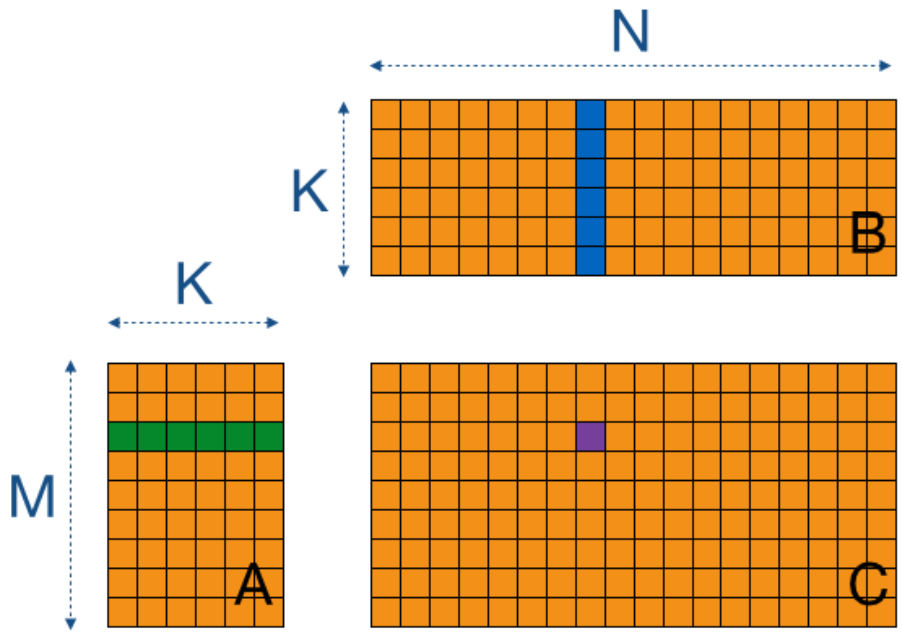
Matrix multiplication

Assignment 2

Milan Cimbaljević

1.Naive approach

To get an element of the (i,j) of the resulting we multiply i-th row of first matrix and j-th column of second matrix. Then we add elements of the resulting vector. We do this for every element.



CPU code:

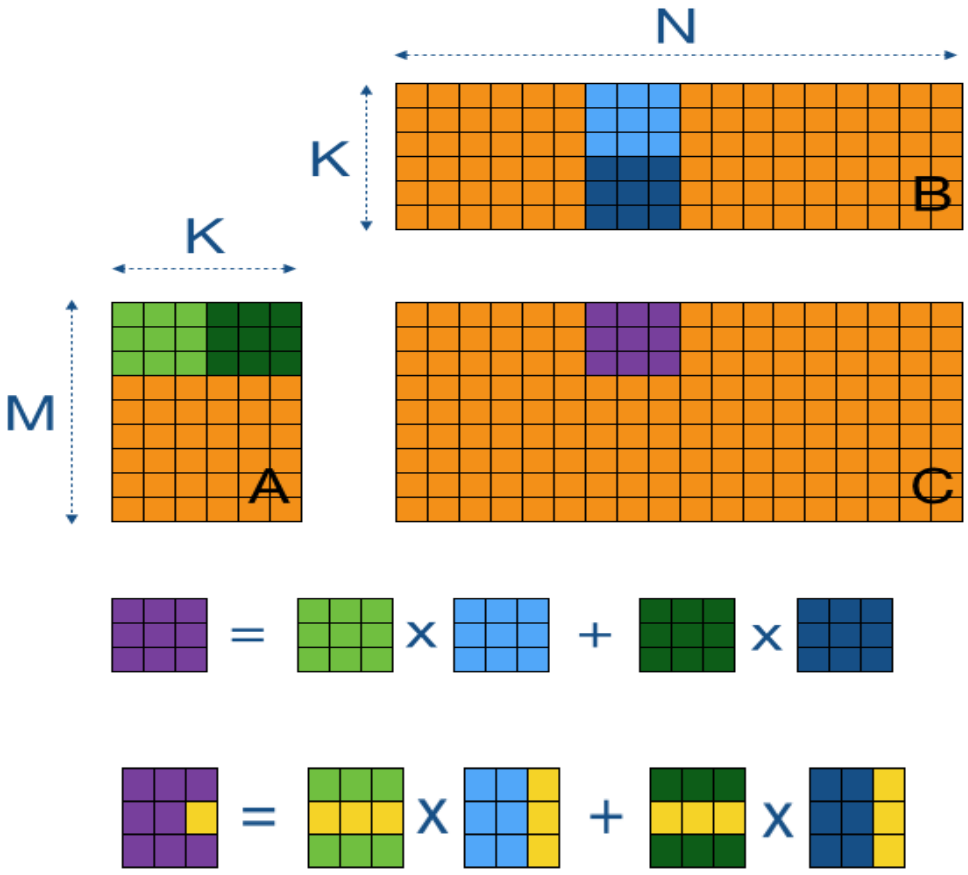
```
void matrix_multipication(float* A, float* B, float* C, int M, int K, int N) {

	for (int m = 0; m < M; m++) {
		for (int n = 0; n < N; n++) {
			float acc = 0.0f;
			for (int k = 0; k < K; k++) {
				acc += A[m * K + k] * B[k * N + n];
			}
			C[m * N + n] = acc;
		}
	}
}
```

GPU kernel:

```
__kernel void matrix_multiplication(__global const float *A, __global const float* B,
__global float* C, int m, int n, int k) {

	const int tx = get_global_id(0);
	const int ty = get_global_id(1);

	// k = width of A and height of B
	// n = width of B

	float acc = 0.0f;
	for (int cnt = 0; cnt < k; cnt++) {
		acc += A[ty * k + cnt] * B[cnt*n + tx];
	}

	C[ty * n + tx] = acc;
}
```

We are going to have M*N*K multiplications and M*N*(K-1) additions. So we are going to have around 2*M*N*K operations.

2. Tiling in the local memory



GPU kernel:

```
__kernel void matrix_multiplication(__global const float* A, __global const float* B,
__global float* C, int m, int n, int k) {

	const int tx = get_global_id(0);
	const int ty = get_global_id(1);
	const int lx = get_local_id(0);
	const int ly = get_local_id(1);

	__local float A_sub[32][32];
	__local float B_sub[32][32];

	float acc = 0.0f;
	for (int currentTile = 0; currentTile < k / 32; currentTile++) {
		A_sub[ly][lx] = A[ty*k + currentTile * 32 + lx];
		B_sub[ly][lx] = B[(currentTile * 32 + ly) * n + tx];

		barrier(CLK_LOCAL_MEM_FENCE);

		for (int cnt = 0; cnt < 32; cnt++) {
			acc += A_sub[ly][cnt] * B_sub[cnt][lx];
		}

		barrier(CLK_LOCAL_MEM_FENCE);
	}

	C[ty * n + tx] = acc;
}
```

We section matrices into tiles, so each tile can be loaded into the local memory and shared between threads in the same workgroup.

In the naive approach we used to have 2*k loads from the global memory per thread, in this case we have 2*(k/32) loads from the global memory per thread. Each tile will be in the local memory. So each access will be much faster.