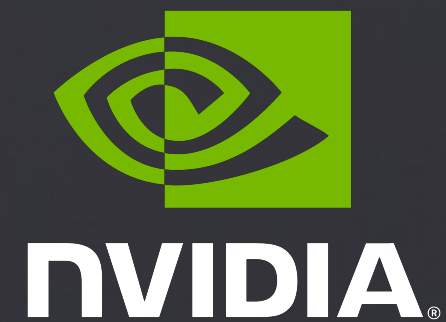
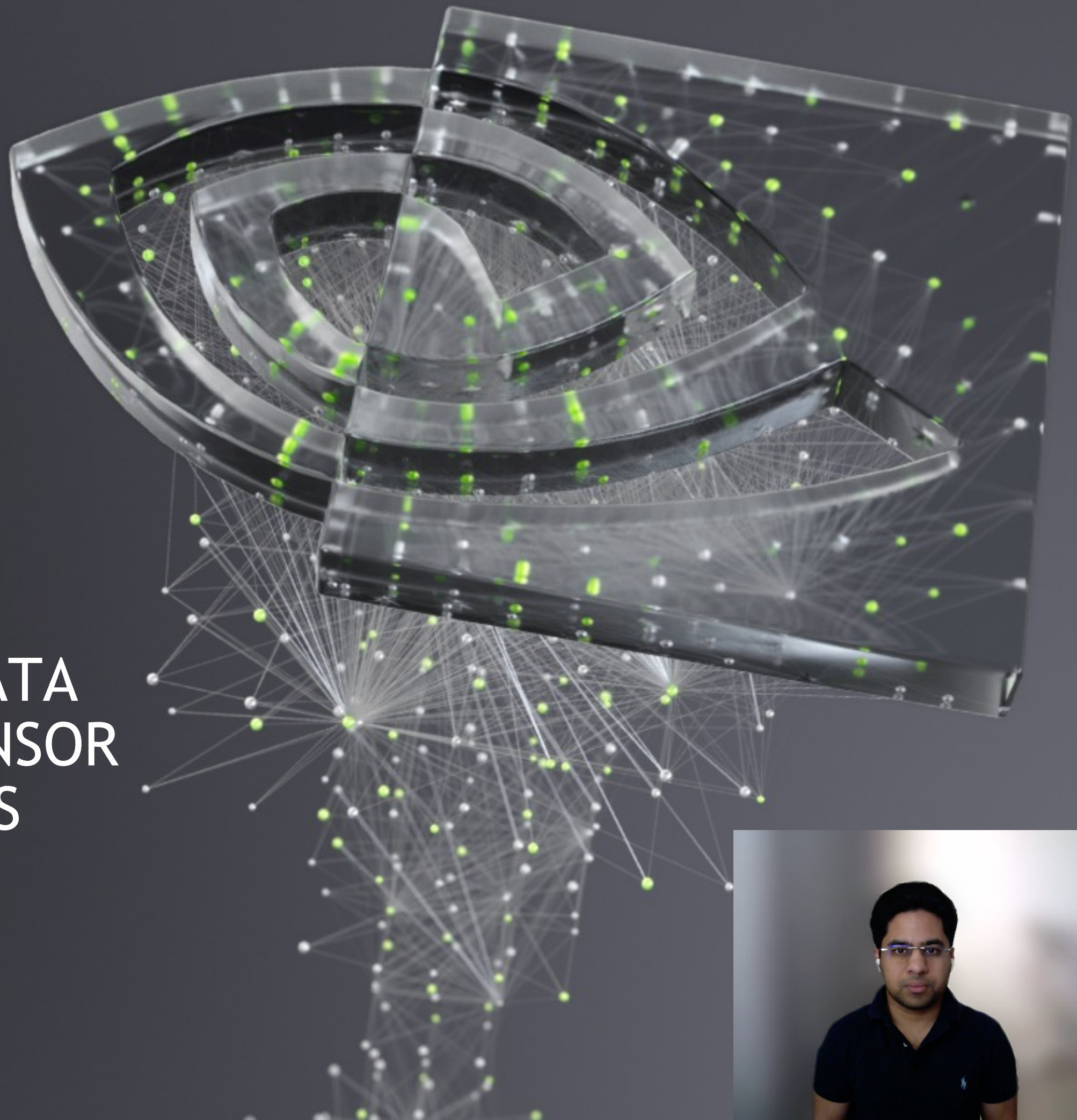




ACCELERATING BACKWARD DATA GRADIENT BY INCREASING TENSOR CORE UTILIZATION IN CUTLASS

Manish Gupta, March 21, 2021



A background image featuring a complex network diagram with numerous white and yellow nodes connected by thin white lines, set against a dark grey background.

ACKNOWLEDGEMENTS

CUTLASS GitHub Community

75K clones/month, 1.6K stars, and many active users and customers

CUTLASS Team

Developers

Andrew Kerr, Haicheng Wu, Manish Gupta, Dustyn Blasig, Duane Merrill, Pradeep Ramani, Aniket Shivam

Product Management

Matthew Nicely, Vartika Singh, Timothy Costa

Contributors

Cris Ceka, Vijay Thakkar, Naila Farooqui, Markus Hohnerbach, Alan Kaatz, Wei Liu, Piotr Majcher, Gautam Jain, Dhiraj Reddy Nallapa, Kyrylo Perelygin, Paul Springer, Pawel Tabaszewski, Chinmay Talegaonkar, John Tran, Jin Wang, Yang Xu, Scott Yokim, Jack Chen

Acknowledgements

Olivier Giroux, Mostafa Hagog, Bryce Lebach, Julien Demouth, Joel McCormack, Artem 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 2.6 (GTC March 2021) - CUTLASS 2.9 (GTC March 2022)

Convolution Performance (CUTLASS 2.9)

Fprop, Dgrad, and Wgrad performance with CUDA 11.6

Strided Dgrad

Naïve strided dgrad (does redundant MMAs)

CUTLASS strided dgrad (removes redundant MMAs)

Implicit Gemm Convolutions

Building coherent and complete abstractions

Abstractions to put together complex algorithms efficiently

Do More with CUTLASS

Accelerated single-precision arithmetic using Tensor Cores (3xTF32)

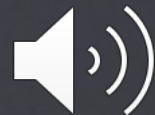
Grouped GEMM

CUTLASS Python Example



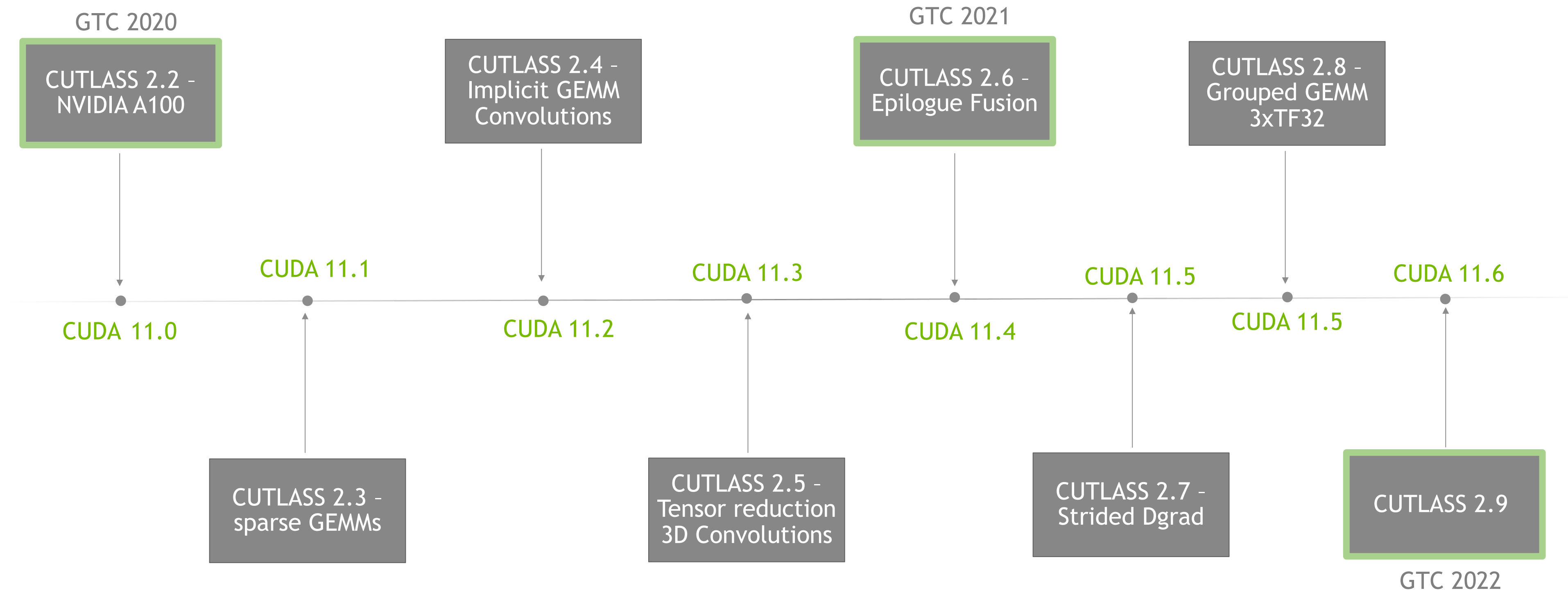


OVERVIEW



CUTLASS

CUDA C++ Templates for Deep Learning and Linear Algebra



CUTLASS

What's new since we last met?

CUTLASS 2.6 - Jul 2021

- New and improved strided dgrad (Analytic Iterators)
- Epilogue fusion pattern

CUTLASS 2.7 - Sep 2021

- New and improved strided dgrad (Optimized Iterators)
- Half-precision GELU_taylor activation function
- Performance tuning for fused GEMM + GEMM example
- Support for smaller than 128b aligned Convolutions

CUTLASS 2.8 - Nov 2021

- Accelerated single-precision arithmetic using Tensor Cores (3xTF32)
- Variable-sized GEMM problem size for each batch (Grouped GEMM)

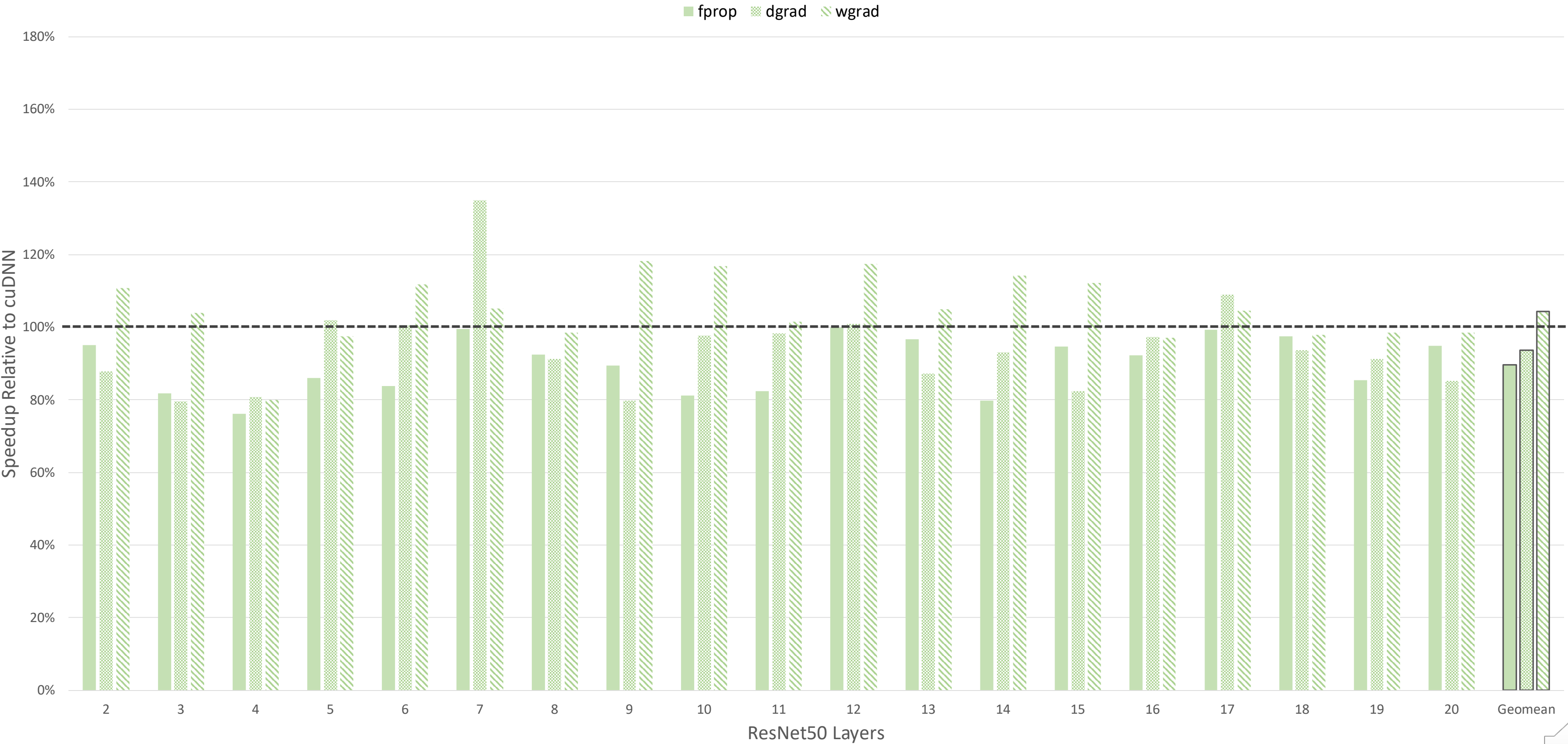
CUTLASS 2.9 - Upcoming release

- First layer convolution
- CUTLASS python example
- Many more ...



CUTLASS CONVOLUTION PERFORMANCE

CUTLASS 2.8 - Performance Relative to cuDNN on NVIDIA A100 - CUDA 11.6
Mixed Precision Training ($F16 \leftarrow F16 * F16 + F32$)



* For the best performance use CUDA 11.4 or later



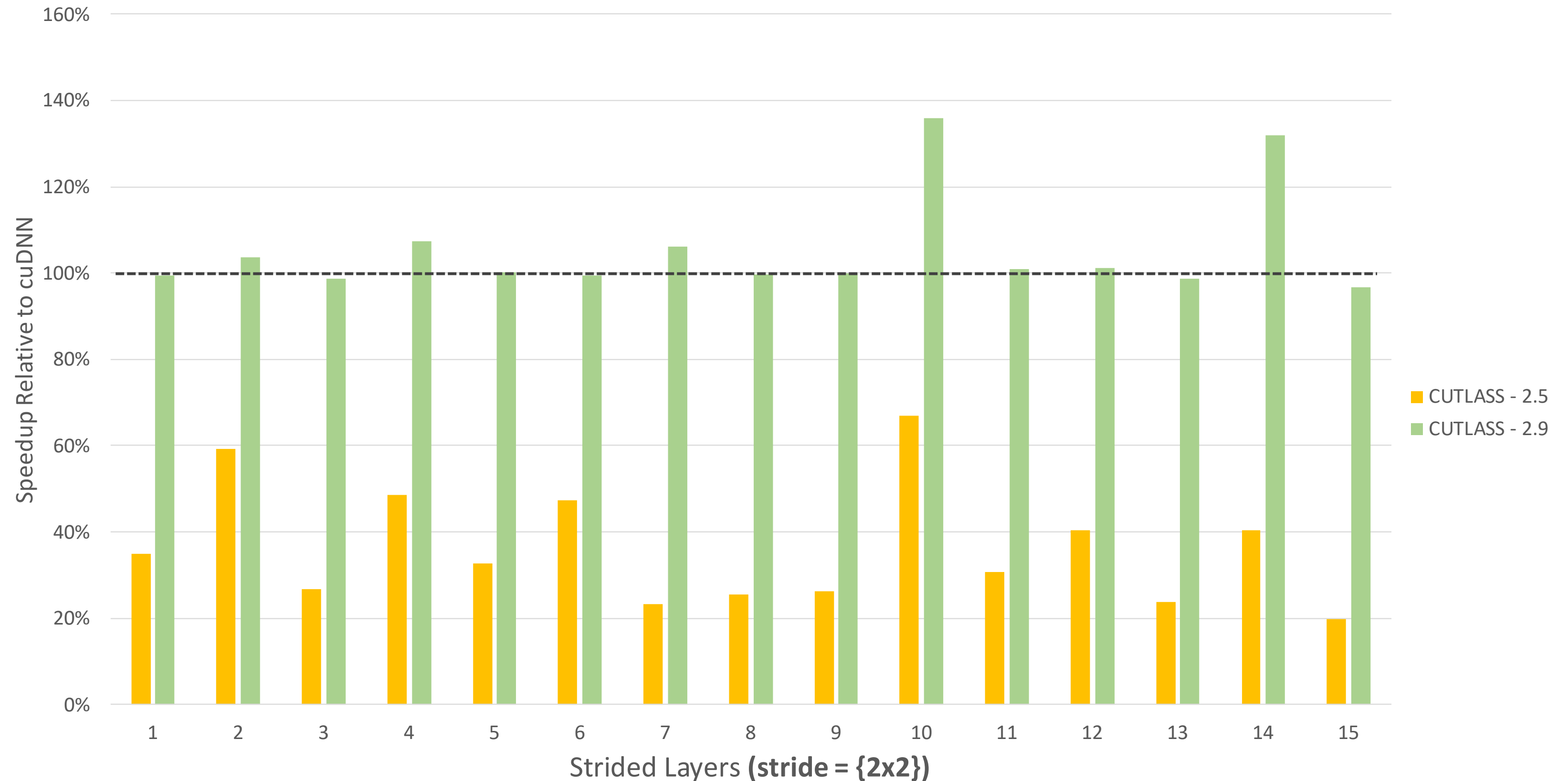


CUTLASS STRIDED BACKWARD DATA GRADIENT (DGRAD)



CUTLASS 2.5 VS. CUTLASS 2.8 (STRIDED DGRAD)

CUTLASS Strided Dgrad Performance vs. cuDNN 8.3.3.21 with CUDA 11.6 Toolkit
NVIDIA GA100@1005Mhz , TensorOp (F16 $\leftarrow$ F16*F16+F32)



Layer 1-6: [ResNet50, 1x1 filter] - Layer 7-9: [RNXT JoC, 3x3 filter] - Layer 10-15: [MaskRCNN MLPERF, 1x1 filter]

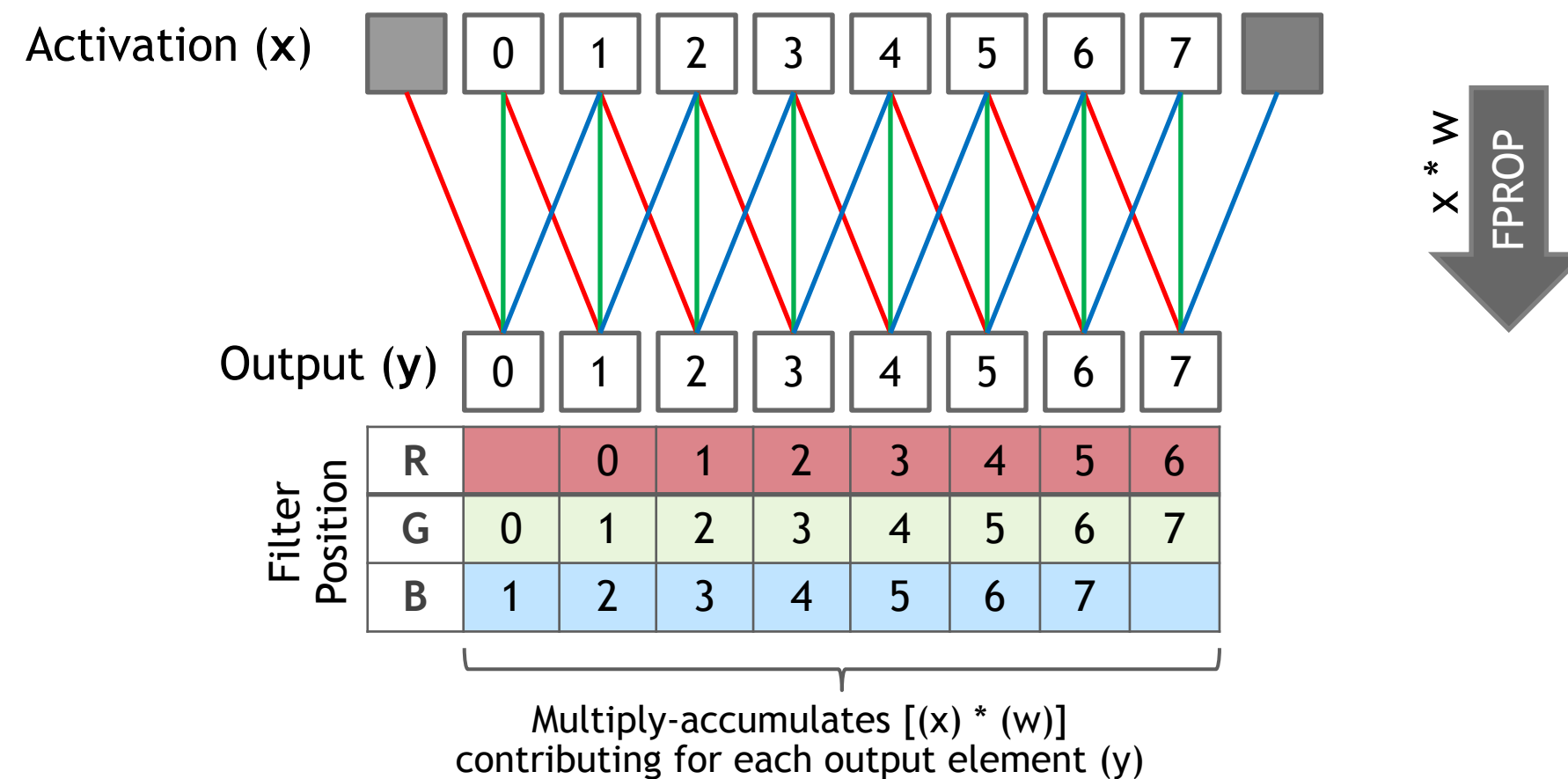


STRIDED DGRAD (UNDERSTANDING IN 1D)



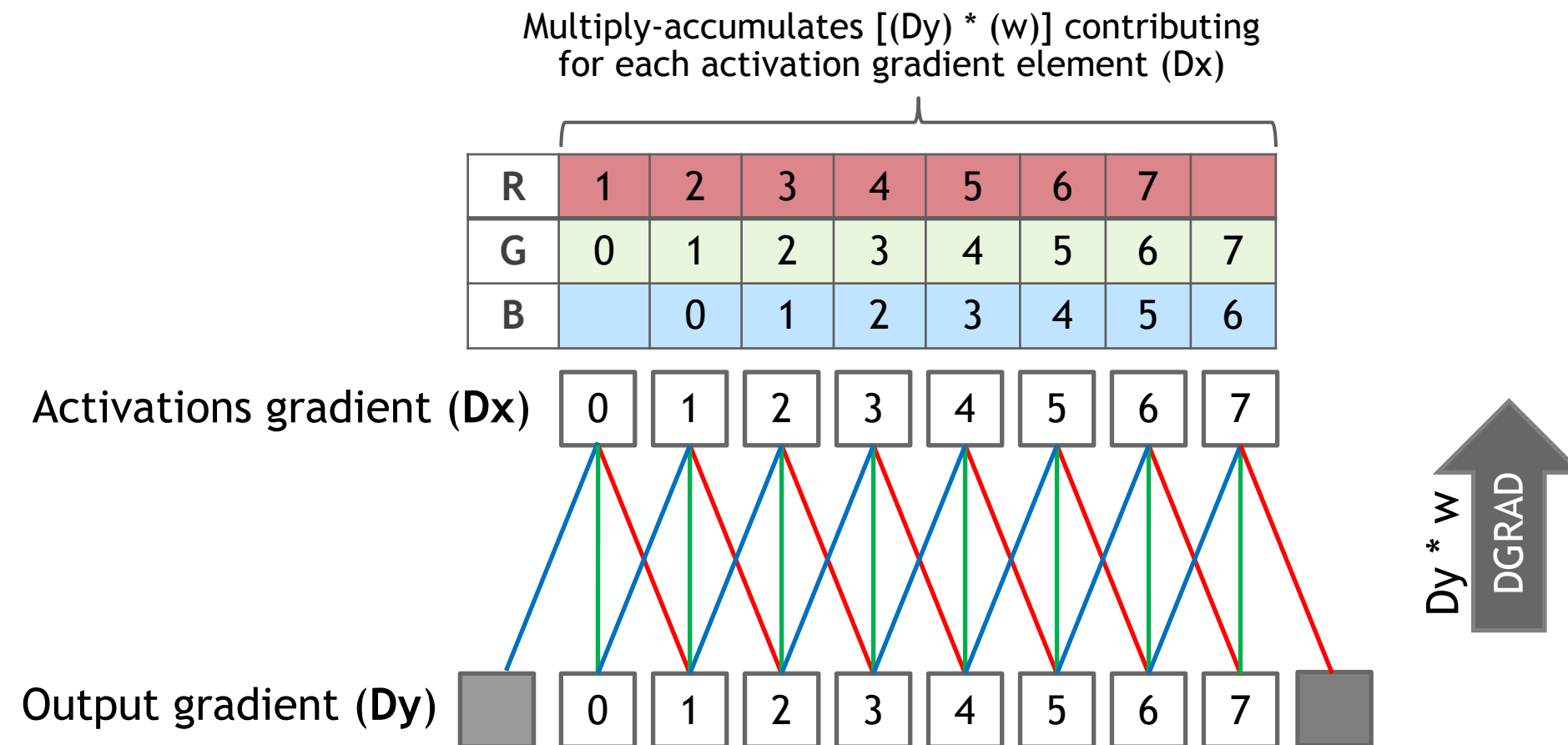
FORWARD PROPAGATION | $Y = \text{FPROP}(X, W)$

1D Convolution (Stride = 1, Filter = 3)



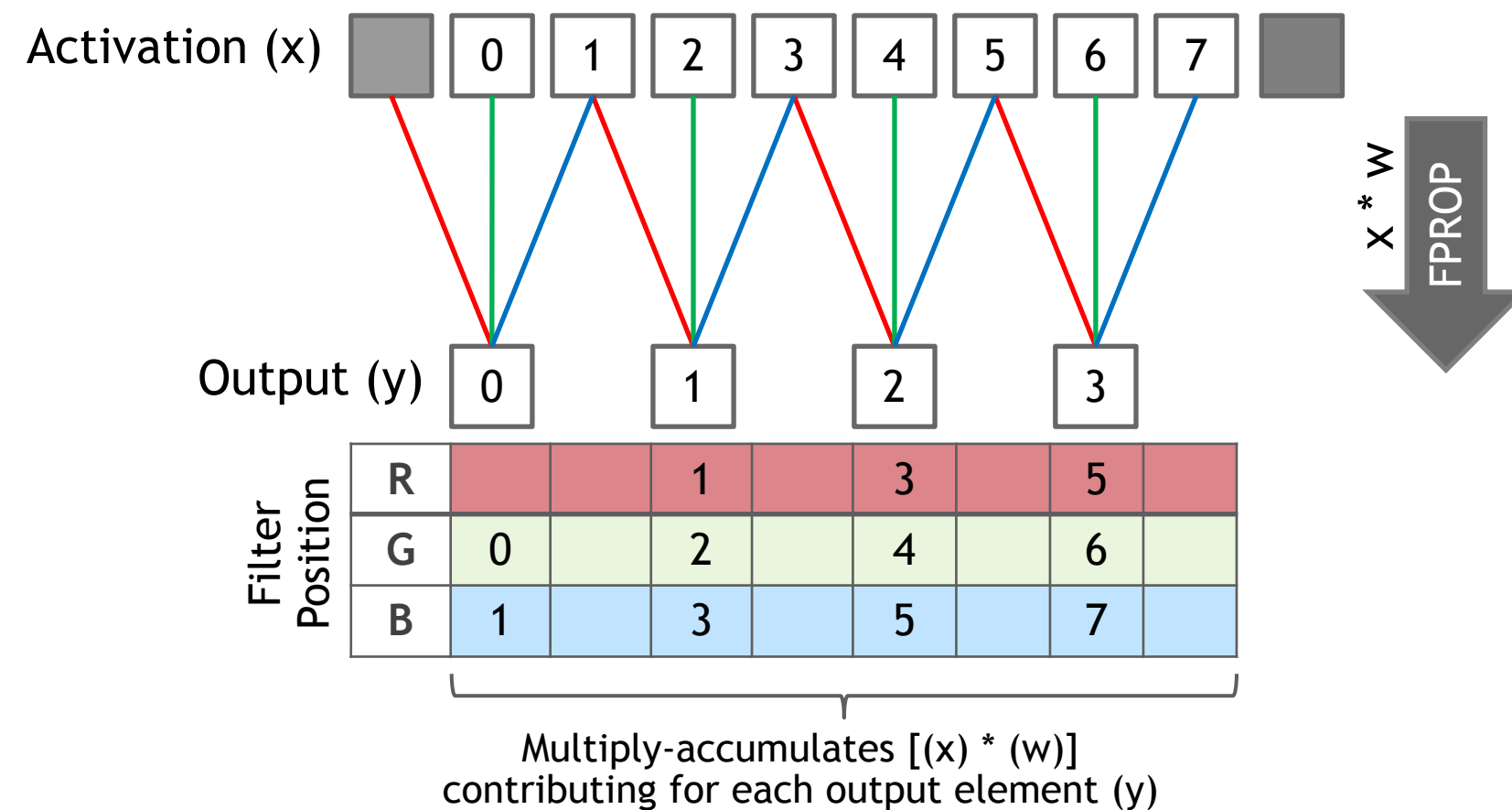
BACKWARD DATA GRADIENT | $DX = DGRAD(DY, W)$

1D Convolution (Stride = 1, Filter = 3)



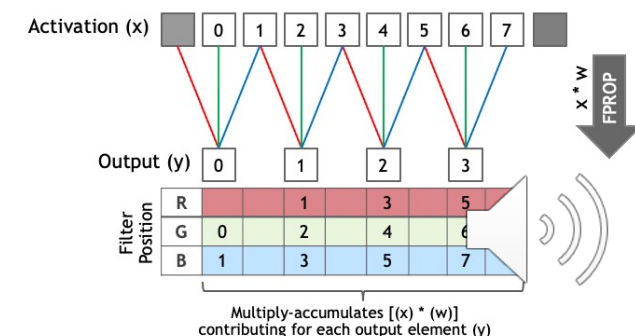
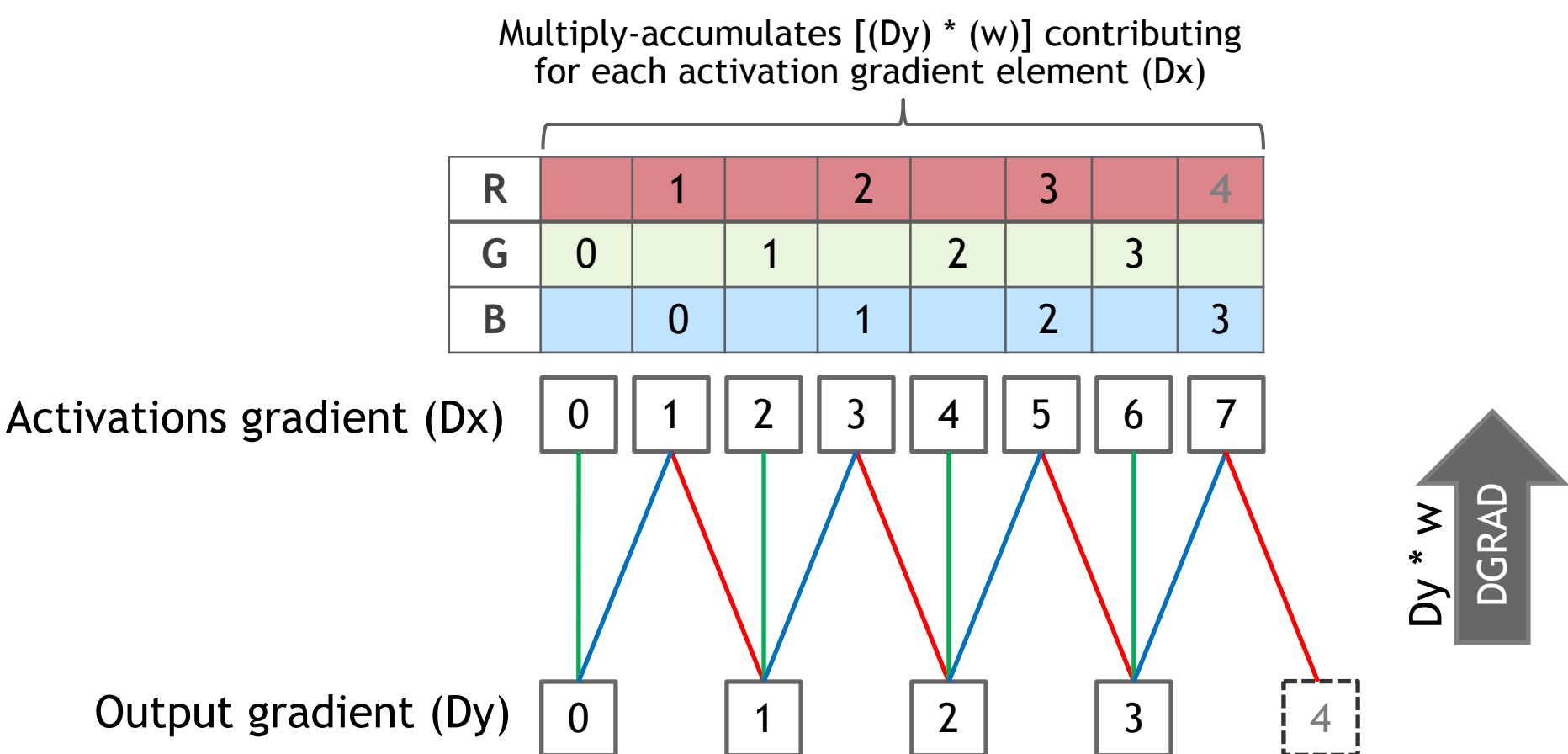
STRIDED FORWARD PROPAGATION | $Y = \text{FPROP}(X, W)$

1D Convolution (Stride = 2, Filter = 3)



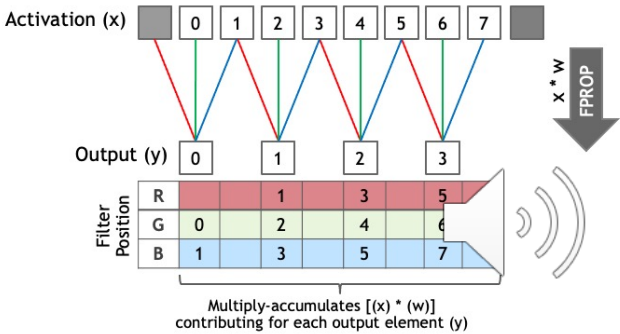
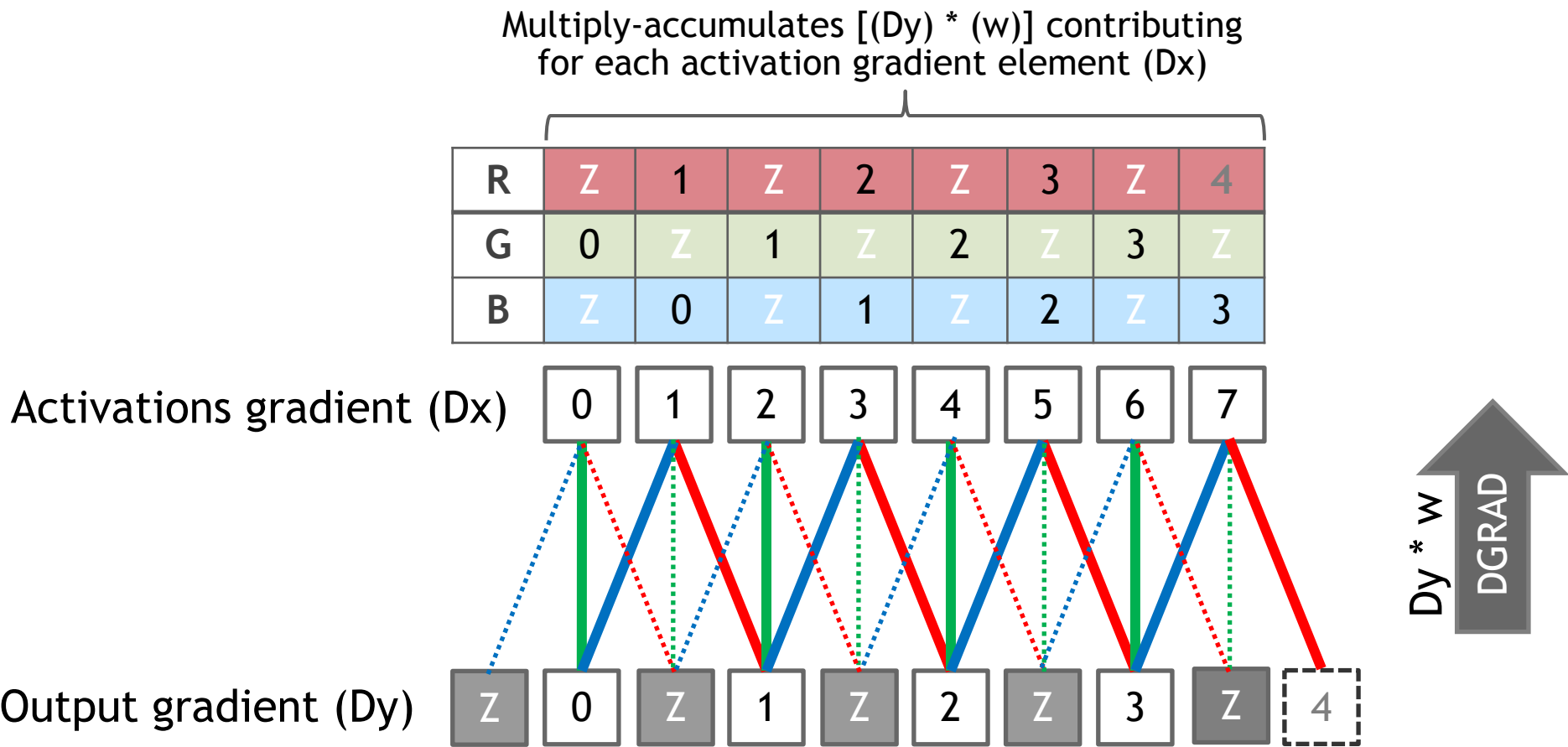
STRIDED BACKWARD DATA GRADIENT | $DX = DGRAD(DY, W)$

1D Convolution (Stride = 2, Filter = 3)



BACKWARD DATA GRADIENT (DGRAD) [STRIDED] NAÏVE IMPLEMENTATION

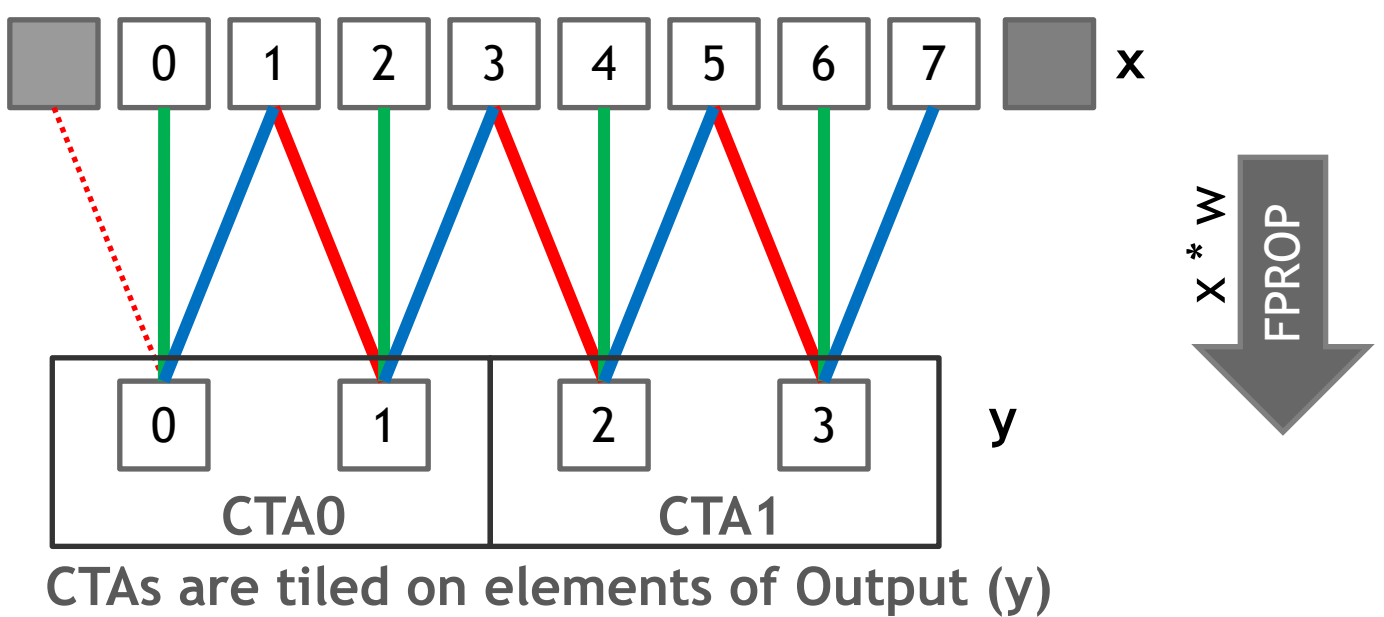
1D Convolution (Stride = 2, Filter = 3)



STRIDED FPROP VS. STRIDED DGRAD

1D Convolution (Stride = 2, Filter = 3)

STRIDED FPROP

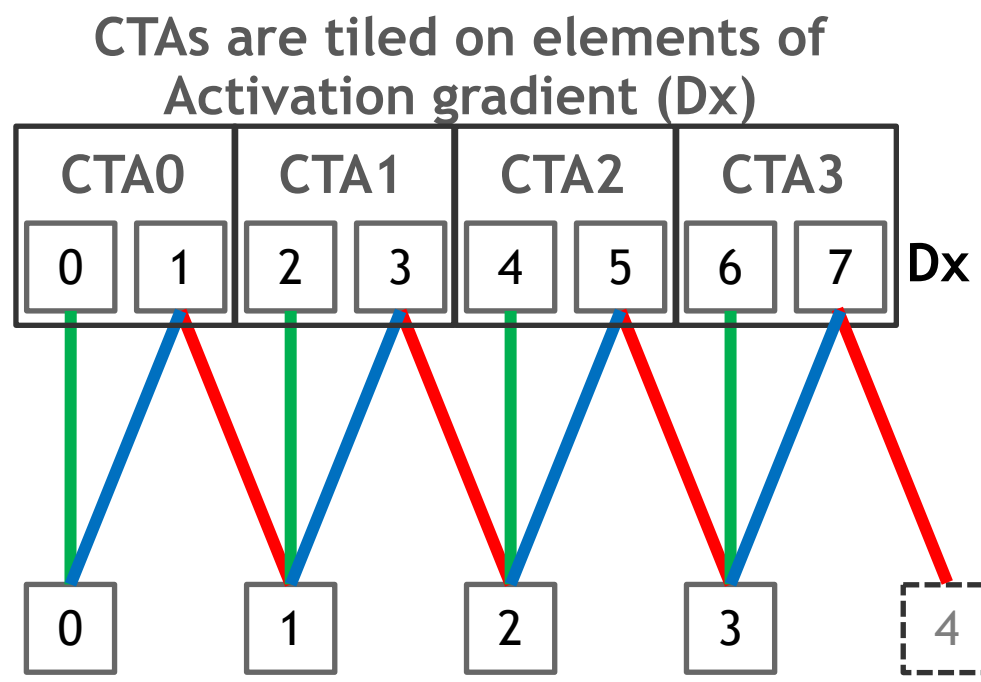


R			1		3		5	
G	0		2		4		6	
B	1		3		5		7	

iterations

R		1		2		3		
G	0		1		2		3	
B		0		1		2		3

iterations



STRIDED DGRAD

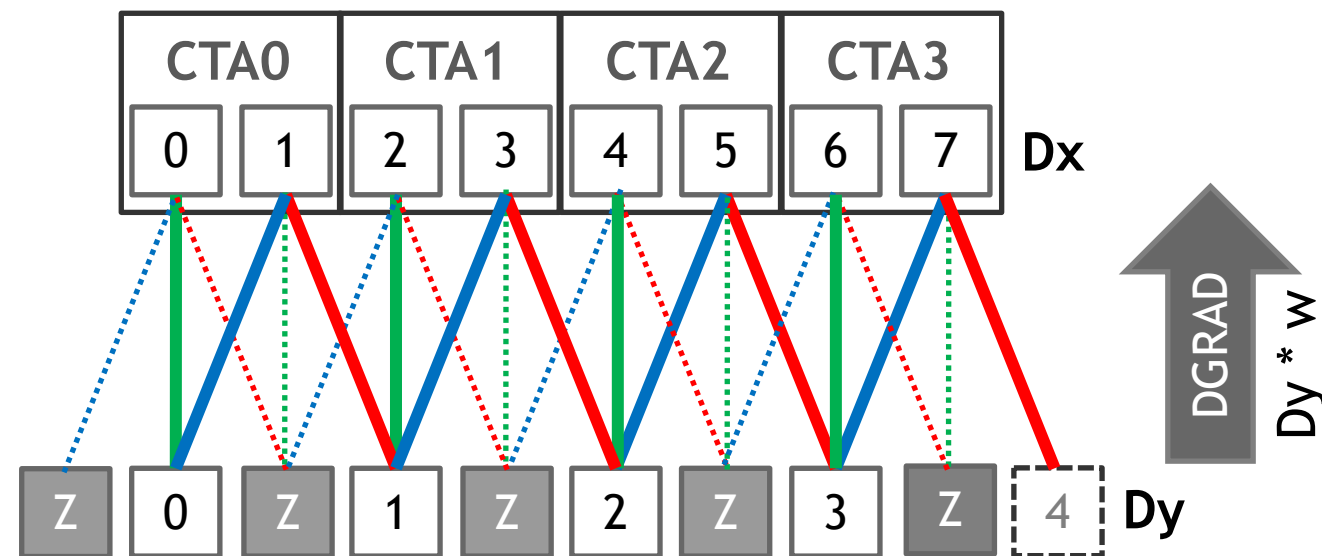


NAÏVE STRIDED DGRAD VS CUTLASS STRIDED DGRAD

1D Convolution (Stride = 2, Filter = 3)

R	Z	1	Z	2	Z	3	Z	4
G	0	Z	1	Z	2	Z	3	Z
B	Z	0	Z	1	Z	2	Z	3

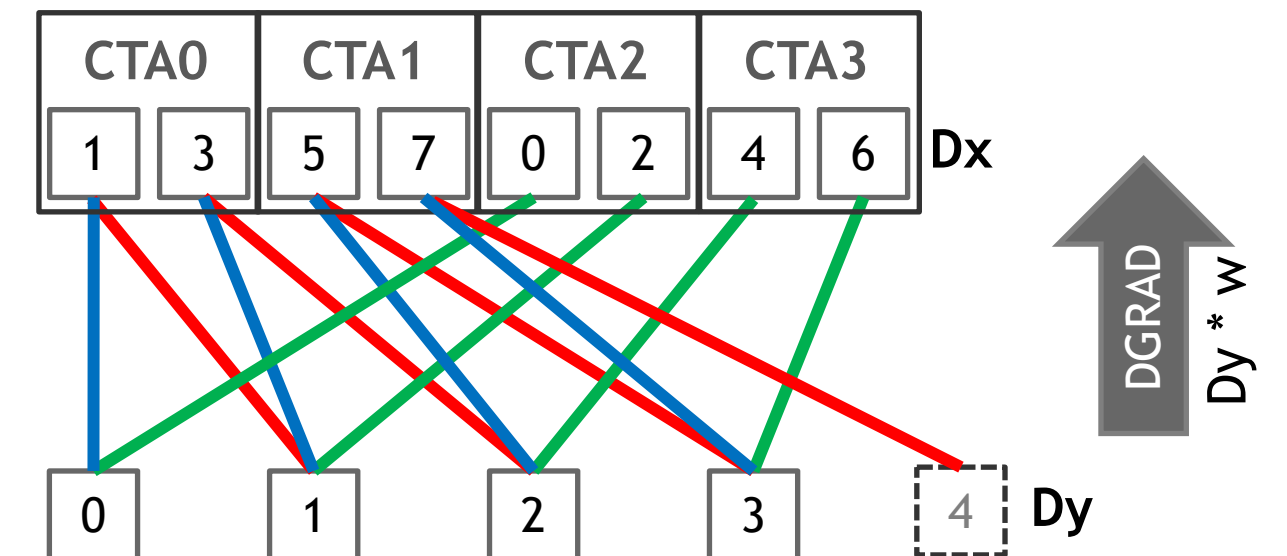
↓ iterations
3 GEMM-K



NAÏVE STRIDED DGRAD
(CUTLASS 2.5)

R	1	2	3	4				
G					0	1	2	3
B	0	1	2	3				

↓ iterations
1-2 GEMM-K

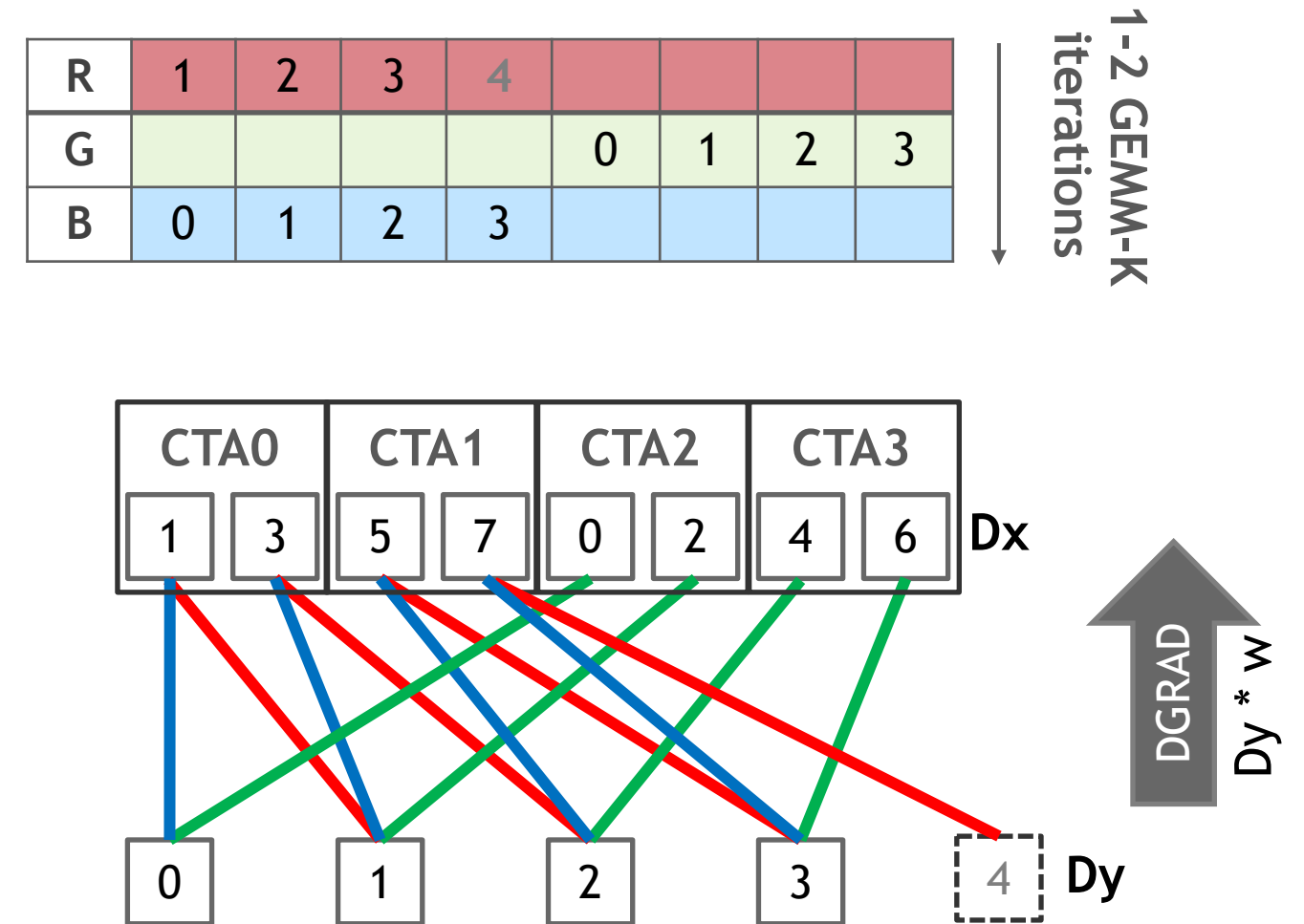


CUTLASS STRIDED DGRAD
(CUTLASS 2.6)

CUTLASS STRIDED DGRAD

1D Convolution (Stride = 2, Filter = 3)

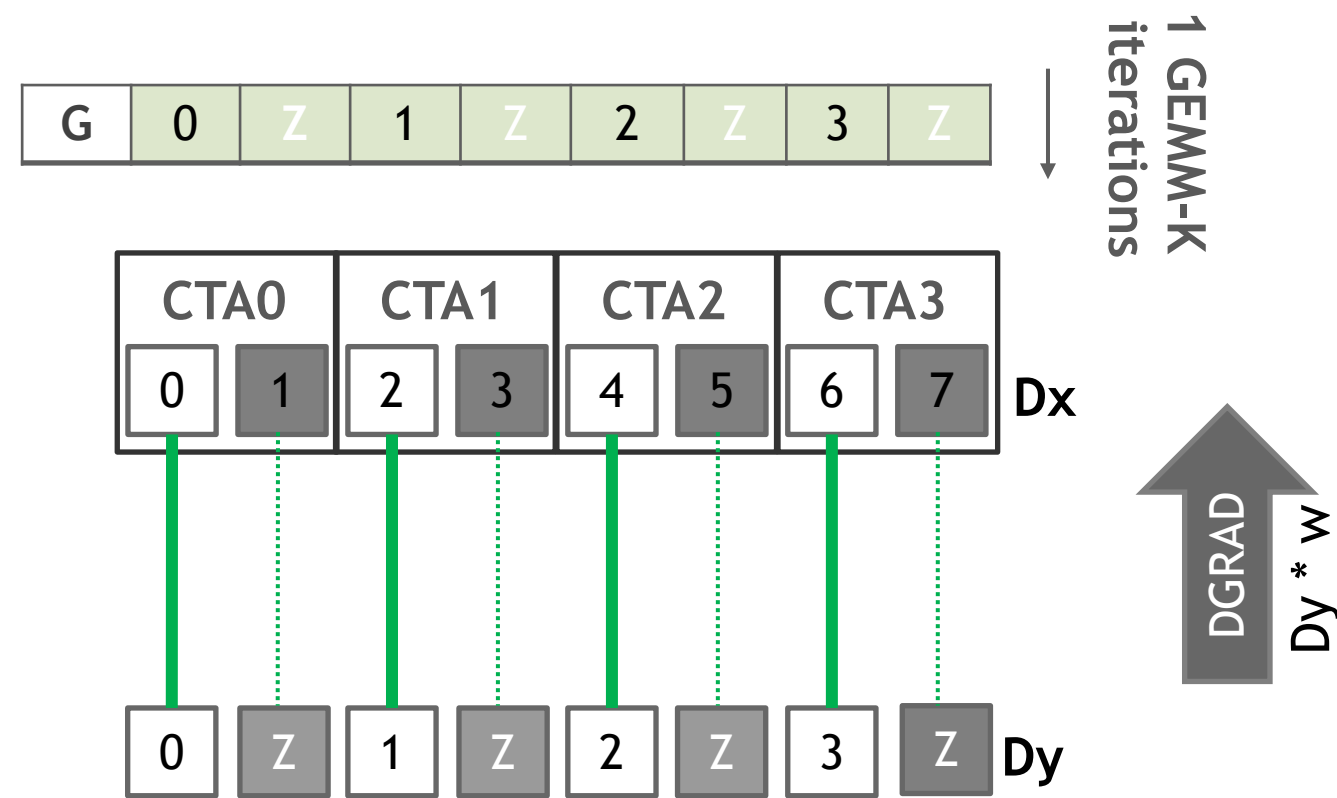
- **Prologue** maps Dx elements to skip redundant multiply-accumulate
- **Mainloop** runs only the useful GEMM-K iterations
 - CTA 0-1 computes Dx[i]; for $i=\{1, 3, 5, 7\}$ (2 multiply-accumulate)
 - CTA 2-3 computes Dx[i]; for $i=\{0, 2, 4, 6\}$ (1 multiply-accumulate)
- **Epilogue** maps Dx elements to store at correct destination locations



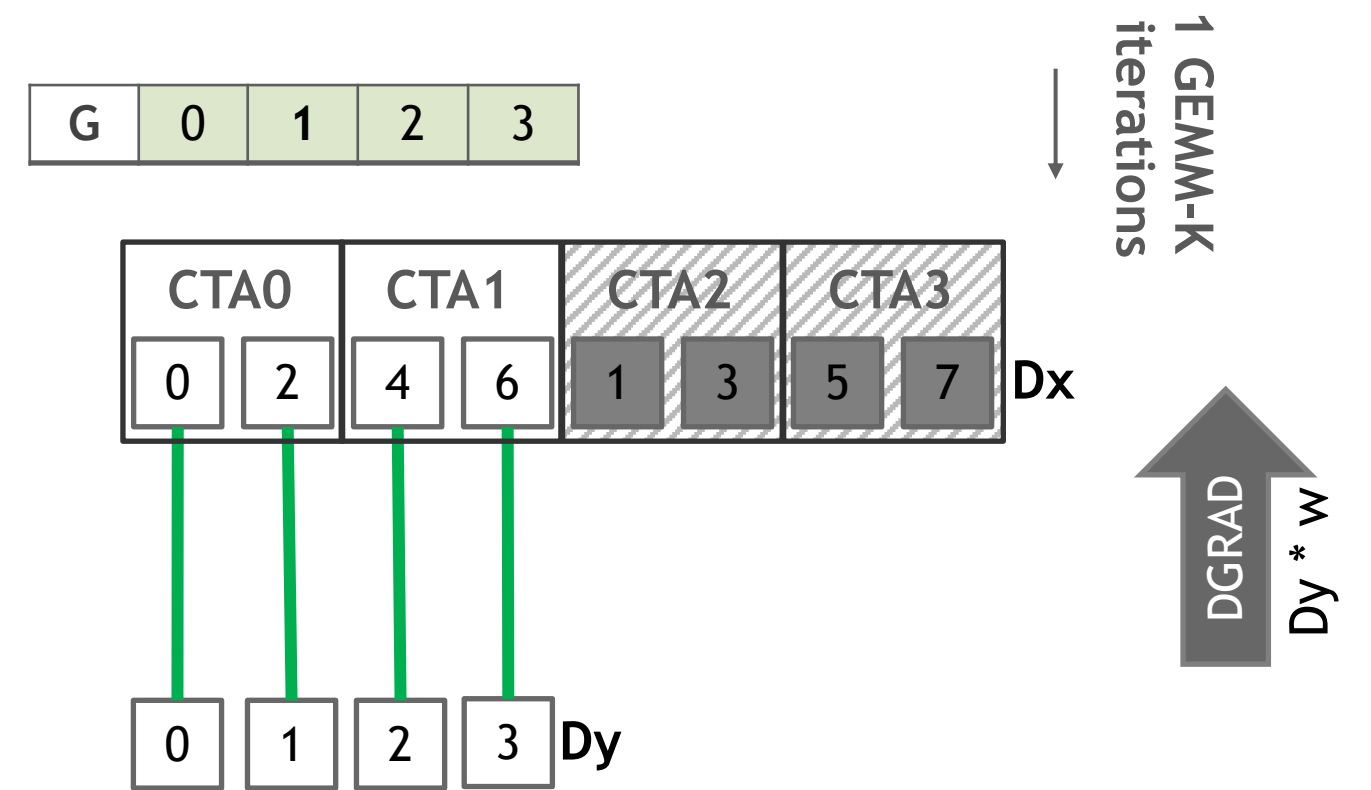
CUTLASS STRIDED DGRAD

CUTLASS STRIDED DGRAD [SMALL FILTER]

1D Convolution (Stride = 2, Filter = 1) | Generalization (Stride > Filter)



NAÏVE STRIDED DGRAD
(CUTLASS 2.5)



CUTLASS STRIDED DGRAD
(CUTLASS 2.6)

CUTLASS STRIDED DGRAD

(Stride > 1)

```
//
// Prologue
//
id -> mapped_id // Map CTA to compute a valid Dx element

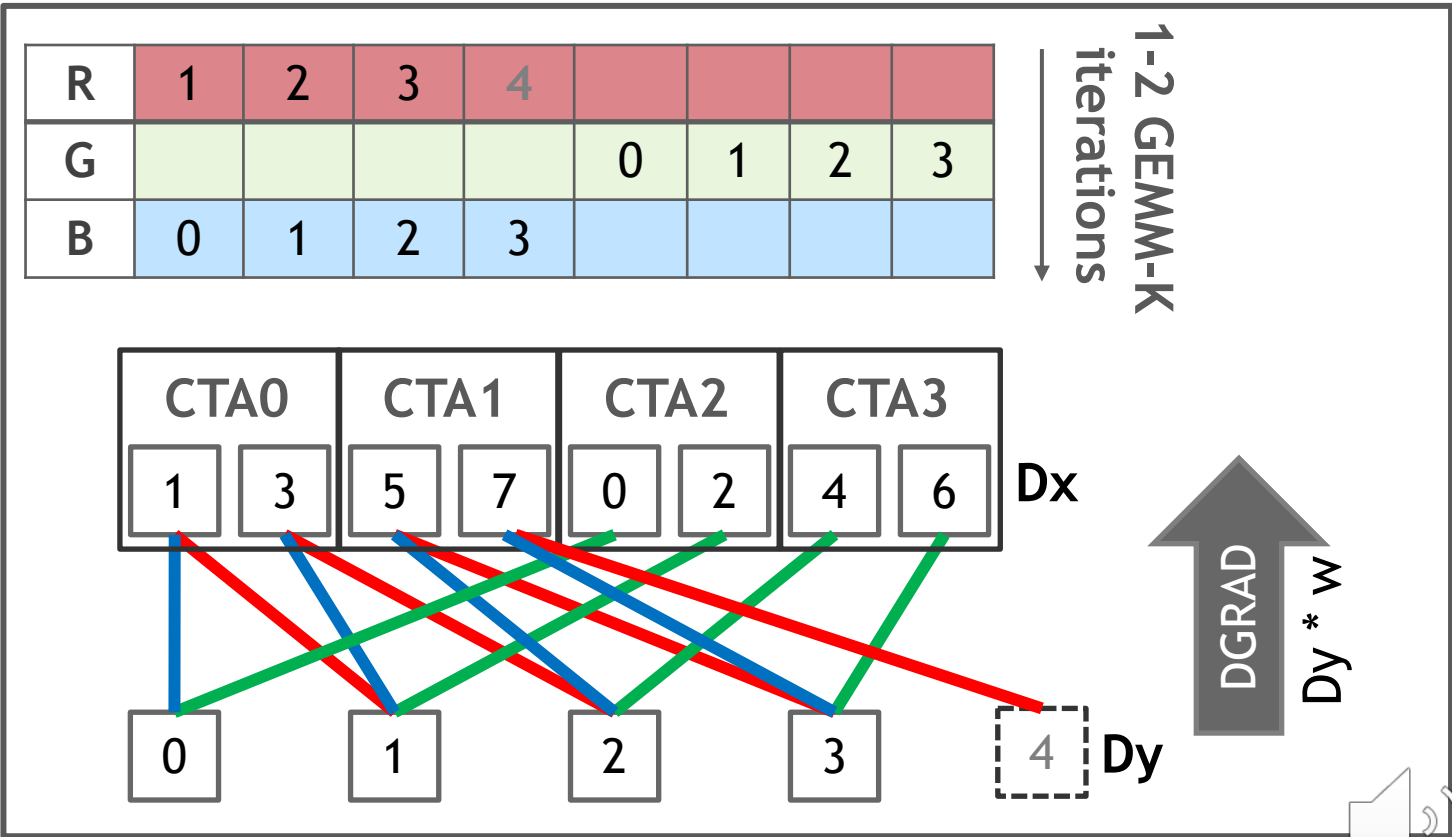
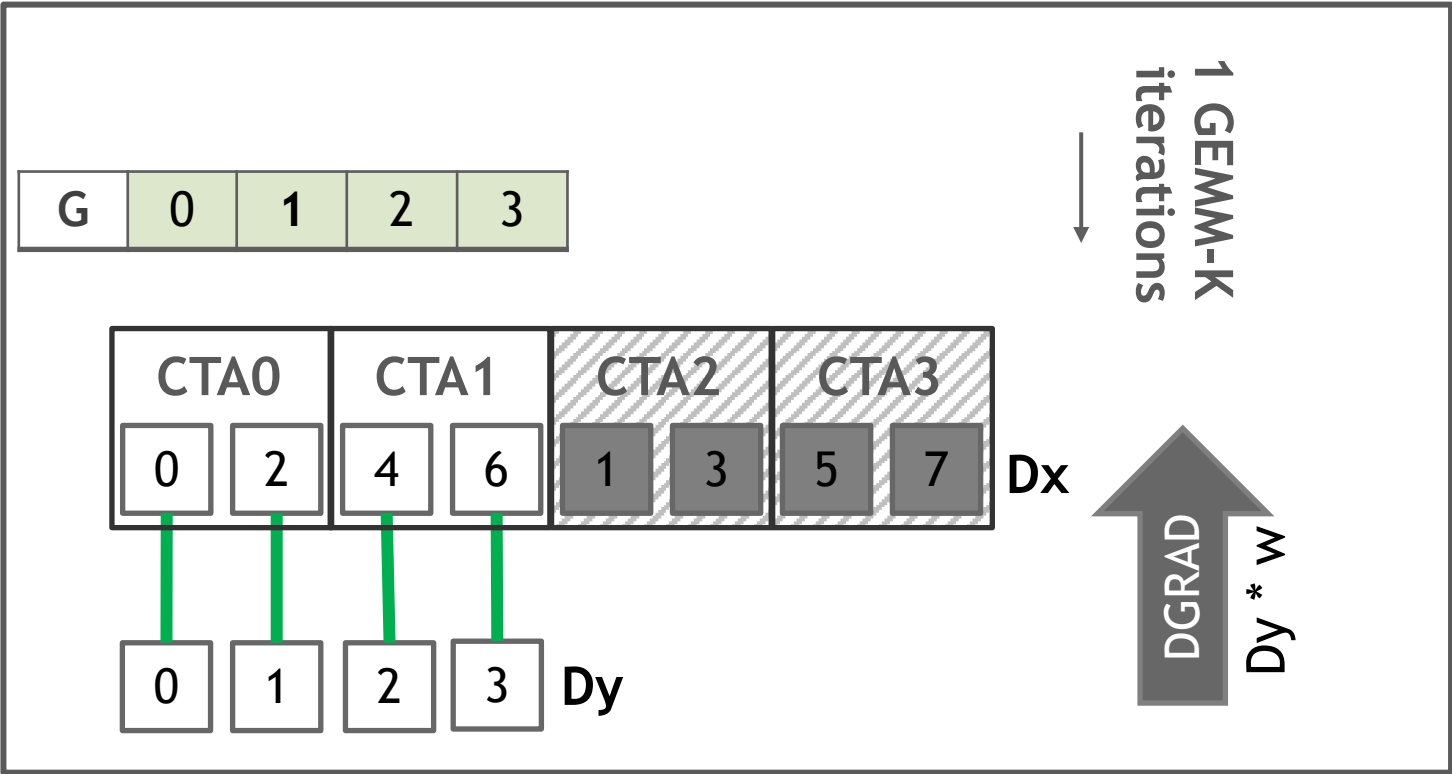
//
// Mainloop
//
accumulators = 0 // Clear accumulators

if (isMainloopRequired(blockIdx.x)) {
    accumulators = Dy * w
}

//
// Epilogue
//
id -> mapped_id // Map back to destination Dx element location

Dx_source = ((beta==0) ? 0 : Dx[mapped_id])

Dx[mapped_id]= alpha * (accumulators) + beta * Dx_source // Store Dx
```





STRIDED DGRAD IN 2D (IMPLICIT GEMM CONVOLUTIONS)



2D DGRAD ON 4D TENSORS - DEFINITION

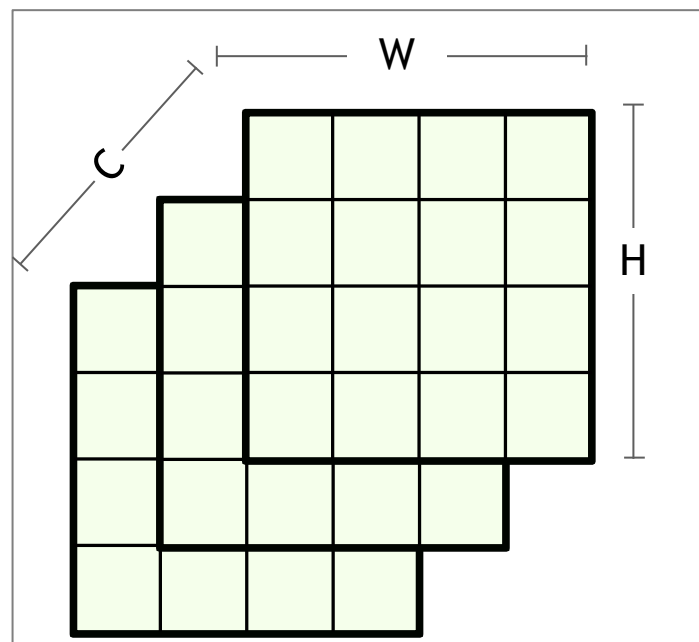
Backward Data Propagation | $\mathbf{Dx} = \text{CONV}(\mathbf{Dy}, \mathbf{w})$

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

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

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

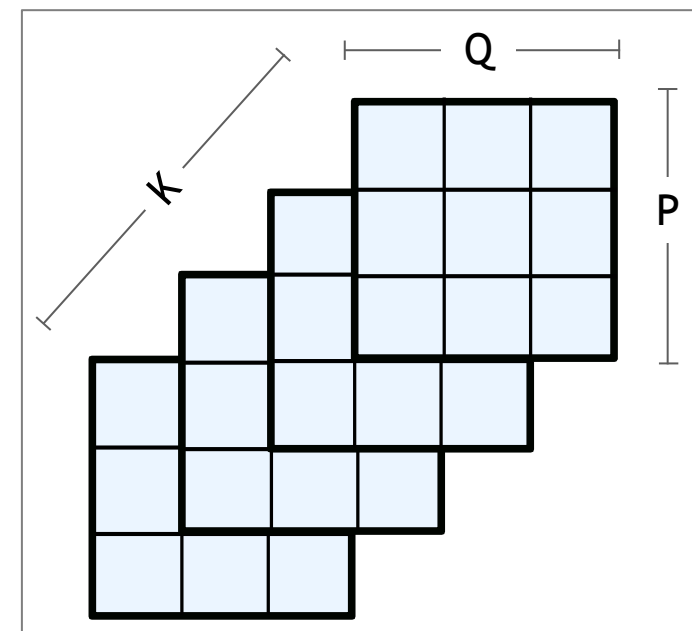
N (batch = 1)



Activation Gradient
Tensor ($\mathbf{Dx}$)

NHWC = {1, 4, 4, 3}

N (batch = 1)

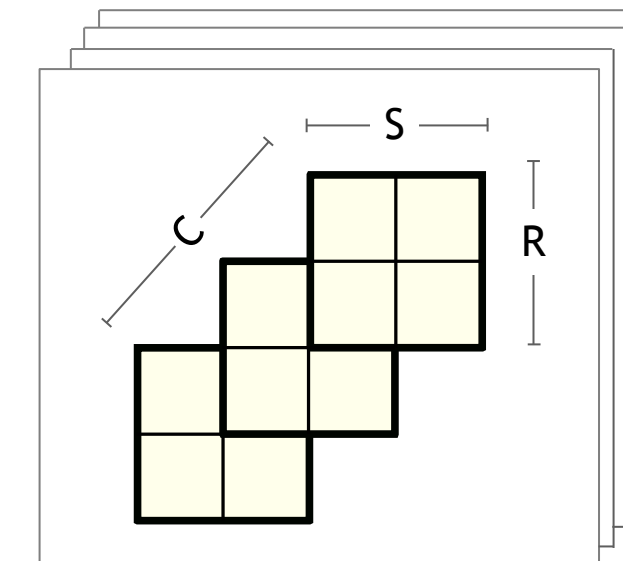


Output Gradient
Tensor ($\mathbf{Dy}$)

NPQK = {1, 3, 3, 4}

*

K (filters = 4)



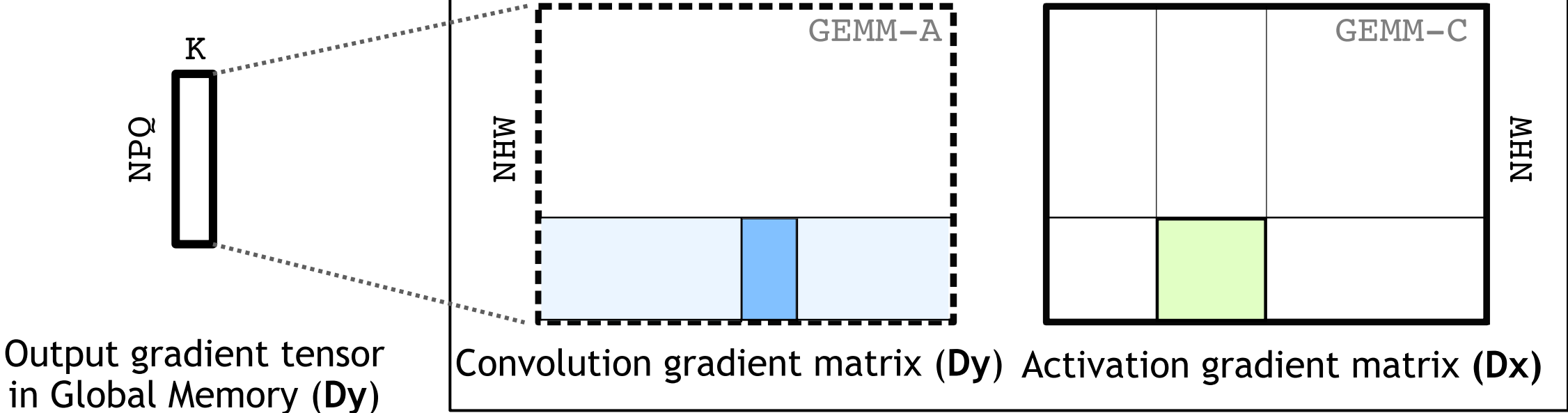
Filter
Tensor ($\mathbf{w}$)

KRSC = {4, 2, 2, 3}

IMPLICIT GEMM CONVOLUTION

Backward Data Gradient (Dgrad)

4D Tensors
$\mathbf{Dx} = CONV(\mathbf{Dy}, \mathbf{w})$
$\mathbf{Dy}[N, P, Q, K]$: 4D output gradient tensor
$\mathbf{w}[K, R, S, C]$: 4D filter tensor
$\mathbf{Dx}[N, H, W, C]$: 4D activation gradient tensor



2D Matrices
$\mathbf{Dx} = CONV(\mathbf{Dy}, \mathbf{w})$
$\mathbf{GEMM-A}[NHW, KRS]$: 2D matrix
$\mathbf{GEMM-B}[KRS, C]$: 2D $\mathbf{w}$ matrix
$\mathbf{GEMM-C}[NHW, C]$: 2D $\mathbf{Dx}$ matrix

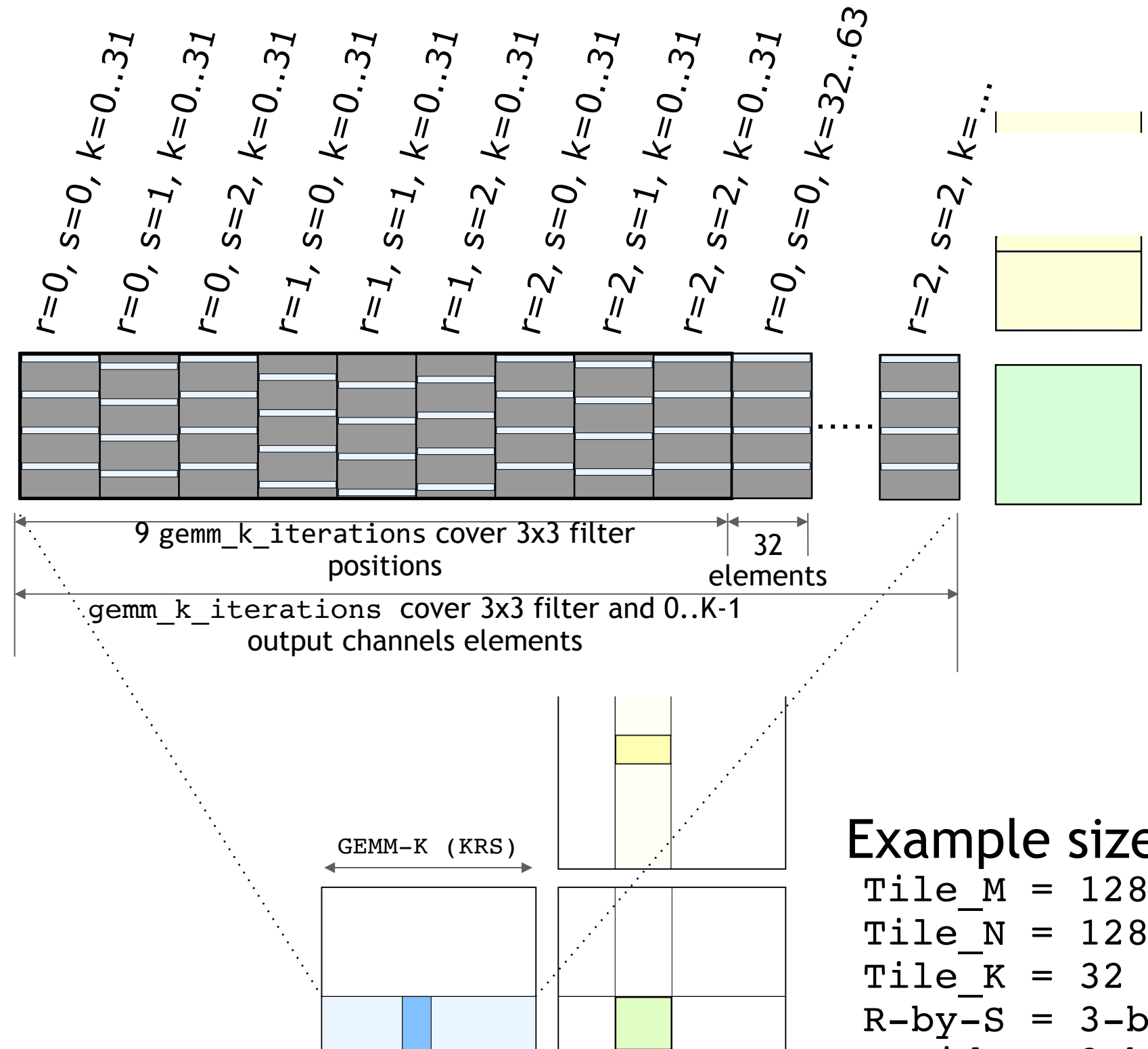


NAÏVE STRIDED DGRAD (IMPLICIT GEMM) - CUTLASS 2.5

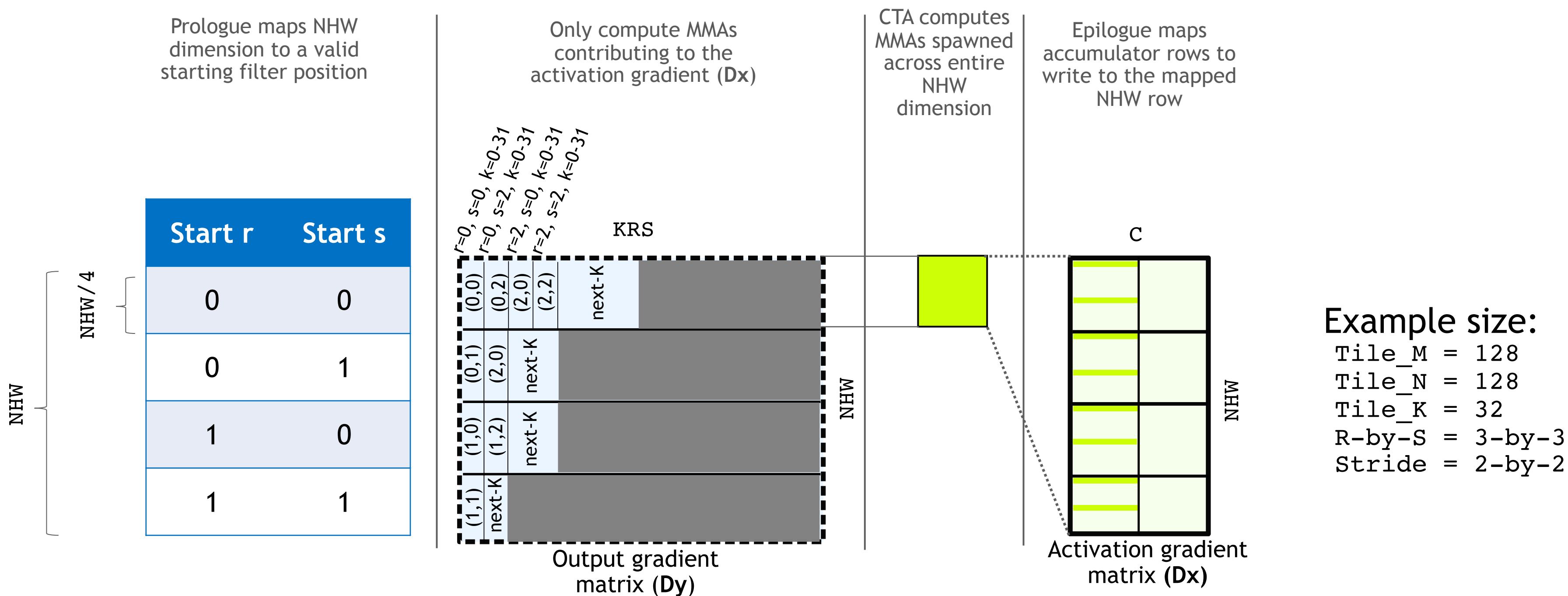
To cover the entire GEMM-K (KRS) dimension

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

$$\text{gemm_k_iterations} = R * S * ((K + \text{Tile_K} - 1) / \text{Tile_K})$$



CUTLASS STRIDED DGRAD (IMPLICIT GEMM) - CUTLASS 2.6+





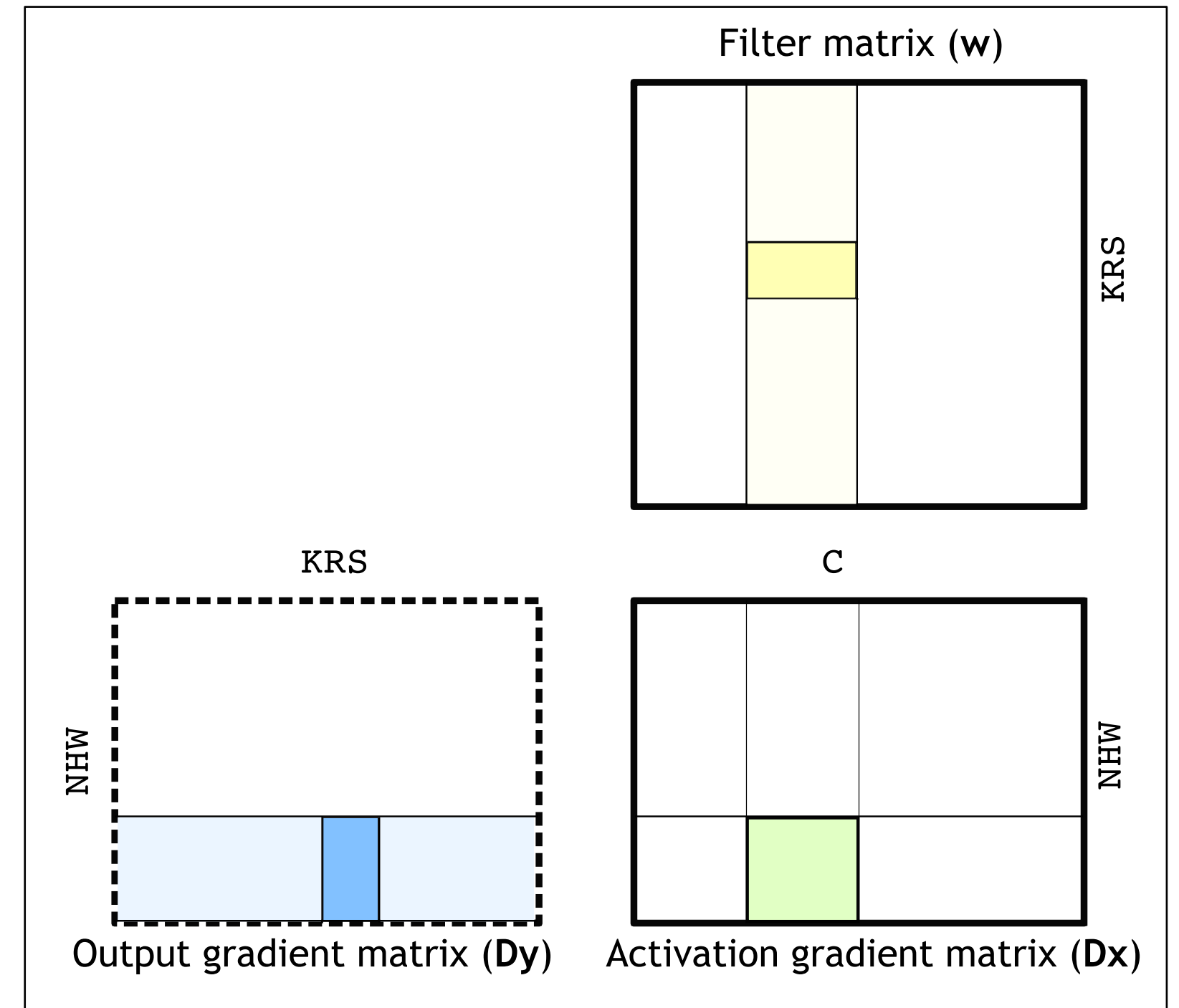
IMPLICIT GEMM CONVOLUTIONS



IMPLICIT GEMM CONVOLUTION

Implicit GEMM convolution algorithm:

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

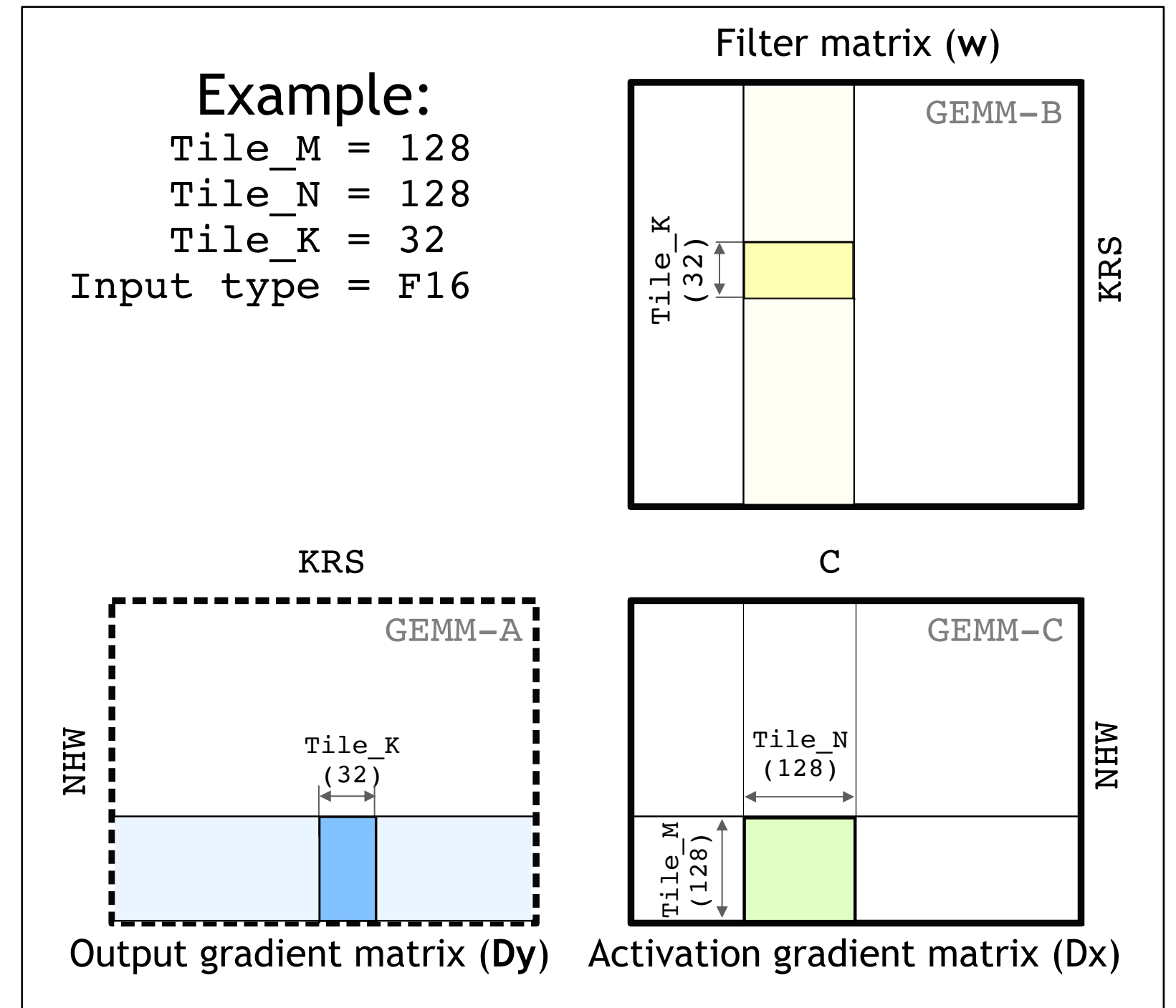


IMPLICIT GEMM CONVOLUTION

Load a tile of Output gradient matrix (**Dy**) and filter matrix (**w**)

Implicit GEMM convolution algorithm:

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



IMPLICIT GEMM CONVOLUTION

Load a tile of Output gradient matrix (**Dy**) and filter matrix (**w**)

GEMM-A tile of output gradient matrix (Dy)

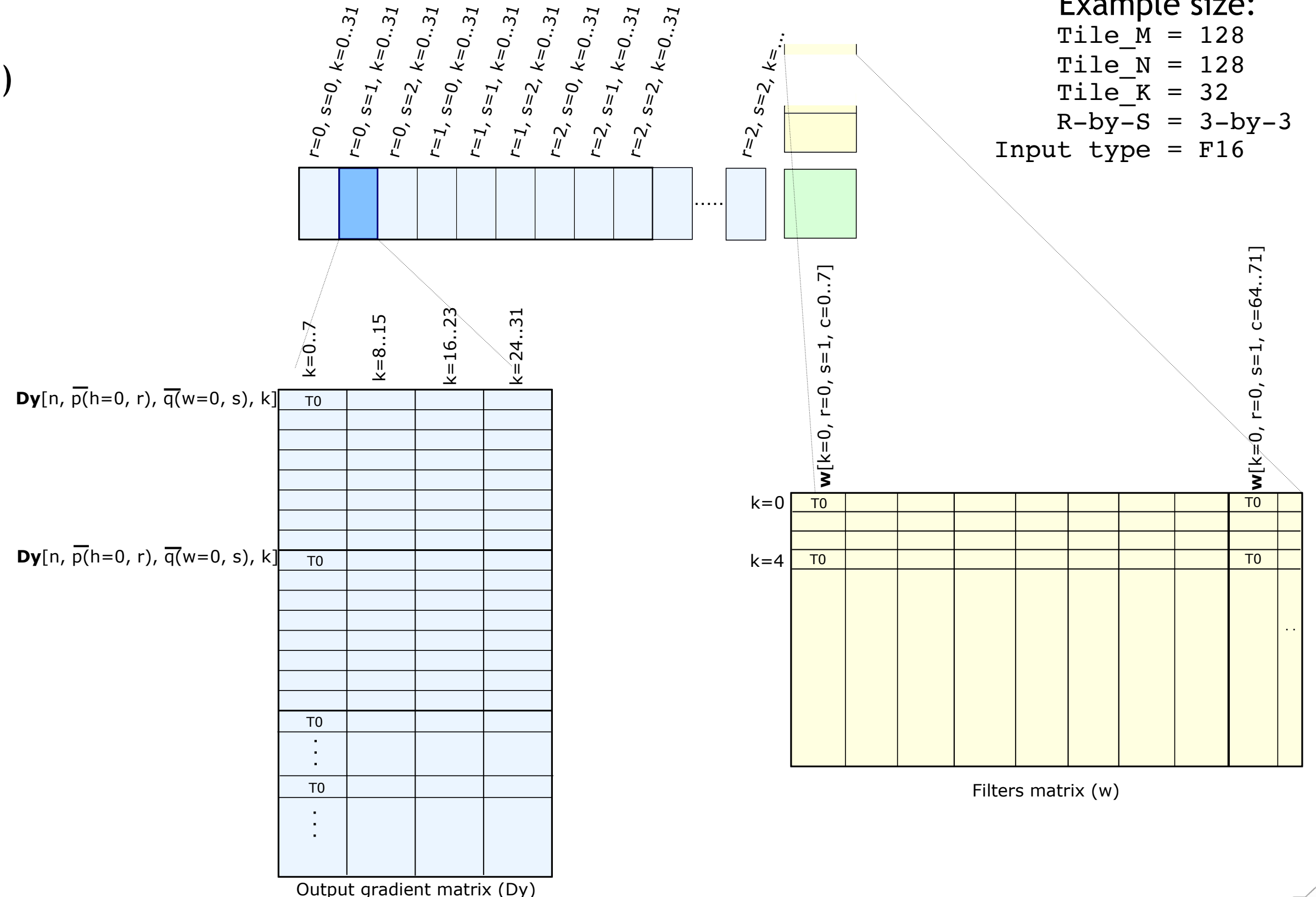
- Loads 128-by-32 elements
- Each row mapped to unique (n, h, w) coordinate
- Each column mapped to unique filter k coordinate

GEMM-B tile of filter matrix (w)

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

Each thread issues multiple vector loads from each tile

- For example, T0's first load brings k=0..7 to Shared Memory for F16 operands

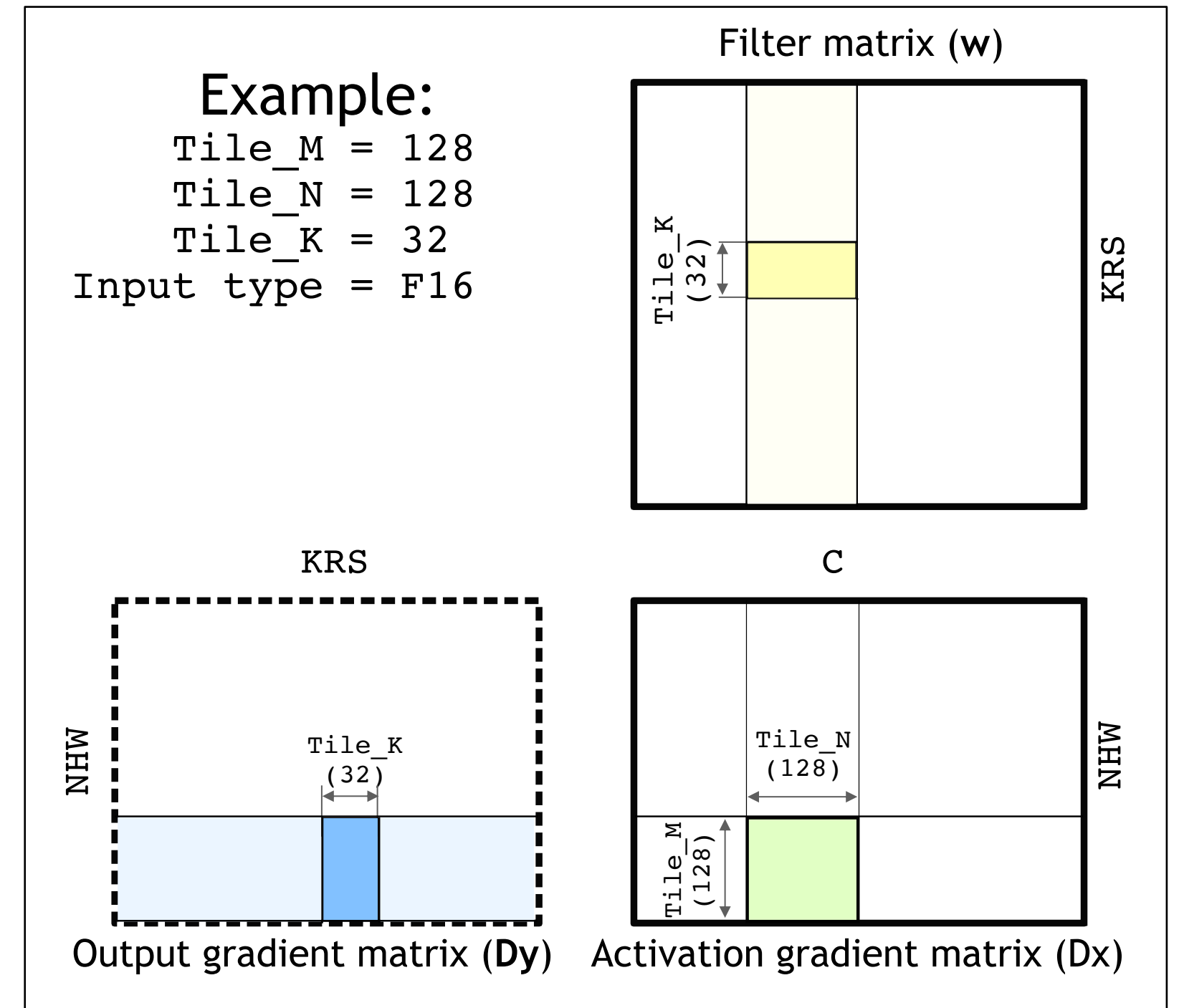


IMPLICIT GEMM CONVOLUTION

Compute **matrix-multiply accumulate**

Implicit GEMM convolution algorithm:

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

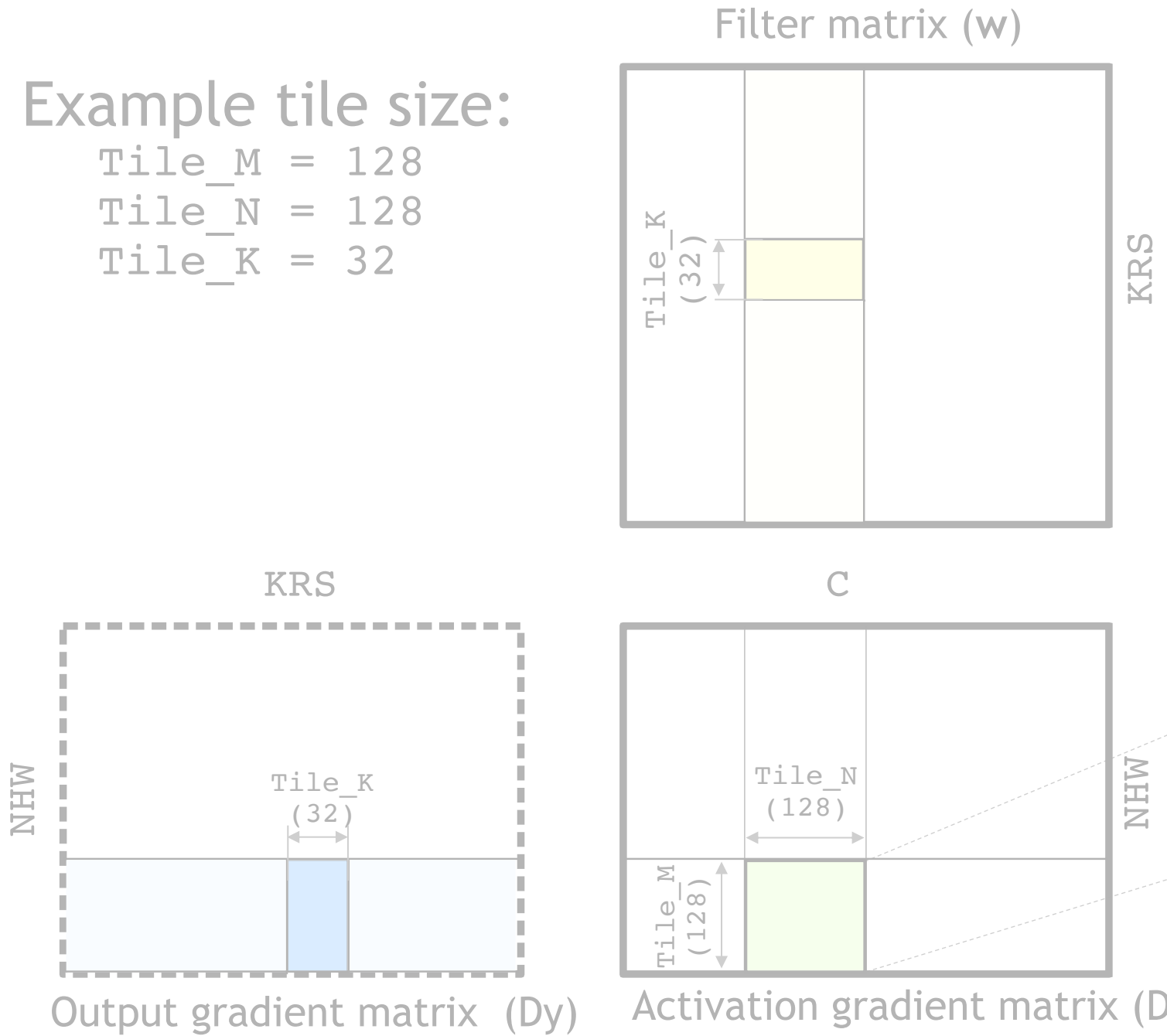


IMPLICIT GEMM CONVOLUTION

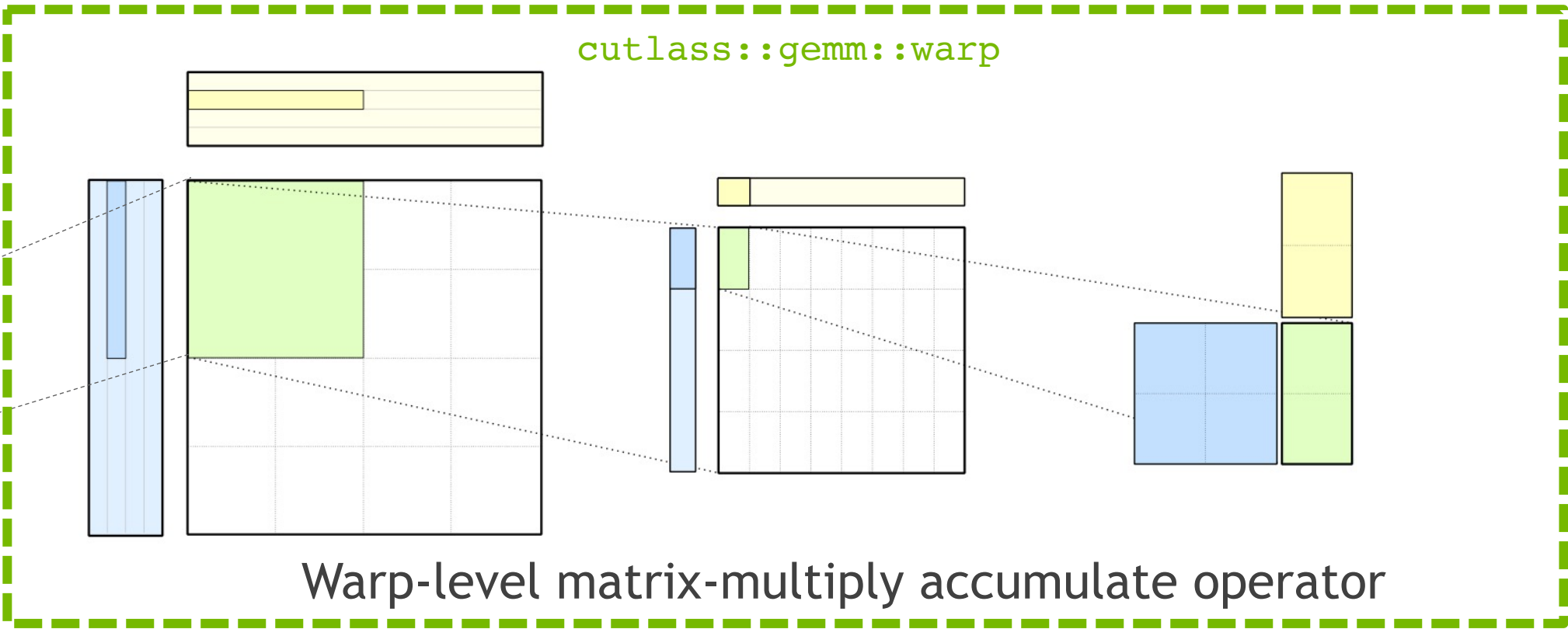
Compute matrix-multiply accumulate

Example tile size:

Tile_M = 128
Tile_N = 128
Tile_K = 32



The warp-level mma operators are used directly from `cutlass::gemm::warp` to use NVIDIA Tensor Cores

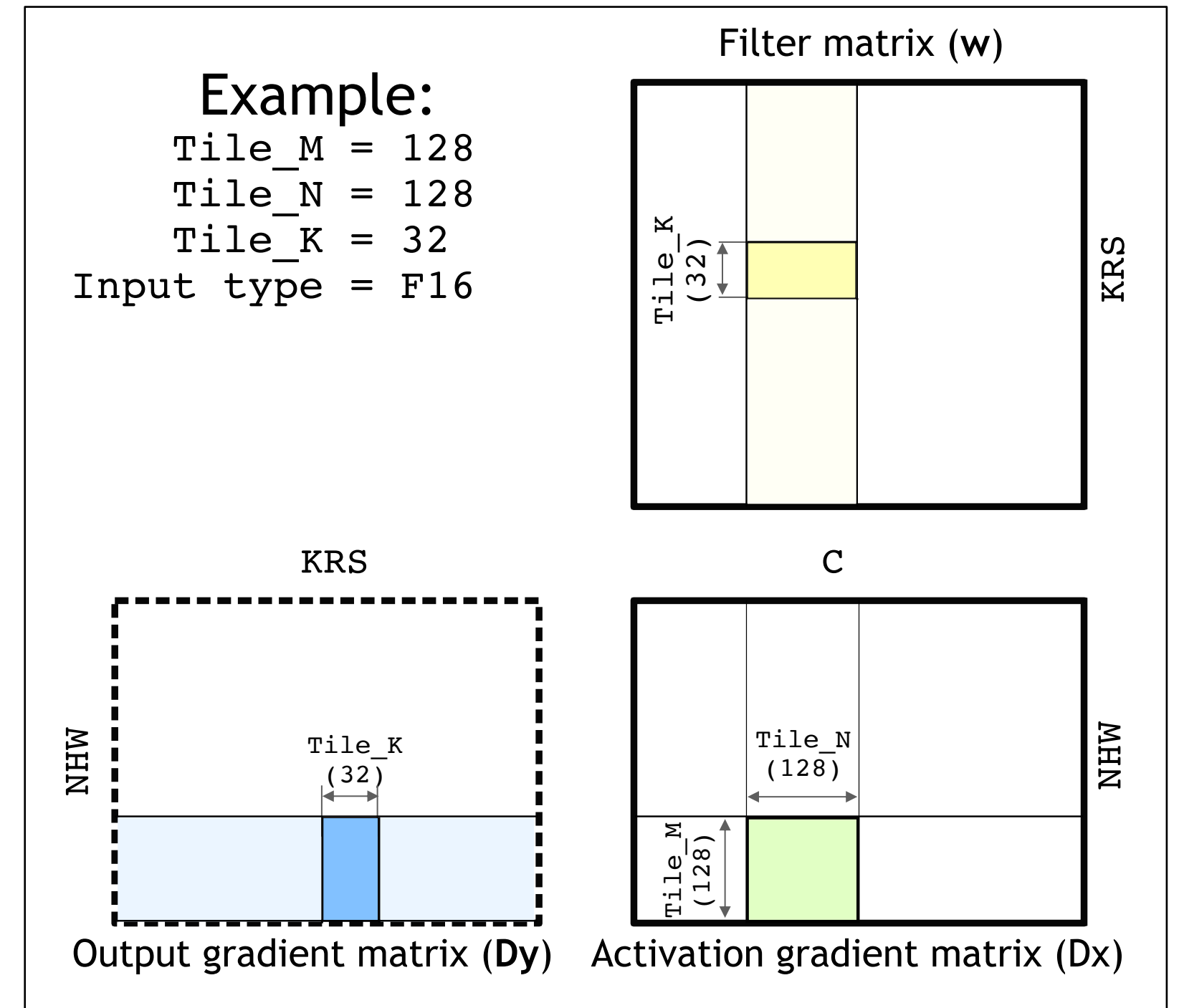


IMPLICIT GEMM CONVOLUTION

Iterate over KRS dimension

Implicit GEMM convolution algorithm:

1. Load a tile of Dy and filter w into Shared Memory
2. Compute matrix-multiply accumulate (mma) on operands in Shared Memory
3. Iterate over KRS dimension
 - a) Advance to load next tiles in Shared Memory
 - b) Ensure accumulation for all filter positions (r, s) and output channels k



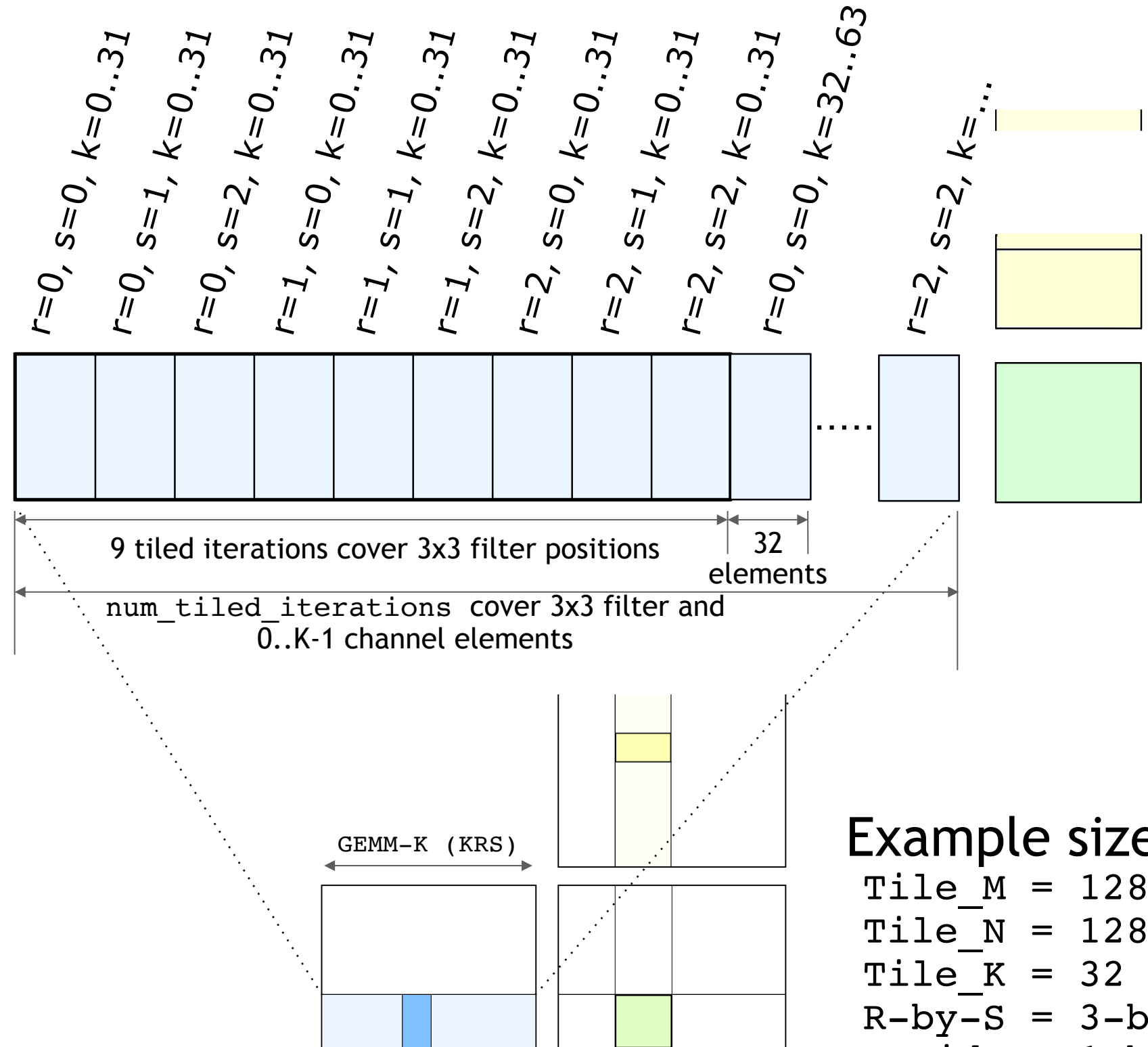
IMPLICIT GEMM CONVOLUTION

Iterate over KRS dimension

To cover the entire GEMM-K (KRS) dimension

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

$$\text{num_tiled_iterations} = R * S * ((K + \text{Tile_K} - 1) / \text{Tile_K})$$



Example size:

Tile_M = 128
 Tile_N = 128
 Tile_K = 32
 R-by-S = 3-by-3
 Stride = 1-by-1

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

operator++()

moves to the next load position for the thread

at()

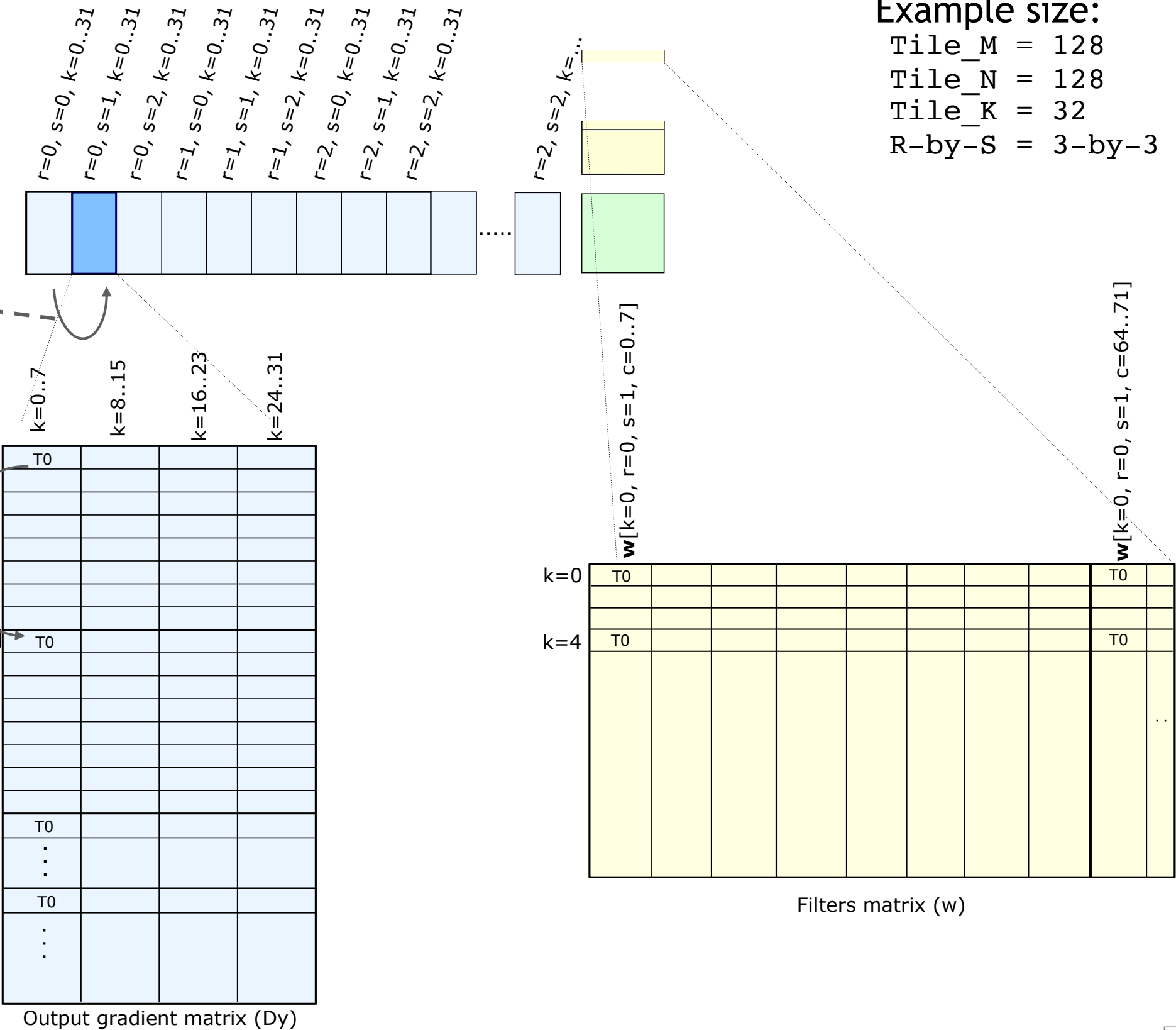
applies $\bar{p}(h,r)$ and $\bar{q}(w,s)$ functions to map hw to pq
and returns coordinates in Dy tensor $\{n,p,q,k\}$

valid()

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

get()

fetches pointer in Global Memory based on tensor
coordinates



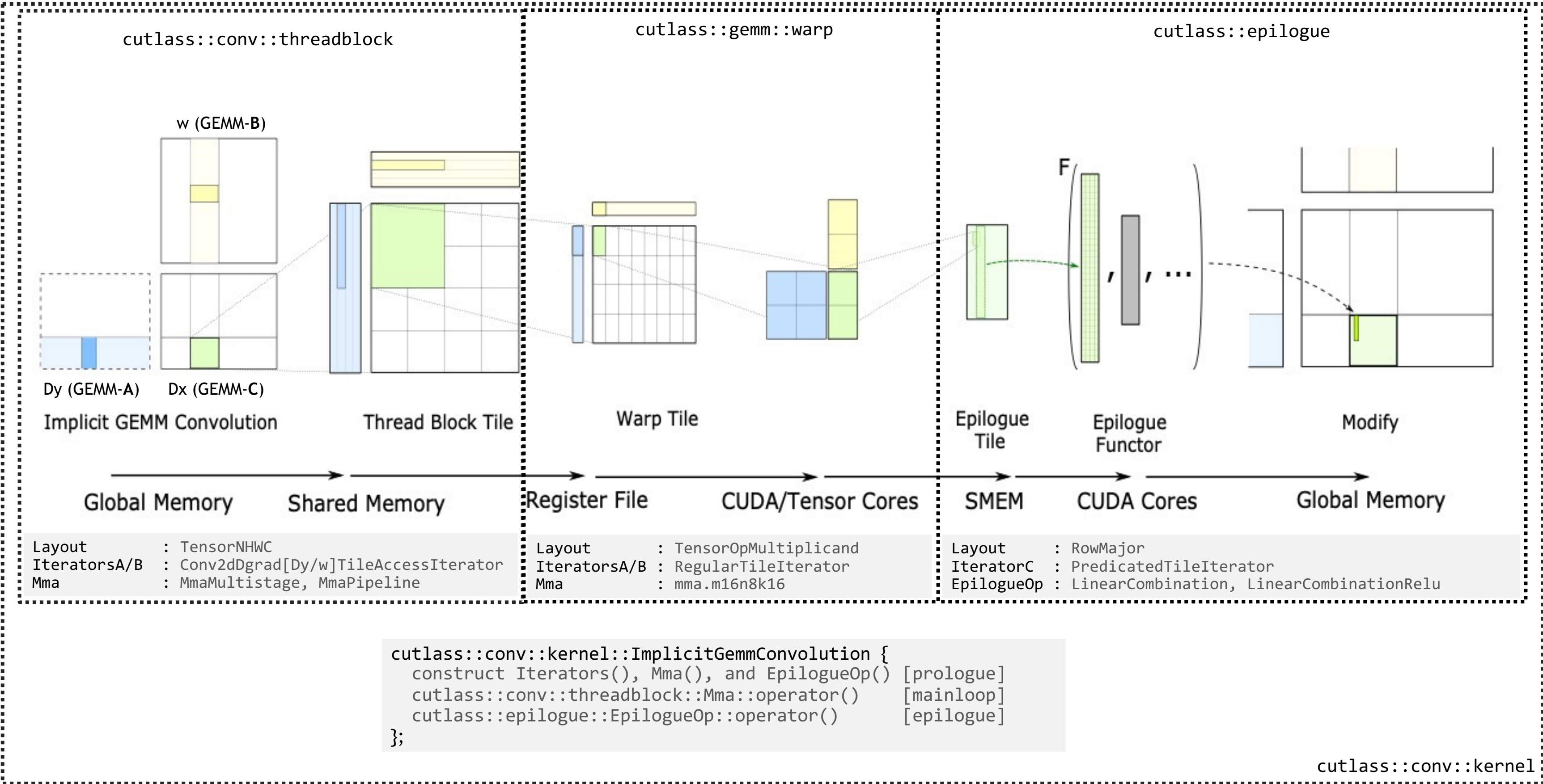


ABSTRACTIONS TO PUT TOGETHER
COMPLEX ALGORITHMS EFFICIENTLY



IMPLICIT GEMM CONVOLUTION

CUTLASS 2.5 components for implicit GEMM convolution

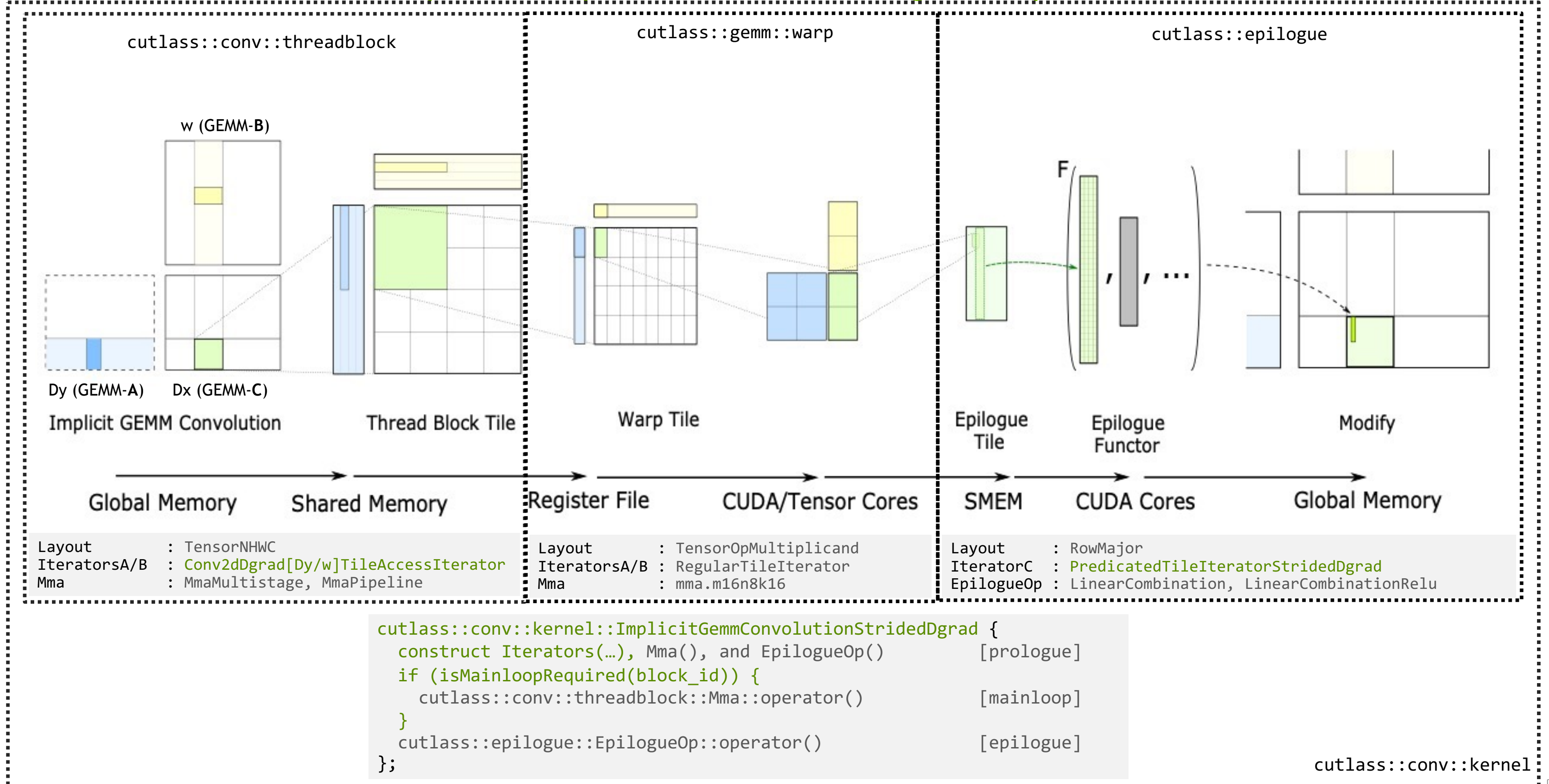


cutlass::conv::kernel



IMPLICIT GEMM CONVOLUTION - STRIDED DGRAD

CUTLASS 2.6+ components updated for strided dgrad implicit GEMM convolution





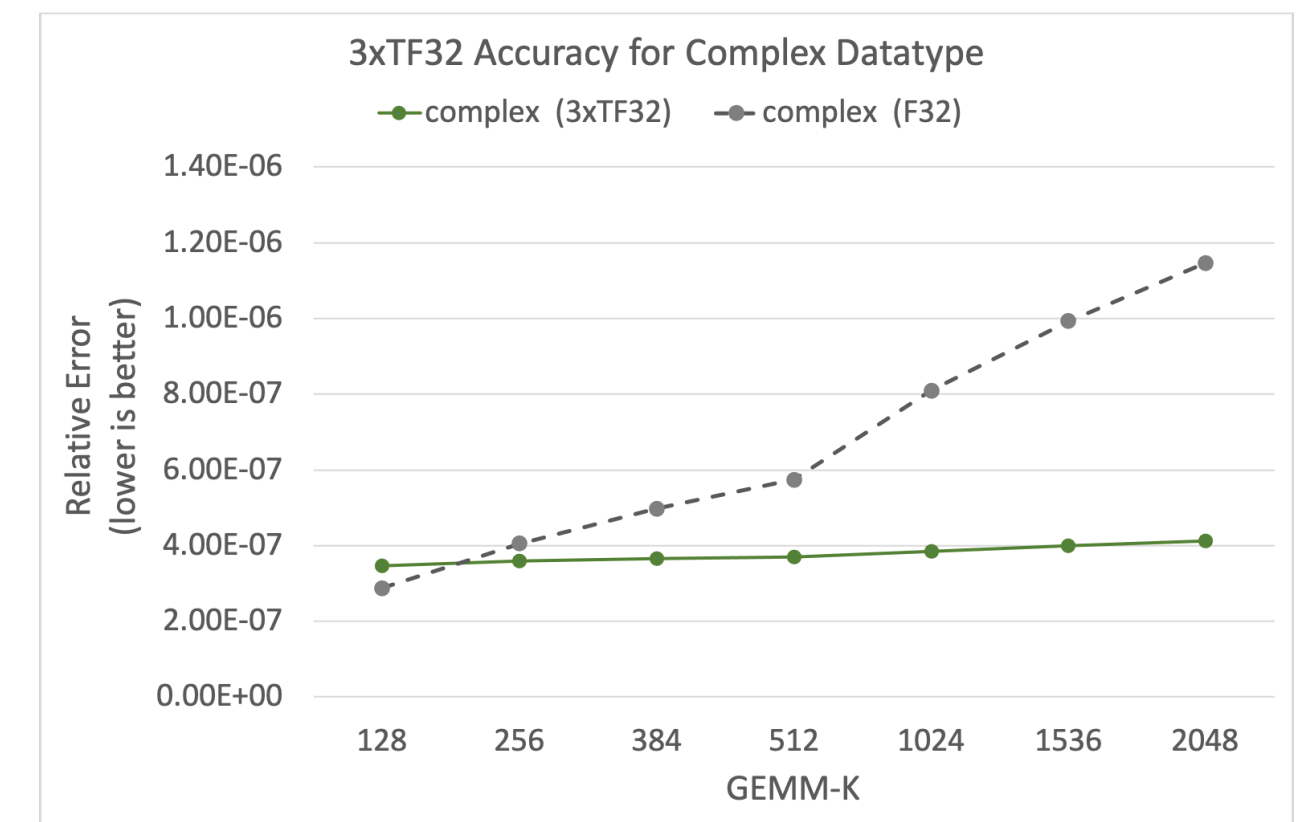
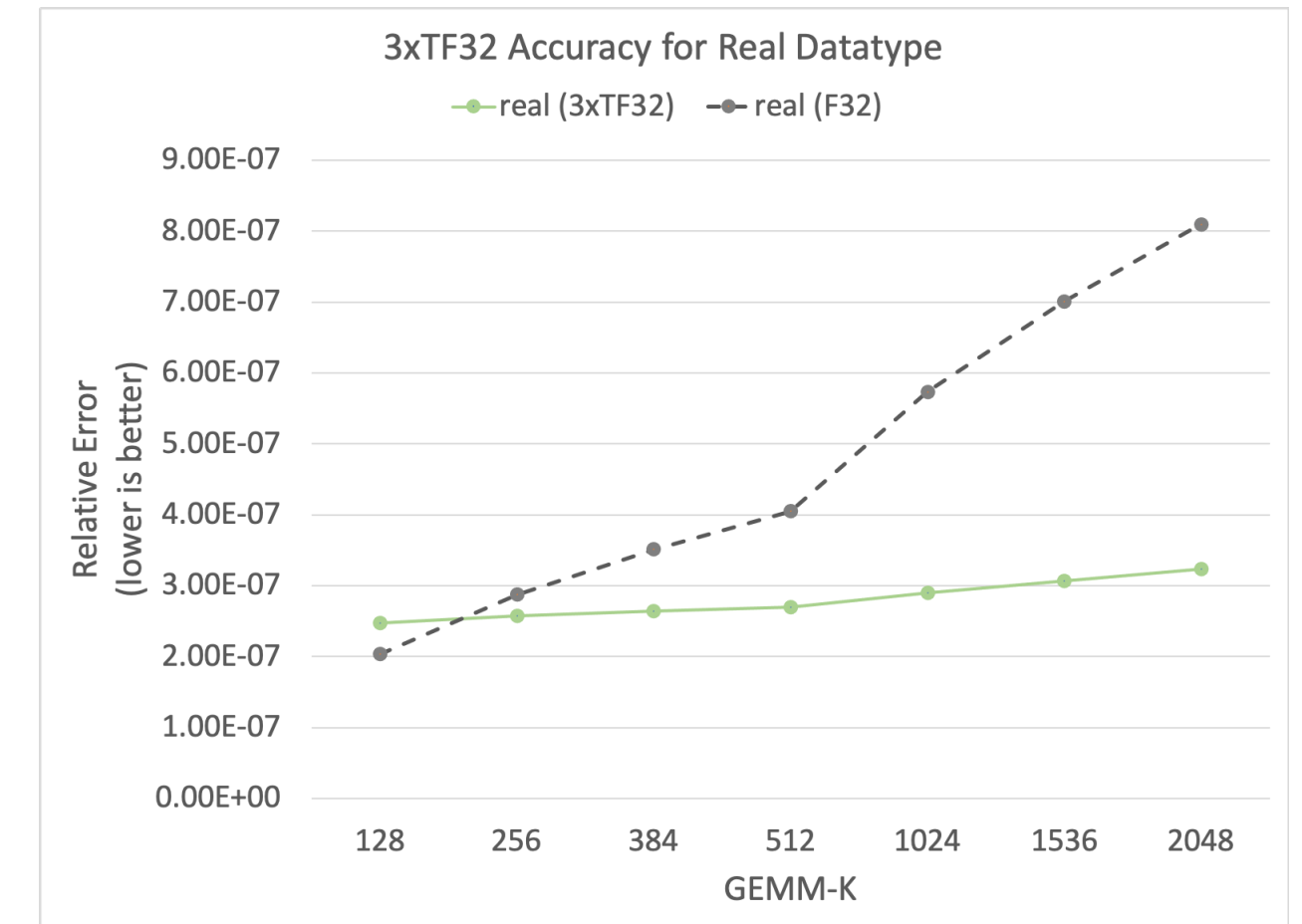
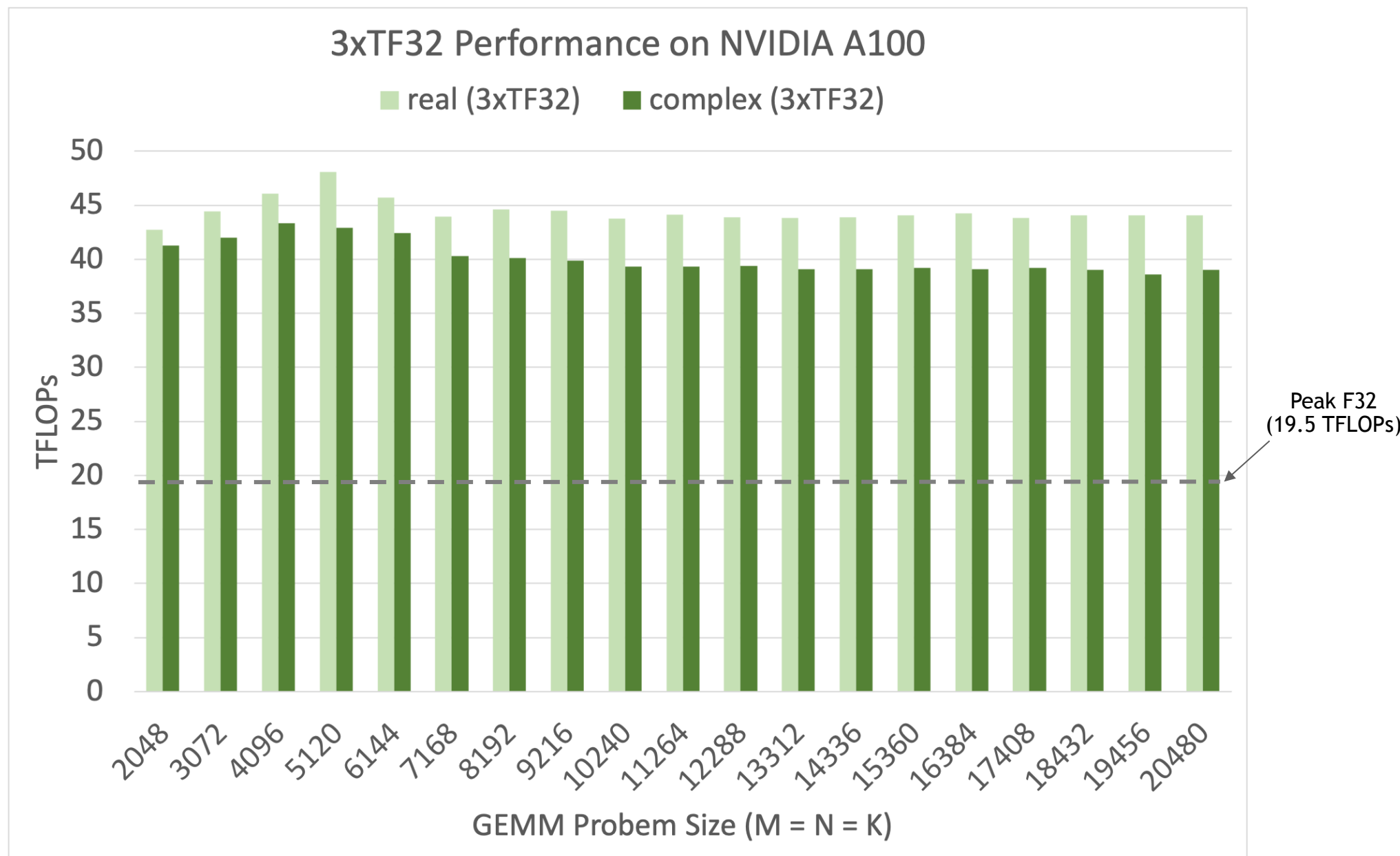
DO MORE WITH CUTLASS



ACCELERATED SINGLE-PRECISION USING TENSOR CORES

3xTF32 accelerates single-precision arithmetic at 48 TFLOPs on A100

- More than 2x performance vs. peak single-precision
- Better accuracy vs. single-precision (not IEEE compliant)
- Supports real and complex data types
- Example implementation for GEMMs and Convolutions



GROUPED GEMM

Batched GEMMs with unique problem sizes

CUTLASS enables an efficient implementation targeting Tensor Cores on NVIDIA Volta, Turing, and Ampere

- A “persistent kernel” launches enough threadblocks to perfectly fill the GPU
- An outer loop and “scheduler” maps the threadblock to a tile of the output problem

Performance exceeds batched GEMM and does not require pre-processing

- Mixture of Experts natural language model **1.75x** faster end to end
- **1.39x** geomean speedup over random sizes from 32 ... 4096

```
// Outer 'persistent' loop to iterate over tiles
while (problem_visitor.next_tile()) {

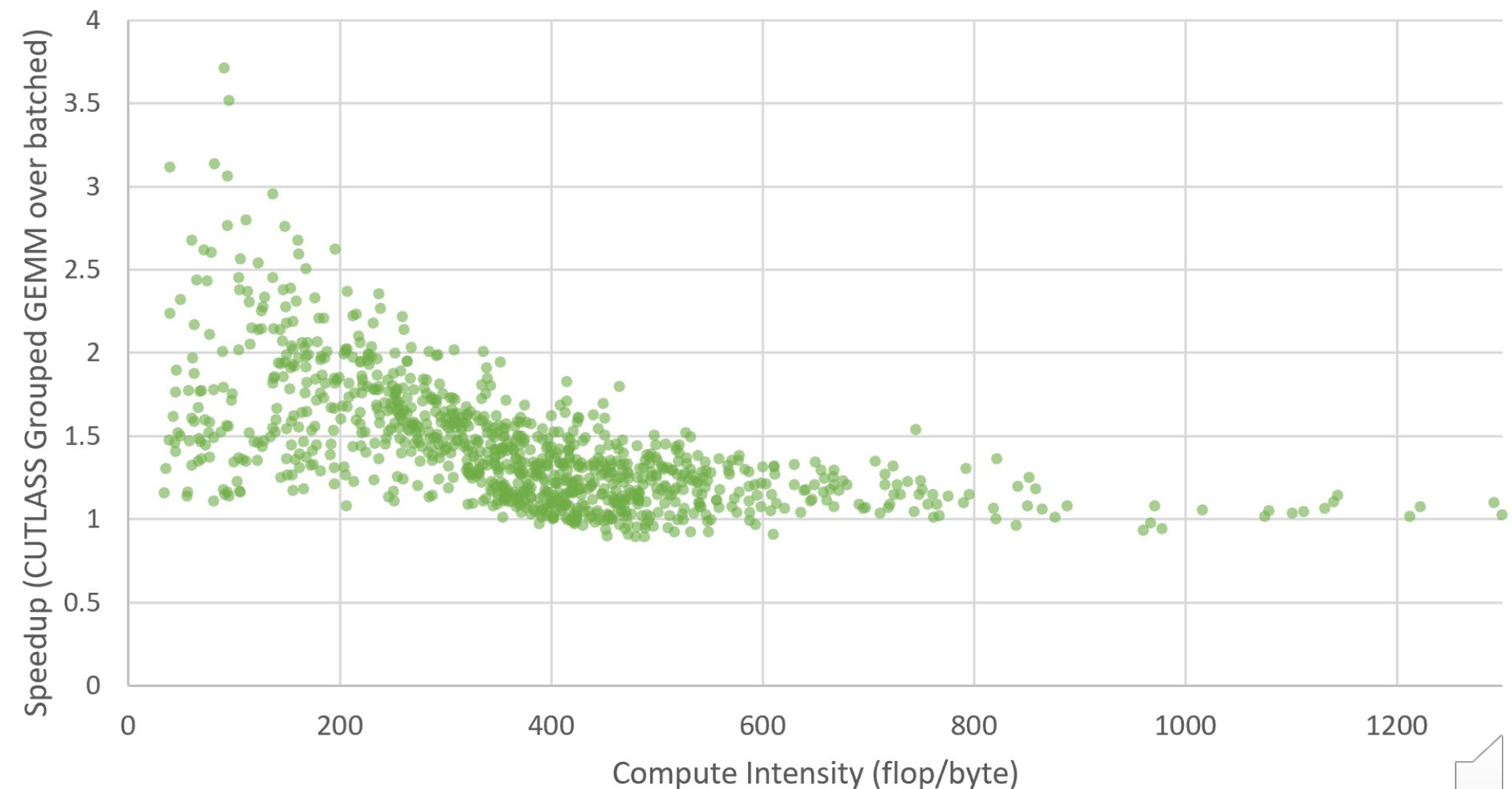
    // Fetch the problem size from memory
    GemmCoord problem_size = problem_visitor.problem_size();
    int32_t problem_idx    = problem_visitor.problem_index();
    int32_t cta_idx        = problem_visitor.threadblock_index();

    // Determine threadblock coordinates
    GemmCoord threadblock_offset
        = problem_visitor.threadblock_offset(cta_idx, problem_size);

    //
    // Compute a tile of the GEMM output using existing CUTLASS
    // pipelined matrix multiply.
    //
    mma( ... );

    problem_visitor.advance(gridDim.x);
}
```

Speedup of CUTLASS grouped GEMM vs Batched
A100 - Tensor Cores (F16 * F16 + F32)



CUTLASS PYTHON

Dynamic compilation of CUTLASS with CUDA Python

CUDA Python exposes the CUDA Driver API and NVRTC compiler to Python programmers

CUTLASS uses a Python-based IR to emit device-wide CONV and GEMM operators

A new host-side runtime component enables JIT compilation and launch of CUTLASS GEMM kernels from Python

```
# Manifest of cutlass operations
manifest = cutlass_manifest.Manifest()

# Construct a SGEMM operation
generator.GenerateSM50_Simt(manifest, "11.5.0")

operation = manifest.operations_by_name['cutlass_simt_sgemm_128x128_8x2_nt_align1']

# Construct the GEMM runtime component
sgemm = rt.Gemm(operation)

# Compile the CUTLASS GEMM operation and load as a CUDA module
architectures = [80,]

compilation_options = rt.CompilationOptions(architectures, include_paths)

module = rt.Module('module.cu', [ sgemm ], compilation_options)
```

```
# Initialize the SGEMM object
arguments = rt.GemmArguments()

# Pack arguments: problem size, pointers, and strides
arguments.problem_size = rt.GemmCoord(M, N, K)

arguments.A = rt.TensorRef(tensor_A_d, M)
arguments.B = rt.TensorRef(tensor_B_d, N)
arguments.C = rt.TensorRef(tensor_C_d, M)
arguments.D = rt.TensorRef(tensor_D_d, M)

host_workspace = bytearray(sgemm.get_host_workspace_size(arguments))

# Plan the CUDA grid launch and initialize the GEMM object
launch_config = sgemm.plan(arguments)
sgemm.initialize(host_workspace, None, launch_config, arguments)

# Launch the kernel
status = sgemm.run(host_workspace, device_workspace, launch_config)
```





CONCLUSION



CONCLUSION

CUTLASS convolutions

- CUTLASS convolution performance is on par with cuDNN 8.3.3 (> 95%)
- Strided Dgrad in CUTLASS 2.6 provides upto 4x speedup compared to CUTLASS 2.5
- Supports smaller than 128b alignment

Grouped GEMM

- Achieves 1.75x faster end to end performance for Mixture of Experts natural language model
- Variable batch size GEMMs grouped into single kernel
- Does not require pre-processing

Accelerated single-precision using 3xTF32 on Tensor Cores provides

- 2x performance gains vs. single-precision on CUDA cores
- Better accuracy vs. single-precision on CUDA cores

CUTLASS python example

- Looking forward to hearing more from python programmers

CUTLASS 2.9: Upcoming release lookout for it 😊

REFERENCES

GTC 2022 Sessions

[GTC 2022 \[S41491\]](#) Recent Developments in NVIDIA Math Libraries

[GTC 2022 \[S41486\]](#) CUDA: New Features and Beyond

[GTC 2022 \[S41606\]](#) Use CUTLASS to Fuse Multiple GEMMs to Extreme Performance

[GTC 2022 \[S41611\]](#) General Framework for Automatic Model Compression and Acceleration using Int4/Int8 Mixed Precision

GTC 2022 Connect with Experts (Live Session)

[\[CWE41721\]](#) Connect with the Experts: NVIDIA Math Libraries (Monday, March 21, 2:00 PM - 2:50 PM PDT)

GTC CUTLASS Sessions from Previous Years (2018 - 2021)

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

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

[GTC 2020 \[S21745\]](#) Developing CUDA kernels to push Tensor Cores to the Absolute Limit on NVIDIA A100

[GTC 2021 \[S31883\]](#) Accelerating convolution with Tensor Cores in CUTLASS

