The NPU Hardware and Software Stack

NexNPU Tutorial @ ISCA'26

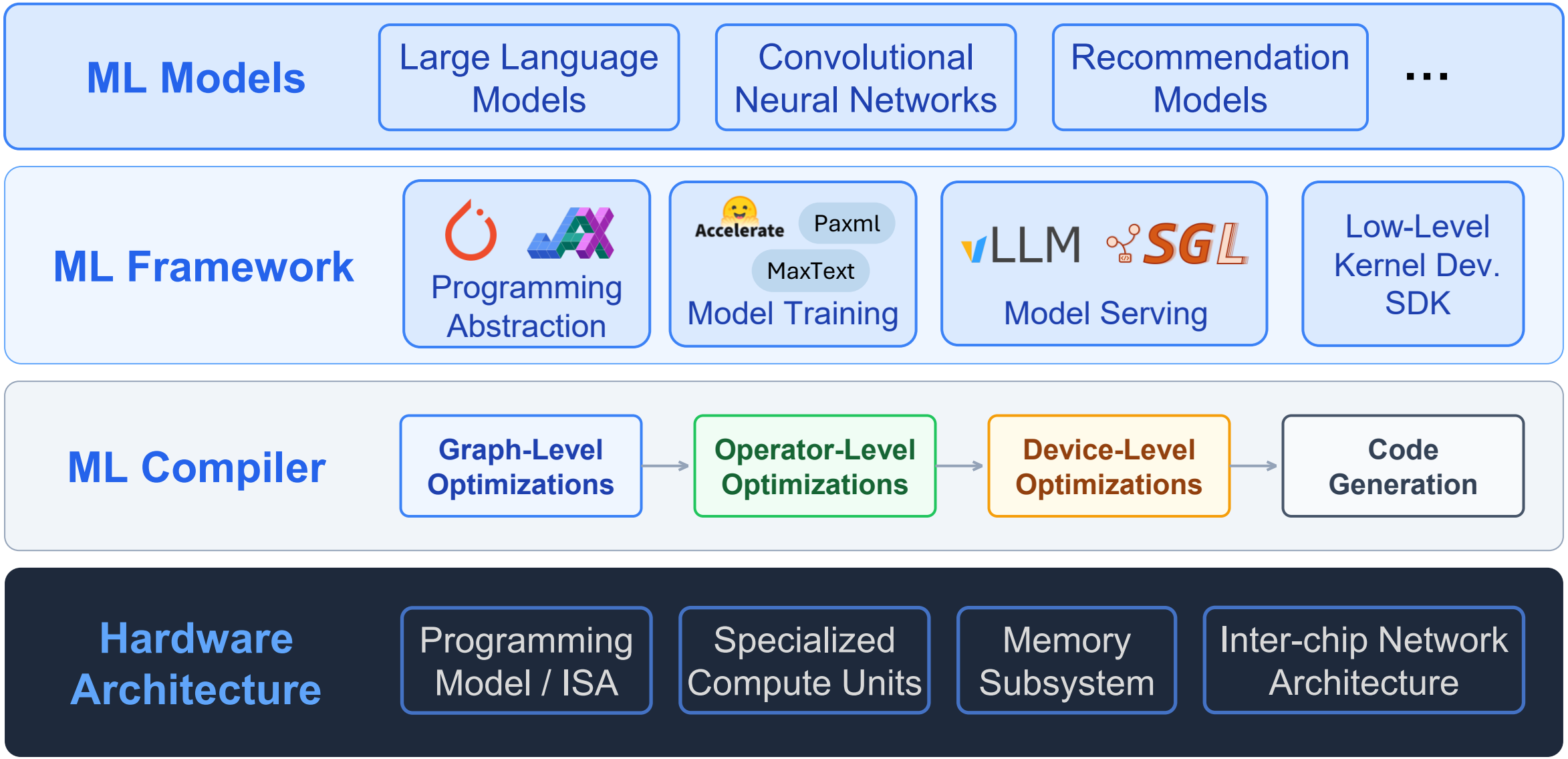
Yuqi Xue

Ph.D. Student, Electrical & Computer Engineering,

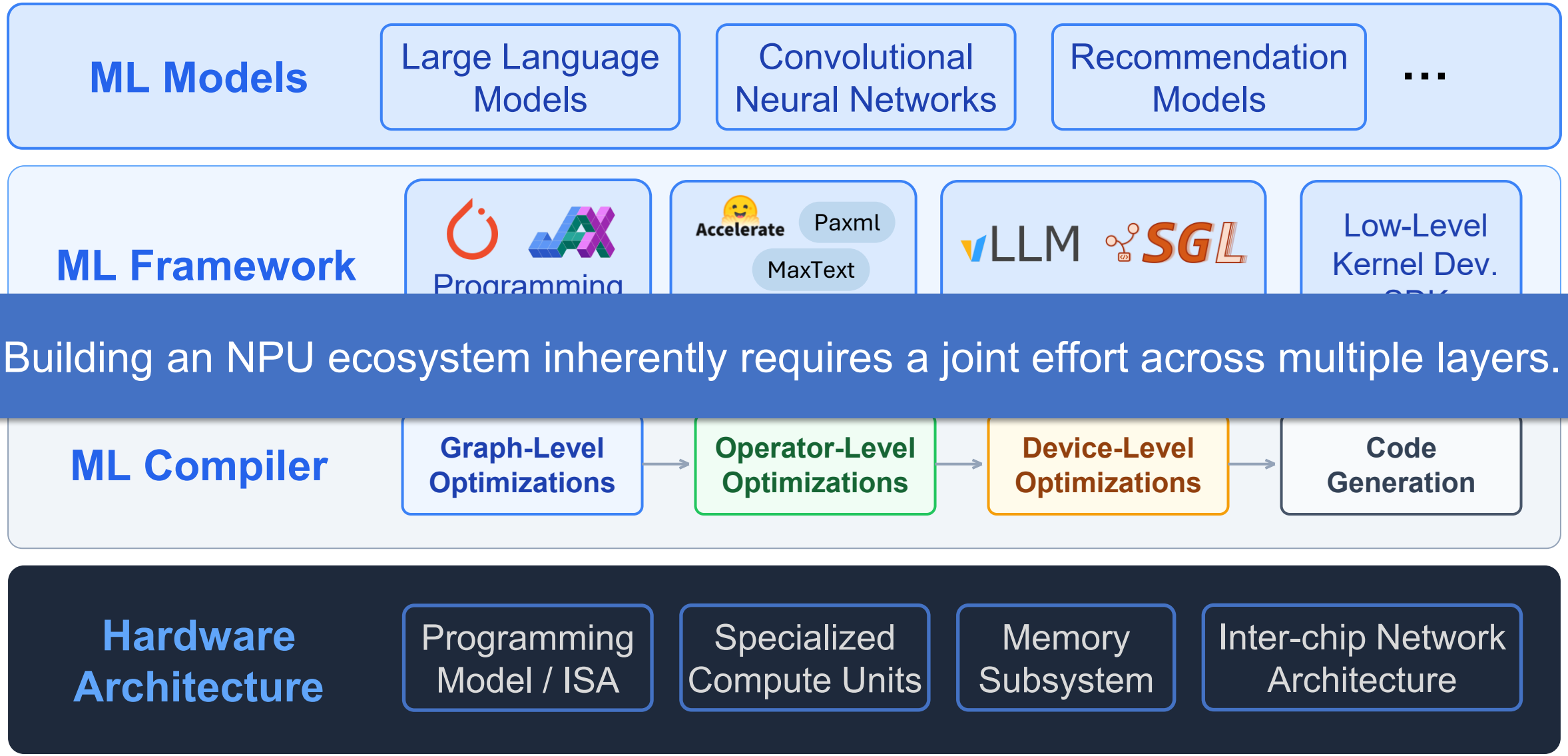
Systems Platform & Intelligence Lab @ University of Illinois Urbana-Champaign

yuqixue2@illinois.edu

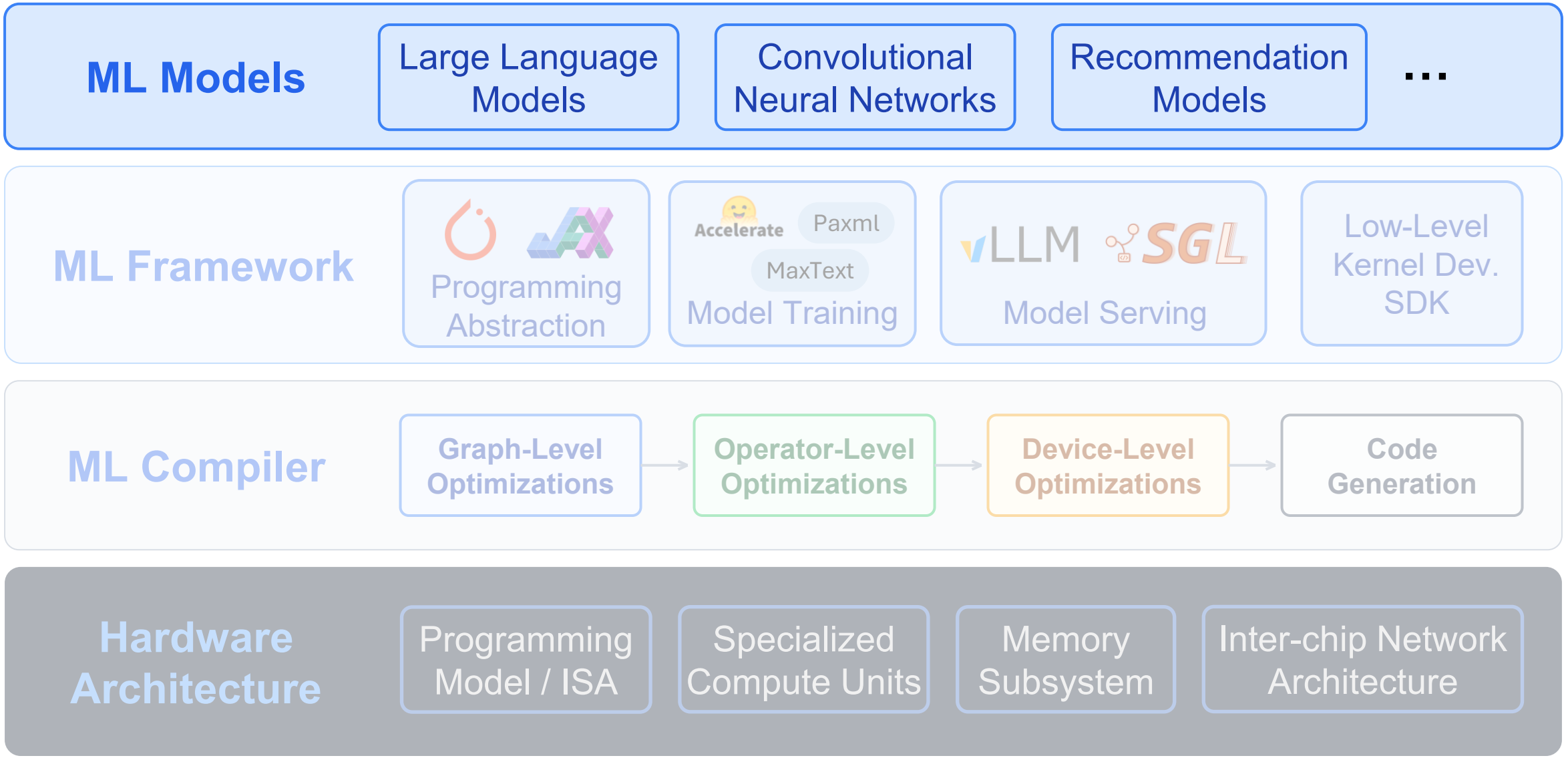
From ML Model To Hardware Execution: Overview of Modern AI Stack



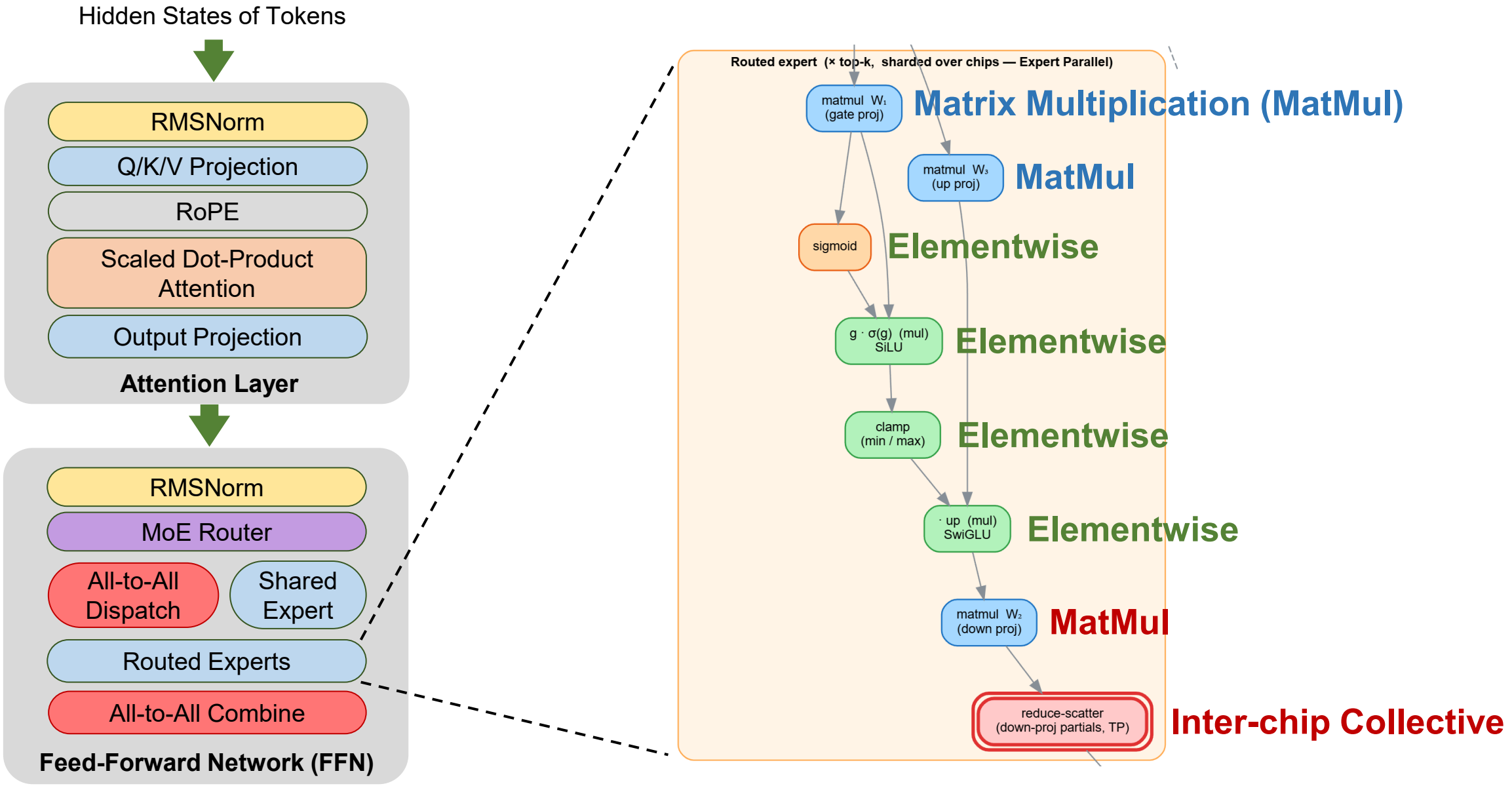
From ML Model To Hardware Execution: Overview of Modern AI Stack



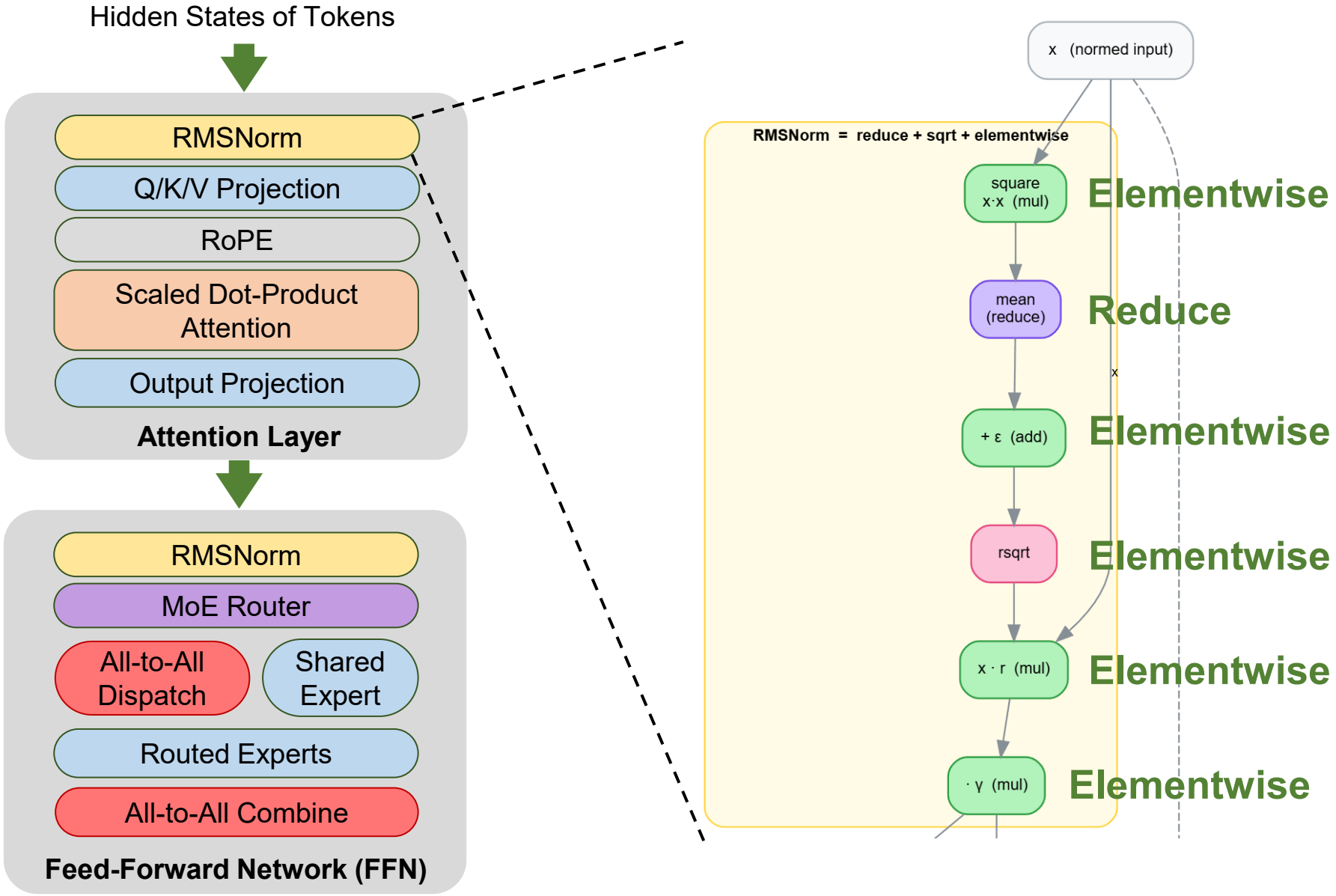
From ML Model To Hardware Execution: Overview of Modern AI Stack



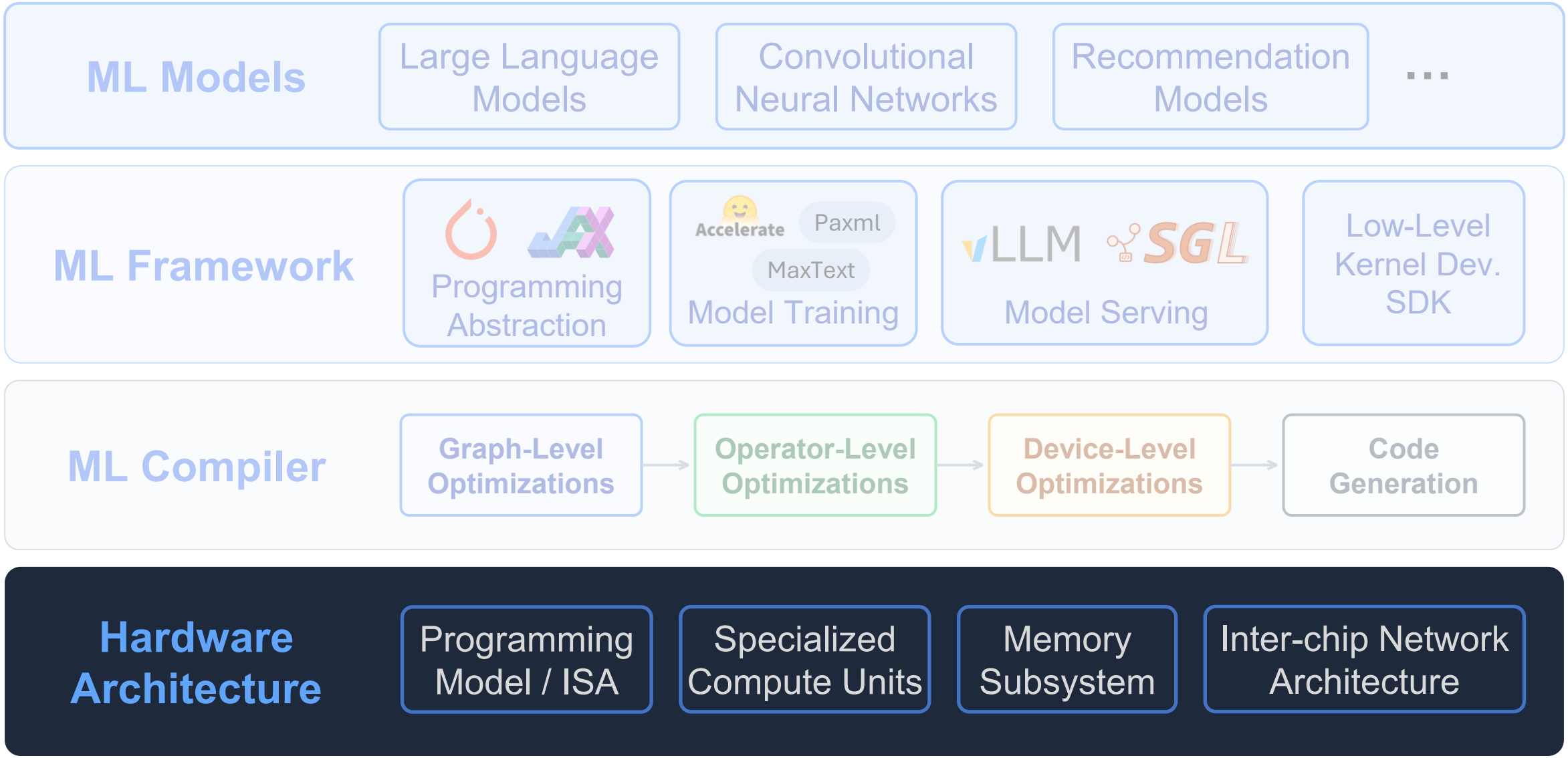
ML Models: The Computational Graph & Diverse Tensor Operators



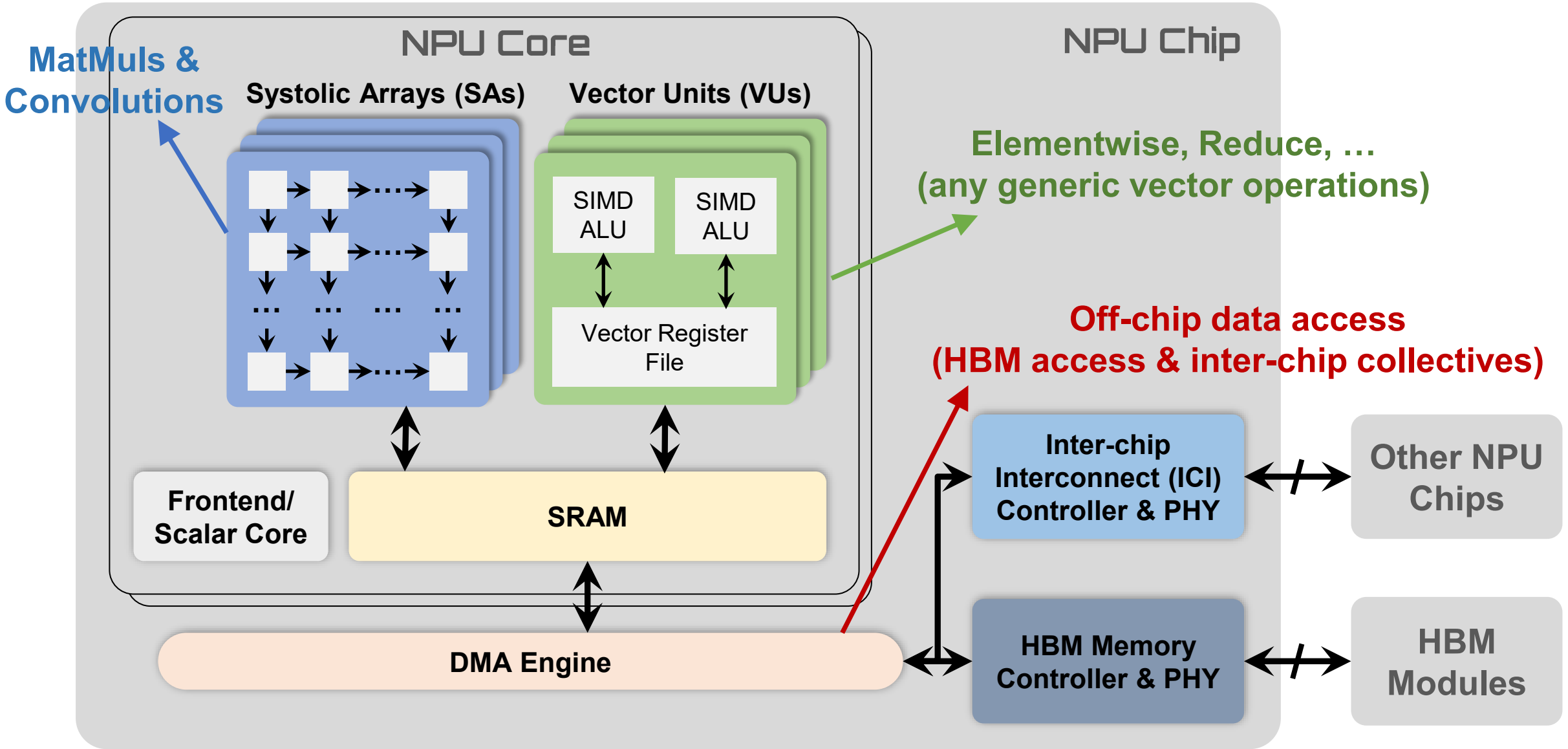
ML Models: The Computational Graph & Diverse Tensor Operators



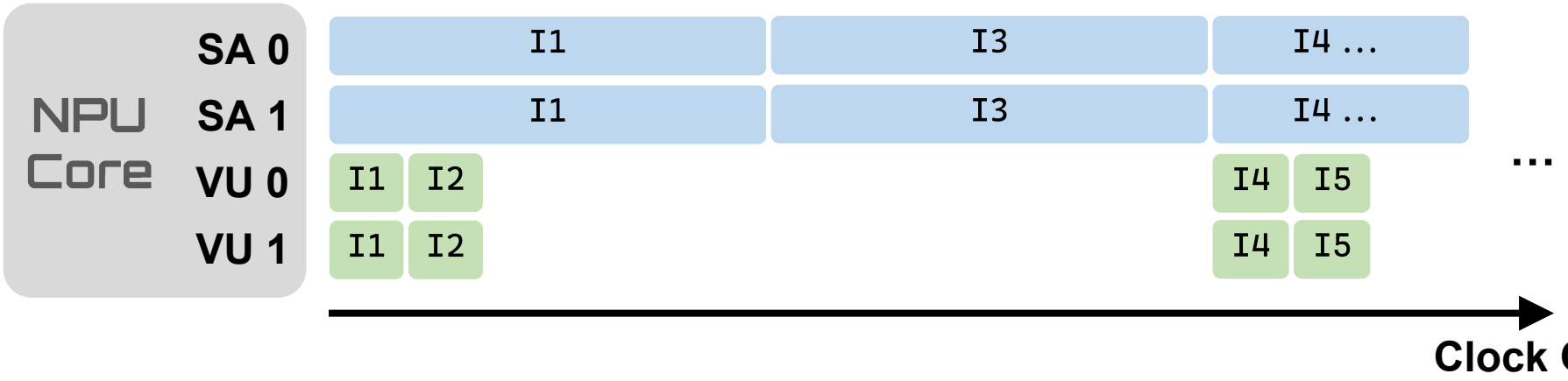
Overview of Modern AI Stack



NPU Chip Architecture Overview



NPU Programming Model: Single-Threaded Core, VLIW ISA



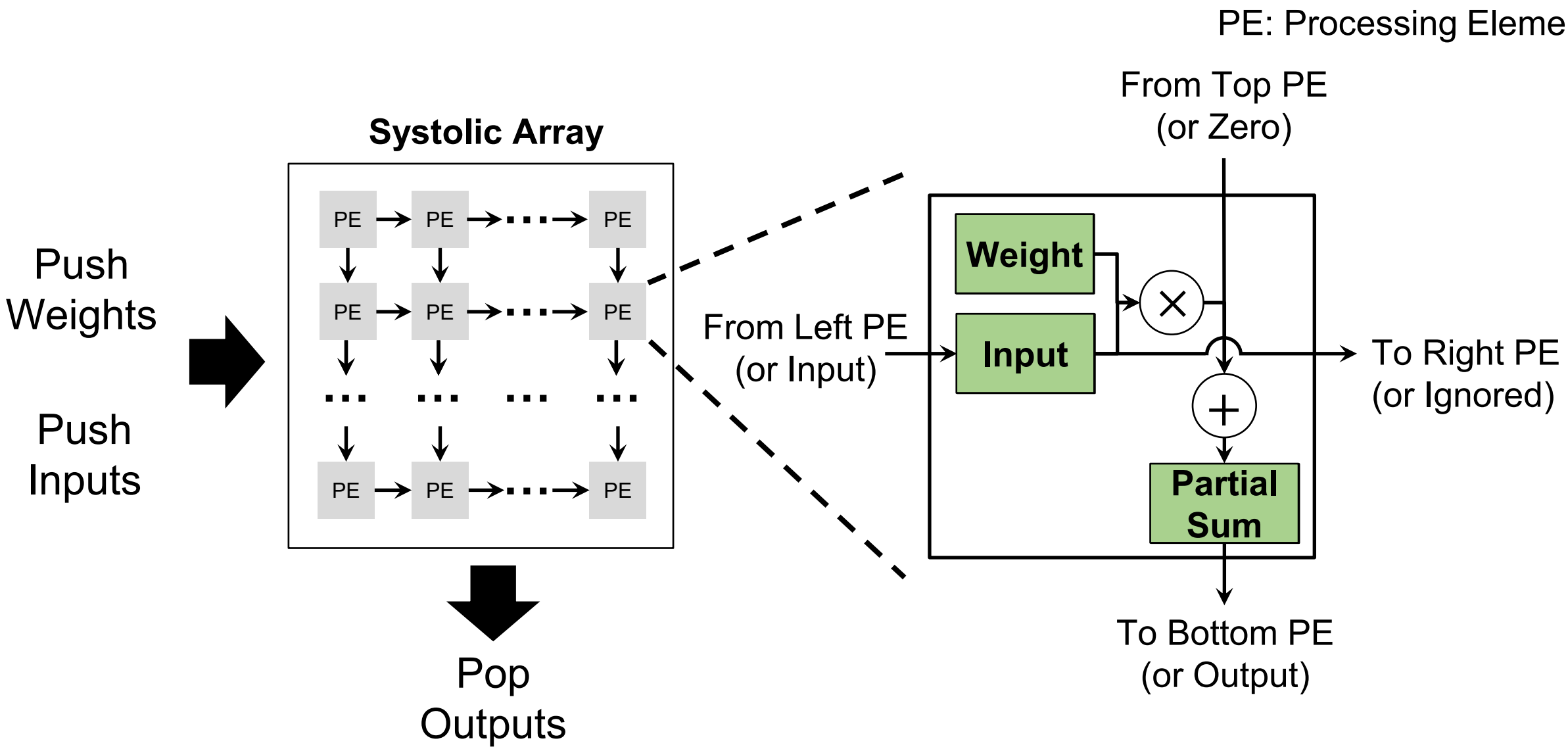

```
I1: { vpop.sa0;  vpop.sa1;  vadd.vu0;  vadd.vu1; }
I2: {                       vadd.vu0;  vadd.vu1; }
I3: { vpop.sa0;  vpop.sa1;                       }
I4: { vpop.sa0;  vpop.sa1;  vadd.vu0;  vadd.vu1; }
I5: {                       vadd.vu0;  vadd.vu1; }
I6: {                  ...                       }
```

A Simplified NPU VLIW Code Snippet

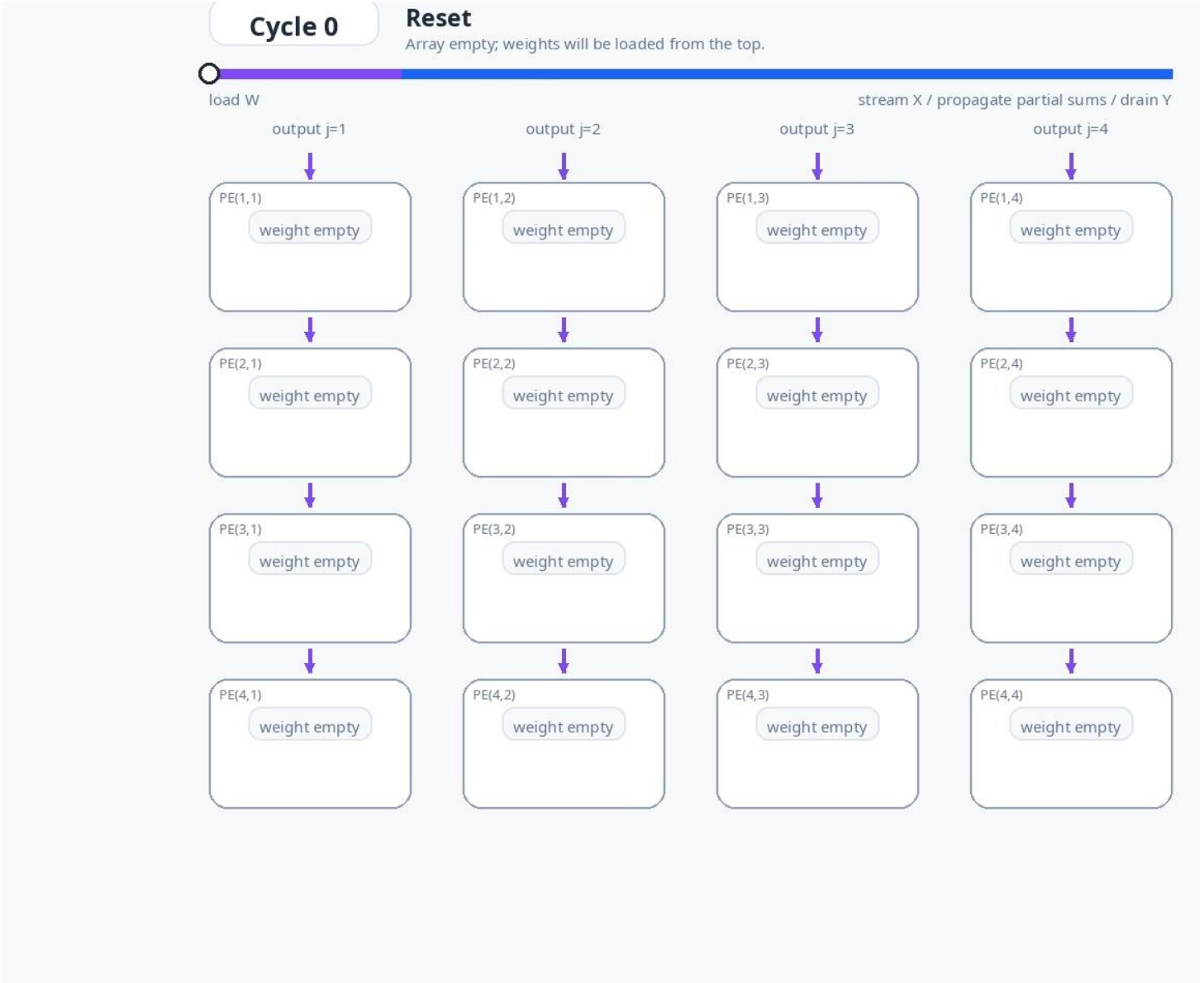
- **Scalar instructions:** `sbr, sadd, sld/sst, ...`
- **VU instructions:** `vadd, vld/vst, ...`
- **SA instructions:** `vpop, vmatpush, vmatmul, ...`
- **DMA instructions:** `dma.{general,hbm,vmem}`
- **Other:** ...

Special Instructions for each Component

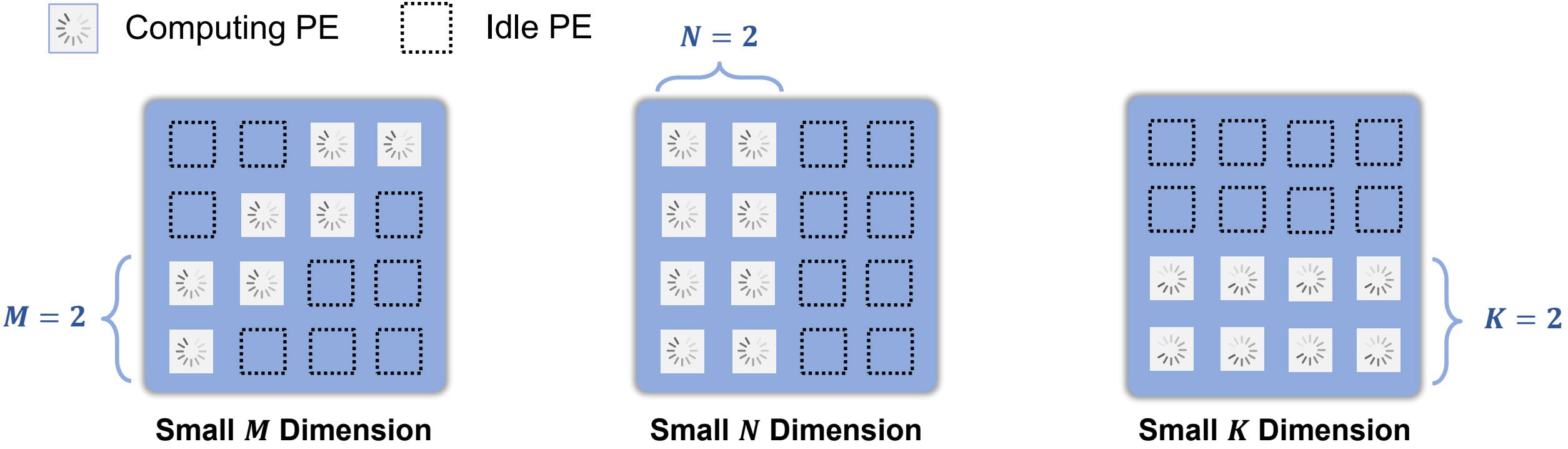
The Systolic Array: The Specialized Engine for Matrix Multiplications



The Systolic Array in Action



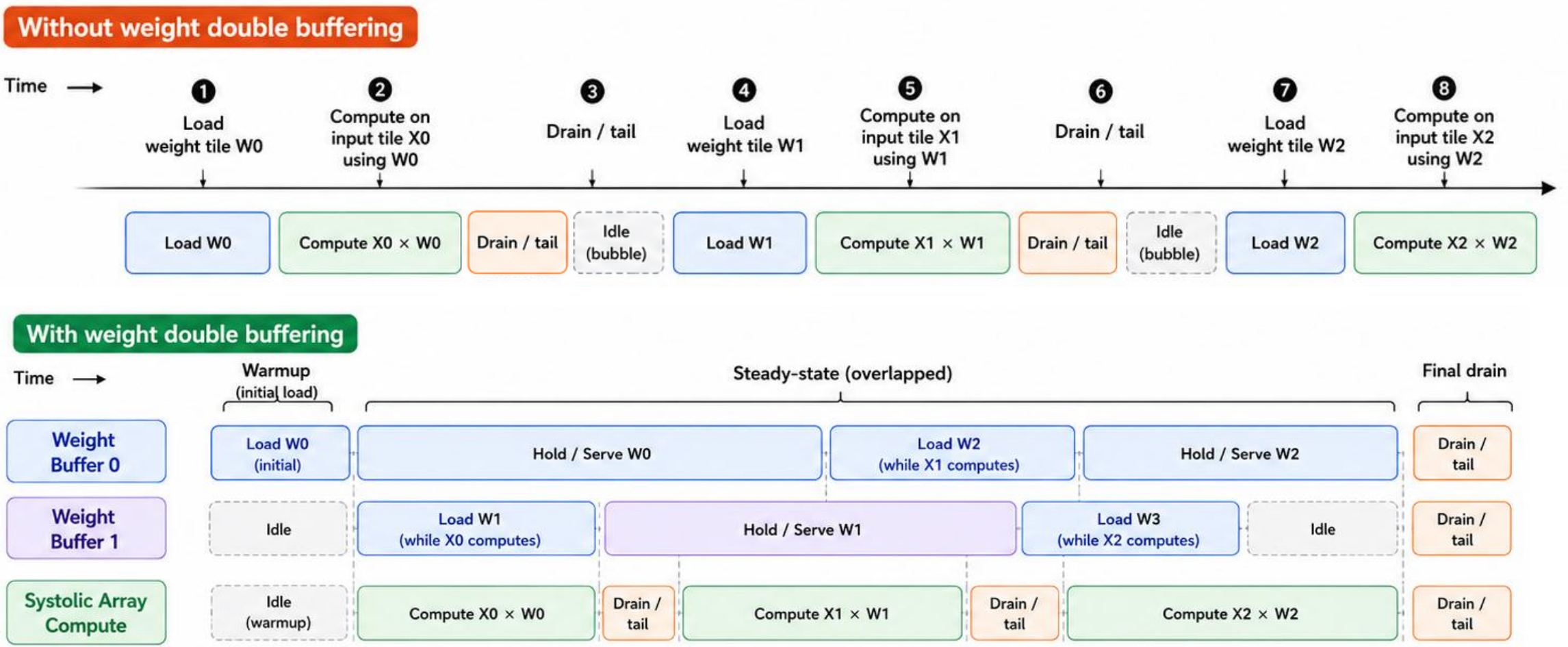
Design Concerns of Systolic Arrays: Padding & PE Utilization



SA spatial underutilization caused by small tensors in a matrix multiplication of shape $[M, K] \times [K, N] \rightarrow [M, N]$.

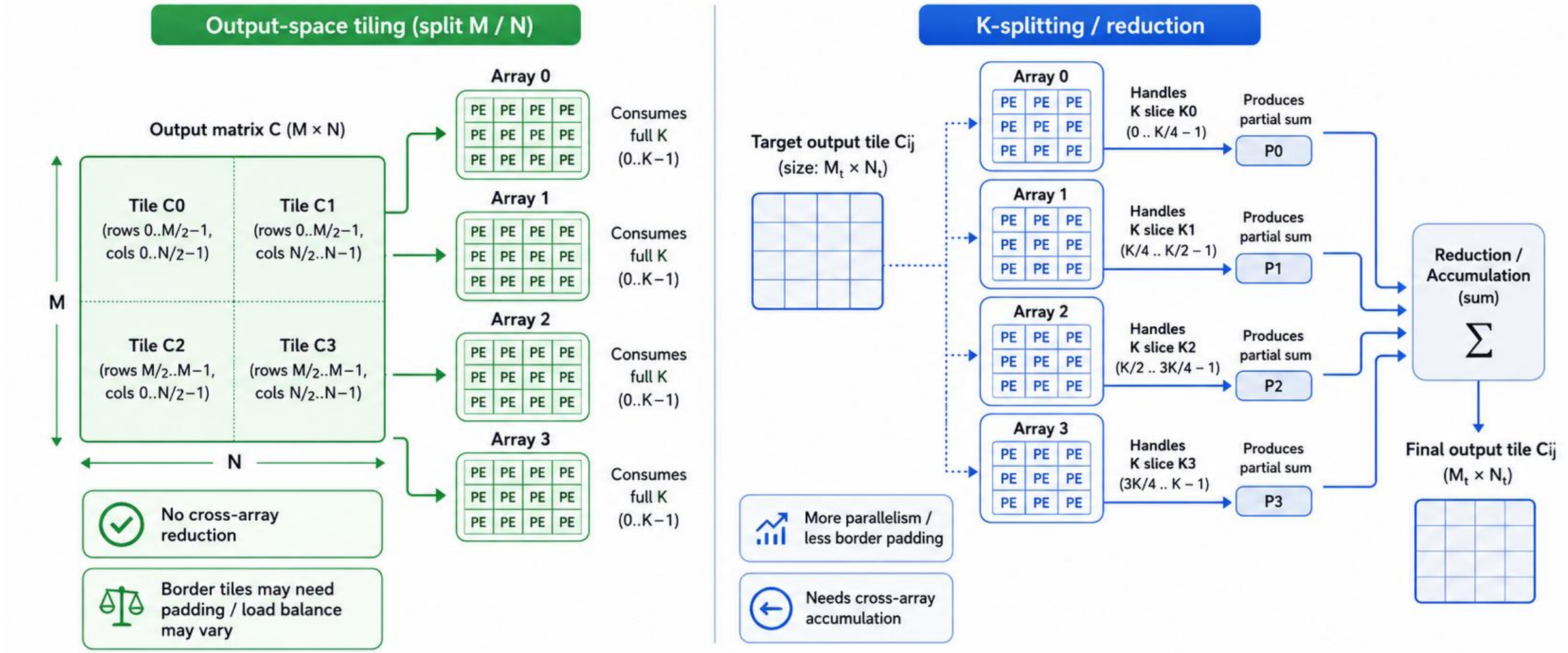
Larger SAs:
+ High peak FLOPS
+ Less control logic overhead
- More padding, PE underutilization

Design Concerns of Systolic Arrays: Warm-up Latency



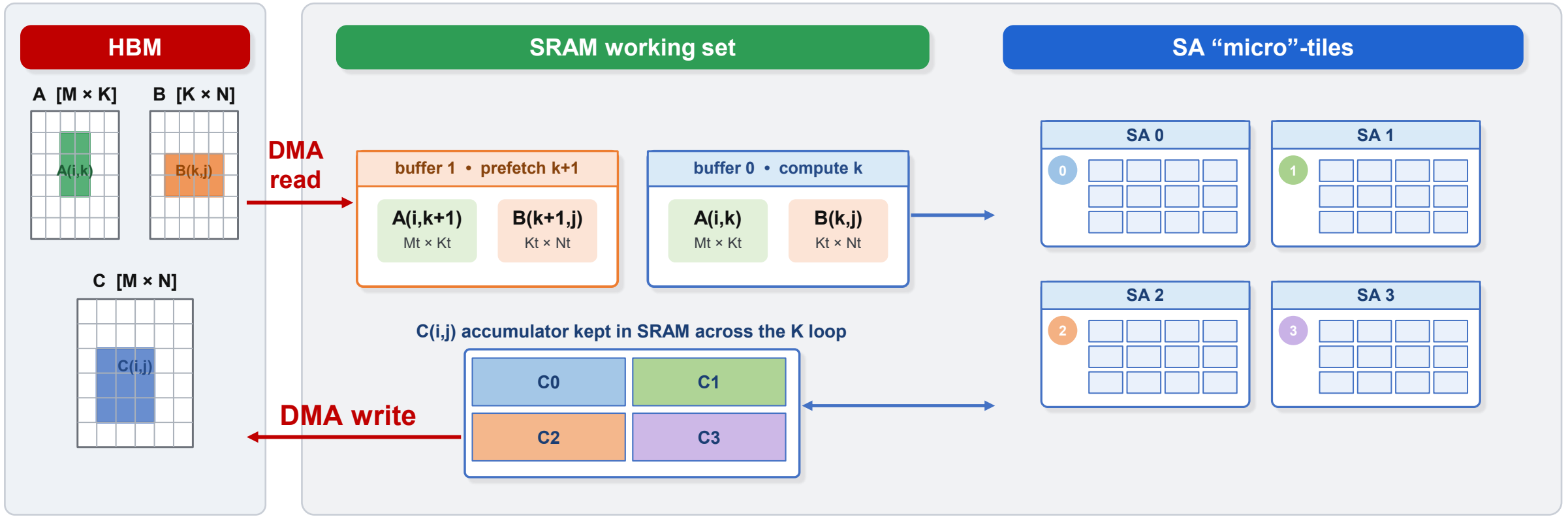
Double buffering for loading weights is often required to hide the warm-up latency of systolic arrays.

Design Concerns of Systolic Arrays: Multi-SA Mapping & Tiling



Mapping a tensor operator to multiple SAs must consider both SA hardware specifications and tensor shapes.

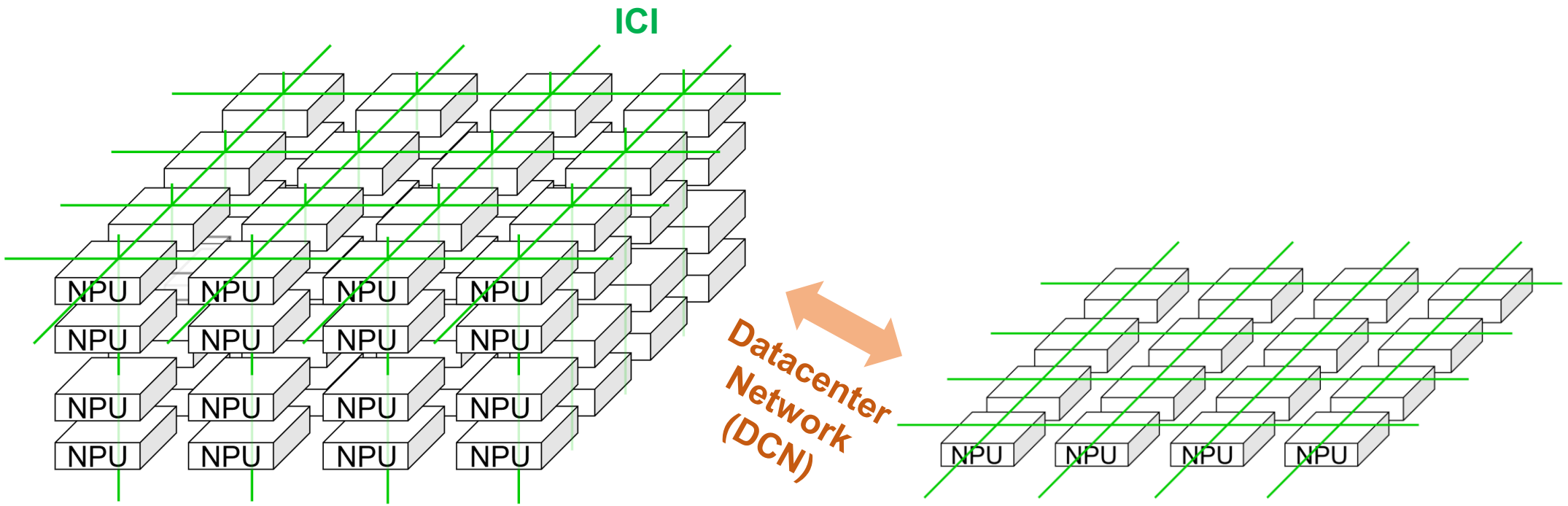
NPU Memory Subsystem: SRAM Tiling & Double Buffering



Ping-pong buffering hides HBM latency: load K-tile k+1 while the SAs compute K-tile k

Pipeline step	0	1	2	3	4	5
HBM → SRAM	Load k0 → B0	Load k1 → B1	Load k2 → B0	Load k3 → B1	—	—
SA compute		Compute k0 from B0	Compute k1 from B1	Compute k2 from B0	Compute k3 from B1	—
SRAM → HBM	—	—	—	—	—	Store C(i,j)

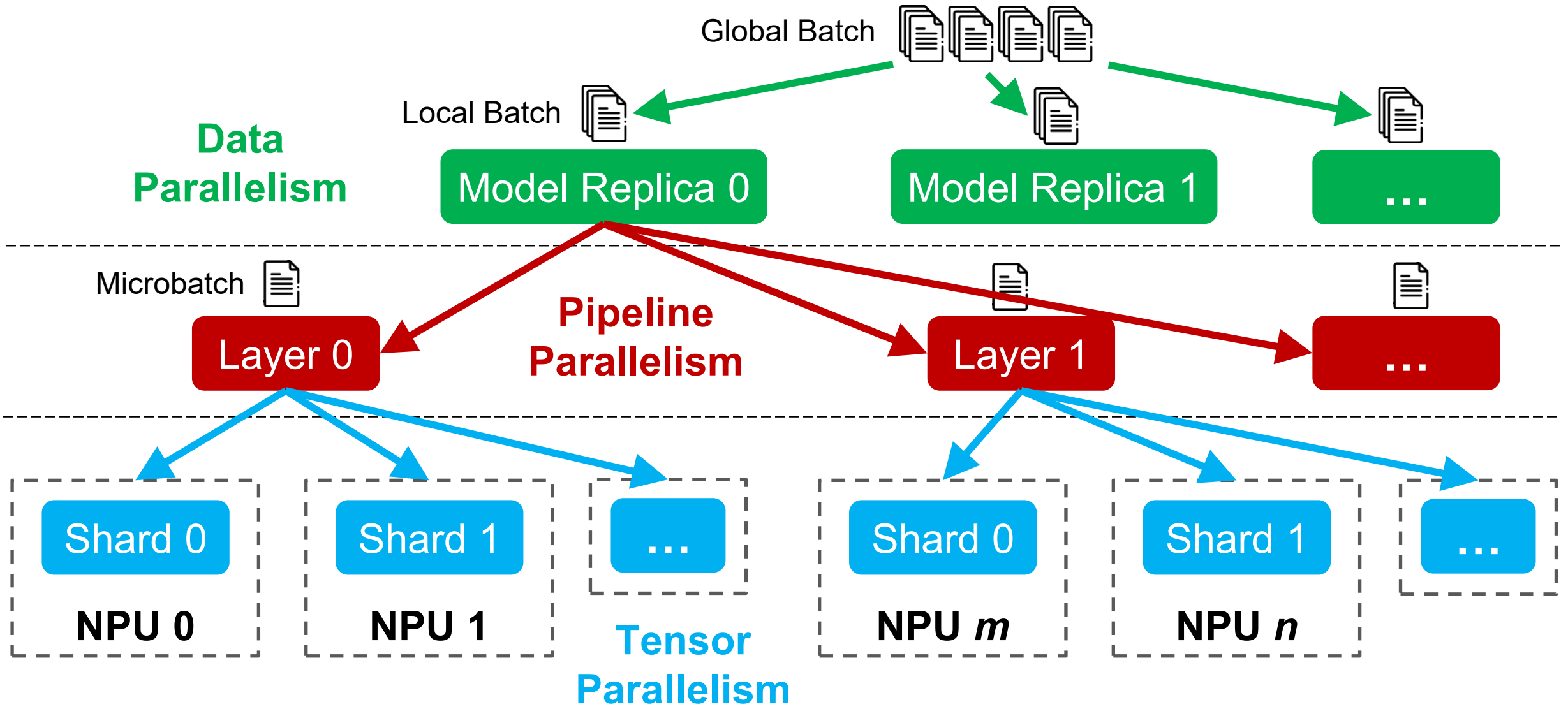
Scaling Up: The NPU Pod Architecture



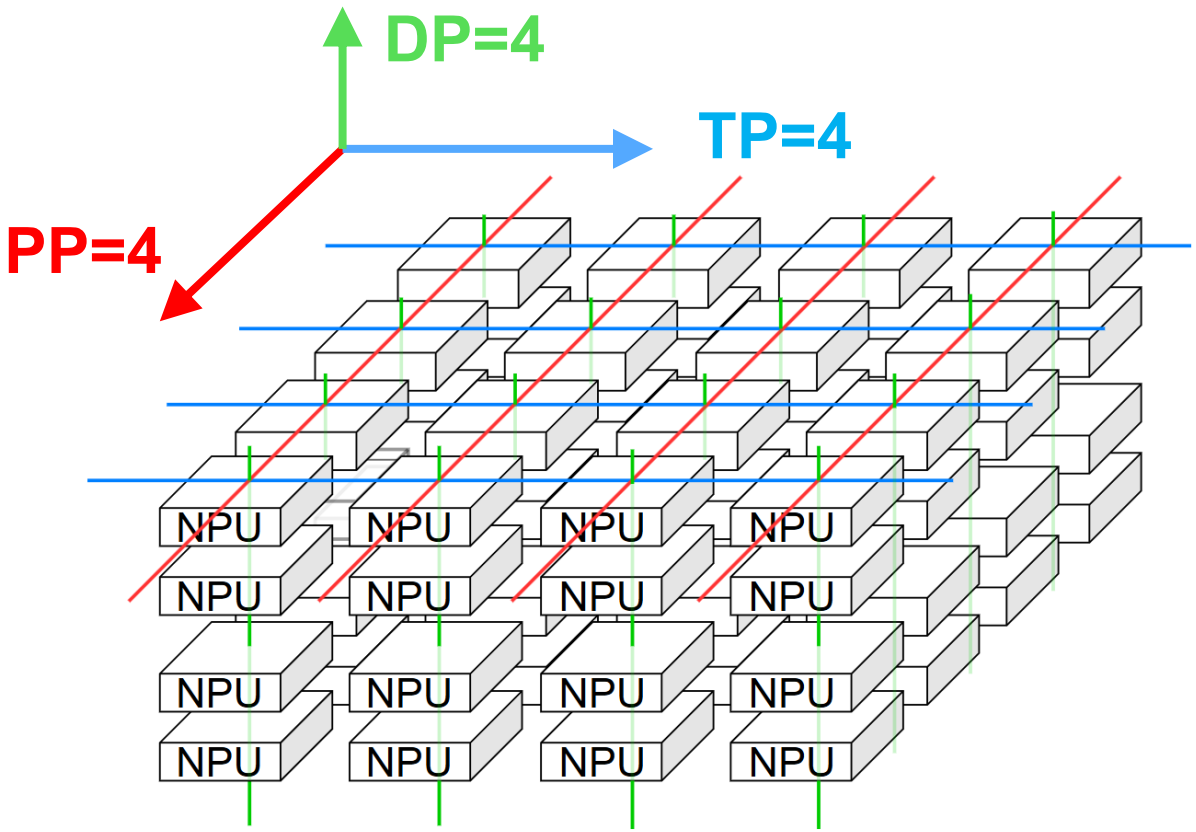
NPU Pod (3D Torus)

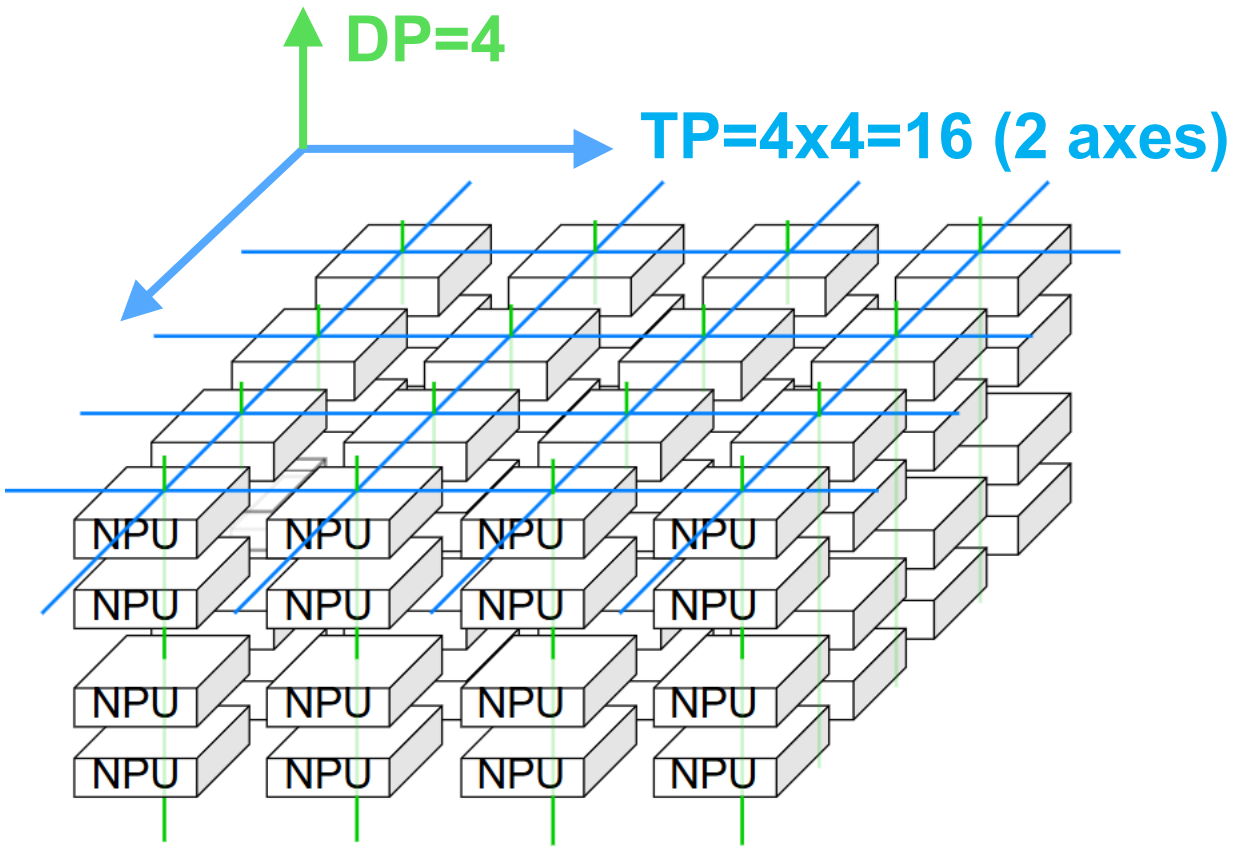
NPU Pod (2D Torus)

Mapping ML Computation to an NPU Pod: Parallelism Strategies



Mapping Logical Parallelism Axes to Physical Axes





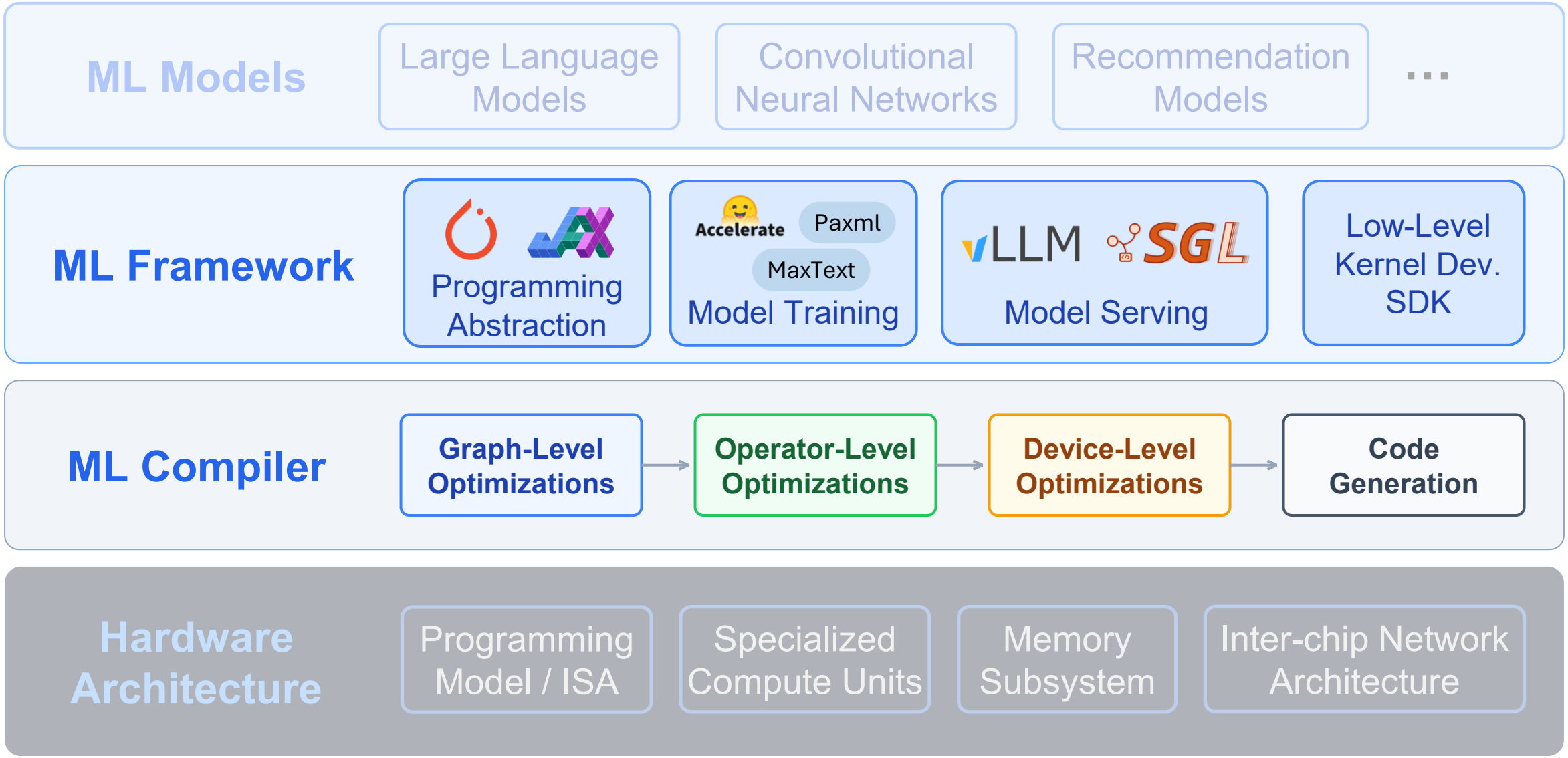
+ Balanced trade-off between 3 parallelisms
- Single axis has a limited bandwidth

+ Higher bandwidth for tensor parallelism
- Limited choices of parallelism combinations

NPU vs. GPU: Architectural Differences

	GPU	NPU
Hardware Organization	**Massively Parallel Wimpy SMs** → Chip → **NVLink system**	**A Few Brawny Cores** → Chip → **ICI pod**
Memory Hierarchy	**Per-thread registers** / Scratchpad SRAM / **L2** / HBM	**Vector registers** / Scratchpad SRAM / HBM
Compute Units	CUDA cores **/ Tensor Cores**	Vector units **/ Systolic arrays**
Interconnect	NVLink / **NVSwitch**	ICI link / **2D/3D Torus**
Execution Model	**thread / warp / thread block / grid, SIMT**	**Single thread per core, VLIW**
Parallelism Granularity	**Thread / warp / block**	**Tile / core**
Instruction Set	Scalar + **Tensor Core** + TMA	Scalar + **VU** + **SA** + DMA

Overview of Modern AI Stack

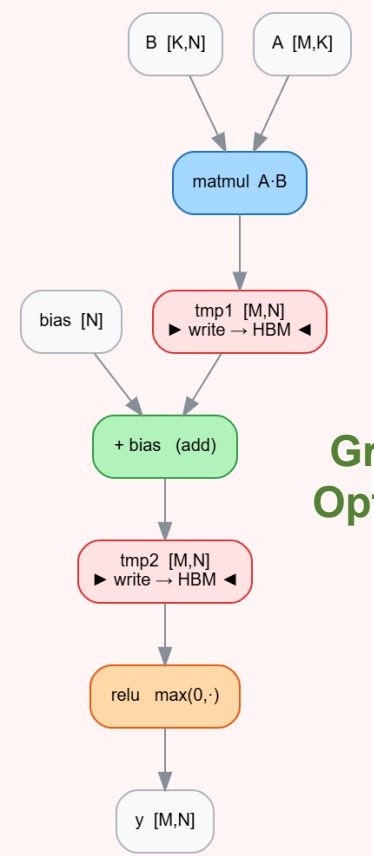


ML Compiler Workflow

ML Program (torch, JAX, etc.)

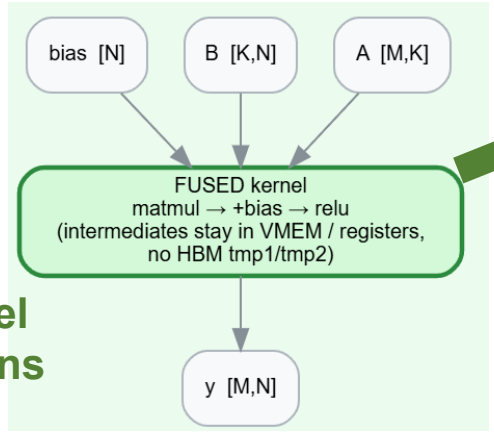
```
# y = relu(A · B + bias)
def layer(A, B, bias):
    return jnp.maximum(A @ B + bias, 0.0)
```

Original Graph



Graph-Level Optimizations

Optimized (Fused) Graph



Operator IR

```
for i in 0..256:
  for j in 0..256:
    acc = 0
    for k in 0..512:
      acc += A[i,k] * B[k,j]
    C[i,j] = relu(acc + bias[j])
```

Operator-Level Optimizations

Optimized (Tiled) Operator IR

```
for io in 0..256 step 128:                    // row tile
  for jo in 0..256 step 128:                  // col tile
    Acc[128,128] = 0
    for ko in 0..512 step 128:
      Atile = copy A[io:io+128, ko:ko+128]    // HBM→VMEM
      Btile = copy B[ko:ko+128, jo:jo+128]    // HBM→VMEM
      Acc  += Atile * Btile
    Ctile = relu(Acc + bias[jo:jo+128])
    copy Ctile → C[io:io+128, jo:jo+128]      // VMEM→HBM
```

ML Compiler Workflow

Lower Operator IR to VLIW

Instruction-level IR (SSA form)

```
...   ; ── ko-loop body: Acc[128,128] += Atile · Btile  (Btile pre-gated into MXU0) ──
0x3e: { %v310 = vld.f32 [vmem:Atile+0x000]  ;;  %v311 = vld.f32 [vmem:Atile+0x400] }  ; A rows → vregs
0x3f: { vmatpush.bf16 %v310                 ;;  vmatpush.bf16 %v311 }                  ; stream A into MXU0
0x40: { vmatmul.bf16  m0                    ;;  %s58 = sadd.s32 %s57, 128 }            ; fire MAC + ko+=128
...
0x47: { %v312 = vpop.f32 m0                 ;;  %v313 = vpop.f32 m0 }                  ; pop result (2 of 16)
0x48: { %v330 = vadd.f32 %v300, %v312       ;;  %v331 = vadd.f32 %v301, %v313 }        ; Acc' = Acc + result
...
```

Device-Level Optimizations & Transformations (e.g., register allocation)

Final VLIW

```
...   ; ── same body, physical vregs; SSA names collapsed onto a finite file ──
0x3e: { v8 = vld.f32 [vmem:Atile+0x000]     ;;  v9 = vld.f32 [vmem:Atile+0x400] }     ; → v8, v9
0x3f: { vmatpush.bf16 v8                    ;;  vmatpush.bf16 v9 }                    ; v8,v9 DEAD after push
0x40: { vmatmul.bf16  m0                    ;;  s9 = sadd.s32 s9, 128 }
...
0x47: { v8 = vpop.f32 m0                    ;;  v9 = vpop.f32 m0 }                    ; REUSE v8,v9
0x48: { v2 = vadd.f32 v2, v8                ;;  v3 = vadd.f32 v3, v9 }                ; Acc PINNED in v2,v3 16-
vreg Acc + tiles > 32-vreg file → spill a COLD accumulator lane to VMEM:
0x49: { vst.f32 [vmem:spill+0x080], v16     ;;  v16 = vld.f32 [vmem:spill+0x0c0] }    ; SPILL ↔ RELOAD
...
```

ML Compiler Optimizations

Graph-Level

- **Operator Fusion:** Merge operators to reduce overhead.
- **Algebraic Simplification:** Rewrites expressions into cheaper forms.
- **Dead-code Elimination:** Removes unused operations.
- **Layout Propagation:** Optimizes reshapes and transposes.
- **Multi-device Sharding:** Partitions tensors across NPUs.
- ...

Mostly device-agnostic, requires minimal HW info

Operator-Level

- **Tiling:** Improves on-chip data reuse with spatial locality.
- **Loop Transformation:** Improves locality and parallelism.
- **Vectorization:** Uses SIMD/Tensor instructions.
- **Auto-tuning:** Selects optimal schedule parameters.
- ...

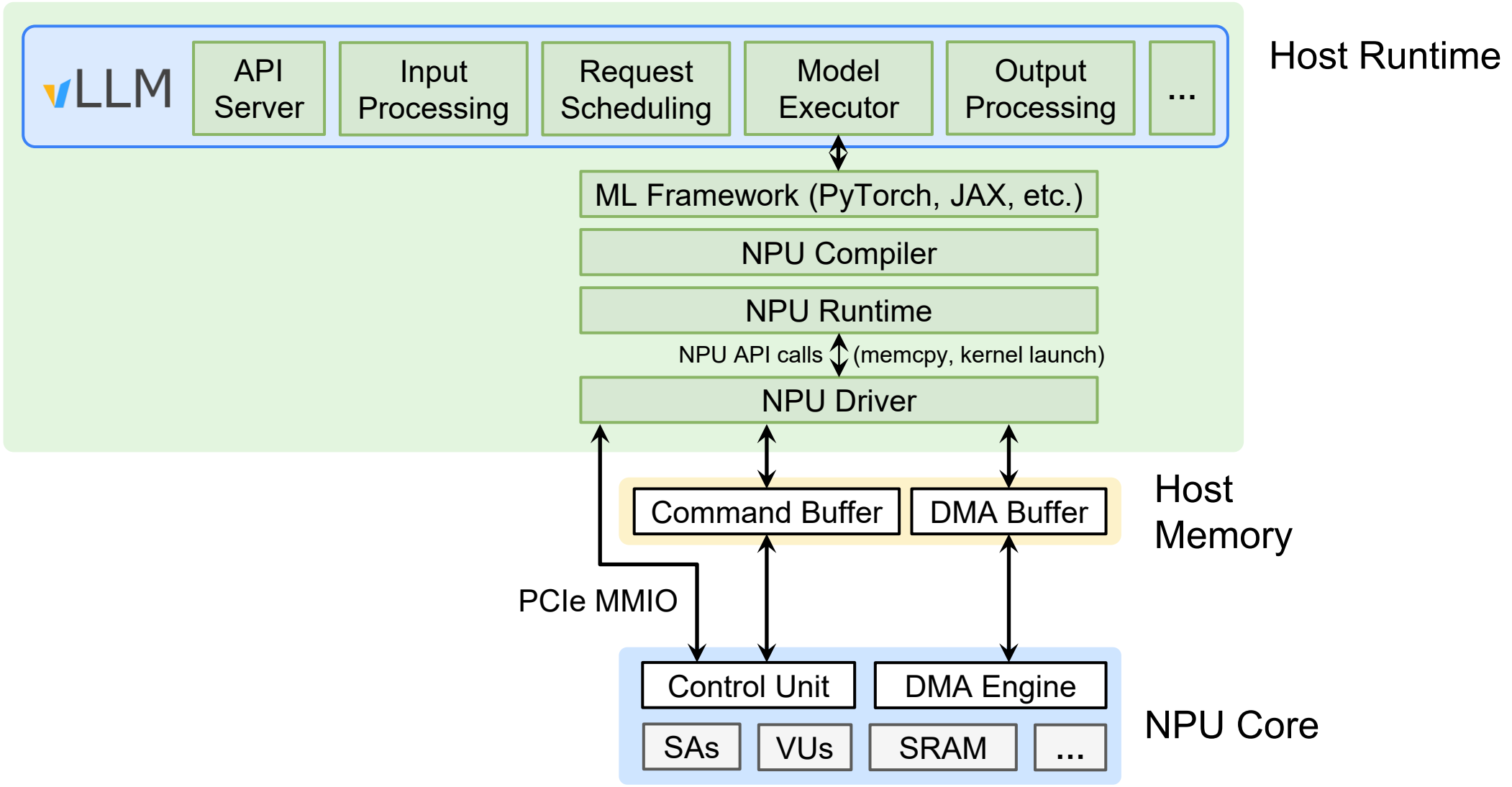
Requires generic HW specs (# ALUs, SRAM size, etc.)

Device-Level

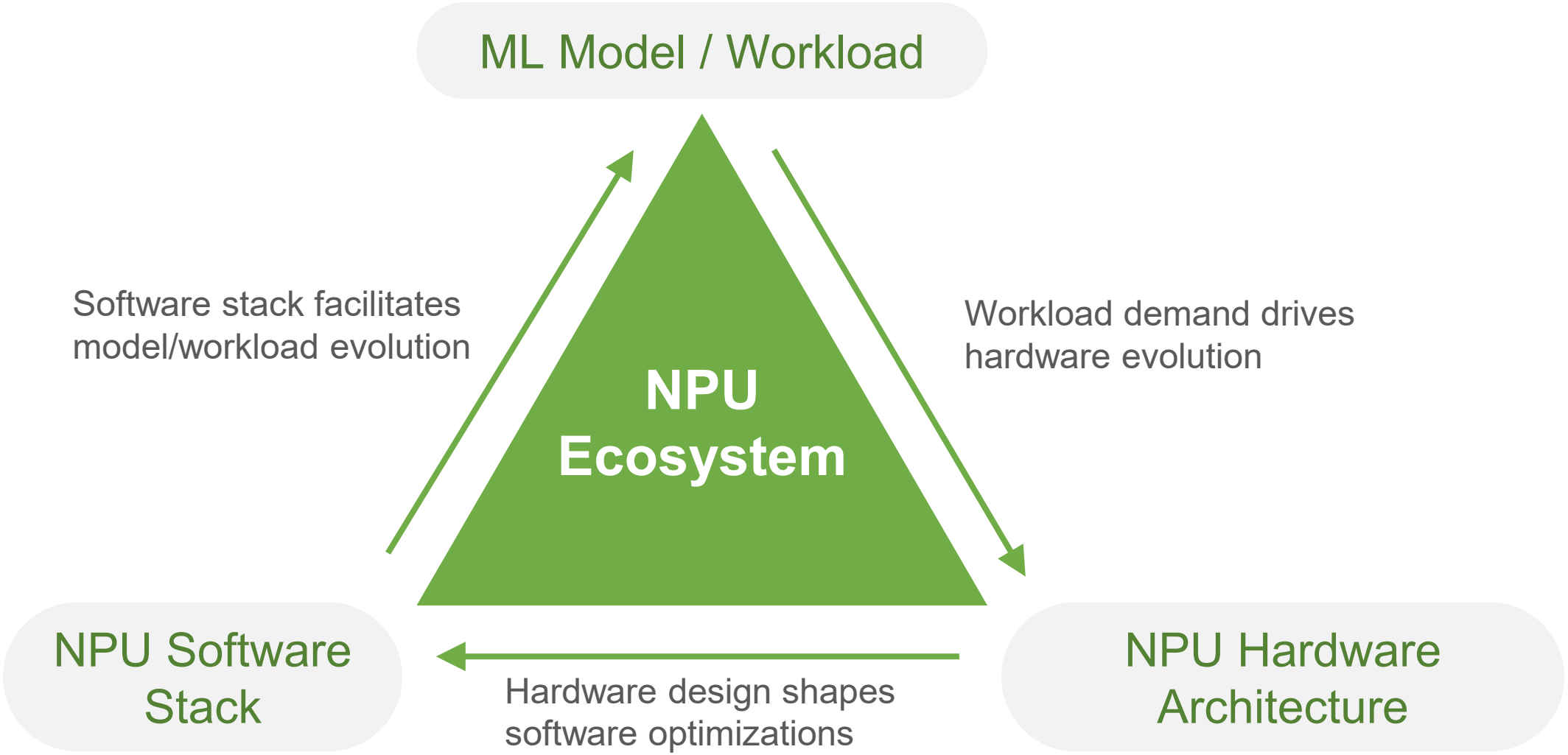
- **DMA Buffering:** Overlaps data transfers with compute.
- **VLIW Packetization:** Bundles instructions for execution.
- **Software Pipelining:** Overlaps loop iterations.
- **SRAM Swizzling:** Minimizes SRAM bank conflicts.
- **Register Allocation:** Minimizes spills and movement.
- ...

Requires detailed HW specifications

ML Frameworks & NPU Runtime: The LLM Serving Stack as an Example



NPU Ecosystem: Co-Evolution of Workloads, Software, and Hardware



All layers in the NPU ecosystem must be developed in a coordinated manner.

The Missing Pieces for NPU Ecosystem

	Kernel SDK	Compiler: Graph/ Operator-Level	Compiler: Device-Level	Training/ Serving Framework	Open ISA/ Programming Model	Arch. Simulator	Hardware Prototype
GPU	CUDA, Triton	PyTorch, XLA, TVM	nvcc, PTXAS	vLLM, SGLang, PyTorch	CUDA PTX, SASS	Accel-Sim	Vortex, Ventus
NPU	**Missing** Pallas, NKI (proprietary)	**Partial support** PyTorch, XLA, TVM	**Fragmented** TVM BYOC, XLA custom backends, MLIR stacks	**Partial support** vLLM-TPU, PyTorch	**Missing / Fragmented**	**Fragmented** SCALE-Sim, ASTRA-Sim, NeuSim, …	**Fragmented** Gemmini, NVDLA, …

Missing: No open-source solution for NPUs.
Partial support: Existing GPU-centric solutions can partially support NPUs but may still require nontrivial adaptation.
Fragmented: Multiple solutions exist for NPUs, but they cannot be directly composed into a unified, end-to-end ecosystem.

An end-to-end NPU ecosystem requires not only separate components at each layer but also composability.