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
BUFFER	(n)	`ParamArg`	Persistent storage on device.
ALLOC	(n)	`ParamArg`	Unbound storage of n elements.
PARAM	(n)	`ParamArg`	Caller-provided storage.
CONST	()	value	A scalar constant with shape ( ).
BINARY	()	data	Raw binary data, has dtype uint8 and shape len($data$)

`ParamArg` contains slot, dtype, value bounds, alignment, name, addrspace, device, volatility, optional image shape, and an optional bound device buffer (absent for ALLOCs). **addrspace** is `GLOBAL`, `LOCAL`, or `REG`. The flat size n is stated as a concrete weakint CONST in src (an empty STACK for a scalar), followed by a `DEVICE` RANGE for multi-device storage.

Call Ops — function abstraction, like the lambda calculus

Op	src	arg	Semantics
CALL	(body, a_0, a_1, …)	—	Substitute each PARAM k in body with a_k.

A value CALL is void: its SINK body stores to output PARAMs bound positionally to the call's ALLOC arguments; output a_k is AFTER(a_k, CALL). ALLOCs are scoped to their CALL: they are never implicit inputs of the enclosing graph.

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

Op	src	arg	Semantics
PERMUTE	$(T,)$	axis order π	Reorder axes. $\pi = (1, 0)$ is transpose.
RESHAPE	$(T, \mathbf{s}')$	—	Reinterpret in row-major order. $\prod s_k = \prod s'_k$.
EXPAND	$(T, \mathbf{n})$	—	Prepend axes $\mathbf{n}$ on the left. Output shape is $\mathbf{n} + T$.shape.
PAD	$(T, \mathbf{o}, \mathbf{s}')$	—	Place T at offset o_k in an invalid-filled output of shape s'_k.
SHRINK	$(T, \mathbf{o}, \mathbf{s}')$	—	Keep s'_k elements starting at offset o_k per axis. Inverse of PAD.
INDEX	$(T, i_0, i_1, \ldots)$	—	Index from left. ()-shaped i removes dim; $(k,)$-shaped makes it k.
STACK	$(T_0, T_1, \ldots)$	—	Join along a newly created leading axis. All shapes must match.
BITCAST	$(T,)$	dtype	Reinterpret storage as target dtype; preserve total bytes.
UNSHARD	$(T, R_0, R_1, \ldots)$	axes $(a_0, a_1, \ldots)$	Concatenate shards of RANGE R_k along axis a_k; R_k is outer.

Reduce Ops — remove axes

Op	src	arg	Semantics
REDUCE	(T, r_0, r_1, …)	op, n	Reduce the first n axes of T. Op is ADD, MAX, or MUL.

Load Ops — can change device or addrspace, anonymous store

Op	src	arg	Semantics
LOAD	(buf, alt?, gate?)	—	Read from buffer into AddrSpace.ALU.
COPY	(buf,)	device	Copy buf to device arg.
STAGE	$(T, r_0, r_1, \ldots)$	`BufferizeOpts?`	Materialize T into storage. Ranges become leading dimensions.

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. 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, $range_0$, $range_1$, …)	—	Close one or more RANGE loop.
BACKEDGE	(body, range, condition)	—	Conditional loop.
AFTER	(buf, deps…)	—	Passthrough of buf; guarantees deps execute first.
SINK	$(s_0, s_1, \ldots)$	—	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	trunc(x): round toward zero.
		CAST	Convert to target dtype (specified in arg).
Binary	(A, B)	ADD, MUL, MAX	$a+b$, $a \cdot b$, $\max(a,b)$
		FLOORDIV, FLOORMOD	$\lfloor a/b \rfloor$, $a - b\lfloor a/b \rfloor$
		CMPLT, CMPNE	$[a < b]$, $[a \neq b]$
		XOR, OR, AND, SHR, SHL	$a \oplus b$, $a \mid b$, $a \,\&\, b$, $a \gg b$, $a \ll b$
Ternary	(P, A, B)	WHERE	$A[\mathbf{i}]$ if $P[\mathbf{i}] \neq 0$, else $B[\mathbf{i}]$

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

Op	Decomposition	Semantics
NEG	MUL(A, -1)	$-x$
SUB	ADD(A, NEG(B))	$a - b$
DIV	MUL(A, RECIP(B))	a/b
CDIV	$q + c$	Integer division toward zero.
CMOD	$r - bc$	Remainder with dividend's sign (or zero).
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· LOG2(A))	$\sqrt{x}$
POW	EXP2(LOG2(A) · 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
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. `gidx0`, `lidx1`).
IF	(gate,)	—	Begin conditional execution block.
ENDIF	(if,)	—	End conditional execution block.
WMMA	(A, B, acc)	config	Warp matrix multiply-accumulate (tensor cores).
CUSTOM	(args…)	fmt	Inject custom code string into generated source.
ATOMICADD	(idx, val)	—	Atomic read-modify-write: `buf[idx] += val`.
CUSTOMFUNCTION	(meta…)	name	Opaque device function (e.g. HW decode). Via CALL.
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, ALLOC	from arg	(n) from src, or ()	from arg	dtype range
PARAM	from arg	(n) from src, or ()	from arg	from src or dtype range
CONST	from arg	()	NULL	$[v, v]$
Movement ops	src[0].dtype	(see op)	src[0].device	src[0]
UNSHARD	src[0].dtype	src[0], each $a_k \times n_k$	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	$[\min(b,c),\ \max(B,C)]$
CALL	void	—	first non-null src device	—
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].

sharding tracks multi-device sharding as a set of (axis, RANGE) pairs. UNSHARD defines it: arg is the tuple of sharded axes, one RANGE in src per axis (positional: the k-th RANGE shards the k-th axis). BUFFER, ALLOC, or PARAM with n-tuple device: sharded on axis 0 (device dim). The single-axis convenience **axis** is NULL unless exactly one axis is sharded. RESHAPE remaps each sharded axis to preserve its shard boundary. PERMUTE follows the permutation. EXPAND shifts all sharded axes right by $|\mathbf{n}|$. REDUCE on a sharded axis drops it. REPLICATED on the shard axis → NULL. COPY → 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/outer	Workgroup local dimension (shared memory).
WARP	w	(created by TC)		Warp-level lanes for tensor cores.
WEAK	L	—	—	Optimizable range (default).
LOOP	L	—	—	Pinned sequential loop.
UPCAST	u	g, l, L	inner/outer	Vectorization, including reduction unrolling.

Reduction axes are identified by the ranges closed by REDUCE, not by a separate AxisType.

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	axis_i	axis_j	Swap two axes $i \leftrightarrow j$.
TC	reduce idx	(tc, opt, mode)	Apply tensor core WMMA: split reduce/output axes into `WARP` and `UPCAST` 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: out[i] = T[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[idx[i]] += 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 $D = (d_0, \ldots, d_{n-1})$ 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. A tensor may be sharded along several axes at once: UNSHARD$(T, R_0, R_1, \ldots;\ a_0, a_1, \ldots)$ carries one RANGE per sharded axis, and every movement op maps each sharded axis independently.

```
# 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 CALL 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 each returned value.
3. **Parameterizes inputs**: replaces each input UOp with a positional PARAM(k) placeholder.
4. **Parameterizes outputs**: for each returned value v_i, creates an output PARAM$(m+i)$ and a matching ALLOC b_i (unique symbolic storage identity), where m is the number of inputs.
5. **Builds the call**: stores every v_i into its output parameter and creates CALL(SINK(STORE(PARAM(m), v_0), …), x, y, b_0, …).
6. **Returns values**: exposes each result as AFTER$(b_i$, CALL$)$.

The result is a reusable graph fragment: the body's storage is represented by PARAM references and local ALLOCs, not concrete buffers, and a single call can return any number of values. At schedule time, an ordinary value-producing CALL is inlined by positional PARAM substitution and each output AFTER resolves to the value stored in the body, preserving required output writes. A precompiled call keeps the body opaque and binds its ALLOCs to concrete buffers for execution.

Lowering Pipeline — from Tensor graph to machine code

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