Capstone Notes

Questions/Doubts:

- [x] /*.no_alloc =*/ false, // NOTE: this should be false when using the legacy API —> ??
  Ans: I'm guessing that legacy API ends up allocating memory for tensors by default. ggml_init did not care to check this param before calculating mem size to allocate. Well it just allocated whatever value it was asked to by the caller.
- [ ] ggml_nbytes() - logic esp wrt number of number of blocks and block size is NOT understood! Gotta clarify this sometime. Skipping this since it is not directly needed during new tensor creation.
- [ ] ggml_new_tensor_impl() - not entirely clear esp wrt the case when the view_src is not null. No idea what is going on wrt nb and its calculations. Got the overall gist though.
- [ ] Find out how hash tables work and why size of hash table is a prime number

Notes from llama.cpp:

Note: NPU acceleration support can be limited to specific ops and not extend fully to things like de-quantization.
Major takeaways from ggml-metal.m

- Thanks to UMA (unified memory arch) of Apple silicon, the CPU, GPU (and hence NPU) share the same virtual address space. Hence tensor/data alloc/cpy/free are just system calls to do the same as with normal memory.
- ggml_backend_buffer_i struct has pointers to the functions of the respective backend which handle with copy and memset for tensors and buffer respectively.
- **ggml_metal_encode_node** maps actions to kernels.
- ggml-metal.m (obj c) -> ggml_init() -> GGML_METAL_ADD_KERNEL(xyz) -> template [... kernel_xyz] in ggml.metal (mps code)
- Found the basic math and activation functions in mps code. kernel_add_row kernel_sub_row kernel_mul_row kernel_div_row, kernel_relu and so on. Around LN 800 in ggml-metal.metal

ggml thread pool contains compute graph and compute plan. Once all these are ready, ggml_graph_compute_thread kicks off all (have not explored on the ordering. Presuming that you start with all the max_leaf_node_level-1 and nodes at a given level are computed in parallel). The threads themselves perform ggml_compute_forward (incl the main thread).

ggml_compute_forward contains the actual math logic. Destination is also a portion of the tensor's data memory space and is passed to the external matmul api (ex: accelerate)

Cpumasking is/can be used in threadpools to specify which core or cores the given thread is allowed to run on. Without cpumask, threads may jump across CPUs, leading to cache thrashing. Useful in NUMA (Non-Uniform Memory Access) systems to keep threads on the same memory node.

A massive buffer of type *GGML_OBJECT_TYPE_WORK_BUFFER is allocated by* ggml_graph_compute_with_ctx (based on info from ggml_graph_plan)
to accommodate for all the ops that are going to happen, taking into account data types, quantization, etc

ggml_graph_node uses python like indexing which can be either positive or negative

During mem alloc for context, graph overhead func takes into account the mem size needed for default graph (size 2048)
Interesting way to ensure a number is always odd: `num | 1`

view_offs seems to be offset within the ctx mem space (i.e ctx mem base address + view_offset = address of a given object)

ggml_new_object -> create a new object per se (since mem is already allocated, just gotta reserve it) and then moves the cur off and cur size pointers accordingly. It then adds the new member to the ggml objects linked list and moves the next pointer (objects_end ptr).

Clever way to calculate size of a N dimensional tensor

```
size_t data_size = ggml_row_size(type, ne[0]);
for (int i = 1; i < n_dims; i++) {
   data_size *= ne[i];
}
```

view_src maybe something else. Not just a ptr to view the data but the actual ref point for the data within the larger backend ctx mem space.

ggml_new_tensor_2d - controls the tensor data type info (fp32/16/etc)

Looks like tensor data, graph data, etc are allocated and stored within the ggml

context memory area since ctx size calc includes space required to store the tensor data and 1KB extra space.
ggml_graph_overhead -> memory required to store the graph with all nodes and other metadata being aligned to uintptr_t (8 bytes in 64 bit) memory boundaries.

Ggml context as a whole is aligned as per GGML_MEM_ALIGN (16 bytes)

ggml_context
Buffer and memory status, pointers to ggml object linked list
ggml_object => size , offset, object type and next pointer (could perhaps point to the actual tensor in memory)

ggml object types:
GGML_OBJECT_TYPE_TENSOR,
GGML_OBJECT_TYPE_GRAPH,
GGML_OBJECT_TYPE_WORK_BUFFER

ggml_tensor
buffer and backend interface,
type of op (probably the one it is going through at a given moment)
Pointers for actual data and for viewing
ggml_tensor * [10] array -> pointers to source tensors that make up the actual tensor. Perhaps this is needed for multi device split training?
Is 336 bytes in total! Lot of stuff indeed!

iface member of a structure is the gateway to calling all the functions related to it

Ggml backend buffer usage types:
GGML_BACKEND_BUFFER_USAGE_ANY **=> iniating the backend buffer (ggml_backend_buffer_init)**
GGML_BACKEND_BUFFER_USAGE_WEIGHTS **=> loading models (like gpt2)**
GGML_BACKEND_BUFFER_USAGE_COMPUTE **=> cuda buffer and misc things like that (ggml_backend_cuda_buffer_init_tensor, ggml_gallocr_reserve_n)**

Follow **https://www.reddit.com/r/LocalLLaMA/comments/1at3hu2/how_to_learn_the_base_code_of_llamacpp_im/ and other tabs!**
Llama cpp first commit - https://github.com/ggml-org/llama.cpp/tree/26c084662903ddaca19bef982831bfb0856e8257 - ,arch 10, 2023 (Friday)

llama.cpp uses ggml library! So lot of stuff could actually outside llama.cpp repo (although the ggml dir should ideally have it all)
https://github.com/ggml-org/ggml/blob/d013aa56cdcccefd9086ac93a58951256c84ff16/src/ggml-metal/ggml-metal.metal#L4

https://github.com/ggml-org/ggml/tree/master/examples/simple -> contains simple example usage of ggml backend for the simple case of matmul

ggml_backend_sched_reset -> ggml_backend_sched_set_eval_callback -> llama_build_graph -> ggml_backend_sched_alloc_graph -> llama_graph_compute -> -> ggml_backend_sched_get_tensor_backend -> ggml_backend_tensor_get_async

HellaSwag score
winograde
get only sentence embedding
 enable reranking support on server - ?

 imatrix params?

 add npu to ggml_backend_registry

Consider using llama_split_mode (LLAMA_SPLIT_MODE_LAYER LLAMA_SPLIT_MODE_ROW)

Imp funcs:
llama_model_load
llm_load_tensors

Openmp not found! cMake error
Param reranking - ?

Tensors are loaded one by one, by name using **llama_tensor_weight()**

ggml_context - mem buffer and a linked list of ggml objects which could be

tensor, graph or work buffer
if the provided gguf_context is no_alloc , **Model params are loaded as a binary blob in continuous memory using calloc and then ggml structs are pointed to the right locations within the blob**

If alloc is needed, the first port contains the struct and the data follows it.
_____
_____
___
_____
_____
___
Notes from internet

Running on ANE using coreML - https://developer.apple.com/forums/thread/729942

- It seems like ANE will work with EnumeratedShapes within Flexible Shapes. https://apple.github.io/coremltools/docs-guides/source/faqs.html#neural-engine-with-flexible-input-shapes
- I figured it out; apparently flexible shapes do not run on the ANE.
- I really wish this was documented; the docs just state to use enumerated shapes for best performance.
- But in this case, using flexible shapes is nearly **10 times slower** and I don't understand why they are *supported* at all with that kind of penalty.

https://github.com/ggml-org/llama.cpp/discussions/336

Apple has a reference implementation for transformers on the ANE.
https://github.com/apple/ml-ane-transformers

I've done some research on what would be required to utilize the Neural Engine on Apple devices as a ggml backend.
It turns out that there are new CoreML APIs that are available since the latest OS releases (macOS 15+, iOS 18+, etc.) that allow allocating tensors directly (without having to use a model in the coreml format like before) and applying operations on them efficiently using the Neural Engine.
It's only available for usage from Swift and not Objective-C/C++, but we can expose the functions we need from Swift and use them from a cpp wrapper.
Here's an example of a matmul operation using CoreML in Swift from the Apple

documentation:

```
let v1 = MLTensor([1.0, 2.0, 3.0, 4.0])
let v2 = MLTensor([5.0, 6.0, 7.0, 8.0])
let v3 = v1.matmul(v2)
v3.shape // is []
await v3.shapedArray(of: Float.self) // is 70.0


let m1 = MLTensor(shape: [2, 3], scalars: [
    1, 2, 3,
    4, 5, 6
], scalarType: Float.self)
let m2 = MLTensor(shape: [3, 2], scalars: [
     7,  8,
     9, 10,
    11, 12
], scalarType: Float.self)
let m3 = t1.matmul(r2)
m3.shape // is [2, 2]
await m3.shapedArray(of: Float.self) // is [[58, 64], [139, 154]]


// Supports broadcasting
let m4 = MLTensor(randomNormal: [3, 1, 1, 4], scalarType: Float.self)
let m5 = MLTensor(randomNormal: [4, 2], scalarType: Float.self)
let m6 = t4.matmul(t5)
m6.shape // is [3, 1, 1, 2]
```

To use the Neural Engine, the tensor operation needs to be wrapped with withMLTensorComputePolicy with a MLComputePolicy initialized with MLComputeUnits.cpuAndNeuralEngine (it can also be initialized with MLComputeUnits.all to let the OS spread the load between the Neural Engine, GPU and CPU).
This is a new API that was not available previously, so using this would mean that the Neural Engine support (using CoreML) will only be supported on recent OS versions.
The main benefit of this would be that we would utilize more of the compute available on Apple chips, which will allow performing more parallel operations that are also optimized on the chip level, which should lead to faster inference.
I'm not sure I'll have enough time to implement this myself soon, though.
I'll update if I do get to it to ensure we don't do duplicate work on this.

https://machinelearning.apple.com/research/neural-engine-transformers

- ANE prefers 4D tensors
- Last axis is unpacked
- Tensors are chunked to maximize L2 redundancy
- Multi head attention split into single chunked head attention
- Minimizing transpose ops using einsum
- Minimize men fetch since that gets bottlenecked on the io bandwidth -> either increase batch size (so that weights can be applied to as many inputs as possible before going back to fetch next set of weights from the DRAM and/or reduce param size by quantization
-

Principle 1: Picking the Right Data Format
In general, the Transformer architecture processes a 3D input tensor that comprises a batch of B sequences of S embedding vectors of dimensionality C. We represent this tensor in the (B, C, 1, S) data format because the most conducive data format for the ANE (hardware and software stack) is 4D and channels-first.
The native torch.nn.Transformer and many other PyTorch implementations use either the (B, S, C) or the (S, B, C) data formats, which are both channels-last and 3D data formats. These data formats are compatible with nn.Linear layers, which constitute a major chunk of compute in the Transformer. To migrate to the desirable (B, C, 1, S) data format, we swap all nn.Linear layers with nn.Conv2d layers. Furthermore, to preserve compatibility with previously trained checkpoints using the baseline implementation, we register a load_state_dict_pre_hook to automatically unsqueeze the nn.Linear weights twice in order to match the expected nn.Conv2d weights shape as shown here.

“The mapping of the sequence (S) axis to the last axis of the 4D data format is very important because the last axis of an ANE buffer is not packed; it must be contiguous and aligned to 64 bytes. “

———

PyTorch mps source code - https://github.com/pytorch/pytorch/tree/main/aten/src/ATen/native/metal, https://github.com/pytorch/pytorch/tree/main/aten/src/ATen/mps