2025.01.13

since uniform memory
actually, specify stream
expr. transformations work on funcion objects
specify device (cpu, gpu) at the expression / result array
but imperative and with stateful Module class
grad, vmap, vjp, jvp
pythonic: classes, generators
expression based like JAX
good C interface
Apple's MLX
array vs. vector
tracing -> compute graph
formats .npy, .npz, .safetensors (HuggingFace), .gguf (GGreganov)
explicitly executed with eval()
implicitly executed by print, memoryview, save
currently not compiled
distributed comp. via MPI
compile() optimizes graph e.g. by fusion
all_sum
all_gather
send
recv
an array
only for pure functions
exporting functions, can be imported cross-frontend
can compile the whole SGD update, support for capturing mutable state
exporting modules with or w/o params
only for fixed shapes, via example inputs

2025.01.15

can compile from commandline or at runtime a/synchronously from string: newLibraryWithSource:options: completionHandler/error

multilevel command buffer: blit/compute encoders

Apple Metal quick glance

doesn't support `double` (?)

2025.01.19

habits
atomic
obvious
define trigger:
definite time+location,
or end of earlier habit
attractive
community
temptation / ritual
satisfying
immediate reward
on completion
use habit tracker
easy
reduce friction,
prime the environment
downscale habit to two minutes
break down big ones into multiple
establish a <2min
minimal satisfying version
modify habit
keep trigger and reward,
replace routine
Stepladder -- small changes
Community
Important -- dwell on priorities
Easy
Neurohack -- behavior drives identity
Captivating rewards
Engrained -- repetition

2025-01.20

Andrej Karpathy's educational resources (github repositories)
Lecture 8
optimizes minGPT
nanoGPT
the first GPT in PyTorch repo, purely educational
minGPT
ng-video-lecture
follows the first, 2-hours lecture reproducing nanoGPT
improved Lecture
build-nanogpt
follows latest, 4-hours lecture reproducing nanoGPT
introductory and experimental parts of the lectures
nn-zero-to-hero
Lecture 1
micrograd
minimal NumPy NN framework
Lecture 8
lectures 2-5
makemore
bigram character-level LM
minBPE
tokenizer for GPT
llm.c
very efficient C/CUDA, multi-GPU with ZeRO
optimized for pretraining GPT-2/3
simple reference impls train_gpt2.c in pure C, ~~train_gpt2.cu in CUDA,~~ train_gpt2.py in PyTorch
this is the entry point to main impl.
llama2.c
train.py in PyTorch
run.c in FP32 pure C
runq.c in int8 quantized
char-rnn
character-level LM
old, in Lua Torch
vanilla Recurrent NN, GRU, LSTM

2025.01.21

llm.c

- uses cuBLAS, cuBLASLt in code for **matmul** forward and backward, manual kernel for backward bias term
- manual kernels for: encoder forward and backward, layernorm forward and backward, softmax forward and (in-place) backward, Cross-Entropy forward fused with backward, AdamW
- hyperparams for loading any GPT2 or GPT3
- random init scheme from GPT2, init computed on the CPU
- uses ZeRO for multi-GPU
  - see slide 2024.11.20
  - uses MPI to distribute

To Be Continued

- two variants: with & w/o cuDNN, used for attention
- judiciously uses size_t to not overflow int
- uses stochastic rounding from FP32 to BF16
- conserves memory by reusing buffers
- manually manages memory (no per-tensor alloc/free)

2025.01.23

llm.c distributed

- Sharded Data Parallel and FSDP not supported yet -- only sharding optimizer states aka. ZeRO-1
- dedicated CUDA stream for NCCL ops
- llmc/zero.cuh here, NCCL and MPI overview on future maps
- `multi_gpu_async_reduce_gradient`: reduce-scatter if ZeRO else all-reduce
- different network socket API causes some duplication between Windows and Linux
- `gpt2_calculate_grad_norm`: reuses the activations buffer; without ZeRO, gradients already averaged across all GPUs, sums norms locally; with ZeRO, need to all-reduce-sum the norms

2025.01.24

**CUDA Cooperative Groups**

API for thread subsets

first-class thread blocks

`coalsesced_group active = coalesced_threads()`

currently executing in a warp

`active.sync()` no deadlock

protects from deadlocks wrt. `__syncthreads()`

`group.sync()`

no. of this thread in this group

`group.thread_rank()`

`tiled_partition(group, size)`

if `size=32`, gives warps

`thread_block_tile<size>`
`tiled_partition<size>(group)`

`#pragma unroll` since size is static

then, compiler can remove synchronizations (unsafe when done manually)

```
thread_block_tile::...
shfl()
shfl_down()
shfl_up()
shfl_xor()
any()
all()
ballot()
match_any()
match_all()
```

**CUDA Warp-level primitives**

32-bit (int) masks pick threads of a warp

synchronized data exchange:
`__all_sync`, `__any_sync`, `__uni_sync`,
`__ballot_sync`;
`__shfl_sync`, `__shfl_up_sync`,
`__shfl_down_sync`, `__shfl_xor_sync`;
`__match_any_sync`, `__match_all_sync`

Get `__activemask;`
Sync with memory fence
`__syncwarp(mask=FULL_MASK)`
0xffffffff

`__ffs`,
`__popc`

To Be Continued

2025.01.25

**CUDA warp functions:**
**Vote, Match, Reduce, Shuffle**

`__match_any_sync`: returns the mask of threads with the same value as the calling thread

`__match_all_sync`: returns the given mask if all its threads have the same value, otherwise 0

`__all_sync`, `__any_sync`: the value is non-zero for all/any of the threads

each calling thread must be in the mask, all masks must be the same

`__ballot_sync`: returns the mask of threads with non-zero value

`__reduce_add/min/max_sync`: reduces the int or unsigned values

`__reduce_and/or/xor_sync`: logical op reduces unsigned values

Warp Matrix Functions leverage Tensor Cores for Matrix Multiply Add

shfl: exchange a variable between threads of a warp (faster than shared mem)

unlike reduce functions, works with all numeric types, including `__half2` and `__nv_bfloat162`

optional width (one of 2,4,8,16) subdivides operation into groups, with group-relative addressing

`__shfl_sync` specifies source lane explicitly

`__shfl_up/down_sync` specify delta, source is lower/higher

`__shfl_xor_sync` bitwise XORs calling thread lane ID with the given lane mask

see: butterfly pattern

2025.01.26

- **cuDNN: CUDA Deep Neural Network**
  - Backend: C API
    - opt-in to use Tensor Core (in backend, not in frontend?)
  - multiple heuristic-based execution plans
    - autotuning
    - filtering: numerical, behavior, functional properties
    - policy-based selection
  - Graph: fusion etc.
    - cuDNN graph performs inference: shapes for virtual/temp tensors, strides, precisions
      - configurable defaults for: io, intermediate, and compute data type
      - via `Graph::validate`
  - Frontend: C++ and Python APIs
    - C++ based on shared pointers
  - all tensors have from 3 to 8 axes (with leading dims 1 if not needed)
  - Matrix Mult.
    - matmul broadcasts even non-1 batch axes if needed
  - Attention: forward, backprop
    - only for FP16, BF16, FP8
  - multinode graphs do not support in-place operations
  - Convolution: forward, data grad, weight grad
  - Pointwise: add, bias, scale, sub, mul, rsqrt, relu, elu, gelu, cmp_gt
  - Batch Normalization: forward, backprop, finalize stats
  - Layernorm: forward, backprop
  - Instancenorm: forward, backprop

To Be Continued

2025.01.27

virtual tensors can be any type, but recommended FP32

mixed precision inputs via (pointwise) identity

on Ada Lovelace, FP8 inputs trigger FP8 Tensor Cores

compute type FP32 / CUDNN_DATA_FLOAT (recommended for backward pass) and CUDDN_DATA_FAST_FLOAT_FOR_FP8

require exactly one batch axis

generic runtime fusion engines: only for pointwise ops --> matmul or convolution (or none) --> pointwise ops [--> reduction op]

pre-compiled single operation engines: convolution and normalization ops

cuDNN Graphs

specialized pre-compiled engines

specialized runtime fusion patterns

Convolution-BatchNorm with ReLU activation

Fused Attention with max seq length 512, forward and backward, e.g. similar to BERT and T5

ResNet helpers: BachNorm forward (with optional Add, ReLU, and (> 0) side output) and backward (with optional dReLU and side grad output for fwd's Add)

FP8 Fused Flash Attention max sequence length 512

support multi-GPU batches

allows optional: scalar key scaling, padding or causal masks, softmax, dropout

Fused Flash Attention forward and backward, usable GPT and BERT like models

configurable with many scaling, mask and dropout options

2025.01.28

**MPI: Message Passing Interface and collective operations**

llm.c usage

can initialize NCCL with: tcp, mpi, fs (fs: file system synchronization)

MPI_Bcast to initialize NCCL rank

MPI_Allgather to find the GPU's ordinal on a machine

1 GPU = 1 process

Send, Recv: point-to-point communication

Reduce-Scatter: pointwise reduces a vector and scatters the results

Broadcast: send from one to all

Barrier: achieves global synchronization

Gather: send from all, all values received by one node in a container

All-Gather: send from all, every node recevies all values in a container

All-to-all: each node has a container, from which it sends a different value to every node's receiving container

Reduce: send from all, reduce on the fly into a value received by one

All-Reduce: send from all, receive the same reduced value by all

Scan: each node receives a partially reduced value depending on its rank

Scatter: send from a container on one node, a different value to every node

vectorized versions of these, and dedicated versions where values are arrays

2025.01.29

NCCL: NVIDIA Collective Communications Library

- deals with different GPU and interconnect types
  - PCIe, NVLINK, InfiniBand Verbs, IP sockets
- single kernel handling both communication and computation
- All-Reduce, Broadcast, Reduce, All-Gather, Reduce-Scatter, Send, Recv
  - root rank gets the result of Reduce
- control: single-threaded, multi-threaded, multi-process including MPI
- a communicator has nodes (ranks) and issues collective operations, a communicator *object* is a node
  - `ncclUniqueId` is the root rank, w/ it a given rank communicates. Can be a set `ncclUniqueIds`, all nodes must have the same set.
- optionally can block, default async via stream queues
  - but enqueueing can block on other ranks to arrive first
- Using communicators concurrently can cause deadlocks. Even with separate streams, e.g. one uses too many blocks.
- Cooperative Thread Arrays: threads of a warp?
- group calls
  - dynamic scope ncclGroupStart/End
  - must be used when single thread manages multiple devices
  - aggregating operations might optimize communication
- a communicator object of a device can use host pointers, cannot use pointers of a peer device, to avoid programmer errors
- to avoid internal copies: `ncclMemAlloc` & `ncclCommRegister`
- Don't use streams while NCCL uses them

2025.01.30

NCCL usage in llm.c
NCCL setup and helpers in zero.cuh
gpt2_update: All-Gather updated shards of params
gpt2_backward_and_reduce: All-Reduce gradients per-layer in the last microbatch step, also All-Reduce accumulated_mean_loss
multi_gpu_async_reduce_gradient: if not sharding then All-Reduce, if ZeRO-1 then Reduce-Scatter
gpt2_calculate_grad_norm: All-Reduce grad_norm_squared
All-Reduce to compute memory stats
main: Periodically compute and All-Reduce: validation loss, HellaSwag accuracy

2025.01.31

unlike OCANNL compilation is optional

all graphs can be compiled with dynamic shapes (unlike tinygrad and current OCANNL)

lazy like tinygrad, but more control like OCANNL: explicitly forced, can be explicitly compiled

API the same as NumPy/PyTorch?

cumbersome, verbose notation

compilation pipeline: shape inference, high-level -> scheduler i.e. kernel demarcation and ordering -> low-level -> polyhedral IR -> rendering kernels

High-level (API) and low-level (AIR) graph interfaces, codegen, graph runner

quick first glance at Caten

focus on "NN inference runtime"

interesting to look at:
source/api/**tensor**.lisp
source/codegen/**scheduler**.lisp
source/codegen/**jit**.lisp
source/codegen/**memory-planner**.lisp
(minimizes peak memory usage)
source/byoc/**metal**.lisp
external/llm/**layers**.lisp

verdict: the code is too sprawling (e.g. too many files) to be worth digging into; better focus on tinygrad

but it's impressive, maybe I'll have some luck finding inspiration