CSE 4373/5373 - General Purpose GPU Programming

GPU Pattern: Convolution

Alex Dillhoff

University of Texas at Arlington

Convolution

- In lab 1, you implemented a blur filter.
- Typically the filter would be weighted, but we used a simple average.
- The operation you implemented was a convolution.

Convolution

- Convolutions are common in image processing.
- They are also the primary operator for Convolutional Neural Networks (CNNs).
- Their output can be thought of as a response between the input and a pattern, represented by a filter.

Convolution

This operator can and has been efficiently implemented in a GPU.

Through studying this pattern, you will learn to utilize constant memory storage and shared memory storage to efficiently implement a convolution kernel.

Convolution

- A convolution is a function that takes two functions as input and produces a third function as output.
- The first function is the input and the second function is the kernel.
- The output is called the feature map.
- The kernel is also sometimes called the filter.

Convolution

$$(f * g)(t) = \int f(t-a)g(a)da$$

Convolution

We can view them more concretely by considering the functions to be vectors.

For example, let the function f be an input vector x and w be a kernel representing a filter.

Convolution

$$(x * w)(t) = \int x(t-a)w(a)da.$$

Convolution

The result is a **feature map** representing the response of the kernel at each location in the input.

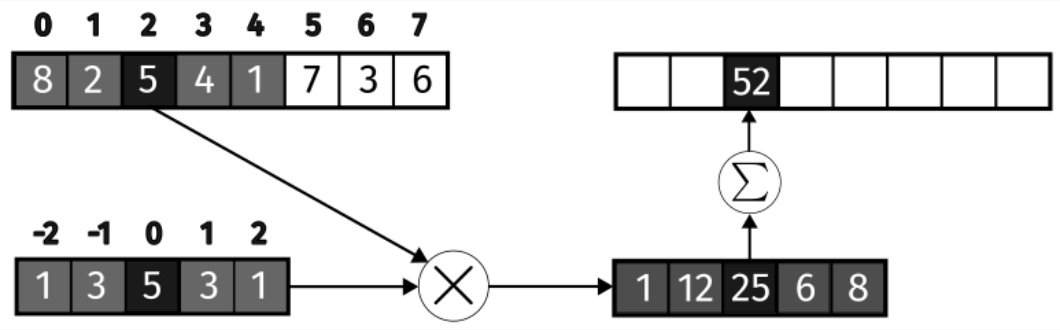
In the case of discrete values, it is common to use an odd-sized kernel and center it on an input value.

The kernel size is given by some radius r.

Convolution

$$(x * w)(t) = \sum_{-r}^{r} x(t-r)w(r).$$

Convolution



1D Convolution between a vector of size 8 and a kernel of size 5.

Convolution

Convolution is defined in such a way that the kernel is traversed in an inverted manner.

In the previous example, y_2 is computed by applying the kernel to $\mathbf{x}$ centered on x_2.

Convolution

The calculation in terms of the location accesses is

$$y_2 = x_4 w_{-2} + x_3 w_{-1} + x_2 w_0 + x_1 w_1 + x_0 w_2.$$

Convolution

This operation is very similar to the *correlation* operator, which is defined as

$$(x \star w)(t) = \sum_{-r}^{r} x(t+r)w(r).$$

Convolution

- We can use the correlation operator to compute the convolution by flipping the kernel.
- In this case, the calculation can be represented using the dot product.
- We can also slightly adjust the indexing so that the first index is 0.

Convolution

$$y_i = \sum_{k=0}^{2r} x_{i+k-r} w_{2r-k}.$$

Convolution

- Note that the convolution shown above would be undefined for $i = 0$ and $i = 1$ since the kernel would be accessing negative indices.
- Based on the definition, we would ignore these values.
- This is called a *valid* convolution.
- The output size is then $n - 2r$.

Convolution

- There is also a *full* convolution where the output size is n.
- In this case, the kernel is padded with zeros so that it can be applied to all elements of the input.

2D Convolution

Image convolutions use 2D filters applied to 2D images.

For a filter with radius r, size of the filter is $(2r + 1) \times (2r + 1)$.

2D Convolution

$$(x * w)(i, j) = \sum_{-r}^{r} \sum_{-r}^{r} x(i - r, j - r) w(r, s).$$

Properties of Convolutions

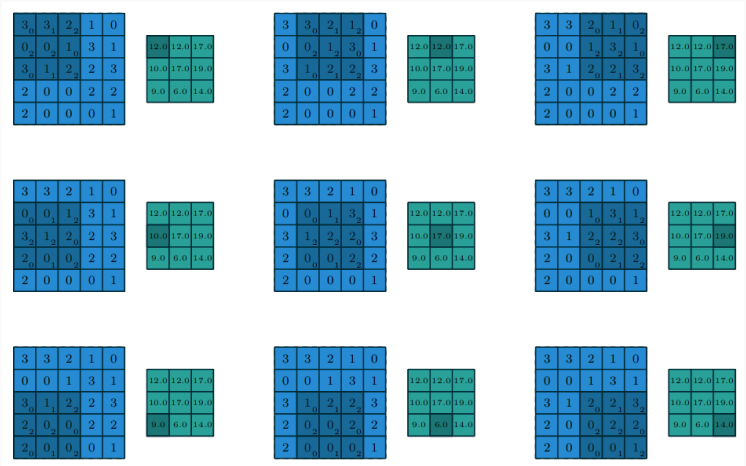
Properties of Convolutions

- Convolutional networks are commonly built on *full* or *valid* convolutions.
- There are many variants of convolutions that produce different effects on the input.
- Common variations are *padding*, *strides*, and *dilation*.

Padding

- By definition, a convolution of an input with a filter of size $n \times n$ will produce an output of size $(m - n + 1) \times (m - n + 1)$, where m is the size of the input.
- This means that the output will be smaller than the input.
- This is often referred to as a **valid** convolution.

Padding



A valid convolution (Dumoulin et al.)

Padding

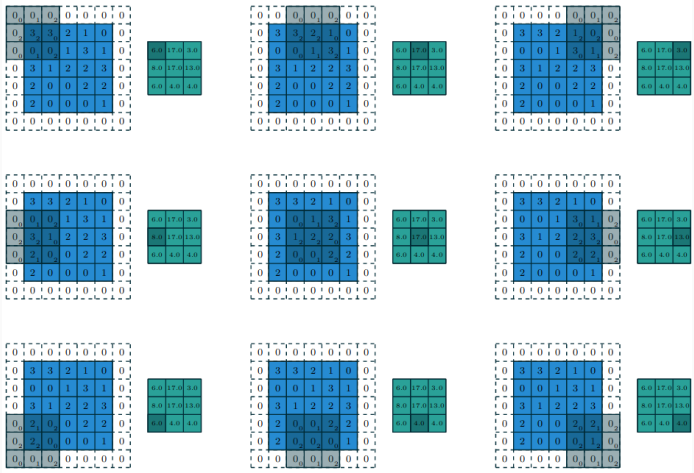
- The output of this convolution is a 3×3 feature map.
- This is a problem if we want our output map to remain the same size.
- Each convolution will reduce the size of the input.
- If we were to stack multiple convolutional layers, the output would eventually be too small to be useful.

Padding

If we want our output to be same size as the input, we can add padding to the original input image before convolving it.

This is often known as a **full** convolution.

Padding



A full convolution (Dumoulin et al.)

Stride

- We have only looked at convolutions which step by 1 unit as they shift over the image.
- We can control the size of this step, or **stride**, to produce different outcomes.
- Picking a non-unit stride has a number of effects on the features that are learned in a convolutional neural network, for example.

Stride

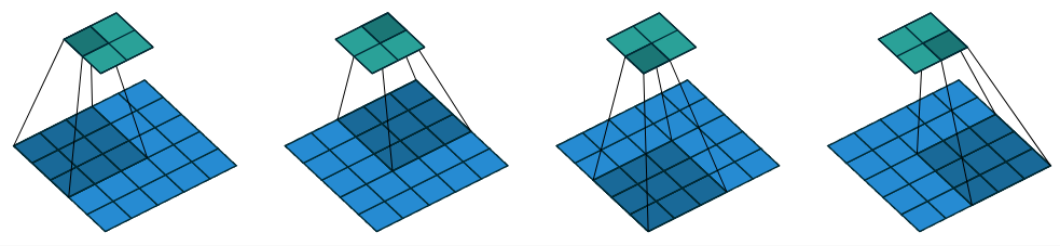
- **Dimensionality reduction**: Skipping over pixels reduces the size of the output feature map. This provides another way of downsampling the input.
- **Less computation**: Fewer computations are required to produce the output feature map.
- **Increased field of view**: A larger stride increases the field of view of the kernel, leading to larger receptive fields in deeper layers.

Stride

Given an input of size $m \times m$ and a kernel of size $n \times n$, the output size of a convolution with stride s is given by

$$\left\lfloor \frac{m-n}{s} \right\rfloor + 1.$$

Padding



A convolution with stride 2 (Dumoulin et al.)

Kernel Size

- The size of the kernel has a large impact on the features that are learned, or pixels that are covered.
- A larger kernel will have a larger receptive field.
- This means that the kernel will be able to capture more information about the input.

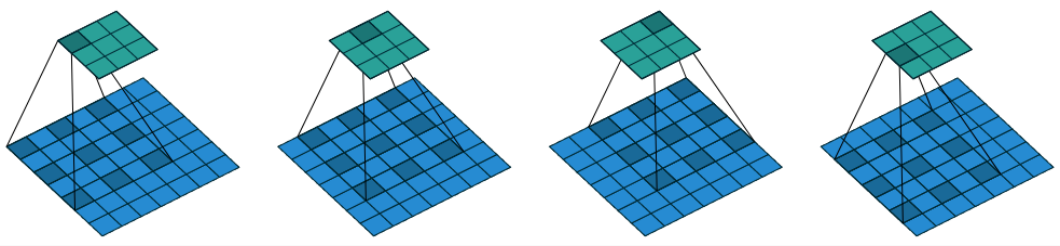
Kernel Size

- This comes at the cost of increased computation.
- Common kernel sizes in most CNNs are 3×3, 5×5, and 7×7.
- Image processing often uses larger kernels.
- It is also convenient to pick an odd kernel size so that the kernel has a center pixel.

Dilation

- Around 2015, a research trend for CNNs was to find a way to increase the receptive field without adding more parameters.
- The result is a **dilated** convolution.
- The output of a dilated convolution is computed by skipping over pixels in the input.

Padding



A dilated convolution (Dumoulin et al.)

Dilation

The output size is computed as

$$\left\lfloor \frac{m + 2p - n - (n-1)(d-1)}{s} \right\rfloor + 1,$$

where p is the amount of padding and d is the dilation factor.

Convolution in CUDA

Implementing a Convolutional Kernel

- It is straightforward to write the convolution operation in CUDA C++.
- Each thread will compute the value for a single output pixel using the filter.
- We already implemented something very similar with the blurring kernel.

Implementing a Convolutional Kernel

The kernel itself should accept the following arguments:

- input image
- output image
- kernel
- radius of the kernel
- width of the output image
- height of the output image

Implementing a Convolutional Kernel

A more robust implementation would consider things like padding, stride, dilation, and whether or not a valid or full convolution is desired.

Focus on the simplest case: a valid convolution with a stride of 1 and no padding or dilation.

Implementing a Convolutional Kernel

```
__global__ void conv2D(float *input, float *filter, float *output,
                       int r, int width, int height) {
    int outCol = blockIdx.x * blockDim.x + threadIdx.x;
    int outRow = blockIdx.y * blockDim.y + threadIdx.y;
    float sum = 0.0f;
    for (int row = 0; row < 2*r+1; row++) {
        for (int col = 0; col < 2*r+1; col++) {
            int inRow = outRow + row - r;
            int inCol = outCol + col - r;
            if (inRow >= 0 && inRow < height && inCol >= 0 && inCol < width) {
                sum += input[inRow * width + inCol] *
                       filter[row * (2*r+1) + col];
            }
        }
    }
```

Implementing a Convolutional Kernel

- The input and output sizes are assumed to be the same.
- There is a boundary check in the inner-most loop to account for pixels for which a convolution cannot be computed.
- Based on this check, we can see that this kernel is performing a *valid* convolution.
- The extra $2r$ pixels on each side of the output are skipped.
- This presents a computational problem in the form of control divergence.

Implementing a Convolutional Kernel

Recall that all threads in a warp must execute the same instruction.

If the boundary check fails for some threads, they will still execute the instructions in the loop, but will not contribute to the output.

This is a waste of resources.

Implementing a Convolutional Kernel

It is also a waste of resources in terms of memory used for the output.

If we already know that we want to perform a valid convolution, we can allocate the output image to be the appropriate size before calling it.

Implementing a Convolutional Kernel

```
__global__ void conv2D(float *input, float *filter, float *output,
                       int r, int width, int height) {
    int outCol = blockIdx.x * blockDim.x + threadIdx.x;
    int outRow = blockIdx.y * blockDim.y + threadIdx.y;
    float sum = 0.0f;
    for (int row = 0; row < 2*r+1; row++) {
        for (int col = 0; col < 2*r+1; col++) {
            int inRow = outRow + row;
            int inCol = outCol + col;
            sum += input[inRow * width + inCol] * filter[row * (2*r+1) + col];
        }
    }
    output[outRow * width + outCol] = sum;
}
```

Constant Memory and Caching

Constant Memory and Caching

There is a much larger issue present in both versions of this kernel in terms of memory bandwidth.

Similar to the matrix multiplication kernel, this kernel can benefit from tiling. **But...**

Constant Memory and Caching

There is a much larger issue present in both versions of this kernel in terms of memory bandwidth.

Similar to the matrix multiplication kernel, this kernel can benefit from tiling. **But...**

The same filter is accessed by every single thread.

Constant Memory and Caching

This filter does not change for the entire duration of the kernel.

This means that we are wasting memory bandwidth by having every thread access the same filter.

Constant Memory and Caching

- Given its relatively small size, this kernel is a perfect candidate for constant memory.
- This is a special type of memory that is cached on the GPU.
- It is read-only and has a limited size, but it is much faster than global memory.
- We can write to the devices constant memory from the host code.

Constant Memory and Caching

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

This informs the compiler to allocate a 2D array of floats in constant memory.

Constant Memory and Caching

The data can be copied to the device using the `cudaMemcpyToSymbol` function.

```
cudaMemcpyToSymbol(F_d, filter_h, size);
```

Constant Memory and Caching

At this point, `F_d` is accessible from the kernel as a global variable.

There is no need to pass it as an argument.

Constant Memory and Caching

```
__global__ void conv2D(float *input, float *output,
                       int r, int width, int height) {
    int outCol = blockIdx.x * blockDim.x + threadIdx.x;
    int outRow = blockIdx.y * blockDim.y + threadIdx.y;
    float sum = 0.0f;
    for (int row = 0; row < 2*r+1; row++) {
        for (int col = 0; col < 2*r+1; col++) {
            int inRow = outRow + row;
            int inCol = outCol + col;
            sum += input[inRow * width + inCol] * F_d[row][col];
        }
    }
    output[outRow * width + outCol] = sum;
}
```

Constant Memory and Caching

Implementation Note

If you organize your files such that the kernel is in a separate file from the host code, you will need to declare the constant variable in the kernel file as well.

Constant Memory and Caching

- Constant memory variables are stored in DRAM with global memory.
- The CUDA runtime will cache them since it knows they will not be modified.
- Processors use caches to reduce the latency of memory accesses by keeping frequently used data in a small, fast memory that is often located directly on the chip.

Constant Memory and Caching

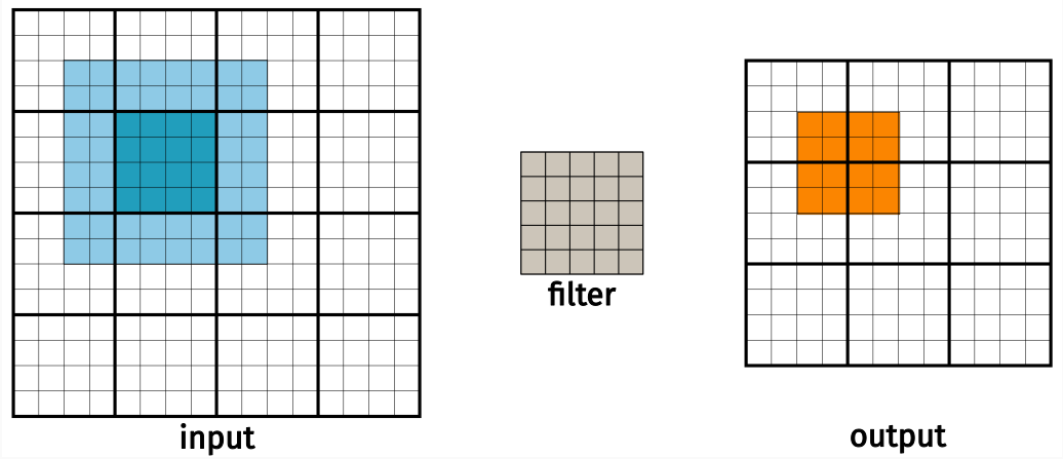
- This type of *constant cache* is preferable to one that would support high-throughput writes in terms of chip design.
- It would require specialized hardware to support both which would increase the cost of the chip.

Tiled Convolutions

Tiled Convolutions

- The convolutional kernel still makes many accesses to DRAM.
- We can tile the input image to reduce the number of accesses.
- Similar to that example, we will use a 4×4 tile size.
- If the input is a 16×16 image and we apply a kernel with radius $r = 2$, the output image under a valid convolution will be 12×12.

Tiled Convolutions



Left: Input image and tiling. Middle: Filter. Right: Output image.

Tiled Convolutions

- The parallel solution to this problem will follow the tiled approach used for matrix multiplication.
- One key difference in this case is that the input tile size will be larger than the output tile size.
- This size difference would further be complicated if we left the kernel size as a parameter.

Tiled Convolutions

Following the design presented by Hwu et al. in Chapter 7, there are two immediate approaches to this problem based on the tile size.

The first is to choose a block size that matches the size of the input tiles.

The benefit to this approach is that each thread can load a single input element into shared memory.

Tiled Convolutions

The drawback is that some of the threads will be disabled when computing the output value since the output tile is smaller.

This is a form of control divergence and will result in wasted resources.

Tiled Convolutions

The first part of a kernel that follows this approach would read the assigned input tile into shared memory.

```
int col = blockIdx.x * OUT_TILE_DIM + threadIdx.x;
int row = blockIdx.y * OUT_TILE_DIM + threadIdx.y;
if (row < height && col < width) {
    inputTile[threadIdx.y][threadIdx.x] = input[row * width + col];
} else {
    inputTile[threadIdx.y][threadIdx.x] = 0.0f;
}
__syncthreads();
```

Tiled Convolutions

Then the output operation is applied as normal.

```
float sum = 0.0f;
for (int fRow = 0; fRow < 2*FILTER_RADIUS+1; fRow++) {
    for (int fCol = 0; fCol < 2*FILTER_RADIUS+1; fCol++) {
        sum += inputTile[tileRow + fRow][tileCol + fCol] *
               kFilter_d[fRow][fCol];
    }
}
output[row * (width - 2 * FILTER_RADIUS) + col] = sum;
```

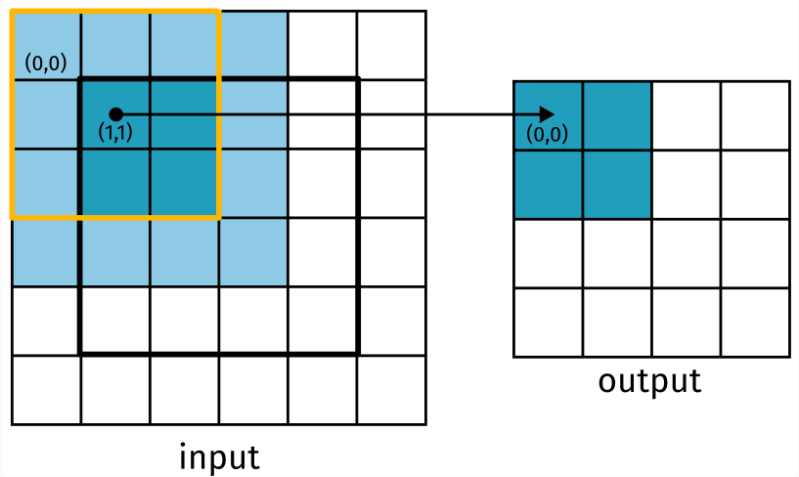
Tiled Convolutions

- The first phase of this kernel collaboratively loads data into a shared memory space.
- This kernel assumes a convenient indexing scheme where the row and column will always be $>= 0$.

Tiled Convolutions

- We could adopt a scheme that centers the convolution on the center point of the kernel by allowing for negative indices.
- It would be necessary to check if the row and column are less than 0.
- This implementation only needs to verify that the row and column are within the given image size.

Tiled Convolutions



The active thread for computing the output tile.

Tiled Convolutions

- When it comes to computing the output, not every thread will contribute.
- Only the threads corresponding to the darker blue on the left contribute to the output calculation.
- Since this one block computes 4 output values, the next block should start 2 units to the right of this one.

Performance Analysis

The purpose of this approach was to increase the ratio of arithmetic operations to global memory accesses.

For the threads that compute an output tile, there is one multiplication and one addition which yields $\texttt{OUT\_TILE\_DIM}^2 * (2 * \texttt{FILTER\_RADIUS} + 1)^2 * 2$ operations total.

Performance Analysis

Each thread in the input tile loads a single `float` value for a total of $\texttt{IN\_TILE\_DIM}^2 * 4$ bytes. For our small example above, this gives

$$\frac{2^2 * 3^2 * 2}{4^2 * 4} = 1.125 \text{ Ops/byte.}$$

Performance Analysis

- In a more realistic example, we would maximize our input tile size to take advantage of the available threads on the device.
- The maximum number of supported threads is 1024.
- The resulting operations per byte under this tile size is $\frac{30^2 * 3^2 * 2}{32^2 * 4} = 3.955$ Ops/byte.
- This ratio increases with the size of the filter.

Caching the Halo Cells

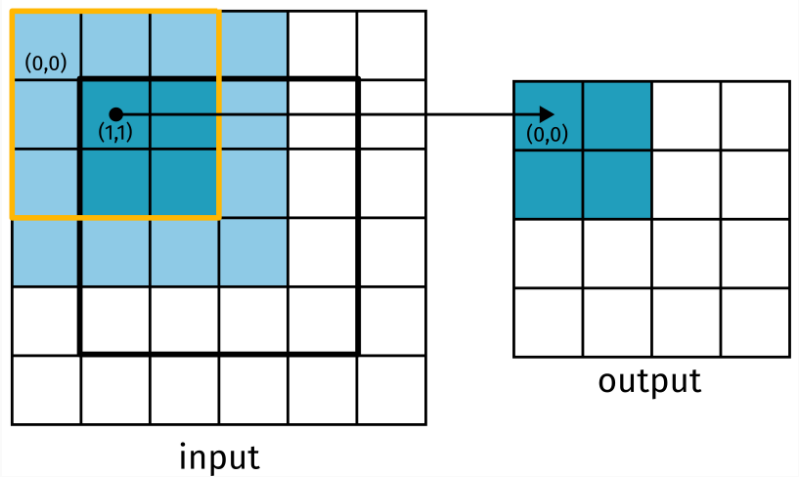
Caching the Halo Cells

The size of the input tile compared to the output tile means that there were some threads that did not contribute to the output computation.

These are the threads managing the lightly shaded cells.

We will refer to these as *halo cells*.

Caching the Halo Cells



The active thread for computing the output tile.

Caching the Halo Cells

This implementation is going to take advantage of the caching behavior in the chip itself.

Values that have been recently used are more likely to already be in L2 cache.

Caching the Halo Cells

This is a safe assumption since the neighboring blocks will have loaded these values into shared memory.

This means that the input and output tile sizes can be the same; there is no need to waste any threads in the block.

Caching the Halo Cells

```
// If this value is in shared memory, access it there
if (tileCol + fCol >= 0 &&
    tileCol + fCol < IN_TILE_DIM &&
    tileRow + fRow >= 0 &&
    tileRow + fRow < IN_TILE_DIM) {
    sum += inputTile[tileRow + fRow][tileCol + fCol] * kFilter_d[fRow][fCol];
} else {
    // Otherwise, access it from global memory
    sum += input[(row + fRow) * width + (col + fCol)] * kFilter_d[fRow][fCol];
}
```