

结合SIMD和GEMM的快速卷积 及其在高斯模糊中的应用

汇报人：金閔奇

名词解释：

SIMD: Single Instruction Multiple Data 单指令多数据技术，同时对4个double做+*/读写

GEMM: General Matrix Multiplication 通用矩阵相乘的优化算法， $A*B=C$

高斯模糊：高斯核在图像上做滑动乘积求和的卷积运算，使图像产生模糊效果

项目源码：

<https://github.com/LeonJinC/Fast-Convolution-with-SIMD-and-GEMM>

二维高斯函数的采样和归一化

0.1019	0.1154	0.1019
0.1154	0.1308	0.1154
0.1019	0.1154	0.1019

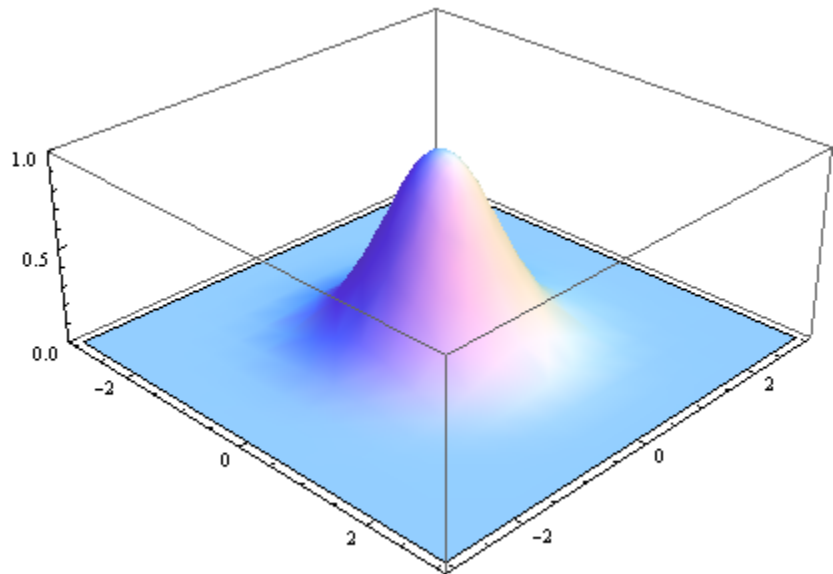
3x3 高斯核($\sigma = 2.0$)

0.0232	0.0338	0.0383	0.0338	0.0232
0.0338	0.0492	0.0558	0.0492	0.0338
0.0383	0.0558	0.0632	0.0558	0.0383
0.0338	0.0492	0.0558	0.0492	0.0338
0.0232	0.0338	0.0383	0.0338	0.0232

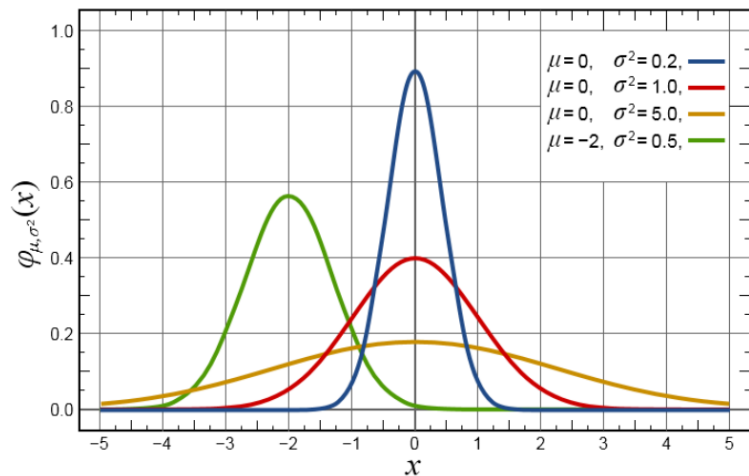
5x5 高斯核($\sigma = 2.0$)

0.0049	0.0092	0.0134	0.0152	0.0134	0.0092	0.0049
0.0092	0.0172	0.025	0.0283	0.025	0.0172	0.0092
0.0134	0.025	0.0364	0.0412	0.0364	0.025	0.0134
0.0152	0.0283	0.0412	0.0467	0.0412	0.0283	0.0152
0.0134	0.025	0.0364	0.0412	0.0364	0.025	0.0134
0.0092	0.0172	0.025	0.0283	0.025	0.0172	0.0092
0.0049	0.0092	0.0134	0.0152	0.0134	0.0092	0.0049

7x7 高斯核($\sigma = 2.0$)



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



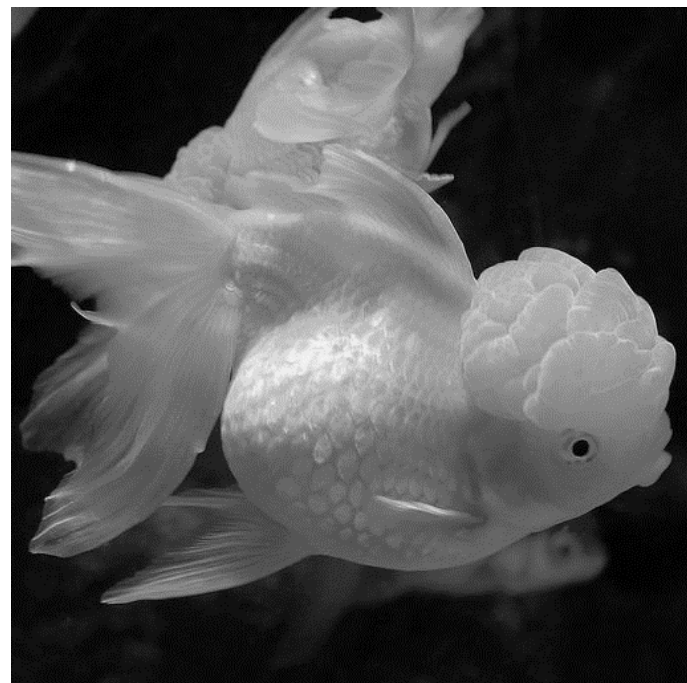
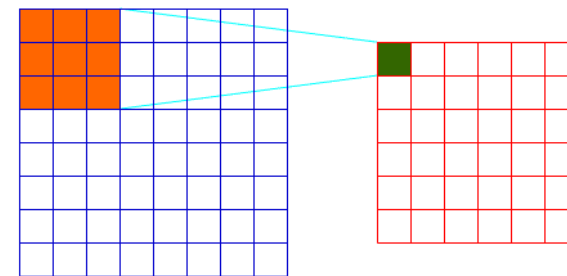
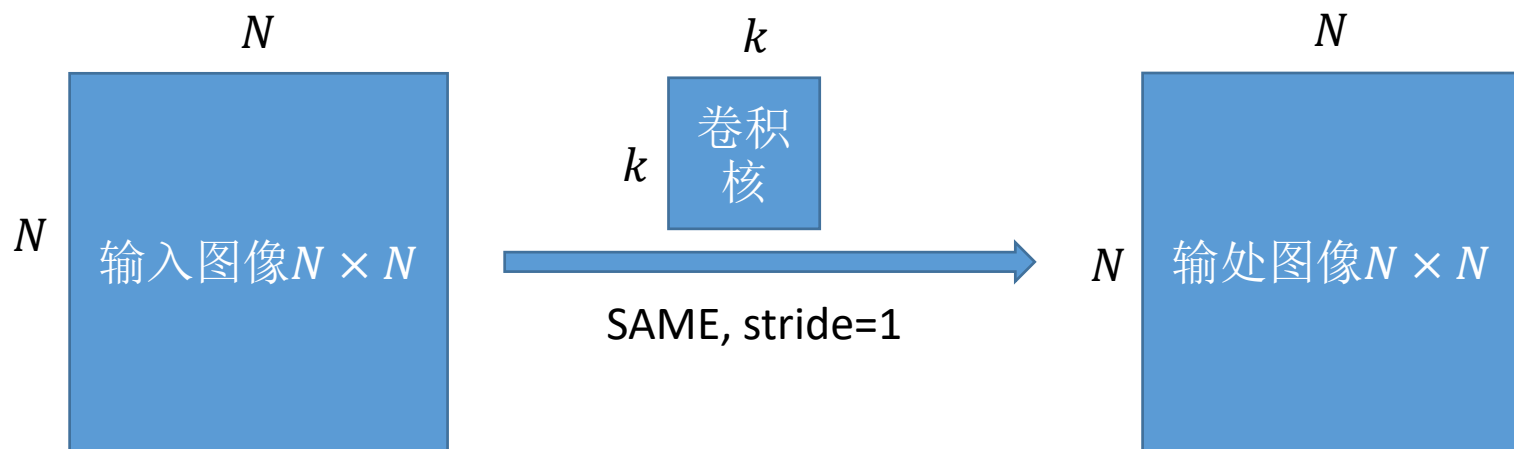
归一化：高斯核中每个元素的范围约束在[0,1]之间，并且总和为1

归一化之后，通过卷积计算出来的模板中心像素被限制到了0-255的灰度区间中。

假若某一邻域内所有像素的灰度值为255，利用该模板进行卷积之后，求得的模板中心像素灰度值仍然为255；

假若计算出来的高斯模板参数之和小于1，那么通过该模板进行卷积之后，模板中心像素的灰度值将小于255，偏离了实际的灰度值，产生了误差。

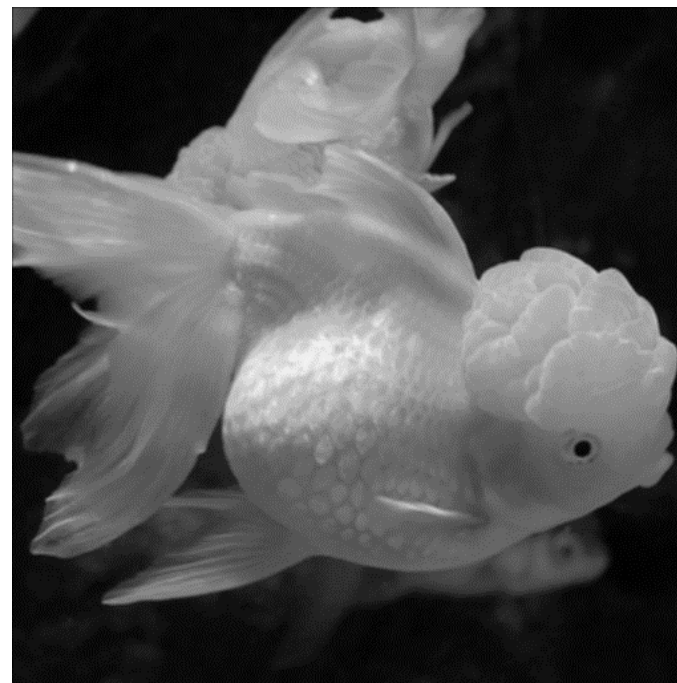
高斯模糊（灰度图）



3x3 高斯核

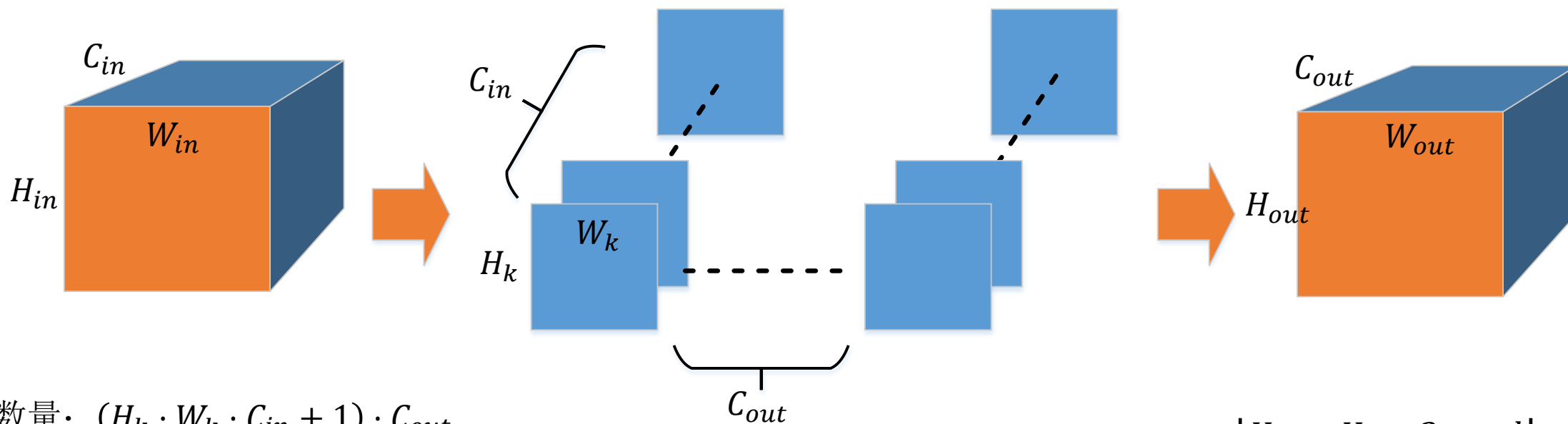
0.1019	0.1154	0.1019
0.1154	0.1308	0.1154
0.1019	0.1154	0.1019

SAME, stride=1



传统卷积(三维矩阵)

$$(C_{in}, H_{in}, W_{in}) \xrightarrow{(C_{in}, C_{out}, H_k, W_k, pad, stride)} (C_{out}, H_{out}, W_{out})$$



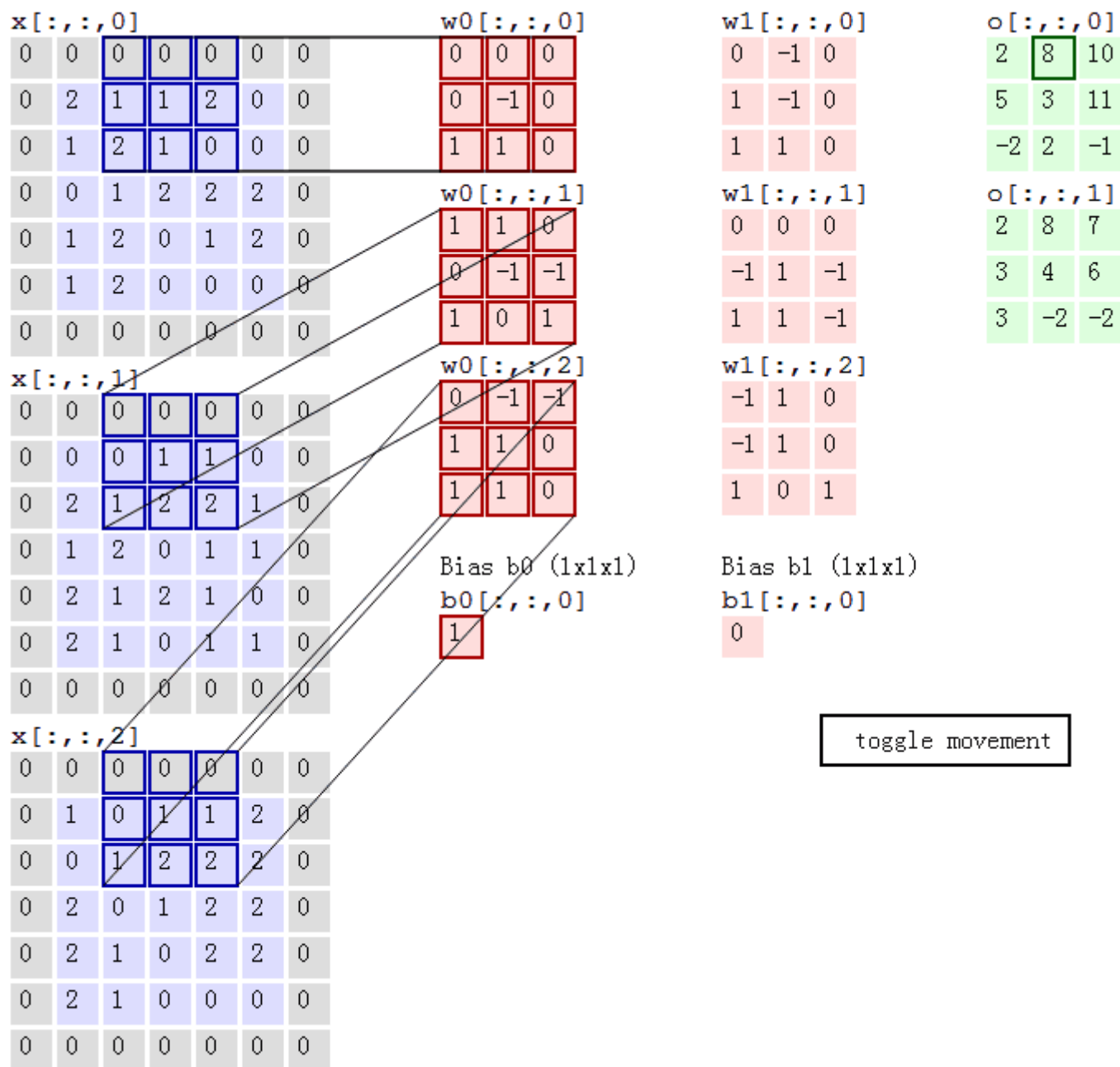
参数量: $(H_k \cdot W_k \cdot C_{in} + 1) \cdot C_{out}$

运算量: $(H_k \cdot W_k \cdot C_{in} + 1) \cdot C_{out} \cdot H_{out} \cdot W_{out}$

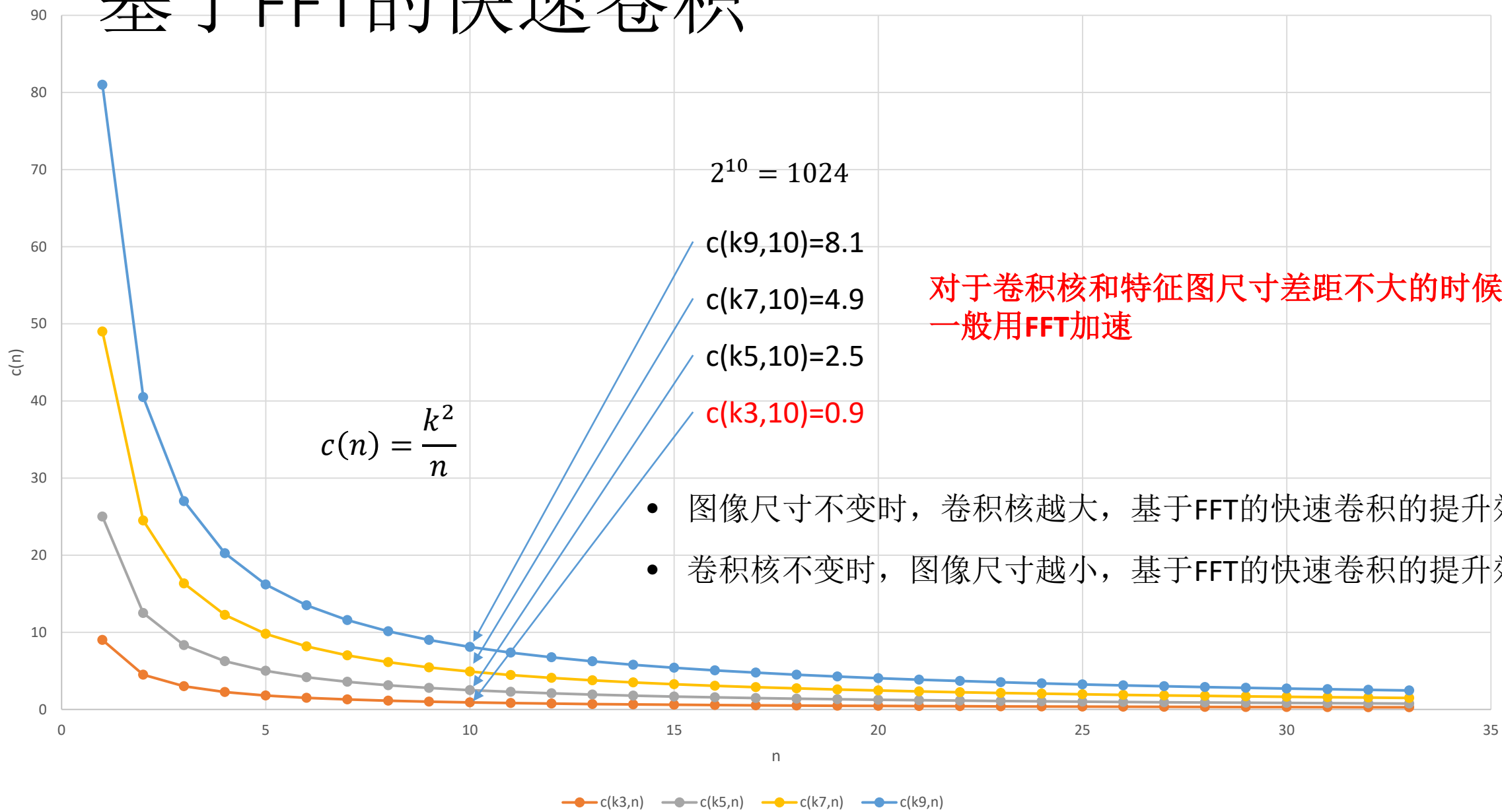
*这里的运算量是权值 W 相关的乘法和偏置 b 相关的加法

$$H_{out} = \left\lfloor \frac{H_{in} - H_k + 2 \cdot pad}{stride} \right\rfloor + 1$$

$$W_{out} = \left\lfloor \frac{W_{in} - W_k + 2 \cdot pad}{stride} \right\rfloor + 1$$



基于FFT的快速卷积

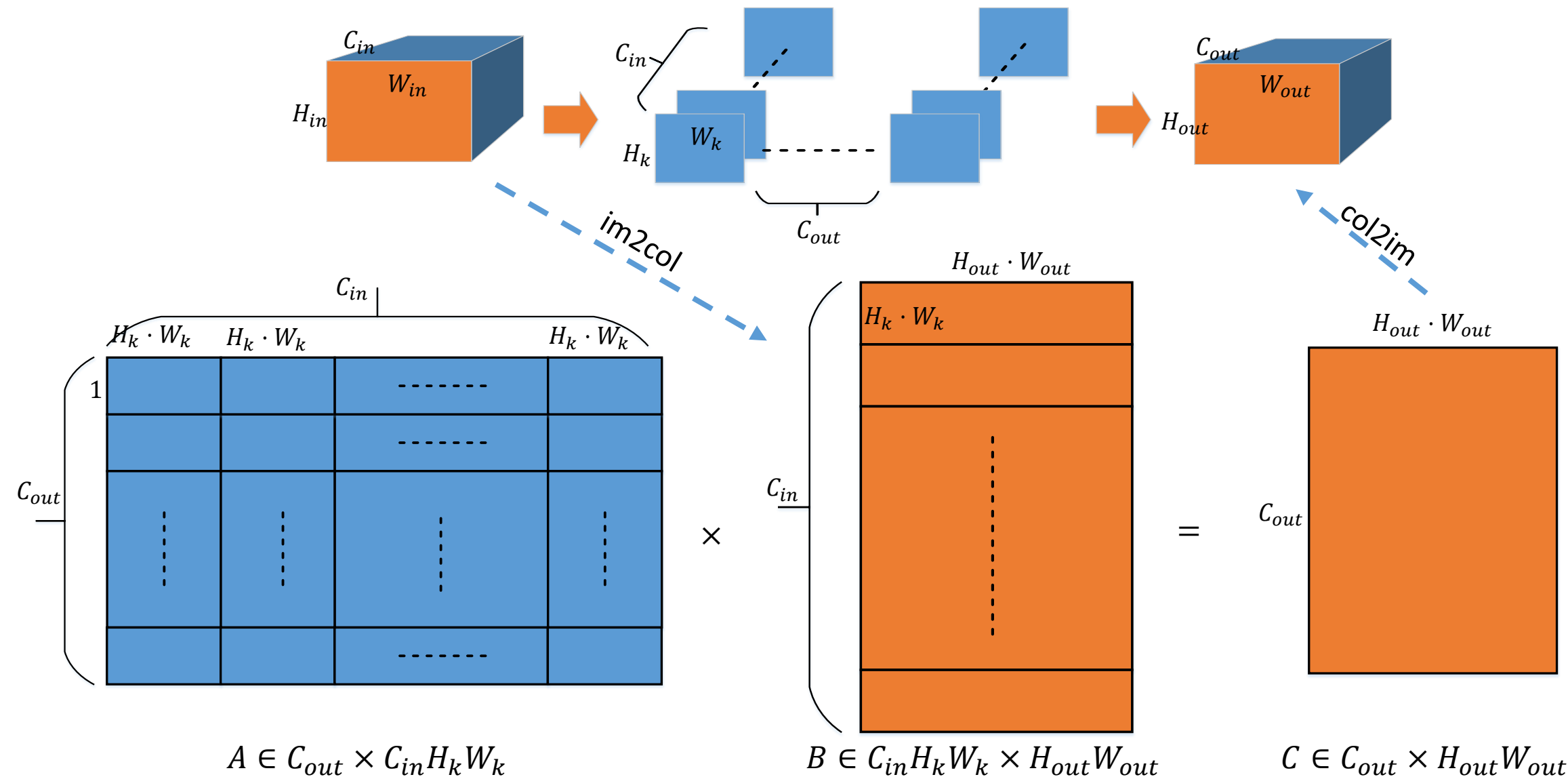


Caffe、MXNet、cuDNN

基于GEMM的快速卷积

优点：从参数量和运算量来看，两者并无不同，但是转化为矩阵后，运算时需要的数据被存放在连续的内存上，访问速度大大加快，同时，矩阵乘法有许多高效实现方法，比如BLAS、MKL等。

缺点：空间换时间，占用更多内存



im2col: 三维矩阵转二维矩阵

caffe im2col详解

<https://blog.csdn.net/zhanghenan123/article/details/81984829>

caffe源码深入学习6: 超级详细的im2col绘图解析, 分析caffe卷积操作的底层实现

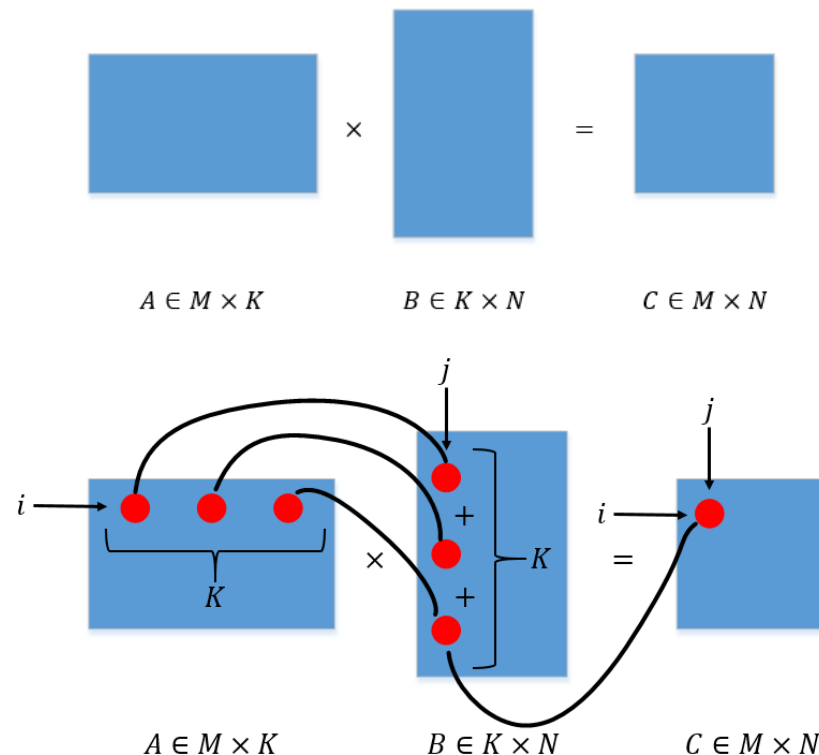
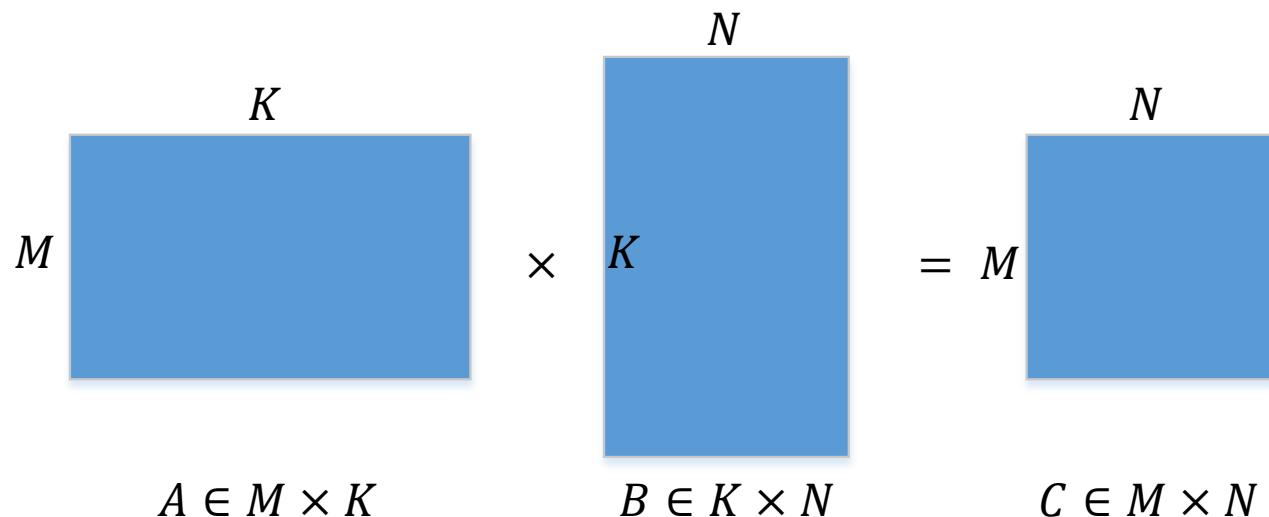
<https://blog.csdn.net/jiongnima/article/details/69736844>

0	0	0	0	0	0	0
0	1	2	3	4	5	0
0	6	7	8	9	10	0
0	11	12	13	14	15	0
0	16	17	18	19	20	0
0	21	22	23	24	25	0
0	0	0	0	0	0	0

$k = 3$
 $\xrightarrow{\quad}$
 $pad = 1$
 $stride = 2$

0	0	0	0	7	9	0	17	19
0	0	0	6	8	10	16	18	20
0	0	0	7	9	0	17	19	0
0	2	4	0	12	14	0	22	24
1	3	5	11	13	15	21	23	25
2	4	0	12	14	0	22	24	0
0	7	9	0	17	19	0	0	0
6	8	10	16	18	20	0	0	0
7	9	0	17	19	0	0	0	0

GEMM通用矩阵相乘:multi



$$c_{ij} = \sum_{k=0}^K a_{ik} b_{kj}$$

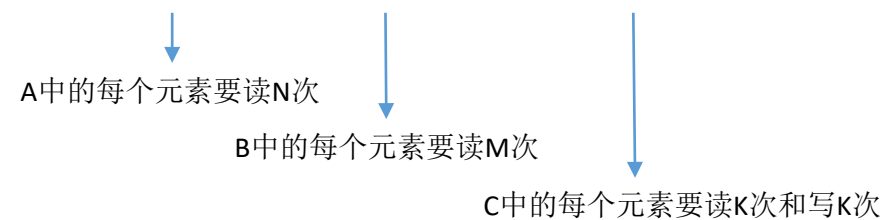
```

for (i = 0; i < M; i++) {
    for (j = 0; j < N; j++) {
        for (k = 0; k < K; k++) {
             $c_{ij} += a_{ik} \cdot b_{kj}$ 
        }
    }
}
    
```

简单multi:

总计算量 = $(1 + 1)MNK = 2MNK$

访存总数 = $(2 + 1 + 1)MNK = N(MK) + M(KN) + 2K(MN) = 4MNK$



GEMM通用矩阵相乘: multi+register

```
register temp = 0
for (i = 0; i < M; i++) {
    for (j = 0; j < N; j++) {
        temp = 0
        for (k = 0; k < K; k++) {
            temp +=  $a_{ik} \cdot b_{kj}$ 
        }
         $c_{ij}$  = temp
    }
}
```

multi+register:

总计算量 = $(1 + 1)MNK = 2MNK$

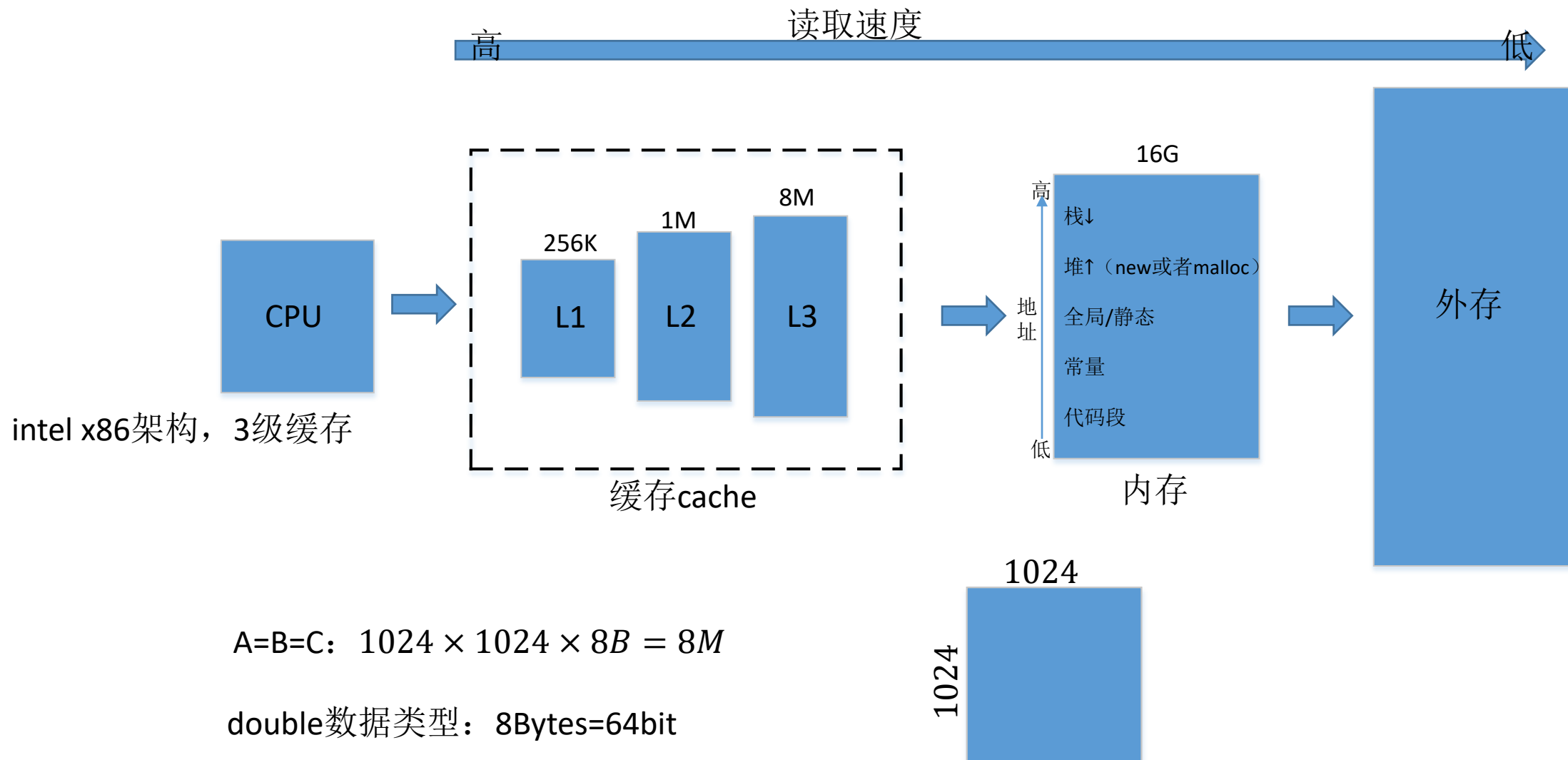
访存总数 = $(1 + 1)MNK + MN = N(MK) + M(KN) + MN = 2MNK + MN$

↓
A中的每个元素要读N次

↓
B中的每个元素要读M次

↓
C中的每个元素要写1次

CPU的缓存(cache)和内存



$$A=B=C: 1024 \times 1024 \times 8B = 8M$$

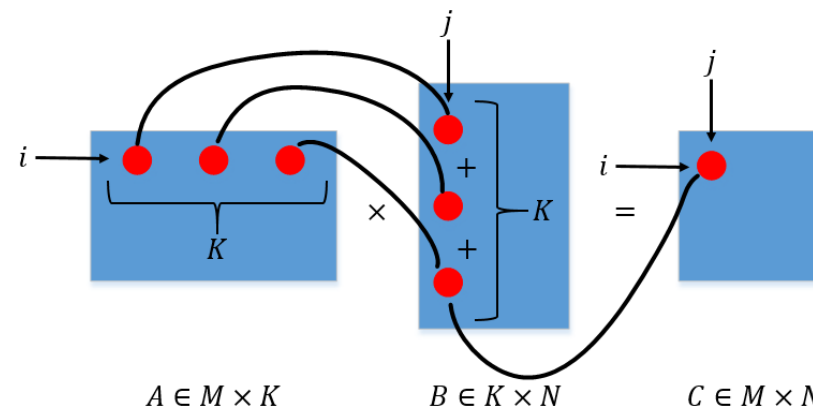
double数据类型: 8Bytes=64bit

GEMM通用矩阵相乘: multi+ikj

$$A[3][3] = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

行优先存储: {1 2 3 4 5 6 7 8 9} 比如, C++中的二维数组

列优先存储: {1 4 7 2 5 8 3 6 9}



```
for (i = 0; i < M; i++) {  
    for (j = 0; j < N; j++) {  
        for (k = 0; k < K; k++) {  
             $c_{ij} += a_{ik} \cdot b_{kj}$   
        }  
    }  
}
```

L2 利用率不高, L3

↓

B每读一次都要换行,
将数据从内存加载到cache

↓

cache的命中率不高

```
for (i = 0; i < M; i++) {  
    for (k = 0; k < K; k++) {  
        for (j = 0; j < N; j++) {  
             $c_{ij} += a_{ik} \cdot b_{kj}$   
        }  
    }  
}
```

L3 L2

↓

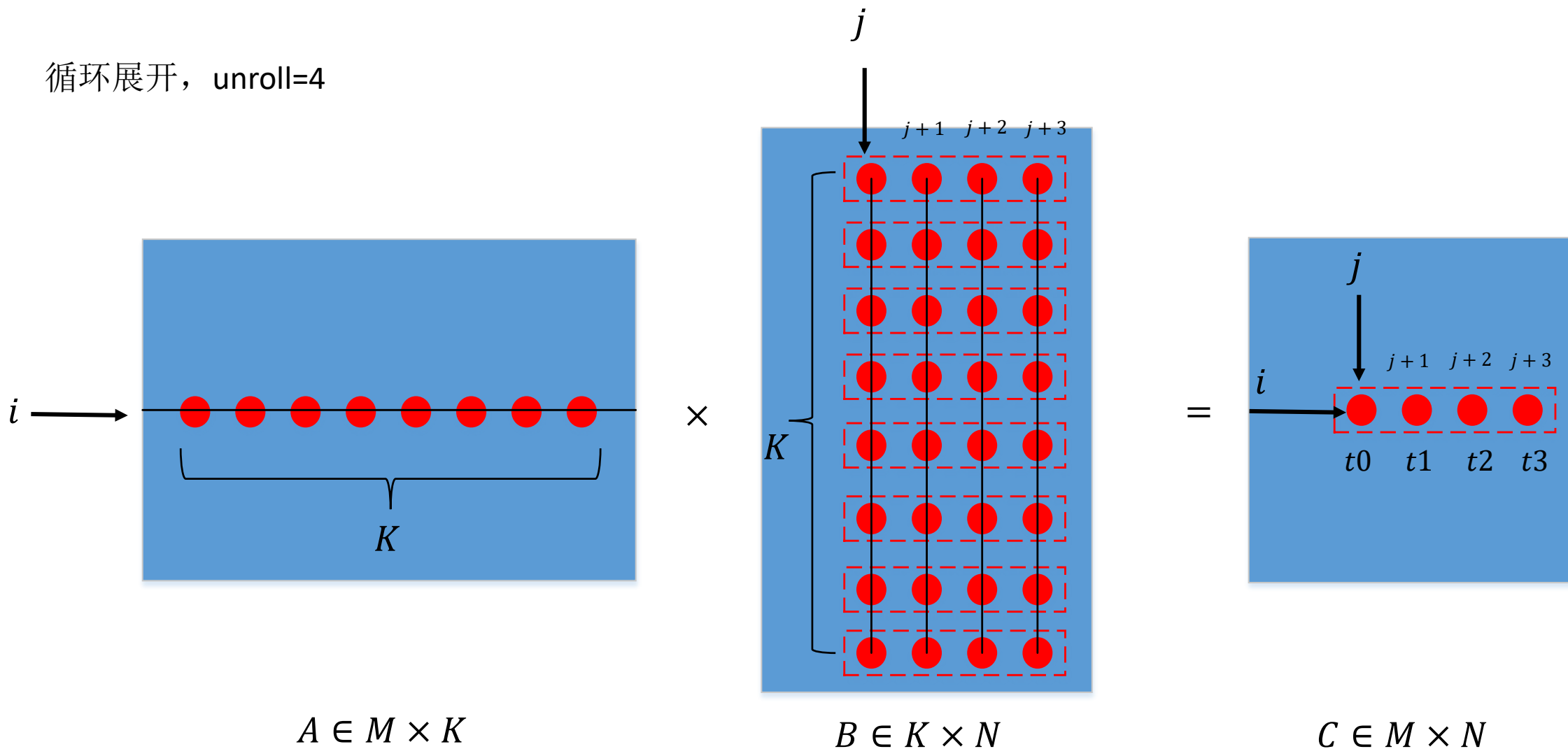
B不用频繁换行, L3

↓

cache的命中率较高

GEMM通用矩阵相乘: unroll 4

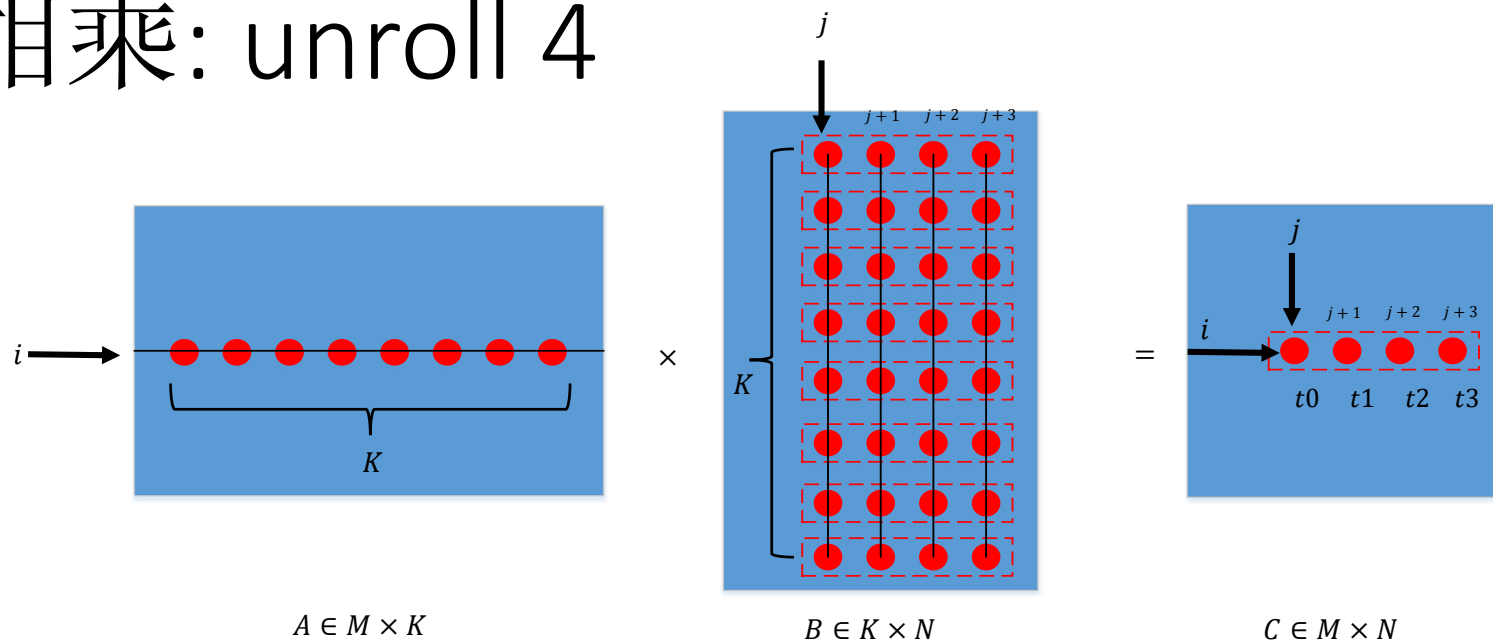
循环展开, unroll=4



GEMM通用矩阵相乘: unroll 4

```

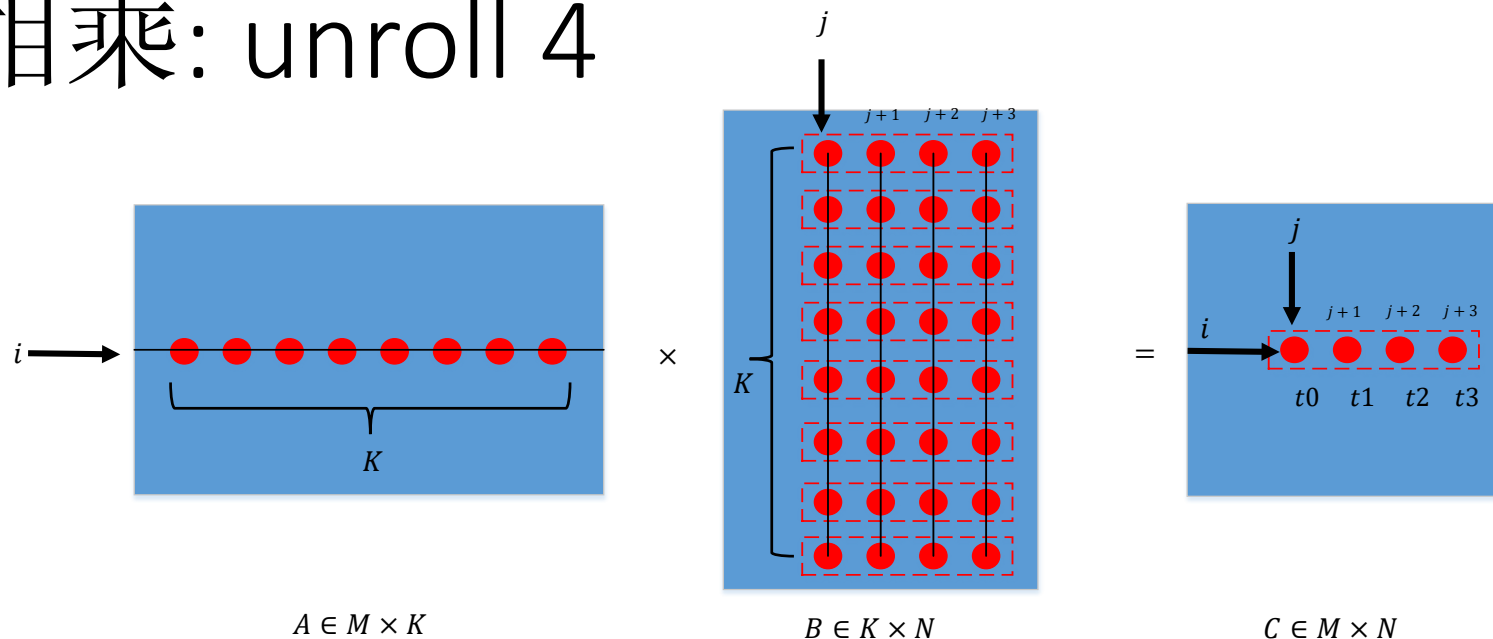
register t0, t1, t2, t3
for (i = 0; i < M; i++) {
    for (j = 0; j < N; j+=4) {
        t0=0, t1=0, t2=0, t3=0
        for (k = 0; k < K; k++) {
            t0 += aik · bkj
            t1 += aik · bk(j+1)
            t2 += aik · bk(j+2)
            t3 += aik · bk(j+3)
        }
        cij = t0
        ci(j+1) = t1
        ci(j+2) = t2
        ci(j+3) = t3
    }
}
    
```



GEMM通用矩阵相乘: unroll 4

```

register t0, t1, t2, t3, a0
for (i = 0; i < M; i++) {
    for (j = 0; j < N; j+=4) {
        t0=0, t1=0, t2=0, t3=0
        for (k = 0; k < K; k++) {
            a0 = aik
            t0 += a0 · bkj
            t1 += a0 · bk(j+1)
            t2 += a0 · bk(j+2)
            t3 += a0 · bk(j+3)
        }
        cij = t0
        ci(j+1) = t1
        ci(j+2) = t2
        ci(j+3) = t3
    }
}
    
```



优点：减少B矩阵频繁换行带来的损失（1次计算4个再换行）；
减少对A矩阵的访问次数（每个A元素复用4次）

unroll 1x4:

总计算量 = $(1 + 1)MNK = 2MNK$

访存总数 = $N/4(MK) + M(KN) + MN = 5/4MNK + MN$

A中的每个元素复用4次

B中的每个元素要读M次

C中的每个元素要写1次

SIMD技术及上层抽象实现

SIMD256: CPU同时对256bit的数据进行读写或者运算 (256 bit = 32 Bytes = 4 double = 8 float = 8 int)

烤面包: 单指令多数据SIMD技术



具体实现 (烤架): intel x86架构的AVX2.0指令集支持SIMD256



上层抽象实现: C++的immintrin.h封装了支持AVX2.0的api和数据类型

英特尔® Intrinsics Guide

<https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=AVX>



数据类型: `_m256d`

加载/存放api:

`_mm256_broadcast_sd`//加载64bit并填充

`_mm256_load_pd`//加载256bit

`_mm256_store_pd`//存放256bit

运算api:

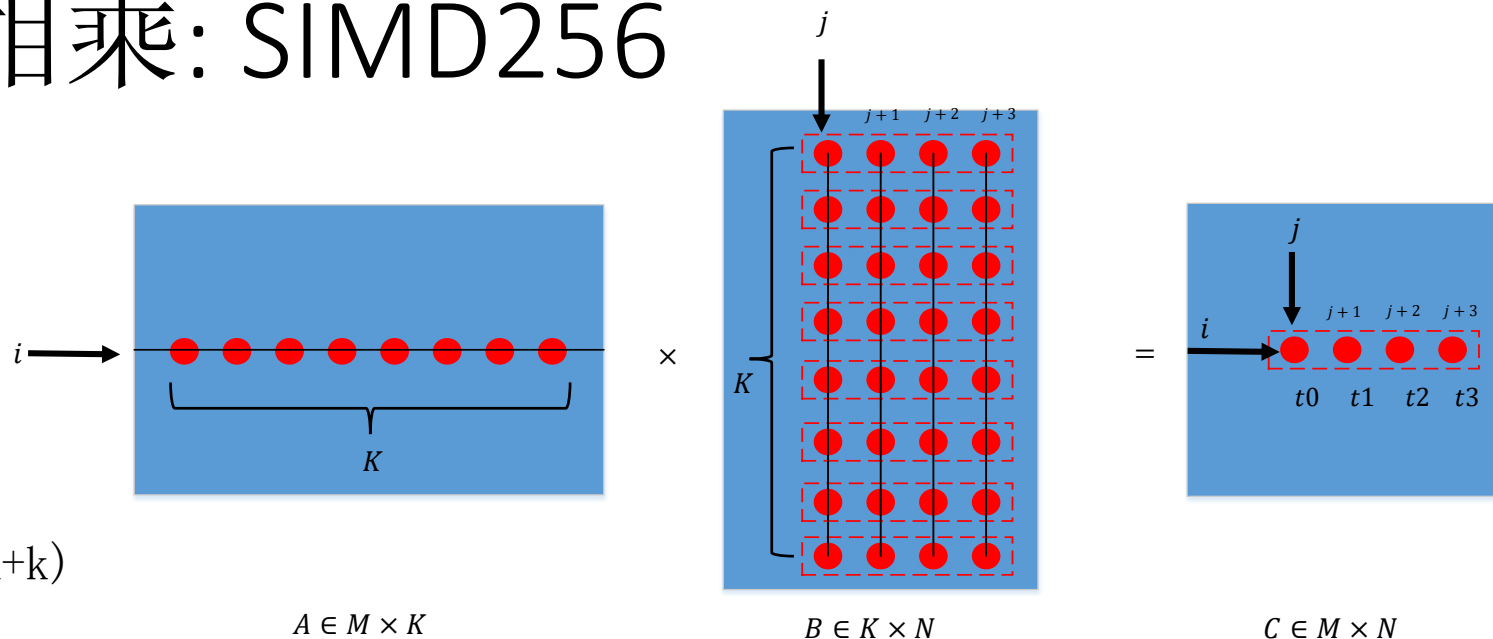
`_mm256_add_pd`//加

`_mm256_mul_pd`//乘

GEMM通用矩阵相乘: SIMD256

```

_m256d c0, a0, b0
for (i = 0; i < M; i++) {
    for (j = 0; j < N; j+=4) {
        c0=load(C+i*N+j)
        for (k = 0; k < K; k++) {
            a0=broacast(A+i*K+k)
            b0=load(B+k*N+j)
            c0=add(c0, mul(a0, b0))
        }
        store(C+i*N+j, c0)
    }
}
    
```



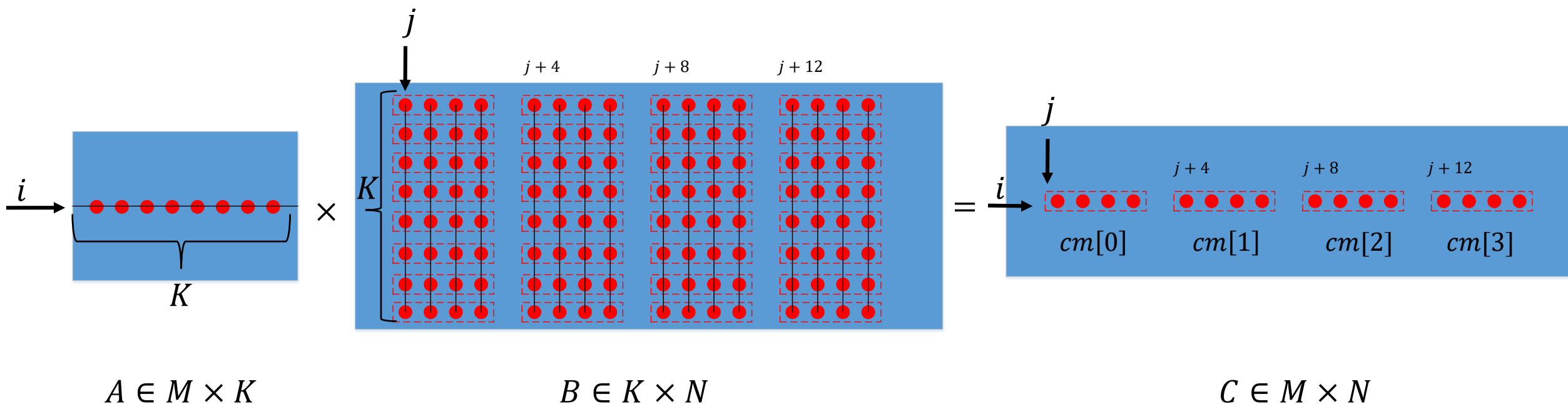
SIMD256:

总计算量 = $(1 + 1) \cdot M \cdot N \cdot K = 2MNK$

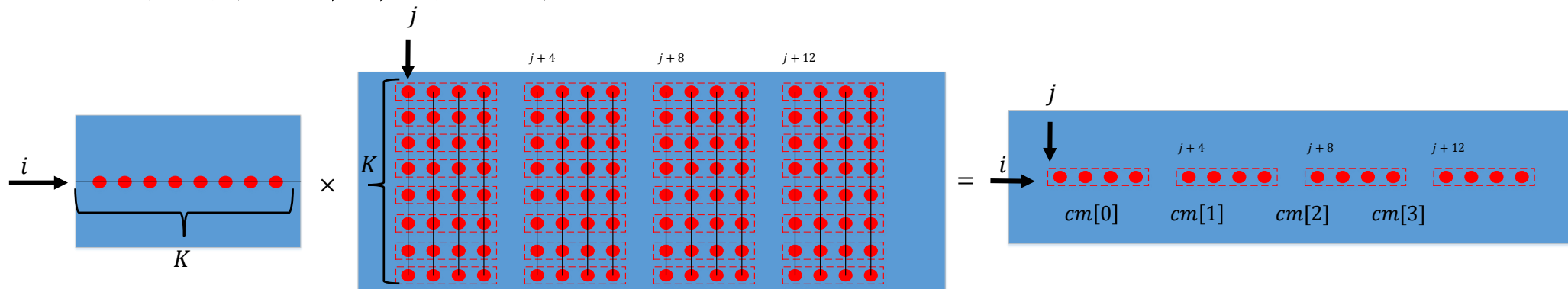
访存总数 = $\frac{N}{4}(MK) + M \left(K \frac{N}{4} \right) + 2 \left(M \frac{N}{4} \right) = \frac{1}{2}MNK + \frac{1}{2}MN$

↓
 A中的每个元素复用4次
 ↓
 B中的最小元素变为1x4
 ↓
 C中的每个元素要读1次和写1次，最小元素为1x4

GEMM通用矩阵相乘: SIMD256+unroll 4



GEMM通用矩阵相乘: SIMD256+unroll 4



```

_m256d cm[4], a0, b0
for (i = 0; i < M; i++) {
    for (j = 0; j < N; j+=4*4) {
        for (x = 0; x < 4; x++) {
            cm[x]=load(C+i*N+j+x*4)
        }
        for (k = 0; k < K; k++) {
            a0=broadcast(A+i*K+k)
            cm[0]+=mul(a0, load(B+k*N+j))
            cm[1]+=mul(a0, load(B+k*N+j+4))
            cm[2]+=mul(a0, load(B+k*N+j+8))
            cm[3]+=mul(a0, load(B+k*N+j+12))
        }
        for (x = 0; x < 4; x++) {
            store(C+i*N+j+x*4, cm[x])
        }
    }
}

```

$A \in M \times K$

$B \in K \times N$

$C \in M \times N$

SIMD256+unroll4:

总计算量 = $(1 + 1) \cdot M \cdot N \cdot K = 2MNK$

访存总数 = $\frac{1}{4} \frac{N}{4} (MK) + M \left(K \frac{N}{4} \right) + 2 \left(M \frac{N}{4} \right) = \frac{5}{16} MNK + \frac{1}{2} MN$

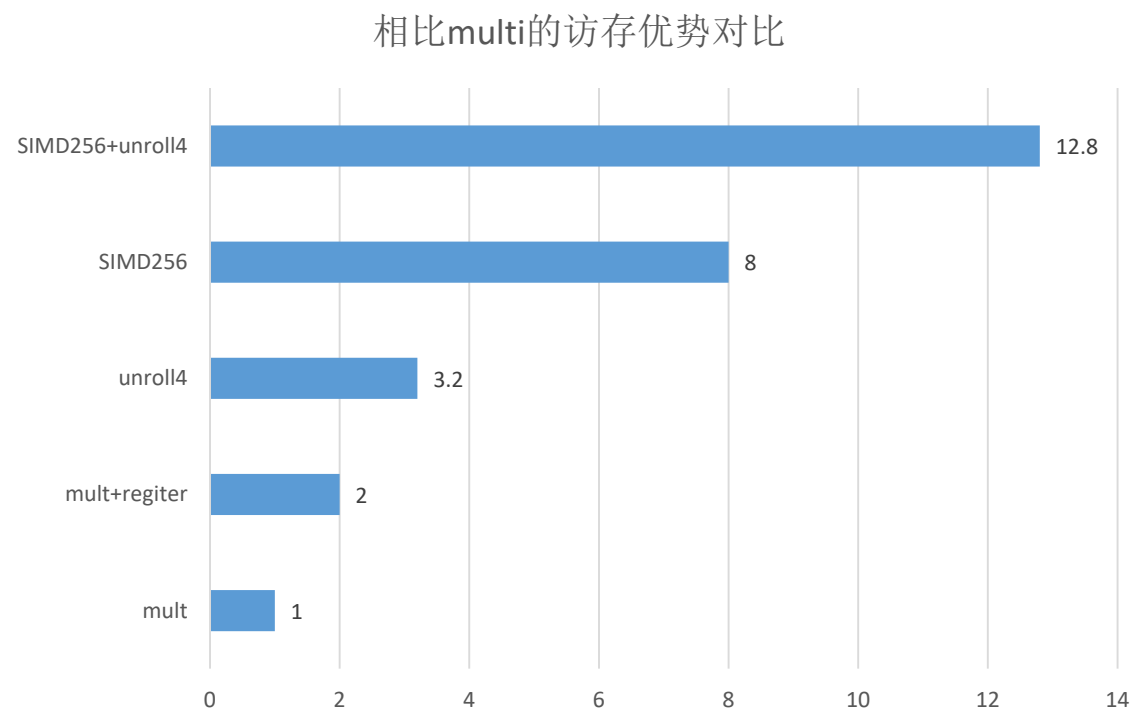
A中的每个元素复用16次

B中的最小元素变为1x4

C中的每个元素要读1次和写1次，最小元素为1x4

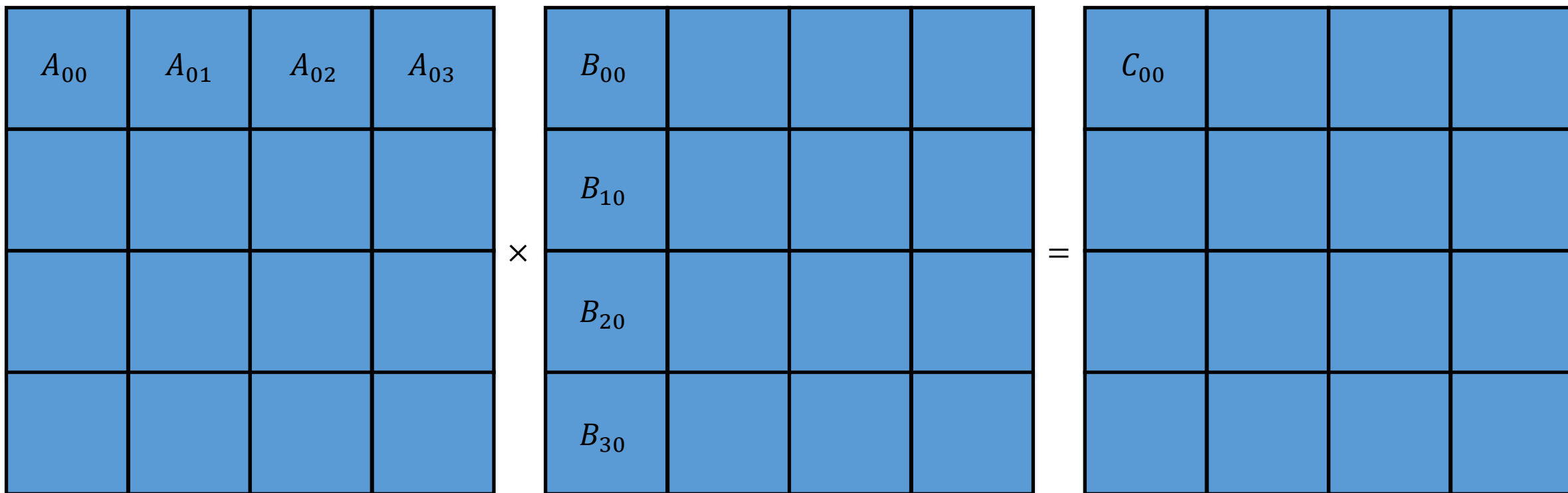
GEMM通用矩阵相乘: 优化方法对比

	multi	multi+register	unroll4	SIMD256	SIMD256+unroll4
访存总数	4MNK	2MNK+MN	5/4MNK+MN	1/2MNK+1/2MN	5/16MNK+1/2MN



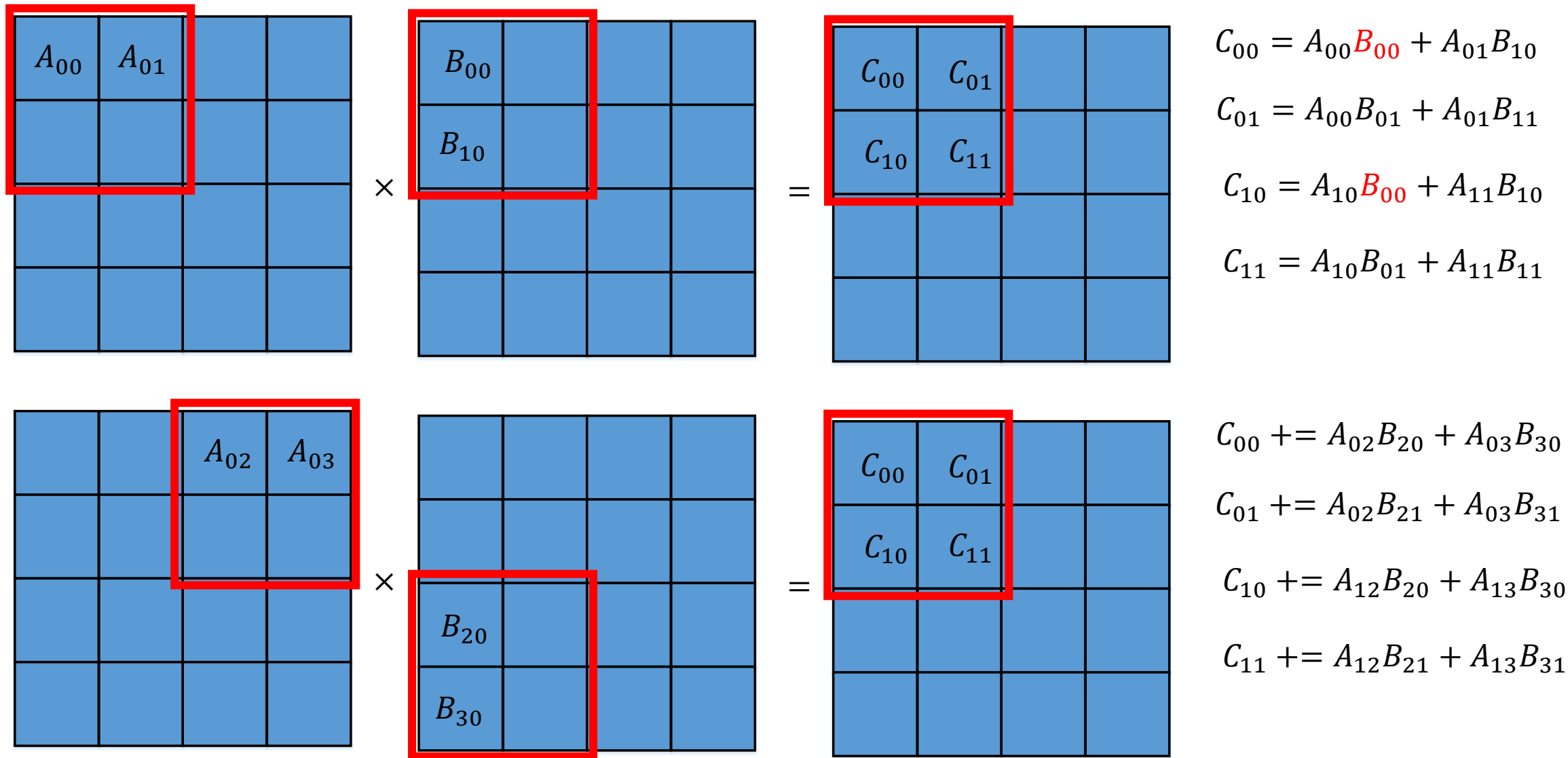
GEMM通用矩阵相乘: cache blocking

$$C_{00} = A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20} + A_{03}B_{30}$$

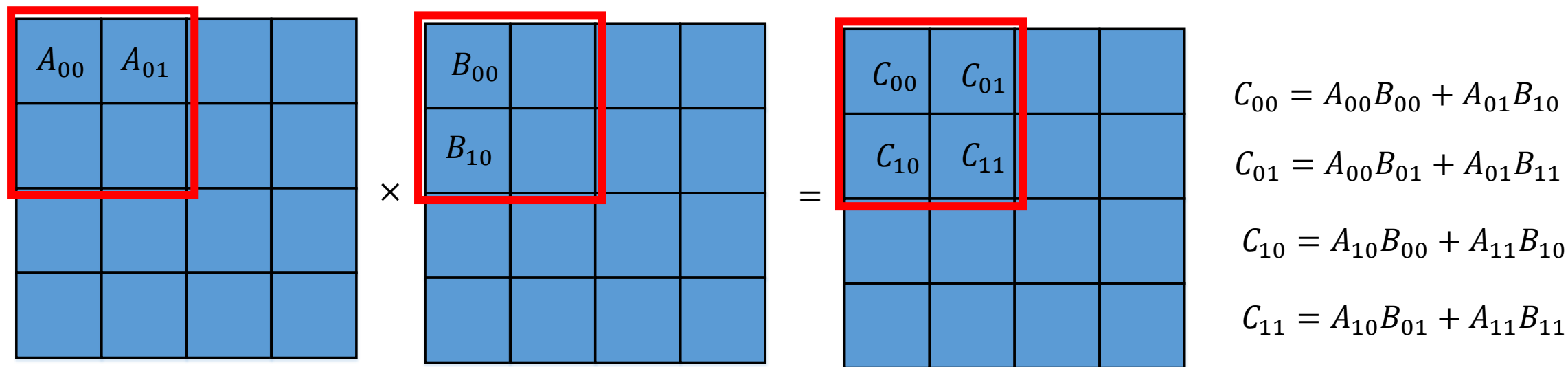


- B矩阵的每个block的每个单元（比如 B_{00} ）复用2次，提高cache命中率
- 矩阵B的读取不用频繁将数据在内存和缓存之间转移，提高cache命中率

GEMM通用矩阵相乘: cache blocking



GEMM通用矩阵相乘: cache blocking+openMP



根据block以及block中的每个单元的计算独立性，

利用多线程技术对每个单元进行独立计算（比如 C_{00} 、 C_{01} 、 C_{10} 、 C_{11} ）

GEMM通用矩阵相乘： 优化方法比较

VS -> 分析 -> 性能探测器 -> 性能向导 -> 检测

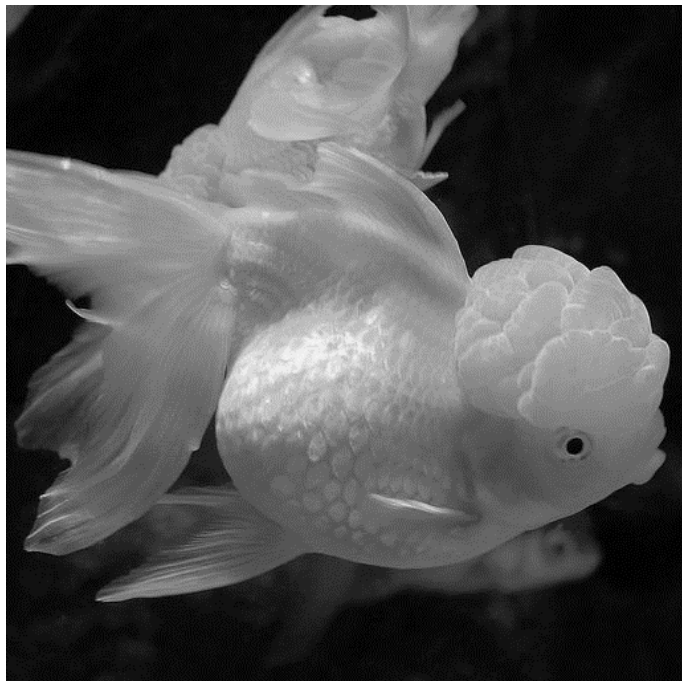
调用 main 的函数						
函数名	调用数	已用非独占时间百分比	已用独占时间百分比	平均已用非独占时间	平均已用独占时间	模块名
_srt_common_main_seh	1	97.76	0.04	15,841.22	7.03	测试SIMD和GEMM.exe
当前函数						
main	1	97.76	0.04	15,841.22	7.03	测试SIMD和GEMM.exe
由 main 调用的函数						
Matrix::multi	1	35.51	35.49	5,754.59	5,751.31	测试SIMD和GEMM.exe
Matrix::multi_register	1	30.94	30.92	5,013.63	5,010.41	测试SIMD和GEMM.exe
Matrix::multi_loopunrolling	1	11.98	11.96	1,941.39	1,938.34	测试SIMD和GEMM.exe
Matrix::multi_avx	1	7.17	7.15	1,161.74	1,158.73	测试SIMD和GEMM.exe
Matrix::multi_midk	1	5.42	5.40	878.19	874.61	测试SIMD和GEMM.exe
Matrix::multi_avx_unrollx4	1	4.84	4.82	784.73	781.73	测试SIMD和GEMM.exe
Matrix::multi_avx_unrollx4_blk	1	1.42	1.40	229.66	226.65	测试SIMD和GEMM.exe
Matrix::multi_avx_unrollx4_blk_omp	1	0.39	0.03	62.51	4.69	测试SIMD和GEMM.exe

采用行优先的
A:1024*1024
B:1024*1024
C:1024*1024

/O2 /arch:AVX2 /openmp

	multi	register	unroll4	SIMD256	multi+ikj	SIMD256+unroll4	SIMD256+cache block	SIMD256+cache block+omp
非独占时间(ms)	5754.59	5013.63	1941.39	1161.74	878.19	784.73	229.66	62.51
加速比		1.14	2.96	4.95	6.55	7.33	25.05	92.06

高斯模糊的三种实现方法

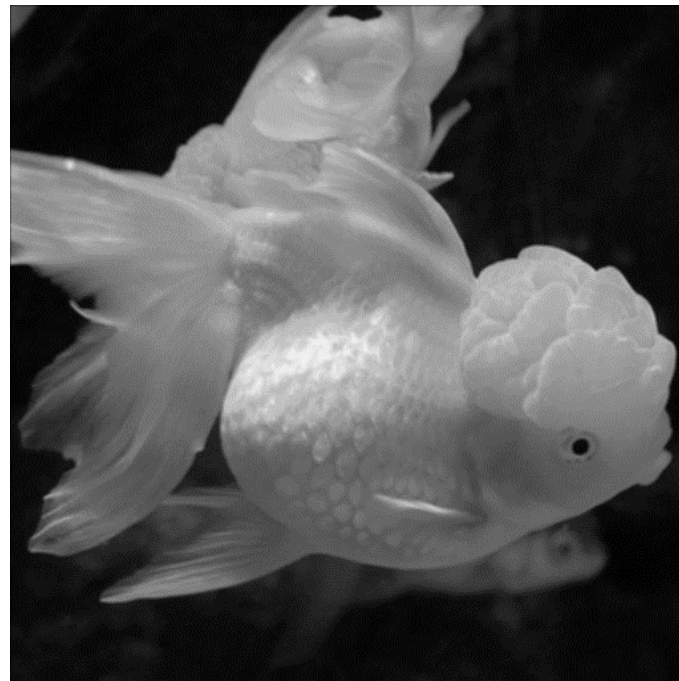


3x3 高斯核

0.1019	0.1154	0.1019
0.1154	0.1308	0.1154
0.1019	0.1154	0.1019



SAME, stride=1



- 普通卷积
- 基于FFT的快速卷积
- 基于GEMM的快速卷积

快速卷积方法的性能比较

srcIm:							
8	7	9	143	147	20	6	14
5	7	168	189	145	17	15	16
153	141	174	158	137	18	16	13
7	128	139	133	169	165	174	154
70	104	119	207	167	134	145	144
15	85	34	98	104	78	87	106
20	25	87	45	91	98	62	3
15	23	11	24	8	13	12	10

输入图像 8*8

3x3 高斯核($\sigma = 2.0$)

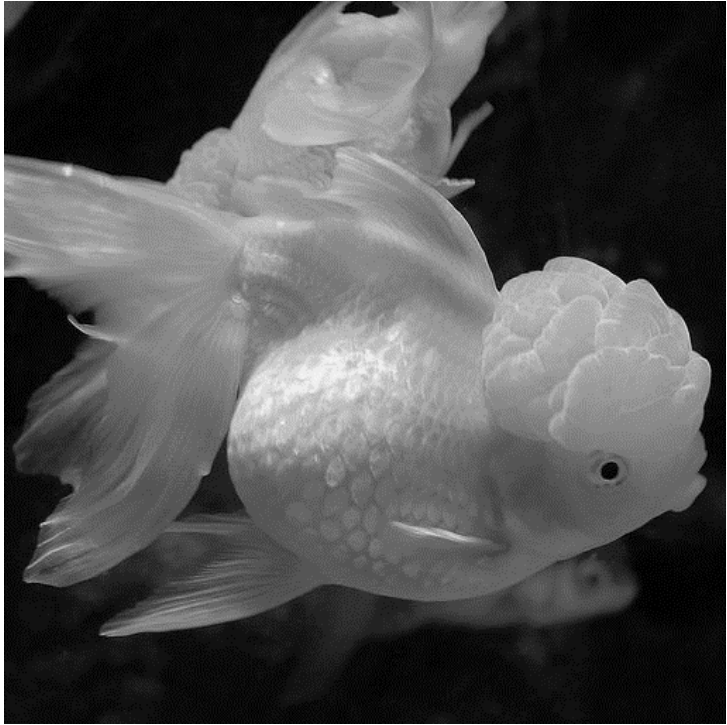
```
myGaussianBlur_nature:
3.14467 21.3091 57.8506 90.4132 75.7655 38.5383 9.8026 5.89873
35.123 73.0118 111.462 143.169 110.061 56.2507 14.9322 9.18205
51.4258 104.268 139.265 156.995 125.823 92.2637 63.3453 42.4238
66.3898 114.44 144.27 155.897 144.261 126.09 109.644 74.752
45.3984 79.8711 116.664 132.187 141.004 135.989 132.423 92.1727
35.3035 61.8179 88.1619 105.104 113.005 106.399 95.1215 61.9626
19.9665 35.7279 48.084 56.5088 63.0359 62.478 52.2827 31.0242
9.47228 19.7953 24.0374 28.6594 30.3888 30.9072 21.67 9.35531
```

```
myGaussianBlur_GEMM:
3.14467 21.3091 57.8506 90.4132 75.7655 38.5383 9.8026 5.89873
35.123 73.0118 111.462 143.169 110.061 56.2507 14.9322 9.18205
51.4258 104.268 139.265 156.995 125.823 92.2637 63.3453 42.4238
66.3898 114.44 144.27 155.897 144.261 126.09 109.644 74.752
45.3984 79.8711 116.664 132.187 141.004 135.989 132.423 92.1727
35.3035 61.8179 88.1619 105.104 113.005 106.399 95.1215 61.9626
19.9665 35.7279 48.084 56.5088 63.0359 62.478 52.2827 31.0242
9.47228 19.7953 24.0374 28.6594 30.3888 30.9072 21.67 9.35531
```

```
GaussianBlur:
3.145 21.309 57.851 90.413 75.766 38.538 9.803 5.899
35.123 73.012 111.462 143.169 110.061 56.251 14.932 9.182
51.426 104.268 139.265 156.995 125.823 92.264 63.345 42.424
66.390 114.440 144.270 155.897 144.261 126.090 109.644 74.752
45.398 79.871 116.664 132.187 141.004 135.989 132.423 92.173
35.304 61.818 88.162 105.104 113.005 106.399 95.121 61.963
19.967 35.728 48.084 56.509 63.036 62.478 52.283 31.024
9.472 19.795 24.037 28.659 30.389 30.907 21.670 9.355
```

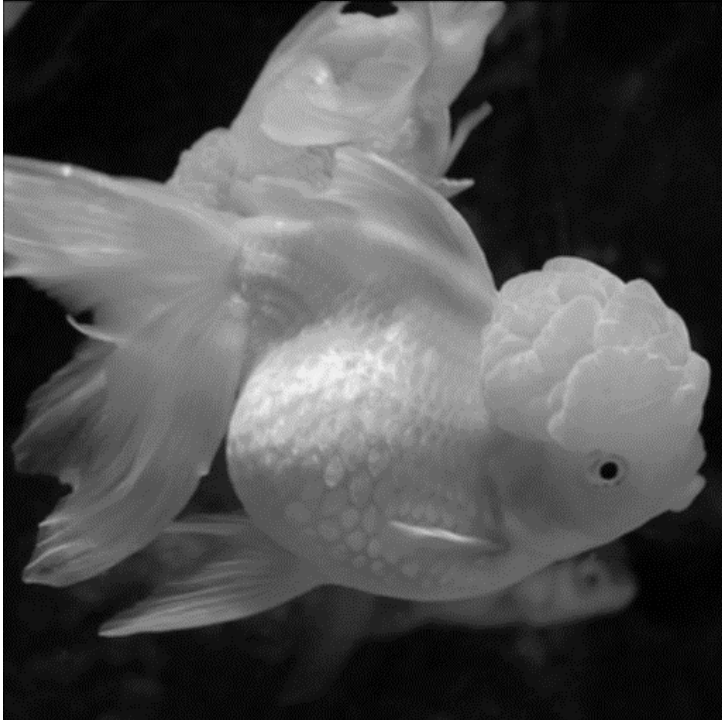
```
myGaussianBlur_FFT:
19.03 57.19 97.81 118.31 106.06 66.62 23.78 4.55
63.31 107.41 153.44 175.34 159.99 115.27 67.93 46.65
116.39 159.55 206.87 231.76 219.72 176.80 128.45 103.31
148.59 183.55 227.12 255.00 251.13 216.66 172.14 143.74
139.20 163.84 201.24 230.68 235.32 211.27 172.93 142.86
91.94 111.07 144.26 173.40 182.03 163.73 129.24 99.22
36.08 57.57 90.15 116.52 121.82 101.12 66.29 39.05
6.78 36.13 71.30 93.57 89.78 60.14 22.05 -0.00
```

原图



3x3 高斯核($\sigma = 2.0$)

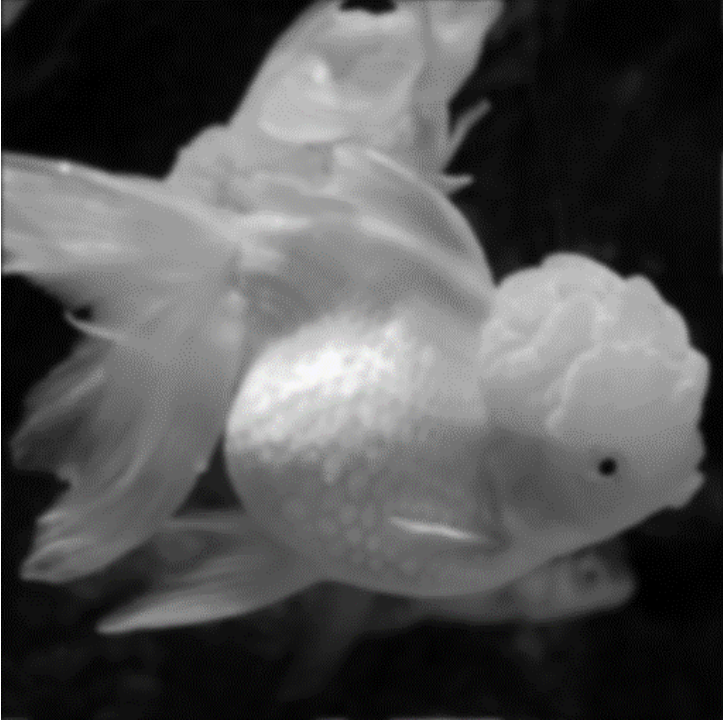
GEMM



普通卷积



FFT



cv::GaussianBlur



快速卷积方法的性能比较

测试高斯模糊的三种实现...01107(8).vsp × testFFT.cpp testFFT.h MyMatrix.h MyMatrix.cpp 源.cpp

← → 当前视图: 调用方/被调用方(C)

调用 main 的函数

函数名	调用数	已用非独占时间百分比	已用独占时间百分比	平均已用非独占时间	平均已用独占时间	模块名
_sclr_common_main_seh	1	98.90	6.03	5.19	0.32	测试高斯模糊的三种实现方法

当前函数

main	1	98.90	6.03	5.19	0.32	测试高斯模糊的三种实现方法
------	---	-------	------	------	------	---------------

由 main 调用的函数

cv::imread	1	55.78	55.78	2.93	2.93	opencv_imgcodecs340.dll
myGaussianBlur_FFT	1	11.76	0.25	0.62	0.01	测试高斯模糊的三种实现方法
cv::GaussianBlur	1	7.41	7.41	0.39	0.39	opencv_imgproc340.dll
Matrix::conv_nature	1	5.56	4.43	0.29	0.23	测试高斯模糊的三种实现方法
myGaussianBlur_GEMM	1	3.54	0.00	0.19	0.00	测试高斯模糊的三种实现方法

输入图像 128*128
3x3 高斯核($\sigma = 2.0$)
/O2 /arch:AVX2 /openmp

	基于FFT的快卷	cv::GaussianBlur	普通卷积	SIMD256+unroll8 基于GEMM的快卷
非独占时间(ms)	0.62	0.39	0.29	0.19
加速比	0.46	0.74		1.52

快速卷积方法的性能比较



函数名	调用数	已用非独占时间百分比	已用独占时间百分比	平均已用非独占时间	平均已用独占时间	模块名
调用 main 的函数						
_scrt_common_main_seh	1	99.32	20.60	9.53	1.98	测试高斯模糊的三种实现方法
当前函数						
main	1	99.32	20.60	9.53	1.98	测试高斯模糊的三种实现方法
由 main 调用的函数						
cv::imread	1	31.06	31.06	2.98	2.98	opencv_imgcodecs340.dll
Matrix::conv_nature	1	13.47	12.75	1.29	1.22	测试高斯模糊的三种实现方法
myGaussianBlur_GEMM	1	12.91	0.01	1.24	0.00	测试高斯模糊的三种实现方法
myGaussianBlur_FFT	1	6.92	0.18	0.66	0.02	测试高斯模糊的三种实现方法
cv::GaussianBlur	1	4.65	4.65	0.45	0.45	opencv_imgproc340.dll

输入图像 128*128











7x7 高斯核($\sigma = 2.0$)

/O2 /arch:AVX2 /openmp

	普通卷积	基于GEMM的快卷	基于FFT的快卷	cv::GaussianBlur
非独占时间(ms)	1.29	1.24	0.66	0.45
加速比		1.04	1.95	2.87

欢迎批评指正

<https://github.com/LeonJinC/Fast-Convolution-with-SIMD-and-GEMM>

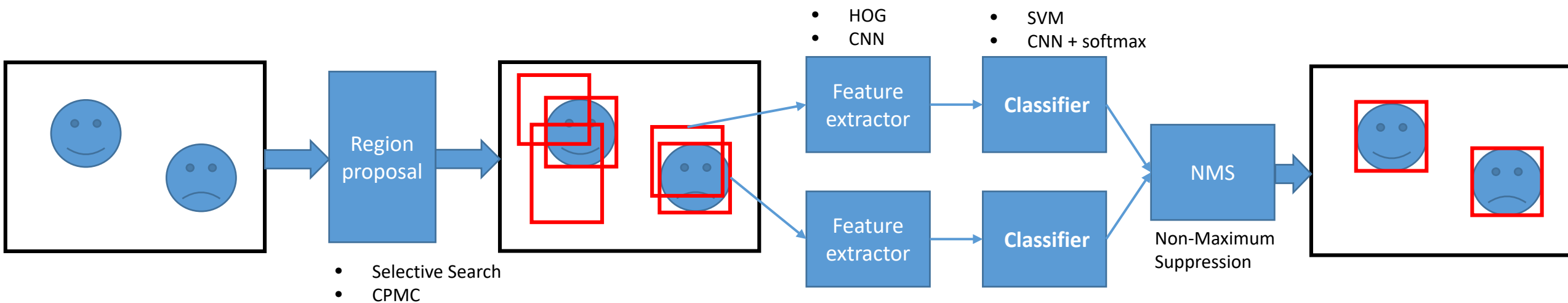
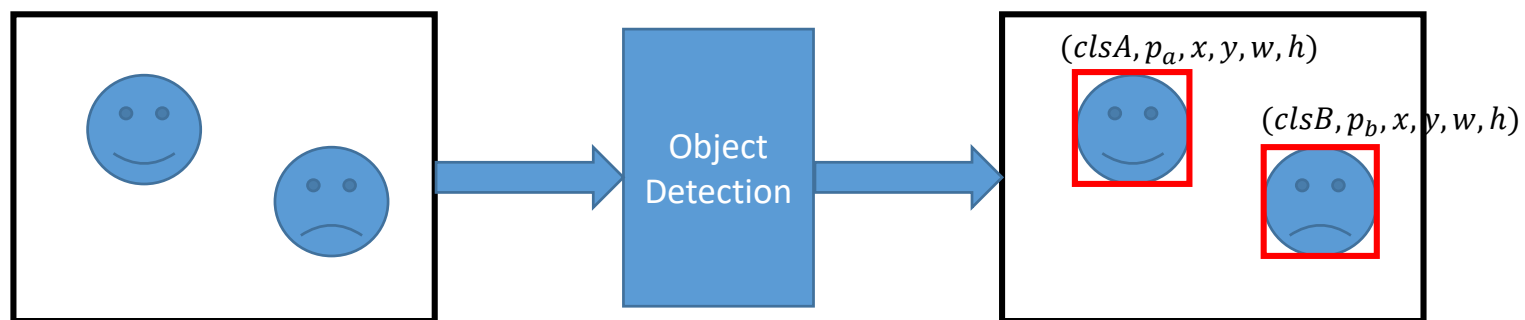
名称	类型
 Colfirst_Matrix_and_GEMM_SIMD	文件夹
 Rowfirst_Matrix_and_GEMM_SIMD	文件夹
 Three_methods_of_Gaussian_Blur_based_on_Fast_Convolution	文件夹
 README	MD 文件
 RESULTS0GEMM	JPG 文件
 RESULTS1Fast_Convolution3	JPG 文件
 RESULTS2Fast_Convolution7	JPG 文件
 RESULTS3Image	JPG 文件
 RESULTS4Matrix	JPG 文件
 结合SIMD和GEMM的快速卷积及其在高斯模糊中的应用	Microsoft PowerPoint 演示文稿

【下回预讲】

目标检测框架：two stage算法

改进方法：

- 边框回归
- CNN特征提取
- 全连接层SVD分解
- RPN+ROI pooling
- 多尺度+anchor机制



- Region proposal生成方法 <https://www.cnblogs.com/alexanderkun/p/6128058.html>
- RCNN论文 <https://arxiv.org/pdf/1311.2524.pdf>