Qualcomm

2025 LLVM Developers' Meeting

Oct 28, 2025

Accelerating ML on Hexagon™ A Glimpse into Qualcomm's MLIR-based Compiler

Speakers: Franck Slama, Muthu Baskaran

Qualcomm Technologies, Inc.

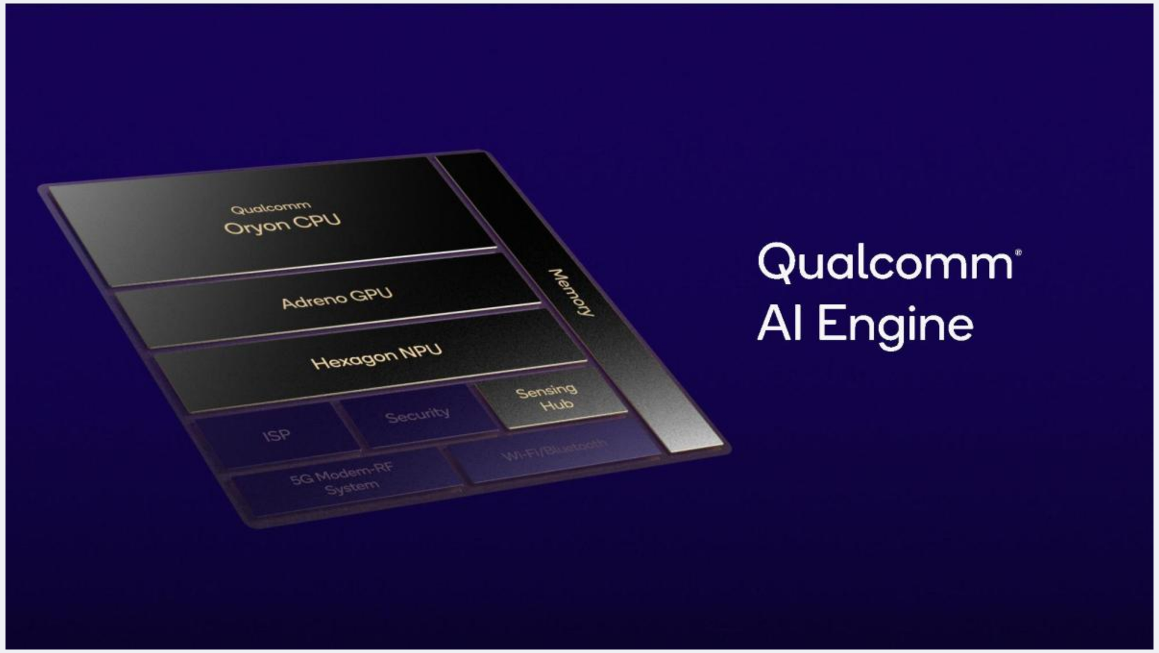
Authors: Mohammed Javed Absar, Ankit Aggarwal, Iulian Brumar, Snigdha Suresh Dalvi, Shalini Jain, Venkat Rasagna Reddy Komatireddy, Mitesh Kothari, Samarth Narang, Tasmia Rahman, Arun Rangasamy, Abhikrant Sharma, Franck Slama, Jyotsna Verma, Zachary Zipper, Muthu Baskaran

MLIR Hexagon Compiler

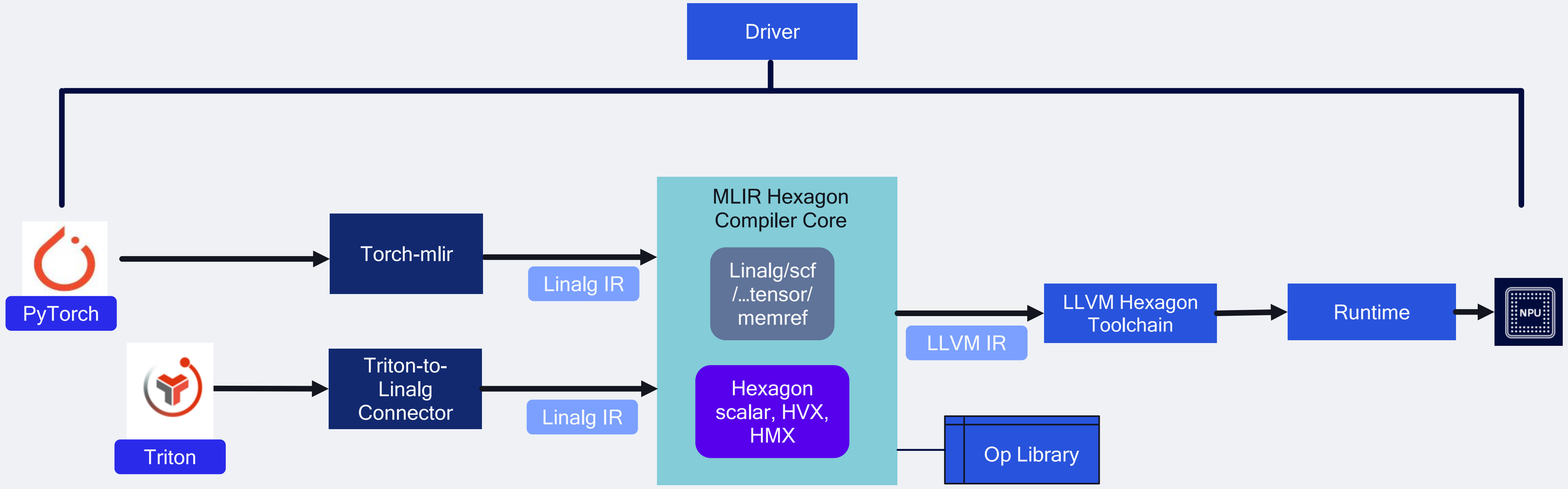
- MLIR-based AI compiler targeting Hexagon NPUs
  - Leverages advancements from both open-source and Qualcomm legacy AI compilers
- Major workstream areas
  - **MLIR Hexagon compiler core**: open-source MLIR infrastructure + Hexagon NPU pass pipeline
  - **PyTorch model compilation** and execution on Hexagon NPU targets
  - **Triton kernel compilation**
- Key results
  - **Functional MLIR Hexagon pipeline** w/ **developer-friendly Python driver**
  - **Successful demonstration of LLM models** w/ tiling, fusion, vectorization, and memory optimizations
  - **Successful mapping of simple to non-trivial Triton kernels** (flash attention, softmax, argmax, matmul, and many more)
  - Initial performance of Triton kernels mapped via MLIR compiler **up to 80% of hand-written kernels**

Target Platform

- Hexagon NPUs
  - ❑ 4-way multi-threaded VLIW
  - ❑ Vector registers and dedicated memory
  - ❑ Hexagon Vector eXtensions (HVX)
    - ○ Long vectors (128B) that can process 128 int8 elements or 64 fp16 elements or 32 fp32 elements
  - ❑ Hexagon Matrix eXtensions (HMX)

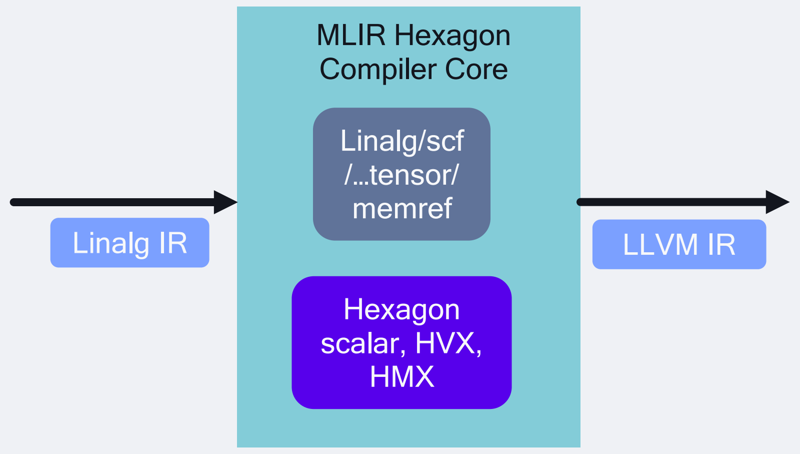


PyTorch and Triton Workflow via Hexagon NPU Backend



- Approach: Leverage upstream Triton and MLIR developments in addition to building downstream target-specific optimizations

Hexagon Passes for Lowering and Optimizations



Passes	Description
MLIR core built-in passes	Named to generic, add-fast-math, vector-to-scf, scf-to-cf, vector-to-llvm, math-to-llvm, canonicalize, cse, dce, …
Fusion	Fuse linalg ops that have producer consumer relation and perform multi-user fusion with re-computation
Tiling	Tiling for scratchpad memory, vectorization, and extracting parallelism
Vectorization	Generate vectorized HVX code
Vectorized math functions	Replace llvm.math.* intrinsics with hand-optimized vectorized math library
Memory optimization passes	Scratchpad management; elimination of redundant creation of memory buffers and associated memory copies
Mapping to tensor units	Lowering to transform data layout and compute for calling tensor unit microkernels

Illustrative Example - 1

```
%62 = linalg.generic {indexing_maps = [affine_map<(d0,
d1, d2, d3) -> (0, d1, d3)>, affine_map<(d0, d1, d2, d3)
-> (d3)>, affine_map<(d0, d1, d2, d3) -> (d3, d2)>,
affine_map<(d0, d1, d2, d3) -> (d0, d1, d2)>],
iterator_types = ["parallel", "parallel", "parallel",
"reduction"]}
    ins(%X, %cst_35, %cst_36 : tensor<?x7x3072xf32>,
tensor<3072xf32>, tensor<3072x768xf32>)
    outs(%expanded_193 : tensor<1x7x768xf32>) {
  ^bb0(%in: f32, %in_374: f32, %in_375: f32, %out: f32):
    %269 = arith.addf %in, %in_374 : f32
    %270 = math.powf %269, %cst_11 : f32
    %271 = arith.truncf %cst_4 : f64 to f32
    %272 = arith.mulf %270, %271 : f32
    %273 = arith.addf %269, %272 : f32
    %274 = arith.truncf %cst_3 : f64 to f32
    %275 = arith.mulf %273, %274 : f32
    %276 = math.tanh %275 : f32
    %277 = arith.addf %276, %cst_12 : f32
    %278 = arith.mulf %269, %cst_10 : f32
    %279 = arith.mulf %278, %277 : f32
    %280 = arith.mulf %279, %in_375 : f32
    %281 = arith.addf %out, %280 : f32
    linalg.yield %281 : f32
  } -> tensor<?x7x768xf32>
```

Fused computation (fusion done on 'linalg graph')

```
scf.for %i= %c0 to %cI step %istep{
  scf.for %j= %c0 to %cJ step %jstep {
    %subview = memref.subview %X[0, %i, %j]
... : memref<1x7x3072xf32> to memref<3072xf32,
strided<[1], offset: ?>>
    %s_memory = hexagonmem.alloc() :
memref<3072xf32>
    hexagonmem.copy %subview to %s_memory :
memref<3072xf32, strided<[1], offset: ?>> to
memref<3072xf32>

    scf.for %k= %c0 to %xK step %kstep {
     ...
      %153 = arith.addf %149, %150
fastmath<fast> : vector<32xf32>
      %154 = math.powf %153, %cst_12
fastmath<fast> : vector<32xf32>
     ...
    }
    hexagonmem.copy %s_memory to %subview  :
memref<3072xf32> to memref<3072xf32,
strided<[1], offset: ?>>
    hexagonmem.dealloc %s_memory :
memref<3072xf32>
     }
   }
```

Tiling, Vectorization, Scratchpad memory alloc/dealloc and data copy

```
 scf.for %i= %c0 to %cI step %istep{
      scf.for %j= %c0 to %cJ step %jstep {
         %subview = memref.subview %X[0, %i,
%j] ... : memref<1x7x3072xf32> to
memref<3072xf32, strided<[1], offset: ?>>
         %s_memory = hexagonmem.alloc() :
memref<3072xf32>
         memref.dma_start %subview[..] %
s_memory [...]
         memref.dma_wait …
         scf.for %k= %c0 to %xK step %kstep {
...
           %153 = arith.addf %149, %150
fastmath<fast> : vector<32xf32>
           %154 = math.powf %153, %cst_12
fastmath<fast> : vector<32xf32>
            ...
          }
        memref.dma_start %s_memory[…]
%subview[..]
        memref.dma_wait …
        hexagonmem.dealloc %s_memory :
memref<3072xf32>
      }
    }
```

Tiling, Vectorization, Scratchpad memory alloc/dealloc and data copy **w/ DMA**

Illustrative Example - 2

Flash Attention core in Triton

```
@triton.jit
def
_attn_fwd_inner(acc,l_i,m_i,q,K_block_ptr,V_block_ptr,start_m,qk_scale,
    BLOCK_M: tl.constexpr,BLOCK_DMODEL: tl.constexpr,BLOCK_N:
tl.constexpr,
    STAGE: tl.constexpr,offs_m: tl.constexpr,offs_n: tl.constexpr,
    N_CTX: tl.constexpr):
    lo, hi = 0, N_CTX
    K_block_ptr = tl.advance(K_block_ptr, (lo, 0))
    V_block_ptr = tl.advance(V_block_ptr, (lo, 0))
    # loop over k, v and update accumulator
    for start_n in range(lo, hi, BLOCK_N):
        start_n = tl.multiple_of(start_n, BLOCK_N)
        # -- compute qk ----
        k = tl.load(K_block_ptr)
        qk = tl.dot(q, tl.trans(k))
        m_ij = tl.maximum(m_i, tl.max(qk, 1) * qk_scale)
        qk = qk * qk_scale - m_ij[:, None]
        p = tl.math.exp2(qk)
        l_ij = tl.sum(p, 1)
        # -- update m_i and l_i
        alpha = tl.math.exp2(m_i - m_ij)
        l_i = l_i * alpha + l_ij
        # -- update output accumulator --
        acc = acc * alpha[:, None]
        # update acc
        v = tl.load(V_block_ptr)
        # p = p.to(tl.float32)
        acc = tl.dot(p, v, acc)
        # update m_i and l_i
        m_i = m_ij
        V_block_ptr = tl.advance(V_block_ptr, (BLOCK_N, 0))
        K_block_ptr = tl.advance(K_block_ptr, (BLOCK_N, 0))
    return acc, l_i, m_i
```

Parallelism, fusion

```
%42 = linalg.generic {indexing_maps = [#map3, #map6, #map3],
iterator_types = ["parallel", "parallel"]} ins(%37,
%expanded_8 : tensor<256x64xf32>, tensor<256x1xf32>) outs(%3
: tensor<256x64xf32>) {
      ^bb0(%in: f32, %in_11: f32, %out: f32):
        %53 = arith.mulf %in, %29 : f32
        %54 = arith.subf %53, %in_11 : f32
        %55 = math.exp2 %54 : f32
        linalg.yield %55 : f32
      } -> tensor<256x64xf32>
```

Vectorization

```
 scf.for %arg38 = %c0 to %c256 step %c1 {
        scf.for %arg39 = %c0 to %c64 step %c32 {
          %subview = memref.subview %alloc_17[%arg38, %arg39] [1, 32] [1, 1] :
memref<256x64xf32> to memref<32xf32, strided<[1], offset: ?>>
          %subview_29 = memref.subview %expand_shape_20[%arg38, 0] [1, 1] [1, 1]
: memref<256x1xf32> to memref<f32, strided<[], offset: ?>>
          %subview_30 = memref.subview %alloc_7[%arg38, %arg39] [1, 32] [1, 1] :
memref<256x64xf32> to memref<32xf32, strided<[1], offset: ?>>
          %32 = vector.transfer_read %subview[%c0], %cst_4 {in_bounds = [true]} :
memref<32xf32, strided<[1], offset: ?>>, vector<32xf32>
          %33 = memref.load %subview_29[] : memref<f32, strided<[], offset: ?>>
          %34 = vector.broadcast %33 : f32 to vector<32xf32>
          %35 = vector.broadcast %24 : f32 to vector<32xf32>
          %36 = arith.mulf %32, %35 fastmath<fast> : vector<32xf32>
          %37 = arith.subf %36, %34 fastmath<fast> : vector<32xf32>
          %38 = math.exp2 %37 fastmath<fast> : vector<32xf32>
          vector.transfer_write %38, %subview_30[%c0] {in_bounds = [true]} :
vector<32xf32>, memref<32xf32, strided<[1], offset: ?>>
        }
      }
```

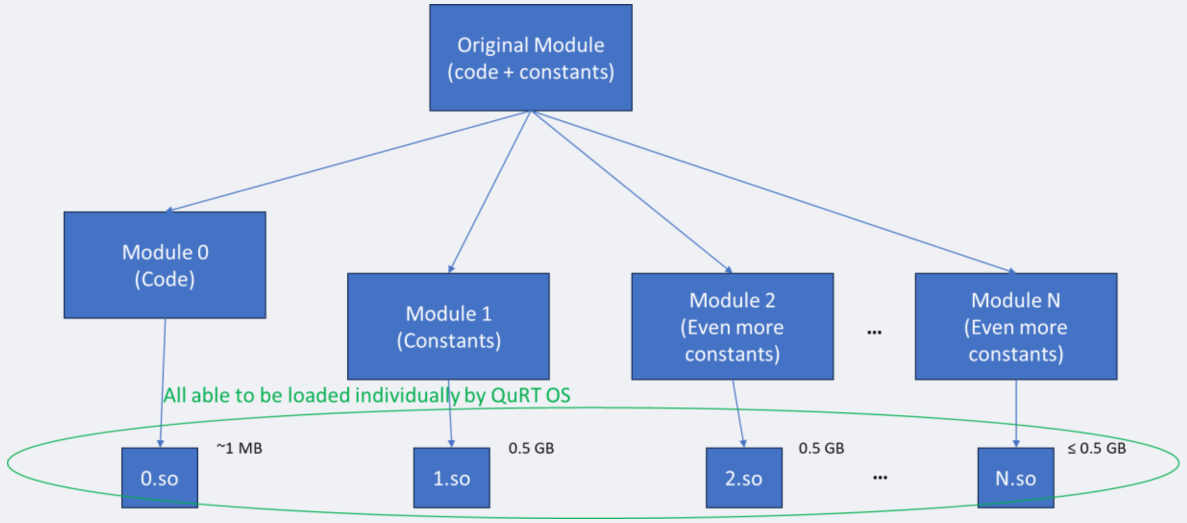
Call to hand-optimized library

```
  %525 = llvm.intr.vector.extract %523[0] : vector<32xf32> from vector<64xf32>
  %526 = llvm.call @hexagon.exp.vsf.128B(%525) : (vector<32xf32>) -> vector<32xf32>
  %527 = llvm.intr.vector.insert %526, %524[0] : vector<32xf32> into vector<64xf32>
```

Scalability: An MLIR pass to lower the constants separately

- For large models, we want to split the module into various parts, loaded independently by the OS
  - Since 99% of the memory footprint is due to constant parameters (weights, biases, factors) → need to lower separately the constants

- Module splitting implemented at the MLIR level
  - New MLIR pass that separates out constants:
    1 module in, N module out
  - Creation of new MLIR modules (constants-only) on the fly
  - In the main module (for code):
    Replace the *arith.constant* by an extern


Original Module (code + constants)
Module 0 (Code)
Module 1 (Constants)
Module 2 (Even more constants)
...
Module N (Even more constants)
All able to be loaded individually by QuRT OS
~1 MB
0.5 GB
0.5 GB
≤ 0.5 GB
0.so
1.so
2.so
...
N.so

- Compiling (and linking) all these modules appropriately (the principal module needs to be linked against all the constants-only modules, as it uses their constants)

- Result: Ability to lower the constants separately, helps to run large models on device, on Qualcomm Hexagon NPU

Conclusion

- MLIR-based AI compiler targeting Hexagon NPUs
  - Leverages advancements from both open-source and Qualcomm legacy AI compilers
- Demonstrated mapping of PyTorch models and Triton kernels via the compiler on to Hexagon NPUs
  - Can generate code for a variety of ML models. Currently improving the efficiency of the code being produced.
- Current capabilities include support for single NPU configurations
  - Next: Support for multi-NPU configurations
- Planning to release/open-source our compiler in December 2025

Thank you

Follow us on:

For more information, visit us at qualcomm.com & qualcomm.com/blog