

HVM2: Interaction Combinator Evaluator - Extended Abstract

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

Francisco Javier Grecco
Carman
fgreccocarman@gmail.com

Nicolas Abril
nicolas@higherorderco.com
Higher Order Company
Curitiba, Brazil

Enrico Zandomeni Borba
enricozb@higherorderco.com
Higher Order Company
Amsterdam, Netherlands

Abstract

We present a preview of HVM2, an efficient and massively parallel GPU and CPU evaluator for a system of extended Interaction Combinators. When compiling non-sequential programs from a high-level programming language to HVM2 and executing them on the C and CUDA runtimes, we achieved a large parallel speedup as a function of cores available. We give an overview of HVM2's theoretical foundations, implementation, early benchmarks, current limitations, and future work.

CCS Concepts

• Computing methodologies → Massively parallel algorithms • Theory of computation → Parallel computing models; Lambda calculus.

Keywords

Interaction Combinators, Parallel Computation, Lambda Calculus

1 Introduction

Interaction Nets (INs) [3] and Interaction Combinators (ICs) [4] were introduced by Lafont as a minimal and concurrent model of computation. Lafont proved that ICs were not only Turing Complete, but also that the “complexity class and degree of parallelism” is preserved when encoding a TM in an Interaction Combinator system [4]. The locality and strong confluence properties of ICs make it a promising target for massively parallel computation. However, 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 in a lock-free fashion and with minimal synchronization. We also extend this system with global definitions for fast function applications and with native numbers for fast numeric operations.

2 Syntax

HVM2's syntax consists of an Interaction Calculus system which textually represents an Interaction Combinator system [2]. This textual system is capable of representing any arbitrary Interaction Net, and it is therefore possible to represent “vicious circles” [4]. We only consider HVM2 programs which do not contain any vicious circles.

HVM2 extends Lafont's ICs and has seven different types of agents or *nodes*. HVM2 also has *variables* to represent wires which connect ports across nodes, and top-level *definitions*. Lastly, *trees* are either variables or nodes. The entry-point of an HVM2 program is the definition `@main`.

The syntax as a BNF grammar is:

```
<Def> ::= "@" <alphanumeric> "=" <Net>

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

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

<Numeric> ::=
| <u32, i32, f32>
| "[" <op> (<u32, i32, f32>)? "]"
```

Nets represent packages [4], which are a collection of active pairs and one free port. Active pairs are also called *redexes*, since in HVM2 all active pairs can be reduced. For example,

$$t_1 \& v_1 \sim w_1 \& \dots \& v_n \sim w_n$$

represents the Net depicted in Figure 1, where t_1 , v_1 , w_1 , v_n , and w_n are trees and ω is the wiring specified by pairs of VAR nodes.

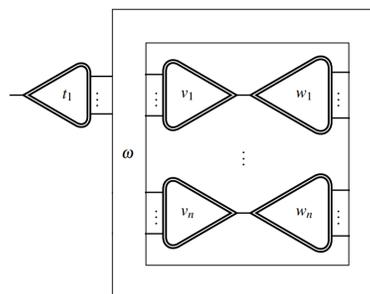


Figure 1: A sample Net¹

3 Extensions to Interaction Combinators

The CON, DUP, and ERA nodes correspond accordingly to Lafont's *constructor*, *duplicator*, and *eraser* symbols [4], but behave like Mazza's Symmetric Interaction Combinators [6].

REF nodes are an extension to Lafont's ICs, and they represent a reference to an immutable net that is expanded when it is one of the two nodes in a redex. While not essential for the expressivity of the system, REF nodes are essential for performance. REFs enable fast global functions and a degree of laziness in a strict setup (critical to making GPU implementations viable).

Lastly, the NUM, OPE, and SWI nodes are also extensions, and they enable efficient mathematical operations and conditional branching. This is in contrast to less-efficient constructs such as Church or Scott numerals.

¹This is a modified image of a configuration with multiple free ports [7].

4 Interactions

Interactions are the term-rewriting rules that govern the reduction of an HVM2 program. They are as follows:

$A, B, C, D := \langle \text{Tree} \rangle$
 $(), \{ \} := \text{CON} \mid \text{DUP} \mid \text{OPA} \mid \text{SWI}$ ²
 $\bullet, \circ := \text{ERA} \mid \text{REF} \mid \text{NUM}$ ³
 $N, M := \text{NUM}$ (Numbers or Operations)
 $\#n, \#m := \text{NUM}$ where $n, m \in \mathbb{Q}$
 $*$:= ERA
 $x, y, z, w := \text{VAR}$

$(\text{LINK}^4) \frac{B \text{ contains } x \quad x \sim A}{B[x \leftarrow A]}$ $(\text{CALL}) \frac{A \text{ is not a VAR node} \quad @\text{foo} \sim A}{\text{expand}(@\text{foo}) \sim A}$

$(\text{VOID}) \frac{\bullet \sim \circ}{}$ $(\text{ERASE}) \frac{\bullet \sim (A B)}{\bullet \sim A \quad \bullet \sim B}$

$(\text{COMMUTE}) \frac{(A B) \sim \{C D\} \quad \{x y\} \sim A \quad \{z w\} \sim B \quad (x z) \sim C \quad (y w) \sim D}{(A B) \sim \{C D\}}$ $(\text{ANNIHILATE}) \frac{(A B) \sim (C D) \quad A \sim C \quad B \sim D}{(A B) \sim (C D)}$

$(\text{OPERATE } 1) \frac{N \sim \$(M A)}{\text{op}(N, M) \sim A}$ $(\text{SWITCH } 1) \frac{\#0 \sim ?(A B)}{A \sim (B *)}$

$(\text{OPERATE } 2) \frac{A \text{ is not a NUM node} \quad N \sim \$(A B)}{A \sim \$(N B)}$ $(\text{SWITCH } 2) \frac{\#n+1 \sim ?(A B)}{A \sim (* (\#n B))}$

Note that rules are *symmetric*: if a rule applies to a redex $A \sim B$ then it also applies to $B \sim A$. Additionally, the type of the two nodes in an active pair uniquely determine which interaction rule to apply.

5 Implementation

5.1 Memory Layout

HVM2 is based on a 32-bit architecture. We represent wires connected to some main port with a 32-bit value. The lower 3 bits identify the type of node (VAR, REF, ERA, etc.) whose main port the wire is connected to. This is a port’s *tag*. The upper 29 bits hold a port’s *value*. The interpretation of the value is dependent on the tag. The value is either an address (for binary CON, DUP, OPE, and SWI nodes), a virtual function address (for REF nodes), a 29-bit number⁵ (for NUM nodes), a variable name (for VAR nodes), or \emptyset (for ERA nodes). Binary nodes are represented in memory as a pair of two ports. Notice that *ports store the type of node they are connecting to*. Nodes do not store their own type.

Port = [29-bit Value][3-bit Tag]
 Node = [32-bit Port][32-bit Port]

5.2 Multi-Threading

Like in Lafont’s ICs, the interaction rules are *local* [3] and therefore can be reduced in parallel. HVM2 maintains a global collection of redexes that is mutated in parallel. However, because HVM2 has variables, distant parts of a net can be connected. When two

² $()$ and $\{ \}$ refer to *different* binary nodes.

³ \bullet, \circ refer to *potentially different* nilary nodes.

⁴ This is not technically an interaction in an IC sense, as variables are not present in the graphical representation.

⁵ Number nodes are 24-bit numerals with a 5-bit tag. This tag is either a type (U24, I24, F24), or an operator (ADD, SUB, etc) for partially-applied operations.

threads each attempt to reduce neighboring redexes, there could be contention. For example, consider a subset of a Net,

$.. \& (a b) \sim (d c) \& .. \& (c d) \sim (f e) \& ..$

represented graphically by Figure 2.

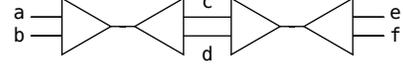


Figure 2: A pair of redexes.

After reduction, a must be wired to e “through” d . However, both threads will want to affect what d is pointing to. We resolve this potential source of contention through *linking*.

5.3 Linking

Since every variable occurs exactly twice, the LINK interaction with a variable x will also occur at most twice, but *possibly at very different times*. When the first LINK interaction occurs, a substitution must be “deferred” until the second LINK interaction.

This is accomplished by a global, atomic substitution map, which tracks these deferred substitutions. When a variable is linked to a node, or to another variable, it is inserted into the substitution map. When that same variable is linked again, it will already have an entry in the substitution map, and then the proper redex will be constructed, and returned to the global redex collection.

The substitution map can be represented efficiently with a flat buffer, where the index is the variable name, and the value is the node that has been substituted. This can be done atomically⁶, via a simple lock-free linker. In pseudocode, this roughly looks like:

```

def link(subst: [Port; NUM_VARS], A: Port, B: Port):
  while True:
    if type(A) != VAR: swap(A, B)
    if type(A) != VAR: push_redex(A, B)
    got: Port = subst.set_atomic(A.var, B)
    if got is None: break
    subst.set_atomic(A.var, None)
    A = got
  
```

6 Benchmarks

With an appropriate encoding of a subset⁷ [1,5] of λ -calculus terms to HVM2 interaction combinators, we can write high-level programs to compare their execution speed and measure speed gains with respect to the number of threads.

Program	HVM2 C ⁸	HVM2 CUDA ⁹
sort_bitonic($n = 2^{19}$)	9.38s	0.27s
sort_radix($n = 2^{19}$)	2.95s	0.82s
sum_tree($d = 14, b = 2^{20}$)	128.83s	6.97s

Table 1: Benchmarks for Sorting Algorithms

With the C runtime utilizing 24 threads and the CUDA runtime utilizing 32,768 threads, we see clear gains with respect to the number of available threads. Due to the 32-bit architecture, the node address space is limited to 4GB. A future 64-bit implementation could be possible on platforms supporting 128-bit atomics.

⁶ This critically depends on platforms providing 64-bit atomic operations.

⁷ Not all λ -calculus terms reduce correctly in our system, as we do not include brackets or croissant nodes. The subset that do reduce soundly is still Turing Complete.

⁸ Executed on an AMD Ryzen 9 7900X 12-Core Processor

⁹ Executed on an NVIDIA RTX 4090.

References

- [1] Andrea Asperti and Stefano Guerrini. 1998. *The optimal implementation of functional programming languages*.
- [2] Maribel Fernández and Ian Mackie. 1999. *Principles and Practice of Declarative Programming*. Springer Berlin Heidelberg. <https://doi.org/10.1007/10704567>
- [3] Yves Lafont. 1990. Interaction Nets. *POPL '90* (1990). <https://doi.org/10.1145/96709.96718>
- [4] Yves Lafont. 1997. Interaction Combinators. *Information and Computation* 137, 1 (August 1997), 69–101. <https://doi.org/10.1006/inco.1997.2643>
- [5] John Lamping. 1990. An algorithm for optimal lambda calculus reduction. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '90 (POPL '90)*, 1990. ACM Press. <https://doi.org/10.1145/96709.96711>
- [6] Damiano Mazza. 2007. A denotational semantics for the symmetric interaction combinators. *Mathematical Structures in Computer Science* 17, (2007), 527–562. <https://doi.org/10.1017/S0960129507006135>
- [7] Anton Salikhmetov. 2016. Token-passing Optimal Reduction with Embedded Read-back. *Electronic Proceedings in Theoretical Computer Science* 225, (September 2016), 45–54. <https://doi.org/10.4204/eptcs.225.7>