Chapter-5
Performance considerations

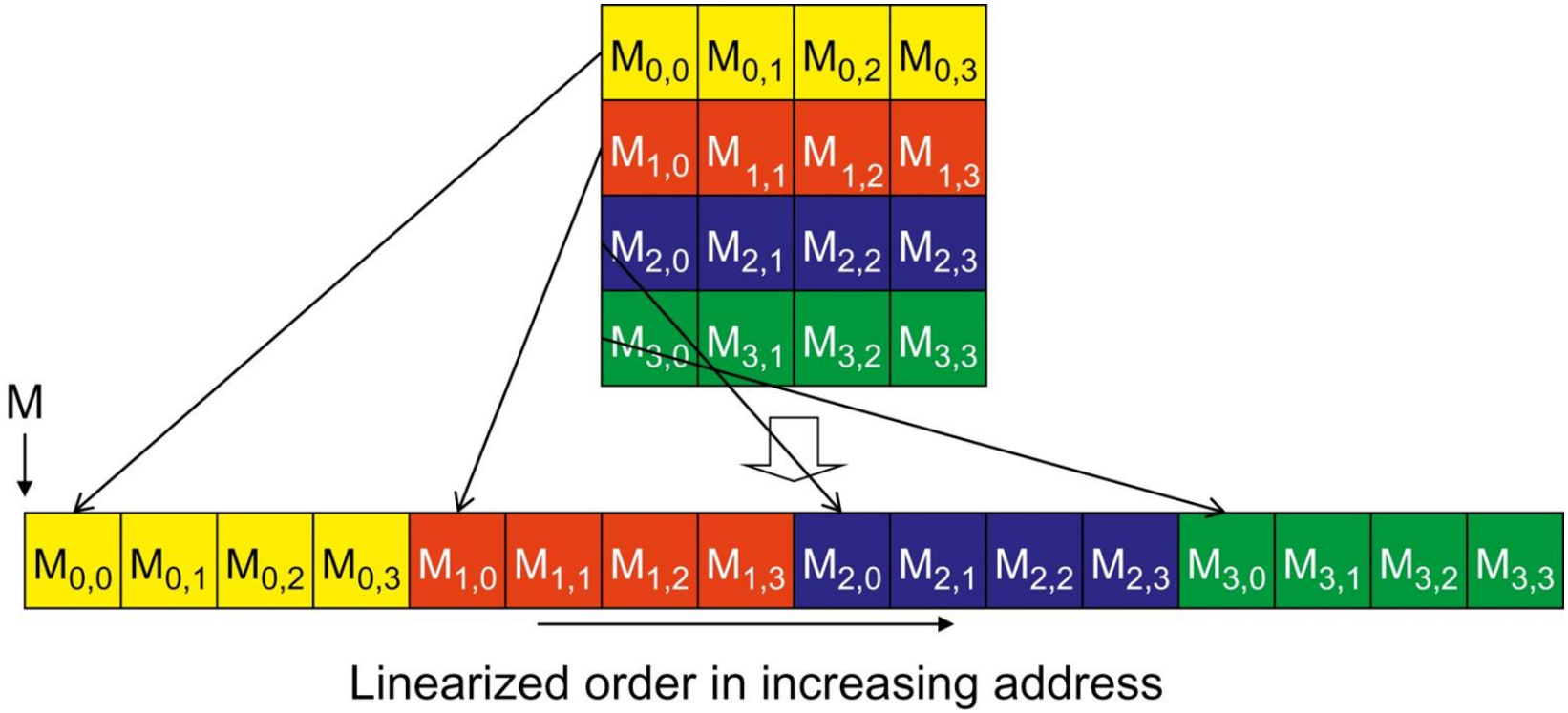

FIGURE 5.1: Placing matrix elements into linear order.

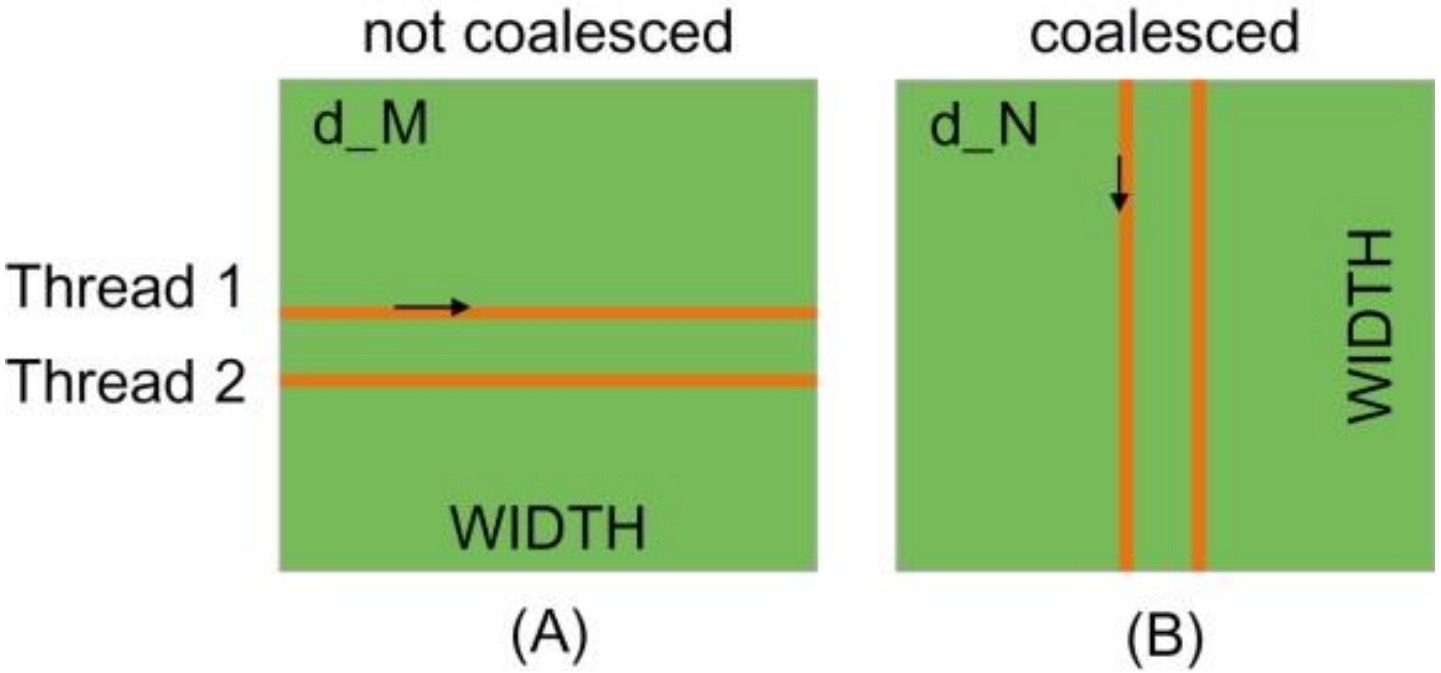

FIGURE 5.2: Memory access patterns in C 2D arrays for coalescing.

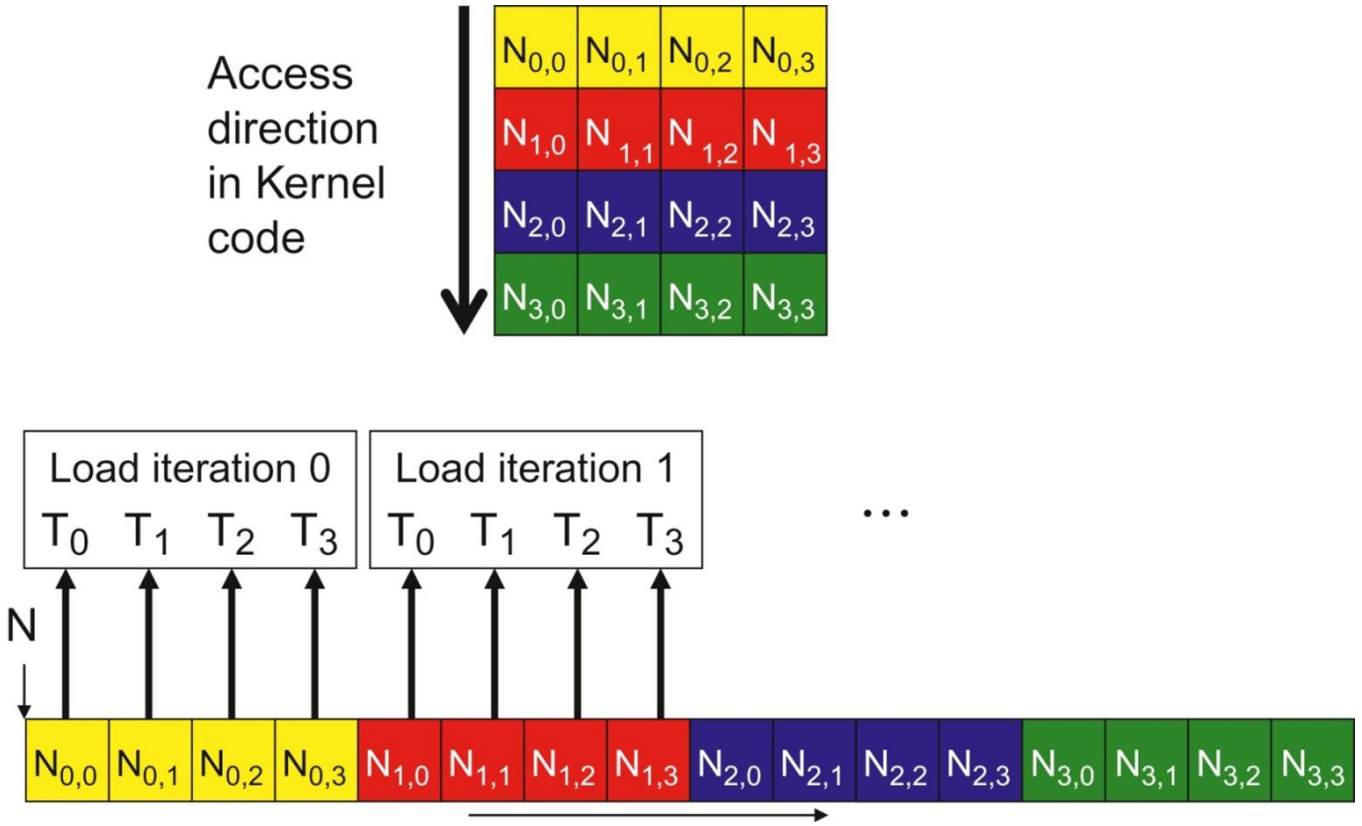

FIGURE 5.3: A coalesced access pattern.

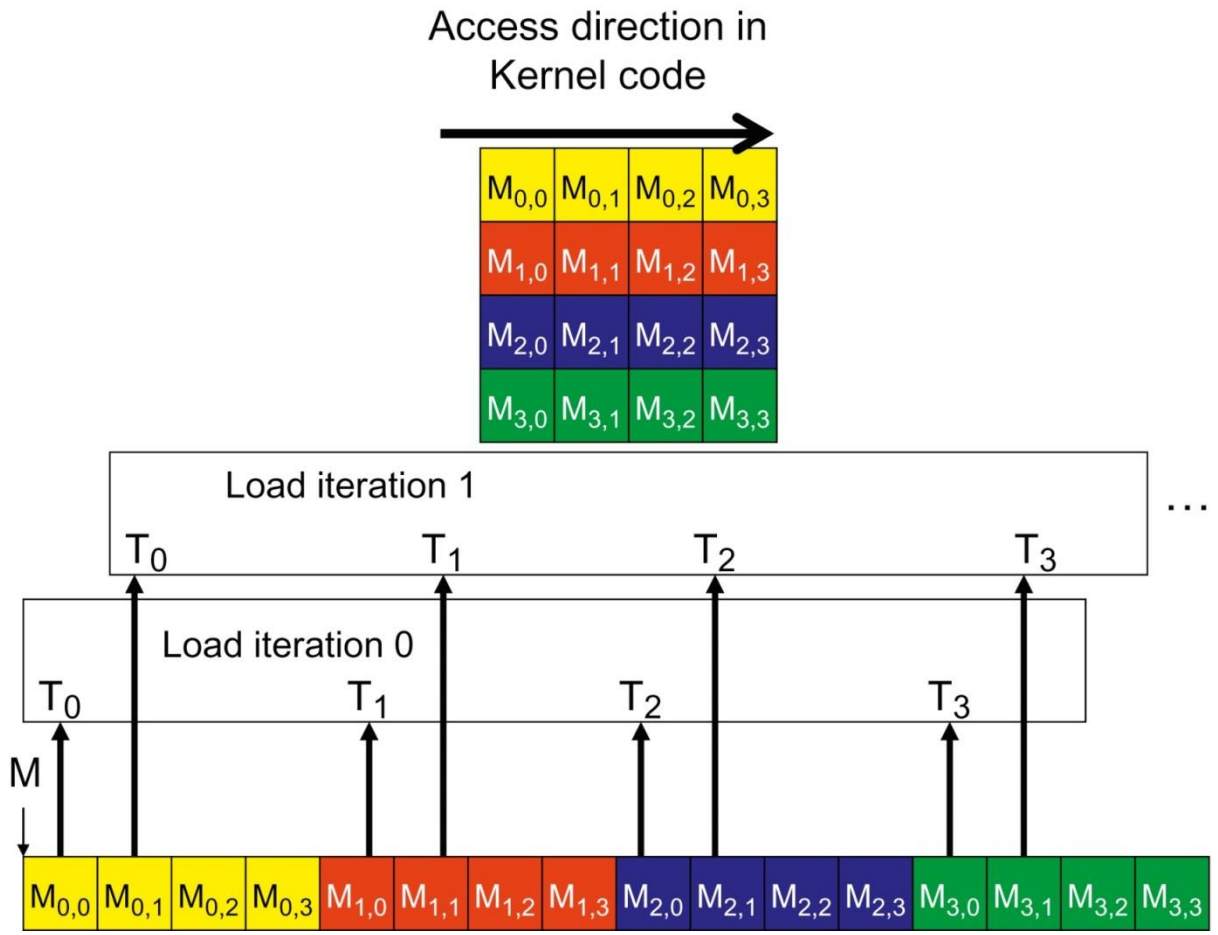

FIGURE 5.4: An un-coalesced access pattern.

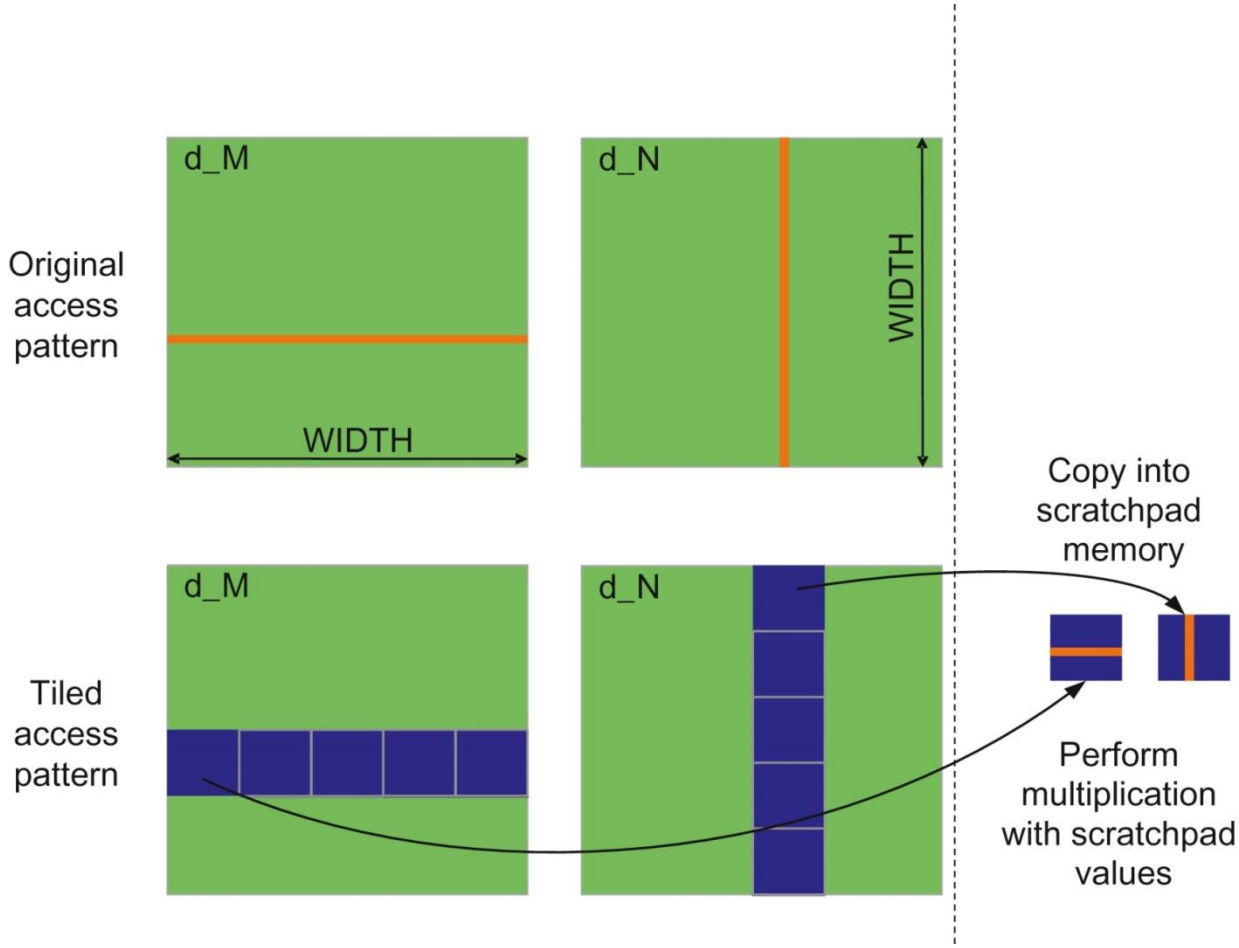

FIGURE 5.5: Using shared memory to enable coalescing.

```
 __global__ void  MatrixMulKernel(float* M, float* N, float* P, int Width)
{

 1.   __shared__ float  Mds[TILE_WIDTH][TILE_WIDTH];
 2.   __shared__ float  Nds[TILE_WIDTH][TILE_WIDTH];

 3.   int bx = blockIdx.x;  int by = blockIdx.y;
 4.   int tx = threadIdx.x; int ty = threadIdx.y;

      // Identify the row and column of the P element to work on
 5.   int Row = by * TILE_WIDTH + ty;
 6.   int Col = bx * TILE_WIDTH + tx;

 7.   float Pvalue = 0;
      // Loop over the M and N tiles required to compute the P element
 8.   for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {

        // Collaborative loading of M and N tiles into shared memory
 9.     Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
10.     Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
11.     __syncthreads();

12.       for (int k = 0; k < TILE_WIDTH; ++k) {
13.         Pvalue += Mds[ty][k] * Nds[k][tx];
          }
14.       __syncthreads();
      }
15.   P[Row*Width + Col] = Pvalue;
    }
```

FIGURE 5.6: Tiled Matrix Multiplication Kernel using shared memory.

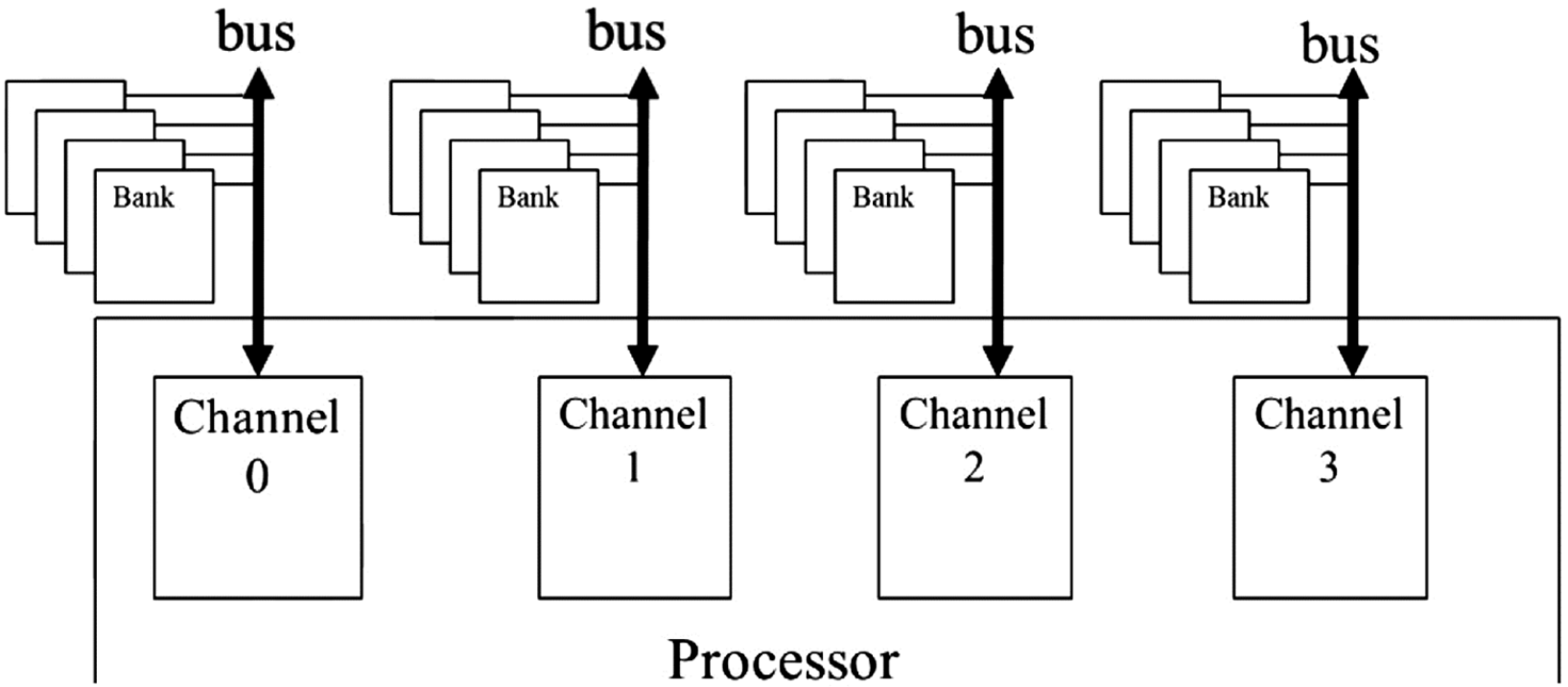

FIGURE 5.7: Channels and banks in DRAM systems.

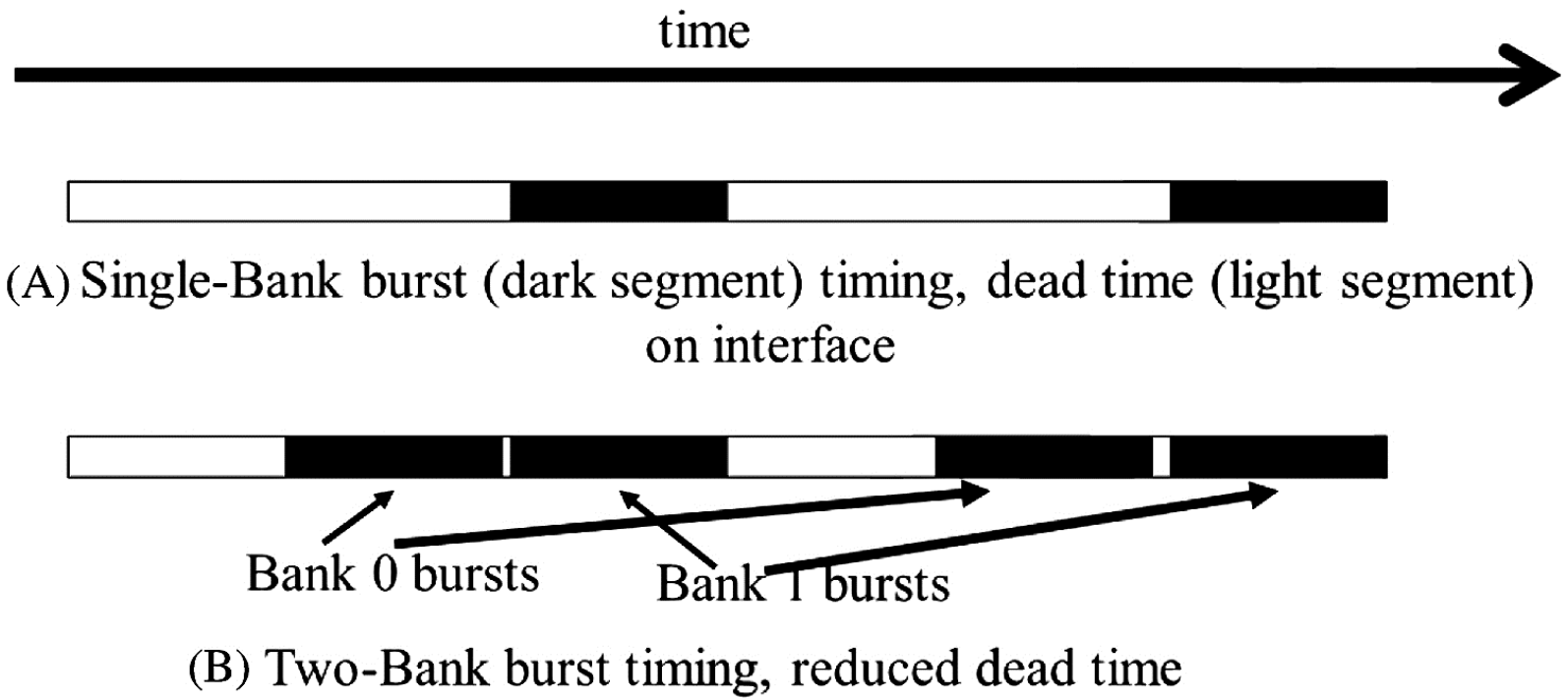

FIGURE 5.8: Banking improves the utilization of data transfer bandwidth of a channel.

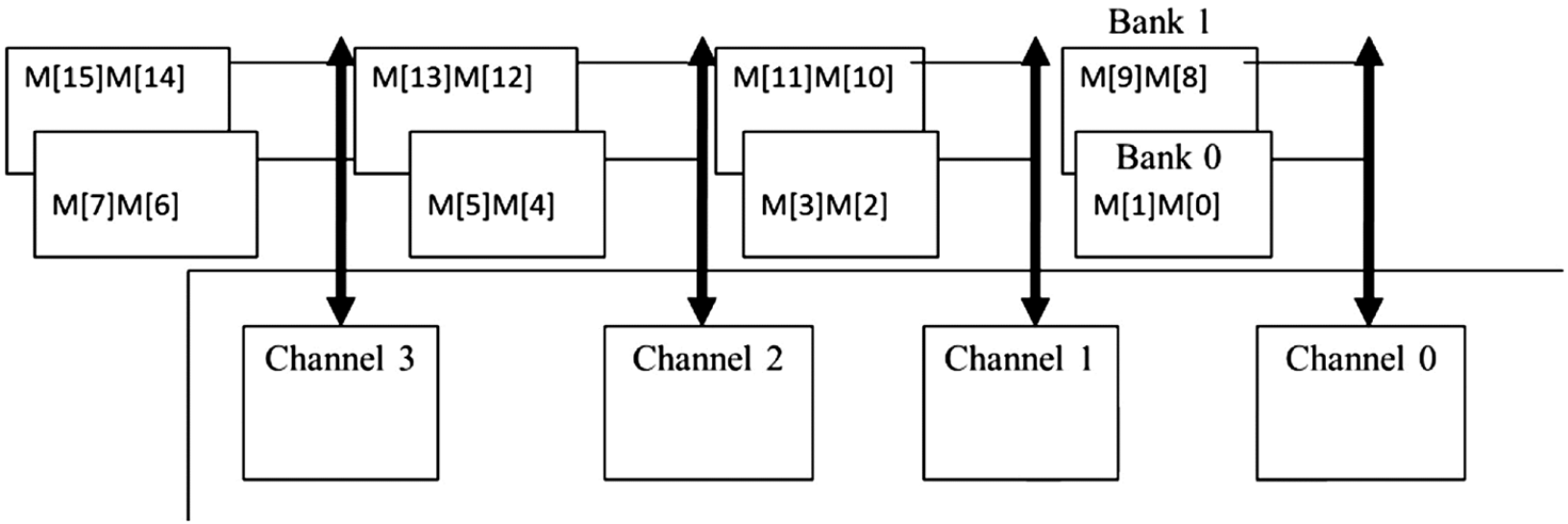

FIGURE 5.9: Distributing array elements into channels and banks.

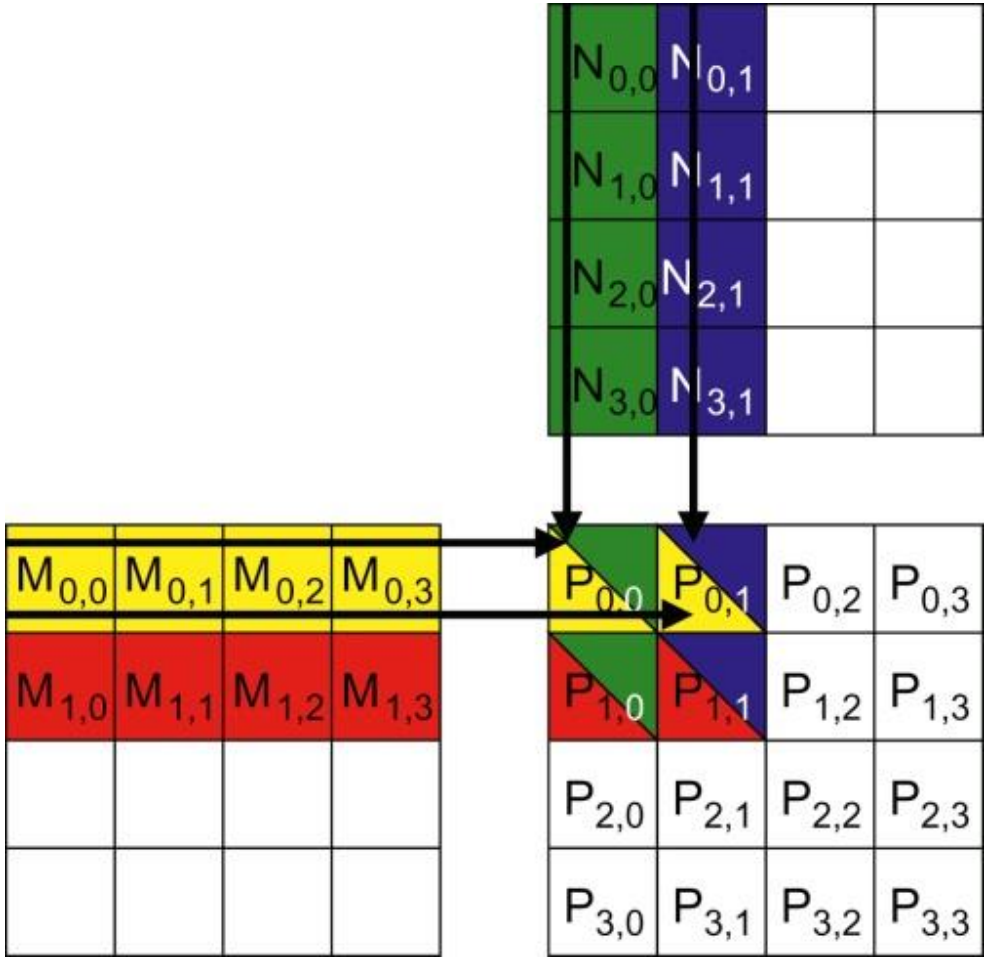

FIGURE 5.10: A small example of matrix multiplication (replicated from Fig. 4.9).

Tiles loaded by	Block 0,0	Block 0,1	Block 1,0	Block 1,1
Phase 0 (2D index)	M[0][0],M[0][1], M[1][0], M[1][1]	M[0][0], M[0][1], M[1][0], M[1][1]	M[2][0], M[2][1], M[3][0], M[3][1]	M[2][0], M[2][1], M[3][0], M[3][1]
Phase 0 (linearized index)	M[0], M[1], M[4], M[5]	M[0], M[1], M[4], M[5]	M[8], M[9], M[12], M[13]	M[8], M[9], M[12], M[13]
Phase 1 (2D index)	M[0][2],M[0][3], M[1][2], M[1][3]	M[0][2],M[0][3], M[1][2], M[1][3]	M[2][2], M[2][3], M[3][2], M[3][3]	M[2][2], M[2][3], M[3][2], M[3][3]
Phase 1 (linearized index)	M[2], M[3], M[6], M[7]	M[2], M[3], M[6], M[7]	M[10], M[11], M[14], M[15]	M[10], M[11], M[14], M[15]

FIGURE 5.11: M elements loaded by thread blocks in each phase.

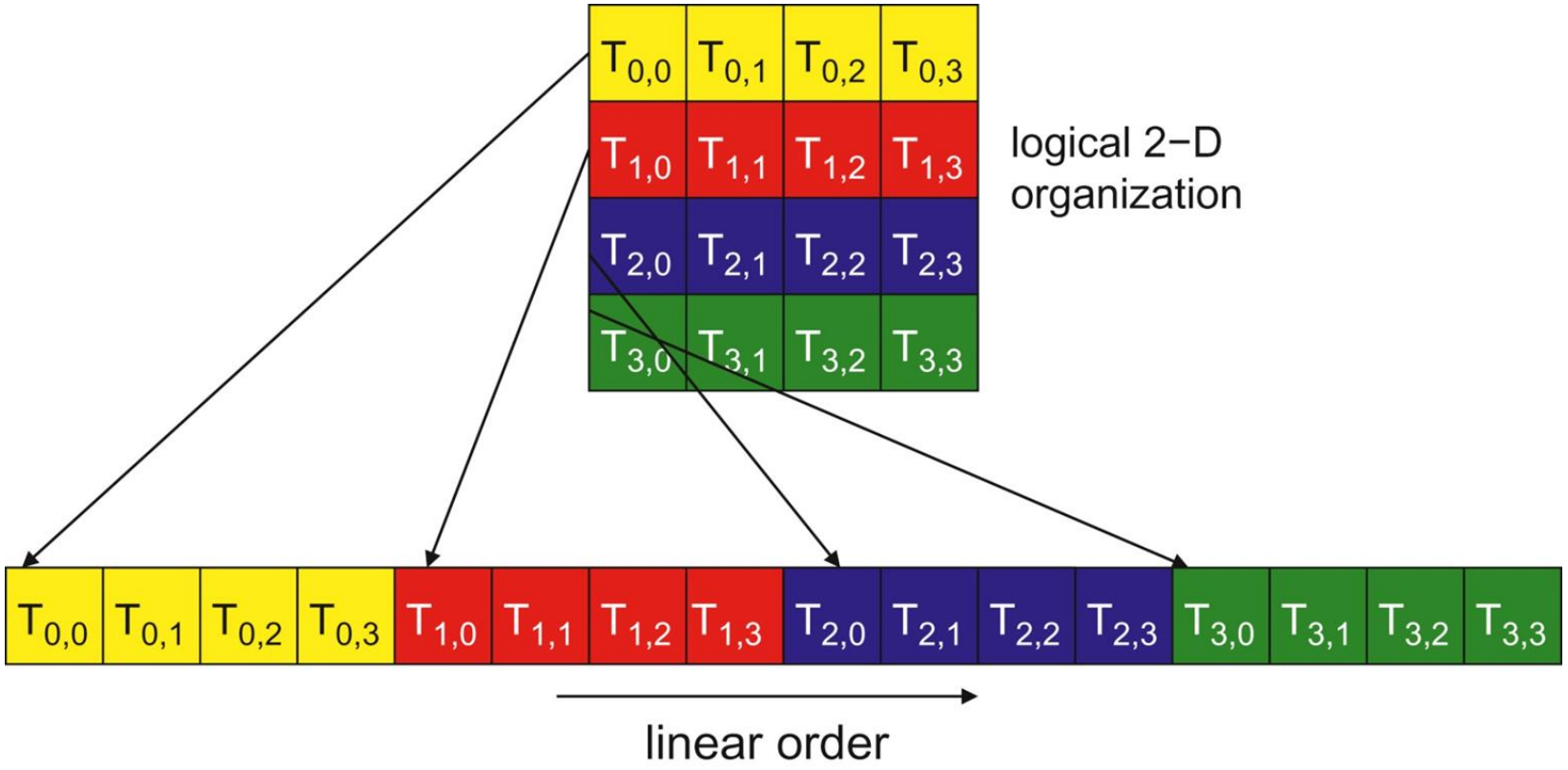

FIGURE 5.12: Placing 2D threads into linear order.

```
1.  __shared__ float partialSum[SIZE];
    partialSum[threadIdx.x] = X[blockIdx.x*blockDim.x+threadIdx.x];
2.  unsigned int t = threadIdx.x;
3.  for (unsigned int stride = 1; stride < blockDim.x; stride *= 2)
4.  {
5.    __syncthreads();
6.    if (t % (2*stride) == 0)
7.      partialSum[t] += partialSum[t+stride];
8.  }
```

FIGURE 5.13: A simple sum reduction kernel.

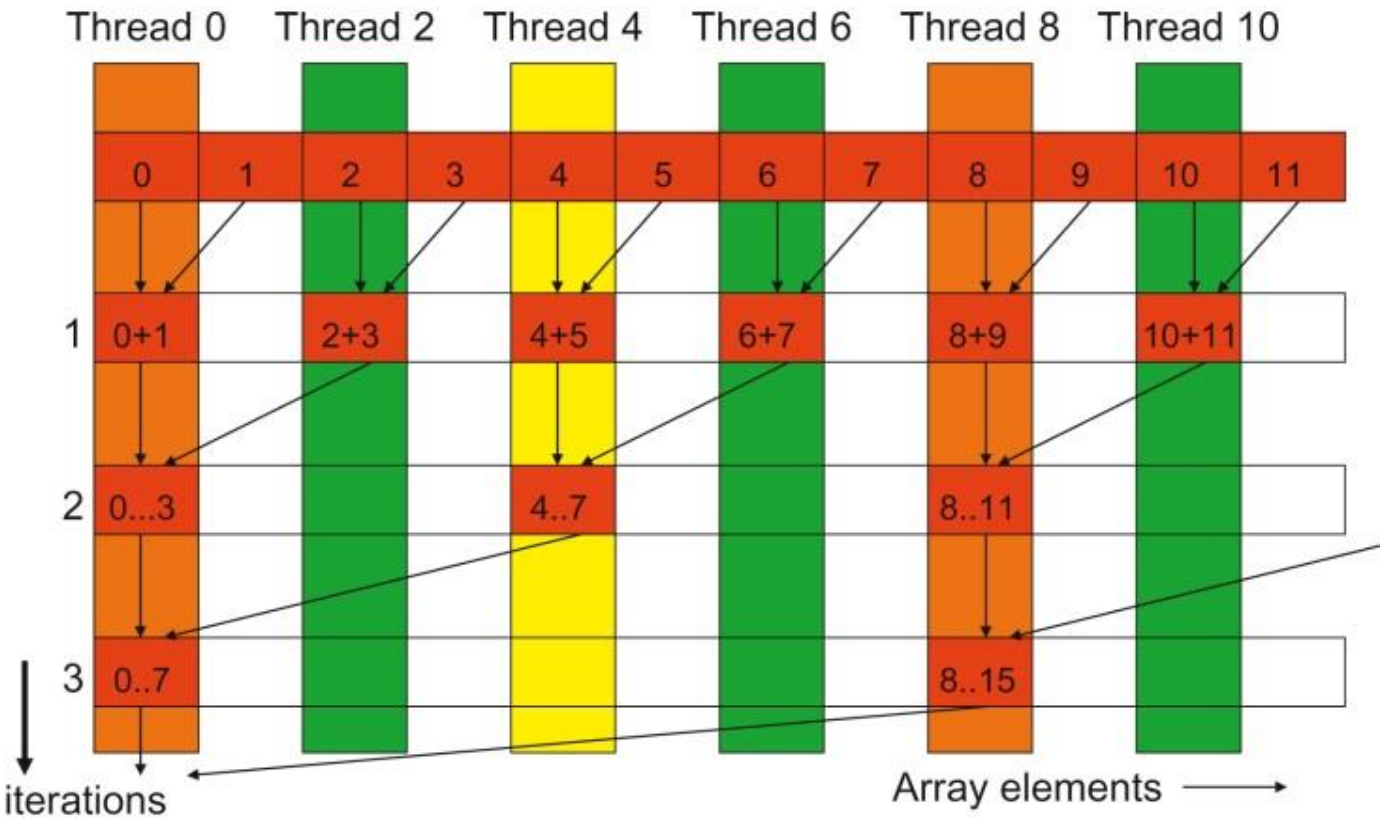

FIGURE 5.14: Execution of the sum reduction kernel.

```
1.  __shared__ float partialSum[SIZE];
    partialSum[threadIdx.x] = X[blockIdx.x*blockDim.x+threadIdx.x];

2.  unsigned int t = threadIdx.x;
3.  for (unsigned int stride = blockDim.x/2; stride >= 1; stride = stride>>1)
4.  {
5.    __syncthreads();
6.    if (t < stride)
7.      partialSum[t] += partialSum[t+stride];
8.  }
```

FIGURE 5.15: A kernel with fewer thread divergence.

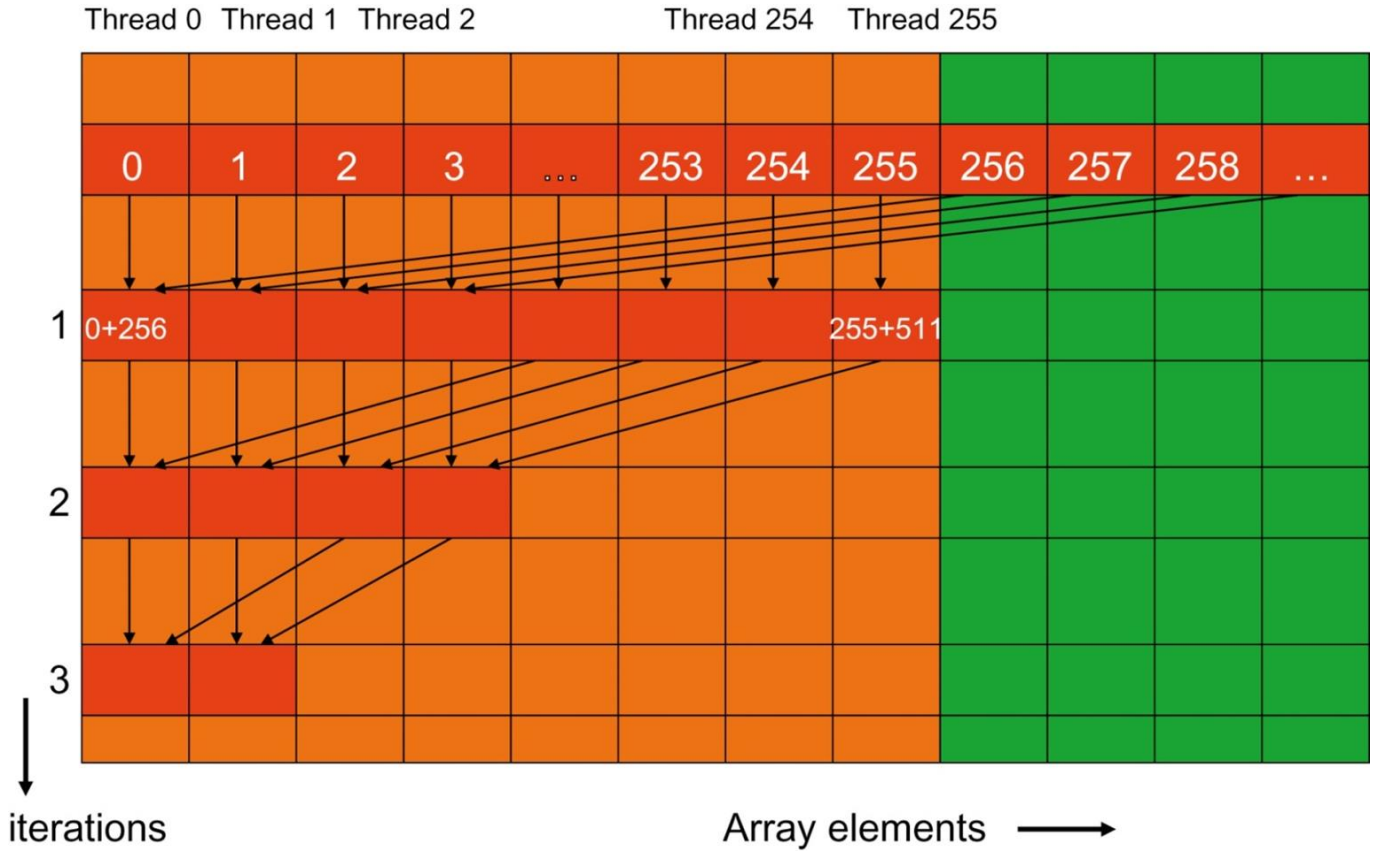

FIGURE 5.16: Execution of the revised algorithm.

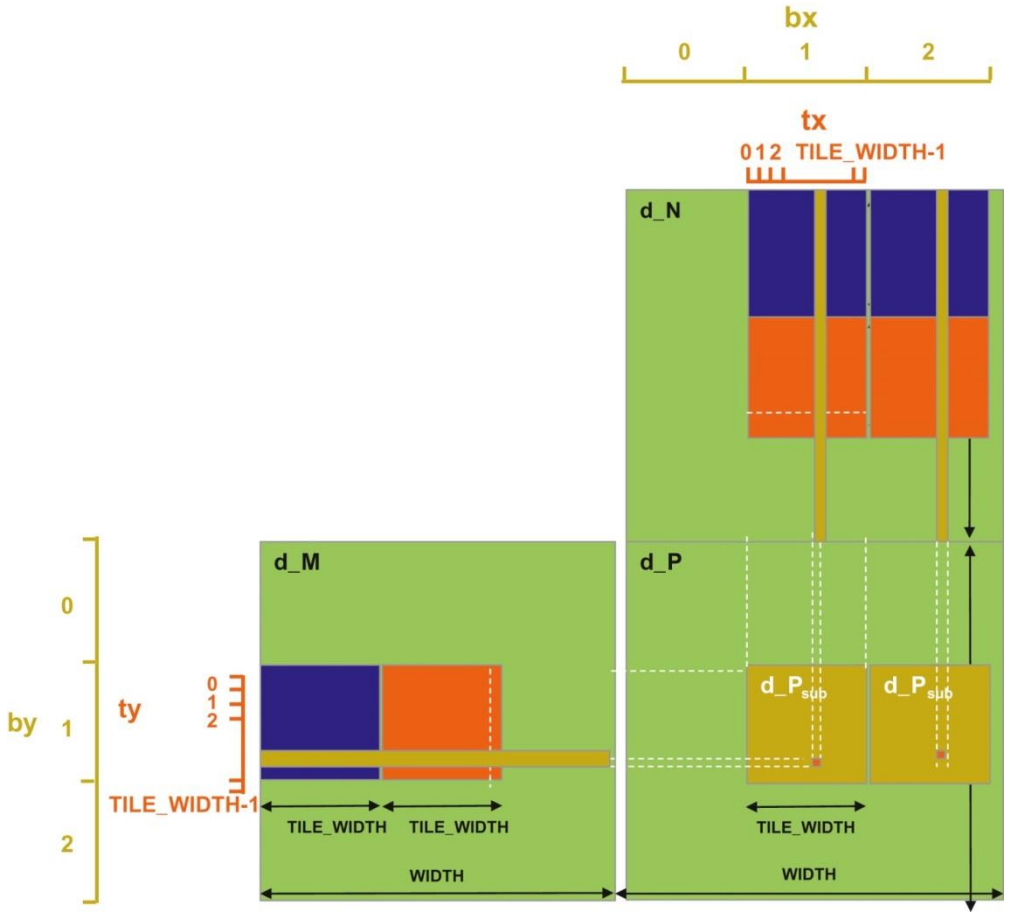

FIGURE 5.17: Increased thread granularity with rectangular tiles.