Target-aware vectorization for irregular loops or instruction patterns

Wei Wei, Mindong Chen
BiSheng Compiler Team

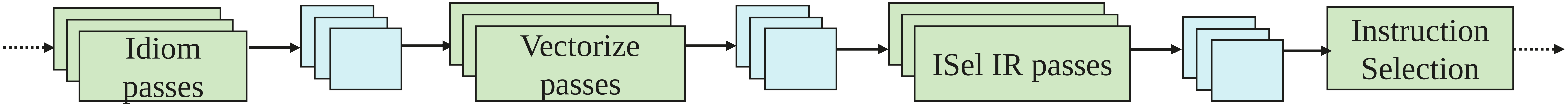
SIMD & Vectorization

- **Parallelism** – key to CPU performance in post-Moore's Law era
  - ✓ Thread-level Parallel. (TLP) – multi-core and multi-threading
  - ✓ Instruction-level Parallel. (ILP) – OOO and superscalar
  - ✓ Data-level Parallel. (DLP) – **Single-Instruction Multi-Data (SIMD)**
- **SIMD operations, or ISA**
  - ✓ Intel: MMX, SSE, AVX2, AVX512, AVX10, …
  - ✓ PPC: VMX128, VSX, Altivec, …
  - ✓ ARM: Neon, SVE, SVE2, …
- **SIMD generation**
  - ✓ **Assembly manually/compiler intrinsics**: requires expert experience or developers familiar with hardware ISA
  - ✓ **Compiler automatic vectorization:** loop vectorization, SLP(superword level parallelism)

    Two assumptions: 1) SIMD instruction performs isomorphic operations across all lanes.

    2) SIMD instruction applies the operations elementwise without cross-lane operation.
- **Irregular/complicated SIMD(non-SIMD)** – violate above two assumptions, like operations cross-lane reassociated
  - ✓ COMPLEX, COMPACT, DOT-PRODUCT, HISTCNT, MINMAX… …
  - ✓ Vector library function

SIMD Passes Pipeline



Idiom passes

Vectorize passes

ISel IR passes

Instruction Selection

LoopIdiom

AArch64Idiom

HexagonIdiom

…

LoopVectorize

SLPVectorizer

VectorCombine

…

(target) IRPasses

CodeGenPrepare

(target) PreISel

…

SelectionDAG

GlobalISel

FastISel

…

“Target Aware” Implementation?

Loop idiom recognition：

[RFC] Vector math function loop idiom recognition - IR & Optimizations / Loop Optimizations - LLVM Discussion Forums

[RFC] CRC Recognition in LoopIdiomRecognizer - IR & Optimizations / Loop Optimizations - LLVM Discussion Forums

[AArch64] Add an AArch64 pass for loop idiom transformations (#72273) · llvm/llvm-project@c714846 · GitHub

VPlan：

[RFC] Vectorization support for histogram count operations - IR & Optimizations / Loop Optimizations - LLVM Discussion Forums

⚙ D158836 [LoopVectorize] Vectorize the compact pattern (llvm.org)

Isel IR Passes：

[AArch64] Generate DOT instructions from matching IR by huntergr-arm · Pull Request #69583 · llvm/llvm-project · GitHub

⚙ D114174 [ARM][CodeGen] Add support for complex deinterleaving (llvm.org)

⚙ D129066 [AArch64][CodeGen] Add AArch64 support for complex deinterleaving (llvm.org)

Instruction selection：

⚙ D49636 [X86] Add matching for another pattern of PMADDWD. (llvm.org)

⚙ D49829 [X86] Add pattern matching for PMADDUBSW (llvm.org)

[AArch64] Lower mathlib call ldexp into fscale when sve is enabled by huhu233 · Pull Request #67552 · llvm/llvm-project · GitHub

Loop Idiom Recognition, Pros and Cons

- **LoopIdiomRecognize pass:** transforms simple loops into a non-loop form

✓ countable loop: MemCpy， MemSet

✓ non-countable loop: Popcnt(Aarch64, PPC,X86, AMDGPU, RISCV with zbb… …)， CTLZ/CTTZ…

- **Target-specific loop idiom recognition:**

✓ HexagonLoopIdiomRecognition: memcpy, memmove, polynomial multiply(pmpyw)

✓ AArch64LoopIdiomTransform: memcmp like pattern(first-faulting loads)

- **Pros:**

✓ **Early enough** in the pipeline, not affected by later optimizations

✓ Target specific or target independent idioms, flexible and easy to implement

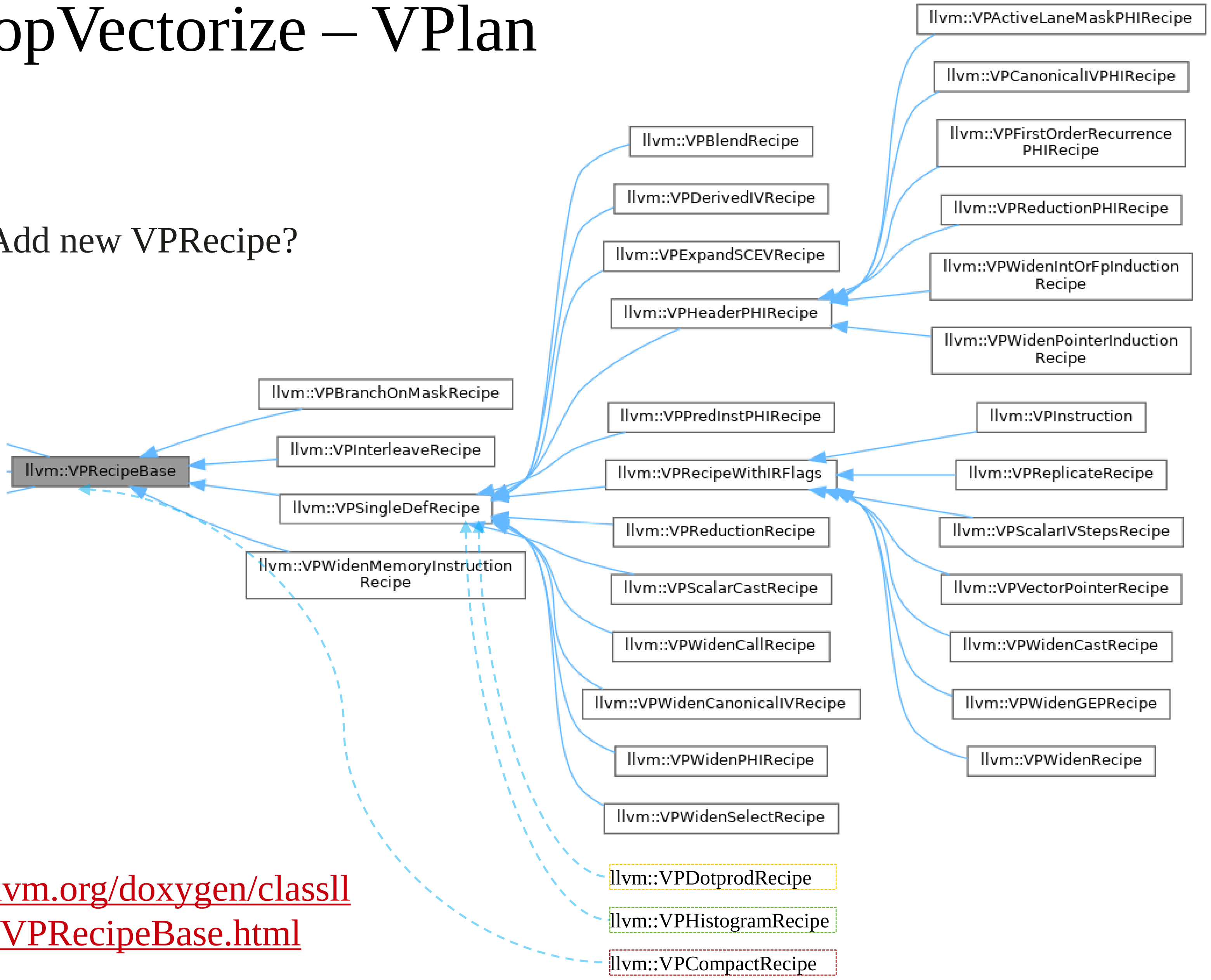
- **Cons:**

✓ Matching the idiom code was hard and pretty fragile, the actual IR was changing (by instcombine, or other passes)

✓ **Too early** in the pipeline, may lose many opportunities for subsequent loop optimization

LoopVectorize – VPlan

- Add new VPRecipe?



llvm::VPRecipeBase

llvm::VPBranchOnMaskRecipe

llvm::VPInterleaveRecipe

llvm::VPSingleDefRecipe

llvm::VPWidenMemoryInstruction Recipe

llvm::VPBlendRecipe

llvm::VPDerivedIVRecipe

llvm::VPExpandSCEVRecipe

llvm::VPHeaderPHIRecipe

llvm::VPPredInstPHIRecipe

llvm::VPRecipeWithIRFlags

llvm::VPReductionRecipe

llvm::VPScalarCastRecipe

llvm::VPWidenCallRecipe

llvm::VPWidenCanonicalIVRecipe

llvm::VPWidenPHIRecipe

llvm::VPWidenSelectRecipe

llvm::VPActiveLaneMaskPHIRecipe

llvm::VPCanonicalIVPHIRecipe

llvm::VPFirstOrderRecurrence PHIRecipe

llvm::VPReductionPHIRecipe

llvm::VPWidenIntOrFpInduction Recipe

llvm::VPWidenPointerInduction Recipe

llvm::VPInstruction

llvm::VPReplicateRecipe

llvm::VPScalarIVStepsRecipe

llvm::VPVectorPointerRecipe

llvm::VPWidenCastRecipe

llvm::VPWidenGEPRecipe

llvm::VPWidenRecipe

llvm::VPDotprodRecipe

llvm::VPHistogramRecipe

llvm::VPCompactRecipe

https://llvm.org/doxygen/classllvm_1_1VPRecipeBase.html

Dotprod:

(sum: int; a, b: char*)

```
for (int i = 0; i < N; i++) {
   sum += a[i] * b[i];
}
```

Histogram:

```
for (int i = 0; i < N; ++i) {
   buckets[indices[i]]++;
}
```

Compact:

```
for(i=0; i<N; i++) {
   if(x[i]<a) ref[n++]=B[i];
}
```

SVE COMPACT

COMPACT <Zd>.<T>, <Pg>, <Zn>.<T>

Shuffle active elements of vector to the right and fill with zero

```
for(i=0; i<N; i++)
  if(x[i]<a) ref[n++]=B[i];
```

b[i]	5	7	3	2
x[i] < a	1	0	1	0
ref	0	0	5	3

SVE COMPACT

COMPACT <Zd>.<T>, <Pg>, <Zn>.<T>
Shuffle active elements of vector to the right and fill with zero

```
for(i=0; i<N; i++)
  if(x[i]<a) ref[n++]=B[i];
```

b[i]	5	7	3	2
x[i] < a	1	0	1	0
ref	0	0	5	3

- Besides AArch64, x86 AVX512 also supports VPCOMPRESS*
- But until now, LLVM cannot automatically vectorize the above pattern in neither LoopVectorize nor others passes.

```
$ opt -passes=loop-vectorize compact.ll -debug-only=loop-vectorize -force-vector-width=4
LV: Not vectorizing: Found an unidentified PHI   %n.013 = phi i32 [ 0, %for.body.preheader ], [ %n.1, %for.inc ]
remark: <unknown>:0:0: loop not vectorized: value that could not be identified as reduction is used outside the loop
LV: Can't vectorize the instructions or CFG
LV: Not vectorizing: Cannot prove legality.
```

- LLVM already have **llvm.masked.compressstore.*** support, which is exactly a scenario to make use of COMPACT

Why LoopVectorize fails?

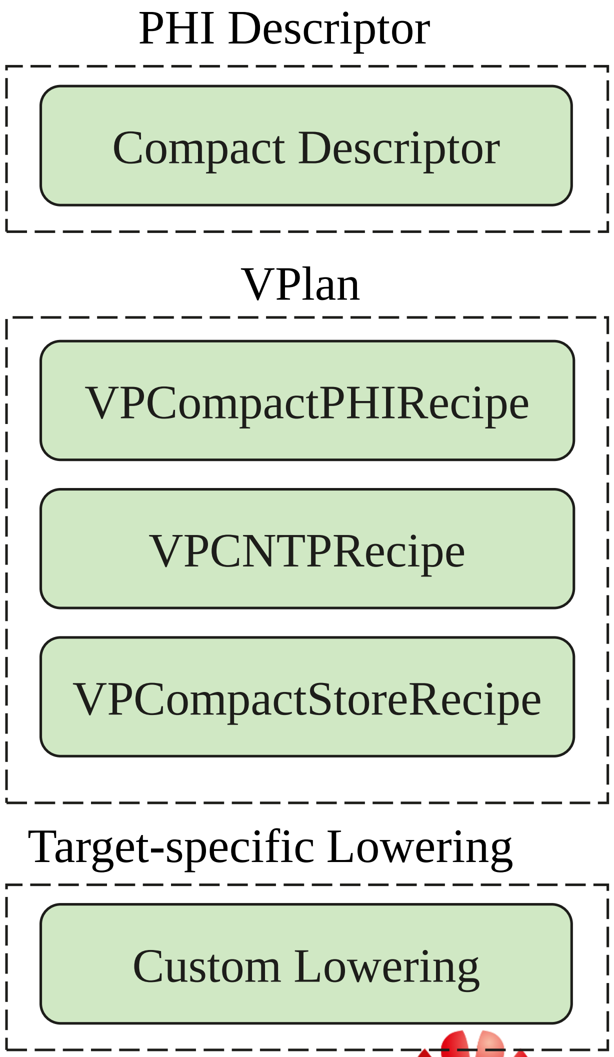
- Currently, only limited patterns of phi are supported
  - Induciton phis
  - Reduction phis
  - Fixed-order Recurrence phis
  - If not the above ones, STUCK!

- The presence of unrecognized phis hints the control flow too complex to analyze
  - Need to teach the Loop Vectorizer to understand the control flow

```
for.body:                                          ; preds = %for.body.preheader, %for.inc
  %indvars.iv = phi i64 [ 0, %for.body.preheader ], [ %indvars.iv.next,
%for.inc ]
  %n.013 = phi i32 [ 0, %for.body.preheader ], [ %n.1, %for.inc ]
  %arrayidx = getelementptr inbounds i32, ptr %comp, i64 %indvars.iv
  %0 = load i32, ptr %arrayidx, align 4
  %cmp1 = icmp slt i32 %0, %a
  br i1 %cmp1, label %if.then, label %for.inc

if.then:                                           ; preds = %for.body
  %arrayidx3 = getelementptr inbounds i32, ptr %B, i64 %indvars.iv
  %1 = load i32, ptr %arrayidx3, align 4
  %inc = add nsw i32 %n.013, 1
  %idxprom4 = sext i32 %n.013 to i64
  %arrayidx5 = getelementptr inbounds i32, ptr %Out_ref, i64 %idxprom4
  store i32 %1, ptr %arrayidx5, align 4
  br label %for.inc

for.inc:                                           ; preds = %for.body, %if.then
  %n.1 = phi i32 [ %inc, %if.then ], [ %n.013, %for.body ]
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %exitcond.not = icmp eq i64 %indvars.iv.next, %wide.trip.count
  br i1 %exitcond.not, label %for.end, label %for.body
```

What we need to do?

```
for.body:                                          ; preds = %for.body.preheader, %for.inc
  %indvars.iv = phi i64 [ 0, %for.body.preheader ], [ %indvars.iv.next,
%for.inc ]
  %n.013 = phi i32 [ 0, %for.body.preheader ], [ %n.1, %for.inc ]
  %arrayidx = getelementptr inbounds i32, ptr %comp, i64 %indvars.iv
  %0 = load i32, ptr %arrayidx, align 4
  %cmp1 = icmp slt i32 %0, %a
  br i1 %cmp1, label %if.then, label %for.inc

if.then:                                           ; preds = %for.body
  %arrayidx3 = getelementptr inbounds i32, ptr %B, i64 %indvars.iv
  %1 = load i32, ptr %arrayidx3, align 4
  %inc = add nsw i32 %n.013, 1
  %idxprom4 = sext i32 %n.013 to i64
  %arrayidx5 = getelementptr inbounds i32, ptr %Out_ref, i64 %idxprom4
  store i32 %1, ptr %arrayidx5, align 4
  br label %for.inc

for.inc:                                           ; preds = %for.body, %if.then
  %n.1 = phi i32 [ %inc, %if.then ], [ %n.013, %for.body ]
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %exitcond.not = icmp eq i64 %indvars.iv.next, %wide.trip.count
  br i1 %exitcond.not, label %for.end, label %for.body
```



New Header PHI Descriptor

```
for.body:                                   ; preds = %for.body.preheader, %for.inc
  %n.013 = phi i32 [ 0, %for.body.preheader ], [ %n.1, %for.inc ]



  %cmp1 = icmp slt i32 %0, %a

if.then:                                    ; preds = %for.body


  %inc = add nsw i32 %n.013, 1
  %idxprom4 = sext i32 %n.013 to i64
  %arrayidx5 = getelementptr inbounds i32, ptr %Out_ref, i64 %idxprom4
  store i32 %1, ptr %arrayidx5, align 4
  br label %for.inc

for.inc:                                    ; preds = %for.body, %if.then
  %n.1 = phi i32 [ %inc, %if.then ], [ %n.013, %for.body ]


  br i1 %exitcond.not, label %for.end, label %for.body
```

Compact Descriptor

Compact Header PHI

New Header PHI Descriptor

```
for.body:                                   ; preds = %for.body.preheader, %for.inc
  %n.013 = phi i32 [ 0, %for.body.preheader ], [ %n.1, %for.inc ]



  %cmp1 = icmp slt i32 %0, %a

if.then:                                    ; preds = %for.body


  %inc = add nsw i32 %n.013, 1
  %idxprom4 = sext i32 %n.013 to i64
  %arrayidx5 = getelementptr inbounds i32, ptr %Out_ref, i64 %idxprom4
  store i32 %1, ptr %arrayidx5, align 4
  br label %for.inc

for.inc:                                    ; preds = %for.body, %if.then
  %n.1 = phi i32 [ %inc, %if.then ], [ %n.013, %for.body ]



  br i1 %exitcond.not, label %for.end, label %for.body
```

🟩 Compact Descriptor

🟦 Compact Header PHI

🟥 Guard

New Header PHI Descriptor

```
for.body:                                    ; preds = %for.body.preheader, %for.inc
  %n.013 = phi i32 [ 0, %for.body.preheader ], [ %n.1, %for.inc ]



  %cmp1 = icmp slt i32 %0, %a

if.then:                                     ; preds = %for.body


  %inc = add nsw i32 %n.013, 1
  %idxprom4 = sext i32 %n.013 to i64
  %arrayidx5 = getelementptr inbounds i32, ptr %Out_ref, i64 %idxprom4
  store i32 %1, ptr %arrayidx5, align 4
  br label %for.inc

for.inc:                                     ; preds = %for.body, %if.then
  %n.1 = phi i32 [ %inc, %if.then ], [ %n.013, %for.body ]


  br i1 %exitcond.not, label %for.end, label %for.body
```

Compact Descriptor

Compact Header PHI

Guard

Conditional Inc

New Header PHI Descriptor

```
for.body:                                     ; preds = %for.body.preheader, %for.inc
  %n.013 = phi i32 [ 0, %for.body.preheader ], [ %n.1, %for.inc ]



  %cmp1 = icmp slt i32 %0, %a

if.then:                                      ; preds = %for.body


  %inc = add nsw i32 %n.013, 1
  %idxprom4 = sext i32 %n.013 to i64
  %arrayidx5 = getelementptr inbounds i32, ptr %Out_ref, i64 %idxprom4
  store i32 %1, ptr %arrayidx5, align 4
  br label %for.inc

for.inc:                                      ; preds = %for.body, %if.then
  %n.1 = phi i32 [ %inc, %if.then ], [ %n.013, %for.body ]



  br i1 %exitcond.not, label %for.end, label %for.body
```

Compact Descriptor

Compact Header PHI

Guard

Conditional Inc

Memory Address

New Header PHI Descriptor

```
for.body:                                  ; preds = %for.body.preheader, %for.inc

  %n.013 = phi i32 [ 0, %for.body.preheader ], [ %n.1, %for.inc ]



  %cmp1 = icmp slt i32 %0, %a

if.then:                                   ; preds = %for.body


  %inc = add nsw i32 %n.013, 1
  %idxprom4 = sext i32 %n.013 to i64
  %arrayidx5 = getelementptr inbounds i32, ptr %Out_ref, i64 %idxprom4
  store i32 %1, ptr %arrayidx5, align 4
  br label %for.inc

for.inc:                                   ; preds = %for.body, %if.then
  %n.1 = phi i32 [ %inc, %if.then ], [ %n.013, %for.body ]


  br i1 %exitcond.not, label %for.end, label %for.body
```

Compact Descriptor

Compact Header PHI

Guard

Conditional Inc

Memory Address

Compact Store

New Header PHI Descriptor

```
for.body:                                    ; preds = %for.body.preheader, %for.inc
  %n.013 = phi i32 [ 0, %for.body.preheader ], [ %n.1, %for.inc ]



  %cmp1 = icmp slt i32 %0, %a

if.then:                                     ; preds = %for.body


  %inc = add nsw i32 %n.013, 1
  %idxprom4 = sext i32 %n.013 to i64
  %arrayidx5 = getelementptr inbounds i32, ptr %Out_ref, i64 %idxprom4
  store i32 %1, ptr %arrayidx5, align 4
  br label %for.inc

for.inc:                                     ; preds = %for.body, %if.then
  %n.1 = phi i32 [ %inc, %if.then ], [ %n.013, %for.body ]


  br i1 %exitcond.not, label %for.end, label %for.body
```

Compact Descriptor

Compact Header PHI

Guard

Conditional Inc

Memory Address

Compact Store

Compact Latch PHI

New VPlan Recipes

Compact Descriptor

Compact Header PHI

Guard

Conditional Inc

Memory Address

Compact Store

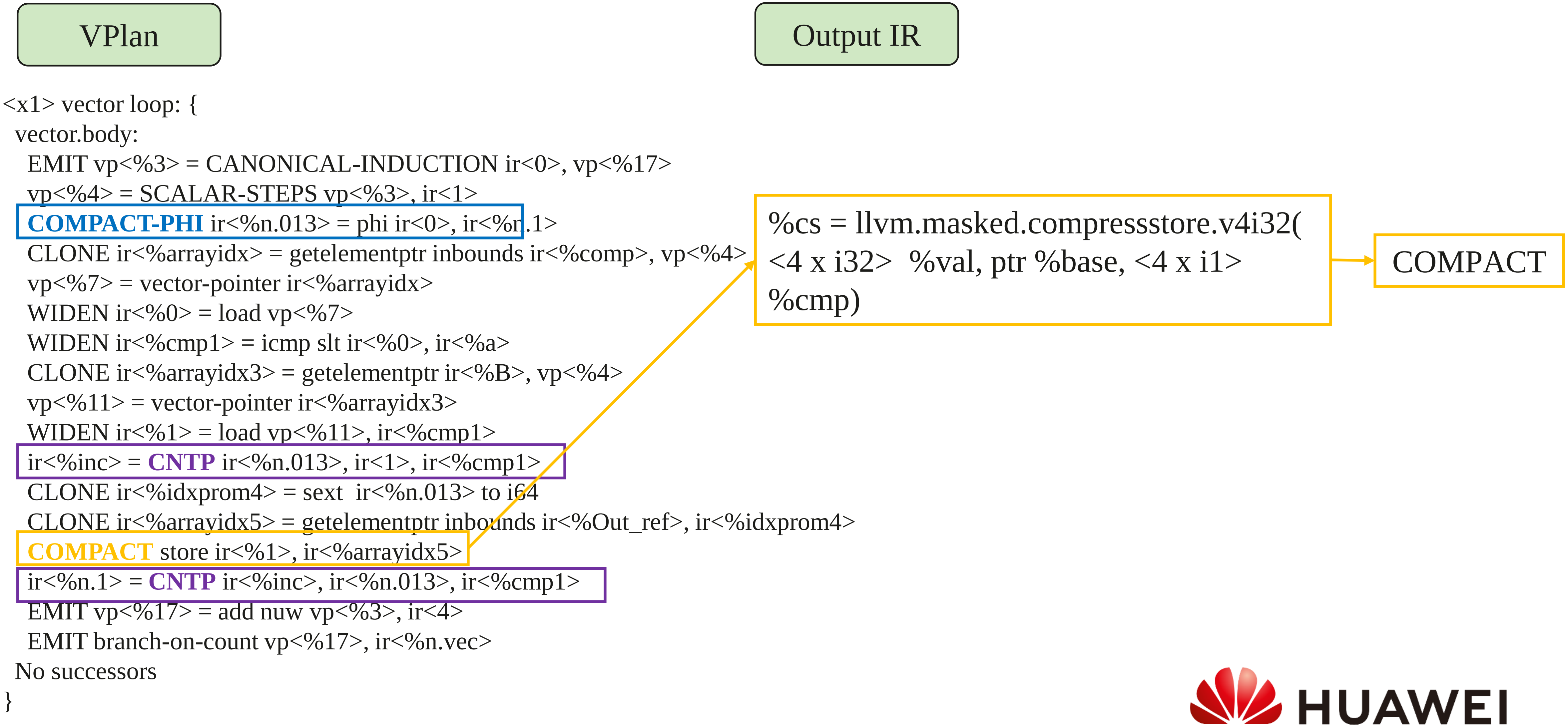
Compact Latch PHI

New VPlan Recipes

Compact Descriptor	New VPlan Recipes
Compact Header PHI	VPCompactPHIRecipe
Guard	
Conditional Inc	VPCNTPRecipe
Memory Address	
Compact Store	VPCompactRecipe
Compact Latch PHI	

```
<x1> vector loop: {
  vector.body:
    EMIT vp<%3> = CANONICAL-INDUCTION ir<0>, vp<%17>
    vp<%4> = SCALAR-STEPS vp<%3>, ir<1>
    COMPACT-PHI ir<%n.013> = phi ir<0>, ir<%n.1>
    CLONE ir<%arrayidx> = getelementptr inbounds ir<%comp>, vp<%4>
    vp<%7> = vector-pointer ir<%arrayidx>
    WIDEN ir<%0> = load vp<%7>
    WIDEN ir<%cmp1> = icmp slt ir<%0>, ir<%a>
    CLONE ir<%arrayidx3> = getelementptr ir<%B>, vp<%4>
    vp<%11> = vector-pointer ir<%arrayidx3>
    WIDEN ir<%1> = load vp<%11>, ir<%cmp1>
    ir<%inc> = CNTP ir<%n.013>, ir<1>, ir<%cmp1>
    CLONE ir<%idxprom4> = sext  ir<%n.013> to i64
    CLONE ir<%arrayidx5> = getelementptr inbounds ir<%Out_ref>, ir<%idxprom4
    COMPACT store ir<%1>, ir<%arrayidx5>
    ir<%n.1> = CNTP ir<%inc>, ir<%n.013>, ir<%cmp1>
    EMIT vp<%17> = add nuw vp<%3>, ir<4>
    EMIT branch-on-count vp<%17>, ir<%n.vec>
  No successors
}
```

Transform VPlan to IR



VPlan

Output IR

```
<x1> vector loop: {
  vector.body:
    EMIT vp<%3> = CANONICAL-INDUCTION ir<0>, vp<%17>
    vp<%4> = SCALAR-STEPS vp<%3>, ir<1>
    COMPACT-PHI ir<%n.013> = phi ir<0>, ir<%n.1>
    CLONE ir<%arrayidx> = getelementptr inbounds ir<%comp>, vp<%4>
    vp<%7> = vector-pointer ir<%arrayidx>
    WIDEN ir<%0> = load vp<%7>
    WIDEN ir<%cmp1> = icmp slt ir<%0>, ir<%a>
    CLONE ir<%arrayidx3> = getelementptr ir<%B>, vp<%4>
    vp<%11> = vector-pointer ir<%arrayidx3>
    WIDEN ir<%1> = load vp<%11>, ir<%cmp1>
    ir<%inc> = CNTP ir<%n.013>, ir<1>, ir<%cmp1>
    CLONE ir<%idxprom4> = sext  ir<%n.013> to i64
    CLONE ir<%arrayidx5> = getelementptr inbounds ir<%Out_ref>, ir<%idxprom4>
    COMPACT store ir<%1>, ir<%arrayidx5>
    ir<%n.1> = CNTP ir<%inc>, ir<%n.013>, ir<%cmp1>
    EMIT vp<%17> = add nuw vp<%3>, ir<4>
    EMIT branch-on-count vp<%17>, ir<%n.vec>
  No successors
}
```

```
%cs = llvm.masked.compressstore.v4i32(
<4 x i32>  %val, ptr %base, <4 x i1>
%cmp)
```

COMPACT

Transform VPlan to IR

VPlan

```
<x1> vector loop: {
  vector.body:
    EMIT vp<%3> = CANONICAL-INDUCTION ir<0>, vp<%17>
    vp<%4> = SCALAR-STEPS vp<%3>, ir<1>
    COMPACT-PHI ir<%n.013> = phi ir<0>, ir<%n.1>
    CLONE ir<%arrayidx> = getelementptr inbounds ir<%comp>, vp<%4>
    vp<%7> = vector-pointer ir<%arrayidx>
    WIDEN ir<%0> = load vp<%7>
    WIDEN ir<%cmp1> = icmp slt ir<%0>, ir<%a>
    CLONE ir<%arrayidx3> = getelementptr ir<%B>, vp<%4>
    vp<%11> = vector-pointer ir<%arrayidx3>
    WIDEN ir<%1> = load vp<%11>, ir<%cmp1>
    ir<%inc> = CNTP ir<%n.013>, ir<1>, ir<%cmp1>
    CLONE ir<%idxprom4> = sext  ir<%n.013> to i64
    CLONE ir<%arrayidx5> = getelementptr inbounds ir<%Out_ref>, ir<%idxprom4>
    COMPACT store ir<%1>, ir<%arrayidx5>
    ir<%n.1> = CNTP ir<%inc>, ir<%n.013>, ir<%cmp1>
    EMIT vp<%17> = add nuw vp<%3>, ir<4>
    EMIT branch-on-count vp<%17>, ir<%n.vec>
  No successors
}
```

Output IR

```
%cs = llvm.masked.compressstore.v4i32(
<4 x i32>  %val, ptr %base, <4 x i1>
%cmp)
```

COMPACT

```
%prom = zext <4 x i1> %cmp to <4 x i32>
%cntp = llvm.vector.reduce.add.v4i32(<4 x i32> %prom)
```

CNTP

- GitHub POC : https://github.com/llvm/llvm-project/pull/68980

HUAWEI

Can be generalized further

- Now we have support vectorization of:

```
for(i=0; i<N; i++)
  if(x[i]<a) ref[n++]=B[i];
```

- But the following patterns still fail:

```
j = -1;
for (int i = 0; i < LEN_1D; i++)
    if (a[i] < (real_t)0.) {
        j = i;
```

Example: TSVC s331

CLASTB <R><dn>, <Pg>, <R><dn>, <Zm>.<T>
Conditionally extract last element to general-purpose register.

```
for(i=0; i<N; i++)
  if(x[i]<a) sum += n++;
```

```
for.body4:                                ; preds = %for.cond1.preheader, %for.body4
  %indvars.iv = phi i64 [ %indvars.iv.next, %for.body4 ], [ 0, %for.cond1.preheader ]
  %j.015 = phi i32 [ %spec.select, %for.body4 ], [ -1, %for.cond1.preheader ]
  %arrayidx = getelementptr inbounds i32, ptr %a, i64 %indvars.iv
  %0 = load i32, ptr %arrayidx, align 4, !tbaa !8
  %cmp5 = icmp slt i32 %0, 0
  %1 = trunc i64 %indvars.iv to i32
  %spec.select = select i1 %cmp5, i32 %1, i32 %j.015
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %exitcond.not = icmp eq i64 %indvars.iv.next, %wide.trip.count
  br i1 %exitcond.not, label %for.cond.cleanup3, label %for.body4, !llvm.loop !12
```

Control flow simplified by SimplifyCFG pass

- SELECT is generated by if-conversion to simplify CFG, which was important for the legacy loop vectorizer, but for VPlan?
- Currently, phis with select input are modeled in recurrence recipes, while those with "unsafe" control flow unrecognized, it might be worth thinking about unifying them, given the similar underneath control flow.

Low-hanging fruit: one-to-one native instruction

- SVE FSCALE is a native instruction for ldexp("load exponent"), and LLVM already supports ‘llvm.ldexp.*’intrinsic

FSCALE <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
Floating-point adjust exponent by vector (predicated)

num	5	7	3	2
exp	1	2	3	4
mask	1	0	1	0
result	$5*2^1$	0	$3*2^3$	0

```
for (int i = 0 ; i < n; ++i) {
   sum += ldexp(a[i], exps[i]);
 }
```

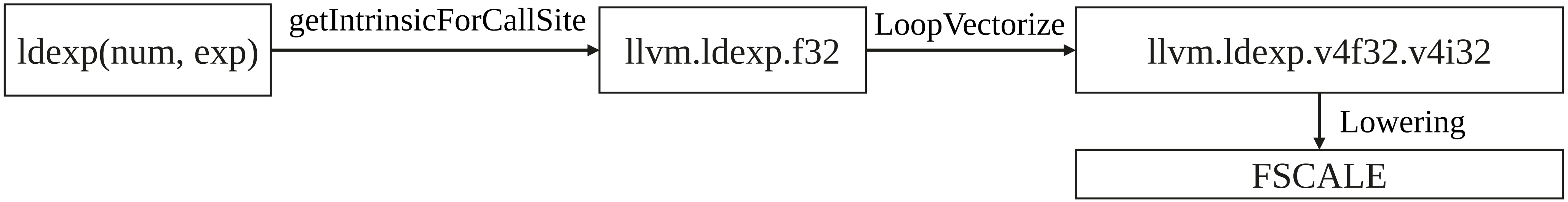
LV: Not vectorizing: **Found a non-intrinsic callsite**
%call = tail call double @ldexp(double noundef %0, i32 noundef %1) #2

Low-hanging fruit: one-to-one native instruction

- SVE FSCALE is a native instruction for ldexp("load exponent"), and LLVM already supports ‘llvm.ldexp.*’intrinsic

FSCALE <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
Floating-point adjust exponent by vector (predicated)

num	5	7	3	2
exp	1	2	3	4
mask	1	0	1	0
result	$5*2^1$	0	$3*2^3$	0

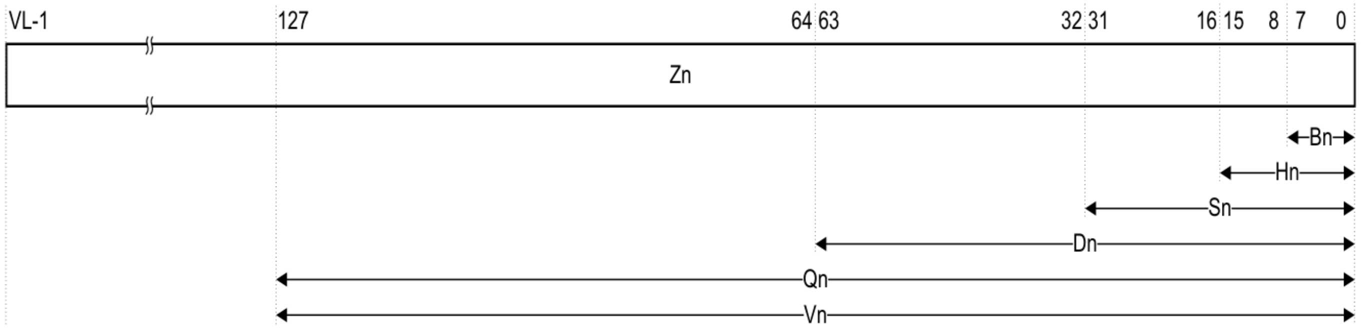


Low-hanging fruit: one-to-one native instruction

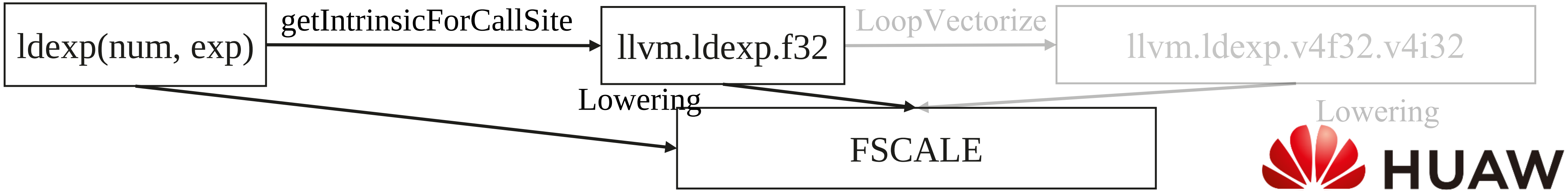
- SVE FSCALE is a native instruction for ldexp("load exponent"), and LLVM already supports ‘llvm.ldexp.*’intrinsic

FSCALE <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
Floating-point adjust exponent by vector (predicated)

num	5	*	*	*
exp	1	*	*	*
mask	1	1	1	1
result	$5*2^1$	*	*	*



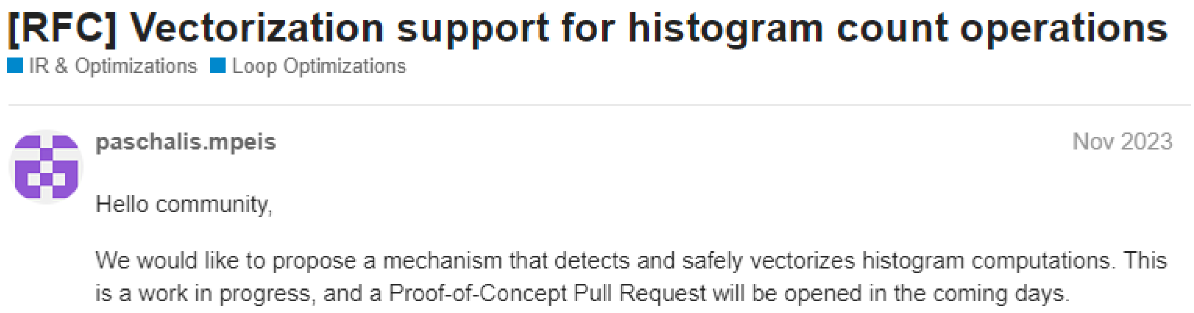
> Even there’s no loop or the loop cannot be vectorized, due to SIMD register file overlapping, it can also bring performance benefit



More ongoing work

HISTCNT <Zd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
Count matching elements in vector

- Also SVE2 HISTCNT is a native instruction for histogram computation
  - More generally, can be used for “conflict detection” like x86 AVX512-CD instructions
  - In community there’s already RFC raised by Arm to support it: Discourse#74788



[RFC] Vectorization support for histogram count operations

🟦 IR & Optimizations 🟦 Loop Optimizations

paschalis.mpeis Nov 2023

Hello community,

We would like to propose a mechanism that detects and safely vectorizes histogram computations. This is a work in progress, and a Proof-of-Concept Pull Request will be opened in the coming days.

Thank you.

Bring digital to every person, home and organization for a fully connected, intelligent world.

The information in this document may contain predictive statements including, without limitation, statements regarding the future financial and operating results, future product portfolio, new technology, etc. There are a number of factors that could cause actual results and developments to differ materially from those expressed or implied in the predictive statements. Therefore, such information is provided for reference purpose only and constitutes neither an offer nor an acceptance. Huawei may change the information at any time without notice.

HUAWEI