Loop Transformations & Quantization

Intelligent Architectures (5LIL0)

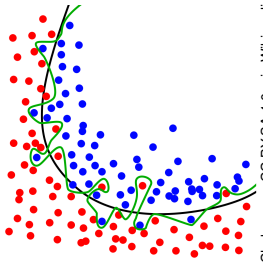
dr Alexios Balatsoukas-Stimming

Department of Electrical Engineering, Electronic Systems Group

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

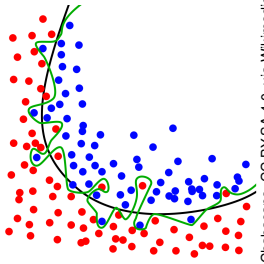
Last Time

- **Overfitting** happens when a model has too much expressive power.
- Can be mitigated by constraining the model: regularization, dropout, etc.

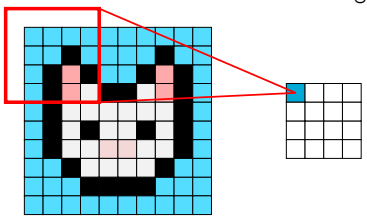


Last Time

- **Overfitting** happens when a model has too much expressive power.
- Can be mitigated by constraining the model: regularization, dropout, etc.



- **Convolutional neural networks** make efficient use of spatial image properties.
- Stride, pooling/unpooling can manipulate internal representation dimensions.



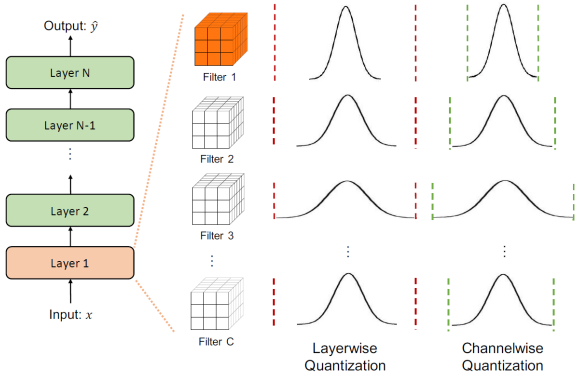
Today

- **Loop transformations:**
  1. Loop interchange
  2. Loop tiling

Today

- **Loop transformations:**
  1. Loop interchange
  2. Loop tiling

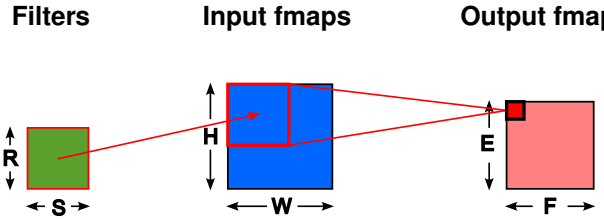
- **Quantization:**
  1. Uniform/non-uniform and symmmetric/asymmetric.
  2. Layer-wise and channel-wise.
  3. Post-training quantization and quantization-aware training.
  4. Advanced topics: non-uniform, mixed-precision, binary.



Figures are taken from [1] unless stated otherwise.

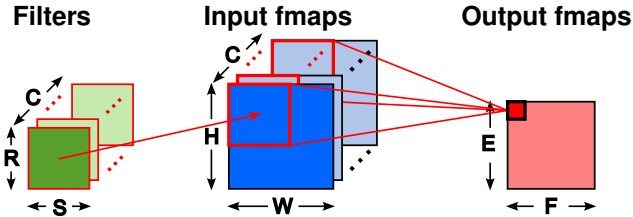
[1] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, K. Keutzer, "A Survey of Quantization Methods for Efficient Neural Network Inference," 2021.

Refresher: The Convolutional Layer



- **Filters**: $S \times R$
- **Input fmaps**: $W \times H$
- **Output fmaps**: $F \times E$

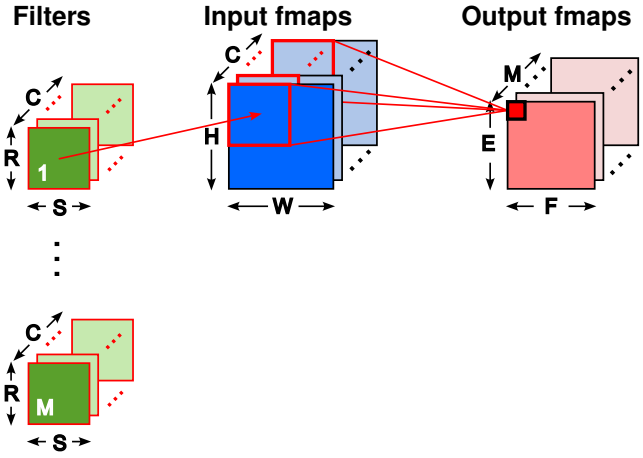
Refresher: The Convolutional Layer



- **Filters**: $S \times R \times C$
- **Input fmaps**: $W \times H \times C$
- **Output fmaps**: $F \times E$

- **Input channels:** C

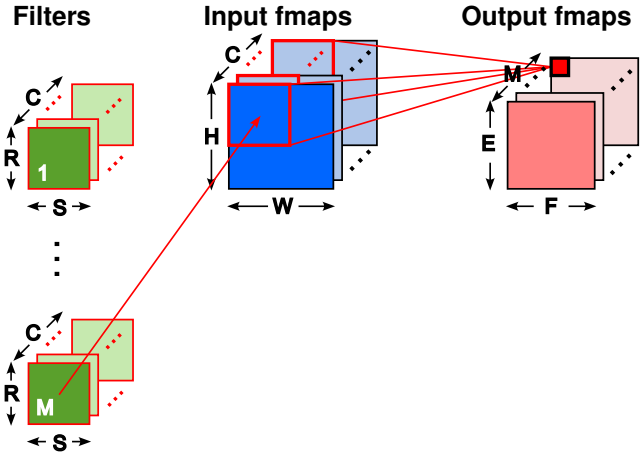
Refresher: The Convolutional Layer



- **Filters**: $S \times R \times C \times M$
- **Input fmaps**: $W \times H \times C$
- **Output fmaps**: $F \times E \times M$

- **Input channels:** C
- **Output channels:** M

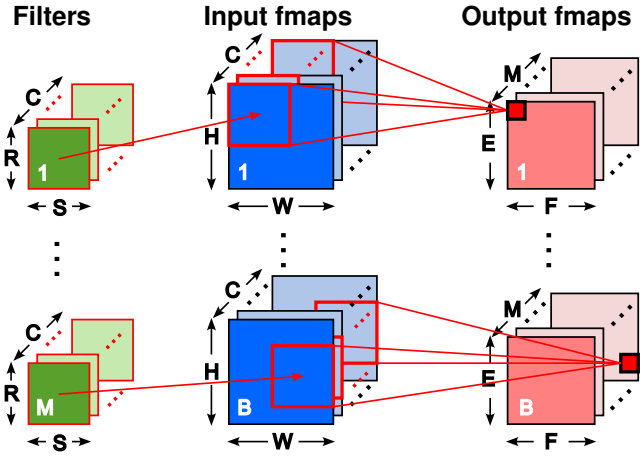
Refresher: The Convolutional Layer



- **Filters**: $S \times R \times C \times M$
- **Input fmaps**: $W \times H \times C$
- **Output fmaps**: $F \times E \times M$

- **Input channels:** C
- **Output channels:** M

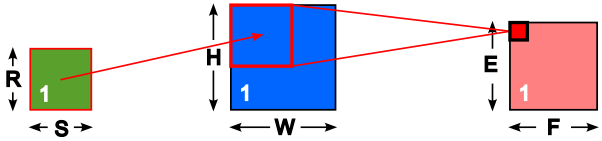
Refresher: The Convolutional Layer



- **Filters**: $S \times R \times C \times M$
- **Input fmaps**: $W \times H \times C \times B$
- **Output fmaps**: $F \times E \times M \times B$

- **Input channels:** C
- **Output channels:** M
- **Batch size:** B

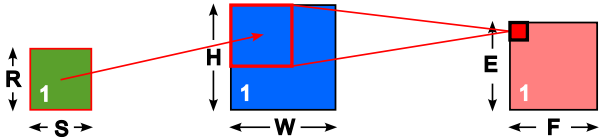
The Convolutional Layer as a Nested Loop



One output fmap item...

```
for(s=0;s<S;s++)
   for(r=0;r<R;r++)
      output+=input[s][r]*weights[s][r];
```

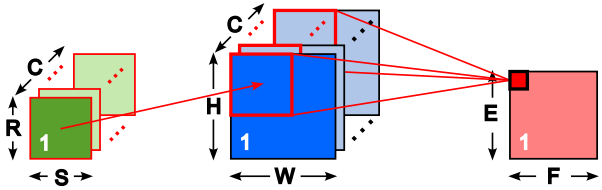
The Convolutional Layer as a Nested Loop



All output fmap items...

```
for(f=0;f<F;f++)
   for(e=0;e<E;e++)
      for(s=0;s<S;s++)
         for(r=0;r<R;r++)
            output[f][e]+=input[f+s][e+r]*weights[s][r];
```

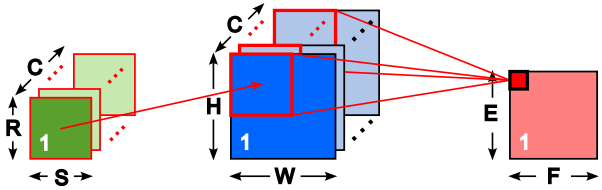
The Convolutional Layer as a Nested Loop



All output fmap items with multiple input channels...

```
for(c=0;c<C;c++)
   for(f=0;f<F;f++)
      for(e=0;e<E;e++)
         for(s=0;s<S;s++)
            for(r=0;r<R;r++)
               output[f][e]+=input[c][f+s][e+r]*weights[c][s][r];
```

The Convolutional Layer as a Nested Loop

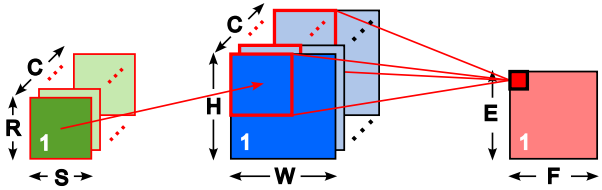


All output fmap items with multiple input channels...

```
for(c=0;c<C;c++)
    for(f=0;f<F;f++)
        for(e=0;e<E;e++)
            for(s=0;s<S;s++)
                for(r=0;r<R;r++)
                    output[f][e]+=input[c][f+s][e+r]*weights[c][s][r];
```

Number of Multiply-and-Accumulate Operations (MACs)

The Convolutional Layer as a Nested Loop



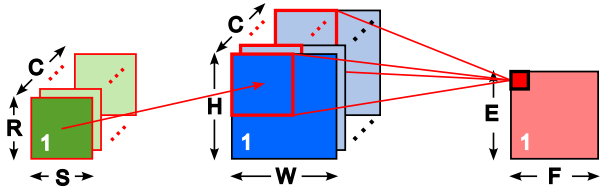
All output fmap items with multiple input channels...

```
for(c=0;c<C;c++)
   for(f=0;f<F;f++)
      for(e=0;e<E;e++)
         for(s=0;s<S;s++)
            for(r=0;r<R;r++)
               output[f][e]+=input[c][f+s][e+r]*weights[c][s][r];
```

Number of Multiply-and-Accumulate Operations (MACs)

$$\#\text{MACs} = R \times S \times E \times F \times C$$

The Convolutional Layer as a Nested Loop



All output fmap items with multiple input channels... and so on (output channels, batch).

```
for(c=0;c<C;c++)
   for(f=0;f<F;f++)
      for(e=0;e<E;e++)
         for(s=0;s<S;s++)
            for(r=0;r<R;r++)
               output[f][e]+=input[c][f+s][e+r]*weights[c][s][r];
```

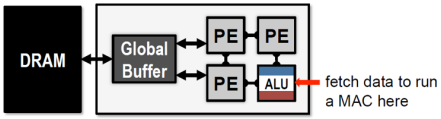
Number of Multiply-and-Accumulate Operations (MACs)

$$\#\text{MACs} = R \times S \times E \times F \times C \times M \times B$$

Memory Hierarchy and Energy Cost

- Computation systems have **multiple levels of memory**.

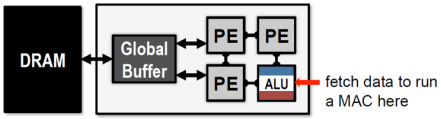
Memory Hierarchy and Energy Cost

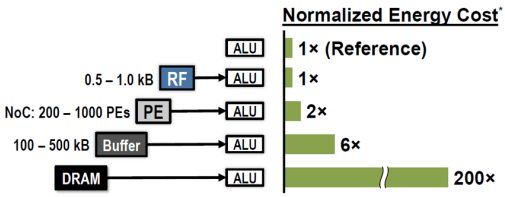


- Computation systems have **multiple levels of memory**. For example [1]:
  1. **Ultra-fast and expensive register files** (RFs) in processing elements (PEs).
  2. **Fast but less expensive SRAM global buffers** for PE groups.
  3. **Slow and cheap DRAM memory** for entire system.

[1] V. Sze, Y.-H. Chen, T.-J. Yang, J. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," 2017.

Memory Hierarchy and Energy Cost

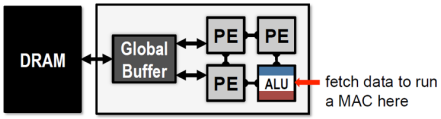


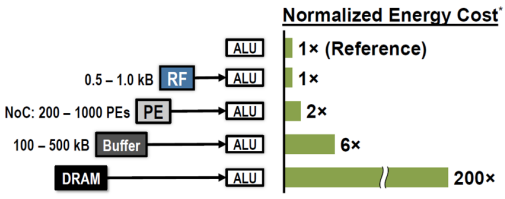


- Computation systems have **multiple levels of memory**. For example [1]:
  1. **Ultra-fast and expensive register files** (RFs) in processing elements (PEs).
  2. **Fast but less expensive SRAM global buffers** for PE groups.
  3. **Slow and cheap DRAM memory** for entire system.
- Energy cost of memory access **varies by orders of magnitude**!

[1] V. Sze, Y.-H. Chen, T.-J. Yang, J. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," 2017.

Memory Hierarchy and Energy Cost





- Computation systems have **multiple levels of memory**. For example [1]:
  1. **Ultra-fast and expensive register files** (RFs) in processing elements (PEs).
  2. **Fast but less expensive SRAM global buffers** for PE groups.
  3. **Slow and cheap DRAM memory** for entire system.
- Energy cost of memory access **varies by orders of magnitude**!

Loop Transformations:

Reduce memory traffic by exploiting data re-use!

[1] V. Sze, Y.-H. Chen, T.-J. Yang, J. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," 2017.

Exploiting Temporal Locality

Definitions:

- **Temporal locality:** The tendency of programs to re-use data items within a relatively small number of operations.

Exploiting Temporal Locality

Definitions:

- **Temporal locality:** The tendency of programs to re-use data items within a relatively small number of operations.
- **Re-use distance (RD):** The number of unique memory addresses accessed since the last access to a given memory address. It is a measure of temporal locality.

Exploiting Temporal Locality

Definitions:

- **Temporal locality:** The tendency of programs to re-use data items within a relatively small number of operations.
- **Re-use distance (RD):** The number of unique memory addresses accessed since the last access to a given memory address. It is a measure of temporal locality.

Example: Re-use Distance

Address	A	B	C	A	B	B	A	C	D	B
RD(Address)										

Exploiting Temporal Locality

Definitions:

- **Temporal locality:** The tendency of programs to re-use data items within a relatively small number of operations.
- **Re-use distance (RD):** The number of unique memory addresses accessed since the last access to a given memory address. It is a measure of temporal locality.

Example: Re-use Distance

Address	A	B	C	A	B	B	A	C	D	B
RD(Address)	∞									

Exploiting Temporal Locality

Definitions:

- **Temporal locality:** The tendency of programs to re-use data items within a relatively small number of operations.
- **Re-use distance (RD):** The number of unique memory addresses accessed since the last access to a given memory address. It is a measure of temporal locality.

Example: Re-use Distance

Address	A	B	C	A	B	B	A	C	D	B
RD(Address)	∞	∞	∞							

Exploiting Temporal Locality

Definitions:

- **Temporal locality:** The tendency of programs to re-use data items within a relatively small number of operations.
- **Re-use distance (RD):** The number of unique memory addresses accessed since the last access to a given memory address. It is a measure of temporal locality.

Example: Re-use Distance

Address	A	B	C	A	B	B	A	C	D	B
RD(Address)	∞	∞	∞	2						

Exploiting Temporal Locality

Definitions:

- **Temporal locality:** The tendency of programs to re-use data items within a relatively small number of operations.
- **Re-use distance (RD):** The number of unique memory addresses accessed since the last access to a given memory address. It is a measure of temporal locality.

Example: Re-use Distance

Address	A	B	C	A	B	B	A	C	D	B
RD(Address)	∞	∞	∞	2	2					

Exploiting Temporal Locality

Definitions:

- **Temporal locality:** The tendency of programs to re-use data items within a relatively small number of operations.
- **Re-use distance (RD):** The number of unique memory addresses accessed since the last access to a given memory address. It is a measure of temporal locality.

Example: Re-use Distance

Address	A	B	C	A	B	B	A	C	D	B
RD(Address)	∞	∞	∞	2	2	0				

Exploiting Temporal Locality

Definitions:

- **Temporal locality:** The tendency of programs to re-use data items within a relatively small number of operations.
- **Re-use distance (RD):** The number of unique memory addresses accessed since the last access to a given memory address. It is a measure of temporal locality.

Example: Re-use Distance

Address	A	B	C	A	B	B	A	C	D	B
RD(Address)	∞	∞	∞	2	2	0	1	2	∞	3

Exploiting Temporal Locality

Definitions:

- **Temporal locality:** The tendency of programs to re-use data items within a relatively small number of operations.
- **Re-use distance (RD):** The number of unique memory addresses accessed since the last access to a given memory address. It is a measure of temporal locality.

Example: Re-use Distance

Address	A	B	C	A	B	B	A	C	D	B
RD(Address)	∞	∞	∞	2	2	0	1	2	∞	3

- If the local memory is larger than the RD, we have a **cache hit** $\Rightarrow$ no data to transfer!

Transformations: Loop-Invariant Code Motion

Original:

```
for(i=0;i<N;i++)
    for(j=0;j<N;j++)
        A[i][j]=B[j][i]+C[i];
```

- `C[i]` does not change in inner loop

Transformations: Loop-Invariant Code Motion

Original:

```
for(i=0;i<N;i++)
    for(j=0;j<N;j++)
        A[i][j]=B[j][i]+C[i];
```

Transformed:

```
for(i=0;i<N;i++)
    T = C[i];
    for(j=0;j<N;j++)
        A[i][j]=B[j][i]+ T ;
```

- `C[i]` does not change in inner loop
- Move to outer loop, store in register, and re-use

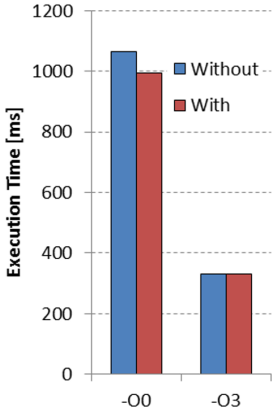
Transformations: Loop-Invariant Code Motion

Original:

```
for(i=0;i<N;i++)
    for(j=0;j<N;j++)
        A[i][j]=B[j][i]+C[i];
```

Transformed:

```
for(i=0;i<N;i++)
    T = C[i];
    for(j=0;j<N;j++)
        A[i][j]=B[j][i]+ T ;
```



- `C[i]` does not change in inner loop
- Move to outer loop, store in register, and re-use

- **Not so interesting:** effect is small and compilers can do this automatically

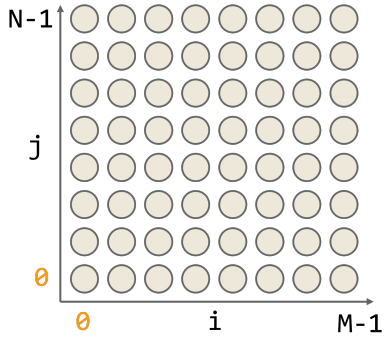
Iteration Space

```
for(j=0;j<N;j++)
   for(i=0;i<M;i++)
      A[i][j]=B[j][i];
```

Iteration Space

```
for(j=0;j<N;j++)
   for(i=0;i<M;i++)
      A[i][j]=B[j][i];
```

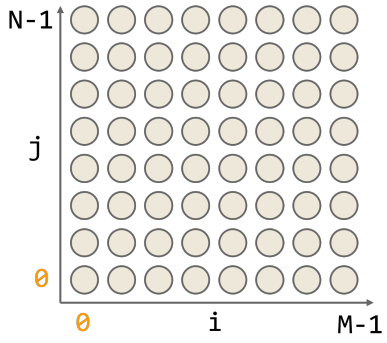
- Each point represents one `(i,j)` pair



Iteration Space

```
for(j=0;j<N;j++)
   for(i=0;i<M;i++)
      A[i][j]=B[j][i];
```

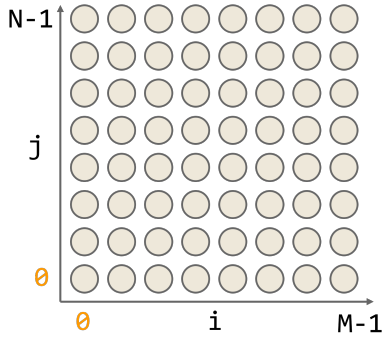
- Each point represents one `(i,j)` pair
- Many orders are possible (careful with data dependencies)



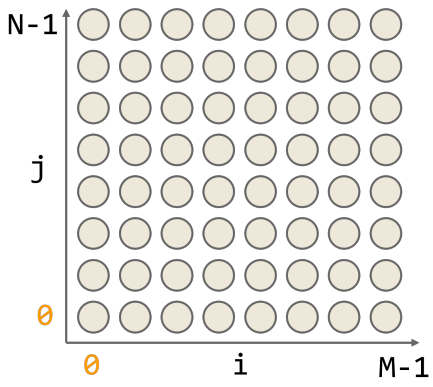
Iteration Space

```
for(j=0;j<N;j++)
   for(i=0;i<M;i++)
      A[i][j]=B[j][i];
```

- Each point represents one `(i,j)` pair
- Many orders are possible (careful with data dependencies)
- Order **greatly impacts**:
  1. Data re-use
  2. Parallelizability
  3. Loop control overhead



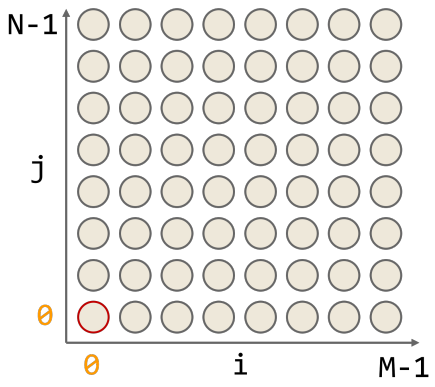
Transformations: Loop Interchange



Original:

```
for(j=0;j<N;j++)
   for(i=0;i<M;i++)
      B[j][i]+=A[i];
```

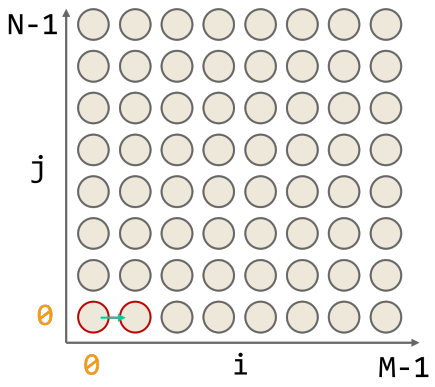
Transformations: Loop Interchange



Original:

```
for(j=0;j<N;j++)
   for(i=0;i<M;i++)
      B[j][i]+=A[i];
```

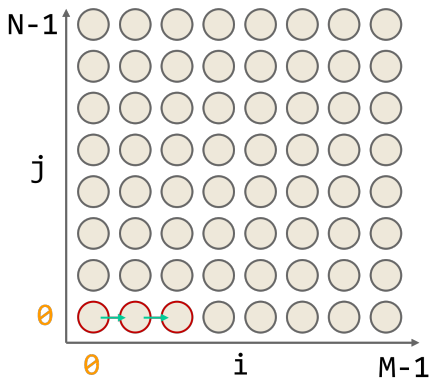
Transformations: Loop Interchange



Original:

```
for(j=0;j<N;j++)
   for(i=0;i<M;i++)
      B[j][i]+=A[i];
```

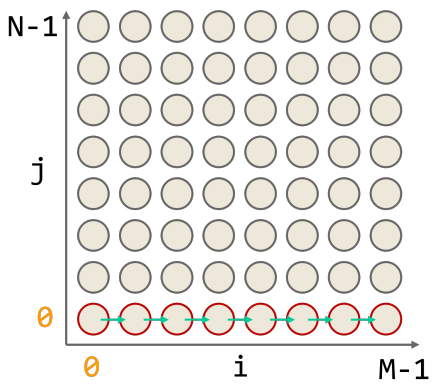
Transformations: Loop Interchange



Original:

```
for(j=0;j<N;j++)
   for(i=0;i<M;i++)
      B[j][i]+=A[i];
```

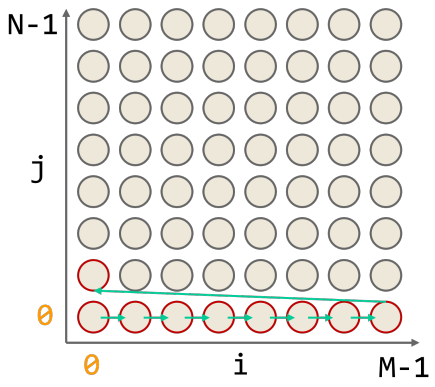
Transformations: Loop Interchange



Original:

```
for(j=0;j<N;j++)
   for(i=0;i<M;i++)
      B[j][i]+=A[i];
```

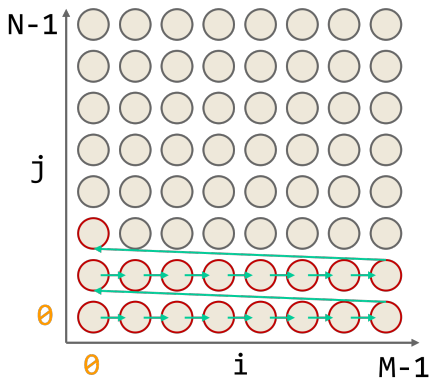
Transformations: Loop Interchange



Original:

```
for(j=0;j<N;j++)
   for(i=0;i<M;i++)
      B[j][i]+=A[i];
```

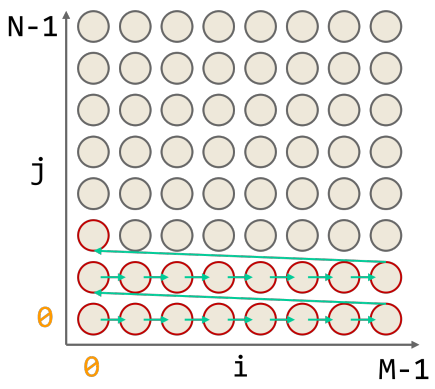
Transformations: Loop Interchange



Original:

```
for(j=0;j<N;j++)
   for(i=0;i<M;i++)
      B[j][i]+=A[i];
```

Transformations: Loop Interchange

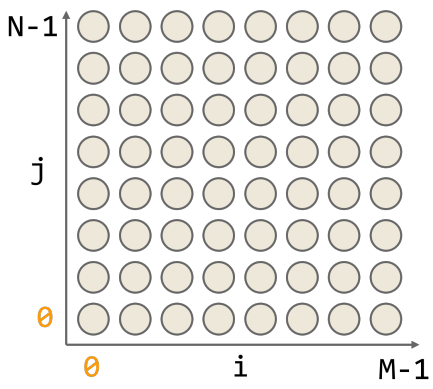


Original:

```
for(j=0;j<N;j++)
   for(i=0;i<M;i++)
      B[j][i]+=A[i];
```

- `A[0]` accessed once for every run of internal loop, so $RD(\texttt{A[0]}) = M - 1$.

Transformations: Loop Interchange



Original:

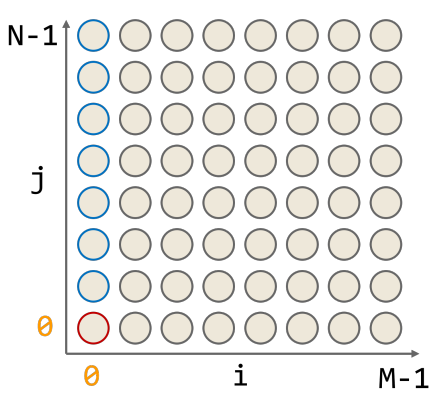
```
for(j=0;j<N;j++)
   for(i=0;i<M;i++)
      B[j][i]+=A[i];
```

- `A[0]` accessed once for every run of internal loop, so $RD(\texttt{A[0]}) = M - 1$.

Interchanged:

```
for(i=0;i<M;i++)
   for(j=0;j<N;j++)
      B[j][i]+=A[i];
```

Transformations: Loop Interchange



Original:

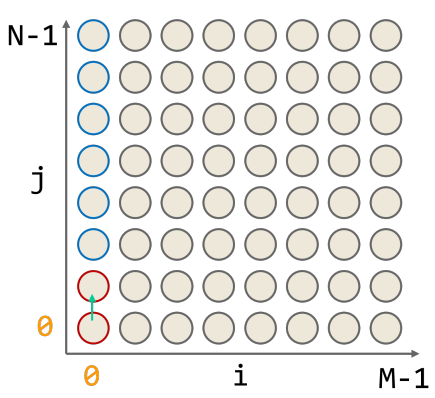
```
for(j=0;j<N;j++)
   for(i=0;i<M;i++)
      B[j][i]+=A[i];
```

- `A[0]` accessed once for every run of internal loop, so $RD(\texttt{A[0]}) = M - 1$.

Interchanged:

```
for(i=0;i<M;i++)
   for(j=0;j<N;j++)
      B[j][i]+=A[i];
```

Transformations: Loop Interchange



Original:

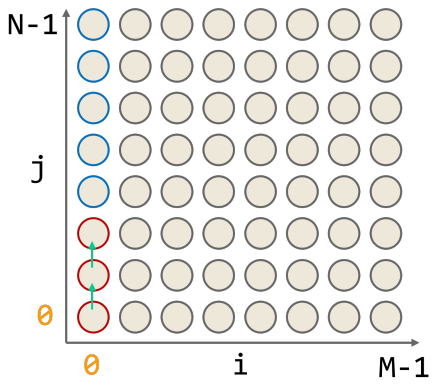
```
for(j=0;j<N;j++)
   for(i=0;i<M;i++)
      B[j][i]+=A[i];
```

- `A[0]` accessed once for every run of internal loop, so $RD(\texttt{A[0]}) = M - 1$.

Interchanged:

```
for(i=0;i<M;i++)
   for(j=0;j<N;j++)
      B[j][i]+=A[i];
```

Transformations: Loop Interchange



Original:

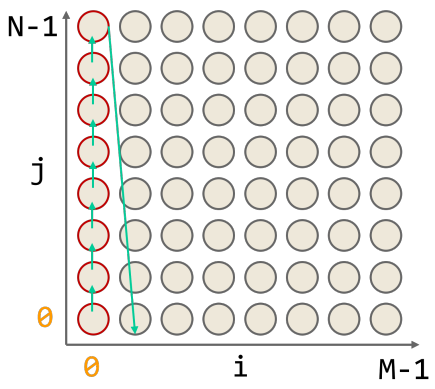
```
for(j=0;j<N;j++)
   for(i=0;i<M;i++)
      B[j][i]+=A[i];
```

- `A[0]` accessed once for every run of internal loop, so $RD(\texttt{A[0]}) = M - 1$.

Interchanged:

```
for(i=0;i<M;i++)
   for(j=0;j<N;j++)
      B[j][i]+=A[i];
```

Transformations: Loop Interchange



Original:

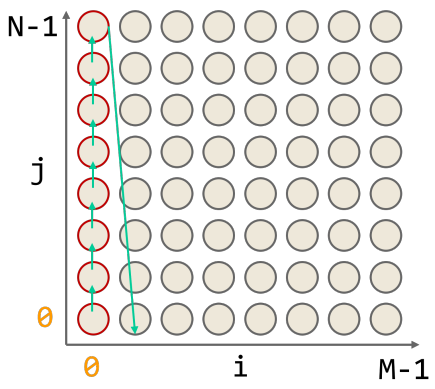
```
for(j=0;j<N;j++)
   for(i=0;i<M;i++)
      B[j][i]+=A[i];
```

- `A[0]` accessed once for every run of internal loop, so $RD(\texttt{A[0]}) = M - 1$.

Interchanged:

```
for(i=0;i<M;i++)
   for(j=0;j<N;j++)
      B[j][i]+=A[i];
```

Transformations: Loop Interchange



Original:

```
for(j=0;j<N;j++)
   for(i=0;i<M;i++)
      B[j][i]+=A[i];
```

- `A[0]` accessed once for every run of internal loop, so $RD(\texttt{A[0]}) = M - 1$.

Interchanged:

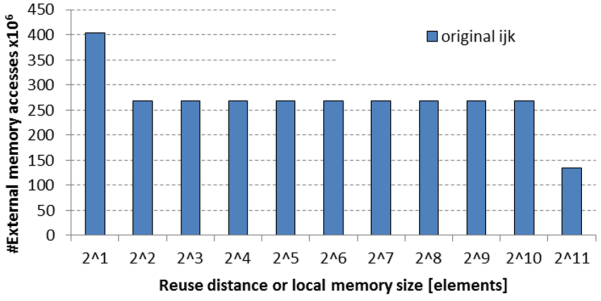
```
for(i=0;i<M;i++)
   for(j=0;j<N;j++)
      B[j][i]+=A[i];
```

- `A[0]` accessed for all iterations of internal loop when `i=0`, so $RD(\texttt{A[0]}) = 0$.

Loop Interchange Example: Matrix Multiplication

Original

```
for(i=0;i<Ni;i++)
   for(j=0;j<Nj;j++)
      for(k=0;j<Nk;k++)
         C[i][j]+=A[i][k]*B[k][j];
```



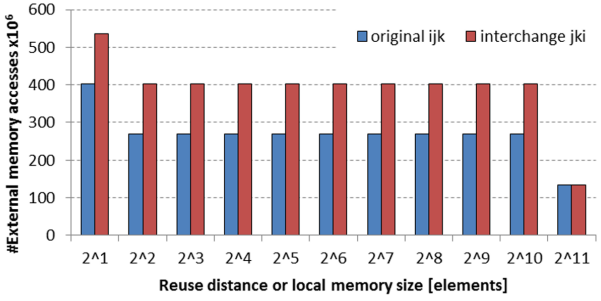
Loop Interchange Example: Matrix Multiplication

Original

```
for(i=0;i<Ni;i++)
   for(j=0;j<Nj;j++)
      for(k=0;j<Nk;k++)
         C[i][j]+=A[i][k]*B[k][j];
```

Interchanged

```
for(j=0;j<Nj;j++)
   for(k=0;j<Nk;k++)
      for(i=0;i<Ni;i++)
         C[i][j]+=A[i][k]*B[k][j];
```



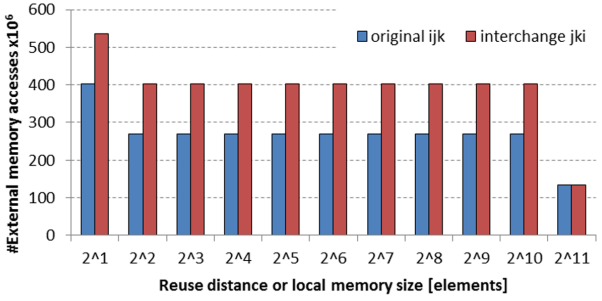
Loop Interchange Example: Matrix Multiplication

Original

```
for(i=0;i<Ni;i++)
    for(j=0;j<Nj;j++)
        for(k=0;j<Nk;k++)
            C[i][j]+=A[i][k]*B[k][j];
```

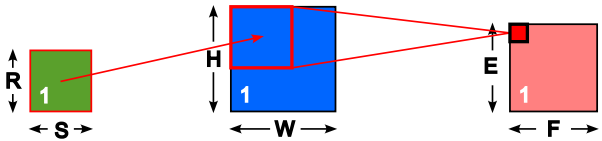
Interchanged

```
for(j=0;j<Nj;j++)
    for(k=0;j<Nk;k++)
        for(i=0;i<Ni;i++)
            C[i][j]+=A[i][k]*B[k][j];
```



No guarantee that interchange is beneficial for re-use → **tread carefully!**

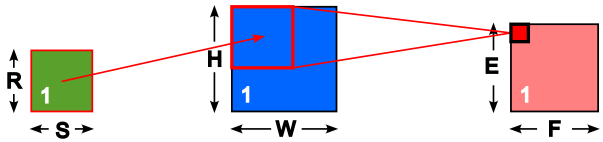
Loop Interchange and CNNs




```
for(f=0;f<F;f++)
   for(e=0;e<E;e++)
      for(s=0;s<S;s++)
         for(r=0;r<R;r++)
            output[f][e]+=input[f+s][e+r]*weights[s][r];
```

- **Example:** Convolutional layer with one input channel and one output channel

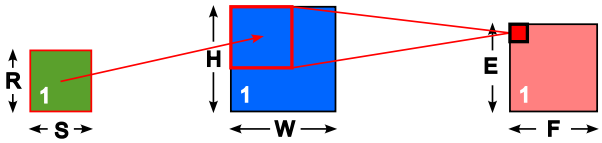
Loop Interchange and CNNs




```
for(f=0;f<F;f++)
   for(e=0;e<E;e++)
      for(s=0;s<S;s++)
         for(r=0;r<R;r++)
            output[f][e]+=input[f+s][e+r]*weights[s][r];
```

- **Example:** Convolutional layer with one input channel and one output channel
  1. **Original order:** re-use of `output` is maximized (“output-stationary”)

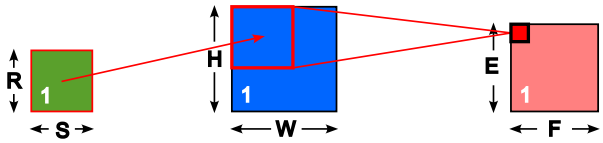
Loop Interchange and CNNs




```
for(s=0;s<S;s++)
   for(r=0;r<R;r++)
      for(f=0;f<F;f++)
         for(e=0;e<E;e++)
            output[f][e]+=input[f+s][e+r]*weights[s][r];
```

- **Example:** Convolutional layer with one input channel and one output channel
  1. **Original order:** re-use of `output` is maximized (“output-stationary”)
  2. **Interchange inner/outer loops:** re-use of `weights` is maximized (“weight-stationary”)

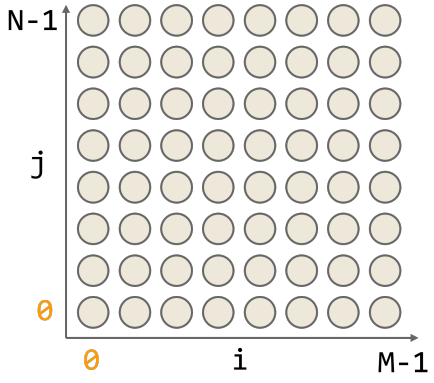
Loop Interchange and CNNs




```
for(s=0;s<S;s++)
   for(r=0;r<R;r++)
      for(f=0;f<F;f++)
         for(e=0;e<E;e++)
            output[f][e]+=input[f+s][e+r]*weights[s][r];
```

- **Example:** Convolutional layer with one input channel and one output channel
  1. **Original order:** re-use of `output` is maximized (“output-stationary”)
  2. **Interchange inner/outer loops:** re-use of `weights` is maximized (“weight-stationary”)
- Re-use of `input` is more tricky because it depends on all loop variables

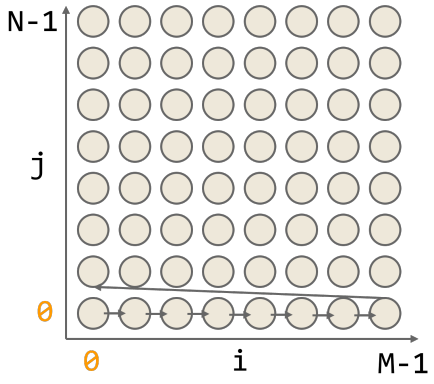
Transformations: Loop Tiling



Original:

```
for(j=0;j<N;j++)
    for(i=0;i<M;i++)
        B[j][i]+=A[i]+C[j];
```

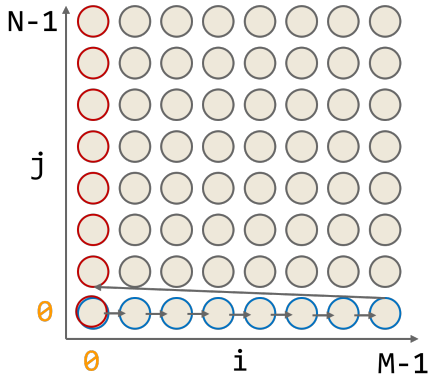
Transformations: Loop Tiling



Original:

```
for(j=0;j<N;j++)
    for(i=0;i<M;i++)
        B[j][i]+=A[i]+C[j];
```

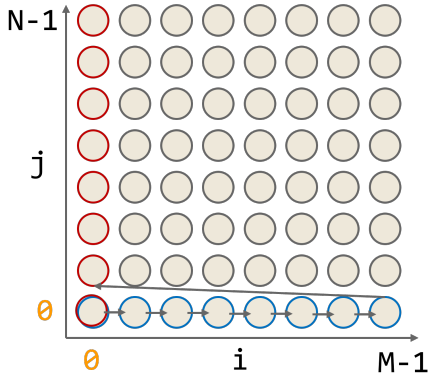
Transformations: Loop Tiling



Original:

```
for(j=0;j<N;j++)
    for(i=0;i<M;i++)
        B[j][i]+=A[i]+C[j];
```

Transformations: Loop Tiling

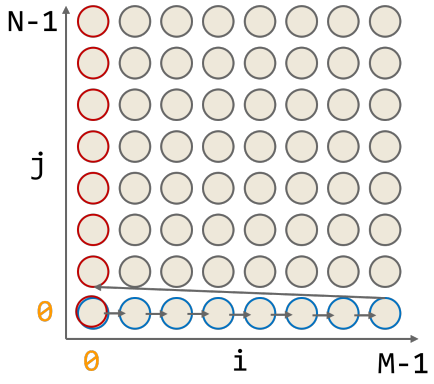


Original:

```
for(j=0;j<N;j++)
    for(i=0;i<M;i++)
        B[j][i]+=A[i]+C[j];
```

- $RD(\texttt{A[0]}) = M - 1$.

Transformations: Loop Tiling

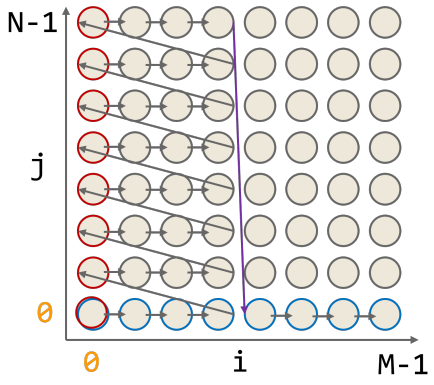


Original:

```
for(j=0;j<N;j++)
    for(i=0;i<M;i++)
        B[j][i]+=A[i]+C[j];
```

- $RD(\texttt{A[0]}) = M-1$.
- $RD(\texttt{C[0]}) = 0$.

Transformations: Loop Tiling



Original:

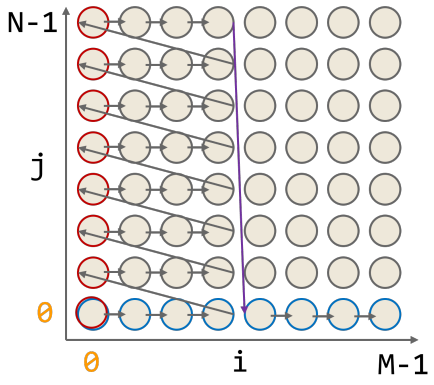
```
for(j=0;j<N;j++)
    for(i=0;i<M;i++)
        B[j][i]+=A[i]+C[j];
```

- $RD(\texttt{A[0]}) = M-1$.
- $RD(\texttt{C[0]}) = 0$.

Tiled with tile size `Ti`:

```
for(ii=0;ii<M;ii+=Ti)
    for(j=0;j<N;j++)
        for(i=ii;i<ii+Ti;i++)
            B[j][i]+=A[i]+C[j];
```

Transformations: Loop Tiling



Original:

```
for(j=0;j<N;j++)
    for(i=0;i<M;i++)
        B[j][i]+=A[i]+C[j];
```

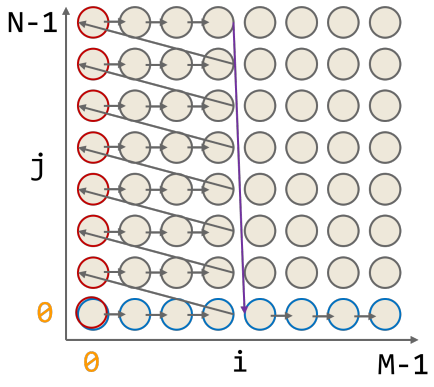
- $RD(\texttt{A[0]}) = M-1$.
- $RD(\texttt{C[0]}) = 0$.

Tiled with tile size `Ti`:

```
for(ii=0;ii<M;ii+=Ti)
    for(j=0;j<N;j++)
        for(i=ii;i<ii+Ti;i++)
            B[j][i]+=A[i]+C[j];
```

- $RD(\texttt{A[0]}) = Ti-1$.

Transformations: Loop Tiling



Original:

```
for(j=0;j<N;j++)
    for(i=0;i<M;i++)
        B[j][i]+=A[i]+C[j];
```

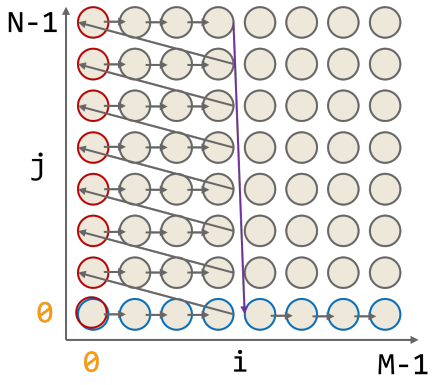
- $RD(\texttt{A[0]}) = M-1$.
- $RD(\texttt{C[0]}) = 0$.

Tiled with tile size `Ti`:

```
for(ii=0;ii<M;ii+=Ti)
    for(j=0;j<N;j++)
        for(i=ii;i<ii+Ti;i++)
            B[j][i]+=A[i]+C[j];
```

- $RD(\texttt{A[0]}) = Ti-1$.
- $RD(\texttt{C[0]}) = 0$ or $N-1$.

Transformations: Loop Tiling



More balanced re-use distance!

Original:

```
for(j=0;j<N;j++)
    for(i=0;i<M;i++)
        B[j][i]+=A[i]+C[j];
```

- $RD(\texttt{A[0]}) = M - 1$.
- $RD(\texttt{C[0]}) = 0$.

Tiled with tile size `Ti`:

```
for(ii=0;ii<M;ii+=Ti)
    for(j=0;j<N;j++)
        for(i=ii;i<ii+Ti;i++)
            B[j][i]+=A[i]+C[j];
```

- $RD(\texttt{A[0]}) = Ti - 1$.
- $RD(\texttt{C[0]}) = 0$ or $N - 1$.

Loop Tiling Example: Matrix Multiplication

Original

```
for(i=0;i<Ni;i++)
    for(j=0;j<Nj;j++)
        for(k=0;j<Nk;k++)
            C[i][j]+=A[i][k]*B[k][j];
```

Loop Tiling Example: Matrix Multiplication

Original

```
for(i=0;i<Ni;i++)
    for(j=0;j<Nj;j++)
        for(k=0;j<Nk;k++)
            C[i][j]+=A[i][k]*B[k][j];
```

Tiled

```
for(i=0;i<Ni;i++)
    for(jj=0;jj<Nj;jj+=Tj)
        for(k=0;j<Nk;k++)
            for(j=jj;j<j+Tj;j++)
                C[i][j]+=A[i][k]*B[k][j];
```

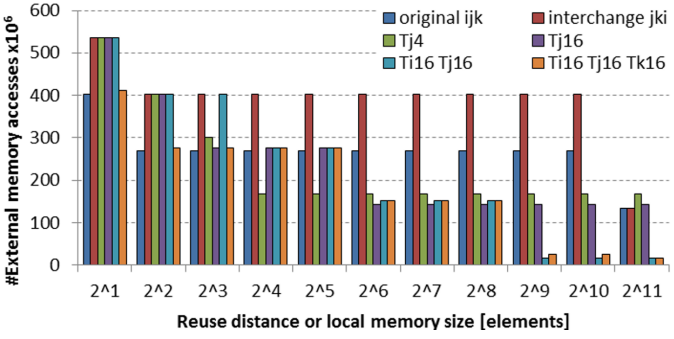
Loop Tiling Example: Matrix Multiplication

Original

```
for(i=0;i<Ni;i++)
    for(j=0;j<Nj;j++)
        for(k=0;j<Nk;k++)
            C[i][j]+=A[i][k]*B[k][j];
```

Tiled

```
for(i=0;i<Ni;i++)
    for(jj=0;jj<Nj;jj+=Tj)
        for(k=0;j<Nk;k++)
            for(j=jj;j<j+Tj;j++)
                C[i][j]+=A[i][k]*B[k][j];
```



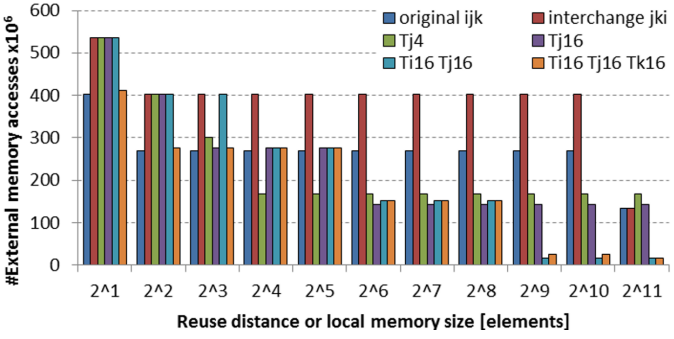
Loop Tiling Example: Matrix Multiplication

Original

```
for(i=0;i<Ni;i++)
    for(j=0;j<Nj;j++)
        for(k=0;j<Nk;k++)
            C[i][j]+=A[i][k]*B[k][j];
```

Tiled

```
for(i=0;i<Ni;i++)
    for(jj=0;jj<Nj;jj+=Tj)
        for(k=0;j<Nk;k++)
            for(j=jj;j<j+Tj;j++)
                C[i][j]+=A[i][k]*B[k][j];
```



Search space is **very large**:

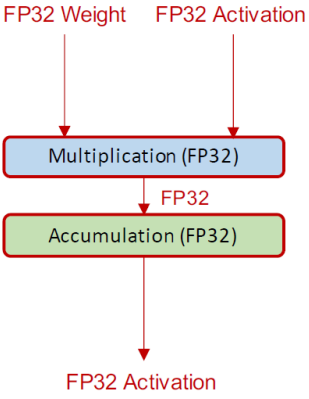
- Any combination of three loops can be tiled
- Each tile can have `Nx` different sizes

Time For A Break

"*Any intelligent fool can make things bigger, more complex, and more violent. It takes a touch of genius – and a lot of courage – to move in the opposite direction.*"

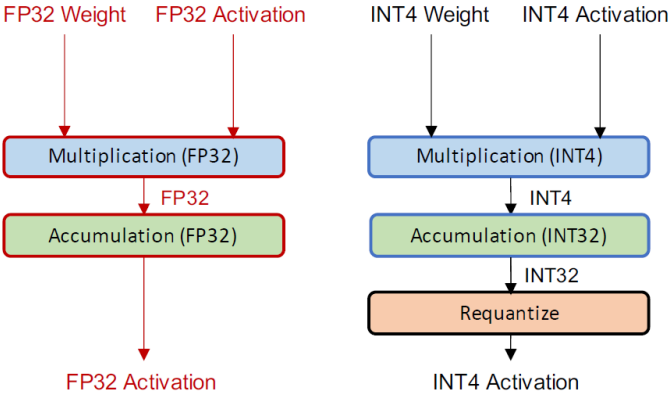
– Ernst F. Schumacher, "Small Is Beautiful: A Study of Economics As If People Mattered," 1973.

From Floating-Point to Integer



- So far, you used 32/64-bit **floating-point** arithmetic
- **Accurate** but **complex**

From Floating-Point to Integer

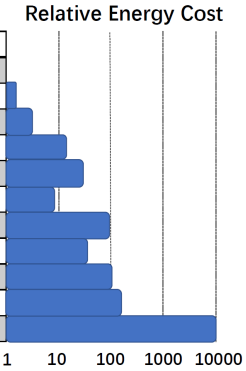


- So far, you used 32/64-bit **floating-point** arithmetic
- **Accurate** but **complex**

- **Quantization** enables less complex **integer** arithmetic
- Care needs to be taken to **limit accuracy loss**!

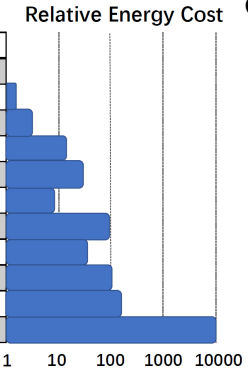
Energy Consumption Per Operation

Operation:	Energy(pJ):
8b Add	0.03
16b Add	0.05
32b Add	0.1
16b FP Add	0.4
32b FP Add	0.9
8b Mult	0.2
32b Mult	3.1
16b FP Mult	1.1
32b FP Mult	3.7
32b SRAM Read (8kb)	5.0
32b DRAM Read	640



Energy Consumption Per Operation

Operation:	Energy(pJ):
8b Add	0.03
16b Add	0.05
32b Add	0.1
16b FP Add	0.4
32b FP Add	0.9
8b Mult	0.2
32b Mult	3.1
16b FP Mult	1.1
32b FP Mult	3.7
32b SRAM Read (8kb)	5.0
32b DRAM Read	640

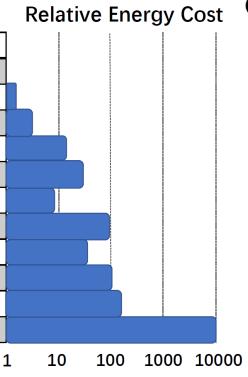


Observations:

1. Memory access is very expensive, avoid if possible (first part today)

Energy Consumption Per Operation

Operation:	Energy(pJ):
8b Add	0.03
16b Add	0.05
32b Add	0.1
16b FP Add	0.4
32b FP Add	0.9
8b Mult	0.2
32b Mult	3.1
16b FP Mult	1.1
32b FP Mult	3.7
32b SRAM Read (8kb)	5.0
32b DRAM Read	640

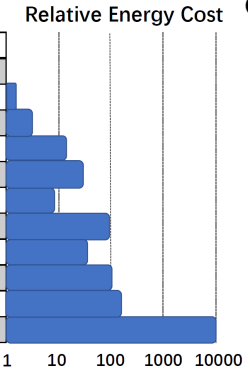


Observations:

1. Memory access is very expensive, avoid if possible (first part today)
2. Floating-point is much more expensive than integer

Energy Consumption Per Operation

Operation:	Energy(pJ):
8b Add	0.03
16b Add	0.05
32b Add	0.1
16b FP Add	0.4
32b FP Add	0.9
8b Mult	0.2
32b Mult	3.1
16b FP Mult	1.1
32b FP Mult	3.7
32b SRAM Read (8kb)	5.0
32b DRAM Read	640

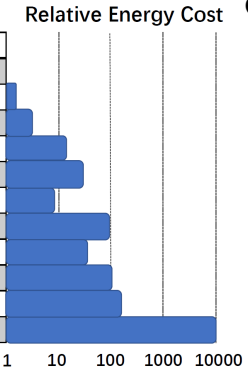


Observations:

1. Memory access is very expensive, avoid if possible (first part today)
2. Floating-point is much more expensive than integer
3. Multiplications are much more expensive than addditions

Energy Consumption Per Operation

Operation:	Energy(pJ):
8b Add	0.03
16b Add	0.05
32b Add	0.1
16b FP Add	0.4
32b FP Add	0.9
8b Mult	0.2
32b Mult	3.1
16b FP Mult	1.1
32b FP Mult	3.7
32b SRAM Read (8kb)	5.0
32b DRAM Read	640

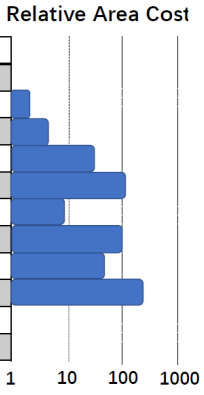


Observations:

1. Memory access is very expensive, avoid if possible (first part today)
2. Floating-point is much more expensive than integer
3. Multiplications are much more expensive than addditions
4. Lower bit-width $\Rightarrow$ lower energy (linear for addition, quadratic for multiplication)

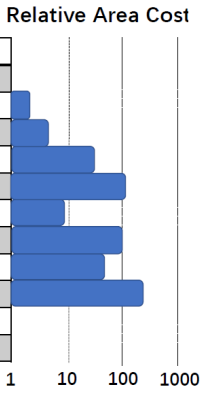
Area Per Operation and Operations Per Second

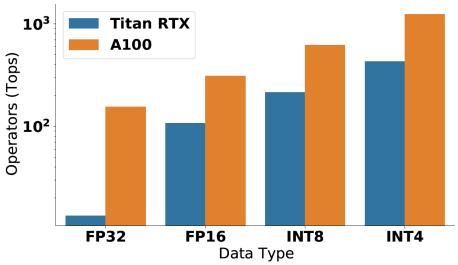
Operation:	Area(μm^2):
8b Add	36
16b Add	67
32b Add	137
16b FP Add	1360
32b FP Add	4184
8b Mult	282
32b Mult	3495
16b FP Mult	1640
32b FP Mult	7700
32b SRAM Read (8kb)	N/A
32b DRAM Read	N/A



Area Per Operation and Operations Per Second

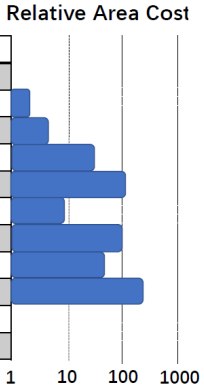
Operation:	Area(μm^2):
8b Add	36
16b Add	67
32b Add	137
16b FP Add	1360
32b FP Add	4184
8b Mult	282
32b Mult	3495
16b FP Mult	1640
32b FP Mult	7700
32b SRAM Read (8kb)	N/A
32b DRAM Read	N/A

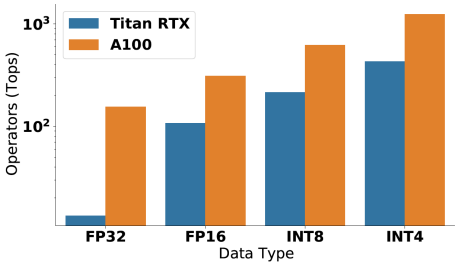




Area Per Operation and Operations Per Second

Operation:	Area(μm^2):
8b Add	36
16b Add	67
32b Add	137
16b FP Add	1360
32b FP Add	4184
8b Mult	282
32b Mult	3495
16b FP Mult	1640
32b FP Mult	7700
32b SRAM Read (8kb)	N/A
32b DRAM Read	N/A





Why Quantize?

Quantization 1) enables integer arithmetic and 2) reduces bit-width: **very beneficial**!

Quantization Loss

Example: Portray of Ada Lovelace, often regarded as **the first computer programmer**

E. Le Morvan, CC BY-SA 4.0, Wikimedia Commons

Original image

Quantization Loss

Example: Portray of Ada Lovelace, often regarded as **the first computer programmer**

E. Le Morvan, CC BY-SA 4.0, Wikimedia Commons

Original image

4-bit quantization

Quantization Loss

Example: Portray of Ada Lovelace, often regarded as **the first computer programmer**

E. Le Morvan, CC BY-SA 4.0, Wikimedia Commons

Original image

4-bit quantization

3-bit quantization

Quantization Loss

Example: Portray of Ada Lovelace, often regarded as **the first computer programmer**

Original image

4-bit quantization

3-bit quantization

2-bit quantization

Quantization Loss

Example: Portray of Ada Lovelace, often regarded as **the first computer programmer**

Original image

4-bit quantization

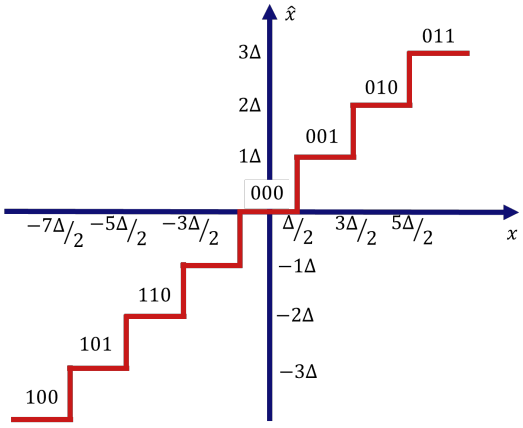
3-bit quantization

2-bit quantization

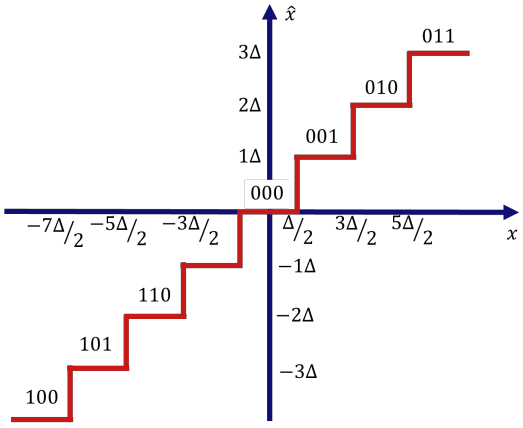
Fortunately, neural networks are also **very robust** to quantization!

Uniform Quantization

- Simplest form of quantization



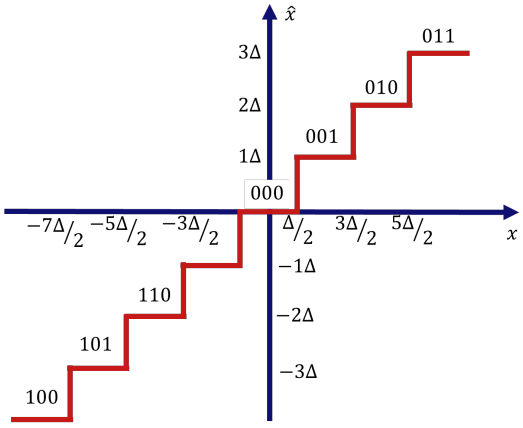
Uniform Quantization



- Simplest form of quantization
- Quantization function:

$$\hat{x} = \text{round}\left(\frac{x}{\Delta}\right) \cdot \Delta$$

Uniform Quantization

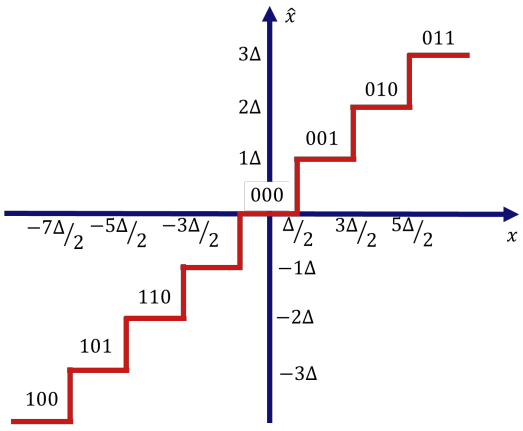


- Simplest form of quantization
- Quantization function:

$$\hat{x} = \text{round}\left(\frac{x}{\Delta}\right) \cdot \Delta$$

- Step size Δ determined by range of x

Uniform Quantization



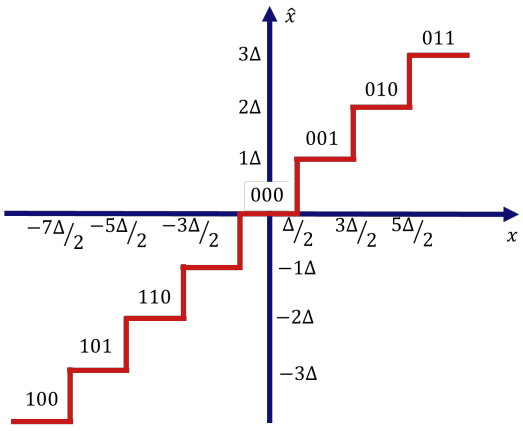
- Simplest form of quantization
- Quantization function:

$$\hat{x} = \text{round}\left(\frac{x}{\Delta}\right) \cdot \Delta$$

- Step size Δ determined by range of x
- In practice, we store:

$$\hat{x} = \text{round}\left(\frac{x}{\Delta}\right)$$

Uniform Quantization



- Simplest form of quantization
- Quantization function:

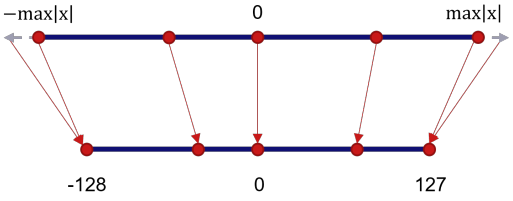
$$\hat{x} = \text{round}\left(\frac{x}{\Delta}\right) \cdot \Delta$$

- Step size Δ determined by range of x
- In practice, we store:

$$\hat{x} = \text{round}\left(\frac{x}{\Delta}\right)$$

- **Variants** we discuss:
  1. **Symmetric** or **asymmetric**
  2. **Per-layer** or **per-channel**

Symmetric Uniform Quantization



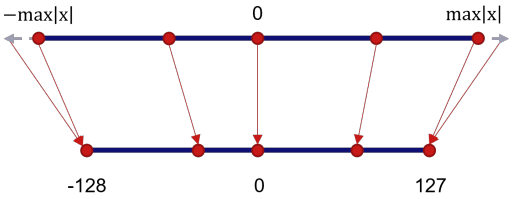
Symmetric uniform quantization with $n = 8$ bits

- Range is symmetric around zero
- Step size is given by:

$$\Delta = \frac{2\max(|x|)}{2^n - 1},$$

where n is number of quantization bits

Symmetric Uniform Quantization



Symmetric uniform quantization with $n = 8$ bits

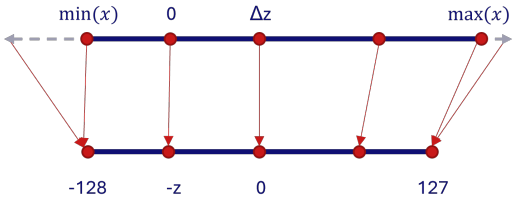
- Range is symmetric around zero
- Step size is given by:

$$\Delta = \frac{2\max(|x|)}{2^n - 1},$$

where n is number of quantization bits

- Equivalent to conventional fixed-point format when Δ is a power of two

Asymmetric Uniform Quantization



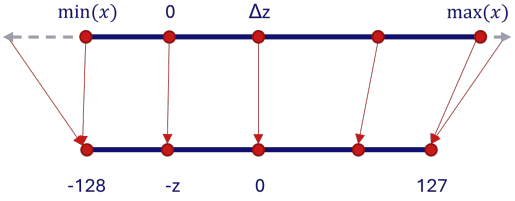
Asymmetric uniform quantization with $n = 8$ bits

- Range is **not** symmetric around zero
- Step size is given by:

$$\Delta = \frac{\max(x) - \min(x)}{2^n - 1},$$

where n is number of quantization bits

Asymmetric Uniform Quantization



Asymmetric uniform quantization with $n = 8$ bits

- Range is **not** symmetric around zero
- Step size is given by:

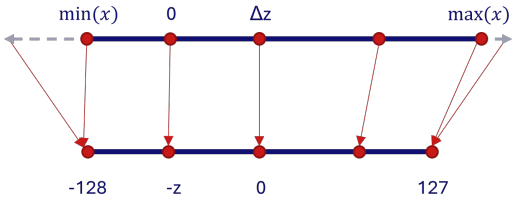
$$\Delta = \frac{\max(x) - \min(x)}{2^n - 1},$$

where n is number of quantization bits

- Use a zero-point z to shift the distribution:

$$\hat{x} = \text{round}\left(\frac{x}{\Delta}\right) - z$$

Asymmetric Uniform Quantization



Asymmetric uniform quantization with $n = 8$ bits

- Range is **not** symmetric around zero
- Step size is given by:

$$\Delta = \frac{\max(x) - \min(x)}{2^n - 1},$$

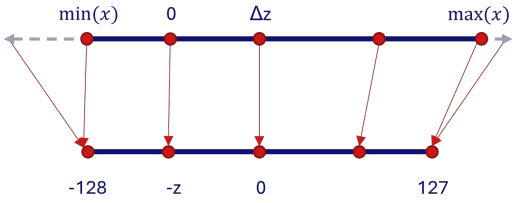
where n is number of quantization bits

- Use a zero-point z to shift the distribution:

$$\hat{x} = \text{round}\left(\frac{x}{\Delta}\right) - z$$

- In practice, z is chosen so that zero is exactly represented by an integer

Asymmetric Uniform Quantization



Asymmetric uniform quantization with $n = 8$ bits

Useful for ReLU-style activations!

- Range is **not** symmetric around zero
- Step size is given by:

$$\Delta = \frac{\max(x) - \min(x)}{2^n - 1},$$

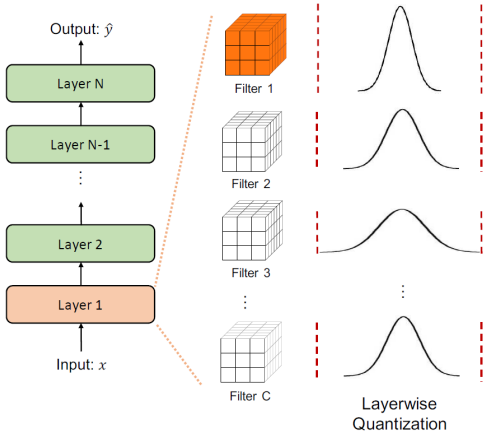
where n is number of quantization bits

- Use a zero-point z to shift the distribution:

$$\hat{x} = \text{round}\left(\frac{x}{\Delta}\right) - z$$

- In practice, z is chosen so that zero is exactly represented by an integer

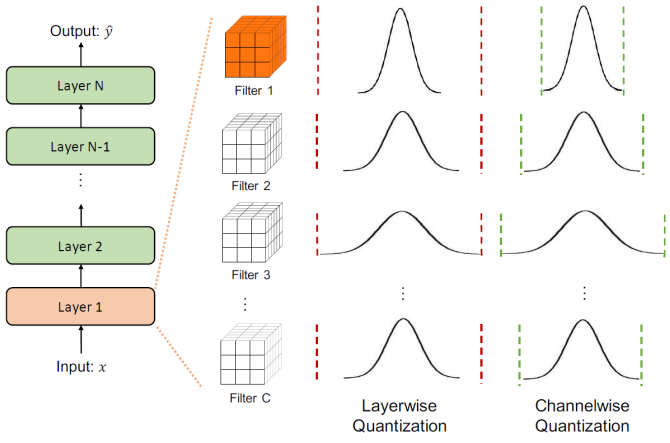
Layer-Wise vs Channel-Wise Quantization



Quantization granularity:

- **Layer-wise:** entire layer uses same quantization parameters

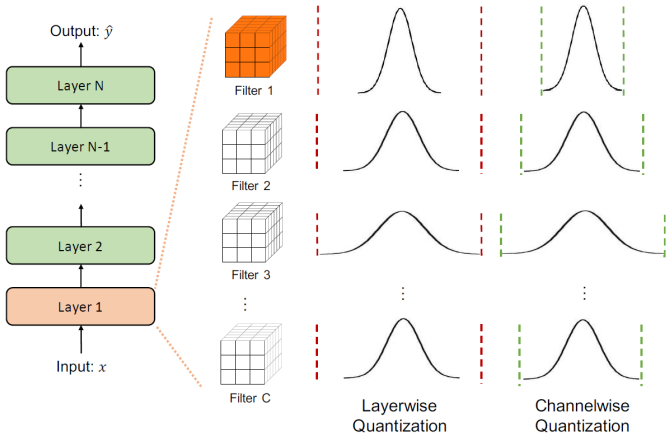
Layer-Wise vs Channel-Wise Quantization



Quantization granularity:

- **Layer-wise:** entire layer uses same quantization parameters
- **Channel-wise:** each channel uses different quantization parameters

Layer-Wise vs Channel-Wise Quantization



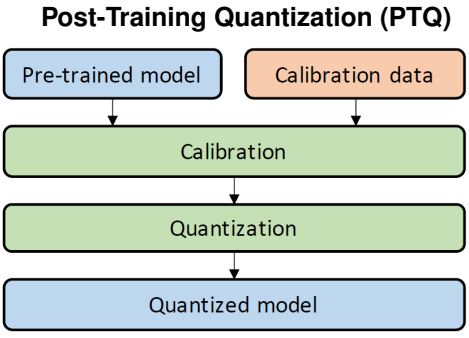
Quantization granularity:

- **Layer-wise:** entire layer uses same quantization parameters
- **Channel-wise:** each channel uses different quantization parameters

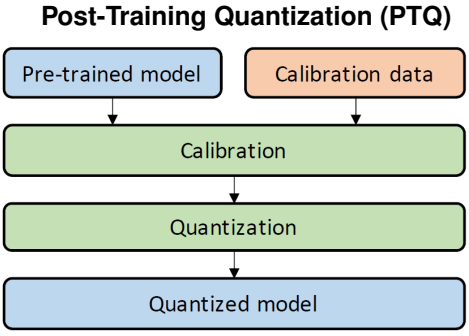
Variants

Other approaches are: 1) group-wise quantization and 2) subchannel-wise quantization.

Post-Training Quantization vs Quantization-Aware Training

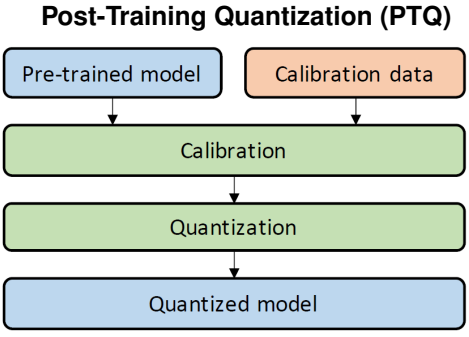


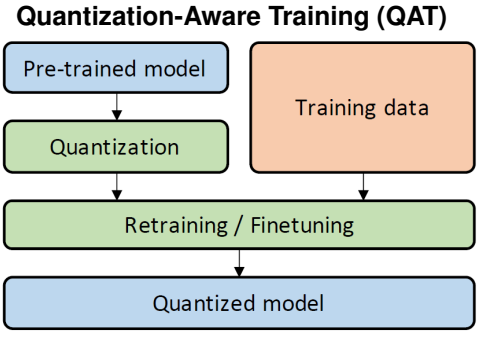
Post-Training Quantization vs Quantization-Aware Training



- **Simple**, no re-training needed
- Performance can be **poor**

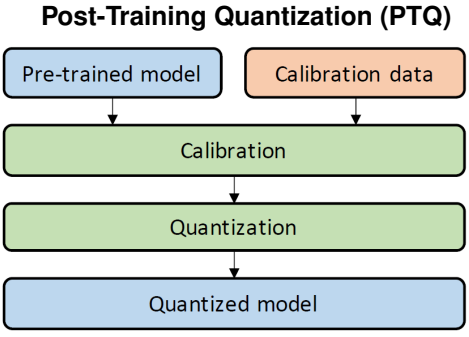
Post-Training Quantization vs Quantization-Aware Training



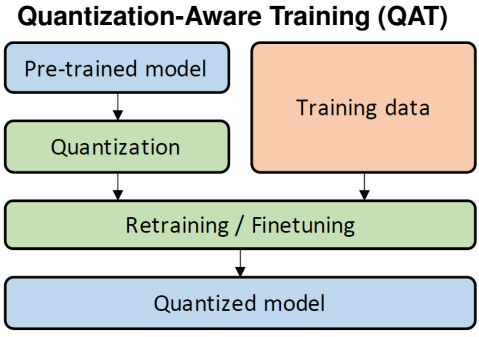


- **Simple**, no re-training needed
- Performance can be **poor**

Post-Training Quantization vs Quantization-Aware Training

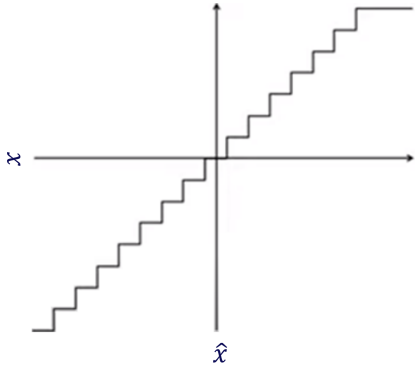


- **Simple**, no re-training needed
- Performance can be **poor**



- **Complex**, multiple re-training runs
- Performance is **superior** to PTQ

Straight-Through Estimator For Training

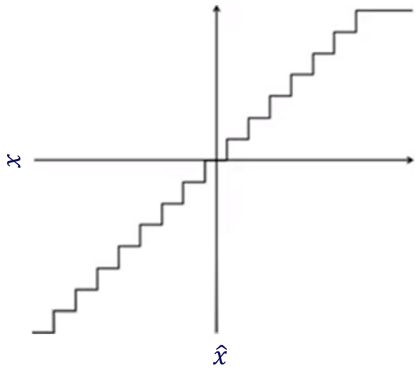


- Forward pass:

$$\hat{x} = Q(x),$$

where $Q(x)$ is the quantization function

Straight-Through Estimator For Training



- Forward pass:

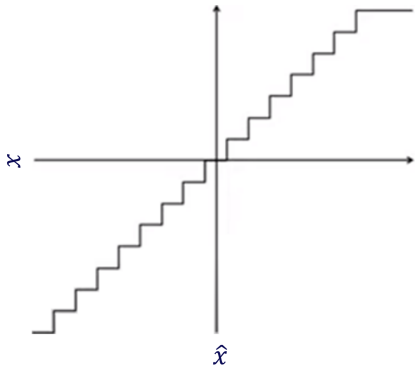
$$\hat{x} = Q(x),$$

  where $Q(x)$ is the quantization function

- For backpropagation, we need:

$$\frac{\partial Q(x)}{\partial x}$$

Straight-Through Estimator For Training



- Forward pass:

$$\hat{x} = Q(x),$$

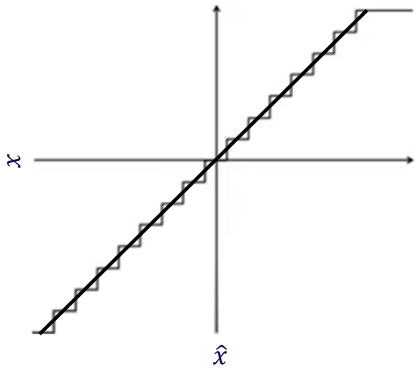
where $Q(x)$ is the quantization function

- For backpropagation, we need:

$$\frac{\partial Q(x)}{\partial x} = 0$$

...which is zero almost everywhere!

Straight-Through Estimator For Training



- Forward pass:

$$\hat{x} = Q(x),$$

where $Q(x)$ is the quantization function

- For backpropagation, we need:

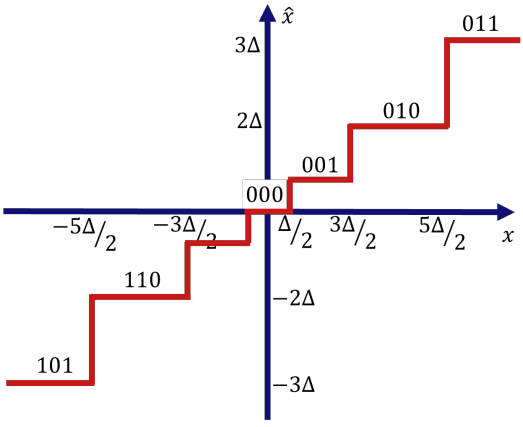
$$\frac{\partial Q(x)}{\partial x} = 0$$

...which is zero almost everywhere!

Solution

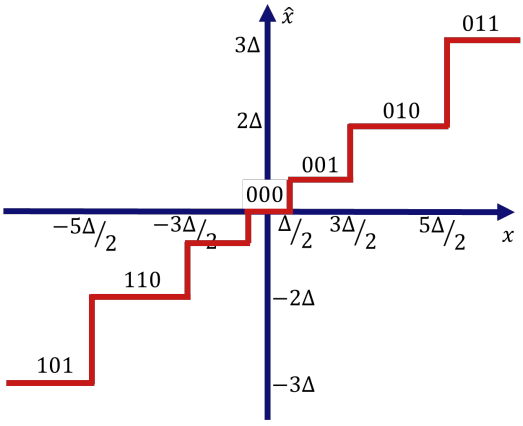
Straight-through estimator: simply replaces quantization function by straight line!

Advanced Quantization Topics: Non-Uniform Quantization



- Weights/activations are usually not uniformly distributed
- More sampling points where more values reside

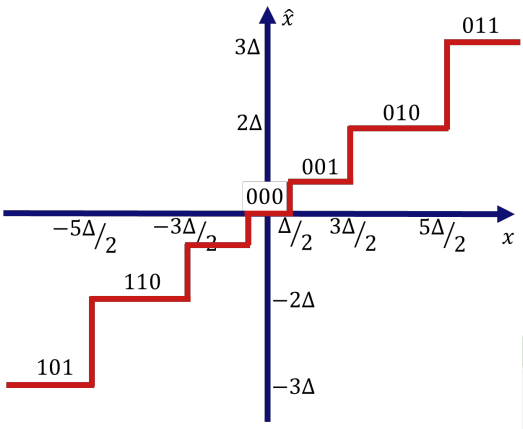
Advanced Quantization Topics: Non-Uniform Quantization



- Weights/activations are usually not uniformly distributed
- More sampling points where more values reside

- Variable step size Δ
- Typically some regularity in step size:
  1. Power-of-two step
  2. Logarithmic step

Advanced Quantization Topics: Non-Uniform Quantization



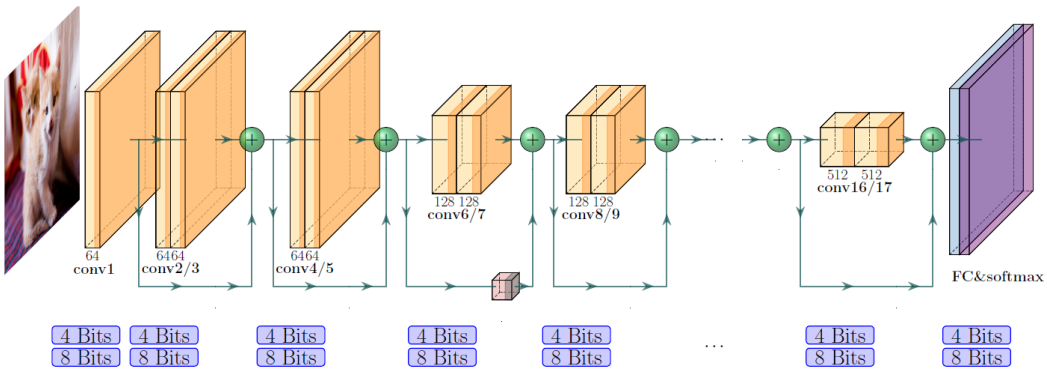
- Weights/activations are usually not uniformly distributed
- More sampling points where more values reside

- Variable step size Δ
- Typically some regularity in step size:
  1. Power-of-two step
  2. Logarithmic step

Drawback

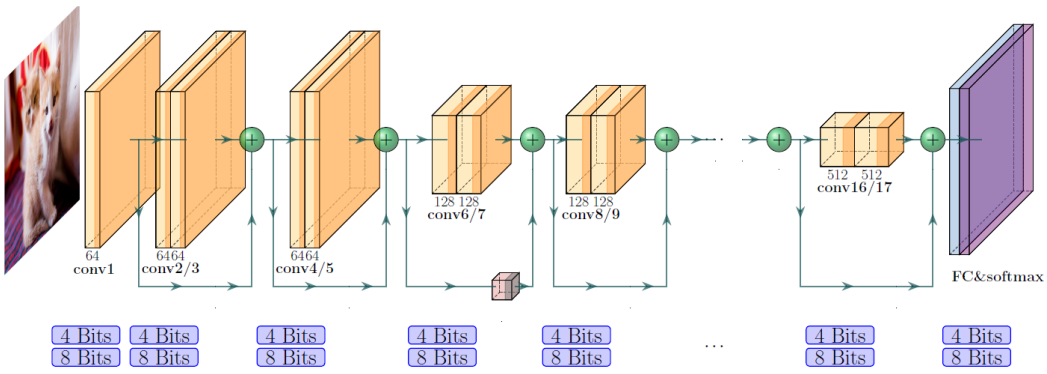
Simple and efficient integer arithmetic hardware is no longer possible!

Advanced Quantization Topics: Mixed Precision Quantization



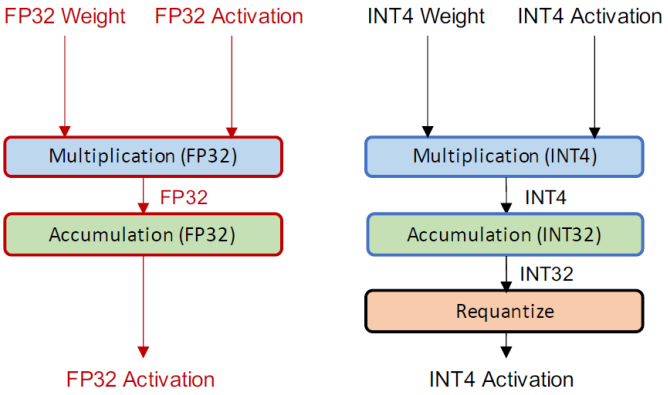
- Each layer can have different carefully selected quantization.

Advanced Quantization Topics: Mixed Precision Quantization

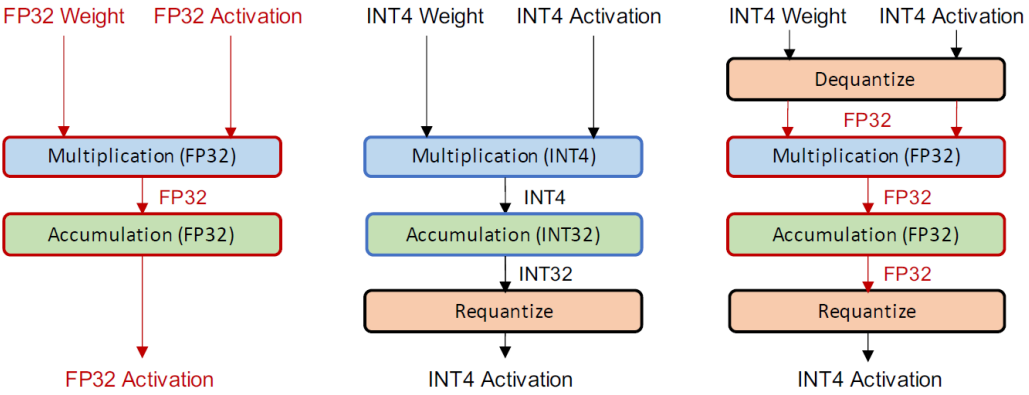


- Each layer can have different carefully selected quantization.
- **Huge design space**, how to choose?

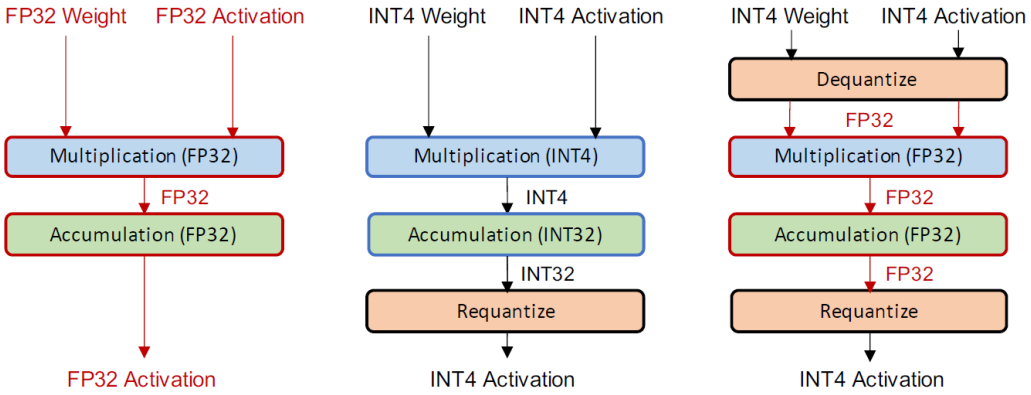
A Word of Caution



A Word of Caution



A Word of Caution



- Without explicit support for quantized operations, quantization may make your implementation **slower**!

Advanced Quantization Topics: Extreme Quantization

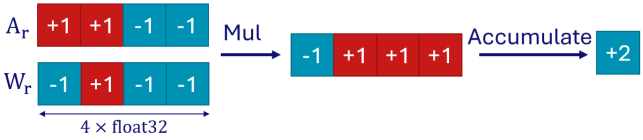
- Usually, 8-bit quantization works well. Can we be more extreme?

Advanced Quantization Topics: Extreme Quantization

- Usually, 8-bit quantization works well. Can we be more extreme?
- Yes! Extremest example: quantization to ± 1
- Binary representation: $+1 \rightarrow 1$ and $-1 \rightarrow 0$

Advanced Quantization Topics: Extreme Quantization

- Usually, 8-bit quantization works well. Can we be more extreme?
- Yes! Extremest example: quantization to ± 1
- Binary representation: $+1 \rightarrow 1$ and $-1 \rightarrow 0$



Advanced Quantization Topics: Extreme Quantization

- Usually, 8-bit quantization works well. Can we be more extreme?
- Yes! Extremest example: quantization to ± 1
- Binary representation: $+1 \rightarrow 1$ and $-1 \rightarrow 0$

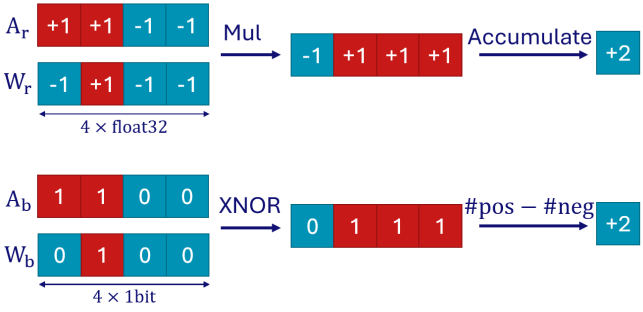


Image credit: Floran de Putter

Advanced Quantization Topics: Extreme Quantization

- Usually, 8-bit quantization works well. Can we be more extreme?
- Yes! Extremest example: quantization to ± 1
- Binary representation: $+1 \rightarrow 1$ and $-1 \rightarrow 0$

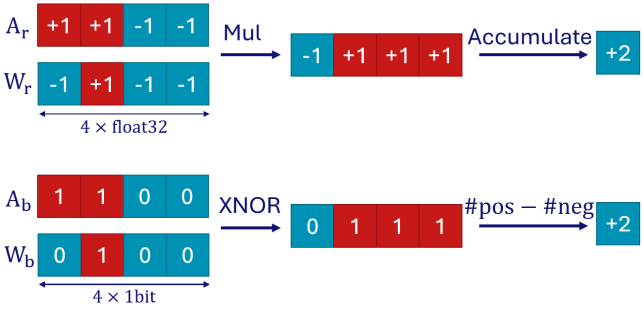


Image credit: Floran de Putter

- **But:** Quantization loss can be large and specialized hardware is required!

Advanced Quantization Topics: Extreme Quantization

- Usually, 8-bit quantization works well. Can we be more extreme?
- Yes! Extremest example: quantization to ± 1
- Binary representation: $+1 \rightarrow 1$ and $-1 \rightarrow 0$

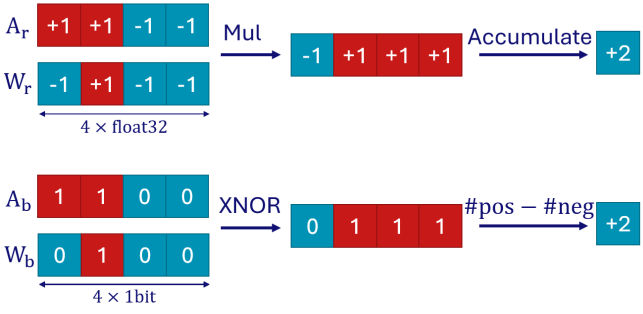


Image credit: Floran de Putter

- **But:** Quantization loss can be large and specialized hardware is required!
- Less extreme solutions are also possible, e.g., ternary quantization to $\{-1, 0, +1\}$

Conclusion

Summary:

1. **Loop transformations:**
   - CNNs are “just” a nested loop.
   - Loop transformations can improve data re-use.
2. **Quantization:**
   - Many ways to quantize, including extreme binary quantization.
   - Re-training generally improves performance.
   - Hardware support for quantization is crucial.

Conclusion

Summary:

1. **Loop transformations:**
   - CNNs are “just” a nested loop.
   - Loop transformations can improve data re-use.
2. **Quantization:**
   - Many ways to quantize, including extreme binary quantization.
   - Re-training generally improves performance.
   - Hardware support for quantization is crucial.

Next time:

1. Network pruning.
2. Compact-aware network design.