Programming CPU/FPGA Heterogeneous Systems for Image Processing from Halide DSL

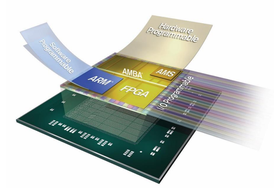
Jing Pu, Steven Bell, Xuan Yang, Stephen Richardson,
Jonathan Ragan-Kelley, Mark Horowitz

Stanford | ENGINEERING
Electrical Engineering

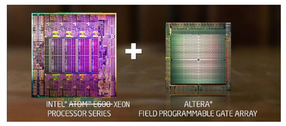
Introduction

Motivation

- Image Processing Domain
  - Computer Vision
  - Computational Photography
  - Augmented Reality
- New SoCs with FPGA are promising platforms for acceleration



Xilinx Zynq SoC



Intel + Altera

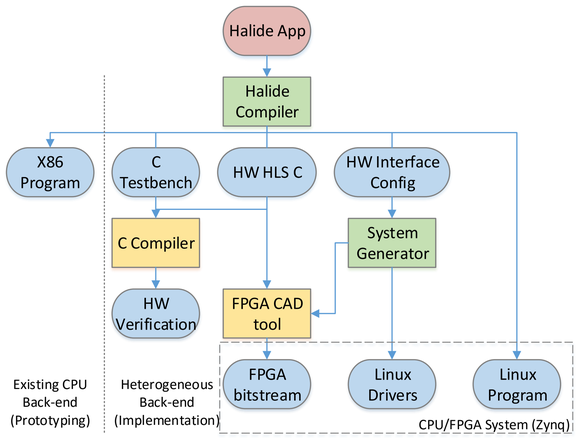
Challenges: Programmability

- Designing FPGA accelerator is HARD
- Integrating new HW to existing systems is HARD
- Writing software to access HW is HARD

Opportunities: Halide

- Halide splits *algorithm* from *schedule*
  - Algorithms define essential computation
  - Different architectures (CPU, GPU, FPGA) use different schedules for optimization

Design Flow

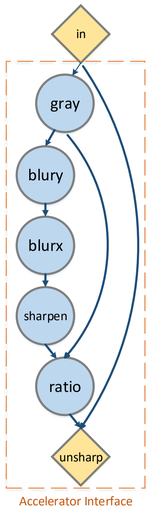


Prototyping algorithms uses X86 backend.
Accelerating uses heterogeneous backend.

High-level Language: Halide

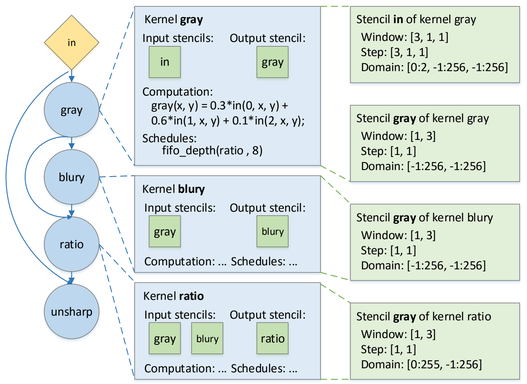
```
Func unsharp(Func in) {
  Func gray, blurx, blury, sharpen, ratio, unsharp;
  Var x, y, c, xi, yi;

  // 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;
}
```

Halide algorithm and schedule for *unsharp*

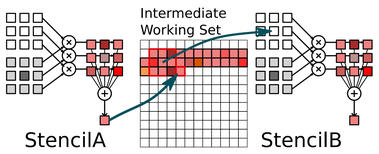


DAG of functions

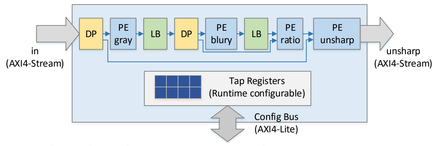
Accelerator Architecture



Pipeline Architectural parameters for *unsharp*

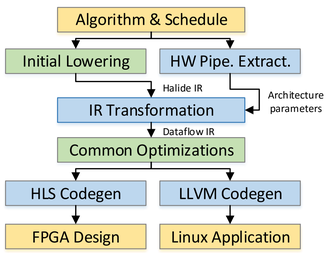


Line buffer (LB) exploits maximum data reuse in image processing pipelines

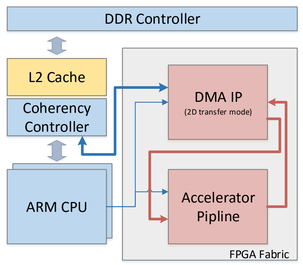


Pipeline implementation for *unsharp*

Compiler and Platform

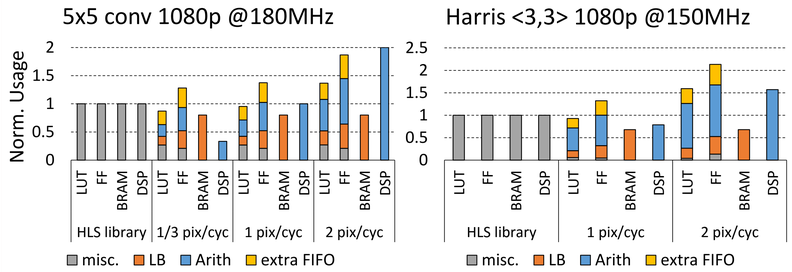


Compiler based on Halide compiler



Platform based on Xilinx Zynq SoC

Hardware Generation

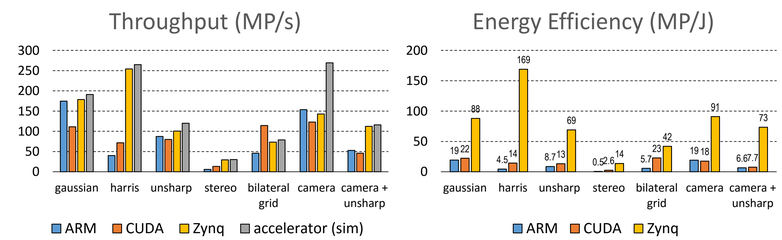


Our compiler generates hardware as good or better than hand-optimized kernels from a commercial HLS video library.

Our system generates different throughput designs controlled by `unroll()` primitive specified by users.

System Performance and Energy Efficiency

		Application						
		gaussian	harris	unsharp	stereo	bilateral grid	camera	camera + unsharp
HW accelerator	Block size (px)	480×640	480×640	480×640	600×400	480×640	640×480	640×480
	Frequency (MHz)	100	142	125	125	90.9	142	125
	Pixel rate (px/cyc)	2	2	1	0.25	1	2	1
	Bandwidth (MB/s)	200	284	375	62	182	568	250
	LUT	9%	22%	6%	55%	23%	8%	15%
	FF	5%	19%	5%	32%	20%	6%	8%
	BRAM	3%	7%	7%	5%	14%	71%	44%
	DSP	33%	30%	7%	0%	13%	5%	16%
	Throughput (MP/s)	191	264	120	30	79	280%	116%
CPU workload		tiling	tiling	tiling	tiling + padding	tiling + shuffling	tiling	tiling



Our system vs. NVIDIA Tegra K1's four ARM cores and GPU:

Baseline	Throughput	Energy Efficiency
K1 CPUs	6x (avg. 2.6x)	38x (avg. 14x)
K1 GPU	3.5x (avg. 2x)	12x (avg. 6x)

As image processing pipeline grows deeper, the FPGA acceleration becomes more effective.

Contact Information

Jing Pu, jingpu@stanford.edu