PYTORCH AMPERE® OPTIMIZED FRAMEWORK Documentation

v1.11.0

AMPERE COMPUTING

Table of Contents

RELEASE NOTES 1

OVERVIEW 4

PYTORCH FRAMEWORK 4

Versions Compatibility 4

PYTHON 4

CONFIGURATIONS 4

QUICKSTART 5

Launching Docker Container 6

Running Examples 6

AMPERE OPTIMIZED PYTORCH PROGRAMMING GUIDE 7

Overview 7

Supported Inference Ops 8

PyTorch JIT Trace 12

Torch Compile (beta) 12

Threading 13

Programming Tips 13

RELEASE NOTES

V1.11.0:

- PyTorch version upgraded to 2.2
- Enhance Whisper performance and compile time
- Enhance Unet model performance
- New layer ops support:
- Min, max, sum (reduce ops), Hardswish, Hardsigmoid, Conv1D, Conv3D (Transposed version too), Depthwise type Convolution 2D, InstanceNorm3D, GroupNorm3D, improved indexing support, permute and transpose on integer types, Index_add, scatter, expand, zeros, TopK, where (nonzero overload)
- Linear merge optmization with opt out switch
- Libampere-aio updated to 0.12.0:
    - Mixed precision mode - FC, Matmuls, EmbeddingBag
    - LLM dedicated ops
    - Native_fp16-int8 conversion merging
    - GPTQ 4-bit support
    - GPTQ 8-bit support
    - New custom ROPE op added (see below for details)

V1.10.0:

- Fix AIO backend bug that breaks and BART model
- LLM models performance increase
- Optimized AIO fuser, which speed ups initialization stage of bigger models like Whisper
- libampere-aio updated to 0.11.0
    - Support of implicit mixed precision (fp16 + int8). Major speed up of memory bound models like DLRM. Please see following instructions in how to use it
    - New custom RMSNorm op added (see below for details)

V1.9.0:

- libampere-aio updated to 0.10.0
    - New upsampling2d_nearest kernel to speed up YOLO models

- Experimental support of int8 FC to speed up models like DLRM
- Improved support of in place operators by the Pytorch-aio
- Updated to PyTorch 2.1
- Framework now is setting AIO_SKIP_MASTER_THREAD=1 env var by default, no need to specify it

V1.8.0:

- libampere-aio updated to 0.9.0
    - Resolve an issue when input shape changes frequently
    - Performance enhancements of some NLP models

V1.7.0:

- libampere-aio updated to 0.8.0
- Bug fixes and performance enhancements
- Improved memory management
- Improved model compilation times
- Improved algorithm calculating graph handled by AIO
- New operators supported: Baddbmm, sub, slice, max (elementwise), min (elementwise), neg, index (some cases), split_with_sizes, NumToTensor, Float, Int
- Options custom argument in AIO torch.dynamo backend (see below)

V1.6.0:

- libampere-aio updated to 0.7.0
- PyTorch updated to 2.0.0
- Bug fixes and performance enhancements
- New operator supported: LogSoftmax
- Torch.compile() supported (see section about it)

V1.5.2:

- **Note: v1.5.2 is a bug fix release to v1.5.1 and v1.5.0. It fixes an issue related to YOLO models. Please discard v1.5.0 and v1.5.1 you've installed.**
- libampere-aio updated to 0.6.1
- Bug fixes and performance enhancements
- New operators supported: Split, Chunk, Sqrt, Rsqrt, Exp, Log, Zeros_Like, Embedding, Mean

- TorchScript loops are supported.
- Improved lifetime handling of Torchscript models

V1.4.0:

- libampere-aio updated to 0.5.0
- Pytorch framework updated to 1.12.1 from 1.11.0
- Support of FP16 ops (automatic mode)
- New operators supported: deconv2d, embedding bag
- Improved memory management
- Bug fixes: Instance Norm op fix, thread safety

V1.3.0:

- Binary integer operations support.
- libampere-aio updated to 0.4.0
- New operators supported: Reshape, Squeeze, Unsqueeze, Flatten, PixelShuffle, GroupNorm, InstanceNorm.
- Using custom compiled OpenBLAS, as Pytorch BLAS backend.
- Bug fixes

V1.2.0:

- libampere-aio updated to 0.3.0
- New optimized operators: Gelu, Silu, Softmax, Div, Binary ops between Tensor and Scalar, Permute, View, Layer Norm, Size, Pow, Tanh, Sigmoid
- Improved Concat support
- Graph optimizations
- Various bugfixes

V1.1.0:

- Libampere-aio updated to 0.2.1
- Batch Matmul supported (enhancing DLRM performance)
- Adaptive Avg Pool supported
- LeakyRelu supported
- AIO_NUM_THREADS no longer needed to set Ampere Optimized PyTorch threads, inherits Pytorch intra-op thread count.

OVERVIEW

Ampere Optimized PyTorch inference acceleration engine is fully integrated with the PyTorch framework. PyTorch models and software written with the PyTorch API can run as-is, without modifications.

PYTORCH FRAMEWORK

Python is installed with Ampere Optimized PyTorch and all dependencies. No additional installation steps are needed.

Versions Compatibility

This release is based on Pytorch 2.0.0 and comes with the compatible Torchvision 0.15.1 installed.

PYTHON

Pytorch 2.0.0 is built for Python 3.10, supporting Ubuntu 22.04. Regarding other Python versions, please contact your Ampere sales representative. If you are using the software through a third party, contact their customer support team for help. You can also contact the AI team at ai-support@amperecomputing.com.

CONFIGURATIONS

Ampere Optimized PyTorch inference engine can be configured by a set of environment variables for performance and debugging purposes. They can be set in the command line when running Pytorch models (e.g., AIO_NUM_THREADS=16 python run.py -p fp32) or set in the shell initialization script.

AIO_PROCESS_MODE

This variable controls whether the Ampere Optimized PyTorch inference engine is used to run the Pytorch model:

- 0: disabled.
- 1: enabled (Default).

AIO_CPU_BIND

Enables core binding. If enabled, each Ampere Optimized PyTorch thread will bind itself to a single core:

- 0: Core binding disabled.
- 1: Core binding enabled (Default).

AIO_MEM_BIND

Binds memory to NUMA (Non-uniform memory access) node 0. For optimal performance, numactl (https://linux.die.net/man/8/numactl) is preferred. numactl bind will affect both the Pytorch framework and the optimized framework buffers, while the optimized framework is unable to affect buffers allocated by the Pytorch framework:

- 0: Membind disabled.
- 1: Membind to node 0 (Default).

AIO_NUMA_CPUS

Select the cores that Ampere Optimized PyTorch should bind to (if CPU_BIND is enabled):

- Not set: use the first N cores of the machine, excluding hyper-threaded (Default).
- Set: use N first cores from the list of cores for N threads. The list is in space separated, 0-based number format. For example, selecting cores 0 to 1: AIO_NUMA_CPUS="0 1".

AIO_DEBUG_MODE

Control the verbosity of debug messages:

- 0: No messages
- 1: Errors only
- 2: Basic information, warnings, and errors (Default)
- 3: Most messages
- 4: All messages

QUICKSTART

The following instructions run on Altra/Altra Max Linux machines installed **with Docker**. When you are already using a virtual machine pre-installed with the version of Ampere Optimized PyTorch (e.g. on a cloud service provider) that you need, you can skip the following step of launching Docker container.

Note: This docker image is developed for benchmarking and evaluation purpose, not for deployment into production environment. We will provide required Debian, RPM and Python packages as needed for your production deployment.

Launching Docker Containermas

Pulling Docker Image from Docker Hub repository

```
$ docker pull amperecomputingai/pytorch:1.11.0
```

Launching Docker Container

```
$ docker run --privileged=true --rm --name pytorch-aio --network host -it amperecomputingai/pytorch:1.11.0
```

```
Warning: This user has, by default, root privileges with Docker. Please limit permission according to your security policy.
```

Running Examples

You can try Ampere Optimized PyTorch by either running the Jupyter Notebook examples or Python scripts on the CLI level.

To run the Jupyter Notebook QuickStart examples follow the instructions below:

Set AIO_NUM_THREADS to the requested value first.

```
$ export AIO_NUM_THREADS=16; export OMP_NUM_THREADS=16
$ cd /workspace/aio-examples/
$ bash start_notebook.sh
```

If you run the Jupyter Notebook Quickstart on a cloud instance, make sure your machine has port 8080 open and on your local device run:

```
$ ssh -N -L 8080:localhost:8080 -I <ssh_key> your_user@xxx.xxx.xxx.xxx
```

Use a browser to point to the URL printed out by the Jupyter Notebook launcher.

You will find Jupyter Notebook examples (examples.ipynb) under the /classification and /object detection folders.

The examples run through several inference models, visualize results they produce, and present the performance numbers.

To use CLI-level scripts:

Set AIO_NUM_THREADS to the requested value first.

```
$ export AIO_NUM_THREADS=16; export OMP_NUM_THREADS=16
$ cd /workspace/aio-examples/
```

Go to the directory of choice, e.g.

```
$ cd classification/resnet_50_v1
```

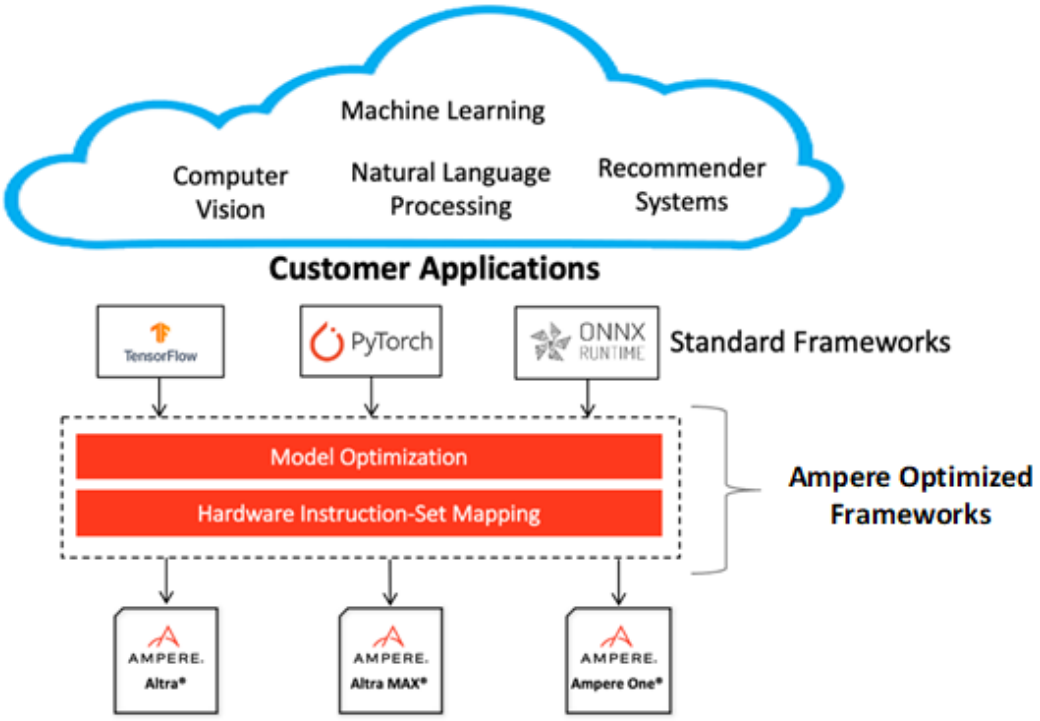
Evaluate the model.

```
$ numactl --physcpubind=0-15 python3 run.py -p fp32
```

AMPERE OPTIMIZED PYTORCH PROGRAMMING GUIDE

Overview

Ampere Optimized PyTorch is powered by Ampere® AI backend that accelerates Deep Learning (DL) operations on the Ampere® Altra family of processors. Ampere Optimized PyTorch accelerates DL operations through model optimization, highly vectorized compute kernels and multi-thread operations that are automatically tuned to deliver the best latency and throughput on Ampere Altra processors. It delivers 2-5x gains over alternative backend solutions.



Supported Inference Ops

Ampere Optimized Pytorch accelerates the most common Pytorch ops that are used in diverse types of models. Here is a list of accelerated ops and formats (Note: non-accelerated ops will still run without a problem, at the original framework operator speed):

Layer	FP32	Explicit FP16 (Model defined)	Implicit FP16 (Automatic on-the-fly conversion)	Int8 (Automatic on-the-fly) [Experimental]	Notes
Conv2d	Y	Y	Y		
Deconv2d	Y	Y	Y		Without bias
Conv3d	Y	Y	Y		
Deconv3d	Y	Y	Y		
Conv1d	Y	Y	Y		
Linear	Y	Y	Y	Y	
MaxPool2d	Y		Y		
AvgPool2d	Y		Y		
AdaptiveAvgPool2d	Y		Y		
Relu	Y	Y	Y		
Relu6	Y	Y	Y		
LeakyRelu	Y	Y	Y		
Softmax	Y	Y	Y		
LogSoftmax	Y	Y	Y		
Gelu	Y	Y	Y		
Silu	Y	Y	Y		
Sigmoid	Y		Y		
Tanh	Y		Y		
Transpose	Y	Y	Y		
Permute	Y	Y	Y		
BatchNorm	Y		Y		
LayerNorm	Y	Y	Y		
GroupNorm	Y				2D and 3D

InstanceNorm	Y				2D and 3D
RmsNorm	Y	Y	Y		
Add	Y	Y	Y		Int version not optimized
Mul	Y	Y	Y		Int version not optimized
Div	Y	Y	Y		Int version not optimized
Pow	Y	Y	Y		Int version not optimized
Matmul	Y	Y	Y	Y	
MM	Y	Y	Y		
BMM	Y	Y	Y		
PixelShuffle	Y				
View	Y	Y	Y		
Reshape	Y	Y	Y		
Squeeze	Y	Y	Y		
Unsqueeze	Y	Y	Y		
Flatten	Y	Y	Y		
Contiguous	Y	Y	Y		
Size	Y	Y	Y		One dimension case
EmbeddingBag	Y	Y	Y	Y	Sum mode
Embedding	Y	Y	Y		
Split	Y	Y	Y		
Chunk	Y	Y	Y		
Sqrt	Y		Y		
Rsqrt	Y		Y		
Exp	Y		Y		
Log	Y		Y		
Zeros_like	Y				

Mean	Y		Y		
Baddbmm	Y		Y		
Slice	Y	Y	Y		
Select	Y		Y		
Neg	Y	Y	Y		
Split with sizes	Y		Y		
Index	Y		Y		Limited support
Max	Y		Y		Elementwise and reduce
Min	Y		Y		Elementwise and reduce
Sum	Y		Y		
Sub	Y		Y		
UpsampleNearest2d	Y		Y		Support only constant scale factor and output size (cannot change in model lifetime)
Sum	Y		Y		
Hardswish	Y		Y		
Hardsigmoid	Y		Y		
Index_add	Y		Y		
Scatter	Y		Y		
Expand	Y		Y		
Where	Y		Y		nonzero overload
Zeros	Y		Y		
TopK	Y		Y		

New custom op: RMSNorm

We added custom operation for RMS norm, since Pytorch does not have fused kernel for it. There are two methods to use it:

1. Low level with
   torch.nn.functional.layer_norm(input, shape, weight, None, epsilon, True)

Interface it same as in layer_norm however there is extra bool argument which must be set to True to trigger RMSnorm kernel. Also note that bias must be set to None since RMS does not support bias.

2. High level with torch.nn.RMSNorm module.

```
class RMSNorm(Module):

def __init__(self, normalized_shape: _shape_t, eps: float = 1e-5, elementwise_affine: bool = True, device=None, dtype=None) -> None:
```

Again, interface is like torch.nn.LayerNorm however it does not have bias argument since the op does not support it.

New operation is much more performant than computing the norm by the series of layers, new op can be implemented on model level to take advantage of it.

Note that new op does not support backward pass.

New custom op: RoPE

We added custom operation for RoPE, since Pytorch does not have fused kernel for it. There is a new function:

```
torch.aio_rope(query_states, position_ids, self.rope_theta)
```

Now instead of :

```
def rotate_half(x):
"""Rotates half the hidden dims of the input."""
x1 = x[..., : x.shape[-1] // 2]
x2 = x[..., x.shape[-1] // 2 :]
return torch.cat((-x2, x1), dim=-1)

def apply_rotary_pos_emb(q, k, cos, sin, position_ids=None, unsqueeze_dim=1):
cos = cos.unsqueeze(unsqueeze_dim)
sin = sin.unsqueeze(unsqueeze_dim)
q_embed = (q * cos) + (rotate_half(q) * sin)
k_embed = (k * cos) + (rotate_half(k) * sin)
return q_embed, k_embed
```

```
query_states, key_states = apply_rotary_pos_emb(query_states, key_states, cos, sin)
```

PyTorch JIT Trace

While Pytorch Eager Execution provides excellent model building, programming, and debugging experience, it is slower than graph execution. So, Torchscript is typically used for inference deployment. In the current version of Ampere Optimized Pytorch, Torchscript mode is also accelerated.

To use Ampere Optimized Pytorch, conversion of Pytorch module to Torchscript is needed. There are two ways to convert: torch.jit.script() or torch.jit.trace(input) API calls. See https://pytorch.org/docs/stable/jit.html for more details. After converting to Torchscript user should call torch.jit.freeze() to freeze the models and enable model optimizations for inference.

Torch Compile (beta)

Ampere Optimized Pytorch support torch.compile API introduced in Pytorch 2.0 release. This is a new mode for optmizing model for infenence. To take advantage of it user must compile the model with AIO backend by using compiled_model = torch.compile(model, backend="aio", options={"modelname": "model"}). It is important to explicitly select "aio" backend and pass additional parameter named options with "modelname" field. See https://pytorch.org/get-started/pytorch-2.0/ for more information.

Note: In this release this is a beta feature. Torchscript is likely to be faster than torch.compile.

Implicit FP16 mode

Ampere Optimized PyTorch backend now provides automatic FP16 operator conversion that can boost the performance of your FP32 model on-the-fly. It automatically performs FP16 conversion and computation for certain whitelisted operators through regular expression. To take advantage of that, you can set environment variable.

```
$export AIO_IMPLICIT_FP16_TRANSFORM_FILTER=".*"
```

This activates automatic FP16 conversion for all supported operators. It is estimated that this has minor impact on the accuracy of common models. Please contact us if you have any questions about this feature.

Linear merge opt out switch

Quite often in the transformer architecture linear is run on same input tensor with different weights (for example qkv projection in the attention mechanism) we implemented the optimization which finds such situations and merges them to one computation. It is faster however it costs extra memory to keep extra weight tensor. In some cases (large LLMs) it may cause out of memory errors. In such cases user can turn off the optmization with

```
$export AIO_MERGE_LINEAR=0
```

Implicit INT8 mixed precision mode (experimental)

Another performance boost can be achieved with implicit model quantization.
It is enabled with:

```
$export AIO_QUANTIZE_INT8_FILTER=".*"
```

It will replace supported operations with their quantized INT8 versions. Supported operations are FullyConnected (linear layer in Pytorch), Matmul (different flavours like mm, bmm, etc) and EmbeddingBag. It

worst bests with big models, ale Matmul sometimes introduces the slowdown in that case you should use different filter like:

```
$export AIO_QUANTIZE_INT8_FILTER="FullyConnected"
```

The new environmental variable replaced the old `AIO_QUANTIZE_INT8=ALL`

It is possible to use this mode along with implicit FP16 mode. You can specify both variables to enable that mode.

If you have fp16 model (for example via mode.half() call), and it works in Pytorch (Pytorch lacks some layers FP16 support on CPU, so not all models work in FP16 on CPU). You can use int8 mixed precision too. However, you must specify extra environmental variable:

```
AIO_QUANTIZE_INT8_CONVERT_OUT_DTYPE=FP16
```

This would be preffered solution for better performance because extra FP16->FP32 converstions are eliminated in that mode.

Threading

Ampere Optimized PyTorch controls the number of Ampere Optimized Pytorch intra_op threads with torch.set_num_threads(). This controls both the number of threads used for ops delegated to Ampere Optimized Pytorch as well as the ops running on default CPU backend.

Some default CPU backend ops (non-AIO) also need to set OMP_NUM_THREADS environment variable to control the intra_op threads.

Programming Tips

In the first two inference passes, Ampere Optimized Pytorch performs a runtime compilation of PyTorch script and prepares Ampere Optimized Pytorch network. So, the latency of the first two passes is expected to be longer. Subsequent passes will be accelerated.

Ampere Optimized PyTorch provides much better latency scaling as core count increases, compared to other platforms. You can easily try the optimal number of cores with the above set_num_threads() function that can give you the best price / performance, while meeting your latency requirements.

Models are optimized for the shape of the tensors that are used during the compilation phase (see above). Passing different shape tensors will work but is suboptimal. To get the best performance pad varying shape tensors when running inference.

If any issues occur, Ampere AI team is ready to help. Typically, the first step is to get more debug logs and send them to ai-support@amperecomputing.com. Please set environment variable AIO_DEBUG_MODE=4 to capture low level logs.

Limitations

Ampere Optimized PyTorch does not support dynamic ranks of tensors (different rank in subsequent passes). Dynamic shapes of Tensors are supported but not recommended, ideally one should pad inputs to the network to get best performance.

We can also provide more in-depth profiling of your model to help enhance performance to meet your needs.

Ampere Computing® / 4655 Great America Parkway, Suite 601 / Santa Clara, CA 95054 / www.amperecomputing.com

AMP 2019-0039