ECE 6775
High-Level Digital Design Automation
Fall 2024

More Hardware Specialization

Cornell University

Announcements

- Lab 1 and Vivado HLS setup guide released
  - Complete the tool setup this week
- Paper reading this Thursday
  - A. Boutros and V. Betz, "FPGA Architecture: Principles and Progression", IEEE CAS-M 2021

Agenda

- Common hardware specialization techniques
  - More on fixed point types
  - Case study on customized memory hierarchy
  - An example of customized communication architecture
- Roofline-based performance modeling
  - Operational intensity (OI) analysis

Recap: Principles for Improving Energy Efficiency

Do less work!

– **Amortize overhead** in control and data supply across multiple instructions

Do even less work!

– **Use smaller (or simpler) data** => cheaper operations, lower storage & communication costs
– **Move data locally and directly**
    - Store data nearby in simpler memory (e.g., scratchpads are cheaper than cache)
    - Wire compute units for direct (or even combinational) communication when possible

Recap: Common HW Specialization Techniques

Lab 1

Custom Compute Units: Use complex instructions to amortize overhead (e.g., SIMD, “ASIC”-in-an-instruction)

Lab 2

Custom Numeric Types: Balance accuracy and efficiency with data types that use smaller bit widths or simpler arithmetic

Lab 3
Lab 4

Custom Memory Hierarchy: Exploit data access patterns to reduce energy per memory operation

Custom Communication Architecture: Tailor on-chip networks to data movement patterns

Recap: Fixed-Point Representation

- The positional binary encoding can also represent fractional values, by using a **fixed** position of the binary point and place values with negative exponents
  - (–) Less convenient to use in software, compared to floating point
  - (+) Much more efficient in hardware

Signed (2'c) fixed-point number

$$\begin{array}{cccccccc} \multicolumn{4}{c}{\overbrace{\hspace{10em}}^{\textbf{Integer part}\ \text{(4 bits)}}} & \multicolumn{2}{c}{\overbrace{\hspace{5em}}^{\textbf{Fractional part}\ \text{(2 bits)}}} & \\ 2^{3} & 2^{2} & 2^{1} & 2^{0} & 2^{-1} & 2^{-2} & 2'c \\ \hline 1 & 0 & 1 & 1\ \underset{\underset{\text{Binary point}}{\uparrow}}{\bullet} & 0 & 1 & = -4.75 \end{array}$$

Overflow and Quantization Issues

- When working with fixed-point types, quantization and overflow issues often arise due to the limited <u>precision</u> and <u>range</u> of representation

- **Overflow** occurs when a value exceeds the maximum representable range for the fixed-point format
  - In our class, overflow mainly concerns the integer part of the fixed-point number

- **Quantization** is needed when converting real numbers from a higher-precision format (like float) to fixed point

Common Modes for Handling Overflow (1)

- **Wrapping** or wraparound: The value wraps around within the range using modulo arithmetic
  - Efficient in hardware, as it involves simply dropping the MSB(s) of the original number

-2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2'c
$\textcircled{1}$	0	1	1	0	1	= -4.75

Dropping MSB when integer width is reduced

					2'c
0	1	1	0	1	= ??

Wrapping can cause a negative number to become positive, or a positive to negative

Common Modes for Handling Overflow (2)

- **Saturation**: The closest representable value (either maximum or minimum) is used, preventing wraparound
  - Implementing saturation in hardware requires additional logic to check for overflows and apply the clamping

-2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2’c
1	0	1	1	0	1	= -4.75

Saturation

					2’c
?	?	?	?	?	= ??

What is the saturated result for the example above?

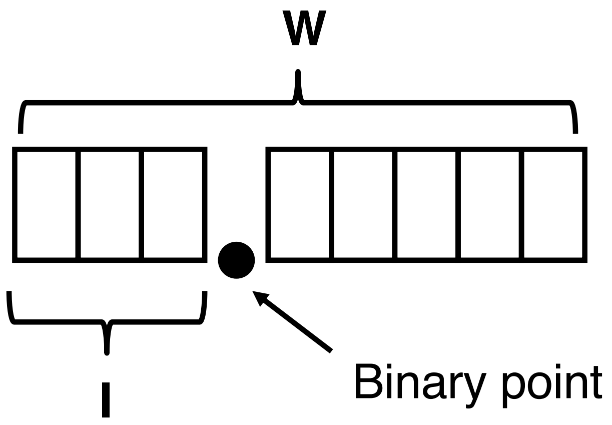
Common Quantization Modes

- **Truncation**: Cuts off the excess bits that don't fit in the target precision
  - Efficient in hardware, as it involves simply dropping the LSB(s) of the original number

- **Rounding:** Rounds the value to a representable fixed-point number, potentially reducing quantization error compared to truncation
  - Round to Nearest: The most common method, where the value is rounded to the nearest fixed-point representation
  - Round Toward Zero: Rounds towards zero, effectively truncating the fractional part
  - Round Toward Infinity: Rounds away from zero, towards positive or negative infinity

Fixed-Point Types in Vivado HLS

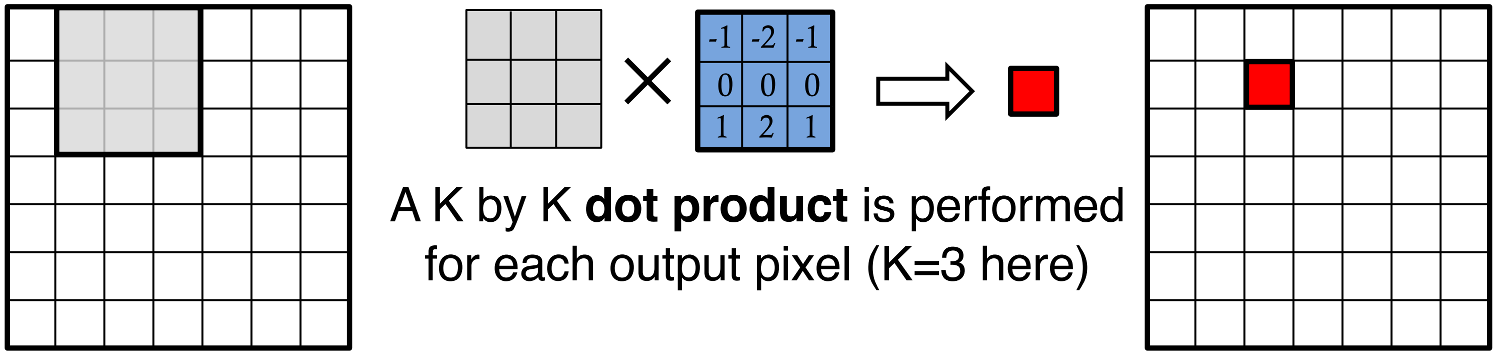
- **ap_fixed** is a templated C++ data type used for representing fixed-point numbers
  - Signed: **ap_fixed**; Unsigned: **ap_ufixed**
  - Template parameters ap_(u)fixed<W, I, Q, O>
    - W: total bitwidth
    - I: integer bitwidth
    - Q: quantization mode (optional, default is AP_TRN)
    - O: overflow mode (optional, default is AP_WRAP)



Custom Memory Hierarchy: Case Study on Convolution

- **Convolution** is pervasive in image/video processing and ML – performed over overlapping windows (aka stencils)

$$(Img \otimes f)_{\left[n+\frac{k-1}{2},m+\frac{k-1}{2}\right]} = \sum_{i=0}^{k-1}\sum_{j=0}^{k-1} Img_{[n+i][m+j]} \cdot f_{[i,j]}$$



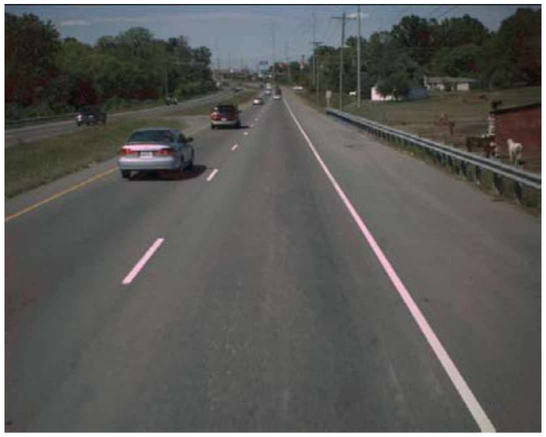
A K by K **dot product** is performed for each output pixel (K=3 here)

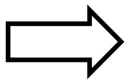
Input image frame

Output image frame

An Application of Convolution: Edge Detection

- Identifies discontinuities in an image where brightness (or image intensity) changes sharply
  - Very useful for feature extractions in computer vision



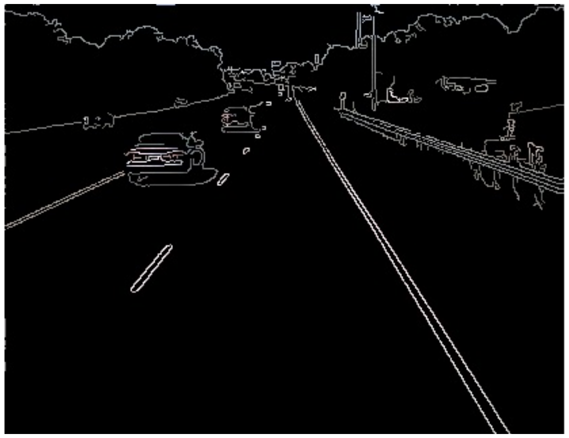


Sobel operator

$G = (G_X, G_Y)$

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

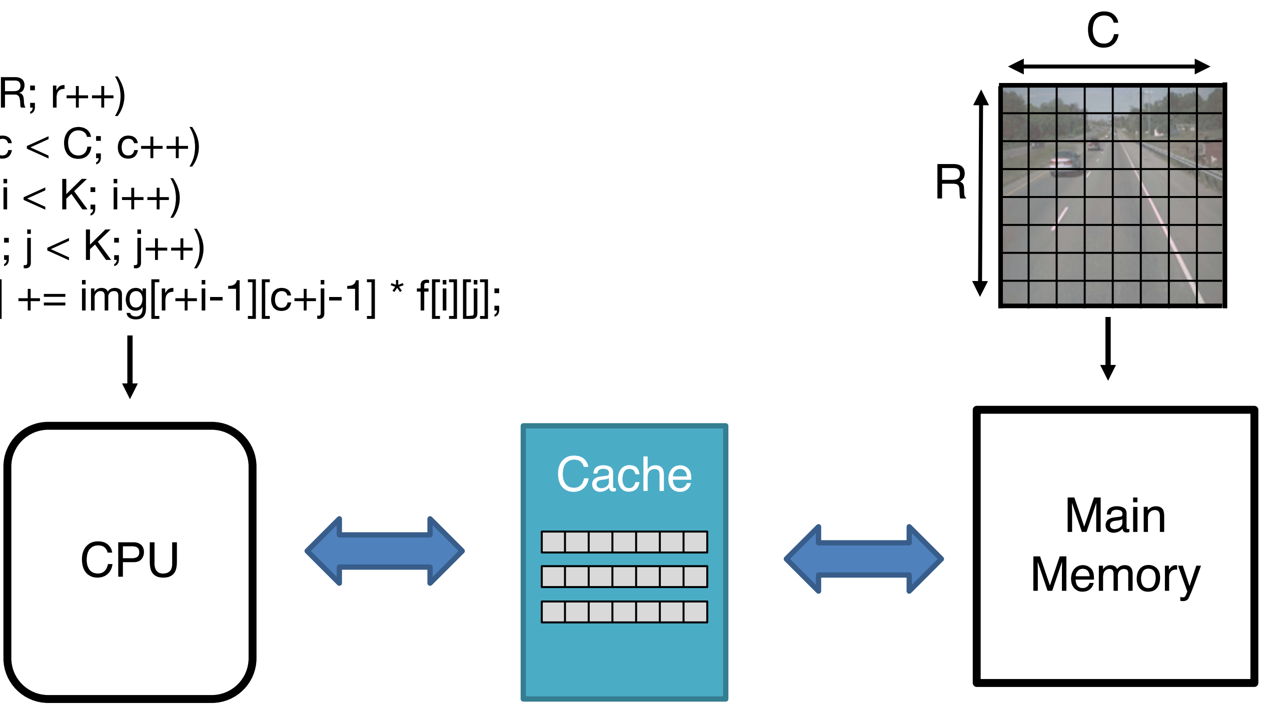
$$G_Y = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$



Figures: Pilho Kim, GaTech

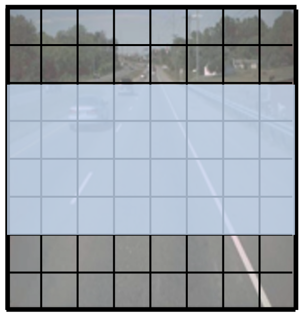
CPU Implementation of a 3x3 Convolution

```
for (r = 1; r < R; r++)
  for (c = 1; c < C; c++)
   for (i = 0; i < K; i++)
    for (j = 0; j < K; j++)
     out[r][c] += img[r+i-1][c+j-1] * f[i][j];
```

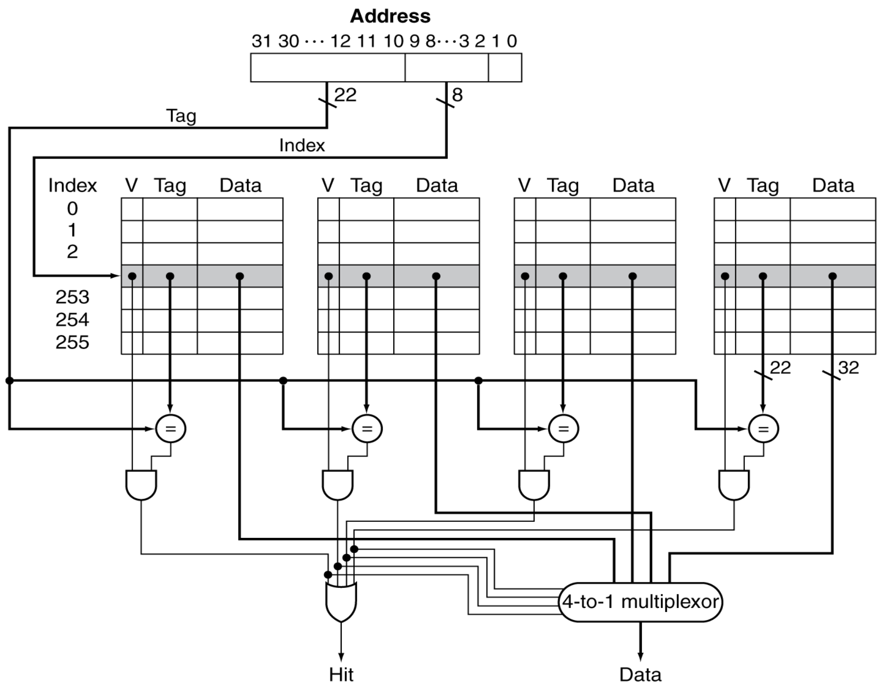


General-Purpose Cache for Convolution

- A general-purpose cache can significantly reduce external memory accesses, but it is costly and incurs high energy overhead

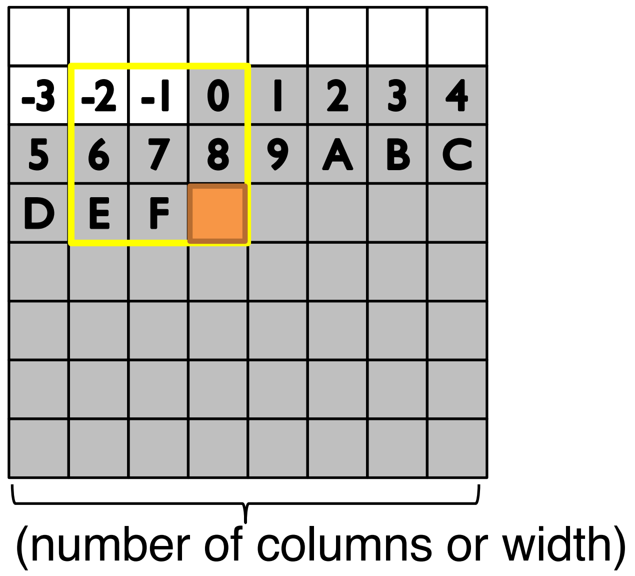


A subset of image data stored in cached

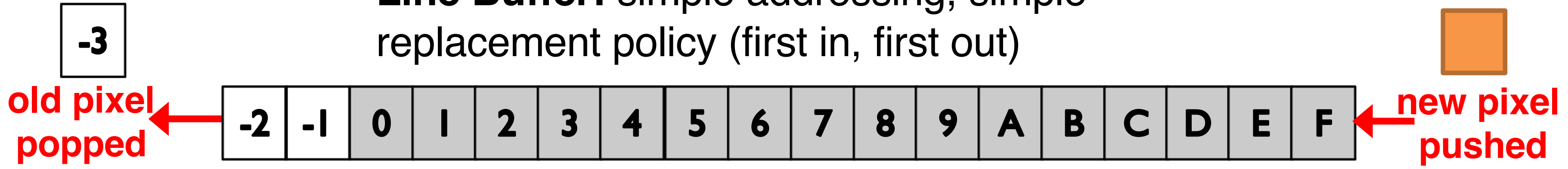


Line Buffer: Customized "Cache" for Convolution

- **"Cache" the input pixels in a line buffer**: Each time we move the KxK window (in yellow) to the right and push in a new pixel (in orange) to the specialized "cache"

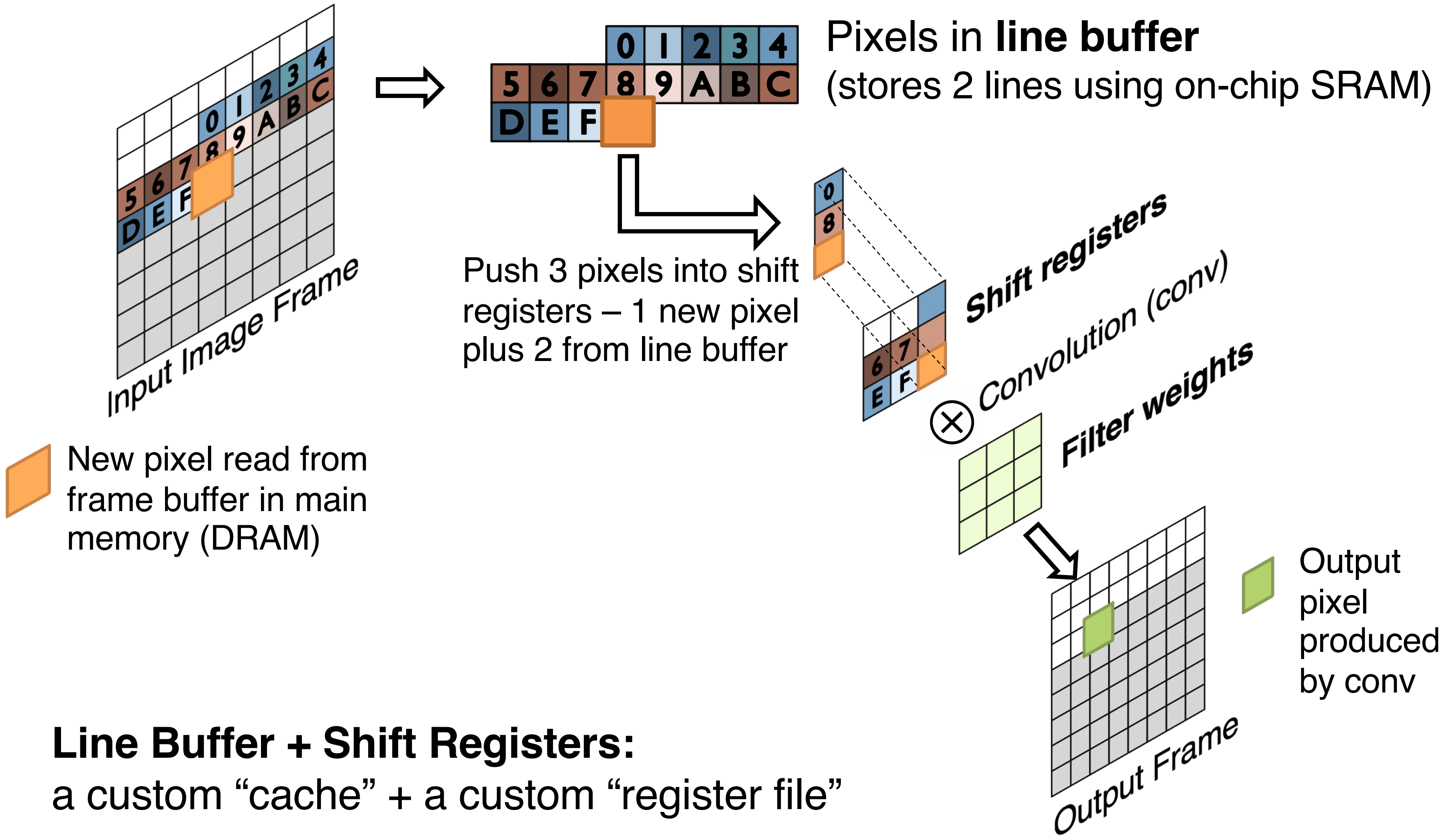


Line Buffer: simple addressing, simple replacement policy (first in, first out)



(K-1)*C+K pixels in flight, K=3 and C=8 here

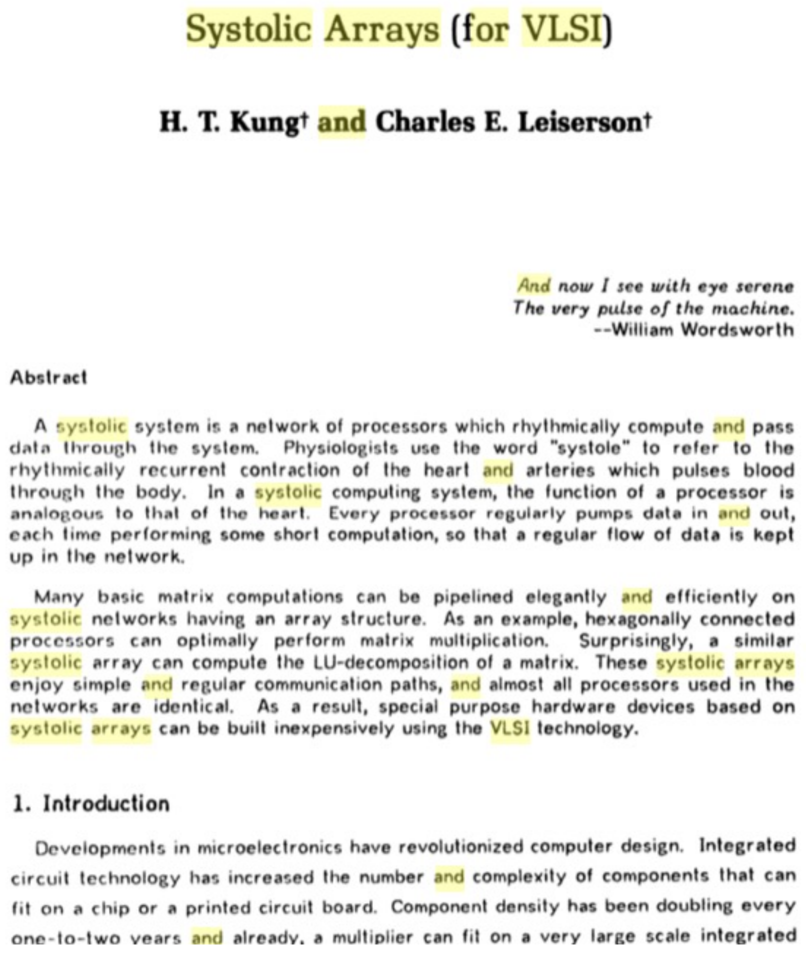
A More Complete Picture of the Customized On-Chip Memory Hierarchy


0 1 2 3 4
5 6 7 8 9 A B C
D E F
Input Image Frame
Pixels in **line buffer**
(stores 2 lines using on-chip SRAM)
Push 3 pixels into shift registers – 1 new pixel plus 2 from line buffer
New pixel read from frame buffer in main memory (DRAM)
Shift registers
Convolution (conv)
Filter weights
Output Frame
Output pixel produced by conv

Line Buffer + Shift Registers:
a custom “cache” + a custom “register file”

Custom Communication Architecture: Systolic Arrays as an Example

- An array of processing elements (PEs) that process data in a systolic manner using nearest-neighbor communication



Systolic Arrays (for VLSI)

H. T. Kung† and Charles E. Leiserson†

And now I see with eye serene
The very pulse of the machine.
--William Wordsworth

Abstract

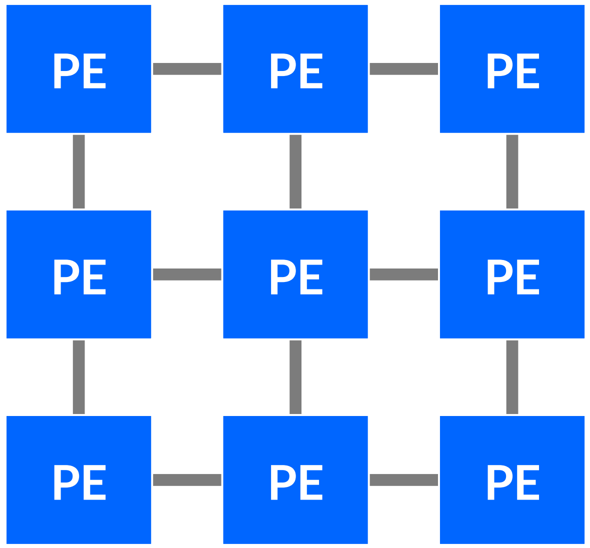
A systolic system is a network of processors which rhythmically compute and pass data through the system. Physiologists use the word "systole" to refer to the rhythmically recurrent contraction of the heart and arteries which pulses blood through the body. In a systolic computing system, the function of a processor is analogous to that of the heart. Every processor regularly pumps data in and out, each time performing some short computation, so that a regular flow of data is kept up in the network.

Many basic matrix computations can be pipelined elegantly and efficiently on systolic networks having an array structure. As an example, hexagonally connected processors can optimally perform matrix multiplication. Surprisingly, a similar systolic array can compute the LU-decomposition of a matrix. These systolic arrays enjoy simple and regular communication paths, and almost all processors used in the networks are identical. As a result, special purpose hardware devices based on systolic arrays can be built inexpensively using the VLSI technology.

1. Introduction

Developments in microelectronics have revolutionized computer design. Integrated circuit technology has increased the number and complexity of components that can fit on a chip or a printed circuit board. Component density has been doubling every one-to-two years and already, a multiplier can fit on a very large scale integrated

In Sparse Matrix Proceedings, 1978

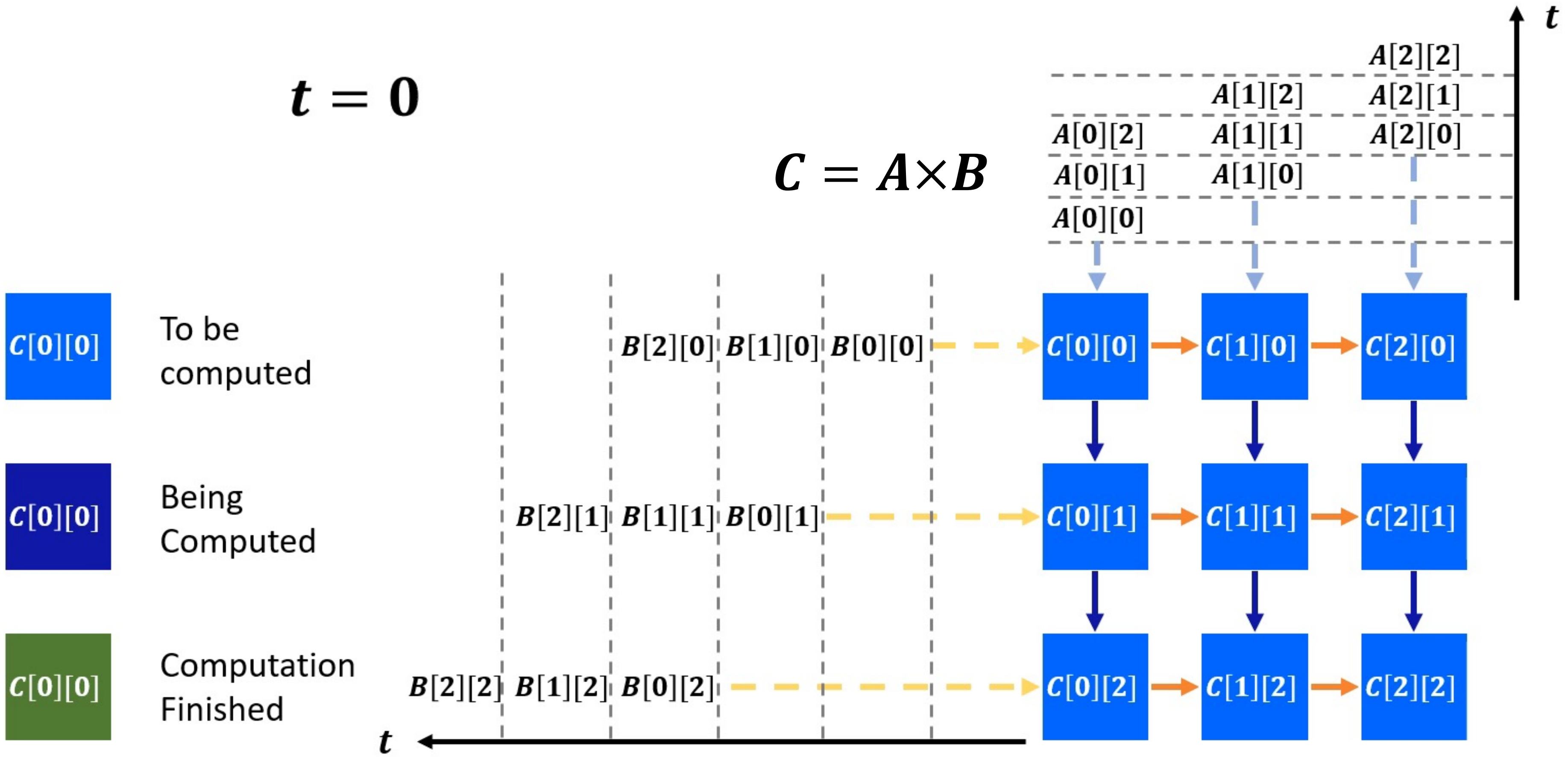


+ Simple & regular design
+ Massive parallelism
+ Short nearest-neighbor interconnection
+ Balancing compute with I/O

Matrix Multiplication (MatMul) on a Systolic Array

$t = 0$

$C = A \times B$



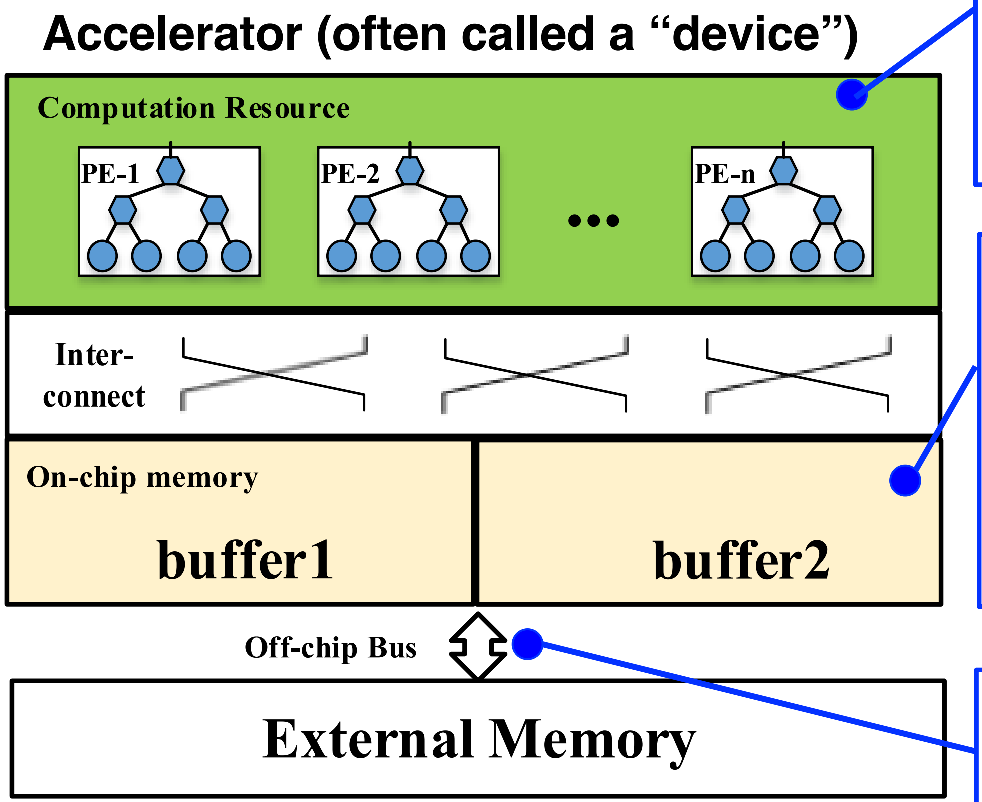
t
$A[2][2]$
$A[1][2]$
$A[2][1]$
$A[0][2]$
$A[1][1]$
$A[2][0]$
$A[0][1]$
$A[1][0]$
$A[0][0]$
$C[0][0]$ To be computed
$C[0][0]$ Being Computed
$C[0][0]$ Computation Finished
$B[2][0]$ $B[1][0]$ $B[0][0]$
$B[2][1]$ $B[1][1]$ $B[0][1]$
$B[2][2]$ $B[1][2]$ $B[0][2]$
$C[0][0]$ $C[1][0]$ $C[2][0]$
$C[0][1]$ $C[1][1]$ $C[2][1]$
$C[0][2]$ $C[1][2]$ $C[2][2]$
t

Accelerator Performance Modeling

- How do we model the performance of an application running on a compute device (e.g., an accelerator)?
  - What's the maximum (peak) throughput of the device?
  - What's the actual attainable throughput?
  - Where're the bottlenecks?

- It requires characterization of both application and hardware

Key Metrics to Consider



Computational Throughput

$$= \frac{\text{Total number of operations}}{\text{Total execution time}}$$

(OPs / Sec)

Operational Intensity* **(OI)**

$$= \frac{\text{Total number of operations}}{\text{Total external memory accesses}}$$

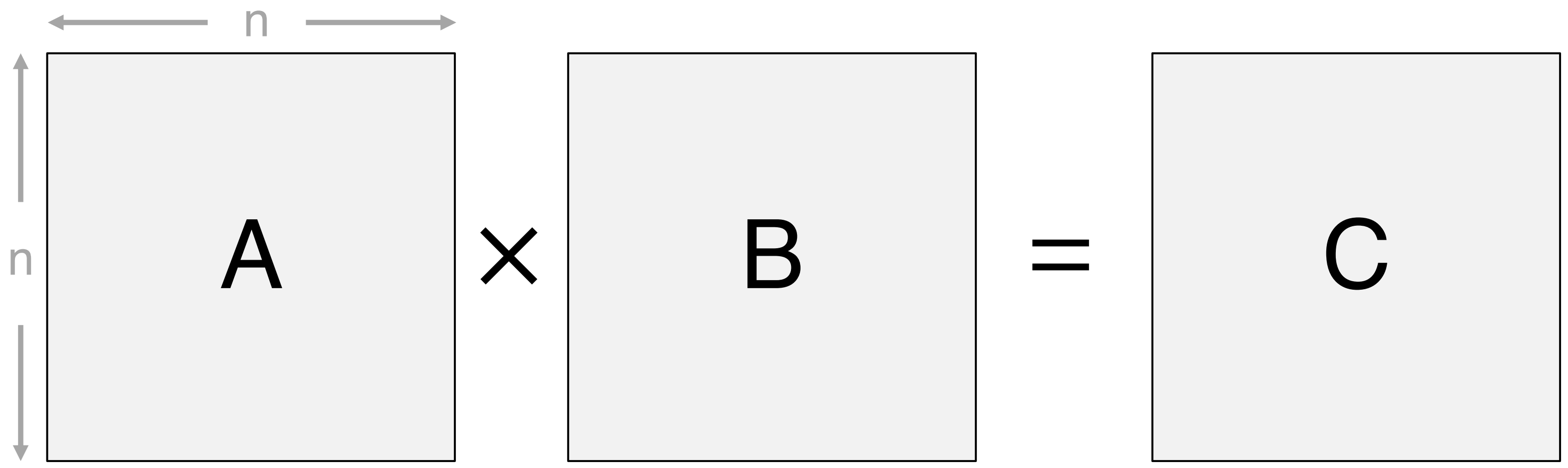
(OPs / Byte Accessed)

Memory Bandwidth

(Bytes / Sec)

* OI is also known as computation to communication ratio (CTC) or arithmetic intensity (AI)

Estimating OI: Matrix Matrix Multiplication



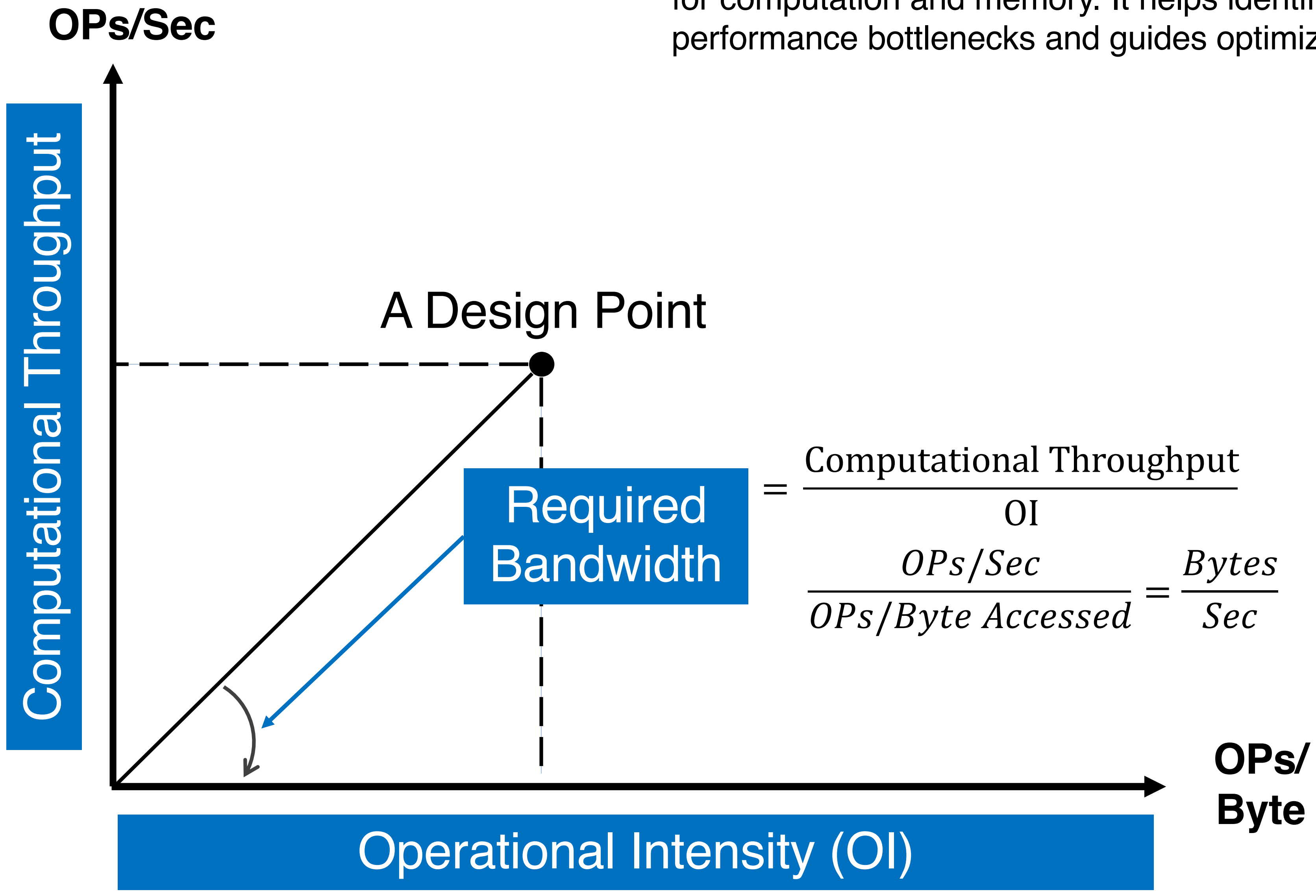
A, B, C are in single-precision float (32-bit per element)

Number of operations: $2n^3$ (multiply & add counted as two ops)
External mem accesses: $3n^2$ x 4 bytes read or written
(with perfect data reuse)

$$\text{OI} = \frac{\text{\# of operations}}{\text{\# of bytes read/written}} = \frac{2n^3}{12n^2} = \frac{n}{6}$$

Roofline Model [1]

The Roofline Model visualizes a system's performance limits, showing how an application's throughput compares to the theoretical maximums for computation and memory. It helps identify performance bottlenecks and guides optimization.

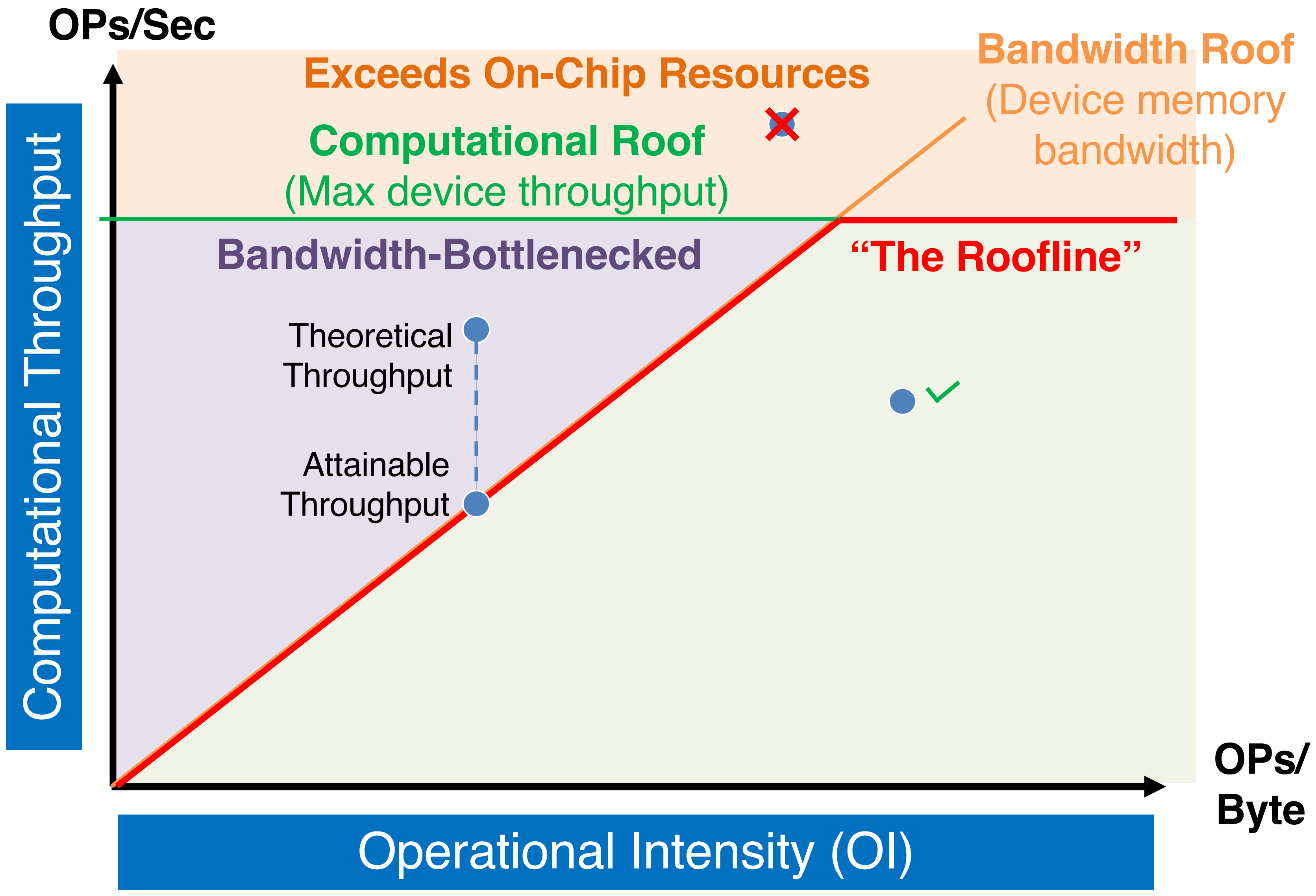


$$= \frac{\text{Computational Throughput}}{\text{OI}}$$

$$\frac{OPs/Sec}{OPs/Byte\ Accessed} = \frac{Bytes}{Sec}$$

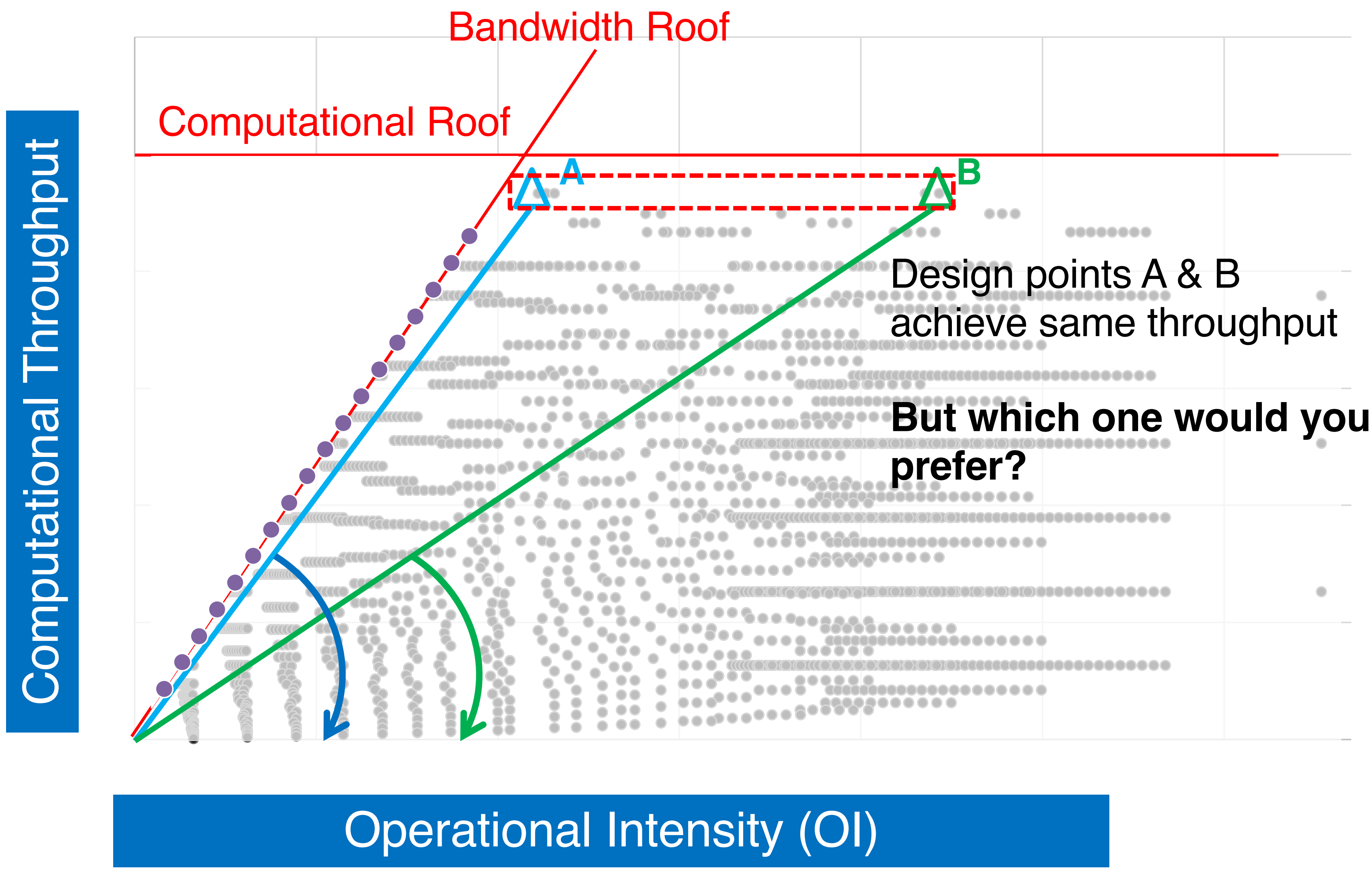
[1] S. Williams, A. Waterman, and D. Patterson, Roofline: an insightful visual performance model for multicore architectures, CACM, 2009.

Roofline Model [1]



[1] S. Williams, A. Waterman, and D. Patterson, Roofline: an insightful visual performance model for multicore architectures, CACM, 2009.

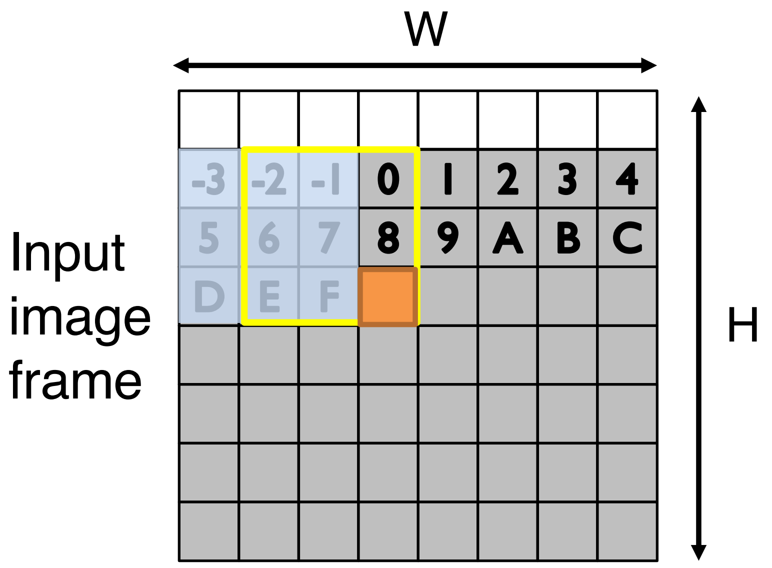
Design Space Exploration with Roofline



Design points A & B achieve same throughput

But which one would you prefer?

Exercise: OI Analysis of 2D Convolution



Estimate the OI for the 2D convolution kernel both without and with the use of a line buffer

Summary

- End of Dennard scaling leads to increasing **hardware specialization** to sustain improvement in hardware performance and energy efficiency

- Special-purpose hardware **accelerators** commonly leverage customized (1) compute units, (2) numeric types, (3) memory hierarchy, and (4) communication architecture

- **Roofline modeling** is a useful tool for first-order analysis of the accelerator performance

Next Lecture

- Field-programmable gate arrays (FPGAs)
  - Read this paper before class

Acknowledgements

- These slides contain/adapt materials developed by
  - Prof. Jason Cong (UCLA)