BendRT: A Parallel Runtime for CPUs and GPUs

Victor Taelin
Higher Order Company
Rio de Janeiro, Brazil
taelin@higherorderco.com

AI Disclosure. Bend and BendRT were designed by the human author. This paper was written by Claude Fable 5.1 from the author's code and design notes, and reviewed by the author. Human paper soon™.

Abstract

BendRT is the runtime of Bend, a pure functional language with an affine dependent type system. The compiler emits one C file per program; that file runs the same code on one thread, on CPU threads, and on the GPU through Metal or CUDA, over one heap, with no garbage collector and no user-written kernels. Three ideas carry the design. Affinity comes from the checker: a value has one owner, so a match frees the node it consumed, a forked task owns its arguments, and sharing exists only where the compiler placed a counted share or a borrow. The evaluator is flat: every function is a case tree inside one worklist function with no C call stack, and every call site reads two ways, sequential and parallel, so one text serves every executor. Tasks live in a *cube*, a fixed 128 by 128 grid of rings that forks along its rows in a grow phase and drains along its columns in a work phase, with no shared queue, no lock and no work stealing. The price is a contract: the program's forks must split their work evenly. On an Apple M4 Max (pin of 2026-08-31) the sequential build runs within 0.8 to 1.5 times the time of hand-written C, sixteen threads give 9 to 12 times over one, and the integrated GPU up to 67 times.

1 Introduction

Bend's type system, developed in a companion paper [13], is affine: a live value is consumed at most once unless its binder is marked `+`, and a `+` binder forms only at the kind `Data`, which excludes functions, arrays and handles. The theory paper argues that this wall makes the language consistent. This paper argues that it also makes the language fast. The costs that runtimes for functional languages pay most for all come from not knowing who owns a value, and affinity settles ownership at compile time. Other runtimes need a garbage collector because nobody knows when a value's last owner leaves; in Bend the one owner is the match that consumes it, so deallocation is compiled code. They need work stealing because tasks are cheap to make and hard to place; in Bend a task owns its arguments, so a fork moves memory and never shares it, and the only cross-thread protocol is delivering an answer into a join. They stay off the GPU, which wants flat memory, no recursion and no runtime; Bend's evaluator needs no collector, and with one discipline on calls no C stack either, so the C file is also the shader.

We follow one program from source to C, to tasks, to the GPU dispatch that runs it. Readers of the author's earlier runtimes may expect interaction nets [7, 12] here; there are none (Section 11).

2 The Program and Its Contract

```
import Base

type Tree is Type:
  Leaf{x: U32}
  Node{l: Tree, r: Tree}

def build(+d: Nat, +i: U32) -> Tree:
  match d:
    case 0n:
      Leaf{i}
    case 1n+p:
      l r = build(p, i * 2) build(p, i * 2 + 1)
      Node{l, r}

def sum(t: Tree) -> U32:
  match t:
    case Leaf{x}:
      x
    case Node{l, r}:
      a b = sum(l) sum(r)
      a + b

def main() -> IO(Unit):
  t = build(20n, 0)
  IO.print(U32.show(sum!(t)))
```

The program builds a tree of 2^{20} leaves and sums it. Two marks carry all the parallelism Bend has. The *parallel let* `l r = f(x) g(y)` binds n names to n calls in one statement and means: run these calls in parallel; they split the work into roughly equal parts. The *mark* `sum!(t)` means: run this call on the GPU. Nothing else creates parallelism, and the runtime trusts the equal-parts promise absolutely (Section 6).

The checker hands the compiler more than types. A match consumes its scrutinee, always a parameter or a field of one, never a computed value. A def body is flattened at parse into a *case tree*, lambdas and constructor matches over leaves, and validation annotates every node with its type; erased binders (`-`), types and proofs never reach the emitted code. One law governs what follows: a compiler may drop a copy the source spelled, never add one.

3 Compilation

3.1 Tail, Cut, Fork

The runtime has no C call stack for user code, so every call must sit where a jump can replace it. The first pass rewrites each reachable body until every call has one of three shapes and all else is call-free expression code. A *tail* is a saturated call in return position. A *cut* is one call whose result is named; its continuation is minted as a sequential definition `f$k`. A *fork* is a let of two or more calls; its continuation is minted as a joiner `f$j`. Lambdas lift into definitions `f$c` over their captures, so a closure is a definition applied to all but one of its arguments.

Two whole-program analyses then decide the memory traffic. *Share inference* finds the constructor types that wear reference counts: a type is *hot* when some binder of it is used more than once, when a hot type's live field reaches it, or when it is boxed in an array; everything else is built and consumed through plain stores and loads. *Borrow inference* finds the arguments a callee only reads: each live argument of a datatype starts borrowed and flips to owned on any escape, to a fixpoint. A borrowed argument is lent raw and read in place, so a fold over a reused tree costs no count traffic. A small fork-free callee is inlined by running the checker's own evaluator on the call.

3.2 The Segment

Each definition becomes one *segment* of the worklist function. Listing 1 shows what the emitter prints for `sum`.

Three things happen here. The match takes ownership: `Tree` is not hot, so the arm reads both fields and frees the node with two stores onto a free list. A `Leaf` never touched memory: a constructor with one word-sized field is packed into the term word. And the fork is read twice. In the parallel world the arm mints a *join task* with one empty slot per child, spawns each child pointing back at its slot, and returns the join as its *reply* for the scheduler to deal out (Section 6). In the sequential world it pushes a frame and runs the first call in place; each step pushes the next frame, and the last jumps into the joiner with both results in the bank.

The emitted file is the runtime template, the program's tables and segments, and the C source of every effect it imports; `--threads`, `--parallel` and `--gpu` pick the executor at run time.

```
WL_CASE(FID_SUM) {
  Term t = r0;
  WL_SPIN
  if (term_aux(t) == CID_LEAF) {
    WL_RET(term_loc(t));  // packed: no node
  } else {
    u64 nd = term_loc(t);
    Term l = e.mem[nd + 0];
    Term r = e.mem[nd + 1];
    heap_free(e, cls_fit(2), nd);  // owned
    if (!seq) {  // parallel: a join, two kids
      u64 j = task_node(e, FID_SUM_J,
        WL_CONT, WL_IDX, 2);
      Term jt = term_tsk(FID_SUM_J, j);
      u64 k0 = task_node(e, FID_SUM, jt, 0, 0);
      e.mem[k0] = l;  WL_KID(j, 0, FID_SUM, k0);
      u64 k1 = task_node(e, FID_SUM, jt, 1, 0);
      e.mem[k1] = r;  WL_KID(j, 1, FID_SUM, k1);
      return jt;
    }
    STK(0) = r;  STK(1) = FID_SUM_S_1;
    WL_PUSHN(2);
    t = l;  WL_AGAIN;  // sequential: in place
  }
  WL_SPUN
}
WL_CASE(FID_SUM_S_1) {  // after sum(l)
  Term r = STK(-1);
  STK(0) = r0;  STK(1) = FID_SUM_S_0;
  WL_PUSHN(2);
  r0 = r;  WL_JMP(FID_SUM);
}
WL_CASE(FID_SUM_S_0) {  // after sum(r)
  WL_POPN(2);
  r1 = r0;  r0 = STK(1);
  WL_JMP(FID_SUM_J);
}
WL_CASE(FID_SUM_J) {
  WL_RET(U32_BIN(r0, +, r1));
}
```

Listing 1. The segments of `sum`, locals renamed and unboxing elided.

4 Memory

4.1 Terms and the Corpus

A term is one 64-bit word: a tag, a 16-bit aux (constructor or segment id) and a 40-bit heap location. A machine word, a `Nat` below 2^{48} and a constructor with at most one word-sized field are *packed* into the word itself and never touch the heap; a constructor, a closure, a task, and a flat *block* (an array) point at a node. `HOLE`, all ones, marks an undelivered task slot.

Every executor shares one flat array of words, the *corpus*: a header, the 2^{14} task rings of Section 6, the lane stacks, and the heap. The host reserves 8 TB and pages fault in on touch; the GPU fixes its span before the first dispatch (Section 7). Both processors address the same words. Each lane allocates from its own free lists and claims fresh pages off one global counter, so an allocation is a pop or a bump, a free is two stores, and neither crosses a thread.

4.2 Ownership at Run Time

Generated code moves values. Three operations cover the cases where it cannot. *Take* consumes a value at a match: read the fields, free the node. *Keep* gives a value a second owner and is emitted exactly at a binder's second use: the first keep mints a *redirect*, one word holding the node's address and a count, and turns the local into a pointer at it; the count lives there, never on the node, so an unshared value carries no count anywhere. *Drop* releases an owner: a packed term costs nothing, a counted pointer decrements and only the last one collects, a sole owner collects at once.

Take respects sharing: at count one it collapses the redirect and owns the node; above one it copies the fields out and releases its pointer. Every decrement is a release and whoever sees zero acquires first, so a field read never races a free. Only hot types pay any of this, and they seal every stored field with a count at build time.

Drop is the runtime's only collector: an iterative walk that threads its worklist through the nodes being freed, so it allocates nothing and runs from any code on either processor. There is no tracing, no epoch and no deep copy anywhere: a `+` variable is emitted at every use, and each extra use is the share or the borrow the compiler placed.

4.3 Arrays

In the source, `Array<T>` is a perfect binary tree walked by index bits. The backends run it as one flat block: packed 32-bit cells for an array of machine words, one owned term per word otherwise. A read is an indexed load, a write an indexed store, and a read of a boxed cell shares it. Affinity makes the store sound: `Array<T>` is `Type`, never `Data`, so an array has one owner, nobody can observe the mutation, and a copy is explicit. A match on the tree takes both halves in place, and joining two adjacent halves back is free.

5 The Machine

On the host every segment is a `preserve_none` function of the machine's words, entered by a `musttail` call (a dynamic jump goes through a table of them); on the device the segments are the cases of one switch inside an error-polling loop. The state is a bank of words `r0..rn` (a segment's parameters, or the results it receives), the current segment id, a value stack, and the world flag `seq`.

The function takes one task. Its prologue pushes the task's continuation, slot index and the exit segment onto the value stack, loads the arguments into the bank, frees the task node, and jumps. A segment ends in one of four ways. A tail call loads the bank and jumps; a self call reloads its own parameters and loops. A cut, in the sequential world, pushes a frame holding the continuation's captures and segment id and jumps to the callee; in the parallel world it mints a continuation task expecting one result, makes it the current continuation, and jumps. A fork was shown above. A return puts the value in the bank and pops the stack: the popped word is the segment to enter next, a step or a minted continuation, which reads its captures from the stack and its result from the bank. The exit segment pops the task's continuation and delivers the bank into it; a marked call in the parallel world returns a spawned task instead of jumping.

Recursion depth is bounded by the value stack alone: a guarded 2 GB mapping per host worker, a fixed window per device lane, and the same numbered error on overflow.

ring 0	t0	t4	t8	
ring 1	t1	t5	t9	
ring 2	t2	t6	w	
ring 3	t3	t7		

Figure 1. The cube, shown 4 lanes wide. Tasks pushed along a row during one phase are drained down the columns of the next: the flip swaps the index map, an $O(1)$ transposition that moves no task. A task born during a work pass (`w`, darker) lands past the snapshot and belongs to the next round.

6 The Task Cube

6.1 Tasks are Terms

A task is a heap term like any other: a node of arity plus two words, `[args..., cont, idx|rem]`, tagged with the segment to enter. A join's argument slots hold `HOLE` until its children answer; `cont` is the parent continuation task, or `HOLE` at the root; `idx` says which slot of the parent this task's own answer fills, and `rem` how many children still owe one. Delivery writes the slot and decrements `rem` with release order; the lane that takes it to zero owns the runnable join and runs it. Children of one join write disjoint slots and race only on the countdown. A `HOLE` continuation is the root: its delivery is the program's answer.

6.2 The Cube

The frontier lives in the *cube*: 2^{14} rings arranged as a 128 by 128 square, one ring per lane, the same on every target; a single thread serves the whole square when parallelism is off. A ring is a fixed FIFO of 1024 slots with one producer step, a fetch-add and two stores, and one consumer per phase, the lane that owns it. The square is stored slot-major, so a row and a column are both $O(1)$ index maps, and *flipping* the cube, turning the rows one phase filled into the columns the next phase drains, swaps the indexer and moves no task (Figure 1).

6.3 Grow and Work

The driver reads one number per round, the *frontier* f: how many tasks were pushed into the cube during the last round. While $f < 2^{14}$ it runs a grow phase; it always runs a work phase; it stops when the root has delivered.

Grow widens the frontier one fork step at a time. Each of the 128 rows is served by one worker, a thread on the host or a threadgroup on the device. It pops the head of each ring in its row, runs it in the parallel world, and pushes what comes back into the row: a fork reply deals its children round-robin across the row's rings, and a runnable reply, a completed continuation or a spawned marked call, is pushed back whole. Tasks whose segment can never fork are skipped: they cannot widen the frontier, and they wait for work. A row stops growing when every ring in it has work or a sweep grew nothing.

Work runs at saturation. Every lane drains its column ring, down to a snapshot of the put counter taken at the turn's start, in the sequential world: forks run as consecu-

tive calls on the value stack and cuts as frames, and most of a program's time is spent here, in straight-line code. A join completed by the lane's last delivery runs at once, in the parallel world, so the forks it meets are dealt into the cube through the global cursor and counted toward the next round's frontier. This is the bulk-synchronous rhythm [16]: bursts of exponential fork, then deep sequential focus, then the next wave.

The host pool is up to 128 threads, which claim rows by one fetch-add and meet at one barrier per turn; it opens at a program's first fork. The GPU runs a grow as a dispatch of 128 groups (seeded by one group while $f < 128$) and a work as one thread per lane; between dispatches the host only reads f.

6.4 Why Nothing Contends

Count what crosses a lane. A push is one fetch-add and two stores. A delivery is one release fetch-sub on the join's own node. A page claim is one fetch-add on the global cursor, once per quantum. That is all: no shared deque, no lock on the task path, no migration, no rebalance, and no task is ever re-read to be placed elsewhere. Every phase is a closed step, which is what lets the same phase bodies run as persistent host threads and as device dispatches.

The price is written into the language. Work stealing [2, 4] repairs an unequal split at run time; the cube does not. If a program forks unequal parts, lanes idle at the end of a work turn, and the runtime declines to correct that: balance is the program's job. The idioms are teachable: fork equal halves of the data or the index space, sequence full-width phases instead of forking phases against each other, keep light work out of forks.

7 The GPU

The runtime speaks two device APIs, Metal and CUDA, and the device code is not a port: the binary carries *its own source text*, the build compiles the segments a `!` can reach into a device program beside the binary (a Metal binary archive of the pipeline, a CUDA cubin behind a hash of the text), and a launch loads it, or compiles it once when the file is missing or stale, so host and device run the same functions by construction.

The span is decided once, before the first dispatch: `--gpu-memory`, else 2 GB on Metal, where a buffer cannot grow under a running kernel, and the whole card on CUDA, whose managed pages fault in on demand. Metal wraps the host mapping in one zero-copy buffer at the same addresses. A device cannot abort, so failure is a protocol: the first failing lane compare-and-swaps its error into the header word, every long loop polls that word and drains, and the host reads it after the dispatch, prints one line and exits.

The mark `f!(x)` is honored by the event loop (Section 8) at a sequential program point, where nothing else runs. The solo thread detaches the marked call whole, makes its continuation the root, seeds it into ring zero and runs the phase loop on the device until the root delivers; then it hands the answer into the original continuation and continues on the CPU. Met anywhere else the mark degrades: in the parallel world it spawns a task rather than nesting, in the sequential world it is inert, and without a device it is inert everywhere. The CPU and the GPU never compute at the same time. Floats follow one semantics on every executor (contraction off, safe math on the device), and the harness checks that all three print the same bytes.

8 Effects

`IO(A)` is a definition of the base library, not a primitive: a continuation over one datatype `IO.OP` with two constructors, `Emit` and `Halt`. A *foreign fill*, a def whose body is `import` lines naming a `.c` and a `.js` file, compiles to one segment that packs its arguments and its continuation into a request. The event loop runs on one thread: it evaluates `main` to a request through the pure machine, runs the effect's C handler, applies the continuation to the answer, and evaluates again; `Emit` ends the program and `Halt` exits with its code. Effects never run inside an evaluator, and the machine performs no IO. A fallible operation answers a `Result` carrying errno, and a handle threads back outside the `Result`, so even a failure cannot lose it.

9 The JavaScript Backend

The same rewritten definitions (Section 3) print as plain JavaScript over the host's garbage collector: one function per live definition, constructors as tagged objects; tail calls trampoline, forks run in sequence, and a request unwinds to the loop as a thrown value. `bend file.bend` runs this backend in memory, and a loader makes a `.bend` file a module under node and Bun, so a Bend program is also a JavaScript library.

10 Results

Table 1 supports three readings. The sequential build runs within 0.8 to 1.5 times the time of its C twin: affinity buys the memory story at no sequential cost. Sixteen threads give 8.8 to 12.1 times over one: the suite forks equal halves, and each wave ends together. The GPU is bimodal. Uniform work reaches 52 to 67 times the sequential build (game of life, n-body, mandelbrot, merkle), while divergent and skewed work loses to sixteen threads (n-queens, symbolic regression), and the design stance is to report the loss rather than tune the scheduler toward it.

11 Related Work

Interaction nets. HVM [7, 12] evaluates interaction combinators: optimal sharing and parallelism at every redex. BendRT keeps the goals and drops the mechanism: affinity gives unique ownership directly, so the graph and its duplication machinery become unnecessary. Lost is optimal reduction of shared redexes; gained are native-speed sequential code, flat memory and a cost model a programmer can read.

bench	1 thread	16 threads	GPU	C twin
tree bitonic sort	6.18 s	0.62 s	0.43 s	5.48 s
game of life	5.46 s	0.48 s	0.08 s	6.82 s
k-means	2.26 s	0.26 s	0.18 s	2.03 s
mandelbrot	4.78 s	0.40 s	0.08 s	3.79 s
tree matmul	3.50 s	0.31 s	0.22 s	2.89 s
merkle tree	4.30 s	0.39 s	0.08 s	3.48 s
n-body	4.89 s	0.43 s	0.08 s	4.99 s
n-queens	5.51 s	0.46 s	1.24 s	4.61 s
tree radix sort	3.89 s	0.36 s	0.25 s	2.67 s
raytrace	6.15 s	0.53 s	0.21 s	4.67 s
symbolic regr.	3.31 s	0.29 s	0.54 s	2.22 s
terrain	2.81 s	0.25 s	0.13 s	2.19 s

Table 1. The pinned suite, `bench/runtime/`: seconds on one Apple M4 Max (pin of 2026-08-31, commit `64fc4b7`) for one thread, sixteen threads, the integrated GPU through Metal, and the hand-written C twin, one C function per Bend definition. Each cell is a warm run then a timed run on an idle machine, and every executor must print the pinned checksum.

Scheduling. Work stealing [2, 4] is the standard answer to irregular parallelism, and its absence here is the scheduling novelty: the balance obligation moves into the language contract and the runtime stays wave-structured [16], which is what lets one scheduler run on a GPU at all. Lazy task creation [10] anticipates the work phase: a fork met while draining runs as a plain call. GHC's sparks [8] and Multilisp's futures [5] are hints like the parallel let, but back onto stealing pools and a collected heap.

Functional GPU compilation. Futhark [6], Accelerate [3] and NESL [1] flatten array combinators into kernels. Bend's unit of GPU execution is the whole language, recursion, allocation and algebraic data included, run by a scheduler that lives on the device; the price is younger, less specialized device code, visible in the losses of Table 1.

Memory. Regions [14] and linear types [17] aimed at collector-free functional memory; Rust [9] made ownership with static borrows mainstream; Lean [15] and Perceus [11] count precisely and reuse in place. Bend differs in where counts live and when they exist: contraction exists only at `Data`, so counts are minted only where a whole-program analysis finds sharing, in a redirect word beside the node, and the unshared majority of a program compiles to moves, borrows and frees with no count traffic.

12 Limitations

The equal-parts contract is unverified: a skewed program silently loses its parallelism. The CPU and the GPU never compute together, and a mark is honored only at a sequential point. A ring holds 1024 tasks, a count saturates at 2^{24}, a natural at 2^{48}, a block at depth 31; each overflow is a numbered fail-stop, not a fallback. The Metal lane is measured; the CUDA lane is in the source and not measured here. The C runtime is unverified: the correctness argument is the checksum discipline plus the type system's guarantees, and the companion paper's theorems stop at the calculus.

13 Conclusion

BendRT compiles an affine functional language to one C file and runs it unchanged from a single thread to a GPU. Affinity places the frees, the flat evaluator gives every executor its unit of work, and the cube forks and drains a fixed grid of rings without contending. The benchmarks show both sides of the bargain: near-C sequential speed and large parallel wins when the fork contract holds, and honest losses on divergent work.

References

[1] Guy E. Blelloch. 1996. Programming Parallel Algorithms. *Communications of the ACM* 39, 3 (1996), 85–97.

[2] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *Journal of the ACM* 46, 5 (1999), 720–748.

[3] Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell Array Codes with Multicore GPUs. In *Declarative Aspects of Multicore Programming (DAMP)*, 2011. ACM, 3–14.

[4] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *Programming Language Design and Implementation (PLDI)*, 1998. ACM, 212–223.

[5] Robert H. Halstead. 1985. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems* 7, 4 (1985), 501–538.

[6] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-Programming with Nested Parallelism and In-Place Array Updates. In *Programming Language Design and Implementation (PLDI)*, 2017. ACM, 556–571.

[7] Yves Lafont. 1997. Interaction Combinators. *Information and Computation* 137, 1 (1997), 69–101.

[8] Simon Marlow, Simon Peyton Jones, and Satnam Singh. 2009. Runtime Support for Multicore Haskell. In *International Conference on Functional Programming (ICFP)*, 2009. ACM, 65–78.

[9] Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust Language. In *High Integrity Language Technology (HILT)*, 2014. ACM, 103–104.

[10] Eric Mohr, David A. Kranz, and Robert H. Halstead. 1990. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. In *LISP and Functional Programming (LFP)*, 1990. ACM, 185–197.

[11] Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. 2021. Perceus: Garbage Free Reference Counting with Reuse. In *Programming Language Design and Implementation (PLDI)*, 2021. ACM, 96–111.

[12] Victor Taelin. 2024. *HVM2: A Parallel Evaluator for Interaction Combinators.* Retrieved from https://github.com/HigherOrderCO/HVM2/blob/main/paper/HVM2.pdf

[13] Victor Taelin. 2026. *BendTT: An Affine Dependent Type Theory.*

[14] Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Information and Computation* 132, 2 (1997), 109–176.

[15] Sebastian Ullrich and Leonardo de Moura. 2019. Counting Immutable Beans: Reference Counting Optimized for Purely Functional Programming. In *Implementation and Application of Functional Languages (IFL)*, 2019. ACM.

[16] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Communications of the ACM* 33, 8 (1990), 103–111.

[17] Philip Wadler. 1990. Linear Types Can Change the World. In *Programming Concepts and Methods*, 1990. North-Holland, 561–581.