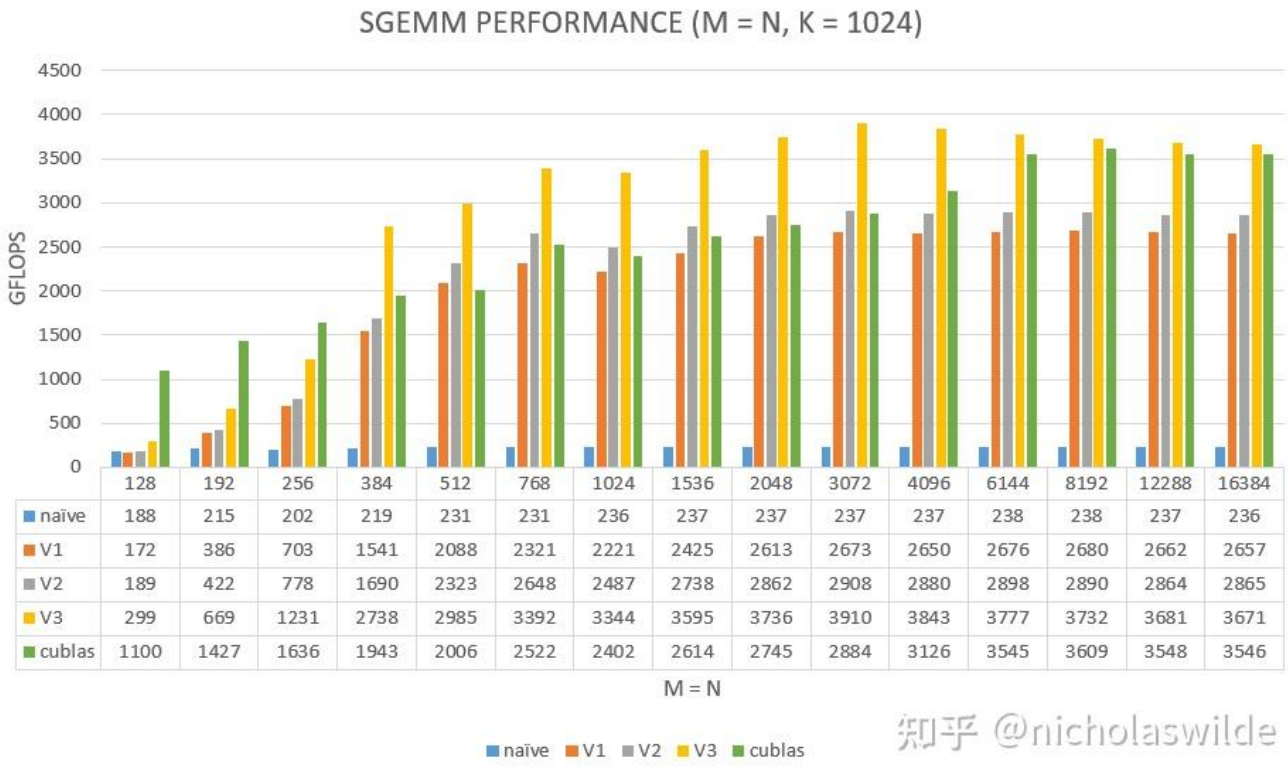
CUDA SGEMM矩阵乘法优化笔记——从入门到cublas

1 Introduction

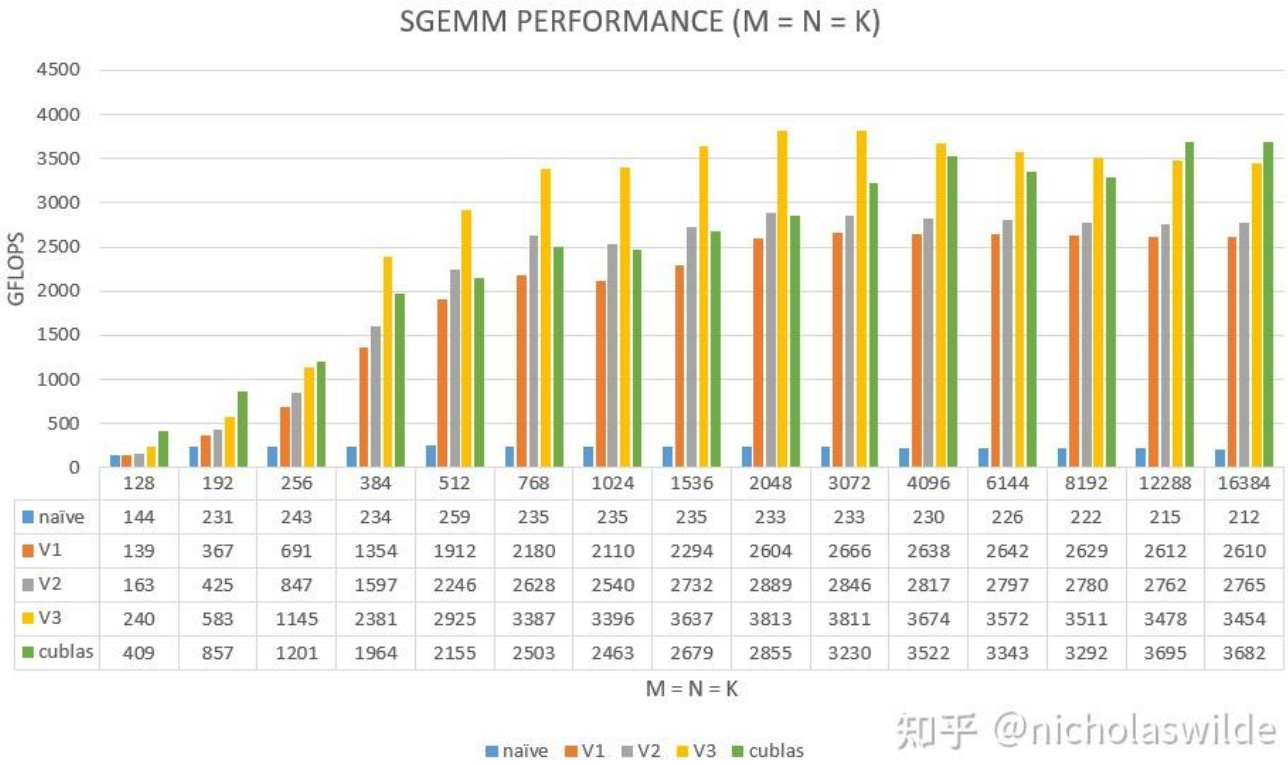
最近开始入门CUDA，初步了解GPU的工作原理后，选择了单精度矩阵乘法作为练习的kernal，尝试从最简单的SGEMM kernal开始，逐步优化到cublas的性能水平。

下面的两张图是在自己的笔记本上（古老的GTX1060 Mobile 6GB，我也想要A100啊啊啊）的测试结果，naive SGEMM kernal性能非常低，在200 GFLOPS左右，而最终优化版本SGEMM V4最高可达到3900 GFLOPS左右，基本达到或超过了cublas的性能。代码上传在 https://github.com/nicolaswilde/cuda-sgemm。



	128	192	256	384	512	768	1024	1536	2048	3072	4096	6144	8192	12288	16384
naïve	188	215	202	219	231	231	236	237	237	237	237	238	238	237	236
V1	172	386	703	1541	2088	2321	2221	2425	2613	2673	2650	2676	2680	2662	2657
V2	189	422	778	1690	2323	2648	2487	2738	2862	2908	2880	2898	2890	2864	2865
V3	299	669	1231	2738	2985	3392	3344	3595	3736	3910	3843	3777	3732	3681	3671
cublas	1100	1427	1636	1943	2006	2522	2402	2614	2745	2884	3126	3545	3609	3548	3546

Performance M = N, K = 1024



	128	192	256	384	512	768	1024	1536	2048	3072	4096	6144	8192	12288	16384
naïve	144	231	243	234	259	235	235	235	233	233	230	226	222	215	212
V1	139	367	691	1354	1912	2180	2110	2294	2604	2666	2638	2642	2629	2612	2610
V2	163	425	847	1597	2246	2628	2540	2732	2889	2846	2817	2797	2780	2762	2765
V3	240	583	1145	2381	2925	3387	3396	3637	3813	3811	3674	3572	3511	3478	3454
cublas	409	857	1201	1964	2155	2503	2463	2679	2855	3230	3522	3343	3292	3695	3682

Performance M = N = K

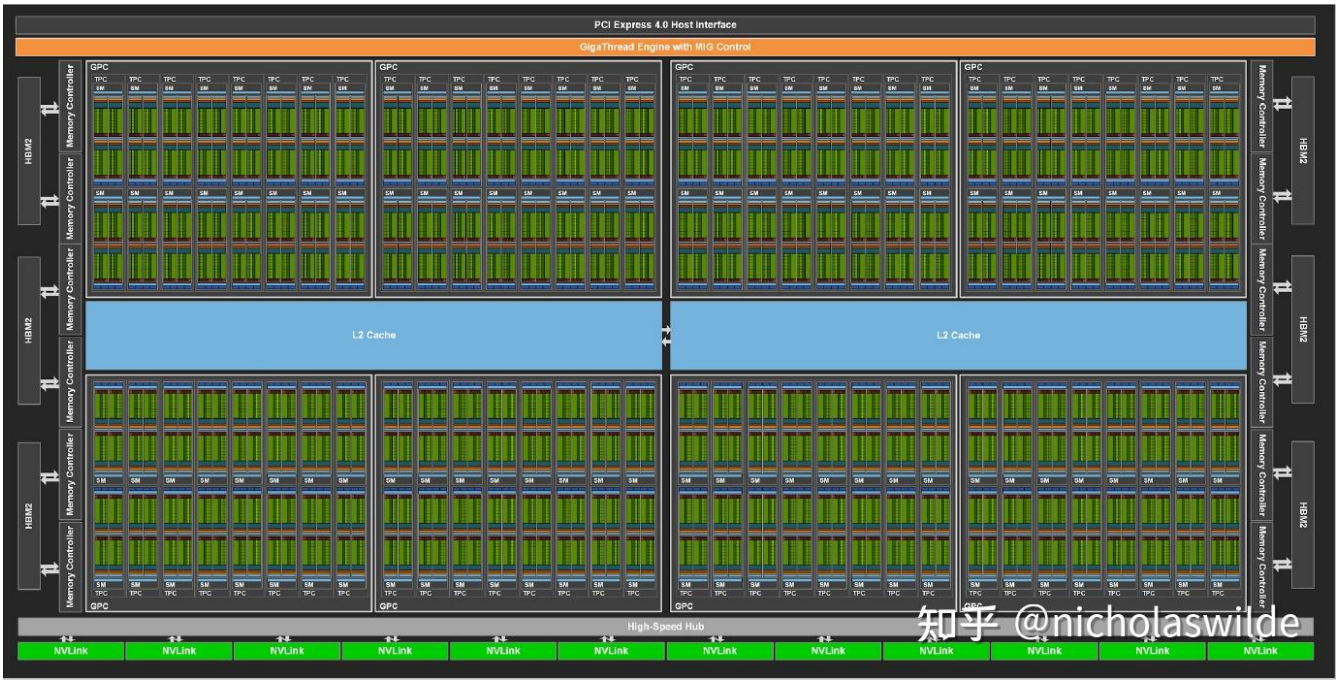
作为初学者，站在巨人的肩膀上尤为重要，我参考了CUDA 矩阵乘法终极优化指南和深入浅出GPU优化系列：GEMM优化两篇文章学习了CUDA SGEMM kernal的优化方法。

2 Background

在编写CUDA代码之前，首先介绍一点GPU和CUDA的背景知识，有CUDA基础的同学可以跳过，只学过C语言的同学们我们一起来了解一下GPU。

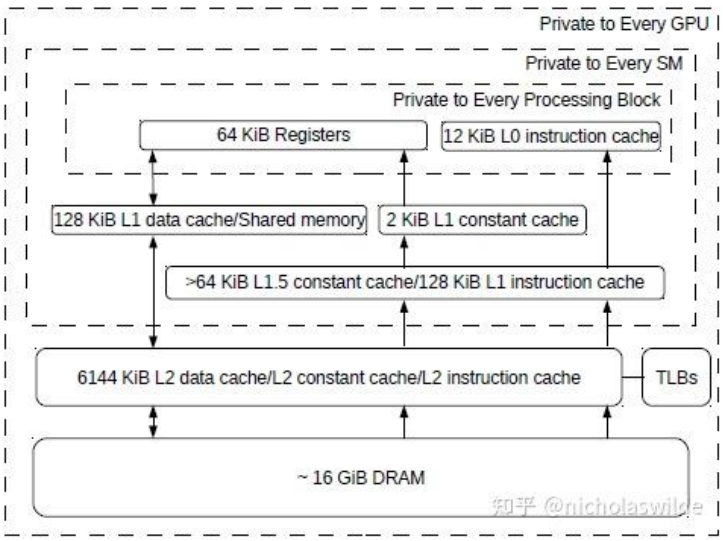
2.1 GPU Architecture

GPU是一种SIMT (Single Instruction Multiple Threads)式的处理器，以其众核结构并行处理成千上万个线程。以英伟达次新的A100为例，下图是NVIDIA A100 Whitepaper中提供的完整的GA100的结构图，A100是GA100阉割了1/8的版本。一块A100一共划分为8个GPC (GPU Processing Cluster)，每个GPC划分为7个TPC (Texture Processing Cluster)，每个TPC包含2个SM (Streaming Multiprocessor)。



NVIDIA GA100 GPU

GPU的存储层次如下图所示（图中数据是V100的数据，但是存储层次都是一样的），所有SM共享L2 Cache，拥有私有的L1 Cache，私有的Shared Memory和私有的寄存器堆。在GPU的内存层次中，越往上访问延迟越低，越往下访问延迟越高。其中DRAM和各级Cache与CPU的存储层次类似；同样是SM私有的Shared Memory和Register的区别是，Register是线程私有的，而Shared Memory则是线程共享的，Register单拍即可访问，Shared Memory访问延迟大概需要二三十拍。



GPU Memory Hierarchy

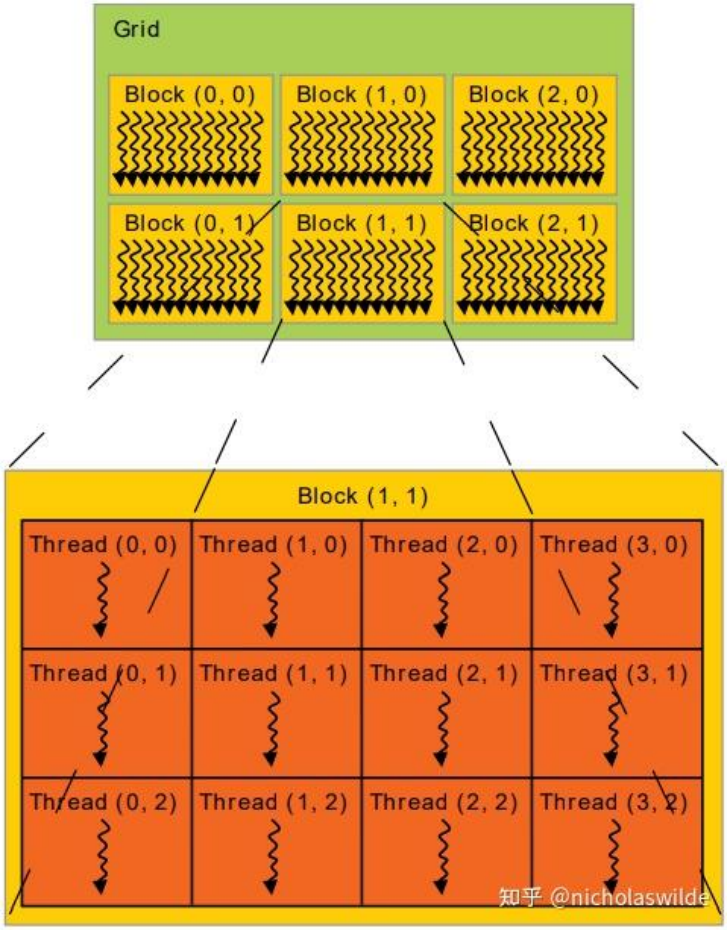
A100的SM如下图所示，划分为四个SM Block，每个SM Block都有自己的寄存器堆和运算部件，每个SM Block包含16个INT32部件、16个FP32部件以及8个FP64部件，它们通常被称为CUDA Core；而右侧的Tensor Core专门用于矩阵乘法运算，我实验用的GPU是Pascal架构的GTX 1060，只有CUDA Core没有Tensor Core，事实上Volta架构的GPU才开始引入Tensor Core。

NVIDIA A100 SM

2.2 CUDA编程模型

了解了GPU的大致结构，现在我们介绍CUDA的编程模型，并将编程模型对应到GPU上去。

CUDA的编程模型分为三级：Grid、Block、Thread，如下图所示。每一个CUDA Kernal对应一个Grid，Grid可以包含一维、二维或三维的多个Block，Block可以包含一维、二维或三维的多个Thread。



Grid of Thread Blocks

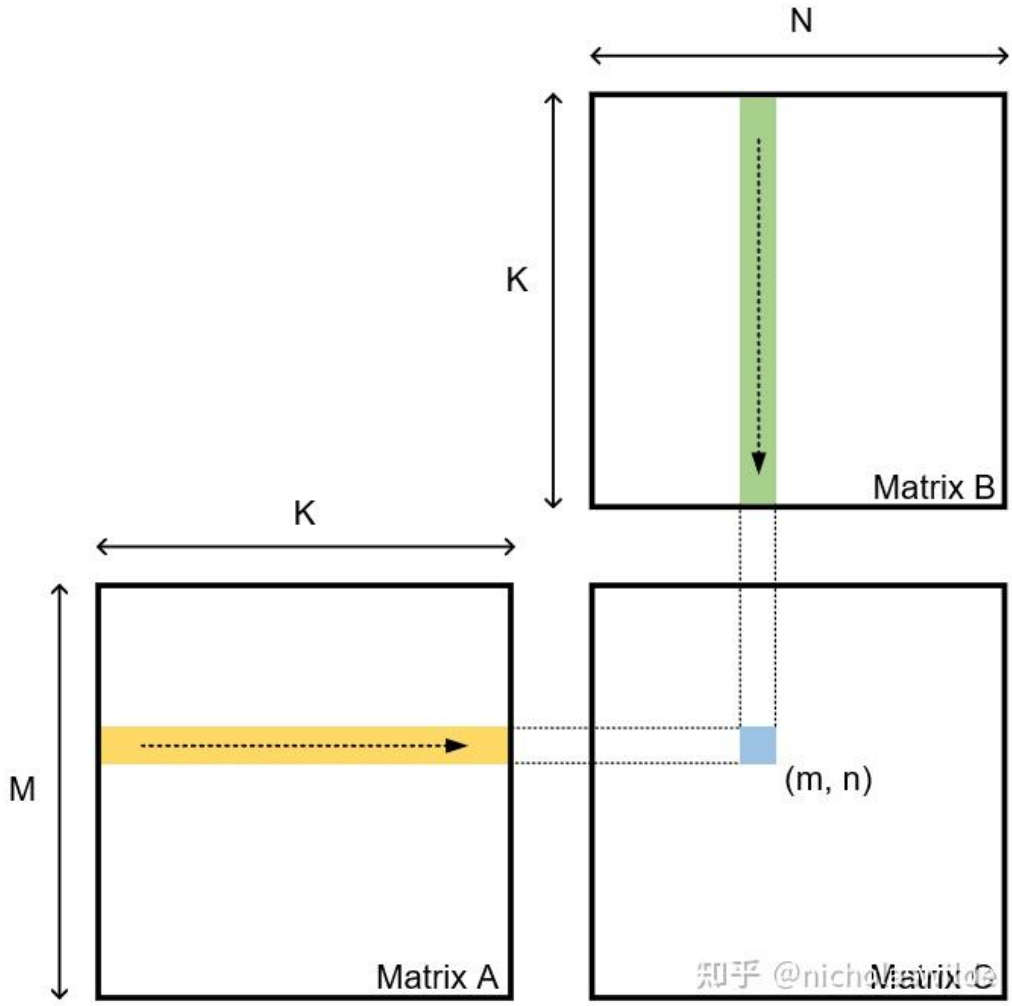
一个Block能且只能在一个SM中运行，一个SM可以同时运行多个Block；在SM中，一个Block的线程按照32的粒度分组，每32个线程称为一个Warp，Warp是SM调度运行的最小单位，每个时钟周期，SM Block都会尝试选择一个可以调度的Warp发射执行。

Grid在全局内存Global Memory中共享数据，Block在共享内存Shared Memory中共享数据；SM中的寄存器是线程私有的，一个线程最多可以分配到256个寄存器，线程间的寄存器交互比较困难，因此相同Block线程间需要共享的数据通常从Global Memory预取到Shared Memory中，线程需要时再load到Register中使用。

3 Naive SGEMM

SGEMM是单精度矩阵乘法，计算C = A * B，其中A是M * K的矩阵，B是K * N的矩阵，C是M * N的矩阵，数据类型是Float32。

我们从最简单的矩阵乘法的Kernal写起，一共使用M * N个线程完成整个矩阵乘法。每个线程负责矩阵C中一个元素的计算，需要完成K次乘累加：



Naive SGEMM

```
#define OFFSET(row, col, ld) ((row) * (ld) + (col))
__global__ void naiveSgemm(
    float * __restrict__ a, float * __restrict__ b, float * __restrict__ c,
    const int M, const int N, const int K) {

    int n = blockIdx.x * blockDim.x + threadIdx.x;
    int m = blockIdx.y * blockDim.y + threadIdx.y;
    if (m < M && n < N) {
        float psum = 0.0;
        #pragma unroll
        for (int k = 0; k < K; k++) {
            psum += a[OFFSET(m, k, K)] * b[OFFSET(k, n, N)];
        }
        c[OFFSET(m, n, N)] = psum;
    }
}
```

使用下面的代码对SGEMM kernal进行性能测试，这里naiveSgemm选取BM = 32, BN = 32，也就是一个Block有1024个线程；gridDim为((N + BN - 1) / BN, (M + BM - 1) / BM)，blockDim为(BN, BM)，当然啦测试结果比较惨淡，只能达到200 GFLOPS左右，只有cublas 6%左右的样子：

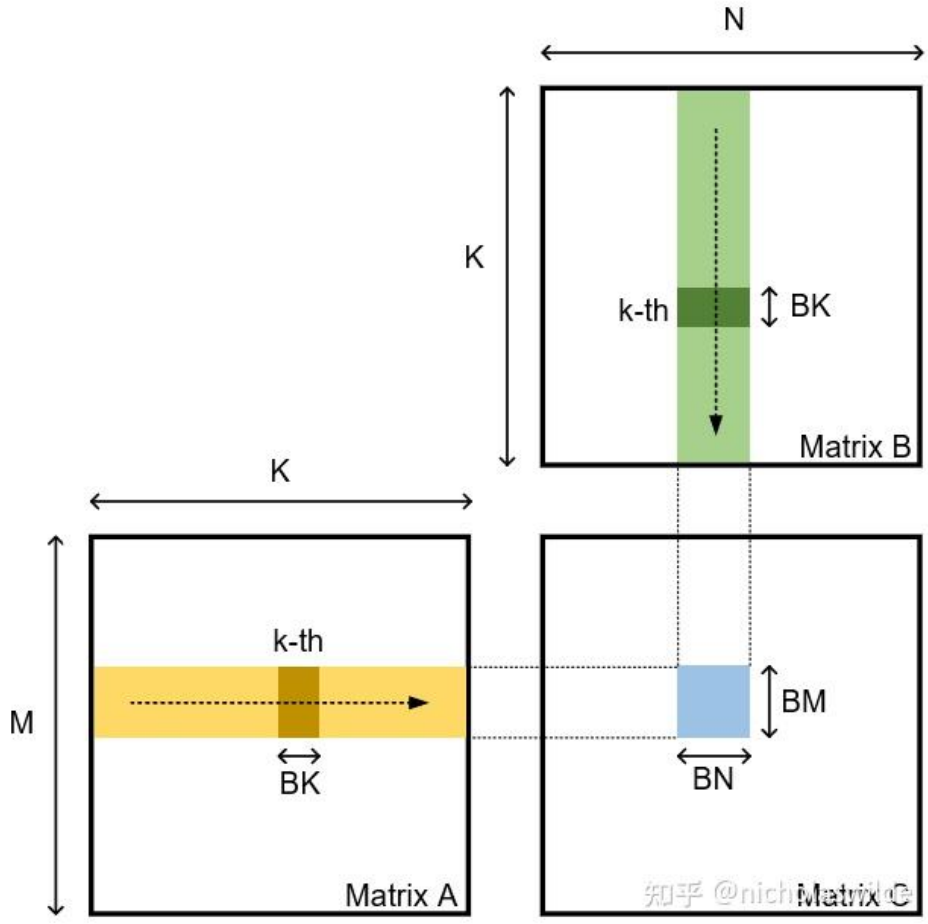
```
float testPerformance(
    void (*gpuSgemm) (float *, float *, float *, const int, const int, const int),
    dim3 gridDim, dim3 blockDim, const int M, const int N, const int K, const int repeat) {
```

```
    size_t size_a = M * K * sizeof(float);
    size_t size_b = K * N * sizeof(float);
    size_t size_c = M * N * sizeof(float);

    float *d_a, *d_b, *d_c;
    cudaMalloc(&d_a, size_a);
    cudaMalloc(&d_b, size_b);
    cudaMalloc(&d_c, size_c);

    cudaEvent_t start, end;
    cudaEventCreate(&start);
    cudaEventCreate(&end);
    cudaEventRecord(start);
    for (int i = 0; i < repeat; i++)
        gpuSgemm<<<gridDim, blockDim>>>(d_a, d_b, d_c, M, N, K);
    cudaEventRecord(end);
    cudaEventSynchronize(end);

    float msec, sec;
    cudaEventElapsedTime(&msec, start, end);
    sec = msec / 1000.0 / repeat;

    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    return sec;
}
```

4 矩阵分块

上面这种每个线程计算一个元素简单的SGEMM kernal显然是效率非常低的，需要两次Global Memory的load才能完成一次乘累加运算，计算访存比极低，没有有效的数据复用。

那么，我们接下来尝试这样一种方法：每个Block负责计算矩阵C中的BM * BN个元素，每个Thread负责计算矩阵C中的TM * TN个元素；使用Shared Memory来存放线程间可以复用的矩阵A和矩阵B的元素，一个Block的所有线程每次一起从Global Memory中load BM * BK个矩阵A的元素和BK * BN个矩阵B的元素，存放到Shared Memory中，之后每个线程再分别从Shared Memory中取数运算，这样的循环一共需要K / BK次以完成整个矩阵乘法操作，如下图所示。



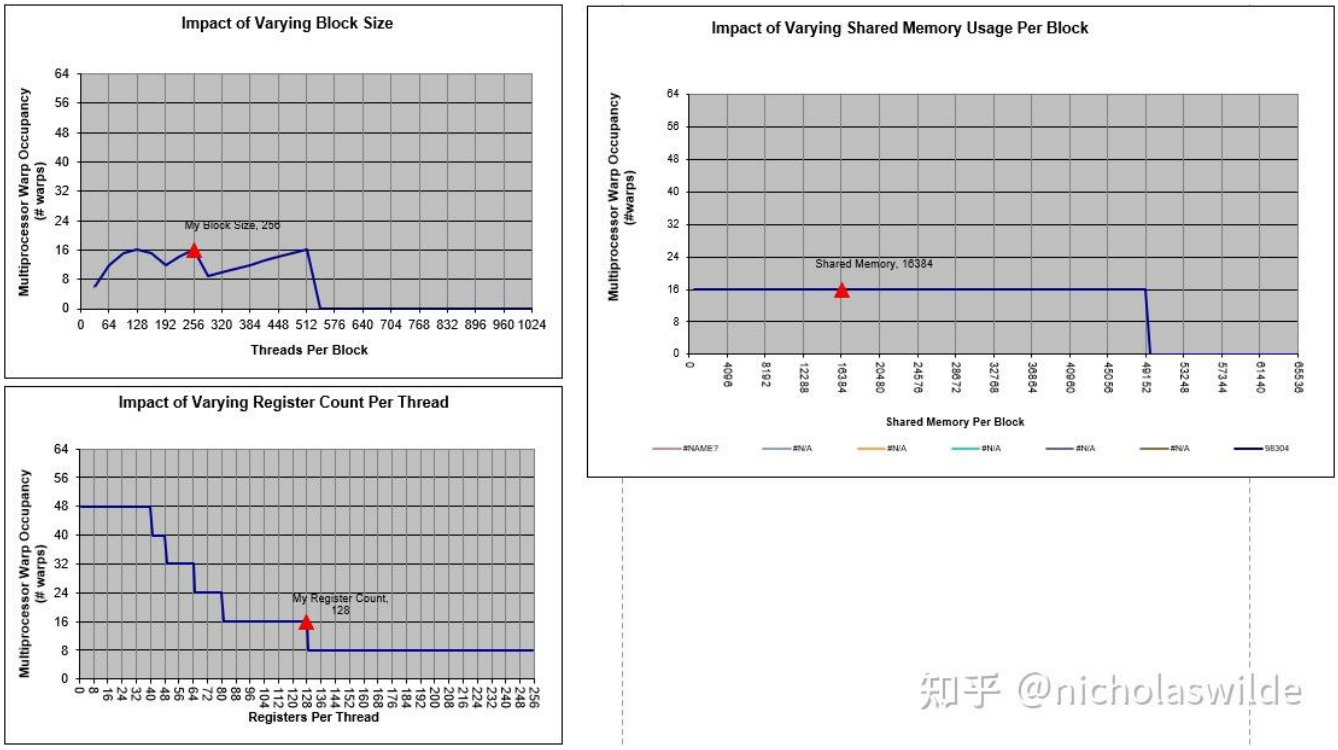
SGEMM Block Tiling

这种运算方法可以极大地增加数据复用，减少Global Memory的访存。只计算对矩阵A和矩阵B的访问，naiveSgemm一共需要M * N * 2K = 2MNK次load，那么这种矩阵分块的方法只需要M / BM * N / BN * K / BK * (BM * BK + BN * BK) = (1 / BN + 1 / BM) * MNK次load，BM和BN的取值越大，访问Global Memory的次数越少。

这里有几个参数需要确定：BM、BN、BK、TM、TN。理论上来说BM和BN越大，访问Global Memory的次数就越少，所以BM和BN越大越好，BK则可以小一些；但是BM和BN的大小还受到Shared Memory的限制，一个Block需要占用BK * (BM + BN) * 4 Bytes大小的Shared Memory，如果采用double buffering的话还要再多一倍，我的GTX1060一个SM只有96 KB的Shared Memory。TM和TN的取值也受到两方面限制，一方面是线程数的限制，一个Block中有BM / TM * BN / TN个线程，这个数字不能超过1024，且不能太高防止影响SM内Block间的并行；另一方面是寄存器数目的限制，一个线程至少需要TM * TN个寄存器用于存放矩阵C的部分和，再加上一些其它的寄存器，所有的寄存器数目不能超过256，且不能太高防止影响SM内同时并行的线程数目。

上面提到的每个Block占用的Shared Memory数目、每个Block中的线程数目、以及每个线程占用的寄存器数目，这三者和你GPU的硬件配置共同决定了所谓的Occupancy，也就是一个SM中可以同时并行运行多少Block/Warp/Thread，在某些Warp阻塞时可以调度其它Warp发射执行，这样一来就可以利用并行性来提高硬件利用率。这里我们选取BM = BN = 128, BK = 8, TM = TN = 8, CUDA 矩阵乘法终极优化指南和深入浅出GPU优化系列：GEMM优化两篇文章都采用了这样的参数，这个参数可以说是相当舒服了；这样每个Block有256个线程，每个线程限制最多需要128个寄存器（存放矩阵C的部分和需要64个，其它寄存器不能超过64个，否则寄存器总数超过128就会影响Occupancy），每个Block（采用double buffering）需要16384 Bytes的Shared

Memory，我们将这三个数据输入到CUDA_Occupancy_Calculator中就可以看到计算得到的Occupancy，同时有16个Warp并行：



CUDA Occupancy

大体思路确定了，直接实现一个简单版本出来，为了方便起见假设M、N、K可以整除BM(128)、BN(128)、BK(8)，也就是没有考虑边缘情况的判断。

这份代码里需要关注的点包括：

- 每个Block一次从Global Memory中读1024个A的元素和1024个B的元素，每个Block有256个线程，也就是说每个线程一次只需要读4个A的元素和4个B的元素，在给每个线程分配需要load的元素时，尽可能让threadIdx.x方向相邻的线程load连续的地址，以达到合并访存的目的
- 使用float4类型访存，用向量化的LDG.128和STG.128指令一次读4个元素，以减少指令数
- 从Global Memory取数时没有使用double buffering，因此主循环体中每一次循环都是先从Global Memory中load到Shared Memory中，使用__syncthreads()将Block中的所有线程同步，然后再计算，注意计算完后需要再一次__syncthreads()，防止有的线程还没算完，而其它线程已经执行到下一次循环并修改了Shared Memory中对应的数据

```
#define OFFSET(row, col, ld) ((row) * (ld) + (col))
#define FLOAT4(pointer) (reinterpret_cast<float4*>(&(pointer))[0])
__global__ void mySgemmV1Aligned(
    float * __restrict__ a, float * __restrict__ b, float * __restrict__ c,
    const int M, const int N, const int K) {

    const int BM = 128;
    const int BN = 128;
    const int BK = 8;
    const int TM = 8;
```

```
    const int TN = 8;

    const int bx = blockIdx.x;
    const int by = blockIdx.y;
    const int tx = threadIdx.x;
    const int ty = threadIdx.y;
    const int tid = ty * blockDim.x + tx;

    __shared__ float s_a[BM][BK];
    __shared__ float s_b[BK][BN];

    float r_c[TM][TN] = {0.0};

    int load_a_smem_m = tid >> 1;
    int load_a_smem_k = (tid & 1) << 2;
    int load_b_smem_k = tid >> 5;
    int load_b_smem_n = (tid & 31) << 2;

    int load_a_gmem_m = by * BM + load_a_smem_m;
    int load_b_gmem_n = bx * BN + load_b_smem_n;

    for (int bk = 0; bk < (K + BK - 1) / BK; bk++) {
        int load_a_gmem_k = bk * BK + load_a_smem_k;
        int load_a_gmem_addr = OFFSET(load_a_gmem_m, load_a_gmem_k, K);
        FLOAT4(s_a[load_a_smem_m][load_a_smem_k]) = FLOAT4(a[load_a_gmem_addr]);
        int load_b_gmem_k = bk * BK + load_b_smem_k;
        int load_b_gmem_addr = OFFSET(load_b_gmem_k, load_b_gmem_n, N);
        FLOAT4(s_b[load_b_smem_k][load_b_smem_n]) = FLOAT4(b[load_b_gmem_addr]);

        __syncthreads();

        #pragma unroll
        for (int k = 0; k < BK; k++) {
            #pragma unroll
            for (int m = 0; m < TM; m++) {
                #pragma unroll
                for (int n = 0; n < TN; n++) {
                    int comp_a_smem_m = ty * TM + m;
                    int comp_b_smem_n = tx * TN + n;
                    r_c[m][n] += s_a[comp_a_smem_m][k] * s_b[k][comp_b_smem_n];
                }
            }
        }

        __syncthreads();
    }

    #pragma unroll
    for (int i = 0; i < TM; i++) {
        int store_c_gmem_m = by * BM + ty * TM + i;
        #pragma unroll
        for (int j = 0; j < TN; j += 4) {
            int store_c_gmem_n = bx * BN + tx * TN + j;
            int store_c_gmem_addr = OFFSET(store_c_gmem_m, store_c_gmem_n, N);
            FLOAT4(c[store_c_gmem_addr]) = FLOAT4(r_c[i][j]);
        }
    }
}
```

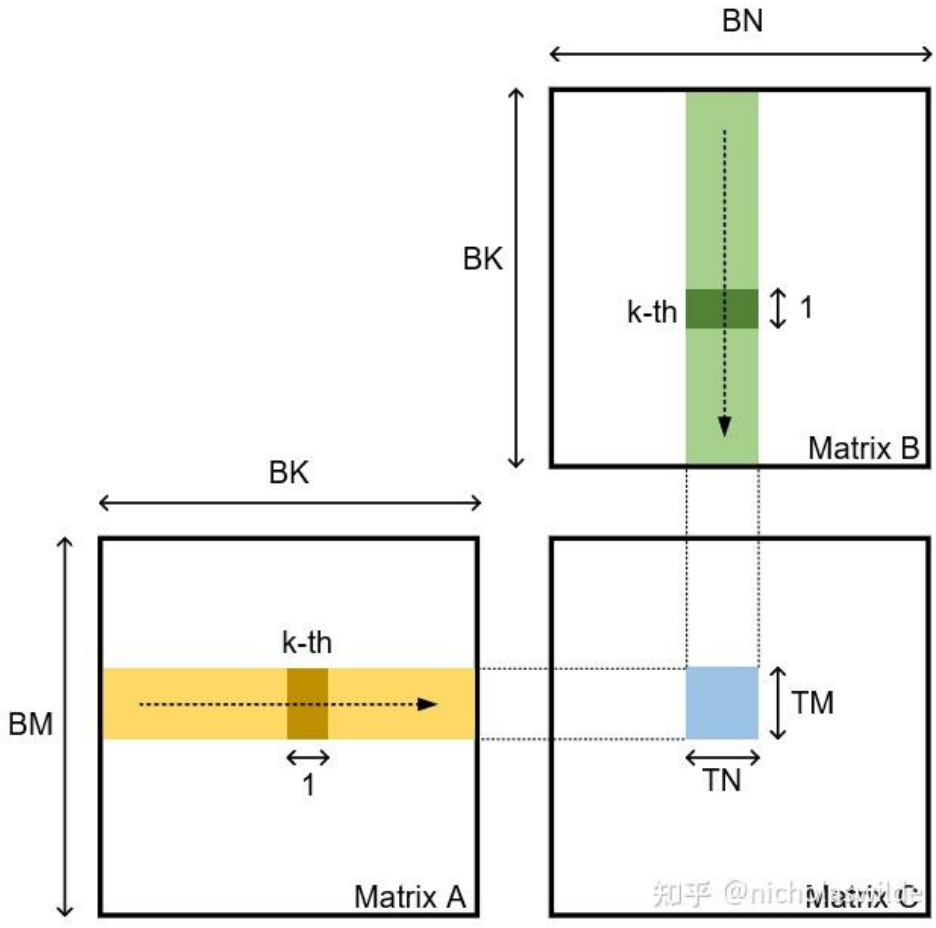
5 处理Bank Conflict

上一节做完矩阵分块的代码，终于摆脱了naiveSgemm的低效，达到了2600 GFLOPS，差不多有cublas的70%。之前我们重点关注了一下Global Memory的访存，本节将优化Shared Memory的访问。

Shared Memory一共划分为32个Bank，每个Bank的宽度为4 Bytes，如果需要访问同一个Bank的多个数据，就会发生Bank Conflict。例如一个Warp的32个线程，如果访问的地址分别为0、4、8、...、124，就不会发生Bank Conflict，只占用Shared Memory一拍的时间；如果访问的地址为0、8、16、...、248，这样一来地址0和地址128对应的数据位于同一Bank、地址4和地址132对应的数据位于同一Bank，以此类推，那么就需要占用Shared Memory两拍的时间才能读出。

我们来观察mySgemmV1Aligned主循环体中计算部分的三层循环，一个线程的计算过程如下图所示，每次从Shared memory中取矩阵A的长度为TM的向量和矩阵B的长度为TN的向量，这两个向量做外积并累加到部分和中，一次外积共TM * TN次乘累加，一共需要循环BK次取数和外积。在每一次从Shared Memory load的过程中，存在着显而易见的Bank Conflict：

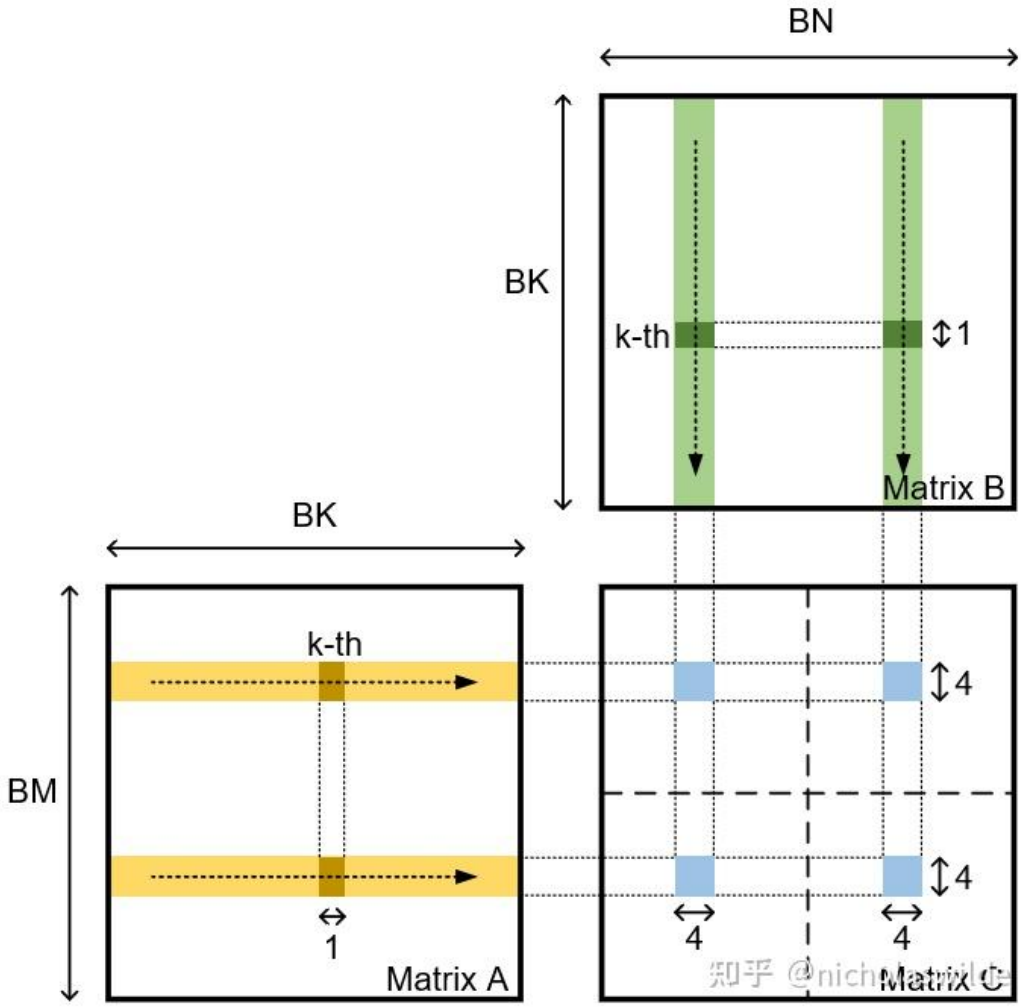
- 取矩阵A需要取一个列向量，而矩阵A在Shared Memory中是按行存储的；
- 在TM = TN = 8的情况下，无论矩阵A还是矩阵B，从Shared Memory中取数时需要取连续的8个数，即便用LDS.128指令一条指令取四个数，也需要两条指令，由于一个线程的两条load指令的地址是连续的，那么同一个Warp不同线程的同一条load指令的访存地址就是被间隔开的，便存在着Bank Conflict。



SGEMM Thread Tiling

为了解决上述的两点Shared Memory的Bank Conflict，采用了一下两点优化：

- 为矩阵A分配Shared Memory时形状分配为[BK][BM]，也就是让矩阵A在Shared Memory中按列存储
- 将原本每个线程负责计算的TM * TN的矩阵C，划分为下图中这样的四块4 * 4的矩阵C（实验中实测划分成两块，也就是解决A/B中一个矩阵的Bank Confilict就足够，划分成四块并没有比两块带来更高的性能）



SGEMM Thread Tiling Avoiding Shared Memory Bank Conflict

这样Shared Memory的Bank Conflict就处理完毕啦。至于Register的Bank Conflict，需要对SASS汇编下手，因此我选择相信nvcc，毕竟SGEMM中主要的FMMA指令一条指令只需要访问三个寄存器，而寄存器堆有4个Bank。经此优化后的SGEMM kernal如下所示，其性能最高可以达到2800 GFLOPS，接近cublas的80%。

```
__global__ void mySgemmV2Aligned(
    float * __restrict__ a, float * __restrict__ b, float * __restrict__ c,
    const int M, const int N, const int K) {

    const int BM = 128;
    const int BN = 128;
    const int BK = 8;
    const int TM = 8;
    const int TN = 8;

    const int bx = blockIdx.x;
```

```
const int by = blockIdx.y;
const int tx = threadIdx.x;
const int ty = threadIdx.y;
const int tid = ty * blockDim.x + tx;

__shared__ float s_a[BK][BM];
__shared__ float s_b[BK][BN];

float r_load_a[4];
float r_load_b[4];
float r_comp_a[TM];
float r_comp_b[TN];
float r_c[TM][TN] = {0.0};

int load_a_smem_m = tid >> 1;
int load_a_smem_k = (tid & 1) << 2;
int load_b_smem_k = tid >> 5;
int load_b_smem_n = (tid & 31) << 2;

int load_a_gmem_m = by * BM + load_a_smem_m;
int load_b_gmem_n = bx * BN + load_b_smem_n;

for (int bk = 0; bk < (K + BK - 1) / BK; bk++) {

    int load_a_gmem_k = bk * BK + load_a_smem_k;
    int load_a_gmem_addr = OFFSET(load_a_gmem_m, load_a_gmem_k, K);
    int load_b_gmem_k = bk * BK + load_b_smem_k;
    int load_b_gmem_addr = OFFSET(load_b_gmem_k, load_b_gmem_n, N);
    FLOAT4(r_load_a[0]) = FLOAT4(a[load_a_gmem_addr]);
    FLOAT4(r_load_b[0]) = FLOAT4(b[load_b_gmem_addr]);

    s_a[load_a_smem_k    ][load_a_smem_m] = r_load_a[0];
    s_a[load_a_smem_k + 1][load_a_smem_m] = r_load_a[1];
    s_a[load_a_smem_k + 2][load_a_smem_m] = r_load_a[2];
    s_a[load_a_smem_k + 3][load_a_smem_m] = r_load_a[3];
    FLOAT4(s_b[load_b_smem_k][load_b_smem_n]) = FLOAT4(r_load_b[0]);

    __syncthreads();

    #pragma unroll
    for (int tk = 0; tk < BK; tk++) {
        FLOAT4(r_comp_a[0]) = FLOAT4(s_a[tk][ty * TM / 2         ]);
        FLOAT4(r_comp_a[4]) = FLOAT4(s_a[tk][ty * TM / 2 + BM / 2]);
        FLOAT4(r_comp_b[0]) = FLOAT4(s_b[tk][tx * TN / 2         ]);
        FLOAT4(r_comp_b[4]) = FLOAT4(s_b[tk][tx * TN / 2 + BN / 2]);

        #pragma unroll
        for (int tm = 0; tm < TM; tm++) {
            #pragma unroll
            for (int tn = 0; tn < TN; tn++) {
                r_c[tm][tn] += r_comp_a[tm] * r_comp_b[tn];
            }
        }
    }

    __syncthreads();
}
```

```
    #pragma unroll
    for (int i = 0; i < TM / 2; i++) {
        int store_c_gmem_m = by * BM + ty * TM / 2 + i;
        int store_c_gmem_n = bx * BN + tx * TN / 2;
        int store_c_gmem_addr = OFFSET(store_c_gmem_m, store_c_gmem_n, N);
        FLOAT4(c[store_c_gmem_addr]) = FLOAT4(r_c[i][0]);
        FLOAT4(c[store_c_gmem_addr + BN / 2]) = FLOAT4(r_c[i][4]);
    }
    #pragma unroll
    for (int i = 0; i < TM / 2; i++) {
        int store_c_gmem_m = by * BM + BM / 2 + ty * TM / 2 + i;
        int store_c_gmem_n = bx * BN + tx * TN / 2;
        int store_c_gmem_addr = OFFSET(store_c_gmem_m, store_c_gmem_n, N);
        FLOAT4(c[store_c_gmem_addr]) = FLOAT4(r_c[i + TM / 2][0]);
        FLOAT4(c[store_c_gmem_addr + BN / 2]) = FLOAT4(r_c[i + TM / 2][4]);
    }
}
```

6 Double Buffering

Double Buffering，顾名思义双缓冲，就是对于”访存1—计算1—访存2—计算2—.....—访存n—计算n“这样的任务序列，由于计算i和访存i+1是不相关的，因此可以将下一次访存和上一次计算并行，变成”访存1—（访存2 & 计算1）—.....—（访存i+1 & 计算i）—.....—（访存n & 计算n-1）—计算n“这样的任务序列。

具体到SGEMM这个任务中来，就可以将”从Global Momery将A和B的子矩阵load到Shared Memory“和”计算已经load到Shared Memory中的A和B的上一个子矩阵“并行起来，以掩盖load Global Memory长达300多拍的延迟。当然，使用Double Buffering的代价是需要两倍的Shared Memory，之前只需要BK * (BM + BN) * 4 Bytes的Shared Memory，采用Double Buffering之后需要2BK * (BM + BN) * 4 Bytes的Shared Memory。

修改后的代码如下，需要注意的地方是：

- 主循环从K / BK次变为K / BK - 1次，第一次load在主循环之前，最后一次计算在主循环之后
- 由于计算和下一次访存使用的Shared Memory不同，因此主循环中每次循环只需要一次__syncthreads()即可
- 由于GPU不能向CPU那样支持乱序执行，主循环中需要先将下一次循环计算需要的Gloabal Memory中的数据load到寄存器，然后进行本次计算，之后再将load到寄存器中的数据写到Shared Memory，这样在LDG指令向Global Memory做load时，不会影响后续FFMA及其它运算指令的发射执行，也就达到了Double Buffering的目的。如果调换一下顺序，将load到寄存器中的数据写到Shared Memory放在本次计算之前，那么就会有寄存器的相关，STS指令会等待LDG指令写回后再继续发射执行，指令序列位于STS指令之后的一大堆FFMA指令也就不能在LDG指令访存时发射执行。
- 我在编译时发现，如果不加编译选项的话会编译出使用133个寄存器的版本，使得Occupancy下降，因此我在编译选项中添加了-maxrregcount=128，编译得到使用127个寄存器的版本。

```
__global__ void mySgemmV3Aligned(
    float * __restrict__ a, float * __restrict__ b, float * __restrict__ c,
    const int M, const int N, const int K) {
```

```
const int BM = 128;
const int BN = 128;
const int BK = 8;
const int TM = 8;
const int TN = 8;

const int bx = blockIdx.x;
const int by = blockIdx.y;
const int tx = threadIdx.x;
const int ty = threadIdx.y;
const int tid = ty * blockDim.x + tx;

__shared__ float s_a[2][BK][BM];
__shared__ float s_b[2][BK][BN];

float r_load_a[4];
float r_load_b[4];
float r_comp_a[TM];
float r_comp_b[TN];
float r_c[TM][TN] = {0.0};

int load_a_smem_m = tid >> 1;
int load_a_smem_k = (tid & 1) << 2;
int load_b_smem_k = tid >> 5;
int load_b_smem_n = (tid & 31) << 2;

int load_a_gmem_m = by * BM + load_a_smem_m;
int load_b_gmem_n = bx * BN + load_b_smem_n;

{
    int load_a_gmem_k = load_a_smem_k;
    int load_a_gmem_addr = OFFSET(load_a_gmem_m, load_a_gmem_k, K);
    int load_b_gmem_k = load_b_smem_k;
    int load_b_gmem_addr = OFFSET(load_b_gmem_k, load_b_gmem_n, N);
    FLOAT4(r_load_a[0]) = FLOAT4(a[load_a_gmem_addr]);
    FLOAT4(r_load_b[0]) = FLOAT4(b[load_b_gmem_addr]);

    s_a[0][load_a_smem_k    ][load_a_smem_m] = r_load_a[0];
    s_a[0][load_a_smem_k + 1][load_a_smem_m] = r_load_a[1];
    s_a[0][load_a_smem_k + 2][load_a_smem_m] = r_load_a[2];
    s_a[0][load_a_smem_k + 3][load_a_smem_m] = r_load_a[3];
    FLOAT4(s_b[0][load_b_smem_k][load_b_smem_n]) = FLOAT4(r_load_b[0]);
}

for (int bk = 1; bk < (K + BK - 1) / BK; bk++) {

    int smem_sel = (bk - 1) & 1;
    int smem_sel_next = bk & 1;

    int load_a_gmem_k = bk * BK + load_a_smem_k;
    int load_a_gmem_addr = OFFSET(load_a_gmem_m, load_a_gmem_k, K);
    int load_b_gmem_k = bk * BK + load_b_smem_k;
    int load_b_gmem_addr = OFFSET(load_b_gmem_k, load_b_gmem_n, N);
    FLOAT4(r_load_a[0]) = FLOAT4(a[load_a_gmem_addr]);
    FLOAT4(r_load_b[0]) = FLOAT4(b[load_b_gmem_addr]);

    #pragma unroll
```

```
        for (int tk = 0; tk < BK; tk++) {
            FLOAT4(r_comp_a[0]) = FLOAT4(s_a[smem_sel][tk][ty * TM / 2         ]);
            FLOAT4(r_comp_a[4]) = FLOAT4(s_a[smem_sel][tk][ty * TM / 2 + BM / 2]);
            FLOAT4(r_comp_b[0]) = FLOAT4(s_b[smem_sel][tk][tx * TN / 2         ]);
            FLOAT4(r_comp_b[4]) = FLOAT4(s_b[smem_sel][tk][tx * TN / 2 + BN / 2]);

            #pragma unroll
            for (int tm = 0; tm < TM; tm++) {
                #pragma unroll
                for (int tn = 0; tn < TN; tn++) {
                    r_c[tm][tn] += r_comp_a[tm] * r_comp_b[tn];
                }
            }
        }

        s_a[smem_sel_next][load_a_smem_k    ][load_a_smem_m] = r_load_a[0];
        s_a[smem_sel_next][load_a_smem_k + 1][load_a_smem_m] = r_load_a[1];
        s_a[smem_sel_next][load_a_smem_k + 2][load_a_smem_m] = r_load_a[2];
        s_a[smem_sel_next][load_a_smem_k + 3][load_a_smem_m] = r_load_a[3];
        FLOAT4(s_b[smem_sel_next][load_b_smem_k][load_b_smem_n]) = FLOAT4(r_load_b[0]);

        __syncthreads();
    }

    #pragma unroll
    for (int tk = 0; tk < BK; tk++) {
        FLOAT4(r_comp_a[0]) = FLOAT4(s_a[1][tk][ty * TM / 2         ]);
        FLOAT4(r_comp_a[4]) = FLOAT4(s_a[1][tk][ty * TM / 2 + BM / 2]);
        FLOAT4(r_comp_b[0]) = FLOAT4(s_b[1][tk][tx * TN / 2         ]);
        FLOAT4(r_comp_b[4]) = FLOAT4(s_b[1][tk][tx * TN / 2 + BN / 2]);

        #pragma unroll
        for (int tm = 0; tm < TM; tm++) {
            #pragma unroll
            for (int tn = 0; tn < TN; tn++) {
                r_c[tm][tn] += r_comp_a[tm] * r_comp_b[tn];
            }
        }
    }

    #pragma unroll
    for (int i = 0; i < TM / 2; i++) {
        int store_c_gmem_m = by * BM + ty * TM / 2 + i;
        int store_c_gmem_n = bx * BN + tx * TN / 2;
        int store_c_gmem_addr = OFFSET(store_c_gmem_m, store_c_gmem_n, N);
        FLOAT4(c[store_c_gmem_addr]) = FLOAT4(r_c[i][0]);
        FLOAT4(c[store_c_gmem_addr + BN / 2]) = FLOAT4(r_c[i][4]);
    }
    #pragma unroll
    for (int i = 0; i < TM / 2; i++) {
        int store_c_gmem_m = by * BM + BM / 2 + ty * TM / 2 + i;
        int store_c_gmem_n = bx * BN + tx * TN / 2;
        int store_c_gmem_addr = OFFSET(store_c_gmem_m, store_c_gmem_n, N);
        FLOAT4(c[store_c_gmem_addr]) = FLOAT4(r_c[i + TM / 2][0]);
        FLOAT4(c[store_c_gmem_addr + BN / 2]) = FLOAT4(r_c[i + TM / 2][4]);
    }
}
```

这里还要提一句，从Shared Memory到Register的load也是可以分配两组寄存器做Double Buffering的，但是我没有在代码里显式写出来的原因是：我查看了反汇编出来的SASS指令，编译器自动优化了！自动分配了两组寄存器存放矩阵A和矩阵B对应的向量，自动做了Double Buffering！我越来越相信nvcc了！

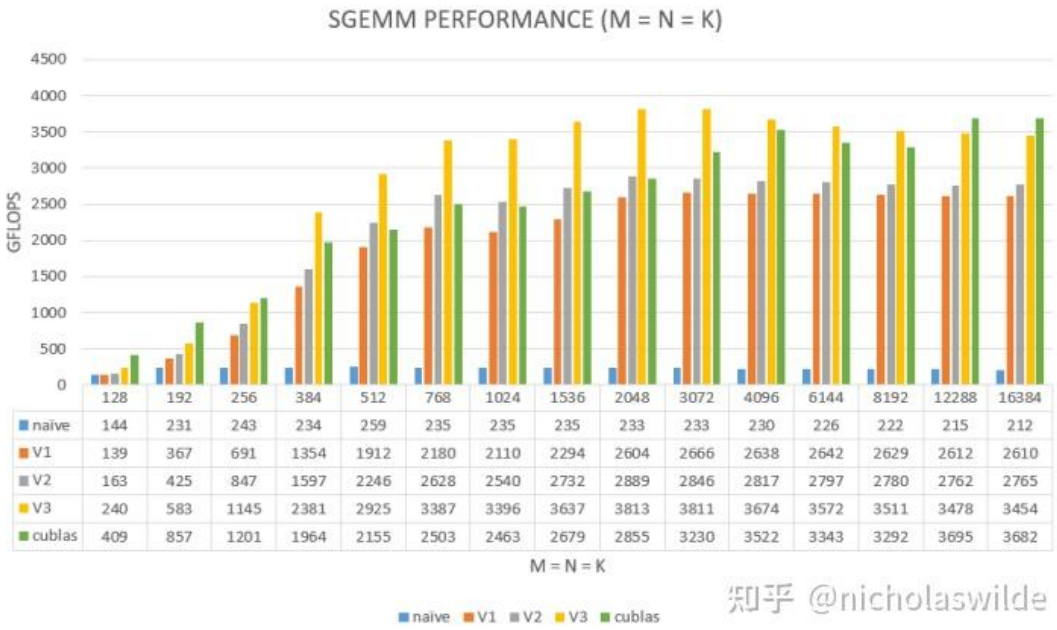
7 Experiment

我在自己笔记本的GTX1060 Mobile 6G上测试了一下cublas以及自己实现的几个SGEMM kernal的性能。我的GTX1060只有10个SM，有1280个CUDA Core；GPU频率1.404 GHz，Boost频率1.670 GHz，不过在测试时使用GPU-Z实时看到的GPU频率最高维持在1.809 GHz，在1.809 GHz的理论峰值性能为1280 * 1.670 * 2 = 4631 GFLOPS。

我进行了两组实验，一组是K保持1024不变，M和N同步变化；另一组是M、N、K同步变化，跑10次取平均时间计算浮点性能。结果如下，mySgemmV3Aligned最高达到了3900 GFLOPS，且在大部分case下超过了cublas：

	128	192	256	384	512	768	1024	1536	2048	3072	4096	6144	8192	12288	16384
naïve	188	215	202	219	231	231	236	237	237	237	237	238	238	237	236
V1	172	386	703	1541	2088	2321	2221	2425	2613	2673	2650	2676	2680	2662	2657
V2	189	422	778	1690	2323	2648	2487	2738	2862	2908	2880	2898	2890	2864	2865
V3	299	669	1231	2738	2985	3392	3344	3595	3736	3910	3843	3777	3732	3681	3671
cublas	1100	1427	1636	1943	2006	2522	2402	2614	2745	2884	3126	3545	3609	3548	3546

Performance M = N, K = 1024



	128	192	256	384	512	768	1024	1536	2048	3072	4096	6144	8192	12288	16384
naive	144	231	243	234	259	235	235	235	233	233	230	226	222	215	212
V1	139	367	691	1354	1912	2180	2110	2294	2604	2666	2638	2642	2629	2612	2610
V2	163	425	847	1597	2246	2628	2540	2732	2889	2846	2817	2797	2780	2762	2765
V3	240	583	1145	2381	2925	3387	3396	3637	3813	3811	3674	3572	3511	3478	3454
cublas	409	857	1201	1964	2155	2503	2463	2679	2855	3230	3522	3343	3292	3695	3682

Performance M = N = K

当然啦，我的kernal没有处理不对齐的情况，加入条件判断会使得性能下降。不过测试发现cublas在计算不对其（例如1023 * 1023 * 1023）的矩阵乘法时性能也会比整齐的有所下降。

最后聊一聊BM、BN、BK、TM、TN参数选取的情况，感觉cublas会根据不同的M、N、K来选择不同的kernal进行计算，而我把这些参数写死导致有些case尤其是小规模的case性能极差。但是大规模的矩阵乘法下，BM = BN = 128, BK = 8, TM = TN = 8这组参数的表现还是很好的，感觉大家选取的很有讲究：如果BM、BN比128更小，对Global Memory的访存就会增加；如果BM、BN更大，一个Block对应的Shared Memory和寄存器就不够用了；在BM = BN = 128的情况下，TM和TN比8更大，需要的寄存器数超过128，会导致Occupancy降低到8 Warp；TM和TN更小，线程增加，一个SM中并行的Block就会变少，而且现在刚好访存时一个线程从Global Memory中load矩阵A和矩阵B的各一个float4。总之这个前人的经验参数直觉上还是很舒服的。

8 Reference

1. CUDA 矩阵乘法终极优化指南
2. 深入浅出GPU优化系列：GEMM优化
3. NVIDIA A100 Tensor Core GPU Architecture Whitepaper
4. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking
5. CUDA C++ Programming Guide v11.4
