## key points

Blur image: blur_size leads to the core of the blurring pixel that accumulates other pixels around it.

Matrix multiplication:

inner product:

$$P_{row,col}=\sum M_{row,k}\cdot N_{k,col} fork=0,1,\cdots Width-1 \tag{1}$$

Col-major and row-major: two different ways of showing a matrix's index. A row-major counts in the same row consecutively ($r \cdot Cols + c$), while a col-major counts in columns ($c \cdot Rows + r$) .

As the table 1 shows, the **Bold** is for row-major, *Italic* is for col-major.

Table 1: **Detailed matrix index.**

| | | | |
|---|---|---|---|
| (0,0)**0** *0* | (0,1)**1** *3* | (0,2)**2** *6* | (0,3)**3** *9* |
| (1,0)**4** *1* | (1,1)**5** *4* | (1,2)**6** *7* | (1,3)**7** *10* |
| (2,0)**8** *2* | (2,1)**9** *5* | (2,2)**10** *8* | (2,3)**11** *11* |

## 1 Solutions

### 1.1

(1) Input matrices are A($M \times K$) and B$($K \times N$). Output matrix is C($M \times N$).

Every thread is responsible for calculating one complete row of the output matrix C.

```
__global__ void matmul_row(float* A, float* B, float* C, int M, int N, int
    K) {
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (col < N) {
        for (int row = 0; row < M; ++row) {
            float sum = 0;
            for (int k = 0; k < K; ++k){
                sum += A[row*K + k] * B[k*N + col];
            }
            C[row * N + col] = sum;
        }
    }
}
```

(2)

```
__global__ void matmul_row(float* A, float* B,float* C, int M,int N,int
    K){
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < M) {
        for (int col = 0; col < N; ++col) {
            float sum = 0;
            for (int k = 0;k < K; ++k) {
                sum += A[row * K + k] * B[k * N + col];
            }
            C[row * N + col] = sum;
        }
    }
}
```

(3)

For row-wise Kernel A and C have a better memory access, for col-wise Kernel B has a better memory access.

This shall depend on the matrix dimensions.

Further consideration: Both kernels can be further optimized using tiling to improve data locality and cache utilization. Shared memory and warp shuffling

Explanation:

### 1.1.1 Tiling

```
__global__ void matmul_row_tiled(float* A, float* B, float* C, int M,
    int N, int K) {
     // Shared memory for tiles from A and B
     __shared__ float As[TILE_WIDTH][TILE_WIDTH];
     __shared__ float Bs[TILE_WIDTH][TILE_WIDTH];

     int row = blockIdx.x * blockDim.x + threadIdx.x;
     int col = blockIdx.y * blockDim.y + threadIdx.y;

     float sum = 0;

     // Loop over tiles, where K is a multiple of TILE_WIDTH
     for (int tile = 0; tile < K / TILE_WIDTH; ++tile) {
          // Load tiles from global to shared memory
          int tile_k = tile * TILE_WIDTH;
          As[threadIdx.y][threadIdx.x] = A[row * K + tile_k + threadIdx.x
              ];
          Bs[threadIdx.y][threadIdx.x] = B[(tile_k + threadIdx.y) * N +
              col];

          // Synchronize threads within the block to ensure tiles are
              loaded
          __syncthreads();

          // Compute partial sum for the current tile
          for (int k = 0; k < TILE_WIDTH; ++k) {
               sum += As[threadIdx.y][k] * Bs[k][threadIdx.x];
          }

          // Synchronize again before loading the next tile
          __syncthreads();
     }

     // Store the final result in the output matrix
     if (row < M && col < N) {
          C[row * N + col] = sum;
     }
}
```

Compare:

Without tiling: Each thread calculates one element of the output C, which should load rows of A and cols of B from global memory leading to a high number of global memory access.

With tiling: Each thread block cooperatively computes a small tile of the output matrix. All threads within the block can access this data repeatedly from shared memory.

Tiling helps decreasing exchange between global memory and local memory.

## 1.2

Computes each element of the output vector as the dot product, manages memory allocation,data transfer, kernel invocation, and retrives the result.

```
#include <cuda_runtime.h>
#include <iostream>
#include <vector>

// CUDA Kernel for Matrix-Vector Multiplication
```

```
__global__ void matrixVectorMulKernel(const float* B, const float* C,
    float* A, int N) {// input matrix pointers and number of rows/
    columns in the square matrix.
     int row = blockIdx.x * blockDim.x + threadIdx.x;
     if (row < N) {// number of row that over N won't work.
         float sum = 0.0f;
         for (int j = 0; j < N; ++j) {
             sum += B[row * N + j] * C[j];//the dot product of the
                 corresponding row in matrix B and vector C.
         }
         A[row] = sum;
     }
     //each thread computes one element of the output vector A.
}

// Host Function for Matrix-Vector Multiplication
void matrixVectorMul(const float* h_B, const float* h_C, float* h_A,
    int N) {// host input matrix pointers and N.
     float *d_B = nullptr, *d_C = nullptr, *d_A = nullptr;
     size_t sizeMatrix = N * N * sizeof(float);
     size_t sizeVector = N * sizeof(float);
     // Allocate device memory
     //error checking.
     cudaError_t err = cudaMalloc((void**)&d_B, sizeMatrix);
     if (err != cudaSuccess) {
         std::cerr << "Failed to allocate device memory for matrix B (
             error code "
                   << cudaGetErrorString(err) << ")!\n";
         exit(EXIT_FAILURE);
     }

     err = cudaMalloc((void**)&d_C, sizeVector);
     if (err != cudaSuccess) {
         std::cerr << "Failed to allocate device memory for vector C (
             error code "
                   << cudaGetErrorString(err) << ")!\n";
         cudaFree(d_B);
         exit(EXIT_FAILURE);
     }

     err = cudaMalloc((void**)&d_A, sizeVector);
     if (err != cudaSuccess) {
         std::cerr << "Failed to allocate device memory for vector A (
             error code "
                   << cudaGetErrorString(err) << ")!\n";
         cudaFree(d_B);
         cudaFree(d_C);
         exit(EXIT_FAILURE);
     }

     // Copy data from host to device
     err = cudaMemcpy(d_B, h_B, sizeMatrix, cudaMemcpyHostToDevice);
     if (err != cudaSuccess) {
         std::cerr << "Failed to copy matrix B from host to device (
             error code "
                   << cudaGetErrorString(err) << ")!\n";
         cudaFree(d_B);
         cudaFree(d_C);
```

```
        cudaFree(d_A);
        exit(EXIT_FAILURE);
    }

    err = cudaMemcpy(d_C, h_C, sizeVector, cudaMemcpyHostToDevice);
    if (err != cudaSuccess) {
        std::cerr << "Failed to copy vector C from host to device (
           error code "
                  << cudaGetErrorString(err) << ")!\n";
        cudaFree(d_B);
        cudaFree(d_C);
        cudaFree(d_A);
        exit(EXIT_FAILURE);
    }

    // Launch the CUDA kernel
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
    matrixVectorMulKernel<<<blocksPerGrid, threadsPerBlock>>>(d_B, d_C,
        d_A, N);

    // Check for any kernel launch errors
    err = cudaGetLastError();
    if (err != cudaSuccess) {
        std::cerr << "Failed to launch matrixVectorMulKernel (error
           code "
                  << cudaGetErrorString(err) << ")!\n";
        cudaFree(d_B);
        cudaFree(d_C);
        cudaFree(d_A);
        exit(EXIT_FAILURE);
    }

    // Copy the result vector A back to host
    err = cudaMemcpy(h_A, d_A, sizeVector, cudaMemcpyDeviceToHost);

    if (err != cudaSuccess) {
        std::cerr << "Failed to copy vector A from device to host (
           error code "
                  << cudaGetErrorString(err) << ")!\n";
        cudaFree(d_B);
        cudaFree(d_C);
        cudaFree(d_A);
        exit(EXIT_FAILURE);
    }

    // Free device memory
    cudaFree(d_B);
    cudaFree(d_C);
    cudaFree(d_A);
}

int main() {
    // Define the size of the matrix and vectors
    int N = 1024; // Example size; can be modified as needed

    // Initialize host vectors and matrix
    std::vector<float> h_B(N * N, 1.0f); // Initialize all elements to
        1.0
    std::vector<float> h_C(N, 1.0f);     // Initialize all elements to
        1.0
    std::vector<float> h_A(N, 0.0f);     // Output vector

    // Perform matrix-vector multiplication
    matrixVectorMul(h_B.data(), h_C.data(), h_A.data(), N);
    //A = sum_{j}(B[i][j]*C[i])

    // Optional: Verify the result (since B and C are all ones(the
        initial value), A should be filled with N)
    bool correct = true;
    for (int i = 0; i < N; ++i) {
        if (h_A[i] != static_cast<float>(N)) {
            correct = false;
            std::cerr << "Mismatch at index " << i << ": " << h_A[i] <<
                " != " << N << "\n";
            break;
        }
    }

    if (correct) {
        std::cout << "Matrix-vector multiplication successful. All
            elements are " << N << ".\n";
    } else {
        std::cout << "Matrix-vector multiplication failed.\n";
    }

    return 0;
}
```

## 1.3

**I'm a foolish.**

a.~~32~~ 512

The number of threads per block is multiple of bd: 16·32 = 512. Whole block contains M,N for mapping 2D matrix.

b.~~16·32~~ 48640

gridDim.x = (N - 1)/16 + 1 = (300 - 1)/16 + 1 = 19.

gridDim.y = (M - 1)/32 + 1 = 5.

So the number of all blocks is 19·5 = 95. All threads is 95*512 = 48640.

c. ~~$16 \cdot 32 \cdot [(N-1)/16+1] \cdot [(M-1)/32+1]$~~95 as b solved.

d.150·30

Directly multiple M and N(only row¡M and col¡N).

## 1.4

Well, I messed up with threads calculation and r/c major definiation.

a.~~$20 \cdot 500$~~ for row-major order, it should be $20 \cdot 400 + 10 = 8010$

b.~~$400 \cdot 50$~~ for col-major order, it should be $10 \cdot 500 + 20 = 5020$

(As the Formula in key points showed)

## 1.5

**I'm still a little confusion on 3 dimensions.**

~~$10 \cdot 400 + 20 \cdot 500 + 5$~~

In 3D, x is colums, y is rows and z is the depth. So, for a row-major, it should be $x * height * width + y * width + z$. Then the result is $(5 \cdot 500 \cdot 400) + (20 \cdot 400) + 10$
Besides, for a col-major, it shoule be $z * height * width + y * width + x$.