英特尔® Extension for Transformer 一石双鸟：让 LLM CPU 推理加速达 40x+ 攻克聊天场景应用难题！

作者：英特尔公司 沈海豪、罗屿、孟恒宇、董波、林俊

英特尔® Extension for Transformer 是什么？

英特尔® Extension for Transformers 是英特尔推出的一个创新工具包，可基于英特尔® 架构平台，尤其是第四代英特尔® 至强® 可扩展处理器（代号 Sapphire Rapids，SPR）显著加速基于 Transformer 的大语言模型（Large Language Model, LLM）。其主要特性包括：

- 通过扩展 Hugging Face transformers API 和利用英特尔® Neural Compressor，为用户提供无缝的模型压缩体验；
- 提供采用低位量化内核（NeurIPS 2023：在 CPU 上实现高效 LLM 推理）的 LLM 推理运行时，支持 Falcon、 LLaMA、MPT、 Llama2、 BLOOM、 OPT、 ChatGLM2、GPT-J-6B、Baichuan-13B-Base、Baichuan2-13B-Base、Qwen-7B、Qwen-14B 和 Dolly-v2-3B 等常见的 LLM；
- 先进的压缩感知运行时（NeurIPS 2022：在 CPU 上实现快速蒸馏 和 QuaLA-MiniLM：量化长度自适应 MiniLM；NeurIPS 2021：一次剪枝，一劳永逸：对预训练语言模型进行稀疏/剪枝）。

本文将重点介绍其中的 LLM 推理运行时（简称为“LLM 运行时”），以及如何利用基于 Transformer 的 API 在英特尔® 至强® 可扩展处理器上实现更高效的 LLM 推理和如何应对 LLM 在聊天场景中的应用难题。

LLM 运行时（LLM Runtime）

英特尔® Extension for Transformers 提供的 LLM Runtime 是一种轻量级但高效的 LLM 推理运行时，其灵感源于 GGML，且与 llama.cpp 兼容，具有如下特性：

- 内核已针对英特尔® 至强® CPU 内置的多种 AI 加速技术（如 AMX、VNNI），以及 AVX512F 和 AVX2 指令集进行了优化；
- 可提供更多量化选择，例如：不同的粒度（按通道或按组）、不同的组大小（如：32/128）；
- 拥有更优的 KV 缓存访问以及内存分配策略；
- 具备张量并行化功能，可助力在多路系统中进行分布式推理。

LLM Runtime 的简化架构图如下：

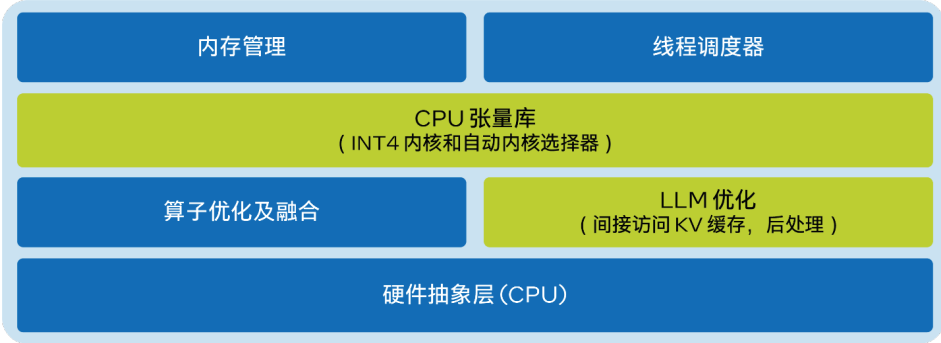

图 1. 英特尔® Extension for Transformers 的 LLM Runtime 简化架构图

使用基于 Transformer 的 API，在 CPU 上实现 LLM 高效推理

只需不到 9 行代码，即可让您在 CPU 上实现更出色的 LLM 推理性能。用户可以轻松地启用与 Transformer 类似的 API 来进行量化和推理。只需将 'load_in_4bit' 设为 true，然后从 HuggingFace URL 或本地路径输入模型即可。下方提供了启用仅限权重的（weight-only）INT4 量化的示例代码：

```
from transformers import AutoTokenizer, TextStreamer
from intel_extension_for_transformers.transformers import AutoModelForCausalLM
model_name = "Intel/neural-chat-7b-v3-1"
prompt = "Once upon a time, there existed a little girl,"

tokenizer = AutoTokenizer.from_pretrained(model_name, trust_remote_code=True)
inputs = tokenizer(prompt, return_tensors="pt").input_ids
streamer = TextStreamer(tokenizer)

model = AutoModelForCausalLM.from_pretrained(model_name, load_in_4bit=True)
outputs = model.generate(inputs, streamer=streamer, max_new_tokens=300)
```

默认设置为：将权重存储为 4 位，以 8 位进行计算。但也支持不同计算数据类型（dtype）和权重数据类型组合，用户可以按需修改设置。下方提供了如何使用这一功能的示例代码：

```
from transformers import AutoTokenizer, TextStreamer
from intel_extension_for_transformers.transformers import AutoModelForCausalLM, WeightOnlyQuantConfig
model_name = "Intel/neural-chat-7b-v3-1"
prompt = "Once upon a time, there existed a little girl,"

woq_config = WeightOnlyQuantConfig(compute_dtype="int8", weight_dtype="int4")
tokenizer = AutoTokenizer.from_pretrained(model_name, trust_remote_code=True)
inputs = tokenizer(prompt, return_tensors="pt").input_ids
streamer = TextStreamer(tokenizer)

model = AutoModelForCausalLM.from_pretrained(model_name,quantization_config=woq_config)
outputs = model.generate(inputs, streamer=streamer, max_new_tokens=300)
```

性能测试

经过持续努力，上述优化方案的 INT4 性能得到了显著提升。本文在搭载英特尔® 至强® 铂金 8480+ 的系统上与 llama.cpp 进行了性能比较；系统配置详情如下：@3.8 GHz，56 核/路，启用超线程，启用睿频，总内存 256 GB (16 x 16 GB DDR5 4800 MT/s [4800 MT/s])，BIOS 3A14.TEL2P1，微代码 0x2b0001b0，CentOS Stream 8。

当输入大小为 32、输出大小为 32、beam 为 1 时的推理性能测试结果，详见下表：

模型	首个 token 时延 (ms)	下一个 token 时延 (ms)	llama.cpp 首个 token (ms)	llama.cpp 下一个 token (ms)
GPT-J-6B	53.17	19.98	546.33	28.4
LLAMA2-7B-Chat	55.52	21.96	344.49	32.19
LLAMA-7B	56.73	22.04	324.85	31.55
MPT-7B	61.8	21.05	321.31	31.55
Falcon-7B	49.44	22.26	344.7	31.2
GPT-NeoX-20B	179.08	61.21	1361.19	100.03
ChatGLM-6B	61.89	22.28	423.8	31.09
ChatGLM2-6B	56.97	21.69	484.1	26.83
Mistral-7B	65.88	25.03	n/a	n/a
Starcoder-3b	126.94	16.91	217.07	33.71
Dolly-v2-3b	57.9	13.41	907.44	23.1
Baichuan-13B	143.8	43.28	842.63	74.06
Baichuan2-13B	129.65	41.64	831.89	69.22
Qwen-7B	135.53	26.87	n/a	n/a
Qwen-14B	115.31	45.87	n/a	n/a

表 1. LLM Runtime 与 llama.cpp 推理性能比较（输入大小 = 32，输出大小 = 32， beam = 1）

输入大小为 1024、输出大小为 32、beam 为 1 时的推理性能的测试结果，详见下表：

模型	首个 token 时延 (ms)	下一个 token 时延 (ms)	llama.cpp 首个 token (ms)	llama.cpp 下一个 token (ms)
GPT-J-6B	685.42	21.72	10953.2	31.4
LLAMA2-7B-Chat	857.86	24.46	11957.3	35.9
LLAMA-7B	848.21	24.26	12239.28	36.92
MPT-7B	1144.15	23.04	27676.7	61.92
Falcon-7B	929.32	27.19	14305.8	34.67
GPT-NeoX-20B	3637.11	65.45	33850.3	85.4
ChatGLM-6B	2897.44	27.38	12315.4	35.113
ChatGLM2-6B	929.97	23.88	10684	29.431
Mistral-7B	790.21	30.69	n/a	n/a
Starcoder-3b	2648.3	22.35	n/a	n/a
Dolly-v2-3b	899.98	15.2	9653.8	26
Baichuan-13B	1426	45.35	58088	89.94
Baichuan2-13B	1216	47.78	44567	102.69
Qwen-7B	1395.04	30.04	n/a	n/a
Qwen-14B	1827.66	48.98	n/a	n/a

表 2. LLM Runtime 与 llama.cpp 推理性能比较（输入大小 = 1024，输出大小 = 32， beam = 1）

根据上表 2 可见：与同样运行在第四代英特尔® 至强® 可扩展处理器上的 llama.cpp 相比，无论是首个 token 还是下一个 token，LLM Runtime 都能显著降低时延，且首个 token 和下一个 token 的推理速度分别提升多达 40 倍[1]（Baichuan-13B，输入为 1024）和 2.68 倍[2]（MPT-7B，输入为 1024）。llama.cpp 的测试采用的是默认代码库。

而综合表 1 和表 2 的测试结果，可得：与同样运行在第四代英特尔® 至强® 可扩展处理器上的 llama.cpp 相比，LLM Runtime 能显著提升诸多常见 LLM 的整体性能：在输入大小为 1024 时，实现 3.58 到 21.5 倍的提升；在输入大小为 32 时，实现 1.76 到 3.43 倍的提升[3]。

准确性测试

英特尔® Extension for Transformers 可利用英特尔® Neural Compressor 中的 SignRound、RTN 和 GPTQ 等量化方法，并使用 lambada_openai、piqa、winogrande 和 hellaswag 数据集验证了 INT4 推理准确性。下表是测试结果平均值与 FP32 准确性的比较。

模型	方法	平均值		
		FP32	int4	
		ACC	ACC	比率
EleutherAI/gpt-j-6b	RTN	0.643375	0.64105	0.996386
decapoda-research/llama-7b-hf	signround	0.688675	0.682925	0.991651
meta-llama/Llama-2-7b-hf	signround	0.69015	0.6868	0.99521
databricks/dolly-v2-3b	GPTQ	0.61345	0.609	0.992746
EleutherAI/gpt-neox-20b	GPTQ	0.673975	0.6687	0.992173
mosaicml/mpt-7b	RTN	0.689	0.683425	0.991909
tiiuae/falcon-7b	GPTQ	0.69815	0.693975	0.99402
baichuan-inc/Baichuan-13B-Base	GPTQ	0.49885	0.49475	0.991781
baichuan-inc/Baichuan2-13B-Base	GPTQ	0.6932	0.68785	0.992282
Qwen-7B	GPTQ	0.683175	0.6726	0.984521

表 3. INT4 与 FP32 准确性对比

从上表 3 可以看出，多个模型基于 LLM Runtime 进行的 INT4 推理准确性损失微小，几乎可以忽略不记。我们验证了很多模型，但由于篇幅限制此处仅罗列了部分内容。如您欲了解更多信息或细节，请访问此链接：https://medium.com/@NeuralCompressor/llm-performance-of-intel-extension-for-transformers-f7d061556176。

更先进的功能：满足 LLM 更多场景应用需求

同时，LLM Runtime 还具备双路 CPU 的张量并行化功能，是较早具备此类功能的产品之一。未来，还会进一步支持双节点。

然而，LLM Runtime 的优势不仅在于其更出色的性能和准确性，我们也投入了大量的精力来增强其在聊天应用场景中的功能，并且解决了 LLM 在聊天场景中可能会遇到的以下应用难题：

1. 对话不仅关乎 LLM 推理，对话历史也很有用。
2. 输出长度有限：LLM 模型预训练主要基于有限的序列长度。因此，当序列长度超出预训练时使用的注意力窗口大小时，其准确性便会降低。
3. 效率低下：在解码阶段，基于 Transformer 的 LLM 会存储所有先前生成的 token 的键值状态（KV），从而导致内存使用过度，解码时延增加。

关于第一个问题，LLM Runtime 的对话功能通过纳入更多对话历史数据以及生成更多输出加以解决，而 llama.cpp 目前尚未能很好地应对这一问题。

关于第二和第三个问题，我们将流式 LLM（Steaming LLM）集成到英特尔® Extension for Transformers 中，从而能显著优化内存使用并降低推理时延。

Streaming LLM

与传统 KV 缓存算法不同，我们的方法结合了***注意力汇聚（Attention Sink）（4 个初始 token）***以提升注意力计算的稳定性，并借助***滚动 KV 缓存保留最新的 token***，这对语言建模至关重要。该设计具有强大的灵活性，可无缝集成到能够利用旋转位置编码 RoPE 和相对位置编码 ALiBi 的自回归语言模型中。

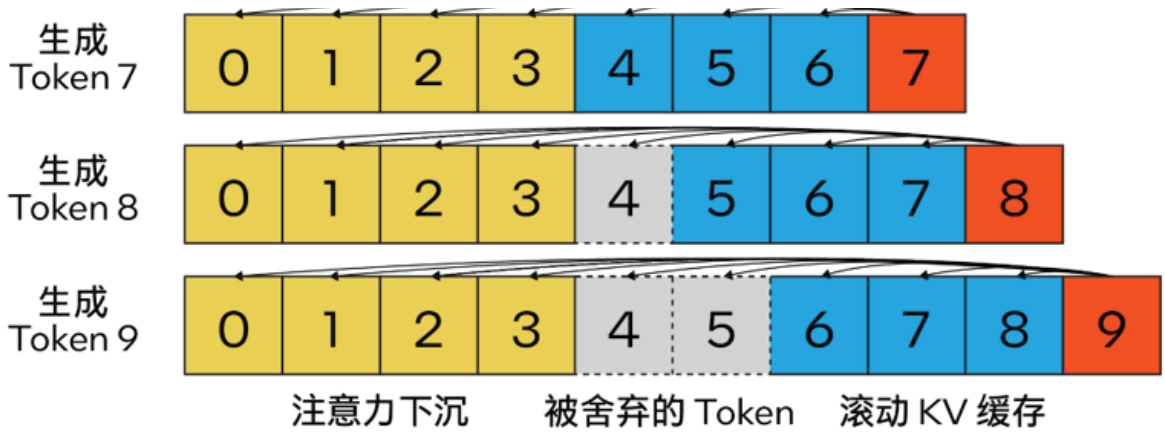

图 2. Steaming LLM 的 KV 缓存（图片来源：通过注意力下沉实现高效流式语言模型 ）

此外，与 llama.cpp 不同，本优化方案还引入了“*n_keep*”和“*n_discard*”等参数来增强 Streaming LLM 策略。用户可使用前者来指定要在 KV 缓存中保留的 token 数量，并使用后者来确定在已生成的 token 中要舍弃的数量。为了更好地平衡性能和准确性，系统默认在 KV 缓存中舍弃一半的最新 token。

同时，为进一步提高性能，我们还将 Streaming LLM 添加到了 MHA 融合模式中。如果模型是采用旋转位置编码（RoPE）来实现位置嵌入，那么只需针对现有的 K-Cache 应用“移位运算（shift operation）”，即可避免对先前生成的、未被舍弃的 token 进行重复计算。这一方法不仅充分利用了长文本生成时的完整上下文大小，还能在 KV 缓存上下文完全被填满前不产生额外开销。

“shift operation” 依赖于旋转的交换性和关联性，或复数乘法。例如：如果某个 token 的 K-张量初始放置位置为 m 并且旋转了 $m \times \theta_i$ for $i \in [0, d/2)$，那么当它需要移动到 $m-1$ 这个位置时，则可以旋转回到 $(-1) \times \theta_i$ for $i \in [0, d/2)$。这正是每次舍弃 n_discard 个 token 的缓存时发生的事情，而此时剩余的每个 token 都需要“移动” n_discard 个位置。下图以 “n_keep = 4、n_ctx = 16、n_discard = 1” 为例，展示了这一过程。

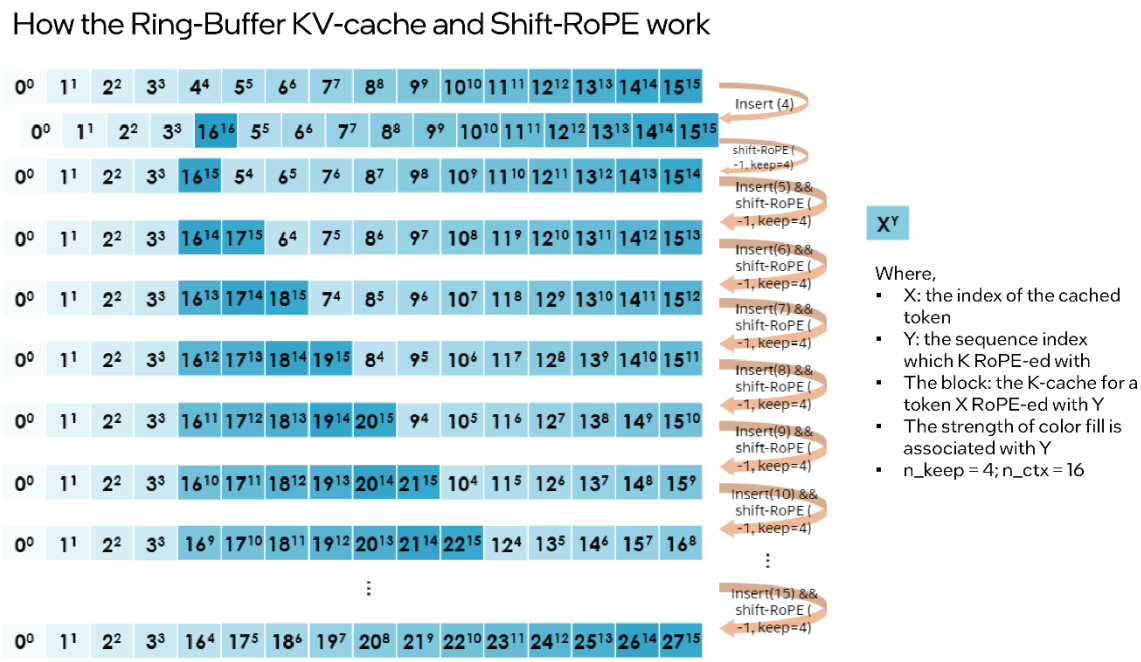

图 3. Ring-Buffer KV-Cache 和 Shift-RoPE 工作原理

需要注意的是：融合注意力层无需了解上述过程。如果对 K-cache 和 V-cache 进行相同的洗牌，注意力层会输出几乎相同的结果（可能存在因浮点误差导致的微小差异）。

您可通过以下代码启动 Streaming LLM：

```
from transformers import AutoTokenizer, TextStreamer
from intel_extension_for_transformers.transformers import AutoModelForCausalLM, WeightOnlyQuantConfig
model_name = "Intel/neural-chat-7b-v1-1"    # Hugging Face model_id or local model
woq_config = WeightOnlyQuantConfig(compute_dtype="int8", weight_dtype="int4")
prompt = "Once upon a time, a little girl"

tokenizer = AutoTokenizer.from_pretrained(model_name, trust_remote_code=True)
inputs = tokenizer(prompt, return_tensors="pt").input_ids
streamer = TextStreamer(tokenizer)

model = AutoModelForCausalLM.from_pretrained(model_name, quantization_config=woq_config, trust_remote_code=True)

# Recommend n_keep=4 to do attention sinks (four initial tokens) and n_discard=-1 to drop half rencetly tokens when meet length threshold

outputs = model.generate(inputs, streamer=streamer, max_new_tokens=300, ctx_size=100, n_keep=4, n_discard=-1)
```

结论与展望

本文基于上述实践经验，提供了一个在英特尔® 至强® 可扩展处理器上实现高效的低位（INT4）LLM 推理的解决方案，并且在一系列常见 LLM 上验证了其通用性以及展现了其相对于其他基于 CPU 的开源解决方案的性能优势。未来，我们还将进一步提升 CPU 张量库和跨节点并行性能。

欢迎您试用英特尔® Extension for Transformers，并在英特尔® 平台上更高效地运行 LLM 推理！也欢迎您向代码仓库（repository）提交修改请求（pull request）、问题或疑问。期待您的反馈！

特别致谢

在此致谢为此篇文章做出贡献的英特尔公司人工智能资深经理张瀚文及工程师许震中、余振滔、刘振卫、丁艺、王哲、刘宇澄。

[1] 根据表 2 Baichuan-13B 的首个 token 测试结果计算而得。

[2] 根据表 2 MPT-7B 的下一个 token 测试结果计算而得。

[3] 整体性能 = 首个 token 性能 + 31 * 下一个 token 性能。