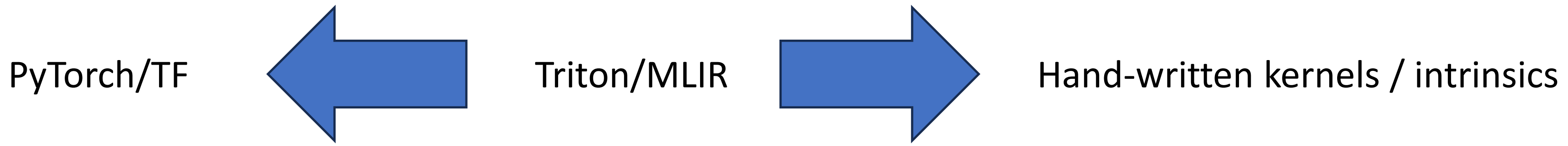
Experience with Triton Lowering and Optimization for Qualcomm® Hexagon™ NPU Target

Javed Absar, Principal Engineer, Qualcomm Technologies International, Ltd.
Muthu M. Baskaran, Principal Engineer, Qualcomm Technologies, Inc.

Contents

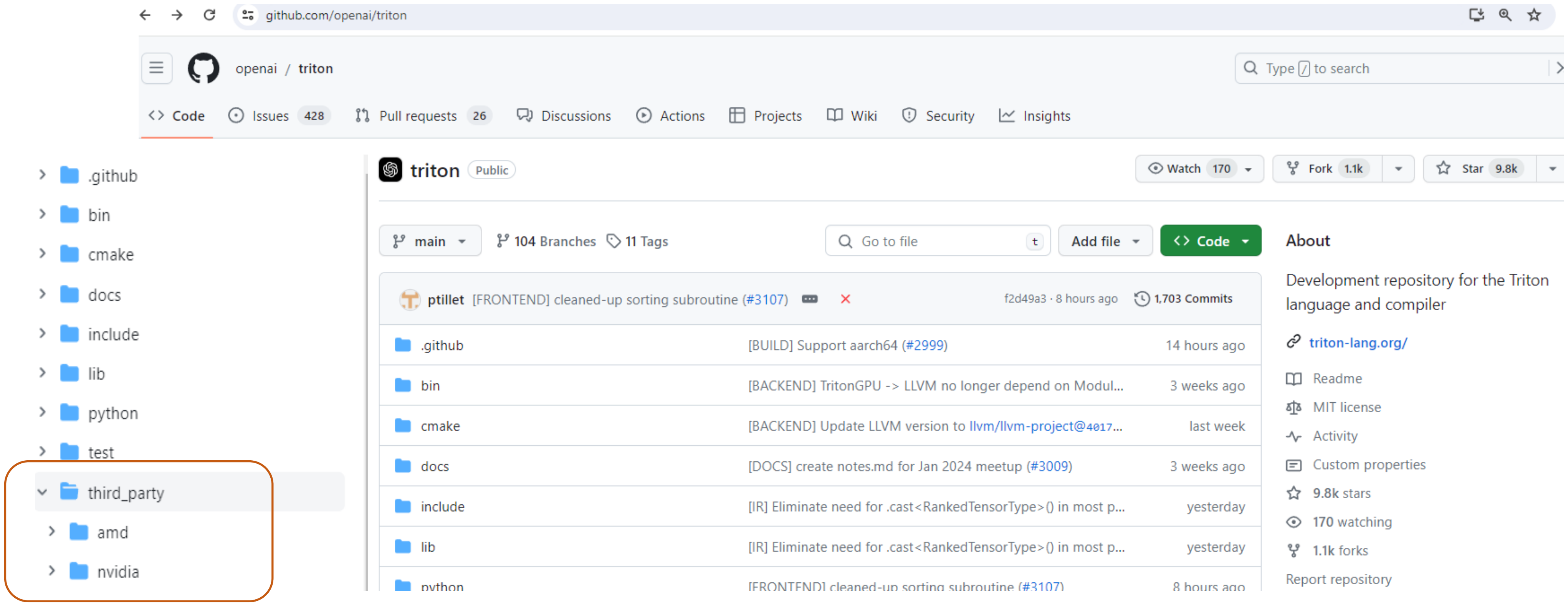
What is Triton ?

- Python DSL to write ML kernels
  - Low development time / high performance
- Language and Compiler



What is Triton ?

- Pioneered at Open-AI for NVIDIA GPUs
- Open-sourced



github.com/openai/triton

openai / triton

Code · Issues 428 · Pull requests 26 · Discussions · Actions · Projects · Wiki · Security · Insights

triton Public — Watch 170 · Fork 1.1k · Star 9.8k

main · 104 Branches · 11 Tags · Go to file · Add file · Code

ptillet [FRONTEND] cleaned-up sorting subroutine (#3107) — f2d49a3 · 8 hours ago — 1,703 Commits

.github	[BUILD] Support aarch64 (#2999)	14 hours ago
bin	[BACKEND] TritonGPU -> LLVM no longer depend on Modul...	3 weeks ago
cmake	[BACKEND] Update LLVM version to llvm/llvm-project@4017...	last week
docs	[DOCS] create notes.md for Jan 2024 meetup (#3009)	3 weeks ago
include	[IR] Eliminate need for .cast<RankedTensorType>() in most p...	yesterday
lib	[IR] Eliminate need for .cast<RankedTensorType>() in most p...	yesterday
python	[FRONTEND] cleaned-up sorting subroutine (#3107)	8 hours ago

About

Development repository for the Triton language and compiler

triton-lang.org/

Readme · MIT license · Activity · Custom properties · 9.8k stars · 170 watching · 1.1k forks

Report repository

- .github
- bin
- cmake
- docs
- include
- lib
- python
- test
- third_party
  - amd
  - nvidia

Softmax using Torch

$$softmax(X)_i = \frac{e^{X_i - X_{max}}}{\sum_{i=1}^{N} e^{X_i - X_{max}}}$$

```python
@torch.jit.script
def naive_softmax(x):
    """Compute row-wise softmax of X using native pytorch.
    We subtract the maximum element in order to avoid overflows. Softmax is invariant to
    this shift.
    """
    # read  MN elements ; write M  elements
    x_max = x.max(dim=1)[0]

    # read MN + M elements ; write MN elements
    z = x - x_max[:, None]

    # read  MN elements ; write MN elements
    numerator = torch.exp(z)

    # read  MN elements ; write M  elements
    denominator = numerator.sum(dim=1)

    # read MN + M elements ; write MN elements
    ret = numerator / denominator[:, None]

    # in total: read 5MN + 2M elements ; wrote 3MN + 2M elements
    return ret
```

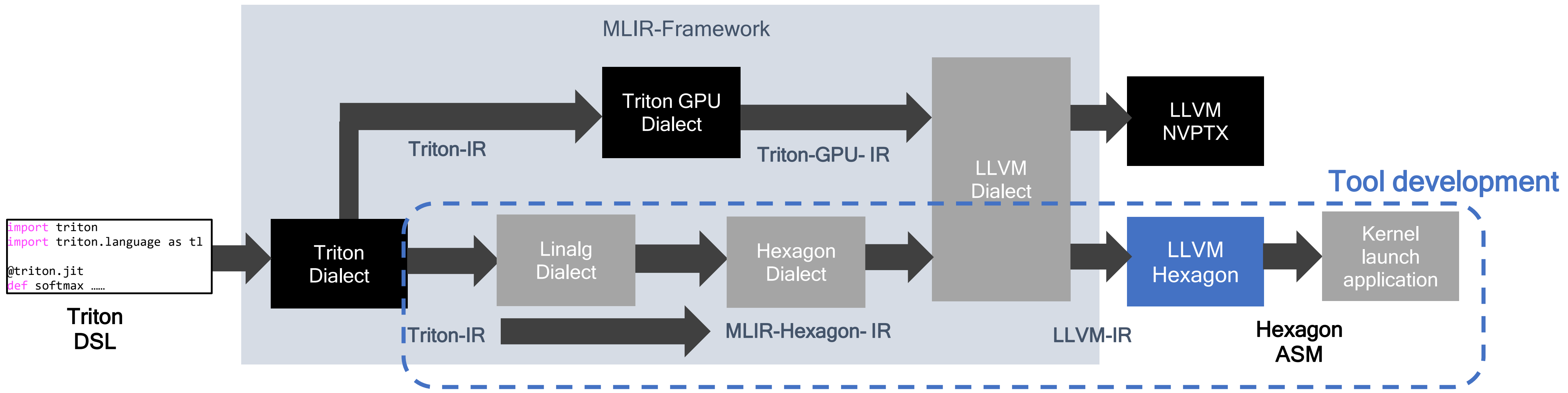
Softmax using Triton

```
# Triton softmax kernel works as follows: each program loads a row of the input matrix X,
# normalizes it and writes back the result to the output Y.
@torch.jit.script
def softmax_kernel(output_ptr, input_ptr, input_row_stride, output_row_stride, n_cols, BLOCK_SIZE: tl.constexpr):
    # The rows of the softmax are independent, so we parallelize across those
    row_idx = tl.program_id(0)
    # The stride represents how much we need to increase the pointer to advance 1 row
    row_start_ptr = input_ptr + row_idx * input_row_stride

    col_offsets = tl.arange(0, BLOCK_SIZE)
    input_ptrs = row_start_ptr + col_offsets
    # Load the row into SRAM, using a mask since BLOCK_SIZE may be > than n_cols
    row = tl.load(input_ptrs, mask=col_offsets < n_cols, other=-float('inf'))

    # Subtract maximum for numerical stability
    row_minus_max = row - tl.max(row, axis=0)

    # Note that exponentiation in Triton is fast but approximate (i.e., think __expf in CUDA)
    numerator = tl.exp(row_minus_max)
    denominator = tl.sum(numerator, axis=0)
    softmax_output = numerator / denominator

    # Write back output to DRAM
    output_row_start_ptr = output_ptr + row_idx * output_row_stride
    output_ptrs = output_row_start_ptr + col_offsets
    tl.store(output_ptrs, softmax_output, mask=col_offsets < n_cols)
```

Our Triton Work

- Started last year
- Small team
- Two approaches
  - Direct from Triton-IR to LLVM-IR
  - Triton to Linalg

Our Triton Flow


```
import triton
import triton.language as tl

@triton.jit
def softmax ……
```

Target Architecture

- Hexagon Processors
  - 4-way multi-threaded VLIW
  - Hexagon Vector eXtensions (HVX) – SIMD co-processor
  - Vector registers, vector compute elements, and dedicated memory
  - Two vector lengths – 512 bits (64B), 1024 bits (128B)

Triton Lowering

$$softmax(X)_i = \frac{e^{X_i - X_{max}}}{\sum_{i=1}^{N} e^{X_i - X_{max}}}$$

```
# Triton softmax kernel works as follows: each program loads a row of the input matrix X,
# normalizes it and writes back the result to the output Y.
@torch.jit.script
def softmax_kernel(output_ptr, input_ptr, input_row_stride, output_row_stride, n_cols, BLOCK_SIZE: tl.constexpr):
    # The rows of the softmax are independent, so we parallelize across those
    row_idx = tl.program_id(0)
    # The stride represents how much we need to increase the pointer to advance 1 row
    row_start_ptr = input_ptr + row_idx * input_row_stride

    col_offsets = tl.arange(0, BLOCK_SIZE)
    input_ptrs = row_start_ptr + col_offsets
    # Load the row into SRAM, using a mask since BLOCK_SIZE may be > than n_cols
    row = tl.load(input_ptrs, mask=col_offsets < n_cols, other=-float('inf'))

    # Subtract maximum for numerical stability
    row_minus_max = row - tl.max(row, axis=0)

    # Note that exponentiation in Triton is fast but approximate (i.e., think __expf in CUDA)
    numerator = tl.exp(row_minus_max)
    denominator = tl.sum(numerator, axis=0)
    softmax_output = numerator / denominator

    # Write back output to DRAM
    output_row_start_ptr = output_ptr + row_idx * output_row_stride
    output_ptrs = output_row_start_ptr + col_offsets
    tl.store(output_ptrs, softmax_output, mask=col_offsets < n_cols)
```

Triton IR

```
 tt.func public @softmax (%arg0: !tt.ptr<f32, 1> … ) {
    %0 = tt.make_range {end = 1024 : i32, start = 0 : i32} : tensor<1024xi32> loc(#loc2)
    %1 = arith.muli %arg6, %arg2 : i32 loc(#loc3)
    %2 = tt.addptr %arg1, %1 : !tt.ptr<f32, 1>, i32 loc(#loc4)
    ...
    %9 = tt.load %6, %8, %cst {cache = 1 : i32, evict = 1 : i32, isVolatile = false} : tensor<1024xf32> loc(#loc1)
    %10 = "tt.reduce"(%9) ({
    ^bb0(%arg9: f32 loc(unknown), %arg10: f32 loc(unknown)):
      %23 = arith.maximumf %arg9, %arg10 : f32 loc(#loc28)
      tt.reduce.return %23 : f32 loc(#loc24)
    }) {axis = 0 : i32} : (tensor<1024xf32>) -> f32 loc(#loc24)
    %11 = tt.splat %10 : (f32) -> tensor<1024xf32> loc(#loc12)
    %12 = arith.subf %9, %11 : tensor<1024xf32> loc(#loc12)
    %13 = math.exp %12 : tensor<1024xf32> loc(#loc13)
    %14 = "tt.reduce"(%13) ({
    ^bb0(%arg9: f32 loc(unknown), %arg10: f32 loc(unknown)):
      %23 = arith.addf %arg9, %arg10 : f32 loc(#loc29)
      tt.reduce.return %23 : f32 loc(#loc26)
    }) {axis = 0 : i32} : (tensor<1024xf32>) -> f32 loc(#loc26)
    ...
    tt.return loc(#loc23)
  } loc(#loc)
} loc(#loc)
```

Linalg Conversion (Microsoft Team)

```
...
%6 = bufferization.to_tensor %alloc restrict writable : memref<1024xf32>
    %7 = bufferization.alloc_tensor() : tensor<f32>
    %inserted = tensor.insert %cst into %7[] : tensor<f32>

    %reduced = linalg.reduce ins(%6 : tensor<1024xf32>) outs(%inserted : tensor<f32>)
        dimensions = [0] (%in: f32, %init: f32) {
        %21 = arith.maximumf %in, %init : f32
        linalg.yield %21 : f32
      }
    %extracted = tensor.extract %reduced[] : tensor<f32>
    %8 = tensor.empty() : tensor<1024xf32>
    %9 = linalg.fill ins(%extracted : f32) outs(%8 : tensor<1024xf32>) -> tensor<1024xf32>
    %10 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
      ins(%6, %9 : tensor<1024xf32>, tensor<1024xf32>) outs(%6 : tensor<1024xf32>) {
    ^bb0(%in: f32, %in_7: f32, %out: f32):
      %21 = arith.subf %in, %in_7 : f32
      linalg.yield %21 : f32
    } -> tensor<1024xf32>
    %11 = linalg.generic {indexing_maps = [#map, #map],
                          iterator_types = ["parallel"]}
      ins(%10 : tensor<1024xf32>) outs(%10 : tensor<1024xf32>) {
    ^bb0(%in: f32, %out: f32):
      %21 = math.exp %in : f32
      linalg.yield %21 : f32
    } -> tensor<1024xf32>
```

Fusion

```
 %6 = bufferization.to_tensor %alloc restrict writable : memref<1024xf32>
    %7 = bufferization.alloc_tensor() : tensor<f32>
    %inserted = tensor.insert %cst into %7[] : tensor<f32>
    %8 = linalg.generic {indexing_maps = [#map, #map1], iterator_types = ["reduction"]} ins(%6 : tensor<1024xf32>)
outs(%inserted : tensor<f32>) {
    ^bb0(%in: f32, %out: f32):
      %18 = arith.maximumf %in, %out : f32
      linalg.yield %18 : f32
    } -> tensor<f32>
    %extracted = tensor.extract %8[] : tensor<f32>
    %9 = bufferization.alloc_tensor() : tensor<f32>
    %inserted_2 = tensor.insert %cst_0 into %9[] : tensor<f32>
    %10 = linalg.generic {indexing_maps = [#map, #map1], iterator_types = ["reduction"]} ins(%6 : tensor<1024xf32>)
outs(%inserted_2 : tensor<f32>) {
    ^bb0(%in: f32, %out: f32):
      %18 = arith.subf %in, %extracted : f32
      %19 = math.exp %18 : f32
      %20 = arith.addf %19, %out : f32
      linalg.yield %20 : f32
    } -> tensor<f32>
    %extracted_3 = tensor.extract %10[] : tensor<f32>
    %11 = tensor.empty() : tensor<1024xf32>
    %12 = linalg.generic {indexing_maps = [#map, #map], iterator_types = ["parallel"]} ins(%6 : tensor<1024xf32>) outs(%11 :
tensor<1024xf32>) {
    ^bb0(%in: f32, %out: f32):
      %18 = arith.subf %in, %extracted : f32
      %19 = math.exp %18 : f32
      %20 = arith.divf %19, %extracted_3 : f32
      linalg.yield %20 : f32
    } -> tensor<1024xf32>
```

Tiling/Bufferization/…

```
 %8 = scf.for %arg15 = %c0 to %c1024 step %c32 iter_args(%arg16 = %alloc_4) -> (memref<f32>) {
     %subview_9 = memref.subview %alloc[%arg15] [32] [1] : memref<1024xf32> to memref<32xf32, strided<[1], offset: ?>>
     %16 = vector.transfer_read %subview_9[%c0], %cst_1 {in_bounds = [true]} : memref<32xf32, strided<[1], offset: ?>>, vector<32xf32>
     %17 = memref.load %arg16[] : memref<f32>
     %18 = vector.broadcast %7 : f32 to vector<32xf32>
     %19 = arith.subf %16, %18 fastmath<fast> : vector<32xf32>
     %20 = math.exp %19 fastmath<fast> : vector<32xf32>
     %21 = vector.reduction <add>, %20, %17 fastmath<fast> : vector<32xf32> into f32
     %22 = vector.insertelement %21, %cst[%c0 : index] : vector<1xf32>
     %23 = vector.extract %22[0] : f32 from vector<1xf32>
     %24 = vector.broadcast %23 : f32 to vector<f32>
     %25 = vector.extractelement %24[] : vector<f32>
     memref.store %25, %arg16[] : memref<f32>
     scf.yield %arg16 : memref<f32>
   }
   %9 = memref.load %8[] : memref<f32>
   %alloc_5 = memref.alloc() {alignment = 64 : i64} : memref<1024xf32>
   %10 = scf.for %arg15 = %c0 to %c1024 step %c32 iter_args(%arg16 = %alloc_5) -> (memref<1024xf32>) {
     %subview_9 = memref.subview %alloc[%arg15] [32] [1] : memref<1024xf32> to memref<32xf32, strided<[1], offset: ?>>
     %subview_10 = memref.subview %arg16[%arg15] [32] [1] : memref<1024xf32> to memref<32xf32, strided<[1], offset: ?>>
     %16 = vector.transfer_read %subview_9[%c0], %cst_1 {in_bounds = [true]} : memref<32xf32, strided<[1], offset: ?>>, vector<32xf32>
     %17 = vector.broadcast %7 : f32 to vector<32xf32>
     %18 = arith.subf %16, %17 fastmath<fast> : vector<32xf32>
     %19 = math.exp %18 fastmath<fast> : vector<32xf32>
     %20 = vector.broadcast %9 : f32 to vector<32xf32>
     %21 = arith.divf %19, %20 fastmath<fast> : vector<32xf32>
     vector.transfer_write %21, %subview_10[%c0] {in_bounds = [true]} : vector<32xf32>, memref<32xf32, strided<[1], offset: ?>>
     %subview_11 = memref.subview %arg16[%arg15] [32] [1] : memref<1024xf32> to memref<32xf32, strided<[1], offset: ?>>
     memref.copy %subview_10, %subview_11 : memref<32xf32, strided<[1], offset: ?>> to memref<32xf32, strided<[1], offset: ?>>
     scf.yield %arg16 : memref<1024xf32>
   }
```

LLVM-IR

```
^bb11:  // pred: ^bb10
 %155 = llvm.getelementptr %40[%153] : (!llvm.ptr, i64) -> !llvm.ptr, f32
 %156 = llvm.getelementptr %155[%29] : (!llvm.ptr, i64) -> !llvm.ptr, f32
 %157 = llvm.load %156 {alignment = 4 : i64} : !llvm.ptr -> vector<32xf32>
 %158 = llvm.mlir.undef : vector<32xf32>
 %159 = llvm.insertelement %107, %158[%25 : i32] : vector<32xf32>
 %160 = llvm.shufflevector %159, %158
         [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] : vector<32xf32>
 %161 = llvm.fsub %157, %160  {fastmathFlags = #llvm.fastmath<fast>} : vector<32xf32>
 %162 = llvm.intr.exp(%161)  {fastmathFlags = #llvm.fastmath<fast>} : (vector<32xf32>) -> vector<32xf32>
 %163 = llvm.mlir.undef : vector<32xf32>
 %164 = llvm.insertelement %139, %163[%25 : i32] : vector<32xf32>
 %165 = llvm.shufflevector %164, %163
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] : vector<32xf32>
 %166 = llvm.fdiv %162, %165  {fastmathFlags = #llvm.fastmath<fast>} : vector<32xf32>
 %167 = llvm.getelementptr %150[%153] : (!llvm.ptr, i64) -> !llvm.ptr, f32
 %168 = llvm.getelementptr %167[%29] : (!llvm.ptr, i64) -> !llvm.ptr, f32
 llvm.store %166, %168 {alignment = 4 : i64} : vector<32xf32>, !llvm.ptr
 %169 = llvm.mul %12, %15  : i64
 %170 = llvm.mlir.zero : !llvm.ptr
```

Example 2

y= $\frac{x-E[x]}{\sqrt{(var(x)+\epsilon)}} * w + b$

```
@triton.jit
def _layer_norm_fwd_fused(
    X, Y, W, B, Mean, Rstd, stride, N, eps, BLOCK_SIZE: tl.constexpr):
    # Map the program id to the row of X and Y it should compute.
    row = tl.program_id(0)
    Y += row * stride
    X += row * stride
    # Compute mean
    mean = 0
    _mean = tl.zeros([BLOCK_SIZE], dtype=tl.float32)
    for off in range(0, N, BLOCK_SIZE):
        cols = off + tl.arange(0, BLOCK_SIZE)
        a = tl.load(X + cols, mask=cols < N, other=0.).to(tl.float32)
        _mean += a
    mean = tl.sum(_mean, axis=0) / N
    # Compute variance
    _var = tl.zeros([BLOCK_SIZE], dtype=tl.float32)
    for off in range(0, N, BLOCK_SIZE):
        cols = off + tl.arange(0, BLOCK_SIZE)
        x = tl.load(X + cols, mask=cols < N, other=0.).to(tl.float32)
        x = tl.where(cols < N, x - mean, 0.)
        _var += x * x
    var = tl.sum(_var, axis=0) / N
    rstd = 1 / tl.sqrt(var + eps)
    # Write mean / rstd
    tl.store(Mean + row, mean)
```

Example 2

$$y = \frac{x - E[x]}{\sqrt{(var(x) + \in)}} * w + b$$

```
// X - E[X]
%15 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
               ins(%10, %14 : tensor<256xf32>, tensor<256xf32>) outs(%10 : tensor<256xf32>) {
^bb0(%in: f32, %in_19: f32, %out: f32):
  %33 = arith.subf %in, %in_19 : f32
  linalg.yield %33 : f32
} -> tensor<256xf32>

// Mask: x = tl.where(block < N_COLS, x - mean, 0.)
%16 = linalg.generic {indexing_maps = [#map, #map, #map, #map], iterator_types = ["parallel"]}
     ins(%7, %15, %1 : tensor<256xi1>, tensor<256xf32>, tensor<256xf32>) outs(%15 : tensor<256xf32>) {
^bb0(%in: i1, %in_19: f32, %in_20: f32, %out: f32):
  %33 = arith.select %in, %in_19, %in_20 : f32
  linalg.yield %33 : f32
} -> tensor<256xf32>

// (X-E[X])^2
%17 = linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel"]}
                        ins(%16, %16 : tensor<256xf32>, tensor<256xf32>) outs(%16 : tensor<256xf32>) {
^bb0(%in: f32, %in_19: f32, %out: f32):
  %33 = arith.mulf %in, %in_19 : f32
  linalg.yield %33 : f32
} -> tensor<256xf32>

// E[X-E[X]]
%reduced_7 = linalg.reduce ins(%17 : tensor<256xf32>) outs(%inserted_6 : tensor<f32>) dimensions = [0]
  (%in: f32, %init: f32) {
    %33 = arith.addf %in, %init : f32
    linalg.yield %33 : f32
  }  ..
```

Fuse, Vectorized core

$$y = \frac{x - E[x]}{\sqrt{(var(x) + \in)}} * w + b$$

```
    %alloc_20 = memref.alloc() {alignment = 64 : i64} : memref<256xf32>
      %12 = scf.for %arg14 = %c0 to %c256 step %c32 iter_args(%arg15 = %alloc_20) -> (memref<256xf32>) {
        %subview_24 = memref.subview %alloc[%arg14] [32] [1] : memref<256xf32> to memref<32xf32, strided<[1], offset: ?>>
        %subview_25 = memref.subview %alloc_14[%arg14] [32] [1] : memref<256xf32> to memref<32xf32, strided<[1], offset: ?>>
        %subview_26 = memref.subview %alloc_17[%arg14] [32] [1] : memref<256xf32> to memref<32xf32, strided<[1], offset: ?>>
        %subview_27 = memref.subview %arg15[%arg14] [32] [1] : memref<256xf32> to memref<32xf32, strided<[1], offset: ?>>
        %13 = vector.transfer_read %subview_24[%c0], %cst_3 {in_bounds = [true]} : memref<32xf32, strided<[1], offset: ?>>, vector<32xf32>
        %14 = vector.transfer_read %subview_25[%c0], %cst_3 {in_bounds = [true]} : memref<32xf32, strided<[1], offset: ?>>, vector<32xf32>
        %15 = vector.transfer_read %subview_26[%c0], %cst_3 {in_bounds = [true]} : memref<32xf32, strided<[1], offset: ?>>, vector<32xf32>
        %16 = vector.broadcast %4 : f32 to vector<32xf32>
        %17 = arith.subf %13, %16 fastmath<fast> : vector<32xf32>
        %18 = vector.broadcast %arg14 : index to vector<32xindex>
        %19 = arith.addi %18, %cst_2 : vector<32xindex>
        %20 = arith.index_cast %19 : vector<32xindex> to vector<32xi32>
        %21 = arith.cmpi slt, %20, %cst_1 : vector<32xi32>
        %22 = arith.select %21, %17, %cst_0 : vector<32xi1>, vector<32xf32>
        %23 = vector.broadcast %10 : f32 to vector<32xf32>
        %24 = arith.mulf %22, %23 fastmath<fast> : vector<32xf32>
        %25 = arith.mulf %24, %14 fastmath<fast> : vector<32xf32>
        %26 = arith.addf %25, %15 fastmath<fast> : vector<32xf32>
        vector.transfer_write %26, %subview_27[%c0] {in_bounds = [true]} : vector<32xf32>, memref<32xf32, strided<[1], offset: ?>>
        %subview_28 = memref.subview %arg15[%arg14] [32] [1] : memref<256xf32> to memref<32xf32, strided<[1], offset: ?>>
        memref.copy %subview_27, %subview_28 : memref<32xf32, strided<[1], offset: ?>> to memref<32xf32, strided<[1], offset: ?>>
        scf.yield %arg15 : memref<256xf32>
      }
```

Discussions

- Triton – efficient GPU / CPU? Kernel writing
- Triton-Linalg versus direct approach
- Adapting Triton for Hexagon Target (multi-threading, vectorization, ..)
- Other options…

Thank you

Qualcomm

Follow us on:

For more information, visit us at:

qualcomm.com & qualcomm.com/blog