



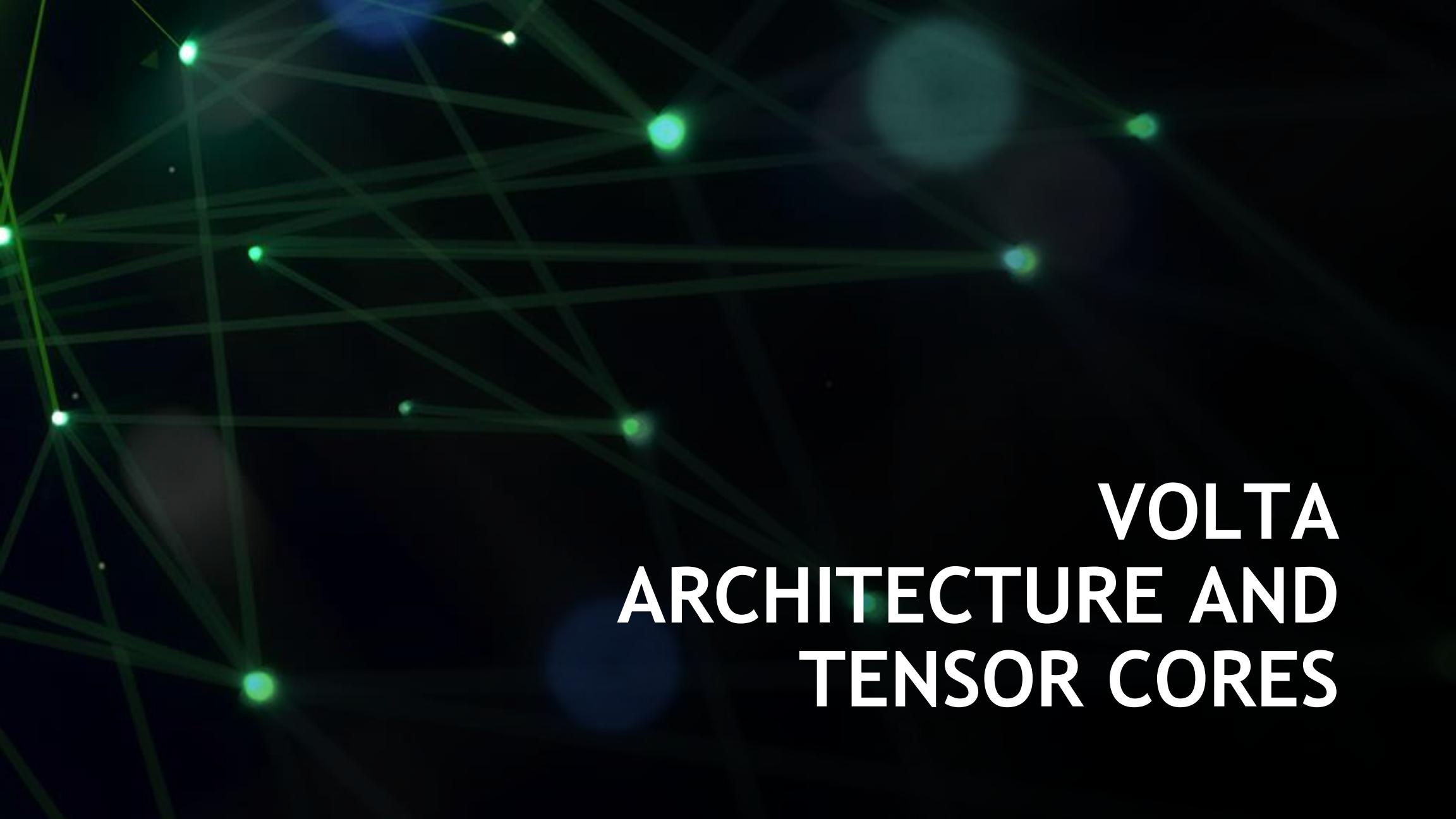
# VOLTA TENSOR CORE TRAINING

ORNL, August 2019



# AGENDA

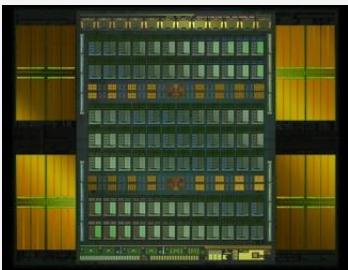
- V100 Architecture & Tensor Cores
- Anatomy of a GEMM
- Programming Approaches
  - Libraries
    - cublas
  - Iterative Refinement
- Frameworks
- WMMA & MMA.sync
- CUTLASS
- NVIDIA Tools
- Case Studies
  - Asgard + HPL-AI
  - PICTC
  - DL Framework
  - Non-Traditional Uses



# VOLTA ARCHITECTURE AND TENSOR CORES

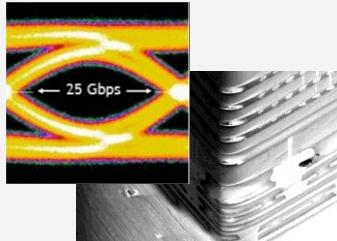
# TESLA V100

## Volta Architecture



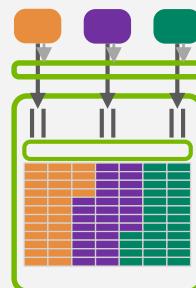
Most Productive GPU

## Improved NVLink & HBM2



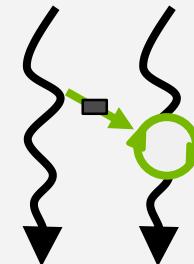
Efficient Bandwidth

## Volta MPS



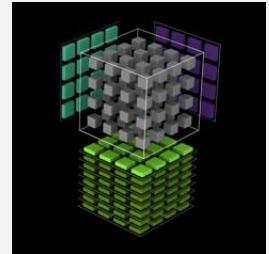
Inference Utilization

## Improved SIMT Model



New Algorithms

## Tensor Core



120 Programmable  
TFLOPS Deep Learning

The Fastest and Most Productive GPU for Deep Learning and HPC

# TESLA V100

# 21B transistors 815 mm<sup>2</sup>

**80 SM**  
**5120 CUDA Cores**  
**640 Tensor Cores**

**16 GB HBM2  
900 GB/s HBM2  
300 GB/s NVLink**

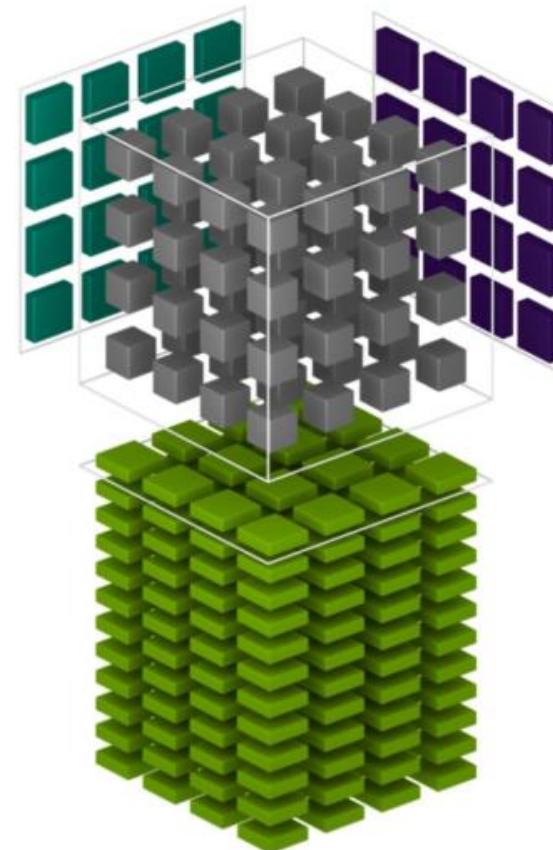


# VOLTA GV100 SM

GV100	
FP32 units	64
FP64 units	32
INT32 units	64
Tensor Cores	8
Register File	256 KB
Unified L1/Shared memory	128 KB
Active Threads	2048

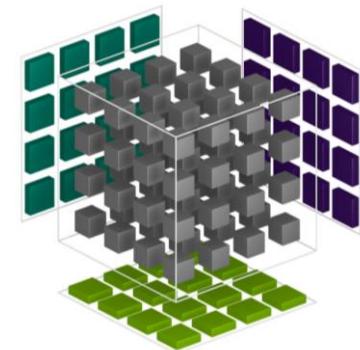


# VOLTA TENSOR CORE



# TENSOR CORE

Mixed Precision Matrix Math  
4x4 matrices

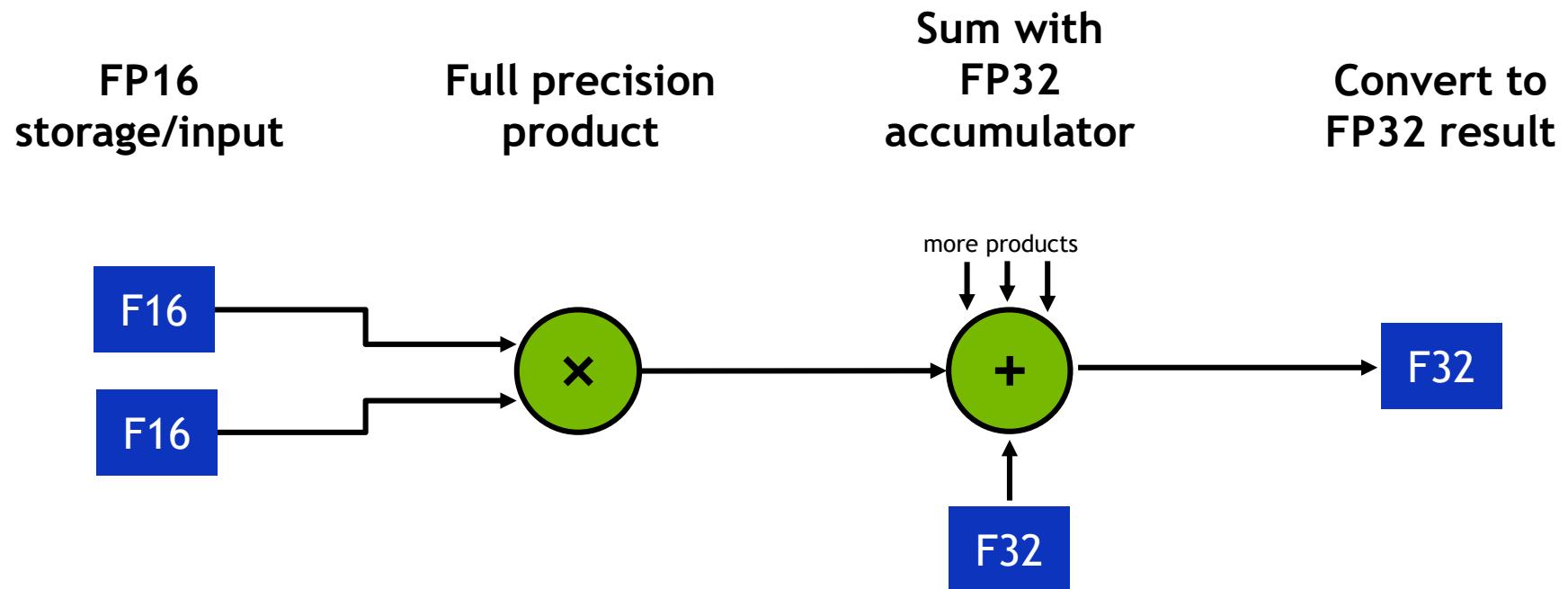


$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} + \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

FP16 or FP32                    FP16                    FP16                    FP16 or FP32

$$D = AB + C$$

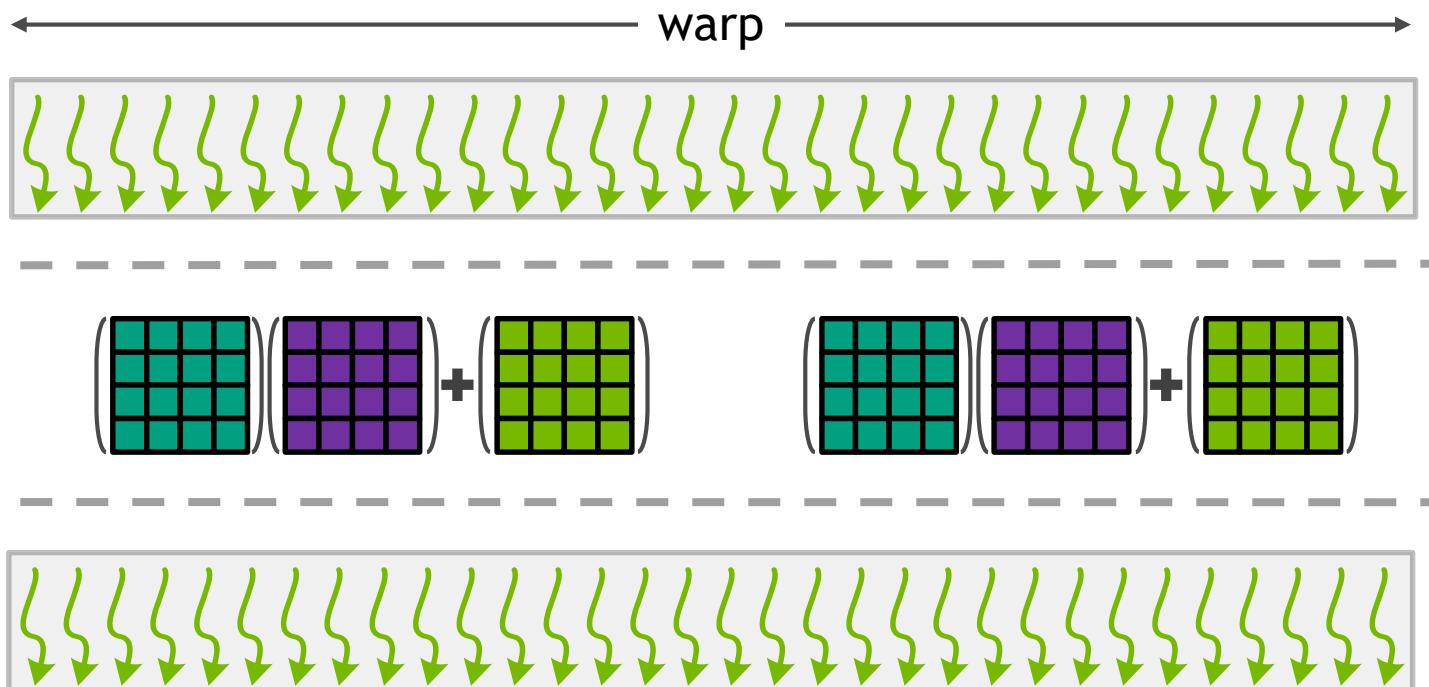
# VOLTA TENSOR OPERATION



*Also supports FP16 accumulator mode for inferencing*

# TENSOR SYNCHRONIZATION

## Full Warp 16x16 Matrix Math

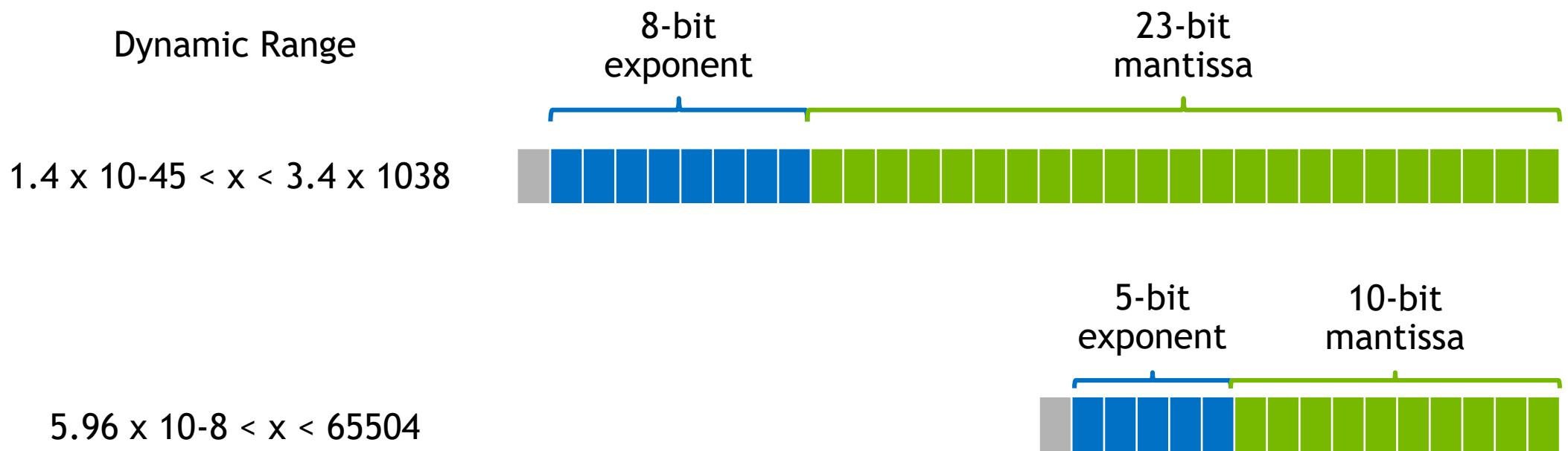


Warp-synchronizing operation

Composed Matrix Multiply and  
Accumulate for 16x16 matrices

Result distributed across warp

# FP32 AND FP16 REPRESENTATION





# EFFICIENT LINEAR ALGEBRA COMPUTATIONS ON GPUS

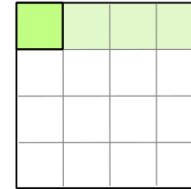
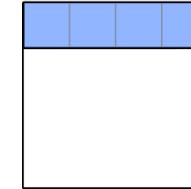
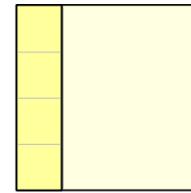
# GENERAL MATRIX PRODUCT

## Basic definition

General matrix product

$$C = \alpha \operatorname{op}(A) * \operatorname{op}(B) + \beta C$$

$C$  is  $M$ -by- $N$ ,  $\operatorname{op}(A)$  is  $M$ -by- $K$ ,  $\operatorname{op}(B)$  is  $K$ -by- $N$



Compute independent dot products

```
// Independent dot products
for (int i = 0; i < M; ++i)
    for (int j = 0; j < N; ++j)
        for (int k = 0; k < K; ++k)
            C[i][j] += A[i][k] * B[k][j];
```

Inefficient due to large working sets to hold parts of  $A$  and  $B$

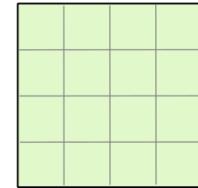
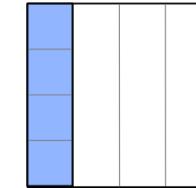
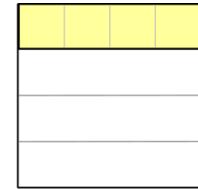
# GENERAL MATRIX PRODUCT

Accumulated outer products

General matrix product

$$C = \alpha \text{op}(A) * \text{op}(B) + \beta C$$

$C$  is  $M$ -by- $N$ ,  $\text{op}(A)$  is  $M$ -by- $K$ ,  $\text{op}(B)$  is  $K$ -by- $N$



~~Compute independent dot products~~

```
// Independent dot products
for (int i = 0; i < M; ++i)
    for (int j = 0; j < N; ++j)
        for (int k = 0; k < K; ++k)
            C[i][j] += A[i][k] * B[k][j];
```



Permute loop nests

```
// Accumulated outer products
for (int k = 0; k < K; ++k)
    for (int i = 0; i < M; ++i)
        for (int j = 0; j < N; ++j)
            C[i][j] += A[i][k] * B[k][j];
```

Load elements of  $A$  and  $B$  exactly once

# GENERAL MATRIX PRODUCT

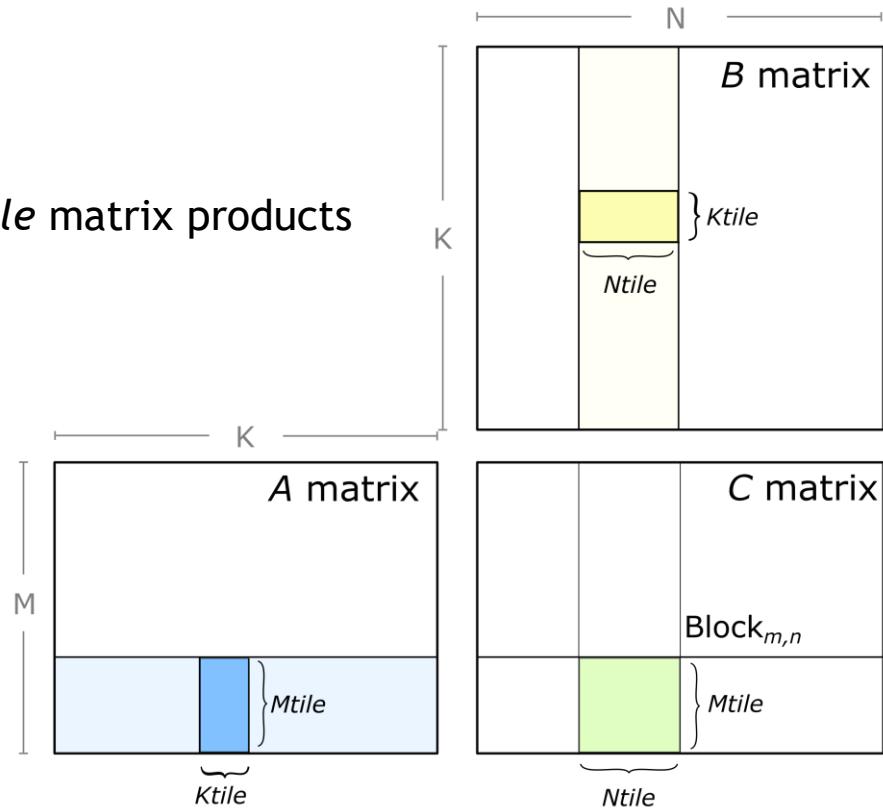
Computing matrix product one block at a time

Partition the loop nest into *blocks* along each dimension

- Partition into *Mtile*-by-*Ntile* independent matrix products
- Compute each product by accumulating *Mtile*-by-*Ntile*-by-*Ktile* matrix products

```
for (int mb = 0; mb < M; mb += Mtile)
    for (int nb = 0; nb < N; nb += Ntile)
        for (int kb = 0; kb < K; kb += Ktile)
    {
        // compute Mtile-by-Ntile-by-Ktile matrix product
        for (int k = 0; k < Ktile; ++k)
            for (int i = 0; i < Mtile; ++i)
                for (int j = 0; j < Ntile; ++j)
                {
                    int row = mb + i;
                    int col = nb + j;

                    C[row][col] +=
                        A[row][kb + k] * B[kb + k][col];
                }
    }
```



# BLOCKED GEMM IN CUDA

## Parallelism Among CUDA Thread Blocks

Launch a CUDA kernel grid

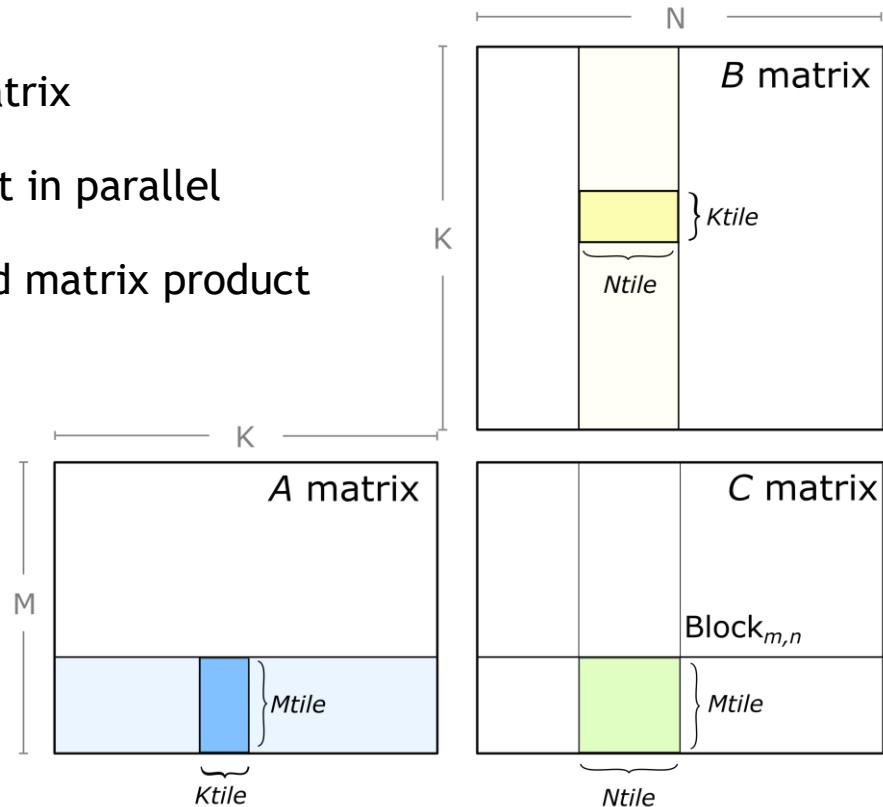
- Assign CUDA thread blocks to each partition of the output matrix

CUDA thread blocks compute  $Mtile$ -by- $Ntile$ -by- $K$  matrix product in parallel

- Iterate over  $K$  dimension in steps, performing an accumulated matrix product

```
for (int mb = 0; mb < M; mb += Mtile)
    for (int nb = 0; nb < N; nb += Ntile)
        for (int kb = 0; kb < K; kb += Ktile)
        {
            .. compute  $Mtile$  by  $Ntile$  by  $Ktile$  GEMM
        }
```

by each CUDA thread block



# THREAD BLOCK TILE STRUCTURE

## Parallelism Within a CUDA Thread Block

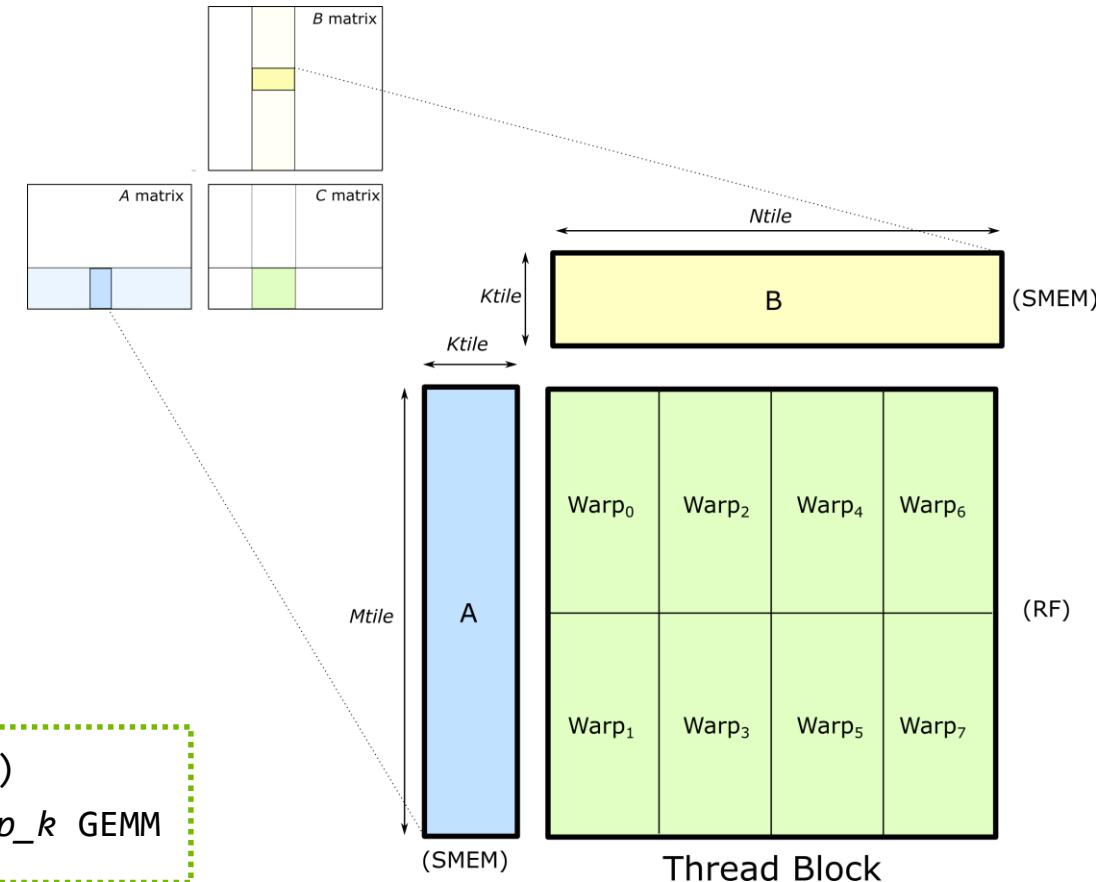
Decompose thread block into warp-level tiles

- Load **A** and **B** operands into Shared Memory (reuse)
- **C** matrix distributed among warps

Each warp computes an independent matrix product

```
for (int kb = 0; kb < K; kb += Ktile)
{
    .. load A and B tiles to shared memory

    for (int m = 0; m < Mtile; m += warp_m)
        for (int n = 0; n < Ntile; n += warp_n)
            for (int k = 0; k < Ktile; k += warp_k)
                .. compute warp_m by warp_n by warp_k GEMM
}
by each CUDA warp
```



# WARP-LEVEL TILE STRUCTURE

## Warp-level matrix product

Warps perform an accumulated matrix product

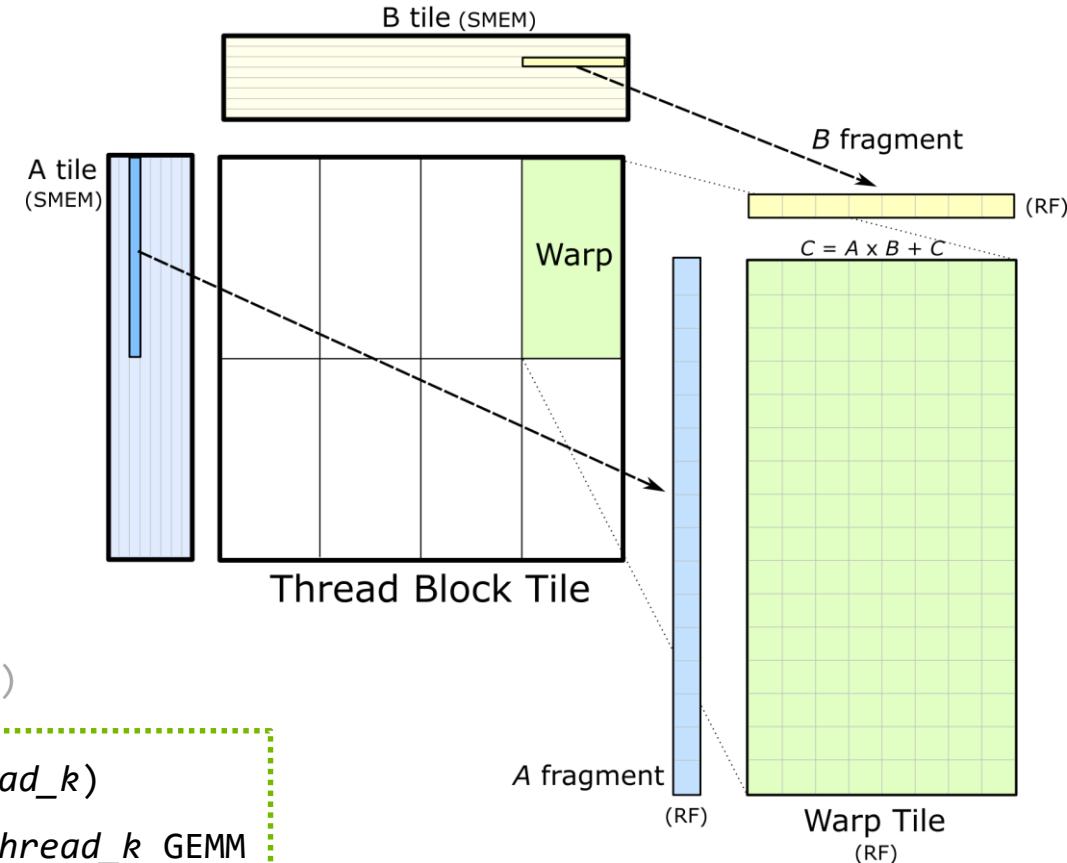
- Load **A** and **B** operands from SMEM into registers
- **C** matrix held in registers of participating threads

Shared Memory layout is  $K$ -strided for efficient loads

```
for (int k = 0; k < Ktile; k += warp_k)
{
    .. load A tile from SMEM into registers
    .. load B tile from SMEM into registers

    for (int tm = 0; tm < warp_m; tm += thread_m)
        for (int tn = 0; tn < warp_n; tn += thread_n)

            for (int tk = 0; tk < warp_k; tk += thread_k)
                .. compute thread_m by thread_n by thread_k GEMM
}
by each CUDA thread
```



# THREAD-LEVEL TILE STRUCTURE

## Parallelism within a thread

Threads compute accumulated matrix product

- $A$ ,  $B$ , and  $C$  held in registers

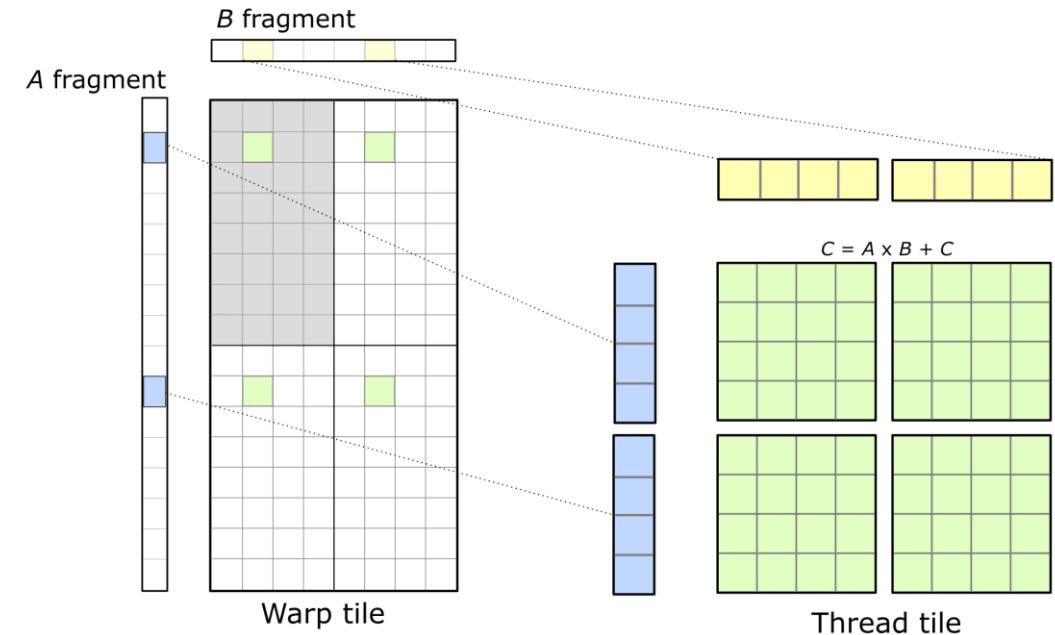
Opportunity for data reuse:

- $O(M^2)$  computations on  $O(M+N)$  elements

```
for (int m = 0; m < thread_m; ++m)
    for (int n = 0; n < thread_n; ++n)

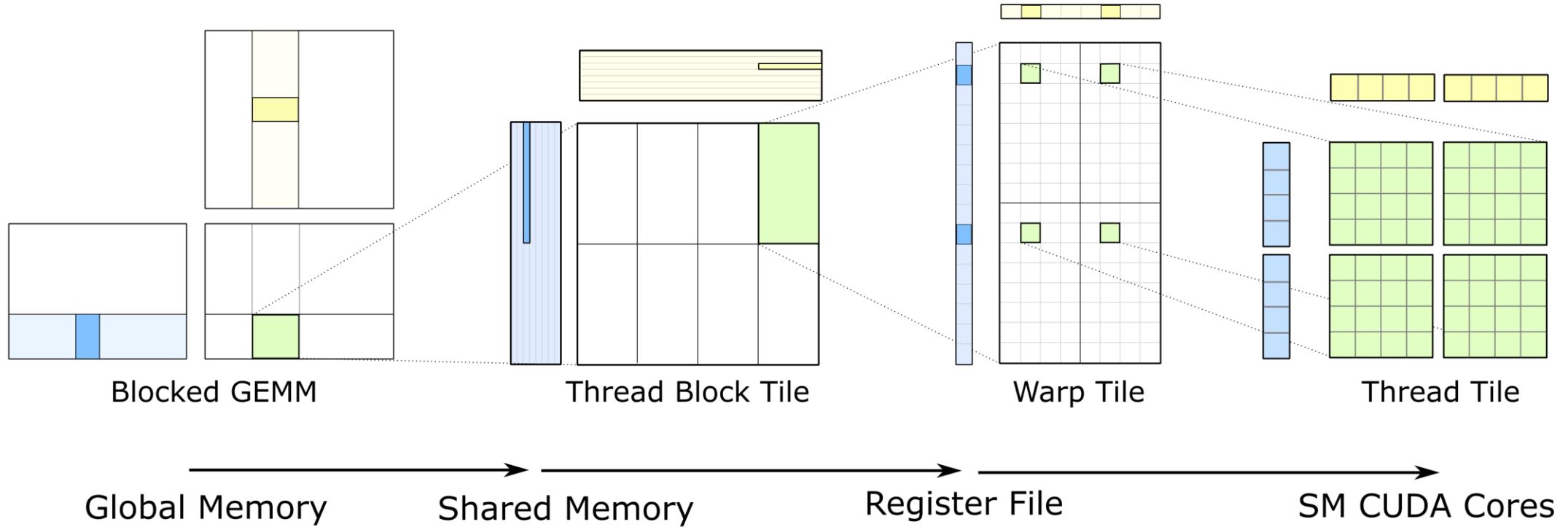
        for (int k = 0; k < thread_k; ++k)
            C[m][n] += A[m][k] * B[n][k];
```

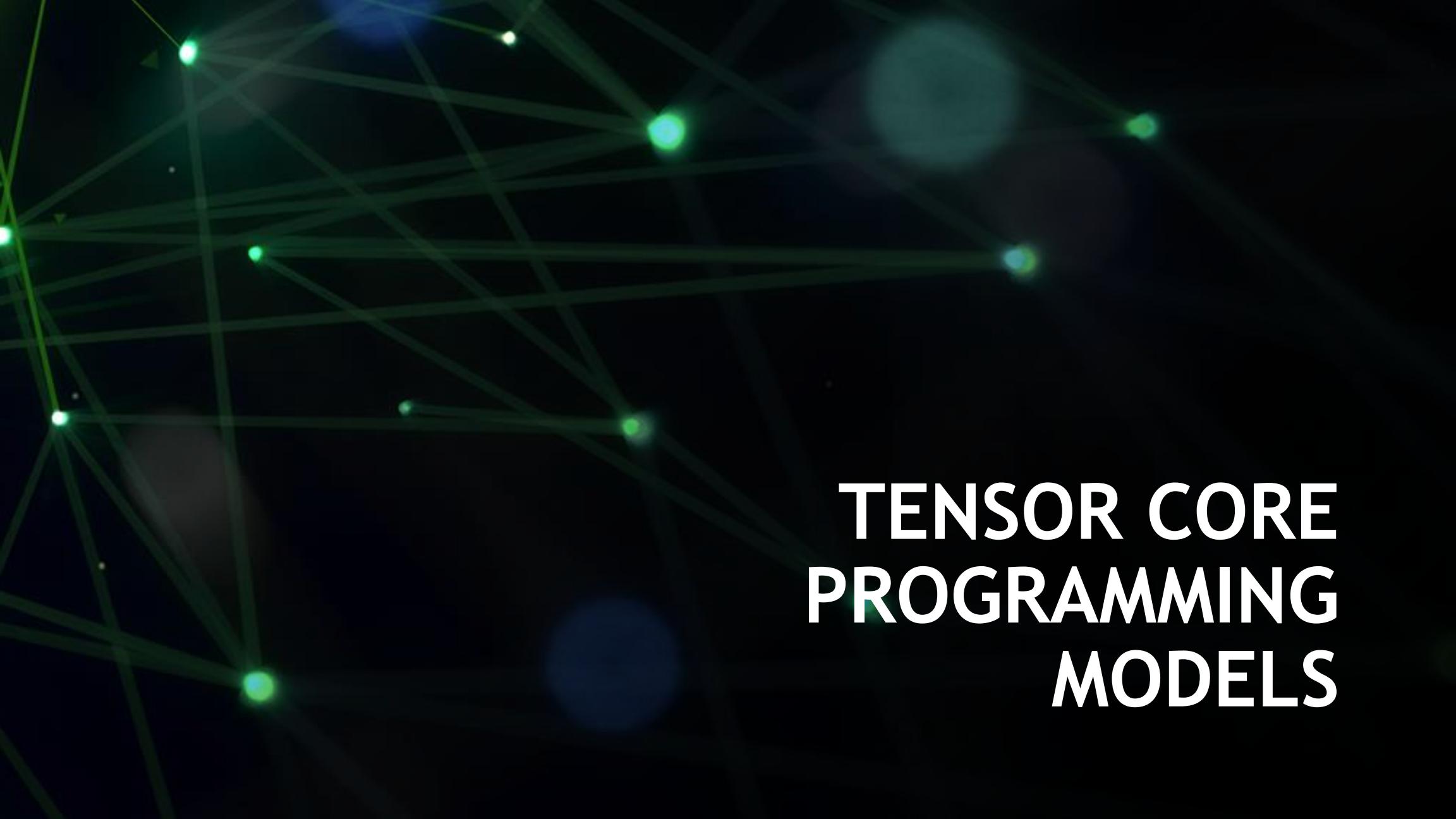
Fused multiply-accumulate instructions



# COMPLETE GEMM HIERARCHY

Data reuse at each level of the memory hierarchy





# TENSOR CORE PROGRAMMING MODELS

# USING TENSOR CORES



**Volta Optimized  
Frameworks and Libraries**

```
__device__ void tensor_op_16_16_16(
    float *d, half *a, half *b, float *c)
{
    wmma::fragment<matrix_a, ...> Amat;
    wmma::fragment<matrix_b, ...> Bmat;
    wmma::fragment<matrix_c, ...> Cmat;

    wmma::load_matrix_sync(Amat, a, 16);
    wmma::load_matrix_sync(Bmat, b, 16);
    wmma::fill_fragment(Cmat, 0.0f);

    wmma::mma_sync(Cmat, Amat, Bmat, Cmat);

    wmma::store_matrix_sync(d, Cmat, 16,
                           wmma::row_major);
}
```

**CUDA C++  
Warp-Level Matrix Operations**

# CUBLAS TENSOR CORE HOW-TO

	mathMode = CUBLAS_DEFAULT_MATH	mathMode = CUBLAS_TENSOR_OP_MATH
cublasHgemm, cublasSgemm, cublasGemmEx(algo=DEFAULT)	Disallowed	Allowed
cublasGemmEx(algo=*_TENSOR_OP	Allowed	Allowed

Math Mode set with **cublasSetMathMode** function.

Volta and Turing family Tensor Core can be used with in mixed precision (FP16 inputs, FP32 accumulation, FP16 or FP32 output) routines.

Pure single precision routines use tensor core (when allowed) by down-converting inputs to half (FP16) precision on the fly.

Constraint: M,N,K,LDA,LDB,LDC and A,B,C pointers must ALL be aligned to 8 because of high memory bandwidth needed to efficiently use Tensor Cores.

# CUBLAS FUTURE IMPROVEMENTS

- Loosening constraints on Tensor Core usage:
  1. CUDA 10.1 Update 2 will lift some restrictions so that only requirements remaining are: $m \% 4 == 0$  $k \% 8 == 0$  $lda, ldb, ldc, A, B, C$  are aligned to 16 bytes,
  2. Plan to lift the restriction completely by adding new kernels to work on mis-aligned memory in a future release.
- Plans to make Tensor Core “opt-out” instead of “opt-in” for all directly applicable data type combinations.
- Plans to add NVTX based feedback to add information on tensor-core usage for detailed profiling.

# CUBLASLT: NEW MATRIX MULTIPLICATION LIBRARY

- Has its own header file, binary and lightweight context
- Intended for power users of GEMMs that need advanced features and optimizations for their workflows
  - cuBLASL<sup>t</sup> is not a replacement for cuBLAS
- Adds flexibility in:
  - new matrix data layouts: IMMA, and planar complex (Tensor Ops)
  - algorithmic implementation choices and heuristics
- Workspace support enables new optimizations - e.g. split-k
- Non-traditional memory ordering enables hardware optimizations such as INT8 IMMA on Turing GPUs

```
#include <cublasLt.h>

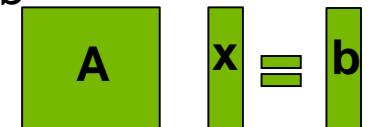
cublasLtCreate()
cublasLtMatmul()
cublasLtMatmulAlgoGetHeuristic()
cublasLtMatmulAlgoConfigSetAttribute()
```

# TENSOR CORE ACCELERATED IRS SOLVING LINEAR SYSTEM $AX = B$

solving linear system  $Ax = b$   
LU factorization

- LU factorization is used to solve a linear system  $Ax=b$

$$A \quad x = b$$

A diagram showing the LU factorization of a matrix A. On the left is a green square labeled 'A'. To its right is an equals sign followed by a green vertical bar labeled 'x' to its left, which is followed by another equals sign and a green vertical bar labeled 'b' to its left.

$$LUx = b$$

A diagram showing the LU factorization of a matrix A. On the left is a green square divided diagonally from top-left to bottom-right, with 'L' in the lower-left triangle and 'U' in the upper-right triangle. To its right is an equals sign followed by a green vertical bar labeled 'x' to its left, which is followed by another equals sign and a green vertical bar labeled 'b' to its left.

$$Ly = b$$

A diagram showing the LU factorization of a matrix A. On the left is a green triangle labeled 'L' at its apex. To its right is an equals sign followed by a green vertical bar labeled 'y' to its left, which is followed by another equals sign and a green vertical bar labeled 'b' to its left.

$$\text{then } Ux = y$$

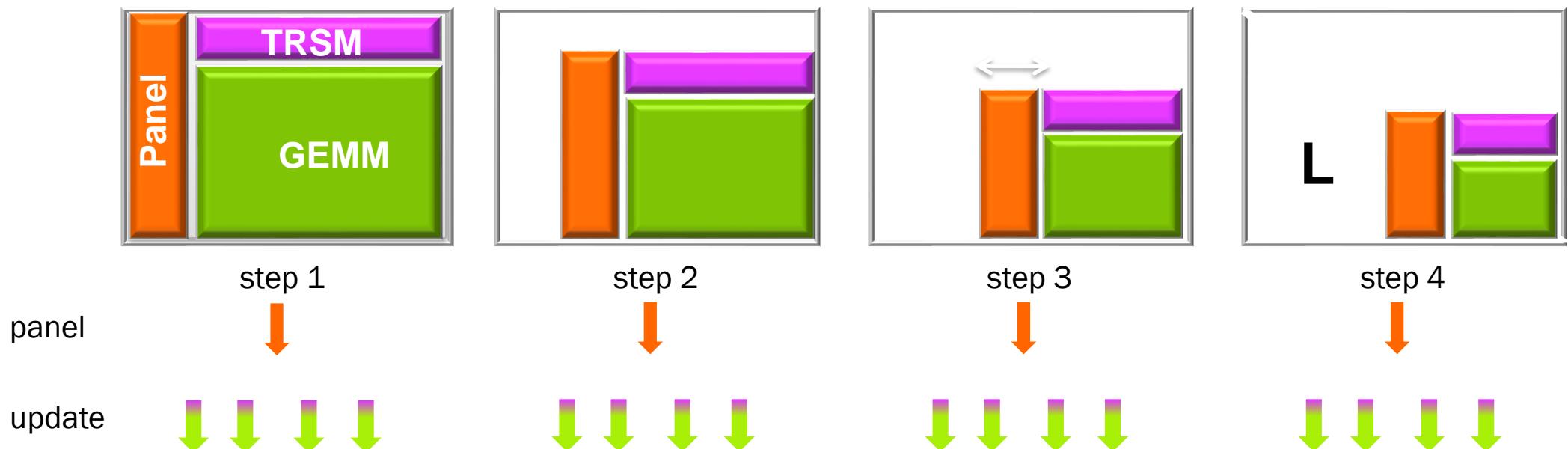
A diagram showing the LU factorization of a matrix A. On the left is a green triangle labeled 'U' at its apex. To its right is an equals sign followed by a green vertical bar labeled 'x' to its left, which is followed by another equals sign and a green vertical bar labeled 'y' to its left.

# TENSOR CORE ACCELERATED IRS SOLVING LINEAR SYSTEM $AX = B$

For  $s = 0, nb, \dots N$

1. panel factorize
2. update trailing matrix

LU factorization requires  $O(n^3)$   
most of the operations are spent in GEMM

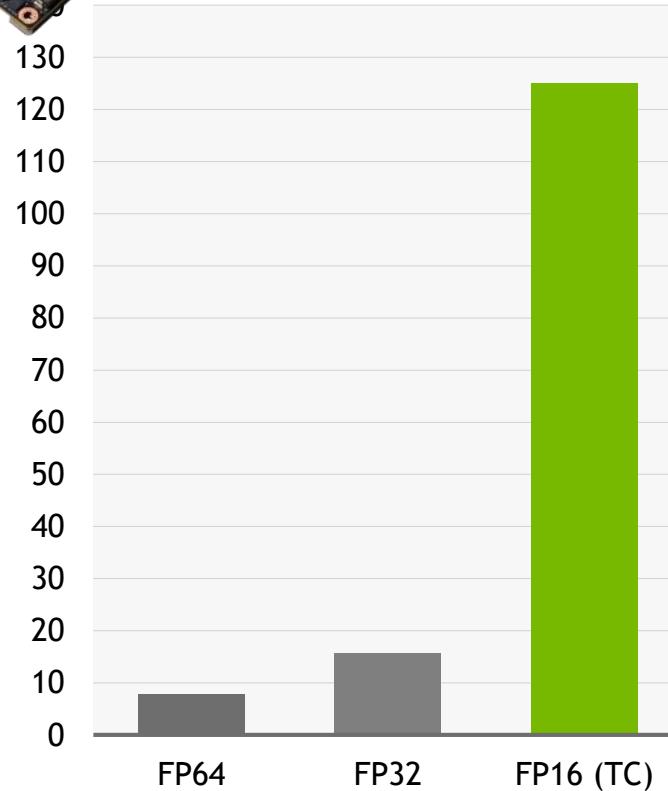


# TENSOR CORE ACCELERATED LIBRARIES

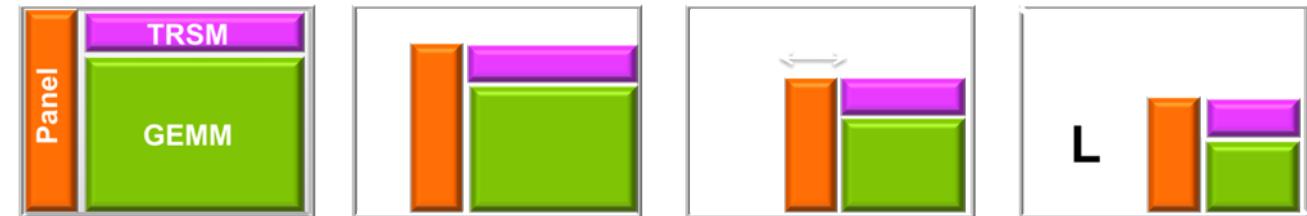
Multi-precision numerical methods



V100 TFLOPS

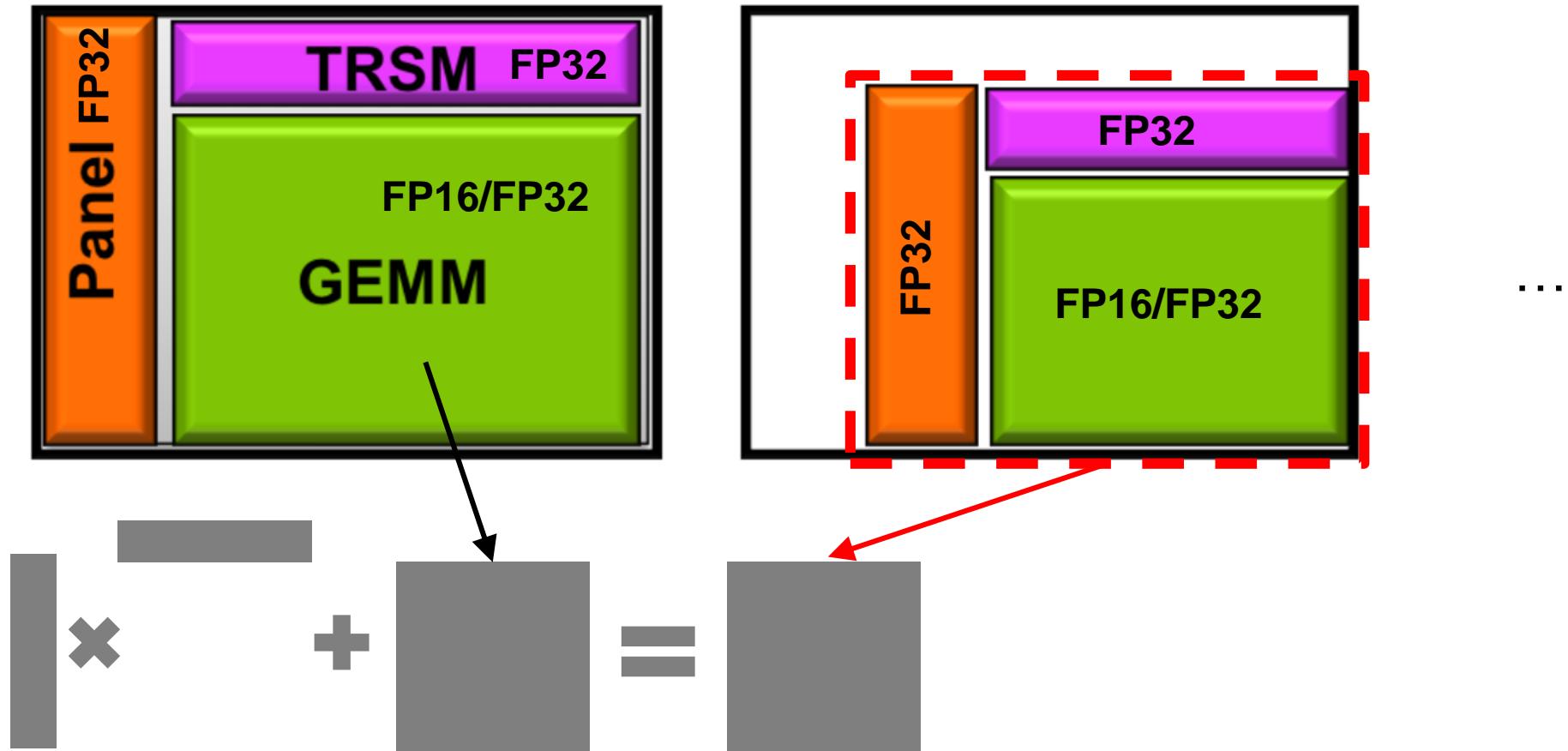


LU factorization used to solve  $Ax=b$  is dominated by GEMMs

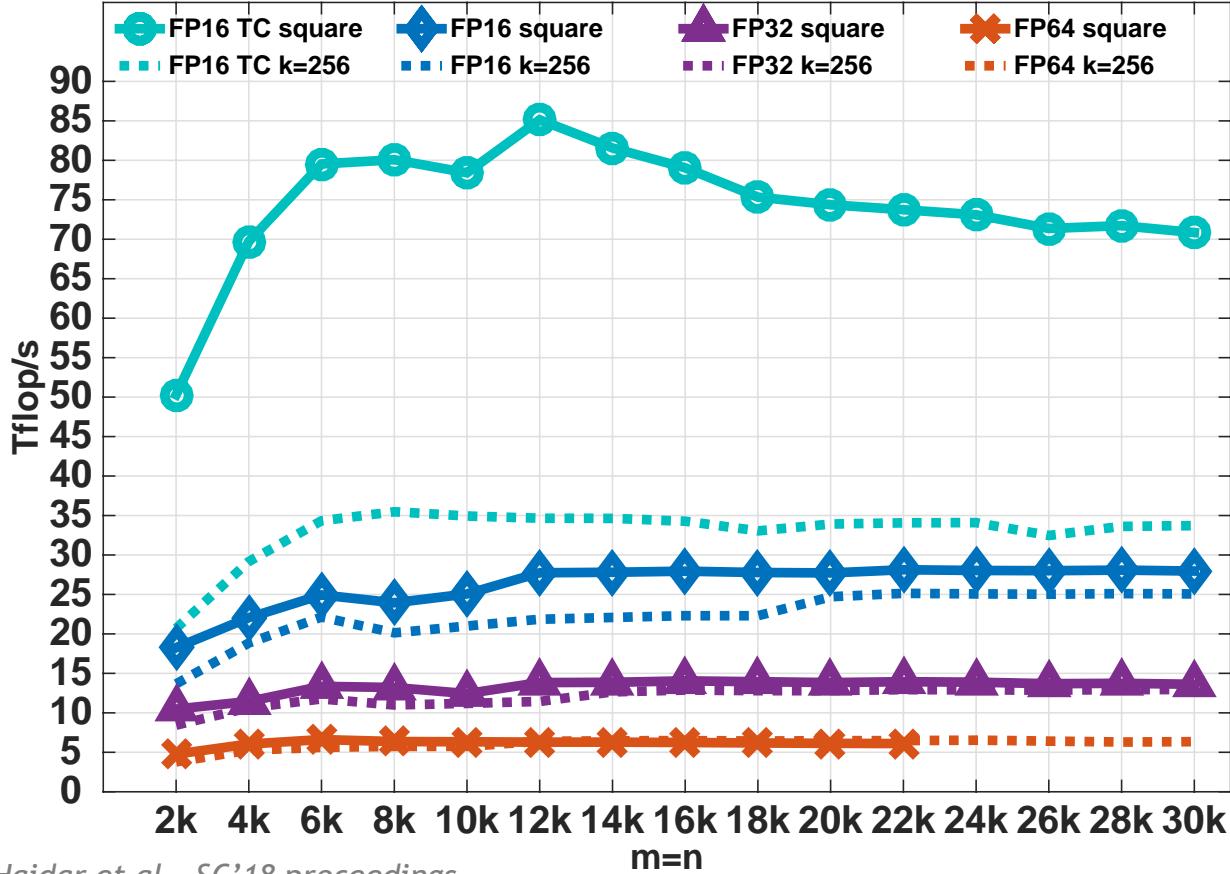


Can it be accelerated using Tensor Cores  
and still get fp64 accuracy?

# DIFFERENT LEVELS OF PRECISIONS USED DURING FACTORIZATION WITH TENSOR CORES



# STUDY OF THE MATRIX MATRIX MULTIPLICATION KERNEL ON NVIDIA V100

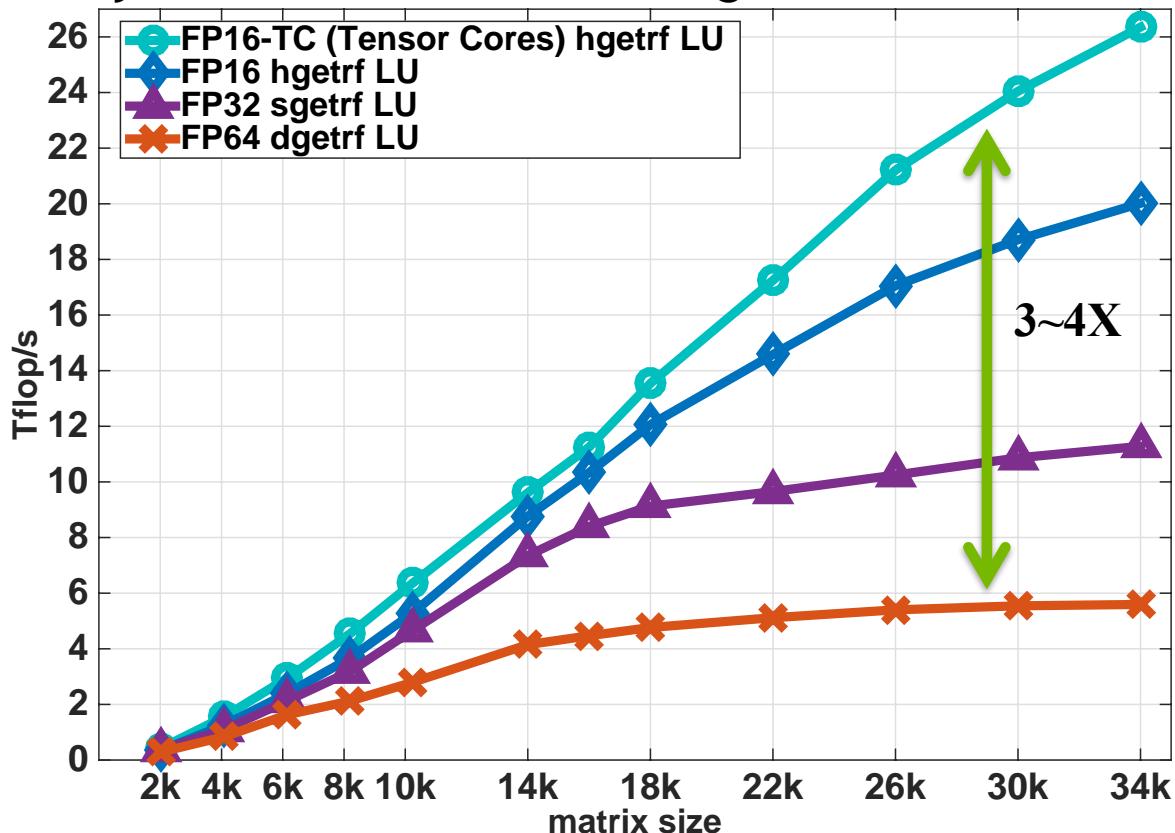


- dgemm achieve about 6.4 Tflop/s
- sgemm achieve about 14 Tflop/s
- hgemm achieve about 27 Tflop/s
- Tensor cores gemm reach about 85 Tflop/s
- Rank-k GEMM needed by LU does not perform as well as square but still OK

$$\boxed{\text{---}} = \boxed{\text{---}} + \boxed{|} \times \boxed{\text{---}}$$

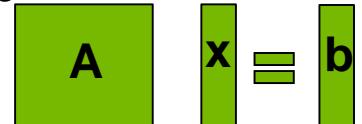
# LEVERAGING HALF PRECISION IN HPC ON V100 MOTIVATION

Study of the LU factorization algorithm on Nvidia V100



- LU factorization is used to solve a linear system  $Ax=b$

$$A \quad x = b$$



$$LUX = b$$



$$Ly = b$$



$$Ux = y$$

$$Ux = y$$

# Leveraging Tensor Cores

## Iterative Refinement Solver

Main idea is to use lower precision to compute the expensive flops (**LU  $O(n^3)$** ) and then iteratively refine the solution in order to achieve the FP64 arithmetic

$$\begin{aligned} L \ U &= \text{lu}(A) \\ x &= U \backslash (L \backslash b) \\ r &= b - Ax \end{aligned}$$

lower precision	$O(n^3)$
lower precision	$O(n^2)$
FP64 precision	$O(n^2)$

**WHILE** || r || not small enough

1. find a correction “z” to adjust x that satisfy Az=r  
solving Az=r could be done by either:

➤ $z = U \backslash (L \backslash r)$	lower precision	$O(n^2)$
➤ <u>GMRes</u> with LU preconditioner to solve <u>Az=r</u>	lower precision	$O(n^2)$

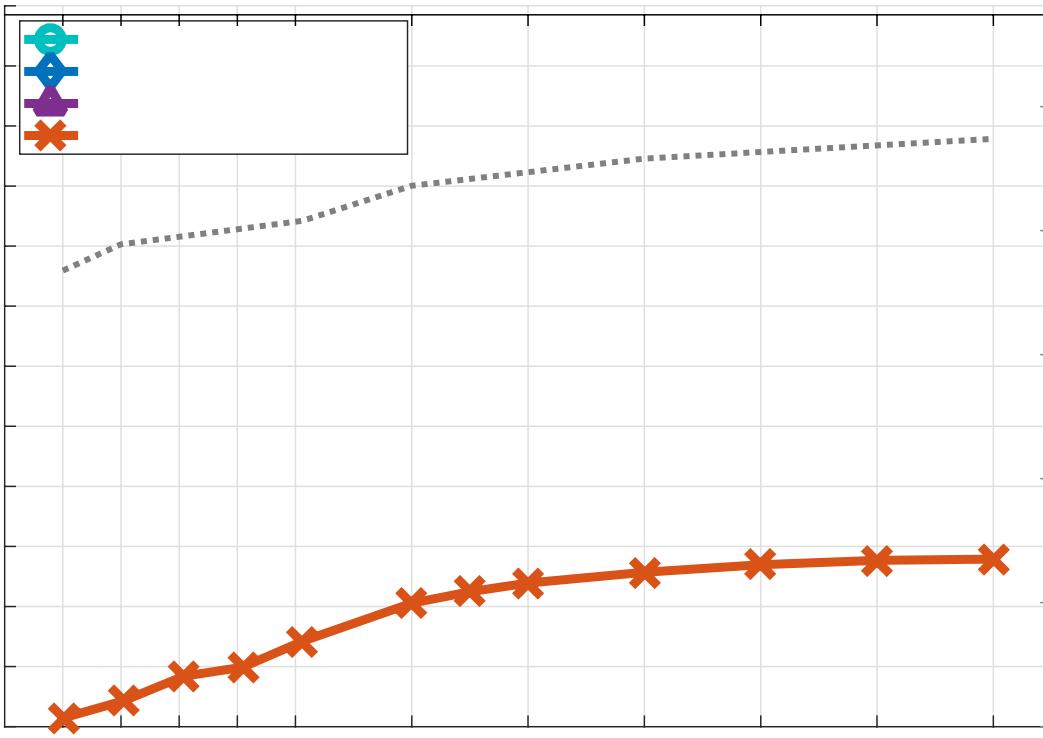
2.  $x = x + z$
3.  $r = b - Ax$

lower precision	$O(n^2)$
FP64 precision	$O(n^1)$
FP64 precision	$O(n^2)$

**END**

- Wilkinson, Moler, Stewart, & Higham provide error bound for SP fl pt results when using DP fl pt.
- E. Carson and N. J. Higham. Accelerating the solution of linear systems by iterative refinement in three precisions.
- It can be shown that using this approach we can compute the solution with residual similar to the 64-bit floating point precision.

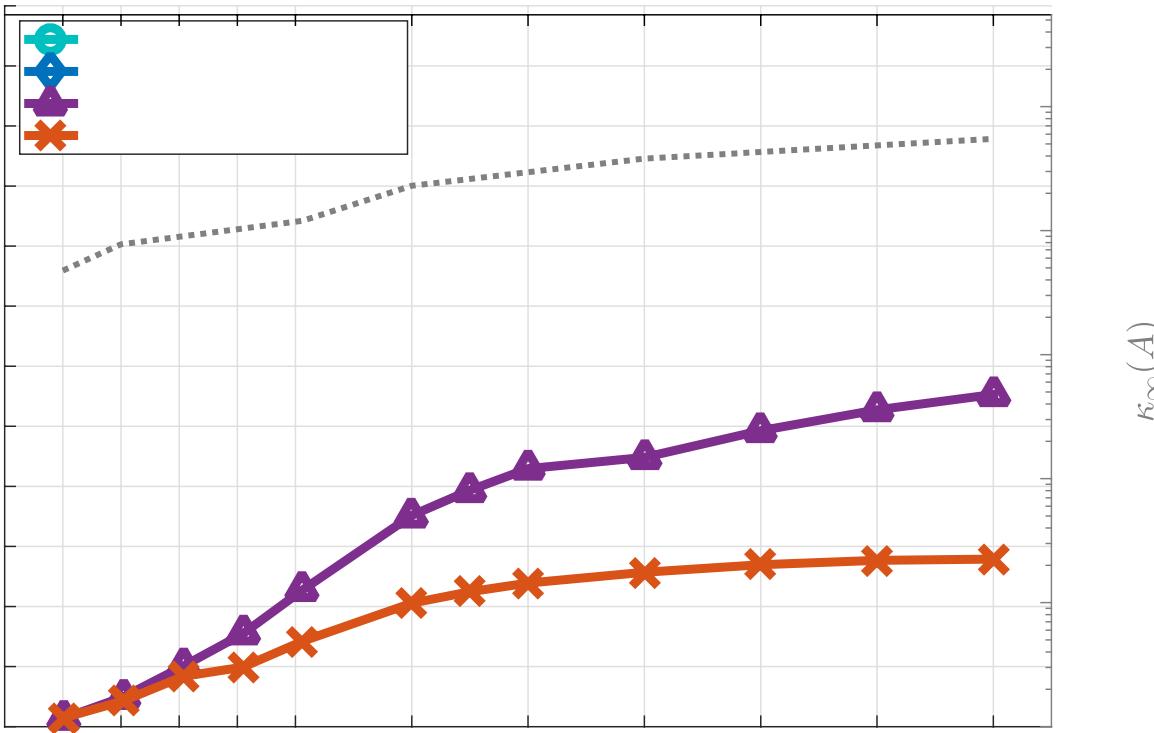
# Tensor Core Accelerated IRS solving linear system $Ax = b$ Performance Behavior



Flops =  $2n^3/(3 \text{ time})$   
meaning twice higher is twice faster

- solving  $Ax = b$  using **FP64 LU**

# Tensor Core Accelerated IRS solving linear system $Ax = b$ Performance Behavior

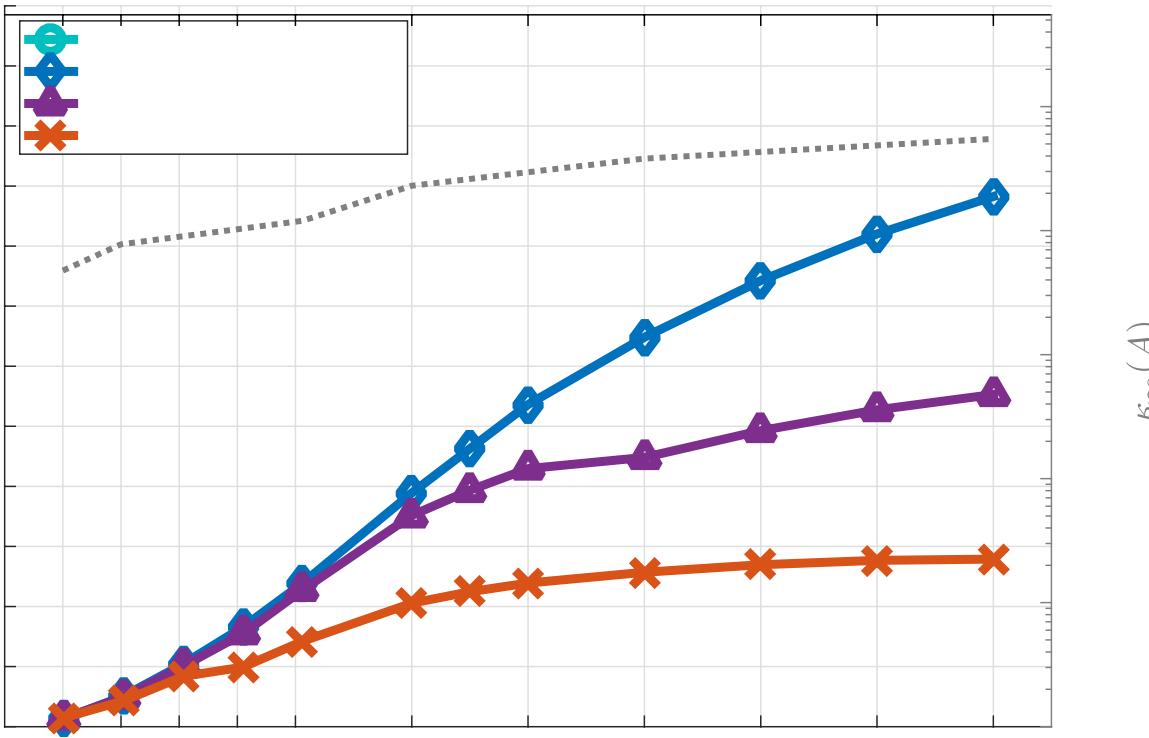


Flops =  $2n^3/(3 \text{ time})$   
meaning twice higher is twice faster

- solving  $Ax = b$  using **FP64 LU**
- solving  $Ax = b$  using **FP32 LU** and iterative refinement to achieve FP64 accuracy

Problem generated with an arithmetic distribution of the singular values  $s_i = 1 - (\frac{i-1}{n-1})(1 - \frac{1}{cond})$  and positive eigenvalues.

# Tensor Core Accelerated IRS solving linear system $Ax = b$ Performance Behavior

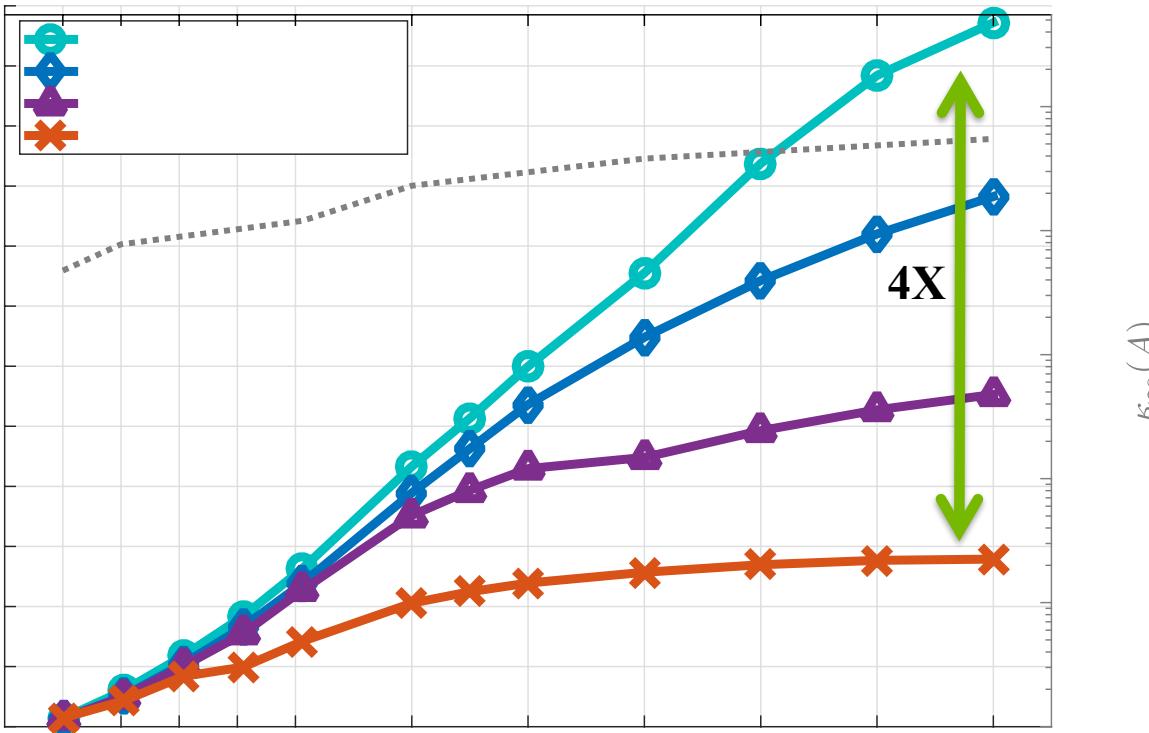


Flops =  $2n^3/(3 \text{ time})$   
meaning twice higher is twice faster

- solving  $Ax = b$  using **FP64 LU**
- solving  $Ax = b$  using **FP32 LU** and iterative refinement to achieve FP64 accuracy
- solving  $Ax = b$  using **FP16 LU** and iterative refinement to achieve FP64 accuracy

Problem generated with an arithmetic distribution of the singular values  $s_i = 1 - (\frac{i-1}{n-1})(1 - \frac{1}{\text{cond}})$  and positive eigenvalues.

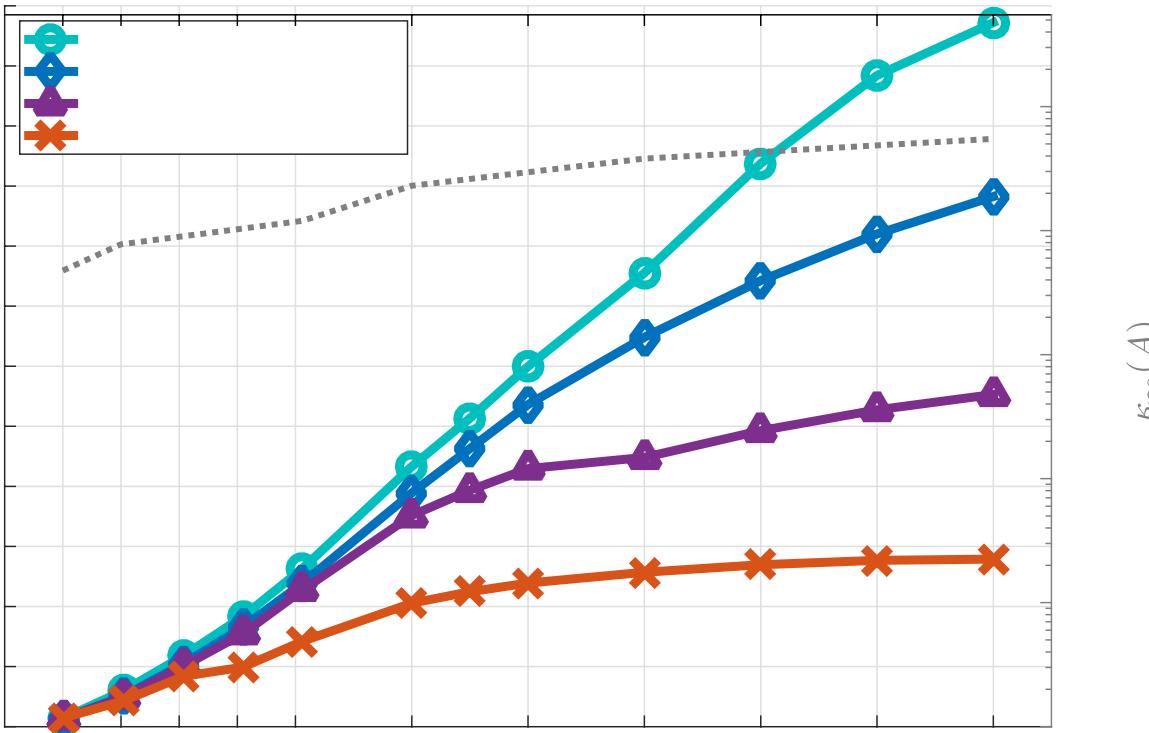
# Tensor Core Accelerated IRS solving linear system $Ax = b$ Performance Behavior



Flops =  $2n^3/(3 \text{ time})$   
meaning twice higher is twice faster

- solving  $Ax = b$  using **FP64 LU**
- solving  $Ax = b$  using **FP32 LU** and iterative refinement to achieve FP64 accuracy
- solving  $Ax = b$  using **FP16 LU** and iterative refinement to achieve FP64 accuracy
- solving  $Ax = b$  using **FP16 Tensor Cores LU** and iterative refinement to achieve FP64 accuracy

# Tensor Core Accelerated IRS solving linear system $Ax = b$ Performance Behavior

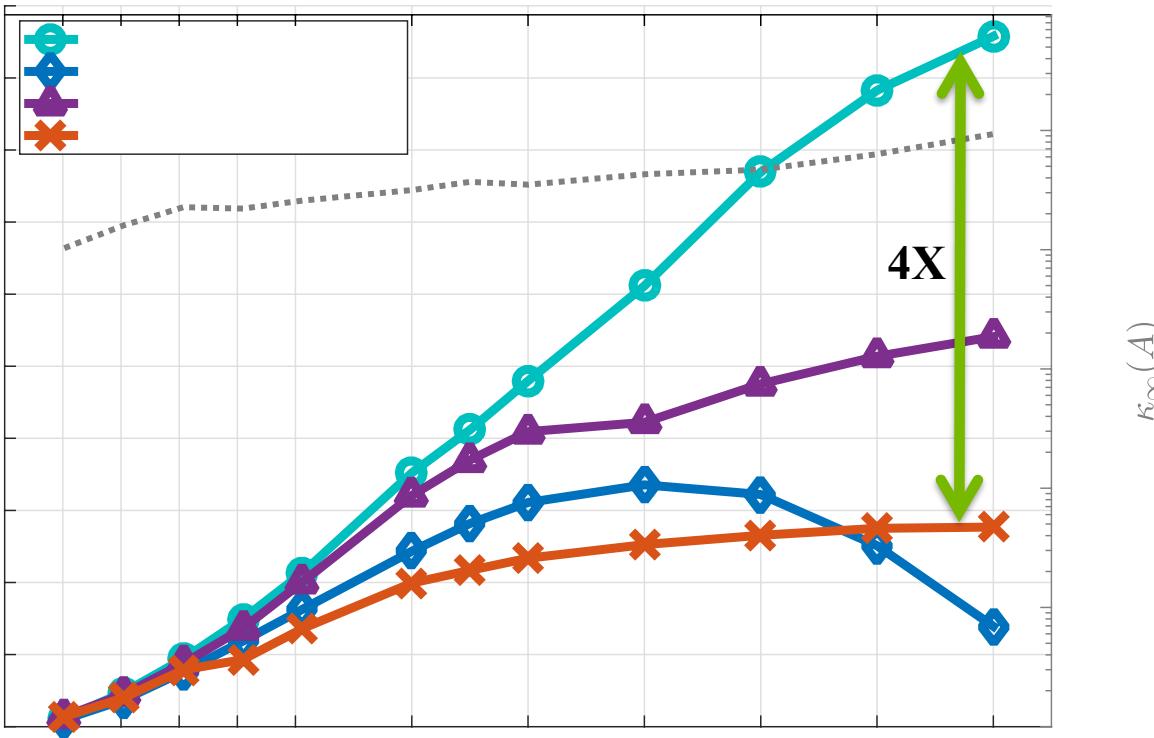


Flops =  $2n^3/(3 \text{ time})$   
meaning twice higher is twice faster

- solving  $Ax = b$  using **FP64 LU**
- solving  $Ax = b$  using **FP32 LU** and iterative refinement to achieve FP64 accuracy
- solving  $Ax = b$  using **FP16 LU** and iterative refinement to achieve FP64 accuracy
- solving  $Ax = b$  using **FP16 Tensor Cores LU** and iterative refinement to achieve FP64 accuracy

Problem generated with an arithmetic distribution of the singular values  $s_i = 1 - (\frac{i-1}{n-1})(1 - \frac{1}{cond})$  and positive eigenvalues.

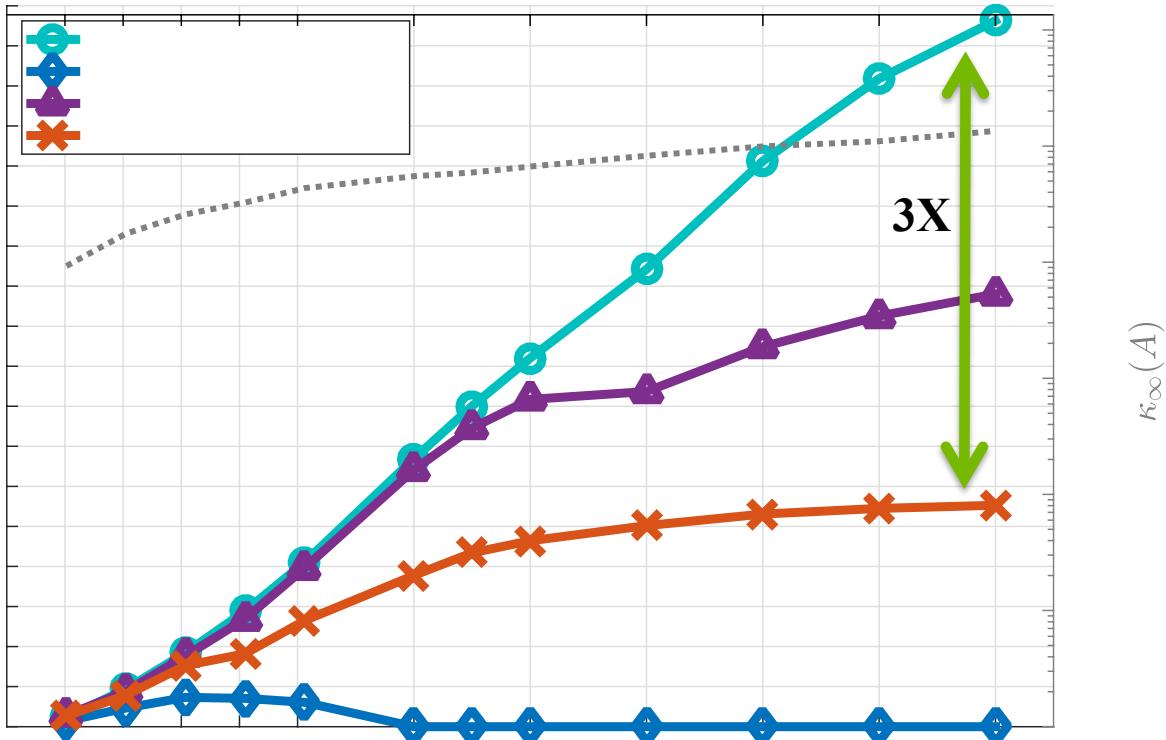
# Tensor Core Accelerated IRS solving linear system $Ax = b$ Performance Behavior



Flops =  $2n^3/(3 \text{ time})$   
meaning twice higher is twice faster

- solving  $Ax = b$  using **FP64 LU**
- solving  $Ax = b$  using **FP32 LU** and iterative refinement to achieve FP64 accuracy
- solving  $Ax = b$  using **FP16 LU** and iterative refinement to achieve FP64 accuracy
- solving  $Ax = b$  using **FP16 Tensor Cores LU** and iterative refinement to achieve FP64 accuracy

# Tensor Core Accelerated IRS solving linear system $Ax = b$ Performance Behavior



Flops =  $2n^3/(3 \text{ time})$   
meaning twice higher is twice faster

- solving  $Ax = b$  using **FP64 LU**
- solving  $Ax = b$  using **FP32 LU** and iterative refinement to achieve FP64 accuracy
- solving  $Ax = b$  using **FP16 LU** and iterative refinement to achieve FP64 accuracy
- solving  $Ax = b$  using **FP16 Tensor Cores LU** and iterative refinement to achieve FP64 accuracy

# Convergence Checks

Is the solution really the same as the fp64 solver?

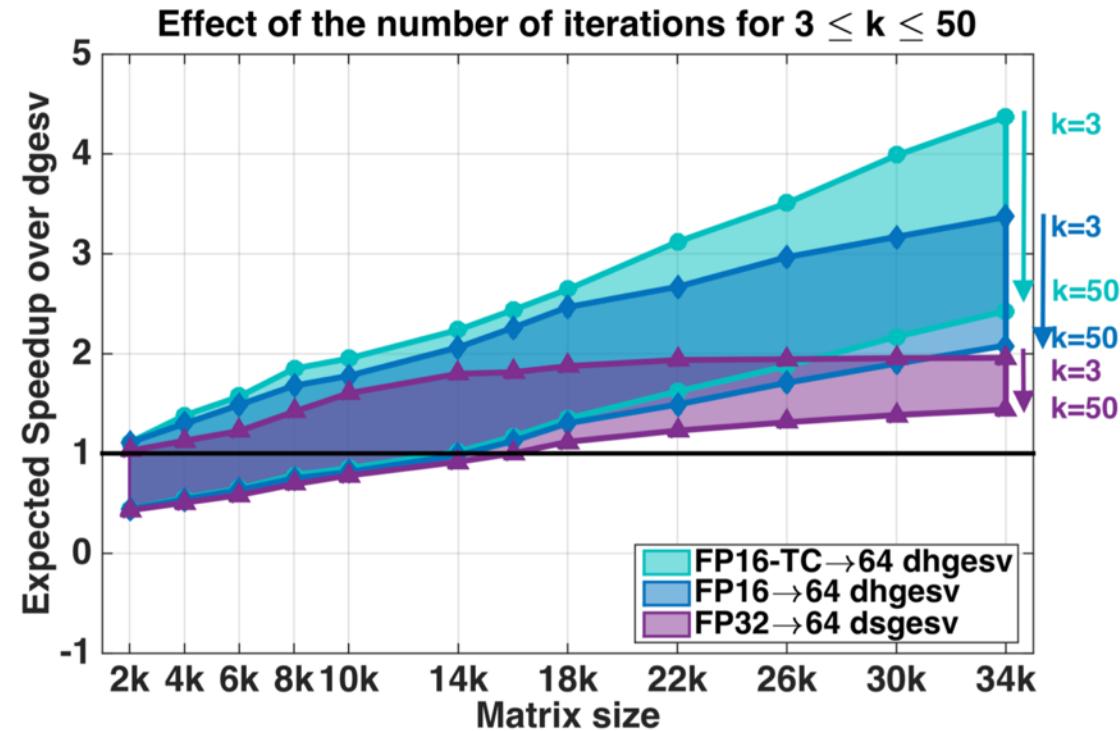
Iteration	Full FP64	FP32->FP64	FP16(TC)-FP64
1 $\frac{\ b - Ax\ }{\ b\ }$		3.17E-02	4.97E+00
2 $\ b - Ax\  / \ b\ $		2.66E-07	4.35E+00
3 $\ b - Ax\  / \ b\ $		1.17E-10	2.15E+00
4 $\ b - Ax\  / \ b\ $		3.24E-14	8.98E-01
5 $\ b - Ax\  / \ b\ $			2.23E-01
6 $\ b - Ax\  / \ b\ $			1.44E-01
7 $\ b - Ax\  / \ b\ $			9.37E-02
8 $\ b - Ax\  / \ b\ $			3.68E-02
9 $\ b - Ax\  / \ b\ $			1.12E-02
10 $\ b - Ax\  / \ b\ $			1.19E-03
11 $\ b - Ax\  / \ b\ $			1.09E-04
12 $\ b - Ax\  / \ b\ $			1.69E-05
Total iterations		4	18
Final solution $ b - Ax  /  A  n$	1.50e-16	1.20e-16	1.19e-16
Final solution $ b - Ax  /  A   x  n$	1.49e-21	1.19e-21	1.18e-21
Final solution $ b - Ax  / ( A   x  +  b )n$ $ b - Ax _{\infty}$ $( A  \cdot  x  +  b )n$	1.49e-21	1.19e-21	1.18e-21

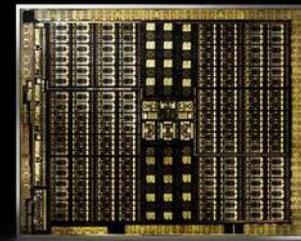
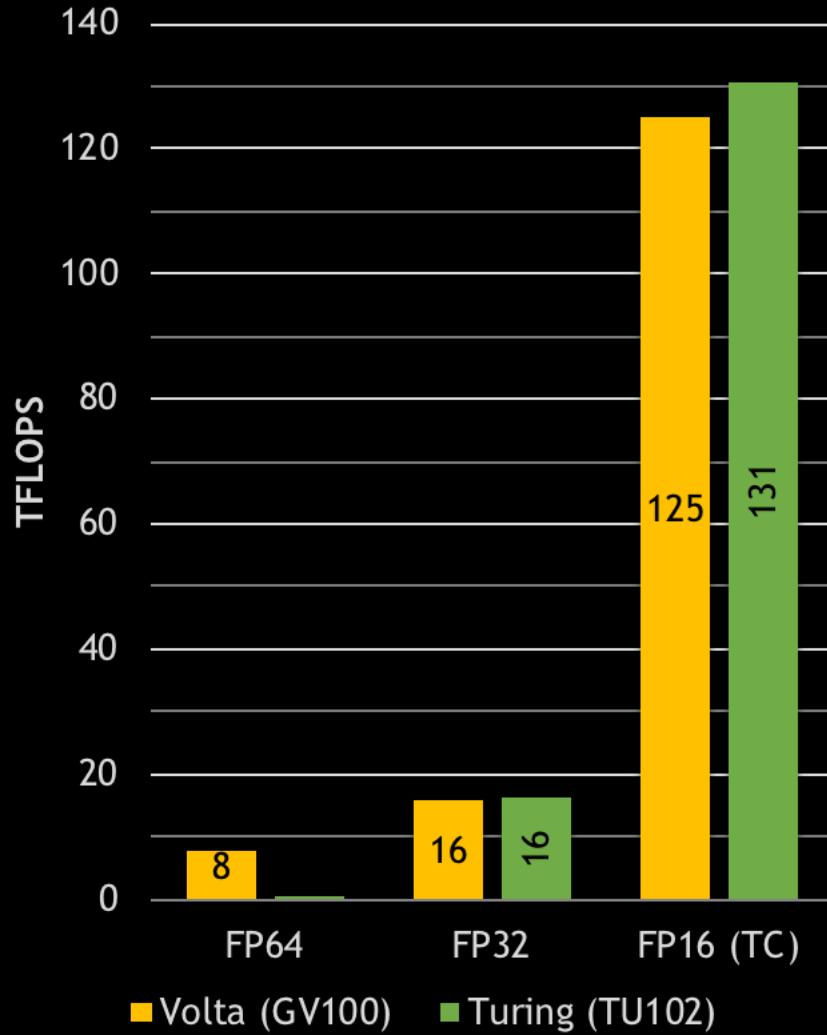
# Leveraging Tensor Cores

## Iterative Refinement Solver

$$\text{time for FP64} = \frac{2n^3}{3P_{dgetrf}} + \frac{2n^2}{P_{dtrsv}}$$

$$\text{time for MP} = \frac{2n^3}{3P_{Xgetrf}} + k \left( \frac{2n^2}{P_{dgemv}} + \frac{2n^2}{P_{Xtrsv}} + \xi \right)$$



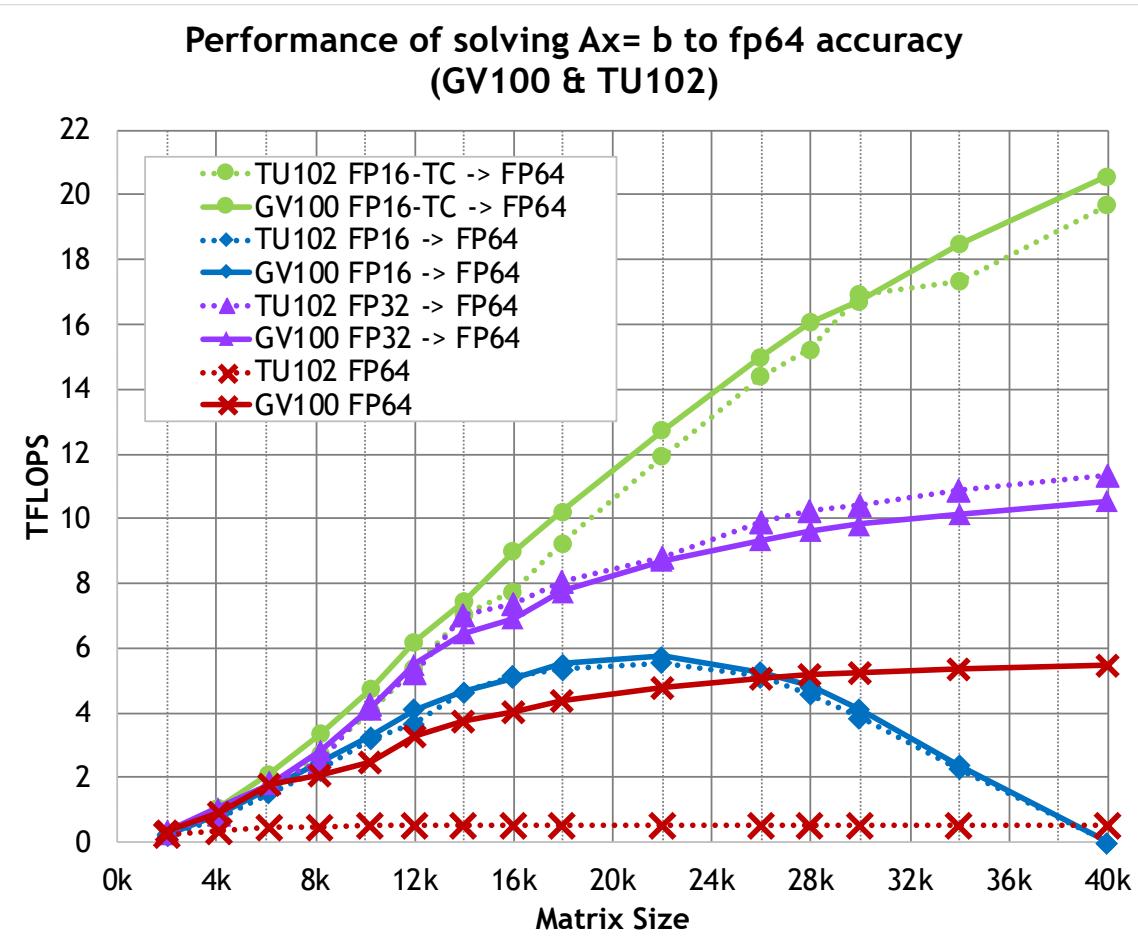


# Mixed-precision iterative refinement solver

## GV100 vs TU102

$$\text{time for FP64} = \frac{2n^3}{3P_{dgetrf}} + \frac{2n^2}{P_{dtrsv}}$$

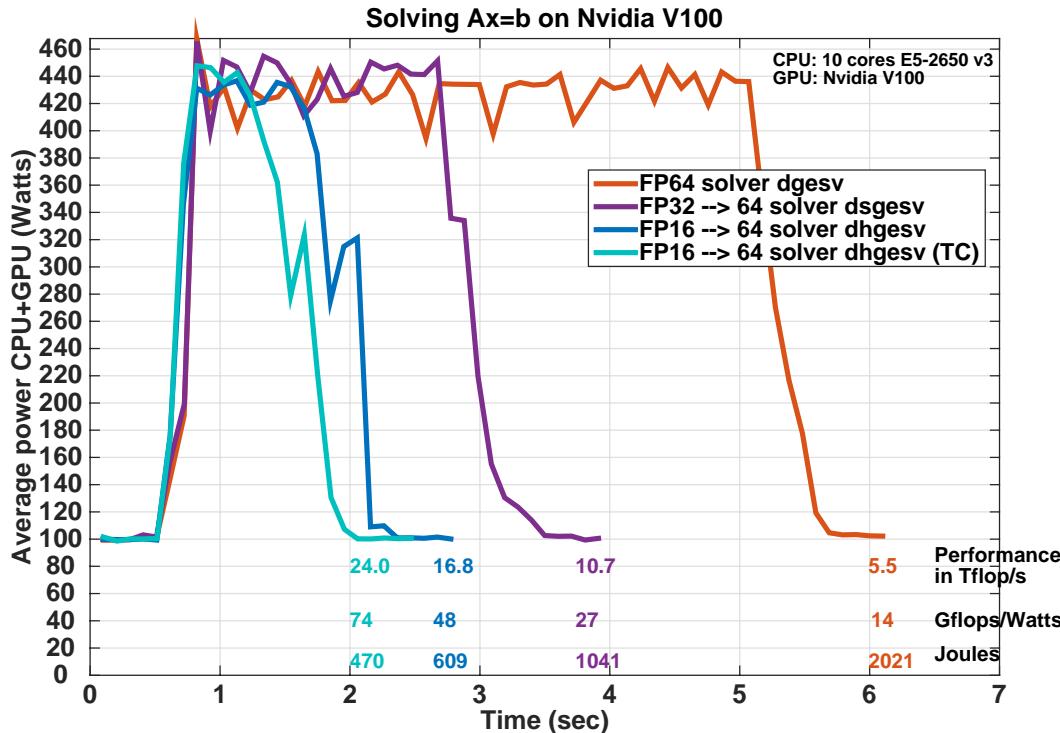
$$\text{time for MP} = \frac{2n^3}{3P_{Xgetrf}} + k \left( \frac{2n^2}{P_{dgemv}} + \frac{2n^2}{P_{Xtrsv}} + \xi \right)$$



Results obtained using MAGMA 2.5.0

# Mixed-precision iterative refinement solver

## Energy Efficiency



### Green500 List for November 2018

Listed below are the November 2018 The Green500's energy-efficient supercomputers ranked from 1 to 10.

Note: Shaded entries in the table below mean the power data is derived and not measured.

TOP500			Cores	Rmax (TFlop/s)	Power (kW)	Efficiency (GFlops/watts)
Rank	Rank	System				
1	375	Shoubu system B - ZettaScaler-2.2, Xeon D-1571 16C 1.3GHz, Infiniband EDR, PEZY-SC2 , PEZY Computing / Exascaler Inc. Advanced Center for Computing and Communication, RIKEN Japan	953,280	1,063.3	60	17.604
2	374	DGX SaturnV Volta - NVIDIA DGX-1 Volta36, Xeon E5-2698v4 20C 2.2GHz, Infiniband EDR, NVIDIA Tesla V100 , Nvidia NVIDIA Corporation United States	22,440	1,070.0	97	15.113
3	1	Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM DOE/SC/Oak Ridge National Laboratory United States	2,397,824	143,500.0	9,783	14.668

# TENSOR CORE ACCELERATED ITERATIVE REFINEMENT SOLVERS

## cuSOLVER productization plans

- Real & Planar Complex
- FP32 & FP64 support

LU Solver

~September 2019

Cholesky  
Solver

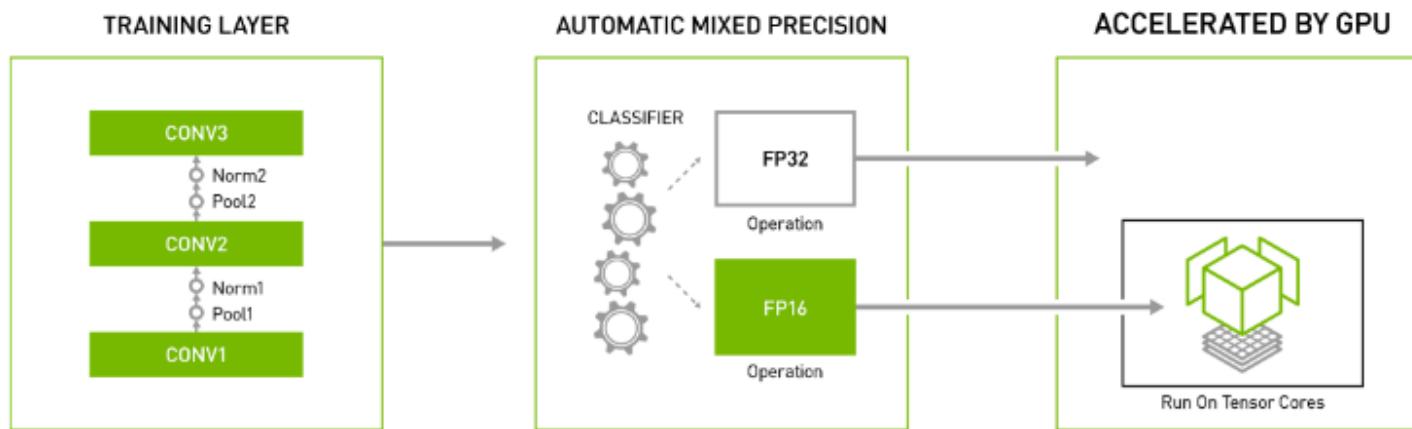
~November 2019

QR Solver

~November 2019

# AUTOMATIC MIXED PRECISION

Easy to Use, Greater Performance and Boost in Productivity



Insert ~ two lines of code to introduce Automatic Mixed-Precision and get upto 3X speedup

AMP uses a graph optimization technique to determine FP16 and FP32 operations

Support for TensorFlow, PyTorch and MXNet

Unleash the next generation AI performance and get faster to the market!

# ENABLING AUTOMATIC MIXED PRECISION

Add Just A Few Lines of Code, Get Up to 3X Speedup

TensorFlow  
W

```
os.environ['TF_ENABLE_AUTO_MIXED_PRECISION'] = '1'  
OR  
export TF_ENABLE_AUTO_MIXED_PRECISION=1  
  
Explicit optimizer wrapper available in NVIDIA Container 19.07+, TF 1.14+, TF  
2.0:  
  
opt = tf.train.experimental.enable_mixed_precision_graph_rewrite(opt)
```

GA

PyTorch

```
model, optimizer = amp.initialize(model, optimizer, opt_level="O1")  
with amp.scale_loss(loss, optimizer) as scaled_loss:  
    scaled_loss.backward()
```

GA

MXNet

```
amp.init()  
amp.init_trainer(trainer)  
with amp.scale_loss(loss, trainer) as scaled_loss:  
    autograd.backward(scaled_loss)
```

GA  
Coming  
Soon

More details: <https://developer.nvidia.com/automatic-mixed-precision>

# ENABLING AUTOMATIC MIXED PRECISION

Add Just A Few Lines of Code

- TensorFlow
  - NVIDIA container 19.03+:
    - `export TF_ENABLE_AUTO_MIXED_PRECISION=1` [automatic casting and automatic loss scaling]
  - Available in NVIDIA container 19.07+, TF 1.14+, TF 2.0:
    - We provide an explicit optimizer wrapper to perform loss scaling - which can also enable auto-casting for you:

```
import tensorflow as tf
```

```
opt = tf.train.GradientDescentOptimizer(0.5)
```

```
opt = tf.train.experimental.enable_mixed_precision_graph_rewrite(opt)
```

# ENABLING AUTOMATIC MIXED PRECISION

Add Just A Few Lines of Code

- PyTorch

- Two steps: initialization and wrapping backpropagation

```
from apex import amp
model = ...
optimizer = SomeOptimizer(model.parameters(), ...)
# ...
model, optimizer = amp.initialize(model, optimizer, opt_level="O1")
# ...
for train_loop():
    loss = loss_fn(model(x), y)
    with amp.scale_loss(loss, optimizer) as scaled_loss:
        scaled_loss.backward()
    # Can manipulate the .grads if you'd like
    optimizer.step()
```

# ENABLING AUTOMATIC MIXED PRECISION

Add Just A Few Lines of Code

- MXNET

- NVIDIA container 19.03+ and MXNET 1.5:

```
from mxnet.contrib import amp
amp.init()
net = get_network()
trainer = mx.gluon.Trainer(...)
amp.init_trainer(trainer)
for data in dataloader:
    with autograd.record(True):
        out = net(data)
        l = loss(out)
        with amp.scale_loss(l, trainer) as scaled_loss:
            autograd.backward(scaled_loss)
    trainer.step()
```

# AUTOMATIC MIXED PRECISION IN TENSORFLOW

Upto 3X Speedup



TensorFlow Medium Post: [Automatic Mixed Precision in TensorFlow for Faster AI Training on NVIDIA GPUs](#)

All models can be found at:

<https://github.com/NVIDIA/DeepLearningExamples/tree/master/TensorFlow>, except for ssd-rn50-fpn-640, which is here: [https://github.com/tensorflow/models/tree/master/research/object\\_detection](https://github.com/tensorflow/models/tree/master/research/object_detection)

All performance collected on 1xV100-16GB, except bert-squadqa on 1xV100-32GB.

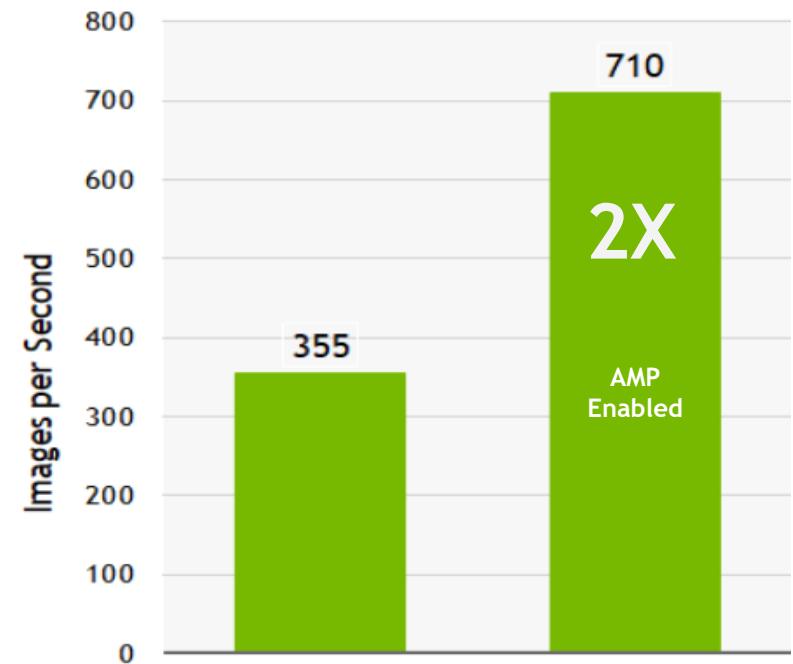
Speedup is the ratio of time to train for a fixed number of epochs in single-precision and Automatic Mixed Precision. Number of epochs for each model was matching the literature or common practice (it was also confirmed that both training sessions achieved the same model accuracy).

Batch sizes: rn50 (v1.5): 128 for FP32, 256 for AMP+XLA; ssd-rn50-fpn-640: 8 for FP32, 16 for AMP+XLA; NCF: 1M for FP32 and AMP+XLA; bert-squadqa: 4 for FP32, 10 for AMP+XLA; GNMT: 128 for FP32, 192 for AMP.

# AUTOMATIC MIXED PRECISION IN PYTORCH

<https://developer.nvidia.com/automatic-mixed-precision>

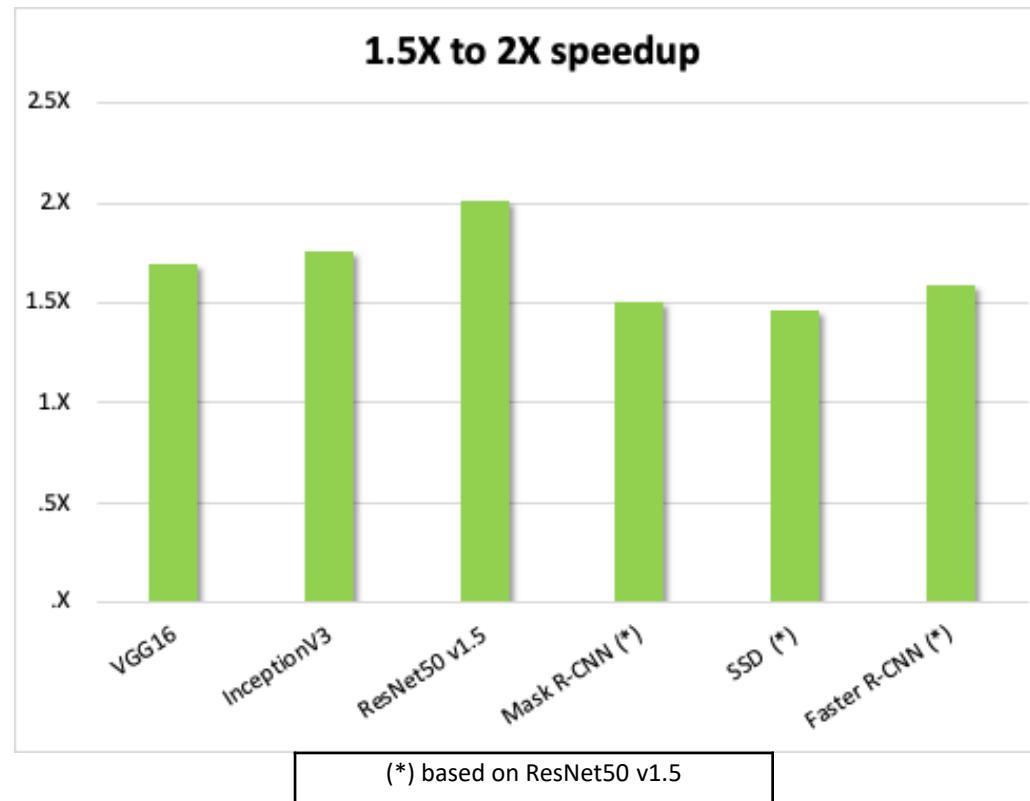
- Plot shows ResNet-50 result with/without automatic mixed precision(AMP)
- More AMP enabled model scripts coming soon:  
Mask-R CNN, GNMT, NCF, etc.



<https://github.com/NVIDIA/apex/tree/master/examples/imagenet>

# AUTOMATIC MIXED PRECISION IN MXNET

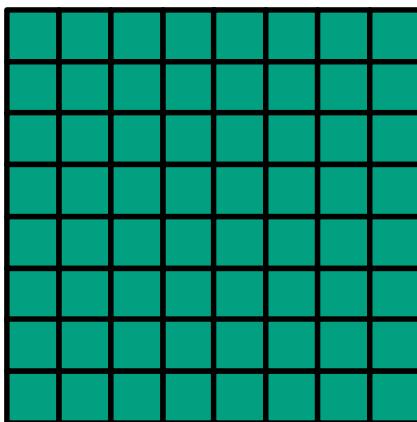
AMP speedup ~1.5X to 2X in comparison with FP32



<https://github.com/apache/incubator-mxnet/pull/14173>

# CUDA TENSOR CORE PROGRAMMING

## WMMA datatypes



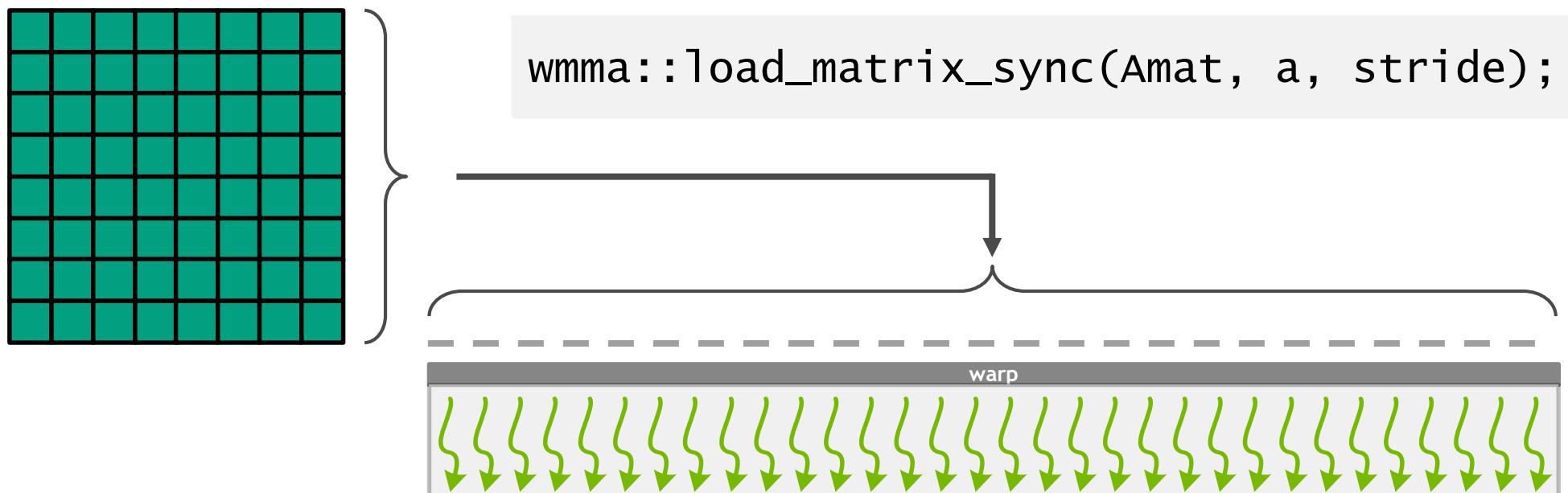
Per-Thread fragments to hold components of matrices for use with Tensor Cores

```
wmma::fragment<matrix_a, ...> Amat;
```

# CUDA TENSOR CORE PROGRAMMING

## WMMA load and store operations

Warp-level operation to fetch components of matrices into fragments



# CUDA TENSOR CORE PROGRAMMING

## WMMA Matrix Multiply and Accumulate Operation

Warp-level operation to perform matrix multiply and accumulate

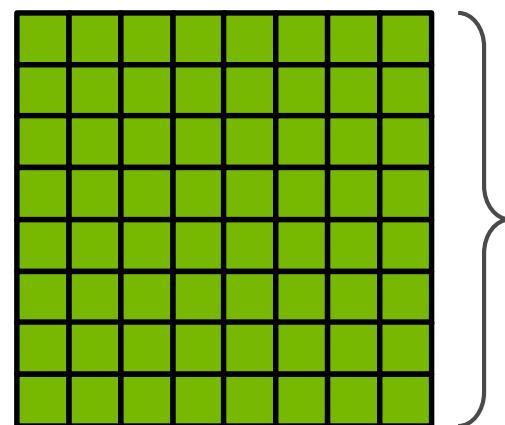
```
wmma::mma_sync(Dmat, Amat, Bmat, Cmat);
```

$$D = \left( \begin{array}{|c|c|c|c|c|} \hline & & & & \\ \hline \end{array} \right) \left( \begin{array}{|c|c|c|c|c|} \hline & & & & \\ \hline \end{array} \right) + \left( \begin{array}{|c|c|c|c|c|} \hline & & & & \\ \hline \end{array} \right)$$

# CUDA TENSOR CORE PROGRAMMING

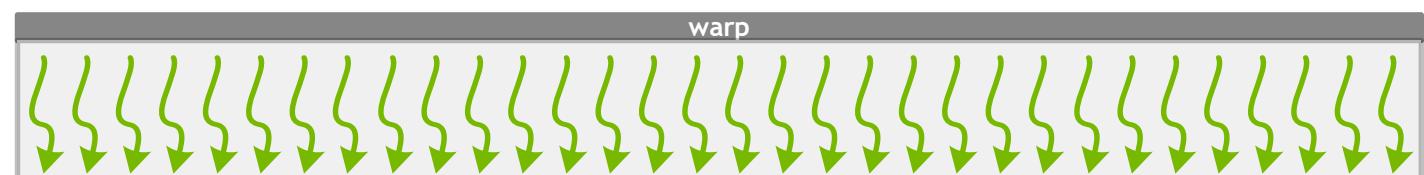
## WMMA load and store operations

Warp-level operation to fetch components of matrices into fragments



**Result**

```
wmma::store_matrix_sync(d, Dmat, stride);
```



# TENSOR CORE EXAMPLE

Create Fragments

Initialize Fragments

Perform MatMul

Store Results

```
__device__ void tensor_op_16_16_16(
    float *d, half *a, half *b, float *c)
{
    wmma::fragment<matrix_a, ...> Amat;
    wmma::fragment<matrix_b, ...> Bmat;
    wmma::fragment<matrix_c, ...> Cmat;

    wmma::load_matrix_sync(Amat, a, 16);
    wmma::load_matrix_sync(Bmat, b, 16);
    wmma::fill_fragment(Cmat, 0.0f);

    wmma::mma_sync(Cmat, Amat, Bmat, Cmat);

    wmma::store_matrix_sync(d, Cmat, 16,
                           wmma::row_major);
}
```

CUDA C++

Warp-Level Matrix Operations

# TENSOR CORES IN CUDA FORTRAN

Similar to CUDA C WMMA API, with some name changes

real(2) support for half-precision data available (on both host and device) in PGI 19.7 compilers

Requires `wmma` Fortran module and macros in `cuf_macros.CUF` file

# CUDA FORTRAN TENSOR CORE EXAMPLE

## Device Code

```
#include "cuf_macros.CUF"
module m
contains
    attributes(global) subroutine wmma_16x16(a, b, c)
        use wmma
        real(2), intent(in) :: a(16,*), b(16,*)
        real(4) :: c(16,*)
        WMMASubMatrix(WMMAMatrixA, 16, 16, 16, Real, WMMAColMajor) :: sa
        WMMASubMatrix(WMMAMatrixB, 16, 16, 16, Real, WMMAColMajor) :: sb
        WMMASubMatrix(WMMAMatrixC, 16, 16, 16, Real, WMMAKind4) :: sc
        sc = 0.0_4
        call wmmaLoadMatrix(sa, a(1,1), 16)
        call wmmaLoadMatrix(sb, b(1,1), 16)
        call wmmaMatMul(sc, sa, sb, sc)
        call wmmaStoreMatrix(c(1,1), sc, 16)
    end subroutine wmma_16x16
end module m
```

WMMA Definitions

WMMA “fragments”

Assignment overloaded to call fill\_fragment()

# CUDA FORTRAN TENSOR CORE EXAMPLE

## Host Code

Launch with a single warp of threads

```
program main
    use m
    use cudafor
    integer, parameter :: m = 16, n=m, k=m
    real(4) :: a(m,k), b(k,n), c(m,n), cref(m,n)
    real(4), device :: c_d(m,n)
    real(2), device :: ah_d(m,k), bh_d(k,n)

    call random_number(a); a = int(4.*a); ah_d = a
    call random_number(b); b = int(4.*b); bh_d = b

    cref = matmul(a, b)
    c = 0.0
    call wmma_16x16<<<1,32>>>(ah_d, bh_d, c_d)
    c = c_d

    if (sum(abs(c-cref)) == 0.0) write(*,*) 'Test passed'
end program main
```

Host-device transfer and 4- to 2-byte conversion



MMA SYNC

# VOLTA MMA SYNC

Warp-scoped matrix multiply instruction

mma.sync: new instruction in CUDA 10.1

- Directly targets Volta Tensor Cores

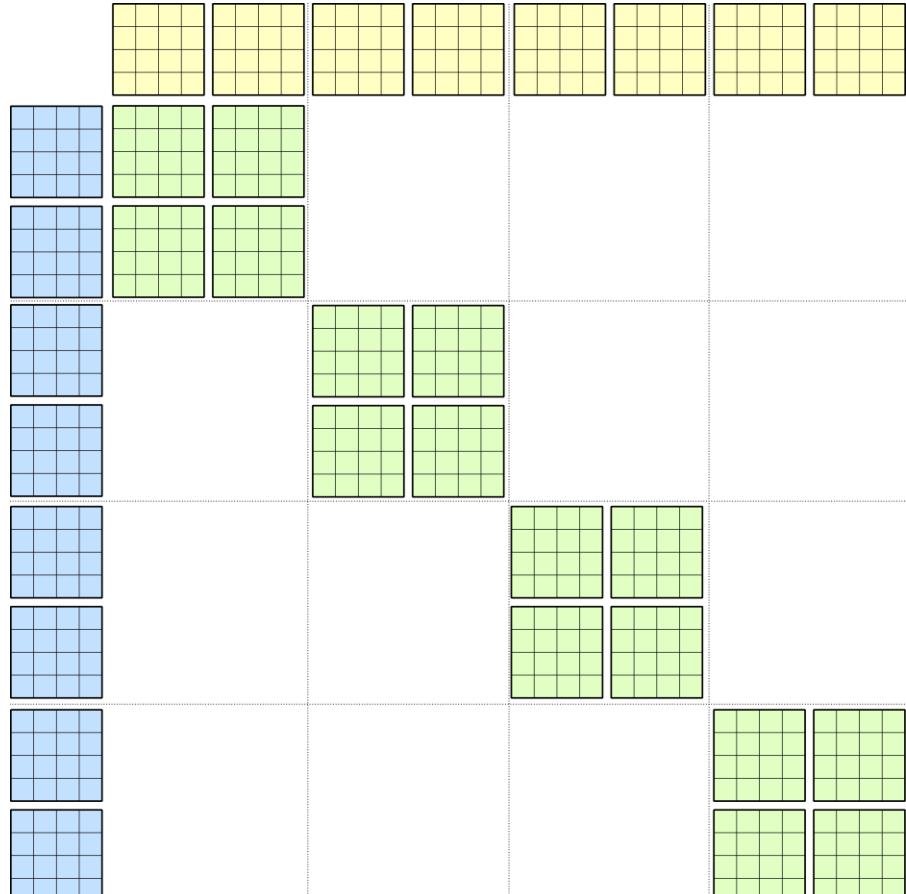
Matrix multiply-accumulate

$$D = A * B + C$$

- A, B: half
- C, D: float or half

Warp-synchronous:

- Four independent **8-by-8-by-4** matrix multiply-accumulate operations

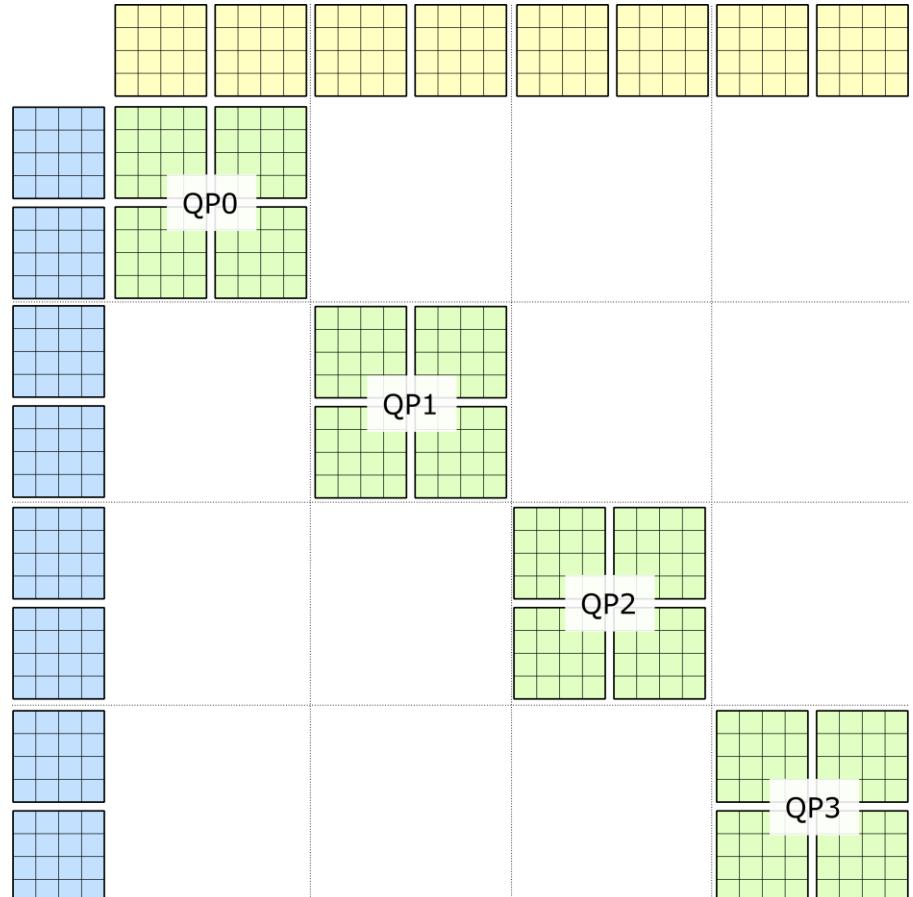


# VOLTA MMA SYNC

Warp-scoped matrix multiply instruction

Warp is partitioned into Quad Pairs

- QP0: T0..T3      T16..T19
- QP1: T4..T7      T20..T23
- QP2: T8..T11      T24..T27
- QP3: T12..T15      T28..T31  
(eight threads each)

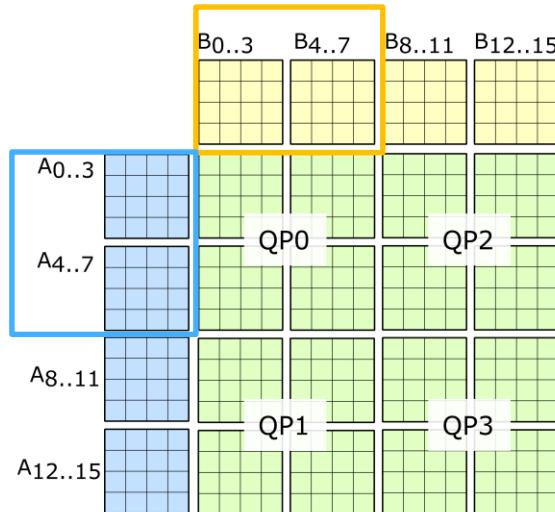


Each Quad Pair performs one **8-by-8-by-4** matrix multiply

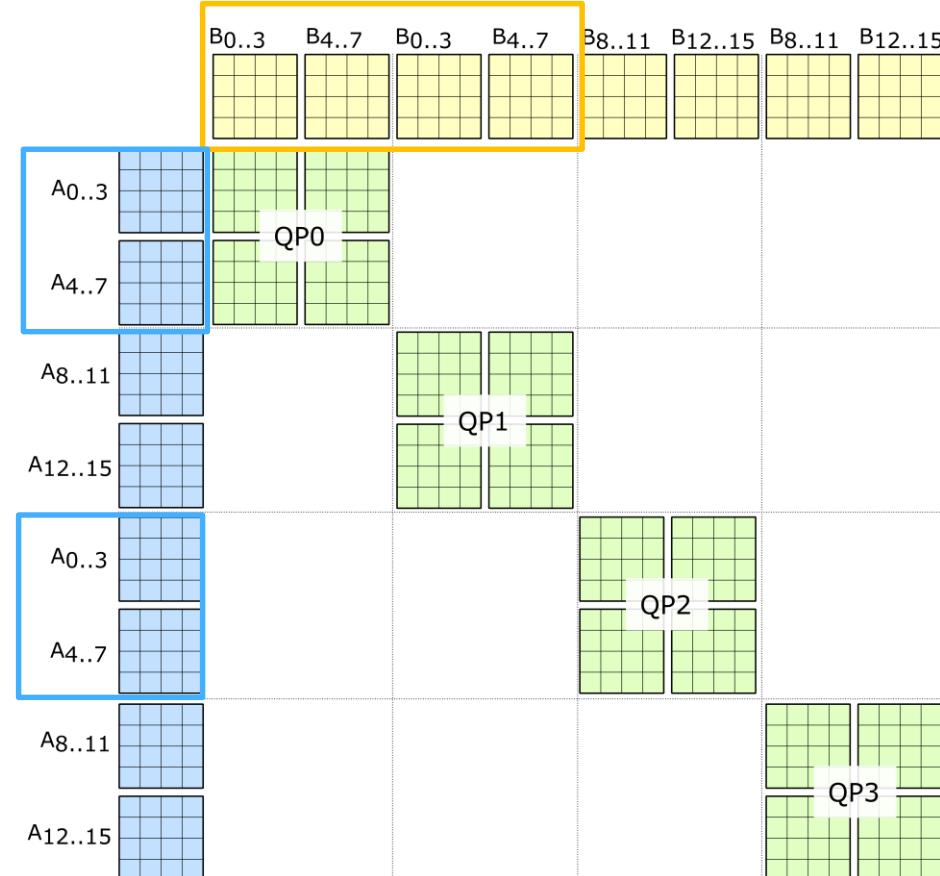
# COMPOSING MATRIX MULTIPLIES

Replicate data to compute warp-wide 16-by-16-by-4 matrix product

- $A_{0..7}$ : QP0, QP2     $A_{8..15}$ : QP1, QP3
- $B_{0..7}$ : QP0, QP1     $B_{8..15}$ : QP2, QP3



1 x mma.sync: 16-by-16-by-4



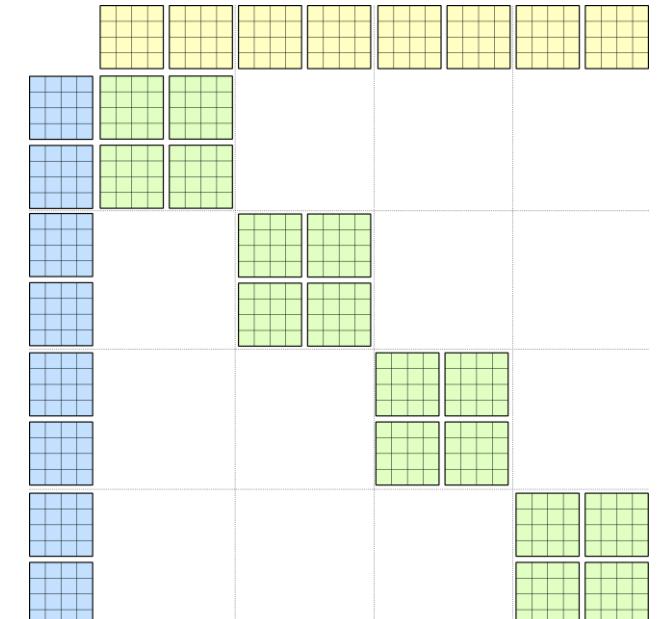
# VOLTA MMA SYNC      $D = A * B + C$

## PTX Syntax

```
mma.sync.aligned.m8n8k4.alayout.blayout.dtype.f16.f16.ctype    d, a, b, c;
```

```
.alayout = { .row, .col};  
.blayout = { .row, .col};  
.ctype   = { .f16, .f32};  
.dtype   = { .f16, .f32};
```

d: 8 x .dtype  
a: 4 x .f16  
b: 4 x .f16  
c: 8 x .ctype



Note: .f16 elements must be packed into .f16x2

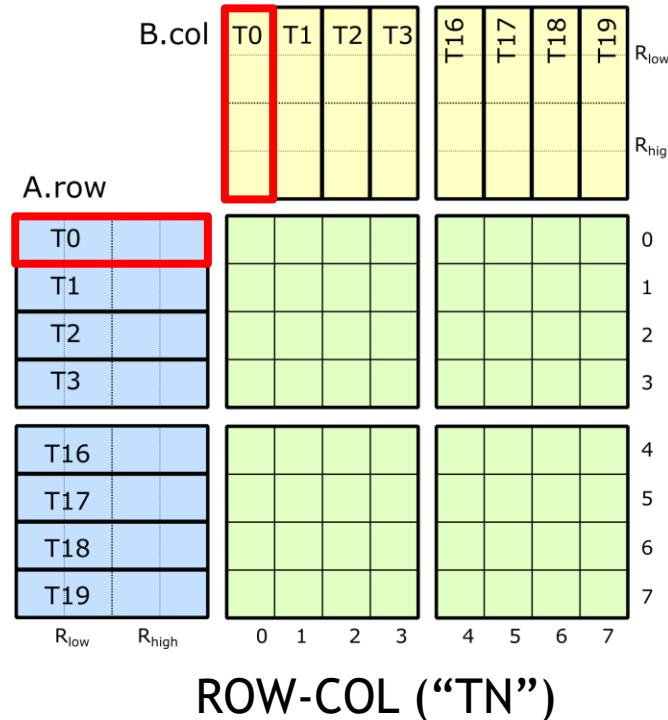
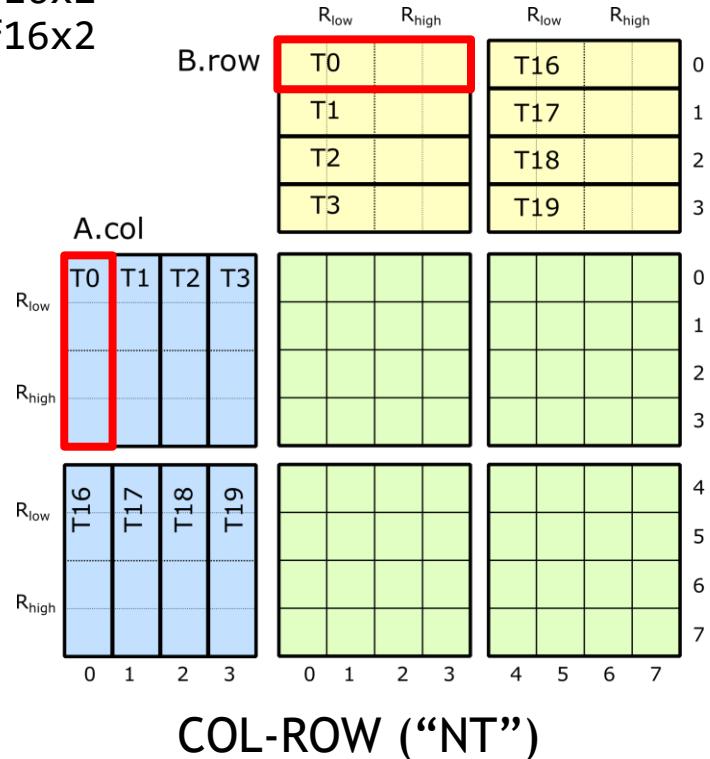
# THREAD-DATA MAPPING - F16 MULTIPLICANDS

Distributed among threads in quad pair (QP0 shown)

```
mma.sync.aligned.m8n8k4.alayout.blayout.dtype.f16.f16.ctype d, a, b, c;
```

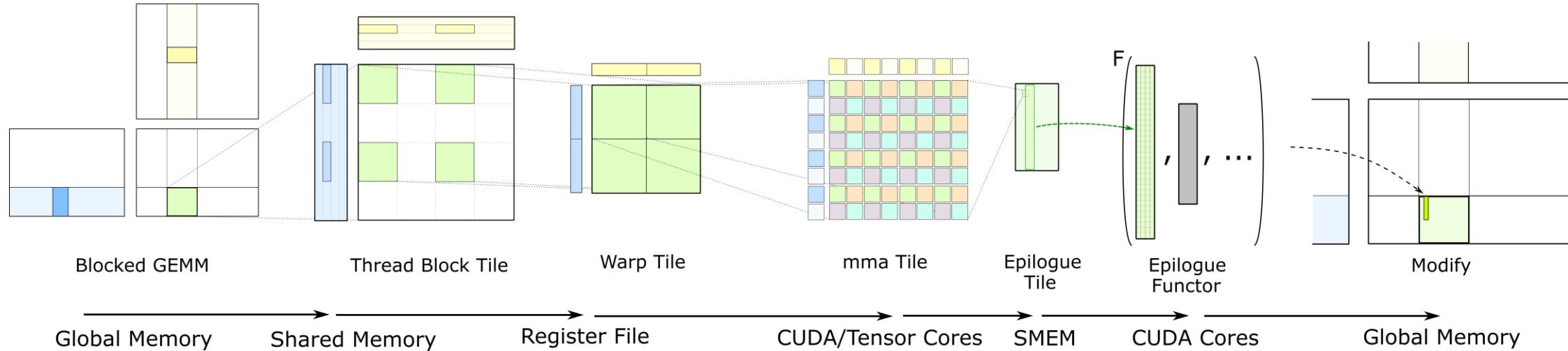
```
.alayout = { .row, .col };
.blayout = { .row, .col };
```

**a:** 2 x .f16x2  
**b:** 2 x .f16x2



# CUTLASS

CUDA C++ Template Library for Matrix Algebra



## CUTLASS template library for GEMM computations

- Blocked structure to maximize data reuse
- Software pipelined to hide latency
- Conflict-free Shared Memory access to maximize data throughput

See [CUTLASS GTC 2018](#) talk.  
69

# CUTLASS DESIGN PATTERNS

## Design patterns and template concepts in CUTLASS

**Templates:** generic programming and compile-time optimizations

**Traits:** describes properties, types, and functors used to specialize CUTLASS concepts

**Params:** structure containing parameters and precomputed values; passed to kernel as POD

**Vectorized Memory Accesses:** load and store as 32b, 64b, or 128b vectors

**Shape<>:** describes size of a 4D vector quantity

**TileTraits<>:** describes a 4D block of elements in memory

**Fragment<>:** partitioning of a tile across a collection of threads

**TileIterator<>:** loads a tile by a collection of threads; result is held in Fragment

# GEMM TEMPLATE KERNEL

CUTLASS provides building blocks for efficient device-side code

- Helpers simplify common cases

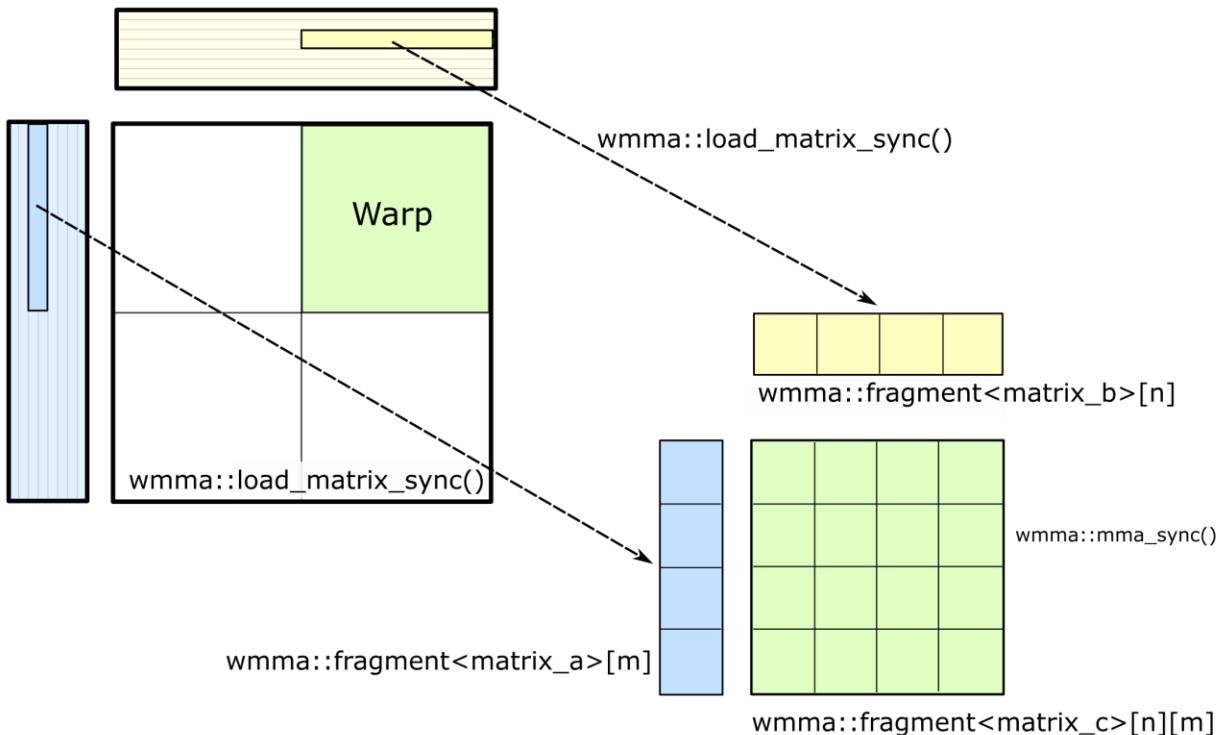
```
//  
// Specialization for single-precision  
//  
typedef cutlass::gemm::SgemmTraits<  
    cutlass::MatrixLayout::kColumnMajor,  
    cutlass::MatrixLayout::kRowMajor,  
    cutlass::Shape<8, 128, 128>  
> SgemmTraits;  
  
// Simplified kernel launch  
Gemm<SgemmTraits>::launch(params);  
  
//  
// CUTLASS GEMM kernel  
//  
template <typename Gemm>  
__global__ void gemm_kernel(typename Gemm::Params params) {  
  
    // Declare shared memory  
    __shared__ typename Gemm::SharedStorage shared_storage;  
  
    // Construct the GEMM object with cleared accumulators  
    Gemm gemm(params);  
  
    // Compute the matrix multiply-accumulate  
    gemm.multiply_add(shared_storage.mainloop);  
  
    // Update output memory efficiently  
    gemm.update(shared_storage.epilogue);  
}
```

# EXAMPLE: VOLTA TENSOR CORES

## Targeting the CUDA WMMA API

WMMA: Warp-synchronous Matrix Multiply-Accumulate

- API for issuing operations to Volta Tensor Cores



```
/// Perform warp-level multiply-accumulate using WMMA API
template <
    /// Data type of accumulator
    typename ScalarC,
    /// Shape of warp-level accumulator tile
    typename WarpTile,
    /// Shape of one WMMA operation - e.g. 16x16x16
    typename WmmaTile
>
struct WmmaMultiplyAdd {

    /// Compute number of WMMA operations
    typedef typename ShapeDiv<WarpTile, WmmaTile>::Shape
        Shape;

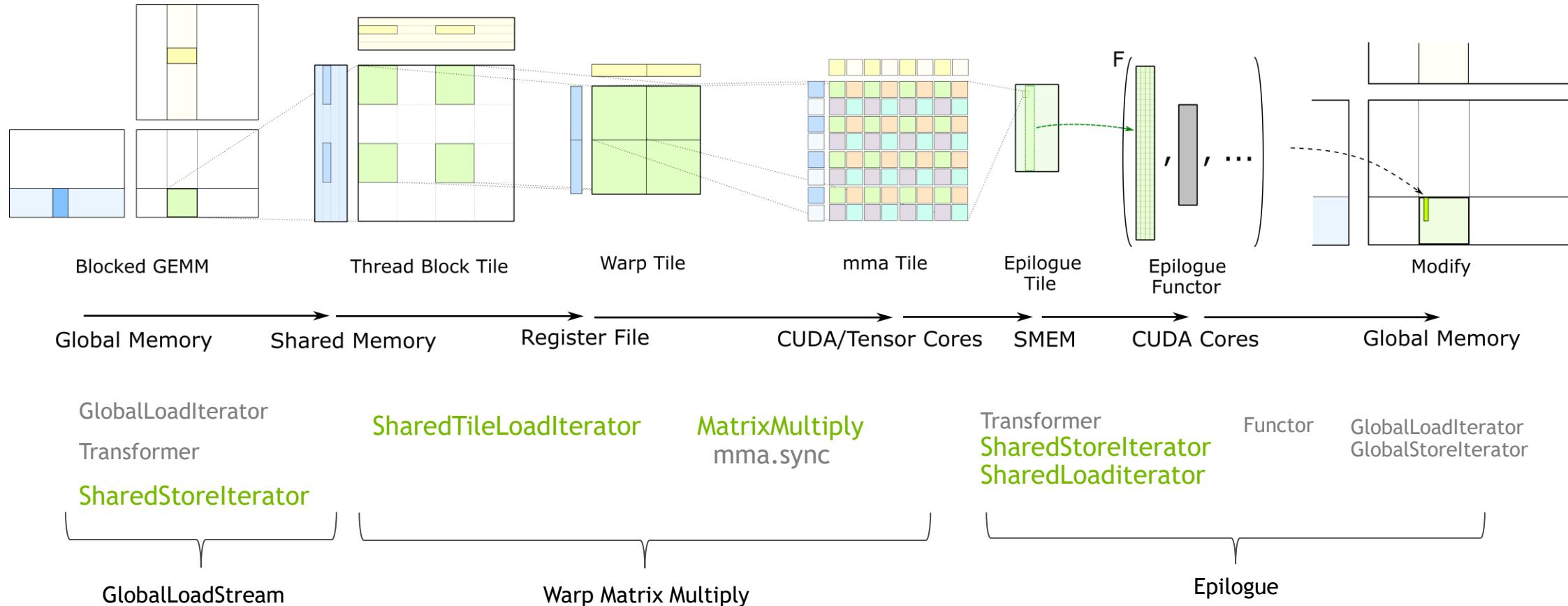
    /// Multiply: D = A*B + C
    inline __device__ void multiply_add(
        FragmentA const & A,
        FragmentB const & B,
        FragmentC const & C,
        FragmentD & D) {

        // Perform M-by-N-by-K matrix product using WMMA
        for (int n = 0; n < Shape::kH; ++n) {
            for (int m = 0; m < Shape::kW; ++m) {

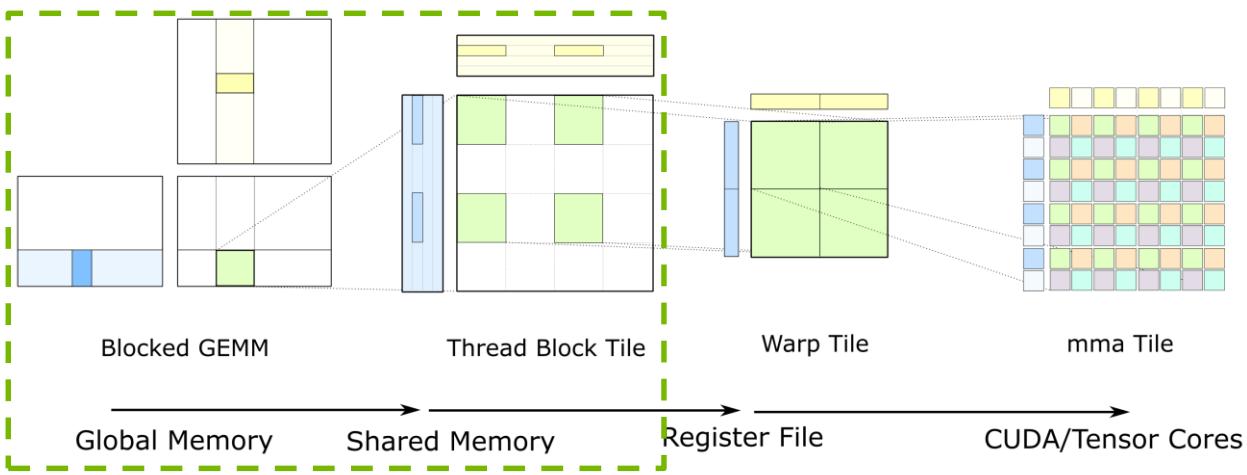
                // WMMA API to invoke Tensor Cores
                nvcuda::wmma::mma_sync(
                    D.elements[n][m],
                    A.elements[k][m],
                    B.elements[k][n],
                    C.elements[n][m]
                );
            }
        }
    };
}
```

# CUTLASS 1.3

Reusable components targeting Volta Tensor Cores



# STORING TO SHARED MEMORY



CUTLASS Tile Iterators to transform:

- Global Memory: Canonical matrix layout → Shared Memory: permuted shared memory layout

cutlass/gemm/volta884\_multiplicand.h

```
// Defines iterators for loading and storing multiplicands
template <
    /// Identifies multiplicand of GEMM (A or B)
    GemmOperand::Kind Operand,
    /// Specifies layout of data in source memory
    MatrixLayout::Kind Layout,
    /// Specifies threadblock tile shape
    typename Tile,
    /// Specifies warp tile shape
    typename WarpTile,
    /// Specifies the number of participating warps
    int WarpCount,
    /// Specifies the delta between warp tiles
    typename WarpDelta
>
struct Volta884Multiplicand {
    // Thread-block load iterator (canonical matrix layout)
    // ...
    typedef ... LoadIterator;

    // Thread-block store iterator (permuted SMEM layout)
    // ...
    typedef ... StoreIterator;

    // Warp-level load iterator
    // ...
    typedef ... WarpLoadIterator;
};
```

# LOADING FROM SHARED MEMORY

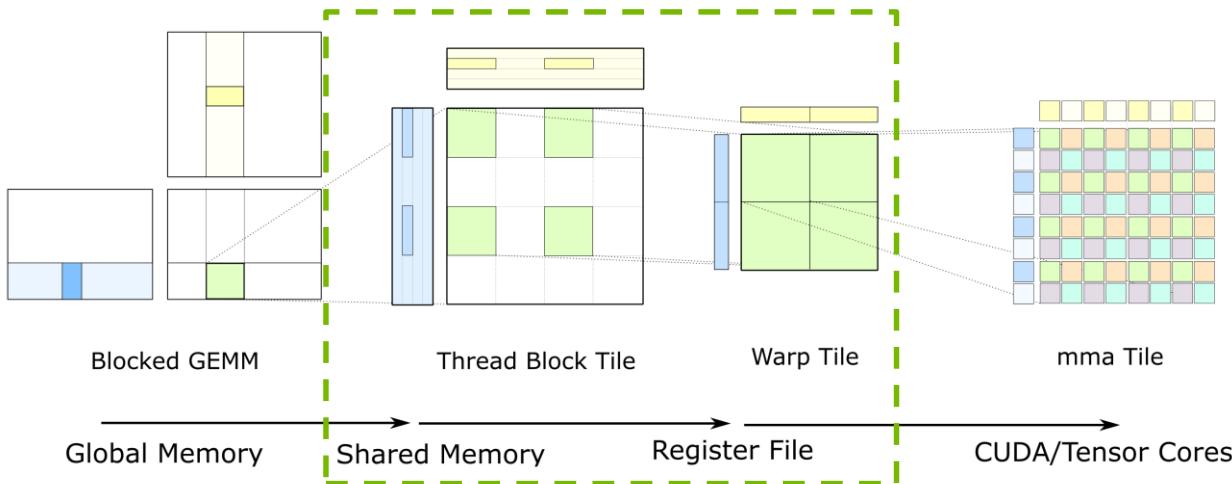
cutlass/gemm/volta884\_multiplicand.h

```
// Defines iterators for loading and storing multiplicands
template <
    /// Identifies multiplicand of GEMM (A or B)
    GemmOperand::Kind Operand,
    /// Specifies layout of data in source memory
    MatrixLayout::Kind Layout,
    /// Specifies threadblock tile shape
    typename Tile,
    /// Specifies warp tile shape
    typename WarpTile,
    /// Specifies the number of participating warps
    int WarpCount,
    /// Specifies the delta between warp tiles
    typename WarpDelta
>
struct Volta884Multiplicand {

    // Thread-block load iterator (canonical matrix layout)
    // ...
    typedef ... LoadIterator;

    // Thread-block store iterator (permuted SMEM layout)
    // ...
    typedef ... StoreIterator;

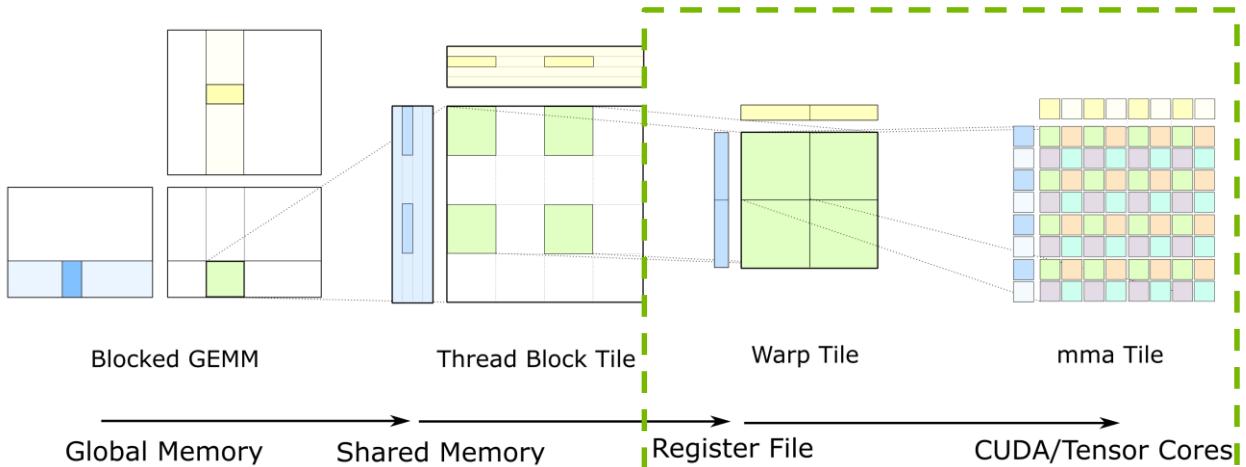
    // Warp-level load iterator
    // ...
    typedef ... WarpLoadIterator;
};
```



CUTLASS Tile Iterators to transform:

- Shared Memory: permuted shared memory layout → Register File: mma.sync thread-data mapping

# EXECUTING MMA SYNC



## CUTLASS Warp-scoped matrix multiply

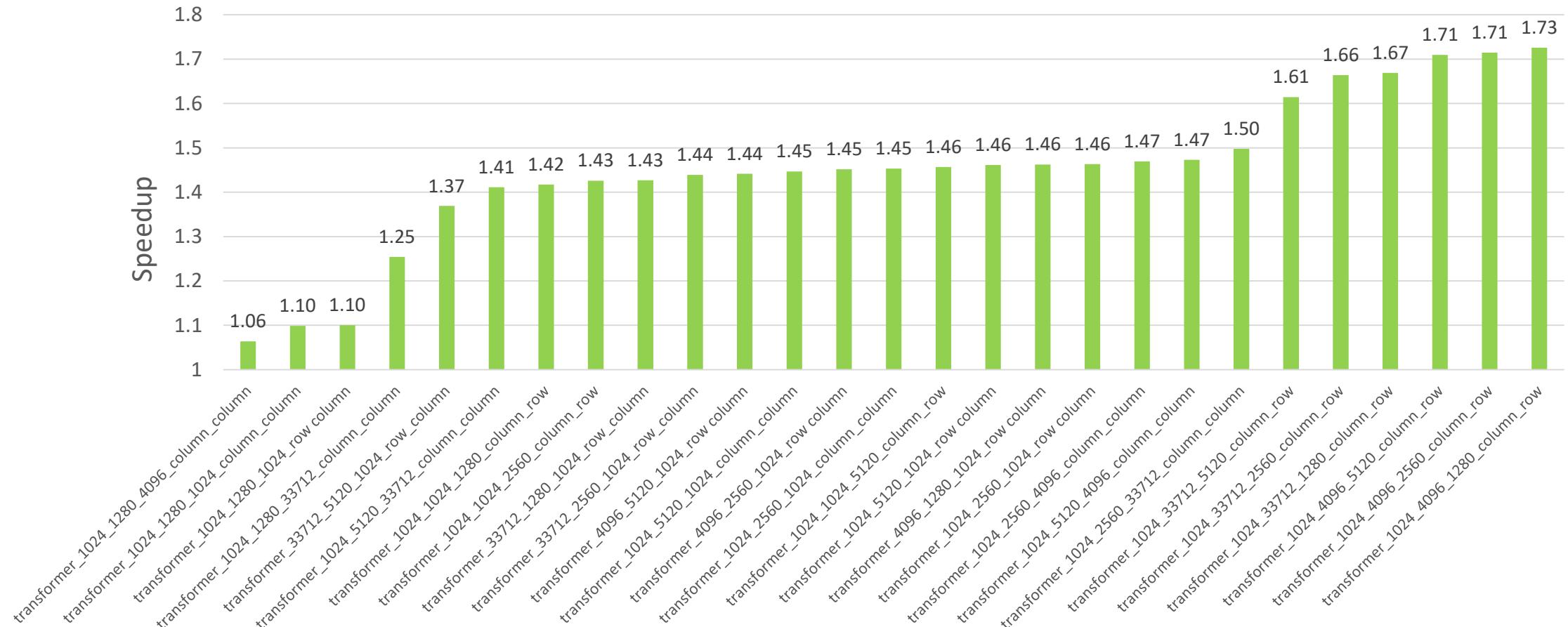
- Register File: mma.sync thread-data mapping → Tensor Cores: mma.sync

cutlass/gemm/volta884\_multiply\_add.h

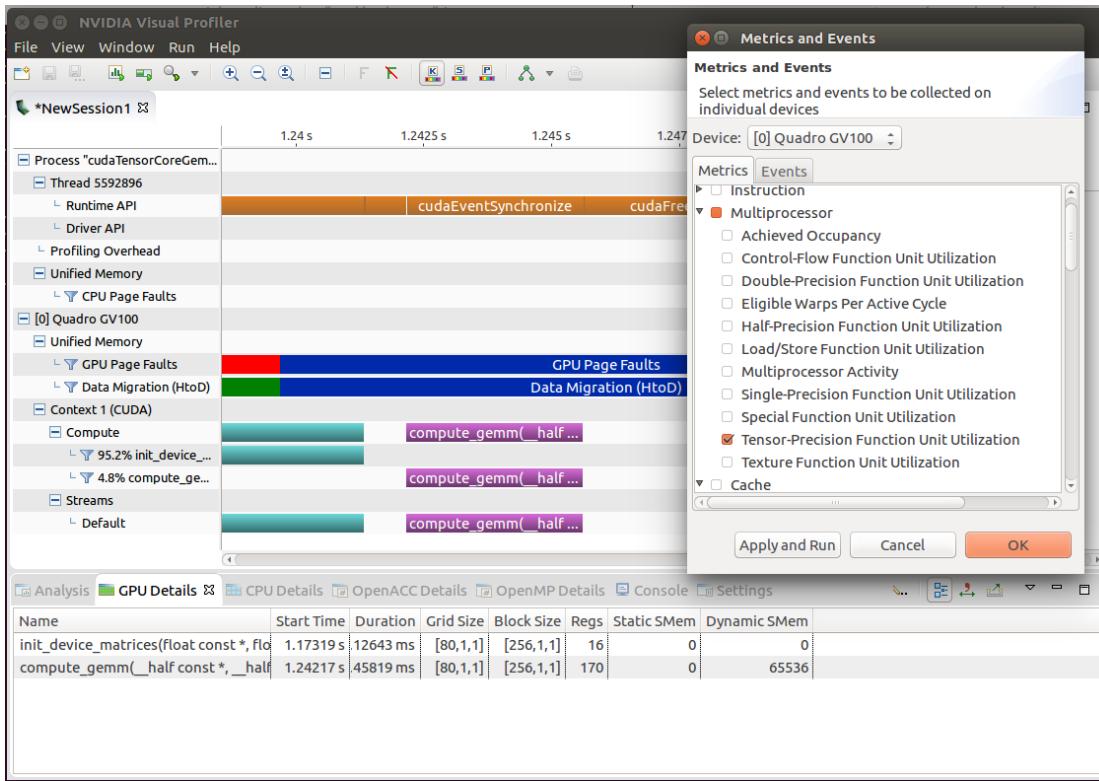
```
template <
    /// Shape of a warp-level GEMM (K-by-N-by-M)
    typename WarpGemmShape_,
    /// Layout of A multiplicand
    MatrixLayout::Kind LayoutA,
    /// Data type of A multiplicand
    typename ScalarA,
    /// Layout of B multiplicand
    MatrixLayout::Kind LayoutB,
    /// Data type of A multiplicand
    typename ScalarB,
    /// Data type of accumulators
    typename ScalarC,
    /// Whether infinite results are saturated to +-MAX_FLOAT
    bool SatFinite = false
>
struct Volta884MultiplyAdd {
    // ...
    // Multiply : d = (-)a*b + c.
    //
    CUTLASS_DEVICE
    void multiply_add(
        FragmentA const& A,
        FragmentB const& B,
        Accumulators const& C,
        Accumulators& D,
        bool negate = false) {
        ...
    }
};
```

# SPEEDUP RELATIVE TO WMMA

Transformer - CUTLASS 1.3 - mma.sync speedup vs WMMA  
V100 - CUDA 10.1

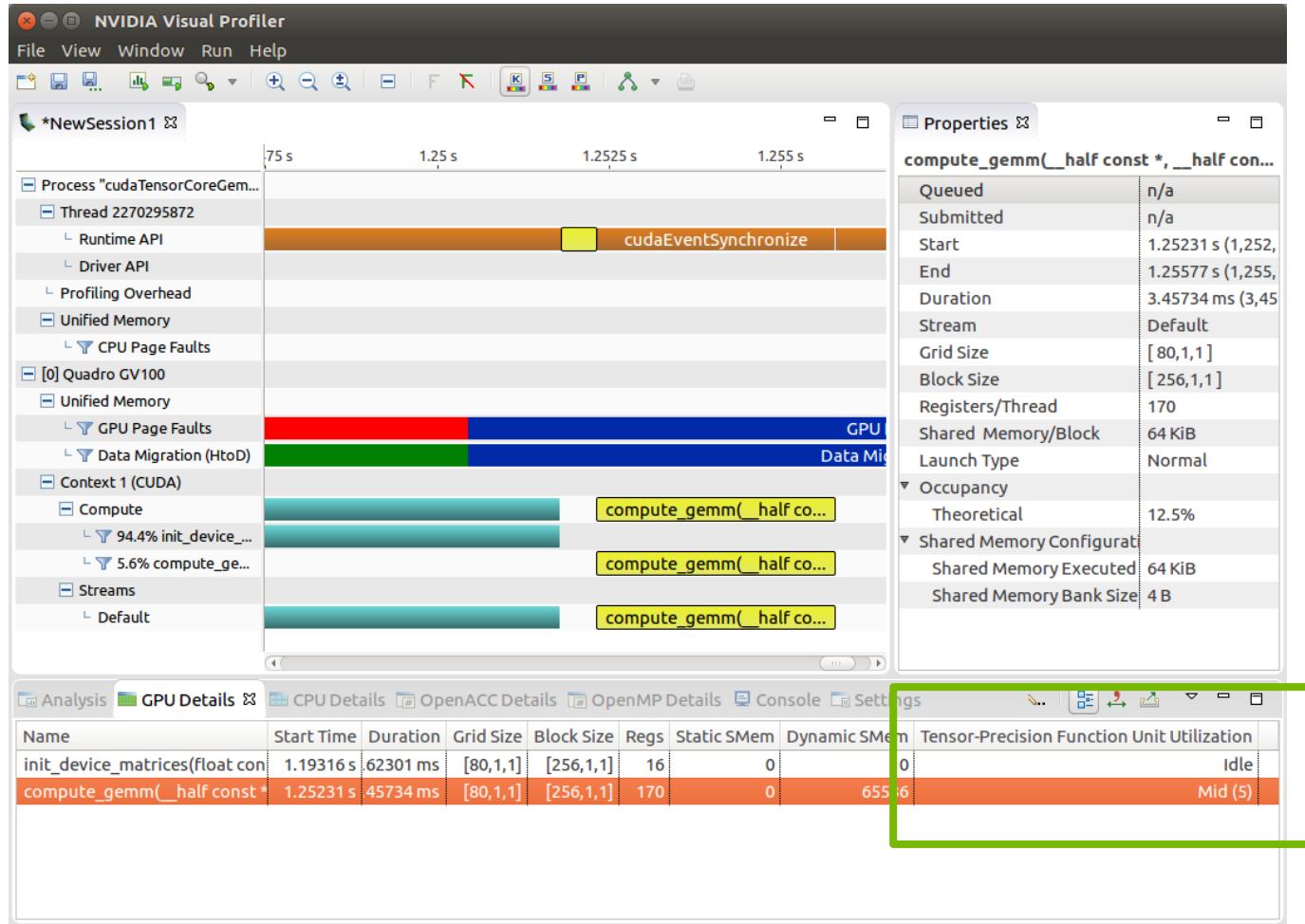


# TENSOR CORES WITH VISUAL PROFILER



- Visual Profiler allows gathering of Tensor Core Utilization after gathering a timeline.
- Use the menu option “Run->Collect Metrics and Events” to select the “Tensor-Precision Function Unit Utilization” metric under “Metrics->Multiprocessor”

# TENSOR CORES WITH VISUAL PROFILER



After clicking on the kernel of interest, select “GPU Details”

# TENSOR CORES WITH NVPROF

- Nvprof supports the `tensor_precision_fu_utilization` metric which reveals the utilization level of Tensor Cores in each kernel of your model. (Since CUDA9)
- The utilization level of the multiprocessor function units that execute tensor core instructions on a scale of 0 to 10

```
nvprof -m tensor_precision_fu_utilization ./cudaTensorCoreGemm
```

Invocations	Metric Name	Metric Description	Min	Max	Avg
Device "Quadro GV100 (0)"					
Kernel: compute_gemm(__half const *, __half const *, float const *, float*, float, float)					
1	tensor_precision_fu_utilization	Tensor-Precision Function Unit Utilization	Mid (5)	Mid (5)	Mid (5)

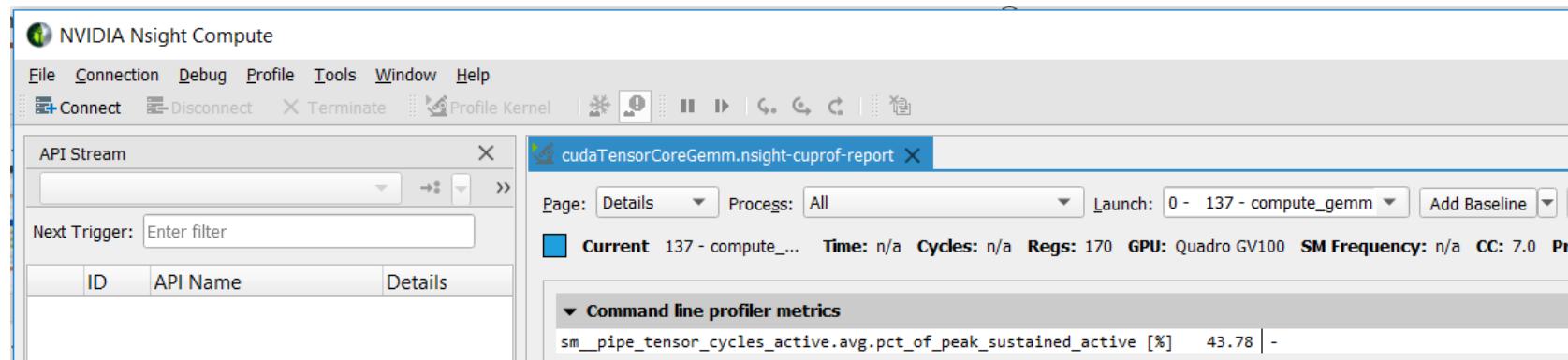
# TENSOR CORES WITH NSIGHT COMPUTE

- The Nsight Compute CLI allows collecting several metrics related to tensor core usage
- This data can be view from the CLI or via the Nsight Compute GUI

```
nv-nsight-cu-cli --metrics sm__pipe_tensor_cycles_active.avg.pct_of_peak_sustained_active ./cudaTensorCoreGemm
```

```
compute_gemm, 2019-Aug-08 12:48:39, Context 1, Stream 7
Section: Command line profiler metrics
```

```
-----  
sm__pipe_tensor_cycles_active.avg.pct_of_peak_sustained_active % 43.44  
-----
```





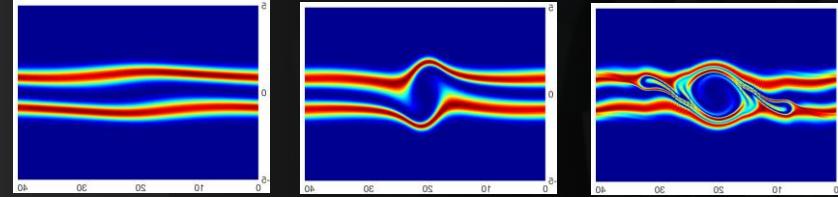
# CASE STUDIES



# ITERATIVE REFINEMENT (ASGARD & HPL)

# ADVANCING FUSION DISCOVERIES

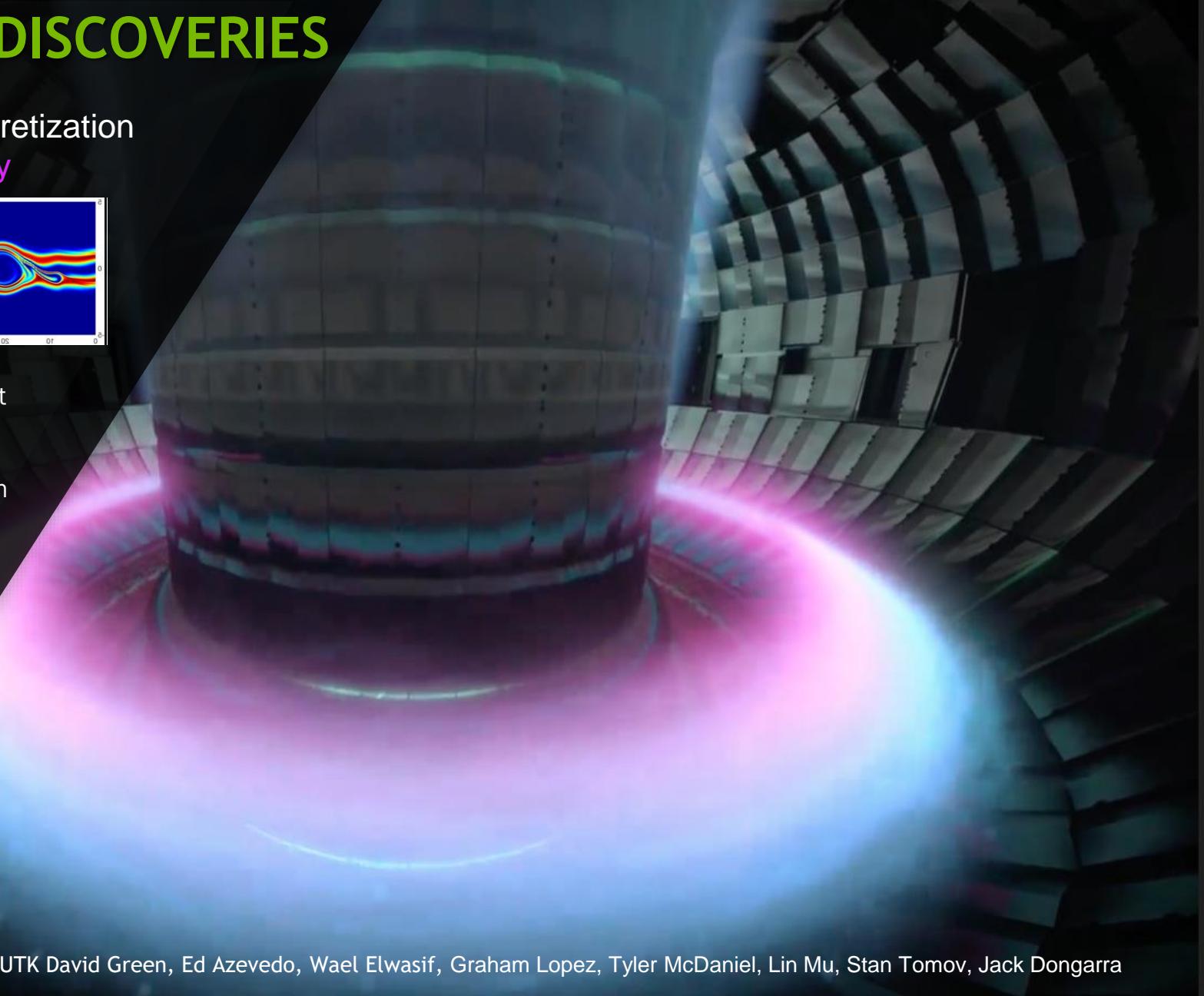
## ASGarD: Adaptive Sparse Grid Discretization Two stream instability study



Scientists believe fusion is the future of energy but maintaining plasma reactions is challenging and disruptions can result in damage to the tokamak. Researchers at ORNL are simulating instabilities in the plasma to provide physicists a better understanding of what happens inside the reactor.

### With NVIDIA Tensor Cores the simulations run 3.5X faster

than previous methods so the team can simulate significantly longer physical times and help advance our understanding of how to sustain the plasma and generate energy



Joint work with ORNL & UTK David Green, Ed Azevedo, Wael Elwasif, Graham Lopez, Tyler McDaniel, Lin Mu, Stan Tomov, Jack Dongarra

# ADVANCING FUSION DISCOVERIES

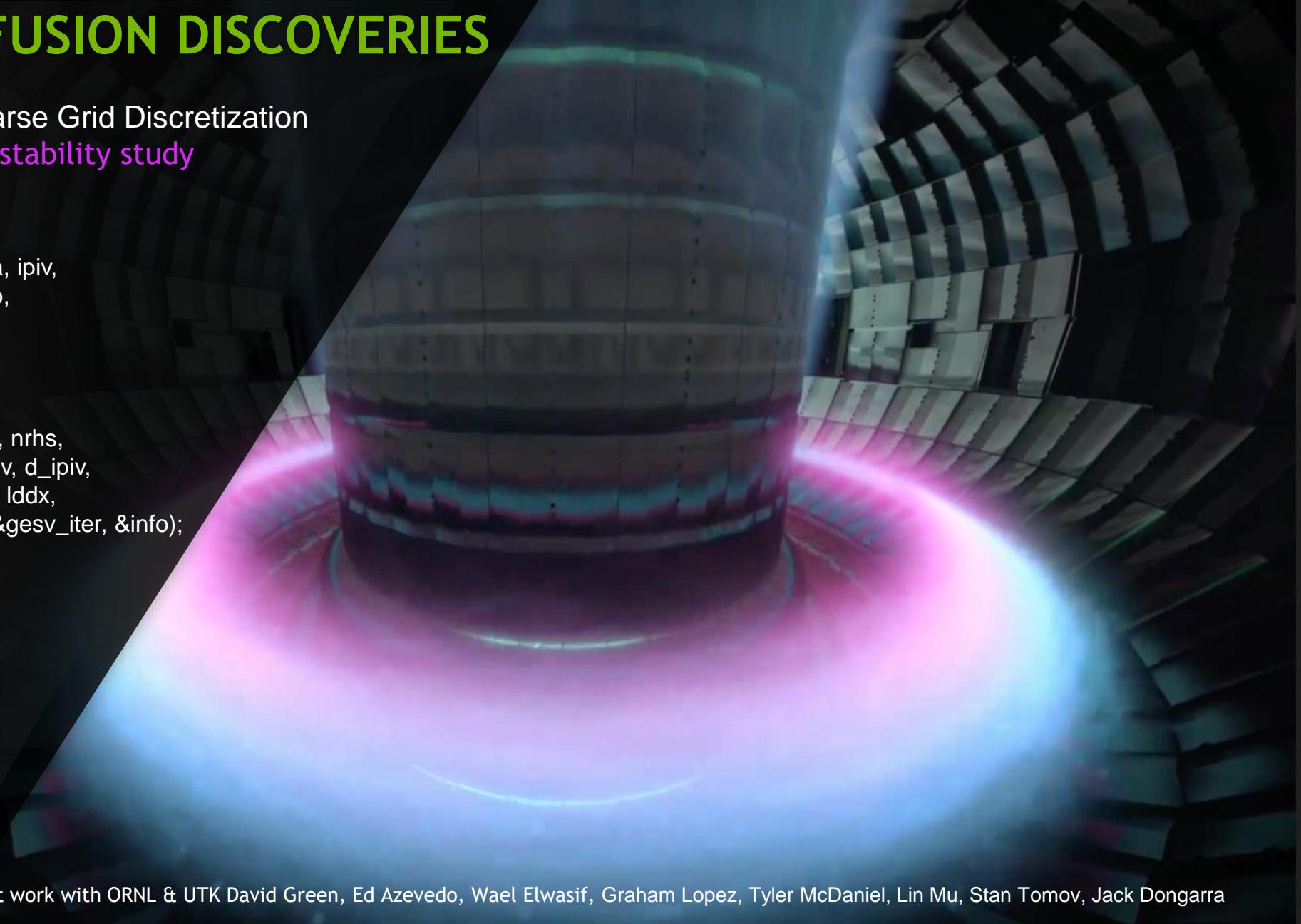
ASGarD: Adaptive Sparse Grid Discretization

Two stream instability study

```
magma_dgesv_gpu( N, nrhs,  
                  d_A, ldda, ipiv,  
                  d_B, lddb,  
                  &info );
```

is replaced by

```
magma_dhgesv_iteref_gpu( N, nrhs,  
                          d_A, ldda, h_ipiv, d_ipiv,  
                          d_B, lddb, d_X, lddx,  
                          d_workspace, &gesv_iter, &info);
```



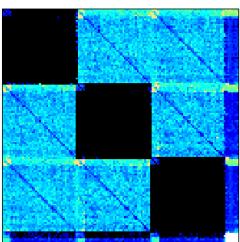
Joint work with ORNL & UTK David Green, Ed Azevedo, Wael Elwasif, Graham Lopez, Tyler McDaniel, Lin Mu, Stan Tomov, Jack Dongarra

# Mixed-precision iterative refinement solver

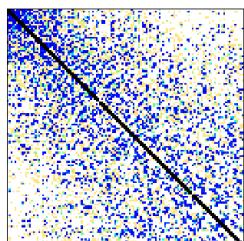
## Performance on a wider range of problems

PERFORMANCE FOR REAL-LIFE MATRICES FROM THE SUITESPARSE COLLECTION AND FROM DENSE MATRIX ARISING FROM RADAR DESIGN

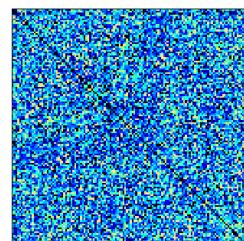
name	Description	size	$k_*(A)$	dgesv time(s)	dsgesv			dhgesv			dhgesv- TC		
					# iter	time (s)	speedup	# iter	time (s)	speedup	# iter	time (s)	speedup
em192	radar design	26896	$10^6$	5.70	3	3.11	1.8328	40	5.21	1.0940	10	2.05	2.7805
appu	NASA app benchmark	14000	$10^4$	0.43	2	0.27	1.5926	7	0.24	1.7917	4	0.19	2.2632
ns3Da	3D Navier Stokes	20414	$7.6 \cdot 10^3$	1.12	2	0.69	1.6232	6	0.54	2.0741	4	0.43	2.6047
nd6k	ND problem set	18000	$3.5 \cdot 10^2$	0.81	2	0.45	1.8000	5	0.36	2.2500	3	0.30	2.7000
nd12k	ND problem set	36000	$4.3 \cdot 10^2$	5.36	2	2.75	1.9491	5	1.86	2.8817	3	1.31	4.0916
Poisson	2D Poisson problem	32000	$2.1 \cdot 10^6$	3.81	2	2.15	1.7721	59	2.04	1.8676	10	1.13	3.3717
Vlasov	2D Vlasov problem	22000	$8.3 \cdot 10^3$	1.65	2	0.95	1.7368	4	0.67	2.4627	3	0.48	3.4375



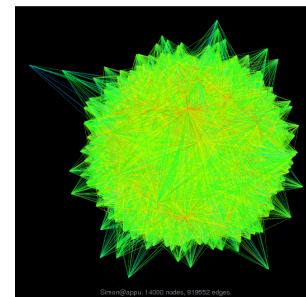
nd12k  
1,679,599  
nnz



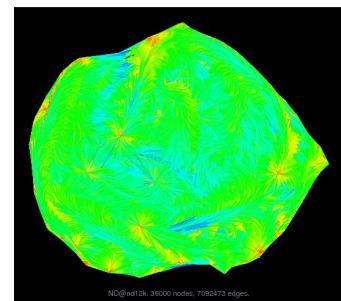
nd12k  
14,220,946  
nnz



appu  
1,853,104  
nnz



Simone@apgu: 14000 nodes, 919532 edges.



ND@nd12k: 31000 nodes, 7050473 edges.

# WORLD'S FASTEST SUPERCOMPUTER TRIPLES ITS PERFORMANCE RECORD...



# WORLD'S FASTEST SUPERCOMPUTER TRIPLES ITS PERFORMANCE RECORD...

Using mixed precision iterative refinement approach we solved a matrix of order 10,091,520 on the DOE's Summit system.

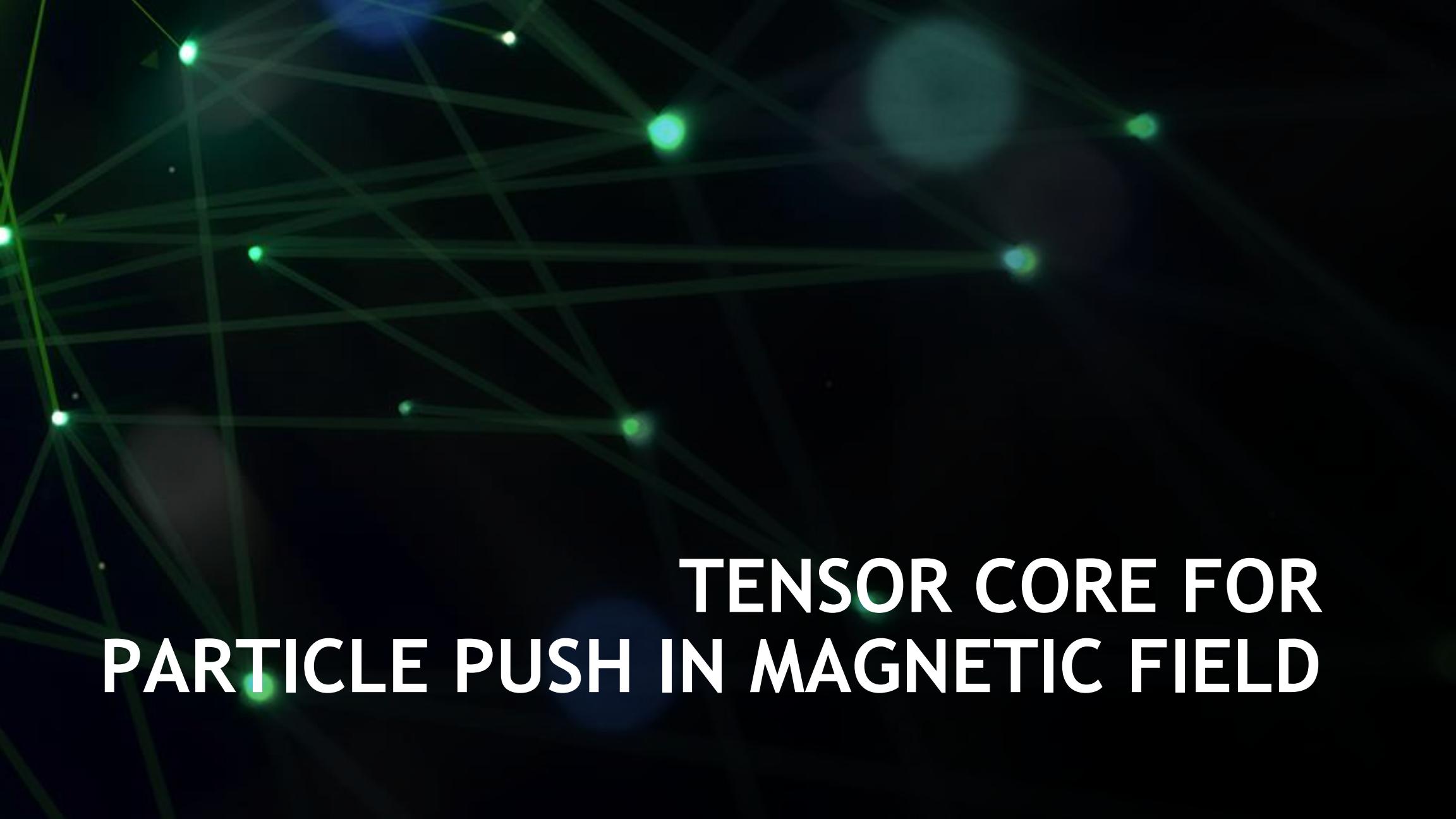
Composed of nodes made up of 2 IBM Power-9 processors, 22 cores each, and 6 Nvidia V100 GPUs

The run used 4500 nodes

Used a random matrix with large diagonal elements to insure convergence of the method.

Mixed precision HPL achieved 445 PFLOPS or 2.95X over DP precision HPL result on the Top500. (148 PFLOPS)



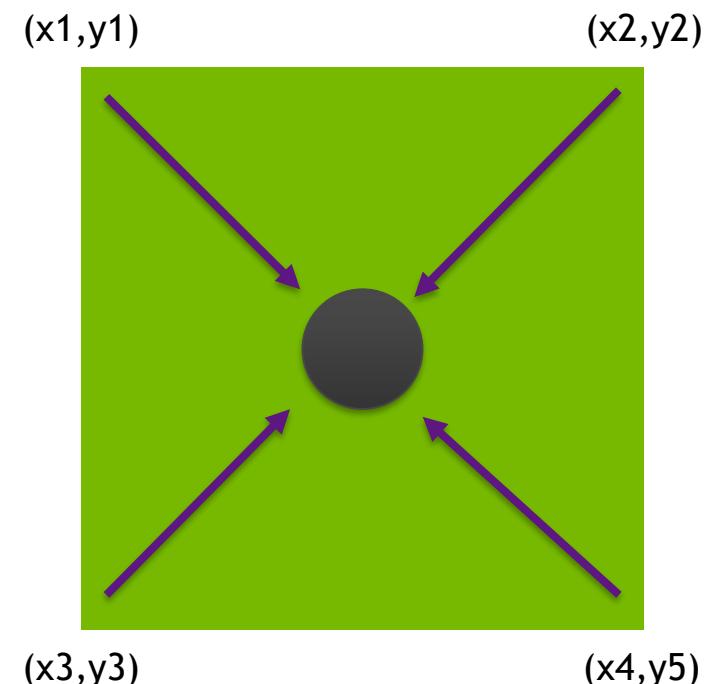


# TENSOR CORE FOR PARTICLE PUSH IN MAGNETIC FIELD

# PARTICLE PUSH

- The governing equation for particle velocity in magnetic field is given by:
- $\frac{dv}{dt} = \frac{q}{m(v \times B)}$ ,  $v = \text{velocity}$ ,  $q = \text{charge}$ ,  $m = \text{mass}$ ,  $B = \text{magnetic field}$
- \*Discretizing the above equation in 2 dimension can lead to :

```
/*grab magnetic field at current position*/  
B=EvalB(x);  
  
/*get new velocity at n+1*/  
v2[0] = v[0] + q/m*B*v[1]*dt;  
v2[1] = v[1] - q/m*B*v[0]*dt;  
  
/*update position*/  
x2[0] = x[0] + v2[0]*dt;  
x2[1] = x[1] + v2[1]*dt;  
  
/*push down*/  
v[0]=v2[0];  
v[1]=v2[1];
```



Gather by magnetic forces from the cell vertices.

# BORIS METHOD

\*Boris method is the *de facto* standard for particle pushing in plasma simulation codes  
It is an explicit technique

The following equations summarize Boris method

In the absence of Electric Field.  $\nabla^+$  acts as velocity update. Electric field can be easily added.

$$\frac{v^+ - v^-}{\Delta t} = \frac{q}{2m} (v^+ + v^-) \times B$$

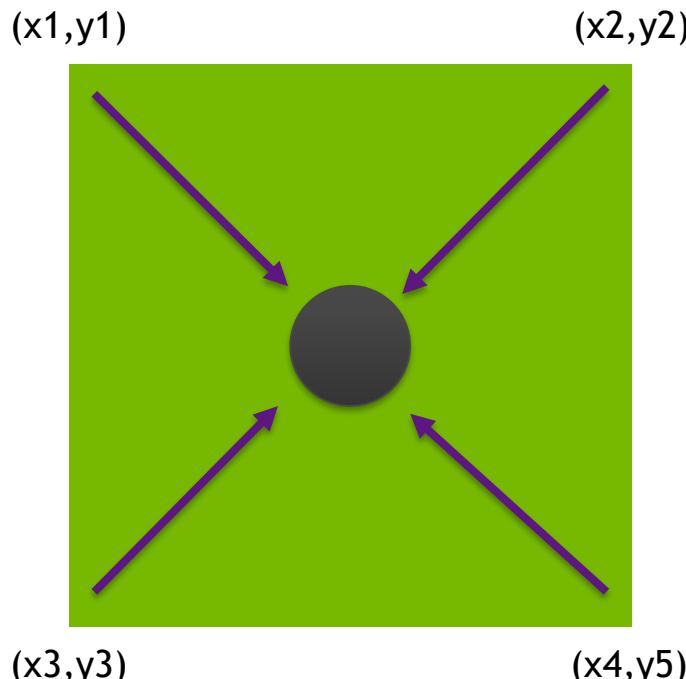
$$v' = v^- + v^- \times t \quad t = (q B / m) \Delta t / 2$$

$$v^+ = v^- + v' \times s \quad s = \frac{2t}{1 + t^2}$$

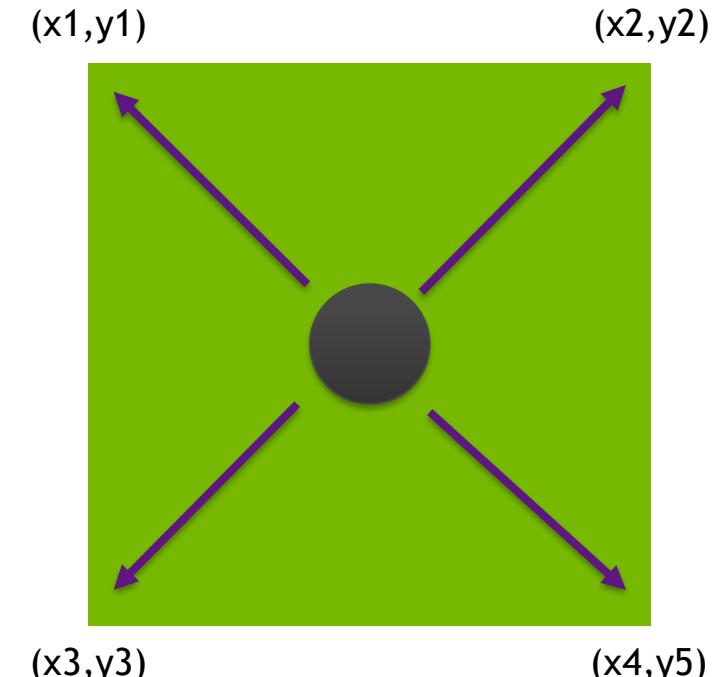
\*Ref: <https://www.particleincell.com/2011/vxb-rotation/>

# SCATTER PARTICLE INSTEAD OF GATHER

- We separate velocity direction and magnitude. Magnitude in FP32 while directions in FP16
- We pack velocity, t and s vectors into Tensor Core format. This is basically the scatter operation.
- The GEMM updates velocities and add them back to particle final velocity at a given time step in FP32



To use Tensor core, scatter properties  
of the particles and use WMMA to  
compute and assemble



Gather by interpolation forces from the cell  
vertices.

Scatter particle properties to nodes and add  
compute at nodes.

# IMPLEMENTING WITH WMMA

```
// Half-precision, no tensor core
int id = (int)(threadIdx.x % 32);

for(int k = 0;k<8;k++)
{
    float t =0;
    for(int i =0;i<16;i++)
    {
        t += __half2float(
            __hmul(X[i + k*16] ,
                    Y[id + i * 32]) );
    }
    accmat[id + k*32] = t;
}

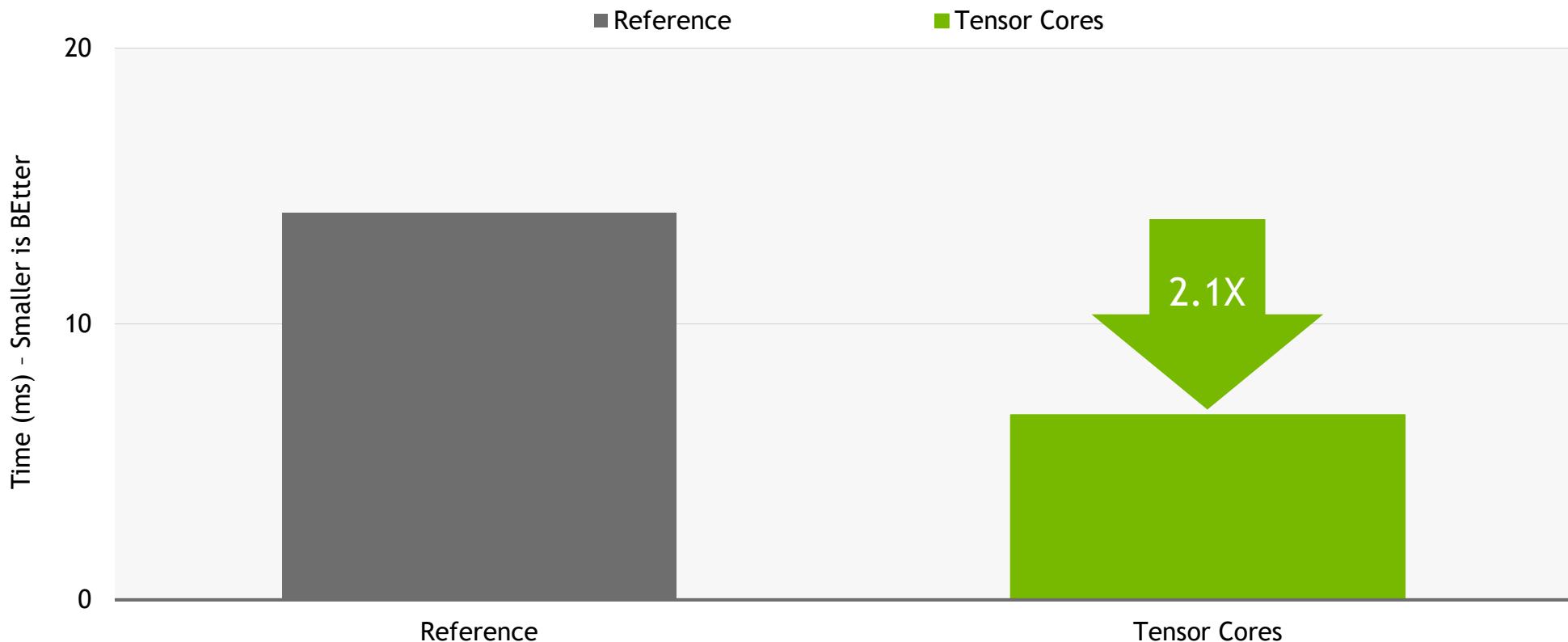
return;
```

```
// Half-precision, with tensor core
// Declare the fragments
nvcuda::wmma::fragment<nvcuda::wmma::matrix_a,
8, 32, 16, half, nvcuda::wmma::col_major>
a_frag;
nvcuda::wmma::fragment<nvcuda::wmma::matrix_b,
8, 32, 16, half, nvcuda::wmma::row_major>
b_frag;
nvcuda::wmma::fragment<nvcuda::wmma::accumulator,
8, 32, 16, float> acc_frag;
nvcuda::wmma::fill_fragment(acc_frag, 0.0f);
nvcuda::wmma::load_matrix_sync(a_frag, a, 8);
nvcuda::wmma::load_matrix_sync(b_frag, b, 32);

nvcuda::wmma::mma_sync(acc_frag, a_frag,
b_frag, acc_frag);

nvcuda::wmma::store_matrix_sync(c , acc_frag,
32, nvcuda::wmma::mem_row_major);
return;
```

# PICTC PERFORMANCE COMPARISON



Source: CUDA 10.1, Summit



# MACHINE LEARNING

# DGX Mixed-Precision Led MLPerf

World's Fastest Industry-Wide AI Benchmark Achieved on NVIDIA GPUs

## Time to Accuracy on Single Node

Image Classification  
RN50 v.1.5  
MXNet

70 minutes

Object Detection  
Mask R-CNN  
PyTorch

167 minutes

Object Detection  
SSD  
PyTorch

14 minutes

Translation (recurrent)  
GNMT  
PyTorch

10 minutes

Translation (non-  
recurrent)  
Transformer

19 minutes

Recommendation  
NCF  
PyTorch

0.4 minutes

Test Platform: DGX-2H - Dual-Socket Xeon Platinum 8174, 1.5TB system RAM, 16 x 32 GB Tesla V100 SXM-3 GPUs connected via NVSwitch

# NVIDIA NGC MODEL SCRIPTS

## Tensor Core Optimized Deep Learning Examples

14 Available today!

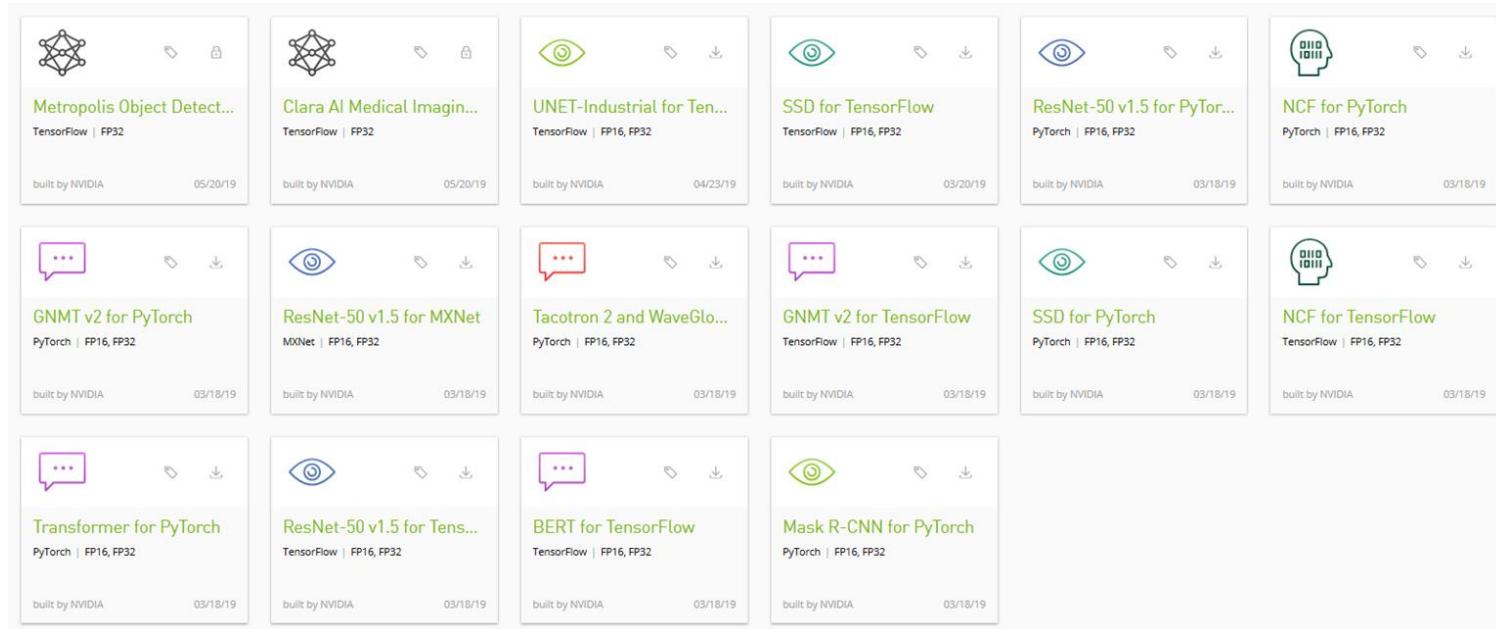
- Tensor Core optimized for greater performance
- Test drive automatic mixed precision
- Actively updated by NVIDIA
- State-of-the-art accuracy using Tensor Cores
- Serves as a reference implementation
- Exposes hyperparameters and source code for further adjustment

Accessible via:

- NVIDIA NGC <https://ngc.nvidia.com/catalog/model-scripts>
- GitHub <https://www.github.com/NVIDIA/deeplearningexamples>
- NVIDIA NGC Framework containers <https://ngc.nvidia.com/catalog/containers>

# NVIDIA NGC MODEL SCRIPTS

Tensor Core Examples Built for Multiple Use Cases and Frameworks



A dedicated hub to download Tensor Core Optimized Deep Learning Examples on NGC

<https://ngc.nvidia.com/catalog/model-scripts?quickFilter=deep-learning>

# MODEL SCRIPTS FOR VARIOUS APPLICATIONS

<https://developer.nvidia.com/deep-learning-examples>

## Computer Vision

- SSD PyTorch
- SSD TensorFlow
- UNET-Industrial TensorFlow
- UNET-Medical TensorFlow
- ResNet-50 v1.5 MXNet
- ResNet-50 PyTorch
- ResNet-50 TensorFlow
- Mask R-CNN PyTorch

## Speech & NLP

- GNMT v2 TensorFlow
- GNMT v2 PyTorch
- Transformer PyTorch
- BERT (Pre-training and Q&A)  
TensorFlow

## Recommender Systems

- NCF PyTorch
- NCF TensorFlow

## Text to Speech

- Tacotron2 and WaveGlow  
PyTorch

# IMAGE CLASSIFICATION: MXNet ResNet-50 v1.5

[https://ngc.nvidia.com/catalog/model-scripts/nvidia:resnet\\_50\\_v1\\_5\\_for\\_mxnet](https://ngc.nvidia.com/catalog/model-scripts/nvidia:resnet_50_v1_5_for_mxnet)

DGX-1V 8GPU 16G	MXNet ResNet FP32	MXNet ResNet Mixed Precision
Time to Train [Hours]	<b>11.1</b>	<b>3.3</b>
Train Accuracy Top 1%	<b>76.67%</b>	<b>76.49%</b>
Perf.	<b>2,957</b> Img/sec	<b>10,263</b> Img/sec
Data set	ImageNet	

ResNet-50 v1.5 for MXNet

Publisher: NVIDIA Application: Classification Version: 1 Last Modified: March 18, 2019 Training Framework: MXNet

Model Format: MXNet params + json Precision: FP16, FP32

Description: MXNet scripts for defining, training and using ResNet-50 v1.5 model optimized for Tensor Cores. With modified architecture and initialization this ResNet50 version gives ~0.5% better accuracy than original.

Labels: DEEP LEARNING, TRAINING

Overview Setup Quick Start Guide Performance Version History File Browser Release Notes

### Overview

The ResNet50 v1.5 model is a modified version of the [original ResNet50 v1 model](#). The difference between v1 and v1.5 is in the bottleneck blocks which require downampling. ResNet v1 has stride = 2 in the first 1x1 convolution, whereas v1.5 has stride = 2 in the 3x3 convolution. This difference makes ResNet50 v1.5 slightly more accurate (~0.5% top1) than v1, but comes with a small performance drawback (~5% imgs/sec).

### Default configuration

#### Optimizer

This model trains for 90 epochs, with the standard ResNet v1.5 setup:

- SGD with momentum (0.9)
- Learning rate = 0.1 for 256 batch size, for other batch sizes we linearly scale the learning rate.
- Learning rate decay - multiply by 0.1 after 30, 60, and 80 epochs
- Linear warmup of the learning rate during first 5 epochs according to [Accurate Large Minibatch SGD: Training ImageNet in 1 Hour](#).
- Weight decay: 1e-4

#### Data Augmentation

During training, we perform the following augmentation techniques:

- Normalization
- Random resized crop to 224x224
- Scale from 5% to 100%
- Aspect ratio from 3/4 to 4/3
- Random horizontal flip

During inference, we perform the following augmentation techniques:

- Normalization
- Scale to 256x256
- Center crop to 224x224

See `data.py` for more info.

NGC 18.12+ MXNet container

Source: <https://github.com/NVIDIA/DeepLearningExamples/tree/master/MxNet/Classification/RN50v1.5>

GPU: 1xV100-16GB | DGX-1V | Batch Size: 208 (FP16), 96 (FP32)

# SPEECH SYNTHESIS: Tacotron 2 And WaveGlow v1.0

[https://ngc.nvidia.com/catalog/model-scripts/nvidia:tacotron\\_2\\_and\\_waveglow\\_for\\_pytorch](https://ngc.nvidia.com/catalog/model-scripts/nvidia:tacotron_2_and_waveglow_for_pytorch)

DGX-1V 16G	Tacotron 2 FP32	Tacotron 2 Mixed Precision	WaveGlow FP32	WaveGlow Mixed Precision
Time to Train [Hours]	<b>44</b> @ 1500 epochs	<b>33.14</b> @ 1500 epochs	<b>109.96</b> @ 1000 epochs	<b>54.83</b> @ 1000 epochs
Train Accuracy Loss (@1000 Epochs)	<b>0.3629</b>	<b>0.3645</b>	<b>-6.1087</b>	<b>-6.0258</b>
Perf.	<b>10,843</b> tokens/sec	<b>12,742</b> tokens/sec	<b>257,687(*)</b> samples/sec	<b>500,375(*)</b> samples/sec
Data set	LJ Speech Dataset			

(\*) With sampling rate equal to 22050, one second of audio is generated from 22050 samples

< Tacotron 2 and WaveGlow for PyTorch

Publisher NVIDIA Application Text To Speech Version 1 Last Modified March 18, 2019 Training Framework PyTorch

Model Format Pytorch PTH Precision FP16, FP32

Description PyTorch scripts for defining, training and using Tacotron 2 and WaveGlow model optimized for Tensor Cores. The Tacotron 2 and WaveGlow model form a text-to-speech system that enables user to synthesise a natural sounding speech from raw transcripts.

Labels DEEP LEARNING TRAINING

Overview Setup Quick Start Guide Performance Version History File Browser Release Notes

### Overview

This text-to-speech (TTS) system is a combination of two neural network models:

- a modified Tacotron 2 model from the [Natural TTS Synthesis by Conditioning WaveNet on Mel Spectrogram Predictions](#) paper
- a flow-based neural network model from the [WaveGlow: A Flow-based Generative Network for Speech Synthesis](#) paper.

The Tacotron 2 and WaveGlow model form a text-to-speech system that enables user to synthesise a natural sounding speech from raw transcripts without any additional prosody information.

Our implementation of Tacotron 2 model differs from the model described in the paper. Our implementation uses Dropout instead of Zoneout to regularize the LSTM layers. Also, the original text-to-speech system proposed in the paper used the [WaveNet](#) model to synthesize waveforms. In our implementation, we use the WaveGlow model for this purpose.

Both models are based on implementations of NVIDIA GitHub repositories [Tacotron 2](#) and [WaveGlow](#), and are trained on a publicly available [Speech dataset](#).

This model trains with mixed precision tensor cores on Volta, therefore researchers can get results much faster than training without tensor cores. This model is tested against each NGC monthly container release to ensure consistent accuracy and performance over time.

# LANGUAGE MODELING: BERT for TensorFlow

[https://ngc.nvidia.com/catalog/model-scripts/nvidia:bert\\_for\\_tensorflow](https://ngc.nvidia.com/catalog/model-scripts/nvidia:bert_for_tensorflow)

DGX-1V 8GPU 32G	TF BERT FP32	TF BERT Mixed Precision
Time to Train [Hours]	<b>0.77</b> (BSxGPU = 4)	<b>0.51</b> (BSxGPU = 4)
Train F1 (mean)	<b>90.83</b>	<b>90.99</b>
Perf. (BSxGPU = 4)	<b>66.65</b> sentences/sec	<b>129.16</b> sentences/sec
Data set	SQuAD (fine-tuning)	

< BERT for TensorFlow

Publisher	Application	Version	Last Modified	Training Framework
NVIDIA	Translation	1	March 18, 2019	TensorFlow

**Model Format** TensorFlow CKPT    **Precision** FP16, FP32

**Description**  
TensorFlow scripts for defining, training and using BERT model optimized for Tensor Cores. BERT is a new method of pre-training language representations which obtains state-of-the-art results on a wide array of NLP tasks.

**Labels** DEEP LEARNING TRAINING

[Overview](#) [Setup](#) [Quick Start Guide](#) [Performance](#) [Version History](#) [File Browser](#) [Release Notes](#)

---

## Overview

BERT, or Bidirectional Encoder Representations from Transformers, is a new method of pre-training language representations which obtains state-of-the-art results on a wide array of Natural Language Processing (NLP) tasks. This model is based on [BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding](#) paper. NVIDIA's BERT 19.03 is an optimized version of [Google's official implementation](#), leveraging mixed precision arithmetic and tensor cores on V100 GPUs for faster training times while maintaining target accuracy.

The repository also contains scripts to interactively launch data download, training, benchmarking and inference routines in a Docker container for both pretraining and fine tuning for Question Answering. The major differences between the official implementation of the paper and our version of BERT are as follows:

- Mixed precision support with TensorFlow Automatic Mixed Precision (TF-AMP), which enables mixed precision training without any changes to the code-base by performing automatic graph rewrites and loss scaling controlled by an environmental variable.
- Scripts to download dataset for
  - Pretraining - [Wikipedia](#), [BookCorpus](#)
  - Fine Tuning - [SQuAD](#) (Stanford Question Answering Dataset), Pretrained Weights from Google

NGC 19.03 TensorFlow container

Source: <https://github.com/NVIDIA/DeepLearningExamples/tree/master/TensorFlow/LanguageModeling/BERT>

GPU:8xV100-32GB | DGX-1 | Batch size per GPU: 4

# OBJECT DETECTION: TensorFlow SSD

[https://ngc.nvidia.com/catalog/model-scripts/nvidia:ssd\\_for\\_tensorflow](https://ngc.nvidia.com/catalog/model-scripts/nvidia:ssd_for_tensorflow)

DGX-1V 8GPU 16G	TF SSD FP32	TF SSD Mixed Precision
Time to Train	<b>1h 37min</b>	<b>1h 19min</b>
Accuracy (map)	<b>0.268</b>	<b>0.269</b>
Perf. (BSxGPU = 32)	<b>569</b> Img/sec	<b>752</b> Img/sec
Data set	COCO 2017	

**SSD for TensorFlow**

Publisher: NVIDIA Application: Object Detection Version: 1 Last Modified: March 20, 2019 Training Framework: TensorFlow

Model Format: TensorFlow CKPT Precision: FP16, FP32

Description: TensorFlow scripts for defining, training and using SSD model optimized for Tensor Cores. With a ResNet-50 backbone and a number of architectural modifications, this version provides better accuracy and performance.

Labels: DEEP LEARNING, TRAINING

[Overview](#) [Setup](#) [Quick Start Guide](#) [Performance](#) [Version History](#) [File Browser](#) [Release Notes](#)

### Overview

The SSD320 v1.2 model is based on the [SSD: Single Shot MultiBox Detector](#) paper, which describes SSD as "a method for detecting objects in images using a single deep neural network". We have altered the network in order to improve accuracy and increase throughput. Changes we have made include:

- Replacing the VGG backbone with the more popular ResNet50.
- Adding multi-scale detection to the backbone using [Feature Pyramid Networks](#).
- Replacing the original hard negative mining loss function with [Focal Loss](#).
- Decreasing the input size to 320 x 320.

Our implementation is based on the existing [model from the TensorFlow models repository](#). This model trains with mixed precision tensor cores on NVIDIA Volta GPUs, therefore you can get results much faster than training without tensor cores. This model is tested against each NGC monthly container release to ensure consistent accuracy and performance over time.

The following features were implemented in this model:

- Data-parallel multi-GPU training with Horovod.
- Training with TensorFlow Automatic Mixed Precision (TF-AMP), which enables mixed precision training without any changes to the code-base by performing automatic graph rewrites and loss scaling controlled by an environmental variable.
- Tensor Core operations to maximize throughput using NVIDIA Volta GPUs.
- Dynamic loss scaling for tensor cores (mixed precision training).

Because of these enhancements, the SSD320 v1.2 model achieves higher accuracy.

### Default configuration

We trained the model for 12500 steps (27 epochs) with the following setup:

- SGDR with cosine decay learning rate
- Learning rate base = 0.16
- Momentum = 0.9
- Warm-up learning rate = 0.0693312
- Warm-up steps = 1000
- Batch size per GPU = 32
- Number of GPUs = 8

## NGC 19.03 TensorFlow container

Source: <https://github.com/NVIDIA/DeepLearningExamples/tree/master/TensorFlow/Detection/SSD>

GPU:8xV100-16GB | DGX-1V | Batch Size: 32 (FP32, Mixed)

# TRANSLATION: PyTorch GNMT

[https://ngc.nvidia.com/catalog/model-scripts/nvidia:gnmt\\_v2\\_for\\_pytorch](https://ngc.nvidia.com/catalog/model-scripts/nvidia:gnmt_v2_for_pytorch)

DGX-2V 16GPU 32G	PyTorch GNMT FP32	PyTorch GNMT Mixed Precision
Time to Train [min]	<b>58.6</b>	<b>26.3</b>
Train Accuracy BLEU score	<b>24.16</b>	<b>24.22</b>
Perf.	<b>314.831</b> tokens/sec	<b>738,521</b> tokens/sec
Data set	WMT16 English to German	

< GNMT v2 for PyTorch

Publisher	NVIDIA	Application	translation	Version	1	Last Modified	March 18, 2019	Training Framework	PyTorch
Model Format	Pytorch PTH	Precision	FP16, FP32						
Description	PyTorch scripts for defining, training and using GNMT v2 model optimized for Tensor Cores. The GNMT v2 model is an improved version of the first Google's Neural Machine Translation System with a modified attention mechanism.								
Labels	<span>DEEP LEARNING</span> <span>TRAINING</span>								

[Overview](#) [Setup](#) [Quick Start Guide](#) [Performance](#) [Version History](#) [File Browser](#) [Release Notes](#)

**Overview**

The GNMT v2 model is similar to the one discussed in the [Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation](#) paper.

The most important difference between the two models is in the attention mechanism. In our model, the output from the first LSTM layer of the decoder goes into the attention module, then the re-weighted context is concatenated with inputs to all subsequent LSTM layers in the decoder at the current timestep.

The same attention mechanism is also implemented in the default GNMT-like models from [TensorFlow Neural Machine Translation Tutorial](#) and [NVIDIA OpenSeq2Seq Toolkit](#).

**Default configuration**

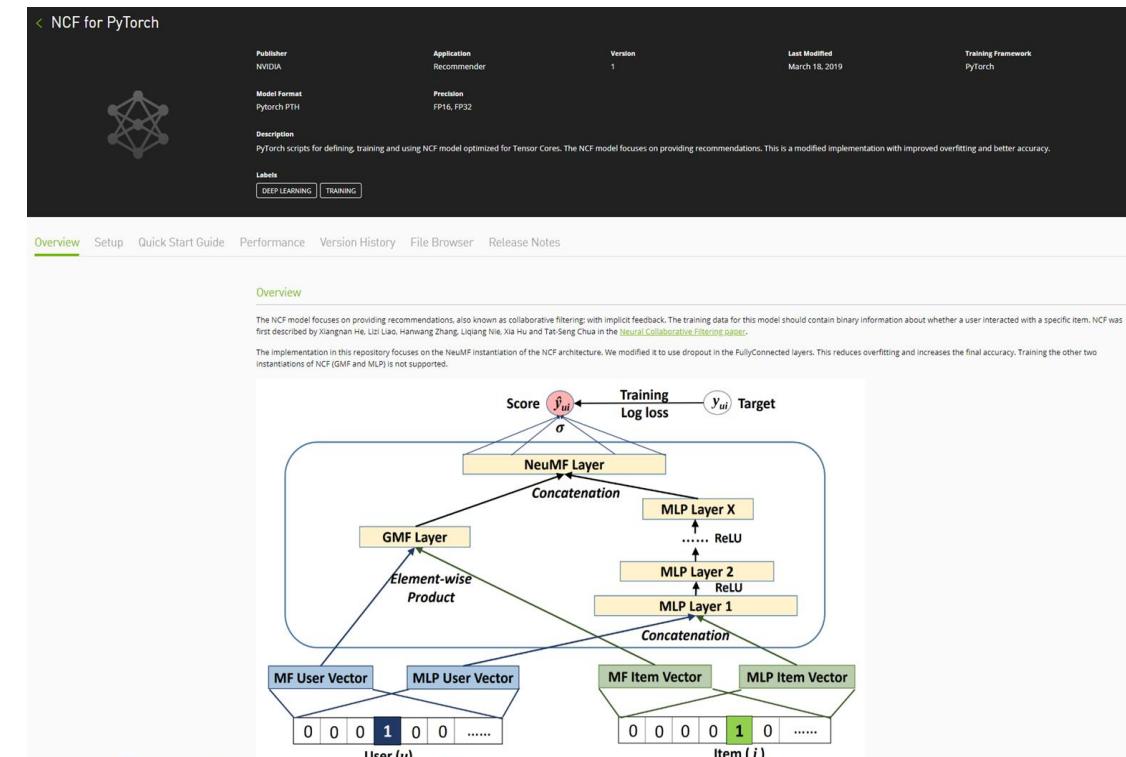
- general:
  - encode and decode are using shared embeddings
  - data-parallel multi-gpu training
  - dynamic loss scaling with backoff for Tensor Cores (mixed precision) training
  - trained with label smoothing loss (smoothing factor 0.1)
- encoder:
  - 4-layer LSTM, hidden size 1024, first layer is bidirectional, the rest are unidirectional
  - residual connections starting from 3rd layer
  - uses standard pytorch nn.LSTM layer
  - dropout is applied on input to all LSTM layers, probability of dropout is set to 0.2
  - hidden state of LSTM layers is initialized with zeros
  - weights and bias of LSTM layers is initialized with uniform(-0.1, 0.1) distribution
- decoder:
  - 4-layer unidirectional LSTM with hidden size 1024 and fully-connected classifier
  - with residual connections starting from 3rd layer
  - uses standard pytorch nn.LSTM layer
  - dropout is applied on input to all LSTM layers, probability of dropout is set to 0.2
  - hidden state of LSTM layers is initialized with zeros
  - weights and bias of LSTM layers is initialized with uniform(-0.1, 0.1) distribution
  - weights and bias of fully-connected classifier is initialized with uniform(-0.1, 0.1) distribution
- attention:
  - normalized Bahdanau attention
  - output from first LSTM layer of decoder goes into attention, then re-weighted context is concatenated with the input to all subsequent LSTM layers of the decoder at the current timestep
  - linear transform of keys and queries is initialized with uniform(-0.1, 0.1), normalization scalar is initialized with  $1.0 / \sqrt{1024}$ , normalization bias is initialized with zero
- inference:
  - beam search with default beam size of 5

NGC 19.01 PyTorch container

# RECOMMENDER: PyTorch Neural Collaborative Filter

[https://ngc.nvidia.com/catalog/model-scripts/nvidia:ncf\\_for\\_pytorch](https://ngc.nvidia.com/catalog/model-scripts/nvidia:ncf_for_pytorch)

DGX-1V 8GPU 16G	PyTorch NCF FP32	PyTorch NCF Mixed Precision
Time to Accuracy [seconds]	32.68	20.42
Accuracy Hit Rate @ 10	0.96	0.96
Perf.	55,004,590 smp/sec	99,332,230 smp/sec
Data set	MovieLens 20M	



NGC 18.12 PyTorch container

Source: <https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/Recommendation/NCF>  
GPU:8xV100-16GB | DGX-1 | Batch size: 1,048,576

# INDUSTRIAL DEFECT DETECTION: TensorFlow U-Net

[https://ngc.nvidia.com/catalog/model-scripts/nvidia:unet\\_industrial\\_for\\_tensorflow](https://ngc.nvidia.com/catalog/model-scripts/nvidia:unet_industrial_for_tensorflow)

DGX-1V 8GPU 16G	TF U-Net FP32	TF U-Net Mixed Precision
Time to Train	<b>1 min 44 sec</b>	<b>1 min 36 sec</b>
IOU (Th=0.75 Class #4)	<b>0.965</b>	<b>0.960</b>
IOU (Th=0.75 Class #9)	<b>0.988</b>	<b>0.988</b>
Perf.	<b>445</b> Img/sec	<b>491</b> Img/sec
Data set	DAGM 2007	

UNET-Industrial for TensorFlow

Publisher: NVIDIA Application: Segmentation Version: 1 Last Modified: April 23, 2019 Training Framework: TensorFlow

Model Format: TensorFlow CKPT Precision: FP16, FP32

Description: TensorFlow scripts for defining, training and using UNET-Industrial model optimized for Tensor Cores. This model is a convolutional neural network for 2D image segmentation tuned to avoid overfitting.

Labels: DEEP LEARNING, TRAINING

Overview Setup Quick Start Guide Performance Version History File Browser Release Notes

### Overview

This U-Net model is adapted from the original version of the [U-Net model](#) which is a convolutional auto-encoder for 2D image segmentation. U-Net was first introduced by Olaf Ronneberger, Philipp Fischer, and Thomas Brox in the paper: [U-Net: Convolutional Networks for Biomedical Image Segmentation](#).

This work proposes a modified version of U-Net, called [TinyUNet](#) which performs efficiently and with very high accuracy on the industrial anomaly dataset [DAGM2007](#). [TinyUNet](#), like the original U-Net is composed of two parts:

- an encoding sub-network (left-side)
- a decoding sub-network (right-side).

It repeatedly applies 3 downsampling blocks composed of two 2D convolutions followed by a 2D max pooling layer in the encoding sub-network. In the decoding sub-network, 3 upsampling blocks are composed of a upsample2D layer followed by a 2D convolution, a concatenation operation with the residual connection and two 2D convolutions.

[TinyUNet](#) has been introduced to reduce the model capacity which was leading to a high degree of over-fitting on a small dataset like DAGM2007. The complete architecture is presented in the figure below:

## NGC 19.03 TensorFlow container

Source: [https://github.com/NVIDIA/DeepLearningExamples/tree/master/TensorFlow/Segmentation/UNet\\_Industrial](https://github.com/NVIDIA/DeepLearningExamples/tree/master/TensorFlow/Segmentation/UNet_Industrial)

GPU:8xV100-16GB | DGX-1 | Batch size: 16

DAGM 2007 has 10 classes (for the competition). Each class has an independent IOU.

# Matching Accuracy for FP32 and Mixed Precision

Model Script	Framework	Data Set	Automatic or Manual Mixed-Precision	FP32 Accuracy	Mixed-Precision Accuracy	FP32 Throughput	Mixed-Precision Throughput	Speedup
<a href="#">BERT Q&amp;A</a> <small>(2)</small>	<b>TensorFlow</b>	SQuAD	AMP	90.83 Top 1	90.99 Top 1	66.65 sentences/sec	129.16 sentences/sec	<b>1.94</b>
<a href="#">SSD w/RN50</a> <small>(1)</small>	<b>TensorFlow</b>	COCO 2017	AMP	0.268 mAP	0.269 mAP	569 images/sec	752 images/sec	<b>1.32</b>
<a href="#">GNMT</a> <small>(3)</small>	<b>PyTorch</b>	WMT16 English to German	Manual	24.16 BLEU	24.22 BLEU	314,831 tokens/sec	738,521 tokens/sec	<b>2.35</b>
<a href="#">Neural Collaborative Filter</a> <small>(1)</small>	<b>PyTorch</b>	MovieLens 20M	Manual	0.959 HR	0.960 HR	55,004,590 samples/sec	99,332,230 items/sec	<b>1.81</b>
<a href="#">U-Net Industrial</a> <small>(1)</small>	<b>TensorFlow</b>	DAGM 2007	AMP	0.965-0.988	0.960-0.988	445 images/sec	491 images/sec	<b>1.10</b>
<a href="#">ResNet-50 v1.5</a> <small>(1)</small>	<b>MXNet</b>	ImageNet	Manual	76.67 Top 1%	76.49 Top 1%	2,957 images/sec	10,263 images/sec	<b>3.47</b>
<a href="#">Tacotron 2 / WaveGlow 1.0</a> <small>(1)</small>	<b>PyTorch</b>	LJ Speech Dataset	AMP	0.3629/-6.1087	0.3645/-6.0258	10,843 tok/s 257,687 smp/s	12,742 tok/s 500,375 smp/s	<b>1.18/1.94</b>

Values are measured with model running on (1) DGX-1V 8GPU 16G, (2) DGX-1V 8GPU 32G or (3) DGX-2V 16GPU 32G



# NON-TRADITIONAL USES

# NON-TRADITIONAL USE OF TENSOR CORES

When you have a GEMM-shaped hammer...

Many problems can be reformulated in terms of dense matrix multiplication

May not be most algorithmically efficient, but tensor core performance can make up for large constant factor differences

Example: computing correlations between sets of binary vectors  
e.g. for clustering points in  $\{0,1\}^N$  [Joubert et al, 2018]

$$C(i,j) = \sum_k^N [A_i(k) \wedge B_j(k)] = \sum_k^N [A_i(k)] * [B_j(k)] \Rightarrow C_{ij} = \sum_k A_{ik} B_{jk} \Rightarrow C = AB^T$$

Perfect use case for reduced precision: inputs are all in  $[0,1]$ , outputs in  $[0,N]$  (or better)

See [https://www.olcf.ornl.gov/wp-content/uploads/2018/10/joubert\\_2019OLCFUserMeeting.pdf](https://www.olcf.ornl.gov/wp-content/uploads/2018/10/joubert_2019OLCFUserMeeting.pdf)

# HGEMM VS GEMMEX

```
cublasSetMathMode(handle, CUBLAS_TENSOR_OP_MATH);
const __half *A = ...;
const __half *B = ...;
__half *C = ...; ←
cublasHgemm(handle, transa, transb, m, n, k,
            alpha, A, lda, B, ldb,
            beta, C, ldc);
```

accumulates in FP16

exact results only up to  $N < 2048$

VS.

```
...
float *C = ...; ←
cublasGemmEx(handle, transa, transb, m, n, k,
            alpha, A, CUDA_R_16F, lda, B, CUDA_R_16F, ldb,
            beta, C, CUDA_R_32F, ldc,
            CUDA_R_32F,
            CUBLAS_GEMM_DEFAULT_TENSOR_OP); ←
```

accumulates in FP32

nearly as fast as cublasHgemm  
(same datapath, just a bit more I/O)

exact results up to  $N < 2^{24}$

make sure to ask for tensor cores!

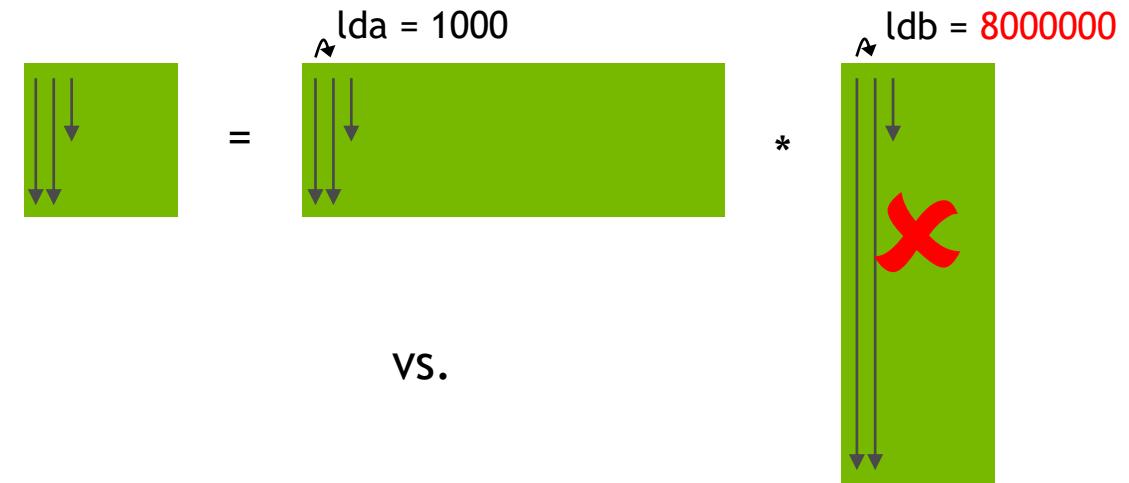
# PITFALL: LARGE VALUES FOR LD{A,B,C}

When N gets large, A and B matrices can get very long and skinny

Prefer the memory layout that keeps lda and ldb small

Change the transa/transb parameters on cublas\*Gemm\* to match

Your caches and TLBs will thank you!



# CONCLUSIONS

# CONCLUSIONS

When used appropriately Tensor Cores can achieve as much as an 8X performance increase.

Real-world applications have seen > 2X performance improvement.

High-throughput Matrix Multiplication requires careful data considerations

A variety of High and Low-level approaches are available for programming Tensor Cores

Tensor Cores show promise beyond Machine Learning applications

# ADDITIONAL RESOURCES

CUTLASS Basics - <http://on-demand.gputechconf.com/gtc/2018/presentation/s8854-cutlass-software-primitives-for-dense-linear-algebra-at-all-levels-and-scales-within-cuda.pdf>

cuTensor & CUTLASS -

<https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/S9593/>

cuBLAS - <https://docs.nvidia.com/cuda/cublas/index.html>

Mixed Precision Training Guide - <https://docs.nvidia.com/deeplearning/sdk/mixed-precision-training/index.html>

CUDA C Programming Guide (WMMA) - <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#wmma>

PTX ISA - <https://docs.nvidia.com/cuda/parallel-thread-execution/index.html>

CUDA Tensor Core Sample - <https://docs.nvidia.com/cuda/cuda-samples/index.html#cuda-tensor-core-gemm>