Making LoopAccessAnalysis more precise

Ramkumar Ramachandra

Codasip

April 16, 2025

LoopAccessAnalysis is a dependency analysis built on ScalarEvolution that is used by:

- LoopVectorize
- SLPVectorize
- LoopVersioning
- LoopDistribute
- LoopLoadElimination

Runtime checks are the raison d'être of LAA:

```
void saxpy(size_t n, const float a, const float *x, float *y) {
  for (size_t iv = 0; iv < n; ++iv)
    y[iv] += a * x[iv];
}
```

Trivally safe to vectorize with `float *` replaced with `float *restrict`.

Non-trivial analysis (determined safe):

```
for (size_t iv = 1; iv < n; ++iv)
  x[2 * iv] = x[2 * iv - 1];
```

Dependence is either between a load and a store, or a store and another store.

Forward dependence:

```
for (size_t iv = 1; iv < n; ++iv)
  x[iv - 1] += x[iv];
```

Backward (loop-carried) dependence:

```
for (size_t iv = 0; iv < n - 1; ++iv)
  x[iv + 1] = x[stride * iv];
```

Here, stride is symbolic: LAA generates a predicate `stride == 1`, which is used by LoopVersioning to generate a unit-strided version of the loop.

Variations where LAA falls over:

```
for (size_t iv = 1; iv < n; ++iv)
  x[iv][iv] = x[iv][iv - 1];
```

```
for (size_t iv = 0; iv < n; ++iv)
  x[2 * iv] = x[2 * iv + 1];
```

```
for (size_t iv = 0; iv < n; ++iv)
  x[3 * iv] = x[7 * iv];
```

LAA reasons based on simple SCEV expressions. It is not theory-based, and is something that works in practice.

```
struct DepDistanceStrideAndSizeInfo {
    const SCEV *Dist;
    uint64_t MaxStride;
    std::optional<uint64_t> CommonStride;
    bool ShouldRetryWithRuntimeCheck;
    uint64_t TypeByteSize;
    bool AIsWrite;
    bool BIsWrite;
};
```

Emphasis: The reasoning within LAA is pure engineering.

Dependence distance is an SCEV minus:

```
const SCEV *Dist = SE.getMinusSCEV(Sink, Src);
```

Strides of Src and Sink from AddRecs:

```
std::optional<int64_t>
getStrideFromAddRec(const SCEVAddRecExpr *AR, const Loop *Lp, Type *AccessTy,
                    Value *Ptr, PredicatedScalarEvolution &PSE);
```

Stride versioning in case of non-constant stride:

```
// Stride >= TripCount
if (SE->isKnownPositive(StrideMinusBETaken)) {
  LLVM_DEBUG(
      dbgs() << "LAA: Stride>=TripCount; No point in versioning as the "
                "Stride==1 predicate will imply that the loop executes "
                "at most once.\n");
  return;
}
```

```
// We can only analyze innermost loops.
if (!TheLoop->isInnermost()) {
  LLVM_DEBUG(dbgs() << "LAA: loop is not the innermost loop\n");
  recordAnalysis("NotInnerMostLoop") << "loop is not the innermost loop";
  return false;
}
```

In contrast, DependenceAnalysis is a complex beast that is theory-based: users are LoopUnrollAndJam and LoopInterchange, which fundamentally need outer-loop analysis.

The kind of indexing and loop-nests that LAA can analyze, and where it really shines:

```
for (size_t oiv = 32; oiv < n; ++oiv)
  for (size_t iv = 0; iv < 256; ++iv)
    x[oiv + iv] = x[iv];
```

Here, LAA could deem it safe for a certain maximum vector-width, or generate RT-checks.

```
Memory dependences are safe
Dependences:
  Forward:
      %1 = load i32, ptr %gep.mul.2, align 4 ->
      store i32 %add, ptr %gep, align 4

Run-time memory checks:
Grouped accesses:

Non vectorizable stores to invariant address were not found in loop.
SCEV assumptions:

Expressions re-written:
```

```
Memory dependences are safe with run-time checks
Dependences:
Run-time memory checks:
Check 0:
  Comparing group ([[GRP1:0x[0-9a-f]+]]):
    %gep.dst = getelementptr i32, ptr %dst, i64 %iv.2
  Against group ([[GRP2:0x[0-9a-f]+]]):
    %gep.src = getelementptr inbounds i32, ptr %src, i32 %iv.3
  Grouped accesses:
    Group [[GRP1]]:
      (Low: ((4 * %iv.1) + %dst) High: (804 + (4 * %iv.1) + %dst))
        Member: {((4 * %iv.1) + %dst),+,4}<%inner.loop>
      Group [[GRP2]]:
      (Low: %src High: (804 + %src))
        Member: {%src,+,4}<nuw><%inner.loop>

Non vectorizable stores to invariant address were not found in loop.
SCEV assumptions:
  Equal predicate: %offset == 1

Expressions re-written:
  [PSE] %gep.dst = getelementptr i32, ptr %dst, i64 %iv.2:
    {((4 * %iv.1) + %dst),+,(4 * (sext i32 %offset to i64))<nsw>}<%inner.loop>
    --> {((4 * %iv.1) + %dst),+,4}<%inner.loop>
```

Issues with LAA:

1. Inability to reason about outer-loops
2. Inability to reason about multiple array indices
3. Relies on finding "array bounds" to insert RT-checks
4. Always-false runtime checks
5. Spurious false dependencies
6. Few contributions from a small contributor-pool

LoopAccessAnalysis: The analysis we have, but is it the analysis we deserve? □