Amirkabir University of Technology

(Tehran Polytechnic)

Department of Computer Engineering,
Multicore Programming (Spring 2024),
Assignment 5

Optimizing a matrix multiplication kernel step by step

Step Zero: Naïve approach

Below you will see a naïve implementation of a general matrix multiplication kernel as defined in the Basic Linear Algebra Subprograms specification. Given matrices A, B, C and scalars α and β, this kernel computes $C = \alpha(AB) + \beta C$. Note that x is a row of the result matrix and y is the column. For all the following exercises, you can assume that M, N and K are multiples of 32.

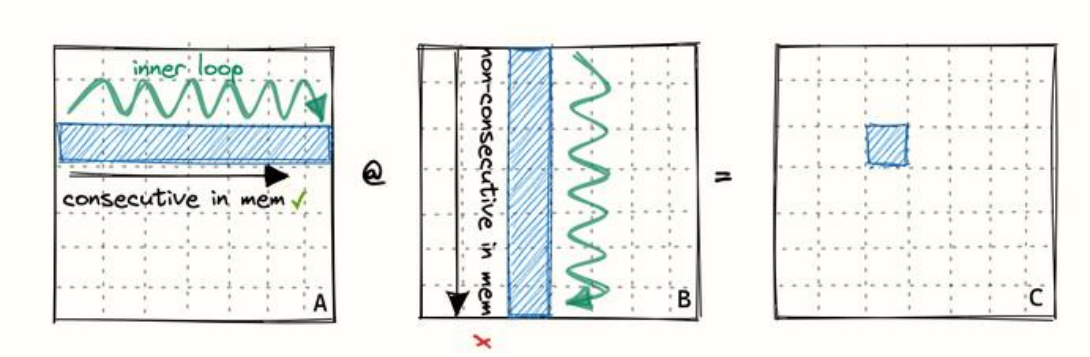
```
__global__ void sgemm_naive(int M, int N, int K, float alpha, const float * A,
    const float * B, float beta, float * C) {
    // compute position in C that this thread is responsible for
    const uint x = blockIdx.x * blockDim.x + threadIdx.x;
    const uint y = blockIdx.y * blockDim.y + threadIdx.y;

    // `if` condition is necessary for when M or N aren't multiples of 32.
    if (x < M && y < N) {
        float tmp = 0.0;
        for (int i = 0; i < K; ++i) {
            tmp += A[x * K + i] * B[i * N + y];
        }
        // C = α*(A@B)+β*C
        C[x * N + y] = alpha * tmp + beta * C[x * N + y];
    }
}
```

And the code calling the kernel:

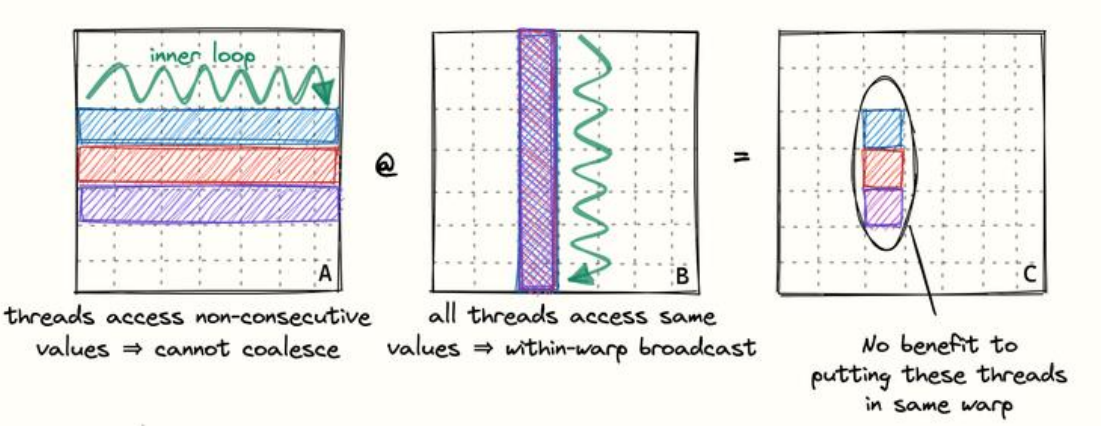
```
// create as many blocks as necessary to map all of C
dim3 gridDim(CEIL_DIV(M, 32), CEIL_DIV(N, 32), 1);
// 32 * 32 = 1024 thread per block
dim3 blockDim(32, 32, 1);
// launch the asynchronous execution of the kernel on the device
// The function call returns immediately on the host
sgemm_naive <<<gridDim, blockDim>>>(M, N, K, alpha, A, B, beta, C);
```

Step one: Coalesced memory access

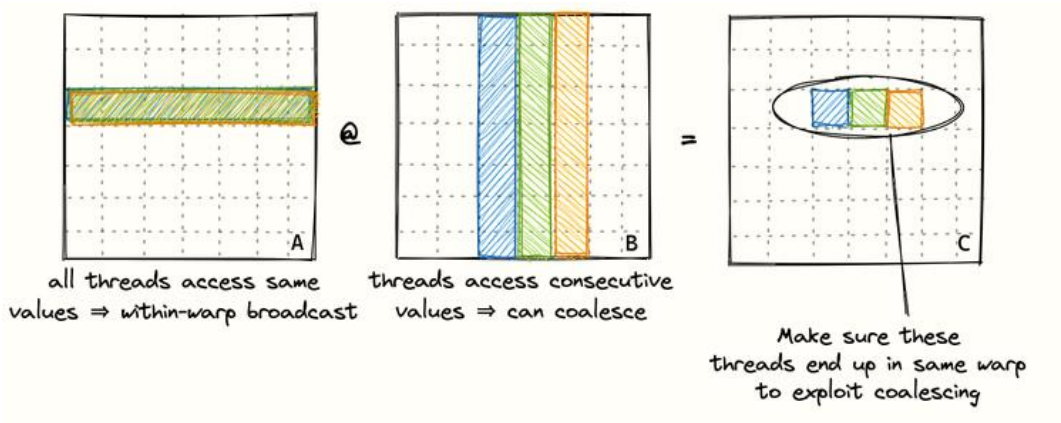
The memory access in the naïve kernel is incredibly inefficient and results in very low performance. Since we're doing output decomposition, let's have a look to see how the access pattern for computing a single index of the C matric looks like:



Now, let's visualize three consecutive threads in a single warp. They only differ in their x variables so they're computing three rows in a single column of the result matrix. The access pattern would be like this:



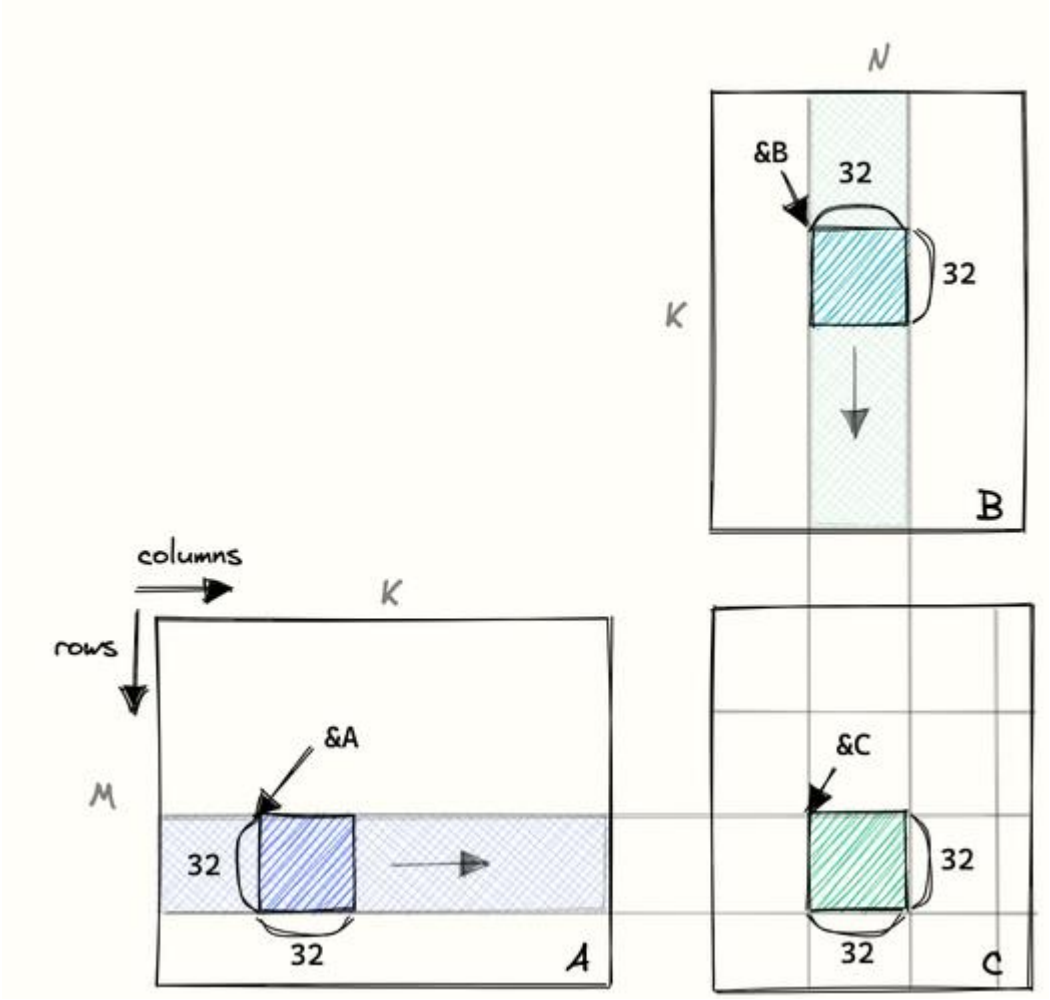
If we instead change our order of computation to have each warp compute a row of the result matrix instead, we could have much better memory access patterns:



Without using shared memory, change the naïve code to use the improved memory access pattern. (You only need to change how the x and y variables are computed)

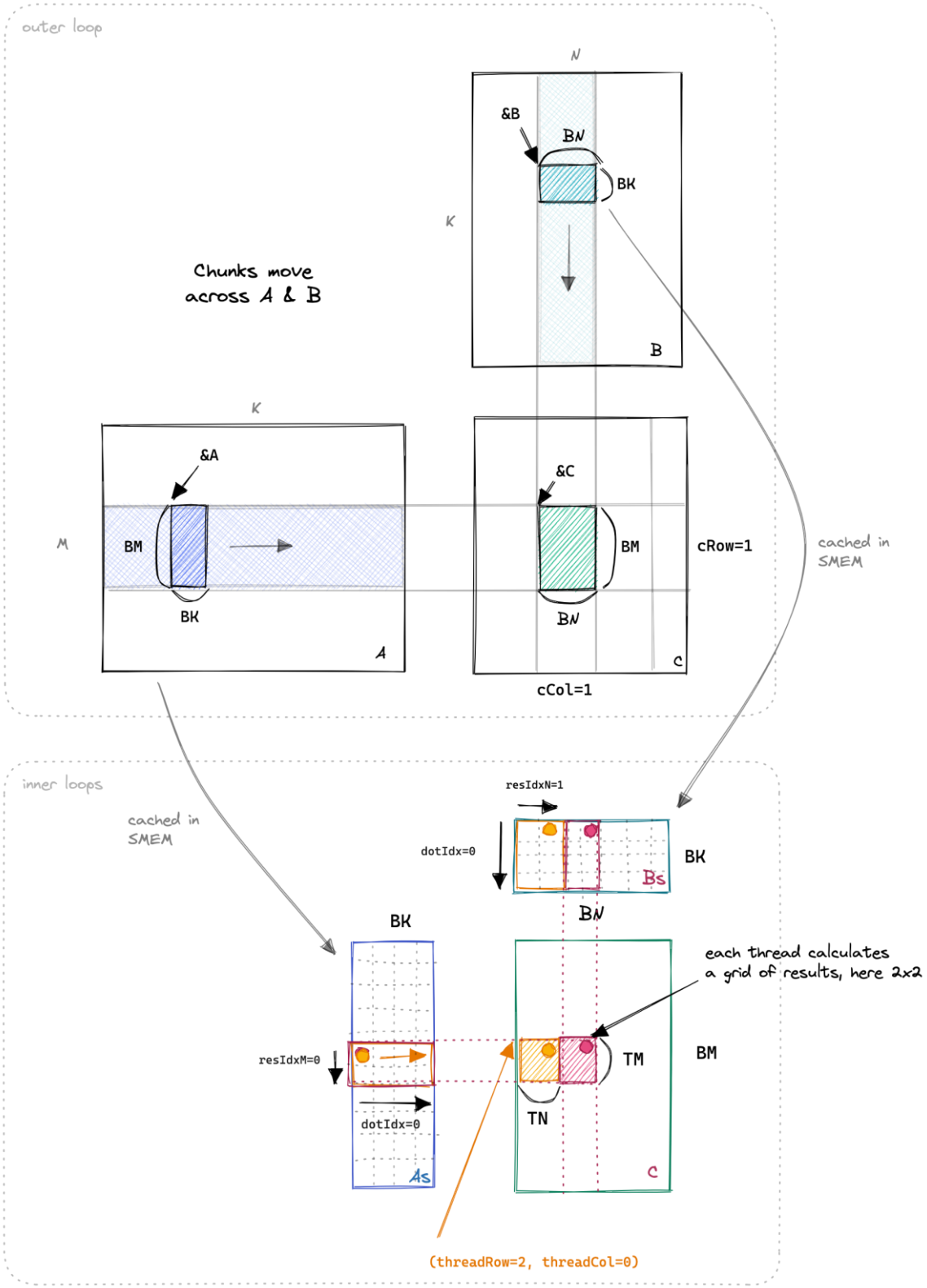
Step Two: Shared Memory Cache-Blocking

Right now, we're only utilizing the global memory. While GPUs do have the traditional cache, they also have shared memory which is orders of magnitude faster than the global memory. Allocate a 32x32 block of shared memory for the A and B matrices and compute the result block by block utilizing the shared memory for each block.



Step Three: Increasing Arithmetic Intensity via 2D Blocktiling

The arithmetic intensity of our kernels thus far has been not very good. We can improve this by making each thread compute more than one part of the result. Change the kernel such that each thread computes the temporary results for a 8x8 section of the blocks in shared memory. Change shared memory block dimensions to BK=8, BM=128 and BN=128.



Deliverables:

You should upload a zip file named STUDENTID_HW6.zip containing:

- Code for each of the steps (including step zero)
- Benchmark results, showing how much time each kernel takes to compute a 2048x2048 matrix
- Roofline analysis result of each kernel