Impacts of floating-point non-associativity on reproducibility for HPC and deep learning applications

Sanjif Shanmugavelu[‖§*], Mathieu Taillefumier[‖†*], Christopher Culver[§], Oscar Hernandez[‡], Mark Coletti[‡] and Ada Sedova[‖*‡]

[§]Maxeler Technologies, a Groq Company. 3 Hammersmith Grove, W6 0ND, London, UK

[†]ETH Zurich/Swiss National Supercomputing Centre (CSCS). Via Trevano 131, 6900 Lugano, Switzerland

[‡]Oak Ridge National Laboratory. Oak Ridge, TN, USA

[*]Corresponding emails: sshanmugavelu@groq.com tmathieu@ethz.ch sedovaaa@ornl.gov

***Abstract*—Run to run variability in parallel programs caused by floating-point non-associativity has been known to significantly affect reproducibility in iterative algorithms, due to accumulating errors. Non-reproducibility can critically affect the efficiency and effectiveness of correctness testing for stochastic programs. Recently, the sensitivity of deep learning training and inference pipelines to floating-point non-associativity has been found to sometimes be extreme. It can prevent certification for commercial applications, accurate assessment of robustness and sensitivity, and bug detection. New approaches in scientific computing applications have coupled deep learning models with high-performance computing, leading to an aggravation of debugging and testing challenges. Here we perform an investigation of the statistical properties of floating-point non-associativity within modern parallel programming models, and analyze performance and productivity impacts of replacing atomic operations with deterministic alternatives on GPUs. We examine the recently-added deterministic options in PyTorch within the context of GPU deployment for deep learning, uncovering and quantifying the impacts of input parameters triggering run to run variability and reporting on the reliability and completeness of the documentation. Finally, we evaluate the strategy of exploiting automatic determinism that could be provided by deterministic hardware, using the Groq LPU™ accelerator for inference portions of the deep learning pipeline. We demonstrate the benefits that a hardware-based strategy can provide within reproducibility and correctness efforts.**

***Index Terms*—Reproducibility of results, floating-point arithmetic, parallel programming, high-performance computing, deep learning**

I. Introduction

Run to run variability of a program, despite identical inputs and software stack, is usually assumed to be negligible, even in heterogeneous parallel programming [1]. However, variability caused by floating-point non-associativity (FPNA) coupled with asynchronous parallel operations such as reductions can be substantial [2], especially for massively parallel programs using iterative stochastic routines, such as those implementing optimization algorithms like conjugate gradient [3]. It can also mask errors within threshold-based correctness testing schemes [4], which are often used in scientific computing programs such as molecular simulation [5]–[7], making debugging difficult. Deep neural network (DNN) training also involves iterative, stochastic algorithms coupled with non-linear activation functions. This combination has been found to cause extreme sensitivity to bit-level numerical changes, such as those caused by FPNA [8]–[10]. DNN inference can also suffer from such sensitivity, due to the influence of non-linear activation functions, although the effects may be reduced due to the absence of the compounding errors caused by the iterative training schemes. Recently, a number of studies have shown that this run to run variability in the full DNN training and inference pipeline can lead to unacceptably large differences in the predictions produced by a model. They have thwarted efforts to release reproducible commercial applications in safety-critical sectors such as medical diagnostics [11] and autonomous driving [8]. In addition, recent reports have found that all of the major deep learning (DL) software frameworks such as TensorFlow and PyTorch contain hundreds of bugs originating from all software levels, including environment misuse, incorrect assignment, and concurrency issues such as data races [12]; bugs in these frameworks can be disastrous [13], especially when they are silent [14], and high runtime variability can make it extremely difficult to detect bugs in these deep, multi-language, multi-level parallel stacks.

Within scientific high-performance computing (HPC), the incorporation of DL into traditional approaches for simulation has become increasingly popular [15]–[17]. Molecular simulation, for example, has used DNN models for interatomic potentials, which promise quantum mechanical accuracy at the cost of simpler, empirical models; this translates to speedups of several orders of magnitude and the approach received the

[‖]Equal contributions

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00028

ACM Gordon Bell Award in 2020 [18]. Our preliminary tests using identical inputs for training and inference pipelines for these types of models revealed a level of non-reproducibility for prediction of forces that would be unacceptable for traditional quantum mechanical HPC programs [19]. Besides non-deterministic kernels within the deep learning framework itself, any external kernels programmed by the scientific simulation developers which contain non-determinism, that feed into training pipelines [18], can also introduce large runtime non-reproducibility.

Here we present a systematic analysis of the effects of FPNA within asynchronous parallel kernels on graphics processing unit (GPU) accelerators, together with several metrics that quantify this variability, within parallel programming schemes and in PyTorch functions, as well as within full end-to-end training and inference pipelines. We assess the impact of these effects on the ability to perform correctness testing and to produce reproducible scientific results. We also study solutions, using programming approaches, and via deterministic hardware for which we perform experiments with the GroqLPU™ deterministic inference chip.

II. Metrics for measuring the variability of non-deterministic functions

We defined metrics V for quantifying the run to run variability in output values for implementations of functions with scalar and multidimensional (array) inputs and outputs, following a similar approach used in error analysis [20]. These are explained below. In all cases, $V = 0$ if two implementations are bitwise identical, nonzero otherwise, and increasing as variability increases.

1) Scalar-valued outputs: We use $V_s(f) = 1 - |f_{\text{ND}}/f_{\text{D}}|$ to quantify the bitwise non-determinism between the outputs of two implementations of some function f, where the subscripts ND and D label the non-deterministic and deterministic implementations respectively.

2) Array outputs: To quantify the bitwise variability of a non-deterministic implementation of a function producing a multidimensional array output, we define two different metrics. Let two implementations of a function f be f_1 and f_2, which produce as outputs the arrays $\mathbf{A}$ and $\mathbf{B}$, respectively, each with dimensions $d_1, d_2, \ldots, d_k$ and D total elements. The first metric is the elementwise relative mean absolute variation (ERMV),

$$V_{\text{ermv}}(f) = \frac{1}{D} \sum_{i_1=1}^{d_1} \cdots \sum_{i_k=1}^{d_k} \frac{|A_{i_1,\ldots,i_k} - B_{i_1,\ldots,i_k}|}{|A_{i_1,\ldots,i_k}|}. \quad (1)$$

The second metric, the count (C) variability, measures how many elements in the multidimensional array are different,

$$V_c(f) = \frac{1}{D} \sum_{i_1=1}^{d_1} \cdots \sum_{i_k=1}^{d_k} \mathbf{1}(A_{i_1,i_2,\ldots,i_k} \neq B_{i_1,i_2,\ldots,i_k}) \quad (2)$$

where $\mathbf{1}(\cdot)$ is the indicator function, which is 1 if the condition inside the parentheses is true, and 0 otherwise. Each of these metrics is zero if and only if the two multidimensional arrays $\mathbf{A}$ and $\mathbf{B}$ are bitwise identical. The count variability V_c informs us of the percentage of varying array elements between the two output arrays, while the V_{ermv} produces a global metric for the variability of the array outputs.

III. Programming deterministic parallel sums

In this section we first introduce FPNA impacts for the parallel sum, then demonstrate several deterministic programming solutions, for GPUs using CUDA, and also using OpenMP for CPU or GPU. We then examine the performance impacts of these solutions, as well as programming productivity considerations. Parallel sums, as we will show in the subsequent sections, contribute some of the most substantial sources of variability to PyTorch functions on GPUs, and are thus responsible for concerning non-determinism in real-world DL applications such as graph neural network (GNN)s, which make heavy use of them. A concerning connection to molecular physics and chemistry applications in scientific computing is the increased popularity of GNNs for surrogate models in these applications [21], [22], especially considering the stringent precision requirements for these applications, as discussed below.

Consider the sum $S_{\text{D}} = \sum_i^n x_i$ that we evaluate with a deterministic algorithm (the summation order is always the same). The simplest implementation adds the floating-point numbers (FP) x_i in serial in the order they are stored. When parallelized with asynchronous operations and executed in an unspecified order, this is equivalent to applying a random permutation P to the series x_i before computing the serial sum, $S_{\text{ND}} = \sum_i^n x_{P(i)}$. To demonstrate the magnitude of the variability, we can generate lists x_i of double precision floating point numbers (FP64) numbers of various lengths using Python, with, for example, $x_i \in \mathcal{N}(0, 1)$ the normal distribution of zero mean and $\sigma = 1$, and computed S_{D} and S_{ND} before and after applying a random permutation P to x_i, repeating ten times; Table 1 shows the results. The variability can be larger than the tolerance of some of the correctness tests included in computational physics and chemistry programs; this same result obtains with a Boltzmann (exponential) distribution as well, which is the expected distribution for such calculations. For example, the quantum mechanics simulation program CP2K [5] uses a tolerance-based correctness testing scheme, and some of the tests have tolerances as low as 10^{-14} for values such as energy. This illustrates the problems

TABLE 1
Effects of permutations on sums of floating-point numbers

size	$\mathbf{S_{nd} - S_d}$	$\mathbf{V_s}$
100	$-1.776356839400250\text{e}^{-15}$	$-2.220446049250313\text{e}^{-16}$
1000	$1.776356839400250\text{e}^{-15}$	$-6.661338147750939\text{e}^{-16}$
1000	$5.329070518200751\text{e}^{-15}$	$-1.776356839400250\text{e}^{-15}$
10000	$-2.486899575160351\text{e}^{-14}$	$1.110223024625157\text{e}^{-16}$
10000	$3.197442310920451\text{e}^{-14}$	$8.881784197001252\text{e}^{-16}$
100000	$2.842170943040401\text{e}^{-13}$	$1.110223024625157\text{e}^{-16}$
100000	$5.684341886080801\text{e}^{-14}$	$1.110223024625157\text{e}^{-16}$
1000000	$4.263256414560601\text{e}^{-13}$	$3.108624468950438\text{e}^{-15}$
1000000	$1.705302565824240\text{e}^{-13}$	$2.220446049250313\text{e}^{-15}$

that FPNA can introduce in correctness testing schemes for computational tasks with stringent accuracy requirements. In addition, such programs often employ iterative solvers such as conjugate gradient, leading to accumulating FP errors which can approach or exceed 20% after six or seven iterations using double precision, as has been shown for HPC systems such as massively multithreaded machines like the Cray XMT [3].

A. Examples of deterministic parallel sum implementations on GPUs with CUDA/HIP

A difficulty for programming on GPUs is the absence of a global synchronization barrier at the kernel level. To avoid races one can use (i) a single `atomicAdd` function, (ii) the implicit global synchronization happening when multiple kernels are added to the same stream (iii) one GPU thread to calculate the sum, (iv) the `__threadfence` instruction.

```
1
2  // implementation (AO) of the sum
3  __global__ void reduce_atomic_only(double *sdata
       , int size, double *res) {
4      const int tid = threadIdx.x + blockIdx.x *
       blodkDim.x;
5      if (tid >= size__) return;
6      atomicAdd(res, sdata[tid]);
7  }
8
9  __device__ int retirementCount = 0;
10 // Implementation of the SPRG method
11 __global__ void reduce_sprg(double *sdata, int
       size, double *res) {
12     int tid = threadIdx.x + blockIdx.x *
       blodkDim.x;
13     extern __shared__ double smem[];
14
15     if (tid >= size) smem[threadIdx.x] = 0.0;
16     else smem[threadIdx.x] = sdata[tid];
17     __syncthreads();
18
19     for (int offset = blockDim.x / 2; offset >
       0; offset  /= 2) {
20         if (threadIdx.x < offset)
21             smem[threadIdx.x] += smem[threadIdx.
       x + offset];
22         __syncthreads();
23     }
24
25     if (threadIdx.x == 0) res[blockIdx.x] = smem
       [0];
26     __threadfences();
27
28     bool __shared__ amLast = false;
29     if (threadIdx.x == 0) {
30         int prev = atomicInc(&retirementCount,
       gridDim.x);
31         amLast = (prev == (gridDim.x - 1));
32     }
33     __syncthreads();
34
35     if ((amLast) && (threadIdx.x == 0)) {
36         for (int i = 1; i < gridDim.x; i++)
37             res[0] += res[i];
38     }
39 }
```

Listing 1. Implementation of parallel sum using the `atomicAdd` instruction, and deterministic implementation using pairwise algorithm.

Option (iii) is deterministic, but slow, as only one GPU thread computes the reduction. A code snippet can be found in the GitHub repository associated with this paper. Option (i) is only available for AMD GPUs (with HIP) in an unsafe compiler mode and is not considered for AMD GPUs here. This `atomicAdd`-only (AO) method is shown in Listing 1 for the CUDA version. This approach is the easiest to program (only a few lines of code), but is actually sequential, and yet the instruction order is runtime dependent, making it non-deterministic. Option (ii) and (iv) use a pairwise algorithm [20], [23] coupled to data blocking for distributing the data over the different thread blocks and specifying the order for summation. Here, each thread of a given block adds the elements x_i in pairs $t_i = x_i + x_{i+n/2}$, $i = 1, \cdots, n/2$, done in parallel on GPU. This process is repeated $log_2(n)$ times on t_i. We use the `__syncthreads` instruction after each step of the partial reduction for thread synchronization within the thread block. Shared memory is also used to improve data locality and performance. Partial results are then stored back in global memory. We tested three different methods for accumulating these partial results. The simple-pass-with `atomicAdd` (SPA), uses an `atomicAdd` instruction to accumulate all partial sums instead of storing the results back in memory. This is again a simple solution from the programmatic perspective, but the implementation is not deterministic. The two-passes-with-final-reduction-on-CPU (TPRC) method uses the property that two kernels launched on the same stream will execute sequentially following the submission order, introducing a synchronization barrier between them, or between a kernel and a data transfer between GPU and CPU. We choose the data transfer, and compute the final sum on CPU using the sequential recursive method. TPRC is deterministic, but more sensitive to compiler optimizations because of vectorization. The last approach uses an integer counter to keep track of the completed thread blocks. The sum function from the CUB/HIPCUB library (CU) uses a similar technique. Each thread block increments this variable atomically before exiting. The last block incrementing the counter is responsible for the remaining reduction, and can be performed in two ways. The single-pass-with-tree-reduction (SPTR) method uses the same block reduction algorithm than for the first stage while the single-pass-with-final-recursive-sum-on-GPU (SPRG) variant (see listing 1) uses the recursive sum. SPTR and SPRG are both deterministic by construction. SPRG and SPTR are only possible if we use the `__threadfence` instruction to avoid data race conditions. This instruction ensures that all writes issued by the calling thread are finished before the calling thread runs the next instruction, and notifies all the other running threads that the memory operation is over. The instruction is not a global synchronization barrier. A summary of the main properties of each implementation is provided in Table 2 while codes snippets and full reference codes are provided in the GitHub repository [24].

B. Other approaches: Parallel sums with OpenMP

OpenMP, a directive-based API for shared memory and accelerator parallel programming, supports data reduction to perform parallel calculations that are portable across archi-

TABLE 2
DIFFERENT IMPLEMENTATIONS OF THE PARALLEL SUM IN CUDA. NON-DETERMINISTIC IMPLEMENTATIONS SHOWN IN RED.

Method	deterministic	# of kernels	synchronization methods
CU	Yes	-	`__threadfence`
SPTR	Yes	1	`__threadfence`
SPRG	Yes	1	`__threadfence`
TPRC	Yes	2	stream synchronization
SPA	No	1	`atomicAdd`
AO	No	1	`atomicAdd`

tectures. The OpenMP programming model uses reduction scoping clauses to specify the regions of code where the reduction takes place. These regions can include parallel fork-join, tasks, sections, SIMD, and scope constructs [25], which can be executed on a host or a device such as a GPU. The OpenMP specification does not specify the location and ordering in which the values are combined, thus, bitwise determinism is not guaranteed. As a result, implementing a deterministic parallel reduction in OpenMP will require using constructs that can enforce ordering when reducing private and shared variables. An ordered construct can define a structured block within a loop, SIMD, or loop SIMD region. This construct enforces sequential execution for a specified region of code according to the loop iterations, while enabling parallel execution for code outside the region. When the thread handling the first iteration of the loop reaches the ordered construct, any other thread that encounters the ordered construct in its iteration will wait at the start of the ordered region until all ordered regions from previous iterations have been executed. To order the entire loop, an ordered clause can be used.

```
#pragma omp parallel for ordered &
        reduction(+:sum)
    for (i = 0; i < size; i++) {
       #pragma omp ordered
        {
            sum += array[i];
        }
    }
```

Listing 2. An ordered parallel reduction implementation OpenMP

```
#pragma omp target parallel for simd ordered &
        reduction(+:sum)
    for (i = 0; i < size; i++) {
            sum += array[i];
        }
```

Listing 3. An ordered parallel reduction implementation OpenMP using target construct

Listing 2 shows the use of a block-ordered directive on a reduction. A similar effect can be achieved while using an ordered clause. Listing 3 shows how to use an ordered clause inside a target region that executes on a device or GPU. When comparing the results of listing 2 with a reduction without the ordered directive, using gcc 12.2.0 on CPU, we get the results shown in Table 3; the ordered version produces deterministic results. We achieve this determinism by taking advantage of OpenMP static scheduling and the ordered clause and directive. However, the ordered directive is not available

TABLE 3
RESULTS OF THE NORMAL AND ORDERED REDUCTIONS USING OPENMP ON CPU

Trial	Normal Reduction	Ordered Reduction
1	2.3542548638889723e-07	2.3542548638889725e-07
2	2.3542548638889731e-07	2.3542548638889725e-07
3	2.3542548638889725e-07	2.3542548638889725e-07
4	2.3542548638889723e-07	2.3542548638889725e-07
5	2.3542548638889717e-07	2.3542548638889725e-07
6	2.3542548638889725e-07	2.3542548638889725e-07
7	2.3542548638889728e-07	2.3542548638889725e-07
8	2.3542548638889728e-07	2.3542548638889725e-07
9	2.3542548638889731e-07	2.3542548638889725e-07
10	2.3542548638889728e-07	2.3542548638889725e-07

for the team distribute directive. This limitation is important for reductions that execute on a device, as the order of iterations between teams is not possible; a user-defined or lower-level reduction implementation as previously described will be needed.

C. Statistical properties of the variability of non-deterministic parallel sums using CUDA or HIP on different GPUs

It is often assumed that the variability due to FPNA can be described as Gaussian noise, but we could not find evidence supporting this in the literature. We therefore performed numerical experiments to estimate the probability density function (PDF) of $V_s(S)$. We generated a set of 100 arrays of one million FP64 elements each taken from the uniform distribution $U(0, 10)$, applied SPA 10000 times to each array, and evaluated the variability V_s, using SPTR for the deterministic summation, returning one million values for V_s. S_{d} thus refers here to SPTR while S_{nd} refers to SPA. We then repeated the experiment, replacing the uniform distribution with a normal distribution of zero mean and standard deviation 1. The resulting PDFs are shown in Fig.1 for both distributions

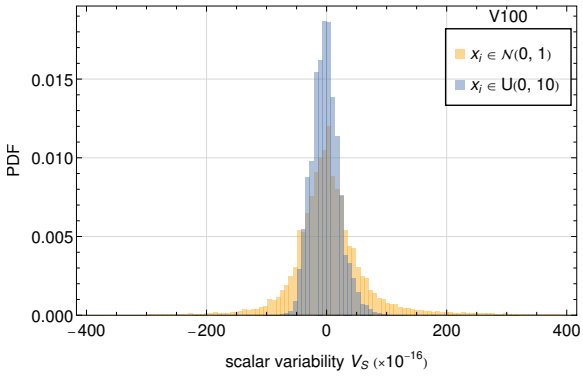

Fig. 1. Probability density of the variability V_s for sums of 1 M FP64 numbers sampled from normal and uniform distributions on the V100 GPU. Kernel parameters are $N_t = 64$ and $N_b = 7813$ for both SPTR and SPA.

for the V100; for the Mi250X and GH200 GPUs the results can be found in the GitHub repository [24]. The means and standard deviations of V_s are different between the GPU types, while the shapes are similar. Using Kullback–Leibler divergence criterion (KL) analysis, we find that all PDFs for

SPA do converge towards a normal distribution whose mean and standard deviation depend on x_i and the GPU family. However, when we repeated the experiments replacing SPA with AO on the NVIDIA GPUs, using a sample size of 500000 sums (results shown in Fig. 2 for the V100), the distribution is found not to be normal, showing that the assumption of Gaussian noise is invalid in general. The reasons for this distribution are unclear, as the NVIDIA runtime scheduler details are proprietary.

We also calculated the dependence Max $|V_s|$ as a function of the array size n for the same sequences used for Fig. 1. The data can be fit with a power law of the form βn^{α}. Max $|V_s|$ is proportional to $\sqrt{n}$ when $x_i \in U(0, 10)$. The exponent is larger for $x_i \in \mathcal{N}(0, 1)$, showing that the range of the numbers also plays a role.

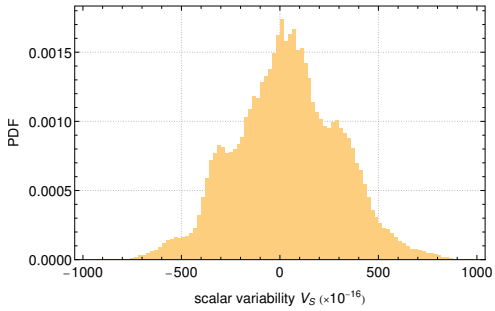

Fig. 2. PDF of the scalar variability V_s for 1 M FP64 numbers sampled from the uniform distribution $U(0, 10)$ when AO is used for the nondeterministic implementation, on V100.

D. Performance comparisons

To illustrate the performance impact of these different strategies, we measured the time needed to compute 100 sums of 4194304 FP64 elements taken from the uniform distribution, using SPTR, TPRC, CU, AO and SPA. The only nondeterministic algorithms in this tests are AO and SPA. We compute the negative quantity $P_s = 100(1 - t_i)/\min(t_i)$ where $\min(t_i)$ is the minimum of all timings t_i, to compare the performance of all implementations to the fastest one. The other parameters controlling the algorithm are the thread block size and the number of thread blocks. Details on the choice of these settings are provided in the associated repository [24].

The main results are summarized on Tab. 4. We find that AO is 2 orders of magnitude slower than the fastest implementations. The fastest sum implementation depends on the GPU version. The non-deterministic SPA seems to be favored on NVIDIA GPU, followed by SPTR and CU. The difference in timings between SPA and SPTR is less than 0.2% for the V100 GPU but can reach up to 7.8% on the GH200. CU suffers a 4.5% penalty on GH200. Most implementations on V100 are within 0.5% to 1% of each other. The performance penalty on GH200 is more spread than on V100. TPRC is the fastest implementation on the Mi250X GPU followed by CU. Our results show that the performance of a deterministic implementation can be faster or only slightly slower than its non-deterministic counterpart, depending on the GPU type. While timings can also change depending on the GPU load, these results suggest that there is no reason to calculate a parallel sum using nondeterministic `atomicAdd` operations, as the performance benefit is marginal at best.

IV. Non-determinism in PyTorch Functions

As discussed in the introduction, FPNA can lead to large variations in identical training and inference pipelines for deep learning, due to compounding effects within training, along with the impact of nonlinear activation functions. Analogously to the above section, we explore FPNA-induced variability for the PyTorch operations that are documented to have non-deterministic behavior [26]. PyTorch makes it easy for end users to construct neural networks out of high-level modules and deploy software on devices such as GPUs without the need for direct GPU programming. GPU kernels are provided by vendor libraries such as NVIDIA's cuDNN and AMD's MIOpen, and sometimes by PyTorch developers. As demonstrated in the previous section, any kernels PyTorch uses that rely on atomic operations will not be deterministic, leading to output variability. Furthermore, to be hardware agnostic and computationally efficient there are cases where strategies have been developed to determine the optimal computational kernel at runtime, also causing non-determinism.

Apart from the choice of kernel at runtime and atomic operations, there are several other ways in which PyTorch may induce variability. These include having an unset random number generator seed, having unset CUDA environment variables, uninitialized GPU memory and communication between devices. To focus exclusively on variability emergent from the first two sources, we a single GPU and remove those other sources of variability. We control whether or not non-deterministic kernels are available through the environment variable that PyTorch provides [26]. We note that this documentation may not necessarily be completely accurate across all systems and software versions, as we received a runtime error when trying to obtain a deterministic result for `scatter_reduce`, suggesting the potential difficulties high level users may experience when trying to control determinism within this deep software stack.

We performed experiments in two ways, depending on whether or not a deterministic kernel exists. If there is a deterministic option, then the output multidimensional array $\mathbf{A}$ is fixed by that implementation. We compare it with the tensors B_i with i labelling N runs of the non-deterministic implementation. If there is no non-deterministic kernel, we choose the first non-deterministic invocation as a reference, $\mathbf{A} = \mathbf{B}_0$. For each experiment we also present results for measurements of kernel runtime for non-deterministic implementations on the GPU, and deterministic implementations on the GPU and LPU architectures, to expose the performance costs of using deterministic operations. Runtime measurements are only for the execution time of the kernel, excluding data transfers to/from the device. For the GPU we make many measurements to report the average and standard deviation of the execution time. On the LPU architecture the runtime for

TABLE 4
TIMING AND PERFORMANCE PENALTY OF PARALLEL SUM IMPLEMENTATIONS ON DIFFERENT GPUS FOR 100 SUMS OF 4194304 FP64 NUMBERS AND VARYING KERNEL PARAMETERS. TIMINGS AVERAGED OVER 10 CONSECUTIVE RUNS; NON-DETERMINISTIC ALGORITHMS INDICATED IN RED. STANDARD DEVIATIONS IN PARENTHESES.

GPU	implementation	$(N_t \times N_b)$	time for 100 sums (in ms)	P_s (in %)
V100	SPA	(512×128)	6.456(0.008)	0
	SPTR	(512×128)	6.469(0.011)	−0.198538
	TPRC	(512×128)	6.491(0.006)	−0.528162
	CU	(unknown)	6.87697	−6.51331
	AO	(fixed parameters)	872.004(0.022)	−28781.3
GH200	SPA	(512×512)	3.019(0.022)	0
	CU	(unknown)	3.1553	−4.50533
	TPRC	(512×512)	3.226(0.030)	−6.87139
	SPTR	(512×512)	3.254(0.014)	−7.79916
	AO	(fixed parameters)	738.687(0.037)	−24365.7
Mi250X	TPRC	(512×256)	6.275(0.012)	0
	CU	(unknown)	6.378	−1.63577
	SPA	(512×256)	6.394(0.032)	−1.90101
	SPTR	(256×512)	6.552(0.023)	−4.4171

the PyTorch function is reported as a fixed number since the cycle-by-cycle execution is determined ahead of time [27].

We performed a sweep over operation hyperparameters using 10000 runs on an H100 and observed only a handful of functions specified in the PyTorch documentation [26] to exhibit non-determinism. We list these operations in Table 5 alongside minimum and maximum V_{ermv}. We refer the reader to our repository [24] for in-depth details on the full hyperparameter sweep, where, for example, we consider kernel size, stride and padding when testing convolution operations.

Our test results indicated that the main factor in ability to observe output variability in these kernels is the size of the input tensor, which gives more chances to observe the switching of order at runtime. For functions which perform reductions on an input tensor, we found that the ratio of the output dimension to input dimension is another key factor. We explore this further, restricting our analysis to `scatter_reduce` and `index_add`, which have such reductions.

TABLE 5
MAX AND MIN VARIABILITY FOR NON-DETERMINISTIC PYTORCH OPERATIONS OVER ALL HYPERPARAMETERS TESTED.

Operation	**min**$(V_{\text{ermv}})/10^{-7}$	**max**$(V_{\text{ermv}})/10^{-6}$
ConvTranspose1d	1.52	2.36
ConvTranspose2d	1.29	4.52
ConvTranspose3d	0	1.34
cumsum	0	0.52
index_add	0	5.03
index_copy	0.36	2.08
index_put	0	1.68
scatter	0	3.82
scatter_reduce	0	3.35

A. *Case studies of PyTorch kernels*

The `scatter_reduce` function updates an output array Y by applying a reduction operation on values from the input array X, according to indices specified in an index array I, for example, when reducing a one dimensional array with summation reduction, $Y[i] = \sum_j (\{X[j] \mid j = I[i]\})$. This can be generalized to arrays with arbitrary dimensions and different reduction functions. The `index_add` operation updates the output array Y by adding values from an input array X according to the indices specified in an index array I. As an example, for two dimensional input and output arrays with a summation reduction over the first dimension, the elements of Y are $Y[i,j] = Y[i,j] + X[k,j]$, for $i = I[k], \forall k$. This can similarly be generalized to arbitrary dimensions and different reductions other than addition. Both of these operations reduce the size of an input tensor, so we define the reduction ratio R as the ratio of the output dimension size to source dimension size. When $R = 1$, the arrays have the same size, and smaller values of R correspond to larger reductions of the source array down to at most a singular value, along the axis of reduction. For the index tensors of both operations, we use random integers drawn from a uniform distribution to choose arbitrary values from the source tensor. The motivation for this is to mimic an arbitrary graph structure, where the reduction is happening over nodes that share an edge, although in this case we do not *a priori* expect any specific structure or hierarchy to be emergent.

Fig. 3. Heatmaps of count variability (V_c) per run/iteration for 1,000 runs of the non-deterministic implementation of `scatter_reduce` (left) and `index_add` (right) for different reduction ratios and input dimensions. Note the input dimension for `index_add` is two dimensional square arrays, while the input dimension for `scatter_reduce` is one dimensional.

Fig. 3 shows heat maps of V_c as a function of R and in-

put dimension for the `scatter_reduce` and `index_add` operations with summation reductions. There is a clear trend of increased variability as a function of each of these parameters. In the case of larger input dimension, there are more opportunities for runtime non-determinism to occur. For small reduction ratios there is less variation. We suspect this is due to the fact that our index tensor is random, meaning that elements being reduced will not be located locally in memory. For both operations we observe V_c near one, meaning that most runs have a unique output. This is the worst case for reproducibility and error debugging.

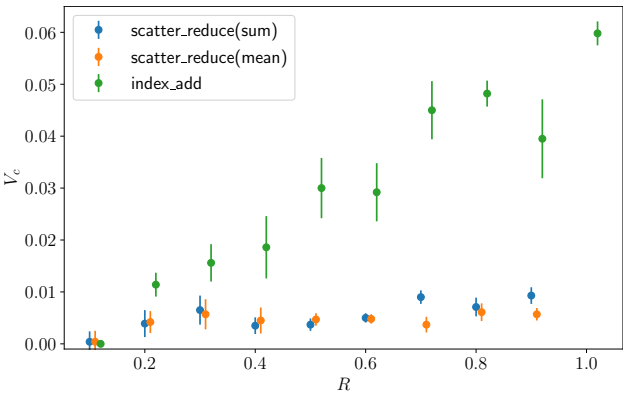

Fig. 4. Plot of the count variability for different reduction ratios for the scatter reduce and index add pytorch kernels. For the scatter reduce kernel we use an array of 2,000 elements, while for the index add we use an array of 100 elements.

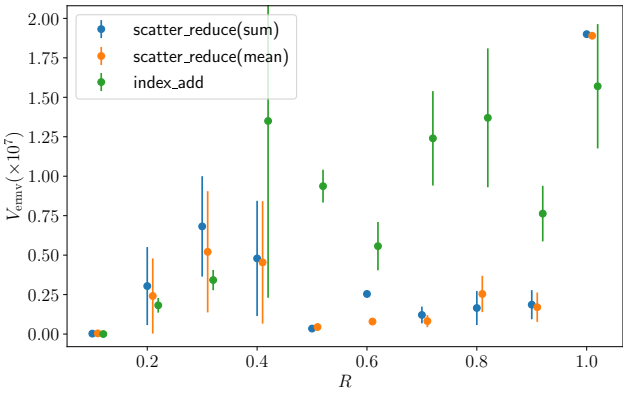

Fig. 5. Plot of the tensor variability for different reduction ratios for the scatter reduce and index add PyTorch kernels. For the scatter reduce kernel we use an array of 2,000 elements, while for the index add we use an array of 100 elements.

Figures 4 and 5 show V_{ermv} and V_c as a function of reduction ratio. For the `scatter_reduce` operation we use a one dimensional array with 2,000 elements, and for `index_add` we use a two dimensional square array with 100 $\times$ 100 elements. These array sizes are selected to be in an interesting regime of reduction ratio behavior as observed from the heatmaps. We notice a fairly constant V_c between 0.005 and 0.01 for `scatter_reduce` with both the `sum` and `mean` reductions. The V_c for this operation at a reduction ratio of 1.0 has a value around 0.10 (not plotted) which is a significant jump. We notice similar behavior for V_{ermv} for this operation. Note for the case where the output array has only one element, we recover the previous problem statement of computing the sum of an array, covered in Section III. For `index_add`, V_c is increasing almost linearly with reduction ratio. The errors are inconsistently sized across reduction ratios, indicating different behavior at each reduction ratio. We also notice an approximately linear trend between V_{ermv} and reduction ratio, with a particularly large error bar at $R = 0.4$. The lack of trend for the errors on the variability requires further analysis.

TABLE 6
AVERAGE KERNEL RUNTIME FOR `SCATTER_REDUCE` AND `INDEX_ADD` KERNELS ON THE H100 AND GROQ USING DETERMINISTIC (D) AND NON-DETERMINISTIC (ND) IMPLEMENTATIONS. STANDARD DEVIATIONS IN PARENTHESES.

Operation	Implementation	H100 (μs)	Groq (μs)
`scatter_reduce` (`sum`)	D	N/A	10.5
	ND	30.2(1.4)	N/A
`scatter_reduce` (`mean`)	D	N/A	28.9
	ND	74.9(1.4)	N/A
`index_add`	D	161(4)	12.0
	ND	12.8(26)	N/A

We now investigate performance effects of deterministic and non-deterministic operations, performed on the H100 and LPU architectures. Up until now, the LPU architecture has been left out of the discussion of variability because its hardware operates deterministically [27]. In Table 6 we report the average kernel runtime for the `scatter_reduce` with input dimension 1,000 and reduction ratio 0.5 and `index_add` with input dimension 1,000 $\times$ 1,000 and reduction ratio 0.5 on both the H100 and LPU architectures. For the latter we found the non-deterministic implementation on the H100 to be faster. This trend is not observed across all the operations in the documented list [26], however. For `scatter_reduce` and `index_add` the LPU architecture, which is by default deterministic, is faster than all GPU implementations. Runtimes for the complete set of functions in [26] are given in the supplemental repository [24].

V. EFFECT OF NON-DETERMINISM ON FULL DEEP LEARNING WORKFLOWS

Non-determinism in PyTorch kernels can adversely affect upstream tasks in model training and inference as found in real-world applications. To highlight this, we focus on training and inference for a Graph Sample and Aggregate (GraphSAGE) neural network.

A. GraphSAGE Convolution Network

Graph Neural Networks (GNNs) are extensively used for analyzing graph-structured data, such as social networks and molecular structures. GNNs operate on the principle of message passing, where each node in the graph aggregates information from its neighbors to update its

own representation. The aggregation of information in software is often implemented via scatter and gather operations, which we have shown above have severe runtime variability. A layer in a GNN is defined by $h_v^{(k)} = \mathrm{U}^{(k)}\left(h_v^{(k-1)},\ \mathrm{A}^{(k)}\left(\{h_u^{(k-1)} \mid u \in \mathcal{N}(v)\}\right)\right)$ where $h_v^{(k)}$ is the representation of node v at layer k, $\mathcal{N}(v)$ denotes the set of neighbors of node v, and U and A are functions defining the update and aggregation steps, respectively. GraphSAGE is a popular graph convolution that uses functions such as the sum and mean for the aggregation. Implementations often use the `scatter_reduce` and `index_add` operations discussed above, for example in the PyTorch Geometric [28] library.

B. Results

We trained a GNN with two SAGEConv layers on the Cora dataset. This dataset is widely used for GNN research and benchmarks; it consists of 2,708 scientific publications classified into one of seven classes. The graph structure is created with 5,429 links representing citations between these publications. Each publication is described by a 1,433-dimensional feature vector. Training of the SAGEConv model is performed on this data set with a 10 epoch training run. The only source of non-determinism in our implementation of this DNN is the `index_add` operation. We studied the variability of model weights over 10 epochs and found that mean V_{ermv} increased from from epoch 1 to 10, while the standard deviations also increased: 1.414(0.05) to 1.418(0.23). This may indicate that, as expected, non-determinism results in a compounding increase in the variability with more kernel calls. At the end of the training loop, all 1,000 models had a unique set of model weights: V_c 1. Despite this variability all models converge to similar loss values. Unlike stochastic training using random number generators for initialization, the resulting 1,000 models are completely non-reproducible, even for a single user on a single machine.

TABLE 7
V_{ERMV} AND V_c FOR DIFFERENT TRAINING AND INFERENCE DETERMINISTIC (D) AND NON-DETERMINISTIC (ND) COMBINATIONS ON GPUS. STANDARD DEVIATIONS IN PARENTHESES.

Training	Inference	$V_{\text{ermv}}/10^{-6}$	V_c
D	D	0(0)	0(0)
	ND	2.63(0.12)	0.12(0.2)
ND	D	4.27(0.21)	0.16(0.2)
	ND	5.08(0.23)	0.21(0.4)

We also measured the different combinations of training and inference stages cumulatively, by creating 1,000 models under four conditions. These are deterministic training and deterministic inference, deterministic training and non-deterministic inference, non-deterministic training and deterministic inference, and lastly non-deterministic training and non-deterministic inference. The V_{ermv} and V_c for each of these experiments is presented in Table 7. As expected, the non-deterministic training with non-deterministic inference has the most severe variations, and while training seems to incur more variability, inference variability contributes a non-negligible amount. The runtime for the GraphSAGE model training for the full 10 epochs is $0.48(1)$ seconds with a deterministic operation and $0.18(1)$ seconds with a non-deterministic one. The inference runtimes for a single input are given in Table 8; the deterministic implementation is slower than the non-deterministic one on GPU. Inference on the LPU accelerator is 30 times faster than the fastest PyTorch implementation, consistent with results presented in [29].

TABLE 8
H100 AND GROQ KERNEL RUNTIME FOR DETERMINISTIC AND NON DETERMINISTIC INFERENCE OF THE GRAPHSAGE MODEL.

Inference	H100 (ms)	Groq (ms)
Deterministic	3.92(0.2)	0.066
Non Deterministic	2.17(0.1)	N/A

VI. CONCLUSIONS

The analyses presented here highlight variability from non-deterministic kernels for basic parallel reductions, PyTorch functions, and resulting adverse effects on a full DL training and inference pipeline. The variability from non-deterministic reductions can approach the tolerance thresholds used in high-accuracy molecular simulation correctness tests, and for DL, will produce a unique and non-reproducible set of model weights with each run using identical inputs. These effects were pronounced for `scatter_reduce` and `index_add` operations, which are used extensively in GNNs. GNNs have become a key tool for DL models used in molecular simulation. In terms of productivity and performance, using deterministic programming patterns may require more effort or experience, especially to maintain performance, and may not be attainable for all algorithms. Within large DL framework stacks, providing deterministic solutions is left to the developers. We found that documentation on which functions in PyTorch are deterministic may be erroneous or incomplete, highlighting the large burden placed on end users in development of reproducible DL workflows. The benefits of deterministic chip designs, therefore, may extend beyond performance improvements, to reproducibility facilitation and support, increasing productivity and correctness. The non-determinism studied here results from asynchronous atomic operations on GPUs, but in HPC and distributed settings there will also be inter-chip and inter-node communication, such as with MPI, leading to more runtime variation. On the LPU architecture, inter-chip communication can be software scheduled, removing such communication variations [30]. Future work should explore effects such as this in addition to the ones presented here. In addition, to mitigate the non-reproducibility produced by the training portions of a DL workflow, deterministic hardware for training could also be proposed; emerging DL chip designs may support such solution and can be evaluated.

VII. ACKNOWLEDGEMENTS

This work was supported in part by the ORNL AI LDRD Initiative and in part by Swiss Platform For Advanced Scientific Computing (PASC), and used resources of the OLCF, a DOE Office of Science User Facility [DE-AC05-00OR22725], and Swiss National Supercomputing Centre (CSCS).

REFERENCES

[1] C. Lin *et al.*, *Principles of Parallel Programming*. Pearson Education India, 2008.

[2] W. Ahrens, J. Demmel, and H. D. Nguyen, "Algorithms for efficient reproducible floating point summation," *ACM Trans. Math. Softw.*, vol. 46, p. 22, 2020.

[3] O. Villa, D. Chavarria-Miranda, V. Gurumoorthi, A. Márquez, and S. Krishnamoorthy, "Effects of floating-point non-associativity on numerical computations on massively multithreaded systems," in *Proceedings of Cray User Group Meeting (CUG)*, vol. 3, 2009.

[4] M. Thavappiragasam, W. Elwasif, and A. Sedova, "Portability for GPU-accelerated molecular docking applications for cloud and HPC: can portable compiler directives provide performance across all platforms?" in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2022, pp. 975–984.

[5] T. D. Kühne, M. Iannuzzi, M. Del Ben, V. V. Rybkin, P. Seewald, F. Stein, T. Laino, R. Z. Khaliullin, O. Schütt, F. Schiffmann, D. Golze, J. Wilhelm, S. Chulkov, M. H. Bani-Hashemian, V. Weber, U. Borštnik, M. Taillefumier, A. S. Jakobovits, A. Lazzaro, H. Pabst, T. Müller, R. Schade, M. Guidon, S. Andermatt, N. Holmberg, G. K. Schenter, A. Hehn, A. Bussy, F. Belleflamme, G. Tabacchi, A. Glöß, M. Lass, I. Bethune, C. J. Mundy, C. Plessl, M. Watkins, J. VandeVondele, M. Krack, and J. Hutter, "CP2K: An electronic structure and molecular dynamics software package - Quickstep: Efficient and accurate electronic structure calculations," *The Journal of Chemical Physics*, vol. 152, no. 19, p. 194103, 05 2020. [Online]. Available: https://doi.org/10.1063/5.0007045

[6] R. Salomon-Ferrer, D. A. Case, and R. C. Walker, "An overview of the Amber biomolecular simulation package," *Wiley Interdisciplinary Reviews: Computational Molecular Science*, vol. 3, no. 2, pp. 198–210, 2013.

[7] S. LeGrand, A. Scheinberg, A. F. Tillack, M. Thavappiragasam, J. V. Vermaas, R. Agarwal, J. Larkin, D. Poole, D. Santos-Martins, L. Solis-Vasquez *et al.*, "GPU-accelerated drug discovery with docking on the summit supercomputer: Porting, optimization, and application to COVID-19 research," in *Proceedings of the 11th ACM international conference on bioinformatics, computational biology and health informatics*, 2020, pp. 1–10.

[8] D. Riach, "Framework reproducibility: Determinism (d9m)," https://github.com/NVIDIA/framework-reproducibility/blob/master/doc/d9m/README.md, accessed: 2024-8-07.

[9] P. Nagarajan, G. Warnell, and P. Stone, "The impact of nondeterminism on reproducibility in deep reinforcement learning," in *2nd Reproducibility in Machine Learning Workshop at ICML 2018*, 2018, pp. 1–10.

[10] C. Summers and M. J. Dinneen, "Nondeterminism and instability in neural network optimization," in *International Conference on Machine Learning*. PMLR, 2021, pp. 9913–9922.

[11] L. Heumos, P. Ehmele, L. Kuhn Cuellar, K. Menden, E. Miller, S. Lemke, G. Gabernet, and S. Nahnsen, "mlf-core: a framework for deterministic machine learning," *Bioinformatics*, vol. 39, no. 4, p. btad164, 2023.

[12] J. Chen, Y. Liang, Q. Shen, J. Jiang, and S. Li, "Toward understanding deep learning framework bugs," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 6, pp. 1–31, 2023.

[13] L. Jia, H. Zhong, X. Wang, L. Huang, and X. Lu, "An empirical study on bugs inside tensorflow," in *Database Systems for Advanced Applications*, Y. Nah, B. Cui, S.-W. Lee, J. X. Yu, Y.-S. Moon, and S. E. Whang, Eds. Cham: Springer International Publishing, 2020, pp. 604–620.

[14] F. Tambon, A. Nikanjam, L. An, F. Khomh, and G. Antoniol, "Silent bugs in deep learning frameworks: an empirical study of keras and tensorflow," *Empirical Software Engineering*, vol. 29, no. 1, p. 10, 2024.

[15] S. Partee, M. Ellis, A. Rigazzi, A. E. Shao, S. Bachman, G. Marques, and B. Robbins, "Using machine learning at scale in numerical simulations with SmartSim: An application to ocean climate modeling," *Journal of Computational Science*, vol. 62, p. 101707, 2022.

[16] M. Boyer, W. Brewer, D. Jude, and I. Dettwiller, "Scalable integration of computational physics simulations with machine learning," in *2022 IEEE/ACM International Workshop on Artificial Intelligence and Machine Learning for Scientific Applications (AI4S)*. IEEE, 2022, pp. 44–49.

[17] H. Wang, L. Zhang, J. Han, and E. Weinan, "DeePMD-kit: A deep learning package for many-body potential energy representation and molecular dynamics," *Computer Physics Communications*, vol. 228, pp. 178–184, 2018.

[18] W. Jia, H. Wang, M. Chen, D. Lu, L. Lin, R. Car, E. Weinan, and L. Zhang, "Pushing the limit of molecular dynamics with ab initio accuracy to 100 million atoms with machine learning," in *SC20: International conference for high performance computing, networking, storage and analysis*. IEEE, 2020, pp. 1–14.

[19] A. Sedova, G. Sivaraman, M. Coletti, W. Elwasif, M. Smith, and O. Hernandez, "Avoiding a reproducibility crisis in deep learning for surrogate potentials: How massively parallel programming, millions of training steps, and numerics combine to create non-determinism in models and what this means for the simulated physics," in *APS March Meeting 2024*, ser. Bulletin of the American Physical Society, vol. March 4–8, 2024; Minneapolis & Virtual. American Physical Society, 2024.

[20] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*. SIAM, 2002.

[21] S. Batzner, A. Musaelian, L. Sun, M. Geiger, J. P. Mailoa, M. Kornbluth, N. Molinari, T. E. Smidt, and B. Kozinsky, "E (3)-equivariant graph neural networks for data-efficient and accurate interatomic potentials," *Nature communications*, vol. 13, no. 1, p. 2453, 2022.

[22] D. P. Kovács, I. Batatia, E. S. Arany, and G. Csányi, "Evaluation of the mace force field architecture: From medicinal chemistry to materials science," *The Journal of Chemical Physics*, vol. 159, no. 4, 2023.

[23] P. Blanchard, N. J. Higham, and T. Mary, "A class of fast and accurate summation algorithms," *SIAM Journal on Scientific Computing*, vol. 42, pp. A1541–A1557, 2020.

[24] S. Shanmugavelu, M. Taillefumier, C. Culver, O. Hernandez, M. Coletti, and A. Sedova, "Correctness," https://github.com/minnervva/correctness, 2024.

[25] OpenMP Architecture Review Board, *OpenMP Application Programming Interface Version 5.2*, OpenMP Architecture Review Board, Nov. 2021. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

[26] PyTorch, "PyTorch documentation for determinism," https://pytorch.org/docs/stable/generated/torch.use_deterministic_algorithms.html#torch.use_deterministic_algorithms, accessed: 2023-5-08.

[27] D. Abts, J. Ross, J. Sparling, M. Wong-VanHaren, M. Baker, T. Hawkins, A. Bell, J. Thompson, T. Kahsai, G. Kimmell, J. Hwang, R. Leslie-Hurd, M. Bye, E. Creswick, M. Boyd, M. Venigalla, E. Laforge, J. Purdy, P. Kamath, D. Maheshwari, M. Beidler, G. Rosseel, O. Ahmad, G. Gagarin, R. Czekalski, A. Rane, S. Parmar, J. Werner, J. Sproch, A. Macias, and B. Kurtz, "Think fast: A tensor streaming processor (tsp) for accelerating deep learning workloads," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 145–158.

[28] PyTorch, "PyTorch Geometric documentation," https://pytorch-geometric.readthedocs.io/en/latest/, accessed: 2023-5-08.

[29] R. Hosseini *et al.*, "Exploring the use of dataflow architectures for graph neural network workloads," in *High Performance Computing. ISC High Performance 2023*, ser. Lecture Notes in Computer Science, A. Bienz, M. Weiland, M. Baboulin, and C. Kruse, Eds., vol. 13999. Springer, Cham, 2023.

[30] D. Abts, G. Kimmell, A. Ling, J. Kim, M. Boyd, A. Bitar, S. Parmar, I. Ahmed, R. DiCecco, D. Han, J. Thompson, M. Bye, J. Hwang, J. Fowers, P. Lillian, A. Murthy, E. Mehtabuddin, C. Tekur, T. Sohmers, K. Kang, S. Maresh, and J. Ross, "A software-defined tensor streaming multiprocessor for large-scale machine learning," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 567–580. [Online]. Available: https://doi.org/10.1145/3470496.3527405

ARTIFACT DESCRIPTION

All source code and inputs for all tests and programs reported in this paper can be found in our devoted GitHub repository at https://github.com/minnervva/correctness. The main directories `codes/test_reduce` and `codes/test-suite` contain the codes used in Section III and IV of the main text. Repository organization, installation, usage, and contributing instructions are documented in the README.

A. Experimental Methods

In the following we describe the hardware and systems software used to perform our numerical analyses and the mathematical definitions developed to quantify output variability caused by non-determinism due to FPNA in parallel kernels, studied as isolated test cases and within the PyTorch deep learning framework high-level operations.

Tests on GH200 are run on the Alps supercomputing system located at CSCS running SLE 15 (enterprise). Each node has 4 GH200 with 72 ARM cores, 128 GB LPDDR5, and one H100 GPU with 96 GB HBM3 memory each.

Tests for the V100 are run on the Summit supercomputer at the Oak Ridge Leadership Computing Facility (OLCF), running Redhat OS 8. Summit is an IBM system; each IBM Power System AC922 node has two Power9 CPUs with 512 GB of memory and 6 V100 NVIDIA GPU with 16GB of HBM2 memory.

Tests on Mi250X AMD GPU are obtained on the Frontier supercomputer at OLCF, running SLE 15 (enterprise). Frontier is an HPE Cray EX supercomputer; each Frontier compute node has a 64-core AMD "Optimized 3rd Gen EPYC" CPU with 512 GB of DDR4 memory and 4 AMD MI250X GPUs, each with 2 Graphics Compute Dies (GCDs) for a total of 8 GCDs per node.

Tests on the NVIDIA H100 are performed on a Groq host node3 in Groq ZankerLab running Ubuntu LTS 22.04.06. This machine has 2 H100's with 40GB HBM3 memory each and has an AMD EPYC 7302 16-Core Processor host CPU with 2 sockets, 16 cores per socket and 2 threads per core. Tests on the LPU accelerator are run on the GroqNode™ server r01-gn-01 in Groq ZankerLab with GroqWare™ SDK 0.11 and Ubuntu LTS 22.04. A GroqNode has 8 Groq LPUs connected with a fully connected 88 RealScale™ chip-to-chip connectors and 2 AMD EPYC 7313 processors (3GHz, 16C/32T, 155W TDP each).