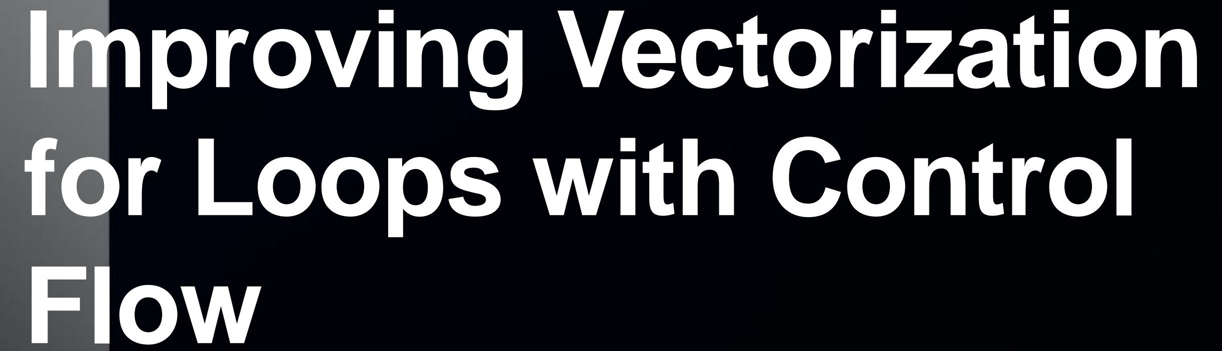


Improving Vectorization for Loops with Control Flow

Ashutosh Nema & Anupama Rasale
AMD Compilers Team

AMD
together we advance_

[Public]

Vectorization Of Conditional Statements

- Auto-vectorization is an essential compiler optimization.
- In the presence of control flow, it gets challenging.
- Generally, compilers deploy if-conversion and code flattening approach to vectorize control flow.
- These approaches sometimes suboptimal and leave an optimization opportunity on table.

```
Example:
for (unsigned i = 0; i < len; i++) {
  if (X[i])
    A[i] = B[i] + C[i];
}
```


AMD
together we advance_

[Public]

Current State Of Loop Vectorization

- LLVM already supports the vectorization of conditional statements.
- LLVM auto vectorization techniques deploy flattening of control flow where the guarded code is executed in all the paths with the help of predicated mask instructions.

AMD
together we advance_

[Public]

Flattened Style Vectorization

```
Example:
for (unsigned i = 0; i < len; i++) {
  if (X[i])
    A[i] = B[i] + C[i];
}
```

The conditional vector code flattened into vector loop body

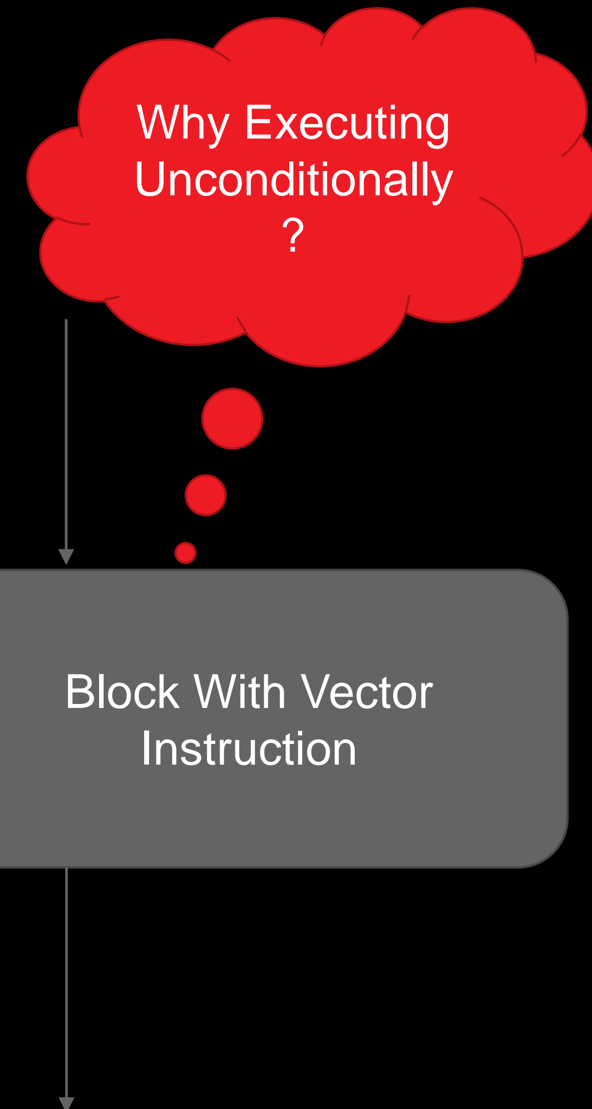
```
vector.body:                                      ; preds = %vector.body, %vector.ph
  %index = phi i64 [ 0, %vector.ph ], [ %index.next, %vector.body ]
  %0 = getelementptr inbounds i32, ptr %X, i64 %index
  %wide.load = load <8 x i32>, ptr %0, align 4, !tbaa !5
  %1 = icmp ne <8 x i32> %wide.load, zeroinitializer
  %2 = getelementptr i32, ptr %B, i64 %index
  %wide.masked.load = tail call <8 x i32> @llvm.masked.load.v8i32.p0(ptr %2, i32 4, <8 x i1> %1, <8 x i32> poison), !tbaa !5
  %3 = getelementptr i32, ptr %C, i64 %index
  %wide.masked.load15 = tail call <8 x i32> @llvm.masked.load.v8i32.p0(ptr %3, i32 4, <8 x i1> %1, <8 x i32> poison), !tbaa !5
  %4 = add nsw <8 x i32> %wide.masked.load15, %wide.masked.load
  %5 = getelementptr i32, ptr %A, i64 %index
  tail call void @llvm.masked.store.v8i32.p0(<8 x i32> %4, ptr %5, i32 4, <8 x i1> %1), !tbaa !5
  %index.next = add nuw i64 %index, 8
  %6 = icmp eq i64 %index.next, %n.vec
  br i1 %6, label %middle.block, label %vector.body, !llvm.loop !9
```


AMD
together we advance_

[Public]

Challenges With Flattened Code

- The flattened code at runtime execute the instructions in all control paths unconditionally.
- It slows down the performance when mask is all set to false and the guarded instructions are not supposed to be executed.
- The memory access safety is ensured using the predicated mask instructions, i.e., mask-load and mask-store.
  - If the mask is set to ‘false’ then the memory location won't be accessed.

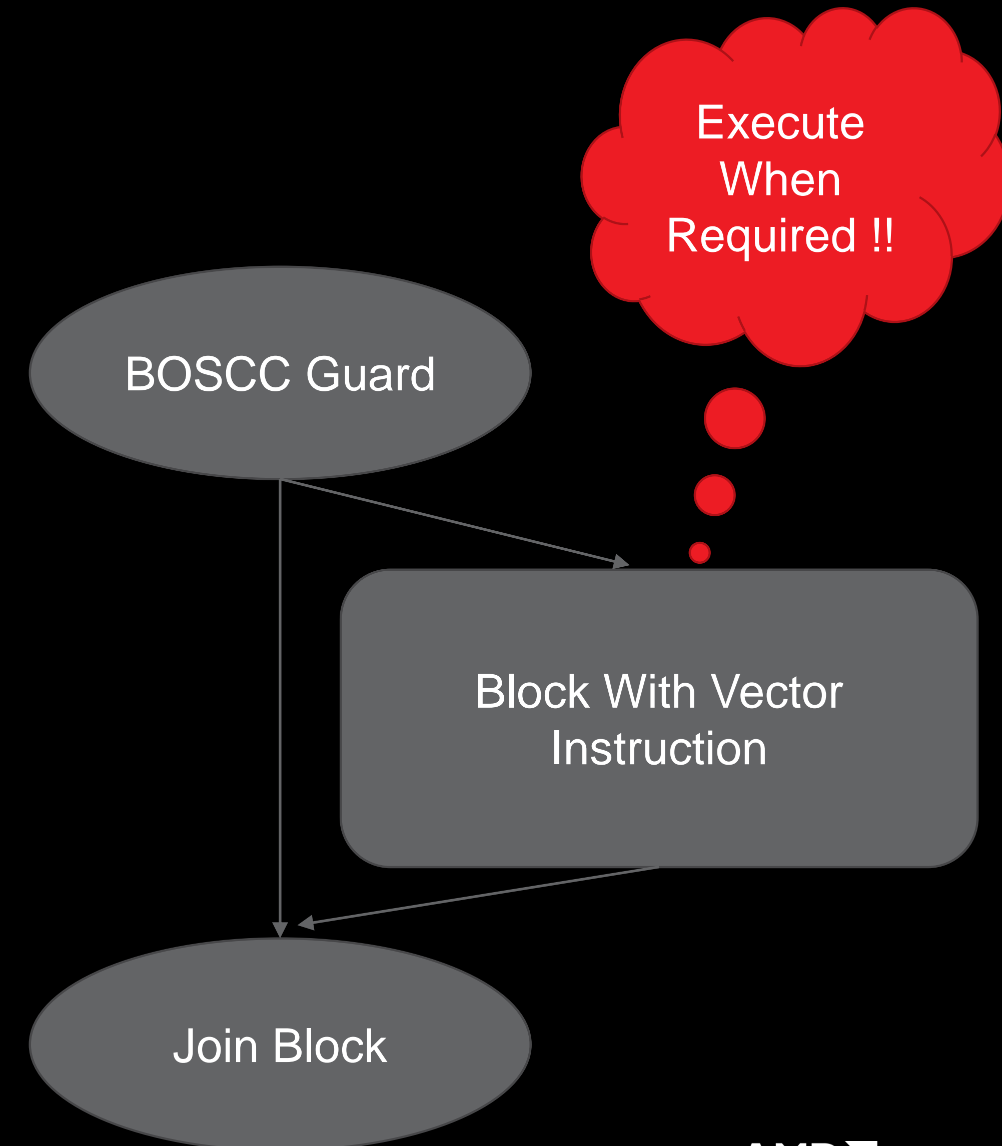


AMD
together we advance_

[Public]

BOSCC Style Vectorization

- Here we introduce the implementation of Branch-On-Super-Word-Conditional-Codes (BOSCC) way of vectorization in the presence of conditional statements.
  - *Paper "Introducing Control Flow into Vectorized Code" by Jaewook Shin.*
- BOSCC introduces a branch instruction that can be conditionally taken based on the comparison result of two vector variables.
- BOSCC encloses the vector instructions guarded by vector predicate inside an if-statement.
- It by-passes vector instruction the guarding vector predicate has all false values.



AMD
together we advance_

[Public]

BOSCC Style Vectorization

```
vector.body:                                    ; preds = %vector.ph, %for.inc17
  %index = phi i64 [ 0, %vector.ph ], [ %index.next, %for.inc17 ]
  %0 = getelementptr inbounds i32, ptr %X, i64 %index
  %wide.load = load <8 x i32>, ptr %0, align 4, !tbaa !5
  %1 = icmp ne <8 x i32> %wide.load, zeroinitializer
  br label %if.then.boscc.guard
```

```
Example:
for (unsigned i = 0; i < len; i++) {
  if (X[i])
    A[i] = B[i] + C[i];
}
```

Guard Block

```
if.then.boscc.guard:                            ; preds = vector.body
  %2 = bitcast <8 x i1> %1 to i8
  %.not = icmp eq i8 %2, 0
  br i1 %.not, label %for.inc17, label %if.then.boscc
```

Vector Block

```
if.then.boscc:                                  ; preds = %if.then.boscc.guard
  %3 = getelementptr i32, ptr %B, i64 %index
  %wide.masked.load = tail call <8 x i32> @llvm.masked.load.v8i32.p0(ptr %3, i32 4, <8 x i1> %1, <8 x i32> poison), !tbaa !5
  %4 = getelementptr i32, ptr %C, i64 %index
  %wide.masked.load16 = tail call <8 x i32> @llvm.masked.load.v8i32.p0(ptr %4, i32 4, <8 x i1> %1, <8 x i32> poison), !tbaa !5
  %5 = add nsw <8 x i32> %wide.masked.load16, %wide.masked.load
  %6 = getelementptr i32, ptr %A, i64 %index
  tail call void @llvm.masked.store.v8i32.p0(<8 x i32> %5, ptr %6, i32 4, <8 x i1> %1), !tbaa !5
  br label %for.inc17
```

```
for.inc17:                                      ; preds = %if.then.boscc.guard, %if.then.boscc
  %index.next = add nuw i64 %index, 8
  %7 = icmp eq i64 %index.next, %n.vec
  br i1 %7, label %middle.block, label %vector.body, !llvm.loop !9
```


AMD
together we advance_

[Public]

BOSCC Benefits

- BOSCC aims to avoid the execution for the vector instructions where the guard compare condition results in all false mask.

- When mask is set to all false(i.e. <0,0,0,0>) values the corresponding vector instructions never get executed, but in flatten it always get executed.

- BOSCC brings the performance uplift by avoiding the unconditional execution for the vector instructions.

AMD
together we advance_

[Public]

MemSafety to avoid masked instructions

- For some architecture the masked version of memory instructions are expensive compared to regular vector memory instructions
  - i.e. mask-load mask-store
- During vectorization due to high cost for these instructions the vectorization for some cases can be avoided. And even when it's done its sub optimal.

```
for (unsigned i = 0 ; i < len; i++) {
  if (X[i])
    A[i] = B[i] + C[i];
  else
    B[i] = A[i] * C[i];
}
```


AMD
together we advance_

[Public]

MemSafety to avoid masked instructions

- Cases where memory access are guaranteed to be accessed in all the paths of the loop, we aim to avoid the masked memory instructions by generating an alternate sequence of instruction.
- Please note the memory accesses ‘A’, ‘B’ & ‘C’ are accessed in all the path of the loop.

```
for (unsigned i = 0 ; i < len; i++) {
  if (X[i])
    A[i] = B[i] + C[i];
  else
    B[i] = A[i] * C[i];
}
```

```
if.then.vec.bb:                                    ; preds = %vector.body
  %3 = getelementptr inbounds i32, ptr %C, i64 %index
  %wide.load33 = load <8 x i32>, ptr %3, align 4, !tbaa !5, !alias.scope !12
  %19 = getelementptr i32, ptr %B, i64 %index
  %wide.load34 = load <8 x i32>, ptr %19, align 4, !tbaa !5, !alias.scope !23, !noalias !25
  %20 = add nsw <8 x i32> %wide.load33, %wide.load34
  %21 = getelementptr i32, ptr %A, i64 %index
  %filler.load = load <8 x i32>, ptr %21, align 4
  %22 = select <8 x i1> %2, <8 x i32> %filler.load, <8 x i32> %20
  store <8 x i32> %22, ptr %21, align 4, !tbaa !5, !alias.scope !26
  br label %if.else.vec.cond.bb
```

Mask Store is replaced with alternate sequence

together we advance_

[Public]

BOSCC Design

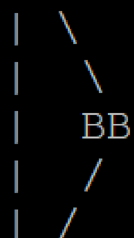
BOSCCBlockPlanner	• Facilitates to generate the required block layout for BOSCC blocks during Vplan • BOSCC Legal & Profitability
VPBranchOnBOSCCGuardSC	• This recipe is responsible for generating the required conditional entry check on a vector block
VPBOSCCLiveOutRecipe	• This recipe is responsible to generate PHI for the live out from the guarded vector blocks.

AMD
together we advance_

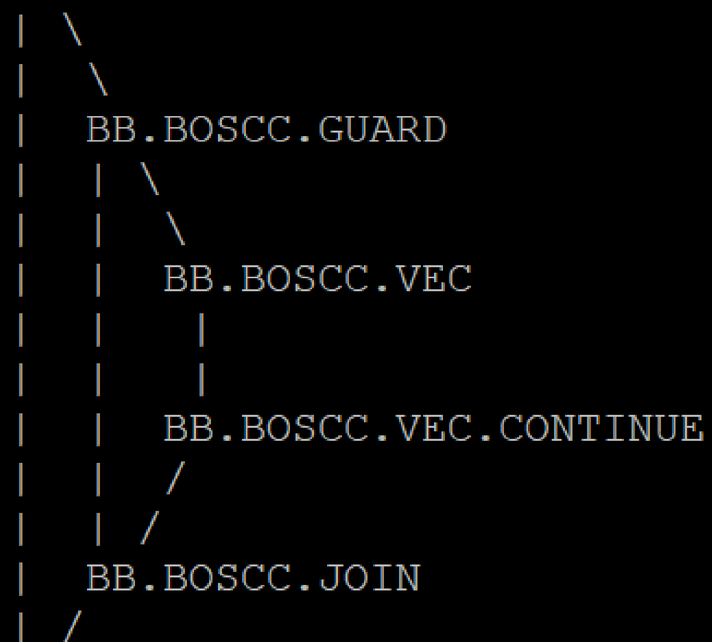
[Public]

BOSCC Block Layout

Consider below block layout



It gets transformed to:




```
BLOCK DESCRIPTION:
------------------
 BB.BOSCC.GUARD : This serves the purpose of guarding with right condition

 BB.BOSCC.VEC : This is the vector block corresponds to BB

 BB.BOSCC.VEC.CONTINUE : Auxiliary block to facilitate control flow

 BB.BOSCC.JOIN : Required for PHI generation for the live out from BB
```


AMD
together we advance_

[Public]

BOSCC Results

For our experiments tried TSVC vectorization suite, observed great benefit for some loops:

Runtime in Secs

TSVC Loop	BOSCC	Baseline(Flatten)	Uplift Percentage
s123	11.49	12.25	6.62
s124	5.44	6.23	14.5
s272	1.33	26.02	1863.47
s273	17.59	19.27	9.55
s278	15.83	28.13	77.71
s279	9.26	22.34	141.28
s1279	1.95	15.2	679.28
s2710	17.46	28.96	65.83
s3111	0.48	0.52	9.24
s3113	3.82	4.14	8.16
s441	16.71	49.81	198.06
s443	8.14	9.36	15.06
s253	27.39	26.545	-3.09

AMD
together we advance_

[Public]

Summary

- Executing vector blocks unconditionally is sub optimal.
- BOSCC inserts a guard check to avoid execution for cases where the condition guarding a block remains false.
- Patch is available for review - https://reviews.llvm.org/D139074

AMD
together we advance_

Copyright and disclaimer

- ©2023 Advanced Micro Devices, Inc. All rights reserved.

- AMD, the AMD Arrow logo, [insert all other AMD trademarks used in the material IN ALPHABETICAL ORDER here per AMD's Guidelines on Using Trademark Notice and Attribution] and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

- 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