1D Convolution & Shared Memory

By Nicolas Agostini

This is the **discrete convolution** formula

$$y[i] = input[i] * kernel[i] = \sum_{k} input[k] \cdot kernel[i-k]$$

This is the **discrete convolution** formula

$$y[i] = input[i] * kernel[i] = \sum_k input[k] \cdot kernel[i-k]$$

input				2	1	4	1	1	0	1	3	1	2	2	4

kernel		1	4	2	-1	-5

This is the **discrete convolution** formula

$$y[i] = input[i] * kernel[i] = \sum_k input[k] \cdot kernel[i-k]$$

input														
			2	1	4	1	1	0	1	3	1	2	2	4

kernel						
		1	4	2	-1	-5

Note the minus (-) sign?

This is the **discrete convolution** formula

$$y[i] = input[i] * kernel[i] = \sum_k input[k] \cdot kernel[i-k]$$

input				2	1	4	1	1	0	1	3	1	2	2	4

-5	-1	2	4	1		kernel

Note the minus (-) sign?

It would require the kernel to be flipped during computation

This is the **discrete cross-correlation** formula

$$y[i] = input[i] * kernel[i] = \sum_k input[k] \cdot kernel[i + k]$$

input				2	1	4	1	1	0	1	3	1	2	2	4

kernel		1	4	2	-1	-5

Note the plus sign

Kernel does not have to be flipped

This is the **discrete** ~~**cross-correlation**~~ formula
This is the **discrete convolution** formula

$$y[i] = input[i] * kernel[i] = \sum_k input[k] \cdot kernel[i+k]$$

input				2	1	4	1	1	0	1	3	1	2	2	4

kernel		1	4	2	-1	-5

Outside of signal processing, we will call cross-correlations as convolutions.

It is fine…
Everyone does it

For a given output y[i], let say y[2]

$$y[2] = input[2] * kernel[2] = \sum_k input[k] \cdot kernel[2+k]$$

k	**input[k]**	**kernel[2+k]**
-10	??	??

input				2	1	4	1	1	0	1	3	1	2	2	4

kernel		1	4	2	-1	-5

For a given output y[i], let say y[2]

$$y[2] = input[2] * kernel[2] = \sum_k input[k] \cdot kernel[2+k]$$

k	**input[k]**	**kernel[2+k]**
-10	0	0

Accessing outside of the arrays = 0

input	2	1	4	1	1	0	1	3	1	2	2	4

kernel	1	4	2	-1	-5

For a given output y[i], let say y[2]

$$y[2] = input[2] * kernel[2] = \sum_k input[k] \cdot kernel[2+k]$$

k	**input[k]**	**kernel[2+k]**
-10	0	0
-3	0	0

input				2	1	4	1	1	0	1	3	1	2	2	4

kernel		1	4	2	-1	-5

For a given output y[i], let say y[2]

$$y[2] = input[2] * kernel[2] = \sum_k input[k] \cdot kernel[2+k]$$

k	**input[k]**	**kernel[2+k]**
-10	0	0
-3	0	0
-2	0	1

Value of kernel[2-2]=kernel[0]

input				2	1	4	1	1	0	1	3	1	2	2	4

kernel		1	4	2	-1	-5

For a given output y[i], let say y[2]

$$y[2] = input[2] * kernel[2] = \sum_k input[k] \cdot kernel[2+k]$$

k	**input[k]**	**kernel[2+k]**
-10	0	0
-3	0	0
-2	0	1
-1	0	4

input				2	1	4	1	1	0	1	3	1	2	2	4

kernel		1	4	2	-1	-5

For a given output y[i], let say y[2]

$$y[2] = input[2] * kernel[2] = \sum_k input[k] \cdot kernel[2+k]$$

k	**input[k]**	**kernel[2+k]**
-10	0	0
-3	0	0
-2	0	1
-1	0	4
0	2	2

input				2	1	4	1	1	0	1	3	1	2	2	4

kernel		1	4	2	-1	-5

For a given output y[i], let say y[2]

$$y[2] = input[2] * kernel[2] = \sum_k input[k] \cdot kernel[2+k]$$

k	input[k]	kernel[2+k]
-10	0	0
-3	0	0
-2	0	1
-1	0	4
0	2	2
1	1	-1

input				2	1	4	1	1	0	1	3	1	2	2	4

kernel		1	4	2	-1	-5

For a given output y[i], let say y[2]

$$y[2] = input[2] * kernel[2] = \sum_k input[k] \cdot kernel[2+k]$$

k	**input[k]**	**kernel[2+k]**
-10	0	0
-3	0	0
-2	0	1
-1	0	4
0	2	2
1	1	-1
2	4	-5

input				2	1	4	1	1	0	1	3	1	2	2	4

kernel		**1**	**4**	**2**	**-1**	**-5**

For a given output y[i], let say y[2]

$$y[2] = input[2] * kernel[2] = \sum_k input[k] \cdot kernel[2+k]$$

k	**input[k]**	**kernel[2+k]**
-10	0	0
-3	0	0
-2	0	1
-1	0	4
0	2	2
1	1	-1
2	4	-5
3	1	0
4	1	0
5	0	0
…		

input				2	1	4	1	1	0	1	3	1	2	2	4

kernel		**1**	**4**	**2**	**-1**	**-5**

The formula tells me to do a
"sum of the products"

$$y[2] = input[2] * kernel[2] = \sum_k input[k] \cdot kernel[2+k]$$

input[k]		**kernel[2+k]**
0	X	0
0	X	0
0	X	1
0	X	4
2	X	2
1	X	-1
4	X	-5
1	X	0
1	X	0
0	X	0
…		
	…	

The formula tells me to do a
"sum of the products"

$$y[2] = input[2] * kernel[2] = \sum_k input[k] \cdot kernel[2+k]$$

input[k] **kernel[2+k]**

$$\sum$$

0
0
0
0
4
-1
-20
0
0
0

We found one output value (for i=2), now we must do it for all the other "i"s

$$y[2] = input[2] * kernel[2] = \sum_k input[k] \cdot kernel[2+k]$$

$$y[2] = -17$$

input[k] **kernel[2+k]**

$$\sum$$

0
0
0
0
4
-1
-20
0
0
0

But what are we doing visually?

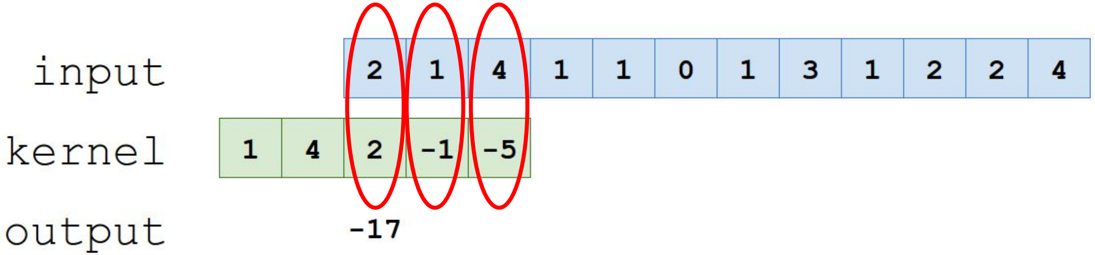
Perform the dot product of the areas that overlap

$$\sum_j a[j] \cdot b[j]$$

input			2	1	4	1	1	0	1	3	1	2	2	4
kernel	1	4	2	-1	-5									
output			-17											

Perform the dot product of the areas that overlap

$$\sum_j a[j] \cdot b[j]$$

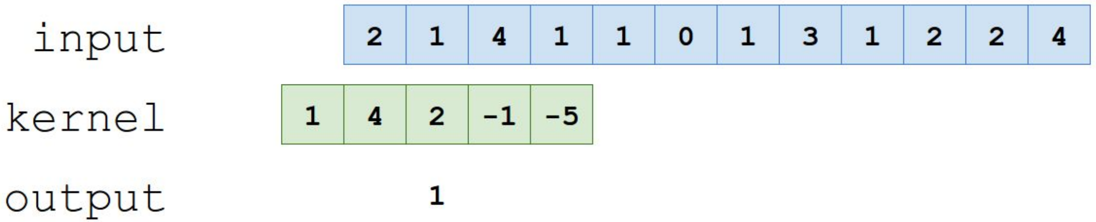


Perform the dot product of the areas that overlap

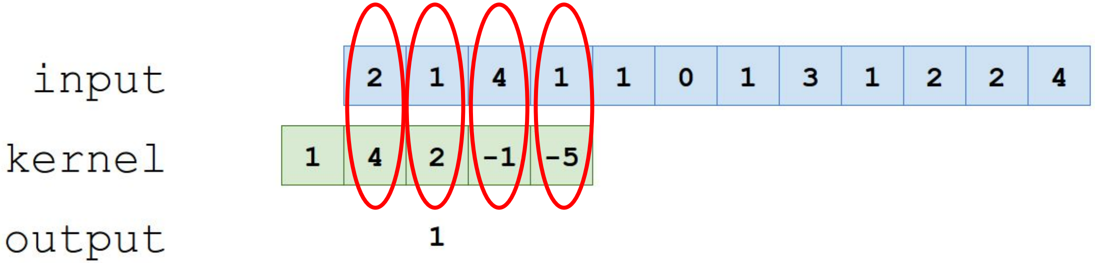
$$\sum_j a[j] \cdot b[j]$$

input			2	1	4	1	1	0	1	3	1	2	2	4
kernel	1	4	2	-1	-5									
output			-17											

Slide the filter and repeat



Now 4 elements overlap

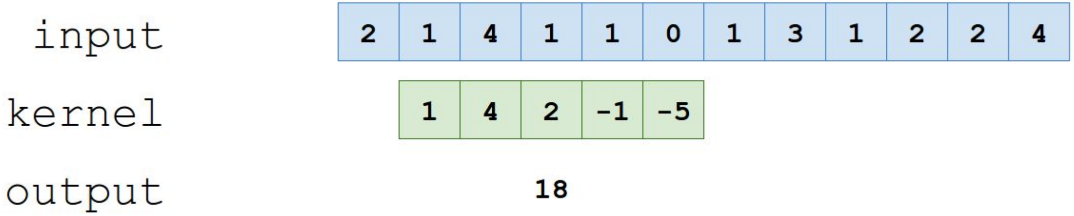


Now 5 elements overlap

input	2	1	4	1	1	0	1	3	1	2	2	4
kernel	1	4	2	-1	-5							
output			8									

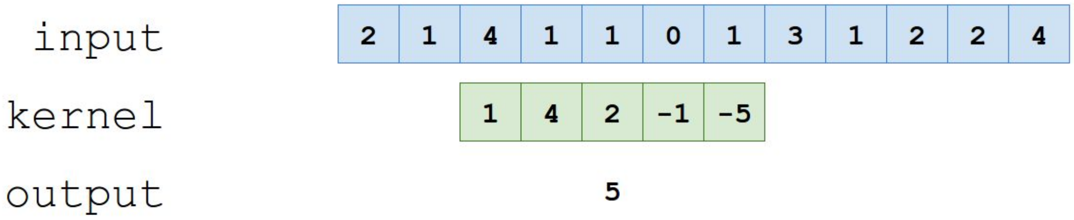
Now 5 elements overlap

On GPU: each thread calculates one output value



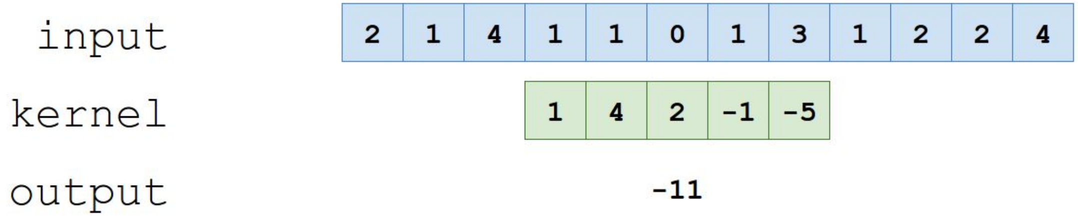
Now 5 elements overlap

On GPU: each thread calculates one output value



And so on…

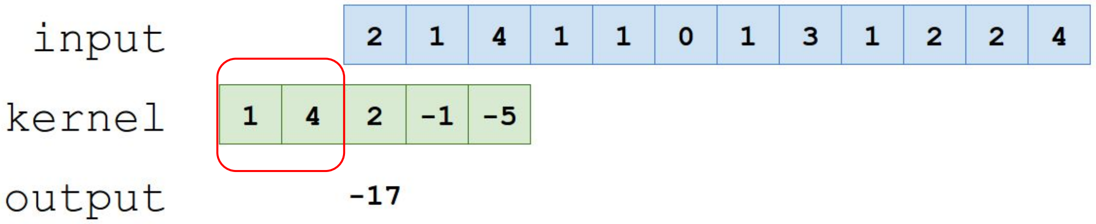
On GPU: each thread calculates one output value



Final output

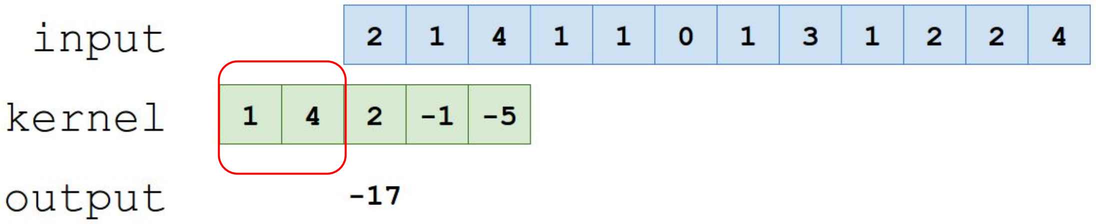
input	2	1	4	1	1	0	1	3	1	2	2	4					
kernel													1	4	2	-1	-5
output	-17	1	8	18	5	-11	-5	-1	-3	-11	9	18					

The “RADIUS”?



The "RADIUS"?

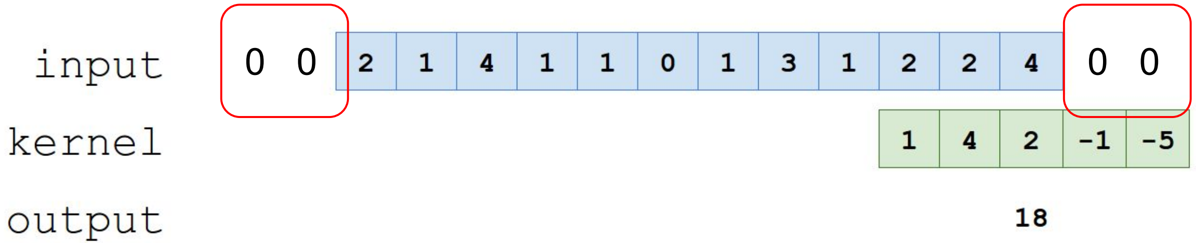
Kernel of size 5: radius of size 2!



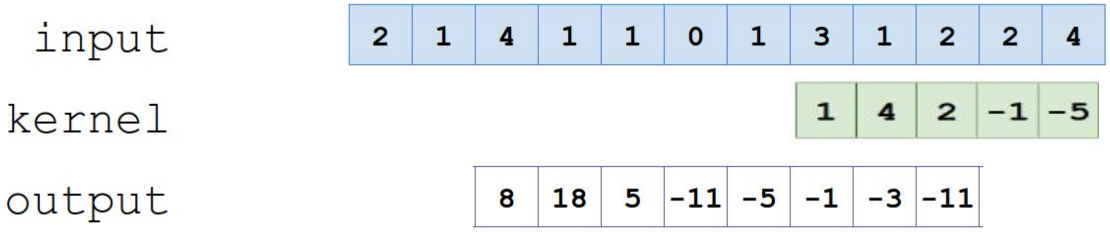
The "RADIUS" -> padding with zeros

input	0	0	2	1	4	1	1	0	1	3	1	2	2	4
kernel	1	4	2	-1	-5									
output			-17											

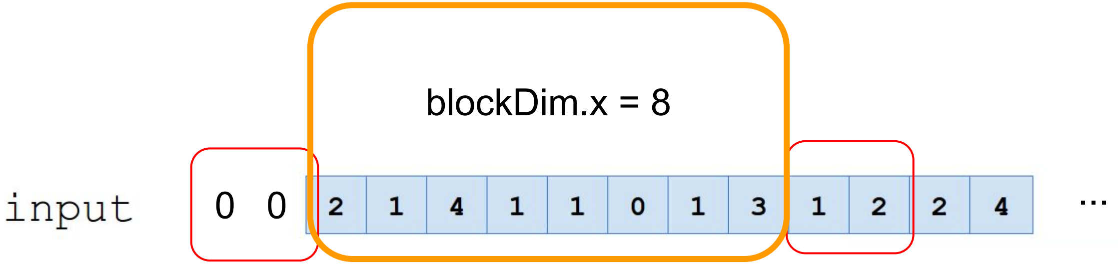
The “RADIUS” -> padding with zeros
Must do this at the beginning and at the end



Otherwise you will end up with less elements in the output array

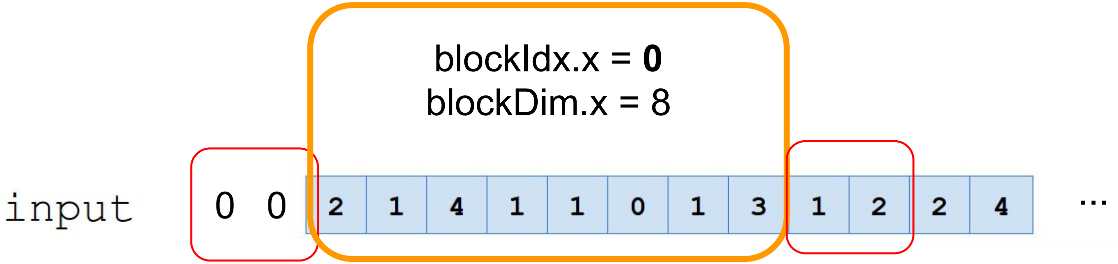


How about shared memory?



Each block must allocate the
blockDim.x + 2 times RADIUS

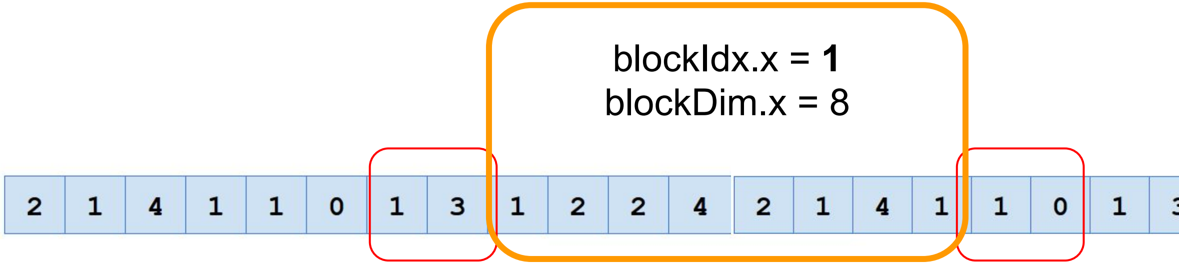
How about shared memory?



Each block must copy into shared memory elements in the orange **and** red boxes

How about shared memory?

input



Each block must copy into shared memory elements in the orange **and** red boxes

Carry On!