

# HVM2: A PARALLEL EVALUATOR FOR INTERACTION COMBINATORS

VICTOR TAE LIN

**ABSTRACT.** We present HVM2, an efficient, massively parallel evaluator for extended interaction combinators. When compiling non-sequential programs from a high-level programming language to C and CUDA, we achieved a near-ideal parallel speedup as a function of cores available, scaling from 400 million interactions per second (MIPS) (Apple M3 Max; single thread), to 5,200 MIPS (Apple M3 Max; 16 threads), to 74,000 MIPS (NVIDIA RTX 4090; 32,768 threads). In this paper, we describe HVM2’s architecture, present snippets of the reference implementation in Rust, share early benchmarks and experimental results, and discuss current limitations and future plans.

**This paper is a work in progress. It will be continuously updated on HVM’s repository ([github.com/HigherOrderCO/HVM](https://github.com/HigherOrderCO/HVM)) as it is written.**

## 1. INTRODUCTION

Interaction Combinators (IC) were introduced by Lafont (Lafont 1997) as a minimal and concurrent model of computation. Lafont proved that IC’s were not only Turing Complete, but that they also preserve the complexity class and degree of parallelism. Lafont argued that while Turing Machines are a universal model of sequential computation, IC’s are a universal model of *distributed* computation. The locality and strong confluence of Lafont’s ICs make it suitable for massive parallel computation. This heavily implied IC’s are an optimal model of computation, in a very fundamental sense. Yet, it remained to be seen if this system could be implemented efficiently in practice.

In this paper, we answer this question positively. By storing Interaction Combinator nodes in a memory-efficient format, we’re able to implement its core operations (annihilation, commutation, and erasure) as lightweight C procedures and CUDA kernels. Furthermore, by representing wires as atomic variables, we’re able to perform interactions atomically, in a lock-free fashion and with minimal synchronization overhead. We also extend our system with global definitions (for fast function applications) and native numbers (for fast numeric operations). The result, HVM2, is an efficient, massively parallel evaluator for ICs that achieves near-ideal speedup up to at least 16,384 concurrent cores, peaking at 74 billion interactions per second on an NVIDIA RTX 4090.

This level of performance makes it compelling to propose HVM2 as a general framework for parallel computing. By translating constructs such as functions, algebraic data types, pattern matching, and recursion to HVM2, we see it as a potential compilation target for modern programming languages such as Python and Haskell. As a demonstration of this possibility, we also introduce Bend, a high-level programming language that compiles to HVM2. We explain how some of these translations work, and set up a general framework to translate arbitrary languages, procedural or functional, to HVM2.

## 2. SIMILAR WORKS

**Work In Progress**

## 3. SYNTAX

HVM2's syntax consists of an IC system with eight node types. Nodes form a tree-like structure. They are represented syntactically as:

```

<Tree> ::=
| <name>                -- (VAR)iable
| "*"                  -- (ERA)ser
| "@" <name>            -- (REF)erence
| <number>              -- (NUM)ber
| "(" <Tree> <Tree> ")" -- (CON)structor
| "{" <Tree> <Tree> "}" -- (DUP)licator
| "$(" <Tree> <Tree> ")" -- (OPE)rator
| "?(" <Tree> <Tree> ")" -- (SWI)tch

```

The first 4 node types (VAR, ERA, REF, NUM) are nullary, and the last 4 types (CON, DUP, OPE, SWI) are binary. As in Lafont's Interaction Nets, every node has an extra distinguished edge, called the main or principal port (Lafont 1997). Thus, nullary nodes have one port (one main and zero auxiliary), while binary nodes have three ports (one main and two auxiliary). With the syntax above, the main port of the root node is “free”, as it is not wired to another port. We can connect two main ports and form a reducible expression (redex), using the following syntax:

```

<Redex> ::= <Tree> "~" <Tree>

```

An HVM2 Net consists of a root tree, plus a (possibly empty) list of &-separated redexes:

```

<Net> ::= <Tree> ("&" <Redex>)*

```

Thus, HVM2 Nets contain only a single free main port. Wirings are possible between trees through pairs of VAR nodes with the same names.

Lastly, an HVM2 Book consists of a list of top-level definitions, or, “named” Nets:

```

<Book> ::= ("@" <name> "=" <Net>)*

```

Each .hvm2 file contains a book, which is executed by HVM2. The entry point for HVM2 programs is the @main definition.

## 3.1. An Example.

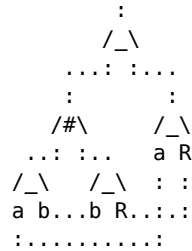
The following definition:

```

@succ = ({(a b) (b R)}) (a R)

```

Represents the HVM2 encoding of the Lambda Calculus term  $\lambda s \lambda z (s (s n))$ , and can be drawn as the following Interaction Combinator net:



Notice how `CON/DUP` nodes in HVM2 correspond directly to constructor and duplicator nodes in Lafont’s Interaction Combinators (Lafont 1997). Aux-to-main wires are implicit through the tree-like structure of the syntax, while aux-to-aux wires are explicit through variable nodes, which are always paired.

Additionally, main ports being implicit is critical to storing nodes efficiently in memory. Nodes can be represented as just two ports (the HVM2 memory model for wires) rather than three. In HVM2, since every port is 32 bits, this allows us to store a single node in a 64-bit word. This compact representation lets us use built-in atomic operations in various parts of the code, which was key to making parallel C and CUDA versions efficient. For details on the precise memory representation, see Section 7.

### 3.2. Interpretation of the Syntax.

Semantically, `CON`, `DUP`, and `ERA` nodes correspond accordingly to Lafont’s *constructor*, *duplicator*, and *eraser* symbols (Lafont 1997), and behave like Mazza’s Symmetric Interaction Combinators (Mazza 2007). The `VAR` node represents a wiring in the graph, connecting two ports of a Net. They are linear and paired in the sense that (except for the free main port) every port is connected to exactly one other port, and therefore each variable occurs exactly twice.

`REF` nodes are an extension to Lafont’s IC’s, and they represent an immutable net that is expanded in a single interaction. While not essential for the expressivity of the system, `REF` nodes are essential for performance, as they enable fast global functions, a degree of laziness in a strict setup (critical to making GPU implementations viable), and allow us to represent tail recursion in constant space.

`NUM`, `OPE` and `SWI` nodes are also not essential expressivity-wise, but are too important for performance reasons. Modern processors are equipped with native machine integer operations. Emulating these operations with IC constructs analogous to Church Numerals would be very inefficient, thus, these numeric nodes are necessary for HVM2 to be efficient in practice.

## 4. INTERACTIONS

The AST above specifies HVM2’s data format. As a virtual machine, it also provides a mechanism to compute with that data. In traditional VMs, these are called **instructions**. In term rewriting systems, there are usually **reductions**. In HVM2, the mechanism for computation is called **interactions**. There are ten of them. All interactions are listed below, using Gentzen-style rules (the line above reduces to the line below).

$$\begin{array}{ll}
(\text{LINK}) \frac{A \sim B}{\text{link}(A, B)} & (\text{CALL}) \frac{@\text{foo} \sim B}{\text{expand}(@\text{foo}) \sim B} \\
(\text{ERASE}) \frac{* \sim *}{* \sim *} & (\text{VOID}) \frac{* \sim (B1 \ B2)}{* \sim B1} \\
& \quad \quad \quad * \sim B2 \\
(\text{COMMUTE}) \frac{(A1 \ A2) \sim \{B1 \ B2\}}{\begin{array}{l} \{x \ y\} \sim A1 \\ \{z \ w\} \sim A2 \\ (x \ z) \sim B1 \\ (y \ w) \sim B2 \end{array}} & (\text{ANNIHILATE}) \frac{(A1 \ A2) \sim (B1 \ B2)}{\begin{array}{l} A1 \sim B1 \\ A2 \sim B2 \end{array}} \\
(\text{OPERATE } 1) \frac{\#A \sim \$(\#B \ B2)}{\#OP(A, B) \sim B2} & (\text{SWITCH } 1) \frac{\#0 \sim ?(B1 \ B2)}{B1 \sim (B2 \ *)} \\
(\text{OPERATE } 2)^1 \frac{\#A \sim \$ (B1 \ B2)}{B1 \sim \$(\#A \ B2)} & (\text{SWITCH } 2) \frac{\#A+1 \sim ?(B1 \ B2)}{B1 \sim (* \ (\#A \ B2))}
\end{array}$$

Note that these rules are *symmetric*: if a rule applies to a redex  $A \sim B$  then it also applies to  $B \sim A$ . Detailed explanations of each of the rules follow.

4.1. **Link.** “Links” two ports  $A$  and  $B$ . If either  $A$  or  $B$  are **VAR** nodes, a **global substitution** is formed. If neither is a **VAR** node, a **new redex** is created. In pure IC semantics, linking two ports isn’t technically an interaction, and it doesn’t count as such. Yet, for code simplicity, HVM2 allows variables to occur on redexes, forming a temporary **LINK** interaction that must be handled.

4.2. **Call.** Expands a **REF**, replacing it with its definition. The definition is copied, with fresh **VARs**.

4.3. **Erase.** Erases a binary node  $(B1 \ B2)$  connected to an nullary node, propagating the nullary node towards both nodes  $B1$  and  $B2$ . The rule performs a granular, parallel garbage collection of nets that go out of scope.

When the nullary node is a **NUM** or a **REF**, the **ERASE** rule actually behaves as a copy operation, cloning the **NUM** or **REF**, and connecting to both ports. *However*, when a copy operation is applied to a **REF** which contains **DUP** nodes, it instead is computed as a normal **CALL** operation. This allows us to perform fast copy of “function pointers”, while still preserving Interaction Combinator semantics.

---

<sup>1</sup>OPERATE 1 takes precedence over OPERATE 2 if both can be applied.

4.4. **Void.** Erases two nullary nodes connected to each other. The result is nothing: both nodes are consumed, fully cleared from memory. The **VOID** rule completes a garbage collection process.

4.5. **Commute.** Commutes two binary nodes of different types, essentially cloning them. The **COMMUTE** rule can be used to clone data and to perform loops and recursion, although these are preferably done via **CALLS** (preferably in what sense?).

4.6. **Annihilate.** Annihilates two binary nodes of the same type connected to each-other, replacing them with two redexes. The **ANNIHILATE** rule is the most essential computation rule, and is used to implement beta-reduction and pattern-matching.

4.7. **Operate.** Performs a numeric operation between two **NUM** nodes **#A** and **#B** connected by an **OPE** node. If **B1** is not a **NUM**, it is partially applied instead. There is only one binary operation interaction. Dispatching to different native numeric operations depends on the numbers themselves. See Section 6.1 for details.

Note that when counting the number interactions, **OPERATE 2** is not counted, as this would cause the number of interactions to be non-deterministic.

4.8. **Switch.** Performs a switch on a **NUM** node **#A** connected to a **SWI** node, treating it like a **Nat ::= Zero | (Succ pred)**. Here, **B1** is expected to be a tuple (first reference of the term “tuple”, it’s unclear what the encoding is) with both cases: **zero** and **succ**, and **B2** is the return port. If **A** is 0, we return the **zero** case, and erase the **succ** case. Otherwise, we return the **succ** case applied to **A-1**, and erase the **zero** case.

4.9. **Interaction Table.** Since there are eight node types, there is a total of 64 possible pairwise node interactions. The table below shows which interaction rule is triggered for each possible pair of nodes that form a redex. This table is symmetric across the diagonal. **CON-SWI being COMM is problematic; should be removed or the reduction for SWIT should change**

A \ B	VAR	REF	ERA	NUM	CON	DUP	OPR	SWI
VAR	LINK	CALL	LINK	LINK	LINK	LINK	LINK	LINK
REF	CALL	VOID	VOID	VOID	CALL	ERAS	CALL	CALL
ERA	LINK	VOID	VOID	VOID	ERAS	ERAS	ERAS	ERAS
NUM	LINK	VOID	VOID	VOID	ERAS	ERAS	OPER	SWIT
CON	LINK	CALL	ERAS	ERAS	ANNI	COMM	COMM	COMM
DUP	LINK	ERAS	ERAS	ERAS	COMM	ANNI	COMM	COMM
OPR	LINK	CALL	ERAS	OPER	COMM	COMM	ANNI	COMM
SWI	LINK	CALL	ERAS	SWIT	COMM	COMM	COMM	ANNI

```
# Attempts to link A and B.
def link(subst: Dict[str, Node], A: Node, B: Mode):
    while True:
        # If A is not a VAR: swap A and B, and continue.
        if type(A) != VAR:
            swap(A, B)

        # If A is not a VAR: both are non-vars. Create a new redex.
        if type(A) != VAR:
            push_redex(A, B)

        # Here, A is a VAR. Create a `A: B` entry in the map.
        got: Port = subst.set_atomic(A, B)

        # If there was no `A` entry, stop.
        if got is None:
            break

        # Otherwise, delete `A` and link `got` to `B`.
        del subst[A]
        A = got
```

To see how this algorithm works, let's consider, again, the scenario above:

```

      Thread_0      Thread_1
--a--|\_____/|--c--|\_____/--e--
--b--|/\_____|--d--|/\_____|--f--

```

Assume we start with a substitution **a: #42**, and let both threads compute a redex in parallel. Since both redexes are an ANNI rule, their effect is to link both ports; thus, the resulting net (focusing on the **a** wire) should be:

```

#42-----,-----e
          |-----|

```

That is, **e** must be directly linked to **#42**. Let's now evaluate the algorithm in an arbitrary order, step-by-step. Recall that the initial Net is:

```

& (a b) ~ (d c)
& (c d) ~ (f e)

```

And we're observing ports **a**, **d**, and **e**. Two links therefore must be performed: **link(a, d)** and **link(d, e)**. There are many possible orders of execution:

#### 5.1. Possible Execution Order 1.

```

- a: #42
===== Thread_2: link(d, e)
- a: #42
- d: e
===== Thread_1: link(a, d)
- a: d
- d: e
===== Thread_1: got `a: #42`, thus, delete `a` and link(d, #42)
- d: #42
===== Thread_1: got `d: e`, thus, delete `d` and link(e, #42)
- e: #42

```

The resulting substitution map is linking **e** to **42**, as required.

#### 5.2. Possible Execution Order 2.

```

- a: #42
===== Thread_1: link(d, a)
- a: #42
- d: a
===== Thread_2: link(d, e)
- a: #42
- d: e
===== Thread_2: got `d: a`, thus, delete `d` and link(a, e)
- a: e
===== Thread_2: got `a: #42`, thus, delete `a` and link(e, #42)
- e: 42

```

The resulting substitution map is linking, again, **e** to **42**, as required.



### 5.3. Possible Execution Order 3.

```

- a: #42
===== Thread_1: link(d, a)
- a: #42
- d: a
===== Thread_2: link(e, d)
- a: #42
- d: a
- e: d

```

In this case, the result isn't directly linking `e` to `#42`. But it does link `e` to `d`, which links to `a`, which links to `#42`. Thus, `e` is, indirectly, linked to `#42`. While it does temporarily use more memory in this case, it is, semantically, the same result. Additionally, the indirect links will be cleared as soon as `e` is linked to something else. It is easy enough to see that this holds for all possible evaluation orders.

## 6. NUMBERS

HVM2 has a built-in support for 32-bit numerics represented by the `NUM` node type. Numbers in HVM2 have a 5-bit tag and a 24-bit payload. Depending on the tag, numbers can represent unsigned integers (U24), signed integers (I24), IEEE 754 binary32 floats (F24), or partially applied operators. These choices mean any number can be represented in 29 bits, which can be unboxed in a 32-bit pointer with a 3 bit tag for the node type.

### 6.1. Number Tags.

There are three number tags that represent types:

tag	SYM syntax
U24	[u24]
I24	[i24]
F24	[f24]

And fifteen that represent operations:

tag	SYM syntax
ADD	[+]
SUB	[-]
MUL	[*]
DIV	[/]
REM	[%]
EQ	[=]
NEQ	[!]
LT	[<]
GT	[>]
AND	[&]
OR	[ ]
XOR	[^]
FLIP-SUB	[: -]
FLIP-DIV	[: /]
FLIP-REM	[: %]

Finally, there is the **SYM** tag, which is treated specially and is used to cast between tags.

### 6.2. The **SYM** Operation.

The **SYM** operation is special — its payload represents another numeric tag, and when it is applied to another number, it combines the stored numeric tag with the other payload, effectively casting the other number to a different numeric tag. For example, **[+]** is a **SYM** number with payload **ADD**, and when it operates on **1** (a U24 number with payload **0x000001**), it outputs **[+1]** (an **ADD** number with payload **0x000001**).

### 6.3. U24 - Unsigned 24-bit Integer.

U24 numbers represent unsigned integers from 0 to 16,777,215 ( $2^{24} - 1$ ).

The 24-bit payload directly encodes the integer value. For example:

```
0000 0000 0000 0000 0000 0001 = 1
0000 0000 0000 0000 0000 0010 = 2
1111 1111 1111 1111 1111 1111 = 16,777,215
```

### 6.4. I24 - Signed 24-bit Integer.

I24 numbers represent signed integers from  $-8,388,608$  to  $8,388,607$ .

The 24-bit payload uses two's complement encoding. For example:

```
0000 0000 0000 0000 0000 0000 = 0
0000 0000 0000 0000 0000 0001 = 1
0111 1111 1111 1111 1111 1111 = 8,388,607
1000 0000 0000 0000 0000 0000 = -8,388,608
1111 1111 1111 1111 1111 1111 = -1
```

### 6.5. F24 - 24-bit IEEE 754 binary32 Float.

F24 numbers represent a subset of IEEE 754 binary32 floating point numbers.

The 24-bit payload is laid out as follows:

SEEE EEEE EMMM MMMM MMMM MMMM

Where:

- **S** is the sign bit (1 = negative, 0 = positive)
- **E** is the 7-bit exponent, with a bias of 63
- **M** is the 16-bit significand precision

The value is calculated as:

- If **E** = 0 and **M** = 0, the value is signed zero
- If **E** = 0 and **M**  $\neq$  0, the value is a subnormal number:  
 $-1^S \cdot 2^{-62} \cdot 0.M$
- If  $0 < \mathbf{E} < 127$ , the value is a normal number:  
 $-1^S \cdot 2^{E-63} \cdot 1.M$
- If **E** = 127 and **M** = 0, the value is signed infinity
- If **E** = 127 and **M**  $\neq$  0, the value is NaN (Not a Number)

F24 supports a range of about  $\pm 3.4 \times 10^{38}$ . The smallest positive normal number is  $2^{-62} \approx 2.2 \times 10^{-19}$ . Subnormal F24 numbers go down to about  $1.4 \times 10^{-45}$ .

### 6.6. Number Operations.

When two **NUM** nodes are connected by an **OPE** node, a numeric operation is performed. The operation to be performed depends on the tags of each number.

Some operations are invalid, and simply return zero:

- If both number tags represent types, the result is zero.
- If both number tags represent operations, the result is zero.
- If both number tags are **SYM**, the result is zero.

Otherwise:

- If one of the tags is **SYM**, the output has the tag represented by the **SYM** number and the payload of the other operand. For example:

```
OP([+], 10) = [+10]
OP(-1, [*]) = [*0xffffffff]
```

- If one of the tags is an operation, and the other is a type, a native operation is performed, according to the following table:

	U24	I24	F24
ADD	+	+	+
SUB	-	-	-
MUL	*	*	*
DIV	/	/	/
REM	%	%	%
EQ	==	==	==
NEQ	!=	!=	!=
LT	<	<	<
GT	>	>	>
AND	&	&	atan2
OR			log
XOR	^	^	pow

The result type is the same as the input type, except for comparison operators (**EQ**, **NEQ**, **LT**, **GT**) which always return U24 0 or 1.

The number tagged with the operation is the left operand of the native operation, and the number tagged with the type is the right operand.

Note that this means that the number type used in an operation is always determined by the right operand; if the left operand is of a different type, its bits will be reinterpreted.

Finally, flipped operations (such as **FLIP-SUB**) interpret their operands in the opposite order (e.g. **SUB** represents **a-b** whereas **FLIP-SUB** represents **b-a**). This allows representing e.g. both **1 - x** and **x - 1** with partially-applied operations (**[ -1]** and **[ : -1]** respectively).

```
OP([-2], +1) = +1
OP([:-2], 1) = -1
```

Note that **OP** is a symmetric function (since the order of the operands is determined by their tags). This makes the “swap” pseudo-interaction in **OPER** valid.

## 7. THE 32-BIT ARCHITECTURE

The initial version of HVM2, as implemented in this paper, is based on a 32-bit architecture. In this section, we'll use snippets of the Rust reference implementation of the interpreter, available at the [HVM2](#) repository, to document this architecture.

### 7.1. Memory Layout.

Ports are 32-bit values that represent a wire connected to a main port. The low 3-bits are reserved to identify the type of the node (VAR, REF, ERA, etc) whose main port the wire is connected to. This is called a Tag. The upper 29-bits hold the value. The interpretation of this value is dependent on the tag. It is either an address (for CON, DUP, OPR, and SWI nodes), a virtual function address (for REF nodes), an unboxed 29-bit number (for NUM nodes), or a variable name (for VAR nodes), or 0 (for ERA nodes). Nodes are represented in memory as a pair of two ports. Notice that the type of a node (VAR, REF, ERA, etc.) is stored in the port. That is, ports store the kind of node they are connecting to.

```
pub type Tag = u8; // 3 bits (rounded up to u8)
pub type Val = u32; // 29 bits (rounded up to u32)

pub struct Port(pub Val); // Tag + Val (32 bits)
pub struct Pair(pub u64); // Port + Port (64 bits)
```

The Global Net structure includes three shared memory buffers: `node`: a node buffer where nodes are allocated, `vars`: a buffer representing the substitution map (with the 29-bit key being the variable name, and the value being the current substitution), and `rbag`: a collection of active redexes. `APair` and `APort` are atomic variants of `Pair` and `Port`.

```
pub struct GNet<'a> {
    pub node: &'a mut [APair],
    pub vars: &'a mut [APort],
    pub rbag: &'a mut [APair],
}
```

Since this 32-bit architecture has 29-bit values, that means we can address a total of  $2^{29}$  nodes and variables, making the `node` buffer at most 4 GB long, and the `vars` buffer at most 2 GB long. A 64-bit architecture would increase this limit to match the overwhelming majority of use cases, and will be incorporated in a future revision of the HVM2 runtime.

Since top-level definitions are just static nets, they are stored in a similar structure, with the key difference that they include an explicit `root` port, which, is used to connect the expanded definition its target port in the graph. We also include a `safe` flag, which indicates whether this definition has duplicator nodes or not. This affects the DUP-REF interaction, which will just copy the REF port, rather than expanding the definition, when it is `safe`.

```
pub struct Def {
    pub safe: bool, // has no dups
    pub root: Port, // root port
    pub rbag: Vec<Pair>, // def redex bag
    pub node: Vec<Pair>, // def node buffer
}
```

Finally, the global Book is just a map of names to defs:

```
pub struct Book {
    pub defs: Vec<Def>,
}
```

This concludes HVM2's memory layout. For more details, check the reference Rust implementation in the [HVM2](#) repository.

### 7.2. Example Net.

Consider, again, the following net:

```
(a b)
& (b a) ~ (x (y *))
& {y x} ~ @foo
```

In HVM2's memory, it would be represented as:

RBAG	FST-TREE	SND-TREE
----	-----	-----
0800	CON 0001	CON 0002 // '& (b a) ~ (x (y *))'
1800	DUP 0005	REF 0000 // '& {x y} ~ @foo'
----	-----	-----
NODE	PORT-1	PORT-2
----	-----	-----
0001	VAR 0000	VAR 0001 // '(a b)' node (root)
0002	VAR 0001	VAR 0000 // '(b a)' node
0003	VAR 0002	CON 0004 // '(x (y *))' node
0004	VAR 0003	DUP 0000 // '(y *)' node
0005	VAR 0003	VAR 0002 // '{y x}' node
----	-----	-----
VARS	VALUE	
----	-----	
FFFF	CON 0001	// points to root node

Note that the **VARS** buffers has only one entry, because there are no substitutions, but we always use the last variable to represent the root port, serving as an entry point to the graph.

### 7.3. Example Interaction.

Interactions can be implemented in five steps:

1. Allocate the needed resources.
2. Loads nodes from global memory to registers.
3. Initialize fresh variables on the substitution map.
4. Stores fresh nodes on the node buffer.
5. Atomically links outgoing wires.

For example, the **COMMUTE** interaction is implemented in Rust as:

```
pub fn interact_comm(&mut self, net: &GNet, a: Port, b: Port) -> bool {
    // Allocates needed resources.
    if !self.get_resources(net, 4, 4, 4) {
        return false;
    }
}
```

```

// Loads nodes from global memory.
let a_ = net.node_take(a.get_val() as usize);
let a1 = a_.get_fst();
let a2 = a_.get_snd();
let b_ = net.node_take(b.get_val() as usize);
let b1 = b_.get_fst();
let b2 = b_.get_snd();

// Stores new vars.
net.vars_create(self.v0, NONE);
net.vars_create(self.v1, NONE);
net.vars_create(self.v2, NONE);
net.vars_create(self.v3, NONE);

// Stores new nodes.
net.node_create(self.n0, pair(port(VAR, self.v0), port(VAR, self.v1)));
net.node_create(self.n1, pair(port(VAR, self.v2), port(VAR, self.v3)));
net.node_create(self.n2, pair(port(VAR, self.v0), port(VAR, self.v2)));
net.node_create(self.n3, pair(port(VAR, self.v1), port(VAR, self.v3)));

// Links.
self.link_pair(net, pair(port(b.get_tag(), self.n0), a1));
self.link_pair(net, pair(port(b.get_tag(), self.n1), a2));
self.link_pair(net, pair(port(a.get_tag(), self.n2), b1));
self.link_pair(net, pair(port(a.get_tag(), self.n3), b2));

return true;
}

```

Note that, other than the linking, all operations here are local. Taking nodes from global memory is safe, because the thread that holds a redex implicitly owns both trees it contains, and storing vars and nodes is safe, because these spaces have been allocated by the thread. A fast concurrent allocator for small values is assumed. In HVM2, we just use a simple linear bump allocator, which is fast and fragmentation-free in a context where all allocations are at most 64-bit values (the size of a single node).

## 8. MASSIVELY PARALLEL EVALUATION

Provided the architecture we just constructed, evaluating an HVM2 program in parallel is surprisingly easy: **just compute global redexes concurrently, until there is no more work to do.**

HVM2's local interactions exposes the original program's full degree of parallelism, ensuring that every work that **can** be done in parallel **will** be done in parallel. In other words, it maximizes the theoretical speedup, per Amdahl's law. The atomic linking procedure ensures that points of synchronization that emerge from the original program are solved safely and efficiently, without no room for race conditions. Finally, the strong confluence property ensures that the total work done is independent of the order that redexes are computed, giving us freedom to evaluate in parallel without generating extra work.

### 8.1. Redex Sharing.

The last question, thus, is: how do we actually distribute that workload through all cores of a modern processor? The act of sharing a redex is, itself, a point of synchronization. If this is done without enough caution, it can result in contention, and slowing up execution. HVM2 solves this by two different approaches:

On CPUs, a simple task-stealing queue is used, where each thread pushes and pops from its own local redex bag, while a starving neighbor thread actively attempt to steal a redex from it. Since a redex is just a 64-bit value, stealing can be done with a single `atomic_exchange` operation, making it very lightweight. To reduce contention, and to force threads to steal “old redexes”, which are more likely to produce long independent workloads, this stealing is done from the other end of the bag. In our experiences, this works extremely well in practice, achieving full CPU occupancy in all cases tested, with minimal overhead, and low impact on non-parallelizable programs.

On GPUs, this matter is more complex in many ways. First, there are two scales on which we want sharing to occur:

1. Within a running block, where stealing between local threads can be accomplished by fast shared-memory operations and warp-sync primitives.
2. Across global blocks, where sharing requires either a global synchronization (i.e., calling the kernel again) or direct communication via global memory.

Unfortunately, the cost of global synchronization (i.e., across blocks) is very high, so, having a globally shared redex bag, as in the C version, and accessing it within the context of a kernel, would greatly impact performance. To improve this, we, initially, attempted to implement a fast block-wise scheduler, which simply lets local threads pass redexes to starving ones with warp syncs:

```
__device__ void share_redexes(TM* tm) {
    __shared__ Pair pool[TPB];
    Pair send, rcv;
    u32* ini = &tm->rbag.lo_ini;
    u32* end = &tm->rbag.lo_end;
    Pair* bag = tm->rbag.lo_buf;
    for (u32 off = 1; off < 32; off *= 2) {
        send = (*end - *ini) > 1 ? bag[*ini%RLEN] : 0;
        rcv = __shfl_xor_sync(__activemask(), send, off);
        if (!send && rcv) bag[( (*end)++)%RLEN] = rcv;
        if ( send && !rcv) ++(*ini);
    }
    for (u32 off = 32; off < TPB; off *= 2) {
        u32 a = TID();
        u32 b = a ^ off;
        send = (*end - *ini) > 1 ? bag[*ini%RLEN] : 0;
        pool[a] = send;
        __syncthreads();
        rcv = pool[b];
        if (!send && rcv) bag[( (*end)++)%RLEN] = rcv;
        if ( send && !rcv) ++(*ini);
    }
}
```

Such procedure is efficient enough to be called between every few interactions, allowing redexes to quickly fill the whole block. With that, all we had to do is let the kernel perform a constant number of local interactions (usually in the range of  $2^9$  to  $2^{13}$ ), and, once it completes, i.e., across kernel invocations, the global redex bag was transposed (rows become columns), letting the entire GPU to fill naturally by just the block-wise sharing function above, and nothing else. This approach worked very well in practice, and let us achieve a peak of 74,000 MIPS in a shader-like functional program (i.e., a “tree-map” operation).

Unfortunately, it didn’t work so well in cases where the implied communication was more involved. For example, consider the following implementation of a purely functional Bitonic Sort:

```
data Tree = (Leaf val) | (Node fst snd)

// Swaps distant values in parallel; corresponds to a Red Box
(warp s (Leaf a) (Leaf b)) = (U60.swap (^ (> a b) s) (Leaf a) (Leaf b))
(warp s (Node a b) (Node c d)) = (join (warp s a c) (warp s b d))

// Rebuilds the warped tree in the original order
(join (Node a b) (Node c d)) = (Node (Node a c) (Node b d))

// Recursively warps each sub-tree; corresponds to a Blue/Green Box
(flow s (Leaf a)) = (Leaf a)
(flow s (Node a b)) = (down s (warp s a b))

// Propagates Flow downwards
(down s (Leaf a)) = (Leaf a)
(down s (Node a b)) = (Node (flow s a) (flow s b))

// Bitonic Sort
(sort s (Leaf a)) = (Leaf a)
(sort s (Node a b)) = (flow s (Node (sort 0 a) (sort 1 b)))
```

Since this is an  $O(N \cdot \log(N))$  algorithm, its recursive structure unfolds in such a manner that is much less regular than a tree-map. As such, the naive task sharing approach had a consequence that greatly impacted performance on GPUs: threads would give and receive “misaligned” redexes, causing warp-local threads to compute different calls at any given point. For example, a given warp thread might be processing `(flow 5 (Node _ _))`, while another might be processing `(down 0 (Leaf _))` instead. This divergence has the consequence of producing sequentialism in the GPU architecture, where warp-local threads are in lockstep.

To improve this, a different task-sharing mechanism has been implemented, which requires a minimal annotation: redexes corresponding to branching recursive calls are flagged with a `!` on the global Book. With this annotation, the GPU evaluator will then only share redexes from functions that recurse in a parallelizable fashion. This is extremely effective, as it allows threads to always get “equivalent” redexes in a regular recursive algorithm. For example, if given thread is processing `(flow 5 (Node _ _))`, it is very likely that another warp local thread is too. This minimizes warp divergence, and has a profound impact in performance across many



cases. On the `bitonic_sort` example, this new policy alone resulted in a jump from 1,300 MIPS to 12,000 MIPS (9x).

### 8.2. Optimization: Shared Memory Interactions.

GPUs also have another particularity that, if exploited properly, can result in significant speedups: shared memory. The NVIDIA RTX 4090, for example, includes A L1 Cache memory space of about 128KB, and GPU languages like CUDA usually allow a programmer to manually read and write from that cache, in a shared memory buffer that is accessible by all threads of a block. Reading and writing from shared memory can be up to 2 orders of magnitude faster than doing so from global memory, so, using that space properly is essential to fully harness a GPU's computing power.

On HVM32, we use that space to store a local node buffer, and a local subst map, with 8192 nodes and variables. This occupies exactly 96KB, just enough to fit most modern processors. When a thread allocates a fresh node or variable, that allocation occurs in the shared memory, rather than the global memory. With a configuration of 128 threads per block, each thread has a “scratchpad” of 64 nodes and vars to work locally, with no global memory access. This is often enough to compute long-running tail-loops, which is what makes HVM2 so efficient on shader-like programs.

There is one problem, though: what happens when an interaction links a locally allocated node to a global variable? This would cause a pointer to a local node to “leak” to another block, which would then be unable to retrieve its information, causing a runtime error. To handle this situation, we extend the LINK interaction with a “LEAK” sub-interaction, which is specific to GPUs only. That interaction essentially allocates a “global view” of the local node, filled with two placeholder variables, such that one copy is local, and the other copy is global (remember: variables are always paired). That way, we can continue the local reduction without interruptions. If another block does get this “leaked” node, it will be filled with two variables, which, in this case, act as “future” values which will be resolved when the local thread links it.

```

^A ~ (b1 b2)
----- LEAK
^X ~ b1
^Y ~ b2
^A ~ ^(^X ^Y)

```

The LEAK interaction allows us to safely work locally for as long as desired, which has great impact on performance. On the stress test benchmark, the throughput jumps from about 13,000 MIPS to 54,000 MIPS by this change alone.

## 9. GARBAGE COLLECTION

Since HVM2 is based on Interaction Combinators, which are fully linear, there is no global “garbage collection” pass required. By IC evaluation semantics alone, data is granularly allocated as needed, and freed as soon as they become unreachable. This is specifically accomplished by the ERAS and VOID interactions, which consume sub-nets that go out of scope in parallel, clearing them from memory. When HVM2 completes evaluation (i.e., after all redexes have been processed), the

memory should be left with just the final result of the program, and no remaining garbage or byproduct. No further work is needed.

## 10. IO

TODO: explain how we do IO

## 11. BENCHMARKS

TODO: include some benchmarks

## 12. TRANSLATIONS

TODO: include example translations from high-level languages to HVM2

## 13. LIMITATIONS

The HVM2 architecture, as currently presented, is capable of evaluating modern, high-level programs in massively parallel hardware with near-ideal speedup, which is a remarkable feat. That said, it has severe impactful limitations that must be understood. Below is a list of many of these limitations, and how they can be addressed in the future.

### 13.1. Only one Duplicator Node.

Since HVM2 is affine, duplicator nodes are often used to copy non-linear variables. For example, when translating the  $\lambda$ -term  $\lambda x. x + x$ , which is not linear (because  $x$  occurs twice), one might use a **DUP** node to clone  $x$ . Due to Interaction Net semantics, though, **DUP** nodes don't always match the behavior of cloning a variable on traditional languages. This, if not handled properly, can lead to **unsound reductions**. For example, the  $\lambda$ -term:

$C4 = (\lambda f. \lambda x. (f (f x))) \lambda f. \lambda x. (f (f x))$

Which computes the Church-encoded exponentiation  $2^2$ , can not be soundly reduced by HVM2. To handle this, a source language must use either a type system or similar mechanism to verify that the following invariant holds:

> A higher-order lambda that clones its variable can not be cloned.

While restrictive, it is important to stress that this limitation only applies to cloning higher-order functions. A language that targets the HVM can still clone datatypes with no restrictions, and it is still able to perform loops, recursion and pattern-matches with no limitations. In other words, HVM is Turing Complete, and can evaluate procedural languages with no restrictions, and functional languages with some restrictions. That said, this can be amended by:

1. Adding more duplicator nodes. This would allow “nested copies” of higher-order functions. With a proper type system (such as EAL inference), this can greatly reduce the set of practical programs that are affected by this restriction.
2. Adding bookkeeping nodes. These nodes, originally proposed by Lamping (1990), allow interaction systems to evaluate the full  $\lambda$ -calculus with no restrictions. Adding bookkeeping to HVM should be easy. Sadly, this has

the consequence of bringing a constant-time overhead, decreasing performance by about 10x. Because of that, it wasn't included in this model.

Ideally, a combination of both approaches should be used: a type-checker that flags safe programs, which can be evaluated safely on HVM2, and a fallback bookkeeper, which ensures sound reductions of programs that do not. Implementing such system is outside the scope of this work, and should be done as a future extension. [cite: the optimizing optimal evaluation paper]

### 13.2. Ultra-Eager Evaluation Only.

In our first implementation, HVM1, we used a lazy evaluation model. This not only ensured that no unnecessary work was done, but also allowed one to compute with infinite structures, like lists. Since the implementation presented here reduces **all** available redexes eagerly, that means neither of these hold. For example, if you allocate a big structure, but only read one branch, HVM2 will allocate the entire structure, while HVM1 wouldn't. And if you do have an infinite structure, HVM2 will never halt (because the redex bag will never be empty). This applies even to code that doesn't look like it is an infinite structure. For example, consider the JavaScript function below:

```
foo = x => x == 0 ? 0 : 1 + foo(x-1);
```

In JavaScript, this is a perfectly valid function. In HVM2, if called as-is, this would hang, because `foo(x-1)` would unroll infinitely, as we do not “detect” that it is in a branch. To make recursive functions computable, the usual approach is to split it into multiple definitions, as in:

```
foo    = x => x == 0 ? fooZ : foo_S(x - 1);
foo_Z  = 0;
foo_S  = x => 1 + foo(x-1);
```

Since REFs unfold lazily, the program above will properly erase the `foo_S` branch when it reaches the base case, avoiding the infinite recursion.

Extending HVM2 with a full lazy mode would require us to store uplinks, allowing threads to navigate through the graph and only reduce redexes that are reachable from the root port. While not technically hard to do, doing so would make the task scheduler way more complex to implement efficiently, specially in the GPU version. We reserve this for a future extension.

### 13.3. Single Core Inefficiency.

While HVM2 achieves near-linear speedup, allowing it to make programs run arbitrarily faster by just using more cores (as long as there is sufficient degree of parallelism), its compiler is still extremely immature, and not nearly as fast as state-of-art alternatives like GCC or GHC. In single-thread CPU evaluation, HVM2, is, baseline, still about 5x slower than GHC, and this number can grow to 100x on programs that involve loops and mutable arrays, since HVM2 doesn't feature these yet.

For example, a single-core C program that adds numbers from 0 to a few billions will easily outperform an HVM2 one that uses thousands of threads, given the C version is doing no allocation, while C is allocating a tree-like recursive stack. That said, not every program can be implemented as an allocation-free, register-mutat-

ing loop. For real programs that allocate tons of short memory objects, HVM2 is expected to perform extremely well.

Moreover, and unlike some might think, HVM2 is not incompatible with loops or mutable types, because it isn't a functional runtime, but one based on interaction combinators, which are fully linear. Extending HVM2 with arrays is as easy as creating nodes for it, and implementing the interactions, and can be done in a timely fashion as a fork of this repository. Similarly, loops can be implemented by optimizing tail-calls. We plan to add such optimization soon.

Finally, there are many other low-hanging fruits that could improve HVM2's performance considerably. For example, currently, we do not have native constructors, which means that algebraic datatypes have to be  $\lambda$ -encoded, which brings a 2x-5x memory overhead. Adding proper constructors and eliminating this overhead would likely bring a proportional speedup. Similarly, adding more numeric types like vectors would allow using more of the available GPU instructions, and adding read-only types like immutable strings and textures with 1-interaction reads would allow one to implement many algorithms that, currently, wouldn't be practical, specially for graphics rendering.

#### 13.4. 32-bit Architecture Limitations.

Since this architecture is 32-bit, and since 3 bits are reserved for a tag, that leaves us with a 29-bit addressable space. That amounts for a total of about 500 million nodes, or about 4 GB. Modern GPUs come with as much as 256 GB integrated memory, so, HVM2 isn't able to fully use the available space, due to addressing constraints. Moreover, its 29-bit unboxed numbers only allow for 24-bit machine ints and floats, which may not be enough for many applications.

All these problems should be solved by extending ports to 64-bit and nodes to 128-bits, but this requires some additional considerations, since modern GPUs don't come with 128-bit atomic operations. We'll do this in a future extension.

#### 13.5. More.

#### Work In Progress

### 14. CONCLUSION

By starting from a solid, inherently concurrent model of computation, Interaction Combinators, carefully designing an efficient memory format, implementing lock-free concurrent interactions via lightweight atomic primitives, and granularly distributing workload across all cores, we were able to design a parallel compiler and evaluator for high-level programming languages that achieves near-linear speedup as a function of core count. While the resulting system still has many limitations, we proposed sensible plans to address them in the future.

This work creates a solid foundation for arallel programming languages that are able to harness the massively parallel capabilities of modern hardware, without demanding explicit low-level management of threads, locks, mutexes and other complex synchronization primitives by the programmer.

### REFERENCES

1. Lafont Y. 1997. Interaction Combinators. *Information and Computation*. 137(1):69–101

2. Mazza D. 2007. A denotational semantics for the symmetric interaction combinators. *Mathematical Structures in Computer Science*. 17:527–62

*Email address:* [taelin@HigherOrderCO.com](mailto:taelin@HigherOrderCO.com)