Arbitor: A Numerically Accurate Hardware Emulation Tool for DNN Accelerators

Chenhao Jiang and Anand Jayarajan, *University of Toronto and Vector Institute;* Hao Lu, *University of Toronto;* Gennady Pekhimenko, *University of Toronto and Vector Institute*

https://www.usenix.org/conference/atc23/presentation/jiang-chenhao

This paper is included in the Proceedings of the 2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Arbitor: A Numerically Accurate Hardware Emulation Tool for DNN Accelerators

Chenhao Jiang
University of Toronto
Vector Institute

Anand Jayarajan
University of Toronto
Vector Institute

Hao Lu
University of Toronto

Gennady Pekhimenko
University of Toronto
Vector Institute

Abstract

Recently there has been considerable attention on designing and developing hardware accelerators for deep neural network (DNN) training workloads. However, designing DNN accelerators is often challenging as many commonly used hardware optimization strategies can potentially impact the final accuracy of the models. In this work, we propose a hardware emulation tool called Arbitor for empirically evaluating DNN accelerator designs and accurately estimating their effects on DNN accuracy. Arbitor takes advantage of modern machine learning compilers to enable fast prototyping and numerically accurate emulation of common DNN optimizations like low-precision arithmetic, approximate computing, and sparsity-aware processing on general-purpose GPUs. Subsequently, we use Arbitor to conduct an extensive sensitivity study to understand the effects of these optimizations on popular models such as ResNet, Transformers, Recurrent-CNN, and GNNs. Based on our analysis, we observe that DNN models can tolerate arithmetic operations with much lower precision than the commonly used numerical formats support. We also demonstrate that piece-wise approximation is effective in handling complex non-linear operations in DNN models without affecting their accuracy. Finally, enforcing a high degree of structured sparsity in the parameters and gradients can significantly affect the accuracy of the models.

1 Introduction

Deep neural networks (DNNs) have shown unprecedented accuracy on many complex tasks like computer vision [26,44], natural language processing [15,65], recommendation systems [50], speech recognition [6], and robotics [55]. This superior performance of DNNs, however, comes at the expense of high computational costs in the training process. As the models are getting larger and more complex, the computational requirement for training these models has also been growing steadily. Therefore, designing and developing specialized hardware accelerators for DNN training has become an active area of research in both industry and academia [16].

Modern DNN accelerators are equipped with thousands of parallel processing units to leverage the abundant computational parallelism available in DNN training workloads. Moreover, they also employ many hardware-level optimizations that are designed to take advantage of the error-resilient nature of the DNN models. For example, DNN accelerators commonly use *low-precision numerical formats* for arithmetic operations to improve hardware utilization and energy efficiency [18,48,74]. Additionally, *approximate computing* techniques like the linear approximation of non-linear functions are widely used to reduce the hardware design complexity and cost [7,43]. Finally, modern accelerators support *sparsity-aware processing* cores to minimize redundant computations in DNN workloads by skipping arithmetic operations over zeros. In recent years, there have been many studies proposing different variations of such hardware-level optimizations that have shown to be highly effective in improving the performance of DNN training workloads [56,72].

Despite being a well-studied area, making the right design decisions for DNN accelerators is still a challenging task as many of the proposed hardware optimizations have non-trivial effects on the convergence accuracy of the DNN training algorithm and can potentially hurt the final accuracy of the model [18,48]. Understanding the influence of these optimizations on the convergence of different DNN models requires rigorous experimental analysis on the DNN accelerator design. Unfortunately, currently available methods and tools are inadequate for such analysis as we explain below.

Software-based architectural simulators are widely used for the initial prototyping and analysis of different hardware design choices [8,9,34,64]. These tools are primarily designed to perform hardware simulations at circuit-level precision and measure accurate low-level performance counters like instructions per cycle (IPC), cache miss rate, and branch prediction accuracy. As a result, architectural simulators are generally $6-7$ orders of magnitude slower compared to the real silicon performance [20,63]. Since standard DNN benchmark models [49,75] take anywhere from hours to months of training to reach peak accuracy, using software-based simulators to analyze DNN accelerator designs can get prohibitively time-consuming. Even though other approaches like FPGA-based prototyping can offer performance close to the real hardware, programming FPGAs requires unique expertise that is rare even among hardware researchers. Moreover, building the software stack that can provide runtime support for training DNNs on FPGAs requires substantial engineering effort.

To address the limitations of these traditional methods, recent works [45, 47, 59, 70] have proposed hardware emulation frameworks specifically designed for analyzing the accuracy effects of common DNN optimizations like low-precision training. These tools provide convenient APIs to configure and emulate low-precision arithmetic in the training process by modifying the computation graphs of DNN models. Unlike architectural simulators, DNN emulators are built as extensions over popular machine learning frameworks like TensorFlow [3] or PyTorch [54] and can be executed over general-purpose accelerators like GPUs. This enables fast prototyping of arbitrary low-precision numerical formats and evaluating their effects on standard DNN benchmark models within reasonable time frames. However, we observe that extending these tools to support other hardware features like approximate or sparsity-aware computing require significant effort from the user. On top of this, current DNN emulators are designed to perform low-precision arithmetic emulation at the DNN layer-level granularity. Under this strategy, the computation of individual layers in the DNNs like matrix multiplication and convolution is performed using standard 32-bit floating point format, and the output is rounded down to the user-defined numerical format. We observe that this *coarse-grained emulation* approach can produce numerically inaccurate results compared to the real hardware. Our empirical evaluation reveals that the relative numerical error of low-precision arithmetic emulation in state-of-the-art DNN emulators can be as high as 15% (more details in section 3). We argue that such inaccurate emulations could lead to incorrect evaluation of hardware designs in DNN accelerators.

In this work, we build an easy-to-use, extensible, and more importantly numerically accurate emulation tool called Arbitor for empirically evaluating DNN accelerator designs. Arbitor is built on top of TensorFlow and provides extensions to its Keras front-end APIs for users to easily configure and emulate common hardware features like low-precision training, approximate computing, and sparsity-aware processing on standard DNN models. In contrast to other DNN emulators, Arbitor emulates the user-defined hardware features at the granularity of the primitive mathematical operations in the DNN layers to accurately mimic the behavior of real hardware. We support this *fine-grained emulation* in Arbitor with the help of the XLA compiler-backend in TensorFlow [23]. Modern machine learning compilers like XLA use domain-specific intermediate representations (IR) for defining the operations in the DNN computation graphs. These IR definitions of DNN operations are designed to be hardware-independent and are progressively lowered to GPU executable kernels by the compiler. We make a key observation that it is possible to automatically generate GPU kernels that emulate the user-defined hardware features by modifying the code generation pipeline of TensorFlow XLA. We empirically show that, compared to the state-of-the-art layer-level emulation approach, the operator-level emulation strategy of Arbitor can perform arithmetic operations and generate results that accurately match the results from real hardware.

We subsequently use Arbitor to conduct a series of sensitivity studies on the effects of low-precision training, approximate, and sparse computing on popular DNN architectures like Transformer [65], ResNet-18 [26], Convolutional Recurrent Neural Network (CRNN) [66], and Graph Neural Network (GNN) [73]. First, we conduct an extensive analysis of various non-standard floating point specifications and find that DNN models can maintain their accuracy with much lower precision than many standard floating point formats supported in current DNN accelerators. We also observe that the numerical precision can be reduced even further by combing low-precision formats with the standard single-precision floating point. Second, we also analyze the effectiveness of newly proposed non-floating point numerical formats like Posit [25] on DNN training. We find that, contrary to the prior observations [59], Posit does not yield better model accuracy compared to floating point. Third, we evaluate a popular approximate computing optimization technique called piece-wise linear approximation [35] and find that natural language processing (NLP) models like CRNN can maintain their baseline accuracy even with aggressive approximations. Finally, we analyze the effect of sparse computing and observe that enforcing more than 50% sparsity on DNN training can significantly affect the model accuracy. To the best of our knowledge, we are the first open study to conduct such an extensive analysis of common DNN accelerator designs using numerically accurate methods and tools.

In summary, we make the following contributions:

- We highlight that the hardware research community is currently lacking a fast and accurate hardware analysis tool for DNN training accelerators as state-of-the-art tools are either prohibitively slow or are susceptible to numerical inaccuracy.
- We build a hardware emulation tool Arbitor using a compiler-assisted fine-grained emulation strategy for numerically accurate emulation of optimizations like low-precision, approximate, and sparse computation.
- Using Arbitor, we conduct the first in-depth empirical analysis on the effects of low-precision arithmetic, approximate computing, and sparsity-aware processing on the accuracy of popular DNN models like Transformer, ResNet-18, CRNN, and GNN. We will be open-sourcing Arbitor soon for supporting empirical research in DNN accelerators.

2 Hardware Accelerators for DNN Training

Modern DNN models [15, 17, 26, 44] contain millions to even trillions of parameters and are trained for hours to months by iteratively processing large batches of data from humongous datasets and updating the model parameters to minimize the prediction error until the model converges to the desired accuracy. The DNN training process primarily consists of the repeated execution of computationally expensive algebraic functions such as matrix multiplication, convolution, and element-wise operations. Fortunately, these functions exhibit abundant computational parallelism which can be leveraged to speed up the training process using parallel processing hardware accelerators like graphics processing units (GPUs).

While GPUs have been the mainstay for DNN training, in recent years, there has been an increasing interest in developing more specialized accelerators. For example, Google TPU [31], Habana Gaudi [30], Graphcore IPU [24], Cerebras Wafer-Scale engine [14], and Amazon Trainium [5] are a few notable examples of commercial DNN accelerators. In addition to massively parallel processing capabilities, these accelerators also employ several hardware-level optimizations that exploit the inherent error-resilient nature of DNNs to improve training performance and hardware utilization. Below, we describe the three most common categories of hardware-level optimizations supported in DNN accelerators.

1. Low-precision arithmetic. Unlike regular parallel processing applications that require high-precision floating point computation, DNN models are highly tolerant towards using reduced-precision arithmetic due to the error-correcting nature of the training algorithm. Therefore, many DNN accelerators support low-precision numerical formats that use fewer bits than the standard 32-bit single-precision floating point (FP32). This enables accelerators to provide higher processing power from the same hardware budget. For example, Nvidia GPUs can perform 2× higher floating point operations (FLOPS) with half-precision (FP16) than using single-precision format [52]. Additionally, low-precision data formats can also help reduce the memory footprint of DNNs and in turn, improve the cache utilization and lower the memory bandwidth pressure during training. As a result, many modern DNN accelerators support a wide range of low-precision floating point formats outside the traditional IEEE-754 standard [2] as shown in Figure 1.

In general, floating points are represented using a sign bit, exponent bits, and mantissa bits. The range and the precision of a particular format are determined by the number of exponent and mantissa bits respectively. Therefore, different low-precision formats make the fundamental trade-off between the range and the precision of values the type can represent. For example, FP16 uses 5 exponent and 10 mantissa bits and has a limited range (i.e., $\pm 65,504$) compared to FP32 (i.e., $\pm 3.40 \times 10^{38}$). An alternative 16-bit format called brain float 16 (BF16) [32], on the other hand, uses 8 exponent bits and 7 mantissa bits and can support a similar range of values as that of FP32. Since BF16 format trades precision in favor of a wider range, it has shown to be better suited for DNN training than the standard FP16 and is widely supported in many DNN accelerators [32].

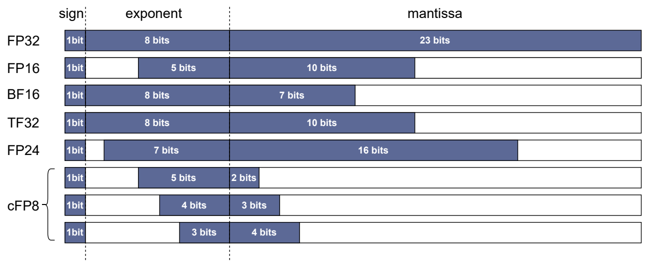


Figure 1: Common floating point formats in DNN accelerators

In addition to floating point formats, hardware researchers have also been exploring other data formats e.g., fixed-point arithmetic [41, 71, 74]. In contrast to floating points, fixed-point arithmetic lends itself to a simpler hardware design with a smaller chip area and lower power consumption [27] but lacks the dynamic range and precision that floating point formats have. In recent years, there has also been proposals [13, 46] in using the novel Posit formats [25] for DNN training as it takes less circuitry than floating point processing units while providing a wider dynamic range. Despite this significant research attention, finding numerical formats that make the right trade-offs in DNN training workloads is still an open research problem.

2. Approximate computation. The training computation of DNN models is mostly dominated by linear algebraic functions like matrix multiplication and convolution. These functions are composed of numerous primitive mathematical operations like addition and multiplication that maps very well with the thousands of parallel arithmetic units provided by the DNN accelerators. In addition to these, many DNN models also contain operations that use non-linear functions. For example, modern NLP and image classification models use complex activation functions like tanh, sigmoid, GeLU [28], and Swish [58] which are shown to play an important role in achieving high accuracy for the models. However, the exact computation of these functions is often very expensive to perform in the hardware because of the exponentiation and division terms present in the functions. Instead, DNN accelerators employ approximations of such functions that are cheaper to implement in hardware. Examples of such approaches include piece-wise linear/non-linear approximations [35, 61], lookup table [37], bit-level mapping [69], or a hybrid of these methods. Such approximations in DNN accelerators enable them to achieve improved hardware performance and lower chip area at the expense of imprecise computation of the non-linear functions. Therefore, more aggressive approximations can potentially affect the convergence of DNN training.

3. Sparsity-aware computation. Large DNN models are known to exhibit a significant amount of redundancy due to the over-parameterization of the model architecture [22]. There have been several efforts in exploiting this redundancy to improve performance and reduce the memory footprint of DNN workloads. For example, software-level optimizations such as DNN pruning [10] has found effective in significantly reducing the model size by removing the redundant parameters from the model. However, naiv̈ely applying DNN pruning often ends up with models having randomly distributed sparsity patterns. Since DNN accelerators are primarily designed to process dense data structures, extracting performance benefits from DNNs with unstructured sparsity is often challenging. To address this issue, modern DNN accelerators are equipped with specialized processing cores that can improve the performance of DNN applications in the presence of semi-structured sparsity patterns. For example, the latest Nvidia A100 GPUs support Sparse Tensor Cores that can accelerate matrix multiplication operations by $2\times$ for operands having 2:4 structured sparsity.[1] Recent works [56, 72] have proposed hardware-software co-optimizations that follow structured DNN pruning strategies during the training process to take advantage of such sparsity-aware processing units. Despite the potential performance improvements, enforcing structured sparsity during training is shown to have a significant impact on the convergence of the DNNs [68]. Finding the right balance between structural sparsity and model accuracy is currently an active area of research.

3 Need for an Accurate Hardware Analysis Tool for DNN Accelerator Design

Many of the aforementioned optimization strategies have the potential to significantly improve the performance of DNN training. However, at the same time, depending on the degree and aggressiveness of the optimizations, they can also negatively affect the model accuracy. Therefore, designing accelerators for DNN training requires taking both hardware performance and accuracy effects of the optimizations into consideration. Even though there are standard methods and tools available to prototype different chip designs and estimate their performance, they fail to be a good fit for analyzing their effects on model accuracy as we explain below.

Limitations of Traditional Verification Tools. One of the most common and cost-effective approaches for prototyping and verifying hardware designs is to use architectural simulators [4, 29]. Hardware researchers use simulators like GPGPU-sim [8], gem5 [9], and Multi2Sim [64] to evaluate architectural design choices by running software-based simulations of the proposed features and collecting hardware performance counters like instructions per cycle (IPC), cache utilization, and energy consumption. These tools are designed to run simulations with circuit-level precision, but they come at the cost of high execution time. Our empirical evaluation shows that running a single 1024×1024 FP32 matrix multiplication using GPGPU-sim is more than 5 orders of magnitude slower than running on bare-metal GPUs. Since training involves iteratively executing many such operations and can take anywhere from hours to months to finish even on powerful hardware [49, 75], it becomes prohibitively time-consuming to use these simulators to analyze the convergence effects of hardware optimizations on modern DNN models.

FPGA-based prototyping is another approach followed in the industry for hardware design verification. Programmable chips like FPGAs allow accurate verification of the functional logic of a design and the ability to run benchmark applications on custom-designed chips at a speed closer to the performance of the physical hardware. But programming FPGAs to reliably implement the desired hardware features is a labor-intensive task and requires special expertise and infrastructure support that is often beyond the reach of many independent researchers [60]. On top of this, building a full-fledged software stack that can provide the requisite runtime support for training DNN models is a major engineering undertaking that requires significant time and financial investment.

Since analyzing the statistical effects of DNN accelerator optimizations using traditional hardware verification tools is difficult and time-consuming in practice, hardware researchers often end up making design decisions using limited empirical analysis of the design or based on speculations. As hardware manufacturing is an extremely lengthy and expensive process, this could potentially cost a substantial amount of time, money, and resources. To take a real-world example, Nvidia first introduced half-precision floating point (FP16) support in Tesla P100 GPUs [52] in 2016 to deliver higher performance for deep learning workloads. However, it was soon observed that using half-precision in training can cause serious convergence issues on many models as algebraic functions like matrix multiplication, convolution, and batch normalization perform reduction over large dimensions of matrices and can suffer from higher accumulation error compared to single-precision (FP32) training. To correct this issue, it took researchers another two years to find a software-level fix called mixed-precision training [48] that uses a combination of FP16 and FP32 precision (for computation and reduction respectively) to curtail the numerical errors. More recently, Nvidia introduced a 19-bit floating point format called Tensor Float (TF32) in their Ampere architecture-based GPUs [51] as a drop-in replacement for FP32 data type. This again caused major pushback from the machine learning community for the numerical instability it caused on certain non-standard deep learning workloads [57]. These anecdotes suggest the importance of having rigorous methods and tools for analyzing the accuracy effects of DNN accelerator designs as early in the hardware manufacturing process as possible.

[1] Here, an N:M sparsity indicates that out of every block of M contiguous values in the input operands, only N values are non-zero.

Inadequacy of Current Emulation Tools. To meet this unique requirement of DNN accelerator research and development, recent works have proposed hardware emulation tools such as TensorQuant [45], QPyTorch [70], and GoldenEye [47]. Unlike architectural simulators that perform expensive cycle-accurate simulations of the entire hardware, these tools are specifically designed for analyzing DNN optimizations like low-precision training by emulating arbitrary numerical formats on general-purpose accelerators like GPUs. State-of-the-art DNN emulators are built on top of popular machine learning (ML) frameworks like TensorFlow [3] and PyTorch [54], and provide extensions to their front-end APIs to configure and train DNN models using user-defined numerical formats. These tools take advantage of the fact that the ML frameworks represent DNN models as layers of well-defined algebraic functions, and therefore, the numerical errors introduced by low-precision arithmetic in DNN training can be emulated by replacing the individual layers with corresponding software-emulated functions. Since these tools are well-integrated with ML frameworks and allow fast experimentation through GPU-accelerated emulation, they make a convenient tool for quick exploration of the low-precision data formats on a wide range of DNN models.

Despite being a promising approach, current DNN emulation tools suffer from two major limitations. First, they are primarily designed to target only a *narrow scope* of DNN optimizations, namely, low-precision training. Supporting other optimizations like approximate or sparsity-aware computation require intrusive changes in these tools due to their tightly coupled design and implementation with the underlying ML frameworks and the targeting GPU backend. Second and more importantly, we observe that the current DNN emulators are susceptible to *numerical inaccuracies* due to a coarse-grained emulation strategy that they follow. Under this strategy, the low-precision arithmetic emulation is achieved by performing individual algebraic functions in the DNN model using the standard FP32 format supported in GPUs and then rounding the output down to the user-defined low-precision format. Even though this layer-level rounding approach lends itself to a simpler DNN emulator design, we observe that this strategy fails to accurately reproduce numerical errors that can occur with low-precision arithmetic. For instance, many DNN layers like matrix multiplication and convolution internally perform several primitive mathematical operations like multiplication and addition. Under the coarse-grained layer-level emulation approach, these primitive operations are performed using higher-precision FP32 arithmetic. As a result, they fail to account for the low-precision multiplication and accumulation error occurring within the algebraic functions.

$$rnd((a_1 * b_1) + (a_2 * b_2) + (a_3 * b_3)) \tag{1}$$

To illustrate this limitation, we use a simple matrix multiplication between a 1×3 matrix ($[a_1 \quad a_2 \quad a_3]$) and a 3×1 matrix ($[b_1 \quad b_2 \quad b_3]^T$) as an example. Equation 1 shows the definition of this matrix multiplication performed using FP16 arithmetic following the layer-level emulation strategy. In this case, all the primitive multiplication and addition operations are performed using FP32 arithmetic, and only the final result is rounded to FP16 using $rnd()$. In contrast, an accurate emulation of FP16 matrix multiplication requires rounding on every primitive operation as shown in Equation 2.

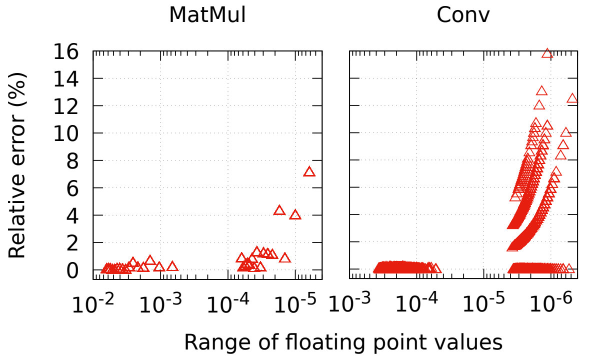


Figure 2: The numerical difference between real hardware and the layer-level emulation

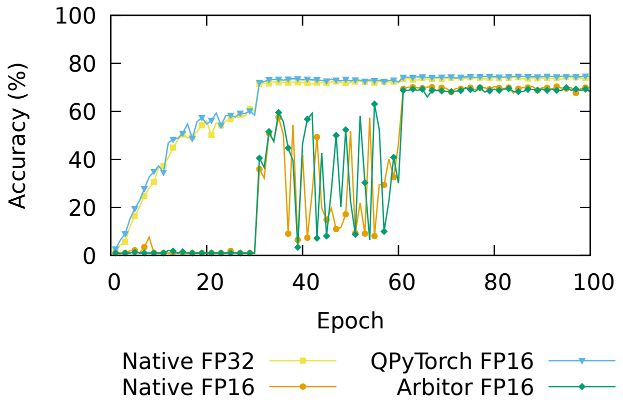


Figure 3: Comparison of training curves for ResNet-50 with native FP32, native FP16, QPyTorch-emulated FP16, and Arbitor-emulated FP16

$$rnd(rnd(rnd(a_1 * b_1) + rnd(a_2 * b_2)) + rnd(a_3 * b_3)) \tag{2}$$

To empirically estimate the extent of the inaccuracy, we emulate FP16 arithmetic using the layer-level strategy on matrix multiplication and 3×3 convolution operation over 128×128 matrices. Then we compare the output matrix generated with that obtained using the native FP16 supported in real hardware such as Nvidia 2080 Ti GPU [53]. Figure 2 shows the relative error between the emulated and the hardware native outputs on different input matrices with floating point values randomly generated from the range $[10^{-6}, 10^{-2}]$. As we show, in comparison to the GPU native FP16 results, the result generated by the emulated version can differ by as much as 15.74%. Since DNN training is an iterative process, such numerical inaccuracies in emulation can accumulate over the course of training, leading to making incorrect assessments about the effects of low-precision data formats on the DNN accuracy. To demonstrate this, we first train ResNet-50 using both FP32 and FP16 on Nvidia 2080 Ti and get validation accuracy curves as shown in Figure 3. The highest accuracy with FP16 is 70.36%, which is 3.7% lower than FP32 due to the numerical error with low-precision arithmetic. We subsequently use the state-of-the-art QPyTorch which applies layer-level emulation to run the training with emulated FP16. However, the validation accuracy using emulated FP16 in QPyTorch is observed to be close to the native FP32 accuracy. This shows that the layer-level emulation strategy fails accurately reproduce the numerical error of FP16 arithmetic.

Based on the above observations, we argue that current DNN emulators are ill-suited for accurately estimating the accuracy effects of low-precision arithmetic on DNN training due to their inherent design limitations.

4 Arbitor: Overview

To address the aforementioned limitations, we propose Arbitor. Arbitor strives to achieve three main goals. First, to propose a DNN emulator with an *extensible design* that can emulate common hardware-level optimizations like low-precision training, approximate computation, and sparsity-aware processing. Second, to enable *easy prototyping* of these optimizations and facilitate empirical analysis on their statistical effects on popular DNN models. Finally, we strive to provide *numerically accurate emulation* support that precisely mimics the functional logic of the hardware optimizations on general-purpose accelerators like GPUs. To achieve these goals, we make the following implementation choices.

First, we build Arbitor on top of the popular machine learning framework TensorFlow [3]. TensorFlow supports the convenient and user-friendly Keras front-end APIs for writing DNN training applications and offers implementations of a wide range of state-of-the-art DNN models [33]. To allow fast prototyping of low-precision arithmetic, approximate computation, and sparsity-aware processing on DNN models, Arbitor provides extensions in the Keras APIs for the users to define two emulation policies: (i) Data type policy for defining arbitrary precision numerical formats and the implementations of basic primitive mathematical operations on top of the user-defined data types. (ii) Masking policy for enforcing $N : M$ sparsity patterns on the parameter updates during training.

Providing numerically accurate emulation support for these hardware features requires fine-grained manipulations of the computations involved in DNN training workloads. However, manual modifications to the implementation of algebraic functions in the ML framework are intrusive and require significant engineering effort. Such modifications can also hamper the extensibility of the emulation tool, as supporting new DNN models or hardware features would require potential code changes. To address these challenges, we make a key observation: despite the apparent differences among hardware features like low-precision arithmetic, approximate computing, and sparsity-aware processing, all of them can be emulated by manipulating a small set of primitive operations, such as addition, multiplication, memory read and write, which constitute the DNN algebraic function implementations.

Based on this observation, we design Arbitor with the help of the XLA compiler in TensorFlow [23] to provide fine-grained emulation support. The XLA compiler operates on an intermediate representation (IR) named HLO IR, which precisely represents the computation through a concise set of fundamental primitive operations. This representation format of DNN computation is particularly well-suited for fine-grained manipulations to support numerically accurate emulation. Moreover, due to the model and hardware-independent nature of HLO IR, this compiler-based design of Arbitor provides seamless support for a wide range of DNN models and easy extensions for the emulation of more DNN accelerator optimizations. The XLA compiler compiles the high-level computation graph of DNN progressively down to hardware-specific kernels using standard code generation techniques, as illustrated in Figure 5. Initially, the computation graph described in the Keras front-end is transformed to HLO IR. The XLA compiler takes the HLO IR as the input and proceeds to perform a series of optimizations and analyses, ultimately lowering the HLO IR to hardware-specific kernel implementations. Notably, XLA leverages LLVM infrastructure for generating kernels on GPU, which is the target environment of Arbitor. During this process, Arbitor incorporates user-defined emulation policies into the computation by modifying the compilation pipeline.

Below, we provide details about the emulation policies and the fine-grained emulation strategy supported in Arbitor.

4.1 Emulation Policies

Figure 5 shows an example of Keras computation using the two emulation policies supported in Arbitor. Users can define and configure the `dtype` and `mask` policies using the Keras APIs. The policies can be assigned either to specific layers (in the Figure 5), or as a global policy for the entire DNN model.

The data type policy allows users to configure low-precision arithmetic and approximate computing emulation. Under this policy, the user can define a custom data type specification as an implementation of an abstract C++ class called `Cus`. The specification should include two components. First, two **casting functions** to convert the custom data type to and from the standard FP32 data type. Second, implementations of **primitive mathematical operations** over the custom data type such as addition and multiplication, and mathematical functions such as exponent and logarithm.

```
class EmuBF16: public Cus {
public:
    unsigned int v;
    EmuBF16(unsigned int v) : v(v) {}
};
EmuBF16 from_float(float f) {
    unsigned int bits = *(unsigned int*)&f;
    unsigned int rounding_bias = 0x7fff + ((bits >> 16) & 1)
    return EmuBF16((bits + roounding_bias) >> 16);
}
float to_float(EmuBF16 b) {
    unsigned int bits = b.v << 16;
    return *(float*)&tmp;
}
EmuBF16 operator+(const EmuBF16& a, const EmuBF16& b) {
    return from_float(to_float(a) + to_float(b));
}
EmuBF16 operator*(const EmuBF16& a, const EmuBF16& b) {
    return from_float(to_float(a) * to_float(b));
}
....
```

Figure 4: Custom data type specification of BF16

Figure 4 shows a specification of BF16 [32] data type in Arbitor named `EmuBF16`. In this example, `from_float` and `to_float` are the two casting functions. In addition, the Figure also shows example implementations of addition and multiplication operations defined over `EmuBF16`. It should be noted that the abstract class `Cus` makes very few assumptions about the specification of the data type and the implementations of the primitive operators. This allows Arbitor to support a wide range of arbitrary numerical formats and customized operator implementations on top of them to emulate low-precision arithmetic and approximate computing.

Next, the masking policy allows the users to configure an arbitrary $N : M$ sparsity pattern on the layers during the training process. In addition to N and M, the masking policy also takes a scoring function as a configuration parameter. The scoring function is a user-defined function that takes an array of M values as input and assigns an importance score for each value in the array. The masking policy defined in Figure 5 uses the absolute value as the scoring function. Based on these configurations, Arbitor dynamically generates masks for each parameter and gradient matrices accessed during the training and selects the top N values with the highest importance score on every block of M contiguous values in the matrix.

Once the policies are defined and assigned, Arbitor automatically takes care of emulating the corresponding hardware feature in the DNN training computation as we explain below.

4.2 Compiler-Aided Fine-Grained Emulation

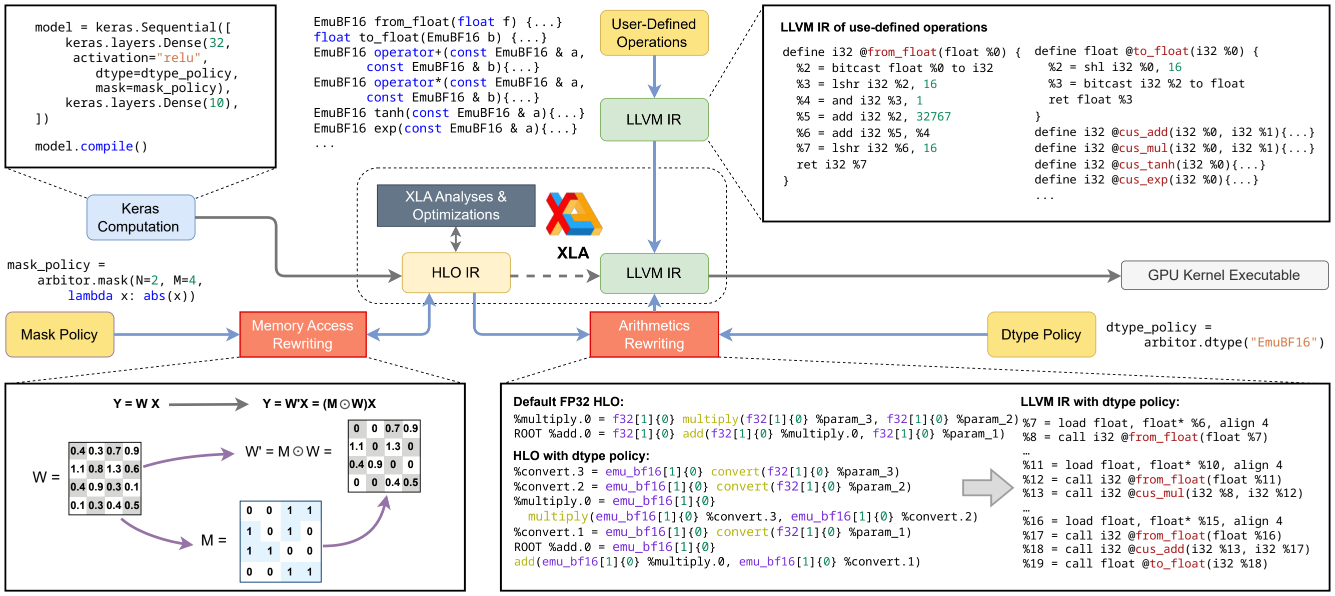
Arbitor generates customized GPU kernel implementations to achieve numerically accurate emulation of user-defined data types and masking policies by leveraging the XLA compiler. For the emulation of low-precision arithmetic and approximate computing, Arbitor injects user-defined data types and operator specifications into the generated kernels. This is accomplished by compiler passes that replace the primitive operations in the HLO IR with the corresponding user-defined operations and automatically insert casting operations for FP32 inputs, as shown in Figure 5. For example, when emulating the user-defined `EmuBF16` arithmetic in matrix multiplication, the default FP32 addition and multiplication operations within the kernel are replaced with the corresponding functions defined in the data type specification. This approach could also support emulating approximation of intricate mathematical operations, such as exponential function and hyperbolic tangent (tanh) function, where users are granted the capability to define customized implementations of these complex operations based on approximation techniques like the piecewise-linear approximation [35]. All of these are made possible because Arbitor makes few assumptions of specifics of data type representation and operator behavior that users provide.

Similarly, the masking policy in Arbitor enforces sparsity patterns on weight and gradient matrices by overriding the memory access operations to these matrices with masking operations, as shown in Figure 5. Therefore, whenever the values in the matrices are read during the training process, Arbitor dynamically generates the corresponding mask according to users' specifications and computes the element-wise product between the user-defined mask and the accessed matrix, returning the resulting masked values. This approach allows Arbitor to preserve the original matrices to handle dynamically changing sparsity patterns during training.

Emulating at the granularity of these primitive operations allows Arbitor to accurately mimic the numerical behaviors of the algebraic functions on real hardware, as shown in Equation 2. To empirically validate this, we re-run the experiment in Figure 2 and Figure 3 using Arbitor-emulated FP16 and compare the results against the hardware native FP16 results. The results of matrix multiplication and convolution using Arbitor's emulated FP16 precisely match with the hardware-native FP16 results with zero relative error, affirming that Arbitor is numerically accurate. Furthermore, we also show in Figure 3 that the validation accuracy curve of ResNet-50, employing Arbitor's emulated FP16 arithmetic, closely matches that obtained with native FP16. The margin of error in the validation accuracy achieved is only 0.48%. In contrast, the accuracy obtained by QPyTorch exhibits a discrepancy of 4.32% from native FP16 accuracy. These findings underscore Arbitor's capability to accurately estimate the effects of low-precision arithmetic on DNN training. In addition, we also analyse the emulation overhead of Arbitor compared to hardware native performance that can be found in Appendix B.

5 Arbitor: Case Studies

We build Arbitor to provide a reliable analysis tool for hardware research to explore DNN accelerator design space. We



```
model = keras.Sequential([
    keras.layers.Dense(32,
     activation="relu",
        dtype=dtype_policy,
        mask=mask_policy),
    keras.layers.Dense(10),
])

model.compile()
```

```
EmuBF16 from_float(float f) {...}
float to_float(EmuBF16 b) {...}
EmuBF16 operator+(const EmuBF16 & a,
       const EmuBF16 & b){...}
EmuBF16 operator*(const EmuBF16 & a,
       const EmuBF16 & b){...}
EmuBF16 tanh(const EmuBF16 & a){...}
EmuBF16 exp(const EmuBF16 & a){...}
...
```

```
define i32 @from_float(float %0) {
  %2 = bitcast float %0 to i32
  %3 = lshr i32 %2, 16
  %4 = and i32 %3, 1
  %5 = add i32 %2, 32767
  %6 = add i32 %5, %4
  %7 = lshr i32 %6, 16
  ret i32 %7
}
```

```
define float @to_float(i32 %0) {
  %2 = shl i32 %0, 16
  %3 = bitcast i32 %2 to float
  ret float %3
}
define i32 @cus_add(i32 %0, i32 %1){...}
define i32 @cus_mul(i32 %0, i32 %1){...}
define i32 @cus_tanh(i32 %0){...}
define i32 @cus_exp(i32 %0){...}
...
```

```
mask_policy =
    arbitor.mask(N=2, M=4,
        lambda x: abs(x))
```

```
dtype_policy =
     arbitor.dtype("EmuBF16")
```

$$W = \begin{bmatrix} 0.4 & 0.3 & 0.7 & 0.9 \\ 1.1 & 0.8 & 1.3 & 0.6 \\ 0.4 & 0.9 & 0.3 & 0.1 \\ 0.1 & 0.3 & 0.4 & 0.5 \end{bmatrix} \quad M = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \quad W' = M \odot W = \begin{bmatrix} 0 & 0 & 0.7 & 0.9 \\ 1.1 & 0 & 1.3 & 0 \\ 0.4 & 0.9 & 0 & 0 \\ 0 & 0 & 0.4 & 0.5 \end{bmatrix}$$

```
%multiply.0 = f32[1]{0} multiply(f32[1]{0} %param_3, f32[1]{0} %param_2)
ROOT %add.0 = f32[1]{0} add(f32[1]{0} %multiply.0, f32[1]{0} %param_1)
```

```
%convert.3 = emu_bf16[1]{0} convert(f32[1]{0} %param_3)
%convert.2 = emu_bf16[1]{0} convert(f32[1]{0} %param_2)
%multiply.0 = emu_bf16[1]{0}
  multiply(emu_bf16[1]{0} %convert.3, emu_bf16[1]{0} %convert.2)
%convert.1 = emu_bf16[1]{0} convert(f32[1]{0} %param_1)
ROOT %add.0 = emu_bf16[1]{0}
add(emu_bf16[1]{0} %multiply.0, emu_bf16[1]{0} %convert.1)
```

```
%7 = load float, float* %6, align 4
%8 = call i32 @from_float(float %7)
…
%11 = load float, float* %10, align 4
%12 = call i32 @from_float(float %11)
%13 = call i32 @cus_mul(i32 %8, i32 %12)
…
%16 = load float, float* %15, align 4
%17 = call i32 @from_float(float %16)
%18 = call i32 @cus_add(i32 %13, i32 %17)
%19 = call float @to_float(i32 %18)
```

Figure 5: Arbitor workflow and multiple layers of IR

illustrate the benefits of Arbitor through a series of case studies on common hardware-level optimizations proposed for DNN accelerators. Specifically, we conduct an in-depth empirical analysis on the sensitivity of low-precision training, approximate computing, and sparsity-aware processing on DNN training. To the best of our knowledge, we are the first open study to conduct a numerically accurate and extensive study on these optimizations. We would also like to highlight that the case studies presented in this section are just a few examples of the many potential applications of Arbitor. We believe Arbitor is flexible and extensible to provide emulation support for an even wider range of hardware features.

5.1 Experiment Setup

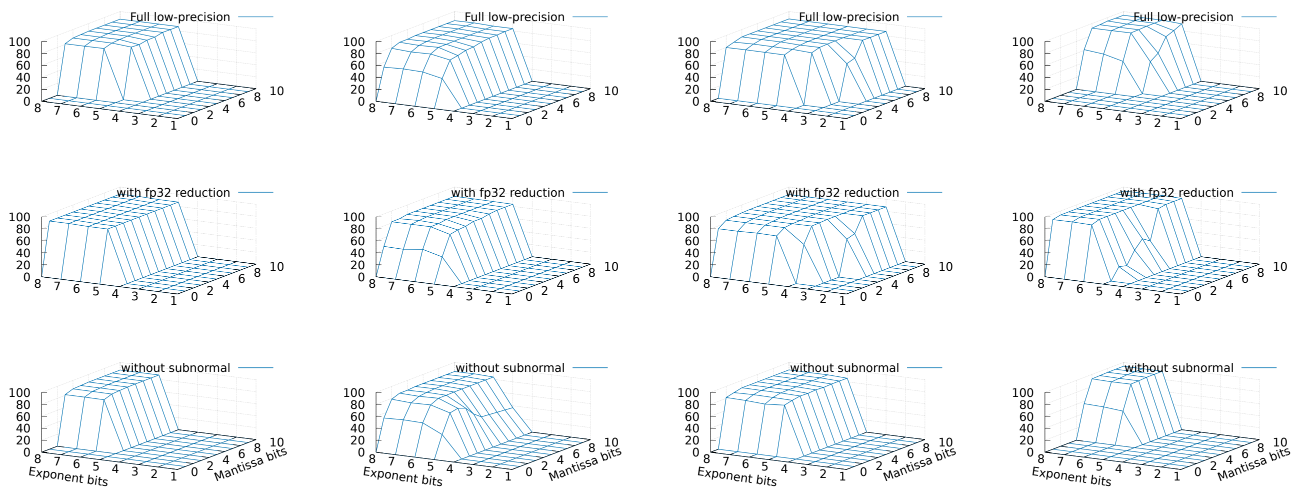
Model	Dataset	Accuracy
ResNet-18 [26]	CIFAR-10 [36]	93.78
GNN [73]	Cora [62]	84.06
Transformer [65]	FordA [1]	87.24
CRNN [66]	eng-fra [12]	87.25

Table 1: Benchmark models, datasets, and baseline accuracy

Workloads: Table 1 shows the DNN models we use in our experimental study, namely ResNet-18 [26], Transformer [65], GNN [73], and CRNN [66], selected from the Keras model hub [33]. These model implementations are based on a diverse range of model architectures that are part of the standard DNN training benchmark suite MLPerf [49]. For instance, ResNet-18 is a convolutional neural network (CNN) that is used for image classification applications. Transformer model is an attention-based DNN used in sequence classification and translation. GNN is a graph neural network used for node prediction in graph datasets. CRNN is a recurrent neural network (RNN) based model used for natural language translation.
Hardware and Runtime: We conduct our experiments on 32-core AMD EPYC 7371 machines with four Nvidia 2080Ti GPUs each with 12 GB memory. The runtime environment uses Ubuntu 20.04 with CUDA 11.0, cuDNN 8.0, and CUTLASS 2.6. Arbitor is built on top of TensorFlow v2.4.0 [3].
Metrics: In our experiments, we use the peak validation accuracy as the main evaluation metric. To measure the accuracy, we train ResNet-18 for 100 epochs, Transformer for 120 epochs, GNN for 300 epochs, and CRNN for 100 epochs on their respective data sets. We train each model three times and use their averaged accuracy for comparison to minimize the effects of slight variations in the final accuracy. We use the validation accuracy of each model trained using single-precision floating point (FP32) as the baseline for all our comparisons. The baseline accuracy of each model is shown in Table 1.

5.2 Case Study #1: Low-Precision Training

We use Arbitor to investigate the impact of low-precision arithmetic on the model accuracy by training the DNN models in Table 1 with different numerical formats. Current DNN accelerators support a variety of floating point formats with different numbers of exponent and mantissa bits, as described in Section 2. Therefore, we conduct a sensitivity study on different floating point formats and analyze how they affect the model accuracy. In addition, we also evaluate the effects of other numerical formats like Posit [25] on DNN training.



(a) CRNN (b) GNN (c) Transformer (d) ResNet-18

Figure 6: Validation accuracy of models trained using different exponent and mantissa widths of floating point format

5.2.1 Sensitivity Study on Floating Point Numbers

Single-precision floating point (FP32) is one of the most commonly used numerical formats for DNN training. According to IEEE 754 standard [2], FP32 is specified as follows.

$$value = \begin{cases} (-1)^s \times 2^{-126} \times 0.m & \text{if} \quad e = 0x00 \quad \text{(subnormal)} \\ NaN \text{ or } inf & \text{if} \quad e = 0xFF \\ (-1)^s \times 2^{(e-127)} \times 1.m & otherwise \quad \text{(normal)} \end{cases} \tag{3}$$

Here, s, e, and m correspond to the sign, exponent, and mantissa of the FP32 value. As shown in Equation 3, the specification primarily follows two different modes, normal and subnormal, depending on the value of the exponent. Other floating point formats supported in DNN accelerators also follow a similar specification but with different numbers of exponent and mantissa bits, each making different trade-offs in the range and precision of values it can represent.

To analyze the sensitivity of model accuracy towards different floating point specifications, we define a data type policy in Arbitor of a generic floating point specification with configurable exponent and mantissa bits based on the IEEE 754 standard. We use *eXmY* to represent a floating point specification with 1 sign bit, X exponent bits, and Y mantissa bits. Using the custom floating point format, we train the DNN models in two ways. First, we train and measure the accuracy of the models while only using the custom floating point format for the whole model. Second, we use a combination of the custom floating point and FP32 arithmetic following the mixed-precision training strategy described in Section 3. Under this strategy, all arithmetic operations are performed using the custom floating point except the reduction operations in algebraic functions like matrix multiplication, convolution, and batch normalization which are performed using FP32 arithmetic. Figures 6a to 6d shows the validation accuracy measured on models trained using custom floating point format by varying the number of exponent and mantissa bits.

Observation 1: DNN models can maintain their accuracy with much lower precision than many standard low-precision floating point formats supported in current DNN accelerators (e.g., BF16 and TF32). Additionally, the precision can be reduced even further by using single-precision arithmetic for reduction operations.

From our results, CRNN, GNN, Transformer, and ResNet-18 can train to the FP32 accuracy with *e6m6*, *e6m3*, *e6m4*, and *e6m6* respectively. That means, reserving 6 bits for the exponent is sufficient to represent the floating point values generated during the training of these models which is lower than the standard floating point formats like FP32 (*e8m23*), BF16 (*e8m7*), and TF32 (*e8m10*), but higher than FP16 (*e5m10*). Reducing the exponent and mantissa bits beyond these configurations either causes a drop in accuracy or makes the models fail to converge altogether. We also observe that the accuracy of the models is less sensitive towards mantissa bits than exponent bits. This reasserts the fact that an optimal numerical format for DNN training workloads should allocate more bits for exponent and less for mantissa to be able to represent a wide range of values. Moreover, the mantissa bits can be further reduced for CRNN, GNN, and ResNet-18 to *e6m1*, *e6m2*, and *e6m2* respectively using FP32 arithmetic for reduction operations. This suggests that the numerical errors in low-precision training are primarily contributed by reduction operations. Therefore, using a combination of low and high-precision floating point formats in training can provide higher training performance with little to no loss in accuracy.

Next, we evaluate the importance of handling subnormal numbers in floating point formats. The subnormal mode in the floating point specification is introduced to gracefully handle underflow for the values that fall between zero and the smallest floating point number normal mode can represent. However, handling the subnormal mode in floating point implementation adds extra complexity to the hardware design. Therefore, certain implementations like BF16 in TPUs [31] do not support subnormal mode. In Figures 6a to 6d, we show the model accuracy measured using the custom floating point specification but with the subnormal numbers set to zero.

Observation 2: Subnormal mode has negligible impact on the model accuracy.

Comparing the accuracy measurements with and without subnormal mode, we observe that the subnormal numbers can often be safely ignored on configurations with sufficient exponent bits without affecting the model accuracy in almost all cases. This is due to the fact that floating point specifications with larger exponent bits have a smaller subnormal range and therefore approximating a small range of subnormal numbers to zero only introduces minimal numerical error in low-precision training. However, in certain cases like ResNet-18, low-precision training without subnormal mode can cause a slight drop in the accuracy of about 1.46%. We also observe that this accuracy loss can be compensated by adding more mantissa bits. For instance, ResNet-18 can achieve the baseline accuracy using *e6m8* without subnormal compared to *e6m6* with subnormal mode.

5.2.2 Posit as an Alternative for Floating Point

Even though floating point formats are the industry standard, there have been proposals for alternative numerical formats in the literature. Posit [25] is one such example and is a novel data type designed as a direct drop-in replacement for IEEE standard 754 floating point formats. Posit format is purportedly more hardware-friendly with lower power use and a smaller silicon footprint and can perform more operations per watt and per dollar than floating points under the same hardware budget. Moreover, the original paper [25] and subsequent studies [42] have shown that Posit can represent decimal numbers more precisely than floating point format on common arithmetic and linear algebra operations. Therefore, Posit is considered to be a viable alternative for floating point in deep learning applications. In this section, we evaluate the effectiveness of Posit in DNN training with Arbitor.

An n-bit Posit format with es exponent bits, abbreviated as $P(n, es)$, is represented using a sign (s), regime (k), exponent (e), and fraction (f) bits as follows:

$$value = (-1)^s \times 2^{2^{es} \times k} \times 2^e \times (1.f) \quad (4)$$

Similar to the floating point sensitivity study, we specify a data type emulation policy in Arbitor using the software-based Posit implementation available in the BFP library [40]. Then we train CRNN, GNN, and Transformer models using Posit configurations with different n and es values. Figure 7 shows the validation accuracy measured on these models.

Observation 3: Contrary to the observations made in prior works [59], low-precision training with Posit does not yield better accuracy compared to floating point.

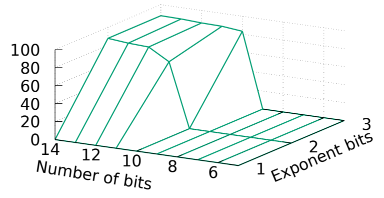
CRNN, GNN, and Transformer achieves the FP32 accuracy with *P(13,3)*, *P(11,3)*, and *P(9,2)* respectively. However, comparing a Posit and floating point configuration that uses the same number of bits in total, we observe they both converge to similar accuracy. This suggests that the higher precision of the Posit representation has limited benefits for DNN training workloads which are known to be tolerant of low-precision arithmetic. Despite this, we believe that Posit can still be a viable replacement for floating point in DNN accelerators due to its comparatively simpler and hardware-friendly design.

It is important to note that, the observation we make above goes against some of the prior works that claim that Posit can achieve better accuracy compared to floating point with a smaller hardware budget. For instance, Raposo et. al [59] have shown that 8-bit Posit can substitute 32-bit floating point for DNN training with no impact on accuracy. This underscores the importance of using a numerically accurate hardware emulation tool like Arbitor for such analysis. Moreover, the experimental evaluation conducted in this work was based on smaller models and datasets than what we use in our study. We believe our emulation tool can help hardware researchers to conduct accurate experimental analysis in the future and avoid making suboptimal design decisions.

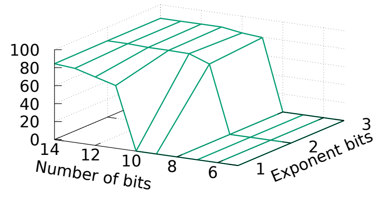
5.3 Case Study #2: Approximate Computing

In this section, we use Arbitor to conduct a sensitivity study on a common approximate computing technique called *piece-wise linear approximation*. As described in Section 2, piece-wise linear approximation has been proposed as a cost-effective way to support non-linear algebraic functions like exponents and tanh that are common in natural language processing (NLP) models [35, 61, 69]. The key idea of this optimization is to break down curves of a non-linear function into pieces of line segments each approximating a part of the curve. Therefore, the fewer the pieces of line segments, the higher the numerical error in the approximation.

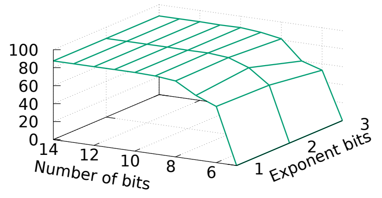
We analyze piece-wise linear approximation of the non-linear functions tanh and sigmoid in the CRNN model and their effect on the model accuracy by varying the number of pieces in the approximation. For this, we use the data type emulation policy in Arbitor to override the tanh and sigmoid function implementations of FP32 arithmetic with the piece-wise approximated version. Table 2a shows the validation accuracy curve of CRNN measured during the training.



(a) CRNN



(b) GNN



(c) Transformer

Figure 7: Validation accuracy of models trained using Posit format with different total bit widths and exponent bits

Observation 4: CRNN model can tolerate aggressive approximation on non-linear functions without affecting the accuracy.

We observe that even with an aggressive 3-piece approximation on both tanh and sigmoid functions, CRNN can maintain the baseline FP32 accuracy. This shows that piece-wise linear approximation is an effective optimization for DNN training. We also observe that a more aggressive 2-piece approximation on either function can significantly reduce the accuracy by up to 5%, and the accuracy drops further by 12% if both use 2-piece approximation. Since piece-wise linear approximation is often used in FPGAs and ASICs designed for NLP models, we believe our tool can be helpful in estimating the trade-offs of the approximation for specific models.

5.4 Case Study #3: Sparsity-Aware Processing

Finally, we use Arbitor to analyze how structured sparsity can affect the model accuracy. Sparsity-aware processing units are one of the recent innovations in DNN accelerators, e.g., Sparse Tensor Cores in Nvidia A100 [51]. Sparse tensor processing cores are designed to accelerate matrix multiplication over matrices with $N : M$ sparsity patterns, i.e., at most N values in every contiguous M block of values are non-zero. However, prior works [72] have pointed out that sparsity in DNN training is inherently unstructured, and enforcing any kind of structure to it in order to leverage Sparse Tensor Cores can affect the accuracy of the model.

We analyze the extent of these effects on ResNet-18 and Transformer with different sparsity patterns enforced on the weight parameter update operations during training. For this experiment, we define a masking policy in Arbitor with different N and M values. We use the absolute value of the weight parameter as the scoring function as it is one of the common heuristics for estimating the importance score in DNN pruning techniques [22]. Tables 2b and 2c shows the validation accuracy measured on ResNet-18 and Transformer.

Observation 5: The accuracy effects of sparse weight updates is highly model dependent. Moreover, enforcing more than 50% sparsity can have a significant impact on the model accuracy.

We observe that ResNet-18 can achieve close to the baseline FP32 accuracy with sparse weight updates. We also observe that ResNet-18 accuracy has slightly improved from the baseline accuracy with 50% sparsity patterns $1:2$, $2:4$, and $4:8$. We believe this improvement is due to the regularization effects of the sparse computation. On the other hand, Transformer shows a significant 4.3% drop in accuracy with 50% sparsity. This suggests that different models affect differently with sparsity-aware processing and require rigorous experimental analysis to accurately estimate the trade-offs. Applying patterns with more than 50% sparsity shows a significant drop in accuracy on both models by up to 7.5%.

6 Related Work

TensorQuant [45] is one of the first DNN emulation tools proposed and is specially designed for analyzing fixed point arithmetic primarily using layer-level quantization strategy. In addition, TensorQuant offers provision for fine-grained emulation strategy but requires users to write custom implementations for the operators in the DNN model. In contrast, Arbitor can support fine-grained emulation automatically with minimal effort from the user because of the compiler-based design that we follow. Moreover, due to the narrow focus on fixed point arithmetic emulation, the applicability of TensorQuant is limited to the analysis of DNN quantization methods. Arbitor, on the other hand, is capable of emulating arbitrary numerical formats using the data type emulation support and has a wider application. PositNN [59] is another DNN emulator built specifically for analyzing the efficacy of Posit numerical formats in DNN workloads. Therefore, like TensorQuant, PositNN also only has limited applicability. Moreover, PositNN uses a considerable amount of hand-written code for emulating Posit and only support a narrow range of DNN models. As we explain in Section 5.2.2, our analysis on a wider range of standard DNN models has refuted some

CRNN		
sigmoid	**tanh**	**Accuracy (%)**
3-piece	3-piece	88.12
2-piece	None	82.26
None	2-piece	84.92
2-piece	2-piece	70.95

(a) CRNN with piece-wise linear approximation

ResNet-18			
N \ **M**	**2**	**4**	**8**
1	94.07	93.64	92.99
2	-	94.09	93.59
4	-	-	94.19

(b) ResNet-18 with sparse weight updates

Transformer			
N \ **M**	**2**	**4**	**8**
1	82.11	78.78	75.45
2	-	82.94	80.72
4	-	-	82.94

(c) Transformer with sparse weight updates

Table 2: Validation accuracy of CRNN, ResNet-18, and Transformer with approximate and sparse computing

of the observations made using PositNN. Specifically, the authors of PositNN claim that 8-bit Posit format can substitute $32 - bit$ floating point, which we find to be untrue on the models that we analyzed.

More recent emulators, QPyTorch [70] and GoldenEye [47] are both implemented on top of PyTorch. These tools are primarily designed for exploring low-precision formats on DNN training and provide emulation support for a wide range of numerical formats like floating point, fixed point, and block floating point [21] and follow layer-level emulation strategy with the help of the `hook` functionality in the framework. GoldenEye also provides hardware error injection support to evaluate the reliability of DNN accelerators. However, as described in Section 3, these tools are susceptible to numerical inaccuracies in emulation which can lead to incorrect assessment of the hardware design. We argue that numerically accurate emulation should be a necessary quality of hardware emulators. Moreover, unlike QPyTorch and GoldenEye, Arbitor is capable of supporting other common optimizations like approximate and sparse computing.

7 Discussion

We have presented a numerically accurate approach for emulating hardware optimizations and estimating their effects on DNN accuracy to guide the hardware design. Below, we discuss the generalizability and limitations of Arbitor, as well as our future plans to overcome these limitations.

Supporting a broader set of numerical data formats. Although we focus on case studies of floating point formats and Posit formats in this work, Arbitor is designed to support any arbitrary data formats through the generic data type emulation policy. Hence, other floating point formats like FP8 and fixed point formats like INT8 can also be emulated under this generic emulation policy. We have also extended Arbitor to support more complex block-based data formats, such as block floating point format [21] and MSFP [19]. These formats employ a block-based representation, where values within each block share the same exponent. Emulating such data formats is more challenging compared to regular data formats that only require considering individual data points, as the operations for each value depend on global information like the value of the shared exponent. The extensible design of Arbitor allows for adding compiler passes to rewrite operations for different blocks to support such data formats.

Supporting more front-end frameworks. We chose to implement Arbitor on top of XLA because we find it to be the most mature ML compiler with training support and is well-integrated with popular front-end frameworks like TensorFlow [3] and JAX [11]. However, the compiler-based approach is not tied to any specific compiler backend. We plan to migrate Arbitor to the latest OpenXLA that is based on the MLIR [38] infrastructure. Additionally, since the concept of operation-level emulation is not tied to any specific compiler backend, it is possible to apply the design of Arbitor to other ML compilers like TorchDynamo [67].

Supporting other complex hardware features. In this work, we demonstrate Arbitor's support for the three most common categories of hardware-level optimizations supported in DNN accelerators. However, it is worth noting that there exist intricate hardware features that are out of Arbitor's current scope. For instance, complex custom processing units, such as $3D$ cube in Huawei NPU [39], are currently not supported by Arbitor. We consider addressing these advanced hardware features as future work.

8 Conclusion

In this paper, we showcase that hardware researchers currently lack an accurate hardware analysis tool for empirically evaluating different design choices for DNN training accelerators. To fill this gap, we propose Arbitor, a hardware emulation tool for analysing common hardware optimization strategies like low-precision training, approximate computing, and sparsity-aware processing. Unlike prior emulators, Arbitor follows an extensible design and numerically accurate emulation support with the assistance of modern machine learning compilers like TensorFlow XLA. We subsequently demonstrate the utility of Arbitor by conducting an extensive sensitivity analysis on the aforementioned optimization strategies and their influence on the accuracy of popular DNN models.

Acknowledgments

We would like to thank our shepherd and the anonymous reviewers for their valuable feedback and comments. We also thank members of the EcoSystem lab for their support, inspiration, and constructive feedback during the development of this work. This project was supported by the Canada Foundation for Innovation JELF grant, NSERC Discovery grant, AWS Machine Learning Research Award (MLRA), Facebook Faculty Research Award, Google Scholar Research Award, and VMware Early Career Faculty Grant.

Availability

The artifact of this paper is open-sourced on GitHub (https://github.com/arbitor-project/artifact).

References

[1] Forda: A dataset for time series classification.

[2] Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.

[3] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[4] Ayaz Akram and Lina Sawalha. A survey of computer architecture simulation techniques and tools. *IEEE Access*, 7:78120–78145, 2019.

[5] Amazon Web Services. AWS Trainium.

[6] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse Engel, Linxi Fan, Christopher Fougner, Tony Han, Awni Hannun, Billy Jun, Patrick LeGresley, Libby Lin, Sharan Narang, Andrew Ng, Sherjil Ozair, Ryan Prenger, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Yi Wang, Zhiqian Wang, Chong Wang, Bo Xiao, Dani Yogatama, Jun Zhan, and Zhenyao Zhu. Deep speech 2: End-to-end speech recognition in english and mandarin, 2015.

[7] Giorgos Armeniakos, Georgios Zervakis, Dimitrios Soudris, and Jörg Henkel. Hardware approximate techniques for deep neural network accelerators: A survey. *ACM Computing Surveys*, 55(4):1–36, nov 2022.

[8] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 163–174, 2009.

[9] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[10] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning? In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 129–146, 2020.

[11] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[12] Jason Callaway. fra-txt-details. Kaggle Dataset.

[13] Zachariah Carmichael, Hamed F. Langroudi, Char Khazanov, Jeffrey Lillie, John L. Gustafson, and Dhireesha Kudithipudi. Deep positron: A deep neural network using the posit number system. In *2019 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1421–1426, 2019.

[14] Cerebras. Cerebras Wafer-Scale Engine.

[15] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa,

Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

[16] Yiran Chen, Yuan Xie, Linghao Song, Fan Chen, and Tianqi Tang. A survey of accelerator architectures for deep neural networks. *Engineering*, 6(3):264–274, 2020.

[17] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways, 2022.

[18] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications, 2014.

[19] Bita Darvish Rouhani, Daniel Lo, Ritchie Zhao, Ming Liu, Jeremy Fowers, Kalin Ovtcharov, Anna Vinogradsky, Sarah Massengill, Lita Yang, Ray Bittner, et al. Pushing the limits of narrow precision inferencing at cloud scale with microsoft floating point. *Advances in neural information processing systems*, 33:10271–10281, 2020.

[20] David Kaplan. When hardware must just work, 2015.

[21] Mario Drumond, Tao Lin, Martin Jaggi, and Babak Falsafi. Training dnns with hybrid block floating point. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 451–461, Red Hook, NY, USA, 2018. Curran Associates Inc.

[22] Trevor Gale, Erich Elsen, and Sara Hooker. The state of sparsity in deep neural networks, 2019.

[23] Google. TensorFlow XLA, 2018.

[24] GraphCore. Introducing the Colossus MK2 GC200 IPU.

[25] Gustafson and Yonemoto. Beating floating point at its own game: Posit arithmetic. *Supercomput. Front. Innov.: Int. J.*, 4(2):71–86, jun 2017.

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[27] Meyr Heinrich, Lüthje Olaf, Holger Keding, and Coors Martin. Design and dsp implementation of fixed-point systems. *EURASIP Journal on Advances in Signal Processing*, 2002, 09 2002.

[28] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus), 2016.

[29] James C. Hoe, Doug Burger, Joel Emer, Derek Chiou, Resit Sendag, and Joshua Yi. The future of architectural simulation. *IEEE Micro*, 30(3):8–18, 2010.

[30] Intel. Gaudi2: High Performance Training and Inference Solution.

[31] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, 2017. ACM.

[32] Dhiraj D. Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth

Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep Dubey. A study of BFLOAT16 for deep learning training. *CoRR*, abs/1905.12322, 2019.

[33] Keras. Keras Code Examples.

[34] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. Accel-sim: An extensible simulation framework for validated gpu modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 473–486, 2020.

[35] Seok Young Kim, Chang Hyun Kim, and Seon Wook Kim. Applying piecewise linear approximation for dnn non-linear activation functions to bfloat16 macs. In *2021 International Conference on Electronics, Information, and Communication (ICEIC)*, pages 1–4, 2021.

[36] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research).

[37] Abhisek Kundu, Alex Heinecke, Dhiraj Kalamkar, Sudarshan Srinivasan, Eric C. Qin, Naveen K. Mellempudi, Dipankar Das, Kunal Banerjee, Bharat Kaul, and Pradeep Dubey. K-tanh: Efficient tanh for deep learning, 2019.

[38] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021.

[39] Heng Liao, Jiajin Tu, Jing Xia, and Xiping Zhou. Davinci: A scalable architecture for neural network computing. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–44, 2019.

[40] libcg. bfp - Beyond Floating Point.

[41] Darryl D. Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 2849–2858. JMLR.org, 2016.

[42] Peter Lindstrom, Scott Lloyd, and Jeffrey Hittinger. Universal coding of the reals: Alternatives to ieee floating point. In *Proceedings of the Conference for Next Generation Arithmetic*, CoNGA '18, New York, NY, USA, 2018. Association for Computing Machinery.

[43] Zhenhong Liu, Amir Yazdanbakhsh, Taejoon Park, Hadi Esmaeilzadeh, and Nam Sung Kim. Simul: An algorithm-driven approximate multiplier design for machine learning. *IEEE Micro*, 38(4):50–59, 2018.

[44] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 11966–11976, 2022.

[45] Dominik Marek Loroch, Norbert Wehn, Franz-Josef Pfreundt, and Janis Keuper. Tensorquant - a simulation toolbox for deep neural network quantization, 2017.

[46] Jinming Lu, Siyuan Lu, Zhisheng Wang, Chao Fang, Jun Lin, Zhongfeng Wang, and Li Du. Training deep neural networks using posit number system, 2019.

[47] Abdulrahman Mahmoud, Thierry Tambe, Tarek Aloui, David Brooks, and Gu-Yeon Wei. Goldeneye: A platform for evaluating emerging numerical data formats in dnn accelerators. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 206–214, 2022.

[48] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training, 2017.

[49] MLPerf. MLPerf Training v0.6 Results, 2019.

[50] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *CoRR*, abs/1906.00091, 2019.

[51] Nvidia. NVIDIA AMPERE GA102 GPU ARCHITECTURE.

[52] Nvidia. NVIDIA Tesla P100 White Paper.

[53] Nvidia. NVIDIA Turing GPU ARCHITECTURE.

[54] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy,

Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[55] Harry A. Pierson and Michael S. Gashler. Deep learning in robotics: A review of recent research, 2017.

[56] Jeff Pool and Chong Yu. Channel permutations for n:m sparsity. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 13316–13327. Curran Associates, Inc., 2021.

[57] PyTorch. RFC: Should matmuls use tf32 by default?

[58] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions, 2018.

[59] Gonçalo Raposo, Pedro Tomás, and Nuno Roma. PositNN: Training Deep Neural Networks with Mixed Low-Precision Posit. In *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, page 7908–7912. IEEE, jun 2021.

[60] run.ai. FPGA for Deep Learning.

[61] Maicon A. Sartin and Alexandre C. R. da Silva. Approximation of hyperbolic tangent activation function using hybrid methods. In *2013 8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–6, 2013.

[62] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Gallagher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93–93, 2008.

[63] Yifan Sun, Trinayan Baruah, Saiful A. Mojumder, Shi Dong, Xiang Gong, Shane Treadway, Yuhui Bao, Spencer Hance, Carter McCardwell, Vincent Zhao, Harrison Barclay, Amir Kavyan Ziabari, Zhongliang Chen, Rafael Ubal, José L. Abellán, John Kim, Ajay Joshi, and David Kaeli. Mgpusim: Enabling multi-gpu performance modeling and optimization. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 197–209, New York, NY, USA, 2019. Association for Computing Machinery.

[64] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2sim: A simulation framework for cpu-gpu computing. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 335–344, 2012.

[65] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[66] Ruishuang Wang, Zhao Li, Jian Cao, Tong Chen, and Lei Wang. Convolutional recurrent neural networks for text classification. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–6, 2019.

[67] Peng Wu. Pytorch 2.0: The journey to bringing compiler technologies to the core of pytorch (keynote). In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, pages 1–1, 2023.

[68] Zhuliang Yao, Shijie Cao, Wencong Xiao, Chen Zhang, and Lanshun Nie. Balanced sparsity for efficient dnn inference on gpu. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence*, AAAI'19/IAAI'19/EAAI'19. AAAI Press, 2019.

[69] Babak Zamanlooy and Mitra Mirhassani. Efficient vlsi implementation of neural networks with hyperbolic tangent activation function. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(1):39–48, 2014.

[70] Tianyi Zhang, Zhiqiu Lin, Guandao Yang, and Christopher De Sa. Qpytorch: A low-precision arithmetic simulation framework. *ArXiv*, abs/1910.04540, 2019.

[71] Xishan Zhang, Shaoli Liu, Rui Zhang, Chang Liu, Di Huang, Shiyi Zhou, Jiaming Guo, Qi Guo, Zidong Du, Tian Zhi, and Yunji Chen. Fixed-point back-propagation training. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2327–2335, 2020.

[72] Aojun Zhou, Yukun Ma, Junnan Zhu, Jianbo Liu, Zhijie Zhang, Kun Yuan, Wenxiu Sun, and Hongsheng Li. Learning n:m fine-grained structured sparse neural networks from scratch. In *International Conference on Learning Representations*, 2021.

[73] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, and Maosong Sun. Graph neural networks: A review of methods and applications. *CoRR*, abs/1812.08434, 2018.

[74] Feng Zhu, Ruihao Gong, Fengwei Yu, Xianglong Liu, Yanfei Wang, Zhelong Li, Xiuqi Yang, and Junjie Yan. Towards unified int8 training for convolutional neural network, 2019.

[75] H. Zhu, M. Akrout, B. Zheng, A. Pelegris, A. Jayarajan, A. Phanishayee, B. Schroeder, and G. Pekhimenko. Benchmarking and analyzing deep neural network training. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 88–100, Sep. 2018.

A Artifact Appendix

Abstract

We provide the source code and scripts for Arbitor to reproduce the results of our experiments (in Section 3 and 4) and case studies (in Section 5) presented in the paper. While our paper explores three hardware optimizations: low-precision arithmetic, approximate computing, and sparsity-aware processing, the artifact focuses specifically on low-precision arithmetic. To this end, it contains the validation of accurate emulation of Arbitor. Additionally, the artifact conducts a sensitivity study between the validation accuracy and a subset of floating-point formats we study in the paper, similar to Figure 6. By following our provided instructions, users can expect to obtain results that closely align with those presented in the paper.

Scope

The artifact comprises two main components. Firstly, it generates the validation accuracy of training ResNet-18 using FP16 format emulated with Arbitor and QPyTorch and compares it to the native FP16 accuracy on GPU. This analysis highlights the negligible difference in accuracy between Arbitor-emulated FP16 and native FP16 while revealing a significant accuracy difference when using QPyTorch. Therefore, it could validate our claim that prior approaches like QPyTorch struggle to accurately replicate the numerical behavior of training with specific data formats, such as FP16, while Arbitor is designed to overcome this limitation.

Furthermore, the artifact automates the training process of GNN using a subset of floating-point formats evaluated in the paper, employing full low-precision training. This sensitivity study explores the variations in validation accuracy as the floating-point data format changes, thus validating the results presented in the corresponding section (Section 5) and supporting all associated claims. The results of all these training instances will be aggregated to generate a figure similar to Figure 6b but with fewer data points.

Contents

The artifact includes the source code of Arbitor, providing researchers with the ability to reproduce and modify the implementation. In addition to the evaluated ResNet-18 and GNN model, the artifact also includes the other two models, CRNN and Transformer, used in the case studies. Their corresponding datasets, including CIFAR-10 for ResNet-18, Cora for GNN, FordA for Transformer, and eng-fra for CRNN (as outlined in Table 1), are provided in the artifact. Therefore, researchers can readily evaluate these models and datasets, allowing for result replication, fast customization, and further investigation. Furthermore, the artifact provides a set of predefined data formats tailored to the experiments and case studies in the paper, including the floating-point and Posit formats, along with guidelines for incorporating these data formats in the training.

Hosting

The artifact can be downloaded from the main branch of GitHub link https://github.com/arbitor-project/artifact.

Requirements

A.0.1 Hardware requirement:

Arbitor requires the use of a multi-core CPU and an NVIDIA GPU with the Turing architecture or a more advanced counterpart to run the artifact. In our experiment, we employed four NVIDIA 2080Ti cards, but augmenting computation power by utilizing GPUs like NVIDIA A100, could further optimize the efficiency of artifact execution.

A.0.2 Software requirement:

The experiments provided in this artifact is prepared to run inside a docker container. We recommend using a machine with Ubuntu 20.04 with docker installed to reproduce the results.

Installation and Environment Setup

We provide docker files to set up the runtime environment for all the experiments.

- Install docker following the instructions in https://docs.docker.com/engine/install/ubuntu/.
- Make sure the machine has NVIDIA GPU(s) and the corresponding driver installed. If the NVIDIA driver is not installed, follow the instructions at NVIDIA tutorial to install it.
- Clone the git repository using the following command:
  ```
  git clone --recursive
  https://github.com/arbitor-project/artifact
  ```
- Build a docker image and enter the docker environment:
  ```
  cd artifact && bash run.sh
  ```

Experiment Workflow

We provide an end-to-end script to run everything all at once. When inside the docker environment, execute `e2e.sh` to run all the experiments and generate the results. The total execution may take several days to finish.

We also provide the step-by-step workflow as shown below:

1. Reproduce QPytorch ResNet-18 result with emulated FP16: `bash qpytorch.sh`

2. Reproduce Tensorflow ResNet-18 result with native FP16:
   `cd native_half && bash ./expr.sh resnet`

3. Reproduce Arbitor ResNet-18 result with emulated FP16:
   `cd /root/arbitor && bash ./expr.sh resnet`

4. Generate a subset of data formats for GNN training sensitivity study:
   `cd /root/ && bash ./gnn.sh`

Evaluation and Expected Results

Once the above execution has finished, two files, namely `results/validation.csv` and `results/sens.pdf` will be generated. `results/validation.csv` presents the final validation accuracy for three configurations: QPyTorch, native, and Arbitor FP16. It is expected that the validation accuracy of QPyTorch will be closely aligned with the FP32 baseline accuracy (93.78%), while both results of native and Arbitor are around 2% less than FP32 accuracy, consistent with the findings from Figure 3. Since the training process involves inherent randomness, it may be necessary to run multiple trials and compute an average for more accurate results. The file `results/sens.pdf` encompasses a figure depicting the relationship between accuracy, exponent bits, and mantissa bits. This is anticipated to bear resemblance to the first figure of Figure 6b, with fewer data points presented.

Experiment Customization

The emulated data format can be modified by modifying the `arbitor/data_format.sh` script to specify properties of data formats. Specifically, `F_OR_P` decides whether to emulate float or posit numbers. `ACC` represents the data type for accumulation, where `f32_acc` is to use FP32 to accumulate during a dot product, and `cus_acc` is to use the same type as computation for accumulation. In floating point configs, `EXP` and `MANTISSA` represent the bit-width of exponent and mantissa respectively. Setting `SUBNORMAL=_subnormal` causes subnormal numbers to be enabled during emulation and `SUBNORMAL=_wo_subnormal` otherwise. For Posit configs, `POSIT_NBITS` is the whole width of the number format, and `POSIT_ES` is the width of the exponent of Posit.

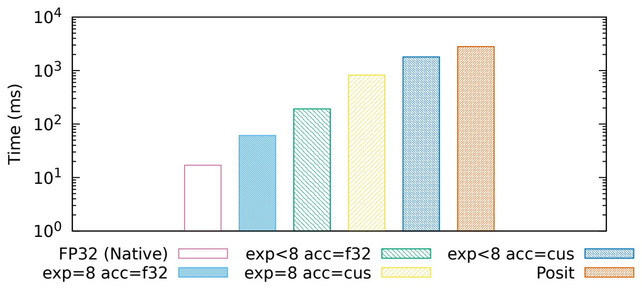


Figure 8: GNN training time with different number formats emulated by Arbitor

After changing `data_format.sh`, run

```
bash ./expr.sh [gnn | transformer | crnn |
resnet]
```

to train the model with the specified data format.

B Overhead of Arbitor

In addition to accurate emulation, generating customized GPU kernels enables execution of Arbitor on GPU, in contrast to accurate software-based simulators that primarily rely on CPU-based execution. This allows Arbitor to leverage the high bandwidth and massive parallelism of GPU to achieve a level of performance that is more practical for executing DNN training workload compared to software-based simulators. To demonstrate the performance impact of Arbitor, we train the whole GNN model with different data formats emulated by Arbitor using the same experiment setup in Section 5.1. Figure 8 shows the training time for each step with different data formats, where `exp` is the width of exponent bits, and `acc=cus` or `acc=f32` represents whether the accumulation type is the same as computation type or is FP32. Emulating different data formats introduces different amounts of overhead, as the execution of emulated operations for each format requires a distinct number of cycles on the GPU. For the floating point format with 8 exponent bits with FP32 accumulation, Arbitor incurs a 3.59× overhead. When the data format is significantly different from FP32, such as Posit, Arbitor could introduce a slowdown of up to 164.7×. These results show that with the GPU acceleration, the performance of Arbitor is significantly better than software-based simulators and is practical for the accurate emulation of DNN training. Moreover, Arbitor can also leverage data parallel training to scale the training over multiple GPUs to compensate for the emulation overhead.