CSCI 5451: Introduction to Parallel Computing

Lecture 24: Convolutions in Cuda

computer science
& engineering

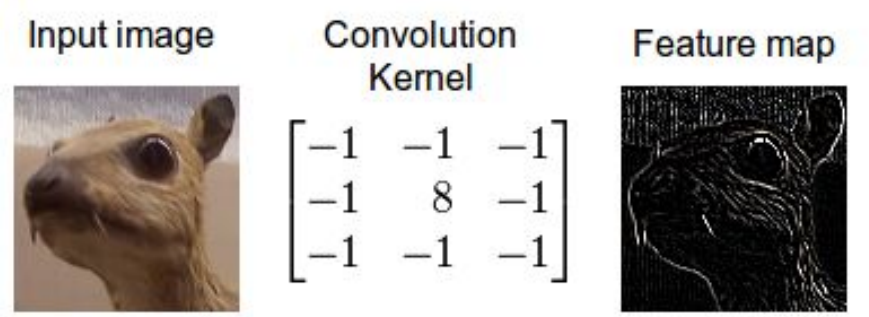
University of Minnesota
Driven to Discover®

Announcements (12/01)

- Project responses given - be sure to check your group slack to ensure that you see expectations
- HW3 Due Yesterday
- HW4 Released (Due Dec 7)
  - Profiling a convolutional kernel
  - Done in Colab
- HW5 Released (Due Dec 18)
  - Group Assignment
  - Batch GEMM algorithm in CUDA

Convolutions

- Convolutional filters are arrays (1-d), matrices (2-d) and higher dimensional tensors (3-d) applied to input data
- These filters are also called kernels (we will use filters in later slides to avoid the confusion with cuda kernels)
- Have different uses
  - 1-D (Audio)
  - 2-D (Images)
  - 3-D (Video)



Convolutions in 1-D

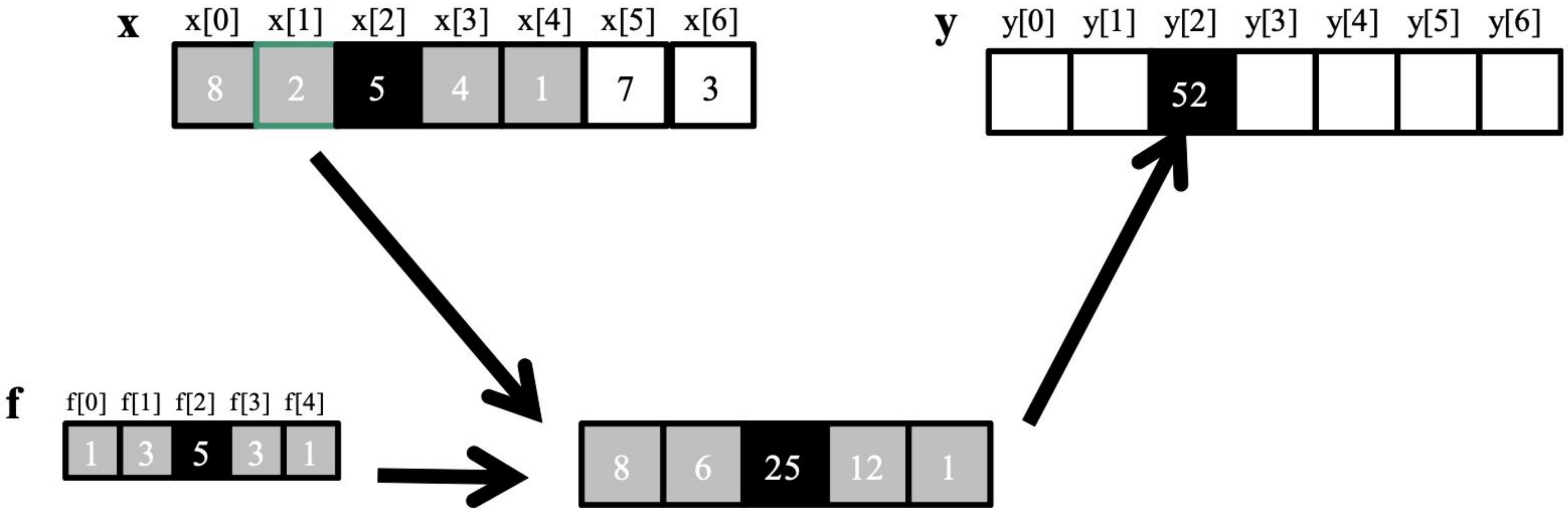
- Consider input data x of length n, filter f of length $2r+1$
- r is often considered the *radius* of the convolution filter
- The output is some vector of data y

$$[x_0,\ x_1,\ \ldots,\ x_{n-1}]$$

$$[f_0, f_1, \ldots, f_{2r}]$$

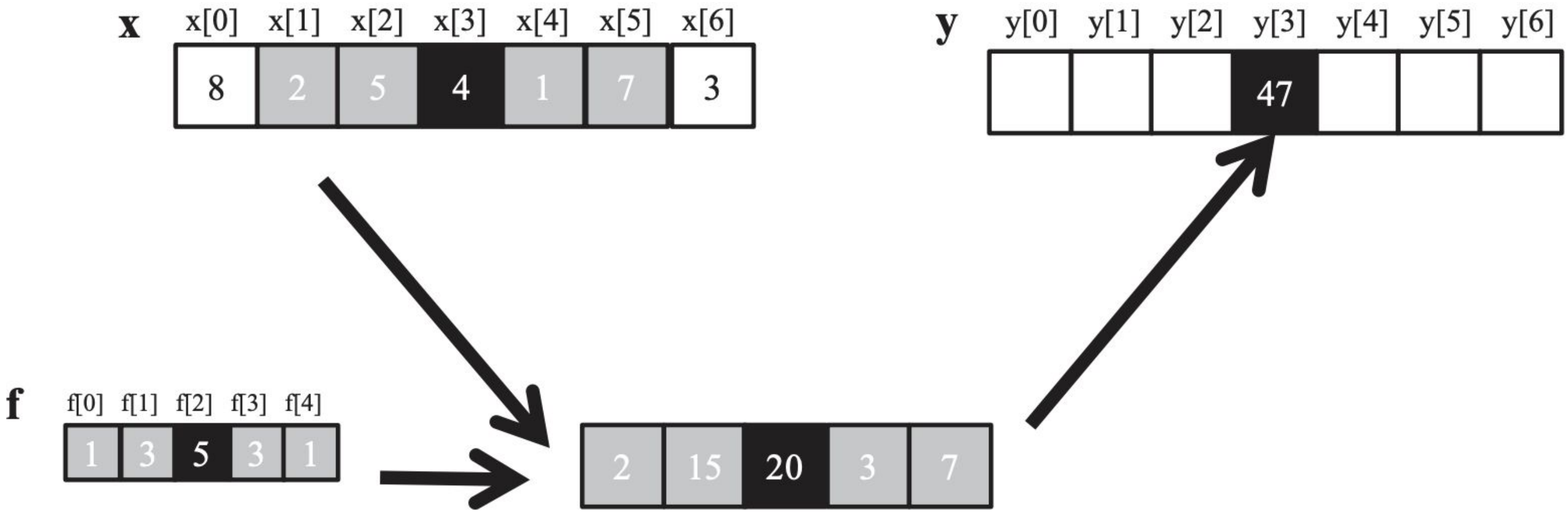
$$y_i = \sum_{j=-r}^{r} f_{i+j} \times x_i$$

1-D examples



x
x[0] x[1] x[2] x[3] x[4] x[5] x[6]
8 2 5 4 1 7 3
y
y[0] y[1] y[2] y[3] y[4] y[5] y[6]
52
f
f[0] f[1] f[2] f[3] f[4]
1 3 5 3 1
8 6 25 12 1

1-D examples



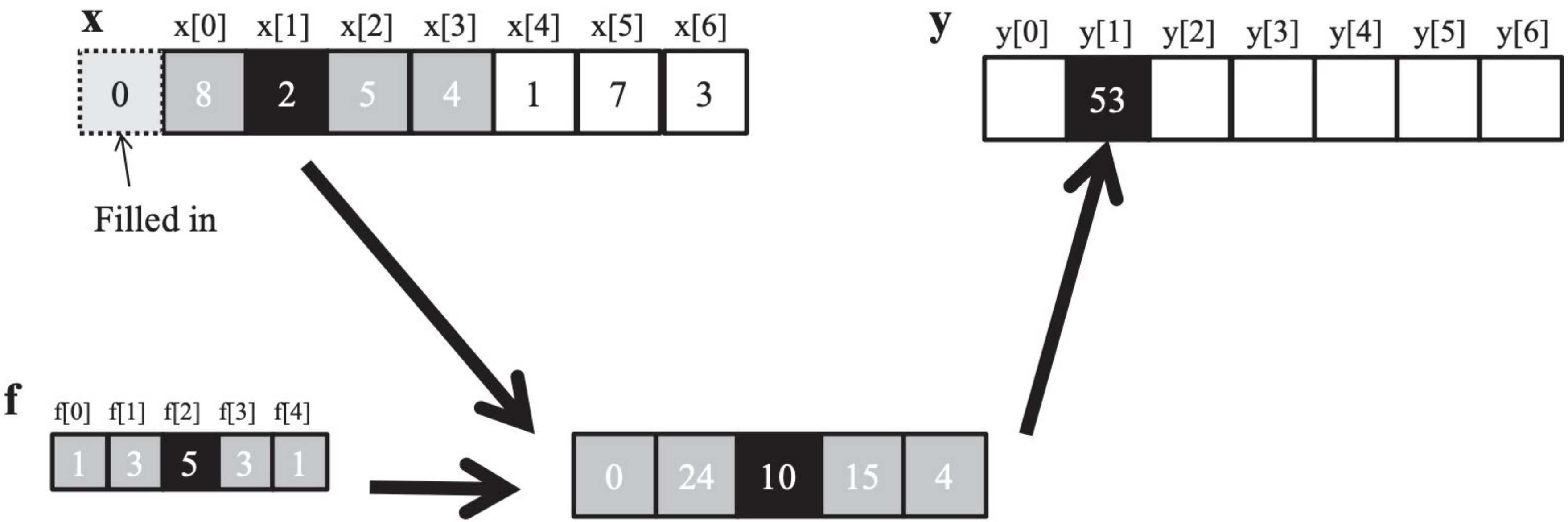
x
x[0] x[1] x[2] x[3] x[4] x[5] x[6]
8 2 5 4 1 7 3
y
y[0] y[1] y[2] y[3] y[4] y[5] y[6]
47
f
f[0] f[1] f[2] f[3] f[4]
1 3 5 3 1
2 15 20 3 7

How do we handle the data on the edge?

How do we handle the data on the edge?

Pad the inputs with zeros.

1-D examples


x
x[0]
x[1]
x[2]
x[3]
x[4]
x[5]
x[6]
0
8
2
5
4
1
7
3
Filled in
y
y[0]
y[1]
y[2]
y[3]
y[4]
y[5]
y[6]
53
f
f[0]
f[1]
f[2]
f[3]
f[4]
1
3
5
3
1
0
24
10
15
4

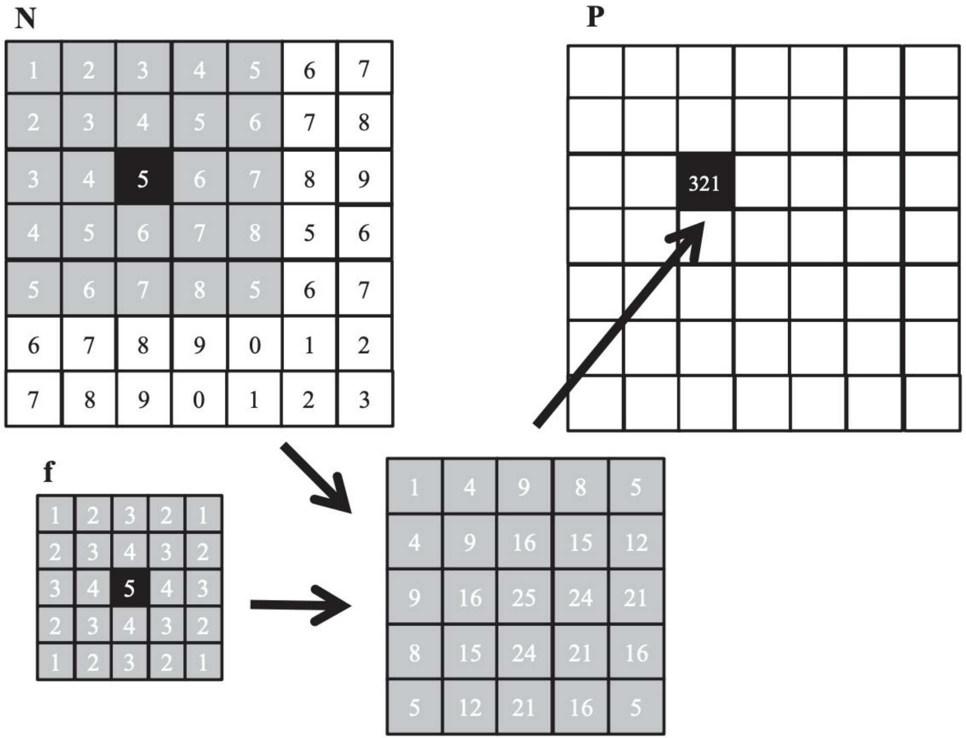
2-D Convolution

-

$$P_{y,x} = \sum_{j=-r_y}^{r_y} \sum_{k=-r_x}^{r_x} f_{y+j,x+k} \times N_{y,x}$$

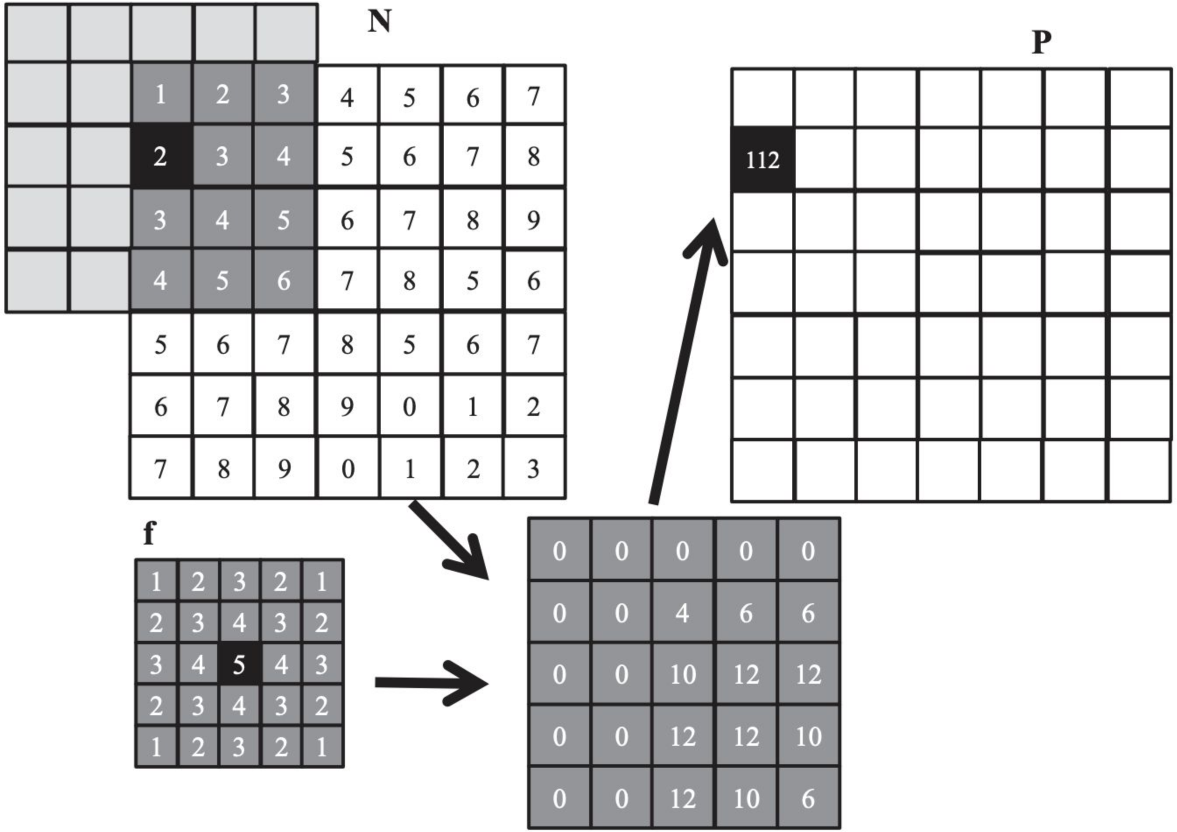
2-D Convolution

```
P2,2 = N0,0*M0,0 + N0,1*M0,1 + N0,2*M0,2 + N0,3*M0,3 + N0,4*M0,4
     + N1,0*M1,0 + N1,1*M1,1 + N1,2*M1,2 + N1,3*M1,3 + N1,4*M1,4
     + N2,0*M2,0 + N2,1*M1,1 + N2,2*M2,2 + N2,3*M2,3 + N2,4*M2,4
     + N3,0*M3,0 + N3,1*M3,1 + N3,2*M3,2 + N3,3*M3,3 + N3,4*M3,4
     + N4,0*M4,0 + N4,1*M4,1 + N4,2*M4,2 + N4,3*M4,3 + N4,4*M4,4
   = 1*1 + 2*2 + 3*3 + 4*2 + 5*1
     + 2*2 + 3*3 + 4*4 + 5*3 + 6*2
     + 3*3 + 4*4 + 5*5 + 6*4 + 7*3
     + 4*2 + 5*3 + 6*4 + 7*3 + 8*2
     + 5*1 + 6*2 + 7*3 + 8*2 + 5*1
   = 1 + 4 + 9 + 8 + 5
     + 4 + 9 + 16 + 15 + 12
     + 9 + 16 + 25 + 24 + 21
     + 8 + 15 + 24 + 21 + 16
     + 5 + 12 + 21 + 16 + 5
   = 321
```



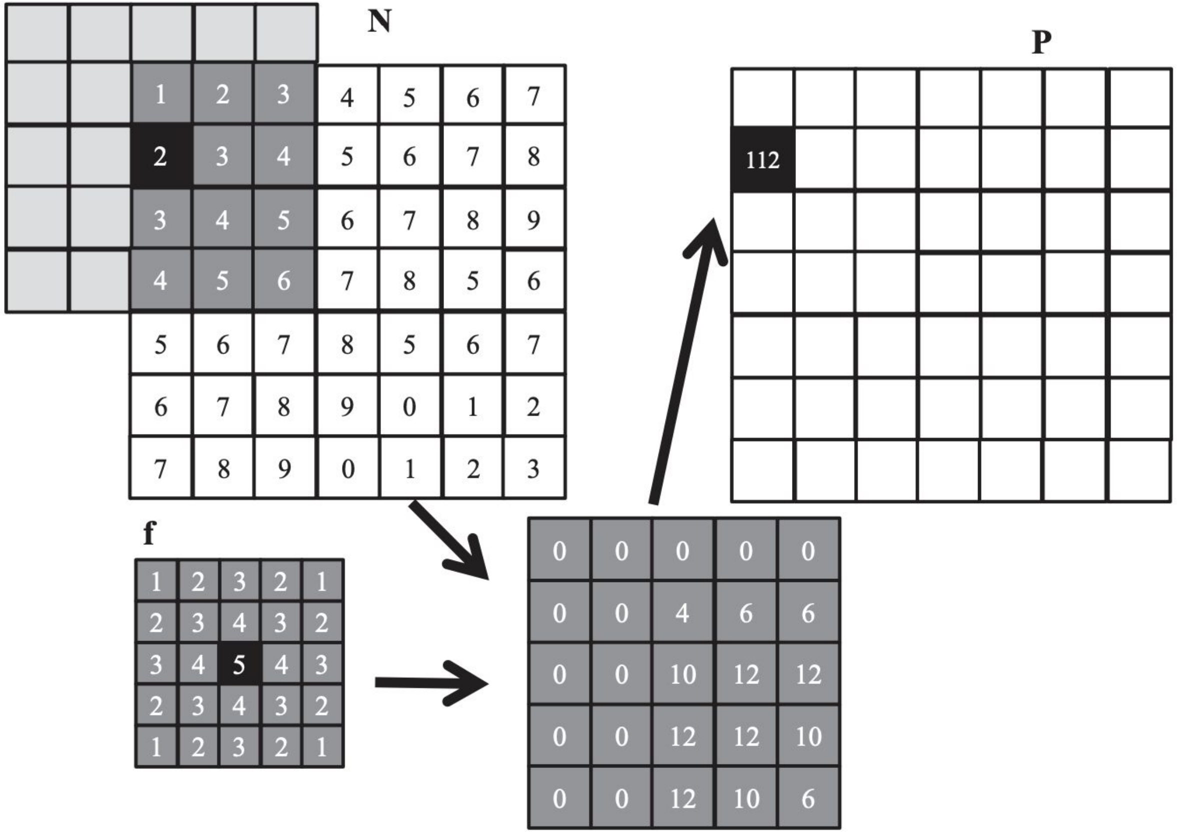
2-D Convolution

Similar to the 1-d case, we pad with zeros for the edge cases



2-D Convolution

How should we parallelize this?



2-D Kernel without Constant Memory

```
__global__ void convolution_2D_basic_kernel(float *N, float *F, float *P,
    int r, int width, int height) {
    int outCol = blockIdx.x*blockDim.x + threadIdx.x;
    int outRow = blockIdx.y*blockDim.y + threadIdx.y;
    float Pvalue = 0.0f;
    for (int fRow = 0; fRow < 2*r+1; fRow++) {
      for (int fCol = 0; fCol < 2*r+1; fCol++) {
          inRow = outRow - r + fRow;
          inCol = outCol - r + fCol;
          if (inRow >= 0 && inRow < height && inCol >= 0 && inCol < width) {
             Pvalue += F[fRow][fCol]*N[inRow*width + inCol];
          }
      }
    }
    P[outRow][outCol] = Pvalue;
}
```

2-D Kernel without Constant Memory

Issues with this approach?

```
__global__ void convolution_2D_basic_kernel(float *N, float *F, float *P,
    int r, int width, int height) {
    int outCol = blockIdx.x*blockDim.x + threadIdx.x;
    int outRow = blockIdx.y*blockDim.y + threadIdx.y;
    float Pvalue = 0.0f;
    for (int fRow = 0; fRow < 2*r+1; fRow++) {
      for (int fCol = 0; fCol < 2*r+1; fCol++) {
        inRow = outRow - r + fRow;
        inCol = outCol - r + fCol;
        if (inRow >= 0 && inRow < height && inCol >= 0 && inCol < width) {
           Pvalue += F[fRow][fCol]*N[inRow*width + inCol];
        }
      }
    }
    P[outRow][outCol] = Pvalue;
}
```

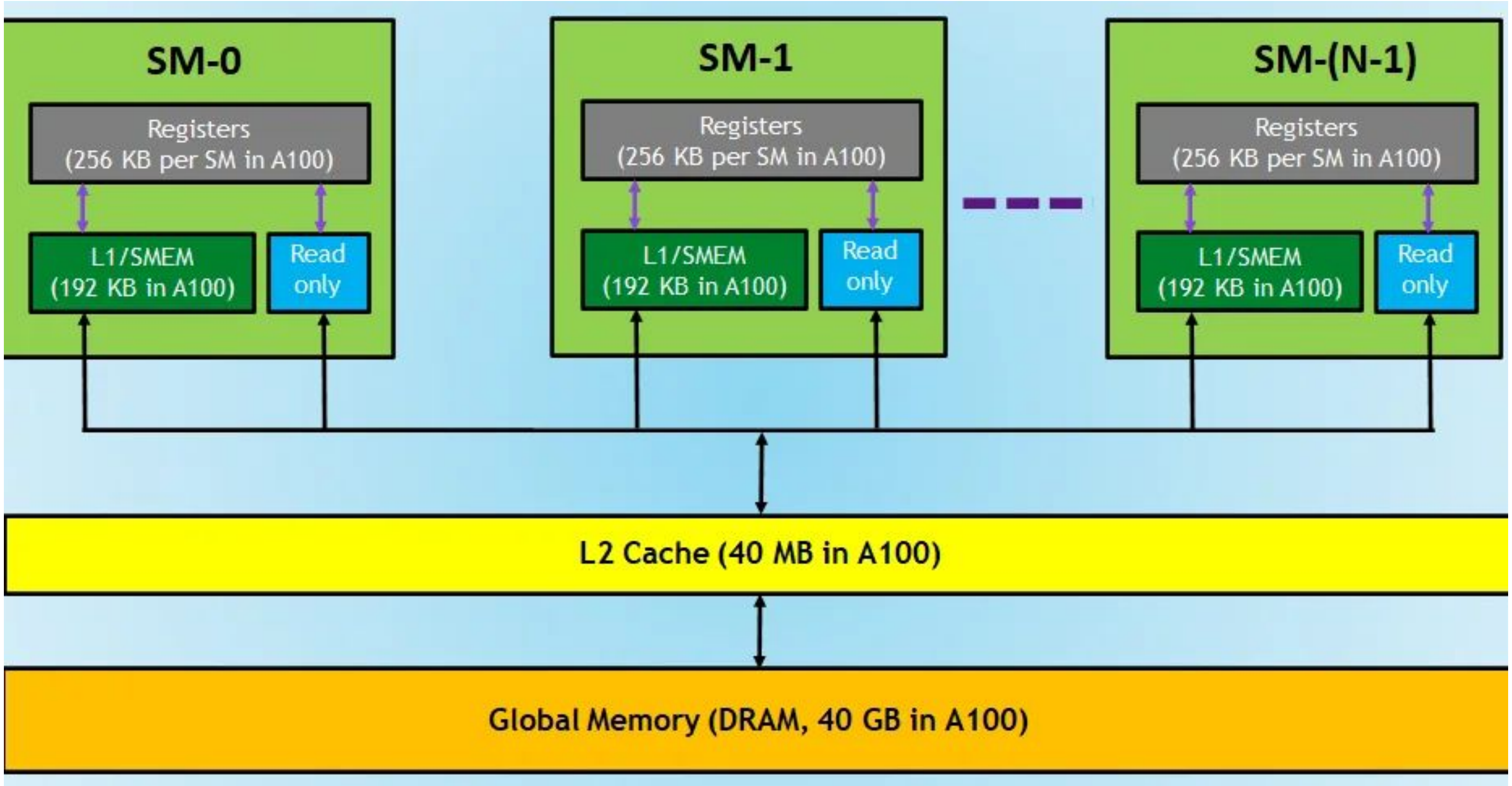
2-D Kernel without Constant Memory

Issues with this approach?

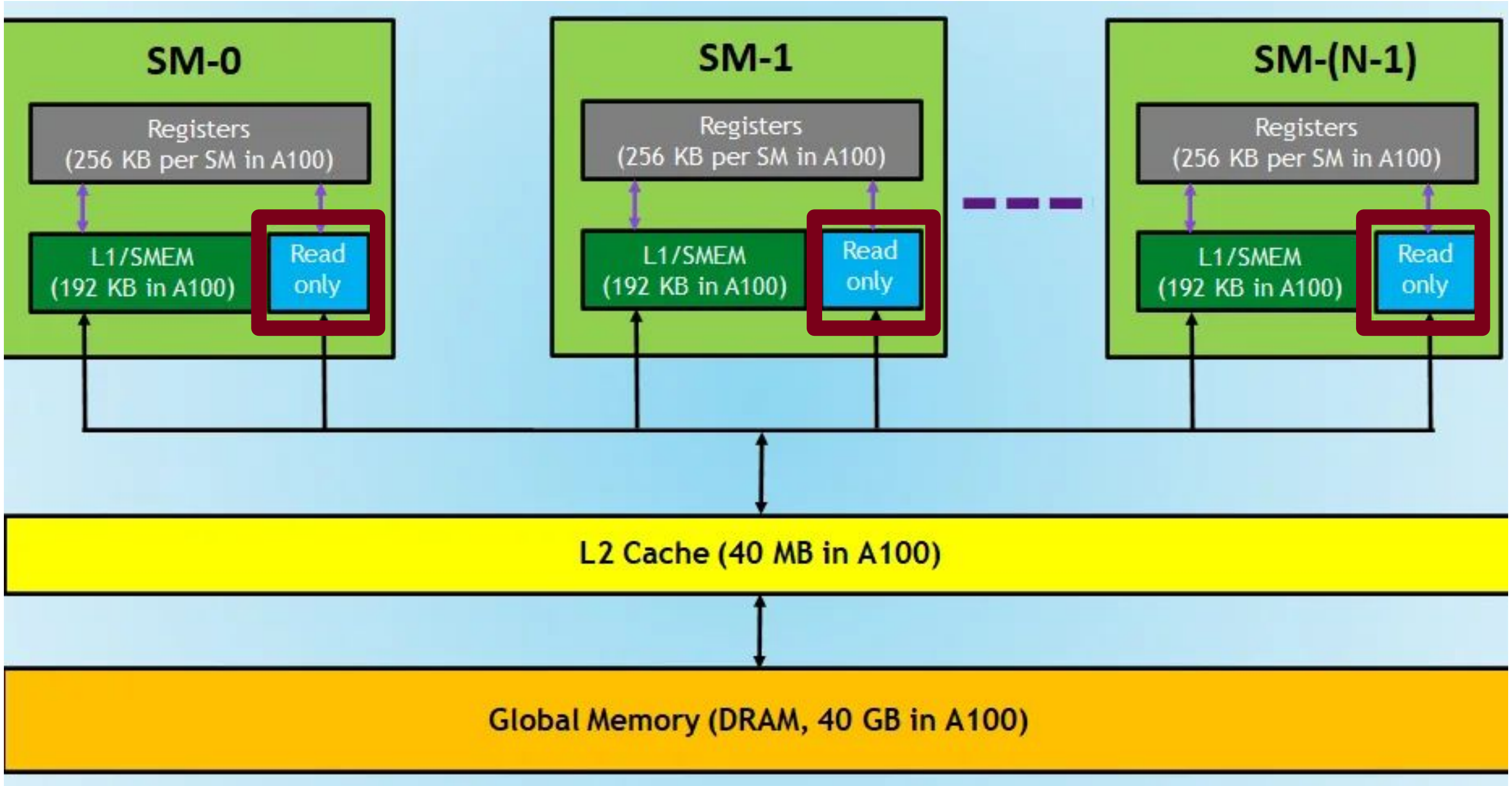
Divergence, no constant memory, potentially low arithmetic intensity

```
__global__ void convolution_2D_basic_kernel(float *N, float *F, float *P,
    int r, int width, int height) {
    int outCol = blockIdx.x*blockDim.x + threadIdx.x;
    int outRow = blockIdx.y*blockDim.y + threadIdx.y;
    float Pvalue = 0.0f;
    for (int fRow = 0; fRow < 2*r+1; fRow++) {
      for (int fCol = 0; fCol < 2*r+1; fCol++) {
        inRow = outRow - r + fRow;
        inCol = outCol - r + fCol;
        if (inRow >= 0 && inRow < height && inCol >= 0 && inCol < width) {
          Pvalue += F[fRow][fCol]*N[inRow*width + inCol];
        }
      }
    }
    P[outRow][outCol] = Pvalue;
}
```

Revisiting Constant Memory



Revisiting Constant Memory



Revisiting Constant Memory

- Constant memory is a read-only memory on each SM
- Allows for reduced logic in hardware as coherence is not necessary
- Access is faster than global and shared across all threads
- Useful for convolutions, where the filter is constant in execution

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

2-D Kernel with Constant Memory

```
#define FILTER_RADIUS 2
__constant__ float F[2*FILTER_RADIUS+1][2*FILTER_RADIUS+1];
```

```
__global__ void convolution_2D_const_mem_kernel(float *N, float *P, int r,
    int width, int height) {
   int outCol = blockIdx.x*blockDim.x + threadIdx.x;
   int outRow = blockIdx.y*blockDim.y + threadIdx.y;
   float Pvalue = 0.0f;
   for (int fRow = 0; fRow < 2*r+1; fRow++) {
     for (int fCol = 0; fCol < 2*r+1; fCol++) {
        inRow = outRow - r + fRow;
        inCol = outCol - r + fCol;
        if (inRow >= 0 && inRow < height && inCol >= 0 && inCol < width) {
           Pvalue += F[fRow][fCol]*N[inRow*width + inCol];
        }
      }
   }
   P[outRow*width+outCol] = Pvalue;
}
```

```
cudaMemcpyToSymbol(F,F_h,(2*FILTER_RADIUS+1)*(2*FILTER_RADIUS+1)*sizeof(float));
```

2-D Kernel with Constant Memory

```
#define FILTER_RADIUS 2
__constant__ float F[2*FILTER_RADIUS+1][2*FILTER_RADIUS+1];
```

```
__global__ void convolution_2D_const_mem_kernel(float *N, float *P, int r,
    int width, int height) {
   int outCol = blockIdx.x*blockDim.x + threadIdx.x;
   int outRow = blockIdx.y*blockDim.y + threadIdx.y;
   float Pvalue = 0.0f;
   for (int fRow = 0; fRow < 2*r+1; fRow++) {
     for (int fCol = 0; fCol < 2*r+1; fCol++) {
        inRow = outRow - r + fRow;
        inCol = outCol - r + fCol;
        if (inRow >= 0 && inRow < height && inCol >= 0 && inCol < width) {
           Pvalue += F[fRow][fCol]*N[inRow*width + inCol];
        }
     }
   }
   P[outRow*width+outCol] = Pvalue;
}
```

Same kernel as before - now using constant *F*

```
cudaMemcpyToSymbol(F,F_h,(2*FILTER_RADIUS+1)*(2*FILTER_RADIUS+1)*sizeof(float));
```

2-D Kernel with Constant Memory

```
#define FILTER_RADIUS 2
__constant__ float F[2*FILTER_RADIUS+1][2*FILTER_RADIUS+1];
```

```
__global__ void convolution_2D_const_mem_kernel(float *N, float *P, int r,
    int width, int height) {
   int outCol = blockIdx.x*blockDim.x + threadIdx.x;
   int outRow = blockIdx.y*blockDim.y + threadIdx.y;
   float Pvalue = 0.0f;
   for (int fRow = 0; fRow < 2*r+1; fRow++) {
     for (int fCol = 0; fCol < 2*r+1; fCol++) {
         inRow = outRow - r + fRow;
         inCol = outCol - r + fCol;
         if (inRow >= 0 && inRow < height && inCol >= 0 && inCol < width) {
            Pvalue += F[fRow][fCol]*N[inRow*width + inCol];
         }
      }
   }
   P[outRow*width+outCol] = Pvalue;
}
```

Same kernel as before - now using constant *F*

This line inside of *main*

```
cudaMemcpyToSymbol(F,F_h,(2*FILTER_RADIUS+1)*(2*FILTER_RADIUS+1)*sizeof(float));
```

2-D Kernel with Constant Memory

How is memory loading from *N* working here?

```
#define FILTER_RADIUS 2
__constant__ float F[2*FILTER_RADIUS+1][2*FILTER_RADIUS+1];
```

```
__global__ void convolution_2D_const_mem_kernel(float *N, float *P, int r,
    int width, int height) {
   int outCol = blockIdx.x*blockDim.x + threadIdx.x;
   int outRow = blockIdx.y*blockDim.y + threadIdx.y;
   float Pvalue = 0.0f;
   for (int fRow = 0; fRow < 2*r+1; fRow++) {
     for (int fCol = 0; fCol < 2*r+1; fCol++) {
        inRow = outRow - r + fRow;
        inCol = outCol - r + fCol;
        if (inRow >= 0 && inRow < height && inCol >= 0 && inCol < width) {
           Pvalue += F[fRow][fCol]*N[inRow*width + inCol];
        }
     }
   }
   P[outRow*width+outCol] = Pvalue;
}
```

Same kernel as before - now using constant *F*

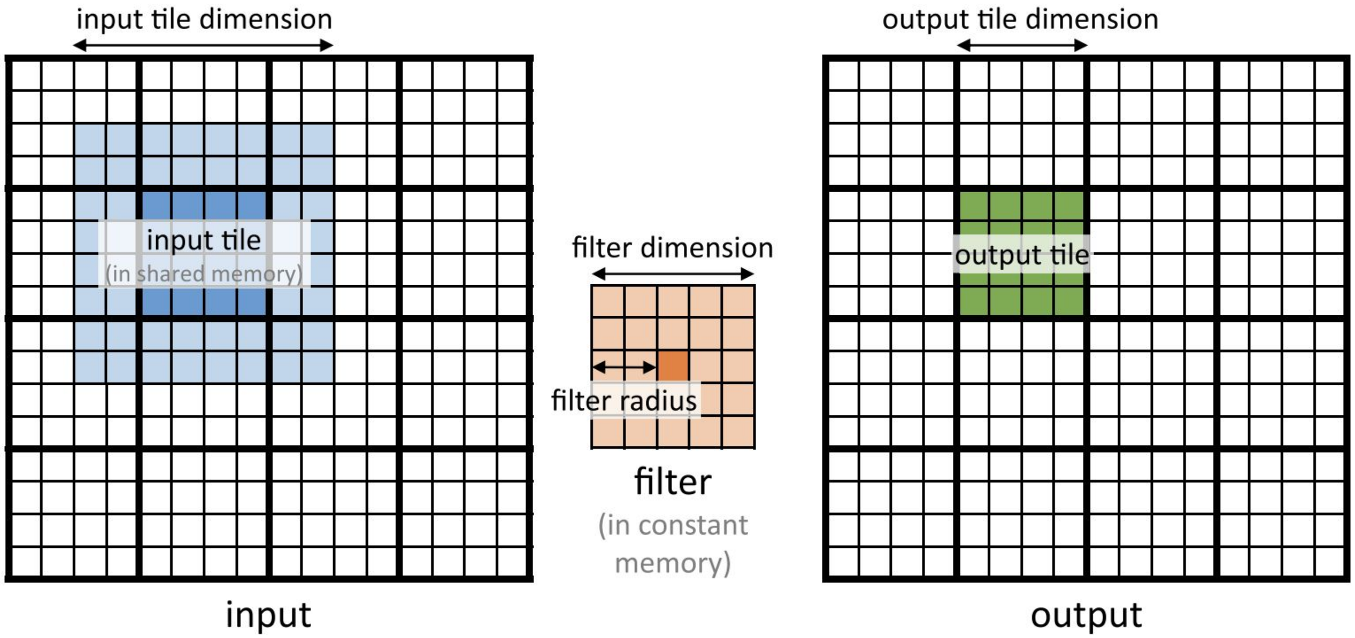
This line inside of *main*

```
cudaMemcpyToSymbol(F,F_h,(2*FILTER_RADIUS+1)*(2*FILTER_RADIUS+1)*sizeof(float));
```

How else can we implement this?

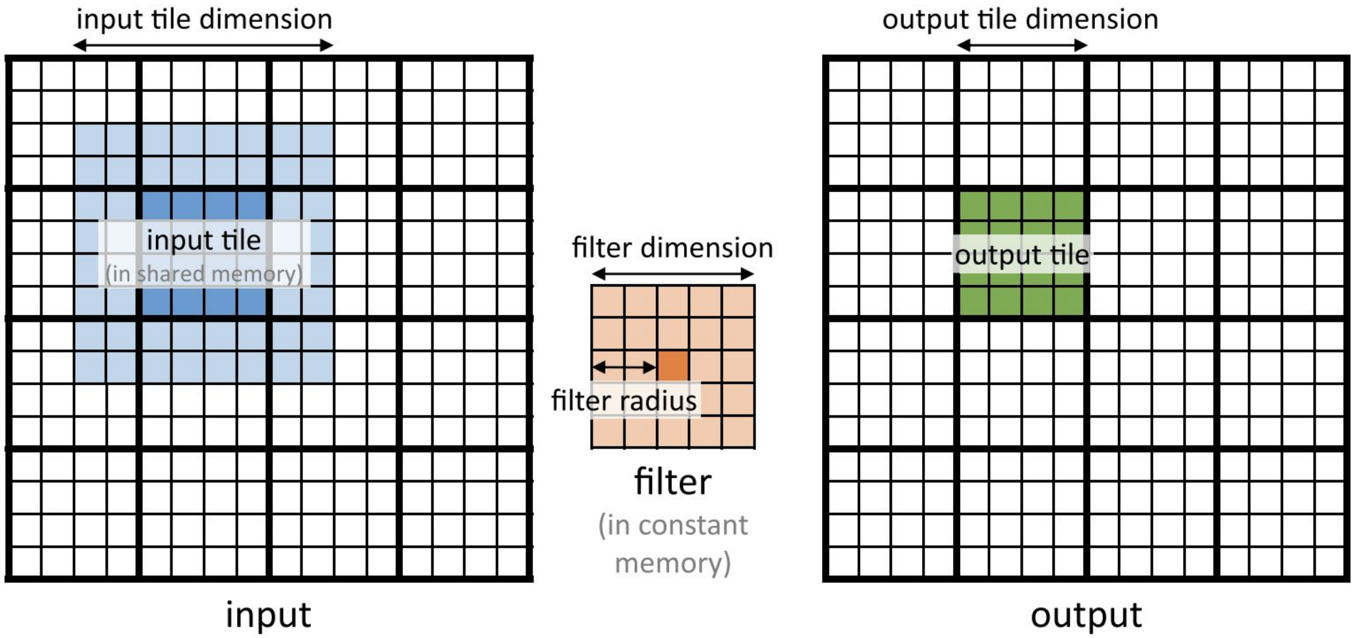
Shared Memory Tiling

Tiled Kernel



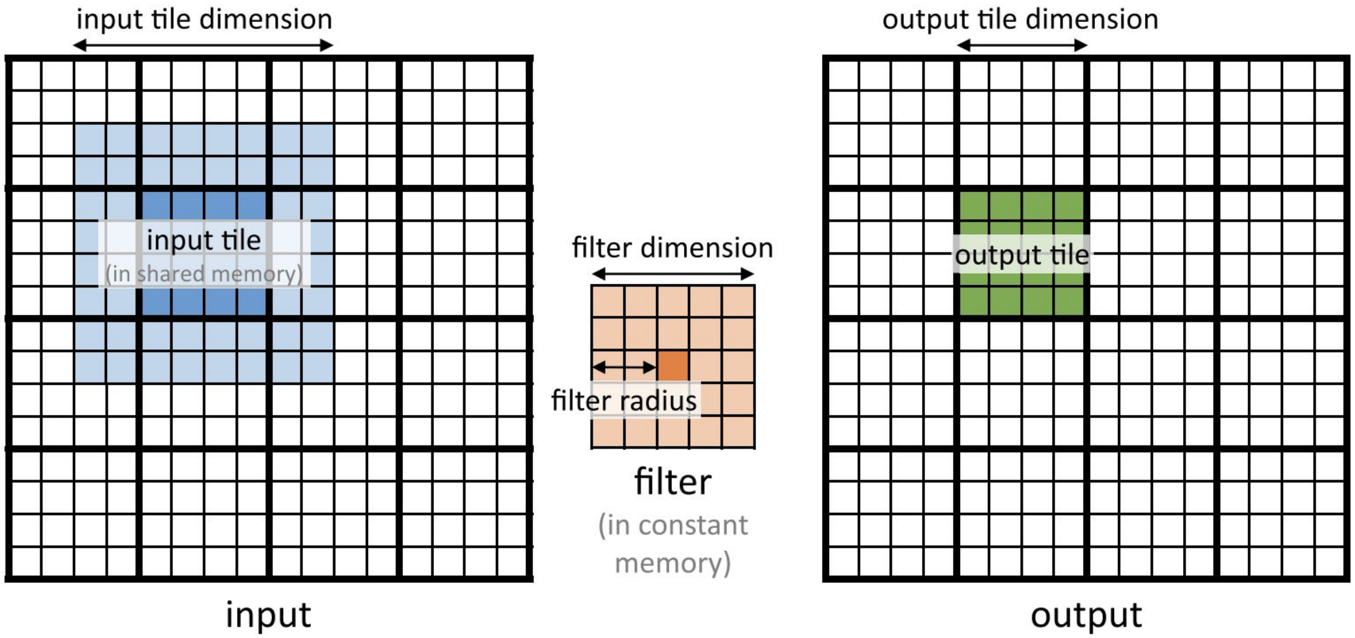
Tiled Kernel

Assume we launch 4x4 threadblocks. (What is the problem with this size of threadblock?)



Tiled Kernel

We have to choose either to tile by input or by output. In other words, either every thread loads and only some compute (**input**) or every thread computes and loads more than once (**output**)



Tiled Kernel

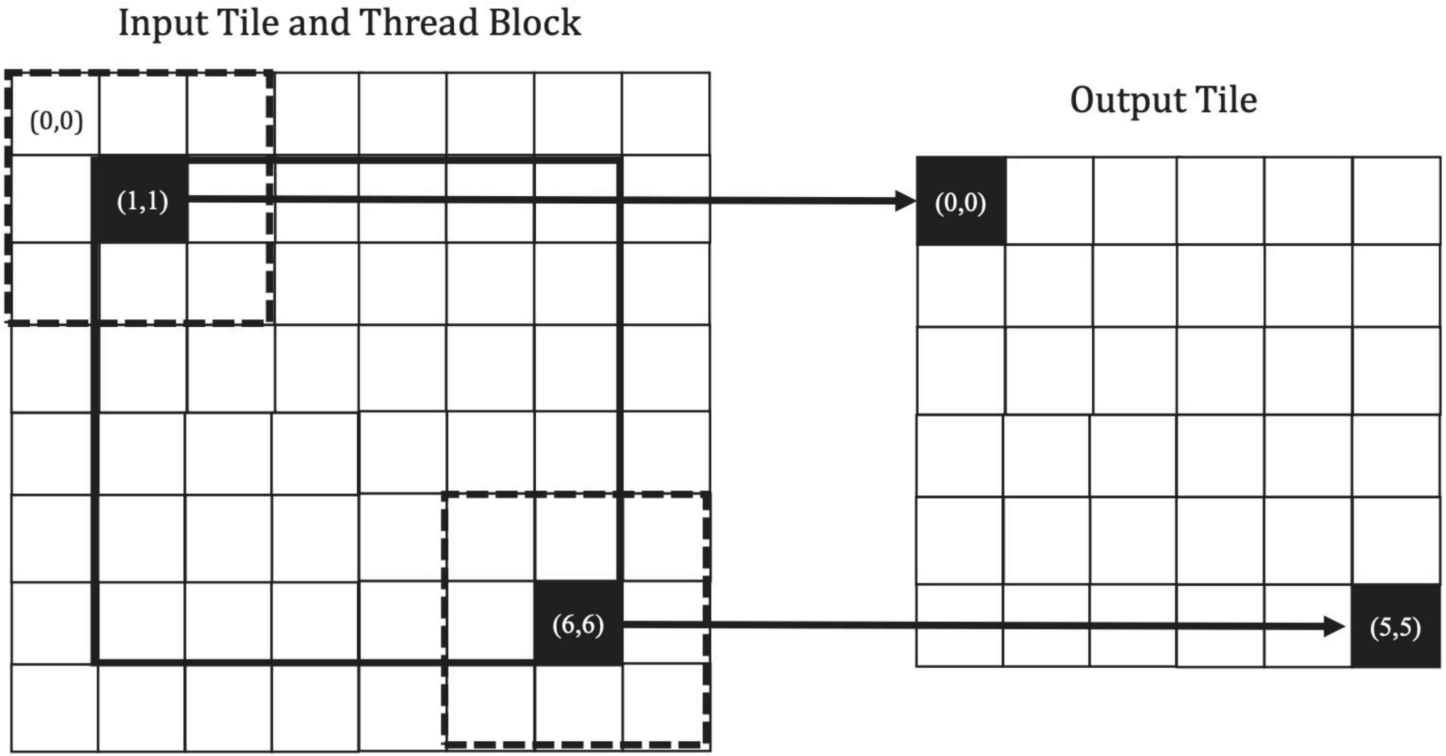
```
#define IN_TILE_DIM 32
#define OUT_TILE_DIM ((IN_TILE_DIM) - 2*(FILTER_RADIUS))
__constant__ float F_c[2*FILTER_RADIUS+1][2*FILTER_RADIUS+1];
__global__ void convolution_tiled_2D_const_mem_kernel(float *N, float *P,
                                                      int width, int height) {
  int col = blockIdx.x*OUT_TILE_DIM + threadIdx.x - FILTER_RADIUS;
  int row = blockIdx.y*OUT_TILE_DIM + threadIdx.y - FILTER_RADIUS;
  //loading input tile
  __shared__ N_s[IN_TILE_DIM][IN_TILE_DIM];
  if(row>=0 && row<height && col>=0 && col<width) {
    N_s[threadIdx.y][threadIdx.x] = N[row*width + col];
  } else {
    N_s[threadIdx.y][threadIdx.x] = 0.0;
  }
  __syncthreads();
  // Calculating output elements
  int tileCol = threadIdx.x - FILTER_RADIUS;
  int tileRow = threadIdx.y - FILTER_RADIUS;
  // turning off the threads at the edges of the block
  if (col >= 0 && col < width && row >=0 && row < height) {
    if (tileCol>=0 && tileCol<OUT_TILE_DIM && tileRow>=0
               && tileRow<OUT_TILE_DIM){
      float Pvalue = 0.0f;
      for (int fRow = 0; fRow < 2*FILTER_RADIUS+1; fRow++) {
        for (int fCol = 0; fCol < 2*FILTER_RADIUS+1; fCol++) {
          Pvalue += F[fRow][fCol]*N_s[tileRow+fRow][tileCol+fCol];
        }
      }
      P[row*width+col] = Pvalue;
    }
  }
}
```

Tiled Kernel

All load, some compute

```
#define IN_TILE_DIM 32
#define OUT_TILE_DIM ((IN_TILE_DIM) - 2*(FILTER_RADIUS))
__constant__ float F_c[2*FILTER_RADIUS+1][2*FILTER_RADIUS+1];
__global__ void convolution_tiled_2D_const_mem_kernel(float *N, float *P,
                                                      int width, int height) {
  int col = blockIdx.x*OUT_TILE_DIM + threadIdx.x - FILTER_RADIUS;
  int row = blockIdx.y*OUT_TILE_DIM + threadIdx.y - FILTER_RADIUS;
  //loading input tile
  __shared__ N_s[IN_TILE_DIM][IN_TILE_DIM];
  if(row>=0 && row<height && col>=0 && col<width) {
    N_s[threadIdx.y][threadIdx.x] = N[row*width + col];
  } else {
    N_s[threadIdx.y][threadIdx.x] = 0.0;
  }
  __syncthreads();
  // Calculating output elements
  int tileCol = threadIdx.x - FILTER_RADIUS;
  int tileRow = threadIdx.y - FILTER_RADIUS;
  // turning off the threads at the edges of the block
  if (col >= 0 && col < width && row >=0 && row < height) {
    if (tileCol>=0 && tileCol<OUT_TILE_DIM && tileRow>=0
                  && tileRow<OUT_TILE_DIM){
      float Pvalue = 0.0f;
      for (int fRow = 0; fRow < 2*FILTER_RADIUS+1; fRow++) {
        for (int fCol = 0; fCol < 2*FILTER_RADIUS+1; fCol++) {
          Pvalue += F[fRow][fCol]*N_s[tileRow+fRow][tileCol+fCol];
        }
      }
      P[row*width+col] = Pvalue;
    }
  }
}
```

Tiled Kernel (previous slide) Approach



Alternative Tiled Kernel

We can also use a kernel which has every thread load in one entry, and compute one entry - then load all outside entries directly from global memory (this assumes we rely more heavily on L2 cache to store values across threadblocks)

```
#define TILE_DIM 32
__constant__ float F_c[2*FILTER_RADIUS+1][2*FILTER_RADIUS+1];
__global__ void convolution_cached_tiled_2D_const_mem_kernel(float *N,
                                    float *P, int width, int height) {
  int col = blockIdx.x*TILE_DIM + threadIdx.x;
  int row = blockIdx.y*TILE_DIM + threadIdx.y;
  //loading input tile
  __shared__ N_s[TILE_DIM][TILE_DIM];
  if(row<height && col<width) {
    N_s[threadIdx.y][threadIdx.x] = N[row*width + col];
  } else {
    N_s[threadIdx.y][threadIdx.x] = 0.0;
  }
  __syncthreads();
  // Calculating output elements
  // turning off the threads at the edges of the block
  if (col < width && row < height) {
    float Pvalue = 0.0f;
    for (int fRow = 0; fRow < 2*FILTER_RADIUS+1; fRow++) {
      for (int fCol = 0; fCol < 2*FILTER_RADIUS+1; fCol++) {
        if (threadIdx.x-FILTER_RADIUS+fCol >= 0 &&
            threadIdx.x-FILTER_RADIUS+fCol < TILE_DIM &&
            threadIdx.y-FILTER_RADIUS+fRow >= 0 &&
            threadIdx.y-FILTER_RADIUS+fRow < TILE_DIM){
          Pvalue += F[fRow][fCol]*N_s[threadIdx.y+fRow][threadIdx.x+fCol];
        }
        else {
          if (row-FILTER_RADIUS+fRow >= 0 &&
              row-FILTER_RADIUS+fRow < height &&
              col-FILTER_RADIUS+fCol >=0 &&
              col-FILTER_RADIUS+fCol < width) {
            Pvalue += F[fRow][fCol]*
                               N[(row-FILTER_RADIUS+fRow)*width+col-
FILTER_RADIUS+fCol];
          }
        }
      }
      P[row*width+col] = Pvalue;
    }
  }
}
```

Alternative Tiled Kernel

We can also use a kernel which has every thread load in one entry, and compute one entry - then load all outside entries directly from global memory (this assumes we rely more heavily on L2 cache to store values across threadblocks)

```
#define TILE_DIM 32
__constant__ float F_c[2*FILTER_RADIUS+1][2*FILTER_RADIUS+1];
__global__ void convolution_cached_tiled_2D_const_mem_kernel(float *N,
                                            float *P, int width, int height) {
  int col = blockIdx.x*TILE_DIM + threadIdx.x;
  int row = blockIdx.y*TILE_DIM + threadIdx.y;
  //loading input tile
  __shared__ N_s[TILE_DIM][TILE_DIM];
  if(row<height && col<width) {
    N_s[threadIdx.y][threadIdx.x] = N[row*width + col];
  } else {
    N_s[threadIdx.y][threadIdx.x] = 0.0;
  }
  __syncthreads();
  // Calculating output elements
  // turning off the threads at the edges of the block
  if (col < width && row < height) {
    float Pvalue = 0.0f;
    for (int fRow = 0; fRow < 2*FILTER_RADIUS+1; fRow++) {
      for (int fCol = 0; fCol < 2*FILTER_RADIUS+1; fCol++) {
        if (threadIdx.x-FILTER_RADIUS+fCol >= 0 &&
            threadIdx.x-FILTER_RADIUS+fCol < TILE_DIM &&
            threadIdx.y-FILTER_RADIUS+fRow >= 0 &&
            threadIdx.y-FILTER_RADIUS+fRow < TILE_DIM){
          Pvalue += F[fRow][fCol]*N_s[threadIdx.y+fRow][threadIdx.x+fCol];
        }
        else {
          if (row-FILTER_RADIUS+fRow >= 0 &&
              row-FILTER_RADIUS+fRow < height &&
              col-FILTER_RADIUS+fCol >=0 &&
              col-FILTER_RADIUS+fCol < width) {
            Pvalue += F[fRow][fCol]*
                           N[(row-FILTER_RADIUS+fRow)*width+col-
FILTER_RADIUS+fCol];
          }
        }
      }
      P[row*width+col] = Pvalue;
    }
  }
}
```

Connecting to HW4

- HW4 deals with all three of the discussed implementations using constant memory (+ 1 additional implementation)
- You will have to profile each of these & see what practical speedups look like
- So far, we have discussed that we should expect speedups when using shared memory → You have to verify whether this is true in practice for this kernel