Massively Parallel Computing

Lecture 5: Tiled Matrix Multiplication with Boundary Conditions

Acknowledgement

A lot of contents in this course are referred to the following sources. We deeply appreciate their effort and sharing. We promise not to use the contents for any commercial purpose.

1. Course materials of "Heterogeneous Parallel Programming", University of Illinois at Urbana-Champaign, Wen-mei W. Hwu, (www.cousera.org)

2. Course materials of "Massively Parallel Processors with CUDA", Stanford University, (iTunes University)

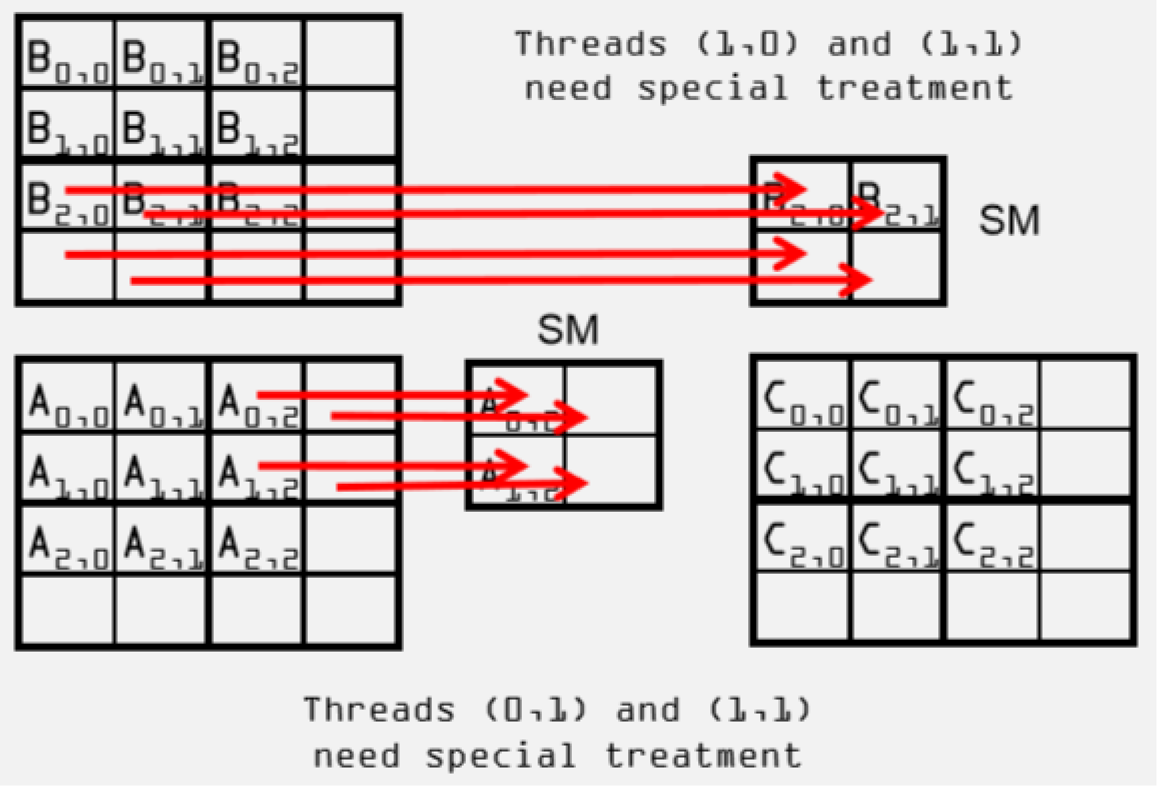
3. Course materials of "GPU Programming for High Performance Computing", University of North Carolina at Charlotte, Barry Wilkinson

4. Overheads of text book, CUDA by examples

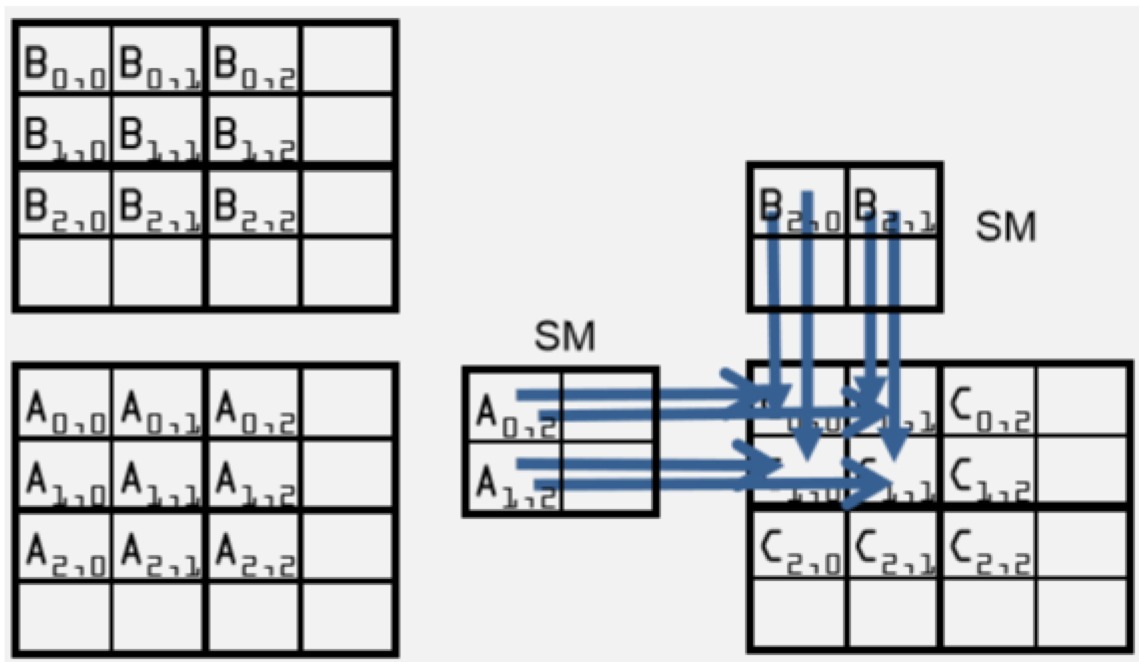
Handling Matrix of Arbitrary Size

- The tiled matrix multiplication kernel can handle only the matrices whose dimensions are multiples of the tile width
  - However, real applications need to handle arbitrary sized matrices.
  - One could pad (add elements to) the rows and columns into multiples of the tile size, but would have significant space and data transfer time overhead.
- We will take a different approach

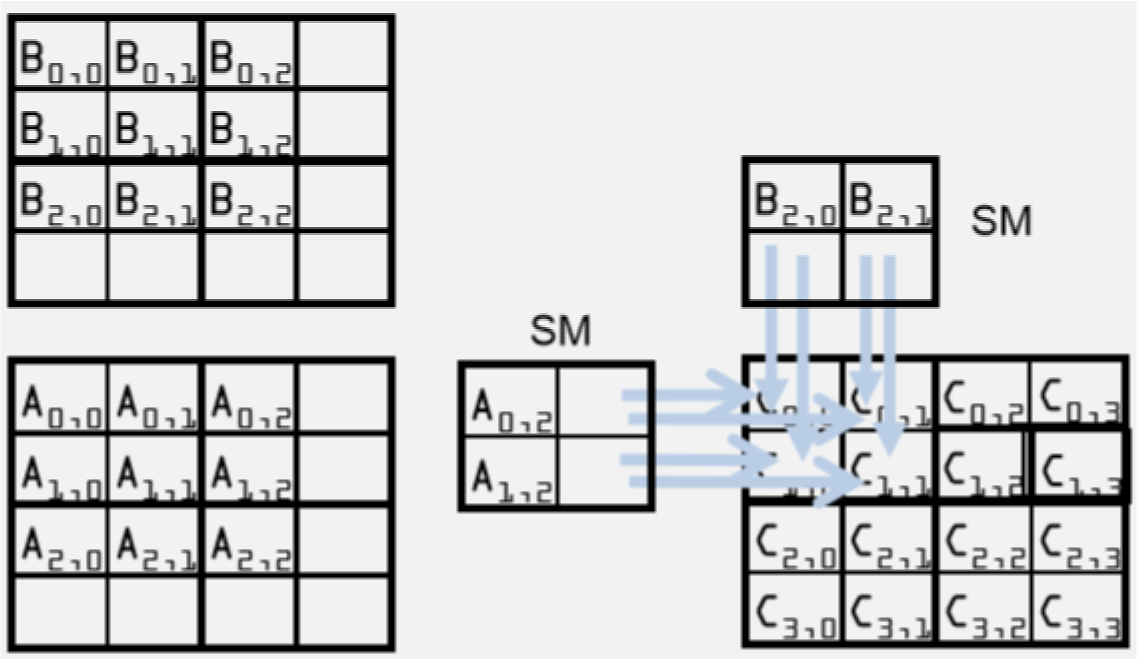
Phase 1 load for Block (0,0) for a 3x3 example



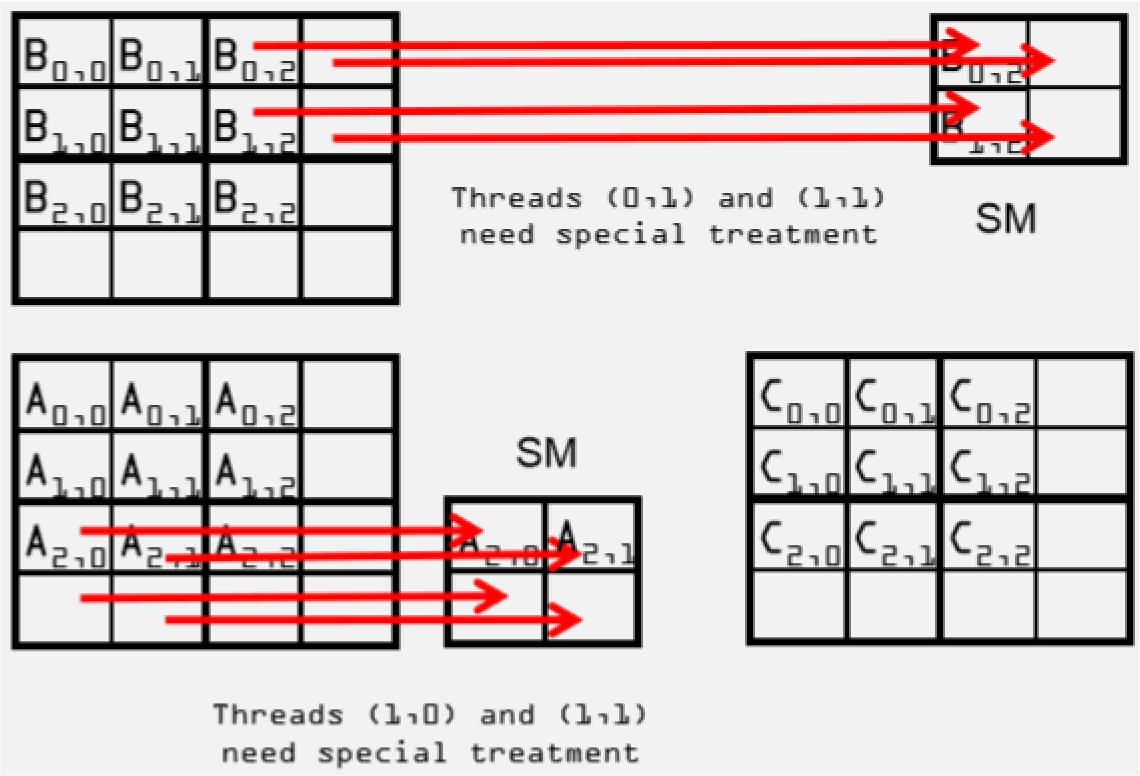
Phase 1 Use for Block (0,0) (iteration 0)



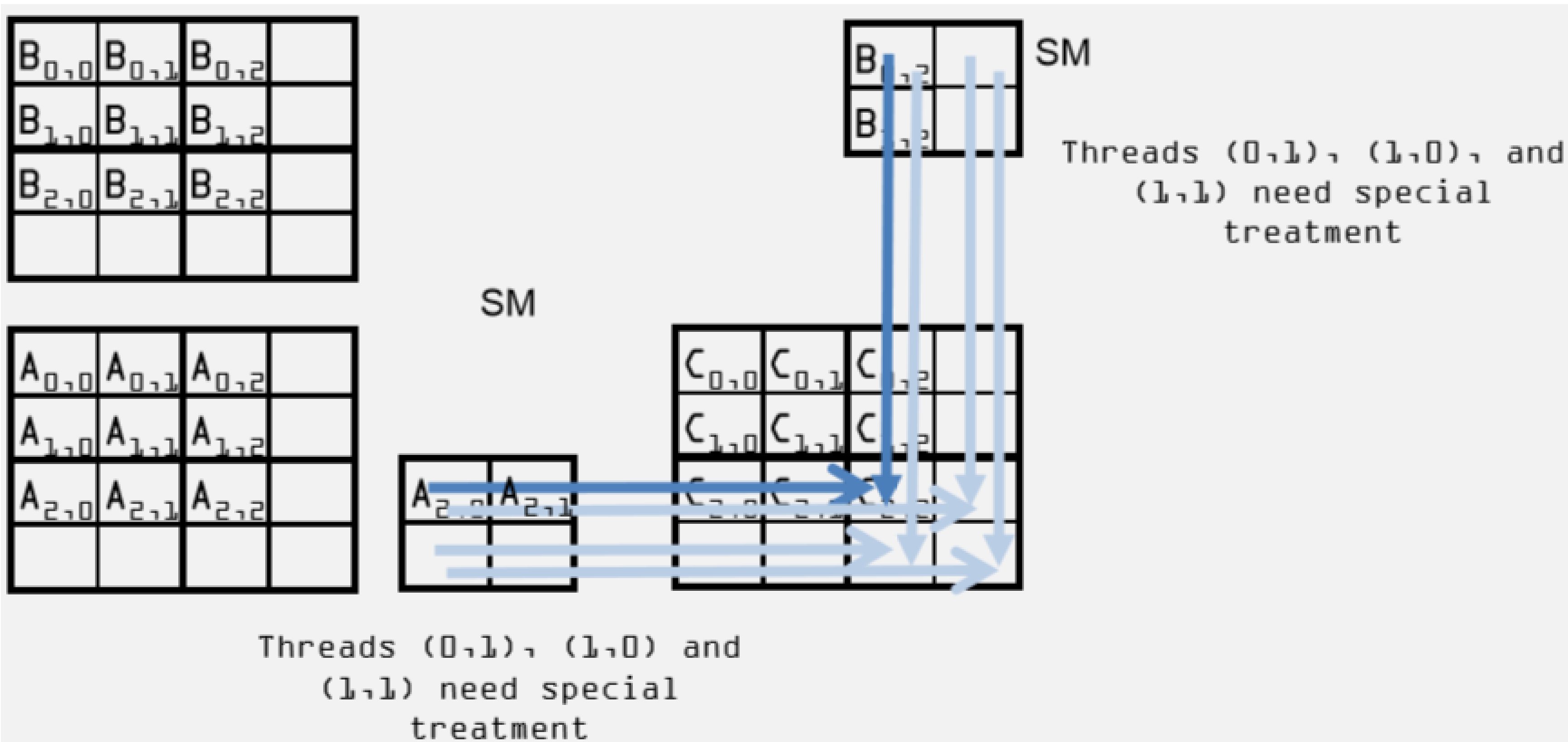
Phase 1 Use for Block (0,0) (iteration 1)



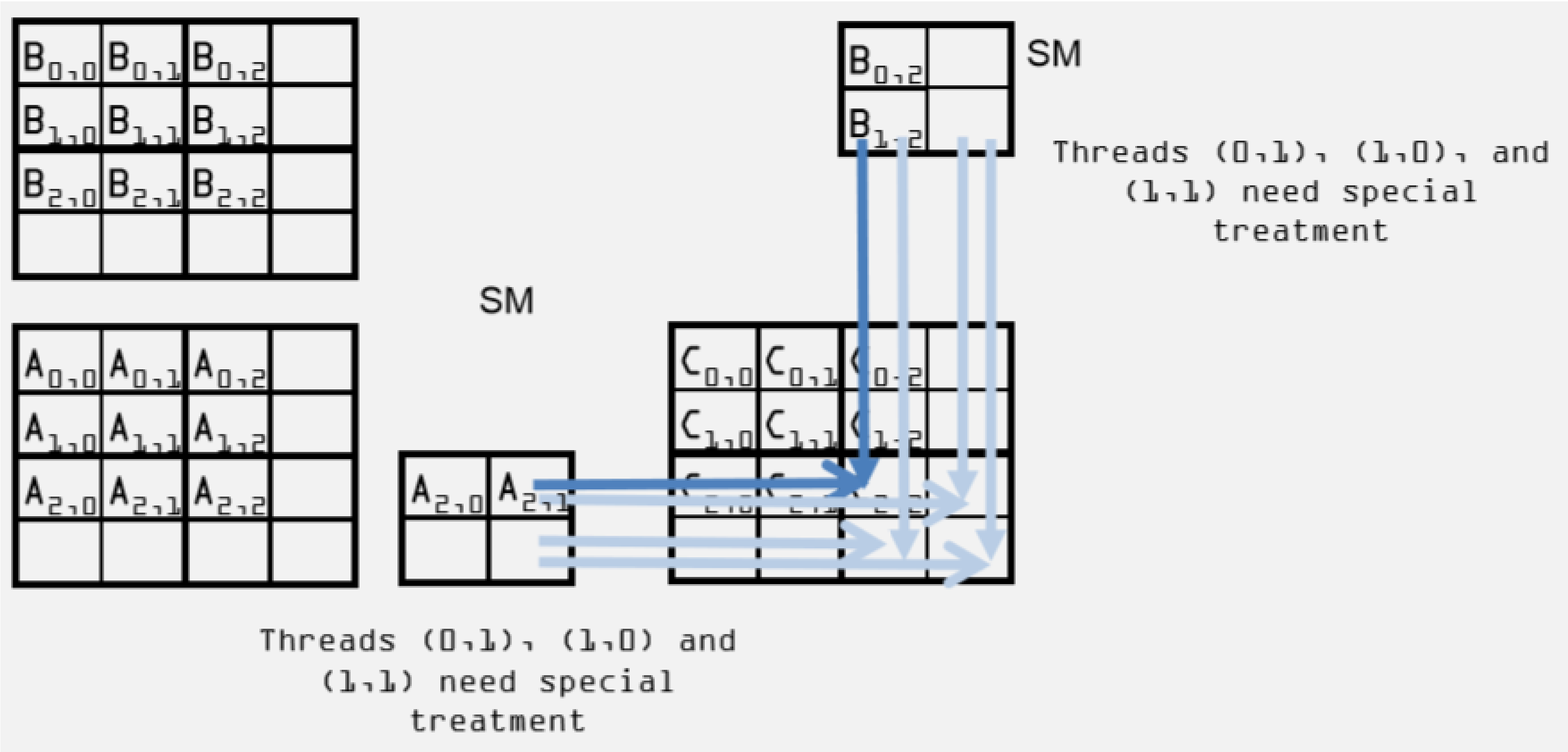
Phase 0 Load for Block (1,1) for a 3x3 example



Phase 0 use for Block(1,1) iteration 0



Phase 0 use for Block(1,1) iteration 1

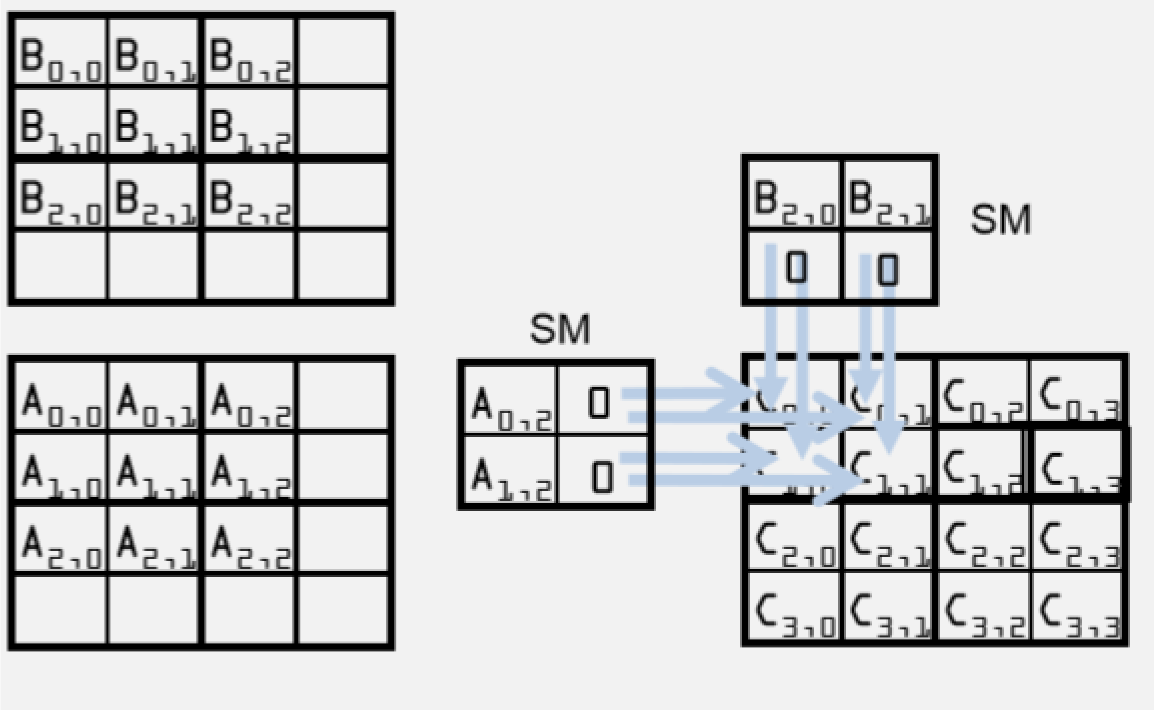


A “Simple” Solution

- When a thread is to load any input element, test if it is in the valid index range
  - If valid, proceed to load
  - Else, do not load, just write a 0

- Rationale: a 0 value will ensure that that the multiply-add step does not affect the final value of the output element

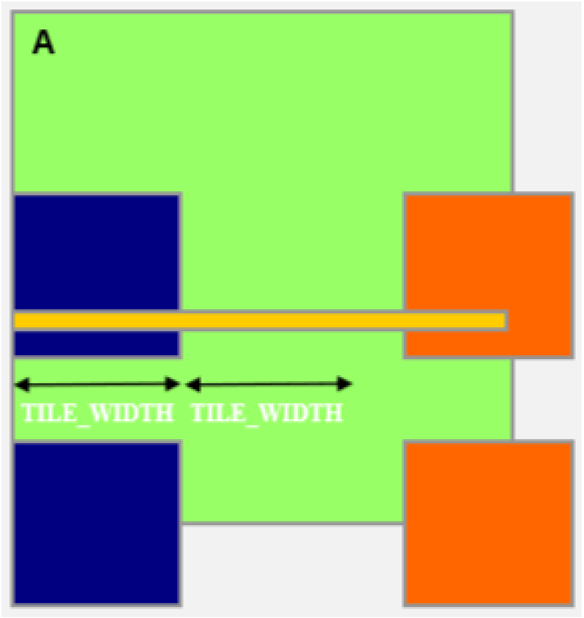
Phase 1 use for Block(0,0) iteration 1



The multiply-add will not affect the output due to 0's.

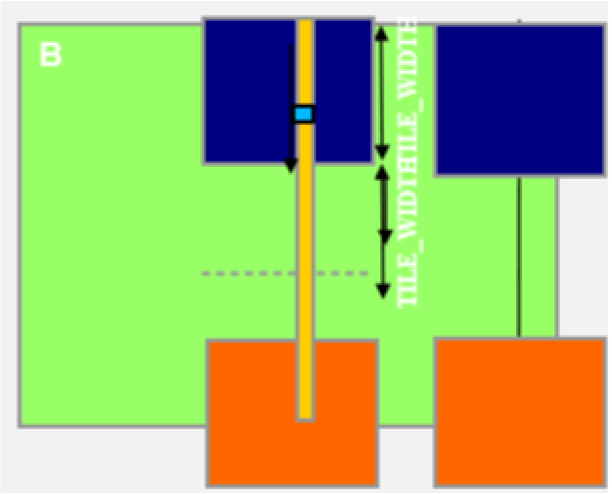
Boundary Condition for Input A Tile

- Each thread loads
  - A[Row][t*TILE_WIDTH+tx]
  - A[Row*Width + t*TILE_WIDTH+tx]
- Need to test
  - (Row < m) && (t*TILE_WIDTH+tx < n)
  - If true, load A element
  - Else , load 0



Boundary Condition for Input B Tile

- Each thread loads
  - B[t*TILE_WIDTH+ty][Col]
  - B[(t*TILE_WIDTH+ty)*k+ Col]
- Need to test
  - (t*TILE_WIDTH+ty < n) && (Col< k)
  - If true, load B element
  - Else , load 0



A "Simple" Solution (Q)

```
__global__ void MatrixMulOnDeviceWithSM(int m, int n, int k, float* A, float* B, float* C)
{
        __shared__ float ds_A[TILE_WIDTH][TILE_WIDTH];
        __shared__ float ds_B[TILE_WIDTH][TILE_WIDTH];
        int bx = blockIdx.x; int by = blockIdx.y;
        int tx = threadIdx.x; int ty = threadIdx.y;
        int Row = by * blockDim.y + ty;
        int Col = bx * blockDim.x + tx;

        float Cvalue = 0;
        for (int t = 0; t < (n-1)/TILE_WIDTH+1; ++t) {// iterate over phases
                // load A and B tiles into shared memory
                ds_A[ty][tx] = .........
                ds_B[ty][tx] = ........
                __syncthreads();

                for (int i = 0; i < TILE_WIDTH; ++i)
                        Cvalue += ds_A[ty][i] * ds_B[i][tx];
                __syncthreads();
        }
        if ( ....  )
                C[Row*k+Col] = Cvalue;
}
```

Codes (Q)

```
#define LEN_M (2*1024+3)
#define LEN_N (2*1024+3)
#define LEN_K (1*1024+3)
#define TILE_WIDTH 32

int main()
{
   // Allocate and initialize the matrices A, B, C
   float * A, *B, *C, *D;
   clock_t start, end;

   A = (float*) malloc( LEN_M*LEN_N*sizeof(float) );
   B = (float*) malloc( LEN_N*LEN_K*sizeof(float) );
   C = (float*) malloc( LEN_M*LEN_K*sizeof(float) );
   D = (float*) malloc( LEN_M*LEN_K*sizeof(float) );

   for( int i=0 ; i<LEN_M*LEN_N ; i++ ) A[i] = i%3;
   for( int i=0 ; i<LEN_N*LEN_K ; i++ ) B[i] = i%4;
   for( int i=0 ; i<LEN_M*LEN_K ; i++ ) C[i] = 0.0;
   for( int i=0 ; i<LEN_M*LEN_K ; i++ ) D[i] = 0.0;

   // I/O to read the input matrices A and B
   float * dev_A, * dev_B, * dev_C;
   HANDLE_ERROR( cudaMalloc( (void**)&dev_A, LEN_M*LEN_N*sizeof(float)));
   HANDLE_ERROR( cudaMalloc( (void**)&dev_B, LEN_N*LEN_K*sizeof(float)));
   HANDLE_ERROR( cudaMalloc( (void**)&dev_C, LEN_M*LEN_K*sizeof(float)));

   HANDLE_ERROR( cudaMemcpy( dev_A, A, LEN_M*LEN_N*sizeof(float)
      , cudaMemcpyHostToDevice ));
   HANDLE_ERROR( cudaMemcpy( dev_B, B, LEN_N*LEN_K*sizeof(float)
      , cudaMemcpyHostToDevice ));

   start = clock();
   // A*B on the device
   dim3 dimGrid( (LEN_K-1)/TILE_WIDTH+1, (LEN_M-1)/TILE_WIDTH+1 );
   dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
   MatrixMulOnDeviceWithSM<<<dimGrid, dimBlock>>>
                  ( LEN_M, LEN_N, LEN_K, dev_A, dev_B, dev_C );

   cudaDeviceSynchronize();
   end = clock();

   // I/O to write the output matrix C
   cudaMemcpy( C, dev_C, LEN_M*LEN_K*sizeof(long), cudaMemcpyDeviceToHost );

   printf("kernel execution time: %f sec\n", (float)(end-start)/CLOCKS_PER_SEC );

   MatrixMulOnHost( LEN_M, LEN_N, LEN_K, A, B, D );

   printf("Check\n");
   for( int i=0 ; i<LEN_M*LEN_K ; i++ ){
      if( C[i] != D[i] ) printf("Error! i=%d, C:%d, D:%d\n", i, C[i], D[i] );
   }
   printf("Done\n");

   // Free matrices A, B, C
   HANDLE_ERROR( cudaFree(dev_A) );
   HANDLE_ERROR( cudaFree(dev_B) );
   HANDLE_ERROR( cudaFree(dev_C) );

   free(A);
   free(B);
   free(C);
   return 0;
}

void MatrixMulOnHost(int m, int n, int k, float * A, float * B, float * C)
{
   for (int Row = 0; Row < m; ++Row){
      for (int Col = 0; Col < k; ++Col) {
         float sum = 0;
         for (int i = 0; i < n; ++i) {
            float a = A[Row * n + i];
            float b = B[Col+i*k];
            sum += a * b;
         }
         C[Row * k + Col] = sum;
      }
   }
   printf("end of matrixMulOnHost\n");
}
```