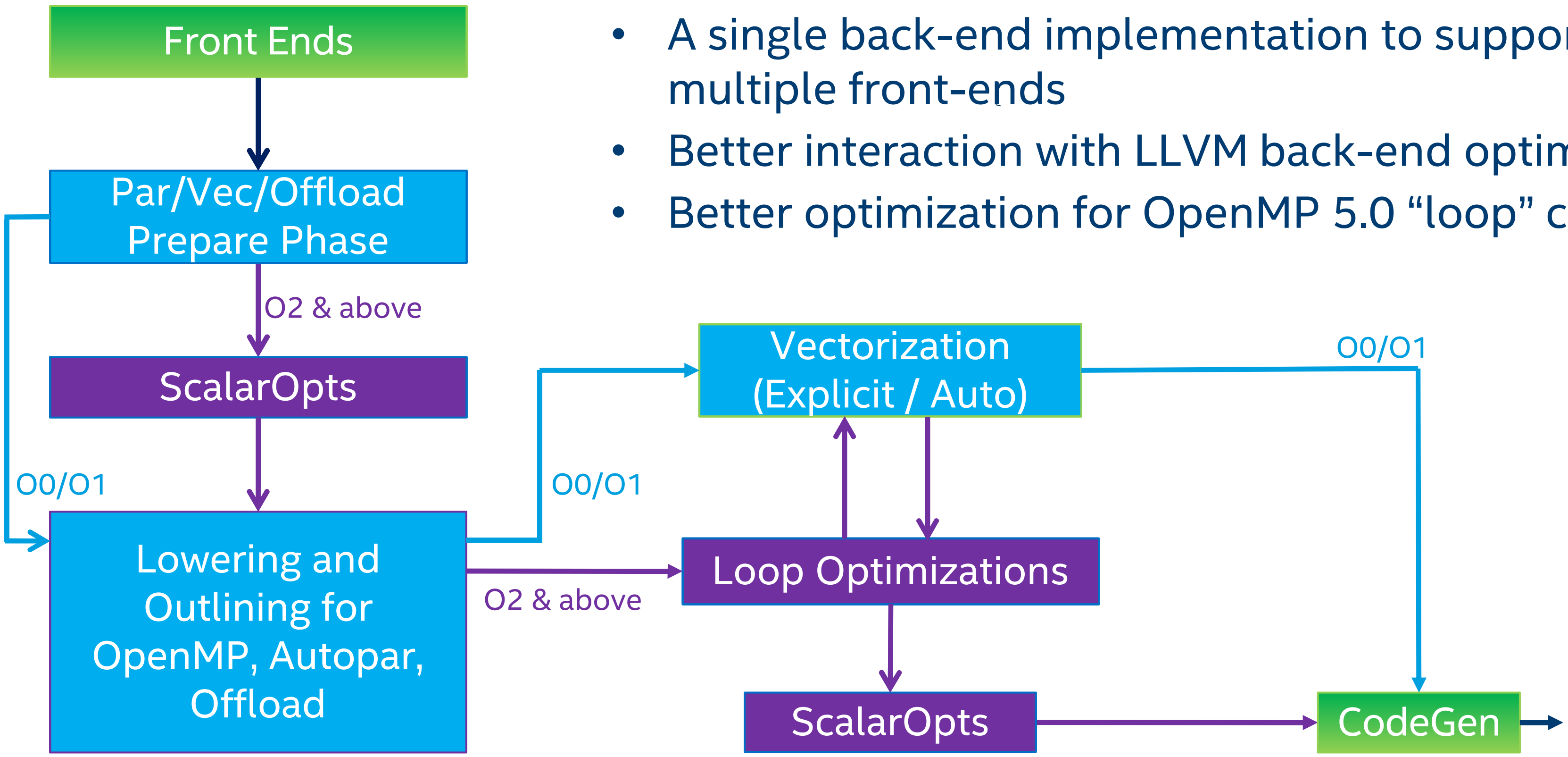
Methods for Maintaining OpenMP* Semantics without Being Overly Conservative

Jin Lin, Ernesto Su, Xinmin Tian
Intel Corporation

LLVM Developers' Meeting 2018, October 17-18, San Jose

OpenMP Backend Outlining in LLVM Compiler



- A single back-end implementation to support multiple front-ends
- Better interaction with LLVM back-end optimizations
- Better optimization for OpenMP 5.0 “loop” construct

Issues to be Addressed for OpenMP Transformations in the LLVM Backend

- How to represent OpenMP loops?
- How to handle code motion of instructions across OpenMP region that violates OpenMP semantics?
- How to update SSA form during OpenMP transformations?
- How to preserve alias information of memory references in outlined functions?

Agenda

- **Overview of representing OpenMP directives**
- Representing OpenMP loops
- Handling code motion that violates OpenMP semantics
- Updating SSA form during transformations
- Preserving alias information in outlined function
- Summary

Representing OpenMP Directives

```
void foo() {
  #pragma omp parallel
  {
    int x = foo();
    printf("%d\n", x);
  }
}
```

IR Dump After Clang FE

```
define dso_local void @_Z3foov() #0 {
entry:
  %x = alloca i32, align 4
  %0 = call token @llvm.directive.region.entry() [ "DIR.OMP.PARALLEL"(),
                                                  "QUAL.OMP.PRIVATE"(i32* %x) ]
  ...
  call void @llvm.directive.region.exit(token %0) [ "DIR.OMP.END.PARALLEL"() ]
  ret void
}
```

Agenda

- Overview of representing OpenMP directives
- **Representing OpenMP loops**
- Handling code motion that violates OpenMP semantics
- Updating SSA form during transformations
- Preserving alias information in outlined function
- Summary

Issues with Representing OpenMP Loops in LLVM IR

- OpenMP loops compiled at different optimization levels come in different forms.
- An OpenMP loop can be
  - rotated or not
  - normalized or not
- After optimizations, an OpenMP loop structure may
  - become hard to recognize
  - be optimized away

Our Approach of Representing OpenMP Loops

- Clang FE performs normalization for OpenMP loops.
- Add two operand bundle Tag Names to represent the OpenMP loop structure throughout optimizations.
  - QUAL.OMP.NORMALIZED.IV
  - QUAL.OMP.NORMALIZED.UB
- Generate a canonical form of the OpenMP loop.
  - Perform register promotion for loop index and upper bound.
  - Apply loop rotation to create bottom-test loop.
  - Apply loop regularization to generate the canonical form.

OpenMP Loop Representation

C/C++ Source

```
#pragma omp parallel for
  for (int i = M; i < N; i+=1)
     y[i] = i;
```

IR Dump After Clang FE

```
DIR.OMP.PARALLEL.LOOP.1:
 %15 = call token @llvm.directive.region.entry() [ "DIR.OMP.PARALLEL.LOOP"(),
        "QUAL.OMP.NORMALIZED.IV"(i32* %.omp.iv),
        "QUAL.OMP.NORMALIZED.UB"(i32* %.omp.ub), ...]
 br label %DIR.OMP.PARALLEL.LOOP.2
```

```
omp.inner.for.cond:
 %17 = load i32, i32* %.omp.iv
 %18 = load i32, i32* %.omp.ub,
 %cmp5 = icmp sle i32 %17, %18
 br i1 %cmp5, label %omp.inner.for.body, label %omp.for.end
```

```
omp.inner.for.inc:
 %26 = load i32, i32* %.omp.iv
 %add7 = add nsw i32 %26, 1
 store i32 %add7, i32* %.omp.iv
 br label %omp.inner.for.cond
```

OpenMP Loop Representation (Cont.)

IR Dump After Clang FE

```
DIR.OMP.PARALLEL.LOOP.1:
 %15 = call token @llvm.directive.region.entry() [
"DIR.OMP.PARALLEL.LOOP"(),
        "QUAL.OMP.NORMALIZED.IV"(i32* %.omp.iv),
        "QUAL.OMP.NORMALIZED.UB"(i32* %.omp.ub), ...]
 br label %DIR.OMP.PARALLEL.LOOP.2
```

```
omp.inner.for.cond:
 %17 = load i32, i32* %.omp.iv
 %18 = load i32, i32* %.omp.ub,
 %cmp5 = icmp sle i32 %17, %18
 br i1 %cmp5, label %omp.inner.for.body, label %omp.for.end
```

```
omp.inner.for.inc:
 %26 = load i32, i32* %.omp.iv
 %add7 = add nsw i32 %26, 1
 store i32 %add7, i32* %.omp.iv
 br label %omp.inner.for.cond
```

IR Dump Before OpenMP Transformations

```
DIR.OMP.PARALLEL.LOOP.1:
 %15 = call token @llvm.directive.region.entry() [
"DIR.OMP.PARALLEL.LOOP"(),
        "QUAL.OMP.NORMALIZED.IV"(i32* nullptr),
        "QUAL.OMP.NORMALIZED.UB"(i32* nullptr), ...]
 br label %DIR.OMP.PARALLEL.LOOP.2
```

```
DIR.OMP.PARALLEL.LOOP.113:
 %4 = load i32, i32* %.omp.lb
 %cmp514 = icmp sgt i32 %4, %sub4
 br i1 %cmp514, label %omp.loop.exit, label %omp.lr.ph
```

```
omp,body:
 %.omp.iv.0 = phi i32 [ %4, %omp.inner.for.body.lr.ph ],
                      [ %add7, %omp.for.body ]
 ....
 %add7 = add nsw i32 %.omp.iv.0, 1
 %cmp5 = icmp sle i32 %add7, %sub4
 br i1 %cmp5, label %omp.body, label %omp.exit_crit_edge
```

Transformations on Canonical Loops

- Canonical form of an OpenMP loop

```
do {  // pseudo-code dump
    %omp.iv = phi(%omp.lb, %omp.inc)
    ...
    %omp.inc = %omp.iv + 1
} while (%omp.inc <= %omp.ub)
```

- Advantages of the canonical form
  - Simplifies loop analyses
  - Simplifies loop transformations
    - Update the loop upper bound directly without introducing extra induction variables

Agenda

- Overview of representing OpenMP directives
- Representing OpenMP loops
- **Handling code motion that violates OpenMP semantics**
- Updating SSA form during transformations
- Preserving alias information in outlined function
- Summary

Example of Code Motion that Violates OpenMP Semantics

C/C++ Source

```
void foo() {
 int pvtPtr[10];
 pvtPtr[4] = 4;
 #pragma omp parallel firstprivate(pvtPtr)
 {
     printf("%d\n", pvtPtr[4]);
  }
}
```

IR after Clang FE

```
%arrayidx = getelementptr inbounds [10 x i32], [10 x
i32]* %pvtPtr, i64 0, i64 4
 store i32 4, i32* %arrayidx
 br label %DIR.OMP.PARALLEL.1
```

```
DIR.OMP.PARALLEL.1:
 %1 = call token @llvm.directive.region.entry() [
"DIR.OMP.PARALLEL"(), "QUAL.OMP.FIRSTPRIVATE"([10 x
i32]* %pvtPtr) ]
 br label %DIR.OMP.PARALLEL.2
```

```
DIR.OMP.PARALLEL.2:
 %arrayidx1 = getelementptr inbounds [10 x i32], [10 x
i32]* %pvtPtr, i64 0, i64 4
 %2 = load i32, i32* %arrayidx1
 ...
 br label %DIR.OMP.END.PARALLEL.3
```

Example of Code Motion that Violates OpenMP Semantics (cont.)

IR after Clang FE

```
%arrayidx = getelementptr inbounds [10 x i32],
                         [10 x i32]* %pvtPtr, i64 0, i64 4
 store i32 4, i32* %arrayidx
 br label %DIR.OMP.PARALLEL.1
```

```
DIR.OMP.PARALLEL.1:
 %1 = call token @llvm.directive.region.entry() [
"DIR.OMP.PARALLEL"(), "QUAL.OMP.FIRSTPRIVATE"([10 x
i32]* %pvtPtr) ]
 br label %DIR.OMP.PARALLEL.2
```

```
DIR.OMP.PARALLEL.2:
 %arrayidx1 = getelementptr inbounds [10 x i32], [10 x
i32]* %pvtPtr, i64 0, i64 4
 %2 = load i32, i32* %arrayidx1
 ...
 br label %DIR.OMP.END.PARALLEL.3
```

IR after Early CSE

```
%arrayidx = getelementptr inbounds [10 x i32],
                         [10 x i32]* %pvtPtr, i64 0, i64 4
 store i32 4, i32* %arrayidx
 br label %DIR.OMP.PARALLEL.1
```

```
DIR.OMP.PARALLEL.1:
 %1 = call token @llvm.directive.region.entry() [
"DIR.OMP.PARALLEL"(), "QUAL.OMP.FIRSTPRIVATE"([10 x
i32]* %pvtPtr) ]
 br label %DIR.OMP.PARALLEL.2
```

DIR.OMP.PARALLEL.2:
~~%arrayidx1 = getelementptr inbounds [10 x i32], [10 x i32]* %pvtPtr, i64 0, i64 4~~
%2 = load i32, i32* %arrayidx
...
br label %DIR.OMP.END.PARALLEL.3

Solution to Handle Code Motion

- Generate the llvm.launder.invariant.group intrinsic to perform SSA renaming in OpenMP Prepare phase.
  - The renamed SSA value refers to a structure or array in the OpenMP region.
- Clean up the llvm.launder.invariant.group intrinsic before the OpenMP Transformation Pass.

The ‘llvm.launder.invariant.group’ intrinsic can be used when an invariant established by invariant.group metadata no longer holds, to obtain a new pointer value that carries fresh invariant group information. It is an experimental intrinsic, which means that its semantics might change in the future.

Example of Using @llvm.launder.invariant.group

IR After Prepare Phase

```
%arrayidx = getelementptr inbounds [10 x i32],
                              [10 x i32]* %pvtPtr, i64 0, i64 4
 store i32 4, i32* %arrayidx
 br label %DIR.OMP.PARALLEL.1
```

```
DIR.OMP.PARALLEL.1:
 %1 = call token @llvm.directive.region.entry() [
"DIR.OMP.PARALLEL"(), "QUAL.OMP.FIRSTPRIVATE"([10 x
i32]* %pvtPtr) ]
 %2 = bitcast [10 x i32]* %pvtPtr to i8*
 %3 = call i8* @llvm.launder.invariant.group.p0i8(i8* %2)
 %4 = bitcast i8* %3 to [10 x i32]*
 br label %DIR.OMP.PARALLEL.2
```

```
DIR.OMP.PARALLEL.2:
 %arrayidx1 = getelementptr inbounds [10 x i32], [10 x
i32]* %4, i64 0, i64 4
 %5 = load i32, i32* %arrayidx1
 ...
 br label %DIR.OMP.END.PARALLEL.3
```

IR Before OpenMP Transformations

```
%arrayidx = getelementptr inbounds [10 x i32],
                              [10 x i32]* %pvtPtr, i64 0, i64 4
 store i32 4, i32* %arrayidx
 br label %DIR.OMP.PARALLEL.1
```

```
DIR.OMP.PARALLEL.1:
 %1 = call token @llvm.directive.region.entry() [
"DIR.OMP.PARALLEL"(), "QUAL.OMP.FIRSTPRIVATE"([10 x
i32]* %pvtPtr) ]
 %2 = bitcast [10 x i32]* %pvtPtr to i8*
 ~~%3 = call i8* @llvm.launder.invariant.group.p0i8(i8* %2)~~
 %3 = bitcast i8* %2 to [10 x i32]*
 br label %DIR.OMP.PARALLEL.2
```

```
DIR.OMP.PARALLEL.2:
 %arrayidx1 = getelementptr inbounds [10 x i32], [10 x
i32]* %3, i64 0, i64 4
 %4 = load i32, i32* %arrayidx1
 ...
 br label %DIR.OMP.END.PARALLEL.3
```

Agenda

- Overview of representing OpenMP directives
- Representing OpenMP loops
- Handling code motion that violates OpenMP semantics
- **Updating SSA form during transformations**
- Preserving alias information in outlined function
- Summary

Issue of SSA Form Update during OpenMP Transformations

- OpenMP transformations need to update the SSA form in the following two cases.
  - Generate a new top test expression in the front of the OMP loop.
  - New outer dispatching loop is introduced for some schedule types.
- The existing LCSSA update utility is insufficient to support the SSA form update for those two cases.

Example of SSA Form Update

```
int foo(int n, int *v) {
 int t = 0;
 #pragma omp for
schedule(static, 10)
 for(int i = 0; i < n; i++)
{
    int vx = v[i * 3 + 0];
    int vy = v[i * 3 + 1];
    int vz = v[i * 3 + 2];
    t += vx * vx +
         vy * vy +
         vz * vz;
}
 return t;
}
```

IR Before OpenMP Transformation

```
omp.inner.for.body:
 %.omp.iv.0 = phi i32 [ %2,  %omp.inner.for.body.lr.ph ],
[ %add22, %omp.inner.for.body ]
 %t.035 = phi i32 [ 0, %omp.inner.for.body.lr.ph ],
                   [%add21, %omp.inner.for.body ]
 ....
 br i1 %cmp4, label %omp.inner.for.body, label
%omp.inner..exit_crit_edge
```

```
omp.inner.exit_crit_edge:
 %split = phi i32 [ %add21, %omp.inner.for.body ]
 br label %omp.loop.exit
```

```
omp.loop.exit:
 %t.034 = phi i32 [ 0, %DIR.OMP.LOOP.132 ], [ %split,
%omp.inner.exit_crit_edge ]
...
br label %omp.precond.end
```

IR During OpenMP Transformation

```
dispatch.header:
 ...
 br i1 %ub.min, label %dispatch.body, label %dispatch.min.ub
```

```
dispatch.min.ub:
 store i32 %sub2, i32* %upper.bnd
 br label %dispatch.body
```

```
dispatch.body:
 ...
 br i1 %3, label %omp.inner.for.body, label %dispatch.latch
```

```
omp.inner.for.body:
 %.omp.iv.0 = phi i32 [ %add22, %omp.inner.for.body ], [ %lb.new, %dispatch.body ]
 %t.035 = phi i32 [ 0, %dispatch.body ], [ %add21, %omp.inner.for.body ]
 ....
 br i1 %cmp4, label %omp.inner.for.body, label %omp.inner.exit_crit_edge
```

```
omp.inner.exit_crit_edge:
 %split = phi i32 [ %add21, %omp.inner.for.body ]
 br label %dispatch.inc
```

```
dispatch.inc:
 ...
 br label %dispatch.header
```

```
dispatch.latch:
 call void @__kmpc_for_static_fini({...})
 br label %omp.loop.exit
```

```
omp.loop.exit:
 %t.034 = phi i32 [ 0, %DIR.OMP.LOOP.132 ], [ %split, %dispatch.latch ]
...
br label %omp.precond.end
```

General SSA Update Utility

- Compute live-in and live-out information for the OpenMP loop, including the generated dispatch loop.
- Analyze the live-range of the live-in and live-out values to build the equivalence class among those values.
  - An equivalence class contains values corresponding to the same induction or reduction variable.
- Replace the use of live-in values with live-out values if there exists loop-carried dependence.
- Leverage the SSA updater to perform SSA form update.

Example of SSA Form Update (Cont.)

%omp.inner.for.body.lr.ph

```
dispatch.header:
%split37 = phi i32 [ %split, %dispatch.inc ], [ 0, %omp.inner.for.body.lr.ph ]
%t.03536 = phi i32 [ %t.035, %dispatch.inc ], [ 0, %omp.inner.for.body.lr.ph ]
...
br i1 %ub.min, label %dispatch.body, label %dispatch.min.ub
```

```
dispatch.min.ub:
store i32 %sub2, i32* %upper.bnd
br label %dispatch.body
```

```
dispatch.body:
...
br i1 %3, label %omp.inner.for.body, label %dispatch.latch
```

```
omp.inner.for.body:
%.omp.iv.0 = phi i32 [ %add22, %omp.inner.for.body ], [ %lb.new, %dispatch.body ]
%t.035 = phi i32 [ 0, %dispatch.body ], [ %add21, %omp.inner.for.body ]
%t.035 = phi i32 [%t.035, %dispatch.body ], [ %add21, %omp.inner.for.body ]
%t.035 = phi i32 [ %t.03536, %dispatch.body ], [ %add21, %omp.inner.for.body ]
%add21 = ...
br i1 %cmp4, label %omp.inner.for.body, label %omp.inner.exit_crit_edge
```

```
omp.inner.exit_crit_edge:
%split = phi i32 [ %add21, %omp.inner.for.body ]
br label %dispatch.inc
```

```
dispatch.inc:
...
br label %dispatch.header
```

```
dispatch.latch:
call void @__kmpc_for_static_fini({...})
br label %omp.loop.exit
```

```
omp.loop.exit:
%t.034 = phi i32 [ 0, %DIR.OMP.LOOP.132 ], [ %split, %dispatch.latch ]
%t.034 = phi i32 [ 0, %DIR.OMP.LOOP.132 ], [ %split37, %dispatch.latch ]
...
br label %omp.precond.end
```

Live-out:

%t.035	available_value {0, %omp.inner.for.body.lr.ph}
%split	available_value {0 , %omp.inner.for.body.lr.ph}

Equivalence Classes = { %t.035, %split , %add21 }

Agenda

- Overview of representing OpenMP directives
- Representing OpenMP loops
- Handling code motion that violates OpenMP semantics
- Updating SSA form during transformations
- **Preserving alias information in outlined function**
- Summary

Preserving the Alias Information

```
void foo(double *glob)
{
 double tmp[5] = { 1.0, 2.0, 3.0, 4.0, 5.0 };

#pragma omp parallel for shared(tmp, glob)
 {
  for (int i = 0; i < 1000; ++i) {
    glob[i] = tmp[0] * tmp[1] +  tmp[2];
  }
 }
}
```

```
MayAlias:
 %0 = load double, double* %arrayidx1 <->   store double
%add, double* %arrayidx4
 %1 = load double, double* %arrayidx2 <->   store double
%add, double* %arrayidx4
 %2 = load double, double* %arrayidx3 <->   store double
%add, double* %arrayidx4
```

```
void
@foo_DIR.OMP([5 x double]* %tmp, double* %glob)
{ ....  }
```

```
for.cond:
  %storemerge = phi i32 [ 0, %DIR.OMP ], [ %inc, %for.body ]
  %cmp = icmp slt i32 %storemerge, 1000
  br i1 %cmp, label %for.body, label %for.cond.cleanup
```

```
for.body:
 %arrayidx1 = getelementptr inbounds [5 x double], [5 x double]*
%tmp, i64 0, i64 0
 %0 = load double, double* %arrayidx1
 %arrayidx2 = getelementptr inbounds [5 x double], [5 x double]*
%tmp, i64 0, i64 1
 %1 = load double, double* %arrayidx2
 %mul = fmul double %0, %1
 %arrayidx3 = getelementptr inbounds [5 x double], [5 x double]*
%tmp, i64 0, i64 2
 %2 = load double, double* %arrayidx3
 %add = fadd double %mul, %2
 %idxprom = sext i32 %storemerge to i64
 %arrayidx4 = getelementptr inbounds double, double* %glob, i64
%idxprom
 store double %add, double* %arrayidx4
 %inc = add nsw i32 %storemerge, 1
 br label %for.cond
```

Approach to Preserve the Alias Information

- Construct the alias matrix for all the memory references before the OpenMP region is outlined.
  - The initialization of alias matrix is based on the alias analysis results.
- Derive the alias-scope and no-alias metadata based on the alias matrix.

Using Scoped AA Metadata to Preserve Alias Information

```
void foo(double *glob)
{
 double tmp[5] = { 1.0, 2.0, 3.0, 4.0, 5.0 };

#pragma omp parallel for shared(tmp, glob)
 {
  for (int i = 0; i < 1000; ++i) {
    glob[i] = tmp[0] * tmp[1] +  tmp[2];
  }
 }
}
```

```
NoAlias:
 %0 = load double, double* %arrayidx1 <->   store double %add, double* %arrayidx4
 %1 = load double, double* %arrayidx2 <->   store double %add, double* %arrayidx4
 %2 = load double, double* %arrayidx3 <->   store double %add, double* %arrayidx4
```

```
void
@foo_DIR.OMP([5 x double]* %tmp, double* %glob)
{ .... }
```

```
for.cond:
  %storemerge = phi i32 [ 0, %DIR.OMP], [ %inc, %for.body ]
  %cmp = icmp slt i32 %storemerge, 1000
  br i1 %cmp, label %for.body, label %for.cond.cleanup
```

```
for.body:
 %arrayidx1 = getelementptr inbounds [5 x double], [5 x double]* %tmp, i64 0, i64 0
 %0 = load double, double* %arrayidx1, !alias.scope !1, !noalias !2
 %arrayidx2 = getelementptr inbounds [5 x double], [5 x double]* %tmp, i64 0, i64 1
 %1 = load double, double* %arrayidx2, !alias.scope !1, !noalias !2
 %mul = fmul double %0, %1
 %arrayidx3 = getelementptr inbounds [5 x double], [5 x double]* %tmp, i64 0, i64 2
 %2 = load double, double* %arrayidx3, !alias.scope !1, !noalias !2
 %add = fadd double %mul, %2
 %idxprom = sext i32 %storemerge to i64
 %arrayidx4 = getelementptr inbounds double, double* %glob, i64 %idxprom
 store double %add, double* %arrayidx4, !alias.scope !2, !noalias !1
 %inc = add nsw i32 %storemerge, 1
 br label %for.cond
```

Agenda

- Overview of representing OpenMP directives
- Representing OpenMP loops
- Handling code motion that violates OpenMP semantics
- Updating SSA form during transformations
- Preserving alias information in outlined function
- **Summary**

Summary

- Proposed a canonical representation for OpenMP loops to simplify analyses and transformations.
- Leveraged the llvm.launder.invariant.group intrinsic to perform SSA renaming that serves as “fence”.
- Implemented a generic SSA update utility.
- Utilized scoped alias metadata representation to preserve no-alias information after outlining.

experience what's inside™