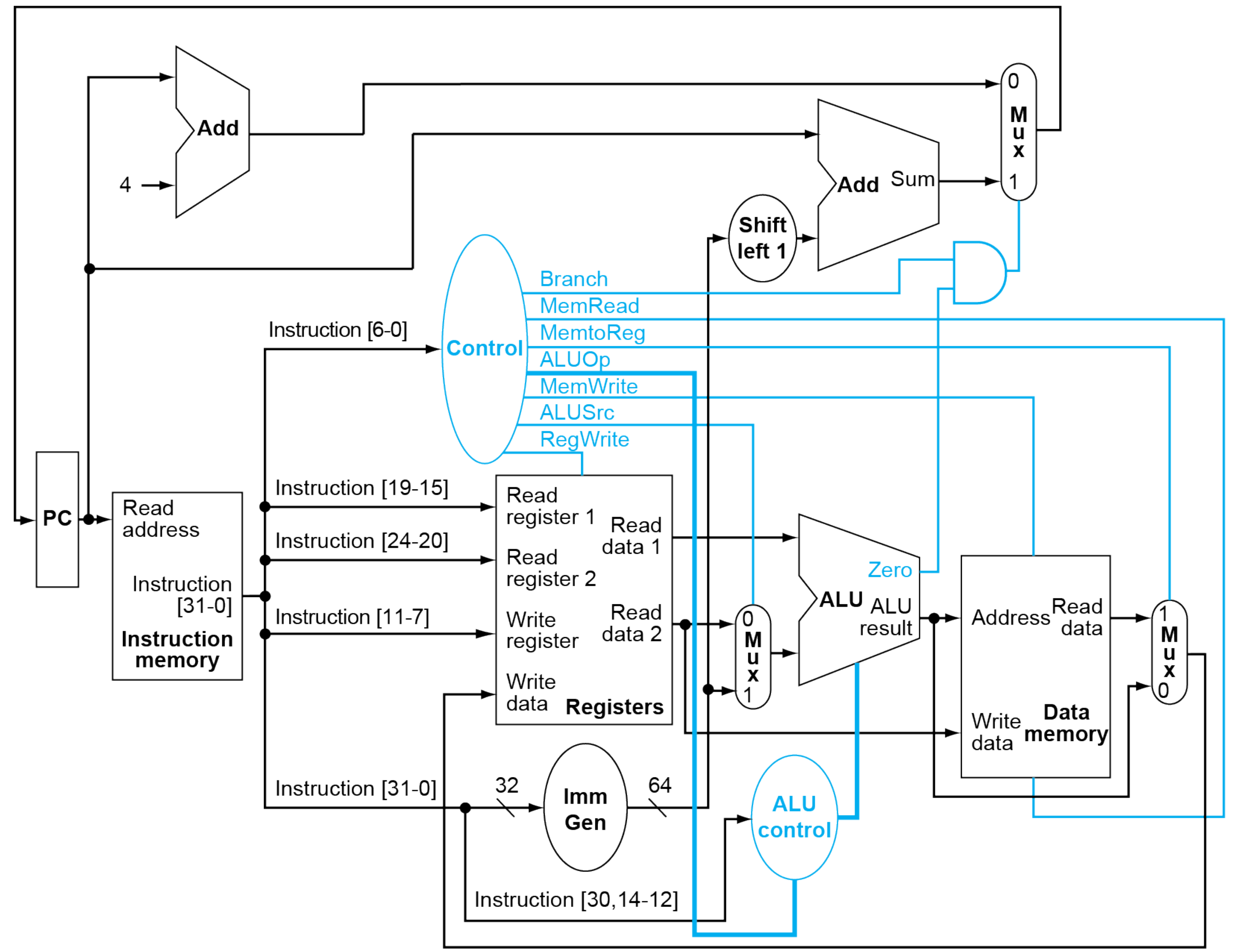
NATIONAL RESEARCH UNIVERSITY

Computer Architecture and Operating Systems Lecture 16: Domain-specific architectures. Tensor Processing Unit.

Alexey Kanakhin

akanakhin@hse.ru

Conventional 5-stages pipeline



Possible improvements

- Modern performance tuning techniques:
  - Deep memory hierarchy
  - Wide SIMD units
  - Deep pipelines
  - Branch prediction
  - Out-of-order execution
  - Speculative prefetching
  - Multithreading
  - Multiprocessing
- Further improvement:
  - Domain-specific architectures

Guidelines for DSAs

- Use **dedicated memories** to minimise data movement
- Invest resources into **more arithmetic units** or bigger memories
- Use the easiest form of **parallelism** that matches the domain
- Reduce **data size and type** to the simplest needed for the domain
- Use a **domain-specific** programming language

DNN Summary

- Batches
  - Reuse weights once fetched from memory across multiple inputs
  - Increases operational intensity
- Quantisation
  - Use 8- or 16-bit fixed point or integer numbers
- Operations
  - Matrix-vector multiply
  - Matrix-matrix multiply
  - Stencil
  - ReLU
  - Sigmoid
  - Hyperbolic tangent

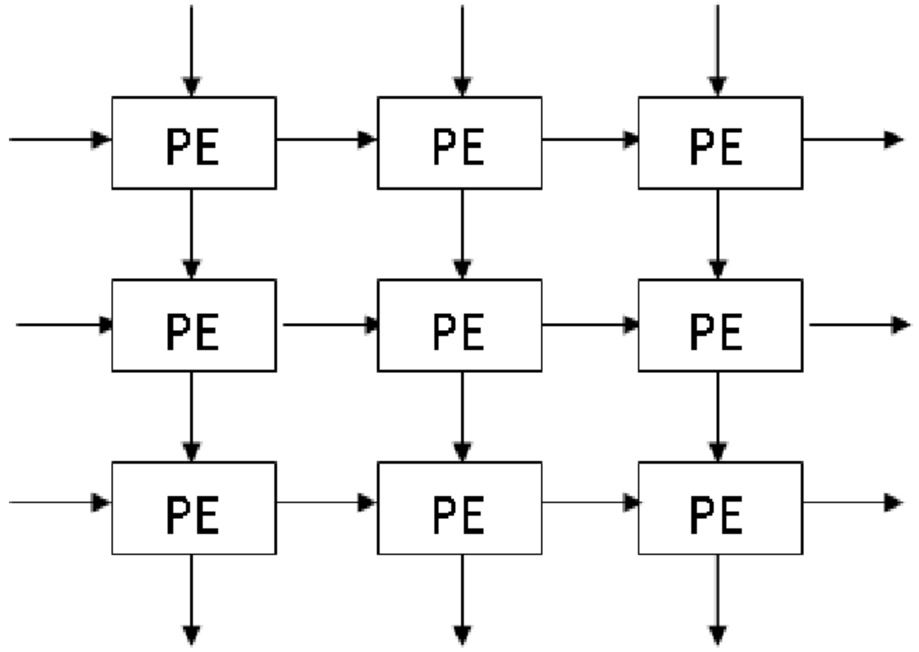
Matrix-Matrix multiplication

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} & a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} & a_{21}b_{13} + a_{22}b_{23} + a_{23}b_{33} \\ a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} & a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} & a_{31}b_{13} + a_{32}b_{23} + a_{33}b_{33} \end{bmatrix}$$

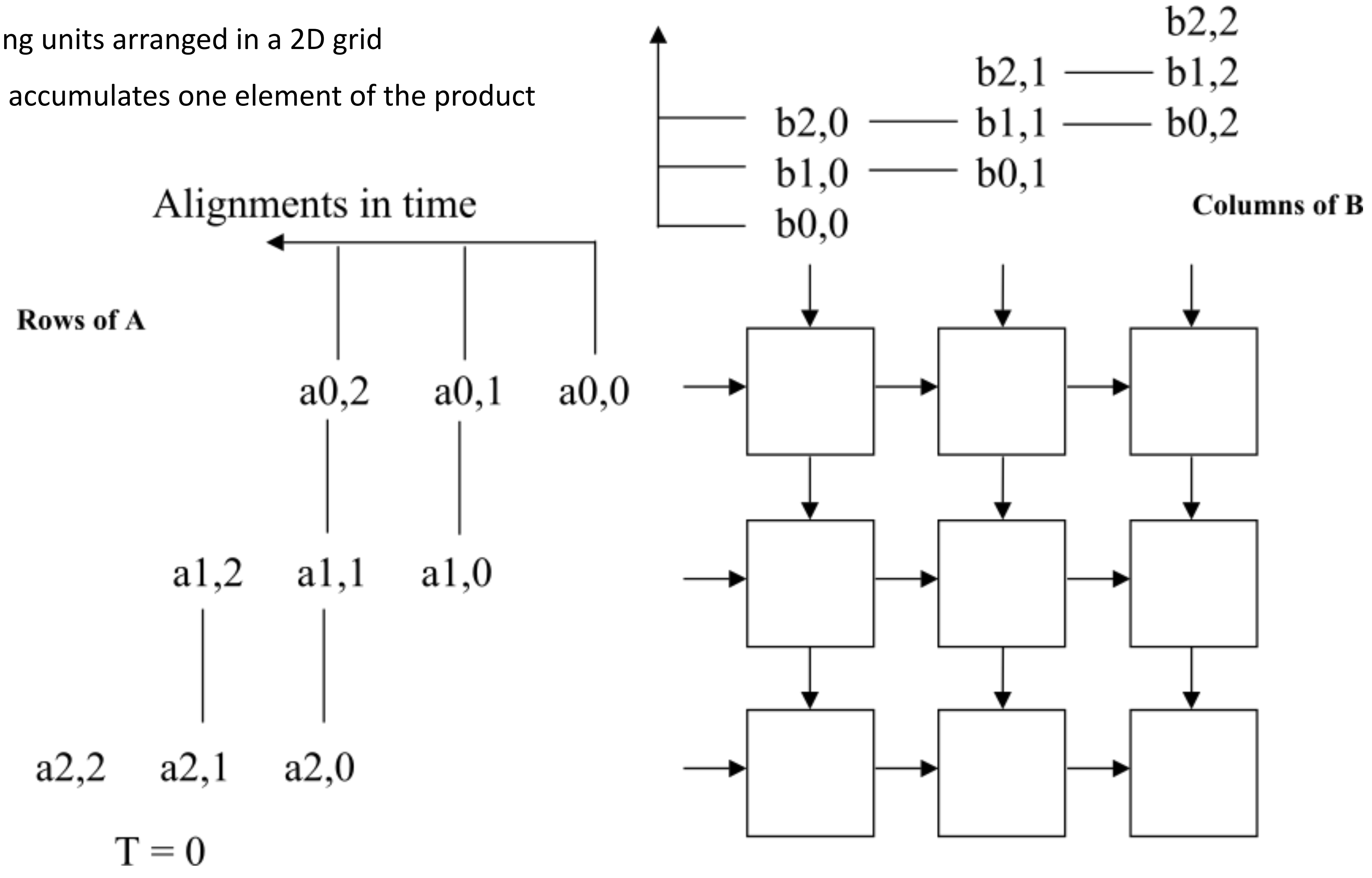
Systolic array (2D pipeline)

- Different from pipelining
  - Nonlinear array structure, multi-direction data flow, each PE may have (small) local instruction and data memory
- Different from SIMD
  - each PE may do something different



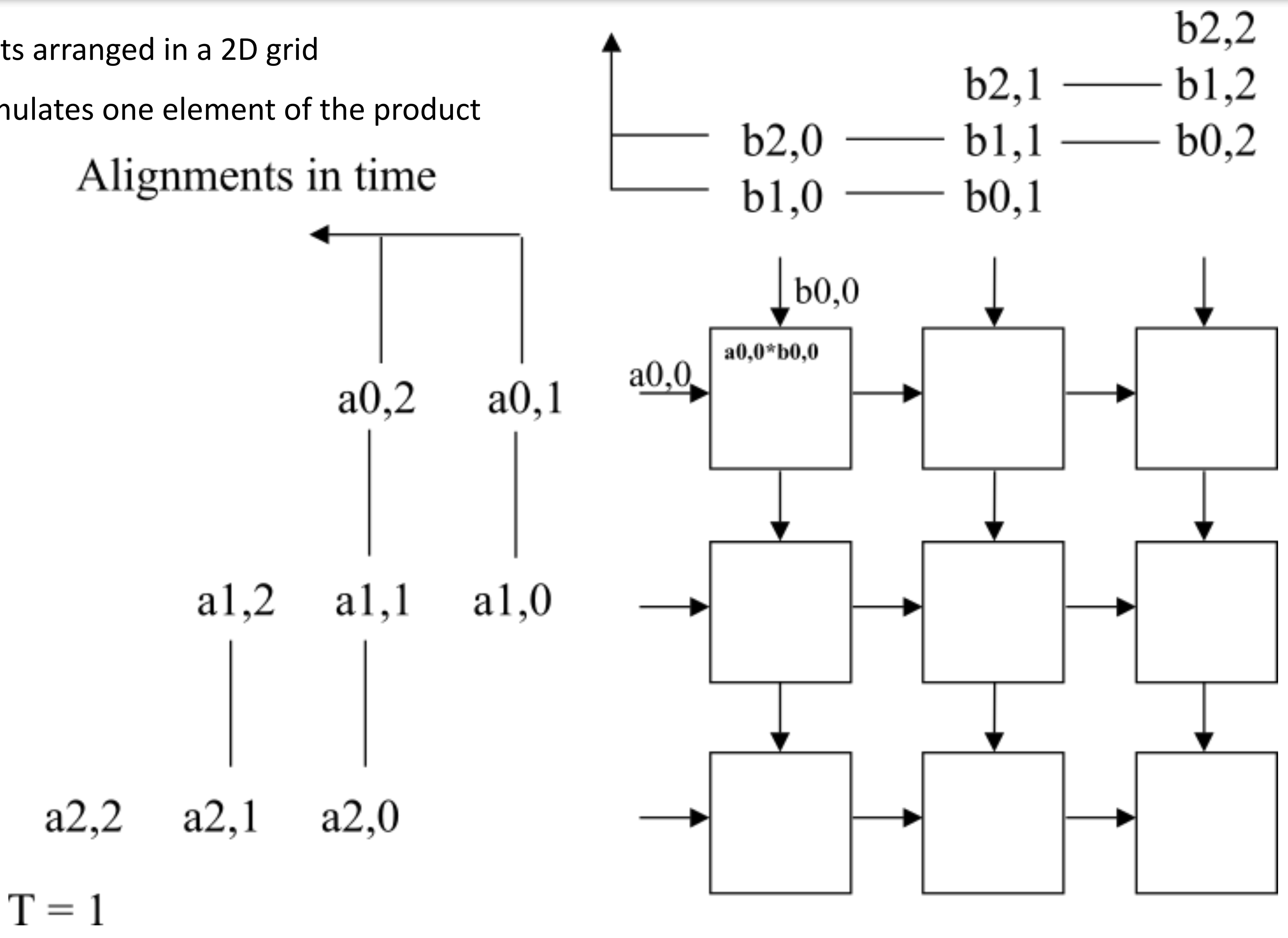
Computation example

- Processing units arranged in a 2D grid
- Each PU accumulates one element of the product



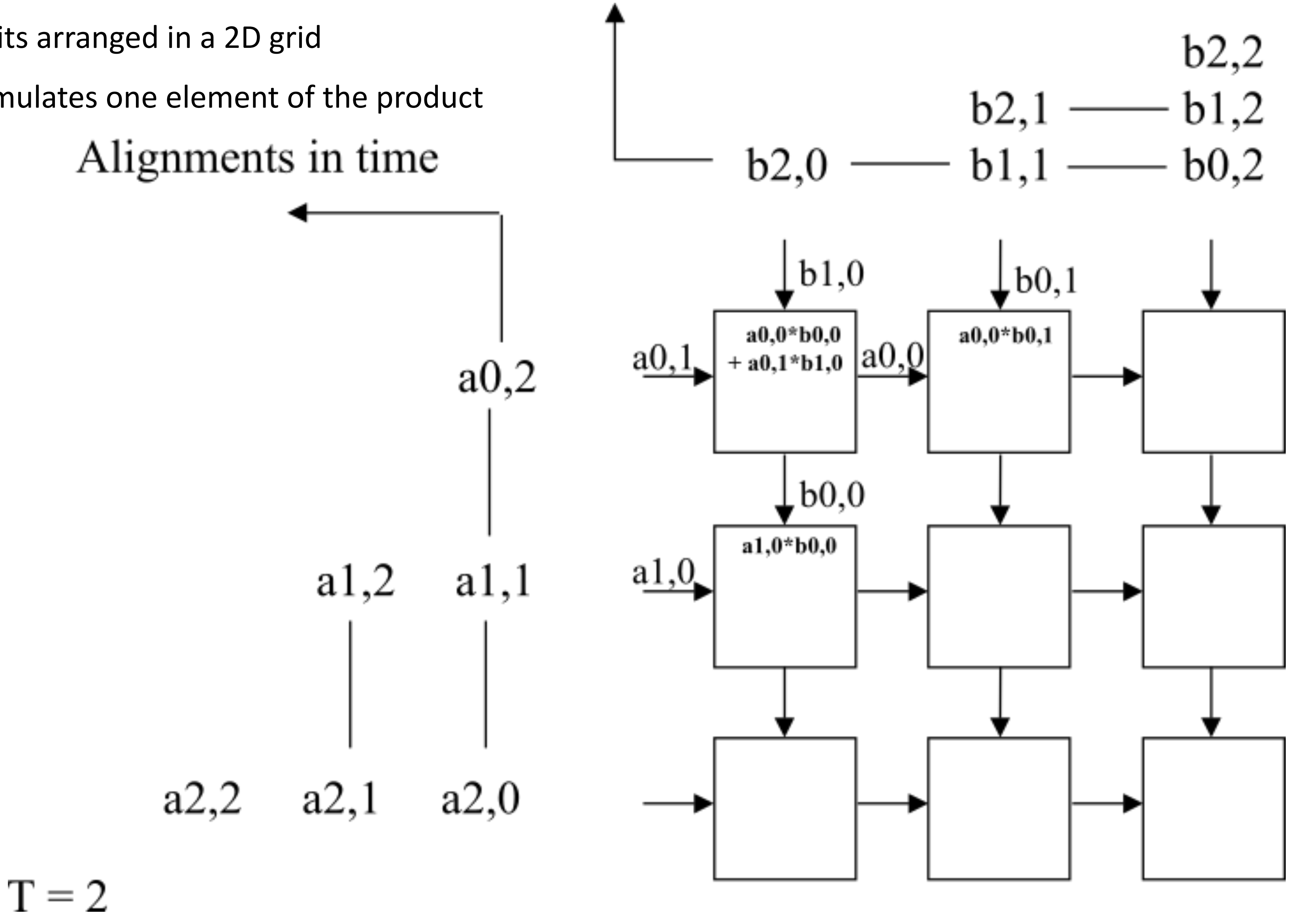
Computation example

- Processing units arranged in a 2D grid
- Each PU accumulates one element of the product



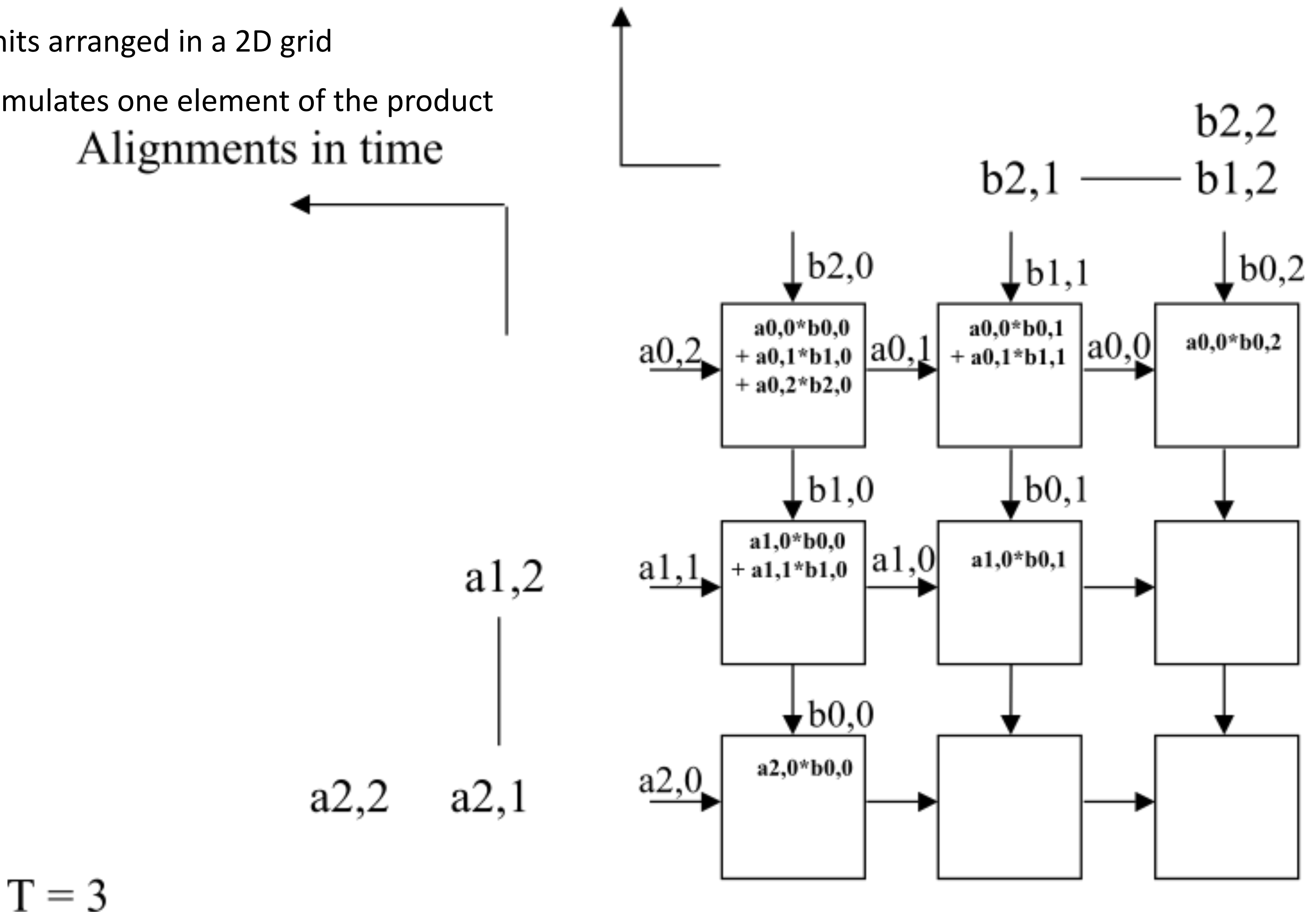
Computation example

- Processing units arranged in a 2D grid
- Each PU accumulates one element of the product



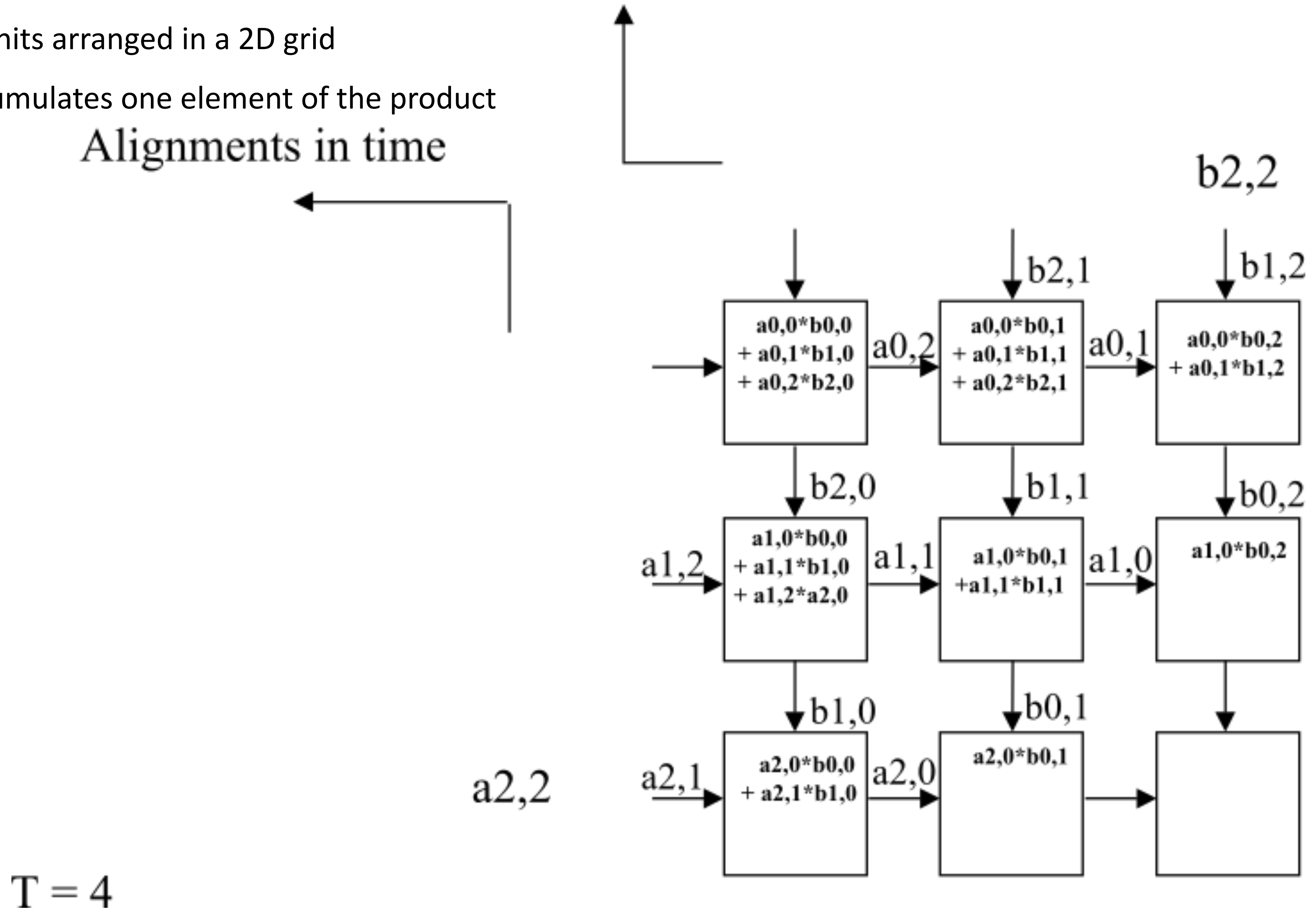
Computation example

- Processing units arranged in a 2D grid
- Each PU accumulates one element of the product



Computation example

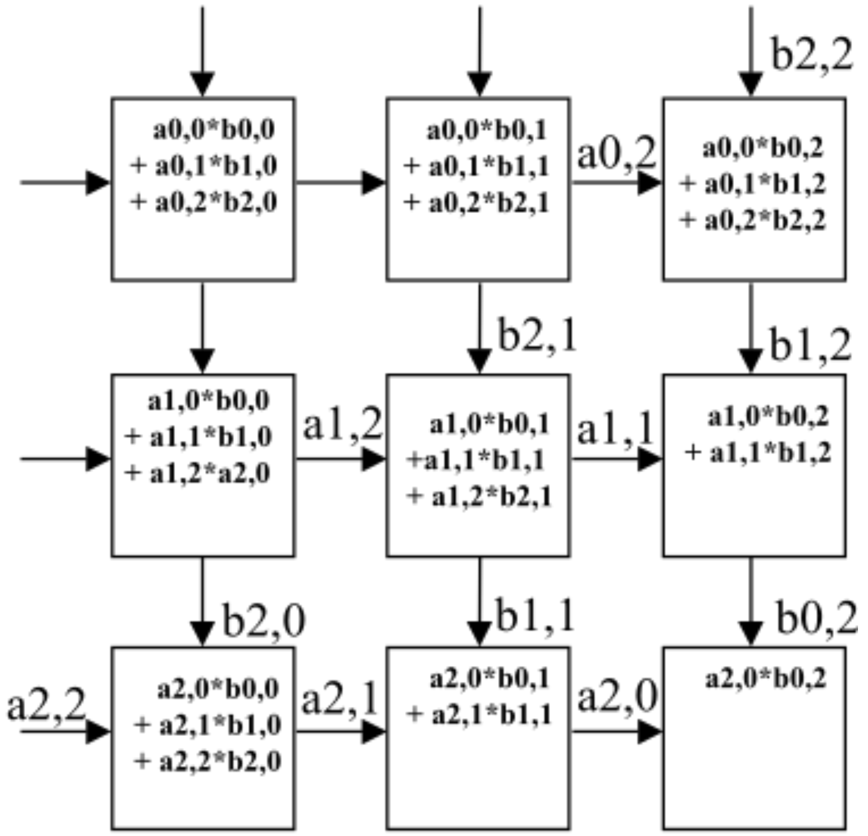
- Processing units arranged in a 2D grid
- Each PU accumulates one element of the product



Computation example

- Processing units arranged in a 2D grid
- Each PU accumulates one element of the product

Alignments in time

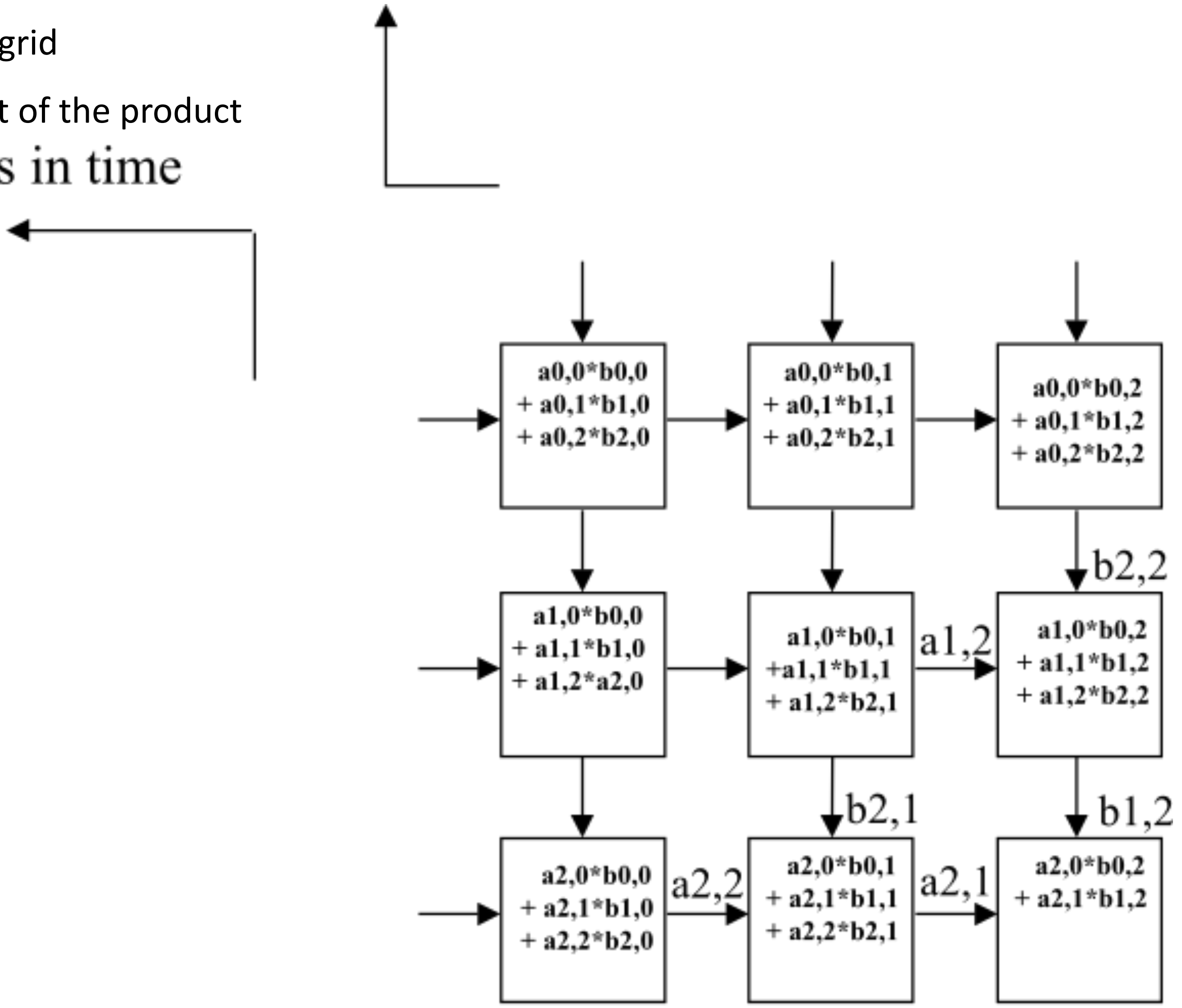

b2,2
a0,0*b0,0 + a0,1*b1,0 + a0,2*b2,0
a0,0*b0,1 + a0,1*b1,1 + a0,2*b2,1
a0,2
a0,0*b0,2 + a0,1*b1,2 + a0,2*b2,2
b2,1
b1,2
a1,0*b0,0 + a1,1*b1,0 + a1,2*a2,0
a1,2
a1,0*b0,1 +a1,1*b1,1 + a1,2*b2,1
a1,1
a1,0*b0,2 + a1,1*b1,2
b2,0
b1,1
b0,2
a2,2
a2,0*b0,0 + a2,1*b1,0 + a2,2*b2,0
a2,1
a2,0*b0,1 + a2,1*b1,1
a2,0
a2,0*b0,2

T = 5

Computation example

- Processing units arranged in a 2D grid
- Each PU accumulates one element of the product

Alignments in time


a0,0*b0,0 + a0,1*b1,0 + a0,2*b2,0
a0,0*b0,1 + a0,1*b1,1 + a0,2*b2,1
a0,0*b0,2 + a0,1*b1,2 + a0,2*b2,2
b2,2
a1,0*b0,0 + a1,1*b1,0 + a1,2*a2,0
a1,0*b0,1 +a1,1*b1,1 + a1,2*b2,1
a1,2
a1,0*b0,2 + a1,1*b1,2 + a1,2*b2,2
b2,1
b1,2
a2,0*b0,0 + a2,1*b1,0 + a2,2*b2,0
a2,2
a2,0*b0,1 + a2,1*b1,1 + a2,2*b2,1
a2,1
a2,0*b0,2 + a2,1*b1,2

T = 6

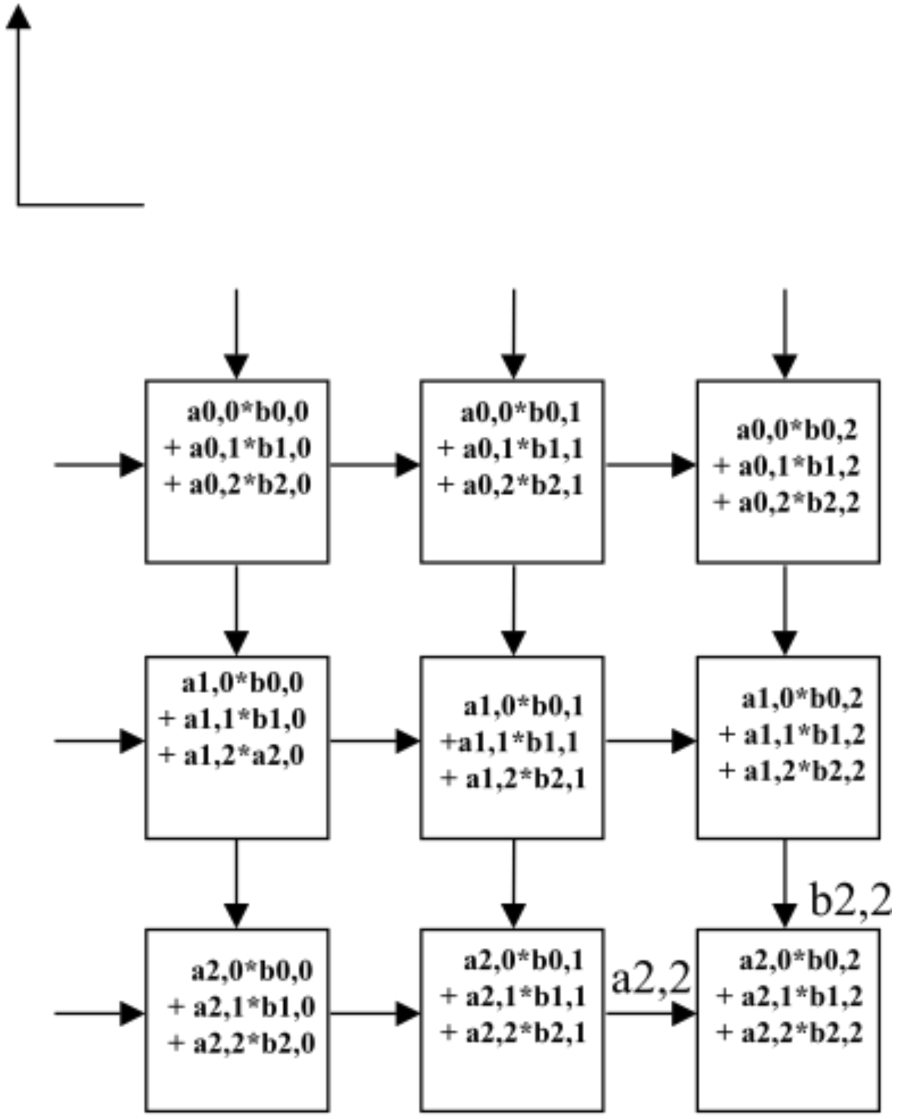
Computation example

- Processing units arranged in a 2D grid
- Each PU accumulates one element of the product

Alignments in time

Done

T = 7



a0,0*b0,0 + a0,1*b1,0 + a0,2*b2,0
a0,0*b0,1 + a0,1*b1,1 + a0,2*b2,1
a0,0*b0,2 + a0,1*b1,2 + a0,2*b2,2
a1,0*b0,0 + a1,1*b1,0 + a1,2*a2,0
a1,0*b0,1 +a1,1*b1,1 + a1,2*b2,1
a1,0*b0,2 + a1,1*b1,2 + a1,2*b2,2
b2,2
a2,0*b0,0 + a2,1*b1,0 + a2,2*b2,0
a2,0*b0,1 + a2,1*b1,1 + a2,2*b2,1
a2,2
a2,0*b0,2 + a2,1*b1,2 + a2,2*b2,2

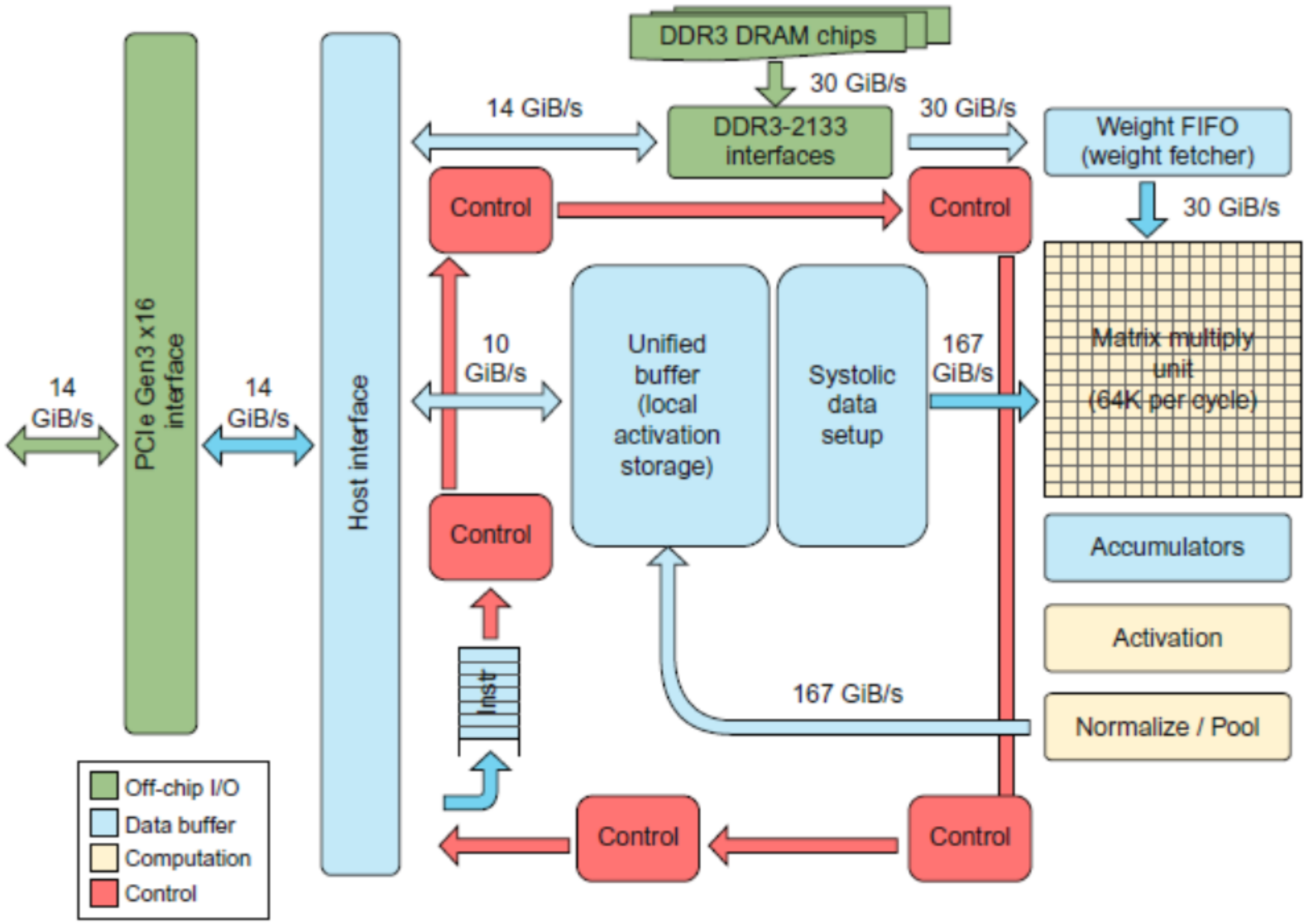
Tensor Processing Unit

- Google’s DNN ASIC
- 256 x 256 8-bit matrix multiply unit
- Large software-managed scratchpad
- Coprocessor on the PCIe bus

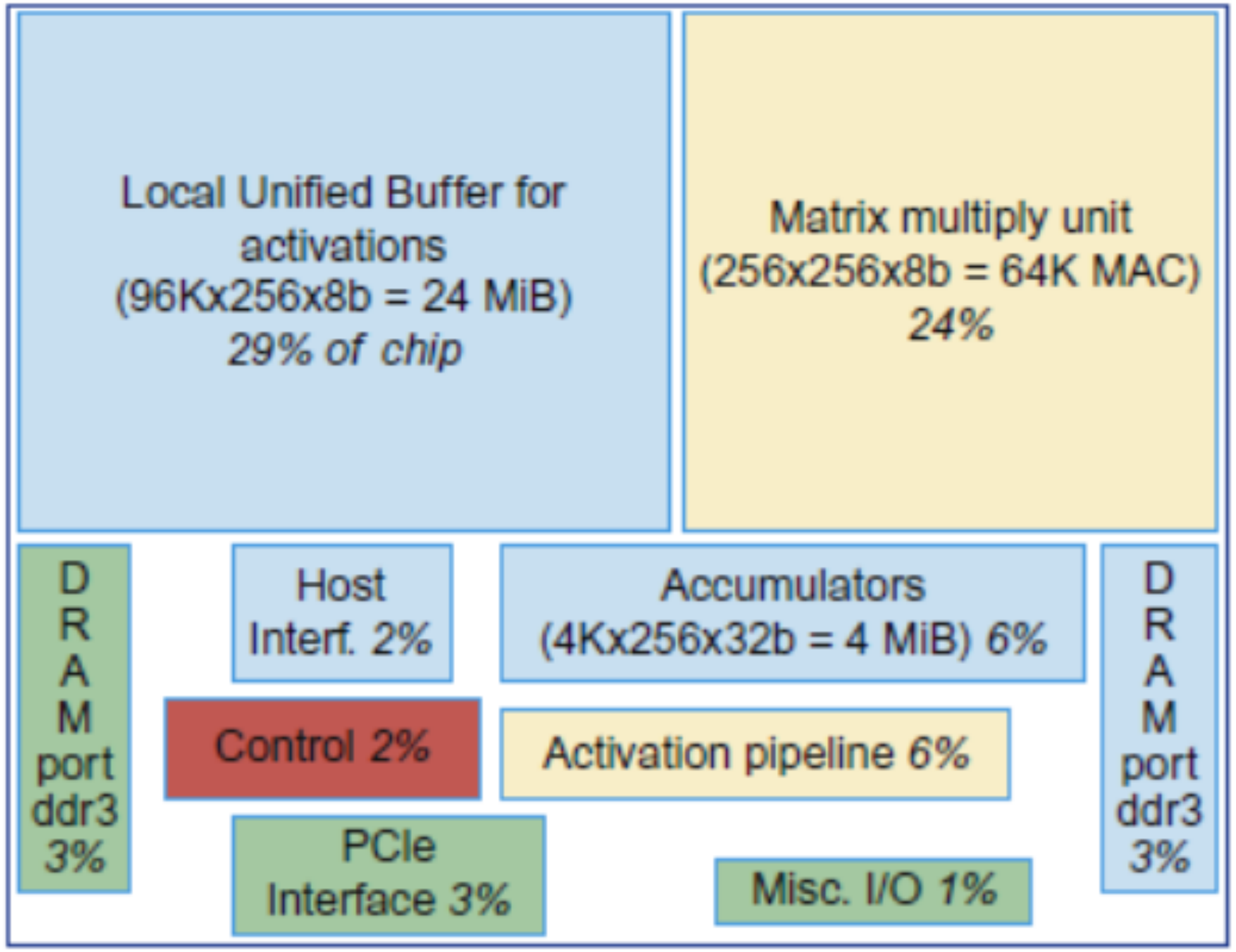
TPU ISA

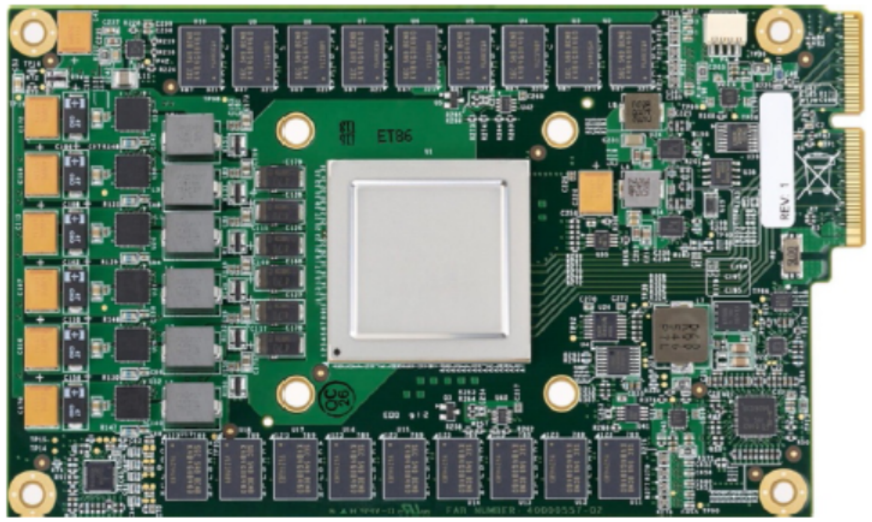
- **Read_Host_Memory**
  - Reads memory from the CPU memory into the unified buffer
- **Read_Weights**
  - Reads weights from the Weight Memory into the Weight FIFO as input to the Matrix Unit
- **MatrixMatrixMultiply/Convolve**
  - Perform a matrix-matrix multiply, a vector-matrix multiply, an element-wise matrix multiply, an element-wise vector multiply, or a convolution from the Unified Buffer into the accumulators
  - takes a variable-sized B*256 input, multiplies it by a 256x256 constant input, and produces a B*256 output, taking B pipelined cycles to complete
- **Activate**
  - Computes activation function
- **Write_Host_Memory**
  - Writes data from unified buffer into host memory

Tensor Processing Unit



Tensor Processing Unit

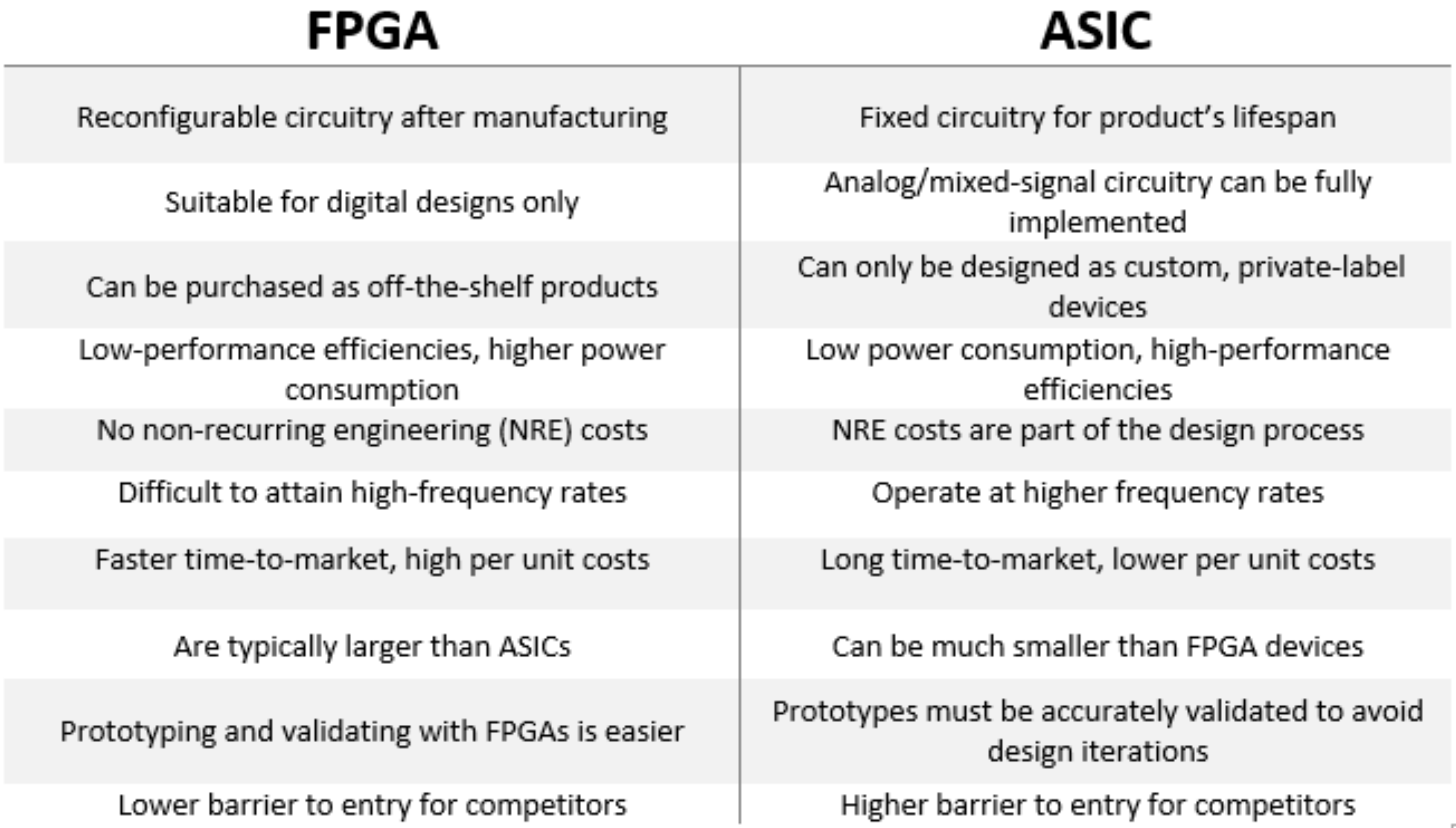

Local Unified Buffer for activations (96Kx256x8b = 24 MiB) *29% of chip*
Matrix multiply unit (256x256x8b = 64K MAC) *24%*
D R A M port ddr3 *3%*
Host Interf. *2%*
Accumulators (4Kx256x32b = 4 MiB) *6%*
D R A M port ddr3 *3%*
Control *2%*
Activation pipeline *6%*
PCIe Interface *3%*
Misc. I/O *1%*



The TPU and the Guidelines

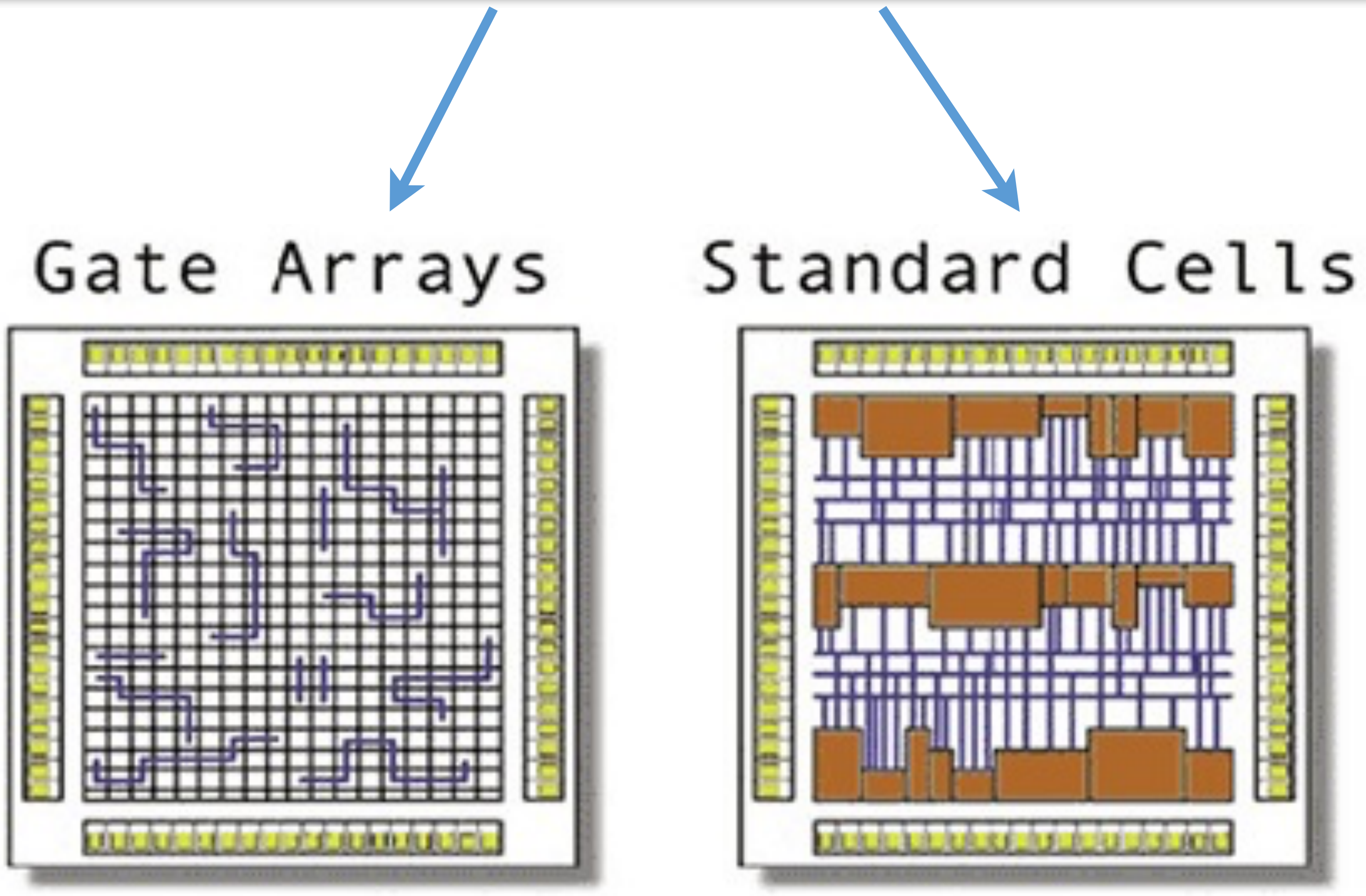
- Use dedicated memories
  - 24 MiB dedicated buffer, 4 MiB accumulator buffers
- Invest resources in arithmetic units and dedicated memories
  - 60% of the memory and 250X the arithmetic units of a server-class CPU
- Use the easiest form of parallelism that matches the domain
  - Exploits 2D SIMD parallelism
- Reduce the data size and type needed for the domain
  - Primarily uses 8-bit integers
- Use a domain-specific programming language
  - Uses TensorFlow

FPGA vs ASIC



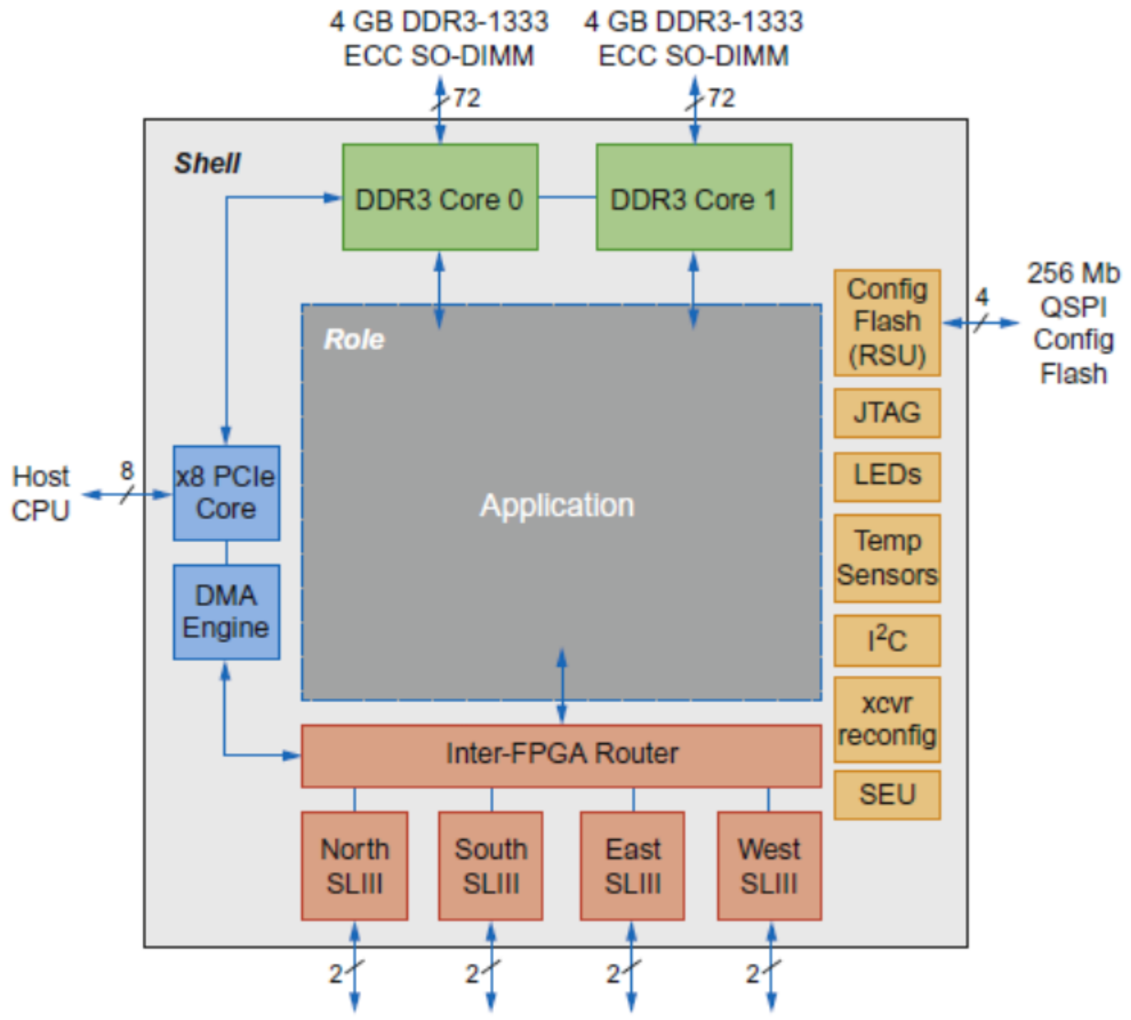
FPGA	ASIC
Reconfigurable circuitry after manufacturing	Fixed circuitry for product's lifespan
Suitable for digital designs only	Analog/mixed-signal circuitry can be fully implemented
Can be purchased as off-the-shelf products	Can only be designed as custom, private-label devices
Low-performance efficiencies, higher power consumption	Low power consumption, high-performance efficiencies
No non-recurring engineering (NRE) costs	NRE costs are part of the design process
Difficult to attain high-frequency rates	Operate at higher frequency rates
Faster time-to-market, high per unit costs	Long time-to-market, lower per unit costs
Are typically larger than ASICs	Can be much smaller than FPGA devices
Prototyping and validating with FPGAs is easier	Prototypes must be accurately validated to avoid design iterations
Lower barrier to entry for competitors	Higher barrier to entry for competitors

FPGA vs ASIC

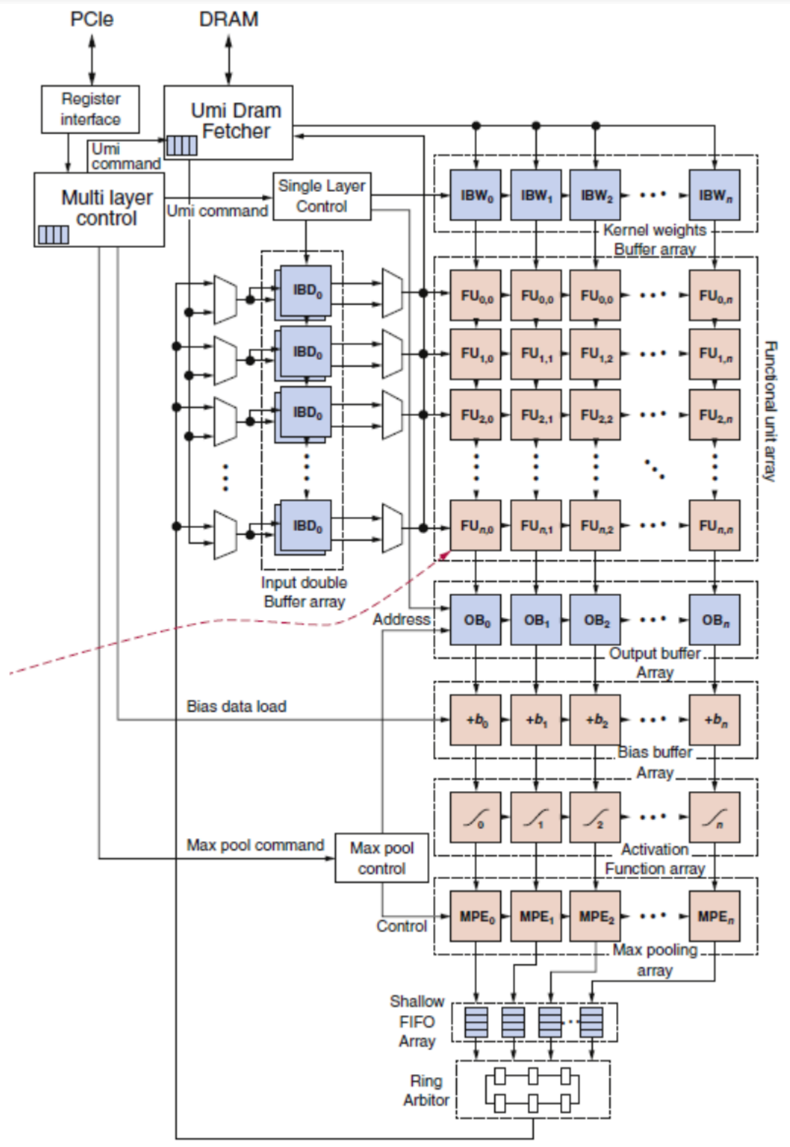


Microsoft Catapult

- **Needed to be general purpose and power efficient**
  - Uses FPGA PCIe board with dedicated 20 Gbps network in 6 x 8 torus
  - Each of the 48 servers in half the rack has a Catapult board
  - Limited to 25 watts
  - 32 MiB Flash memory
  - Two banks of DDR3-1600 (11 GB/s) and 8 GiB DRAM
  - FPGA (unconfigured) has 3962 18-bit ALUs and 5 MiB of on-chip memory
  - Programmed in Verilog RTL
  - Shell is 23% of the FPGA

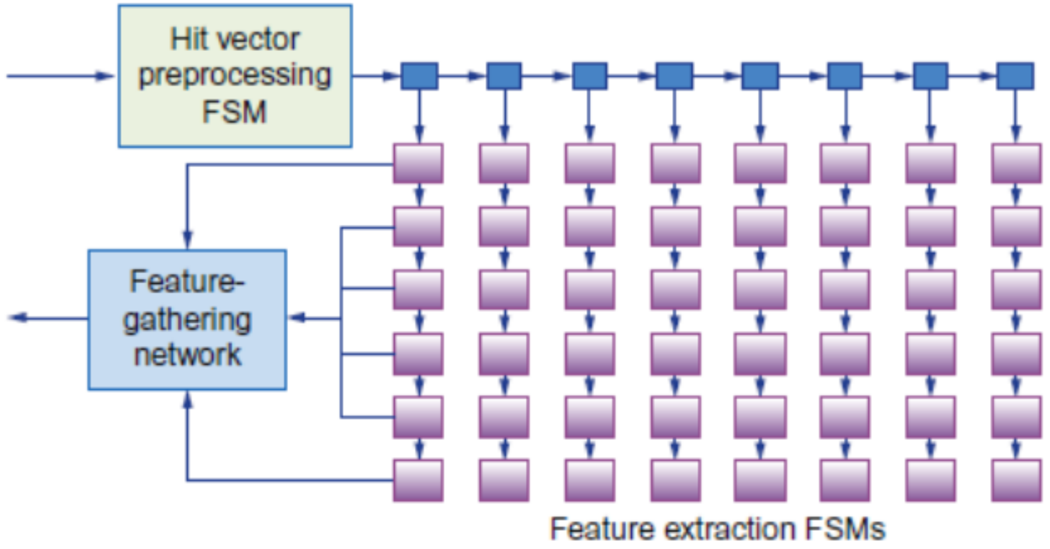


Microsoft Catapult: CNN configuration

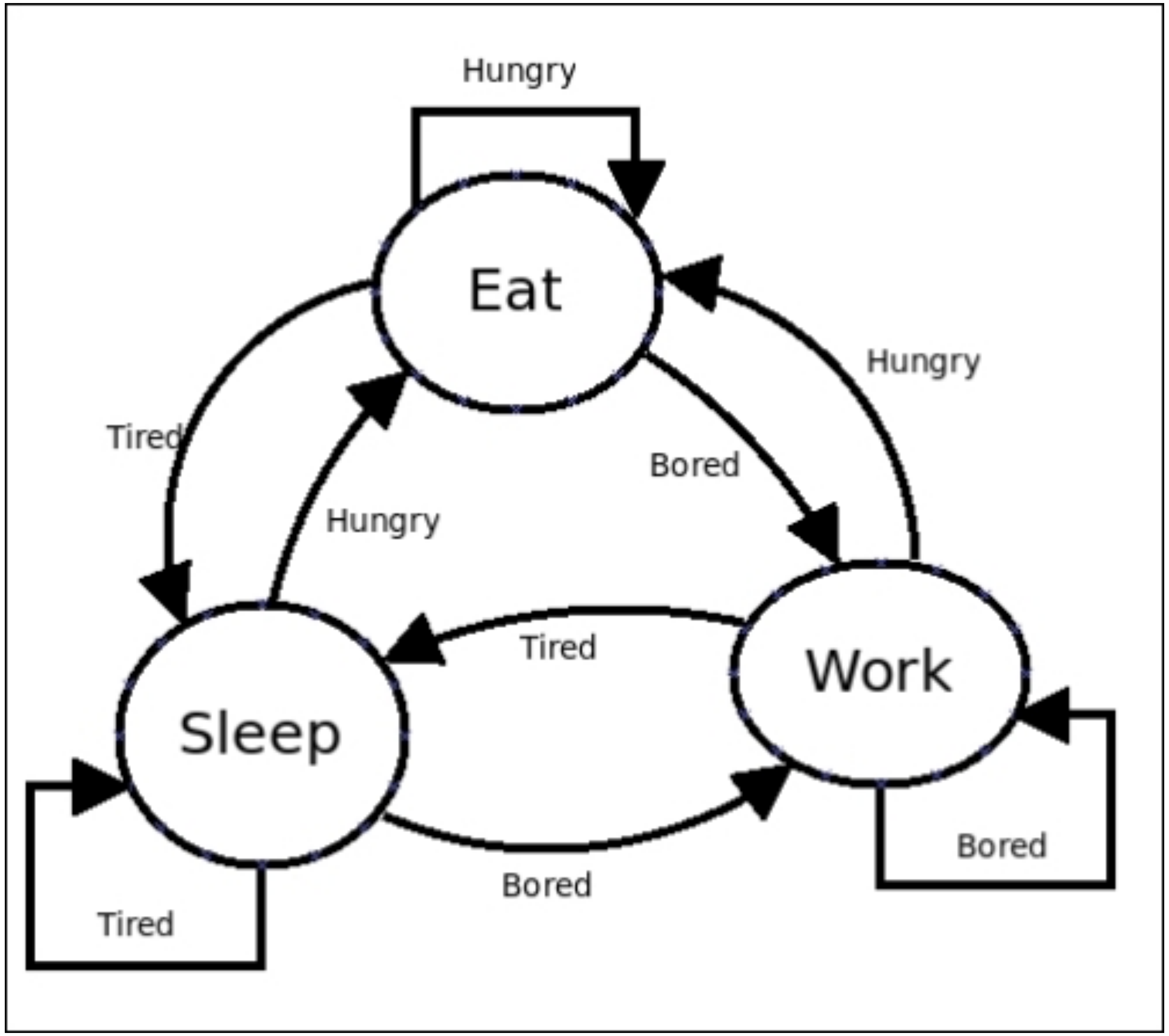

PCIe
DRAM
Register interface
Umi Dram Fetcher
Umi command
Multi layer control
Single Layer Control
Umi command
IBW$_0$ IBW$_1$ IBW$_2$ IBW$_n$
Kernel weights Buffer array
IBD$_0$
Input double Buffer array
FU$_{0,0}$ FU$_{1,0}$ FU$_{2,0}$ FU$_{n,0}$
FU$_{0,n}$ FU$_{1,n}$ FU$_{2,n}$ FU$_{n,n}$
Functional unit array
Address
OB$_0$ OB$_1$ OB$_2$ OB$_n$
Output buffer Array
Bias data load
$+b_0$ $+b_1$ $+b_2$ $+b_n$
Bias buffer Array
Activation Function array
Max pool command
Max pool control
Control
MPE$_0$ MPE$_1$ MPE$_2$ MPE$_n$
Max pooling array
Shallow FIFO Array
Ring Arbitor

Microsoft Catapult: Search Ranking Configuration

- Feature extraction (1 FPGA)
  - Extracts 4500 features for every document-query pair, e.g. frequency in which the query appears in the page
  - Systolic array of FSMs
- Free-form expressions (2 FPGAs)
  - Calculates feature combinations
- Machine-learned Scoring (1 FPGA for compression, 3 FPGAs calculate score)
  - Uses results of previous two stages to calculate floating-point score



FSM Example


Hungry
Eat
Hungry
Tired
Bored
Hungry
Work
Tired
Sleep
Bored
Tired
Bored

Microsoft Catapult and Guidelines

- Use dedicated memories
  - 5 MiB dedicated memory
- Invest resources in arithmetic units and dedicated memories
  - 3926 ALUs
- Use the easiest form of parallelism that matches the domain
  - 2D SIMD for CNN, MISD parallelism for search scoring
- Reduce the data size and type needed for the domain
  - Uses mixture of 8-bit integers and 64-bit floating-point
- Use a domain-specific programming language
  - Uses Verilog RTL; Microsoft did not follow this guideline

Any Questions?

```
            .text
__start:    addi t1, zero, 0x18
            addi t2, zero, 0x21
cycle:      beq t1, t2, done
            slt t0, t1, t2
            bne t0, zero, if_less
            nop
            sub t1, t1, t2
            j cycle
            nop
if_less:    sub t2, t2, t1
            j cycle
done:            add t3, t1, zero
```