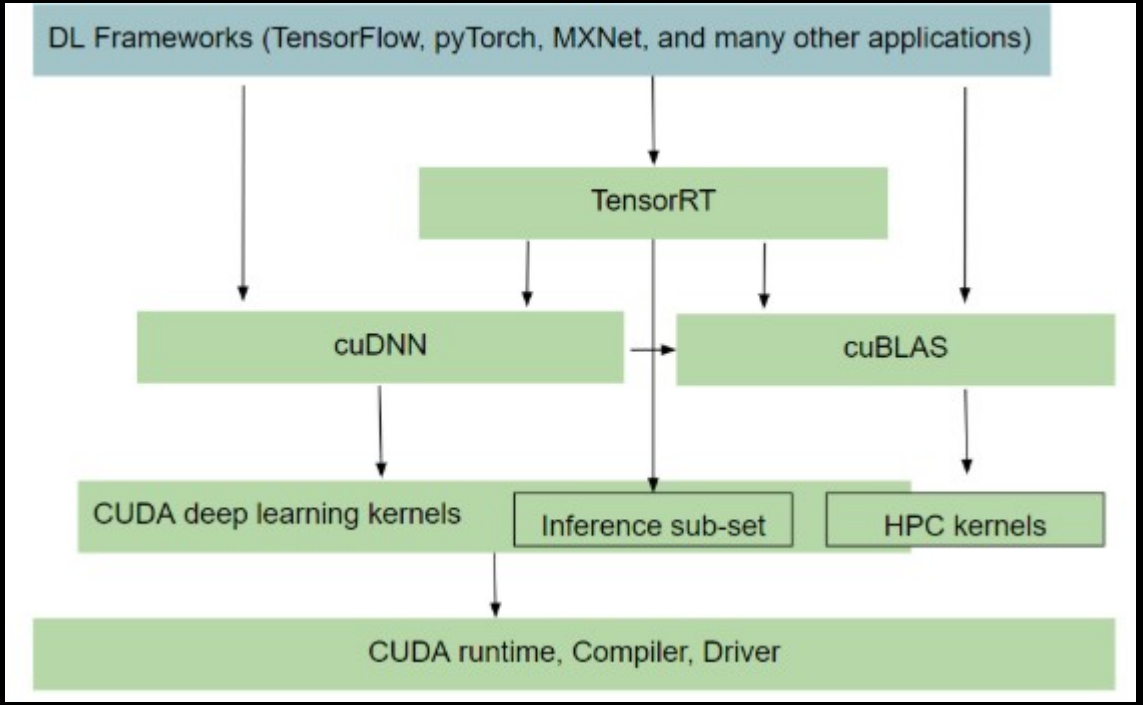
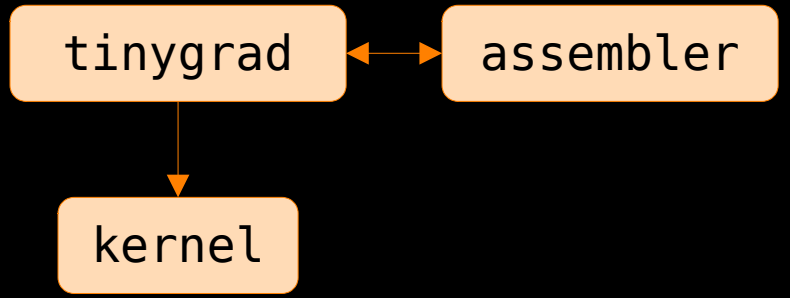
tinygrad: from MNIST to ALUs

What is tinygrad?

- A neural network framework
- Pure Python (seriously)
- Very small (<8000 lines)
- Yet fully functional

The tinygrad stack





Almost no dependencies => it’s easy to port new accelerators

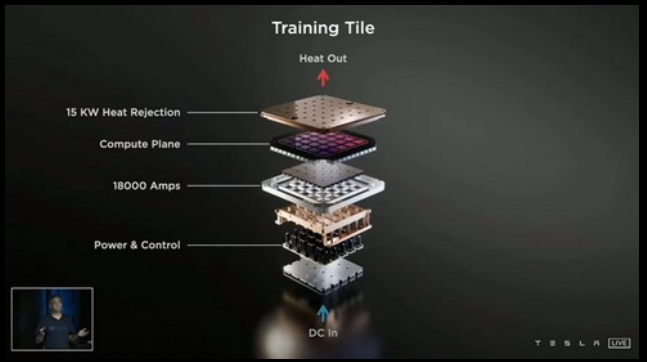
Why a new framework?

- To commoditize the petaflop
- The graveyard of AI chip companies is big.
- To be successful with your chip, you must be able to create your own stack

groq

tenstorrent

GRAPHCORE



A torch-like frontend

```python
from tinygrad import Tensor, nn

class Model:
  def __init__(self):
    self.l1 = nn.Conv2d(1, 32, kernel_size=(3,3))
    self.l2 = nn.Conv2d(32, 64, kernel_size=(3,3))
    self.l3 = nn.Linear(1600, 10)

  def __call__(self, x:Tensor) -> Tensor:
    x = self.l1(x).relu().max_pool2d((2,2))
    x = self.l2(x).relu().max_pool2d((2,2))
    return self.l3(x.flatten(1).dropout(0.5))
```

- No `nn.Module` class
- No `forward`
- No classes for stateless operations
- Many Tensor methods

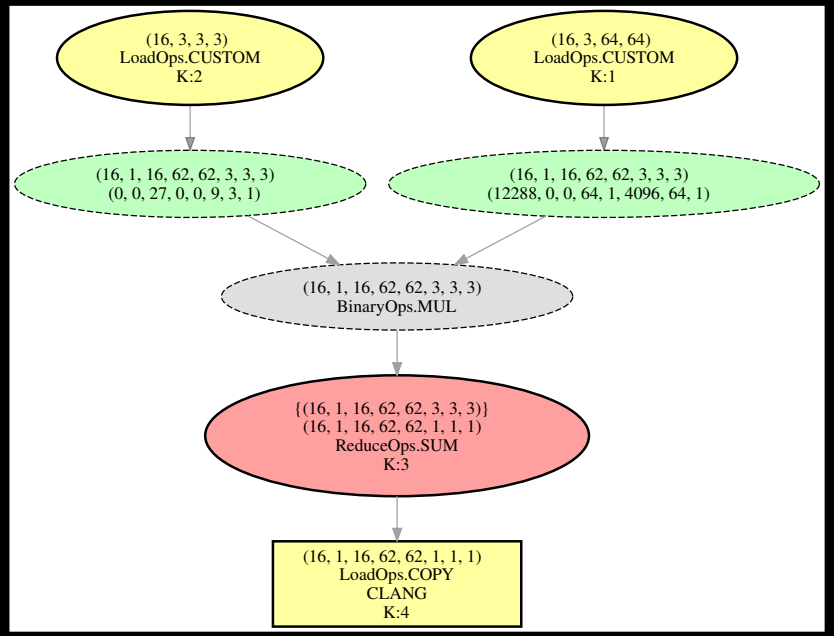
docs.tinygrad.org

tinygrad is lazy

- Eager – operations happen when they run (PyTorch)
- Graph – operations happen after the graph is compiled (TensorFlow, torch.compile)
- Lazy – implicit graph, the simplicity of eager with the power of graph

The LazyBuffer graph

```
jesse@x1:~/tinygrad$ GRAPH=1 python3 -c "from tinygrad import Tensor; Tensor.rand(16,3,64,64).conv2d(Tensor.rand(16,3,3,3)).numpy()"
saving DiGraph with 7 nodes and 6 edges to /tmp/net.svg
jesse@x1:~/tinygrad$
```



- LoadOps.CUSTOM is Tensor.rand
- Green is a “view”
- A conv is two views, a MUL, and a SUM
- We copy back to the CPU (aka CLANG)

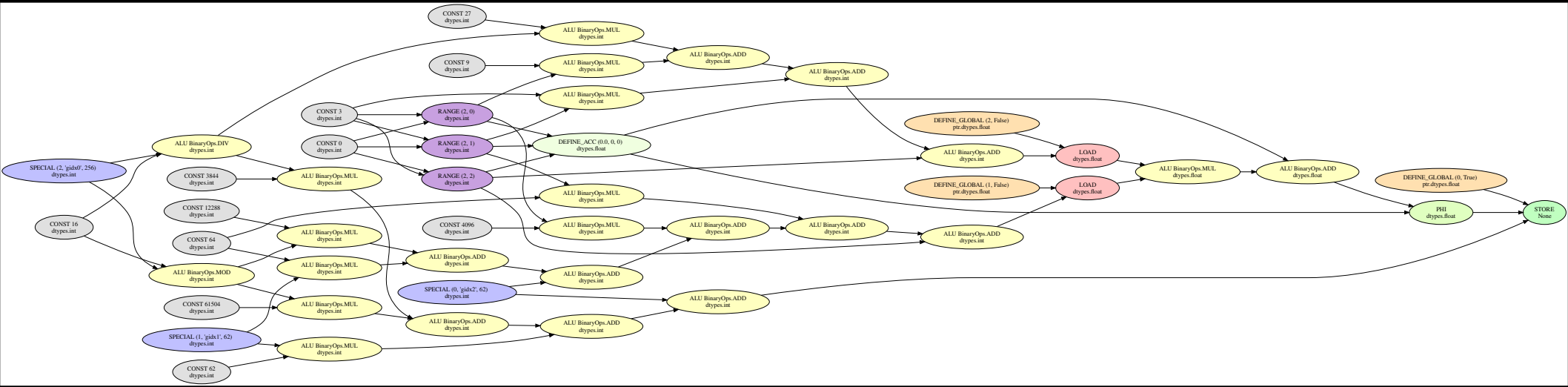
The code (conv2d)

```
jesse@x1:~/tinygrad$ NOOPT=1 DEBUG=4 python3 -c "from tinygrad import Tensor; Tensor.rand(16,3,64,64).conv2d(Tensor.rand(16,3,3,3)).numpy()"
CLDevice: got 1 platforms and 1 devices
opened device GPU from pid:73068
opened device NPY from pid:73068
*** CUSTOM     1 custom_random                       arg   1 mem  0.00 GB
*** CUSTOM     2 custom_random                       arg   1 mem  0.00 GB
  0 ┌─ STORE MemBuffer(idx=0, dtype=dtypes.float, st=ShapeTracker(views=(View(shape=(16, 1, 16, 62, 62, 1, 1, 1), strides=(61504, 0, 3844, 62, 1, 0, 0, 0), offset=0, mask=None, contiguous=True),)))
  1 └┬─ SUM (7, 6, 5)
  2  └┬─ MUL
  3   ├── LOAD MemBuffer(idx=1, dtype=dtypes.float, st=ShapeTracker(views=(View(shape=(16, 1, 16, 62, 62, 3, 3, 3), strides=(12288, 0, 0, 64, 1, 4096, 64, 1), offset=0, mask=None, contiguous=False),)))
  4   └── LOAD MemBuffer(idx=2, dtype=dtypes.float, st=ShapeTracker(views=(View(shape=(16, 1, 16, 62, 62, 3, 3, 3), strides=(0, 0, 27, 0, 0, 9, 3, 1), offset=0, mask=None, contiguous=False),)))
__kernel void r_16_16_62_62_3_3_3(__global float* data0, const __global float* data1, const __global float* data2) {
  int gidx2 = get_group_id(0); /* 62 */
  int gidx1 = get_group_id(1); /* 62 */
  int gidx0 = get_group_id(2); /* 256 */
  int alu0 = (gidx0%16);
  int alu1 = (gidx0/16);
  float acc0 = 0.0f;
  for (int ridx0 = 0; ridx0 < 3; ridx0++) {
    for (int ridx1 = 0; ridx1 < 3; ridx1++) {
      for (int ridx2 = 0; ridx2 < 3; ridx2++) {
        float val0 = data1[(alu0*12288)+(gidx1*64)+gidx2+(ridx0*4096)+(ridx1*64)+ridx2];
        float val1 = data2[(alu1*27)+(ridx0*9)+(ridx1*3)+ridx2];
        acc0 = ((val0*val1)+acc0);
      }
    }
  }
  data0[(alu0*61504)+(alu1*3844)+(gidx1*62)+gidx2] = acc0;
}
*** GPU        3 r_16_16_62_62_3_3_3                 arg   3 mem  0.00 GB tm   6686.67us/     6.69ms (    7.95 GFLOPS,    0.71 GB/s)
opened device CLANG from pid:73068
*** CLANG      4 copy    3.94M,   CLANG <- GPU       arg   2 mem  0.01 GB tm   5523.24us/    12.21ms (    0.00 GFLOPS,    0.71 GB/s)
avg:     4.35 GFLOPS     0.71 GB/s          total:     4 kernels     0.05 GOPS     0.01 GB    12.21 ms
jesse@x1:~/tinygrad$
```

An OpenCL kernel implementing a 3x3 conv

The UOps (conv2d)

```
jesse@x1:~/tinygrad$ NOOPT=1 GRAPHUOPS=1 python3 -c "from tinygrad import Tensor; Tensor.rand(16,3,64,64).conv2d(Tensor.rand(16,3,3,3)).numpy()"
saving DiGraph with 50 nodes and 68 edges to /tmp/net.uops.svg
jesse@x1:~/tinygrad$
```



Slow?

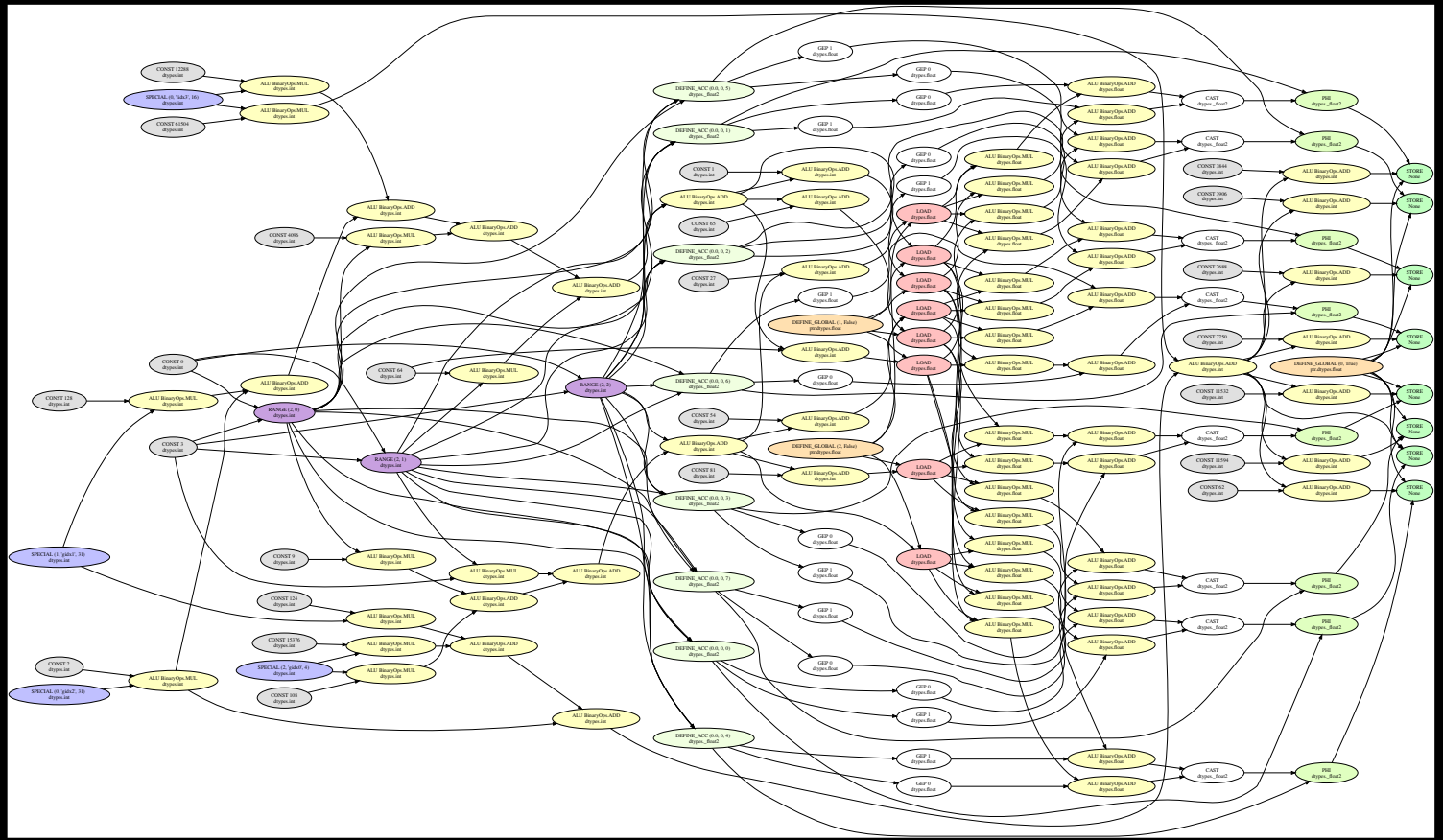
- Problem: Tons of ops are spent on indexing
- Solution: compute multiple outputs (a chunk) in the kernel
- Question: what size chunk is optimal?
- Answer: search the possible kernels!

BEAM search

```python
class OptOps(Enum):
  TC = auto(); UPCAST = auto(); UPCASTMID = auto(); UNROLL = auto(); LOCAL = auto() # noqa: E702
  GROUP = auto(); GROUPTOP = auto(); NOLOCALS = auto(); PADTO = auto() # noqa: E702
  def __lt__(self, x:OptOps): return self.value < x.value
```

```
jesse@x1:~/tinygrad$ DEBUG=2 BEAM=2 python3 -c "from tinygrad import Tensor; Tensor.rand(16,3,64,64).conv2d(Tensor.rand(16,3,3,3)).numpy()"
CLDevice: got 1 platforms and 1 devices
opened device GPU from pid:75057
opened device NPY from pid:75057
*** CUSTOM     1 custom_random                          arg   1 mem  0.00 GB
*** CUSTOM     2 custom_random                          arg   1 mem  0.00 GB
   0.00s:                 from   1 ->   1 actions   16   16   62   62    3    3    3
   1.65s:       935.62 us from  26 ->  26 actions   16   62   62   16    3    3    3
  12.05s:       367.71 us from  44 ->  44 actions    4   62   62   16    3    3    3    4
  22.80s:       253.23 us from  38 ->  38 actions   62   62   16    3    3    3    4    4
  33.40s:       219.48 us from  32 ->  32 actions    4   31   31   16    3    3    3    4    2    2
  43.18s:       219.48 us from  30 ->  30 actions    4   31   31   16    3    3    3    4    2    2
beam2  : 4 31 31 16 3 3 3 4 2 2        :   275.00 us  <  hc     : 4 31 31 4 2 2 3 4 4 3 3       :   322.39 us
*** GPU        3 r_4_31_31_16_3_3_3_4_2_2               arg   3 mem  0.00 GB tm    392.81us/     0.39ms (  135.28 GFLOPS,   12.03 GB/s)
opened device CLANG from pid:75057
*** CLANG      4 copy    3.94M,   CLANG <- GPU          arg   2 mem  0.01 GB tm   2443.85us/     2.84ms (    0.00 GFLOPS,    1.61 GB/s)
avg:    18.73 GFLOPS     3.05 GB/s          total:     4 kernels     0.05 GOPS     0.01 GB     2.84 ms
jesse@x1:~/tinygrad$
```

The Optimized UOps (conv2d)



Philosophy of tinygrad

- Surface all complexity
  - Don’t rely on libraries, many of which are vendor specific with quirks.
- No Turing complete abstractions
  - Rules out use of LLVM, LLVM IR has thrown away too much information.
- Embrace ”The Bitter Lesson”
  - There’s many choices to be made, don’t spend time designing heuristics, use search.

Model training

Follow along with the MNIST tutorial on docs.tinygrad.org

```
model = Model()
optim = nn.optim.Adam(nn.state.get_parameters(model))
batch_size = 128
@TinyJit
def train_step():
  Tensor.training = True  # makes dropout work
  samples = Tensor.randint(batch_size, high=X_train.shape[0])
  X, Y = X_train[samples], Y_train[samples]
  optim.zero_grad()
  loss = model(X).sparse_categorical_crossentropy(Y).backward()
  optim.step()
  return loss
```

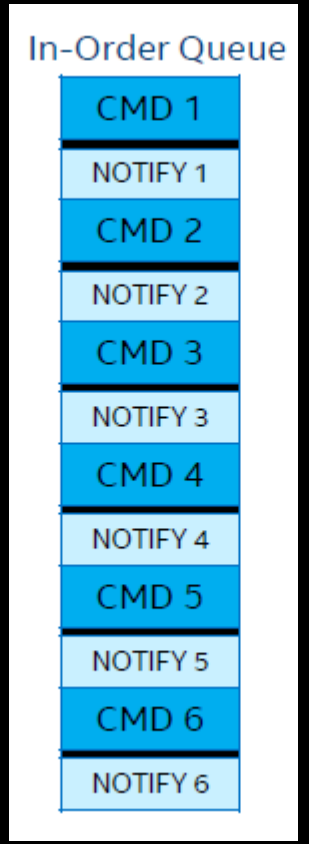
What is @TinyJit (DEBUG=2)

```
jit execs 45 kernels
*** GPU       1 E_                                    arg  1 mem  0.12 GB tm     14.90us/     0.01ms (     0.00 GFLOPS,    0.00 GB/s)
*** GPU       2 E_n1                                  arg  1 mem  0.12 GB tm     11.56us/     0.03ms (     0.00 GFLOPS,    0.00 GB/s)
*** CUSTOM    3 custom_random                         arg  1 mem  0.12 GB
*** GPU       4 r_625_32_15000_3_4                    arg  1 mem  0.12 GB tm     28.33us/     0.05ms (    45.88 GFLOPS,    8.47 GB/s)
*** CUSTOM    5 custom_random                         arg  1 mem  0.12 GB
*** GPU       6 r_5_2_10                              arg  1 mem  0.12 GB tm     11.25us/     0.07ms (     0.01 GFLOPS,    0.00 GB/s)
*** GPU       7 E_n2                                  arg  2 mem  0.12 GB tm     11.04us/     0.08ms (     0.00 GFLOPS,    0.00 GB/s)
*** GPU       8 E_n2                                  arg  2 mem  0.12 GB tm     11.25us/     0.09ms (     0.00 GFLOPS,    0.00 GB/s)
*** GPU       9 E_32_4                                arg  2 mem  0.12 GB tm     17.50us/     0.11ms (     0.03 GFLOPS,    0.06 GB/s)
*** GPU      10 r_8_49_5_4_4_8_375_4_4_4              arg  4 mem  0.12 GB tm     23.37ms/    23.48ms (   733.04 GFLOPS,    2.19 GB/s)
*** GPU      11 r_125_32_2_60_4_4                     arg  4 mem  0.12 GB tm    154.06us/    23.63ms (   199.40 GFLOPS,    2.16 GB/s)
*** GPU      12 r_3136_32_10_4                        arg  2 mem  0.12 GB tm    192.71us/    23.82ms (    20.83 GFLOPS,   21.35 GB/s)
*** GPU      13 r2_4_32_250                           arg  3 mem  0.12 GB tm    104.17us/    23.93ms (     0.31 GFLOPS,    0.31 GB/s)
*** GPU      14 r_32_13_13_8_2_2_4_4_3_3              arg  4 mem  0.12 GB tm    983.75us/    24.91ms (    56.29 GFLOPS,   11.36 GB/s)
*** GPU      15 r_16_8                                arg  2 mem  0.12 GB tm     13.33us/    24.93ms (     0.01 GFLOPS,    0.01 GB/s)
*** GPU      16 r_1664_13_32_2_2                      arg  2 mem  0.12 GB tm    567.50us/    25.49ms (     4.88 GFLOPS,   24.40 GB/s)
*** GPU      17 r_416_13_32_4_2_2                     arg  3 mem  0.12 GB tm    576.88us/    26.07ms (    14.40 GFLOPS,   28.80 GB/s)
*** GPU      18 r_4_11_11_8_16_32_4_4_3_3             arg  4 mem  0.12 GB tm   1958.33us/    28.03ms (   292.56 GFLOPS,    3.48 GB/s)
*** GPU      19 r_256_5_5_32_2_2                      arg  2 mem  0.12 GB tm    437.40us/    28.47ms (     1.87 GFLOPS,    9.36 GB/s)
*** GPU      20 r_64_5_5_32_4_2_2                     arg  3 mem  0.12 GB tm    357.08us/    28.82ms (     6.88 GFLOPS,   13.76 GB/s)
*** GPU      21 E_1600_32_4                           arg  3 mem  0.12 GB tm     69.48us/    28.89ms (    11.79 GFLOPS,   35.37 GB/s)
*** GPU      22 r_128_10_16_100                       arg  4 mem  0.12 GB tm    289.69us/    29.18ms (    14.14 GFLOPS,    3.07 GB/s)
*** GPU      23 r_4_32_10                             arg  2 mem  0.12 GB tm     18.96us/    29.20ms (     0.07 GFLOPS,    0.30 GB/s)
*** GPU      24 r_4_32_10n1                           arg  3 mem  0.12 GB tm     16.56us/    29.22ms (     0.23 GFLOPS,    0.37 GB/s)
*** GPU      25 r_4_32_10n2                           arg  3 mem  0.12 GB tm     19.38us/    29.24ms (     0.26 GFLOPS,    0.32 GB/s)
*** GPU      26 r_4_32_10n3                           arg  6 mem  0.12 GB tm     15.31us/    29.25ms (     0.51 GFLOPS,    0.11 GB/s)
*** GPU      27 r_4_32_10n4                           arg  8 mem  0.12 GB tm     26.35us/    29.28ms (     0.53 GFLOPS,    0.28 GB/s)
*** GPU      28 E_5_32_2_4                            arg 10 mem  0.12 GB tm     30.42us/    29.31ms (     0.59 GFLOPS,    0.43 GB/s)
*** GPU      29 r3_10_16_8n1                          arg  7 mem  0.12 GB tm     20.83us/    29.33ms (     0.06 GFLOPS,    0.25 GB/s)
*** GPU      30 r_4_25_8_16_4_4_10                    arg  4 mem  0.12 GB tm     87.92us/    29.42ms (    55.91 GFLOPS,   19.42 GB/s)
*** GPU      31 E_64_5_5_32_2_2_4                     arg  5 mem  0.12 GB tm    878.02us/    30.30ms (     3.73 GFLOPS,   11.05 GB/s)
*** GPU      32 r3_5_25_2_16_32_4_4n1                 arg  8 mem  0.12 GB tm    158.65us/    30.45ms (    26.12 GFLOPS,    5.60 GB/s)
*** GPU      33 E_16_11_11_8_16_4                     arg  3 mem  0.12 GB tm    887.71us/    31.34ms (     2.23 GFLOPS,   12.62 GB/s)
*** GPU      34 r3_64_16_8_121n1                      arg  7 mem  0.12 GB tm    683.54us/    32.03ms (     1.45 GFLOPS,    5.80 GB/s)
*** GPU      35 r_16_2_121_2_16_3_16_3_4_4            arg  3 mem  0.12 GB tm   2546.88us/    34.57ms (   224.18 GFLOPS,    8.59 GB/s)
*** GPU      36 r_16_2_13_13_8_16_4_4                 arg  2 mem  0.12 GB tm   1156.67us/    35.73ms (     9.58 GFLOPS,   17.82 GB/s)
*** GPU      37 r3_8_2_2_16_3_128_11_3_4_11n1         arg  8 mem  0.12 GB tm   4190.83us/    39.92ms (   136.25 GFLOPS,    1.62 GB/s)
*** GPU      38 E_32_13_13_32_2_2_4                   arg  5 mem  0.12 GB tm   1600.52us/    41.52ms (    10.38 GFLOPS,   19.03 GB/s)
*** GPU      39 r_32_8_3_4_26_3_4_26                  arg  3 mem  0.12 GB tm    549.48us/    42.07ms (    90.70 GFLOPS,   20.61 GB/s)
*** GPU      40 r_4_8_8_16_169_4                      arg  2 mem  0.12 GB tm    360.31us/    42.43ms (     7.68 GFLOPS,   30.78 GB/s)
*** GPU      41 r3_288_16_8n1                         arg  7 mem  0.12 GB tm     20.42us/    42.45ms (     1.85 GFLOPS,    7.28 GB/s)
*** GPU      42 r3_32_16_8n1                          arg  7 mem  0.12 GB tm     15.52us/    42.47ms (     0.27 GFLOPS,    1.06 GB/s)
*** GPU      43 E_32_4n1                              arg  2 mem  0.12 GB tm     20.00us/    42.49ms (     0.01 GFLOPS,    0.05 GB/s)
*** GPU      44 r_16_8n1                              arg  2 mem  0.12 GB tm     11.67us/    42.50ms (     0.01 GFLOPS,    0.01 GB/s)
*** GPU      45 r_16_8_10                             arg  8 mem  0.12 GB tm     32.60us/    42.53ms (     0.27 GFLOPS,    0.21 GB/s)
```

It captures the run kernels and replays them with new data

What are CUDA Graphs?

- GPUs use command queues to execute kernels. They are what they sound like.
- Model training runs can be ~10,000 kernels.
- The CPU time spent enqueuing the kernels can exceed the GPU runtime
- So...reuse the same command queue!



NV/AMD backends

```
jesse@x1:~/tinygrad$ ./sz.py | grep runtime/ops_
tinygrad/runtime/ops_nv.py                  516
tinygrad/runtime/ops_amd.py                 419
tinygrad/runtime/ops_hsa.py                 225
tinygrad/runtime/ops_python.py              180
tinygrad/runtime/ops_cuda.py                162
tinygrad/runtime/ops_metal.py                99
tinygrad/runtime/ops_gpu.py                  91
tinygrad/runtime/ops_disk.py                 55
tinygrad/runtime/ops_llvm.py                 41
tinygrad/runtime/ops_clang.py                22
tinygrad/runtime/ops_npy.py                   7
```

- These backends replace the CUDA/HIP runtimes and speak directly with the kernel using ioctl.
- Aside from the assembler, no CUDA is used

code walkthrough

Tensor Flow

- Tensor → LazyBuffer (function.py)
  - Forward/backward pass handled here
- LazyBuffer → LazyOp (scheduler.py)
  - Breaking into Kernels here
- LazyOp → UOp (linearizer.py)
  - Generate kernel code in an LLVM-like IR
- UOp → Code (renderer)
  - This code is CUDA code or C code
- Code → /accelerator/ (runtime)

Code: tensor.py:Tensor

```python
class Tensor:
  """
  A `Tensor` is a multi-dimensional matrix containing elements of a single data type.

  ```python exec="true" session="tensor"
  from tinygrad import Tensor, dtypes, nn
  import numpy as np
  import math
  np.set_printoptions(precision=4)
  ```
  """
  __slots__ = "lazydata", "requires_grad", "grad", "_ctx"
  __deletable__ = ('_ctx',)
  training: ClassVar[bool] = False
  no_grad: ClassVar[bool] = False

  def __init__(self, data:Union[None, ConstType, List, Tuple, LazyBuffer, np.ndarray, bytes, MultiLazyBuffer, Variable],
               device:Optional[Union[str, tuple, list]]=None, dtype:Optional[DType]=None, requires_grad:Optional[bool]=None):
    assert dtype is None or isinstance(dtype, DType), f"invalid dtype {dtype}"
    device = tuple(Device.canonicalize(x) for x in device) if isinstance(device, (tuple, list)) else Device.canonicalize(device)

    # tensors can have gradients if you have called .backward
    self.grad: Optional[Tensor] = None

    # NOTE: this can be in three states. False and None: no gradient, True: gradient
    # None (the default) will be updated to True if it's put in an optimizer
    self.requires_grad: Optional[bool] = requires_grad

    # internal variable used for autograd graph construction
    self._ctx: Optional[Function] = None
```

The main class. Methods are the useful functions. Where forward and backward are handled. The lazydata property contains a LazyBuffer

Code: function.py

```python
class Mul(Function):
  def forward(self, x:LazyBuffer, y:LazyBuffer) -> LazyBuffer:
    self.x, self.y = x, y
    return x.e(BinaryOps.MUL, y)

  def backward(self, grad_output:LazyBuffer) -> Tuple[Optional[LazyBuffer], Optional[LazyBuffer]]:
    return self.y.e(BinaryOps.MUL, grad_output) if self.needs_input_grad[0] else None, \
           self.x.e(BinaryOps.MUL, grad_output) if self.needs_input_grad[1] else None
```

Thanks to the chain rule, 28 derivatives are all you need to handcode

Code: lazy.py:LazyBuffer

```python
class LazyBuffer:
  def __init__(self, device:str, st:ShapeTracker, dtype:DType,
               op:Optional[Op]=None, arg:Any=None, srcs:Tuple[LazyBuffer, ...]=(),
               base:Optional[LazyBuffer]=None):
    self.device, self.st, self.dtype, self.shape, self.size = device, st, dtype, st.shape, st.size
    self._base: Optional[LazyBuffer] = None
    if base is None:
      # properties on base
      self.op, self.arg, self.srcs = op, arg, srcs  # this is a LazyOp, except the src is LazyBuffers and not LazyOps
      assert self.op is not LoadOps.ASSIGN or srcs[1].base.realized is not None, "assign target must be realized"

      if (self.op is LoadOps.CONTIGUOUS or self.op is UnaryOps.BITCAST) and srcs[0].st.consecutive and \
          not srcs[0].is_unrealized_const() and device.split(":")[0] in view_supported_devices:
        # some LazyBuffers can be processed with only a view, no AST required
        self.buffer: Buffer = srcs[0].base.buffer.view(st.size, dtype, srcs[0].st.views[0].offset * srcs[0].dtype.itemsize)
        self.op = LoadOps.VIEW
      else:
        self.buffer = srcs[1].base.buffer if self.op is LoadOps.ASSIGN else Buffer(device, self.size, dtype)
      self.buffer.ref(1)
      self.contiguous_child: Optional[Tuple[ReferenceType[LazyBuffer], ShapeTracker]] = None
      self.forced_realize = False
    else:
      # properties on view
      assert base.base == base, "base must be a base itself"
      self._base = base
```

The container of computation, specifies how to construct the buffer.
Below the forward/backward layer, can be constructed from simple ops.

Code: ops.py

```python
# these are the llops your accelerator must implement, along with toCpu
# the Enum class doesn't work with mypy, this is static. sorry it's ugly
# NOTE: MOD, CMPLT don't have to be implemented on vectors, just scalars
# NOTE: many GPUs don't have DIV, but UnaryOps.RECIP doesn't work for integer division
class UnaryOps(Enum):
  """A -> A (elementwise)"""
  EXP2 = auto(); LOG2 = auto(); CAST = auto(); BITCAST = auto(); SIN = auto(); SQRT = auto(); NEG = auto() # noqa: E702
class BinaryOps(Enum):
  """A + A -> A (elementwise)"""
  ADD = auto(); SUB = auto(); MUL = auto(); DIV = auto(); MAX = auto(); MOD = auto(); CMPLT = auto(); CMPNE = auto(); XOR = auto() # noqa: E702
  SHR = auto(); SHL = auto() # noqa: E702
class TernaryOps(Enum):
  """A + A + A -> A (elementwise)"""
  WHERE = auto(); MULACC = auto() # noqa: E702
class ReduceOps(Enum):
  """A -> B (reduce)"""
  SUM = auto(); MAX = auto() # noqa: E702
class BufferOps(Enum): LOAD = auto(); CONST = auto(); STORE = auto() # noqa: E702
class LoadOps(Enum): EMPTY = auto(); CONST = auto(); COPY = auto(); CONTIGUOUS = auto(); CUSTOM = auto(); ASSIGN = auto(); VIEW = auto() # noqa: E702

Op = Union[UnaryOps, BinaryOps, ReduceOps, LoadOps, TernaryOps, BufferOps]
```

The 32 simple ops.

Code: shape/shapetracker.py

```python
@dataclass(frozen=True)
class ShapeTracker:
  views: Tuple[View, ...]
```

```python
def pad(self, arg: Tuple[Tuple[sint, sint], ...]) -> ShapeTracker: return ShapeTracker(self.views[0:-1] + (self.views[-1].pad(arg), ))
def shrink(self, arg: Tuple[Tuple[sint, sint], ...]) -> ShapeTracker: return ShapeTracker(self.views[0:-1] + (self.views[-1].shrink(arg), ))
def expand(self, new_shape: Tuple[sint, ...]) -> ShapeTracker: return ShapeTracker(self.views[0:-1] + (self.views[-1].expand(new_shape), ))
def permute(self, axis: Tuple[int, ...]) -> ShapeTracker: return ShapeTracker(self.views[0:-1] + (self.views[-1].permute(axis), ))
def stride(self, mul: Tuple[int, ...]) -> ShapeTracker: return ShapeTracker(self.views[0:-1] + (self.views[-1].stride(mul), ))

def reshape(self, new_shape: Tuple[sint, ...]) -> ShapeTracker:
  if getenv("MERGE_VIEW", 1) and (new_view := self.views[-1].reshape(new_shape)) is not None: return ShapeTracker(self.views[0:-1] + (new_view,))
  return ShapeTracker(self.views + (View.create(new_shape), ))
```

- One of the pieces of tinygrad magic, all “movement” operations are tracked here.
- Reshape can create “multiview” ShapeTracker, aka the length of the views tuple is > 1

Code: shape/view.py

```python
@dataclass(frozen=True)
class View:
  shape:Tuple[sint, ...]
  strides:Tuple[sint, ...]
  offset:sint
  mask:Optional[Tuple[Tuple[sint, sint], ...]]
  contiguous:bool
```

A view has a shape, strides, an offset, and a mask.

This handles all pad, shrink, expand, permute, and stride + some reshapes.

Throwback: conv2d

```
jesse@x1:~/tinygrad$ DEBUG=3 python3 -c "from tinygrad import Tensor; Tensor.rand(16,3,64,64).conv2d(Tensor.rand(16,3,3,3)).realize()"
CLDevice: got 1 platforms and 1 devices
opened device GPU from pid:91707
opened device NPY from pid:91707
*** CUSTOM     1 custom_random                       arg   1 mem  0.00 GB
*** CUSTOM     2 custom_random                       arg   1 mem  0.00 GB
  0 ┌─ STORE MemBuffer(idx=0, dtype=dtypes.float, st=ShapeTracker(views=(View(shape=(16, 1, 16, 62, 62, 1, 1, 1), strides=(61504, 0, 3844, 62, 1, 0, 0, 0), offset=0, mask=None, contiguous=True),)))
  1 └┬─ SUM (7, 6, 5)
  2  └┬─ MUL
  3   ├── LOAD MemBuffer(idx=1, dtype=dtypes.float, st=ShapeTracker(views=(View(shape=(16, 1, 16, 62, 62, 3, 3, 3), strides=(12288, 0, 0, 64, 1, 4096, 64, 1), offset=0, mask=None, contiguous=False),)))
  4   └── LOAD MemBuffer(idx=2, dtype=dtypes.float, st=ShapeTracker(views=(View(shape=(16, 1, 16, 62, 62, 3, 3, 3), strides=(0, 0, 27, 0, 0, 9, 3, 1), offset=0, mask=None, contiguous=False),)))
*** GPU       3 r_4_31_31_4_2_2_3_4_4_3_3           arg   3 mem  0.00 GB tm    457.29us/     0.46ms (  116.20 GFLOPS,   10.33 GB/s)
avg:   116.20 GFLOPS    10.33 GB/s          total:     3 kernels     0.05 GOPS     0.00 GB     0.46 ms
```

```
0 ┌─ STORE MemBuffer(idx=0, dtype=dtypes.
1 └┬─ SUM (7, 6, 5)
2  └┬─ MUL
3   ├── LOAD MemBuffer(idx=1, dtype=dt
4   └── LOAD MemBuffer(idx=2, dtype=dt
*** GPU       3 r_4_31_31_4_2_2_3_4_4_3_3
```

LOAD, MUL, SUM, STORE are Ops defining a Kernel

```
View(shape=(16, 1, 16, 62, 62, 3, 3, 3), strides=(12288, 0, 0, 64, 1, 4096, 64, 1),
View(shape=(16, 1, 16, 62, 62, 3, 3, 3), strides=(0, 0, 27, 0, 0, 9, 3, 1), offset=0
```

There’s two single view ShapeTrackers for the inputs

the tiny corp

A company in 2024

- We are a GitHub and a Discord.
- We raised $5M, and will be profitable this year by selling computers.
- “remote” jobs are fine, but it begins to deconstruct what a job is.
- We are now 5 people, and hire exclusively from the pool of tinygrad contributors.
- “collective”

Bounties

	Short Description	Value
1	Split UnaryOps.CAST into UnaryOps.CAST and UnaryOps.BITCAST	$100
2	Refactor UOps -> UPat with full regression tests	$200
3	Replace tqdm with <= 5 clean lines of code, tested to match real tqdm char for char	$200
4	Intel XMX Tensor Core Support	$400
5	PTX never worse (1.1x max on kernel, winning overall on all models) than CUDA	$400
6	Taylor approximations for LOG2/EXP2/SIN in function.py passing all tests	$400
7	<10s (wall time) hlb_cifar training on up to 6x 7900XTX	$500
8	Fast mean+stddev fusion into 1 kernel without new ops	$500
9	O(n) arange with uops optimization	$500
10	JIT cache loading/saving, restore thneed behavior and rebase openpilot, https://github.com/tinygrad/tinygrad/issues/3397	$500
11	Buffer offset support (clean!)	$500
12	(mlperf) Training RetinaNet	$600
13	(mlperf) Training Unet3D	$600
14	(mlperf) Training Bert	$600
15	Apple AMX support in LLVM or CLANG	$600
16	Qualcomm Kernel level GPU driver (like NV/AMD) with HCQ graph support	$600
17	>10 tok/s running LLaMA 2 70B in FP16 on a tinybox. loading in <10s	$700
18	RDNA3 assembler within 10% of perf to HIP (see speed_compare_cuda_ptx for compare style)	$700
19	(mlperf) Training Stable Diffusion	$1,000
20	(mlperf) Training llama2 70B lora	$1,000
21	Proof or disproof of the mergeability of two arbitrary ShapeTrackers in Lean (see docs/reshape_without_symbolic.md)	$1,000
22	Kernels support multiple outputs (clean, well tested)	$1,000
23	Beautiful website (stats.tinygrad.org)	$1,000
24	(mlperf) Training Resnet	$1,200
25	Qualcomm DSP support (<= speed as SNPE)	$10,000
26		
27	Uncolored bounties are up for grabs. Lock it by submitting a good WIP PR (stays locked if I see forward progress in last 5 days)	
28	Yellow bounties are locked (only to be claimed by owner)	
29	Green bounties are complete	

tinybox

hardware sales that match the main development platform...

...is ethical value capture

MLPerf

- As promised, we got AMD on MLPerf.
- tinybox green (6x 4090), ResNet-50, 122 minutes
- tinybox red (6x 7900XTX), ResNet-50, 167 minutes
- Done using tinygrad, none of the ML libraries from either company.
- Our next submission will use none of the userspace.

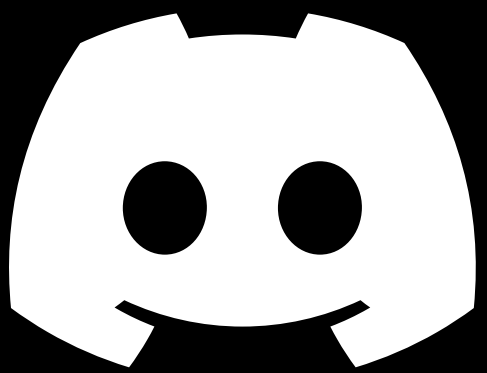
Where we are going

1) Build the best training framework for NVIDIA/AMD/Intel/Qualcomm/etc.

2) Capture all existing chips in a generic framework. Search for the best possible chip given a set of tasks.

3) Build that chip. Sell chips and build clouds at the task abstraction, not the computer abstraction.

How to join tiny



- Permissionless company! (who has read ?s doc)
- Skills are all that matters
- We don’t discriminate against silicon based life

live coding...