MORE SCIENCE WITH FEWER BITS

Julien Demouth & Mathias Wagner, Developer Technology

WHY MIXED PRECISION?

MORE SCIENCE

There are many reasons to consider mixed precision methods in HPC

- Accelerated hardware in current architectures
- Reduce memory traffic
- Reduce network traffic
- Reduce memory footprint
- Suitable numerical properties for the problem at hand

Accelerate or even *improve* the algorithm without compromising quality of science

WHY USE MIXED PRECISION ?

SPEED, STUPID!

Higher precision needs more memory and is slower

- Bandwidth on network and memory
- Higher precision is slower (usually 2x for simple ops, >2x for fancy math)

Which precision does my calculation require?

- Do all parts need high precision? Or maybe just accumulations / reductions ?

But if I just need high precision ?

- Accept it or maybe …?

MIXED PRECISION

for bandwidth bound application

---

for Deep Learning / AI

---

for compute bound applications

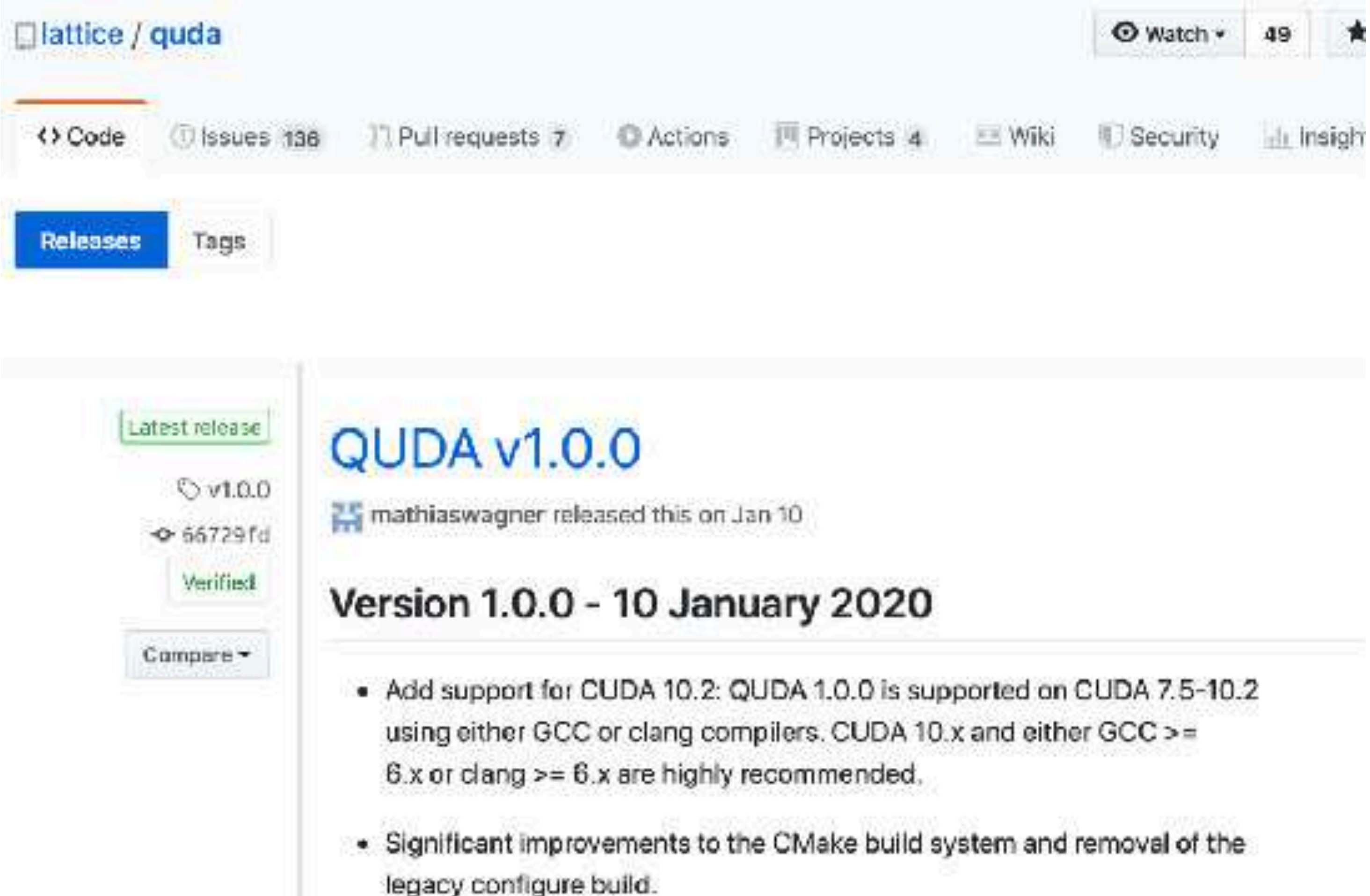
MIXED PRECISION FOR BANDWIDTH

QUDA

- Effort started at Boston University in 2008, now in wide use as the GPU backend for BQCD, Chroma, CPS, MILC, TIFR, tmLQCD, etc.
- Provides:
  - Various solvers for all major fermionic discretizations, with multi-GPU support
  - Additional performance-critical routines needed for gauge-field generation
- Maximize performance
  - Exploit physical symmetries to minimize memory traffic
  - Mixed-precision methods
  - Autotuning for high performance on all CUDA-capable architectures
  - Eigenvector and deflated solvers (Lanczos, EigCG, GMRES-DR)
  - Multigrid solvers for optimal convergence Multi-source solvers
  - Domain-decomposed (Schwarz) preconditioners for strong scaling
  - Strong-scaling improvements
- A research tool for how to reach the exascale

QUDA - LATTICE QCD ON GPUS

http://lattice.github.com/quda, BSD license



QUDA CONTRIBUTORS

10+ years - lots of contributors

Ron Babich (NVIDIA)
Simone Bacchio (Cyprus)
Michael Baldhauf (Regensburg)
Kip Barros (LANL)
Rich Brower (Boston University)
Nuno Cardoso (NCSA)
Kate Clark (NVIDIA)
Michael Cheng (Boston University)
Carleton DeTar (Utah University)
Justin Foley (Utah -> NIH)
Joel Giedt (Rensselaer Polytechnic Institute)
Arjun Gambhir (William and Mary)
Steve Gottlieb (Indiana University)
Kyriakos Hadjiyiannakou (Cyprus)

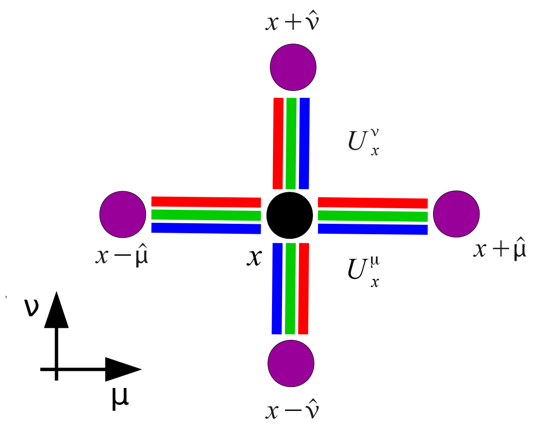
Dean Howarth (LLNL)
Bálint Joó (Jlab)
Hyung-Jin Kim (BNL -> Samsung)
Bartek Kostrzewa (Bonn)
Claudio Rebbi (Boston University)
Hauke Sandmeyer (Bielefeld)
Guochun Shi (NCSA -> Google)
Mario Schröck (INFN)
Alexei Strelchenko (FNAL)
Jiqun Tu (Columbia -> NVIDIA)
Alejandro Vaquero (Utah University)
Mathias Wagner (NVIDIA)
Evan Weinberg (NVIDIA)
Frank Winter (Jlab)

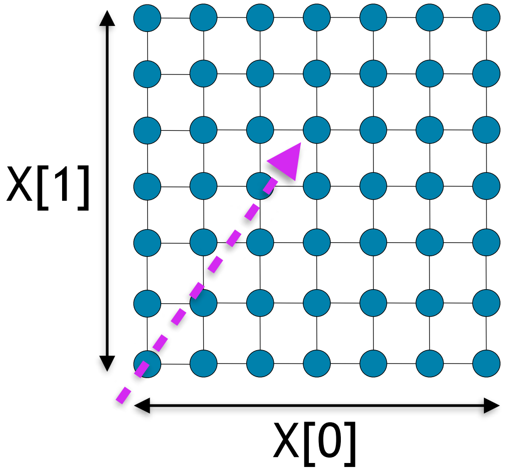
THE LATTICE QCD STENCIL (DSLASH)

Solve Ax=b

- Assign a single space-time point to each thread
  - V = XYZT threads, e.g., V = 24^4 => 3.3×10^6 threads
- Looping over direction each thread must
  - Load the neighboring spinor (24 numbers x8)
  - Load the color matrix connecting the sites (18 numbers x8)
  - Do the computation
  - Save the result (24 numbers)
- Each thread has (Wilson Dslash) 0.92 naive arithmetic intensity
- QUDA reduces memory traffic
  - Exact SU(3) matrix compression (18 => 12 or 8 real numbers)
  - Use 16-bit fixed-point representation with mixed-precision solver

$$D_{x,x'} =$$





QUDA'S 16-BIT FIXED-POINT FORMAT

In production since 2009

Link field - Defines the sparse matrix elements

- SU(3) matrices that live between all adjacent sites on the 4-d grid
- All elements $\in [-1,1]$ => very natural to use 16-bit fixed point representation

Fermion field - the vector that appears in the linear solver

- Each 4-d grid point consists of a 12-component complex vector
- No a priori bounds the elements
- Use per-site L_{inf} norm to normalize the site vector and use 16-bit fixed point
- Optimal use of precision: retains global dynamic range with local 16-bit mantissa

Low precision used only as a storage type with computation done in FP32

LINEAR SOLVERS

QCD dominate by sparse Ax=b

LQCD requires a range of sparse iterative linear solvers

- CG, BiCGstab, GCR, Multi-shift solvers, etc.

Condition number inversely proportional to mass

- Light (realistic) masses are highly singular
- Naive Krylov solvers suffer from critical slowing down at decreasing mass

Entire solver algorithm must run on GPUs

- Time-critical kernel is the stencil application Also require BLAS level-1 type operations

$$\text{while } (|r_k| > \varepsilon)\ \{$$
$$\beta_k = (r_k, r_k)/(r_{k-1}, r_{k-1})$$
$$p_{k+1} = r_k - \beta_k p_k$$
$$q_{k+1} = A\, p_{k+1}$$
$$\alpha = (r_k, r_k)/(p_{k+1}, q_{k+1})$$
$$r_{k+1} = r_k - \alpha q_{k+1}$$
$$x_{k+1} = x_k + \alpha p_{k+1}$$
$$k = k+1$$
$$\}$$

conjugate gradient

*Sleijpen and Van der Worst, 1996

Clark, Babich, Barros, Brower, Rebbi (2009)

RELIABLE UPDATES FOR MIXED PRECISION

Traditional approach to mixed precision is to use iterative refinement

- Disadvantage: each restart means we discard the Krylov space

Instead we use reliable updates*

- As low-precision solver progresses the iterated residual will drift
- Occasionally replace the iterated residual with high-precision residual
- Retains Krylov space information
- Maintain a separate partial-solution accumulator

Aside: reductions are always done in fp64 regardless of the data precision

```
while (|r_k| > ε) {
    r_k = b - Ax_k
    solve Ap_k = r_k
    x_k+1 = x_k + p_k
}
```

```
if (|r_k| < δ|b|) {
    r_k = b - Ax_k
    b = r_k
    y = y + x_k
    x_k = 0
}
```

(STABLE) MIXED-PRECISION CG

Three key ingredients

CG convergence relies on gradient vector being orthogonal to residual vector

Re-project when injecting new residual (Strzodka and Gödekke, 2006)

Precision is lost if we keep the partial solution vector in low precision

Always keep the (partial) solution vectors in high precision

β computation relies on $(r_i, r_j) = |r_i|^2 \delta_{i,j}$

Not true in finite precision

Polak-Ribière form is equivalent and self-stabilizing

$$\beta_k = \frac{(r_k, (r_k - r_{k-1}))}{|r_{k-1}|^2}$$

LINEAR SOLVERS

QUDA supports a wide range of linear solvers

- CG, BiCGstab, GCR, Multi-shift solvers, etc.

Condition number inversely proportional to mass

- Light (realistic) masses are highly singular
- Naive Krylov solvers suffer from critical slowing down at decreasing mass

Entire solver algorithm must run on GPUs

- Time-critical kernel is the stencil application
- Also require BLAS level-1 type operations

while ($|\mathbf{r}_k| > \varepsilon$) {
$\beta_k = (\mathbf{r}_k,\mathbf{r}_k)/(\mathbf{r}_{k-1},\mathbf{r}_{k-1})$
$\mathbf{p}_{k+1} = \mathbf{r}_k - \beta_k\mathbf{p}_k$
$\mathbf{q}_{k+1} = A\,\mathbf{p}_{k+1}$
$\alpha = (\mathbf{r}_k,\mathbf{r}_k)/(\mathbf{p}_{k+1},\mathbf{q}_{k+1})$
$\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha\mathbf{q}_{k+1}$
$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha\mathbf{p}_{k+1}$
$k = k+1$
}

conjugate gradient

MIXED-PRECISION CG

Apply Dslash in sloppy precision (single, half)

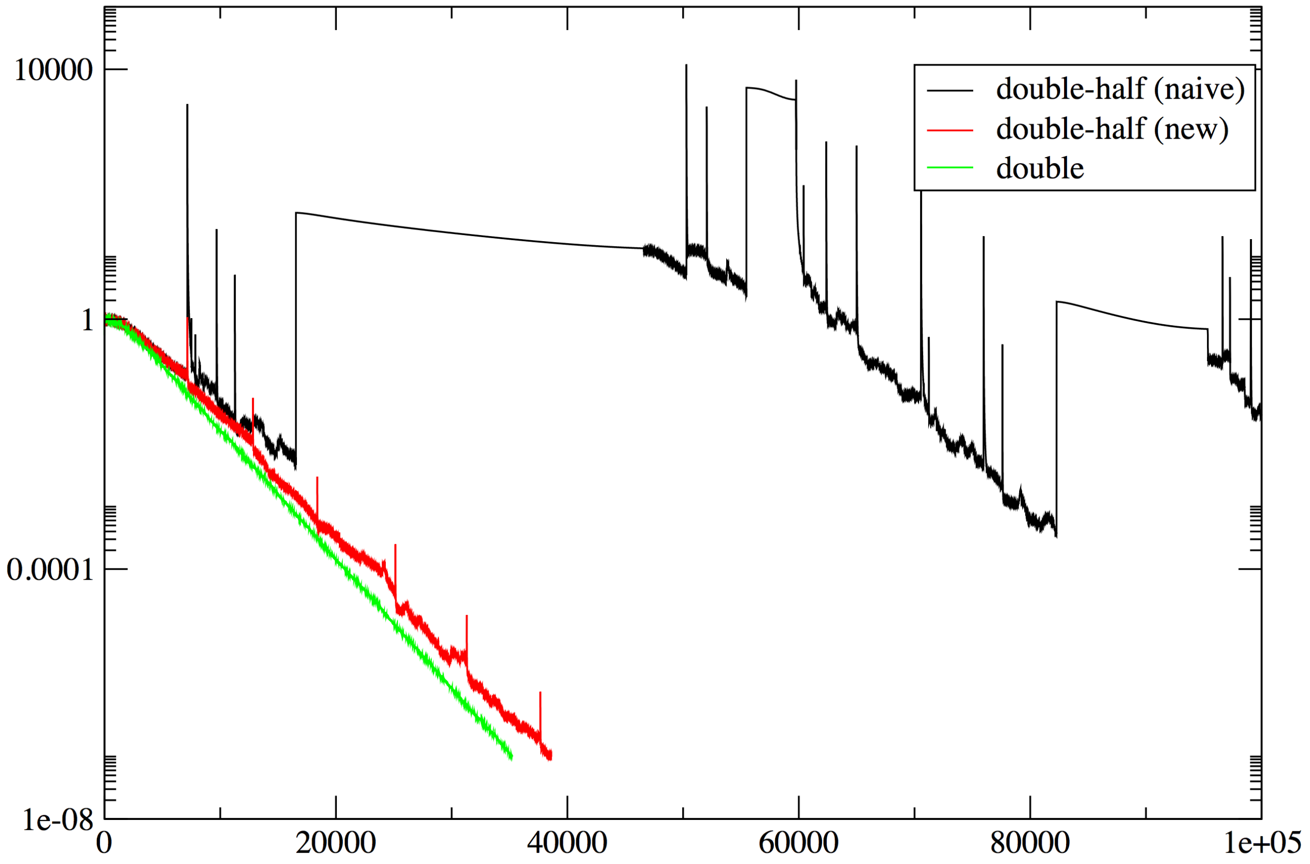
Reliable residual replacement in high precision

Ensures accuracy of final result

Half-precision **storage**:

- Stencil elements ∈ [-1,1] (Link):
  - 16-bit fixed point
- Grid elements (Spinor):
  - 16-bit fixed point (24 numbers)
  - float (exponent, 1 number)

Use fp32 for actual arithmetics



MIXED-PRECISION CG

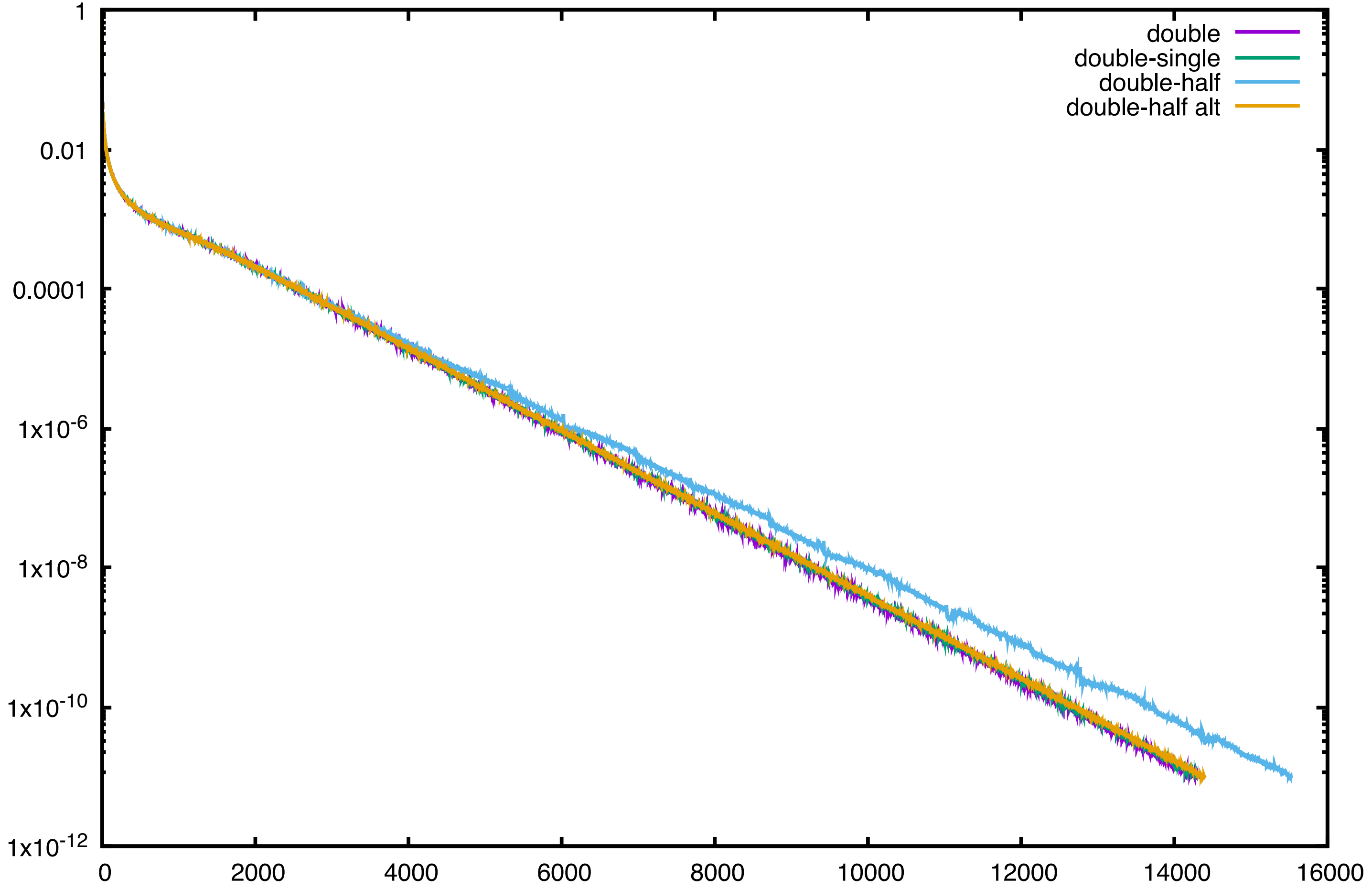
double-half

- Maintain solution vectors in high precision
  - Including the partial accumulator
- When true residual is injected, re-project the direction vector
- Use Polak-Ribière formula

$$\beta_k := \frac{\mathbf{z}_{k+1}^T (\mathbf{r}_{k+1} - \mathbf{r}_k)}{\mathbf{z}_k^T \mathbf{r}_k}$$

double-half alt

- Residual replacement strategy of van der Worst and Ye



MIXED-PRECISION CG

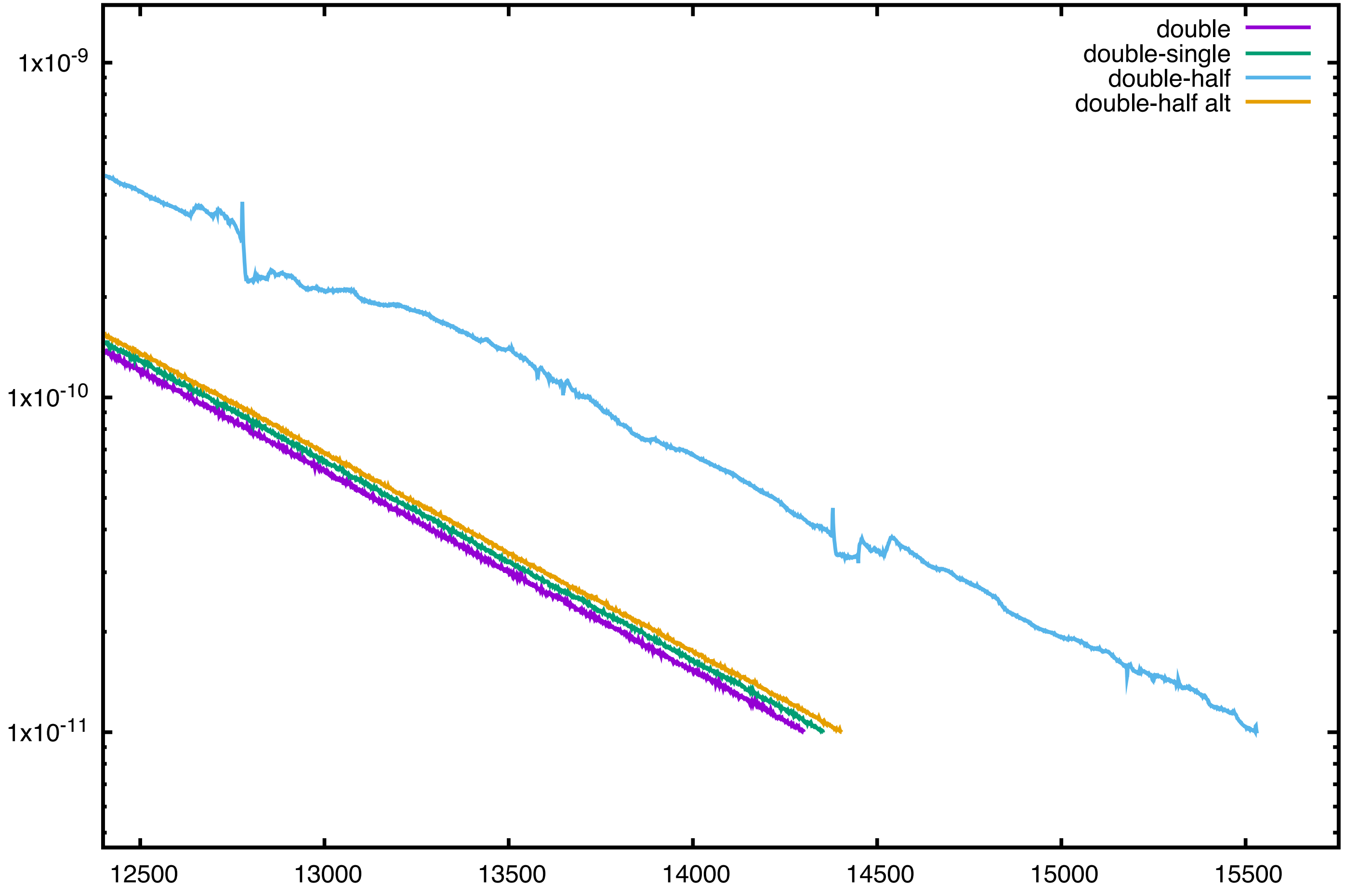
double-half

- Maintain solution vectors in high precision
  - Including the partial accumulator
- When true residual is injected, re-project the direction vector
- Use Polak-Ribière formula

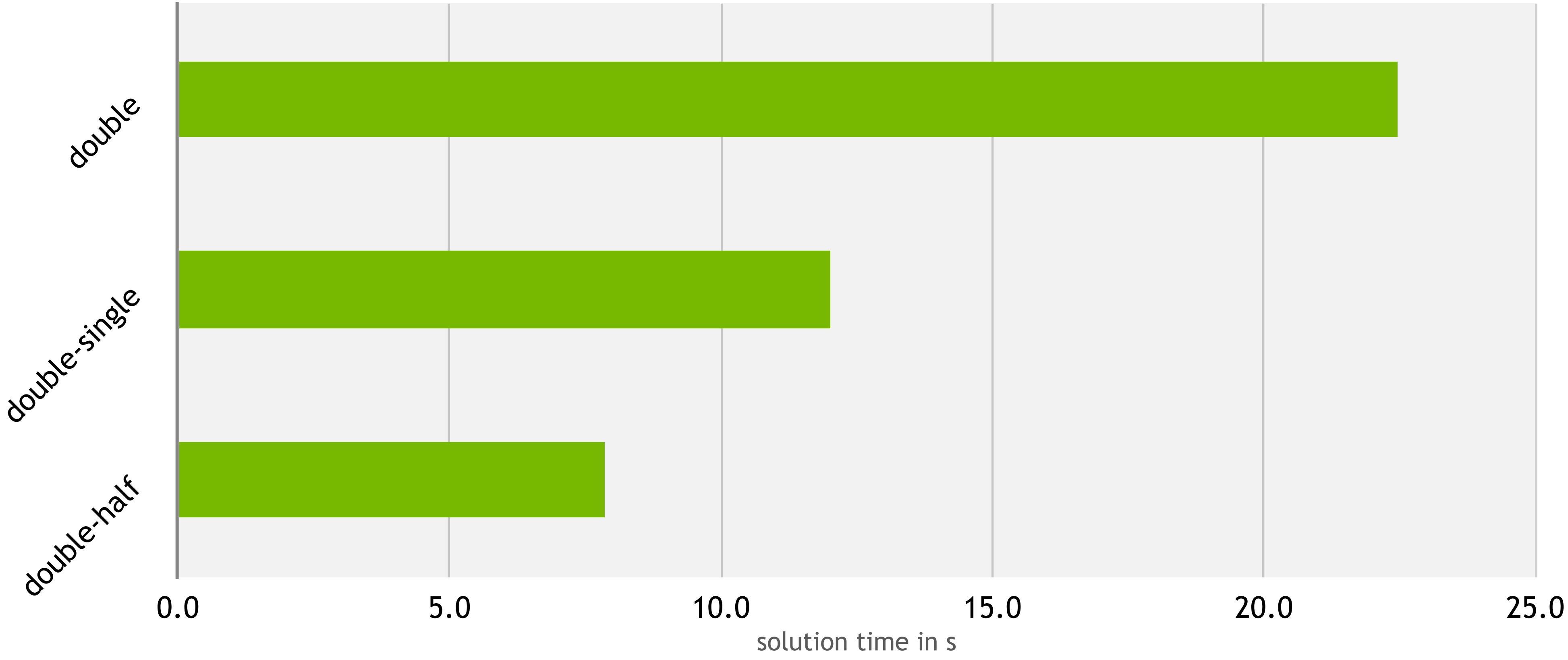
$$\beta_k := \frac{\mathbf{z}_{k+1}^{\mathsf{T}} \left(\mathbf{r}_{k+1} - \mathbf{r}_k\right)}{\mathbf{z}_k^{\mathsf{T}} \mathbf{r}_k}$$

double-half alt

- Residual replacement strategy of van der Worst and Ye

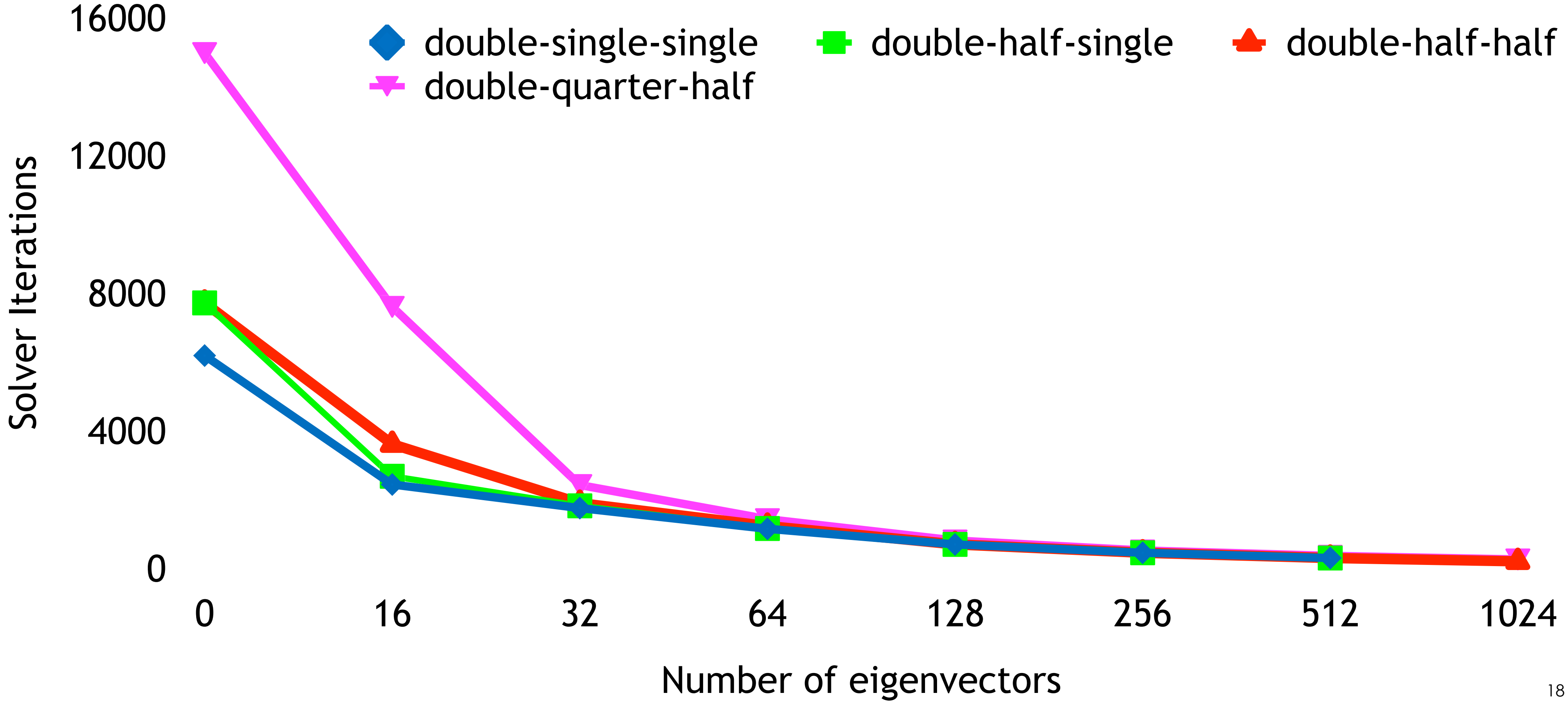


MIXED-PRECISION MILC CG SOLVER



DEFLATION STABILIZES LOW PRECISION

$V=48^3x12$, HISQ operator, physical light quarks, tol 10^{-10}, 2xV100

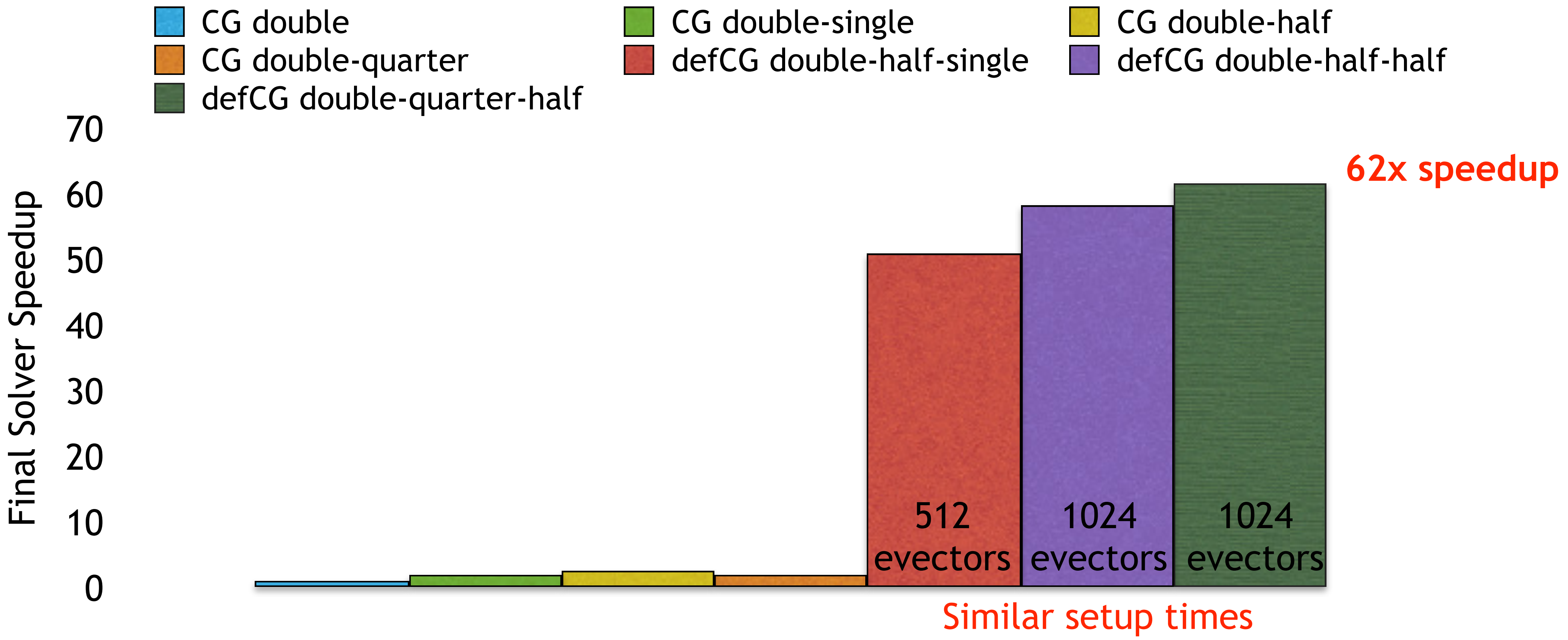


NVIDIA

Configuration provided by
HotQCD collaboration (Mukherjee *et al*)

MIXED-PRECISION DEFLATION

$V=48^3x12$, HISQ operator, physical light quarks, tol 10^{-10}, 2xV100



DEEP LEARNING AND MIXED PRECISION

Julien Demouth

REFERENCE

This slide deck was built from two presentations at GTC 2020

Training Neural Networks with Tensor Core [S22082], Dusan Stosic

Accelerating Sparsity in the NVIDIA Ampere Architecture [S22085], Jeff Pool

The presentations are available from https://www.nvidia.com/en-us/gtc/on-demand/

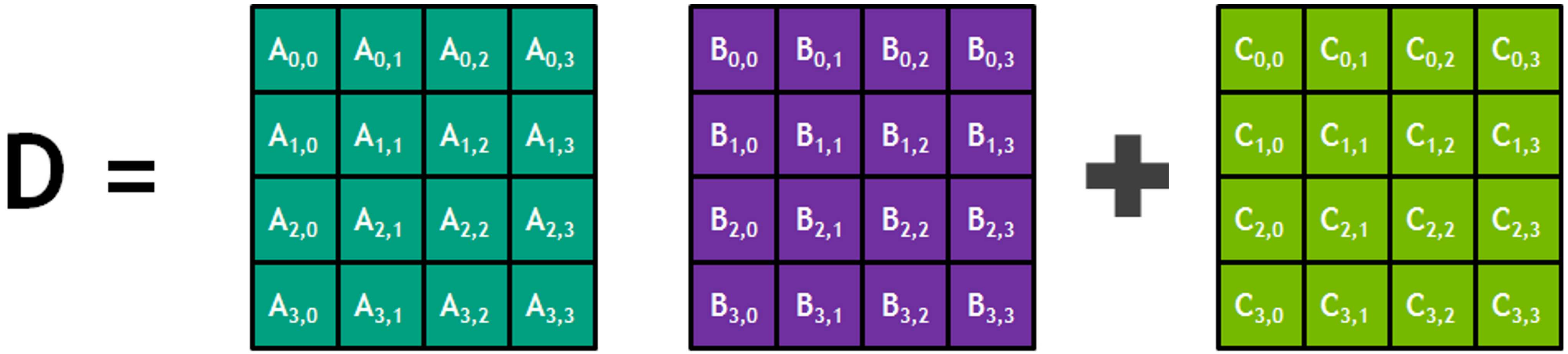
[QUICK REMINDER] TENSOR CORES

Specialized hardware execution units for performing matrix and convolution operations

Compared to scalar FP32 operations, Tensor Cores are:

- 8-16x faster (up to 32x faster with sparsity)
- More energy efficient

D = AB + C



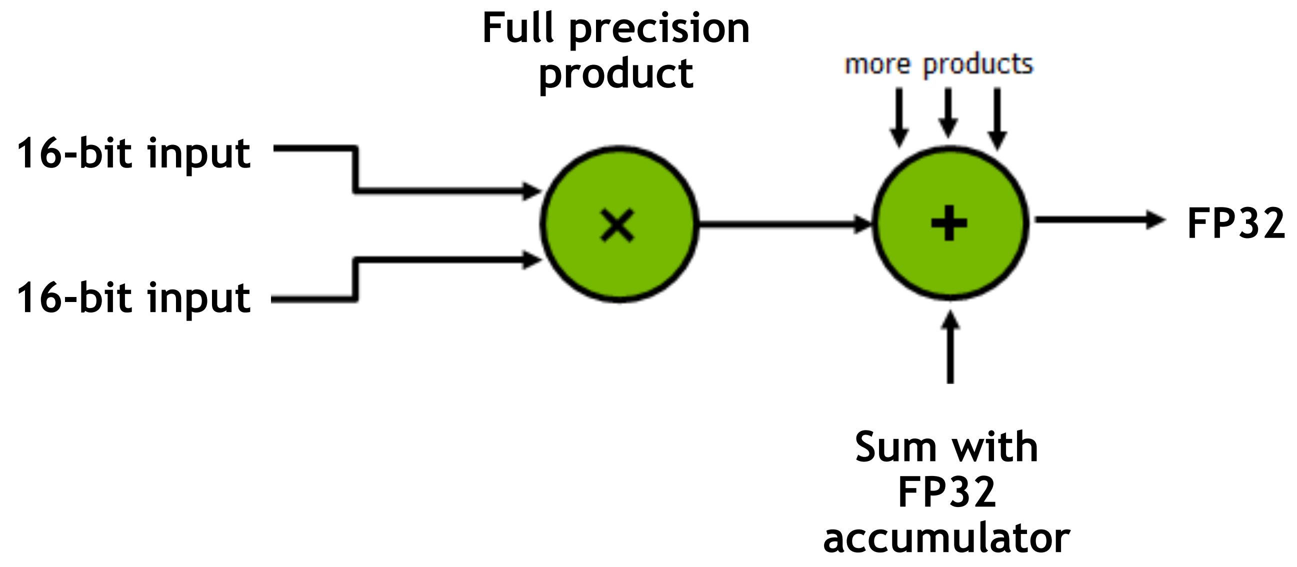
[QUICK REMINDER] TENSOR CORES FOR 16-BIT FORMATS

Operation:

Multiply and add FP16 or BF16 tensors

Products are computed without loss of precision, accumulated in FP32

Final FP32 output is rounded to FP16 or BF16 before writing to memory



NVIDIA Ampere Architecture enhancements:

New tensor core design: 2.5x throughput for dense operations (A100 vs V100)

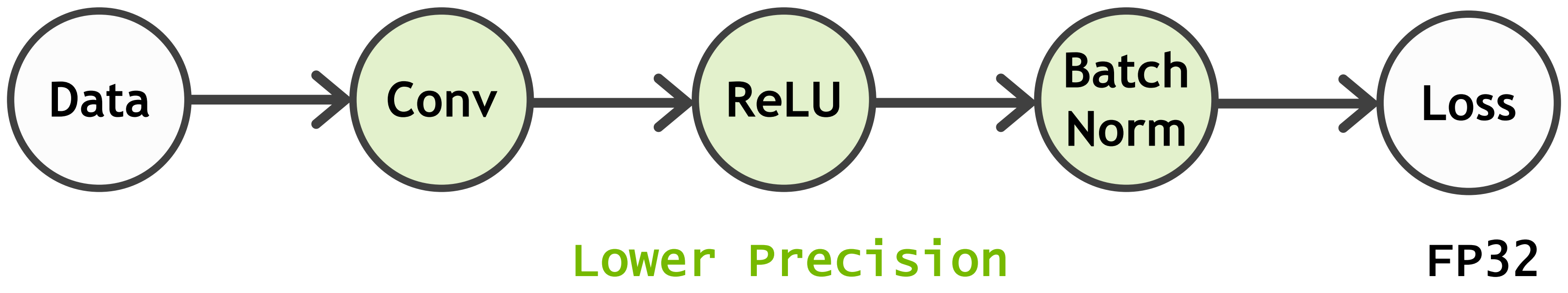
Sparsity support: additional 2x throughput for sparse operations

BFloat16 (BF16): Same rate as FP16

MIXED PRECISION TRAINING

Combines single-precision (FP32) with lower precision (e.g. FP16) when training a network

Achieves the same accuracy as FP32 training, uses all the same hyper-parameters



Benefits:

- Accelerates math-intensive operations with specialized hardware (GPU Tensor Cores)
- Accelerates memory-intensive operations by reducing memory traffic (16-bit; not tf32)
- Reduce memory requirements, enables training of larger models, larger minibatches, larger inputs (16-bit; not tf32)

MIXED PRECISION IS GENERAL PURPOSE

3 years of networks trained with 16-bit formats

Proven to match FP32 results across a wide range of **tasks, problem domains, deep neural network architectures**

Image Classification
- AlexNet
- DenseNet
- Inception
- MobileNet
- EfficientNet
- ResNet
- ResNeXt
- ShuffleNet
- SqueezeNet
- VGG
- Xception

Detection / Segmentation
- DeepLab
- Faster R-CNN
- Mask R-CNN
- SSD
- NVIDIA Automotive
- RetinaNet
- UNET

Recommendation
- DeepRecommender
- NCF

Generative Models (Images)
- DLSS
- GauGAN
- Partial Image Inpainting
- Progress GAN
- Pix2Pix

Speech
- Deep Speech 2
- Jasper
- Tacotron
- Wave2vec
- WaveNet
- WaveGlow

Language Modeling
- BERT
- TrellisNet
- Gated Convolutions
- BigLSTM/mLSTM
- RoBERTa
- Transformer XL

Translation
- Convolutional Seq2Seq
- Dynamic Convolutions
- GNMT (RNN)
- Levenshtein Transformer
- Transformer (Self-Attention)

The chart only represents a small sampling of networks trained in mixed precision

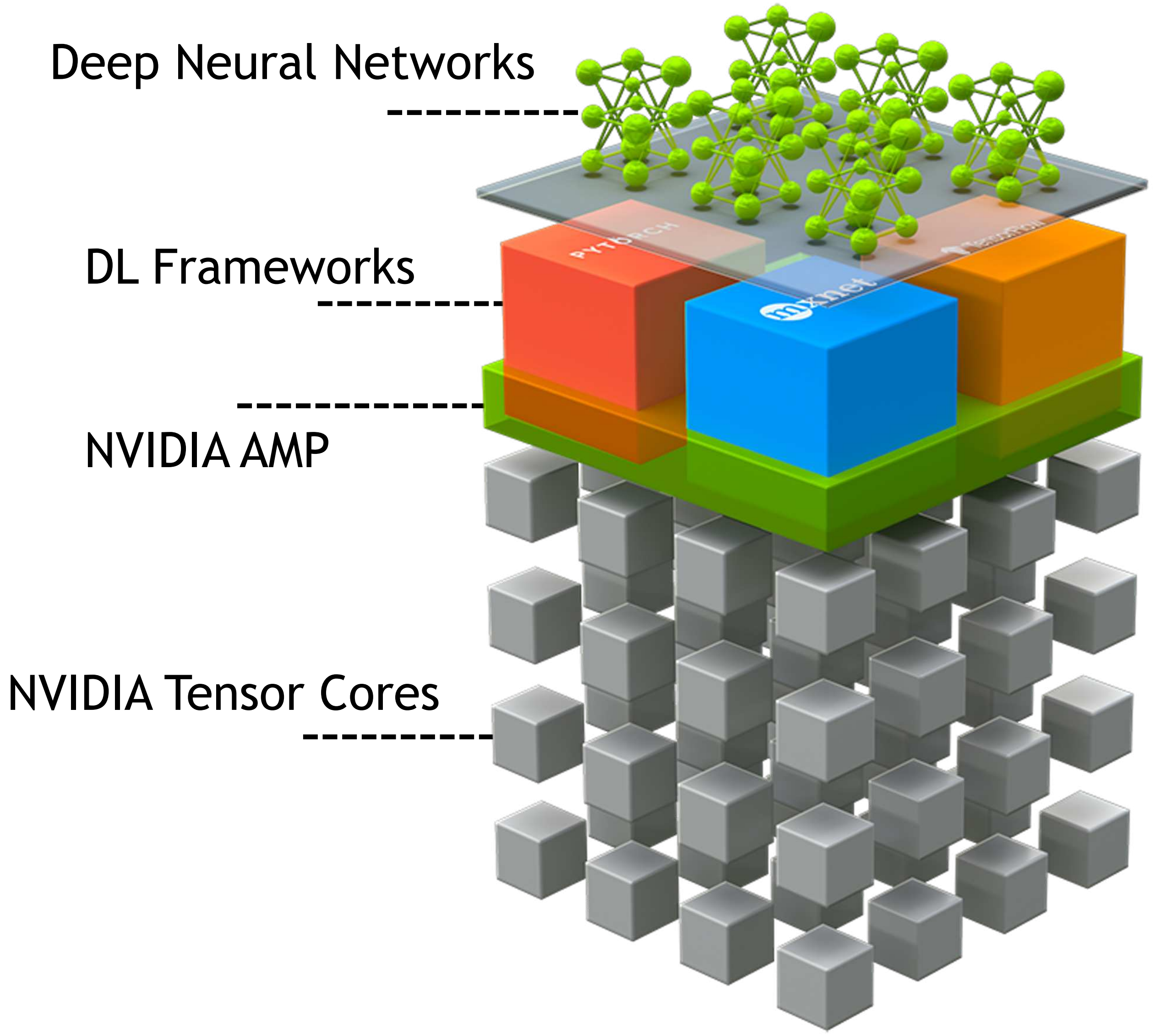
AUTOMATIC MIXED PRECISION FOR 16-BITS

Automatic Mixed Precision (AMP) makes mixed precision training with FP16/BF16 easy in frameworks

- AMP automates process of training in mixed precision
- e.g. Converts matrix multiplies/convolutions to 16-bits for Tensor Core acceleration

Works with multiple models, optimizers, and losses

BF16 will be available in future releases



NATIVE AMP FOR PYTORCH

Merged into master end of April

Will be available in future NVIDIA NGC containers

Proven to work on ~40 deep neural network workloads

NVIDIA Deep Learning Examples have used PyTorch APEX AMP for over a year and will soon update all models to PyTorch Native AMP

```
import torch

# Creates once at the beginning of training
scaler = torch.cuda.amp.GradScaler()

for data, label in data_iter:
    optimizer.zero_grad()

    # Casts operations to mixed precision
    with torch.cuda.amp.autocast():
        loss = model(data)

    # Scales the loss, and calls backward()
    # to create scaled gradients
    scaler.scale(loss).backward()

    # Unscales gradients and calls
    # or skips optimizer.step()
    scaler.step(optimizer)

    # Updates the scale for next iteration
    scaler.update()
```

FEW CODE CHANGES TO ENABLE AMP IN FRAMEWORKS

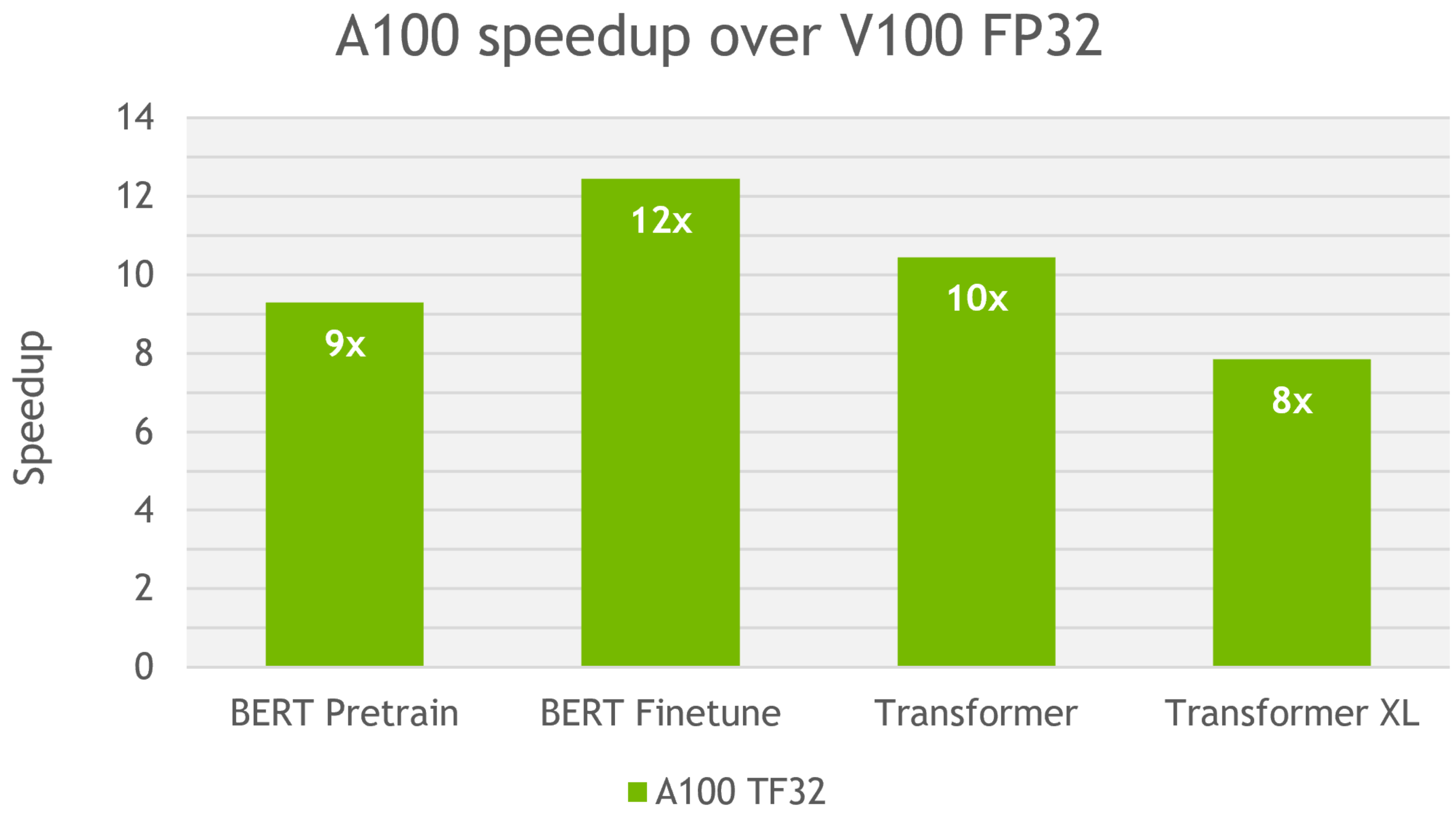
TensorFlow	NVIDIA NGC Container 19.07+, TF 1.14+ and TF 2+, explicit optimizer wrapper available:
	
	`opt = tf.train.experimental.enable_mixed_precision_graph_rewrite(opt)`
	`Keras mixed precision API in TF 2.1+ for eager execution`
	
	https://tensorflow.org/api_docs/python/tf/train/experimental/enable_mixed_precision_graph_rewrite

PyTorch	Native support in PT, see official docs for usage: https://pytorch.org/docs/stable/amp.html https://pytorch.org/docs/stable/notes/amp_examples.html

MXNet	NVIDIA NGC Container 19.04+, MXNet 1.5+, few lines of code:
	
	`amp.init()`
	`amp.init_trainer(trainer)`
	`with amp.scale_loss(loss, trainer) as scaled_loss:`
	`    autograd.backward(scaled_loss)`
	
	https://mxnet.apache.org/api/python/docs/tutorials/performance/backend/amp.html

SAMPLE OF ACHIEVED TRAINING SPEEDUPS

Mixed precision training on A100 is up to 12x faster than V100 FP32



1 month to train on Volta using FP32 → 2-3 days on A100 using next-gen Tensor Cores

INTRODUCING TENSOR FLOAT 32 (TF32)

DL TRAINING OPTIONS

FP16 and BF16 Tensor Cores

Best choice for performance

Both are well established formats with proven success across a wide breadth of AI networks

Does require model changes (FP32 weight storage, loss scaling, per-layer precision choices)

Automatic Mixed Precision (AMP) makes it easy

TF32 Tensor Cores

New default for A100 – no model changes required

10x peak rate of Volta FP32 (but ½ of peak rates of FP16/BF16)

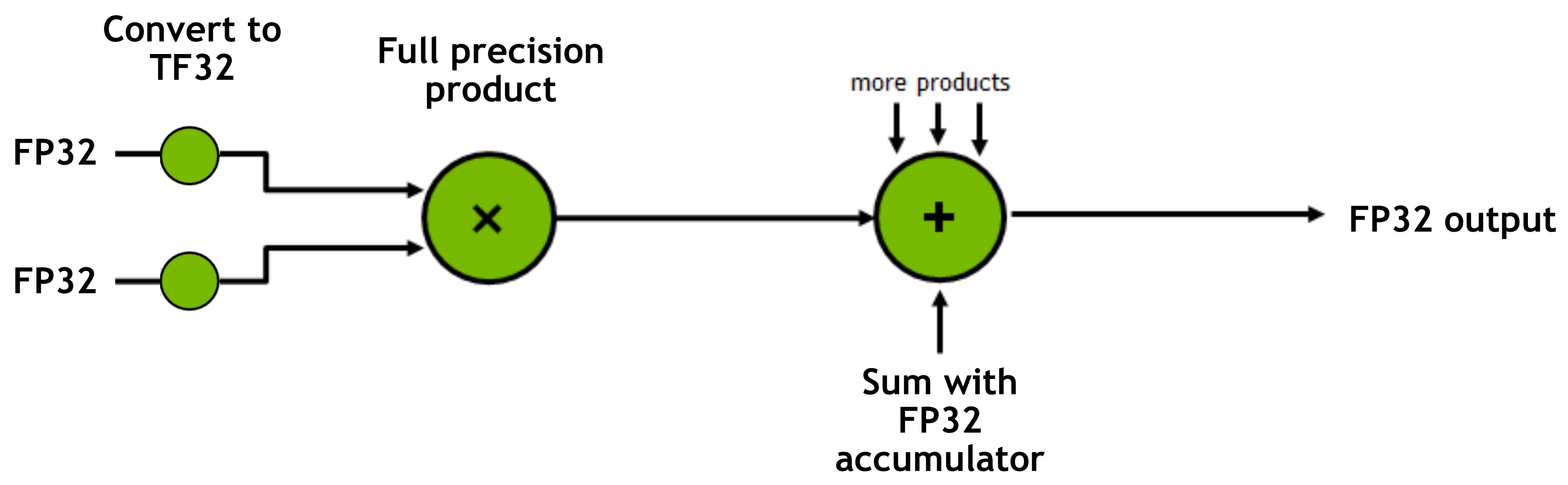
FP32 – non-Tensor Core

Default for Volta (on A100 it is 1/16 of peak rate of FP16, 1/8 of peak of TF32)

[QUICK REMINDER] TENSOR FLOAT 32

A Tensor Core math mode for single-precision training

- Multiply and add of FP32 tensors
- Tensor Core inputs are rounded to TF32
- Products are computed without loss of precision, accumulated in FP32



TF32 DETAILS

8-bit exponent:

Matches FP32, covers the same range of values

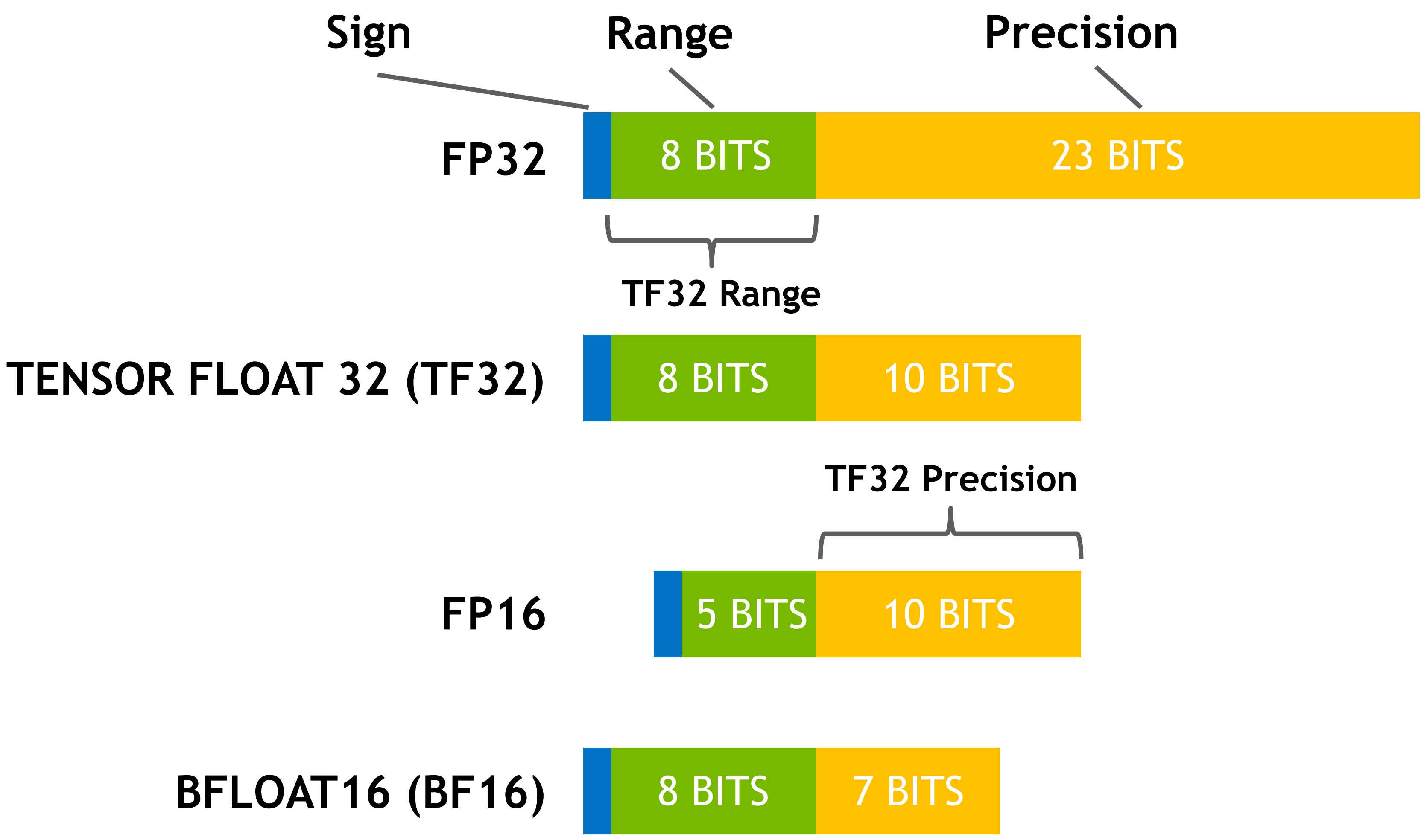
10-bit mantissa:

Higher precision than BF16

The only difference from FP32

TF32 will match FP32 results for any network trained with FP16 or BF16 mixed precision

Shown to have sufficient margin for DL training by networks trained in 16-bits over the past 3 years



TF32 VERIFICATION

Further verification based on unmodified model scripts for 80+ networks

- Model architectures: Convnets, MLPs, RNNs, Transformers, BERT, GANs, etc.
- Various tasks, including:
  - image tasks (classification, detection, segmentation, generation, gaze)
  - language tasks (translation, modeling, question answering)
  - Recommenders
  - Meta learning
  - More niche tasks (logic reasoning, combinatorial problems)
- First and second order methods

Matches FP32 accuracy and loss values

SINGLE PRECISION TRAINING WITH TF32

Default mode for A100 in next release of NVIDA NGC containers

- Supported frameworks: TensorFlow, PyTorch, MXNet
- Operation:
  - TF32 acceleration is enabled for single-precision convolution and matrix-multiply layers:
    - Including linear/fully-connected layers, recurrent cells, attention blocks
  - TF32 acceleration is not enabled for:
    - Convolutions or matrix-multiply layers that operate on non-FP32 tensors
    - Any layers that are not convolutions or matrix-multiplies
    - Optimizer/solver operations
  - No tensor storage is changed – remains in FP32 (or whichever format is specified in the script)

Support in mainline frameworks coming soon

GLOBAL PLATFORM CONTROL FOR TF32

Global variable **NVIDIA_TF32_OVERRIDE** to toggle TF32 mode at system level (and override libraries/frameworks)

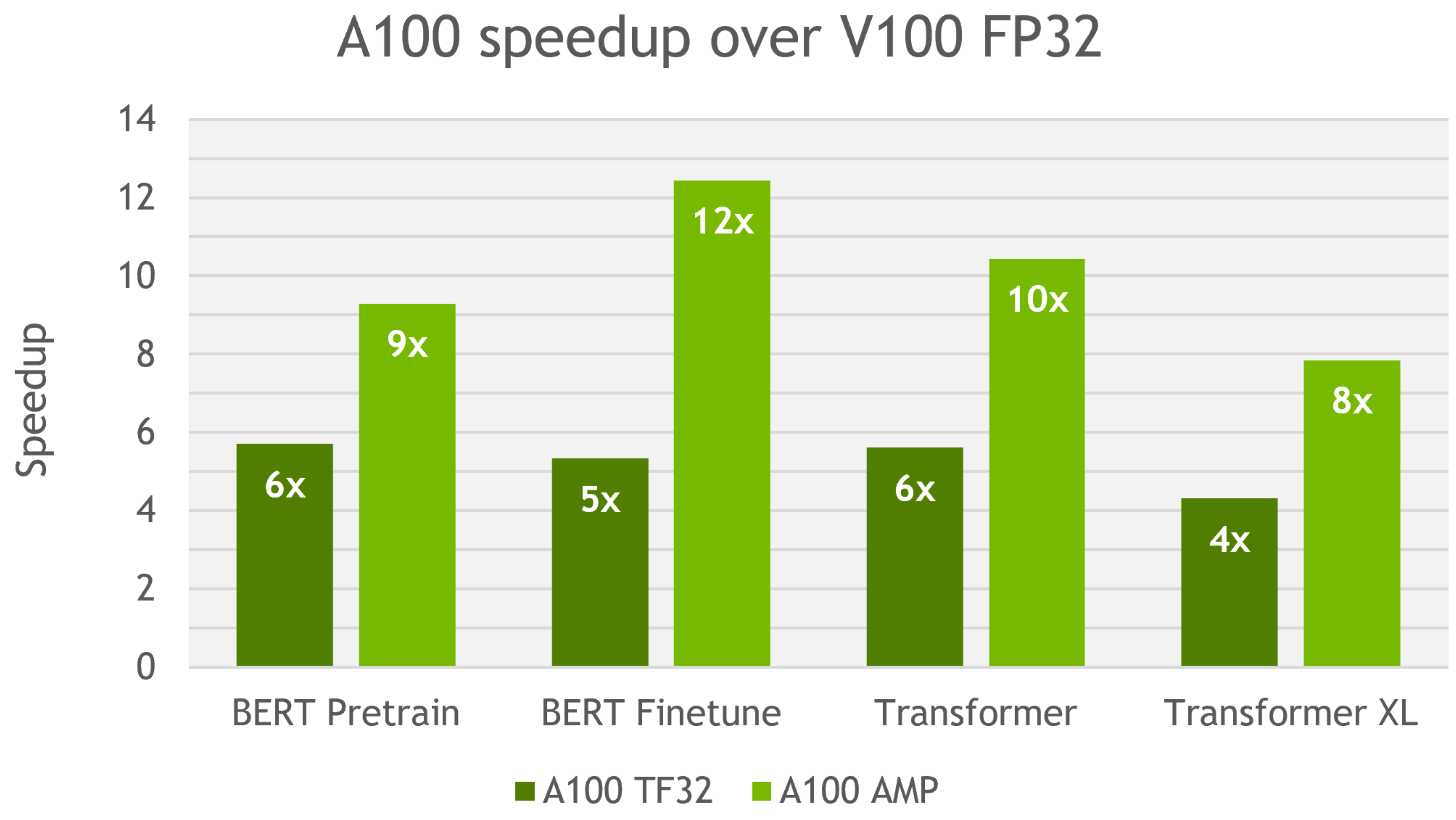
NVIDIA_TF32_OVERRIDE=0	Not Set
Disables TF32 so that FP32 is used	Defaults to library and framework settings

Debugging tool - quick way to rule out any concern regarding TF32 libraries and look for other issues

SAMPLE OF ACHIEVED TRAINING SPEEDUPS

A100 single precision training is up to 5x faster because of TF32 acceleration

A100 mixed precision gives an additional 2x



CHOOSING TRAINING OPTIONS ON A100

Mixed-precision with FP16 or BF16:

Option to use if you:

- Use mixed-precision training (FP16 or BF16) on Volta and other processors
- Are using single-precision on A100 training and want further speedup

Fastest options for training: up to 2x faster than single-precision with TF32

Requires minimal additions to training scripts with AMP (detailed in previous sections)

Single-precision with TF32:

Great starting point if you used FP32 training on Volta and other processors

Default math mode for AI, does not require changes to training scripts

Uses Tensor Cores (10X over Volta default)

SPARSITY SUPPORT INTRODUCED IN NVIDIA AMPERE ARCHITECTURE

SPARSITY: ONE OF MANY OPTIMIZATION TECHNIQUES

Optimization goals for inference:

- Reduce network model size
- Speed up network model execution

Observations that inspire sparsity investigations

- Biology: neurons are not densely connected
- Neural networks:
  - Trained model weights have many small-magnitude values
  - Activations may have 0s because of ReLU

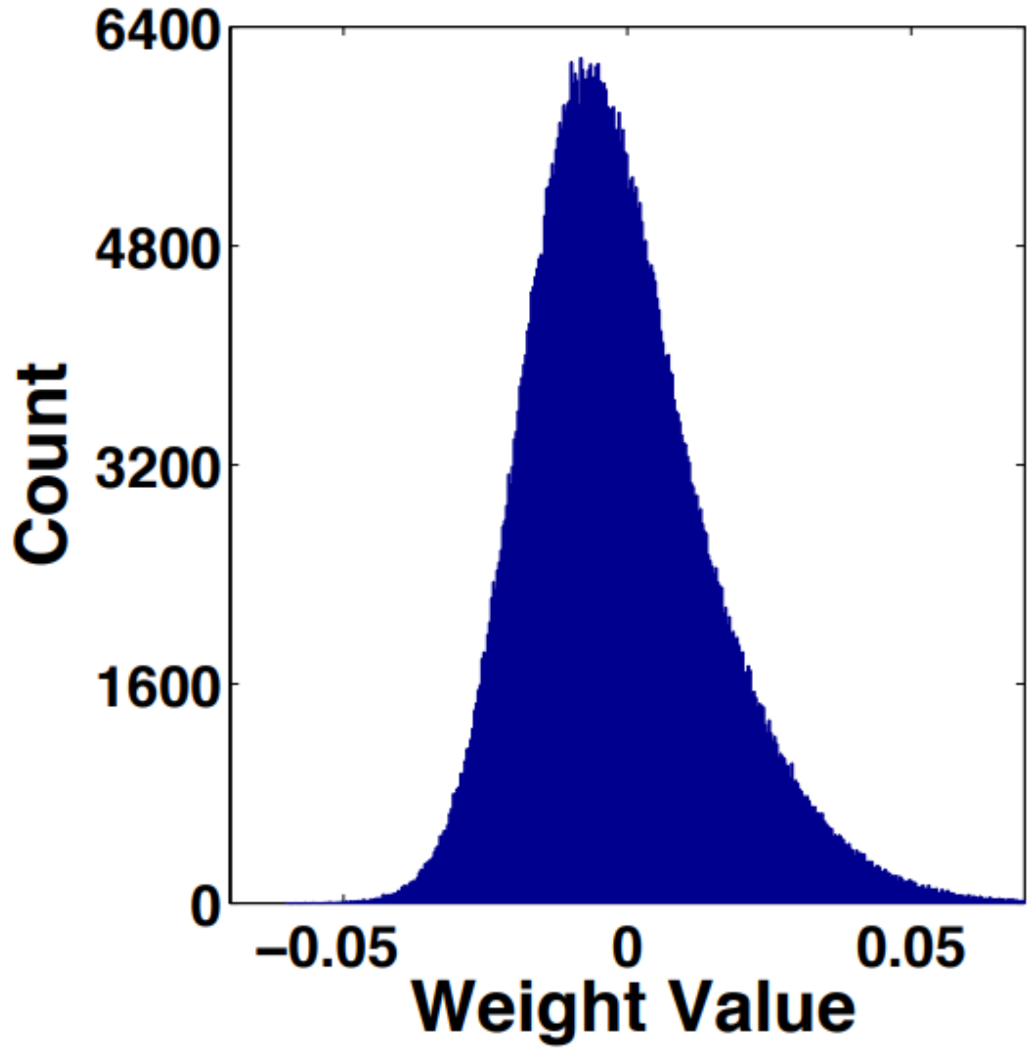


Figure: “DSD: Dense-Sparse-Dense Training for Deep Neural Networks” S. Han et al.

SPARSITY AND PERFORMANCE

Do not store or process 0 values -> smaller and hopefully faster model

- Eliminate (prune) connections: set some weights to 0
- Eliminate (prune) neurons
- Etc.

But, must also:

- Maintain model accuracy
- Efficiently execute on hardware to gain speedup

SPARSITY TAXONOMY

Structure:

Unstructured: irregular, no pattern of zeros

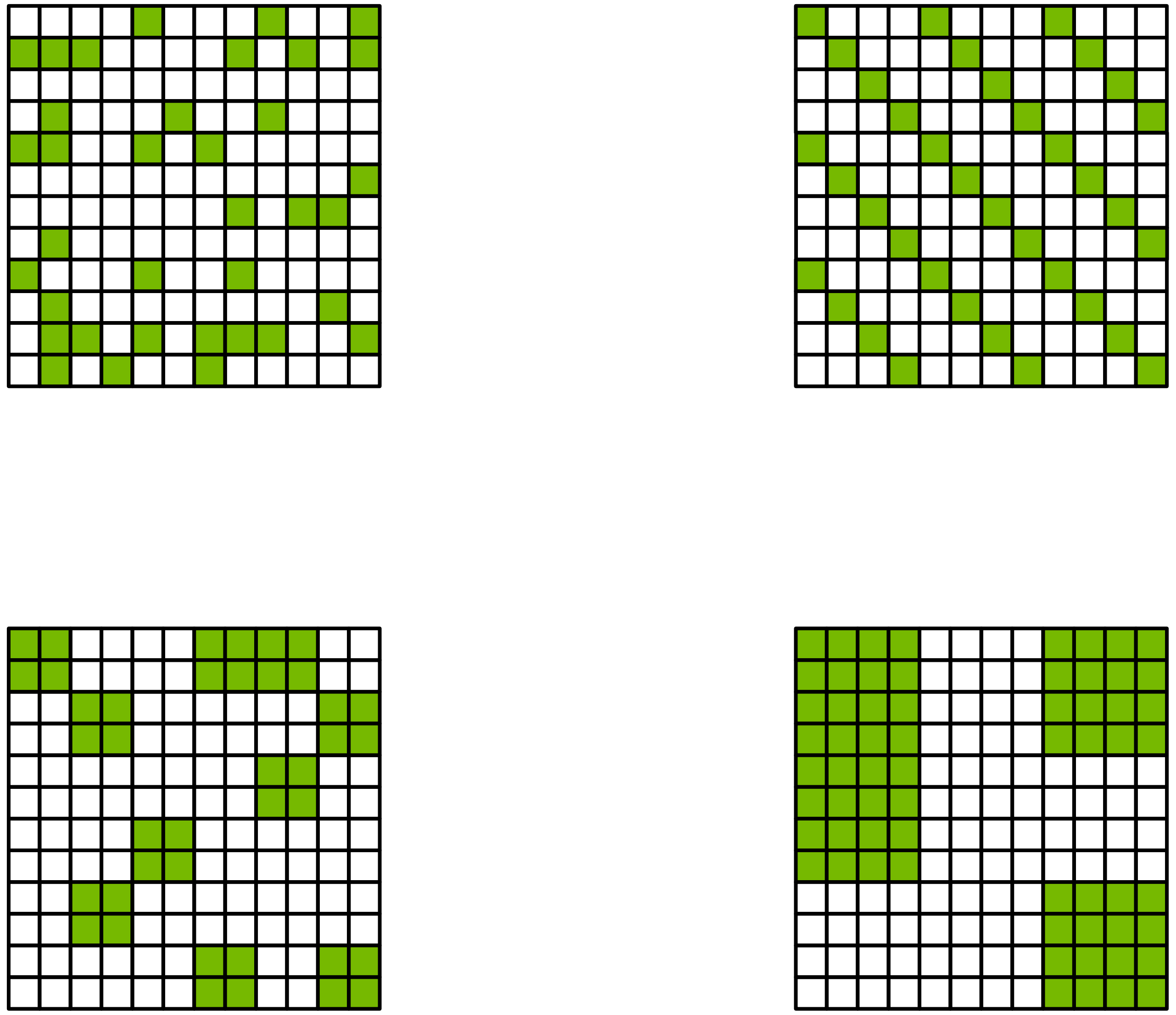
Structured: regular, fixed set of patterns to choose from

Granularity:

Finest: prune individual values

Coarser: prune blocks of values

Coarsest: prune entire layers



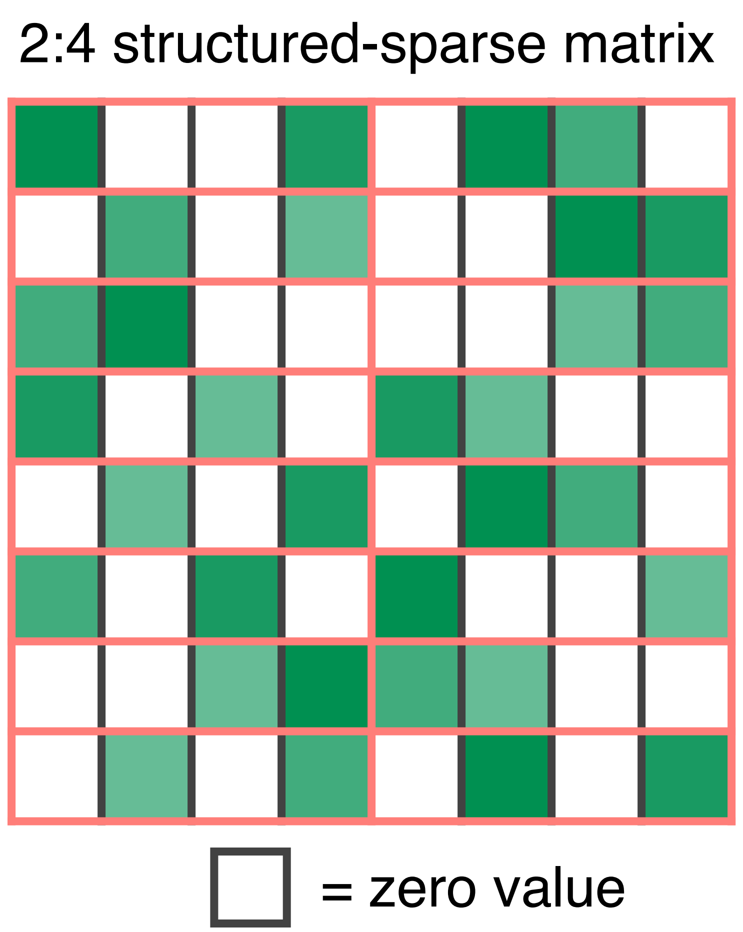
SPARSITY IN A100 GPU

Fine-grained structured sparsity for Tensor Cores

- 50% fine-grained sparsity
- 2:4 pattern: 2 values out of each contiguous block of 4 must be 0

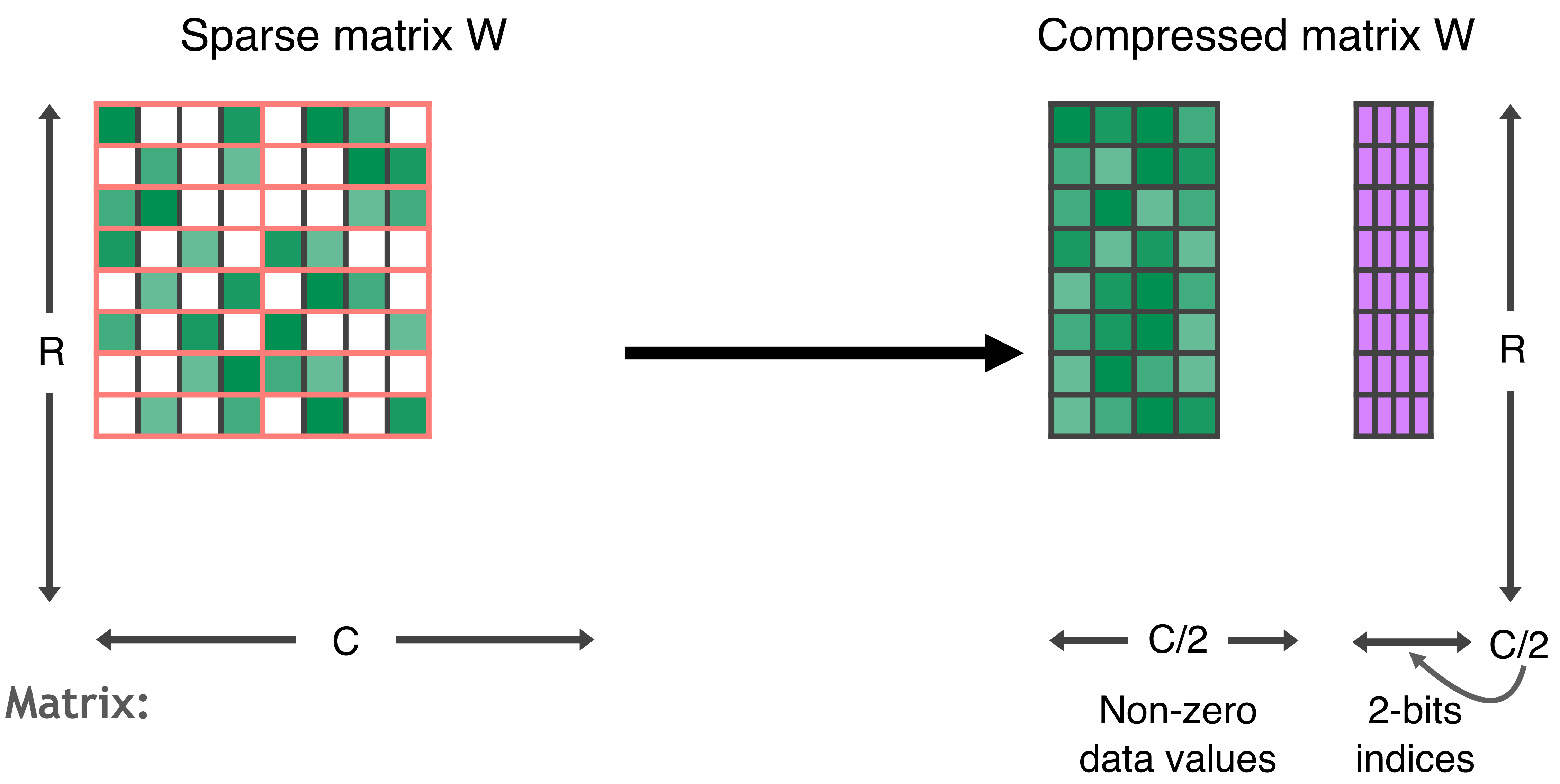
Addresses the 3 challenges:

- Accuracy: maintains accuracy of the original, unpruned network
  - Medium sparsity level (50%), fine-grained
- Training: a recipe shown to work across tasks and networks
- Speedup:
  - Specialized Tensor Core support for sparse math
  - Structured: lends itself to efficient memory utilization



2:4 COMPRESSED MATRIX FORMAT

At most 2 non-zeros in every contiguous group of 4 values



Compressed Matrix:

Data: ½ size

Metadata: 2b per non-zero element

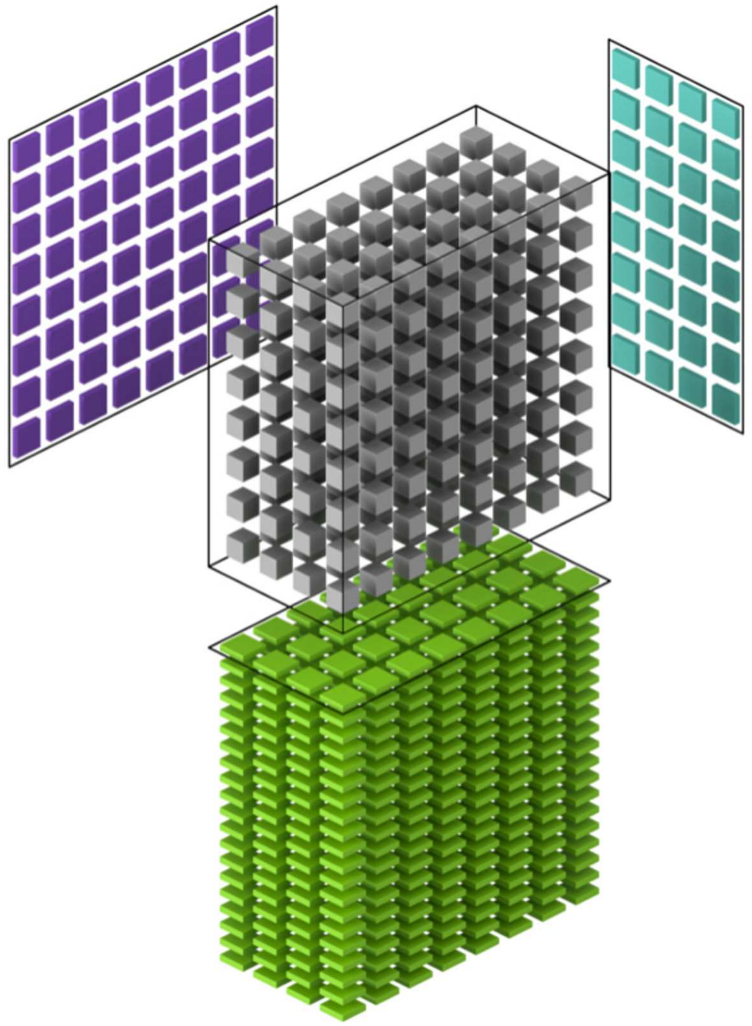
16b data => 12.5% overhead

8b data => 25% overhead

TENSOR CORE MATH THROUGHPUT

2x with Sparsity

INPUT OPERANDS	ACCUMULATOR	TOPS	Dense vs. FFMA	Sparse Vs. FFMA
FP32	FP32	19.5	-	-
TF32	FP32	156	8X	16X
FP16	FP32	312	16X	32X
BF16	FP32	312	16X	32X
FP16	FP16	312	16X	32X
INT8	INT32	624	32X	64X
INT4	INT32	1248	64X	128X
BINARY	INT32	4992	256X	-



SPARSE TENSOR CORES

Measured GEMM Performance with Current Software

M	N	K	Speedup
1024	8192	1024	1.44x
1024	16384	1024	1.73x
4096	8192	1024	1.53x
4096	16384	1024	1.78x

GEMM sizes selected from BERT-Large

SPARSE TENSOR CORES

Measured Convolution Performance With Current Software

N	C	K	H,W	R,S	Speedup
32	1024	2048	14	1	1.52x
32	2048	1024	14	1	1.77x
32	2048	4096	7	1	1.64x
32	4096	2048	7	1	1.75x
256	256	512	7	3	1.85x

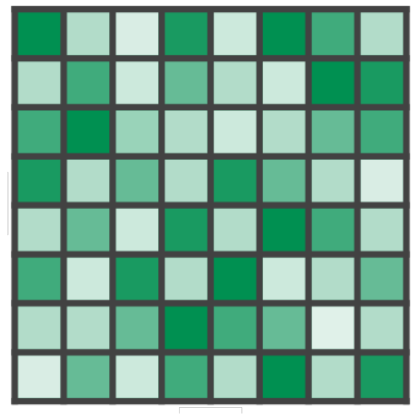
Kernel sizes selected from ResNeXt-101_32x16d/ResNet-50

NETWORK PERFORMANCE

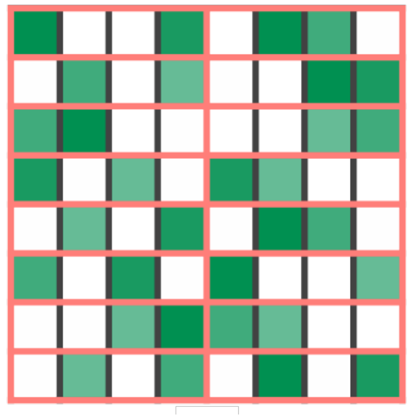
End to End Inference Speedup

NETWORK	DATA TYPE	SCENARIO	PERFORMANCE
BERT-Large	INT8	BS=256, SeqLen=128	6200 seq/s
		BS=1-256, SeqLen=128	1.3X-1.5X
ResNeXt-101_32x16d	FP16	BS=256	2700 images/second
		BS=1-256	Up to 1.3X
	INT8	BS=256	4400 images/second
		BS=1-256	Up to 1.3X

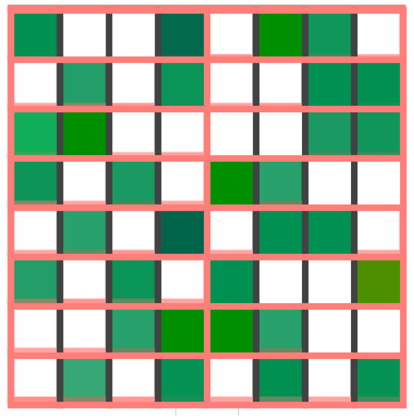
RECIPE FOR 2:4 SPARSE NETWORK TRAINING



Dense weights



2:4 sparse weights



Retrained 2:4 sparse weights

1) Train (or obtain) a dense network

2) Prune for 2:4 sparsity

3) Repeat the original training procedure

- Same hyper-parameters as in step-1
- Initialize to weights from step-2
- Maintain the 0 pattern from step-2: no need to recompute the mask

IMAGE CLASSIFICATION

ImageNet

Network	Accuracy				
	Dense FP16	Sparse FP16		Sparse INT8	
ResNet-34	73.7	73.9	0.2	73.7	-
ResNet-50	76.6	76.8	0.2	76.8	0.2
ResNet-101	77.7	78.0	0.3	77.9	-
ResNeXt-50-32x4d	77.6	77.7	0.1	77.7	-
ResNeXt-101-32x16d	79.7	79.9	0.2	79.9	0.2
DenseNet-121	75.5	75.3	-0.2	75.3	-0.2
DenseNet-161	78.8	78.8	-	78.9	0.1
Wide ResNet-50	78.5	78.6	0.1	78.5	-
Wide ResNet-101	78.9	79.2	0.3	79.1	0.2
Inception v3	77.1	77.1	-	77.1	-
Xception	79.2	79.2	-	79.2	-
VGG-16	74.0	74.1	0.1	74.1	0.1
VGG-19	75.0	75.0	-	75.0	-

IMAGE CLASSIFICATION

ImageNet

Network	Accuracy				
	Dense FP16	Sparse FP16		Sparse INT8	
ResNet-50 (SWSL)	81.1	80.9	-0.2	80.9	-0.2
ResNeXt-101-32x8d (SWSL)	84.3	84.1	-0.2	83.9	-0.4
ResNeXt-101-32x16d (WSL)	84.2	84.0	-0.2	84.2	-
SUNet-7-128	76.4	76.5	0.1	76.3	-0.1
DRN-105	79.4	79.5	0.1	79.4	-

WSL = Weakly Supervised Learning
SWSL = Semi-Weakly Supervised Learning

SEGMENTATION/DETECTION

COCO 2017, bbox AP

Network	Accuracy				
	Dense FP16	Sparse FP16		Sparse INT8	
MaskRCNN-RN50	37.9	37.9	-	37.8	-0.1
SSD-RN50	24.8	24.8	-	24.9	0.1
FasterRCNN-RN50-FPN-1x	37.6	38.6	1.0	38.4	0.8
FasterRCNN-RN50-FPN-3x	39.8	39.9	-0.1	39.4	-0.4
FasterRCNN-RN101-FPN-3x	41.9	42.0	0.1	41.8	-0.1
MaskRCNN-RN50-FPN-1x	39.9	40.3	0.4	40.0	0.1
MaskRCNN-RN50-FPN-3x	40.6	40.7	0.1	40.4	-0.2
MaskRCNN-RN101-FPN-3x	42.9	43.2	0.3	42.8	-0.1
RetinaNet-RN50-FPN-1x	36.4	37.4	1.0	37.2	0.8
RPN-RN50-FPN-1x	45.8	45.6	-0.2	45.5	-0.3

RN = ResNet Backbone
FPN = Feature Pyramid Network
RPN = Region Proposal Network

NLP - TRANSLATION

EN-DE WMT'14

		Accuracy				
Network	Metric	Dense FP16	Sparse FP16		Sparse INT8	
GNMT	BLEU	24.6	24.9	0.3	24.9	0.3
FairSeq Transformer	BLEU	28.2	28.5	0.3	28.3	0.1
Levenstein Transformer	Validation Loss	6.16	6.18	-0.2	6.16	-

NLP – LANGUAGE MODELING

Transformer-XL, BERT

			Accuracy				
Network	**Task**	**Metric**	**Dense FP16**	**Sparse FP16**		**Sparse INT8**	
Transformer-XL	enwik8	BPC	1.06	1.06	-	-	
BERT-Base	SQuAD v1.1	F1	87.6	88.1	0.5	88.1	0.5
BERT-Large	SQuAD v1.1	F1	91.1	91.5	0.4	91.5	0.4

SUMMARY (MIXED PRECISION)

A100 introduces wide variety to Tensor Cores for DL training - FP16/BF16/TF32

- TF32 is the default on A100
- FP16/BF16 options are for maximum performance

To enable Tensor Cores:

- No code changes for TF32
- AMP for FP16/BF16

To maximize perf:

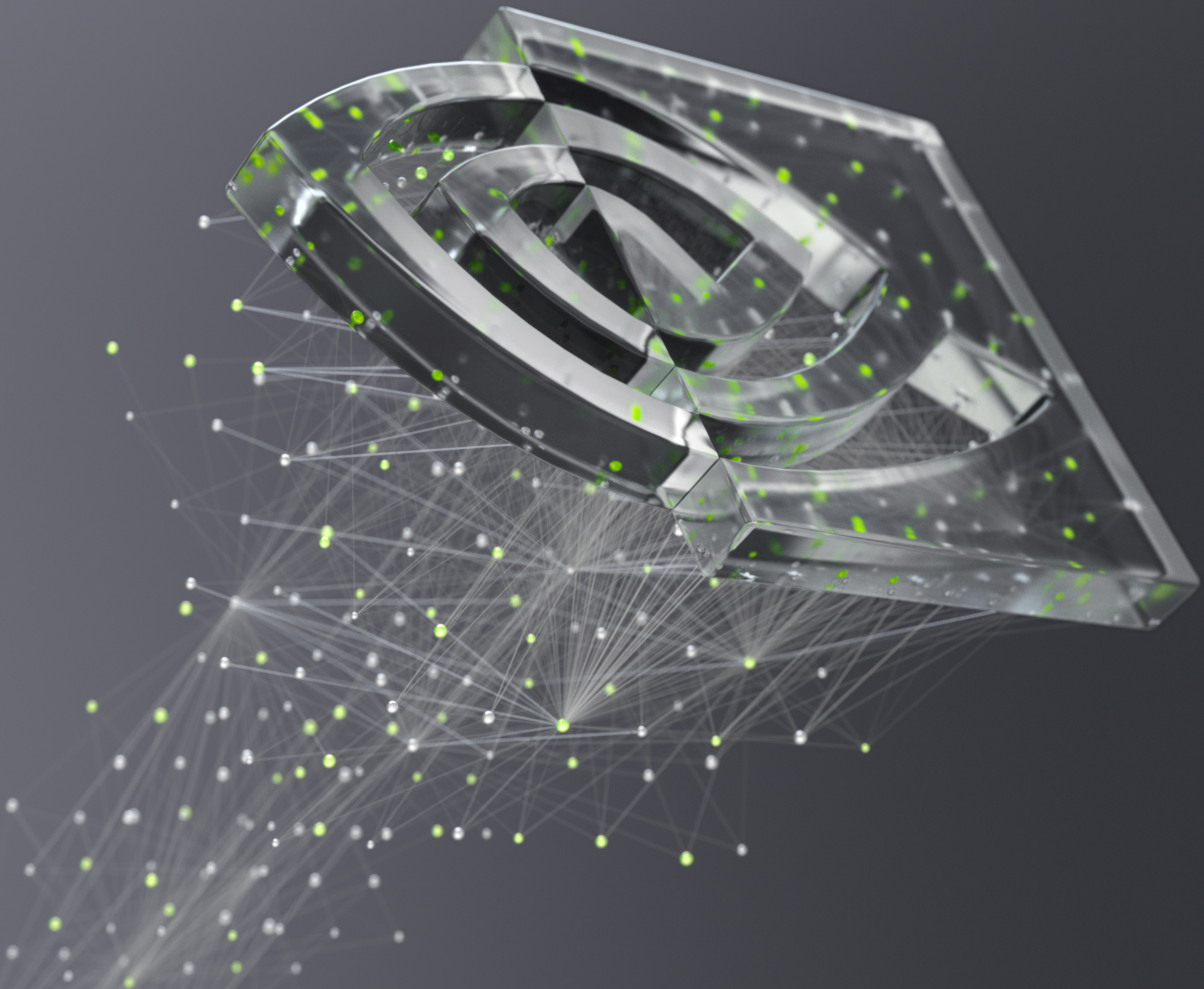
- Make use of DL Profilers
- Ensure training time spent on GPU and math-bound layers, as well as TC utilization

SUMMARY (SPARSITY)

We moved fine-grained weight sparsity from research to production

Fine-grained structured sparsity is:

- 50% sparse, 2 out of 4 elements are zero
- Accurate with our 3-step universal fine-tuning recipe
    - Simple recipe: train dense, prune, re-train sparse
    - Across many tasks, networks, optimizers
- Fast with the NVIDIA Ampere Architecture's Sparse Tensor Cores
    - Up to 1.85x in individual layers
    - Up to 1.5x in end-to-end networks



NVIDIA

MIXED PRECISION FOR COMPUTE

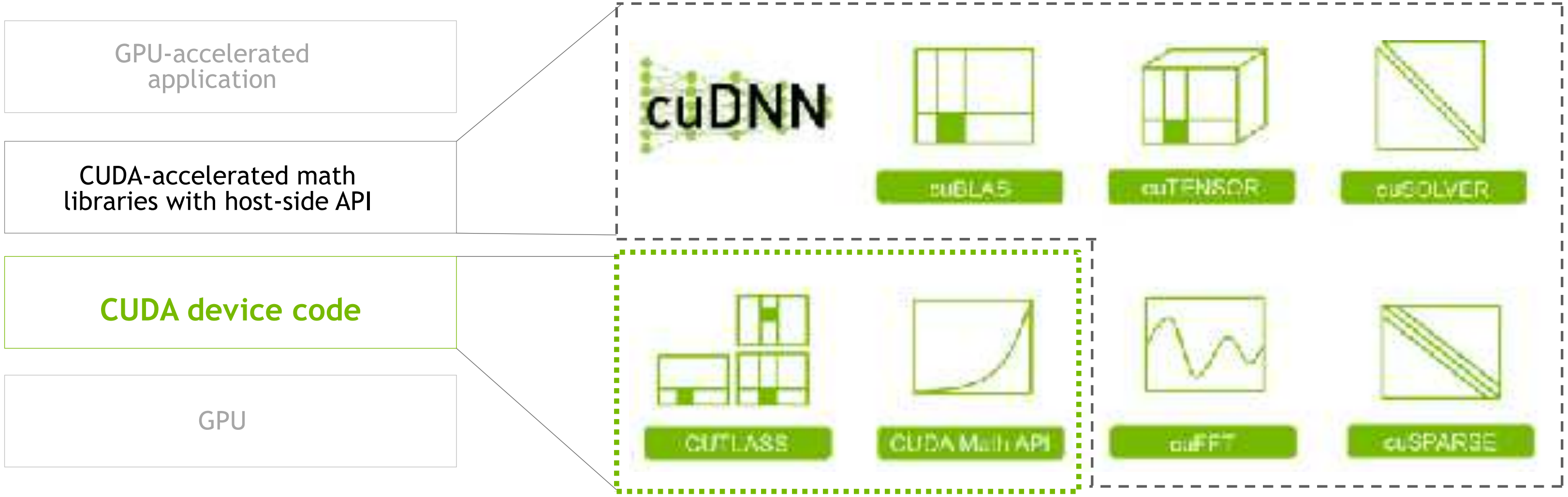
PROGRAMMING NVIDIA AMPERE ARCHITECTURE

Deep Learning and Math Libraries using Tensor Cores (with CUDA kernels under the hood)

- cuDNN, cuBLAS, cuTENSOR, cuSOLVER, cuFFT, cuSPARSE
- "CUDNN V8: New Advances in Deep Learning Acceleration" (GTC 2020 - S21685)
- "How CUDA Math Libraries Can Help you Unleash the Power of the New NVIDIA A100 GPU" (GTC 2020 – S21681)
- "Inside the Compilers, Libraries and Tools for Accelerated Computing" (GTC 2020 – S21766)

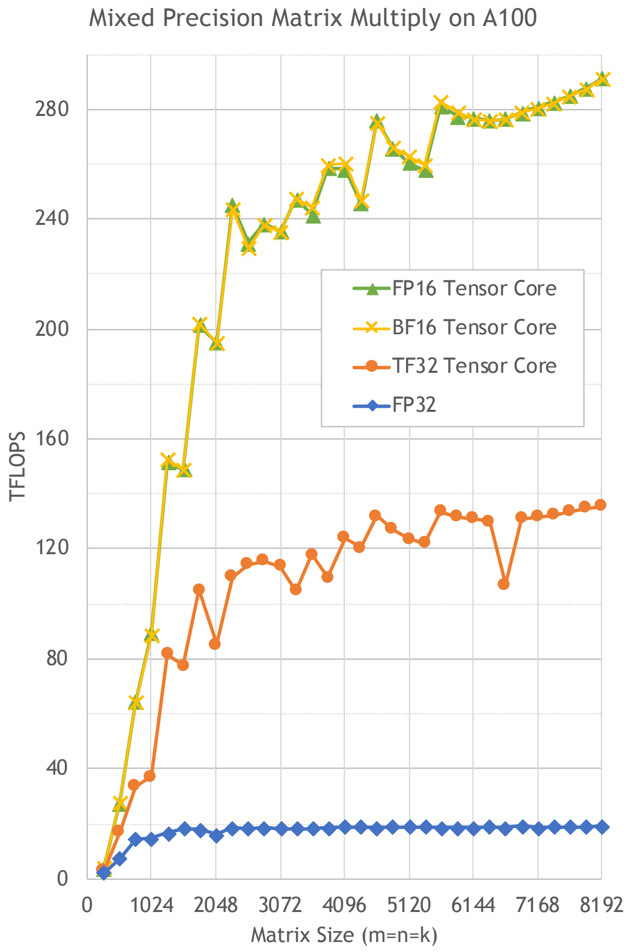
CUDA C++ Device Code

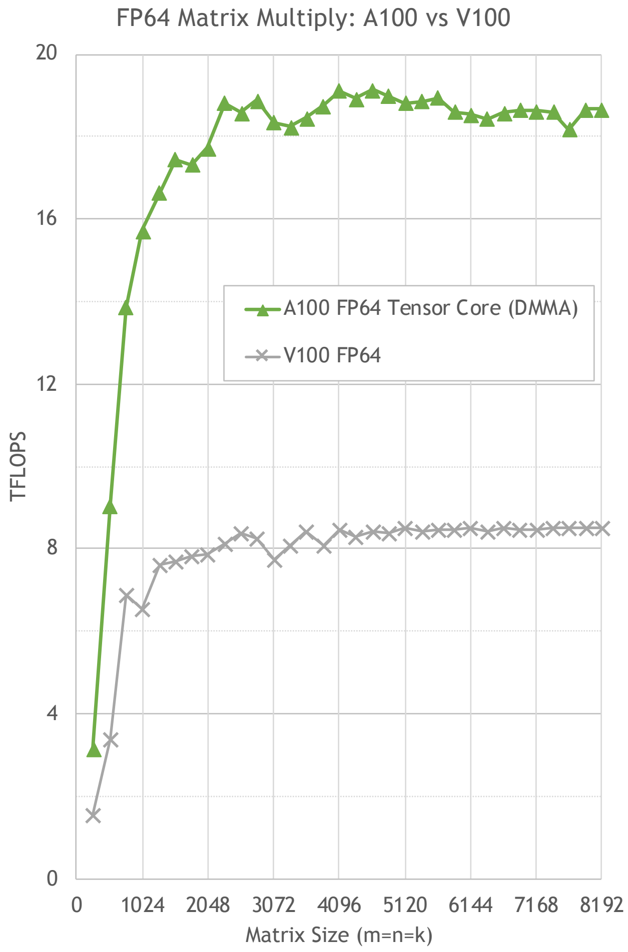
- CUTLASS, CUDA Math API, CUB, Thrust, libcu++

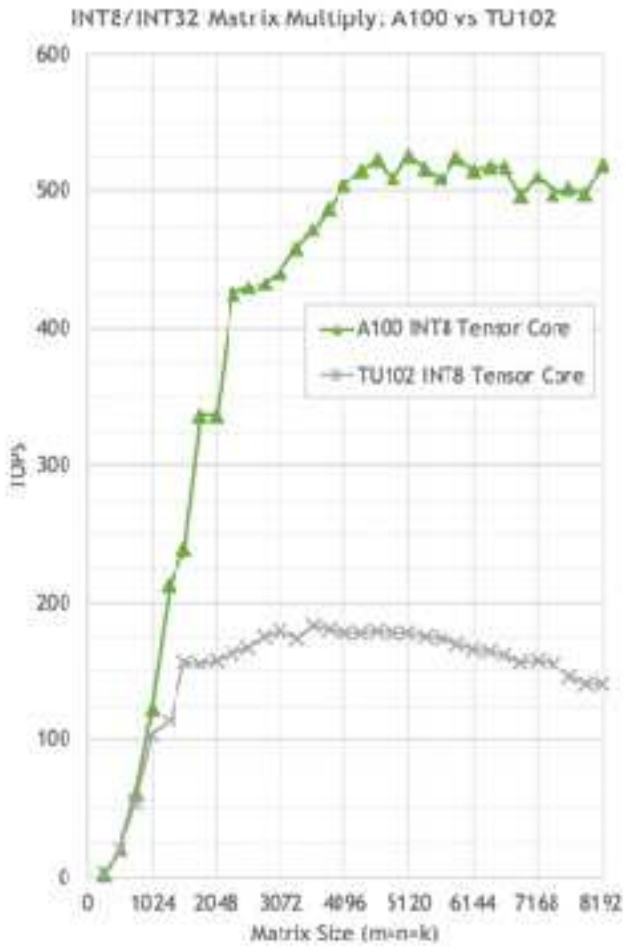


cuBLAS

3rd GENERATION TENSOR CORES ADD SUPPORT FOR FP64 & NEW TYPE BF16 & COMPUTE TYPE TF32

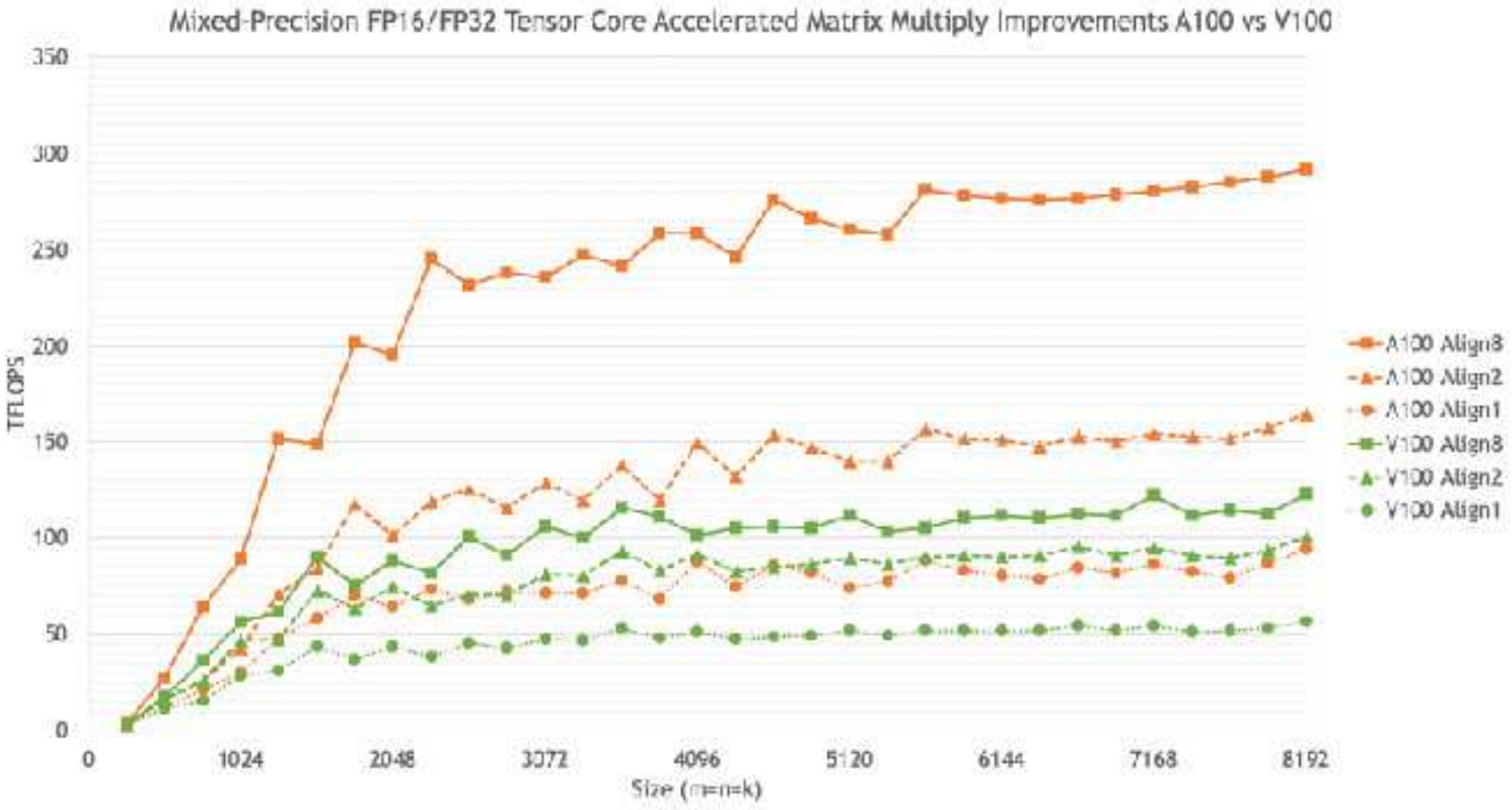






cuBLAS

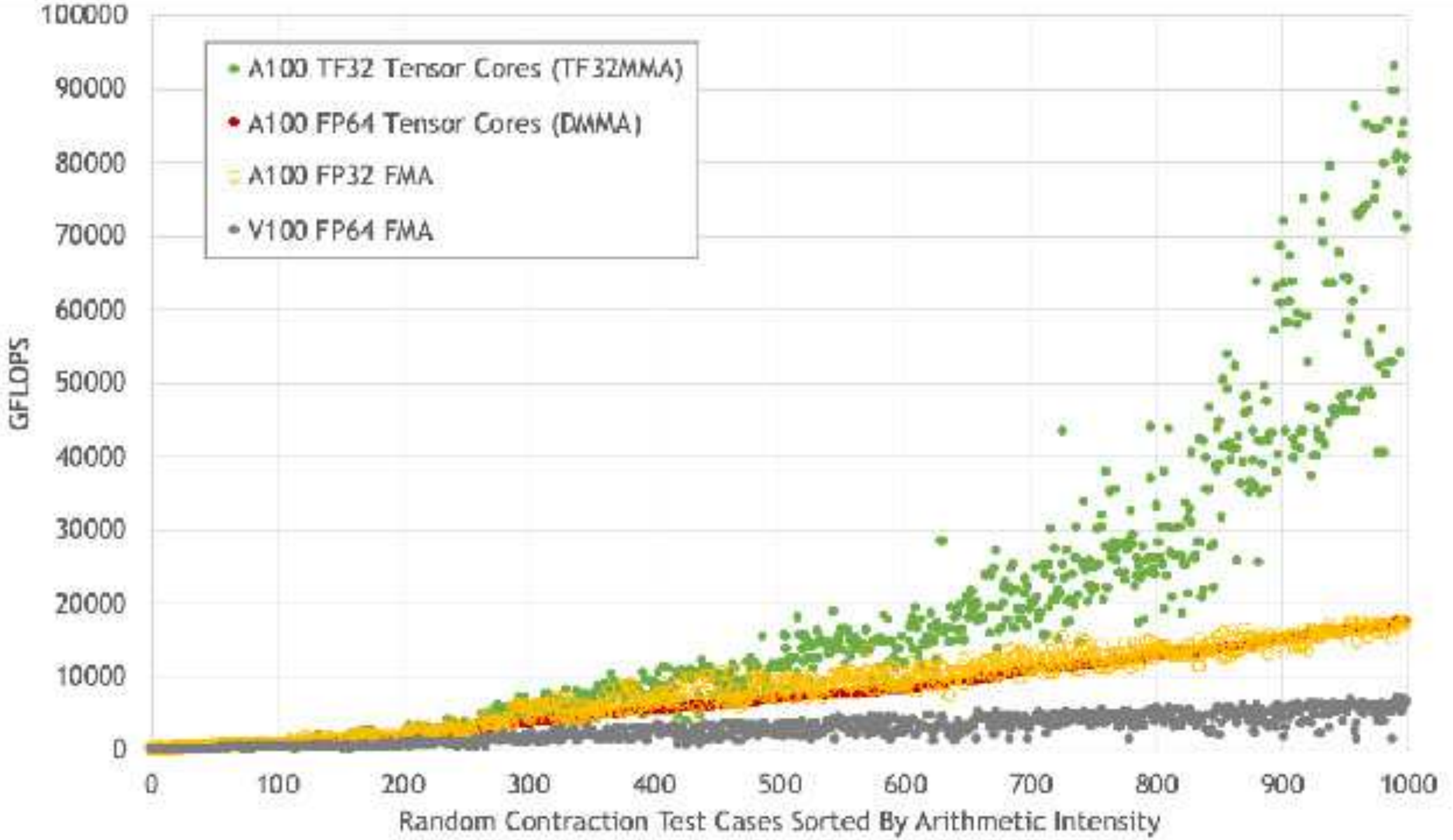
NO MORE ALIGNMENT RESTRICTIONS FOR TENSOR CORE EXECUTION ELIBIGILITY OF MATRIX MULTIPLIES*



AlignN means alignment to 16bit multiples of N. For example, align8 are problems aligned to 128bits or 16 bytes.
(*) In some cases if heuristics determine it will result in better performance kernels that do not use the tensor cores might be selected

cuTENSOR

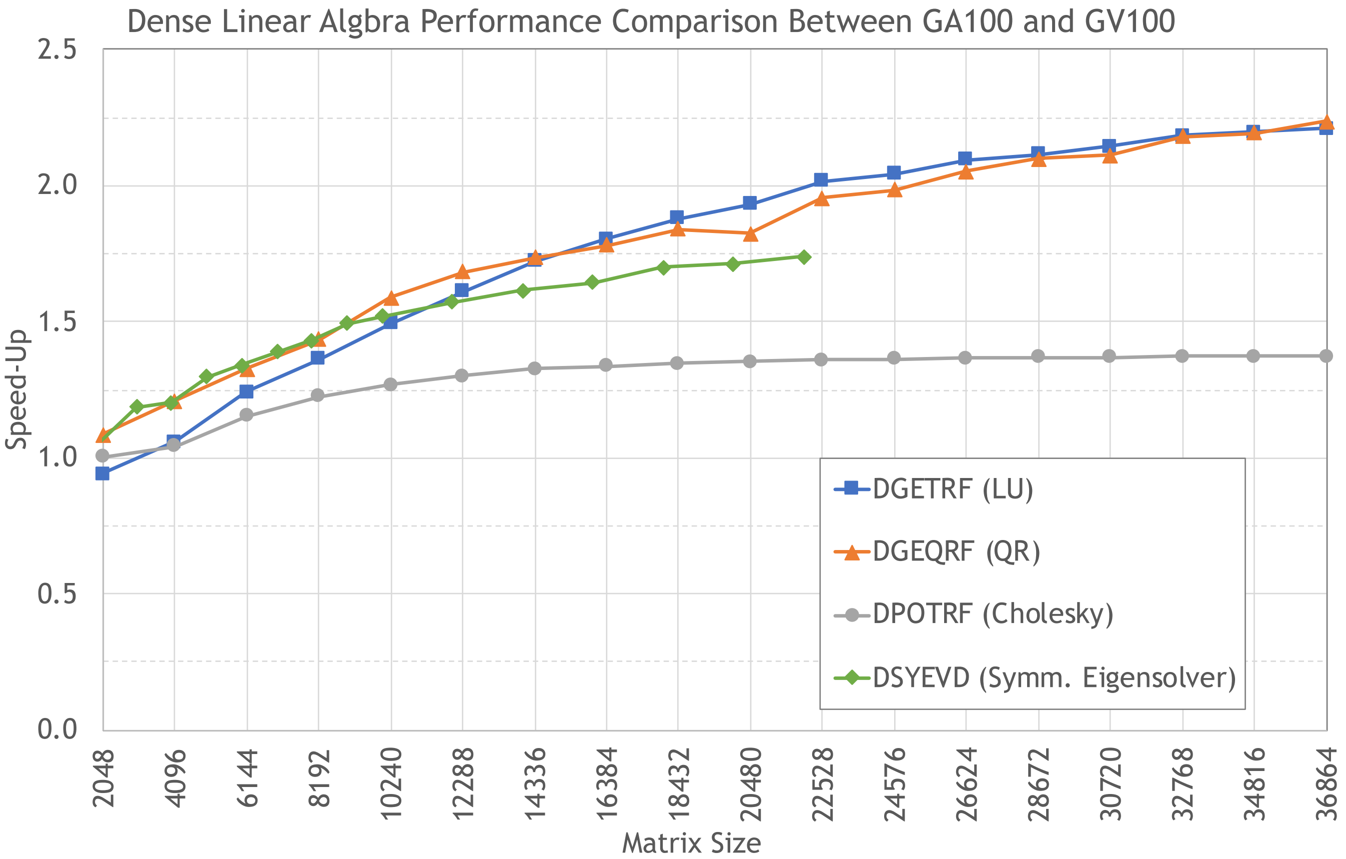
PERFORMANCE IMPROVEMENTS FROM FP64 AND TF32 COMPUTE TENSOR CORES

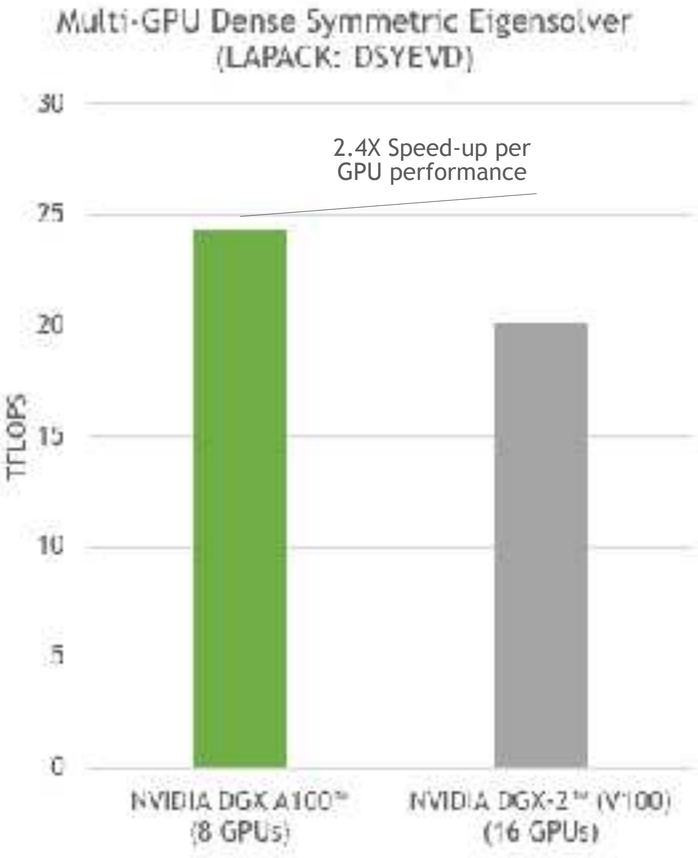


TENSOR CORE ACCELERATED ITERATIVE REFINEMENT SOLVER

cuSOLVER

DENSE LINEAR ALGEBRA PERFORMANCE ON THE NEW NVIDIA A100 & DGX-A100™





Results comparing CUDA 11.0 cuSOLVER NVIDIA A100 to CUDA 10.2 on V100.

TENSOR CORE ACCELERATED LIBRARIES

Multi-precision numerical methods

Solving linear system of dense equations *Ax=b*

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

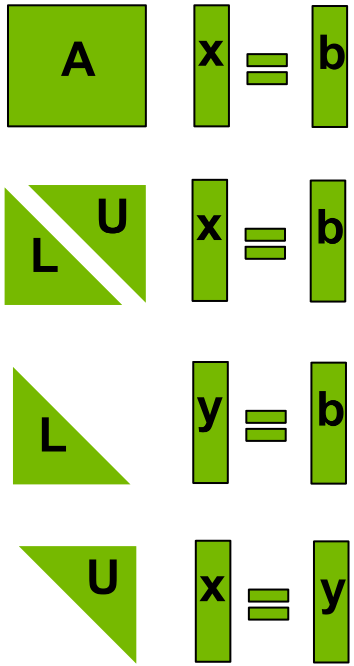
A x = b

LUx = b

Ly = b

then

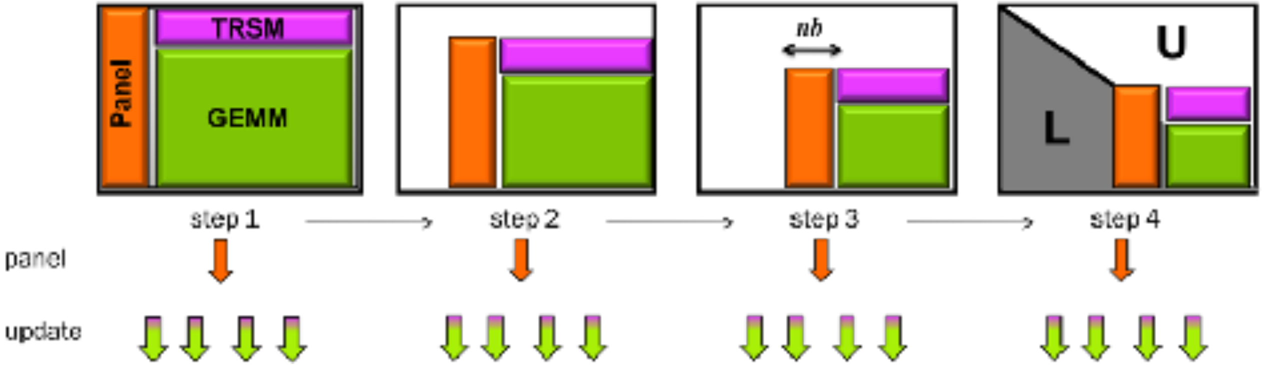
Ux = y



TENSOR CORE ACCELERATED LIBRARIES

Multi-precision numerical methods

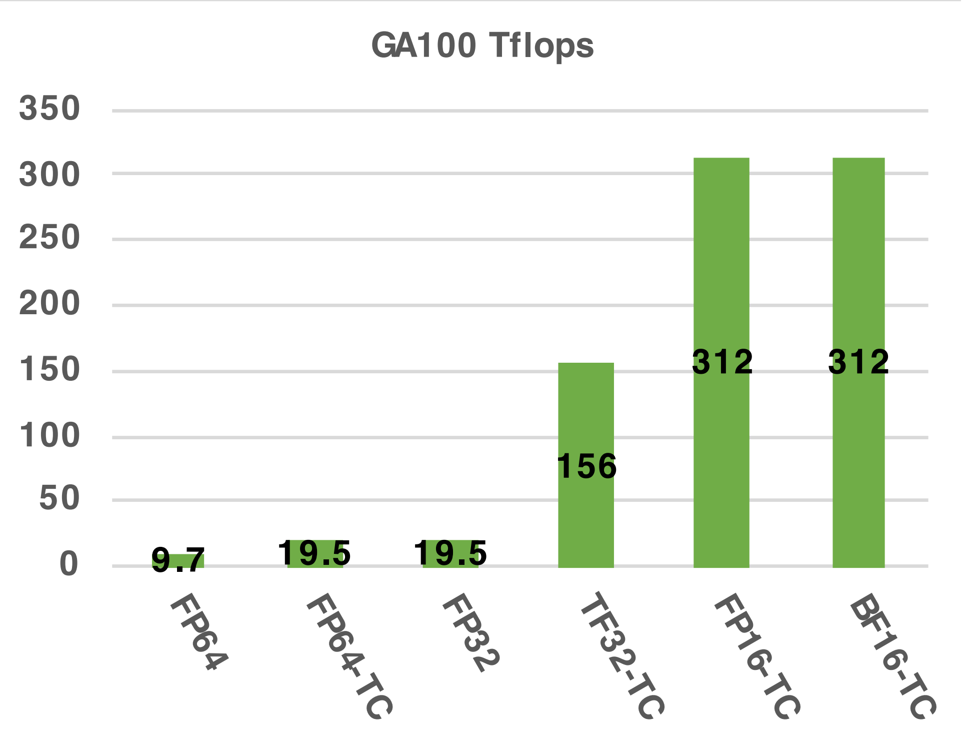
Solving linear system of dense equations *Ax=b*

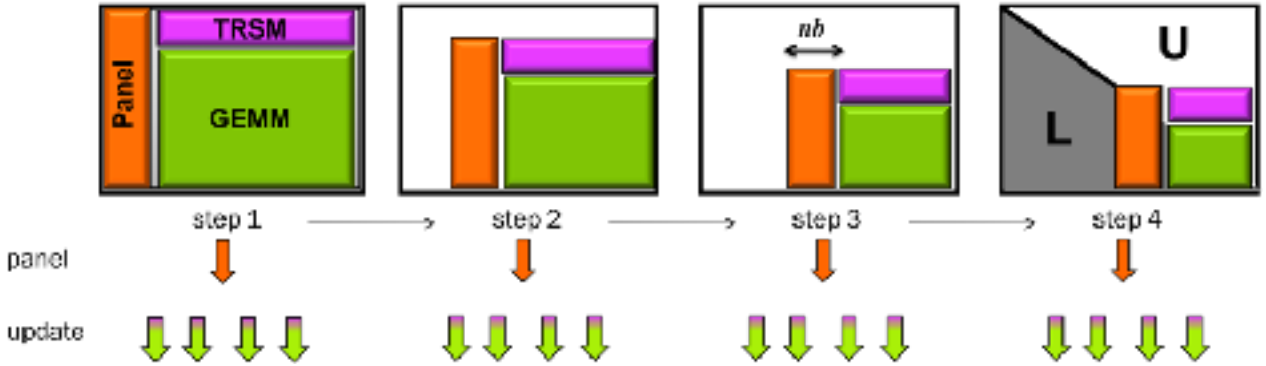


TENSOR CORE ACCELERATED LIBRARIES

Multi-precision numerical methods

Solving linear system of dense equations *Ax=b*

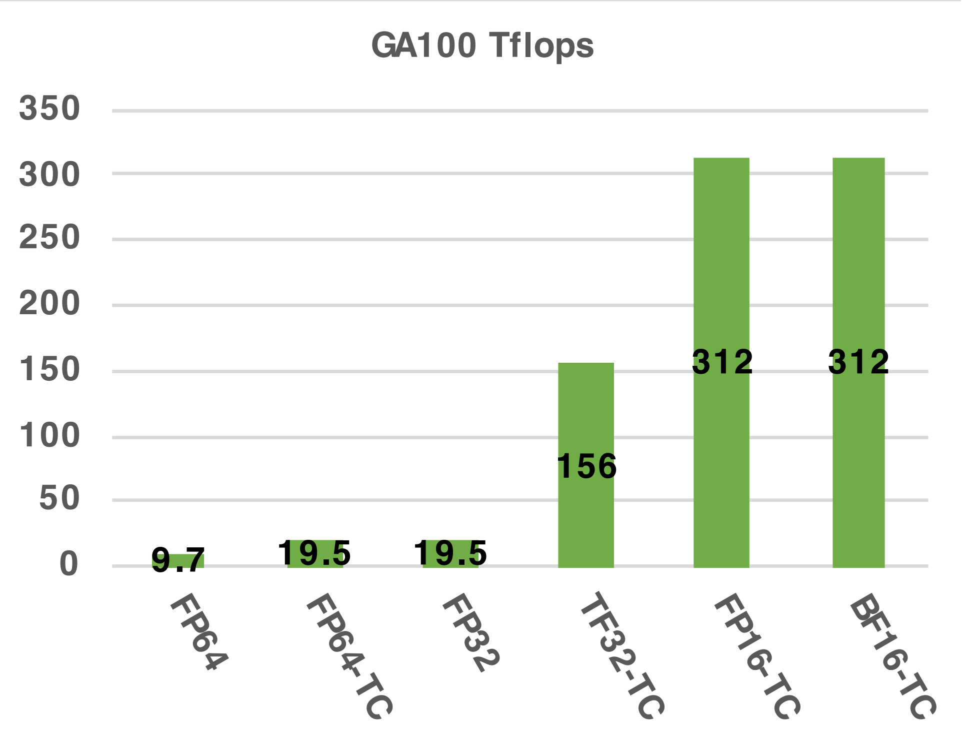


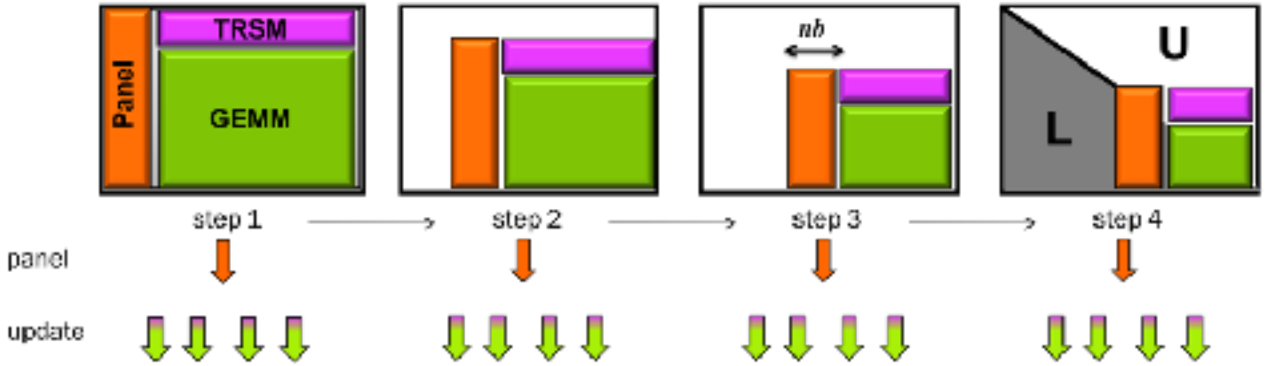


TENSOR CORE ACCELERATED LIBRARIES

Multi-precision numerical methods

Solving linear system of dense equations *Ax=b*



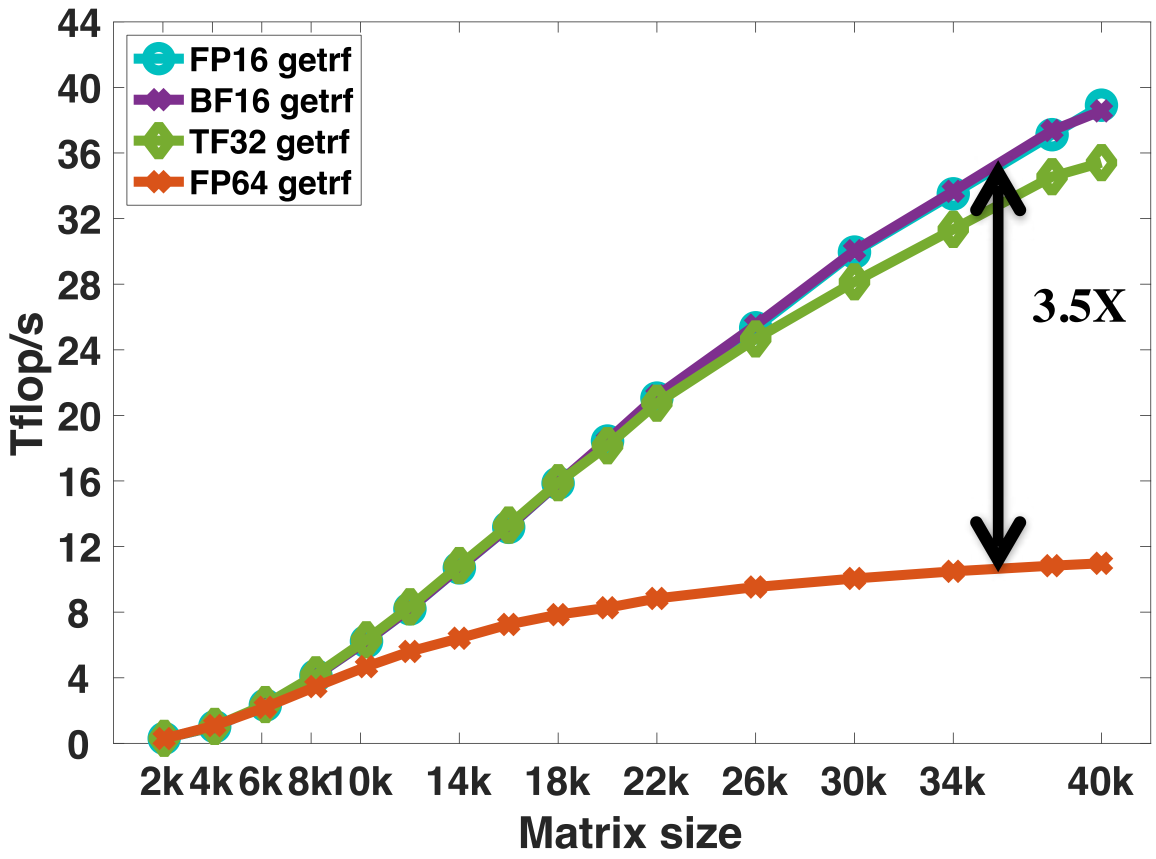


How about a multi-precision LU then ?

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

TENSOR CORE ACCELERATED LIBRARIES

Performance of the LU factorization with different precisions



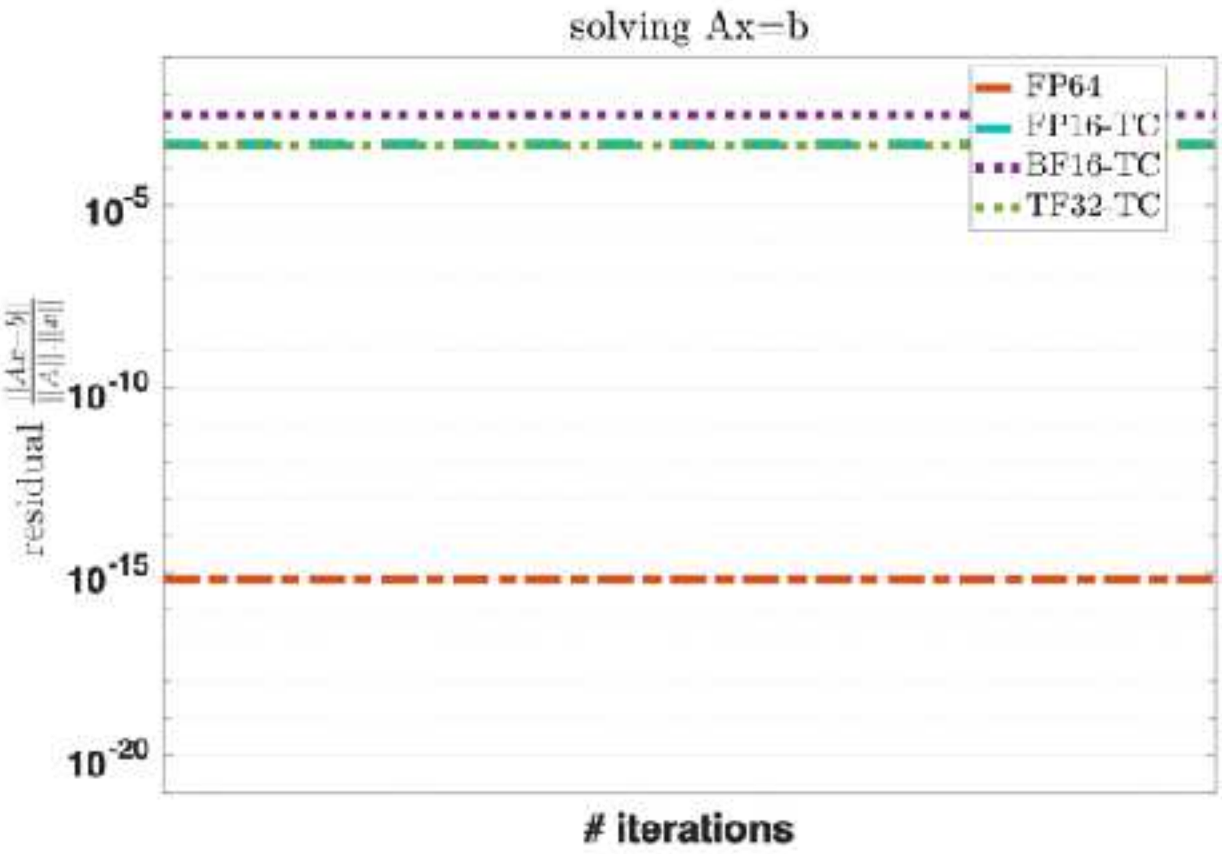
Performance of the LU with different precisions
Flops = $2n^3/(3 \text{ time})$ e.g.,twice higher is twice faster

- LU using FP64-TC
- LU using FP16-TC
- LU using BF16-TC
- LU using TF32-TC

Results obtained using CUDA 11.0 and A100 GPU.

TENSOR CORE ACCELERATED LIBRARIES

Accuracy just after the reduced precision LU factorization



Accuracy of the obtained solution

- ➢ FP64-TC provide a solution down to the FP64 accuracy
- ➢ TF32 and FP16 provide a solution to around 1E-05 accuracy
- ➢ **Obtained solution has 11 digits loss compared to the FP64 one,**
- ➢ **can we do better and achieve the FP64 accuracy?**

Results obtained using CUDA 11.0 and A100 GPU.

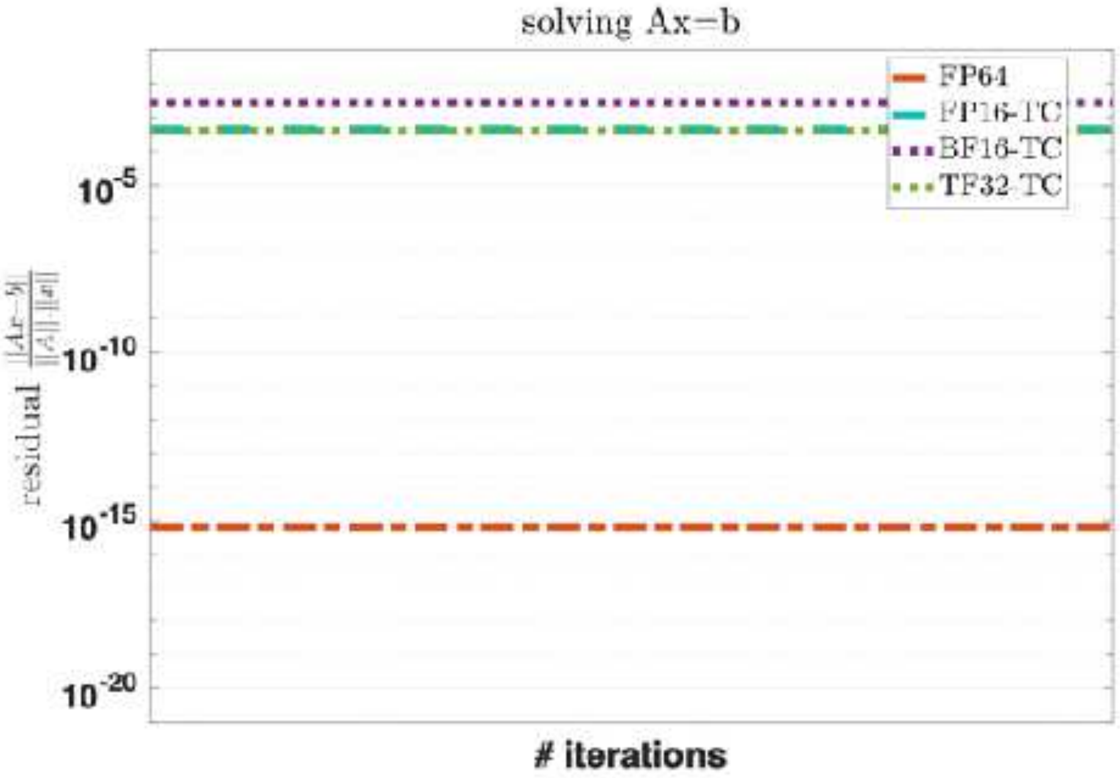
TENSOR CORE ACCELERATED LIBRARIES

How can we get to FP64 accuracy?

Idea: use reduced precision to compute the expensive flops (LU $O(n^3)$) and then iteratively refine the solution ($O(n^2)$) in order to achieve the FP64 level of accuracy

Iterative refinement for solving *Ax = b*:

Perform a factorization in reduced precision *A = LU*



E. Carson, N. J. Higham, **Accelerating the Solution of Linear Systems by Iterative Refinement in Three Precisions,** SIAM J. Sci. Comput., 40(2), A817–A847.
A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham, **Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers**, SC-18 Dallas, 2018
A.Haidar, H. Bayraktar, S. Tomov, J. Dongarra, N. J. Higham **Mixed-Precision Iterative Refinement using Tensor Cores on GPUs to Accelerate Solution of Linear Systems,** submitted Royal Society Journal UK 2020.

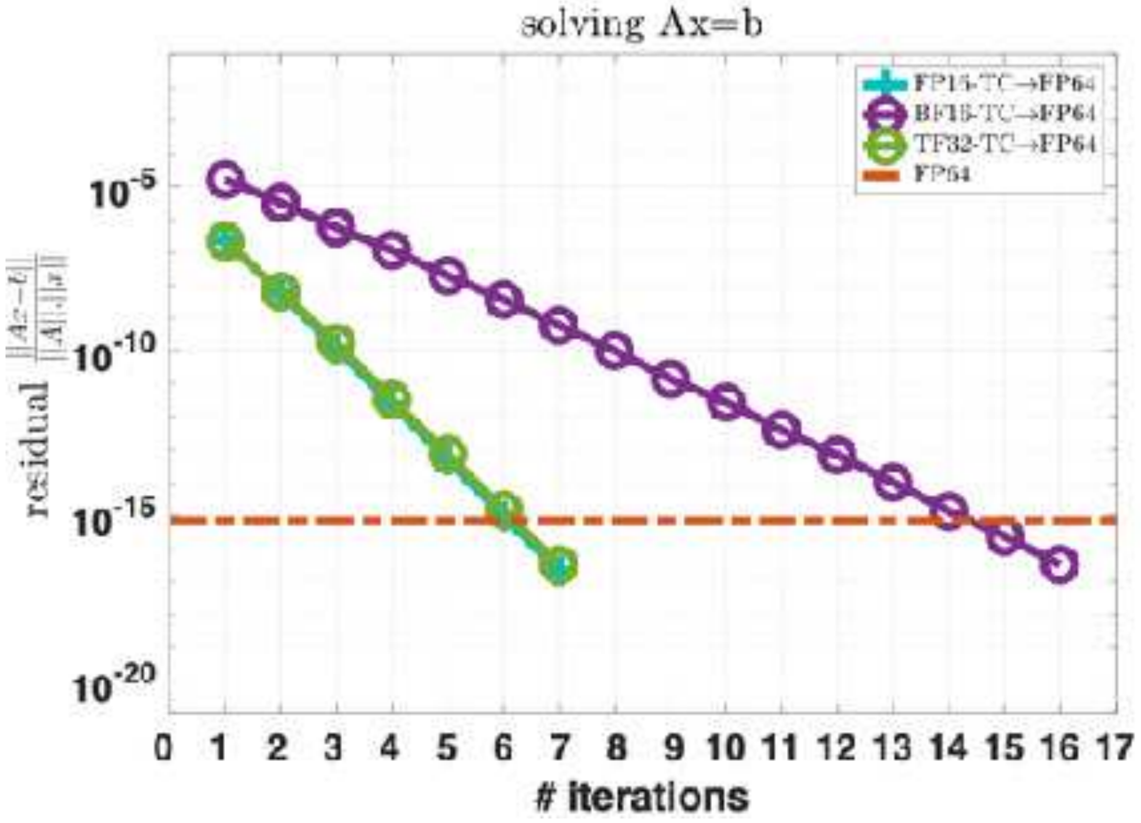
TENSOR CORE ACCELERATED LIBRARIES

How can we get to FP64 accuracy?

Idea: use reduced precision to compute the expensive flops (LU $O(n^3)$) and then iteratively refine the solution ($O(n^2)$) in order to achieve the FP64 level of accuracy

Iterative refinement for solving *Ax = b*:

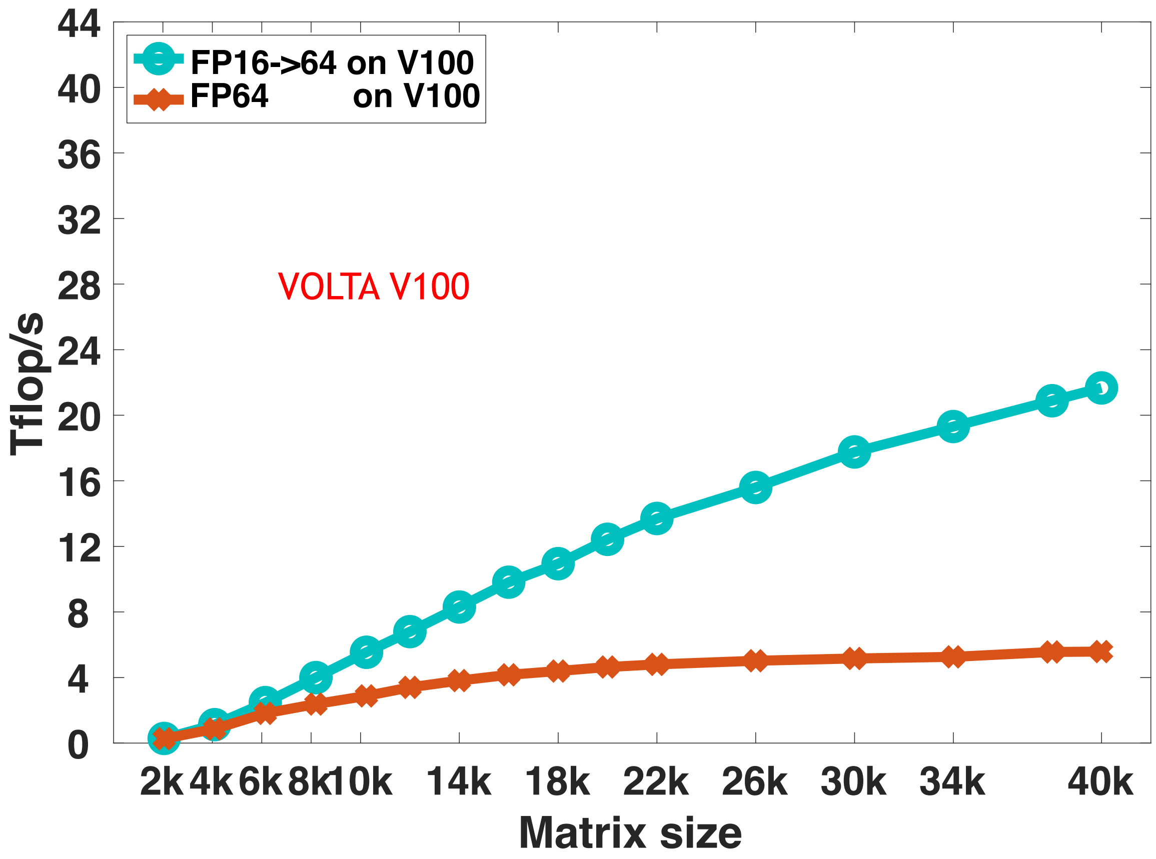
Perform a factorization in reduced precision *A = LU*
refine
WHILE || r || > eps_FP64
1. Find correction c such that *Ac = r*, *c* = U\(L\r)
2. *x = x + c*
3. *r = b - Ax* (with original A).
END



E. Carson, N. J. Higham, **Accelerating the Solution of Linear Systems by Iterative Refinement in Three Precisions**, SIAM J. Sci. Comput., 40(2), A817–A847.
A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham, **Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers**, SC-18 Dallas, 2018
A.Haidar, H. Bayraktar, S. Tomov, J. Dongarra, N. J. Higham **Mixed-Precision Iterative Refinement using Tensor Cores on GPUs to Accelerate Solution of Linear Systems**, submitted Royal Society Journal UK 2020.

TENSOR CORE ACCELERATED LIBRARIES

Performance Behavior, Hilbert matrices, V100



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

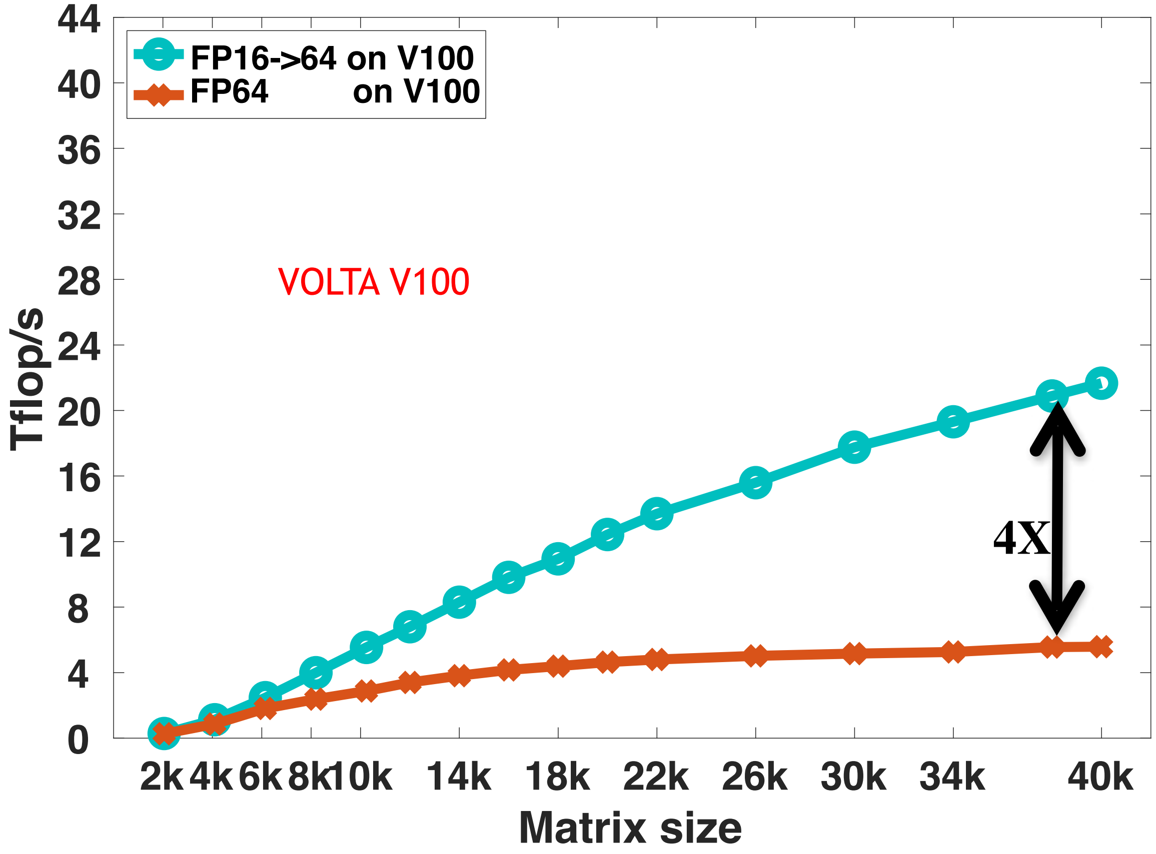
- solving Ax = b using **FP64 LU**
- solving Ax = b using **FP16 Tensor Cores LU** and iterative refinement to achieve FP64 accuracy
- **FP16** is about **4X** faster within a solution to the FP64 accuracy.

Results obtained using CUDA 11.0 and V100 GPU.

Problem generated with Hilbert matrices.

TENSOR CORE ACCELERATED LIBRARIES

Performance Behavior, Hilbert matrices, V100



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

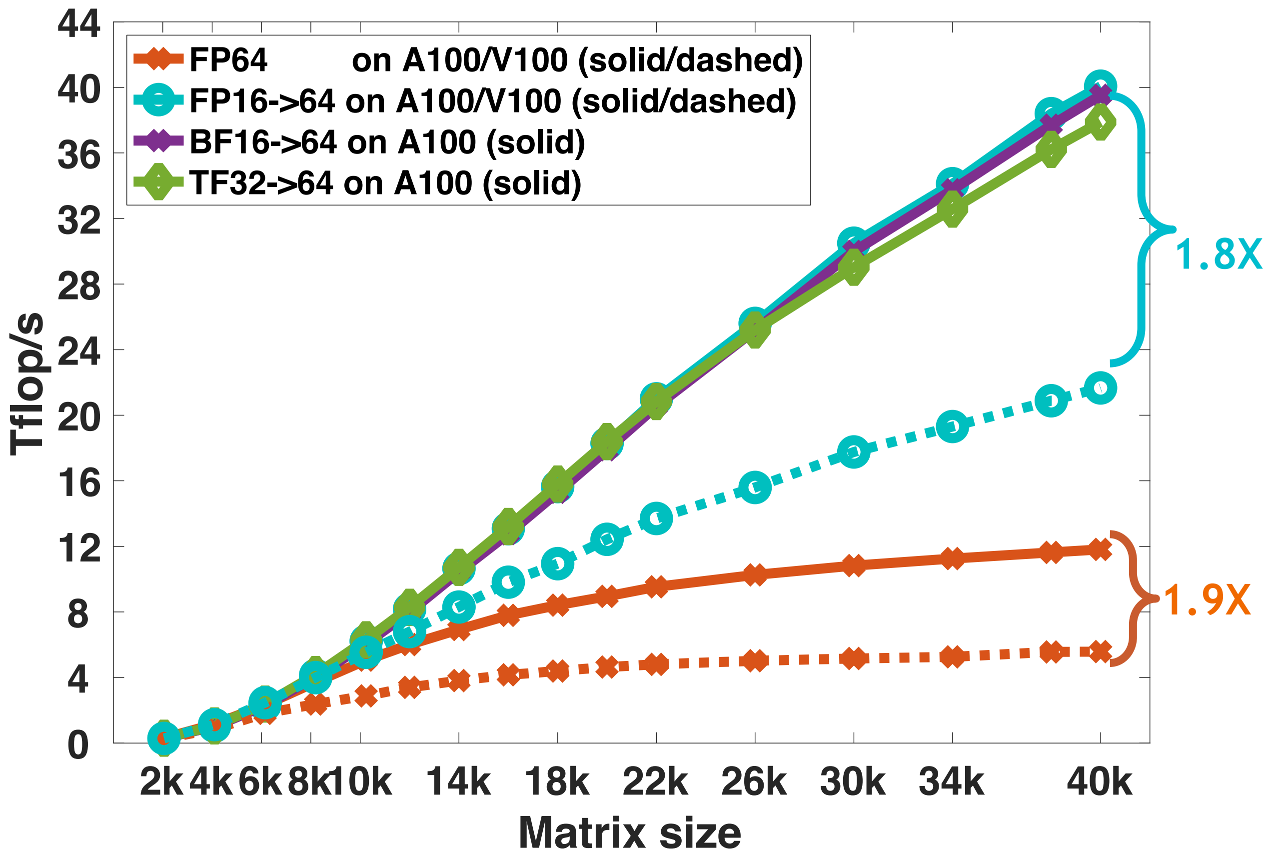
- solving Ax = b using **FP64 LU**
- solving Ax = b using **FP16 Tensor Cores LU** and iterative refinement to achieve FP64 accuracy
- **FP16** is about **4X** faster within a solution to the FP64 accuracy.

Results obtained using CUDA 11.0 and V100 GPU.

Problem generated with Hilbert matrices.

TENSOR CORE ACCELERATED ITERATIVE REFINEMENT SOLVER

Performance Behavior, Hilbert matrices, V100 v.s. A100



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

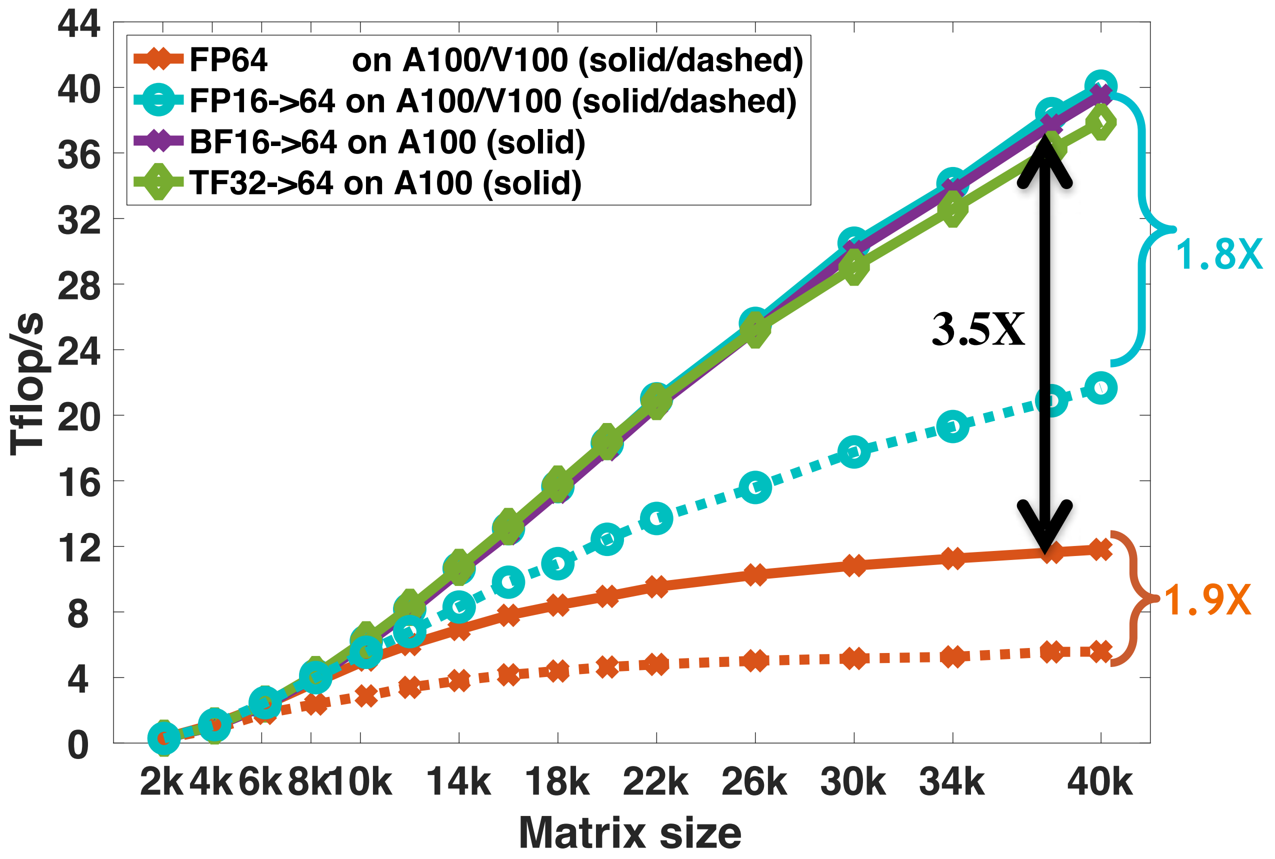
- ➢ Speedup compared to FP64 has same trend on both hardware.
- ➢ TF32 is 3.3X faster within a solution to the FP64 accuracy.
- ➢ FP16 is 3.5X faster within a solution to the FP64 accuracy.
- ➢ A100 provides about 1.8X speedup over V100 for both FP16 and FP64 variants

Results obtained using CUDA 11.0 and V100, A100 GPU.

Problem generated with Hilbert matrices.

TENSOR CORE ACCELERATED ITERATIVE REFINEMENT SOLVER

Performance Behavior, Hilbert matrices, V100 v.s. A100



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

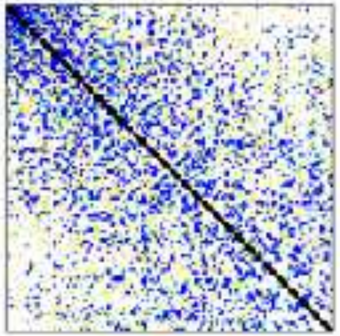
- Speedup compared to FP64 has same trend on both hardware.
- TF32 is 3.3X faster within a solution to the FP64 accuracy.
- FP16 is 3.5X faster within a solution to the FP64 accuracy.
- A100 provides about 1.8X speedup over V100 for both FP16 and FP64 variants

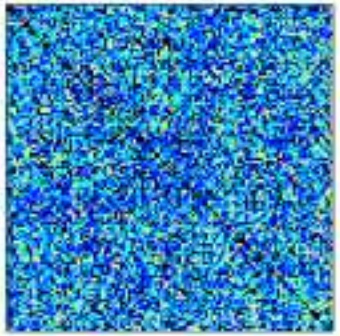
Results obtained using CUDA 11.0 and V100, A100 GPU.

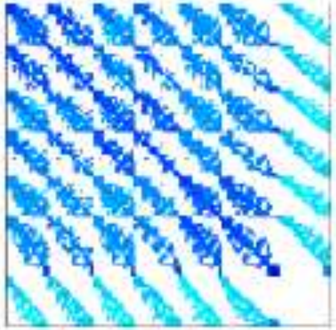
Problem generated with Hilbert matrices.

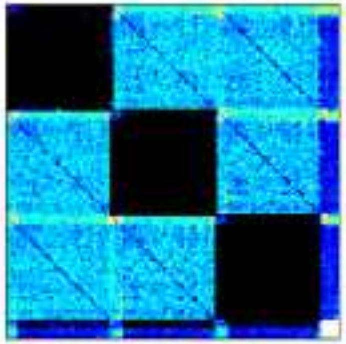
TENSOR CORE ACCELERATED ITERATIVE REFINEMENT SOLVER

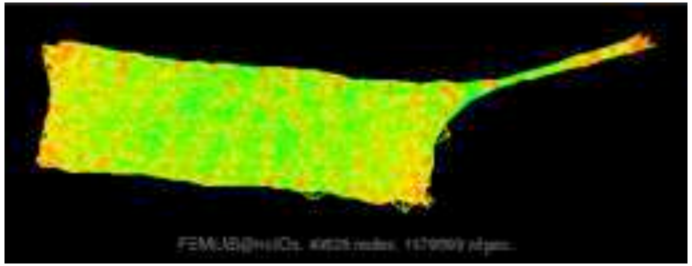
Matrices from SuiteSparse, A100

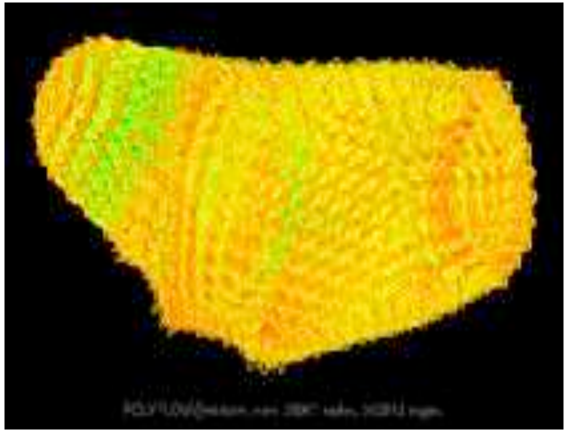


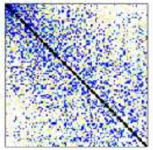
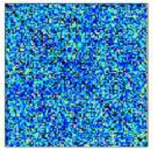






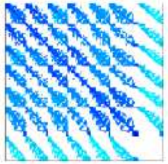
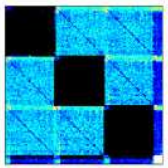




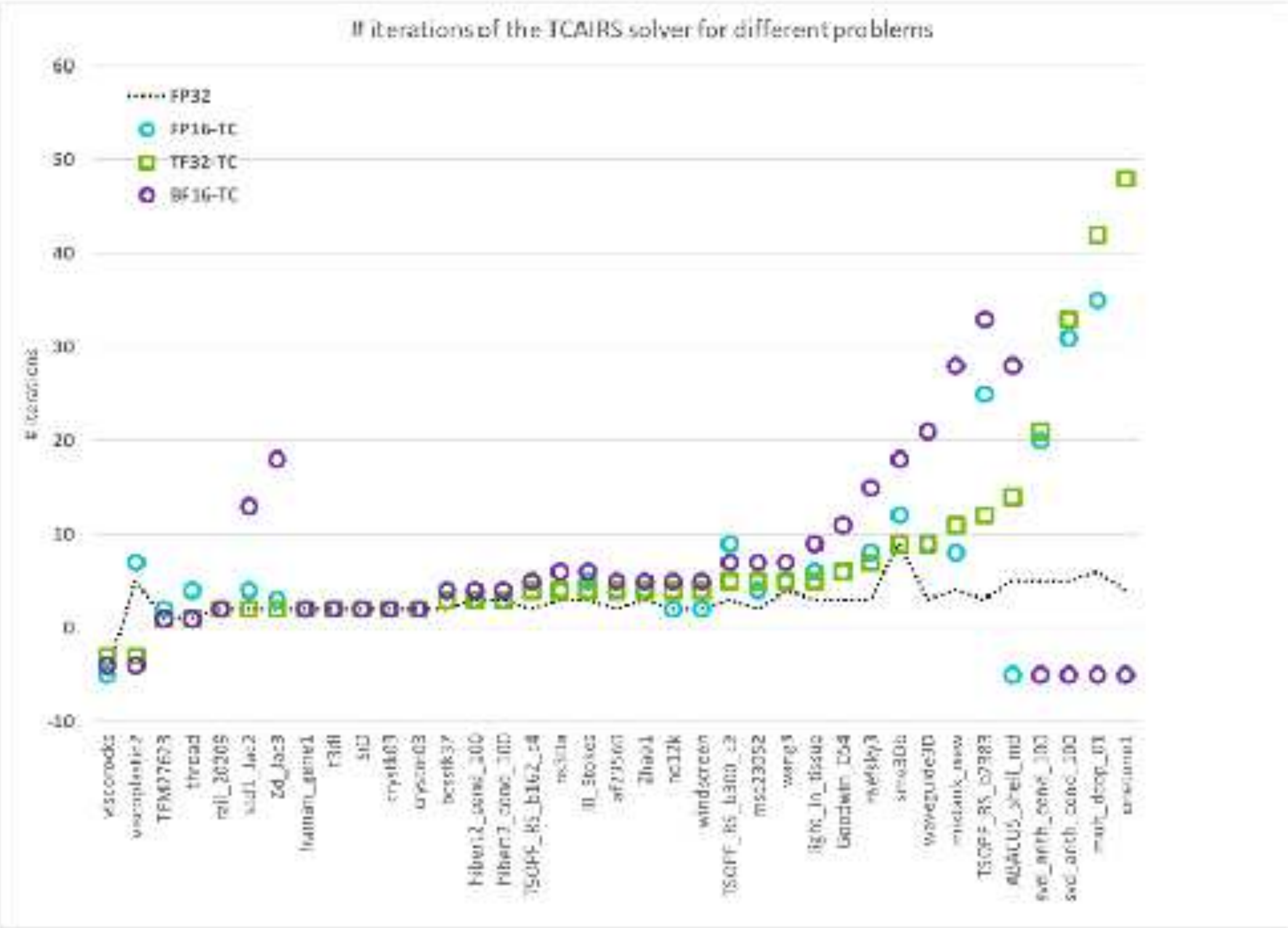
TCAIRS NUMERICAL BEHAVIOR

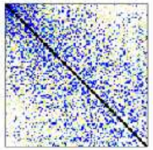
Matrices from SuiteSparse and other problems, A100

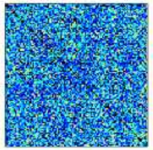



- ➢ Solving matrices from the SuiteSparse collection corresponding to a wide range of applications in fluid dynamics, structural mechanics, materials science, nuclear energy, oil and gas exploration and others
- ➢ TF32 converges faster than both FP16 and BF16 and is able to solve wider range of problems

Results obtained using CUDA 11.0 and A100 GPU.

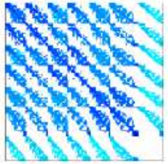


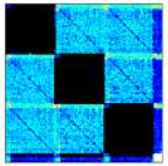




TCAIRS PERFORMANCE BEHAVIOR

Matrices from SuiteSparse and other problems, A100

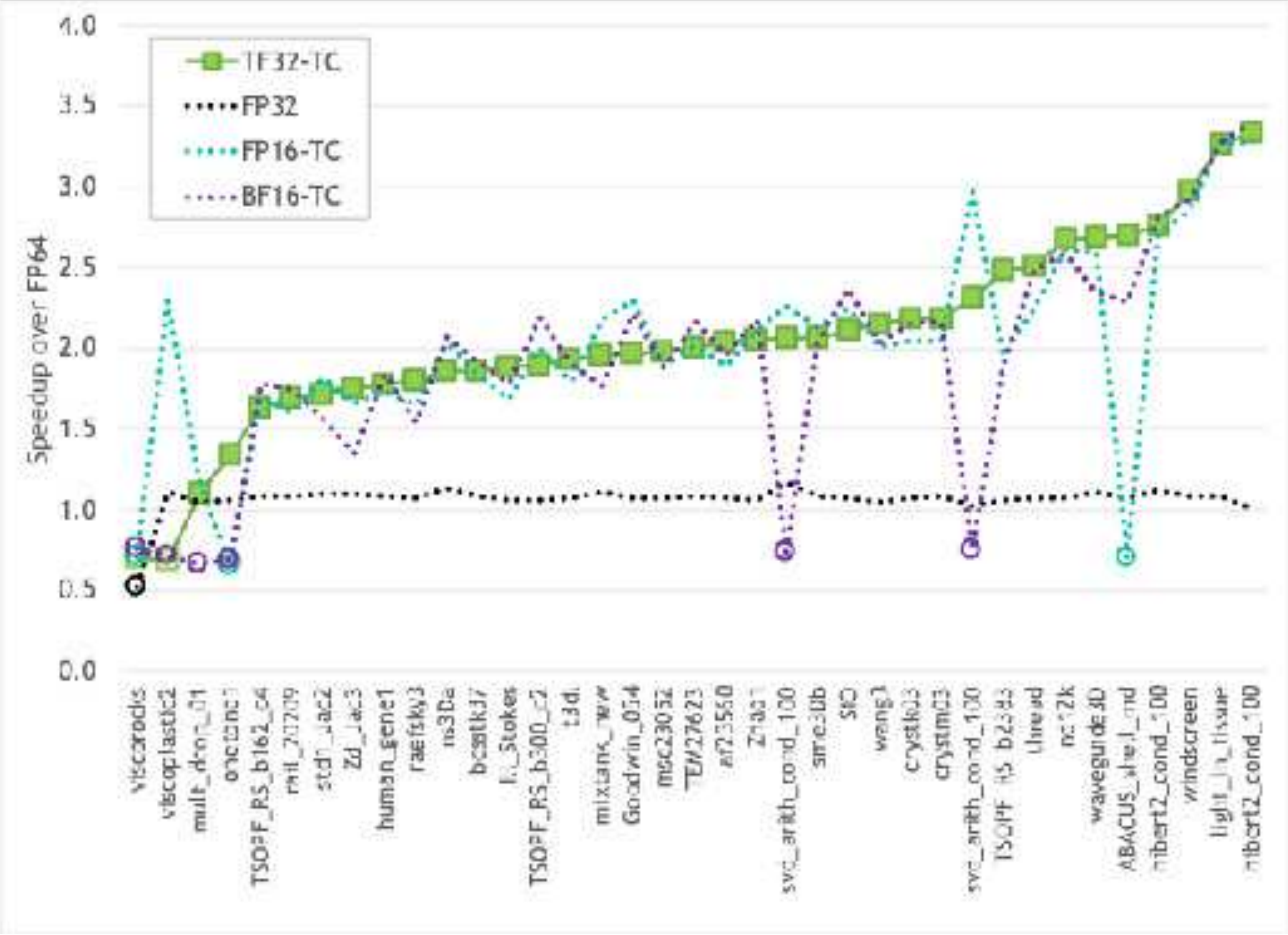




	Performance	Fallback cases	Notes
FP32	1x	1	Hard case
TF32	2x	2	Hard case
FP16 scaled	2x	3	Scaling fixes many cases
BF16	2x	6	Loss of precision is an issue for several cases

- ➢ TF32 converges faster than both FP16 and BF16 and is able to solve wider range of problems
- ➢ In terms of performance TF32 provide time to solution close or better than both BF16 and FP16
- ➢ **In summary, TF32 can be considered the most robust and the fastest variant**

Results obtained using CUDA 11.0 and A100 GPU.



TENSOR CORE ACCELERATED ITERATIVE REFINEMENT SOLVER

Tensor Core Accelerated Iterative Refinement Solver (TCAIRS)

January 2019

Magma 2.5.0

TCAIRS-LU real
FP64, FP32, FP16-TC

Nov 2019

cuSOLVER
CUDA 10.2

TCAIRS-LU real
TCAIRS-LU complex
TCAIRS-LU NRHS
FP64, FP32, FP16-TC

May 2020

cuSOLVER
CUDA 11.0

10.2 +
TCAIRS-QR real
TCAIRS-QR complex
TCAIRS-QR NRHS
FP32, FP16-TC
FP64-TC, TF32-TC, BF16-TC
Many other advancements

Mixed Precision Solvers are gaining a lot of attention for their power to provide a solution up to 4X-5X faster and for their energy efficiency.

CONCLUSION

Don’t blindly use double precision without considering what precision is required

Judicious use of precision tuning can lead to >4x speedup

Optimal approach may utilize 3 or even more different precisions

Mixed precision can accelerate compute and bandwidth bound parts

- use libraries where applicable
- design your code so it is easy to play with precision

NVIDIA