AI编译器-后端优化

算子计算与调度

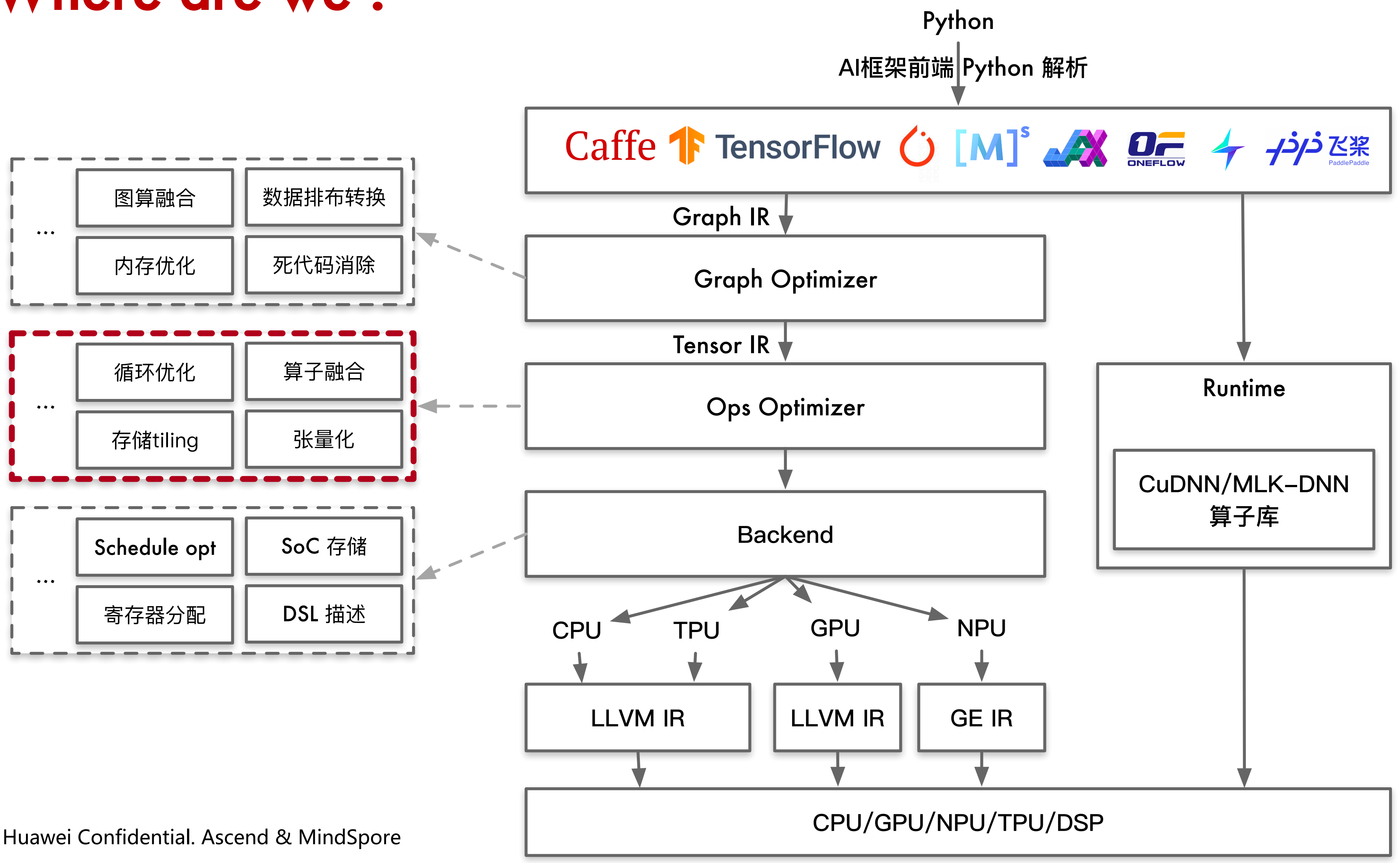
ZOMI

Talk Overview

I. AI 编译器后端优化

- 后端优化概念
- 算子计算与调度
- 算子循环优化
- Auto-Tuning
- Polyhedral

Where are we ?


Python
AI框架前端 Python 解析
Caffe
TensorFlow
ONEFLOW
飞桨
PaddlePaddle
Graph IR
Graph Optimizer
Tensor IR
Ops Optimizer
Backend
CPU
TPU
GPU
NPU
LLVM IR
LLVM IR
GE IR
Runtime
CuDNN/MLK-DNN 算子库
CPU/GPU/NPU/TPU/DSP
…
图算融合
数据排布转换
内存优化
死代码消除
…
循环优化
算子融合
存储tiling
张量化
…
Schedule opt
SoC 存储
寄存器分配
DSL 描述

算子
计算与调度

What is Operator?

算子：

- 深度学习算法由一个个计算单元组成，我们称这些计算单元为算子（Operator，简称Op）。算子是一个函数空间到函数空间上的映射O：X→Y；从广义上讲，对任何函数进行某一项操作都可以认为是一个算子。于AI 框架而言，所开发的算子是网络模型中涉及到的计算函数。

算法：

- 算法（Algorithm）是指解题方案的准确而完整的描述，是一系列解决问题的清晰指令，算法代表着用系统的方法描述解决问题的策略机制。

算子的计算与调度

- **计算**是算子的定义，回答算子是什么，如何通过具体的算法得到正确的定义结果。
- **调度**是算子的执行策略和具体实现，回答系统具体如何执行算子的计算定义。

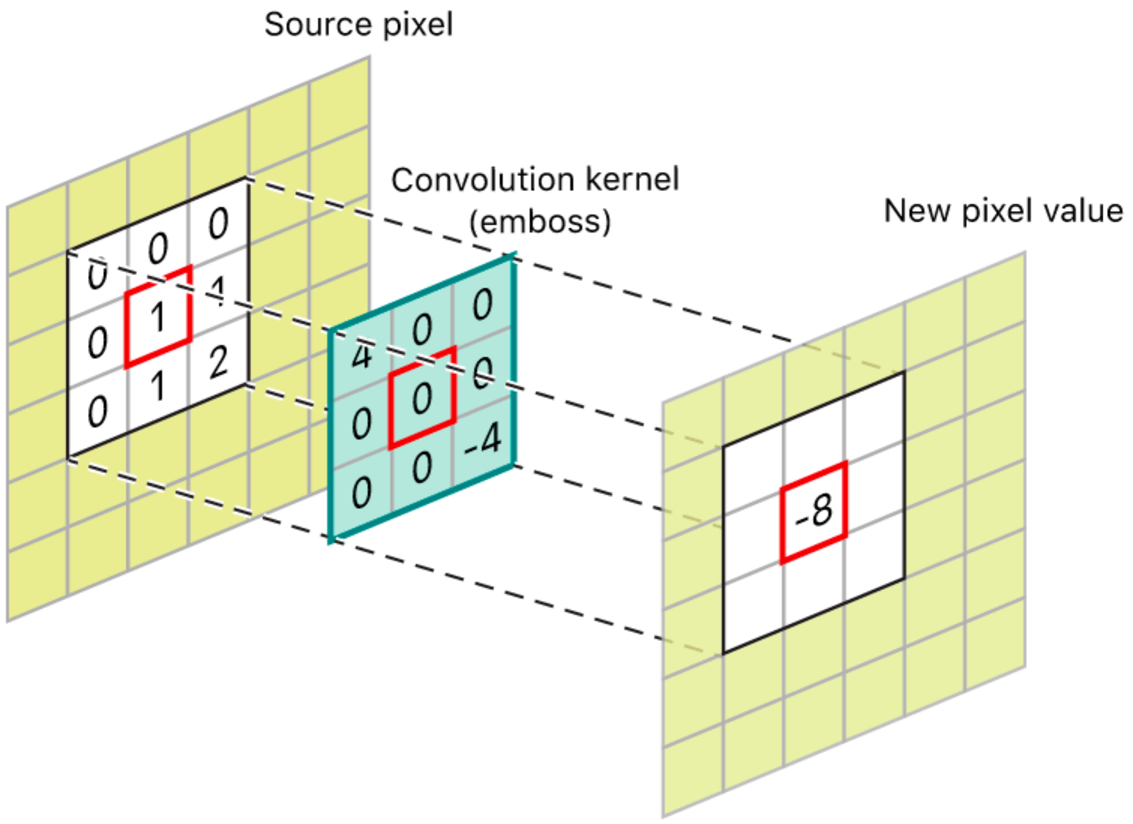
- 同一算子会有不同的实现方式（即不同的调度方式），但是只会有一种计算形态（具体定义）。
- 计算实现（Function / Expression）和计算在硬件单元上的调度（Schedule）是分离。

算子的计算

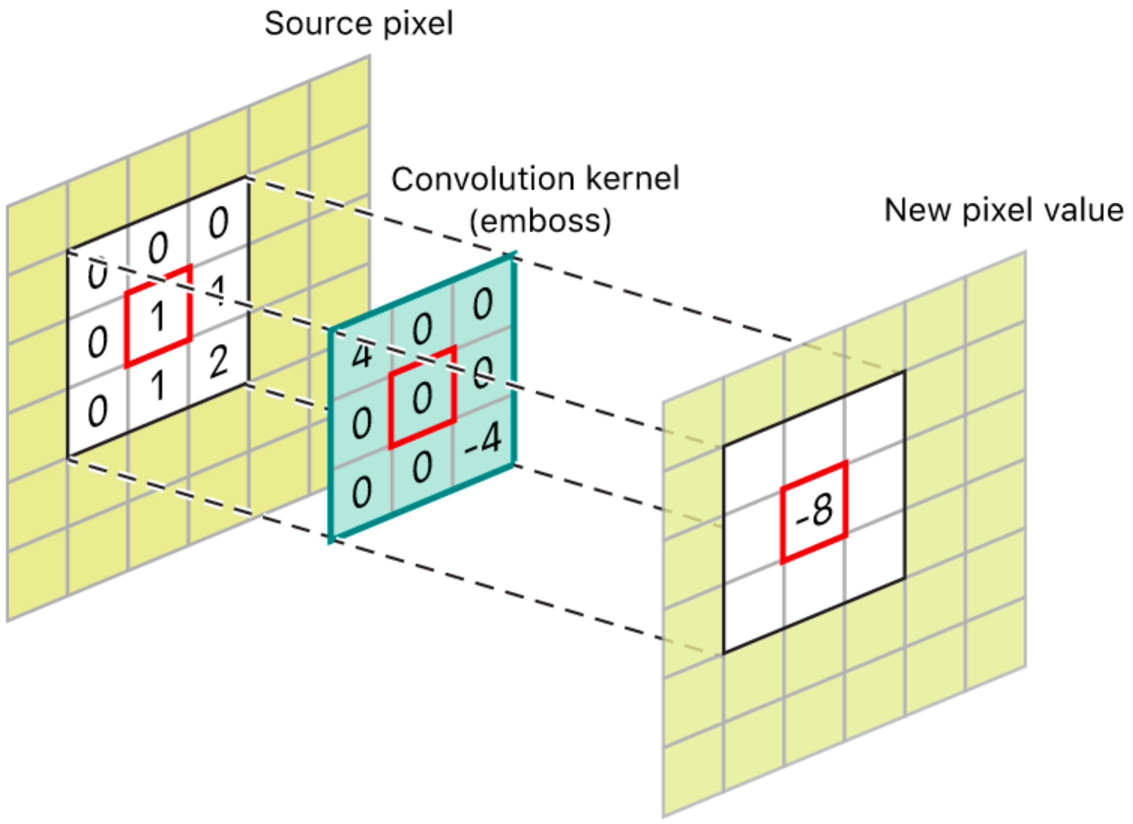
- 高斯滤波

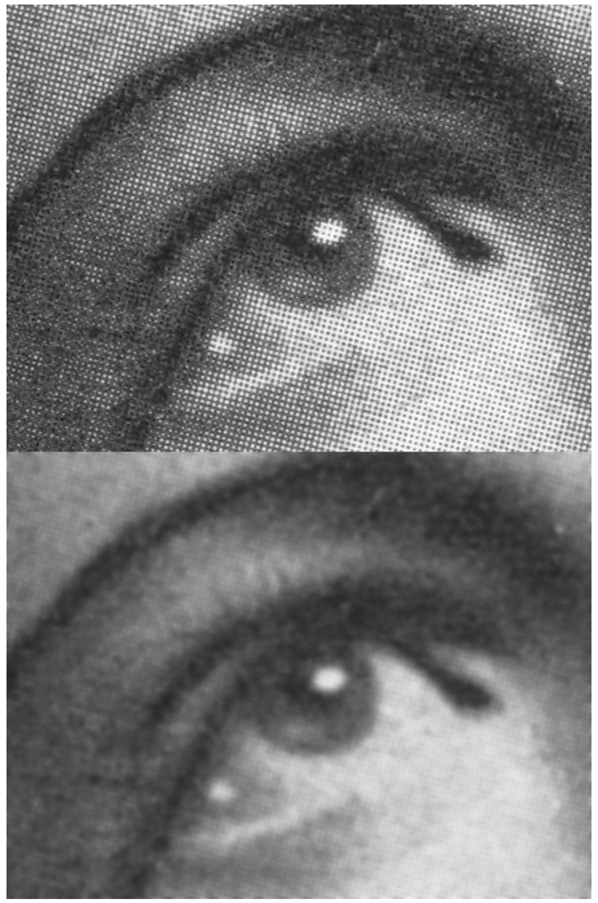
$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$



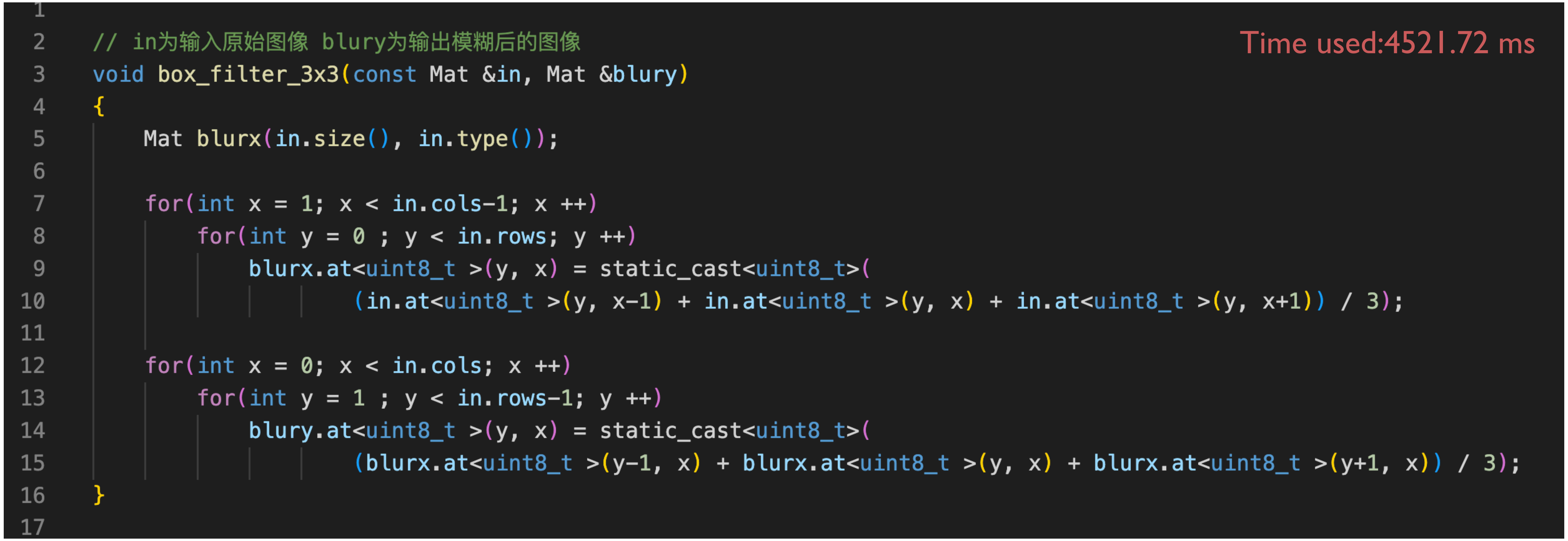
算子的计算


Source pixel
Convolution kernel (emboss)
New pixel value
0 0 0
0 1 1
0 1 2
4 0 0
0 0 0
0 0 -4
-8



算子的调度

- 对图像进行模糊的操作函数：首先在x轴上对每个像素点以及周围的两个点进行求和平均，然后再到y轴上进行同样的操作，这样相当于一个3×3平均卷积核对整个图像进行操作。

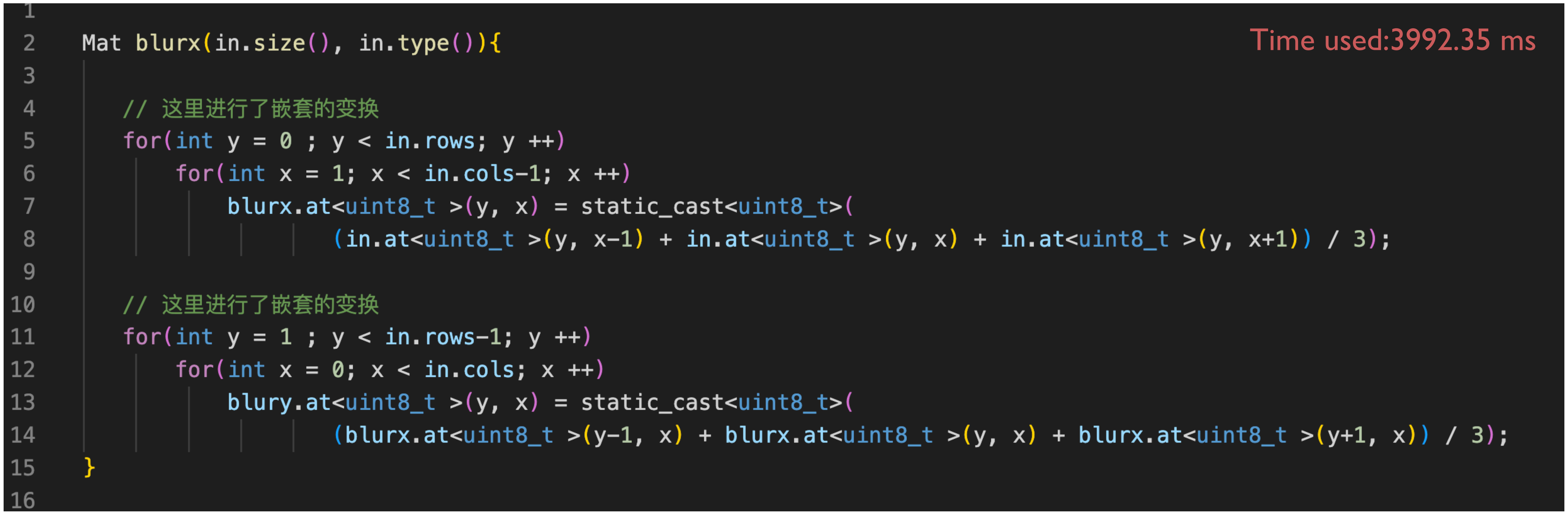


Time used:4521.72 ms

```cpp

// in为输入原始图像 blury为输出模糊后的图像
void box_filter_3x3(const Mat &in, Mat &blury)
{
    Mat blurx(in.size(), in.type());

    for(int x = 1; x < in.cols-1; x ++)
        for(int y = 0 ; y < in.rows; y ++)
            blurx.at<uint8_t >(y, x) = static_cast<uint8_t>(
                    (in.at<uint8_t >(y, x-1) + in.at<uint8_t >(y, x) + in.at<uint8_t >(y, x+1)) / 3);

    for(int x = 0; x < in.cols; x ++)
        for(int y = 1 ; y < in.rows-1; y ++)
            blury.at<uint8_t >(y, x) = static_cast<uint8_t>(
                    (blurx.at<uint8_t >(y-1, x) + blurx.at<uint8_t >(y, x) + blurx.at<uint8_t >(y+1, x)) / 3);
}

```

算子的调度

- 简单改变一下执行次序，将上述循环嵌套中的x和y的顺序改变一下：



Time used:3992.35 ms

```cpp

Mat blurx(in.size(), in.type()){

    // 这里进行了嵌套的变换
    for(int y = 0 ; y < in.rows; y ++)
        for(int x = 1; x < in.cols-1; x ++)
            blurx.at<uint8_t >(y, x) = static_cast<uint8_t>(
                    (in.at<uint8_t >(y, x-1) + in.at<uint8_t >(y, x) + in.at<uint8_t >(y, x+1)) / 3);

    // 这里进行了嵌套的变换
    for(int y = 1 ; y < in.rows-1; y ++)
        for(int x = 0; x < in.cols; x ++)
            blury.at<uint8_t >(y, x) = static_cast<uint8_t>(
                    (blurx.at<uint8_t >(y-1, x) + blurx.at<uint8_t >(y, x) + blurx.at<uint8_t >(y+1, x)) / 3);
}

```

Why Difference?

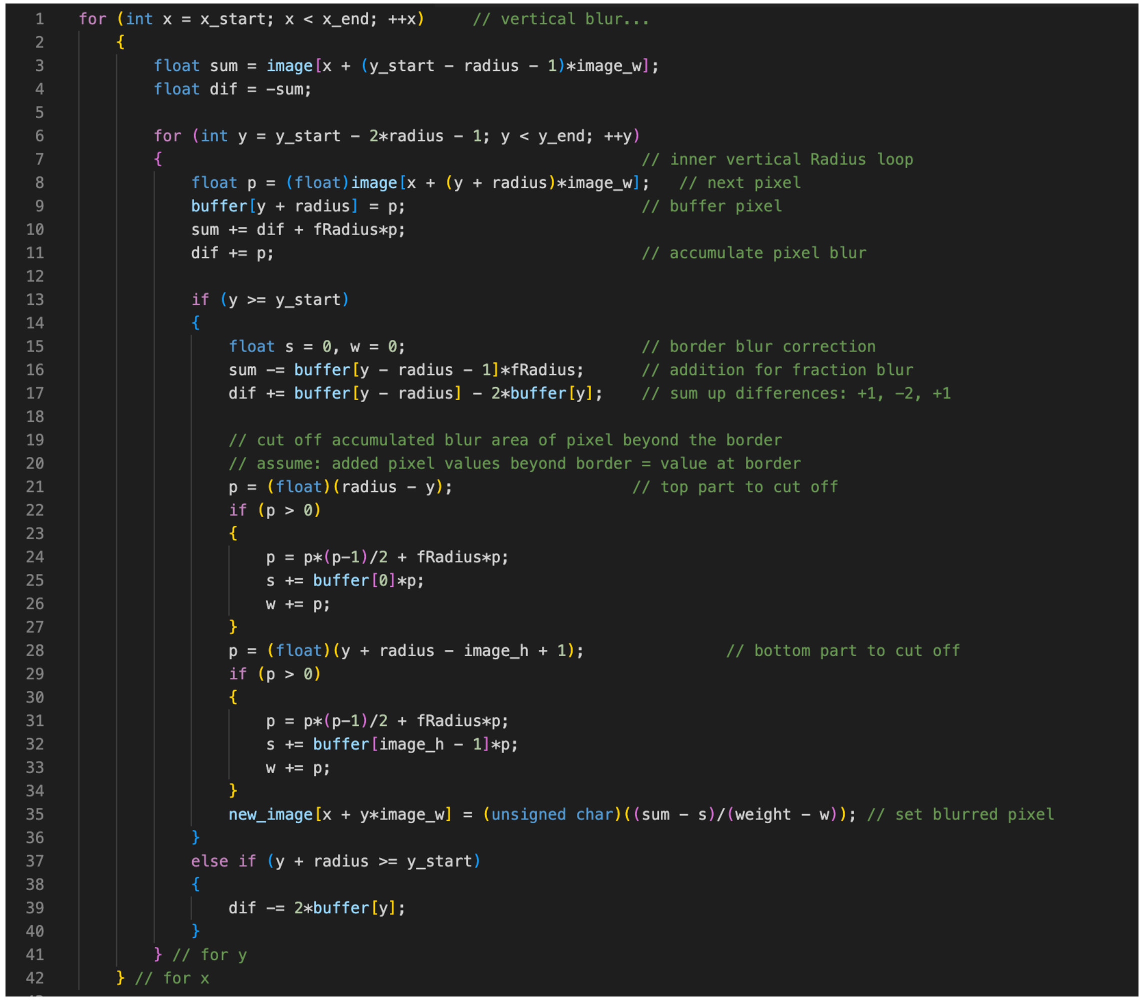
- 上述两种操作执行的算法功能是一样的，但是速度为什么会有差别。究其原因，这差别和算法本身没什么关系，而与硬件的设计是有巨大关系，例如并行性和局部性。

- 在硬件层面上极致优化结果，比之前的算法快了10倍，其中用到了SIMD(单指令多数据流)、以及平铺（Tiling）、展开（Unrolling）和向量化（Vectorization）等常用技术。充分利用了硬件的性能，从而不改变算法本身设计的前提下最大化提升程序执行的速度。

算子的调度

在硬件层面上极致优化结果，比之前的算法快了10倍，其中用到了SIMD（单指令多数据流）、以及平铺（Tiling）、展开（Unrolling）和向量化（Vectorization）等常用技术。

充分利用了硬件的性能，从而不改变算法本身设计的前提下最大化提升程序执行的速度。


```
for (int x = x_start; x < x_end; ++x)     // vertical blur...
    {
        float sum = image[x + (y_start - radius - 1)*image_w];
        float dif = -sum;

        for (int y = y_start - 2*radius - 1; y < y_end; ++y)
        {                                                     // inner vertical Radius loop
            float p = (float)image[x + (y + radius)*image_w];   // next pixel
            buffer[y + radius] = p;                           // buffer pixel
            sum += dif + fRadius*p;
            dif += p;                                         // accumulate pixel blur

            if (y >= y_start)
            {
                float s = 0, w = 0;                           // border blur correction
                sum -= buffer[y - radius - 1]*fRadius;        // addition for fraction blur
                dif += buffer[y - radius] - 2*buffer[y];      // sum up differences: +1, -2, +1

                // cut off accumulated blur area of pixel beyond the border
                // assume: added pixel values beyond border = value at border
                p = (float)(radius - y);                     // top part to cut off
                if (p > 0)
                {
                    p = p*(p-1)/2 + fRadius*p;
                    s += buffer[0]*p;
                    w += p;
                }
                p = (float)(y + radius - image_h + 1);                  // bottom part to cut off
                if (p > 0)
                {
                    p = p*(p-1)/2 + fRadius*p;
                    s += buffer[image_h - 1]*p;
                    w += p;
                }
                new_image[x + y*image_w] = (unsigned char)((sum - s)/(weight - w)); // set blurred pixel
            }
            else if (y + radius >= y_start)
            {
                dif -= 2*buffer[y];
            }
        } // for y
    } // for x
```

算子的计算与调度

- 算子调度具体执行的所有可能的调度方式称为调度空间，AI 编译器优化的目的就是为算子提供一种最优的调度，使得算子在硬件上运行时间最优。

调度树
schedule trees

调度策略 Schedule Primitives

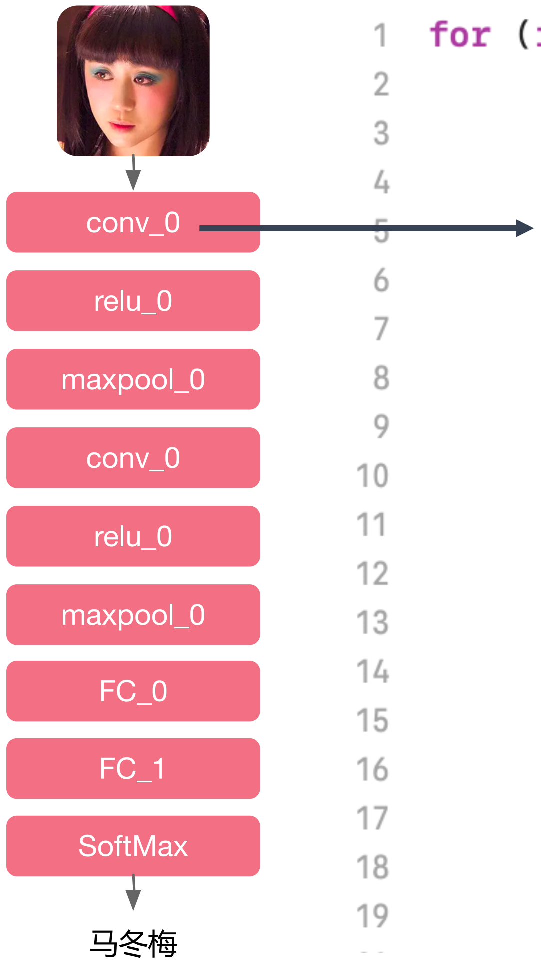
Halide：

- Reorder(交换)、Split(拆分)、Fuse(融合)、Tile(平铺)、Vector(向量化)、展开(Unrolling)、并行(Parallelizing)

TVM：

- Reorder(交换)、Split(拆分)、Fuse(融合)、Tile(平铺)、Vector(向量化)、展开(Unrolling)、绑定(binding)

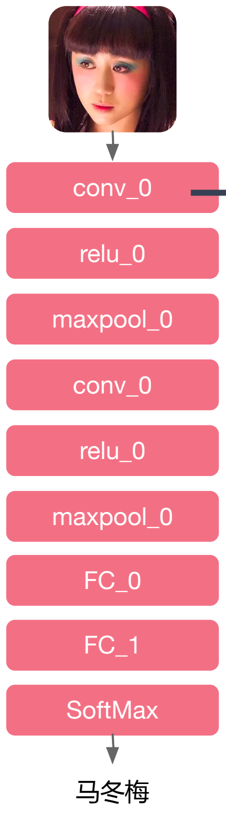
计算特征



```
for (int n = 0; n < o_n; ++n) {
        for (int c = 0; c < o_c; ++c) {
            for (int j = 0; j < o_h; ++j) {
                for (int i = 0; i < o_w; ++i) {
                    int d_start = n * i_c * i_h * i_w + j * i_w + i;
                    int temp = 0;
                    for (int kk = 0; kk < k_c; ++kk) {
                        for (int kj = 0; kj < k_h; ++kj) {
                            for (int ki = 0; ki < k_w; ++ki) {
                                int k_idx = kk * k_h * k_w + kj * k_w + ki;
                                int d_idx = d_start + kk * i_h * i_w + kj * i_w + ki;
                                temp += inputs->data[d_idx] * kernel->data[k_idx];
                            }
                        }
                    }
                    res[n * o_c * o_h * o_w + j * o_w + i] = temp;
                }
            }
        }
```

计算特征

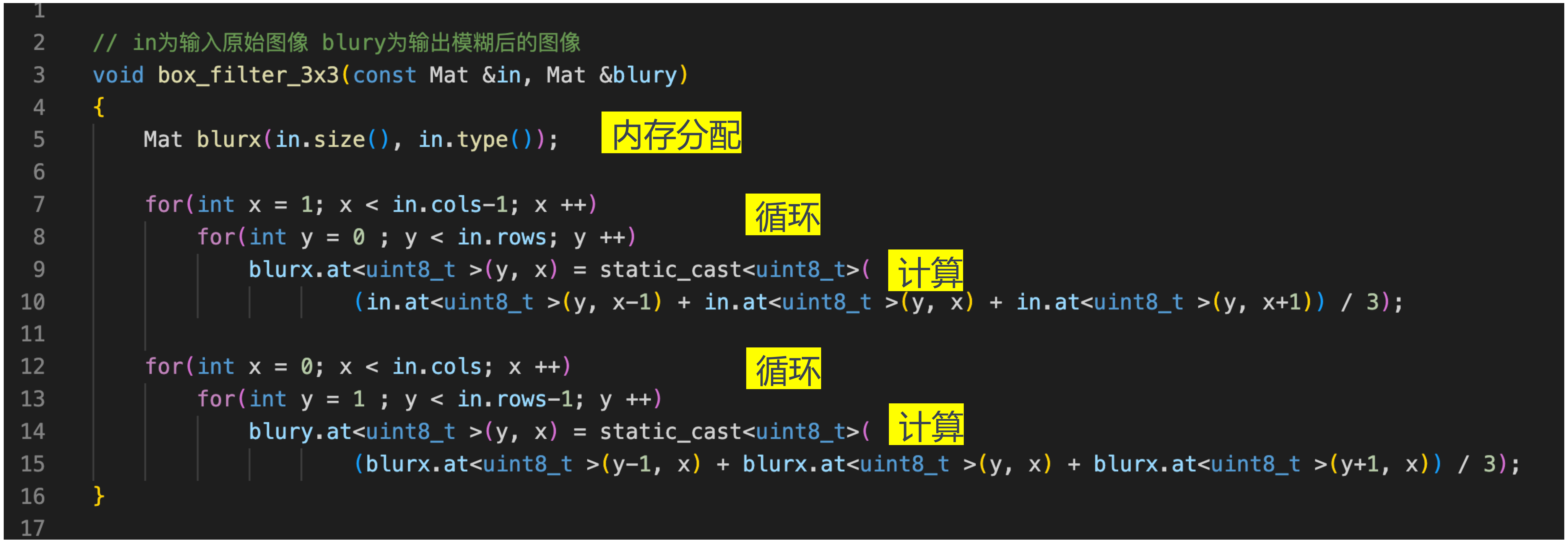
- 1）多重循环为特点；2）没有复杂控制流；3）以多维张量计算为主要数据；



```
for (int n = 0; n < o_n; ++n) {
    for (int c = 0; c < o_c; ++c) {
      for (int j = 0; j < o_h; ++j) {
        for (int i = 0; i < o_w; ++i) {
          int d_start = n * i_c * i_h * i_w + j * i_w + i;
          int temp = 0;
          for (int kk = 0; kk < k_c; ++kk) {
            for (int kj = 0; kj < k_h; ++kj) {
              for (int ki = 0; ki < k_w; ++ki) {
                int k_idx = kk * k_h * k_w + kj * k_w + ki;
                int d_idx = d_start + kk * i_h * i_w + kj * i_w + ki;
                temp += inputs->data[d_idx] * kernel->data[k_idx];
              }
            }
          }
          res[n * o_c * o_h * o_w + j * o_w + i] = temp;
        }
      }
    }
```

Schedule trees

- 算子调度程序主要由三种代码结构组成：



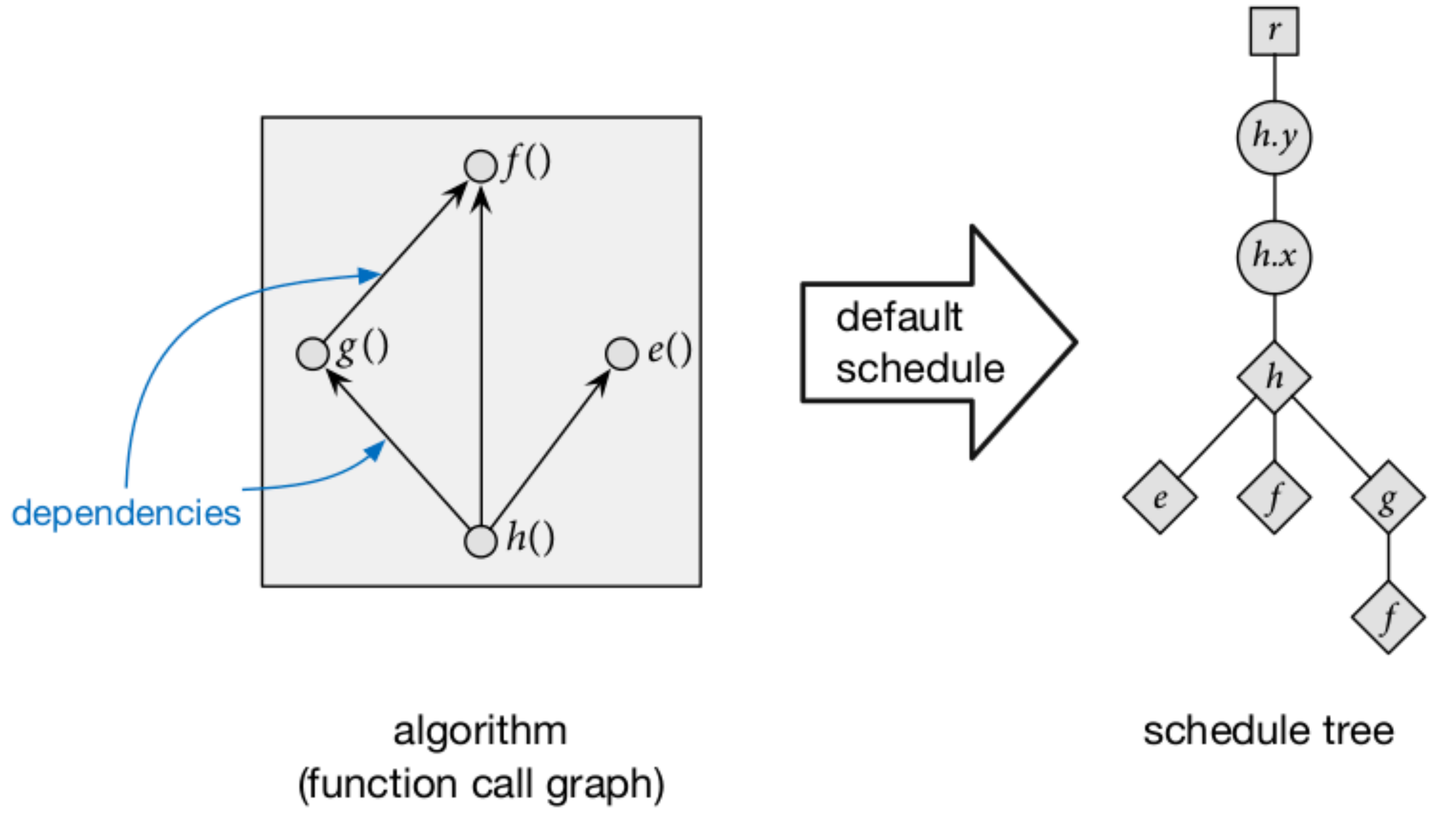
```cpp

// in为输入原始图像 blury为输出模糊后的图像
void box_filter_3x3(const Mat &in, Mat &blury)
{
  Mat blurx(in.size(), in.type());

  for(int x = 1; x < in.cols-1; x ++)
    for(int y = 0 ; y < in.rows; y ++)
      blurx.at<uint8_t >(y, x) = static_cast<uint8_t>(
          (in.at<uint8_t >(y, x-1) + in.at<uint8_t >(y, x) + in.at<uint8_t >(y, x+1)) / 3);

  for(int x = 0; x < in.cols; x ++)
    for(int y = 1 ; y < in.rows-1; y ++)
      blury.at<uint8_t >(y, x) = static_cast<uint8_t>(
          (blurx.at<uint8_t >(y-1, x) + blurx.at<uint8_t >(y, x) + blurx.at<uint8_t >(y+1, x)) / 3);
}

```

Schedule trees

- **Root nodes** represent the top of the schedule tree.
- **Loop nodes** represent the traversal of how the function is computed along a given dimension. Loop nodes are associated with a function and a variable (dimension). Since functions are assumed two-dim ensional, by default functions have two variables: *x* and *y*. Loop nodes also contain information such as whether the loop is run sequentially, run in parallel, or vectorized.
- **Storage nodes** represent storage for intermediate results to be used later.
- **Compute nodes** are the leaves of the schedule tree, and they represent computation being perform ed. Compute nodes can have other compute nodes as children to represent functions that are inlined instead of loaded from intermediate storage.

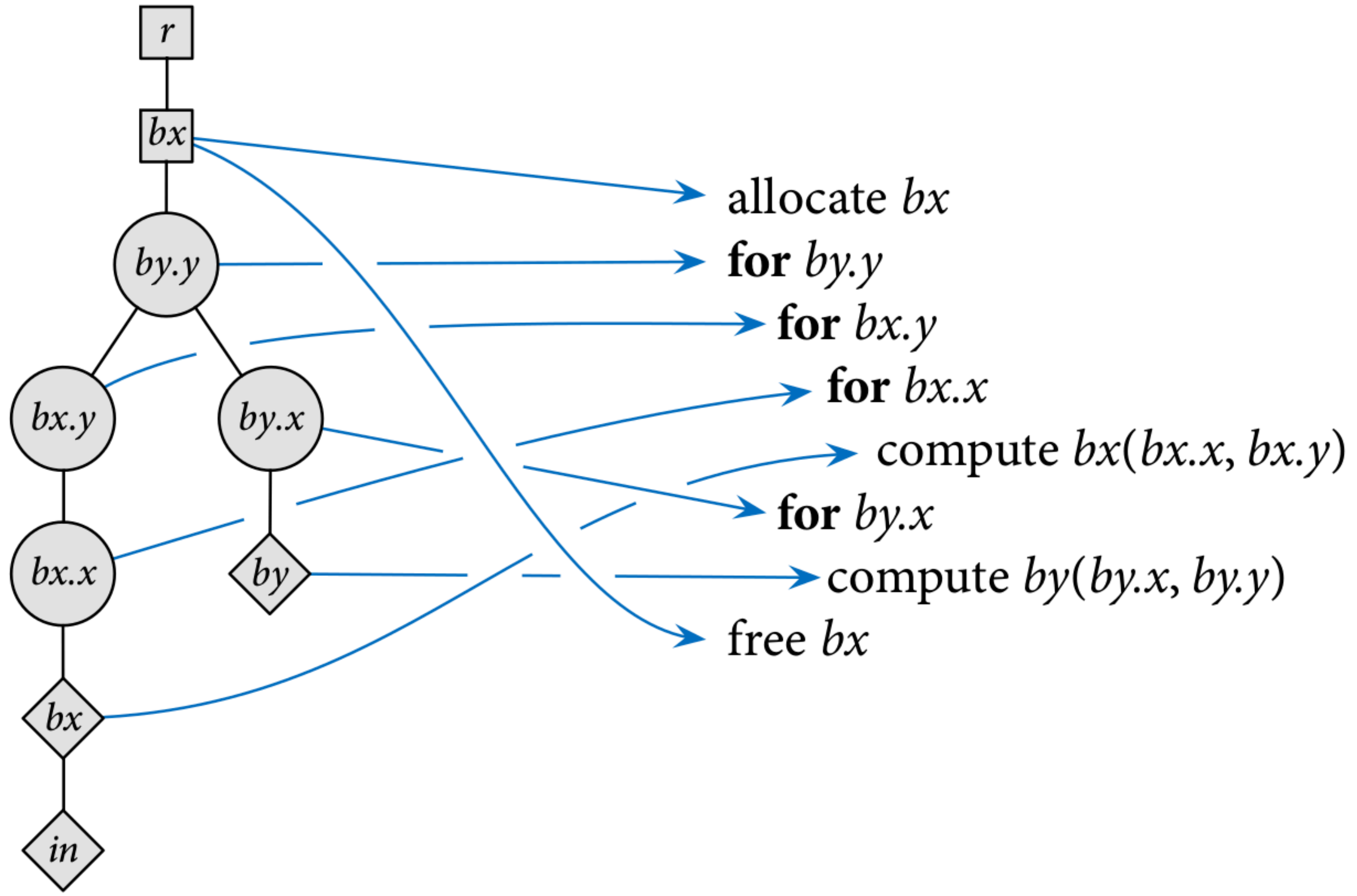
Schedule trees

- For any function we can define the *default schedule*, which traverses the output function in row-major order and inlines all called functions, like so:



Schedule trees

- Root nodes
- Loop nodes
- Storage nodes
- Compute nodes



Question?

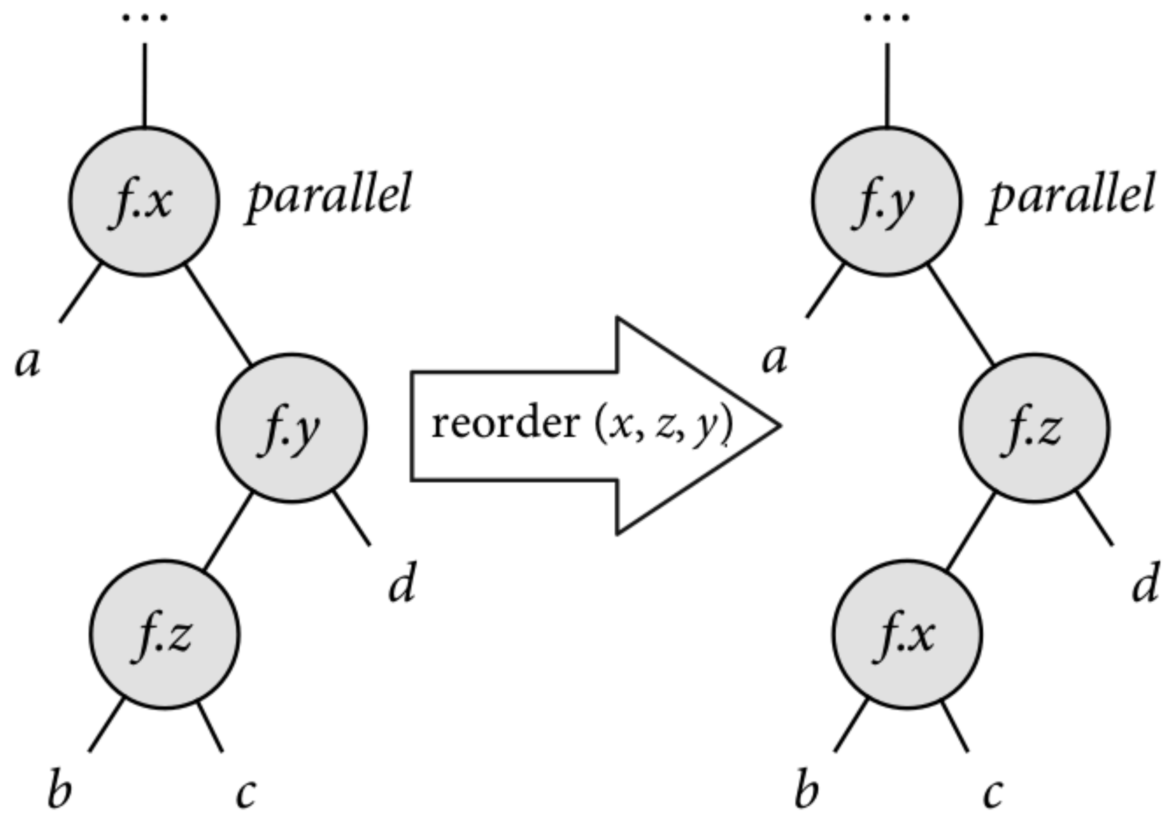
- 如何理解调度树的语义？

- 对调度树进行优先遍历 DFS，然后转换成对应程序代码。

调度转换
Schedule Transformers

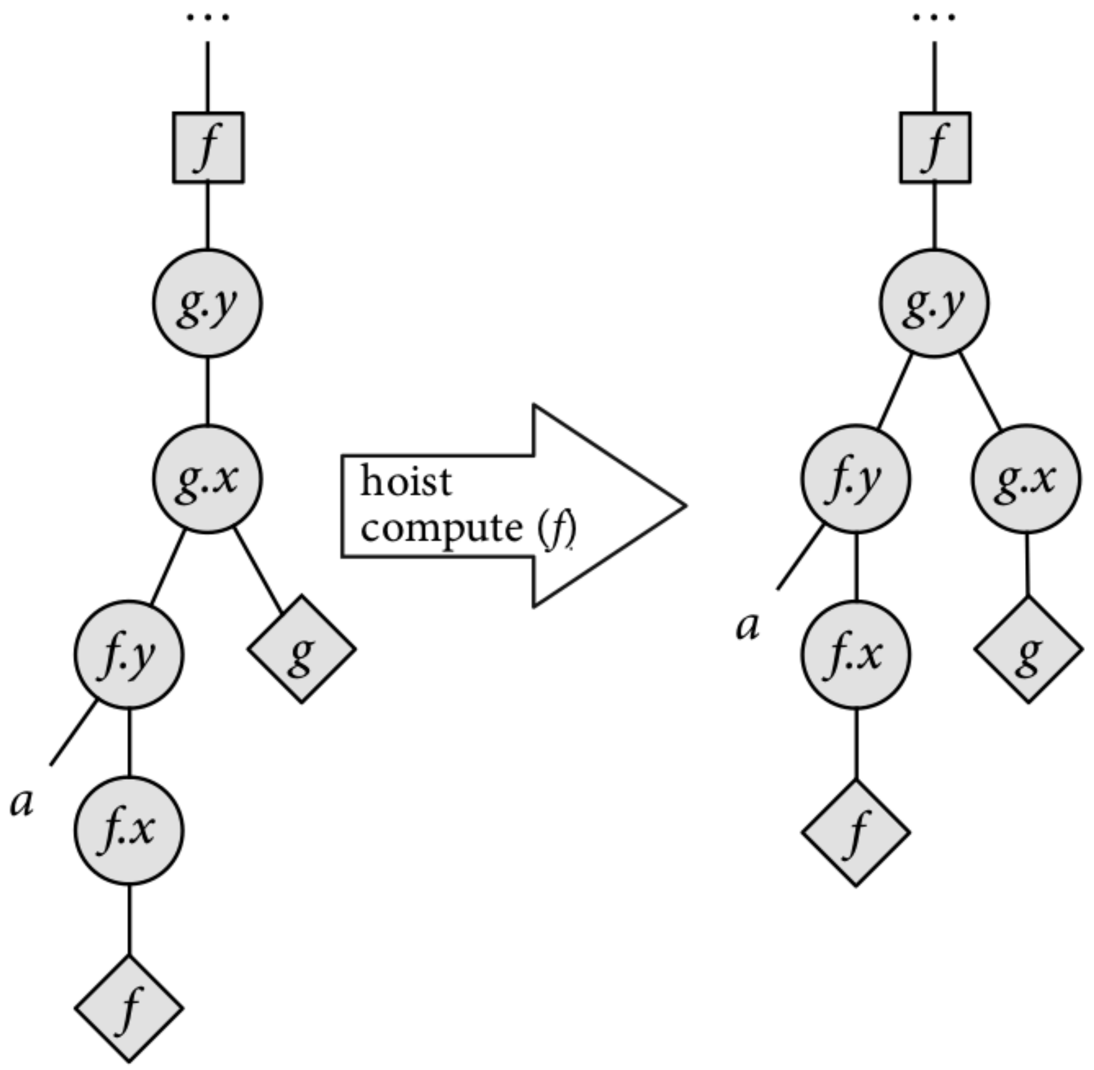
Schedule Transformers

- **Reorder -** switch loop nodes for the same function.为相同的功能切换循环节点



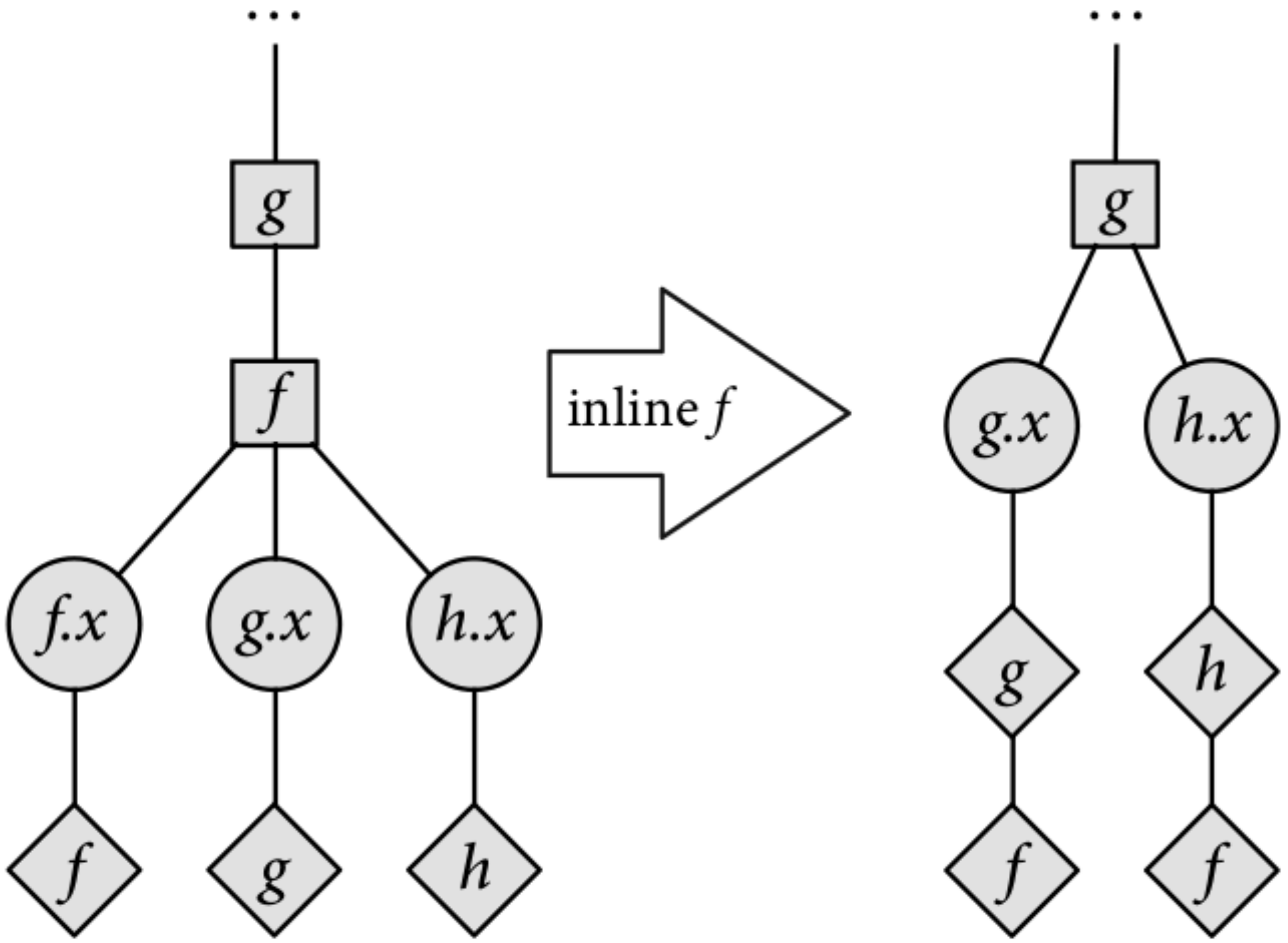
Schedule Transformers

- **Hoist / lower compute -** change the granularity in which intermediate results are computed. 更改计算中间结果的粒度



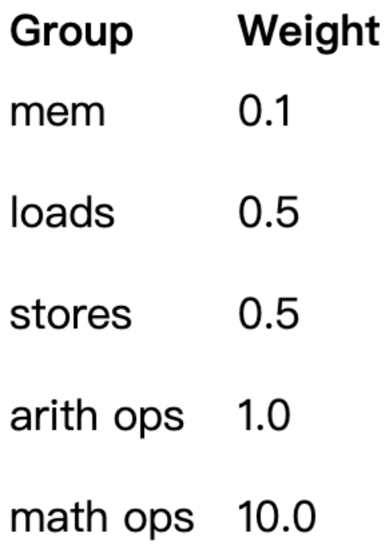
Schedule Transformers

- **Inline / deinline -** inline functions into callers. 将函数内联到调用者中（不要将它们的结果存储在中间存储中）或从调用者中取消函数。直观上，内联函数以较小的内存使用量换取冗余计算，而去内联函数以较高的内存使用量换取较少的冗余计算

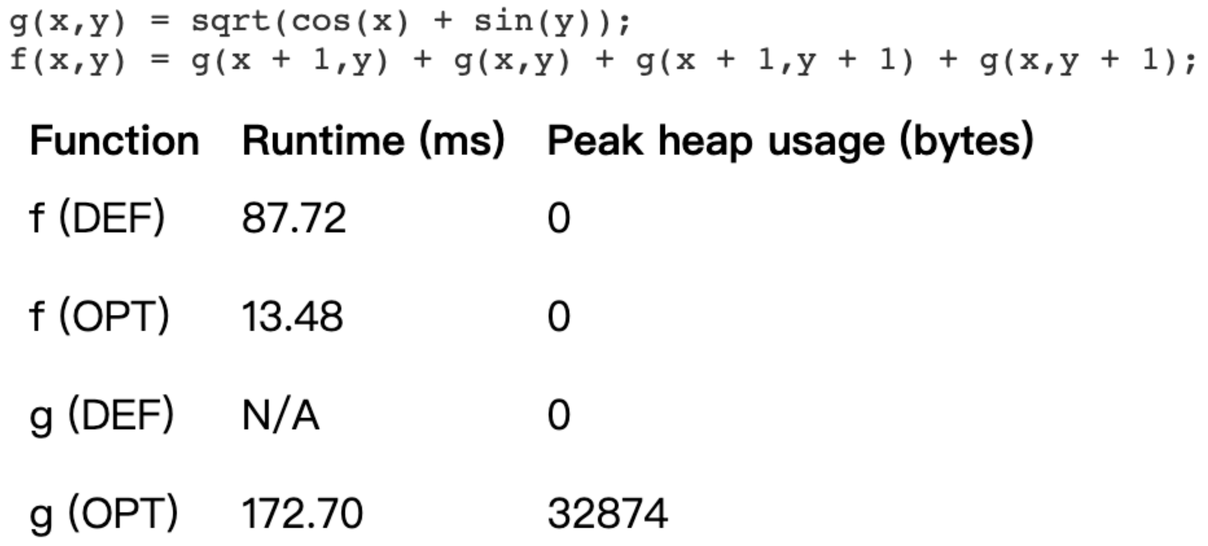


Schedule Evaluation

- **Search Algorithm for Schedules:** Once loop sizes have been inferred, we have enough information to determine important execution features of the schedule, such as how much memory it will allocate and how many operations it will perform. The *cost* of the schedule is then a weighted sum of these data points.



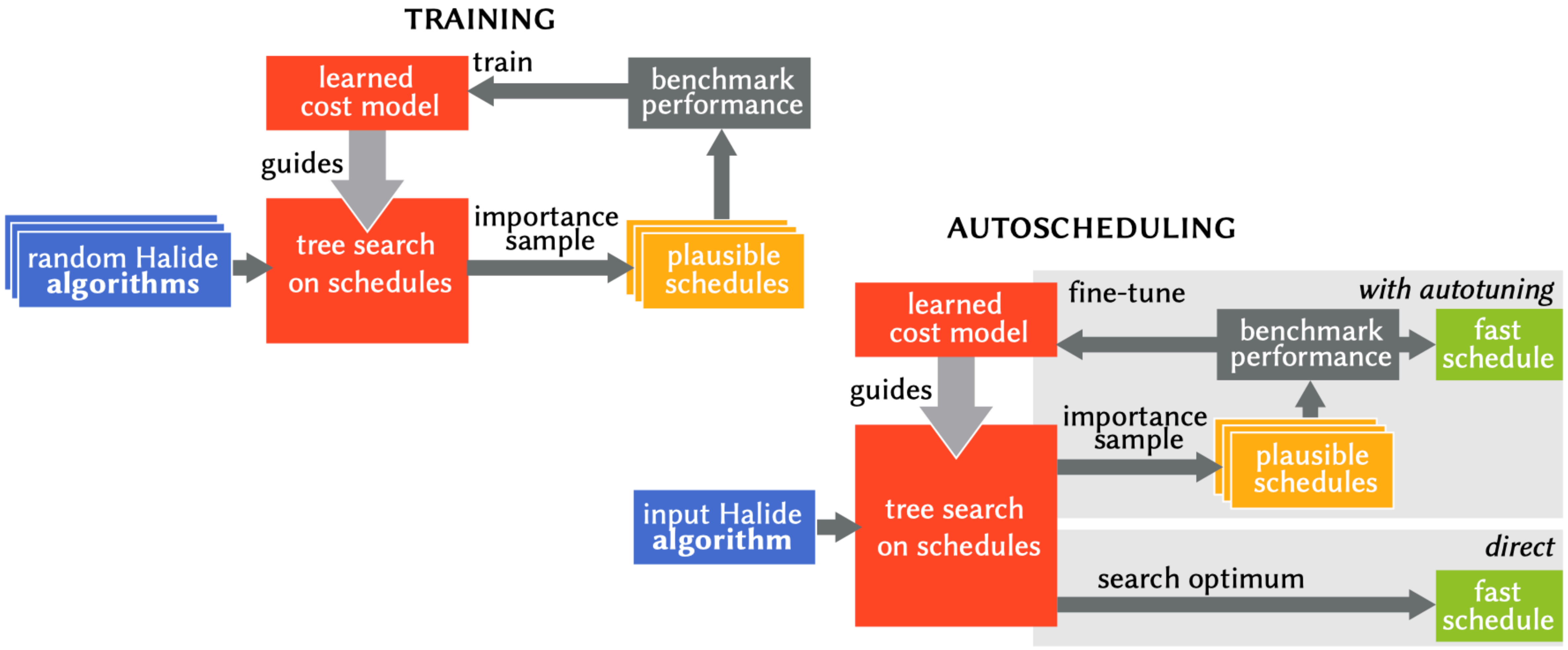
Group	Weight
mem	0.1
loads	0.5
stores	0.5
arith ops	1.0
math ops	10.0



```
g(x,y) = sqrt(cos(x) + sin(y));
f(x,y) = g(x + 1,y) + g(x,y) + g(x + 1,y + 1) + g(x,y + 1);
```

Function	Runtime (ms)	Peak heap usage (bytes)
f (DEF)	87.72	0
f (OPT)	13.48	0
g (DEF)	N/A	0
g (OPT)	172.70	32874

Schedule Evaluation



Inference

1. Adams, Andrew, et al. "Learning to optimize halide with tree search and random programs." *ACM Transactions on Graphics (TOG)* 38.4 (2019): 1-12.
2. https://www.cs.cornell.edu/courses/cs6120/2019fa/blog/halide-autoscheduler/
3. https://halide-lang.org/papers/autoscheduler2019.html
4. Halide代码优化思想简述 https://www.modb.pro/db/326253
5. 图像、神经网络优化利器:了解Halide https://oldpan.me/archives/learn-a-little-halide

BUILDING A BETTER CONNECTED WORLD
THANK YOU