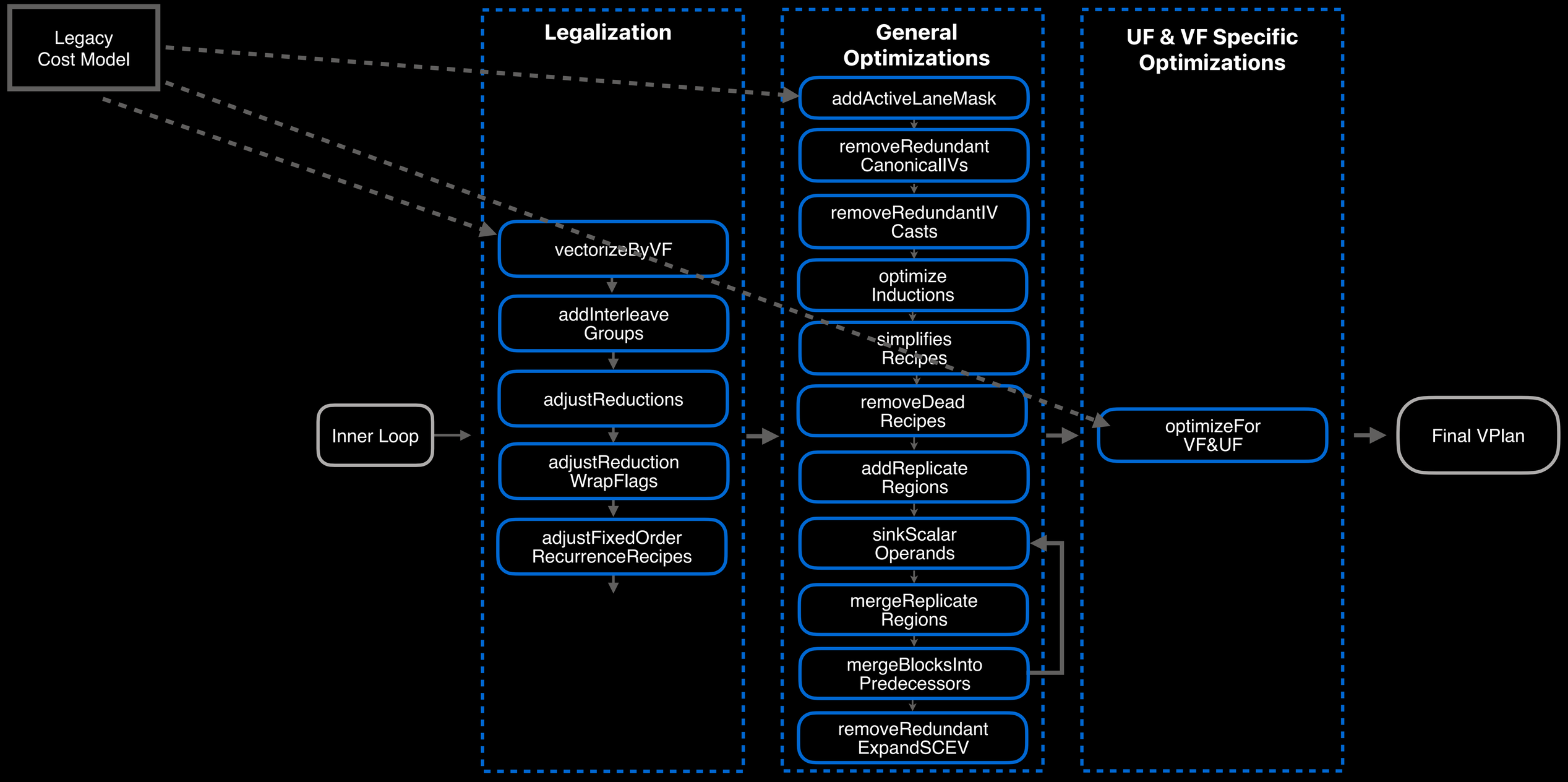
```
vector.body:
  EMIT vp<%2> = CANONICAL-INDUCTION
  WIDEN-INDUCTION %iv = phi 0, %iv.next
  vp<%4> = SCALAR-STEPS vp<%2>, ir<1>
  CLONE ir<%gep.a> = getelementptr ir<%a>,
                                   vp<%4>
- WIDEN ir<%mul> = mul ir<%iv>, ir<1>
  WIDEN store ir<%gep.a>, ir<%mul>
  EMIT vp<%7> = VF * UF + nuw vp<%2>
  EMIT branch-on-count vp<%7>, vp<%0>
```

Cost
0
1
1
0
1
1
0
0

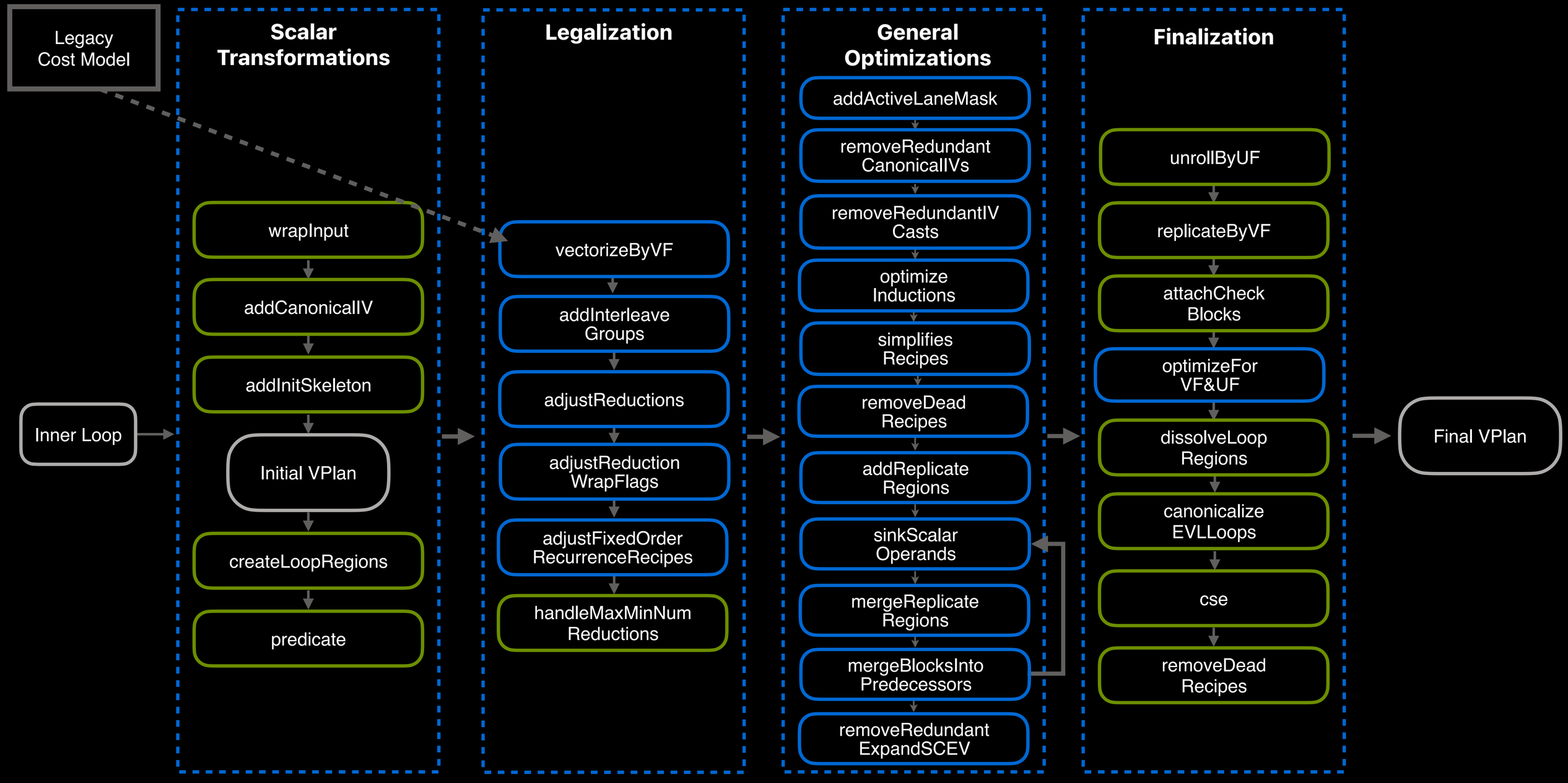
Legacy cost model requires (incomplete) special casing

```
- // If we're speculating on the stride being 1, the multiplication may
- // fold away.  We can generalize this for all operations using the notion
- // of neutral elements.  (TODO)
- if (I->getOpcode() == Instruction::Mul &&
-     ((TheLoop->isLoopInvariant(I->getOperand(0)) &&
-       PSE.getSCEV(I->getOperand(0))->isOne()) ||
-      (TheLoop->isLoopInvariant(I->getOperand(1)) &&
-       PSE.getSCEV(I->getOperand(1))->isOne())))
-   return 0;

- // Certain instructions can be cheaper to vectorize if they have a constant
- // second vector operand. One example of this are shifts on x86.
- Value *Op2 = I->getOperand(1);
- if (!isa<Constant>(Op2) && TheLoop->isLoopInvariant(Op2) &&
-     PSE.getSE()->isSCEVable(Op2->getType()) &&
-     isa<SCEVConstant>(PSE.getSCEV(Op2))) {
-   Op2 = cast<SCEVConstant>(PSE.getSCEV(Op2))->getValue();
- }
```

VPlan Transformation Pipeline ‘23

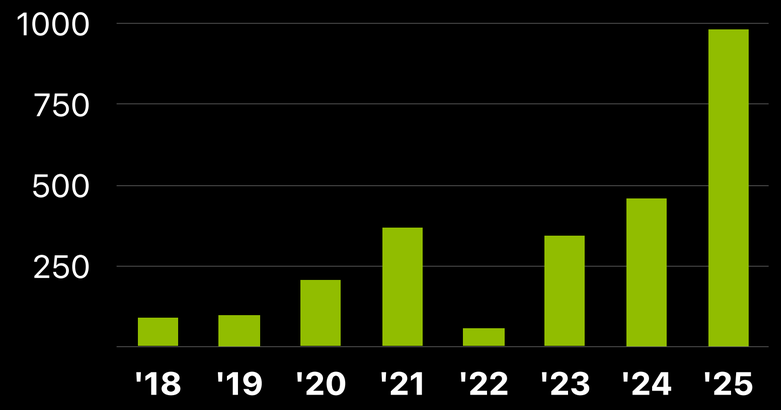


VPlan Transformation Pipeline ‘25

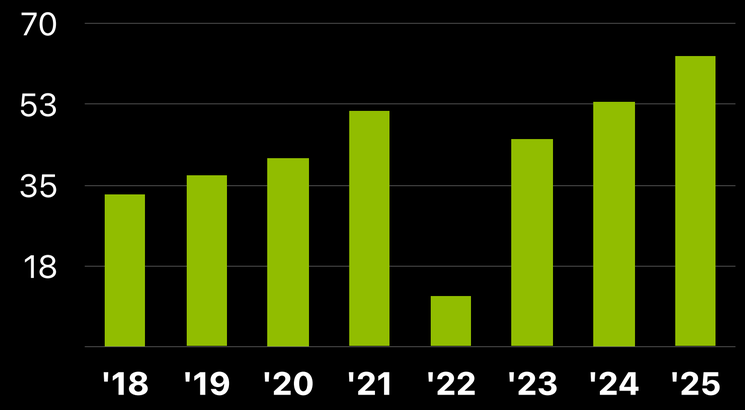


LoopVectorize Progress

Lots of improvements over the last 2 years!



📈 LV-related commits per year



👥 Committers per year

LoopVectorize Progress

FMax/FMin without FastMath

```
float max = -1e7f;
for (int iv = 0; iv < n; iv++)
  max = fmaxf(max, src[iv]);
return max;
```

```
vector.body:
  %index = phi i64 [ 0, %vector.ph ], [ %index.next, %
  %vec.phi = phi <4 x float> [ splat (float -1.000000e
[ %1, %vector.body ]
  %0 = getelementptr inbounds nuw float, ptr %src, i64
  %wide.load = load <4 x float>, ptr %0, align 4
  %1 = call <4 x float> @llvm.maxnum.v4f32(<4 x float>
%wide.load)
  %index.next = add nuw i64 %index, 4
  %2 = icmp eq i64 %index.next, %n.vec
  %3 = fcmp uno <4 x float> %wide.load, %wide.load
  %4 = freeze <4 x i1> %3
  %5 = call i1 @llvm.vector.reduce.or.v4i1(<4 x i1> %4
  %broadcast.splatinsert = insertelement <4 x i1> pois
  %broadcast.splat = shufflevector <4 x i1> %broadcast
poison, <4 x i32> zeroinitializer
  %6 = or i1 %5, %2
  br i1 %6, label %middle.block, label %vector.body, !

middle.block:
  %7 = select <4 x i1> %broadcast.splat, <4 x float> %
  %8 = select i1 %5, i64 %index, i64 %n.vec
  %9 = call float @llvm.vector.reduce.fmax.v4f32(<4 x
  %cmp.n = icmp eq i64 %n, %n.vec
```

Initial Early Exit Loops

```
for (int i = 0; i < 128; i++)
  if (src[i] == 10) return i;

return -1;
```

```
vector.body:
  %index = phi i64 [ 0, %vector.ph ], [ %index.next, %
  %0 = getelementptr inbounds i32, ptr %src, i64 %inde
  %wide.load = load <4 x i32>, ptr %0, align 4
  %1 = icmp eq <4 x i32> %wide.load, splat (i32 10)
  %index.next = add nuw i64 %index, 4
  %2 = freeze <4 x i1> %1
  %3 = call i1 @llvm.vector.reduce.or.v4i1(<4 x i1> %2
  %4 = icmp eq i64 %index.next, 128
  %5 = or i1 %3, %4
  br i1 %5, label %middle.split, label %vector.body, !

middle.split:
  br i1 %3, label %vector.early.exit, label %middle.bl

middle.block:
  br label %exit

vector.early.exit:
  br label %exit

exit:
  %p = phi i64 [ 1, %middle.block ], [ 0, %vector.earl
```

Find First Induction

```
for (int64_t i = 19999; i >= 0; i--)
   if (a[i] > 3) rdx = i;

return rdx;
```

```
vector.body:
  %index = phi i64 [ 0, %vector.ph ], [ %index.next, %vec
  %vec.ind = phi <4 x i64> [ <i64 19999, i64 19998, i64 1
%vector.ph ], [ %vec.ind.next, %vector.body ]
  %vec.phi = phi <4 x i64> [ splat (i64 92233720368547758
[ %4, %vector.body ]
  %offset.idx = sub i64 19999, %index
  %0 = getelementptr inbounds i64, ptr %a, i64 %offset.id
  %1 = getelementptr inbounds i64, ptr %0, i32 0
  %2 = getelementptr inbounds i64, ptr %1, i32 -3
  %wide.load = load <4 x i64>, ptr %2, align 8
  %reverse = shufflevector <4 x i64> %wide.load, <4 x i64
<i32 3, i32 2, i32 1, i32 0>
  %3 = icmp sgt <4 x i64> %reverse, splat (i64 3)
  %4 = select <4 x i1> %3, <4 x i64> %vec.ind, <4 x i64>
  %index.next = add nuw i64 %index, 4
  %vec.ind.next = add <4 x i64> %vec.ind, splat (i64 -4)
  %5 = icmp eq i64 %index.next, 20000
  br i1 %5, label %middle.block, label %vector.body, !llv

middle.block:
  %6 = call i64 @llvm.vector.reduce.smin.v4i64(<4 x i64>
  %rdx.select.cmp = icmp ne i64 %6, 9223372036854775807
  %rdx.select = select i1 %rdx.select.cmp, i64 %6, i64 33
```

LoopVectorize Progress

Partial Reductions

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

return dotp;
```

```
vector.body:
  %index = phi i64 [ 0, %vector.ph ], [ %index.next, %
  %vec.phi = phi <4 x i32> [ zeroinitializer, %vector.
%vector.body ]
  %0 = getelementptr inbounds nuw i8, ptr %A, i64 %ind
  %wide.load = load <16 x i8>, ptr %0, align 1
  %1 = sext <16 x i8> %wide.load to <16 x i32>
  %2 = getelementptr inbounds nuw i8, ptr %B, i64 %ind
  %wide.load13 = load <16 x i8>, ptr %2, align 1
  %3 = sext <16 x i8> %wide.load13 to <16 x i32>
  %4 = mul nsw <16 x i32> %3, %1
  %partial.reduce = tail call <4 x i32>
@llvm.vector.partial.reduce.add.v4i32.v16i32(<4 x i32>
%4)
  %index.next = add nuw i64 %index, 16
  %5 = icmp eq i64 %index.next, %n.vec
  br i1 %5, label %middle.block, label %vector.body

middle.block:
  %6 = tail call i32 @llvm.vector.reduce.add.v4i32(<4
  %cmp.n = icmp eq i64 %n.vec, %wide.trip.count
```

Tail-folding by Default for RISCV

```
for (int64_t i = 0; iv < n; i++)
  a[i] += v;
```

```
vector.body:
  %evl.based.iv = phi i64 [ 0, %vector.ph ], [ %index.
  %avl = phi i64 [ %n, %vector.ph ], [ %avl.next, %vec
  %0 = call i32 @llvm.experimental.get.vector.length.i
  %1 = getelementptr inbounds i64, ptr %a, i64 %evl.ba
  %vp.op.load = call <vscale x 2 x i64> @llvm.vp.load.
true), i32 %0)
  %2 = add <vscale x 2 x i64> %vp.op.load, %broadcast.
  call void @llvm.vp.store.nxv2i64.p0(<vscale x 2 x i6
i32 %0)
  %3 = zext i32 %0 to i64
  %index.evl.next = add i64 %3, %evl.based.iv
  %avl.next = sub nuw i64 %avl, %3
  %4 = icmp eq i64 %avl.next, 0
  br i1 %4, label %middle.block, label %vector.body
```

And many more

LoopVectorize Progress

Thanks to all the contributors!

Committers to LoopVectorize/VPlan code over the last 2 years

Aiden Grossman
Alex Bradbury
Alexandros Lamprineas
Alexey Bataev
Andrew Rogers
Anna Thomas
Arthur Eubanks
Benjamin Kramer
Benjamin Maxwell
Cameron McInally
Craig Topper
Daniil Fukalov
David Green
David Sherwood
Ellis Hoag
Elvis Wang
ErikHogeman
Finn Plummer
Florian Hahn
Florian Mayer
George Chaltas
Graham Hunter
Hans Wennborg
Hari Limaye
Hassnaa Hamdi
Igor Kirillov
James Chesterman
Jay Foad
Jeremy Morse
John Brawn
Jon Chesterfield
Jon Roelofs
Julian Nagele
Karlo Basioli
Kazu Hirata
Kerry McLaughlin
Kirill Stoimenov
Kolya Panchenko
Krzysztof Drewniak
LiqinWeng
Lou Knauer
Luke Lau
Maciej Gabka
Madhur Amilkanthwar
Martin Storsjö
Maryam Moghadas
Matthias Braun
Mel Chen
Michael Maitland
Min-Yih Hsu
Muhammad Omair Javaid
Nicholas Guy
Nicolai Hähnle
Nikita Popov
Nilanjana Basu
Ningning Shi(史宁宁)
Niwin Anto
Noah Goldstein
Paschalis Mpeis
Patrick O'Neill
Paul Kirth
Paul Walker
Philip Reames
Pietro Ghiglio
Piotr Fusik
Rahul Joshi
Ramkumar Ramachandra
Sam Tebbs
Samuel Tebbs
Sander de Smalen
Shao-Ce SUN
Shih-Po Hung
Simon Pilgrim
Sjoerd Meijer
Stephen Tozer
Sterling-Augustine
Tobias Stadler
Troy Butler
Vitaly Buka
Walter Lee
Yingwei Zheng
Youngsuk Kim
Zequan Wu
calebwat
chrisPyr
offsake