Chapter-4
Memory and data locality

```
      for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {
4.      for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol) {

5.        int curRow = Row + blurRow;
6.        int curCol = Col + blurCol;
          // Verify we have a valid image pixel
7.        if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
8.          pixVal += in[curRow * w + curCol];
9.          pixels++; // Keep track of number of pixels in the avg
          }
        }
      }
```

FIGURE 4.1: The most executed part of the image blurring kernel in Fig. 3.8.

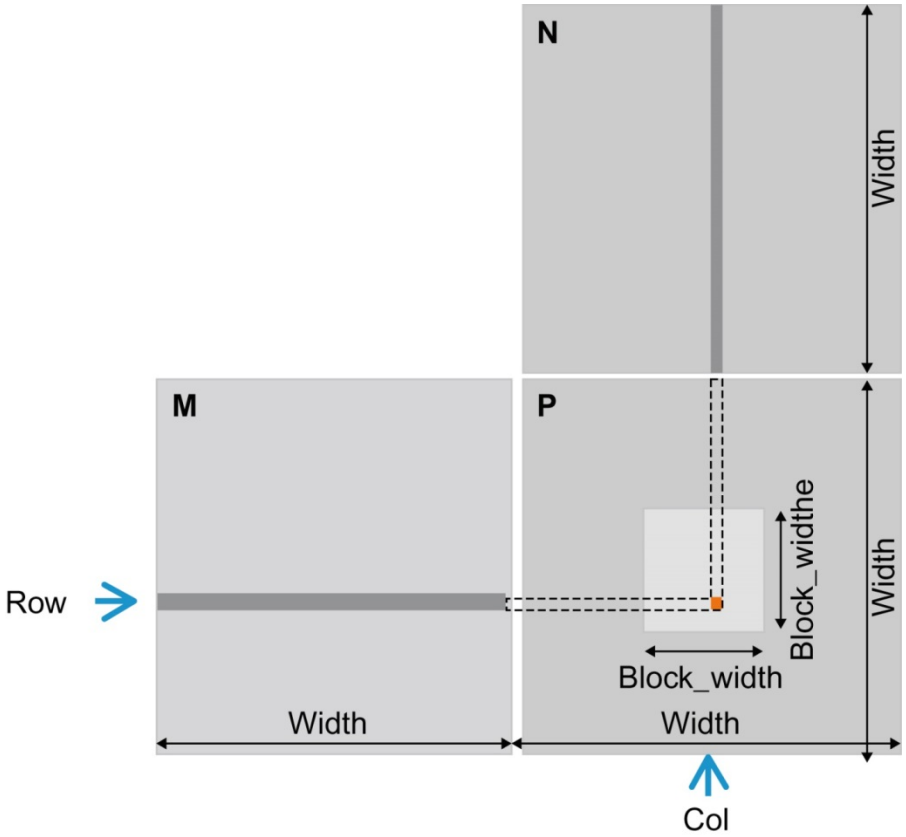

FIGURE 4.2: Matrix multiplication using multiple blocks by tiling P.

```
__global__ void MatrixMulKernel(float* M, float* N, float* P,
  int Width) {
  // Calculate the row index of the P element and M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;
  // Calculate the column index of P and N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;
  if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k) {
      Pvalue += M[Row*Width+k]*N[k*Width+Col];
    }
    P[Row*Width+Col] = Pvalue;
  }

}
```

FIGURE 4.3: A simple matrix multiplication kernel using one thread to compute one P element.

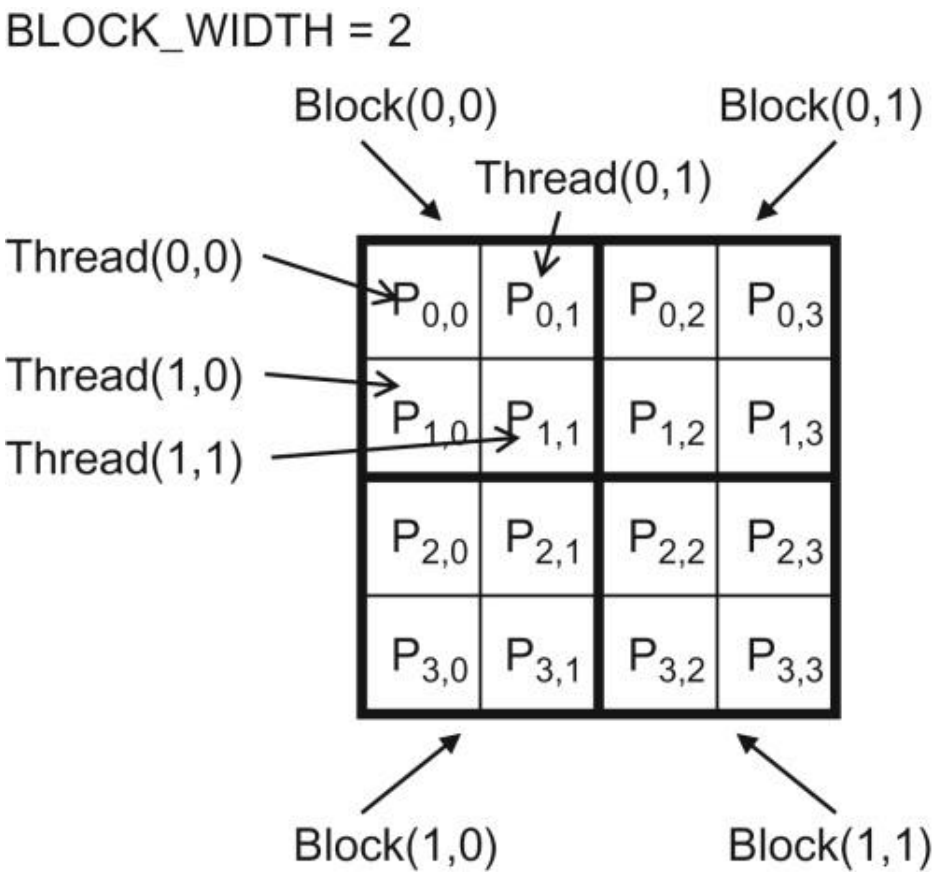

FIGURE 4.4: A small execution example of matrixMulKernel.

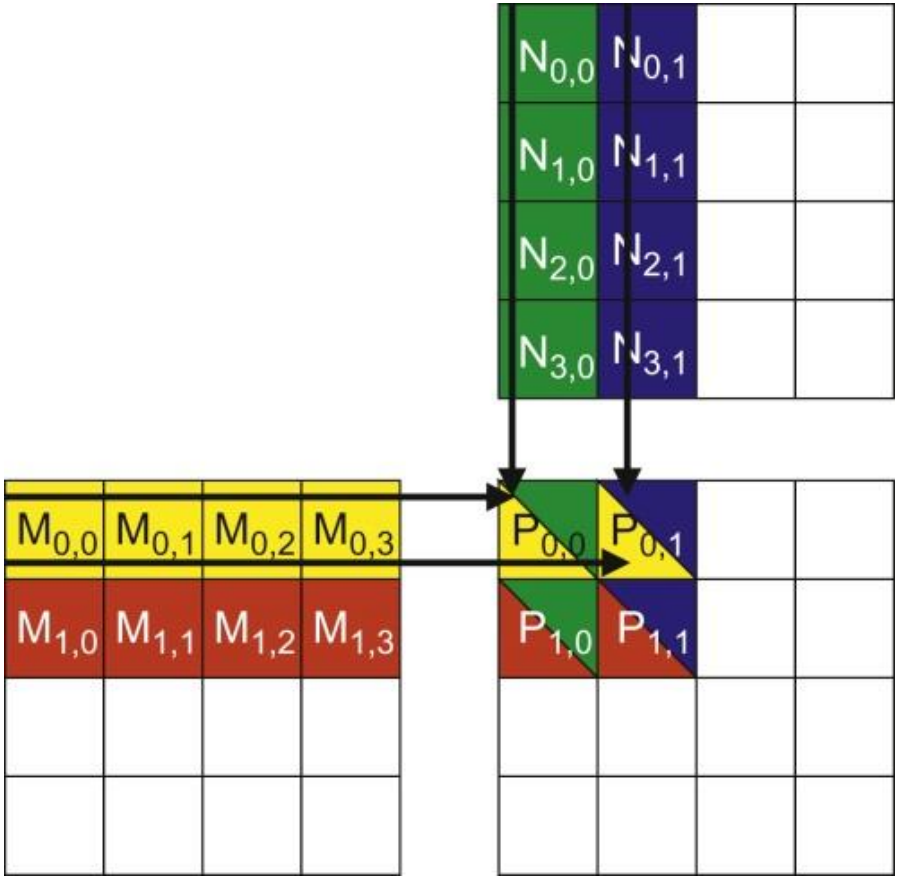

FIGURE 4.5: Matrix multiplication actions of one thread block.

Device code can:

- R/W per-thread registers
- R/W per-thread local memory
- R/W per-block shared memory
- R/W per-grid global memory
- Read only per-grid constant memory

Host code can

- Transfer data to/from per grid global and constant memories

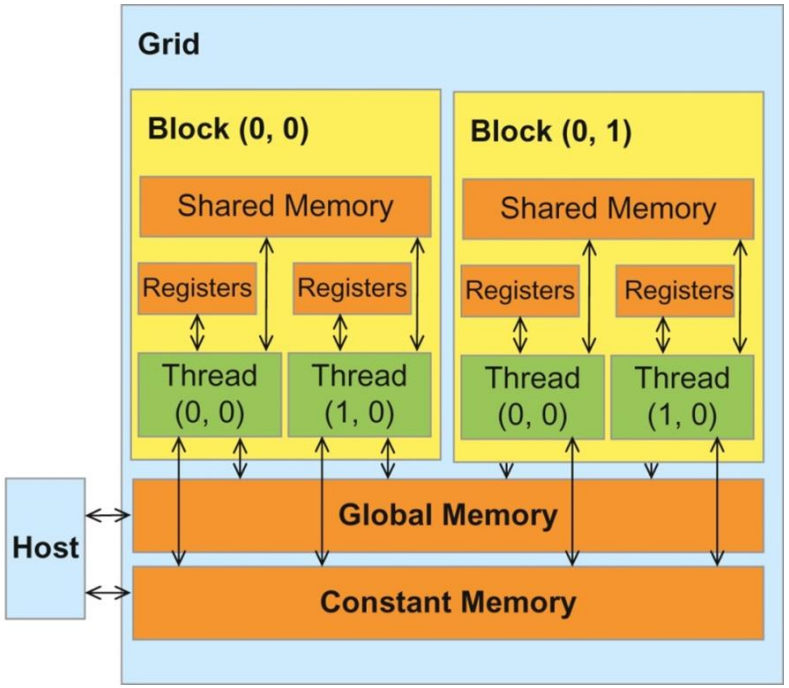

FIGURE 4.6: Overview of the CUDA device memory model.

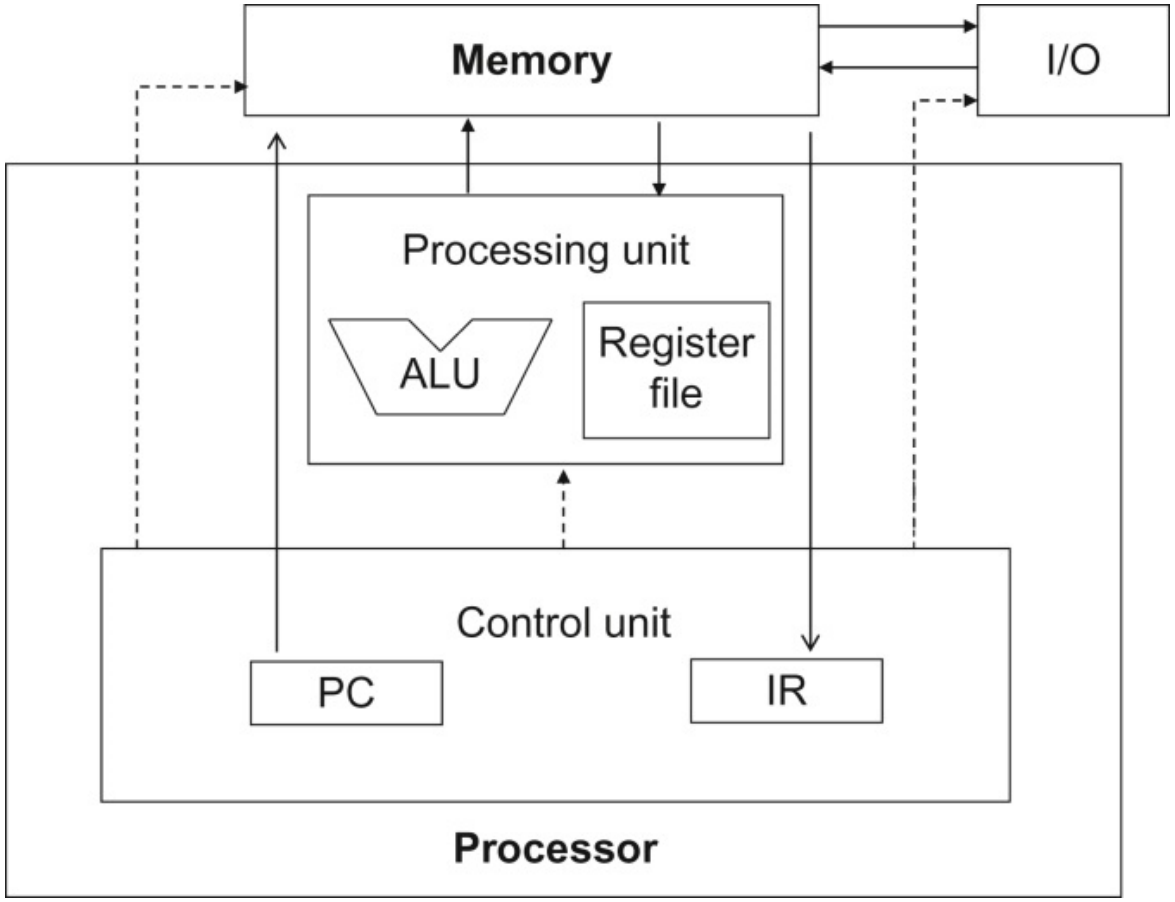

FIGURE 4.7: Memory vs. registers in a modern computer based on the von Neumann model.

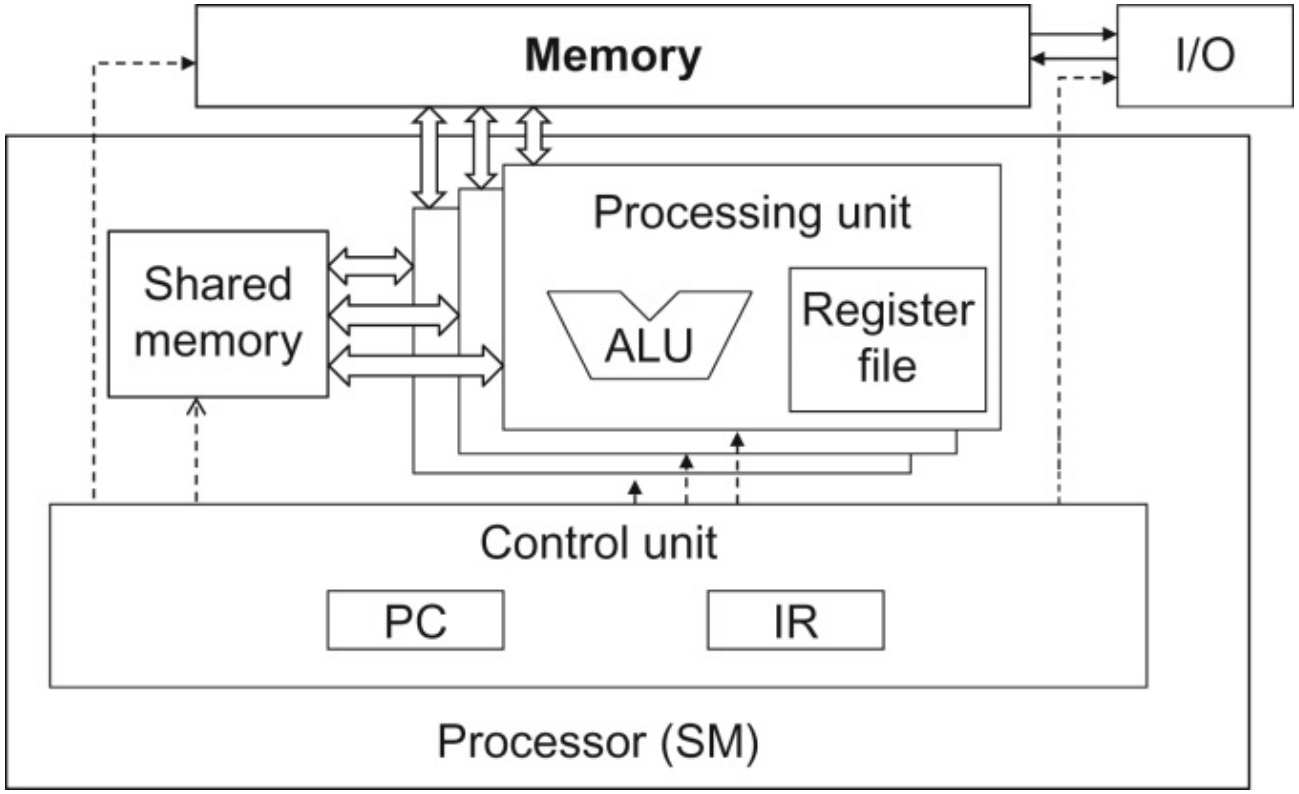

FIGURE 4.8: Shared memory vs. registers in a CUDA device SM.

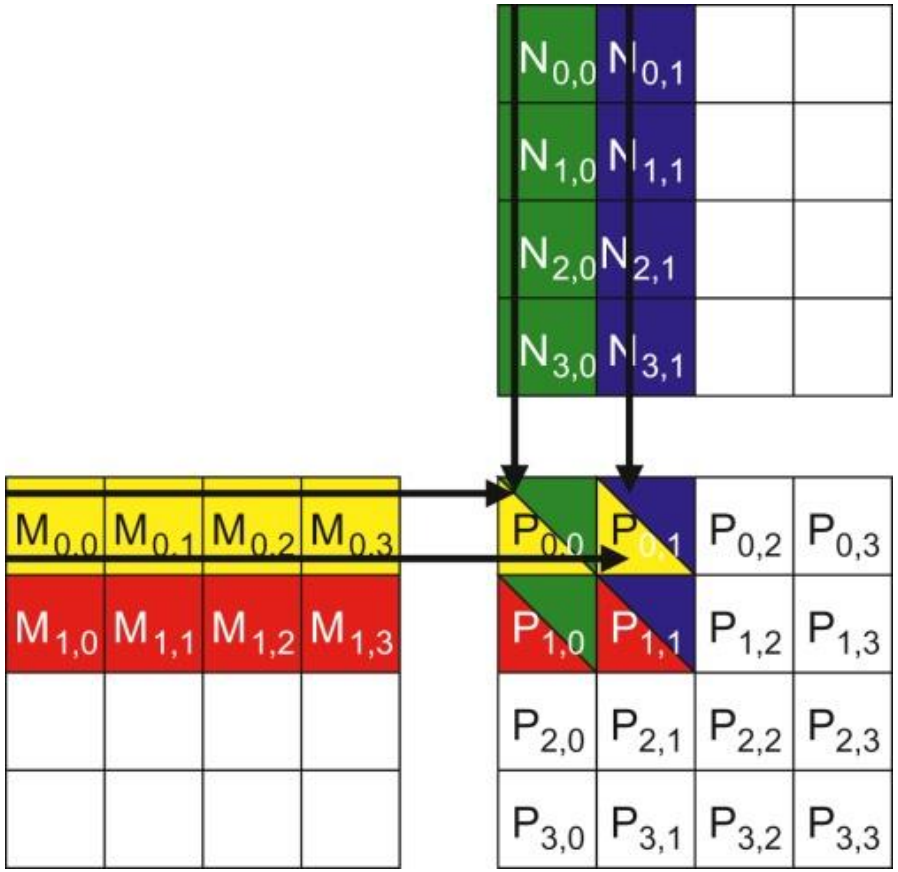

FIGURE 4.9: A small example of matrix multiplication. For brevity, we show *M*[*y**Width+ *x*], *N*[*y**Width + *x*], *P*[y*Width+ *x*] as *My,x*, *Ny,x Py,x*.

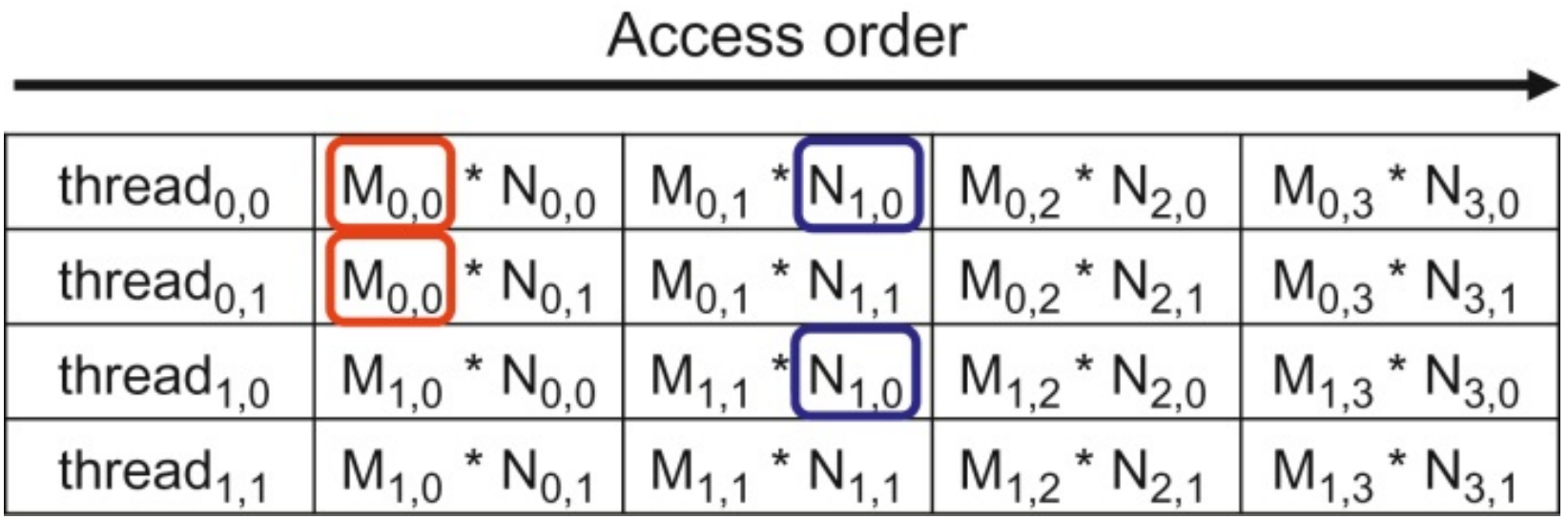

FIGURE 4.10: Global memory accesses performed by threads in block0,0.

FIGURE 4.11: Reducing traffic congestion in highway systems.

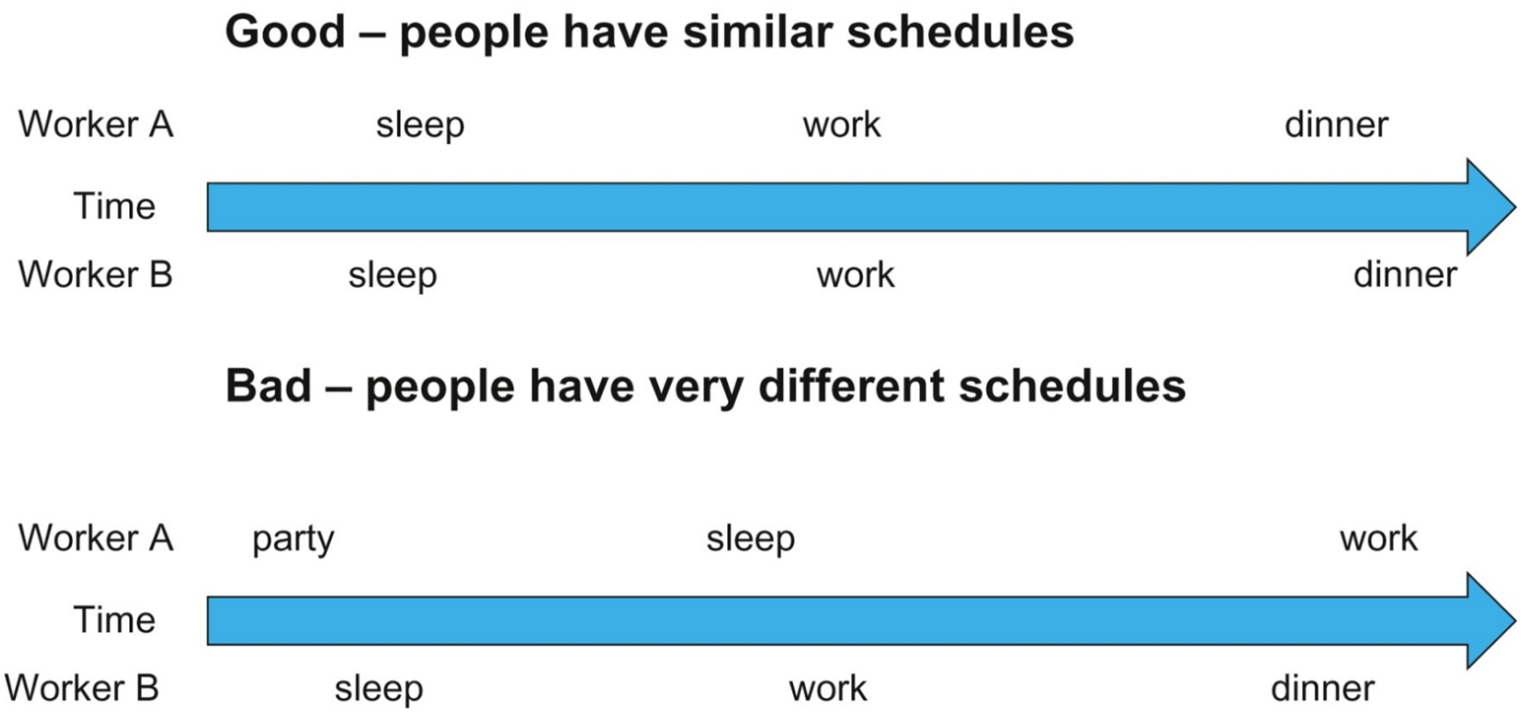

FIGURE 4.12: Carpooling requires synchronization among people.

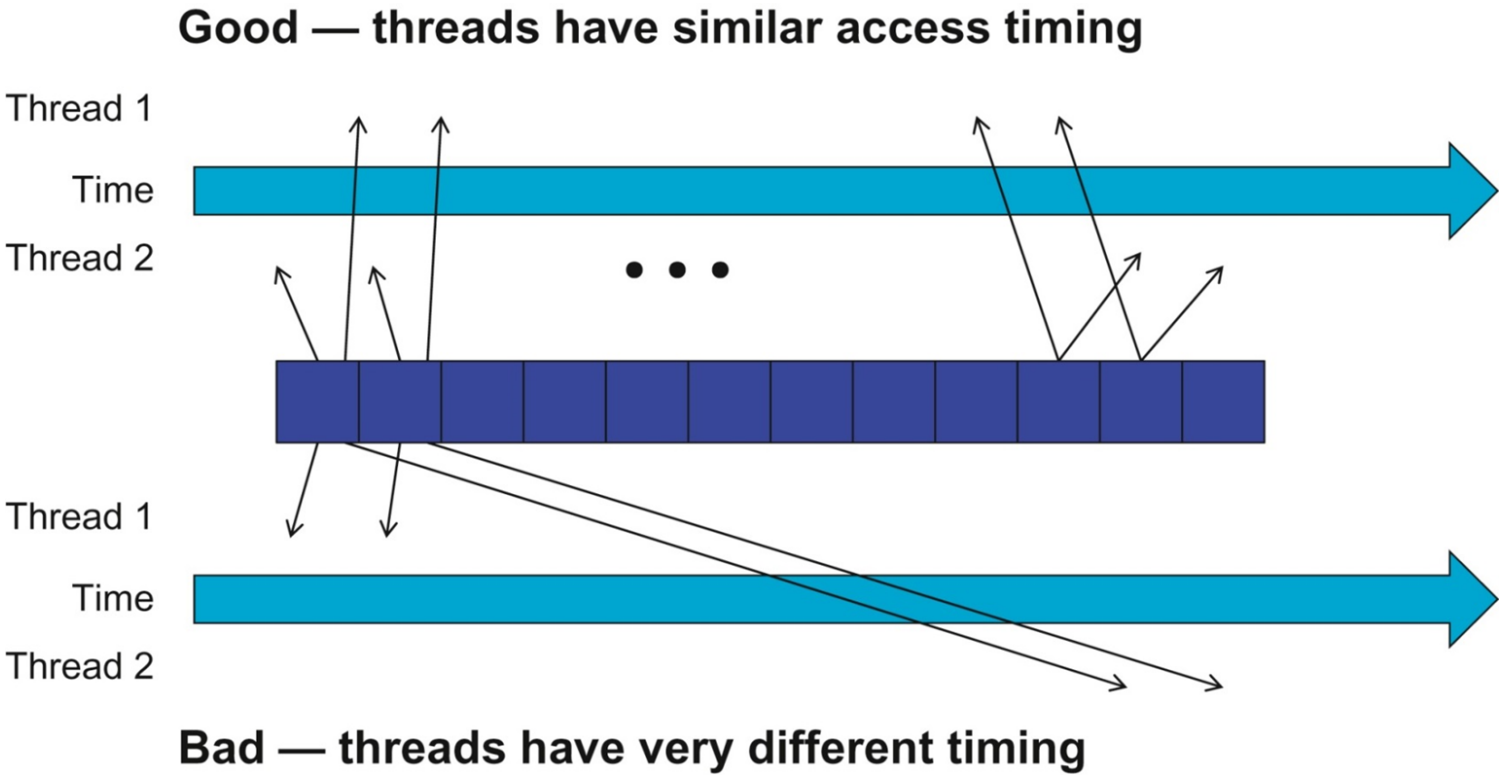

FIGURE 4.13: Tiled Algorithms require synchronization among threads.

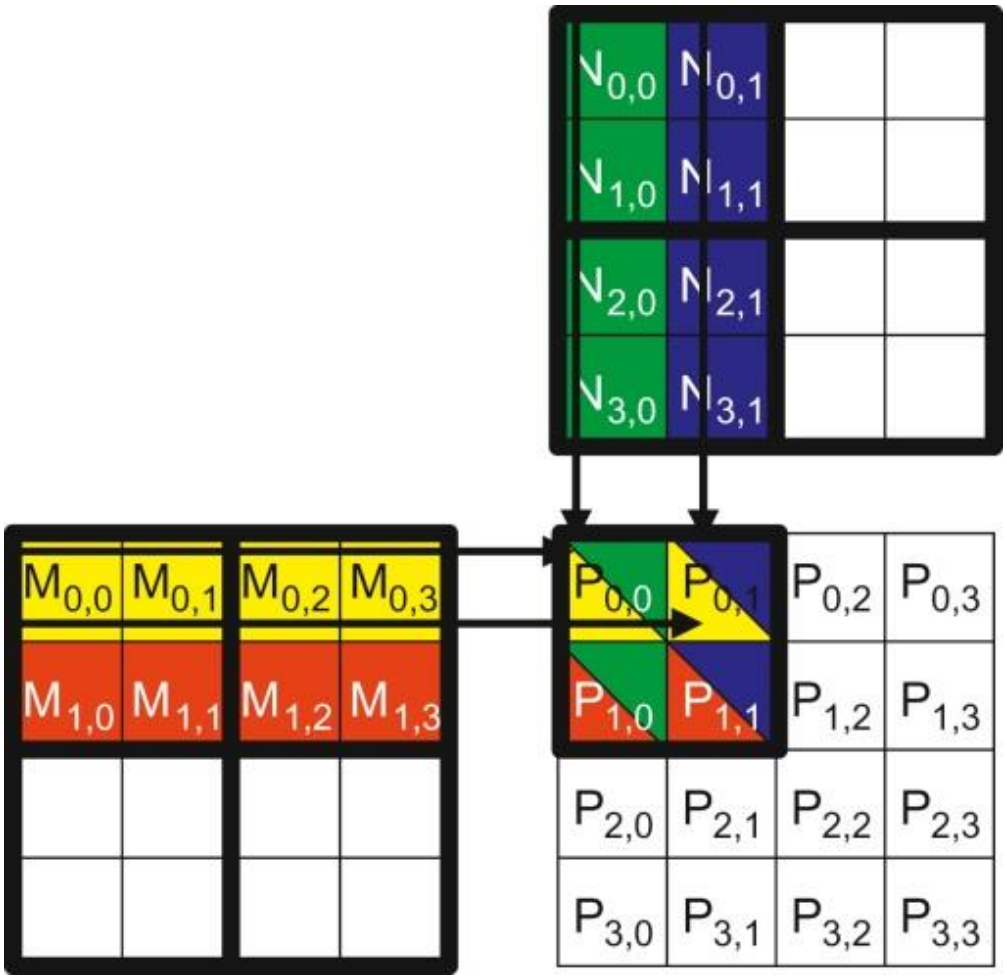

FIGURE 4.14: Tiling *M* and *N* to utilize shared memory.

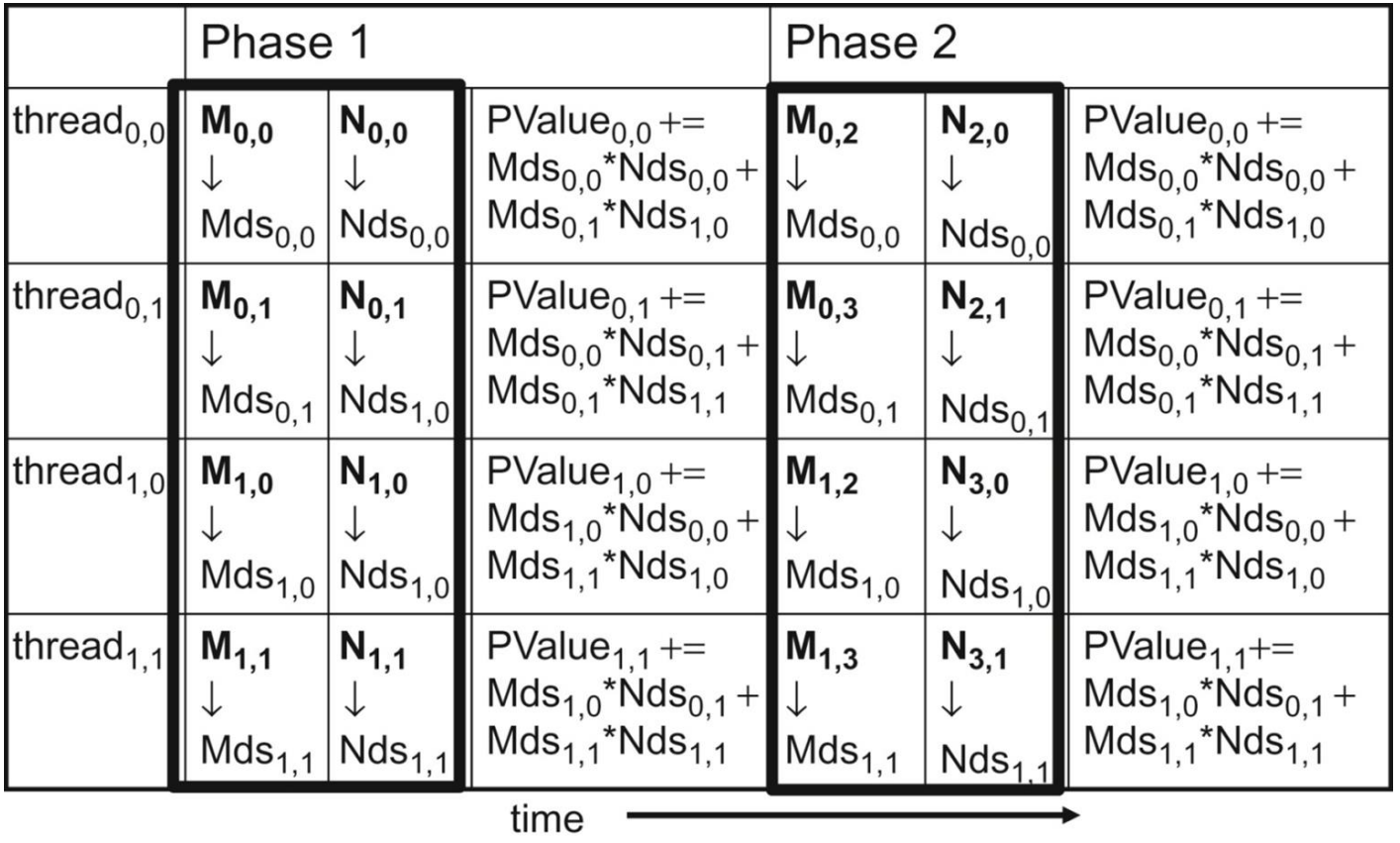

FIGURE 4.15: Execution phases of a tiled matrix multiplication.

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_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 d_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 d_M and d_N tiles required to compute d_P element
8.   for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {

       // Collaborative loading of d_M and d_N tiles into shared memory
9.     Mds[ty][tx] = d_M[Row*Width + ph*TILE_WIDTH + tx];
10.    Nds[ty][tx] = d_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.  d_P[Row*Width + Col] = Pvalue;
   }
```

FIGURE 4.16: A tiled Matrix Multiplication Kernel using shared memory.

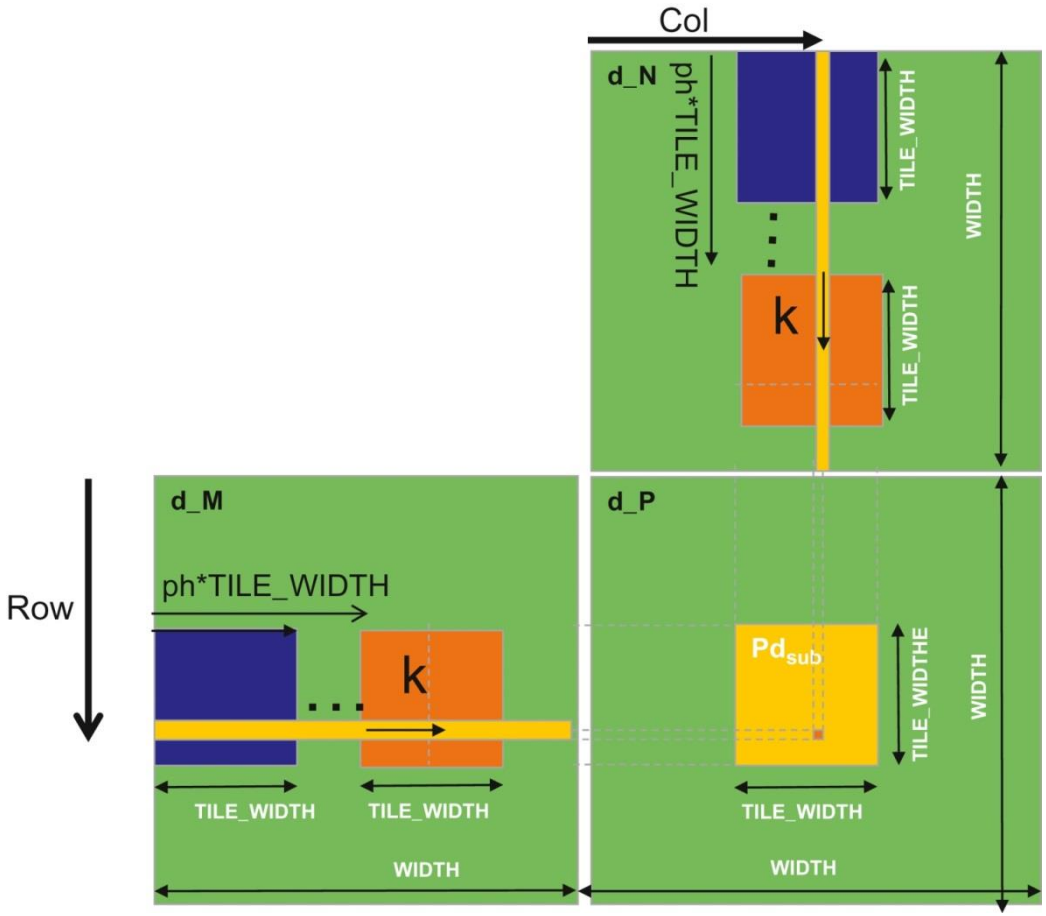

FIGURE 4.17: Calculation of the matrix indexes in tiled multiplication.

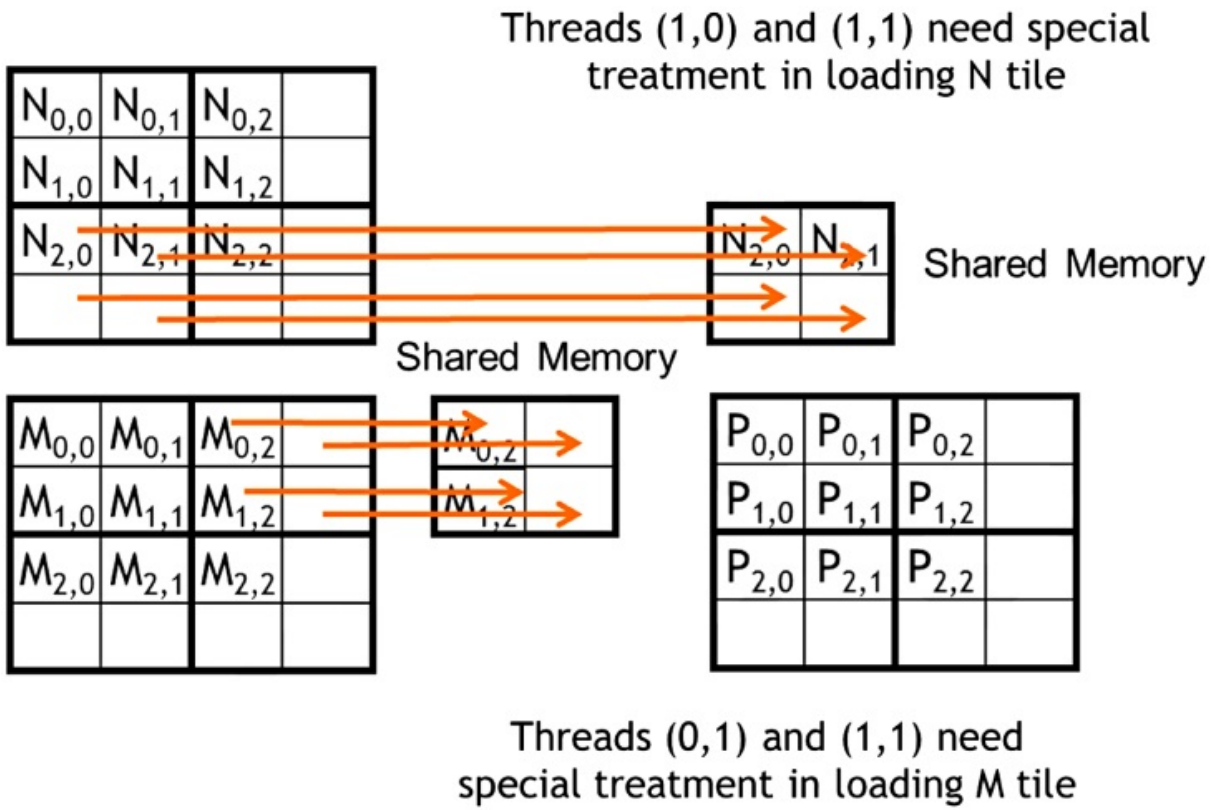

FIGURE 4.18: Loading input matrix elements that are close to the edge–phase 1 of Block0,0.

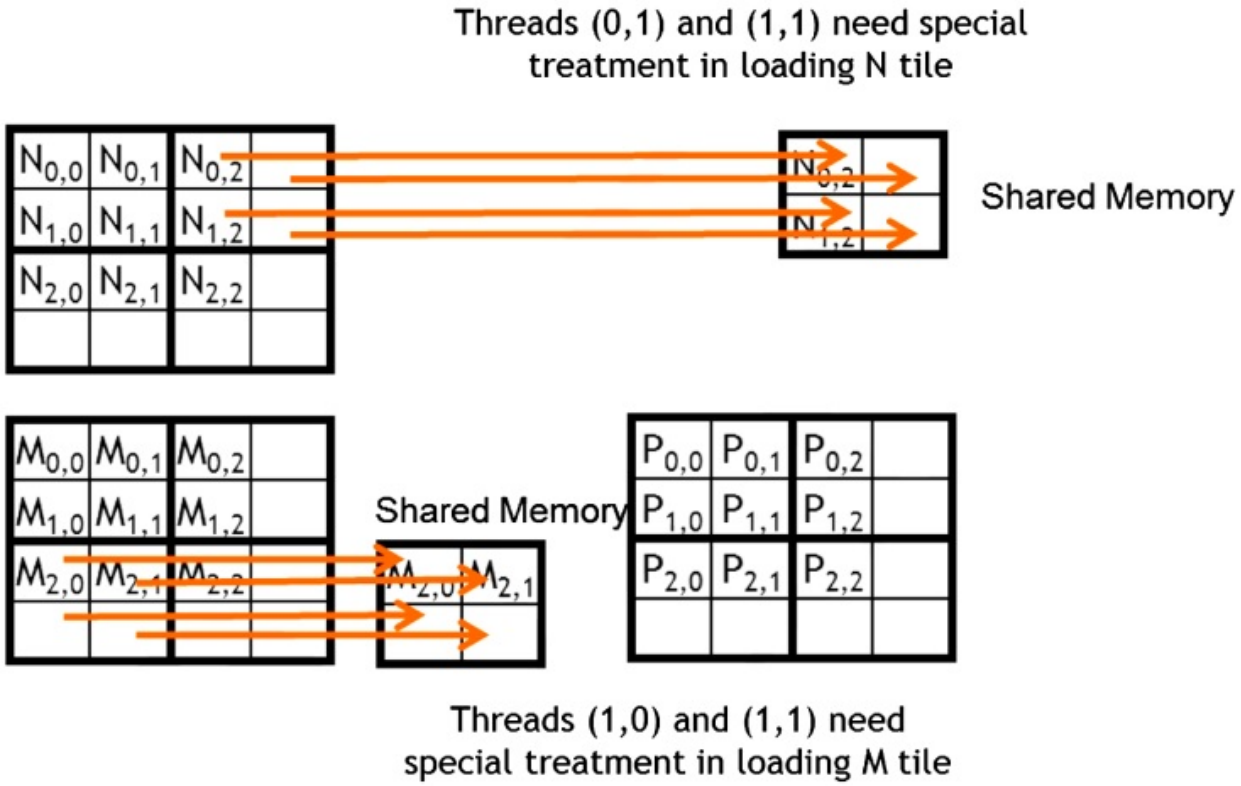

FIGURE 4.19: Loading input elements during phase 0 of block1,0.

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

        // Collaborative loading of M and N tiles into shared memory
 9.     if ((Row< Width) && (ph*TILE_WIDTH+tx)< Width)
            Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
10.     if ((ph*TILE_WIDTH+ty)<Width && Col<Width)
            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.   if ((Row<Width) && (Col<Width)P[Row*Width + Col] = Pvalue;
```

FIGURE 4.20: Tiled matrix multiplication kernel with boundary condition checks.