Triton CPU

May 30, 2024

Direction

- Lowering TTIR to `vector` MLIR is an effective approach. We will use it.

- What about the approach using direct vectorization?
    - It needs the same information that we need to lower to `vector` dialect
    - E.g., `tt.load` is lowered to `load <128 x float>` or `call <1024 x float> @llvm.masked.load.v1024f32.p0`.
    - If necessary, we could add another pass for manual intrinsic-based vectorization

Lowering to vector dialect (1/2)

- Pointers
  - Two styles
    - `tt.make_tensor_ptr`
    - “Idiom”: `tt.make_range` + `tt.splat` + `tt.addptr`
  - `make_tensor_ptr` is easier for CPU
  - In GPUs, `add_rewrite_tensor_pointer` converts it into the idiom
  - Simple cases for the idiom are implemented (e.g., using `vector.splat`)
  - An idea of “reverse” such idioms into `make_tensor_ptr`? Too fragile?

Lowering to vector dialect (2/2)

- Reductions
  - Lowers to vector.multi_reduction
  - Fused-softmax runs

```
%11 = vector.maskedload %10[%c0], %7, %cst_3 : ... into vector<128xf32>
%12 = vector.multi_reduction <maxnumf>, %11, %cst_1 [0] : vector<128xf32> to f32
%13 = vector.splat %12 : vector<128xf32> loc(#loc10)
%14 = arith.subf %11, %13 : vector<128xf32> loc(#loc10)
%15 = math.exp %14 : vector<128xf32> loc(#loc11)
%16 = vector.multi_reduction <add>, %15, %cst_0 [0] : vector<128xf32> to f32
```

Study: Generated Code Quality for Vector Addition

- We evaluated the quality of the generated code (`vector` dialect + LLVM)
  - Example: vector addition with different BLOCK_SIZE, and with or without masks
  - TritonCPU AVX code examples: link
- We compared the code from C++ code with AVX intrinsics
  - LLVM code gen: https://godbolt.org/z/f1KE7frxe
  - GCC code gen: https://godbolt.org/z/zjdvEqYnr
  - LLVM generated slightly inefficient mask register spill code. GCC spills %k register to GPR.
- Findings
  - The two approaches seem to generate similar quality code
- TODO: Evaluate the quality for the reductions

Potential fast path optimization

- Eliding mask operations?
- For certain cases, masks are not necessary except for the last iteration
- Measured expected speedups for GEMV/GEMM? Good enough to invest?

Supporting ARM (Neon/SVE)

- We can build and run triton-cpu on a M1 machine (only Neon)
- We want to cross compile to leverage SVE
- Code quality?
    - For vector addition, the code quality looks on par with AVX
    - Masked code seems much complex, but looks okay
    - TODO: evaluate the code quality for the reductions

Pipeline structure

- Current Nvidia/AMD have a single big pipeline with many `populate*` style
- For CPU, we attempt to use more fine-grained pipeline style

Plans

- Feature completeness
- Performance
- Runtime
- CI
    - Intel will support for x86
    - Meta might support for ARM(?)
- Coordination: spreadsheet ⇒ Will move to Github Issues

Contributors

- Intel: Ilya Enkovich
- Meta: Minjang Kim, Ruiqui Gao (intern), Digant Desai, Elliot Gorokhovsky