NVIDIA®

NVIDIA THE AI COMPANY

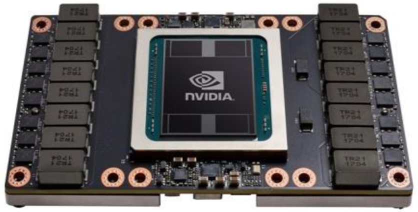
Gunter Roth, Senior Solution Architect gunterr@nvidia.com

Romuald Josien

GPUS FOR HPC AND DEEP LEARNING

Huge demand on compute power (FLOPS)

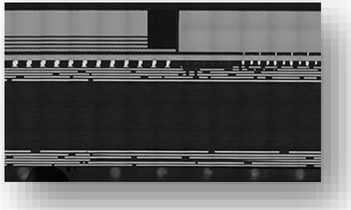
NVIDIA Tesla V100



5120 energy efficient cores + TensorCores
7.8 TF Double Precision (fp64), 15.6 TF Single Precision (fp32) ,
125 Tensor TFLOP/s mixed-precision

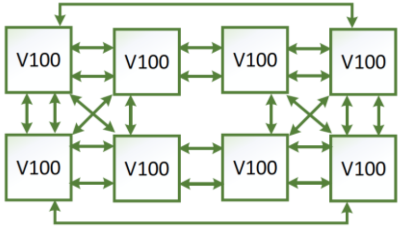
Huge demand on communication and memory bandwidth

CoWoS with HBM2



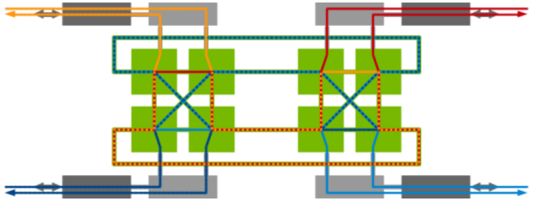
900 GB/s Memory Bandwidth
Unifying Compute & Memory
in Single Package

NVLink



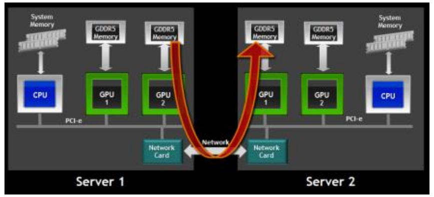
6 links per GPU a 50 GB/s bi-directional for maximum scalability between GPU’s

NCCL



High-performance multi-GPU and multi-node collective communication primitives optimized for NVIDIA GPUs

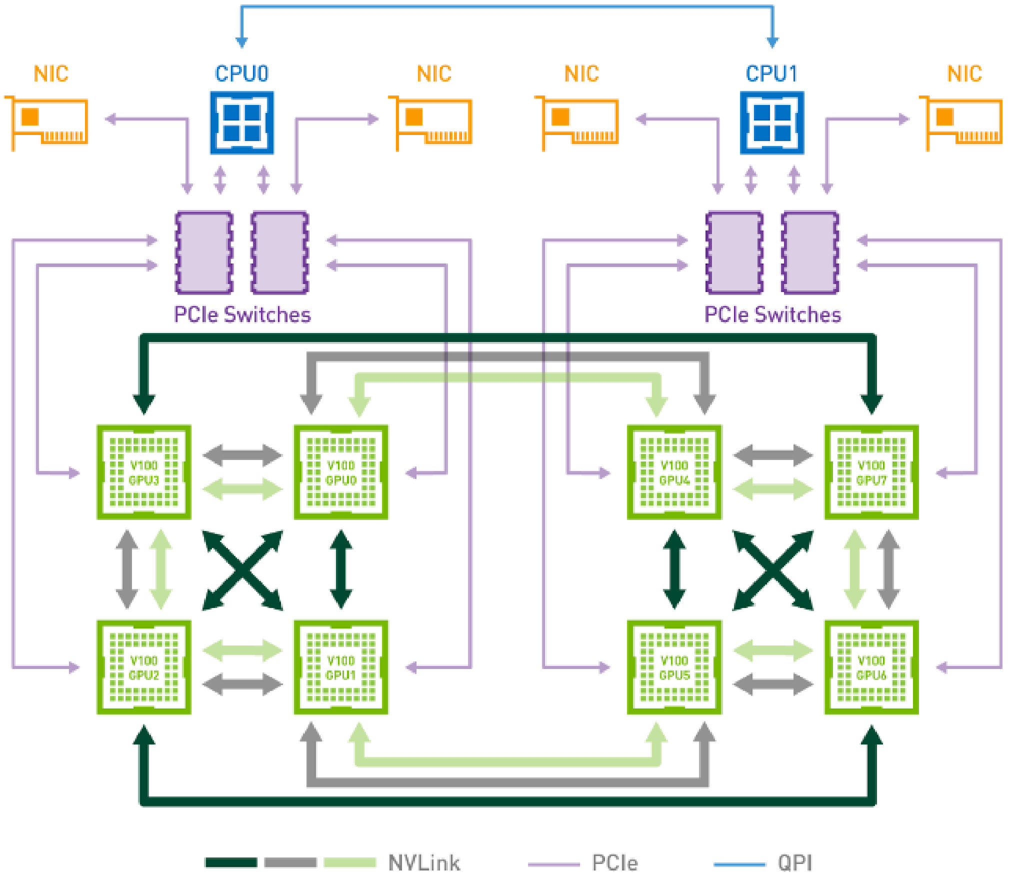
GPU Direct /
GPU Direct RDMA

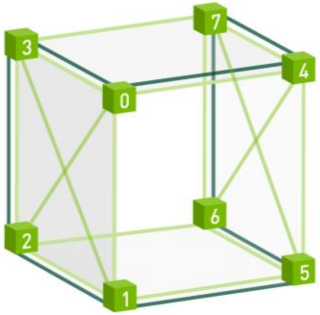


Direct communication between GPUs by eliminating the CPU from the critical path

DL MULTI GPU

DL DATA PARALLELISM – NVLINK

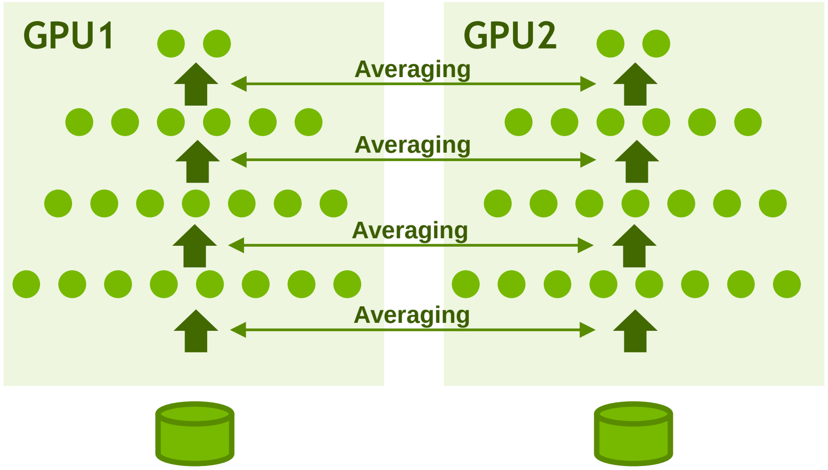




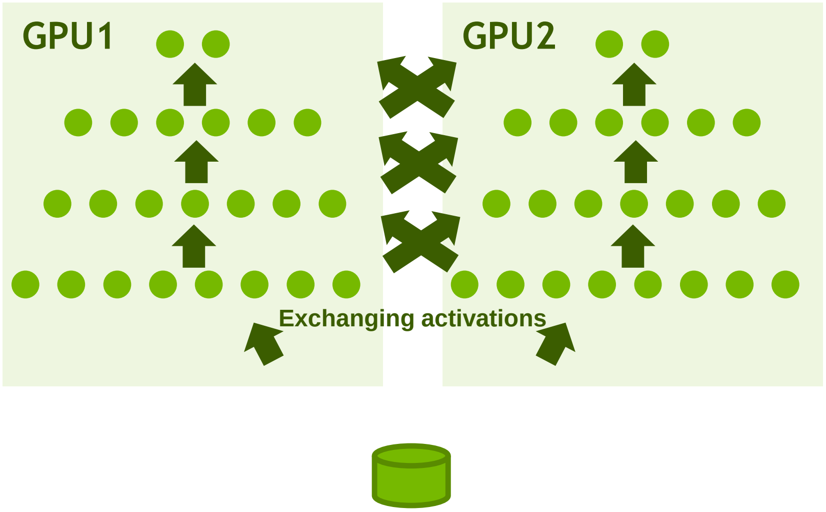
DATA VS MODEL PARALLELISM

Comparison

- Data Parallelism



- Model Parallelism



NVIDIA. DEEP LEARNING INSTITUTE

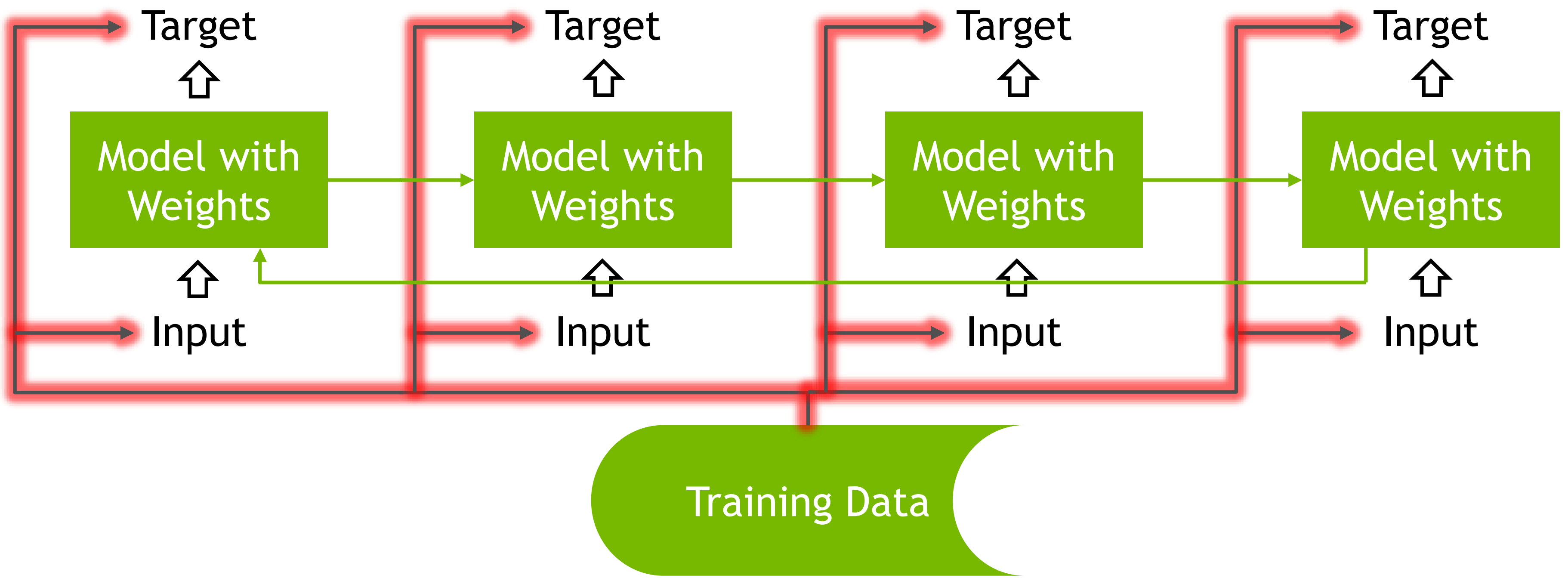
DATA VS MODEL PARALLELISM

Comparison

- Data Parallelism
  - Allows to speed up training
  - All workers train on different data
  - All workers have the same copy of the model
  - Neural network gradients (weight changes) are exchanged

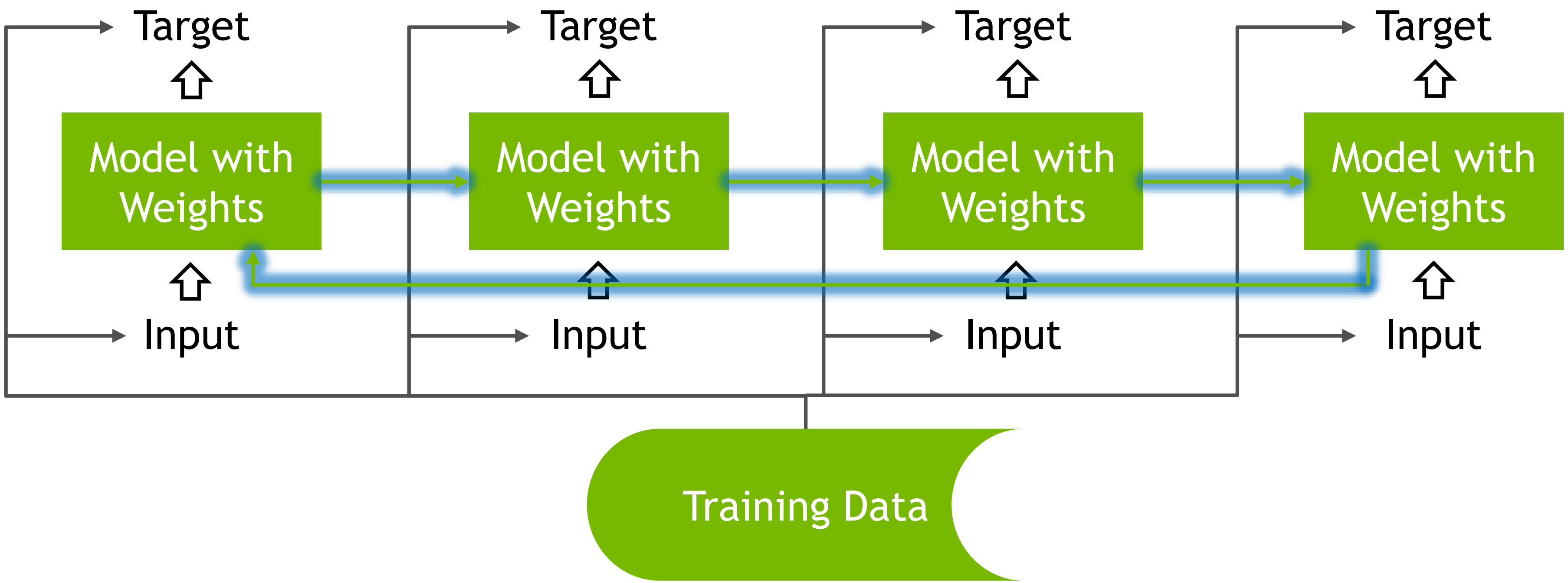
- Model Parallelism
  - Allows for a bigger model
  - All workers train on the same data
  - Parts of the model are distributed across GPUs
  - Neural network activations are exchanged

MULTI GPU DATA PARALLEL DL TRAINING (SYNC SGD)



First load training data (different to each GPU)

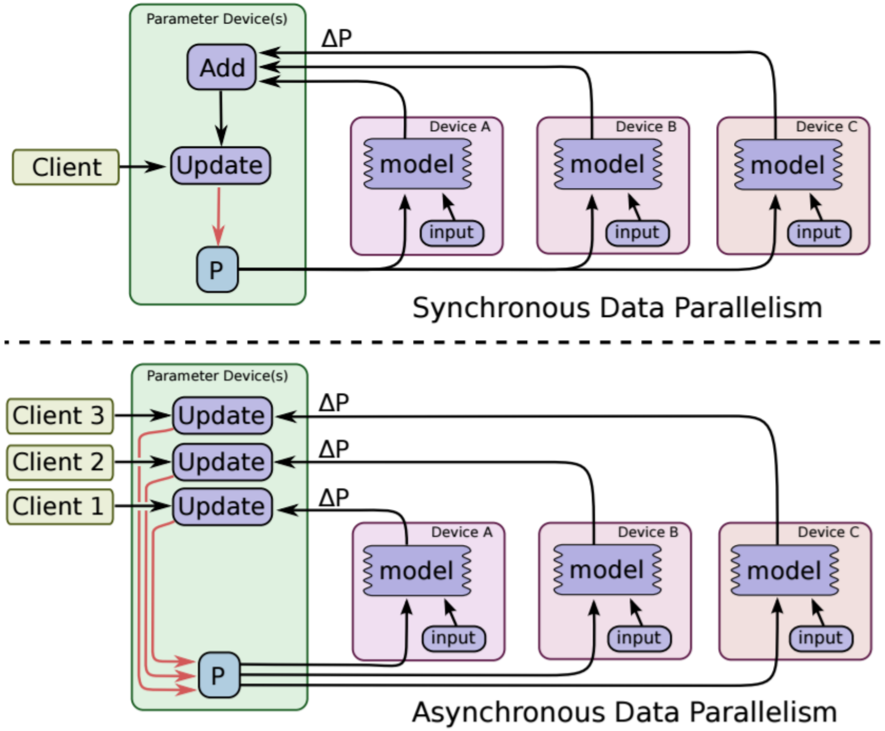
MULTI GPU DATA PARALLEL DL TRAINING



Secondly, synchronize weights between the GPUs

TF DEFAULT IMPLEMENTATION

Parameter server



- Substantial limitations of the parameter server approach:
  - Server becomes a bottleneck
  - Generates substantially more communication than the all reduce based approach
- Makes sense when asynchronous SGD is being used (as all reduce assumes existence of a synchronisation point)

HOROVOD

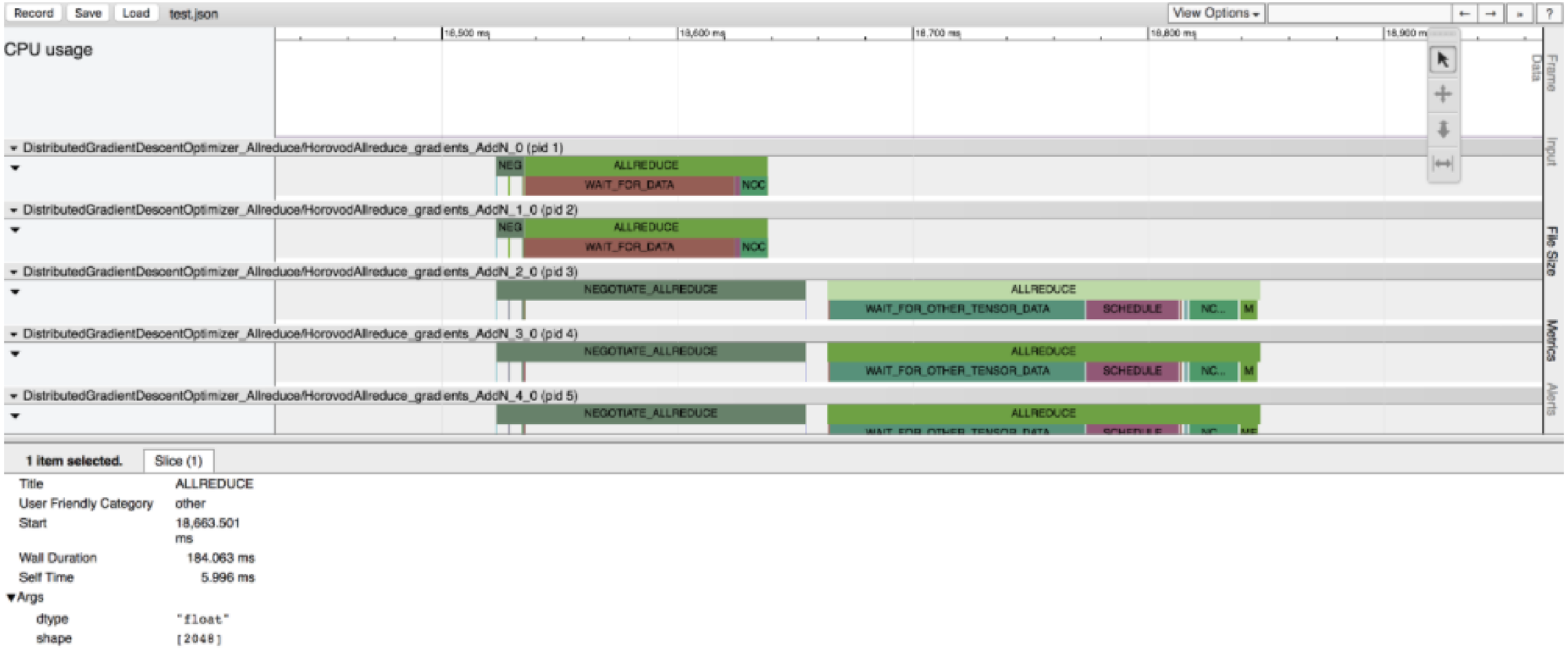
“Making distributed Deep Learning fast and easy to use”

- Leverages TensorFlow + MPI + NCCL 2 to simplify development of synchronous multigpu/multinode TensorFlow
- Instead of Parameter Server architecture leverages MPI and NCCL based all reduce
- Owing to NCCL it leverages features such as:
  - NVLINK
  - RDMA
  - GPUDirectRDMA
  - Automatically detects communication topology
  - Can fall back to PCIe and TCP/IP communication

https://github.com/uber/horovod https://eng.uber.com/horovod/

HOROVOD TIMELINE

Powerful tool to help you understand the performance of your code

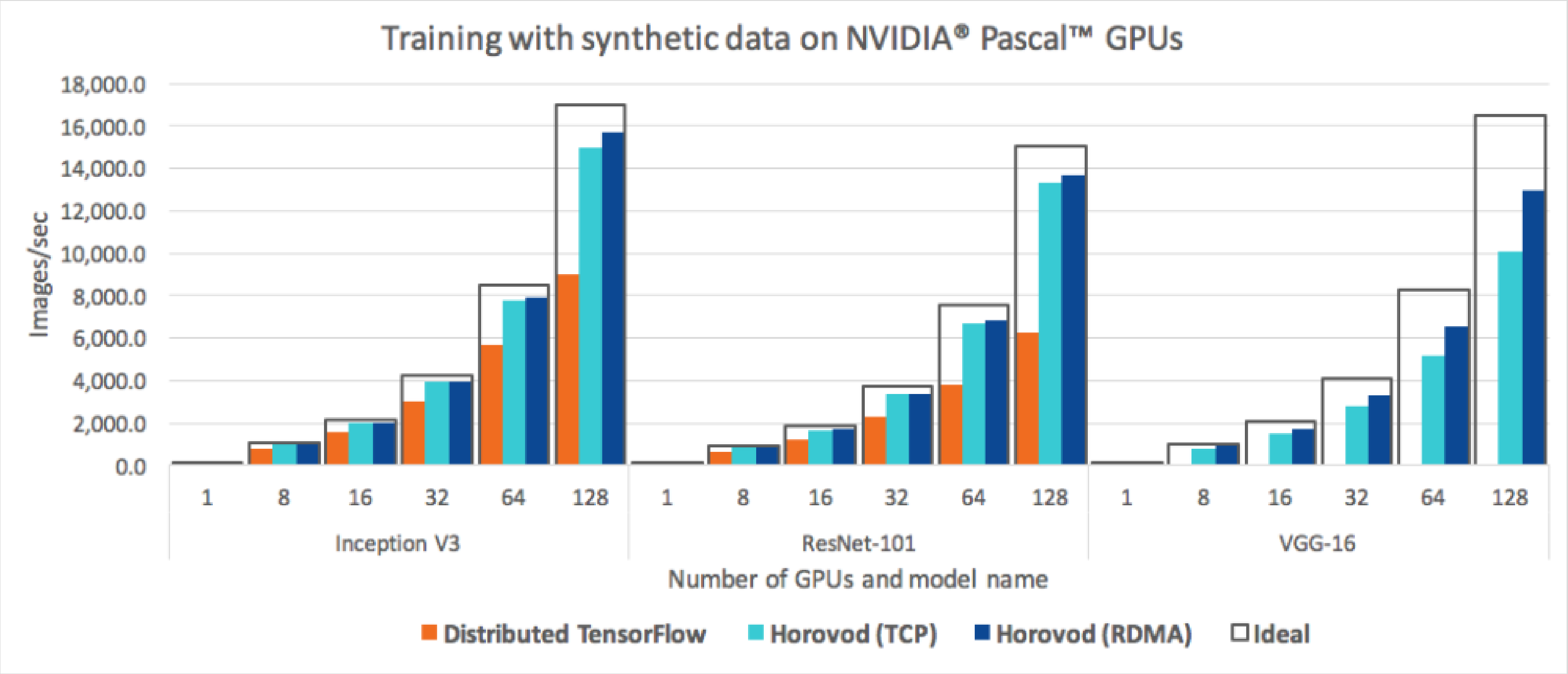


Record
Save
Load
test.json
View Options
18,500 ms
18,600 ms
18,700 ms
18,800 ms
18,900 ms
CPU usage
DistributedGradientDescentOptimizer_Allreduce/HorovodAllreduce_gradients_AddN_0 (pid 1)
NEG
ALLREDUCE
WAIT_FOR_DATA
NOC
DistributedGradientDescentOptimizer_Allreduce/HorovodAllreduce_gradients_AddN_1_0 (pid 2)
NEG
ALLREDUCE
WAIT_FOR_DATA
NOC
DistributedGradientDescentOptimizer_Allreduce/HorovodAllreduce_gradients_AddN_2_0 (pid 3)
NEGOTIATE_ALLREDUCE
ALLREDUCE
WAIT_FOR_OTHER_TENSOR_DATA
SCHEDULE
NC...
M
DistributedGradientDescentOptimizer_Allreduce/HorovodAllreduce_gradients_AddN_3_0 (pid 4)
NEGOTIATE_ALLREDUCE
ALLREDUCE
WAIT_FOR_OTHER_TENSOR_DATA
SCHEDULE
NC...
M
DistributedGradientDescentOptimizer_Allreduce/HorovodAllreduce_gradients_AddN_4_0 (pid 5)
NEGOTIATE_ALLREDUCE
ALLREDUCE
WAIT_FOR_OTHER_TENSOR_DATA
SCHEDULE
NC
ME
Frame Data
Input
File Size
Metrics
Alerts
1 item selected.
Slice (1)

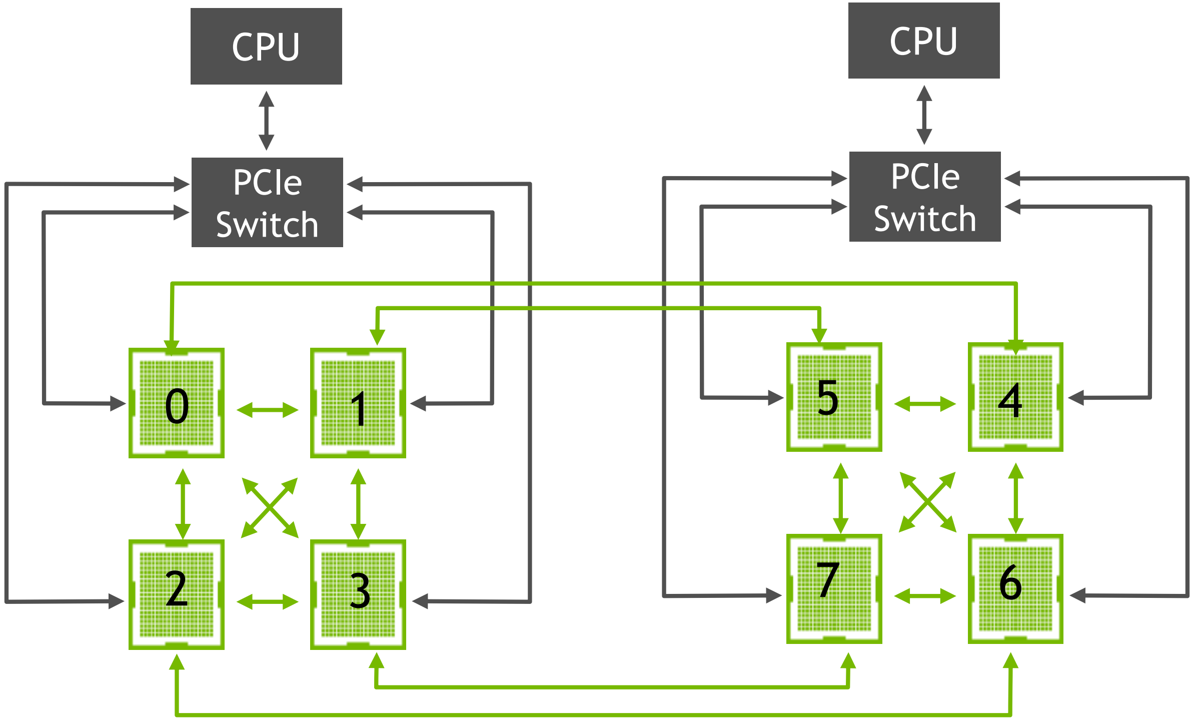
Title	ALLREDUCE
User Friendly Category	other
Start	18,663.501 ms
Wall Duration	184.063 ms
Self Time	5.996 ms
▾Args	
dtype	"float"
shape	[2048]

HOROVOD

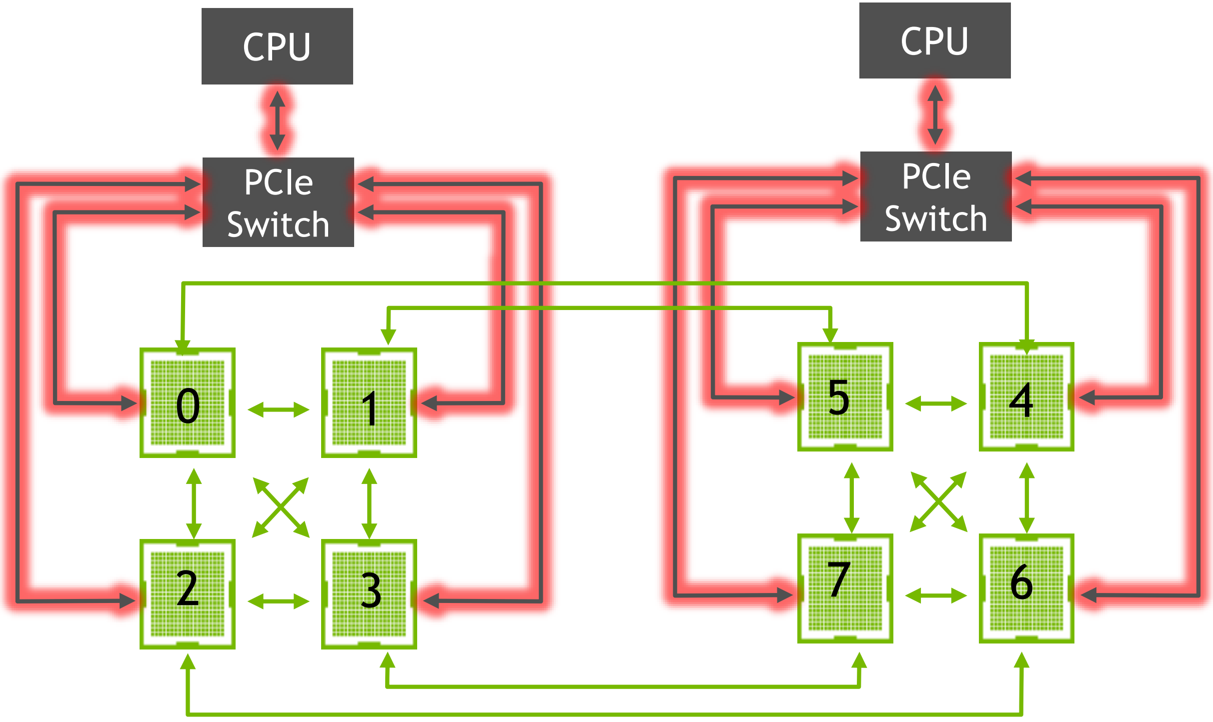
Implications



DL DATA PARALLELISM – NVLINK

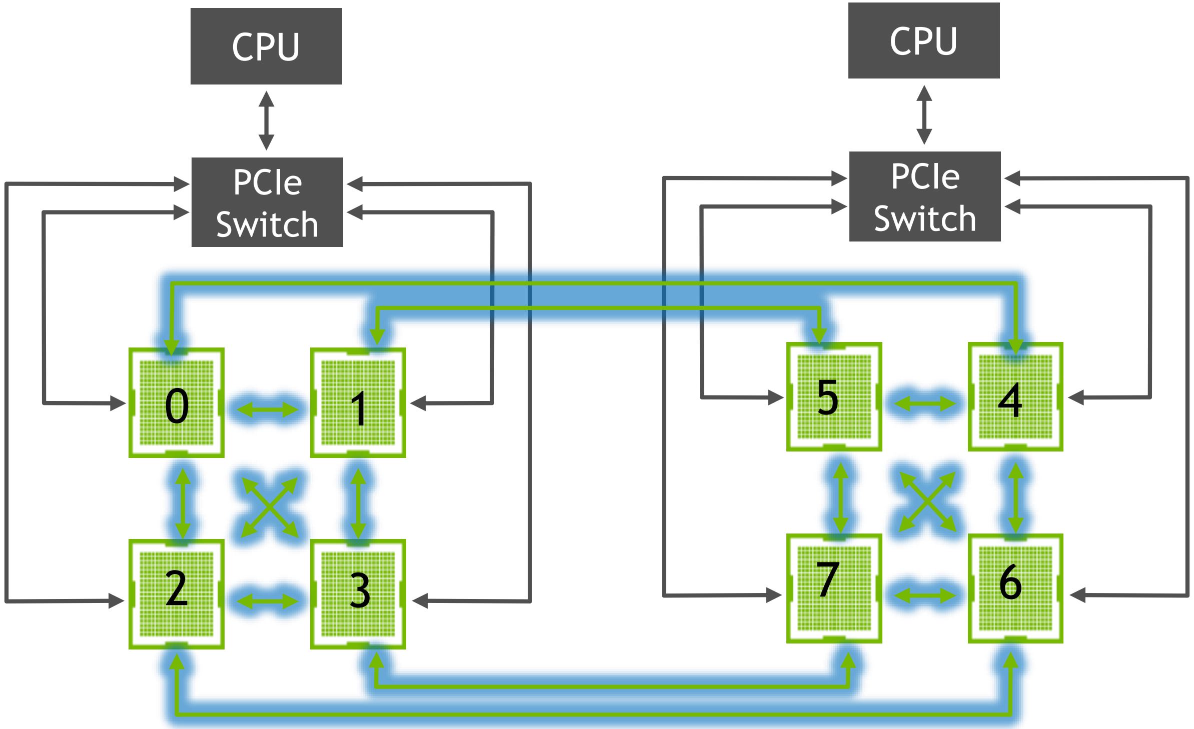


DL DATA PARALLELISM – NVLINK



Data loading over PCIe

DL DATA PARALLELISM – NVLINK



Gradient averaging over NVLink

NCCL

NVIDIA Collective Communications Library (NCCL) 2

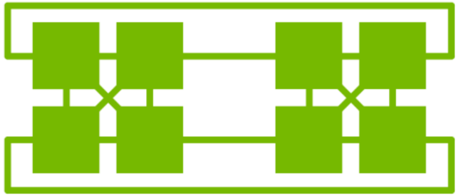
Multi-GPU and multi-node collective communication primitives

High-performance multi-GPU and multi-node collective communication primitives optimized for NVIDIA GPUs

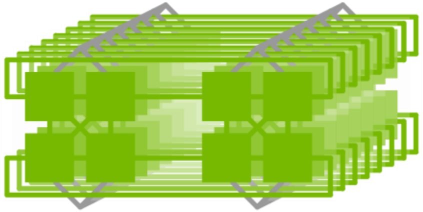
Fast routines for multi-GPU multi-node acceleration that maximizes inter-GPU bandwidth utilization

Easy to integrate and MPI compatible. Uses automatic topology detection to scale HPC and deep learning applications over PCIe and NVink

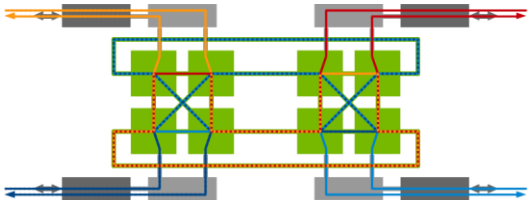
Accelerates leading deep learning frameworks such as Caffe2, Microsoft Cognitive Toolkit, MXNet, PyTorch and more



Multi-GPU:
NVLink
PCIe



Multi-Node:
InfiniBand verbs
IP Sockets

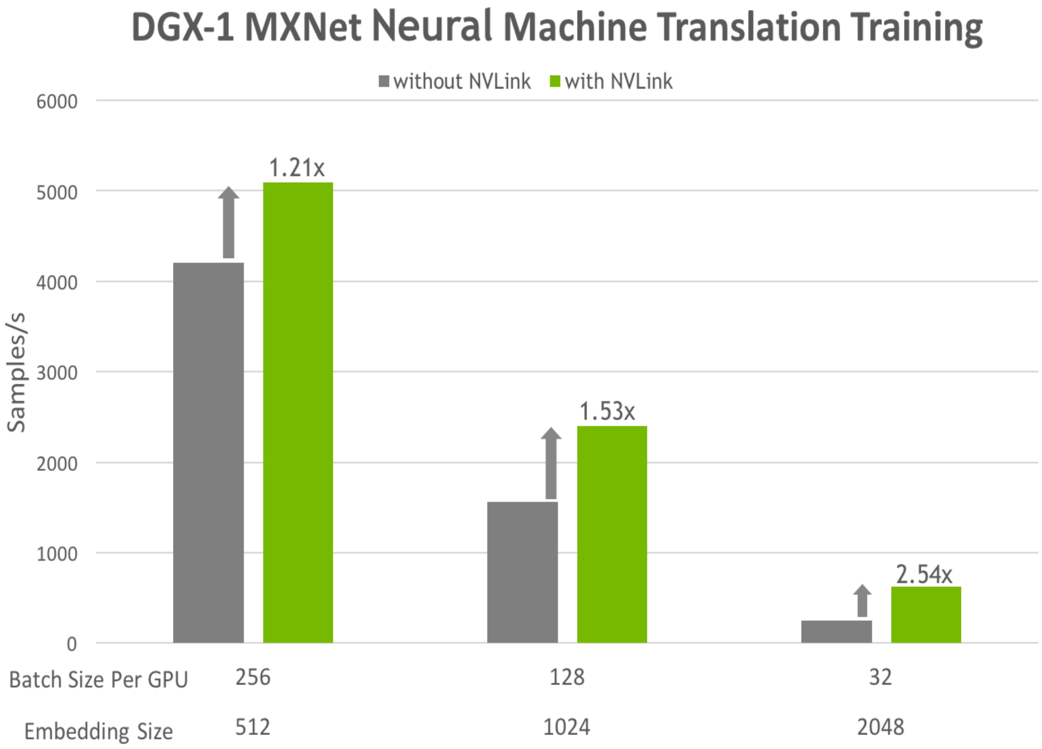


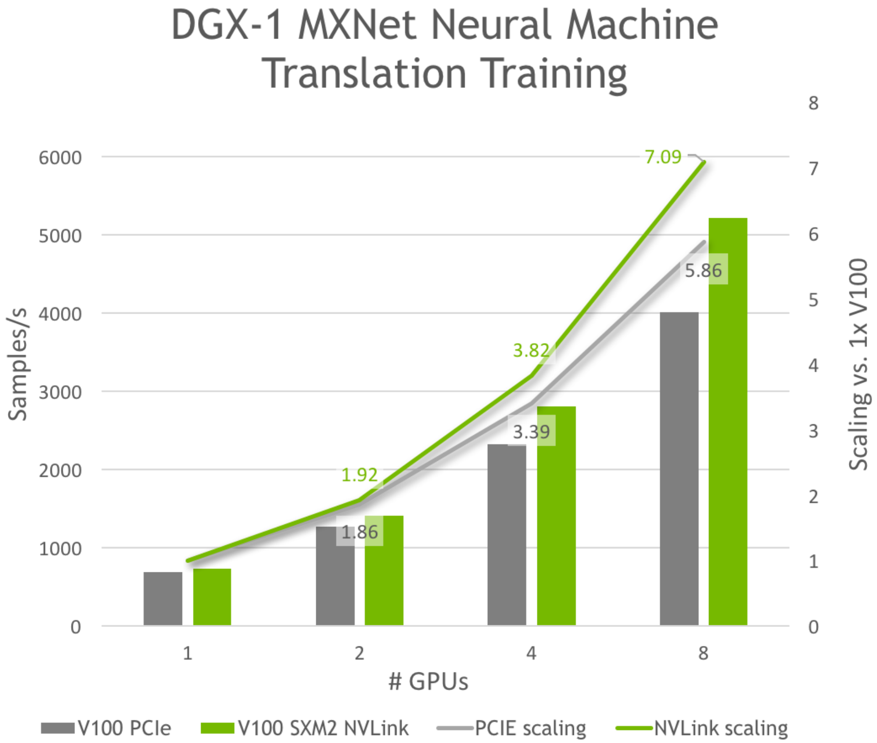
Automatic
Topology
Detection

developer.nvidia.com/nccl

IMPLICATIONS

Requirements highly dependent on the workload





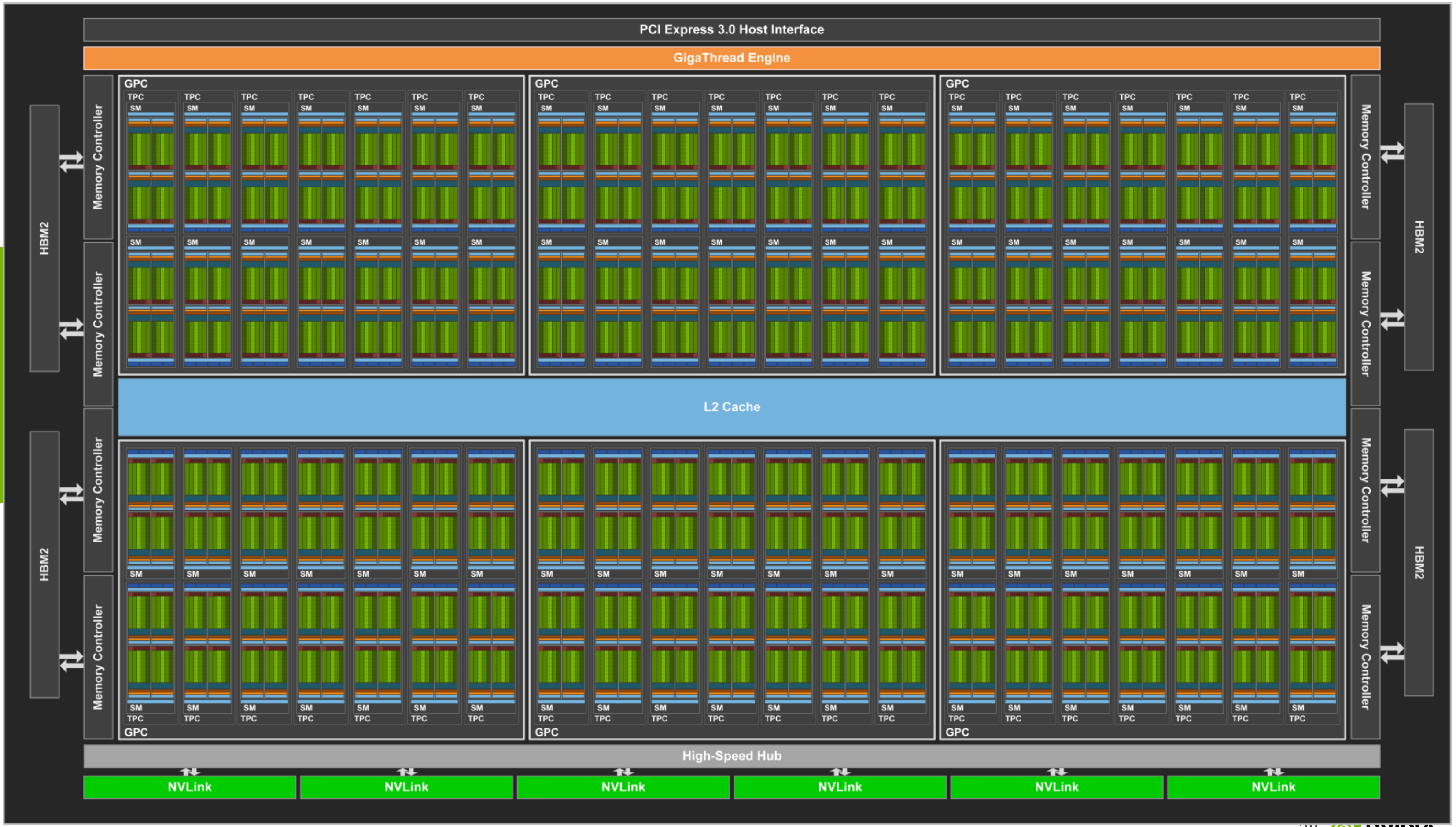
VOLTA AND TURING TENSOR CORE

TESLA V100

21B transistors
815 mm^2

80 SM
5120 CUDA Cores
640 Tensor Cores

16/32 GB HBM2
900 GB/s HBM2
300 GB/s NVLink



*full GV100 chip contains 84 SMs

VOLTA GV100 SM

Redesigned for Productivity

Completely new ISA

Twice the schedulers

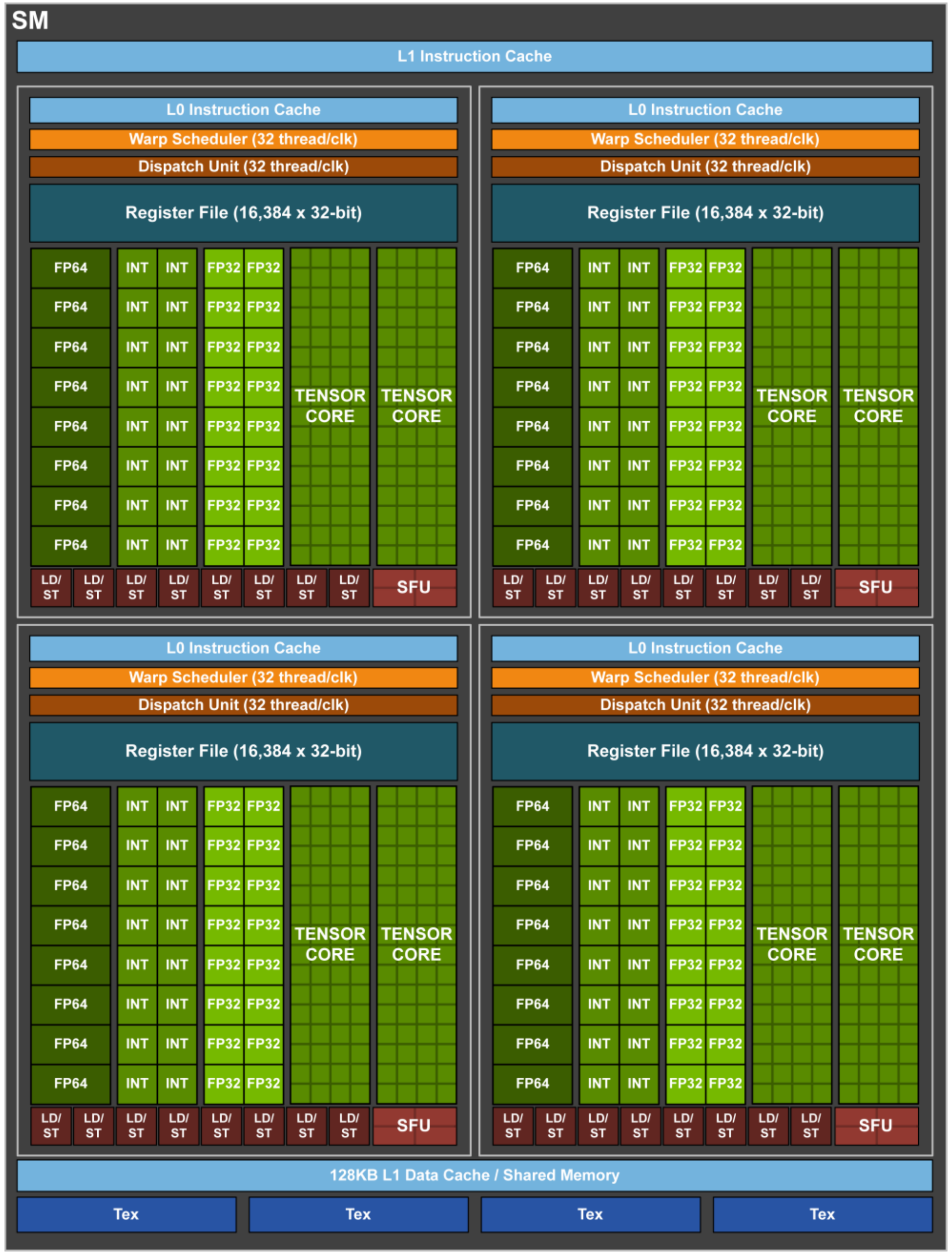
Simplified Issue Logic

Large, fast L1 cache

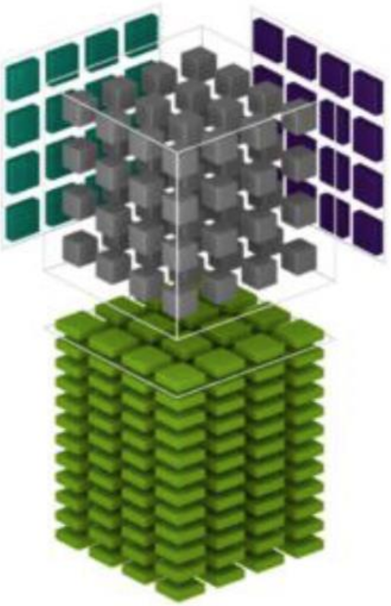
Improved SIMT model

Tensor acceleration

	GP100	GV100
FP32 units	64	64
FP64 units	32	32
INT32 units	NA	64
Tensor Cores	NA	8
Register File	256 KB	256 KB
Unified L1/Shared memory	L1: 24KB Shared: 64KB	128 KB
Active Threads	2048	2048


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex

TURING TENSOR CORE



	FP16	Int8	Int4	Binary
	8x CUDA core	16x CUDA core	32x CUDA core	128x CUDA core
T4	40-70 TFLOPs	80-140 TOPs	160-280 TOPs	640-1100 TOPs
SW at Launch	TensorRT, Libraries		CUTLASS Open Source Tensor Library, CUDA	

TENSOR CORE

Mixed Precision Matrix Math - 4x4 matrices

New CUDA TensorOp instructions & data formats

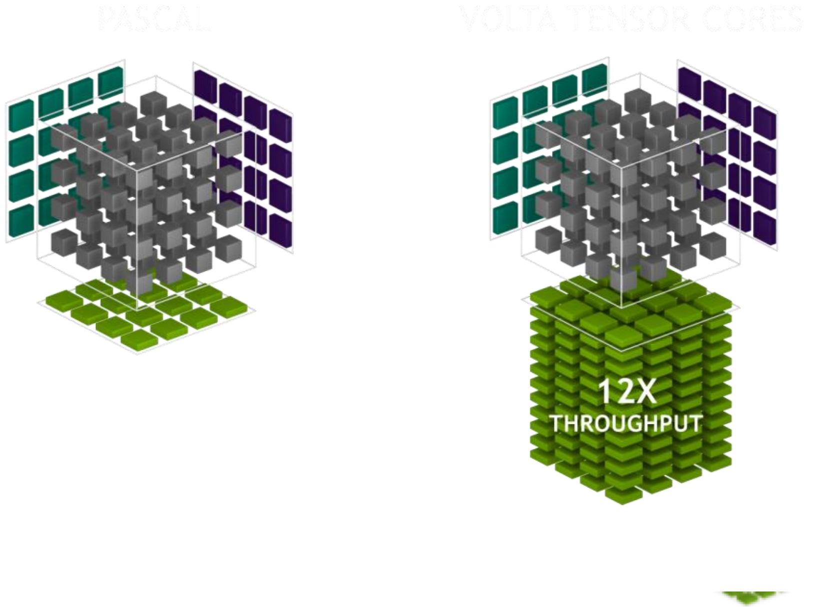
125 Tensor TFLOP/s mixed-precision

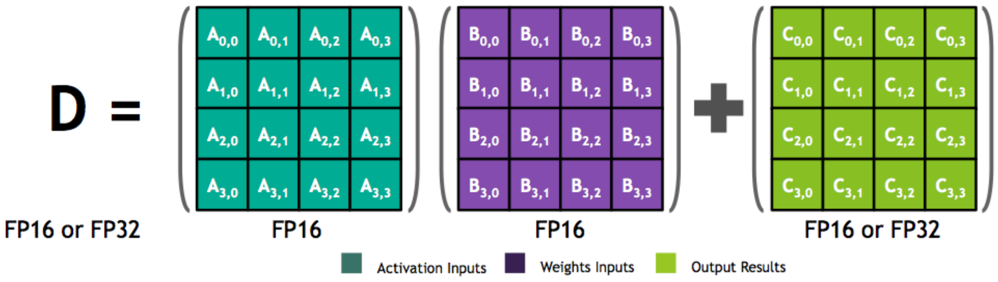
4x4x4 matrix processing array

D[FP32] = A[FP16] * B[FP16] + C[FP32]

Using Tensor cores via

- Volta optimized frameworks and libraries (cuDNN, CuBLAS, TensorRT, ..)
- CUDA C++ Warp Level Matrix Operations



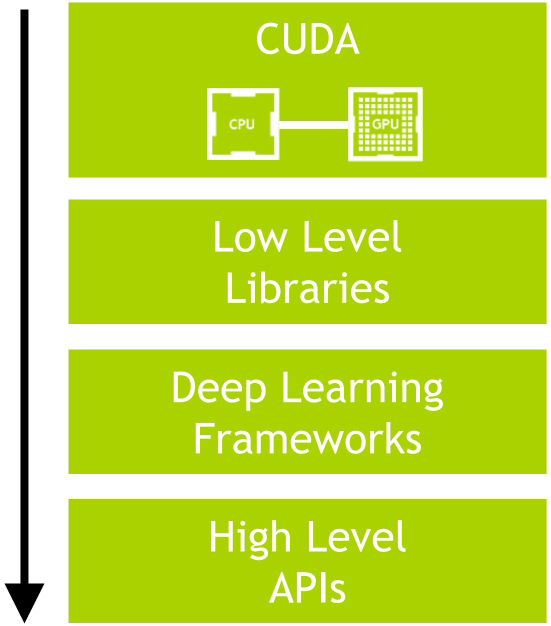


$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0}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

FP16 or FP32 FP16 FP16 FP16 or FP32

■ Activation Inputs ■ Weights Inputs ■ Output Results

HOW TO USE TENSOR CORES



CUDA

Low Level Libraries

Deep Learning Frameworks

High Level APIs

- Exposed as instructions in CUDA under WMMA API (**W**arp **M**atrix **M**ultiply **A**ccumulate)
- Used by cuDNN, cuBLAS, CUTLASS to accelerate matrix multiplications and convolution
- Tensor Core kernels used implicitly on FP16 ops from DL frameworks PyTorch / TensorFlow / etc…
- High-level tools (e.g. PyTorch Apex) convert everything automatically and safely

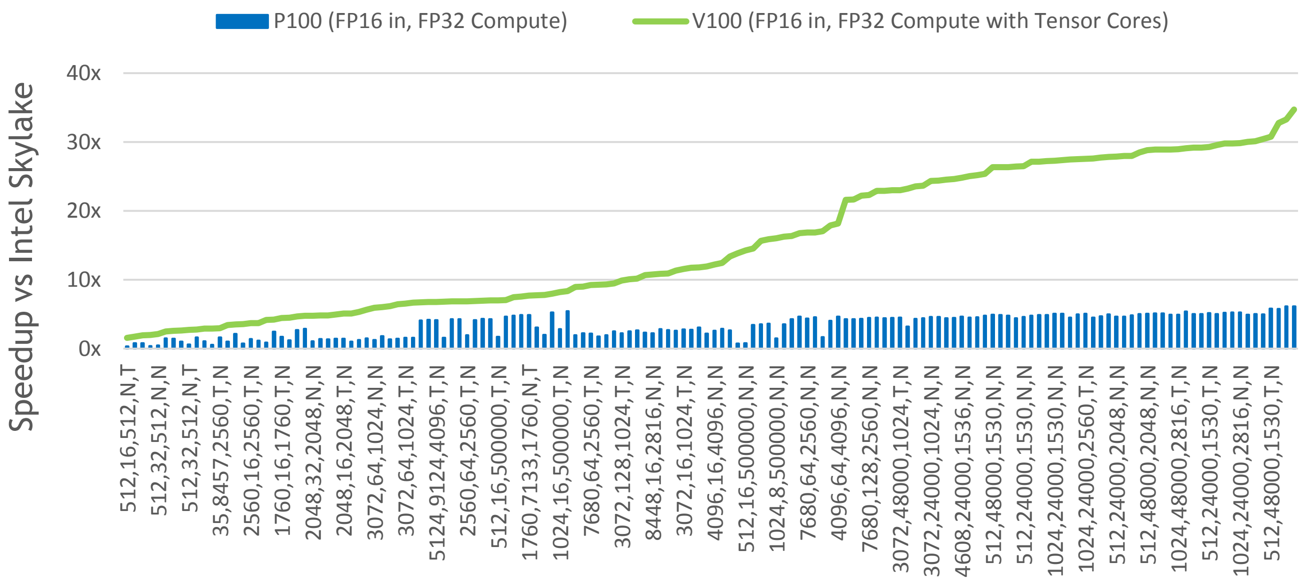
cuBLAS 10.0

Optimized GEMM Performance for Deep Learning

- Turing optimized GEMMs & GEMM extensions for Tensor Cores
- GEMM Performance Tuned for sizes used in various DL models
- API and Error Logging for debug and traceability

https://developer.nvidia.com/cublas

Up to 90TF of Deepbench GEMM Performance



Deepbench training performance runs with Tesla P100, Tesla V100 and Intel Skylake 6140 Gold 2.3 GHz with hyperthreading off

CUBLAS EXAMPLE

```
// First, create a cuBLAS handle:
cublasStatus_t cublasStat = cublasCreate(&handle);
// Set the math mode to allow cuBLAS to use Tensor Cores:
cublasStat = cublasSetMathMode(handle, CUBLAS_TENSOR_OP_MATH);

// Allocate and initialize your matrices (only the A matrix is shown):
size_t matrixSizeA = (size_t)rowsA * colsA; T_ELEM_IN **devPtrA = 0;
cudaMalloc((void**)&devPtrA[0], matrixSizeA * sizeof(devPtrA[0][0]));
T_ELEM_IN A  = (T_ELEM_IN *)malloc(matrixSizeA * sizeof(A[0]));
memset( A, 0xFF, matrixSizeA* sizeof(A[0]));

status1 = cublasSetMatrix(rowsA, colsA, sizeof(A[0]), A, rowsA, devPtrA[i], rowsA);

// ... allocate and initialize B and C matrices (not shown) ...
// Invoke the GEMM, ensuring k, lda, ldb, and ldc are all multiples of 8,
// and m is a multiple of 4:
cublasStat = cublasGemmEx(handle, transa, transb, m, n, k, alpha, A, CUDA_R_16F,
                                                             lda, B, CUDA_R_16F,
                                                             ldb, beta, C, CUDA_R_16F,
                                                             ldc, CUDA_R_32F, algo);
```

NVIDIA cuDNN 7

Deep Learning Primitives

High performance building blocks for deep learning frameworks

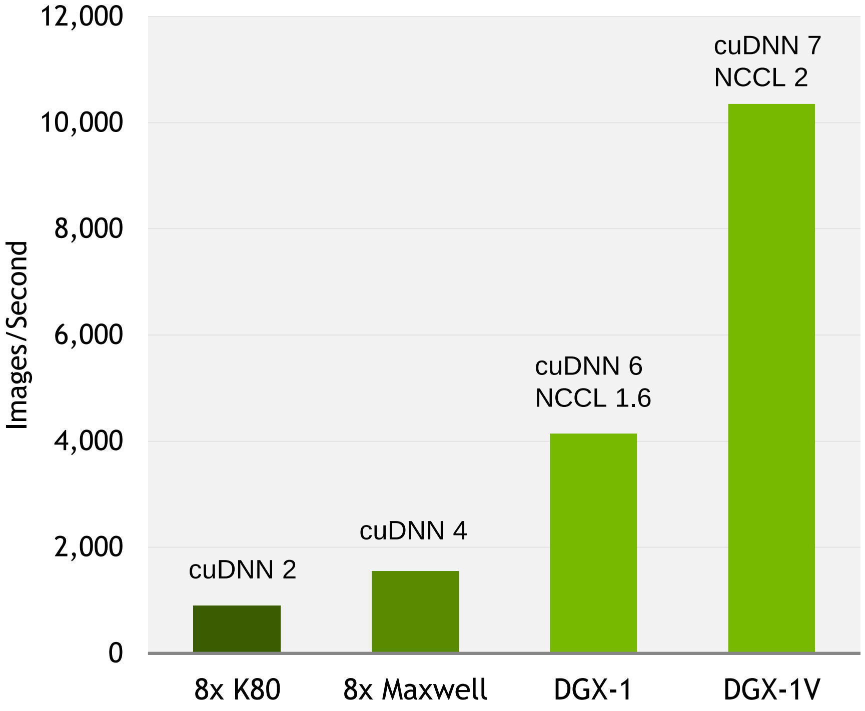
- Drop-in acceleration for widely used deep learning frameworks such as Caffe2, Microsoft Cognitive Toolkit, PyTorch, Tensorflow, Theano and others

- Accelerates industry vetted deep learning algorithms, such as convolutions, LSTM RNNs, fully connected, and pooling layers

- Fast deep learning training performance tuned for NVIDIA GPUs

developer.nvidia.com/cudnn

Deep Learning Training Performance



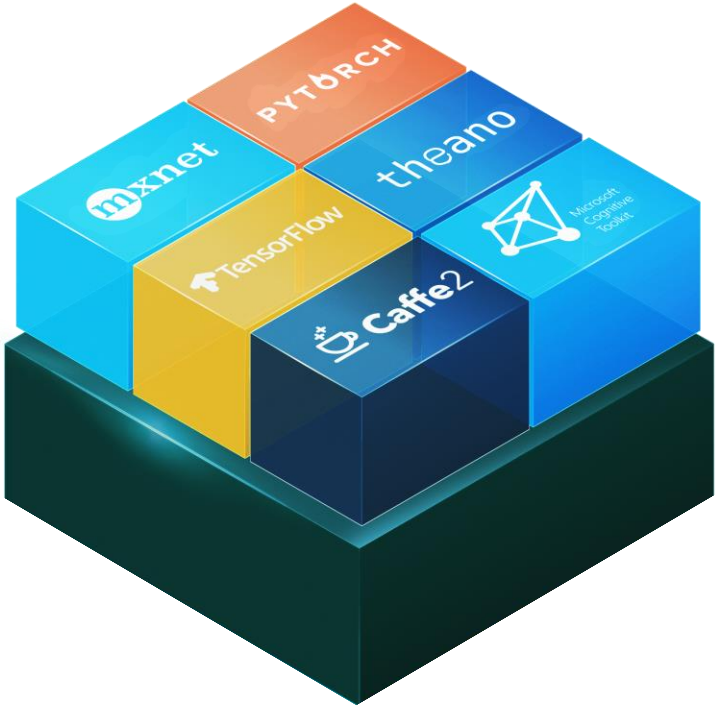
> “NVIDIA has improved the speed of cuDNN with each release while extending the interface to more operations and devices at the same time.”
>
> *— Evan Shelhamer, Lead Caffe Developer, UC Berkeley*

CUDNN EXAMPLE

```
// Set the compute data type (below as CUDNN_DATA_FLOAT):
checkCudnnErr( cudnnSetConvolutionNdDescriptor(cudnnConvDesc,
                                               convDim,
                                               padA,
                                               convstrideA,
                                               dilationA,
                                               CUDNN_CONVOLUTION,
                                               CUDNN_DATA_FLOAT) );

// Set the math type to allow cuDNN to use Tensor Cores:
checkCudnnErr( cudnnSetConvolutionMathType(cudnnConvDesc,
CUDNN_TENSOR_OP_MATH) );

// Choose a supported algorithm:
cudnnConvolutionFwdAlgo_t algo =
CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM;
```

CUDA AND CUTLASS

USING TENSOR CORES



NVIDIA cuDNN, cuBLAS, TensorRT

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

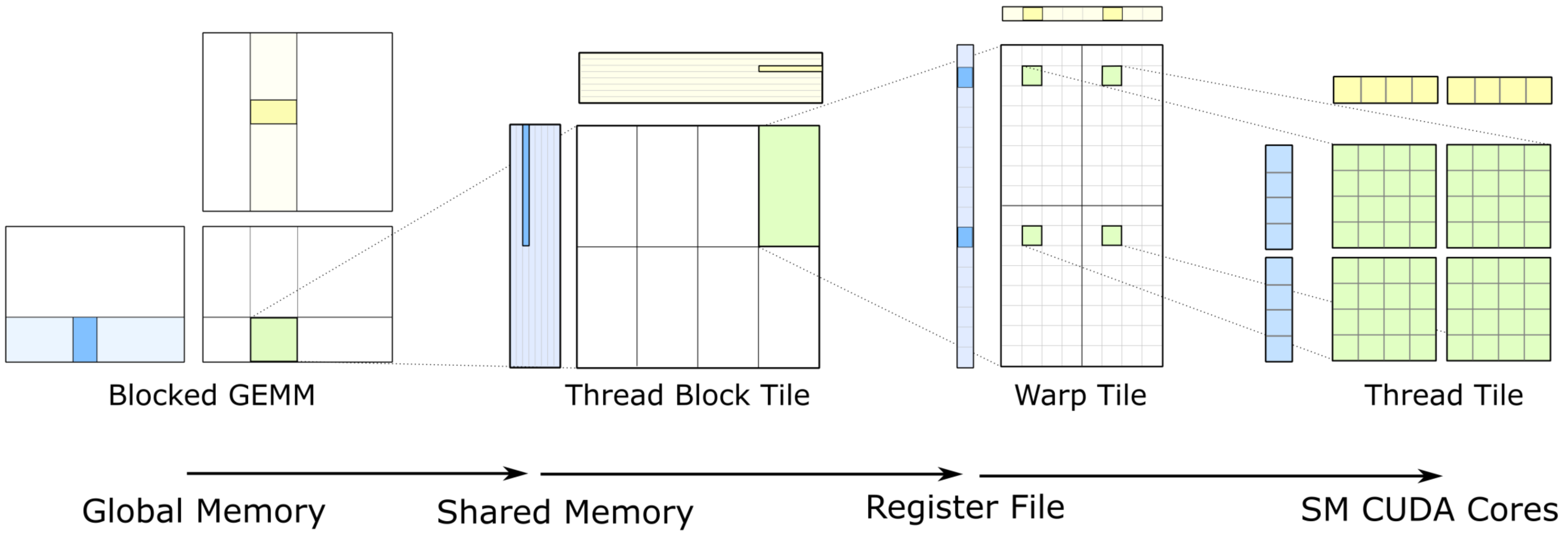
DESIGN OBJECTIVES

Span the Design Space with Generic Programming

CUDA C++ templates for composable algorithms

Performance: Implement efficient dense linear algebra kernels

Structured, reusable components: flexibility and productivity

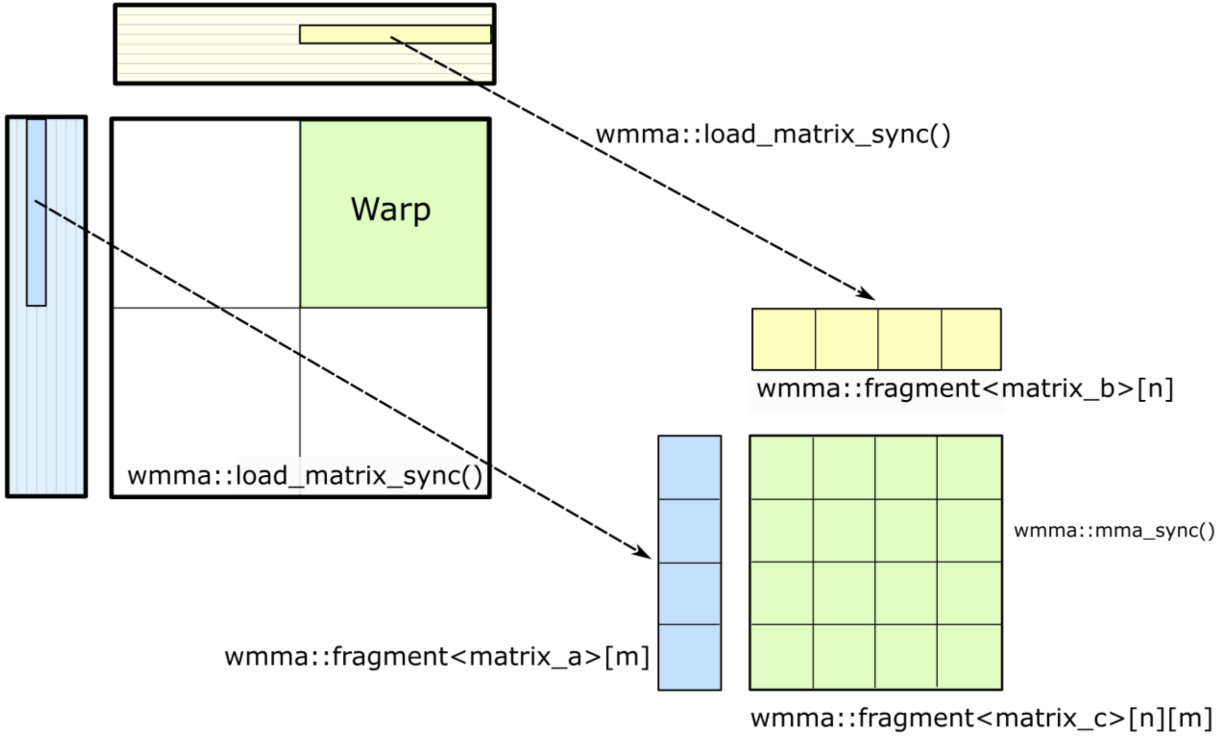


EXAMPLE: VOLTA TENSOR CORES

Targeting the CUDA WMMA API

WMMA: Warp-synchronous Matrix Multiply-Accumulate

- API for issuing operations to Volta Tensor Cores



wmma::load_matrix_sync()
Warp
wmma::load_matrix_sync()
wmma::fragment<matrix_b>[n]
wmma::mma_sync()
wmma::fragment<matrix_a>[m]
wmma::fragment<matrix_c>[n][m]

```
/// 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

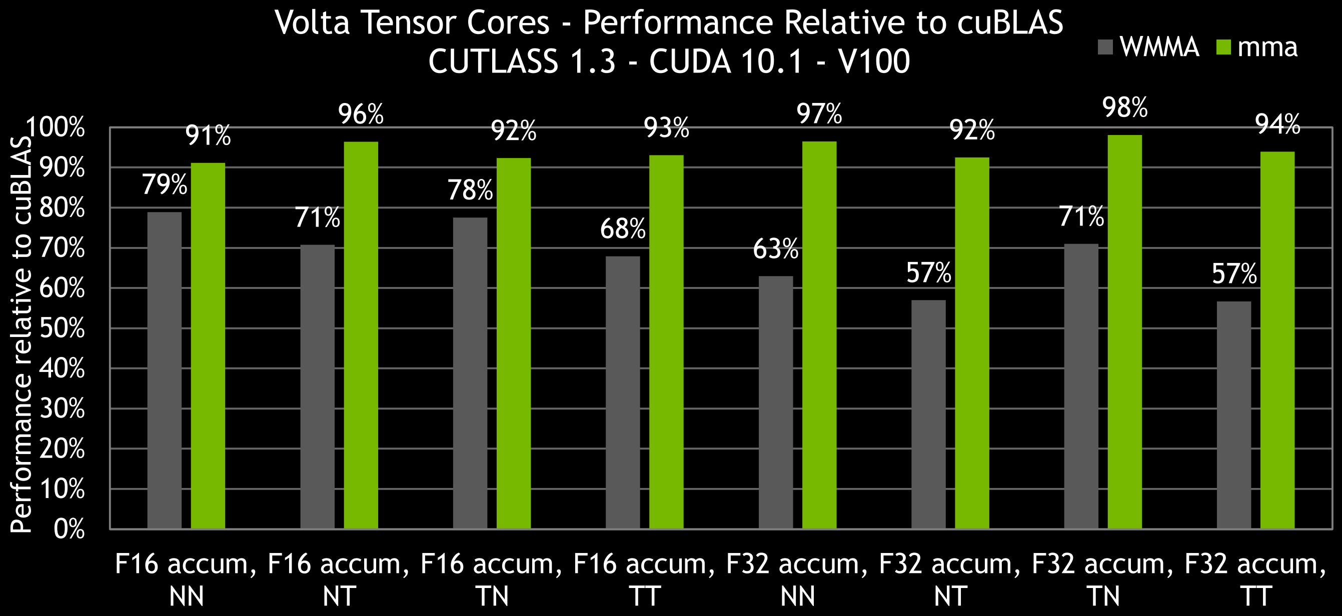
GEMM kernels targeting Volta Tensor Cores natively with **mma.sync**

Template abstractions for high-performance matrix-multiplication

Header-only open source library

Thread-wide, warp-wide, block-wide, and device-wide primitives

Specialized data-movement and multiply-accumulate abstractions



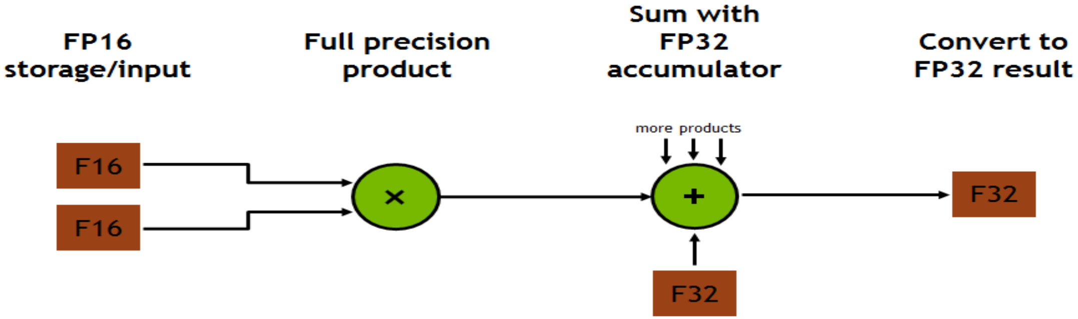
Volta Tensor Cores - Performance Relative to cuBLAS
CUTLASS 1.3 - CUDA 10.1 - V100

New in CUDA 10.1 & CUTLASS 1.3: **mma.sync**

- PTX assembly instruction enables maximum efficiency of Volta Tensor Cores operation

TENSOR CORES FOR AI

- Simple trick for 2x to 5x faster deep learning training
  - Accomplished in few lines of code
  - Models can use same hyperparameters
  - Models converge to same accuracy
- Half the memory traffic and storage enabling larger batch sizes
- AI community is trending towards low precision as common practice



MIXED PRECISION TRAINING

- **Make an FP16 copy of the weights**
- Forward propagate using FP16 weights and activations
- **Multiply the resulting loss by the scale factor S**
- Backward propagate using FP16 weights, activations, and their gradients
- **Multiply the weight gradients by 1/S**
- Optionally process the weight gradients (gradient clipping, weight decay, etc.)
- Update the master copy of weights in FP32

1. MODEL CONVERSION

- Make simple type updates to each layer:
  - Use FP16 values for the weights and inputs

```
# PyTorch
layer = torch.nn.Linear(in_dim, out_dim).half()

# TensorFlow
layer = tf.layers.dense(tf.cast(inputs, tf.float16), out_dim)
```

2. MASTER WEIGHTS

- FP16 alone is sufficient for some networks but not others; keep FP32 copy of weights

```
param = torch.cuda.FloatTensor([1.0])
print(param + 0.0001)
```

1.0001

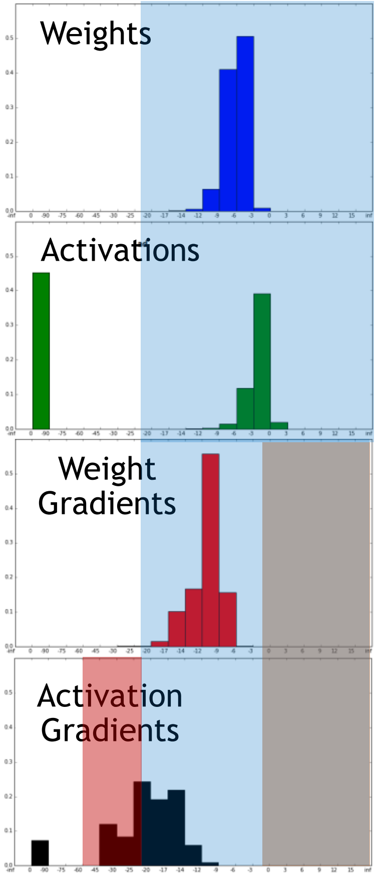
```
param = torch.cuda.HalfTensor([1.0])
print(param + 0.0001)
```

1

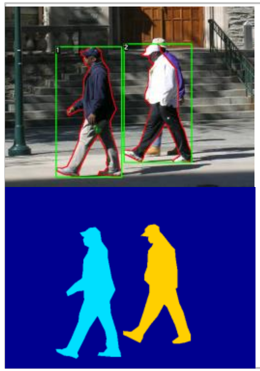
When *update/param* < 2^{-11}, updates have no effect.

3. LOSS SCALING

- Range representable in FP16: ~40 powers of 2
- Gradients are small:
    - Some lost to zero
    - While ~15 powers of 2 remain unused
- Loss scaling:
    - Multiply loss by a constant S
    - All gradients scaled up by S (chain rule)
    - Unscale weight gradient (in FP32) before weight update



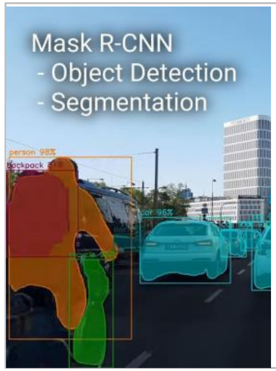
AUTOMATIC MIXED PRECISION ADOPTION IN PYTORCH COMMUNITY



TORCHVISION

Popular datasets, model architectures, common image transformations for computer vision

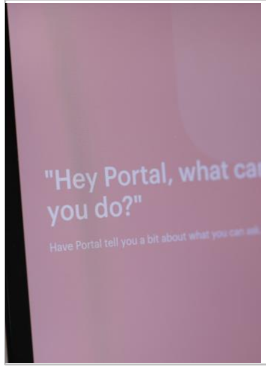
Facebook
PyTorch



MASK R-CNN

Fast, modular reference implementation of Instance Segmentation and Object Detection

Facebook Research
PyTorch



PYTEXT

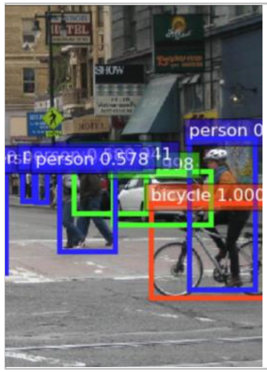
A natural language modeling framework

Facebook Research
PyTorch

PIX2PIXHD

Synthesizing and manipulating 2048x1024 images with conditional GANs

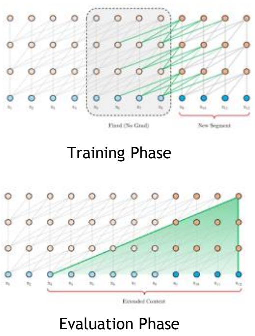
NVIDIA
PyTorch



GLUON-CV

Deep learning toolkit for computer vision

DMLC
MXNet



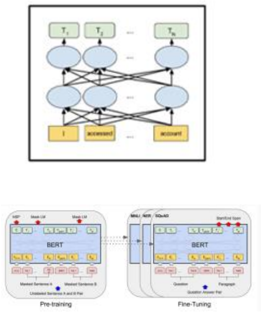
TRANSFORMER-XL

Attentive language models beyond a fixed-length context

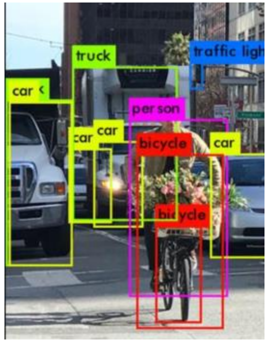
CMU & Google
PyTorch

Link to NVIDIA Developer Page
Link to Github NVIDIA DL Examples

AUTOMATIC MIXED PRECISION ADOPTION IN TENSORFLOW COMMUNITY

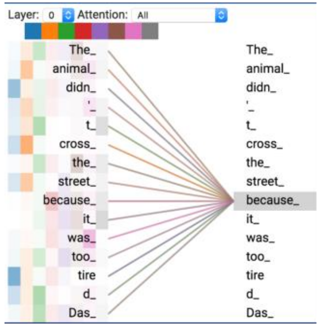


BERT [TF2.0]
Unsupervised, deep-bidirectional encoders to deliver SOTA performance on many NLP tasks
NVIDIA
TensorFlow

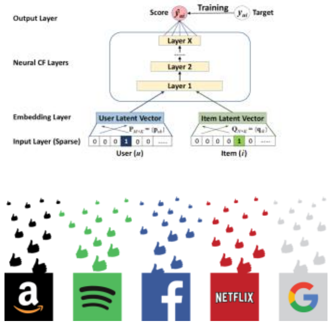


Resnet-50 [TF2.0]
Image classification model based on residual blocks

NVIDIA
TensorFlow

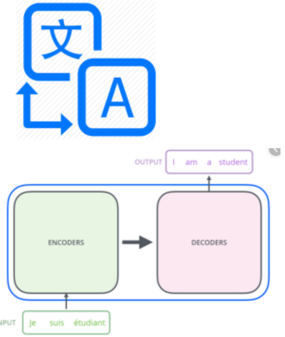


Tensor2Tensor
Library of deep learning models & datasets maintained by Google Brain team
NVIDIA
TensorFlow



NCF [TF2.0]
Neural Network based recommendation engine

NVIDIA
TensorFlow



Transformer [TF2.0]
Image classification model based on residual blocks

NVIDIA
TensorFlow

Sonnet

Sonnet
Library on top of TF designed to provide simple,composable abstractions for ML research.

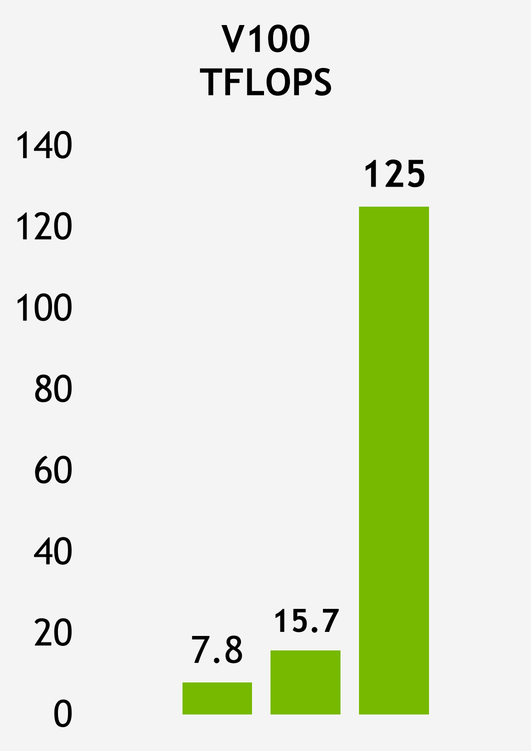
NVIDIA
TensorFlow

TENSOR CORES FOR HPC

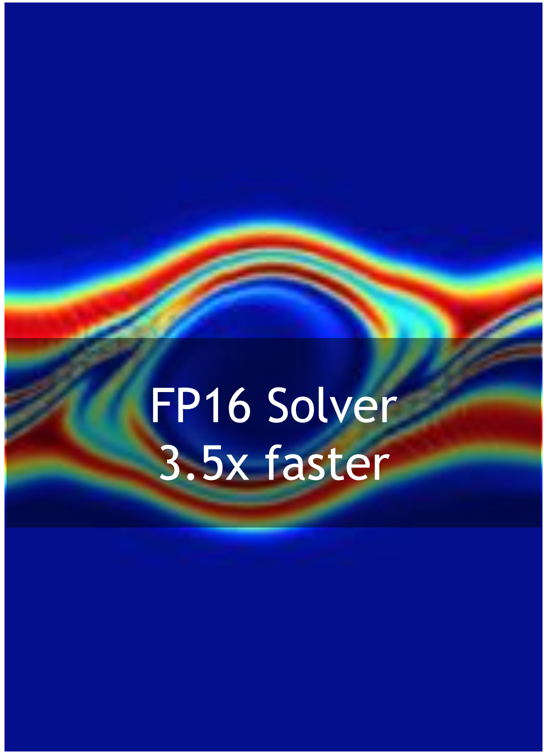
- Mixed precision algorithms are increasingly popular
  - It is common to combine double + single precision, or floating point + integer
- Similar to AI:
  - Use low precision to reduce memory traffic and storage
  - Use Tensor Core instructions for large speedups

TENSOR CORES FOR SCIENCE

Mixed-Precision Computing



FP64+ MULTI-PRECISION

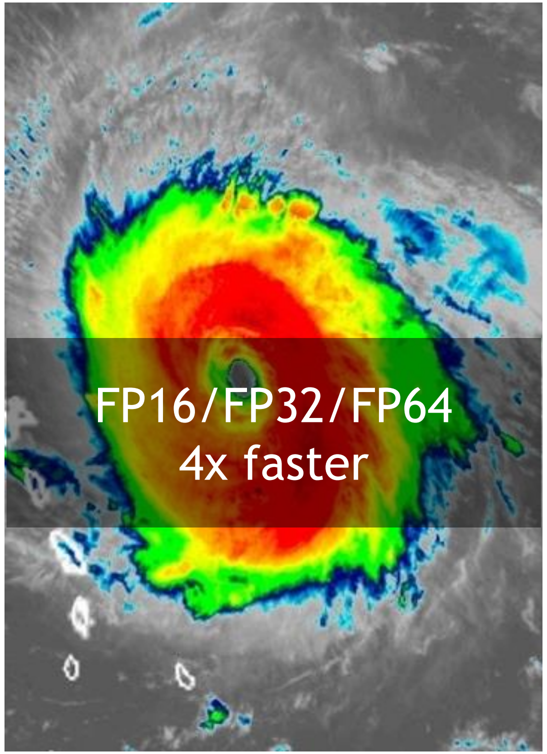


FP16 Solver
3.5x faster

PLASMA FUSION APPLICATION

FP16-FP21-FP32-FP64
25x faster

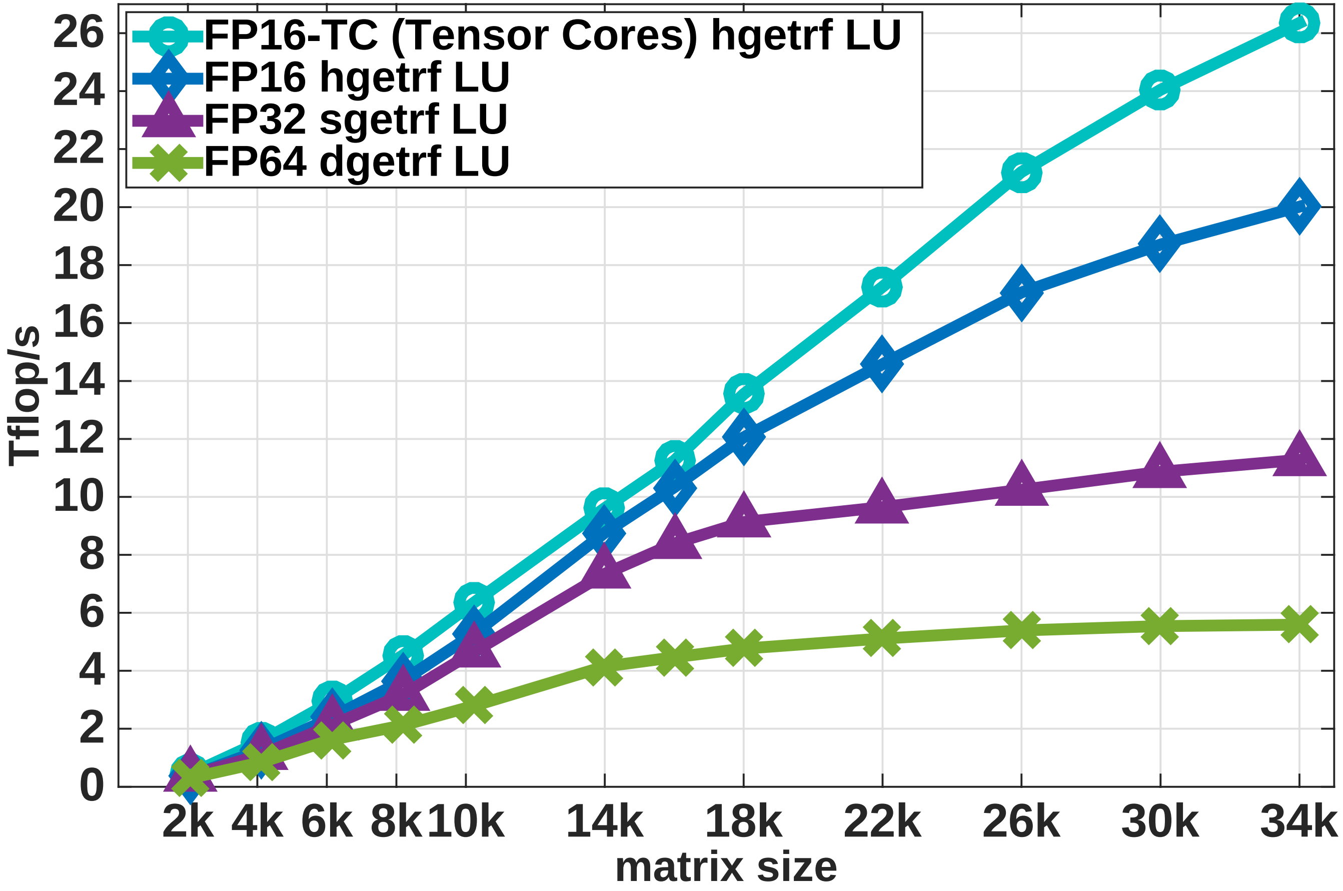
EARTHQUAKE SIMULATION



FP16/FP32/FP64
4x faster

MIXED PRECISION WEATHER PREDICTION

LINEAR ALGEBRA + TENSOR CORES



Double Precision LU Decomposition

- Compute initial solution in FP16
- Iteratively refine to FP64

Achieved FP64 Tflops: 26

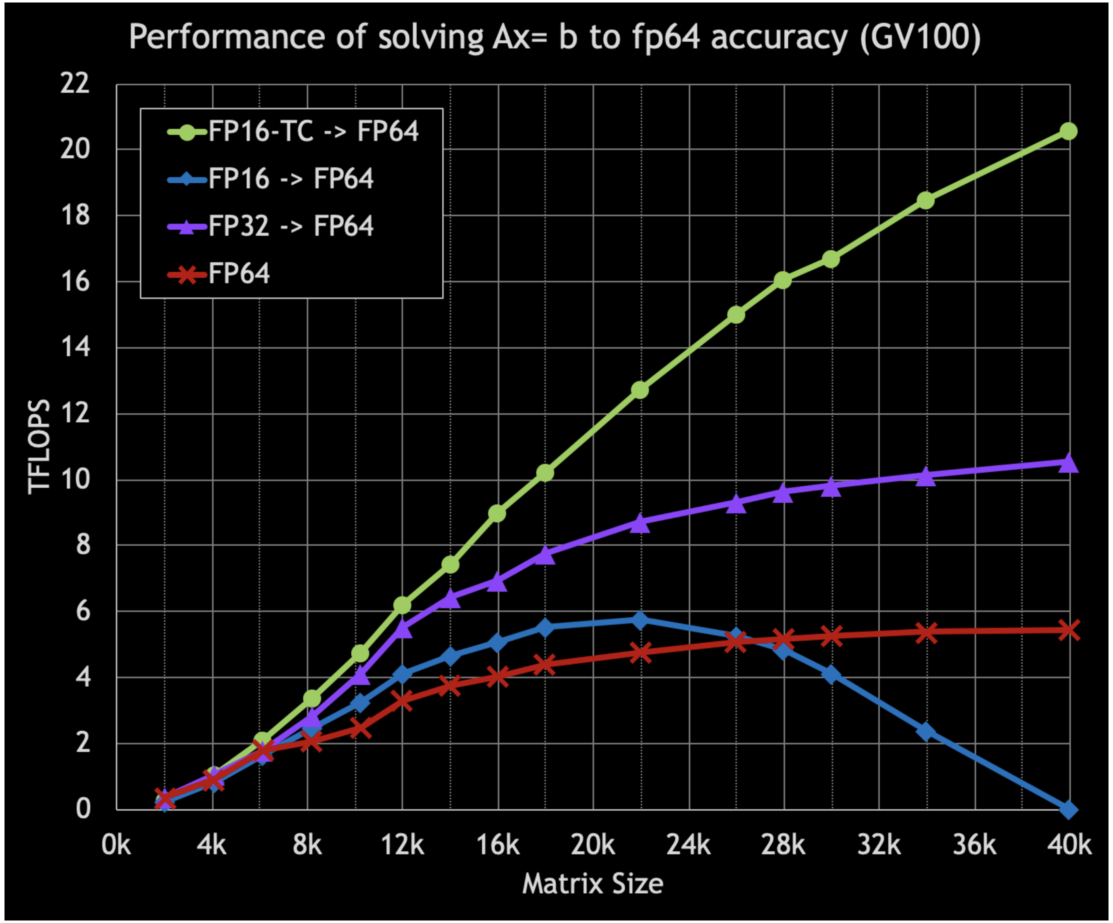
Device FP64 Tflops: 7.8

Data courtesy of: Azzam Haidar, Stan. Tomov & Jack Dongarra, Innovative Computing Laboratory, University of Tennessee
“*Investigating Half Precision Arithmetic to Accelerate Dense Linear System Solvers*”, A. Haidar, P. Wu, S. Tomov, J. Dongarra, SC’17
GTC 2018 Poster P8237: *Harnessing GPU’s Tensor Cores Fast FP16 Arithmetic to Speedup Mixed-Precision Iterative Refinement Solves*

INNOVATIVE COMPUTING LABORATORY MAGMA

THE UNIVERSITY OF TENNESSEE

TENSOR CORE-ACCELERATED ITERATIVE REFINEMENT SOLVERS



Productization Plans

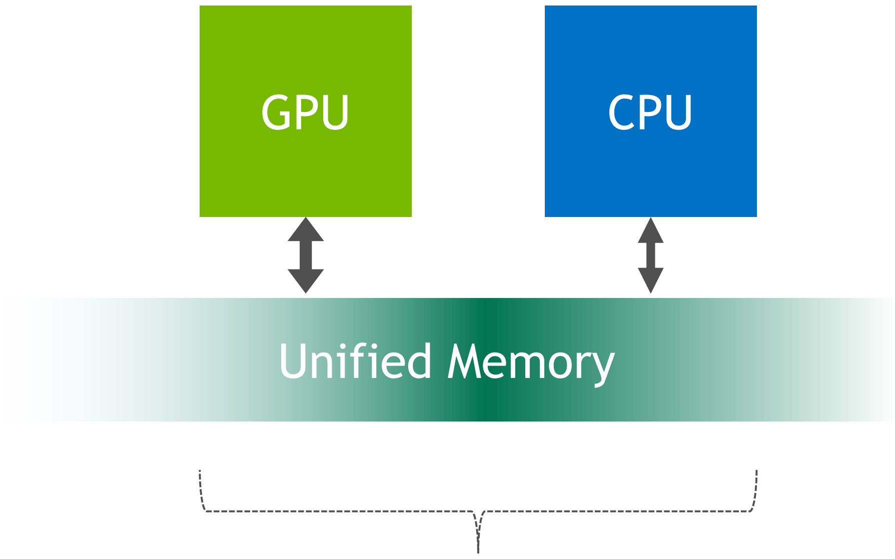
LU Solver	• ~August 2019 • Real & Complex FP32 & FP64
Cholesky Solver	• ~October-November 2019 • Real & Complex FP32 & FP64
QR Solver	• ~October-November 2019 • Real & Complex FP 32 & FP64

UNIFIED VIRTUAL MEMORY

UNIFIED MEMORY

Large datasets, simple programming, High Performance

CUDA 8 and beyond



Allocate Beyond
GPU Memory Size

Enable Large Data Models

Oversubscribe GPU memory

Allocate up to system memory size

Tune Unified Memory Performance

Usage hints via cudaMemAdvise API

Explicit prefetching API

Simpler Data Access

CPU/GPU Data coherence

Unified memory atomic operations

SIMPLIFIED MEMORY MANAGEMENT CODE

CPU Code

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

CUDA 6 Code with Unified Memory

```
void sortfile(FILE *fp, int N) {
  char *data;
  cudaMallocManaged(&data, N);

  fread(data, 1, N, fp);

  qsort<<<...>>>(data,N,1,compare);
  cudaDeviceSynchronize();

  use_data(data);

  cudaFree(data);
}
```

UNIFIED MEMORY EXAMPLE

On-Demand Paging

```
__global__
void setValue(int *ptr, int index, int val)
{
  ptr[index] = val;
}


void foo(int size) {
  char *data;
  cudaMallocManaged(&data, size);

  memset(data, 0, size);

  setValue<<<...>>>(data, size/2, 5);
  cudaDeviceSynchronize();

  useData(data);

  cudaFree(data);
}
```

Unified Memory allocation

Access all values on CPU

Access one value on GPU

HOW UNIFIED MEMORY WORKS IN CUDA 6

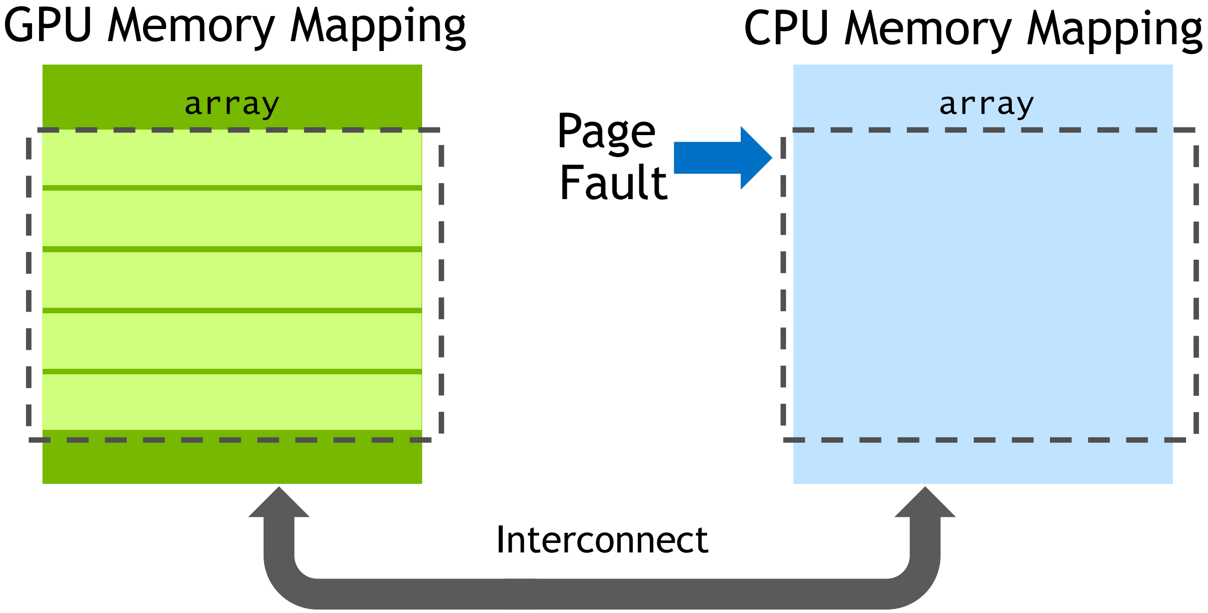
Servicing CPU page faults

GPU Code

```
__global__
void setValue(char *ptr, int index, char val)
{
  ptr[index] = val;
}
```

CPU Code

```
cudaMallocManaged(&array, size);

memset(array, size);

setValue<<<...>>>(array, size/2, 5);
```



HOW UNIFIED MEMORY WORKS ON PASCAL

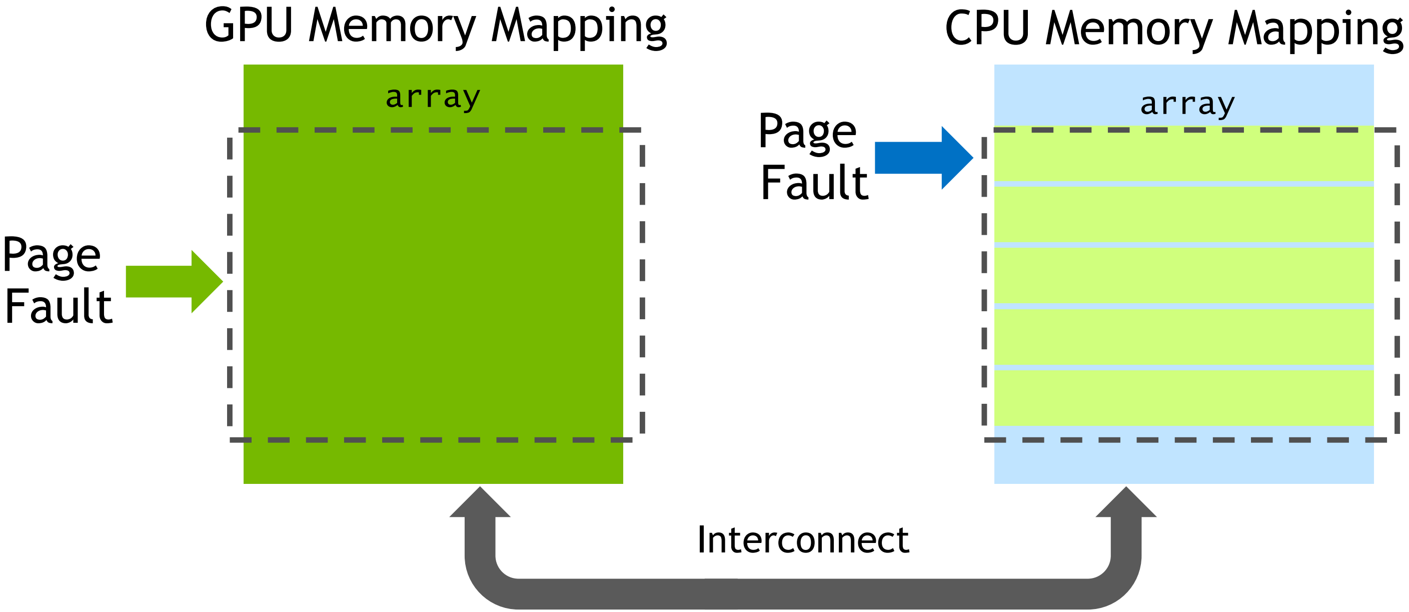
Servicing CPU *and* GPU Page Faults

GPU Code

```
__global__
Void setValue(char *ptr, int index, char val)
{
  ptr[index] = val;
}
```

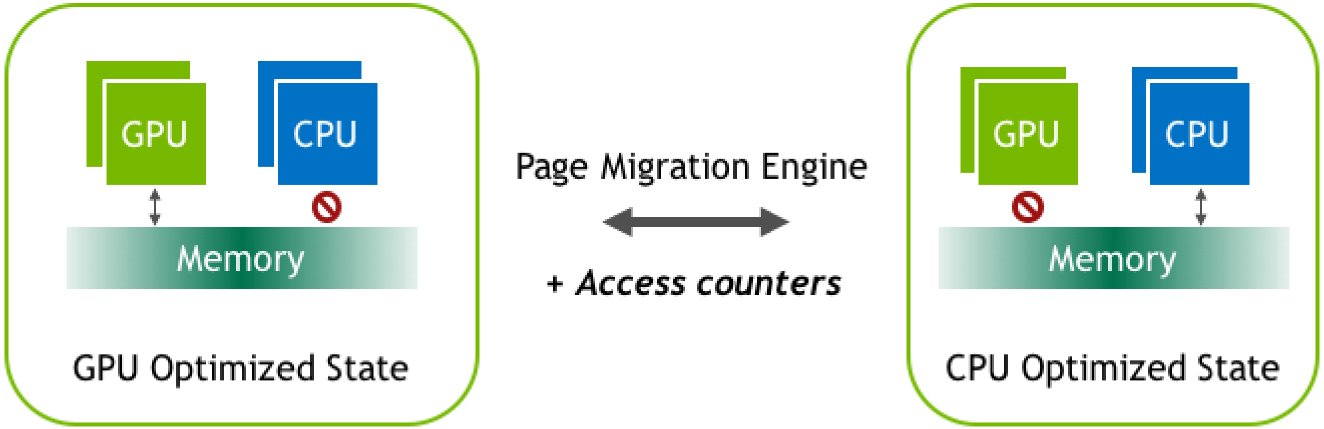
CPU Code

```
cudaMallocManaged(&array, size);

memset(array, size);

setValue<<<...>>>(array, size/2, 5);
```



VOLTA + UNIFIED MEMORY

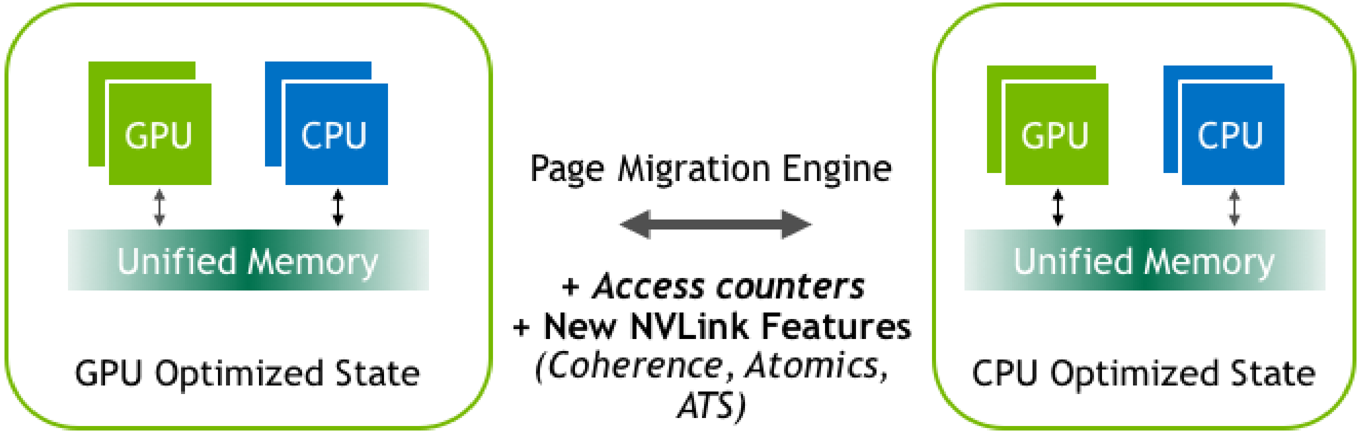
VOLTA + PCIE CPU



Page Migration Engine

+ *Access counters*

VOLTA + NVLINK CPU



Page Migration Engine

+ *Access counters*
+ **New NVLink Features** *(Coherence, Atomics, ATS)*

NGC

NGC: GPU-OPTIMIZED SOFTWARE HUB

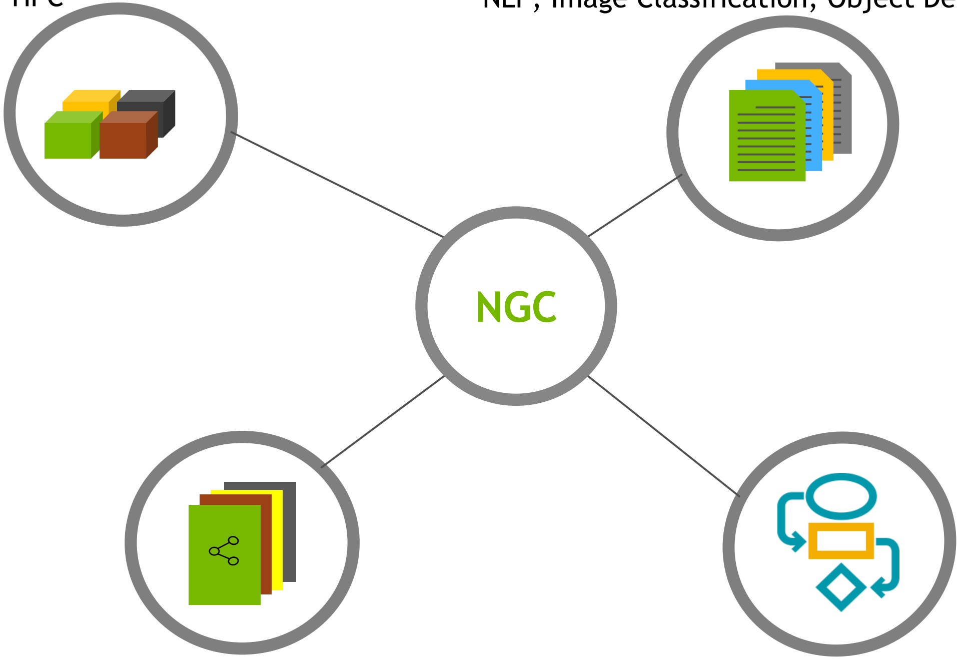
Simplifying DL, ML and HPC Workflows

50+ Containers

DL, ML, HPC

15+ Model Training Scripts

NLP, Image Classification, Object Detection & more



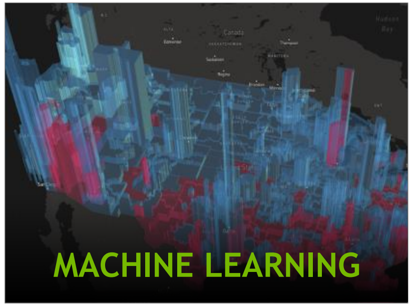
60 Pre-trained Models

NLP, Image Classification, Object Detection & more

Industry Workflows

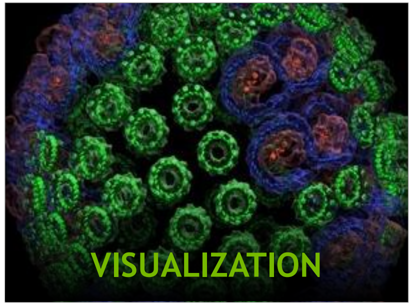
Medical Imaging, Intelligent Video Analytics

TensorFlow | PyTorch | more



RAPIDS | H2O | more

NAMD | GROMACS | more



ParaView | IndeX | more

NVIDIA GPU CLOUD REGISTRY

Common Software stack across NVIDIA GPUs

Deep Learning

All major frameworks with multi-GPU optimizations Uses NCCL for NVLINK data exchange Multi-threaded I/O to feed the GPUs

Caffe, Caffe2,CNTK, mxnet, PyTorch, Tensorflow, Theano, Torch

HPC

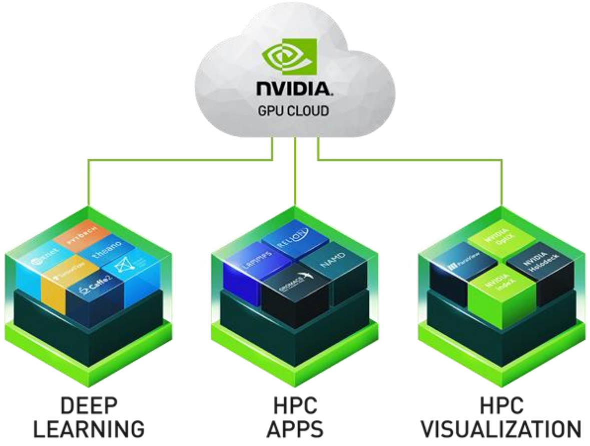
NAMD, Gromacs, LAMMPS, GAMESS, Relion, Chroma, MILC

HPC Visualization

Paraview with Optix, Index and Holodeck with OpenGL visualization base on NVIDIA Docker 2.0, IndeX, VMD

Single NGC Account

For use on GPUs everywhere - https://ngc.nvidia.com



NVIDIA GPU Cloud containerizes GPU-optimized frameworks, applications, runtimes, libraries, and operating system, available at no charge

Gunter Roeth (gunterr@nvidia.com)