CPU-less parallel execution of Lambda calculus in digital logic

[Removed For Submission] [Removed For Submission]
[Removed For Submission]

Abstract

While transistor density is still increasing, clock speeds are not, so Hennessy and Patterson's 'new golden age' of architecture calls for radial new parallel architectures. One approach is to completely abandon the concept of central processing units – and thus serial imperative programming – and instead to specify and execute tasks in parallel, compiling from programming languages to data flow digital logic. It is well known that pure functional languages are inherently parallisable, due to the Church-Rosser theorem, and CPU-based parallel compilers exist for many functional languages. However, these still rely on conventional CPUs and their Von Neumann bottlenecks. An alternative is to compile functional languages directly into digital logic to maximize available parallelism. It is difficult to work with complete modern functional languages due to their many features, so we here demonstrate a proof-of-concept system using Lambda calculus as the source language and compiling to digital logic. We show how functional hardware can be tailored to a simplistic functional language forming the grounds for a new model of CPU-less functional computation.

At the algorithmic level, we use a tree-based representation, with data localized with nodes and message passing between them. This is implemented by physical digital logic blocks corresponding to nodes, and busses enabling message passing. Node types and behaviors correspond to Lambda grammar forms, and beta-reductions are performed in parallel allowing branches independent from one another to perform transformations simultaneously. As evidence for this approach, we present a digital logic implementation, and digital logic simulation results showing successful execution of a test suite of Lambda expressions, suggesting that the approach could be scaled to larger functional languages.

CCS Concepts

• **Computer systems organization → Parallel architectures**; • **Software and its engineering → Functional languages**.

Keywords

parallel, compiler, lambda calculus, digital logic, functional

ACM Reference Format:
[Removed For Submission] [Removed For Submission]. 2025. CPU-less parallel execution of Lambda calculus in digital logic. In *Proceedings of PACT 25 (PACT '25).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

PACT '25, Los Angeles, CA

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Transistor density continues to increase, while clock speeds cannot. As demand for computational power continues to increase, the pressure is thus now to find ways to make maximal use of parallelism [4]. While a few algorithms are inherently parallelizable at the algorithmic level – such as the neural networks run on GPUs for AI [8][10] – most are still expressed as traditional serial imperative programs which make them difficult to accelerate.

There is still much discussion as to the most efficient methods of parallelization. Two general approaches exist: First, *explicit parallelization* is where it is up to the programmer to design a new parallel version of their initial serial program [9, 21]. Second, *implicit parallelization* aims to automatically determine optimal parallelization [20] from user programs. This remains challenging for imperative languages due to the difficulty of tracking required interactions between tasks. Implicit parallelism often relies on finding decompositions of functions into sub-functions such that separate independent sub-functions can be parallelized. Pure functional programs naturally have this form due to their lack of state, so are easy to implicitly parallelize [22][16].

Functional languages have existed since Lambda Calculus [5] in 1936. Lambda Calculus and derived languages benefit from the Church-Rosser theorem, which proves that if any terminating sequence of reductions exist, then all terminating sequences of executions – such as call-by-name, call-by-value, and eager execution strategies – will evaluate to the same expression. In particular, this means that all arguments to a function can be evaluated in parallel, along with the function itself using Name placeholders until the argument evaluations are complete and can be substituted for them. Some functional programs are represented and processed as tree-like graphs by implementations, referred to as graph simplification or reduction, which can be parallelized as multiple nodes reducing together [1][25].

Functional languages based on Lambda calculus such as Lisp, Clean, ML, and Haskell have implementations on current CPU-based hardware [18][7]. However, the von Neuman architecture, from which CPUs derive, was designed to process serial imperative programs, and while modern CPUs include some parallel elements which can be exploited [22], individual CPU cores still require imperative machine code, so functional languages typically compile programs to such code, even for performing parallel graph reduction [23][17]. Much interest has been placed in maximizing the efficacy of functional compilers. Logical steps have been refined for reducing Lambda Calculus resulting in Term Graph Rewriting which speeds up one of the most costly steps of graph reduction, rewriting [2]. Techniques such as Half Combustion – a type of lazy execution [21][14] – have been used to optimize compilers. PELCR, a Lambda-based language compiler, uses these techniques to achieve a near 80% speedup compared to prior established methods [13][12][14]. Alternate more abstract expression types have been proposed to allow for predicting futures appearing in λ^{as}

these expressions can evaluate branches below them to skip unnecessary steps [12][14][11]. Other language implementations aim to reduce the number of nodes required to represent a tree by adding expressions to represent abstract data types such as lists.

Following these decisions, hardware engineers have asked what is the most efficient way to represent and execute functional languages, under the particular constraints that hardware imposes [24]. Such hardware designs are split into two camps, based on localized or global data access. Global data access is the more conventional approach, in which data is stored in large globally accessible memory components. Any computational unit currently executing a function may access this shared memory to retrieve information needed for the execution, such as values or pointer information. This method has seen relative success and wide-scale adoption due to its compatibility with well-known imperative programming and von Neumann architectures. However, when multiple computational units are trying to compute simultaneously, all requiring memory access, bottlenecks can appear. The shared memory component must be accessed one address at a time, requiring computational units to wait. In trivially parallizable programs this delay is minimal, however others may have more interaction complexity between their tasks which creates these bottlenecks. Alternatively, localized data stores values in registers and memory components only accessible by a single or limited number processing unit. Localized solutions have been built by representing functions and task across a number of computational units. Examples include distributed, packet-based [23][15], [3], and transputer implementations [7], in which separate machines work simultaneously. This can bypass the read/write bottleneck by limiting the number of machines that can access a memory unit. Previously this has been accomplished through distributed solutions in which nodes send packets to one another to pass information.

All of these previous localized data architectures for parallel execution of functional languages rely on distributed computation based on von Neumann machines. In contrast, we here explore the possibility of compiling a simple functional language – Lambda calculus – directly into digital logic. Digital logic gates are inherently parallel and may remove the need for CPUs altogether. Connecting them with integrated circuit level wiring may also be faster than relying on distributed networking levels of computation.

2 Hardware Considerations

Unlike software implementations of graph structures, digital logic imposes design constraints. Hardware is also significantly more costly to implement; and after design, manufacture, and distribution, bug fixing, and optimization are expensive. Therefore, any hardware implementation must be optimized as design redundancies increase manufacturing costs and processing time. This leads to data structures like graph nodes and integers to be much more limited in quantity, size and function compared to software counterparts. This section outlines the constraints, requirements, and considerations on an algorithms imposed by digital logic.

2.1 Localization of Data

A major advantage of graph-based Lambda-like functional languages is their inherent parallelizability. However, to compute in parallel, certain data structures must be used to accommodate multiple units performing a variety of tasks simultaneously. Multiple nodes accessing shared memory, registers or computation units must queue requests to avoid read/write errors and data collision. Therefore, as many computation and memory components should be localized to individual nodes as possible. Ideally the only shared data accessible to individual nodes should be the clock signal.

2.2 Node Connectivity

Shared memory implementations of graph structures allow any node to connect to any other node via memory pointers. However, accessing shared memory in parallel risks data collision and read/write errors, as nodes contain all information locally. In hardware implementation, a physical connection is required to connect two individual nodes. Physical connections are costly in wire and space. However, limiting node connectivity will either waste nodes that go unused, or require more complicated algorithms to perform graph transformations. So any design must balance flexibility against scalability.

Consider two designs that maximize either connectivity or flexibility. First: a rigid graph where each node has exactly three busses allowing it to connect to its designated parent and two children. This minimizes the number of node to node connections required, potentially saving space. However, unless the program being resolved perfectly fits the predefined graph layout, certain nodes will go unused, arguably wasting more space. Second: a fully-connected point-to-point graph allows any node connection to connect to any other node. This allows the graph to fit perfectly to any program, however will require exponentially more connections as nodes increase, again wasting more space than it saves.

The design that will be used here mixes elements of these two approaches. Nodes are grouped together into *work clusters*, with all work clusters containing the same number of nodes. These nodes are free to configure themselves into whatever pattern is required having fully-connected point-to-point connections with all other nodes in their work cluster. One of these nodes is the root node. The root node is hardwired to have a parent connection to the work cluster's input. Some of the other nodes in the work cluster are output nodes, which may be connected to as children to output data out of the work cluster. Multiple work clusters can then be formed into super work clusters, each containing the same number of work clusters, fully-connected with point-to-point connections between all pairs of clusters in the super cluster. These can then be formed into hyper work clusters, and so on, until the desired number of nodes has been achieved. This structure is flexible while not requiring nearly as many connections as design 2. However, some nodes will be unable to connect to one another, as they may be in different work clusters. Therefore this structure will require additional logic to route information between nodes that do not have point-to-point connections with one another.

2.3 Reusable Nodes

Graph transformations often cause groups of nodes to disconnect to the main graph structure. To minimize the number of nodes required, these must reset themselves to a neutral state, allowing

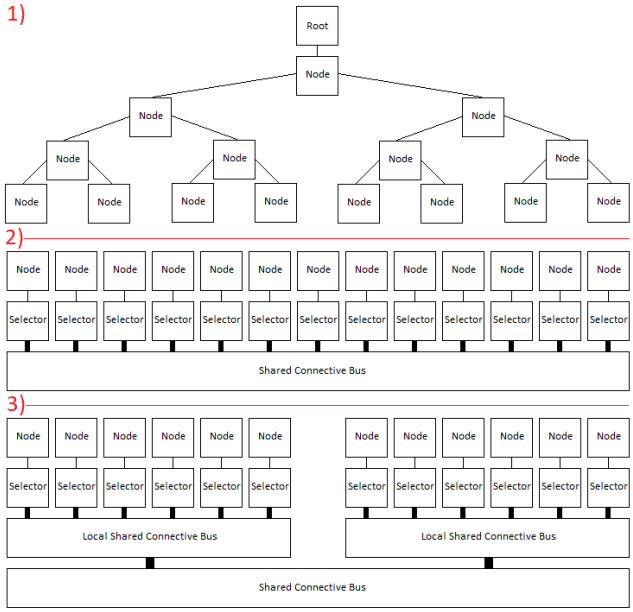

Figure 1: Flexibilty vs. Scalabilty of Graph Structures

them to be reattached to other branches when more space is required. Whenever a node's parent pointer is a null value, it must first propagate a reset signal to its children, then clear all registers.

2.4 Implicit Alpha Conversion

An essential step before reduction in Lambda Calculus is alpha conversion. One or more alpha conversions may be required before some beta reductions can take place. Additional digital logic required to perform these transformations will be costly in both time and space. An alternate, more efficient solution to performing conventional alpha calculus exists due to the rigid and well-defined nature of digital logic.

Don't perform explicit syntactic alpha conversion at all, but have hardware act in equivalent but implicit way on the fly. The purpose of alpha conversion is to prevent multiple possible solutions existing for any Function due to shared name values. For example, without alpha conversion, the expression ${}^{1}(\lambda x.{}^{2}(\lambda x.x)x)^{3}(\lambda x.x)$ could be immediately reduced to two different expressions. In the first option, should the internal Function marked with a 2 be resolved, the expression reduces to ${}^{1}(\lambda x.x)^{3}(\lambda x.x)$ this is the correct solution. In the other option, where the external Function marked with a 1 is reduced initially, we are left with the expression ${}^{2}(\lambda^{3}(\lambda x.x).{}^{3}(\lambda x.x))^{3}(\lambda x.x))$ which contains an invalid expression as the Function marked with a 2 now contains a non-Name figure next to the Lambda symbol. Alpha conversion would convert the expression to ${}^{1}(\lambda x.{}^{2}(\lambda y.y)x)^{3}(\lambda x.x)$, forcing the expression to reduce correctly. So we might say alpha conversion enforces a law that Functions should not reduce until all internal Functions have been reduced. However, sometimes Functions will not have inputs and are unreducible. For example, in the statement ${}^{1}(\lambda x.{}^{2}(\lambda x.x))^{3}(\lambda x.x)$ the internal Function has no input and cannot reduce itself. Alpha conversion causes currently unreducible Functions to protect their contents by converting them to a new value. For example, ${}^{1}(\lambda x.{}^{2}(\lambda y.y))^{3}(\lambda x.x)$. So alpha conversion enforces two laws: (1) that all internal reducible Functions must reduce themselves before the Function in which they are contained can reduce itself, and (2) Functions determined to be currently unreducible must protect their contents from transformation until the Function becomes reducible. So long as these laws are maintained, our model can prevent multiple solutions existing for any Function without performing explicit syntactic alpha conversion.

3 Data structure for expressions

We here present a data structure for lamdba expressions, to fit the hardware limitations above. It is a graph based structure which can be reduced in parallel, with nodes intended to be easily mappable to digital logic structures. We define the different expressions that a node can represent, the different values that are communicated, and the general functionality of each expression type.

3.1 Superclass Expression

Regardless of what expression nodes represent, they all share a number of attributes and methods, which can be captured by the object-oriented notion of a Node superclass from which specific Node types will inherit. Nodes have the potential to connect to three other nodes: a parent node and two child nodes which we will refer to as child left and child right. Three attributes will contain these connected nodes: the parent pointer, the child left pointer and the child right pointer. There are 6 values that must be communicated between any two connected nodes:

First: the graph as a whole requires a mechanism to determine if the graph can no longer be simplified. A boolean attribute called the *Resolve flag* can be used to determine if all nodes on a branch below any given node currently have a raised Resolve flag and thus no node on the branch will be performing any graph transformations. Using this, nodes can determine themselves to be resolved so long as their two children have a raised Resolve flag, and when the root node has a raised Resolve flag we can determine the graph to have been simplified.

Second: as established in the previous section, some nodes that normally perform beta reduction may be determined to be *unreducible*. An expression will raise an *Unreducible flag* to indicate that there is no valid Function input routed through it. Starting from the root node, a raised Unreducible flag signal will be passed to child right. If a node ever receives a raised Unreducible flag from its parent, it first determines itself to be unreducible, then propagates this signal to child right. This will cause all nodes on the rightmost branch of the graph to have raised Unreducible flags. Additionally, some nodes may protect their contents from receiving inputs. For example, in the Function $(\lambda^{1}x.(\lambda^{2}y.y))z$, the internal Function expression marked with a 2 cannot access the z Name expression due to being encapsulated by the Function expression marked with a 1. To simulate this, expressions that encapsulate their contents should always pass a raised Unreducible flag to child right.

Third and Fourth: the nodes must be able to send and receive instructions between each other. Unlike the prior two communicated values any node must be able to send and receive instructions from any other connected node. Some expressions may produce

instructions internally, and send them to connected nodes. Others may simply receive instructions from a connected node and pass them on to another connected node, effectively routing instructions between nodes. Others may act on received instructions, before sending their own instruction to connected nodes. The type of send-receive behavior will depend entirely on which expression a node represents. Instruction data is comprised of an instruction and a node ID which loosely equates to an address. Not all instructions require a node ID, and in these cases the node ID value will equal 0.

Fifth and Sixth: the two final connections are for sending and receiving expression information, including the state of a Resolve flag, expression type, and the contents of the child left and right pointers. All nodes must be able to both send their current expression information represented as an integer, and receive the connected nodes expression information. Again, depending on expression type, nodes may route, transmit or replace expression information. This is used for sending and receiving the values needed during graph transformation.

3.2 Undefined Expression

The first node type is the Undefined expression, representing nodes disconnected from the tree structure. If any information is stored in the parent, child left and child right pointer it will be replaced with NULL values. It may seem redundant, but it is common during simplification that nodes will be disconnected from the graph structure. In a hardware implementation, nodes cannot be deleted and a NULL value must overwrite pointer information to avoid these disconnected nodes flooding busses with irrelevant information causing data collisions. These nodes where possible will be reinserted into the graph structure and may transform themselves into a different expression if appropriate instructions are received.

3.3 GoTo Expression

During graph transformations, for example when a Function expression is reduced, it must be replaced by a GoTo node. While not strictly necessary for a software implementation, this expression type is necessary for non-point-to-point implementations in digital logic. Its two purposes are to remove branches from a tree structure that need to be discarded, and to connect nodes together that may not have the ability to connect to each other. A GoTo node has a single child, all attributes and methods related to child left are not used. The need for GoTo nodes will be made clearer in section 5.

3.4 Name Expression

The Name expression is one of three expressions accessible to the programmer. Take the Function $(\lambda^1 x.(\lambda^2 y.y))z$ the values x,y and z are examples of Name expressions. Name expressions have a few properties that cause them to differ from the super class node. First Name expressions are the only terminator nodes in our expression set, Meaning any branch will always end in Name nodes. Due to this, Name nodes can automatically and permanently raise their Resolve Flag. In conventional Lambda Calculus, Name expressions can store any alphabetic value from $[a-z]$. However, ASCII alphabetical values will be cumbersome to implement in digital logic, so we will use integers to represent Name values. The Name expression is the only expression to store a value internally. All other expressions reference Name expressions when a value is needed. To accommodate this, an additional attribute will be required to store the Name's value. Since Name nodes do not have children, Name information can stored in the attributes usually used as child pointers, as a union datatype (as in the C language).

Name values may replace themselves with another node type or expression value. During beta reductions, Names may have to change their internal value or change their expression type and expand the branch below itself. Details on how and when this is done will be provided in section 5.

3.5 Application Expression

The application expression is the second expression accessible to the programmer. Applications are the main building blocks of the tree structure. An application can contain any expression, as either child, and acts as a branching point in the graph. For example, the Function $(\lambda^1 x.x)(\lambda^2 y.yy)$ contains two applications. The first contains both Functions, Function 1 being child left and and Function 2 being child right. The second application is within the Function marked with a 2 and contains two Name expressions, both with values of y. Applications perform no computation and are primarily concerned with routing instruction and expression information detailed in the super class between connected nodes. An Application routes information differently depending on the state of its Resolve flag. An Application can determine itself to be resolved so long as both its children have their Resolve flags raised. Figure 2 illustrates applications in both these states.

Applications that have a raised Resolve Flag, meaning no nodes on the branch below them are reducible Functions, must route instructions and expression data from potential ancestor Function expressions above them, that at some point may attempt to perform a beta reduction, to descendant Name nodes on the branch below. Instruction data received from both children is OR-ed together and passed to the parent node, as data may be received from either or both children. This allows Names to communicate to their ancestor Functions when beta reductions have been completed. Relevant examples on when this happens will be shown in section 4. In the expression $(\lambda^1 x.x)(\lambda^2 y.yy)$, the application containing two y-valued Names is an example of an application in this state.

Applications with an unraised Resolve flag must route expression data from a determined input node to a Function, and route instruction data from the Function to the input. Despite any individual application not being able to determine where on the tree an input and Function are, so long as all applications route information in a standardized way, it is possible for Functions with a lowered Unreducible flag to always receive their inputs no matter how many applications are between them. To accomplish this, applications in this state pass expression data received from their parent node to their child right node, and instruction data received from the parent to Child Left. Then expression data received from the child right node is passed to the child left node, and instruction data is passed to the applications parent node. Finally, expression information received from the child left node is passed to the application's parent node, and instruction information is passed to the child right node. For example, in $(\lambda^1 x.x)(\lambda^2 y.yy)$, the Function marked with a 1's input is the Function marked with a 2. Luckily they are both

contained within an application whose Resolve flag is unraised. The Application's child left is the Function marked with a 1, and its child right is the Function marked with a 2. So the application passes expression data received from Function 2 to Function 1, and passes instruction data in the opposite direction.

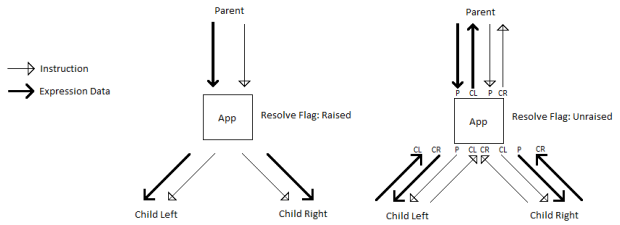

Figure 2: Application Routing Examples

3.6 Function Expression

Functions are the final type of expression accessible to the programmer. Functions always have two children. The first, child left, is always assumed to be, or reduce to, a Name. (If not, the tree will simplify in a unpredictable manner likely causing errors). The second, child right, can be any expression. Functions coordinate beta reduction steps between the Function's input and all applicable Name nodes. First, Functions must determine if they are reducible. If a Function's parent node passes a raised Unreducible flag that Function must raise its own Unreducible flag. That Function is determined to be unreducible. Functions are assumed to be reducible unless the Unreducible flag is raised. In both cases, regardless of whether a Function's Unreducible flag is raised, it must pass a raised flag to its child right. This prevents other descendant Function nodes on the branch below the Function from retrieving input values not encapsulated by the Function. A Function will also expect to receive expression information from its child left and parent nodes. The expression data is passed to child right, and instruction data is passed to both its parent and child right.

As outlined in section 2.4, Functions must comply with certain laws to implicitly perform alpha conversion. The first law applies to Functions that are considered reducible: all internal reducible Functions must reduce themselves before the Function in which they are contained can reduce itself. If a Function's children both have a raised Resolve flag, that Function can determine that there are no descendant Functions with lowered Unreducible flags on the branch below itself. Additionally, a Function should check to see if its input is resolved. Any Function with a lowered Unreducible flag can do this by reading the expression data sent by its input. This step isn't entirely necessary so long as the input value is not mid-reduction when its connected Function begins reducing, potentially a model can be made that allows Functions with a lowered Unreducible flag to be used as inputs. However, this model avoids inserting reducible Functions as it potentially increases the number of graph transformations required to reach a solution, and fewer graph transformations require less computation time. So after a Function with a lowered Unreducable flag receives a raised resolve flag from its two children and its input, the Function can begin performing a beta reduction. To do this, a series of instructions will be transmitted to the Function's parent node and child right node, as explained in further detail in section 5.

The second law outlined in section 2.4 applies to Functions with a raised Unreducable Flag: Functions determined to be currently unreducible must protect their contents from transformation until the Function becomes Reducible. If a Function's parent passes a raised Unreducible flag signal – and is therefore unreducible – upon receiving two Resolve flags from both of their children can raise its own Resolve flag. This allows potential ancestor Functions above it to begin reducing. However, some instructions – those related to performing beta reductions – received from parent nodes will never be passed to the Function's children.

4 Node Intercommunication and Instructions

So far we have detailed how nodes exchange values through a series of shared busses. As established in section 3, nodes communicate to their parents and two children through an expression bus and an instruction bus. We will next show the various 'instructions' that nodes can send to one another to cause transformations. Instructions share similarities with instructions found in von Neuman architectures, but this should not be used to assume a global processing unit in control of all nodes within a cluster. A node may send an instruction to a connected node which may cause it update the some of its values, return information and/or forward that instruction to another node.

4.1 Node Defaults

Nodes contain a series of registers for storing internal values, and methods needed to execute instructions. All nodes within the graph will internally contain a Unique Node ID, which distinguishes that node from any other node in the cluster. Registers for storing an expression type and two child values. Each node includes a stack that can contain a number of Unique Node ID's. Two variables front stack and back stack pointer will record the front of the stack, and the back of the stack respectively. Every node will have access to a temporary variable that can contain an expression type called the expression buffer. Figure 3 shows a list of all internal values, input/output busses and the contents of these busses.

Nodes act differently depending on which expression they are representing. Received instructions may cause the node to change its outputs or update its contents, however when no valid instructions are being received, they will default to a specific output pattern, which will be referred to as the expressions default. deriving from the definitions in section 3 a list Figure 4 contains a complete list of output patterns for expression defaults. Note that the '-' symbol can be interpreted as an empty output bus.

4.2 Instruction Set

Nodes communicate Expression and Instruction information to coordinate graph reduction. Instructions may be *ancestor instructions* received from a parent, or *descendant instructions* from a child. Ancestor instructions are used to coordinate graph transformations, while descendant instructions, save for two instance, are reactionary markers, used to communicate when certain thresholds have been met. Instructions only affect specific expression types,

Bus and Node Contents	Memonic
Internal Values	
Unique Node ID	UNI
Resolve Flag	RSF
Reduce Flag	RDF
Expression	EXR
Expression Buffer	EXB
child left pointer	CLP
child right pointer	CRP
front stack	FSP
back stack	BSP
stack location indicated by front stack	SFS
stack location indicated by back stack	SBS
Expression Bus Contents	
Resolve Flag	RSF
Expression	EXR
child left pointer	CLP
child right pointer	CRP
Information Bus Contents	
Instruction	INS
Unique Node ID	UNI

Inputs	Memonic
Parent Bus	
Parent Expression Bus	PEB
Parent Instruction Bus	PIB
Child Left Bus	
Child Left Expression Bus	CLE
Child Left Instruction Bus	CLI
Child Right Bus	
Child Right Expression Bus	CRE
Child Right Instruction Bus	CRI

As an example [PEB][RSF] represents the Resolve Flag Value a node has received from it's Parent Expression Bus

If [RSF] is written without being prefixed with an intput it reprents an internally produced value in this case the Resolve Flag state of the Node

Figure 3: Internal Value and Input/Output Bus Definitions

Expression	Output Values					
	Parent Expression	Parent Instruction	Child Left Expression	Child Left Instruction	Child Left Expression	Child Left Instruction
Undefined	-	-	-	-	-	-
GoTo	[CRI]	[CRE]	-	-	[PEB]	[PIB]
Name	[RSF][RDF][CLP][CRP]	-	-	-	-	-
Application (lowered resolve flag)	-	-	[PEB]	[PIS]	[PEB]	[PIS]
Application (Raised resolve flag)	[CLE]	[CRI]	[CRE]	[PIB]	[PEB]	[CLI]
Function	[RSF][RDF][CLP][CRP]	-	-	-	-	-

Figure 4: Expression Defaults

causing their nodes to change stored values and output values from their defaults. Pseudocode examples are provided in algorithms 1-12 using keywords from fig. 3.

4.2.1 Nullification. This solution may require new nodes to be added to the graph. To prevent previously discarded data from reappearing before a node can be reused its contents must be nullified. This way when a new node is required the next available undefined nodes Unique Node ID can be added as a child pointer value where the new branch is needed. When a node receives a nullification instruction from its parent it will pass it to all its children. This ensures every node on a branch will receive this instruction. Then the node set its expression type to undefined and clears its child and stack pointers.

Algorithm 1: Nullification, Effects (GoTo, Name, Application and Function)

$[CRI] \leftarrow [PIB]; [CLI] \leftarrow [PIB];$
$[EXR] \leftarrow [Undefined]; [CLP] \leftarrow [0]; [CRP] \leftarrow [0]; [SFS] \leftarrow [0]; [SBS] \leftarrow [0]$

4.2.2 Return and Update. Nodes require instructions to update and retrieve the internal values of other nodes. This is used to setup/output the contents of a work cluster or to update/query branches during a graph transformation. To prevent multiple nodes being updating or returning simultaneously these instructions also send a the unique node ID of the node that needs to perform this instruction. When a node receives these instructions they are passed to its children then if the unique node ID received from the parent instruction bus matches its own unique node ID it performs a return or update.

First the update instructions UpdateExpression shown in (Alg. 2) which causes recipient nodes matching the attached Unique Node ID to update their resolve flag, expression type and child pointer

Algorithm 2: Update Expression, Effects (Undefined, GoTo, Name, Application and Function)

$[CRI] \leftarrow [PIB]; [CLI] \leftarrow [PIB]$ **if** $[PIB][UNI] == [UNI]$ **then**
  $[EXR] \leftarrow [PEB][EXR];$
  $[CRP] \leftarrow [PEB][CRP]; [CLP] \leftarrow [PEB]; [CLP];$
  $[RSF] \leftarrow 0$

values to match values received via the parent expression bus. Variations of this instruction exist for updating a specific pointer values seen in

Algorithm 3: Update Expression, Effects (GoTo, Name, Application and Function)

$[CRI] \leftarrow [PIB]; [CLI] \leftarrow [PIB]$ **if** $[PIB][UNI] == [UNI]$ **then**
  $[CRP] \leftarrow [PEB][CRP]$

Algorithm 4: Update Expression, Effects (Name, Application and Function)

$[CRI] \leftarrow [PIB]; [CLI] \leftarrow [PIB]$ **if** $[PIB][UNI] == [UNI]$ **then**
  $[CLP] \leftarrow [PEB][CLP]$

Next the instruction for returning an expression ReturnExpression (Alg. 5) which cause nodes to return their expression information through the parent expression bus if their unique node ID matches or, if a match is found further down the branch. To accomplish this a descendant marker instruction is sent to indicate which branch the desired node is on and thus which child expression bus to pass onward.

Algorithm 5: Return Expression, Effects (GoTo, Name, Application and Function)

$[CRI] \leftarrow [PIB]; [CLI] \leftarrow [PIB];$
if $[PIB][UNI] == [UNI]$ **then**
  $[PEB] \leftarrow [RSF], [EXR], [CLP], [CRP]; [PIB] \leftarrow [(Mark)]$
else
  if $[[CLI] == [(Mark)]]$ **then**
    $[PEB] \leftarrow [CLE]; [PIB] \leftarrow [CLI][INS]$
  if $[CRI] == [(Mark)]$ **then**
    $[PEB] \leftarrow [CLE]; [PIB] \leftarrow [CLI][INS]$

4.2.3 Branch and GoTo Chop. There are two descendant instruction that are not marker signals used to add and remove GoTo nodes. During a graph transformation branches will need to be removed from the graph. This is done by converting applications into GoTo nodes which chop off a child. The first instruction, Branchchop (Alg. 6), causes applications to send a nullification signal to the relevant child then converts then update its expression.

Later GoTo nodes can remove themselves from the graph by sending a GoToChop (Alg. 7) instruction to its parent then nullifying itself. When a node receives a GoToChop instruction, it updates its relevant child pointer with its child GoTo node's child pointer value, effectively jumping over the GoTo node.

Algorithm 6: Branch Chop, Effects (Application)

```
if [CLI] == [BranchChop] then
    [EXR] ← [GoTo]; [CLI] ← [Nullify]; [CLP] ← [CRP]
else
    [EXR] ← [GoTo]; [CRI] ← [Nullify]
```

Algorithm 7: GoTo Chop, Effects (Function, Application, GoTo)

```
if [CLI] == [GoToChop then
    [CLP] ← [CLE][CRP]
else
    [CRP] ← [CLE][CRP]
```

4.2.4 Graph Transformation Instructions. The following instructions relate to performing graph transformations. For a graph transformation to happen a name or function, known as the ancestor input, needs to search its branch then another name node, known as the descendant input, needs to recreate the values being searched for. This is achieved through sending ReturnExpression and UpdateExpression instructions. However, while the expression produced by the Ancestor is directly copyable, new child pointers are required. To allow both nodes to keep track of a transformation, TransformationPrepare (Alg. 8) is used to update the localized stack with the nodes that have or will be queried. Anytime a node is queried, its front stack pointer (SFS) is increased by the number of children that the node's expression type should have; and the back stack pointer (SBS) is incremented by 1. Values X and Y are added to the stack, which are the unique node ID's of nodes that will be queried in the future. This way, a copy of the branch can be replicated any number of times, and both ancestor and descendant Inputs can simultaneously yet independently know when the transformation has been completed, by checking if the front stack pointer equals the back stack pointer.

Algorithm 8: Transformation Prepare, Effects (N/A)

```
Input(X, Y, Z)
if [SBS]! = [SFS] then
    if [PEB][EXP]! = [Name]or[GoTo] then
        [RAM(Front)] ← [X]; [RAM(Front)] ← [Y]; [SFS] ← [SFS] + 2
    if [PEB][EXP] == [GoTo] then
        [RAM(Front)] ← [X]; [SFS] ← [SFS] + 1
else
    [PIB] ← [Z]]
```

4.2.5 Ancestor Transformation Instructions. There are two instruction that pertain to ancestor inputs the first ImmediateResolution (Alg. 9) which causes the ancestor input to immediately send a BranchChop instruction to its parent, instantly chopping the ancestor's branch off the graph. The second AncestorTransformation (Alg. 10) which either passes its internal values to its parent through the PEB or passes a nodes value from the branch below. Then when the branch has been fully queried the ancestor input sends a GoToChop instruction to its parent removing it branch from the graph.

4.2.6 Descendant Transformation Instructions. Two instructions are again required for Descendant Inputs. First Descendant Inputs need to compare a value to check if a transformation is necessary, for example in the expression $(\lambda x.y))$, name y will not perform a transformation as it contains a different value to name x. CompareValue (Alg. 11) checks name values, sends a marker signal to indicate an equivalence and marks the name node as 'allowed for transformation'. The second required instruction is DescendantTransformation which either updates its internal buffer with values received from its parent or sends UpdateExpression instructions to its children. Then retrieves new unique node IDs to be used as the recently updated nodes child which added to the stack. Then a marker signal is passed when the new branch has been created.

Algorithm 9: Immediate Resolution, Effects (Name and Function)

```
if ClockCycle = 0 then
    [PIB] ← [BranchChop], [UNI]
```

Algorithm 10: Ancestor Transformation, Effects (Function, Name)

```
[CLI] ← [ReturnExpression][RAM]; [CRI] ← [ReturnExpression][RAM]
if [RAM(Back)] == [UNI] then
    TransformationPrepare([CLP], [CRP]], GotoChop);
    [PEB] ← [RSF][EXR][CLP][CRP]]
else
    if [CLI] == [Mark] then
        TransformationPrepare([CLE][CLP], [CLE][CRP]], GotoChop);
        [PEB] ← [CLE]
    else
        TransformationPrepare([CRE][CLP], [CRE][CRP]], GotoChop); [PEB] ←
        [CRE]
    if [SBS == [SFS]] then
        [EXR] ← [EXB]
```

Algorithm 11: Compare Value, Effects (Name)

```
if [EXP] == [(Name)] then
    if ([PEB][CLP] == [CLP]AND[PEB][CRP] == [CRP]) then
        [PIB] ← [(mark)]; Allow for Transformation
if [EXP] == [(Application)] then
    if [CLI][INS] == [(Mark)] then
        [PIB] ← [CLI]
else
    [PIB] ← [CRI]
```

Algorithm 12: Descendant Transformation, Effects (Name)

```
if [RAM(Back)] == [UniqueNodeID] then
    [EXB] ← [PEB][EXR]; [CLP] ← [NewNode1]; [CRP] ← [NewNode1]
else
    [CLI] ← [UpdateExpression][RAM]; [CRI] ← [UpdateExpression][RAM];
    [CLE] ← [PIB][RSF], [PIB][EXR], [NewNode1], [NewNode2];
    [CRE] ← [PIB][RSF], [PIB][EXR], [NewNode1], [NewNode2];
if [CLI] == [Mark] then
    TransformationPrepare([CLE][CLP], [CLE][CRP], [Mark]])
else
    TransformationPrepare([CRE][CLP], [CRE][CRP], [Mark]])
[SBS] ← [SBS] + 1
```

5 Beta Reduction Example

This section will demonstrate a step-by-step beta reduction graph transformation. As an example, we will consider the expression $(x(\lambda^1 y.y))(\lambda^2 z.z)$, which reduces to $(x(\lambda^2 z.z))$.

5.1 Beta Reduction Initialization

Figure 5(a) is an abstract visual representation of a graph structure representing the example expression $(x(\lambda^1 y.y))(\lambda^2 z.z)$. During

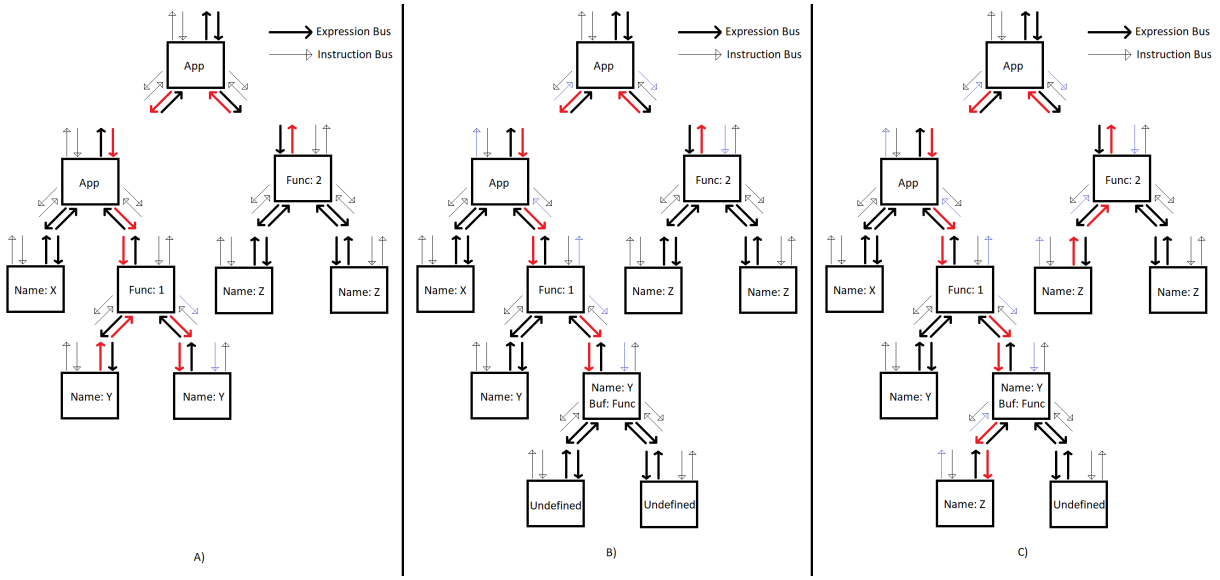

Figure 5: Beta Reduction Examples initialization, step 2 and step 3

transformation steps, buses containing information relevant to the transformation step will be highlighted to show how expression and instruction data is routed throughout the graph. This example begins mid-reduction. Function 2 is receiving an unreducible flag from its parent, and raised resolve flags from both children. This has caused it to raise its own resolve flag, and pass its internal values to its parent through the expression bus, which eventually arrives at Function 1. Function 1 is now receiving a raised resolve flag from both its children, and a raised resolve flag from its Ancestor Input, Function 2. Beta Reduction can now begin. Function 1 instructs its second child to CompareValue expression data retrieved from its first child. If a marker signal is returned, Function 1 sends a AncestorTransformation instruction to the Ancestor Input. Likewise, a DecendantTransfromation instruction is sent to all Descendant Inputs, If Function 1 does not receive a CompareValue marker signal, it can send a ImmediateResolution to its Ancestor Input, and jump to the final step of beta reduction. Both inputs have readied themselves for transformation, by incrementing their front stack by 1, and appending their own unique node ID to their stacks.

5.2 Beta Reduction Transformation Steps

Function 1 will continue to output Descendant/Ancestor Transformation instructions, and will route its received parent expression bus to its child right expression bus, until it receives a marker indicating a resolution. The Ancestor Input's stack currently contains one Unique Node ID: its own. This is pointed to by the back stack pointer. So the first node to be copied during transformation is function 2, which will send the current state of its resolve flag, expression and child pointers to its parent. The Descendant Input also updates the node pointed to by its back stack, which is currently itself. To prevent an expression change – which may cause synchronization errors – the Descendant input stores the expression type in its expression buffer, and will update its expression type later when the tranformation is complete. Its child pointers are updated to point at two new undefined nodes. Next, both inputs add the Unique Node ID's of the children of the node just queried. For the Ancestor Input, this is the Node ID's of the two 'Name: Z' nodes, and for the descendant, it is the two new undefined nodes. Afterwards, the Ancestor Input and the Descentent Input both increment their front stack pointers by the number of children that the relevant expression type should have. In this example step, the expression type just queried was a function which has two children. So both inputs update their FSF value from 1 to 3. Finally, the back stack pointer is increased by one, and both inputs must check to see if the transformation has been completed. Currently, they both have a SBS of 1, and a SFS of 3, which are not equal, meaning the transformation is incomplete. The inputs ready for a second transformation step.

The second transformation has both inputs query a node whose ID matches the back of their stack, as indicated by each Input's back stack pointer. In this example, the ancestor retrieves the contents of its leftmost child name Z using the ReturnExpression instruction. The descendant checks its stack, and also updates its leftmost child, currently undefined, using the UpdateExpression instruction transforming it into a name node with matching contents. Again, both inputs go to update their front stack pointer. However, the expression most recently queried was a name which has no child nodes. So neither the descendant nor ascendant input increase their SFS or append any values to the stack. SBS in then incremented and with SBS and SFS still not equivalent a third transformation step begins.

The third and final transformation step of this example begins with both inputs querying the only remaining ID in their stacks pointed at by the back stack pointer. In this case the rightmost child of both inputs. The final name Z node is retreived then copied into the last undefined node and again the front stack pointer does not

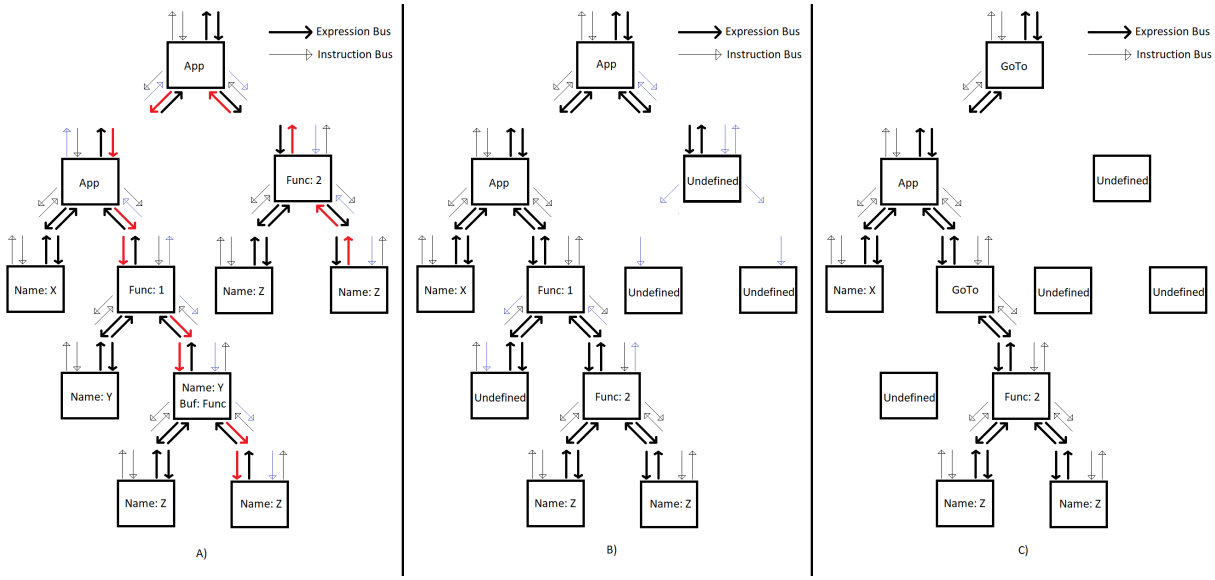

Figure 6: Beta Reduction Examples step 4, nullification and resolution

increase as names have no children. In both nodes, SBS increases to 3, which now matches their stored SFS value. Both inputs independently conclude that the transformation has been completed, and send markers/instructions to cause the resolution and nullification steps of a beta reduction.

The three transformation steps described in the section are visually represented in Figure 5(b) , Figure 5(c) and Figure 6(a).

5.3 Beta Reduction Nullification

Function 1, the function that has just been reduced, has now successfully copied the Ancestor Input's branch into the newly created Descendant Input's branch and has received a marker signal from its descendant input. The function must now remove the compared name value always found as the left most child of any function. To accomplish this, the function will nullify the undesired branch, then convert itself into a GoTo node. The nullification step involves function 1 sending a Nullification instruction to its leftmost child.

The ancestor input has been used as a functions ancestor input and needs to be removed from the graph. So sends a BranchChop signal and its own Unqiue Node ID to its parent node through the parent instruction bus. The parent node, an application, will needs to convert itself into a GoTo node by first nullifying the undesired branch. The BranchChop instruction compares the accompanying Unique Node ID with the applications two child pointers and sends a Nullification instruction to the matching branch. This results in the branch containing the ancestor input nullifying itself.

The Descendant Input is currently still a name node. However, it has two children – which names are not supposed to have. This is an invalid node type, and to resolve this error the descendant input must replace its expression type with the expression stored in its expression buffer register.

The nullification step of beta reduction is demonstrated in Figure 6(b)

5.4 Beta Reduction Resolution

Function 1 and the application that was previously the parent of the ancestor input update their expression type to become GoTo nodes. With this step function 1 has been fully reduced resulting in the graph shown in Figure 6(c).

The undefined nodes have been properly disconnected and, if this graph where to perform another transformation, could be reinserted back into the graph. If a work cluster were to run out of excess undefined nodes during a transformation, the graph could be flooded with GoToChop instructions, and potentially reclaim more undefined nodes. However, this process causes functions to take longer to reduce, so GoTo nodes should only be reclaimed in the absence of undefined nodes.

To convert this graph back into a conventional lambda calculus expression assume that all GoTo nodes do not exist and their parents are directly connected with their children. Then trace the graph downwards from the root node to derive the expression. To demonstrate, in the example shown in Figure 6(c), the graph begins with an application pointing to two children a name node storing an 'X' and a function. continuing down the graph the function has a futher two children both name nodes storing a 'Z'. Therefore this graph can be written as $(x(\lambda^2 z.z))$.

Following this reduction resolve flags will propagate upwards and as there are no reducible functions left the GoTo node at the top of the graph will eventually raise its resolve flag, passing this information to the work cluster's root node. Once this has happened any external components reading the root nodes output can determine that there are no nodes within this work cluster that still require a transformation meaning the cluster as been reduced to its simplest possible form.

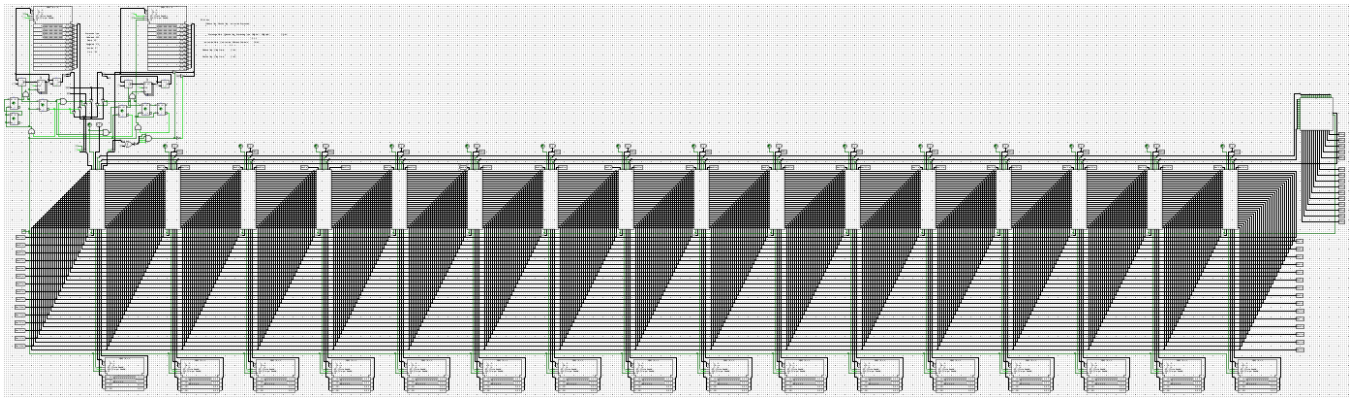

Figure 7: Simulation showing Work Cluster Implementation

6 Validation Simulation

The system is validated by a simulation of a work cluster available as at github.com/LAMB-TARK/CPU-less-parallel-execution-of-Lambda-calculus-in-digital-logic. This contains a copy of Logisim Evolution *Logisim Evolution.jar*, A circuit file *LambdaCalculus.circ* and a sample Lambda expression execution *ExampleExecution.txt*. Steps on how to load, reduce and read the example execution can be found in the repo's *README.txt* file. Alternatively a video presenting the work cluster can be found at https://www.youtube.com/watch?v=ch1fyDx1kn0&t=1s. This video shows how to load, reduce and read lambda expressions using the example discussed in Section 3. Figure 7 is a screenshot of the work cluster containing 16 nodes, their localized RAM components and their shared connective bus.

7 Conclusion

This solution is able to reduce any non-infinite Lambda Calculus expressions however inherits a number of flaws from the language. The primary issue is Church's proposed method for representing numbers through Church numerals. While theoretically Church numerals are a novel and simplistic way of representing numbers and iterations and were apt for a theoretical system. In this practical implementation Church numerals are costly in computation time and node space making this solution impractical for many algorithms. Increasing the number of expression types and expression methods would combat this. For instance adding expression types to allow for list like structures or branch prediction through future nodes [19], allowing expression types to perform arithmetic operations and logical comparisons and allowing functions to apply multiple inputs to its contents either through multiple reduction iterations or allowing multiple simultaneous ancestor inputs similar to Green style sugaring could greatly reduce computation time. However, more complex functionality requires more complex digital logic increasing manufacturing costs, circuit size and energy/heat costs. Determining which expressions and methods should be added is a delicate balancing act and a future research question.

There are a number of limitations this solution will need to address in order to viably compete with modern hardware counterparts. Primarily there is a setup/output bottleneck currently the only way to write to a node within work cluster from an external source is through a UpdateExpression instructions passed to the cluster's root node which is propagated through the graph. Due to the only external accessible point being the root node's. Only one UpdateExpression instruction can be sent per clock pulse. In our current simulation to setup the largest possible graph of 16 nodes it takes 16 clock pulse and this problem will worsen with larger node numbers per work cluster. This issue is mirrored when reading the graph as each node's contents must be retrieved through a ReturnExpression instruction passed through the root node.

Finally while the initial statements that "Digital logic gates are inherently parallel" and that "[Functional Languages] are easy to implicitly parallelize" are true. Our assumption that the combination of these two attributes would reduce the need for optimization compilation steps is not strictly correct. While our solution implicitly reduces any graph given following the most efficient solution. A singular Lambda Calculus expression can be represented through multiple different graphs which reduce differently. Our solution takes more time to compute unevenly weighted graphs where a single branch is significantly longer than any others. When compared to evenly weighted graphs where each branch has a similar node depth. A digital logic solution to balance graphs pre-reduction would significantly increase computation time and circuit complexity therefore this solution will still likely require a programmer or compiler to optimize graph structures pre-reduction.

A hypothetical more advanced CPU-less hardware component taking inspiration from this project would be unlikely to replace Von Neuman-descended hardware. There are likely applications where this the comparatively simpler yet parallelized design would be advantageous for example custom embedded systems, FPGA and low-cost ASICs [6] for digital signal processing on board robots and other real-time fielded systems, where low latency, low cost, and low power consumption are needed.

References

[1] R Banach and GA Papadopoulos. 1993. Parallel term graph rewriting and concurrent logic programs. In *Proc. WPDP-93, Bulgarian Acad. of Sci., Boyanov (ed.)*. 303–322. http://www.cs.man.ac.uk/~banach/some.pubs/Parallel.TGR.Conc.LP.pdf

[2] Andrea Corradini and Frank Drewes. 2011. Term Graph Rewriting and Parallel Term Rewriting. In *TERMGRAPH 2011 EPTCS 48, 2011, pp. 3-18.* https://arxiv.org/pdf/1102.2651

[3] Peter Harrison and M Reeves. 2005. The parallel graph reduction machine, ALICE. In *Graph Reduction Architecture.* https://link.springer.com/chapter/10.1007/3-540-18420-1_55

[4] John L Hennessy and David A Patterson. 2019. A new golden age for computer architecture. *Commun. ACM* 62, 2 (2019), 48–60.

[5] J Roger Hindley and Jonathan P Seldin. 2008. *Lambda-calculus and combinators: an introduction.* Cambridge University Press.

[6] K. Jurkans and C. Fox. 2024. Low-Cost Open Source ASIC Design and Fabrication: Creating your own chips with open source software and multiproject wafers. *IEEE Solid-State Circuits Magazine* (2024).

[7] Marco HG Kesseler. 1996. *The implementation of functional languages on parallel machines with distributed memory.* [Sl: sn]. https://repository.ubn.ru.nl/bitstream/handle/2066/146178/mmubn000001_220469350.pdf

[8] Peter M Kogge. 1985. Function-based computing and parallelism: A review. *Parallel computing* 2, 3 (1985), 243–253. https://www.sciencedirect.com/science/article/abs/pii/0167819185900067

[9] Michael McCool, James Reinders, and Arch Robison. 2012. *Structured parallel programming: patterns for efficient computation.* Elsevier.

[10] Kallas Konstantinos Zdancewic Steve Bastani Osbert Mell, Stephen. 2024. Opportunistically Parallel Lambda Calculus. Or, Lambda: The Ultimate LLM Scripting Language. *arXivLabs* (2024). https://arxiv.org/pdf/2405.11361

[11] Akimasa Morihata. 2019. Lambda calculus with algebraic simplification for reduction parallelization by equational reasoning. In *Proceedings of the ACM on Programming Languages, Volume 3, Issue ICFPArticle No.: 80, Pages 1 - 25.* https://dl.acm.org/doi/abs/10.1145/3341644

[12] J niehren, J Schwinghammer, and G Smolka. 2006. A concurrent lambda calculus with futures. *Sience Direct* (2006). https://www.sciencedirect.com/science/article/pii/S030439750600555X

[13] Marco Pedicini and Francesco Quaglia. 2000. A Parallel Implementation for Optimal Lambda-Calculus Reduction. In *Proceedings of the 2nd ACM SIGPLAN international conference on Principles and practice of declarative programming Pages 3 - 14, september 01, 2000.* https://dl.acm.org/doi/pdf/10.1145/351268.351270

[14] Marco Pedicini and Francesco Quaglia. 2007. PELCR: Parallel environment for optimal lambda-calculus reduction. *ACM Transactions on Computational Logic (TOCL), Volume 8, Issue 3 Pages 14, july 01, 2007* (2007). https://dl.acm.org/doi/pdf/10.1145/1243996.1243997

[15] Simon Peyton Jones. 1987. GRIP - A high-performance architecture. In *Functional Programming Languages and Computer Architecture.* https://www.researchgate.net/publication/221305624_GRIP_-_A_high-performance_architecture_for_parallel_graph_reduction

[16] Simon L Peyton Jones. 1989. Parallel implementations of functional programming languages. *Comput. J.* 32, 2 (1989), 175–186. https://www.microsoft.com/en-us/research/publication/parallel-implementations-of-functional-programming-languages/

[17] Simon L Peyton Jones, Chris Clack, and Jon Salkild. 1995. High-performance parallel graph reduction. In *Programming languages for parallel processing.* springer, 234–247. https://www.microsoft.com/en-us/research/publication/high-performance-parallel-graph-reduction/

[18] Rinus Plasmeijer, Marko Van Eekelen, and MJ Plasmeijer. 1993. *Functional programming and parallel graph rewriting.* Vol. 857. Addison-wesley Reading.

[19] G. Revesz. 1998. *Lambda-calculus, Combinators and Functional Programming: 4.* Cambridge Tracts in Theoretical Computer Science, Series Number 4. https://www.amazon.co.uk/Lambda-calculus-Combinators-Functional-Programming-Theoretical/dp/0521114292#

[20] Masako Takahashi. 1995. Parallel reductions in λ-calculus. *Information and computation* 118, 1 (1995), 120–127.

[21] Phil Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon Peyton Jones. 1998. *Algorithm + strategy = parallelism.* Cambridge University Press. https://www.cambridge.org/core/services/aop-cambridge-core/content/view/24CE696A9E76AEA63C2D6132BE25FC09/S0956796897002967a.pdf/algorithm-strategy-parallelism.pdf

[22] Ian Watson and Paul Watson. 1986. Graph reduction in a parallel virtual memory environment. In *Workshop on Graph Reduction.* Springer, 265–274. https://link.springer.com/chapter/10.1007/3-540-18420-1_60

[23] Ian Watson, Viv Woods, Paul Watson, Richard Banach, Mark Greenberg, and John Sargeant. 1988. Flagship: A parallel architecture for declarative programming. *ACM SIGARCH Computer Architecture News* 16, 2 (1988).

[24] Paul Watson. 1986. *The parallel reduction of lambda calculus expressions.* The University of Manchester (United Kingdom). https://www.proquest.com/docview/2371174486?pq-origsite=gscholar&fromopenview=true

[25] Reinhard Wilhelm, Martin Alt, Florian Martin, and Martin Raber. 1997. Parallel implementation of functional languages. In *Analysis and Verification of Multiple-Agent Languages: 5th LOMAPS Workshop Stockholm, Sweden, June 24–26, 1996 Selected Papers 5.* Springer, 279–295. https://link.springer.com/chapter/10.1007/3-540-62503-8_13

Received XX XXX XXXX; revised XX XXX XXXX; accepted XX XXXX XXXX