IR Design, Transformations, and Code Generation

CS448h
Oct. 8, 2015

A refresher: Regex & NFA ADTs

```
re = Char ( char )
   | Seq ( re list )
   | Or ( re list )
   | Star ( re )
   | Maybe ( re )
```

```
nfa = NFA ( node list, start : node )

node = Node ( edge list, accepts : bool, id : int )

edge = EpsEdge ( pointsTo : int )
     | CharEdge ( token : char, pointsTo : int )

nodemap = map int → node
```

Let’s design an IR!

A simple expression language

A simple expression language

2*4+3

A simple expression language

```
2*4+3
```

```
let x = 4
  in 2*x+3
```

A simple expression language

```
expr = Add ( expr, expr )
     | Sub ( expr, expr )
     | Mul ( expr, expr)
     | Div ( expr, expr )
     | Val ( float )
     | Var ( var )
     | Let ( var, expr, expr )
```

or

```
expr = BinOp ( op, expr, expr )
     | Val ( float )
     | Var ( var )
     | Let ( var, expr, expr )

op = Add | Sub | Mul | Div
```

```
var = string
```

Lowering through IRs

AST: *user code*

High-level: *user intent*

Low-level: *execution strategy*

Instruction-level: *machine operations*

lowering

Patterns in lowering

Recursive traversals

generate the next IR from current

In Lua:

method dispatch on different node types

or pattern match on a type tag

```
binop:lower()          if e.kind == ‘binop’ then…
let:lower()            elsif e.kind == ‘let’ then…
```

The visitor pattern

```
class LoweringVisitor : IRVisitor {
  Expr* visit(ValNode *n) { … }
  Expr* visit(BinopNode *n) {
    Expr *lhs = visit(n->lhs);
    Expr *rhs = visit(n->rhs);
    // ...do something with the op
    return result;
  }
  …
}
```

Let’s generate some code!

A few lessons from experience

building compiler transformations

A few lessons from experience

building compiler transformations

Use immutable IR nodes

generate new trees, instead of updating in place

A few lessons from experience
building compiler transformations

Use immutable IR nodes
generate new trees, instead of updating in place

Don’t do too much in one pass
more, simpler passes (and representation variants) are your friend!

A few lessons from experience

(or, “eating your vegetables”)

Track *(file:line)* origin for every node
in your IRs

Make pretty-printers as early as possible

Our image processing language

```
in = loadppm(…)

blurH = (in:shift(-1,0)
         + in
         + in:shift(1,0)) / 3

blurV = (blurH:shift(0,-1)
         + blurH
         + blurH:shift(0,1)) / 3
```

How should we represent these programs?

A simple image processing language

```
img = Op ( op, img, img )
    | Shift ( img, int, int )
    | Load ( buf )

op = Add | Sub | Mul | Div
```

A simple image processing language

```
img = Op ( op, img, img )
    | Shift ( img, int, int )
    | Load ( buf )
    | Const ( int )

op = Add | Sub | Mul | Div
```

How can we generate code for this?

An image processing loop IR

```
stmt = Loop ( var, base : int, extent : int, body : stmt )
     | Store ( buf, idx : expr, body : stmt )
     | Alloc ( buf, size : int, stmt )
     | Block ( stmt list )

expr = img -- from before
     | Var ( var )
```

An image processing loop IR

```
stmt = Loop ( var, base : int, extent : int, body : stmt )
     | Store ( buf, idx : expr, body : stmt )
     | Alloc ( buf, size : int, stmt )
     | Block ( stmt list )

expr = img -- from before
     | Var ( var )
```

^ *, kind*

An image processing loop IR

stmt = Loop (var, base : int, extent : int, body : stmt)
  | Store (buf, idx : expr, body : stmt)
  | Alloc (buf, size : int, stmt)
  | Block (stmt list)

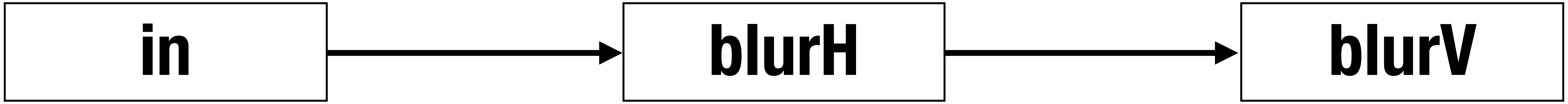
^ , *kind*

expr = img *-- from before*
  | Var (var)

kind = Serial
  | Parallel
  | Vectorized (int)

How else might we represent these programs?

How else might we represent these programs?



How else might we represent these programs?

How else might we represent these programs?

Synchronous dataflow graph
[Cf. StreamIt, Darkroom, …]

Foreshadowing…

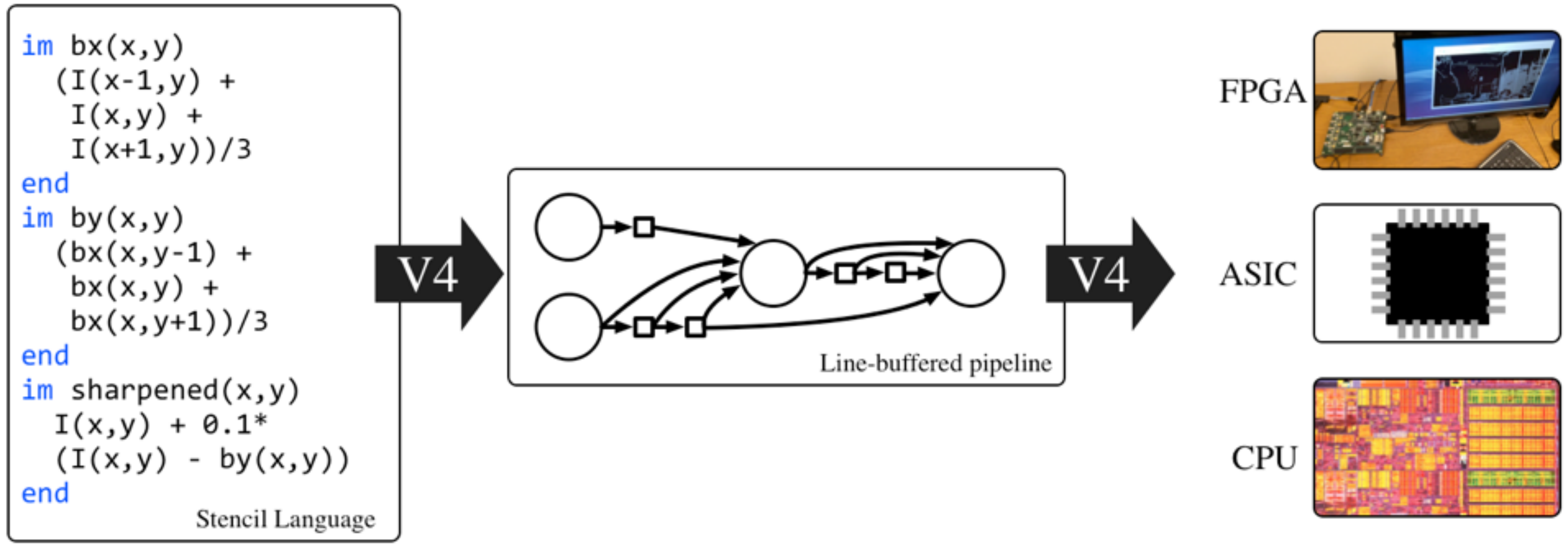
Camera Raw Pipeline	Local Laplacian Filter	Bilateral Grid	*Snake* Image Segmentation
Optimized NEON ASM: 463 lines Nokia N900: 772 ms	**C++, OpenMP+IPP:** 262 lines Quad-core x86: 335 ms	**Tuned C++:** 122 lines Quad-core x86: 472ms	**Vectorized MATLAB:** 67 lines Quad-core x86: 3800 ms
Halide algorithm: 145 lines **schedule:** 23 lines Nokia N900: 741 ms	**Halide algorithm:** 62 lines **schedule:** 7 lines Quad-core x86: 158 ms	**Halide algorithm:** 34 lines **schedule:** 6 lines Quad-core x86: 80 ms	**Halide algorithm:** 148 lines **schedule:** 7 lines Quad-core x86: 55 ms
2.75x shorter 5% *faster* than tuned assembly	3.7x shorter 2.1x faster	3x shorter 5.9x faster	2.2x longer 70x faster

Porting to new platforms does not change the algorithm code, only the schedule

Quad-core x86: 51 ms	CUDA GPU: 48 ms (7x)	CUDA GPU: 11 ms (42x) Hand-written CUDA: 23 ms [Chen et al. 2007]	CUDA GPU: 3 ms (1250x)

Halide

http://halide.io



Darkroom

http://darkroom-lang.org