Conditionals in Tensor IR

Fused vs Unfused Comparisons

One of the big distinctions between the various conditional representations is whether or not the conditional fuses an arbitrary comparison. This is best understood with some small examples. In an unfused approach, we use two fundamental operations. The first, comparison, would be implemented as `C = A cmp B` while the second, selection, would be `F = C ? D : E`. In contrast, the fused approach uses only a single operation to evaluate the entire thing: `F = (A cmp B) ? D : E`.

It's clear that the fused approach is strictly more expressive than the unfused approach (each of the unfused operators can be implemented with the fused operator). Thus the criteria for selecting between must be driven by either performance or ease of use.

On the performance side, the fused approach can make a meaningful difference in performance when dealing with CPU vectors. In the unfused approach the representation of a boolean is baked into the IR as either 0 or 1, since the comparison operation has to produce it for the selection operation to consume it. In many vector ISAs, though, per-lane booleans are better represented as 0 or -1, or by using dedicated mask/predicate registers. The fused approach effectively defers the decision of how to represent the boolean to the code generator, allowing it to choose the representation that is right for the particular HW context.

On the ease of use side, there's not a clear winner. The individual operations are simpler in the unfused approach, at the cost of a single logical operation sometimes requiring multiple operations. The fused approach gets more done in a single operation at the cost of some mental overhead from having four operands - but often resulting in a smaller expression tree overall. In either case, we will at some point need boolean expression simplification to simplify redundant conditions.

So far the recommendation was to standardize on fused conditionals.

Expression Conditionals vs Statement Conditionals

Tensor IR contains both expression conditionals (`CompareSelect` and `IfThenElse`), as well as statement conditionals (`Cond`). Expression conditionals are defined by being functional in nature: there is no side effect from duplicating the conditional, evaluating it twice, etc. They are an important ingredient in expression important operators like ReLU:

```
store (((load A) >= 0.0) ? (load A) : 0.0), B
```

Statement conditionals, on the other hand, are needed to represent conditional side-effects, typically stores. These are less common, but they may arise in scatter operations. As far as I can tell nothing currently generates these.

Lazy vs Eager Conditionals

Within the space of expression conditionals, Tensor IR contains a distinction between eager conditionals (`CompareSelect`) and lazy (or short-circuit) conditionals (`IfThenElse`). The important distinction between these two is when the selectands (the `C` and `D` sub-expressions in the operation `(A cmp B) ? C : D`) are evaluated. In eager conditionals, all of the operands are allowed to be evaluated before the conditional selection is made. In lazy/short-circuit conditionals, the `A` sub-expression must be evaluated first, and then only one of `C` or `D` is evaluated.

This may seem like an esoteric distinction, but it becomes an important distinction in kernels like concat loops. In that situation, we often have code of the form:

```
for i = 0 ... N
    store ((i < 16) ? load A[i] : load B[i-16]), C
```

The problem that arises is that evaluating `A[i]` when `i >= 16` will cause an out-of-bounds access that will crash the program.

It would be tempting, then, to make all conditionals lazy to avoid this problem, but there are equally situations where it's desirable to have eager conditionals. The ReLU example earlier is a canonical case where eager evaluation is fine. And we can't just defer to the code generator to determine whether eagerness is safe, because vectorization inherently means speculating the selectands.

One option in here would be to make all expression conditionals eager, and use statement conditionals in place of lazy conditionals. This would result in some code duplication, and some added complexity in kernel generation for reshaping ops.