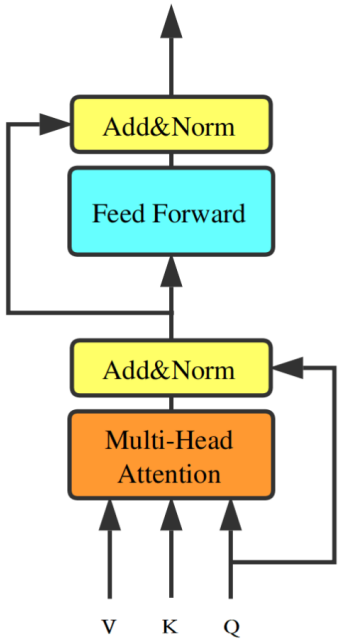
Keyboard LM Final

Introduction

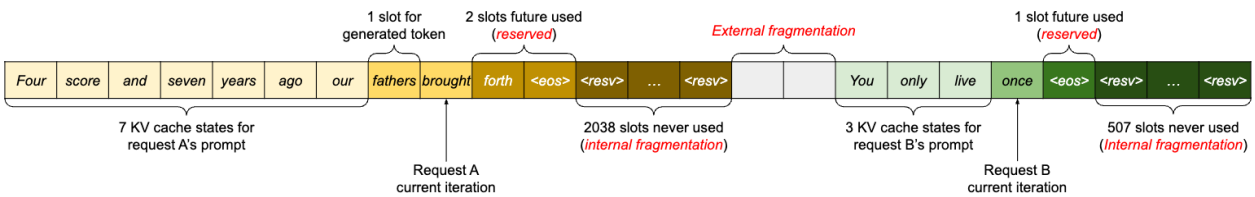
Transformer Architecture



当前主流的语言模型以Transformer decoder only为基础。如上所示就是一个典型的Transformer Decoder Block。一般来讲，transformer模型推理分为两阶段进行：prefill和decode。其中注意力机制（MHA）以 `O(n^2)` 复杂度进行Prefill，并以 `O(n)` 复杂度进行Decode。本工作主要聚焦于Decode阶段的优化。

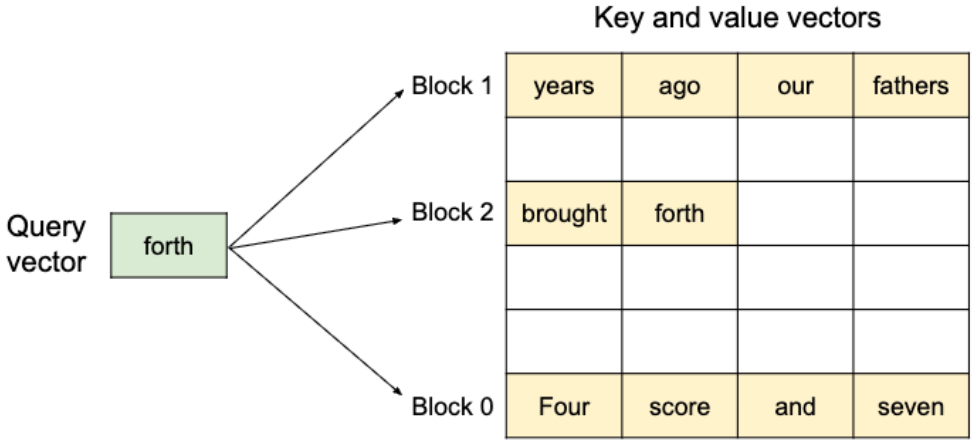
PagedAttention

朴素实现的KV cache管理：



以朴素方式管理KV Cache时，内存存在大量碎片，包括内部碎片和外部碎片等。

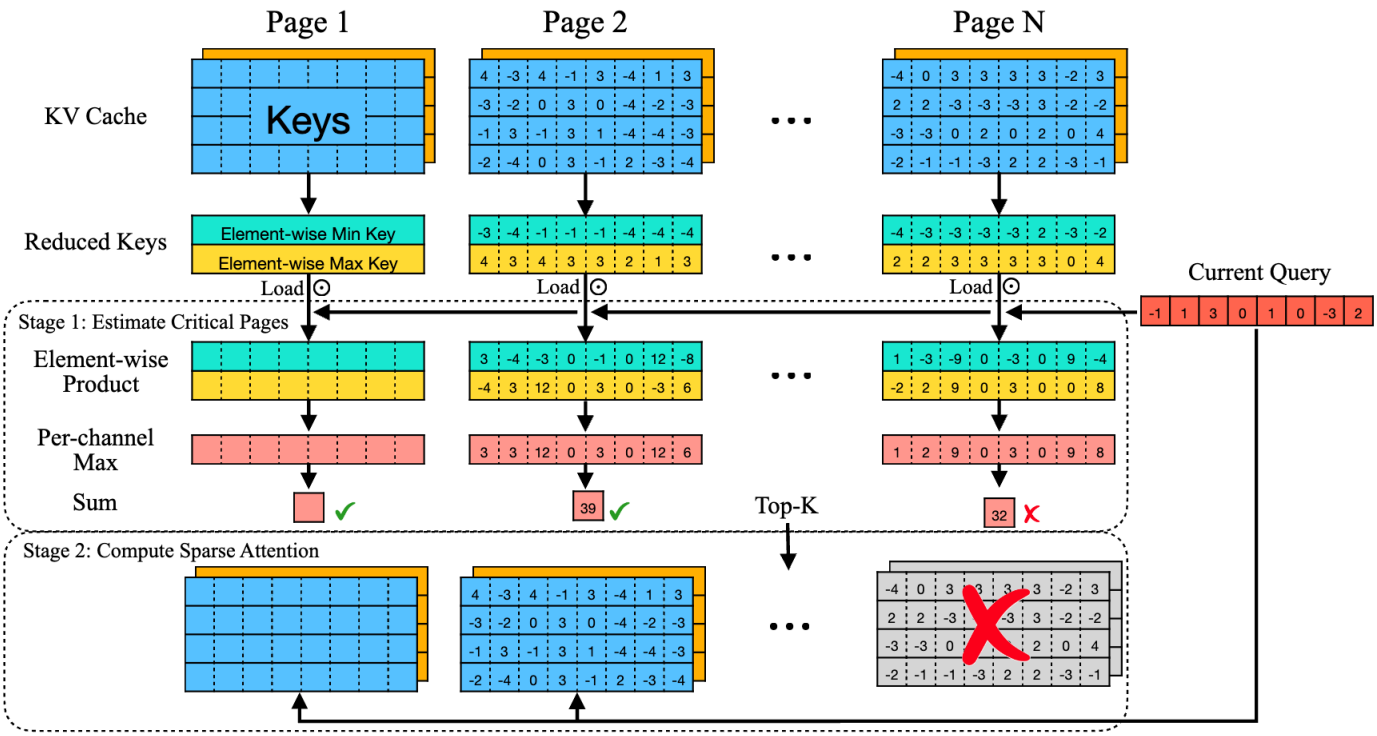
PagedAttention的KV cache管理风格：



由于端侧内存受限，内存管理在端侧设备上尤为重要。

Dynamic KV Cache

Quest算法：

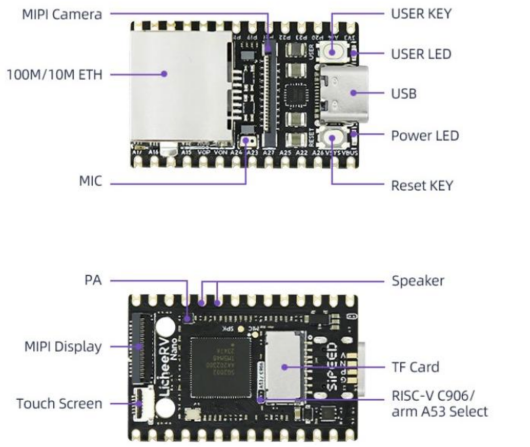


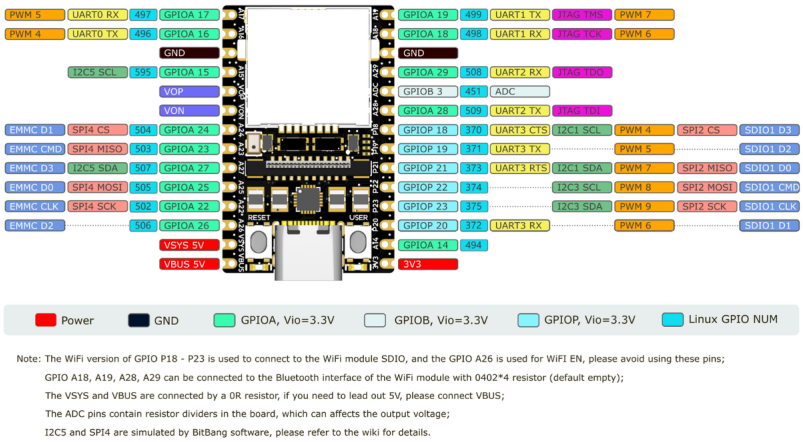
原论文实现主要局限性：

- 面向服务器，使用PyTorch管理内存，无法应用于端侧
- Kv Page依照Token进行保存，然而selection是基于Head进行，当完成选择以后需要加载全页而不能只加载对应attention head的kv
- 核心算子使用CUDA实现，不可迁移到端侧
- 没有对GQA（Group Query Attention）良好的支持

Background

本项目基于LicheeRVNano开发版开发：





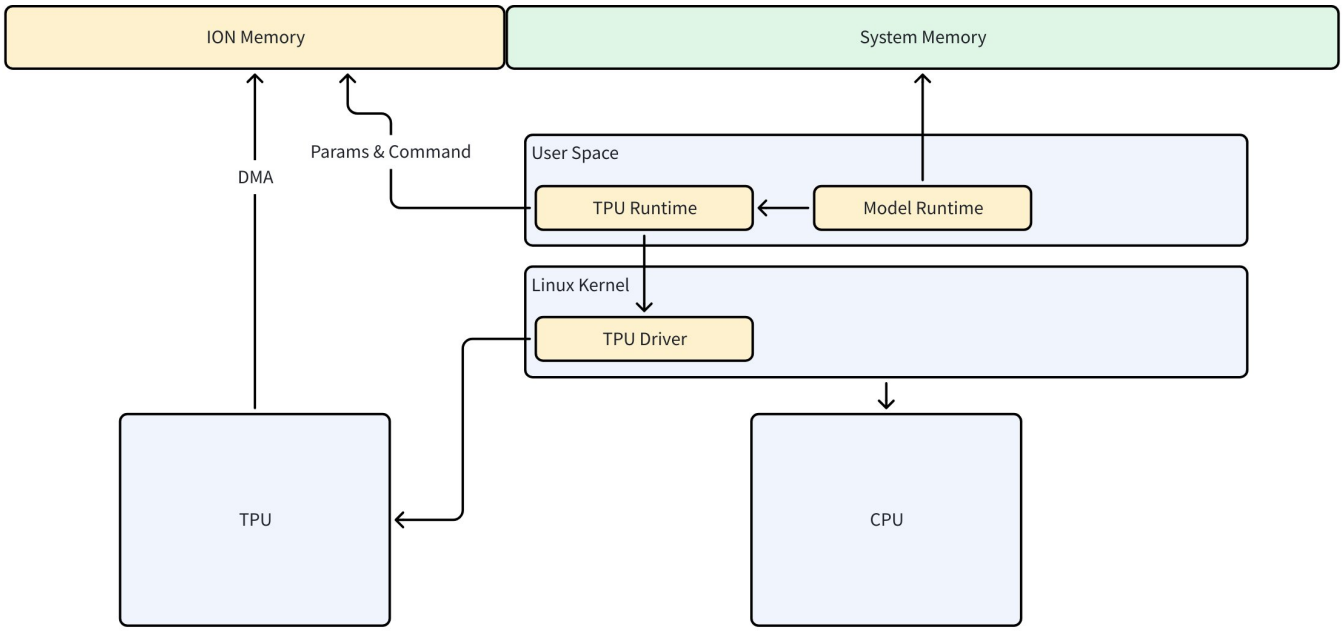
TPU开发版参数：

CPU	算能 SG2002； 大核：1GHz RISC-V C906 / ARM A53 二选一； 小核：700MHz RISC-V C906； 低功耗核：25～300M 8051
NPU	1TOPS INT8，支持 BF16
内存	内封 2Gbit (256MByte) DDR3
存储	TF卡 / SD NAND 二选一启动 （SD NAND 焊盘在 TF 卡槽下）
视频接口	视频输出：2 lane MIPI DSI 输出，标准 31pin 接口，支持 6pin 电容触摸屏 视频输入： 4 lane MIPI CSI 输入，22Pin 接口，支持拆分双路 CSI
音频接口	音频输出： 板载PA功放，可在排针上直接连接1W以内的喇叭 音频输入： 板载模拟硅麦，可直接收音
有线连接	E 后缀版本支持百兆 RJ45 连接器
无线连接	W 后缀版本支持 2.4G / 5.8G 双频 WiFi6 + BLE5.4
USB	1 x USB2.0 OTG Type-C
IO接口	2 x 14pin 2.54 插针接口，间距 800mil，可直插面包板
按键	1 x RST 按键 + 1 x BOOT 按键
指示灯	1 x 电源 LED，1 x 用户 LED
操作系统	Buildroot Linux / Debian

尺寸	22.86*35.56mm

Whole Architecture

TPU Driver & Runtime



代码块

```
**主要功能：**
- 支持 CVITEK TPU 硬件（如 cv181x, cv182x, cv183x 等）。
- 提供共享内存和设备内存管理。
- 支持 TPU 上的命令执行和性能监控。
- 包含 Python 绑定以便快速开发和验证。
- 提供多个 SoC（System on Chip）平台的支持。

---

#### **代码结构**

1. **`src/` 目录**
   - **`common/`**
     - 包含共享逻辑，如内存分配、程序加载和运行时上下文管理。
     - 关键文件包括：
       - `program.cpp`：负责 TPU 程序的加载和执行。
       - `shared_mem.cpp`：管理共享内存的分配和回收。
       - `neuron.cpp`：管理神经元相关的内存操作。
```

```
   - **`soc/`**
     - 包含针对不同 SoC（如 cv181x, cv182x）的具体实现。
     - 关键文件包括：
       - `tpu_pmu.cpp`: 用于 TPU 的电源管理和性能监控。
       - `cvi_device_mem.cpp`: 使用 ION 子系统管理设备内存。

2. **`doc/` 目录**
   - 包含开发手册和文档，比如 `cvitek_tpu_sdk_development_manual.md`，详细介绍了 TPU Runtime 的使用方法和 API。

3. **`samples/` 目录**
   - 提供了多个示例代码，用于展示如何使用 Runtime 开发 TPU 应用。
   - 示例包括分类器、BF16 模型等。

4. **`python/` 目录**
   - 提供 Python 绑定，使开发者可以快速使用 Runtime 的功能。

5. **`README.md` 文件**
   - 提供了项目的总体概述、依赖项和构建说明。

---

#### **如何向 TPU 发送命令（以 cv181x 为例）**

1. **准备命令缓冲区**
   TPU 的命令通常存储在缓冲区（`cmdbuf`）中。可以通过以下函数执行：
   ```cpp
   CVI_RT_RunCmdbufEx(_ctx, buf_mem, baseArray);
   ```
   此函数支持加密和非加密模式。

2. **数据传输到 TPU**
   数据通过以下步骤传输到 TPU：
   - 使用 `Neuron::toTpu()` 方法：
     ```cpp
     CVI_RT_MemFlush(_ctx, _gmem);
     CVI_RT_MemInvld(_ctx, _base_mem);
     ```
   - 此方法会确保数据从主机内存（Host Memory）刷新并加载到 TPU 内存。

3. **性能监控**
   在 `tpu_pmu.cpp` 文件中，TPU 的性能指标（如时钟速率、执行时间）通过以下逻辑计算：
   ```cpp
   percent_tdma = (double)u64TDMATotal / (double)bmnet_p_duration * 100;
   percent_tiu = (double)u64TIUTotal / (double)bmnet_p_duration * 100;
   ```
```

```
---

#### **内存类型及其关系**

1. **ION 内存**
   - 使用 Linux 的 ION 子系统进行内存分配。
   - 在文件 `cvi_device_mem.cpp` 中通过以下代码分配内存：
     ```cpp
     ret = ioctl(fd, ION_IOC_ALLOC, &alloc_data);
     ```

2. **共享内存**
   - 在 `shared_mem.cpp` 中，使用全局列表（`gSharedMemList`）管理共享内存块：
     ```cpp
     CVI_RT_MEM allocateSharedMemory(CVI_RT_HANDLE ctx, size_t size);
     deallocateSharedMemory(ctx, mem);
     ```

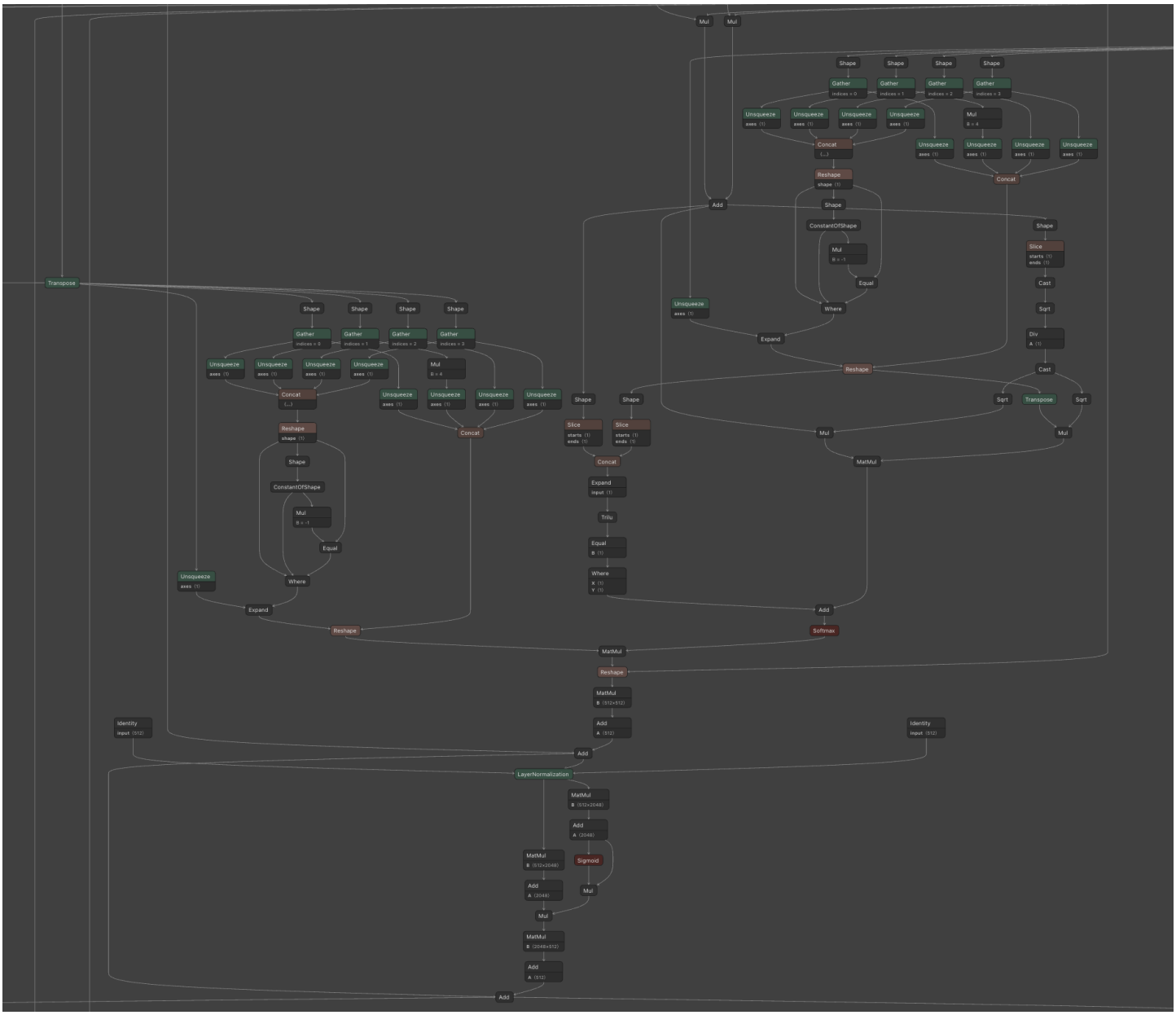
3. **主机内存**
   - 主机内存用于 CPU 端的操作，并在需要时映射到设备内存（TPU 内存）。

4. **关系总结**
   - **ION 内存** 作为底层分配器，为设备和共享内存提供物理内存。
```

Model Runtime

Model Implementation

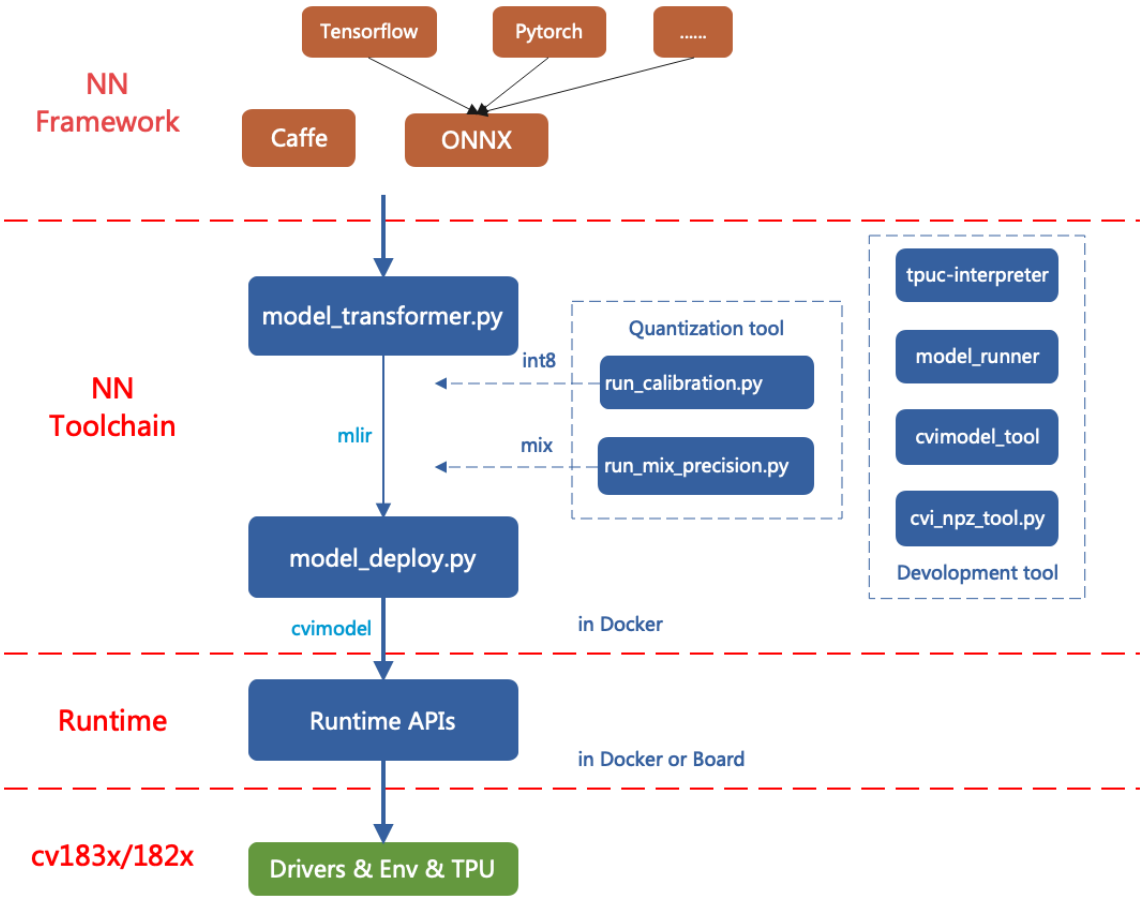
我首先基于PyTorch实现了python版的模型训练与推理代码，适应于模型训练和初步验证。



如上图所示是Prefill计算图节选。

Implementing Model on Edge Device

Model Compiling



模型在完成预训练后，需要经过编译，转换为静态模型。通过将transformer的静态部分与动态部分分离，并且拆成算子模块，实现模型编译，并且适应动态KV Cache管理。

第二步通过模型融合，实现不同计算图的参数共享，节约现存。

Transformer & PagedAttention in C++

基于端侧的具体硬件，我基于TPU SDK在端侧重新实现了Transformer推理+PagedAttention管理的代码，用于模型实际部署推理并管理kv cache。

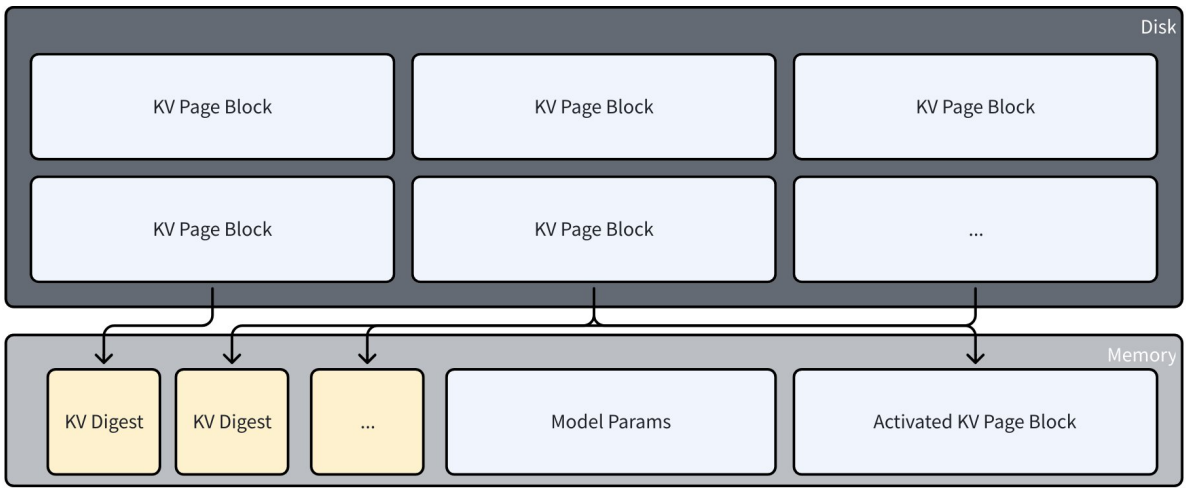
经过设计，Transformer总参数大约25M。关键参数如下：

代码块

```
@dataclass
class KLMConfig:
    max_length: int = 16384
    effective_length: int = 512
    vocab_size: int = 512
    hidden_size: int = 512
    tie_word_embeddings: bool = True
    num_layers: int = 6
    num_attention_heads: int = 8
    num_kv_heads: int = 2
    ffn_hidden_size: int = 2048
```

KV Cache Management

基于Quest本身的实现上的不足，我在端侧上重新对KV Cache进行了管理。使用双层设计，KV Cache的摘要固定在内存中，KV Cache本身按照需要offload到硬盘并动态加载。



按照目前的参数设计，Model Params本身如果以BF16精度存储，则需要占用50M内存。端侧设备目前总共有256M内存，系统本身和程序运行时共计占大约25M。因此有100M用于保存KV Digest和激活的KV Page。每个KV Page Block对应的KV Digest大小为256B。 每个KV Page Block对应的KV Digest大小为6KB。100M内存总计可保存大约16K个KV Digest，如果Block size以32计，则总context window的理论上限是512K Token。

结构体参考：

代码块

```
struct KVPage {
    dtype k[BLOCK_LEN][HEAD_DIM];
    dtype v[BLOCK_LEN][HEAD_DIM];
};

struct KVPageDigest {
    dtype k_max[HEAD_DIM];
    dtype k_min[HEAD_DIM];
};

struct KVPool {
    KVPage *pages[CAPACITY][NUM_LAYERS][NUM_HEADS];
    KVPageDigest *page_digests[CAPACITY][NUM_LAYERS][NUM_HEADS];
};
```

User Interface

键盘与TPU芯片通过串口交流，TPU芯片上会serve一个小型的stdio的接口，键盘通过预定义的沟通协议与TPU芯片通信，实时将键盘输入推流到TPU做encode。在需要的时刻，键盘通过协议发出预测指令，TPU芯片上的模型开始解码，并且将预测结果返回给键盘，键盘将输出内容显示到屏幕上。

Model Architecture

Model Design

模型基本跟随了标准的transformer decoder实现，并且采用了目前一些流行的设计，包括MQA，RotaryEmbedding等。

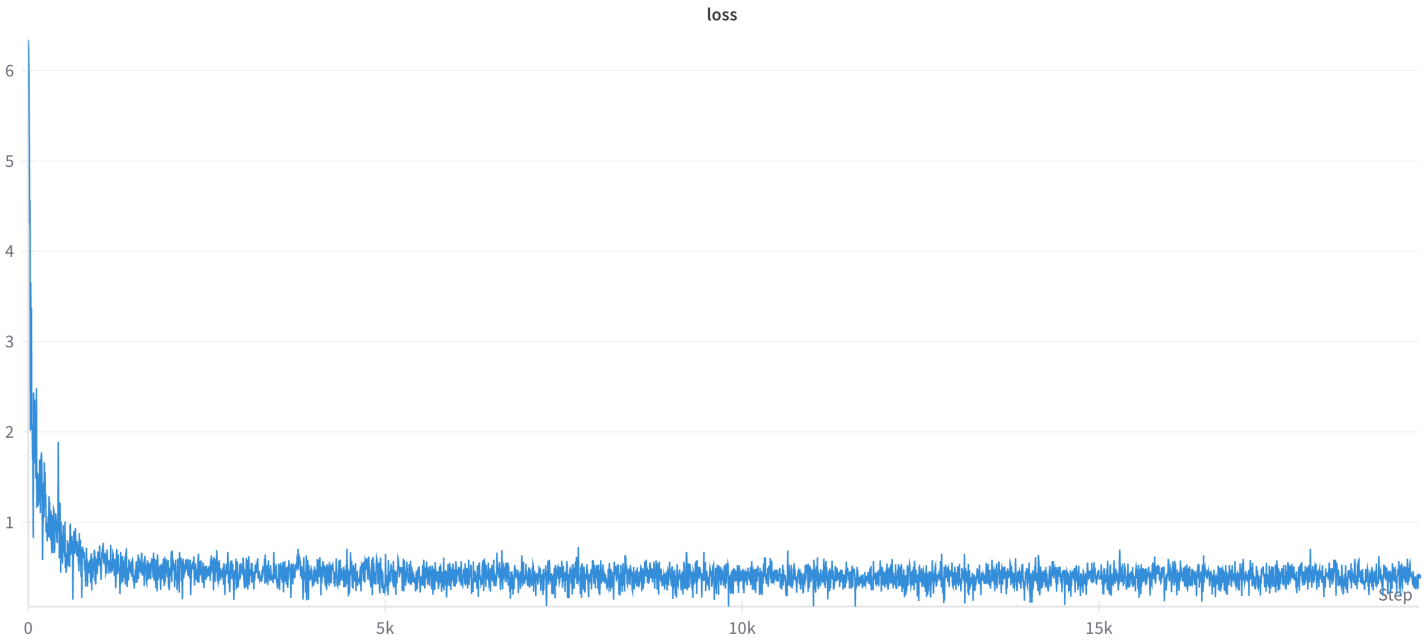
Tokenizing

为了适配键盘的交互设计，模型采用了char级别的Tokenizer，即直接将单个ASCII字符encode成单个Token，适应了键盘上的输入和推理。这样的好处：

- 输入时可以做实时encode，按键与Token之间存在双射，降低推理延迟
- 输出时与键盘的动作空间对齐，可以直接将模型预测结果推流到键盘输出

Training

基于如上的设计，我在模型上进行了40B Token的训练。训练Context Length为65536，训练曲线如下：



Future Plans

Uart delay

现在串口延时较高，会带来一些通信上的延迟，造成整体体验下降，后续方向包括优化串口速度。

LayerNorm Precision

目前TPU不支持混合精度推理。现行LLM推理通常以BF16推理大部分模型部分，而LayerNorm部分以FP32精度进行推理。全部以BF16精度进行推理可能带来精度上的挑战。

Quantization

后续根据实际结果和安排，实现INT8量化推理。

Better Algorithm and Performance

根据实际情况进一步调优性能。