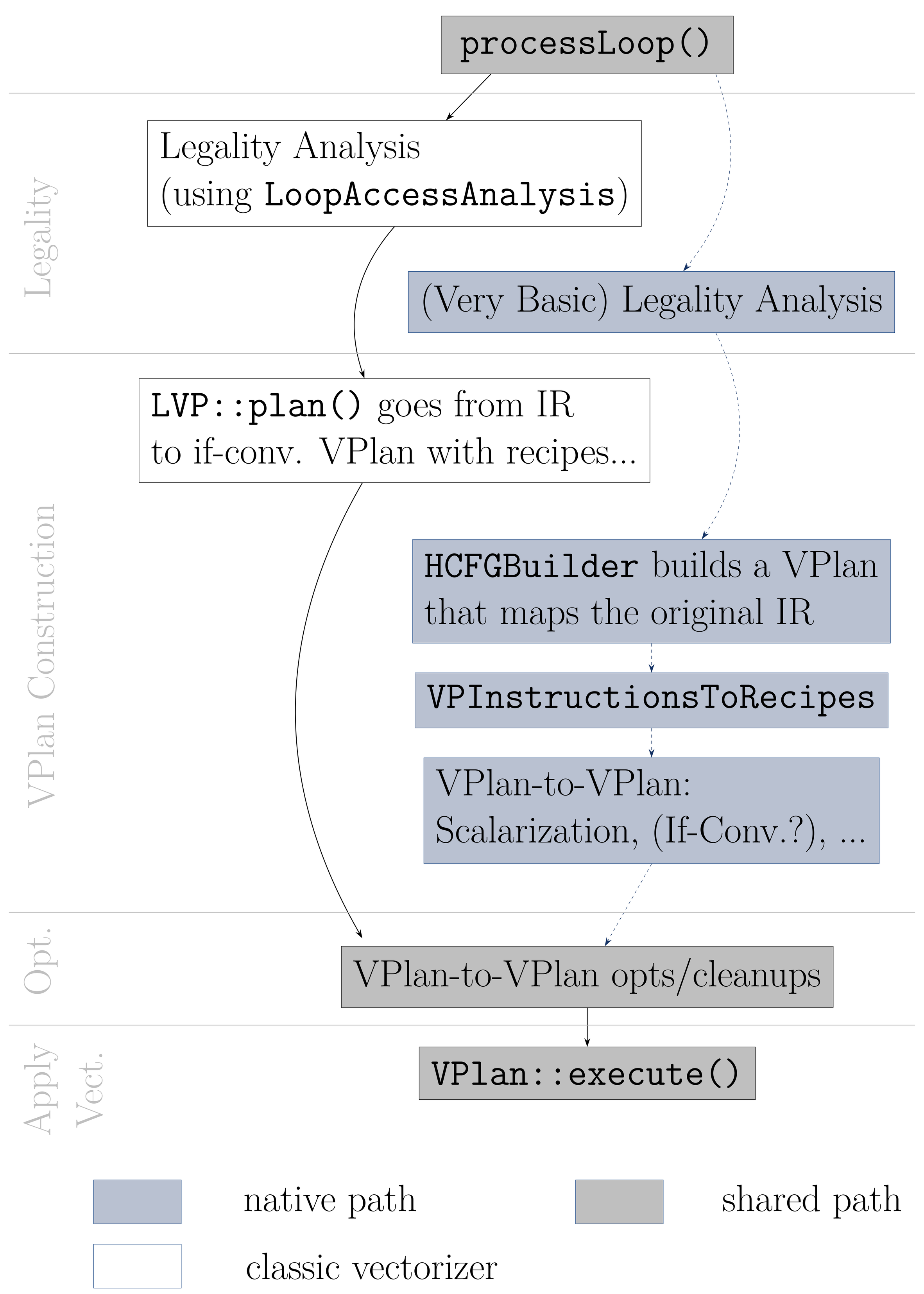
Improved Outer Loop Vectorization in LLVM

Summary

In most loop nests, vectorizing the inner-most loop is the best thing to do. However, there are exceptions where outer-loop vectorization is a better choice (e.g. for matrix multiplication). Currently, outer-loop vectorization is only supported in LLVM through the VPlan-native path. The VPlan-native path is an alternative vectorization code-path that is purely pragma/metadata driven and has currently no memory-dependency checks or cost-model. The quality of the emitted code is also sub-optimal: there is no scalarization and every memory access is done using gathers/scatters.

How to improve this code-path? How can uniform or consecutive memory accesses be identified in outer-loop vectorization? How to avoid unnecessary vectorization of instructions like the address calculation of consecutive accesses?

Vectorization in LLVM: Classic and Native Paths



Scalar

```
for (size_t i = 0; i < N; i++) {
  float sum = 0.;
  for (size_t j = 0; j < M; j++) {
    float x = B[j]        // Access b)
            * C[j][i];    // Access c)
    sum += x;
  }
  A[i] = sum;
}
```

Inner-Loop Vectorization

```
for (size_t i = 0; i < N; i++) {
  float sum = 0.;
  // Pseudo-Vectorized inner loop:
  for (size_t j = 0; j < M; j += 8) {
    float[8] vec1 = B[j..j+8];
    float[8] vec2 =
        strided_load(&C[j][i], N); // Slow!
    sum += reduce_add(vec1 * vec2);
  }
  A[i] = sum;
}
```

Outer-Loop vectorization

```
// Pseudo-Vectorized outer loop:
for (size_t i = 0; i < N; i += 8) {
  float[8] sum = { 0., ... };
  for (size_t j = 0; j < M; j++) {
    float[8] vec1 = dup(B[j]);
    float[8] vec2 = C[j][i..i+8];
    sum += vec1 * vec2;
  }
  A[i..i+8] = sum;
}
```

- Striding Accesses become Consecutive
- Element-Wise Add instead of Reduction

Find Uniform/Consec. Accesses

`LoopAccessAnalysis` looks for `SCEVAddRecExprs` to find consecutive accesses. When doing outer-loop vectorization, this is not enough. For example, the SCEV of access c) above is:

`{{%C,+,4}<%i_loop>,+,(4 * %L)}<%j_loop>`

The approach here is to 'unpeel' SCEV expressions that do not change the distance between steps of the vectorized loops. This means that `SCEVAdd{Expr/RecExpr}`s can be unpeeled if the step/rhs operands are loop invariant.

Find Scalarizeable Instructions

Current limitation: Only works if the def-use chain in the loop body has no cycles.

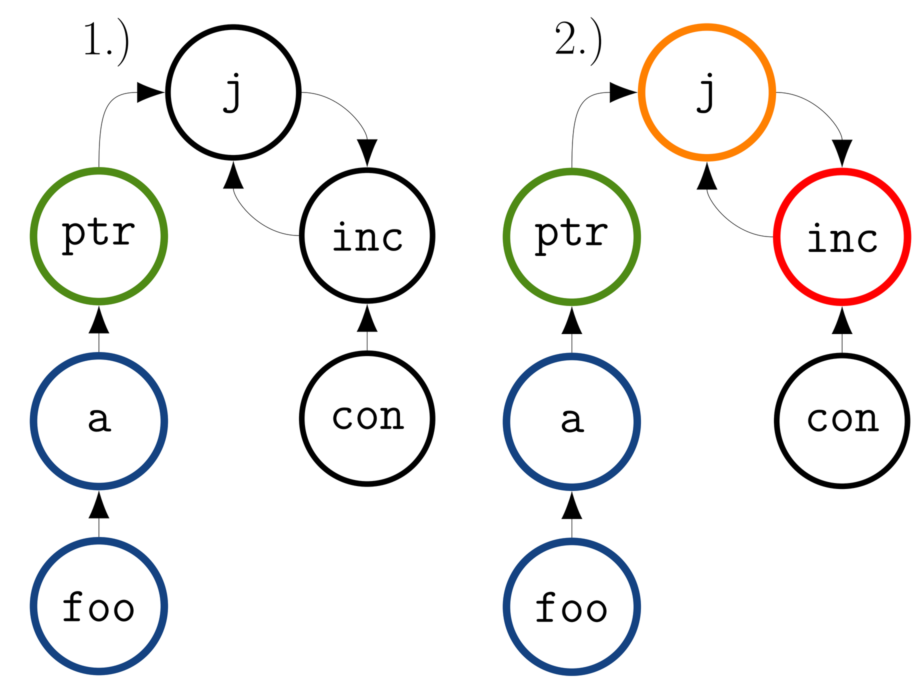
Proposed solution:

- Keep existing idea: Scalarize if all uses are scalar
- Recursively go up operands of scalarized instrs
- **If only non-scalar use is a loop-header PHI, assume it can be scalarized!**
- If all uses of PHI became scalar, all is fine!
- Otherwise, rollback.

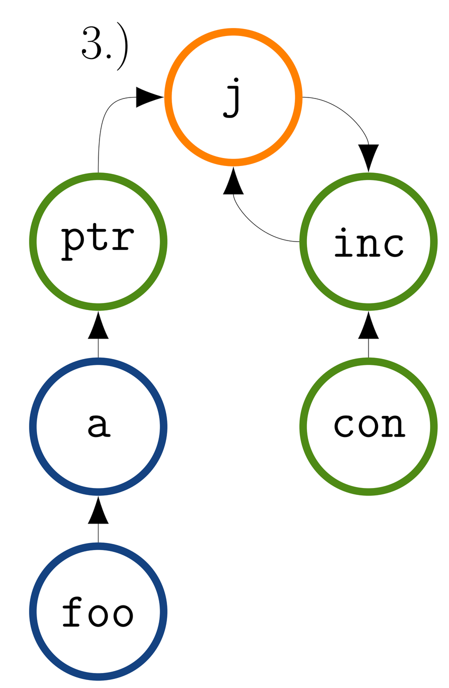
Example (Use Graph):

```
inner_loop:
 %j = phi i64 [0, ...], [%inc, %inner_loop]
 %ptr = getelementptr float, ptr %A, i64 %j
 %a = load float, ptr %ptr
 %used_outside = foo(%a)
 [...]
 %inc = add i64 %j, 1
 %con = icmp eq i64 %inc, %M
 br %con, %inner_loop, %inner_exit
```

1.) 2.)



3.)

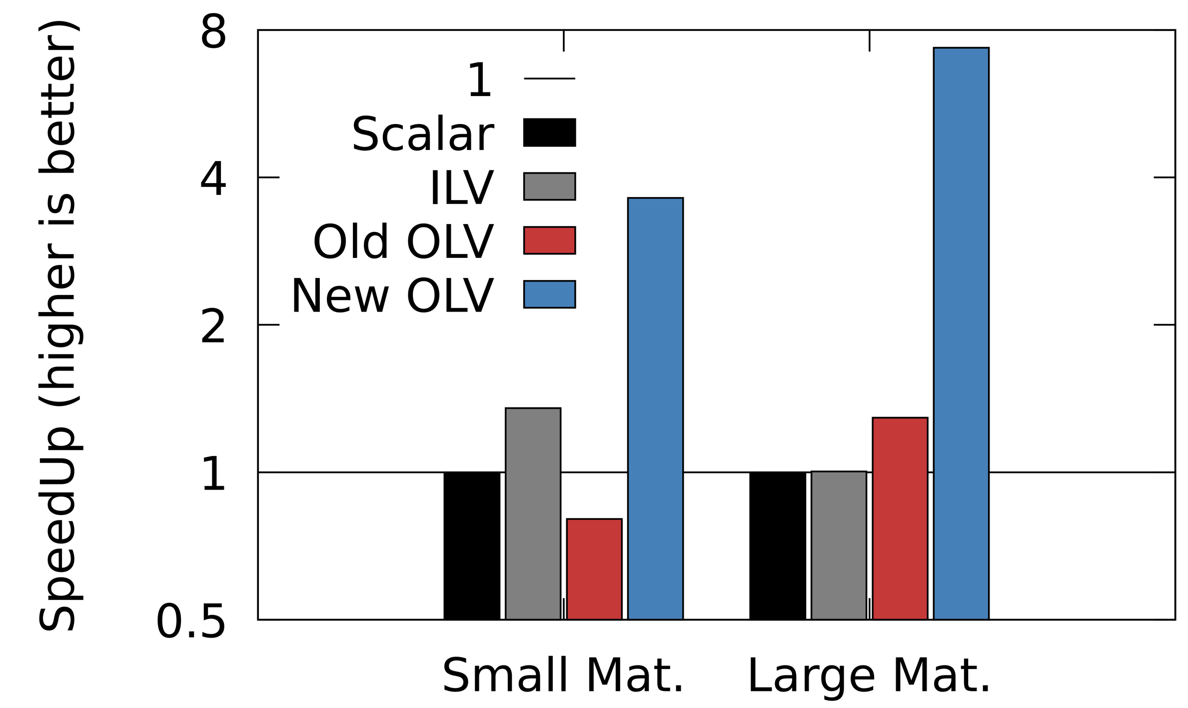


The load `a` uses `ptr` as scalar, and so does `con` use `inc`. There is a def-use cycle between `j` and `inc`. When visiting `j`, assume it can be scalarized. `inc` is has no other non-scalar user, so the assumption was fine!

Visit Order: `foo` → `a` → `ptr` → `j` → `con` → `inc` → `j`

Alternative solution would be to have vectorization-requiring "sinks" (like in DCE, e.g. reductions or value operand of store), and to recursively go up operands.

Results for Matrix Multiplication on aarch64 (Graviton3e)



Small Matrix: 10^5 Entries (Everything fits in L1 Cache, Tiling), Large Matrix: 10^8 Entries

Conclusion and Future Work

Conclusion:

- Very large perf. gains possible!
- Current upstream functionality of very limited use
- Can be improved: memory dep. checks, more flexible code, ...

Future Work:

- Memory Dependency Analysis with runtime pointer checks?
- VPlan-based Cost Model?
- Ability to compare costs of VPlans with different 'root' loops?

Related Work

- 'RV: A Unified Region Vectorizer for LLVM' by Simon Moll was a out-of-tree vectorizer capable of outer-loop vectorization.
- 'Extending LoopVectorize to Support Outer Loop Vectorization Using VPlan' by Intel is the foundation for the improvements suggested here.

Contact Information

- Web: SiPearl.com
- lou.knauer@sipearl.com
- etienne.renault@sipearl.com