SGEMM

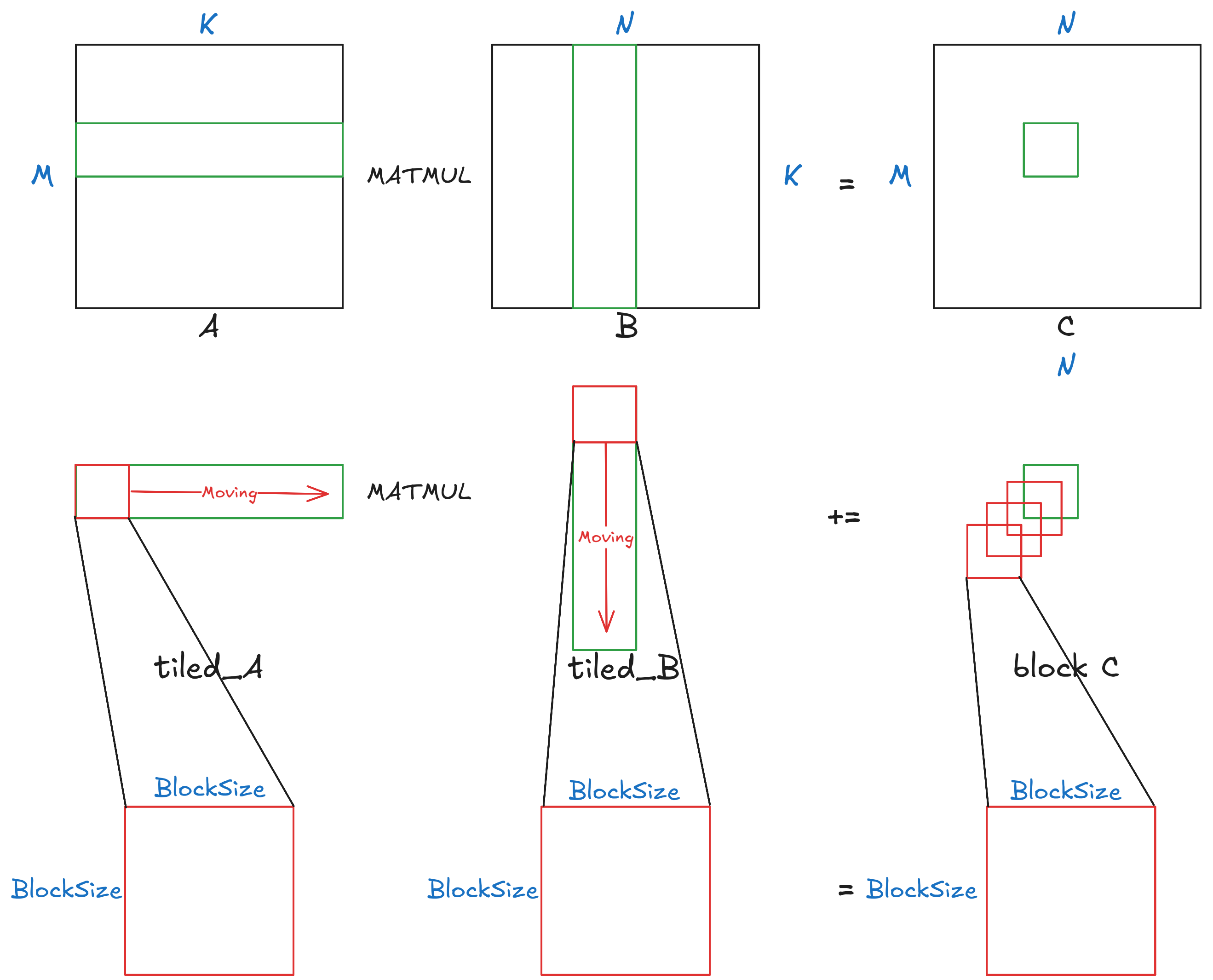
sgemm_v0_global_mem

在第一个版本中我们实现最原始的sgemm，使用naive算法和global mem。

```
__global__ void sgemm0(float *A, float *B, float *C, const int M,
const int N, const int K)
{
    int m = blockIdx.y * blockDim.y + threadIdx.y;
    int n = blockIdx.x * blockDim.x + threadIdx.x;

    if (m < M && n < N)
    {
        float temp = 0.f;
        for (int k = 0; k < K; ++k)
        {
            temp += A[m * K + k] * B[k * N + n];
        }
        C[m * N + n] = temp;
    }
}
```

sgemm_v1_shared_mem

接下来我们使用shared mem来优化访存效率。流程图如下：



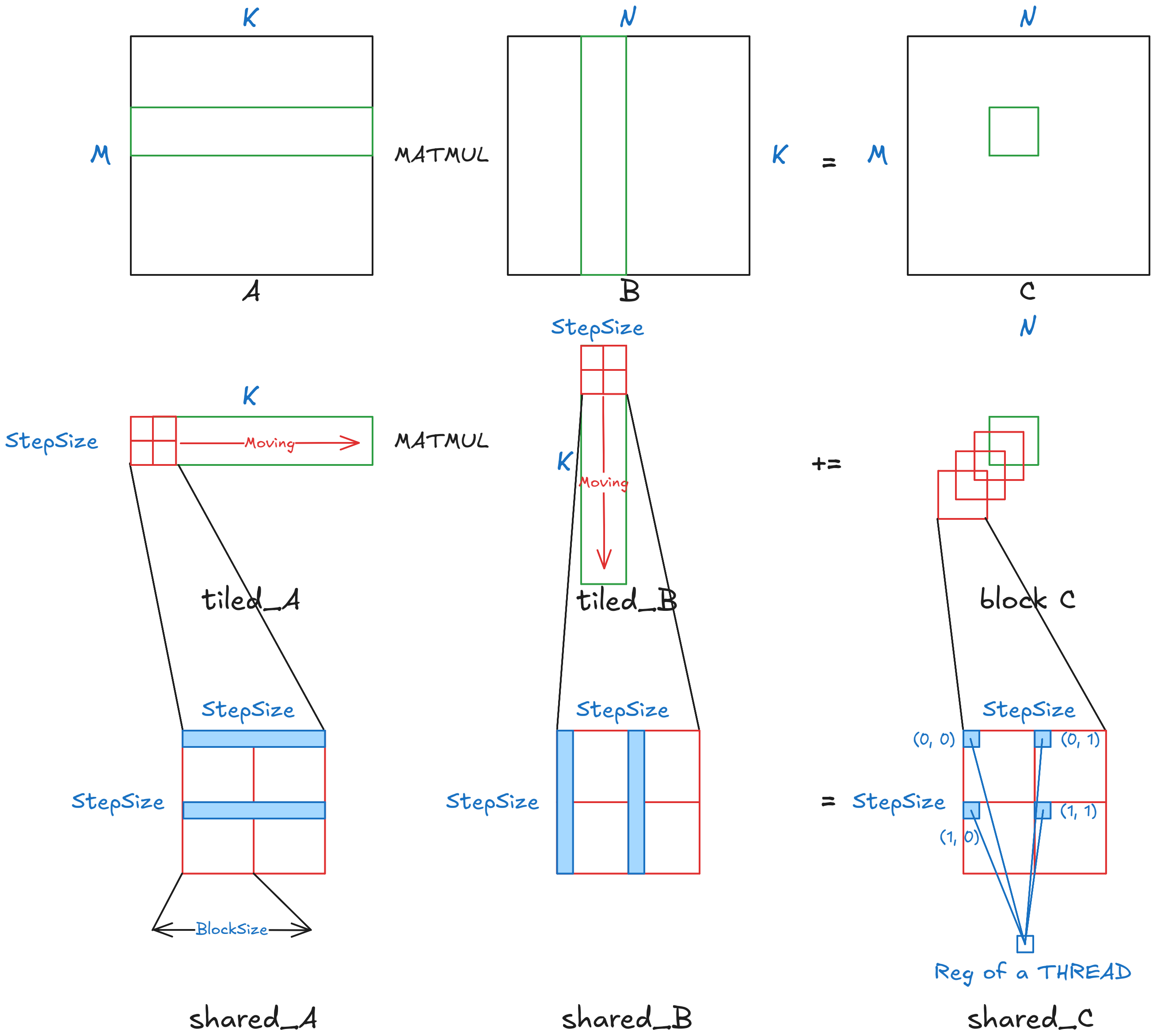
在这个版本中我们依旧使用naive算法，代码如下，因为索引方式在最后会统一，这里先不介绍索引方式。

```
template<int BLOCKSIZE>
__global__ void sgemm1(float *A, float *B, float *C, const int M,
const int N, const int K)
{
    const int tid_x = threadIdx.x;
    const int tid_y = threadIdx.y;

    const int global_block_start_x = blockIdx.x * blockDim.x;
    const int global_block_start_y = blockIdx.y * blockDim.y;

    // Register per thread
    float sum = 0.f;
    for (size_t i = 0; i * BLOCKSIZE < K; i++)
    {
        __shared__ float shared_A[BLOCKSIZE][BLOCKSIZE];
        __shared__ float shared_B[BLOCKSIZE][BLOCKSIZE];
```

```
        size_t offsetA = i * BLOCKSIZE;
        size_t offsetB = i * BLOCKSIZE * N;
        int inner_idx_A = tid_y * K + tid_x;
        int inner_idx_B = tid_y * N + tid_x;

        shared_A[tid_y][tid_x] = A[global_block_start_y * K +
offsetA + inner_idx_A];
        shared_B[tid_y][tid_x] = B[global_block_start_x + offsetB +
inner_idx_B];

        __syncthreads();

#pragma unroll
        for (int i = 0; i < BLOCKSIZE; i++)
        {
            sum += shared_A[tid_y][i] * shared_B[i][tid_x];
        }

        __syncthreads();
    }

    C[(global_block_start_y + tid_y) * N + global_block_start_x +
tid_x] = sum;
}
```

sgemm_v2_increase_workload_of_threads

sgemm是一个访存型的算子，所以相比于访存负载，计算的负载是比较小的。我们可以通过启动更少的线程，让一个线程负责由多个数据组成的小块的计算。这样可以提高计算访存比，同时也可以起到遮掩访存的作用以减少空闲的线程。



在这个版本中，我们在naive算法上进行小改动，只需要让一个线程负责一个小块即可。具体实现方法是，我们通过模板传入STRIDE，表示一个线程负责的小块的边长。所以一个Thread要处理STRIDE * STRIDE个数据，这里我们取STRIDE为2。每个Thread处理的结果可以储存在寄存器中，按照STEP = STRIDE * BLOCKSIZE的步长移动，每个block都读取STRIDE ^ 2倍数的数据并处理（最底下一行所示）。最后再控制每个线程将寄存器中的STRIDE ^ 2数据写回。

代码如下：

```
#define inner_i (ii * BLOCKSIZE + tid_y)
#define inner_j (jj * BLOCKSIZE + tid_x)

template <int BLOCKSIZE, int STRIDE>
__global__ void sgemm2(float *A, float *B, float *C, const int M,
const int N, const int K)
{
    constexpr int STEP = BLOCKSIZE * STRIDE;
```

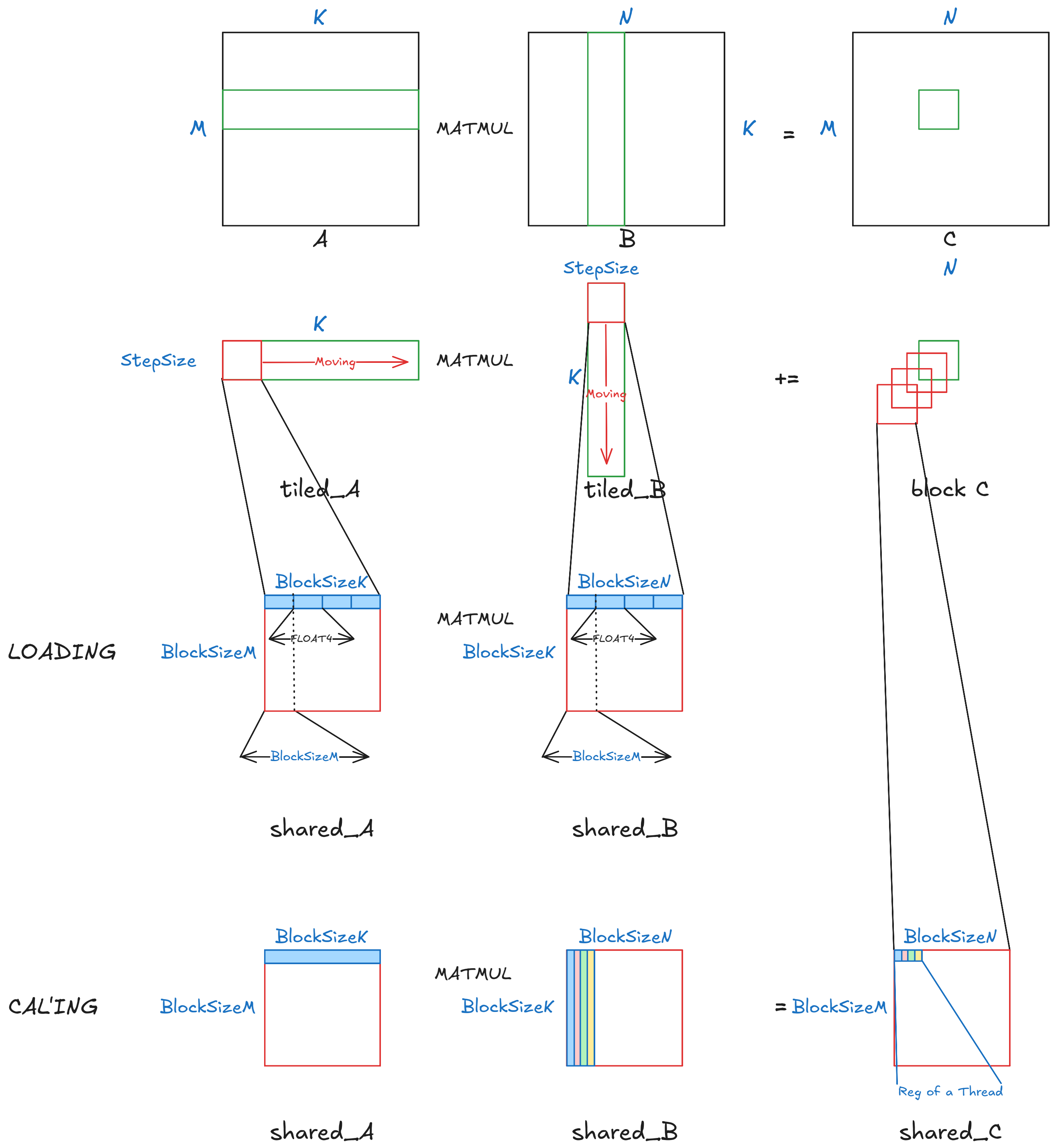
```
    const int tid_x = threadIdx.x;
    const int tid_y = threadIdx.y;

    const int global_block_start_x = blockIdx.x * STEP;
    const int global_block_start_y = blockIdx.y * STEP;

    // Register per thread
    float sum[STRIDE][STRIDE] = {0.f};
    for (size_t i = 0; i * STEP < K; i++)
    {
        __shared__ float shared_A[STEP][STEP];
        __shared__ float shared_B[STEP][STEP];

        size_t offsetA = i * STEP;
        size_t offsetB = i * STEP * N;

#pragma unroll
        for (int ii = 0; ii < STRIDE; ii++)
        {
#pragma unroll
            for (int jj = 0; jj < STRIDE; jj++)
            {
                shared_A[inner_i][inner_j] = A[global_block_start_y
* K + offsetA + inner_i * K + inner_j];
                shared_B[inner_i][inner_j] = B[global_block_start_x
+ offsetB + inner_i * N + inner_j];
            }
        }
        __syncthreads();

#pragma unroll
        for (int ii = 0; ii < STRIDE; ii++)
        {
#pragma unroll
            for (int jj = 0; jj < STRIDE; jj++)
            {
#pragma unroll
                for (int kk = 0; kk < STEP; kk++)
                {

                    sum[ii][jj] += shared_A[inner_i][kk] *
shared_B[kk][inner_j];
                }
```

```
            }
        }
        __syncthreads();
    }

    for (int ii = 0; ii < STRIDE; ii++)
    {
        for (int jj = 0; jj < STRIDE; jj++)
        {

            C[(global_block_start_y + inner_i) * N +
global_block_start_x + inner_j] = sum[ii][jj];
        }
    }
}
```

sgemm_v3_float4

上面版本中我们让一个线程负责处理4个数据，所以每个线程加载时就要加载4个数据。一个很自然的优化方法是让线程加载连续的4个数据，并且使用float4变量，一次性加载4个数据。

下面给出示意图：



在这里我们计算一个Block内的结果时，一个线程负责得到4个位置的结果。我们可以每次循环处理BlockSize个的数据，然后总体循环4次得到。但是这样会带来冗余，因为shared_A中的每一个行都被访问了4次用来计算一个Thread负责的4个位置。

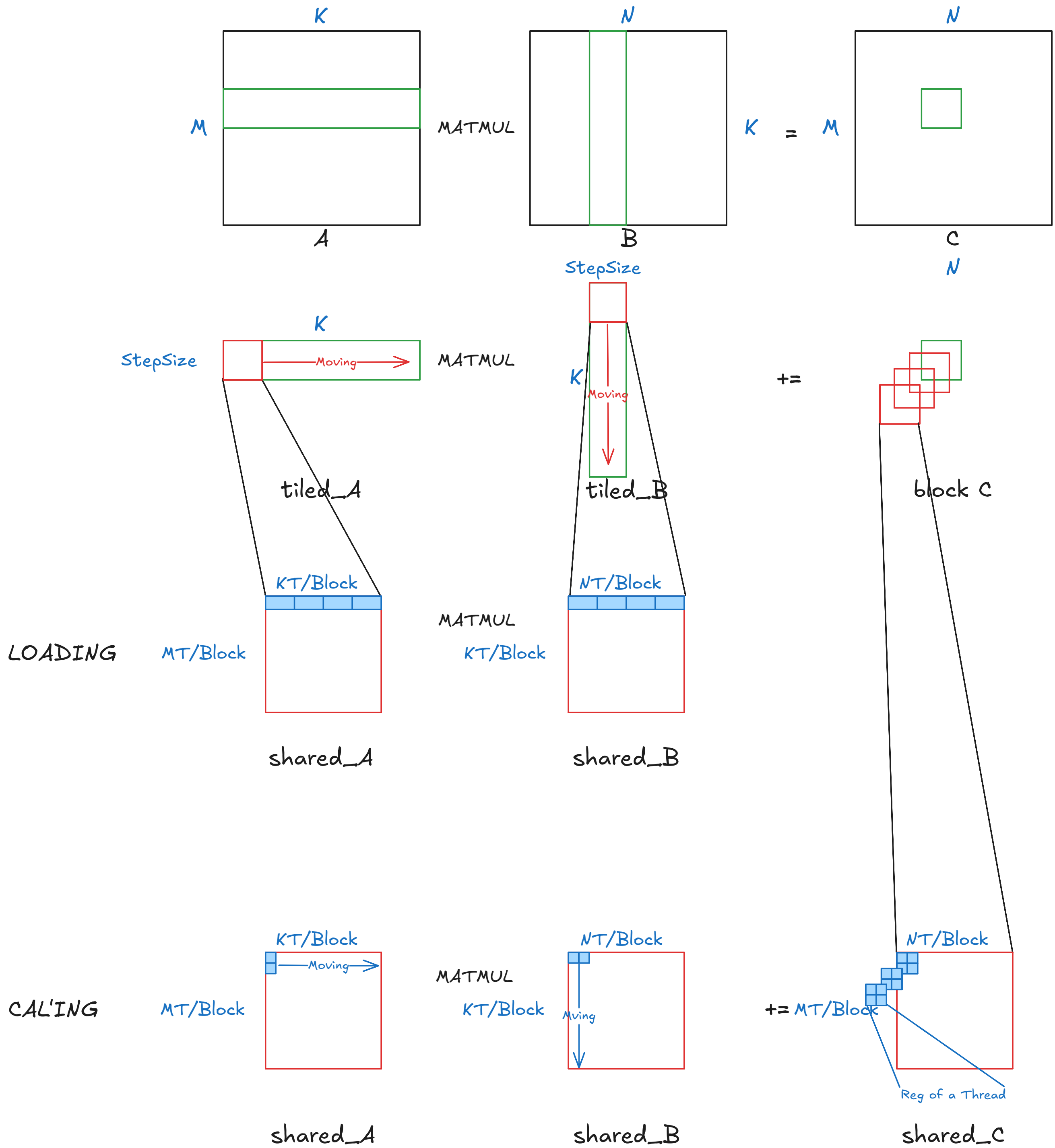
我们这里定义 `FETCH_FLOAT4` 将原本的指针重新解释成float4指针，这样就可以一次读取4个数据。

```
#define inner_j (tid_x * NUM_PER_THREAD)
#define FETCH_FLOAT4(ptr) (reinterpret_cast<float4 *>(&(ptr))[0])
template <int M_NUM_PER_BLOCK, int N_NUM_PER_BLOCK, int
K_NUM_PER_BLOCK, int NUM_PER_THREAD>
```

```
__global__ void sgemm3(float *A, float *B, float *C, const int M,
const int N, const int K)
{
    int tid_x = threadIdx.x;
    int tid_y = threadIdx.y;

    int block_start_x = blockIdx.x * K_NUM_PER_BLOCK;
    int block_start_y = blockIdx.y * M_NUM_PER_BLOCK;

    __shared__ float shared_A[M_NUM_PER_BLOCK][K_NUM_PER_BLOCK];
    __shared__ float shared_B[K_NUM_PER_BLOCK][N_NUM_PER_BLOCK];

    float sum[NUM_PER_THREAD] = {0.f};

#pragma unroll
    for (int s = 0; s < K; s += K_NUM_PER_BLOCK)
    {
        FETCH_FLOAT4(shared_A[tid_y][inner_j]) =
FETCH_FLOAT4(A[(block_start_y + tid_y) * K + s + inner_j]);
        FETCH_FLOAT4(shared_B[tid_y][inner_j]) =
FETCH_FLOAT4(B[(tid_y + s) * N + block_start_x + inner_j]);
        __syncthreads();

#pragma unroll
        for (int i = 0; i < NUM_PER_THREAD; ++i)
        {
#pragma unroll
            for (int k = 0; k < K_NUM_PER_BLOCK; k++)
            {
                sum[i] += shared_A[tid_y][k] * shared_B[k][inner_j
+ i];
            }
        }
        __syncthreads();
    }

    float *C_start = C + blockIdx.y * M_NUM_PER_BLOCK * N +
blockIdx.x * N_NUM_PER_BLOCK;
#pragma unroll
    for (int i = 0; i < NUM_PER_THREAD; ++i)
    {
        C_start[tid_y * N + tid_x * NUM_PER_THREAD + i] = sum[i];
    }
}
```

sgemm_v4_reg

这一个版本我们让一个Thread负责一个2 * 2的方块并且引入新的索引方式。从现在开始kernel内部对线程索引的计算与启动的Block形状无关，我们手动计算线程的索引，这样可以带来更高的灵活性。另外对于global mem的索引针对每一个Block我们计算形容A_start的float指针以简化计算矩阵索引的过程。具体实现流程可以通过额外设置两个寄存器存储来自shared_A和shared_B的数据。通过这个改进，我们可以缓解上一个版本中冗余访问shared_A中行向量的问题。



代码：

```
template <int M_NUM_PER_BLOCK, int N_NUM_PER_BLOCK, int
K_NUM_PER_BLOCK, int NUM_PER_THREAD>
__global__ void sgemm4(float *A, float *B, float *C, const int M,
const int N, const int K)
{
    constexpr int REG_NUM = NUM_PER_THREAD >> 1;
    int ctid[2];
    reIndex(ctid, N_NUM_PER_BLOCK / REG_NUM);
    int ctid_x = ctid[0];
    int ctid_y = ctid[1];

    int ltid[2];
    reIndex(ltid, N_NUM_PER_BLOCK / NUM_PER_THREAD);
    int ltid_x = ltid[0];
    int ltid_y = ltid[1];

    float* A_start = A + (blockIdx.y * M_NUM_PER_BLOCK) * K;
    float* B_start = B + (blockIdx.x * K_NUM_PER_BLOCK);

    __shared__ float shared_A[M_NUM_PER_BLOCK][K_NUM_PER_BLOCK];
    __shared__ float shared_B[K_NUM_PER_BLOCK][N_NUM_PER_BLOCK];

    float a_reg[REG_NUM] = {0.f};
    float b_reg[REG_NUM] = {0.f};
    float sum[REG_NUM][REG_NUM] = {0.f};

int inner_j = ltid_x * NUM_PER_THREAD;
#pragma unroll
    for (int s = 0; s < K; s += K_NUM_PER_BLOCK)
    {
        FETCH_FLOAT4(shared_A[ltid_y][inner_j]) =
FETCH_FLOAT4(A_start[ltid_y * K + (s + inner_j)]);
        FETCH_FLOAT4(shared_B[ltid_y][inner_j]) =
FETCH_FLOAT4(B_start[(ltid_y + s) * N + inner_j]);
        __syncthreads();

        for (int k = 0; k < K_NUM_PER_BLOCK; ++k)
        {
            a_reg[0] = shared_A[ctid_y * REG_NUM][k];
            a_reg[1] = shared_A[ctid_y * REG_NUM + 1][k];
            b_reg[0] = shared_B[k][ctid_x * REG_NUM];
            b_reg[1] = shared_B[k][ctid_x * REG_NUM + 1];
#pragma unroll
```

```
            for (int ii = 0; ii < REG_NUM; ii++)
            {
#pragma unroll
                for (int jj = 0; jj < REG_NUM; jj++)
                {
                    sum[ii][jj] += a_reg[ii] * b_reg[jj];
                }
            }
        }

        __syncthreads();
    }

    float *C_start = C + blockIdx.y * M_NUM_PER_BLOCK * N +
blockIdx.x * N_NUM_PER_BLOCK;
#pragma unroll
    for (int i = 0; i < REG_NUM; i++)
    {
        for (int j = 0; j < REG_NUM; j++)
        {
            C_start[(ctid_y * REG_NUM + i) * N + (ctid_x * REG_NUM
+ j)] = sum[i][j];
        }
    }
}
```

sgemm_v5_reg_float4

在这一版本中我们进一步优化计算访存比，让一个线程的计算负载再次增加，一个很朴素的改进想法就是在计算前也使用float4变量将参与计算的数据从共享内存存入寄存器。同时我们在这一步为我们的kernel添加一些其他模板参数以适应更多的形状。

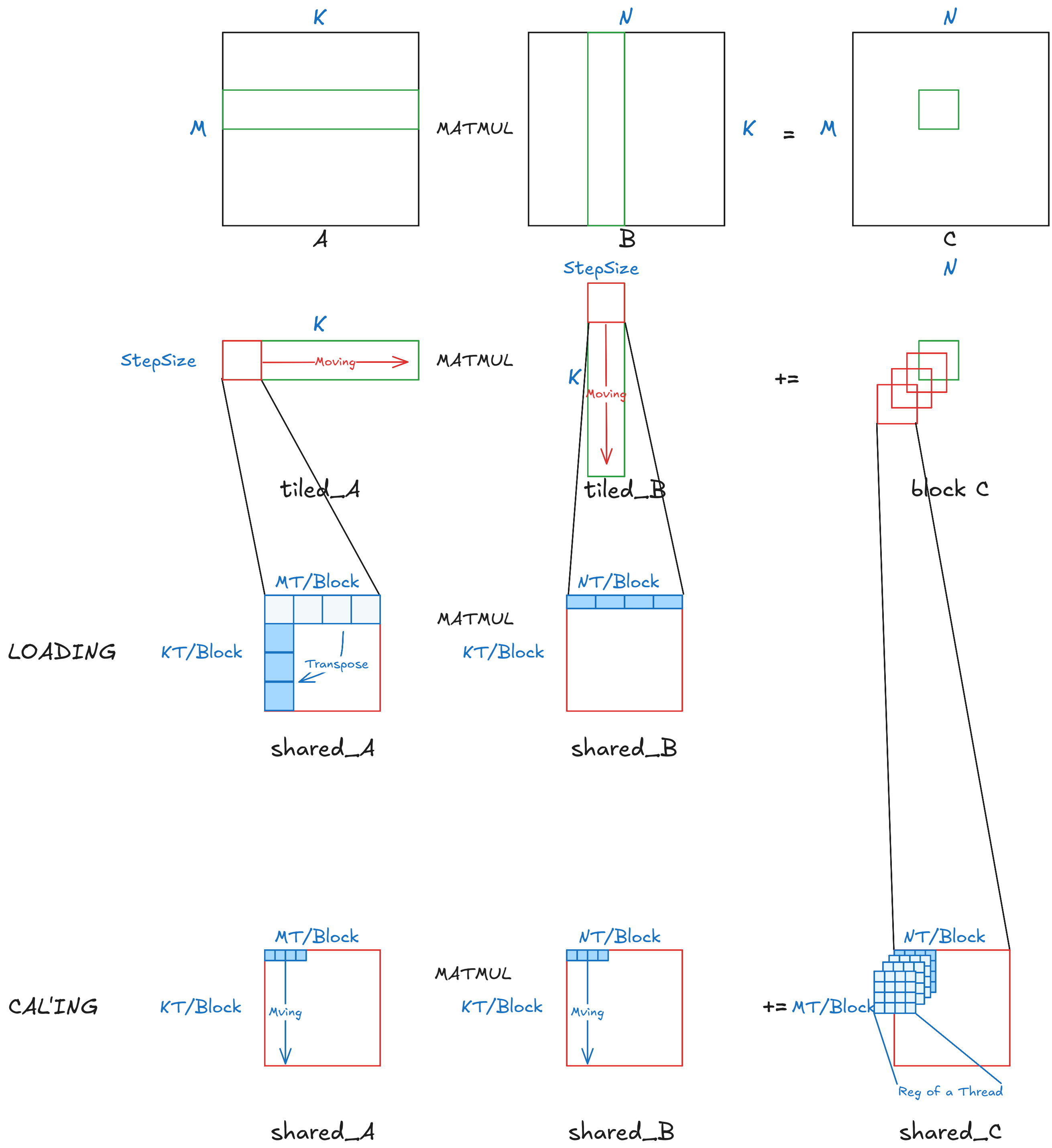
```
#define FETCH_FLOAT4(ptr) (reinterpret_cast<float4 *>(&(ptr))[0])
template <int M_NUM_PER_BLOCK,
          int N_NUM_PER_BLOCK,
          int K_NUM_PER_BLOCK,
          int M_NUM_PER_THREAD,
          int N_NUM_PER_THREAD,
          int K_NUM_PER_THREAD>
__global__ void sgemm5(float *A, float *B, float *C, const int M,
const int N, const int K)
{
```

```
    int tid[2];
    reIndex(tid, N_NUM_PER_BLOCK / N_NUM_PER_THREAD);
    int tid_x = tid[0];
    int tid_y = tid[1];

    float *A_start = A + (blockIdx.y * M_NUM_PER_BLOCK) * K;
    float *B_start = B + (blockIdx.x * K_NUM_PER_BLOCK);

    __shared__ float shared_A[M_NUM_PER_BLOCK][K_NUM_PER_BLOCK];
    __shared__ float shared_B[K_NUM_PER_BLOCK][N_NUM_PER_BLOCK];

    float a_reg[M_NUM_PER_THREAD] = {0.f};
    float b_reg[N_NUM_PER_THREAD] = {0.f};
    float sum[M_NUM_PER_THREAD][N_NUM_PER_THREAD] = {0.f};

    for (int s = 0; s < K; s += K_NUM_PER_BLOCK)
    {
#pragma unroll
        for (int i = 0; i < M_NUM_PER_THREAD; ++i)
        {
            FETCH_FLOAT4(shared_A[tid_y * M_NUM_PER_THREAD + i]
[tid_x * K_NUM_PER_THREAD]) =
                FETCH_FLOAT4(A_start[(tid_y * M_NUM_PER_THREAD + i)
* K + (s + (tid_x * K_NUM_PER_THREAD))]);
        }
#pragma unroll
        for (int i = 0; i < K_NUM_PER_THREAD; ++i)
        {
            FETCH_FLOAT4(shared_B[tid_y * K_NUM_PER_THREAD + i]
[tid_x * N_NUM_PER_THREAD]) =
                FETCH_FLOAT4(B_start[((tid_y * K_NUM_PER_THREAD +
i) + s) * N + (tid_x * N_NUM_PER_THREAD)]);
        }
        __syncthreads();

        for (int k = 0; k < K_NUM_PER_BLOCK; ++k)
        {
            a_reg[0] = shared_A[tid_y * M_NUM_PER_THREAD][k];
            a_reg[1] = shared_A[tid_y * M_NUM_PER_THREAD + 1][k];
            a_reg[2] = shared_A[tid_y * M_NUM_PER_THREAD + 2][k];
            a_reg[3] = shared_A[tid_y * M_NUM_PER_THREAD + 3][k];
            FETCH_FLOAT4(b_reg[0]) = FETCH_FLOAT4(shared_B[k][tid_x
* K_NUM_PER_THREAD]);
#pragma unroll
```

```
            for (int ii = 0; ii < M_NUM_PER_THREAD; ii++)
            {
#pragma unroll
                for (int jj = 0; jj < N_NUM_PER_THREAD; jj++)
                {
                    sum[ii][jj] += a_reg[ii] * b_reg[jj];
                }
            }
        }

        __syncthreads();
    }

    float *C_start = C + blockIdx.y * M_NUM_PER_BLOCK * N +
blockIdx.x * N_NUM_PER_BLOCK;
#pragma unroll
    for (int i = 0; i < M_NUM_PER_THREAD; i++)
    {
        for (int j = 0; j < N_NUM_PER_THREAD; j++)
        {
            C_start[(tid_y * M_NUM_PER_THREAD + i) * N + (tid_x *
N_NUM_PER_THREAD + j)] = sum[i][j];
        }
    }
}
```

sgemm_v6_transpose_A_smem

在上一步中，我们在从shared_A取数据到寄存器中时，存在跨行读取。为了提高访存合并度，我们可以在把数据存入shared_A的时候做转置。



代码：

```
#define FETCH_FLOAT4(ptr) (reinterpret_cast<float4 *>(&(ptr))[0])
template <int M_NUM_PER_BLOCK,
          int N_NUM_PER_BLOCK,
          int K_NUM_PER_BLOCK,
          int M_NUM_PER_THREAD,
          int N_NUM_PER_THREAD,
          int K_NUM_PER_THREAD>
__global__ void sgemm5(float *A, float *B, float *C, const int M,
const int N, const int K)
{
```

```
    int tid[2];
    reIndex(tid, N_NUM_PER_BLOCK / N_NUM_PER_THREAD);
    int tid_x = tid[0];
    int tid_y = tid[1];

    float *A_start = A + (blockIdx.y * M_NUM_PER_BLOCK) * K;
    float *B_start = B + (blockIdx.x * K_NUM_PER_BLOCK);

    __shared__ float shared_A[M_NUM_PER_BLOCK][K_NUM_PER_BLOCK];
    __shared__ float shared_B[K_NUM_PER_BLOCK][N_NUM_PER_BLOCK];

    float a_reg[M_NUM_PER_THREAD] = {0.f};
    float b_reg[N_NUM_PER_THREAD] = {0.f};
    float sum[M_NUM_PER_THREAD][N_NUM_PER_THREAD] = {0.f};

    for (int s = 0; s < K; s += K_NUM_PER_BLOCK)
    {

#pragma unroll
        for (int i = 0; i < M_NUM_PER_THREAD; ++i)
        {
            FETCH_FLOAT4(a_l_reg[0]) =
                FETCH_FLOAT4(A_start[(tid_y * M_NUM_PER_THREAD + i)
* K + (s + (tid_x * K_NUM_PER_THREAD))]);
            shared_A[tid_x * K_NUM_PER_THREAD + 0][tid_y *
M_NUM_PER_THREAD + i] = a_l_reg[0];
            shared_A[tid_x * K_NUM_PER_THREAD + 1][tid_y *
M_NUM_PER_THREAD + i] = a_l_reg[1];
            shared_A[tid_x * K_NUM_PER_THREAD + 2][tid_y *
M_NUM_PER_THREAD + i] = a_l_reg[2];
            shared_A[tid_x * K_NUM_PER_THREAD + 3][tid_y *
M_NUM_PER_THREAD + i] = a_l_reg[3];
        }

#pragma unroll
        for (int i = 0; i < K_NUM_PER_THREAD; ++i)
        {
            FETCH_FLOAT4(shared_B[tid_y * K_NUM_PER_THREAD + i]
[tid_x * N_NUM_PER_THREAD]) =
                FETCH_FLOAT4(B_start[((tid_y * K_NUM_PER_THREAD +
i) + s) * N + (tid_x * N_NUM_PER_THREAD)]);
        }
        __syncthreads();
```

```
        for (int k = 0; k < K_NUM_PER_BLOCK; ++k)
        {
            FETCH_FLOAT4(a_reg[0]) = FETCH_FLOAT4(shared_A[k][tid_y
* N_NUM_PER_THREAD]);
            FETCH_FLOAT4(b_reg[0]) = FETCH_FLOAT4(shared_B[k][tid_x
* K_NUM_PER_THREAD]);

#pragma unroll
            for (int ii = 0; ii < M_NUM_PER_THREAD; ii++)
            {
#pragma unroll
                for (int jj = 0; jj < N_NUM_PER_THREAD; jj++)
                {
                    sum[ii][jj] += a_reg[ii] * b_reg[jj];
                }
            }
        }

        __syncthreads();
    }

    float *C_start = C + blockIdx.y * M_NUM_PER_BLOCK * N +
blockIdx.x * N_NUM_PER_BLOCK;
#pragma unroll
    for (int i = 0; i < M_NUM_PER_THREAD; i++)
    {
        for (int j = 0; j < N_NUM_PER_THREAD; j++)
        {
            C_start[(tid_y * M_NUM_PER_THREAD + i) * N + (tid_x *
N_NUM_PER_THREAD + j)] = sum[i][j];
        }
    }
}
```

sgemm_v7_double_buffer

当前算法以及达到瓶颈了，我们接下来可以通过乒乓方法遮掩访存。对每一个线程，为了减少线程的空闲时间，我们可以让线程在计算当前批次数据的同时去预取下一批次的数据。算法不用大概，只需要手动写出第一个乒和最后一个乓。

```
#define FETCH_FLOAT4(ptr) (reinterpret_cast<float4 *>(&(ptr))[0])
template <int M_NUM_PER_BLOCK,
          int N_NUM_PER_BLOCK,
```

```
          int K_NUM_PER_BLOCK,
          int X_NUM_PER_THREAD,
          int Y_NUM_PER_THREAD>
__global__ void sgemm7(float *A, float *B, float *C, const int M,
const int N, const int K)
{
    // We have different layout for A and B, so we need two sets of
tid for loading
    int atid[2];
    // Divide by 4 as 1 thread could load 4 floats using float4
    reIndex(atid, K_NUM_PER_BLOCK >> 2);
    int atid_x = atid[0];
    int atid_y = atid[1];

    int btid[2];
    reIndex(btid, N_NUM_PER_BLOCK >> 2);
    int btid_x = btid[0];
    int btid_y = btid[1];

    // And another set of tid for computing/C
    int ctid[2];
    reIndex(ctid, N_NUM_PER_BLOCK / X_NUM_PER_THREAD);
    int ctid_x = ctid[0];
    int ctid_y = ctid[1];

    float *A_start = A + (blockIdx.y * M_NUM_PER_BLOCK) * K;
    float *B_start = B + (blockIdx.x * N_NUM_PER_BLOCK);

    // Here we need two shared matrices for ping-pong buffering
    __shared__ float shared_A[2][K_NUM_PER_BLOCK][M_NUM_PER_BLOCK];
    __shared__ float shared_B[2][K_NUM_PER_BLOCK][N_NUM_PER_BLOCK];

    float a_reg[Y_NUM_PER_THREAD] = {0.f};
    float b_reg[X_NUM_PER_THREAD] = {0.f};
    float a_l_reg[4] = {0.f};
    float sum[Y_NUM_PER_THREAD][X_NUM_PER_THREAD] = {0.f};

    // Initial load and sync for the first stage (ping)
    FETCH_FLOAT4(a_l_reg[0]) = FETCH_FLOAT4(A_start[(atid_y * K) +
(atid_x * 4)]);
    shared_A[0][atid_x * 4 + 0][atid_y] = a_l_reg[0];
    shared_A[0][atid_x * 4 + 1][atid_y] = a_l_reg[1];
    shared_A[0][atid_x * 4 + 2][atid_y] = a_l_reg[2];
    shared_A[0][atid_x * 4 + 3][atid_y] = a_l_reg[3];
```

```
    FETCH_FLOAT4(shared_B[0][btid_y][btid_x * 4]) =
FETCH_FLOAT4(B_start[btid_y * N + btid_x * 4]);
    __syncthreads();

    int write_stage_idx = 1;
    for (int s = K_NUM_PER_BLOCK; s < K; s += K_NUM_PER_BLOCK)
    {
        // Load next stage (pong)
        FETCH_FLOAT4(a_l_reg[0]) = FETCH_FLOAT4(A_start[(atid_y *
K) + s + (atid_x * 4)]);
        shared_A[write_stage_idx][atid_x * 4 + 0][atid_y] =
a_l_reg[0];
        shared_A[write_stage_idx][atid_x * 4 + 1][atid_y] =
a_l_reg[1];
        shared_A[write_stage_idx][atid_x * 4 + 2][atid_y] =
a_l_reg[2];
        shared_A[write_stage_idx][atid_x * 4 + 3][atid_y] =
a_l_reg[3];
        FETCH_FLOAT4(shared_B[write_stage_idx][btid_y][btid_x * 4])
= FETCH_FLOAT4(B_start[(btid_y + s) * N + (btid_x * 4)]);

        write_stage_idx ^= 1;
        for (int k = 0; k < K_NUM_PER_BLOCK; ++k)
        {
            FETCH_FLOAT4(a_reg[0]) =
FETCH_FLOAT4(shared_A[write_stage_idx][k][ctid_y * Y_NUM_PER_THREAD
+ 0]);
            FETCH_FLOAT4(a_reg[4]) =
FETCH_FLOAT4(shared_A[write_stage_idx][k][ctid_y * Y_NUM_PER_THREAD
+ 4]);
            FETCH_FLOAT4(b_reg[0]) =
FETCH_FLOAT4(shared_B[write_stage_idx][k][ctid_x * X_NUM_PER_THREAD
+ 0]);
            FETCH_FLOAT4(b_reg[4]) =
FETCH_FLOAT4(shared_B[write_stage_idx][k][ctid_x * X_NUM_PER_THREAD
+ 4]);

            // Unroll the loops to improve performance
#pragma unroll 4
            for (int ii = 0; ii < Y_NUM_PER_THREAD; ii++)
            {
#pragma unroll 4
                for (int jj = 0; jj < X_NUM_PER_THREAD; jj++)
                {
```

```
                    sum[ii][jj] += a_reg[ii] * b_reg[jj];
                }
            }
        }
        __syncthreads();
    }
    // Process the last stage
    write_stage_idx ^= 1;
    for (int k = 0; k < K_NUM_PER_BLOCK; ++k)
    {
        FETCH_FLOAT4(a_reg[0]) =
FETCH_FLOAT4(shared_A[write_stage_idx][k][ctid_y * Y_NUM_PER_THREAD
+ 0]);
        FETCH_FLOAT4(a_reg[4]) =
FETCH_FLOAT4(shared_A[write_stage_idx][k][ctid_y * Y_NUM_PER_THREAD
+ 4]);
        FETCH_FLOAT4(b_reg[0]) =
FETCH_FLOAT4(shared_B[write_stage_idx][k][ctid_x * X_NUM_PER_THREAD
+ 0]);
        FETCH_FLOAT4(b_reg[4]) =
FETCH_FLOAT4(shared_B[write_stage_idx][k][ctid_x * X_NUM_PER_THREAD
+ 4]);

#pragma unroll 4
        for (int ii = 0; ii < Y_NUM_PER_THREAD; ii++)
        {
#pragma unroll 4
            for (int jj = 0; jj < X_NUM_PER_THREAD; jj++)
            {
                sum[ii][jj] += a_reg[ii] * b_reg[jj];
            }
        }
    }

    float *C_start = C + blockIdx.y * M_NUM_PER_BLOCK * N +
blockIdx.x * N_NUM_PER_BLOCK;
#pragma unroll 4
    for (int i = 0; i < Y_NUM_PER_THREAD; i++)
    {
        FETCH_FLOAT4(C_start[(i + ctid_y * Y_NUM_PER_THREAD) * N +
ctid_x * X_NUM_PER_THREAD + 0]) = FETCH_FLOAT4(sum[i][0]);
        FETCH_FLOAT4(C_start[(i + ctid_y * Y_NUM_PER_THREAD) * N +
ctid_x * X_NUM_PER_THREAD + 4]) = FETCH_FLOAT4(sum[i][4]);
```

```
        }
}
```