Quantized GEMM

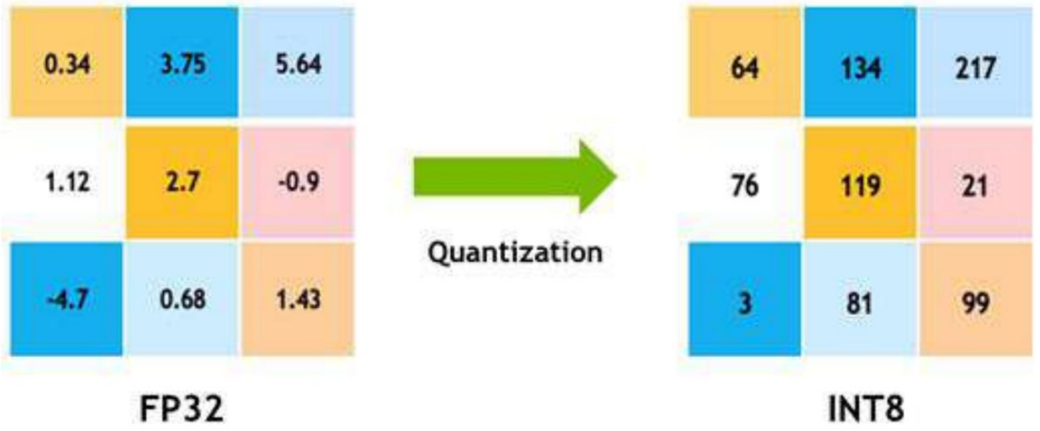
Student: Alexey Belkov
Mentor: Nikita Shapovalov

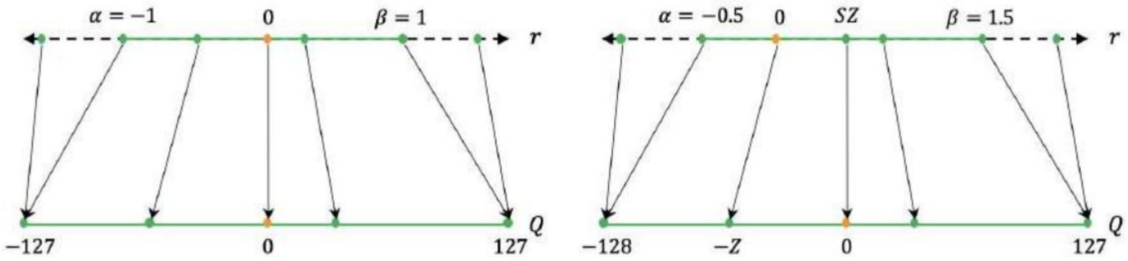
project repository: https://github.com/alexeybelkov/YSDA-CPU-inference/tree/pytorch

What does **Quantized** mean?

What is **GEMM**?

Quantization





Quantize:

$$Q(r) = \mathrm{Int}(r/S) - Z$$

$$S = \frac{\beta - \alpha}{2^b - 1}$$

$$Z = -\left(\frac{\alpha}{S} - \alpha_q\right)$$

Dequantize:

$$\tilde{r} = S(Q(r) + Z)$$

Quantization [static vs dynamic]

- Static quantization
  - Post training procedure
  - Activations are fused to layers if possible
  - Scaling factors are computed on the representative dataset
  - Suitable for CNNs *
- Dynamic quantization
  - On the fly during inference
  - Weights are converted to int8, activations are in full precision
  - Scaling factors are computed on the fly in full precision based on activations
  - Suitable for Transformers **
- Quantization aware training is out of scope of the project

For * and ** see https://pytorch.org/docs/stable/quantization.html#quantization-support-matrix

GEMM, GEneral Matrix Multiplication

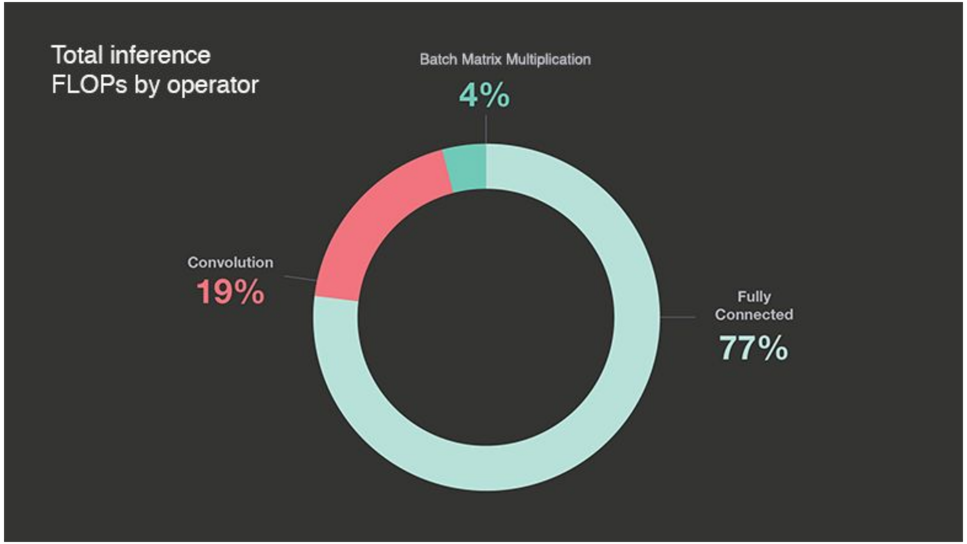
Basic Linear Algebra Subprograms (**BLAS**) has 3 levels:

1. **axpy** $\boldsymbol{y} \leftarrow \alpha \boldsymbol{x} + \boldsymbol{y}$
2. **gemv** $\boldsymbol{y} \leftarrow \alpha \boldsymbol{A} \boldsymbol{x} + \beta \boldsymbol{y}$
3. **gemm** $\boldsymbol{y} \leftarrow \alpha \boldsymbol{A} \boldsymbol{B} + \beta \boldsymbol{C}$

The purpose of the project

- Quantization
    - Less memory storage, consumes less energy (in theory)
    - Allows to run models on embedded devices, which sometimes only support integer data types.
    - Operations like **matrix multiplication** can be performed much faster with integer arithmetic.

The pie chart below shows the distribution of the deep learning inference FLOPs in Meta data centers



picture source: https://engineering.fb.com/2018/11/07/ml-applications/fbgemm/

The purpose of the project

- Fully Connected operators are just plain **GEMM**, so overall efficiency directly depends on **GEMM** efficiency

The main purpose of the project is to investigate quantized and non-quantized **GEMMs** in neural networks

Quantization experiments environment

Laptop with CPU Intel i7-8550U (8) @ 4.000GHz , 16 GB RAM

Python 3.10.12

PyTorch 2.1.0

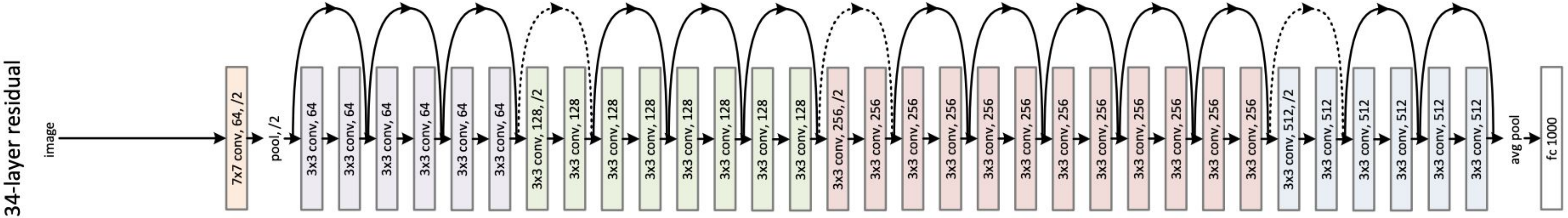
fbgemm backend

```
ATen/Parallel:
	at::get_num_threads() : 4
	at::get_num_interop_threads() : 4
OpenMP 201511 (a.k.a. OpenMP 4.5)
	omp_get_max_threads() : 4
Intel(R) oneAPI Math Kernel Library Version 2022.2-Product Build 20220804 for Intel(R) 64 architecture applications
	mkl_get_max_threads() : 4
Intel(R) MKL-DNN v3.1.1 (Git Hash 64f6bcbcbab628e96f33a62c3e975f8535a7bde4)
std::thread::hardware_concurrency() : 8
Environment variables:
	OMP_NUM_THREADS : [not set]
	MKL_NUM_THREADS : [not set]
ATen parallel backend: OpenMP
```

https://pytorch.org/blog/int8-quantization/ as a reference for benchamrking

Quantization experiments, CNN

Pretrained **ResNet34**, Post Training Static int-8 Quantization
https://pytorch.org/vision/stable/models/generated/torchvision.models.resnet34.html#torchvision.models.resnet34



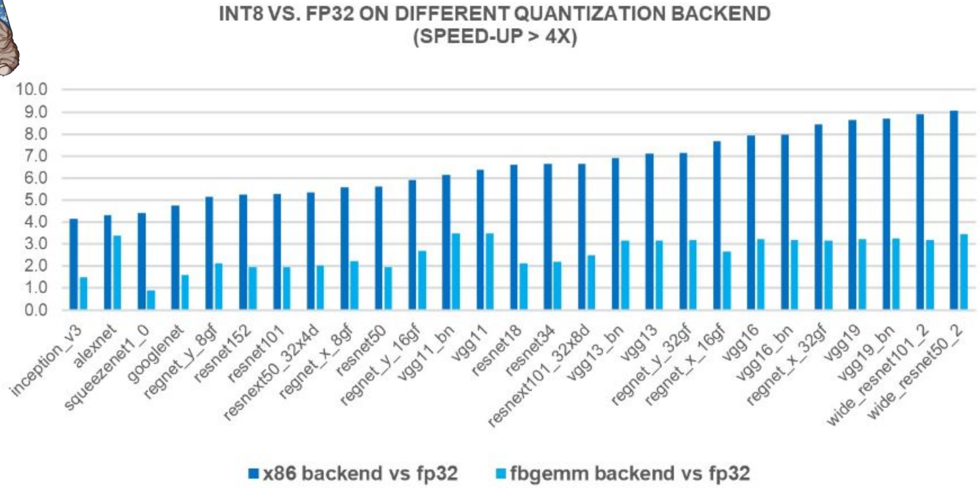
Dataset: **Imagenette** - a smaller subset of 10 easily classified classes from Imagenet, 3925 items
https://huggingface.co/datasets/frgfm/imagenette

ResNet34, Post Training Static int-8 Quantization, Per Channel Affine

Size compression: **81.1 MB** to **20.2 MB, ~4**x

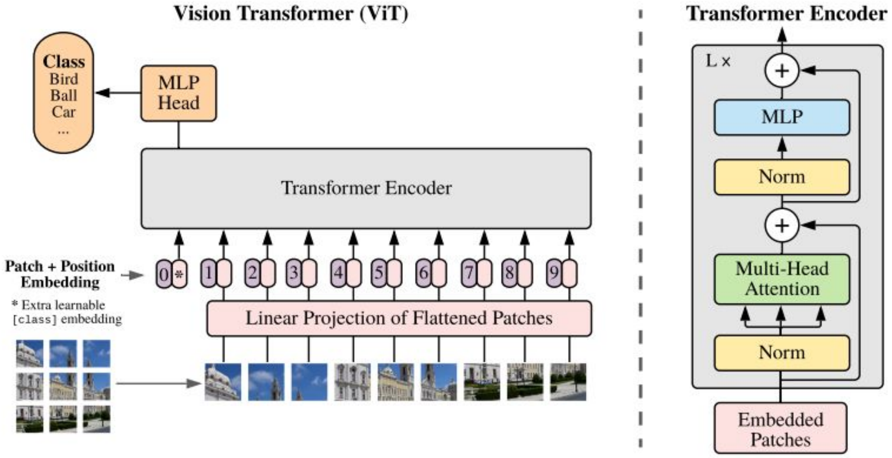
Speed-up: **4:40** min ⟶ **2:18** min, **2**x

Accuracy drop (**?**): **81.3%** ⟶**82%**



Quantization experiments, Transformer

Pretrained **ViT_B_16**, Dynamic int-8 quantization



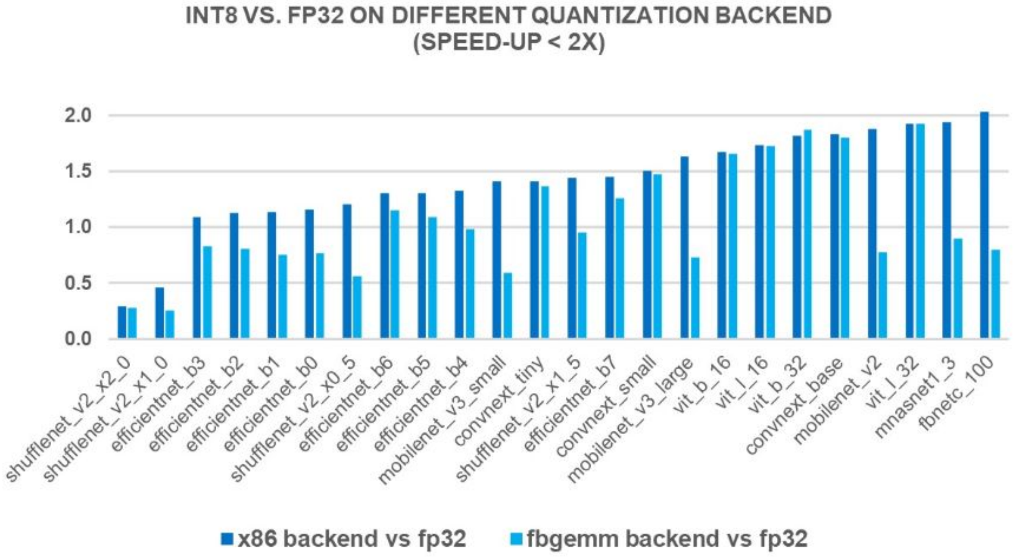
Dataset: Same as for **ResNet34**

ViT_B_16, Dynamic int-8 quantization, per tensor affine scheme

Size compression: **330.2 MB** to **111.1 MB, ~3**x

Speed-up: ~**17** min ⟶ ~**12** min, **1.4**x

Accuracy drop: **91.4**% ⟶ **89**%



Now let's take a closer look with [PyTorch Profiler]

FP32 ResNet34 linear layer profiling

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	# of Calls
model_inference	23.49%	156.178ms	100.00%	664.999ms	169.426us	3925
aten::linear	3.02%	20.079ms	76.23%	506.899ms	129.146us	3925
aten::addmm	63.28%	420.826ms	68.13%	453.065ms	115.431us	3925
aten::t	1.94%	12.869ms	5.36%	35.677ms	9.090us	3925
aten::transpose	2.54%	16.894ms	3.41%	22.696ms	5.782us	3925
aten::copy_	2.56%	17.010ms	2.56%	17.010ms	4.334us	3925
aten::expand	2.19%	14.576ms	2.29%	15.199ms	3.872us	3925
aten::as_strided	0.98%	6.537ms	0.98%	6.537ms	0.833us	7850
aten::resolve_conj	0.00%	30.000us	0.00%	30.000us	0.004us	7850

Self CPU time total: 664.999ms

Int8 ResNet34 QuantizedLinear layer profiling

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	# of Calls
model_inference	58.16%	330.290ms	100.00%	567.864ms	144.679us	3925
quantized::linear	28.00%	159.015ms	30.78%	174.778ms	44.529us	3925
aten::quantize_per_tensor	4.84%	27.488ms	5.83%	33.131ms	8.441us	3925
aten::dequantize	2.96%	16.804ms	4.11%	23.357ms	5.951us	3925
aten::item	2.08%	11.796ms	2.10%	11.948ms	0.761us	15700
aten::empty	1.98%	11.236ms	1.98%	11.236ms	1.431us	7850
aten::resize_	1.06%	6.014ms	1.06%	6.014ms	1.532us	3925
aten::_empty_affine_quantized	0.88%	5.006ms	0.88%	5.006ms	1.275us	3925
aten::_local_scalar_dense	0.03%	155.000us	0.03%	155.000us	0.010us	15700
aten::q_scale	0.01%	35.000us	0.01%	35.000us	0.009us	3925
aten::q_zero_point	0.00%	25.000us	0.00%	25.000us	0.006us	3925

Self CPU time total: 567.864ms

Int8 ViT_B_16 DynamicQuantizedLinear layer

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	# of Calls
model_inference	2.98%	298.000us	100.00%	10.000ms	10.000ms	1
quantized::linear_dynamic	92.37%	9.237ms	97.01%	9.701ms	9.701ms	1
aten::empty_like	4.45%	445.000us	4.51%	451.000us	451.000us	1
aten::empty	0.19%	19.000us	0.19%	19.000us	9.500us	2
aten::to	0.01%	1.000us	0.01%	1.000us	1.000us	1

Self CPU time total: 10.000ms

aten::addmm ?
aten - A TENsor library

TORCH.ADDMM

`torch.addmm(input, mat1, mat2, *, beta=1, alpha=1, out=None) → Tensor`

Performs a matrix multiplication of the matrices `mat1` and `mat2`. The matrix `input` is added to the final result.

If `mat1` is a $(n \times m)$ tensor, `mat2` is a $(m \times p)$ tensor, then `input` must be broadcastable with a $(n \times p)$ tensor and `out` will be a $(n \times p)$ tensor.

`alpha` and `beta` are scaling factors on matrix-vector product between `mat1` and `mat2` and the added matrix `input` respectively.

$$\text{out} = \beta\ \text{input} + \alpha\ (\text{mat1}_i\ @\ \text{mat2}_i)$$

This looks familiar to **gemm** $\boldsymbol{y} \leftarrow \alpha \boldsymbol{A}\boldsymbol{B} + \beta \boldsymbol{C}$

Let’s go deeper, PyTorch anatomy

android	Add TorchFix to the CI (#113403)	last month
aten	[1/N] Use std::in_place (#115170)	19 hours ago
benchmarks	[CI] Fix a missing write_csv_when_exception problem (#11…	yesterday
binaries	Remaining replacement of c10::stoi with std::stoi (#109482)	3 months ago
c10	Increased hardcoded limit for number of GPUs. (#115368)	5 hours ago
caffe2	Revert "[Reland2] Update NVTX to NVTX3 (#109843)"	4 days ago
cmake	Revert "[Reland2] Update NVTX to NVTX3 (#109843)"	4 days ago
docs	[docs, dynamo] fix typos in dynamo custom backend docs (…	yesterday
fb/vulkan/tests/perf_benchmark	[PyTorch][Vulkan] Refactor performance test binary (#114712)	5 days ago
functorch	Add python and C++ support for LPPool3d (#114199)	yesterday
ios	Add TorchFix to the CI (#113403)	last month
modules	[Cmake] Check that gcc-9.4 or newer is used (#112858)	last month
mypy_plugins	Enable UFMT on a bunch of low traffic Python files outside …	5 months ago
scripts	[ONNX] Add sanity check in CI for onnxbench (#110178)	last week
test	[AOTI] move model runner into a library (#115220)	4 hours ago
third_party	[cuDNN][cuDNN frontend] Bump `cudnn_frontend` submodu…	2 days ago
tools	[torchgen] Add logic in custom ops to return empty tensor (…	yesterday
torch	[BE][JIT] Do not wrap shared_ptr with optional (#115473)	2 hours ago
torchgen	[torchgen] Add logic in custom ops to return empty tensor (…	yesterday

Let’s go deeper, PyTorch anatomy

> We want to investigate internal structure of **nn.Linear** layer or, which is the same thing, **torch.linear**

> We know that PyTorch is basically a C++ $\longleftrightarrow$Python bindings

> We can profile C++ analogue **torch::linear** from **C++ libtorch** with **gprof** or use **debugger**

PyTorch for C++

- **Default LibTorch guide** https://pytorch.org/cppdocs/installing.html **?**
- **Problems**:
  - **gprof** gives very high-level info
  - No debug symbols, so you'll stuck in call of dispatcher for high level function, like **at::linear** which is basically an alias for **torch::linear**
  - You can even call **at::addmm** but you'll also get stuck in dispatcher call for it

```cpp
#include <ATen/ATen.h>
#include <iostream>
#include <torch/torch.h>

int main() {
    at::Tensor weight = at::rand({64, 256}, at::requires_grad(false));
    at::Tensor input = at::rand({128, 256}, at::requires_grad(false));
    at::Tensor output = at::linear(input, weight);
    std::cout << output.size(0);
}
```

- **We found out that we can build libtorch.so directly with cmake**
  - **gprof** still can’t go low level
  - We will call **at::linear** and debug it with **GDB**
  - ```
    static void addmm_impl_cpu_(
        Tensor &result, const Tensor &self, Tensor m1, Tensor m2, const Scalar& beta, const Scalar& alpha) {
    ```

C++ config

In this branch we used directly builded libtorch as in [Building libtorch using CMake](#)
We builded it in Debug mode, to do this the one needs to run the following commands in **/cpp** folder.

> ⚠ Warning
>
> Overall build will require a little less than 23 GB of disk space and about 14 GB of CPU RAM

```
git clone -b main --recurse-submodule https://github.com/pytorch/pytorch.git
mkdir pytorch-build
cd pytorch-build
cmake -DBUILD_SHARED_LIBS:BOOL=ON -DCMAKE_BUILD_TYPE:STRING=Debug -DPYTHON_EXECUTABLE:PATH=`wh:
cmake --build . --target install
```

Screenshot from a course project repo: https://github.com/alexeybelkov/YSDA-CPU-inference/tree/pytorch
```

At last, the main course is on the table

We will call **at::linear** and debug it with **GDB**

Backtrace

```
#0  at::native::addmm_impl_cpu_ (result=..., self=..., m1=..., m2=..., beta=..., alpha=...) at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch/aten/src/ATen/native/LinearAlgebra.cpp:1513
#1  0x00007fffe40c80d1 in at::native::structured_mm_out_cpu::impl (this=0x7fffffffc440, self=..., mat2=..., result=...)
    at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch/aten/src/ATen/native/LinearAlgebra.cpp:1617
#2  0x00007fffe5b0cdd6 in at::(anonymous namespace)::wrapper_CPU_mm (self=..., mat2=...) at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch-build/aten/src/ATen/RegisterCPU.cpp:8643
#3  0x00007fffe5ceb6b6 in c10::impl::detail::WrapFunctionIntoFunctor_<c10::CompileTimeFunctionPointer<at::Tensor(const at::Tensor&, const at::Tensor&), at::(anonymous namespace)::wrapper_CPU_mm>, at::T
ensor, c10::guts::typelist::typelist<const at::Tensor&, const at::Tensor&> >::operator() (args#1=..., args#0=..., this=0x55555604ba60)
    at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch/aten/src/ATen/core/boxing/impl/WrapFunctionIntoFunctor.h:13
#4  c10::impl::wrap_kernel_functor_unboxed_<c10::impl::detail::WrapFunctionIntoFunctor_<c10::CompileTimeFunctionPointer<at::Tensor(const at::Tensor&, const at::Tensor&), at::(anonymous namespace)::wrap
per_CPU_mm>, at::Tensor, c10::guts::typelist::typelist<const at::Tensor&, const at::Tensor&> >, at::Tensor(const at::Tensor&, const at::Tensor&)>::call(c10::OperatorKernel *, c10::DispatchKeySet, const
 at::Tensor &, const at::Tensor &) (functor=0x55555604ba60, args#0=..., args#1=...)
    at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch/aten/src/ATen/core/boxing/impl/make_boxed_from_unboxed_functor.h:468
#5  0x00007fffe4dcf95a in c10::callUnboxedKernelFunction<at::Tensor, at::Tensor const&, at::Tensor const&> (
    unboxed_kernel_func=0x7fffe5ceb61d <c10::impl::wrap_kernel_functor_unboxed_<c10::impl::detail::WrapFunctionIntoFunctor_<c10::CompileTimeFunctionPointer<at::Tensor(const at::Tensor&, const at::Tenso
r&), at::(anonymous namespace)::wrapper_CPU_mm>, at::Tensor, c10::guts::typelist::typelist<const at::Tensor&, const at::Tensor&> >, at::Tensor(const at::Tensor&, const at::Tensor&)>::call(c10::Operator
Kernel *, c10::DispatchKeySet, const at::Tensor &, const at::Tensor &)>, functor=0x55555604ba60, dispatchKeySet=...)
    at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch/aten/src/ATen/core/boxing/KernelFunction_impl.h:50
#6  0x00007fffe4c7155f in c10::KernelFunction::call<at::Tensor, at::Tensor const&, at::Tensor const&> (dispatchKeySet=..., opHandle=..., this=0x5555555a6d48)
    at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch/aten/src/ATen/core/boxing/KernelFunction_impl.h:103
#7  c10::Dispatcher::redispatch<at::Tensor, at::Tensor const&, at::Tensor const&>(c10::TypedOperatorHandle<at::Tensor (at::Tensor const&, at::Tensor const&)> const&, c10::DispatchKeySet, at::Tensor con
st&, at::Tensor const&) const (this=0x7ffff7bd8040 <c10::Dispatcher::realSingleton()::_singleton>, op=..., currentDispatchKeySet=...)
    at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch/aten/src/ATen/core/dispatch/Dispatcher.h:690
#8  0x00007fffe561205e in c10::TypedOperatorHandle<at::Tensor (at::Tensor const&, at::Tensor const&)>::redispatch(c10::DispatchKeySet, at::Tensor const&, at::Tensor const&) const (args#1=...,
    args#0=..., currentDispatchKeySet=..., this=<optimized out>) at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch/aten/src/ATen/core/dispatch/Dispatcher.h:526
#9  at::_ops::mm::redispatch (dispatchKeySet=..., self=..., mat2=...) at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch-build/aten/src/ATen/Operators_3.cpp:3896
#10 0x00007fffe86c7c17 in at::redispatch::mm (dispatchKeySet=..., self=..., mat2=...) at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch-build/aten/src/ATen/RedispatchFunctions.h:5112
#11 0x00007fffe85dab1e in operator() (__closure=0x7fffffffc8e0) at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch/torch/csrc/autograd/generated/VariableType_3.cpp:12504
#12 0x00007fffe85db28d in torch::autograd::VariableType::(anonymous namespace)::mm (ks=..., self=..., mat2=...)
    at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch/torch/csrc/autograd/generated/VariableType_3.cpp:12505
#13 0x00007fffe86913ee in c10::impl::detail::WrapFunctionIntoFunctor_<c10::CompileTimeFunctionPointer<at::Tensor(c10::DispatchKeySet, const at::Tensor&, const at::Tensor&), torch::autograd::VariableTyp
e::(anonymous namespace)::mm>, at::Tensor, c10::guts::typelist::typelist<c10::DispatchKeySet, const at::Tensor&, const at::Tensor&> >::operator() (args#2=..., args#1=..., args#0=...,
    this=0x5555574f7370) at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch/aten/src/ATen/core/boxing/impl/WrapFunctionIntoFunctor.h:13
#14 c10::impl::wrap_kernel_functor_unboxed_<c10::impl::detail::WrapFunctionIntoFunctor_<c10::CompileTimeFunctionPointer<at::Tensor(c10::DispatchKeySet, const at::Tensor&, const at::Tensor&), torch::aut
ograd::VariableType::(anonymous namespace)::mm>, at::Tensor, c10::guts::typelist::typelist<c10::DispatchKeySet, const at::Tensor&, const at::Tensor&> >, at::Tensor(c10::DispatchKeySet, const at::Tensor
&, const at::Tensor&)>::call(c10::OperatorKernel *, c10::DispatchKeySet, const at::Tensor &, const at::Tensor &) (functor=0x5555574f7370, dispatchKeySet=..., args#0=..., args#1=...)
    at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch/aten/src/ATen/core/boxing/impl/make_boxed_from_unboxed_functor.h:485
#15 0x00007fffe4dcf95a in c10::callUnboxedKernelFunction<at::Tensor, at::Tensor const&, at::Tensor const&> (
    unboxed_kernel_func=0x7fffe8691339 <c10::impl::wrap_kernel_functor_unboxed_<c10::impl::detail::WrapFunctionIntoFunctor_<c10::CompileTimeFunctionPointer<at::Tensor(c10::DispatchKeySet, const at::Ten
sor&, const at::Tensor&), torch::autograd::VariableType::(anonymous namespace)::mm>, at::Tensor, c10::guts::typelist::typelist<c10::DispatchKeySet, const at::Tensor&, const at::Tensor&> >, at::Tensor(c
10::DispatchKeySet, const at::Tensor&, const at::Tensor&)>::call(c10::OperatorKernel *, c10::DispatchKeySet, const at::Tensor &, const at::Tensor &)>, functor=0x5555574f7370, dispatchKeySet=...)
    at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch/aten/src/ATen/core/boxing/KernelFunction_impl.h:50
#16 0x00007fffe5611e16 in c10::KernelFunction::call<at::Tensor, at::Tensor const&, at::Tensor const&> (dispatchKeySet=..., opHandle=..., this=0x5555555a7728)
    at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch/aten/src/ATen/core/boxing/KernelFunction_impl.h:103
#17 c10::Dispatcher::call<at::Tensor, at::Tensor const&, at::Tensor const&>(c10::TypedOperatorHandle<at::Tensor (at::Tensor const&, at::Tensor const&)> const&, at::Tensor const&, at::Tensor const&) con
st (op=..., this=0x7ffff7bd8040 <c10::Dispatcher::realSingleton()::_singleton>) at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch/aten/src/ATen/core/dispatch/Dispatcher.h:673
#18 c10::TypedOperatorHandle<at::Tensor (at::Tensor const&, at::Tensor const&)>::call(at::Tensor const&, at::Tensor const&) const (args#1=..., args#0=..., this=<optimized out>)
    at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch/aten/src/ATen/core/dispatch/Dispatcher.h:521
#19 at::_ops::mm::call (self=..., mat2=...) at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch-build/aten/src/ATen/Operators_3.cpp:3889
#20 0x00007fffe40de0ea in at::Tensor::mm (this=0x7fffffffd528, mat2=...) at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch-build/aten/src/ATen/core/TensorBody.h:2991
#21 0x00007fffe40cbe51 in at::native::_matmul_impl (out=..., tensor1=..., tensor2=...) at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch/aten/src/ATen/native/LinearAlgebra.cpp:1996
#22 0x00007fffe40cd413 in at::native::matmul (tensor1=..., tensor2=...) at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch/aten/src/ATen/native/LinearAlgebra.cpp:2144
#23 0x00007fffe61bc23c in at::(anonymous namespace)::(anonymous namespace)::wrapper_CompositeImplicitAutograd__matmul (self=..., other=...)
    at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch-build/aten/src/ATen/RegisterCompositeImplicitAutograd.cpp:2753
#24 0x00007fffe62c11dc in c10::impl::detail::WrapFunctionIntoFunctor_<c10::CompileTimeFunctionPointer<at::Tensor(const at::Tensor&, const at::Tensor&), at::(anonymous namespace)::(anonymous namespace):
:wrapper_CompositeImplicitAutograd__matmul>, at::Tensor, c10::guts::typelist::typelist<const at::Tensor&, const at::Tensor&> >::operator() (args#1=..., args#0=..., this=0x555556901550)
    at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch/aten/src/ATen/core/boxing/impl/WrapFunctionIntoFunctor.h:13
#25 c10::impl::wrap_kernel_functor_unboxed_<c10::impl::detail::WrapFunctionIntoFunctor_<c10::CompileTimeFunctionPointer<at::Tensor(const at::Tensor&, const at::Tensor&), at::(anonymous namespace)::(ano
                                              nymous namespace)::wrapper_CompositeImplicitAutograd__matmul>, at::Tensor, c10::guts::typelist::typelist<const at::Tensor&, const at::Tensor&> >, at::Ten
sor(const at::Tensor&, const at::Tensor&)>::call
```

Backtrace

```
#26 0x00007fffe4dcf95a in c10::callUnboxedKernelFunction<at::Tensor, at::Tensor const&, at::Tensor const&> (
    unboxed_kernel_func=0x7fffe62c1143 <c10::impl::wrap_kernel_functor_unboxed_<c10::impl::detail::WrapFunctionIntoFunctor_<c10::CompileTimeFunctionPointer<at::Tensor(const at::Tensor&, const at::Tenso
r&), at::(anonymous namespace)::(anonymous namespace)::wrapper_CompositeImplicitAutograd__matmul>, at::Tensor, c10::guts::typelist::typelist<const at::Tensor&, const at::Tensor&> >, at::Tensor(const at
::Tensor&, const at::Tensor&)>::call(c10::OperatorKernel *, c10::DispatchKeySet, const at::Tensor &, const at::Tensor &)>, functor=0x555556901550, dispatchKeySet=...) at /home/alexey/YSDA/YSDA-CPU-infe
rence/cpp/pytorch/aten/src/ATen/core/boxing/KernelFunction_impl.h:50
#27 0x00007fffe5898682 in c10::KernelFunction::call<at::Tensor, at::Tensor const&, at::Tensor const&> (dispatchKeySet=..., opHandle=..., this=0x55555565b9d8)
    at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch/aten/src/ATen/core/boxing/KernelFunction_impl.h:103
#28 c10::Dispatcher::call<at::Tensor, at::Tensor const&, at::Tensor const&>(c10::TypedOperatorHandle<at::Tensor (at::Tensor const&, at::Tensor const&)> const&, at::Tensor const&, at::Tensor const&) con
st (op=...,
    this=0x7ffff7bd8040 <c10::Dispatcher::realSingleton()::_singleton>) at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch/aten/src/ATen/core/dispatch/Dispatcher.h:673
#29 c10::TypedOperatorHandle<at::Tensor (at::Tensor const&, at::Tensor const&)>::call(at::Tensor const&, at::Tensor const&) const (args#1=..., args#0=..., this=<optimized out>)
    at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch/aten/src/ATen/core/dispatch/Dispatcher.h:521
#30 at::_ops::matmul::call (self=..., other=...) at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch-build/aten/src/ATen/Operators_4.cpp:3052
#31 0x00007fffe39bd781 in at::matmul (self=..., other=...) at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch-build/aten/src/ATen/ops/matmul.h:27
#32 0x00007fffe40a1e0b in at::native::linear (input=..., weight=..., bias_opt=...) at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch/aten/src/ATen/native/Linear.cpp:106
#33 0x00007fffe61bbba3 in at::(anonymous namespace)::(anonymous namespace)::wrapper_CompositeImplicitAutograd__linear (input=..., weight=..., bias=...)
    at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch-build/aten/src/ATen/RegisterCompositeImplicitAutograd.cpp:2620
#34 0x00007fffe62bf064 in c10::impl::detail::WrapFunctionIntoFunctor_<c10::CompileTimeFunctionPointer<at::Tensor(const at::Tensor&, const at::Tensor&, const c10::optional<at::Tensor>&), at::(anonymous
namespace)::(anonymous namespace)::wrapper_CompositeImplicitAutograd__linear>, at::Tensor, c10::guts::typelist::typelist<const at::Tensor&, const at::Tensor&, const c10::optional<at::Tensor>&> >::opera
tor() (args#2=..., args#1=..., args#0=..., this=0x5555568ef350)
    at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch/aten/src/ATen/core/boxing/impl/WrapFunctionIntoFunctor.h:13
#35 c10::impl::wrap_kernel_functor_unboxed_<c10::impl::detail::WrapFunctionIntoFunctor_<c10::CompileTimeFunctionPointer<at::Tensor(const at::Tensor&, const at::Tensor&, const c10::optional<at::Tensor>&
), at::(anonymous namespace)::(anonymous namespace)::wrapper_CompositeImplicitAutograd__linear>, at::Tensor, c10::guts::typelist::typelist<const at::Tensor&, const at::Tensor&, const c10::optional<at::
Tensor>&> >, at::Tensor(const at::Tensor&, const at::Tensor&, const c10::optional<at::Tensor>&)>::call(c10::OperatorKernel *, c10::DispatchKeySet, const at::Tensor &, const at::Tensor &, const c10::opt
ional<at::Tensor> &) (functor=0x5555568ef350, args#0=..., args#1=..., args#2=...)
    at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch/aten/src/ATen/core/boxing/impl/make_boxed_from_unboxed_functor.h:468
#36 0x00007fffe4de1f5e in c10::callUnboxedKernelFunction<at::Tensor, at::Tensor const&, at::Tensor const&, c10::optional<at::Tensor> const&> (
    unboxed_kernel_func=0x7fffe62befa0 <c10::impl::wrap_kernel_functor_unboxed_<c10::impl::detail::WrapFunctionIntoFunctor_<c10::CompileTimeFunctionPointer<at::Tensor(const at::Tensor&, const at::Tenso
r&, const c10::optional<at::Tensor>&), at::(anonymous namespace)::(anonymous namespace)::wrapper_CompositeImplicitAutograd__linear>, at::Tensor, c10::guts::typelist::typelist<const at::Tensor&, const a
t::Tensor&, const c10::optional<at::Tensor>&> >, at::Tensor(const at::Tensor&, const at::Tensor&, const c10::optional<at::Tensor>&)>::call(c10::OperatorKernel *, c10::DispatchKeySet, const at::Tensor &
, const at::Tensor &, const c10::optional<at::Tensor> &)>, functor=0x5555568ef350, dispatchKeySet=...)
    at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch/aten/src/ATen/core/boxing/KernelFunction_impl.h:50
#37 0x00007fffe4afe192 in c10::KernelFunction::call<at::Tensor, at::Tensor const&, at::Tensor const&, c10::optional<at::Tensor> const&> (dispatchKeySet=..., opHandle=..., this=0x5555557484e8)
    at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch/aten/src/ATen/core/boxing/KernelFunction_impl.h:103
#38 c10::Dispatcher::call<at::Tensor, at::Tensor const&, at::Tensor const&, c10::optional<at::Tensor> const&>(c10::TypedOperatorHandle<at::Tensor (at::Tensor const&, at::Tensor const&, c10::optional<at
::Tensor> const&)> const&, at::Tensor const&, at::Tensor const&, c10::optional<at::Tensor> const&) const (op=..., this=0x7ffff7bd8040 <c10::Dispatcher::realSingleton()::_singleton>) at /home/alexey/YSD
A/YSDA-CPU-inference/cpp/pytorch/aten/src/ATen/core/dispatch/Dispatcher.h:673
#39 c10::TypedOperatorHandle<at::Tensor (at::Tensor const&, at::Tensor const&, c10::optional<at::Tensor> const&)>::call(at::Tensor const&, at::Tensor const&, c10::optional<at::Tensor> const&) const (ar
gs#2=..., args#1=..., args#0=...,
    this=<optimized out>) at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch/aten/src/ATen/core/dispatch/Dispatcher.h:521
#40 at::_ops::linear::call (input=..., weight=..., bias=...) at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch-build/aten/src/ATen/Operators_0.cpp:3601
#41 0x000055555555aa8a in at::linear (input=..., weight=..., bias=...) at /home/alexey/YSDA/YSDA-CPU-inference/cpp/pytorch-install/include/ATen/ops/linear.h:27
#42 0x00005555555578e0 in main () at /home/alexey/YSDA/YSDA-CPU-inference/cpp/src/linear.cpp:17
```

Finally, GEMMs

>

```
// Apply BLAS routine
_AT_DISPATCH_ADDMM_TYPES(result.scalar_type(), "addmm_impl_cpu_", [&]{
        using opmath_t = at::opmath_type<scalar_t>;
        at::native::cpublas::gemm(
            transpose_a ? a.is_conj() ? TransposeType::ConjTranspose : TransposeType::Transpose : TransposeType::NoTranspose,
            transpose_b ? b.is_conj() ? TransposeType::ConjTranspose : TransposeType::Transpose : TransposeType::NoTranspose,
            m, n, k,
            alpha.to<opmath_t>(),
            a.const_data_ptr<scalar_t>(), lda,
            b.const_data_ptr<scalar_t>(), ldb,
            beta.to<opmath_t>(),
            c.mutable_data_ptr<scalar_t>(), ldc);
      });
}
```

>

```
void gemm(
    TransposeType transa, TransposeType transb,
    int64_t m, int64_t n, int64_t k,
    const float alpha,
    const float *a, int64_t lda,
    const float *b, int64_t ldb,
    const float beta,
    float *c, int64_t ldc) {
```

GEMM

```
void gemm(
    TransposeType transa, TransposeType transb,
    int64_t m, int64_t n, int64_t k,
    const float alpha,
    const float *a, int64_t lda,
    const float *b, int64_t ldb,
    const float beta,
    float *c, int64_t ldc) {
  internal::normalize_last_dims(transa, transb, m, n, k, &lda, &ldb, &ldc);
#if AT_BUILD_WITH_BLAS()
  if (use_blas_gemm(transa, transb, m, n, k, lda, ldb, ldc)) {
    int m_ = m, n_ = n, k_ = k, lda_ = lda, ldb_ = ldb, ldc_ = ldc;
    float alpha_ = alpha, beta_ = beta;
    #if C10_IOS
    CBLAS_TRANSPOSE transa_ = to_apple_accelerate_transpose(transa);
    CBLAS_TRANSPOSE transb_ = to_apple_accelerate_transpose(transb);
    cblas_sgemm(CblasColMajor,
      transa_, transb_,
      m_, n_, k_,
      alpha_,
      a, lda_,
      b, ldb_,
      beta_,
      c, ldc_);
    #else
    char transa_ = to_blas(transa), transb_ = to_blas(transb);
    sgemm_(
        &transa_, &transb_,
        &m_, &n_, &k_,
        &alpha_,
        a, &lda_,
        b, &ldb_,
        &beta_,
        c, &ldc_);
    #endif
    return;
  }
#endif
  gemm_stub(
      at::kCPU, at::kFloat,
      transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc);
}
```

```
void cpublas_gemm_impl(
    at::ScalarType type,
    TransposeType transa, TransposeType transb,
    int64_t m, int64_t n, int64_t k,
    const Scalar& alpha,
    const void *a, int64_t lda,
    const void *b, int64_t ldb,
    const Scalar& beta,
    void *c, int64_t ldc) {
  _AT_DISPATCH_GEMM_TYPES(type, "cpublas_gemm_impl", [&]{
        using opmath_t = at::opmath_type<scalar_t>;
        gemm_core_(
            transa, transb, m, n, k,
            alpha.to<opmath_t>(),
            static_cast<const scalar_t *>(a), lda,
            static_cast<const scalar_t *>(b), ldb,
            beta.to<opmath_t>(),
            static_cast<scalar_t *>(c), ldc);
      });
}
```

Many special cases are considered

```
template <typename scalar_t, typename opmath_t>
void gemm_core_(
    TransposeType transa, TransposeType transb,
    int64_t m, int64_t n, int64_t k,
    opmath_t alpha,
    const scalar_t *a, int64_t lda,
    const scalar_t *b, int64_t ldb,
    opmath_t beta,
    scalar_t *c, int64_t ldc) {
  if (transa == TransposeType::NoTranspose &&
      transb == TransposeType::NoTranspose) {
    return gemm_notrans_(m, n, k, alpha, a, lda, b, ldb, beta, c, ldc);
  } else if (
      transa == TransposeType::Transpose &&
      transb != TransposeType::Transpose) {
    gemm_transa_(m, n, k, alpha, a, lda, b, ldb, beta, c, ldc);
  } else if (
      transa == TransposeType::NoTranspose &&
      transb == TransposeType::Transpose) {
    gemm_transb_(m, n, k, alpha, a, lda, b, ldb, beta, c, ldc);
  } else { // transa == TransposeType::Transpose && transb ==
           // TransposeType::Transpose
    gemm_transab_(m, n, k, alpha, a, lda, b, ldb, beta, c, ldc);
  }
}
```

FP32 Linear GEMM

```
template <typename scalar_t, typename opmath_t>
void gemm_transa_(
    int64_t m, int64_t n, int64_t k,
    opmath_t alpha,
    const scalar_t *a, int64_t lda,
    const scalar_t *b, int64_t ldb,
    opmath_t beta,
    scalar_t *c, int64_t ldc) {
  // c = alpha * (a.T @ b) + beta * c
  const scalar_t *a_ = a;
  for (const auto i : c10::irange(m)) {
    const scalar_t *b_ = b;
    for (const auto j : c10::irange(n)) {
      const auto dot = sum(k, [&](int64_t l) -> opmath_t {
        return static_cast<opmath_t>(a_[l]) * static_cast<opmath_t>(b_[l]);
      });
      b_ += ldb;
      if (beta == opmath_t(0)) {
        c[j*ldc+i] = alpha*dot;
      } else {
        c[j*ldc+i] = beta*c[j*ldc+i]+alpha*dot;
      }
    }
    a_ += lda;
  }
}
```

```
template <typename Func>
auto sum(int64_t N, Func f) {
  constexpr int ilp_factor = 4;
  using acc_t = decltype(f(0));

  // Calculate independent partial sums then add together at the end
  std::array<acc_t, ilp_factor> partial_sums{};

  int64_t i = 0;
  for (; i + ilp_factor <= N; i += ilp_factor) {
    c10::ForcedUnroll<ilp_factor>{}([&](int k) {
      partial_sums[k] += f(i + k);
    });
  }
  for (; i < N; ++i) {
    partial_sums[0] += f(i);
  }
  for (int k = 1; k < ilp_factor; ++k) {
    partial_sums[0] += partial_sums[k];
  }
  return partial_sums[0];
}
```

Well, where is my ultra-high-performance **parallel** computing?

QuantizedLinear GEMM

> Calling **quantized::linear**

> In **libtorch**, it is impossible to blindly call at least something close in name

> We can use **torch.jit.script** to transfer model from Python to C++

```
jit = torch.jit.script(quantized_model)
print(jit.code)
torch.jit.save(jit, f'fbgemm_linear_{n}x{m}.pt')
```

✓ 0.0s

```
def forward(self,
    input: Tensor) -> Tensor:
  _input_scale_0 = self._input_scale_0
  _input_zero_point_0 = self._input_zero_point_0
  quantize_per_tensor = torch.quantize_per_tensor(input, _input_scale_0, _input_zero_point_0, 13)
  _packed_weight_0 = self._packed_weight_0
  _scale_1 = self._scale_1
  _zero_point_1 = self._zero_point_1
  linear = ops.quantized.linear(quantize_per_tensor, _packed_weight_0, annotate(float, _scale_1), annotate(int, _zero_point_1))
  return torch.dequantize(linear)
```

Debugging…

>

```
at::Tensor weight = at::rand({256, 512}, at::requires_grad(false));
at::Tensor bias = at::rand({256}, at::requires_grad(false));
at::Tensor input = at::rand({256, 512}, at::requires_grad(false));

torch::jit::Module fbgemm_linear = torch::jit::load("../../gitignore/jit_models/fbgemm_linear_256x512.pt")
std::vector<torch::jit::IValue> jit_input = {input};
auto out = fbgemm_linear.forward(jit_input);
```

>

```
template <bool ReluFused>
class QLinearInt8 final {
 public:
  static at::Tensor run(
      at::Tensor input,
      const c10::intrusive_ptr<LinearPackedParamsBase>& packed_weight,
      double output_scale,
      int64_t output_zero_point) {
    if (ReluFused) {
      return packed_weight->apply_relu(
          std::move(input), output_scale, output_zero_point);
    } else {
      return packed_weight->apply(
          std::move(input), output_scale, output_zero_point);
    }
  }
};
```

Then we come to such a thing

> fbgemm? packA? packB?

```
// Do the GEMM
fbgemm::fbgemmPacked(
    /*packA=*/packA,
    /*packB=*/*packB,
    /*C=*/reinterpret_cast<uint8_t*>(output.data_ptr<c10::quint8>()),
    /*C_buffer=*/buffer.data_ptr<int32_t>(),
    /*ldc=*/N,
    /*outProcess=*/outputProcObj,
    /*thread_id=*/task_id,
    /*num_threads=*/num_tasks);
```

FBGEMM: Enabling High-Performance Low-Precision Deep Learning Inference

Daya Khudia, Jianyu Huang, Protonu Basu, Summer Deng, Haixin Liu,
Jongsoo Park, Mikhail Smelyanskiy
Facebook, Inc.

A FBGEMM INTERFACE

FBGEMM is a C++ library, and the following code listing shows the GEMM interface that it exposes. The flexible interface is implemented with the help of C++ templates.

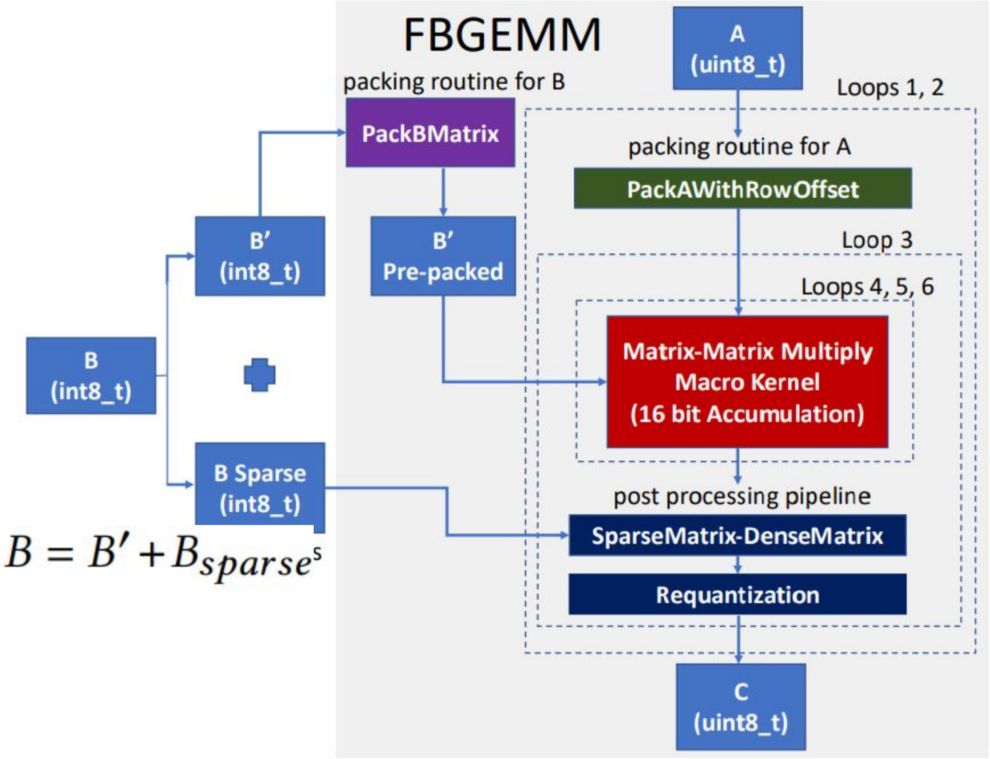
```
template<
  typename packingAMatrix,
  typename packingBMatrix,
  typename cT,
  typename processOutputType>
void fbgemmPacked(
    PackMatrix<packingAMatrix,
      typename packingAMatrix::inpType,
      typename packingAMatrix::accType>& packA,
    PackMatrix<packingBMatrix,
      typename packingBMatrix::inpType,
      typename packingBMatrix::accType>& packB,
    cT* C,
    void* C_buffer,
    std::int32_t ldc,
    const processOutputType& outProcess,
    int thread_id,
    int num_threads);
```

```
template <
    typename packingAMatrix,
    typename packingBMatrix,
    typename cT,
    typename processOutputType>
void fbgemmPacked(
    PackMatrix<
        packingAMatrix,
        typename packingAMatrix::inpType,
        typename packingAMatrix::accType>& packA,
    PackMatrix<
        packingBMatrix,
        typename packingBMatrix::inpType,
        typename packingBMatrix::accType>& packB,
    cT* C,
    int32_t* C_buffer,
    uint32_t ldc,
    const processOutputType& outProcess,
    int thread_id,
    int num_threads,
    const BlockingFactors* blocking_params) {
```

Paper: https://arxiv.org/pdf/2101.05615.pdf

The whole picture

We want $C = AB$, C, A, and B are $M \times N$, $M \times K$, and $K \times N$ matrices, respectively.




```
// Allocate a buffer for fbgemmPacked to use
auto buffer = at::empty(out_sizes, output.options().dtype(at::kInt));

int num_tasks = at::get_num_threads();
at::parallel_for(0, num_tasks, 1, [&](int64_t begin, int64_t end) {
  for (const auto task_id : c10::irange(begin, end)) {
```

```
fbgemm::PackAWithRowOffset<uint8_t> packA(
    /*trans=*/fbgemm::matrix_op_t::NoTranspose,
    /*nRow=*/M,
    /*nCol=*/K,
    /*smat=*/input_ptr,
    /*ld=*/K,
    /*pmat=*/nullptr); // Currently, packA manages ownership of `pmat`.
                       // TODO: Consider a way to pre-allocate and reuse
                       // pmat buffer.
```

```
Loop1 for ic = 0 to M-1 in steps of MCB
Loop2   for kc = 0 to K-1 in steps of KCB
          //Pack MCBxKCB block of A
Loop3     for jc = 0 to N-1 in steps of NCB
//-------------------Inner Kernel-----------
          //Dynamically generated inner kernel
          //Loop4 and Loop5 are in assembly
```

I'm in void fbgemmPacked

```
template <typename packingAMatrix, typename cT, typename processOutputType>
ExecuteKernel<
    packingAMatrix,
    PackBMatrix<int8_t, typename packingAMatrix::accType>,
    cT,
    processOutputType>::
    ExecuteKernel(
        PackMatrix<packingAMatrix, uint8_t, typename packingAMatrix::accType>&
            packA,
        PackMatrix<
            PackBMatrix<int8_t, typename packingAMatrix::accType>,
            int8_t,
            typename packingAMatrix::accType>& packB,
        cT* matC,
        int32_t* C_buffer,
        int32_t ldc,
        const processOutputType& outputProcess,
        thread_type_t th_info,
        const BlockingFactors* params)
    : CodeGenBase<uint8_t, int8_t, int32_t, typename packingAMatrix::accType>(
          params),
      packedA_(packA),
      packedB_(packB),
      matC_(matC),
      C_buffer_(C_buffer),
      ldc_(ldc),
      outputProcess_(outputProcess),
      th_info_(th_info) {
```

```
  for (int g = g_begin; g < g_end; ++g) {
    ExecuteKernel<packingAMatrix, packingBMatrix, cT, processOutputType>
        exeKernelObj(
            packA,
            packB,
            C,
            C_buffer,
            ldc,
            outProcess,
            th_info,
            blocking_params);
    for (int i = i_begin; i < i_end; i += MCB) { // i is the element index
      mc = std::min(i_end - i, MCB);
      for (int kb = 0; kb < kBlocks; ++kb) { // kb is the block index
        kc = (kb != kBlocks - 1 || _kc == 0) ? KCB : _kc;
        // pack A matrix
        blockA = {i, mc, g * KDimPerGroup + kb * KCB, kc};
        packA.pack(blockA);

#ifdef FBGEMM_MEASURE_TIME_BREAKDOWN
        t_end = std::chrono::high_resolution_clock::now();
        dt = std::chrono::duration_cast<std::chrono::nanoseconds>(
                 t_end - t_start)
                 .count();
        packing_time += (dt);
        t_start = std::chrono::high_resolution_clock::now();
#endif

        exeKernelObj.execute(g * kBlocks + kb);

#ifdef FBGEMM_MEASURE_TIME_BREAKDOWN
        t_end = std::chrono::high_resolution_clock::now();
        dt = std::chrono::duration_cast<std::chrono::nanoseconds>(
                 t_end - t_start)
                 .count();
        computing_time += (dt);
        t_start = std::chrono::high_resolution_clock::now();
#endif
      }
    }
  } // for each group
```

Dynamically generated inner kernel

This function generate **fn** functor via `.getOrCreate<...>` function

After some parallelization utils **fn** call looks like this:

```
fn(aBuf,
   bBuf,
   bBuf_pf,
   C_buffer_start,
   packedA_.numPackedCols(),
   leadingDim);
```

```
template <typename packingAMatrix, typename cT, typename processOutputType>
void ExecuteKernel<
    packingAMatrix,
    PackBMatrix<int8_t, typename packingAMatrix::accType>,
    cT,
    processOutputType>::execute(int kBlock) {
  // packedA_.printPackedMatrix("packedA from kernel");
  // packedB_.printPackedMatrix("packedB from kernel");

  int32_t bColBlocks = packedB_.blockCols();

  int8_t* bBuf;
  int8_t* bBuf_pf;

  uint8_t* aBuf = packedA_.getBuf(0);

  int32_t packed_rows_A = packedA_.numPackedRows();
  int32_t row_start_A = packedA_.packedRowStart();

  int group = kBlock / packedB_.blockRows();
  int NDim = packedB_.numCols();
  bool lastKBlock = packedB_.isThisLastKBlock(kBlock % packedB_.blockRows());
  bool accum = (kBlock % packedB_.blockRows()) > 0;

  int64_t jb_begin, jb_end;
  fbgemmPartition1D(
      th_info_.n_thread_id,
      th_info_.n_num_threads,
      bColBlocks,
      jb_begin,
      jb_end);
  if (jb_end == jb_begin) {
    return;
  }

  typename BaseType::jit_micro_kernel_fp fn;

  const inst_set_t isa = fbgemmInstructionSet();
  switch (isa) {
```

And no debug symbols again :(

> From **info locals** in **GDB** we can get **fn** address, e.g. `fn = 0x7ffff7e2e000`

> Because **fn** is presumably an assembler code generation, let's disassemble it

>

```
(gdb) disassemble 0x7ffff7e2e000, +0x300
Dump of assembler code from 0x7ffff7e2e000 to 0x7ffff7e2e300:
   0x00007ffff7e2e000:  push   %r12
   0x00007ffff7e2e002:  push   %r13
   0x00007ffff7e2e004:  push   %r14
   0x00007ffff7e2e006:  push   %r15
   0x00007ffff7e2e008:  vpcmpeqw %ymm13,%ymm13,%ymm13
   0x00007ffff7e2e00d:  vpsrlw $0xf,%ymm13,%ymm13
   0x00007ffff7e2e013:  imul   $0x4,%r9,%r9
   0x00007ffff7e2e017:  mov    %rsi,%r10
   0x00007ffff7e2e01a:  mov    %rdx,%r12
   0x00007ffff7e2e01d:  xor    %r13d,%r13d
   0x00007ffff7e2e020:  inc    %r13
   0x00007ffff7e2e023:  xor    %r14d,%r14d
   0x00007ffff7e2e026:  inc    %r14
   0x00007ffff7e2e029:  vpxor  %xmm0,%xmm0,%xmm0
   0x00007ffff7e2e02d:  vpxor  %xmm1,%xmm1,%xmm1
   0x00007ffff7e2e031:  vpxor  %xmm2,%xmm2,%xmm2
   0x00007ffff7e2e035:  vpxor  %xmm3,%xmm3,%xmm3
   0x00007ffff7e2e039:  vpxor  %xmm4,%xmm4,%xmm4
   0x00007ffff7e2e03d:  vpxor  %xmm5,%xmm5,%xmm5
   0x00007ffff7e2e041:  vpxor  %xmm6,%xmm6,%xmm6
   0x00007ffff7e2e045:  vpxor  %xmm7,%xmm7,%xmm7
   0x00007ffff7e2e049:  vpxor  %xmm8,%xmm8,%xmm8
   0x00007ffff7e2e04e:  vpxor  %xmm9,%xmm9,%xmm9
   0x00007ffff7e2e053:  vpxor  %xmm10,%xmm10,%xmm10
   0x00007ffff7e2e058:  vpxor  %xmm11,%xmm11,%xmm11
   0x00007ffff7e2e05d:  xor    %r15d,%r15d
   0x00007ffff7e2e060:  add    $0x4,%r15
```

```
0x00007ffff7e2e000:  push   %r12
0x00007ffff7e2e002:  push   %r13
0x00007ffff7e2e004:  push   %r14
0x00007ffff7e2e006:  push   %r15
0x00007ffff7e2e008:  vpcmpeqw %ymm13,%ymm13,%ymm13
0x00007ffff7e2e00d:  vpsrlw $0xf,%ymm13,%ymm13
0x00007ffff7e2e013:  imul   $0x4,%r9,%r9
0x00007ffff7e2e017:  mov    %rsi,%r10
0x00007ffff7e2e01a:  mov    %rdx,%r12
0x00007ffff7e2e01d:  xor    %r13d,%r13d
0x00007ffff7e2e020:  inc    %r13
0x00007ffff7e2e023:  xor    %r14d,%r14d
0x00007ffff7e2e026:  inc    %r14
0x00007ffff7e2e029:  vpxor  %xmm0,%xmm0,%xmm0
0x00007ffff7e2e02d:  vpxor  %xmm1,%xmm1,%xmm1
0x00007ffff7e2e031:  vpxor  %xmm2,%xmm2,%xmm2
0x00007ffff7e2e035:  vpxor  %xmm3,%xmm3,%xmm3
0x00007ffff7e2e039:  vpxor  %xmm4,%xmm4,%xmm4
0x00007ffff7e2e03d:  vpxor  %xmm5,%xmm5,%xmm5
0x00007ffff7e2e041:  vpxor  %xmm6,%xmm6,%xmm6
0x00007ffff7e2e045:  vpxor  %xmm7,%xmm7,%xmm7
0x00007ffff7e2e049:  vpxor  %xmm8,%xmm8,%xmm8
0x00007ffff7e2e04e:  vpxor  %xmm9,%xmm9,%xmm9
0x00007ffff7e2e053:  vpxor  %xmm10,%xmm10,%xmm10
0x00007ffff7e2e058:  vpxor  %xmm11,%xmm11,%xmm11
0x00007ffff7e2e05d:  xor    %r15d,%r15d
0x00007ffff7e2e060:  add    $0x4,%r15
0x00007ffff7e2e064:  vmovdqa (%rsi),%ymm14
0x00007ffff7e2e068:  vpbroadcastd (%rdi),%ymm15
0x00007ffff7e2e06d:  vpmaddubsw %ymm14,%ymm15,%ymm12
0x00007ffff7e2e072:  vpmaddwd %ymm12,%ymm13,%ymm12
0x00007ffff7e2e077:  vpaddd %ymm0,%ymm12,%ymm0
0x00007ffff7e2e07b:  vpbroadcastd 0x200(%rdi),%ymm15
0x00007ffff7e2e084:  vpmaddubsw %ymm14,%ymm15,%ymm12
0x00007ffff7e2e089:  vpmaddwd %ymm12,%ymm13,%ymm12
0x00007ffff7e2e08e:  vpaddd %ymm1,%ymm12,%ymm1
0x00007ffff7e2e092:  vpbroadcastd 0x400(%rdi),%ymm15
0x00007ffff7e2e09b:  vpmaddubsw %ymm14,%ymm15,%ymm12
0x00007ffff7e2e0a0:  vpmaddwd %ymm12,%ymm13,%ymm12
0x00007ffff7e2e0a5:  vpaddd %ymm2,%ymm12,%ymm2
0x00007ffff7e2e0a9:  vpbroadcastd 0x600(%rdi),%ymm15
0x00007ffff7e2e0b2:  vpmaddubsw %ymm14,%ymm15,%ymm12
0x00007ffff7e2e0b7:  vpmaddwd %ymm12,%ymm13,%ymm12
0x00007ffff7e2e0bc:  vpaddd %ymm3,%ymm12,%ymm3
0x00007ffff7e2e0c0:  vpbroadcastd 0x800(%rdi),%ymm15
0x00007ffff7e2e0c9:  vpmaddubsw %ymm14,%ymm15,%ymm12
0x00007ffff7e2e0ce:  vpmaddwd %ymm12,%ymm13,%ymm12
0x00007ffff7e2e0d3:  vpaddd %ymm4,%ymm12,%ymm4
0x00007ffff7e2e0d7:  vpbroadcastd 0xa00(%rdi),%ymm15
0x00007ffff7e2e0e0:  vpmaddubsw %ymm14,%ymm15,%ymm12
0x00007ffff7e2e0e0:  vpmaddubsw %ymm14,%ymm15,%ymm12
0x00007ffff7e2e0e5:  vpmaddwd %ymm12,%ymm13,%ymm12
0x00007ffff7e2e0ea:  vpaddd %ymm5,%ymm12,%ymm5
0x00007ffff7e2e0ee:  vpbroadcastd 0xc00(%rdi),%ymm15
0x00007ffff7e2e0f7:  vpmaddubsw %ymm14,%ymm15,%ymm12
0x00007ffff7e2e0fc:  vpmaddwd %ymm12,%ymm13,%ymm12
0x00007ffff7e2e101:  vpaddd %ymm6,%ymm12,%ymm6
--Type <RET> for more, q to quit, c to continue without pagi
0x00007ffff7e2e105:  vpbroadcastd 0xe00(%rdi),%ymm15
0x00007ffff7e2e10e:  vpmaddubsw %ymm14,%ymm15,%ymm12
0x00007ffff7e2e113:  vpmaddwd %ymm12,%ymm13,%ymm12
0x00007ffff7e2e118:  vpaddd %ymm7,%ymm12,%ymm7
0x00007ffff7e2e11c:  vpbroadcastd 0x1000(%rdi),%ymm15
0x00007ffff7e2e125:  vpmaddubsw %ymm14,%ymm15,%ymm12
0x00007ffff7e2e12a:  vpmaddwd %ymm12,%ymm13,%ymm12
0x00007ffff7e2e12f:  vpaddd %ymm8,%ymm12,%ymm8
0x00007ffff7e2e134:  vpbroadcastd 0x1200(%rdi),%ymm15
0x00007ffff7e2e13d:  vpmaddubsw %ymm14,%ymm15,%ymm12
0x00007ffff7e2e142:  vpmaddwd %ymm12,%ymm13,%ymm12
0x00007ffff7e2e147:  vpaddd %ymm9,%ymm12,%ymm9
0x00007ffff7e2e14c:  vpbroadcastd 0x1400(%rdi),%ymm15
0x00007ffff7e2e155:  vpmaddubsw %ymm14,%ymm15,%ymm12
0x00007ffff7e2e15a:  vpmaddwd %ymm12,%ymm13,%ymm12
0x00007ffff7e2e15f:  vpaddd %ymm10,%ymm12,%ymm10
0x00007ffff7e2e164:  vpbroadcastd 0x1600(%rdi),%ymm15
0x00007ffff7e2e16d:  vpmaddubsw %ymm14,%ymm15,%ymm12
0x00007ffff7e2e172:  vpmaddwd %ymm12,%ymm13,%ymm12
0x00007ffff7e2e177:  vpaddd %ymm11,%ymm12,%ymm11
0x00007ffff7e2e17c:  prefetcht0 (%rdx)
0x00007ffff7e2e17f:  add    $0x4,%rdi
0x00007ffff7e2e183:  add    $0x20,%rsi
0x00007ffff7e2e187:  add    $0x20,%rdx
0x00007ffff7e2e18b:  cmp    %r8,%r15
0x00007ffff7e2e18e:  jl     0x7ffff7e2e060
0x00007ffff7e2e194:  xor    %r11d,%r11d
0x00007ffff7e2e197:  vmovups %ymm0,(%rcx,%r11,1)
0x00007ffff7e2e19d:  add    %r9,%r11
0x00007ffff7e2e1a0:  vmovups %ymm1,(%rcx,%r11,1)
0x00007ffff7e2e1a6:  add    %r9,%r11
0x00007ffff7e2e1a9:  vmovups %ymm2,(%rcx,%r11,1)
0x00007ffff7e2e1af:  add    %r9,%r11
0x00007ffff7e2e1b2:  vmovups %ymm3,(%rcx,%r11,1)
0x00007ffff7e2e1b8:  add    %r9,%r11
0x00007ffff7e2e1bb:  vmovups %ymm4,(%rcx,%r11,1)
0x00007ffff7e2e1c1:  add    %r9,%r11
0x00007ffff7e2e1c4:  vmovups %ymm5,(%rcx,%r11,1)
0x00007ffff7e2e1ca:  add    %r9,%r11
0x00007ffff7e2e1cd:  vmovups %ymm6,(%rcx,%r11,1)
0x00007ffff7e2e1d3:  add    %r9,%r11
0x00007ffff7e2e1d6:  vmovups %ymm7,(%rcx,%r11,1)
0x00007ffff7e2e1dc:  add    %r9,%r11
0x00007ffff7e2e1df:  vmovups %ymm8,(%rcx,%r11,1)
0x00007ffff7e2e1e5:  add    %r9,%r11
0x00007ffff7e2e1e8:  vmovups %ymm9,(%rcx,%r11,1)
0x00007ffff7e2e1ee:  add    %r9,%r11
0x00007ffff7e2e1ee:  add    %r9,%r11
0x00007ffff7e2e1f1:  vmovups %ymm10,(%rcx,%r11,1)
0x00007ffff7e2e1f7:  add    %r9,%r11
0x00007ffff7e2e1fa:  vmovups %ymm11,(%rcx,%r11,1)
0x00007ffff7e2e200:  sub    %r8,%rdi
0x00007ffff7e2e203:  mov    %r10,%rsi
0x00007ffff7e2e206:  imul   $0x20,%r14,%r11
0x00007ffff7e2e20a:  add    %r11,%rsi
0x00007ffff7e2e20d:  mov    %r12,%rdx
0x00007ffff7e2e210:  add    %r11,%rdx
0x00007ffff7e2e213:  add    $0x20,%rcx
Type <RET> for more, q to quit, c to continue without paging--
0x00007ffff7e2e217:  cmp    $0x1,%r14
0x00007ffff7e2e21b:  jl     0x7ffff7e2e026
0x00007ffff7e2e221:  add    $0x1800,%rdi
0x00007ffff7e2e228:  sub    $0x20,%rcx
0x00007ffff7e2e22c:  imul   $0xc,%r9,%r11
0x00007ffff7e2e230:  add    %r11,%rcx
0x00007ffff7e2e233:  mov    %r10,%rsi
0x00007ffff7e2e236:  mov    %r12,%rdx
0x00007ffff7e2e239:  cmp    $0xa,%r13
0x00007ffff7e2e23d:  jl     0x7ffff7e2e020
0x00007ffff7e2e243:  pop    %r15
0x00007ffff7e2e245:  pop    %r14
0x00007ffff7e2e247:  pop    %r13
0x00007ffff7e2e249:  pop    %r12
0x00007ffff7e2e24b:  ret
0x00007ffff7e2e24c:  add    %al,(%rax)
0x00007ffff7e2e24e:  add    %al,(%rax)
0x00007ffff7e2e250:  add    %al,(%rax)
0x00007ffff7e2e252:  add    %al,(%rax)
0x00007ffff7e2e254:  add    %al,(%rax)
0x00007ffff7e2e256:  add    %al,(%rax)
0x00007ffff7e2e258:  add    %al,(%rax)
0x00007ffff7e2e25a:  add    %al,(%rax)
0x00007ffff7e2e25c:  add    %al,(%rax)
0x00007ffff7e2e25e:  add    %al,(%rax)
0x00007ffff7e2e260:  add    %al,(%rax)
0x00007ffff7e2e262:  add    %al,(%rax)
0x00007ffff7e2e264:  add    %al,(%rax)
0x00007ffff7e2e266:  add    %al,(%rax)
0x00007ffff7e2e268:  add    %al,(%rax)
0x00007ffff7e2e26a:  add    %al,(%rax)
0x00007ffff7e2e26c:  add    %al,(%rax)
0x00007ffff7e2e26e:  add    %al,(%rax)
0x00007ffff7e2e270:  add    %al,(%rax)
0x00007ffff7e2e272:  add    %al,(%rax)
0x00007ffff7e2e274:  add    %al,(%rax)
0x00007ffff7e2e276:  add    %al,(%rax)
0x00007ffff7e2e278:  add    %al,(%rax)
0x00007ffff7e2e27a:  add    %al,(%rax)
0x00007ffff7e2e27c:  add    %al,(%rax)
0x00007ffff7e2e27e:  add    %al,(%rax)
0x00007ffff7e2e280:  add    %al,(%rax)
```

Perfomance comparison

- **BLAS GEMM** doesn't have any explicit **"parallel_for"** loop
- **FBGEMM** on the other hand does
- We can change number of available threads to 'n' via
  - `at::set_num_interop_threads(n);` and `at::set_num_threads(n);`

Benchmark was done in Release build of PyTorch

```
8 threads
torch::linear 256x512 30.63
jit::x86_linear 256x512 1.86
jit::fbgemm_linear 256x512 1.16
torch::linear 1024x2048 2127.67
jit::x86_linear 1024x2048 9.72
jit::fbgemm_linear 1024x2048 12.27
torch::linear 2048x4096 19331.9
jit::x86_linear 2048x4096 120.86
jit::fbgemm_linear 2048x4096 120.62
```

```
1 thread
torch::linear 256x512 31.5
jit::x86_linear 256x512 1.29
jit::fbgemm_linear 256x512 0.69
torch::linear 1024x2048 2224.84
jit::x86_linear 1024x2048 34.1
jit::fbgemm_linear 1024x2048 33.95
torch::linear 2048x4096 18986.2
jit::x86_linear 2048x4096 245.65
jit::fbgemm_linear 2048x4096 249.47
```

It seems that in some cases (for "small" matrices) parallelism is an overhead

Summary

- We have a method and tools for low-level parsing of PyTorch operations
- We basically have **GEMMs** on a plate
- We found some suboptimalities
- The one can use project repo to run experiments
- It seems that our method scales well to other DL frameworks as well

Future plans

- The project is supposed to be turned into a study for a master's degree thesis or for publications
- Match assembler with **FBGEMM** paper text
- Consider other quantization methods
- Try to optimize **BLAS GEMM** (for example we can use this: https://github.com/flame/how-to-optimize-gemm) or rewrite it
- Create a benchmark of **GEMMs**
- If we'll be able to get more efficient implementation of **Linear** layer, we can create our own inference oriented library
- Try to investigate other layers, like **Conv3d** (https://oneapi-src.github.io/oneDNN/v0/index.html)