SANDIA REPORT
SAND2021-1220
Printed February 2021

Sandia National Laboratories

Benchmarking the NVIDIA A100 Graphics Processing Unit for High-Performance Computing and Data Analytics Workloads

Stefan Seritan and Craig Ulmer

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185
Livermore, California 94550

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@osti.gov
Online ordering: http://www.osti.gov/scitech

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Road
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: https://classic.ntis.gov/help/order-methods

Benchmarking the NVIDIA A100 Graphics Processing Unit for High-Performance Computing and Data Analytics Workloads

Stefan Seritan
Dept. 08754, Quantitative Analysis and Modeling
Sandia National Laboratories
sserita@sandia.gov

Craig Ulmer
Dept. 08753, Scalable Modeling and Analysis
Sandia National Laboratories
cdulmer@sandia.gov

SAND2021-1220

ABSTRACT

The performance of NVIDIA's latest A100 graphics processing unit (GPU) is benchmarked for computing and data analytic workloads relevant to Sandia's missions. The A100 is compared to previous generations of GPUs, including the V100 and K80, as well as multi-core CPUs from two generations of AMD's EPYC processors, Zen and Zen 2. Computing workloads such as sparse matrix operations (e.g. HPCG benchmark) and numerical solver-heavy applications based on Trilinos and Kokkos see a moderate 1.5x to 2x speedups compared to the V100, consistent with the increased core count and memory bandwidth of the A100. Training and inference on machine learning (ML) models such as ResNet-50 for image classification and BERT-Large for natural language processing show the same 2x speedup over the V100.

However, these ML workloads also benefit from increased tensor core capabilities in the V100 and A100 GPUs, yielding a 3.5x speedup using a mixed (single + half) precision strategy for floating point operations. While the performance gap between GPUs and CPUs remains moderate (3x to 8x) for high-performance computing applications, these new hardware features of recent GPU generations give 50x to 100x speedups in out-of-the-box ML workloads compared to CPUs. With additional A100 features still undergoing testing (INT8, structural sparsity, multi-instance GPUs) with clear applications for ML workloads, the A100 GPU seems an extremely promising hardware accelerator for artificial intelligence (AI) and data analytics research at Sandia.

Acknowledgement

The authors would like to thank: Samuel Knight, Jerry A. Friesen, Gavin M. Baker and Joseph P. Kenny for Kahuna support and valuable discussions throughout the benchmarking process; Jerry Watkins for his help with building and running the Albany LandIce workload; and Brodderick Conner Rodriguez for his work on the `ml_benchmarks` package.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1. The NVIDIA A100

Graphics processing units (GPUs) are parallel computing architectures that excel at single-instruction multiple-data (SIMD) parallelism. The original purpose of these accelerators was computer graphics, where each pixel on the screen needs to be updated using the same formula but with different data (i.e. position in the scene). However, the same principle applies to many matrix operations needed in high performance computing (HPC) and high performance data analytics (HPDA) workloads, and GPUs have seen considerable success as accelerators for scientific applications. GPUs typically have two orders of magnitude more cores than CPUs, but much less memory and specialized optimizations.

The NVIDIA A100 GPU is NVIDIA's latest flagship device for datacenter usage, and boasts several performance improvements over previous generations. Key specifications are listed in Table 1-1 below, but effectively the A100 has 1.4x the compute performance and 1.7x the memory bandwidth over its' predecessor, the NVIDIA V100. As with all new GPU releases, it is expected that these upgrades will directly translate into moderate (e.g. roughly 2x) speedups for existing HPC codes.

Table 1-1. Summary of key specifications for the Kepler, Volta, and Ampere GPUs on Kahuna. Specifications taken from the relevant SKU: Tesla K80 [1], V100 32GB PCIe [2], and A100 40GB PCIe [3]

GPU	FP64 Compute (TFLOPS)	FP32 Compute (TFLOPS)	Tensor Compute (TFLOPS)	Memory (GB)	Bandwidth (GB/s)
K80	1.8	5.6	-	12	480
V100	8.2	16.4	112[a]	32	900
A100	9.7 / 19.5[a]	19.5 / 156[a]	312[a]	40	1600

[a] Using tensor cores

However, the most interesting improvements lay in two main new hardware developments: improved support for mixed precision and integer operations in tensor cores, and the multi-instance GPU (MIG) capability which allows each device to be subdivided. Tensor cores were a new feature in the Volta generation, specialized to accelerate operations on stacks of matrices (thus the name tensor core) with reduced precision. This is a common workload in machine learning (ML) and deep learning (DL), and marked a departure from previous trends that had only focused on improving the traditional GPU cores. In the Ampere generation, the tensor cores have been significantly improved and have increased support for low-precision and sparse operations that are common in ML/DL inference workloads. Successive generations architectures

have added low precision types, starting with Volta's FP16 support; INT8, INT4, and INT1 with Turing; and TF32, BF16, and FP64 (tensor core) with Ampere.

On the other hand, the MIG capability is a direct response to the increased use of containerization and virtualization needed by commercial cloud providers such as Amazon Web Services, Google Cloud, or Microsoft Azure. MIG allows each A100 to be partitioned and carry out multiple sandboxed calculations; in the case where each process does not require the full capabilities of the A100 (i.e. ML/DL inference), this allows greater utilization of the GPU. MIG capabilities can be configured via `nvidia-smi` but currently require elevated privileges. Until some mechanisms (e.g. SLURM plugin) are developed that enable users to turn on these features, they are likely to remain of limited use on a cluster like Kahuna and therefore outside the scope of the current work.

Lastly, it is also worth mentioning that a new SKU of A100 GPU with 80 GB of on-card memory was released during the development of this report. In addition to the obvious advantage of fitting larger models or more batches on the device, the memory bandwidth is also increased by a factor of 25% to 2000 GB/s. As many relevant workloads seem to be bandwidth-limited, this could yield additional speedups. However, the 80 GB SKU is only available in the SMX form factor rather than more traditional PCIe connectivity, requiring a specialized motherboard and node configuration. As with MIG, this remains a promising avenue of increased capability but not one that is immediately applicable at this time.

1.2. Node Configurations on Kahuna

The Kahuna HPDA cluster is a hybrid production/research environment with a variety of available hardware configurations. The five hardware configurations tested in this report are listed below in Table 1-2. The primary goal of this benchmarking is to compare the A100 (i.e. *Ampere* configuration) against GPUs of previous generations (i.e. *Kepler* and *Volta*), with a secondary objective of comparing GPU performance to modern high-core count CPUs such as the AMD EPYC processors (i.e. *Zen* and *Zen2*).

1.3. Bandwidth Performance

One straightforward benchmark is to see whether the bandwidth specifications from Table 1-1 are easily achievable. Both CPU to GPU and on-card GPU bandwidth are important. While we expect an incremental increase in on-card bandwidth in line with typical improvements from each generation, the availability of PCIe Gen 4 should give the A100 up to a 2x increase in transfer bandwidth between host and GPU memory when compared older PCIe Gen 3 accelerators.

1.3.1. Methods

The `bandwithTest` sample provided with the CUDA SDK was used to measure host-to-device and device-to-host bandwidth. Pinned host memory was used with the "shmoo" mode to test a

Table 1-2. Hardware configurations on Kahuna used for testing.

Name	CPU	Memory	GPU	I/O
Zen	Dual-socket AMD EPYC 7601 (2.2GHz, 64 cores total)	1 TB	-	PCIe Gen 3
Zen2[a]	Dual-socket AMD EPYC 7452 (2.2GHz, 64 cores total)	256 GB	-	PCIe Gen 4
Kepler	Dual-socket Intel E5-2698v3 (2.35GHz, 64 cores total)	512 GB	1 K80	PCIe Gen 3
Volta	Dual-socket Intel Xeon Gold 6130 (2.1GHz, 32 cores total)	768 GB	4 V100	CPU-GPU: PCIe Gen 3 GPU-GPU: NVLink
Ampere	Dual-socket AMD EPYC 7452 (2.2GHz, 64 cores total)	256 GB	1 A100	PCIe Gen 4

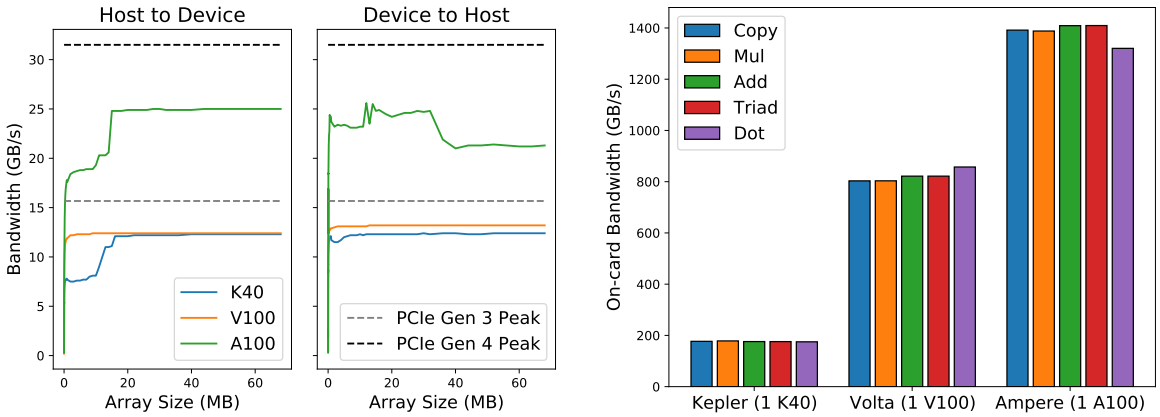
[a] Same node as *Ampere* but setting `CUDA_VISIBLE_DEVICES=` such that the A100 is not available

wide range of transfer sizes. Pinned memory cannot be swapped out of memory by the operating system, and therefore leads to higher and more stable performance. Bandwidth was averaged over 100 `cudaMemcpyAsync()` calls. Local variants of `bandwidthTest` that use warmup calls, multiple (i.e. 4 or 8) streams to pipeline the copies, or the use of `numactl` to ensure the optimal CPU-memory-GPU topology did not result in significant changes in performance.

The BabelStream (formerly GPU-STREAM) benchmark [4] was used to measure on-card memory bandwidth. BabelStream carries out four bandwidth-limited kernels: addition, multiplication, triad (a DAXPY-like operation), and a dot product. Due to their memory-bound nature (i.e. very little compute compared to load/store operations), these kernels give an accurate representation of bandwidth between global memory and the CUDA cores. An array size of 409600000 was used, resulting in a total memory usage of ≈9.8 GB and a runtime of 5 milliseconds per function call on Ampere, as per the recommendations. Increasing the array size did not change the observed performance.

1.3.2. Results and Discussion

The results for both bandwidth tests can be seen in Figure 1-1. Figure 1-1a shows that the A100 does have nearly double the CPU-GPU bandwidth of the other cards. This is in line with both the listed specification in Table 1-1 and the theoretical limits of PCIe 4 over PCIe 3 (i.e. 32 GB/s and 16 GB/s, respectively). However, the `bandwidthTest` only reaches approximately 80% of the peak theoretical bandwidth, despite attempts to use more sophisticated copy pipelining. Additionally, device-to-host transfers for the A100 show high variability and decreased performance at large transfer sizes. It is worth noting that the *Ampere* environment has AMD EPYC-based CPUs, which use a multi-chip module design that leads to a nonuniform memory access (NUMA) model. Preliminary tests with `numactl` did not increase A100 performance on *Ampere*, but testing the A100 GPU in an Intel-based system would be extremely useful to rule out any NUMA issues.



(a) CPU-GPU bandwidth. **(b) On-card GPU bandwidth.**

Figure 1-1. Performance of (a) host-to-device from the `bandwidthTest` sample and (b) on-card memory bandwidth from BabelStream. Larger throughput is higher performance, and all tests were run on 1 GPU (including 1 K40) to avoid any inter-GPU communication.

The on-card bandwidth performance in Figure 1-1b is much more straightforward, with all cards performing between 75% - 90% of the card specifications. Note that when comparing to Table 1-1, the performance for the Kepler card is given for the full K80, but only 1 K40 was tested to avoid any complications from inter-GPU communication; therefore, the expected peak bandwidth should be halved.

2. SCIENTIFIC COMPUTING

2.1. High Performance Conjugate Gradient (HPCG)

The High Performance Conjugate Gradient (HPCG) benchmark [5] is designed to emulate scientific workloads with balanced compute and memory operations. HPCG is meant to complement the High Performance LINPACK (HPL) benchmark [6], which is used in the Top500 ranking of supercomputers. Routines such as matrix-matrix multiplication in HPL scale cubically for FLOPS but only quadratically with memory access; as a result, HPL will tend to favor compute-heavy architectures such as GPUs. In contrast, the routines in the HPCG benchmark, such as sparse matrix-vector multiplication and symmetric Gauss-Seidel smoothing, scale linearly with both compute and memory operations. This leads to memory access patterns that are more difficult to optimize on GPUs. Most highly-ranked Top500 systems typically only achieve between 1-3% of the HPL throughput when running the HPCG benchmark, with a handful of systems reaching 10% of peak performance [7].

2.1.1. Methods

The HPCG benchmark was run on all five test architectures from Table 1-2. The *Zen* and *Zen2* executables were built from the HPCG 3.1 Reference Code available on the HPCG Software webpage [8] using GCC 9.3.0 and OpenMPI 4.0.2. Optimization flags were taken from an AMD EPYC handout on tuning HPC workloads [9], with the addition of the `-march=znver1` or `-march=znver2`, respectively. The December 2019 executable for CUDA 10 was used for the *Kepler* and *Volta* tests, with GCC 9.3.0, OpenMPI 3.1.3, and CUDA 10.0.130 modules loaded. The September 2020 executable for CUDA 11 was used for the *Ampere* tests, with GCC 9.3.0, OpenMPI 4.0.2, and CUDA 11.0.2 modules loaded.

System sizes were chosen based on recommendations in notes included with the December 2019 executable, generally aiming to fill a significant fraction (i.e. more than a quarter) of available memory to ensure reliable benchmarking. The *Zen* test used a local region of 128x256x256 with 64 processes, yielding a memory footprint of 383 GB, while the *Zen2* test used a region of 128x128x128 with 64 processes for a usage of 95 GB; in both cases, this corresponds to roughly 38% utilization of system memory. The GPU tests used 1 process per GPU with a local region of 256x256x256, using 8 GB of memory for the *Kepler* (33% usage) and *Volta* (25% usage) tests. For the *Ampere* test, the 3.1 HPCG executable actually uses just over 11 GB (28% usage) for the 256x256x256 local region. It is unclear what is responsible for the increase in memory usage between HPCG 3.0 and 3.1, although it does not appear to have impacted the performance numbers reported here. Tests were run with a benchmarking runtime of 1 minute, which is

typically enough to give representative performance numbers compared to official HPCG runs of 30 minutes.

2.1.2. Results and Discussion

The total performance can be seen in Figure 2-1. There is a slight increase in performance between Zen generations, while the trend is more significant for the different GPU generations: the V100 is twice the speed of the K80 and the A100 is again 1.5x faster than the V100. This tracks almost exactly with the improved memory bandwidths of the cards, as seen in Table 1-1. This supports the statement that the sparse operations in the HPCG benchmark are bandwidth-limited on the GPUs. As the GPUs are bandwidth-limited and cannot reach peak compute throughput, the performance of the CPU-based systems stay within an order of magnitude to the GPUs. The 64-core Zen systems roughly match half the performance of the K80 (i.e. one K40), although it is worth mentioning that the overall problem sizes for the Zen runs are much larger than the GPU runs as system memory is typically much larger than on-card memory.

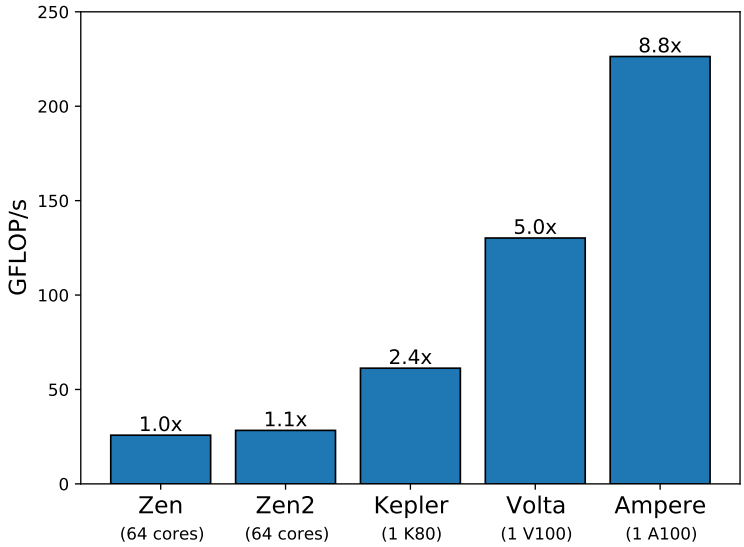


Figure 2-1. Double precision performance in GFLOPS of the reference HPCG benchmark. Larger GFLOPS is higher performance (larger = better typically colored as blue bars throughout the report). Speedups reported relative to the 64 core Zen platform.

One noteworthy point of dicussion is the use of the reference implementation on *Zen* and *Zen2*. As noted in the output, these implementations are designed to be portable and therefore may be suboptimal. In particular, data reorganization (often referred to as "coloring" variants) and performant or architecture-specific communication protocols other than MPI are allowed optimizations. These strategies have enabled higher throughput on a variety of systems [10, 11], and some vendors also provide optimized HPCG binaries (e.g. Intel MKL [12]). There exists a Kokkos-based implementation (KHPCG) [13] that contains a coloring strategy and could potentially be used to provide a consistent benchmark across all devices, as Kokkos is a hardware

abstraction library designed specifically to enable portability of scientific code among different hardware accelerators [14]. The KHPCG implementation was not used for two reasons. First, a comment in the CMake configuration notes that the colored variant of HPCG may not provide valid results, leaving only an equivalent strategy to the reference implementation. Even so, KHPCG would be preferable if it allowed consistent benchmarking across both CPUs and all three GPUs. Unfortuantely, the second issue precludes this advantage as well: at the time of writing, the cuSPARSE module of Kokkos does not run on CUDA 11 due to the use of a deprecated sparse matrix-vector routine [15]. When this bug is resolved, it is certainly worth revisiting KHPCG.

2.2. Albany LandIce Model

Many HPC applications at Sandia are built on top of Kokkos [14], a portability library for hardware accelerators, and Trilinos [16], a linear algebra suite using Kokkos. Albany [17] is a finite element partial differential equation (PDE) solver leveraging Trilinos, and the Albany LandIce model (previously known as Albany/FELIX [18, 19]) for studying ice sheets was chosen for use in the performance study of A100 GPUs. This module is actively being ported to GPUs [20, 21], and offers a unique workload important to the Sandia mod-sim community that is not captured by traditional linear algebra benchmarks.

2.2.1. Methods

Trilinos and Albany were built for all five test architectures using a combination of CMake configurations in the Albany LandIce performance test repository [22]. Specifically, the Weaver configuration informed the GPU builds while the Blake configuration informed the multicore CPU builds. Table 2-1 highlights the main differences between each architecture build for Trilinos. All configurations used the Boost 1.73.0, HDF5 1.10.6, NetCDF 4.7.3, PNetCDF 1.12.1, LAPACK 3.8.0, and OpenMPI 3.1.3 modules available on Kahuna via the `gcc7-support` or `gcc9-support` module packs. Albany then simply points to the respective Trilinos build and uses the same environment. Key configuration parameters for Albany include the `NUM_GPUS_PER_NODE` and the automatic differentiation type and size. For automatic differentiation, the `SFad` type was used with a size of 12, which is large enough to run any input from the performance test repository.

Table 2-1. Key differences for Trilinos configurations between architectures.

Name	Arch Flags	Host/Device	Compilers
Zen	`Kokkos_ARCH_ZEN`	OpenMP/-	GCC 7.5.0/-
Zen2	`Kokkos_ARCH_ZEN2`	OpenMP/-	GCC 9.3.0/-
Kepler	`Kokkos_ARCH_KEPLER37`	Serial/CUDA	GCC 7.5.0/CUDA 10.2.89
Volta	`Kokkos_ARCH_VOLTA70`	Serial/CUDA	GCC 7.5.0/CUDA 10.2.89
Ampere	`Kokkos_ARCH_ZEN2` `Kokkos_ARCH_AMPERE80`	Serial/CUDA	GCC 9.3.0/CUDA 11.0.2

In the LandIce model, the finite element assembly (FEA) was used for benchmarking as this is one of the components actively being ported to GPUs and corresponds to approximately 50% of the cost [20]. The majority of the tests in the performance repository are designed to run on several multi-GPU nodes. The enthalpy FEA test using a tetrahedron element shape (as seen in Figure 2-2) in the `green-3-20km` subdirectory was used in this work as this case fits in memory on a single GPU. Using a single GPU test is more desirable here since it is more consistent across the environments available on Kahuna, especially between *Volta* (with 4 GPUs on one node) and *Ampere* (with 4 GPUs split across four nodes). It is worth noting that for this specific test case, better absolute performance could potentially be gained by using an Albany build with an `SFad` size of 4; however, this should not effect the relative performance discussed here.

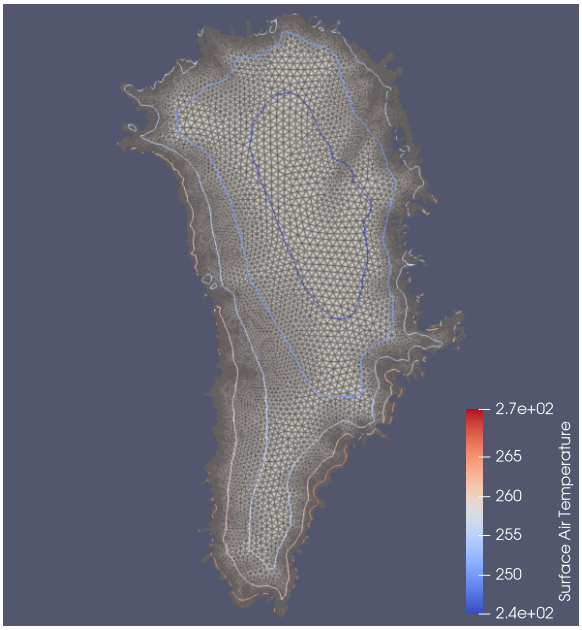


Figure 2-2. Visualization of the Greenland ice sheet (GIS) tetrahedron mesh used in the `green-3-20km` test case. Cell edges shown in gray, with contours of surface air temperature shown as a representative type of data used in the enthalpy finite element assembly calculation.

2.2.2. Results and Discussions

Figure 2-3 shows the wall time for the assembly portion of the FEA calculation excluding the initial setup. As with the HPCG benchmark, *Zen2* shows a slight improvement over *Zen* and the GPUs continue to outperform the CPUs. The *Kepler* configuration is excluded as the `green-3-20km` test case does not fit GPU memory on the K80, leading to significantly decreased performance due to host-to-device transfers. However, the performance delta is closer in this case, as the V100 is only twice as powerful as a dual-socket CPU system. The A100 is only 1.5 times faster than the V100, which is more in line with the compute improvements between the two generations.

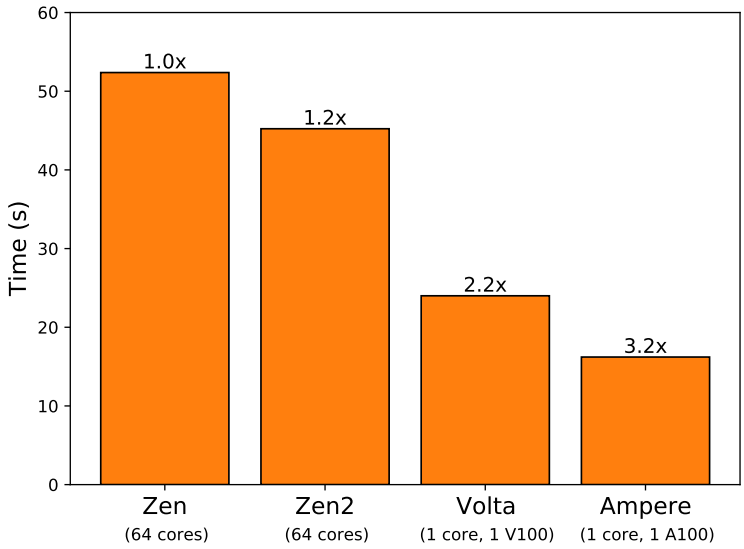


Figure 2-3. Wall time of the finite element assembly component of the Albany LandIce model run for enthalpy with tetrahedron elements on the `green-3-20km` test case. Smaller time is higher performance (smaller = better typically colored as orange bars throughout the report). Speedups reported relative to the 64 core Zen platform.

3. MACHINE LEARNING

Sandia researchers leverage machine learning techniques to analyze data and support mission needs in a wide variety of ways. The availability of low-cost, high-performance GPUs has had a profound impact on the community. For researchers that were already well-versed in machine learning techniques, GPU improvements have enabled researchers to explore larger problems and iterate over datasets more rapidly. For researchers that are new to machine learning, the availability of GPU-enhanced tools and frameworks makes it easier for individuals to apply machine learning techniques to their specific problems. In this chapter we explore the A100's performance characteristics when processing four different machine learning workloads that are relevant to Sandia applications:

Training a ResNet-50 Image Classifier: ResNet [23] is a deep convolutional neural network (CNN) architecture that can be trained to classify many different types of objects in images. While existing models have been trained to identify everyday objects, Sandia researchers often need to perform additional training to identify objects that are unique to a specific mission problem space. This training is computationally expensive and involves exposing the neural network to hundreds of thousands of labeled examples.

Object Detection in YOLOv4: Surveillance researchers require a fast and efficient means of locating and tracking objects in video data. You Only Look Once (YOLO) [24] is a one-stage object detector that is optimized to identify many objects in a video at real-time speeds.

Video Interpolation in DAIN: Surveillance researchers sometimes work with low frame-rate video sources and require a means of determining what activities took place between a pair of frames. The Depth-Aware video frame INterpolation (DAIN) model [25] extracts and exploits depth information to estimate the in-between frames of a video. Processing videos can be time intensive due to the computational overhead of each interpolation and the sheer number of frames in a video.

Natural Language Processing with BERT-Large: Google's BERT (Bidirectional Encoder Representations from Transformers) [26] is a powerful technique that researchers are leveraging in many natural language processing (NLP) tasks (e.g., word and sentence prediction, summarization, classification, etc.). Users typically start with a Google-supplied model that was trained on a massive corpus of text and then apply their own representations to "fine tune" the model to a specific need.

3.1. Image Localization with ResNet-50 v1.5

The ResNet family of models are an example of 2D convolutional neural networks (CNNs) combined with skip connections/residual blocks. These types of models have applications in image classification and localization. TensorFlow is a popular Python machine learning framework with excellent GPU support. Training and inference for image localization via the ResNet-50 model [23] was carried out as an example of a canonical TensorFlow workload.

3.1.1. Methods

NVIDIA provides Docker containers for ease of reproducibility via NVIDIA GPU Cloud (NGC), as well as reference implementations of several machine learning workloads (including both ResNet and BERT) in their `DeepLearningExamples` repository. Although the same CUDA 11.0 and cuDNN 8 libraries are available natively on Kahuna, the Docker container for TensorFlow 1 with Python 3 from November 2020 was used to ensure proper configuration of the TensorFlow installation [27]. However, there were several pushes to the `DeepLearningExamples` repository that improved Ampere performance that were not included in the container. The implementations from the commit `92829376a` were used to provide the most favorable performance for the A100 GPUs [28]. This updated implementation has an almost 30% increase in mixed precision performance (i.e. using single precision via traditional CUDA cores and half precision via tensor cores) for ResNet-50 training on the A100 and was used for all container-based TensorFlow tests.

The ResNet-50 model was trained on the ImageNet dataset. Specifically, the classification with localization task from the 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) challenge was used [29]. The 2012 ILSVRC dataset consists of 1.2 million images for training and 50,000 for validation. The full dataset is 155 GB and therefore fits nicely into system memory on all test architectures. The reported numbers load the images directly from a network-attached Ceph filesystem. Isolated runs have been done by first loading the dataset to local disk or directly into memory with no changes to performance. The model is trained to provide labels and bounding boxes for an object in each image from a choice of 1000 possible classes. Throughput is measured by the number of images that can be trained/infered upon per second. For training, 200 warmup steps were taken before average throughput was calculated on 500 batches. Batch size was 128 for single (FP32 on the V100 and FP32/TF32 on the A100) precision and 256 for mixed precision. Mixed precision runs also use a static loss scaling factor of 128 to help prevent half precision underflow during the backpropagation of the gradient [30]. For inference, 50 warmup steps were taken before average throughput was calculated on 400 batches, and no loss scaling was used as inference requires only forward propagation. The Accelerated Linear Algebra (XLA) optimized graph compiler [31] takes the execution graph of TensorFlow operations and performs operations such as kernel fusion (i.e. combining clusters of operations into single kernels for lower launch overhead) and library calls (i.e. packing operations to allow the use of optimized libraries such as cuDNN and cuBLAS). ResNet training and inference were run with and without XLA.

3.1.2. Results and Discussion

Table 3-1. Throughput for training the ResNet-50 v1.5 model for image classification with TensorFlow 1. Single precision is FP32 on *Volta* and FP32/TF32 on *Ampere*, while mixed precision (AMP) utilizes both single precision and half precision on the tensor cores. XLA is the Accelerated Linear Algebra optimized compiler. Batch sizes of 128 and 256 were used for single and mixed precision, respectively. Speedup is reported for mixed + XLA relative to single + XLA on the same GPU and also to 1 V100.

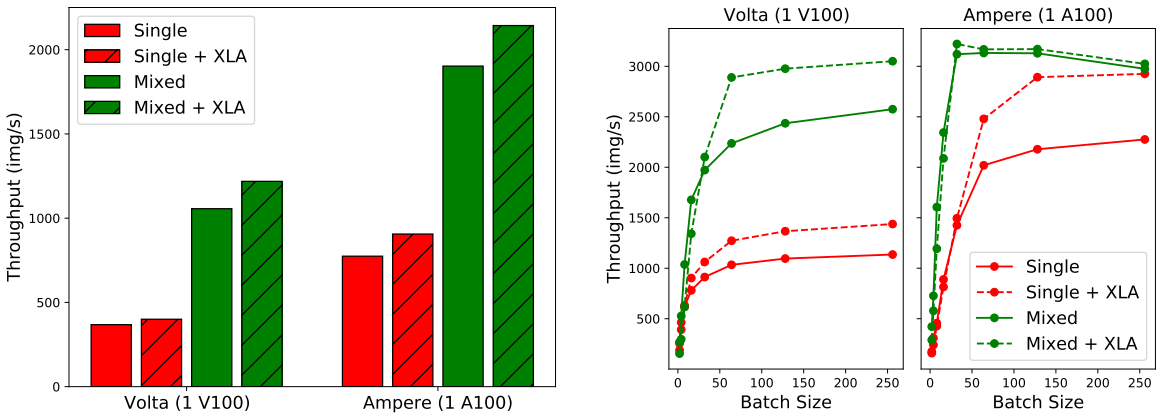
GPU	Single (img/s)	Single + XLA (img/s)	Mixed (img/s)	Mixed + XLA (img/s)	Speedup with mixed precision	Speedup from 1 V100
1 V100	368	400	1053	1217	3.04x	-
4 V100	1402	1561	3983	4723	3.02x	3.88x
1 A100	774	906	1903	2143	2.37x	1.76x

Throughput for ResNet training is shown in detail in Table 3-1, and a visual comparison of training and inference for 1 V100 and 1 A100 GPU is shown in Figure 3-1. There are several points to consider: generational improvements from Volta to Ampere, the use of XLA, and finally the use of mixed precision. The generational improvements are fairly straighforward and track with the results seen in the HPC sections. Although the theoretical compute for tensor core operations is closer to 3.0x for the A100 over the V100, a speedup between 1.5x and 2.0x is observed. This is similar to the HPC workloads and could indicate a bandwidth limited workload. In the case of ResNet, XLA results in moderate 10% to 20% speedups in training and up to 40% during inference. XLA is included by default in recent TensorFlow distributions and can be controlled both programmatically and via environment variable [32]; therefore, it is always worth considering the use of XLA whenever possible.

Mixed precision is perhaps the most interesting statistic since the greatest changes in the A100 are the use of TF32 during single precision and increased tensor core (i.e. half precision) capabilities. For ResNet-50, mixed precision yields speedups from 2.4x to 3.0x during training and 1.5x to 2.0x during inference, as seen in Table 3-1 and Figure 3-1. It is also worth noting that A100 mixed precision inference for ResNet-50 quickly plateaus, reaching maximum throughput at a batch size of 32. This differs from the behavior of all other inference runs and may indicate an additional bottleneck, requiring further investigation. Despite this discrepancy, the diminishing marginal returns in batch size also indicate that partitioning the A100 via MIG could provide additional speedups. For example, increasing batch size from 64 to 256 only gives a marginal increase in inference speed, but using a MIG setup with 4 partitions would still fit the batches of 64 in each partition's memory and potentially offer a 4x speedup based on the observed performance.

3.2. Object Detection with YOLOv4

You Only Look Once (YOLO) is a state-of-the-art framework for real-time object detection in images and videos. One of the recent iterations of the framework, YOLOv4, was chosen for benchmarking [24]. The YOLOv4 benchmarking complements the image detection ResNet tests from Section 3.1. Both architectures use 2D convolutional layers with skip connections, but



(a) Model training. **(b) Model inference.**

Figure 3-1. Performance of training and inference of the ResNet50v1.5 model for image classification with localization. Larger throughput is higher performance, with all other details matching Table 3-1.

ResNet used TensorFlow and YOLOv4 is implemented in DarkNet, a C-based neural network framework [33].

3.2.1. Methods

The AlexeyAB fork of DarkNet was compiled for each of the target architectures. The *Zen* and *Zen2* builds used GCC 9.3.0 and were built with `AVX` and `OPENMP` support. Although not included in the original Makefiles, the respective `-march` flags were also added to the compiler flags. The *Kepler* and *Volta* builds used GCC 7.5.0, CUDA 10.2.89, and cuDNN 8.0.4.30. The *Ampere* build used GCC 9.3.0, CUDA 11.0.2, and cuDNN 8.0.4.30. All GPU executables were built with `GPU` and `CUDNN` support; additionally, *Volta* and *Ampere* were also built with `CUDNN_HALF` to allow the use of mixed precision via the tensor cores.

3.2.2. Results and Discussion

The evaluation of 20288 images from the Common Objects in Context (COCO) 2017 test set [34] was carried out to measure the throughput of object detection. The results are shown in Figure 3-2. One can see that GPUs are considerably faster for this workload than the HPC benchmarks - the evaluation task takes over 4 hours on 64 cores but less than 4 minutes on a single A100 GPU.

The *Zen* results are omitted as they are more than 3x slower than the *Zen2* results, despite the nearly identical compilation procedures. As this is the only test thus far to depend on OpenMP for parallelization, it is possible that there are significant architecture improvements for shared memory parallelization in the new Zen processors that explain this performance difference. It is also worth noting that the *Kepler* results are run on a single K40 GPU, i.e. only half of the K80. Presumably a straightforward speedup of 2x can be gained by parallelizing across both GPUs

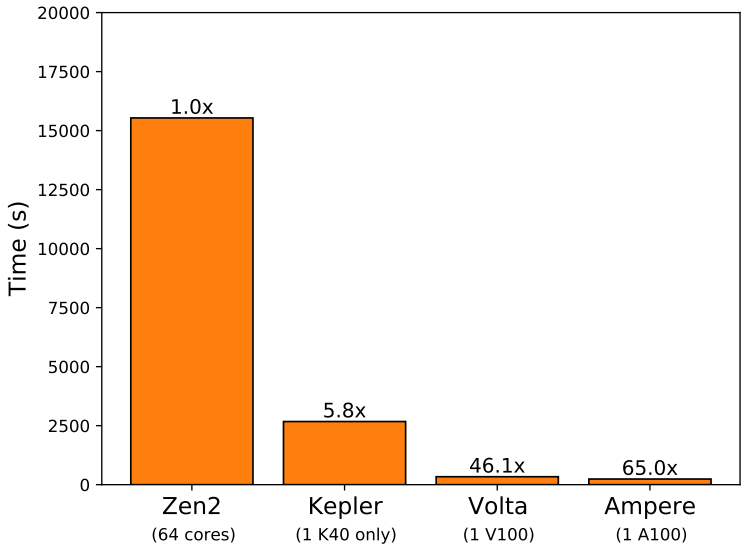


Figure 3-2. Wall time of bounding box detection in the COCO 2017 test set using YOLOv4. Smaller time is higher performance, and speedups reported relative to the 64 core Zen2 platform.

(even if only by dividing the image set into two). As the goal of this report is determining the performance of the A100 GPUs, neither of these two performance issues were investigated further.

3.3. Video Interpolation with DAIN

The Depth-Aware INterpolation (DAIN) model [25] combines previous models for motion estimation and motion compensation [35] with depth information to provide state-of-the-art performance for video frame interpolation. The tested model is implemented in PyTorch, which is another popular Python-based ML framework and alternative to TensorFlow.

3.3.1. Methods

A Python virtual environment (venv) with PyTorch 1.7.1+cu110 and TorchVision 0.8.2+cu110 served as the starting point for installation of the DAIN model. The DAIN implementation was originally designed to run on an older version of PyTorch. Necessary changes include updating the compiler arguments to use C++14 instead of C++11, as well as enabling compute capabilities 3.7, 7.0, and 8.0 for use with K80s, V100s, and A100s, respectively. Additionally, one of the plugins required mild restructuring to abide by PyTorch's new interface of only using static functions for forward/back propogation. A fork of DAIN containing these changes can be found on Sandia's CEE-GitLab [36]. The CUDA extensions and necessary plugins were built with the GCC 7.5.0, CUDA 11.0.2, and cuDNN 8.0.4.30-11.0 modules loaded. These plugins were then installed into the Python venv containing PyTorch.

Pre-trained model parameters for DAIN on the Vimeo90K triplet dataset [37] are available online and were used during this benchmark. The Middlebury dataset for optical flow [38] was used for testing as the demo script for running through the dataset is provided in the DAIN repository. The Middlebury training set consists of twelve test cases with start and end frames as input; additionally, the ground truth interpolations are also available for scoring accuracy of the model. Each picture has a resolution of 640 by 480 and no alpha channel (which cannot currently be used by the DAIN model).

3.3.2. Results and Discussion

An example interpolation is shown in Figure 3-3 and average speedups for each interpolation in the Middlebury training set between GPUs is given in Figure 3-4. The depth-aware portion of the model seems to be functioning properly as objects in the foreground (e.g. the beanbags in Figure 3-3) are prioritized over background objects (e.g. shirt and neck). No CPU timings are given as certain components of the model strictly require GPUs. Both the V100 and A100 see large speedups over the single K40, but in this case, the A100 only sees a speedup of around 1.2x relative to the V100. Several tests carried out with larger images (2048 by 1536) show the same result, indicating that this is not a problem of low resolution not providing enough compute to the GPU. It is clear that some work is needed to update the implementation to gain more performance from the A100.

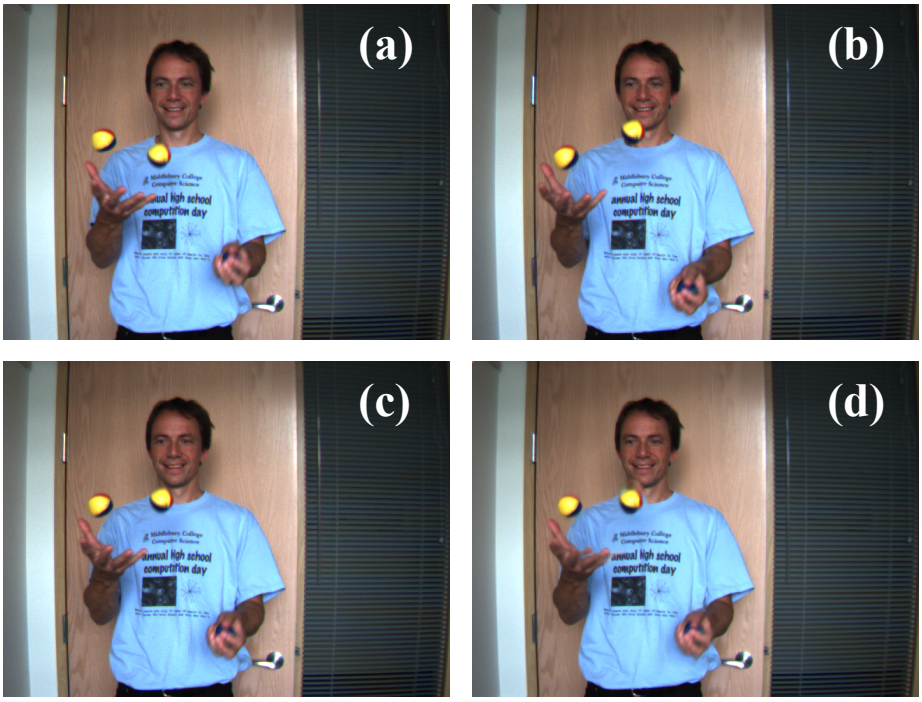


Figure 3-3. Demonstration of video frame interpolation with DAIN, with the (a) start frame, (b) end frame, (c) ground truth interpolation, and (d) output of the DAIN model. Images are the *Beanbags* case from the Middlebury test set.

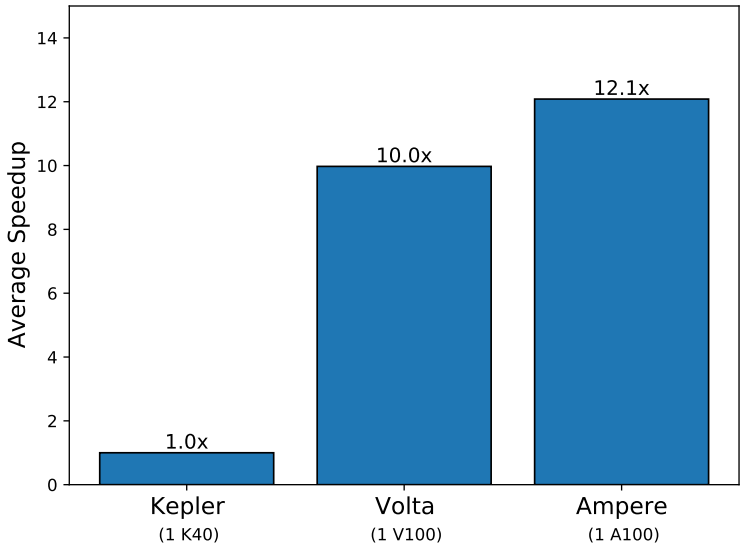


Figure 3-4. Average speedup of video frame interpolation on the Middlebury training set using DAIN. Larger speedup is higher performance, and speedups reported relative to 1 K40.

3.4. Natural Language Processing with BERT-Large

The Bidirectional Encoder Representations from Transformers (BERT) model [26] is another canonical example of a ML workflow with open-source implementations available in TensorFlow. BERT is designed for natural language processing (NLP) and consists of two phases: a pre-training phase where general relationships are formed, and a smaller fine-tuning phase to obtain better performance on specific tasks.

3.4.1. Methods

The same NVIDIA container and model implementation from the `DeepLearningExamples` repository as in Section 3.1 were used. Training and inference for the fine-tuning stage are tested here as this is the more relevant workload for single GPU nodes (whereas pre-training would benefit heavily from a dense GPU machine such as the DGX series from NVIDIA). A pre-trained BERT-Large checkpoint from NGC was obtained as the starting point for training [39]. Version 1.1 of the Stanford Question Answering Dataset (SQuAD) was used as the fine-tuning dataset [40]. Fine-tuning used a learning rate of 5e-6, a sequence length of 384, and a document stride of 128. Training throughput was taken as an average over 2 epochs, while prediction throughput is measured over the entire validation set. Batch sizes were set to avoid running out of GPU memory as listed in Table 3-2. Given the success of XLA during ResNet, XLA was turned on by default during all BERT tasks. Although a combined FP32/TF32 strategy (multiply in TF32 and accumulate in FP32) is used by default on the A100, a full FP32 run was done by using the `NVIDIA_TF32_OVERRIDE` environment flag to quantify the advantage of the TF32 format [41].

3.4.2. Results and Discussion

Table 3-2. Throughput in sequences per second for fine-tuning and validation the BERT-Large model for natural language processing. FP32 is pure single precision, FP32/TF32 is the default "single" precision mode for the A100, and AMP is mixed single and half precision using tensor cores. XLA is enabled throughout. Speedup is reported for FP32/TF32 and AMP compared to pure FP32 on each GPU.

GPU	Batch Sizes (FP32/TF32/AMP)	Run Mode	FP32 Only (seq/s)	FP32/TF32 (seq/s)	AMP (seq/s)
1 V100	10 / - / 24	Fine-tuning	14.2	-	52.4 (3.69x)
		Prediction	42.6	-	148.0 (3.47x)
1 A100	10 / 16 / 32	Fine-tuning	15.7	57.9 (3.68x)	101.5 (6.46x)
		Prediction	49.8	176.1 (3.53x)	290.9 (5.84x)

Throughput for BERT fine-tuning and prediction is listed in Table 3-2. The mixed precision results show the 2x speedup of the A100 over the V100 that is observed in other workloads. However, the investigation of pure FP32 on the A100 reveals not much improvement over the V100, while the use of FP32/TF32 provides a factor of 3.5x speedup. The implementation is clearly not optimized for pure FP32 on the A100 since one would expect a speedup between 1.5x and 2x from the V100; however, even assuming the empirically observed 2x speedup from the V100, an additional 1.75x factor is gained from using the FP32/TF32 mode. Since this is little to no penalty to accuracy incurred by the use of FP32/TF32, it is recommended to leave this as the default math mode.

4. CONTAINERIZATION

4.1. Case Study for TensorFlow

TensorFlow 2 (TF2) is a significant departure from TensorFlow 1 (TF1), primarily marked by subsuming the Keras package as a submodule of TF2. Keras enables researchers to prototype model architectures quickly, and thus TF2 + Keras is one of the most popular ML packages in use. The goals of this section are two-fold: to compare TF1 and TF2 implementations, and to compare native TF2 to the optimized NVIDIA TF2 container. The ResNet model from Section 3.1 is used as a case study.

4.1.1. Methods

Two environments for TF2 were tested: native `pip` and NVIDIA container via Singularity. Singularity is a container runtime similar to Docker but with the advantage of running in userspace (i.e. without elevated privileges) [42]. Singularity images can be built from Docker containers, and therefore provide a convenient mechanism for running containerized workflows on traditional computing clusters where Docker may not be installed. The native environment can be easily achieved by installing TF2 directly via `pip`, and will likely be the most straightforward approach for most users. A Python virtual environment (venv) was created with the `venv` module to contain the TF2 package. The CUDA 11.0 and cuDNN 8.0.4.30 modules were loaded on Kahuna while using the native TF2 venv. Note that only version 2.4.0 (released on December 11th, 2020) and beyond have built-in support for CUDA 11 and the Ampere GPUs, and that currently `pip` TF2 packages do not have XLA support. For the Singularity-based environment, the corresponding NVIDIA container with Python 3 from November 2020 to Section 3.1 was used but with TF2 instead of TF1 installed.

Instead of using NVIDIA's `DeepLearningExamples` implementation of ResNet which only runs on GPUs, we use the TF2 official model garden repository [43]. This requires installing the `tf-models-official` python package in our environment, which is simply done in the native TF2 venv but less easy for the Singularity container. A new Docker container that installed all necessary Python packages was built using the NVIDIA container as a base, making sure that the NVIDIA-provided TF2 packages were not overwritten. This Docker container was tagged as `sserita/nv-tf2-models`, saved as a Docker archive, copied to Kahuna, and rebuilt as a Singularity image (see A.6 for detailed instructions). It is worth noting a slight difference in versioning. The native environment uses version 2.4.0 for TF2 and the model garden for CUDA 11 support. Meanwhile, the version of TF2 in the Singularity container is constrained to 2.3.1, necessitating the use of version 2.3.0 for the model garden. The TF2 model garden contains a ResNet-50 implementation that can use the same 2012 ILSVRC dataset from Section 3.1, which

was used for the comparative runs here. Training throughput was taken as an average over the first epoch (excluding the first batch which includes some initialization). The *Zen2* and 1 K40 run on *Kepler* timed out before completing the first epoch; however, all other runs showed no large variation throughout the epoch, so it is expected that the timed out runs are still representative of the overall throughput.

4.1.2. Results and Discussion

Table 4-1. Comparison of ResNet-50 training in Singularity-based TF1 and native TF2. Single precision is FP32 on *Volta* and FP32/TF32 on *Ampere*, while mixed precision (AMP) utilizes both single precision and half precision on the tensor cores. Batch sizes of 128 for TF1 single precision and 256 for all other runs were used.

GPU	Environment	Single (img/s)	Single + XLA (img/s)	Mixed (img/s)	Mixed + XLA (img/s)
1 V100	TF 1, Sing.[a]	368	400	1053	1217
	TF 2, Sing.	369	392	1054	1008
	TF 2, Native	370	-	1057	-
1 A100	TF 1, Sing.[a]	774	906	1903	2143
	TF 2, Sing.	723	885	1954	1686
	TF 2, Native	711	-	1961	-

[a] Taken from Table 3-1

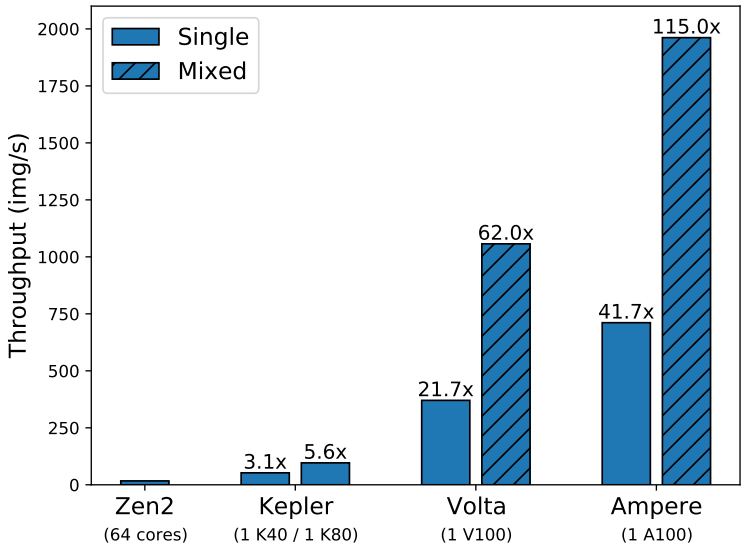


Figure 4-1. Performance of model training of ResNet50v1.5 using native CUDA & cuDNN and a standard `pip` installation of TensorFlow 2. Batch sizes of 64 were used on *Kepler* and 256 otherwise. Single precision is FP32 except for FP32/TF32 on *Ampere*, and XLA is not enabled. Larger throughput is higher performance.

A comparison of Singularity-based TF1, native TF2, and Singularity-based TF2 is shown in Table 4-1. There are no significant differences in performance between TF1 and TF2 without XLA

enabled. Singularity does not seem to add any overhead as the container-based runs match native TF2 performance. However, the mixed precision with XLA seems to have more inconsistent results for the TF2 runs. Recall that there were also some performance issues with XLA and mixed precision for the TF1 runs on Ampere that required using the latest implementation of ResNet in the `DeepLearningExamples` repository. It seems possible that a similar performance issue is occurring here and cutting-edge features such as mixed precision + XLA are not a high priority for the model garden maintainers, especially while XLA is not enabled by default yet.

Despite the issues with mixed precision and XLA, it is a clear advantage to the container-based approach that TF1 and TF2 are available with features such as XLA enabled at no additional computational cost. The alternative is compiling TensorFlow from source via Bazel and ensuring the native environment is setup properly. The only disadvantage of using Singularity is user unfamiliarity. Many users have a workflow revolving around `pip` packages and Python virtual environments, while Singularity is not mainstream yet. In this case, users would only need to know the flags to enable GPU support (i.e. `-nv`) and mounting data directories (e.g. `-B /path/to/imagenet:/path/in/container`). For more advanced use cases, such as parallelizing training over multiple nodes via Keras parameter servers, Singularity containers would also need to be launched with the correct ports exposed to enable interworker communication.

5. CONCLUSION

5.1. Role of the A100 in HPDA Workloads

In general, the moderate speedup of 1.5x to 2x can be expected for HPC codes that only use the traditional CUDA cores, correlating well with the higher compute throughput and memory bandwidth of the A100 over the V100. Compared to the only 10% increase in performance from *Zen* to *Zen2*, this is still a significant gain from previous GPUs and results in overall speedups on the order to 3x to 8x for the A100 over multi-core CPUs. Workload such as sparse numerical solvers, the Trilinos/Kokkos software stack, and the mod-sim community will all benefit from these incremental but substantial generational improvements.

Most of the new features in the A100 can be used much more heavily in machine learning workloads, as shown in Figure 5-1. In particular, the introduction of the TF32 format sees a moderate increase with no loss of accuracy. Together with using the tensor cores to train in single + half precision, these mixed precision strategies yield a factor of 3.5x in both model training and inference. Overall, the ML workloads benchmarked here give speedups of the A100 over multi-core CPUs between 50x and 100x. These performance improvements are shown for image classification, object detection, and natural language processing tasks, which are increasingly relevant to Sandia's projects.

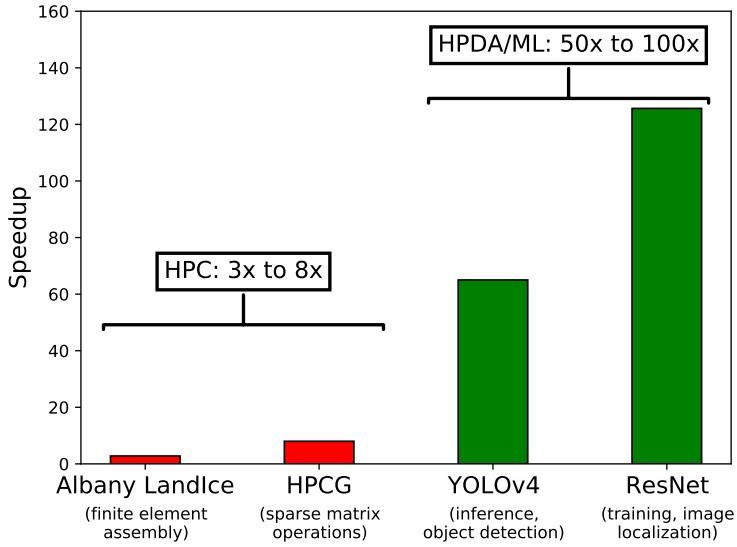


Figure 5-1. Speedup of 1 A100 over the 64-core Zen 2 CPU over traditional HPC (e.g. Albany Landlce, HPCG) and machine learning (YOLOv4, ResNet) workloads.

Several novel features of the A100 were not benchmarked here but could yield further speedups for ML and data analytic workloads. The tensor cores were tested through the use of FP16, or half precision for floating point numbers. Tensor core improvements post Turing accelerate small integer data types, which are used in the TensorRT library for low-latency ML inference. Additionally, the multi-instance GPU (MIG) capability could allow the GPU to be partitioned and more effectively utilized for models that do not require the full capabilities of the A100 GPU (i.e. multiple instances of inference can be carried in parallel). Testing for these features and their impact on workloads relevant to Sandia's mission are ongoing.

REFERENCES

[1] NVIDIA Corporation. NVIDIA® TESLA® GPU Accelerators. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/TeslaK80-datasheet.pdf, 2014. Accessed 2020-11-13.

[2] NVIDIA Corporation. NVIDIA V100 Tensor Core GPU. https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf, 2017. Accessed 2020-11-13.

[3] NVIDIA Corporation. NVIDIA A100 Tensor Core GPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/a100-80gb-datasheet-update-a4-nvidia-1485612-r12-web.pdf, 2020. Accessed 2020-11-18.

[4] Tom Deakin, James Price, Matt Martineau, and Simon McIntosh-Smith. GPU-STREAM v2.0: Benchmarking the Achievable Memory Bandwidth of Many-Core Processors Across Diverse Parallel Programming Models. In Michela Taufer, Bernd Mohr, and Julian M. Kunkel, editors, *High Performance Computing*, pages 489–507, Cham, 2016. Springer International Publishing.

[5] Jack Dongarra, Michael A. Heroux, and Piotr Luszczek. HPCG Benchmark: a New Metric for Ranking High Performance Computing Systems. Technical Report UT-EECS-15-736, Electrical Engineering and Computer Science Department, Knoxville, Tennessee, November 2015.

[6] Antoine Petitet, R. Clint Whaley, Jack Dongarra, Andy Cleary, and Piotr Luszczek. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed Memory Computers. https://www.netlib.org/benchmark/hpl/. Accessed 2020-11-12.

[7] June 2020 HPCG Results. https://www.hpcg-benchmark.org/custom/index.html?lid=155&slid=303. Accessed 2020-11-13.

[8] HPCG Software Releases. https://www.hpcg-benchmark.org/software/index.html. Accessed 2020-11-12.

[9] Advanced Micro Devices. HPC Tuning Guide for AMD EPYC™ Processors. Publication 56420, December 2018. Revision 0.7.

[10] Yulong Ao, Chao Yang, Fangfang Liu, Wanwang Yin, Lijuan Jiang, and Qiao Sun. Performance Optimization of the HPCG Benchmark on the Sunway TaihuLight Supercomputer. *Trans. Archit. Code Optim.*, 15(1):11:1–20, 2018.

[11] Daniel Ruiz, Filippo Spiga, Marc Casas, Marta Garcia-Gasulla, and Filippo Mantovani. Open-Source Shared Memory implementation of the HPCG benchmark: analysis, improvements and evaluation on Cavium ThunderX2. In *2019 International Conference on High Performance Computing Simulation (HPCS)*, pages 225–232, 2019.

[12] Intel® Optimized High Performance Conjugate Gradient Benchmark. `https://software.intel.com/content/www/us/en/develop/documentation/mkl-linux-developer-guide/top/intel-math-kernel-library-benchmarks/intel-optimized-high-performance-conjugate-gradient-benchmark.html`. Accessed 2020-11-13.

[13] KHPCG3.0. `https://github.com/hpcg-benchmark/KHPCG3.0`. Accessed 2020-11-13.

[14] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel Distrib. Comput.*, 74(12):3202–3216, 2014.

[15] KokkosKernels: Failed to compile spGEMM_cuSPARSE due to deprecated cusparseXcsrgemmNnz with CUDA 11. `https://github.com/trilinos/Trilinos/issues/8281`. Accessed 2020-11-13.

[16] Michael A. Heroux, Roscoe A. Bartlettt, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James W. Willenbring, Alan Williams, and Kendall S. Stanley. An Overview of the Trilinos Project. *ACM Trans. Math. Software*, 31(3):397–423, 2005.

[17] Andrew G. Salinger, Roscoe A. Bartlett, Andrew M. Bradley, Qiushi Chen, Irina P. Demeshko, Xujiao Gao, Glen A. Hansen, Alejandro Mota, Richard P. Muller, Erik Nielsen, Jakob T. Ostien, Roger P. Pawlowski, Mauro Perego, Eric T. Phipps, WaiChing Sun, and Irina K. Tezaur. Albany: Using Component-Based Design To Develop A Flexible, Generic Multiphysics Analysis Code. *Int. J. Multiscale Computat. Eng.*, 14(4):415–438, 2016.

[18] Irina K. Tezaur, Mauro Perego, Andrew G. Salinger, Raymond S. Tuminaro, and Stephen F. Price. Albany/FELIX: a parallel, scalable and robust, finite element, first-order Stokes approximation ice sheet solver built for advanced analysis. *Geosci. Model Dev.*, 8(4):1197–1220, 2015.

[19] Irina K. Tezaur, Raymond S. Tuminaro, Mauro Perego, Andrew G. Salinger, and Stephen F. Price. On the Scalability of the Albany/FELIX first-order Stokes Approximation ice Sheet Solver for Large-Scale Simulations of the Greenland and Antarctic ice Sheets. *Procedia Comput. Sci.*, 51:2026–2035, 2015.

[20] Jerry E. Watkins and Irina K. Tezaur. Albany: Unstructured Finite Element Codebase. `https://kokkos.org/wp-content/uploads/2019/04/KUG2019-Albany.pdf`, 2019. Accessed 2020-12-01.

[21] Jerry Watkins, Irina Tezaur, and Irina Demeshko. A Study on the Performance Portability of the Finite Element Assembly Process Within the Albany Land Ice Solver. volume 132 of *Lecture Notes in Computational Science and Engineering*, pages 177–188. Springer, Cham.

[22] ali-perf-tests. https://github.com/ikalash/ali-perf-tests. Accessed 2020-12-01.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *arXiv*, 2015.

[24] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. YOLOv4: Optimal Speed and Accuracy of Object Detection. *arXiv*, 2020.

[25] Wenbo Bao, Wei-Sheng Lai, Chao Ma, Xiaoyun Zhang, Zhiyong Gao, and Ming-Hsuan Yang. Depth-aware video frame interpolation. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2019.

[26] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv*, 2018.

[27] TensorFlow | NVIDIA NGC. https://ngc.nvidia.com/catalog/containers/nvidia:tensorflow. Accessed 2020-12-07.

[28] NVIDIA Deep Learning Examples for Tensor Cores. https://github.com/NVIDIA/DeepLearningExamples/commit/92829376a126286932496ff10d7cc655cb79af05. Accessed 2021-01-07.

[29] ImageNet Large Scale Visual Recognition Challenge 2012 (ILSVRC2012). http://image-net.org/challenges/LSVRC/2012/. Accessed 2021-01-07.

[30] TensorFlow. Mixed precision: Loss scaling. https://www.tensorflow.org/guide/mixed_precision#loss_scaling. Accessed 2021-01-07.

[31] TensorFlow. XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla. Accessed 2021-01-05.

[32] NVIDIA Corporation. Deep Learning Frameworks Documentation: XLA Best Practices. https://docs.nvidia.com/deeplearning/frameworks/tensorflow-user-guide/index.html#enabling-xla. Accessed 2021-01-05.

[33] AlexeyAB/darknet. https://github.com/AlexeyAB/darknet. Accessed 2020-12-13.

[34] COCO 2017 Object Detection Task. https://cocodataset.org/#detection-2017. Accessed 2020-12-13.

[35] Wenbo Bao, Wei-Sheng Lai, Xiaoyun Zhang, Zhiyong Gao, and Ming-Hsuan Yang. MEMC-Net: Motion Estimation and Motion Compensation Driven Neural Network for Video Interpolation and Enhancement. *IEEE Trans. Pattern Anal. Mach. Intell.*, 2018.

[36] Stefan Seritan. https://cee-gitlab.sandia.gov/sserita/dain/-/commit/e21c1d7011ca3655cb12ed35a197d358c972f413. Accessed 2021-01-25.

[37] Tianfan Xue, Baian Chen, Jiajun Wu, Donglai Wei, and William T Freeman. Video Enhancement with Task-Oriented Flow. *Int. J. Comput. Vis.*, 127:1106–1125, 2019.

[38] Simon Baker, Daniel Scharstein, J. P. Lewis, Stefan Roth, Michael J. Black, and Richard Szeliski. A Database and Evaluation Methodology for Optical Flow. *International Journal of Computer Vision*, 92(1):1–31, 2011.

[39] BERT TF checkpoint (Large, pretraining, AMP, LAMB) | NVIDIA NGC. `https://ngc.nvidia.com/catalog/models/nvidia:bert_tf_ckpt_large_pretraining_amp_lamb`. Accessed 2021-01-06.

[40] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ Questions for Machine Comprehension of Text. *arXiv*, 2016.

[41] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed Precision Training. *arXiv*, 2017.

[42] Sylabs.io. `https://sylabs.io/docs/`. Accessed 2021-02-01.

[43] TensorFlow Model Garden. `https://github.com/tensorflow/models`. Accessed 2021-01-12.

APPENDIX A. Build and Run Instructions

This appendix is designed to provide a short overview of instructions needed to build and run each experiment. For detailed instructions, please see the relevant `README.md` file in the amp-testing repository.

A.1. CPU-GPU Bandwidth

The `bandwidthTest` sample can be compiled and run via the following procedure:

1. Navigate to the bandwidthTest folder in the CUDA samples (usually `samples/1_Utilities/bandwidthTest`)
2. Load the CUDA/11.0.2 module and adjust the Makefile to point to the module using the `$CUDA_HOME` variable
3. Compile `bandwidthTest`
4. Run `bandwidthTest -mode=shmoo` to test a wide range of transfer sizes

A.2. BabelStream

The BabelStream executable can be built and run via the following procedure:

1. Clone the repository from `https://github.com/UoB-HPC/BabelStream`
2. Load the relevant CUDA module
3. Compile with `make -j CUDA.make`
4. Run with `-device 0 -s 409600000` to run on a single GPU with the array size given in Section 1.3

A.3. HPCG

HPCG needs to be handled differently based on the CPU or GPU run. For GPUs, the relevant HPCG executable needs only to be downloaded and extracted from the HPCG software page. For CPUs, the reference HPCG can be built by:

1. Download, unzip, and extract the reference code from `https://www.hpcg-benchmark.org/software/index.html`
2. Copy or symlink the `config/Make.Kahuna-*` files to the `setup` folder in the HPCG source
3. Load the appropriate GCC, GCC support, and OpenMPI modules
4. Set up a build directory
5. configure with one of the copied/symlinked Makefiles (e.g. `` `pwd`/../configure Kahuna-<arch> ``)
6. Compile with Make

Running HPCG can then be done in the following way:

1. Load the relevant modules (CUDA and OpenMPI for GPU, GCC and OpenMPI for CPU)
2. Run the HPCG executable with either `mpirun -np <num of cores>` for CPU or `mpirun -np <num of GPUs>` for GPU, and setting the X, Y, and Z dimensions as the first three command line arguments
3. Optionally set the runtime using the fourth command line argument. This was not done for this benchmarking, defaulting to 60 seconds for the benchmark execution time

A.4. Albany LandIce

The Albany LandIce model is perhaps the most complex native software stack used during this benchmarking. Luckily, much of the configuration is saved in several scripts so reproducing the build should be straightforward.

Note: These scripts used the June 2020 set of Kahuna modules, not the newer December 2020 ones. The modules can either be swapped out for the older set, or hopefully minor version updates should enable it to work on the newer modules.

1. Clone Trilinos
2. Make a build directory and copy/symlink the relevant `do-cmake-trilinos-<arch>`, `nvcc_wrapper_<arch>`, and `env.<arch>`. Note that *Zen2* uses `env.ampere` and *Zen* uses `env.volta-kepler`.
3. Source the `env.<arch>` file to load the proper modules
4. Run `do-cmake-trilinos-<arch>` to configure the build against the loaded modules

5. Build Trilinos with Make (highly recommended on a compute node with `make -j 12` at least)
6. Clone Albany
7. Make a build directory and copy/symlink the relevant directory `do-cmake-albany-<arch>` and `env.<arch>`, analogous to above.
8. Ensure that the Trilinos installation from the previous step can be found by CMake (i.e. Trilinos' `<install>/lib/cmake` is in `$CMAKE_PREFIX_PATH`).
9. Source the environment, configure, and build Albany just as with Trilinos

Running Albany is also one of the more involved steps. There was an attempt to automate this with mild success in the `gen_input.sh` script, but the overall steps are:

1. Decompose the mesh with the `decomp` utility if needed (i.e. if using more than one processor)
2. Use `Albany` to populate the mesh with requisite data for the type of finite element assembly (i.e. enthalpy or velocity)
3. Run the finite element assembly calculation in `Albany`

Both Albany steps should use `mpirun -np <num of cores/GPUs>`, just as with HPCG above. Due to their size, the meshes are not available in the repository but are available in `/home-emu/sserita/data/ali-perf-test-meshes`.

A.5. TensorFlow 1

Unlike Albany, the environment for the TF1 experiments is straightforward as only the NVIDIA container needs to be downloaded:

1. Load the Singularity module
2. Download the NVIDIA container from NGC: `singularity pull docker://nvcr.io/nvidia/tensorflow:20.11-tf1-py3`

However, the TF1 experiments require significant setup for ImageNet and SQuAD, the respective datasets for ResNet and BERT. For ImageNet:

1. Download and extract the ILSVRC 2012 training, validation, and bounding boxes datasets. This requires an account with ImageNet.
2. Preprocess the validation data by mapping each suffix to the proper synset
3. Preprocess the bounding box data for each synset
4. Generate the TFRecords from the training/validation images and preprocessed metadata

BERT preprocess for fine-tuning is less onerous as the SQuAD dataset is much smaller, but I had issues with using the automated `bertPrep.py` script. In the end I just manually downloaded the datasets after scraping the URLs out of the Downloader classes in `DeepLearningExamples` repository.

1. Download and extract the NVIDIA pretrained BERT-Large checkpoint from NGC. **Note:** Had trouble with `unzip` on Kahuna, but `unzip` worked fine inside Singularity container.
2. Download the SQuAD training, dev, and evaluation sets

As noted in Section 3.1, the best performance was observed when using a more recent implementation of the models. This workflow is as follows:

1. Clone the `DeepLearningExamples` repository
2. Copy the relevant driver scripts: `training_perf.sh` and `inference_benchmark.sh` for ResNet, and `run_squad.sh` for BERT fine-tuning.
3. Modify the scripts to use relative paths to the `DeepLearningExamples` repositories. This could have probably been achieved with no modifications and the use of the `$PYTHONPATH` variable.
4. Run the benchmarking scripts with the relevant parameters (e.g. batch size, precision, XLA) inside the Singularity containers

As noted in Section 4.1, the `-nv` and `-B` flags are needed for Singularity to enable GPUs and make the datasets available (if not in the user's home directory).

A.6. TensorFlow 2

The datasets from the TF1 experiments can be reused, so only new environments need to be set up for the experiments. For native TF2:

1. Create and activate the virtual environment
2. Install the TensorFlow and model garden packages via `pip install -r tf2-tests/requirements.txt`

For building the Docker container with the model garden prerequisites:

1. Set the working directory to `tf2-tests/model-garden-docker`
2. Build the Docker container locally. For example, `docker build . -t sserita/nv-tf2-models:nv20.11-models2.3.0`
3. Push the Docker container up to Docker Hub (e.g. `docker push sserita/nv-tf2-models:nv20.11-models2.3.0`)
4. Pull the container down with Singularity on Kahuna: `singularity pull docker://sserita/nv-tf2-models:nv20.11-models2.3.0`

Hosting the container on Docker Hub is not always desired, so there are two alternatives. First, the Sandia Docker repository available on Nexus can be used for hosting, although this requires users to be added to the `wg-nexus-deployment` metagroup. Alternatively, the Singularity container can be built on Kahuna through the use of Docker archives as follows:

1. Save the Docker image as an archive: `docker save sserita/nv-tf2-models:nv20.11-model2.3.0 -o nv-tf2-models.tar`
2. Copy the archive to Kahuna (pretty slow)
3. Build the Singularity container directly from the archive: `singularity build nv-tf2-models.sif docker-archive://nv-tf2-models.tar`

Running follows a similar philosophy where the driver script is copied out of the repository. In this case, no modifications/`$PYTHONPATH` manipulations are necessary since the model garden package is available in the environment. Note that the native and container-based runs use different driver scripts due to the different versions of TensorFlow.

1. Clone the TensorFlow Model Garden repository
2. Checkout tag `v2.4.0` or `v2.3.0` based on whether looking for native or container driver script
3. Copy the driver script `official/vision/image_classification/classifier_trainer.py`
4. Run `classifier_trainer.py` in the native or container environment

A.7. YOLOv4

The canonical YOLOv4 implementation needs DarkNet, which follows a similar process for compiling as HPCG and Albany.

1. Clone DarkNet (the AlexeyAB version)
2. Source the relevant `yolo-tests/configure/env.<arch>`
3. Copy or symlink the relevant Makefile from `yolo-tests/configure`
4. Make DarkNet

Running YOLOv4 needs the parameters and COCO test2017 dataset, both of which are lightweight. My runs are essentially the first six steps of `https://github.com/AlexeyAB/darknet#how-to-evaluate-ap-of-yolov4-on-the-ms-coco-evaluation-server`.

DISTRIBUTION

Email—Internal (encrypt for OUO)

Name	Org.	Sandia Email Address
CA Technical Library	8551	cateclib@sandia.gov

Sandia National Laboratories

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.