Dragon-Alpha&cu32: Fully Open-source Java Deep-Learning Framework And High-Performance CUDA Library

Anonymous Authors[1]

Abstract

Java is powerful, but its potential in deep-learning (DL) realm remains untapped. At present, Python DL frameworks are predominantly preferred over those based on Java, mainly attributed to their superior usability and flexibility. To leverage Java's capabilities, Dragon-Alpha, a Java tensor computing framework, has been developed. The primary goal of Alpha is to provide ease of use, high-performance, and scalability. Alpha offers high-level user-friendly APIs for DL, and other levels of APIs to meet diverse requirements. Alpha achieves GPU acceleration via cu32 library, and improves efficiency through techniques like async APIs, C-K-S algorithm, Im2col-Winograd, and fused operators. Alpha has the potential to consolidate computing power across heterogeneous platforms and devices, based on its multi-layer architecture and compatibility with Java big-data ecosystem. Alpha&cu32 are fully open-source, with no reliance on external libraries. Experiments show that Alpha&cu32 surpass PyTorch&cuDNN on Cifar10 and ILSVRC2012 in certain scenarios.

1. Introduction

Deep learning (DL) is a highly hot field of artificial intelligence. Deep neural networks (DNNs) have been widely applied in diverse fields, yielding remarkable achievements. However, the impressive capabilities of DNNs come with increased complexity. To simplify the representation of DNNs and accelerate their execution, DL frameworks have been developed.

Python DL frameworks (Paszke et al., 2019; Abadi et al., 2016; Jia et al., 2014; Tokui et al., 2015; Chollet) have gained immense popularity, due to their user-friendly and flexible nature. On the other hand, Java DL frameworks (Gibson et al., 2016; Feenster et al.) are not as popular as their Python counterparts, mainly because of their more intricate APIs and limited adaptability. However, Java itself has many benefits, including robustness, speed, flexibility, platform independence, community support, and a strong big-data ecosystem, suggesting that Java's potential in DL field has not been fully exploited.

For efficiency, DL frameworks integrate acceleration libraries (acclibs) to take advantage of parallel processors like GPUs, CPUs, and FPGAs. Nevertheless, many widely-used GPU acclibs such as cuBLAS and cuDNN (Chetlur et al., 2014) are not open-source. The details of high-performance GPU programming remain somewhat opaque. As a result, there needs open-source GPU acclibs, that can help users better understand and utilize GPUs.

To address these issues, Dragon-Alpha&cu32 have been developed. As a Java tensor computing framework, Alpha can be used to express and execute DL algorithms. Cu32 is an efficient acclib for *float32* computations on GPUs, and has been integrated into Alpha. Alpha&cu32 are fully open-source, and only require JDK and CUDA for execution. Besides, cu32 adopts C-K-S algorithm and Im2col-Winograd to accelerate convolution.

In experiments conducted on Cifar10 and ILSVRC2012 datasets, Alpha&Cu32 exhibit comparable efficacy and convergence to PyTorch&cuDNN, and outperform them in certain circumstances.

2. Background

2.1. Deep Learning Frameworks

DNNs are multi-layer neural networks, with complex computational graphs and numerous parameters. Their complexity allows the learning and expression of sophisticated patterns, but brings challenges for concise representation and rapid execution. DL frameworks are designed to tackle these problems, with ease of use and efficiency as their key priorities.

DL frameworks offer a set of reusable primitives, based on the commonly used structures of DNNs. These primitives can be linked to acclibs for faster processing. Users are

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

required to define the forward propagation of DNNs using these primitives, while the automatic differentiation takes care of the backpropagation. Such propagation can be conceptualized as dataflows. Many DL frameworks, like Caffe (Jia et al., 2014), TensorFlow (Abadi et al., 2016), Keras (Chollet), and DeepLearning4j (DL4j) (Gibson et al., 2016), use static flows, whereas PyTorch (Paszke et al., 2019) and Chainer (Tokui et al., 2015) use dynamic flows. Static flows are pre-compiled and remain constant throughout the repeated batch calculations, while dynamic flows are created by the interaction between layers in forward propagation. Static flows are more predictable, making them easier to optimize and potentially leading to higher performance; dynamic dataflows have greater flexibility and usability, as they have larger state space.

The encapsulation of acclibs provides convenience, but may hide certain specifics and restrict functionalities. While operators in DL frameworks are easy to use, it may be challenging to fine-tune computing strategies, use operators beyond the parameter constraints, or avoid redundant mechanisms in particular scenarios.

2.2. Java or Python

At present, the primary advantages of Python DL frameworks over the Java ones are user-friendliness and flexibility. Java frameworks, like DL4j and Deep Java Library (DJL) (Feenster et al.), typically adhere to conventional Java programming patterns, which are proficient in managing complex relations, but may be cumbersome for progress-oriented tasks such as executing DNNs. Relatively, their APIs are more complex and overly encapsulated. In DL4j and DJL, DNNs are created using a series of chained invocations to builders, allowing for flexible hyperparameter configuration but may need a substantial amount of code. The training process is encapsulated into objects, including Trainer and Listener, which promotes standardization but may conceal certain details and limit customized options. In DL4j, the model, optimizer, and loss function are configured in a single builder, instead of completely decoupled, potentially compromising flexibility.

Conversely, Java frameworks have advantages over the Python ones. They can be integrated into Java big-data ecosystem to harness computing power across different platforms, particularly beneficial for training large models. Besides, they leverage Java's advantages over Python's limitations. Java is a compiled language with static datatype, while Python is an interpreted language using dynamic datatype. Given their nature, Java is generally more efficient, robust, and secure. Java uses multiple threads for parallelism, whereas Python relies on costly processes. Therefore, Java potentially has higher parallelism and better hardware utilization. At times, Python's simplicity may obscure specifics and impose constraints, while Java's complexity can offer advanced capabilities for tailing solutions to diverse cases.

By capitalizing on Java's strengths and adopting innovative designs, it's feasible to develop a DL framework with ease of use and efficiency.

3. Characteristics

To develop Alpha&Cu32, the goal is to attain a balance of usability, high-performance, and scalability. The focus is not only on the advancement of specific technologies, but also on the overall coherence of the entire system.

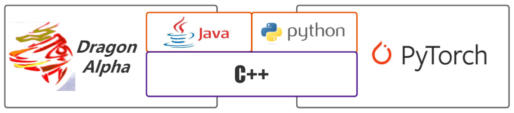


Figure 1. Alpha is implemented by C++ at bottom and Java at top

3.1. Different Levels of APIs

Alpha supports different levels of APIs, including user-friendly high-level APIs and complex low-level APIs. The top-level APIs enable Alpha to work as a DL framework; the lower-level APIs serve as intermediary zones between the top-level APIs and acclibs, offering better control over system specifics. Users can overcome the limitations of higher-level APIs via lower-level APIs, to directly manipulate memory addresses, adjust computing strategies, and circumvent certain mechanisms. Based on different levels of APIs, it's feasible to develop various applications, even a different DL framework.

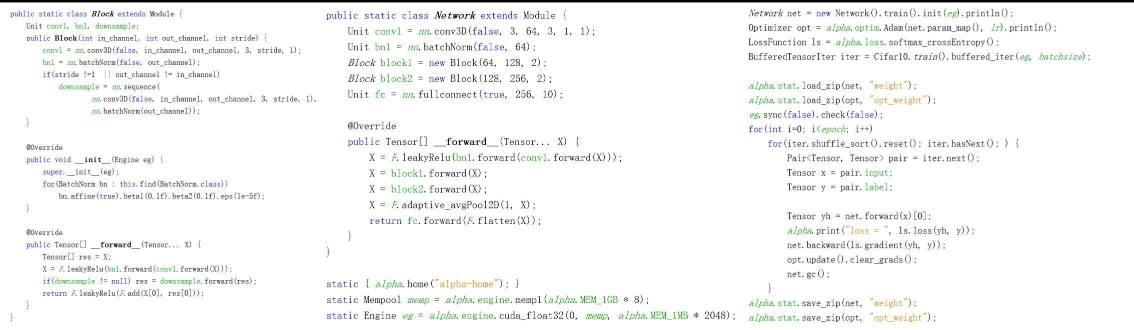
3.2. Easy-to-use

Alpha's high-level APIs are succinct and easy to use, incorporating the facade pattern, factory pattern, and chained invocation. To facilitate programming and identifying key information, the naming conventions are brief and minimize the use of capital letters. The coding style combines the elements of Java and Python, making it recognizable to users of both languages.

Alpha supports dynamic dataflows for DNNs. The provided operators resemble mathematical formulas in their structure. Hyperparameters can be specified using constructors and factories, and further modified through chain invocations. To construct a directed computational graph of DNNs, users can easily create a subclass of Module and override the *forward* method. Backpropagation can be accomplished with the assistance of automatic differentiation. Figure 2 illustrates a relevant example.

3.3. Scalability

As a Java application, Alpha is platform-independent and can be integrated into Java big-data frameworks such as Hadoop (Cutting et al.) and Spark (Paszke et al., 2010). Its multi-layer architecture separates the hardware specifics from the top layers, enabling polymorphism for different



```
public static class Block extends Module {
    Unit conv1, bn1, downsample;
    public Block(int in_channel, int out_channel, int stride) {
        conv1 = nn.conv3D(false, in_channel, out_channel, 3, stride, 1);
        bn1 = nn.batchNorm(false, out_channel);
        if(stride !=1  || out_channel != in_channel)
            downsample = nn.sequence(
                    nn.conv3D(false, in_channel, out_channel, 3, stride, 1),
                    nn.batchNorm(out_channel));
    }

    @Override
    public void __init__(Engine eg) {
        super.__init__(eg);
        for(BatchNorm bn : this.find(BatchNorm.class))
            bn.affine(true).beta1(0.1f).beta2(0.1f).eps(1e-5f);
    }

    @Override
    public Tensor[] __forward__(Tensor... X) {
        Tensor[] res = X;
        X = F.leakyRelu(bn1.forward(conv1.forward(X)));
        if(downsample != null) res = downsample.forward(res);
        return F.leakyRelu(F.add(X[0], res[0]));
    }
}
```

```
public static class Network extends Module {
    Unit conv1 = nn.conv3D(false, 3, 64, 3, 1, 1);
    Unit bn1 = nn.batchNorm(false, 64);
    Block block1 = new Block(64, 128, 2);
    Block block2 = new Block(128, 256, 2);
    Unit fc = nn.fullconnect(true, 256, 10);

    @Override
    public Tensor[] __forward__(Tensor... X) {
        X = F.leakyRelu(bn1.forward(conv1.forward(X)));
        X = block1.forward(X);
        X = block2.forward(X);
        X = F.adaptive_avgPool2D(1, X);
        return fc.forward(F.flatten(X));
    }
}

static { alpha.home("alpha-home"); }
static Mempool memp = alpha.engine.memp1(alpha.MEM_1GB * 8);
static Engine eg = alpha.engine.cuda_float32(0, memp, alpha.MEM_1MB * 2048);
```

```
Network net = new Network().train().init(eg).println();
Optimizer opt = alpha.optim.Adam(net.param_map(), lr).println();
LossFunction ls = alpha.loss.softmax_crossEntropy();
BufferedTensorIter iter = Cifar10.train().buffered_iter(eg, batchsize);

alpha.stat.load_zip(net, "weight");
alpha.stat.load_zip(opt, "opt_weight");
eg.sync(false).check(false);
for(int i=0; i<epoch; i++)
    for(iter.shuffle_sort().reset(); iter.hasNext(); ) {
        Pair<Tensor, Tensor> pair = iter.next();
        Tensor x = pair.input;
        Tensor y = pair.label;

        Tensor yh = net.forward(x)[0];
        alpha.print("loss = ", ls.loss(yh, y));
        net.backward(ls.gradient(yh, y));
        opt.update().clear_grads();
        net.gc();
    }
alpha.stat.save_zip(net, "weight");
alpha.stat.save_zip(opt, "opt_weight");
```

Figure 2. Using Alpha's high-level APIs to construct and train DNNs

types of devices. Alpha can utilize specific devices based on the appropriate underlying layers, while maintaining the stability of upper layers.

3.4. High-performance

Alpha utilizes GPUs through cu32, whose kernel functions are well-optimized. Some kernels are general solutions to ensure minimum performance thresholds, while others are designed to enhance effectiveness in specific contexts.

Alpha's async APIs enable the concurrent execution of operators, leading to enhanced parallelism and hardware utilization. Alpha can optimize static dataflows within dynamic dataflows, through inplace operators, fused operators, parallel execution of branches, and etc.

Alpha uses memory-pools to cache memory blocks, to minimize the overhead required for memory-allocation system-calls. Memory-pools work on the abstract *malloc* and *free* methods, which are implemented by lower-level components. A certain amount of pinned memory is managed to act as a cache between JVM and devices, to expedite data transmission through direct memory access. To save bandwidth, images are stored&transferred using *int8*, and converted to float datatypes at destination devices.

4. Architectures

As shown in Figure 3, Alpha is composed of 7 layers.

4.1. Native Libraries

Native libraries contain the most fundamental computing logic and strategies. Cu32 comprises 14 native dynamic-link-libraries. Unlike cuDNN (Chetlur et al., 2014), cu32 is entirely developed in C++ without ptx or sass code. Although this approach may not achieve the maximum hardware efficacy, it offers better portability, readability, and maintainability. Trading 10% speed for these benefits is acceptable; 100% is not. The code in cu32 is well-organized for better understanding. The computing strategies are mostly placed outside of cu32 to facilitate strategy adjustments. Cu32 has been tested on GTX1050, RTX3060ti, and RTX4090 GPUs, and achieves comparable performance to cuDNN in many cases.

4.2. EngineBase

EngineBase defines a set of computing primitives presented as abstract methods. Based on polymorphism, various subclasses of EngineBase can be implemented for specific devices. *CudaFloat32EngineBase* is an example of such subclasses, which serves as a higher-level architecture of cu32. This subclass maps Java methods to cu32 functions via JNI (Java native interface), and the computing strategies can be modified by configuring its properties.

4.3. EngineCore

EngineCore encapsulates the primitives of EngineBase, to shield hardware details from higher-level layers. It includes mechanisms like memory-pool, parameter-check, and

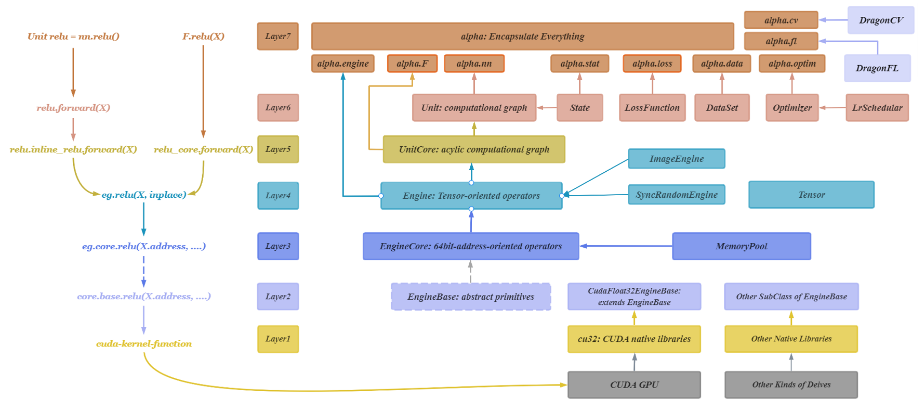


Figure 3. Alpha's 7-layer architecture

error-handling, where the memory-pool is abstract and customizable. EngineCore can work on specific devices via suitable EngineBases. For instance, It can perform *float32* operations on GPUs using *CudaFloat32EngineBase*.

4.4. Engine

Engine encapsulates EngineCore to enhance usability, by warping parameters and 64bit addresses in Tensor objects. The last dimension of Tensors is implicitly padded to 4*x*, to vectorize memory access using 128bit units. Tensors play a crucial role in automatic differentiation, and act as semaphores to coordinate operators. Tensors can be released manually or automatically by JVM (Java virtual machine).

'Sync' and 'check' properties act as the switches of async APIs and parameter-check. Typically, the variability of computational graphs in DNNs is limited and predictable. Therefore, checking parameters at the initial few batches is necessary; at every time could be wasteful. Once certain batches have been processed to ensure correctness, such 2 properties can be set to false, to enable async APIs and disable parameter-check, thereby improving efficiency.

ImageEngine provides operators for image processing and augmentation, with the ability to deal with hyper-spectrum images. SyncRandomEngine generates random numbers using 1 thread, whereas Engine uses multi-threads. These 2 methods may result in different quality of random numbers.

4.5. UnitCore

UnitCores are applications of Engines, which are used to construct the acyclic computational graphs of DNNs. Their public *forward* method establishes connections between UnitCores via callbacks, to represent the directed edges. The protected *backward* method is invoked by automatic differentiation. In this process, UnitCores gather gradients from their successors along the resolved edges, and then calculate gradients for their predecessors.

4.6. Unit

Units are built upon UnitCores, which have weights and hierarchical structures. A cyclic graphs formed by Units can be transformed to an acyclic graph composed of UnitCores. Compared to the garbage collection of JVM, the *gc* method of Unit is more cost-effective and timely for recycling resources. Units collaborate with DataSets, LossFunctions, and Optimizers to train DNNs. DataSet supports multi-thread data loading and preprocessing, which can run in parallel with DNN-execution through a buffer. DragonCV and DragonFL are developed for basic image and file processing.

4.7. Alpha

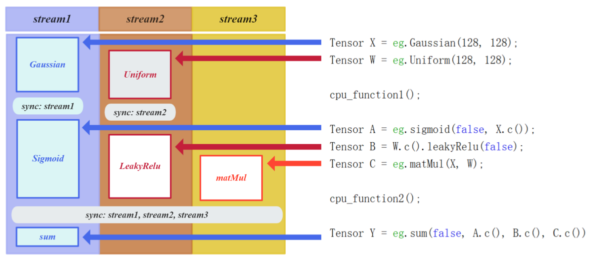
alpha packaged almost everything. The functionalities can easily accessed via keywords like *alpha.engine*, *alpha.nn*, *alpha.F*, *alpha.data*, *alpha.optim*, *alpha.loss*.

5. Techniques

5.1. Asynchronous APIs

Alpha's asynchronous (async) APIs enable the parallel execution of multi operators. For example, one CPU thread can launch many CUDA kernels without blocking. When executing operators on devices, JVM can undertake work such as generating logs and managing data structures. Async APIs are easy to use. Users only need to focus on the relation among operators, rather than underlying details.

When async APIs are activated, operators promptly return result Tensors, regardless of whether the computation has been completed. Each Tensor is associated with a Syncer, that can be triggered by *Tensor.c* method. Syncers act as semaphores to wait for the end of corresponding operators, and can be programmed to perform tasks like resource reclamation. Async APIs can be used to optimize dataflows, by processing unrelated multi branches simultaneously. An illustrative example is shown in Figure 4.



```
Tensor X = eg.Gaussian(128, 128);
Tensor W = eg.Uniform(128, 128);

cpu_function1();

Tensor A = eg.sigmoid(false, X.c());
Tensor B = W.c().leakyRelu(false);
Tensor C = eg.matMul(X, W);

cpu_function2();

Tensor Y = eg.sum(false, A.c(), B.c(), C.c())
```

Figure 4. Async APIs enable the concurrent execution of multi operators alongside CPU instructions. At most 3 operators can be executed simultaneously in 3 CUDA streams. 2 CPU functions are performed in parallel with the GPU.

5.2. C-K-S Algorithms: skip zeros in convolution

Zero items (0s) are commonly included in convolutional operators, such as 0-padding on feature-maps, and 0-insertion in deconvolution and dilated convolution. These 0s cause unnecessary calculations and strain on hardware.

To skip these 0s, C-K-S algorithm is proposed, including ConvV2, KS-deconv, and Sk-dilated. ConvV2 excludes padded 0s, providing a constant factor speedup; KS-deconv and SK-dilated transform sparse tensors to dense ones, accelerating ND deconvolution and dilated convolution by $stride^N$ and $dilate^N$ times. C-K-S is based on math and not reliant on specific systems. Its operations have minimal interdependence to comply with the nature of SIMD.

KS-deconv is designed for non-unit-stride deconvolution, to bypass $(stride - 1)$ 0s inserted between adjacent items of input-feature-maps (*ifms*). As shown in Figure 5, KS-deconv has 3 stages: the filters are split into $stride^N$ parts; each part is used to conduct stride-1 convolution on the corresponding subset of *ifms*; the resulting outputs are composed to obtain the output-feature-maps (*ofms*). This kernel-split idea is based on 2 observations. Firstly, among the patches of *ifms*, the 0-distributions can be categorized

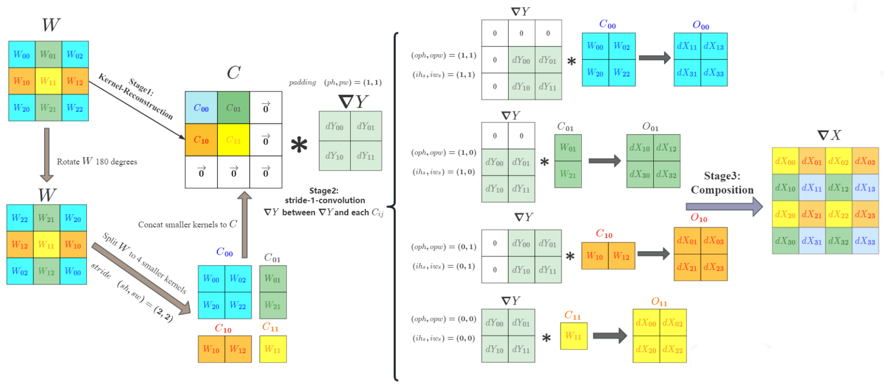

Stage1: Kernel-Reconstruction
Rotate W 180 degrees
Split W to 4 smaller kernels stride $(sh, sw) = (2, 2)$
Concat smaller kernels to C
Stage2: stride-1-convolution ∇Y between ∇Y and each C_{ij}
Stage3: Composition

Figure 5. KS-deconv in backpropagation of 2D conv-layers. In stage1, $W(3 \times 3)$ is rotated 180 degrees, and split to $C_{00}(2 \times 2)$, $C_{01}(2 \times 1)$, $C_{10}(1 \times 2)$ and $C_{11}(1 \times 1)$. C_{00-11} are concatenated to $C(2 \times 2 \times 2 \times 2)$ which has continuous memory. In stage2, C_{00-11} are respectively used to perform stride-1 convolution with ∇Y, and the outputs are O_{00}, O_{01}, O_{10} and O_{11}. In stage3, O_{00-11} are composed to obtain ∇X.

into $stride^N$ classes, which can be distinguished by the $(coordinates\%stride)$ of these patches. Secondly, each item in *ofms* can be calculated, by performing a dot-product on two nonzero segmentations where are respectively derived from the filters and *ifms*.

Sk-dilated is used for dilated convolution, where sparse filters have $(dilate - 1)$ 0s placed between adjacent items. In each filter, except for the $channel$ and $batch$ axes, the coordinates of nonzero items must be integral multiples of $dilate$. Adhering to this rule, Sk-dilated does not add 0s to filters. Instead, for each dot-product of dilated convolution, it fetches items in filters with unit step-size, and selects items in *ifms* with steps-size of $dilate$.

ConvV2 utilizes filter-trimming to exclude padded 0s at the boundary of *ifms*, thereby enabling access only to nonzero elements in the central region. This technique is achieved by moving pointers and constraining memory access, without auxiliary workspace. Filter-trimming's efficacy is positively correlated to the size of 0-padding, so it usually plays a greater role on small feature-maps.

Filter-trimming has been integrated in KS-deconv and Sk-dilated, to develop their V2 version. As illustrated in Figure 6, KS-deconv-V2 are compared with PyTorch-1.12 in backpropagation on RTX 3060ti. As the feature-maps become smaller, the speed of KS-deconv-V2 increases.

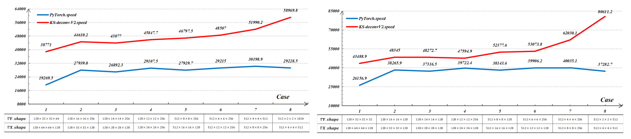


Figure 6. Compare KS-deconv-V2 with PyTorch. The stride is 2, and the tensors are represented in *NHWC* format. On the left side, the filter size is (3×3) with a padding of 1; on the right side, the filer size is (5×5) with a padding of 2.

5.3. Im2col-Winograd

Winograd (Lavin & S.Gray, 2016) has been extensively used to accelerate convolutions. When using filters of size r to calculate n outputs, Winograd $F(n, r)$ needs $(n + r - 1)$ multiplications, less than $(n * r)$ multiplications required by standard convolution, thereby improving efficiency.

Previous studies have implemented Winograd on GPUs to accelerate 2D convolution (Castro et al., 2021; Chetlur et al., 2014; Yan et al., 2020; Yang & Lai, 2021). These studies mainly use the 2D variant of Winograd, and arrange tensors in *NCHW* or *CHWN* format. The fused-Winograd2D is limited to (3×3) filters, while the non-fused has been applied to other filter sizes but requires a large amount of memory to store intermediate variables. *NHWC* is a widely used tensor format, but it is not well-suited for Winograd2D due to its discontinuous memory access, which reduces the hit ratio of GPU L2-cache.

To implement a versatile and flexible fused-Winograd on GPUs for *NHWC* format, we propose Im2col-Winograd. As shown in Figure 8, Im2col-Winograd has 2 stages: lower the filters and feature-maps to 2D matrices through Im2col; cumulatively apply 1D Winograd on these matrices. The 2 stages are integrated into a single operator, eliminating the need for auxiliary memory.

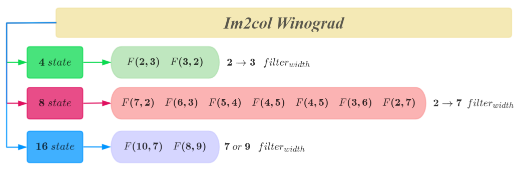

Im2col Winograd
4 state
$F(2,3)$ $F(3,2)$
$2 \to 3$ $filter_{width}$
8 state
$F(7,2)$ $F(6,3)$ $F(5,4)$ $F(4,5)$ $F(4,5)$ $F(3,6)$ $F(2,7)$
$2 \to 7$ $filter_{width}$
16 state
$F(10,7)$ $F(8,9)$
7 or 9 $filter_{width}$

Figure 7. Im2col-Winograd with 4/8/16 states.

$F(n, r)$ uses $\alpha = (n + r - 1)$ variables to get n outputs, so it can be denoted as α state Winograd. Fused-Winograd loads, transforms, and stores the data in shared-memory, where the max shared-memory size for 1 block is 49152 bytes. To hide memory latency and maximize item-reuse, GPUs typically perform $8 * (8 \times 8)$ outer products via 256 threads within a block. To satisfy such outer products, the state α must ≤ 16, and ideally a power of 2. $F(2 \times 2, 3 \times 3)$ for (3×3) filters has 16 states, exactly meeting the upper limit.

As shown in Figure 7, 4/8/16 state Im2col-Winograd has

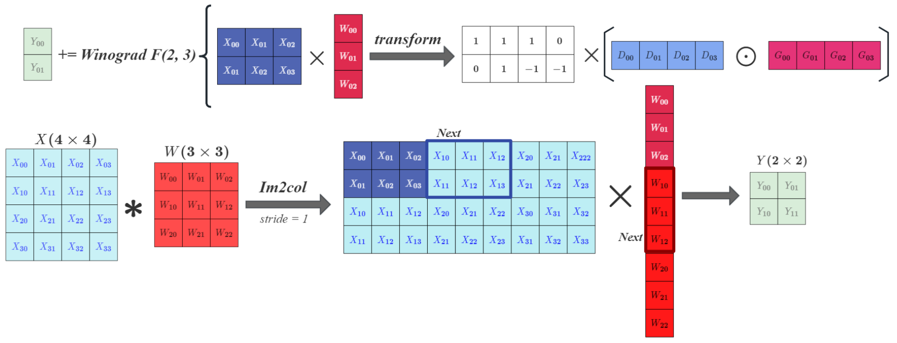


Figure 8. Img2col-Winograd $F(2,3)$. The filter W and input-feature-map X are converted to 2D matrices through Im2col. The items selected by sliding windows are used to perform $F(2,3)$, and the results are accumulated to the output-feature-map Y.

been implemented to cover 2-9 filter-sizes for both the forward and backward propagation. Compared to GEMM convolution, the 4/8 state Im2col-Winograd exhibits almost no loss of precision, whereas the 16-state version yields a relative difference of 10^{-4}. Both the 8 and 16 state versions can be manually enabled or disabled.

Im2col-Winograd is more flexible and lightweight than Winograd2D. Unlike Winograd2D, Im2col-Winograd only imposes constraints on one dimension of filters, as opposed to two. In some cases, Im2col-Winograd requires fewer resources than Winograd2D, to achieve the same acceleration. For instance, both $F(2\times 2, 3\times 3)$ and $F(6,3)$ theoretically reduce the multiplication to $1/2.25$. However, the latter needs 8 states and fetches 33/6 items to calculate an output, while the former uses 16 states and selects 25/4 items. Moreover, Im2col-Winograd can be used for ND convolution, by extending Im2col from 2D to ND.

Im2col-Winograd has been compared with cuDNN-8.9 (Chetlur et al., 2014) on RTX3060ti GPU. As shown in Figure 9, Im2col-Winograd especially its 16-state version is more efficient than cuDNN in many cases.

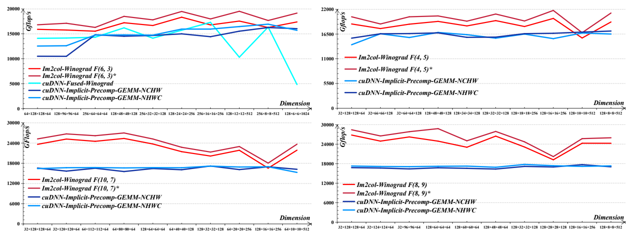


Figure 9. Compare Im2col-Winograd with cuDNN. $F(6,3)$, $F(4,5)$, $F(10,7)$ and $F(8,9)$ are performed with (3×3), (5×5), (7×7) and (9×9) filters respectively. The tensors are represented in *NHWC* format. In forward propagation, filters are transposed from *NHWC* to *HWCN* format with a small expense to improve bandwidth. '*' means ignoring the time of filter-transposition.

5.4. Parallel Random Number Generation

Cu32 generates pseudo-random numbers via multi threads. Each thread performs linear congruence on global seeds to obtain its local seeds, and then conducts linear congruence on local seeds to produce random numbers. Users can specify global seeds or have them randomly generated.

The quality of random numbers used to initialize DNNs significantly impacts the convergence. We have found 2 factors that affect such quality: the volume of random numbers generated by individual threads, and the degree of randomness introduced in local seeds. Figure 10 illustrates the influence of these 2 factors on the convergence of ResNet18 on Cifar10.

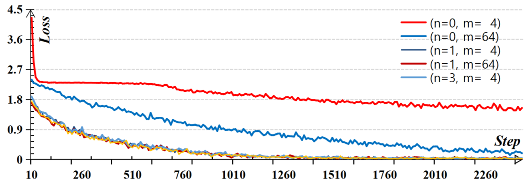
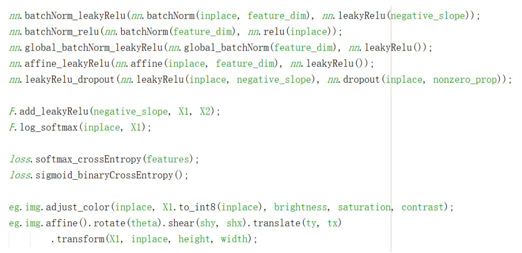


Figure 10. The quality of random numbers impacts the convergence. (n,m) means performing n times linear-congruence on global seeds, and producing m random numbers per thread

5.5. Inplace Operators and Fused Operator

In DNNs, certain operators have predetermined combination orders, and therefore can be fused or inplace to save memory and bandwidth. Furthermore, compared to DL frameworks that process the code, programmers often have more prior knowledge of the DNNs. This enables them to undertake certain optimizations on the computational graphs using inplace or fused operators. Alpha provides inplace and fused operators for users, as demonstrated in Figure 11.



```
nn.batchNorm_leakyRelu(nn.batchNorm(inplace, feature_dim), nn.leakyRelu(negative_slope));
nn.batchNorm_relu(nn.batchNorm(feature_dim), nn.relu(inplace));
nn.global_batchNorm_leakyRelu(nn.global_batchNorm(feature_dim), nn.leakyRelu());
nn.affine_leakyRelu(nn.affine(inplace, feature_dim), nn.leakyRelu());
nn.leakyRelu_dropout(nn.leakyRelu(inplace, negative_slope), nn.dropout(inplace, nonzero_prop));

F.add_leakyRelu(negative_slope, X1, X2);
F.log_softmax(inplace, X1);

loss.softmax_crossEntropy(features);
loss.sigmoid_binaryCrossEntropy();

eg.img.adjust_color(inplace, X1.to_int8(inplace), brightness, saturation, contrast);
eg.img.affine().rotate(theta).shear(shy, shx).translate(ty, tx)
        .transform(X1, inplace, height, width);
```

Figure 11. The inplace and fused operators APIs of Alpha.

5.6. Accuracy Optimizations

For reduction operators, Kahan-summation is used to reduce errors. Besides, 'minimizing float operations' and 'avoiding numerical overflow' can be balanced. For example, when computing the mean of collection $\{x_1, x_2, ..., x_n\}$, $\frac{1}{N}\Sigma x_i$ has better precision than $\Sigma\frac{1}{N}x_i$

due its fewer float operations, but it takes higher risk of numerical overflow. To compromise between these 2 methods, the collection can be divided into M segments, and the sum of each segment S_i can be calculated to determine the average $\frac{1}{N}\Sigma S_i$.

GELU (Hendrycks & Gimpe, 2016) is commonly estimated as $0.5(1+tanh[\sqrt{\pi/2}(x+0.044715x^3)])$, with its derivative $\frac{1}{1+e^u}(1-\frac{e^u(u-0.14271x^3)}{1+e^u})|_{u=-(1.59577x+0.214064x^3)}$. However, e^{-u} may yield *inf* leading to *NaN* derivative, when $x<0$. To avoid this, the derivative can be calculated as $\frac{e^{-u}}{1+e^{-u}}(1-\frac{e^u(u-0.14271x^3)}{1+e^{-u}})$.

6. Experiments

To evaluate Alpha's performance, certain comparisons with PyTorch&cuDNN have been conducted on Cifar10 and ILSVRC2012. PyTorch version is 1.12, CUDA version is 11.5, and the version of Alpha&cu32 is 1.1 or 1.2.

6.1. Methods and Conditions

Several kinds of optimizers (Rumehart et al., 1986; Tieleman & Hinton, 2012; Kingma & Ba, 2015; Loshchilov & Hutter, 2017; Liu et al., 2019) were utilized to train DNNs (Szegedy et al., 2015; Simonyan & Zisserman, 2015; He et al., 2016; Hu et al., 2018), with softmax and 0.001 learning-rate on Cifar10 and ILSVRC2012:

- Cifar10: the input shape is (32×32), the label space is 10, the batchsize is 512, and the data was processed on an RTX3060ti GPU with an i5-12490 CPU.
- ILSVRC2012: the input shape is (128×128), the label space is 1000, the batchsize is 256, and the data was processed on an RTX4090 with an i9-13900KF CPU. Alpha uses 16 threads to load and preprocess data, while PyTorch utilizes 4 workers, which is the optimal configuration we have observed.

DNNs of Alpha and PyTorch were identical, and underwent the same initialization, training, and testing procedures. The activation function is LeakyRelu (Maas et al., 2013). Specific convolutional and full-connect layers were adjusted to accommodate the tensor shape, while the backbone of DNNs remains unaltered. BatchNorm (Ioffe & Szegedy, 2015) was integrated into VGG and GoogLeNet, to prevent gradient-vanishing and expedite convergence. Full-connect and convolutional layers were initialized using kaiming-uniform (He et al., 2015).

The labels were encoded to one-hot formats, and the pixel values were linearly scaled to fall within $[-1, 1]$. The loss-function value was recorded every 10 steps. To plot the loss curves of ILSVRC2012, a sliding window of length 10 was used to average the loss-function values without overlap.

6.2. Results and Discussions

Table 1-2 present the performance of Alpha and PyTorch. The loss curves are shown in figure 12-35 in ***Appendix***.

Table 1. Performance on Cifar10. PyTorch's data is blue. Alpha's data is red. The speed and acceleration of Alpha-1.2 is dark red.

Network	*Training*	*Speed*	*Acceleration*	*Train\Test accuracy*	*GPU memory*	*Reduction*	*GPU utilization*	*CPU utilization*	*Weight file*
GoogLeNet	*Adam* 30 epoch	**9.147 8.674** s/epoch **13.022** s/epoch	**1.423x 1.501x**	97.82% 79.78% 97.07% 79.57%	**2121** MB **4722** MB	0.4491x	90% 183 W 98% 183 W	18.0% 1067 MB 12.8% 3814 MB	32.6 MB 23.8 MB
	SGDM 40 epoch	**9.125 8.665** s/epoch **12.963** s/epoch	**1.421x 1.496x**	92.86% 63.72% 90.87% 61.56%	**2089** MB **4708** MB	0.4437x			
ResNet18	*Adam* 25 epoch	**5.822 5.332** s/epoch **7.312** s/epoch	**1.256x 1.371x**	99.09% 78.11% 98.90% 77.90%	**1067** MB **2486** MB	0.4292x	92% 197 W 98% 162 W	13.6% 1138 MB 12.6% 3812 MB	66.7 MB 48.2 MB
	SGDM 35 epoch	**5.794 5.284** s/epoch **7.158** s/epoch	**1.235x 1.354x**	100.0% 60.60% 99.83% 61.24%	**1010** MB **2464** MB	0.4099.x			
ResNet34	*Adam* 30 epoch	**11.528 10.464** s/epoch **14.149** s/epoch	**1.227x 1.352x**	99.01% 79.45% 98.87% 79.16%	**1685** MB **3122** MB	0.5397x	92% 198 W 98% 165 W	13.6% 1188 MB 12.6% 3813 MB	120 MB 87.3 MB
	SGDM 35 epoch	**11.471 10.269** s/epoch **13.788** s/epoch	**1.202x 1.343x**	100.0% 60.60% 99.83% 61.24%	**1572** MB **3116** MB	0.5045x			
VGG16	*Adam* 35 epoch	**9.709 8.565** s/epoch **11.041** s/epoch	**1.137x 1.289x**	97.91% 82.75% 97.59% 82.62%	**1634** MB **3675** MB	0.4466x	97% 197 W 99% 180 W	12.9% 1007 MB 13.2% 3814 MB	78.7 MB 56.7 MB
	SGDM 35 epoch	**9.698 8.493** s/epoch **10.871** s/epoch	**1.121x 1.279x**	100.0% 75.90% 100.0% 75.96%	**1569** MB **3658** MB	0.4289x			
VGG19	*Adam* 40 epoch	**11.849 10.389** s/epoch **13.142** s/epoch	**1.109x 1.265x**	96.06% 81.13% 96.03% 80.98%	**1786** MB **3755** MB	0.4756x	97% 197 W 99% 181 W	12.7% 1042 MB 13.2% 3817 MB	106 MB 77 MB
	SGDM 40 epoch	**11.802 10.268** s/epoch **12.964** s/epoch	**1.098x 1.262x**	99.94% 76.84% 99.60% 76.56%	**1694** MB **3740** MB	0.4529x			
SENet	*Adam* 30 epoch	**25.767 25.288** s/epoch **29.317** s/epoch	**1.138x 1.159x**	98.65% 81.88% 98.60% 81.46%	**5424 MB** **6947 MB**	0.7808x	94% 197 W 99% 187 W	13.1% 1301 MB 12.3% 3816 MB	207 MB 150 MB
	SGDM 35 epoch	**25.718 25.267** s/epoch **28.820** s/epoch	**1.121x 1.141x**	100.0% 63.74% 100.0% 63.82%	**5261**MB **6928**MB	0.7594x			

Table 2. Performance on ILSVRC2012. PyTorch's data is blue. Alpha's data is red. The speed and acceleration of Alpha-1.2 is dark red.

Network	*Training*	*Speed*	*Acceleration*	*Train set accuracy*	*GPU memory*	*Reduction*	*GPU utilization*	*CPU utilization*	*Weight file*
ResNet18	*Adam* 50 epoch	**664.957 - 627.675** s/epoch **1202.894** s/epoch	**1.809x 1.916x**	99.09% 98.90%	**5718** MB **11848** MB	0.4826x	92% 388 W 96% 368 W	61.9% 5360 MB 24.7% 7321 MB	66.8 MB 50.9 MB
	RMSprop *40* epoch	**661.992 625.202** s/epoch **1185.132** s/epoch	**1.790x 1.896x**	97.28% 97.53%	**5656** MB **11744** MB	0.4742x			
ResNet34	*Adam* 50 epoch	**1264.301 1132.489** s/epoch **2219.791** s/epoch	**1.756x 1.960x**	99.01% 98.87%	**9090** MB **15384** MB	0.5909x	95% 405 W 99% 370 W	36.5% 5180 MB 23.8% 7252 MB	124 MB 89.8 MB
	SGDM 60 epoch	**1233.389 1126.331** s/epoch **2205.236** s/epoch	**1.788x 1.958x**	97.99% 95.63%	**8991** MB **15038** MB	0.5979x			
ResNet50	*AdamW* 60 epoch	**989.484 975.828** s/epoch **1472.867** s/epoch	**1.489x 1.509x**	88.50% 90.36%	**9980** MB **12896** MB	0.7739x	91% 389 W 96% 371 W	45.1% 5406 MB 20.6% 7357 MB	264 MB 199 MB
	Adam 50 epoch	**994.888 971.652** s/epoch **1458.453** s/epoch	**1.466x 1.501x**	96.85% 96.51%	**9980** MB **12896** MB	0.7739x			
VGG16	*Adam* 30 epoch	**1087.803 941.921** s/epoch **1115.235** s/epoch	**1.025x 1.184x**	97.94% 97.65%	**10870** MB **14240** MB	0.7633x	96% 402 W 97% 366 W	40.3% 5106 MB 23.3% 7294 MB	294 MB 224 MB
	SGDM 40 epoch	**1086.935 932.007** s/epoch **1114.341** s/epoch	**1.025x 1.196x**	93.72% 94.61%	**10653** MB **13564** MB	0.7854x			
VGG19	*Adam* 40 epoch	**1217.609 1033.081** s/epoch **1246.722** s/epoch	**1.024x 1.207x**	97.85% 97.42%	**11138** MB **14438** MB	0.7714x	98% 405 W 97% 370 W	38.7% 5031 MB 23.1% 7460 MB	319 MB 244 MB
	SGDM 40 epoch	**1209.913 1016.869** s/epoch **1229.833** s/epoch	**1.016x 1.209x**	97.30% 97.53%	**10880** MB **13766** MB	0.7903x			
SENet	*RAdam* 60 epoch	**910.079 902.903** s/epoch **1346.629** s/epoch	**1.480x 1.491x**	96.78% 96.88%	**10268** MB **12242** MB	0.8384x	90% 355 W 94% 357 W	46.7% 5627 MB 21.4% 7301 MB	208 MB 152 MB
	Adam 60 epoch	**916.739 903.387**s/epoch **1335.822** s/epoch	**1.457x 1.479x**	97.30% 96.25%	**10268** MB **12242** MB	0.8384x			

On both datasets, DNNs trained by Alpha or PyTorch show comparable convergence, accuracy, and efficiency. In terms of speed and memory-usage, Alpha outperforms PyTorch in certain scenarios.

Several factors contribute to Alpha's higher speed compared to PyTorch. Cu32 provides efficient GPU operators, while ensuring correctness and accuracy. The C-K-S algorithm avoids redundant 0-calculations in conv-layers, reducing time complexity and strain on hardware. Specifically, KS-deconv and Sk-dilated significantly simplify the backpropagation of conv-layers with non-unit stride, thereby accelerating the execution of ResNet and SENet. Im2col-Winograd reduces the cost of stride-1 conv-layers, especially those with (3×3) or (5×5) filters. The concurrent execution offered by async APIs enhances the power of GPUs, and conceals the execution of certain CPU functions. The fusion of operators reduces memory access, lowering the computational cost.

Alpha's lower memory-usage compared to PyTorch can be attributed to several reasons. Alpha offers a wider range of inplace and fused operators than PyTorch; and it tries to calculate gradients directly within the memory space of preceding gradients, thereby reducing the need for memory space to store intermediate variables. In both forward and backward propagation, Tensors that are no longer required can be released based on Syncers, thus facilitating the recycling of memory blocks. The experiments mentioned in this paper were carried out using the basic memory pool, and it is possible to further decrease memory usage by configuring a more advanced memory-pool to EngineCore.

GPU-utilization is evaluated using 2 metrics: power and time-slice utilization-rate (UT). Alpha and PyTorch employ distinct methods to scheduling operators, each offering its own advantages. In PyTorch, operators are launched by a single thread, and typically executed in 1 CUDA stream based on a first-in-first-out order. This approach leads to higher UT due to less overhead of synchronization between operators. Conversely, in Alpha, operators are managed via multi-threads, and executed in multiple CUDA streams concurrently, enabling higher power. However, this approach requires heavier synchronization mechanisms, leading to a lower UT.

CPU-utilization is evaluated by 2 metrics: memory-usage and time-slice utilization-rate (UT). The memory-usage is primarily used to indicate the absence of memory leaks, as it can be influenced by the configuration of the Java or Python virtual machine. When training DNNs on ILSVRC2012, Alpha and PyTorch achieve nearly 100% GPU utilization, suggesting that the speed bottleneck is GPU-computing rather than the data loading and preprocessing on CPU. Notably, Alpha's 16 threads have higher parallelism than PyTorch's 4 workers, and therefore result in a higher UT.

Overall, these experimental results are evidence supporting the correctness and efficiency of Alpha&cu32.

7. Conclusion

This paper introduces the background, characteristics, architectures, and techniques of Alpha&cu32. Besides, it demonstrates their correctness and effectiveness, by benchmarking them against PyTorch&cuDNN on Cifar10 and ILSVRC2012.

Alpha tries to raise Java's effectiveness in DL field. It adopts some innovative designs to ensure user-friendliness, and realizes high-performance through cu32 and specific techniques. Based on its multi-layer architecture and the adaptability to Java big-data ecosystem, Alpha achieves scalability and is capable of aggregating heterogeneous computing resources.

The source code for Alpha is available for access at {the link is hidden due to double-blind review}. Subsequent updates will be aligned with our future works.

Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here

References

Abadi, M., Agarwal, A., Barham, P., and et al. TensorFlow: Large-scale machine Learning on heterogeneous system. *arXiv e-prints*, 2016. URL https://arxiv.org/abs/1603.04467v2.

Castro, R. L., Andrade, D., and Fraguela, B. B. OpenCNN: A Winograd Minimal Filtering Algorithm Implementation in CUDA. *Mathematics*, 9(17), 2021.

Chetlur, S., Woolley, C., Vandermersc, P., and et. al. cuDNN: Efficient Primitives for Deep Learning. *CoRR*, abs/1410.0759, 2014, 2014.

Chollet, F. Keras: Deep Learning for humans. URL https://keras.i.

Cutting, D., Cafarella, M., and et. al. Hadoop. URL https://hadoop.apache.org/.

Ding, J., Ren, X., Luo, R., and Sun, X. An Adaptive and Momental Bound Method for Stochastic Learning. *arXiv e-prints*, 2019. URL https://arxiv.org/abs/1910.12249v1.

Feenster, A., Henkelmann, C., Bamberg, E., and et. al. Deep Java Library. URL https://djl.ai/.

Gibson, A., Nicholson, C., Patterson, J., and et al. Deeplearning4j: Distributed, open-source deep learning for Java and Scala on Hadoop and Spark. 2016. doi: 10.6084/M9.FIGSHARE.3362644.

Goodfellow, I., Bengio, Y., and Courville, A. *Deep Learning*. MIT Press, 2016. URL http://www.deeplearningbook.org.

He, K., Zhang, X., Ren, S., and Sun, J. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet. In *Proceedings of International Conference on Machine Learning (ICML)*, pp. 448–456, 2015.

He, K., Zhang, X., Ren, S., and Sunn, J. Deep residual learning for image recognition. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016. doi: 10.1109/CVPR.2016.90.

Hendrycks, D. and Gimpe, K. Gaussian Error Linear Units (GELUs). *arXiv e-prints*, 2016. URL https://arxiv.org/abs/1606.08415v5.

Hu, J., Shen, L., and Sun, G. Squeeze-and-Excitation Networks. In *Proceedings of IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 7132–7141, 2018. doi: 10.1109/CVPR.2018.00745.

Ioffe, S. and Szegedy, C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proceedings of International Conference on Machine Learning (ICML)*, pp. 448–456, 2015.

Jia, Y., Shelhamer, E., Donahue, J., and et al. caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia Retrieval (ICMR)*, pp. 675–678, Newark, NJ, USA, 2014.

Kingma, D. P. and Ba, J. L. Adam: a Method for Stochastic Optimization. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, San Diego, USA, 2015.

Krizhevsky, A. Cifar10, a. URL http://www.cs.toronto.edu/~kriz/cifar.html.

Krizhevsky, A. cuda-convnet2, b. URL https://code.google.com/p/cuda-convnet2.

Krizhevsky, A., Sutskever, I., and Hinto, G. E. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 26th Annual Conference on Neural Information Processing Systems (NIPS)*, pp. 1097–1105, 2012.

Lavin, A. and S.Gray. Fast algorithms for convolutional neural networks. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, USA, 2016. doi: 10.1109/CVPR.2016.435.

Li, F. IMAGENET. URL https://image-net.org.

Liu, L., Jiang, H., He, P., and et. al. On the Variance of the Adaptive Learning Rate and Beyond. *arXiv e-prints*, 2019. URL https://arxiv.org/abs/1908.03265v1.

Loshchilov, H. and Hutter, F. Decoupled Weight decay regularization. *arXiv e-prints*, 2017. URL https://arxiv.org/abs/1711.05101v1.

Maas, A. L., Y, A., Hannun, and Ng, A. Y. Rectifier Nonlinearities Improve Neural Network Acoustic Models. In *Proceedings of International Conference on Machine Learning (ICML)*, 2013.

NVIDIA. cuBLAS. URL https://developer.nvidia.cn/cublas.

NVIDIA. *CUDA C Best Practices Guide*. NVIDIA, 2023a. URL https://docs.nvidia.com/cuda/cuda-c-best-practices-guide.

NVIDIA. *CUDA C Programming Guide*. NVIDIA, 2023b. URL https://docs.nvidia.com/cuda/cuda-c-programming-guide.

Paszke, A., Gross, S., F.Massa, and et.al. Spark: Cluster Computing with Working Sets. *USENIX Association*, 2010.

Paszke, A., Gross, S., Massa, F., and et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proceedings of the 33rd Annual Conference on Neural Information Processing Systems (NIPS)*, pp. 675–678, Vancouver, Canada, 2019.

Rumehart, D. E., Hinton, G. E., and et al. Learning Representations by Back-Propagating errors. *Nature*, 323 (6088):533–536, 1986. doi: 10.1038/323533a.

Simonyan, K. and Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, 2015.

Srivastava, N., Hinton, G. E., Krizhevsky, A., and et al. Dropout: A Simple Way to Prevent Neural Network from Overfitting. *Journal of Machine Learning Research*, 15 (1):1929–1958, 2014.

Szegedy, C., Liu, W., Jia, Y., and et al. Going deeper with convolutions. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, 2015. doi: 10.1109/CVPR.2015.7298594.

Tieleman, T. and Hinton, G. RMSprop. *Neural Networks for Machine Learning*, 2012.

Tokui, S., Oono, K., Hido, S., and Clayton, J. Chainer: a next-generation open-source framework for deep learning. In *Proceedings of the 29th Annual Conference on Neural Information Processing Systems (NIPS)*, Montréal Canada, 2015.

Yan, D., Wang, W., and Chu, X. Optimizing Batched Winograd Convolution on GPUs. In *Proceedings of the 25th ACM Symposium on Principles and Practice of Parallel Programming (SIGPLAN)*, San Diego, CA, USA, 2020.

Yang, J. L. D. and Lai, J. Optimizing Winograd-Based Convolution with Tensor Cores. In *Proceedings of the 50th International Conference on Parallel Processing*, 2021.

Appendix

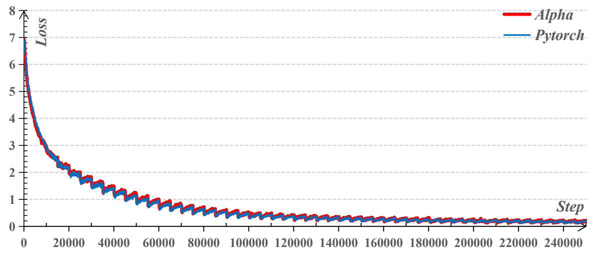


Figure 12. ILSVRC2012: ResNet18 + Adam, 50 epoch

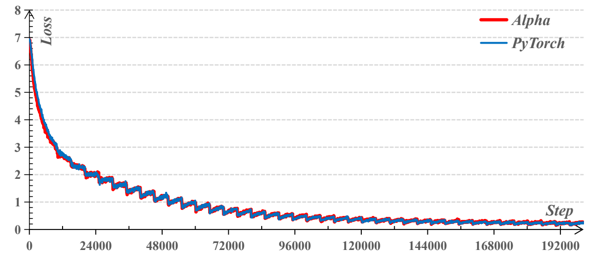


Figure 13. ILSVRC2012: ResNet18 + RMSprop, 40 epoch

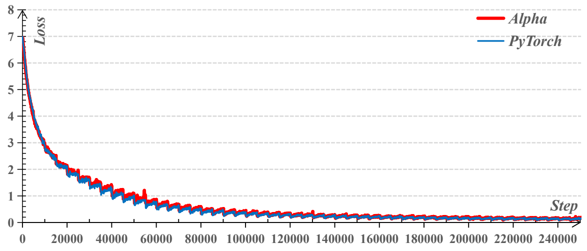


Figure 14. ILSVRC2012: ResNet34 + Adam, 50 epoch

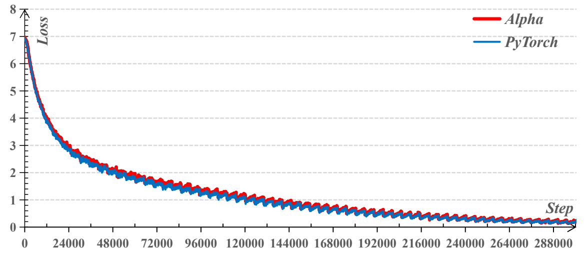


Figure 15. ILSVRC2012: ResNet34 + SGDM, 60 epoch

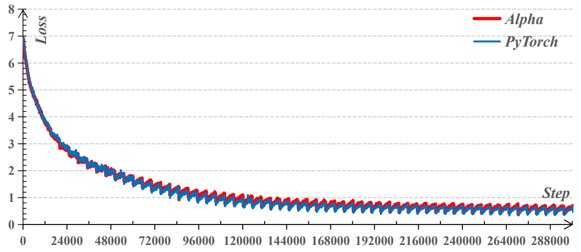


Figure 16. ILSVRC2012: ResNet50 + AdamW, 60 epoch

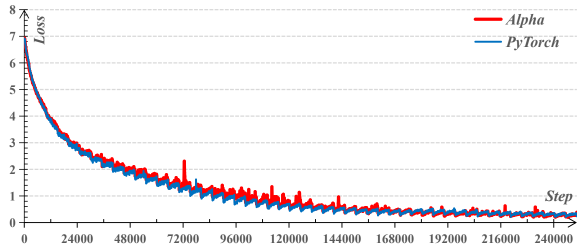


Figure 17. ILSVRC2012: ResNet50 + Adam, 50 epoch

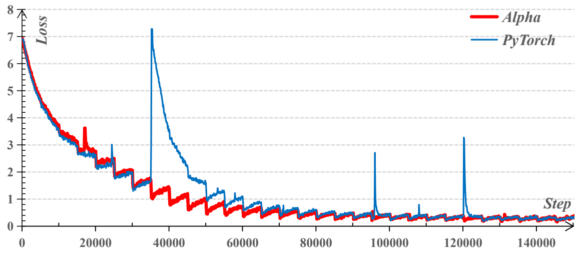


Figure 18. ILSVRC2012: VGG16 + Adam, 30 epoch

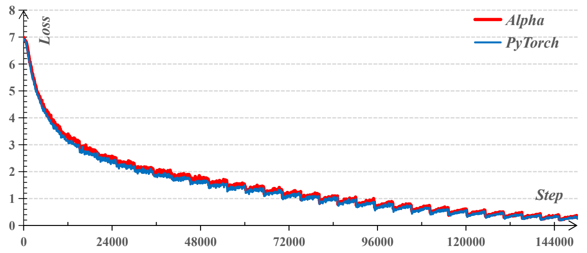


Figure 19. ILSVRC2012: VGG16 + SGDM, 30 epoch

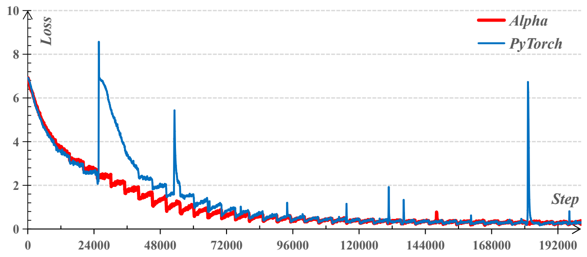


Figure 20. ILSVRC2012: VGG19 + Adam, 40 epoch

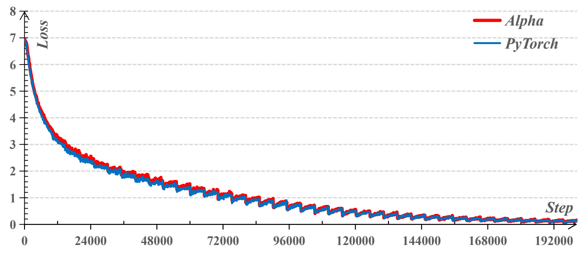


Figure 21. ILSVRC2012: VGG19 + SGDM, 40 epoch

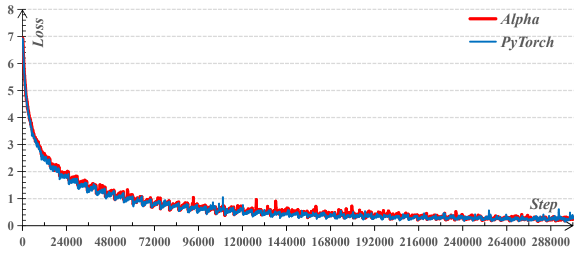


Figure 22. ILSVRC2012: SENet + RAdam, 60 epoch

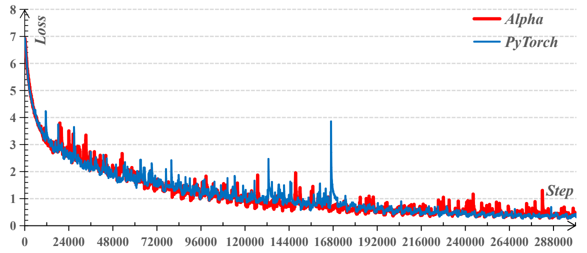


Figure 23. ILSVRC2012: SENet + Adam, 60 epoch

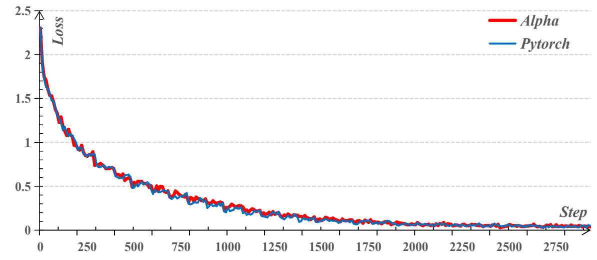


Figure 24. Cifar10: GoogLeNet + Adam, 30 epochs

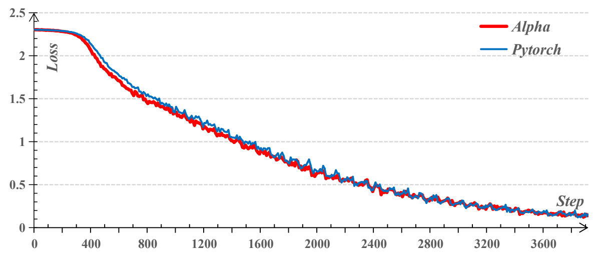


Figure 25. Cifar10: GoogLeNet + SGDM, 40 epoch

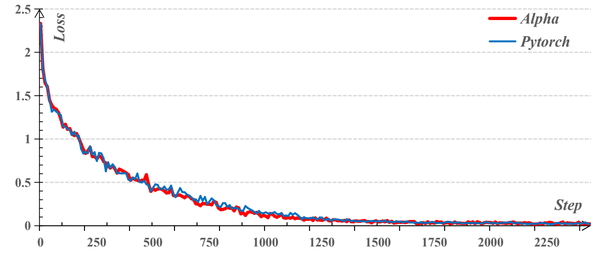


Figure 26. Cifar10: ResNet18 + Adam, 25 epoch

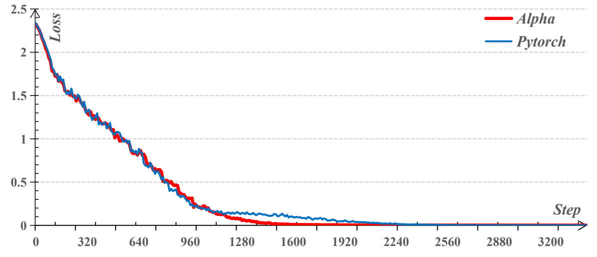


Figure 27. Cifar10: ResNet18 + SGDM, 35 epoch

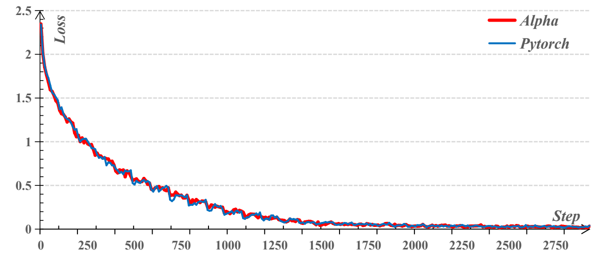


Figure 28. Cifar10: ResNet34 + Adam, 30 epoch

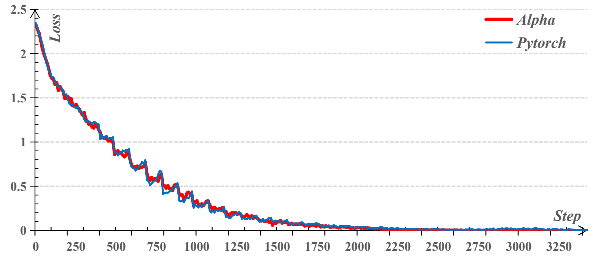


Figure 29. Cifar10: ResNet34 + SGDM, 35 epoch

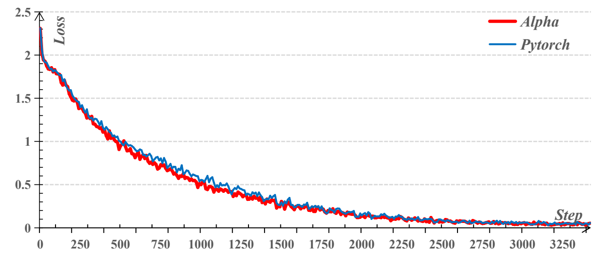


Figure 30. Cifar10: VGG16 + Adam, 35 epoch

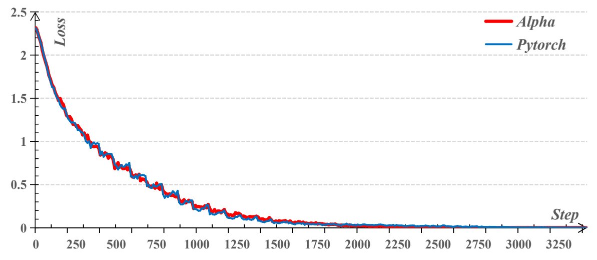


Figure 31. Cifar10: VGG16 + SGDM, 35 epoch

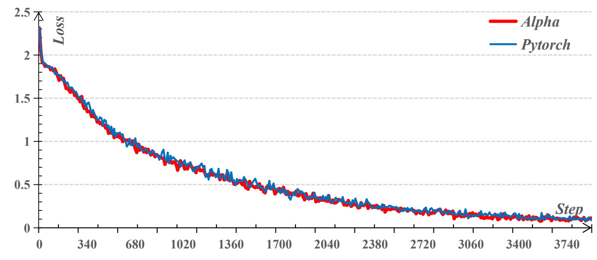


Figure 32. Cifar10: VGG19 + Adam, 40 epoch

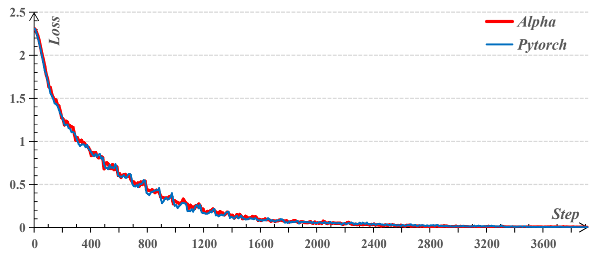


Figure 33. Cifar10: VGG19 + SGDM, 40 epoch

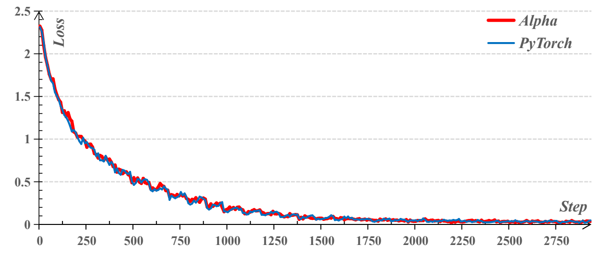


Figure 34. Cifar10: SENet + Adam, 30 epoch

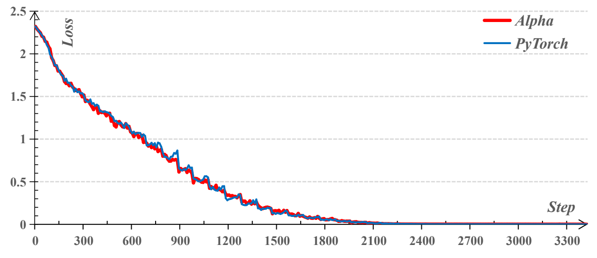


Figure 35. Cifar10: SENet + SGDM, 35 epoch