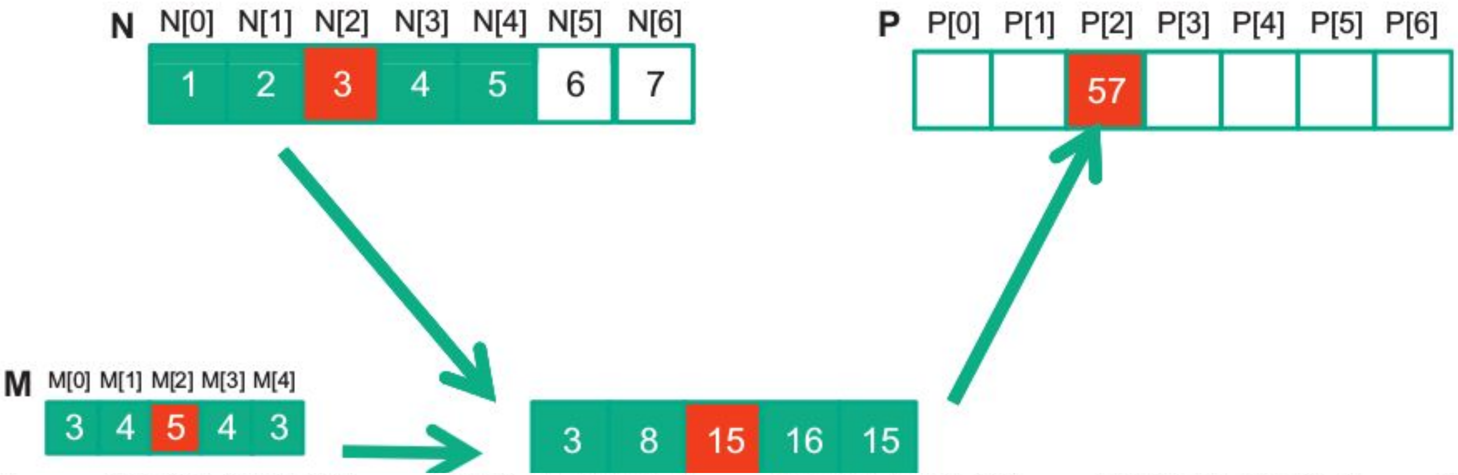
Parallel Patterns: Convolution

John H. Osorio Ríos

Background (1/6)

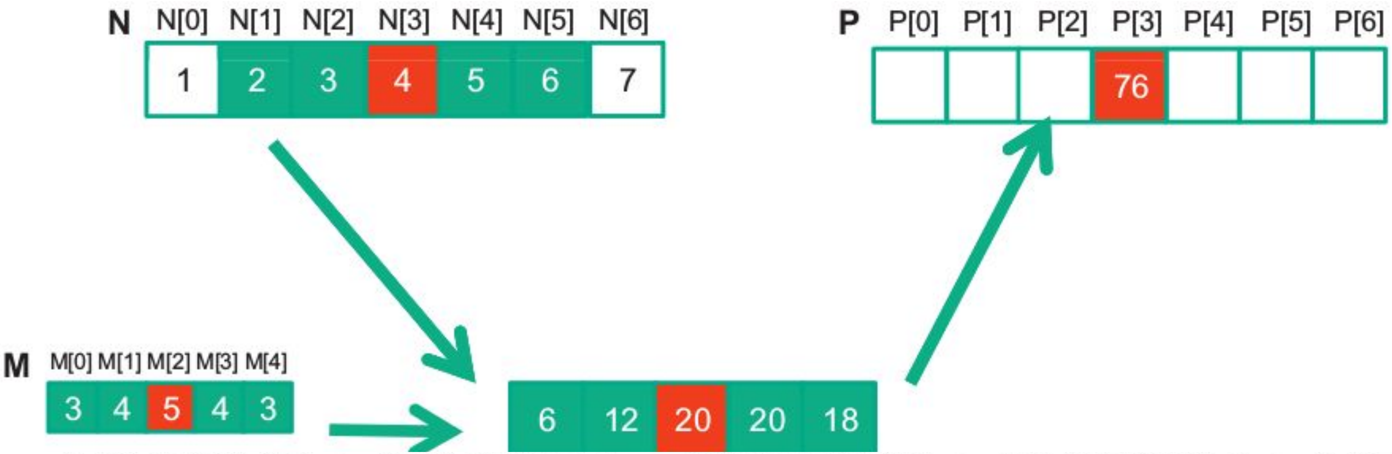
- Mathematically, convolution is an array operation where each output data element is a weighted sum of a collection of neighboring input elements.
- The weights used in the weighted sum calculation are defined by an input mask array, commonly referred to as the **convolution kernel**.

Background (2/6)



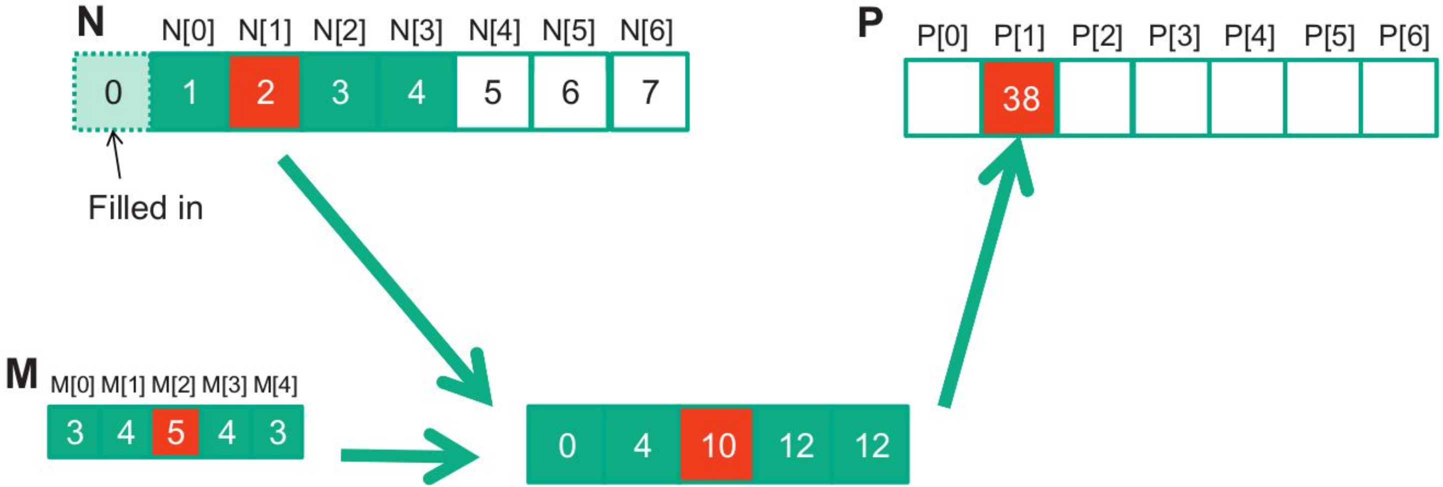
$$
\begin{aligned}
P[2] &= N[0]*M[0] + N[1]*M[1] + N[2]*M[2] + N[3]*M[3] + N[4]*M[4] \\
&= 1*3 + 2*4 + 3*5 + 4*4 + 5*3 \\
&= 57
\end{aligned}
$$

Background (3/6)



$$
\begin{aligned}
P[3] &= N[1]*M[0] + N[2]*M[1] + N[3]*M[2] + N[4]*M[3] + N[5]*M[4] \\
&= 2*3 + 3*4 + 4*5 + 5*4 + 6*3 \\
&= 76
\end{aligned}
$$

Background (4/6)



$$
\begin{aligned}
P[1] &= 0 * M[0] + N[0]*M[1] + N[1]*M[2] + N[2]*M[3] + N[3]*M[4] \\
&= 0 * 3 + 1*4 + 2*5 + 3*4 + 4*3 \\
&= 38
\end{aligned}
$$

Background (5/6)

N

1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	5	6
5	6	7	8	5	6	7
6	7	8	9	0	1	2
7	8	9	0	1	2	3

P

		321				

M

1	2	3	2	1
2	3	4	3	2
3	4	5	4	3
2	3	4	3	2
1	2	3	2	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

Background (6/6)

$$
\begin{aligned}
P2,2 = {} & N0,0*M0,0 + N0,1*M0,1 + N0,2*M0,2 + N0,3*M0,3 + N0,4*M0,4 \\
& + N1,0*M1,0 + N1,1*M1,1 + N1,2*M1,2 + N1,3*M1,3 + N1,4*M1,4 \\
& + N2,0*M2,0 + N2,1*M2,1 + N2,2*M2,2 + N2,3*M2,3 + N2,4*M2,4 \\
& + N3,0*M3,0 + N3,1*M3,1 + N3,2*M3,2 + N3,3*M3,3 + N3,4*M3,4 \\
& + N4,0*M4,0 + N4,1*M4,1 + N4,2*M4,2 + N4,3*M4,3 + N4,4*M4,4 \\
= {} & 1*1 + 2*2 + 3*3 + 4*2 + 5*1 \\
& + 2*2 + 3*3 + 4*4 + 5*3 + 6*2 \\
& + 3*3 + 4*4 + 5*5 + 6*4 + 7*3 \\
& + 4*2 + 5*3 + 6*4 + 7*3 + 8*2 \\
& + 5*1 + 6*2 + 7*3 + 8*2 + 5*1 \\
= {} & 1 + 4 + 9 + 8 + 5 \\
& + 4 + 9 + 16 + 15 + 12 \\
& + 9 + 16 + 25 + 24 + 21 \\
& + 8 + 15 + 24 + 21 + 16 \\
& + 5 + 12 + 21 + 16 + 5 \\
= {} & 321
\end{aligned}
$$

1D Parallel Convolution-Basic Algorithm (1/6)

```
__global__ void convolution_1D_basic_kernel(float *N, float
 *M, float *P,
int Mask_Width, int Width) {
// kernel body
}
```

1D Parallel Convolution-Basic Algorithm (2/6)

```
__global__ void convolution_1D_basic_kernel(float *N, float
 *M, float *P,
int Mask_Width, int Width) {
// kernel body
}
```

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

Output Element Index

1D Parallel Convolution-Basic Algorithm (3/6)

```
__global__ void convolution_1D_basic_kernel(float *N, float
 *M, float *P,
int Mask_Width, int Width) {
// kernel body
}
```

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

Output Element Index

```
Mask_Width= 2*n + 1
```

Mask_Width odd number and the convolution is symmetric

1D Parallel Convolution-Basic Algorithm (4/6)

```
__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;

}
```

1D Parallel Convolution-Basic Algorithm (5/6)

```
__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;

}
```

Memory Bandwidth ?????

Ratio ?????

1D Parallel Convolution-Basic Algorithm (6/6)

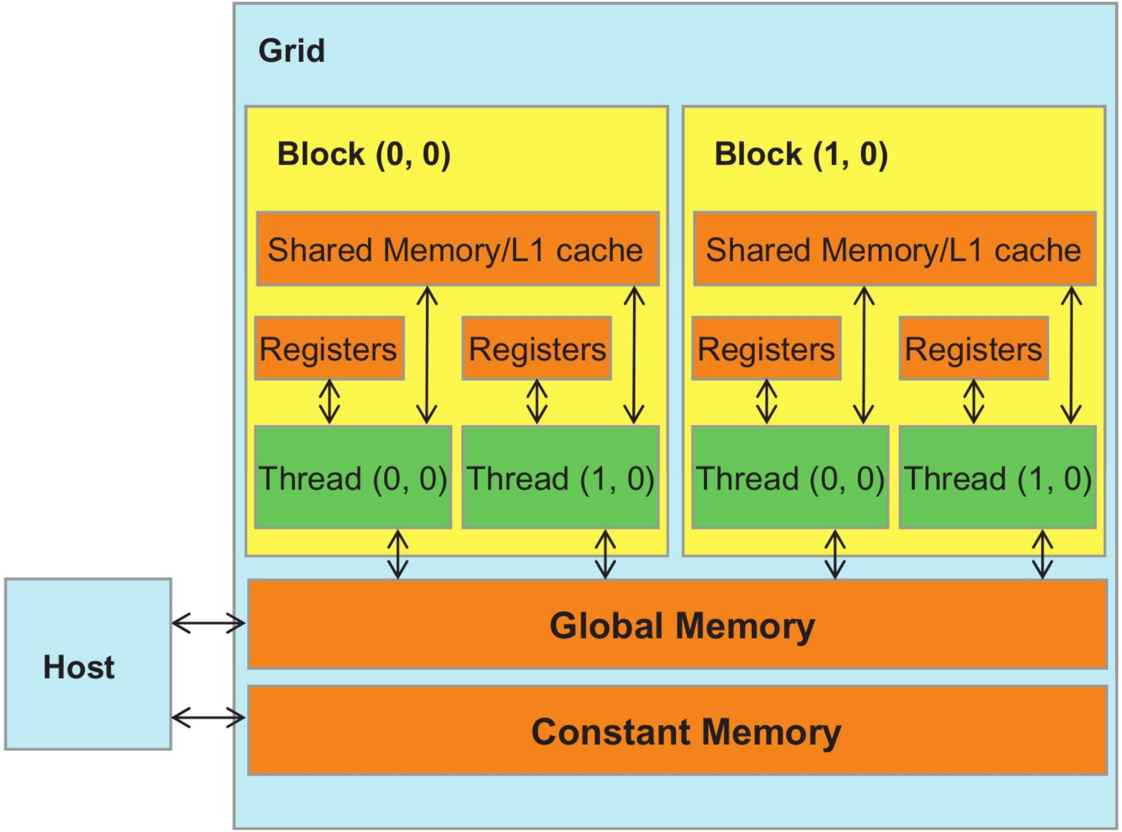
```
__global__ void convolution_1D_basic_kernel(float *N, float *M, float *P,
  int Mask_Width, int Width) {

  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;

}
```

Memory Bandwidth ????? Bad

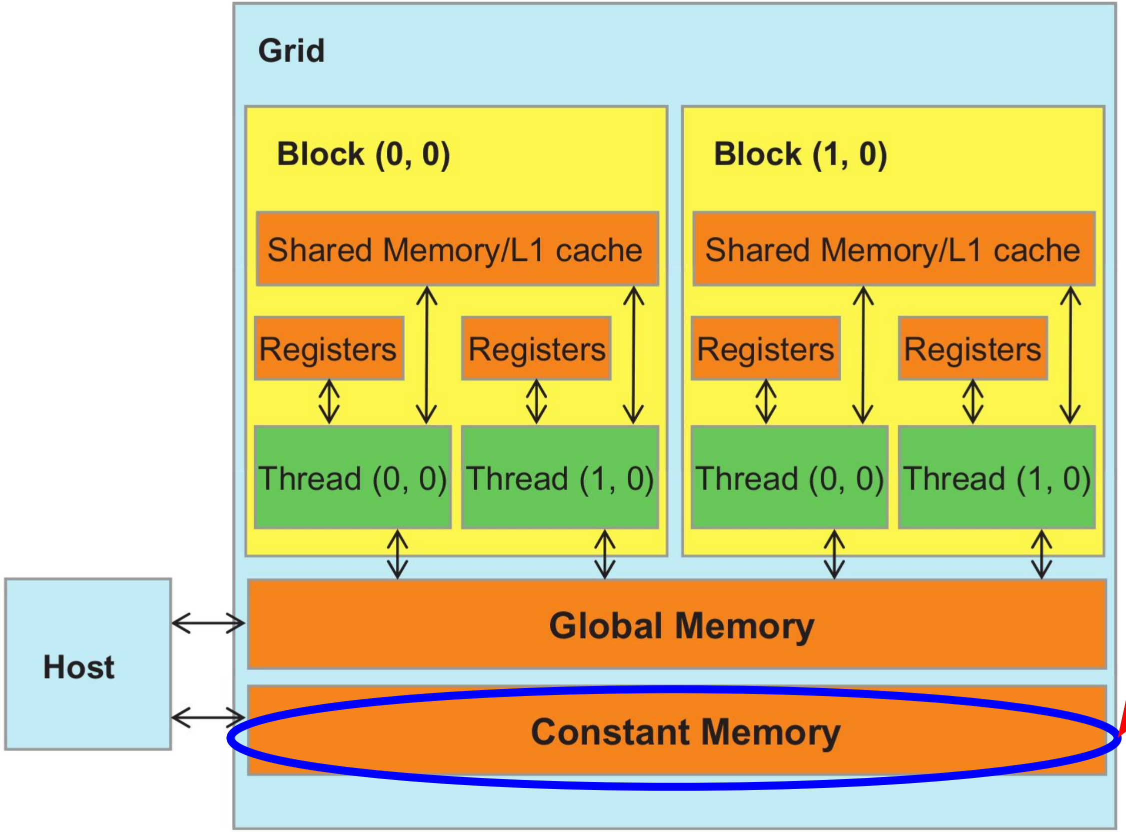
Ratio ?????

1.0

Constant Memory and Caching (1/11)

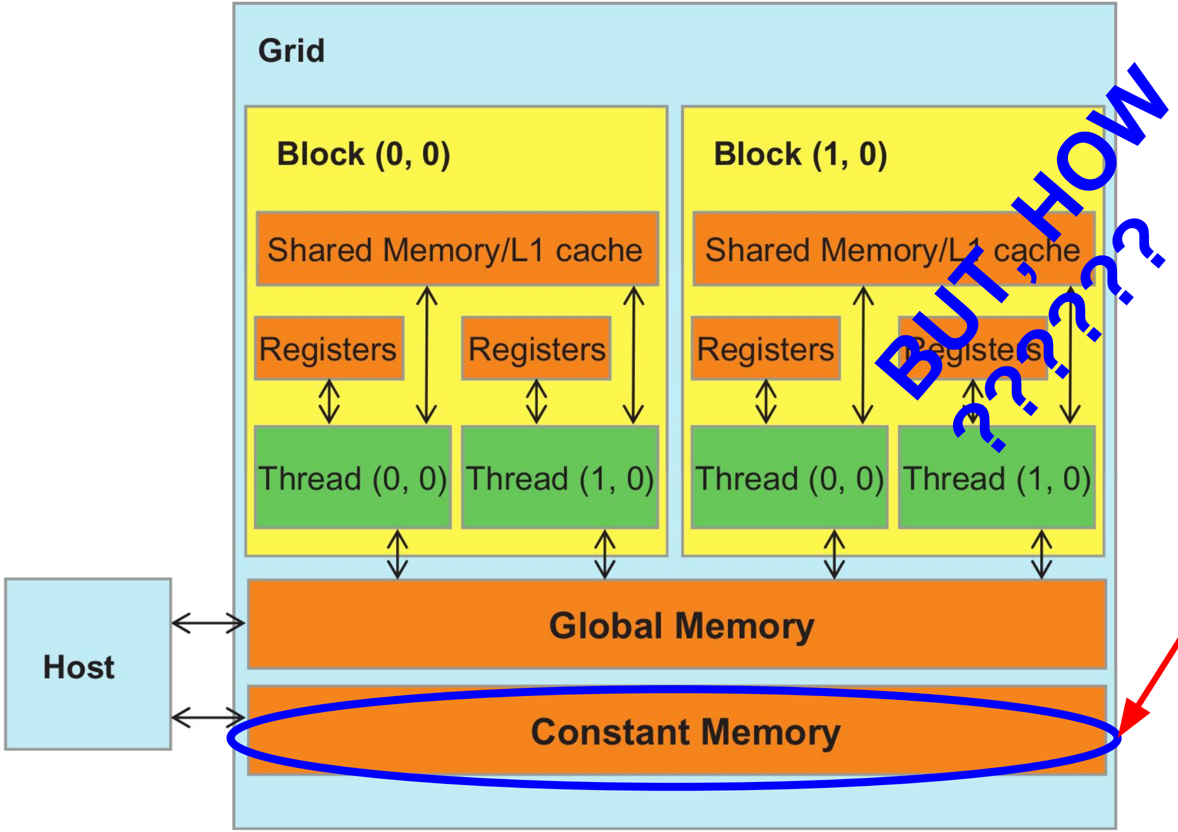


Constant Memory and Caching (2/11)



We need this !!!!!

Constant Memory and Caching (3/11)



We need this !!!!!

Constant Memory and Caching (4/11)

```
#define MAX_MASK_WIDTH 10
__constant__ float M[MAX_MASK_WIDTH];
```

Host Code

Constant Memory and Caching (5/11)

```
#define MAX_MASK_WIDTH 10
__constant__ float M[MAX_MASK_WIDTH];
```

Host Code

And … The DATA

???? In device ????

Constant Memory and Caching (6/11)

```
#define MAX_MASK_WIDTH 10
__constant__ float M[MAX_MASK_WIDTH];
```

Host Code

```
cudaMemcpyToSymbol(M, h_M, Mask_Width*sizeof(float));
```

Special Memory Copy Function ...

Constant Memory and Caching (7/11)

```
#define MAX_MASK_WIDTH 10
__constant__ float M[MAX_MASK_WIDTH];
```

Host Code

```
cudaMemcpyToSymbol(M, h_M, Mask_Width*sizeof(float));
```

Special Memory Copy Function ...

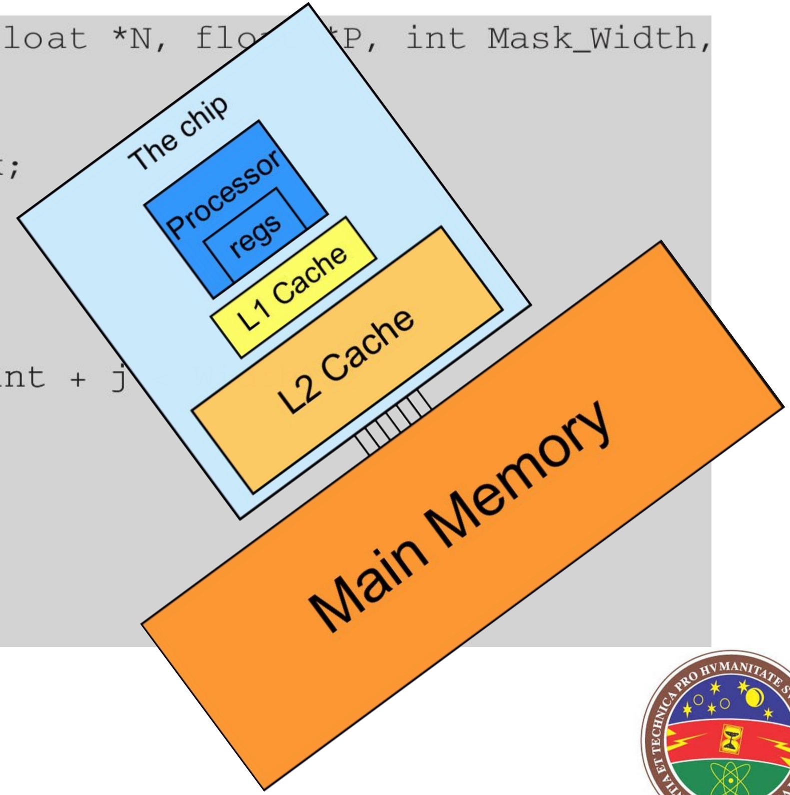
```
cudaMemcpyToSymbol(dest, src, size)
```

Constant Memory and Caching (8/11)

```
__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;

}
```

Constant Memory and Caching (9/11)

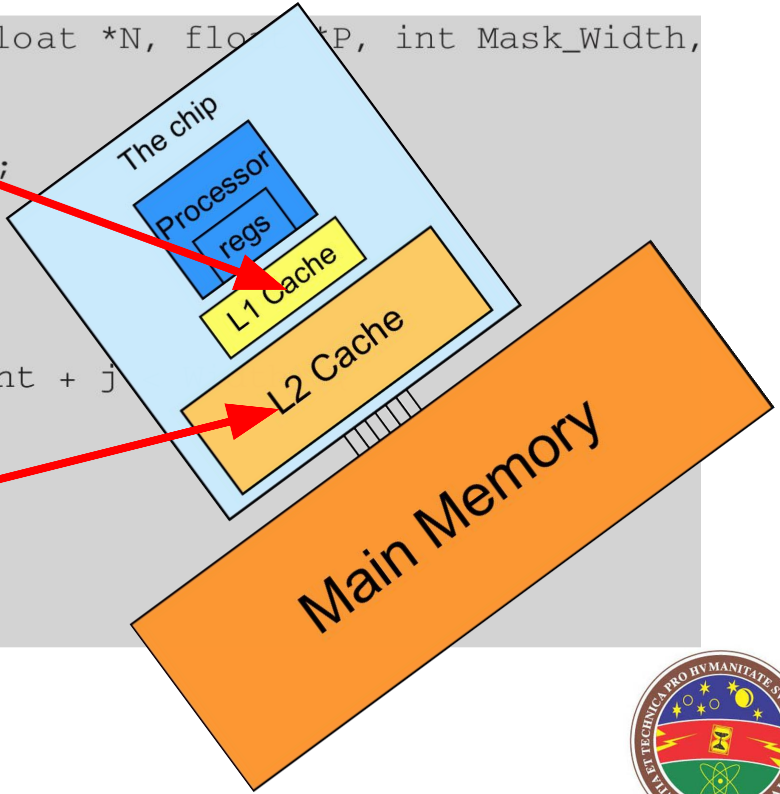
```
__global__ void convolution_1D_ba sic_kernel(float *N, flo   *P, int Mask_Width,
  int Width) {

  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
      Pvalue += N[N_start_point + j]*M[j];
    }
  }
  P[i] = Pvalue;

}
```



Constant Memory and Caching (10/11)

```
__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
      Pvalue += N[N_start_point + j]*M[j];
    }
  }
  P[i] = Pvalue;

}
```

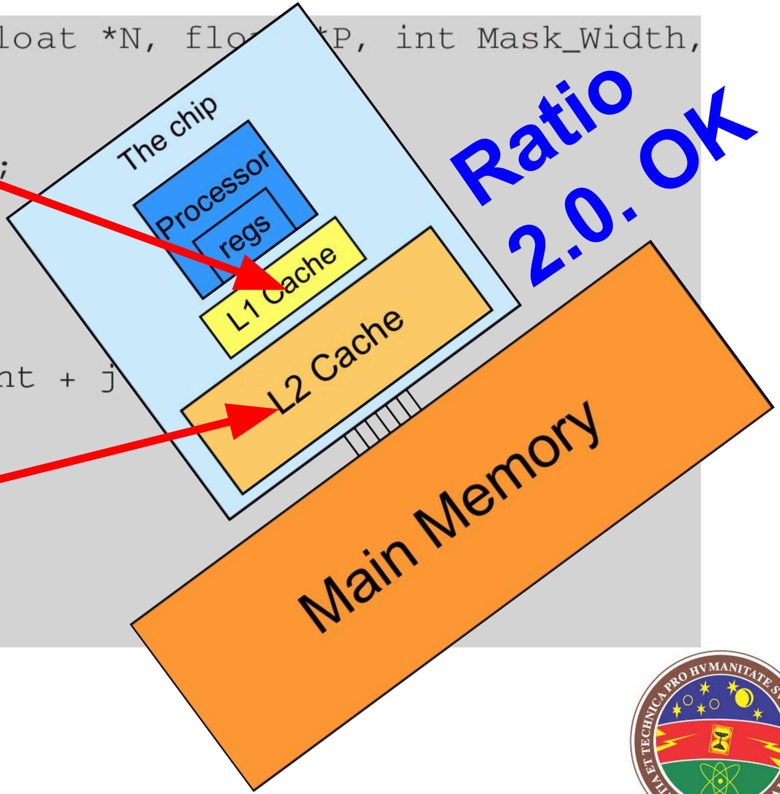
The memory is marked to be used from L1 or L2 Cache



Constant Memory and Caching (11/11)

```
__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;

}
```

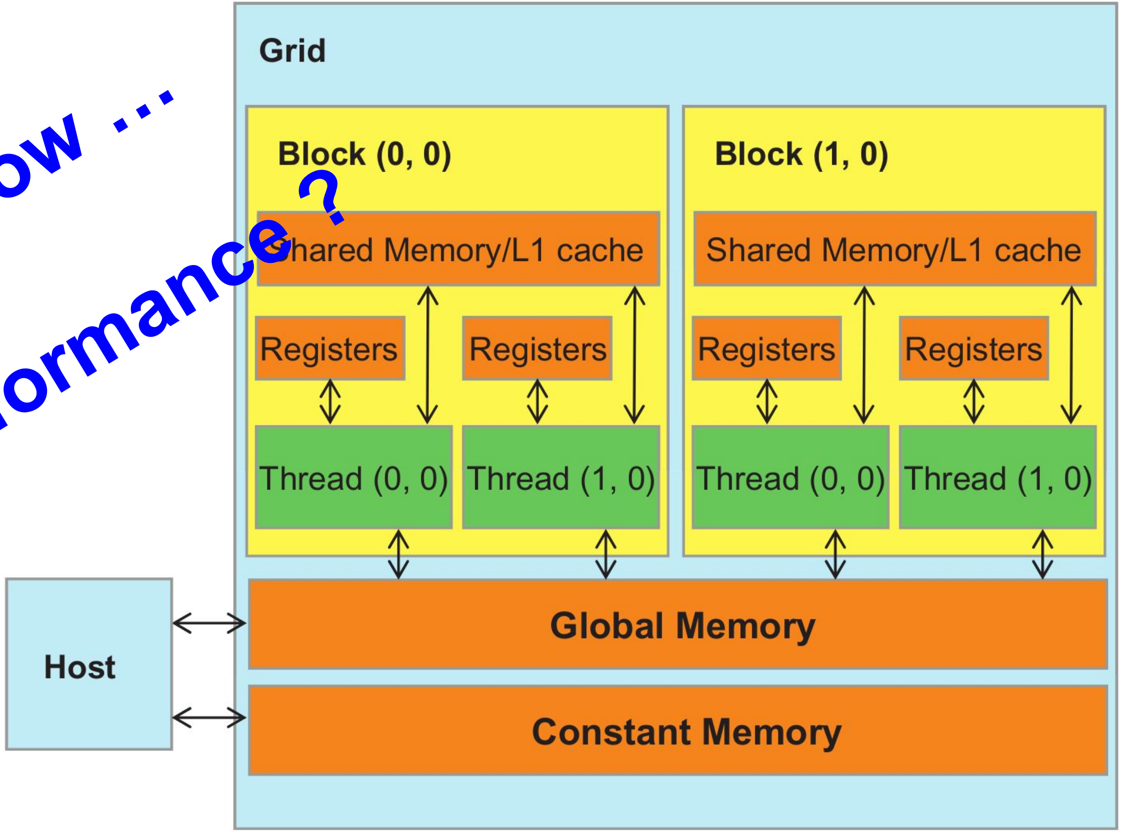
Aggressively cache the constant memory variables during kernel execution.



Ratio 2.0. OK

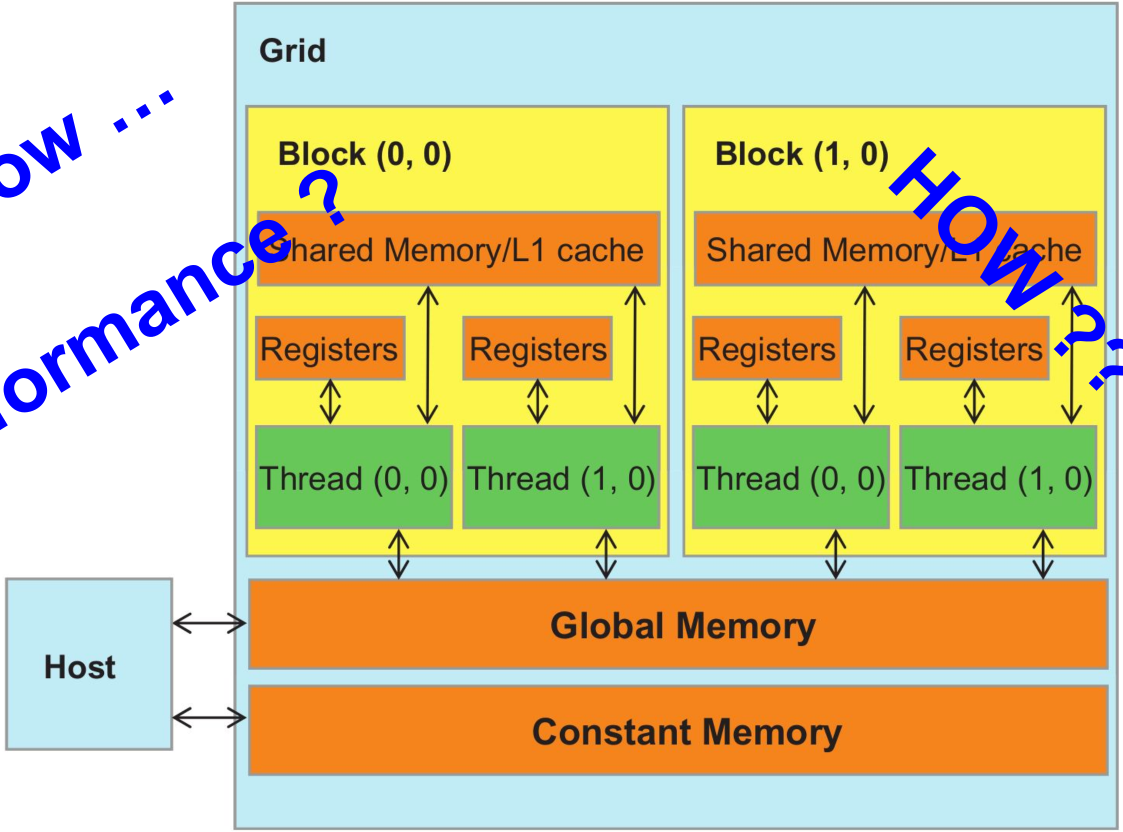
Tiled 1D Convolution With Halo Elements(1/25)

And, now …
More
performance ?



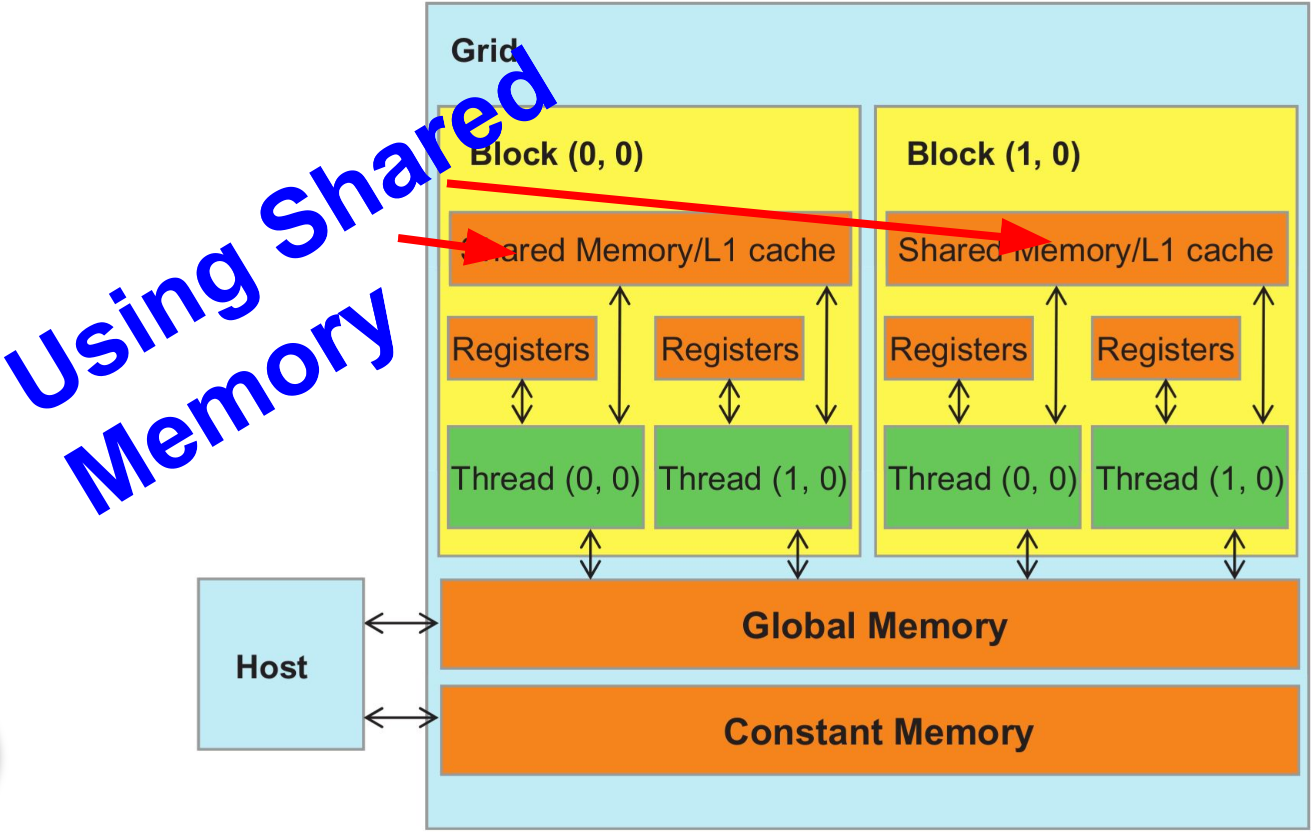
Tiled 1D Convolution With Halo Elements(2/25)

And, now …
More performance ?

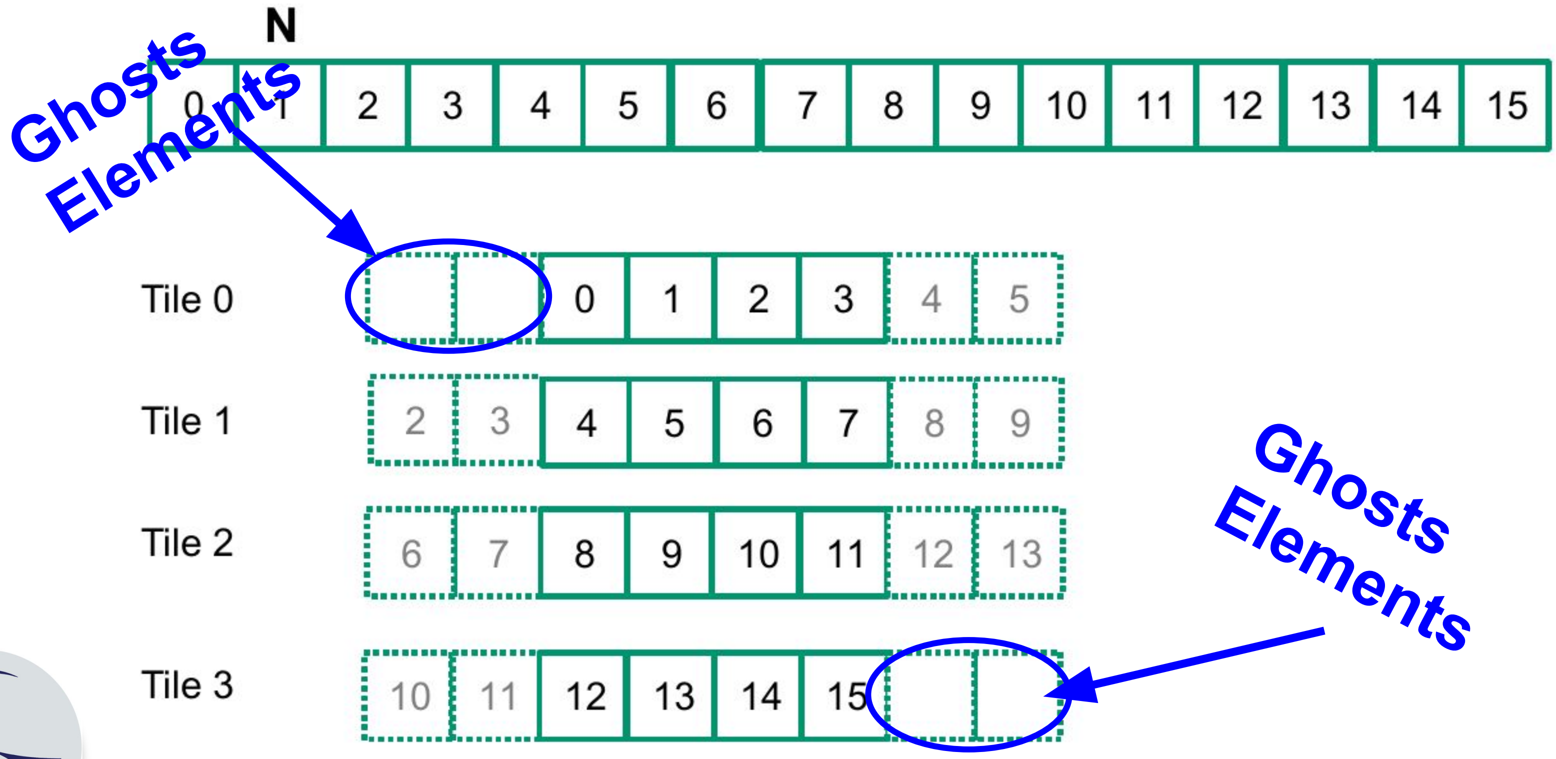


HOW ????

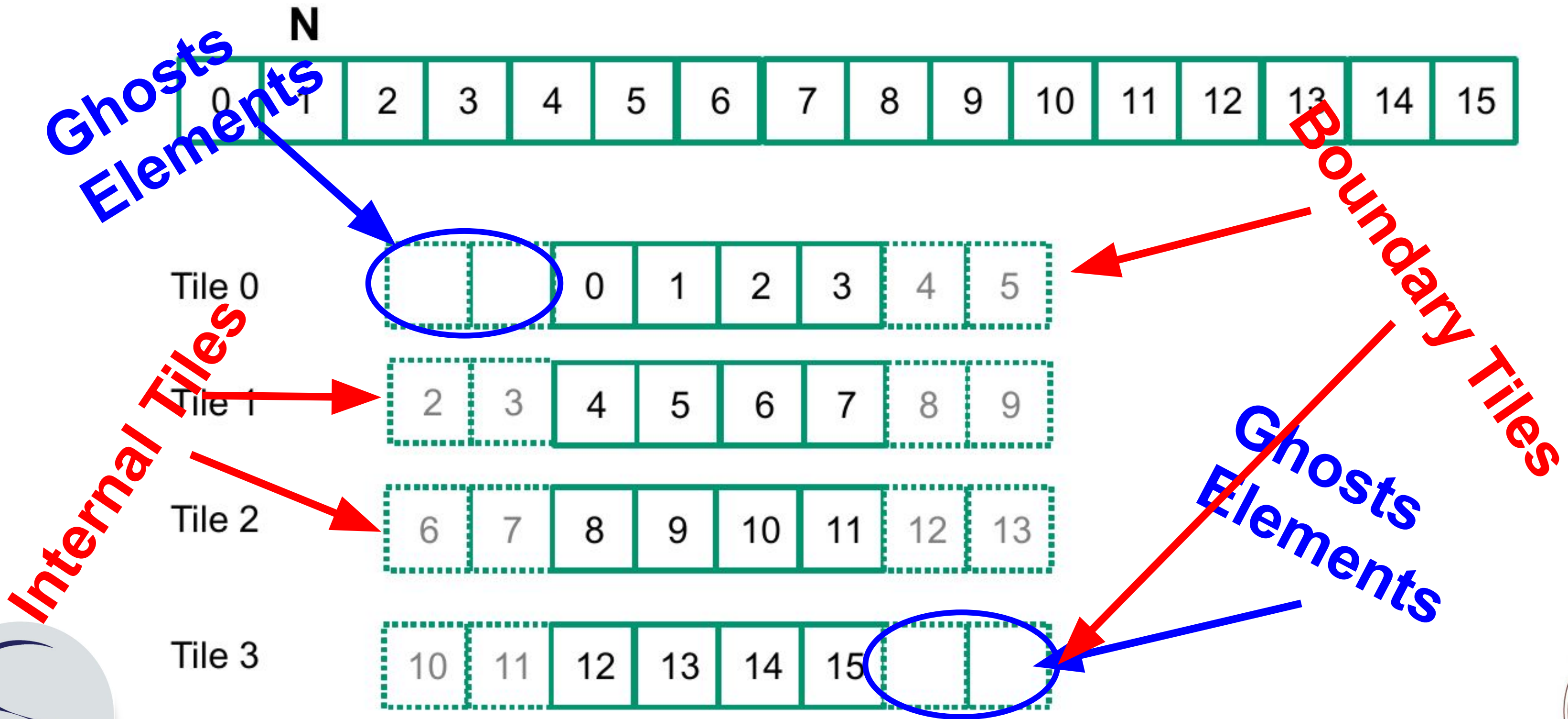
Tiled 1D Convolution With Halo Elements(3/25)



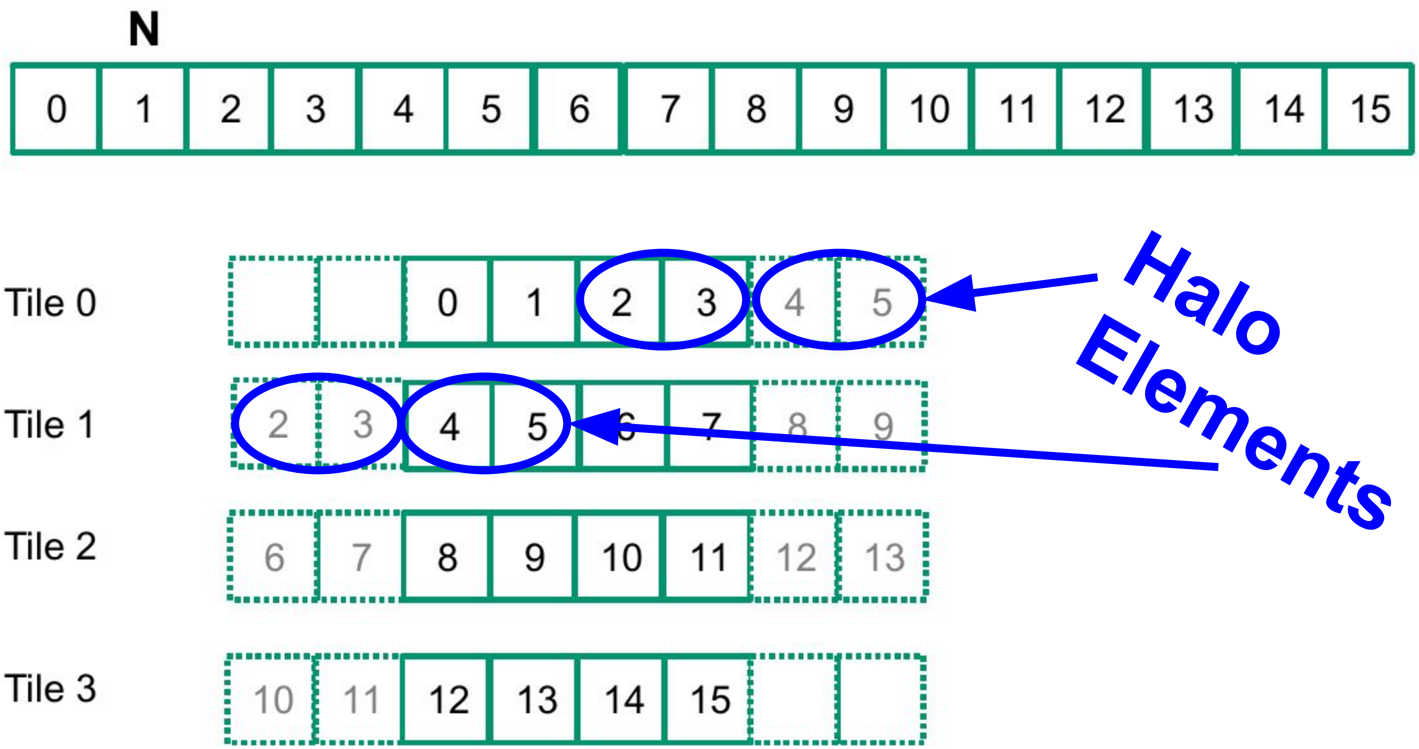
Tiled 1D Convolution With Halo Elements(4/25)



Tiled 1D Convolution With Halo Elements(5/25)



Tiled 1D Convolution With Halo Elements(6/25)



Tiled 1D Convolution With Halo Elements(7/25)

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

In the last example is 4

Tiled 1D Convolution With Halo Elements(8/25)

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

En el ejemplo anterior es igual a 4

Y este es igual a 5

Tiled 1D Convolution With Halo Elements(9/25)

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

En el ejemplo anterior es igual a 4

Y este es igual a 5

```
n = Mask_Width/2
```

Mask_Width es impar. Para el ejemplo n = 2

Tiled 1D Convolution With Halo Elements(10/25)

```
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];
}
```

We map the index using previous block (tile)

Tiled 1D Convolution With Halo Elements(11/25)

```
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];
}
```

We map the index using previous block (tile)

This to load the left halo elements into shared memory

Tiled 1D Convolution With Halo Elements(12/25)

```
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];
}
```

We pick up the last n threads

Tiled 1D Convolution With Halo Elements(13/25)

```
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];
}
```

We check for ghosts elements

We pick up the last n threads

Tiled 1D Convolution With Halo Elements(14/25)

We need to load the center elements

Tiled 1D Convolution With Halo Elements(15/25)

We need to load the center elements

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

Tiled 1D Convolution With Halo Elements(16/25)

We need to load the center elements

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

The first n elements were loaded before

Tiled 1D Convolution With Halo Elements(17/25)

We need to load right halo elements

Tiled 1D Convolution With Halo Elements(18/25)

We need to load right halo elements

```
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];
}
```

Tiled 1D Convolution With Halo Elements(19/25)

We need to load right halo elements

```
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];
}
```

Similar to load left halo elements

Tiled 1D Convolution With Halo Elements(20/25)

We need to load right halo elements

```
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];
}
```

Check yourself ... :)

Similar to load left halo elements

Tiled 1D Convolution With Halo Elements(21/25)

Now we have all data into shared memory

Tiled 1D Convolution With Halo Elements(22/25)

Now we have all data into shared memory

We can make the calculations

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

Tiled 1D Convolution With Halo Elements(23/25)

Now we have all data into shared memory

We can make the calculations

Don't forget syncthreads

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

Tiled 1D Convolution With Halo Elements(24/25)

```
__global__ void convolution_1D_basic_kernel(float *N, float *P, int M
  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(intj = 0; j < Mask_Width; j++) {
    Pvalue += N_ds[threadIdx.x + j]*M[j];
  }
  P[i] = Pvalue;

}
```

TODO (25/25)

- Everything you need to have fun is on the Homework.

THANKS

john@sirius.utp.edu.co

NVIDIA
DEEP LEARNING INSTITUTE
UNIVERSITY AMBASSADOR