TEAM 02

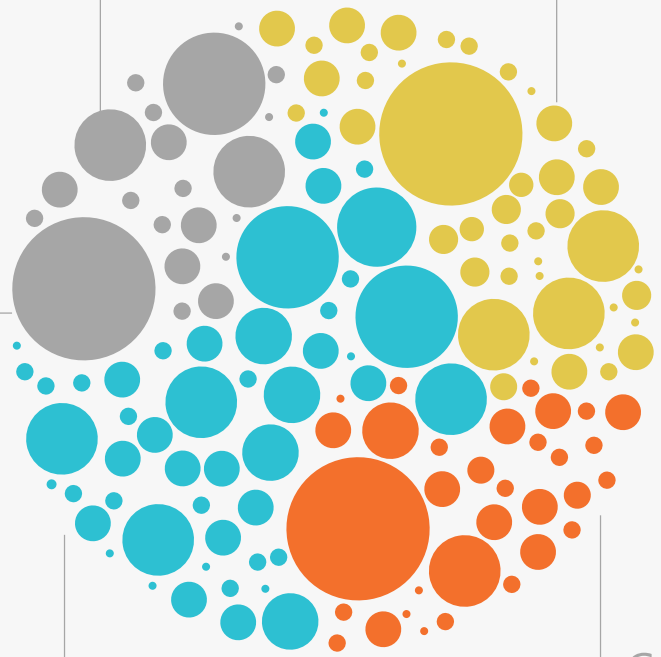
DARKROOM

Compiling High-Level Image Processing Code into Pipelines

Member:

施泽丰	PB14210224
郭振江	PB15030776
钟立	PB15000037

目录



Darkroom

Part One

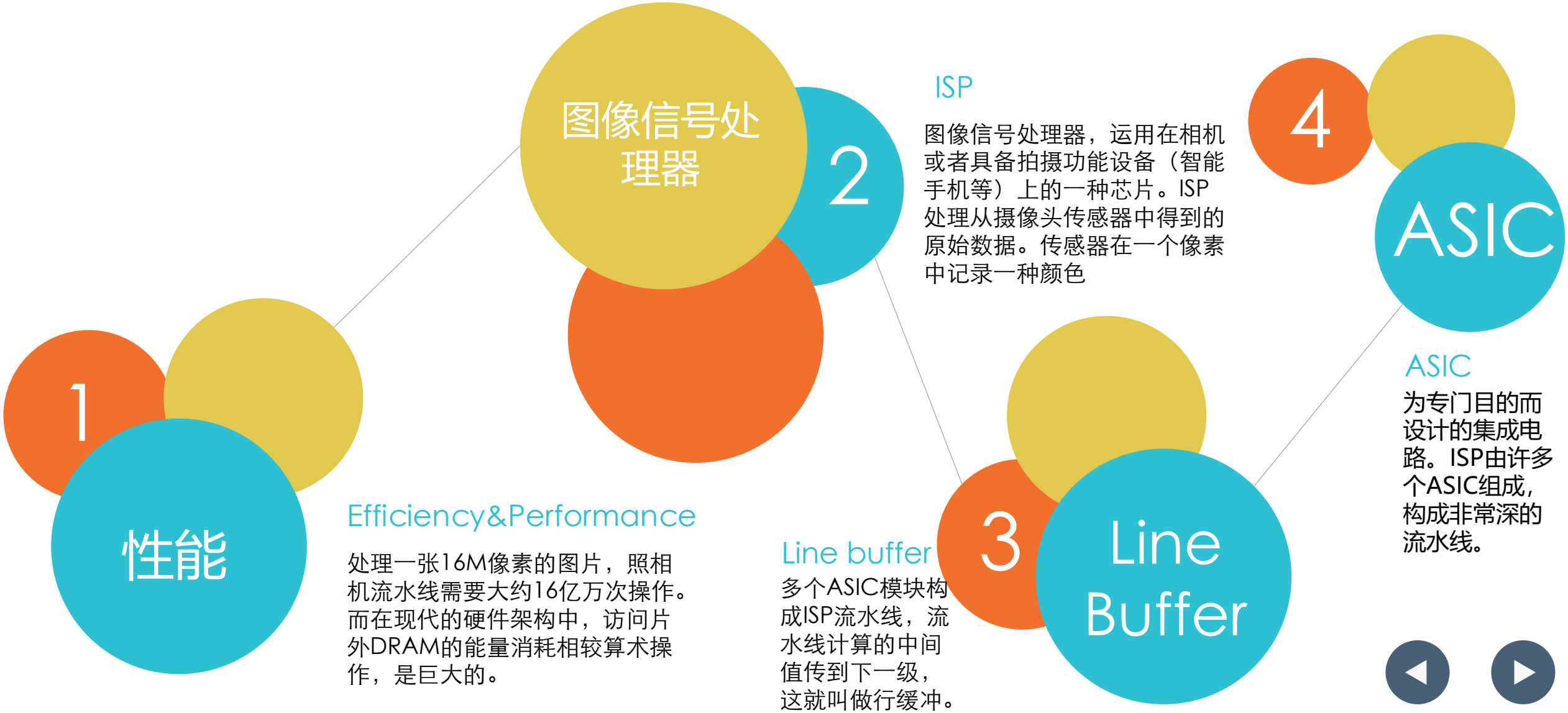
Introduction

Part One

Introduction

Part One

Introduction



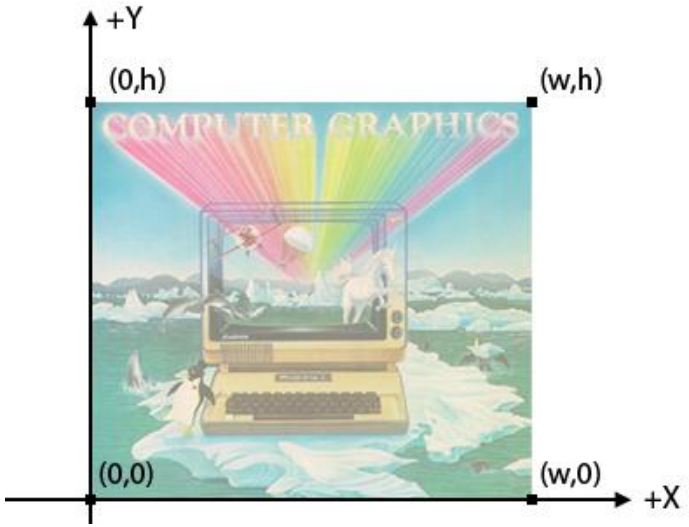
Part One

一次只在一个点上进行。
不需要缓存，因为上一级的输出就
直接输入到下一级。

要从上一级获得多个输入
比如，在计算(x,y)对应的输出时，
可能需要(x,y-1)的值
需要缓存上一级的输入

Part One

Introduction



Darkroom是一种用于描述图像处理流水线的语言，它嵌入于Terra中。

由于其语义特点，Darkroom语言能够将程序直接编译为行缓冲流水线(line-buffer pipelines)，将中间值存储在行缓冲中，减少对片外DRAM的访问。

平台：ASIC, FPGA, CPU

Darkroom

Part Two

Programming Model

Part Two

输入文件：cat.bmp

Example1.t

```
import "darkroom"
darkroomSimple = terralib.require("darkroomSimple")
inputImage = darkroomSimple.load("cat.bmp")

im decreaseBrightness(x,y)
  [uint8[3]] (inputImage(x,y) * 0.5)
end

decreaseBrightness:save("output.bmp")
```

Command Line

```
$ terra Example1.t
```

Part Two

输出文件：output.bmp

Example1.t

```
import "darkroom"
darkroomSimple = terralib.require("darkroomSimple")
inputImage = darkroomSimple.load("cat.bmp")

im decreaseBrightness(x,y)
  [uint8[3]] (inputImage(x,y) * 0.5)
end

decreaseBrightness:save("output.bmp")
```

Command Line

```
$ terra Example1.t
```

Part Two

输出文件：output.bmp

Example2.t

```
import "darkroom"
darkroomSimple = require("darkroomSimple")
inputImage = darkroomSimple.load("cat.bmp")

im areaFilterX(x,y)
  [uint8[3]](
          inputImage(x-1,y)/3
          +inputImage(x,y)/3
          +inputImage(x+1,y)/3)
end

areaFilterX:save("output.bmp")
```

Command Line

```
$ terra Example2.t
```

Part Two

Programming model

1 坐标规定

允许以下坐标表示：
(1)位置(x+A,y+B)，其中A，B是常数
(2)gather操作，显式表示访问的范围。
仿射变换的坐标，像I(x*2,y*2)是不被允许的。这个约束意味着每一步处理消费（像素输入）和生产（像素运算并输出）的速率是一样的，这是行缓冲流水线所要求的。

2 不支持递归

图像函数不能是递归的。因为这样就必须在流水线内部对图像串行处理。

Darkroom语法规定

在darkroom中，图像函数可以用类lambda表达式的语法，im(x,y)。比如，一个简单的亮度调节操作可以写成以下形式：
brighter = im(x,y) I(x,y) * 1.1 end

为了实现模板（stencil），比如卷积，darkroom允许图像函数访问邻近的像素。

Darkroom

Part Three

Optimization

Part Three

Darkroom代码

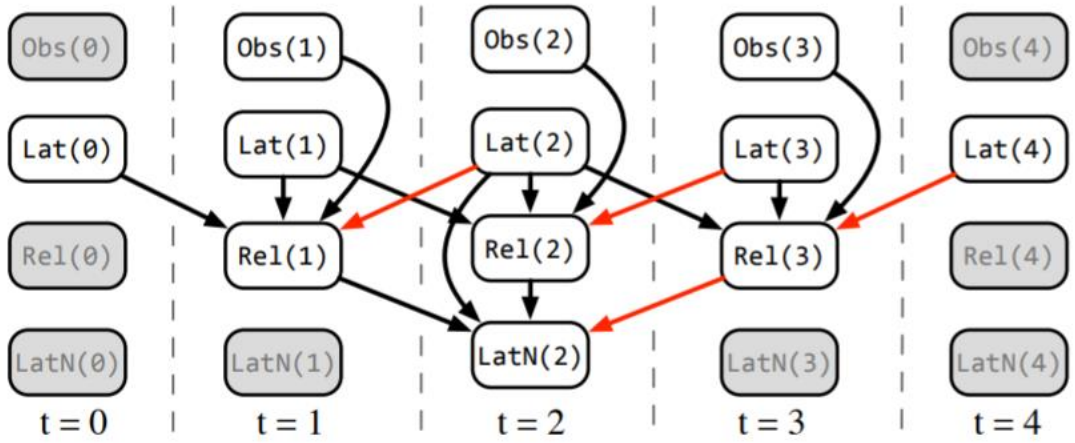
```
Rel = im(x) Obs(x) / (k0*Lat(x-1) + k1*Lat(x) + k2*Lat(x+1)) end
LatN = im(x) Lat(x)*(k2*Rel(x-1) + k1*Rel(x) + k0*Rel(x+1)) end
```

原始darkroom代码，对输入In进行了一维Richardson-Lucy卷积操作。

Part Three

Line Buffer

Rel = im(x) Obs(x) / (k_0*Lat(x-1) + k_1*Lat(x) + k_2*Lat(x+1)) end
LatN = im(x) Lat(x)*(k_2*Rel(x-1) + k_1*Rel(x) + k_0*Rel(x+1)) end

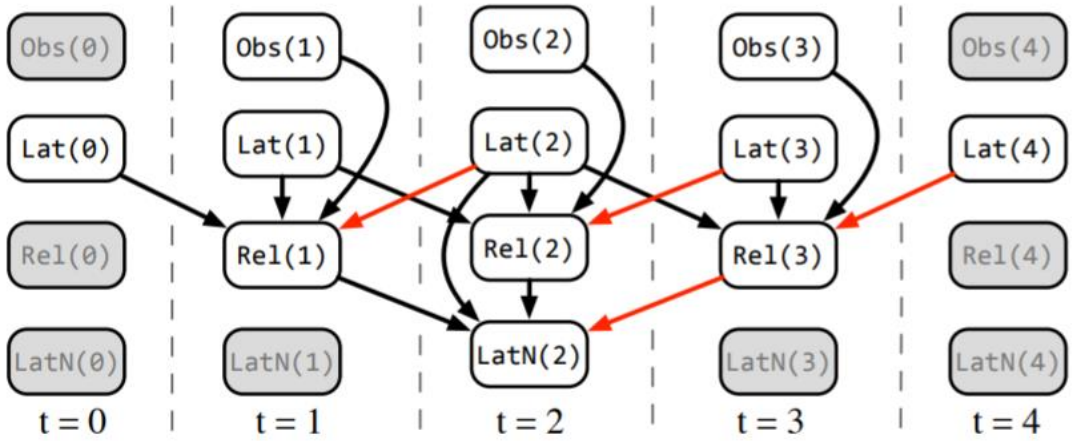


卷积的计算过程

Part Three

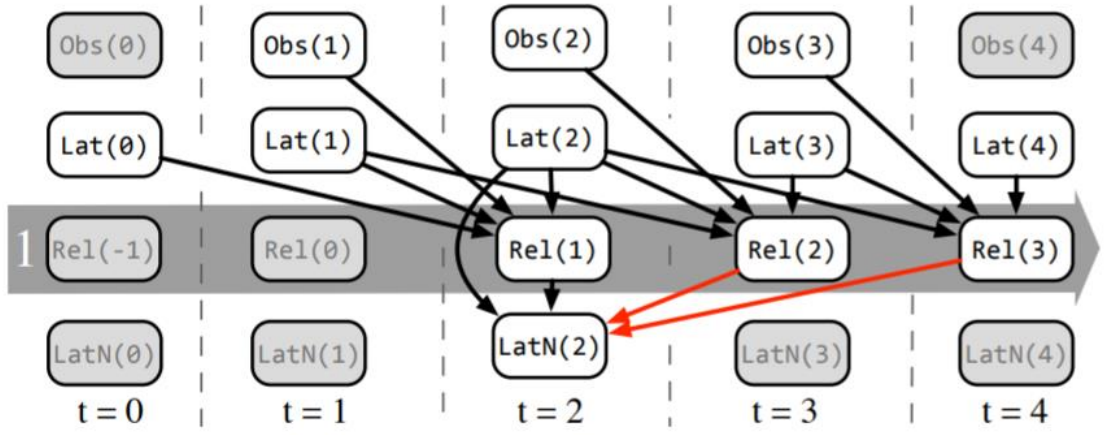
Line Buffer

Rel = im(x) Obs(x) / (k_0*Lat(x-1) + k_1*Lat(x) + k_2*Lat(x+1)) end
LatN = im(x) Lat(x)*(k_2*Rel(x-1) + k_1*Rel(x) + k_0*Rel(x+1)) end



卷积的计算过程

出现了non-casual流水线

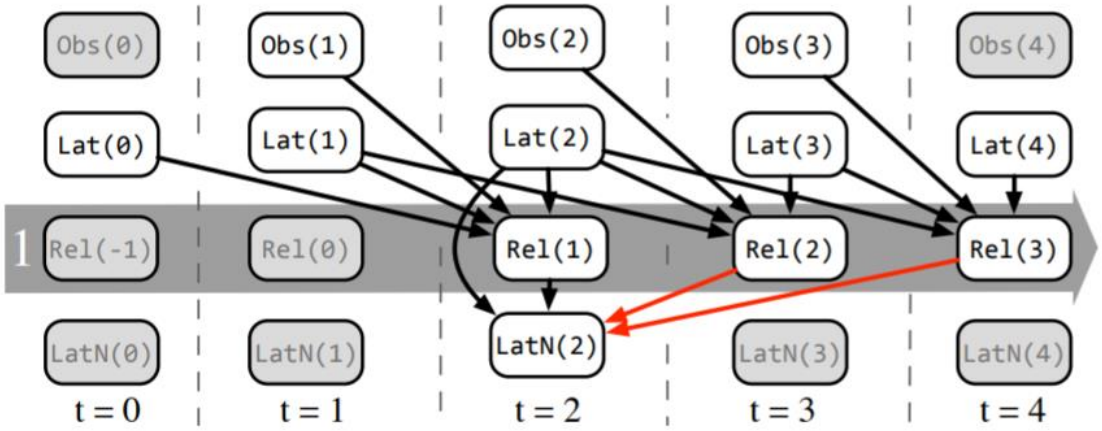


推迟Rel的计算

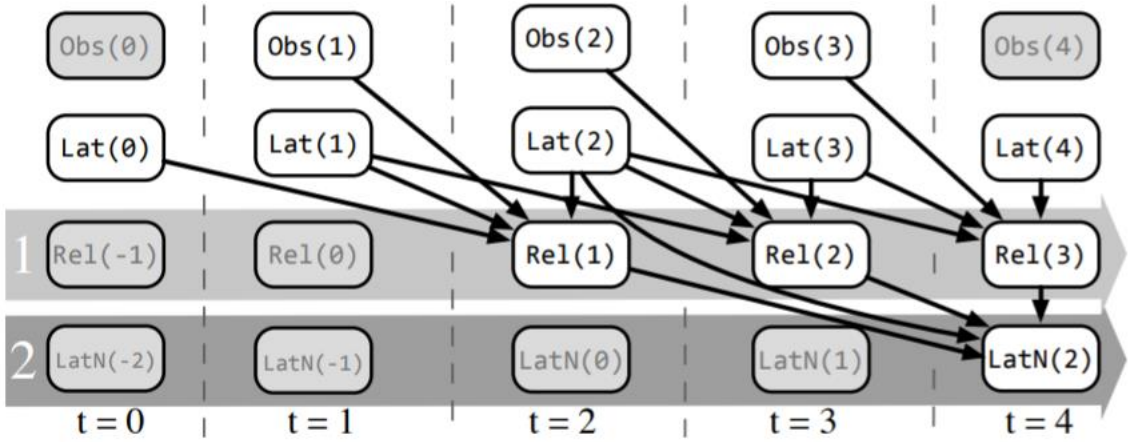
Part Three

Line Buffer

Rel = im(x) Obs(x) / (k_0*Lat(x-1) + k_1*Lat(x) + k_2*Lat(x+1)) end
LatN = im(x) Lat(x)*(k_2*Rel(x-1) + k_1*Rel(x) + k_0*Rel(x+1)) end



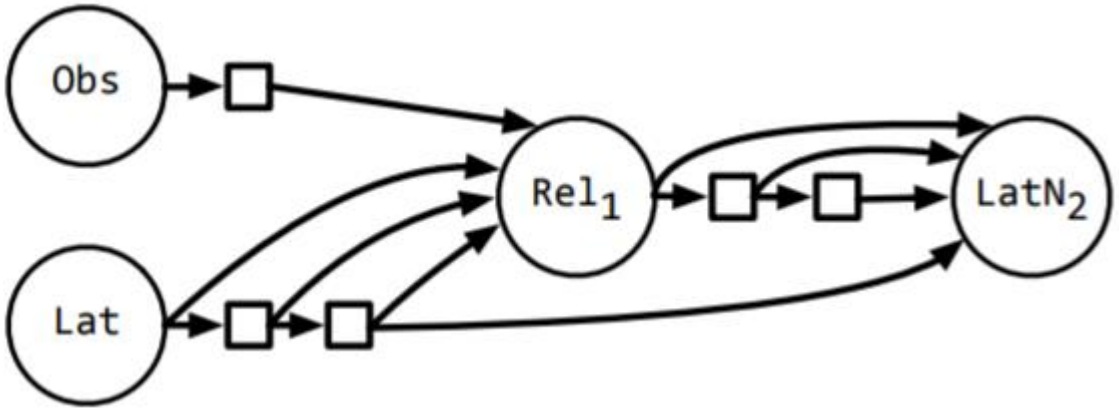
推迟Rel的计算



消除所有non-casual的情形

Part Three

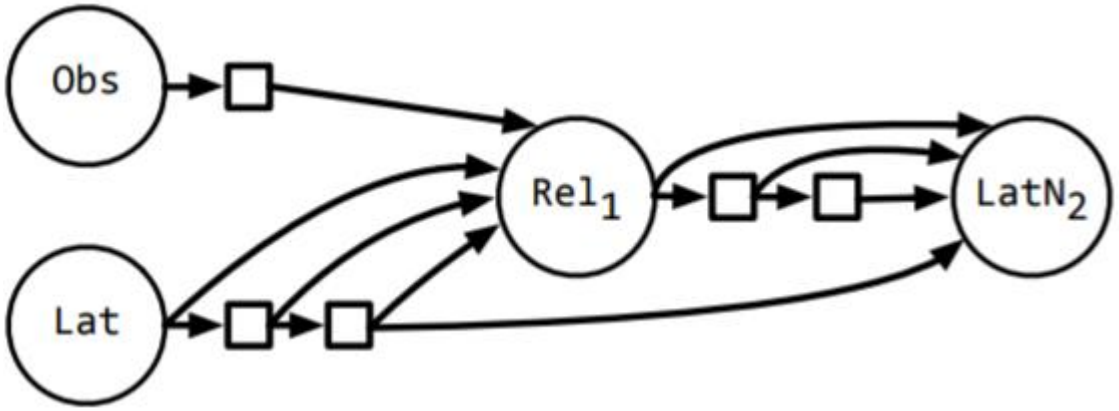
Line Buffer



对应的流水线，当前时刻的输入像素由左边的节点接收，同时行缓冲部分存储了前面计算输出的值，然后由这些值的组合运算得到当前时刻的输出。

Part Three

Shift Optimization



在最后的流水线里可以看到有一个缓冲节点用于存储前一时刻的Obs值，但如果我们将Obs后移一个时刻，则这个缓冲节点就不需要了。

可以通过选择合适的移位操作来保证流水线的因果性以及最小化缓冲长度。

Part Three

我们的目标是使用最少的buffer达到消除非casual计算的目的，这事实上是一个线性规划问题。

$$S = \sum_{p \in F} \left(\max_{(c,p,d) \in U} n_{(c,p,d)} \right) * b_p$$

通过线性规划求解器(ILP Solver)，可以求得进一步优化的buffer数量。

Darkroom

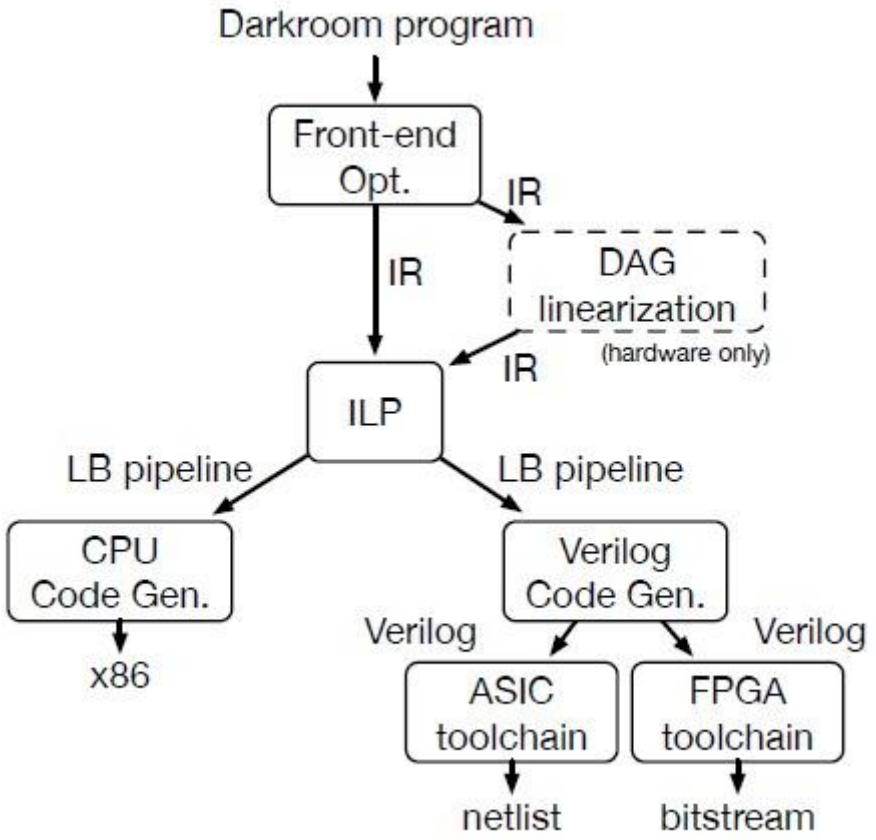
Part Four

Implementation & Optimization

Part Four

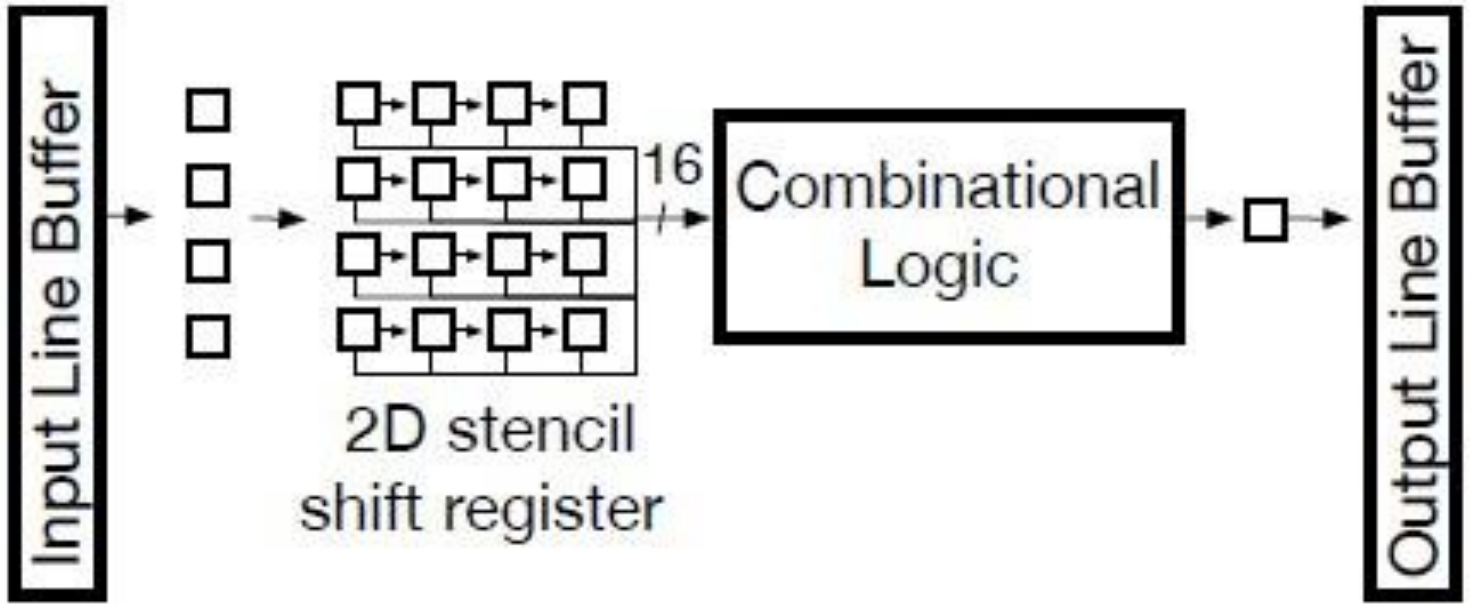
Implementation

- 编译器首先生成一份中间表示(IR)，用有向无环图(DAG)的形式表示高层次的stencil操作。
- 之后对其进行一般编译器的优化例如公共子表达式外提、常量传播等。
- 然后进行程序分析，生成和行缓冲优化等价的整数线性规划方程(ILP)。我们的使用已有的ILP求解器lpsolve来求解这一问题，以此生成优化后的流水线。
- 最后将流水线交给硬件代码生成器，生成ASIC或FPGA代码，或是CPU代码。



Part Four

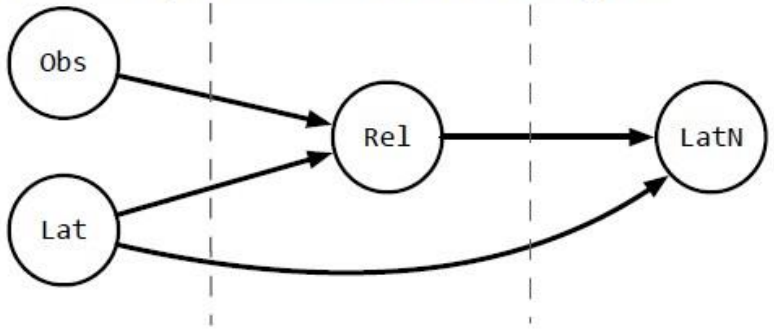
- 将每个行缓冲实现为一个循环SRAM或BRAM。每个clock一列像素数据从行缓冲进入到一个二维寄存器阵列。用户的图像函数实现为组合逻辑，把结果写到输出寄存器，



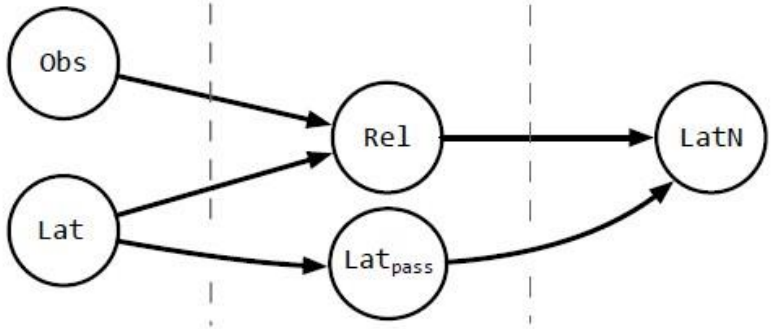
Part Four

- 实际的图像函数往往是多输入多输出的，我们采用结点合并的方法将其转化为单输入单输出的。

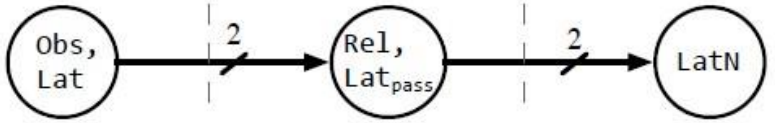
1. Group IR nodes by distance from the input:



2. Add passthrough nodes whenever an edge cross a stage:



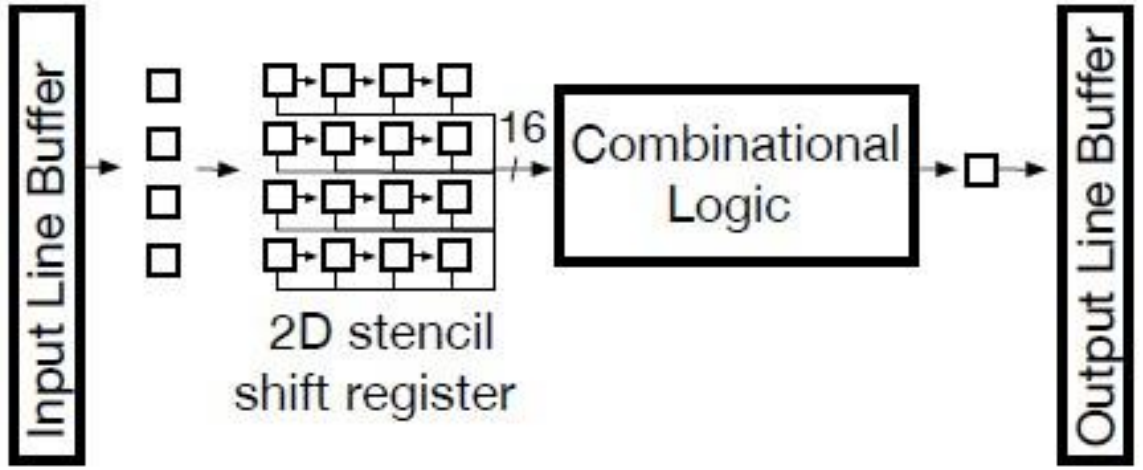
3. Merge values in each stage, producing larger pixel widths:



Part Four

- CPU编译器将行缓冲流水线实现为多线程函数。我们将输入图像分成许多strip然后分别计算。每个线程里，core遵循行缓冲流水线模型，一种简单的方法是每个clock计算一次主循环。

```
for each line y
    for each pixel x in line of S1
        compute S1(x, y)
    for each pixel x in line of S2
        compute S2(x, y) // loading S
    rotate line buffers
```



Part Four

CPU编译

- 整个行缓冲经常超过最高一级cache的大小。我们发现可以通过把计算阻塞在line这个粒度层面上可以提供cache的局部性。
- 我们在外层循环里对行缓冲做模运算，这样内层循环就可以包含尽可能少的指令。
- 如果发现硬件支持向量指令，也会在每个阶段每一行里实现向量计算。

```
for each line y
    for each pixel x in line of S1
        compute S1(x, y)
    for each pixel x in line of S2
        compute S2(x, y) // loading S1 from line buffer
    rotate line buffers
```

Part Four

Result

- ISP
- CORNER DETECTION
- EDGE DETECTION
- DEBLUR

Part Four

- ISP 包含基本的转换，比如去马赛克、白平衡、色矫正，以及一些增强和错误矫正操作，比如死像素点抑制。把ISP转换到Darkroom是简单而直接的：每个流水线对应一个转换成图像函数的stencil操作。

ISP

Part Four

Corner Detection

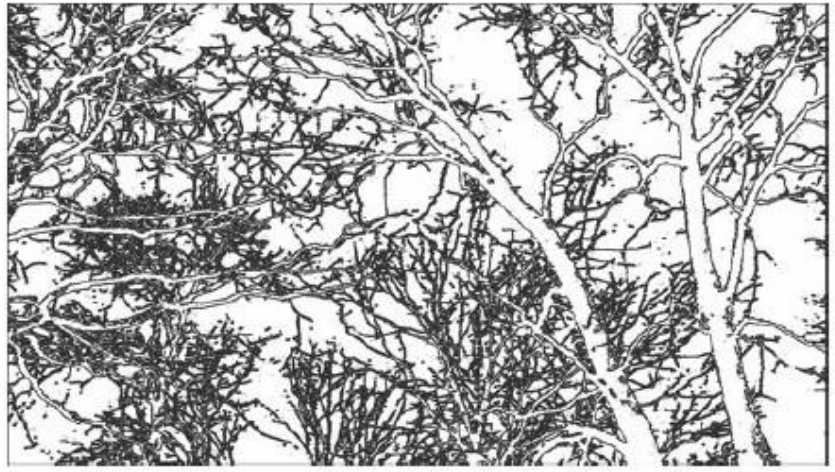
- 经典的角点检测算法，该算法经常用于计算机视觉算法的前期处理，实现为一系列局部stencil操作。
- 将与邻点亮度对比足够大的点定义为角点。

CORNER DETECTION

Part Four

Edge Detection

- 经典的边缘检测算法。
- 首先计算图像在x和y方向的梯度，然后对其分类，最终跟踪其中的序列化像素。
- 边缘检测需要长序列迭代，这和darkroom的模型不完全符合。实现该算法的目的是为了证明Darkroom在其不擅长的领域也能有比较好的表现。



EDGE DETECTION

Part Four

Deblur

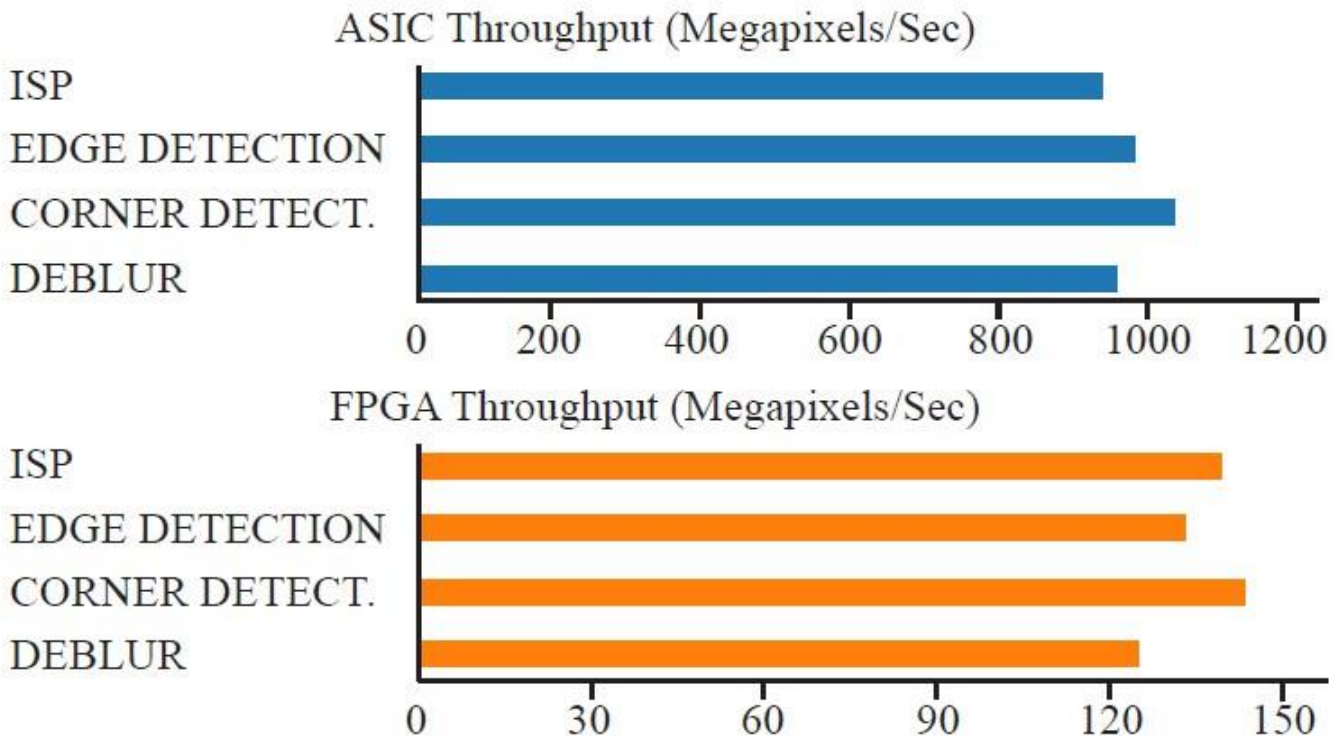
- 实现Richardson-Lucy non-blind deconvolution algorithm。
- 去模糊对迭代计算力的要求非常高，我们实现该算法的目的是对系统进行压力测试。
- 实现了8次迭代，这是由硬件所限制的。

DEBLUR

Part Four

吞吐量

- ASIC单流水线速度可以处理1600万像素的图像60 FPS.
- 在FPGA上这个数字是1080p/60.



Part Four

比较

- 对ISP，非多线程、非向量化的代码比开启了这些优化的代码，后者有七倍的加速效果。在这些加速效果中，3.5倍来自多线程，2倍来自向量化。

- Darkroom和Halide（一种现存的高性能的图像处理语言和编译器）相比，在去模糊这项应用上，二者的运行时时间是相似的。但是编译时间，Darkroom全部优化只用了不到一秒，总的编译时间不超过两分钟，但是Halide用了 八小时。

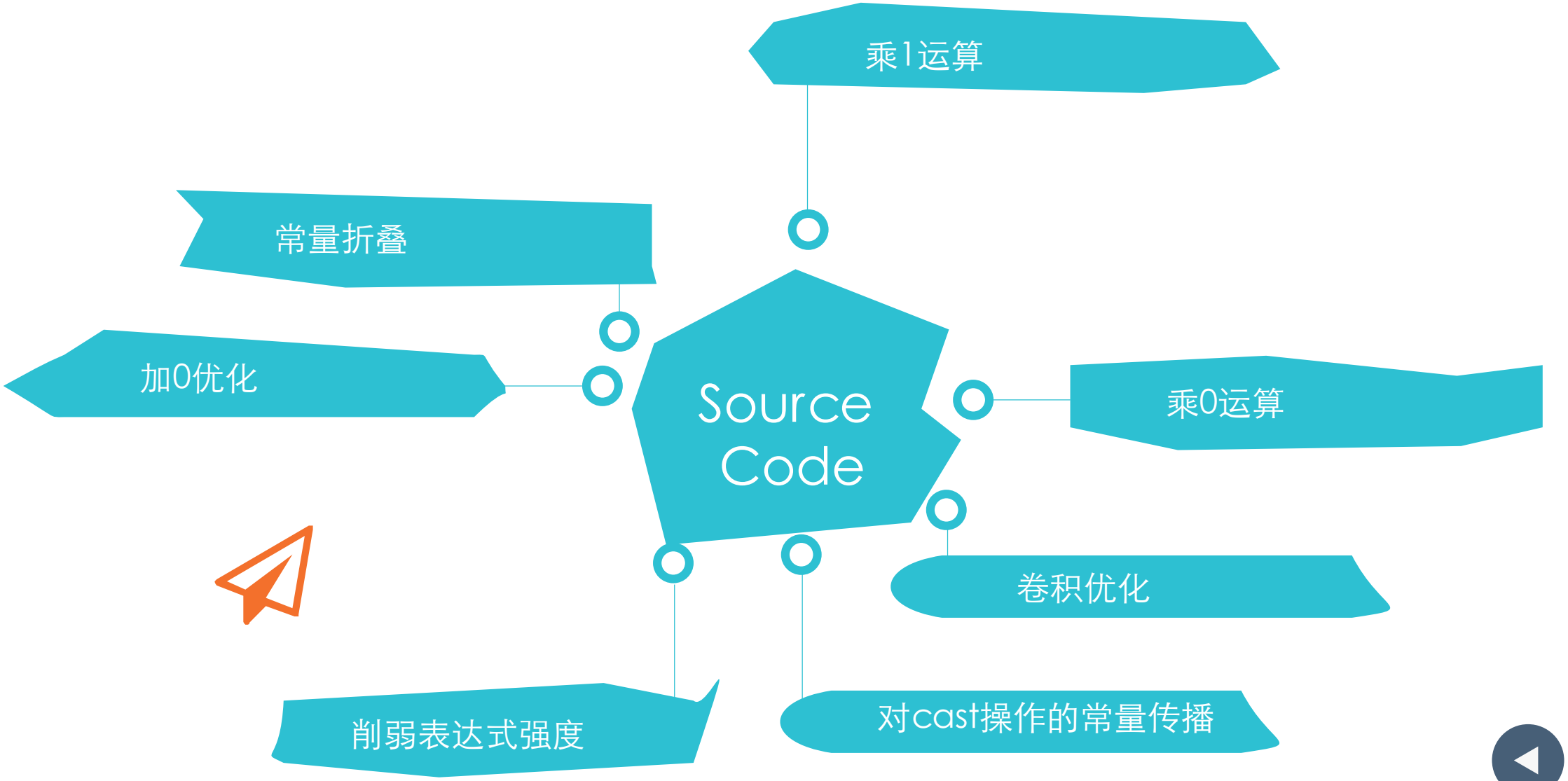
Darkroom

Part Five

Source Code Analysis

Part Five

Source Code Analysis



Part Five

Source Code Analysis

加0优化

加0返回本身。

```
if ast.op=="+" and
lhs.kind=="value"
and
darkroom.optimize.is
Zero(lhs.value) then
    return ast.rhs
```

常量折叠

对能够求值的立即数运算在编译时求值。

```
function
darkroom.optimiz
e.constantFold(ast)
```

强度削弱

对二的幂次的乘除转为移位运算。
但有一个问题：如
对负数-109（10010011）移位运算后得到11001001=-55，但除法运算-109/2=-54

乘0/1运算

乘0则返回0。
乘1返回本身。

源码实现中乘0是返回为0的那个操作数。
乘1返回被乘1的数。

```
elseif ast.op=="*" and
ast.rhs.kind=="value"
and ast.rhs.value==1
then
    return ast.lhs
```

Part Five

Source Code Analysis

卷积优化

对（a+b）/3形式转化为a/3+b/3

$$(Img \overset{CE}{*} f)[n,m] \overset{\text{def}}{=} R_{|l|<c}\{R_{|k|<c}\{Map(Img[k], f[n-k, m-l])\}\}$$

这样有利于卷积引擎（convolutional engine）在map运算中填充更多项（即要除以3的项更多）。

Part Five

cast优化

Darkroom提供显式的数据类型转换，对此进行优化。

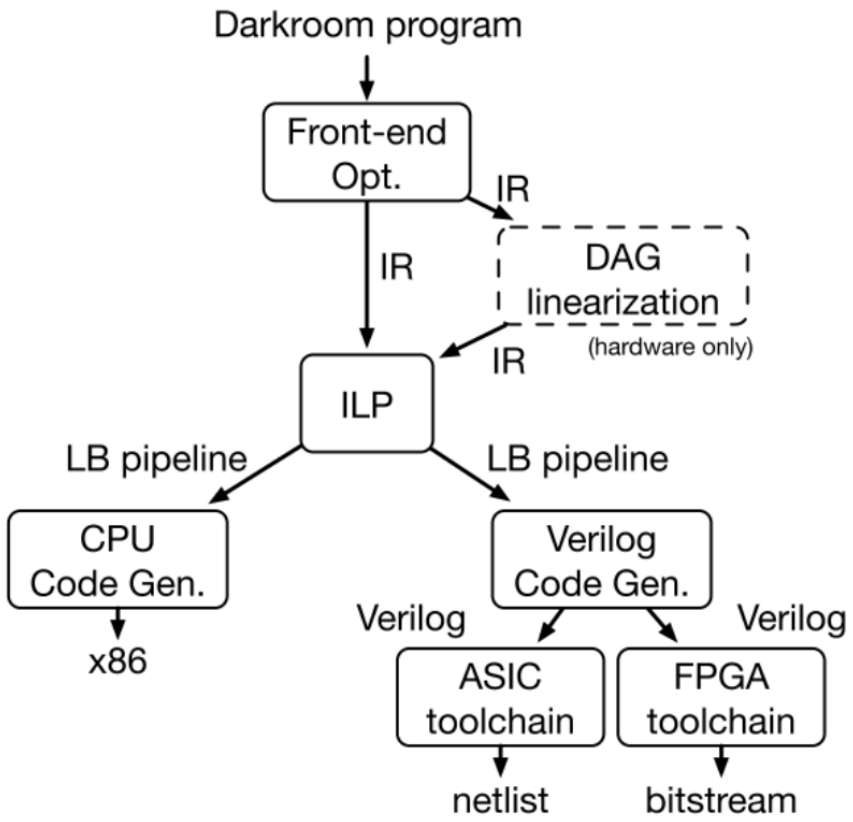
类型转换示例：
[uint8[3]](inputImage(x,y)*0.9)
意思是，将括号中的inputImage(x,y)*0.9转换为一个uint8类型的，长度为3的数组。

如果编译时可以得到数组的长度信息，而且被转换的是常数，那么就直接建立一个数组，并复制常数值到每一项。
这种优化用于二元运算对操作数的处理。

对应源码：
```
for i=1,darkroom.type.arrayLength(ast.type) do newval[i] = ast.expr.value end
```

Part Five

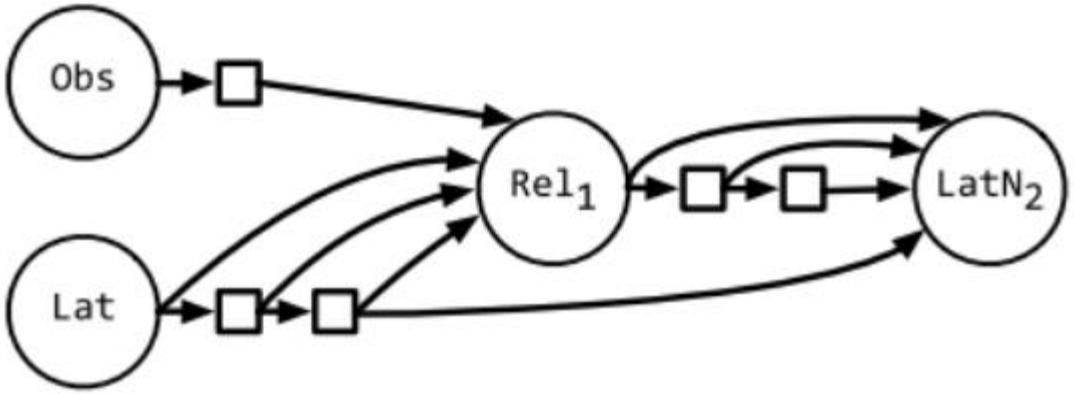
Source Code Analysis



Darkroom程序 ->IR（stencil操作的DAG) -> DAG上做一些优化 -> 产生偏移 -> 优化偏移 -> 后端 -> terra编译产生底层CPU代码（包括向量、线程）-> LLVM编译优化

Part Five

Source Code Analysis



schedule.t中schedule函数:
对该结点每一个输入，找到该输入最晚的引用，再加上该输入目前的偏移，就是该结点的偏移。

schedule.t中shift函数:
偏移计算两次，第二次利用线性规划求解器计算出优化偏移。

由偏移即可算出流水线层级之间的缓存大小。

Darkroom

Part Six

Intergroup Discussion

Part Six

NNVM

Q： 什么是nnvm？它提供什么样的功能?

A：NNVM是一个开源的深度学习编译器，它可以将前端框架的工作直接编译到硬件后端。

Q: nnvm工作的大体流程是什么样的?

A：前端将不同的神经网络框架写成的程序统一成计算图表示，然后进行经典的图优化，之后调用tvm进行进一步的优化。最后根据不同的硬件生成对应的代码。

Part Six

Q. 能具体说明用了哪些优化的手段吗?

A：主要是针对循环cache命中率的优化，向量化和并行化。

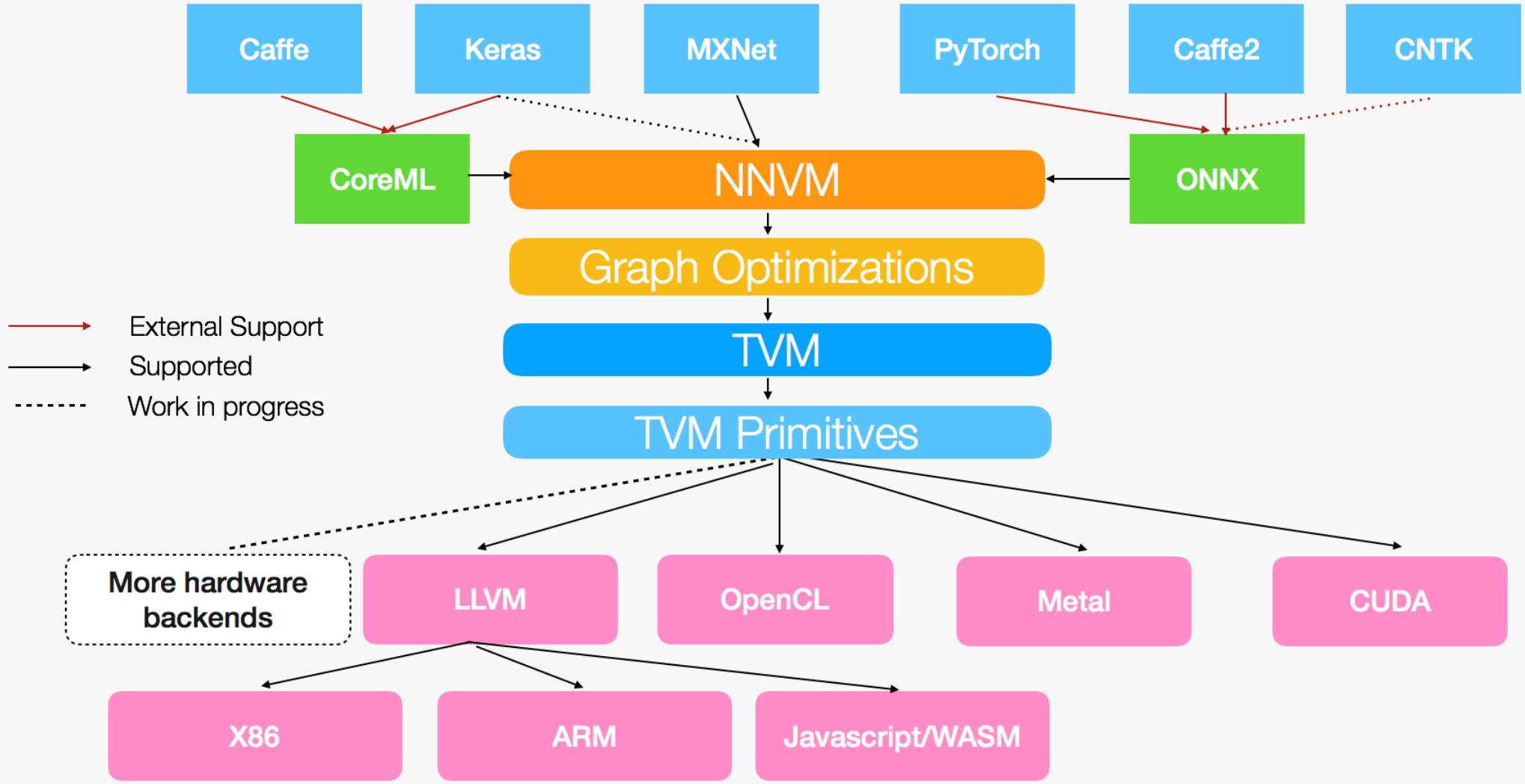
循环cache命中率的提高可以通过循环平铺，让访问空间尽量连续。

向量化是基于连续访存时的一种优化手段，通过cpu硬件指令，单指令多数据同时算，速度比一个个算快很多

gpu有共享缓存可以被所有线程块访问，所以在多线程的时候，这样可以避免多次读入，而且速度很快。这个共享缓存是用户自己维护的，所以 tvm 在这上面给了这种优化方案，但具体因为矩阵有时候会很大，不能全放进去，所以会每次只塞一部分进去。

Part Six

NNVM



Part Six

NNVM和Darkroom的区别与联系

1. 二者都有作为编译器的一面，通过对比可以发现编译器工程上的一些共性特征。比如要面对不同的前端或后端，这时会采用中间表示，并且在中间表示上会进行独立于机器的优化，在直面硬件时根据不同的硬件进一步优化。

2. 二者的优化手段类似。一方面是经典的编译优化手段，比如常量传播、公共子表达式外提、死代码删除；另一方面是二者都之间面对硬件，所以会细致的考察Cache利用率和向量化等问题。

3. NNVM侧重于深度神经网络，而Darkroom则是图像处理领域的专用语言，二者在应用上交集取决于两个领域的重叠程度，但是在编译器这个领域却有很多相似之处。在比较中学习使得我们能够发现不变的东西。

Part Six

与Halide组交流问答

Q:Halide的构建依赖于什么?

A:Halide的构建依赖于C++。

Q:Halide针对图像处理做了哪些方面的优化?

A:优化核心思想为解耦算法和优化，以几分之一的代码量实现出同等或者数倍于手工C++代码的效能，提高了代码的可维护性和开发效率。

Part Six

Darkroom和Halide的异同

语言	依赖	图像处理领域专用语言	并行	流水线与缓冲	Map&reduce	图像金字塔等算法
Darkroom	Terra(llvm)	是	无明确的规划或限制	主要优化途径	有	无法简单处理
Halide	C++(llvm)	是	可以规划	可以规划	有	可以处理

Part Six

测试比较：均值滤波

Part Six

测试比较：均值滤波

语言	编译用时	读写	读写+滤波	实际滤波
Darkroom	/	0.65s	0.70s	0.05s
Halide	3.5s	0.17s	0.79s	0.62s

注：
1. Halide的滤波程序用g++编译后运行，Darkroom的滤波程序用terra解释执行。
2. Halide的规划部分没有写，实际效果可能等同于无优化的直接滤波代码。

Part Six

测试比较：均值滤波

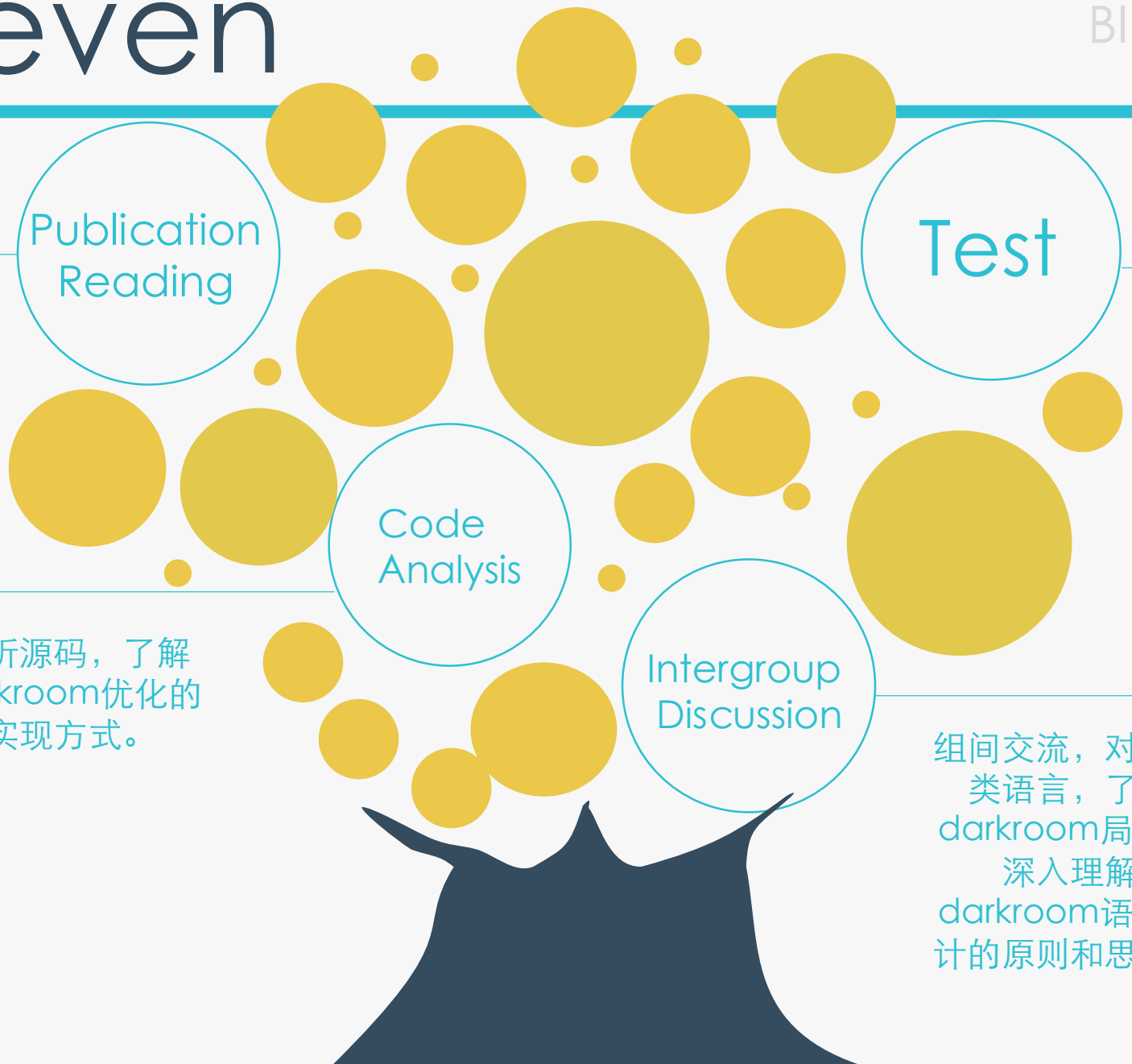
结论：
1. halide的编译时间比较长（在用g++编译来执行的情况下）
2. darkroom对图片文件的读写效率比halide低很多
3. darkroom实现的均值滤波效率较高，halide由于没有编写合适的规划代码，效率较低
4. 另外halide可能对边界的处理不是很智能。

Darkroom

Part Seven

Summary

Part Seven



Publication Reading

通过阅读darkroom论文，以及相关论文，了解有关概念，并掌握了darkroom原理。

Test

样例测试
Darkroom&Halide
样例对比测试

Code Analysis

分析源码，了解darkroom优化的实现方式。

Intergroup Discussion

组间交流，对比同类语言，了解darkroom局限，深入理解darkroom语言设计的原则和思想。