MTIA: FIRST GENERATION SILICON TARGETING META'S RECOMMENDATION SYSTEMS

ACCELERATOR ARCHITECTURE

- It is organized as an array of processing elements (PEs) connected on a grid.
- The grid is connected to a set of on-chip memory blocks and off-chip memory controllers through crossbars on each side.
- A separate control subsystem with dedicated processors and peripherals runs the system's control software.
- The host interface unit which contains a PCIe interface, associated DMA engines, and a secure boot processor also sits alongside this control subsystem.

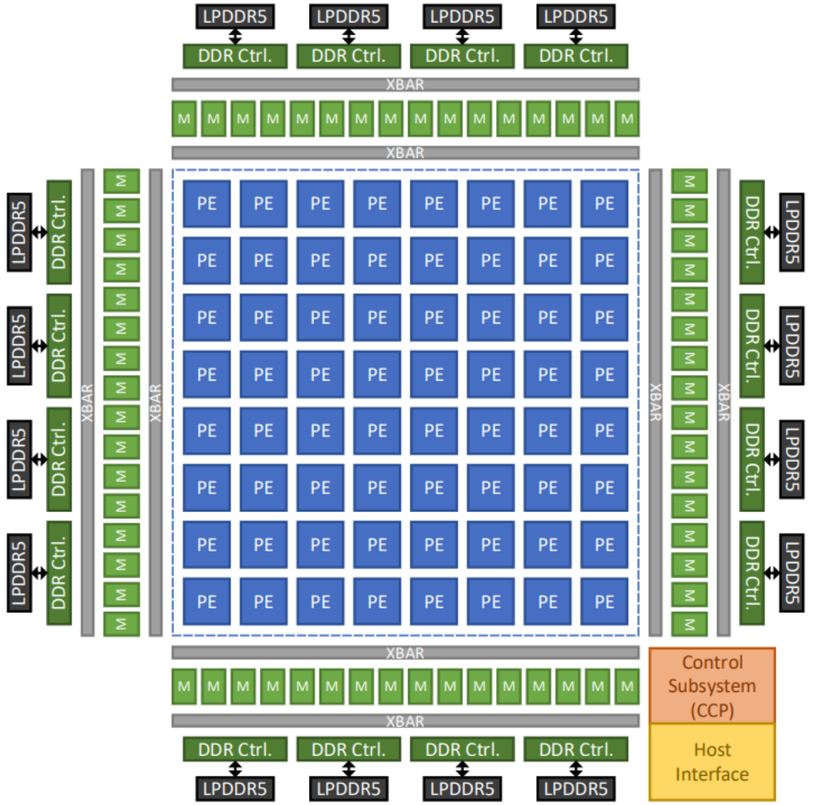

Figure 3: High-level architecture of the accelerator

PROCESSING ELEMENT ARCHITECTURE

Fixed Function Units

1. Memory Layout Unit (MLU)
2. Dot-Product Engine (DPE)
3. Reduction Engine (RE)
4. SIMD Engine (SE)
5. Fabric Interface (FI)
6. command Processor (CP)

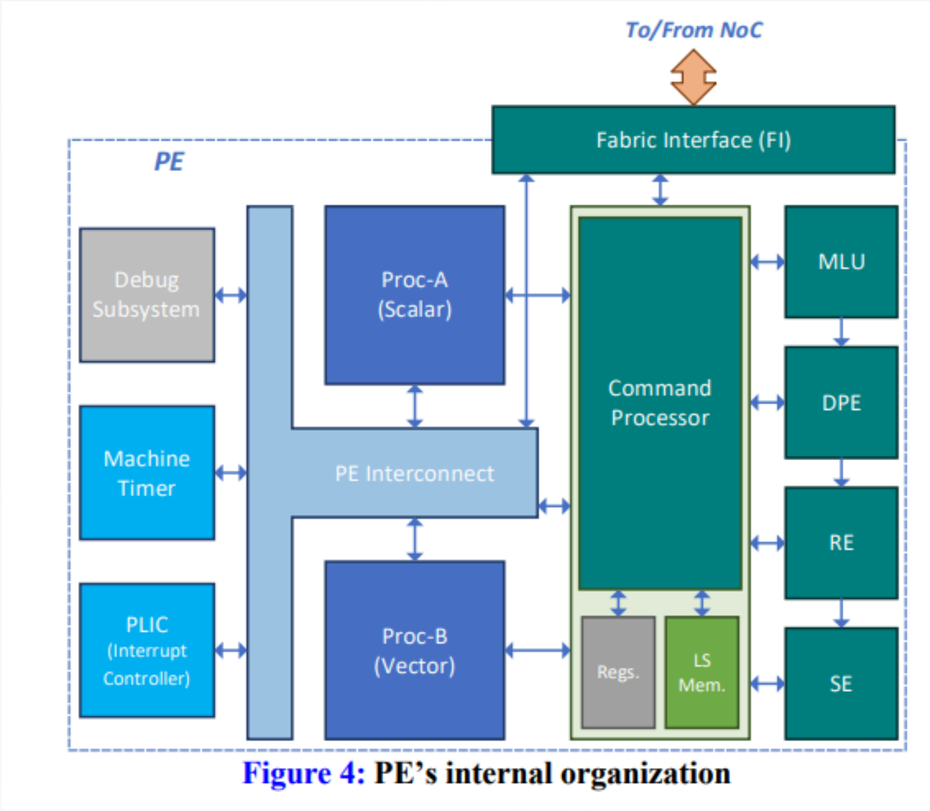

Figure 4: PE's internal organization

PROCESSING ELEMENT ARCHITECTURE

Local Memory Architecture:

- 128KB of local memory, accessible by both processors and fixed function units.
- Local memories are mapped to the system's address space, enabling access via regular load/store instructions.

Circular Buffers (CBs):

- Each CB has an ID, size (depth), and starting address.
- Read and write pointers implement a hardware FIFO mechanism.
- Read operations start from the read pointer; write operations start from the write pointer.
- Fixed function units use CB IDs as operands, ensuring necessary resources before starting operations.

Pointer Management:

- DMA operations adjust read/write pointers automatically.
- Custom instructions allow manual pointer adjustments, enabling data reuse before marking as consumed.

PROCESSING ELEMENT ARCHITECTURE

On-Chip Memory:

- 128MB of on-chip SRAM, organized as slices around the grid.
- Can function as addressable scratchpad memory or shared memory-side cache.
- Four LPDDR5 controllers provide a total of 176 GB/s theoretical off-chip bandwidth.
- Supports up to 128GB off-chip memory capacity.

Memory Address Distribution:

- Addresses distributed across controllers and on-chip SRAM slices.
- On-chip SRAM configured as cache associates slices with memory controllers.

On-Chip Network:

- Based on AXI interconnect with enhancements.
- Consists of separate networks for memory and register accesses.
- Memory access network features multicast, coalescing requests from multiple PEs into one for efficient data retrieval.

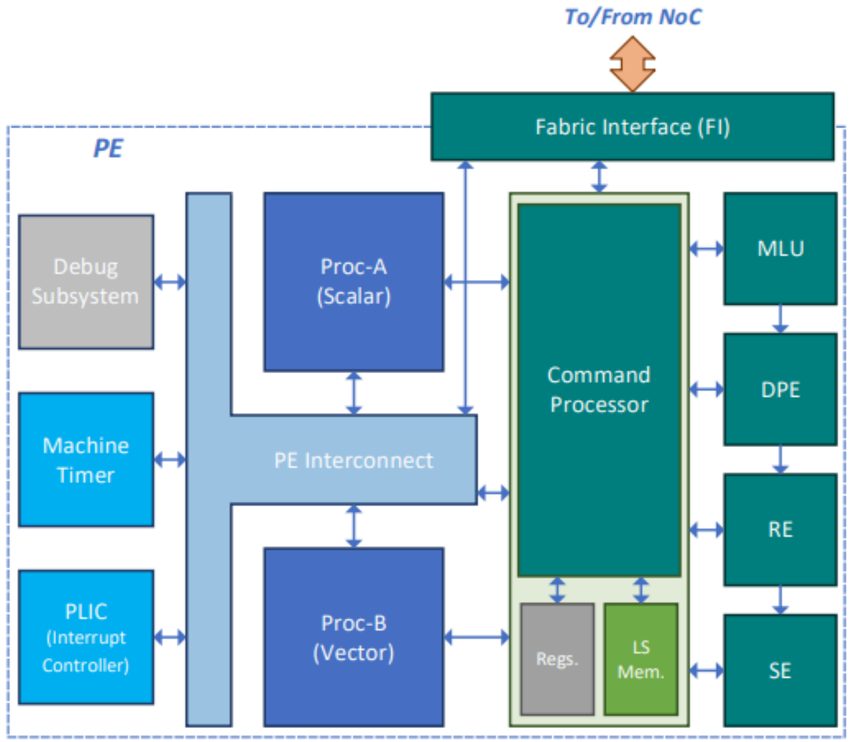

Figure 4: PE's internal organization

MAPPING FC LAYER TO PE'S

- Data Preparation: The data (matrices A and B) are first fetched from the main memory into local buffers within the PE. This is done using DMA (Direct Memory Access) operations, which are efficient ways to move large amounts of data without involving the processor heavily.
- Multicasting: Rows of matrix A are multicast-read by PEs in columns 0 and 2, and columns 1 and 3. This means that a single row of matrix A is read and then sent to multiple PEs that need it. Similarly, columns of the transposed matrix B are multicast-read by all PEs in each column.
- Matrix Multiplication: Each PE performs part of the matrix multiplication. For example, PE[1,0] works on a specific block of the matrices defined by indices for m, k, and n. The calculations are distributed so that each PE handles a chunk of the work.
- Accumulation and Result Storage: After the multiplication, the results are stored in local memory or passed to the next PE for further accumulation if needed. This process reduces the load on the memory and allows efficient data processing within the grid of PEs.

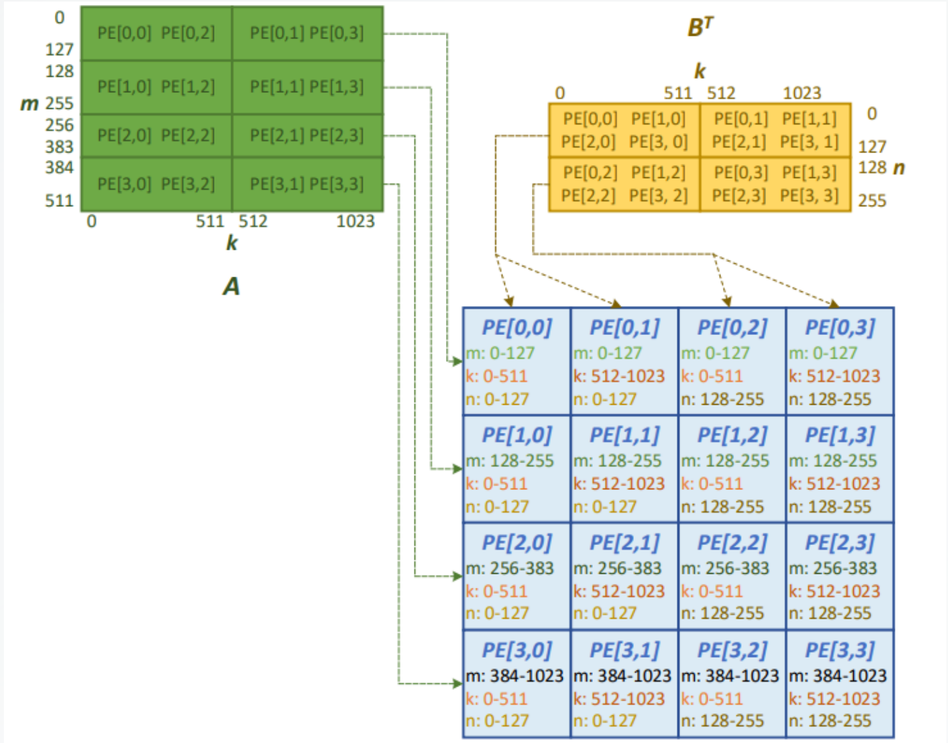

Figure 7: Mapping an FC operator to a sub-grid

CORE OPERATIONS

- Core0: Issues DMA operations to move data from the main memory into local buffers (CB_A and CB_B) within the PE. It essentially manages the data flow into the PE.
- Core1: Handles the actual computation (dot-product operations) using the data in the local buffers. It performs the matrix multiplications and stores the results locally or passes them for further accumulation.
- The cores synchronize at the beginning of the operation for initialization tasks. Post-initialization, there is no explicit per-iteration synchronization. Hardware handles synchronization by stalling the consumer core (Core1) if it attempts to access a CB without sufficient data from Core0, ensuring smooth operation progress
- These details elucidate how the data pipeline into the DPE is managed, how bandwidth constraints are alleviated, and how matrix dimensions and operations are distributed and synchronized across the PEs and their cores.

```
#--------------------------------Core0-------------------------------------
work = GetWorkForMyPE(...)
INIT CB_A, CB_B and CB_C                          # Setup circular buffers
multicast_A, multicast_B = JoinMulticastGroup(...)
Sync(...)                                         # Synchronize with others
read_B = true
for m in range(work.m.begin, work.m.end, 64):  # For every row of "A"...
 read_A = true
 for n in range(work.n.begin, work.n.end, 64): # ...read entire "B"
  for k in range(work.k.begin, work.k.end, 32):
   if read_A:
    DMA GetAddr(A, (m, k)), size=(64,32), CB_A, multicast_A
   if read_B:
    DMA GetAddr(B, (n, k)), size=(64,32), CB_B, multicast_B
  read_A = false
 read_B = false
#--------------------------------Core1-------------------------------------
work = GetWorkForMyPE(...)
Sync(...)                                         # Synchronize with others
for m in range(work.m.begin, work.m.end, 64): # For every two chunks of "A"
 cb_offset_B = 0
 for n in range(work.n.begin, work.n.end, 64):# Multiply two chunks of "B"
  cb_offset_A = 0
  INIT RE acc with 0                              # Initialize accumulators
  for k in range(work.k.begin, work.k.end, 32):
   MML acc=0,size=(32,32,32),CB_B,CB_A,cb_offset_B      ,cb_offset_A
   MML acc=1,size=(32,32,32),CB_B,CB_A,cb_offset_B      ,cb_offset_A+32*32
   MML acc=2,size=(32,32,32),CB_B,CB_A,cb_offset_B+32*32,cb_offset_A
   MML acc=3,size=(32,32,32),CB_B,CB_A,cb_offset_B+32*32,cb_offset_A+32*32
   if ((m + 64) >= work.m.end):                   # If last Iteration...
    POP CB_B, size=2*32*32                        # ...mark "B" data as consumed
   else:                                          # Otherwise...
    cb_offset_B += 2*32*32                        # ...proceed to the next chunk
   if ((n + 64) >= work.n.end):                   # If last Iteration...
    POP CB_A, size=2*32*32                        # ...mark "A" data as consumed
   else:                                          # Otherwise...
    cb_offset_A += 2*32*32                        # ...proceed to the next chunk
  REDUCE destination = neighbor PE or CB_C, size=(64,64))# Send to next PE
  if IsLastPEInReduction(...):                    # If last PE in sequence
   DMA PutAddr(C, (n, m)), size=(64, 64), CB_C # Write result to memory
```

Figure 8: Pseudocode for the FC operator running in PE

SOFTWARE STACK

- **Components**
- ML Serving Platform:
- PyTorch Runtime:
- Compilers:
- PyTorch FX-based Compiler: This compiler applies transformations and optimizations to the PyTorch graph, converts it into LLVM IR, and performs graph optimizations to leverage the PE grid and MTIA's memory subsystem.
- DSL-based Compiler (KNYFE): Used for developing ML kernels, this compiler translates high-level ML kernel descriptions into optimized C++ code, implementing the operator using hardware-specific APIs.
- LLVM-based Compiler Toolchain: Converts LLVM IR into executable code for the device, focusing on low-level optimizations like register allocation and code generation.
- Library of ML Kernels:

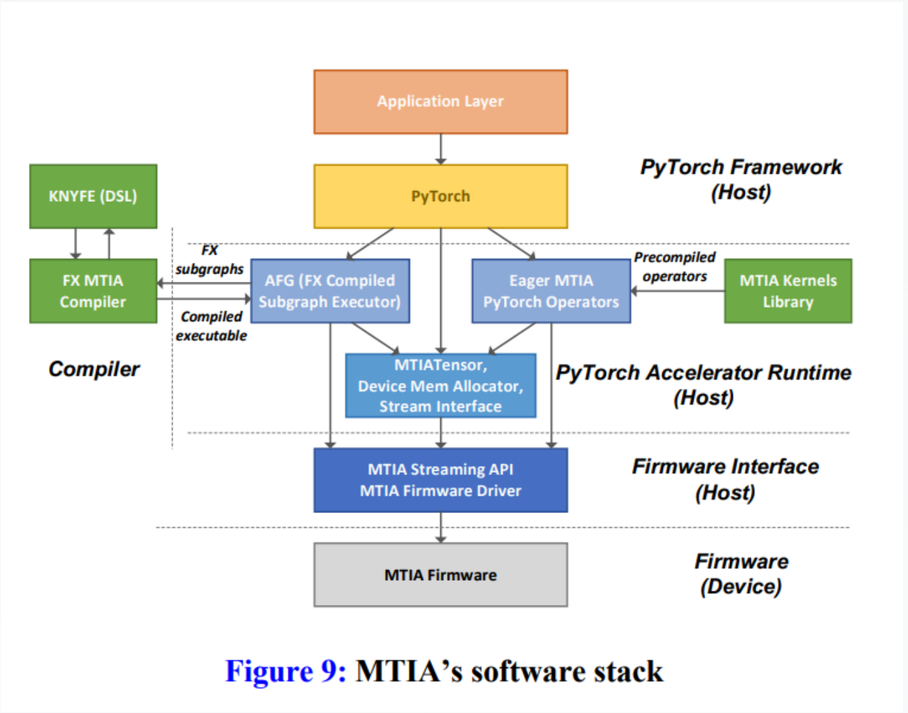

Figure 9: MTIA's software stack