Architectural Requirements for Deep Learning Workloads in HPC Environments

Khaled Z. Ibrahim[1], Tan Nguyen[1], Hai Ah Nam[1], Wahid Bhimji, Steven Farrell, Leonid Oliker, Michael Rowan, Nicholas J. Wright, Samuel Williams
NERSC/CRD, Lawrence Berkeley National Laboratory

***Abstract*—Scientific machine learning (SciML) promises to have a transformational impact on scientific exploration, by combining state-of-the-art AI methods with the latest generation of supercomputers. However, to efficiently leverage ML techniques on high-performance computing (HPC) systems, it is critical to understand the performance characteristics of the underlying algorithms on modern computational systems. In this work, we present a new methodology for developing a detailed performance understanding of ML benchmarks. To demonstrate our approach we investigate two emerging SciML benchmark applications from cosmology and climate, ComsoFlow and DeepCAM, as well as ResNet-50, a well-known image classification model. We develop and validate performance models that explore the key architectural artifacts, including memory requirements, data reuse, and performance efficiency across both single- and multiple-GPU computations. Our methodology also focuses on the complexity of data-movement across storage and memory hierarchies, and leverages our performance models to capture key components of runtime execution while highlighting design tradeoffs. Although our work focuses on image-processing methods on GPU-based HPC systems, our approach is applicable to a variety of ML algorithmic domains and emerging AI accelerators. Overall, our insights will help computer architects and data scientists understand performance bottlenecks and optimization opportunities to improve SciML design and system efficiency.**

I. Introduction

Scientific computing has long been a driver for the acquisition of supercomputers, a $13.7B industry and growing [1]. The #2 (Summit), #3 (Sierra), and #5 (Perlmutter) U.S. Department of Energy (DOE) pre-exascale systems in the June 2021 Top500 List [2] were chosen through competitive procurement that includes rigorous performance benchmarking across a curated list of scientific applications representative of the scientific high-performance computing (HPC) workload [3] [4]. Performance benchmarking is crucial to evaluate and compare systems, and quantify platform characteristics needed to efficiently execute the workload [5].

Scientific machine learning (SciML) is expected to transform science and energy research and will be a driver for the DOE's future investments in HPC platforms [6]. To make informed procurement decisions on hardware that efficiently executes SciML workloads, similar benchmarking efforts are required. Although machine learning involves two phases, training to construct an accurate model and inference to use these models for prediction, we will focus on the more computationally expensive training phase relevant to HPC.

[1]Authors made equal contributions

Training performance of a machine learning model involves carefully balancing the trade-offs of *statistical efficiency* (number of epochs to solution to reach a target accuracy) with *hardware efficiency* (the time to execute a given epoch), where an epoch is a full pass over the training data. The statistical and hardware efficiency can change significantly with modifications to input data, batch size, optimizer, hyperparameters, etc. as well as choice of framework (e.g., TensorFlow or PyTorch), making the benchmark itself a moving target. Despite the industry-wide trend toward hardware specialization to improve ML performance, benchmarking activities typically focus on statistical efficiency with *time to train* as the figure of merit. Although training throughput will be the ultimate metric for a model's performance, we cannot ignore the need to quantify HPC hardware efficiency. Comprehensive ML benchmarking requires a balanced view of both statistically- and hardware-efficient execution of the SciML workload.

In this paper, we outline an approach to benchmarking the hardware efficiency of Deep Learning (DL) models in HPC environments to gain performance insights that can be translated to different architectures and systems. We demonstrate this approach for two SciML HPC applications, CosmoFlow [7] and DeepCAM [8] from the MLPerf^{TM} HPC benchmark [9], and a well-established image classification model, ResNet-50 [10] from the MLPerf Training benchmark [11]. We highlight how all three models stress or benefit from architectural features differently, despite all coming from an "image classification" foundation. Based on our results, we discuss insights on future system architectures, framework optimizations, and balancing model complexity and system characteristics. We find that:

- SciML input and activation sizes limit batching and will ultimately mandate exploitation of model parallelism.
- AI-optimized GPUs running SciML demand more PCIe, NVMe, and Lustre bandwidth than currently provided.
- Local NVMe used to feed SciML training workloads does not provide clear performance benefits at scale and should be evaluated against centralized fabric-attached storage or strong-scaling with static partitioning of training data.
- CosmoFlow moves nearly an order of magnitude more HBM data on GPUs than is necessary and sustains less than 30% of peak bandwidth.
- Data scientists should structure models to exploit unused resources to reduce time per epoch.

II. Related Work

Machine learning benchmarks and implementations are proliferating [12], [13]. Many modern ML benchmarks provide a vast array of model types (e.g., image classification, object detection, translation, reinforcement learning and recommendation), frameworks (e.g. PyTorch, TensorFlow, Caffe), distribution schemes, support mixed precision, synthetic and real data, and training and inference, such as HPE DLBS [14], DawnBench [15], MLPerf Training and Inference [11], and HPL-AI [16]. Despite the number of features to test each benchmark run, using only time (time-to-desired-accuracy) or throughput (number of samples/s) as the metric of performance makes it impossible to articulate whether performance is driven by the model, data or hardware.

Recently, machine learning studies have augmented benchmarking methodologies to distinguish hardware efficiency from statistical efficiency. DeepBench [17] measures the performance of hardware systems on basic ML operations: matrix multiplications, convolutions, recurrent layers and all-reduce. Fathom [18] breaks the execution time down by operation type across eight common models to identify performance similarities across models and scaling trends for each operation type by threads. ParaDNN [19], a parameterized benchmark suite for deep learning, provides analysis of both synthetic models and real world models to study CPU, GPU and TPU performance. They employed roofline models, heat maps of FLOPs sensitivity to hyperparameters, and a wide array of single-node analyses focusing on future TPU improvements.

Given the complexity of profiling ML workloads, tools outside of vendor tools like Google's Tensorboard and NVIDIA's Nsight are being developed. XSP [20] uses tracing to automatically create 15 different analyses (model and layer level profiles, roofline, latency, GPU occupancy, etc.) and demonstrates the utility for 65 ML models and 4 GPU generations. Studies are limited to a single node, but this is a step in the right direction. Deep500 [21] is a meta-framework to ensure benchmark analysis is consistent, reproducible, and applicable to distributed memory environments.

As HPC and ML are becoming more entwined, the need to study performance beyond a single node is imperative. Application behavior changes across scales and, in particular, when considering the entire HPC ecosystem, including network and I/O. Performance benchmarking for machine learning using HPC best practices and a scientific ML workload is still a developing area. HPC AI500 [22] provides an extensive methodology for ML performance benchmarking, including benchmarking rules and ranking for HPC AI systems. They outline the 9-layers of an HPC AI system that contribute to the system performance (e.g. hardware, OS, communication, libraries, framework, programming model, etc.) and test one layer while keeping the others constant. They demonstrate their approach on a climate dataset using the RCNN model and on ResNet-50 with ImageNet. MLPerfHPC [9] is the first benchmark providing reference implementations of full scientific machine learning models. MLPerfHPC includes data staging times to account for I/O and network impacts since SciML workloads include larger data sets than other ML benchmarks. As more use cases of scientific machine learning develop, so also will the number of benchmarks proliferate. Similar to the standard ML landscape, the SciML landscape will have much diversity and we need to be ready to characterize these applications on HPC systems.

By contrast, in this paper, we take a first principals approach to understand the ultimate performance potential of a model and analyze end-to-end execution on a target system. Doing so allows us to understand both GPU efficiency and the potential for novel architectures.

TABLE I
DEEP LEARNING BENCHMARK CHARACTERISTICS

Attribute	**CosmoFlow**	**DeepCAM**	**ResNet-50**
Domain	Cosmology	Climate	General
Benchmark	Parameter Prediction	Semantic Segmentation	Image Classification
Data Source	N-body simulation	Climate simulation	2012 ImageNet
Dataset size	5.1 TB	8.8 TB	150 GB
Input shape	128x128x128 4 channels	768x1152 16 channels	469x387 (avg) 3 channels
Target shape	4 floats	768x1152 ints	1 int
# tr samples	262,144	121,266	∼1.3 M
# tr files	262,144	121,266	1024
file format	HDF5 to TFRecord	HDF5	JPEG to TFRecord
file size	17 MB	61 MB	45 MB

III. Experimental Setup

We examine two SciML deep learning benchmarks: CosmoFlow [7] and DeepCAM [8] in the MLPerf HPC training benchmark suite [9] and the well-established image classification model ResNet-50 [10] from the MLPerf benchmark suite [11]. We evaluate these benchmarks running on 1 to 64 NVIDIA V100 GPUs and observe model utilization of the target architecture as well as the impact of system features on model training performance. In this section, we describe the benchmarks, target systems, and relevant environment details.

A. Deep Learning Benchmarks

Table I presents key attributes of our benchmarks. *CosmoFlow*, based on the 2018 work of Mathuriya et al. [7], uses the distribution and structure of dark matter to predict four cosmological parameters to describe the evolution of the universe. The CosmoFlow model uses a 3D convolutional neural network with five convolutional layers and three fully-connected layers. The MLPerf reference implementation [23] was adapted from the original work. It is written in TensorFlow with the Keras API and uses Horovod for distributed training.

The CosmoFlow dataset was generated through N-body cosmology simulations by the ECP ExaLearn team [24], binned into 3D volumetric histograms of size 512^3 with four redshift channels and stored as HDF5 files. The data was further processed into smaller samples of size 128^3, resulting in a dataset with 262,144 samples for training and 65,536 samples for testing. The data is stored as uncompressed TFRecord files [25], a recommended and optimized data format for TensorFlow. As the dataset grows, as seen in MLPerf HPC v1.0 preliminary dataset [23], `gzip` compression is used to reduce total storage size, although not used in these experiments.

The *DeepCAM* climate benchmark, based on the 2018 work of Kurth et al. [8] and awarded the ACM Gordon Bell Prize, uses deep learning to identify two extreme weather phenomena - atmospheric rivers and tropical cyclones - from background images. This automates a process that previously required climate scientists to use heuristic algorithms or hand-labeled pixel masks corresponding to these climate events. The DeepCAM model implements Google's optimized Deeplabv3+ [26] encoder-decoder architecture with the Xception feature extractor (encoder) for semantic segmentation. DeepCAM partitions the image into segments or pixel masks, analyzes and predicts pixel segmentation masks corresponding to three classes: atmospheric river, tropical cyclone, or background. The new DeepCAM implementation uses PyTorch [27] and PyTorch's native distributed library for data-parallel training, whereas the original Kurth et al. implementation used TensorFlow. Yang et al. [28] provide comparisons between the DeepCAM TensorFlow version and PyTorch versions. They demonstrate how the framework impacts component hardware utilization, but still result in similar convergence properties.

The DeepCAM dataset was created from the Community Atmosphere Model (CAM5) [29] climate simulation, which provides 16 feature channels or climate variables (water vapor, wind, precipitation, temperature, pressure, etc.) on a 1152×768 spatial grid, with a temporal resolution of 3 hours. Over 100 years of simulation data is stored in HDF5 files, used for training, testing and validation.

The *ResNet-50 v1.5* residual network image classification benchmark by He et al. [10] is widely used for image classification and as a feature extractor for computer-vision workloads. The ResNet model was the first to demonstrate training extremely deep neural networks (150+ layers) and overcoming the vanishing gradient problem. The ResNet-50 v1.5 model is a 50-layer deep convolutional neural network. ResNet-50 has been implemented in both TensorFlow and PyTorch with numerous implementations and optimizations that prevent direct comparisons of system performance.

The ResNet-50 dataset comes from the ILSVRC 2012 ImageNet classification challenge, consisting of 1.28 million training images and 50,000 validation images [30]. Images are provided in JPEG format with an average size of 469×387 RGB pixels. Images are bundled into TFRecord format for easier handling and reading using TensorFlow.

B. Target System

All experiments in this paper use NVIDIA's V100 (Volta) GPU [31], however the methodology and analysis will apply to any other GPU. The V100 provides up to: 15.7 TFLOP/s of FP32 performance, 31 TFLOP/s of FP16 performance when using the `half2` data type, and 125 TFLOP/s of FP16 performance when using NVIDIA's Tensor cores. This is coupled with over 900GB/s of bandwidth to 16GB of HBM memory and 16GB/s of host-to-device PCIe bandwidth. This produces machine balances (thresholds to be compute-bound) of approximately 138 FP16 FLOPs per HBM Byte, and over 7,812 FP16 FLOPs per PCIe byte.

In this paper we make use of NERSC's "Cori GPU" partition [32]. The Cori GPU partition is a small test bed of 18 nodes each of which contains two 20-core Xeon 6148 (Skylake) CPUs, each of which is connected to four V100 GPUs via a pair of PCIe switches. Thus, each GPU is provided up to 8GB/s of device-to-host PCIe bandwidth. All GPUs on a node are interconnected via NVLINK while each node has four dual-ported EDR InfiniBand NICs. Each Cori GPU node is equipped with an Intel SSD DC P4500 [33] 1 TB on-node NVMe storage device. Approximately 930 GB is available to user programs. The NVMe is expected to achieve a maximum 3.2 GB/s sequential read bandwidth and 279,500 IOPS random read. Cori mounts a 30.5 PB high performance Lustre file system for temporary storage of large files with a peak aggregate bandwidth of 700 GB/s. However, on a single node, the bandwidth and IOPS using NVMe surpasses Lustre, which was designed for aggregate performance.

C. Environment

In our experiments, we use the reference implementations and only vary the batch size for each benchmark. Batch size is frequently tuned in ML workloads because it is known to provide speedups. However, it does not provide a large speedup for all workloads and requires practitioners to carefully balance the benefit over the memory costs. On Cori GPU, the environment we used includes:

- TensorFlow v2.5.0, compiled with CUDA 11.2.2 and cuDNN 8.1.0 backend. We use Horovod 0.22.1 and NCCL 2.8.4 for fast all-reduces. Our experiments use `--amp` for automatic mixed precision.
- PyTorch v1.8.0, compiled with CUDA 11.1.1 and cuDNN 8.0.5 backend.

IV. Memory Requirements of DL Models

Although storage capacity might seem abundant on today's DDR- and NVMe-augmented CPUs, machine learning models and training can require enormous amounts of storage capacity. Training data can be quite large (approaching 10TB) in the case of DeepCAM, however, such data is read once per epoch and can either be stored in the file system (bandwidth permitting) or be partitioned among all nodes involved in training. Conversely, the memory required for processing a batch of samples is required by every GPU on every node. To quantify the requirements, as part of our methodology, we estimate the requisite HBM capacity for each GPU for both the forward and backward passes in training and categorize it into model parameters, buffers for input samples within a batch, and activations.

To estimate memory requirements, we examine each layer of the neural network and count the connections between one layer to the next for estimating the total model parameters as

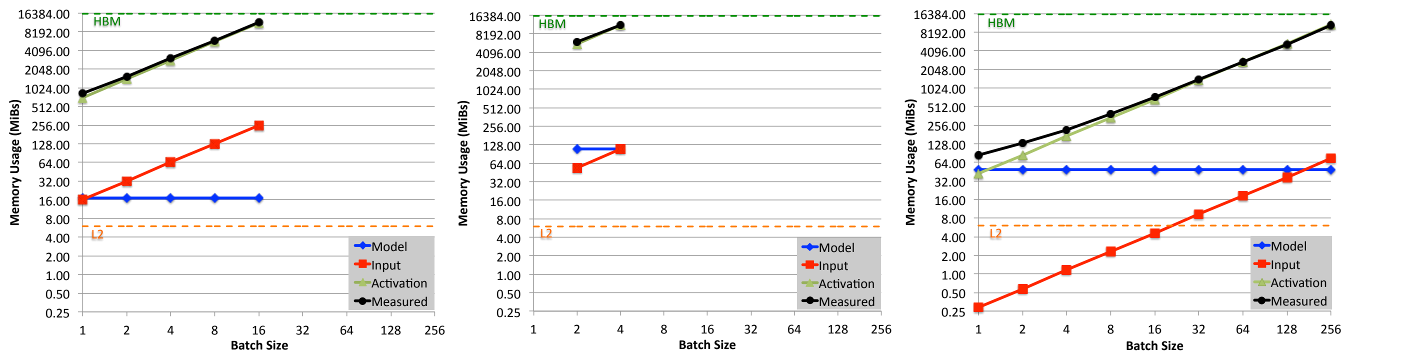

Fig. 1. Space complexity as a function of batch size for CosmoFlow (left), DeepCAM (middle), and ResNet-50 (right). CosmoFlow and DeepCAM run out of memory at lower batch sizes than ResNet-50 due to the high memory requirement for the activation. ResNet-50 activation memory requirements scale with batch size and surpass the fixed model size. Note, the DeepCAM benchmark is hard-coded to use batch sizes of at least 2.

well as the temporary buffer size for the activation. Since the training input buffer and model parameters persist throughout each training iteration, we only count the activations that cannot be immediately released. Data type is another important factor when calculating the total memory requirement. ML frameworks allocate key variables in the form of tensors, a multi-dimensional array representation with the slowest varying dimension encoding batch size and other data such as input and output of a layer. We use the AMP (automatic mixed precision) library, which allows users to allocate data in half-precision tensors. Although AMP can affect conversions of any remaining single-precision variables, we observe that this is rare in the studied applications. As such, we assume that all floating-point variables are 16-bit tensors. To validate our modeling efforts, we compare these estimates against the empirical upper bounds reported by TensorBoard.

Figure 1 plots the space complexity for these three components (model, input, activation) as well as the peak memory reported by TensorBoard as a function of batch size. Horizontal dashed lines denote V100 L2 (red dashed) and HBM (green dashed) capacity and can be used to infer performance relative to the ultimate limits by batch size. As expected, model requirements are independent of batch size while input buffers and activations scale linearly with batch size. For DeepCAM, batch size must be at least 2 due to the use of batch normalization in the model architecture.

We observe that CosmoFlow and DeepCAM space complexity is dominated by activations for all batch sizes. Moreover, their activation layers are so large (1 to 3GB/sample) that they can severely limit batch size. By comparison, ResNet-50 requires relatively little memory for activations per sample and can thus scale to large batch sizes. In CosmoFlow and DeepCAM, the activation memory is dominated by the first few layers that contain a large number of output filters.

Although model memory requirements are low for all three models, none of the models can entirely fit in the V100 L2. Nevertheless, parameters for individual layers can easily fit in the L2. As ResNet-50 is a deep model with many convolutional kernels, it has a rather large model relative to its input.

Across all three models, we see good agreement between our estimates and TensorBoard's measured peak memory usage. For ResNet-50, it is clear that for a batch size of 1, model and activation contribute roughly equally to total memory requirements. However, as batch size increases, TensorBoard peak memory is highly correlated with activation size.

As SciML models continue to grow in input size and model depth, they will either require GPUs with exponentially more memory capacity, or implementations must exploit model parallelism to reduce the per-GPU activation memory requirements. Although reducing precision (e.g. 16-bit) has been effective in the past, it is not clear how much further potential it holds. Hence, developers should prioritize increased memory capacity and model parallelization.

V. Computational Characteristics of DL Models

In this section, we extend our methodology to capture the computational characteristics when mapping deep learning workloads to GPU-accelerated node architectures. We partition this effort into four parts. First, we analyze the compute and HBM memory bandwidth requirements for training to gain insight into data locality. Next we characterize the distribution of run time and hardware efficiency. We then model and analyze the data movement across the full system hierarchy from NVMe to HBM. Finally, we profile execution on both the CPU and GPU in order to identify bottlenecks and show correlation with our performance models.

A. Compute, Memory Bandwidth, and Data Reuse

Memory capacity estimates are important for correctly sizing models, inputs, and GPUs. Equally important is analyzing GPU compute and memory bandwidth requirements toward understanding deep learning training performance. To that end, we construct a performance model that estimates the number of FLOPs and HBM bytes of data movement required for each layer, forward and backward for each model. Essentially, we walk through the DL model making the assumptions that 1) the relevant activations must be read and written on every layer (no inter-layer caching of activations) and 2) the model may be preserved in cache between layers. FLOPs are calculated assuming all convolutions are implemented via GEMMs.

Figure 2 plots the number of FLOPs and HBM data movement bytes per sample per epoch for multiple batch sizes for CosmoFlow, DeepCAM, and ResNet-50 with predictions from our model as open symbols and empirical observations from

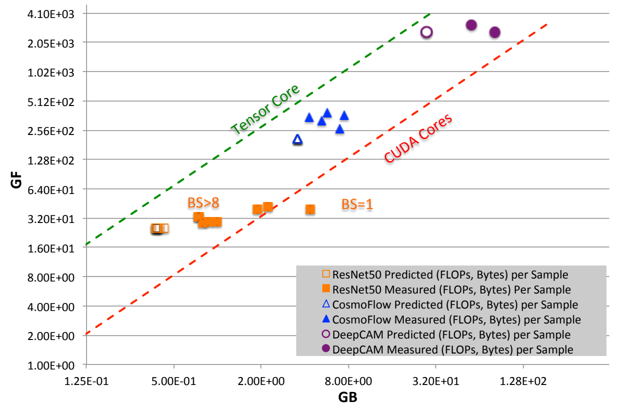

Fig. 2. We calculate the required FLOPs and Bytes for each training sample. Our estimate is more accurate as batch size (BS) increases. As the results show, it requires a large amount of compute and data movement per sample per iteration. It's worth noting that typical datasets may have O(10^5) samples, and models require a number of iterations to converge.

NVIDIA's Nsight Compute as closed symbols. Dotted lines are included to denote GPU machine balance (FP32 FLOP:Byte and TensorCore FLOP:Byte ratios).

Configurations falling below the red line lack arithmetic intensity and are thus ultimately memory-bound. Configurations falling between the red and green lines have sufficient arithmetic intensity to fully utilize the FP32 CUDA cores, but not the TensorCores. Results show that all three models have sufficient arithmetic intensity to utilize the FP32 CUDA cores, but lack the intensity to utilize tensor cores to their full potential (bandwidth-bound).

Although measured results show the V100 GPU requires roughly twice as many FLOPs and Bytes of data movement for SciML compared to our modeled estimates, ResNet-50 requires up to 10$\times$ more data movement. This is not surprising since our model does not account for caching effects or automatic framework optimizations to choose alternatives to GEMM-based convolutions.

One can also use these results to infer data reuse by comparing Figure 1 and 2. Whereas Figure 1 suggests CosmoFlow, DeepCAM, and ResNet-50 asymptotically require 0.72, 2.9, and 0.042GiB of HBM per sample, Figure 2 shows they actually move 6.4, 52, and 0.56GiB per sample. Ideally, each byte of activation memory might be written once (forward pass) and read once (backward pass). Unfortunately, the reality is that the models read or write each byte 8.8, 18, and 13 times respectively. This suggests there could be nearly an order of magnitude reduction in data movement through improvements in model implementation and cache architecture that would obviate the need for increased HBM bandwidth.

B. Isolated GPU Hardware Efficiency

Whereas the previous sections discussed memory and locality properties, they only offer potential performance. In this section, we explore whether a GPU (even in isolation) can attain either peak FP16, peak FP32, or peak bandwidth.

Traditional performance analysis has either enumerated key kernels or presented average application performance. Unfortunately, by themselves, neither is particularly informative — enumerating kernels based on run time lack insights into efficiency or potential speedup while Roofline-based approaches [34] highlight efficiency but obfuscate run time. As a result, computer scientists can be left at a loss, not only as to where to focus their efforts, but perhaps more importantly, how much performance improvement is possible.

Here we present a unified methodology for visualizing and analyzing GPU efficiency and run time. For each kernel invocation on the GPU, we compute the hardware utilization on each hardware component. We define *hardware efficiency* as the maximum hardware usage of [Tensor Cores, FP32 CUDA cores, HBM memory bandwidth], since this indicates the potential limit to further performance improvement. Figure 3 is a histogram of cumulative run time (blue) for 10 bins of hardware efficiency. Each kernel invocation contributes its run time to the bin corresponding to that invocation's hardware efficiency. Concurrently, we note the number of unique kernels in each bin (red triangle).

We observe that the bulk of the run time in CosmoFlow comes from kernels that exceed 40% hardware efficiency. Conversely, the bulk of the run time in DeepCAM comes from kernels that perform below 50% hardware efficiency, and from kernels with more distributed efficiencies in ResNet-50.

To identify which bins (kernels) programmers should prioritize, one can take the product of run time to hardware efficiency (green bars in Figure 3). When the green bar is significantly less (absolute value) than the blue bar, there is potential for substantial speedup. When run time is dominated by kernels with low hardware efficiency, there is substantial potential. Conversely, when run time is dominated by kernels with high hardware efficiency, there is little potential.

We summarize the potential speedup by comparing the summed (total) blue and green columns of Figure 3. Overall, we see the three models perform quite well on the GPU attaining better than 40% utilization. Unfortunately, the largest performance gains in applications like DeepCAM would require optimizing more than 100 kernels.

C. The Cost of Data Movement Across the Memory Hierarchy

As discussed, DL models have sufficiently high arithmetic intensity to attain CUDA core peak performance, but often lack the optimization necessary. As a thought experiment, we can ask how fast a model could execute if all kernels were well-optimized. To answer this question, we must contemplate data movement across the full storage/memory/cache hierarchy.

Using the architectural parameters that define the bandwidth limiting data movement, we estimate the cost of moving various data sets involved in the computation. For instance, the model and activation data typically reside within a GPU when data parallelism is employed. The device bandwidth defines the gap, the reciprocal of bandwidth that limits the data movement. Updating or synchronizing the model is typically limited by the interconnect bandwidth. During training, we stream the data set samples in batches from local storage or network storage, each with different constraints. For example,

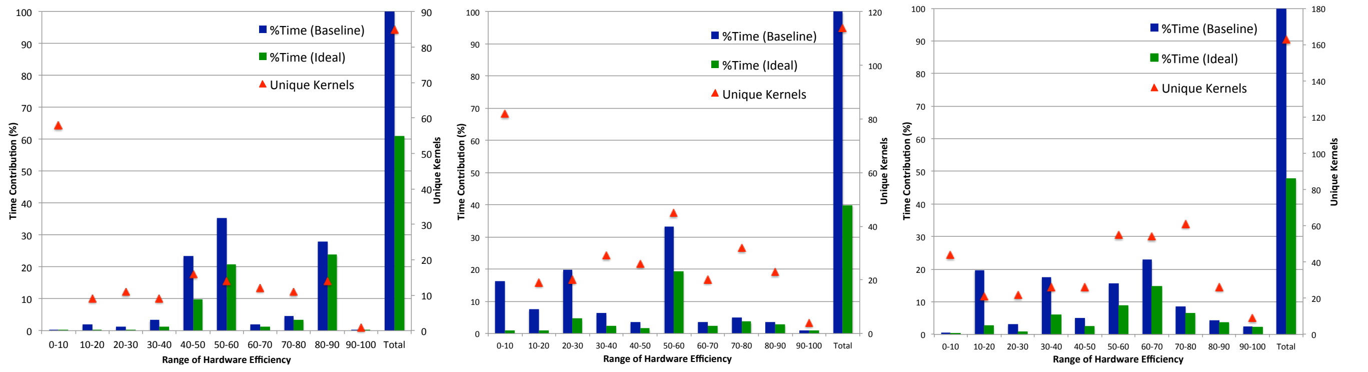

Fig. 3. Histogram of CosmoFlow (left), DeepCAM (middle), and ResNet-50 (right) kernel time (blue) and number of unique kernels (red) binned by hardware efficiency. Potential run time assuming perfect optimization is also shown (green). Note, all results used the largest batch size and a given kernel can appear in multiple bins as inputs can impact efficiency.

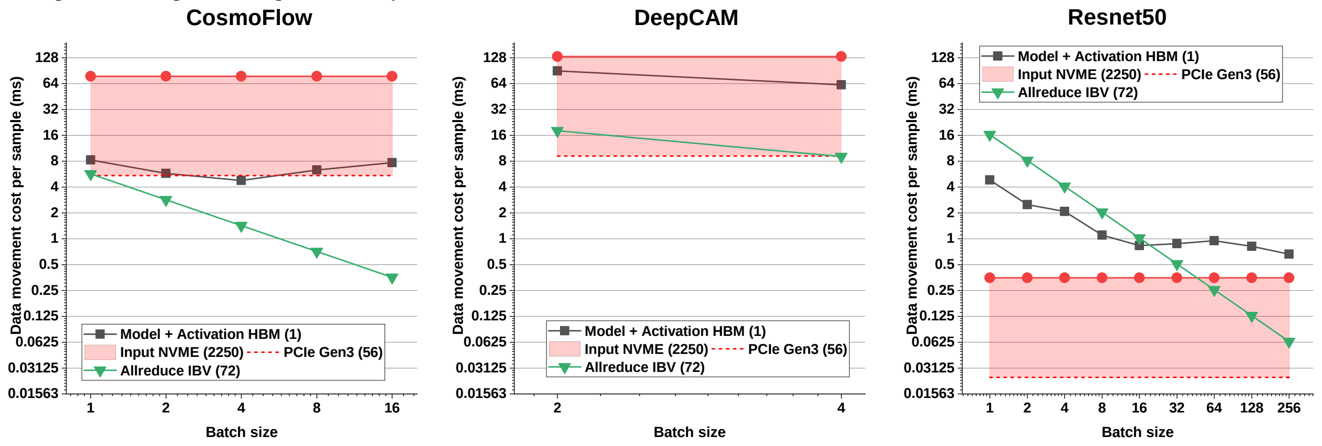

Fig. 4. Cost for data movement across the memory hierarchy (Byte $\times$ gap product) assuming the following limiting bandwidths: model and activation limited by HBM BW, input samples are limited by a range (red) of PCIe Gen 3.0 (lower) and NVMe BW (upper), and `allreduce` is limited by the NIC BW. Relative cost of byte transfer is given with each source of data, normalized to HMB(1): IBV (72), PCI Gen3 (56), NVME (2250). CosmoFlow data flow is bound by streaming input samples, while DeepCAM is bound by streaming inputs and model+activation, and ResNet-50 is bound by `allreduce` and model+activation. Our estimate is based on FP32 `allreduce` with mixed precision processing (FP16 and FP32).

if the number of samples per node is small, the data set may fit in memory, thus alleviating the stress on the storage system.

The three deep learning models in this study have different data movement costs associated with processing model data, sample data, and activation data. To model the contribution of moving these components on the overall execution time, we used the gap component, g, in the LogGP model [35] to weight the cost of moving each byte. This simple approach is justified by the size of the data of each studied source, involving at least multiple megabytes of data, making the data movement bandwidth-limited at most levels of the memory hierarchy. Furthermore, we assume the communication phase uses a simple ring-based `allreduce`. In practice, the `allreduce` activity could be split into smaller pieces to efficiently pipeline the communication and allow overlap with the backpropagation computation. We consider only the limiting level of the memory/storage hierarchy for each data source in our calculation. For instance, moving data across the cache hierarchy within the accelerator is assumed limited by the bandwidth to the memory system because the data does not fit in cache. In a distributed environment, the `allreduce` is limited by the bandwidth to the NIC, because the on-node interconnect bandwidth typically exceeds the NIC's. The movement of the training samples is limited by the bandwidth to the storage system or the linking technology between the host memory to the accelerator if samples fit in memory. Our model shows only the limits for data residing in the NVMe storage because they are predictable. For a shared Lustre-based file system, modeling the performance per node depends on the scale of the run and the behavior of the concurrently running jobs in the system. We refer readers to studies for Lustre performance characterization [36]. We note that while samples may reside in the host memory transparently, the use of NVMe requires an explicit staging of the samples, either by the user or the model.

In general, our model provides a lower bound on the execution time and assumes that a) the system perfectly overlaps transfers across the storage/memory hierarchy, b) the GPU can always sustain peak HBM bandwidth (see Section V-B), c) there is constant and negligible preprocessing time on the CPU, and d) there is no inter-sample variability that would be captured in a globally-synchronizing event like `allreduce`. Despite these unaccounted-for performance influencing factors, the model highlights the relative importance of various data movements and potential bottlenecks.

Figure 4 shows the estimated data movement cost per sample when executing the deep learning models as we increase batch size. We use gap parameters for Cori GPU, presented in Section III, to estimate the cost per byte for moving the data. Note, the width of the red region indicates the mismatch between the NVMe and PCIe bandwidths.

For CosmoFlow, we observe that the data movement for a sample to the GPU memory could significantly affect the execution time depending on where the data resides within the memory hierarchy. If the data set fits in host memory, the PCIe becomes the bottleneck. On the other hand, if the processed samples do not fit in memory, the limits depend on whether the samples fit in the NVMe storage or the Lustre file system. The cost of streaming data from NVMe is almost an order of magnitude higher than streaming the model and activation data. The impact of `allreduce` on total CosmoFlow execution time should be minimal, except if some variability increases the perceived cost for communication. The model shows that slight improvement is likely to manifest if we locally batch samples together in processing

Resent-50, Figure 4(right), shows quite the opposite relative importance of components, where the cost for `allreduce` and model+activation data movement exceed the cost of moving samples. As we increase the local batch size, the `allreduce` cost yields its dominance to the model+activation cost. Moreover, the data movement activity is generally insignificant except at large local batch size.

DeepCAM, sits in between CosmoFlow and ResNet-50, in terms of the relation between data movement activities. For samples fitting in memory, the model+activation activities dominate data movement cost. If samples need to be streamed from storage, they likely become a significant contributor to the data movement cost. Memory requirements preclude exploration of batch sizes greater than four.

D. Node-Level Bottlenecks for DL Workloads

We profile single-node execution limiting training set and epochs separating execution time into two classes: one for the GPU activities and the other for the CPU side (including NVMe/Lustre/PCIe). When CPU time dominates, strategies to improve performance include reducing the GPU:CPU computational ratio, increasing NVMe/Lustre bandwidth, or offloading preprocessing to the GPUs.

Figure 5 shows CosmoFlow's execution time does not improve significantly with increased batch size, except from a batch size of one to two, where the execution time improves by roughly 29%. This improvement is likely due to better preprocessing of the samples with threaded execution. The figure shows execution time using a large number of samples, 16K per node, exceeding what can be cached by the memory system. The utilization of the GPU is degraded by the inability to feed the sample data to the GPU efficiently. Note that the GPU execution, including both computation and communication, is only 10% of the total time. We attribute such low utilization of the GPU system to multiple reasons. In addition to the significant sample movement overhead across the memory/storage hierarchy (Figure 4), the preprocessing of

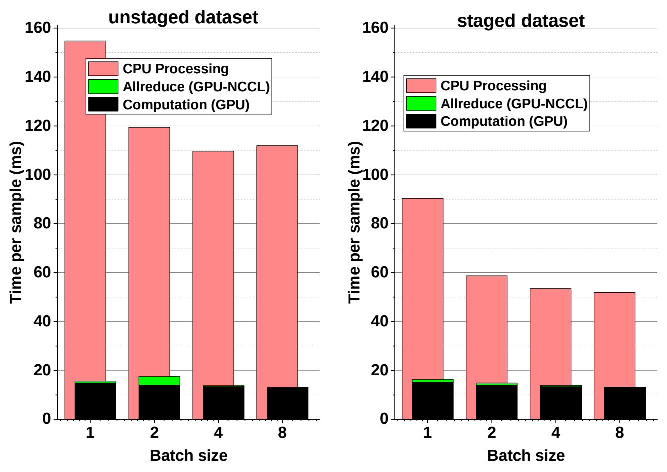

Fig. 5. CosmoFlow: Execution time decomposition per sample (per GPU) for unstaged data - Lustre (left) and staged data - NVMe (right). GPU activities are a small percentage of the total execution time due to overhead of preprocessing the sample data and the cost of feeding data to the GPU memory.

CosmoFlow data involve casting the data from integer to float and applying a log operator to all data points. These operations are time-consuming and are unfortunately not offloaded to the accelerator thus exacerbating the gap.

CosmoFlow Figure 5(right) shows the performance with staging the data to the on-node NVMe. Staging improves the performance by up to 2.16×, clearly illustrating the dependency of CosmoFlow on the efficiency of feeding data to the GPU. It, unfortunately, limits the ability to shuffle accessing samples between iterations. The performance model, illustrated in Section V-C, suggests that tackling the samples data movement issue would require either better machine balance for feeding data to the GPU, or moving the sample data compressed to the accelerator and having a decompressor optimized for the accelerator technology.

Figure 6 shows the execution time decomposition of ResNet-50 as we increase the local batch size. Increasing the sample batch size by 256× improves the processing per sample by roughly 50×. We notice that the `allreduce` time dominates the GPU processing at the low batch count as predicted by the model presented in Section V-C. The model computation starts dominating the execution on the GPU time at a large batch count. The CPU preprocessing of samples, which is not captured by our model, significantly contributes to the processing at low batch count. An optimized preprocessing on the CPU, for instance, using NVIDIA DALI [37] should prove essential for execution at low batch count. We also notice the diminishing return from batching - decreasing from an initial 2× benefit to only 20% between 128 and 256.

Figure 7, shows the dominance of the GPU compute time and a reasonably high utilization of the GPU for DeepCAM. As discussed in Section V-C, the model and activation data movement cost are dominant, especially when the sample data set fits in memory. Computationally, the arithmetic intensity associated with the model and activation is more significant than other sources of data movement.

When the dataset does not fit in memory, Figure 7(right),

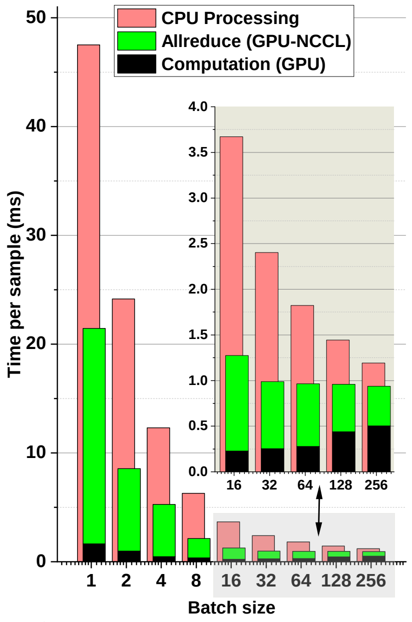

Fig. 6. ResNet-50: Execution time decomposition per sample (per GPU) as a function of local batch size. Increases in local batch size decreases the contribution of the `allreduce` to total GPU processing time. For small batch sizes, the overhead of CPU processing is significant. Overall, batching helps improve the execution time per sample as model time dominates.

we notice an increase in execution time by up to 20% for batch size two. For a batch size of four, the performance is not significantly affected by the size of the sample data set or from where it is streamed. We also notice that doubling the batch size improves the performance by up to 1.6×, when streaming data from Lustre. The GPU processing for DeepCAM is intensive enough to hide the incurred latency of moving the data for GPU memory. When the data does not fit in memory, we notice some increase in the `allreduce` time because it captures variability in the I/O operations.

E. Architectural Balance and DL Workloads

The three deep learning models presented in this study stress the system architectural features distinctively, although they belong to the image processing (interpretation) class of ML applications. For instance, although batching is widely known to improve the efficiency of deep learning processing, the amount of improvement depends on the architectural resources that the model stresses. For CosmoFlow, which stresses the data streaming of input samples, the improvement with batching is minimal because the speed of feeding data to the accelerator is a bottleneck. In contrast, DeepCAM stresses the computational requirement as we increase batching, thus yielding a consistent improvement. The challenge, though, is that the memory requirement limits the batch size. Finally, for ResNet-50, communication is a major bottleneck unless a very large batch size is used.

Understanding these characteristics is essential to both system architects and application/framework developers. For example, for systems dominated with CosmoFlow-like workloads, limiting the accelerators per attached storage bandwidth

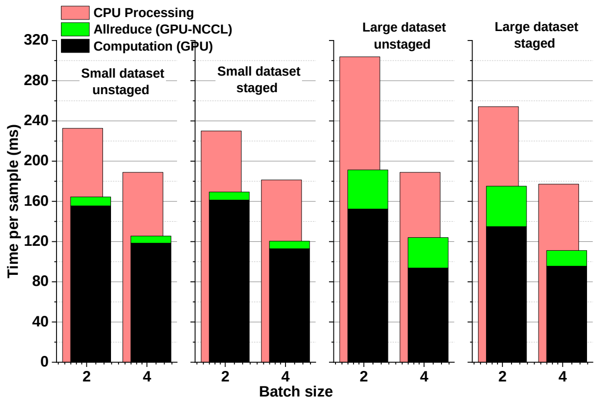

Fig. 7. DeepCAM: Execution time decomposition per sample (per GPU). GPU processing dominate the execution time. Staging to NVMe improves the processing efficiency over reading from Lustre, especially when the samples storage exceeds what can fit into cache by the memory system.

would make efficient use of resources. Meanwhile, increasing the accelerator memory for DeepCAM would allow amortizing the latency to the slow storage system.

Developers should explore architecting the model or the training sample size to balance resources utilization. For instance, reducing the sample size processed by the CosmoFlow model is likely to improve computational performance significantly. Alternatively, accelerator-optimized compression of the data could result in significant execution speedup.

VI. Scalability of DL Models

We study scaling characteristics of CosmoFlow and DeepCAM on Cori GPU to probe system architectural bottlenecks and opportunities. Previous CosmoFlow and DeepCAM studies focused on “weak” scaling behavior, keeping the overall number of training samples and local batch size fixed, and scaled out the global batch size (local batch size × number of ranks). CosmoFlow has shown good scaling behavior up to 8,192 KNL nodes on Cori, with efficiency at 77% when using the burst buffer [7]. Kurth et al. [8] showed DeepCAM has good scaling behavior up to 27,360 GPUs on Summit [38] and sensitivity to input data location (NVMe versus Lustre), choice of encode-decoder architecture and floating-point precision. Since the original works, both CosmoFlow and DeepCAM have been run at a variety of scales in the v0.7 MLPerfHPC benchmark [9] runs. ResNet-50 has been studied extensively with some notable recent scaling results from [21] and [39]. Results show optimal single-GPU throughput at a local batch size of 256 on V100 GPUs [20] and strong scaling depends on the choice of communication schemes [21].

We extend the previous data parallel “weak” scaling experiments on Cori GPU for CosmoFlow and DeepCAM, however, since the problem size is kept constant, we refer to this approach as *strong scaling* from an HPC perspective. The total data set is fixed and the number of samples processed per node is reduced with the growing number of nodes. Other ML studies [8] note that this form of scaling requires either fixing the global batch size or searching for new optimal hyperparameters to optimize statistical efficiency while increasing

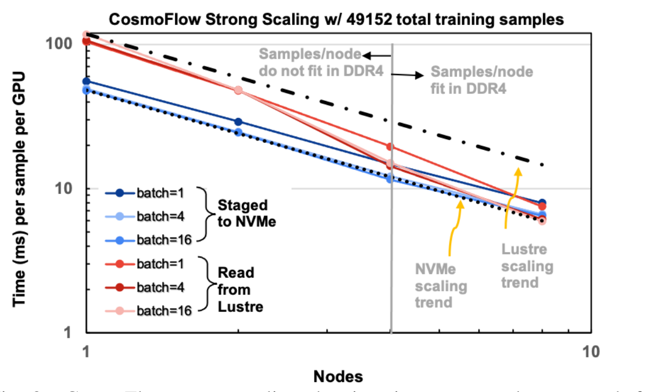

Fig. 8. CosmoFlow strong scaling showing time per sample per epoch for local batch sizes 1, 4, and 16. Samples are staged to NVMe (blue) or read from Lustre (red). The dashed lines (top) represent the perfect parallel scaling from reading for Lustre and from NVMe. Observe the strong, superlinear scaling for the Lustre tend lines at four nodes due to caching samples in DDR4.

batch size. Our experiments focus only on hardware efficiency, thus we keep the hyperparameters constant. On Cori GPU, we strong scale the training from a single node (8 GPUs) to 8 nodes (64 GPUs), and statically partition the total training data among nodes. As an artifact of Cori GPU's small size (18 nodes total), we downsize the total number of training samples to fit into the 1TB NVMe on a single node to compare the scaling behavior when staging input data to NVMe versus reading it from the Lustre file system. At 4 nodes, the number of training samples per node can fit into the 384 GB DDR4 memory. Due to the small size of the ImageNet data set (150 GB), ResNet-50/ImageNet does not benefit from the use of NVMe since the samples can fit into DDR4 memory.

Figure 8 shows CosmoFlow strong scaling using 49,152 samples. On one node, time to train per sample per GPU is roughly 2$\times$ greater when reading from Lustre than staging to NVMe. However, at four nodes, the performance difference is roughly 15% since the caching effect of fitting the samples into DDR4 reduces the benefit of staging in NVMe. With little performance difference as a function of batch size, one should choose the batch size that minimizes the number of epochs.

Interestingly, the same performance difference is not seen in Figure 9's DeepCAM results. There is roughly a 15% performance difference for a single node between staging to NVMe and reading from Lustre. The performance difference is negligible at four nodes. This difference in behavior between CosmoFlow and DeepCAM, is primarily due to the total number of respective samples. CosmoFlow has nearly five times the number of samples compared to DeepCAM that can fit into NVMe due to the smaller file size. The higher number of samples in CosmoFlow exercise the benefits of high IOPS in NVMe over Lustre resulting in the more dramatic performance difference. Practitioners should consider both the size of files and number of files to optimize SciML model performance.

A. Compression Commentary

The dataset of CosmoFlow and ResNet-50 use integer-based representations, while DeepCAM CMIP-5 dataset uses a

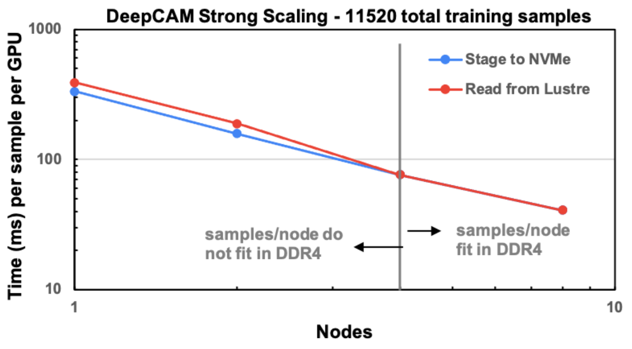

Fig. 9. DeepCAM strong scaling showing time per sample per epoch for local batch size 2. Samples are staged to NVMe (blue) or read from Lustre (red). Observe the DDR caching benefit for Lustre beyond four nodes.

floating-point-based representation. As a result, both ResNet-50 and CosmoFlow could leverage compression techniques to reduce the size of the dataset, allowing for staging into node dedicated volatile memory and reducing the pressure on I/O. The downside of such compression optimization is that it could increase the preprocessing overhead if the decompression algorithm is not particularly optimized for the target GPU architecture. Moreover, the decompression time varies depending on the content image. Such variability introduces processing imbalance that is typically captured by synchronization events, particularly the `allreduce` operation used to estimate the model's gradient.

VII. Discussion and Conclusions

Superficially, deep learning applications, including SciML, seem like the quintessential target for GPUs and specialized hardware. In this paper, we constructed a characterization methodology that allows the analysis of the computational characteristics of deep learning training models and assessment of potential bottlenecks on GPU-accelerated supercomputers. To that end, we select three deep learning models as exemplars: CosmoFlow, DeepCAM, and ResNet-50. The first two are SciML benchmarks, while the third is a well-studied image classification benchmark. Table II highlights the relevant application and computational characteristics.

We find that SciML presents a number of challenges for GPU-accelerated systems. First, the curse of dimensionality (e.g. 3D data) coupled with batch size can result in the first few layers of a SciML model exhausting a single GPU's memory capacity. A doubling of input image size in CosmoFlow or DeepCAM would result in a 8$\times$ or 4$\times$ increase in memory capacity requirements. Although one could wait a couple of years for HBM memory capacity to double, it is more realistic to decrease batch size. Unfortunately, CosmoFlow's and DeepCAM's maximum batch size is already small and further reductions might not be possible. Rather, scientists should pursue exploitation of automatic model parallelism to affect intra- and inter-layer parallelization. Inter-layer parallelization is relatively straightforward, but will likely see limited impact on such models as the first couple of layers dominate the

TABLE II
MODEL COMPUTATIONAL CHARACTERISTICS ON V100. [1]COMPARE TO SYSTEM FLOP:(LUSTRE, NVME, AND PCIE)BYTE BALANCE.

	CosmoFlow	DeepCAM	ResNet-50
Training set size	5.1 TiB	8.8 TiB	0.150 TiB
sample size	16 MiB	27 MiB	0.14 MiB
FLOPs/sample (measured)	247 GiF	2887 GiF	31 GiF
HBM/sample (limit)	1.44 GiB	5.8 GiB	0.084 GiB
HBM/sample (measured)	6.4 GiB	52 GiB	0.68 GiB
FLOP:sample Byte[1]	15,400	107,000	221,000
FLOP:HBM Byte (limit)	171	498	369
FLOP:HBM Byte (measured)	38.6	55.5	55.35
sustained HBM bandwidth	127 GB/s	270 GB/s	145 GB/s

memory capacity requirements (easy 2×). Conversely, intra-layer parallelization will be far more difficult, but ultimately more scalable (3D parallelization of 3D convolutions is a well-understood and well-studied problem in the HPC community). However, as one exploits ever more model parallelism, depending on topology, the computation may become bottlenecked on PCIe (or NVLINK) bandwidth.

Second, with approximately 1TB of NVMe and 372GB of DDR on our test machine, it is clear that only ResNet-50 can stage its entire training set on a single node. Failing that, CosmoFlow and DeepCAM must either continually read samples from the distributed file system on each epoch, or statically partition the training set among roughly a dozen nodes. In either case, the model's astronomically high FLOP:sample byte arithmetic intensity and paltry machine's FLOP:Lustre (or FLOP:NVMe) balance become relevant. For an 1PF node peak, CosmoFlow and DeepCAM will demand a bandwidth of 65GB/s and 9GB/s respectively for input samples to have any hope of attaining peak performance. Although the latter is likely attainable over either Lustre or NVMe, the former is only possible if the node's partition of the training set fits in DDR (high concurrency) and the CPUs are afforded with sufficient PCIe bandwidth. Today, on-node NVMe capacities are roughly an order of magnitude too small for CosmoFlow and DeepCAM while only the highest performing CPU-GPU interconnects are sufficient.

Practitioners wishing to preserve centralized training sets gain statistical efficiency and simplified capacity, bandwidth, and QoS apportionment, but must ensure network bandwidth is commensurate with GPU performance. Conversely, those that embrace distributed training sets free themselves of network bottlenecks, but must balance the cost of increased DDR/NVMe capacity against node concurrency, acknowledging that some SciML problems will not run at low concurrency. In the former, the first tier (NVMe) of the three-tier hierarchy is amalgamated in a fabric-attached form, while in the latter, it is eliminated altogether.

Third, an idealized AI accelerator could read an input and propagate it forward and backwards thru both model and activations holding both on-chip. Whereas a model like ResNet-50 might only need about 50MB of SRAM to achieve this, models like CosmoFlow and DeepCAM would require an unrealistic 1 to 3GB of on-chip SRAM. Even if such capacities are not possible, one might hope an architecture could simply write out activations once in the forward pass and read them once in the backward pass. With V100's roughly 900GB/s of HBM bandwidth, such an architecture might support 154-448 TFLOP/s of FP16 performance. In reality, CosmoFlow on the V100 moves between 4 and 9× more data to and from HBM. As a result, the V100's bandwidth only supports 35-50 TFLOP/s of FP16 performance — far less than its 125 TFLOP/s peak. The aforementioned numbers assume perfect overlap of (HBM) communication and computation throughout the forward and backward pass. In practice, layers are executed in a bulk synchronous manner with some layers simply lacking the parallelism to fully utilize a V100 GPU. As such, instead of sustaining 900GB/s of HBM bandwidth, we observe the V100 only sustains 127-270GB/s. As the trends in technology incentivize increasing FLOP/s over increasing bandwidth, one should expect such discrepancies to persist. As such, it will likely fall upon computer scientists to develop superior implementations and/or frameworks that ensure actual data movement is comparable to the theoretical lower bounds.

Fourth, whereas this paper was geared towards computer scientists, computer architects, and procurement officers, we believe this methodology would be of use to data scientists developing machine learning models. Rather than viewing the discussed bottlenecks as impediments, data scientists should view them as opportunities. For example, when HBM arithmetic intensity is low, data scientists can generally increase the size of convolutional kernels without substantial penalty. When data scientists find their models are bound by Lustre, NVMe, or PCIe bandwidth, they can increase model depth and complexity (not the sample size) without penalty. Such changes may seem a gratuitous use of "free" FLOPs, but in reality, they afford the designer with optimization avenues that may decrease the number of epochs required to train and/or increase ultimate model accuracy.

Ultimately, although the insights gained here were derived from analysis of execution on V100 GPUs, we believe they will apply to not only CPUs or GPUs, but also to any specialized AI accelerators. As all three of our models were basically image processing, one should not conclude the observations and bottlenecks will manifest on other SciML domains. Nevertheless, we believe our modeling methodology will apply and future work will focus on broadening our analysis.

ACKNOWLEDGEMENTS

This material is based upon work supported by the Advanced Scientific Computing Research Program in the U.S. Department of Energy, Office of Science, under Award Number DE-AC02-05CH11231 and used resources of the National Energy Research Scientific Computing Center (NERSC) which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] “Hyperion Research.” https://hyperionresearch.com/, 2021.

[2] “Top 500.” https://top500.org/, 2021.

[3] “What is CORAL?.” https://asc.llnl.gov/coral-benchmarks, 2021.

[4] “Benchmarking & workload characterization.” https://www.nersc.gov/research-and-development/benchmarking-and-workload-characterization/, 2021.

[5] M. Cordery, B. Austin, H. J. Wassermann, C. Daley, N. J. Wright, S. Hammond, and D. Doerfler, “Analysis of Cray XC30 performance using Trinity-NERSC-8 benchmarks and comparison with Cray XE6 and IBM BG/Q,” in *PMBS@SC*, 2013.

[6] N. Baker, F. Alexander, T. Bremer, A. Hagberg, Y. Kevrekidis, H. Najm, M. Parashar, A. Patra, J. Sethian, S. Wild, K. Willcox, and S. Lee, “Workshop report on basic research needs for scientific machine learning: Core technologies for artificial intelligence,” 2 2019.

[7] A. Mathuriya, D. Bard, P. Mendygral, L. Meadows, J. Arnemann, L. Shao, S. He, T. Kärnä, D. Moise, S. J. Pennycook, K. Maschhoff, J. Sewall, N. Kumar, S. Ho, M. F. Ringenburg, Prabhat, and V. Lee, “CosmoFlow: Using Deep Learning to Learn the Universe at Scale,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC ’18, IEEE Press, 2018.

[8] T. Kurth, S. Treichler, J. Romero, M. Mudigonda, N. Luehr, E. Phillips, A. Mahesh, M. Matheson, J. Deslippe, M. Fatica, Prabhat, and M. Houston, “Exascale deep learning for climate analytics,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC ’18, IEEE Press, 2018.

[9] “MLPerf Training: HPC.” https://mlcommons.org/en/training-hpc-07/, 2021.

[10] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

[11] P. Mattson *et al.*, “MLPerf Training Benchmark,” in *Proceedings of Machine Learning and Systems* (I. Dhillon, D. Papailiopoulos, and V. Sze, eds.), vol. 2, pp. 336–349, 2020.

[12] “TensorFlow Model Garden.” https://github.com/tensorflow/models, 2021.

[13] “NVIDIA Deep Learning Examples for Tensor Cores.” https://github.com/NVIDIA/DeepLearningExamples, 2021.

[14] “HPE Deep Learning Benchmarking Suite.” https://github.com/HewlettPackard/dlcookbook-dlbs/, 2021.

[15] “DAWNBench: An End-to-End Deep Learning Benchmark and Competition.” https://databricks.com/research/dawnbench-an-end-to-end-deep-learning-benchmark-and-competition, 2021.

[16] J. Dongarra, P. Luszczek, and Y. Tsai, “HPL-AI mixed-precision benchmark.” https://icl.bitbucket.io/hpl-ai/, 2021.

[17] “DeepBench.” https://github.com/baidu-research/DeepBench, 2021.

[18] R. Adolf, S. Rama, B. Reagen, G.-y. Wei, and D. Brooks, “Fathom: reference workloads for modern deep learning methods,” in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 1–10, 2016.

[19] Y. Wang, G. Wei, and D. Brooks, “Benchmarking TPU, GPU, and CPU platforms for deep learning,” *CoRR*, vol. abs/1907.10701, 2019.

[20] C. Li, A. Dakkak, J. Xiong, W. Wei, L. Xu, and W.-m. Hwu, “XSP: Across-Stack Profiling and Analysis of Machine Learning Models on GPUs,” in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 326–327, 2020.

[21] T. Ben-Nun, M. Besta, S. Huber, A. N. Ziogas, D. Peter, and T. Hoefler, “A modular benchmarking infrastructure for high-performance and reproducible deep learning,” in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 66–77, 2019.

[22] Z. Jiang, L. Wang, X. Xiong, W. Gao, C. Luo, F. Tang, C. Lan, H. Li, and J. Zhan, “HPC AI500: the methodology, tools, roofline performance models, and metrics for benchmarking HPC AI systems,” *CoRR*, vol. abs/2007.00279, 2020.

[23] “CosmoFlow TensorFlow Keras benchmark implementation.” https://github.com/sparticlesteve/cosmoflow-benchmark, 2021.

[24] “CosmoFlow datasets.” https://portal.nersc.gov/project/m3363/, 2021.

[25] “TFRecord and tf.train.Example.” https://www.tensorflow.org/tutorials/load_data/tfrecord, 2021.

[26] L. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam, “Encoder-decoder with atrous separable convolution for semantic image segmentation,” *CoRR*, vol. abs/1802.02611, 2018.

[27] “Deep Learning Climate Segmentation Benchmark.” https://github.com/sparticlesteve/mlperf-deepcam, 2021.

[28] Y. Wang, C. Yang, S. Farrell, T. Kurth, and S. Williams, “Hierarchical roofline performance analysis for deep learning applications,” *CoRR*, vol. abs/2009.05257, 2020.

[29] “NCAR Community Atmosphere Model (CAM 5.0).” https://www.cesm.ucar.edu/models/cesm1.0/cam/docs/description/cam5_desc.pdf, 2021.

[30] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “ImageNet: A large-scale hierarchical image database,” in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248–255, 2009.

[31] T. NVIDIA, “V100 gpu architecture. the world’s most advanced data center gpu. version wp-08608-001_v1. 1,” *NVIDIA. Aug*, p. 108, 2017.

[32] “Cori GPU nodes.” https://docs-dev.nersc.gov/cgpu/, 2021.

[33] “Intel SSD DC P4500 Series.” https://ark.intel.com/content/www/us/en/ark/products/99030/intel-ssd-dc-p4500-series-1-0tb-2-5in-pcie-3-1-x4-3d1-tlc.html, 2021.

[34] S. Williams, A. Waterman, and D. Patterson, “Roofline: an insightful visual performance model for multicore architectures,” *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[35] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, “LogGP: Incorporating long messages into the LogP model—one step closer towards a realistic model for parallel computation,” in *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA ’95, (New York, NY, USA), p. 95–105, Association for Computing Machinery, 1995.

[36] T. Patel, S. Byna, G. K. Lockwood, and D. Tiwari, “Revisiting i/o behavior in large-scale storage systems: The expected and the unexpected,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’19, (New York, NY, USA), Association for Computing Machinery, 2019.

[37] “NVIDIA Data Loading Library (DALI).” https://docs.nvidia.com/deeplearning/dali/, 2021.

[38] S. S. Vazhkudai *et al.*, “The design, deployment, and evaluation of the CORAL pre-exascale systems,” 7 2018.

[39] Y. Ren, S. Yoo, and A. Hoisie, “Performance analysis of deep learning workloads on leading-edge systems,” *CoRR*, vol. abs/1905.08764, 2019.