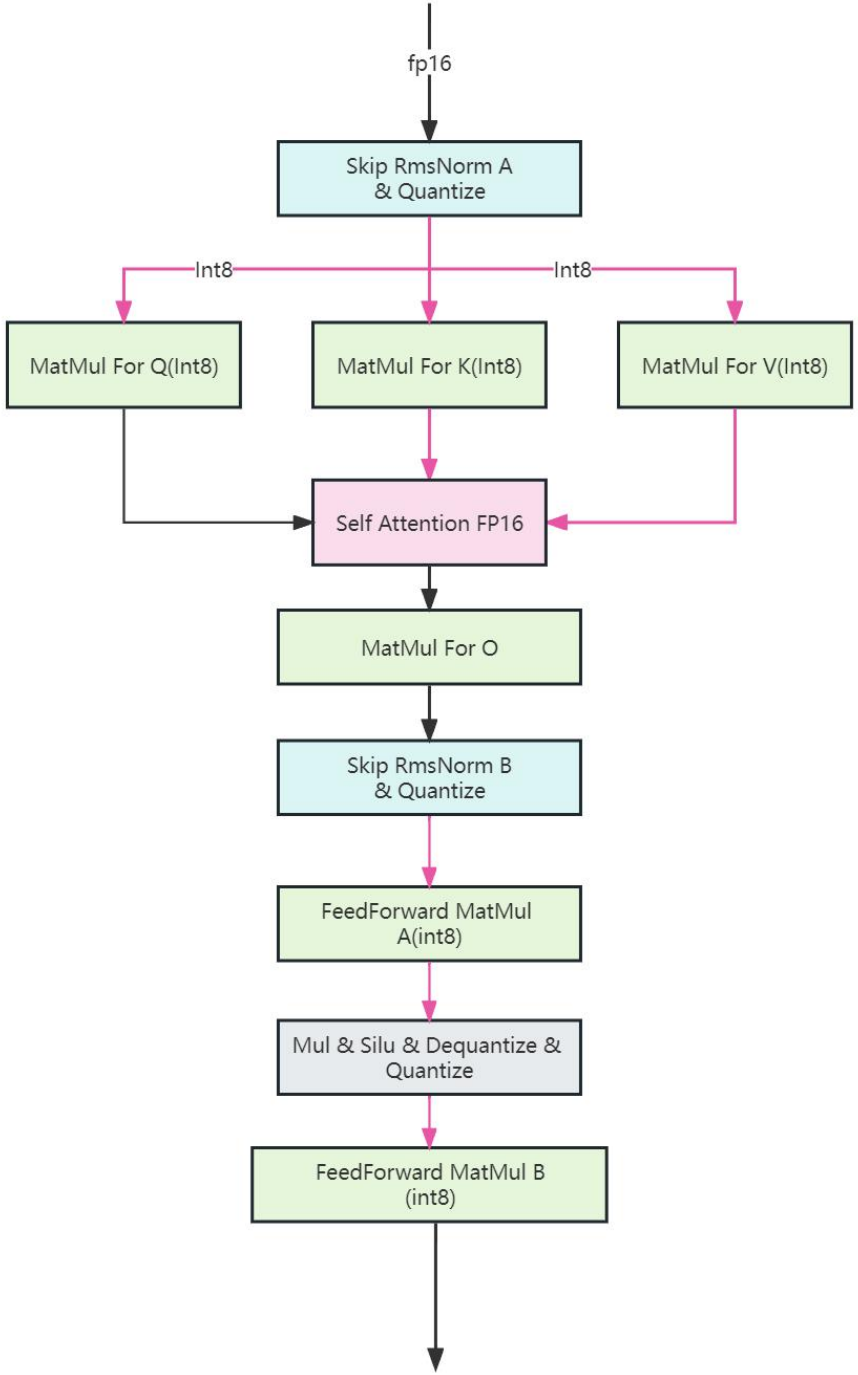
高性能大语言模型推理

Sensetime HPC Group

LLM

- 大语言模型的基本结构
  - Layernorm
  - Attention
  - Silu
  - MatMul
  - Rotary Embedding
  - KV Cache



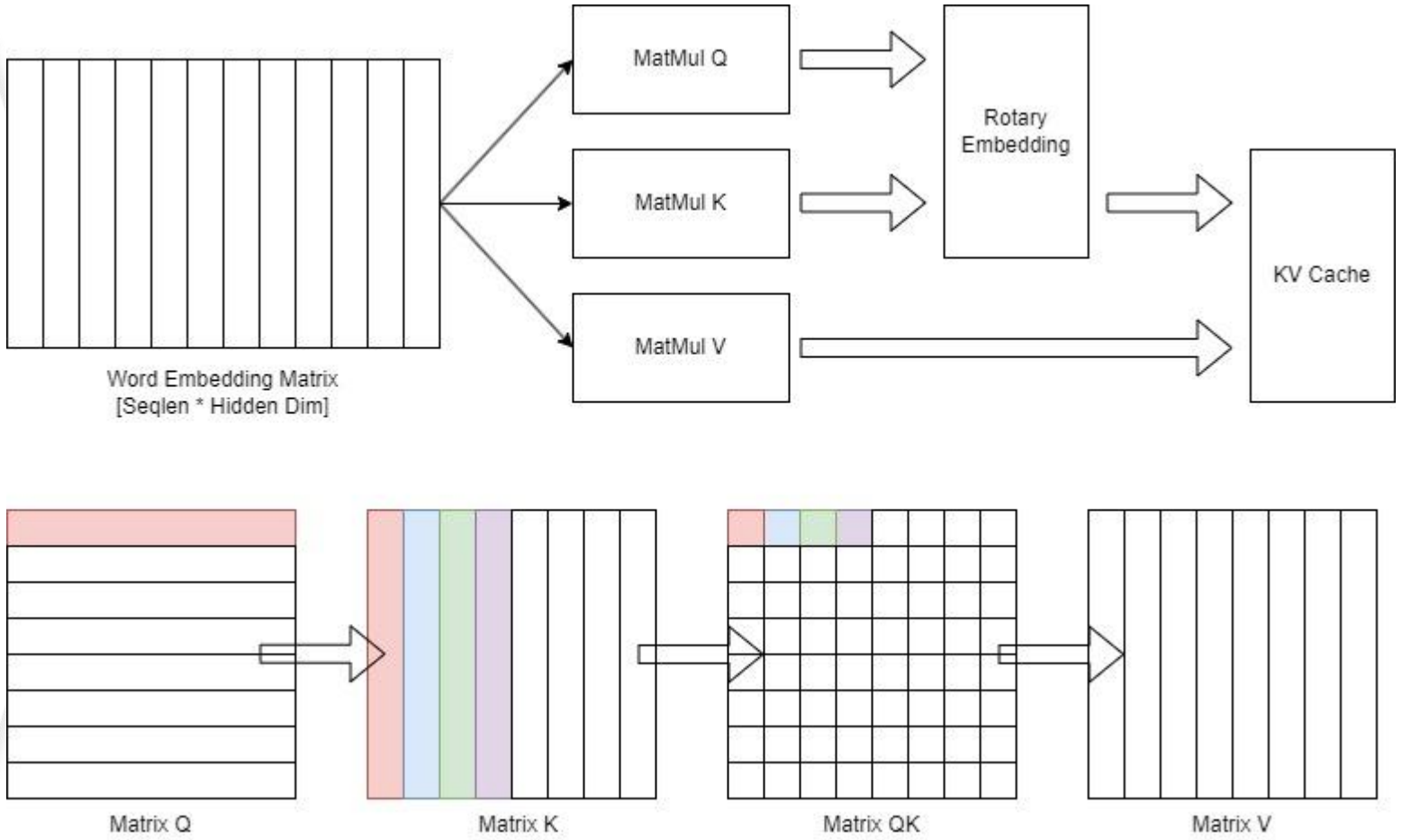
LLM

- 大语言模型的基本结构
  - Layernorm
  - Attention
  - Silu
  - MatMul
  - Rotary Embedding
  - KV Cache

Opname	Type	Gpu time
Matmul For Q	Gemm	30
Matmul For K	Gemm	31
Matmul For V	Gemm	30
MatMul For O	Gemm	31
SkipRMSNorm A	Normalization	5
FeedForward Gemm A	Gemm	70
FeedForward Gemm B	Gemm	71
FeedForward Gemm C	Gemm	73
SkipRMSNorm B	Normalization	5
Attention	Attention	9 (会根据输入长度变化哦)

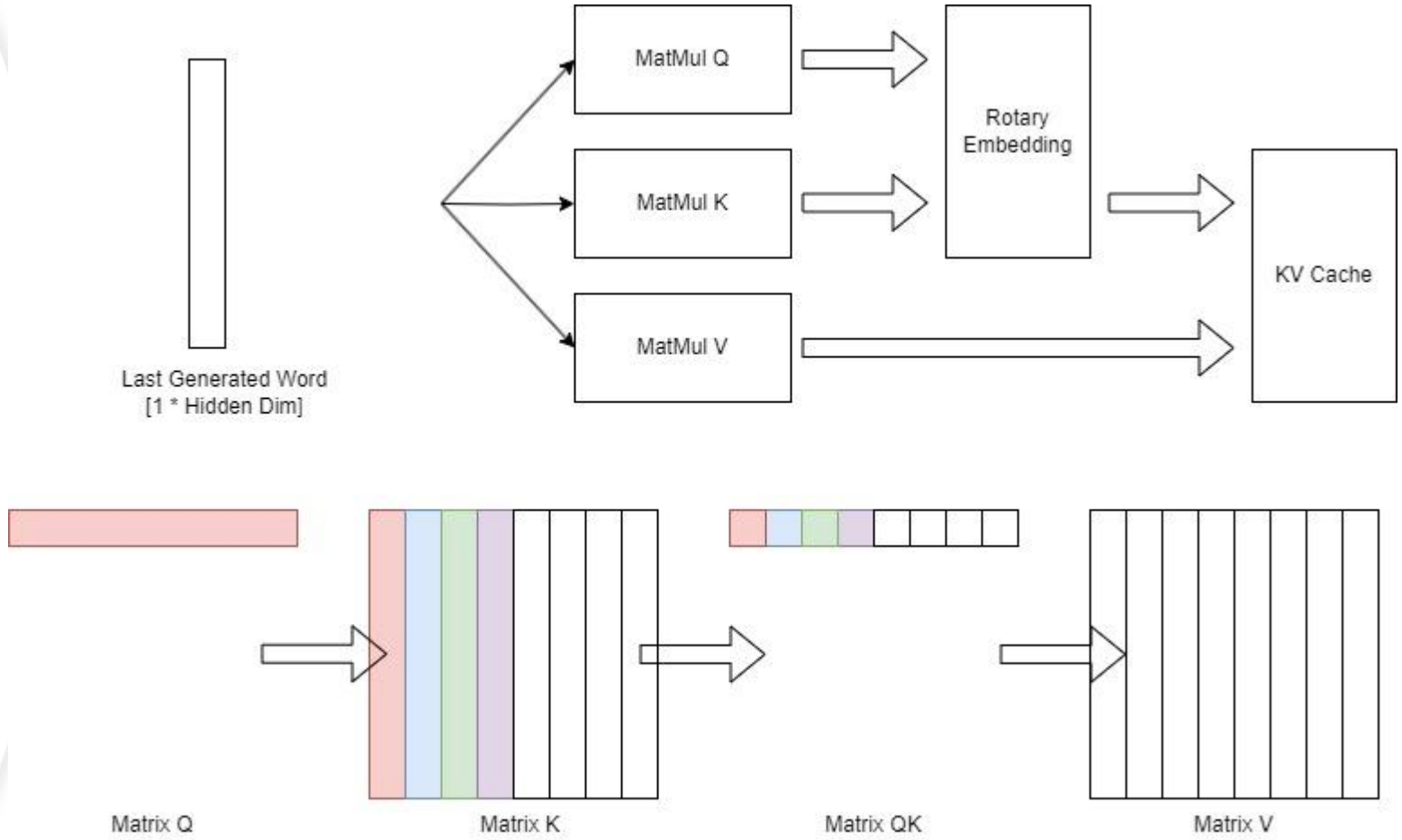
LLM Inference

- PPL.LLM 将推理分成
  - Prefill 阶段
  - Decoding 阶段



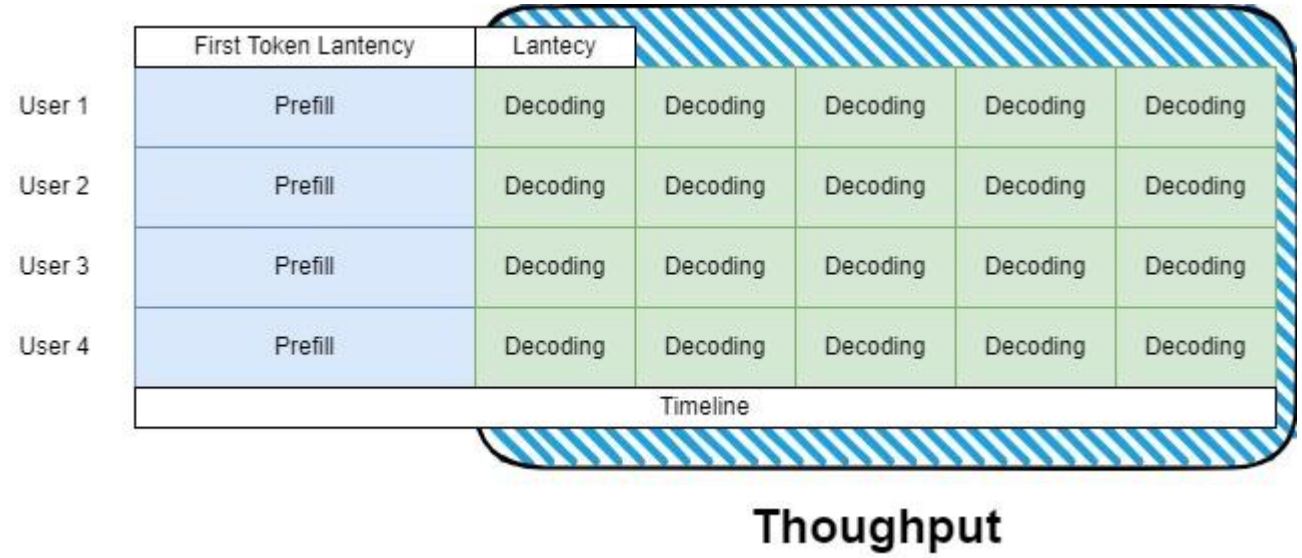
LLM Inference

- PPL.LLM 将推理分成
  - Prefill 阶段
  - Decoding 阶段



LLM Benchmark

- Thoughtput(吞吐量)
- First Token Latency(首字延迟)
- Latency(延迟)
- QPS(每秒请求数)



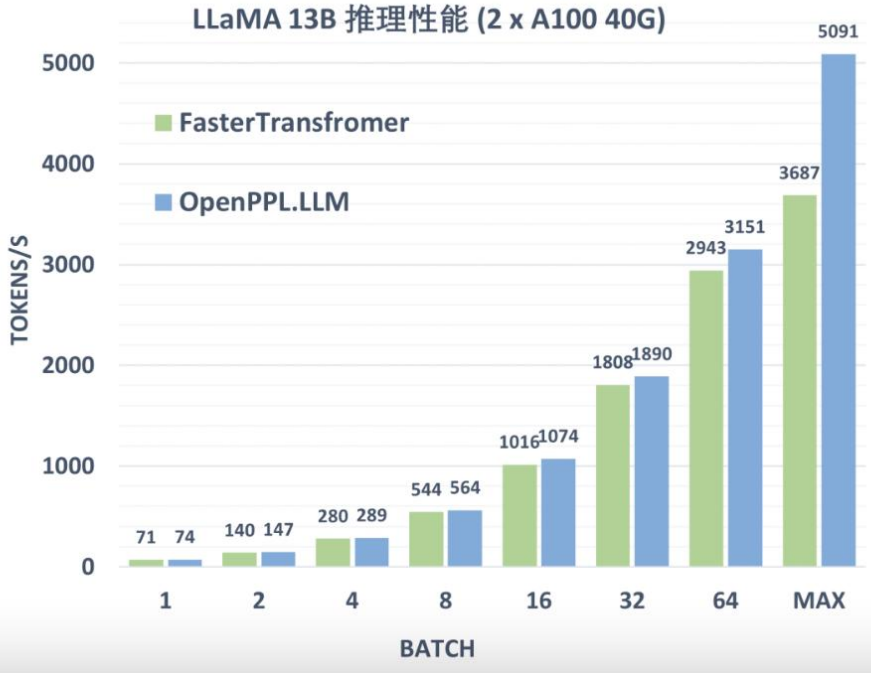
LLM Benchmark

- Thoughtput(吞吐量)
- First Token Latency(首字延迟)
- Latency(延迟)
- QPS(每秒请求数)

LLaMA		FP16	H800	PPL.NN.LLM				CUDA 12.0
model size(B)	gpu	batch	input len	output len	prefill(ms)	decode(ms)	throughput(tokens/s)	mem(GiB)
7	1	1	8	256	6.78	6.83	145.85	13.26
7	1	2	8	256	6.84	6.91	288.32	13.34
7	1	4	8	256	7.12	6.95	573.66	13.50
7	1	8	8	256	7.10	7.05	1131.10	13.82
7	1	16	8	256	7.94	7.15	2229.65	14.47
7	1	32	8	256	10.41	7.98	3992.18	15.77
7	1	64	8	256	16.36	8.55	7429.85	18.36
7	1	128	8	256	30.03	10.47	12095.66	23.57
7	1	256	8	256	58.69	15.20	16597.23	33.97
7	1	384	8	256	83.42	22.90	16536.86	44.36
7	1	512	8	256	110.82	25.45	19785.23	54.78
7	1	768	8	256	165.19	37.19	20298.52	75.57
7	1	1	1024	1024	30.86	8.64	115.40	13.86
7	1	2	1024	1024	58.33	8.75	227.09	14.56
7	1	4	1024	1024	109.13	8.88	445.36	15.93
7	1	8	1024	1024	213.48	9.23	848.04	18.70
7	1	16	1024	1024	415.95	10.06	1528.73	24.22
7	1	32	1024	1024	830.35	13.84	2184.17	35.27
7	1	64	1024	1024	1633.89	20.18	2939.07	57.37
7	1	80	1024	1024	2078.37	22.15	3309.25	68.42

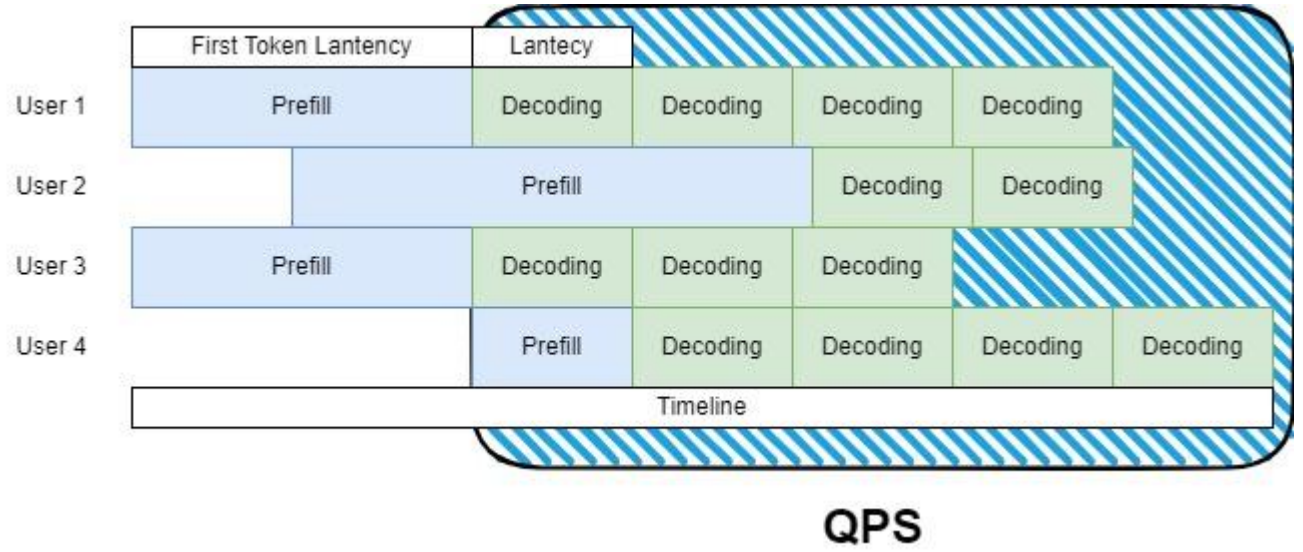
LLM Benchmark

- Thoughtput(吞吐量)
- First Token Latency(首字延迟)
- Latency(延迟)
- QPS(每秒请求数)



PPL Serving

- Thoughtput(吞吐量)
- First Token Latency(首字延迟)
- Latency(延迟)
- QPS(每秒请求数) = K / 完成K个请求的时间

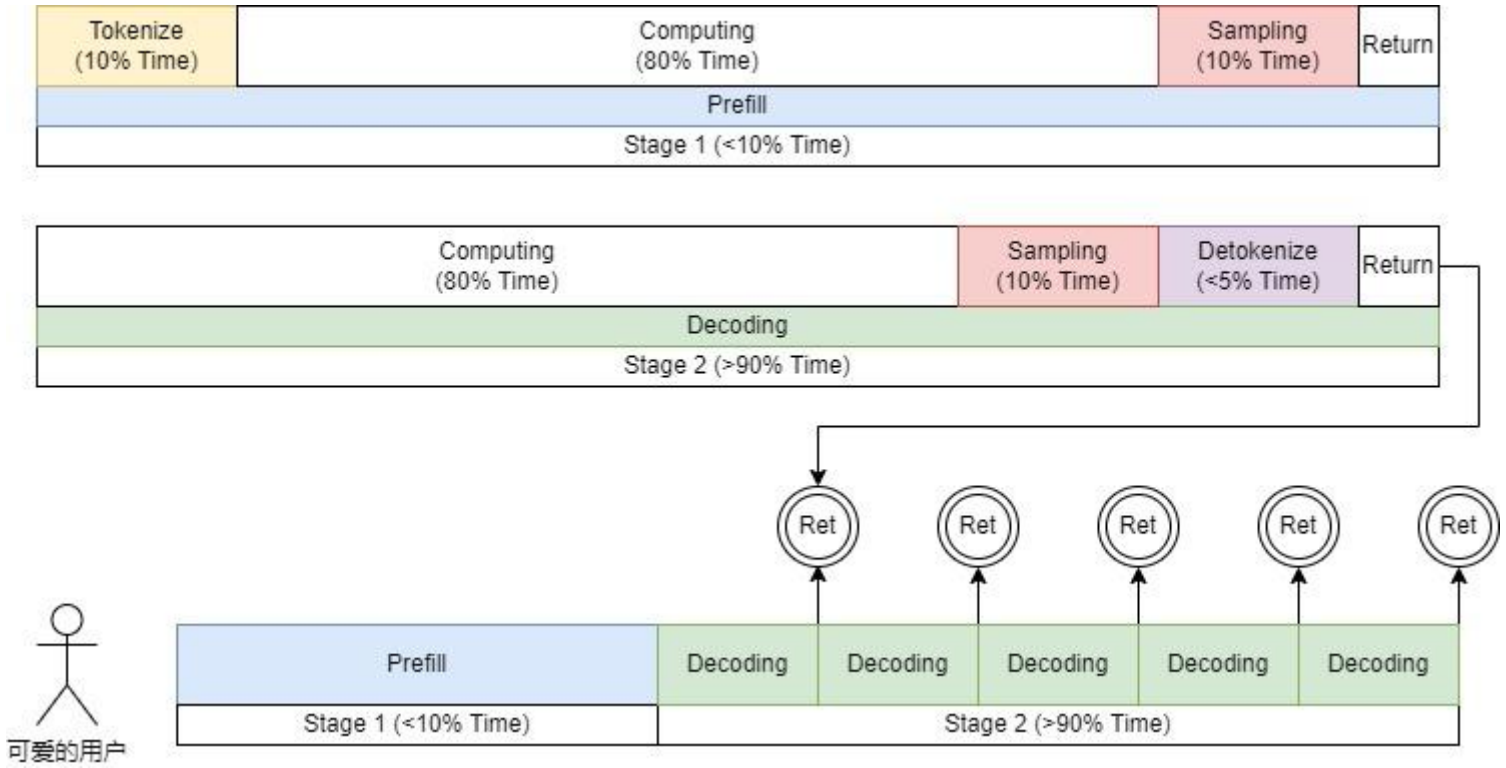


Dataset: https://github.com/openppl-public/ppl.llm.serving/blob/master/tools/samples_1024.json

LLM Inference

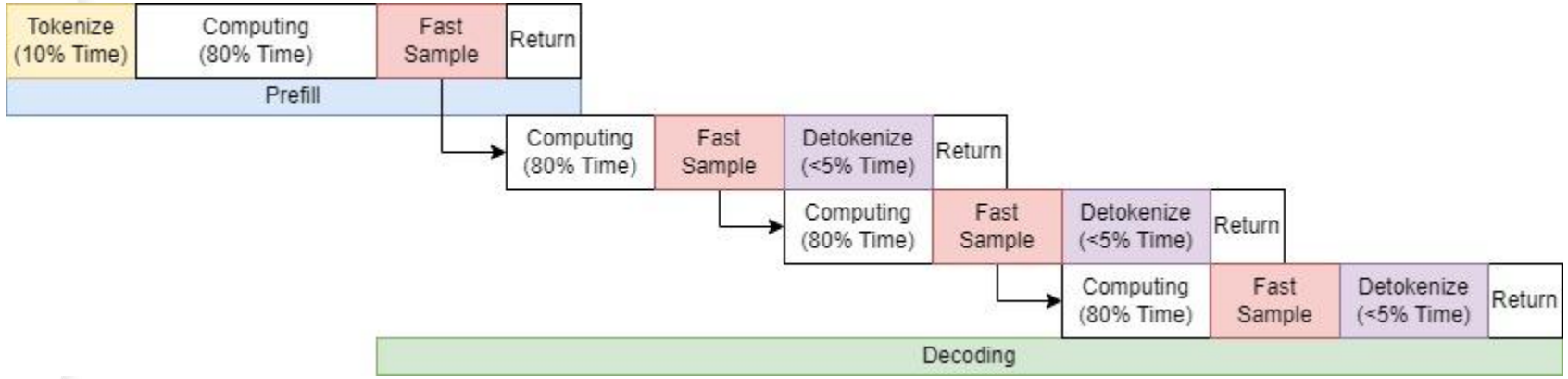
- LLM 的推理由一些子过程构成：
  - Tokenize(将文本转换为向量)
  - Computing(模型推理)
  - Detokenize(将向量转换为文本)
  - Sampling(依据推理结果进行采样)

在每一个 Decoding 阶段结束时，我们都返回一次当前生成的结果，这也被称为流式传输。



优化：流水线前后处理与高性能采样

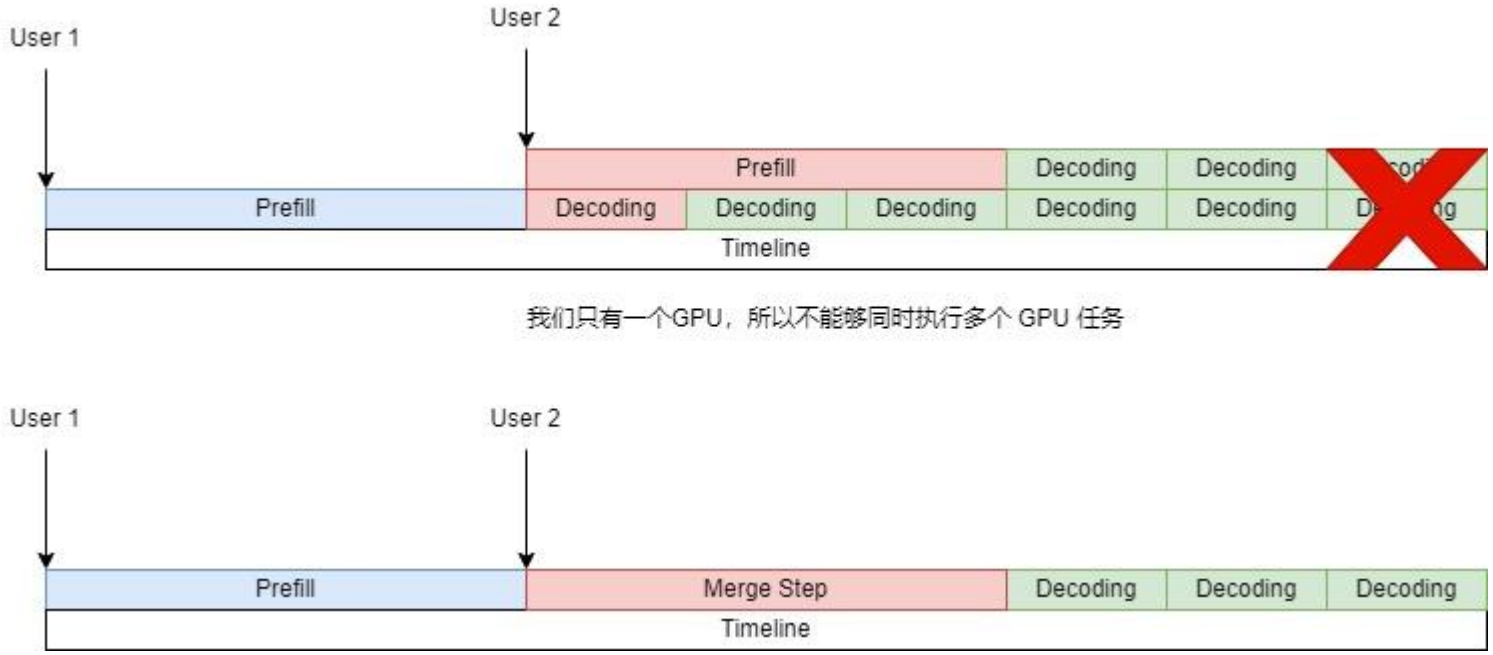
- 使用流水线策略将 Tokenize(CPU)，Computing(GPU)，Detokenize(CPU)之间的延迟相互掩盖，这将提升系统5%~10%的QPS 。
- 使用更高性能的采样算子将提升系统10%~20%的QPS 。



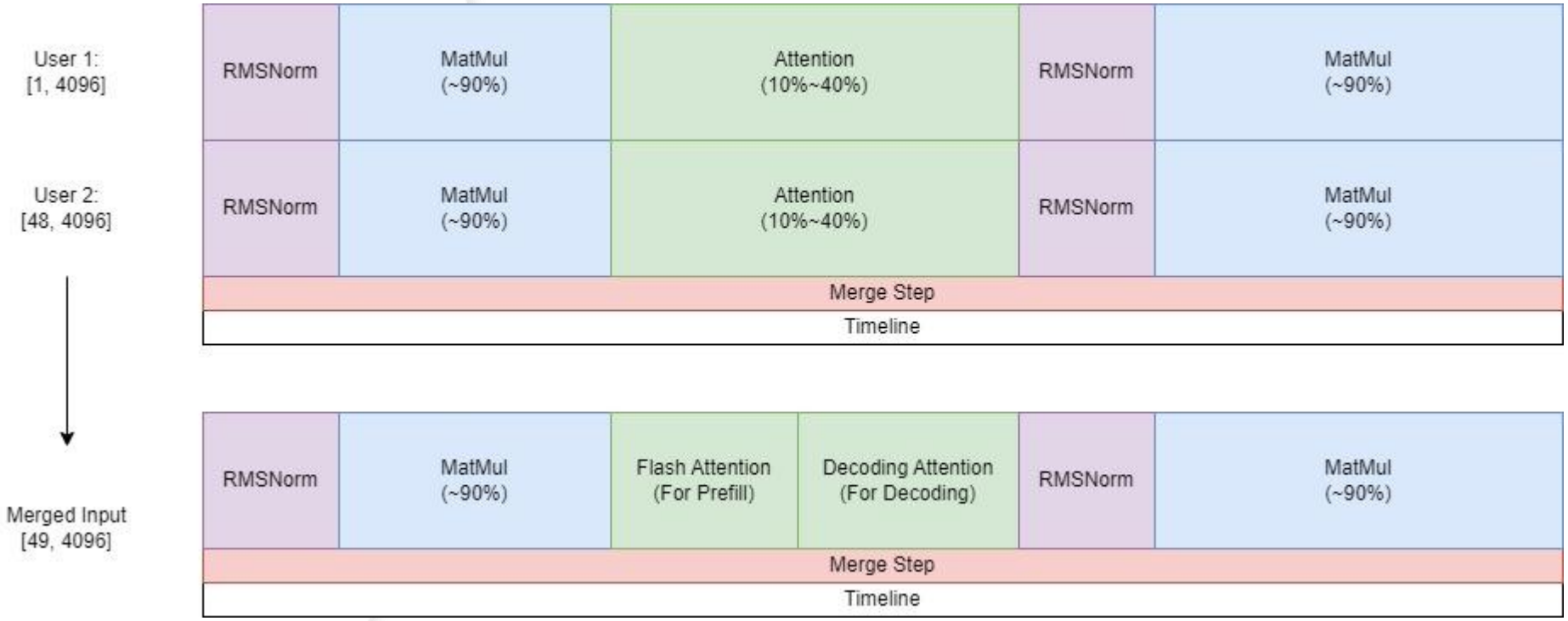
链接：https://github.com/openppl-public/ppl.llm.kernel.cuda/blob/master/src/ppl/kernel/llm/cuda/pmx/sample.cu

优化：动态批处理

- PPL.LLM 能够在多用户推理中动态地重组任务，从而大幅降低用户请求延迟。
- 这将使得系统的 QPS 提升 100%。



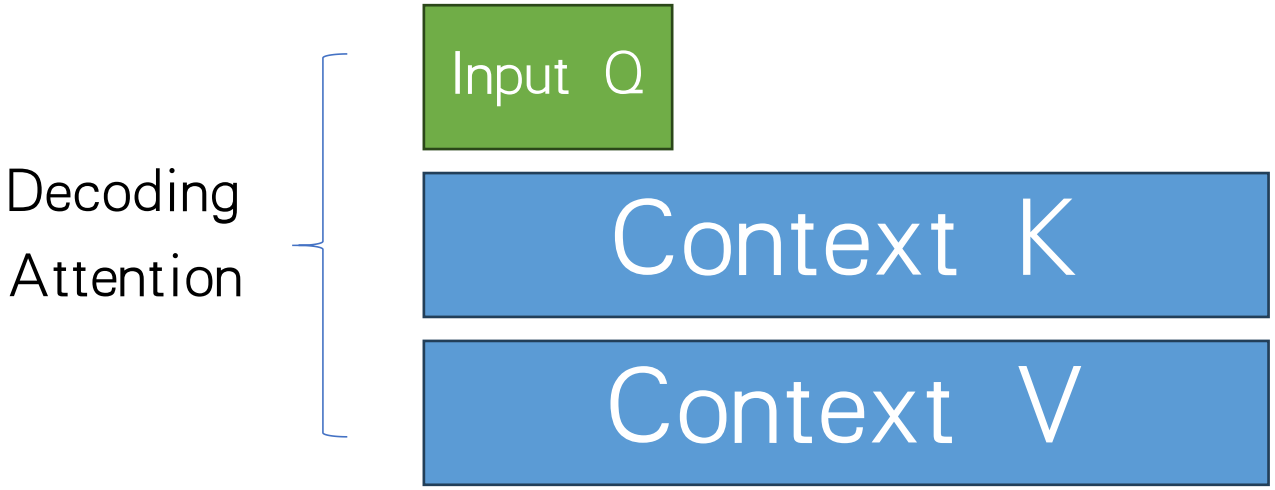
优化：
Decoding Attention



在Merge Step中，用户会被分流使用两种不同的 Attention 算子，
其中使用 Decoding Attention 将比 Flash attention 快 2 倍。

优化：
Decoding Attention

- 这是一种专为 Decoding 任务设计的算子，其完全依赖 Cuda Core 完成计算，非常灵活且容易修改。
- 在 Decoding Attention 运算中，要求 Q 的长度为 1，K，V 长度可以不为 1




```
attn = query @ key.transpose(-2, -1)
attn = attn.softmax(-1)
return attn @ value
```

优化：
Decoding Attention

- 这是一种专为 Decoding 任务设计的算子，其完全依赖 Cuda Core 完成计算，非常灵活且容易修改。
- 在 Decoding Attention 运算中，要求 Q 的长度为 1，K，V 长度可以不为 1

```cpp
template<
    int32_t HEAD_SIZE,
    int32_t THREAD_GROUP_SIZE,        // how many thread group inside a warp
    int32_t TPB>
__global__
void _DecodingAttention_fp16(
    fp16_t* __restrict__ out,          // [context_lens, num_heads, head_size]
    const fp16_t* __restrict__ Q,      // [seq_lens, num_heads, head_size]
    const fp16_t* __restrict__ K,      // [context_lens, num_heads, head_size]
    const fp16_t* __restrict__ V,      // [context_lens, num_heads, head_size]
    const fp32_t  attn_scale,
    const int32_t *context_lens_cumsum // something like: [0, 160, 160+2, 160+2+3]
) {
    /***
    * 请一定记得，这个 Kernel 是一个兄弟在 2023年7月20号 夜里搓出来的，那一天北京经历了入夏以来的最强暴雨

    DecodingAttention is a special operator designed specifically for large language models(LLM) decoding.

    It requires that the length of each input Query is always 1,
        while the Key and Value can have different lengths.

    This operator supports padding removal optimization, meaning that Q, K, and V all need to have their tokens
        concentrated in one sentence for input, with shapes like Q: [seq_lens, num_heads, head_size],
        and K: [context_lens, num_heads, head_size].

    Since the Query sentence length is always 1, this operator is literally a fused matrix-vector multiplications
        It does not utilize tensor cores for computation.

    The calculation logic is divided into three steps: gemv(QK) + softmax(Attention) + gemv(KV).
        In the provided code, it has already been split into these three parts.
    ***/

    /* --- Decoding Attention Kernel Implementation --- */
```

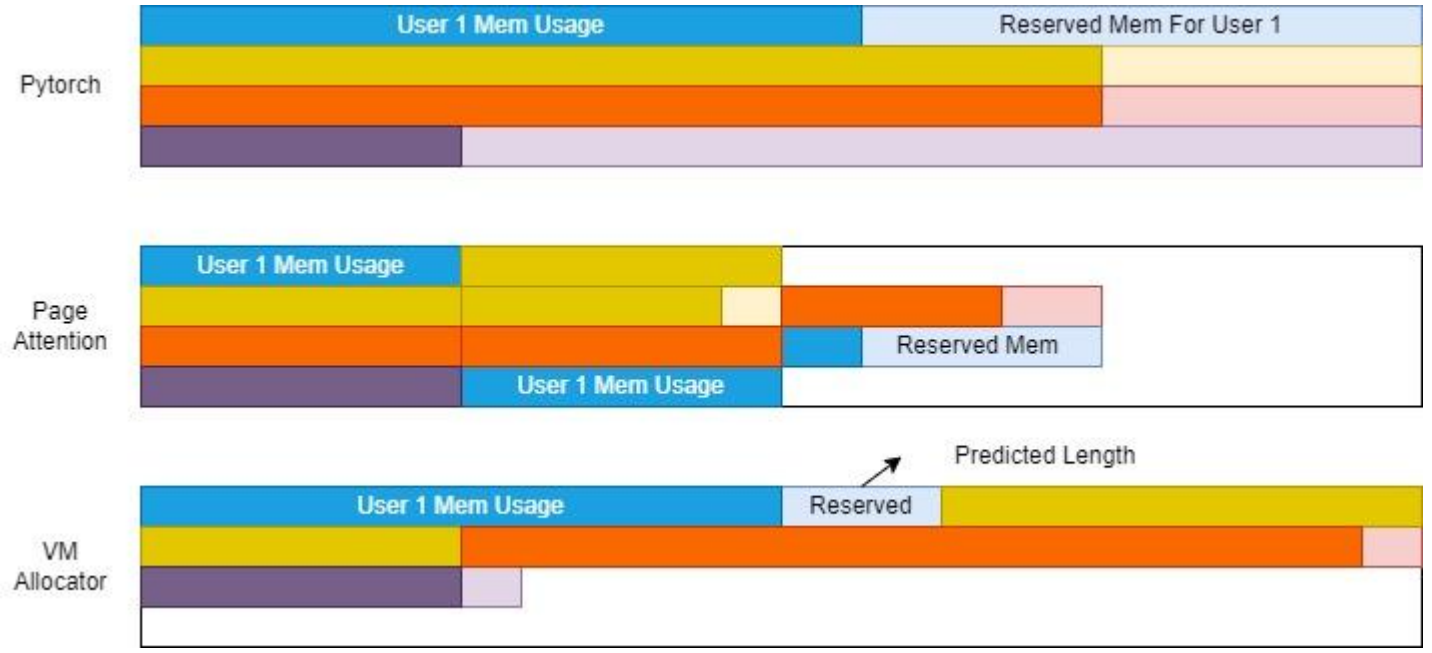
https://github.com/openppl-public/ppl.llm.kernel.cuda/blob/master/src/ppl/kernel/llm/cuda/pmx/multi_head_cache_attention.cu

优化：
VM Allocator

- PPL.LLM 使用 VM Allocator 管理内存，这是一种简单的内存机制，用于分配 KV Cache 所需的空间，并提升系统性能。
- 这将使得系统的 QPS 提升 200%，能够同时服务更多用户。

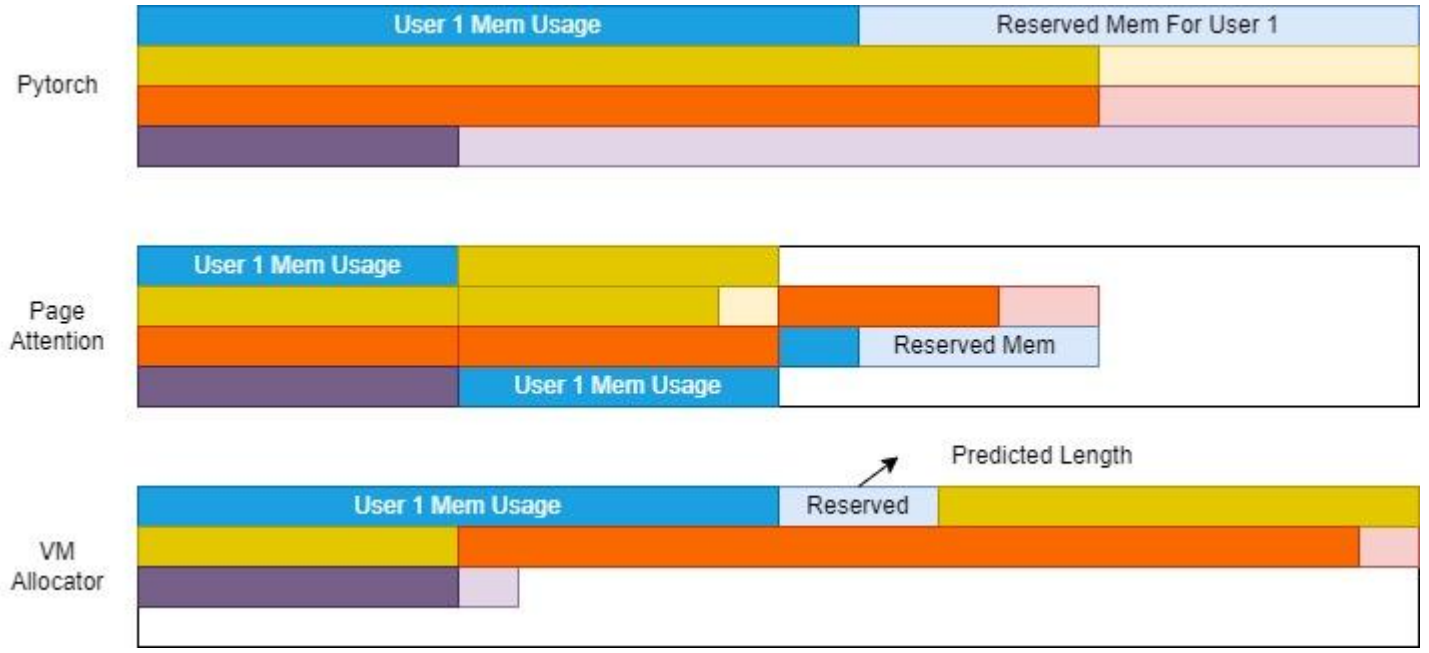
优化：
Cache 显存管理

- PPL.LLM 使用 VM Allocator 管理内存，这是一种简单的内存机制，用于分配 KV Cache 所需的空间，并提升系统性能。
- 这将使得系统的 QPS 提升 200%，能够同时服务更多用户。



优化：
Cache 显存管理

- 并非加入了 Page Attention 或 VM Allocator 就一定可以提升吞吐量，这与模型的显存使用情况有关。
- 上述技术可以减少 KV 缓存的显存占用，提升 batchsize，但 batchsize 超过 512 时对吞吐量的提升将不再明显。



优化：
Cache 显存管理

- ChatGLM2，Llama-2 慎重开 PAGE ATTENTION。
- 这些模型使用了GQA或MQA技术压缩KV缓存，batchsize可以上千，PAGE ATTENTION 不会很有用，反而可能会拖性能。

LLaMA		FP16	H800	PPL.NN.LLM				CUDA 12.0
model size(B)	gpu	batch	input len	output len	prefill(ms)	decode(ms)	throughput(tokens/s)	mem(GiB)
7	1	1	8	256	6.78	6.83	145.85	13.26
7	1	2	8	256	6.84	6.91	288.32	13.34
7	1	4	8	256	7.12	6.95	573.66	13.50
7	1	8	8	256	7.10	7.05	1131.10	13.82
7	1	16	8	256	7.94	7.15	2229.65	14.47
7	1	32	8	256	10.41	7.98	3992.18	15.77
7	1	64	8	256	16.36	8.55	7429.85	18.36
7	1	128	8	256	30.03	10.47	12095.66	23.57
7	1	256	8	256	58.69	15.20	16597.23	33.97
7	1	384	8	256	83.42	22.90	16536.86	44.36
7	1	512	8	256	110.82	25.45	19785.23	54.78
7	1	768	8	256	165.19	37.19	20298.52	75.57
7	1	1	1024	1024	30.86	8.64	115.40	13.86
7	1	2	1024	1024	58.33	8.75	227.09	14.56
7	1	4	1024	1024	109.13	8.88	445.36	15.93
7	1	8	1024	1024	213.48	9.23	848.04	18.70
7	1	16	1024	1024	415.95	10.06	1528.73	24.22
7	1	32	1024	1024	830.35	13.84	2184.17	35.27
7	1	64	1024	1024	1633.89	20.18	2939.07	57.37
7	1	80	1024	1024	2078.37	22.15	3309.25	68.42

Mem Offload

- 我们能否在执行期间进行 GPU – CPU 的显存切换？这是不可能的，因为通信带宽太低。并且切换操作将阻塞下一次 decoding 的到来。

GPU MEM：2,039GB/s

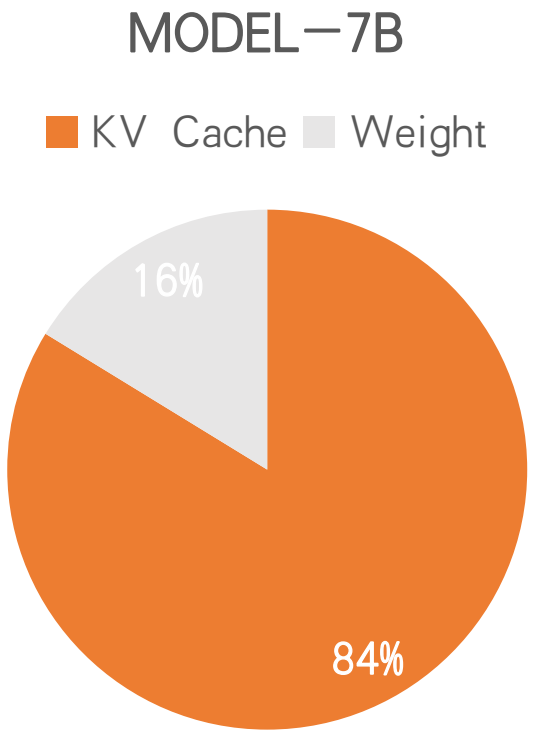
NVLink：600GB/s

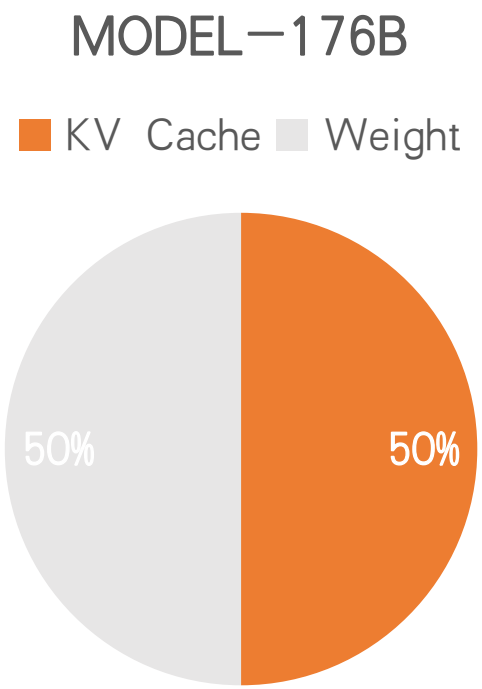
PCIe Gen4：64GB/s

切换 1G 的数据需要 ~ 20ms

优化：
KV Cache 量化

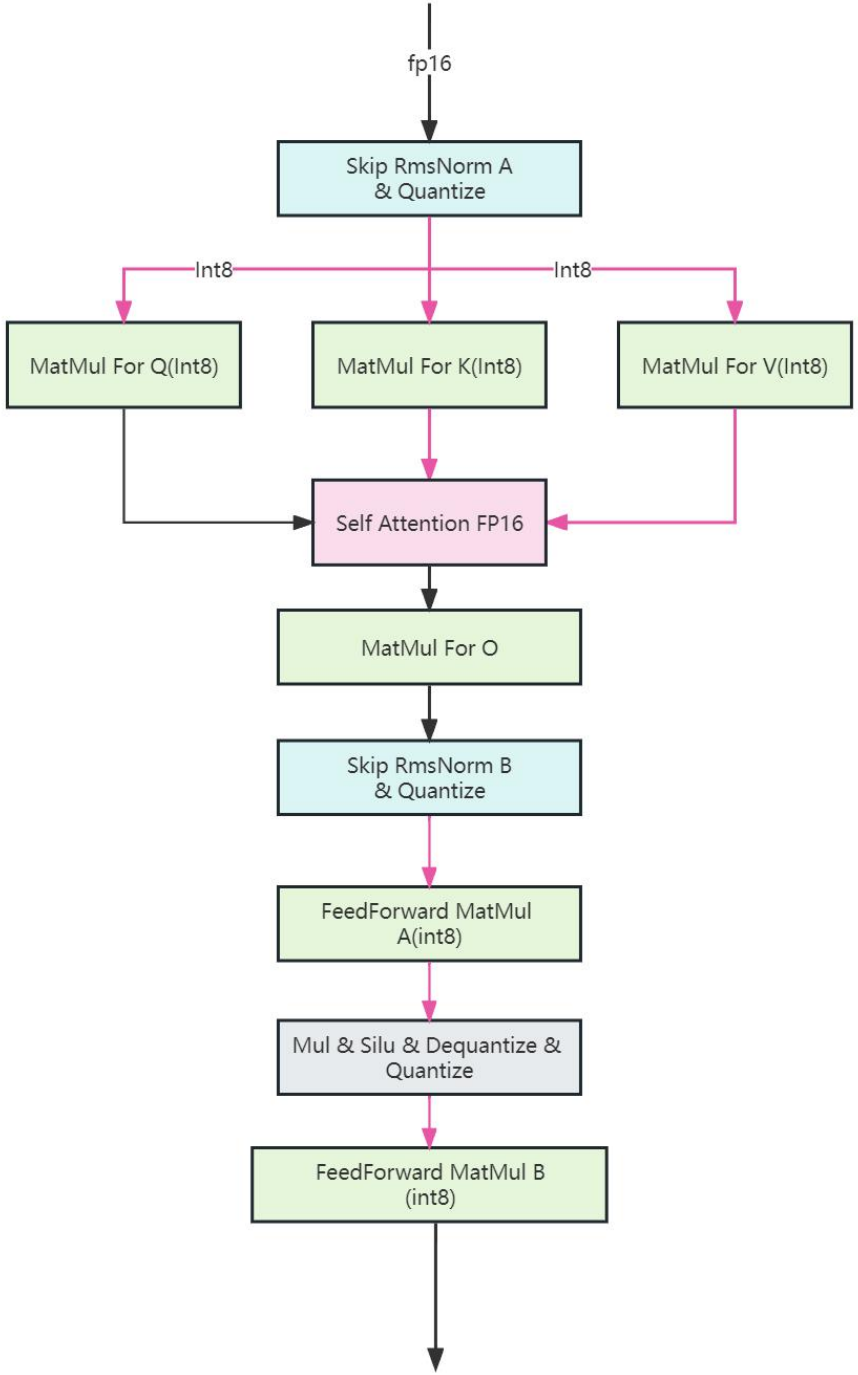
- 在服务端推理时，KV 缓存将占据大部分显存空间，这严重限制了系统的并发请求数量
- PPL.LLM 使用极高精度的分组量化方法对 KV 缓存数据进行压缩。
- 这使得服务端能够容纳的请求数量增加了 100%





优化：
KV Cache 量化

- 在服务端推理时，KV 缓存将占据大部分显存空间，这严重限制了系统的并发请求数量
- PPL.LLM 使用极高精度的分组量化方法对 KV 缓存数据进行压缩。
- 这使得服务端能够容纳的请求数量增加了 100%



优化：
KV Cache 量化

- 在服务端推理时，KV 缓存将占据大部分显存空间，这严重限制了系统的并发请求数量
- PPL.LLM 使用极高精度的分组量化方法对 KV 缓存数据进行压缩。
- 这使得服务端能够容纳的请求数量增加了 100%

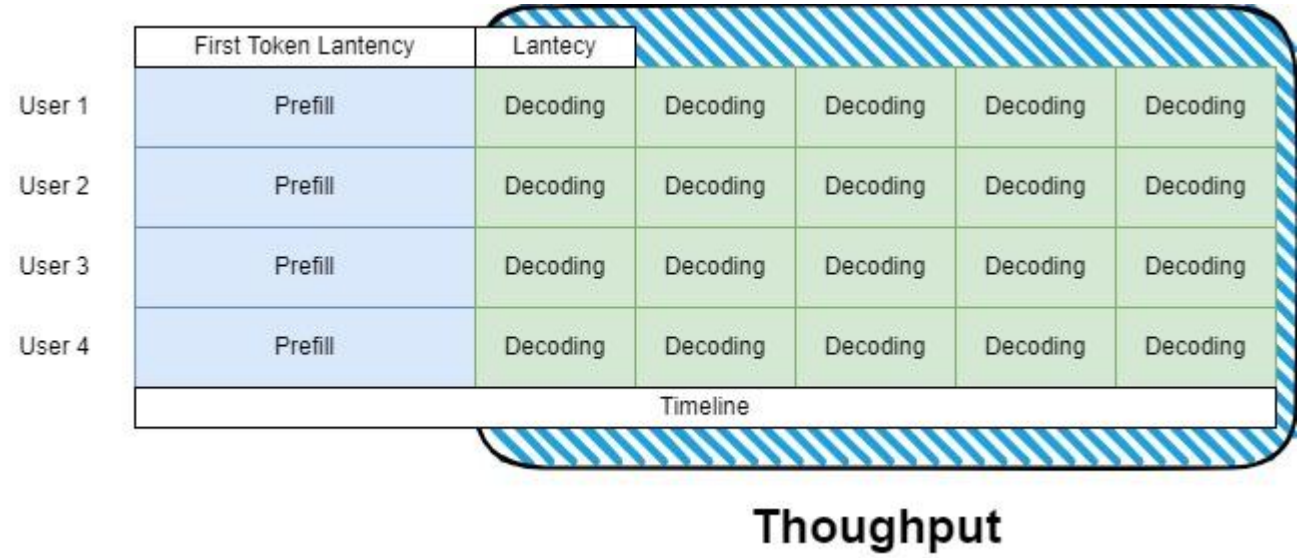
```
import torch

def group_quantize(x: torch.Tensor, group_size: int = 8):
    x = x.reshape((-1, group_size))
    scale = torch.max(x.abs(), dim=-1, keepdim=True)[0] / 127
    qt = torch.round(x / scale).char()
    return qt.flatten(), scale

def per_tensor_quantize(x: torch.Tensor):
    x = x.flatten()
    scale = torch.max(x.abs()) / 127
    qt = torch.round(x / scale).char()
    return qt.flatten(), scale
```

优化：
KV Cache 量化

- 在服务端推理时，KV 缓存将占据大部分显存空间，这严重限制了系统的并发请求数量
- PPL.LLM 使用极高精度的分组量化方法对 KV 缓存数据进行压缩。
- 这使得服务端能够容纳的请求数量增加了 100%



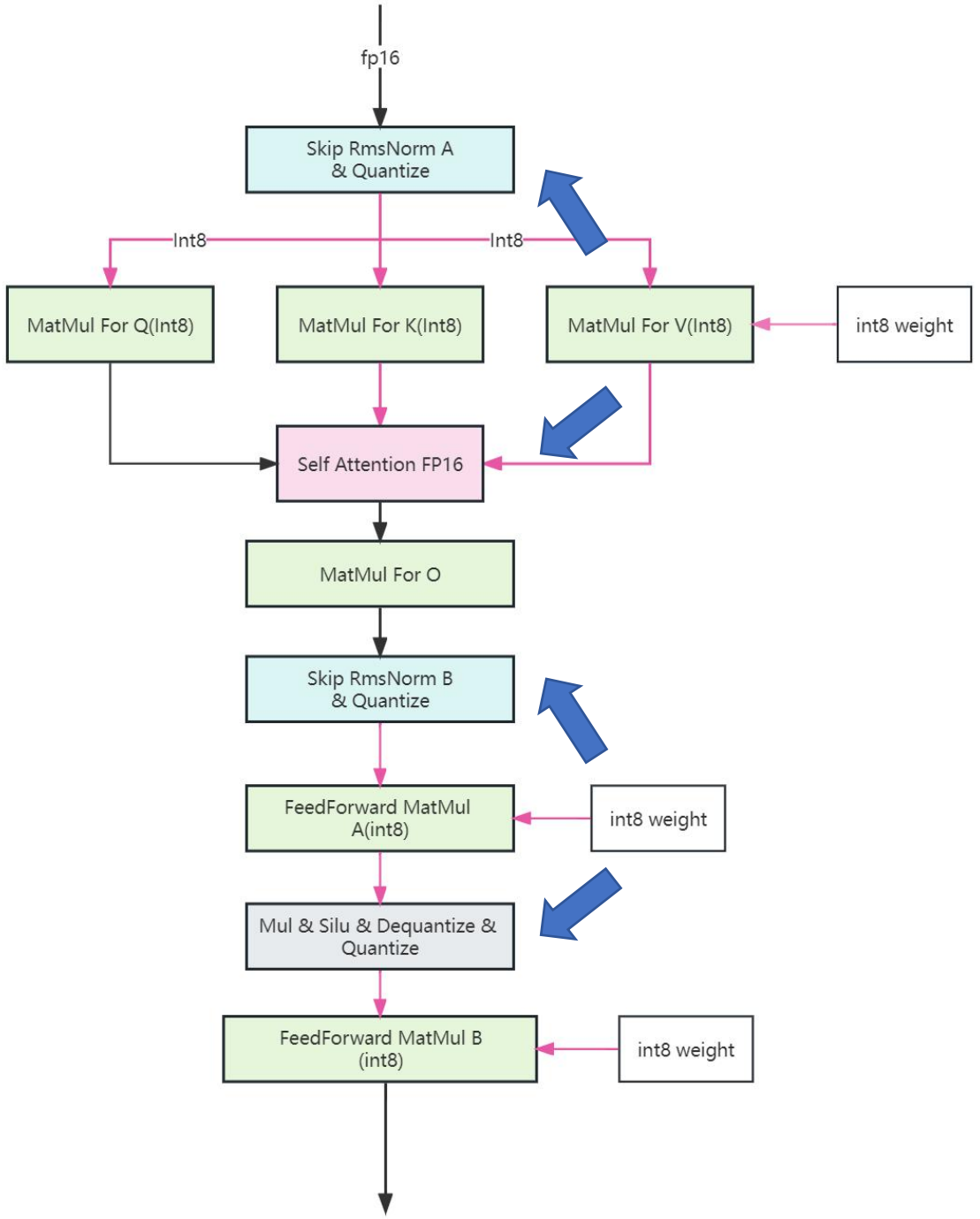
优化：
矩阵乘法量化

- 在服务端推理时，矩阵乘法所需的时间占整个推理的70%以上
- PPL.LLM 使用 dynamic channel-token int8 量化加速矩阵乘法。
- 这些量化同样精度极高，将提供~ 100% 的性能提升。

Operator	GPU TIME Percentage
Attention	7~30%
MatMul	70~90%
Normalization	3%
Other	<5%

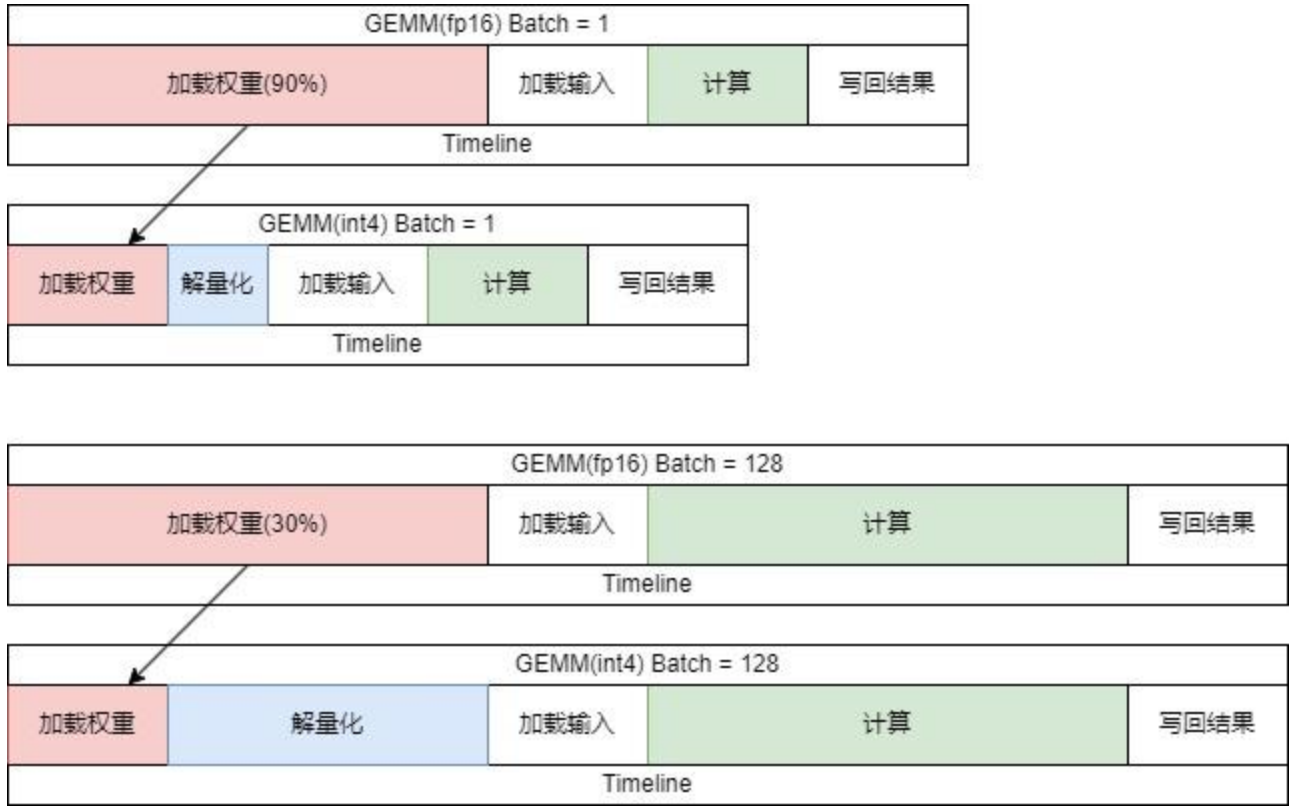
优化：
矩阵乘法量化

- 在服务端推理时，矩阵乘法所需的时间占整个推理的70%以上
- PPL.LLM 使用 dynamic channel-token int8 量化加速矩阵乘法(Calibration Free)。
- 这些量化同样精度极高，将提供~ 100% 的性能提升。



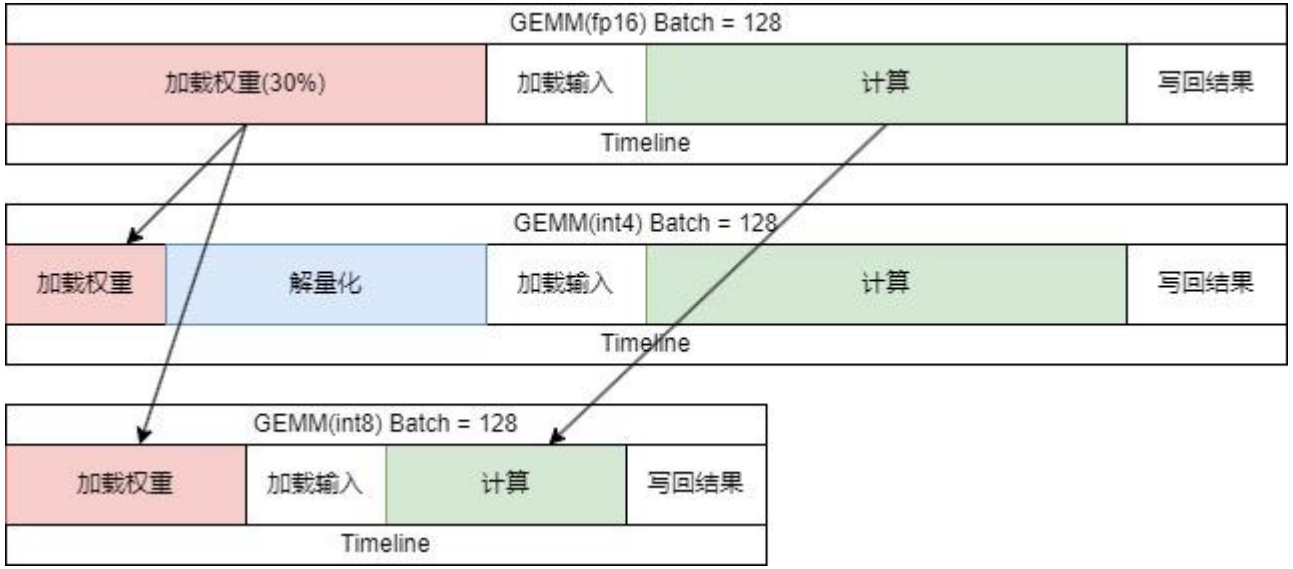
优化：
INT8 vs INT4

- INT4 Weight Only 量化适用于访存密集型的矩阵乘法。
- INT8 的量化适用于计算密集型的矩阵乘法。
- 在服务端场景下，大部分矩阵是计算密集型的，INT8 的效率高于 INT4。



优化：
INT8 vs INT4

- INT4 Weight Only 量化适用于访存密集型的矩阵乘法。
- INT8 的量化适用于计算密集型的矩阵乘法。
- 在服务端场景下，大部分矩阵是计算密集型的，INT8 的效率高于INT4。



https://github.com/openppl-public/ppl.llm.kernel.cuda/pull/6

优化：
INT8 vs INT4

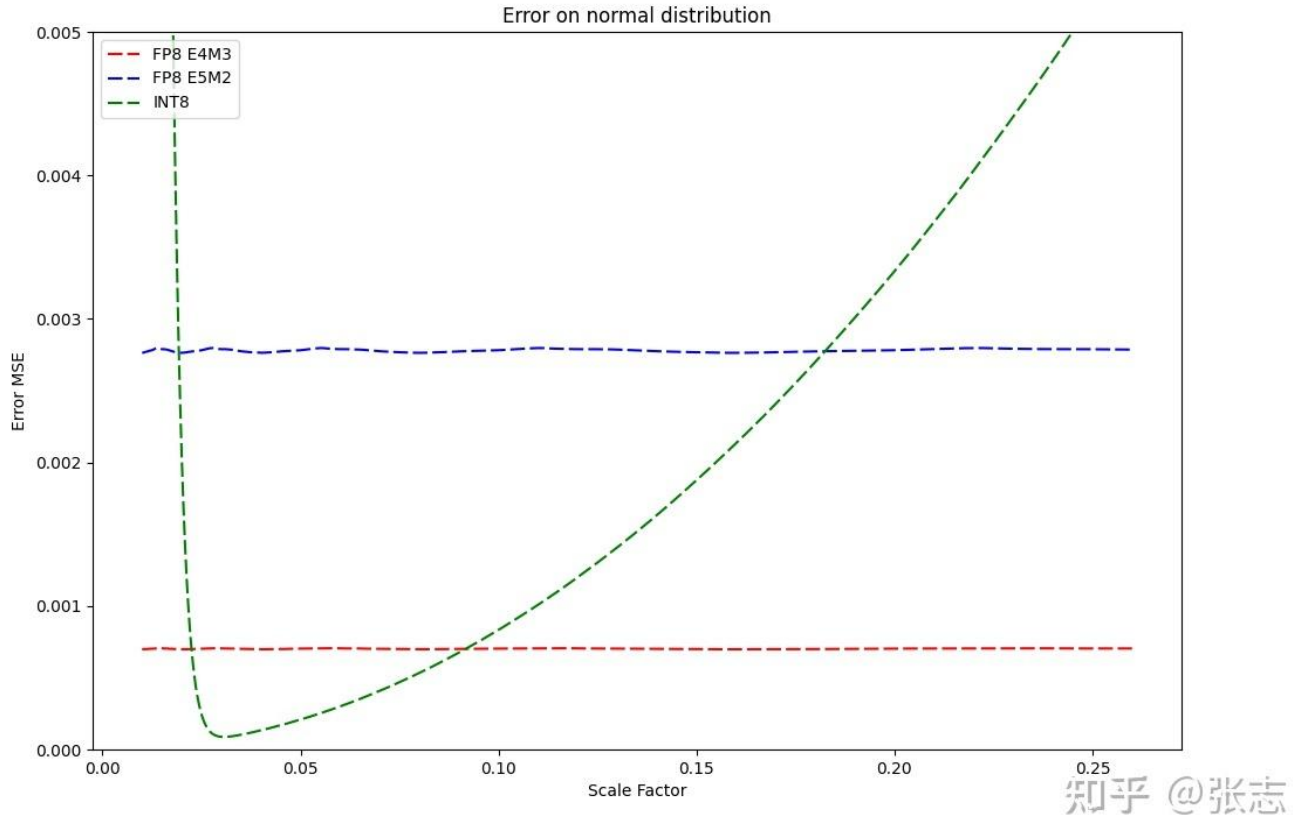
- INT4 Weight Only 量化适用于访存密集型的矩阵乘法。
- INT8 的量化适用于计算密集型的矩阵乘法。
- 在服务端场景下，大部分矩阵是计算密集型的，INT8 的效率高于 INT4。

```
def dequantize(...): pass

def GEMM(X, W, output):
    # X: [I, K] Matrix
    # W: [K, J] Matrix
    for k in range(K):
        for j in range(J):
            for i in range(I):
                output[i][j] += X[i][k] * dequantize(W[k][j])
```

优化：
FP8 vs INT8

- INT8 精度高于 FP8
- 但 FP8 更好用(H800)
- 在服务端应用中 FP8~INT8 > INT4



https://zhuanlan.zhihu.com/p/574825662

优化：
INT4 vs 非线性量化

- Weight Only 量化的解压缩过程可以定制，线性量化不是最优量化方案。

https://github.com/TimDettmers/bitsandbytes

```
__device__ half dhDequantizeNF4(unsigned char val)
{
  // the values for this tree was generated by test_normal_map_tree
  // in the file tests/test_functional.py
  if((val & 0b1000) == 8)
    if((val & 0b0100) == 4) // 1
      if((val & 0b0010) == 2) // 11
        if((val & 0b0001) == 1) // 111
          return 1.0f;
        else
          return 0.7229568362236023f;
      else
        if((val & 0b0001) == 1) // 110
          return 0.5626170039176941f;
        else
          return 0.44070982933044434f;
    else
      if((val & 0b0010) == 2) //10
        if((val & 0b0001) == 1) // 101
          return 0.33791524171829224f;
        else
          return 0.24611230194568634f;
      else
        if((val & 0b0001) == 1) // 100
          return 0.16093020141124725f;
        else
          return 0.07958029955625534f;

```

硬件

- 根据模型结构，我们可以预估所有算子的计算量与访存需求。因此我们可以预估模型在不同硬件上的推理性能（一般般准）。
- 我们提供一份神奇的表格用于帮助用户确定硬件选型。

型号	显存容量(GB)	显存带宽(GB/s)	算力(TFLOPS)	最多请求数(单卡)	最大吞吐量(Tokens/sec)	计算延迟(ms)	访存延迟(ms)	价格	性价比
Tesla V100-PCIe-16G	16	900	112	16	720.0	3.3	22.22	38000	0.39
Tesla V100-SXM2-32G	32	900	125	80	1800.0	14.6	44.44	47000	0.79
RTX 4090	24	1008	165.2	48	1612.8	6.6	29.76	17500	1.89
L40	48	864	181.05	144	2073.6	18.1	69.44	58000	0.73
L4	24	300	121	48	480.0	9.0	100.00	19500	0.50
A100-80G-SXM4	80	2039	312	272	5546.1	19.9	49.04		
A800-80G-SXM4	80	2039	312	272	5546.1	19.9	49.04	103000	1.10
A100-80G-PCIe	80	1935	312	272	5263.2	19.9	51.68	140000	0.77
A800-80G-PCIe	80	1935	312	272	5263.2	19.9	51.68	92000	1.17
A30	24	933	165	48	1492.8	6.6	32.15	28500	1.07
A40	48	696	149.7	144	1670.4	21.9	86.21	29500	1.16
A10	24	600.2	125	48	960.3	8.8	49.98	16000	1.23
A2	16	200	18	16	160.0	20.3	100.00	9500	0.35

硬件

- 根据模型结构，我们可以预估所有算子的计算量与访存需求。因此我们可以预估模型在不同硬件上的推理性能（一般般准）。
- 我们提供一份神奇的表格用于帮助用户确定硬件选型。

资料链接：

https://pan.baidu.com/s/1hKIrHYe0BviQUopY3hUJ1g?pwd=3mv8

提取码：3mv8

PPL.LLM

项目连接：

https://github.com/openppl-public/ppl.llm.serving

https://github.com/openppl-public/ppl.nn.llm

https://github.com/openppl-public/ppl.llm.kernel.cuda

https://github.com/openppl-public/ppl.pmx