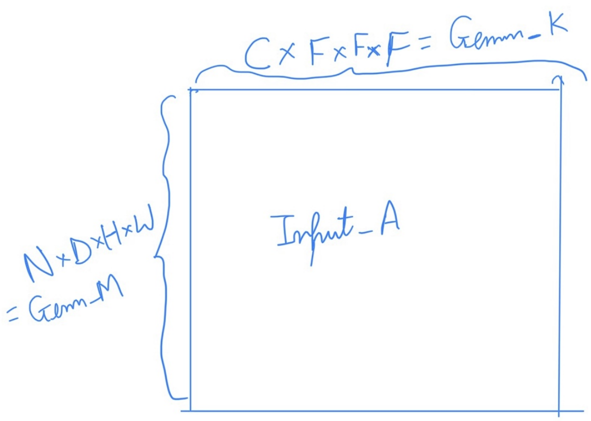
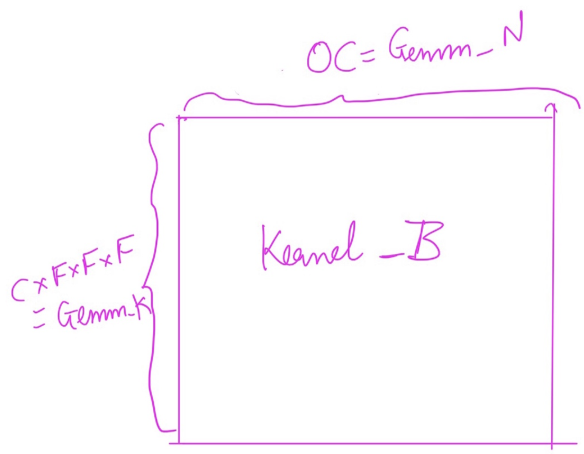
# CUTLASS based 3D conv

https://github.com/NVIDIA/cutlass/blob/master/media/docs/implicit_gemm_convolution.md

# Adapting optimization to 3D CNN:

## Gemm formulation




**Initial convolution dimensions:**

Output = N X D X H X W X OC

Kernel = C X F X F X F X OC

Data layout = NDHWC

**Corresponding gemm dimensions:**

A Input: GEMM_M X GEMM_K

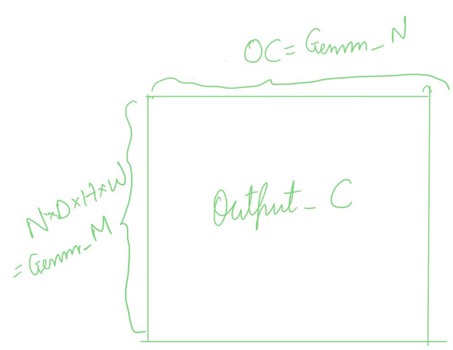
B Kernel: GEMM_K X GEMM_N

C Output = GEMM_M X GEMM_N

GEMM_M = N*D*H*W*C

GEMM_K = C*F*F*F

GEMM_N = OC



## Implicit gemm indexing

```
for gemm_i in 0:batchsize*outdepth*outheight*outwidth # GEMM_M loop
  for gemm_j in 0:outchannels # GEMM_N loop

    n = gemm_i//tmp_dhw  # Indexing the batch size
    nopq_residual = gemm_i % tmp_dhw
    o = nopq_residual//tmp_hw  # Indexing the output depth
    opq_residual = nopq_residual%tmp_hw
    p = opq_residual//outwidth # Indexing the output height.
    q = opq_residual%outwidth # Indexing the output width

    accum = 0
    for gemm_k in 0:inchannels*kdim*kdim*kdim # GEMM_K loop

      c = gemm_k//tmp_kdim3 # Indexing input, kernel channel
      ctrs_residual = gemm_k%tmp_kdim3
      t = ctrs_residual//tmp_kdim2
      trs_residual = ctrs_residual%tmp_kdim2
      r = trs_residual//kdim
      s = trs_residual%kdim
      d = o + t # Indexing the input depth e(o, t)
      h = p + r # Indexing the input height f(p, r)
      w = q + s # Indexing the input width g(q, s)
      accum = accum + imgemm_input[n,d,h,w,c]*imgemm_kernel[gemm_j,t,r,s,c]

  imgemm_output[n,o,p,q,gemm_j] = accum
```

Each index gemm_i in GEMM_M dimension corresponds to a unique (N,O,P,Q) index of the output tensor.

```
    n = gemm_i//tmp_dhw  # Indexing the batch size
    nopq_residual = gemm_i % tmp_dhw
    o = nopq_residual//tmp_hw  # Indexing the output depth
    opq_residual = nopq_residual%tmp_hw
    p = opq_residual//outwidth # Indexing the output height.
    q = opq_residual%outwidth # Indexing the output width
```

The algorithm partitions the GEMM_K dimension into threadblock tiles.

Assigns each threadblock tile to one filter position (t,r,s) and an interval of channels.

## Mapping to CUTLASS Gemm matrix multiplication loop

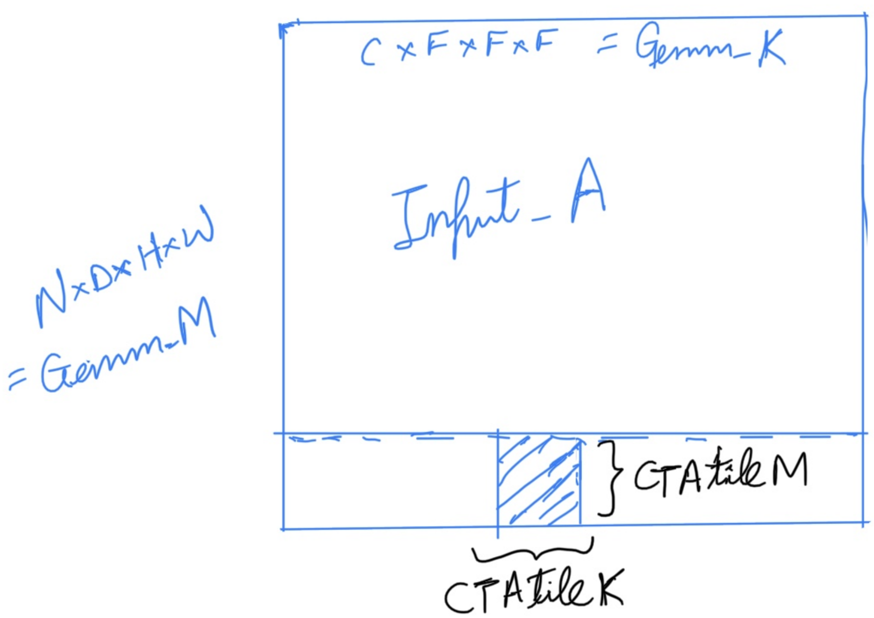
```
#for cta_n in 0:GEMM_N:CTAtileN
for cta_n in 0:outchannels #GEMM_N, gemm_j
  #for cta_m in 0:GEMM_M:CTAtileM
  for cta_m in 0:batchsize*outdepth*outheight*outwidth:CTAtileM #GEMM_M, gemm_i
    #for cta_k in 0:GEMM_K:CTAtileK # Main loop
    for cta_k in 0:inchannels*kdim*kdim*kdim:CTAtileK #GEMM_K, gemm_k

      for warp_n in 0:CTAtileN:WARPtileN
        for warp_m in 0:CTAtileM:WARPtileM
          for warp_k in 0:CTAtileK:WARPtileK # Partitioning the input channels

            for mma_k in 0:WARPtileK
              for mma_n in 0:WARPtileN
                for mma_m in 0:WARPtileM
                  # Execute instruction for accumulating the product
```

One particular iteration of main loop, i.e. when cta_n, cta_m, cta_k are fixed.

The data format for input and kernel is **NDHWC.** So accessing adjacent ‘channel index’ for a fixed depth, height, width is faster both for input and the filter.

## Input computation partitioning

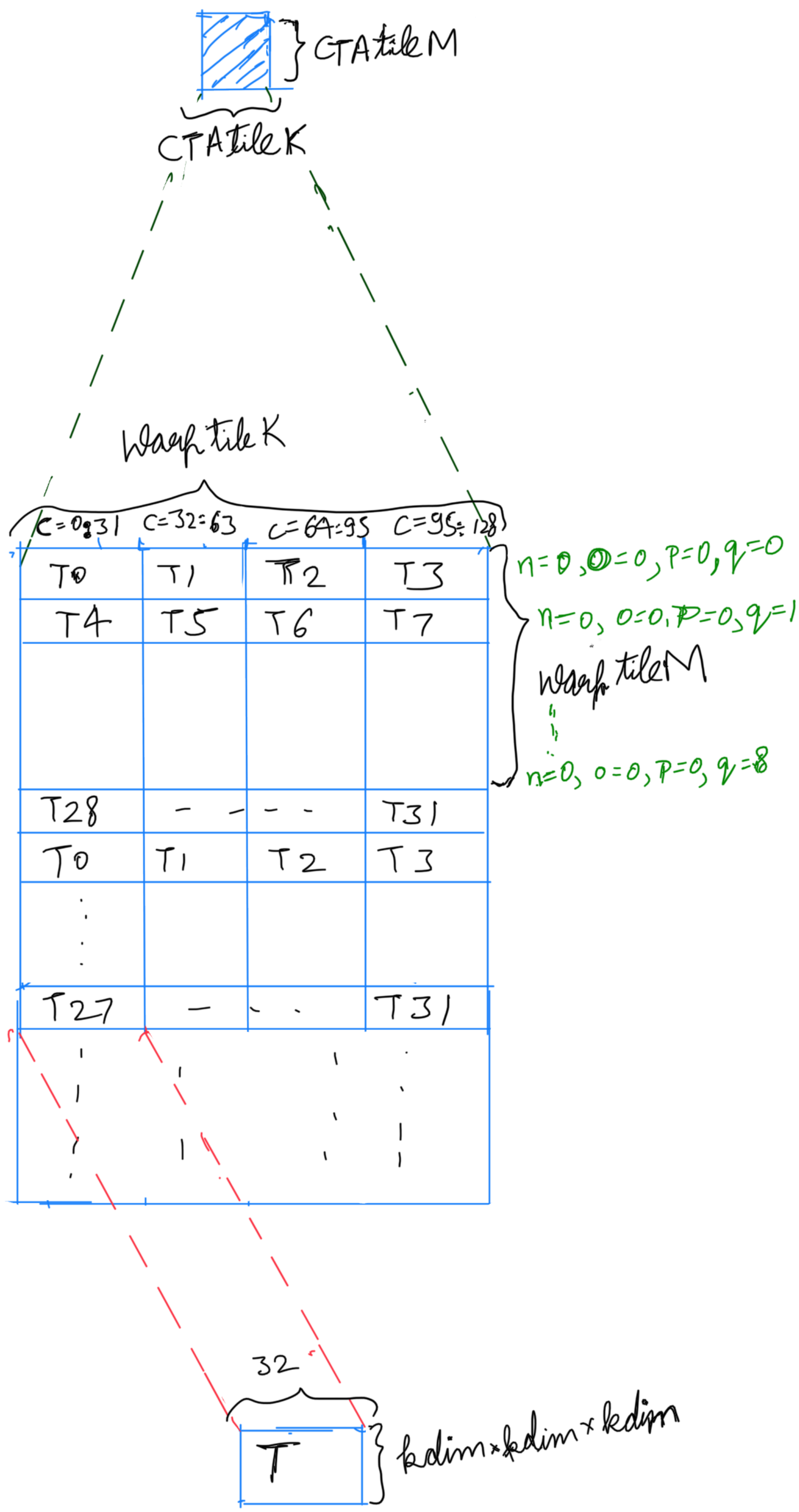
### CTAtileM X CTAtileK



- Gemm_K decides the r, s, t coordinates i.e. the indexing of the kernels.
- Each block has a CTAtileM X CTAtileK size input.
- Because the data layout is NDHWC, adjacent channels are accessed by adjacent threads.

## WarptileK X WarptileM

- Index along the CTAtileM decides which output values are computed.
- r, s, t index decides which CTA tile or block id.
- A range of channels decided = CTAtileK are used in this block
- Assuming CTAtileK = 128

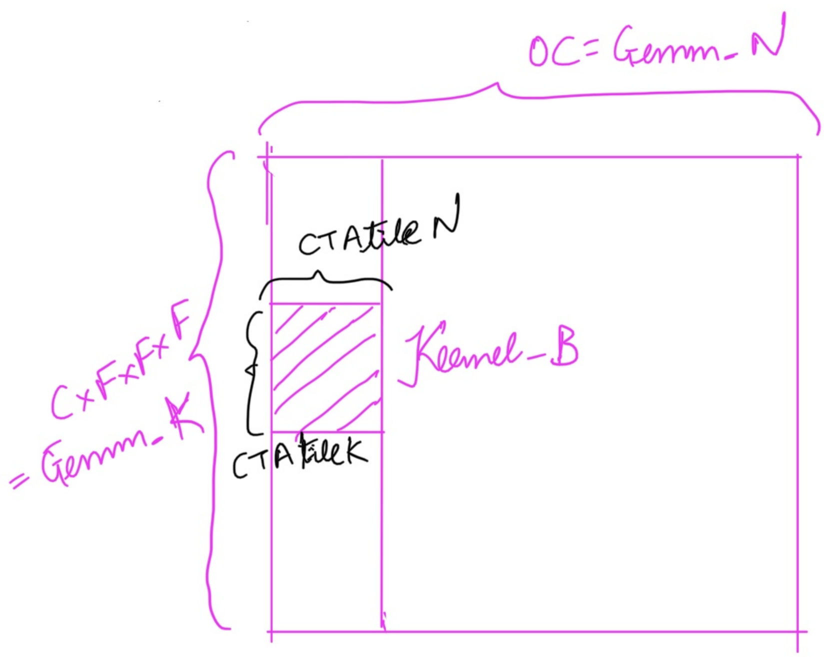

CTA tile M
CTA tile K
Warp tile K
C=0:31
C=32:63
C=64:95
C=95:128
T0
T1
T2
T3
T4
T5
T6
T7
T28
T31
T0
T1
T2
T3
T27
T31
n=0, o=0, p=0, q=0
n=0, o=0, p=0, q=1
Warp tile M
n=0, o=0, p=0, q=8
32
T
kdim×kdim×kdim

- 32 threads in 1 warp, so WARPtileK X WarptileM = 32
- all 4 threads in a row work on 1 output value (indexed by oc, o, p, q)

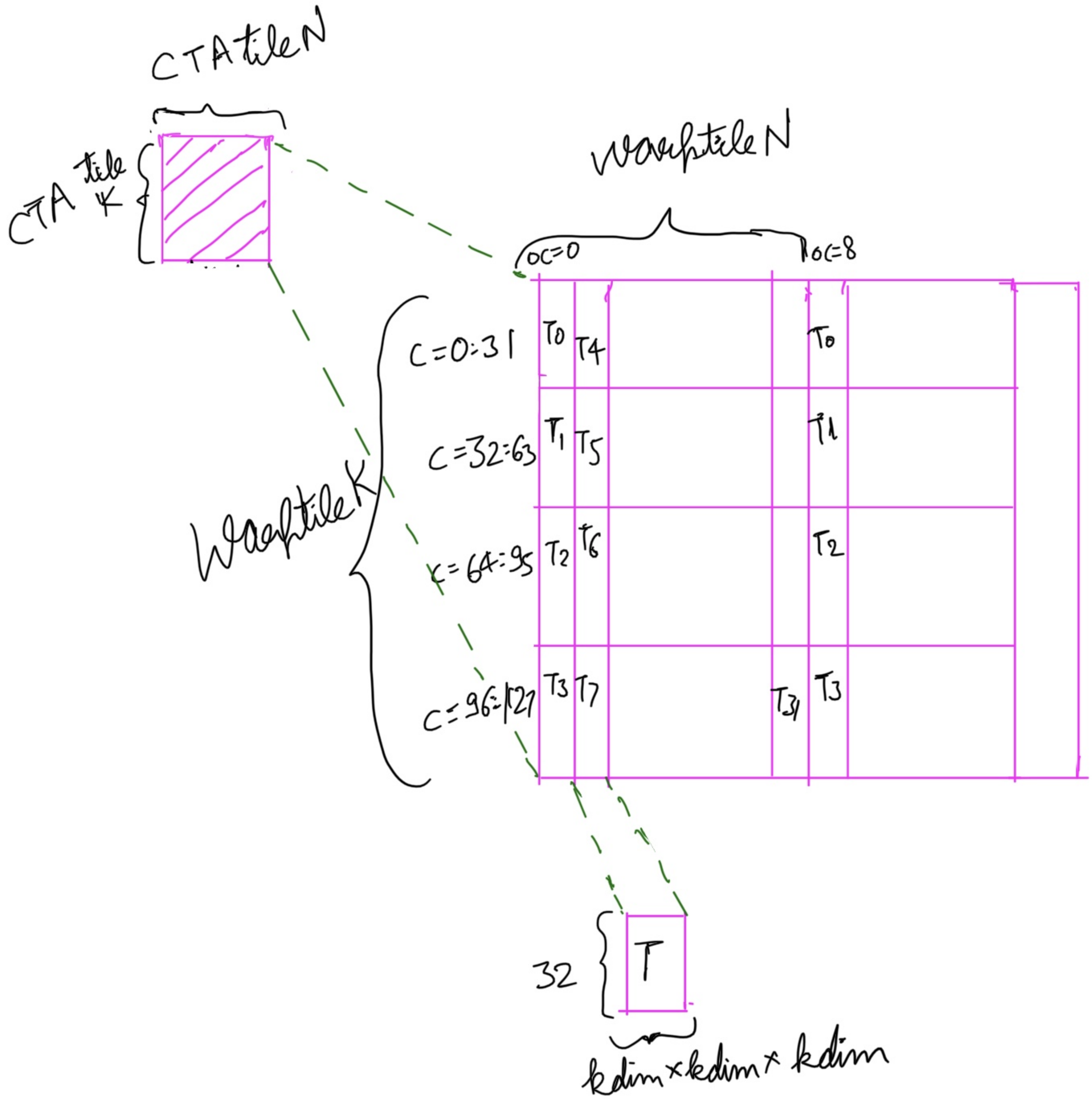
# Filter computation partitioning

## CTAtileK X CTAtileN

- Gemm_N is for indexing output channel
- The CTAtile selected has a fixed r, s, t based on the block ID
- The range of input channels each block uses is decided by CTAtileK
- The range of output channels each block computes is decided by CTAtileN




## WarptileK X WarptileM




- 32 threads in 1 warp, so WARPtileK X WarptileM = 32
- all 4 threads in a column work on 1 output value indexed by (oc, o, p, q)

## Shared memory usage

- The CTAtiles are loaded by each block
- So the CTAtiles corresponding to the input, filter and output should fit inside shared memory

Input = CTAtileM * CTAtileK

Output = CTAtileM * CTAtileN

Kernel = CTAtileK * CTAtileN

Total shared memory needed per block = `[ CTAtileM * CTAtileK + CTAtileM * CTAtileN + CTAtileK * CTAtileN ]`

this should be less than maximum available CUDA shared memory.

## Deciding tile sizes

Goal: maximize the parallelism

Number of blocks launched = `(GEMM_M / CTAtileM) * (GEMM_N / CTAtileN) * (GEMM_K / CTAtileK)`

Use soap analysis ??