UNIVERSITY OF
CAMBRIDGE
Department of Computer
Science and Technology

Research project report

Candidate 2496V

"Using Interaction Nets for Parsing on GPUs"

Submitted in partial fulfillment of the requirements for the
Computer Science Tripos, Part III

Total page count: 52

Main chapters (excluding front-matter, references and appendix): 43 pages (pp 6–48)

Main chapters word count: 11965

Methodology used to generate that word count: `texcount.pl -sum`[1]

Declaration

I, 2496V, being a candidate for Part III of the Computer Science Tripos, hereby declare that this project report and the work described in it are my own work, unaided except as may be specified below, and that the project report does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this project report I did not use text from AI-assisted platforms generating natural language answers to user queries, including but not limited to ChatGPT. I am content for my project report to be made available to the students and staff of the University.

Date: 28th of May, 2024

[1] `https://app.uio.no/ifi/texcount/intro.html`

Abstract

There is a keen interest in lowering the parsing times of code and data in a variety of settings, including compilation and low-latency data processing. In recent year, graphics processing units (GPU) have been at the forefront of optimizing many algorithms, beyond the scope of graphical applications. This dissertation employs interaction nets, an inherently parallel model of computation, to overcome the sequential execution of parsing algorithms.

Until now, it has remained unclear to what extent interaction nets are a worthwhile model of computation in terms of providing more efficient parallel implementations of algorithms. Despite being touted as an inherently model of computation, there has been little work investigating whether their advantage in parallelism actually manifests in practical tasks.

We extend an existing parallel parsing algorithm to include LR grammars, thereby designing a novel parallel LR parser. Additionally, we develop the fastest-yet interaction net evaluator designed to run on GPUs. Nonetheless, it does not reach the standard of the current state-of-the-art interaction net evaluator implemented on a CPU. We then implement our parallel parsing algorithm using interaction nets. However, our interaction net evaluator delivers lackluster performance, resulting in poor parsing times by our interaction net parser. Nevertheless, this presents the first non-trivial practical use of interaction nets.

In our evaluation, we identify two main flaws with using interaction nets for complex computations. First, interaction nets require an immense amount of memory for even small input sizes. Second, the majority of their execution is spent duplicating parts of the interaction net. These two factors combined make interaction nets an unviable model of computation for complex algorithms.

Contents

Chapter 1

Introduction

Aside from speeding up compilation, there is a keen interest in lowering parsing times in many low-latency settings. Modern webpages are written across dozens of Javascript files, each of which needs to be parsed promptly to load the webpage quickly. Additionally, JSON files have become a standard for storing large, structured datasets, such that processing these would benefit from an efficient parser as well. Nevertheless, parsing programming languages remains a computationally expensive task, such that even modern optimizing compilers spend the majority of time in the parsing phase (Table 1.1).

With the breakdown of Dennard scaling, hardware performance improvements in the last 20 years have largely come from increasing parallelism. Taking parallelism to the extreme, graphics processing units (GPUs) have been at the forefront of optimizing many tasks beyond their initial domain in graphics. Their massively-parallel architecture allows for the efficient execution of large, regularly repeating algorithms, driving in part the current revolution in machine learning.

Unfortunately, parsing falls victim to Amdahl's Law [19], which describes the upper bound of improvements that can be derived from increasing parallelism:

$$I = \frac{1}{B + (1 - B)/n}$$

Phase	Time (s)	Percentage of Total
Setup	0.19	0.57%
Parsing	**19.35**	**57.80%**
Lang. Deferred	4.03	12.04%
Optim. and Gen.	9.26	27.66%
Last asm	0.63	1.88%
Finalize	0.02	0.06%

Table 1.1: Time spent in each compilation phase when compiling our project's C++ code. The data given was reported by `g++ -ftime-report`.

Amdahl's law states that the sequential proportion of a program B asymptotically bounds the performance improvement I when increasing the number of threads n. Here lies the reason why parsing has failed to be efficiently parallelized. A parser advances its input one token at a time, such that the parser's internal state is dependent on all the input tokens it has seen. This sequential nature of parsing makes parallelization difficult.

To overcome Amdahl's law, this dissertation employs *interaction nets* [10], an inherently parallel model of computation. In doing so, this investigation is in line with the recently renewed interest in interaction nets that aims to implement them on a GPU [6, 23]. Until now, it has remained unclear to what extent interaction nets are a worthwhile model of computation in terms of providing more efficient parallel implementations of algorithms. Despite being touted as an inherently model of computation, there has been little work investigating whether their advantage in parallelism actually manifests in practical tasks. There remains some concern that interaction nets are too far removed from the standard Von Neumann architecture to provide an efficient abstraction for computation [12]. It is precisely this debate that our investigation wishes to contribute to.

This project aims to provide a massively-parallel parsing algorithm that can run on a GPU via an implementation using interaction nets. This gives us an instance of an interaction net with which to investigate whether a use case exists for them outside academia. In this manner, we aim to answer the question as to whether interaction nets are a useful abstraction for general-purpose massively-parallel programming.

Contributions:

Beyond contributing to the general discussion surrounding interaction nets, in this dissertation we make the following contributions:

1. We extend Skillicorn and Barnard's parallel parsing algorithm [21] to a broader class of grammars, specifically LR grammars, thereby designing a novel parallel LR parser (Section 3.2).

2. We develop the fastest-yet massively-parallel interaction net evaluator designed to run on GPUs. Nonetheless, it does not reach the standard of the current state-of-the-art in interaction net evaluators (Section 3.3).

3. We implement our extended version of Skillicorn and Barnard's parallel parsing algorithm using interaction nets (Section 3.1). When executed on our interaction net evaluator, our parsing algorithm delivers lackluster performance. Nevertheless, this presents the first non-trivial practical use of interaction nets beyond implementations of the λ-calculus and simple benchmarking algorithms.

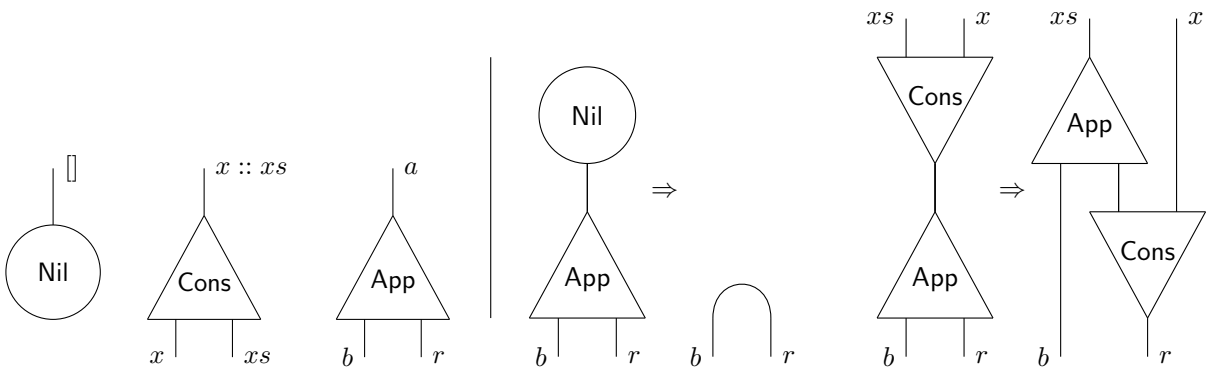
Chapter 2

Background

Before describing the work undertaken in this dissertation, we provide the relevant background on the following subjects: First, in Section 2.1 we describe the interaction net model of computation [10]. Second, in Section 2.2 we give an overview of Skillicorn and Barnard's parallel parsing algorithm [21] which we extend. Third, in Section 2.3, we provide the relevant details of the CUDA architecture for programming on GPUs.

2.1 Interaction Nets

We look to use *interaction nets* [10] to overcome the linear nature of parsing. The aim is to leverage their inherent parallelism to reap the benefits of massively-parallel computing on GPUs. Yves Lafont developed interaction nets as a graphical model of computation, like the Turing machine or the λ-calculus. We utilize the symmetry between interaction nets and the λ-calculus to devise a translation of a parsing algorithm into interaction nets.



(a) Types (b) Interaction rules

Figure 2.1: Definitions for linked lists as interaction nets

2.1.1 Defining Interaction Nets

An interaction net is a graph consisting of *agents*. Each agent has a principal port and any number of auxiliary ports. It is through these ports that we can connect agents. Each agent is also given a *type*, which defines the agent's arity (number of auxiliary ports) and its interactions with other agents. To give a simple example of how interaction nets work, Figure 2.1 illustrates how we might implement a linked list using interaction nets. In this example, we have three types of agents Nil, Cons and Append. Nil has arity zero and is therefore drawn as a circle with only one outgoing edge, namely its principal port. The Cons and Append types have arity two and are thus drawn as triangles with two auxiliary ports opposite the principal port: the edge connected to the top vertex of the triangle.

Computation using interaction nets is done through *interactions*. An interaction between two agents occurs when their principal ports are connected, which results in the rewiring of the agents. The manner in which they are rewired depends on the two interacting agents' types, such that we define an interaction rule between each pair of types that can interact (Figure 2.1b). When defining an interaction rule, the number of free ports (the auxiliary ports of the interacting agents) must be the same before and after the interaction is performed. Otherwise, parts of the interaction net would be left disconnected.

We can compare interaction nets with a standard computer program. Instead of writing the program as a series of instructions, we define an initial network, alongside the types and interaction rules of the interaction net. The execution of a program becomes analogous to performing the interactions in the network and reducing it until there are no connected principal ports remaining. The deterministic property of programs is enforced by strictly defining one interaction rule per pair of types so that all interactions are deterministic performed.

In Figure 2.1b, we see the interaction rules for the linked list types. Here, the Append agent appends the list connected to its left auxiliary port to the list connected to its principal port. The result of the append interaction then comes out of the Append agent's right auxiliary port r. The two interaction rules for Append reflect the following inductive definition:

$$[] \mathbin{@} b = b$$
$$(x :: xs) \mathbin{@} b = x :: (xs \mathbin{@} b))$$

Additionally, Figure 2.2 gives an example of evaluating the expression $(A :: []) \mathbin{@} (B :: C :: [])$ using the definitions in Figure 2.1.

In summary, interaction nets are defined through a set of types with corresponding arities, and interaction rules between agents of these types. To build a program we construct a network from these agents, and to evaluate the program, we perform the interactions in

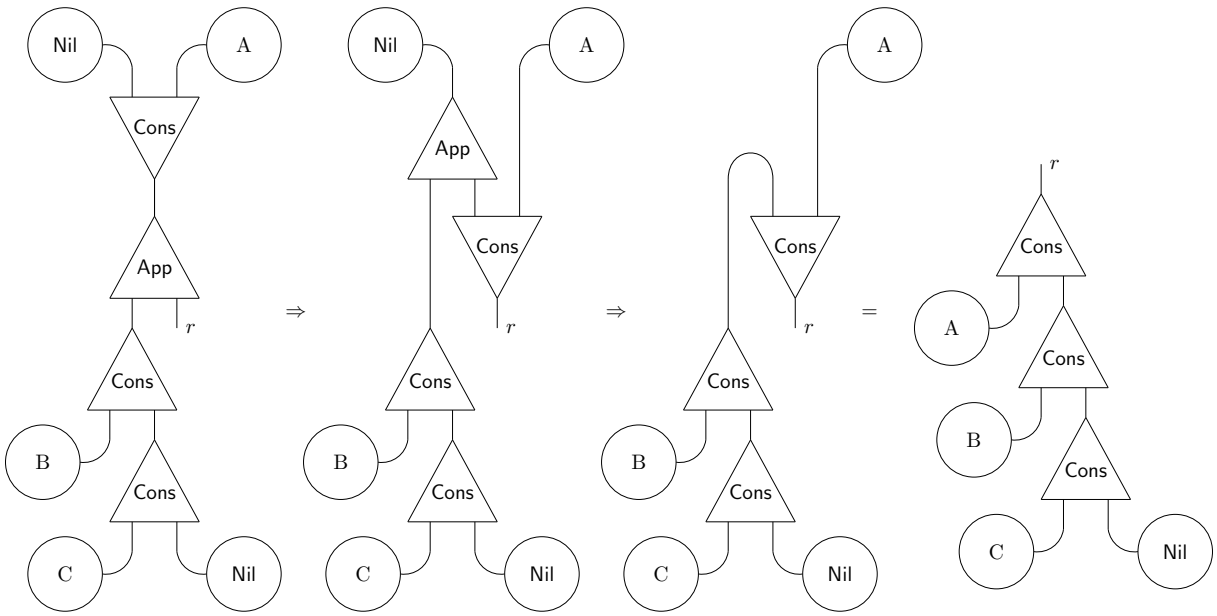


Figure 2.2: Interaction net computation of $(A :: []) \mathbin{@} (B :: C :: []) = A :: B :: C :: []$

the network until it cannot be reduced further.

2.1.2 Properties of Interaction Nets

Interaction nets have been described as an inherently parallel model of computation [20], and we intend to leverage this attribute to parallelize parsing algorithms for running on a GPU. This inherent parallelism comes from interaction nets' *strong confluence* and *determinacy* properties.

Strong Confluence Property

$$\nu \to \nu_1 \ \wedge \ \nu \to \nu_2 \quad \Longrightarrow \quad \nu_1 \to^* \nu' \ \wedge \ \nu_2 \to^* \nu'$$

Let $\nu \to \nu'$ indicate a reduction. If an interaction net ν can reduce in one step to different interaction nets ν_1 and ν_2, then ν_1 and ν_2 will both eventually reduce to a common ν'. This property holds because any two interactions must involve disjoint pairs of agents because agents only have one principal port. Given that an interaction only rewires the two interacting agents, all interactions can be performed independently without being affected by other interactions.

Determinacy Property

$$\nu \Downarrow \nu_1 \ \wedge \ \nu \Downarrow \nu_2 \quad \Longrightarrow \quad \nu_1 = \nu_2$$

Let $\nu \Downarrow \nu'$ indicate that interaction net ν is reduced until it reaches an irreducible net ν'. By the strong confluence property, if ν reduces to an irreducible net ν', then any reduction starting at ν will end at ν' regardless of the order in which the interactions are performed. Thus, given these two properties, we see how interaction nets are inherently

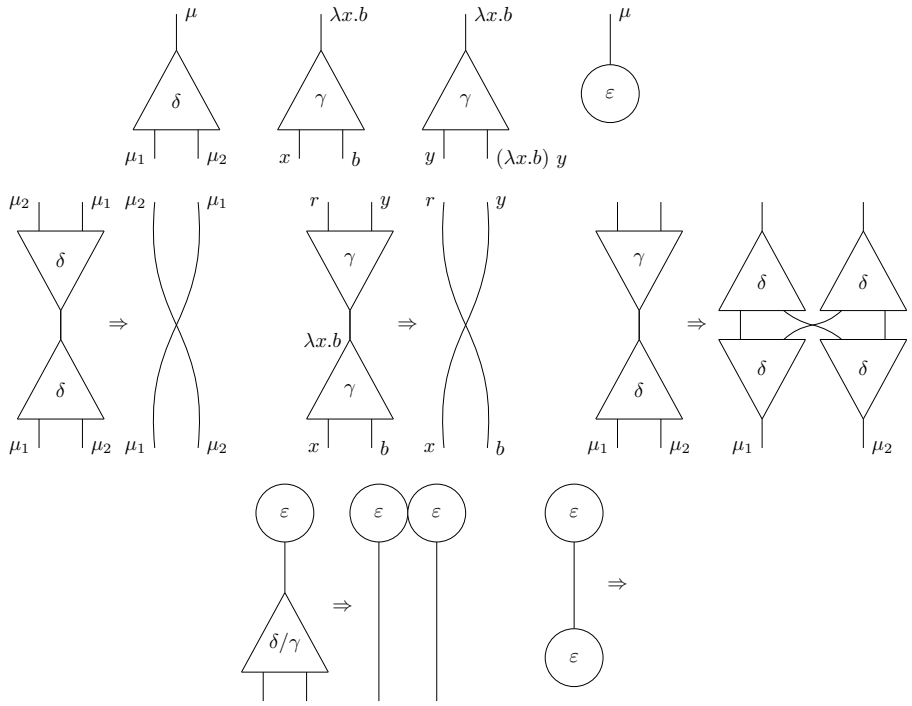


Figure 2.3: The symmetric interaction combinators and their interaction rules

parallel, as they allow active interactions to be performed in any order or at the same time. Interaction nets are, therefore, an asynchronous and deterministic model of computation and, thus, a promising match for GPUs' massively-parallel and asynchronous runtimes.

2.1.3 Programming Using Interaction Nets

In order to familiarize the reader with programming using interaction nets, we can use the λ-calculus as a stepping stone. We can encode the λ-calculus with interaction nets using Mazza's symmetric interaction combinators [13]. These are three types (Figure 2.3) that implement the behavior of the λ calculus: The δ type represents duplication, the γ type serves as both λ-construction and application, and the ε type deletes other agents.

To illustrate the two purposes the γ type has, we provide two illustrations in Figure 2.3. A $\gamma \bowtie \gamma$ interaction corresponds to a β-reduction in the λ-calculus, i.e. $(\lambda x.b)\ y = r$. Looking at the connections for this interaction in Figure 2.3, we see how the lambda variable x is substituted for the term y by connecting these two. Moreover, the result of the reduction r is bound to the body b of the λ-term with x having been substituted for y. We provide an example in Figure 2.4 showing how the interaction net translating the λ-term $(\lambda x.\lambda y.x\ x\ y)\ (\lambda z.z)$ is fully reduced.

The λ-calculus can be extended with complex datatypes such as natural numbers, products and sums through Church encodings. We can translate these Church encodings using the interaction net types just defined to construct these same datatypes as interaction nets. In this manner, we can build up more complex algorithms such our intended parser.

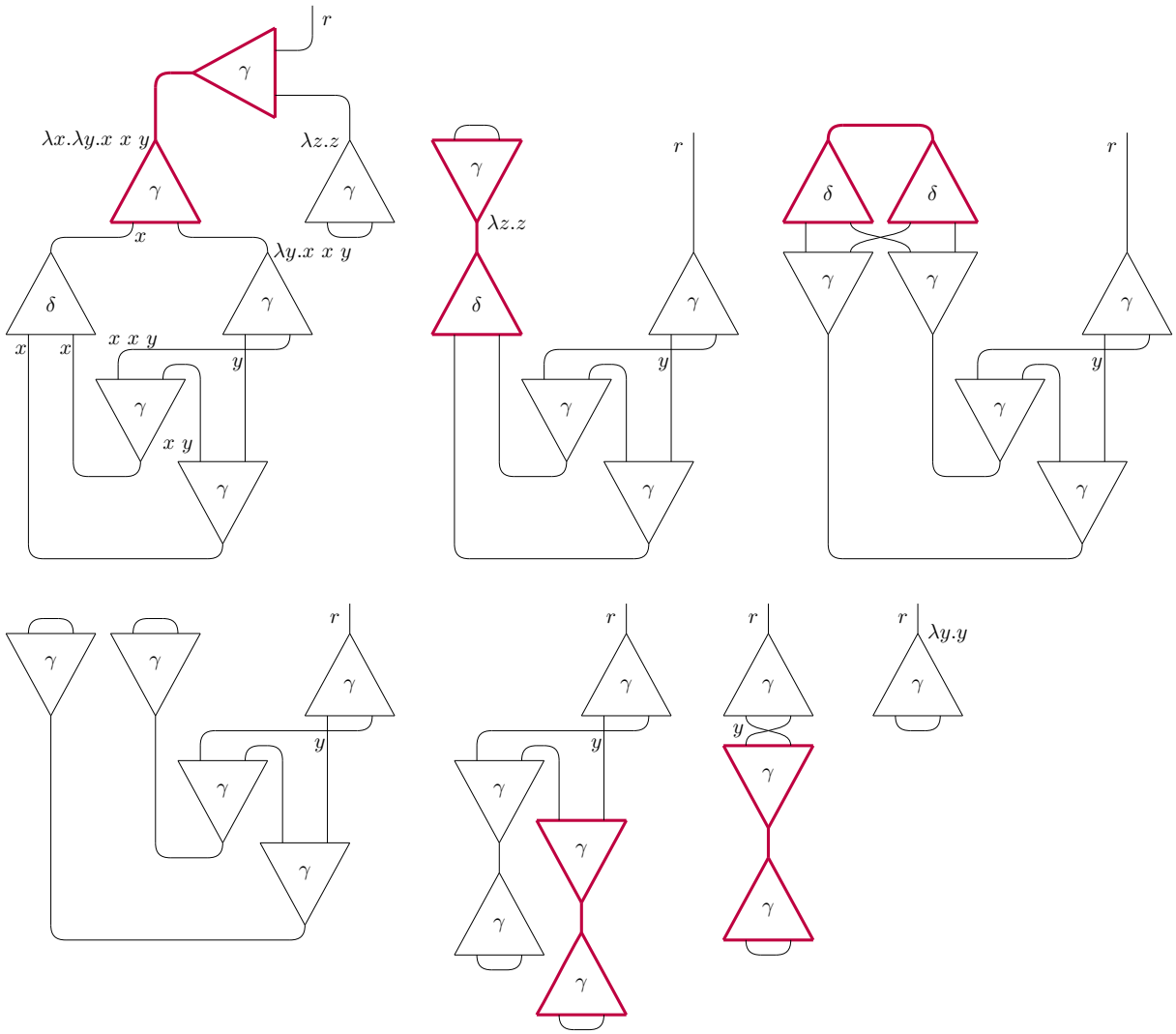


Figure 2.4: Reduction of the interaction net encoding $(\lambda x.\lambda y.x\ x\ y)\ (\lambda z.z)$. Red interactions indicate the ones being performed in each step.

2.2 Parallel Parsing on the Connection Machine

Even though interaction nets can be evaluated in parallel, the degree of parallelism they provide is dependent on the computation being performed [12]. If the given interaction net algorithm takes the form of a linear sequence of agents (e.g. appending a list one agent at a time), then the strong confluence property does not provide any benefit. To effectively utilize this property, we implement Skillicorn and Barnard's parallel parsing algorithm [21] using interaction nets. Their algorithm parses LL(1) grammars in $\mathcal{O}(n)$ time (n being the length of the input string) when run sequentially, but in $\mathcal{O}(\log n)$ when run in parallel on n processors. This algorithm was originally designed for the Connection Machine, a series of massively-parallel computer developed in the 1980s [5]. The Connection Machines' architecture largely resembled that of a modern GPU, having up to 65,536 one-bit processors. Given the similarities in target architecture, Skillicorn and Barnard's algorithm should be well-suited for an implementation on a GPU.

Skillicorn and Barnard's parallel parsing algorithm [21] hinges on the correspondence between table parsers and pushdown automata: machines that manipulate an internal stack based on an input string being read sequentially. When a parser comes across an input token, its stack will be in some configuration $I = i_n \dots i_1$. Then, based on the input token, the parser will perform the shift or reduction that it finds in its action table. This action will cause the parser to pop and push items off the stack, resulting in some other stack configuration $O = o_m \dots o_1$. Skillicorn and Barnard define such an update as a *stack mapping* $M = I/O$. Since stack operations are done exclusively to the top-of-the-stack (ToS), we use the special symbol '-' as a placeholder for the bottom of the stack that is left unchanged. For example, a stack mapping *abc-/dc-* describes the actions of popping off the top two items *ab*, and then pushing the item *d* back on top. Anything after item *c* is unaffected by the mapping.

The parser's grammar rules limit the set of stack mappings $S^{(x)}$ that the parser can perform when it next encounters an input token x. Skillicorn and Barnard describe how we can find these sets of stack mappings for LL(1) grammars from their parsing table [21]. Figure 2.5 gives an example of an LL(1) grammar with its parsing table describing what action the parser should take, given firstly, the next token in the input and, secondly, the current item at the ToS.

Each action in the parsing table under an input token x corresponds to a stack mapping in $S^{(x)}$. A shift action for x is converted to the mapping x-/-. This describes how a shift pops off the top item on the stack if it matches the input token. The stack mappings in $S^{(x)}$ that correspond to reduction actions describe how the stack is transformed when reducing a rule R_i until the input token x is reached. Additionally, we include starting and ending stack mappings $/A$ and -/ for the start and end tokens '⟨' and '⟩' that are added implicitly to the input string.

$R_1: \quad A \to B\ x\ C$
$R_2: \quad B \to b\ B$
$R_3: \quad B \to y$
$R_4: \quad C \to a\ C\ c$
$R_5: \quad C \to z$

(a)

Stack top	Input token a	b	c	x	y	z
A		R_1			R_1	
B		R_2			R_3	
C	R_4					R_5
a	shift					
b		shift				
$\vdots$			$\ddots$			

(b)

Figure 2.5: LL(1) grammar and corresponding parsing table

We give an example of how a reduction rule is transformed into a stack mapping for the reduction of rule $R_1 : A \to B\ x\ C$ when the next input token is y. The reduction of this rule is transformed into the mapping $A\text{-}/BxC\text{-}$. However, when performing this reduction we do not yet advance the input token y. We therefore further reduce the ToS B by rule $R_3 : B \to y$, giving the composed mapping $A\text{-}/yxC\text{-}$. Now that the terminal y is the ToS, we can pop it off to advance the input, producing the final stack mapping $A\text{-}/xC\text{-}$, as seen in Figure 2.6. Moreover, we attach the rules reduced when producing the stack mapping in order to derive the parse tree at the end (e.g. $A\text{-}/xC\text{-} \quad (1,3)$).

a	b	c	x	y	z	$\langle$	$\rangle$
$a\text{-}/\text{-}$	$b\text{-}/\text{-}$	$c\text{-}/\text{-}$	$x\text{-}/\text{-}$	$y\text{-}/\text{-}$		$/A$	$\text{-}/$
$C\text{-}/Cc\text{-} \quad (4)$	$A\text{-}/BxC\text{-} \quad (1,2)$			$A\text{-}/xC\text{-} \quad (1,3)$	$z\text{-}/\text{-}$		
	$B\text{-}/B\text{-} \quad (2)$			$B\text{-}/\text{-} \quad (3)$	$C\text{-}/\text{-} \quad (5)$		

Figure 2.6: Stack mappings for the grammar described in Figure 2.5a, with the reduction rules used in producing the mappings in parentheses.

The stack mappings in Figure 2.6 describe how a parser changes its stack when advancing one input token. The composition of two stack mappings describes how a parser changes its stack when advancing a sequence of two input tokens. We can extend this principle to parse the entire input by recursively composing the stack mappings of adjacent input tokens. More concretely, given two adjacent input tokens xy, we compose the mappings in sets $S^{(x)}$ and $S^{(y)}$. This is done by pairwise checking which mappings from $S^{(x)}$ have output stacks $O_i^{(x)}$ compatible with the initial stacks $I_j^{(y)}$ in $S^{(y)}$. Compatible stack mappings are composed together to form the set $S^{(xy)}$.

$$S^{(xy)} = \left\{ M^{(xy)} \mid M_i^{(x)} \in S^{(x)}, M_j^{(y)} \in S^{(y)} \wedge M^{(xy)} = (M_i^{(x)} \circ M_j^{(y)}) \neq \bot \right\}$$

The composition of two stack mappings $M^{(x)} \circ M^{(y)} = (I^{(x)}/O^{(x)}) \circ (I^{(y)}/O^{(y)})$ is done by matching the stack items top-to-bottom in $O^{(x)}$ with those in $I^{(y)}$. Matching is done until failure or the end of one of the stacks is reached (Algorithm 1). In the latter case, say '-'

$\langle b$	by	xa	zc	$\rangle$
$/BxC$ $(1,2)$	A-$/xC$- $(1,2,3)$	xa-/-	zc-/-	-/
	B-/- $(2,3)$	xC-$/Cc$- (4)	Cc-/- (5)	

(a)

$\langle bby$	$xazc$	$\rangle$
$/xC$ $(1,2,2,3)$	$xaCc$-/- (3)	-/
	xC-/- $(4,5)$	

(b)

$\langle bbyxazc$	$\rangle$
/ $(1,2,2,3,4,5)$	-/

(c)

$\langle bbyxazc\rangle$
/ $(1,2,2,3,4,5)$

(d)

Figure 2.7: Stack mappings produced when parsing the string '$\langle bbyxazc\rangle$' using the grammar described in Figure 2.5a

is the i^{th} symbol in $O^{(x)}$, and it is reached before $I^{(y)}$'s '-' symbol. This means that to perform the stack mapping $M^{(y)}$, the parser expects the stack items that have not been matched $I^{(y)}[i..]$ to be at the ToS. Therefore, we need to add these items to the initial stack configuration in the composed mapping. The result is $I^{(xy)} = I^{(x)}I^{(y)}[i..]$, minus the '-' symbol in $I^{(x)}$ if it exists. Conversely, we have the case where the j^{th} symbol in $I^{(y)}$ is '-' and it is reached before the one in $O^{(x)}$. Now mapping $M^{(x)}$ pushes the items $O^{(x)}[j..]$ but these are not popped off by $M^{(y)}$. Thus, these items need to be added to the output configuration, giving $O^{(xy)} = O^{(y)}/O^{(x)}[j..]$ minus the '-' symbol in $O^{(y)}$.

Algorithm 1 Composing two stack mappings

Input: Stack mappings $I^{(x)}/O^{(x)}$ and $I^{(y)}/O^{(y)}$. Stacks not ending in symbol '-' are extended with a 'bottom-of-stack' symbol.
Output: The composed stack mapping $I^{(xy)}/O^{(xy)}$ or $\perp$ in case of failure

for $i \in \left[0, \min\left(\text{len}\left(O^{(x)}\right), \text{len}\left(I^{(y)}\right)\right)\right]$ **do**
  if $O^{(x)}[i] =$ '-' **then**
    return $I^{(x)}I^{(y)}[i..]/O^{(y)}$
  else if $I^{(y)}[i] =$ '-' **then**
    return $I^{(x)}/O^{(y)}O^{(x)}[i..]$
  else if $O^{(x)}[i] \neq I^{(y)}[i]$ **then**
    return $\perp$
  end if
end for
return $I^{(x)}/O^{(y)}$

In Figure 2.7 we give an example of parsing the input string '*bbyxazc*' extended with starting and ending tokens. The first parsing step composes the stack mappings of adjacent input tokens to produce the sets $S^{(\langle b)}, S^{(by)}, S^{(xa)}, S^{(zc)}$ with $S^{(\rangle)}$ left over (Figure 2.7a). Next, we compose these adjacent sets, giving $S^{(\langle bby)}, S^{(xazc)}$ (Figure 2.7b). We repeat this process until we have finally parsed the entire input. Thus, given that stack mapping compositions can be performed in constant time, the time complexity of the algorithm is $\mathcal{O}(\log n)$ assuming we have a computation thread per active set of mappings [21].

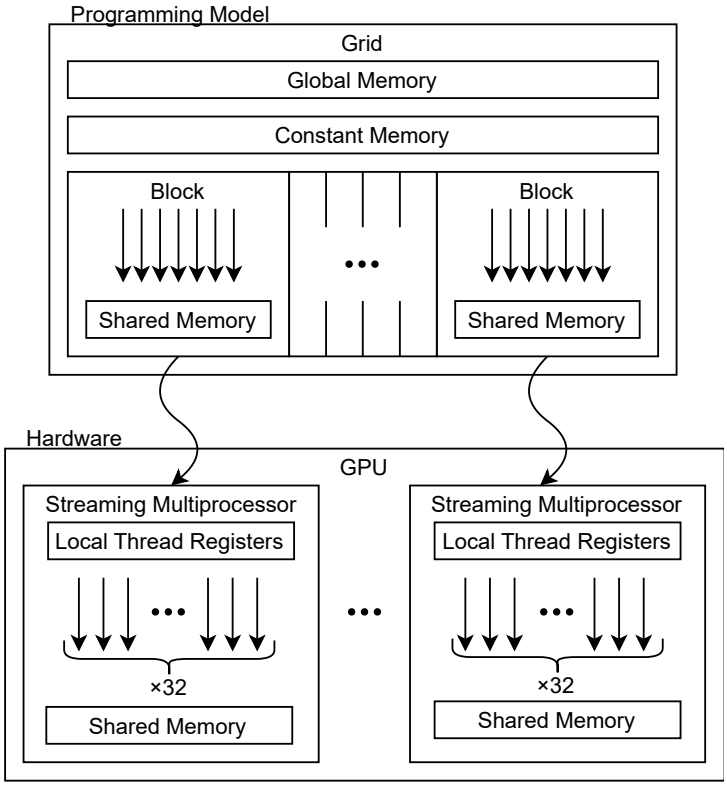


Figure 2.8: Basic CUDA programming model and hardware implementation.

2.3 Programming with CUDA

Our implementation of a parser on a GPU is written using CUDA, NVidia's framework for programming on their GPUs [17]. An efficient implementation requires intimate knowledge of this architecture and of how code is executed on GPUs. Here, we give a brief overview of the CUDA programming model (Figure 2.8) to illuminate the design choices made in the implementation described below.

Functions that are executed on GPU devices are termed *kernels*. A kernel is executed asynchronously and in parallel by up to thousands of different threads. The asynchronous and parallel nature of interaction nets, therefore, allows them to accommodate GPU's execution strategy. On the hardware itself, threads are managed, scheduled and executed together on streaming multiprocessors (SM) in groups of 32, named *warps*. Threads in a warp run code in parallel, executing each instruction together. Each thread is allocated local hardware registers for quick memory operations. Moreover, when a branch instruction is encountered, the two execution paths are serialized. The threads not on the active path are disabled until this path finishes execution. In the interest of efficiency, branching must, therefore, be minimized to limit the amount of time a thread spends disabled.

A scheduler dispatches the warps that are ready for execution to the SM. Scheduling can be done at instruction-level granularity, because there is no cost for switching warps in and out of execution. The scheduler reduces the time the multiprocessor spends idle since warps waiting on expensive memory reads can be quickly switched out. Yet, this is only

possible if other warps available for execution.

However, when programming using CUDA, warps are largely abstracted away. Instead, the programmer groups threads together in thread *blocks*. Threads in the same block are allocated to the same streaming multiprocessor core and are therefore able to coordinate their execution using synchronization barriers and shared memory. Resource limitations per core restrict block size to a maximum of 1024 threads. Furthermore, the shared memory within each block gives faster access than global memory on the GPU. Therefore, the programmer should also restrict block size by the amount of limited shared memory and local registers that each thread requires when running a given kernel.

Thread blocks can then be further organized into *grids*. All the threads in a grid execute the same kernel, and as such, the size of a grid is usually determined by the amount of available work. Note that with a large enough grid, there may not be enough SMs on a particular GPU to accommodate all thread blocks at the same time. Thread blocks are scheduled on available SMs, with new blocks taking the place of ones that finish execution.

All threads in a grid have access to coherent global memory, albeit at a much lower bandwidth than shared block memory. Additionally, there is read-only global constant memory that is cached separately and can be accessed at higher bandwidth than standard global memory. When a kernel is called by the host application, the block and grid dimensions are specified alongside any input arguments. Choosing an appropriate block and grid size involves a careful balance between the resources available on the GPU and the work needed.

Chapter 3

Design and implementation

This chapter describes how we implement Skillicorn and Barnard's parsing algorithm on a GPU using interaction nets. First, we contribute a translation of the parsing algorithm into interaction nets (Section 3.1). Then, we extend the algorithm to parse LR grammars, thus expanding the set of languages our parser captures (Section 3.2). Finally, to run our interaction net algorithm, we provide a GPU implementation of an interaction net evaluator (Section 3.3).

3.1 Parsing with Interaction Nets

The Basic Structure of the Parser

To translate Skillicorn and Barnard's parsing algorithm [21] into interaction nets, we make the tree-like structure of the composition of sets of stack mappings concrete (Figure 3.1): We structure the parser interaction net as a binary tree where the leafs are the starting sets of stack mappings, and the branching nodes are subnets Γ performing the compositions between two sets of stack mappings. Parsing is done by performing all the interactions in this net until it is reduced completely.

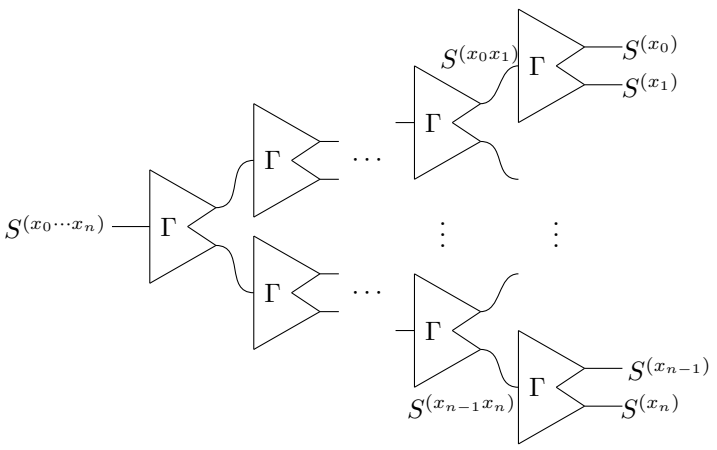


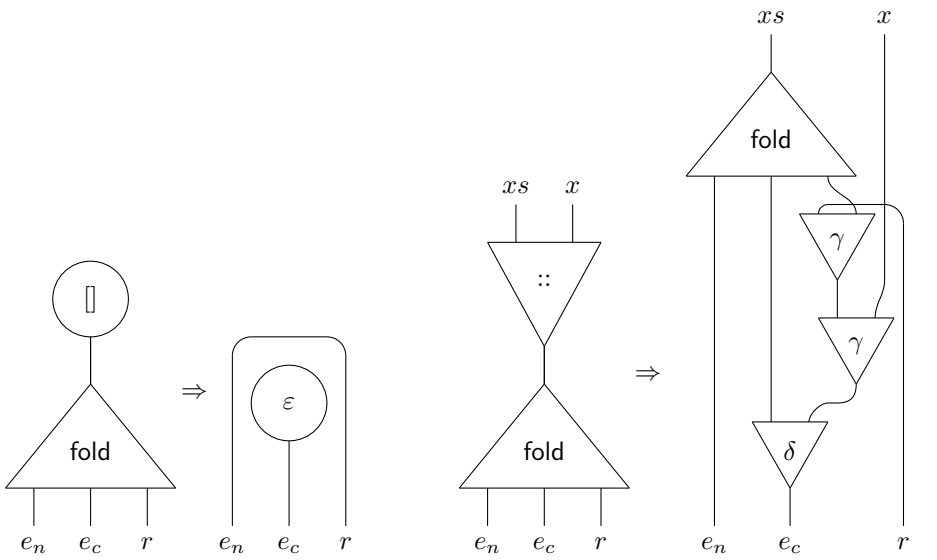
Figure 3.1: Basic structure of the interaction net parser

The implementation of the subnet Γ can be derived from translating a functional programming version of its algorithm using the interaction combinators [13] described in Section 2.1.3. For simplicity, we can start by assuming that all stack mappings are successful while ignoring the reduction rules alongside each stack mapping. Here and in our interaction net translation, we represent the sets of stack mappings as linked lists. Γ's algorithm now boils down to a pair of nested `fold` operations:

```
let compose_sets xs ys =
    fold xs [] (fn x acc_xs ->
        (fold ys [] (fn y acc_ys -> (x ∘ y) :: acc_ys)) @ acc_xs
    )
```

Listing 3.1: Composition of sets of stack mappings.

The composed stack mappings are combined together into a linked list, first using a cons operation in the inner loop and second using an append in the outer loop. We saw how to implement these two operations using interaction nets in Section 2.1.1. The interaction net version of `fold` follows closely from its functional definition given in Figure 3.2:



fold [] e_n $e_c = e_n$ fold $(x :: xs')$ e_n $e_c = e_c$ x (fold xs e_n e_c)

Figure 3.2: Interaction net definition of fold

When applying an empty list agent to a fold agent, the latter agent simply returns the base element e_n and deletes the net corresponding to e_c. When fold instead interacts with a :: agent, we first duplicate the term e_c since it appears twice on the right-hand side (rhs) of fold's definition: 1. in the outer application and 2. as an argument to the recursive fold. Additionally, γ agents are introduced to perform the applications e_c x (fold ...). The lower γ agent is passed x as the argument and the upper agent is passed the result of the inner fold. Using this representation for fold, we get the subnet Γ illustrated in Figure 3.3, which follows directly from Listing 3.1.

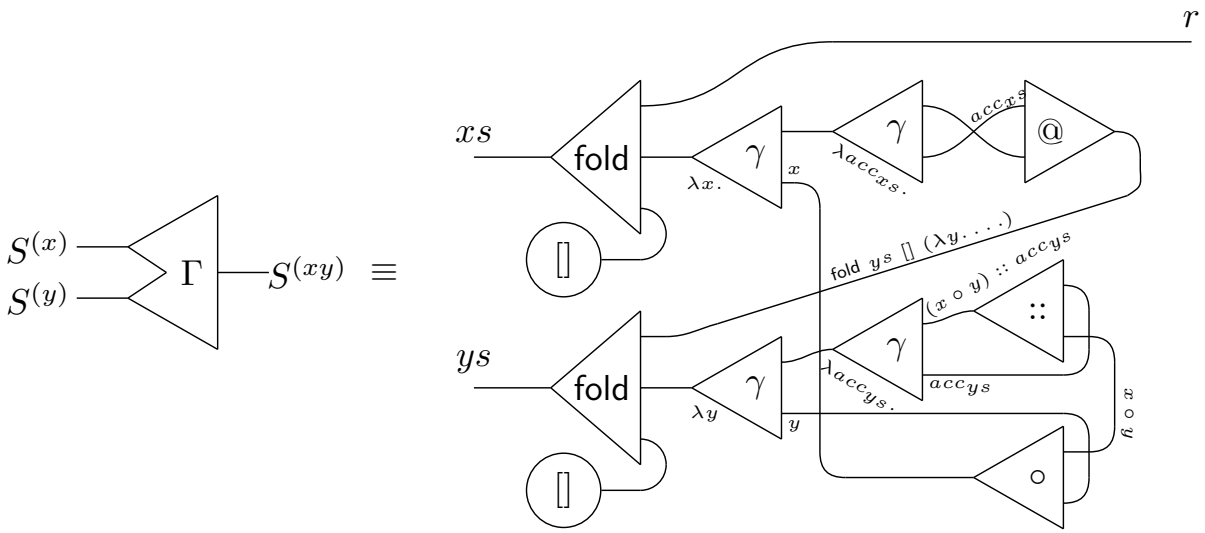


Figure 3.3: Interaction net composing sets of stack mappings xs and ys assuming all mappings compose.

Composing Stack Mappings with Interaction Nets

The parser just outlined requires an interaction net encoding for stack mappings and a definition of the interactions for the composition symbol '$\circ$'. To retain an efficient parsing algorithm, it is imperative that the conversion into interaction nets keeps Skillicorn and Barnard's algorithm's original $\mathcal{O}(\log n)$ time complexity [21]. Specifically, when composing two stack mappings, we want the comparisons between stack items to be done in constant time. Using standard interaction nets, this is not possible unless we define a unique interaction net type for each stack item. Given the limited amount of memory on a GPU and the large number of unique tokens that a programming language may have, this approach is not tractable. The alternative, keeping strictly to the standard definition of interaction nets, would be to encode stack items as Peano natural numbers. Comparing two such numbers would require m interactions (with m being the smaller of the two numbers compared), thus increasing the algorithm's time complexity.

Instead, we extend interaction nets by allowing agents to hold an integer value allowing us to have a singular stack item type SI that are distinguished by this new value. For example, two different stack items x and y would be represented by agents the same type SI holding different integer values. Conversely, two instances of the same stack item x would hold the same integer value.

Additionally, we define separate interaction rules depending on whether the interacting agents have the same value or not. Such an extension is a simplification of Hassan, et al's conditional rewrite rules that allow different interactions for an arbitrary set of disjoint conditional predicates [4]. Moreover, this extension preserves the determinacy property outlined in section 2.1.2: for any given interaction between a pair of agents, only one of the two interaction rules will apply.

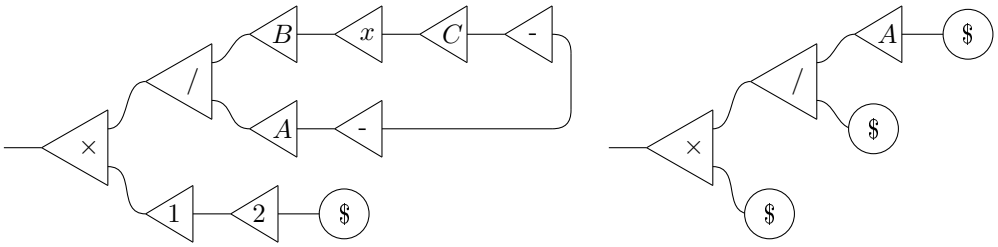


Figure 3.4: Stack mappings $A\text{-}/BxC\text{-}$ $(1, 2)$ and $/A$ encoded as interaction nets.

We can represent stack mappings as interaction nets as follows: Stack items are encoded as a 1-ary type SI as described above. To form a full stack, agents with type SI are chained as shown in Figure 3.4. Here, we illustrate the stack item agents with their corresponding symbol, but note that they all have the same SI type. The 'rest-of-the-stack' item '-' is given as separate a 1-ary type whose auxiliary ports are linked together at the tail of the input and output stacks. Stacks that do not end in a '-' item are instead capped with a 0-ary 'bottom-of-stack' agent '$'. An agent of the 2-ary type '/' connects the input and output stacks via its auxiliary ports. Additionally, stack mappings are accompanied by the reduction rules they encode, which are chained in the same way as the stack items. A stack mapping and its reduction rules are finally joined together with a pairing symbol '×'.

Previously, when describing the parsing algorithm in Listing 3.1, we made the simplifying assumptions that mapping compositions always succeeded and that the accompanying reduction rules could be ignored. We now drop these assumptions in Listing 3.2. First we check that the stack mapping composition succeeds in Line 5. If so, we append their accompanying rules together, and add the stack mapping composition to the set of composed stack mappings (Line 6). This `compose_sets` function is now translated into the interaction net in Figure 3.5.

```
1  let compose_sets xs ys =
2      fold xs [] (fn (R(x), M(x)) acc_xs ->
3          (fold ys [] (fn (R(y), M(y)) acc_ys ->
4              let (success, M(xy)) = x ∘ y in
5              if success then
6                  (R(x) @ R(y), M(xy)) :: acc_ys
7              else
8                  acc_ys
9          )) @ acc_xs
10     )
```

Listing 3.2: Complete composition of sets of stack mappings.

In this interaction net, the composition agent '∘' has two return values, implementing the behavior described in Algorithm 1. The first return value is a boolean indicating whether the composition was successful (`success`), and the second value is the composed stack mapping itself ($M^{(xy)}$). The boolean is directly fed to the principal port of an 'if' agent.

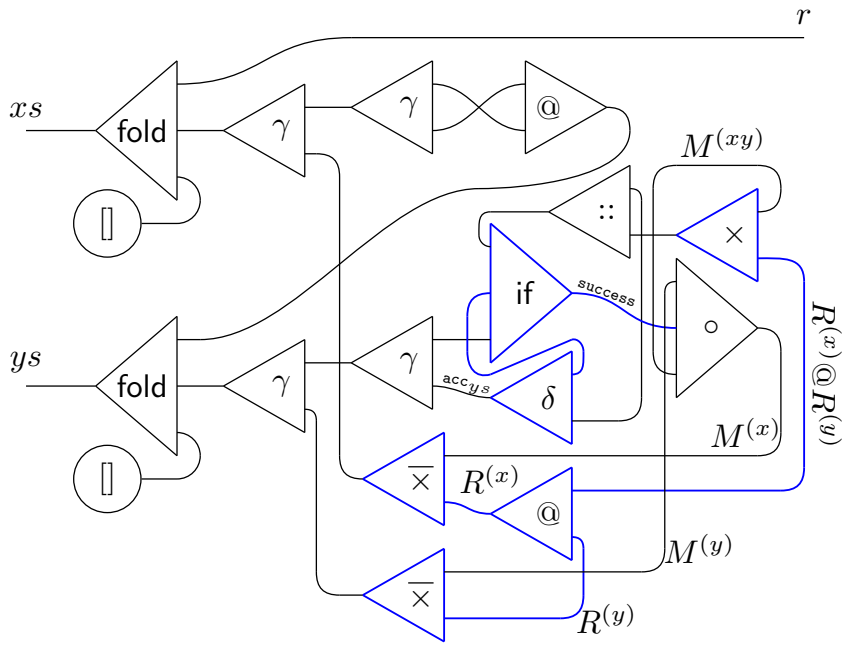


Figure 3.5: Interaction net composing sets of stack mappings xs and ys, with accompanying rule reductions. Blue elements are additions to Figure 3.3.

Then, based on the boolean, the 'if' agent wires either of its first or second auxiliary ports to the third output port. The first two auxiliary ports are in turn connected to interaction nets representing the the two branches of the conditional statement. The final addition to this composition net is the '$\overline{\times}$' agents which eliminate the '$\times$' agents, to retrieve the underlying values. The reduction rules $R^{(x)}$ and $R^{(y)}$ are passed to an '@' agent, which has been overloaded to also append chains of reduction rules.

Lastly, we define the interaction rules for the composition agent '$\circ$'. These are given at the top of Figure 3.6. The '$\circ$' and '$\circ'$' agents rewire the stack mapping net $M^{(x)}$ and $M^{(y)}$, to produce the output mapping $I^{(x)}/O^{(y)}$. These agents also introduce a ξ agent to which the inner stacks $O^{(x)}$ and $I^{(y)}$ are connected. This new agent checks that the inner stacks match, completing the composition. Additionally, a '$\bar{-}$' agent is sent through the outer stacks' chain of agents to remove the extra '-' at the end.

The ξ agent traverses the inner stacks $O^{(x)}$ and $I^{(y)}$ by alternating between the two. When a ξ agent interacts with some SI agent X in stack $O(x)$, it absorbs the SI agent by changing its own internal value to match that of the SI agent (Figure 3.6a). The ξ agent then proceeds to interact with the corresponding symbol in the stack $I^{(y)}$ and either eliminates it if their values match (Figure 3.6d) or returns false if they do not (Figure 3.6e). Once the ξ agent reaches a '-' symbol, the leftover stack is wired to be appended to its complement (Figures 3.6b, 3.6f). Here, and in the case where the 'bottom-of-stack' symbol is reached (Figure 3.6c), we also return true since the composition has succeeded.

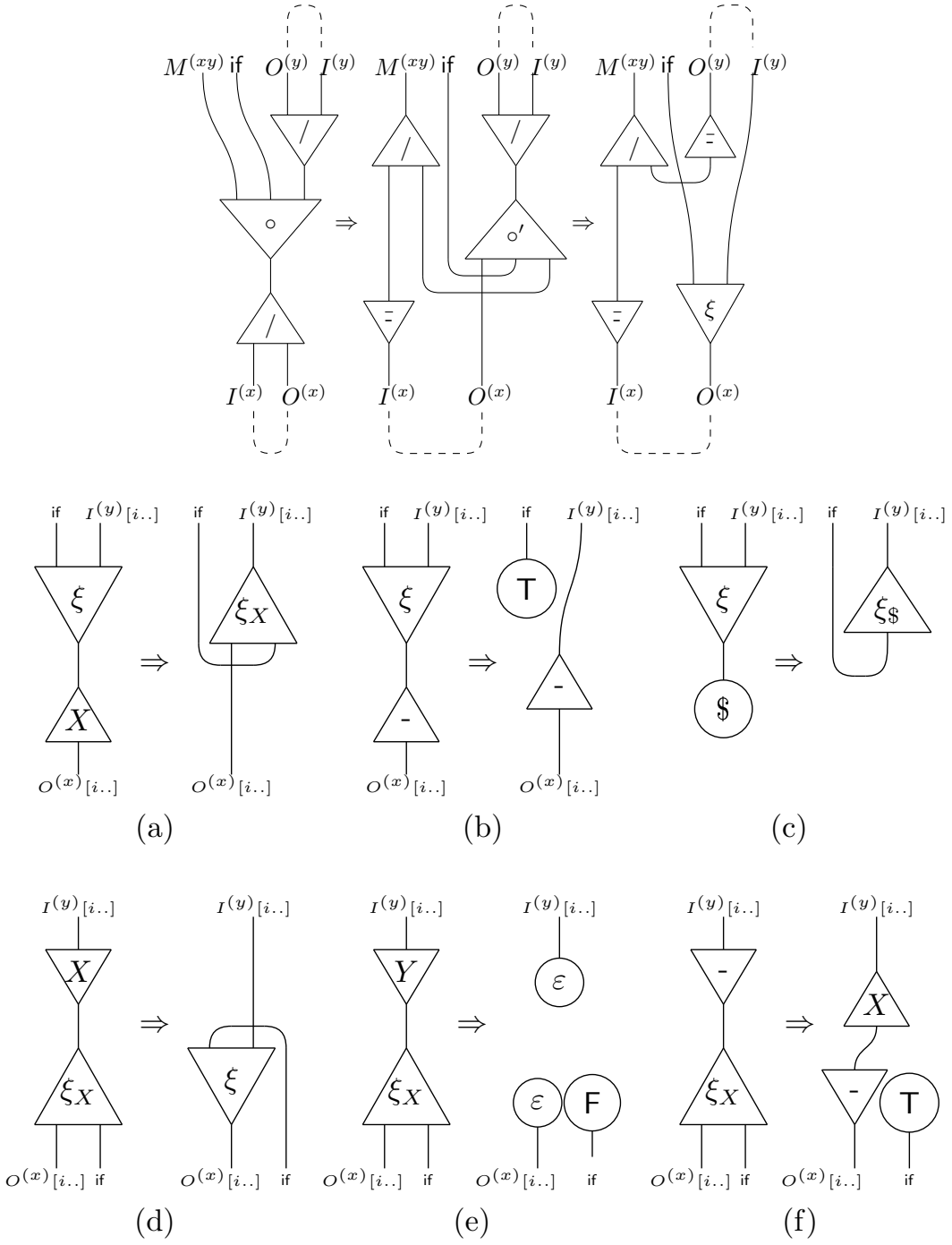


Figure 3.6: Interaction rules of agents involved in stack mapping composition. Interactions involving the agent $\xi_\$$ are analogous to those involving ξ_X.

State	Action							Goto		
	a	b	c	x	y	z	\$	A	B	C
0		s_3			s_4			1	2	
1							acc			
2				s_5						
3		s_3			s_4				6	
4				R_3						
5	s_8					s_9				7
6				R_2						
7							R_1			
8	s_8					s_9				10
9			R_5				R_5			
S			s_{11}							
T			R_4				R_4			

Figure 3.7: LALR(1) parsing table for the grammar in defined in Figure 2.5a, augmented with starting rule $S \rightarrow A\$$.

3.2 Parsing LR Grammars

The parsing algorithm introduced by Skillicorn and Barnard is limited to parsing LL grammars [21]. However, this category of grammars does not allow for left-recursive expressions, which are very common in programming languages (e.g. any left-associative operation). Therefore, to overcome this limitation, we extend Skillicorn and Barnard's method to parse a large subset of LR grammars.

Our approach to this extension follows Skillicorn and Barnard's methodology of translating a parsing table into stack mappings. However, the parsing tables and parsing states for LL and LR grammars function very differently. LR parsers generate their tables from sets of configurations that make up the states of a finite-state automaton. The details of how these are generated are not relevant to this paper, so we refer the interested reader to Aho, Ullman and Jeffrey [1]. Instead of the grammar terminals and non-terminals that LL parsers use in their stack, LR parsers use their automaton states.

Figure 3.7 shows the LALR(1) parsing table for the grammar defined in Figure 2.5a. We use a LALR(1) table solely for illustration as the results presented hold for all LR(1) grammars. Our goal is to produce the stack mappings that emulate the behavior of the parsing table. The LR Action table describes what action should be taken by the parser when it has a state n at the top of its stack, and the next input token is some token x. We can translate the actions into stack mappings as follows:

1. A shift action s_m simply adds the state m to the ToS. If this action is found at row n of the Action table, then this behavior can be translated into the stack mapping $n\text{-}/mn\text{-}$.

2. A reduction action that reduces rule $R_m : X \rightarrow \ldots$ first removes as many items off

a	b	c		x		y	z	$\langle$	$\rangle$	
5-/85-	0-/30-	95-/75-	(5)	2-/52-		0-/40-	5-/95-	/0	1-/	
8-/88-	3-/33-	98-/$S$5-	(5)	40-/20-	(3)	3-/43-	8-/98-		7_0-/10-	(1)
		S-/TS-		43-/63-	(3)				95-/75-	(5)
		T_5-/75-	(4)	6_0-/20-	(2)				98-/$S$5-	(5)
		T_8-/$S$8-	(4)	6_3-/63-	(2)				T_5-/75-	(4)
									T_8-/$S$8-	(4)

Figure 3.8: LR stack mappings for the grammar described in Figure 2.5a. Stack mappings for '$\rangle$' are derived from the '\$' token in the action table.

the ToS as there are symbols in R_m's rhs. Then, if the state p is the resulting ToS, we reference the goto table at row p and column X, adding the state found there to the stack. To translate such a reduction action, we introduce a new special stack symbol '_' that indicates the presence of any stack item. So, for the reduction of rule $R_2 : B \to b\ B$, we start with state 6 at the top, then pop the top two states, and finally, if the ToS is state 0 we push state 2, or if it is state 3 we push state 6. This gives us two stack mappings for either case: 6_0-/20− and 6_0-/63-.

Using this method, we derive the initial sets of stack mappings (Figure 3.8), however, in this state, they are insufficient for parsing. Let these initial stack mappings be called $S'^{(x)}$. We want each stack mapping in the final set $S^{(x)}$ to describe how the stack can be transformed when the input token 'x' is shifted. The reduction action stack mappings are incomplete because an LR parser does not shift the input when performing reductions. To derive the final set $S^{(x)}$, we need to find the closure of the initial set $S'^{(x)}$ under stack mapping composition.

To find the closure, we compose each reduction stack mapping in S' with all the other mappings in this same set. Additionally, we initialize S with a copy of every shift mapping in S'. If a reduction mapping is successfully composed with a shift mapping, we add the result to S. Conversely, if two reduction mappings are composed together, the result is added into S'. We repeat this process until no more unique stack mappings are generated.

In comparison to the sets of stack mappings generated for LL(1) grammars, the closure sets for LR stack mappings tend to be much larger. Even though our extension to parsing LR grammars retains the algorithm's $\mathcal{O}(\log n)$ time complexity, there will be a constant factor slowdown as the pairwise composition is done over larger sets. This problem can be mitigated by noticing that many of the stack mappings generated for the closure set cannot be composed with any other stack mapping. For example, in Figure 3.8, the stack mapping 6_0-/520- produced from the composition of x's first and fourth mappings, cannot be composed with any other mapping in the closure sets of this grammar. This stack mapping and others like it can, therefore, be removed since it will never be used in parsing.

Limitations of this approach

LR parsers are bottom-up parsers, meaning that they only perform a reduction once the entire rhs of the rule being reduced has been seen. Consequently, we can stack any number of right-recursive productions before the parser begins to reduce them all at the end. Thus, right-recursive grammars would produce an infinite number of stack mappings when their closure is calculated. Our example grammar has the right recursive rule $R_2 : B \to b\ B$. When parsing the string '*bbbbbbyxazc*', the parser pushes the state '3' onto the stack for each '*b*' it encounters and only begins to reduce these when it encounters the 'x' token. This repeating reduction of R_2 is represented in set $S'^{(x)}$ as the stack mapping $6\_3 - /63-$, which can be composed with itself any number of times. If we want to parse right-recursive grammars, we must be able to perform these mappings repeatedly for any number of right-recursive productions.

Our solution to this problem is to employ the Kleene star from regular expressions in our stack mappings. The Kleene star is wrapped around all the states that are removed when performing a right-recursive reduction. This allows us to remove any number of such states to account for any number of productions of the right-recursive rule. Using our running example, we have that (6_3-/63-) $\circ$ (6_3-/63-) = 6_33-/63-. This result indicates that for each reduction of R_2, a '3' state is removed from the stack. Adding the Kleene star to the resulting stack mapping allows us to repeatedly make these reductions as many times as needed before advancing the input. This gives the mapping 6_3(3)*-/63- $(2, 2^*)$. Note that the Kleene star should also be extended to the attached rule productions.

When composing stack mappings with a Kleene star, any stack items matching the star's contents are matched greedily. This strategy is safe because for an unambiguous LR grammar, there exists only one unique parse tree for a given input and, hence, only one unique sequence of reductions. Algorithm 3 describes in full the process of generating the closure of stack mapping sets extended by Kleene stars.

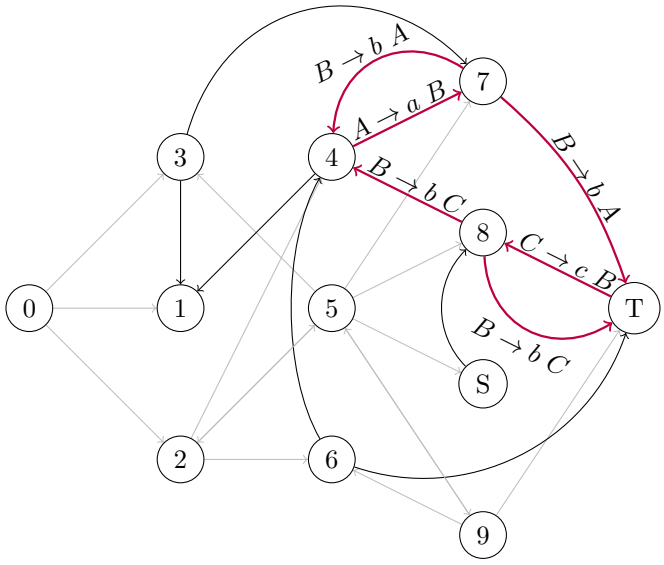
LR grammars not captured by our approach

Even with the Kleene star extension, some LR grammars remain that we cannot parse using this stack mapping algorithm. The inclusion of the Kleene star helps us capture those grammars that exhibit single loops of reductions. However, this extension is not powerful enough when we have overlapping loops of reduction chains. Figure 3.9 gives an example of a grammar and its corresponding state machine. The state machine has three loops that over lap with each other, marked red in the figure: $4 \to 7 \to 4$, $T \to 8 \to T$, and $4 \to 7 \to T \to 8 \to 4$. A LR table-parser is able to reduce any path through these overlapping loops without advancing the input. However, in our parsing method, this behavior would correspond to stacks described by the regular expression $(47|T8)^*$, which we cannot capture solely with a Kleene star.

We choose to disregard these edge-case grammars since an efficient implementation of the

R_0 : $S \to A\$$
R_1 : $A \to a\ B$
R_2 : $A \to x$
R_3 : $B \to b\ A$
R_4 : $B \to b\ C$
R_5 : $B \to y$
R_6 : $C \to c\ B$
R_7 : $C \to z$

(a)



(b)

Figure 3.9: LR grammar and a simplified version of its finite state machine. Gray arrows denote shift actions, black and red arrows denote reduce actions.

parsing algorithm hinges on performing quick compositions of stack mappings. Moreover, these grammars rarely appear in practice, so this restriction is of limited importance. Our parsing algorithm can, thus, parse all LR grammars except those with multiple looping reduction chains containing the same non-terminal. This is formally described as follows:

Let G be a LR grammar with non-terminals $A, X_1, \ldots X_m, Y_1, \ldots, Y_n$ for arbitrary m and n. Additionally, let $\alpha_1, \ldots, \alpha_m$ and $\beta_1, \ldots, \beta_n$ be arbitrary strings of terminals and non-terminals in G. Our parsing algorithm cannot parse grammars with the following production rules:

$$A \to \alpha_1\ X_1 \qquad \forall i \in [1, m-1].X_i \to \alpha_{i+1}\ X_{i+1} \qquad X_m \to \alpha_m\ A$$
$$A \to \alpha_1\ Y_1 \qquad \forall j \in [1, n-1].Y_i \to \beta_{i+1}\ Y_{i+1} \qquad Y_m \to \beta_n\ A$$

These rules all have a non-terminal as the rightmost symbol. Additionally, these non-terminals are found on the left-hand side of rules that chain together when they are reduced. We have two such reduction chains, both of which end up looping back to non-terminal A, thus, giving overlapping looping reduction chains.

3.3 Implementation in CUDA

Having translated the parsing algorithm into using interaction nets, and extending it to parse a broader category of grammars, we look to how we can implement the algorithm on a GPU using the CUDA framework with C++ [17]. Concretely, we implement a general evaluator for interaction nets that be used beyond our specific target application

of parsing.

As this implementation is targeted for GPUs, the design choices we make are heavily guided by their computer architecture. We have discussed the specifics of CUDA in Section 2.3, but in summary we aim to minimize branching control flow, make efficient usage of the limited memory resources, while carefully managing thread conflicts over a large shared data structure (i.e. the interaction net).

3.3.1 A Model of Execution for Interaction Nets

In this section, we describe an abstract machine for evaluating interaction nets. The aim is to describe interaction nets' graphical model of computation as sequences of instructions that can be emulated on a computer. It is this abstract machine that we then implement as a CUDA kernel to parse on GPUs. Therefore, the design choices made here consider the specifics of the CUDA architecture.

A programming language for interaction rules

Our initial aim when designing the abstract machine is to homogenize control flow. Rather than have one control path per interaction rule, we want the same control path for all interaction rules. Such an ideal is unrealistic, but we can get close by breaking down the interaction rules using a small set of operations. With a smaller set of operations, threads are given fewer opportunities to diverge as more of them will be performing the same operation. We define a language of three interaction rule operations (IRO): Creating new agents $\mathsf{new}(T, v)$, connecting ports $c_1 \leftrightarrow c_2$, and freeing agents $\mathsf{free}(A)$. The syntax for this language is given in Figure 3.10.

Rules	$R ::= T_1 \bowtie T_2 \Rightarrow \bar{o}$
Operations	$o ::= \mathsf{new}(T, v) \mid c_1 \leftrightarrow c_2 \mid \mathsf{free}(A)$
Values	$v ::= n \in \mathbb{N} \mid \mathsf{l} \mid \mathsf{r}$
Ports	$c_1, c_2 ::= A.i \mid A[i]$
Agents	$A ::= \mathsf{L} \mid \mathsf{R} \mid \theta_j$

Figure 3.10: Syntax for the language of interaction rule operations.

An interaction rule $T_1 \bowtie T_2$ is defined between two types T_1 and T_2 by specifying a list of IROs $\bar{o} = o_1, \ldots, o_n$ that perform the interaction. A $\mathsf{new}(T, v)$ operation creates a new agent of type T and value v. The value given to this agent can be a natural number, or it can be inherited from either of the left and right interacting agents (l and r respectively). The $c_1 \leftrightarrow c_2$ operation creates a connection between two ports. Ports are given numbered indices, with 0 being an agent's principal port and $1, \ldots, |T|$ its auxiliary ports, where $|T|$ is the type's arity. Moreover, we make a distinction between the port *belonging to*

$$\frac{}{\Theta, \Gamma \vdash} \qquad \frac{\Theta \cup \theta_{|\Theta|}, \Gamma \vdash \bar{o}}{\Theta, \Gamma \vdash \mathsf{new}(T, v), \bar{o}} \qquad \frac{\forall i \in [1, |\mathsf{type}(A)|].i \in \Gamma \qquad \Theta, \Gamma \vdash \bar{o}}{\Theta, \Gamma \vdash \mathsf{free}(A), \bar{o}}$$

$$\frac{i \leq |\mathsf{type}(A)| \qquad i \in \Gamma \qquad \Theta, \Gamma \vdash \bar{o}}{\Theta, \Gamma \vdash A.i \leftrightarrow c, \bar{o}} \qquad \frac{i \leq |\mathsf{type}(A)| \qquad \Theta, \Gamma \cup i \vdash \bar{o}}{\Theta, \Gamma \vdash A[i] \leftrightarrow c, \bar{o}}$$

$$\frac{\theta_j \in \Theta \qquad i \leq |\mathsf{type}(\theta_j)| \qquad \Theta, \Gamma \vdash \bar{o}}{\Theta, \Gamma \vdash \theta_j.i \leftrightarrow c, \bar{o}}$$

Figure 3.11: Well-formedness of interaction rules: Under a set of new agents Θ, and previous connecting indices Γ, the sequence of operations $\bar{o}$ is well-formed. Symmetric operations are judged the same way.

an agent $A.i$ and the port that the agent *is connected to* $A[i]$. In other words, agent A has a port $A.i$ that is connected to port $A[i]$ belonging to (most likely) some other agent. The agents we can reference in making these connections are the left and right interacting agents (denoted L and R) and the newly created agents θ_j, indexed by the order in which they are created. Finally, the $\mathsf{free}(A)$ operation frees the agent A.

In order for a rule's IRO encoding to be well-formed, the following conditions must hold:

1. Any connection involving a newly created agent $\theta_j \in \Theta$ must come after a new operation that creates it.

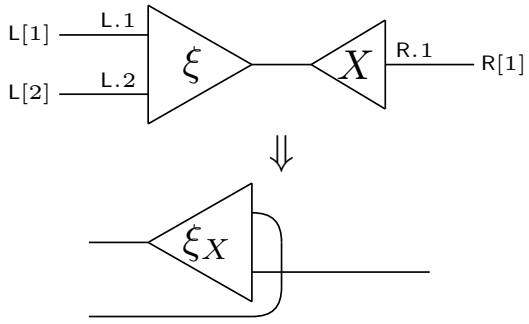
2. When referencing the ports of an agent A with type T, the index i used must be less than or equal to the arity of T.

3. A connection involving an agent's port $A.i$, must always come after the connecting port $A[i]$ has been rewired. Otherwise, if we rewire $A.i$, we lose the reference to the port $A[i]$, leaving a dangling connection.

4. A $\mathsf{free}(A)$ must come after all of the agent's connecting ports $A[i]$ have been rewired for $i \in [1, |\mathsf{type}(A)|]$. Otherwise, the agent to which port $A[i]$ belongs to will have an invalid connection to a freed agent.

These conditions are also given in Figure 3.11, which gives the full set of well-formedness rules for the judgment $\Theta \vdash \bar{o}$. An interaction rule $T_1 \bowtie T_2 \Rightarrow \bar{o}$ is well-formed if $\emptyset, \emptyset \vdash \bar{o}$.

To see how we encode an interaction described graphically into this language of interaction rule operations, we give an example in Figure 3.12. This example encodes the interaction between a stack item agent SI and the ξ agent used in composing stack mappings. The first operation in this interaction's encoding is a new operation where the new ξ_{SI} agent inherits its value from the SI agent. Then, all the connecting ports are rewired to this new node. Finally, the two interacting nodes are freed. Note that we also include a symmetrical interaction $\mathsf{SI} \bowtie \xi$ with instances of L/l swapped with R/r, and vice versa.



(a)

$$\xi \bowtie \mathsf{SI} \Rightarrow \mathsf{new}(\xi_{\mathsf{SI}}, \mathsf{r})$$
$$\mathsf{L}[1] \leftrightarrow \xi_0.0$$
$$\mathsf{L}[2] \leftrightarrow \xi_0.2$$
$$\mathsf{R}[1] \leftrightarrow \xi_0.1$$
$$\mathsf{free}(\mathsf{L})$$
$$\mathsf{free}(\mathsf{R})$$

(b)

Figure 3.12: Implementation of the interaction $\mathsf{SI} \bowtie \xi$ encoded into our language of interaction rule operations.

Execution model

Using the language of interaction rule operations, we can now describe an execution model for interaction nets. The abstract machine that we develop is similar to that of other interaction net evaluators such as that of Mackie and Sato [11], albeit optimized for execution on a GPU. The abstract machine consists of three data structures: First is the interaction network itself. Second, a queue is used to buffer any active interactions in the network that need to be dispatched for execution. Third, a table is initialized, mapping pairs of interacting types to their interaction rule's IRO encoding. The abstract machine then performs Algorithm 2 over these data structures to evaluate the interaction net until it is fully reduced.

The abstract machine first retrieves an interaction from the global queue of interactions and subsequently gets the list of operations corresponding to the interacting agents' types (Lines 2-3). The rest of the main loop's body performs the IROs. Note that to reduce the degree of branch divergence, we require all interaction rules to place all new operations before all connect operations and these before all free operations. In lines (5-8) we create all the new agents, which are subsequently appended to a list. Next, lines (10-18) describe how the connect action is performed. The connection arguments c_1 and c_2 are decoded to find the agents and the ports that need to be connected (11-12). In the case when one of the connection arguments is a new agent θ_j, we fetch the agent from the 'newNodes' list using the index j. The decoded agents are then connected. If the two agents' principal ports are connected, then we push a new interaction to the queue (15-17). Finally, in lines (20-26), we free the corresponding agents. Note that we never free new agents since these would never be created only to be immediately deleted.

Algorithm 2 Performing an interaction.

Data: Interaction net N. Interaction rule table R. Interaction queue Q.

1: **while** !Q.empty() **do**
2: $(l, r) = Q$.pop()
3: $\bar{o} \leftarrow R[l.\texttt{type}][r.\texttt{type}]$
4:
5: newNodes = []
6: **for all** $o \in \bar{o}$. $o = \mathsf{new}(T, v)$ **do**
7: newNodes = newNodes @ createNewAgent(T, v)
8: **end for**
9:
10: **for all** $o \in \bar{o}$. $o = c_1 \leftrightarrow c_2$ **do**
11: $$(a_l, p_l) \leftarrow \begin{cases} (l, i) & c_1 = \mathsf{L}.i \\ (r, i) & c_1 = \mathsf{R}.i \\ l.\texttt{ports}[i] & c_1 = \mathsf{L}[i] \\ r.\texttt{ports}[i] & c_1 = \mathsf{R}[i] \\ (\text{newNodes}[j], i) & c_1 = \theta_j.i \end{cases}$$
12: $$(a_r, p_r) \leftarrow \begin{cases} (l, i) & c_2 = \mathsf{L}.i \\ (r, i) & c_2 = \mathsf{R}.i \\ l.\texttt{ports}[i] & c_2 = \mathsf{L}[i] \\ r.\texttt{ports}[i] & c_2 = \mathsf{R}[i] \\ (\text{newNodes}[j], i) & c_2 = \theta_j.i \end{cases}$$
13:
14: connect(a_l, p_l, a_r, p_r)
15: **if** $p_l = 0, p_r = 0$ **then**
16: Q.push(a_l, a_r)
17: **end if**
18: **end for**
19:
20: **for all** $o \in \bar{o}$. $o = \mathsf{free}(a)$ **do**
21: **if** $a = \mathsf{L}$ **then**
22: free(l)
23: **else if** $a = \mathsf{R}$ **then**
24: free(r)
25: **end if**
26: **end for**
27: **end while**

3.3.2 Mapping the Abstract Model to the CUDA architecture

Structuring the evaluation kernel

Algorithm 2 gives an architecture agnostic description of how a thread performs an interaction using our evaluation model. When implementing this algorithm as a GPU kernel, we initially considered giving the kernel the same structure including the outer while-loop inside the kernel. This would mean that the kernel would only terminate once the interaction net has been fully evaluated, requiring only one call.

This approach, however, suffers from two main sources of overheads. Firstly, the number of threads performing interactions in parallel must be specified before the interaction net is evaluated. Since the number of parallel interactions varies throughout the execution of the interaction net, many threads are inevitably left spinning as they attempt to retrieve work from an empty interaction queue. This leads to GPU resources being split between spinning threads and those actually performing the interactions. Secondly, the added complexity of including the main loop in the kernel introduces points of contention. For instance, the interaction queue must be robust too having many threads attempting to retrieve the same interaction.

We can tackle these inefficiencies by moving the outer while loop from inside the kernel to the host program. Now, the host reads how many active interactions are on the queue and only launches as many threads as there are active interactions. It then waits until these threads finish before calling the kernel again. This "host-managed" execution strategy trades waiting for the threads to finish with ensuring that all active threads are performing useful work. Additionally, by synchronizing thread launches, we eliminate all contention when retrieving interactions from the queue since each thread can be mapped to a separate available interaction.

The main advantage of this approach is enabling a copying garbage collection scheme. Having a stop-the-world garbage collection scheme allows us to delay freeing memory until it is needed. For shorter parses, such a need may never arise. Additionally, we can remove free operations from the interaction rule IROs since the discarded agents are freed during garbage collection. Thus, the cost of having a separate garbage collection subroutine is offset by removing the branching divergence that occurs when performing free actions. Copying garbage collection also improves overall performance by allowing us to ignore all interactions involving ε agents since these perform the same function of freeing memory. Given how ε agents make up around 40% of the total number of interactions, manual garbage collection significantly contributes to cutting execution times.

Memory layout

A core part of optimizing GPU algorithms lies in how memory is managed and accessed. In Section 2.3, we saw how the CUDA architecture provides different regions of memory

Object	Location
Interaction net	Global memory
Queue of active interactions	Global memory
Queue of agents to copy for garbage collection	Global memory
Interaction rule operations table	Constant memory
Interaction type arity table	Constant memory

Table 3.1: Locations in device memory of objects used in our interaction net implementation.

with varying bandwidth and size [17]. The locations in memory for the various objects used in evaluating interaction nets are summarized in Table 3.1.

To execute the interaction net in parallel, we are required to allocate the interaction net in global memory. We cannot keep any partial copies of the net in shared memory, as this will lead to incoherent versions between thread blocks. This unfortunately means that most of our memory accesses will be done on the lower bandwidth global memory, presenting a key point of inefficiency for evaluating interaction nets on GPUs.

The queue of active interactions is also kept in global memory. An initial implementation stored part of the interaction queue in shared memory, employing work stealing to distribute the workload across thread blocks evenly. However, this scheme had a large overhead in branching divergence as cases needed to be introduced when stealing work or when spilling to global memory when the shared memory queue's capacity was reached. Additionally, as each interaction is only written and read once, the effect of storing these in shared memory would be minimal. A further advantage of the "host-managed" execution strategy is that we can have threads in the same block access interactions stored adjacently in memory. The GPU hardware is able to coalesce adjacent accesses to memory, improving overall access times.

One optimization we can make for memory allocation is with the accesses to the arity and interaction rule tables. These two tables are defined before the interaction net is executed, and so can be stored in the higher bandwidth constant memory. However, our parser interaction net uses 28 distinct agent types such that a $28 \times 28 \times 2$ table of lists of interaction rule operations does not fit in the 64 kilobytes of constant memory. In order to fit the table, we remove all the symmetrical entries.

3.3.3 Implementation Details

We have provided a description of how our abstract model of execution maps onto the CUDA architecture. Here, we give a more detailed view of how we represent interaction nets on the GPU and how we implement the execution model's IROs.

```
struct HeaderSegment {
  uint8_t type;
  uint16_t value;
  uint32_t lock;
}
```

```
struct PortSegment {
  uint32_t c_agent;
  uint32_t c_port;
}
```

```
union AgentSegment {
  HeaderSegment header;
  PortSegment port;
}
```

Figure 3.13: Representation of interaction net agents in C++.

Representing interaction nets on the GPU

Our abstract evaluation model makes no assumptions about how the interaction net is implemented. The main constraint that we have for this implementation is that we want our representation of interaction nets to accommodate copying garbage collection efficiently. This garbage collection scheme necessitates a memory-mapped model where all the agents are allocated inside a contiguous array of memory.

An additional constraint comes from the CUDA architecture that limits memory transfers to a maximum size of 128 bits. This restriction prevents us from reading entire interaction net agents in single-memory transactions. Since global memory accesses cost on average hundreds more clock cycles than arithmetic operations [9], we only want to load data that will be used in executing an interaction. We, therefore, split agents into smaller segments that fit within single memory transactions. Our implementation splits an agent into a header segment and one or more port segments. The header segment contains the agent's type and value, as well as a lock field used when resolving contention (Section 3.3.4). A port segment consists of the address of the agent being connected and the port on this agent to which the connection is made. We use nested C 'struct's inside a 'union' to represent each segment as shown in Listing 3.13. Note that instead of using a pointer for field `c_agent`, we use a 32-bit index into the global interaction net array.

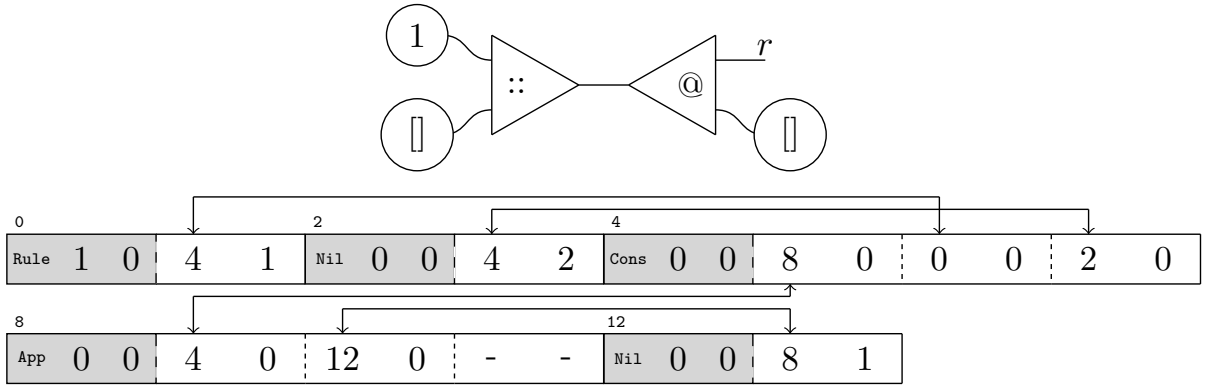


Figure 3.14: Example interaction net and its representation in memory using our encoding scheme in C++. Header agent segments are filled in gray.

Using this datatype, a complete agent is represented by an array of `AgentSegment`s whose length is determined by the arity of the agent's type. The first segment corresponds to the header, the second to the principal port, and any further segments represent the auxiliary ports in left-to-right order. This encoding scheme allows us to store the entire interaction

net as a large, contiguous array of agent segments for efficient garbage collection. To illustrate this encoding scheme, Figure 3.14 gives an example of an interaction net with its corresponding representation in memory.

3.3.4 Contention

Whenever a parallel algorithm operates over a shared data structure, there is a risk of contention between the executing threads. Even though our algorithm dispatches separate interactions to each parallel thread, there remains some overlap when performing some interactions simultaneously. In particular, we need to resolve the case where two adjacent interactions are being resolved in parallel. This scenario is illustrated in Figure 3.15.

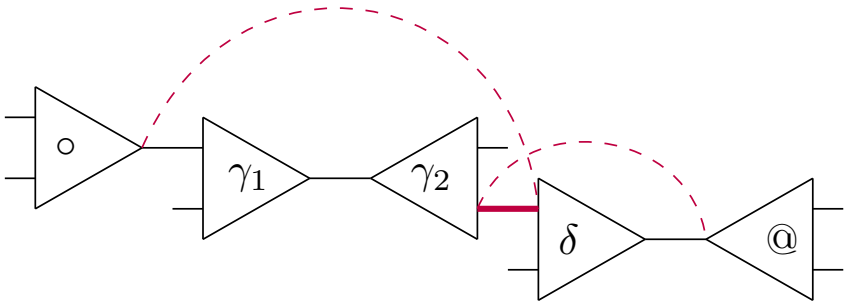


Figure 3.15: Example of an inconsistent interaction net produced by resolving adjacent interactions in parallel.

For our interaction net evaluator to be correct, it must preserve the interaction net's strong confluence property. This property requires that any interactions done in parallel be resolved as if they were done independently in series. Figure 3.15 shows how the port $\gamma_2[1]$ is being rewired at the same time as its connecting port $\delta[1]$. Depending on how the two threads read and write to the interaction net, we may get the inconsistent state illustrated in Figure 3.15. If we think of these connections as atomic transactions, an inconsistent state is produced when the transactions are performed in a non-serializable execution (Figure 3.16).

To enforce conflict-serializability, we employ a conservative two-phase locking mechanism. Before making a connection, threads must lock the agents that are being rewired and the agents whose connections are being replaced. For the example in Figure 3.15, thread 1 must acquire locks to the agents $\circ$, γ_1, γ_2 and δ, while thread 2 must lock agents γ_2, δ and @. This raises the question of deadlock, as the two threads are attempting to lock the same agents. We can resolve the deadlock by imposing an order on the locks acquired. Namely, we acquire the locks in order of the agents' indices on the global interaction net array. Only once all the locks have been acquired does a thread attempt to make the connection.

Any threads that fail to acquire all their locks must try making the connection again. By

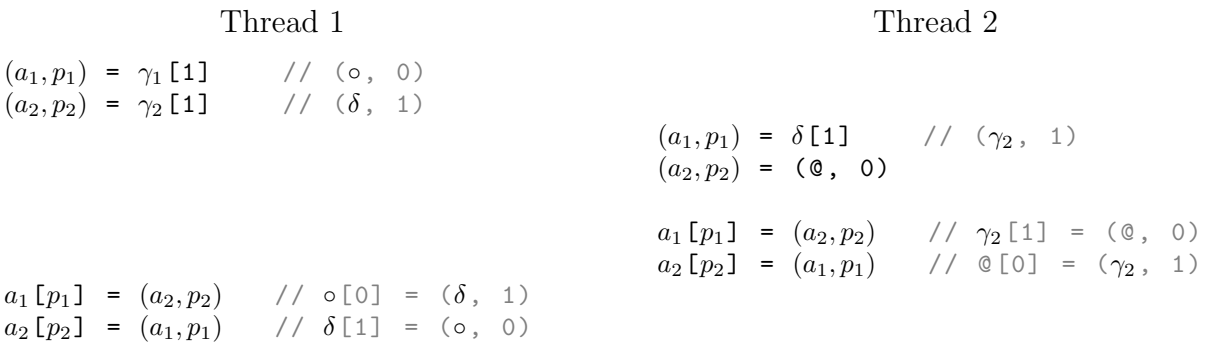


Figure 3.16: Non-serializable execution of parallel adjacent interactions.

keeping locks strictly for the duration of the connection, we can minimize opportunities for failure. This scheme allows concurrent threads to be executed more efficiently than if the locks were acquired for the duration of the entire interaction. For example, the scenario illustrated in Figure 3.15 only has conflicting operations for the pair of connections shown. The rest of the connections involved in performing the two interactions do not conflict in any way. This allows both threads to perform their interactions in parallel, without one having to wait for the other to finish.

3.3.5 Implementing the Interaction Rule Operations

Having addressed the potential issues arising from contention, the final element of the abstract execution model left to implement is the interaction rule operations. Note that by virtue of our copying garbage collection scheme, we do not have to implement a free operation.

The new operation

To create a new agent, we need to reserve space on the global interaction net array. We include a global pointer `net_end` to the end of the allocated space on the contiguous interaction net array. When a new agent is created, we increment this pointer by the number of segments required to allocate the agent. For example, a δ agent of arity 2 takes up 4 segments in total: 1 header segment, 1 segment for the principal port, and 2 segments of the auxiliary ports. However, incrementing this pointer is a clear source of contention as it is shared among all active threads. Therefore, each increment operation must be done using CUDA's atomic addition function.

Listing 3.3 presents this subroutine for the new IRO. In this listing, `net` denotes a *AgentSegment* pointer to the start of the interaction net's array; `left` and `right` are the interacting agents; and `ARITIES` is the table of interaction type arities.

```
1  AgentSegment *createNewAgent(Value v, Type t) {
2    AgentSegment *new_agent = net + atomicAdd(net_end, ARITIES[t]);
3    if (v == L)
4      new_agent[0].header = {t, left[0].header.value, 0};
5    else if (v == R)
6      new_agent[0].header = {t, right[0].header.value, 0};
7    else
8      new_agent[0].header = {t, v, 0};
9
10   return new_agent;
11 }
```

Listing 3.3: Subroutine for the new operation.

The connection operation

The code that performs a connection operation is given in simplified form in Listing 3.4. Here, we also give the relevant functions needed for the two-phase locking mechanism. Locking an agent can be done with an atomic compare-and-swap (CAS) operation (lines 1-3). A thread checks that the agent is unloxked by checking if its lock value is zero. If so, it sets the locks value to the thread's a unique ID. Checking if a lock has been acquired is as simple as seeing if the agent's lock value matches the thread's ID (lines 7-9). To unlock an agent, we perform the reverse compare-and-swap, where we check that the lock value matches the thread ID and swap in a zero (lines 4-6).

When making the connection, we first check that the agents we are connecting are not the interacting agents themselves but rather the surrounding connecting agents. This is the distinction between a $A[i] \leftrightarrow c_2$ and a $A.i \leftrightarrow c_2$ connection operation. In the former case, we need to lock the agent A and its connecting agent $A[i]$ (lines 16-26). This is done in the order of their indices as not to deadlock with other threads. Once the thread has attempted to take the necessary locks, we check that they have been successfully acquired before making the connection (lines 37-39). If the connection is between two principal ports, we add them to the interaction queue. This is done by atomically advancing the tail of the queue before inserting the new interaction (lines 41-42). Finally, the thread releases all the locks it has acquired at the very end (lines 45-48).

```
void lock(AgentSegment *agent) {
  atomicCAS(&agent[0].lock, 0, threadId);
}
void unlock(AgentSegment *agent) {
  atomicCAS(&agent[0].lock, threadId, 0u);
}
bool is_locked(AgentSegment *agent) {
  return agent[0].lock == threadId;
}

AgentSegment *connectAgents(Connection c_1, Connection c_2) {
  AgentSegment *a_1, *a_2;
  uint32_t p_1, *p_2;

  if (c_1 == `A[i]`) {                          // A[i] ↔ c_2 connection
    if (A - net < A[1 + i].port.c_agent) {      // Lock A and A[i] in order
       lock(A);
       if (is_locked(A))
         lock(net + A[1 + i].port.c_agent);
    } else {
       lock(net + A[1 + i].port.c_agent);
       if (is_locked(a_1))
         lock(A);
    }
    a_1 = A[1+i].port.c_agent;
    p_1 = A[1+i].port.c_port;
  } else {                                      // A.i ↔ c_2 connection
    lock(A);
    a_1 = A;
    p_1 = i;
  }

  if (c_2 == `B[j]`) {                          // Similar code for c_2
    ...
  }

  if (is_locked(A) && is_locked(a_1) && is_locked(B) && is_locked(a_2)) {
    a_1[1 + p_1].port = {net - a_2, p_2};
    a_2[1 + p_2].port = {net - a_1, p_1};

    if (p_1 == 0 && p_2 == 0)
      interactionQueue[atomicAdd(InteractionQueue.tail(), 1)] = {a_1, a_2};
  }

  unlock(A);
  unlock(a_1);
  unlock(B);
  unlock(a_2);
}
```

Listing 3.4: Subroutine for connection operations.

Chapter 4

Evaluation

We evaluate the performance of our interaction net parsing algorithm running on our GPU interaction net evaluator. Specifically, we aim to discover whether interaction nets provide an avenue for enhancing the level of parallelism for inherently linear tasks like parsing. This will give evidence for knowing whether interaction nets' inherent parallelism provides a useful abstraction for general computing on a GPU.

Experimental Setup

The experiments we perform in this evaluation were done on a machine with a Intel Xeon Gold 6142 2.60GHz CPU and an NVidia Tesla P100 accelerator with 16GB of device memory. It is worth noting the Tesla P100 can support up to 64 warps per multiprocessor (i.e. $32 \times 64 = 2048$ active threads) [16]. Moreover, as each multiprocessor has 65536 registers that can be allocated to a thread block, for full GPU occupancy, we need our kernel to use $65536 \div 2048 = 32$ registers or fewer. Unfortunately, when compiled our kernel uses 38 registers per thread, thereby not reaching maximum occupancy.

Additionally, we limit the size of the interaction net array to 512MiB. Note that the actual amount of memory used is doubled since we require two such arrays to implement copy-collection.

4.1 Assessing the Interaction Net Evaluator

Before evaluating the efficacy of our parsing algorithm, we assess the performance our GPU interaction net evaluator. We do this by running an implementation of Ackermann's function in unary numbers as an interaction net. Our results are compared with 'Inpla', the state-of-the-art interaction net evaluator running on the CPU [20]. We perform 5 runs of Ackermann(3, n), for $n \in [1, 10]$. The implementation of the Ackermann interaction net is the same for the two evaluators, both performing the same number of interactions for each run. This experiment's results are presented in Figure 4.1.

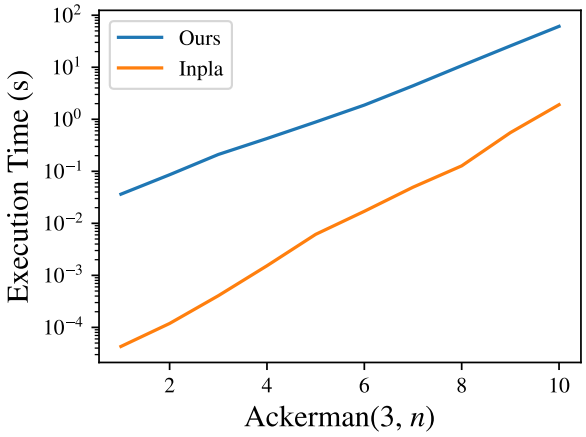


Figure 4.1: Execution times of our interaction net evaluator vs. Inpla on Ackermann(3,n). Modeling exponential regression on the two sets of results gives $\log(y) = 0.813x - 4.117$ for our implementation and $\log(y) = 1.185x - 11.276$.

When evaluating the Ackermann interaction net, our implementation vastly underperforms in comparison to Inpla. As is discussed in Section 3.3.2, the poor performance is likely due to expensive global memory accesses. Nevertheless, if we focus on the trends of the two sets of results by comparing the regression coefficients, we see that our GPU implementation deteriorates slower with increasing inputs. This supports the idea that interaction nets are massively-parallel models of computation, as the larger interaction nets we evaluate can make better use of the many threads on a GPU.

In comparison to other parallel interaction net evaluators, our implementation performs relatively well. Jiresch's reports a time of 30.4 seconds for Ackermann(3,7) using their own GPU interaction net evaluator [6]. This is significantly worse than our implementation's time of 4.4 seconds. As their evaluator is implemented using the deprecated Thrust library, we were unable to verify their results on equal hardware. Kahl's parallel CPU implementation in Haskell beats our time on Ackermann(3,7) with a time of 1.5s when running on 5 threads [8]. However, on larger interaction nets, such as that for Ackermann(3,10), their implementation falls behind, giving a time of 110.8s in comparison to our 61.2s.

Memory usage and parallelism

This experiment, as well as those described below, was performed using increasing input sizes until the memory limit for the interaction net array was reached at 512MiB. However, even the largest inputs tested could be performed trivially on a CPU. For example, the largest parsing input tested was only 365 input tokens, a far cry from the hundreds of lines of code we expect from programming files. This indicates that the memory requirements of interaction nets do not scale well with input size, even for simple programs like Ackermann's function. Such severe memory usage prevents us from fully unlocking the potential of interaction nets' parallelism:

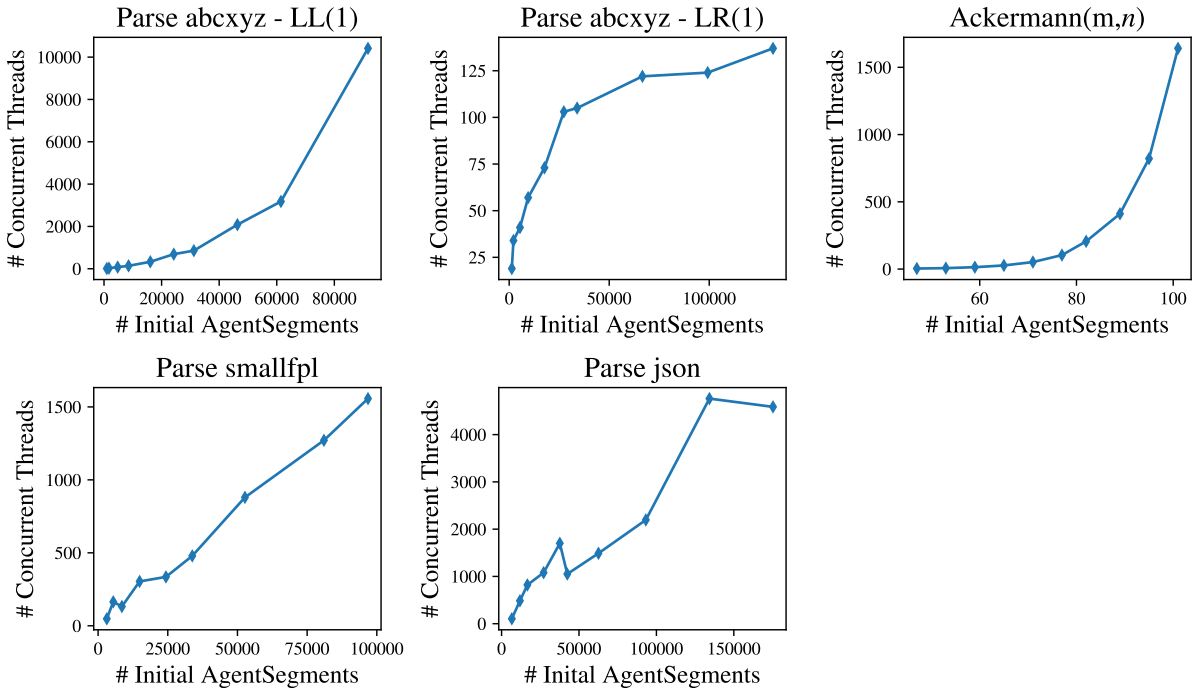


Figure 4.2: Average parallelism for varying initial interaction net sizes.

In Figure 4.2, we plot initial interaction net sizes versus the average number of concurrent threads in the interaction net's execution. This average is taken over the number of threads launched for each call of the interaction kernel. This data shows how the amount of parallelism is, in part, dependent on the algorithm implemented. Parsing LL(1) grammars and running the Ackermann function seem to exhibit exponential growth in the number of available parallel interactions. On the other hand, the LR(1) grammars see linear increases in parallelism with input size, with the exception of the abcxyz grammar, which grows logarithmically. The reasons for this anomaly are discussed below. Overall, our implementation of an interaction net evaluator is bounded by memory. Even with the largest number of concurrent threads, our implementation has not come close to reaching the Tesla P100's limit of 7.34 million active threads [16].

4.2 Assessing the Parser Implementation

Having implemented our interaction net parser in the framework of our interaction net evaluator, we assess the parser using a variety of different grammars. Specifically, we test our parser's performance on the grammar defined in Figure 2.5a (which we now name "abcxyz"), providing two translations of this grammar into stack mappings: 1. following Skillicorn and Barnard's methodology for LL(1) grammars, 2. using our extension of this algorithm for LR(1) grammars. Additionally, we also test our parser on a small grammar of functional programming language expressions and a full Json grammar. Neither of these two grammars is LL(1) so we only test their LR(1) parser implementations. We execute these parsers over a range of input sizes, performing 5 runs for each input. The

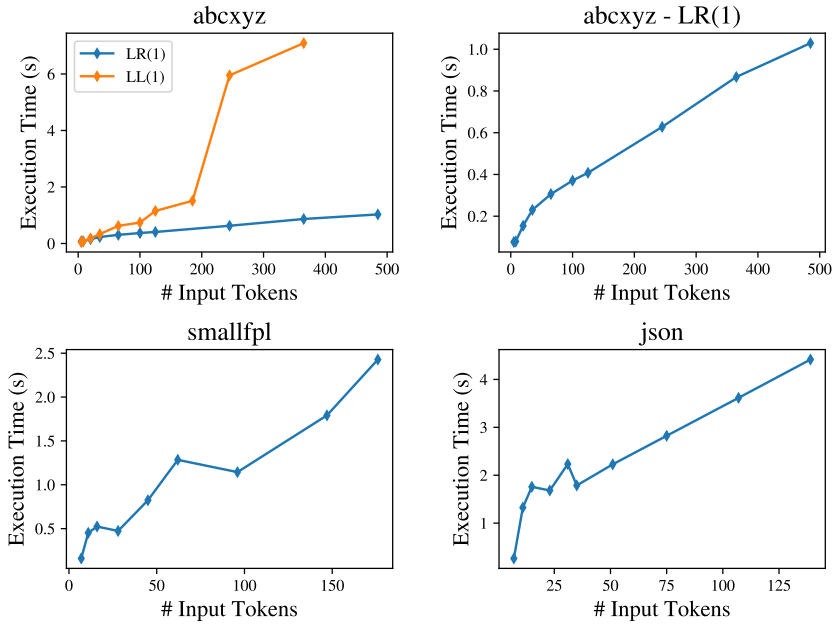


Figure 4.3: Parse times from our interaction net parser over 4 different grammars. The results of the abcxyz LR(1) grammar are included twice for comparison with the LL(1) grammar.

average times of these tests are given in Figure 4.3.

The results for the LR(1) grammars appear to be consistent with each other. We see an initial sharp increase in execution time as the interaction nets corresponding to smaller inputs do not have sufficient parallelism to compensate for the upfront costs of dispatching to a GPU. For inputs of size greater than 30, the parse times of the LR(1) grammars appear to follow a linear trend, except for an anomalous point in each of the smallfpl and json graphs. The trend implies a worse time complexity than the theoretical $\mathcal{O}(\log n)$ of Skillicorn and Barnard's parsing algorithm. It appears when translating the algorithm into interaction nets, the time complexity was not preserved.

Regarding the anomalous points, it should be noted that they appear in both Figure 4.2 and Figure 4.3. This suggests that the input strings' content influences how parallel the parser's execution will be. Therefore, parse times are not linked only to input length but also to input content. This is due to the uneven size of the sets of stack mappings that correspond to each input token. For example, in Figure 3.8, we can see how the tokens x and c have larger stack sizes than a or b. Thus, it follows that input strings with proportionally more x and c tokens will take longer than other strings of the same size with fewer of these tokens. For the smallfpl and json results, the anomalous points must reflect inputs that exhibit a higher degree of parallelism than the average input string.

The LL(1) grammar results imply a worse-than-linear time complexity. Somewhat surprisingly, the 'abcxyz' grammar performed worse for all but the smallest two inputs when parsed as an LL(1) grammar, than as an LR(1) grammar. This is also despite the fact that the initial parser interaction nets were smaller for the LL(1) grammar across all

LR(1) Grammar	% δ-interactions
abcxyz	76.06% ± 2.12%
smallfpl	88.03% ± 4.68%
json	91.97% ± 2.51%

Table 4.1: Percentage of total interactions that had a δ interacting agent for the LR(1) parses. Values are shown with their standard deviations.

input sizes. The reason for this result comes from our addition of the Kleene star in our LR extension to Skillicorn and Barnard's algorithm. With the Kleene star, all right-recursive reductions are performed inside a single stack-mapping composition. This is more efficient than composing multiple stack mappings for each input token, as the LL(1) grammar does. Given how the inputs for this grammar were generated such that half of the reductions were due to the right-recursive rule R_2, we can see the improvement in performance that the Kleene star stack mapping symbol provides. However, the Kleene star may be detrimental in the long run. The execution of a single Kleene star stack composition is much more linear than performing multiple stack compositions in parallel. Thus, for larger inputs, the increase in efficiency that Kleene star introduces is lost. This phenomenon is reflected as logarithmic growth in parallelism as shown in Figure 4.2. The other LR(1) grammars do not make use of Kleene star stack mappings as extensively as the abcxyz grammar and therefore do not share this trend.

4.2.1 Investigating the Sub-Optimal Time Complexity

To find the reason for our parser's sub-optimal time complexity, we investigate the kinds of computations being performed. Having deleted all ε agents from our interaction nets by virtue of our garbage collection scheme, the only agents not directly involved in advancing the parsing algorithm are the duplicating δ agents.

Table 4.1 shows the percentage of interactions involved in the execution of the LR(1) parsers that had a δ interacting agent. This table clearly shows that the majority of work done in evaluating the interaction nets is allocated to copying portions of the nets instead of composing stack mappings. Since our parsing interaction nets grow linearly with input size, it therefore follows that our interaction net parsing algorithm has a linear time complexity if it spends the majority of computation copying portions of the net. Additionally, the results in Table 4.1 provide further evidence that the Kleene star is involved in keeping the abcxyz LR(1) parser's execution times low. The lower proportion of δ-interactions for this parser is exchanged for interactions that resolve Kleene star compositions.

Finally, we note that the proportion of δ-interactions performed in executing the LL(1) parser increases logaritwith input size. It is for this reason that it was excluded from Table 4.1, as all the LR(1) parsers keep a constant proportion of δ-interactions. The proportion

of δ-interactions for the LL(1) parser increases logarithmically from 74.74% to 0.9810%. This relative increase in δ-interactions explains why the execution time for the LL(1) parser grows at a worse than linear time complexity. The source of this phenomenon is likely grammar dependent, given how the underlying parsing mechanism is the same between the LL(1) and LR(1) parsers.

Chapter 5

Related work

Parallel parsing algorithms have existed for nearly as long as their non-parallel counterparts [15, 14]. However, it has only been with recent hardware improvements that we can take advantage of the available parallelism these algorithms provide. When it comes to parsing on GPUs, current research has focused on grammars that are either too simple for defining programming languages, or too complex.

In the former category, we have GPU parsing solutions for simply structured data files such as CSV and OBJ files [22, 18]. The regular structure of these file types allows for much more direct massive parallelism, deriving considerable improvements in performance over CPU implementations.

In contrast, within the field of natural language processing, there has been research in parsing arbitrary CFGs on GPUs [25, 2, 7, 3]. The standard parsing algorithms for the full class of CFGs run in polynomial time with the size of the input, instead of linear time for the LR grammars commonly used to define programming languages. GPU parsing algorithms for natural language CFGs, therefore, have greater room for improvement over CPU implementations. Specifically, Hall, et al.'s [3] GPU parser is the current state-of-the-art, improving parsing times significantly over a CPU implementation.

We also mention Voetter's master thesis in which he implements a compiler's front end on GPUs [24]. His approach is limited in the scope of grammars that his implementation is able to parse, namely being restricted to a subset of LL grammars. Our parallel parsing algorithm, is the first (to the author's best knowledge) to parse the full class of LL grammars, and a large subset of LR grammars on a GPU.

In relation to interaction nets, this model of computation has seen a renewed interest in recent years with the HVM project, which uses interaction nets as the backend target for a virtual machine [23]. Their virtual machine compiles a functional programming language into the interaction combinators we describe in Section 2.1.3. Through the abstraction of the interaction combinators, they are able to provide a virtual machine that can run

generic code on a GPU.

The HVM is still currently underdevelopment and, therefore, has some unresolved issues. In particular, certain λ-expressions cannot be evaluated in their programming model, with our stack mapping composition algorithm (Listing 3.2) being one such case. Additionally, the HVM restricts the set of interaction net agents to the interaction combinators, instead of allowing for arbitrary definition of agents. This limitation necessarily increases the time complexity of many algorithms in the same way that the λ-calculus does not provide an optimal backend. In contrast, we can define arbitrary types in our interaction net evaluator, allowing us to translate a broader range of traditional algorithms into interaction nets while preserving their time complexity.

Recent parallel interaction net evaluators include those of Sato [20], Jiresch [6] and Kahl haskell. Out of these, Jiresch's approach is the only one that targets GPUs. Our implementation improves upon his results, as discussed in Section 4.1. Sato and Kahl provide parallel CPU evaluators that can execute interaction nets on up to 8 parallel CPU threads. Sato's results are by far the best, being the current state-of-the-art interaction net evaluator.

Nevertheless, further research needs to be done in the practical application of interaction nets. Current research exclusively evaluates interaction nets' performance and parallelism on simple benchmarking algorithms such as sorting algorithms or Ackermann's function [12]. In this paper we present the first implementation of a complex and practical algorithm using interaction nets, namely LL and LR parsing. Based on our results, we conclude that evaluating interaction nets is too costly to provide an effective abstraction for executing complex algorithms efficiently

Chapter 6

Conclusions and Further Work

Based on our evaluation, we find that interaction nets do provide parallelism that scales well with problem size. Our results agree with those of Mackie and Sato that certain algorithms, like the Ackermann function over unary numbers, gain a free parallel implementation when using interaction nets [12]. In the context of parsing, our parallel parsing algorithm also benefits from using interaction nets insofar as parallelism is concerned. Across all grammars tested, the number of concurrent threads executing the parser increases with input size.

This result is called into question however, when we consider the overwhelming overheads that interaction nets introduce. From the analysis above, we identify two main flaws with using interaction nets for complex computations. First, interaction nets require an immense amount of memory for even small input sizes. Second, the majority of their execution is spent duplicating parts of the interaction net. In fact, upwards of 90% of the time spent parsing is spent resolving δ-interactions! These two factors combined make interaction nets an unviable model of computation for complex algorithms. Interaction nets' limitations are reflected in our parsing algorithm's poor performance, being unable to parse more than a few hundred input tokens for simple grammars.

It therefore follows, that to improve upon the ideas developed in this project, further work must primarily tackle these limitations. Although we cannot extricate the interaction net model of computation from these two limitations we can seek to optimize out the negative effects they have as much as possible. A starting point for such an improvement would be to interaction nets' layout in memory to coalesce δ-interactions as a single copy of a chunk of memory. However, such an improvement does not solve the memory limitations. To solve this problem we would have to allow agents to reference parts of the interaction net, instead of duplicating them entirely. This must be done while preserving the determinacy and strong confluence properties that enable interaction nets' parallelism.

Beyond these general optimizations for interaction net evaluators, there are several additional improvements which can be made to our specific implementation:

Firstly, the locking mechanism can be ameliorated. Instead of locking whole agents, and subsequently all their ports, we could adjust our approach to lock individual ports. Alternatively, The HVM team has developed a lock-free method to resolving adjacent interactions. This method was rejected because it is tailored to the specific set of interaction combinators they use, while increasing the amount of branching. Further work could therefore be done to try and extend the HVM approach to encompass arbitrary interaction agents, so that this alternate route to dealing with contention can be evaluated within our interaction net evaluator.

Secondly, our current implementation of interaction nets does not fully utilize the resources available on the GPU. Therefore, our interaction net evaluator could be further optimized by breaking the kernel into smaller operations. For example, we could distribute the individual connections involved in an interaction across different threads so that these are performed in parallel.

Thirdly, the main reason why our interaction net evaluator underperformed compared to Inpla, is due to how our evaluator almost exclusively makes accesses to global memory. Making use of faster shared memory accesses would therefore provide a substantial benefit. However, implementing this would, in all likelihood, require some form of coherency mechanism to keep the interaction net consistent, while preserving the determinacy and strong confluence properties.

Lastly, this dissertation implements Skillicorn and Barnard's algorithm using interaction nets while extending it to LR grammars. This work was completed successfully, with the LR extension providing faster parse times for right-recursive grammars. Nevertheless, it remains unclear whether Skillicorn and Barnard's algorithm is an efficient algorithm for parsing on massively-parallel devices without the use of interaction nets. It would therefore be worthwhile to investigate the effectiveness of Skillicorn and Barnard's algorithm implemented directly on a GPU. Such an implementation is likely to yield better results than our interaction net implementation presented in this paper, perhaps being comparable to state-of-the-art parsing methods.

In conclusion, while the inherent parallelism that interaction nets provide is enticing, the reality is that as a model of computation they are far too costly for practical application.

Bibliography

[1] Aho, A. V., and Ullman, J. D. *The theory of parsing, translation, and compiling.* Prentice-Hall, Inc., USA, 1972.

[2] Canny, J., Hall, D., and Klein, D. A multi-teraflop constituency parser using GPUs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing* (Seattle, Washington, USA, Oct. 2013), D. Yarowsky, T. Baldwin, A. Korhonen, K. Livescu, and S. Bethard, Eds., Association for Computational Linguistics, pp. 1898–1907.

[3] Hall, D., Berg-Kirkpatrick, T., and Klein, D. Sparser, better, faster gpu parsing. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (2014), pp. 208–217.

[4] Hassan, A., Mackie, I., and Sato, S. Interaction nets: programming language design and implementation. *Electronic Communications of the EASST 10* (Jul. 2008).

[5] Hillis, W. D. *The connection machine.* MIT Press, Cambridge, MA, USA, 1989.

[6] Jiresch, E. Towards a gpu-based implementation of interaction nets. In Proceedings 8th International Workshop on *Developments in Computational Models,* Cambridge, United Kingdom, 17 June 2012 (2014), B. Löwe and G. Winskel, Eds., vol. 143 of *Electronic Proceedings in Theoretical Computer Science*, Open Publishing Association, pp. 41–53.

[7] Johnson, M. Parsing in parallel on multiple cores and GPUs. In *Proceedings of the Australasian Language Technology Association Workshop 2011* (Canberra, Australia, Dec. 2011), D. Molla and D. Martinez, Eds., pp. 29–37.

[8] Kahl, W. A simple parallel implementation of interaction nets in haskell. In *Proceedings Tenth International Workshop on Developments in Computational Models, DCM 2014, Vienna, Austria, 13th July 2014* (2014), U. D. Lago and R. Harmer, Eds., vol. 179 of *EPTCS*, pp. 33–47.

[9] Kirk, D. B., and Hwu, W.-m. W. *Programming Massively Parallel Processors: A Hands-on Approach*. Applications of GPU Computing Series. Morgan Kaufmann Publishers, Burlington, MA, 2010.

[10] Lafont, Y. Interaction nets. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1989), POPL '90, Association for Computing Machinery, p. 95–108.

[11] Mackie, I., and Sato, S. An implementation model for interaction nets. vol. 183.

[12] Mackie, I., and Sato, S. Parallel evaluation of interaction nets: Case studies and experiments. *Electronic Communications of the EASST 73* (Apr. 2016).

[13] Mazza, D. A denotational semantics for the symmetric interaction combinators. *Mathematical Structures in Computer Science 17*, 3 (2007), 527–562.

[14] Nijholt, A. *Overview of Parallel Parsing Strategies*. Springer US, Boston, MA, 1991, pp. 207–229.

[15] Numazaki, H., and Tanaka, H. A new parallel algorithm for generalized lr parsing. In *Proceedings of the 13th Conference on Computational Linguistics - Volume 2* (USA, 1990), COLING '90, Association for Computational Linguistics, p. 305–310.

[16] NVidia. P100 Data Sheet, 2016.

[17] NVidia. CUDA C++ Programming Guide, May 2024.

[18] Possemiers, A. L., and Lee, I. Fast obj file importing and parsing in cuda. *Computational Visual Media 1*, 3 (2015), 229–238.

[19] Rodgers, D. P. Improvements in multiprocessor system design. *SIGARCH Comput. Archit. News 13*, 3 (jun 1985), 225–231.

[20] Sato, S. *Design and implementation of a low-level language for interaction nets*. PhD thesis, University of Sussex, UK, 2015.

[21] Skillicorn, D., and Barnard, D. Parallel parsing on the connection machine. *Information Processing Letters 31*, 3 (1989), 111–117.

[22] Stehle, E., and Jacobsen, H.-A. Parparaw: massively parallel parsing of delimiter-separated raw data. *Proc. VLDB Endow. 13*, 5 (jan 2020), 616–628.

[23] Taelin, V. HVM2: A parallel evaluator for interaction combinators (Unfinished). May 2024.

[24] Voetter, R. Parallel lexing, parsing and semantic analysis on the gpu. Master's thesis, Leiden University, 2021.

[25] Yi, Y., Lai, C.-Y., Petrov, S., and Keutzer, K. Efficient parallel CKY parsing on GPUs. In *Proceedings of the 12th International Conference on Parsing*

Technologies (Dublin, Ireland, Oct. 2011), H. Bunt, J. Nivre, and Ö. Çetinoglu, Eds., Association for Computational Linguistics, pp. 175–185.

Appendix A

A.1 Closure of Stack Mapping Sets

Algorithm 3 describes how to find the closure of sets of stack mappings, adding the Kleene star whenever a looping reduction chain is encountered. The kleeneStar routine takes a mapping I/O and a location i, and wraps the portion of the initial stack $I[i..\mathsf{len}(I)]$ inside a Kleene star.

Algorithm 3 Finding the closure of a set of stack mappings, extended with Kleene star stacks.

Input: Set $S'^{(x)}$ of initial stack mappings associated with reduction rules.
Output: Closure set $S^{(x)}$ of stack mappings extended with Kleene star stacks.

for all $M^{(x)} \in S'^{(x)}$ **do**
  if isShiftAction($M^{(x)}$) **then**
    $S^{(x)} = S^{(x)} \cup M^{(x)}$
  else
    Reductions.push($\{M^{(x)}, \{M^{(x)} \mapsto \text{len}(I^{(x)}) - 1\}\}$)
  end if
end for

while !Reductions.empty() **do**
  $\{M_l^{(x)}$, visited$\}$ $\leftarrow$ Reductions.pop();
  for all $M_r^{(x)} \in S'^{(x)}$ **do**
    $M'^{(x)} \leftarrow M_l^{(x)} \circ M_r^{(x)}$
    if $M'^{(x)} \neq \perp \wedge$ visited$[M_r^{(x)}] =$ **null then**
      if isShiftAction($M_r^{(x)}$) **then**
        $S^{(x)} = S^{(x)} \cup M'^{(x)}$
      else
        visited$[M_r^{(x)}] \mapsto$
        Reductions.push($\{M'^{(x)}$, visited$\}$)
      end if
    else if $M'^{(x)} \neq \perp \wedge$ visited$[M_r^{(x)}] \neq -1$ **then**
      visited$[M_r^{(x)}] \mapsto -1$
      Reductions.push($\{$kleeneStar($M'^{(x)}$, visited$[M_r^{(x)}]$), visited$\}$)
    end if
  end for
end while
return $S^{(x)}$