CHAPTER 6

Performance considerations

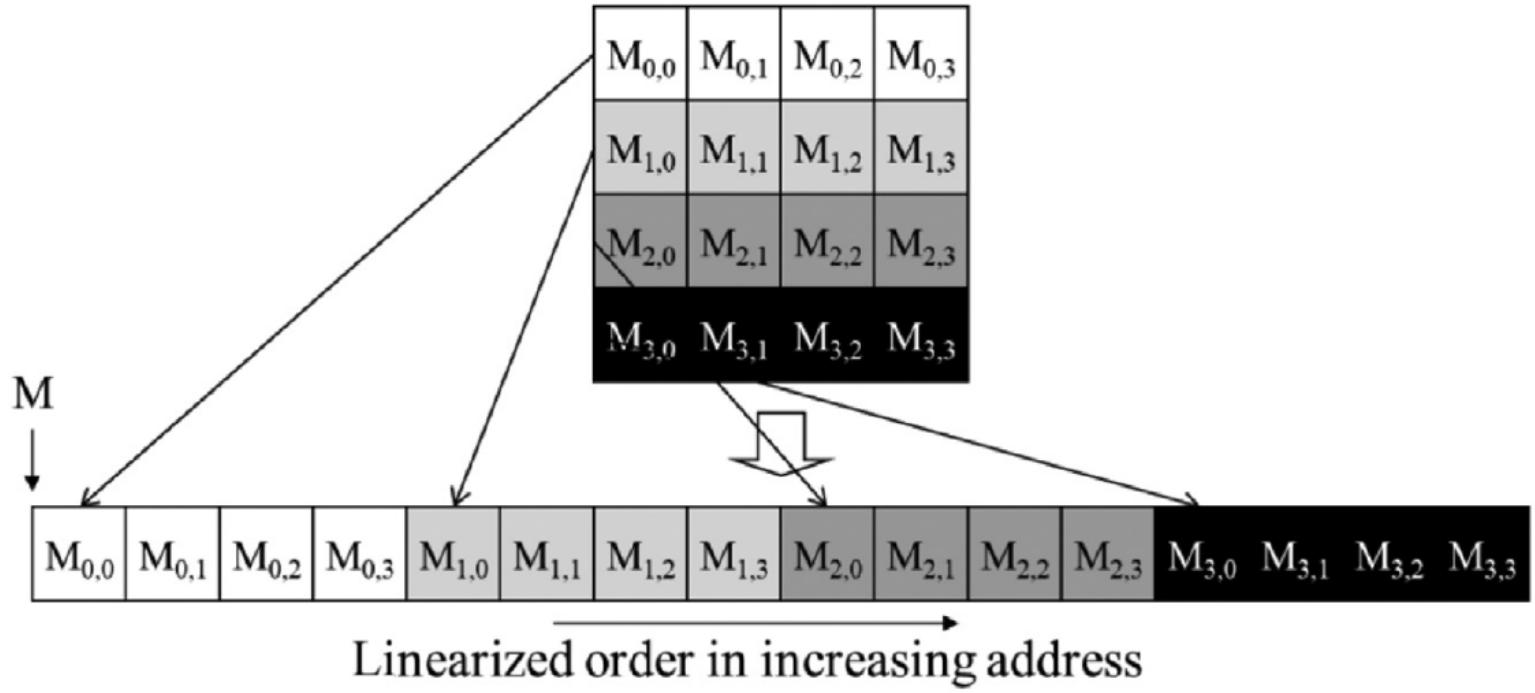


FIGURE 6.1

Placing matrix elements into a linear array based on row-major order.

Code

```
unsigned int row = blockIdx.y*blockDim.y + threadIdx.y;
unsigned int col = blockIdx.x*blockDim.x + threadIdx.x;
if (row < Width && col < Width) {
    float Pvalue = 0.0f;
    for(unsigned int k = 0; k < Width; ++k) {
        Pvalue += N[row*Width + k]*M[k*Width + col];
    }
    P[row*Width + col] = Pvalue;
}
```

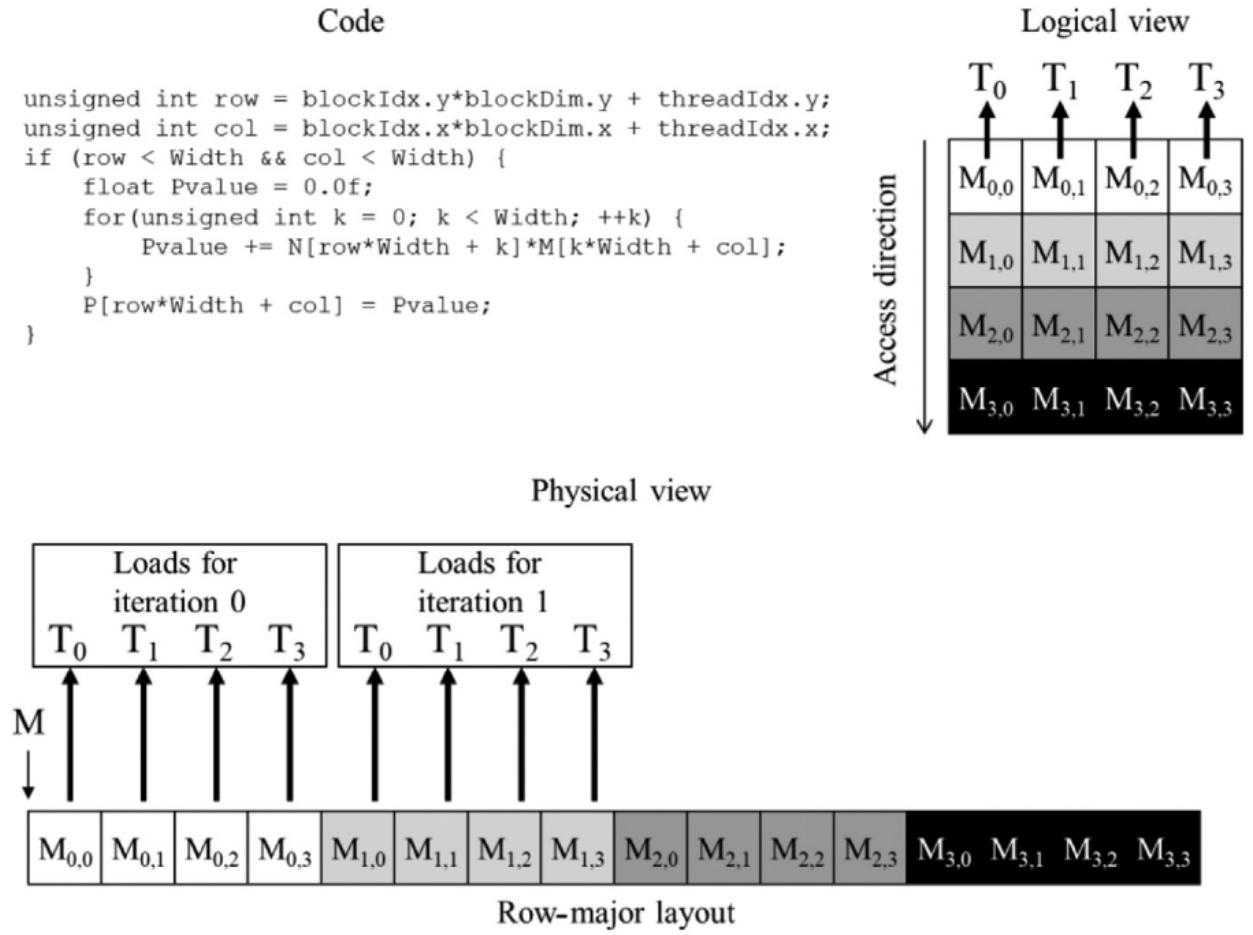


FIGURE 6.2

A coalesced access pattern.

Code

```
unsigned int row = blockIdx.y*blockDim.y + threadIdx.y;
unsigned int col = blockIdx.x*blockDim.x + threadIdx.x;
if (row < Width && col < Width) {
    float Pvalue = 0.0f;
    for(unsigned int k = 0; k < Width; ++k) {
        Pvalue += N[row*Width + k]*M[col*Width + k];
    }
    P[row*Width + col] = Pvalue;
}
```

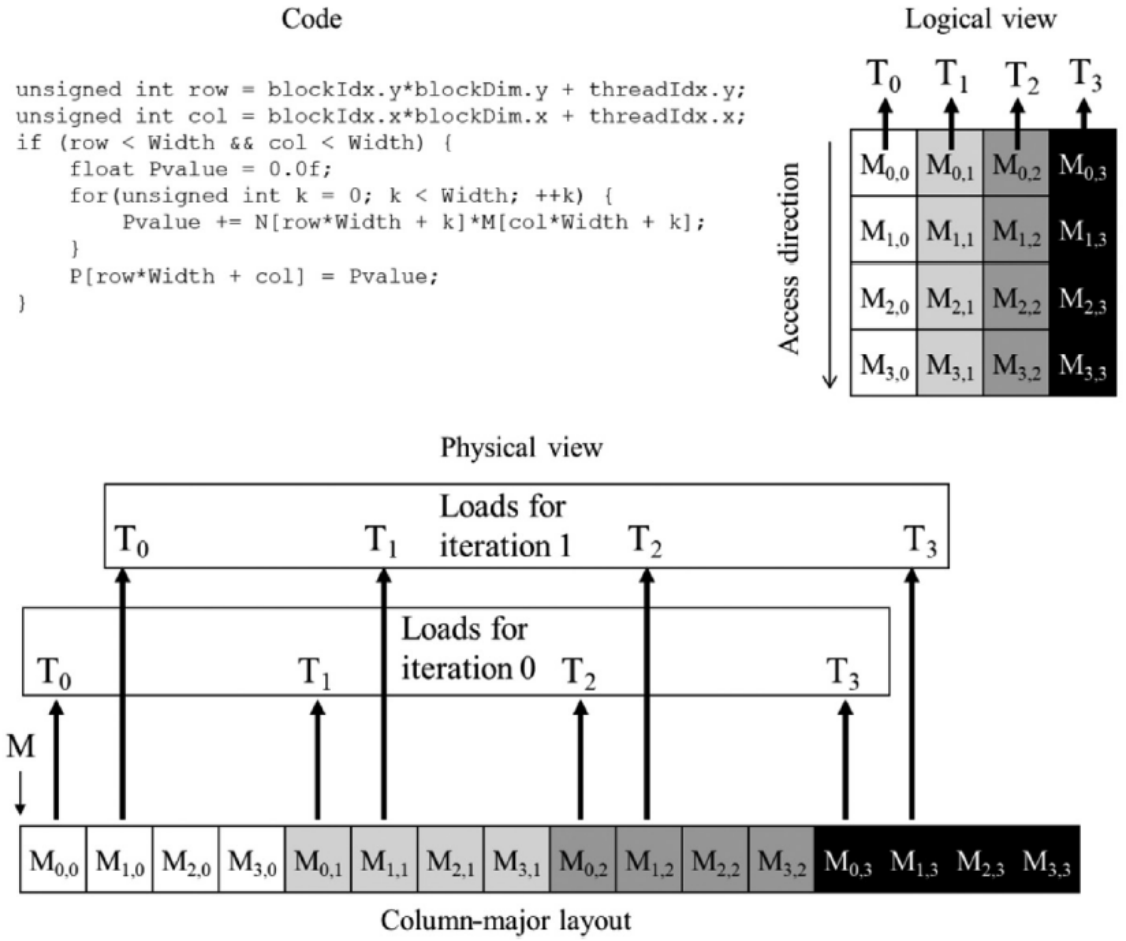


FIGURE 6.3

An uncoalesced access pattern.

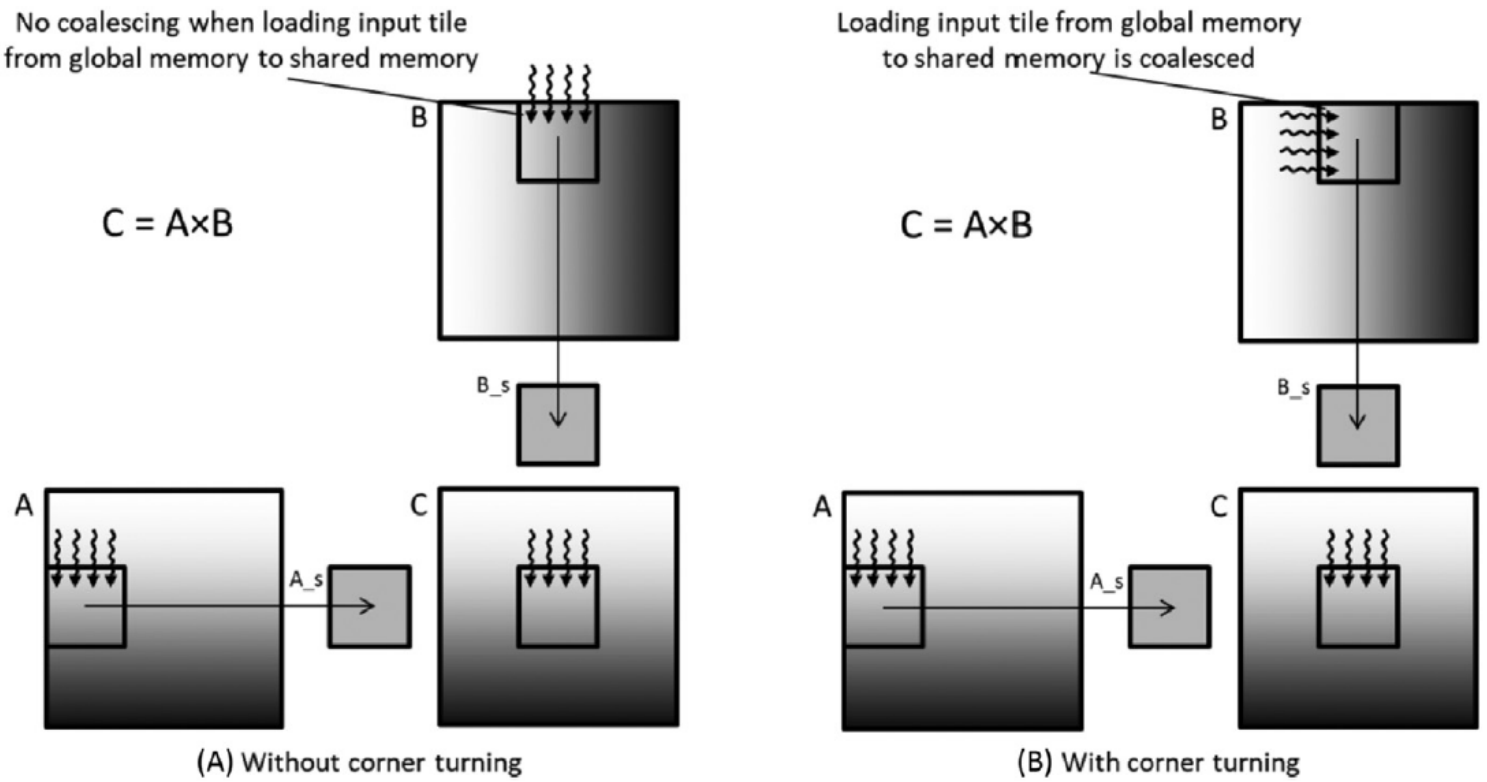


FIGURE 6.4

Applying corner turning to coalesce accesses to matrix B, which is stored in column-major layout.

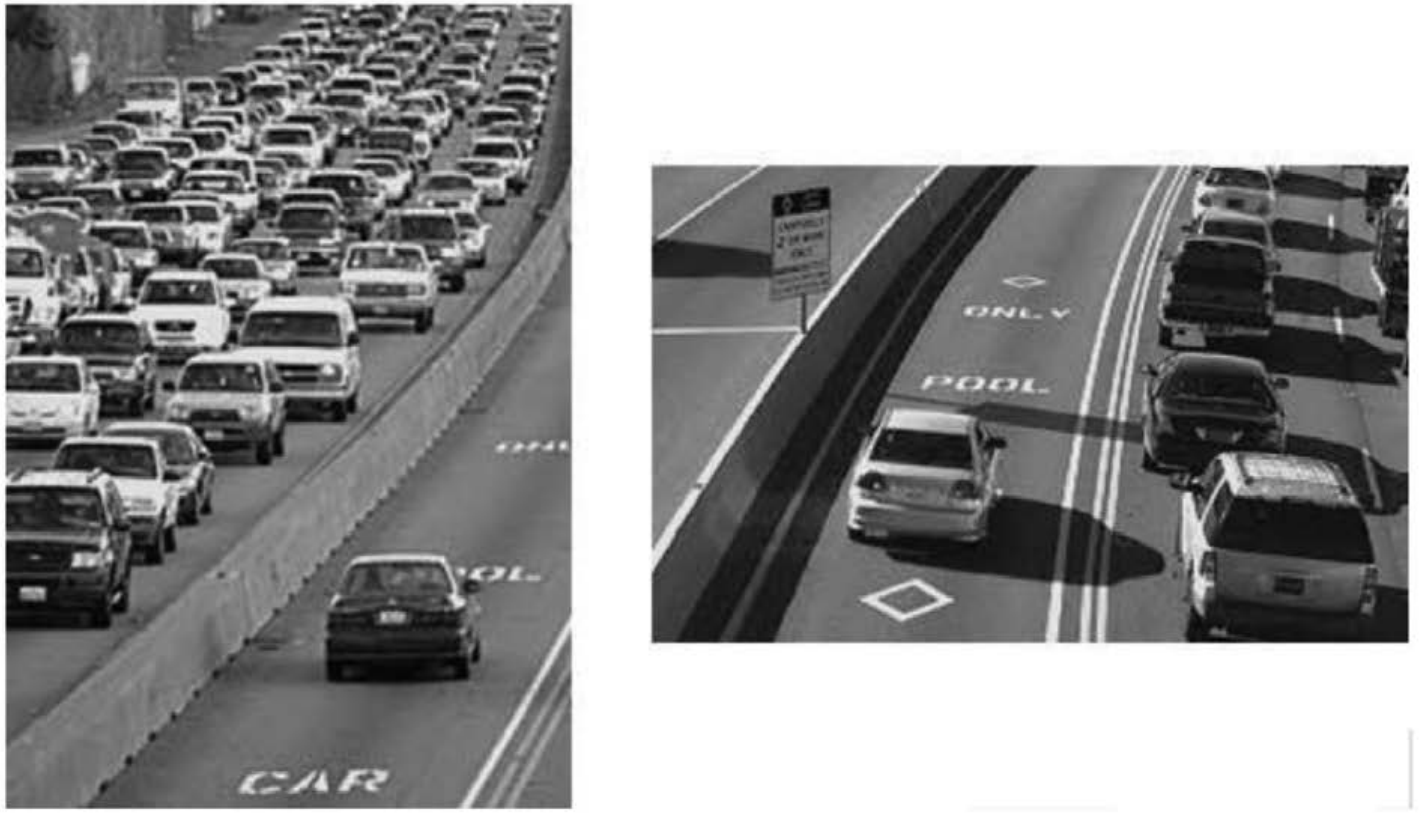


FIGURE 6.5

Reducing traffic congestion in highway systems.

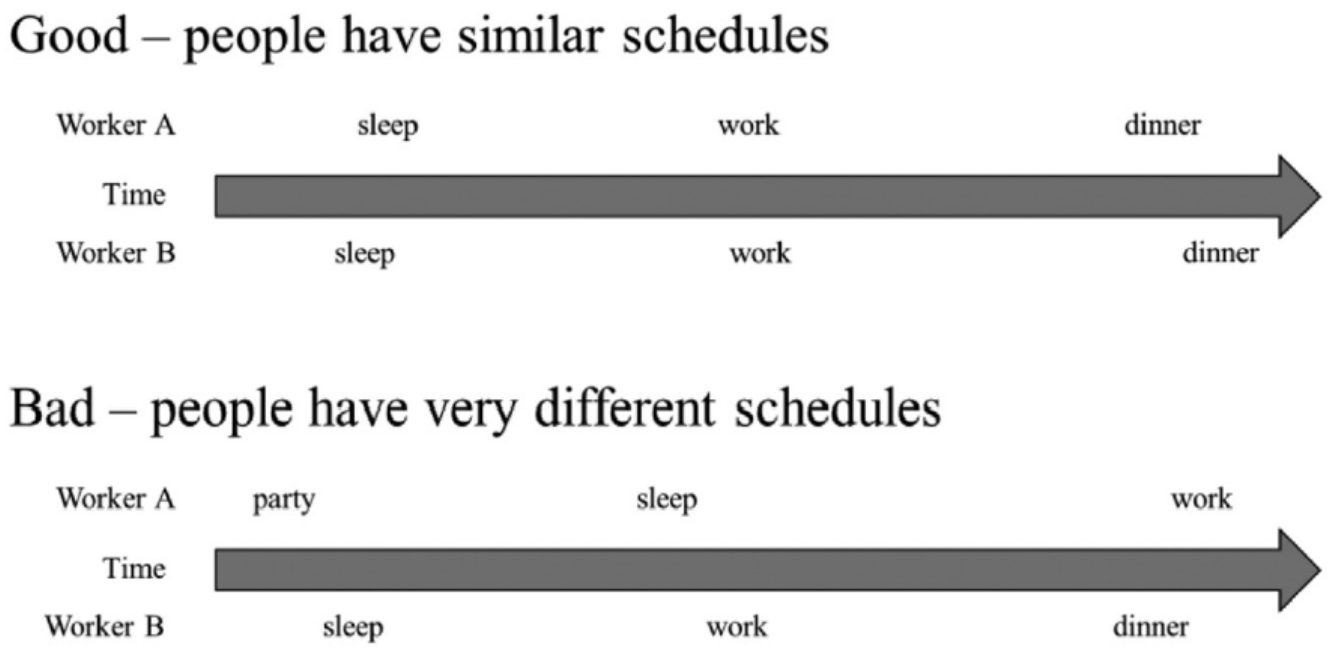


FIGURE 6.6

Carpooling requires synchronization among people.

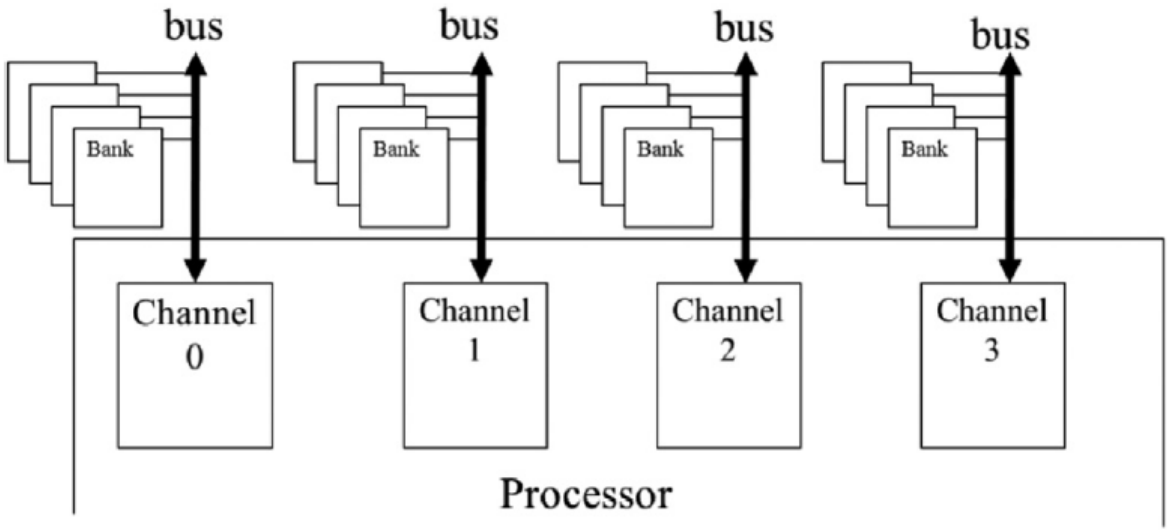


FIGURE 6.7

Channels and banks in DRAM systems.

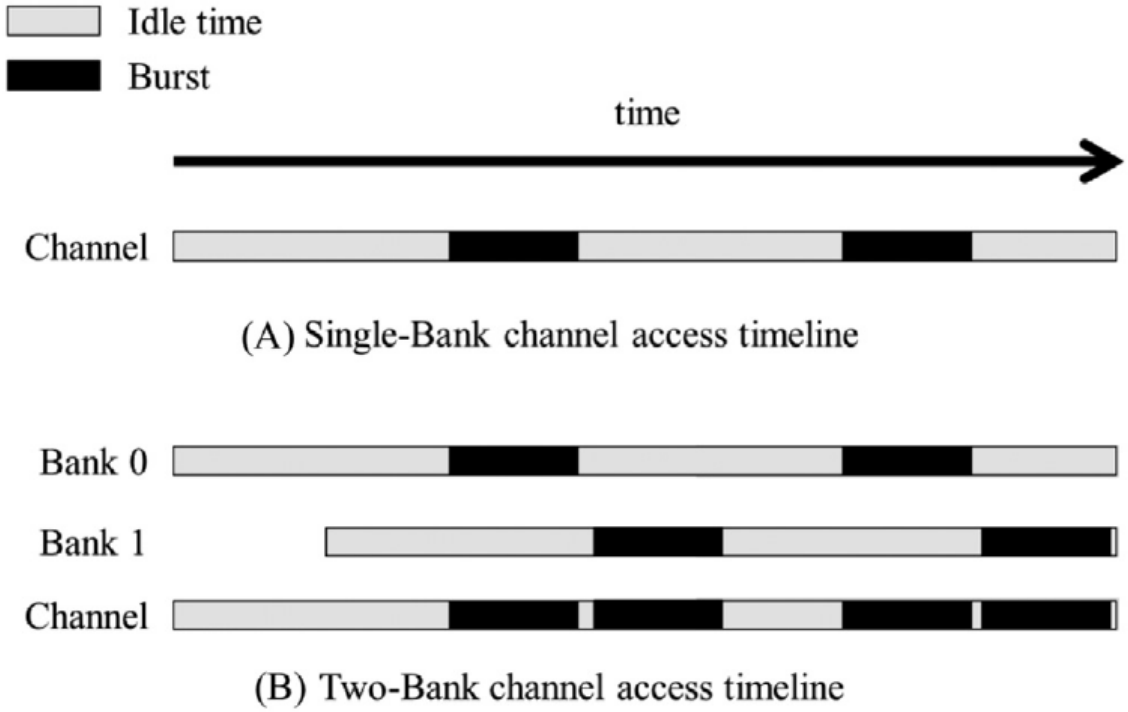


FIGURE 6.8

Banking improves the utilization of data transfer bandwidth of a channel.

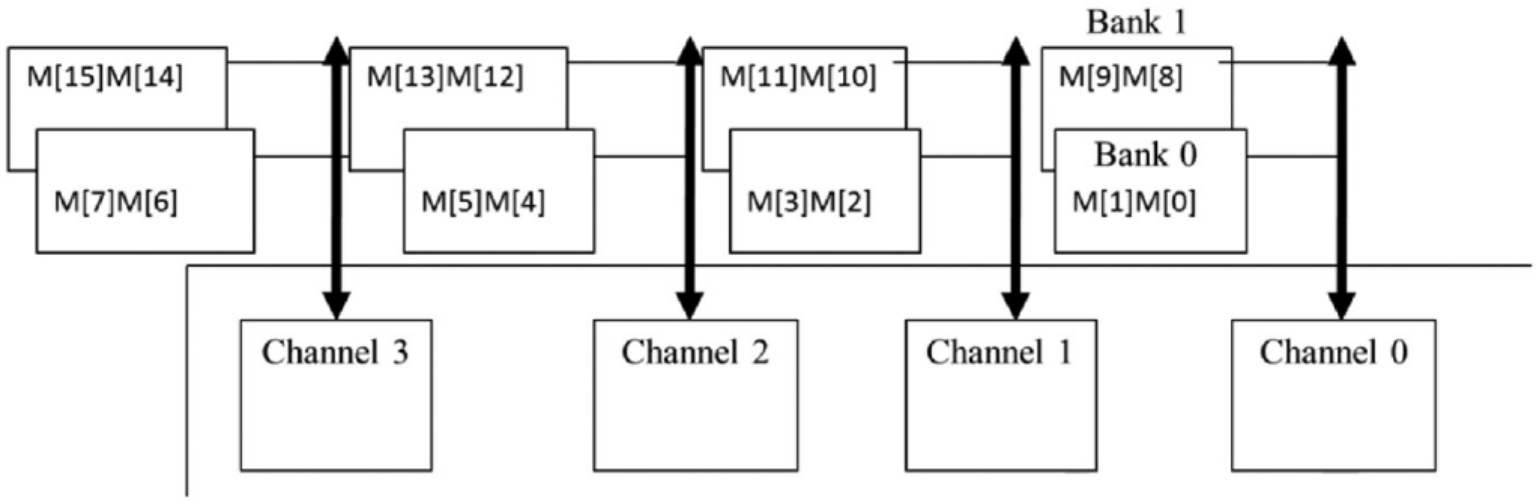


FIGURE 6.9

Distributing array elements into channels and banks.

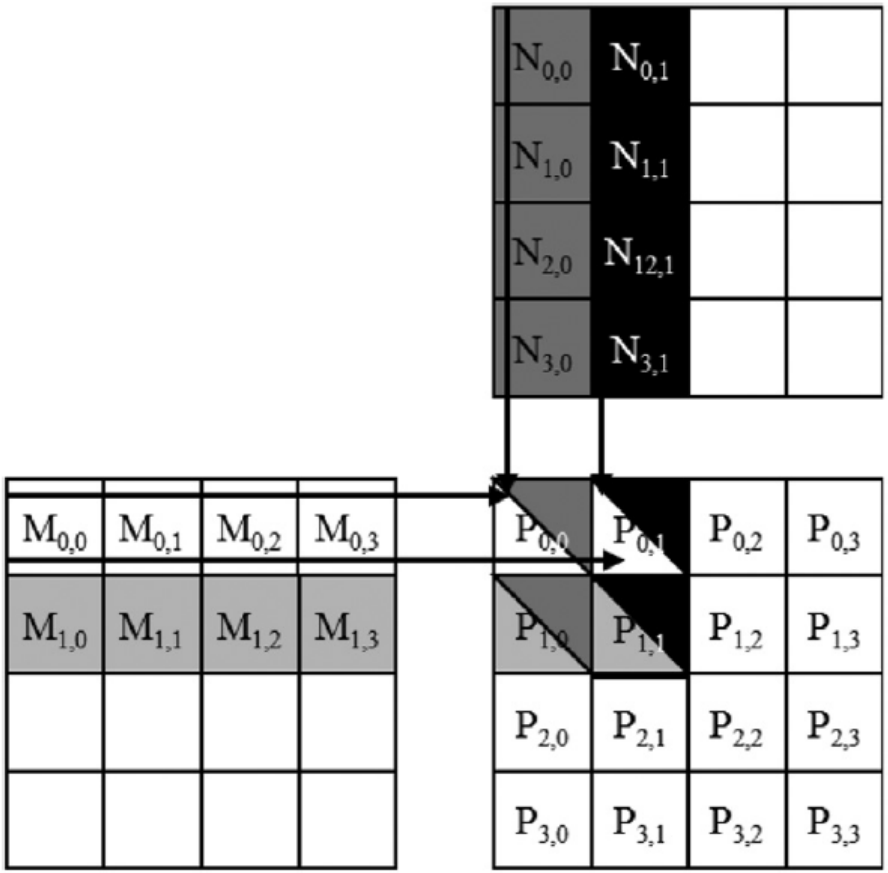


FIGURE 6.10

A small example of matrix multiplication (replicated from Fig. 5.5).

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[19], 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 6.11

M elements loaded by thread blocks in each phase.

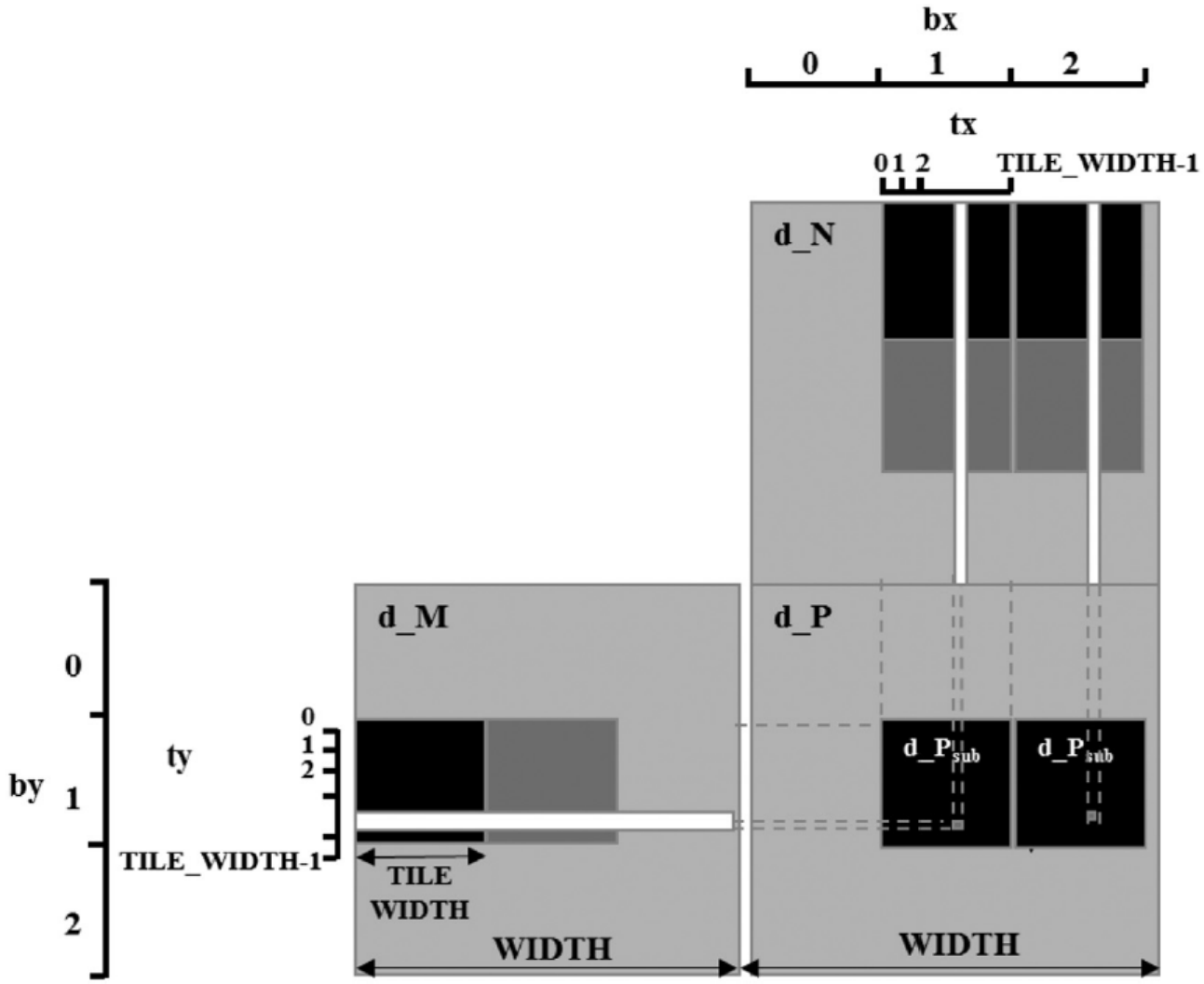


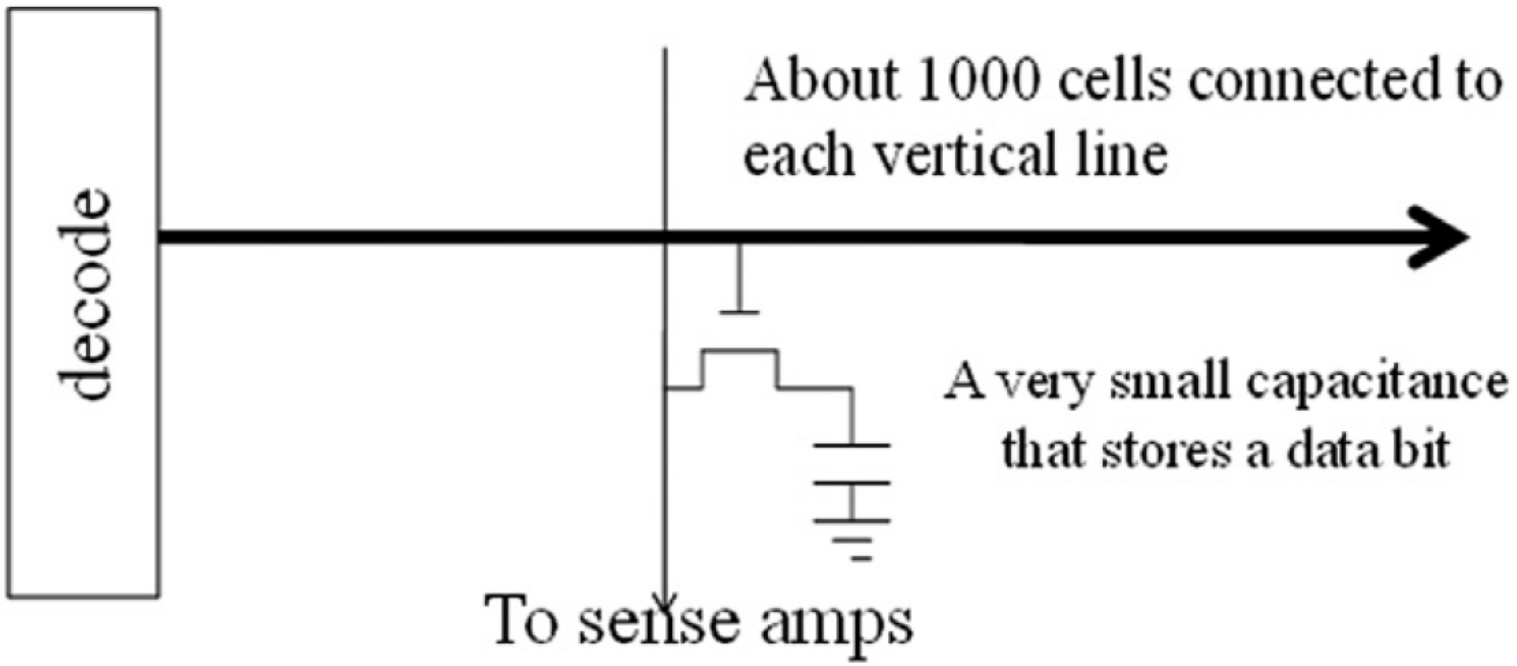
FIGURE 6.12

Thread coarsening for tiled matrix multiplication.

```
01    #define TILE_WIDTH    32
02    #define COARSE_FACTOR 4
03     __global__ void matrixMulKernel(float* M, float* N, float* P, int width)
04    {
05
06          __shared__   float Mds[TILE WIDTH][TILE WIDTH];
07          __shared__   float Nds[TILE WIDTH][TILE WIDTH];
08
09        int bx = blockIdx.x;  int by = blockIdx.y;
10        int tx = threadIdx.x; int ty = threadIdx.y;
11
12        // Identify the row and column of the P element to work on
13        int row = by*TILE_WIDTH + ty;
14        int colStart = bx*TILE_WIDTH*COARSE_FACTOR + tx;
15
16        // Initialize Pvalue for all output elements
17        float Pvalue[COARSE FACTOR];
18        for(int c = 0; c < COARSE FACTOR; ++c) {
19            Pvalue[c] = 0.0f;
20        }
21
22        // Loop over the M and N tiles required to compute P element
23        for(int ph = 0; ph < width/TILE_WIDTH; ++ph) {
24
25            // Collaborative loading of M tile into shared memory
26            Mds[ty][tx] = M[row*width + ph*TILE_WIDTH + tx];
27
28            for(int c = 0; c < COARSE FACTOR; ++c) {
29
30                int col = colStart + c*TILE WIDTH;
31
32                // Collaborative loading of N tile into shared memory
33                Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*width + col];
34                __syncthreads();
35
36                for(int k = 0; k < TILE_WIDTH; ++k) {
37                    Pvalue[c] += Mds[ty][k]*Nds[k][tx];
38                }
39                  __syncthreads();
40
41            }
42
43        }
44
45        for(int c = 0; c < COARSE_FACTOR; ++c) {
46            int col = colStart + c*TILE_WIDTH;
47            P[row*width + col] = Pvalue[c];
48        }
49
50    }
```

FIGURE 6.13

Code for thread coarsening for tiled matrix multiplication.



In-text figure 1

```
01    __global__ void foo_kernel(float* a, float* b, float* c, float* d, float* e) {
02        unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
03        __shared__ float a_s[256];
04        __shared__ float bc_s[4*256];
05        a_s[threadIdx.x] = a[i];
06        for(unsigned int j = 0; j < 4; ++j) {
07            bc_s[j*256 + threadIdx.x] = b[j*blockDim.x*gridDim.x + i] + c[i*4 + j];
08        }
09        __syncthreads();
10        d[i + 8] = a_s[threadIdx.x];
11        e[i*8] = bc_s[threadIdx.x*4];
12    }
```

In-text figure 2

Table 6.1 A checklist of optimizations.

Optimization	Benefit to compute cores	Benefit to memory	Strategies
Maximizing occupancy	More work to hide pipeline latency	More parallel memory accesses to hide DRAM latency	Tuning usage of SM resources such as threads per block, shared memory per block, and registers per thread
Enabling coalesced global memory accesses	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic and better utilization of bursts/ cache lines	Transfer between global memory and shared memory in a coalesced manner and performing uncoalesced accesses in shared memory (e.g., corner turning) Rearranging the mapping of threads to data Rearranging the layout of the data
Minimizing control divergence	High SIMD efficiency (fewer idle cores during SIMD execution)	–	Rearranging the mapping of threads to work and/or data Rearranging the layout of the data
Tiling of reused data	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic	Placing data that is reused within a block in shared memory or registers so that it is transferred between global memory and the SM only once
Privatization (covered later)	Fewer pipeline stalls waiting for atomic updates	Less contention and serialization of atomic updates	Applying partial updates to a private copy of the data and then updating the universal copy when done
Thread coarsening	Less redundant work, divergence, or synchronization	Less redundant global memory traffic	Assigning multiple units of parallelism to each thread to reduce the price of parallelism when it is incurred unnecessarily