

# Project4: BMP Image Processing With C++

## Contents

1. General .....	1.
2. Project Overview .....	1.
3. Ground Class .....	3.
3.1. Memory Management .....	3.
3.2. NHWC or NCHW? .....	4.
3.3. 类型与静态编译技术 .....	5.
3.4. ROI .....	7.
3.5. 基于方法重写的通道视图 .....	7.
4. 卷积计算的优化 - Im2Col .....	8.
4.1. 对输入图进行 Im2Col .....	11.
4.2. 对卷积核进行 Im2Col .....	12.
4.3. 优化点之 Pack 策略 .....	12.
4.4. 优化点之带 Pack 的矩阵乘法 .....	15.
5. 总结 .....	17.

## §1. General

作为本学期第一个 C++ 的项目，我是带着些许对 C++ 的敬畏来完成的。因为 C++ 的语言特性总是日渐复杂，尤其是涉及到大型库的编写，总离不开类的抽象与封装。如何利用 C++ 的特性来简化代码的复杂度，成为了我在这个项目中最重要目标。

## §2. Project Overview

本项目提供了一个基于 C++ 编写、较为综合、对用户使用友好的图像处理库，项目结构如下：

```
project4/ -- root
├── img/
│   └── sustech.jpg -- example image(download with "make assets")
├── CMakeLists.cpp -- CMake config
└── mcv.decl.h      -- lib header
```

```
└─ mcv.cpp      -- lib impl file
└─ example.cpp  -- demo file
```

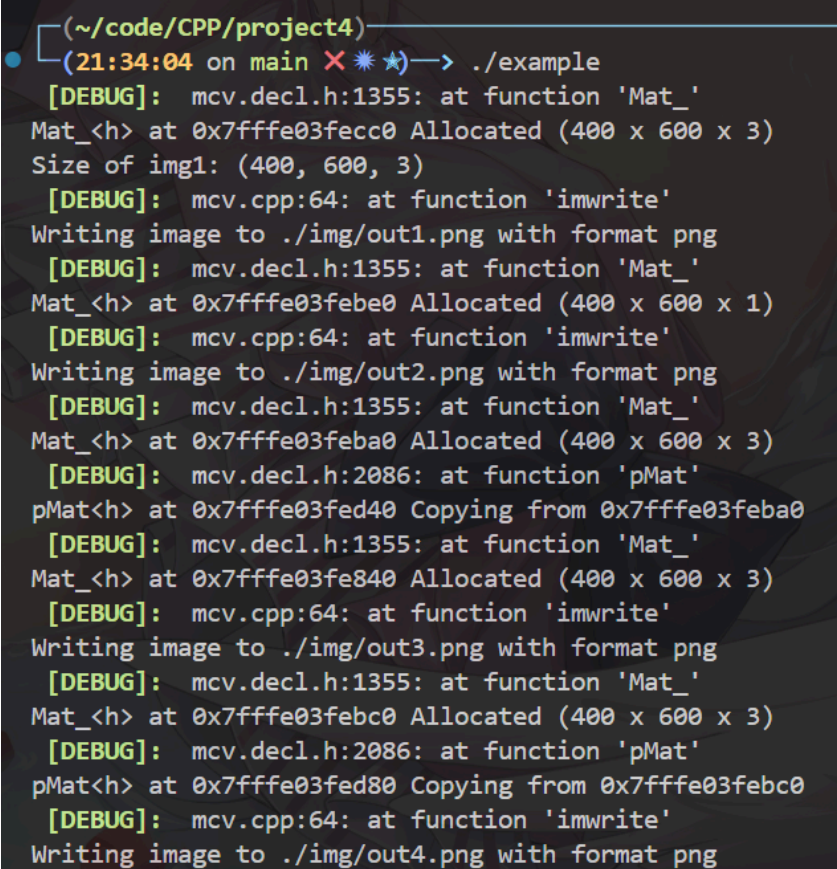
使用如下命令构建：

```
cmake -S . -B build
cmake --build build
```

支持的 make 命令如下：

```
make debug # build with debug out
make omp   # build with OpenMP
make bench # run with timing
make assets # download example image
```

如果使用 make debug，则会开启 -DMCV\_DBG 编译并输出调试信息，输出信息如下：



```
(~/code/CPP/project4)
(21:34:04 on main ✖️ ⚡) → ./example
[DEBUG]: mcv.decl.h:1355: at function 'Mat_'
Mat_<h> at 0x7fffe03fecc0 Allocated (400 x 600 x 3)
Size of img1: (400, 600, 3)
[DEBUG]: mcv.cpp:64: at function 'imwrite'
Writing image to ./img/out1.png with format png
[DEBUG]: mcv.decl.h:1355: at function 'Mat_'
Mat_<h> at 0x7fffe03febe0 Allocated (400 x 600 x 1)
[DEBUG]: mcv.cpp:64: at function 'imwrite'
Writing image to ./img/out2.png with format png
[DEBUG]: mcv.decl.h:1355: at function 'Mat_'
Mat_<h> at 0x7fffe03feba0 Allocated (400 x 600 x 3)
[DEBUG]: mcv.decl.h:2086: at function 'pMat'
pMat<h> at 0x7fffe03fed40 Copying from 0x7fffe03feba0
[DEBUG]: mcv.decl.h:1355: at function 'Mat_'
Mat_<h> at 0x7fffe03fe840 Allocated (400 x 600 x 3)
[DEBUG]: mcv.cpp:64: at function 'imwrite'
Writing image to ./img/out3.png with format png
[DEBUG]: mcv.decl.h:1355: at function 'Mat_'
Mat_<h> at 0x7fffe03febc0 Allocated (400 x 600 x 3)
[DEBUG]: mcv.decl.h:2086: at function 'pMat'
pMat<h> at 0x7fffe03fed80 Copying from 0x7fffe03febc0
[DEBUG]: mcv.cpp:64: at function 'imwrite'
Writing image to ./img/out4.png with format png
```

如果使用 make omp，则会开启 -fopenmp 编译并使用 OpenMP 进行加速。

如果使用 make bench，则会开启 -DMCV\_BENCH 编译并输出性能测试信息，输出信息如下：

```
(~/code/CPP/project4)
(21:35:45 on main ✖ ⚡ ☆) → ./example
Size of img1: (400, 600, 3)
===== Function 'compute()' Timing Results =====
Wall Time      : 0.000077456 seconds
CPU Time       : 0.000076000 seconds
User Time      : 0.000000000 seconds
System Time    : 0.000000000 seconds
Total Process Time : 0.000000000 seconds
Real System Time  : 0.000076000 seconds
=====
Cycles: 241626
Estimated CPU Frequency: 3.293643867 GHz
=====
===== Function 'cvtColor()' Timing Results =====
Wall Time      : 0.001215995 seconds
CPU Time       : 0.001215000 seconds
User Time      : 0.000000000 seconds
System Time    : 0.000000000 seconds
Total Process Time : 0.000000000 seconds
Real System Time  : 0.001214000 seconds
=====
Cycles: 3990294
Estimated CPU Frequency: 3.293663680 GHz
```

这会在耗时函数前后插入计时器，并输出函数的各类时间统计信息；同时也会粗略估计 CPU 的 cycles 数。

如果使用 `make assets`，则会下载一张示例图片到 `img/` 目录下用于 demo。

一些图像处理的基本操作以及用法可以在 `example.cpp` 中找到，这里有很多功能都是复刻 `project3`，在此不一一赘述。另外值得一提的是，除了接受 24 位 BMP 图像，我在引用了单头文件开源库 `stb_image` 的基础上，增加了对 JPEG 和 PNG 格式的支持。

## §3. Ground Class

众所周知 C++ 换汤不换药的就是矩阵类的设计，经过上一次 project 我们已然知道图像的存储和处理实际上就是对矩阵的操作。如果希望提升代码质量，一个优雅而高效的矩阵类是必不可少的。然而和上次 project 不同的是，我们的使用的语言工具从 C 转变为 C++，那么如果依然使用 C 的方式来设计矩阵类，势必会导致代码的复杂度大幅提升。因此我们需要“因地制宜”，使用 C++ 的特性来简化维护和使用的难度。

### §3.1. Memory Management

C/C++ 的内存管理从来都是一个令人头疼的问题，尤其是当我们需要使用动态内存分配时。为了避免内存泄漏和野指针的问题，我们需要在类的设计中考虑到使用何种内存管理方式。如果依然像 C 一样维护一个指针数组，那么我们需要在类的析构函数中手动释放内存，这无疑会增加代码的复杂度和出错的可能性。同样这也不适合图像存储这样一个可能需要频繁复制与移动的场景。这时我们联想到了 C++ 的智能指针，使用 `std::shared_ptr` 来管理内存是一个不错的选择。它既避免了手动释放内存的麻烦，又可以安全的在不同内存持有者之间共享内存。忽略

一些可能的性能损失，直接使用智能指针的确优于手动管理内存。因此一个简单的矩阵类可以设计成如下的形式：

```
class Mat_
{
protected:
    // parameters
    size_t rows_;
    size_t cols_;
    size_t stride_;

    // pointer
    T* ptr;

    // container(shared)
    std::shared_ptr<T> data;
    // ...
}
```

不过这个 `stride_` 参数是做什么的？目前只需要知道它是一个为列宽做参照的步长参数，详细作用我们之后再解释。现在我们可以确定这个矩阵类需要分配内存的场景只有一种，就是根据传入的大小在构造函数中分配内存。而其他的场景都可以直接或间接复制这个类的实例来实现。比如一个复制构造函数可以简单粗暴地写成这样：

```
Mat_(const Mat_ &other) : rows_(other.rows_), cols_(other.cols_),
    stride_(other.stride_), data(other.data), ptr(other.ptr) {
}
```

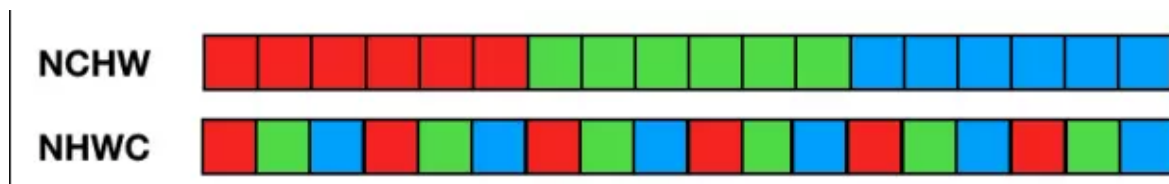
而复制赋值函数由于已经确定了矩阵大小以及分配的内存，因此我们不应该将 `data` 指针直接赋值，而是需要使用深复制的方式来实现。

```
Mat_ &operator = (const Mat_ &other) {
    if (this != &other) {
        MatOp::assign_pwise(*this, other, [](T& val) { return val; });
    }
    return *this;
}
```

这里我们使用了一个 `MatOp::assign_pwise` 函数来实现逐元素的复制。

### §3.2. NHWC or NCHW?

有了存储数据的容器，我们就需要考虑如何排布数据了。由于图像相比于二维矩阵多了一个 Channel 维度，因此我们需要考虑如何在内存中存储这个维度的数据。借用一下图像处理中经常出现的两个术语：NHWC 和 NCHW。他们分别表示 [batch, height, width, channel] 和 [batch, channel, height, width] 的存储方式。



这张图比较能清晰说明三通道 RGB 图像对应的 NHWC 和 NCHW 的存储方式。可以看出 NHWC 的存储方式是以像素为单位进行存储的，而 NCHW 的存储方式是以通道为单位进行存储的。这两种存储方式各有优缺点，下面我们来分析一下。

内存布局	优点	缺点
NCHW	<ol style="list-style-type: none"> <li>1. 在 GPU 中计算卷积时,比 NHWC 要快 2.5 倍左右。这是因为在 GPU 中,NCHW 格式的数据布局更符合 GPU 的内存访问模式和计算方式。</li> <li>2.NCHW 格式更适合那些需要对每个通道单独做运算的操作,比如“MaxPooling”。这是因为 NCHW 格式的同一通道的像素值连续排布,使得对每个通道的数据可以更高效地进行运算。</li> </ol>	<ol style="list-style-type: none"> <li>1.NCHW 格式需要把所有通道的数据都读取到,才能进行运算,因此在计算时需要的存储更多。这可能会限制其在一些具有限制的硬件环境下的应用。</li> <li>2.NCHW 格式的访存与计算的逻辑相对简单,这使得在一些需要精细控制访存和计算的场景下,可能不是最佳的选择。</li> </ol>
NHWC	<ol style="list-style-type: none"> <li>1.NHWC 格式的访存局部性更好。这意味着每三个输入像素就可以得到一个输出像素,因此在一些特定的计算操作中,可以更高效地利用硬件资源。</li> <li>2.NHWC 格式更适合那些需要对不同通道的同一像素做某种运算的操作,比如“Conv1x1”。这是因为 NHWC 格式的不同通道中的同一位置元素顺序存储,使得对不同通道的数据可以进行更高效的运算。</li> <li>3.NHWC 格式在早期的 CPU 开发中应用较多,因此对于主要基于 CPU 开发的深度学习框架和算法,NHWC 格式可能更受欢迎。</li> </ol>	<ol style="list-style-type: none"> <li>1. 在使用 GPU 进行计算加速时,NHWC 格式不如 NCHW 格式高效。这是因为 NCHW 格式更符合 GPU 的内存访问模式和计算方式。</li> <li>2. 对于一些需要精细控制访存和计算的场景,NHWC 格式的控制逻辑可能相对复杂一些。</li> </ol>

说了这么多,我们综合考虑需要批量处理颜色像素值,而且我们也不需要考虑 GPU 的加速问题,选择 NHWC 作为我们存储数据的格式应该会比较合理的选择。

确定了存储格式之后,我们就可以定义矩阵元素访问的操作。一个 numpy 风格的矩阵元素访问操作是通过重载 operator()来实现的,它相比于 operator[]的好处在于可以使用多维索引来访问矩阵元素,对于项目来说可读性更好。

```
T& operator()(size_t i, size_t j, size_t k) {
    return data.get()[i * stride_ * CHANNELS + j * CHANNELS + k];
}
```

### §3.3. 类型与静态编译技术

我们知道,泛型编程一直是 C++ 相对于 C 的一个重要特性,他的强大之处体现在现代 C++ 的各种编程范式中。尤其是 boost 库和 STL 的设计中,泛型编程的优势被发挥到了极致。如果能够将泛型融入到我们的矩阵类中,那么我们就可以在编译时确定一些参数的检查,从而避免运行时错误发生的滞后性。

比如我们用于图像的矩阵通道一般在编译时就可以确定了,因此我们可以将它作为一个模板的非类型参数传入到矩阵类中。

```
template<typename T, size_t CHANNELS>
class Mat_ {
    // ...
}
```



```

    static_assert(CHANNELS > 0, "CHANNELS must be greater than 0");
    static_assert(std::is_arithmetic<T>::value || std::is_enum<T>::value, "T must be
an arithmetic type");
    // ...
};

```

其中这里有两个 `static_assert` 语句，通道数为 0 对于图像来说是没有意义的，因此我们需要在编译时检查通道数是否大于 0。而在进行矩阵运算时，我们需要保证矩阵元素的类型是可计算的，因此我们需要在编译时检查元素类型是否为数值类型或者枚举类型。`std::is_arithmetic` 和 `std::is_enum` 都是 C++11 引入的类型特性，可以在类实例化时就检查类型是否符合要求。

卷积核需要的矩阵大小在编译时也可以确定，因此我们参照 `opencv/Matx` 的设计，将卷积核的大小也作为一个模板的非类型参数传入到矩阵类中，构建一个完全静态的矩阵类。

```

template<typename T, size_t R, size_t C>
class Matx {
    // ...
    static_assert(std::is_arithmetic<T>::value || std::is_enum<T>::value, "T must be
an arithmetic type");
    static_assert(R > 0 && C > 0, "R and C must be greater than 0");
    // ...

private:
    std::array<T, R * C> data;
    // ...
};

```

这样的定义有什么好处呢？如果我们希望特定类型卷积核的生成不产生运行时开销，同时又不想在库中重复定义不同大小和不同参数常量矩阵，那么我们可以利用 C++11 的一个重要特性：`constexpr`，用它标记一个函数，这个函数只接受静态参数，同时其在编译期间就可以展开计算，相当于编译器自动帮我们生成了不同的常量卷积核，比如使用这种方法生成一个归一化 gaussian 卷积核：

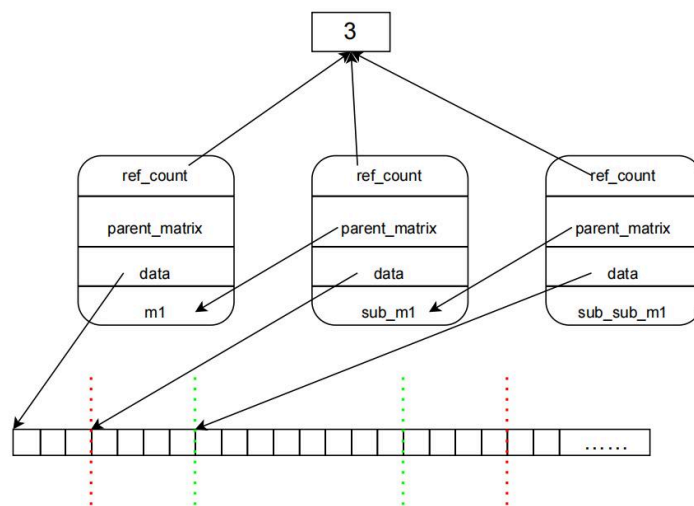
```

template<size_t K>
inline Matx<float, K, K> constexpr getGaussianKernel(float sigma) {
    static_assert(K % 2 == 1, "Kernel size must be odd");
    Matx<float, K, K> kernel;
    float sum = 0.0f;
    float mean = static_cast<float>(K - 1) / 2.0f;
    float sigma2 = 2.0f * sigma * sigma;
    for (int i = 0; i < K; ++i) {
        for (int j = 0; j < K; ++j) {
            float x = i - mean;
            float y = j - mean;
            kernel(i, j) = std::exp(-(x * x + y * y) / sigma2);
            sum += kernel(i, j);
        }
    }
    // normalize
    for (int i = 0; i < K; ++i) {
        for (int j = 0; j < K; ++j) {
            kernel(i, j) /= sum;
        }
    }
    return kernel;
}

```

### §3.4. ROI

在计算机视觉领域中，ROI（Region of Interest）是一个非常重要的概念，它指的是图像中我们感兴趣的区域。ROI可以是一个矩形、圆形或者任意形状的区域，通常用于图像处理、目标检测和跟踪等任务中。这个项目里，我们支持矩形 ROI 的定义，使用 `operator()` 就可以通过传入 `Rect` 对象来访问 ROI 区域的矩阵视图。以下就是 OpenCV 中矩形 ROI 的内存引用方式：



It is unacceptable because `sub_sub_m1[i][j]=sub_sub_m1.data[i * sub_sub_m1->parent_matrix->ncols + j]=sub_sub_m1.data[i * sub_m1.ncols + j]` and it is wrong.

可以看到，我们也可以通过类似的方式，通过偏移数据首地址和重写列宽来实现对 ROI 的引用。这里就是我们使用 `stride_` 参数的原因了，它可以帮助我们在不改变数据存储方式的情况下，改变访问时的步长，从而利用新视图读写原矩阵对应的区域。一个 ROI 的创建的实现可以简单地写成这样：

```
Mat_ Mat_(const Mat_& other, const Rect& roi) :
    rows_(roi.height), cols_(roi.width), stride_(roi.width),
    data(other.data), ptr(other.ptr + roi.y * other.stride_ + roi.x) {
    assert(roi.x >= 0 && roi.y >= 0 && roi.x + roi.width <= other.cols_
           && roi.y + roi.height <= other.rows_);
}
```

### §3.5. 基于方法重写的通道视图

在图像处理中，我们经常需要对图像的不同通道进行操作，比如对 RGB 图像的每个通道进行不同的处理。如果能方便地在一个通道中迭代访问数据，那么处理起来就会非常顺畅。

我们可以使用一个 `ChannelView` 类来实现对通道的视图访问，它需要存储两个参数来保证正确的访问数据：一个是原始矩阵的引用，另一个是通道的索引。

```
class ChannelView {
protected:
    Mat_& par_;
    size_t channel_offset_;
    // ...
public:
```

```

    ChannelView(Mat_& par, size_t channel) : par_(par), channel_offset_(channel)
{
    assert(channel < CHANNELS);
}

T& operator()(size_t i, size_t j, size_t _ = 0) {
    return par_.data.get()[i * par_.stride_ * CHANNELS + j * CHANNELS +
channel_offset_];
}

const T& operator()(size_t i, size_t j, size_t _ = 0) const {
    return par_.data.get()[i * par_.stride_ * CHANNELS + j * CHANNELS +
channel_offset_];
}
};

```

从一个矩阵中获取通道视图的操作可以写成这样：

```

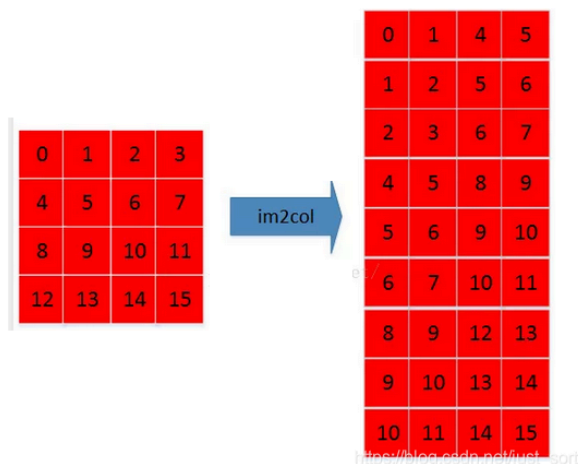
ChannelView Mat_::slice(size_t channel) {
    assert(channel < CHANNELS);
    return ChannelView(*this, channel);
}

```

很显然我们希望这个矩阵视图类能够使用 Mat\_ 的所有矩阵操作，这个可以通过继承 Mat\_ 类来实现。

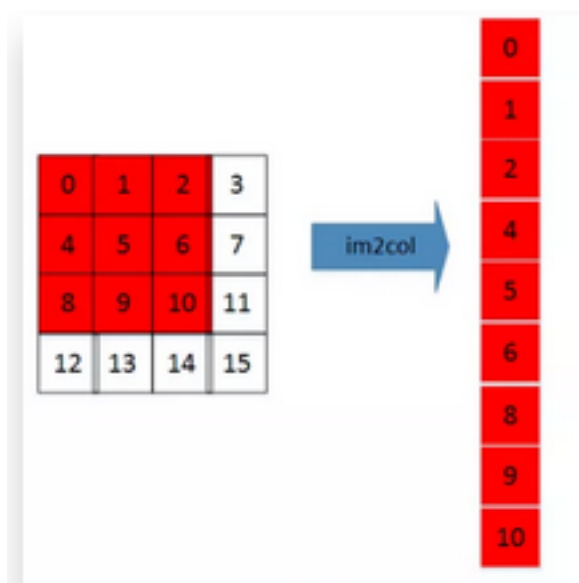
## §4. 卷积计算的优化 - Im2Col

首先, 让我们来考虑一个单通道的长宽均为4的输入特征图, 并且假设卷积核的大小是  $3 \times 3$ , 那么它经过 Im2Col 之后会变成什么呢? 如下图所示:

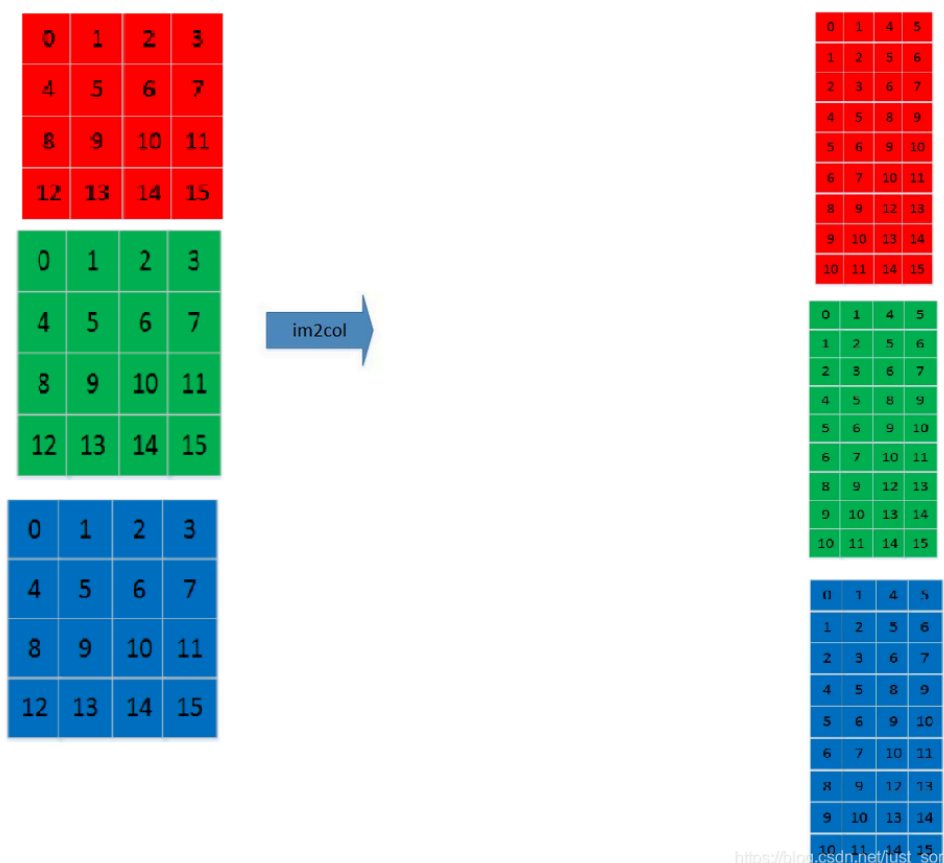


其实变换的方式非常简单，例如变化的第一步如下图所示：

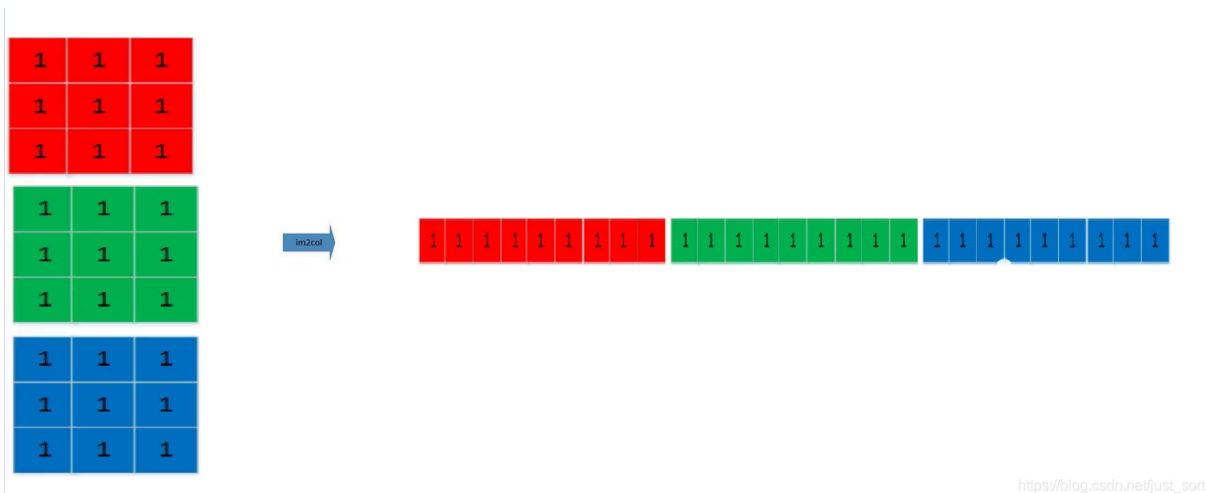




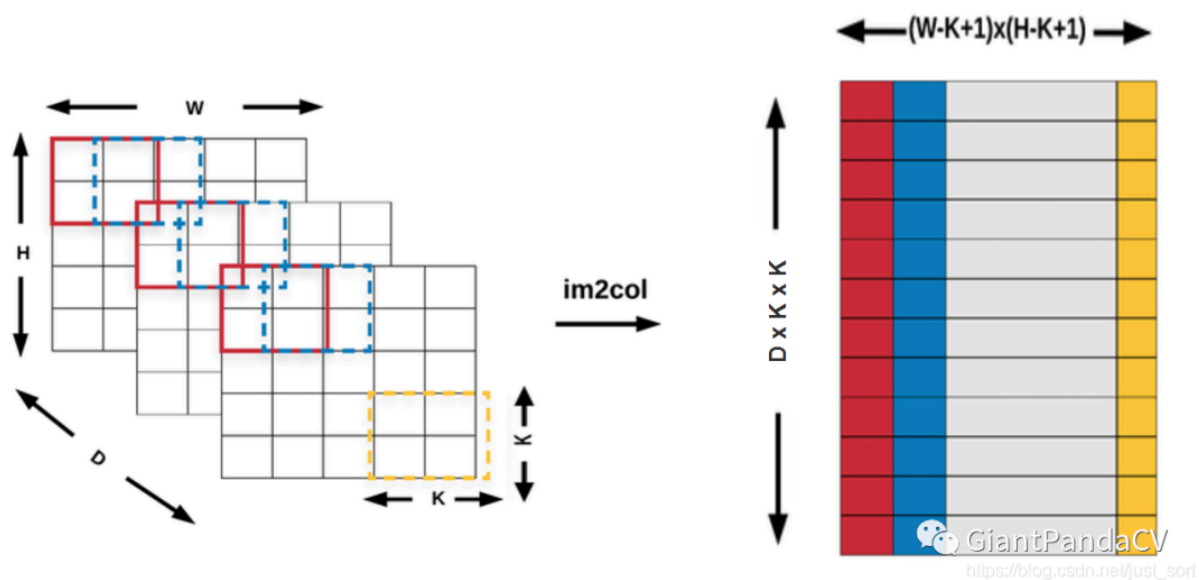
直接在卷积核滑动的范围内，把所有元素顺序排列成一列即可，其它位置处同理即可获得 Im2Col 的结果。多通道和单通道的原理一样，如下图所示：



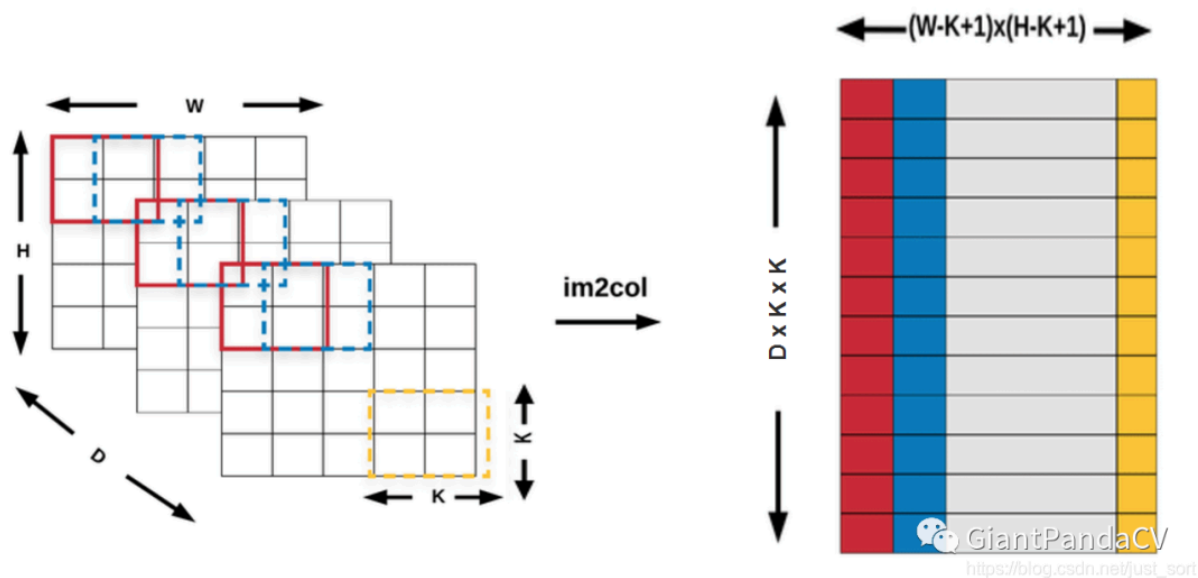
其次，让我们来考虑一下卷积核经过 Im2Col 操作后应该是什么样子的，对于多通道如下图所示（单通道就是其中一种颜色）：



最后，输入特征图和卷积核都进行了 Im2Col 之后其实都变成了一个二维的矩阵，我们就可以使用矩阵乘法来直接计算结果了。下图展示了一个  $4 \times 4 \times 3$  的特征图和一个  $3 \times 3 \times 3$  的卷积核经过 Im2Col 之后之后使用 Sgemm 进行计算的过程，其中每一种颜色都代表一个输出通道。



看了上面几个图之后，Im2Col 的原理也就比较显然了，为了更加严谨，下面还是以公式化的形式再来总结一下：假设输入的特征图维度是  $(1, D, H, W)$ ，表示 Batch 为 1，通道数为  $D$ ，高为  $H$ ，宽为  $W$ ，卷积核的维度是  $(D_{out}, D, K, K)$ ，表示输出通道数为  $D$ ，卷积核大小是  $K \times K$ ，则输入特征图的 Im2Col 过程如下所示，窗口从左到右上到下的顺序在每个输入通道同步滑动，每个窗口内容按行展开成一行，然后再按通道顺序接上填到 im2col buffer 对应的列，并且 im2col buffer 按照从左到右的顺序填写。



由于这个地方没有 padding 并且步长是 1，那么卷积的输出特征图的高为

$H_{\text{out}} = \frac{H+2p-K}{\text{Stride}} + 1 = H - K + 1$  同理宽为  $W - K + 1$ 。所以最后 im2col buffer 的宽维度为  $(H - K + 1) \times (W - K + 1)$ ，高维度则是  $K \times K \times D$ 。最后再把权值 im2col 成  $(D_{\text{out}}, D \times K \times K)$ ，这样卷积就可以直接变成两个二维矩阵的乘法了。

最后再获得了乘积矩阵之后只需要按照通道进行顺序排列就可以获得输出特征图了，这个过程就是 Im2Col 的逆过程，学名叫作 Col2Im，也比较简单就不再赘述了。

这里我们先做一些约定，变量 src 表示输入特征图的数据，输入特征图的维度是  $[1, \text{inChannel}, \text{inHeight}, \text{inWidth}]$ ，输入卷积核的维度是  $[1, \text{outChannel}, \text{KernelH}, \text{kernelW}]$ ，最后卷积操作执行完之后的输出特征图维度就是  $[1, \text{outChannel}, \text{outHeight}, \text{outWidth}]$ 。

#### §4.1. 对输入图进行 Im2Col

首先我们对输入特征图进行 Im2Col 操作，按照之前的介绍可知，我们要得到的结果矩阵的维度是  $[\text{inChannel} * \text{kernelH} * \text{kernelW}, \text{outHeight} * \text{outWidth}]$ ，这个过程只需要窗口从左到右上到下的顺序在每个输入通道同步滑动，每个窗口内容按行展开成一列，然后再按通道顺序接上填到 im2col buffer 对应的列，并且 im2col buffer 按照从左到右的顺序填写。代码实现如下：

```
// 1. im2col
float *src_im2col = new float[outWidth * outHeight * kernelH * kernelW *
inChannel];

const int Stride = kernelW * kernelH * outHeight * outWidth;
//const int inSize = inHeight * inWidth;
const int outSize = outHeight * outWidth;
const int kernelSize = kernelH * kernelW;

// inChannel x kW x kH
// outWidth x outHeight

for(int cc = 0; cc < inChannel; cc++){
    const float *src0 = src + cc * kernelH * kernelW * inChannel;
    int dst_idx = Stride * cc;
```

```

        for(int i = 0; i < kernelH; i++){
            for(int j = 0; j < kernelW; j++){
                for(int x = 0; x < outHeight; x++){
                    for(int y = 0; y < outWidth; y++){
                        int row = x * StrideH + i;
                        int col = y * StrideW + j;
                        int ori_idx = row * inWidth + col;
                        src_im2col[dst_idx] = src0[ori_idx];
                        dst_idx++;
                    }
                }
            }
        }
    }
}

```

## §4.2. 对卷积核进行 Im2Col

按照我们第二节的介绍，我们同样可以轻易写出对卷积核进行 Im2Col 的操作，代码如下：

```

int Stride = 0;
for(int cc = 0; cc < outChannel; cc++){
    int c = cc;
    const float* k0 = kernel + c * inChannel * kernelSize;
    Stride = kernelSize * inChannel;
    float* destptr = dest + c * Stride;
    for(int i = 0; i < inChannel * kernelSize; i++){
        destptr[0] = k0[0];
        destptr += 1;
        k0 += 1;
    }
}

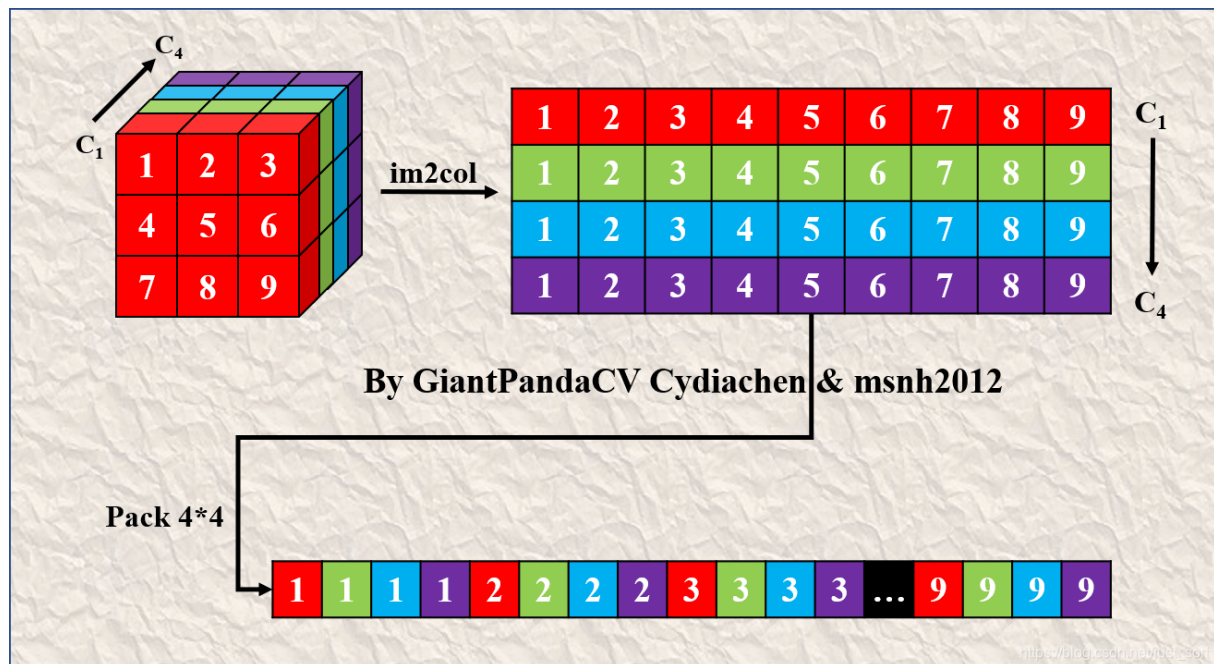
```

## §4.3. 优化点之 Pack 策略

按照上面介绍的思路，在获取了输入特征图和卷积核的 Im2Col 变换矩阵之后其实就可以利用 Sgemm 计算出卷积的结果了。

但是如果直接使用矩阵乘法计算，在卷积核尺寸比较大并且输出特征图通道数也比较大的时候，我们会发现这个时候 Im2Col 获得矩阵是一个行非常多列非常少的矩阵，在做矩阵乘法的时候访存会变得比较差，从而计算效率边地。这是因为当代 CPU 架构的某些原因，导致程序在行方向上的处理始终比列方向慢一个档次，所以我们如果能想个办法使得行方向的元素变少，列方向的元素变多这样就可以有效改善缓存达到加速的目的，另外列方向上元素增多更容易让我们发挥 SIMD 指令集的作用。所以，这就是本此项目将要提到的第一个优化技巧，数据打包 (Pack)。

具体来说，对于卷积核我们进行  $4 \times 4$  的 Pack（所谓  $4 \times 4$  的 Pack 就是在 Im2Col 获得的二维矩阵的高维度进行压缩，在宽维度进行膨胀，参考一下下面的图应该很好理解），如下图所示：



这是一个  $3 \times 3$  的卷积核并且输出通道数为4，它经过 Im2Col 之后首先变成上图的上半部分，然后经过 Pack  $4 \times 4$  之后变成了上图的下半部分，即变成了每个卷积核的元素按列方向交织排列。

这部分的代码也比较容易实现，另外一个技巧是，每次在执行卷积计算时，对于 Image 的 Im2col 和 Pack 每次都会执行，但对于卷积核，Im2col 和 Pack 在任意次只用做一次，所以我们可以模型初始化的时候提前把卷积核给 Pack 好，这样就可以节省卷积核 Im2Col 和 Pack 耗费的时间，代码实现如下：

```
void transformKernel(float *const &kernel, const int &kernelW, const int &kernelH,
float* &dest, const int &inChannel,
                    const int &outChannel){

    int kernelSize = kernelH * kernelW;
    int ccOutChannel = 0;
    int ccRemainOutChannel = 0;
    int Stride = 0;

    ccOutChannel = outChannel >> 2;
    ccRemainOutChannel = ccOutChannel << 2;

    for(int cc = 0; cc < ccOutChannel; cc++){
        int c = cc << 2;
        const float* k0 = kernel + c * inChannel * kernelSize;
        const float* k1 = kernel + (c + 1) * inChannel * kernelSize;
        const float* k2 = kernel + (c + 2) * inChannel * kernelSize;
        const float* k3 = kernel + (c + 3) * inChannel * kernelSize;

        Stride = 4 * kernelSize * inChannel;
        float* destptr = dest + (c / 4) * Stride;

        for(int i = 0; i < inChannel * kernelSize; i++){
            destptr[0] = k0[i];
            destptr[1] = k1[i];
            destptr[2] = k2[i];
            destptr[3] = k3[i];
        }
    }
}
```



```

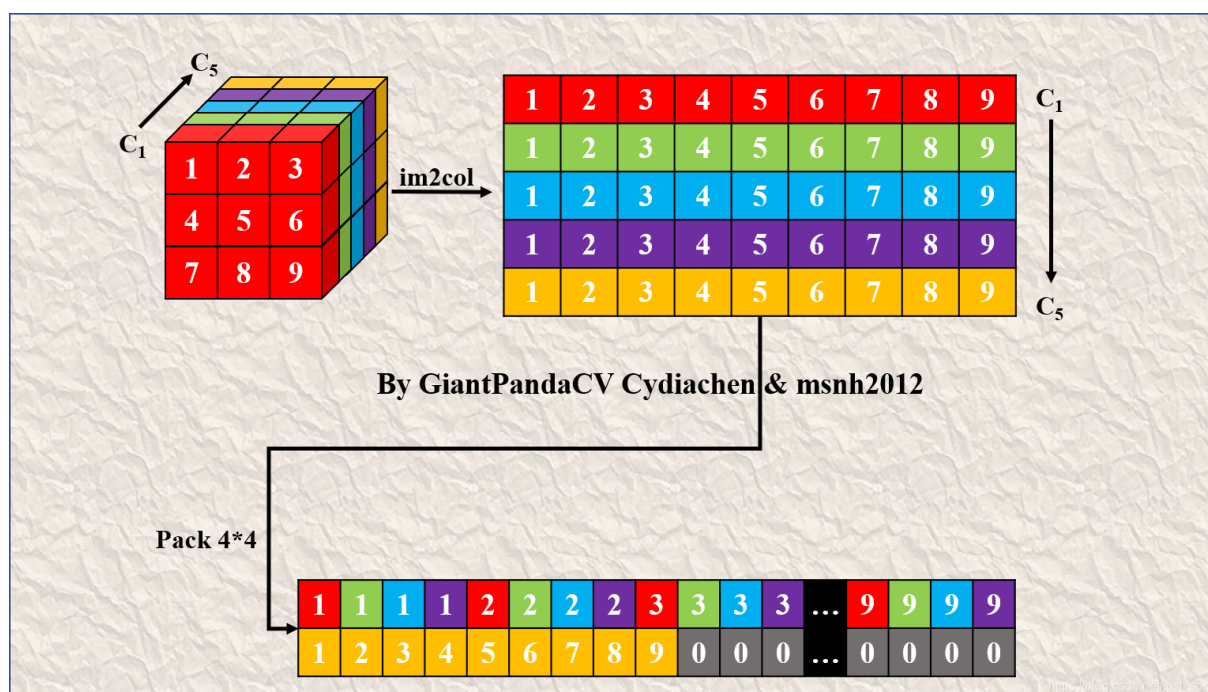
        destptr += 4;

        k0 += 1;
        k1 += 1;
        k2 += 1;
        k3 += 1;
    }
}

for(int cc = ccRemainOutChannel; cc < outChannel; cc++){
    // process the tail
    // ...
}
}

```

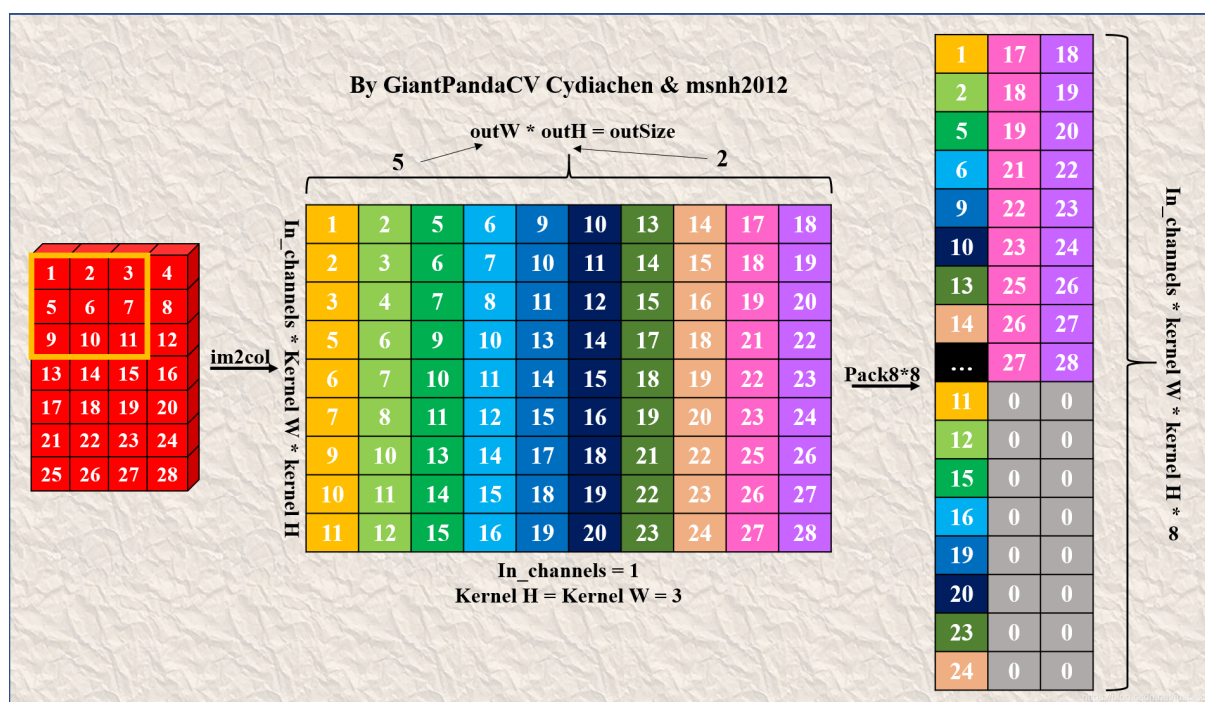
注意这个地方如果 Pack 的时候有拖尾部分，也就是说 outChannel 不能被 4 整除时，那么直接按照原始顺序排列即可，后面在 Sgemm 方法计算的时候需要注意一下。下图展示了一个  $5 \times 4 \times 3$  的卷积核经过 Im2Col 再经过 Pack 4x4 之后获得的结果，可以看到拖尾那一行是直接复制的，不够的长度直接置 0 即可。



接下来继续说一下对于输入数据的 Pack，我这里以对输入数据进行 8x8 的 Pack 为例子来说明输入数据 Pack 之后应该是什么样子，以及如何和刚才已经 4x4Pack 好的卷积核执行矩阵乘法以完成整个卷积过程。至于为什么这里要使用 Pack8x8 而不是 Pack4x4，主要是因为 Pack4x4 的话在行方向上元素太少了，导致 SIMD 指令集的利用率不高，所以这里选择了 Pack8x8。Pack8x8 的原理和 Pack4x4 一样，只不过是将 Im2Col 之后的矩阵在行方向上进行压缩，在列方向上进行膨胀，下面是一个示意图：

下面展示了一个输入维度为  $[1, 1, 7, 4]$  的特征图使用 Im2Col 进行展开并 Pack8x8 之后获得结果（卷积核维度为  $[1, 1, 3, 3]$ ），其中左边代表的是 Im2Col 后的结果，右边是将其进一步 Pack8x8 后获得的结果。这个地方由于  $outHeight \times outWidth$  不能被 8 整除，所以存在拖尾部分无法进行 Pack，即结果图中的第二和第三列，直接顺序放就可以了。

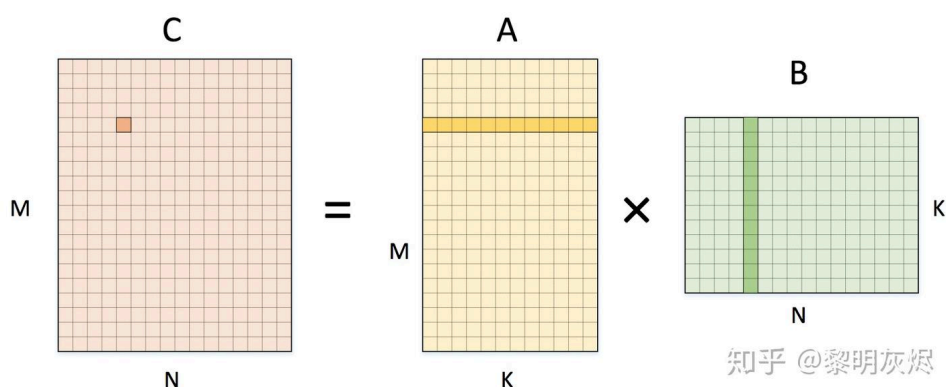




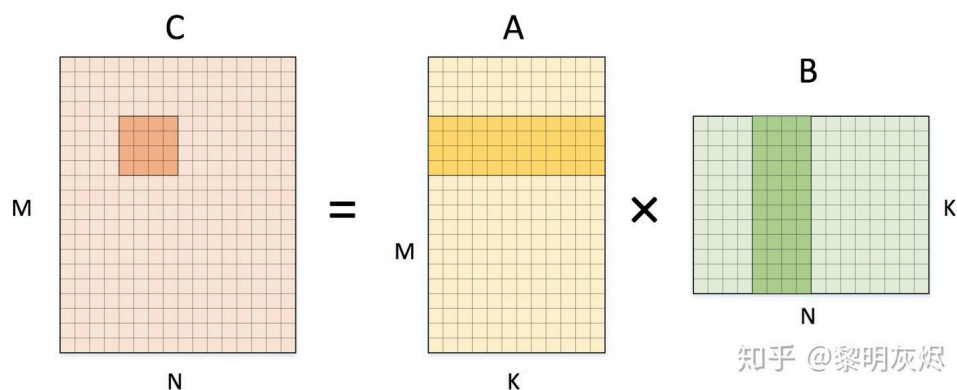
#### §4.4. 优化点之带 Pack 的矩阵乘法

现在我们已经获得了输入特征图和卷积核的 Im2Col+Pack 操作后的矩阵，那么接下来就是 CPP 历代 project 中经典的 optimized 矩阵乘法了(没想到这里也能 GEMM)。这个优化问题在业界普遍接受的思路主要有两种，一种是使用 GPU 的并行计算能力来加速矩阵乘法，另一种则是将矩阵分块并进行缓存友好的数据打包并结合 SIMD 指令集来加速矩阵乘法。

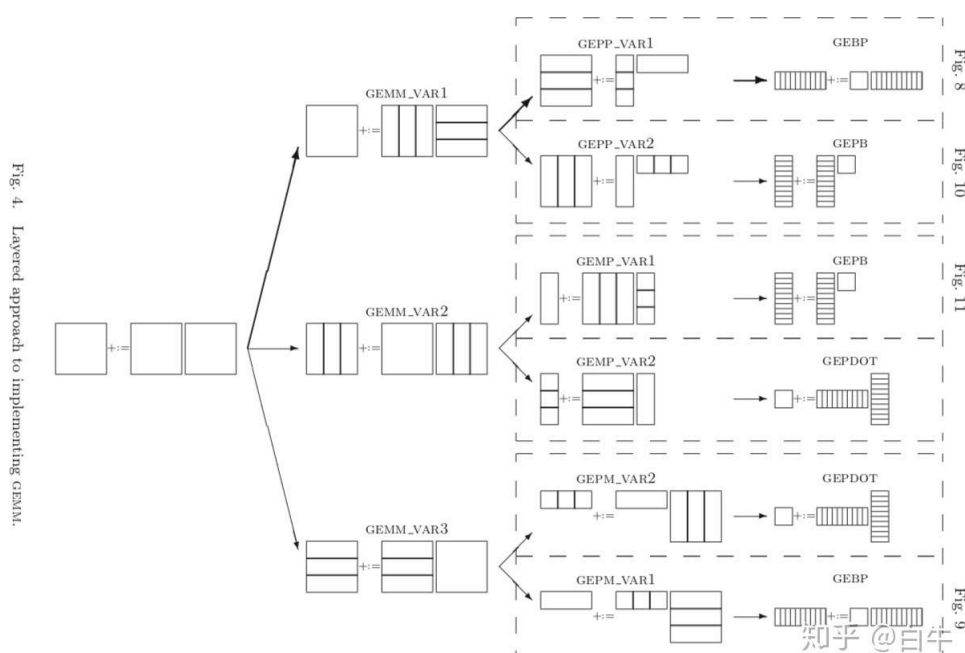
为什么要分块？首先我们观察一下朴素的矩阵乘法算法，假设我们有两个矩阵 A 和 B，分别是  $M \times K$  和  $K \times N$  的矩阵，那么我们可以用下面的图示意来表示它们的乘法过程：



其中可以注意到，在计算 A 行数矩与 B 列数据的点乘时，我们可以用向量化的方法，一次性读入多个数据进行计算，这样可以有效的减少访存次数，提高计算效率。为了向量化，数据分块是必不可少的，而且需要考虑 L1 缓存和 TLB size 大小的影响，一般会选择将数据分成  $4 \times 4$  或者  $8 \times 8$  的块进行计算，这样可以最大限度地利用局部性原理。又由于每个块的计算是独立的，我们又可以引入 openmp 来进行多线程计算，这样可以充分利用多核 CPU 的计算能力。应用了分块的矩阵乘法算法如下所示：



其中分块的思路又有很多，根据这篇文章，我们直接用一张图展示不同主序的矩阵输入分块策略：



考虑到我们先前的 pack 方式，数据在内存中的存储顺序是行优先的，因此 Fig.10 对应的 GEPB 策略更加衔接我们的 im2col 算法。有了具体的分块策略，我们就可以按照 GEMM 的思路来实现我们处理打包数据的矩阵乘法了。这个 sGEMM 的接口定义如下：

```
sgemm (int M, int N, int K, float *A, float *B, float *C)
```

其中输入矩阵 A 的特征维度是  $M \times K$ ，输入矩阵 B 的特征维度是  $K \times N$ ，输出矩阵 C 的特征维度是  $M \times N$ ，不考虑任何优化的情况下复杂度是  $O(N \times K \times K)$ 。因为我们这里 Pack 了数据，所以访存相比于原始版本会变好一些，但计算量实际上还是没变的。除此之外，由于行方向的数据增多我们可以更好的在行方向上进行利用 SIMD 优化使得整个计算过程的效率更好。

所以就目前的情况来说，矩阵 A 就是我们的卷积核经过 Im2Col+Pack4x4 获得的输出矩阵，矩阵 B 就是我们的输入特征图经过 Im2Col+Pack8x8 之后获得的输出矩阵，然后矩阵 C 就是经过矩阵乘法之后获得的输出特征图了。在实现矩阵乘法的时候，以矩阵 A 为基准一次处理 4 行，并且在列方向分别处理 4 或者 8 个元素，这个可以结合上面的输入特征图的 Im2Col+Pack8x8 的示意图来想。最后，对于矩阵 A 不够 4 的拖尾部分，进行暴力处理即可。这个地方考虑的是

当代典型的 CNN 架构一般通道数都是 4 的倍数，暂时没有实现拖尾部分的 SIMD 优化，下面先看一下这个算法实现的整体部分：

这部分的代码实现如下：

```
// sgemm (int M, int N, int K, float *A, float *B, float *C)
// A (M x K)
// B (K x N)
// C (M x N)
```

#### 4. 优化效果

下面给出一些测试数据来验证一下此算法的有效性，下面的测试结果都默认开启了 SIMD 优化，x86\_64 架构下使用 AVX 指令集。测试的机器是 Intel i7-13600KF CPU @ 3.50GHz，内存为 32GB，操作系统为 Ubuntu 22.04。

输入大小	输入通道数	卷积核通道数	卷积核大小	步长	方法	速度
14x14	512	1024	3x3	1	3x3s1 手工优化	430ms
14x14	512	1024	3x3	1	Im2col+Pack+Sgemm	315ms
14x14	512	1024	3x3	2	3x3s2 手工优化	120.49ms
14x14	512	1024	3x3	2	Im2col+Pack+Sgemm	93ms

可以看到这里 Im2Col+Pack+Sgemm 的速度明显快于之前的手动展开版本，在 stride 为 1 时优化了 100+ms。

下面我们再测一组更大的特征图看一下此算法的表现，一般特征图比较大那么通道数就比较小，我们对应修改一下：

输入大小	输入通道数	卷积核通道数	卷积核大小	步长	方法	速度
112x112	64	128	3x3	1	3x3s1 手工优化	542ms
112x112	64	128	3x3	1	Im2col+Pack+Sgemm	588.75ms
112x112	64	128	3x3	2	3x3s2 手工优化	97.43ms
112x112	64	128	3x3	2	Im2col+Pack+Sgemm	138.90ms

相比于手动展开的版本，Im2Col+Pack+Sgemm 的速度反而变慢了，这个时候我们可以看到优化版本比手动展开的版本慢了 50ms 左右。这个时候我们可以看到 Im2Col+Pack+Sgemm 的速度反而比手动展开的版本慢了 50ms 左右。

有一个直观的猜测就是当输入通道数和卷积核尺寸的乘积也就是卷积核 Im2Col 获得的矩阵的宽度比较大时，这里的 Im2Col+Pack+Sgemm 能带来加速效果。在输入特征图通道数较小并且特征图较大时甚至会比手工优化更慢，因为这个时候对访存的优化效果很小，因为这个时候 Im2Col 的矩阵行数和列数差距并不大，这个时候手工优化带来的 Cache Miss 并不会对卷积的计算带来很大的影响。

## §5. 总结

这个项目相比与 C 语言的版本来说，很显然 C++17 的新特性是一把双刃剑。我在项目中大量尝试使用智能指针、函数式接口和模板特征编程等现代 C++ 特性来改进 C 语言版本中不够灵活的部分。起初在处理大量冒出的编译错误时，我也对于这些尝试有些怀疑。不过在一个一个解决每个问题后，我也成功地跳出了 C 语言风格的舒适圈，真正感受到了 C++ 的自由和高效。

而且相比于 C 语言开发的项目来说, C++ 的编译器在编译时就能检查出很多潜在的错误, 这也让很多逻辑问题不会在运行时才暴露出来。

而且, 对于上一个项目没有实现的卷积优化的缺憾, 也在这个项目中得到了弥补。Im2Col 通过将优化建立在已臻完善的 GEMM 之上, 实质上将卷积操作转化为矩阵乘法, 充分利用现代 CPU 和 GPU 架构对矩阵运算的优化特性。这种方法不仅在理论上优雅, 在实践中也体现了显著的性能提升。

在 C++17 的加持下, 我们实现了代码的高度抽象与高效执行的和谐统一, 这正是 Bjarne Stroustrup 设计 C++ 的初衷: “让抽象不付出性能代价”。总结而言, 本项目不仅仅是从 C 到 C++ 的技术迁移, 更是一次编程范式的转变与提升。我们不只获得了一个性能更优、架构更清晰的图像处理库, 更重要的是获得了对现代软件工程思想的深刻理解与实践。