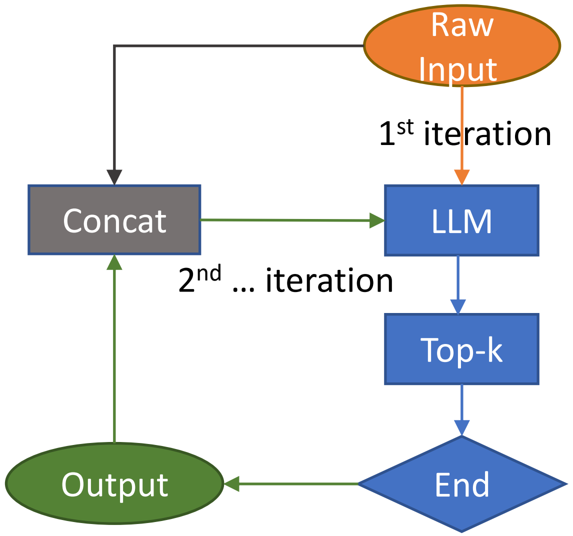
Low-precision Optimization for LLM Inference

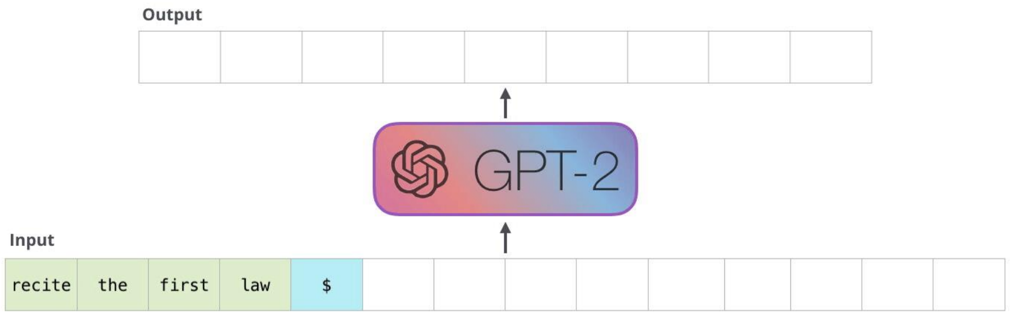
Jiong Gong
Principal Engineer
DCAI/AISE, Intel
Oct. 2023

LLM workload introduction & analysis

-- Typical LLM workload: Text Generation

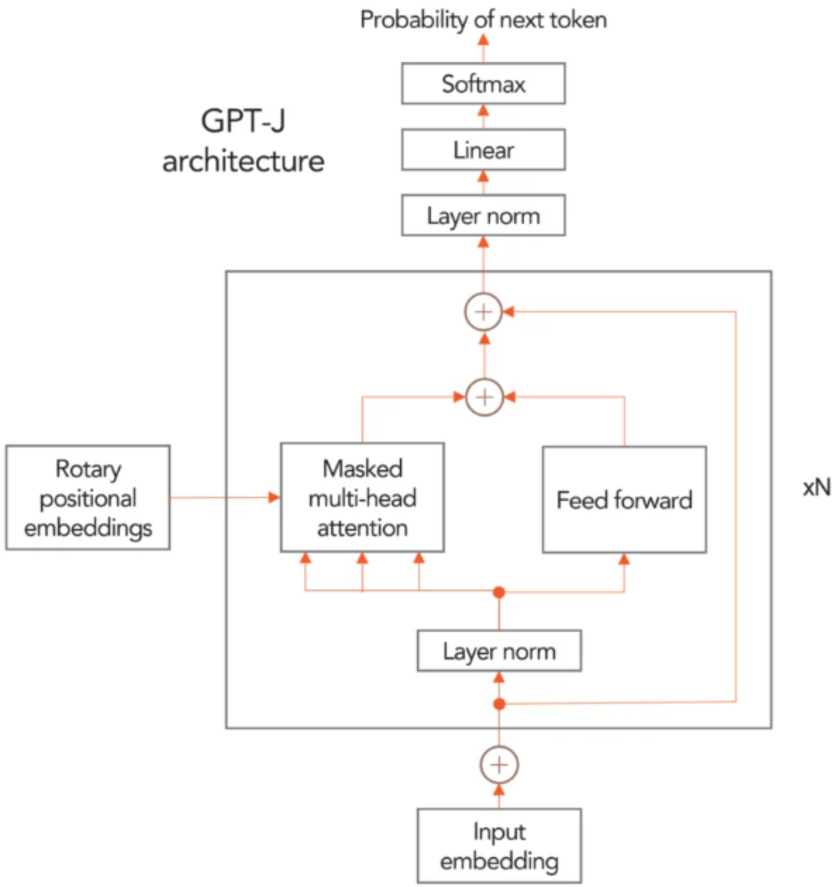
- Text Generation
  - Given a prompt sentence or tokens, the model predicts the following tokens, one by one
  - Typical characteristic
    - Input of 1st iteration: *original raw input*
    - Input of 2nd and following iteration: *raw input + all generated output tokens*
    - Dynamic shape as per model's view





LLM Models overview

- "GPT-like" autoregressive language models
  - Decoder layer
    - Masked multi-heads attention
    - MLP Block(FeedForward)



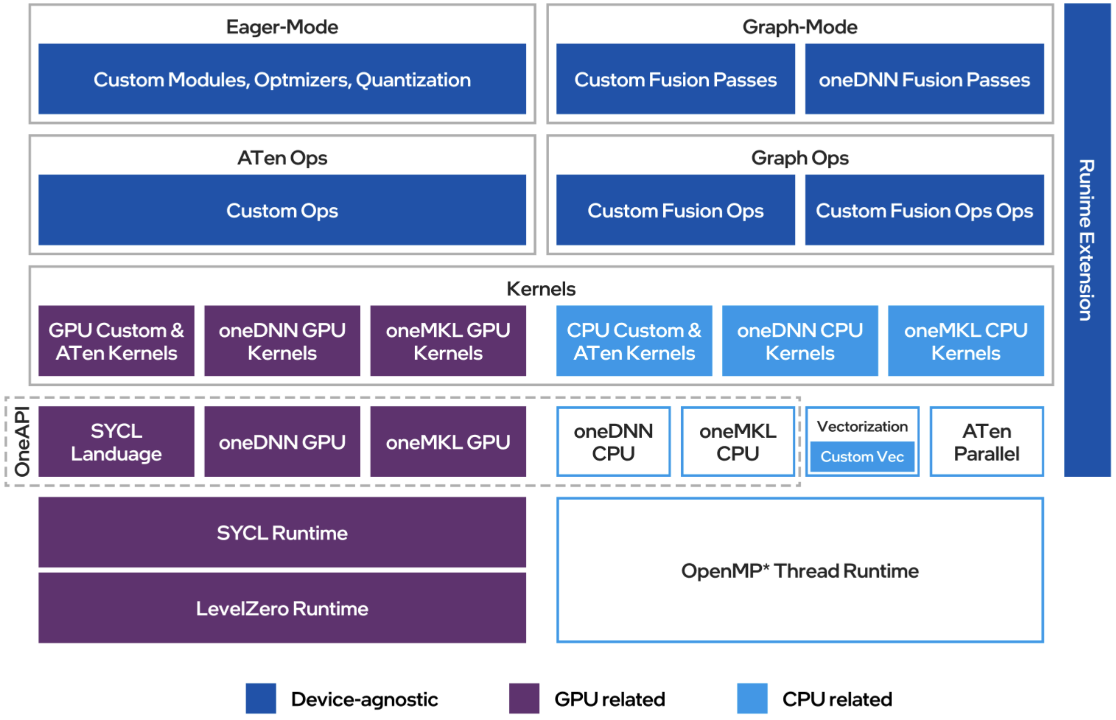
Models	Model size	Layers	Number Head	Key size	hidden size	Vocabulary size
GPT	0.117B	12	12	64	768	40000
GP-2	1.5B	48	25	64	1600	50257
GP-3	175B	96	96	128	12288	
Megatron-LM	8.3B	72	24	128	3072	51200
GPT-J	6B	28	16	256	4096	50400
Bloom	176B	70	112	128	14336	
LLaMA	7B	32	32	128	4096	
Turing-NLG	17.2B	78	28	152	4256	51200
GPT-Neox	20B	44	64	96	6144	50432
Gopher	280B	80	128	128	16384	32000
MT-NLG	530B	105	128	160	20480	

Performance Characteristics and Optimization Strategy of LLM Inference

- Different performance characteristics for first token and second+ tokens
  - First token: mostly computation intensive, and with larger activation buffers
  - Second+ token: memory-intensive, and with smaller activation buffers
- GEMM accounts for >50% of the workload ⇐ *Focus of this presentation*
  - First token: take advantage of FLOPS capacity from HW dot-product accelerators
  - Second token: increase bandwidth capacity (high-bandwidth memory, scale up/out) or reduce bandwidth demand (weight compression)
- Memory-intensive KV cache arrangement (second+ tokens)
  - Concatenation of new cache entries, reordering of cache beams
  - Require dedicated algorithm to save memory accesses
- Employ fusions to reduce kernel launch/schedule overhead and improve data locality for larger activation buffers

Background – Intel Extension for PyTorch (IPEX)

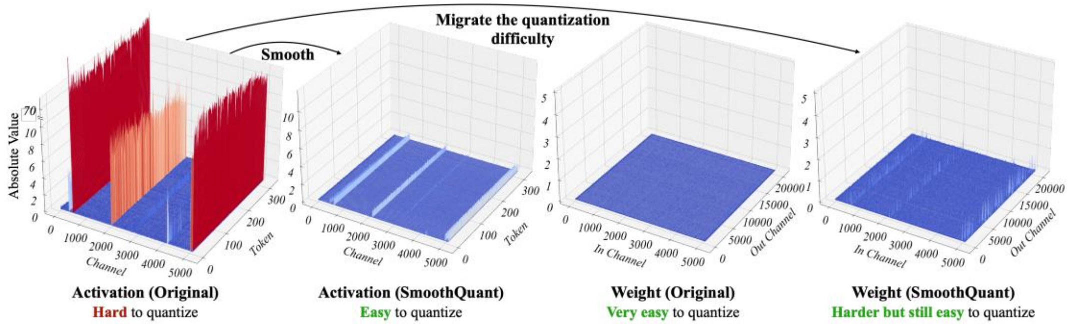
- Staging area for out-of-tree Intel optimizations
- Support both Intel CPU and GPU
- Extra performance boost gained with ease-of-use Python API
- Support both Python and C++ based deployment



Key LLM Optimization Technologies in IPEX

- General GEMM kernel optimizations
  - Systolic array support from HW: AMX on SPR, DPAS on PVC
  - Improve data locality: Register/cache blocking, weight prepacking
- Low precision compute and weight compression
  - BF16 auto-mixed precision
  - INT8 static and smooth quantization
  - INT4/INT8 weight-only quantization (WOQ)
- Indirect access KV cache (a simplified PagedAttention algorithm)
- Fusions
  - Flash attention, GEMM post-op fusion, rotary embedding fusion etc.
- Scale up and scale out via tensor parallel
- Other misc optimizations (e.g., reduced compute for first token with single batch, improved core occupancy for SDPA with parallel reduction)

INT8 SmoothQuant

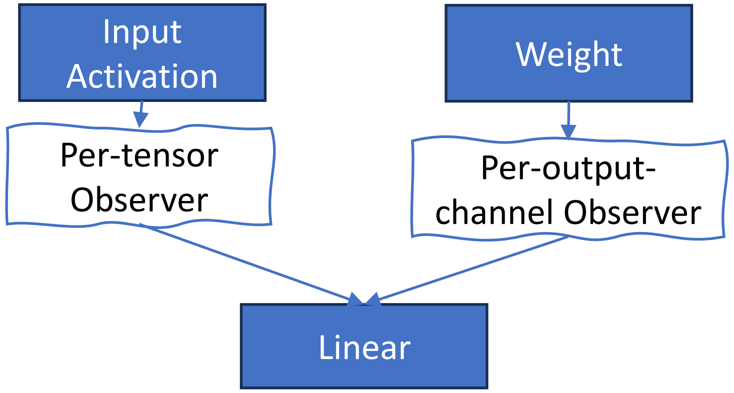


- Background
  - To handle activation outliers in LLM, for linear only. (Paper)
  - Scaling factors s: $\mathbf{Y} = (\mathbf{X}\mathrm{diag}(\mathbf{s})^{-1}) \cdot (\mathrm{diag}(\mathbf{s})\mathbf{W}) = \hat{\mathbf{X}}\hat{\mathbf{W}}$

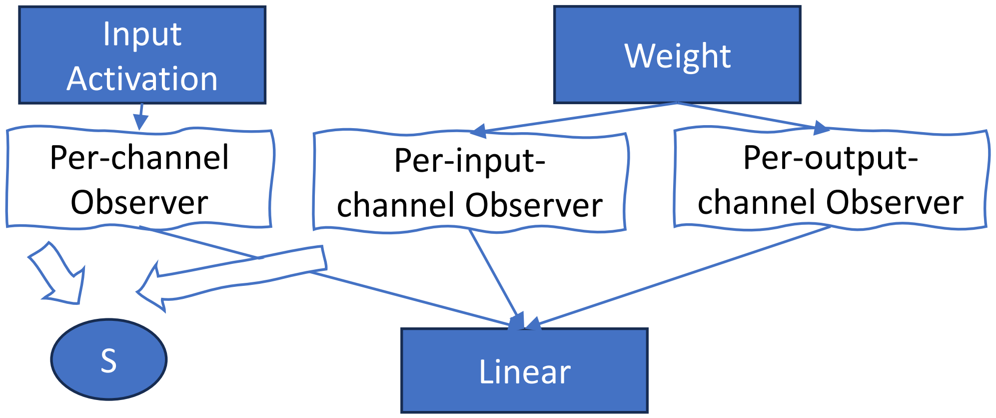
$$\mathbf{s}_j = \max(|\mathbf{X}_j|)^{\alpha} / \max(|\mathbf{W}_j|)^{1-\alpha} \quad j = 1, 2, ..., C_i$$

j: input channel

  - Alpha (α): Hyperparameter, =0.5 by default.
- Quantization flow



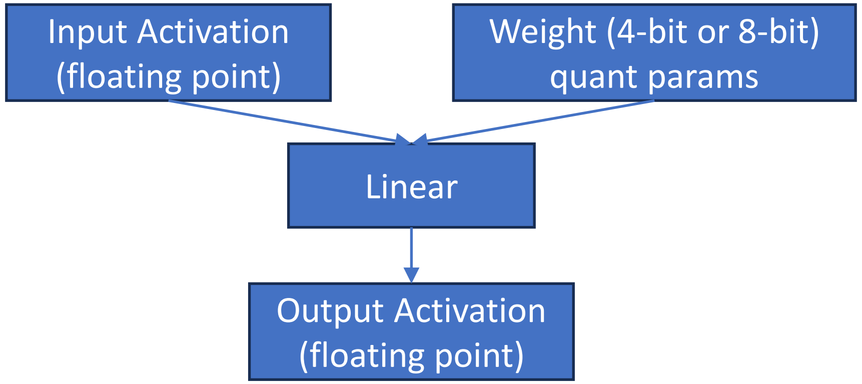
Normal static int8 quantization



Smooth static int8 quantization

Weight-only-Quantization(WOQ)

- Motivation
  - GEMMs in LLM second+ token inference are memory bound
  - Naïve dynamic quantization is harder to meet accuracy goal
- Solution
  - Quantize weights in lower-bit (e.g. int4, int8) while leaving activations in floating point (e.g. fp32, fp16, bf16), computation happens in low precision.
- Benefits
  - Ease-of-use: a single line of model wrapping, no need to insert observers like static quantization
  - Ease of composing with other features, e.g., scale-up/scale-out with DeepSpeed
  - 8-bit: close to next token perf of static quant, better accuracy than bf16
  - 4-bit: various algorithms (e.g., GPTQ) exist to get comparable accuracy to fp32



Popular WOQ Recipes

- Quantization schemes
  - Affine quantization (INT4, INT8 etc.) – symmetric or asymmetric
  - Lookup-table-based 4-bit floating point quantization (FP4, NF4 etc.)
- Quantization granularity for weights
  - Per-output-channel quantization
  - Per-input-channel-group quantization
- Combined with dynamic quantization for activations
  - Per-tensor, per-batch, per-IC quantization to INT8

Int4 = quantize(fp, scale, zero_point) = round(fp / scale) + zero_point
fp = dequantize(int4, scale, zero_point) = (int4 – zero_point) * scale

FP4_or_NF4 = quantize(fp, scale) = Inverse_LUT(fp / scale)
fp = dequantize(FP4_or_NF4, scale) = LUT(FP4_or_NF4) * scale

Map 4-bit values to “normal float”:
“[-1.0, -0.6961928009986877, -0.5250730514526367, -0.39491748809814453, -0.28444138169288635, -0.18477343022823334, -0.09105003625154495, 0.0, 0.07958029955625534, 0.16093020141124725, 0.24611230194568634, 0.33791524171829224, 0.44070982933044434, 0.5626170039176941, 0.7229568362236023, 1.0]”

IPEX.optimize_transformers – one-liner LLM optimization API

- ## Motivation

- ipex.optimize_transformers API allows users to adopt all IPEX LLM related optimizations transparently and have good OOB usage with public transformers.

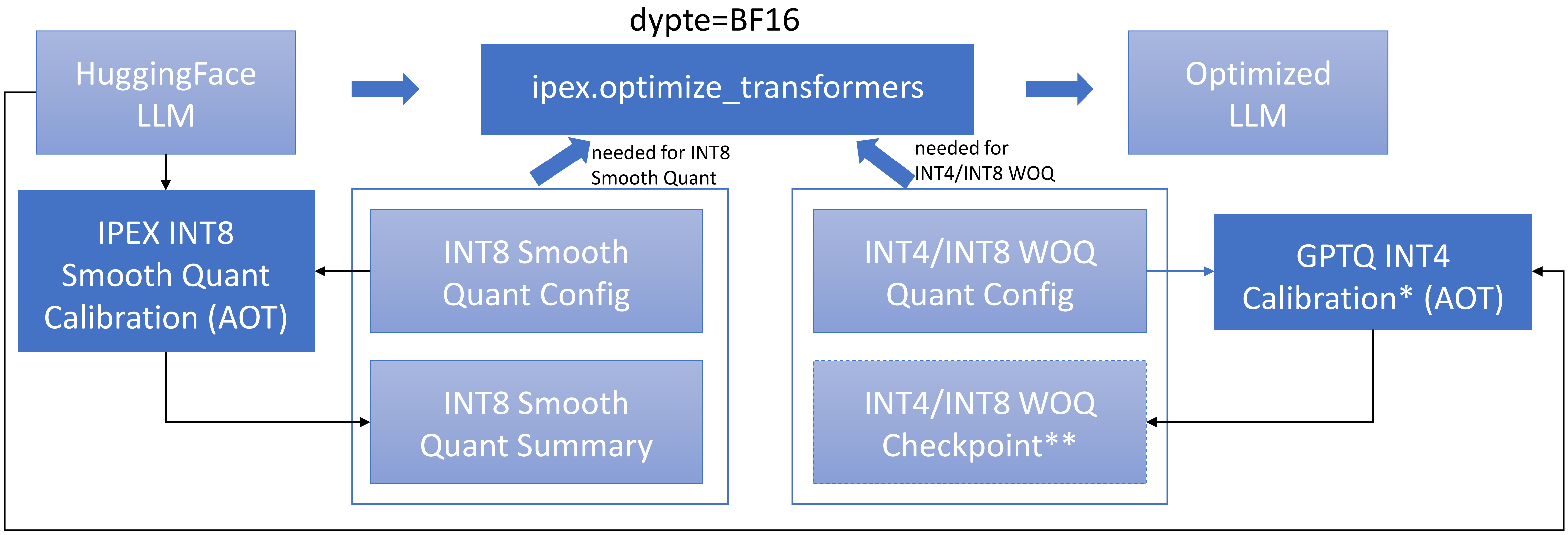
- ## Workflows

1. It converts supported models to a reference model built with common building blocks (reference IRs of module like MHA/MLP) which is device/dtype independent. Meanwhile it abstracts optimizations patterns insides, like ROPE, IAKV and Linear fusions.
2. It lowers the reference model to specific implementations of optimizations on CPU/XPU.

- ## Usage

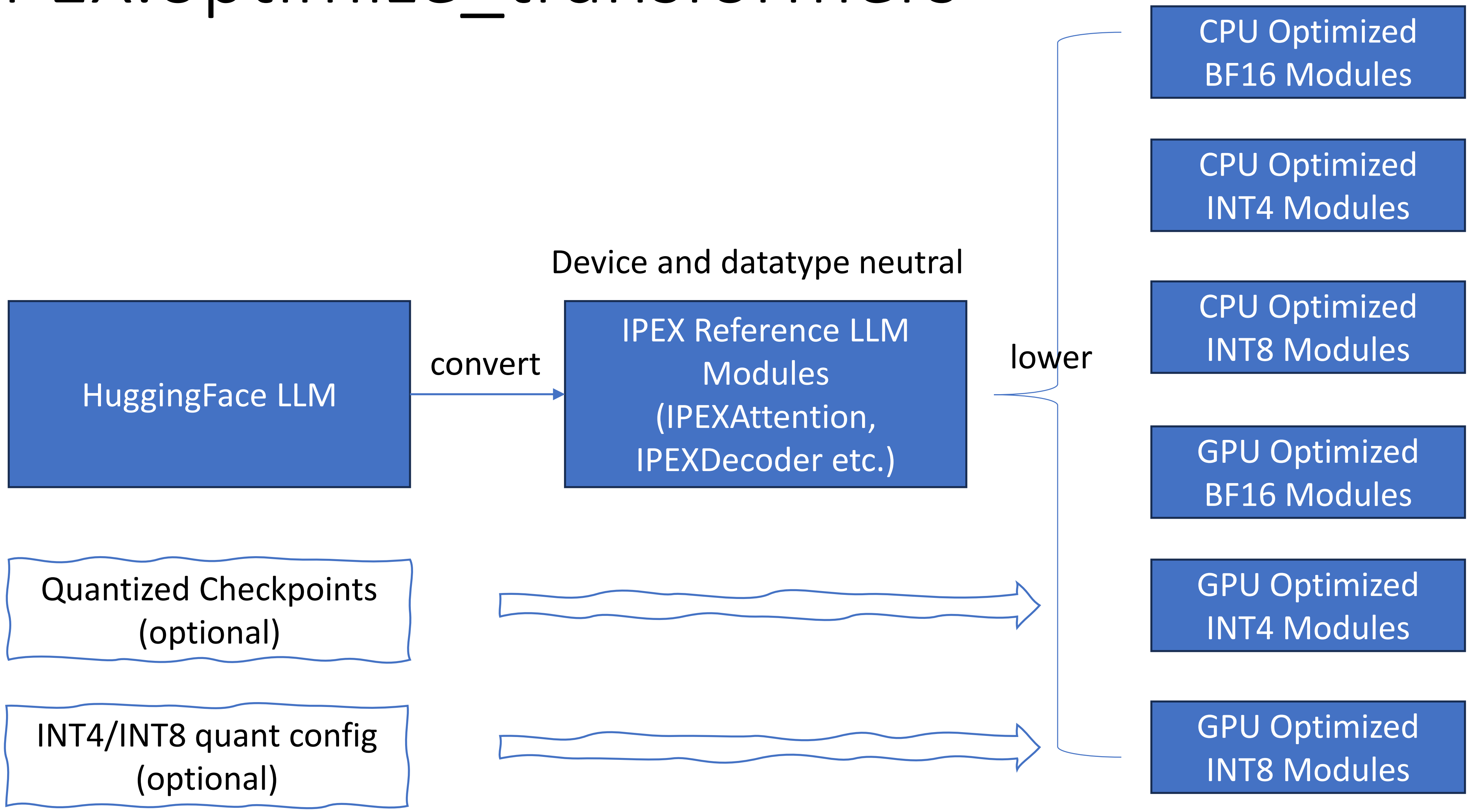
```
import torch
import transformers
import intel_extension_for_pytorch as ipex

model= transformers.AutoModelForCausalLM(model_name_or_path).eval()
model= ipex.optimize_transformers(model, dtype, device)
with torch.no_grad():
   model.generate(generation_args)
```

IPEX LLM Workflow (BF16/INT8/INT4)



* available as a separate script. API will be provided in the future release
** optional

IPEX.optimize_transformers

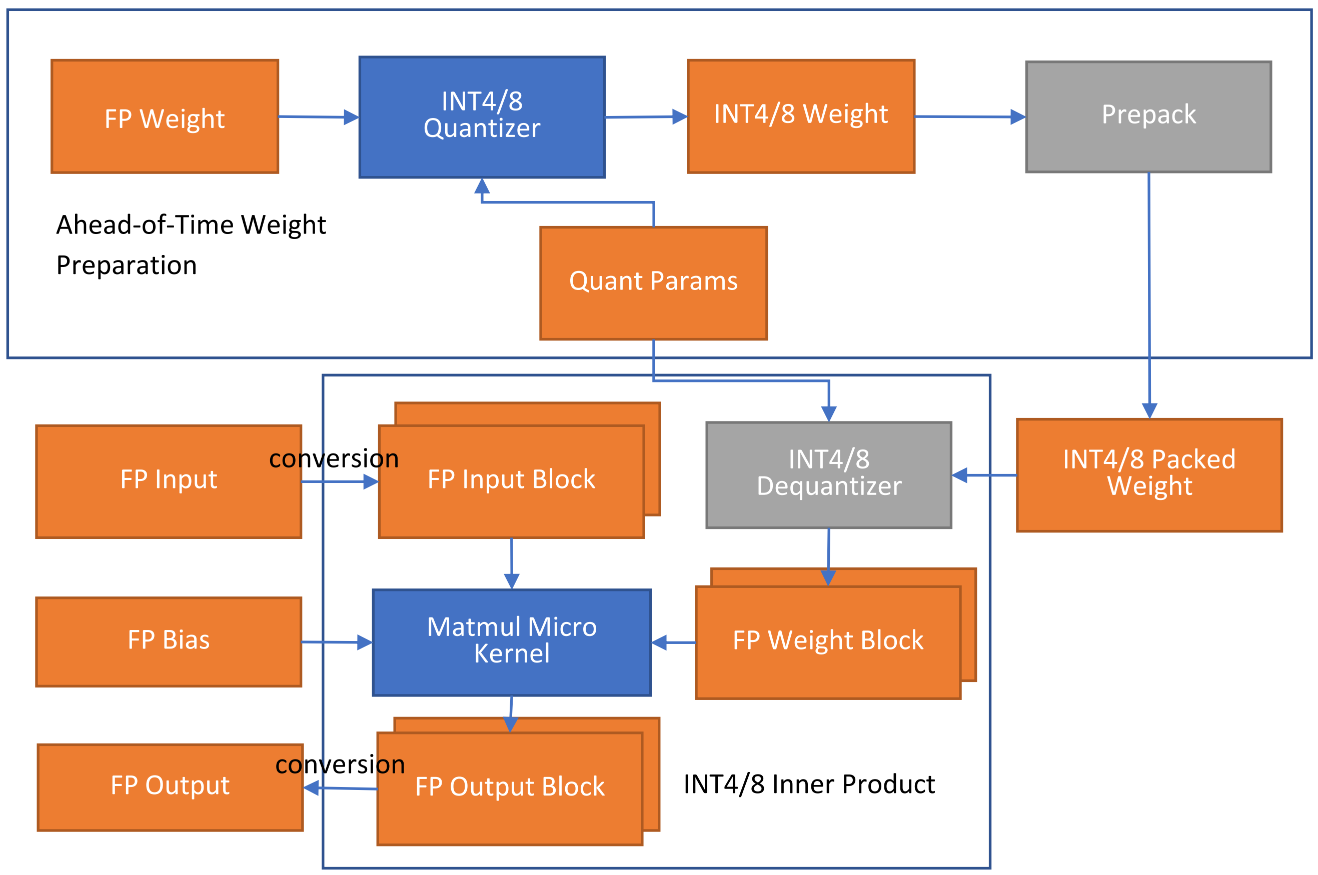


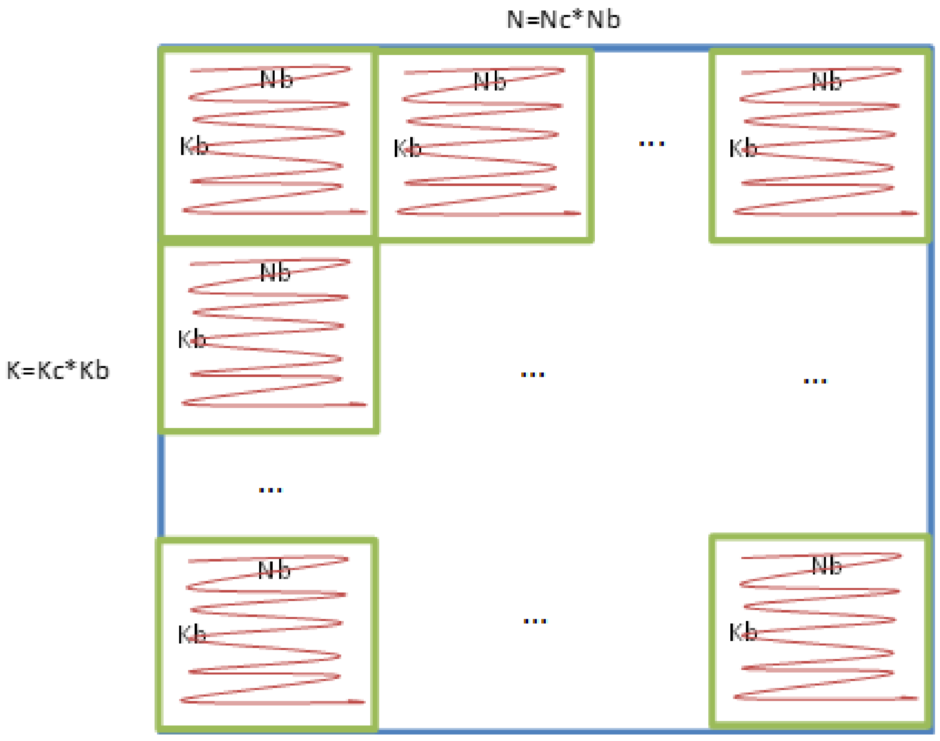
WOQ Quantization Recipes Supported by IPEX

Activation Type	Weight Quant Scheme	Compute Type	Activation Quant Scheme (INT8)
FP32/FP16/BF16	Per-output-channel Asymmetric INT8 Per-output-channel Asymmetric INT4**	FP32/FP16/BF16/INT8 (controlled by WoqLowpMode)	Per-tensor Per-batch Per-IC group

** Adding more weight quant scheme: NF4, FP4, per-IC group

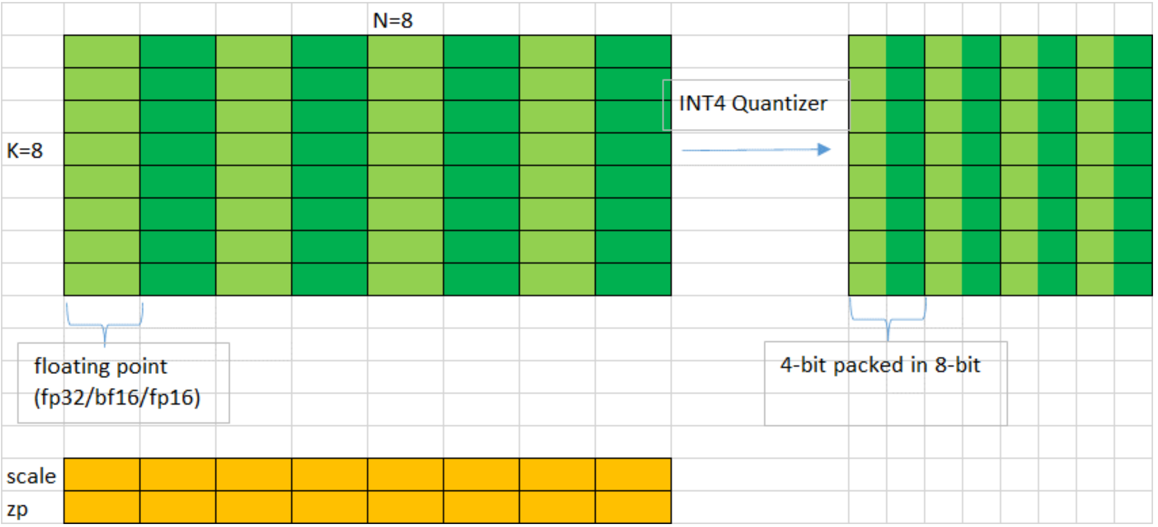
Overall flow of 4/8-bit low-precision weight-only quant inner-product operation from the LLM





Prepacked weight layout on Xeon

Vectorized INT4 Dequant to FP (Xeon)




```
__m128i tmp = _mm_loadu_si64(int4_ptr); // load 16 (8-byte) int4 values
// unpack 16 int4 values into 16 uint8 values
__m128i bytes = _mm_cvtepu8_epi16(tmp);
const __m128i lowMask = _mm_set1_epi8(0xF);
__m128i high = _mm_andnot_si128(lowMask, bytes);
__m128i low = _mm_and_si128(lowMask, bytes);
high = _mm_slli_epi16(high, 4);
bytes = _mm_or_si128(low, high);
// convert to floating points and do affine transformation
__m512 fp_values = _mm512_mul_ps(
        _mm512_sub_ps(
          _mm512_cvtepi32_ps(_mm512_cvtepu8_epi32(bytes)), zero_points
        ),
        scales
      );
```

Vectorized LUT-based INT4 Dequant to FP with Shuffled Layout

x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15 y0 y1 y2 y3 y4 y5 y6 y7 y8 y9 y10 y11 y12 y13 y14 y15

shuffle in per 32 int4 elements

x0 y0 x1 y1 x2 y2 x3 y3 x4 y4 x5 y5 x6 y6 x7 y7 x8 y8 x9 y9 x10 y10 x11 y11 x12 y12 x13 y13 x14 y14 x15 y15

```
 __m512 lut = _mm512_set_ps(15.0f, 14.0f, 13.0f, 12.0f, 11.0f, 10.0f,
9.0f, 8.0f, 7.0f, 6.0f, 5.0f, 4.0f, 3.0f, 2.0f, 1.0f, 0.0f);
  uint64_t packed = reinterpret_cast<uint64_t*>(int4_ptr)[0];
  uint64_t high = packed >> 4;
  __m128i int8 = _mm_set_epi64x(high, packed);
  __m512i int32 = _mm512_cvtepu8_epi32(int8);
  __m512 fp_values = _mm512_mul_ps(
            _mm512_sub_ps(_mm512_permutexvar_ps(int32, lut),
zero_points),
            scales
          );
```

4-bit packed in 8-bit

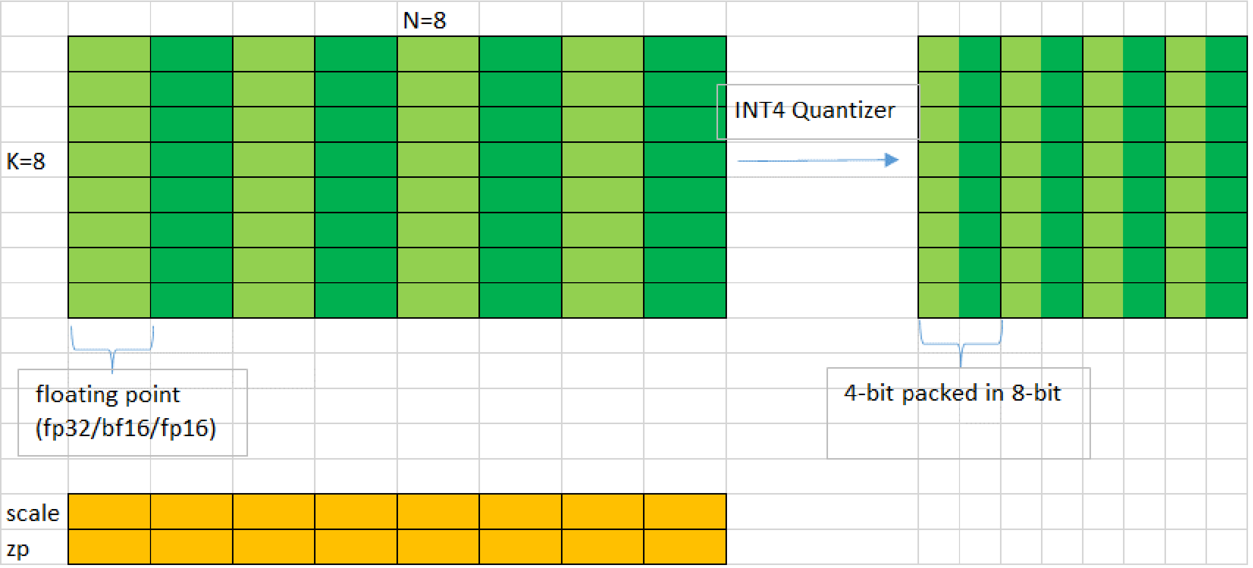
packed	x0	y0	x1	y1	x2	y2	x3	y3	x4	y4	x5	y5	x6	y6	x7	y7
high	y0	x1	y1	x2	y2	x3	y3	x4	y4	x5	y5	x6	y6	x7	y7	0
int8	x0	y0	x1	y1	x2	y2	x3	y3	x4	y4	x5	y5	x6	y6	x7	y7
	y0	x1	y1	x2	y2	x3	y3	x4	y4	x5	y5	x6	y6	x7	y7	0
int32	x0	y0	0	0	0	0	0	0	x1	y1	0	0	0	0	0	0
	x2	y2	0	0	0	0	0	0	x3	y3	0	0	0	0	0	0
	x4	y4	0	0	0	0	0	0	x5	y5	0	0	0	0	0	0
	x6	y6	0	0	0	0	0	0	x7	y7	0	0	0	0	0	0
	y0	x1	0	0	0	0	0	0	y1	x2	0	0	0	0	0	0
	y2	x3	0	0	0	0	0	0	y3	x4	0	0	0	0	0	0
	y4	x5	0	0	0	0	0	0	y5	x6	0	0	0	0	0	0
	y6	x7	0	0	0	0	0	0	y7	0	0	0	0	0	0	0
fp32	fx0								fx1							
	fx2								fx3							
	fx4								fx5							
	fx6								fx7							
	fy0								fy1							
	fy2								fy3							
	fy4								fy5							
	fy6								fy7							

INT4 Dequant to INT8 compute

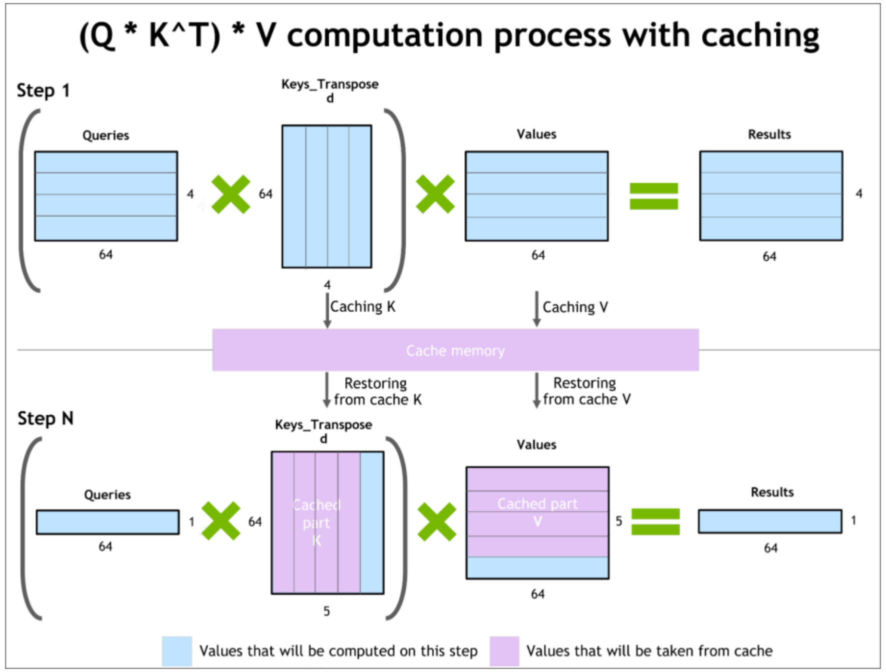
- Dynamic quantize A with qA = A / scaleA + zA
- qB’ = qB – zB, qB in uint4, zB in uint4, qB’ in int8
- C = (qA – zA) * (qB – zB) * scaleA * scaleB, scaleA and scaleB in fp
    - = (qA – zA) * qB’ * scaleA * scaleB
    - = (qA * qB’ – zA * qB’) * scaleA * scaleB
    - = (qC – zA * sum(qB’)) * scaleA * scaleB
- qC = qA * qB’ can be computed with INT8 dot-product instruction
- sum(qB’) can be pre-computed

INT4 Kernel on PVC – Optimized with XeTLA

1. Weight only quantization:
    a. Convert FP32/BF16/FP16 weight to INT4 weight.
    b. Pack 4bit weight into 8 bit.
2. Dequantize weight to FP16 data:
    a. Unpack 8-bit weight and zero_points to 4-bit data.
    b. Convert weight to FP16 data.
3. Compute with FP16.



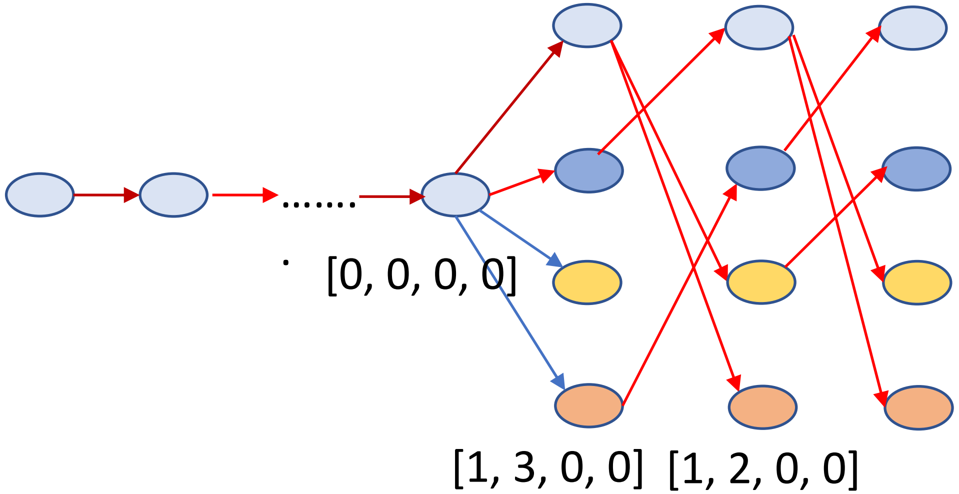
Indirect-access KV cache optimization



KV cache illustrated: https://www.axelera.ai/decoding-transformers-on-edge-devices/

Token	beam*batch_size	hidden state
t0	0	
	1	
	2	
	3	
t1	0	
	1	
	2	
	3	
...	...	
	...	
	...	
	...	
tn	0	
	1	
	2	
	3	

Cache format in pre-allocated buffer, indexing tokens with beam indices

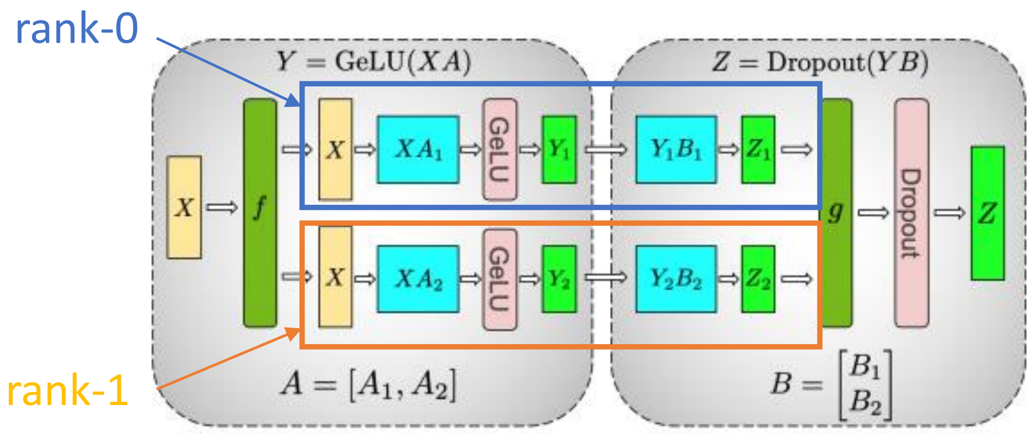


Runtime get beam sequence by traversing beam tree in reverse

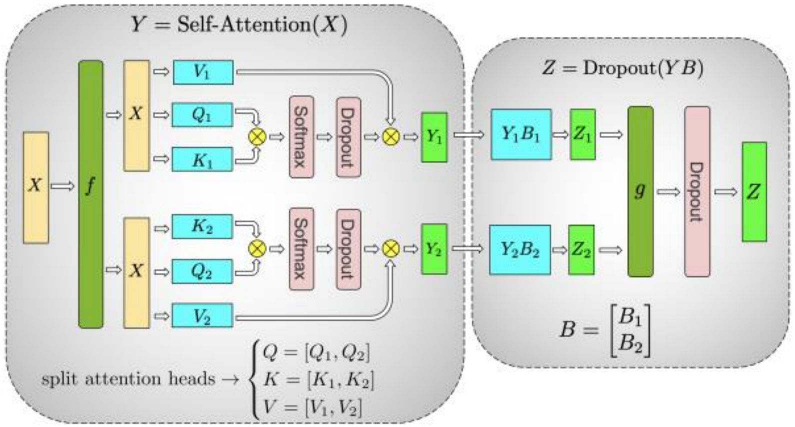
Multi-socket Scaling

- Tensor parallel based on DeepSpeed AutoTP
- Cross-socket communication via shared memory

```
...
import deepspeed
from deepspeed.accelerator import get_accelerator
import deepspeed.comm as dist
...
deepspeed.init_distributed(get_accelerator().communication_backend_name())
...
model = ...

model = deepspeed.init_inference(
    model,
    mp_size=world_size,
    base_dir=repo_root,
    dtype=infer_dtype,
    checkpoint=checkpoints_json,
    **kwargs,
)
model = ipex.optimize_transformers(model, ...)
```



(a) MLP



(b) Self-Attention

Figure 3. Blocks of Transformer with Model Parallelism. f and g are conjugate. f is an identity operator in the forward pass and all reduce in the backward pass while g is an all reduce in the forward pass and identity in the backward pass.

INT4 Accuracy Result – MLPerf GPT-J

fp32: {'rouge1': **42.9865**, 'rouge2': **20.1235**, 'rougeL': **29.9881**, 'rougeLsum': **40.1658**}
target (99.9%): {'rouge1': **42.9435**, 'rouge2': **20.1033**, 'rougeL': **29.9581**, 'rougeLsum': **40.1256**}
target (99%): {'rouge1': **42.5566**, 'rouge2': **19.9222**, 'rougeL': **29.6882**, 'rougeLsum': **39.7641**}

PVC (INT4 weight, FP16 activation), >99% of target

{'rouge1': 43.0556, 'rouge2': 20.0992, 'rougeL': 29.9936, 'rougeLsum': 40.1989}

SPR (INT4 weight, BF16 activation, per-tensor asymmetric dynamic quant to INT8 before compute), >99% of target:

{'rouge1': 42.9565, 'rouge2': 20.0886, 'rougeL': 29.9589, 'rougeLsum': 40.1175}

Both PVC and SPR achieved the 99% accuracy target

INT4 Performance Result – MLPerf GPT-J

MLPerf Model (new in v3.1)	System	Precision	Offline (samples/sec)	Server (samples/sec)
GPT-J (Intel)	2S SPR HBM	BF16 **99.9%, 99%**	1.01	0.30
	2S SPR SP	INT8 99%	2.04	0.59
	2S SPR SP	INT4+BF16 99%	1.899	0.95
	~~4x OAM PVC~~	~~INT4+FP16 99%~~	~~32.35~~	~~19~~

PVC perf numbers were not submitted to MLPerf but are MLPerf-compliant

Thoughts for oneAPI library

- Weight only quantization support for various recipes
  - New recipes are innovated quickly (e.g., int3, finer-grained block-wise quant), how to support them in a flexible way?
  - New data type support in oneDNN primitives or new patterns supported via oneDNN graph?
- Linear op fused with communication as post-op
  - All-reduce on small activation is latency bound, can be pipelined with computation via post-op fusion?
  - Incorporate the fusion support in oneDNN graph?

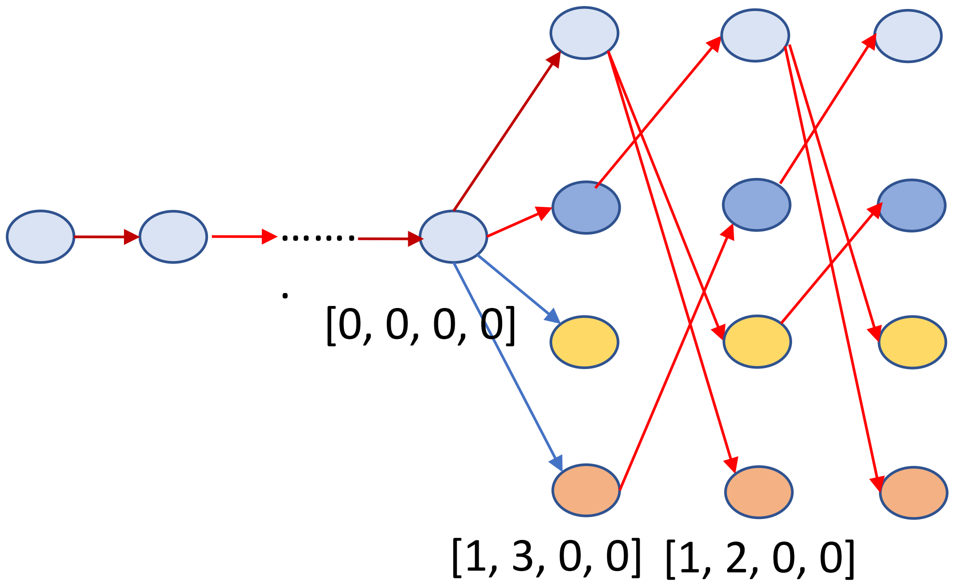
Backup

Long Sequence Optimization
--- indirect access kv_cache

- Concat/reorder_cache free
  - Store kv_cache in pre-allocate buffers -> no concat
  - Use runtime beam id sequence to get past token sequence -> no reorder_cache

Token	beam*batch_size	hidden state
t0	0	
	1	
	2	
	3	
t1	0	
	1	
	2	
	3	
...	...	
	...	
	...	
	...	
tn	0	
	1	
	2	
	3	

Cache format in pre-allocated buffer

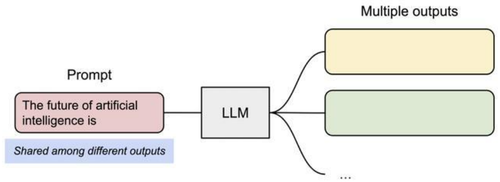


Runtime get beam sequence by traveling beam tree in reverse

Indirect access kv_cache

---Scale Dot Product kernel optimization

- Zero Copy for next token cache
  - Write-on-Computation for latest token cache
- Multiple dimension parallel
  - Sequence length/batch/head
    - Parallel reduction in matmul(attn_weight, value) to enable sequence length dim parallel
    - Distributed inference and super long sequence get obvious performance benefit
- Shared kv_cache for multiple outputs
  - Reduce memory read and get better cache locality



INT4 Kernel on PVC – Optimized with XeTLA

```
... ...
//Unpack weight, block_elems=block_size_x_b*block_size_y_b/2
xetla_vector<uint8_t, block_size_x_b * block_size_y_b> cvt_blk;
cvt_blk.xetla_select<matB_t::block_elems, 2>(0) = matB_blk & 0x0f;
cvt_blk.xetla_select<matB_t::block_elems, 2>(1) = matB_blk >> 4;
... ...
//Unpack zero_points
xetla_vector<uint8_t, block_size_x_b> zero_pt_sub;
zero_pt_sub.xetla_select<block_size_x_b / 2, 2>(0) = zero_pt_vec & 0x0f;
zero_pt_sub.xetla_select<block_size_x_b / 2, 2>(1) = zero_pt_vec >> 4;
... ...
//Convert weight to FP16 with (weight-zero_points)*scales, convert to vnni layout
xetla_vector<int32_t, block_size_x_b * block_size_y_b> cvt_blk_i32;
cvt_blk_i32 = (cvt_blk.xetla_format<int8_t>()
        - zero_pt_blk.xetla_format<int8_t>());
... ...
dst_blk.xetla_select<block_size_x_b * vnni_rows, 1>(k * block_size_x_b)
         = temp_blk.xetla_select<block_size_x_b * vnni_rows, 1>
                 (k * block_size_x_b * vnni_rows) * scale_blk;
... ...
//Compute with FP16, c=c+a*b
tile_mma::mma(matAcc, matAcc, matB_acc, matA_acc);
```