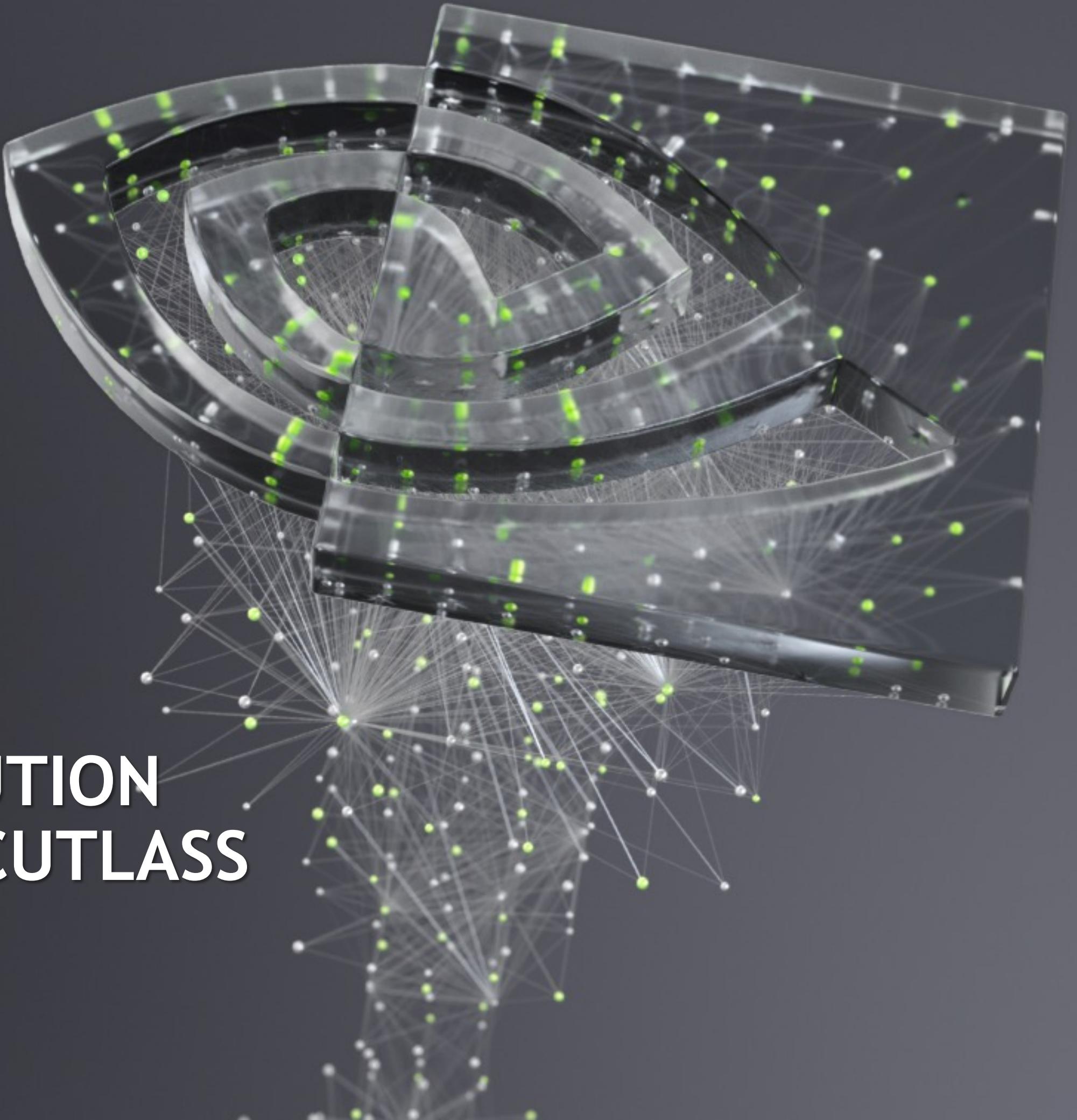


ACCELERATING CONVOLUTION WITH TENSOR CORES IN CUTLASS

Manish Gupta, April 13, 2021



ACKNOWLEDGEMENTS

CUTLASS GitHub Community

35K clones/month, 1.1K stars, and many active users

CUTLASS Team

Andrew Kerr, Haicheng Wu, Manish Gupta, Dustyn Blasig, Duane Merrill, Pradeep Ramani, Vijay Thakkar

Contributors

Cris Ceka, Timothy Costa, Naila Farooqui, Markus Hohnerbach, Alan Kaatz, Wei Liu, Piotr Majcher, Dhiraj Reddy Nallapa, Mathew Nicely, Kyrylo Perelygin, Aniket Shivam, Paul Springer, Pawel Tabaszewski, Chinmay Talegaonkar, John Tran, Jin Wang, Yang Xu, Scott Yokim

Acknowledgements

Olivier Giroux, Mostafa Hagog, Bryce Lelbach, Julien Demouth, Joel McCormack, Aartem Belevich, Peter Han, Timmy Liu, Yang Wang, Nich Zhao, Jack Yang, Vicki Wang, Junkai Wu, Ivan Yin, Aditya Alturi, Shang Zhang, Takuma Yamaguchi, Stephen Jones, Luke Durant, Harun Bayraktar



AGENDA

Overview

CUTLASS 2.4-2.6 and convolution definition

Deep dive implicit GEMM convolutions

Building coherent and complete abstractions

Optimized implementation of convolution abstractions

Precompute invariants

Epilogue Fusion

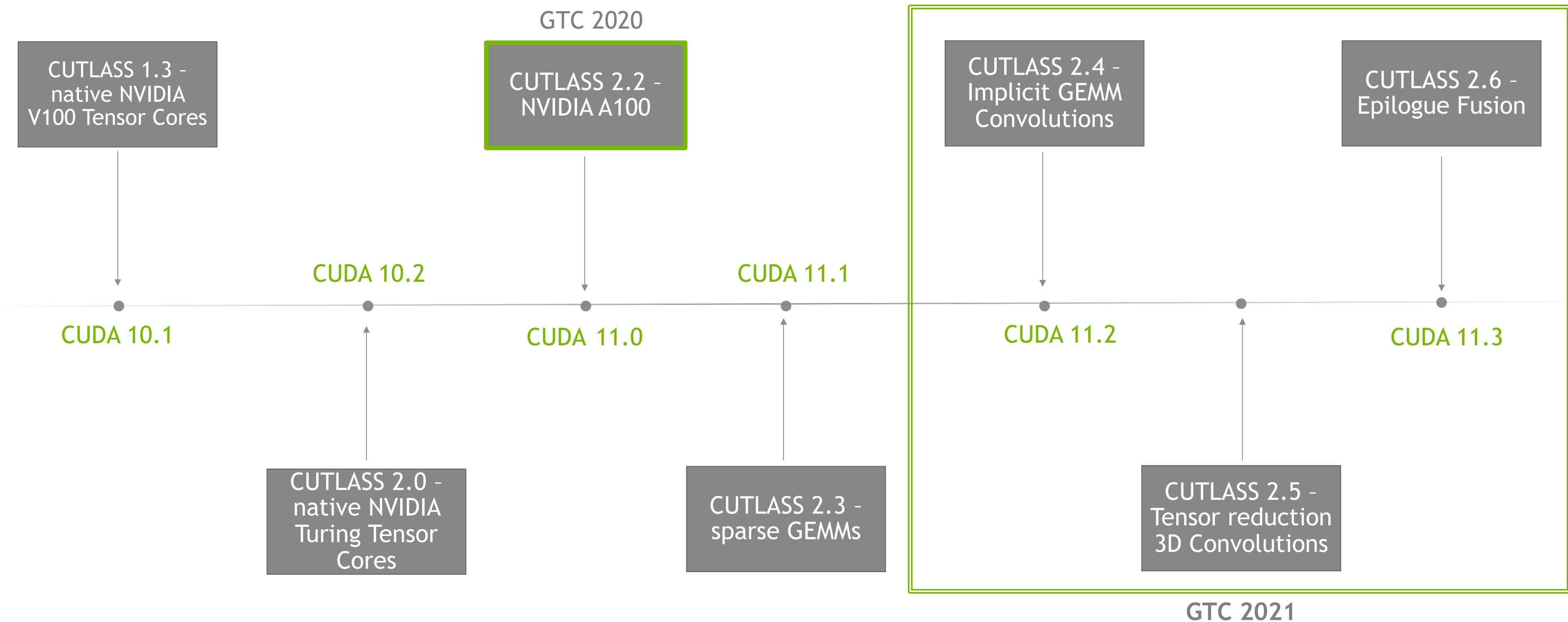
Supported epilogue fusion patterns



OVERVIEW

CUTLASS

CUDA C++ Templates for Deep Learning and Linear Algebra



CUTLASS

What's new since we last met?

CUTLASS 2.4 - Nov 2020

- Implicit GEMM Convolutions 2D
 - Forward propagation (Fprop2D), Backward data gradient (Dgrad2D), and Backward weight gradient (Wgrad2D)
 - 4D tensors with NHWC and NCxHWx layouts
 - S4, S8, S32, F16, BF16, TF32, F32, complex<F32>
 - Tensor cores and CUDA cores
 - Ampere, Turing, Volta, Pascal, Maxwell

CUTLASS 2.5 - Feb 2021

- Implicit GEMM Convolutions 3D
 - Forward propagation (Fprop3D), Backward data gradient (Dgrad3D), and Backward weight gradient (Wgrad3D)
 - 5D tensors with NDHWC layouts
- Tensor Reductions
 - m-to-n reductions of tensors with affine layout
 - Custom reduction functors
 - Large tensor support for 2^{63} elements
- Fused Convolution + Convolution example

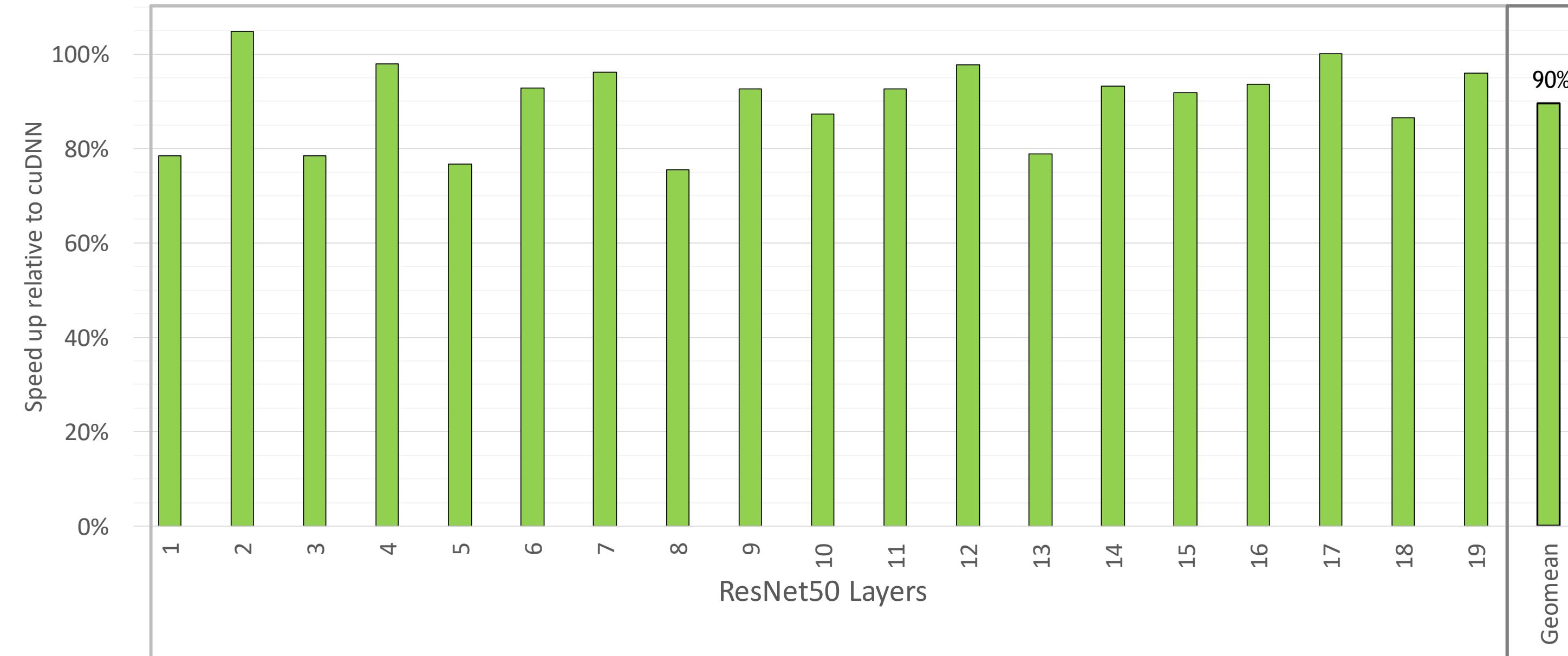
CUTLASS 2.6 - Upcoming release

- Epilogue fusion pattern
 - Broadcast vector over column (Bias add)
 - Partial reduction over column (Batch normalization)

CUTLASS CONVOLUTION PERFORMANCE RELATIVE TO CUDNN

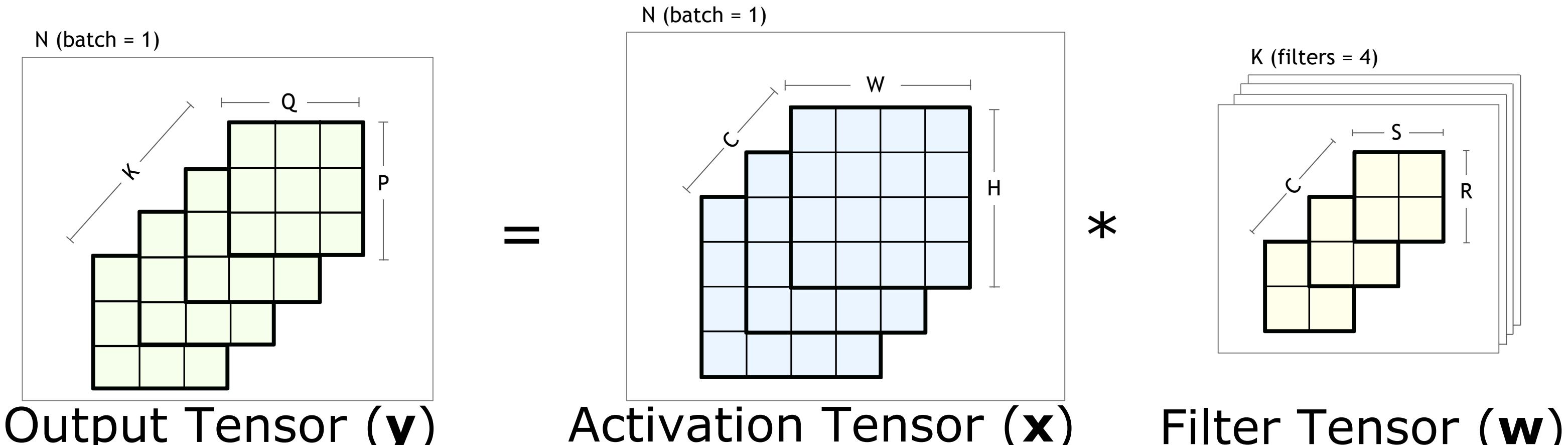
CUTLASS 2.5 - Performance Relative to cuDNN on NVIDIA A100 - CUDA 11.3

Mixed Precision Training ($F16 \leq F16 * F16 + F32$)



2D CONVOLUTION ON 4D TENSORS

Forward Propagation



N = Batch size

P = Height of output tensor

Q = Width of output tensor

K = Number of output channels

N = Batch size

H = Height of input tensor

W = Width of input tensor

C = Number of input channels

K = Number of filters

R = Height of filter tensor

S = Width of filter tensor

C = Number of filter channels

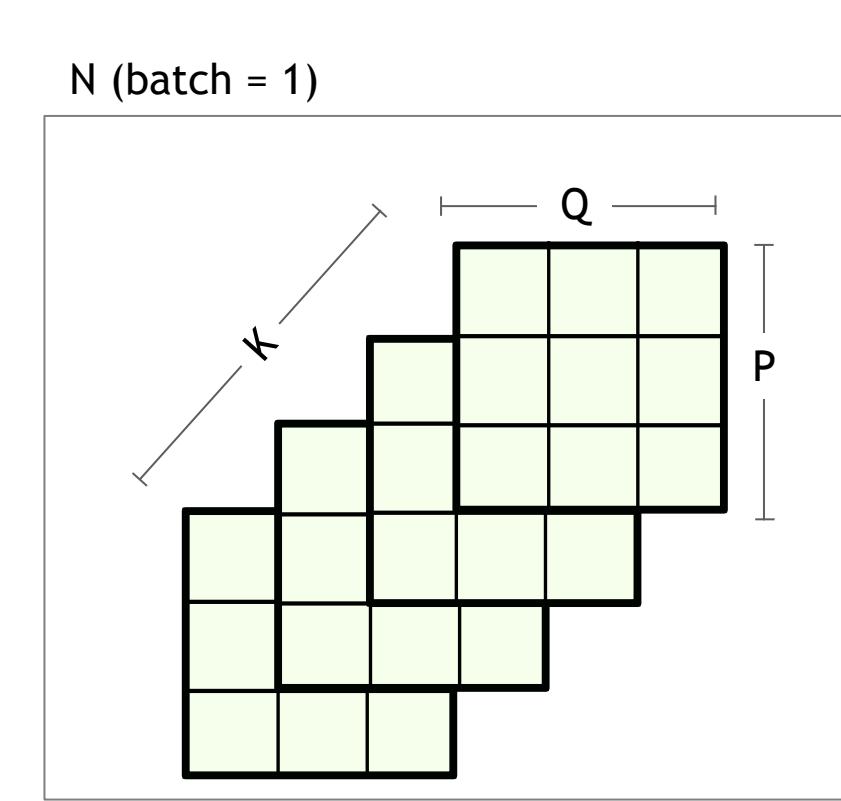
2D CONVOLUTION ON 4D TENSORS - DEFINITION

Forward Propagation

$$y[n, p, q, k] = \sum_{c=0}^{C-1} \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} (x[n, \bar{h}(p, r), \bar{w}(q, s), c] * w[k, r, s, c])$$

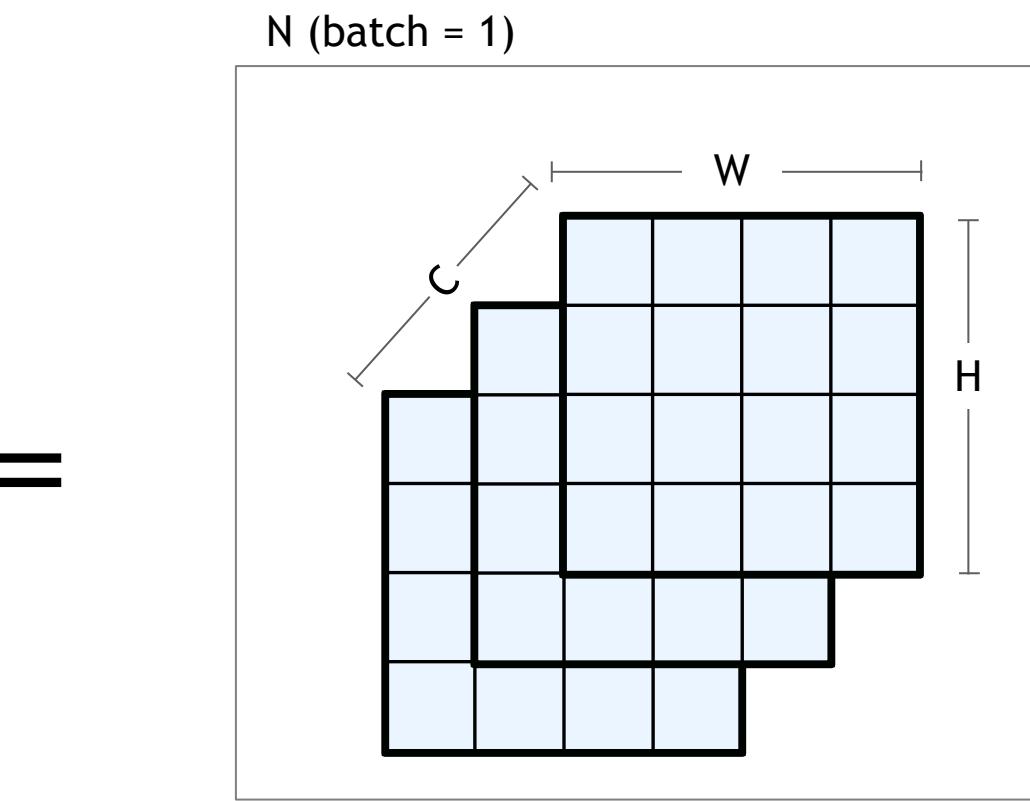
$$\bar{h}(p, r) = p * \text{stride_h} - \text{pad_h} + s * \text{dilation_h}$$

$$\bar{w}(q, s) = q * \text{stride_w} - \text{pad_w} + r * \text{dilation_w}$$



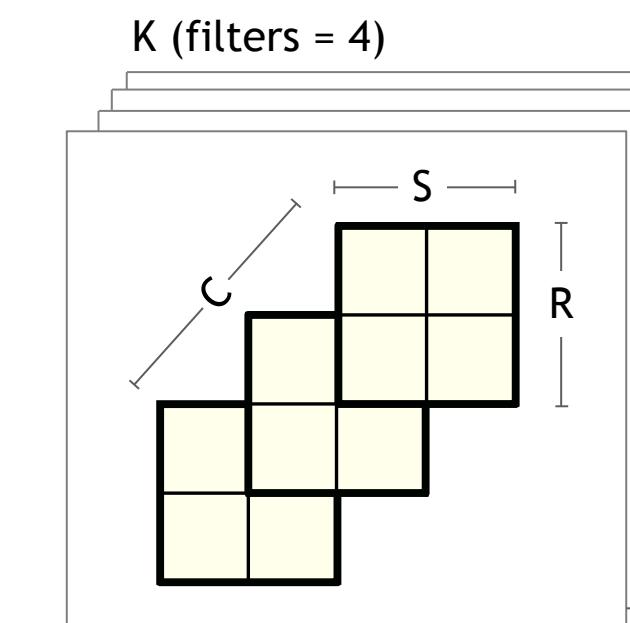
Output Tensor (**y**)

$$NPQK = \{1, 3, 3, 4\}$$



Activation Tensor (**x**)

$$NHWC = \{1, 4, 4, 3\}$$



Filter Tensor (**w**)

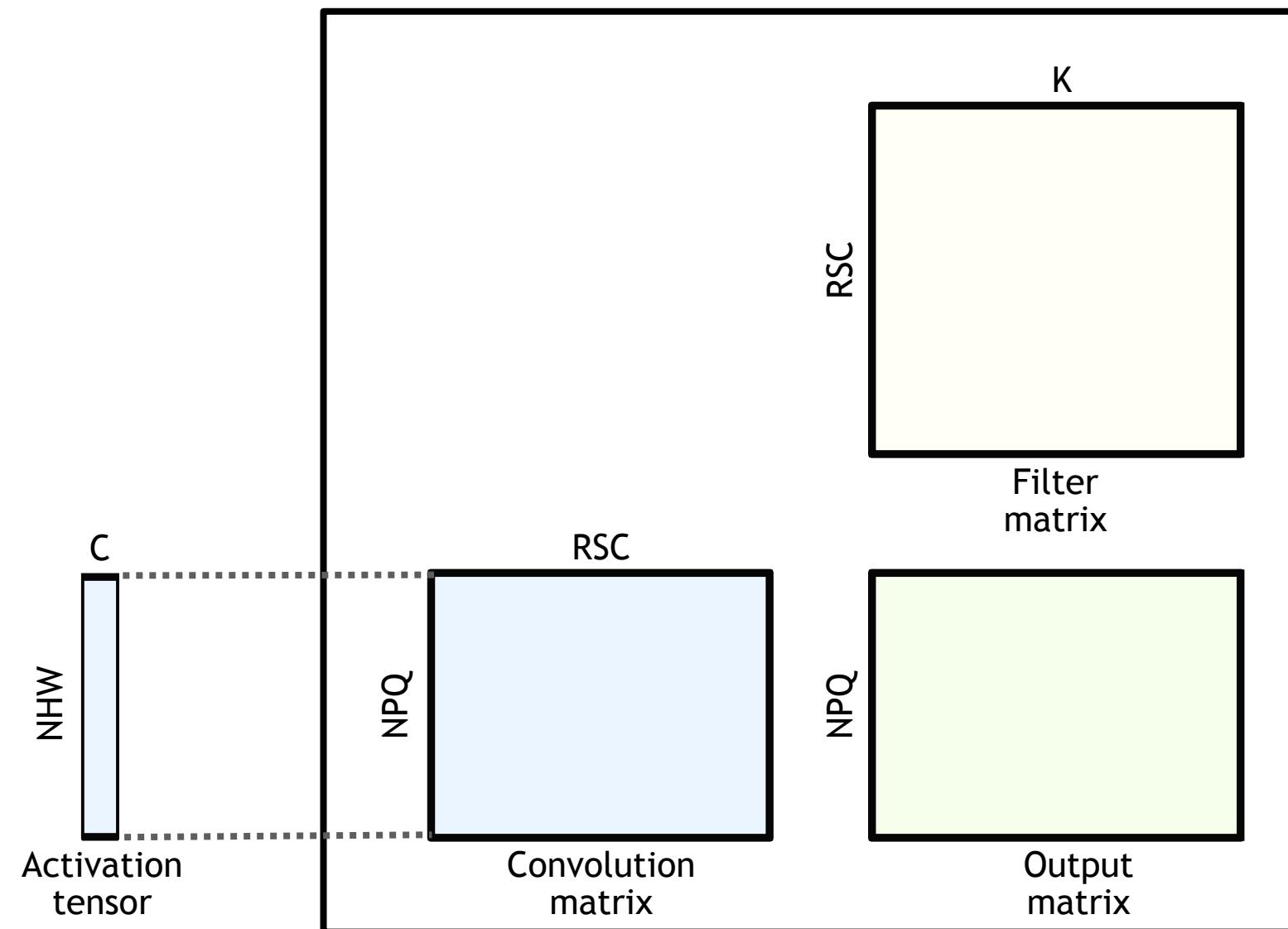
$$KRSC = \{4, 2, 2, 3\}$$

CONVOLUTION TO GEMM MAPPING

Forward Propagation

The convolution operation on 4D tensors can be mapped as matrix-multiply operation on 2D matrices

Convolution	GEMM
$y = \text{CONV}(x, w)$	$C = \text{GEMM}(A, B)$
$x[N, H, W, C]$: 4D activation tensor	→ $A[NPQ, RSC]$: 2D convolution matrix
$w[K, R, S, C]$: 4D filter tensor	→ $B[RSC, K]$: 2D filter matrix
$y[N, P, Q, K]$: 4D output tensor	→ $C[NPQ, K]$: 2D output matrix



CONVOLUTION TO GEMM MAPPING - FILTER MATRIX

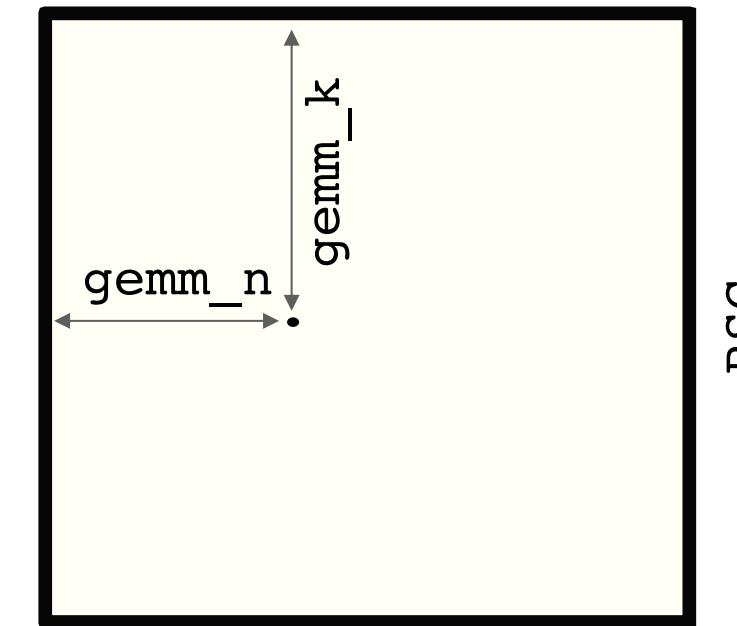
Filter matrix (gemm_k, gemm_n) —> Filter tensor (k, r, s, c)

k = gemm_n
crs_residue = gemm_k / c
r = crs_residue / s
s = crs_residue % s
c = gemm_k % c

GEMM M-by-N-by-K dimensions

GEMM-M = NPQ
GEMM-N = K
GEMM-K = RSC

Filter matrix



RSC

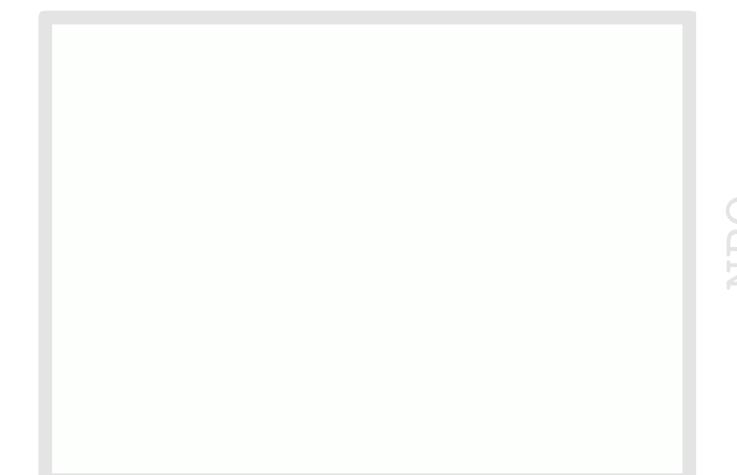
RSC

K

NPQ

NPQ

Convolution matrix



Output matrix

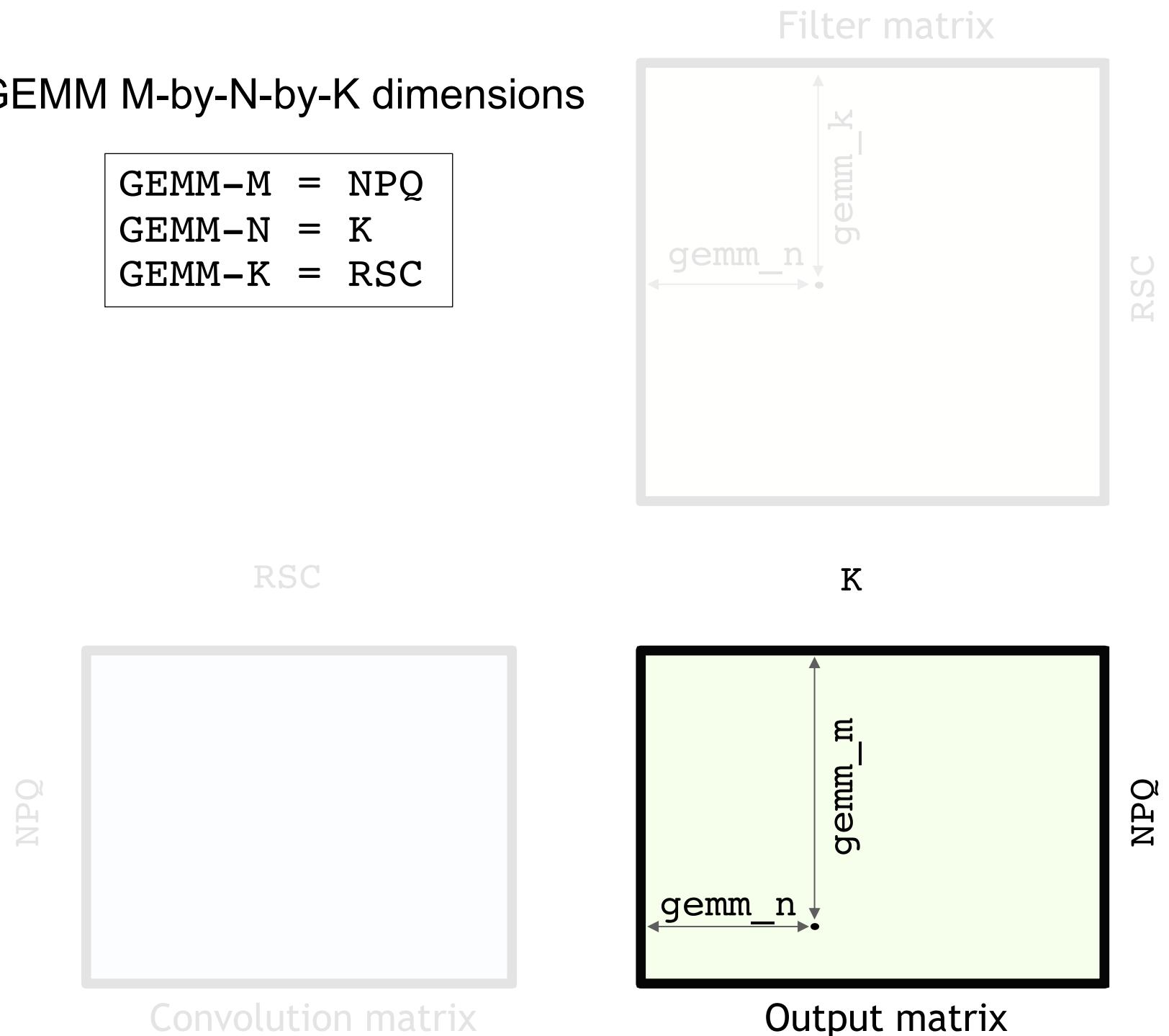
CONVOLUTION TO GEMM MAPPING - OUTPUT MATRIX

Output matrix (gemm_m, gemm_n) → Output tensor (n, p, q, k)

n = gemm_m / (PQ)
ue = gemm_m % (PQ)
p = npq_residue / Q
q = crs_residue % Q
k = gemm_n

GEMM M-by-N-by-K dimensions

GEMM-M	=	NPC
GEMM-N	=	K
GEMM-K	=	RSC



CONVOLUTION TO GEMM MAPPING - CONVOLUTION MATRIX

Convolution matrix (gemm_m, gemm_k) \longrightarrow Activation tensor (n, \bar{h} , \bar{w} , c)

```
n = gemm_m / (PQ)  
npq_residue = gemm_m % (PQ)  
p = npq_residue / Q  
q = crs_residue % Q
```

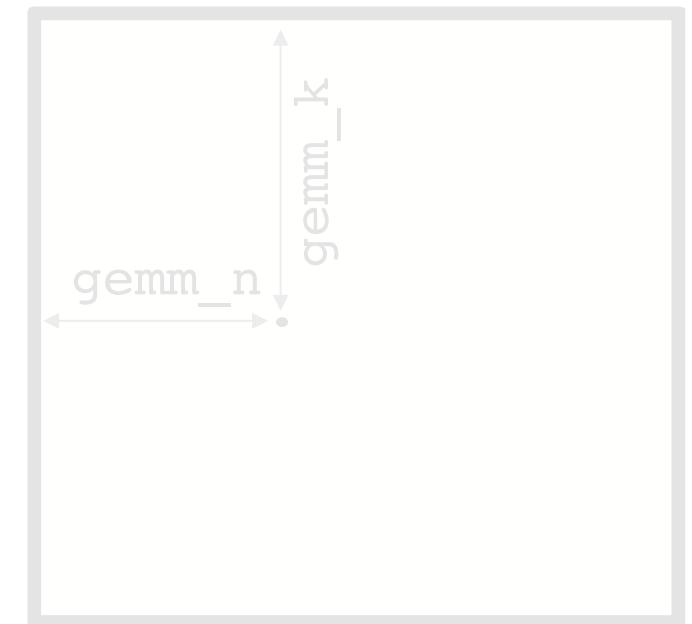
$$\begin{aligned}\bar{h}(p, r) &= p * \text{stride}_h - \text{pad}_h + r * \text{dilation}_h \\ \bar{w}(q, s) &= q * \text{stride}_w - \text{pad}_w + s * \text{dilation}_w\end{aligned}$$

$$c = \text{gemm}_k \% C$$

GEMM M-by-N-by-K dimensions

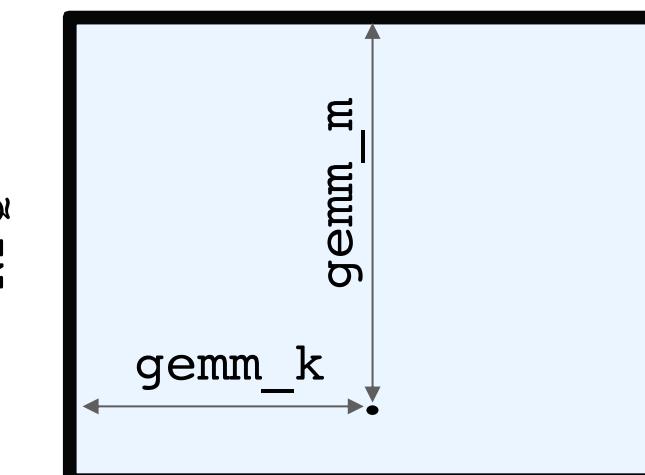
$$\begin{aligned}\text{GEMM-M} &= NPQ \\ \text{GEMM-N} &= K \\ \text{GEMM-K} &= RSC\end{aligned}$$

Filter matrix



RSC

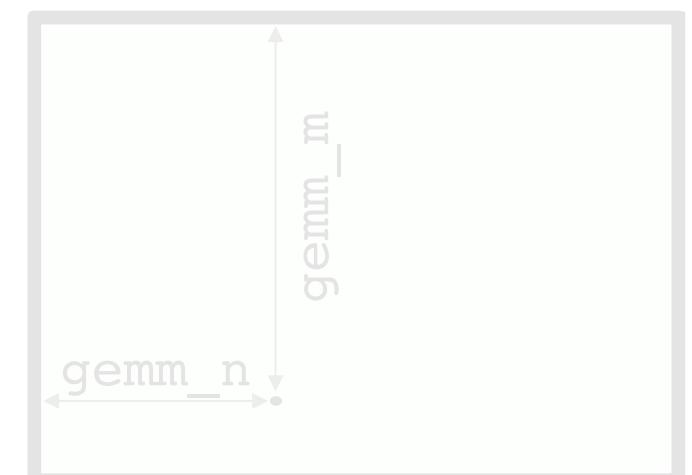
NPQ



Convolution matrix

RSC

NPQ



Output matrix

GEMM TRIPLE NEST LOOP

```
int GEMM_M = N * P * Q;
int GEMM_N = K;
int GEMM_K = R * S * C;

for (int gemm_m = 0; gemm_m < GEMM_M; ++gemm_m) {
    for (int gemm_n = 0; gemm_n < GEMM_N; ++gemm_n) {

        int n = gemm_m / (PQ);
        int npq_residual = gemm_m % (PQ);
        int p = npq_residual / Q;
        int q = npq_residual % Q;

        Accumulator accum = 0;
        for (int gemm_k = 0; gemm_k < GEMM_K; ++gemm_k) {

            int k = gemm_n;
            int crs_residual = gemm_k / C;
            int r = crs_residual / S;
            int s = crs_residual % S;
            int c = gemm_k % C;

            int h = h_bar(p, r);
            int w = w_bar(q, s);

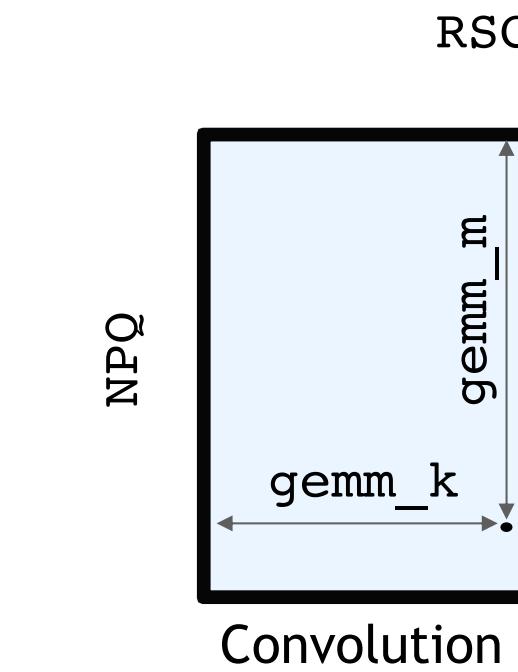
            ElementA a = activation_tensor.at({n, h, w, c});
            ElementB b = filter_tensor.at({k, r, s, c});
            accum += a * b;
        }

        C[gemm_m * K + gemm_n] = accum;
    }
}
```

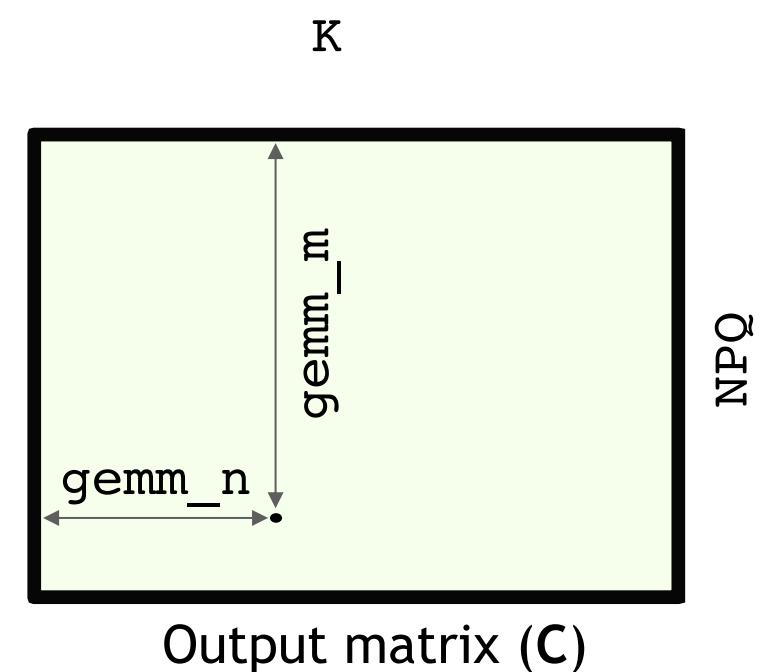
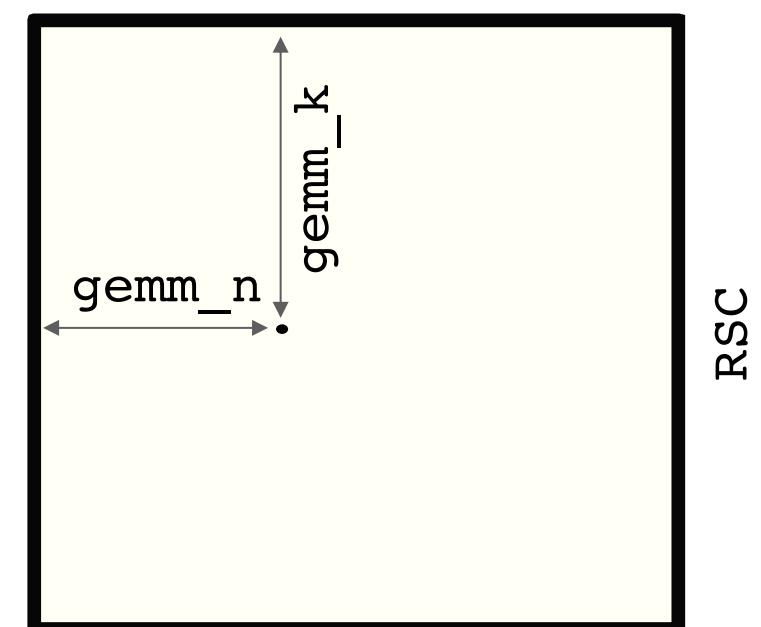
CONVOLUTION MAPPED TO GEMM

GEMM M-by-N-by-K dimensions

GEMM-M = NPQ
GEMM-N = K
GEMM-K = RSC



Filter matrix (B)



GEMM TRIPLE NEST LOOP

```

int GEMM_M = N * P * Q;
int GEMM_N = K;
int GEMM_K = R * S * C;

for (int gemm_m = 0; gemm_m < GEMM_M; ++gemm_m) {
    for (int gemm_n = 0; gemm_n < GEMM_N; ++gemm_n) {

        int n = gemm_m / (PQ);
        int npq_residual = gemm_m % (PQ);
        int p = npq_residual / Q;
        int q = npq_residual % Q;

        Accumulator accum = 0;
        for (int gemm_k = 0; gemm_k < GEMM_K; ++gemm_k) {

            int k = gemm_n;
            int crs_residual = gemm_k / C;
            int r = crs_residual / S;
            int s = crs_residual % S;
            int c = gemm_k % C;

            int h = h_bar(p, r);
            int w = w_bar(q, s);

            ElementA a = activation_tensor.at({n, h, w, c});
            ElementB b = filter_tensor.at({k, r, s, c});
            accum += a * b;
        }
    }

    C[gemm_m * K + gemm_n] = accum;
}
}

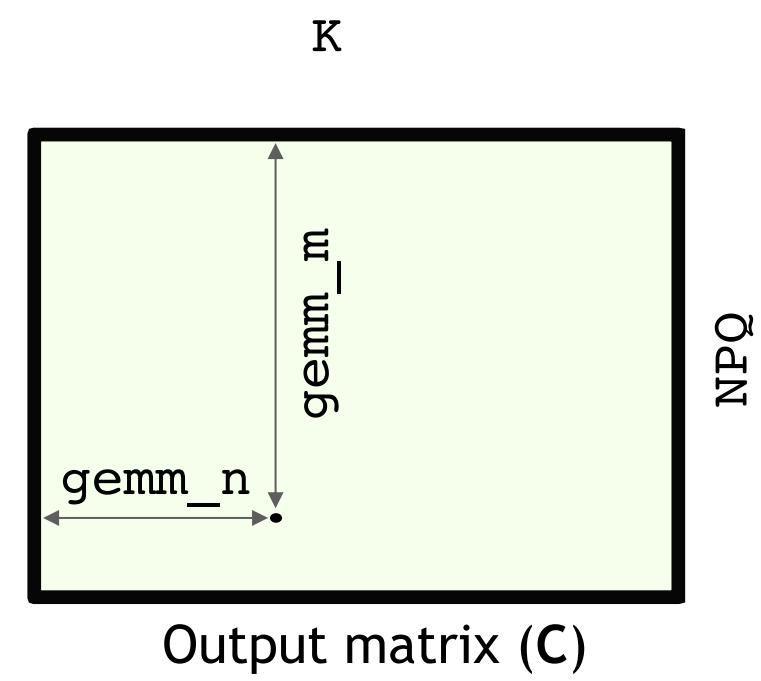
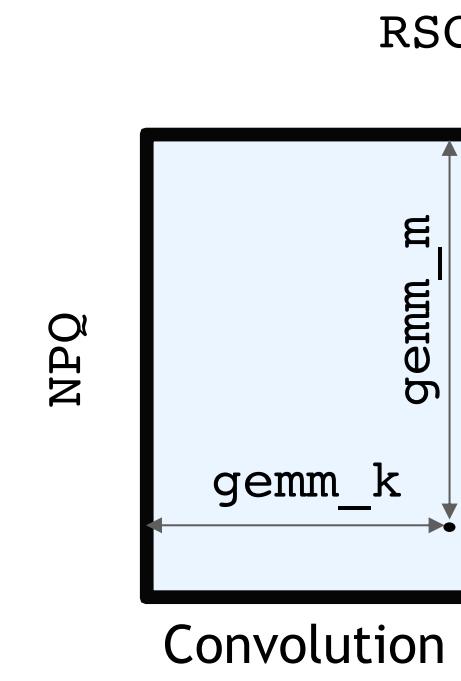
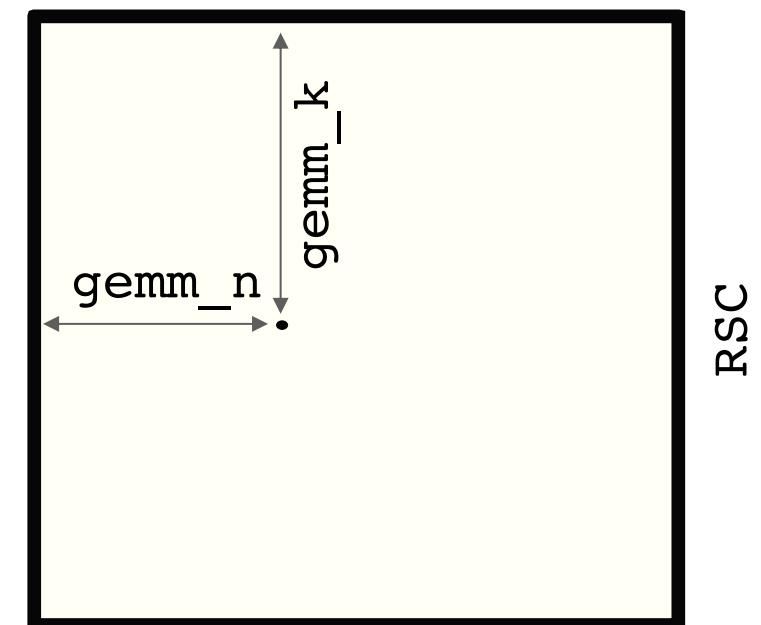
```

CONVOLUTION MAPPED TO GEMM

GEMM M-by-N-by-K dimensions

GEMM-M = NPQ
GEMM-N = K
GEMM-K = RSC

Filter matrix (**B**)



$$\mathbf{C}[gemm_m, gemm_n] = \sum_{gemm_k=0}^{RSC-1} (\mathbf{A}[gemm_m, gemm_k] * \mathbf{B}[gemm_k, gemm_n])$$

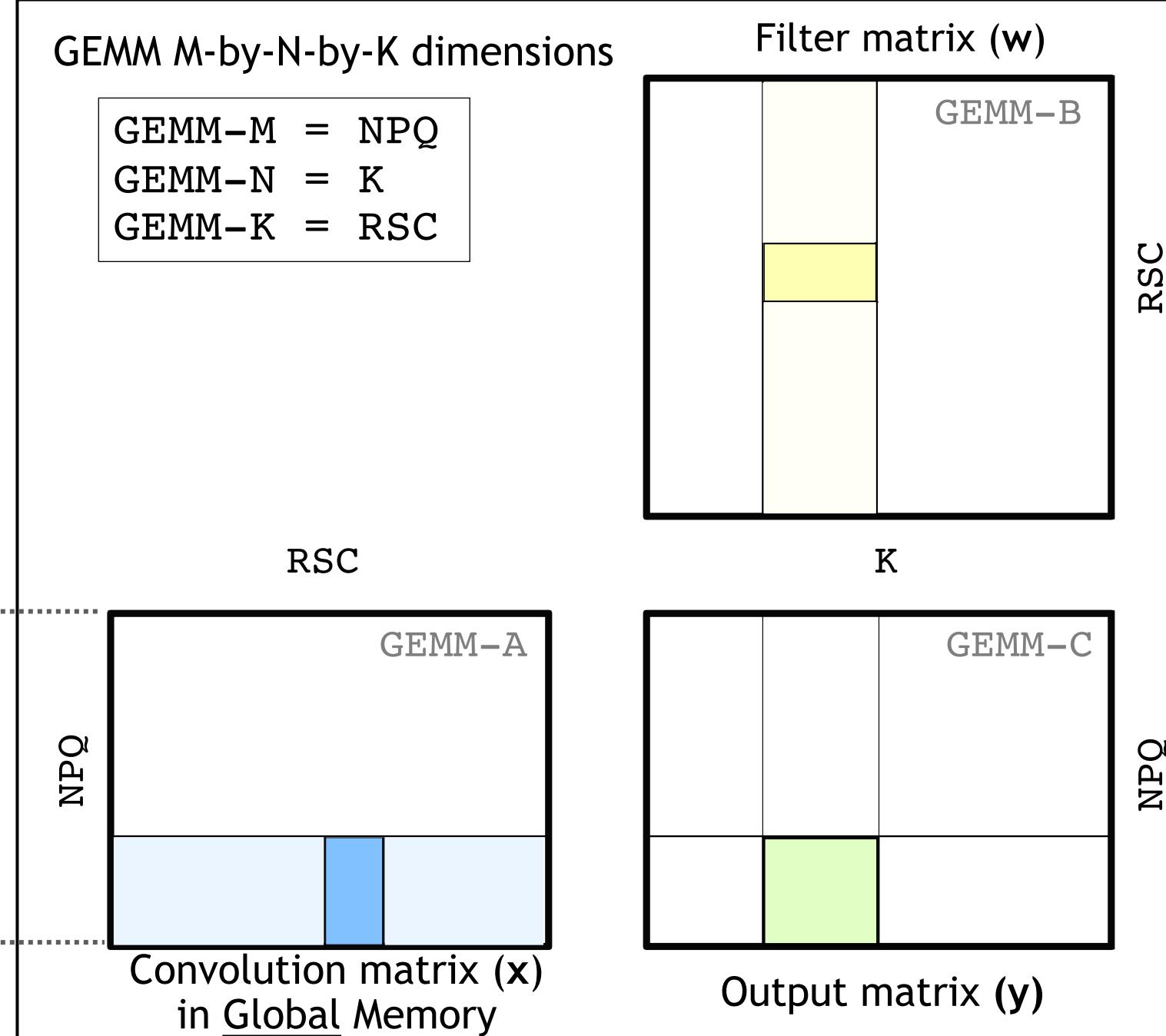
EXPLICIT GEMM CONVOLUTION

Forward Propagation (FPROP)

Forward Propagation (Fprop)

$$y = \text{CONV}(x, w)$$

$x[N, H, W, C]$: 4D activation tensor
 $w[K, R, S, C]$: 4D filter tensor
 $y[N, P, Q, K]$: 4D output tensor



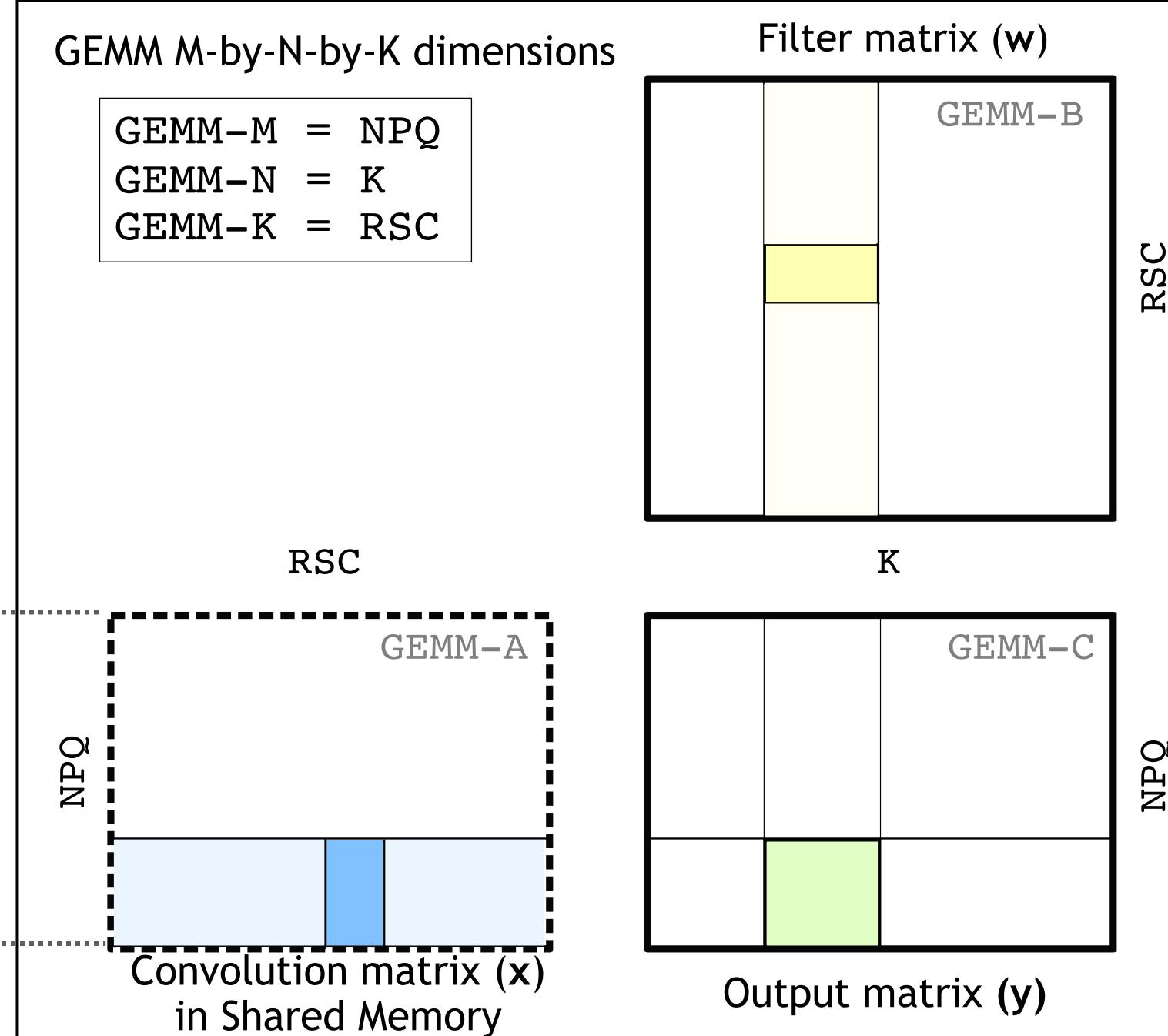
Naïve explicit GEMM convolution implementation creates convolution matrix in Global Memory

Explicit GEMM convolution increases Global Memory footprint and traffic by RS times

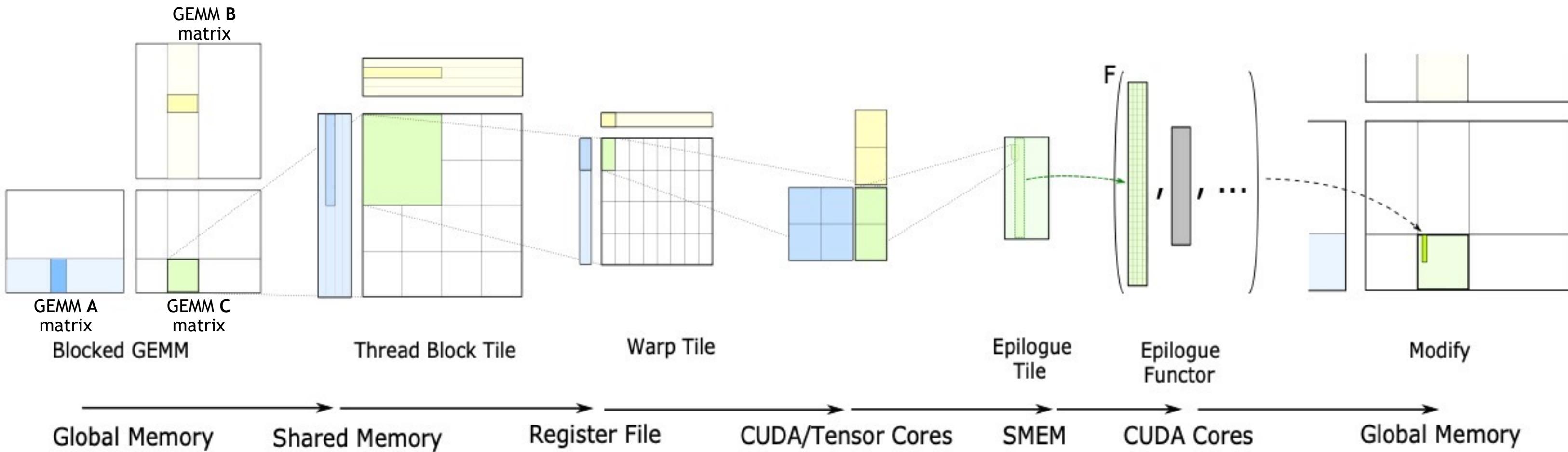
IMPLICIT GEMM CONVOLUTION

Forward Propagation (FPROP)

Forward Propagation (Fprop)
$y = \text{CONV}(x, w)$
$x[N, H, W, C]$: 4D activation tensor
$w[K, R, S, C]$: 4D filter tensor
$y[N, P, Q, K]$: 4D output tensor



BLOCKED GEMM RECAP

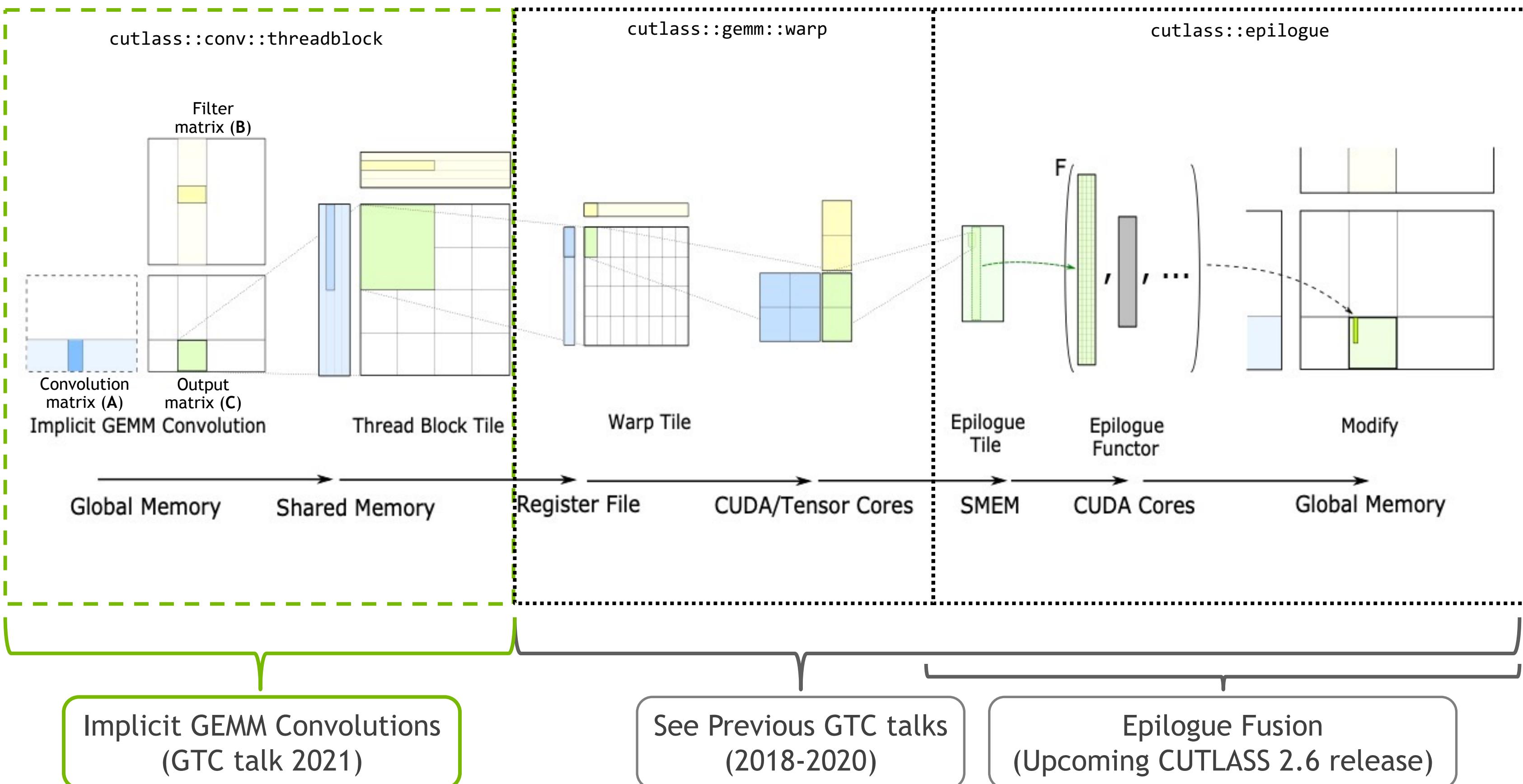


Tiled, hierarchical model: reuse data in Shared Memory and in Registers

See [CUTLASS GTC 2018](#) and [2020](#) talks for more details about this model

IMPLICIT GEMM CONVOLUTION

New CUTLASS components required to implement implicit GEMM convolution



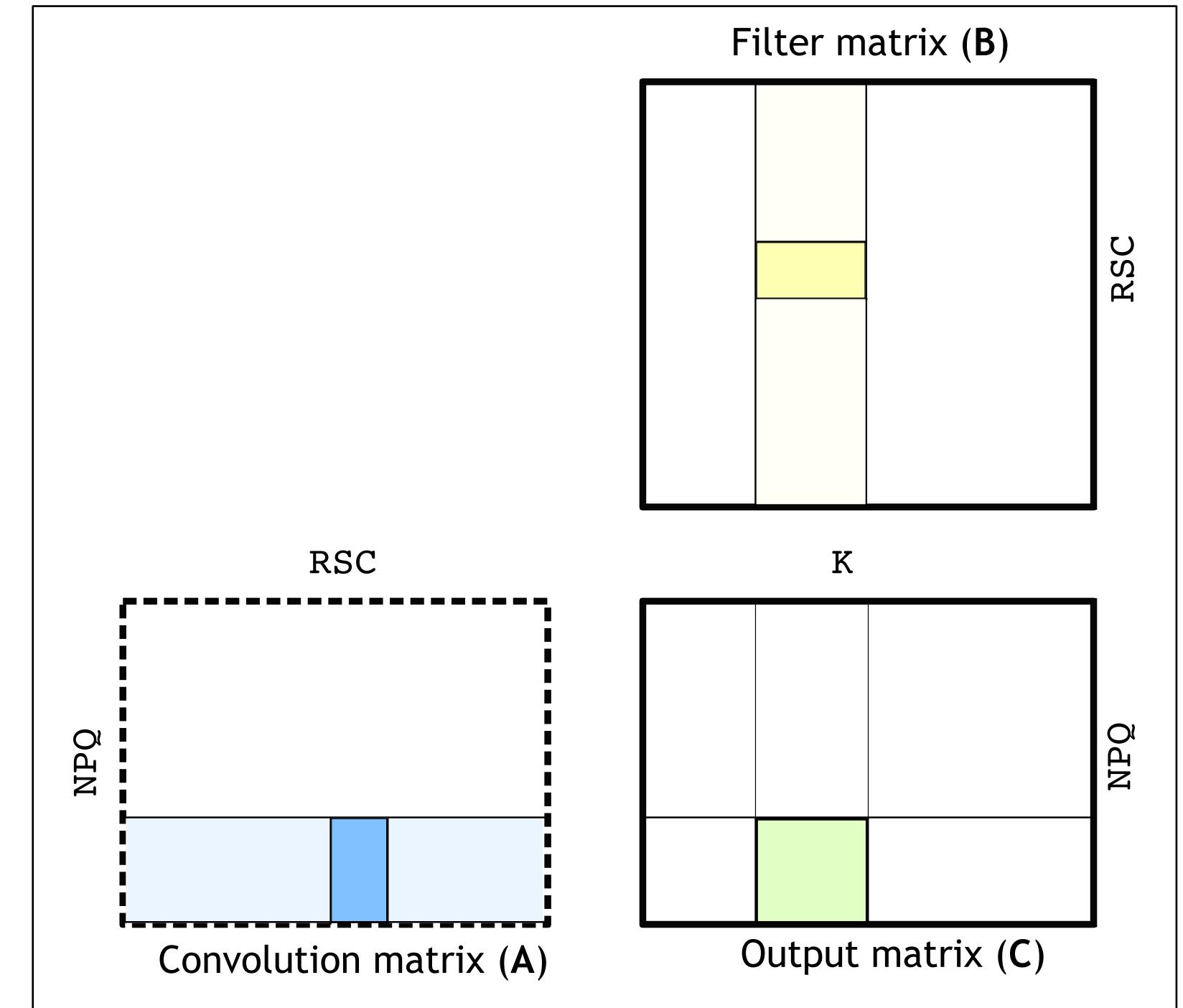


DEEP DIVE IMPLICIT GEMM CONVOLUTIONS

DEEP DIVE IMPLICIT GEMM CONVOLUTION

Implicit GEMM convolution algorithm:

1. Load a tile of convolution and filter matrix into Shared Memory
2. Compute **matrix-multiply accumulate (mma)** on operands in Shared Memory
3. Iterate over RSC dimension



DEEP DIVE IMPLICIT GEMM CONVOLUTION

Loading a tile of convolution and filter matrix through an example tile size

Implicit GEMM convolution algorithm:

1. Load a tile of convolution and filter matrix into Shared Memory

2. Compute matrix-multiply accumulate (mma) on operands in Shared Memory

3. Iterate over RSC dimension

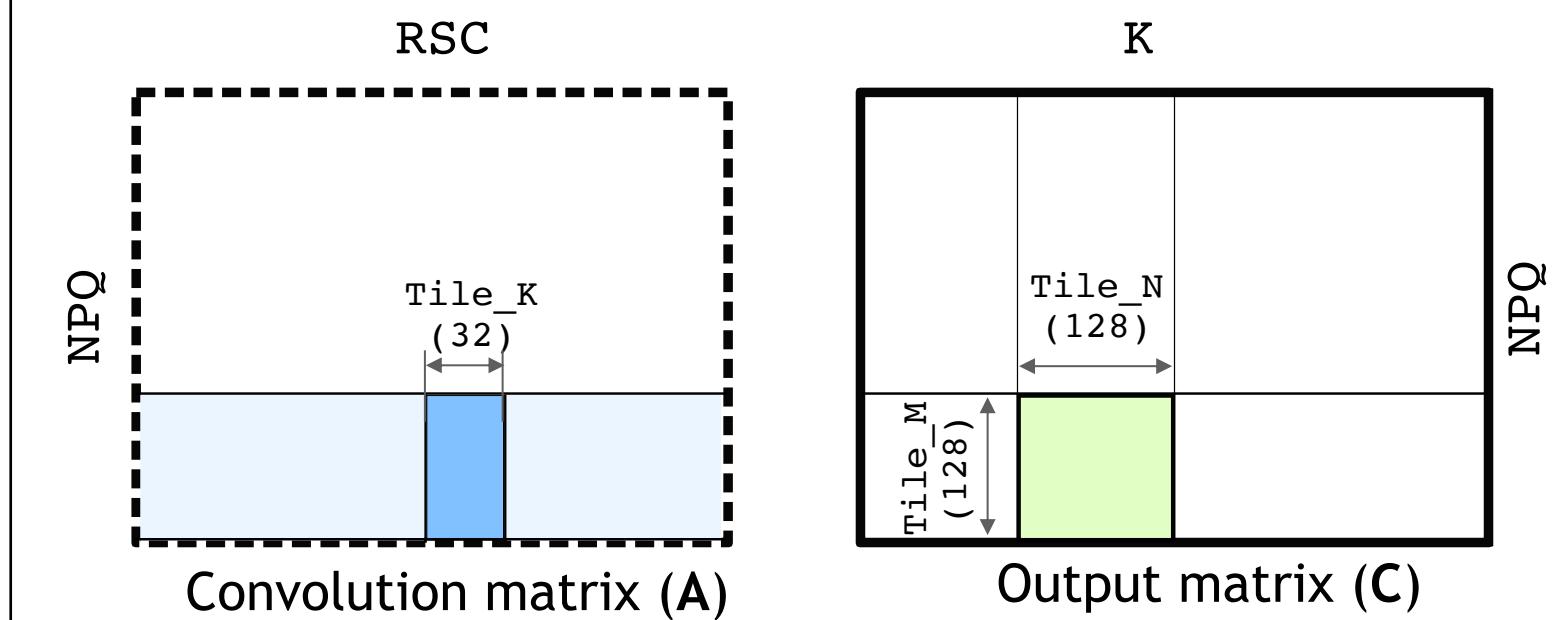
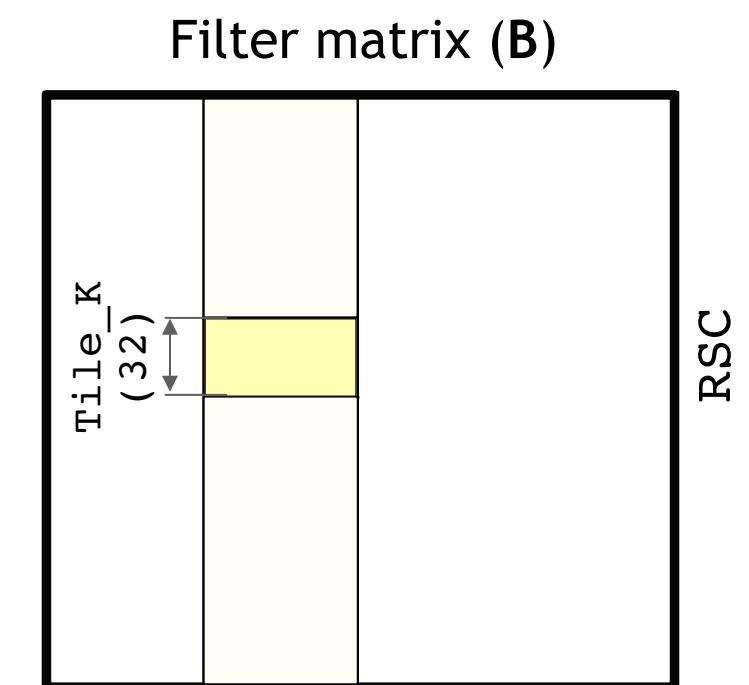
Example:

Tile_M = 128

Tile_N = 128

Tile_K = 32

Input type = F16

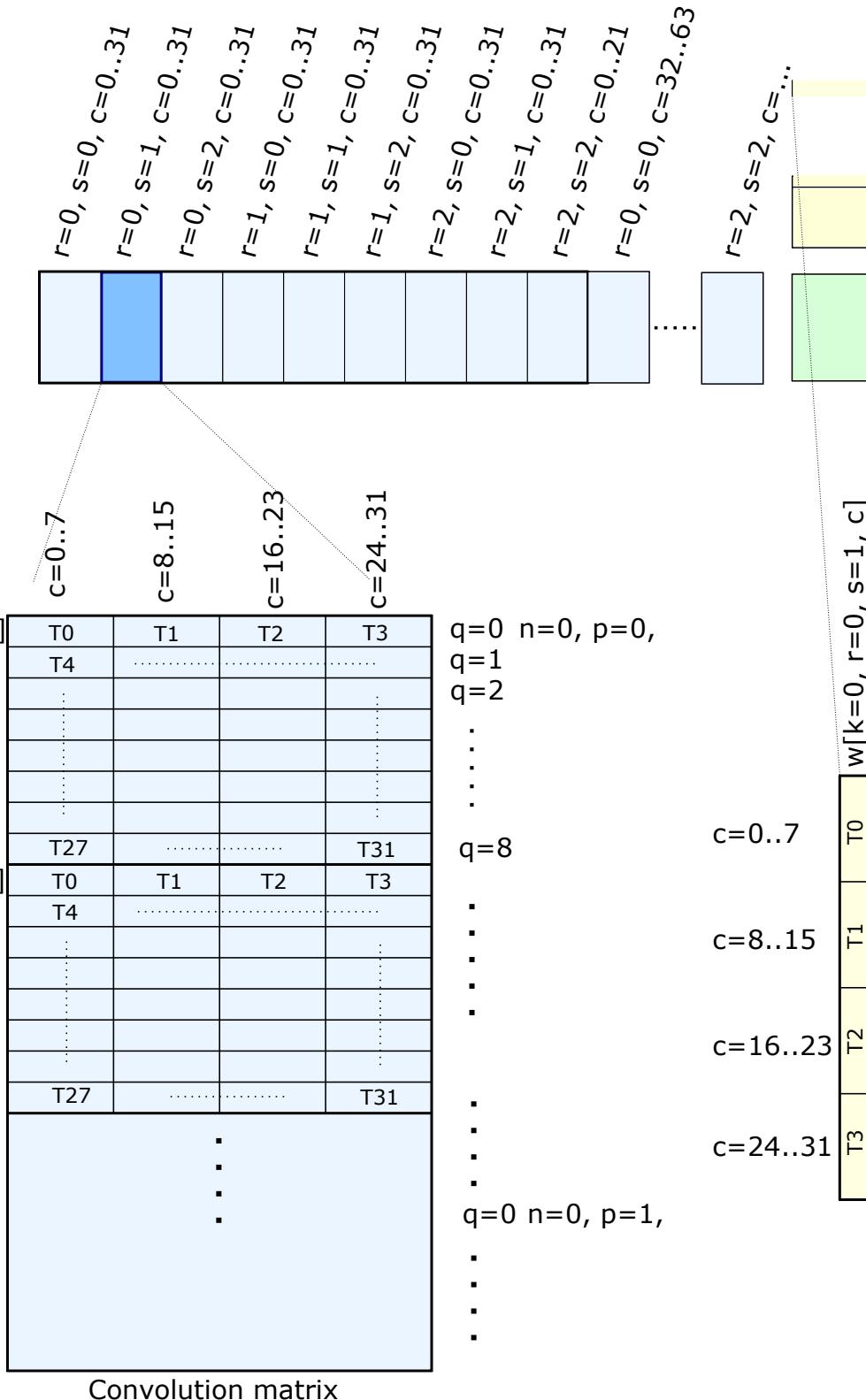


DEEP DIVE IMPLICIT GEMM CONVOLUTION

Load a tile of convolution and filter matrix

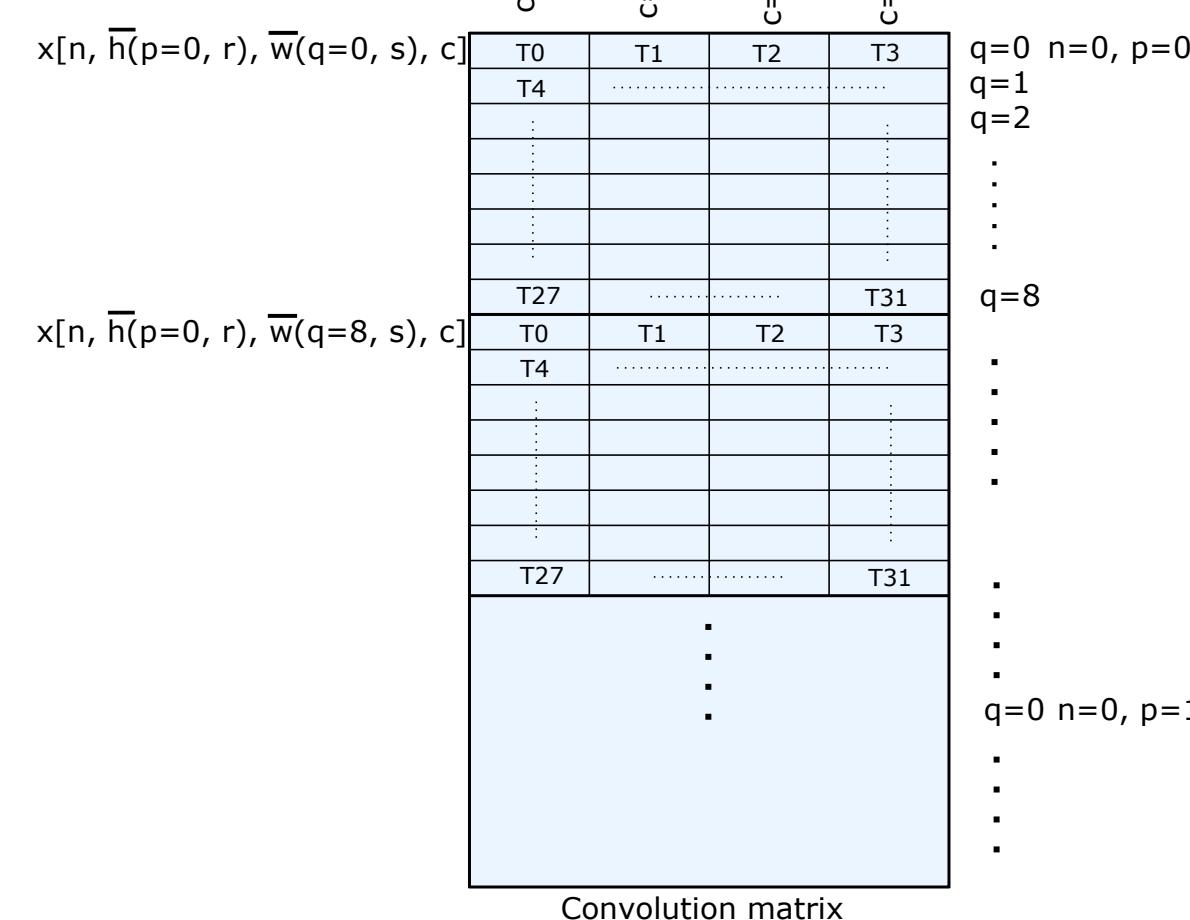
A tile of convolution matrix

- Loads 128-by-32 elements
- Each row mapped to unique (n , p , q) coordinate
- Each column mapped to unique channel c coordinate



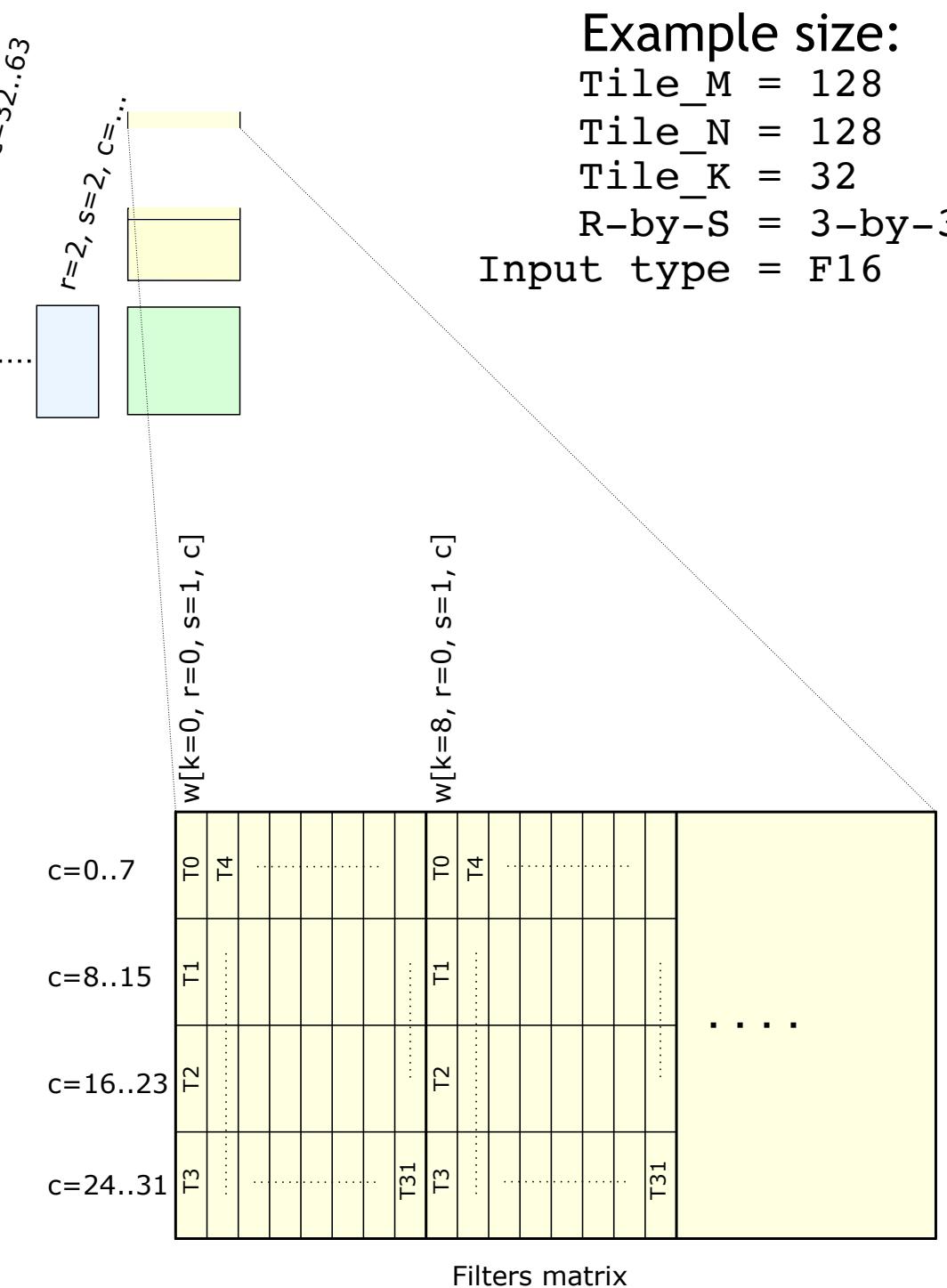
A tile of filter matrix

- Loads 32-by-128 elements
- Each row mapped to unique channel c coordinate
- Each column mapped to unique filter k coordinate



Each thread issues multiple vector loads from each tile

- For example, T_0 's first load brings $c=0..7$ to Shared Memory for F16 operands



Example size:

Tile_M = 128
Tile_N = 128
Tile_K = 32
R-by-S = 3-by-3
Input type = F16

DEEP DIVE IMPLICIT GEMM CONVOLUTION

Compute matrix-multiply accumulate

Implicit GEMM convolution algorithm:

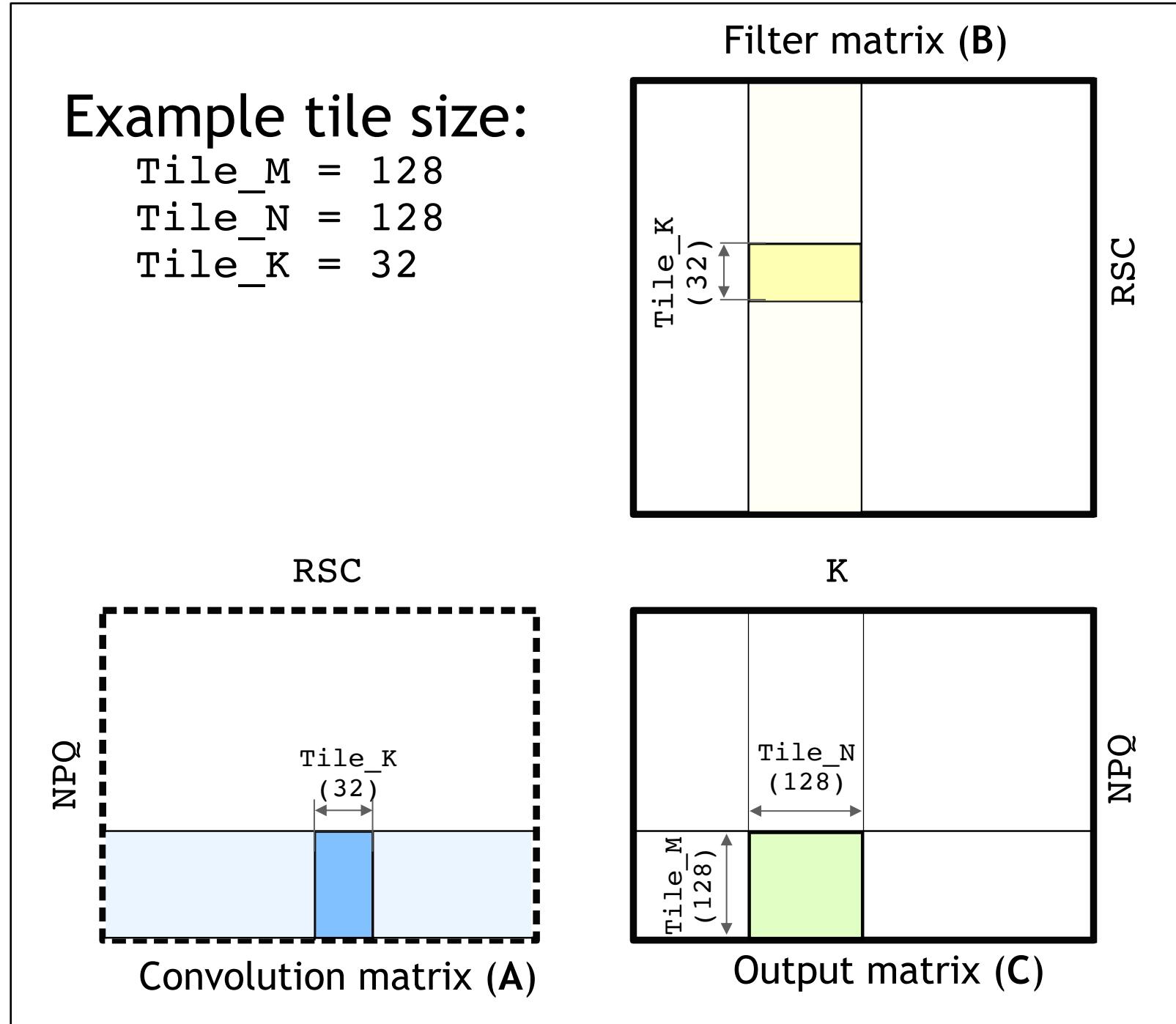
1. Load a tile of convolution and filter matrix into Shared Memory

2. Compute matrix-multiply accumulate (mma) on operands in Shared Memory

3. Iterate over RSC dimension

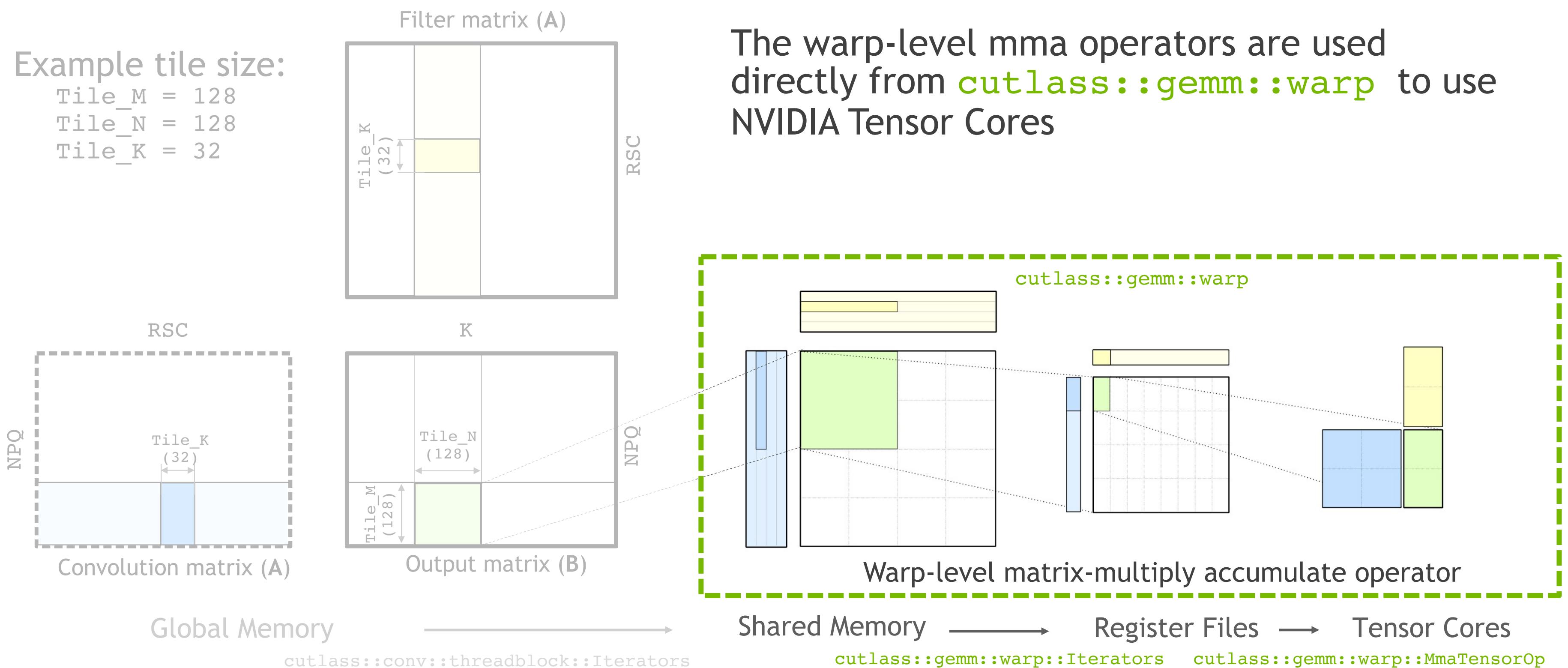
Example tile size:

Tile_M = 128
Tile_N = 128
Tile_K = 32



DEEP DIVE IMPLICIT GEMM CONVOLUTION

Compute matrix-multiply accumulate



DEEP DIVE IMPLICIT GEMM CONVOLUTION

Iterate over RSC dimension

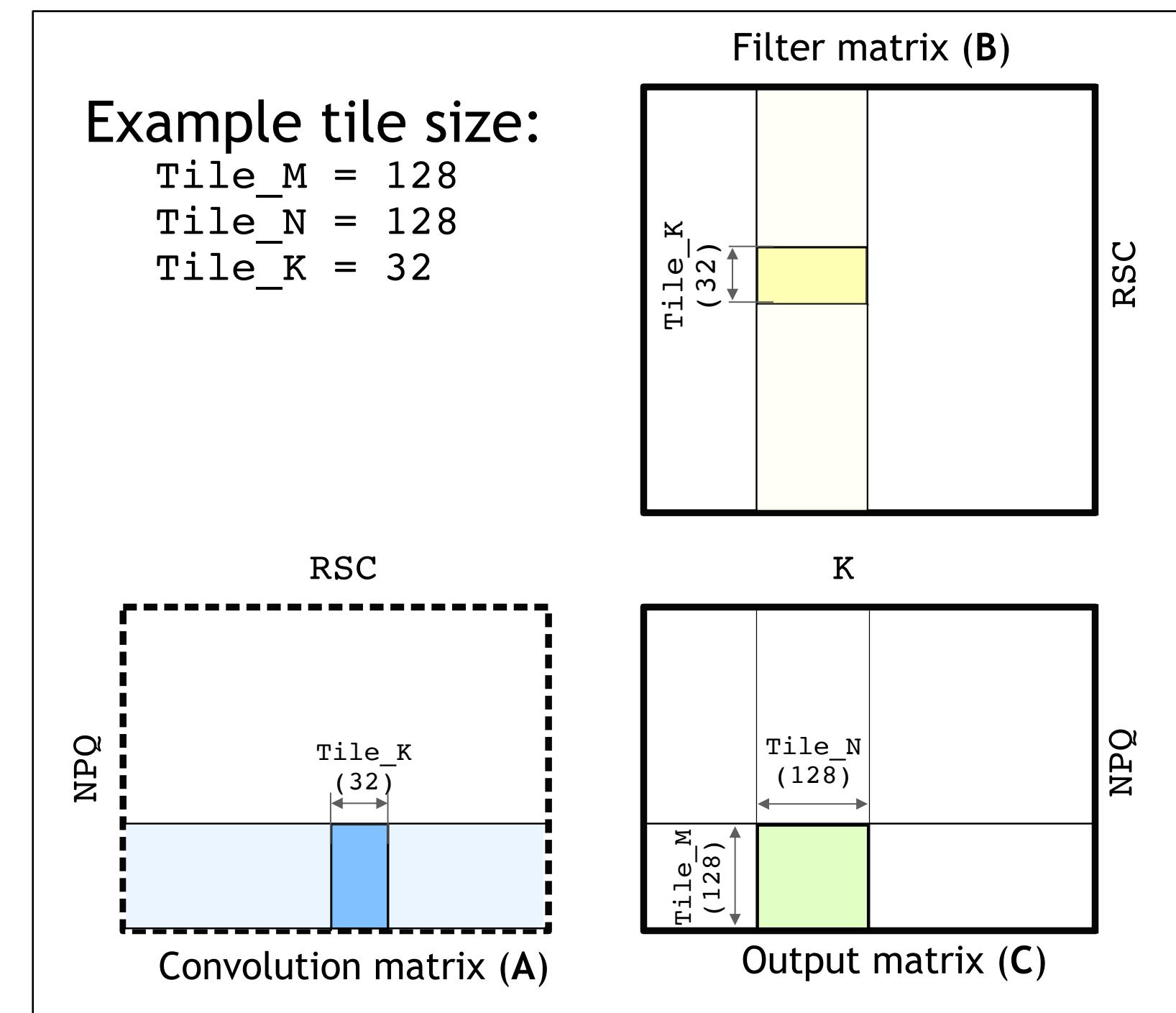
Implicit GEMM convolution algorithm:

1. Load a tile of convolution and filter matrix into Shared Memory

2. Compute matrix-multiply accumulate (mma) on operands in Shared Memory

3. Iterate over RSC dimension

- a) Advance to load next tiles in Shared Memory
- b) Ensure accumulation for all filter positions (r, s) and channels c



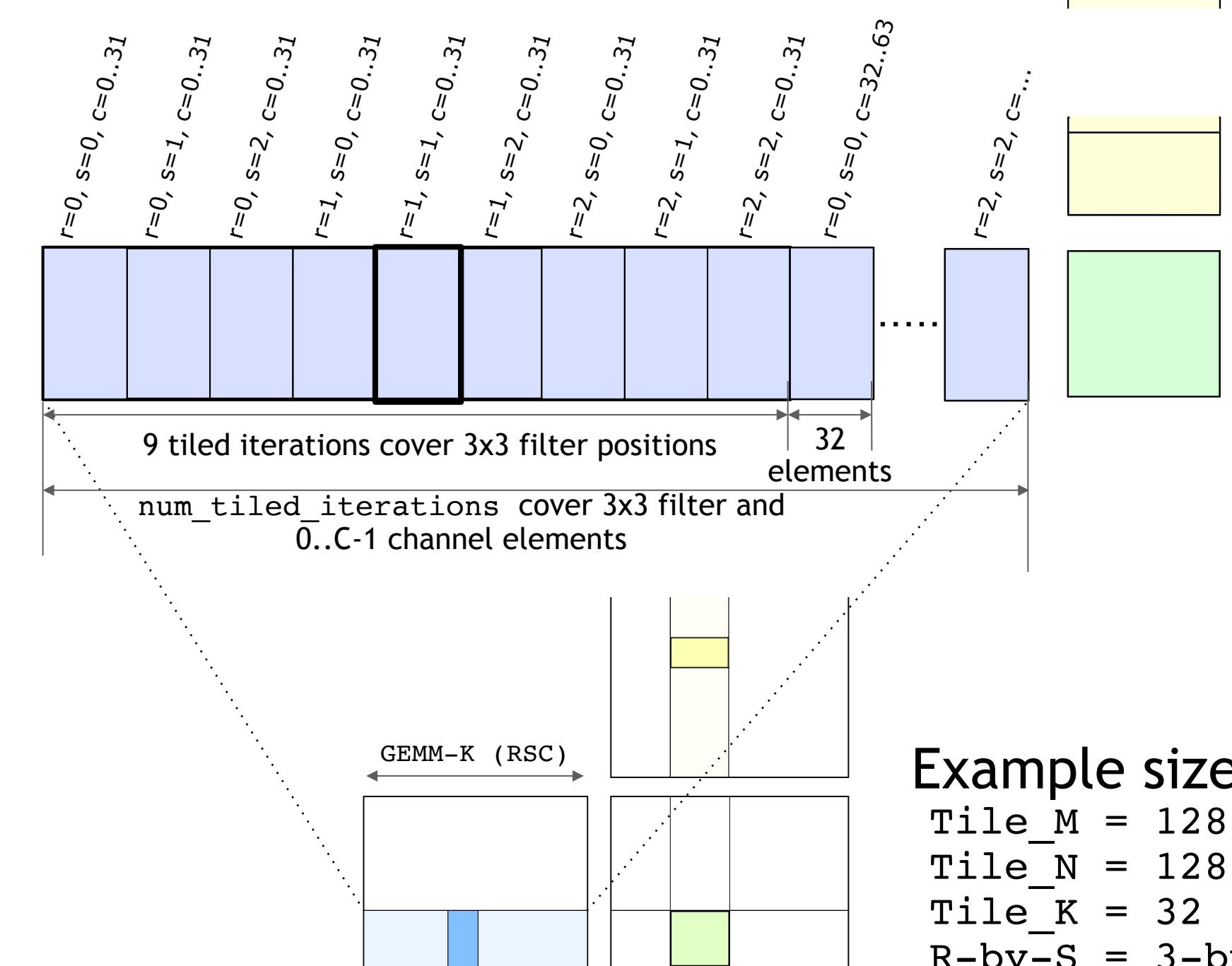
DEEP DIVE IMPLICIT GEMM CONVOLUTION

Iterate over RSC dimension

To cover the entire GEMM-K (RSC) dimension

- Process tiles in RSC dimension by going over:
 - filter s positions
 - filter r positions
 - Tile_K channel c elements
- launch enough tiled iterations to cover all channel elements (C) and filter positions (R-by-S) :

```
num_tiled_iterations =
R * S * ((C + Tile_K - 1)/Tile_K)
```



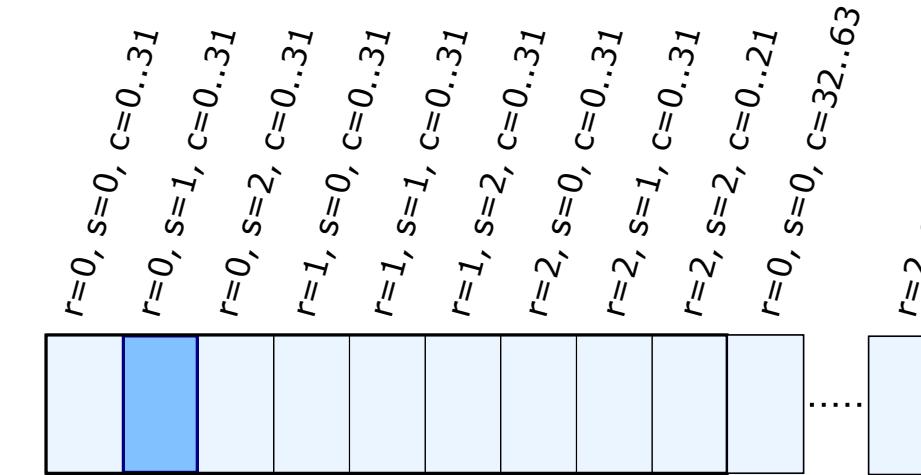
CUTLASS: BUILDING COHERENT AND COMPLETE ABSTRACTIONS

`cutlass::conv::threadblock::Iterators`

CUTLASS convolution iterators
implement the below abstractions:

advance()

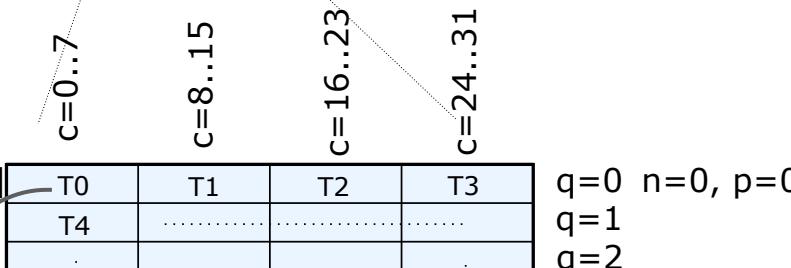
moves to the next tiled iteration in GEMM-K



Example size:
Tile_M = 128
Tile_N = 128
Tile_K = 32
R-by-S = 3-by-3

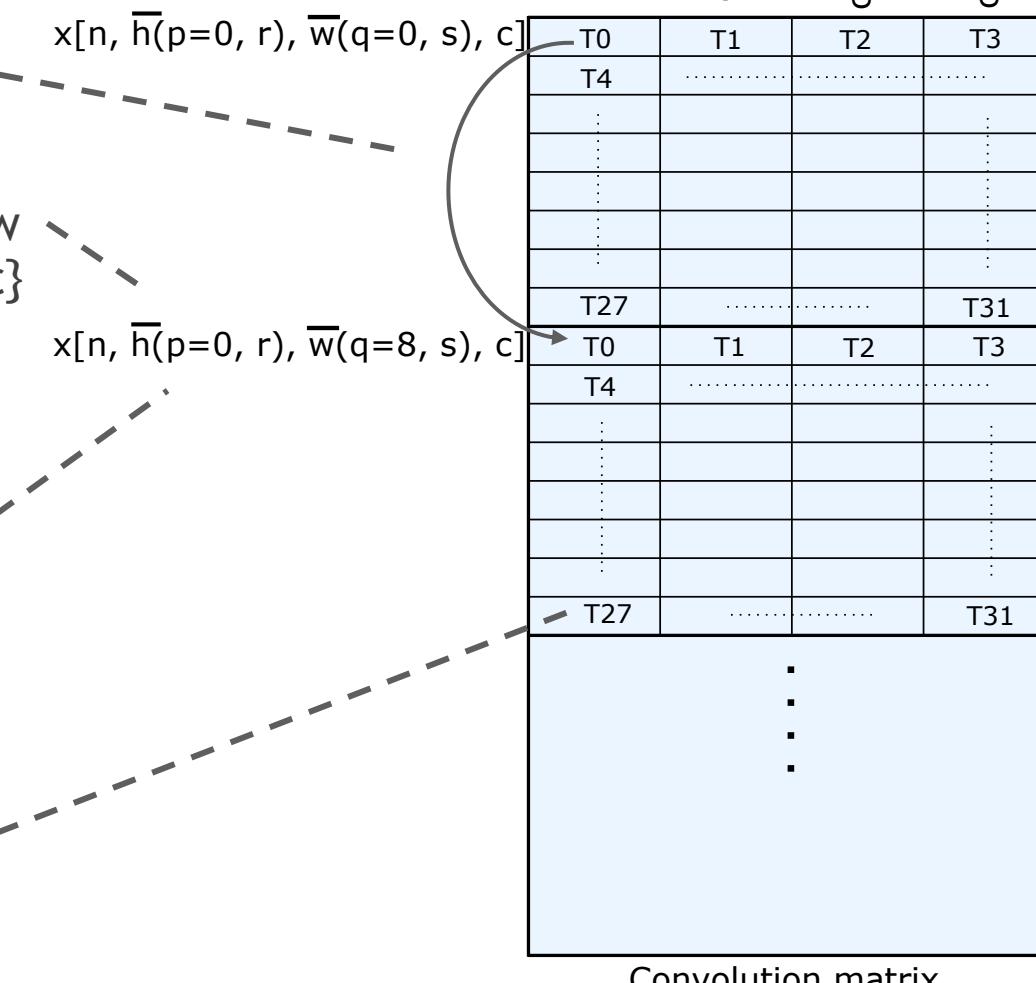
operator++()

moves to the next load position for the thread



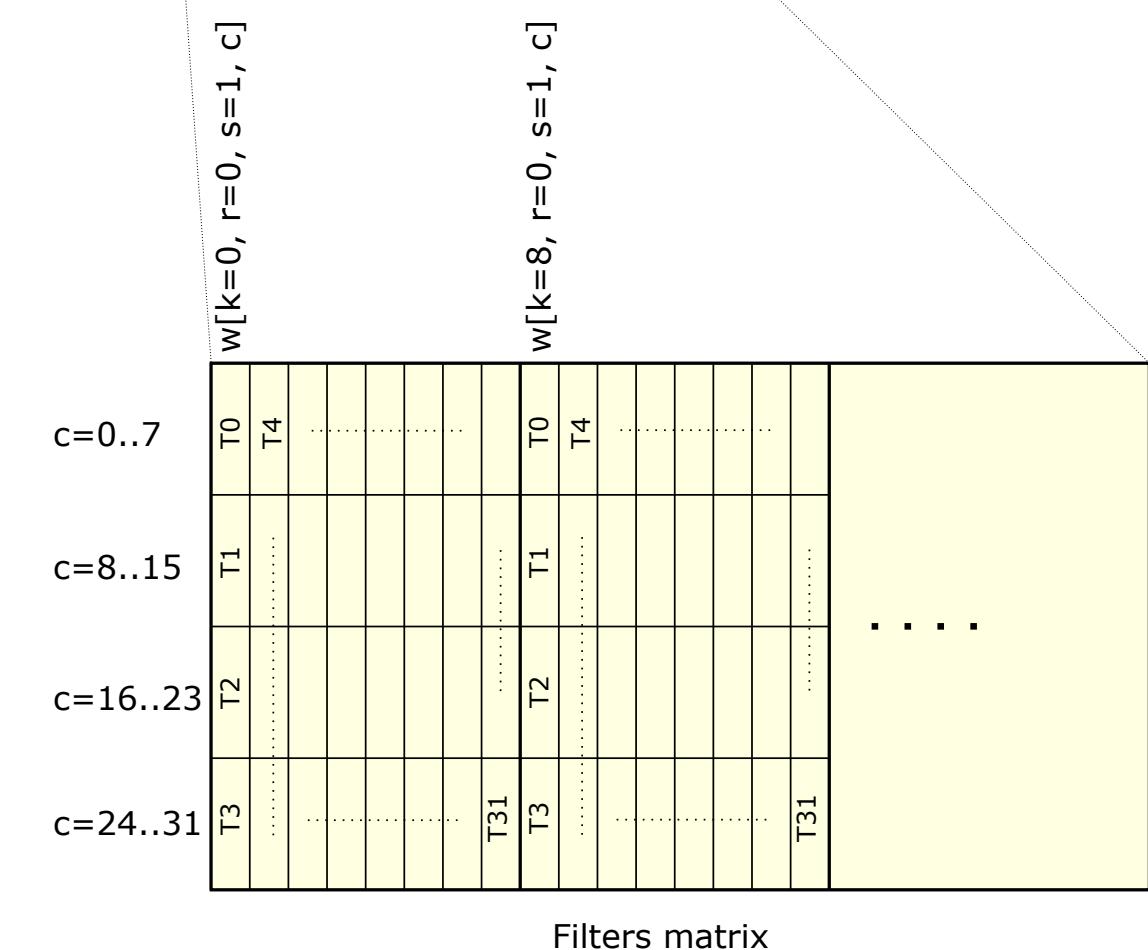
at()

applies $\bar{h}(p, r)$ and $\bar{w}(q, s)$ functions to map pq to hw and returns coordinates in activation tensor $\{n, h, w, c\}$



valid()

checks out-of-bound accesses for tensors in Global Memory



get()

fetches pointer in Global Memory based on tensor coordinates

ANALYTIC TILE ITERATOR OVER CONVOLUTION MATRIX

`cutlass::conv::threadblock`

CUTLASS convolution iterators implement the below abstractions:

[advance\(\)](#)

moves to the next tiled iteration in GEMM-K

[operator++\(\)](#)

moves to the next load position for the thread

[at\(\)](#)

applies $\bar{h}(p, r)$ and $\bar{w}(q, s)$ functions to map pq to hw and returns coordinates in activation tensor {n,h,w,c}

[valid\(\)](#)

checks out-of-bound accesses for tensors in Global Memory

[get\(\)](#)

fetches pointer in Global Memory based on tensor coordinates

```
Iterator(..., Element const* ptr, ...) : ptr(ptr)

Tensor4DCoord at() {
    // n and c are trivially available as Iterator state
    h_bar = p * stride_h - pad_h + r * dilation_h;
    b_bar = q * stride_w - pad_w + s * dilation_w;
    return Tensor4DCoord(n, h_bar, w_bar, c);
}

bool valid() {
    Tensor4DCoord coord = at();

    return (coord.n < N) &&
        (coord.h >= 0 && coord.h < H) &&
        (coord.w >= 0 && coord.w < W) &&
        (coord.c < C);
}

Element* get() {
    Tensor4DCoord coord = at();                                // Strides in NHWC tensor
                                                               // n_stride = HWC
                                                               // h_stride = WC
                                                               // w_stride = C
                                                               // c_stride = 1
    int64_t offset = coord.n * (n_stride) +                  // n_stride = HWC
                  coord.h * (h_stride) +                  // h_stride = WC
                  coord.w * (w_stride) +                  // w_stride = C
                  coord.c;                            // c_stride = 1

    return (ptr + offset);
}
```

ANALYTIC TILE ITERATOR OVER CONVOLUTION MATRIX

`cutlass::conv::threadblock`

CUTLASS convolution iterators implement the below abstractions:

[advance\(\)](#)

moves to the next tiled iteration in GEMM-K

[operator++\(\)](#)

moves to the next load position for the thread

[at\(\)](#)

applies $\bar{h}(p, r)$ and $\bar{w}(q, s)$ functions to map pq to hw and returns coordinates in activation tensor {n,h,w,c}

[valid\(\)](#)

checks out-of-bound accesses for tensors in Global Memory

[get\(\)](#)

fetches pointer in Global Memory based on tensor coordinates

```
Iterator(..., Element const* ptr, ...) : ptr(ptr)

Tensor4DCoord at() {
    // n and c are trivially available as Iterator state
    h_bar = p * stride_h - pad_h + r * dilation_h;
    b_bar = q * stride_w - pad_w + s * dilation_w;
    return Tensor4DCoord(n, h_bar, w_bar, c);
}

bool valid() {
    Tensor4DCoord coord = at();

    return (coord.n < N) &&
        (coord.h >= 0 && coord.h < H) &&
        (coord.w >= 0 && coord.w < W) &&
        (coord.c < C);
}

Element* get() {
    Tensor4DCoord coord = at();                                // Strides in NHWC tensor
                                                               // n_stride = HWC
    int64_t offset = coord.n * (n_stride) +                  // h_stride = WC
                  coord.h * (h_stride) +                  // w_stride = C
                  coord.w * (w_stride) +                  // c_stride = 1
                  coord.c;
    return (ptr + offset);
}
```

A naïve *analytic* implementation issues too many non-tensor core math instructions



OPTIMIZED IMPLEMENTATION OF CONVOLUTION ABSTRACTIONS

OPTIMIZED IMPLEMENTATION OF CONVOLUTION ABSTRACTIONS

Precompute invariants

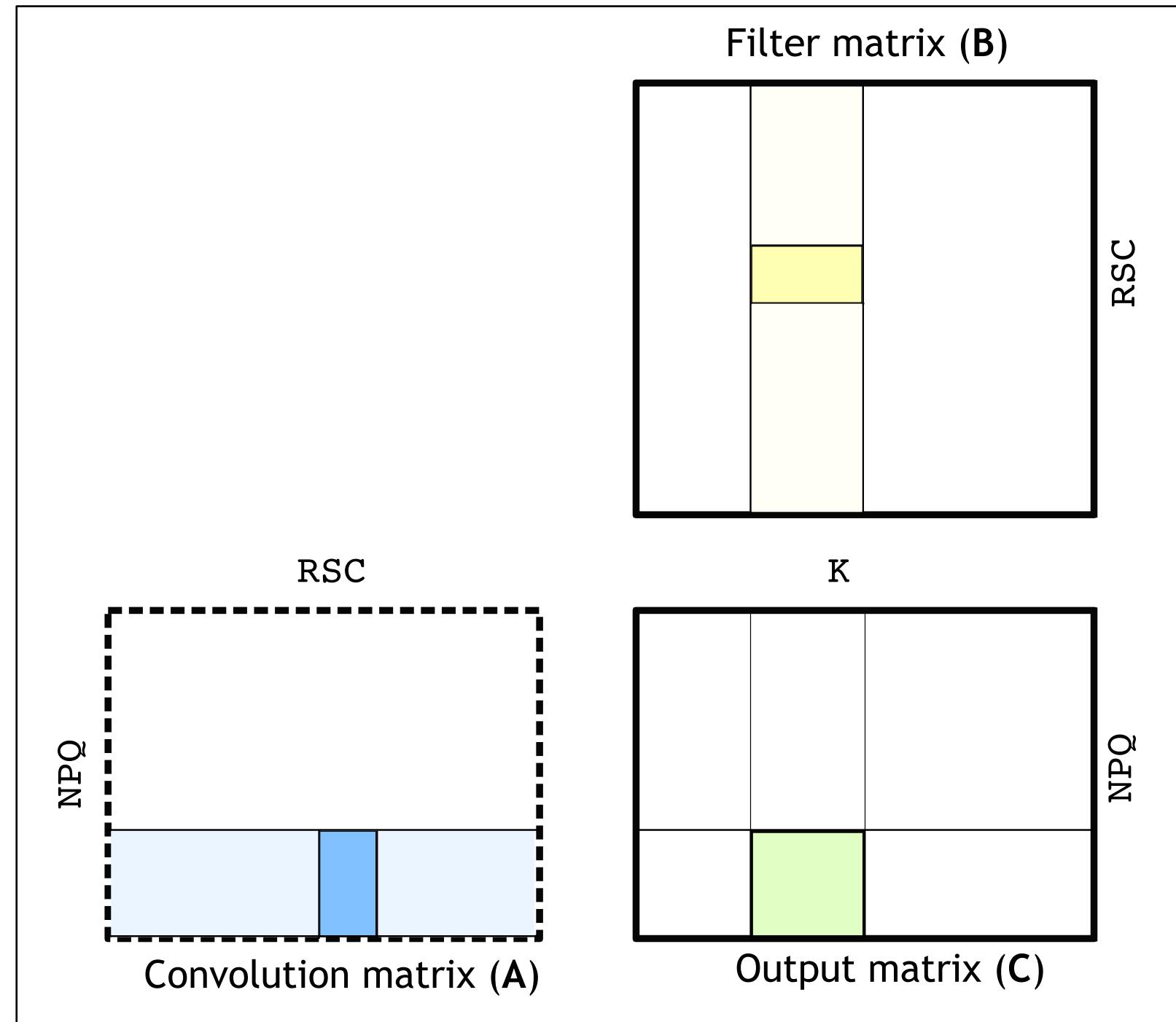
Strategies:

1. Delta tables

- a) Offsets to access activation tensor
- b) Invariant for entire kernel-execution
- c) Reduced pointer arithmetic in the mainloop

2. Mask predicates

- a) 32b predicate vectors for OOB check
- b) Invariant for entire thread block
- c) Reduced logical arithmetic in the mainloop



OPTIMIZED IMPLEMENTATION OF CONVOLUTION ABSTRACTIONS

Precomputed delta tables

Strategies:

1. Delta tables

- a) Offsets to access activation tensor
- b) Invariant for entire kernel-execution
- c) Reduced pointer arithmetic in the mainloop

2. Mask predicates

- a) 32b predicate vectors for OOB check
- b) Invariant for entire thread block
- c) Reduced logical arithmetic in the mainloop

```
Iterator(..., Element const* ptr, ...) : ptr(ptr)

Tensor4DCoord at() {
    // n and c are trivially available as Iterator state
    h_bar = p * stride_h - pad_h + r * dilation_h;
    b_bar = q * stride_w - pad_w + s * dilation_w;
    return Tensor4DCoord(n, h_bar, w_bar, c);
}

bool valid() {
    Tensor4DCoord coord = at();

    return (coord.n < N) &&
        (coord.h >=0 && coord.h < H) &&
        (coord.w >=0 && coord.w < W) &&
        (coord.c < C);
}

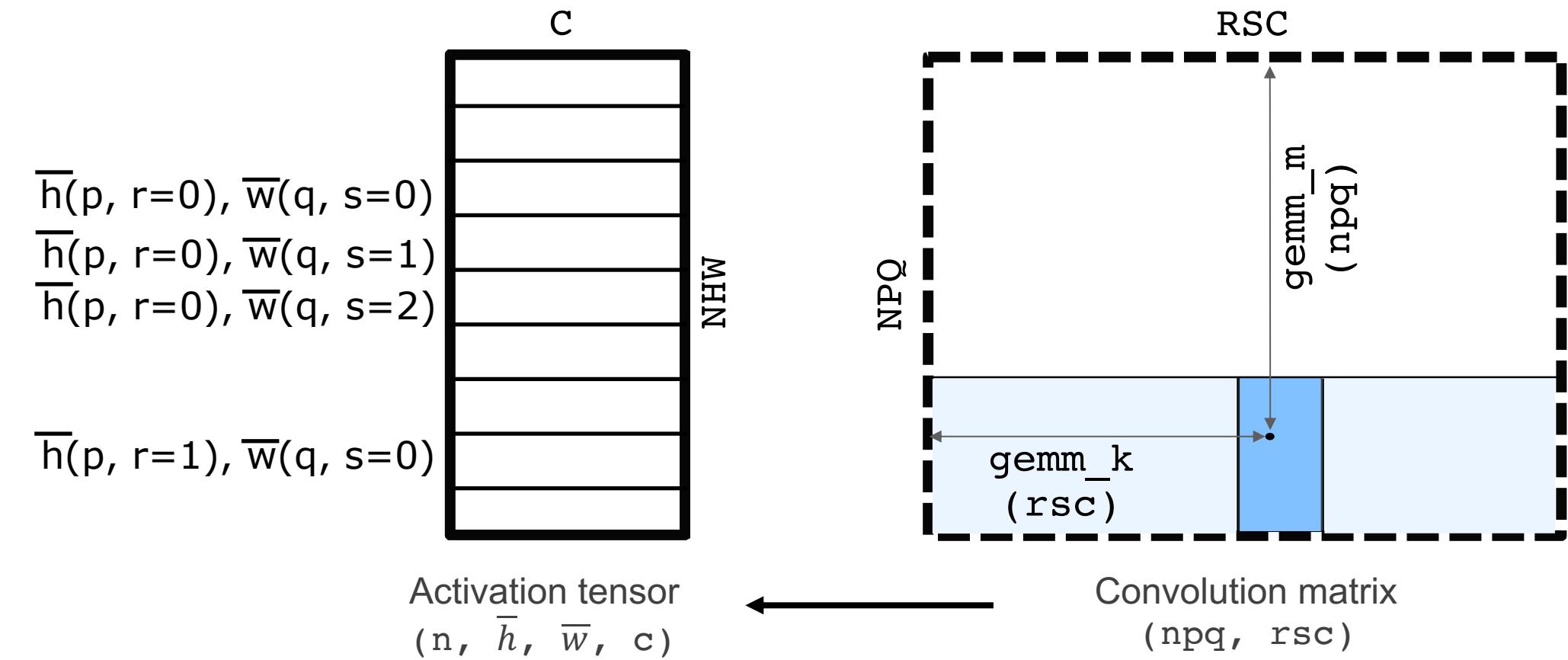
Element* get() {
    Tensor4DCoord coord = at();                                // Strides in NHWC tensor
    int64_t offset = coord.n * (n_stride) +                  // n_stride = HWC
                coord.h * (h_stride) +                  // h_stride = WC
                coord.w * (w_stride) +                  // w_stride = C
                coord.c;                                // c_stride = 1
    return (ptr + offset);
}
```

OPTIMIZED IMPLEMENTATION OF CONVOLUTION ABSTRACTIONS

Precomputed delta tables

To cover RSC dimension
for a unique npq in convolution
matrix:

- index into every rsc position
- move in s, r, and c dimension
- too many non-tensor core
operations!!!



$$\begin{aligned} n &= \text{gemm_m} / (\text{PQ}) \\ \text{npq_residue} &= \text{gemm_m \% (PQ)} \\ p &= \text{npq_residue} / Q \\ q &= \text{crs_residue \% Q} \end{aligned}$$

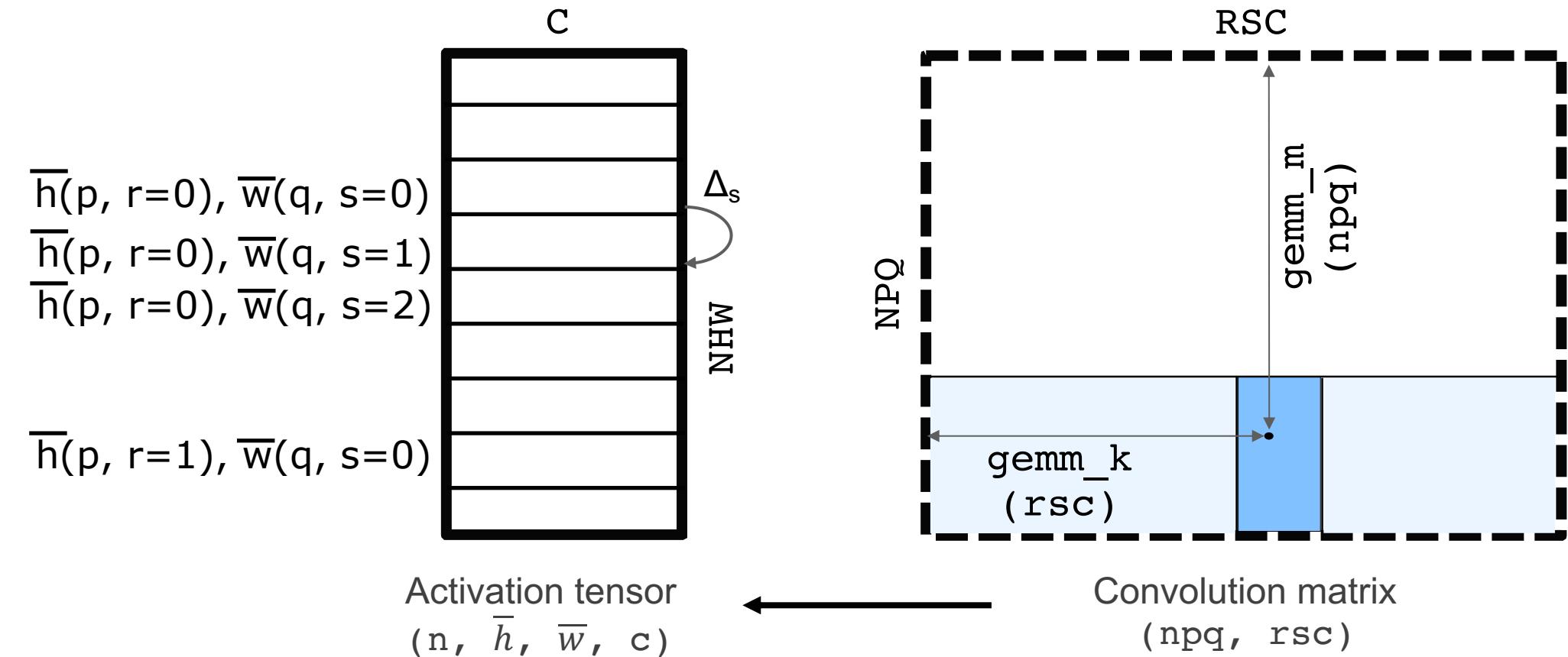
$$\begin{aligned} \bar{h}(p, r) &= p * \text{stride_h} - \text{pad_h} + r * \text{dilation_h} \\ \bar{w}(q, s) &= q * \text{stride_w} - \text{pad_w} + s * \text{dilation_w} \end{aligned}$$

$$c = \{0, 1, 2, \dots, (\text{Tile_k-1})\}$$

OPTIMIZED IMPLEMENTATION OF CONVOLUTION ABSTRACTIONS

Precomputed delta tables

Moving from filter position $s=0$ to $s=1$ for a fixed filter position r and channel c is moving vertically by Δ_s elements in activation tensor (NHW-by-C)



$$\Delta_s = C \text{ elements}$$

```

n = gemm_m / (PQ)
npq_residue = gemm_m % (PQ)
p = npq_residue / Q
q = crs_residue % Q

```

Example problem size:
 $R\text{-by-}S = (3, 3)$
 $(pad_h, pad_w) = (1, 1)$
 $(stride_h, stride_w) = (1, 1)$
 $(dilation_h, dilation_w) = (1, 1)$

Constant for a fixed npq
and problem size

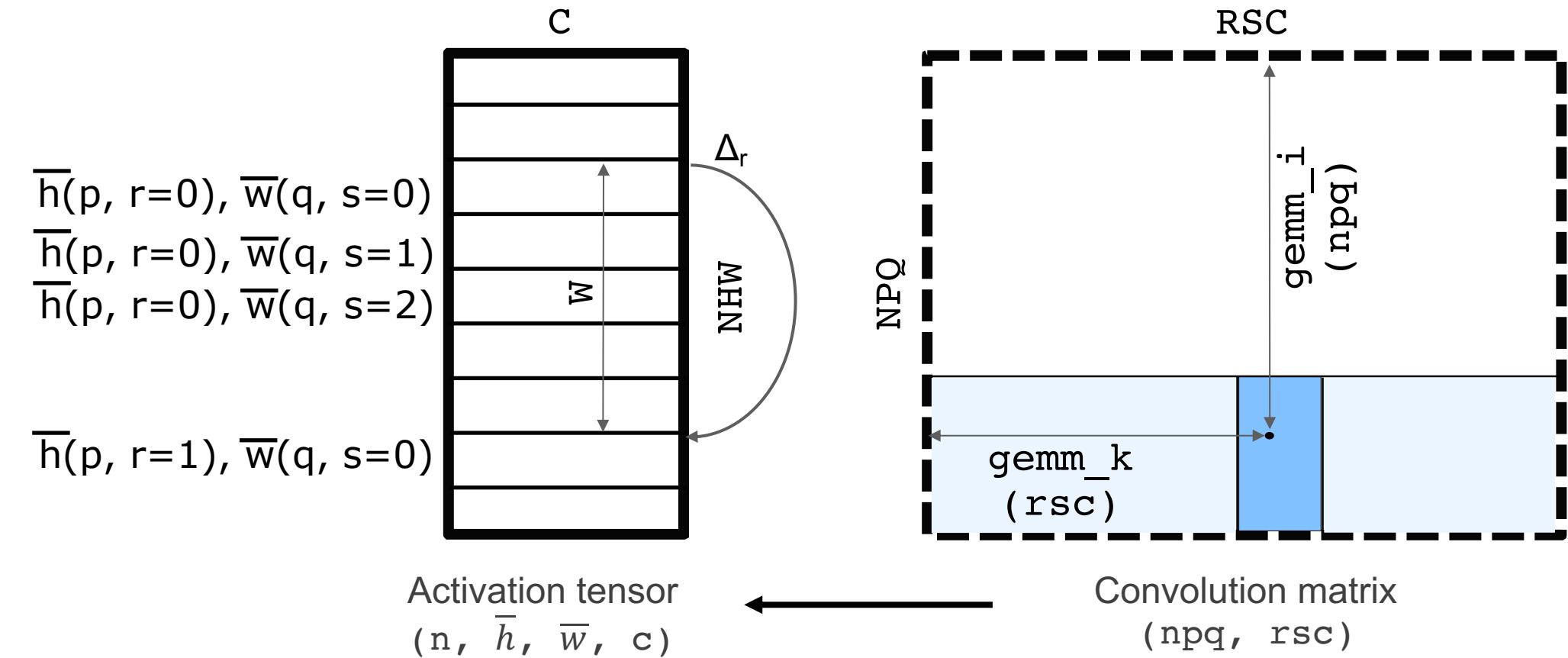
$$\begin{aligned}
\bar{h}(p, r) &= p * \text{stride}_h - \text{pad}_h + r * \text{dilation}_h \\
\bar{w}(q, s) &= q * \text{stride}_w - \text{pad}_w + s * \text{dilation}_w
\end{aligned}$$

$$c = \{0, 1, 2, \dots, (\text{Tile}_k-1)\}$$

OPTIMIZED IMPLEMENTATION OF CONVOLUTION ABSTRACTIONS

Precomputed delta tables

Moving from filter position $r=0$ to $r=1$ for a fixed filter position s and channel c is moving vertically by Δ_r elements in activation tensor (NHW-by-C)



$$\Delta_r = W * C \text{ elements}$$

```

n = gemm_m / (PQ)
npq_residue = gemm_m % (PQ)
p = npq_residue / Q
q = crs_residue % Q

```

Example problem size:
 $R\text{-by-}S = (3, 3)$
 $(pad_h, pad_w) = (1, 1)$
 $(stride_h, stride_w) = (1, 1)$
 $(dilation_h, dilation_w) = (1, 1)$

Constant for a fixed npq
and problem size

$$\bar{h}(p, r) = p * \text{stride}_h - \text{pad}_h + r * \text{dilation}_h$$

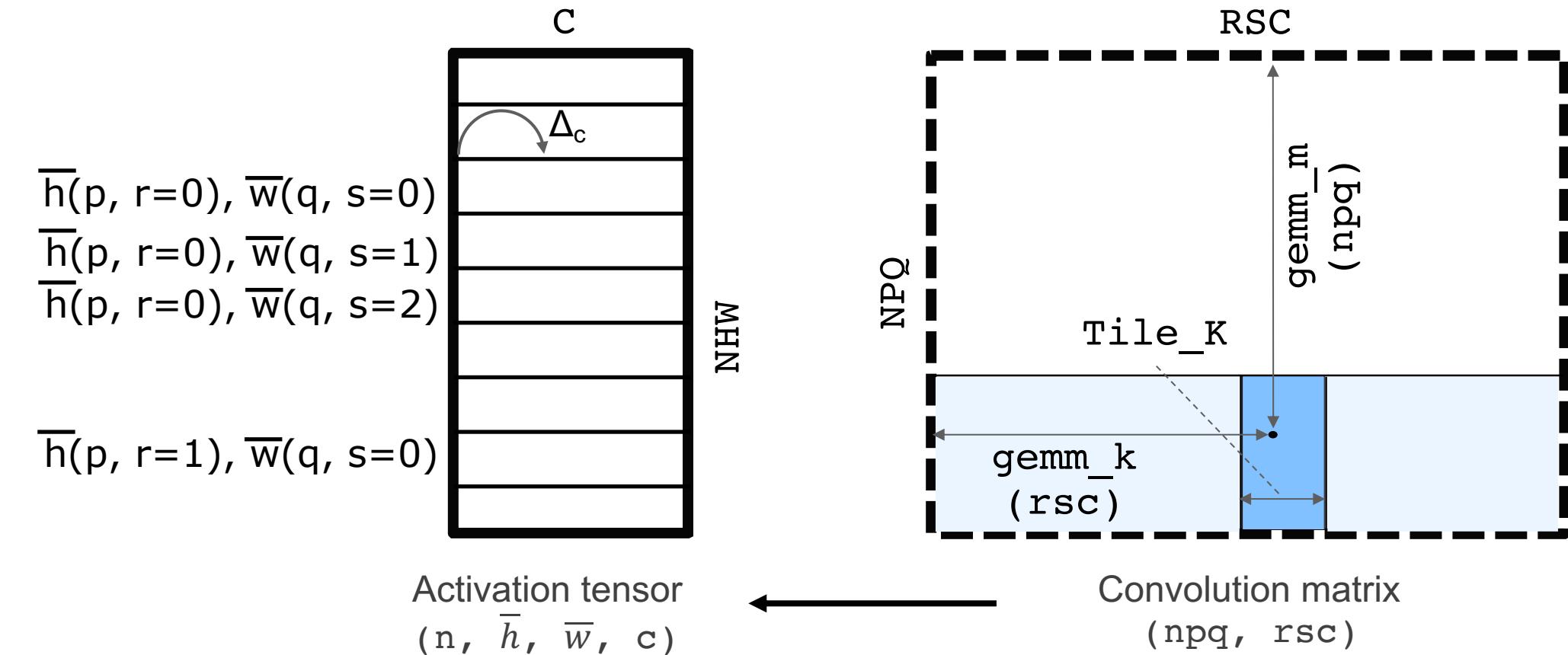
$$\bar{w}(q, s) = q * \text{stride}_w - \text{pad}_w + s * \text{dilation}_w$$

$$c = \{0, 1, 2, \dots, (\text{Tile}_k-1)\}$$

OPTIMIZED IMPLEMENTATION OF CONVOLUTION ABSTRACTIONS

Precomputed delta tables

Moving to next Tile_K channel elements for filter position $r=0$ and $s=0$ is moving horizontally by Δ_c elements in activation tensor (NHW-by-C)



$$\Delta_c = \text{Tile_K elements}$$

$$\begin{aligned} n &= \text{gemm_m} / (\text{PQ}) \\ \text{npq_residue} &= \text{gemm_m \% (PQ)} \\ p &= \text{npq_residue} / Q \\ q &= \text{crs_residue \% Q} \end{aligned}$$

Example problem size:
 $R\text{-by-}S = (3, 3)$
 $(\text{pad}_h, \text{pad}_w) = (1, 1)$
 $(\text{stride}_h, \text{stride}_w) = (1, 1)$
 $(\text{dilation}_h, \text{dilation}_w) = (1, 1)$

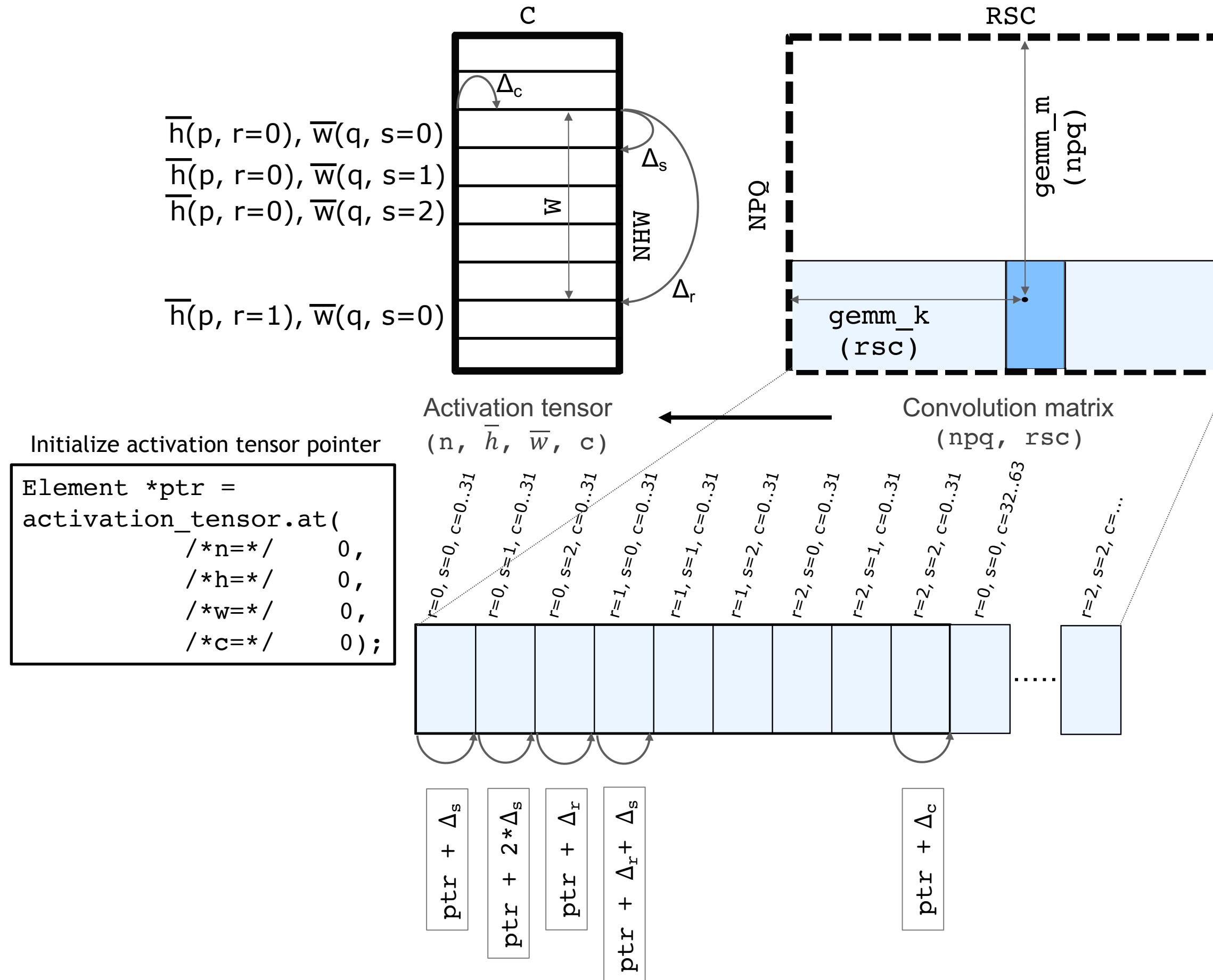
Constant for a fixed npq and problem size

$$\begin{aligned} \bar{h}(p, r) &= p * \text{stride}_h - \text{pad}_h + r * \text{dilation}_h \\ \bar{w}(q, s) &= q * \text{stride}_w - \text{pad}_w + s * \text{dilation}_w \end{aligned}$$

$$c = \{0, 1, 2, \dots, (\text{Tile_k}-1)\}$$

OPTIMIZED IMPLEMENTATION OF CONVOLUTION ABSTRACTIONS

Precomputed delta tables invariance



Example size:

Tile_M = 128	Tile_N = 128
Tile_K = 32	R-by-S = 3-by-3
(pad_h, pad_w) = (1,1)	(stride_h, stride_w) = (1,1)
(dilation_h, dilation_w) = (1,1)	

Δ_s , Δ_r , and Δ_c are *invariants* for the entire kernel execution

OPTIMIZED IMPLEMENTATION OF CONVOLUTION ABSTRACTIONS

Mask predicates

Strategies:

1. Delta tables

- a) Offsets to access activation tensor
- b) Invariant for entire kernel-execution
- c) Reduced pointer arithmetic in the mainloop

2. Mask predicates

- a) 32b predicate vectors for OOB check
- b) Invariant for entire thread block
- c) Reduced logical arithmetic in the mainloop

```
Iterator(..., Element const* ptr, ...) : ptr(ptr)

Tensor4DCoord at() {
    // n and c are trivially available as Iterator state
    h_bar = p * stride_h - pad_h + r * dilation_h;
    b_bar = q * stride_w - pad_w + s * dilation_w;
    return Tensor4DCoord(n, h_bar, w_bar, c);
}

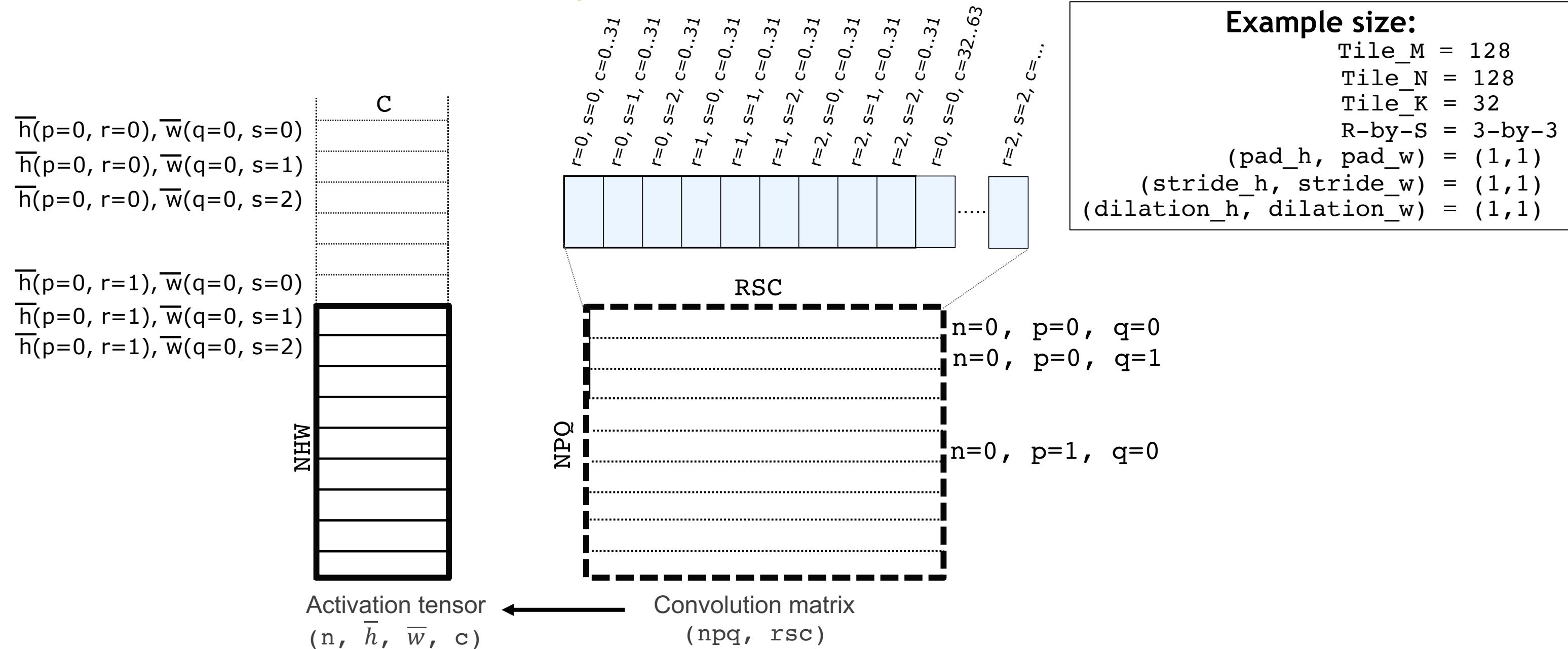
bool valid() {
    Tensor4DCoord coord = at();

    return (coord.n < N) &&
        (coord.h >=0 && coord.h < H) &&
        (coord.w >=0 && coord.w < W) &&
        (coord.c < C);
}

Element* get() {
    Tensor4DCoord coord = at();                                // Strides in NHWC tensor
    int64_t offset = coord.n * (n_stride) +                  // n_stride = HWC
                  coord.h * (h_stride) +                  // h_stride = WC
                  coord.w * (w_stride) +                  // w_stride = C
                  coord.c;                                // c_stride = 1
    return (ptr + offset);
}
```

OPTIMIZED IMPLEMENTATION OF CONVOLUTION ABSTRACTIONS

Mask predicates

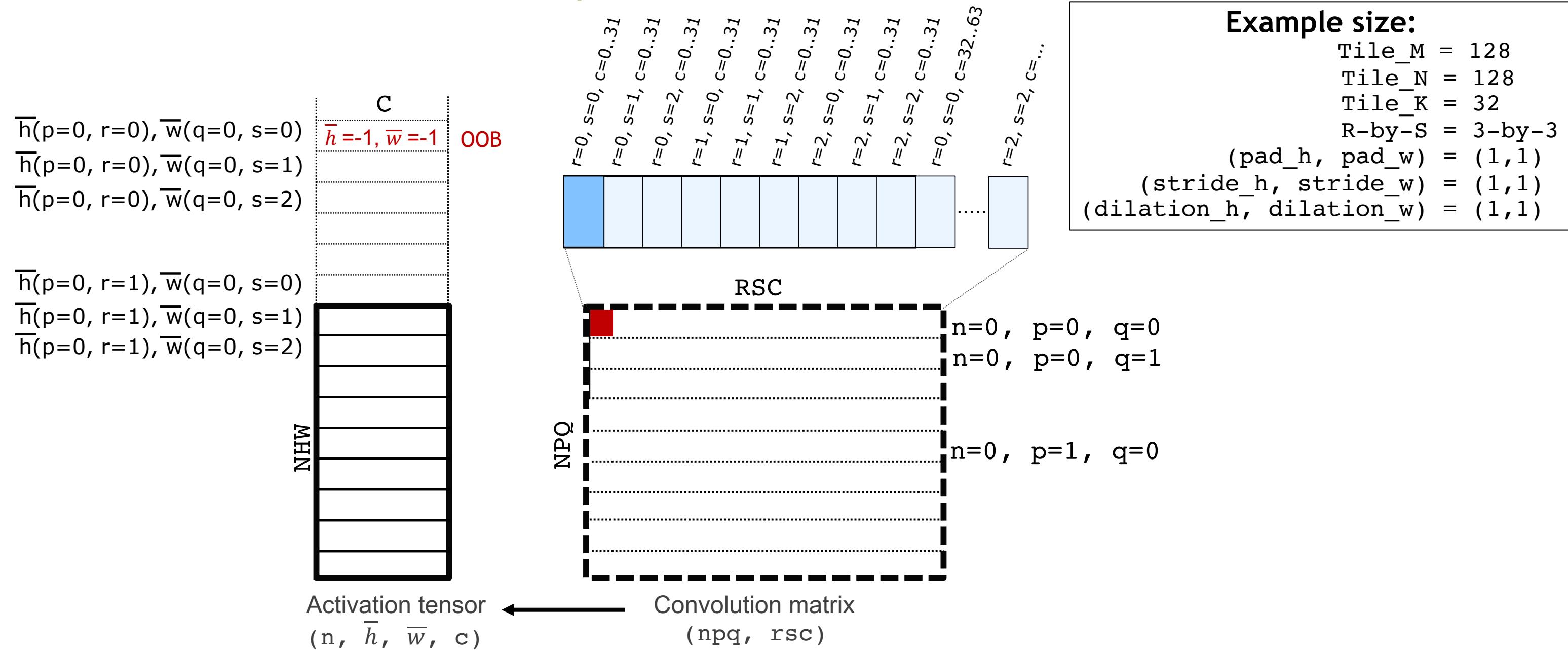


$$n = 0$$

$$\begin{aligned} \bar{h}(p, r) &= p * \text{stride}_h - \text{pad}_h + r * \text{dilation}_h \\ \bar{w}(q, s) &= q * \text{stride}_w - \text{pad}_w + s * \text{dilation}_w \\ c &= \{0, 1, 2, \dots, (\text{Tile}_k-1)\} \end{aligned}$$

OPTIMIZED IMPLEMENTATION OF CONVOLUTION ABSTRACTIONS

Mask predicates



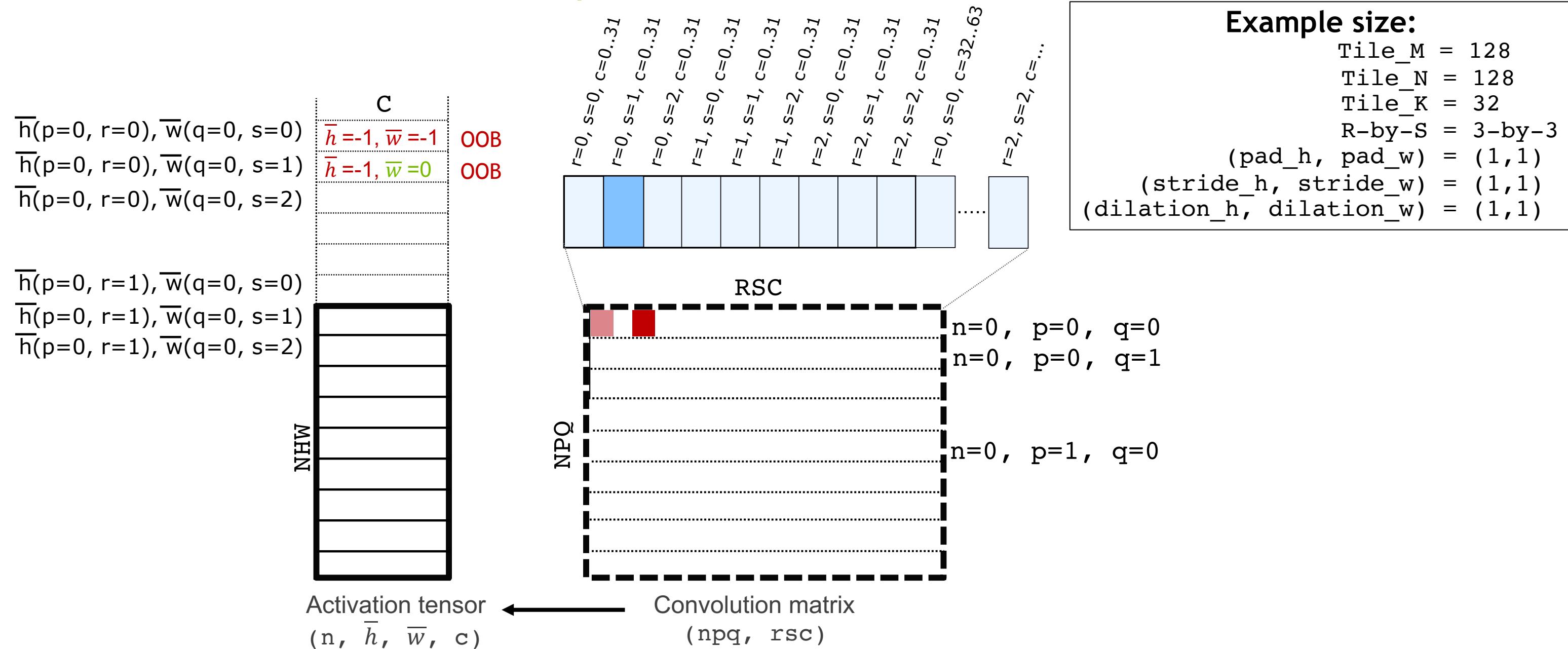
T0's first access for the first Tile_K iteration ($r=0, s=0, c=0-7$)

```

n = 0
 $\bar{h}(p, r) = p * \text{stride}_h - \text{pad}_h + r * \text{dilation}_h$ 
 $\bar{w}(q, s) = q * \text{stride}_w - \text{pad}_w + s * \text{dilation}_w$ 
c = {0, 1, 2, ..., (Tile_k-1)}
```

OPTIMIZED IMPLEMENTATION OF CONVOLUTION ABSTRACTIONS

Mask predicates



T0's first access for the second Tile_K iteration ($r=0, s=1, c=0-7$)

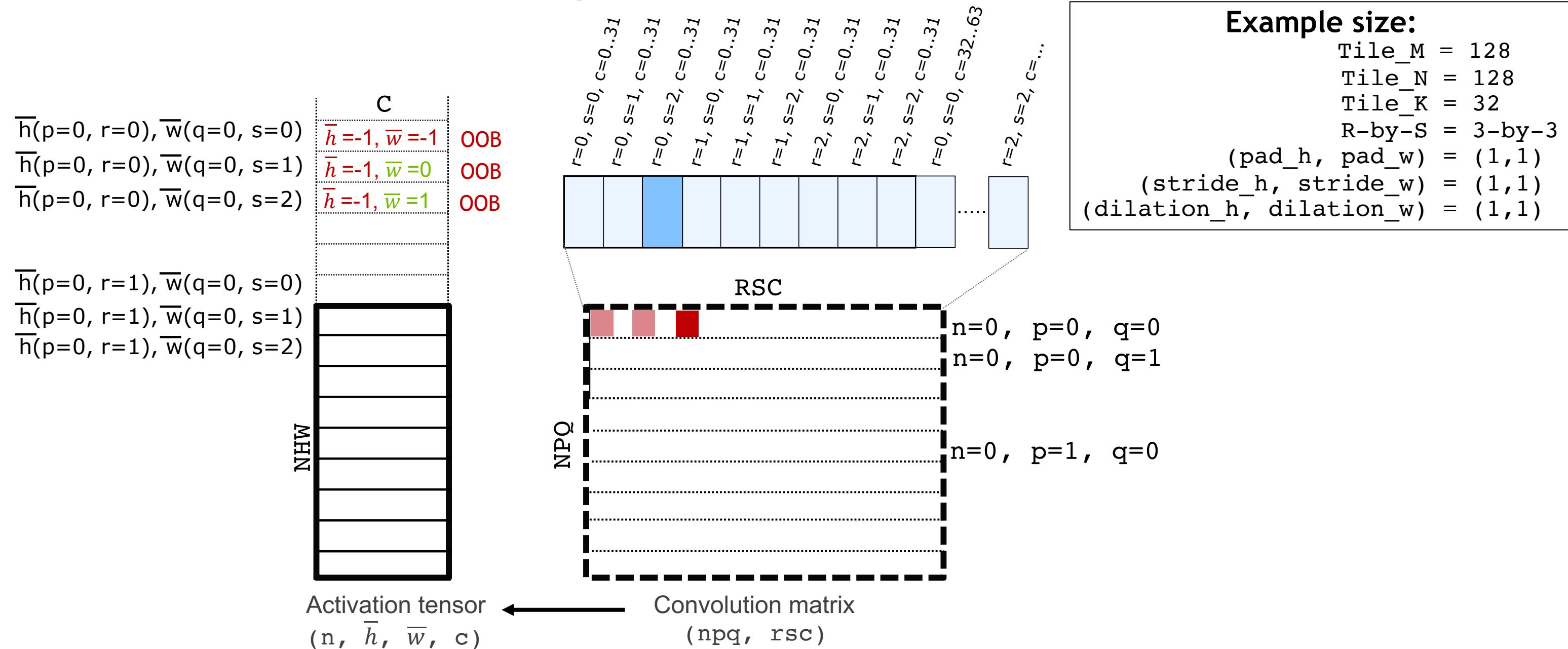
```

n = 0
 $\bar{h}(p, r) = p * \text{stride}_h - \text{pad}_h + r * \text{dilation}_h$ 
 $\bar{w}(q, s) = q * \text{stride}_w - \text{pad}_w + s * \text{dilation}_w$ 
c = {0, 1, 2, ..., (Tile_k-1)}

```

OPTIMIZED IMPLEMENTATION OF CONVOLUTION ABSTRACTIONS

Mask predicates



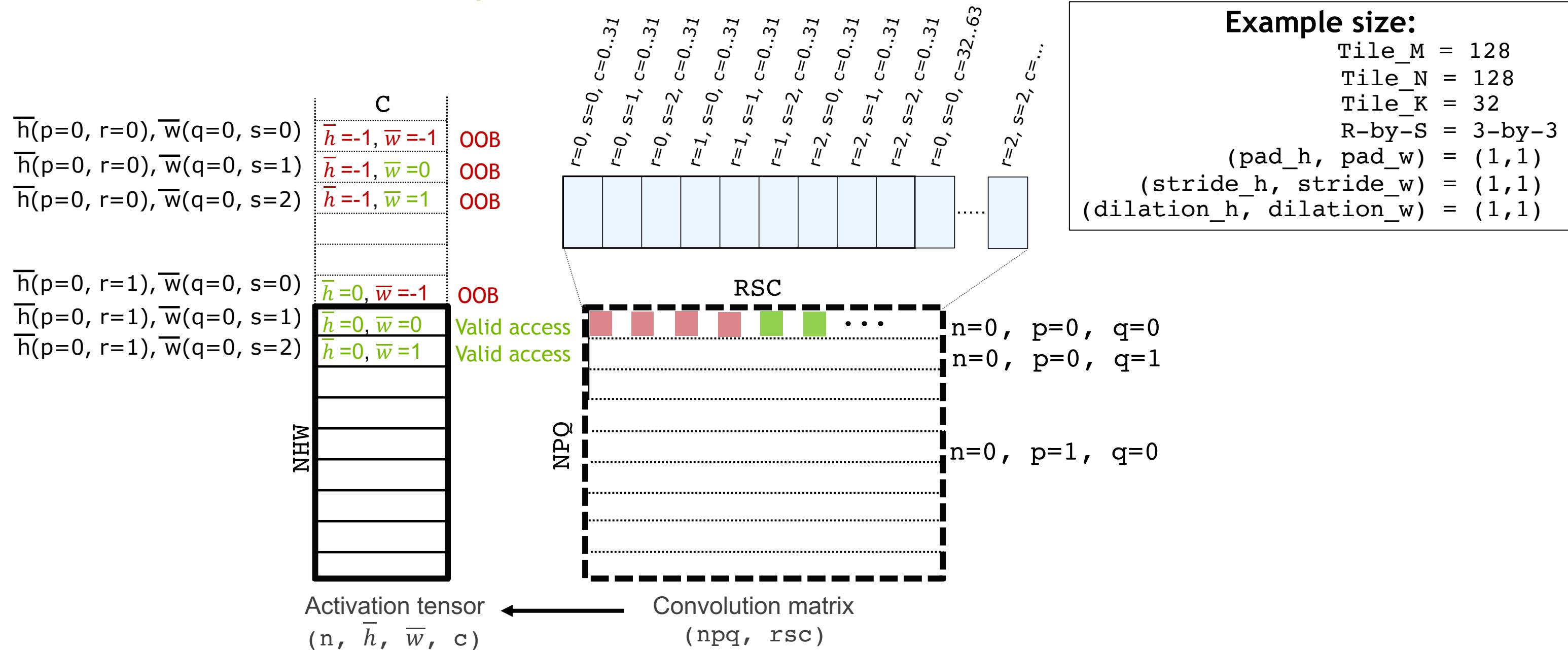
T0's first access for the third Tile_K iteration ($r=0, s=2, c=0..7$)

```

n = 0
 $\bar{h}(p, r) = p * \text{stride}_h - \text{pad}_h + r * \text{dilation}_h$ 
 $\bar{w}(q, s) = q * \text{stride}_w - \text{pad}_w + s * \text{dilation}_w$ 
c = {0, 1, 2, ..., (Tile_k-1)}
```

OPTIMIZED IMPLEMENTATION OF CONVOLUTION ABSTRACTIONS

Mask predicates invariance



T0's first access for the starting few Tile_K iterations

- Mask predicates are *invariant* for a *thread block*
- Mask predicates are precomputed and stored in a bit-vector
- One 32b bit-vector per thread, per access, per spatial filter dimension

IMPLICIT GEMM CONVOLUTION

Backward Data Gradient (Dgrad)

Backward Data Gradient (Dgrad)

$$\mathbf{dx} = \text{CONV}(\mathbf{dy}, \mathbf{w})$$

$\mathbf{dy}[N, P, Q, K]$: 4D output gradient tensor

$\mathbf{w}[K, R, S, C]$: 4D filter tensor

$\mathbf{dx}[N, H, W, C]$: 4D activation gradient tensor

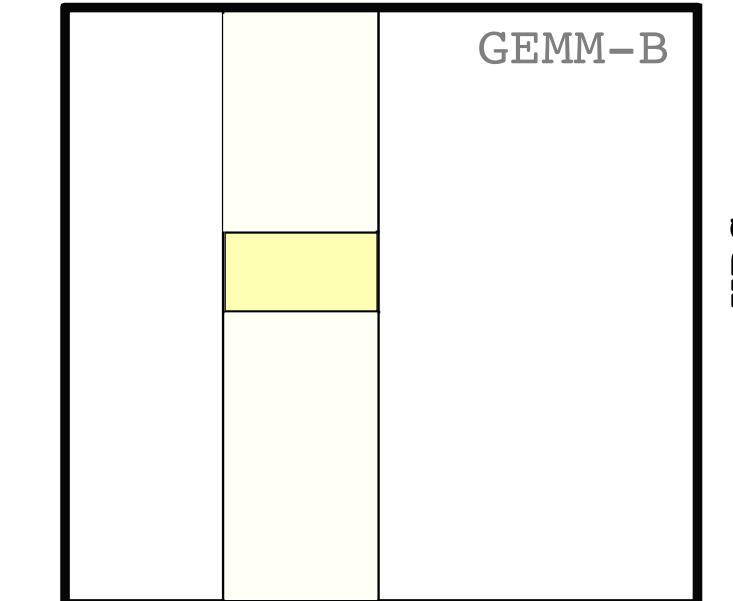
GEMM M-by-N-by-K dimensions

$$\text{GEMM-M} = \text{NHW}$$

$$\text{GEMM-N} = C$$

$$\text{GEMM-K} = \text{KRS}$$

Filter matrix (\mathbf{w})



KRS

KRS

C

GEMM-C

NHW

NHW

GEMM-A

K

NPQ

Output gradient tensor
in Global Memory (\mathbf{dy})

Convolution gradient matrix (\mathbf{dy}) Activation gradient matrix (\mathbf{dx})

IMPLICIT GEMM CONVOLUTION

Backward Weight Gradient (Wgrad)

Backward Weight Gradient (Wgrad)

$$dw = \text{CONV}(dy, x)$$

$dy[N, P, Q, K]$: 4D output gradient tensor

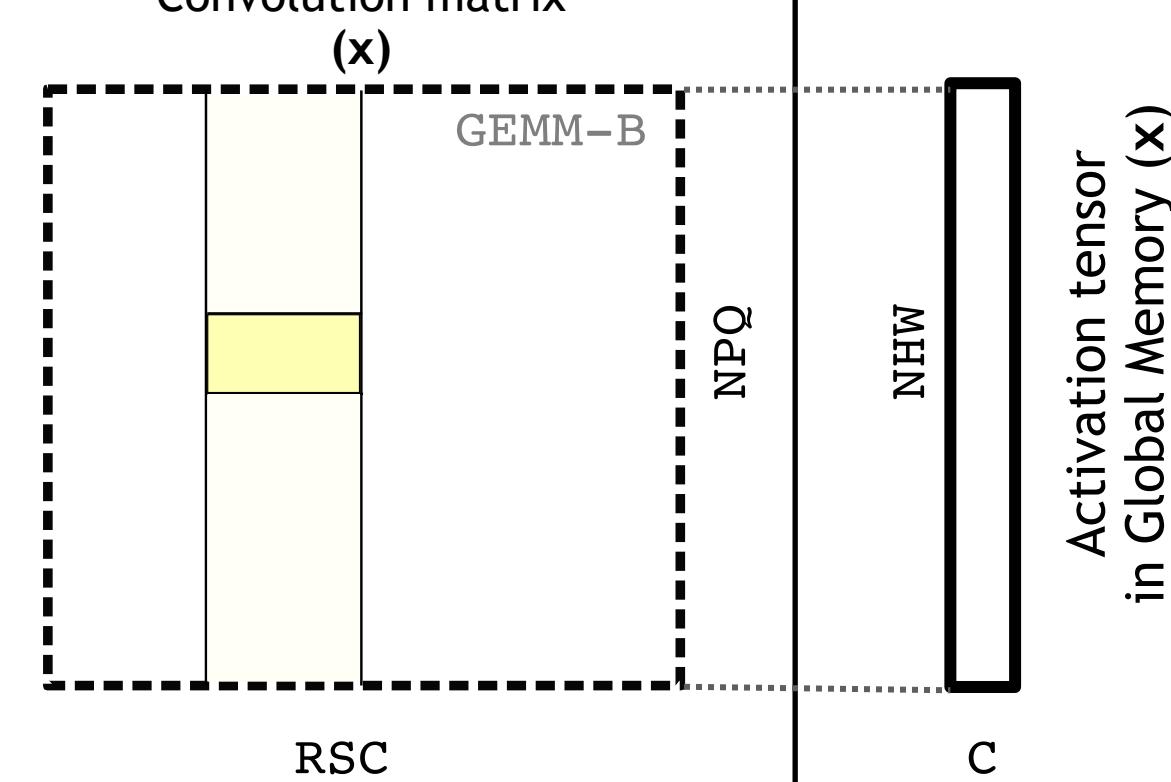
$x[N, H, W, C]$: 4D activation tensor

$dw[K, R, S, C]$: 4D filter tensor

GEMM M-by-N-by-K dimensions

GEMM-M = K
GEMM-N = RSC
GEMM-K = NPQ

Convolution matrix



NPQ

RSC

C

GEMM-A

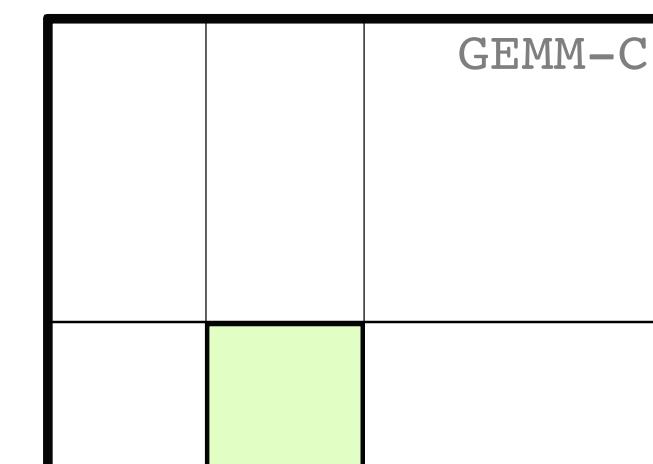
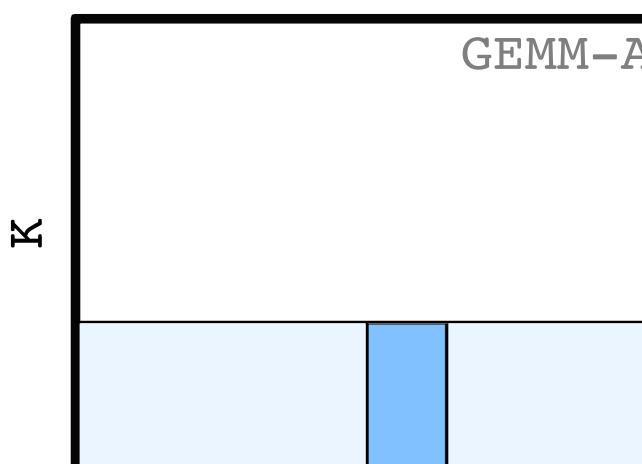
GEMM-C

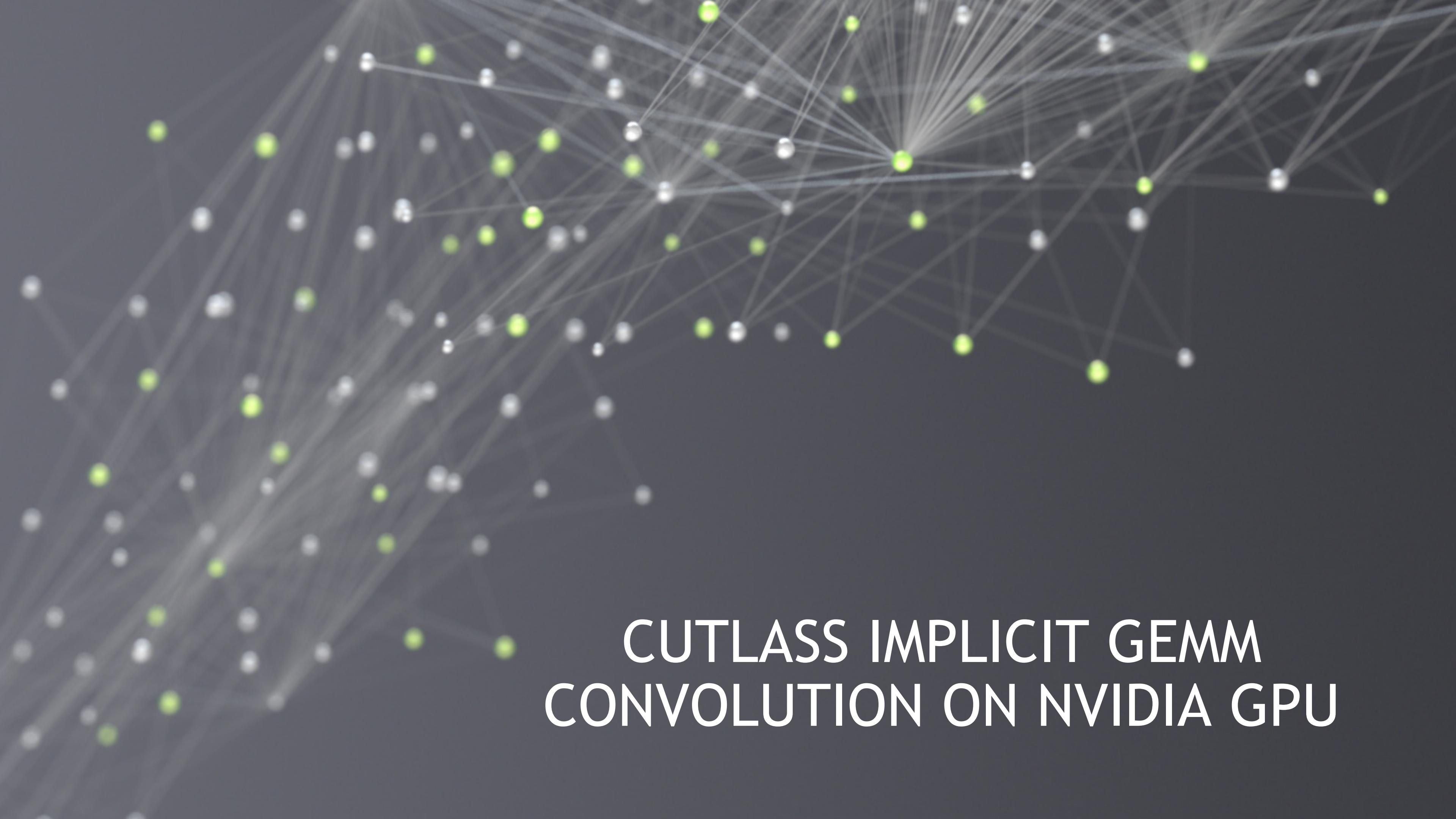
K

K

Output gradient matrix (dy)

Filter gradient matrix (dw)

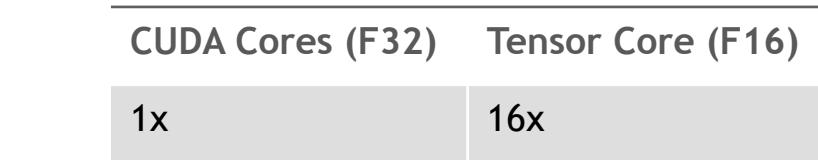




CUTLASS IMPLICIT GEMM CONVOLUTION ON NVIDIA GPU

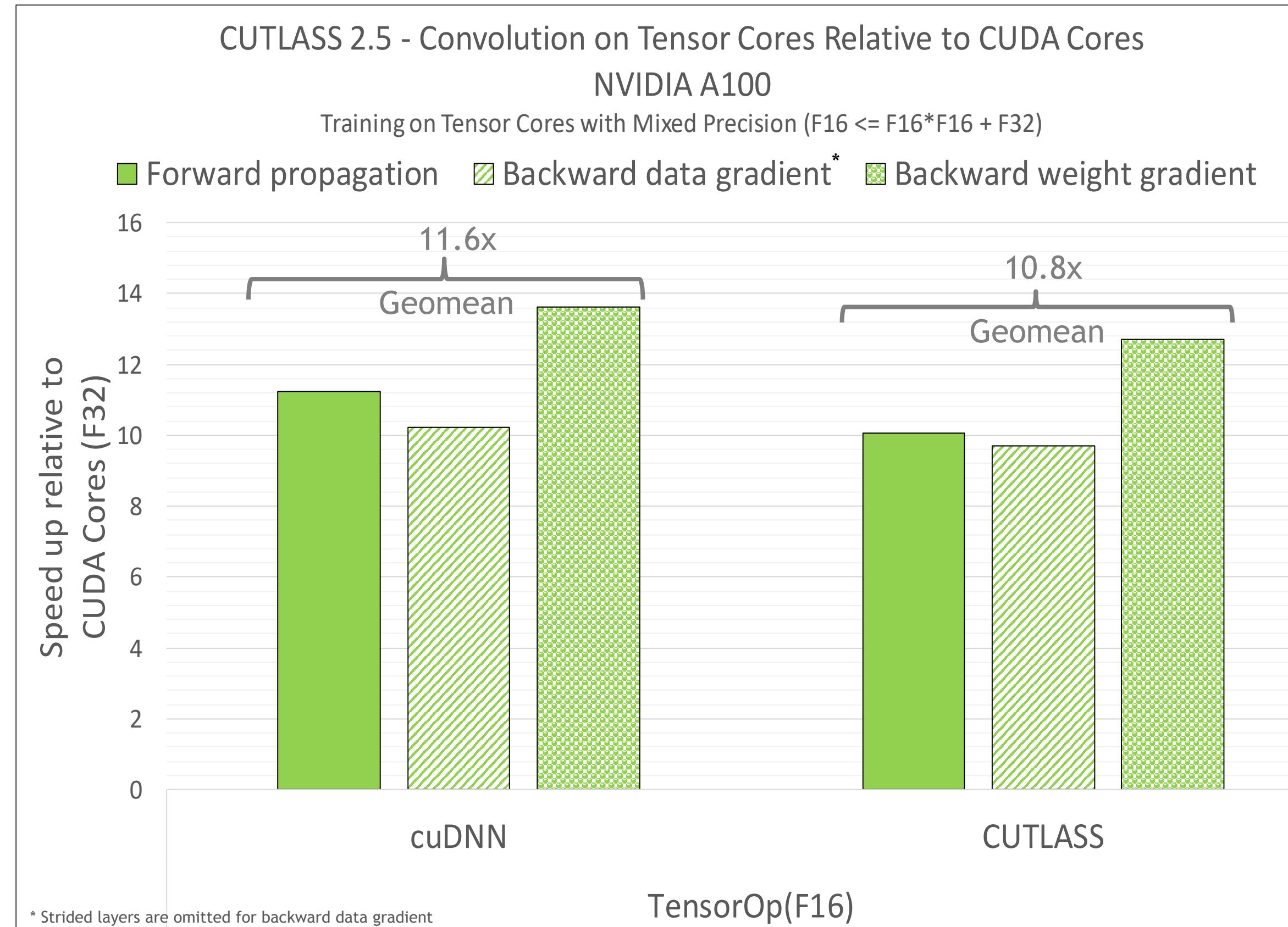
CUTLASS PERFORMANCE ON TENSOR CORES RELATIVE TO CUDA CORES (TRAINING)

Resnet50 Layers on Tensor Cores (F16 \leq F16*F16 + F32)



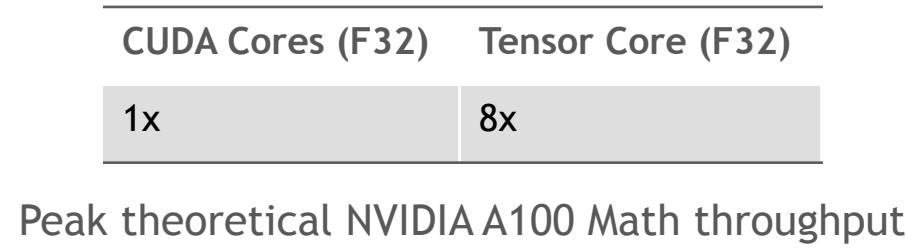
Peak theoretical NVIDIA A100 Math throughput

- Tensor Cores with F16 input is the fastest data type for training on NVIDIA A100
- CUTLASS performance is within 93% (10.8x/11.6x) of cuDNN for F16 input on NVIDIA A100

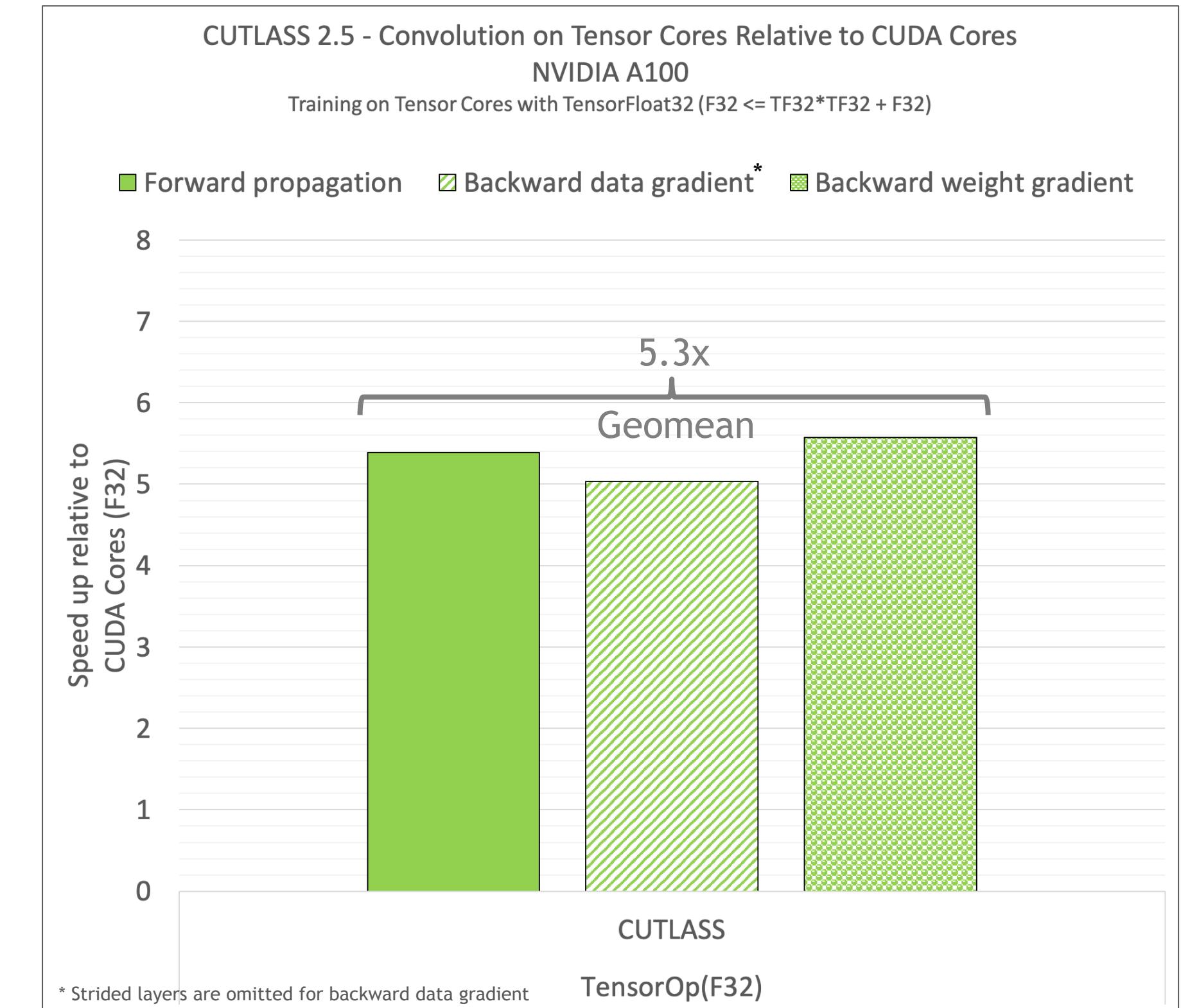


CUTLASS PERFORMANCE ON TENSOR CORES RELATIVE TO CUDA CORES (TRAINING)

Resnet50 Layers on Tensor Cores (F32 \leq TF32*TF32 + F32)



- TensorFloat32 (TF32) enables Tensor Cores directly on F32 input
- CUTLASS achieves 5.3x speed up with Tensor Cores relative to CUDA Cores on F32 input



CUTLASS CONVOLUTION PERFORMANCE RELATIVE TO CUDNN (F16[NHWC]) (INFERENCE)

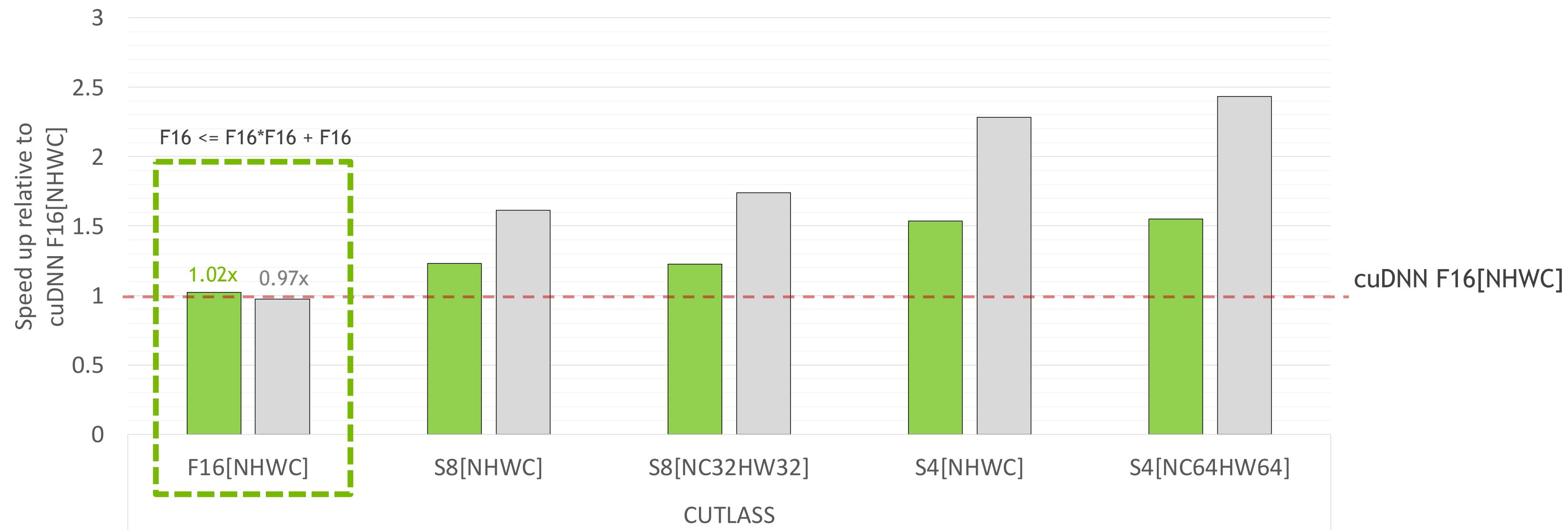
Resnet50 Layers on Tensor Cores

CUTLASS 2.5 - Performance Relative cuDNN

NVIDIA A100, 2080Ti - CUDA 11.3

Inference on Tensor Cores with F16, S8, and S4 data type

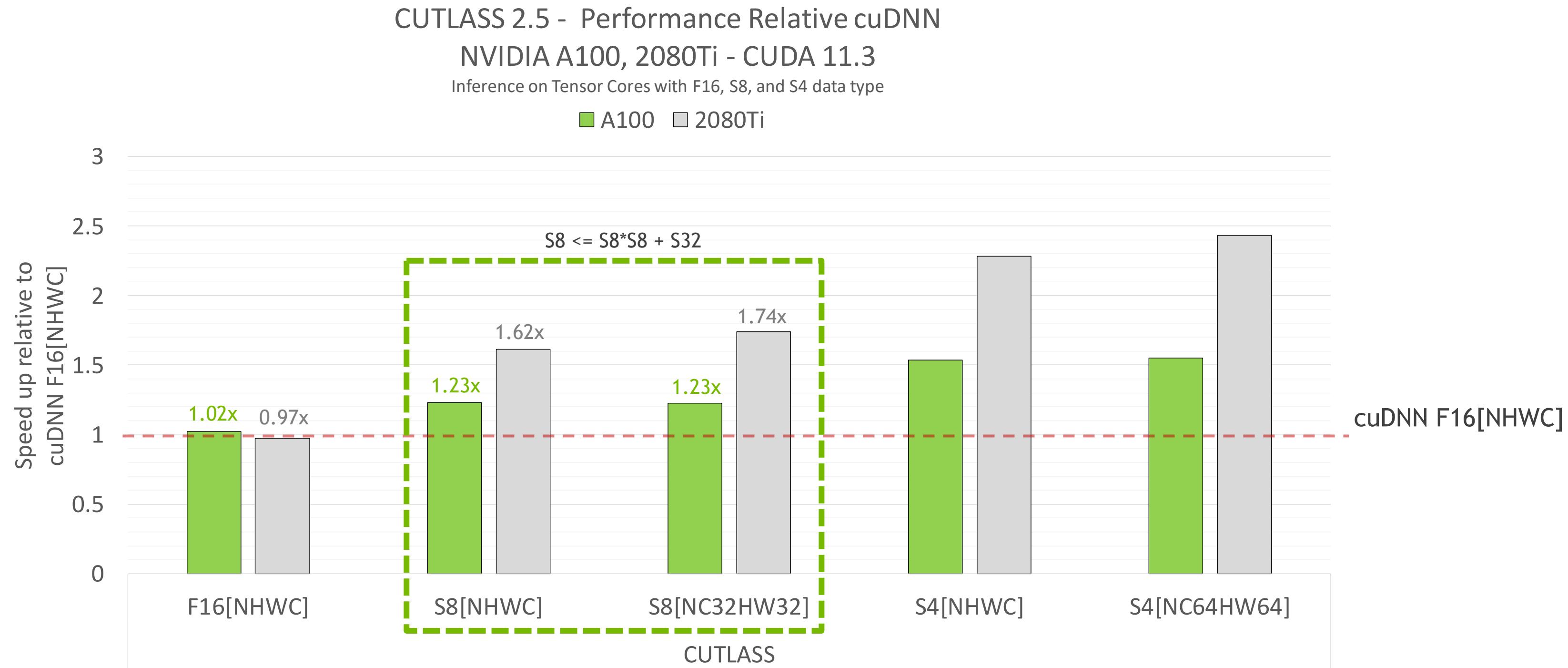
■ A100 □ 2080Ti



CUTLASS inference performance on Tensor Cores with F16 data type is on par with cuDNN for NVIDIA A100 and 2080Ti

CUTLASS CONVOLUTION PERFORMANCE RELATIVE TO CUDNN (F16[NHWC]) (INFERENCE)

Resnet50 Layers on Tensor Cores

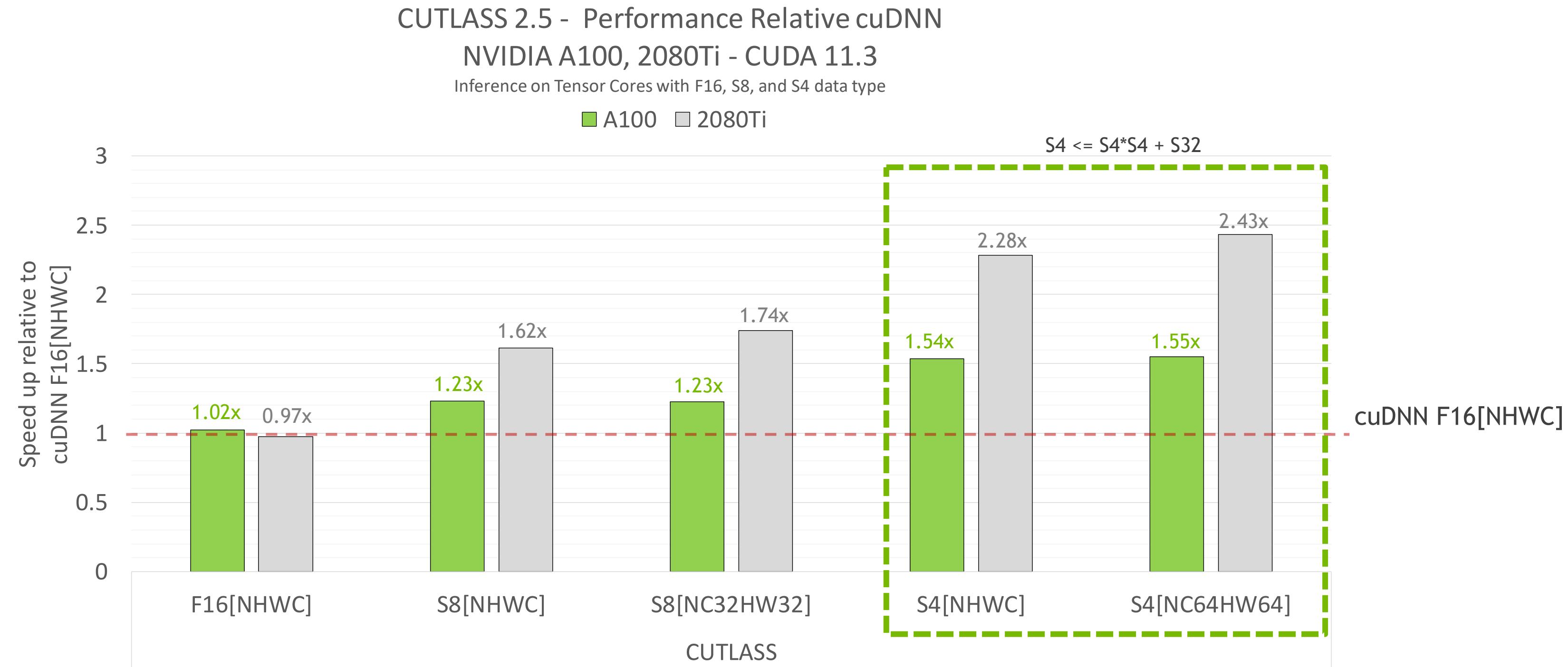


CUTLASS inference performance on Tensor Cores with S8 relative to F16 gains:

- 1.23x on A100 for both NHWC and NC32HW32 interleaved layout
- 1.62x and 1.74x on 2080Ti for NHWC and NC32HW32 interleaved layout, respectively

CUTLASS CONVOLUTION PERFORMANCE RELATIVE TO CUDNN (F16[NHWC]) (INFERENCE)

Resnet50 Layers on Tensor Cores



CUTLASS inference performance on Tensor Cores with S4 relative to F16 gains:

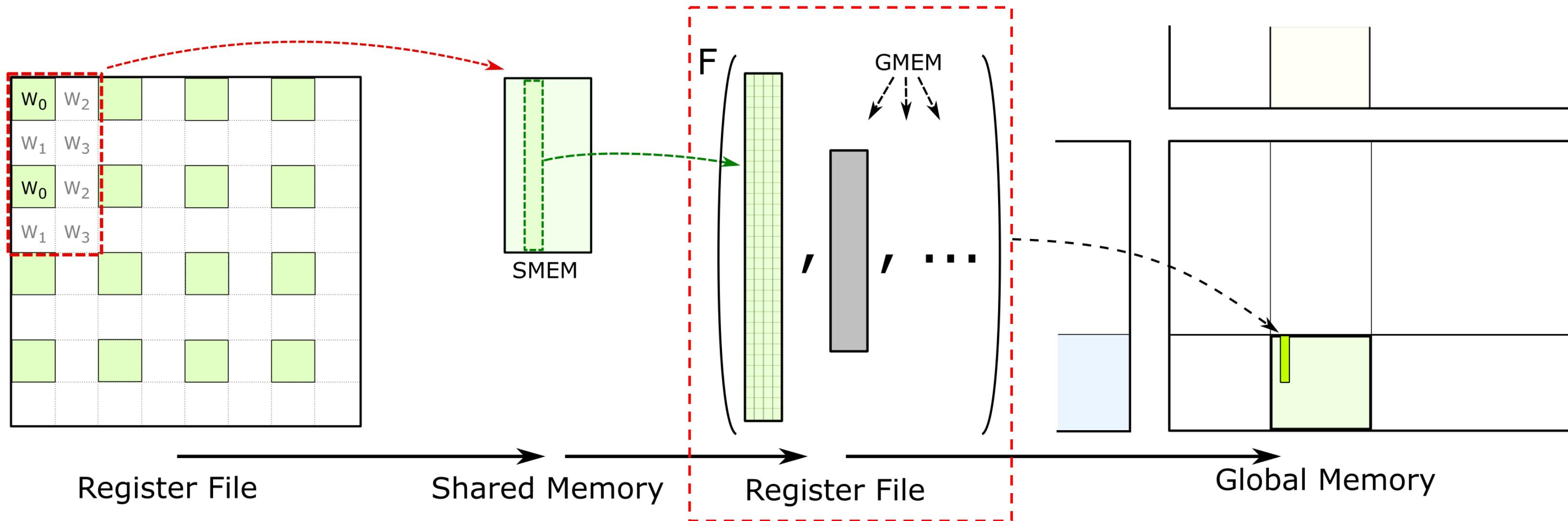
- 1.54x and 1.55x on A100 for NHWC and NC64HW64 interleaved layout, respectively
- 2.28x and 2.43x on 2080Ti for NHWC and NC64HW64 interleaved layout, respectively



EPILOGUE FUSION

EPILOGUE FUSION

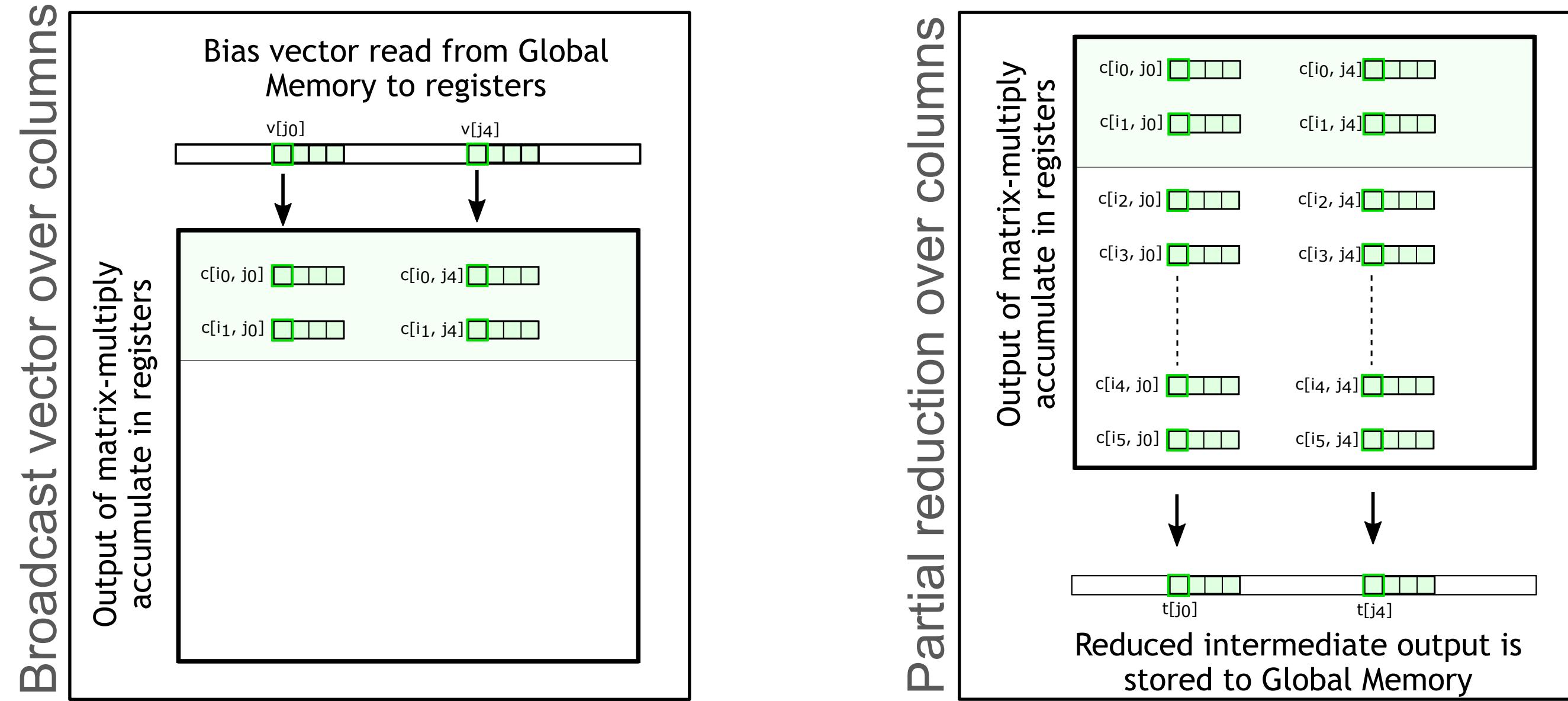
Epilogue fusion patterns supported in CUTLASS



Epilogue Fusion Pattern	Application
Element-wise operators	Scale, bias, activations
Data type conversion	F32->F16, Int32->Int8
Broadcast vector over columns	Bias add
Partial reduction over columns	Sum or sum-of-squares for batch norm

EPILOGUE FUSION

Epilogue fusion patterns supported in CUTLASS



Epilogue Fusion Pattern	Application
Element-wise operators	scale, bias, activations
Data type conversion	F32->F16, Int32->Int8
Broadcast vector over columns	bias add
Partial reduction over columns	sum or sum-of-squares for batch norm

Available in
CUTLASS 2.5

Lookout for
CUTLASS 2.6



CONCLUSION

CONCLUSION: IMPLICIT GEMM CONVOLUTION FOR NVIDIA GPU

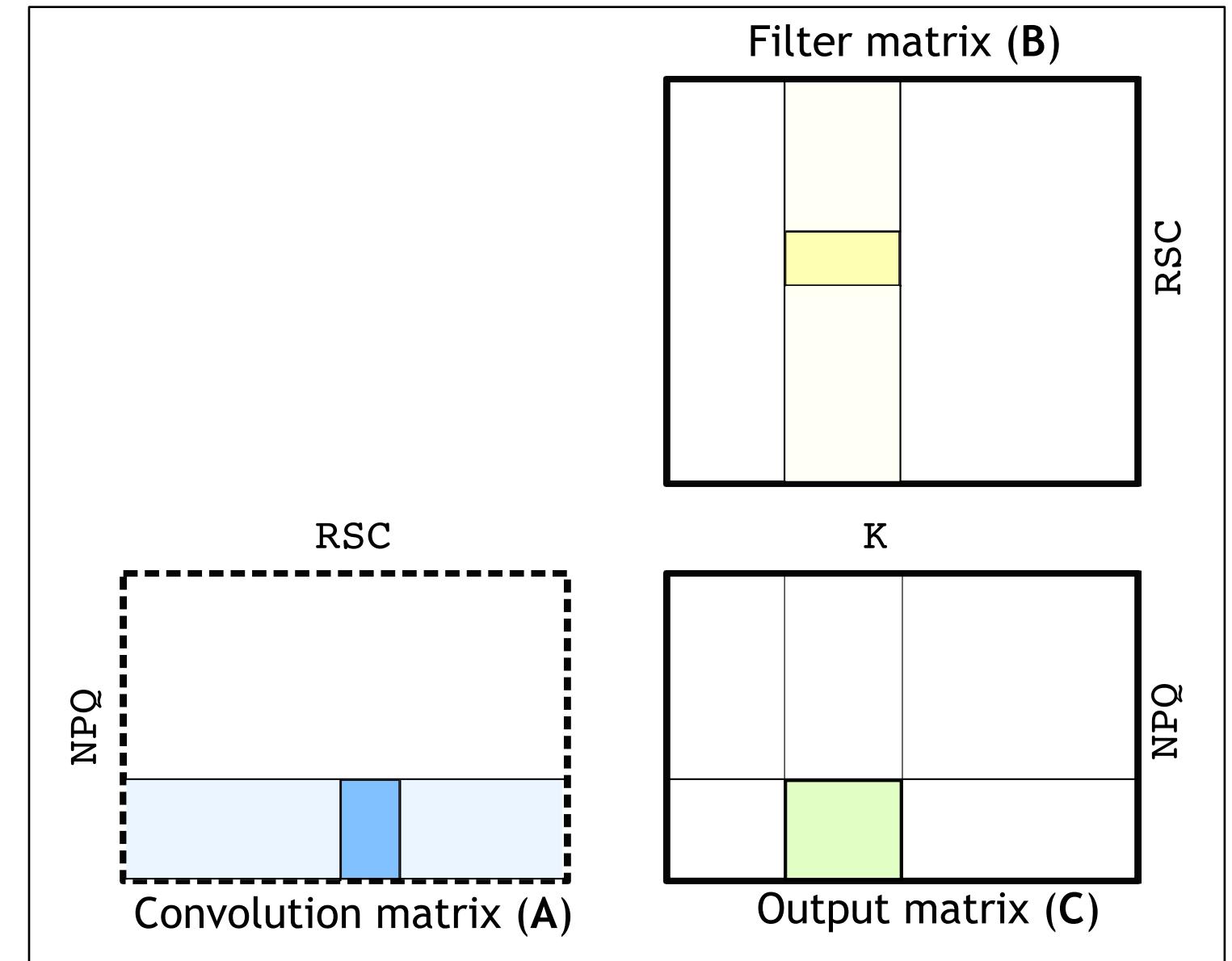
CUTLASS 2.4 and 2.5: Nov 2020 and Feb 2021

- Open source CUDA C++ template library for CUDA development
- Reusable building blocks for utilizing NVIDIA Tensor Cores for GEMMs and convolutions
- CUTLASS convolution performance is on par with cuDNN (> 90%)

CUTLASS 2.6: Upcoming release lookout for it ☺

- Support for new epilogue fusion patterns
 - Broadcast vector over column (Bias add)
 - Partial reduction over column (Batch normalization)

Get started now! <https://github.com/NVIDIA/cutlass>



REFERENCES

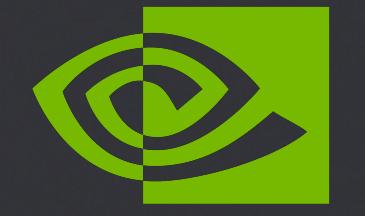
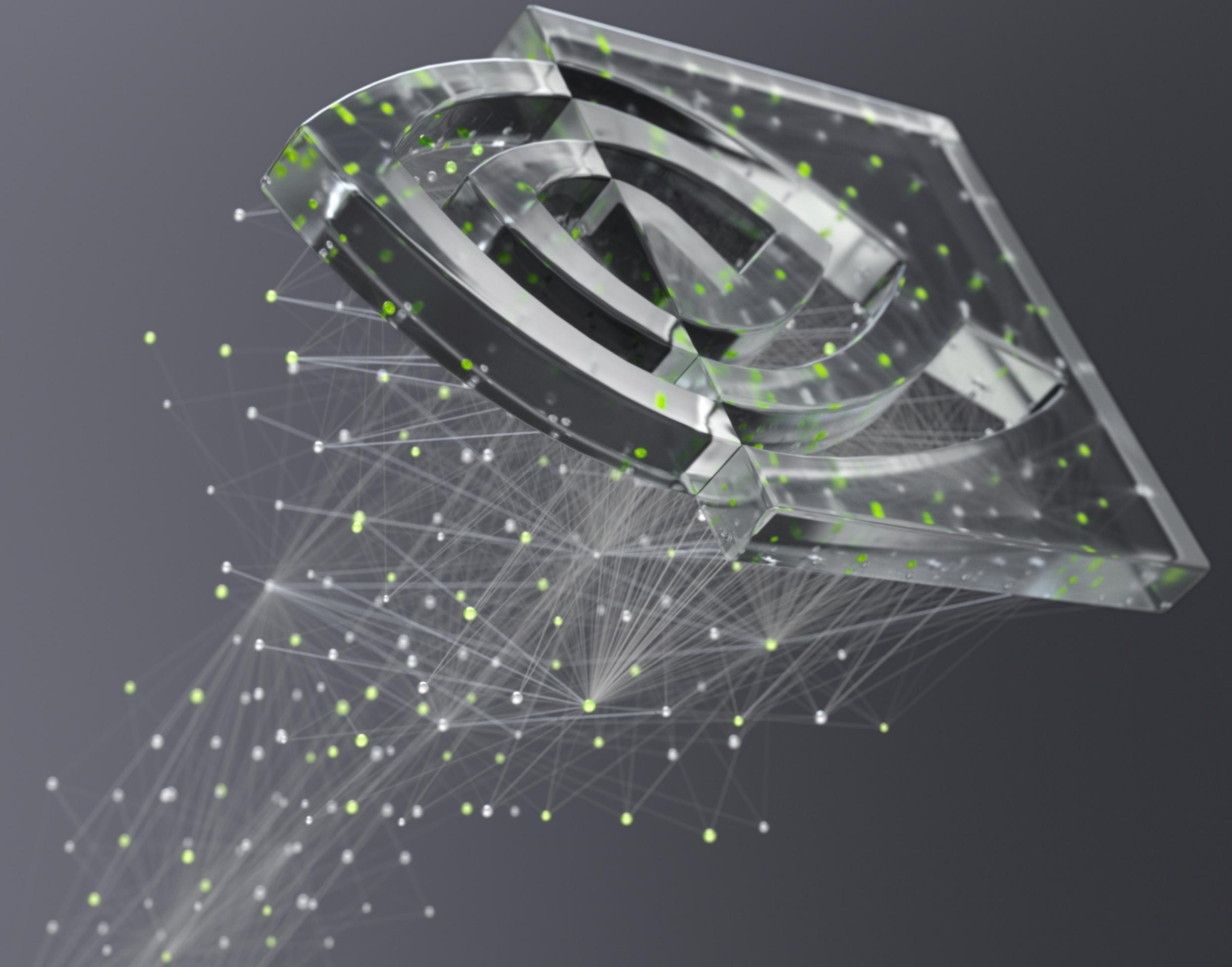
CUTLASS (<https://github.com/NVIDIA/cutlass>)

GTC 2018 talk (S8854) : CUTLASS: Software primitives for dense linear algebra at all levels and scales within CUDA

GTC 2019 talk (S9593) : cuTENSOR:High-performance Tensor Operations in CUDA (joint talk with cuTENSOR)

GTC 2020 talk (S21745) : Developing CUDA kernels to push Tensor Cores to the Absolute Limit on NVIDIA A100

CUTLASS Parallel For All blog post



NVIDIA®