Enabling Loop Vectorizer for Compressing Store Pattern

Tejas Joshi
AMD Compilers Team

AMD
together we advance_

LLVM Loop Vectorizer

- Auto Loop Vectorization is one of the essential transformations targeted for performance.
- Control flow inside the loop makes the transformation challenging.
- Compilers can perform if-conversion or code flattening to vectorize such loops.
- LLVM vectorizer is sophisticated enough to vectorize complex control flow inside loops.
- However, there are certain cases which require additional handling to vectorize them.

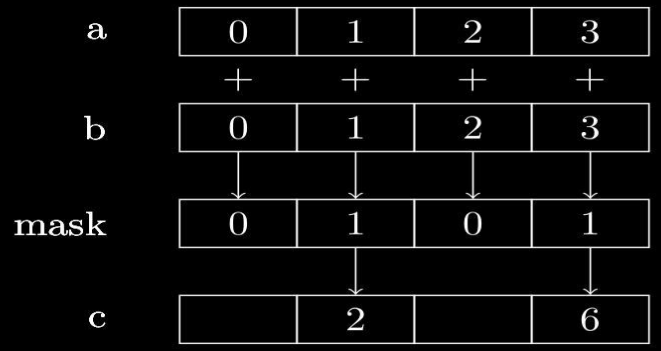
```
for (int i = 0; i < N; ++i) {
    if (Cond)
        c[i] = a[i] + b[i];
}

int j = 0;
for (int i = 0; i < N; ++i) {
    if (Cond)
        c[j++] = a[i] + b[i];
}
```

Flattened vectorized code

- Vectorized instructions based on conditional mask.
- Operations for each vector lane based on this mask.
- Load/Stores may or may not be contiguous.

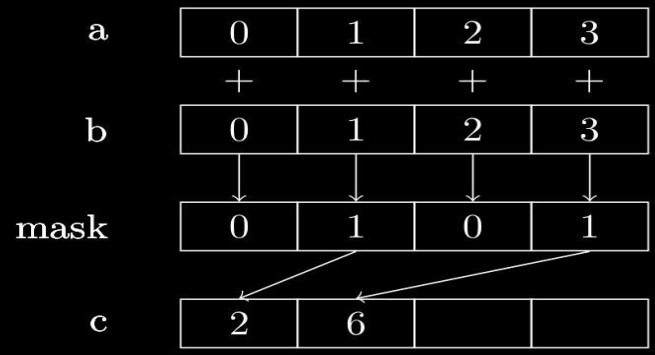
```
for (int i = 0; i < N; ++i) {
    if (Cond)
        c[i] = a[i] + b[i];
}
```



Compressing Store Pattern

- Vectorized instructions based on conditional mask.
- Operations for each vector lane based on this mask.
- Stores to a memory location are contiguous, source to these stores may or may not.
- Spread data gathered in a contiguous location, thus ‘compressing’ pattern.

```
int j = 0;
for (int i = 0; i < N; ++i) {
    if (Cond)
        c[j++] = a[i] + b[i];
}
```



Legality

- Predicated Induction Variable j is being modified at multiple places.
- The stores to destination array (c) may not be contiguous.

```
int j = 0;
for (int i = 0; i < N; ++i) {
    j = ...
    if (Cond)
        c[j++] = a[i] + b[i];
    j = ...
}
```

- Predicated Induction Variable j being used in scalar/vector operation or load/store index.
- Unable to predict its value, apart from the compressing store.

```
int j = 0;
for (int i = 0; i < N; ++i) {
    ... = j
    if (Cond)
        c[j++] = a[i] + b[i];
    d[j] = ...
}
```

Legality

- Multiple stores with Predicated Induction Variable, but with the same predicate mask.
- Other standard legalities such as dependences/memory safety also apply.
- Additionally, the target should support compressing store vector instructions and popcount instructions.

```
int j = 0;
for (int i = 0; i < N; ++i) {
    if (Cond1) {
        c[j] = a[i] + b[i];
        d[j] = a[i] - b[i];
        j++;
    }
}
```

Enabling LLVM Vectorizer for the Pattern

```
for.body:                                         ; preds = %entry, %for.inc
  %i.015 = phi i32 [ 0, %entry ], [ %inc8, %for.inc ]
  %j.014 = phi i32 [ 0, %entry ], [ %j.1, %for.inc ]
  %idxprom = zext i32 %i.015 to i64
  %arrayidx = getelementptr inbounds i32, ptr %a, i64 %idxprom
  %0 = load i32, ptr %arrayidx, align 4, !tbaa !5
  %cmp1 = icmp sgt i32 %0, 0
  br i1 %cmp1, label %if.then, label %for.inc

if.then:                                          ; preds = %for.body
  %arrayidx5 = getelementptr inbounds i32, ptr %b, i64 %idxprom
  %1 = load i32, ptr %arrayidx5, align 4, !tbaa !5
  %add = add nsw i32 %1, %0
  %inc = add nsw i32 %j.014, 1
  %idxprom6 = sext i32 %j.014 to i64
  %arrayidx7 = getelementptr inbounds i32, ptr %c, i64 %idxprom6
  store i32 %add, ptr %arrayidx7, align 4, !tbaa !5
  br label %for.inc

for.inc:                                          ; preds = %for.body, %if.then
  %j.1 = phi i32 [ %inc, %if.then ], [ %j.014, %for.body ]
  %inc8 = add nuw nsw i32 %i.015, 1
  %cmp = icmp ult i32 %i.015, 999
  br i1 %cmp, label %for.body, label %for.cond.cleanup, !llvm.loop !9
```

1. Recognize the PHI as Predicated Induction PHI (Loop Vectorization Legality):

```
LV: Not vectorizing: Found an unidentified PHI   %j.014 = phi i32 [ 0, %entry ], [ %j.1, %for.inc ]
```

2. Get SCEV under the assumption:

```
%i.015 = phi i32 [ 0, %entry ], [ %inc8, %for.inc ]
  -->  {0,+,1}<nuw><nsw><%for.body> U: [0,1000) S: [0,1000)
%j.014 = phi i32 [ 0, %entry ], [ %j.1, %for.inc ]
       -->  %j.014 U: full-set S: full-set
```

Enabling LLVM Vectorizer for the Pattern

```
for.body:                                         ; preds = %entry, %for.inc
  %i.015 = phi i32 [ 0, %entry ], [ %inc8, %for.inc ]
  %j.014 = phi i32 [ 0, %entry ], [ %j.1, %for.inc ]
  %idxprom = zext i32 %i.015 to i64
  %arrayidx = getelementptr inbounds i32, ptr %a, i64 %idxprom
  %0 = load i32, ptr %arrayidx, align 4, !tbaa !5
  %cmp1 = icmp sgt i32 %0, 0
  br i1 %cmp1, label %if.then, label %for.inc

if.then:                                          ; preds = %for.body
  %arrayidx5 = getelementptr inbounds i32, ptr %b, i64 %idxprom
  %1 = load i32, ptr %arrayidx5, align 4, !tbaa !5
  %add = add nsw i32 %1, %0
  %inc = add nsw i32 %j.014, 1
  %idxprom6 = sext i32 %j.014 to i64
  %arrayidx7 = getelementptr inbounds i32, ptr %c, i64 %idxprom6
  store i32 %add, ptr %arrayidx7, align 4, !tbaa !5
  br label %for.inc

for.inc:                                          ; preds = %for.body, %if.then
  %j.1 = phi i32 [ %inc, %if.then ], [ %j.014, %for.body ]
  %inc8 = add nuw nsw i32 %i.015, 1
  %cmp = icmp ult i32 %i.015, 999
  br i1 %cmp, label %for.body, label %for.cond.cleanup, !llvm.loop !9
```

3. Categorize the store with Predicated Induction as Compressing Store:
   - Let Recipes of VPlan know NOT to widen this store but generate @llvm.masked.compressstore.vXXX intrinsic

Enabling LLVM Vectorizer for the Pattern

```
for.body:                                         ; preds = %entry, %for.inc
  %i.015 = phi i32 [ 0, %entry ], [ %inc8, %for.inc ]
  %j.014 = phi i32 [ 0, %entry ], [ %j.1, %for.inc ]
  %idxprom = zext i32 %i.015 to i64
  %arrayidx = getelementptr inbounds i32, ptr %a, i64 %idxprom
  %0 = load i32, ptr %arrayidx, align 4, !tbaa !5
  %cmp1 = icmp sgt i32 %0, 0
  br i1 %cmp1, label %if.then, label %for.inc

if.then:                                          ; preds = %for.body
  %arrayidx5 = getelementptr inbounds i32, ptr %b, i64 %idxprom
  %1 = load i32, ptr %arrayidx5, align 4, !tbaa !5
  %add = add nsw i32 %1, %0
  %inc = add nsw i32 %j.014, 1
  %idxprom6 = sext i32 %j.014 to i64
  %arrayidx7 = getelementptr inbounds i32, ptr %c, i64 %idxprom6
  store i32 %add, ptr %arrayidx7, align 4, !tbaa !5
  br label %for.inc

for.inc:                                          ; preds = %for.body, %if.then
  %j.1 = phi i32 [ %inc, %if.then ], [ %j.014, %for.body ]
  %inc8 = add nuw nsw i32 %i.015, 1
  %cmp = icmp ult i32 %i.015, 999
  br i1 %cmp, label %for.body, label %for.cond.cleanup, !llvm.loop !9
```

4. Mark the latch PHI for the Predicated Induction Variable as scalar:
   - The BLEND VPlan recipe vectorizes such PHIs by default
   - Alternatively, get rid of this PHI (next…)

VPlan

```
for.body:                                         ; preds = %entry, %for.inc
  %i.015 = phi i32 [ 0, %entry ], [ %inc8, %for.inc ]
  %j.014 = phi i32 [ 0, %entry ], [ %j.1, %for.inc ]
  %idxprom = zext i32 %i.015 to i64
  %arrayidx = getelementptr inbounds i32, ptr %a, i64 %idxprom
  %0 = load i32, ptr %arrayidx, align 4, !tbaa !5
  %cmp1 = icmp sgt i32 %0, 0
  br i1 %cmp1, label %if.then, label %for.inc

if.then:                                          ; preds = %for.body
  %arrayidx5 = getelementptr inbounds i32, ptr %b, i64 %idxprom
  %1 = load i32, ptr %arrayidx5, align 4, !tbaa !5
  %add = add nsw i32 %1, %0
  %inc = add nsw i32 %j.014, 1
  %idxprom6 = sext i32 %j.014 to i64
  %arrayidx7 = getelementptr inbounds i32, ptr %c, i64 %idxprom6
  store i32 %add, ptr %arrayidx7, align 4, !tbaa !5
  br label %for.inc

for.inc:                                          ; preds = %for.body, %if.then
  %j.1 = phi i32 [ %inc, %if.then ], [ %j.014, %for.body ]
  %inc8 = add nuw nsw i32 %i.015, 1
  %cmp = icmp ult i32 %i.015, 999
  br i1 %cmp, label %for.body, label %for.cond.cleanup, !llvm.loop !9
```

```
<x1> vector loop: {
  vector.body:
    EMIT vp<%2> = CANONICAL-INDUCTION
    CLONE ir<%j.014>     = PRED-INDUCTION
    vp<%4>    = SCALAR-STEPS vp<%2>, ir<1>
    CLONE ir<%idxprom> = zext vp<%4>
    CLONE ir<%arrayidx> = getelementptr ir<%a>, ir<%idxprom>
    WIDEN ir<%0> = load ir<%arrayidx>
    WIDEN ir<%cmp1> = icmp sgt ir<%0>, ir<0>
    CLONE ir<%arrayidx5> = getelementptr ir<%b>, ir<%idxprom>
    WIDEN ir<%1> = load ir<%arrayidx5>, ir<%cmp1>
    WIDEN ir<%add> = add ir<%1>, ir<%0>
    EMIT vp<%12> = call POPCNT  ir<%cmp1>
    EMIT ir<%inc> = add ir<%j.014> vp<%12>
    CLONE ir<%idxprom6> = sext ir<%j.014>
    CLONE ir<%arrayidx7> = getelementptr ir<%c>, ir<%idxprom6>
    WIDEN store ir<%arrayidx7>, ir<%add>, ir<%cmp1>
    EMIT vp<%16> = cmp UGT POPCNT  vp<%12>
    EMIT vp<%17> = not vp<%16>
    BLEND %j.1 = ir<%inc>/vp<%16> ir<%j.014>/vp<%17>
    EMIT ir<%i.015> = VF * UF +(nuw)  vp<%2>
    EMIT branch-on-count  ir<%i.015> vp<%1>
  No successors
}
```

Costing and Recipes

```
<x1> vector loop: {
  vector.body:
    EMIT vp<%2> = CANONICAL-INDUCTION
    CLONE ir<%j.014>     = PRED-INDUCTION
    vp<%4>    = SCALAR-STEPS vp<%2>, ir<1>
    CLONE ir<%idxprom> = zext vp<%4>
    CLONE ir<%arrayidx> = getelementptr ir<%a>, ir<%idxprom>
    WIDEN ir<%0> = load ir<%arrayidx>
    WIDEN ir<%cmp1> = icmp sgt ir<%0>, ir<0>
    CLONE ir<%arrayidx5> = getelementptr ir<%b>, ir<%idxprom>
    WIDEN ir<%1> = load ir<%arrayidx5>, ir<%cmp1>
    WIDEN ir<%add> = add ir<%1>, ir<%0>
    EMIT vp<%12> = call POPCNT  ir<%cmp1>
    EMIT ir<%inc> = add ir<%j.014> vp<%12>
    CLONE ir<%idxprom6> = sext ir<%j.014>
    CLONE ir<%arrayidx7> = getelementptr ir<%c>, ir<%idxprom6>
    WIDEN store ir<%arrayidx7>, ir<%add>, ir<%cmp1>
    EMIT ir<%i.015> = VF * UF +(nuw)  vp<%2>
    EMIT branch-on-count  ir<%i.015> vp<%1>
  No successors
}
```

- Additional cost of the POPCNT intrinsic:
  - Cost of @llvm.ctpop.iX intrinsic call through getVectorIntrinsicCost
- Cost of widening store instruction replaced by:
  - Cost of @llvm.masked.compressstore.vXXX call through getVectorIntrinsicCost
- New Recipes:
  - New recipe for the Predicated Induction PHI : VPWidenIntPredInductionRecipe
  - EMIT POPCNT handled by VPInstruction, no new recipe

Future Work

- Similar approach can be taken to enable the vectorization for Expanding Loads Pattern.

Copyright and disclaimer

- The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate releases, for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

- THIS INFORMATION IS PROVIDED 'AS IS." AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION


AMD