Lecture 5:

Image Processing Hardware

Visual Computing Systems
Stanford CS348K, Fall 2018

Image processing workload characteristics

- "Pointwise" operations
  - output_pixel = f(input_pixel)
- "Stencil" computations (e.g., convolution, demosaic, etc.)
  - Output pixel (x,y) depends on fixed-size local region of input around (x,y)
- Lookup tables
  - e.g., contrast s-curve
- Multi-resolution operations (upsampling/downsampling)
  - e.g., Building Gaussian/Laplacian pyramids
- Fast-fourier transform
  - We didn't talk about many Fourier-domain techniques in class (but readings had many examples)
- Long pipelines (DAGs) of these operations

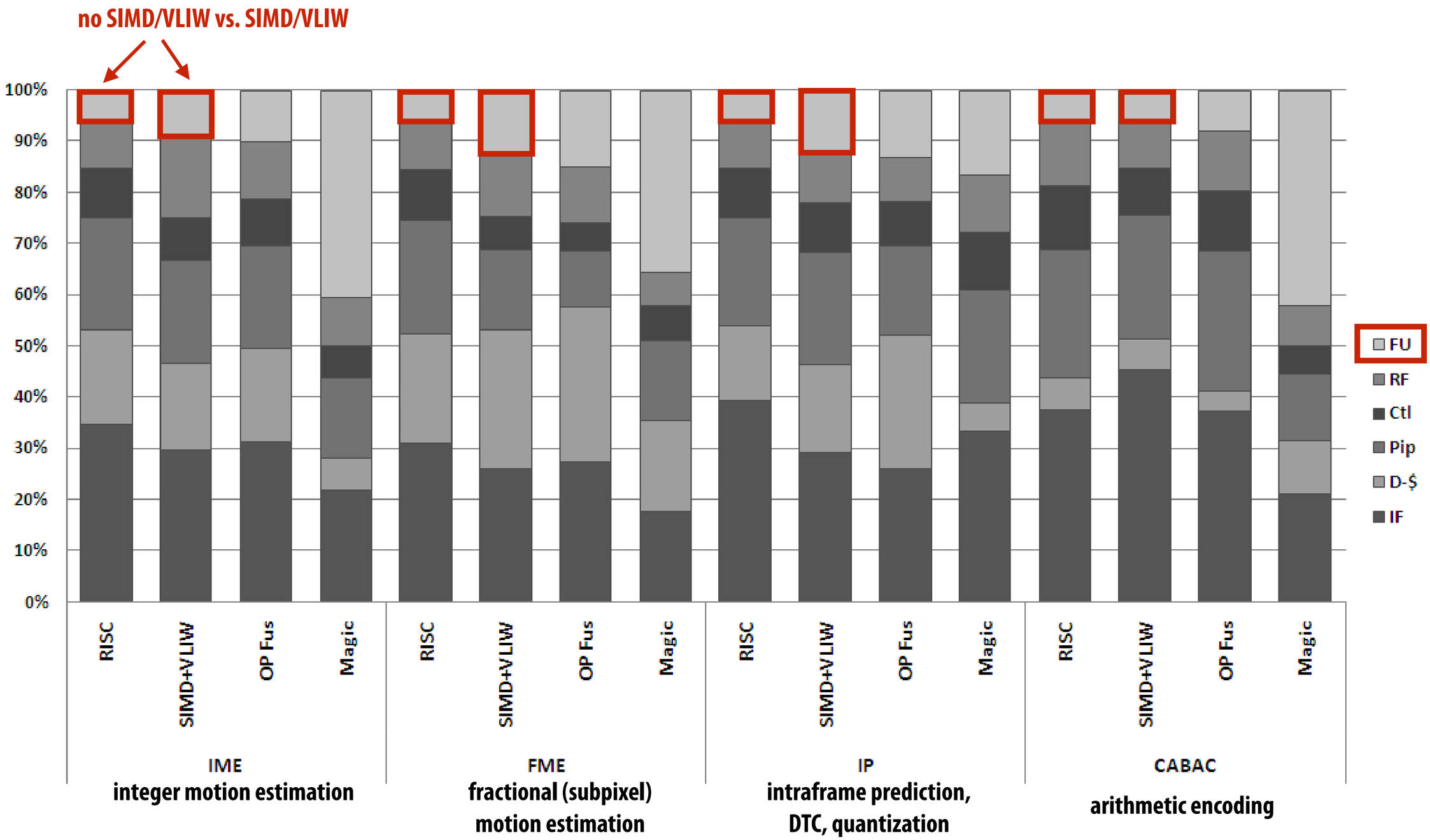
So far, the discussion in this class has focused on generating efficient code for multi-core processors such as CPUs and GPUs.

Consider the complexity of executing an instruction on a modern processor…

Read instruction — Address translation, communicate with icache, access icache, etc.
Decode instruction — Translate op to uops, access uop cache, etc.
Check for dependencies/pipeline hazards
Identify available execution resource
Use decoded operands to control register file (retrieve data)
Move data from register file to selected execution resource
Perform arithmetic operation
Move data from execution resource to register file
Use decoded operands to control write to register file SRAM

Question:
How does SIMD execution reduce overhead when executing certain types of computations?
What properties must these computations have?

Fraction of energy consumed by different parts of instruction pipeline (H.264 video encoding) [Hameed et al. ISCA 2010]



FU = functional units
RF = register fetch
Ctrl = misc pipeline control
Pip = pipeline registers (interstage)
D-$ = data cache
IF = instruction fetch + instruction cache

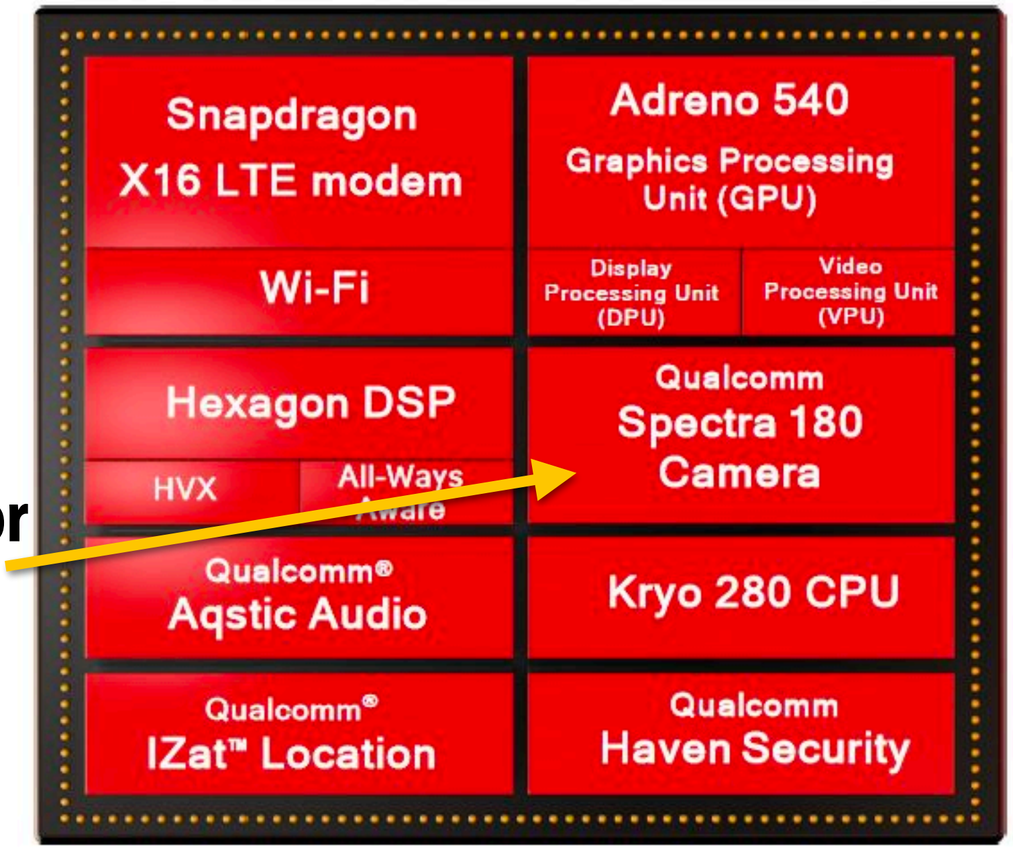
Modern SoC's feature ASIC for image processing

- Implement basic RAW to RGB camera pipeline in silicon
  - Traditionally has been critical for real-time processing like viewfinder or video

Image Signal Processor

ASIC for processing camera sensor pixels

Qualcomm Snapdragon SoC



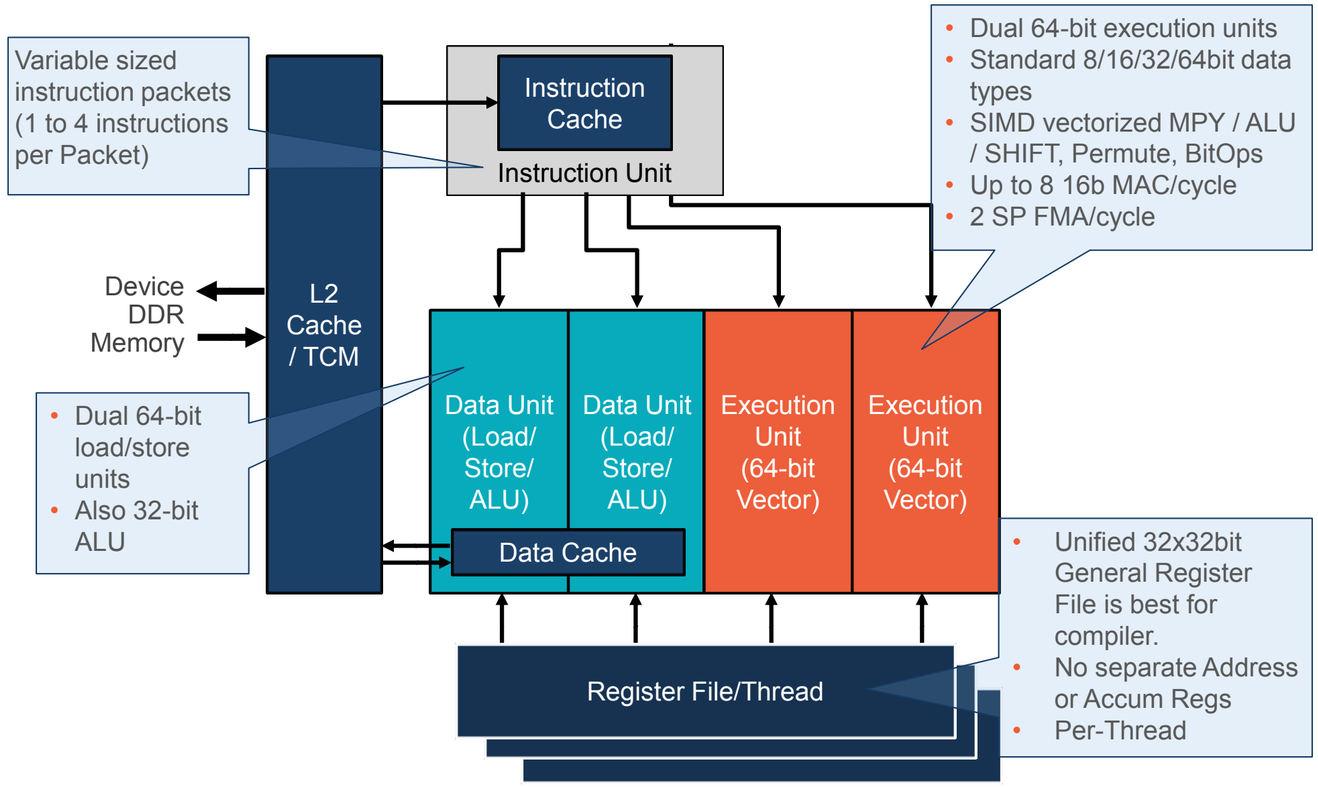
Digital Signal Processor (DSP)

- Typically simpler instruction stream control paths
- Complex instructions (e.g., SIMD/VLIW): perform many operations per instruction

Example: Qualcomm Hexagon

Used for modem, audio, and (increasingly) image processing on Qualcomm Snapdragon SoC processors

Below: innermost loop of FFT
29 "RISC" ops per cycle



64-bit Load and

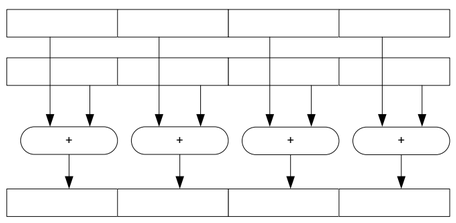
64-bit Store with post-update addressing

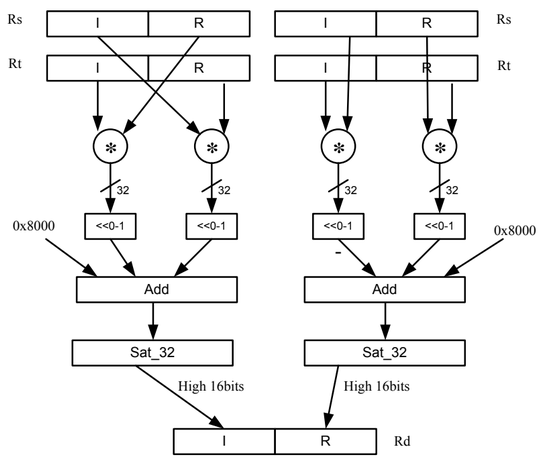
```
{ R17:16 = MEMD(R0++M1)
  MEMD(R6++M1) = R25:24
  R20 = CMPY(R20, R8):<<1:rnd:sat
  R11:10 = VADDH(R11:10, R13:12)
}:endloop0
```

Complex multiply with round and saturation

Zero-overhead loops
- Dec count
- Compare
- Jump top

Vector 4x16-bit Add



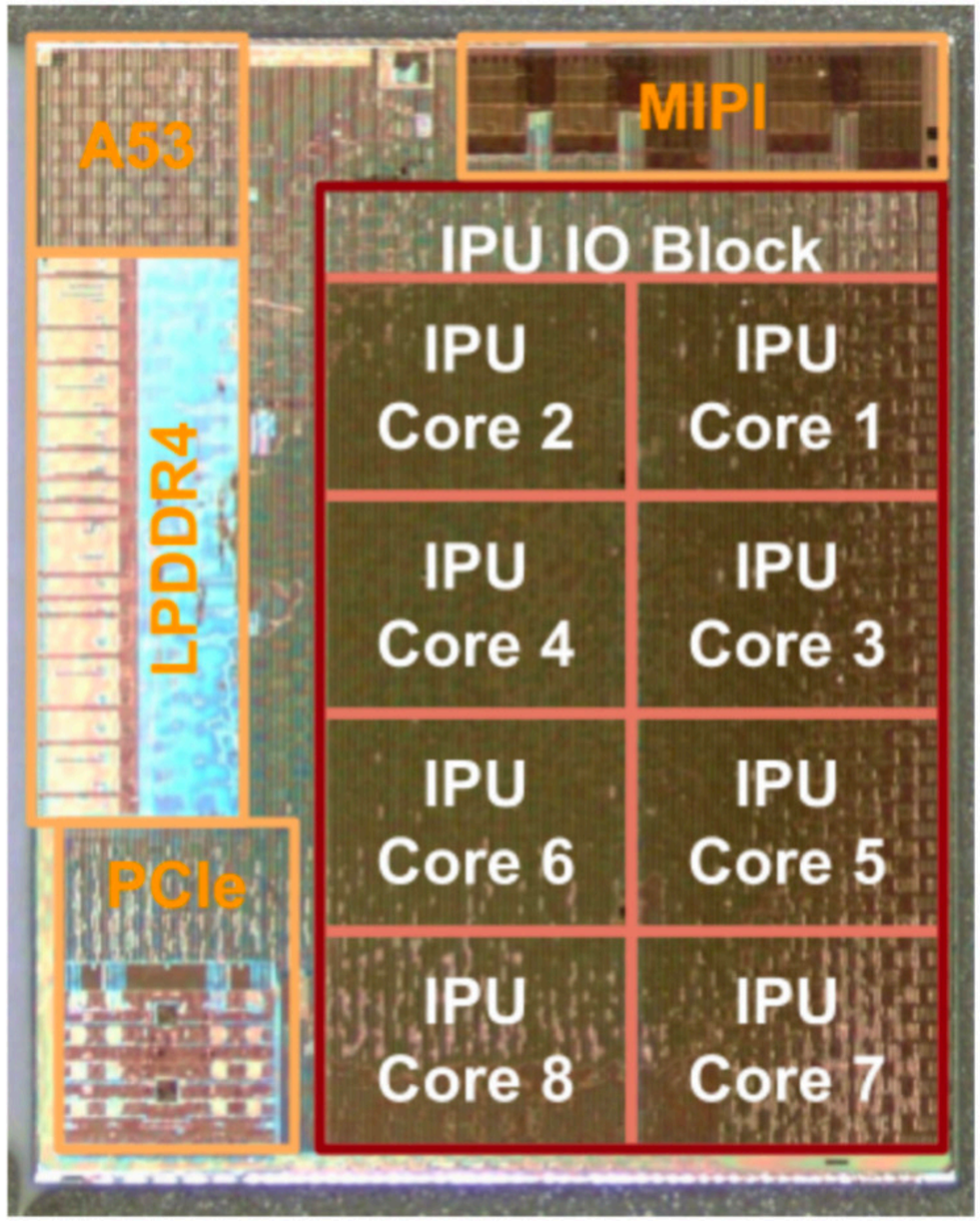


Google's Pixel Visual Core

- Programmable Image Processing Unit (IPU) in Google Pixel 2 phone
  - Augments capabilities of Qualcomm Snapdragon SoC

- Designed for energy-efficient image processing
  - Each core = 16x16 grid of 16 bit mul-add ALUs
  - Goal: 10-20x more efficient than CPU/GPU on SoC

- Programmed using Halide and Tensorflow



Class discussion:

Google Pixel Visual Core

Question

- What is the role of an ISA? (e.g., x86)

  Answer: interface between program definition (software) and hardware implementation

  Compilers produce sequence of instructions

  Hardware executes sequences of instructions as efficiently as possible

  (As shown earlier in lecture, many circuits used to implement/preserve this abstraction, not execute the computation needed by the program)

New ways of defining hardware

- Verilog/VHDL present very low level programming abstractions for modeling circuits (RTL abstraction: register transfer level)
  - Combinatorial logic
  - Registers

- Due to need for greater efficiency, there is significant modern interest in making it easier to synthesize circuit-level designs
  - Skip the ISA, directly synthesize circuits needed to compute the tasks defined by a program.
  - Raise the level of abstraction for direct hardware programming

- Examples:
  - C to HDL (e.g., ROCCC, Vivado)
  - Bluespec
  - CoRAM [Chung 11]
  - Chisel [Bachrach 2012]

Compiling image processing pipelines directly to HW

- Darkroom [Hegarty 2014]
- Rigel [Hegarty 2016]
- RIPL [Stewart 2018]

- Motivation:
  - Convenience of high-level description of image processing algorithms (like Halide)
  - Energy-efficiency of hardware implementations (particularly important for high-frame rate, low-latency, always on, embedded/robotics applications)

Optimizing for minimal buffering

- Recall: scheduling Halide programs for CPUs/GPUs
  - Key challenge: organize computation so intermediate buffers fit in caches
- Scheduling for hardware:
  - Key challenge: minimize size of intermediate buffers (keep buffered data spatially close to combinatorial logic)

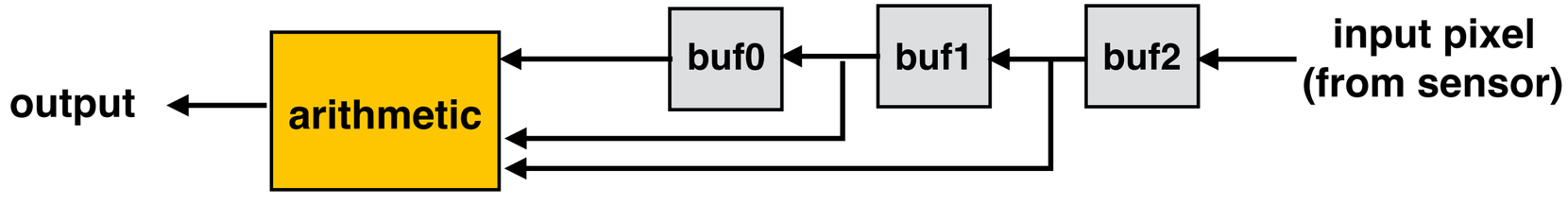
Consider 1D convolution:

```
out(x) =  (in(x-1) + in(x) + in(x+1)) / 3.0
```

Efficient hardware implementation: requires storage for 3 pixels in registers

```
out_pixel = (buf0 + buf1 + buf2) / 3
buf0 = buf1
buf1 = buf2
buf2 = in_pixel
```

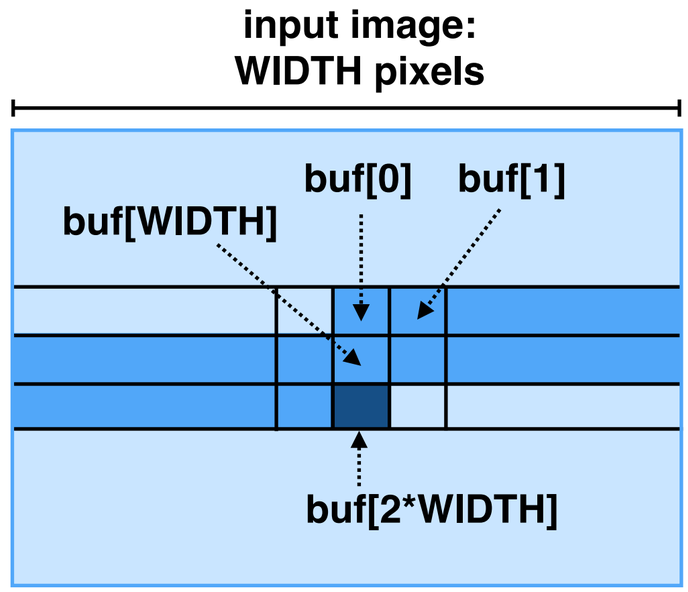
"Shift" new pixel in



"Line buffering"

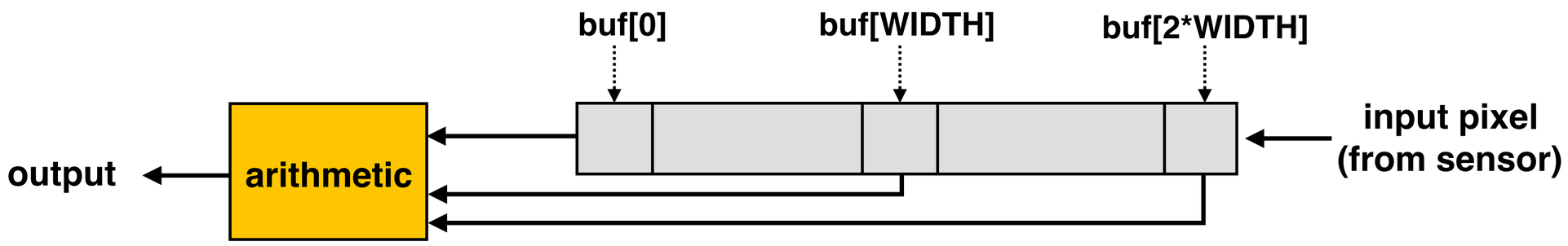
Consider convolution of 2D image in vertical direction:

```
out(x,y) = (in(x,y-1) + in(x,y) + in(x,y+1)) / 3.0
```



Efficient hardware implementation:

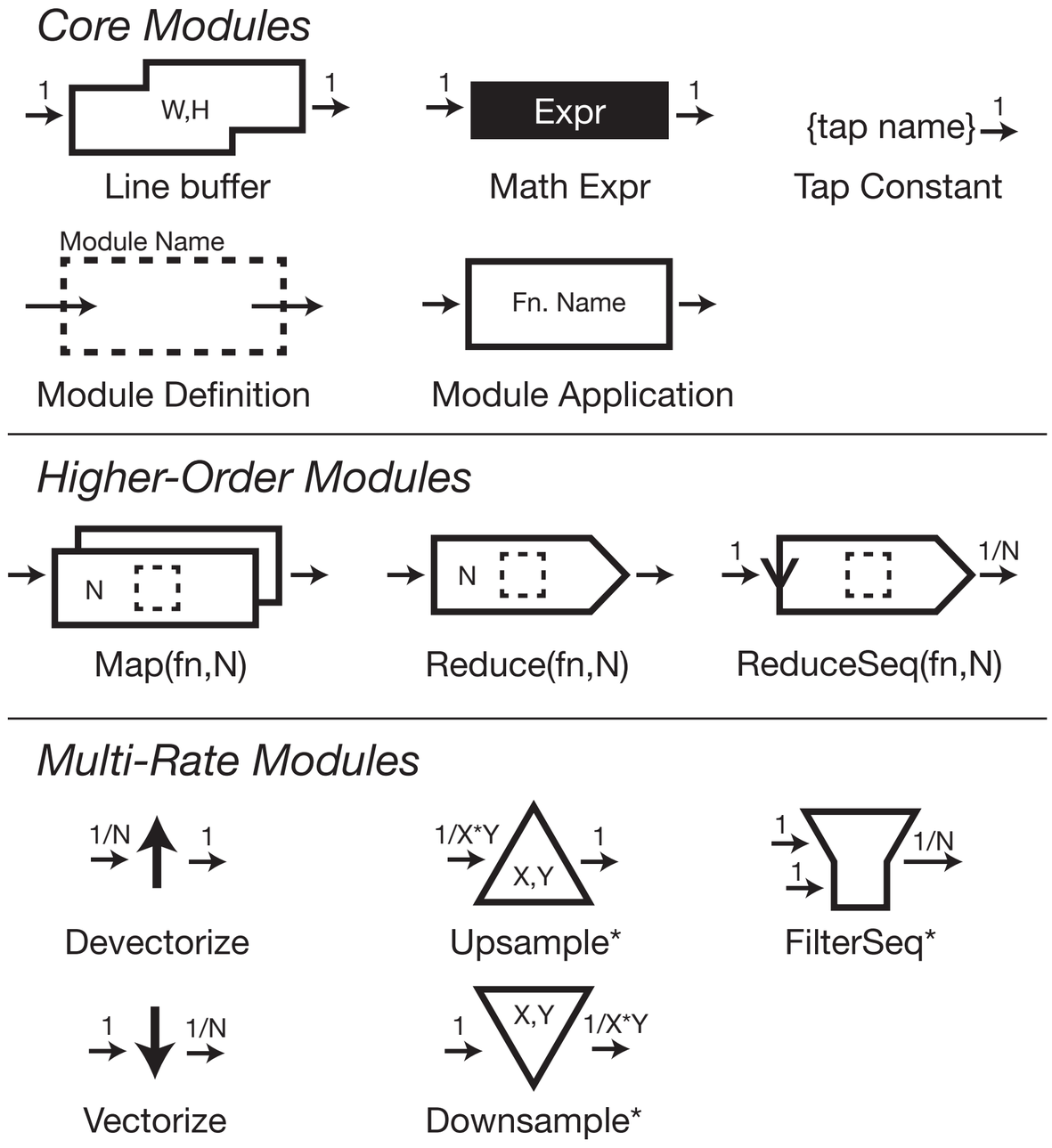
```
let buf be a shift register containing 2*WIDTH+1 pixels

// assume: no output until shift register fills
out_pixel = (buf[0] + buf[WIDTH] + buf[2*WIDTH]) / 3.0
shift(buf);  // buf[i] = buf[i+1]
buf[2*WIDTH] = in_pixel
```



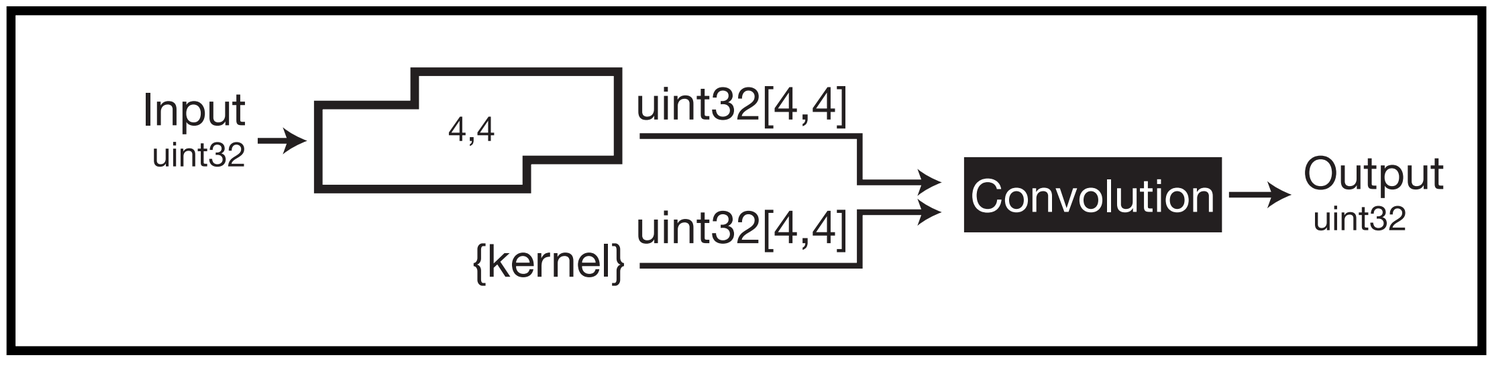
Rigel

[Hegarty et al. 2016]

- **Provides set of well-defined building hardware blocks which can be assembled into image processing data flow graphs**
- **Provides programmer service of gluing modules in a dataflow graph together to form a complete implementation**



Example: 4x4 convolution in Rigel



Halide to hardware

[Pu et al. 2017]

- Reinterpret common Halide scheduling primitives to describe features of hardware circuits
  - unroll() —> replicate hardware
- Add new primitive accelerate()
  - Defines granularity of accelerated task
  - Defines throughput of accelerated task

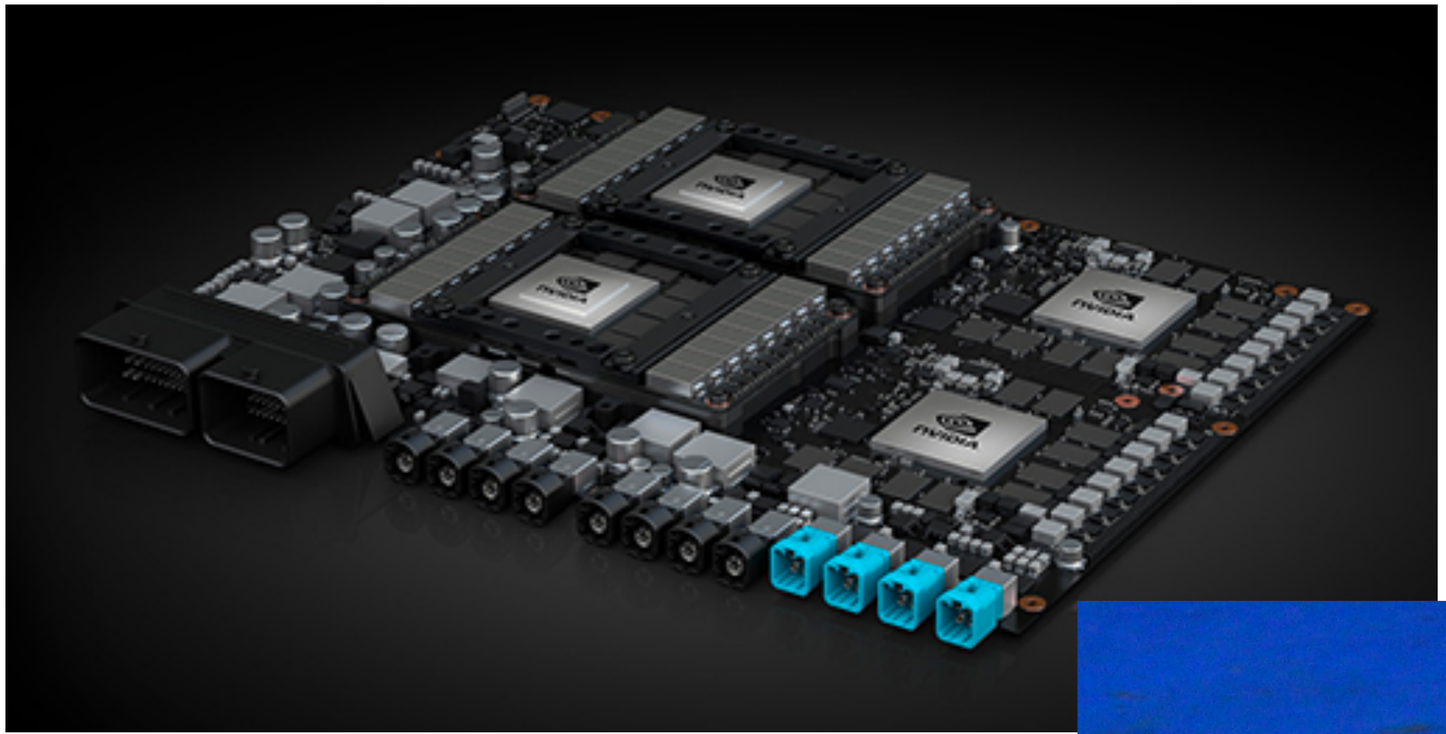
```
Func unsharp(Func in) {
  Func gray, blurx, blury, sharpen, ratio, unsharp;
  Var x, y, c, xi, yi;

  // The algorithm
  gray(x, y) = 0.3*in(0, x, y) + 0.6*in(1, x, y) + 0.1*in(2, x, y);
  blury(x, y) = (gray(x, y-1) + gray(x, y) + gray(x, y+1)) / 3;
  blurx(x, y) = (blury(x-1, y) + blury(x, y) + blury(x+1, y)) / 3;
  sharpen(x, y) = 2 * gray(x, y) - blurx(x, y);
  ratio(x, y) = sharpen(x, y) / gray(x, y);
  unsharp(c, x, y) = ratio(x, y) * input(c, x, y);

  // The schedule
  unsharp.tile(x, y, xi, yi, 256, 256).unroll(c)
         .accelerate({in}, xi, x)
         .parallel(y).parallel(x);
  in.fifo_depth(unsharp, 512);
  gray.linebuffer().fifo_depth(ratio, 8);
  blury.linebuffer();
  ratio.linebuffer();

  return unsharp;
}
```

Parallel units for rgb

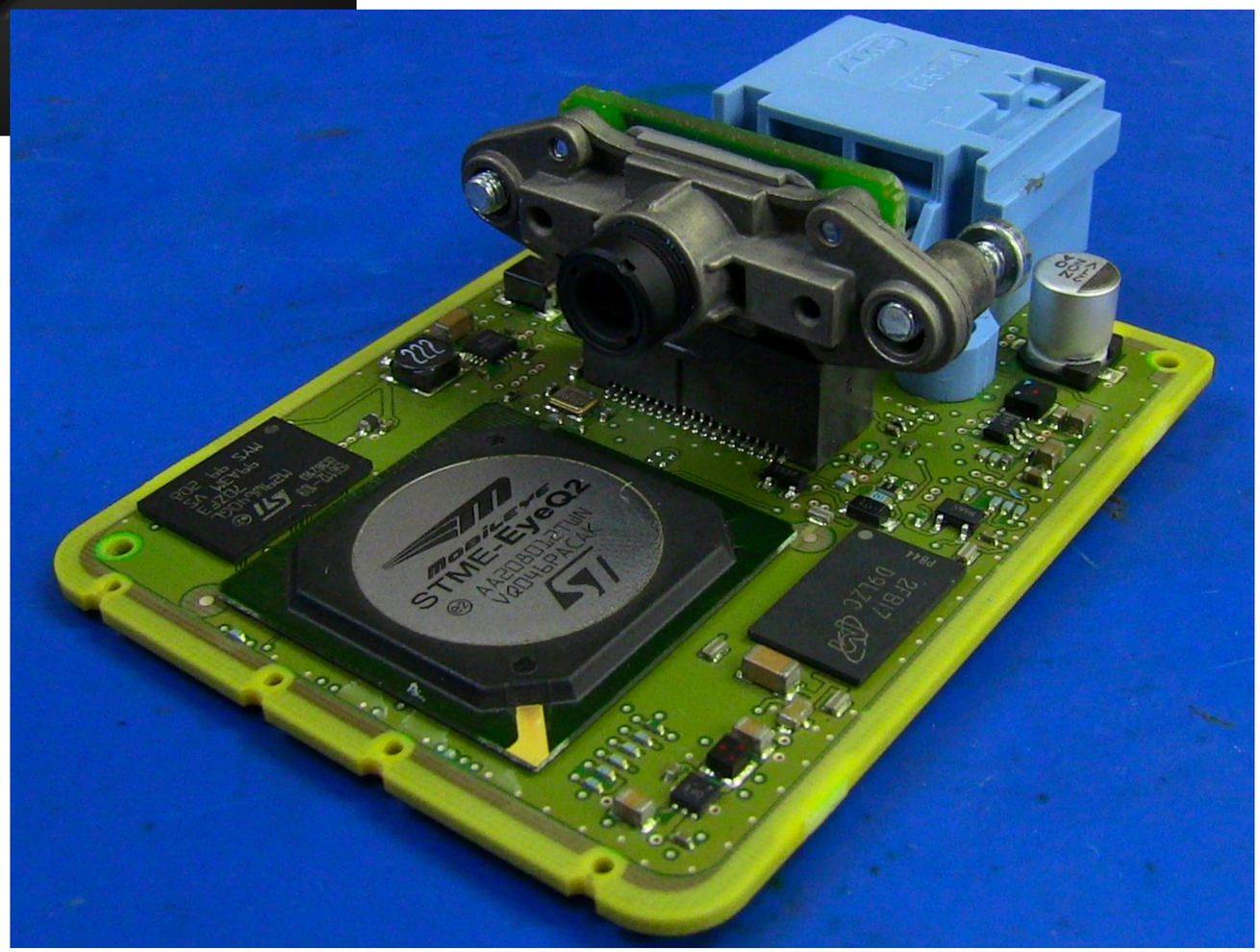
Unit of work done per cycle: one pixel (one iteration of xi loop)

Unit of work given to accelerator: 256x256 tile (one iteration of x loop)

Image processing for automotive/robotics



NVIDIA Drive Xavier



MobileEye EyeQ Processor
Computer Vision accelerator for automotive applications

https://en.wikipedia.org/wiki/Mobileye

Summary

- Image processing workloads: demand high performance
  - Historically: accelerated via ASICs for efficiency on mobile devices
  - Rapidly evolving algorithms dictate need for programmability
  - Workload characteristics are amenable to hardware specialization

- Active industry efforts: programmable image processors
  - Gain efficiency via wide parallelism and by limiting data flows

- Active academic research topic: increasing productivity of custom hardware design
  - Image processing is an application of interest