DSP Computing

Chao-Tsung Huang

National Tsing Hua University
Department of Electrical Engineering

Outline

- **Fast algorithms**
- DSP for VLSI circuits

Product of integers

- Let x, y be n-digit numbers and n=2m

$$x = x_0 + x_1 \cdot b^m \qquad y = y_0 + y_1 \cdot b^m$$ (e.g. base b=2)

$$z = x \cdot y = (x_0 + x_1 b^m)(y_0 + y_1 b^m) = x_0 y_0 + (x_0 y_1 + x_1 y_0) b^m + x_1 y_1 b^{2m}$$

$$\Downarrow$$

$$\boxed{\begin{aligned} x_0 y_0 &= x_0 y_0, \\ x_0 y_1 + x_1 y_0 &= (x_0 - x_1)(y_1 - y_0) + x_0 y_0 + x_1 y_1 \\ x_1 y_1 &= x_1 y_1. \end{aligned}}$$

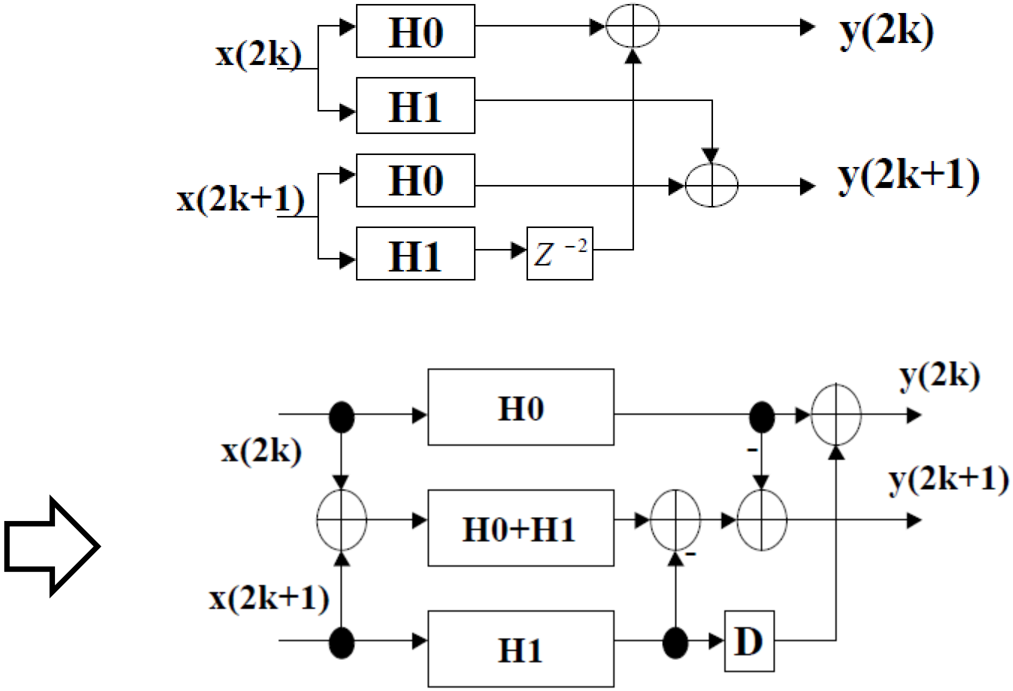
(3 m-digit operations, instead of 4)

- Recursive implementation: $n^{\log_2 3}$ operations

Ref: S. Winograd, *Arithmetic Complexity of Computations*, 1980.

Two-parallel FIR filter

- Express convolution by polynomial multiplication
  - Then partition both of x and h into even and odd parts



Ref: K. K. Parhi, *VLSI Digital Signal Processing Systems Design and Implementation*, Wiley.

Matrix-matrix multiplication

- Let A, B be $n \times n$ matrices and n=2m

$$A \times B = C \qquad \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

⇩

$$P_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22}), \qquad P_2 = (A_{21} + A_{22}) \times B_{11},$$

$$P_3 = A_{11} \times (B_{12} - B_{22}), \qquad P_4 = A_{22} \times (B_{21} - B_{11}), \qquad C_{11} = P_1 + P_4 - P_5 + P_7, \qquad C_{12} = P_3 + P_5,$$

$$P_5 = (A_{11} + A_{12}) \times B_{22}, \qquad P_6 = (A_{21} - A_{11}) \times (B_{11} + B_{12}), \qquad C_{21} = P_2 + P_4, \qquad C_{22} = P_1 + P_3 - P_2 + P_6.$$

$$P_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22}),$$

(7 $m \times m$ matrix multiplications, instead of 8)

- Recursive implementation: $n^{\log_2 7}$ multiplications

Ref: S. Winograd, *Arithmetic Complexity of Computations*, 1980.

1D FIR filter

- F(2,3): 2 outputs and 3-tap filter

Inverse transform (post-process)

$$\begin{pmatrix} z_0 & z_1 & z_2 \\ z_1 & z_2 & z_3 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{pmatrix}$$

Data transform (pre-process)

Filter transform (pre-process)

$$\Downarrow$$

$$\boxed{\begin{gathered} m_1 = (z_0 - z_2)x_0, \quad m_2 = (z_1 + z_2)((x_0 + x_1 + x_2)/2), \\ m_3 = (z_2 - z_1)((x_0 - x_1 + x_2)/2), \text{ and } m_4 = (z_1 - z_3)x_2. \end{gathered}}$$

(4 multiplications, instead of 6)

Point-wise multiplication

- Proven minimality of multiplication
  - $\mu(F(n,k)) = n + k - 1$

Ref: S. Winograd, *Arithmetic Complexity of Computations*, 1980.

Winograd convolution for 2D FIR filter

- Recap of F(2,3): 2 outputs and 3-tap filter

Filter $g = \begin{bmatrix} g_0 & g_1 & g_2 \end{bmatrix}^T$

Input $d = \begin{bmatrix} d_0 & d_1 & d_2 & d_3 \end{bmatrix}^T$

Output $Y = A^T \left[(Gg) \odot (B^T d) \right]$

Point-wise multiplication

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

Data transform

$$G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

Filter transform

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}$$

Inverse transform

- F(2x2,3x3): 16 multiplications, instead of 36

$$Y = A^T \left[\left[G g G^T \right] \odot \left[B^T d B \right] \right] A$$

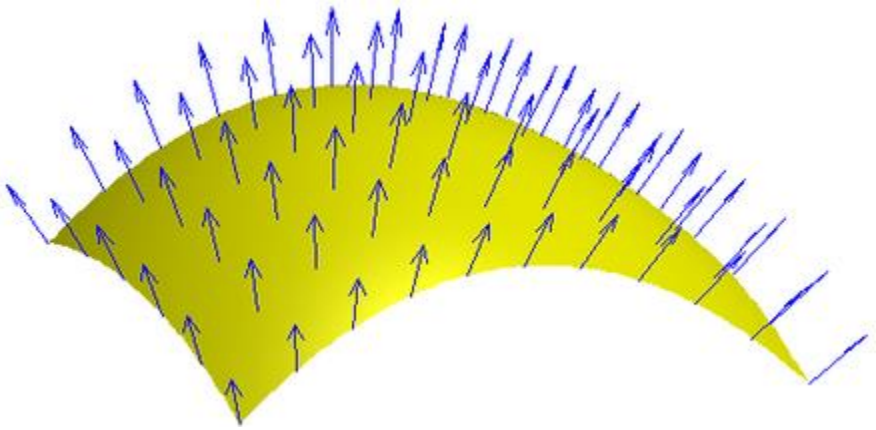
3x3 filter **4x4 input**

Overhead:
- **More adders**
- **Longer critical path**
- **Longer internal bitwidth**

Ref: “Fast algorithms for convolutional neural networks”, *CVPR*, 2016.

Fast (magic) inverse square root

- Motivation
  - Inverse square root is heavily used in computer graphics to find normalized vectors in floating point

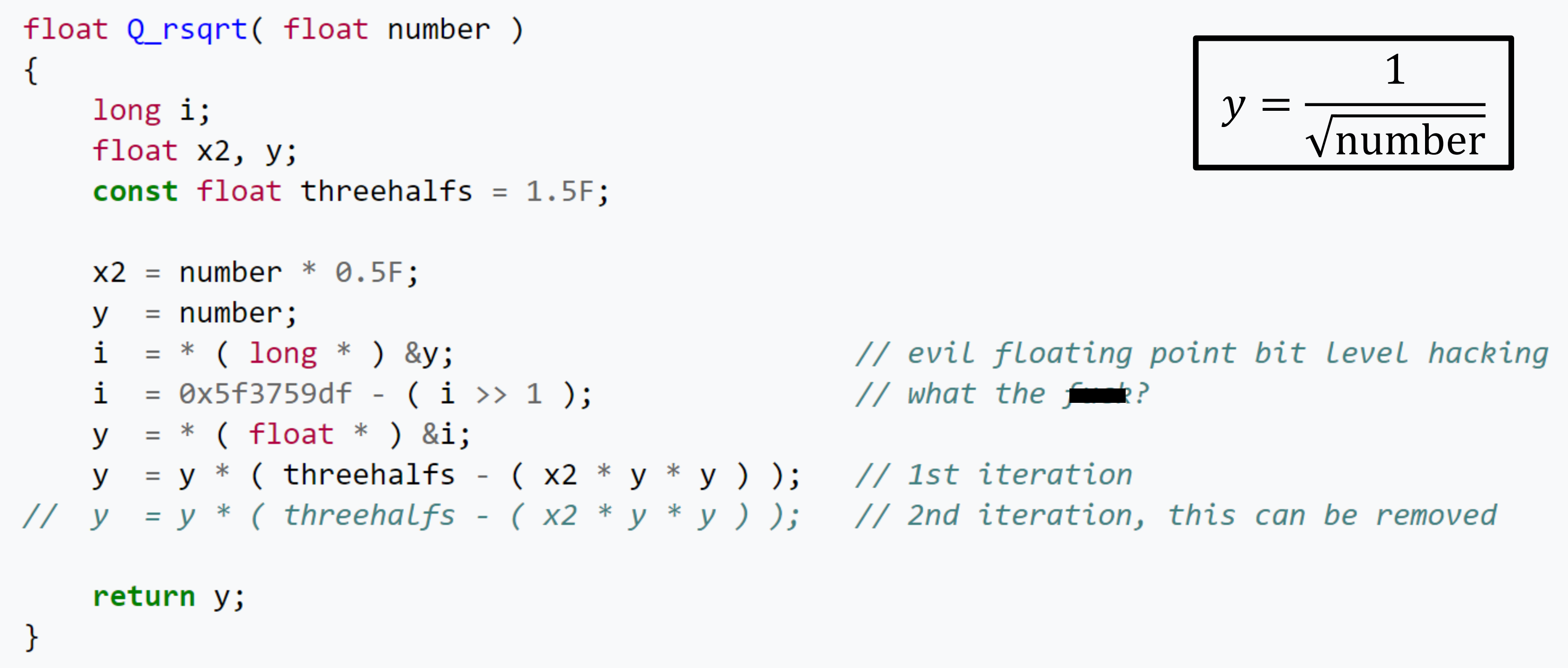


$$\hat{\boldsymbol{v}} = \boldsymbol{v}\frac{1}{\sqrt{\|\boldsymbol{v}\|^2}}$$

Ref: https://en.wikipedia.org/wiki/Fast_inverse_square_root

Fast (magic) inverse square root

- Legend
  - The following magic code was found in the source code of *Quake III Arena* (1999)



```c
float Q_rsqrt( float number )
{
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y  = number;
    i  = * ( long * ) &y;                       // evil floating point bit level hacking
    i  = 0x5f3759df - ( i >> 1 );               // what the [redacted]?
    y  = * ( float * ) &i;
    y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
//  y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration, this can be removed

    return y;
}
```

$$\boxed{y = \frac{1}{\sqrt{\text{number}}}}$$

Ref: https://en.wikipedia.org/wiki/Fast_inverse_square_root

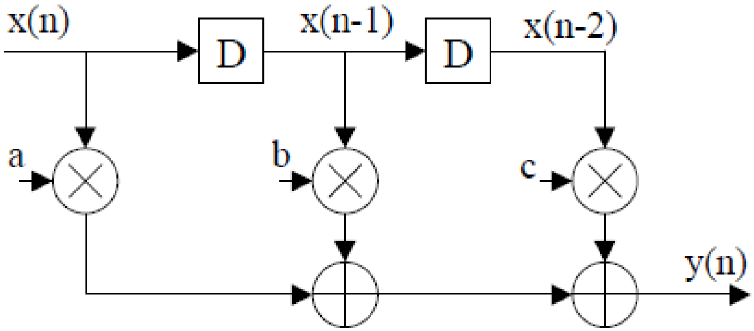
Outline

- Fast algorithms
- **DSP for VLSI circuits**
  - Pipelining
  - Parallel
  - Retiming
  - Systolic array

Pipelining

(Timing-reduction; Same throughput, longer latency)

- Cut feed-forward paths into pipelined stages

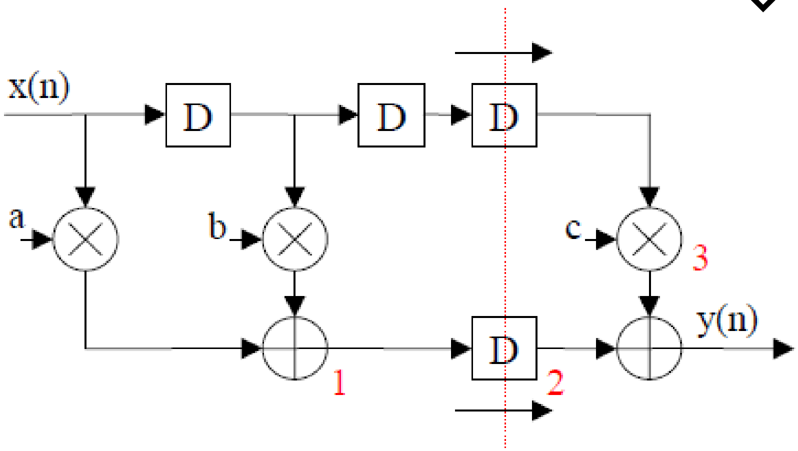


T_M : $multiplica\ tion - time$

T_A : $Addition\ - time$

T_M+2T_A

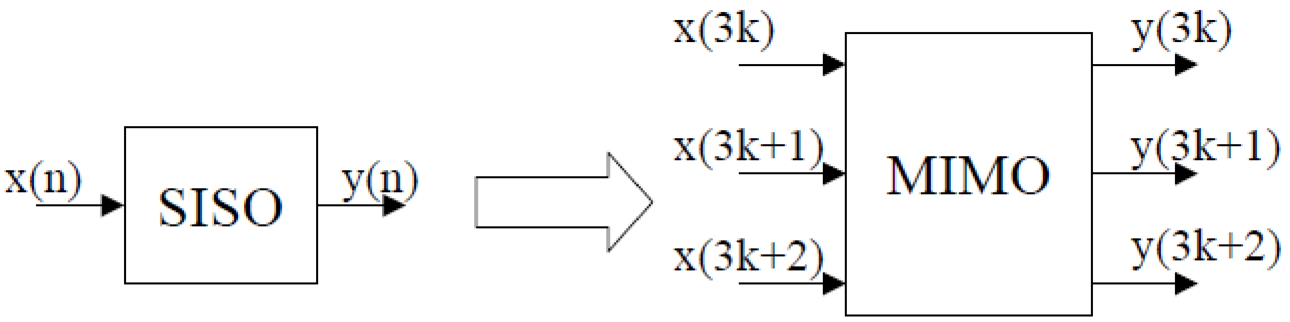
T_M+T_A



Overhead:
Latency and Pipeline registers

Parallel Processing

(Timing-relaxation; Common low-power technique)



Sequential System

3-Parallel System

$$P \propto C \cdot V^2$$

If *V* reduces faster than *C* (area), the power will be reduced.

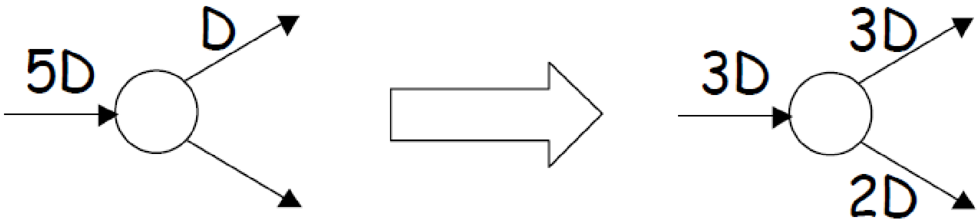
y(3k)=a*x(3k)+b*x(3k-1)+c*x(3k-2)
y(3k+1)=a*x(3k+1)+b*x(3k)+c*x(3k-1)
y(3k+2)=a*x(3k+2)+b*x(3k+1)+c*x(3k)

Area increased to 3x
(can be reduced by algorithmic strength reduction);
Timing can be prolonged to 3x

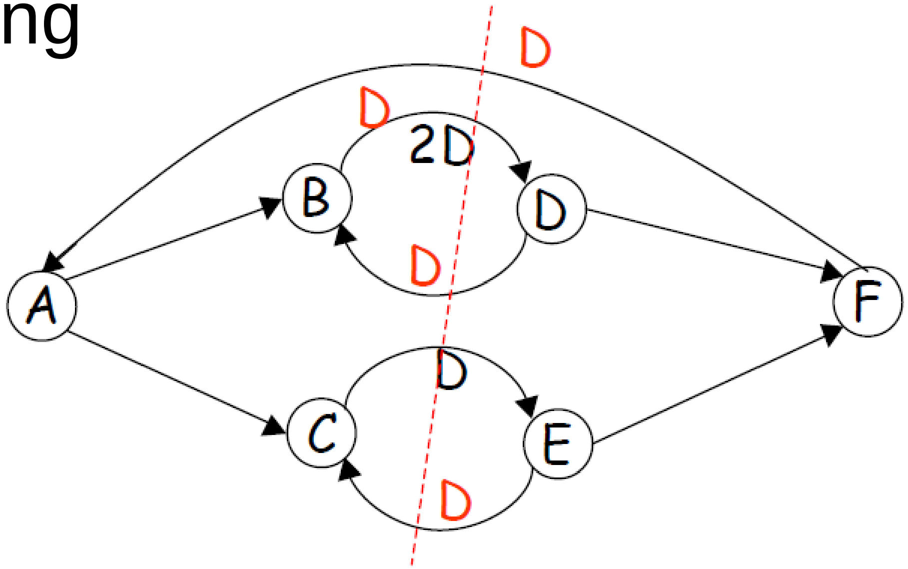
Retiming

(Timing-reduction; Same latency and throughput)

- Move existing delays around for better timing



- Generalization of pipelining

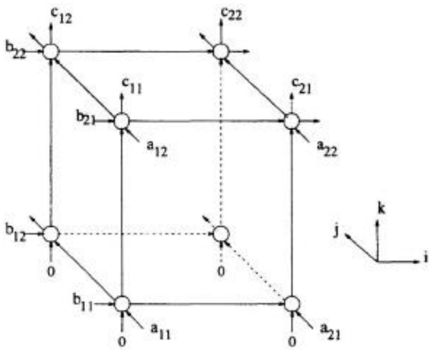


B->D(**2D**) => B->D (**D**) + D->B(**D**) + D->F (**D**)
C->E (**D**) => E->C (**D**) + E->F (**D**)
D->F (**D**) + E->F (**D**) => F->A (**D**)

Systolic Array

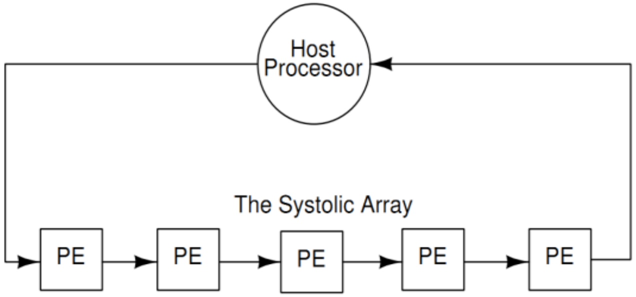
(Dimension-reduction; regularity)

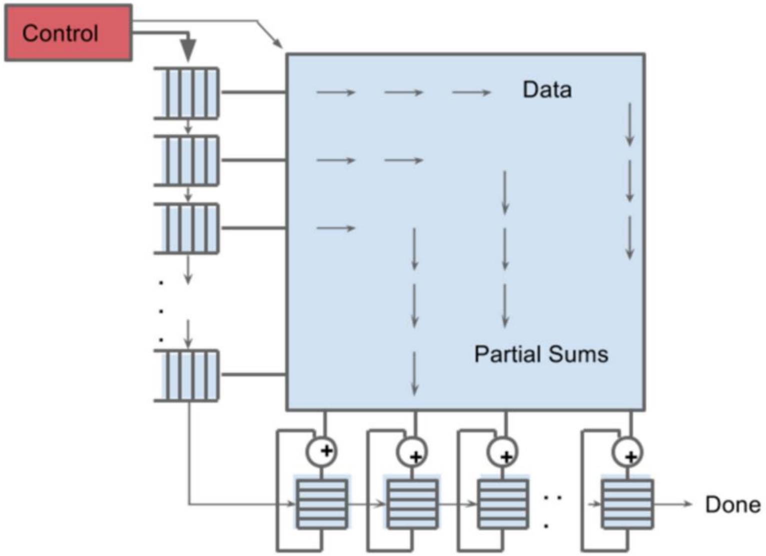
- Can project in many different ways



$$\begin{bmatrix} & \cdots & \\ \vdots & \ddots & \vdots \\ & \cdots & \end{bmatrix}\begin{bmatrix} & \cdots & \\ \vdots & \ddots & \vdots \\ & \cdots & \end{bmatrix} \Longrightarrow$$

Matrix-matrix multiplication
(3D operations)





Google TPU
(2D PE array)