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

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 mm²

80 SM
5120 CUDA Cores
640 Tensor Cores

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

*full GV100 chip contains 84 SMs*
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

\[ D = AB + C \]

\[
D = \begin{pmatrix}
A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\
A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\
A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\
A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3}
\end{pmatrix}
\begin{pmatrix}
B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\
B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\
B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\
B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3}
\end{pmatrix}
+ \begin{pmatrix}
C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\
C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\
C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\
C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3}
\end{pmatrix}
\]
VOLTA TENSOR OPERATION

FP16 storage/input → Full precision product → Sum with FP32 accumulator → Convert to FP32 result

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

Dynamic Range

1.4 x 10^-45 < x < 3.4 x 10^38

5.96 x 10^-8 < x < 65504
EFFICIENT LINEAR ALGEBRA COMPUTATIONS ON GPUS
GENERAL MATRIX PRODUCT

Basic definition

General matrix product

\[ C = \alpha \text{op}(A) \ast \text{op}(B) + \beta \ C \]

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

Compute independent dot products

```cpp
// 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

\[ C = \alpha \, \text{op}(A) \times \text{op}(B) + \beta \, C \]

\( C \) 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)
            \[ C[i][j] += A[i][k] \times B[k][j]; \]

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)
            \[ C[i][j] += A[i][k] \times B[k][j]; \]

Load elements of \( A \) and \( B \) exactly once
GENERAL MATRIX PRODUCT

Computing matrix product one block at a time

Partition the loop nest into \textit{blocks} along each dimension

\begin{itemize}
  \item Partition into \textit{Mtile-by-\textit{Ntile}} independent matrix products
  \item Compute each product by accumulating \textit{Mtile-by-\textit{Ntile}-by-\textit{Ktile}} matrix products
\end{itemize}

\begin{verbatim}
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];
              }
      }
\end{verbatim}
**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 $M_{tile}$-by-$N_{tile}$-by-$K$ matrix product in parallel

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

```c
for (int mb = 0; mb < M; mb += M_{tile})
    for (int nb = 0; nb < N; nb += N_{tile})
        for (int kb = 0; kb < K; kb += K_{tile})
            { .. compute $M_{tile}$ by $N_{tile}$ by $K_{tile}$ GEMM }
```

by each 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

```c
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)
            .. 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

```c
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 \(\text{thread}_m\) by \(\text{thread}_n\) by \(\text{thread}_k\) 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:

- $O(M \times N)$ computations on $O(M+N)$ elements

```c
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

NVIDIA cuDNN, cuBLAS, TensorRT
## CUBLAS TENSOR CORE HOW-TO

<table>
<thead>
<tr>
<th></th>
<th>mathMode = CUBLAS_DEFAULT_MATH</th>
<th>mathMode = CUBLAS_TENSOR_OP_MATH</th>
</tr>
</thead>
<tbody>
<tr>
<td>cublasHgemm, cublasSgemm, cublasGemmEx(algo=DEFAULT)</td>
<td>Disallowed</td>
<td>Allowed</td>
</tr>
<tr>
<td>cublasGemmEx(algo=_TENSOR_OP)</td>
<td>Allowed</td>
<td>Allowed</td>
</tr>
</tbody>
</table>


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:
     \[ m \equiv 0 \mod 4 \]
     \[ k \equiv 0 \mod 8 \]
     \( \text{lda, ldb, ldc, A, B, C are aligned to 16 bytes,} \)
  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.

Plans are subject to change
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
  - cuBLASLt 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()
solving linear system $Ax = b$

LU factorization

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

$A \times x = b$

$L \times U \times x = b$

$L \times y = b$

then

$U \times x = y$
TENSOR CORE ACCELERATED IRS
SOLVING LINEAR SYSTEM $AX = B$

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

1. panel factorize
2. update trailing matrix

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

Panel

TRSM

GEMM

step 1

step 2

step 3

L

step 4

panel

update
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

Haidar et al., SC’18 proceedings
Results obtained using MAGMA 2.5.0 and GV100 GPU
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 $Ax=b$

\[
\begin{align*}
A x &= b \\
LUx &= b \\
y &= b \\
\text{then} \\
Ux &= y
\end{align*}
\]
The 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.

\[
\begin{align*}
L U &= LU(A) \\
x &= U\backslash(L\backslash b) \\
r &= b - Ax
\end{align*}
\]

\[
\begin{array}{ll}
\text{lower precision} & 0(n^3) \\
\text{lower precision} & 0(n^2) \\
\text{FP64 precision} & 0(n^2)
\end{array}
\]

\[
\text{WHILE } || r || \text{ not small enough} \\
1. \text{find a correction “z” to adjust } x \text{ that satisfy } Az=r \\
\quad \text{solve } Az=r \text{ could be done by either:} \\
\quad \quad z = U\backslash(L\backslash r) \\
\quad \quad \text{GMRes with LU preconditioner to solve } Az=r \\
2. \quad x = x + z \\
3. \quad r = b - Ax
\]

\[
\begin{array}{ll}
\text{lower precision} & 0(n^2) \\
\text{lower precision} & 0(n^2) \\
\text{FP64 precision} & 0(n^1) \\
\text{FP64 precision} & 0(n^2)
\end{array}
\]

- 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

- solving \( Ax = b \) using FP64 LU

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

Problem generated with an arithmetic distribution of the singular values \( s_i = 1 - \left( \frac{i-1}{n-1} \right) \left( 1 - \frac{1}{\text{cond}} \right) \) and positive eigenvalues.
Tensor Core Accelerated IRS solving linear system $Ax = b$ Performance Behavior

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

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

Problem generated with an arithmetic distribution of the singular values $s_i = 1 - \left(\frac{i}{n-1}\right)\left(1 - \frac{1}{\text{cond}}\right)$ and positive eigenvalues.
Tensor Core Accelerated IRS solving linear system \( Ax = b \)

Performance Behavior

- 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 \( s_i = 1 - \left( \frac{i-1}{n-1} \right)(1 - \frac{1}{\text{cond}}) \) and positive eigenvalues.
Tensor Core Accelerated IRS solving linear system \( Ax = b \)

Performance Behavior

- 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

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

Problem generated with an arithmetic distribution of the singular values
\[ s_i = 1 - \left( \frac{i-1}{n-1} \right) \left( 1 - \frac{1}{\text{cond}} \right) \]
and positive eigenvalues.
Tensor Core Accelerated IRS
solving linear system $Ax = b$  

Performance Behavior

- 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 $s_i = 1 - \left(\frac{i-1}{n-1}\right)\left(1 - \frac{1}{\text{cond}}\right)$ and positive eigenvalues.
Tensor Core Accelerated IRS solving linear system $Ax = b$ Performance Behavior

- 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

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

Problem generated with an clustered distribution of the singular values $\sigma = [1, \cdots, 1, \frac{1}{\text{cond}}]$;
Tensor Core Accelerated IRS solving linear system $Ax = b$ Performance Behavior

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

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?**

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Full FP64</th>
<th>FP32-&gt;FP64</th>
<th>FP16(TC)-FP64</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ( \frac{</td>
<td></td>
<td>b-Ax</td>
<td></td>
</tr>
<tr>
<td>2 ( \frac{</td>
<td></td>
<td>b-Ax</td>
<td></td>
</tr>
<tr>
<td>3 ( \frac{</td>
<td></td>
<td>b-Ax</td>
<td></td>
</tr>
<tr>
<td>4 ( \frac{</td>
<td></td>
<td>b-Ax</td>
<td></td>
</tr>
<tr>
<td>5 ( \frac{</td>
<td></td>
<td>b-Ax</td>
<td></td>
</tr>
<tr>
<td>6 ( \frac{</td>
<td></td>
<td>b-Ax</td>
<td></td>
</tr>
<tr>
<td>7 ( \frac{</td>
<td></td>
<td>b-Ax</td>
<td></td>
</tr>
<tr>
<td>8 ( \frac{</td>
<td></td>
<td>b-Ax</td>
<td></td>
</tr>
<tr>
<td>9 ( \frac{</td>
<td></td>
<td>b-Ax</td>
<td></td>
</tr>
<tr>
<td>10 ( \frac{</td>
<td></td>
<td>b-Ax</td>
<td></td>
</tr>
<tr>
<td>11 ( \frac{</td>
<td></td>
<td>b-Ax</td>
<td></td>
</tr>
<tr>
<td>12 ( \frac{</td>
<td></td>
<td>b-Ax</td>
<td></td>
</tr>
</tbody>
</table>

Total iterations: 4 \( \text{Full FP64} \), 18 \( \text{FP32->FP64, FP16(TC)-FP64} \)

Final solution:
- \( \frac{||b-Ax||}{||A||} \) \( n \): 1.50e-16 \( \text{Full FP64} \), 1.20e-16 \( \text{FP32->FP64} \), 1.19e-16 \( \text{FP16(TC)-FP64} \)
- \( \frac{||b-Ax||}{||A|| \cdot ||x||} \) \( n \): 1.49e-21 \( \text{Full FP64} \), 1.19e-21 \( \text{FP32->FP64} \), 1.18e-21 \( \text{FP16(TC)-FP64} \)
- \( \frac{||b-Ax||}{\left( ||A|| \cdot ||x|| + ||b|| \right) n} \): 1.49e-21 \( \text{Full FP64} \), 1.19e-21 \( \text{FP32->FP64} \), 1.18e-21 \( \text{FP16(TC)-FP64} \)
Leveraging Tensor Cores
Iterative Refinement Solver

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

\[
time \text{ for MP} = \frac{2n^3}{3P_{X_{getrf}}} + k \left( \frac{2n^2}{P_{dgemv}} + \frac{2n^2}{P_{X_{trsv}}} + \xi \right)
\]

Haidar et al., SC’18 proceedings
Results obtained using MAGMA 2.5.0 and GV100 GPU
Mixed-precision iterative refinement solver
GV100 vs TU102

Results obtained using MAGMA 2.5.0

\[
\text{time for FP64} = \frac{2n^3}{3P_{dgetrf}} + \frac{2n^2}{P_{dtrsv}} \\
\text{time for MP} = \frac{2n^3}{3P_{X_{getrf}}} + k \left( \frac{2n^2}{P_{dgemv}} + \frac{2n^2}{P_{X_{trsv}}} + \xi \right)
\]
Mixed-precision iterative refinement solver

Energy Efficiency

Solving $Ax=b$ on Nvidia V100

CPU: 10 cores E5-2650 v3
GPU: Nvidia V100

Time (sec)
0 1 2 3 4 5 6 7
Average power CPU+GPU (Watts)
0 20 40 60 80 ...

FP64 solver dgesv
FP32 $\rightarrow$ 64 solver dsgesv
FP16 $\rightarrow$ 64 solver dhgesv
FP16 $\rightarrow$ 64 solver dhgesv (TC)

Haidar et al., SC’18 proceedings
Results obtained using MAGMA 2.5.0 and GV100 GPU
**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

*Plans subject to change*
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 Upto 3X Speedup

<table>
<thead>
<tr>
<th>TensorFlow</th>
<th>PyTorch</th>
<th>MXNet</th>
</tr>
</thead>
</table>
| 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:

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

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

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

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:
      ```python
      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

```python
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:

```python
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


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

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


- 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

(*) based on ResNet50 v1.5

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

```
wmma::load_matrix_sync(Amat, a, stride);
```
CUDA TENSOR CORE PROGRAMMING

WMMA Matrix Multiply and Accumulate Operation

Warp-level operation to perform matrix multiply and accumulate

\[
\text{wmma::mma\_sync(Dmat, Amat, Bmat, Cmat)};
\]

\[
D = \begin{pmatrix}
\end{pmatrix} + \begin{pmatrix}
\end{pmatrix}
\]
CUDA TENSOR CORE PROGRAMMING

WMMA load and store operations

Warp-level operation to fetch components of matrices into fragments

```cpp
wmma::store_matrix_sync(d, Dmat, stride);
```
**TENSOR CORE EXAMPLE**

__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
```
CUDA FORTRAN TENSOR CORE EXAMPLE

Host Code

```fortran
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

Launch with a single warp of threads
VOLTA MMA.SYNC

Warp-scoped matrix multiply instruction

mma.sync: new instruction in CUDA 10.1

• Directly targets Volta Tensor Cores

Matrix multiply-accumulate

\[ D = A \times B + C \]

• A, B: half
• C, D: float or half

Warp-synchronous:

• Four independent 8-by-8-by-4 matrix multiply-accumulate operations
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  \[ D = A \times B + C \]

**PTX Syntax**

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

```plaintext
.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)

```c
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.
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

```cpp
/// 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
LOADING FROM SHARED MEMORY

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 B multiplicand
  typename ScalarB,
  /// Data type of accumulators
  typename ScalarC,
  /// Whether infinite results are saturated to +\text{-}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
```

<table>
<thead>
<tr>
<th>Invocations</th>
<th>Metric Name</th>
<th>Metric Description</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device &quot;Quadro GV100 (0)&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel: compute_gemm(__half const *, __half const *, float const <em>, float</em>, float, float)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>tensor_precision_fu_utilization</td>
<td>Tensor-Precision Function Unit Utilization</td>
<td>Mid (5)</td>
<td>Mid (5)</td>
<td>Mid (5)</td>
</tr>
</tbody>
</table>
TENSOR CORES WITH NSIGHT COMPUTE

- The Nsight Compute CLI allows collecting several metrics related to tensor core usage
- This data can be viewed 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

<table>
<thead>
<tr>
<th>sm__pipe_tensor_cycles_active.avg.pct_of_peak_sustained_active</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>43.44</td>
</tr>
</tbody>
</table>

NVIDIA Nsight Compute

API Stream

Next Trigger: Enter filter

ID | API Name | Details
---|---------|--------

Command line profiler metrics

<table>
<thead>
<tr>
<th>sm__pipe_tensor_cycles_active.avg.pct_of_peak_sustained_active</th>
<th>43.70</th>
</tr>
</thead>
</table>
CASE STUDIES
ITERATIVE REFINEMENT (ASGARD & HPL)
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.
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_iterf_gpu( N, nrhs,
    d_A, ldda, h_ipiv, d_ipiv,
    d_B, lddb, d_X, lddx,
    d_workspace, &gesv_iter, &info );
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

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

Performance on a wider range of problems investigating the energy efficiency of the approach compared to working precision implementations and other...
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:

\[ \frac{dv}{dt} = \frac{q}{m(v \times B)}, \quad v = \text{velocity}, \quad q = \text{charge}, \quad m = \text{mass}, \quad B = \text{magnetic field} \]

- *Discretizing the above equation in 2 dimension can lead to:

```c
/*grab magnetic field at current position*/
B = EvalB(x);

/*get new velocity at n+1*/
v[0] = v[0] + q/m*B*v[1]*dt;  
v[1] = v[1] - q/m*B*v[0]*dt;

/*update position*/
x[0] = x[0] + v[0]*dt;  

/*push down*/
v[0]=v[2][0];  
v[1]=v[2][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, $v^+$ acts as velocity update. Electric field can be easily added.

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

\[
v' = v^- + v^- \times t \quad t = (qB/m)\Delta t/2
\]

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

*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
DGX Mixed-Precision Led MLPerf
World’s Fastest Industry-Wide AI Benchmark Achieved on NVIDIA GPUs

<table>
<thead>
<tr>
<th>Task</th>
<th>Platform Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image Classification</td>
<td>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</td>
</tr>
<tr>
<td>Object Detection</td>
<td>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</td>
</tr>
<tr>
<td>Object Detection</td>
<td>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</td>
</tr>
<tr>
<td>Translation (recurrent)</td>
<td>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</td>
</tr>
<tr>
<td>Translation (non-recurrent)</td>
<td>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</td>
</tr>
<tr>
<td>Recommendation</td>
<td>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</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task</th>
<th>Time to Accuracy on Single Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image Classification RN50 v.1.5 MXNet</td>
<td>70 minutes</td>
</tr>
<tr>
<td>Object Detection Mask R-CNN PyTorch</td>
<td>167 minutes</td>
</tr>
<tr>
<td>Object Detection SSD PyTorch</td>
<td>14 minutes</td>
</tr>
<tr>
<td>Translation (recurrent) GNMT PyTorch</td>
<td>10 minutes</td>
</tr>
<tr>
<td>Translation (non-recurrent) Transformer</td>
<td>19 minutes</td>
</tr>
<tr>
<td>Recommendation NCF PyTorch</td>
<td>0.4 minutes</td>
</tr>
</tbody>
</table>
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

## Model Scripts for Various Applications


<table>
<thead>
<tr>
<th>Computer Vision</th>
<th>Speech &amp; NLP</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSD PyTorch</td>
<td>GNMT v2 TensorFlow</td>
</tr>
<tr>
<td>SSD TensorFlow</td>
<td>GNMT v2 PyTorch</td>
</tr>
<tr>
<td>UNET-Industrial TensorFlow</td>
<td>Transformer PyTorch</td>
</tr>
<tr>
<td>UNET-Medical TensorFlow</td>
<td>BERT (Pre-training and Q&amp;A)</td>
</tr>
<tr>
<td>ResNet-50 v1.5 MXNet</td>
<td>TensorFlow</td>
</tr>
<tr>
<td>ResNet-50 PyTorch</td>
<td></td>
</tr>
<tr>
<td>ResNet-50 TensorFlow</td>
<td></td>
</tr>
<tr>
<td>Mask R-CNN PyTorch</td>
<td></td>
</tr>
<tr>
<td>recommender systems</td>
<td></td>
</tr>
<tr>
<td>NCF PyTorch</td>
<td>Tacotron2 and WaveGlow PyTorch</td>
</tr>
<tr>
<td>NCF TensorFlow</td>
<td></td>
</tr>
</tbody>
</table>

**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

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

**Source:** [https://github.com/NVIDIA/DeepLearningExamples/tree/master/MxNet/Classification/RN50v1.5](https://github.com/NVIDIA/DeepLearningExamples/tree/master/MxNet/Classification/RN50v1.5)

**GPU:** 1xV100-16GB | DGX-1V | Batch Size: 208 (FP16), 96 (FP16)
SPEECH SYNTHESIS: Tacotron 2 And WaveGlow v1.0

https://ngc.nvidia.com/catalog/model-scripts/nvidia:totacotron_2_and_waveglow_for_pytorch

<table>
<thead>
<tr>
<th>DGX-1V 16G</th>
<th>Tacotron 2 FP32</th>
<th>Tacotron 2 Mixed Precision</th>
<th>WaveGlow FP32</th>
<th>WaveGlow Mixed Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time to Train (Hours)</strong></td>
<td><strong>44</strong> @ 1500 epochs</td>
<td><strong>33.14</strong> @ 1500 epochs</td>
<td><strong>109.96</strong> @ 1000 epochs</td>
<td><strong>54.83</strong> @ 1000 epochs</td>
</tr>
<tr>
<td><strong>Train Accuracy Loss (@1000 Epochs)</strong></td>
<td><strong>0.3629</strong></td>
<td><strong>0.3645</strong></td>
<td><strong>-6.1087</strong></td>
<td><strong>-6.0258</strong></td>
</tr>
<tr>
<td><strong>Perf.</strong></td>
<td><strong>10,843 tokens/sec</strong></td>
<td><strong>12,742 tokens/sec</strong></td>
<td><strong>257,687(*) samples/sec</strong></td>
<td><strong>500,375(*) samples/sec</strong></td>
</tr>
</tbody>
</table>

Data set | LJ Speech Dataset

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


GPU: 1xV100-16GB | DGX-1V | Batch Size: 208 (FP16), 96 (FP16)
**LANGUAGE MODELING: BERT for TensorFlow**

https://ngc.nvidia.com/catalog/model-scripts/nvidia:bert_for_tensorflow

<table>
<thead>
<tr>
<th>DGX-1V 8GPU 32G</th>
<th>TF BERT FP32</th>
<th>TF BERT Mixed Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to Train [Hours]</td>
<td><strong>0.77</strong> (BSxGPU = 4)</td>
<td><strong>0.51</strong> (BSxGPU = 4)</td>
</tr>
<tr>
<td>Train F1 (mean)</td>
<td><strong>90.83</strong></td>
<td><strong>90.99</strong></td>
</tr>
<tr>
<td>Perf (BSxGPU = 4)</td>
<td><strong>66.65</strong> sentences/sec</td>
<td><strong>129.16</strong> sentences/sec</td>
</tr>
</tbody>
</table>

**NGC 19.03 TensorFlow container**

OBJECT DETECTION: TensorFlow SSD


<table>
<thead>
<tr>
<th>GPU: 8xV100-16GB</th>
<th>DGX-1V</th>
<th>Batch Size: 32 (FP32, Mixed)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th></th>
<th>TF SSD FP32</th>
<th>TF SSD Mixed Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time to Train</strong></td>
<td>1h 37min</td>
<td>1h 19min</td>
</tr>
<tr>
<td><strong>Accuracy (map)</strong></td>
<td>0.268</td>
<td>0.269</td>
</tr>
<tr>
<td><strong>Perf. (BSxGPU = 32)</strong></td>
<td>569 Img/sec</td>
<td>752 Img/sec</td>
</tr>
<tr>
<td><strong>Data set</strong></td>
<td>COCO 2017</td>
<td></td>
</tr>
</tbody>
</table>

**NGC 19.03 TensorFlow container**

## TRANSLATION: PyTorch GNMT


<table>
<thead>
<tr>
<th>DGX-2V 16GPU 32G</th>
<th>PyTorch GNMT FP32</th>
<th>PyTorch GNMT Mixed Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to Train [min]</td>
<td>58.6</td>
<td>26.3</td>
</tr>
<tr>
<td>Train Accuracy BLEU score</td>
<td>24.16</td>
<td>24.22</td>
</tr>
<tr>
<td>Perf.</td>
<td>314,831 tokens/sec</td>
<td>738,521 tokens/sec</td>
</tr>
<tr>
<td>Data set</td>
<td>WMT16 English to German</td>
<td></td>
</tr>
</tbody>
</table>

NGC 19.01 PyTorch container

## RECOMMENDER: PyTorch Neural Collaborative Filter


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

### NGC 18.12 PyTorch container


GPU: 8xV100-16GB | DGX-1 | Batch size: 1,048,576
INDUSTRIAL DEFECT DETECTION: TensorFlow U-Net


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

NGC 19.03 TensorFlow container

Source: https://github.com/NVIDIA/DeepLearningExamples/tree/master/TensorFlow/Segmentation/UNet_Industrial

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

**UNET-Industrial for TensorFlow**

<table>
<thead>
<tr>
<th>Publisher</th>
<th>Application</th>
<th>Version</th>
<th>Last Modified</th>
<th>Training Frameworks</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA</td>
<td>Segmentation</td>
<td>1</td>
<td>April 22, 2019</td>
<td>TensorFlow</td>
</tr>
</tbody>
</table>

**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 trained to avoid overfitting.

**Labels**

DEEP LEARNING TRAINING

**Overview**

This U-Net model is adopted from the original version of the [U-Net model](https://arxiv.org/abs/1505.04597) 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](https://arxiv.org/abs/1505.04597).

Convolutional networks for biomedical image segmentation.

This work proposes a modified version of U-Net, called 2D-CNet, which performs efficiently and with very high accuracy on the industrial anomaly dataset [DAGM2007](https://www.cs.tut.fi/~clv/dagm2007/).

This model is composed of two parts:

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

It repeatedly applies 2 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 2x upsampleD2 layer followed by a 2D convolution, a concatenation operation with the residual connection and two 2D convolutions.

2D-CNet 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.
## Matching Accuracy for FP32 and Mixed Precision

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

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]

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

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

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);

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 FP16
exact results only up to N < 2048

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!
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


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