CHAPTER 5

Memory architecture and data locality

```
for (int k = 0; k < Width; ++k) {
    Pvalue += M[row*Width+k] * N[k*Width+col];
}
```

FIGURE 5.1

The most executed part of the matrix multiplication kernel in Fig. 3.11.

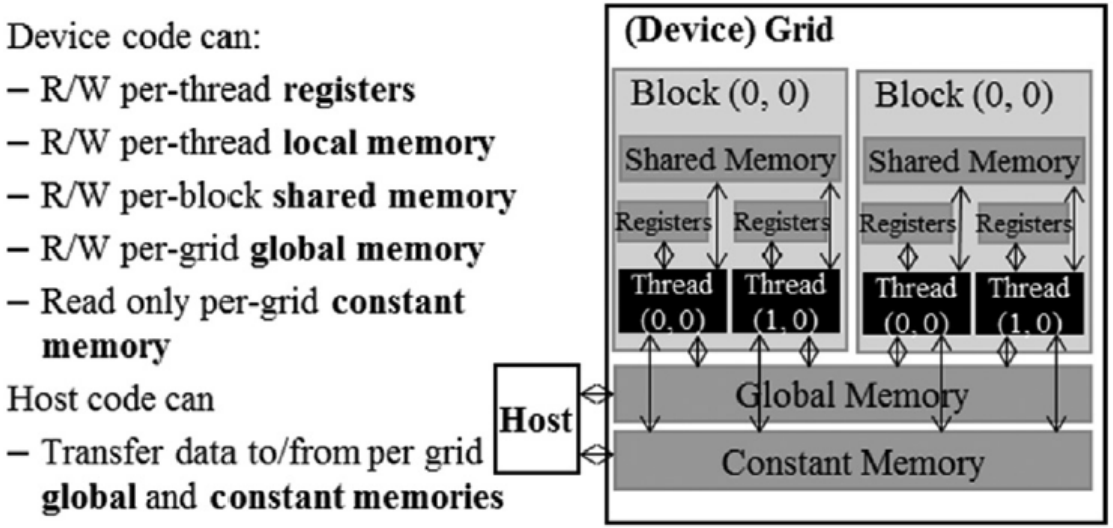


FIGURE 5.2

An (incomplete) overview of the CUDA device memory model. An important type of CUDA memory that is not shown in this figure is the texture memory, since its use is not covered in this textbook.

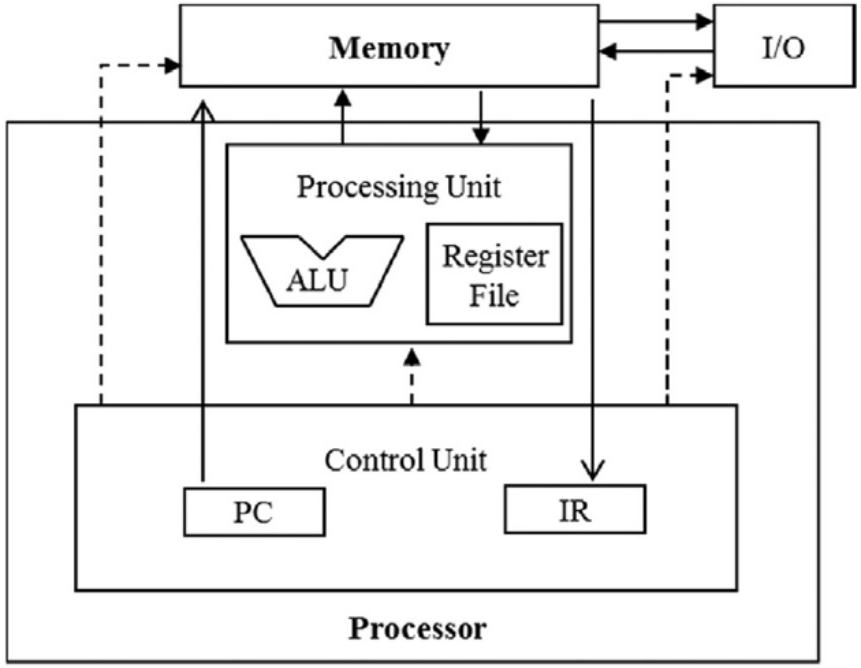


FIGURE 5.3

Memory versus registers in a modern computer based on the von Neumann model.

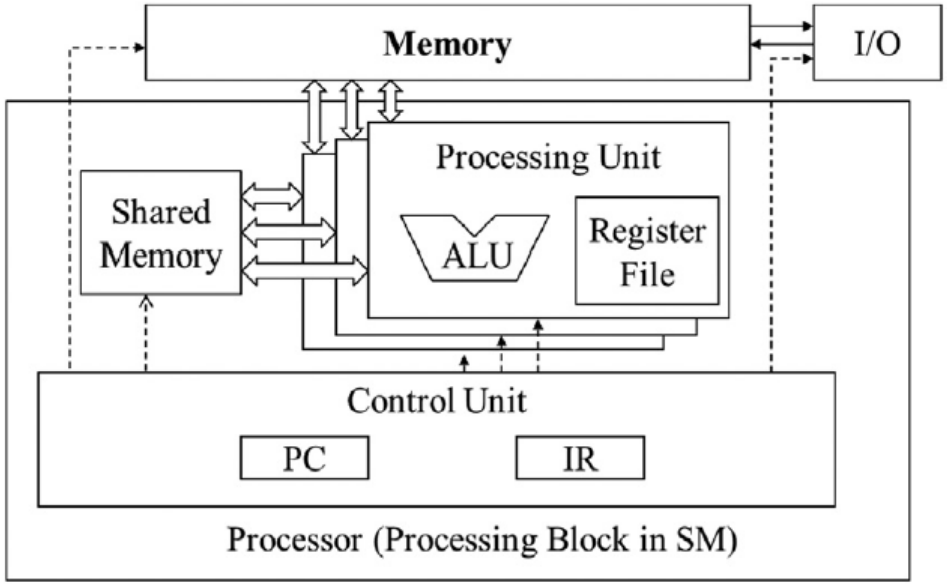


FIGURE 5.4

Shared memory versus registers in a CUDA device SM.

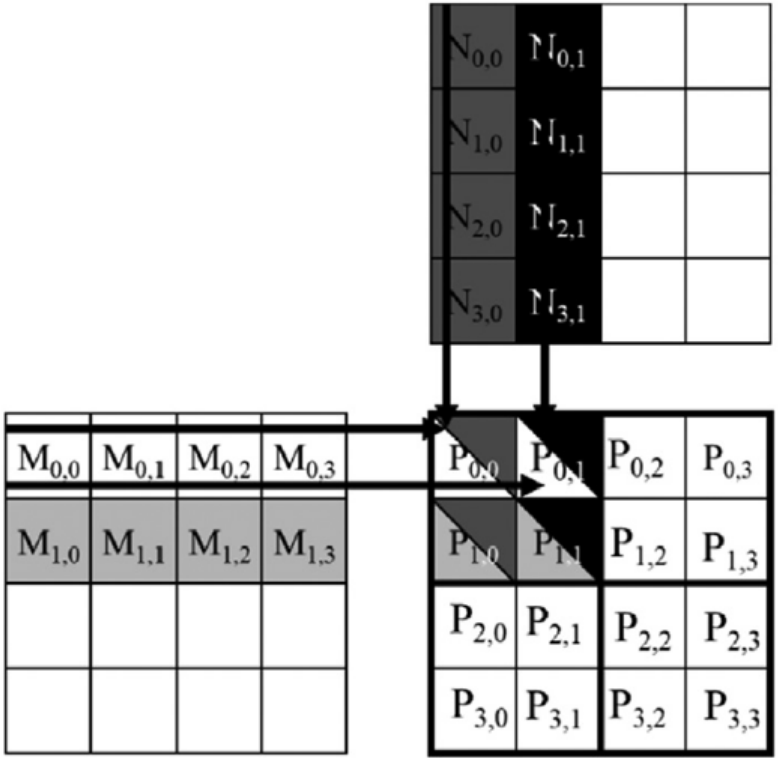


FIGURE 5.5

A small example of matrix multiplication. For brevity we show M[y*Width+x], N[y*Width +x], P[y*Width+x] as $M_{y,x}$, $N_{y,x}$ $P_{y,x}$, respectively.

Access order →

$thread_{0,0}$	$M_{0,0} * N_{0,0}$	$M_{0,1} * N_{1,0}$	$M_{0,2} * N_{2,0}$	$M_{0,3} * N_{3,0}$
$thread_{0,1}$	$M_{0,0} * N_{0,1}$	$M_{0,1} * N_{1,1}$	$M_{0,2} * N_{2,1}$	$M_{0,3} * N_{3,1}$
$thread_{1,0}$	$M_{1,0} * N_{0,0}$	$M_{1,1} * N_{1,0}$	$M_{1,2} * N_{2,0}$	$M_{1,3} * N_{3,0}$
$thread_{1,1}$	$M_{1,0} * N_{0,1}$	$M_{1,1} * N_{1,1}$	$M_{1,2} * N_{2,1}$	$M_{1,3} * N_{3,1}$

FIGURE 5.6

Global memory accesses performed by threads in $block_{0,0}$.

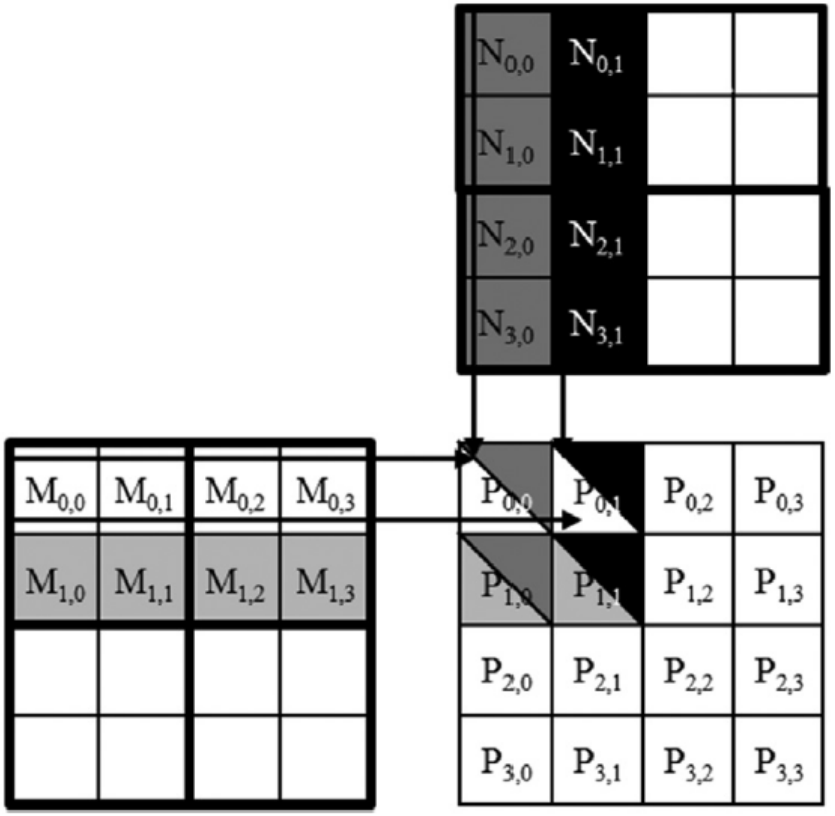


FIGURE 5.7

Tiling M and N to utilize shared memory.

	Phase 0			Phase 1		
$thread_{0,0}$	$\mathbf{M_{0,0}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{0,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}*Nds_{0,0}$ + $Mds_{0,1}*Nds_{1,0}$	$\mathbf{M_{0,2}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{2,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}*Nds_{0,0}$ + $Mds_{0,1}*Nds_{1,0}$
$thread_{0,1}$	$\mathbf{M_{0,1}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{0,1}}$ ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}*Nds_{0,1}$ + $Mds_{0,1}*Nds_{1,1}$	$\mathbf{M_{0,3}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{2,1}}$ ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}*Nds_{0,1}$ + $Mds_{0,1}*Nds_{1,1}$
$thread_{1,0}$	$\mathbf{M_{1,0}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{1,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}*Nds_{0,0}$ + $Mds_{1,1}*Nds_{1,0}$	$\mathbf{M_{1,2}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{3,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}*Nds_{0,0}$ + $Mds_{1,1}*Nds_{1,0}$
$thread_{1,1}$	$\mathbf{M_{1,1}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{1,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}*Nds_{0,1}$ + $Mds_{1,1}*Nds_{1,1}$	$\mathbf{M_{1,3}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{3,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}*Nds_{0,1}$ + $Mds_{1,1}*Nds_{1,1}$

time →

FIGURE 5.8

Execution phases of a tiled matrix multiplication.

```
#define TILE_WIDTH 16
__global__ void matrixMulKernel(float* M, float* N, float* P, int Width) {

    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

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

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

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

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

        for (int k = 0; k < TILE_WIDTH; ++k) {
            Pvalue += Mds[ty][k] * Nds[k][tx];
        }
        __syncthreads();

    }
    P[Row*Width + Col] = Pvalue;

}
```

FIGURE 5.9

A tiled matrix multiplication kernel using shared memory.

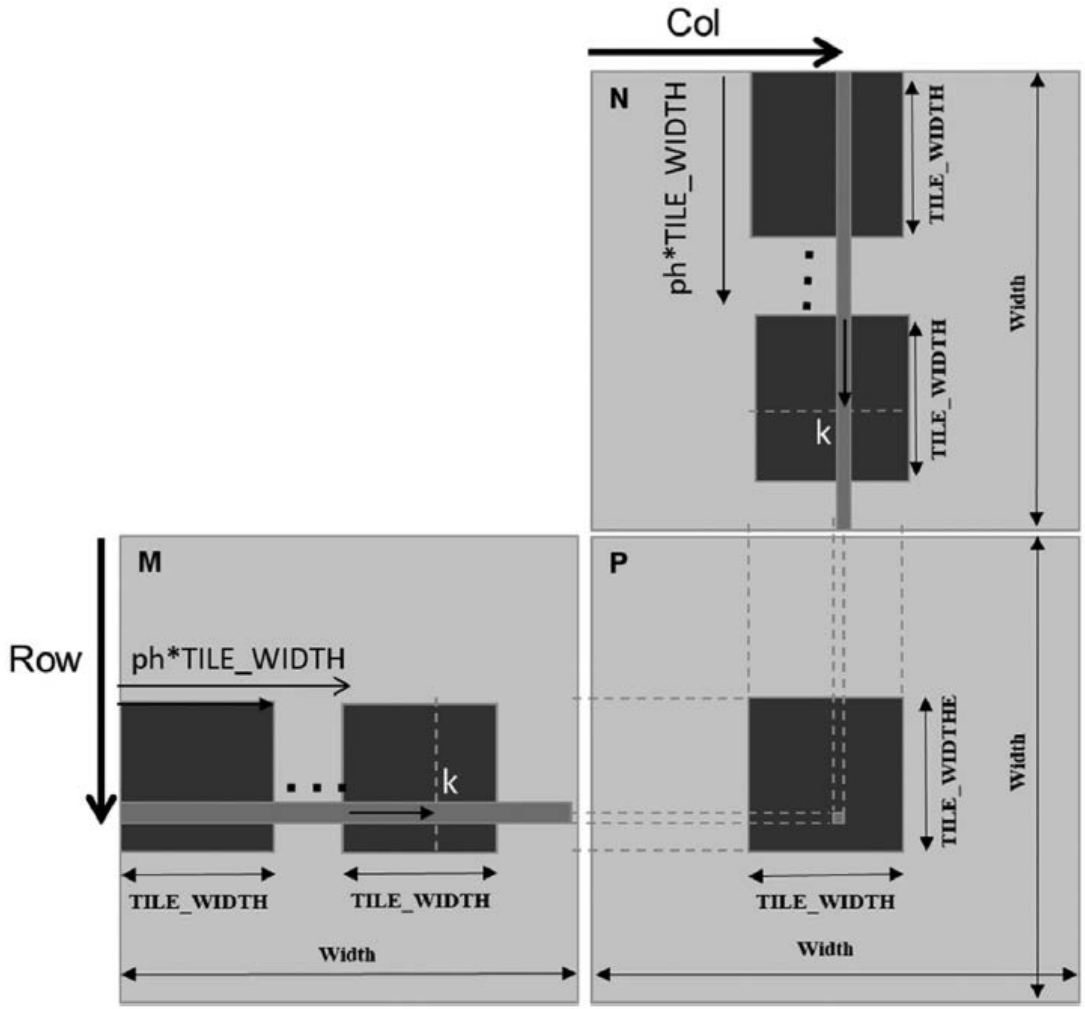


FIGURE 5.10

Calculation of the matrix indices in tiled multiplication.

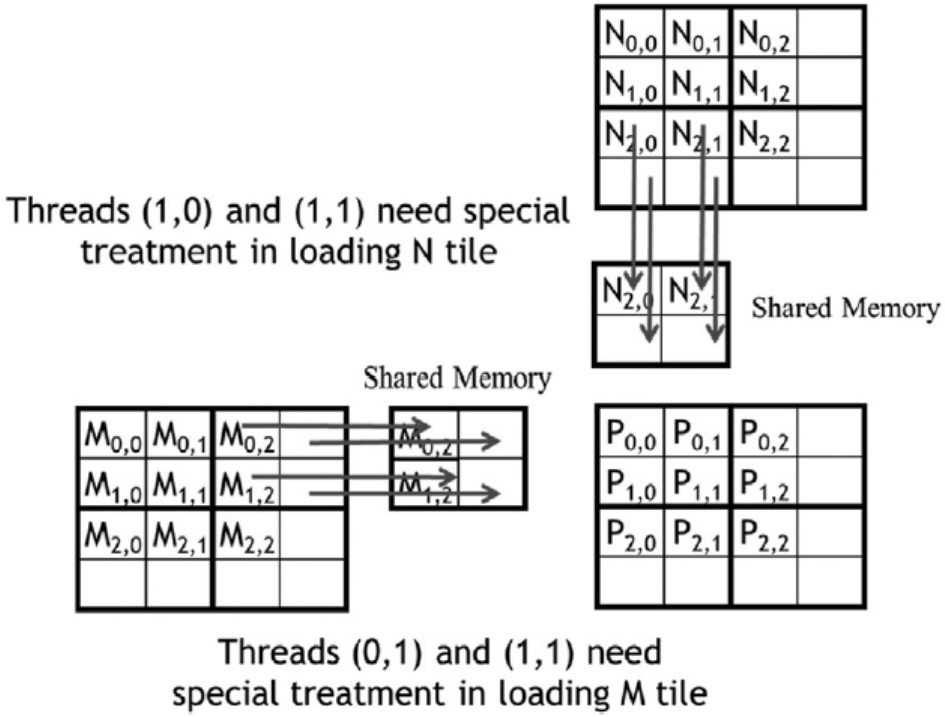


FIGURE 5.11

Loading input matrix elements that are close to the edge: phase 1 of $block_{0,0}$.

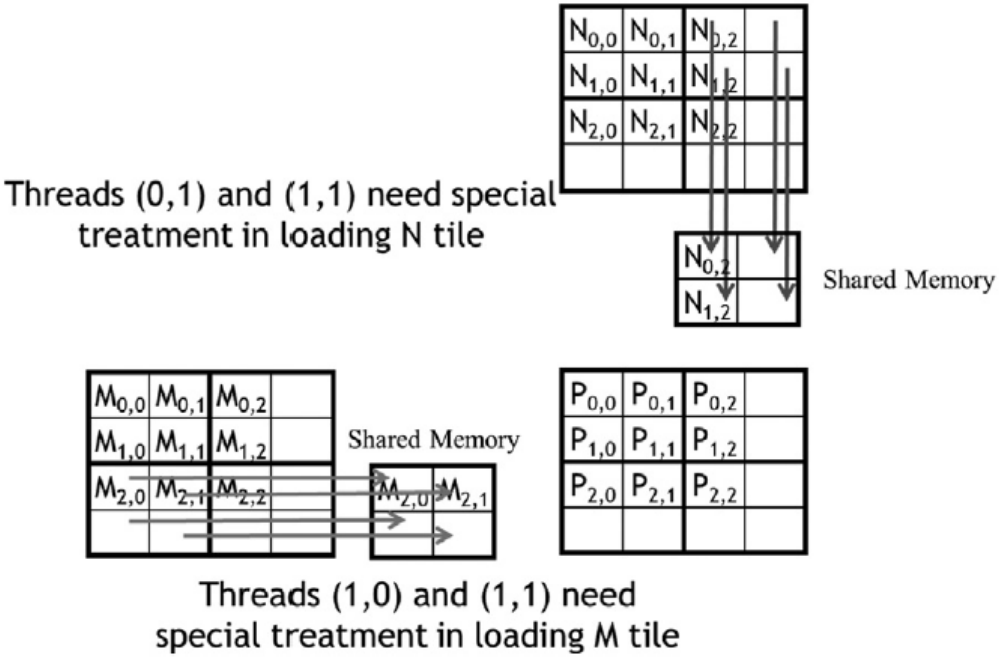


FIGURE 5.12

Loading input elements during phase 0 of $block_{1,1}$.

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

        // Collaborative loading of M and N tiles into shared memory
        if ((Row < Width) && (ph*TILE_WIDTH+tx) < Width)
            Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
        else Mds[ty][tx] = 0.0f;
        if ((ph*TILE_WIDTH+ty) < Width && Col < Width)
            Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
        else Nds[ty][tx] = 0.0f;
        __syncthreads();

        for (int k = 0; k < TILE_WIDTH; ++k) {
            Pvalue += Mds[ty][k] * Nds[k][tx];
        }
        __syncthreads();

    }
    if (Row < Width) && (Col < Width)
        P[Row*Width + Col] = Pvalue;
```

FIGURE 5.13

Tiled matrix multiplication kernel with boundary condition checks.

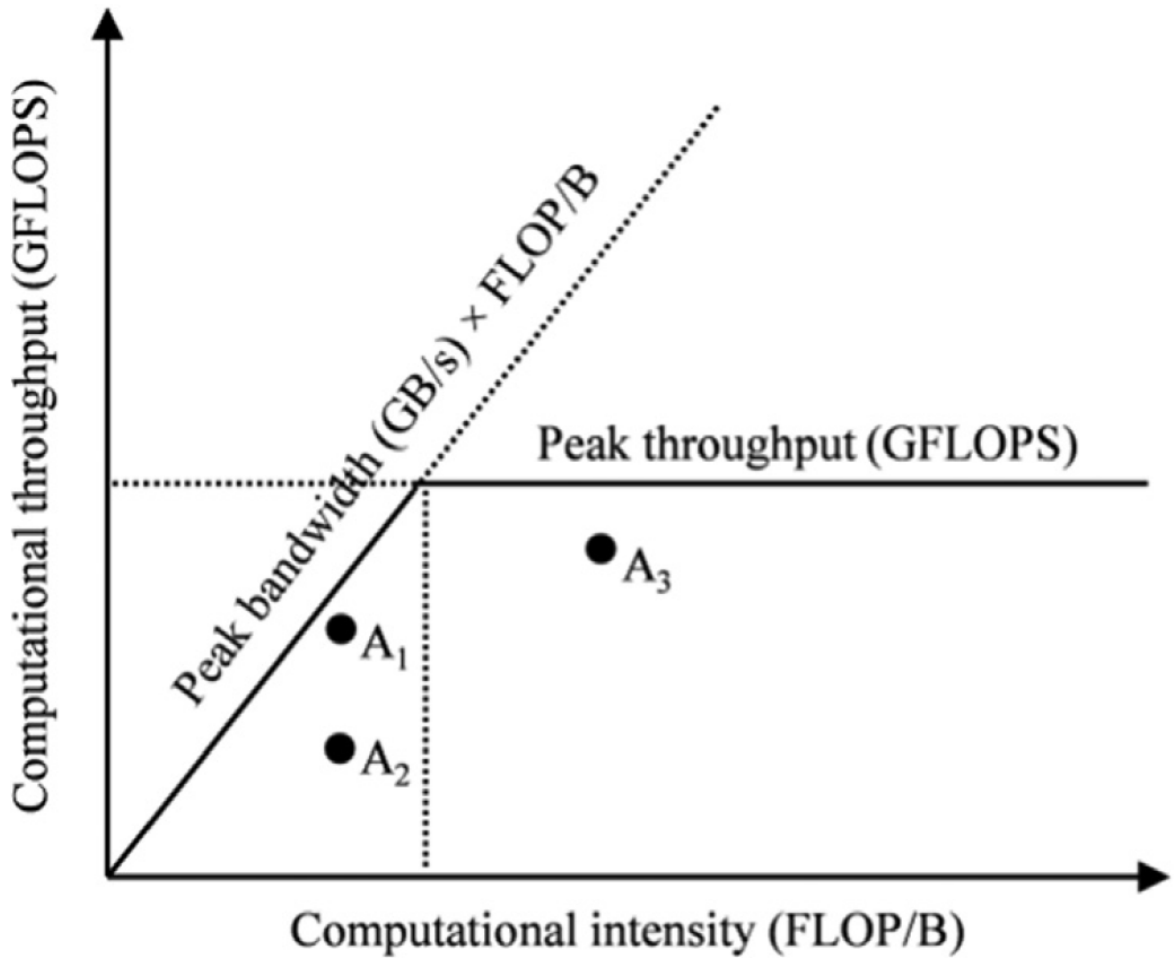
```
#define TILE_WIDTH 16
__global__ void matrixMulKernel(float* M, float* N, float* P, int Width,
                                unsigned Mdz_sz, unsigned Nds_sz) {

   extern __shared__ char float Mds_Nds[];

   float *Mds = (float *) Mds_Nds;
   float *Nds = (float *) Mds_Nds + Mds_sz;
```

FIGURE 5.14

Tiled matrix multiplication kernel with dynamically sized shared memory usage.



In-text figure 1

```
fadd r1, r2, r3
```

In-text figure 2

```
load r2, r4, offset
fadd r1, r2, r3
```

In-text figure 3

```
__shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
__shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
```

In-text figure 4

```
#define TILE_WIDTH 16
```

In-text figure 5

```
extern __shared__ Mds_Nds[];
```

In-text figure 6

```
  size_t size =
calculate_appropriate_SM_usage(devProp.sharedMemPerBlock,
...);

  matrixMulKernel<<<dimGrid,dimBlock,size>>>(Md, Nd, Pd,
Width, size/2, size/2);
```

In-text figure 7

```
dim3 blockDim(BLOCK_WIDTH,BLOCK_WIDTH);
dim3 gridDim(A_width/blockDim.x,A_height/blockDim.y);
BlockTranspose<<<gridDim, blockDim>>>(A, A_width, A_height);

__global__ void
BlockTranspose(float* A_elements, int A_width, int A_height)
{
   __shared__ float blockA[BLOCK_WIDTH][BLOCK_WIDTH];

   int baseIdx = blockIdx.x * BLOCK_SIZE + threadIdx.x;
   baseIdx += (blockIdx.y * BLOCK_SIZE + threadIdx.y) * A_width;

   blockA[threadIdx.y][threadIdx.x] = A_elements[baseIdx];

   A_elements[baseIdx] = blockA[threadIdx.x][threadIdx.y];
}
```

In-text figure 8

```
__global__ void foo_kernel(float* a, float* b) {
   unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
   float x[4];
   __shared__ float y_s;
   __shared__ float b_s[128];
   for(unsigned int j = 0; j < 4; ++j) {
      x[j] = a[j*blockDim.x*gridDim.x + i];
   }
   if(threadIdx.x == 0) {
      y_s = 7.4f;
   }
   b_s[threadIdx.x] = b[i];
   __syncthreads();
   b[i] = 2.5f*x[0] + 3.7f*x[1] + 6.3f*x[2] + 8.5f*x[3]
          + y_s*b_s[threadIdx.x] + b_s[(threadIdx.x + 3)%128];
}
void foo(int* a_d, int* b_d) {
   unsigned int N = 1024;
   foo_kernel <<< (N + 128 - 1)/128, 128 >>>(a_d, b_d);
}
```

In-text figure 9

Table 5.1 CUDA variable declaration type qualifiers and the properties of each type.

Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Grid
Automatic array variables	Local	Thread	Grid
__device__ __shared__ int SharedVar;	Shared	Block	Grid
__device__ int GlobalVar;	Global	Grid	Application
__device__ __constant__ int ConstVar;	Constant	Grid	Application