Quark

Release 0.1.0

AMD

Aug 27, 2024

CONTENTS

Quark is a deep learning model quantization toolkit for quantizing models from PyTorch, ONNX and other frameworks. It provides easy-to-use APIs for quantization and more advanced features than native frameworks, in support for multiple HW backends.

CHAPTER

ONE

QUARK FOR PYTORCH

1.1 New Features (Version 0.1.0):

- **Pytorch Quantizer Enhancements**:
  - Eager mode is supported.
  - Post Training Quantization (PTQ) is now available.
  - Automatic in-place replacement of `nn.module` operations.
  - Quantization of the following modules is supported: `torch.nn.linear`.
  - The customizable calibration process is introduced.
- **Quantization Strategy**:
  - Symmetric and asymmetric quantization are supported.
  - Weight-only, dynamic, and static quantization modes are available.
- **Quantization Granularity**:
  - Support for per-tensor, per-channel, and per-group granularity.
- **Data Types**:
  - Multiple data types are supported, including float16, bfloat16, int4, uint4, int8, and fp8 (e4m3fn).
- **Calibration Methods**:
  - MinMax, Percentile, and MSE calibration methods are now supported.
- **Large Language Model Support**:
  - FP8 KV-cache quantization for large language models(LLMs).

- **Advanced Quantization Algorithms**:
  - Support SmoothQuant, AWQ(uint4), and GPTQ(uint4) for LLMs. (**Note**: AWQ/GPTQ/SmoothQuant algorithms are currently limited to single GPU usage.)
- **Export Capabilities**:
  - Export of Q/DQ quantized models to ONNX and vLLM-adopted JSON-safetensors format now supported.
- **Operating System Support**:
  - Linux (supports ROCM and CUDA)
  - Windows (support CPU only).

CHAPTER

TWO

INSTALLATION GUIDE

2.1 Install from ZIP

1. Install PyTorch for the compute platform(CUDA, ROCM, CPU...). Version of torch >= 2.2.0.
2. Download the quark.zip. Extract the downloaded zip file and there is a whl package in it.
3. Install quark whl package by

```
pip install [quark whl package].whl
```

4. (Optional) Verify the installation by running `python -c "import quark"`. If it does not report error, the installation is done.
5. (Optional) Compile the `fast quantization kernels`. When using Quark's quantization APIs for the first time, it will compile the `fast quantization kernels` using your installed Torch and CUDA if available. This process may take a few minutes but subsequent quantization calls will be much faster. To invoke this compilation now and check if it is successful, run the following command:

```
python -c "import quark.torch.kernel"
```

CHAPTER

THREE

USER GUIDE

There are several steps to quantize a floating-point model with `Quark for PyTorch`:

1. Load original float model
2. Set quantization configuration
3. Define dataloader
4. Use the Quark API to perform in-place replacement of the model's modules with quantized module.
5. (Optional) Export quantized model to other format such as ONNX

More details:

- Quick Start Example
- Configuring Quark for PyTorch
- Adding Calibration Datasets
- Exporting for ONNX & Json-Safetensors(vLLM Adopted)
- Feature Description

CHAPTER

FOUR

APIS

User facing APIs:

4.1 quark.torch.quantization.api

4.1.1 Module Contents

Classes

class quark.torch.quantization.api.**ModelQuantizer**(*config:* quark.torch.quantization.config.config.Config)

Provides an API for quantizing deep learning models using PyTorch. This class handles the configuration and processing of the model for quantization based on user-defined parameters. It is essential to ensure that the 'config' provided has all necessary quantization parameters defined. This class assumes that the model is compatible with the quantization settings specified in 'config'.

Parameters

config (Config) – Configuration object containing settings for quantization.

quantize_model(*model: torch.nn.Module*, *dataloader: Union[torch.utils.data.DataLoader[torch.Tensor], torch.utils.data.DataLoader[List[Dict[str, torch.Tensor]]], torch.utils.data.DataLoader[Dict[str, torch.Tensor]]]*) → torch.nn.Module

This function aims to quantize the given PyTorch model to optimize its performance and reduce its size. This function accepts a model and a torch dataloader. The dataloader is used to provide data necessary for calibration during the quantization process. Depending on the type of data provided (either tensors directly or structured as lists or dictionaries of tensors), the function will adapt the quantization approach accordingly.It's important that the model and dataloader are compatible in terms of the data they expect

and produce. Misalignment in data handling between the model and the dataloader can lead to errors during the quantization process.

Parameters

- **model** (*nn.Module*) – The PyTorch model to be quantized. This model should be already trained and ready for quantization.
- **dataloader** (*Union[DataLoader[torch.Tensor], DataLoader[List[Dict[str, torch.Tensor]]], DataLoader[Dict[str, torch.Tensor]]]*) – The DataLoader providing data that the quantization process will use for calibration. This can be a simple DataLoader returning tensors, or a more complex structure returning either a list of dictionaries or a dictionary of tensors.

Returns

The quantized version of the input model. This model is now optimized for inference with reduced size and potentially improved performance on targeted devices.

Return type

nn.Module

Examples:

```
# Model & Data preparation
from transformers import AutoModelForCausalLM,␣
↪AutoTokenizer
model = AutoModelForCausalLM.from_pretrained("facebook/
↪opt-125m")
model.eval()
tokenizer = AutoTokenizer.from_pretrained("facebook/opt-
↪125m")
from quark.torch.quantization.config.config import Config
from quark.torch.quantization.config.custom_config import␣
↪DEFAULT_W_UINT4_PER_GROUP_CONFIG
quant_config = Config(global_quant_config=DEFAULT_W_UINT4_
↪PER_GROUP_CONFIG)
from torch.utils.data import DataLoader
text = "Hello, how are you?"
tokenized_outputs = tokenizer(text, return_tensors="pt")
calib_dataloader = DataLoader(tokenized_outputs['input_ids
↪'])

from quark.torch import ModelQuantizer
quantizer = ModelQuantizer(quant_config)
quant_model = quantizer.quantize_model(model, calib_
```

(continues on next page)

(continued from previous page)

```
↪dataloader)
```

freeze(*model: torch.nn.Module*) → torch.nn.Module

Freezes the quantized model by replacing FakeQuantize modules with FreezedFakeQuantize modules. If Users want to export quantized model to torch_compile, please freeze model first.

Parameters

model (*nn.Module*) – The neural network model containing quantized layers.

Returns

The modified model with FakeQuantize modules replaced by FreezedFakeQuantize modules.

Return type

nn.Module

4.2 quark.torch.export.api

4.2.1 Module Contents

Classes

class quark.torch.export.api.**ModelExporter**(*config:* quark.torch.export.config.config.ExporterConfig, *export_dir: Union[pathlib.Path, str] = tempfile.gettempdir()*)

Provides an API for exporting quantized Pytorch deep learning models. This class converts the quantized model to json-safetensors files or onnx graph, and saves to export_dir.

Parameters

- **config** (ExporterConfig) – Configuration object containing settings for exporting.
- **export_dir** (*Union[Path, str]*) – The target export diretory. This could be a string or a pathlib.Path(string) object.

export_model_info(*model: torch.nn.Module*, *model_type: str*, *model_dtype: torch.dtype = torch.float16*, *export_type: str = 'vllm-adopt'*) → None

This function aims to export json and safetensors files of the quantized Pytorch model. The model's network architecture is stored in the json file, and parameters including weight, bias, scale, and zero_point are stored in the safetensors file.

Parameters

- **model** (*torch.nn.Module*) – The quantized model to be exported.
- **model_type** (*str*) – The type of the model, e.g. gpt2, gptj, llama or gptnext.
- **model_dtype** (*torch.dtype*) – The weight data type of the quantized model. Default is torch.float16.
- **export_type** (*str*) – The specific format in which the JSON and safetensors files are stored. The choices include 'vllm-adopt' and 'native'. Default is vllm-adopt. If set to 'vllm-adopt', the exported files are customized for the VLLM compiler. The 'native' configuration is currently for internal testing use.

Returns

None

Examples:

```
export_path = "./output_dir"
from quark.torch import ModelExporter
from quark.torch.export.config.custom_config import␣
↪DEFAULT_EXPORTER_CONFIG
exporter = ModelExporter(config=DEFAULT_EXPORTER_CONFIG,␣
↪export_dir=export_path)
exporter.export_model_info(model, model_type, model_dtype,
↪ export_type="vllm-adopt")
```

Note: Since the export_type "native" is only for internal testing use currently, this function is only used to export files required by the VLLM compiler. Supported quantization types include fp8, int4_per_group, and w4a8_per_group. Supported models include Llama2-7b, Llama2-13b, Llama2-70b, and Llama3-8b.

---

export_onnx_model(*model: torch.nn.Module*, *input_args: Union[torch.Tensor, Tuple[float]]*, *input_names: List[str] = []*, *output_names: List[str] = []*, *verbose: bool = False*, *opset_version: Optional[str] = None*, *do_constant_folding: bool = True*, *operator_export_type: torch.onnx.OperatorExportTypes = torch.onnx.OperatorExportTypes.ONNX*, *uint4_int4_flag: bool = False*) → None

This function aims to export onnx graph of the quantized Pytorch model.

Parameters

- **model** (*torch.nn.Module*) – The quantized model to be exported.

- **input_args** (*Union[torch.Tensor, Tuple[float]]*) – Example inputs for this quantized model.
- **input_names** (*List[str]*) – Names to assign to the input nodes of the onnx graph, in order. Default is empty list.
- **output_names** (*List[str]*) – Names to assign to the output nodes of the onnx graph, in order. Default is empty list.
- **verbose** (*bool*) – Flag to control showing verbose log or no. Default is False
- **opset_version** (*Optional[str]*) – The version of the default (ai.onnx) opset to target. If not set, it will be valued the latest version that is stable for the current version of PyTorch.
- **do_constant_folding** (*bool*) – Apply the constant-folding optimization. Default is False
- **operator_export_type** (*torch.onnx.OperatorExportTypes*) – Export operator type in onnx graph. The choices include OperatorExportTypes.ONNX, OperatorExportTypes.ONNX_FALLTHROUGH, OperatorExportTypes.ONNX_ATEN and OperatorExportTypes.ONNX_ATEN_FALLBACK. Default is OperatorExportTypes.ONNX.
- **uint4_int4_flag** (*bool*) – Flag to indicate uint4/int4 quantized model or not. Default is False.

Returns

None

Examples:

```
from quark.torch import ModelExporter
from quark.torch.export.config.custom_config import␣
↪DEFAULT_EXPORTER_CONFIG
exporter = ModelExporter(config=DEFAULT_EXPORTER_CONFIG,␣
↪export_dir=export_path)
exporter.export_onnx_model(model, input_args)
```

Note: Mix quantization of int4/uint4 and int8/uint8 is not supported currently. In other words, if the model contains both quantized nodes of uint4/int4 and uint8/int8, this function cannot be used to export the ONNX graph.

4.3 quark.torch.quantization.config.config

4.3.1 Module Contents

Classes

class quark.torch.quantization.config.config.**Config**

A class that encapsulates comprehensive quantization configurations for a machine learning model, allowing for detailed and hierarchical control over quantization parameters across different model components.

Parameters

- **global_quant_config** (QuantizationConfig) – Global quantization configuration applied to the entire model unless overridden at the layer level.
- **layer_type_quant_config** (*Dict[str, QuantizationConfig]*) – A dictionary mapping from layer types (e.g., 'Conv2D', 'Dense') to their quantization configurations. Default is an empty dictionary.
- **layer_quant_config** (*Dict[str, QuantizationConfig]*) – A dictionary mapping from layer names to their quantization configurations, allowing for per-layer customization. Default is an empty dictionary.
- **exclude** (*List[str]*) – A list of layer names to be excluded from quantization, enabling selective quantization of the model. Default is an empty list.
- **algo_config** (*Optional[AlgoConfig]*) – Optional configuration for the quantization algorithm, such as GPTQ and AWQ. After this process, the datatype/fake_datatype of weights will be changed with quantization scales. Default is None.
- **pre_quant_opt_config** (*Optional[Union[PreQuantOptConfig, List[PreQuantOptConfig]]]*) – Optional pre-processing optimization, such as Equalization and SmoothQuant. After this process, the value of weights will be changed, but the dtype/fake_dtype will be the same. Default is None.

class quark.torch.quantization.config.config.**QuantizationConfig**

A data class that specifies quantization configurations for different components of a module, allowing hierarchical control over how each tensor type is quantized.

Parameters

- **input_tensors** (*Optional[QuantizationSpec]*) – Input tensors quantization specification. If None, following the hierarchical quantiza-

tion setup. e.g. If the input_tensors in layer_type_quant_config is None, the configuration from global_quant_config will be used instead. Defaults to None. If None in global_quant_config, input_tensors are not quantized.

- **output_tensors** (*Optional[QuantizationSpec]*) – Output tensors quantization specification. Defaults to None. If None, the same as above.
- **weight** (*Optional[QuantizationSpec]*) – The weights tensors quantization specification. Defaults to None. If None, the same as above.
- **bias** (*Optional[QuantizationSpec]*) – The bias tensors quantization specification. Defaults to None. If None, the same as above.
- **target_device** (*Optional[DeviceType]*) – Configuration specifying the target device (e.g., CPU, GPU, IPU) for the quantized model.

class quark.torch.quantization.config.config.**QuantizationSpec**

A data class that defines the specifications for quantizing tensors within a model.

Parameters

- **dtype** (*Dtype*) – The data type for quantization (e.g., int8, int4).
- **is_dynamic** (*Optional[bool]*) – Specifies whether dynamic or static quantization should be used. Default is None, which indicates no specification.
- **observer_cls** (*Optional[Type[ObserverBase]]*) – The class of observer to be used for determining quantization parameters like min/max values. Default is None.
- **qscheme** (*Optional[QSchemeType]*) – The quantization scheme to use, such as per_tensor, per_channel or per_group. Default is None.
- **ch_axis** (*Optional[int]*) – The channel axis for per-channel quantization. Default is None.
- **group_size** (*Optional[int]*) – The size of the group for per-group quantization. Default is None.
- **symmetric** (*Optional[bool]*) – Indicates if the quantization should be symmetric around zero. If True, quantization is symmetric. If None, it defers to a higher-level or global setting. Default is None.
- **round_method** (*Optional[RoundType]*) – The rounding method during quantization, such as half_even. If None, it defers to a higher-level or default method. Default is None.
- **scale_type** (*Optional[ScaleType]*) – Defines the scale type to be used for quantization, like power of two or float. If None, it defers to a higher-level setting or uses a default method. Default is None.

class quark.torch.quantization.config.config.**SmoothQuantConfig**

A data class that defines the specifications for Smooth Quantization.

Parameters

- **name** (*str*) – The name of the configuration, typically used to identify different quantization settings. Default is "smoothquant".
- **alpha** (*int*) – The factor of adjustment in the quantization formula, influencing how aggressively weights are quantized. Default is 1.
- **scale_clamp_min** (*float*) – The minimum scaling factor to be used during quantization, preventing the scale from becoming too small. Default is 1e-3.
- **scaling_layers** (*Optional[List[Dict[str, str]]]*) – Specific settings for scaling layers, allowing customization of quantization parameters for different layers within the model. Default is None.
- **embedding_layers** (*Optional[List[str]]*) – A list of embedding layer names that require special quantization handling to maintain their performance and accuracy. Default is None.
- **model_decoder_layers** (*Optional[str]*) – Specifies any particular decoder layers in the model that might have unique quantization requirements. Default is None.

class quark.torch.quantization.config.config.**AWQConfig**

Configuration for Activation-aware Weight Quantization (AWQ).

Parameters

- **name** (*str*) – The name of the quantization configuration. Default is "awq".
- **bit** (*int*) – The bit width for weights, indicating the precision of quantization. Defaults to 4 bits.
- **sym** (*bool*) – Indicates whether symmetric quantization should be used. If True, quantization is symmetric around zero. Default is False.
- **group_size** (*int*) – The size of the group for grouped quantization, specifying how many weights are quantized together using the same scale and zero-point. Default is 128.
- **algo_processor** (*Type[AwqProcessor]*) – The processor type that handles the AWQ algorithm logic.
- **scaling_layers** (*Optional[List[Dict[str, str]]]*) – Configuration details for scaling layers within the model, specifying custom scaling parameters per layer. Default is None.

- **model_decoder_layers** (*Optional[str]*) – Specifies the layers involved in model decoding that may require different quantization parameters. Default is None.
- **embedding_layers** (*Optional[List[str]]*) – Lists the embedding layers within the model that need to be quantized separately. Default is None.

class quark.torch.quantization.config.config.**GPTQConfig**

A data class that defines the specifications for Accurate Post-Training Quantization for Generative Pre-trained Transformers (GPTQ).

Parameters

- **name** (*str*) – The configuration name. Default is "gptq".
- **bit** (*int*) – The bit width for quantization, indicating the precision level of the quantized values. Defaults to 4 bits.
- **sym** (*bool*) – Specifies whether symmetric quantization is used. Symmetric quantization centers the quantized values around zero. Default is False.
- **group_size** (*int*) – Specifies the number of weights to be quantized together as a group, using the same scale and zero-point. Default is 128.
- **damp_percent** (*float*) – The percentage used to dampen the quantization effect, aiding in the maintenance of accuracy post-quantization. Default is 0.01.
- **desc_act** (*bool*) – Indicates whether descending activation is used, typically to enhance model performance with quantization. Default is True.
- **static_groups** (*bool*) – Specifies whether the groups for quantization are static (fixed during initialization) or can be dynamically adjusted during training. Default is False.
- **true_sequential** (*bool*) – Indicates whether the quantization should be applied in a truly sequential manner across the layers. Default is True.
- **inside_layer_modules** (*Optional[List[str]]*) – Lists the names of internal layer modules within the model that require specific quantization handling. Default is None.
- **model_decoder_layers** (*Optional[str]*) – Specifies custom settings for quantization on specific decoder layers of the model. Default is None.
- **embedding_layers** (*Optional[List[str]]*) – Identifies which embedding layers within the model need to be quantized separately to preserve embedding quality. Default is None.

4.4 quark.torch.export.config.config

4.4.1 Module Contents

Classes

class quark.torch.export.config.config.**ExporterConfig**

A class that encapsulates comprehensive exporting configurations for a machine learning model, allowing for detailed control over exporting parameters across different exporting formats.

Parameters

- **json_export_config** (*Optional[JsonExporterConfig]*) – Global configuration for json-safetensors exporting.
- **onnx_export_config** (*Optional[OnnxExporterConfig]*) – Global configuration onnx exporting. Default is None.

class quark.torch.export.config.config.**JsonExporterConfig**

A data class that specifies configurations for json-safetensors exporting.

Parameters

weight_merge_groups (*Optional[List[List[str]]]*) – A list of operators group that share the same weight scaling factor. These operators' names should correspond to the original module names from the model. Additionally, wildcards can be used to denote a range of operators. Default is None.

class quark.torch.export.config.config.**OnnxExporterConfig**

A data class that specifies configurations for onnx exporting.

CHAPTER

FIVE

EXAMPLES

Quark for Pytorch

1. Language Model Quantization & Export

CHAPTER

SIX

FREQUENTLY ASKED QUESTIONS (FAQ)

6.1 Environment Issues

6.1.1 Issue 1

Windows CPU mode does not support fp16.

Solution:

Because of torch issue Windows CPU mode cannot perfectly support fp16.

6.2 C++ Compilation Issues

6.2.1 Issue 1

Stuck in the compilation phase for a long time (over ten minutes), the terminal shows like:

```
[QUARK-INFO]: Configuration checking start.

[QUARK-INFO]: C++ kernel build directory [cache folder path]/torch_
↪extensions/py39...
```

Solution:

delete the cache folder `[cache folder path]/torch_extensions` and run quark again.

PYTHON MODULE INDEX

q

INDEX

S