

# tinygrad: a single dialect from Tensor programs to Command Buffers

tinygrad, Corp.  
research@tinygrad.org

## UOps

All nodes in the tinygrad graph are **UOps**. A UOp is a tuple (op, src, arg, tag) where op is from the set below, src is a tuple of input UOps, arg is op-dependent, and tag is for temporary processing. The full program is a DAG of UOps. Each UOp has five derived properties — **dtype**, **shape**, **device**, **addrspace**, and **min\_max** — determined by the rules at the end of this document.

### Source Ops — leaf nodes

| Op     | src          | arg                                 | Semantics                                                                                                                   |
|--------|--------------|-------------------------------------|-----------------------------------------------------------------------------------------------------------------------------|
| PARAM  | ( <b>s</b> ) | slot, dtype, device?,<br>addrspace? | Placeholder with shape <b>s</b> . Substituted in FUNCTION.                                                                  |
| BUFFER | ( <b>s</b> ) | slot, dtype, device,<br>addrspace   | Concrete buffer slot with shape <b>s</b> . If device is a tuple, it creates the fully sized buffer across multiple devices. |
| CONST  | ()           | value, dtype                        | A scalar constant with shape ().<br>Form vector consts with STACK                                                           |
| BINARY | ()           | data                                | Raw binary data, has dtype uint8 and shape len( <i>data</i> )                                                               |

**addrspace** is GLOBAL, LOCAL, or REG.

### Movement Ops — no arithmetic; view, indexing, and reinterpretation only

| Op      | src                                                    | arg              | Semantics                                                                            |
|---------|--------------------------------------------------------|------------------|--------------------------------------------------------------------------------------|
| PERMUTE | ( <i>T</i> ,)                                          | axis order $\pi$ | Reorder axes. $\pi = (1, 0)$ is transpose.                                           |
| FLIP    | ( <i>T</i> ,)                                          | bools <b>f</b>   | Reverse along flagged axes.                                                          |
| RESHAPE | ( <i>T</i> , <b>s'</b> )                               | —                | Reinterpret in row-major order. $\prod s_k = \prod s'_k$ .                           |
| EXPAND  | ( <i>T</i> , <b>n</b> )                                | —                | Prepend axes <b>n</b> on the left. Output shape is <b>n</b> + <i>T</i> .shape.       |
| PAD     | ( <i>T</i> , <b>o</b> , <b>s'</b> )                    | —                | Place <i>T</i> at offset $o_k$ in an invalid-filled output of shape $s'_k$ .         |
| SHRINK  | ( <i>T</i> , <b>o</b> , <b>s'</b> )                    | —                | Keep $s'_k$ elements starting at offset $o_k$ per axis. Inverse of PAD.              |
| INDEX   | ( <i>T</i> , $i_0, i_1, \dots$ )                       | —                | Index from left. ()-shaped $i$ removes dim; ( $k$ ,)-shaped makes it $k$ .           |
| STACK   | ( <i>T</i> <sub>0</sub> , <i>T</i> <sub>1</sub> , ...) | —                | Join along a newly created leading axis. All shapes must match.                      |
| BITCAST | ( <i>T</i> ,)                                          | dtype            | Reinterpret storage as target dtype; preserve total bytes.                           |
| UNSHARD | ( <i>T</i> , <i>R</i> )                                | axis <i>a</i>    | Concatenate the shards indexed by RANGE <i>R</i> along <i>a</i> ; <i>R</i> is outer. |

### Reduce Ops — remove axes

| Op     | src                              | arg          | Semantics                                                            |
|--------|----------------------------------|--------------|----------------------------------------------------------------------|
| REDUCE | ( <i>T</i> , $r_0, r_1, \dots$ ) | op, <i>n</i> | Reduce the first <i>n</i> axes of <i>T</i> . Op is ADD, MAX, or MUL. |

**Call Ops** — function abstraction, like the lambda calculus

| Op       | src                        | arg | Semantics                                                           |
|----------|----------------------------|-----|---------------------------------------------------------------------|
| FUNCTION | (body, $a_0, a_1, \dots$ ) | —   | Substitute each PARAM $k$ in TUPLE body with $a_k$ . Gradient-able. |
| CALL     | (body, $a_0, a_1, \dots$ ) | —   | Opaque invocation of a compiled kernel or custom function.          |
| TUPLE    | ( $v_0, v_1, \dots$ )      | —   | Pack values; required as FUNCTION body to return a value.           |
| GETTUPLE | ( $T, \text{idx}$ )        | idx | Extract element at idx from a TUPLE.                                |

**Load Ops** — can change device or addrspace

| Op   | src                | arg               | Semantics                                                                                        |
|------|--------------------|-------------------|--------------------------------------------------------------------------------------------------|
| LOAD | (buf, alt?, gate?) | device, addrspace | Read (pull) from buffer into a new anonymous buffer.<br>Note: this replaces COPY and CONTIGUOUS. |

**Store Ops** — the only op with observable side effects

| Op    | src               | arg | Semantics                                                                                                              |
|-------|-------------------|-----|------------------------------------------------------------------------------------------------------------------------|
| STORE | (buf, val, gate?) | —   | Write (push) val into buf. buf.shape = val.shape.<br>If gate is present, write only when gate is true. Output is void. |

**Ordering Ops** — execution order

| Op     | src                   | arg  | Semantics                                                        |
|--------|-----------------------|------|------------------------------------------------------------------|
| RANGE  | (bound,)              | type | Iterator from 0 to bound.                                        |
| END    | (body, range)         | —    | Close a RANGE loop.                                              |
| AFTER  | (buf, deps...)        | —    | Passthrough of buf; guarantees deps execute first.               |
| GROUP  | ( $u_0, u_1, \dots$ ) | —    | Void no-op that merges multiple STORES into one node, unordered. |
| SINK   | ( $s_0, s_1, \dots$ ) | —    | Collect side effects into a single root node.                    |
| LINEAR | (uops...)             | —    | Linearized (toposorted) instruction sequence.                    |

Assign is STORE followed by AFTER: write the value, then return the buffer with an ordering dependency.

**Elementwise Ops** — all inputs same shape, output same shape, applied per-element

| Arity   | src       | Op                       | Semantics                                                      |
|---------|-----------|--------------------------|----------------------------------------------------------------|
| Unary   | (T,)      | RECIP                    | $1/x$                                                          |
|         |           | TRUNC                    | $\text{trunc}(x)$ : round toward zero.                         |
|         |           | CAST                     | Convert to target dtype (specified in arg).                    |
| Binary  | (A, B)    | ADD, MUL, MAX, MOD, IDIV | $a + b, a \cdot b, \max(a, b), a \bmod b, \lfloor a/b \rfloor$ |
|         |           | CMP<LT, CMP<NE           | $[a < b], [a \neq b]$                                          |
|         |           | XOR, OR, AND, SHR, SHL   | $a \oplus b, a \mid b, a \& b, a \gg b, a \ll b$               |
| Ternary | (P, A, B) | WHERE                    | $A[i]$ if $P[i] \neq 0$ , else $B[i]$                          |

**Decomposed elementwise ops** — defined in terms of the primitives above.

| Op       | Decomposition                          | Semantics          |
|----------|----------------------------------------|--------------------|
| NEG      | MUL( $A, -1$ )                         | $-x$               |
| SUB      | ADD( $A, \text{NEG}(B)$ )              | $a - b$            |
| DIV      | MUL( $A, \text{RECIP}(B)$ )            | $a/b$              |
| CMPGT    | CMPLT( $B, A$ )                        | $[a > b]$          |
| CMPGE    | CMPNE(CMPLT( $A, B$ ), 1)              | $[a \geq b]$       |
| CMPLE    | CMPNE(CMPLT( $B, A$ ), 1)              | $[a \leq b]$       |
| CMPEQ    | CMPNE(CMPNE( $A, B$ ), 1)              | $[a = b]$          |
| NOT      | CMPNE( $A, 1$ )                        | $\neg a$           |
| EXP2     | polynomial approx + MUL, ADD           | $2^x$              |
| LOG2     | exponent extract + polynomial approx   | $\log_2 x$         |
| SIN      | argument reduction + polynomial approx | $\sin x$           |
| SQRT     | EXP2( $0.5 \cdot \text{LOG2}(A)$ )     | $\sqrt{x}$         |
| POW      | EXP2( $\text{LOG2}(A) \cdot B$ )       | $a^b$              |
| MULACC   | ADD(MUL( $A, B$ ), $C$ )               | $a \cdot b + c$    |
| THREEFRY | 5 rounds of add-rotate-xor (ARX)       | Threefry 2x32 PRNG |

### Marker Ops — identity on data

| Op                 | src      | arg | Semantics                          |
|--------------------|----------|-----|------------------------------------|
| CONTIGUOUS         | ( $T$ ,) | —   | Force contiguous memory layout.    |
| CONTIGUOUSBACKWARD | ( $T$ ,) | —   | Force contiguous in backward pass. |
| DETACH             | ( $T$ ,) | —   | Stops gradient propagation.        |

### Codegen Ops — generated code primitives, these do not appear in the main graph

| Op             | src                      | arg    | Semantics                                                                   |
|----------------|--------------------------|--------|-----------------------------------------------------------------------------|
| BARRIER        | (deps...)                | —      | Synchronize threads within a workgroup.                                     |
| INS            | ...                      | ...    | A single machine instruction (e.g. AMD ISA).                                |
| GETADDR        | (buf,)                   | dev    | Lower buf to its address on device dev.                                     |
| SPECIAL        | (bound,)                 | name   | GPU thread/workgroup index (e.g. <code>gidx0</code> , <code>lidx1</code> ). |
| IF             | (gate,)                  | —      | Begin conditional execution block.                                          |
| ENDIF          | (if,)                    | —      | End conditional execution block.                                            |
| WMMA           | ( $A, B, \text{acc}$ )   | config | Warp matrix multiply-accumulate (tensor cores).                             |
| CUSTOM         | (args...)                | fnt    | Inject custom code string into generated source.                            |
| ATOMICADD      | (idx, val)               | —      | Atomic read-modify-write: <code>buf[idx] += val</code> .                    |
| CUSTOMFUNCTION | (meta...)                | name   | Opaque device function (e.g. HW decode). Via <code>CALL</code> .            |
| PROGRAM        | (linear, source, binary) | —      | Compiled kernel: instructions, source, and machine code.                    |
| SOURCE         | ()                       | str    | Human-readable rendered source code.                                        |
| BINARY         | ()                       | bytes  | Compiled machine code.                                                      |

These ops are not part of the core specification and are subject to change.

### Derived Properties

Every UOp has a **dtype**, **shape**, **device**, **addrspace**, and **min\_max**, derived from its op, src, and arg:

| Op             | dtype        | shape                   | device        | min_max                    |
|----------------|--------------|-------------------------|---------------|----------------------------|
| BUFFER         | from arg     | from src[0]             | from arg      | dtype range                |
| CONST          | from arg     | ()                      | NULL          | $[v, v]$                   |
| PARAM          | from arg     | from src[0]             | from arg      | from src or dtype range    |
| Movement ops   | src[0].dtype | (see op)                | src[0].device | src[0]                     |
| UNSHARD        | src[0].dtype | src[0], axis $\times n$ | src[0].device | src[0]                     |
| REDUCE         | src[0].dtype | remove first $n$ axes   | src[0].device | dtype range                |
| CAST           | from arg     | src[0].shape            | src[0].device | clamped to dtype           |
| BITCAST        | from arg     | src[0].shape            | src[0].device | dtype range                |
| COPY           | src[0].dtype | src[0].shape            | from arg      | src[0]                     |
| ALU unary      | src[0].dtype | src[0].shape            | src[0].device | dtype range                |
| ADD            | src[0].dtype | broadcast               | src[0].device | $[a + b, A + B]$           |
| MUL            | src[0].dtype | broadcast               | src[0].device | [min, max] of products     |
| MAX            | src[0].dtype | broadcast               | src[0].device | $[\max(a, b), \max(A, B)]$ |
| Other binary   | src[0].dtype | broadcast               | src[0].device | dtype range                |
| CMPLT, CMPNE   | bool         | broadcast               | src[0].device | from intervals             |
| WHERE          | src[1].dtype | broadcast               | src[0].device | $[\min(b, c), \max(B, C)]$ |
| FUNCTION, CALL | src[0].dtype | substitute PARAM shapes | src[1].device | dtype range                |
| RANGE          | index        | ()                      | NULL          | $[0, n-1]$                 |
| INDEX          | src[0].dtype | remaining dims          | src[0].device | src[0]                     |
| STORE          | void         | ()                      | src[0].device | —                          |
| AFTER          | src[0].dtype | src[0].shape            | src[0].device | src[0]                     |

broadcast: right-align shapes, element-wise max; each axis must be equal or 1.  $[a, A]$ ,  $[b, B]$ ,  $[c, C]$  denote min\_max of src[0], src[1], src[2]. Default *dtype range*: [dtype\_min, dtype\_max].

**axis** tracks the multi-device sharding dimension. UNSHARD defines it (axis = arg). BUFFER with  $n$ -tuple device: axis = 0 (device dim). RESHAPE remaps axis to preserve the shard boundary. PERMUTE follows the permutation. EXPAND shifts axis right by  $|\mathbf{n}|$ . REDUCE on the shard axis  $\rightarrow$  NULL (shard axis is among the first  $n$  axes). REPLICATED on the shard axis  $\rightarrow$  NULL. COPY  $\rightarrow$  NULL. ALU ops inherit from sources. Default: NULL.

## Kernel Optimizations (OptOps) — schedule-level transforms on kernel ranges

Each kernel's iteration space is a set of RANGE axes. Every range has an **AxisType**:

| AxisType     | Letter | Split from      | Direction   | Semantics                                  |
|--------------|--------|-----------------|-------------|--------------------------------------------|
| DEVICE       | d      | —               | —           | Multi-device sharding dimension.           |
| GLOBAL       | g      | —               | —           | GPU global workgroup dimension.            |
| LOCAL        | l      | g, L            | inner       | Workgroup local dimension (shared memory). |
| WARP         | w      | (created by TC) | —           | Warp-level lanes for tensor cores.         |
| THREAD       | t      | g               | outer       | CPU thread parallelism.                    |
| LOOP         | L      | —               | —           | Generic sequential loop (initial state).   |
| REDUCE       | R      | —               | —           | Reduction axis.                            |
| GROUP_REDUCE | G      | R               | inner/outer | Shared-memory group reduction.             |
| UPCAST       | u      | g, l, L         | inner       | Register-level vectorization.              |
| UNROLL       | r      | R, G            | inner       | Fully unrolled loop.                       |

An optimization is a triple (op, axis, arg):

| OptOp    | axis            | arg                         | Semantics                                                                                                        |
|----------|-----------------|-----------------------------|------------------------------------------------------------------------------------------------------------------|
| SPLIT    | any             | (factor $k$ , target, top?) | Split axis $n$ by $k$ into $(n/k, k)$ or $(k, n/k)$ if top. New sub-axis gets target AxisType (see table above). |
| PADTO    | any             | multiple $m$                | Pad axis to next multiple of $m$ with validity masks.                                                            |
| SWAP     | $\text{axis}_i$ | $\text{axis}_j$             | Swap two axes $i \leftrightarrow j$ .                                                                            |
| NOLOCALS | —               | —                           | Disable local memory; no workgroup dims emitted.                                                                 |
| TC       | reduce idx      | (tc, opt, mode)             | Apply tensor core WMMA: split reduce/output axes into WARP, UPCAST, and UNROLL dims.                             |

Optimizations compose left-to-right. TC must be first. The search space is explored by BEAM search or hand-coded heuristics.

## Common Ops as Compositions

All high-level tensor operations decompose into the primitives above.

```
# gemm:  $C[M,N] = A[M,K] @ B[K,N]$ 
def gemm(A, B):
    M,K = A.shape; _,N = B.shape
    return (A.reshape(M,K,1) * B.reshape(1,K,N)).sum(1)

# prefix_sum: cumulative sum via repeat+reshape sliding window trick
def prefix_sum(T):
    n = T.shape[0]
    x = T.pad((n-1, 0)) # (2n-1,)
    x = x.reshape(1,2*n-1).expand(n+1,2*n-1) # tile
    x = x.reshape((n+1)*(2*n-1)).shrink_to(2*n*n) # trim
    x = x.reshape(n,2*n).shrink_to(n,n) # windows
    return x.sum(-1) # reduce

# arange: prefix_sum of all 1s gives [1,2,...,n], subtract 1 for [0,1,...,n-1]
def arange(n):
    return prefix_sum(Tensor(1).reshape(1).expand(n)) - 1

# gather:  $\text{out}[i] = T[\text{idx}[i]]$ . one-hot mask along gather axis, then reduce
def gather(T, idx):
    K = T.shape[0]
    pos = arange(K).reshape(K, 1) # (K, 1)
    mask = (pos == idx.reshape(1, -1)).cast(T.dtype) # (K, D)
    return (T.reshape(K, 1) * mask).sum(0) # (D,)

# scatter_add:  $T[\text{idx}[i]] += \text{val}[i]$ 
def scatter_add(T, idx, val):
    K, D = T.shape[0], idx.shape[0]
    pos = arange(K).reshape(K, 1) # (K, 1)
    mask = (pos == idx.reshape(1, D)).cast(T.dtype) # (K, D)
    return T + (mask * val.reshape(1, D)).sum(1) # (K,)
```

## Multi-Device Collectives — derived from primitives

Let  $D = (d_0, \dots, d_{n-1})$  be an  $n$ -tuple device. COPY to an  $n$ -tuple device reshards with axis = 0. COPY never changes shape.

**Sharding** splits a tensor along an axis across  $n$  devices. It opens a RANGE of type DEVICE (a symbolic per-device index  $d$ ), shrinks each device's view to its piece, then closes the range with UNSHARD( $T, R, a$ ). The result is a logical tensor whose shape along axis  $a$  is the full size; each device holds  $1/n$  of it. UNSHARD is the inverse of sharding — it marks the boundary between per-device computation and the logical multi-device tensor. The range need not be

DEVICE; e.g. a WARP range closes the same way, concatenating per-lane shards along  $a$  with the range as the outer factor.

```
# T has shape (s,) on a single device.

# broadcast: replicate T to all n devices
def broadcast(T):
    return T.reshape(1, s).expand(n, s).copy(D).replicated(0)      # (s,) on D, axis=null

# scatter: split T into n chunks, one per device
def scatter(T):
    return T.copy(D)                                                # (s,) on D, axis=0

# T has shape (n*s,) on D with axis=0, so each device holds (s,) elements.

# gather: collect all shards onto one device
def gather(T):
    return T.copy(D[0])                                             # (n*s,) on D[0], axis=null

# reduce: gather + sum
def reduce(T):
    return gather(T).reshape(n, s).sum(0)                          # (s,) on D[0], axis=null

# allgather: collect all shards, replicate to all devices
def allgather(T):
    return T.reshape(1, n*s).expand(n, n*s).copy(D).replicated(0)  # (n*s,) on D, axis=null

# reduce_scatter: reduce across devices, scatter result
def reduce_scatter(T):
    return T.reshape(n, n, s//n).permute(1, 0, 2).copy(D).sum(1).reshape(s)  # (s,) on D, axis=0

# allreduce: reduce_scatter + allgather
def allreduce(T):
    return allgather(reduce_scatter(T))                            # (s,) on D, axis=null
```

## The @function Decorator — graph capture via tracing

The @function decorator transforms a Python function on Tensors into a single FUNCTION node.

```
@function
def f(a: Tensor, b: Tensor) -> Tensor:
    return a + b
```

When  $f(x, y)$  is called, the decorator:

1. **Extracts inputs:** walks all arguments to find every Tensor, deduplicates by identity.
2. **Runs the function lazily** (no device execution), building a UOp graph from the result.
3. **Parameterizes:** replaces each input UOp with a  $\text{PARAM}(k)$  placeholder.
4. **Wraps the body** in a  $\text{TUPLE}$  (even for single returns) and creates  $\text{FUNCTION}(\text{TUPLE}(\text{body}), x, y)$ .
5. **Returns** the result via  $\text{GETTUPLE}(0)$ , or one  $\text{GETTUPLE}$  per element for tuple returns.

The result is a reusable graph fragment: the body contains only  $\text{PARAM}$  references, not concrete buffers. At schedule time, the  $\text{FUNCTION}$  is resolved by substituting each  $\text{PARAM}(k)$  back with its corresponding argument  $a_k$ , or lowered into an opaque  $\text{CALL}$  if it is to be compiled as a reusable kernel.

## Lowering Pipeline — from Tensor graph to machine code

| Stage                        | Semantics                                                                                                |
|------------------------------|----------------------------------------------------------------------------------------------------------|
| <b>Callify</b>               | Transform the Tensor graph into a single stateless function.                                             |
| <b>Rangeify</b>              | Determine the kernel split of the function. Break everything down to shape ()                            |
| <b>Optimize</b>              | Insert local buffers. Swap and split ranges, and determine which axes are parallel and which are serial. |
| <b>Expand</b>                | Expand the parallel ranges into shape.                                                                   |
| <b>Instruction Selection</b> | Select target instructions, including WMMA and devectorization.                                          |
| <b>Linearize</b>             | Topologically sort the graph and determine execution order.                                              |
| <b>Register/Memory Plan</b>  | Allocate and reuse GLOBAL, LOCAL, and REG storage for values with non-overlapping lifetimes.             |
| <b>Render</b>                | Output the machine code.                                                                                 |