

Chapter-7

Parallel patterns: convolution

An introduction to stencil computation

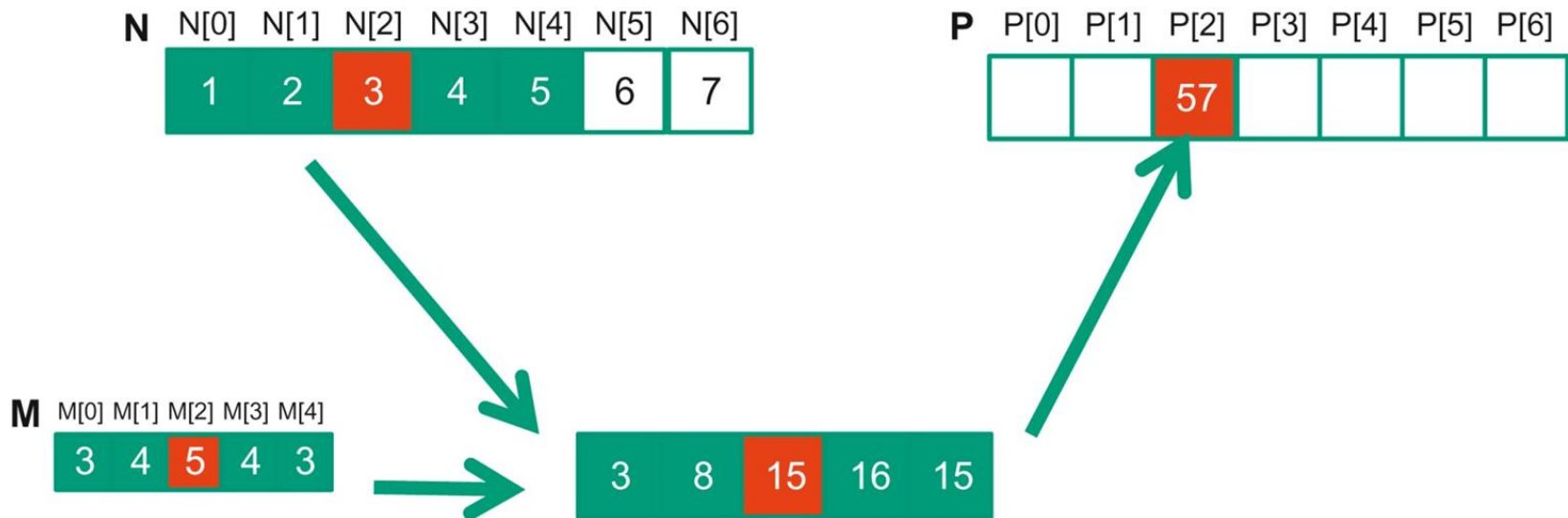


FIGURE 7.1: A 1D convolution example, inside elements.

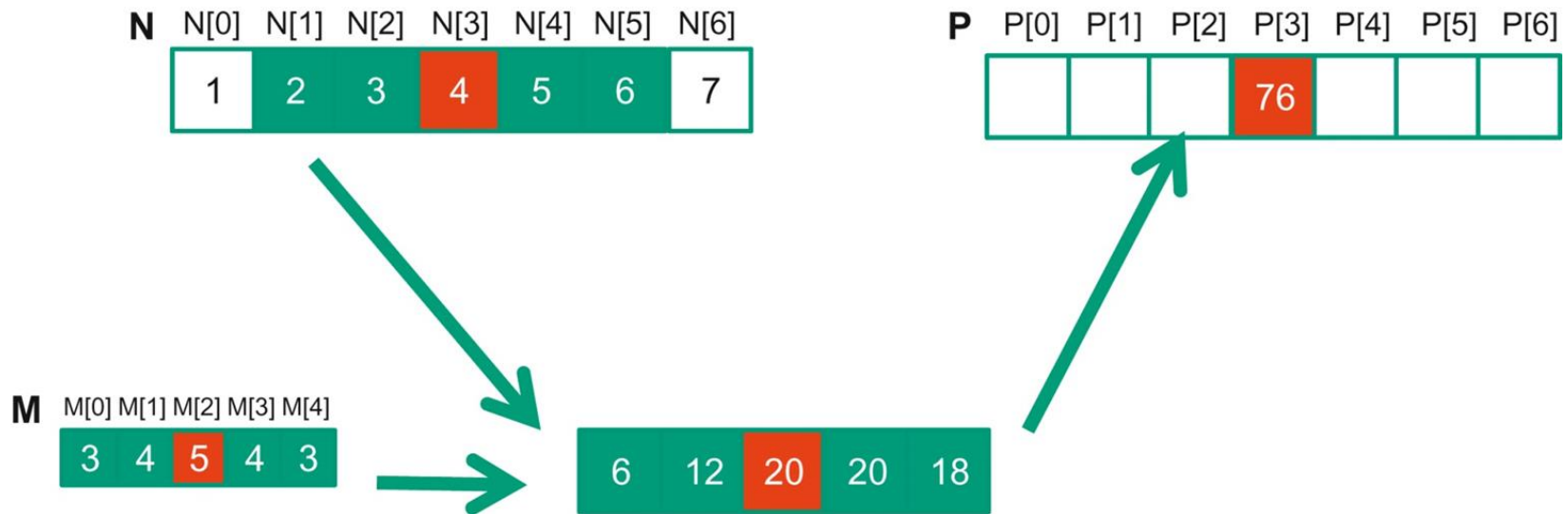


FIGURE 7.2: 1D convolution, calculation of $P[3]$.

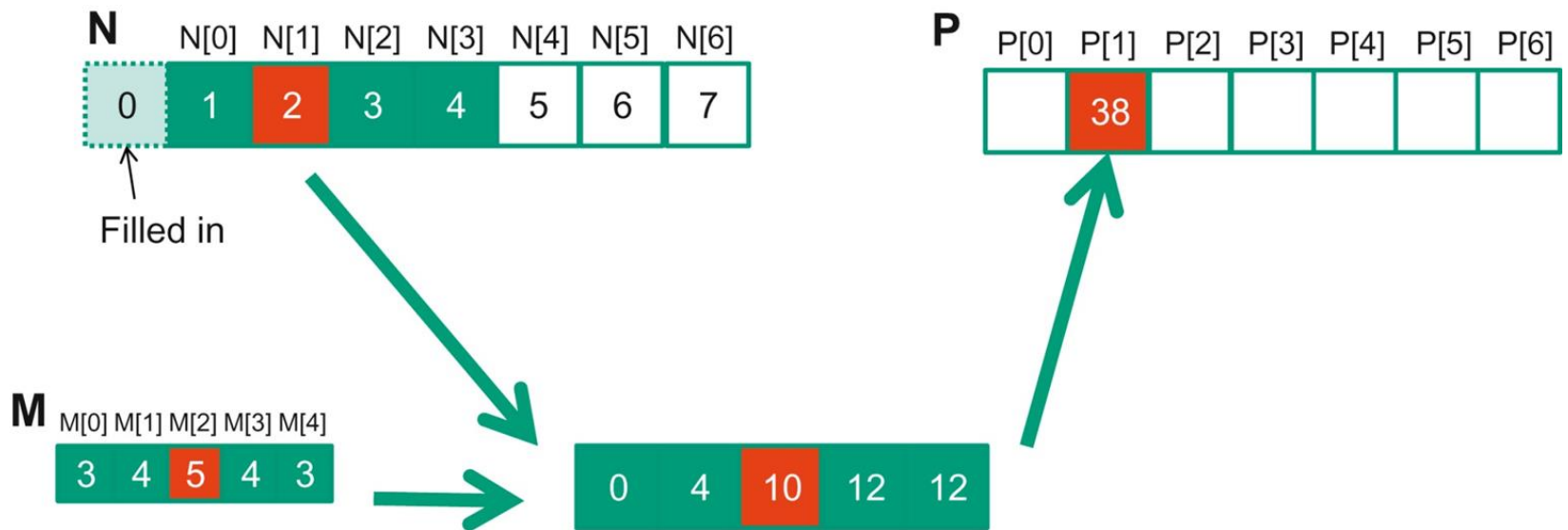


FIGURE 7.3: 1D convolution boundary condition.

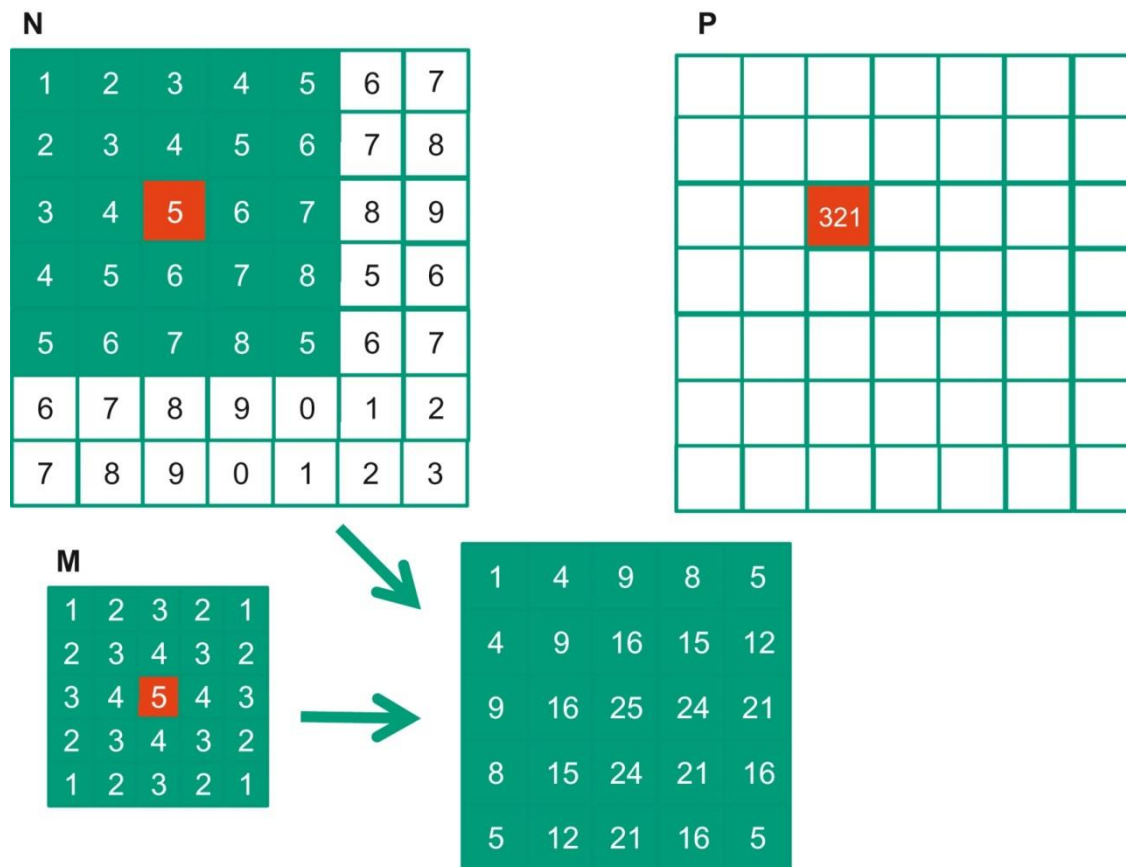


FIGURE 7.4: A 2D convolution example.

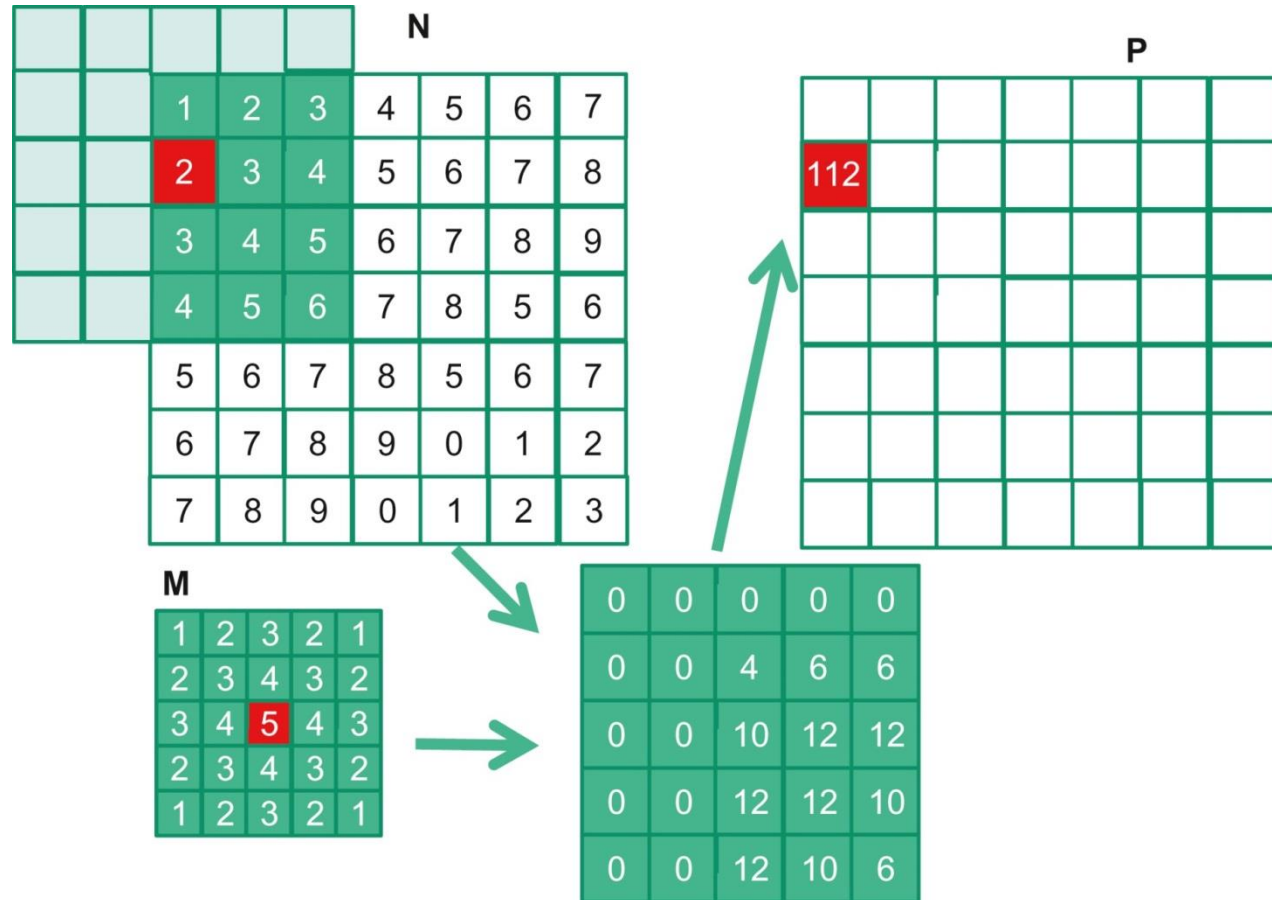


FIGURE 7.5: A 2D convolution boundary condition.

```

__global__ void convolution_1D_basic_kernel(float *N, float *M, float *P,
int Mask_Width, int Width) {

    int i = blockIdx.x*blockDim.x + threadIdx.x;

    float Pvalue = 0;
    int N_start_point = i - (Mask_Width/2);
    for (int j = 0; j < Mask_Width; j++) {
        if (N_start_point + j >= 0 && N_start_point + j < Width) {
            Pvalue += N[N_start_point + j]*M[j];
        }
    }
    P[i] = Pvalue;
}

```

FIGURE 7.6: A 1D convolution kernel with boundary condition handling.

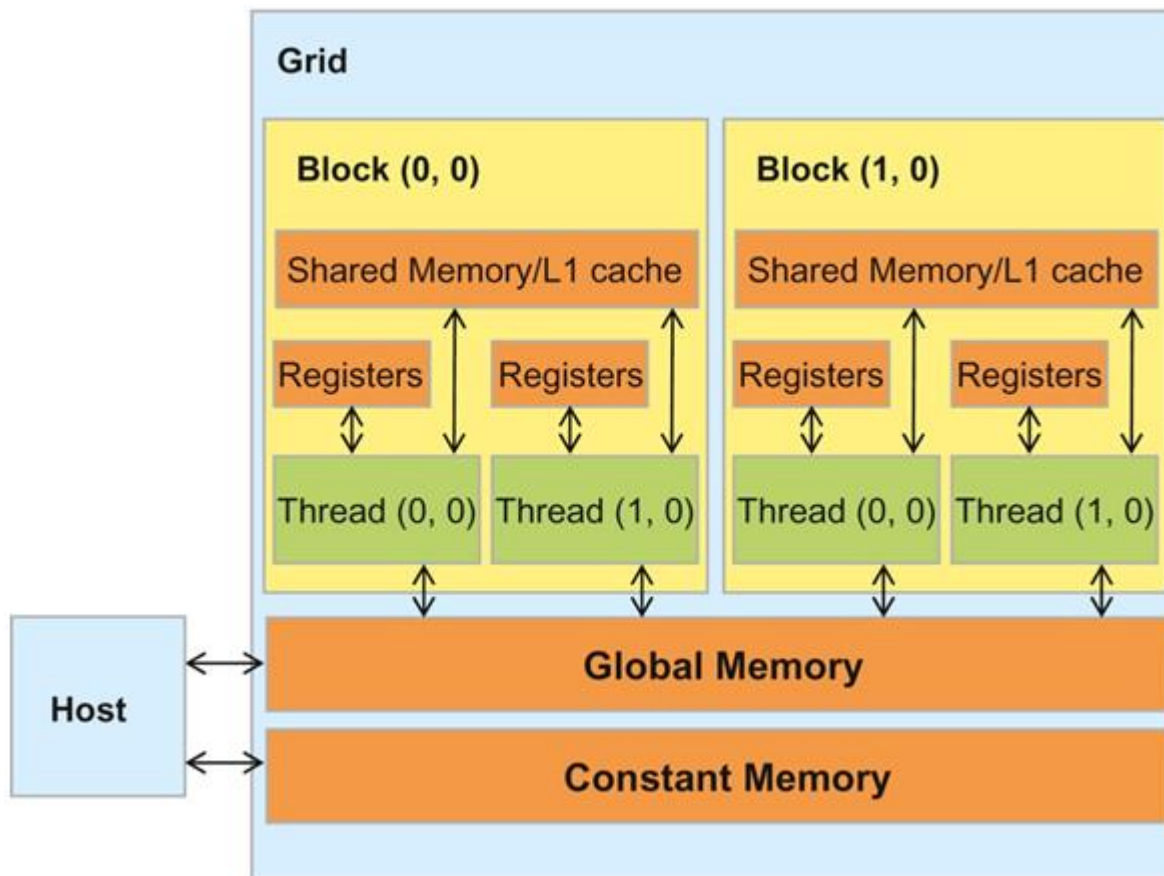


FIGURE 7.7: A review of the CUDA Memory Model.


```

__global__ void convolution_1D_ba sic_kernel(float *N, float *P, int Mask_Width,
int Width) {

    int i = blockIdx.x*blockDim.x + threadIdx.x;

    float Pvalue = 0;
    int N_start_point = i - (Mask_Width/2);
    for (int j = 0; j < Mask_Width; j++) {
        if (N_start_point + j >= 0 && N_start_point + j < Width) {
            Pvalue += N[N_start_point + j]*M[j];
        }
    }
    P[i] = Pvalue;
}

```

FIGURE 7.8: A 1D convolution kernel using constant memory for M.

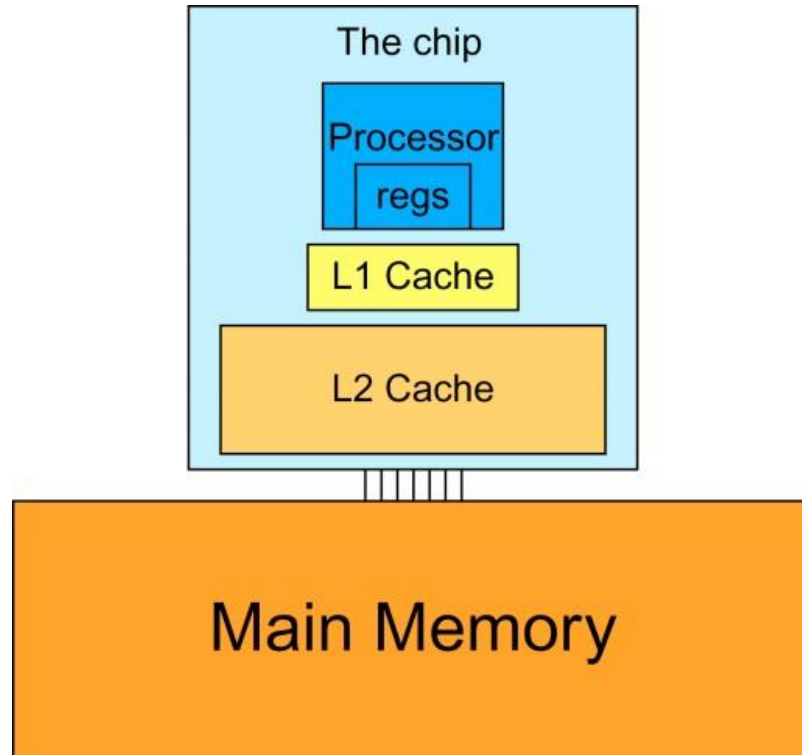


FIGURE 7.9: A simplified view of the cache hierarchy of modern processors.

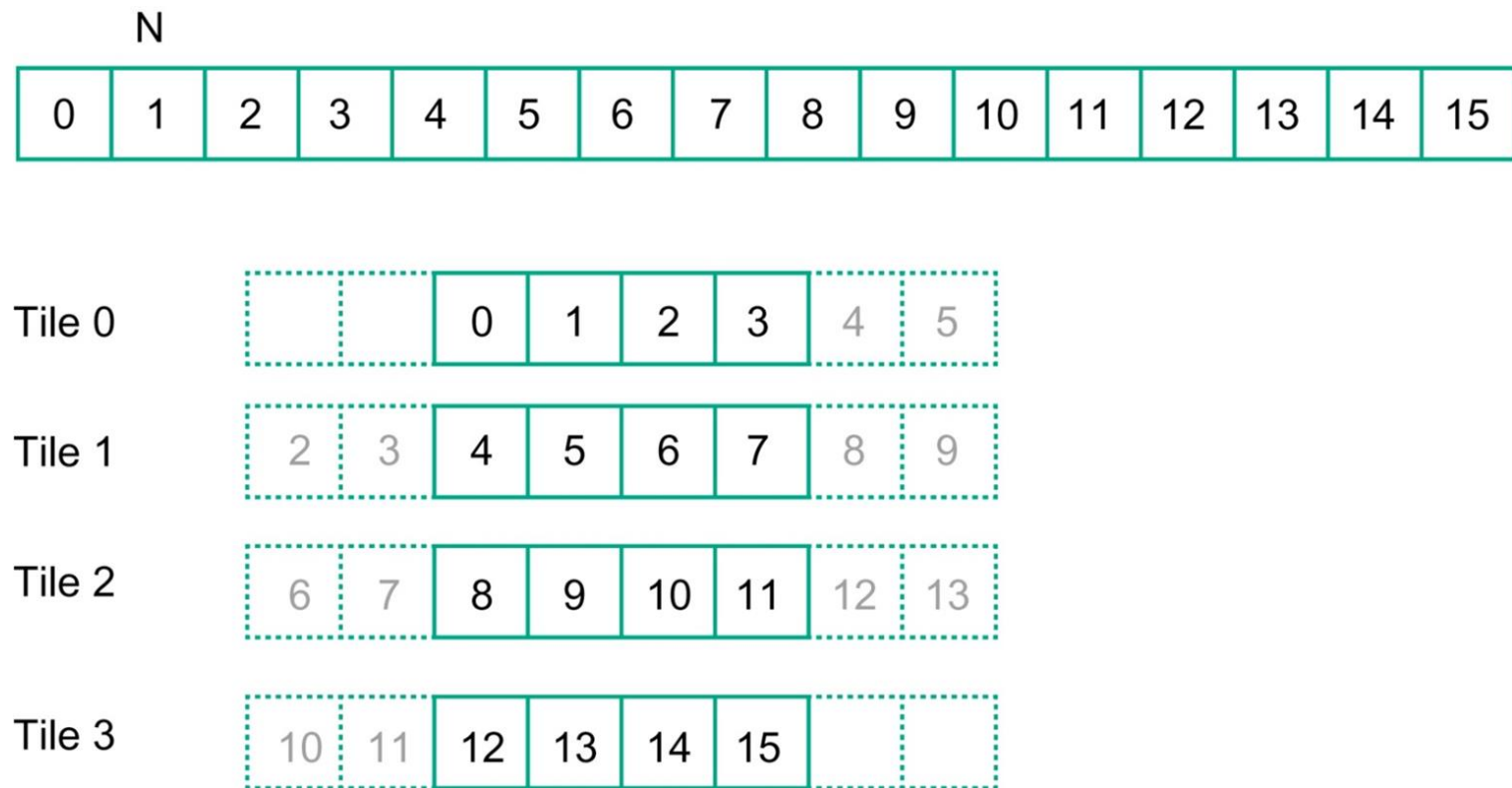


FIGURE 7.10: A 1D tiled convolution example.

```

__global__ void convolution_1D_tiled_kernel(float *N, float *P, int Mask_Width,
int Width) {

    int i = blockIdx.x*blockDim.x + threadIdx.x;
    __shared__ float  N_ds[TILE_SIZE + MAX_MASK_WIDTH - 1];

    int n = Mask_Width/2;

    int halo_index_left = (blockIdx.x - 1)*blockDim.x + threadIdx.x;
    if (threadIdx.x >= blockDim.x - n) {
        N_ds[threadIdx.x - (blockDim.x - n)] =
            (halo_index_left < 0) ? 0 : N[halo_index_left];
    }

    N_ds[n + threadIdx.x] = N[blockIdx.x*blockDim.x + threadIdx.x];

    int halo_index_right = (blockIdx.x + 1)*blockDim.x + threadIdx.x;
    if (threadIdx.x < n) {
        N_ds[n + blockDim.x + threadIdx.x] =
            (halo_index_right >= Width) ? 0 : N[halo_index_right];
    }

    __syncthreads();

    float Pvalue = 0;
    for(int j = 0; j < Mask_Width; j++) {
        Pvalue += N_ds[threadIdx.x + j]*M[j];
    }
    P[i] = Pvalue;
}

```

FIGURE 7.11: A tiled 1D convolution kernel using constant memory for M.

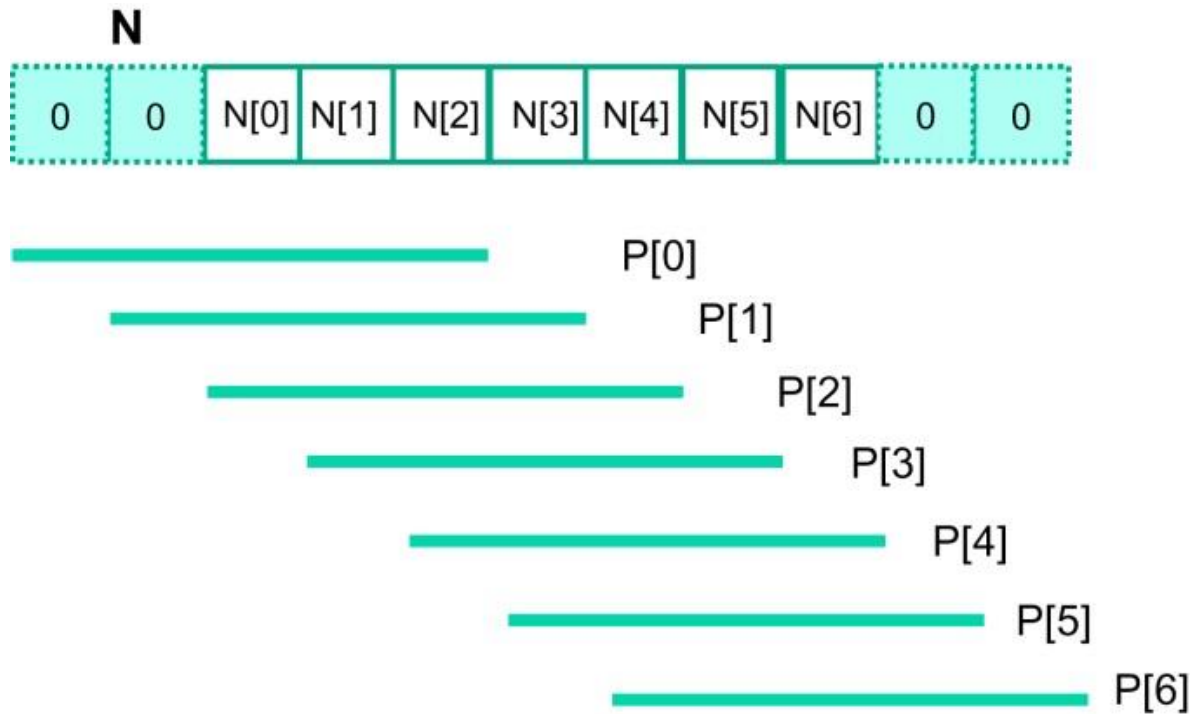


FIGURE 7.12: A small example of accessing N elements and ghost cells.

```

__syncthreads();

int This_tile_start_point = blockIdx.x * blockDim.x;
int Next_tile_start_point = (blockIdx.x + 1) * blockDim.x;
int N_start_point = i - (Mask_Width/2);
float Pvalue = 0;
for (int j = 0; j < Mask_Width; j++) {
    int N_index = N_start_point + j;
    if (N_index >= 0 && N_index < Width) {
        if ((N_index >= This_tile_start_point)
            && (N_index < Next_tile_start_point)) {
            Pvalue += N_ds[threadIdx.x+j-(Mask_Width/2)]*M[j];
        } else {
            Pvalue += N[N_index] * M[j];
        }
    }
}
P[i] = Pvalue;

```

FIGURE 7.13: Using general caching for halo cells.

```

__global__ void convolution_1D_tiled_caching_kernel(float *N, float *P, int
Mask_Width,int Width) {

    int i = blockIdx.x*blockDim.x + threadIdx.x;
    __shared__ float  N_ds[TILE_SIZE];

    N_ds[threadIdx.x] = N[i];

    __syncthreads();

    int This_tile_start_point = blockIdx.x * blockDim.x;
    int Next_tile_start_point = (blockIdx.x + 1) * blockDim.x;
    int N_start_point = i - (Mask_Width/2);
    float Pvalue = 0;
    for (int j = 0; j < Mask_Width; j++) {
        int N_index = N_start_point + j;
        if (N_index >= 0  && N_index < Width) {
            if ((N_index >= This_tile_start_point)
                && (N_index < Next_tile_start_point)) {
                Pvalue += N_ds[threadIdx.x+j-(Mask_Width/2)]*M[j];
            } else {
                Pvalue += N[N_index] * M[j];
            }
        }
    }
    P[i] = Pvalue;
}

```

FIGURE 7.14: A simpler tiled 1D convolution kernel using constant memory and general caching.

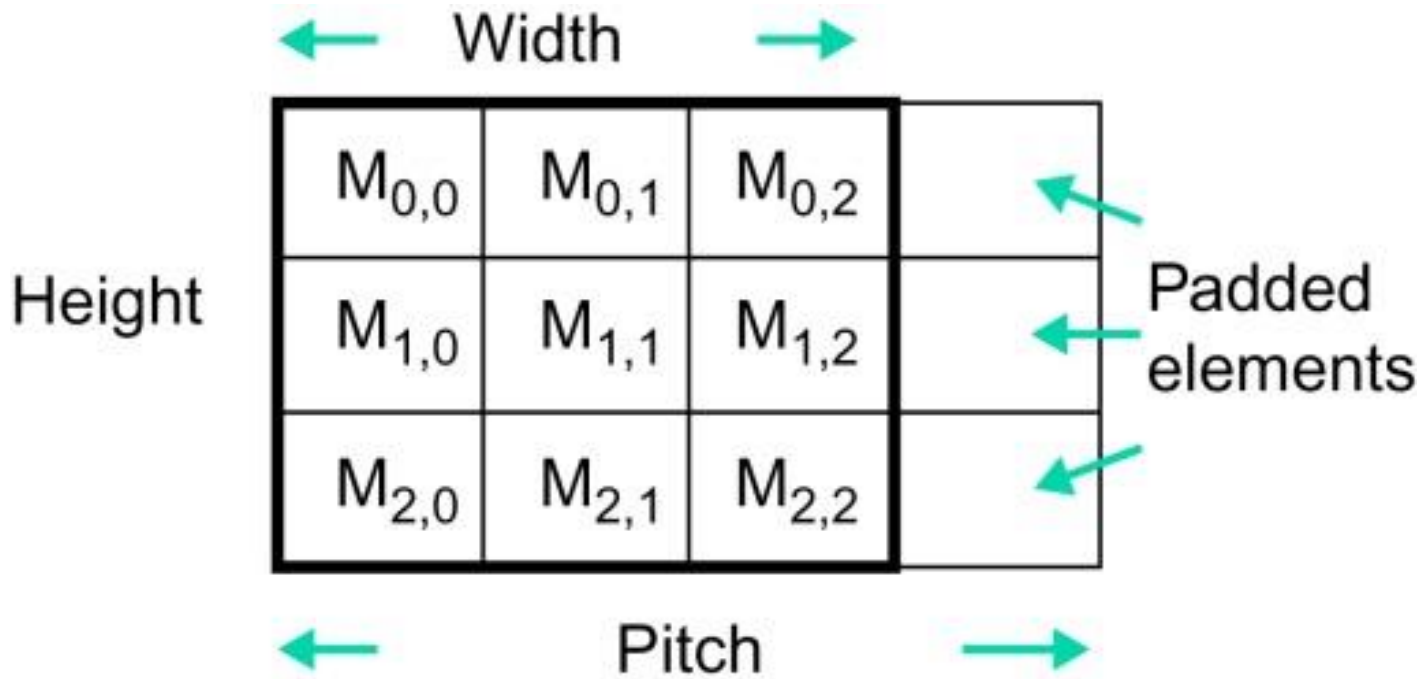


FIGURE 7.15: A padded image format and the concept of pitch.

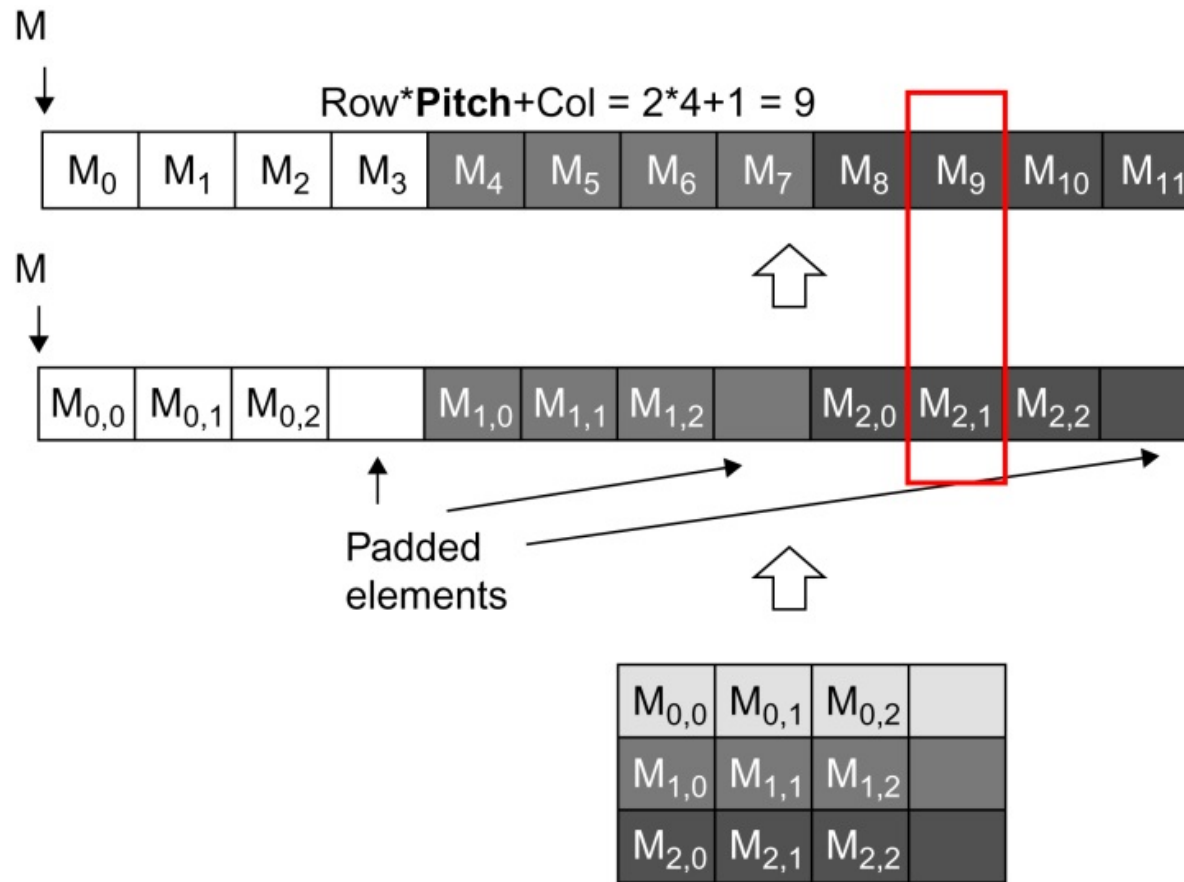


FIGURE 7.16: Row-major layout of a 2D image matrix with padded elements.

```
// Image Matrix Structure declaration
//
typedef struct {
    int width;
    int height;
    int pitch;
    int channels;
    float* data;
} * wbImage_t;
```

FIGURE 7.17: The C type structure definition of the image pixel element.

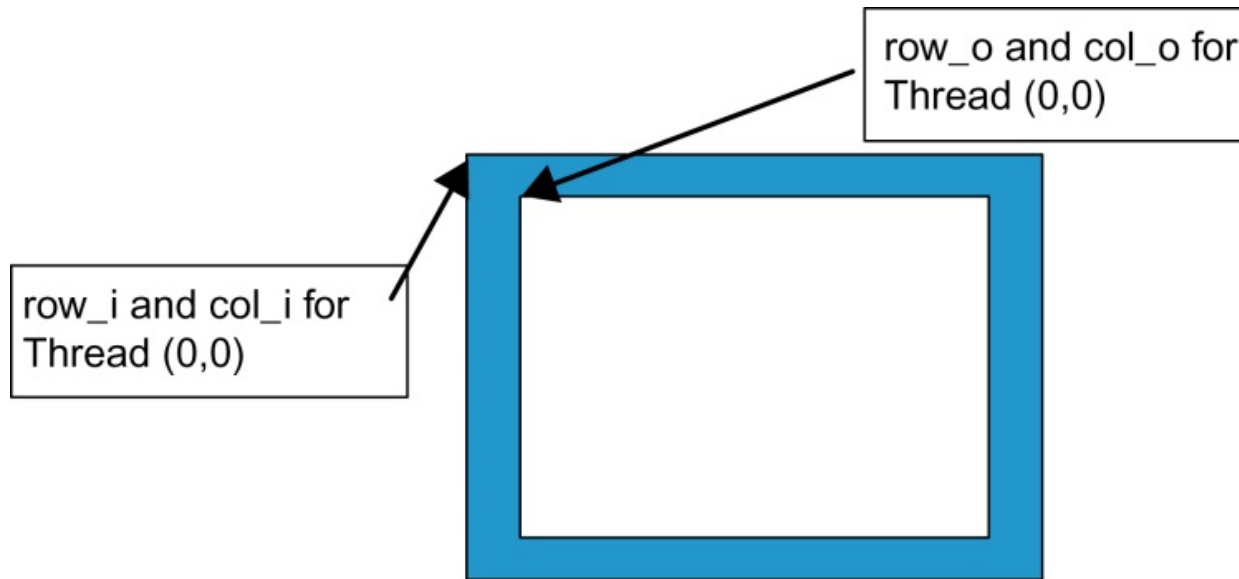


FIGURE 7.18: Starting element indices of the input tile versus output tile.

```

__global__ void convolution_2D_tiled_kernel(float *P, float *N, int height, int width,
                                           int pitch, int channels, int Mask_Width,
                                           const float __restrict__ *M)
{
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int row_o = blockIdx.y*O_TILE_WIDTH + ty;
    int col_o = blockIdx.x*O_TILE_WIDTH + tx;

    int row_i = row_o - Mask_Width/2;
    int col_i = col_o - Mask_Width/2;

```

FIGURE 7.19: Part 1 of a 2D convolution kernel.

```

__shared__ float N_ds[TILE_SIZE+MAX_MASK_WIDTH-1]
                [TILE_SIZE+MAX_MASK_HEIGHT-1];

if((row_i >= 0) && (row_i < height) &&
    (col_i >= 0) && (col_i < width)) {
    N_ds[ty][tx] = data[row_i * pitch + col_i];
} else{
    N_ds[ty][tx] = 0.0f;
}

```

FIGURE 7.20: Part 2 of a 2D convolution kernel.

```

float output = 0.0f;
if(ty < O_TILE_WIDTH && tx < O_TILE_WIDTH){
    for(i = 0; i < MASK_WIDTH; i++) {
        for(j = 0; j < MASK_WIDTH; j++) {
            output += M[i][j] * N_ds[i+ty][j+tx];
        }
    }

    if(row_o < height && col_o < width){
        data[row_o*width + col_o] = output;
    }
}

```

FIGURE 7.21: Part 3 of a 2D convolution kernel.

TiILE_WIDTH	8	16	32	64
Reduction Mask_Width = 5	11.1	16	19.7	22.1
Reduction Mask_Width = 9	20.3	36	51.8	64

FIGURE 7.22: Image array access reduction ratio for different tile sizes.