http://inst.eecs.berkeley.edu/~ee290-2/sp21/

Hardware for Machine Learning
Lecture 7: DNN Accelerators

Sophia Shao

Early Accelerators: Math Coprocessor

The 80387 (387 or i387) is the first Intel coprocessor to be fully compliant with the IEEE 754-1985 standard. Without a coprocessor, the 386 normally performs floating-point arithmetic through (relatively slow) software routines, implemented at runtime through a software exception handler.

“Hello everybody out there using minix -

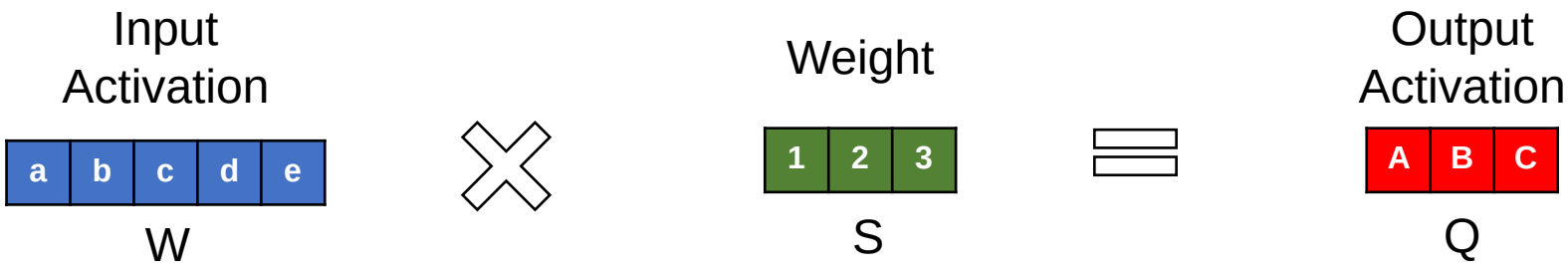
I'm doing a (free) operating system (*just a hobby, won't be big and professional like gnu*) for **386(486)** AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).”

— Linus Torvalds

Review

- Lecture 5: core computation in DNN
- Lecture 6: execution order of the core computation
  - Core principles:
    - Locality and Parallelism
  - Dataflow
    - Defines the execution order of DNNs
    - Represented using a loop nest
      - for: temporal order
      - spatial_for: spatial order/parallelism
    - Output-stationary and weight-stationary dataflow
- This lecture: hardware realization of the core computation

1D Convolution Example



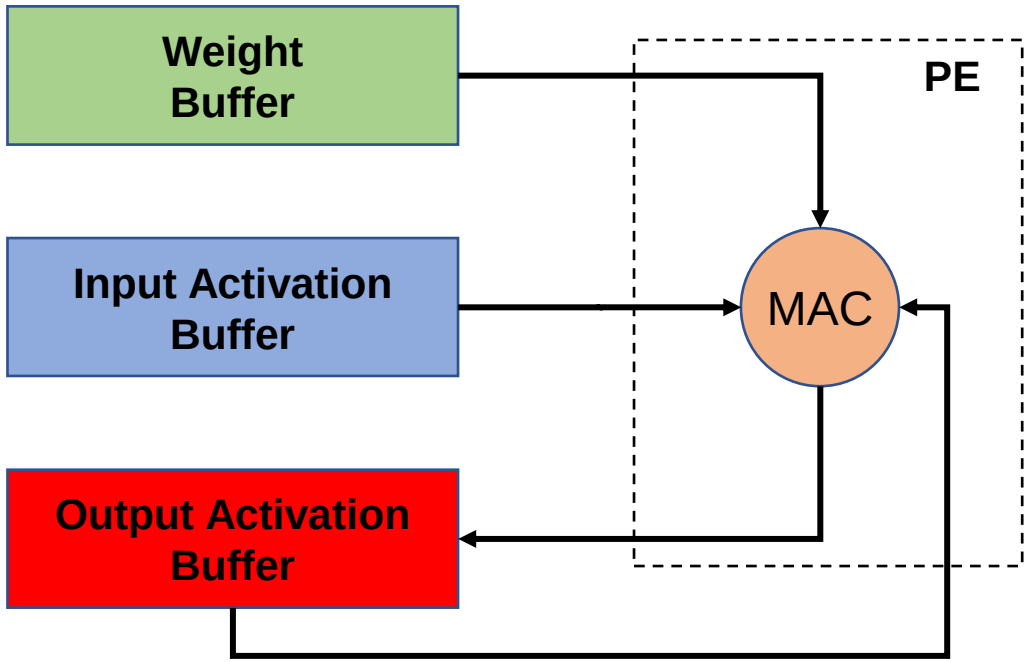

```
for(q=0; q<Q; q++){
  for (s=0; s<S; s++){
    OA[q] += IA[q+s] * W[s];
  }
}
```

Output Stationary (OS) Dataflow

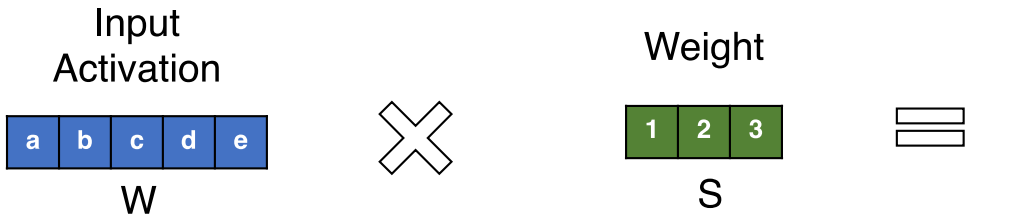
```
for (s=0; s<S; s++){
  for(q=0; q<Q; q++){
    OA[q] += IA[q+s] * W[s];
  }
}
```

Weight Stationary (WS) Dataflow

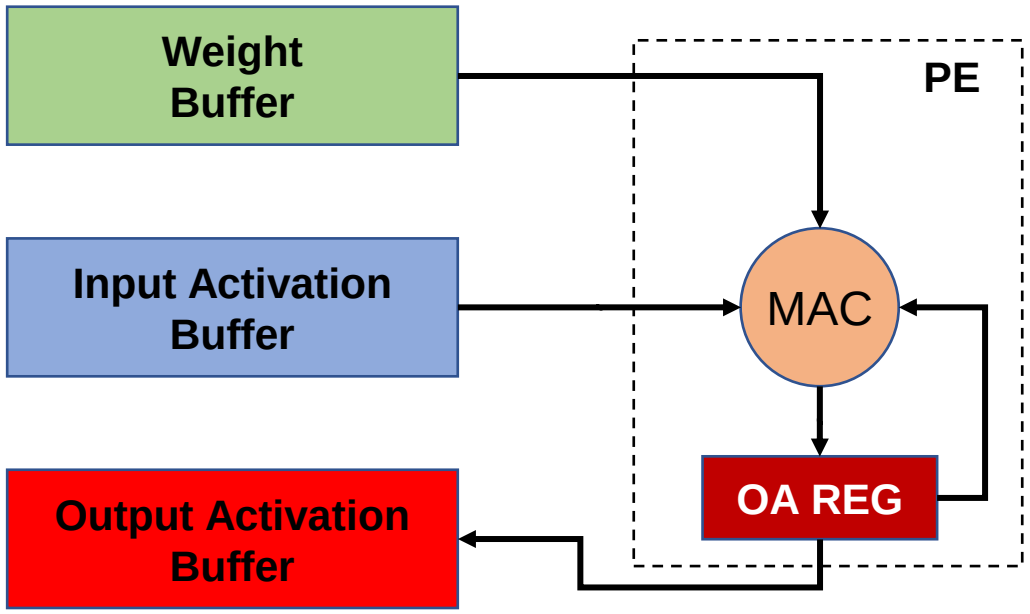
Single Processing Element (PE) Setup

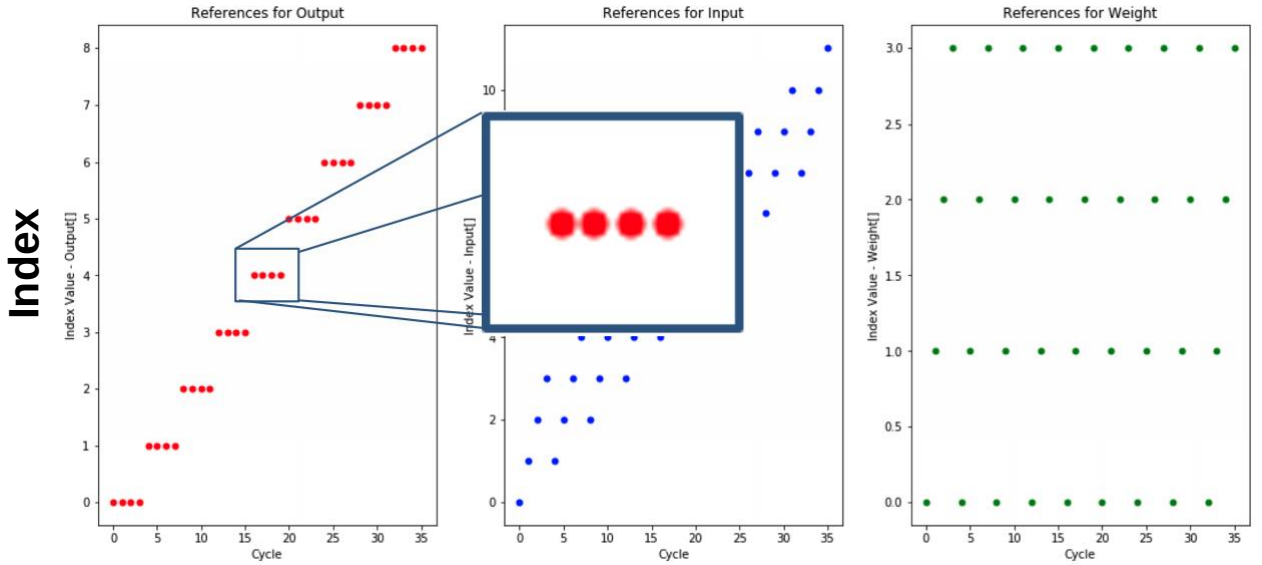


Buffer Access Pattern (OS)




```
for(q=0; q<Q; q++){ // Q =9
  for (s=0; s<S; s++){ // S=4
    OA[q] += IA[q+s] * W[s];
  }
}
```

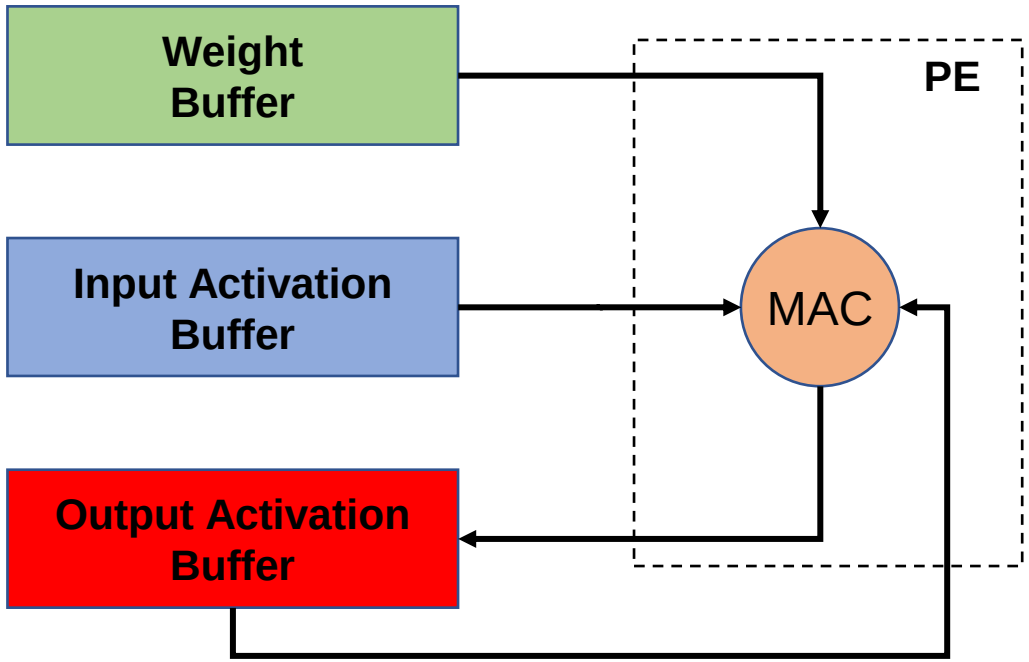




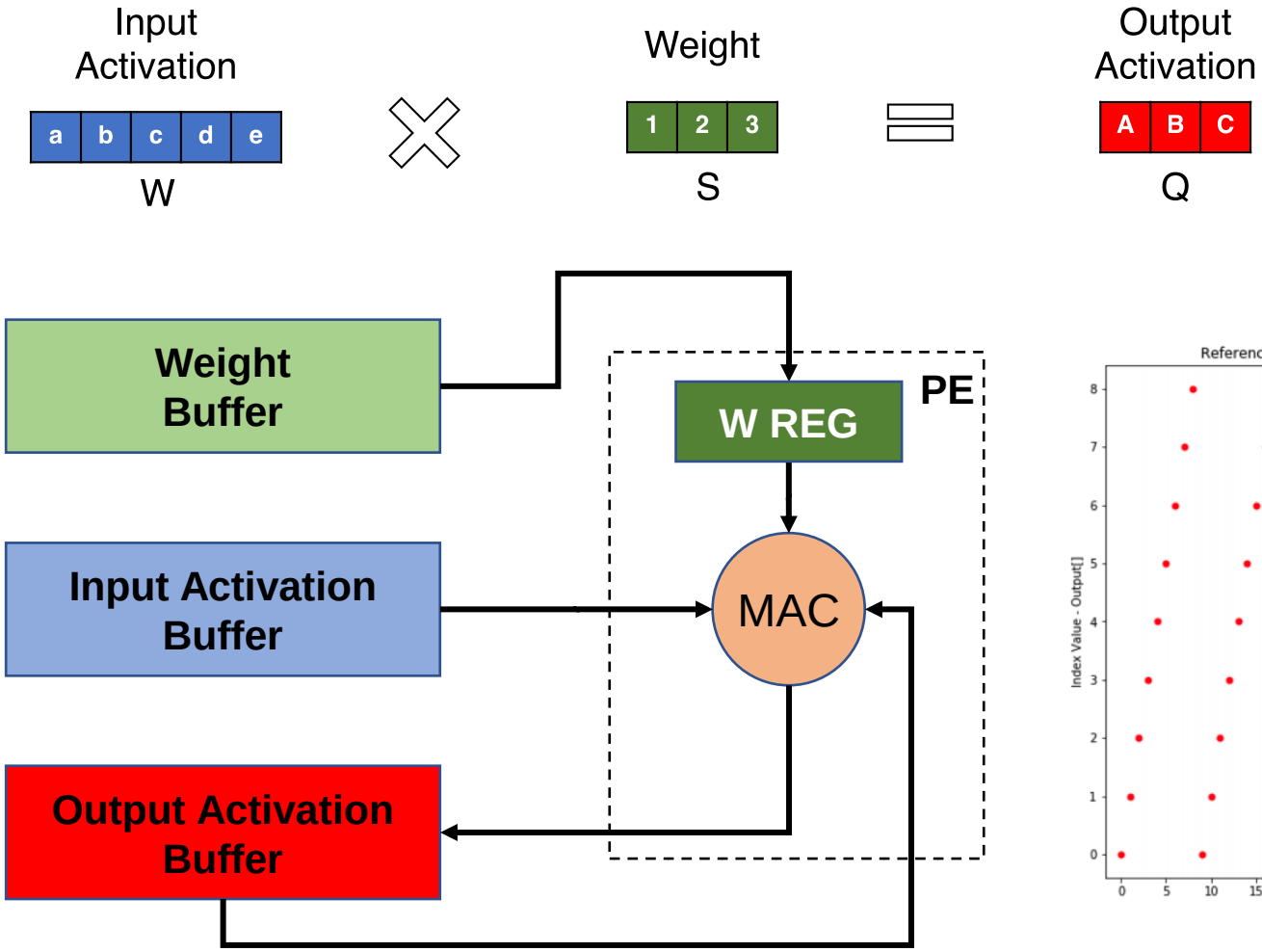
Eyeriss Tutorial

Observation: Single output is reused S times.

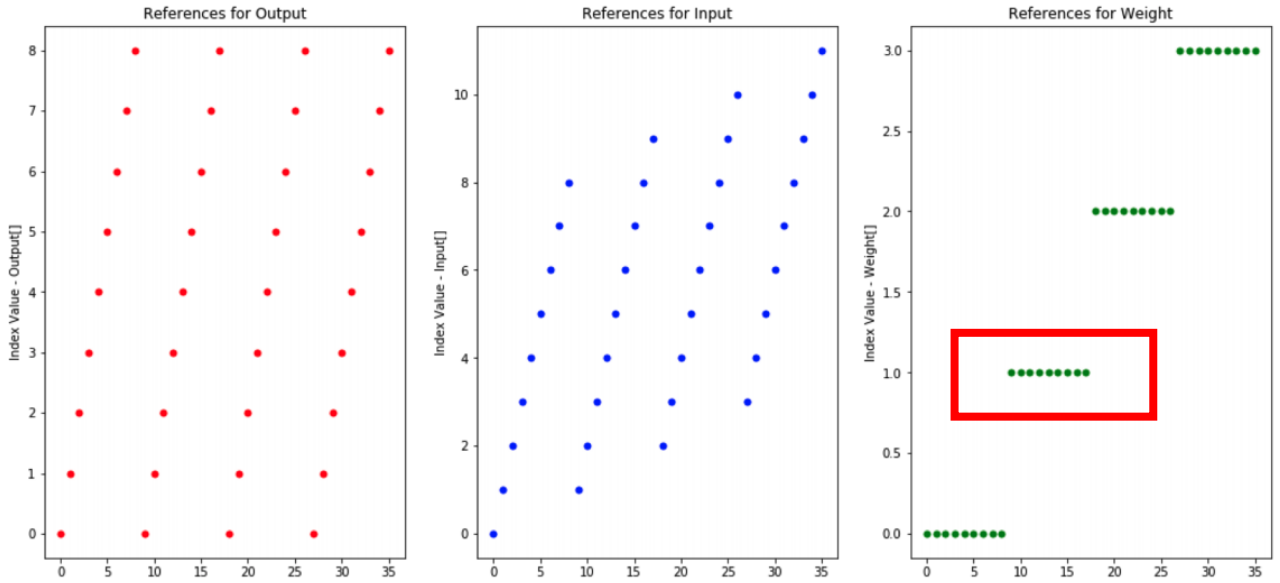
Single Processing Element (PE) Setup



Buffer Access Pattern (WS)

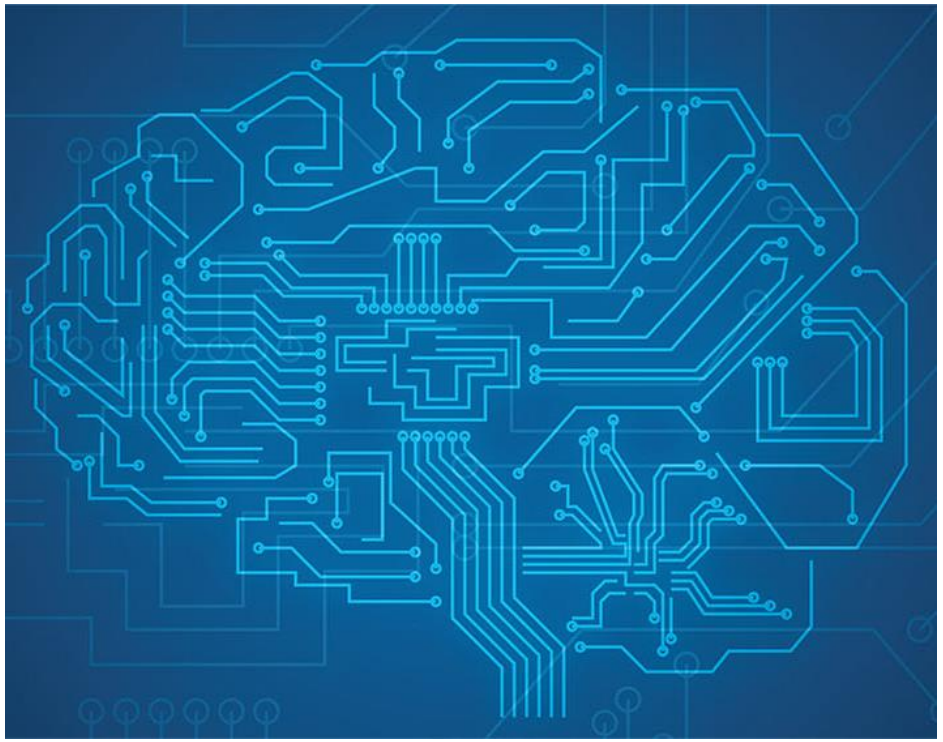



```
for (s=0; s<S; s++){// S=4
  for(q=0; q<Q; q++){// Q =9
    OA[q] += IA[q+s] * W[s];
  }
}
```



Eyeriss Tutorial

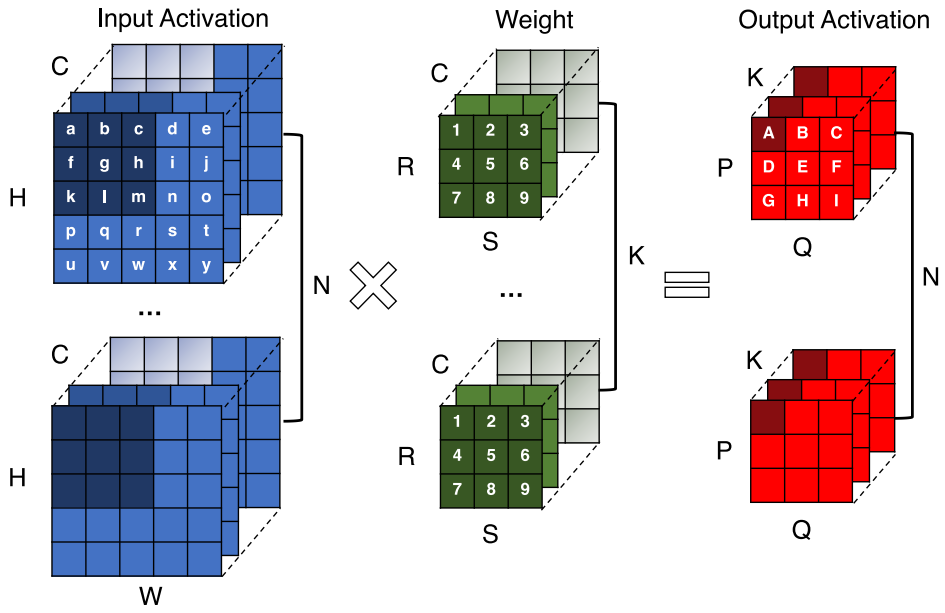
Observation: Single weight is reused Q times.



Accelerators

- Source of Inefficiency
- Core Optimizations:
  - Inst. decoding logic
  - Datapath
  - Memory system
- Putting everything together
  - Google's TPU

Convolution Loop Nest

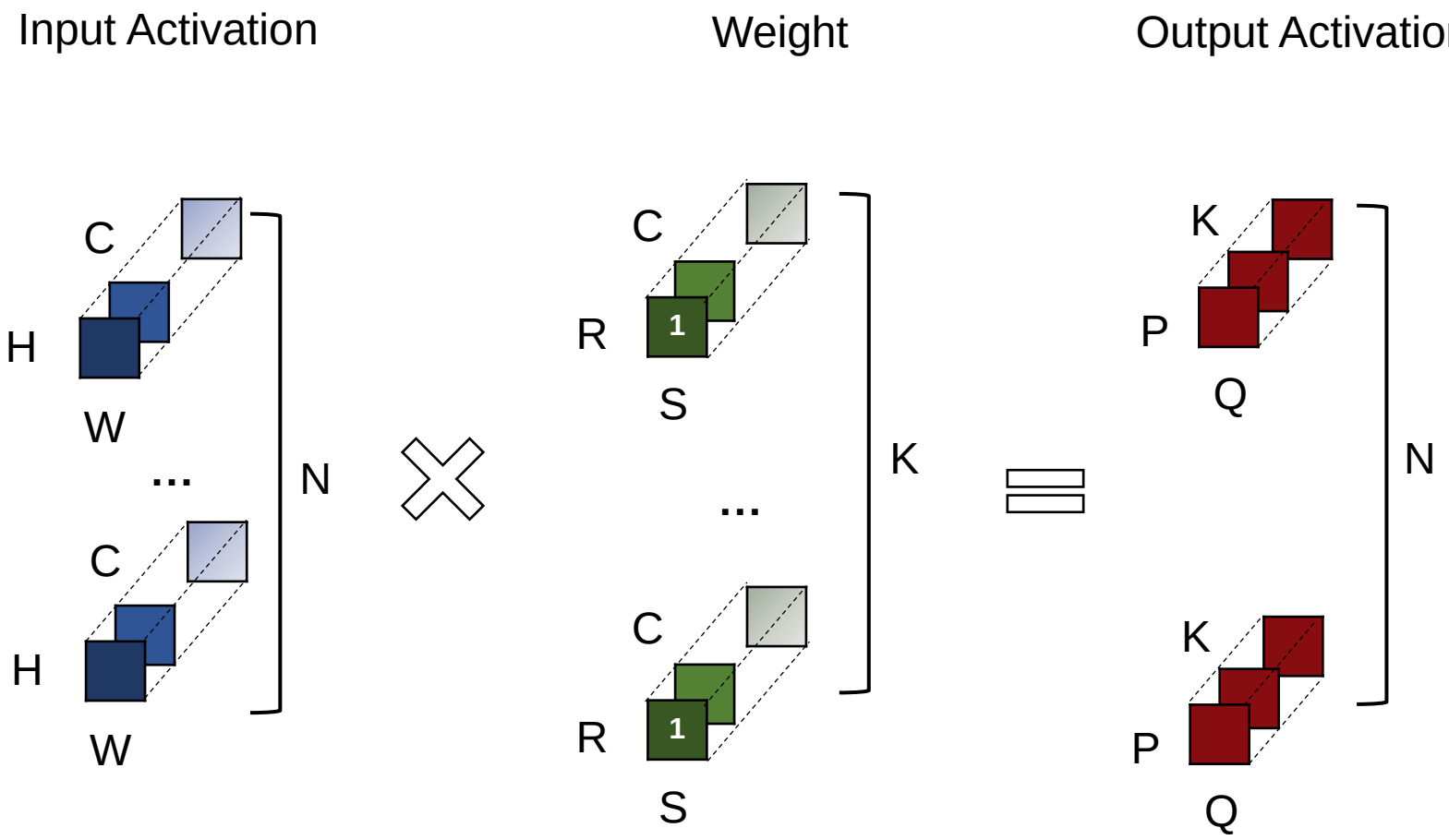



```
for (n=0; n<N; n++) {
  for (k=0; k<K; k++) {
    for (p=0; p<P; p++) {
      for (q=0; q<Q; q++) {
        OA[n][k][p][q]= 0;
        for (r=0; r<R; r++) {
          for (s=0; s<S; s++) {
            for (c=0; c<C; c++) {
              h = p * stride - pad + r;
              w = q * stride - pad + s;
              OA[n][k][p][q] +=
                          IA[n][c][h][w]
                          * W[k][c][r][s];
            }
          }
        }
        OA[n][k][p][q]= Activation(OA[n][k][p][q]);
      }
    }
  }
}
```

for each output activation

convolution window

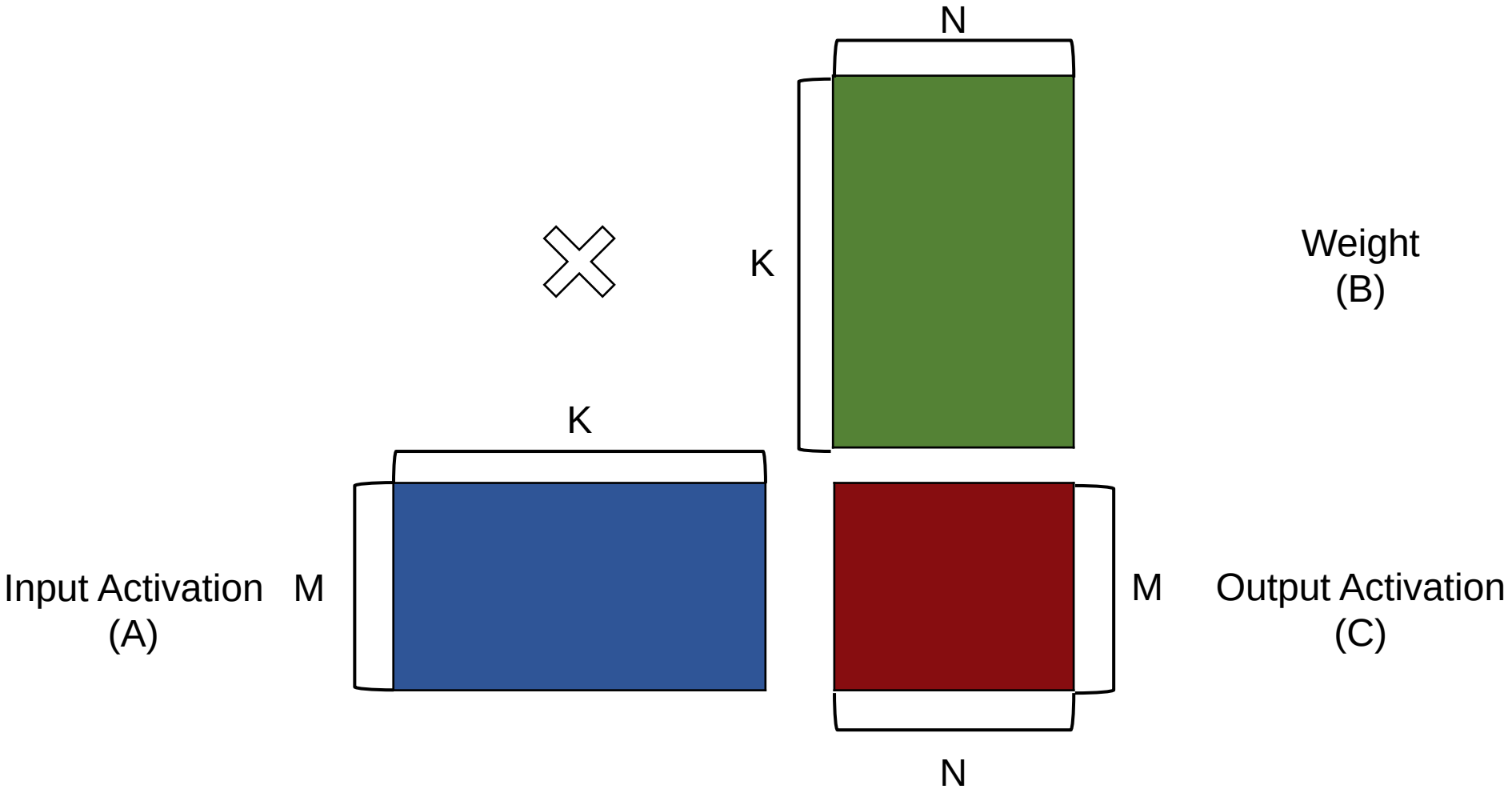
Fully-Connected Layer



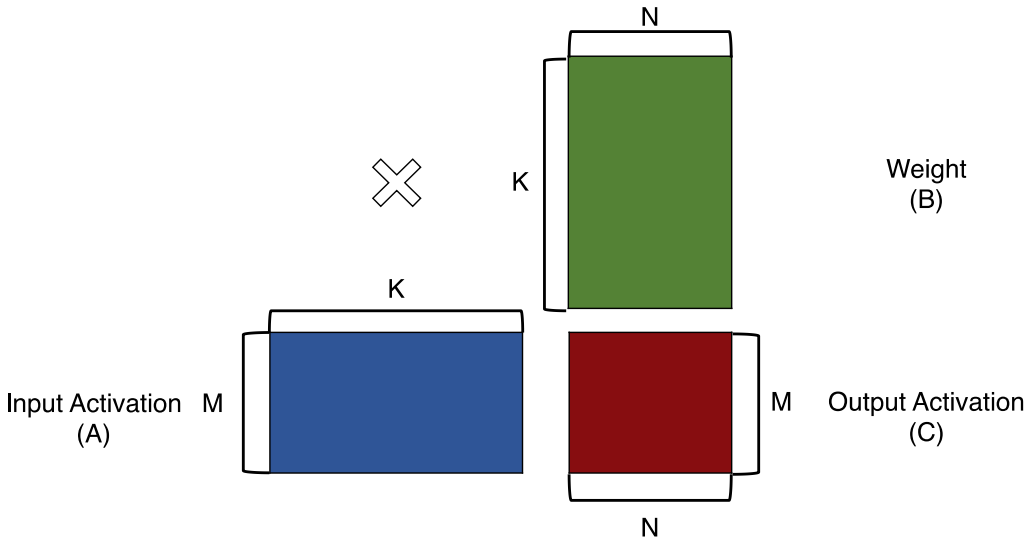
H = 1
W = 1
R = 1
S = 1
P = 1
Q = 1
stride = 1
padding= 0

C: # of Input Channels
K: # of Output Channels
N: Batch size

Matrix Multiply



Matrix Multiply

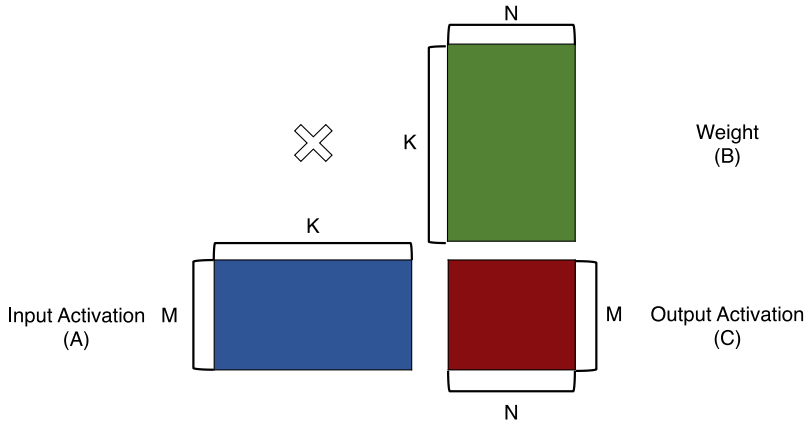



```
for (m=0; m<M; m++) {
  for (n=0; n<N; n++) {
    OA[n,m] = 0;
    for (k=0; k<K; k++) {
      OA[n,m] += IA[m, k]
                 * W[k, n];
    }
    OA[n,m] = Activation(OA[n,m]);
  }
}
```

for each output activation

convolution window

Matrix Multiply




```
for (m=0; m<M; m++) {
  for (n=0; n<N; n++) {
    OA[n,m] = 0;
    for (k=0; k<K; k++) {
      OA[n,m] += IA[m, k]
                 * W[k, n];
    }
    OA[n,m] = Activation(OA[n,m]);
  }
}
```

```
10000170:→ f00c0993          → addi→   x19,x24,-256
10000174:→ 00004a97          → auipc→  x21,0x4
10000178:→ 674a8a93          → addi→   x21,x21,1652 # 100047e8 <B>
1000017c:→ 000a8913          → mv→ x18,x21
10000180:→ 000b8413          → mv→ x8,x23
10000184:→ 00000493          → li→ x9,0
10000188:→ 00092583          → lw→ x11,0(x18)
1000018c:→ 00042503          → lw→ x10,0(x8)
10000190:→ 00440413          → addi→   x8,x8,4
10000194:→ 10090913          → addi→   x18,x18,256
10000198:→ f25ff0ef          → jal→x1,100000bc <times>
1000019c:→ 00a484b3          → add→x9,x9,x10
100001a0:→ 0099a023          → sw→ x9,0(x19)
100001a4:→ ff4412e3          → bne→x8,x20,10000188 <mmult+0x6c>
100001a8:→ 00498993          → addi→   x19,x19,4
100001ac:→ 009b0b33          → add→x22,x22,x9
100001b0:→ 004a8a93          → addi→   x21,x21,4
100001b4:→ fd8994e3          → bne→x19,x24,1000017c <mmult+0x60>
100001b8:→ 10040a13          → addi→   x20,x8,256
100001bc:→ 10098c13          → addi→   x24,x19,256
100001c0:→ 100b8b93          → addi→   x23,x23,256
100001c4:→ fb9a16e3          → bne→x20,x25,10000170 <mmult+0x54>
```

Source of Inefficiency in CPUs

```
10000170:→  f00c0993        → addi→   x19,x24,-256
10000174:→  00004a97        → auipc→  x21,0x4
10000178:→  674a8a93        → addi→   x21,x21,1652 # 100047e8 <B>
1000017c:→  000a8913        → mv→ x18,x21
10000180:→  000b8413        → mv→ x8,x23
10000184:→  00000493        → li→ x9,0
10000188:→  00092583        → lw→ x11,0(x18)
1000018c:→  00042503        → lw→ x10,0(x8)
10000190:→  00440413        → addi→   x8,x8,4
10000194:→  10090913        → addi→   x18,x18,256
10000198:→  f25ff0ef        → jal→x1,100000bc <times>
1000019c:→  00a484b3        → add→x9,x9,x10
100001a0:→  0099a023        → sw→ x9,0(x19)
100001a4:→  ff4412e3        → bne→x8,x20,10000188 <mmult+0x6c>
100001a8:→  00498993        → addi→   x19,x19,4
100001ac:→  009b0b33        → add→x22,x22,x9
100001b0:→  004a8a93        → addi→   x21,x21,4
100001b4:→  fd8994e3        → bne→x19,x24,1000017c <mmult+0x60>
100001b8:→  10040a13        → addi→   x20,x8,256
100001bc:→  10098c13        → addi→   x24,x19,256
100001c0:→  100b8b93        → addi→   x23,x23,256
100001c4:→  fb9a16e3        → bne→x20,x25,10000170 <mmult+0x54>
```

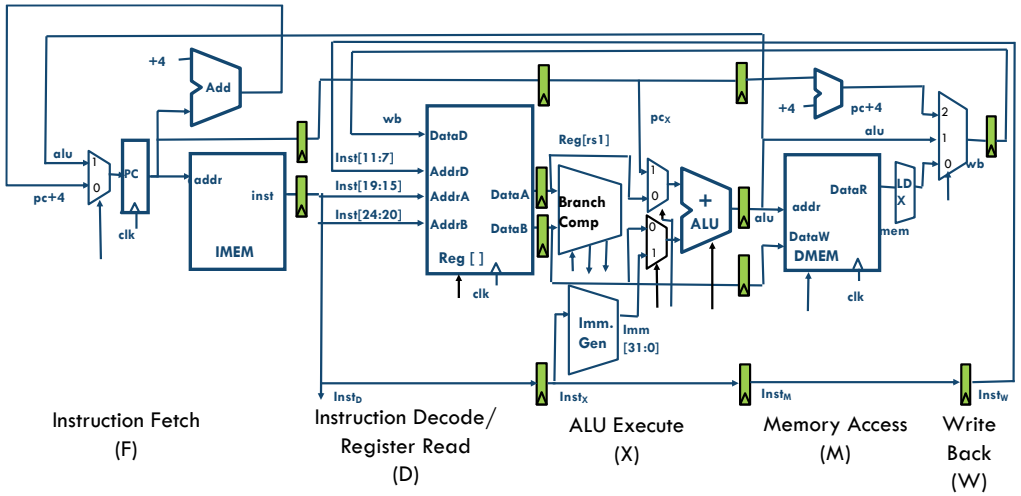
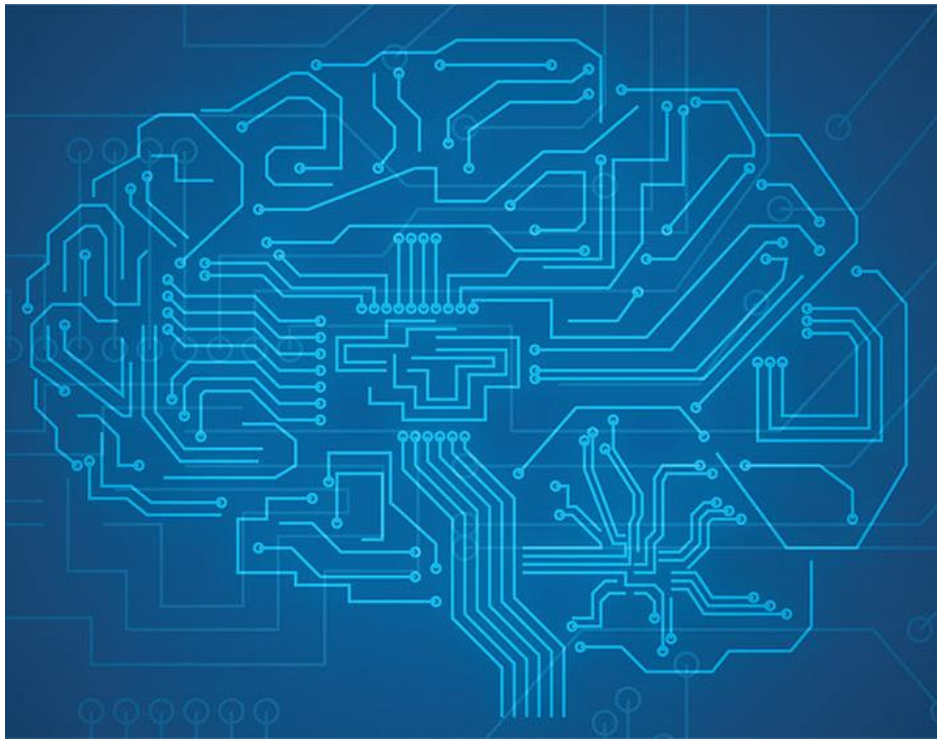


Table 3. Datapath energy breakdown for our base implementation in mJ/frame. IF is instruction fetch/decode (including the I-cache). D-$ is the D-cache. Pip is the pipeline registers, buses, and clocking. Ctl is random control. RF is the register file. FU is the functional elements. Data estimates from processor simulations.

	IF	D-$	Pip	Ctl	RF	FU	Total
IME	410	218	257	113	113	68	1179
FME	286	196	205	90	90	54	921
Intra	54	20	29	13	13	8	137
CABAC	12	2	8	4	4	2	32
Total	762	436	499	220	220	132	2269

Understanding Sources of Inefficiency in General-Purpose Chips, ISCA'2020



Accelerators

- Source of Inefficiency
- Core Optimizations:
  - Inst. decoding logic
  - Datapath
  - Memory system
- Putting everything together
  - Google's TPU

Inst. Decoding Logic Optimization

- Coarse-Grained, Domain-Specific Instructions
  - Key instructions: data movement and compute
- Example: TPU ISAs

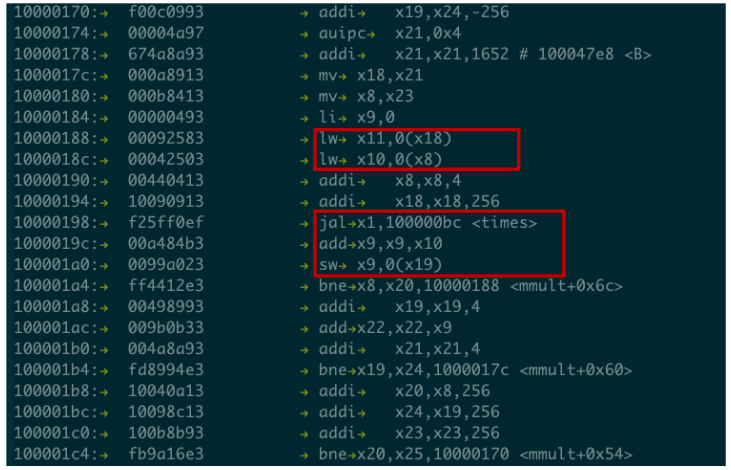
```
10000170:→  f00c0993          → addi→   x19,x24,-256
10000174:→  00004a97          → auipc→  x21,0x4
10000178:→  674a8a93          → addi→   x21,x21,1652 # 100047e8 <B>
1000017c:→  000a8913          → mv→ x18,x21
10000180:→  000b8413          → mv→ x8,x23
10000184:→  00000493          → li→ x9,0
10000188:→  00092583          → lw→ x11,0(x18)
1000018c:→  00042503          → lw→ x10,0(x8)
10000190:→  00440413          → addi→   x8,x8,4
10000194:→  10090913          → addi→   x18,x18,256
10000198:→  f25ff0ef          → jal→x1,100000bc <times>
1000019c:→  00a484b3          → add→x9,x9,x10
100001a0:→  0099a023          → sw→ x9,0(x19)
100001a4:→  ff4412e3          → bne→x8,x20,10000188 <mmult+0x6c>
100001a8:→  00498993          → addi→   x19,x19,4
100001ac:→  009b0b33          → add→x22,x22,x9
100001b0:→  004a8a93          → addi→   x21,x21,4
100001b4:→  fd8994e3          → bne→x19,x24,1000017c <mmult+0x60>
100001b8:→  10040a13          → addi→   x20,x8,256
100001bc:→  10098c13          → addi→   x24,x19,256
100001c0:→  100b8b93          → addi→   x23,x23,256
100001c4:→  fb9a16e3          → bne→x20,x25,10000170 <mmult+0x54>
```

1. `Read_Host_Memory` reads data from the CPU host memory into the Unified Buffer (UB).
2. `Read_Weights` reads weights from Weight Memory into the Weight FIFO as input to the Matrix Unit.
3. `MatrixMultiply/Convolve` causes the Matrix Unit to perform a matrix multiply or a convolution from the Unified Buffer into the Accumulators. A matrix operation takes a variable-sized B*256 input, multiplies it by a 256x256 constant weight input, and produces a B*256 output, taking B pipelined cycles to complete.
4. `Activate` performs the nonlinear function of the artificial neuron, with options for ReLU, Sigmoid, and so on. Its inputs are the Accumulators, and its output is the Unified Buffer. It can also perform the pooling operations needed for convolutions using the dedicated hardware on the die, as it is connected to nonlinear function logic.
5. `Write_Host_Memory` writes data from the Unified Buffer into the CPU host memory.

In-Datacenter Performance Analysis of a Tensor Processing Unit, ISCA'2017

Inst. Decoding Logic Optimization

- Coarse-Grained, Domain-Specific Instructions
  - Key instructions: data movement and compute
- Example: Gemmini ISAs



- Data movement:
  - **mvin**: move data from L2/DRAM to scratchpad
  - **mvout**: move data from scratchpad to L2/DRAM
- Compute:
  - **matmul.preload**: preload weights for weight stationary
  - **matmul.compute.preloaded**: compute using preloaded values
  - **matmul.compute.accumulated**: compute using existing values
- Configurations:
  - **config_ex**: configures the execute pipeline
  - **config_mvin**: configures the load pipeline
  - **config_mout**: configures the store pipeline
  - **flush**: flush the TLB

https://github.com/ucb-bar/gemmini

Inst. Decoding Logic Optimization

- Operation fusion
  - Merging multiple operators together
  - Also commonly used in software optimization
- Example:
  - CRP (convolution-relu-pooling)

- Examples of patterns fused:
  - Conv2D + BiasAdd + <Activation>
  - Conv2D + FusedBatchNorm + <Activation>
  - Conv2D + Squeeze + BiasAdd
  - MatMul + BiasAdd + <Activation>

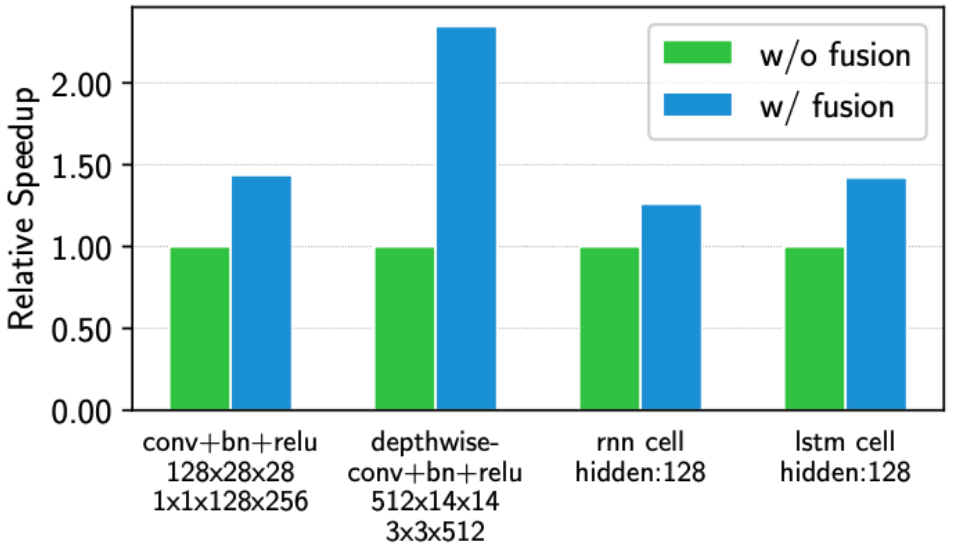
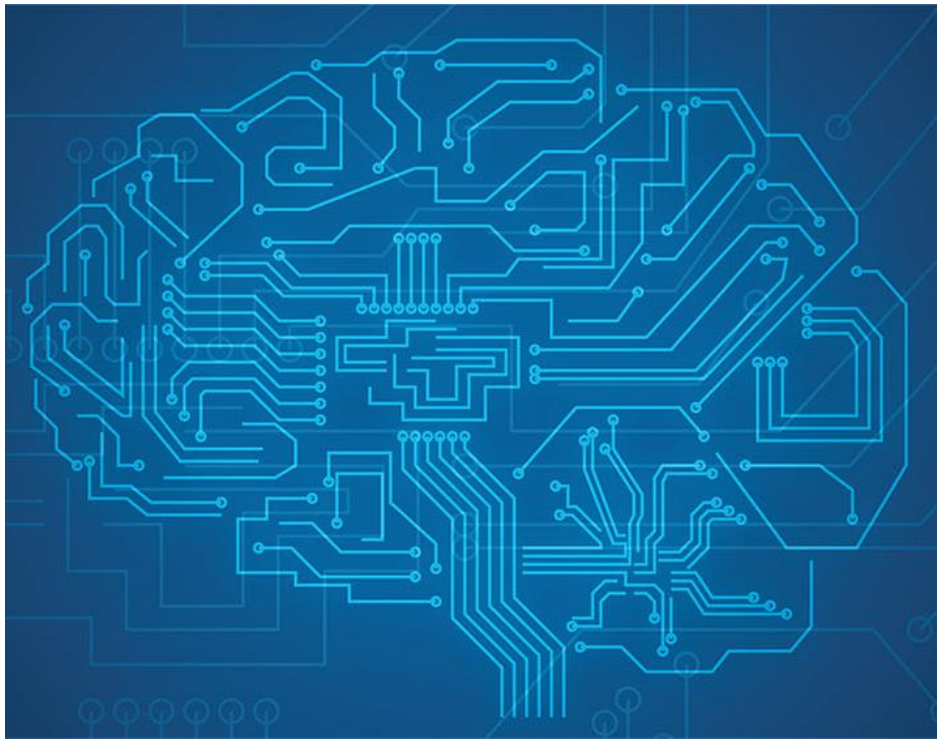


Figure 4: Performance comparison between fused and non-fused operations. TVM generates both operations. Tested on NVIDIA Titan X.



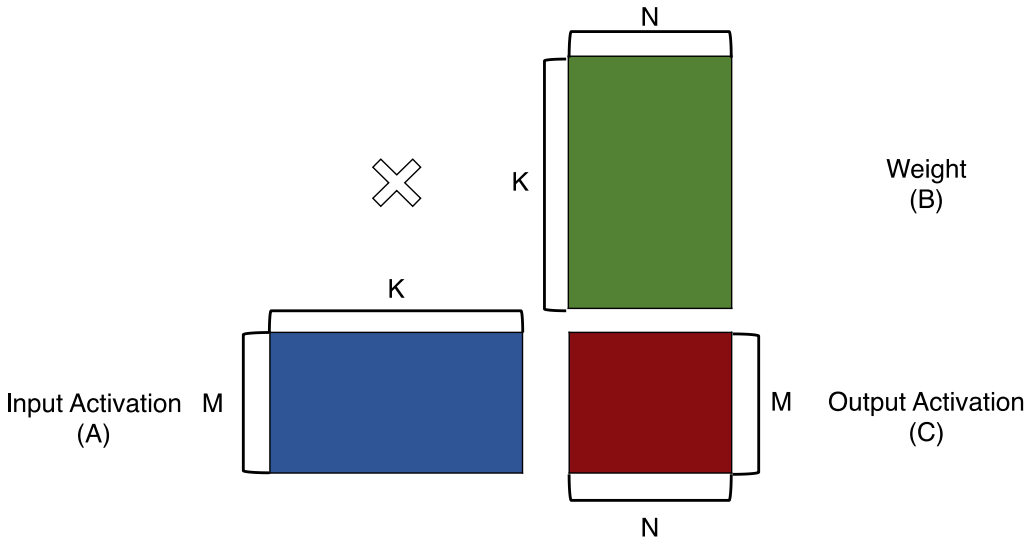
Accelerators

- Source of Inefficiency
- Core Optimizations:
  - Inst. decoding logic
  - Datapath
  - Memory system
- Putting everything together
  - Google's TPU

Dataflow Recap

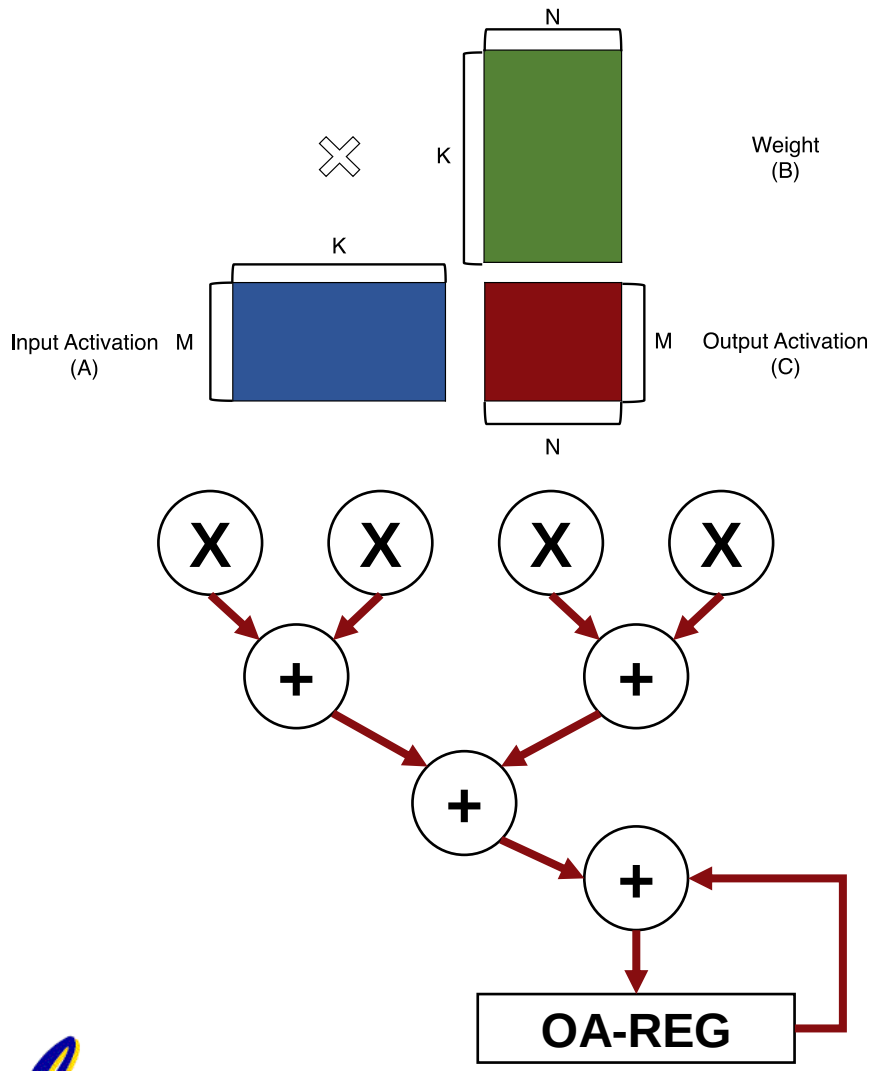
- Defines the execution order of the DNN operations in hardware.
  - Computation order
  - Data movement order
- Loop nest is a compact way to describe the execution order, i.e., dataflow, supported in hardware.
  - `for`: temporal for, describes the temporal execution order.
  - `spatial_for`: describes parallel execution.

Matrix Multiply Dataflow




```
for (m=0; m<M; m++) {
  for (n=0; n<N; n++) {
    OA[n,m] = 0;
    for (k=0; k<K; k++) {
      OA[n,m] += IA[m, k]
                 * W[k, n];
    }
    OA[n,m] = Activation(OA[n,m]);
  }
}
```

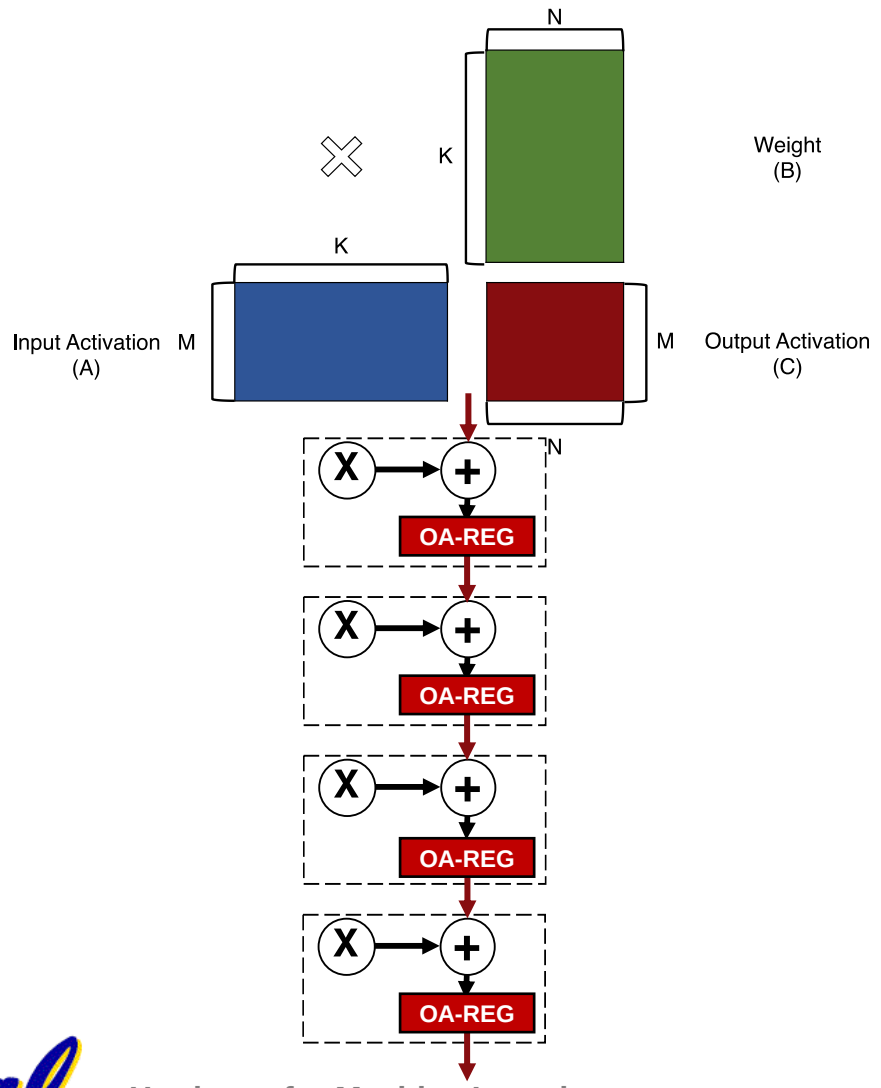
Datapath Optimization 1: Spatial-K




```
for (m=0; m<M; m++) {
  for (n=0; n<N; n++) {
    OA[n,m] = 0;
    spatial_for (k=0; k<K; k++) {
      OA[n,m] += IA[m, k]
                 * W[k, n];
    }
    OA[n,m] = Activation(OA[n,m]);
  }
}
```

- Type 1: Adder Tree
- Example: NVDLA, DianNao
- Typical width: 8-64
- Applicable to any accumulation dimensions
  - E.g., R, S, C in convolution

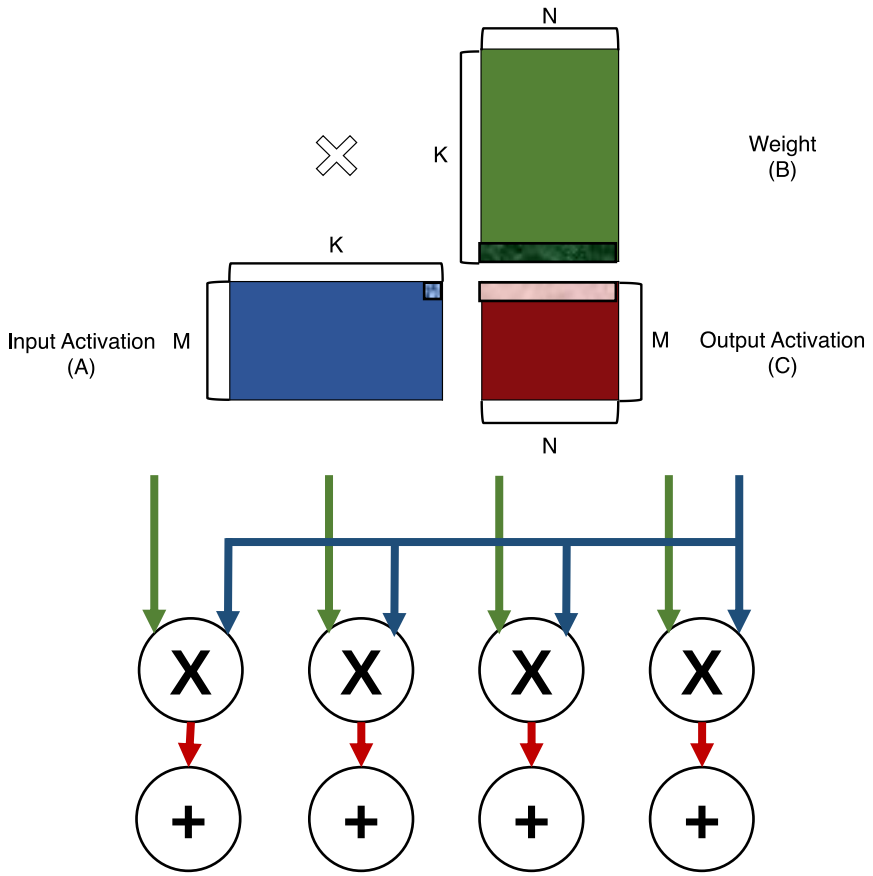
Datapath Optimization 1: Spatial-K




```
for (m=0; m<M; m++) {
  for (n=0; n<N; n++) {
    OA[n,m] = 0;
    spatial_for (k=0; k<K; k++) {
      OA[n,m] += IA[m, k]
                 * W[k, n];
    }
    OA[n,m] = Activation(OA[n,m]);
  }
}
```

- Type 2: Systolic Accumulation
- Example: TPU, Gemmini
- Typical width: 8-256
- Applicable to any accumulation dimensions
  - E.g., R, S, C in convolution

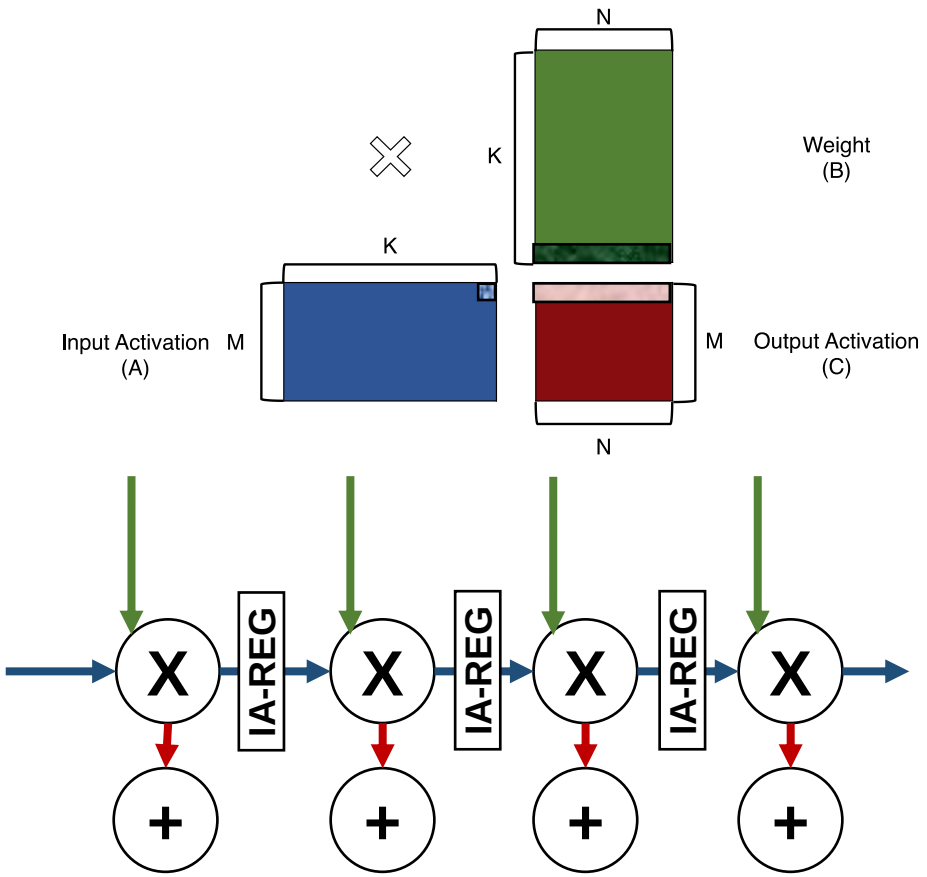
Datapath Optimization 2: Spatial-N




```
for (m=0; m<M; m++) {
  spatial_for (n=0; n<N; n++) {
    OA[n,m] = 0;
    for (k=0; k<K; k++) {
      OA[n,m] += IA[m, k]
                 * W[k, n];
    }
    OA[n,m] = Activation(OA[n,m]);
  }
}
```

- Type 1: Direct-wiring multicast
- Example: NVDLA, DianNao
- Typical width: 8-16
- Applicable to any non-accumulation dimensions

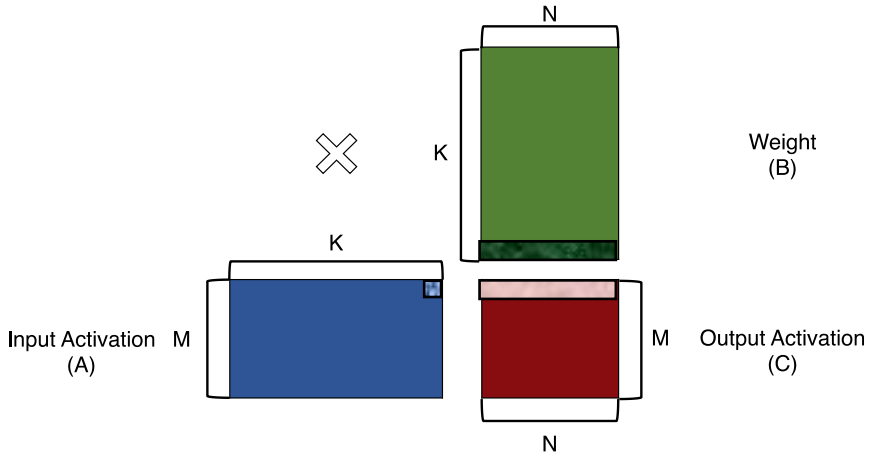
Datapath Optimization 2: Spatial-N



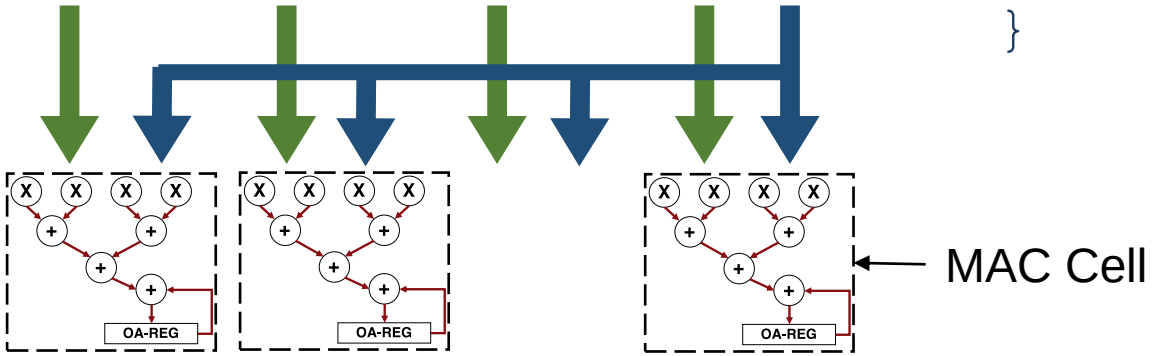

```
for (m=0; m<M; m++) {
  spatial_for (n=0; n<N; n++) {
    OA[n,m] = 0;
    for (k=0; k<K; k++) {
      OA[n,m] += IA[m, k]
                 * W[k, n];
    }
    OA[n,m] = Activation(OA[n,m]);
  }
}
```

- Type 2: Systolic multicast
- Example: TPU, Gemmini
- Typical width: 8-256
- Applicable to any non-accumulation dimensions

Datapath Optimization Combined: NVDLA

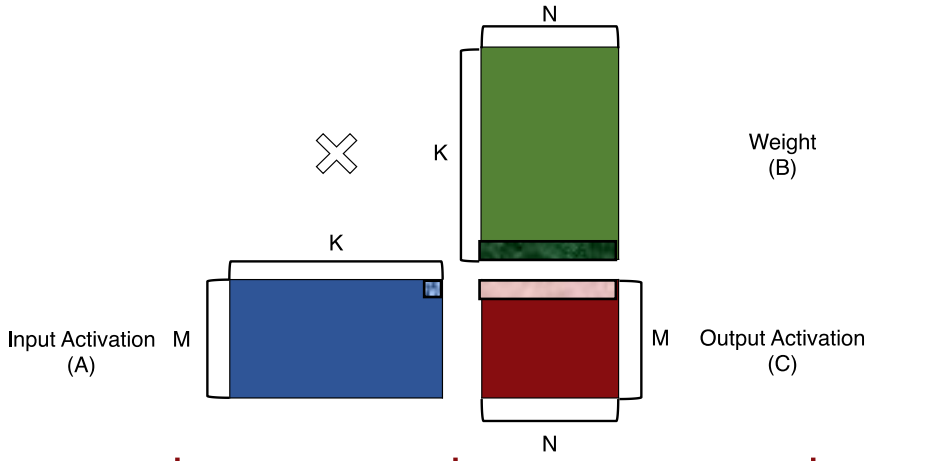


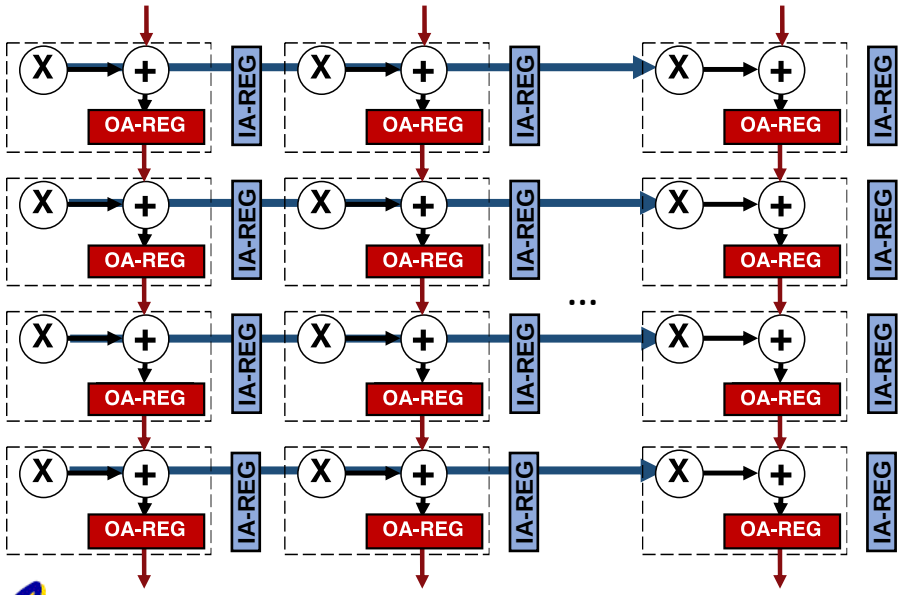

```
for (m=0; m<M; m++) {
  spatial_for (n=0; n<N; n++) {
    OA[n,m] = 0;
    spatial_for (k=0; k<K; k++) {
      OA[n,m] +=  IA[m, k]
                * W[k, n];
    }
    OA[n,m] = Activation(OA[n,m]);
  }
}
```



- Adder-tree accumulation
- Direct-wiring multicast

Datapath Optimization Combined: TPU






```
for (m=0; m<M; m++) {
  spatial_for (n=0; n<N; n++) {
    OA[n,m] = 0;
    spatial_for (k=0; k<K; k++) {
      OA[n,m] +=  IA[m, k]
                * W[k, n];
    }
    OA[n,m] = Activation(OA[n,m]);
  }
}
```

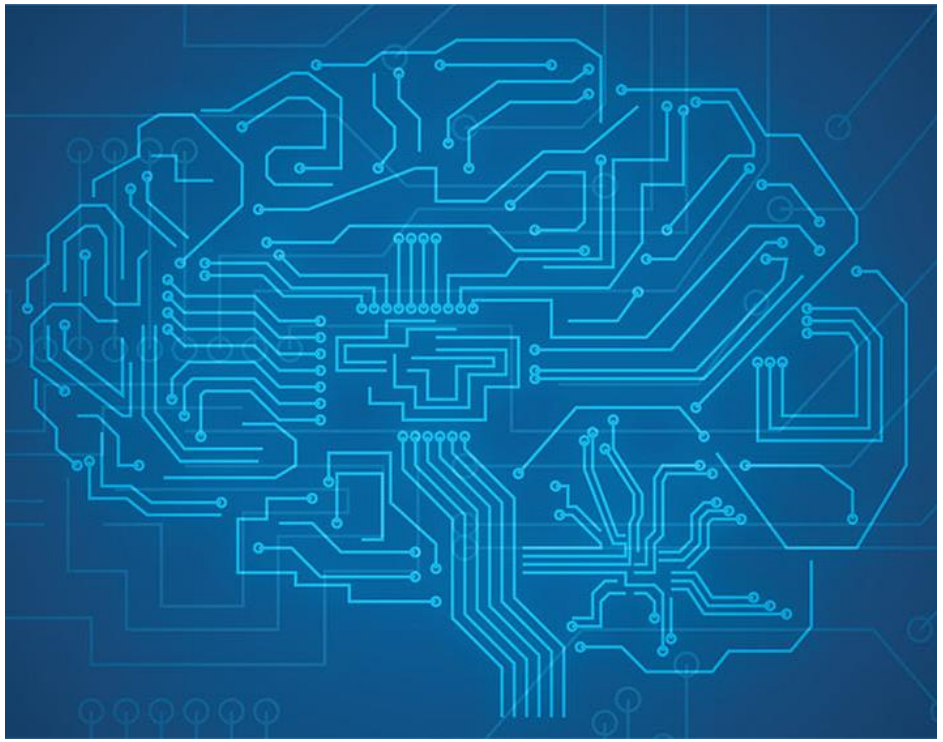
- Systolic accumulation
- Systolic multicast

Datapath Optimization Recap

- `spatial_for`: to explore parallelism
  - Spatial-K: accumulation dimension
    - Adder tree/dot product
    - Systolic accumulation
  - Spatial-M/N: data reuse dimension
    - Direct-wiring multicast
    - Systolic multicast
- State-of-the-art accelerators typically use a combination of both at different levels of the hierarchy.

Administrivia

- Lab 2 is posted.
  - Due 2/19.
- Project meet-up:
  - https://forms.gle/wcmsT7dJby3Q3DzH9
- No lecture 2/15.
  - Presidents' Day.
  - Enjoy!
- Guest Lecture 2/17: Advanced Quantization with DNN
  - Amir Gholami, UC Berkeley

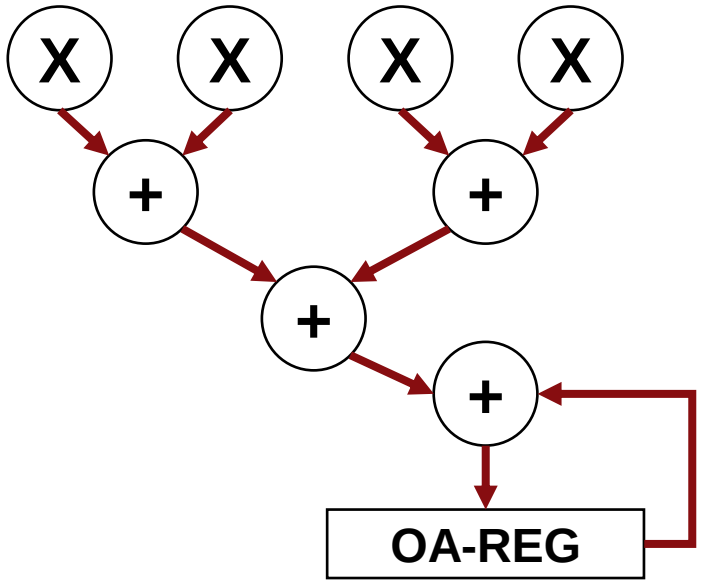


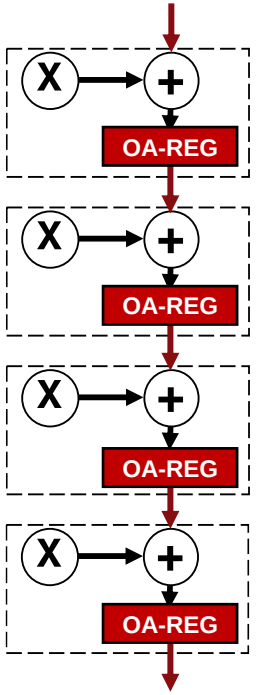
Accelerators

- Source of Inefficiency
- Core Optimizations:
  - Inst. decoding logic
  - Datapath
  - Memory system
- Putting everything together
  - Google's TPU

Mem Opt. 1: Short-lived intermediate results

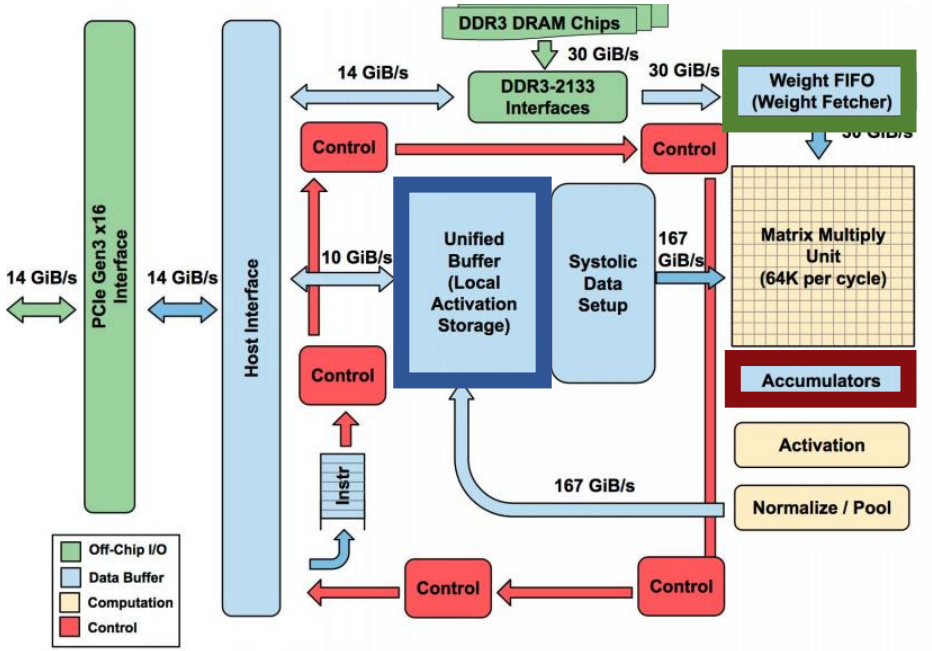
- Instead of accessing SRAM or a shared, large register file
  - Consume the intermediate results directly
- Example: adder tree and systolic accumulation



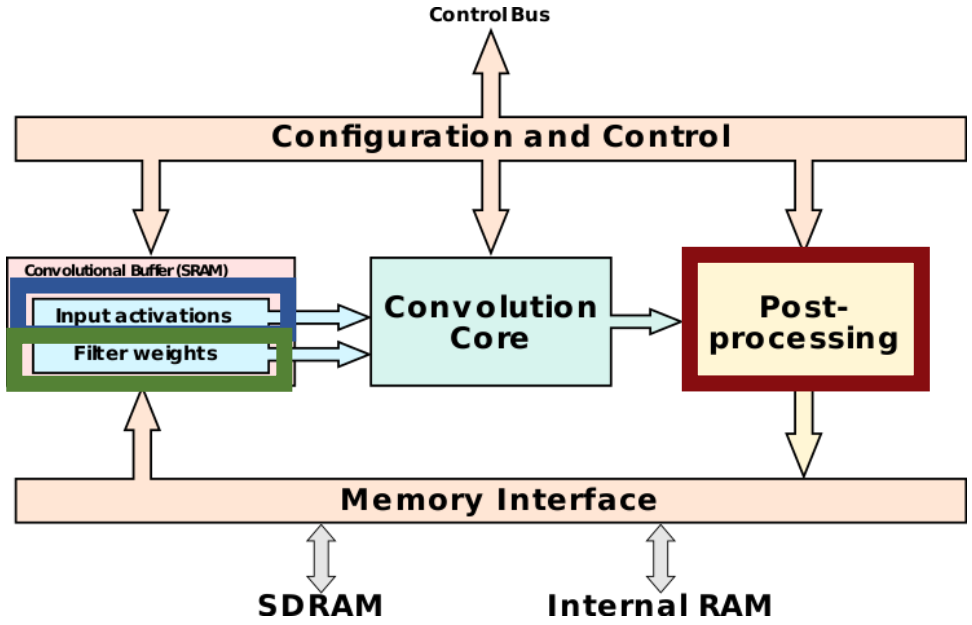


Mem Opt. 2: App-specific storage size/BW

- Dedicated storage for each operand



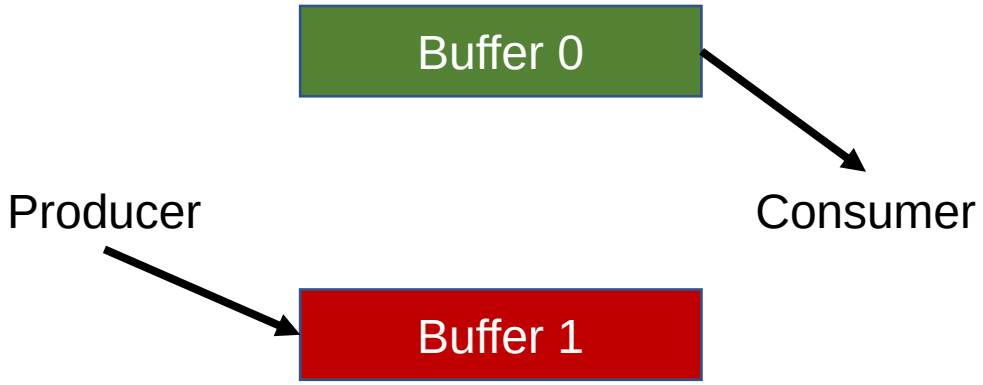
TPU

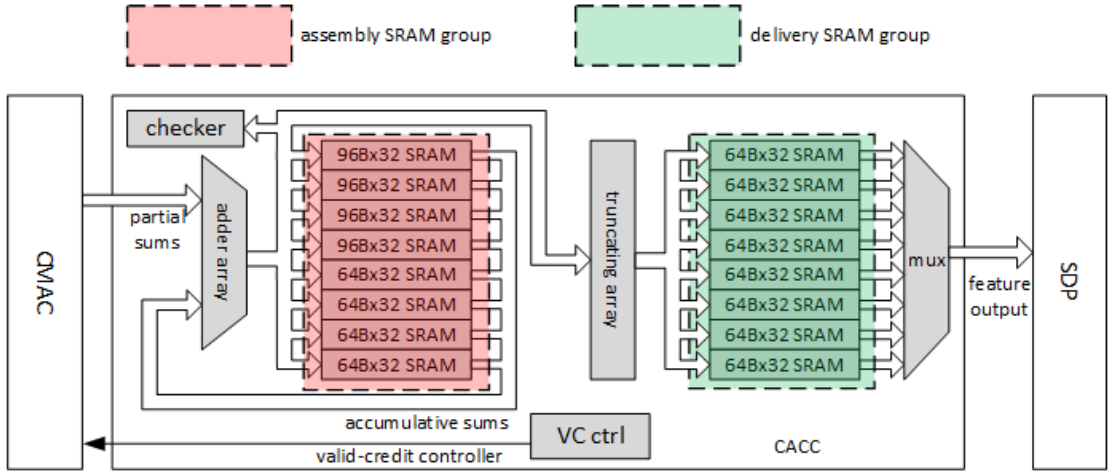


NVDLA

Mem Opt. 3: App-specific data delivery

- Double-buffering
  - Goal: Overlap compute and communication by doubling the SRAM sizes.
  - Producer writes to a buffer while Consumer reads from the other buffer.

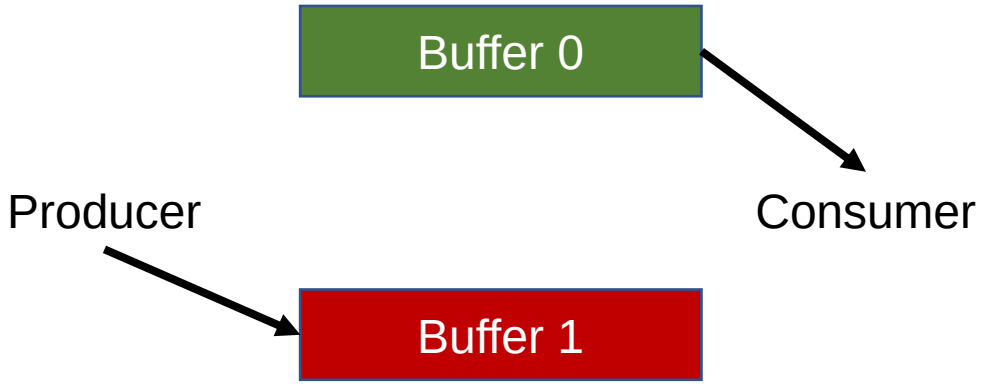


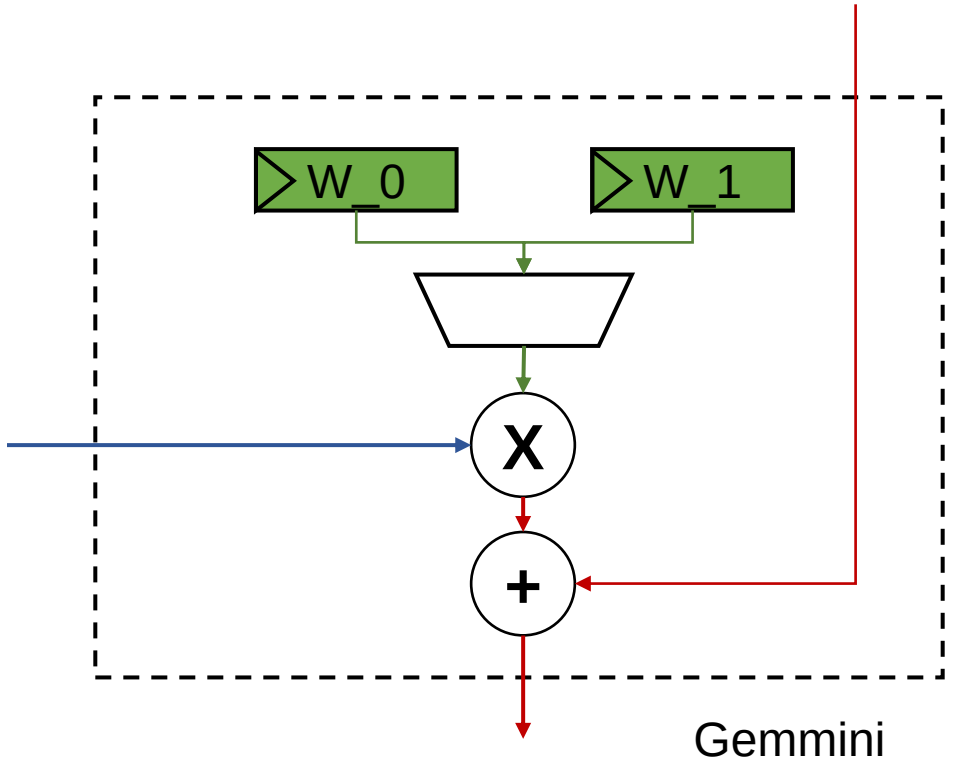


NVDLA

Mem Opt. 3: App-specific data delivery

- Double-buffering
  - Goal: Overlap compute and communication by doubling the SRAM sizes.
  - Producer writes to a buffer while Consumer reads from the other buffer.



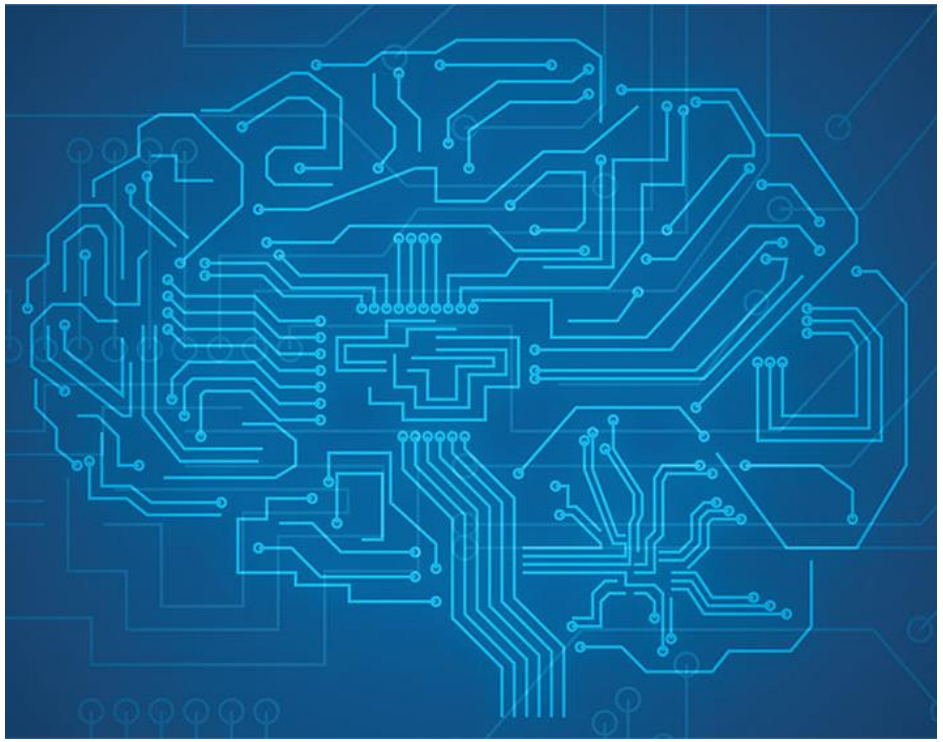


Memory Optimization Recap

- Consume short-lived intermediate results directly
  - Example: adder tree for partial sums
- Application-specific data storage size and bandwidth
  - Example: dedicated weight, input, output buffers
- Application-specific data delivering network
  - Example: double-buffering

Review

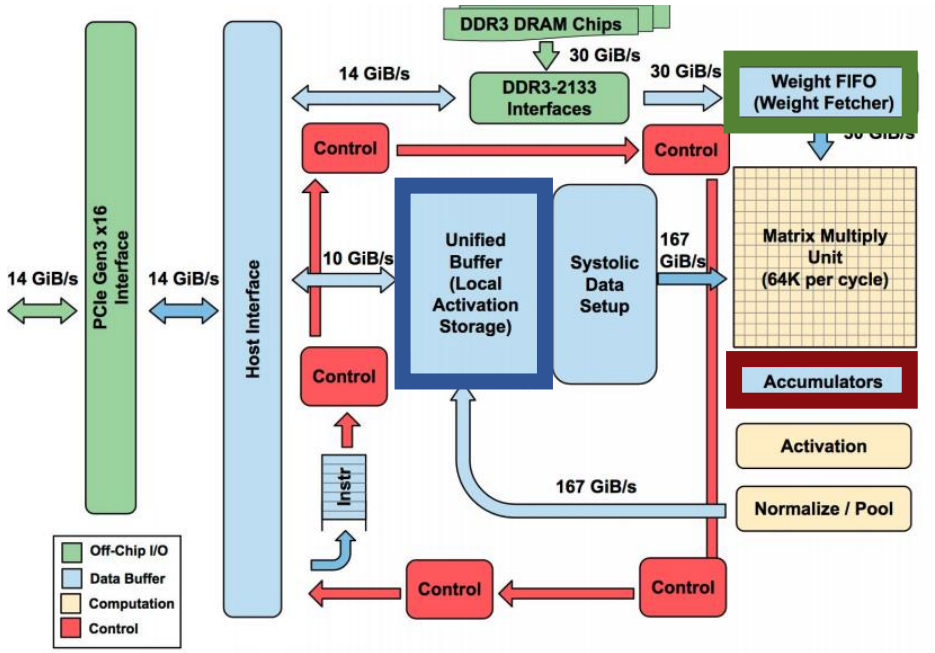
- Lecture 5: core computation in DNN
- Lecture 6: execution order of the core computation
- This lecture: hardware realization of the core computation
  - Overview: source of inefficiency of general-purpose processing
  - Core optimizations:
    - Inst. decoding logic: coarse-grained instructions and operation fusion
    - Datapath: spatial-K and spatial-N/M
    - Memory: short-lived value, app-specific storage and data delivery
  - TPU Example
- Next: Mapping computation to hardware



Accelerators

- Source of Inefficiency
- Core Optimizations:
  - Inst. decoding logic
  - Datapath
  - Memory system
- Putting everything together
  - Google's TPU

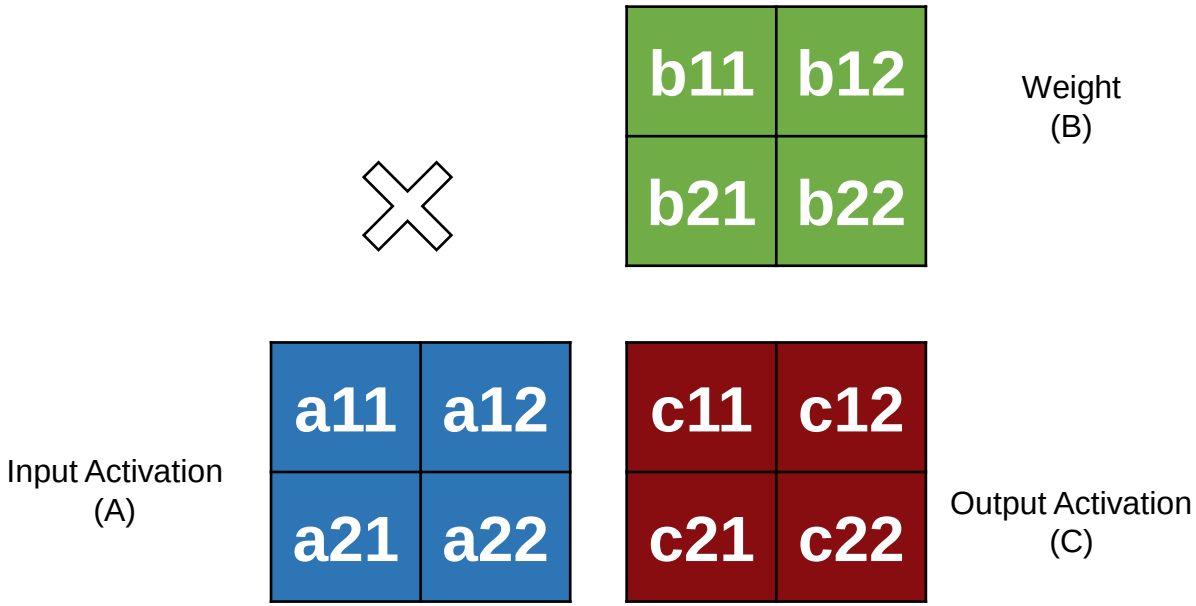
Tensor Processing Unit

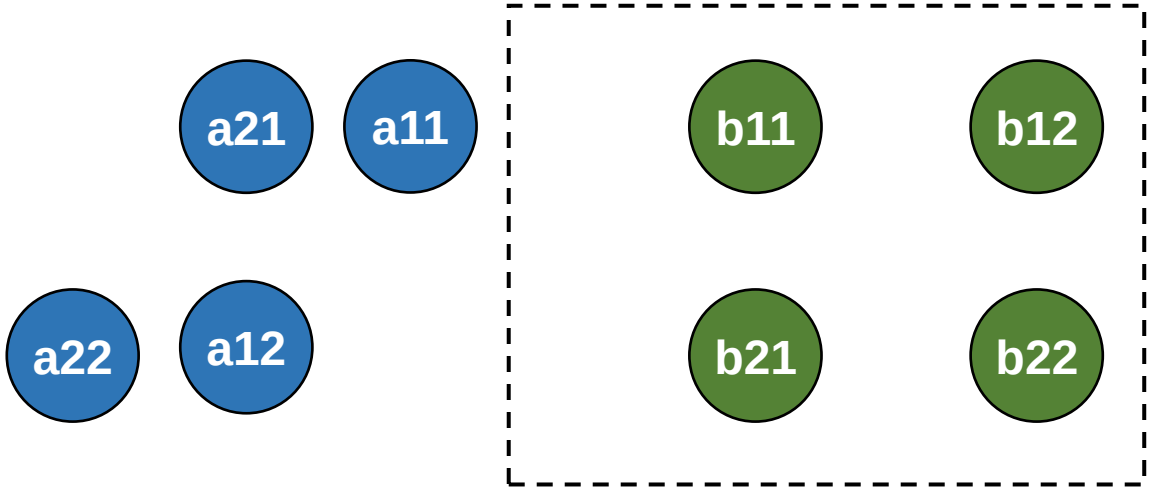


TPU

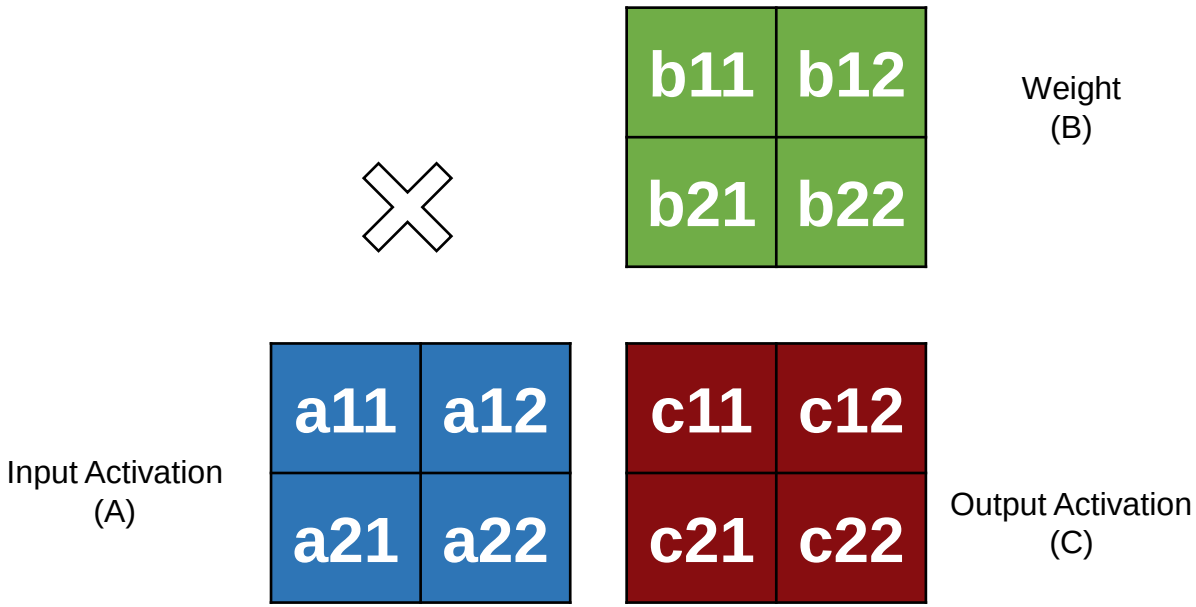
- Inst. Decoding Logic:
  - Coarse-grained matrix multiply and data read/write instructions
- Datapath:
  - Spatial-K: Systolic Accumulation
    - Multi-cycle with registers
  - Spatial-N: Systolic multicast
    - Multi-cycle with registers
  - Better scalability
- Memory
  - Custom systolic registers
  - Dedicated accumulation and weight buffers
  - Double-buffered, weight-stationary dataflow

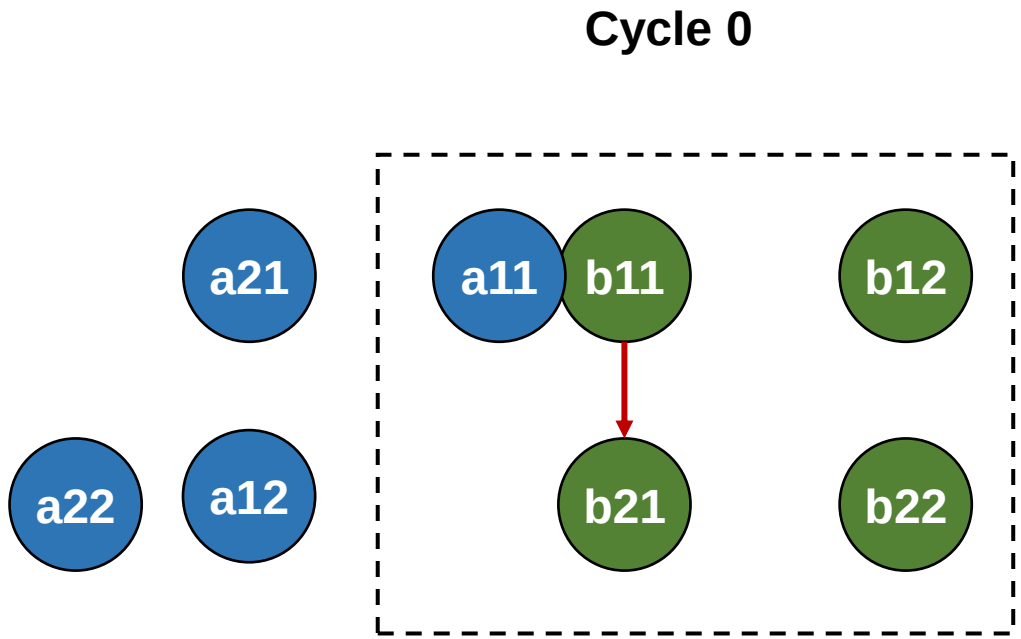
Tensor Processing Unit





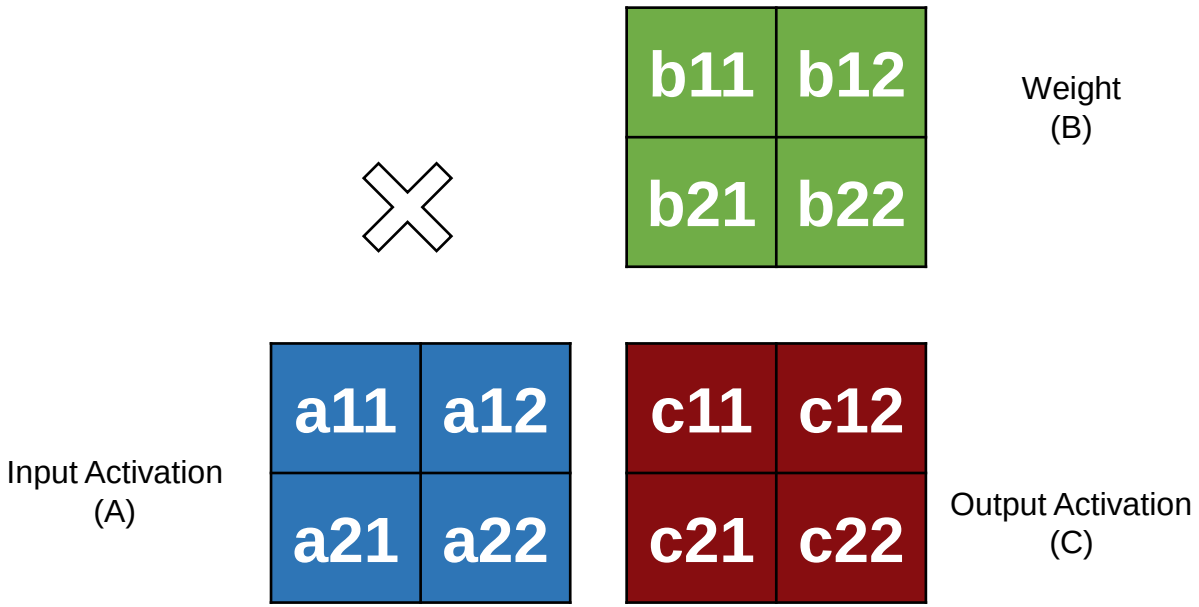
Tensor Processing Unit

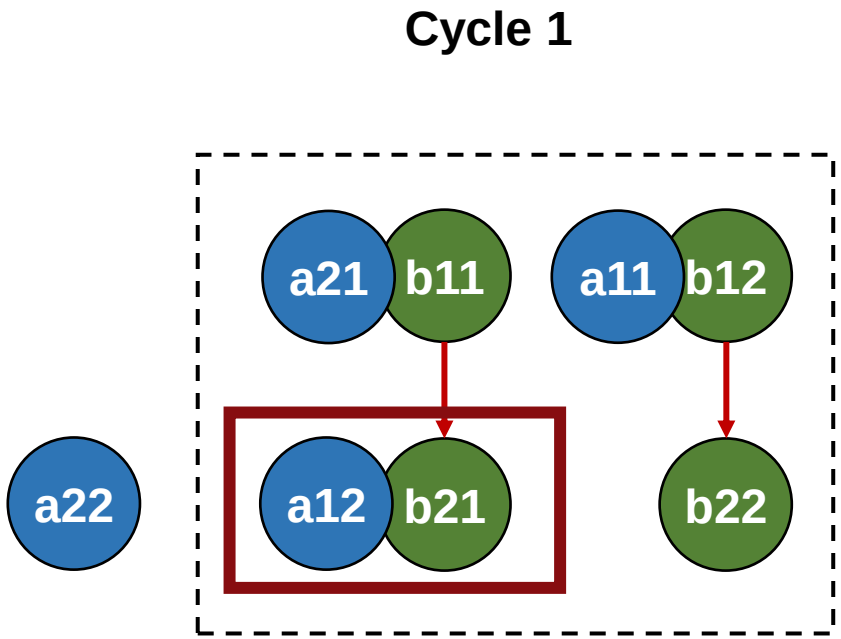




C11 = a11 * b11

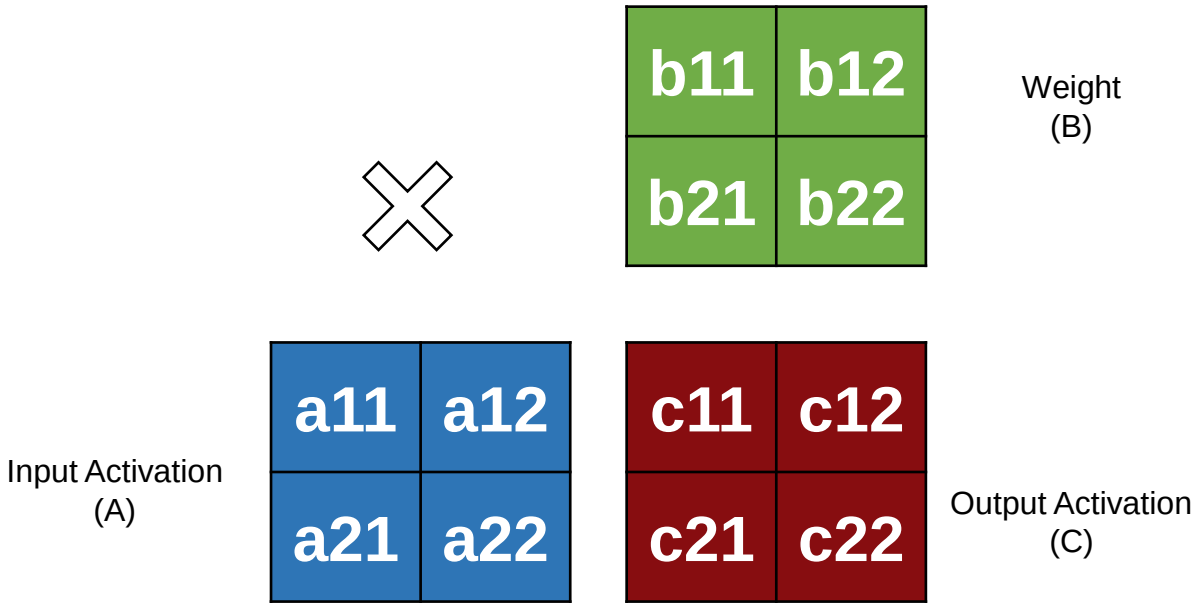
Tensor Processing Unit

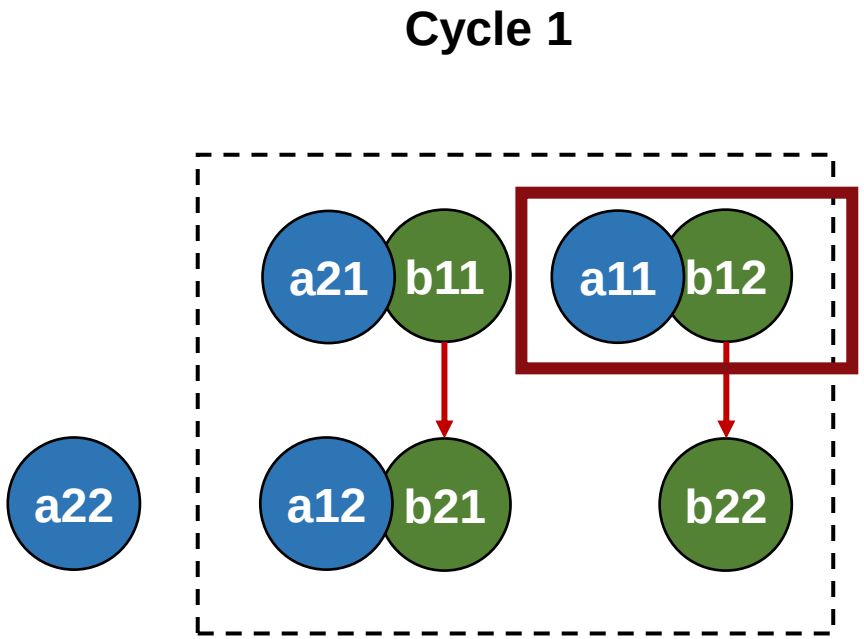




C11 = a11 * b11 + a21 * b21

Tensor Processing Unit

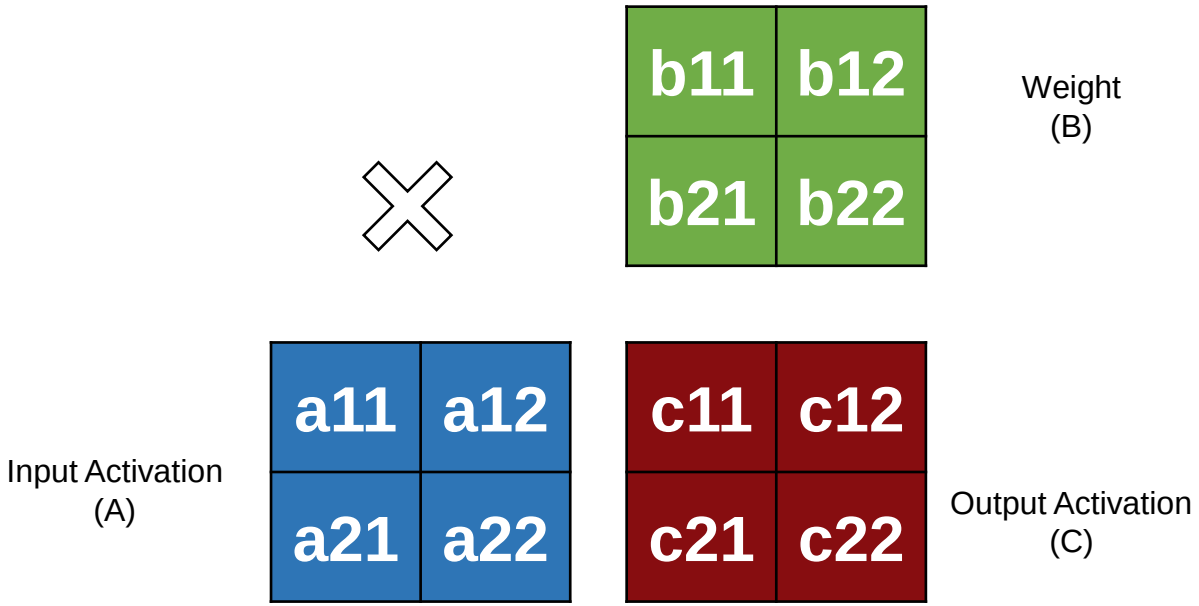


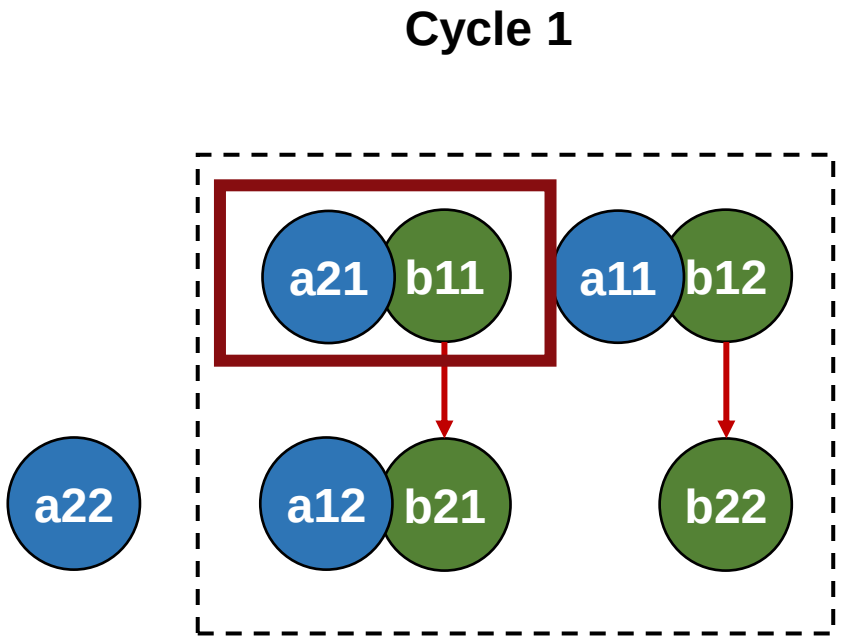


C11 = a11 * b11 + a21 * b21

C12 = a11 * b12

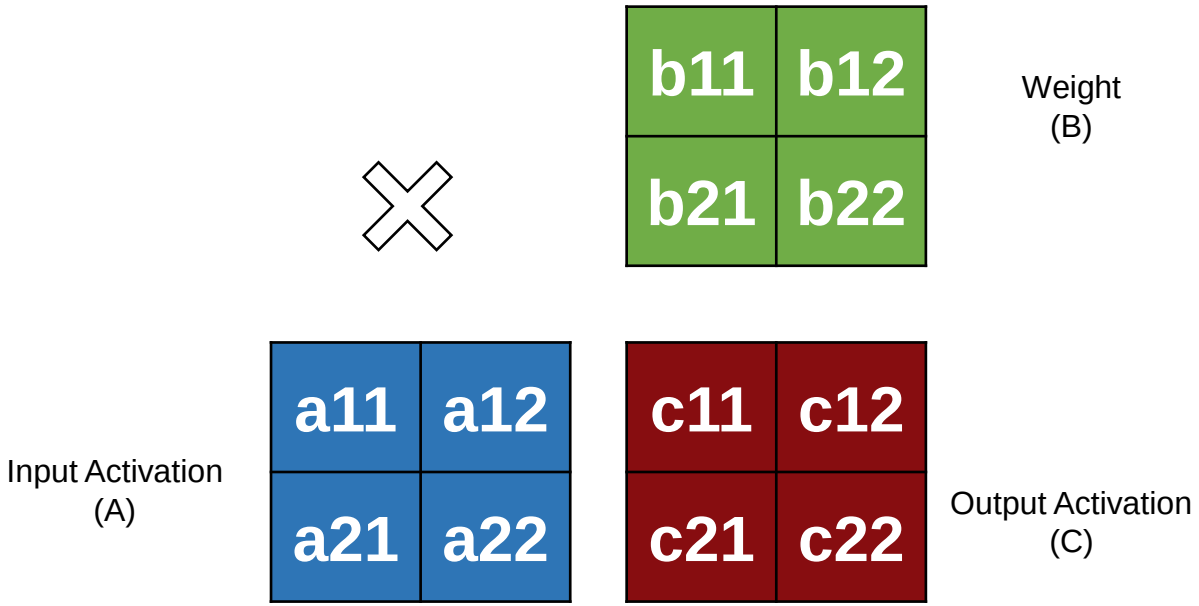
Tensor Processing Unit

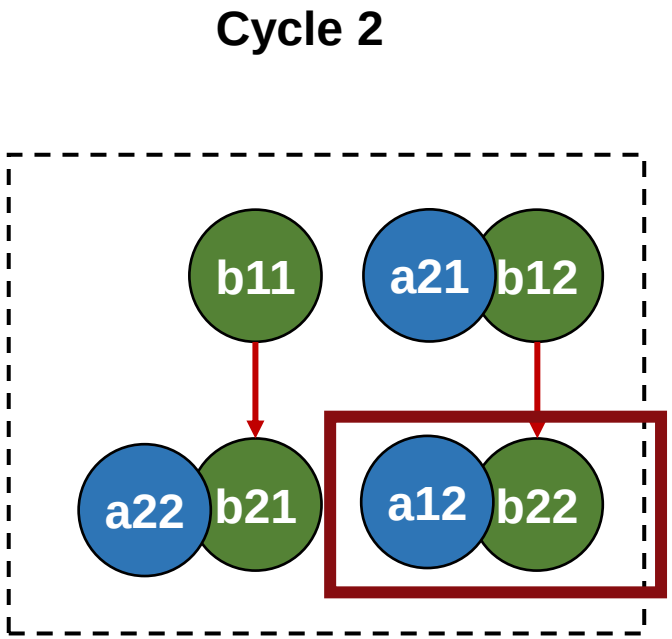




C11 = a11 * b11 + a21 * b21
C12 = a11 * b12
C21 = a21 * b11

Tensor Processing Unit

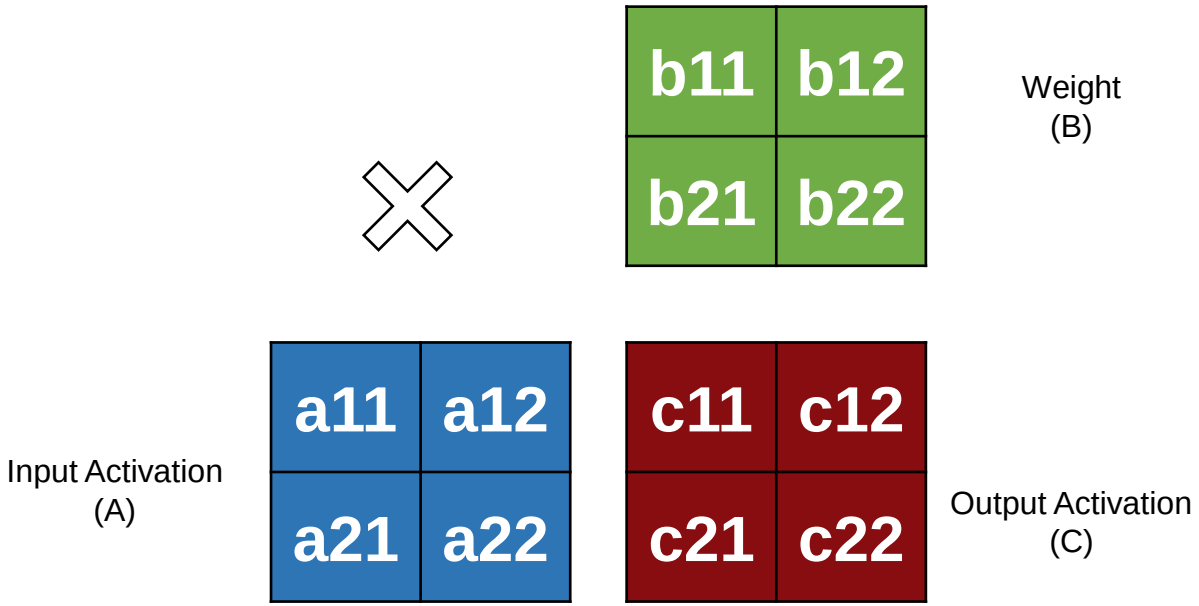


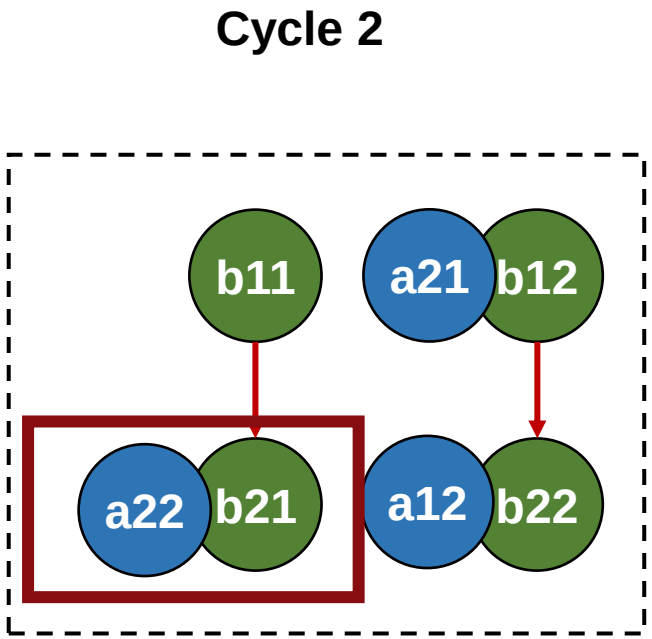


C11 = a11 * b11 + a21 * b21

C12 = a11 * b12 + a12 * b22

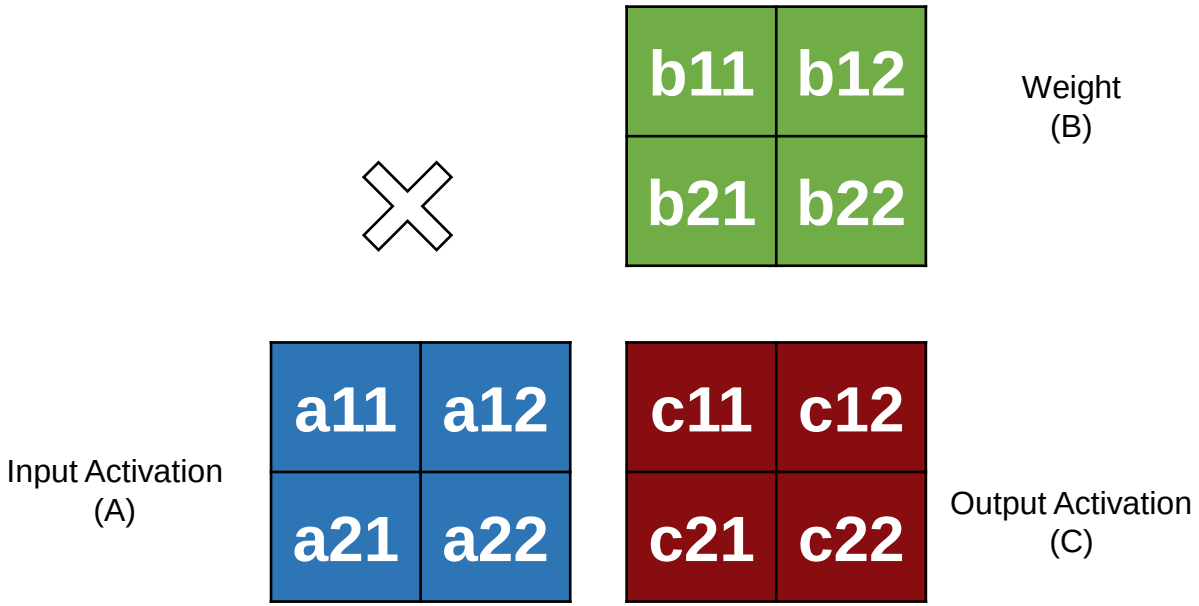
Tensor Processing Unit

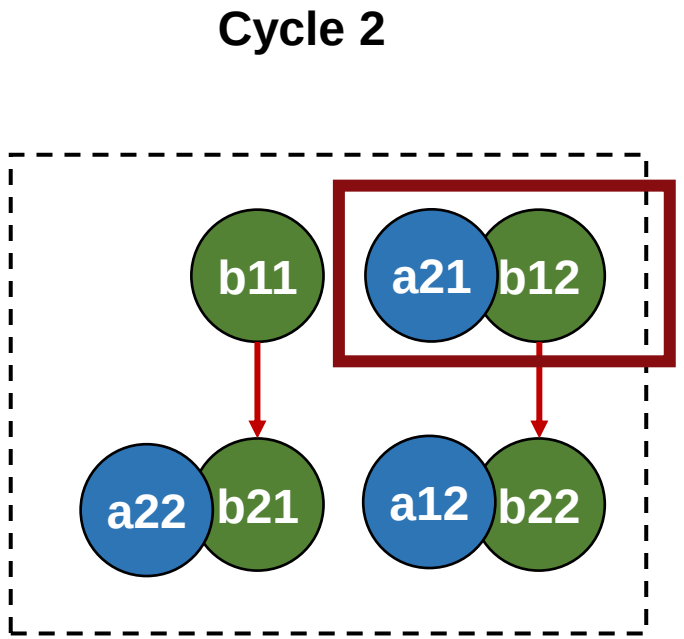




C11 = a11 * b11 + a21 * b21
C12 = a11 * b12 + a12 * b22
C21 = a21 * b11 + a22 * b21

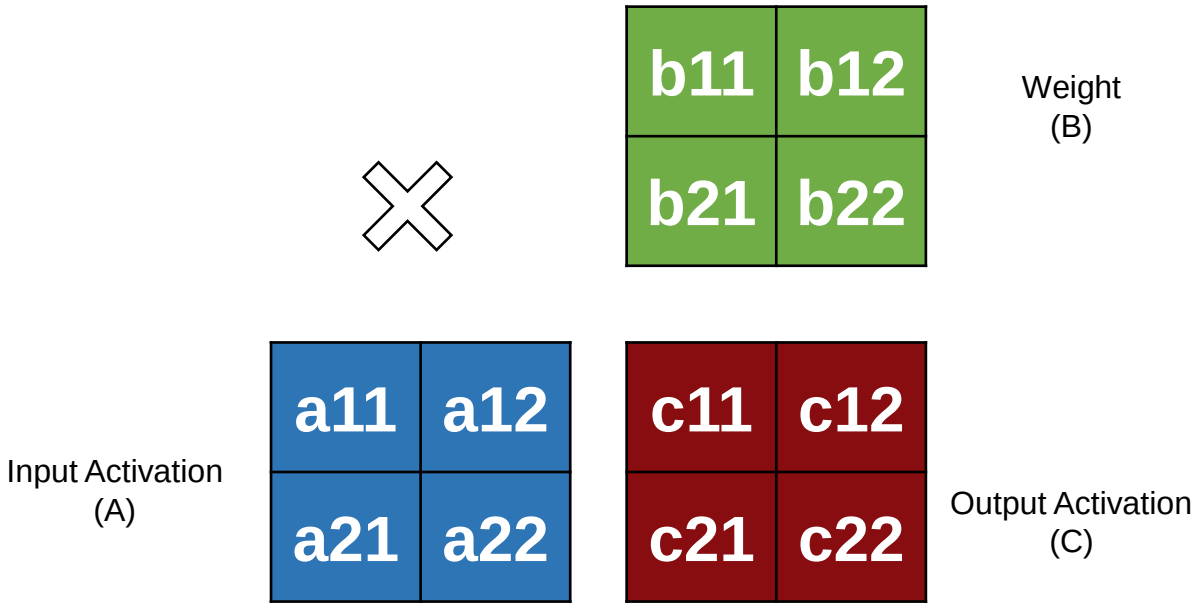
Tensor Processing Unit

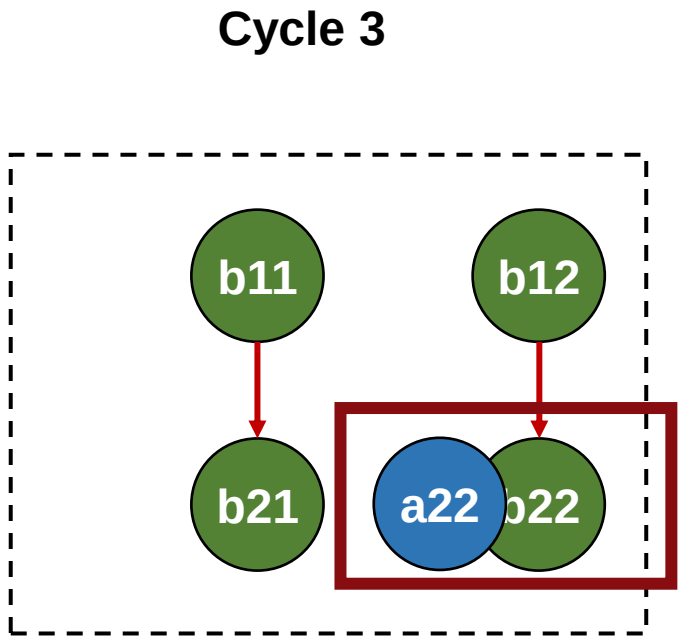




C11 = a11 * b11 + a21 * b21
C12 = a11 * b12 + a12 * b22
C21 = a21 * b11 + a22 * b21
C22 = a21 * b12

Tensor Processing Unit





C11 = a11 * b11 + a21 * b21
C12 = a11 * b12 + a12 * b22
C21 = a21 * b11 + a22 * b21
C22 = a21 * b12 + a22 * b22