Llama.cpp

简单解释一下流程llama.cpp/build/bin/main

- 在main.cpp中会解析命令行的参数，如所用的模型文件，prompt信息等，之后进行一系列操作后，并调用了一次llama_decode()函数来对模型进行了一次warm up, 之后进入一个while循环进行模型的推理，期间会多次调用llama_decode(ctx, batch)函数进行推理，直到不满足while条件。
- 在 llama_decode()中调用了llama_build_graph(), 整个模型结构的推理计算图构建全在该函数内实现
- 下面以llama.cpp/build/bin/simple为例，主要流程都一样，主要是没有太多复杂的参数配置

- cmd参数解析：如gguf模型路径，输入text，采样参数配置topk等
- llama_backend_init(); 后端初始化，如cuda后端，资源初始化

```
void ggml_backend_load_all() {
    ggml_backend_load_all_from_path(nullptr);
}

void ggml_backend_load_all_from_path(const char * dir_path) {
#ifdef NDEBUG
    bool silent = true;
#else
    bool silent = false;
#endif

    ggml_backend_load_best("blas", silent, dir_path);
    ggml_backend_load_best("cann", silent, dir_path);
    ggml_backend_load_best("cuda", silent, dir_path);
    ggml_backend_load_best("hip", silent, dir_path);
    ggml_backend_load_best("kompute", silent, dir_path);
    ggml_backend_load_best("metal", silent, dir_path);
    ggml_backend_load_best("rpc", silent, dir_path);
    ggml_backend_load_best("sycl", silent, dir_path);
    ggml_backend_load_best("vulkan", silent, dir_path);
    ggml_backend_load_best("opencl", silent, dir_path);
    ggml_backend_load_best("musa", silent, dir_path);
    ggml_backend_load_best("cpu", silent, dir_path);
    // check the environment variable GGML_BACKEND_PATH to load an out-of-tree backend
    const char * backend_path = std::getenv("GGML_BACKEND_PATH");
    if (backend_path) {
        ggml_backend_load(backend_path);
    }
}
```

➤ 模型初始化

```
// initialize the model

llama_model_params model_params = llama_model_default_params();
model_params.n_gpu_layers = ngl;

llama_model * model = llama_model_load_from_file(model_path.c_str(), model_params);
const llama_vocab * vocab = llama_model_get_vocab(model);
```

```
∨ model_params = {...}
  > devices = 0x0
    n_gpu_layers = 99
    split_mode = LLAMA_SPLIT_MODE_LAYER
    main_gpu = 0
  > tensor_split = 0x0
    progress_callback = 0x0
    progress_callback_user_data = 0x0
  > kv_overrides = 0x0
    vocab_only = false
    use_mmap = true
    use_mlock = false
    check_tensors = false
```

```
∨ model = 0x55b837e8a0b0
    type = LLM_TYPE_UNKNOWN
    arch = LLM_ARCH_LLAMA
  > name = "Llama 68m"
  > hparams
  > vocab
  > tok_embd = 0x55b838e08b60
  > type_embd = 0x0
  > pos_embd = 0x0
  > tok_norm = 0x0
  > tok_norm_b = 0x0
  > output_norm = 0x55b838e0b120
```

```
struct llama_model {
    llm_type type = LLM_TYPE_UNKNOWN;
    llm_arch arch = LLM_ARCH_UNKNOWN;

    std::string name = "n/a";

    llama_hparams hparams = {};
    llama_vocab   vocab;

    struct ggml_tensor * tok_embd   = nullptr;
    struct ggml_tensor * type_embd  = nullptr;
    struct ggml_tensor * pos_embd   = nullptr;
    struct ggml_tensor * tok_norm   = nullptr;
    struct ggml_tensor * tok_norm_b = nullptr;

    struct ggml_tensor * output_norm     = nullptr;
    struct ggml_tensor * output_norm_b   = nullptr;
    struct ggml_tensor * output          = nullptr;
    struct ggml_tensor * output_b        = nullptr;
    struct ggml_tensor * output_norm_enc = nullptr;

    // classifier
    struct ggml_tensor * cls       = nullptr;
    struct ggml_tensor * cls_b     = nullptr;
    struct ggml_tensor * cls_out   = nullptr;
    struct ggml_tensor * cls_out_b = nullptr;

    struct ggml_tensor * conv1d   = nullptr;
    struct ggml_tensor * conv1d_b = nullptr;

    std::vector<llama_layer> layers;

    llama_model_params params;

    // gguf metadata
    std::unordered_map<std::string, std::string> gguf_kv;

    // list of devices used in this model
    std::vector<ggml_backend_dev_t> devices;
```

- ➢ 具体模型初始化流程
  - ➢ 配置相关后端

```cpp
// use all available devices
for (size_t i = 0; i < ggml_backend_dev_count(); ++i) {
    ggml_backend_dev_t dev = ggml_backend_dev_get(i);
    switch (ggml_backend_dev_type(dev)) {
        case GGML_BACKEND_DEVICE_TYPE_CPU:
        case GGML_BACKEND_DEVICE_TYPE_ACCEL:
            // skip CPU backends since they are handled separately
            break;

        case GGML_BACKEND_DEVICE_TYPE_GPU:
            ggml_backend_reg_t reg = ggml_backend_dev_backend_reg(dev);
            if (ggml_backend_reg_name(reg) == std::string("RPC")) {
                rpc_servers.push_back(dev);
            } else {
                model->devices.push_back(dev);
            }
            break;
    }
}
```

- ➢ 加载其他信息（从gguf文件里加载）

```cpp
llama_model_loader ml(fname, splits, params.use_mmap, params.check_tensors, params.kv_overrides);

ml.print_info();

model.hparams.vocab_only = params.vocab_only;

try {
    model.load_arch(ml);
} catch(const std::exception & e) {
    throw std::runtime_error("error loading model architecture: " + std::string(e.what()));
}
try {
    model.load_hparams(ml);
} catch(const std::exception & e) {
    throw std::runtime_error("error loading model hyperparameters: " + std::string(e.what()));
}
try {
    model.load_vocab(ml);
} catch(const std::exception & e) {
    throw std::runtime_error("error loading model vocabulary: " + std::string(e.what()));
}
```

```cpp
enum llm_arch {
    LLM_ARCH_LLAMA,
    LLM_ARCH_DECI,
    LLM_ARCH_FALCON,
    LLM_ARCH_BAICHUAN,
    LLM_ARCH_GROK,
    LLM_ARCH_GPT2,
    LLM_ARCH_GPTJ,
    LLM_ARCH_GPTNEOX,
    LLM_ARCH_MPT,
    LLM_ARCH_STARCODER,
    LLM_ARCH_REFACT,
    LLM_ARCH_BERT,
    LLM_ARCH_NOMIC_BERT,
    LLM_ARCH_JINA_BERT_V2,
    LLM_ARCH_BLOOM,
    LLM_ARCH_STABLELM,
    LLM_ARCH_QWEN,
    LLM_ARCH_QWEN2,
    LLM_ARCH_QWEN2MOE,
    LLM_ARCH_QWEN2VL,
    LLM_ARCH_PHI2,
    LLM_ARCH_PHI3,
    LLM_ARCH_PHIMOE,
    LLM_ARCH_PLAMO,
    LLM_ARCH_CODESHELL,
    LLM_ARCH_ORION,
    LLM_ARCH_INTERNLM2,
    LLM_ARCH_MINICPM,
    LLM_ARCH_MINICPM3,
    LLM_ARCH_GEMMA,
    LLM_ARCH_GEMMA2,
    LLM_ARCH_STARCODER2,
    LLM_ARCH_MAMBA,
    LLM_ARCH_XVERSE,
```

- llm_load_arch(ml, model); --- 从gguf中读取模型架构，如 qwen
- llm_load_hparams(ml, model); 从gguf中读取从config.json文件保存的模型配置参数 <-- 区分模型，如qwen
- llm_load_vocab(ml, model); 从gguf中读取词表 <-- 区分模型，如qwen
- llm_load_tensors；从gguf中读取模型参数，cpu/cuda等区分，分别进行内存分配与模型参数加载，由 llama_model 管理参数内存

- Tokenize

```
// find the number of tokens in the prompt
const int n_prompt = -llama_tokenize(vocab, prompt.c_str(), prompt.size(), NULL, 0, true, true);

// allocate space for the tokens and tokenize the prompt
std::vector<llama_token> prompt_tokens(n_prompt);
```

- 创建推理上下文，把所有推理相关的上下文聚集在一起，将model绑定到context上，一个model对应一个ctx，kv_cache由ctx管理。

```
// initialize the context

llama_context_params ctx_params = llama_context_default_params();
// n_ctx is the context size
ctx_params.n_ctx = n_prompt + n_predict - 1;
// n_batch is the maximum number of tokens that can be processed in a single call to
ctx_params.n_batch = n_prompt;
// enable performance counters
ctx_params.no_perf = false;

llama_context * ctx = llama_init_from_model(model, ctx_params);
```

推理前确定token数量大小

```
struct llama_context {
    llama_context(const llama_model & model)
        : model(model)
        , t_start_us(model.t_start_us)
        , t_load_us(model.t_load_us) {}

    const struct llama_model & model;

    struct llama_cparams      cparams;
    struct llama_sbatch       sbatch;  // TODO: revisit if needed
    struct llama_kv_cache     kv_self;
    struct llama_adapter_cvec cvec;

    std::unordered_map<struct llama_adapter_lora *, float> lora;

    std::vector<ggml_backend_ptr> backends;
    std::vector<std::pair<ggml_backend_t, ggml_backend_set_n_threads_t>> set_n_threads_fns;

    ggml_backend_t backend_cpu = nullptr;

    ggml_threadpool_t threadpool       = nullptr;
    ggml_threadpool_t threadpool_batch = nullptr;

    bool has_evaluated_once = false;
```

- 根据后端确定缓冲区类型,并计算缓冲区大小
- 判断是否启用流水线并行

在正式计算前，我们需要将后续计算所需要的所有tensor、graph都推算一遍，然后init初始化所需要的空白buffer，以供后续计算时使用。

❖ Context初始化详解（接上一页PPT）

在这一阶段，重点包含以下几个buffer的init

- kv cache buffer init： 需要提前根据超参数hparams确认kv cache最大使用情况，然后分配对应buffer
- graph output buffer: 运行llm时，除了模型权重、运行时产生的kv cache之外，还有输出的中间结果结果也是不确定的，需要根据需求确认最后模型输出可能使用的最大buffer。
- sheduler init: 由于调度器本身可能在多个后端都保存有副本，所以这里需要确认调度器的参数、配置如PP
- cgrpah init： 这里需要注意，想要确认计算图的内存空间大小，就必须先构建计算图。所以在这里作者使用了llama_build_graph() 构建了计算图。
  - 对于一个llm模型来说，其Attention算子内部、FFN等计算的逻辑关系就是在这个函数中进行构建的。所以想要了解llama.cpp中如何实现例如Moe、MLA等方法是如何实现的，仅需查看llama_build_graph()。

需要注意虽然在ctx init这一步骤中，使用了llama_build_graph()函数对目标架构的llm内部算子关系进行了计算图的构建，但构建的graph是用来评估计算是最坏情况使用的内存情况的，并不会在后续计算中使用该graph。所以在后续实际计算中，会看到使用了一模一样的“llama_build_graph()”。

- 每次decode都会调用llama_build_graph并计算 graph

- 推理循环
  - kv-cache的管理， 通过context来管理
  - llama_decode ---- 进行推理 <------
    llama_batch_get_one --- 获取一个输入

```
ggml_cgraph * gf = llama_build_graph(lctx, ubatch, false);

// the output is always the last tensor in the graph
struct ggml_tensor * res  = ggml_graph_node(gf, -1);
struct ggml_tensor * embd = ggml_graph_node(gf, -2);
const auto compute_status = llama_graph_compute(lctx, gf, n_threads, threadpool);
```

```
typedef struct llama_batch {
    int32_t n_tokens;

    llama_token  *  token;
    float        *  embd;
    llama_pos    *  pos;
    int32_t      *  n_seq_id;
    llama_seq_id ** seq_id;
    int8_t       *  logits; // TODO: rename this to "output"
} llama_batch;
```

多条seq

```
llama_batch batch = llama_batch_get_one(prompt_tokens.data(), prompt_tokens.size());

// main loop

const auto t_main_start = ggml_time_us();
int n_decode = 0;
llama_token new_token_id;

for (int n_pos = 0; n_pos + batch.n_tokens < n_prompt + n_predict; ) {
    // evaluate the current batch with the transformer model
    if (llama_decode(ctx, batch)) {
        fprintf(stderr, "%s : failed to eval, return code %d\n", __func__, 1);
        return 1;
    }

    n_pos += batch.n_tokens;

    // sample the next token
    {
        new_token_id = llama_sampler_sample(smpl, ctx, -1);

        // is it an end of generation?
        if (llama_vocab_is_eog(vocab, new_token_id)) {
            break;
        }

        char buf[128];
        int n = llama_token_to_piece(vocab, new_token_id, buf, sizeof(buf), 0, true);
        if (n < 0) {
            fprintf(stderr, "%s: error: failed to convert token to piece\n", __func__);
            return 1;
        }
        std::string s(buf, n);
        printf("%s", s.c_str());
        fflush(stdout);

        // prepare the next batch with the sampled token
        batch = llama_batch_get_one(&new_token_id, 1);

        n_decode += 1;
    }
}
```

构建计算图llama_build_graph()

- struct llm_build_context llm(lctx, batch, cb, worst_case);
- 根据模型结构选择Build方法

```
    case LLM_ARCH_QWEN:
        {
            result = llm.build_qwen();
        } break;
```

```
//      qwen构建流程 --- 按照qwen的模型结构，使用ggml的通用op定义，来搭建qwen的model
struct ggml_cgraph * gf = ggml_new_graph_custom ----- 创建一个计算图
llm_build_norm --- 构建norm-op
{ ---- 构建 self-attention
        ggml_mul_mat
        ggml_add
        ggml_reshape_3d
        ggml_rope_custom
        llm_build_kv
}
{ ---- 构建feed-forward forward
        llm_build_norm
        llm_build_ffn
}
ggml_add
```

```
// ggml_add

static struct ggml_tensor * ggml_add_impl(
        struct ggml_context * ctx,
        struct ggml_tensor  * a,
        struct ggml_tensor  * b,
        bool                  inplace) {
    GGML_ASSERT(ggml_can_repeat(b, a));

    struct ggml_tensor * result = inplace ? ggml_view_tensor(ctx, a) : ggml_dup_tensor(ctx, a);

    result->op     = GGML_OP_ADD;
    result->src[0] = a;
    result->src[1] = b;

    return result;
}
```

构建计算图： 以llama的ffn为例

```
struct ggml_tensor * tmp = up ? llm_build_lora_mm(lctx, ctx, up, cur) : cur;
```

```
// feed-forward network
if (model.layers[il].ffn_gate_inp == nullptr) {

    cur = llm_build_norm(ctx0, ffn_inp, hparams,
            model.layers[il].ffn_norm, NULL,
            LLM_NORM_RMS, cb, il);
    cb(cur, "ffn_norm", il);

    cur = llm_build_ffn(ctx0, lctx, cur,
            model.layers[il].ffn_up,   model.layers[il].ffn_up_b,   NULL,
            model.layers[il].ffn_gate, model.layers[il].ffn_gate_b, NULL,
            model.layers[il].ffn_down, model.layers[il].ffn_down_b, NULL,
            NULL,
            LLM_FFN_SILU, LLM_FFN_PAR, cb, il);
    cb(cur, "ffn_out", il);
```

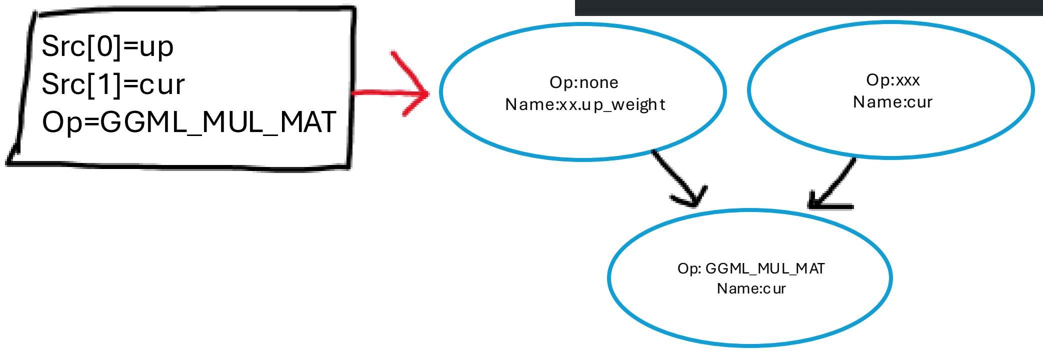
cur^T · up

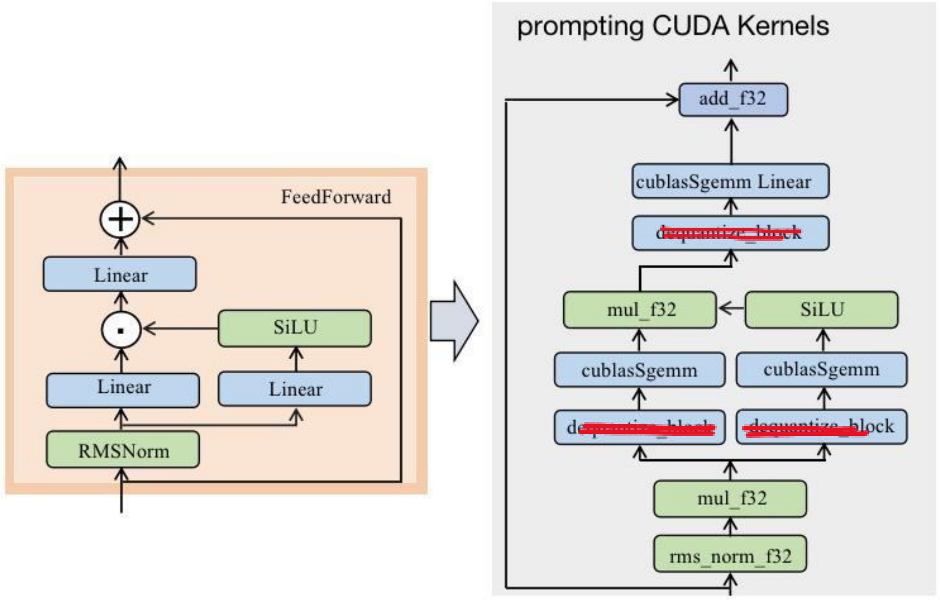
```
struct ggml_tensor * res = ggml_mul_mat(ctx0, w, cur);
```

```
// A: k columns, n rows => [ne03, ne02, n, k]
// B: k columns, m rows  (i.e. we transpose it internally) => [ne03 * x, ne02 * y, m, k]
// result is n columns, m rows => [ne03 * x, ne02 * y, m, n]
GGML_API struct ggml_tensor * ggml_mul_mat(
        struct ggml_context * ctx,
        struct ggml_tensor  * a,
        struct ggml_tensor  * b);
```

```
struct ggml_tensor * result = ggml_new_tensor(ctx, GGML_TYPE_F32, 4, ne);

result->op     = GGML_OP_MUL_MAT;
result->src[0] = a;
result->src[1] = b;

return result;
```

Ggml_tensor *res





➢ 回调函数cb

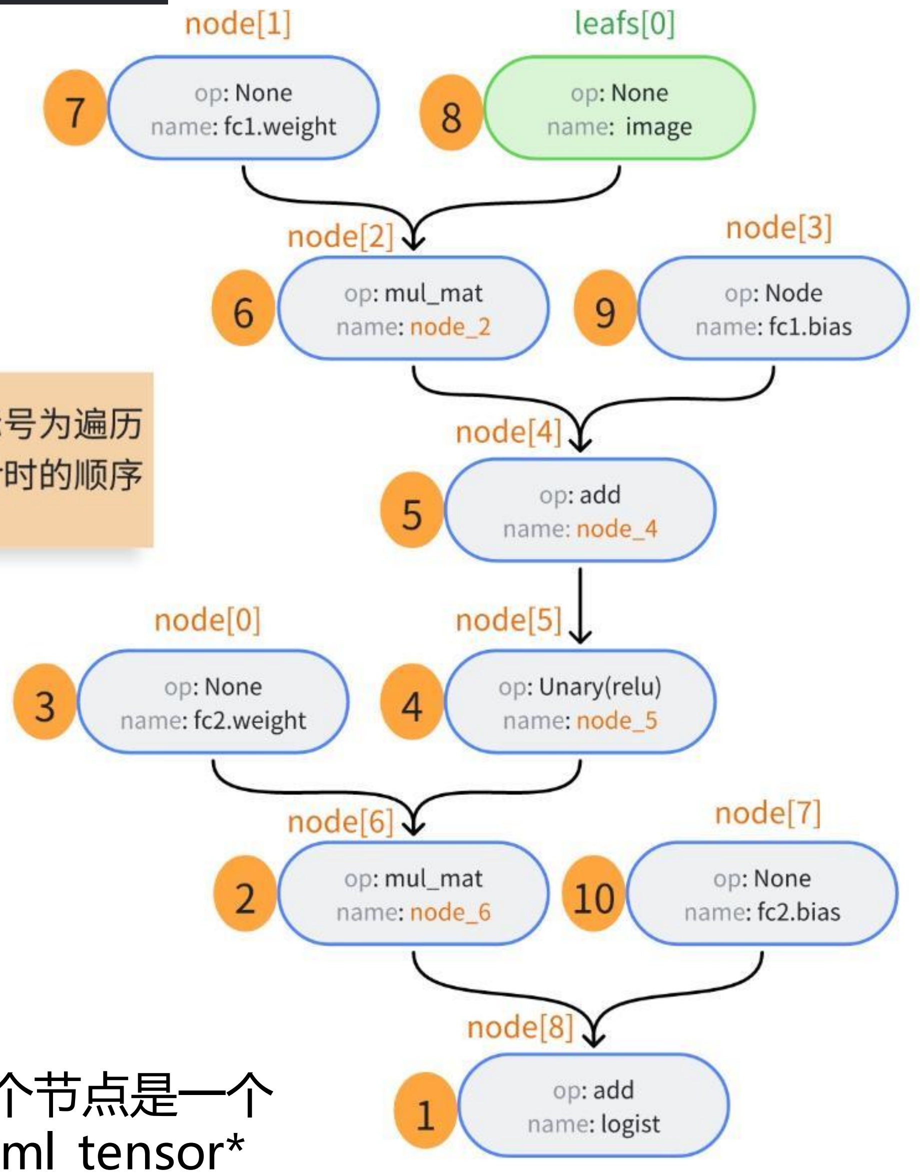
图中绿色代表叶节点、蓝色为普通节点node。而叶节点的定义为：即不是权重weight tensor、也没有任何op操作的节点。不满足这两个调节的都是普通节点node

构建compute_graph可视化

```
using llm_build_cb = std::function<void(struct ggml_tensor * cur, const char * name, int nl)>;
```

```
llm_build_cb cb = [&](struct ggml_tensor * cur, const char * name, int il) {
    if (il >= 0) {
        ggml_format_name(cur, "%s-%d", name, il);
    } else {
        ggml_set_name(cur, name);
    }

    if (!lctx.cparams.offload_kqv) {
        if (strcmp(name, "kqv_merged_cont") == 0) {
            // all nodes between the KV store and the attention output are run on the CPU
            ggml_backend_sched_set_tensor_backend(lctx.sched.get(), cur, lctx.backend_cpu);
        }
    }

    // norm may be automatically assigned to the backend of the previous layer, increasing data transf
    // FIXME: fix in ggml_backend_sched
    const bool full_offload = lctx.model.params.n_gpu_layers > (int) lctx.model.hparams.n_layer;
    if (ubatch.n_tokens < 32 || full_offload) {
        if (il != -1 && strcmp(name, "norm") == 0) {
            const auto & dev_layer = lctx.model.dev_layer(il);
            for (auto & backend : lctx.backends) {
                if (ggml_backend_get_device(backend.get()) == dev_layer) {
                    if (ggml_backend_supports_op(backend.get(), cur)) {
                        ggml_backend_sched_set_tensor_backend(lctx.sched.get(), cur, backend.get());
                    }
                }
            }
        }
    }
};
```

橘色标号为遍历tensor时的顺序



得到计算图后，执行计算图时调ggml对应的cuda算子

```
if (up_b) {
    tmp = ggml_add(ctx, tmp, up_b);
    cb(tmp, "ffn_up_b", il);
}
```

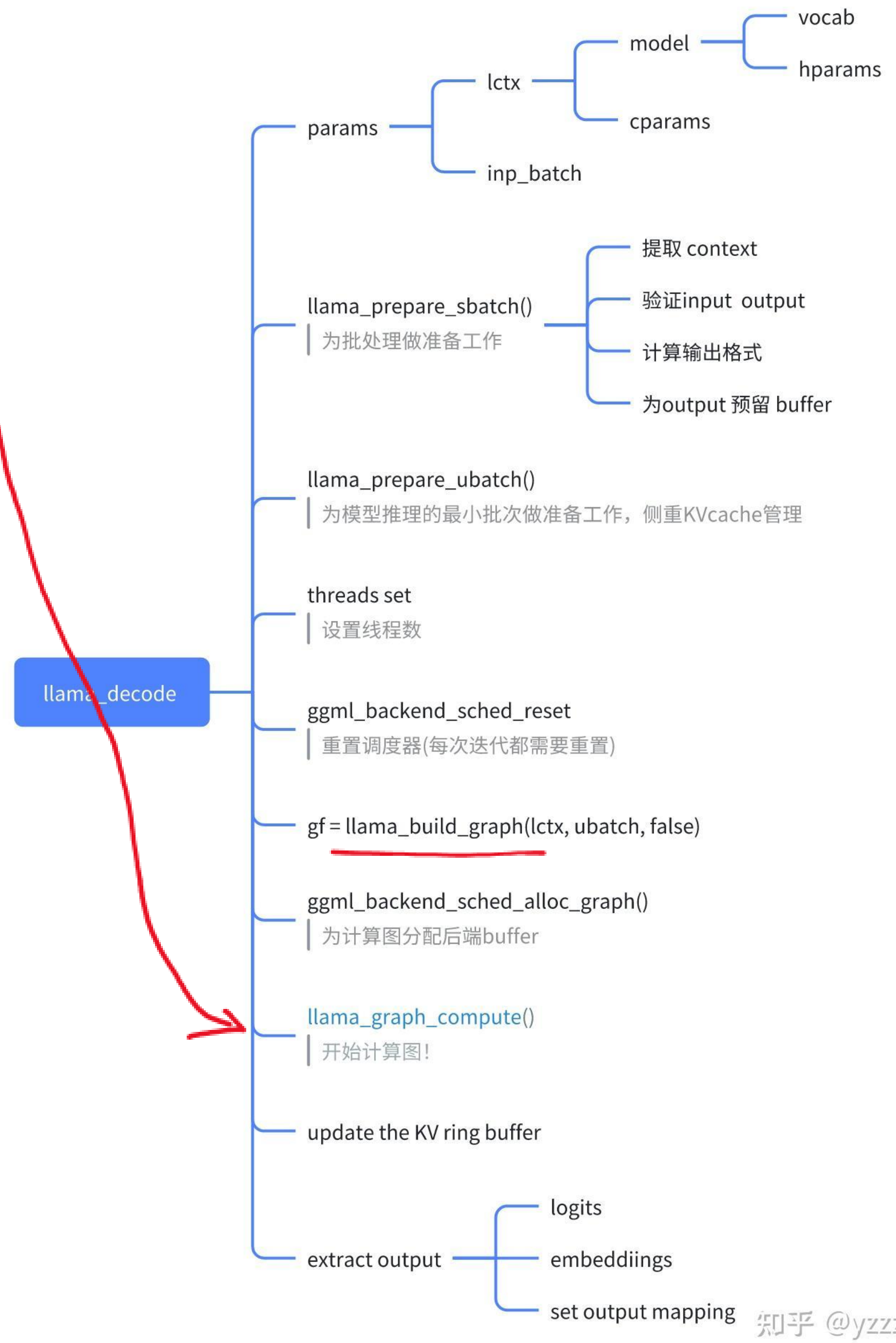
每个节点是一个ggml_tensor*

➢ 异步计算计算图 ggml_backend_sched_graph_compute_async

```
// returns the result of ggml_backend_sched_graph_compute_async execution
static enum ggml_status llama_graph_compute(
          llama_context & lctx,
            ggml_cgraph * gf,
                    int   n_threads,
        ggml_threadpool * threadpool) {
    if (lctx.backend_cpu != nullptr) {
        auto * reg = ggml_backend_dev_backend_reg(ggml_backend_get_device(lctx.backend_cpu));
        auto * set_threadpool_fn = (decltype(ggml_backend_cpu_set_threadpool) *) ggml_backend_reg_get_proc_address(
        set_threadpool_fn(lctx.backend_cpu, threadpool);
    }

    // set the number of threads for all the backends
    for (const auto & set_n_threads_fn : lctx.set_n_threads_fns) {
        set_n_threads_fn.second(set_n_threads_fn.first, n_threads);
    }

    auto status = ggml_backend_sched_graph_compute_async(lctx.sched.get(), gf);
    if (status != GGML_STATUS_SUCCESS) {
        LLAMA_LOG_ERROR("%s: ggml_backend_sched_graph_compute_async failed with error %d\n", __func__, status);
    }

    // fprintf(stderr, "splits: %d\n", ggml_backend_sched_get_n_splits(lctx.sched));

    return status;
}
```

➢ 释放资源

```
llama_sampler_free(smpl);
llama_free(ctx);
llama_model_free(model);
```



➢ 具体算子执行ggml/src/ggml-cuda/ggml-cuda.cu

```
static bool ggml_cuda_compute_forward(ggml_backend_cuda_context & ctx, struct ggml_tensor * dst)
switch (dst->op)
    case GGML_OP_ACC:
        ggml_cuda_op_acc(ctx, dst);
```

```
void ggml_cuda_op_acc(ggml_backend_cuda_context & ctx, ggml_tensor * dst) {
    const ggml_tensor * src0 = dst->src[0];
    const ggml_tensor * src1 = dst->src[1];
    const float * src0_d = (const float *)src0->data;
    const float * src1_d = (const float *)src1->data;
    float * dst_d = (float *)dst->data;
    cudaStream_t stream = ctx.stream();

    GGML_ASSERT(src0->type == GGML_TYPE_F32);
    GGML_ASSERT(src1->type == GGML_TYPE_F32);
    GGML_ASSERT( dst->type == GGML_TYPE_F32);
    GGML_ASSERT(dst->ne[3] == 1); // just 3D tensors supported

    int nb1 = dst->op_params[0] / 4; // 4 bytes of float32
    int nb2 = dst->op_params[1] / 4; // 4 bytes of float32
    // int nb3 = dst->op_params[2] / 4; // 4 bytes of float32 - unused
    int offset = dst->op_params[3] / 4; // offset in bytes

    acc_f32_cuda(src0_d, src1_d, dst_d, ggml_nelements(dst), src1->ne[0], src1->ne[1], src1->ne[2], nb1, nb2, offset, stream);
}
```

➢ Cuda算子实现"acc.cuh"

```
static __global__ void acc_f32(const float * x, const float * y, float * dst, const int ne,
    const int ne10, const int ne11, const int ne12,
    const int nb1, const int nb2, int offset) {
    const int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i >= ne) {
        return;
    }
    int src1_idx = i - offset;
    int oz = src1_idx / nb2;
    int oy = (src1_idx - (oz * nb2)) / nb1;
    int ox = src1_idx % nb1;
    if (src1_idx >= 0 && ox < ne10 && oy < ne11 && oz < ne12) {
        dst[i] = x[i] + y[ox + oy * ne10 + oz * ne10 * ne11];
    } else {
        dst[i] = x[i];
    }
}
```

```
static void acc_f32_cuda(const float * x, const float * y, float * dst, const int n_elements,
    const int ne10, const int ne11, const int ne12,
    const int nb1, const int nb2, const int offset, cudaStream_t stream) {
    int num_blocks = (n_elements + CUDA_ACC_BLOCK_SIZE - 1) / CUDA_ACC_BLOCK_SIZE;
    acc_f32<<<num_blocks, CUDA_ACC_BLOCK_SIZE, 0, stream>>>(x, y, dst, n_elements, ne10, ne11, ne12, nb1, nb2, offset);
}
```

➢ 关键的算子，矩阵乘法

```
case GGML_OP_MUL_MAT:
    ggml_cuda_mul_mat(ctx, dst->src[0], dst->src[1], dst);
    break;
```

矩阵乘法ggml底层:
static void ggml_cuda_mul_mat(ggml_backend_cuda_context & ctx, const ggml_tensor * src0, const ggml_tensor * src1, ggml_tensor * dst)

ggml_cuda_mul_mat主要完成了对 src0 （参数数组），src1（运算数组）两者的矩阵乘法，并存入 dst（结果数组）之中。

根据不同的条件，函数会调用以下实现之一：
•ggml_cuda_mul_mat_vec：适用于小矩阵或没有张量核心的GPU。
•ggml_cuda_mul_mat_batched_cublas：适用于多batch的FP16矩阵乘法。
•ggml_cuda_op_mul_mat：通用的矩阵乘法实现，支持量化类型和其他特殊场景。

代码（ggml_cuda_op_mul_mat）中，运算逻辑会被拆分为三层，batch、device、tilling 三个层面三个循环。最后的矩阵才会交给cublasGemmEx、cublasSgemm

Powerinfer

- 可能是基于llama.cpp的早期版本，与现在llama.cpp结构还是有些不一样
- 删除了一些不用的ggml库，只保留了cuda和cpu相关的库

- Convert.py
- 还有一个convert-hf-to-powerinfer-gguf.py文件，功能好像差不多，类似llama.cpp里的那个

PredictorParams类定义了预测器的参数，目前只有一个参数sparse_threshold，它是一个可选的浮点数，这是powerinfer官方提供的限定稀疏度的模型

main prosparse-llama-2-7b-gguf

Yixin Song minor 72f4a41

- activation
- .gitattributes Safe
- README.md Safe
- config.json Safe
- prosparse-llama-2-7b-clip15.gguf
- prosparse-llama-2-7b.gguf

```python
@dataclass
class PredictorParams:
    sparse_threshold: float | None = None

    @staticmethod
    def loadPredictorJson(model: LazyModel, config_path: Path) -> PredictorParams:
        config = json.load(open(config_path))
        return PredictorParams(
            sparse_threshold = config.get("sparse_threshold"),
        )

    @staticmethod
    def load(model_plus: ModelPlus) -> PredictorParams:
        config_path   = model_plus.paths[0].parent / "config.json"

        if config_path.exists():
            params = PredictorParams.loadPredictorJson(model_plus.model, config_path)
        else:
            params = PredictorParams()

        return params
```

- 预测器结构，参数加载

```python
class ReluMLP(tnn.Module):
    def __init__(self, input_dim: int, hidden_dim: int, output_dim: int):
        super(ReluMLP, self).__init__()
        self.fc1 = tnn.Linear(input_dim, hidden_dim, bias=False)
        self.relu = tnn.ReLU()
        self.fc2 = tnn.Linear(hidden_dim, output_dim, bias=False)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x

    @staticmethod
    def from_file(model_file: Path):
        model = torch.load(model_file, map_location="cpu")
        hidden_size, input_size = model.get("fc1.weight").shape
        output_size, _ = model.get("fc2.weight").shape
        mlp = ReluMLP(input_size, hidden_size, output_size)
        mlp.load_state_dict(model)
        return mlp
```

```python
def get_tensors(self) -> Iterator[tuple[str, Tensor]]:
    for model_layer, part_name in self._get_mlp_part_layer_names():
        print(f"gguf: loading mlp part '{part_name}'")
        mlp_model = ReluMLP.from_file(self.dir_mlp_pred / part_name)
        for name, data in mlp_model.state_dict().items():
            yield f"blk.{model_layer}.{name}", data
```

```
blk.29.fc1.weight f16     [  4096,  1024,     1,     1 ]
blk.29.fc2.weight f16     [  1024, 11008,     1,     1 ]
```

➢ 模型加载 llama_model_load()

```
llm_load_arch    (ml, model);
llm_load_hparams(ml, model);
llm_load_vocab   (ml, model);
```

~~基本步骤差不多~~，加载模型参数有点区别。
这里会利用powerinfer-py里的solver以及模型文件夹activation文件夹的历史激活信息决定冷热神经元的放置

```
        if (llama_use_sparse_inference(&model)) {
            if (params.n_gpu_layers > 0) {
                LLAMA_LOG_WARN("%s: sparse inference ignores n_gpu_layers, you can use --vram-budget option instead\n", __func__);
                return false;
            }
#if defined GGML_USE_CUBLAS
            llama_set_vram_budget(params.vram_budget_gb, params.main_gpu);
#endif
            llm_load_sparse_model_tensors(
                ml, model, cparams, params.main_gpu, vram_budget_bytes, params.reset_gpu_index, params.disable_gpu_index,
                params.use_mlock, params.progress_callback, params.progress_callback_user_data
            );
        } else {
            llm_load_tensors(
                ml, model, params.n_gpu_layers, params.main_gpu, params.tensor_split, params.use_mlock,
                params.progress_callback, params.progress_callback_user_data
            );
        }
```

Llama.cpp版本

具体还没看

- Powerinfer实现了opt等模型，不用自己实现
- 构建计算图也有区别，主要是fnn的区别

```
// feed-forward network
{
    cur = llm_build_norm(ctx0, ffn_inp, hparams,
            model.layers[il].ffn_norm, model.layers[il].ffn_norm_b,
            LLM_NORM, cb, il);

    if(llama_use_sparse_inference(&model)) {
        llm_build_cb_short cbs = [&](ggml_tensor * cur, const char * name) {
            std::string name_str = std::string(name) + "-" + std::to_string(il);
            ggml_set_name(cur, name_str.c_str());
        };
        // We only offload the ffn input to GPU if all neurons are offloaded
        if (model.layers[il].gpu_offload_ratio >= 1.) {
            cb(cur, "ffn_norm", il);
        } else {
            cbs(cur, "ffn_norm");
        }
        cur = llm_build_ffn_sparse(ctx0, cur,
            model.layers[il].ffn_up,   model.layers[il].ffn_up_b,
            NULL,                      NULL,
            model.layers[il].ffn_down_t, model.layers[il].ffn_down_b,
            model.layers[il].mlp_pre_w1,
            model.layers[il].mlp_pre_w2,
            ffn_inp,
            model.layers[il].gpu_idx,
            model.layers[il].gpu_bucket, model.layers[il].ffn_gate_gpu, model.layers[il].ffn_down_gpu, model.layers[il].ffn_up_gpu,
            LLM_FFN_RELU, LLM_FFN_SEQ, model.layers[il].gpu_offload_ratio, cbs);
```

```
switch (model.arch) {
    case LLM_ARCH_LLAMA:
    case LLM_ARCH_BAMBOO:
        {
            result = llm.build_llama_variants();
        } break;
    case LLM_ARCH_BAICHUAN:
        {
            result = llm.build_baichuan();
        } break;
    case LLM_ARCH_FALCON:
        {
            result = llm.build_falcon();
        } break;
    case LLM_ARCH_STARCODER:
        {
            result = llm.build_starcoder();
        } break;
    case LLM_ARCH_PERSIMMON:
        {
            result = llm.build_persimmon();
        } break;
    case LLM_ARCH_REFACT:
        {
            result = llm.build_refact();
        } break;
    case LLM_ARCH_BLOOM:
        {
            result = llm.build_bloom();
        } break;
    case LLM_ARCH_MPT:
        {
            result = llm.build_mpt();
        } break;
     case LLM_ARCH_STABLELM:
        {
            result = llm.build_stablelm();
        } break;
    case LLM_ARCH_OPT:
        {
            result = llm.build_opt();
        } break;
    default:
        GGML_ASSERT(false);
}
```

➢ 底层算子差别

在 ggml_cuda_compute_forward 中，稀疏算子

```
case GGML_OP_MUL_MAT_SPARSE:
    if (!src0_on_device && !ggml_cuda_can_mul_mat(tensor->src[0], tensor->src[1], tensor)) {
        return false;
    }
    func = ggml_cuda_mul_mat_sparse;
    break;
```

```
static void ggml_cuda_mul_mat_sparse(const ggml_tensor * src0, const ggml_tensor * src1, ggml_tensor * dst) {
    GGML_ASSERT(dst->src[2] != NULL && "dst->src[2] must be present for sparse matrix multiplication");
    if (src1->ne[1] == 1 && src0->ne[0] % GGML_CUDA_DMMV_X == 0) {
        switch(src0->type) {
            case GGML_TYPE_F16:
                ggml_cuda_op_mul_mat(src0, src1, dst, ggml_cuda_op_mul_mat_vec_sparse_dequantized, false);
                break;
            case GGML_TYPE_Q4_0:
                ggml_cuda_op_mul_mat(src0, src1, dst, ggml_cuda_op_mul_mat_vec_sparse_q, true);
                break;
            default:
                GGML_ASSERT(false && "unsupported type for sparse matrix multiplication");
        }
    } else {
        ggml_cuda_op_mul_mat(src0, src1, dst, ggml_cuda_op_mul_mat_batch_sparse, false);
    }
}
```

Llama.cpp推测解码

- 支持多种推测解码采样：greedy,stochastic,tree-based(详见文档)

- 大致流程：

1.参数解析与初始化
2.Tokenize prompt
3.主循环：
  - 使用草模型生成多个可能后续token序列（草稿序列,这里生成的长度是不固定的，也有采样策略）。
  - 验证这些稿草稿序列中的token是否符合目标模型的采样分布，若符合则接受该token，否则拒绝。

4.资源释放

➢ Llama.cpp支持的采样方式

```
std::vector<enum common_sampler_type> samplers = {
    COMMON_SAMPLER_TYPE_PENALTIES,
    COMMON_SAMPLER_TYPE_DRY,
    COMMON_SAMPLER_TYPE_TOP_K,
    COMMON_SAMPLER_TYPE_TYPICAL_P,
    COMMON_SAMPLER_TYPE_TOP_P,
    COMMON_SAMPLER_TYPE_MIN_P,
    COMMON_SAMPLER_TYPE_XTC,
    COMMON_SAMPLER_TYPE_TEMPERATURE,
};
```

➢ 重要的推测解码参数

```
struct common_speculative_params {
    int n_draft = 16;  // max drafted tokens
    int n_reuse = 256;

    float p_min = 0.75f; // min probability required to accept a token in the draft
};
```

➢ Draft model 推测解码的抽象

```
struct common_speculative {
    struct llama_context * ctx;
    struct common_sampler * smpl;

    llama_batch batch;
    llama_tokens prompt;
};
```