How to write tests using FileCheck

What is FileCheck

FileCheck can be seen as an advanced version of grep. We use it for writing small annotated unit tests for optimization passes. FileCheck used in PyTorch is inspired by LLVM FileCheck Tool, but is not the same. FileCheck is available for writing both C++ and python tests.

How does it work

Let's look at a test written with FileCheck. The following test verifies that CSE pass removes one out of two similar `aten::mul` nodes. Here is how the test looks like:

```
def test_cse():
    input_str = """graph(%a : Tensor, %b : Tensor):
      # CHECK: aten::mul
      %x : Tensor = aten::mul(%a, %b)
      # Check that the second aten::mul is removed by CSE.
      # CHECK-NOT: aten::mul
      %y : Tensor = aten::mul(%a, %b)
      # CHECK: return
      return (%x, %y)
      """
    parsed = parse_ir(input_str)
    optimized = run_cse(parsed)
    FileCheck().run(input_str, optimized)
```

Let's look in detail at how it works. First, the input string is parsed by `parse_ir`. At that stage all annotations are ignored since they are written in comments, so this is what parser essentially sees:

```
graph(%a : Tensor, %b : Tensor):
      %x : Tensor = aten::mul(%a, %b)
      %y : Tensor = aten::mul(%a, %b)
      return (%x, %y)
```

We then run CSE on the parsed IR and expect it to remove the second `aten::mul`, which is redundant. After CSE our IR looks like this:

```
graph(%a : Tensor, %b : Tensor):
      %x : Tensor = aten::mul(%a, %b)
      return (%x, %x)
```

And now we run `FileCheck` passing to it both original input string and the optimized IR. From the input string `FileCheck` ignores everything except `# CHECK` pragmas and essentially it sees the input string like this:

```
      # CHECK: aten::mul       (1)
```

```
# CHECK-NOT: aten::mul   (2)
# CHECK: return          (3)
```

It then checks that the optimized IR satisfies the specified annotations. It first finds string `%x : Tensor = aten::mul(%a, %b)` matching the annotation (1), then it finds string `return (%x, %x)` matching the annotation (3), and since there were no lines matching `aten::mul` after the match (1) and before the match (3), the annotation (2) is also satisfied.

One could also register FileCheck annotations using a builder API. To generate annotations from the example above one would write:

```
FileCheck().check("aten::mul")      \
           .check_not("aten::mul") \
           .check("return")        \
           .run(optimized)
```

Supported pragmas

- `CHECK: <pattern>` Scans the input until `PATTERN` is found. Fails if the pattern is not found.
- `CHECK-NEXT: <pattern>` Scans the input on the line immediately following the previous CHECK until `PATTERN` is found. Fails if the pattern is not found on that line.
- `CHECK-NOT: <pattern>` Scans the input and fails if `PATTERN` is found on any line. The scan stops when a match for a next `CHECK` is found.
- `CHECK-SAME: <pattern>` Checks that PATTERN is found in the line of the last match.
- `CHECK-COUNT-<num>: <pattern>` Scans the input and succeeds when a line containing at least `NUM` entries of `PATTERN` is found.
- `CHECK-COUNT-EXACTLY-<num>: <pattern>` Scans the input and succeeds when a line containing exactly `NUM` entries of `PATTERN` is found.
- `CHECK-DAG: pattern` Works similar to the usual `CHECK` pragma, but also matches if there exists a way to reorder the CHECK-DAG pragmas to satisfy all patterns. For example the following pattern:

  ```
  # CHECK: foo
  # CHECK-DAG: bar
  # CHECK-DAG: ham
  # CHECK: end
  ```

  would match the following input (note that `ham` and `bar` are swapped):

  ```
  foo
  ham
  bar
  ```

```
end
```