Triton CPU Design Discussion

Lukas Sommer & Mehdi Goli, Codeplay Compiler R&D

2024-04-11

General

- Portability between CPU & GPU
  - What is the goal for portability?
    - Same code, same grid & block size
    - Same code, different grid & block size
    - Slightly altered code
- Mapping for blocks
  - One/Multiple blocks per thread
    - i.e., do not distribute block across threads
  - Vectorize inside block
  - Block Size == Vector Size?

Design

Design Options

- Two main options discussed in community call
  - Implement TritonCPU dialect analog to TritonGPU and use for lowering
  - Use and extend triton-shared project to get to Linalg & “core” MLIR
- TritonCPU dialect
  - Most of the operations in TritonGPU not useful on TritonCPU (async, local memory)
  - What would be in the TritonCPU dialect?
  - Could add operations to represent analysis result/intermediate lowering step (cf. TTS dialect)
- triton-shared
  - Evaluted triton-shared with TorchInductor generated Triton

Failed Compilation

- Used five models used for TPP-MLIR evaluation
  - `mnist`
  - `bert`
  - `linear`
  - `conv`
  - `Resnet18`

- https://github.com/plaidml/mlir-generator/tree/main/pytorch/torch-dynamo/models

Failed Compilation

- For all five models, compilation with triton-shared failed
- For four out of five, `ExtractStridedMetadata` operation from `memref` dialect is created with insufficient information
  - Structured memory access analysis fails to match access
  - In fallback, build function unable to determine result type, causes LLVM assertion to trigger
  - Probably just an implementation error, but shows gaps in Triton language coverage
- Fifth benchmark fails due to unsupported Triton language construct
  - Triton-shared currently only supports scalar condition in assert
  - TorchInductor generates assert with unsupported assert condition
  - Shows another gap in coverage of Triton language

Unnecessary Copies – Input

```
1  tt.func @kernel(%afloat : !tt.ptr<bf16>, %res : !tt.ptr<bf16>) {
2    %0 = tt.make_range {end = 128 : i32, start = 0 : i32} : tensor<128xi32>
3    %1 = tt.splat %afloat : (!tt.ptr<bf16>) -> tensor<128x!tt.ptr<bf16>>
4    %2 = tt.addptr %1, %0 : tensor<128x!tt.ptr<bf16>>, tensor<128xi32>
5    %afm = tt.load %2 {cache = 1 : i32, evict = 1 : i32, isVolatile = false} : tensor<128xbf1
6    ...
7  }
```

Compile through triton-shared with:

```
--triton-to-structured --canonicalize --triton-arith-to-linalg --structured-to-memref
```

Unnecessary copies – Result

```
1  func.func @kernel(%arg0: memref<*xbf16>, %arg1: memref<*xbf16>, %arg2: i32, %arg3: i32, %arg4: i32) {
2      %cst = arith.constant 0.000000e+00 : f32
3      %reinterpret_cast = memref.reinterpret_cast %arg0 to offset: [0], sizes: [128], strides: [1] :
4          memref<*xbf16> to memref<128xbf16, strided<[1]>>
5      %alloc = memref.alloc() : memref<128xbf16>
6      memref.copy %reinterpret_cast, %alloc : memref<128xbf16, strided<[1]>> to memref<128xbf16>
7      %0 = bufferization.to_tensor %alloc restrict writable : memref<128xbf16>
8      ...
9  }
```

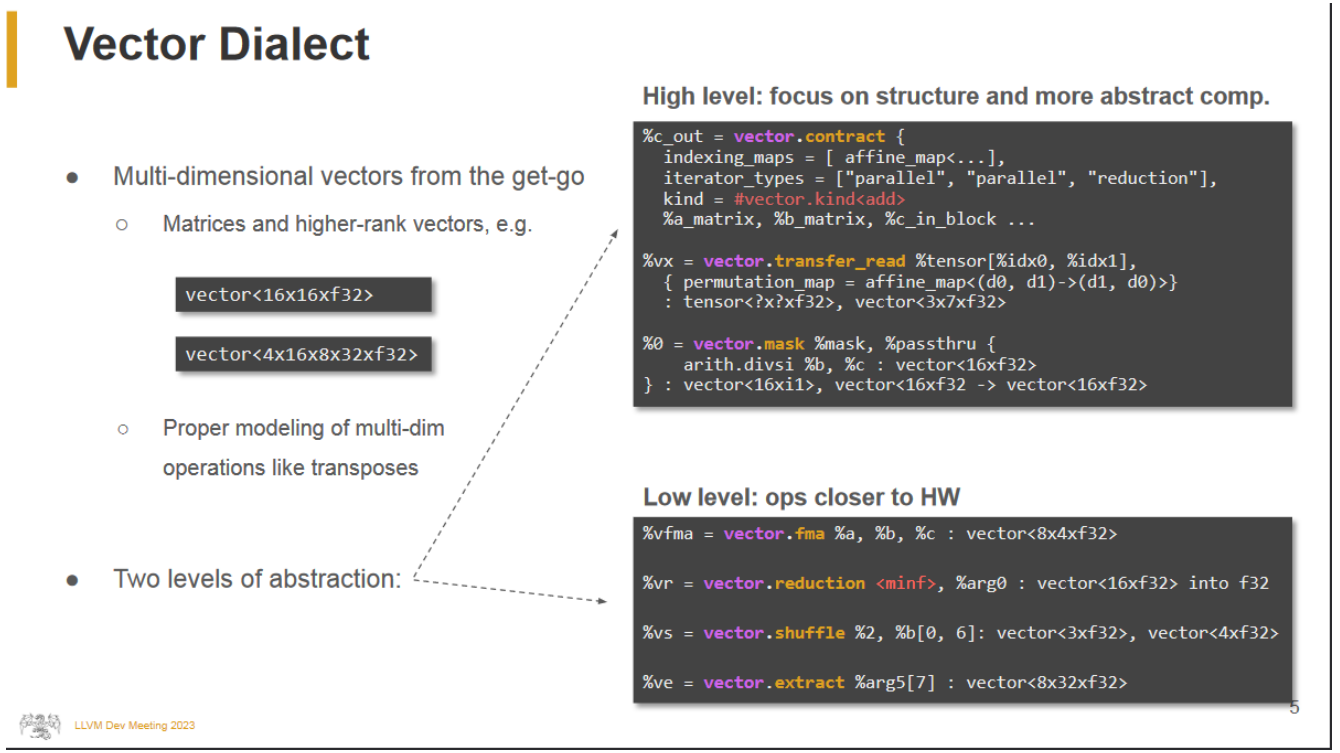
- Copy and allocation seems unnecessary in this case
- Impacts performance, as allocation and memory copy expensive
- What is the reason to insert these copies?

- Also asked for clarification: https://github.com/microsoft/triton-shared/discussions/126

MLIR Entry Dialect

- What is the best dialect to enter the “core” MLIR world?
- Affine
  - Could leverage MLIR SuperVectorizer
  - Cons:
    - SuperVectorizer development slow/stale
    - Suboptimal, as information is first discarded to then be recovered by vectorizer
- Linalg
  - Allows re-use of vectorizer from other flows
  - Cons:
    - Contains traces of raising
    - Not really required, as potentially no tiling required
- Vector dialect
  - Multiple levels of vector abstractions from high-level to HW-level
  - Supports masking

Vector Dialect



"These abstractions serve to separate concerns between operations on memref (a.k.a buffers) and operations on vector values" - https://mlir.llvm.org/docs/Dialects/Vector/

Caballero & Warzynski, Vectorization in MLIR, LLVM Dev Meeting 2023
https://llvm.org/devmtg/2023-10/slides/techtalks/Warzynski-Caballero-VectorizationinMLIR.pdf

Multi-threading

- Avoid non-trivial effort for implementation of multi-threading runtime
- OpenMP provides necessary features
  - Multi-threading
  - NUMA affinity management
  - Different strategies for how threads behave between parallel sections
- LLVM OpenMP (libomp) widely supported
  - Linux, Windows, Mac OS
  - Different architectures, including X86 and Aarch 64

Disclaimers

A wee bit of legal

Performance varies by use, configuration and other factors.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details.

No product or component can be absolutely secure.

Your costs and results may vary.

Intel technologies may require enabled hardware, software or service activation.