NVIDIA A100 GPU:
PERFORMANCE & INNOVATION
FOR GPU COMPUTING

Jack Choquette & Wish Gandhi
A100: UNPRECEDENTED ACCELERATION AT EVERY SCALE

56 BILLION XTORS

3rd GEN TENSOR CORES

SPARSITY ACCELERATION

MIG

3rd GEN NVLINK & NVSWITCH
A100 GPU

54 Billion Transistors in 7nm

108 SMs
6912 CUDA Cores

40MB L2
6.7x capacity

1.56 TB/s HBM2
1.7x bandwidth

SCALe OUT
Multi-Instance GPU
Elastic GPU

SCALe UP
3rd gen. NVLINK
2x BW

56 BILLION XTORS

1.56 TB/s HBM2

GigaThread Engine with MIG Control

3rd gen. NVLINK
A100 SM

- Third-generation Tensor Core
  - Faster and more efficient
  - Comprehensive data types
  - Sparsity acceleration
- Asynchronous data movement and synchronization
- Increased L1/SMEM capacity
**ACCELERATING HPC**

All results are measured. BERT Large Training (FP32 & FP16) measures Pre-Training phase, uses PyTorch including (2/3) Phase 1 with Seq Len 128 and (1/3) Phase 2 with Seq Len 512, V100 is DGX1 Server with 8xV100, A100 is DGX A100 Server with 8xA100, A100 uses TF32 Tensor Core for FP32 training.

BERT Large Inference uses TRT 7.1 for T4/V100, with INT8/FP16 at batch size 256. Pre-production TRT for A100, uses batch size 94 and INT8 with sparsity.
NVIDIA DGX SUPERPOD SETS ALL 8 AT-SCALE AI RECORDS
Uniquely Able to Run Full Breadth of Networks

**Commercially Available Solutions**

- **Translation (Non-recurrent) Transformer**
  - NVIDIA A100: 0.6 (480 A100)
  - NVIDIA V100: 0.7 (1024 A100)

- **Translation (Recurrent) GNMT**
  - NVIDIA A100: 0.8 (1840 A100)
  - NVIDIA V100: 0.8 (1024 A100)

- **Image Classification ResNet-50 v.1.5**
  - NVIDIA A100: 0.8 (1840 A100)
  - NVIDIA V100: 0.8 (1024 A100)

- **Object Detection (Light Weight) SSD**
  - NVIDIA A100: 0.8 (2048 A100)

- **NLP BERT**
  - NVIDIA A100: 0.8 (2048 A100)

- **Recommendation DLRM**
  - NVIDIA A100: 3.3 (8 A100)

- **Object Detection (Heavy Weight) Mask R-CNN**
  - NVIDIA A100: 10.5 (256 A100)

- **Reinforcement Learning MiniGo**
  - NVIDIA A100: 17.1 (1792 A100)

- **Time to Train (Minutes)**
  - NVIDIA A100:
    - Reinforcement Learning MiniGo: 17.1 (1792 A100)
  - NVIDIA V100:
    - Translation (Non-recurrent) Transformer: 0.6 (480 A100)
  - Google TPUv3:
    - Translation (Non-recurrent) Transformer: 0.7 (1024 A100)
  - Huawei Ascend:
    - Translation (Non-recurrent) Transformer: 28.7 (16 TPUv3)

MLPerf 0.7 Performance comparison at Max Scale. Max scale used for NVIDIA A100, NVIDIA V100, TPUv3 and Huawei Ascend for all applicable benchmarks. | MLPerf ID at Scale: :Transformer: 0.7-30, 0.7-52, GNMT: 0.7-34, 0.7-54, ResNet-50 v1.5: 0.7-37, 0.7-55, 0.7-71, 0.7-73, SSD: 0.7-33, 0.7-53, BERT: 0.7-38, 0.7-56, 0.7-71, DLRM: 0.7-17, 0.7-43, Mask R-CNN: 0.7-38, 0.7-48, MiniGo: 0.7-36, 0.7-51 | MLPerf name and logo are trademarks. See [www.mlperf.org](http://www.mlperf.org) for more information.
NVIDIA A100 SETS ALL 8 PER-CHIP AI PERFORMANCE RECORDS

Relative Speedup
Commercially Available Solutions

Speedup Over V100

0x 1x 2x 3x

Image Classification ResNet-50 v.1.5
NLP BERT
Object Detection (Heavy Weight) Mask R-CNN
Reinforcement Learning MiniGo
Object Detection (Light Weight) SSD
Translation (Recurrent) GNMT
Translation (Non-recurrent) Transformer
Recommendation DLRM

Per Chip Performance arrived at by comparing performance at same scale when possible and normalizing it to a single chip. 8 chip scale: V100, A100 Mask R-CNN, MiniGo, SSD, GNMT, Transformer. 16 chip scale: V100, A100, TPUv3 for ResNet-50 v1.5 and BERT. 512 chip scale: Huawei Ascend 910 for ResNet-50. DLRM compared 8 A100 and 16 V100. Submission IDs: ResNet-50 v1.5: 0.7-3, 0.7-1, 0.7-44, 0.7-18, 0.7-21, 0.7-15 BERT: 0.7-1, 0.7-45, 0.7-22 Mask R-CNN: 0.7-40, 0.7-19, MiniGo: 0.7-41, SSD: 0.7-40, 0.7-19, GNMT: 0.7-40, 0.7-19, Transformer: 0.7-40, 0.7-19, DLRM: 0.7-43, 0.7-17 MLPerf name and logo are trademarks. See www.mlperf.org for more information.
NVIDIA A100 DELIVERS FASTEST PERFORMANCE AVAILABLE

Relative Per Chip Speedup
NVIDIA commercially available submission compared to non-commercial submissions

Per Chip Performance arrived at by comparing performance at same scale when possible and normalizing it to a single chip. 8 chip scale: V100, A100, TPUv4 and Intel. ResNet-50 v1.5 (A100, TPUv4, Intel). Mask R-CNN, SSD, GNMT, Transformer, BERT (A100, TPUv4) and DLRM (A100). 16 chip scale V100, TPUv3: DLRM (V100), ResNet-50 V1.5, BERT, MiniGo (V100, A100). 32 chip scale for Intel for MiniGo. 512 chip scale: Huawei Ascend 910 for ResNet-50. Submission IDs: ResNet-50 v1.5: 0.7-61, 0.7-3, 0.7-44, 0.7-18; 0.7-15 BERT: 0.7-1, 0.7-45, 0.7-68, 0.7-19; Mask R-CNN: 0.7-40, 0.7-68, 0.7-19; MiniGo: 0.7-23, SSD: 0.7-4, 0.7-23; GNMT: 0.7-45, 0.7-68, 0.7-19; Transformer: 0.7-45, 0.7-68, 0.7-19 DLRM: 0.7-59, 0.7-43, 0.7-68, 0.7-17. MLPerf name and logo are trademarks. See www.mlperf.org for more information.
LET’S GO DEEPER

Strong Scaling: Top to Bottom

Scale Up and Scale Out: Elastic GPU

Productivity: Asynchronous Programming
STRONG SCALING
DL STRONG SCALING

DL networks:
Long chains of sequentially-dependent compute-intensive layers
Each layer is parallelized across GPU

Weak scaling

Strong scaling

Fixed network runs ~2.5x faster

~2.5x larger network runs in same time

Tile: work for 1 SM

Input Activations

Output Activations

Weights

1 layer

~2.5x faster network runs in same time
# A100 TENSOR CORE

<table>
<thead>
<tr>
<th>Input Operands</th>
<th>Accumulator</th>
<th>TOPS</th>
<th>SPARSE TOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP32</td>
<td>FP32</td>
<td>15.7</td>
<td>-</td>
</tr>
<tr>
<td>FP16</td>
<td>FP32</td>
<td>125</td>
<td>-</td>
</tr>
<tr>
<td>FP32</td>
<td>FP32</td>
<td>19.5</td>
<td>-</td>
</tr>
<tr>
<td>TF32</td>
<td>FP32</td>
<td>156</td>
<td>312</td>
</tr>
<tr>
<td>FP16</td>
<td>FP32</td>
<td>312</td>
<td>624</td>
</tr>
<tr>
<td>BF16</td>
<td>FP32</td>
<td>312</td>
<td>624</td>
</tr>
<tr>
<td>INT8</td>
<td>INT32</td>
<td>624</td>
<td>1248</td>
</tr>
<tr>
<td>INT4</td>
<td>INT32</td>
<td>1248</td>
<td>2496</td>
</tr>
<tr>
<td>BINARY</td>
<td>INT32</td>
<td>4992</td>
<td>-</td>
</tr>
</tbody>
</table>

**FP16 in/out Data**
- **Dense**: 2.5x (2x per SM)
- **Sparse**: 5x (4x per SM)
## A100 TENSOR CORE

<table>
<thead>
<tr>
<th>Input Operands</th>
<th>Accumulator</th>
<th>TOPS</th>
<th>SPARSE TOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP32</td>
<td>FP32</td>
<td>15.7</td>
<td>-</td>
</tr>
<tr>
<td>FP32</td>
<td>FP32</td>
<td>125</td>
<td>-</td>
</tr>
<tr>
<td>FP32</td>
<td>FP32</td>
<td>19.5</td>
<td>-</td>
</tr>
<tr>
<td>TF32</td>
<td>FP32</td>
<td>156</td>
<td>312</td>
</tr>
<tr>
<td>FP16</td>
<td>FP32</td>
<td>312</td>
<td>624</td>
</tr>
<tr>
<td>BF16</td>
<td>FP32</td>
<td>312</td>
<td>624</td>
</tr>
<tr>
<td>INT8</td>
<td>INT32</td>
<td>624</td>
<td>1248</td>
</tr>
<tr>
<td>INT4</td>
<td>INT32</td>
<td>1248</td>
<td>2496</td>
</tr>
<tr>
<td>BINARY</td>
<td>INT32</td>
<td>4992</td>
<td>-</td>
</tr>
</tbody>
</table>

FP32 in/out Data
- **Dense:** 10x (8x per SM)
- **Sparse:** 20x (16x per SM)
HOW TO KEEP TENSOR CORES FED?

Math Bandwidth
(MACs/clock/SM)

Required Data Bandwidth
(A+B operands, B/clock/SM)

TF32 | FP16 | BF16 | INT8 | INT4 | BIN
A100 dense | A100 sparse

V100

3x vs.
V100

2x vs.
V100

NVIDIA
A100 STRONG SCALING INNOVATIONS

Improve speeds & feeds and efficiency across all levels of compute and memory hierarchy
STRONG SCALING: SM CORE
A100 SM DATA MOVEMENT EFFICIENCY

3x SMEM/L1 Bandwidth, 2x In-flight Capacity

V100

Tensor Cores
Load-Shared (4x)

4 reads
1 reads
1 write

Reserved for in-flight data

Load-Global

L1

RF

SMEML1

L2

DRAM

A100

Tensor Cores
Load-Shared (2x)

2 reads

Reserved for in-flight data

Load-Global-Store-Shared (Async-Copy)

L2

DRAM
STRONG SCALING: MEMORY SYSTEM
## A100 L2 BANDWIDTH

### Parallelize across GPU

**Output Activations**

### V100
- 80 SMs
- V100 TC
- 64 L2 slices
- 32 B/clk/slice
- **16 B/clk/SM**
- **94%**

### V100++ (hypothetical)
- 108 SMs
- A100 TC
- 64 L2 slices
- 32 B/clk/slice
- **24 B/clk/SM**
- **253%**

### A100
- 108 SMs
- A100 TC
- 80 L2 slices
- 64 B/clk/slice
- **24 B/clk/SM**
- **101%**

### Split L2 with hierarchical crossbar
- 2.3x increase in bandwidth over V100, lower latency
A100 L2 FEATURES

**Faster Atomics**
Global memory atomics performed near memory
→ 11x improvement for FP16 over V100

**Larger and smarter L2**
40MB L2, 6.7x vs. V100
L2-Residency controls

Keep data resident in L2 to reduce DRAM bandwidth
A100 DRAM BANDWIDTH

Capacity
- 40 GB of HBM2 with 5 active stacks
- 1.25x vs. V100

Faster HBM
- 38% faster clocks
- 1.6 TB/s, 1.7x vs. V100

ECC Resiliency
- SECDED to protect data
- Page re-mapper to isolate bad pages
A100 COMPUTE DATA COMPRESSION

Activation sparsity due to ReLU

Up to 4x DRAM+L2 bandwidth and 2x L2 capacity for fine-grained unstructured sparsity
A100 NVLINK BANDWIDTH

Third Generation NVLink
50 Gbit/sec per signal pair
12 links, 25 GB/s in/out, 600 GB/s total
2× vs. V100
A100 STRONG SCALING INNOVATIONS
Delivering Unprecedented Levels of Performance

A100 Improvements Over V100
- 2.5x Tensor Core math BW (FP16)
  - 5x Sparse Tensor Core math BW (FP16)
- 2.9x Effective RF BW with A100 Tensor Core
- 2.8x Effective RF capacity with Async-Copy bypassing RF
- 3.0x Effective SMEM BW with A100 Tensor Core and Async-Copy
- 2.3x SMEM capacity
- 2.3x L2 BW
- 6.7x L2 capacity, +Residency Control
- 1.7x DRAM BW
- 1.3x DRAM capacity
- 2.0x NVLINK BW
SCALE UP AND SCALE OUT: ELASTIC GPU
A100: MULTI-GPU SCALE UP

Meet growing computational demands for complex DNNs and HPC simulations

8 GA100 connected by new NVLINK3 enabled NVSwitch

System synchronization and error attribution at the requester executing context

NVIDIA DGX POD/SuperPOD include multiple DGX A100 systems for strong scaling
DGX SUPERPOD

- Modular scale: From 1 Scalable Unit (20 DGX A100) to Full SuperPOD (140 DGX A100)

- 1K GPU SuperPOD Cluster
  - 140 DGX A100 nodes (1120 GPUs) in a GPU POD
  - 1st tier fast storage
  - Mellanox HDR 200Gb/s InfiniBand - Full Fat-tree
  - Network optimized for AI and HPC

- Scalable Infrastructure
  - Modular InfiniBand Fat-tree
  - Core IB Switches Distributed Between PODs
  - Direct connect POD to POD
  - Separate network for Compute vs Storage
  - Adaptive routing and SharpV2 support for offload
NEW: MULTI-INSTANCE GPU (MIG)
Optimize GPU Utilization, Allocate Users with Guaranteed QoS

Up to 7 GPU Instances in a Single A100: Dedicated SM, memory, L2 cache, bandwidth for hardware quality of service & isolation

Simultaneous Workload Execution with Guaranteed Quality of Service: All MIG instances run in parallel with predictable throughput & latency

Right-Sized GPU Allocation: Different-sized MIG instances based on target workloads

Flexibility to run any type of workload on a MIG instance

Diverse Deployment Environments: Supported with bare metal, Docker, Kubernetes, virtualized env.
ELASTIC GPU COMPUTING

Each A100 is 1 to 7 GPUs
Each DGX A100 is 1 to 56 GPUs
Each GPU can serve a different user, with full memory isolation and QoS
PRODUCTIVITY: ASYNCHRONOUS PROGRAMMING
COMPUTE CAPABILITY
Programming Model Development at NVIDIA

- Concurrent algorithms
- Managed memory
- Bulk parallelism + atomics
PROGRAMMING MODEL WANTED
Software Pipelining to Hide Latency is Hard

__device__ void exhibit_A1()
{
    memcpy(/* ... */); //< blocks here
    /* more work */

    compute(); //< needed here
    /* more work */
}

__device__ void exhibit_B1()
{
    compute_head();
    __syncthreads(); //< blocks here
    /* more work */

    compute_tail(); //< needed here
    /* more work */
}
PROGRAMMING MODEL WANTED
Software Pipelining to Hide Latency is Hard

__device__ void exhibit_A2() {
    memcpy(/* ... */);  // blocks here
    /* memcpy( ... ); */
    compute();          // needed here
    /* compute(); */
}

__device__ void exhibit_B2() {
    compute_head();
    __syncthreads();   // blocks here
    /* compute_head();
     __syncthreads(); */
    compute_tail();    // needed here
    /* compute_tail(); */
}
NEW:

- Asynchronous algorithms
- Concurrent algorithms
- Managed memory
- Bulk parallelism
CO-DESIGNED: A100 & C++20 BARRIER
Key to Asynchronous Programming in Compute 8.0

#include <cuda/barrier> // ISO C++20 conforming extension
using barrier = cuda::barrier<cuda::thread_scope_block>;

class barrier { // synopsis
    //...
    void arrive_and_wait();
    arrival_token arrive(ptrdiff_t = 1);
    void wait(arrival_token &&) const;
    //...
};

Nonblocking
# ASYNCHRONOUS COPY + BARRIER

<table>
<thead>
<tr>
<th>Capability</th>
<th>PTX ISA</th>
<th>CUDA C++ API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asynchronous barrier</td>
<td>mbARRIER.{&lt;basis functions&gt;}</td>
<td>cuda::barrier&lt;...&gt;</td>
</tr>
<tr>
<td>Asynchronous copy</td>
<td>cp.async.ca + cp.async.mbarrier.arrive</td>
<td>cuda::memcpy_async(...).</td>
</tr>
<tr>
<td>+Cache-bypass</td>
<td>cp.async.cg</td>
<td></td>
</tr>
<tr>
<td>+Zero-fill ragged edge</td>
<td>cp.async.* ... wr-size, rd-size;</td>
<td></td>
</tr>
<tr>
<td>+User-level tracking</td>
<td>cp.async.mbarrier.arrive.noinc</td>
<td></td>
</tr>
<tr>
<td>+Single-threaded mode</td>
<td>cp.async.{commit_group, wait_group}</td>
<td></td>
</tr>
</tbody>
</table>

CUDA 11 preview library in experimental:: namespace
ASYNCHRONOUS PROGRAMMING MODEL

#include <cuda/barrier> // ISO C++20 conforming extension
using barrier = cuda::barrier<cuda::thread_scope_block>;

__device__ void exhibit_A3()
{
__shared__ barrier b1, b2;
// ^^initialization omitted
cuda::memcpy_async(/* ... */), b1);
    cuda::memcpy_async(/* ... */), b2);
    b1.arrive_and_wait();
    compute();
    b2.arrive_and_wait();
    compute();
}

__device__ void exhibit_B3()
{
__shared__ barrier b1, b2;
// ^^initialization omitted
compute_head();
auto t1 = b1.arrive();
compute_head();
    auto t2 = b2.arrive();
    b1.wait(t1);
    compute_tail();
    b2.wait(t2);
    compute_tail();
}
#include <cuda/barrier> // ISO C++20 conforming extension
using barrier = cuda::barrier<cuda::thread_scope_block>;

__global__ void exhibit_C(/* ... */) {
  __shared__ barrier b[2];
  // ^initialization omitted
  barrier::arrival_token t[2];
  cudaMemcpyAsync(/* ... */ , b[0]);
  t[0] = b[0].arrive();
  for (int step = 0, next = 1; step < steps; ++step, ++next) {
    if (next < steps) {
      b[next & 1].wait(t[next & 1]);
      cudaMemcpyAsync(/* ... */ , b[next & 1]);
      t[next & 1] = b[next & 1].arrive();
    }
    b[step & 1].wait(t[step & 1]);
    compute();
    t[step & 1] = b[step & 1].arrive();
  }
}
#include <cuda/barrier> // ISO C++20 conforming extension
using barrier = cuda::barrier<cuda::thread_scope_block>;

__global__ void exhibit_C(/* ... */) {
  __shared__ barrier b[2];
  // ^^initialization omitted
  barrier::arrival_token t[2];
  cudaMemcpyAsync(/* ... */, b[0]);
  t[0] = b[0].arrive();
  for (int step = 0, next = 1; step < steps; ++step, ++next) {
    if (next < steps) {
      b[next & 1].wait(t[next & 1]);
      cudaMemcpyAsync(/* ... */, b[next & 1]);
      t[next & 1] = b[next & 1].arrive();
    }
    b[step & 1].wait(t[step & 1]);
    compute();
    t[step & 1] = b[step & 1].arrive();
  }
}
OUR PRODUCTIVITY GAINS FROM A100

Optimized Tensor Kernels

V100 (Launch)  V100 (6 months)  A100 (Launch)
Tens  Hundreds  Thousands

CUTLASS Relative Performance to cuBLAS (Tensor FP16)

V100  A100
88%  95%
CLOSING
UNPRECEDENTED ACCELERATION AT EVERY SCALE
Whitepaper: NVIDIA A100 Tensor Core GPU Architecture
www.nvidia.com/nvidia-ampere-architecture-whitepaper