A Framework to Enable Algorithmic Design Choice Exploration in *DNNs*

Timothy L. Cronin IV, Sanmukh Kuppannagari
Department of Computer and Data Sciences, Case Western Reserve University
Contact: {tlc107, sxk1942}@case.edu

***Abstract*—Deep learning technologies, particularly deep neural networks (*DNNs*), have demonstrated significant success across many domains. This success has been accompanied by substantial advancements and innovations in the algorithms behind the operations required by *DNNs*. These enhanced algorithms hold the potential to greatly increase the performance of *DNNs*. However, discovering the best performing algorithm for a *DNN* and altering the *DNN* to use such algorithm is a difficult and time consuming task. To address this, we introduce an open source framework which provides easy to use fine grain algorithmic control for *DNNs*, enabling algorithmic exploration and selection. Along with built-in high performance implementations of common deep learning operations, the framework enables users to implement and select their own algorithms to be utilized by the *DNN*. The framework's built-in accelerated implementations are shown to yield outputs equivalent to and exhibit similar performance as implementations in *PyTorch*, a popular *DNN* framework. Moreover, the framework incurs no additional performance overhead, meaning that performance depends solely on the algorithms chosen by the user.**

***Index Terms*—Deep Neural Network, Algorithmic Design Choices, PyTorch**

I. Introduction

Artificial Intelligence (*AI*) and more specifically deep learning which utilizes deep neural networks (*DNNs*), has had a profound impact on society and is now a core part of modern technology. Due to its ability to achieve significant success in many classification, regression and generative problems, deep learning has seen applications in many fields [1]. Fields such as healthcare [2], computer vision [3], language [4], speech recognition [5], cybersecurity [6], and many more.

Due to this impact, extensive research is being conducted in hopes of improving *DNNs'* performance. Efforts to achieve these improvements span advancements in algorithms, hardware, architectures and more [7]. As the number of parameters in state-of-the-art *DNN* models grows to reach one billion in the field of computer vision [8] and far surpass one billion in the field of large language models [9], enhancing performance becomes imperative.

Algorithmic innovations hold the potential to significantly enhance the efficiency of *DNNs*. For example, from $2012 - 2019$ algorithmic innovations lead to doubling the efficiency of computer vision models every 16 months, for large language models algorithmic innovations lead to doubling efficiency every 6 months over a course of 3 years [10]. These improvements can be achieved by reducing the number of computations, reducing memory usage and other methods.

Convolution, which is the backbone of many computer vision models, namely convolutional neural networks (*CNNs*) [11], has had several algorithms developed to perform the operation. Similarly, attention [12] which is the operation behind transformer models and many large language models [13] has seen several algorithms developed for it. Selecting the appropriate algorithm, though, is not a simple task. It depends on the use case of the *DNN* and various other factors, such as input size, operation configuration, and available hardware.

In addition, incorporating different algorithms for the operations used in a *DNN* is a non-trivial task. A task which requires expertise in both the algorithms and the format the *DNN* exists in. Thus, the ability to easily explore the impact of algorithms and select the algorithms in use by a *DNN* has been lost.

To address this, in this work, we present an open source framework[1] that enables a user to easily select the algorithms used in the operations of a *DNN*, facilitating the exploration of various algorithmic design choices and potentially leading to a more efficient *DNN*. The framework is titled *ai3*, an abbreviation for "algorithmic innovations for accelerated implementations of artificial intelligence". *ai3* provides accelerated *C++* [14] implementations of various algorithms for deep learning operations as well as a frontend in *Python* [15] which gives users fine grain control over the algorithms used by their *PyTorch DNN*. *ai3* also allows users to implement their own algorithms in *C++* and easily include these algorithms when installing the package. After installation, the user's implementations can be selected in the same manner the built-in implementations are. The algorithmic selection and ability for users to implement their own algorithms provides the possibility of algorithmic innovations.

II. Background

A. Deep Neural Networks

Deep neural networks are the primary method used to apply deep learning technologies [1]. Deep neural networks chain various layers in sequence to form a single computational model. The different layers represent the various operations performed on the data. Deep neural networks typically include many hidden layers along with the input and output layers. Input layers receive the data and output layers form the output from the altered data they receive. The hidden layers between

[1]Source code at *github.com/KLab-AI3/ai3*

the input and output layers are the networks computational engine.

Usually, *DNN* development is split into two stages. The first stage is the training stage where the parameters of the operations performed to the data are learned. The strategy for learning the parameters can vary greatly between *DNNs*, depending on the type of output the model produces, the data available, the operations performed in the model and more. The second stage is the inference stage in which the capabilities developed during the training stage are used to produce outputs. In this stage the *DNN* infers an output, based on its training, from data it has not encountered before.

B. Algorithmic Design Choices

Fine grain levels of control over the algorithms used in the *DNNs* operations can yield great benefits in performance. For example, changing the algorithm which performs the convolution in *CNNs* can alter the latency and memory usage of the model greatly [16] [17]. These results are reflected in transformer models as well [18]. Despite the existence of numerous algorithms, there is no single best algorithm for a given operation. Selecting the appropriate one is not a trivial task and depends on the specific *DNN* use case, memory and time constraints, available hardware, and other factors [19].

Many different algorithms to perform the convolution operation have been developed, these include *IM2COL*, *KN2ROW* [16], scalar *MM* [20] and more.

(a) *IM2COL:* The image to column technique transforms the convolution operation to a matrix multiplication operation by reshaping the input and kernel to column vectors. Once this is complete, highly optimized general matrix multiply (*GEMM*) routines can be utilized.
(b) *KN2ROW:* The kernel to row technique is also used to transform the convolution to matrix multiplication but differs in that it transforms the kernel into row vectors in order to decrease memory usage. After transforming the problem to one of matrix multiplication the same highly optimized routines can be utilized.
(c) *SMM:* Scalar Matrix Multiplication with Zero Packing avoids matrix multiplications and replaces it with matrix scaling operations. Each output image can be considered as the summation of shifted versions of the input image multiplied by the corresponding kernel weight.
(d) *Direct:* Direct convolution, as the name suggests, is a straight forward approach to convolution. Kernels are applied directly to the input without transforming the data.

The ability to choose between these algorithms is key to enhancing a *CNNs* performance as *CNNs* spend the vast majority of their time inferencing on performing the convolutions [21]. Determining the best algorithm for a convolution operation is a non-trivial task depending on many features of the operation such as input sizes, number of output channels, kernel dimensions, stride of the kernel and more [22].

Similar to *CNN*s, transformer models spend the vast majority of their inferencing time in utilizing one mechanism, attention [13]. Attention is an operation used in *DNNs* for language processing, it is used as it provides the benefit of being able to predict the target word depending on the context associated with the source, the *DNN* is aware of what part of the source should receive the most attention. In response to this, many different algorithms have been developed in order to complete the operation. These include *flash*, *bigbird*, *longformer*, *linformer* attention and more.

(a) *Flash:* An exact attention algorithm that uses tiling to increase the number of cache hits and decrease the number of memory read and writes between *GPU* high bandwidth memory.
(b) *LongFormer:* A self-attention algorithm which scales linearly as opposed to quadradically with input sequence length. This is achieved by combining a smaller windowed attention mechanism with a larger global attention mechanism.
(c) *BigBird:* A sparse attention mechanism which also reduces the quadratic dependency on input size to linear. BigBird is a universal approximator of sequence to sequence functions which preserves the properties of the quadratic full attention models.
(d) *Linformer:* An algorithm that approximates the attention mechanism reducing the time and space complexity from quadratic to linear. The algorithm has been found to perform on par with standard attention implementations.

Again, the algorithm providing the best performance depends on many factors such as, input sequence length, memory limitations and required accuracy [23].

C. Relevant Related Works

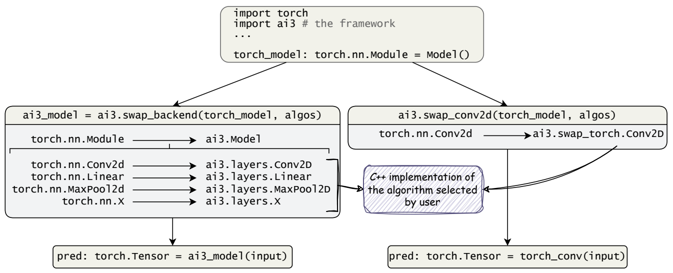
One library that has many implementations of deep learning operations is the *NVIDIA CUDA* Deep Neural Network library *(cuDNN)* [24]. *cuDNN* provides highly tuned, *GPU* accelerated implementations of common routines utilized in deep learning. These implementations include forward and backward operations for convolution, *matmul*, attention, pooling and normalization. The *cuDNN* library solely exposes a *C* [25] *API* meaning that it is non-trivial to explore different algorithms and select the algorithms in use by a *DNN*.

PyTorch is a leading *Python* package used for training and inferencing with *DNNs*. It is a highly optimized framework, especially for *GPUs*. When using *PyTorch's* implementations, however, there is currently no support for selecting the algorithm used to complete the operation. *PyTorch* uses *cuDNN* when possible and if *torch.backends.cudnn.benchmark* is set to true will try to discover and use the fastest implementation from *cuDNN*. However, support for manually setting the algorithm from *cuDNN* to use is not supported, though it may be in the future.

One package which seeks to optimize *PyTorch* models is the Intel Extension for *PyTorch (IPEX)* [26]. *IPEX* provides a very easy way to optimize *PyTorch* models via a *Python* front end to their *C++* implementations. Though this enables users to easily optimize their models, given they have the proper hardware, *IPEX* does not allow users to customize the algorithms used in the optimized models.

III. Framework Overview

The framework, titled *ai3*, enables users to seamlessly select different algorithms to be used in each layer of a *DNN*. Additionally, it provides a straightforward library which can be used to develop custom implementations of the operations used in the *DNN*. These custom implementations are then compiled along with the built-in ones and can be selected for use in the same manner the built-in implementations are. *ai3* currently supports the following operations, linear, convolution, flatten, *ReLU*, and adaptive average, max, and average pooling.



import torch
import ai3 # the framework
...
torch_model: torch.nn.Module = Model()
ai3_model = ai3.swap_backend(torch_model, algos)
torch.nn.Module
ai3.Model
torch.nn.Conv2d
torch.nn.Linear
torch.nn.MaxPool2d
torch.nn.X
ai3.layers.Conv2D
ai3.layers.Linear
ai3.layers.MaxPool2D
ai3.layers.X
ai3.swap_conv2d(torch_model, algos)
torch.nn.Conv2d
ai3.swap_torch.Conv2D
C++ implementation of the algorithm selected by user
pred: torch.Tensor = ai3_model(input)
pred: torch.Tensor = torch_conv(input)

Fig. 1: Example paths of a *PyTorch* module through the framework[2]

A. Use Cases

1) DNN Application Development: The first use case *ai3* seeks to address is a user who does not wish to implement their own algorithms but still wishes to customize the algorithms the existing *DNN* uses. This user is expected to be able to use *Python* and *PyTorch* but does not need to be familiar with *C++* or accelerated computing. For these users all that must be done to enable algorithmic selection is importing the package and adding a single line of *Python* code calling one of two functions which perform algorithmic selection on the *PyTorch DNN*.

2) DNN Algorithm Development: The second use case *ai3* seeks to address is a user developing implementations of custom algorithms for *DNNs*. For these users, *ai3* provides a *C++* library containing a *Tensor* class and various utility functions to do complete simple tasks and improve the developer experience. *ai3* also provides *C++* placeholder functions where users should implement their algorithms. After the package is installed, users can select their custom implementations for use by the *DNN*.

IV. Framework Implementation

Depending on the use case, there are two paths for installing the package. For the first use case, *ai3* is available as a *Python* package that can be installed via *pip*, the package installer for *Python*, with the command, *pip install aithree*. During installation, supported parallel computing platforms and libraries will be searched for and if found, the built-in implementations using such platforms and libraries will be utilized. For the second use case, where a user wishes to implement algorithms manually, users must download the source code instead of installing with *pip*. After downloading, users will find a *C++* header file for each operation. The header file contains the function signature of the operation to be implemented, access to the framework's *C++* library, and a boolean which controls default algorithm selection using the custom algorithm. As the installation process is able to compile arbitrary *C++* code, any existing *C++* compatible library can be used for the users implementation. Therefore, the ability to use existing libraries like *cuDNN*, *oneMKL* [28], etc. is not lost. Operability between *Python* and *C++* is provided via *PyBind11* [29]. The framework utilizes a build system consisting of *scikit-build-core* [30] and *CMAKE* [31]. After implementing their algorithms, users can install the package using *pip* with the installation target pointing to their local directory, again, parallel computing platforms and libraries on the machine will be utilized if possible. After installing the package, *ai3* provides a comprehensive test suite which can be used to ensure the correctness of custom implementations.

After installing the package, two functions to perform algorithmic selection become available for use in *Python*. The first, *swap_backend*, replaces all operations in a *DNN* with *ai3*'s implementation of the operation using the selected algorithm, resulting in a *DNN* fully managed by *ai3*. The second, *swap_(module type)*, swaps out all operations of a specific type with *ai3*'s implementation of the operation using the chosen algorithm. Code sample 1 demonstrates use of these functions.

```python
import torch
from torch import nn
import ai3  # the framework

class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3,
                               out_channels=16,
                               kernel_size=3, padding=1)
        self.maxpool = nn.MaxPool2d(kernel_size=2,
                                    stride=2)
        self.conv2 = nn.Conv2d(in_channels=16,
                               out_channels=32,
                               kernel_size=3, padding=1)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = self.maxpool(x)
        x = torch.relu(self.conv2(x))
        x = torch.flatten(x, 1)
        return x

input_data = torch.randn(10, 3, 224, 224)
orig = ConvNet()
torch_out = orig(input_data)
model: ai3.Model = ai3.swap_backend(orig,
                                    {"conv2d": "direct"})
sb_out = model(input_data)
ai3.swap_conv2d(orig, ["direct", "smm"])
sc_out = orig(input_data)
assert torch.allclose(torch_out, sb_out, atol=1e-6)
assert torch.allclose(torch_out, sc_out, atol=1e-6)
```

Listing 1: Basic Use of the Framework

[2]Created with *draw.io* [27]

The first function called, *swap_backend*, replaces every *PyTorch* module and function used with *ai3*'s implementation. It returns an object completely managed by the framework. The second function, *swap_conv2d*, replaces, in place, every instance of a *PyTorch torch.nn.Conv2d*, with another *torch.nn.Module* that uses *ai3*'s implementation of the selected algorithm. Both of these functions enable fine grain customization of the algorithms in use by the *DNN*. When swapping, each module type is associated with one of three algorithmic selectors. The simplest is a single string containing the name of algorithm to use for all modules of that type. In code sample 1, this strategy is used in the call to *swap_backend*, meaning that all the convolution layers in the *ai3.Model* utilize direct convolution as *"direct"* was passed. Next, a list of algorithms to use can be passed. As modules are encountered, they are replaced with an implementation of the algorithm in the list with the same index as that module has relative to the other modules of the same type. In code sample 1, this strategy is used in the call to *swap_conv2d*, meaning the first convolution module is replaced with an implementation of direct convolution and the second convolution module is replaced with an implementation of *SMM* convolution. The third method for algorithmic selection is a function which returns the algorithm to use and whose single parameter is the module the framework is currently replacing. This strategy enables powerful automated algorithmic selection. Code sample 2 demonstrates this strategy.

```
import torch
import torchvision
import ai3 # the framework

def selector(orig: torch.nn.Conv2d) -> str:
    in_channels = orig.weight.shape[1]
    if in_channels > 200:
        return "smm"
    return "direct"

input_data = torch.randn(1, 3, 224, 224)
vgg16 = torchvision.models.vgg16(
    weights=torchvision.models.VGG16_Weights.DEFAULT)
vgg16.eval()
torch_out = vgg16(input_data)

model: ai3.Model = ai3.swap_backend(vgg16,
                                    {"conv2d": selector})
sb_out = model(input_data)
assert torch.allclose(torch_out, sb_out, atol=1e-4)

ai3.swap_conv2d(vgg16, selector)
sc_out = vgg16(input_data)
assert torch.allclose(torch_out, sc_out, atol=1e-4)
```

Listing 2: Sample Using a Function to Select the Algorithms

When using *swap_backend*, a mapping is passed from the names of the modules to one of the three algorithmic selectors. If the user defines custom algorithms to use, they could select them explicitly by passing *"custom"* for the algorithm name. If the user has not made selections on which algorithms to use for a module, equivalent to passing *"default"* as the algorithm, then one of the framework's implementations is selected unless there is a user defined implementation for that operation in which case that is used. Additionally, if the user is swapping algorithms in a *PyTorch DNN* in place using *swap_(module type)* then a *"torch"* can be passed to keep the current *PyTorch* implementation.

As shown in samples 1 and 2, after calling the functions the altered object can be used to make predictions the same way a *PyTorch DNN* is.

In order to properly construct an equivalent *DNN*, a symbolic pass through the original *DNN* must be performed, storing all the operations encountered in a graph. This symbolic trace is achieved using the *PyTorch* function, *torch.fx.symbolic_trace*, this function performs the pass and returns a graph. This graph is then searched through to either construct a module separate from *PyTorch* in the case of a *swap_backend* call, or alter the *PyTorch DNN* in the case of a *swap_(module type)* call.

In order to construct a *DNN* completely managed by the framework, instances of the framework's modules are created and stored in a list in the order they are reached in the pass. This list is then passed to another module which constructs a single callable, an *ai3.Model*, containing all of them. As the list is formed, each module is given the algorithm the user chose to ensure that it utilizes the proper algorithm. The module which is returned to the user is completely managed by the framework and is able to perform all of its operations in *C++* without any *Python* until it has completed its operations.

On the other hand replacing only some operations of a *PyTorch* module requires that the replacements are descendants of the *torch.nn.Module* class. The replacement *torch.nn.Modules* act as slim wrappers around the frameworks equivalent operation whose forward function calls the implementation of the algorithm selected by the user.

As noted earlier, the framework provides an easy to use and robust testing suite. First, unit tests are employed to validate each operation. By varying hyperparameters, input sizes and more, we ensure that the operations remain equivalent to *PyTorch's* regardless of the operation's configuration. For example, when testing the *MaxPool2D* implementation against *PyTorch's* implementation, parameters such as kernel size, padding, dilation and stride are varied. Additionally, the input shape is adjusted by modifying the batch size, number of channels, height, and width. In addition to the unit tests, tests for both the *swap_backend* and *swap_(module type)* functions are present. These functions are performed by first creating a *PyTorch DNN*, then performing the swap and then performing a forward pass on the same input data, ensuring the results are the same. The original *PyTorch* modules used in these tests include various models in the *torchvision* [32] package along with some manually created *PyTorch* modules. For all tests, including unit tests, if more than one algorithm is available, all algorithms implemented will be tested to ensure the correctness of every implementation.

V. Experimental Evaluations

A. Setup

Experiments and evaluations are performed on the High Performance Computing Resource in the Core Facility for Advanced Research Computing at Case Western Reserve

University. The operations are performed on an *NVIDIA GeForce RTX* 2080 *Ti GPU*. Benchmarks are conducted on a single convolutional module as well as many prevalent *CNNs*. The *CNNs* used for benchmarks are *AlexNet* [33], *DenseNet* [34], *GoogleNet* [35], Inception-*v*3 [36], *ResNet* [37], *Swin* Transformer [38], Vision Transformer [39], *Squeezenet* [40] and *VGG*16 [41]. The input shapes used for the single convolution module are $(10, 3, 224, 224)$, $(10, 64, 112, 112)$, $(10, 256, 28, 28)$, and $(10, 512, 14, 14)$. The input shape used for all the *CNNs* is $(10, 3, 224, 224)$. The dimensions of the input correspond to $(\text{batch size}, \text{channels}, \text{height}, \text{width})$.

When benchmarking, we perform the operation using *PyTorch* in eager mode, along with multiple algorithms implemented by the framework. The benchmarks focus on convolution as it is a computationally expensive operation and acts as the core of many prevalent *DNNs*.

B. Algorithm Implementation

When performing benchmarks on the *GPU*, *PyTorch* makes use of *cuDNN* for implementations. Algorithms implemented by the framework and used in the benchmarks also rely on *cuDNN* to provide times comparable with *PyTorch* and demonstrate the low overhead of *ai3*. Operations are performed within a *torch.inference_mode()* context to avoid any computational overhead associated with preparing for gradient calculation. The algorithms implemented by *cuDNN* and used by *ai3* while performing benchmarks are, implicit *precomp GEMM*, implicit *GEMM*, *Winograd* [42], *GEMM* and using the algorithm selected by the *cuDNN* function *cudnnGetConvolutionForwardAlgorithm_v7*, which serves as a heuristic for obtaining the best suited algorithm.

(a) Implicit *precomp GEMM*: Expresses the convolution as a matrix product while not actually forming the necessary matrix. Various indices are calculated in a precomputation step which requires memory but assists the operation.
(b) Implicit *GEMM*: A zero memory overhead algorithm which expresses the convolution as a matrix product while not actually forming the necessary matrix.
(c) *GEMM*: Expresses the convolution as a matrix product and explicitly forms the matrix necessary.
(d) *Winograd*: Utilizes results of minimal complexity convolutions to increase performance when using smaller filters and batch sizes.

C. Results

Figures 2 and 3 represent the latency of a single convolution operation. The different colored bars represent the different algorithms. All algorithms but *torch* are run utilizing *ai3*. The *guess* algorithm uses the algorithm given by the *cudnnGetConvolutionForwardAlgorithm_v7* function described earlier. Figure 2 shows latency where algorithmic selection was performed via a call to the *swap_backend* function.

In contrast, in figure 3, algorithmic selection was performed via a call to the *swap_conv2d* function.

Figure 4 describes the end-to-end latency of the *CNNs* listed earlier processing 10 samples. The *PyTorch* convolution modules are swapped out for *ai3*'s implementation of the given algorithm via *ai3*'s *swap_conv2d* function. The elements are normalized per row with respect to the latency of *PyTorch*. This means that algorithms with values < 1 have lower latency and algorithms with values > 1 have greater latency than *PyTorch*.

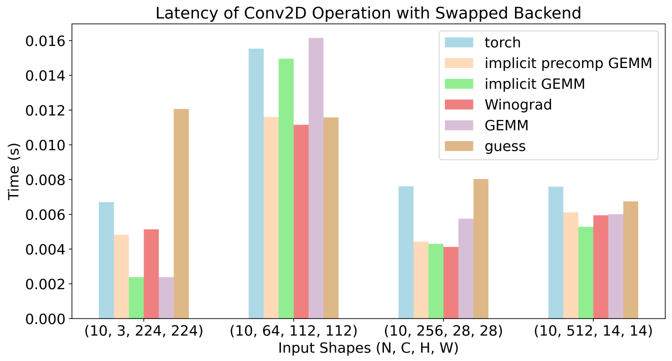


Fig. 2: Latency of Algorithms Using Swapped Backend

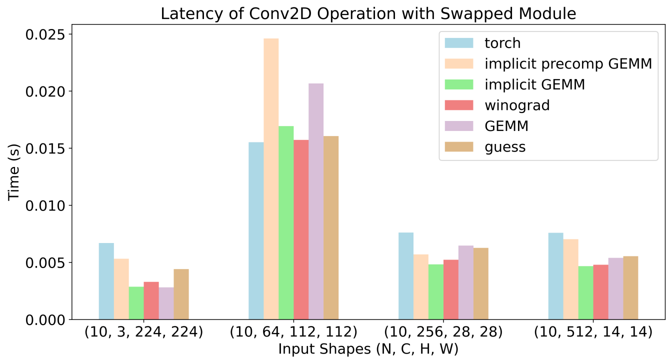
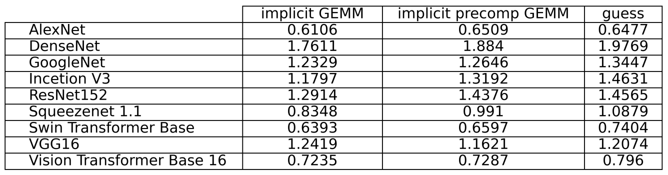


Fig. 3: Latency of Algorithms Using Swapped Modules



	implicit GEMM	implicit precomp GEMM	guess
AlexNet	0.6106	0.6509	0.6477
DenseNet	1.7611	1.884	1.9769
GoogleNet	1.2329	1.2646	1.3447
Incetion V3	1.1797	1.3192	1.4631
ResNet152	1.2914	1.4376	1.4565
Squeezenet 1.1	0.8348	0.991	1.0879
Swin Transformer Base	0.6393	0.6597	0.7404
VGG16	1.2419	1.1621	1.2074
Vision Transformer Base 16	0.7235	0.7287	0.796

Fig. 4: End-to-End Latency of *CNNs* with Swapped Modules Relative to *PyTorch*

Figures 3 and 2 which describe the latency of a single convolution operation generally show lower latency when utilizing *ai3* instead of *PyTorch*. As both packages utilize *cuDNN* implementations to perform the convolution, these results suggest that the latency in starting and finalizing the operation is lower in *ai3* compared to *PyTorch*, demonstrating the low overhead of *ai3*. The difference in latency between *ai3* and *PyTorch* is generally larger on bar graph 2 which shows the latencies of an *ai3.Model* which is returned by *swap_backend*. This again demonstrates the low overhead of the framework as a *ai3.Model* is generally able to commence the operations with lower latency than a *PyTorch nn.Module*. However, these results are not reflected in table 4 which displays the relative latencies of the *CNNs* listed earlier. On average, the original *PyTorch CNNs* have lower latency before swapping the convolution modules for *ai3*'s implementations of the given algorithm. This is likely due to optimizations for successive convolution calls in *PyTorch* that are not reflected in *ai3*. For example, *PyTorch* may be caching data for reuse to avoid reallocating and reconfiguring. Implementing such optimizations for the framework is possible and will be useful for improving performance.

VI. Limitations and Future Directions

Currently, some popular modules in *PyTorch* are still not implemented by *ai3*. This means that *swap_backend* is often not able to replace all operations of the *PyTorch DNN*. Adding support for additional modules will contribute towards *swap_backend* executing successfully in the majority of cases allowing an easy way to configure all algorithms utilized by the *DNN*. Until then, repeated calls to *swap_(module type)* functions can be made to enable algorithmic selection on as many modules as possible.

Secondly, *ai3* does not have built-in support for grouped convolution. However, once implemented, grouped convolution will be trivial to integrate and users will not have to make any change to their code. *ai3* is still able to perform grouped convolution provided the user implements a convolution algorithm that has grouped support using the custom algorithm features of the framework described earlier.

Lastly, some operations do not have accelerated implementations. Once all operations have accelerated implementations, *swap_backend* should provide similar or improved performance compared to the original *PyTorch DNN*. Until this is achieved, users can swap out as many modules as desired for *ai3*'s accelerated implementations using the *swap_(module type)* functions.

All of these limitations are not permanent and being worked on currently.

There are many future directions for *ai3*. First, support for more operations will be implemented. With the rise of transformer models, enabling algorithmic selection between various attention algorithms is becoming more and more crucial. Implementations for attention are being worked on and once implemented can be integrated into *ai3* and easily usable.

Second, in order to improve performance and integrate better with *PyTorch*, the ability to use *ai3*while in *PyTorch* graph mode is planned. This will enable improved performance as the operations surrounding *ai3*'s will be optimized. It is also planned to provide similar optimizations as *torch.compile* when using the *swap_backend* function.

Third, to enhance *ai3*'s portability, support for other *DNN* representations like the *ONNX* [43] format and *TensorFlow* [44] is planned. This will enable broader algorithmic selection and accessibility for a wider range of *DNNs* and users.

Lastly, backward propagation is planned to be supported. Integrating these computations into various *DNN* frameworks may be challenging, but it will enable powerful algorithmic selection for both the forward and backward passes of the *DNN*. Meaning users will have fine grain algorithmic control throughout the *DNN's* entire lifetime.

Acknowledgements

This work was supported in part by a summer research award provided by *CWRU* Undergraduate Research Office and by the U.S. National Science Foundation Award 2411447. Additionally, this work made use of the High Performance Computing Resource in the Core Facility for Advanced Research Computing at Case Western Reserve University.

VII. Conclusion

In this work we justified the existence of and demonstrated the usefulness of a framework that enables easy to use fine grain algorithmic selection for a *DNN*. The framework, *ai3*, features a frontend in *Python*, accelerated implementations of common deep learning operations, and a *C++* library which can be used to implement custom algorithms using any valid *C++* code. After installing the package two types of functions become available, one that swaps the entire backend of an existing *DNN*, swapping all the modules for *ai3*'s, and one that swaps a specific module type for *ai3*'s implementation. These functions allow fine grain customization of the algorithms used by passing the names of the algorithms to use directly or passing a function which examines the module and returns the algorithm to use. These algorithms are benchmarked and *ai3*is shown to have low overhead. In fact, the result of a *swap_backend* call is often able to begin the computations required faster than alternatives. *ai3* provides algorithmic selection at low cost, bringing the possibility of higher performance.

References

[1] I. H. Sarker, "Deep learning: A comprehensive overview on techniques, taxonomy, applications and research directions," *SN Computer Science*, vol. 2, no. 6, p. 420, Aug 2021. [Online]. Available: https://doi.org/10.1007/s42979-021-00815-1

[2] R. Miotto, F. Wang, S. Wang, X. Jiang, and J. T. Dudley, "Deep learning for healthcare: review, opportunities and challenges," *Briefings in Bioinformatics*, vol. 19, no. 6, pp. 1236–1246, 05 2017. [Online]. Available: https://doi.org/10.1093/bib/bbx044

[3] A. Voulodimos, N. Doulamis, A. Doulamis, E. Protopapadakis, and D. Andina, "Deep learning for computer vision: A brief review," *Intell. Neuroscience*, vol. 2018, jan 2018. [Online]. Available: https://doi.org/10.1155/2018/7068349

[4] I. Lauriola, A. Lavelli, and F. Aiolli, "An introduction to deep learning in natural language processing: Models, techniques, and tools," *Neurocomputing*, vol. 470, pp. 443–456, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231221010997

[5] Z. Zhang, J. Geiger, J. Pohjalainen, A. E.-D. Mousa, W. Jin, and B. Schuller, "Deep learning for environmentally robust speech recognition: An overview of recent developments," *ACM Trans. Intell. Syst. Technol.*, vol. 9, no. 5, apr 2018. [Online]. Available: https://doi.org/10.1145/3178115

[6] S. Mahdavifar and A. A. Ghorbani, "Application of deep learning to cybersecurity: A survey," *Neurocomputing*, vol. 347, pp. 149–176, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231219302954

[7] G. Menghani, "Efficient deep learning: A survey on making deep learning models smaller, faster, and better," *CoRR*, vol. abs/2106.08962, 2021. [Online]. Available: https://arxiv.org/abs/2106.08962

[8] M. Oquab, T. Darcet, T. Moutakanni, H. Vo, M. Szafraniec, V. Khalidov, P. Fernandez, D. Haziza, F. Massa, A. El-Nouby, M. Assran, N. Ballas, W. Galuba, R. Howes, P.-Y. Huang, S.-W. Li, I. Misra, M. Rabbat, V. Sharma, G. Synnaeve, H. Xu, H. Jegou, J. Mairal, P. Labatut, A. Joulin, and P. Bojanowski, "Dinov2: Learning robust visual features without supervision," 2024. [Online]. Available: https://arxiv.org/abs/2304.07193

[9] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[10] D. Hernandez and T. B. Brown, "Measuring the algorithmic efficiency of neural networks," 2020. [Online]. Available: https://arxiv.org/abs/2005.04305

[11] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[12] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," 2016. [Online]. Available: https://arxiv.org/abs/1409.0473

[13] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023. [Online]. Available: https://arxiv.org/abs/1706.03762

[14] B. Stroustrup, *The C++ Programming Language*, 2nd ed. Reading, Massachusetts: Addison-Wesley, 1991. [Online]. Available: http://www.worldcat.org/isbn/9780201539929

[15] Python Software Foundation, *Python Language Reference*, n.d. [Online]. Available: https://docs.python.org/reference/

[16] A. Vasudevan, A. Anderson, and D. Gregg, "Parallel multi channel convolution using general matrix multiplication," 2017. [Online]. Available: https://arxiv.org/abs/1704.04428

[17] J. Zhang, F. Franchetti, and T. M. Low, "High performance zero-memory overhead direct convolutions," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. PMLR, 10–15 Jul 2018, pp. 5776–5785. [Online]. Available: https://proceedings.mlr.press/v80/zhang18d.html

[18] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, "Linformer: Self-attention with linear complexity," 2020. [Online]. Available: https://arxiv.org/abs/2006.04768

[19] A. V. Trusov, E. E. Limonova, D. P. Nikolaev, and V. V. Arlazarov, "On fast computing of neural networks using central processing units," *Pattern Recognition and Image Analysis*, vol. 33, no. 4, pp. 756–768, Dec 2023. [Online]. Available: https://doi.org/10.1134/S105466182304048X

[20] A. Ofir and G. Ben-Artzi, "Smm-conv: Scalar matrix multiplication with zero packing for accelerated convolution," in *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2022, pp. 3066–3074.

[21] S. Wang, G. Ananthanarayanan, Y. Zeng, N. Goel, A. Pathania, and T. Mitra, "High-throughput cnn inference on embedded arm big.little multicore processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2254–2267, 2020.

[22] Y. Meng, S. Kuppannagari, R. Kannan, and V. Prasanna, "Dynamap: Dynamic algorithm mapping framework for low latency cnn inference," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '21. ACM, Feb. 2021. [Online]. Available: http://dx.doi.org/10.1145/3431920.3439286

[23] J. Zhang, S. Jiang, J. Feng, L. Zheng, and L. Kong, "Cab: Comprehensive attention benchmarking on long sequence modeling," 2023. [Online]. Available: https://arxiv.org/abs/2210.07661

[24] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," 2014. [Online]. Available: https://arxiv.org/abs/1410.0759

[25] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Englewood Cliffs, N.J.: Prentice Hall, 1988. [Online]. Available: http://www.worldcat.org/search?qt=worldcat_org_all&q=0131103628

[26] "Intel extension for PyTorch: A Python package for extending the official PyTorch that can easily obtain performance on Intel platform," 2024. [Online]. Available: https://github.com/intel/intel-extension-for-pytorch

[27] JGraph Ltd and draw.io AG, "draw.io," 2024, https://github.com/jgraph/drawio.

[28] M. Krainiuk, M. Goli, and V. R. Pascuzzi, "oneapi open-source math library interface," in *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2021, pp. 22–32.

[29] W. Jakob, J. Rhinelander, and D. Moldovan, "pybind11 – seamless operability between c++11 and python," 2017, https://github.com/pybind/pybind11.

[30] H. Schreiner, J. Rickerby, R. Grosse-Kunstleve, W. Jakob, M. Darbois, A. Gokaslan, J.-C. Fillion-Robin, and M. McCormick, "Building Binary Extensions with pybind11, scikit- build, and cibuildwheel," Aug. 2022. [Online]. Available: https://doi.org/10.25080/majora-212e5952-033

[31] kitware, "Cmake: A powerful software build system." [Online]. Available: https://cmake.org/

[32] T. maintainers and contributors, "Torchvision: Pytorch's computer vision library," https://github.com/pytorch/vision, 2016.

[33] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, p. 84–90, may 2017. [Online]. Available: https://doi.org/10.1145/3065386

[34] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," 2018. [Online]. Available: https://arxiv.org/abs/1608.06993

[35] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," 2014. [Online]. Available: https://arxiv.org/abs/1409.4842

[36] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," 2015. [Online]. Available: https://arxiv.org/abs/1512.00567

[37] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015. [Online]. Available: https://arxiv.org/abs/1512.03385

[38] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin transformer: Hierarchical vision transformer using shifted windows," 2021. [Online]. Available: https://arxiv.org/abs/2103.14030

[39] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," 2021. [Online]. Available: https://arxiv.org/abs/2010.11929

[40] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and ¡0.5mb model size," 2016. [Online]. Available: https://arxiv.org/abs/1602.07360

[41] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2015. [Online]. Available: https://arxiv.org/abs/1409.1556

[42] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," 2015. [Online]. Available: https://arxiv.org/abs/1509.09308

[43] O. Developers, "Onnx: Open standard for machine learning interoperability," https://github.com/onnx/onnx, 2024.

[44] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/