

👁️ 点击小眼睛开启蜘蛛网特效

图像、神经网络优化利器:了解Halide

✍️ Oldpan 📅 2019年4月17日 💬 0条评论 👁️ 11,010次阅读 👍 14人点赞



前言

Halide是用C++作为宿主语言的一个图像处理相关的DSL(Domain Specified Language)语言, 全称领域专用语言。主要的作用为在软硬层面上(与算法本身的设计无关)实现对算法的底层加速, 我们有必要对其有一定的了解。因为不论是**传统的图像处理方法**亦或是**深度学习应用**都使用到了halide的思想。

其中, 在OpenCV(传统图像处理库)中部分算法使用了Halide后端, 而TVM(神经网络编译器)也是用了Halide的思想去优化神经网络算子。

Spoiler: Simpler, Faster, Scalable

Reference: 300 lines C++

Adobe: 1500 lines

3 months of work

10x faster (vs. reference)

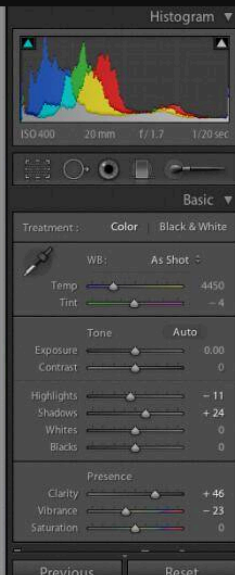
Halide: 60 lines

1 intern-day

20x faster (vs. reference)

2x faster (vs. Adobe)

GPU: 70x faster (vs. reference)



那么Halide到底是干嘛用的，看上面那张图，同样的一个算法处理(局部拉普拉斯变换)，使用直接的C++语言写出来算法速度很慢，Adobe公司使用3个月对这个算法进行了优化(手工优化)使这个算法的速度快了10倍，但是如果你使用了Halide，只需要几行代码，就可以使这个算法比之前普通直接的算法快上**20倍**。

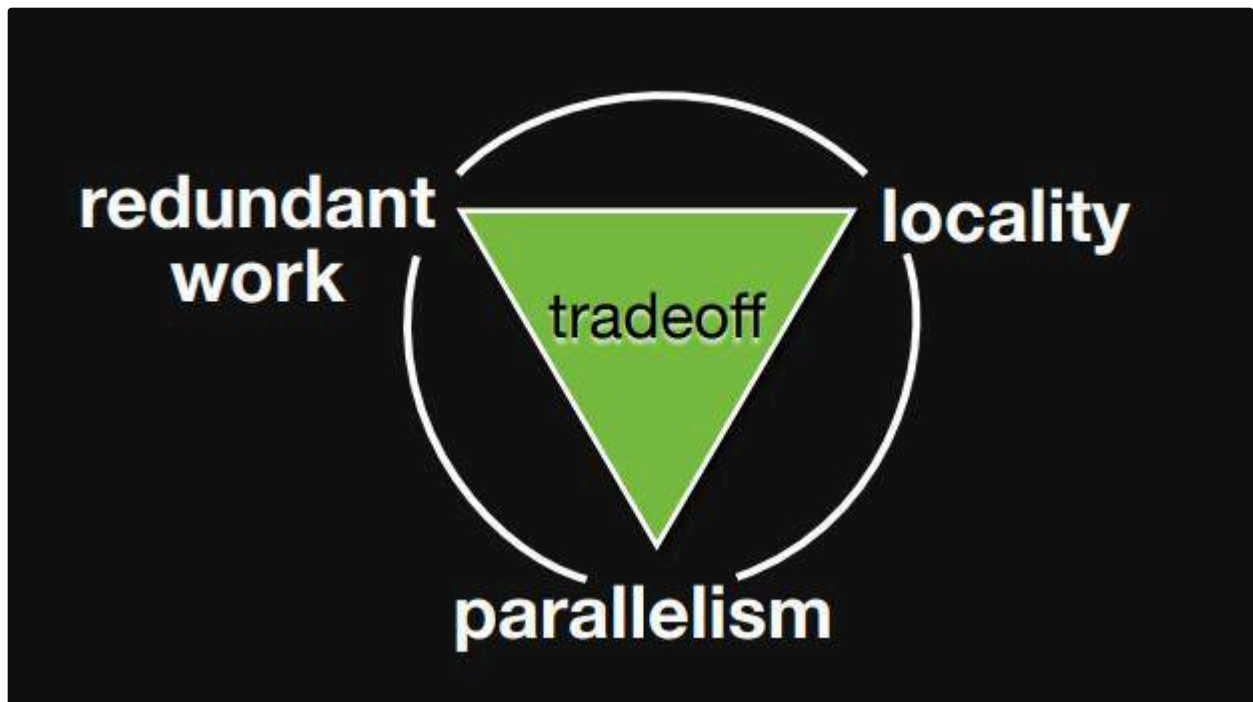
一句话来说，Halide大大节省了我们手动优化底层算法的时间，让我们只需要关心算法的设计。

Halide为什么可以优化算法

Halide的特点是其图像算法的计算的实现 (Function和Expression) 和这些计算在计算硬件单元上的调度 (Schedule) 是分离的，其调度以Function为单位。最终将整个图像算法转换为高效率的多层for循环，for循环的分部数据范围划分和数据加载都是由Halide来完成的，而且可以实现数据的加载和算法计算的Overlay，掩盖数据加载导致的延迟。Halide的Schedule可以由程序员来指定一些策略，指定硬件的buffer大小，缓冲线的相关设置，这样可以根据不同的计算硬件的特性来实现高效率的计算单元的调度，而图像算法的计算实现却不需要修改。

上面这部分截取于知乎：<https://www.zhihu.com/question/294625837/answer/496218375>
(<https://www.zhihu.com/question/294625837/answer/496218375>)

决定算法在某个硬件平台上执行时性能的“三角力量”如下。



其中，算法本身的设计是一方面，一个好的算法往往效率会高很多。而另外一个方面就是算法中计算顺序的组织，而Halide可以改变的就是我们算法在某个硬件平台上的计算顺序：



其中Halide可以在硬件平台上为算法实现**并行**和良好的缓存一致性 (<https://baike.baidu.com/item/%E7%BC%93%E5%AD%98%E4%B8%80%E8%87%B4%E6%80%A7/15814005?fr=aladdin>):

举个例子

我们以Halide中的经典模糊化(blurred)图像的例子来演示一下(以下代码也可以在自己的电脑上测试观察结果)，这里用**OpenCV**来对图像进行操作进行演示：

首先我们设计一个可以对图像进行模糊的操作函数：

```

// in为输入原始图像 blur为输出模糊后的图像
void box_filter_3x3(const Mat &in, Mat &blur)
{
    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++)
            blur.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);
}

```

对图像模糊操作很简单，我们首先在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，这里就不进行详细描述了。

我们准备一张(1920,1080)的图像，对其进行100次上述操作，并记录时间，发现 `Time used:4521.72 ms`。

然后我们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：

```

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++)
        blur.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);
}

```

同样，我们执行100次并记录时间：发现 `Time used:3992.35 ms`，可以发现下面的模糊操作执行的速度比上面的快一些。当然大家可能会想，这也没快多少啊。当然这只是一副示例图像，如果这张图像的长宽差距比较大(例如1:10)、亦或是我们要某一个时刻处理几万次这样的操作，一旦量级起来，那么这两者的差距就不是一点半点了。

硬件层面的原理

为什么会这样呢，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。

我们可以看到下面是Adobe工程师对上述的算法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。

SIMD

tile loops

inner loops inside tiles

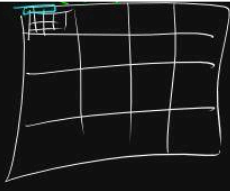
Hand-optimized C++

9.9 → 0.9 ms/megapixel

```
void box_filter_3x3(const Image &in, Image &blur) {
    __m128i one_third = _mm_set1_epi16(33333);
    #pragma omp parallel for
    for (int yTile = 0; yTile < in.height(); yTile += 32) {
        __m128i a, b, c, sum, avg;
        __m128i blurPtr[(256/8)*(32+2)]; // allocate tile blur array
        for (int xTile = 0; xTile < in.width(); xTile += 256) {
            __m128i *blurPtr = blurPtr + xTile;
            for (int y = 0; y < 32; y++) {
                const uint16_t *inPtr = &(in[yTile+y][xTile]);
                for (int x = 0; x < 256; x += 8) {
                    a = _mm_loadu_si128((__m128i*)(inPtr+1));
                    b = _mm_loadu_si128((__m128i*)(inPtr+1));
                    c = _mm_load_si128((__m128i*)(inPtr));
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_store_si128(blurPtr++, avg);
                    inPtr += 8;
                }
                blurPtr = blurPtr;
                for (int y = 0; y < 32; y++) {
                    __m128i *outPtr = (__m128i *)&(blur[yTile+y][xTile]);
                    for (int x = 0; x < 256; x += 8) {
                        a = _mm_load_si128(blurPtr+(2*256)/8);
                        b = _mm_load_si128(blurPtr+256/8);
                        c = _mm_load_si128(blurPtr++);
                        sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                        avg = _mm_mulhi_epi16(sum, one_third);
                        _mm_store_si128(outPtr++, avg);
                    }
                }
            }
        }
    }
}
```

expanding tile size

Tiles



Tiled, fused
Vectorized
Multithreaded
Redundant computation
Near roof-line optimum

官方示例

Halide作为一个DSL，我们很容易就可以使用它，这里我们将其源码 (<https://github.com/halide/Halide>) 下载下来并进行编译。完成之后我们就可以使用它了(这里省略编译步骤，可自行在官网查阅)：

首先我们引用Halide头文件以及其他的文件。


```
#include "Halide.h"
#include <stdio.h>
#include <algorithm>

using namespace Halide;
```

初次使用Halide之前，首先需要知道halide中的一些语法：

Syntax: Main types/keywords

functional language

Func : pure functions over an integer domain

Var : pure abstract variables for domain of Funcs

Expr: algebraic expressions of Funcs and Var
including standard operators and functions (+, -, &, /, **, sqrt, sin, cos...)

Image: arrays used as inputs and outputs

both Halide constructs & under the hood C++ class

然后我们利用Halide定义两个变量，这两个变量单独使用时没有任何意义，同时我们用字符串 `x` 和 `y` 为两个变量起了名字：

```
Var x("x"), y("y");
```

然后利用Func 定义一个待执行的function，并起名为 `gradient` 。

```
Func gradient("gradient");
```

这时我们定义function中每个点的执行逻辑，对于 `(x,y)` 这个点执行的逻辑为 `x + y` 。其中x和y都是Var，而 `x + y` 这个操作在赋予给gradient的时候会自动转化为Expr类型，这里可以理解为将 `x + y` 这个代数表达式的逻辑赋予了 `gradient` ，最后，通过 `realize` 函数来执行整个逻辑：

```
gradient(x, y) = x + y;
// realize 即为实现这个操作 到了这一步才会对上述的操作进行编译并执行
Buffer<int> output = gradient.realize(4, 4);
```

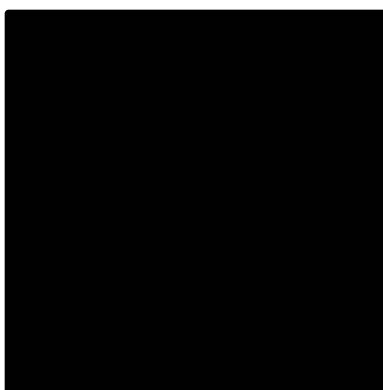
这个逻辑我们用C++来表示即为：

```
for (int y = 0; y < 4; y++) {  
    for (int x = 0; x < 4; x++) {  
        printf("Evaluating at x = %d, y = %d: %d\n", x, y, x + y);  
    }  
}
```

而上述实现的Halide伪代码为:

```
produce gradient:  
  for y:  
    for x:  
      gradient(...) = ...
```

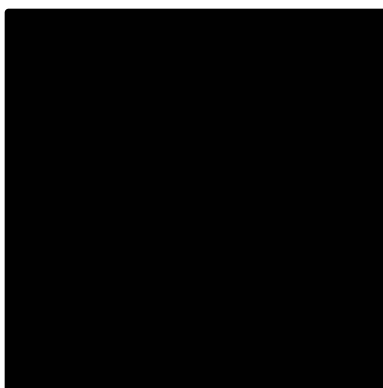
Halide默认的计算顺序是行优先的, 也就是x代表每一行的元素位置, y代表每一列的元素位置:



如果我们将其中y和x的计算顺序换一下:

```
// 将y的顺序提到x之前  
gradient.reorder(y, x);
```

最终的计算过程就为列优先:



相应的伪代码为：

```
produce gradient_col_major:
  for x:
    for y:
      gradient_col_major(...) = ...
```

拆分 Split

我们可以对每个维度进行拆分，假如我们依然是行优先计算，但是我们对x轴进行拆分，将其拆成一个外循环一个里循环，y轴不进行变动：

```
Var x_outer, x_inner;
gradient.split(x, x_outer, x_inner, 2);
```

这时对应的C++代码实现为：

```
for (int y = 0; y < 4; y++) {
  for (int x_outer = 0; x_outer < 2; x_outer++) {
    for (int x_inner = 0; x_inner < 2; x_inner++) {
      int x = x_outer * 2 + x_inner;
      printf("Evaluating at x = %d, y = %d: %d\n", x, y, x + y);
    }
  }
}
```

融合 fuse

或者我们不进行拆分，对x和y两个轴进行融合：

```
Var fused;
gradient.fuse(x, y, fused);
```

此时对应的C++实现代码为：


```

for (int fused = 0; fused < 4*4; fused++) {
    int y = fused / 4;
    int x = fused % 4;
    printf("Evaluating at x = %d, y = %d: %d\n", x, y, x + y);
}

```

但是要知道，上述拆分和融合操作只是对Halide所能进行的操作进行一下演示而是，这种操作方式并没有实际用处，也就是说实际中的**计算顺序并没有改变**。

平铺 tile

这一步中就要进入Halide中比较重要的部分了，这一步中我们将x和y轴以4为因子间隔进行划分，并且重新对计算的路径进行重排序：

```

Var x_outer, x_inner, y_outer, y_inner;
gradient.split(x, x_outer, x_inner, 4);
gradient.split(y, y_outer, y_inner, 4);
gradient.reorder(x_inner, y_inner, x_outer, y_outer);

// 上面的步骤其实可以简化成
gradient.tile(x, y, x_outer, y_outer, x_inner, y_inner, 4, 4);

```

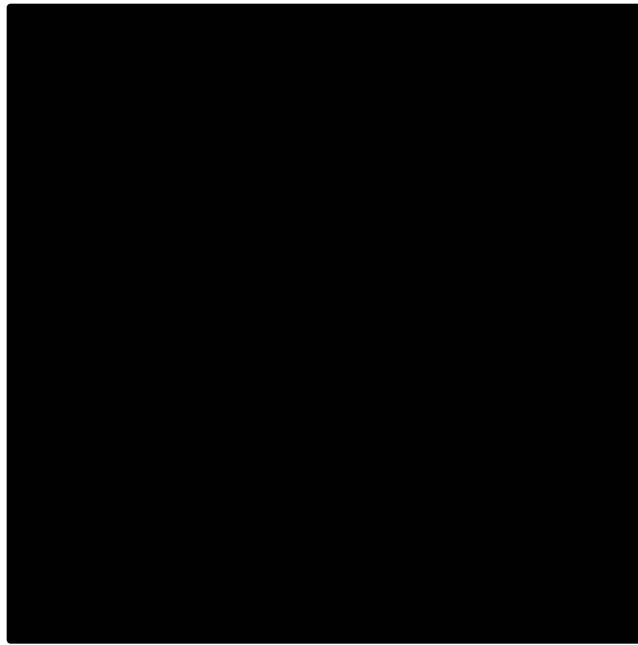
对应的C++计算代码为：

```

for (int y_outer = 0; y_outer < 2; y_outer++) {
    for (int x_outer = 0; x_outer < 2; x_outer++) {
        for (int y_inner = 0; y_inner < 4; y_inner++) {
            for (int x_inner = 0; x_inner < 4; x_inner++) {
                int x = x_outer * 4 + x_inner;
                int y = y_outer * 4 + y_inner;
                printf("Evaluating at x = %d, y = %d: %d\n", x, y, x + y);
            }
        }
    }
}

```

可视化一下就是这个样子(注意这里的示例大小为(8,8))：



这种平铺的好处是可以**充分利用相邻的像素**，例如在模糊中，我们会使用重叠的输入数据(也就是存在一个元素使用两次的情况)，如果采用这种计算方式，可以大大加快计算性能。

向量化 vector

向量化即我们使用cpu中的SIMD技术，一次性计算多个数据，充分利用硬件的特点，例如在x86中我们可以利用SSE技术来实现这个功能。

在Halide中，我们首先将x轴的循环嵌套按照，内侧循环因子4的方式，拆分为两个(也就是内侧循环x执行四次，外侧根据总数进行计算，下例是 $2*4=8$)，然后将内侧的x循环转化为向量的形式：

```
Var x_outer, x_inner;
gradient.split(x, x_outer, x_inner, 4);
gradient.vectorize(x_inner);
```

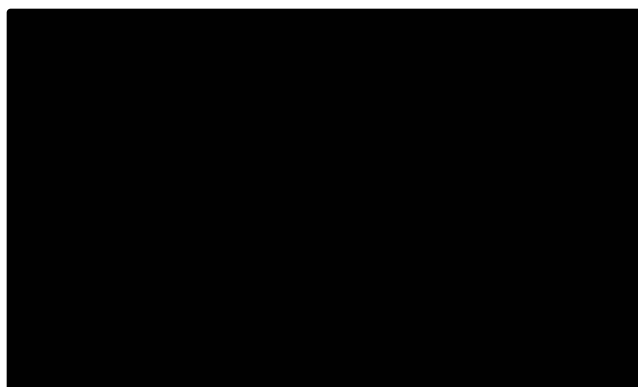
用C++来表示即为：

```

for (int y = 0; y < 4; y++) {
    for (int x_outer = 0; x_outer < 2; x_outer++) {
        // The loop over x_inner has gone away, and has been
        // replaced by a vectorized version of the
        // expression. On x86 processors, Halide generates SSE
        // for all of this.
        int x_vec[] = {x_outer * 4 + 0,
                        x_outer * 4 + 1,
                        x_outer * 4 + 2,
                        x_outer * 4 + 3};
        int val[] = {x_vec[0] + y,
                     x_vec[1] + y,
                     x_vec[2] + y,
                     x_vec[3] + y};
        printf("Evaluating at <%d, %d, %d, %d>, <%d, %d, %d, %d>:"
              " <%d, %d, %d, %d>\n",
              x_vec[0], x_vec[1], x_vec[2], x_vec[3],
              y, y, y, y,
              val[0], val[1], val[2], val[3]);
    }
}

```

可视化后就比较明显了，外部x每一行执行两次，内侧x变为向量的形式，一个指令集就可以执行完成：



展开 unrolling

如果在图像中多个像素同时共享有重叠的数据，这个时候我们就可以将循环展开，从而使那些可以共享使用的数据只计算一次亦或是只加载一次。

在下面中我们将x轴拆分为内侧和外侧，因为每次内侧的数值增长都是从0到1，如果我们将内测循环的x轴展开，就不需要每次循环到这里再读取内测循环的x的值了：

```

Var x_outer, x_inner;
gradient.split(x, x_outer, x_inner, 2);
gradient.unroll(x_inner);

```

相应的C++代码为:

```

printf("Equivalent C:\n");
for (int y = 0; y < 4; y++) {
    for (int x_outer = 0; x_outer < 2; x_outer++) {
        // Instead of a for loop over x_inner, we get two
        // copies of the innermost statement.
        {
            int x_inner = 0;
            int x = x_outer * 2 + x_inner;
            printf("Evaluating at x = %d, y = %d: %d\n", x, y, x + y);
        }
        {
            int x_inner = 1;
            int x = x_outer * 2 + x_inner;
            printf("Evaluating at x = %d, y = %d: %d\n", x, y, x + y);
        }
    }
}

```

融合、平铺、并行 Fusing, tiling, and parallelizing

这一步中，我们将融合、平铺和并行操作都融合到一起，来对一个8×8的图像进行操作。首先，我们将x轴和y轴都按照4因子进行平铺操作。随后我们将外侧的y和外侧的x轴循环进行融合(2+2=4)，再将这个融合后的操作进行并行操作，也就是同时执行这四个(2+2=4)操作：

```

Var x_outer, y_outer, x_inner, y_inner, tile_index;
gradient.tile(x, y, x_outer, y_outer, x_inner, y_inner, 4, 4);
gradient.fuse(x_outer, y_outer, tile_index);
gradient.parallel(tile_index);

```

相应的C++代码为:

```
// This outermost loop should be a parallel for loop, but that's hard in C.
for (int tile_index = 0; tile_index < 4; tile_index++) {
    int y_outer = tile_index / 2;
    int x_outer = tile_index % 2;
    for (int y_inner = 0; y_inner < 4; y_inner++) {
        for (int x_inner = 0; x_inner < 4; x_inner++) {
            int y = y_outer * 4 + y_inner;
            int x = x_outer * 4 + x_inner;
            printf("Evaluating at x = %d, y = %d: %d\n", x, y, x + y);
        }
    }
}
```

可视化后的结果，可以看到8×8中左上、左下、右上、右下四个区域是几乎同时进行的(tile_index)，而每个区域和之前tile那一节的计算方式是一样的，只不过这次换成了并行计算：



整合

这次来点大点的图像，我们输入的图像大小为 `350 x 250`，对其进行最优化的操作：

首先我们将其按照 `64 x 64` 的因子进行平铺，其次融合y轴和x轴外侧的循环操作数，最后对其进行并行操作

（这里注意下，我们可以看到350或者250并不能被64整除，这个不用担心，Halide会自动处理多余或者不够的部分）。

```

Var x_outer, y_outer, x_inner, y_inner, tile_index;
gradient_fast
    .tile(x, y, x_outer, y_outer, x_inner, y_inner, 64, 64)
    .fuse(x_outer, y_outer, tile_index)
    .parallel(tile_index);
// 可以这样连续使用.写, 因为对象函数返回的是对象本身的引用

```

这样还不够, 上面我们已经将整个图像平铺为6*4个部分, 而这一步中对每个平铺后的部分再进行一次平铺操作, 这次将每个小块按照4*2的形式平铺为, 其中y_inner_outer分成两个(每个为y_pairs), x_inner_outer分成四个(每个为x_vectors), 然后将每个x_vectors并行化, 将y_pairs展开。

```

Var x_inner_outer, y_inner_outer, x_vectors, y_pairs;
gradient_fast
    .tile(x_inner, y_inner, x_inner_outer, y_inner_outer, x_vectors, y_pairs, 4, 2)
    .vectorize(x_vectors)
    .unroll(y_pairs);

```

以下可视化的结果为:



对应的c++展示代码为:

```

for (int tile_index = 0; tile_index < 6 * 4; tile_index++) {
    int y_outer = tile_index / 4;
    int x_outer = tile_index % 4;
    for (int y_inner_outer = 0; y_inner_outer < 64/2; y_inner_outer++) {
        for (int x_inner_outer = 0; x_inner_outer < 64/4; x_inner_outer++) {
            // We're vectorized across x
            int x = std::min(x_outer * 64, 350-64) + x_inner_outer*4;
            int x_vec[4] = {x + 0,
                           x + 1,
                           x + 2,
                           x + 3};

            // And we unrolled across y
            int y_base = std::min(y_outer * 64, 250-64) + y_inner_outer*2;
            {
                // y_pairs = 0
                int y = y_base + 0;
                int y_vec[4] = {y, y, y, y};
                int val[4] = {x_vec[0] + y_vec[0],
                             x_vec[1] + y_vec[1],
                             x_vec[2] + y_vec[2],
                             x_vec[3] + y_vec[3]};

                // Check the result.
                for (int i = 0; i < 4; i++) {
                    if (result(x_vec[i], y_vec[i]) != val[i]) {
                        printf("There was an error at %d %d!\n",
                               x_vec[i], y_vec[i]);
                        return -1;
                    }
                }
            }
        }
    }
    {
        // y_pairs = 1
        int y = y_base + 1;
        int y_vec[4] = {y, y, y, y};
        int val[4] = {x_vec[0] + y_vec[0],
                     x_vec[1] + y_vec[1],
                     x_vec[2] + y_vec[2],
                     x_vec[3] + y_vec[3]};

        // Check the result.
        for (int i = 0; i < 4; i++) {
            if (result(x_vec[i], y_vec[i]) != val[i]) {
                printf("There was an error at %d %d!\n",
                       x_vec[i], y_vec[i]);
                return -1;
            }
        }
    }
}

```



```

    }
  }
}
}
}
}

```

到这里Halide中的基本操作就介绍完毕了。

还有一点

哦，对了，如果用Halide来写文章一开头描述的模糊(blur)算法的话，会是这个样子：

```

Func blur_3x3(Func input) {
  Func blur_x, blur_y;
  Var x, y, xi, yi;

  // The algorithm - no storage or order
  blur_x(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;
  blur_y(x, y) = (blur_x(x, y-1) + blur_x(x, y) + blur_x(x, y+1))/3;

  // The schedule - defines order, locality; implies storage
  blur_y.tile(x, y, xi, yi, 256, 32)
    .vectorize(xi, 8).parallel(y);
  blur_x.compute_at(blur_y, x).vectorize(x, 8);

  return blur_y;
}

```

这段著名的代码同时就在官方的主页 (<https://halide-lang.org/>)上挂着，算是一个比较好的示例。

Halide的特点

Halide这个底层优化库有几个比较亮眼的特点：

Explicit programmer control

The compiler does exactly what you say.
 Schedules cannot influence correctness.
 Exploration is fast and easy.

明确的程序控制，也就是说，我们如何按照这个计算的顺序(与算法本身无关)是确定的，一旦我们已经设定好就不会再改变。

Stochastic search (autotuning)

Pick your favorite high-dimensional search.

而自动搜索则是每个具有搜索空间的优化器都可以使用的，因为每次进行优化操作的时候，优化的因子都是不确定的，对于不同的硬件来说，不同的配置可能导致的执行速度也不一样。因此自动随机搜索空间因子是有必要的。

元编程

Halide的思想与元编程有着密切的关系，不仅是其设计思路或者是其执行思路，都遵循了元编程的思想，也就是代码在编译之前并没有明确的执行逻辑，只有编译过后，才会形成执行逻辑。

Metaprogramming

Create C++ objects that describe a Halide program

Essentially algebraic trees (Abstract Syntax Tree, AST)

Once the representation is constructed, call `.realize()` to compile and execute

This calls the C++ Halide compiler, creates binary, executes it

Metaprogramming

Makes it easy to embed in an existing language and codebase

Avoids the need to parse

其他相关

halide既然作为与算法无关的底层优化器，与当今大伙的深度学习的应用肯定也是非常多的。**OpenCV**库就使用了halide去优化底层的神经网络算子，相应的benchmark结论在这里 (<https://github.com/opencv/opencv/wiki/DNN-Efficiency>)，但是我们发现使用了halide的神经网络

运行的速度竟然不如普通的C++实现版。首先表明这个原因与halide本身的设计无关，但是与halide优化和神经网络算子的兼容性有关，如果想要利用halide真正的实现加速还是需要等待一段时间了。

Model	DNN, C++	DNN, Halide	Intel-Caffe, MKLDNN	TensorFlow	Torch w. MKL
AlexNet	14.52	22.31	11.95		
GoogLeNet	17.37	32.43	9.43		
ResNet-50	40.01	76.13	22.75		
SqueezeNet v1.1	4.68	6.61	3.05		
Inception-5h	19.30	35.27		14.6	
ENet @ 512x256	65.93	42.16			226
OpenFace (nn4.small2)	4.20	8.14			25.44
MobileNet-SSD @ 300x300 20 classes, Caffe	22.71	54.36	27.79		
MobileNet-SSD @ 300x300 90 classes, TensorFlow	25.15	60.95		35.86	

相关提问：

<https://stackoverflow.com/questions/47202895/why-is-opencv-dnn-slower-if-i-use-halide> (<https://stackoverflow.com/questions/47202895/why-is-opencv-dnn-slower-if-i-use-halide>)

后记

本文只是简单介绍了Halide的基本知识，对于想要深入理解Halide的童鞋可以看官方的教程或者阅读源码，不论我们是设计算法的算法工程师亦或是在相关硬件平台上实现移植功能的底层工程师，Halide的思想都是值得我们去借鉴和回味的。

另外提一下，Halide的运行有两种方式，一种是JIT的模式，另一种是AOT的模式。JIT模式使用起来比较方便，可以直接将算法和Halide的代码生成generator封装成一个类，在程序的其他部分调用这个类即可。在嵌入式环境和交叉编译环境下一般使用AOT模式，此时需要调用com

piler函数将算法代码和Halide的代码生成generator编译位目标机器的代码，生成一个.o目标文件和.h头文件。然后在独立的目标机器的应用的工程的源代码中通过头文件调用算法实现的计算函数，并在build的时候链接上.o文件，这样就得到一个可以在目标机器上运行的用Halide实现算法的程序了。一般DSP上都是这种方式来做的。

Halide的利用范围很广，我之所以想要深入了解Halide是因为使用了TVM库，TVM借助了Halide的思想去实现神经网络算子的优化并且取得了不错的效果。

参考资料：

上述所采用的图像部分来源于此：

http://stellar.mit.edu/S/course/6/sp15/6.815/courseMaterial/topics/topic2/lectureNotes/14_Halide_print/14_Halide_print.pdf (http://stellar.mit.edu/S/course/6/sp15/6.815/courseMaterial/topics/topic2/lectureNotes/14_Halide_print/14_Halide_print.pdf)

Halide的官网：

<https://halide-lang.org/> (<https://halide-lang.org/>)



机器学习 (<HTTPS://OLDPAN.ME/LABELS/%E6%9C%BA%E5%99%A8%E5%AD%A6%E4%B9%A0>)

深度学习 (<HTTPS://OLDPAN.ME/LABELS/%E6%B7%B1%E5%BA%A6%E5%AD%A6%E4%B9%A0>)

深度学习库 (<HTTPS://OLDPAN.ME/LABELS/%E6%B7%B1%E5%BA%A6%E5%AD%A6%E4%B9%A0%E5%BA%93>)

编译器 (<HTTPS://OLDPAN.ME/LABELS/%E7%BC%96%E8%AF%91%E5%99%A8>)

本篇文章采用 署名-非商业性使用-禁止演绎 4.0 国际 (<https://creativecommons.org/licenses/by-nc-nd/4.0/>) 进行许可

转载请务必注明来源: <https://oldpan.me/archives/learn-a-little-halide>
(<https://oldpan.me/archives/learn-a-little-halide>)

关注Oldpan博客微信公众号，你最需要的及时推送给你。