Extending CPU-less Parallel Execution of Lambda Calculus in Digital Logic with Arrays and Stacks

Jasmine Ritchie, Harry Fitchett, Charles Fox

College of Health and Science, School of Engineering & Physical Sciences, Computer Science

Moore's law has ended for clock speed, but continues for transistor density, motivating the search for new ways to use more transistors to compute more in parallel. GPUs have provided one approach but must be programmed in specialist parallel programming styles. Programming would be easier if we could execute normal-looking programs with parallelization done automatically. Functional programming has been proposed as a basis for this due to its lack of state enabling execution in different orders. Previous work showed how to do this for a basic lambda calculus language but is not practical for real world programming as many everyday constructs would have to be build up via slow recursive definitions. It is more usual to extend lambda calculus into larger languages with additional primitives to do common tasks such as arrays and stacks. We do this here and implement and test our design using LogiSim Evolution.

Keywords: parallel, compiler, lambda calculus, digital logic, functional

Introduction

The end of Moore's Law in terms of clock speed scaling has forced the computing industry to look for alternative ways to improve performance. While clock speeds have largely plateaued, transistor densities have continued to increase, leading us into what some have called a “new golden age” of computer architecture. Instead of relying on faster individual cores, the emphasis has shifted to exploiting massive parallelism [1]. This shift brings both opportunities and challenges: we now have the hardware capability to perform many operations simultaneously, but we need programming models and execution strategies that can take full advantage of this.

Functional programming languages have long been proposed as a natural and elegant way to express parallel computations [2]. Unlike imperative languages, functional languages avoid mutable state and side effects, which makes it easier to reason about program behaviour and to evaluate expressions in any order. This property is particularly well-suited to parallel execution, as independent computations can be carried out concurrently without concern for synchronization or race conditions. As such, functional programming offers a promising foundation for building systems that automatically extract parallelism from conventional-looking code

Project Background

This project supports a corresponding paper titled “CPU-less Parallel Execution of Lambda Calculus in Digital Logic” [8]. The paper introduces a new computational model inspired by lambda calculus, adopting a functional rather than processor-centric approach. Instead of relying on a single powerful processor to execute complex instructions, this model distributes the reduction of lambda calculus programs across many small, simple processors called nodes.

In this framework, programs are divided into expressions, with each node representing one expression. Expressions can take different forms—such as names, functions, or applications—and each type reacts differently to received data stimuli. A single node cannot compute anything on its own. However, when connected through a set of routing patterns, the collective system of nodes is able to reduce entire programs. The exact mechanics of this reduction are outside the scope of this paper, but it is important to understand the registers and communication channels available to each node.

Every node connects to three others: a parent, a left child, and a right child. Because of this branching structure, programs are typically represented as tree-like graphs. Each node has three registers: an expression register, which stores a binary key identifying the expression type, and two child pointers, which hold the unique node IDs of its left and right children. Nodes also communicate with one another through shared buses, exchanging two main types of information: expression and instruction data. Expression data includes the contents of a node’s registers along with flags that help determine when computation is complete. Instruction data, by contrast, consists of a binary key that identifies a particular instruction, along with the ID of the node the instruction targets (when applicable).

The accompanying paper demonstrates that this functional model has potential for addressing some of the major challenges in parallel computation. However, it concludes that performance could be improved by expanding the range of expression types. This project builds on that work by extending the expression set to include a common and practical data type in hardware, lists.

Literature Review

This project supports and expands upon a number of related works. Other researchers have explored and designed logical systems inspired by lambda calculus, this is primarily done by adding new complex expression types in this paper we explored a more practical expression commonly used in hardware programming. However, theoretical designs explore more abstract expressions, such as futures [9], for minimizing reduction paths and reducing transformation times. The flagship project [10] was an early attempt to exploit the implicit parallelisms of functional

programs such as lambda calculus. It also explores data routing techniques to allow a series of low end processors to work together and solve programs. While an important piece of work it came before the widely adopted use of parallel designs in computer architecture. Another high performance parallel machine is GRIP [11]. GRIP also intended to exploit functional languages to implicitly parallelize programs across distributed hardware, using packet switching protocols to allow multiple processors to access a shared memory and communicate with one another. Despite its success the project encountered major design issues with its scalability as its shared high band width bus could only support so many processors at once. ALICE [12] is another proposed architecture that argues that `functional languages provide the most effective means for producing software and that the right approach is to develop a customized architecture' [12]. This paper attempts to translate graph reduction techniques into a series of processing elements communicating via a shared bus. The 1996 paper "The implementation of Functional Languages on Parallel Machines with Distributed Hardware" [13]. Describes the limits of functional languages in distributed hardware, proposing the solution of distributed memory. In which Kessler evaluates a parallel transputer architecture setup to reduce a functional language known as Clean. These models and the model proposed in this paper are currently unlikely to replace von Neumann descended hardware. But there may be some immediate applications where these comparatively simpler yet parallelized design could be advantageous. For example: in custom embedded systems, FPGA and low cost ASICs [5], for digital signal processing on-board robots and other real time, IoT fielded edge computing systems, where low latency, low cost, and low power consumption are needed.

Methodology

List/Array Overview

Lists are used to store items and often create structures like stacks. In CPU design, stacks help manage temporary suspensions of operations and execute sub-functions. This project represents arrays as a chain of nodes, where each node points to the next one and holds a value. Since this expression type will only have one child, we won’t use the Child Left Pointer (CLP) as usual. Instead, we will repurpose the CLP to store a value.

Depth Addressing Overview

In traditional programming lists are accessed using an index. In this project, each list node is assigned a depth value. To determine this, a node checks whether its pointer is connected to another node. If not, the node identifies itself as the last item in the list and assigns itself a depth of zero. If it is connected to another node it retrieves its depth value from its child. Each list item adds one to its depth, passing this updated value to its parent, allowing depth values to propagate upwards. Instructions can now target nodes based on their depth, similar to how previous instructions, such as `UpdateExpression`, target nodes based on their ID.

List/Array Instructions

The four instructions we have added for arrays are: `Update Depth`, `Return Depth`, `Remove Bottom Node`, and `Add Bottom Node`, below you can find descriptions for them.

Update Depth Instruction

This instruction tells a node at a specific depth to update its value. Similar to changing an element in a list using its index. Algorithm 1 shows a pseudocode version of this instruction. In the first line, each node passes the instruction it received from its parent (via the PIB) down to its child (via the CRI), causing every node within the list to receive this instruction. Next, each node checks if the instructions target depth matches its own depth. If it does, that node updates its value (stored in the CLP) using the new value provided by the PEB.

Algorithm 1 Update Depth

$[CRI] \leftarrow [PIB]$

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

end if

Return Depth Instruction

This instruction retrieves the value retrieves the value of a node at a target depth. Similar to accessing an item in a list using its index. Algorithm 2 shows a pseudocode version of this instruction. Like the `UpdateDepth` instruction, the first line ensures that the instruction is propagated to all nodes within the list. Each node then checks if its depth matches the target depth. If there's a match, the node sends its value back to its parent using the PEB. Since nodes automatically pass expression data upwards, the requested value will be returned to the top of the list.

Algorithm 2 Return Depth

$[CRI] \leftarrow [PIB]$

if $[PIB][UNI] == [CRI][UNI]$ **then**$[PEB] \leftarrow [InternalValue]$

end if

Add Bottom Node Instruction

This instruction adds a node to the end of a list. When used together with `RemoveBottomNode` it enables the list to emulate stack-like behaviour, where the bottom node is the top of the stack. Algorithm 3 shows this operation. This instruction targets the node with a depth of zero. When a non targeted node receives this instruction it passes data downwards, eventually reaching the target node. This instruction takes multiple clock cycles to complete. The first clock pulse causes the target node to request a new ID, so it can prepare to link to the new node. During the second clock cycle the target node updates its CRP to point to the new node. Finally,

the target node sends an `UpdateExpression` instruction to its child, ensuring it adopts the expression characteristics defined on its PEB.

Algorithm 3 Add Bottom Node

```
if ClockPulses == 0 then
    [CRI] ← [PIB];
end if
if ClockPulses == 1 then
    [CRI] ← [PIB];
    [CRE] ← [PEB];
    if [CRI][UNI] == 0 then
        [CRP] ← [NewNodeID];
    end if
end if
if ClockPulses == 2 then
    [CRE] ← [PEB];
    if [CRI][UNI] == 0 then
        [CRI] ← [UpdateExpressionInstruction][CRP];
    else
        [CRI] ← [PIB];
    end if
end if
```

Remove Bottom Node Instruction

This instruction removes the bottom node from a list. This process is illustrated in the pseudocode shown in Algorithm 4, taking multiple clock pulses to complete and targeting a node of depth one. During the first clock pulse, each node, other than the target, passes this instruction downwards, ensuring that every node receives it. The target begins removal by sending a `Nullification` instruction to its child. During the second clock pulse, non-targeted nodes continue to pass the instruction, and the target sets its CRP to zero, removing its child from the list.

Algorithm 4 Remove Bottom Node

```
if ClockPulses == 0 then
    [CRI] ← [PIB]
end if
if ClockPulses == 1 then
    if [CRI][UNI] == 1 then
        [CRI] ← [NullificationInstruction]
        [CRP] ← 0
    else
        [CRI] ← [PIB]
    end if
end if
```

Results

The expression type and the four corresponding instructions have been implemented and tested to be compatible with and easy to insert into the previous system using the HDL LogiSim Evolution. Found here at: *https://github.com/LAMB-TARK/CPU-less-parallel-execution-of-Lambda-calculus-in-digital-logic*.

Update Depth Validation Test Bench

Table 1 shows a test bench verifying the `UpdateDepth` instruction, line one is a control, showing an array node receiving no instructions. Instructions are triggered by activation signals, found on the PIB, and the corresponding signal for this instruction is `1010`. In line one the PIB is empty, so the node executes no instruction.

Line two shows an array node receiving an `UpdateDepth` instruction. The PIB now contains the activation signal. The first step of algorithm 1 is for the node to pass the instruction downwards. Demonstrated in line two where data, received from the PIB, is forwarded to CRIOut.

Line three shows the CLP being updated. The ‘NewNode’ output is used to update the nodes internal values. The first four bits represent expression type, the middle four the CLP, and the last four bits the CRP. The `UpdateDepth` instruction updates the CLP with data received from the PEB, which is divided into the same three groups. Therefore, the NewNode output contains an expression of `0110` (a List expression), a value of `1111` retrieved from the PEB, and a CRP of `0000`, causing the node to update its CRP.

Line four shows that the CLP does not update if a nodes depth does not match the target depth. In this example the target depth, found in the PIB, is zero. However the

nodes depth found in last four bits of the CRI is three. Therefore, the nodes CLP does not update, mirrored in our test bench where NewNode is empty. In contrast line five will update its child left pointer as in that example the node depth and target depth are both three.

Discussion

Status	PIB	CRI	Internal Value	PEB	CRI Out	PIB Out	New Node
Pass	0000 0000	0000 0000	0110 0010 0000	0000 0000 0000	0000 0000	0000 0001	xxxx xxxx xxxx
Pass	1010 0000	0000 0000	0110 0010 0000	0000 0000 0000	1010 0000	0000 0001	0110 0000 0000
Pass	1010 0000	0000 0000	0110 0010 0000	0000 1111 0000	1010 0000	0000 0001	0110 1111 0000
Pass	1010 0000	0000 0011	0110 0010 0000	0000 1111 0000	1010 0000	0000 0100	xxxx xxxx xxxx
Pass	1010 0011	0000 0011	0110 0010 0000	0000 1111 0000	1010 0011	0000 0100	0110 1111 0000

Table 1: Update Depth Instruction Test Bench

Return Depth Validation Test Bench

Table 2 shows a test bench verifying the `ReturnDepth` instruction. `ReturnDepth`, initially acts similarly to `UpdateDepth` passing received data down the list ensuring every list item receives the instruction, demonstrated in line one. In line two the in instructions targeted depth matches the node internal depth causing the node, causing the node to pass its internal value to its parent via the PEB.

The final line shows an untargeted node reacting to this instruction. The target depth, found in the PIB is four, and the internal depth found in the CRI is one. Therefore, the node forwards expression data received from its child to its parent. Propagating the targeted nodes internal value upwards

Status	PIB	CRI	Internal Value	CRE	CRI Out	PEB Out
Pass	1011 0100	0000 0100	0110 0000 0001	0000 0000 0000	1011 0100	0110 0000 0001
Pass	1011 0001	0000 0001	0110 1010 0001	0000 0000 0000	1011 0001	0110 1010 0001
Pass	1010 0100	0000 0001	0110 0000 0010	0110 1111 1111	1011 0100	0110 1111 1111

Table 2: Return Depth Instruction Test Bench

Remove Bottom Node Validation Test Bench

Table 3 illustrates a test bench verifying the `RemoveBottomNode` instruction, taking multiple clock pulses to complete. As a result, each execution spans multiple rows, separated by test indices. This instruction targets the node with a depth of one. In the first test index, we see a node that does not meet this condition. During the first and second clock cycles, it simply forwards the instruction downwards.

The second test index shows how the targeted node behaves. In the first clock cycle, the node sends a nullification signal to its child, reflected by the CRIOut containing `0001 0000`, the nullification instruction. In the second clock cycle, the node updates its CLP to zero. This update can be observed by comparing the InternalValue column with the NewNode column. Where, the final four bits, which represent the CLP go from containing `0001` to `0000`.

Status		Clock Pulse	CRI	PIB	Internal Value	CRI Out	New Node
Pass	1	1st	0000 0011	1100 0000	0110 1111 0011	1100 0000	xxxx xxxx xxxx
Pass		2nd	0000 0011	1100 0000	0110 1111 0011	1100 0000	xxxx xxxx xxxx
Pass	2	1st	0000 0001	1100 0000	0110 1111 0001	0001 0000	xxxx xxxx xxxx
Pass		2nd	0000 0001	1100 0000	0110 1111 0001	0001 0000	0110 1111 0000

Table 3: Remove Bottom Node Instruction Test Bench

Add Bottom Node Validation Test Bench

Table 4 demonstrates the final instruction. Like the previous instruction, it requires multiple clock cycles to complete, and each test case is identified by a test index. This instruction targets a depth of zero. The first test index illustrates what happens when an targeted node receives this instruction. Again, no changes are made and the instruction gets passed down. The second test index shows the a target node receiving this instruction. In the first clock pulse, the node requests a new ID by sending a NewNodeReq signal. During the second pulse, the node receives the value `0011` through NewNodeID and updates its CRP so that it points to this new node. In final pulse, the node outputs `0010 0011` to its CRI, which corresponds to an `UpdateExpression` instruction targeting the newly attached node.

Status		Clock Pulse	CRI	PIB	Intern Value	PEB	New Node	New Node ID	CRI Out	CRE Out	New Node Req
Pass	1	1st	0000 0011	1101 0000	0110 1111 0001	0110 1010 0000	xxxx xxxx xxxx	xxxx	1101 0000	0110 1010 0000	0
Pass		2nd	0000 0011	1101 0000	0110 1111 0001	0110 1010 0000	xxxx xxxx xxxx	xxxx	1101 0000	0110 1010 0000	0
Pass		3rd	0000 0011	1101 0000	0110 1111 0001	0110 1010 0000	xxxx xxxx xxxx	xxxx	1101 0000	0110 1010 0000	0
Pass	2	1st	0000 0000	1101 1000	0110 1111 0000	0110 1010 0000	xxxx xxxx xxxx	xxxx	1101 1000	0110 1010 0000	1
Pass		2nd	0000 0000	1101 1000	0110 1111 0000	0110 1010 0000	0110 1010 0011	0011	1101 1000	0110 1010 0000	0
Pass		3rd	0000 0000	1101 1000	0110 1111 0000	0110 1010 0011	xxxx xxxx xxxx	0011	1101 1000	0110 1010 0000	0

Table 4: Add Bottom Node Instruction Test Bench

Conclusion

This project shows that the existing literature and architecture can be extended to handle more complex data types, such as lists. Previously, it was stated that expanding the expression set to include integers, lists, and futures could help address limitations caused by node counts. However, if every node were capable of performing arithmetic and logical operations, each would need something like its own accessible ALU, greatly increasing its size. Therefore as nodes gain the ability to support a wider variety of expressions, their physical size increases. This means that even though a program might be representable with fewer nodes, the total space required to represent the program could actually grow. Because of this, it is important to weigh each new expression type by how much it reduces program size compared to the space requirements of its implementation.

From this, two possible directions emerge. One option is to abandon lambda calculus in favor of a more complex logic system. Larger nodes would allow direct arithmetic and other more traditional non-functional operations, but this approach undermines the very properties that made the original system attractive, essentially turning the model into a GPU. For that reason, it seems less promising. The alternative is to extend lambda calculus by adopting a system derived from it, such as λ^{as} or LISP. These frameworks already provide proven expression types and preserve many of the desirable traits of the original implementation.

To further this project a metric for evaluating an expression types cost to benefit and thus evaluating logical systems as a whole must be researched. The ideal system will minimize both node bulk and overall program graph size. Once identified and implemented, it can then be meaningfully compared against contemporary hardware.

Discussion

Through this project, I have developed a deeper understanding of how low-level data structures like arrays can be represented within hardware. The unique architecture combined with depth-based addressing, which I invented, provided a unique perspective into data representation. By designing instructions, I gained experience with how instructions are implemented, helping me appreciate the complexity of hardware-level operations, where timing is crucial for accurate execution. Designing test benches taught me debugging techniques used by computer architects, highlighting the necessity for planning and documentation.

This project has improved my technical knowledge of data structures in hardware, sharpening my skills in designing and testing complex systems. The experience has provided valuable insight into the challenges of low-level computing, from ensuring efficient data flow to managing memory and processing across multiple pulses. During the project, I challenged my critical thinking, explored academic literature, and contributed to research, sparking a deeper interest in computer hardware.

References

[1] J. L. Hennessy and D. A. Patterson, “A new golden age for computer architecture,” Communications of the ACM, vol. 62, no. 2, pp. 48–60, 2019.

[2] J. R. Hindley and J. P. Seldin, Lambda-calculus and combinators: an introduction. Cambridge University Press, 2008.

[3] R. Banach and G. Papadopoulos, “Parallel term graph rewriting and concurrent logic programs,” in Proc. WPDP-93, Bulgarian Acad. of Sci., Boyanov (ed.), pp. 303–322, 1993.

[4] R. Wilhelm, M. Alt, F. Martin, and M. Raber, “Parallel implementation of functional languages,” in Analysis and Verification of Multiple-Agent Languages: 5th LOMAPS Workshop, pp. 279–295, Springer, 1997.

[5] K. Jurkans and C. Fox, “Low-cost open source ASIC design and fabrication: Creating your own chips with open source software and multiproject wafers,” IEEE Solid-State Circuits Magazine, vol. 16, no. 2, pp. 67–74,

[6] P. M. Kogge, “Function-based computing and parallelism: A review,” Parallel computing, vol. 2, no. 3, pp. 243–253, 1985.

[7] S. Mell, K. Kallas, S. Zdancewic, and O. Bastani, “Opportunistically parallel lambda calculus. or, lambda: The ultimate LLM scripting language,” arXiv:2405.11361, 2024.

[8] H. Fitchett and C. Fox, “Cpu-less parallel execution of lambda calculus in digital logic,” Submitted, 2025.

[9] G. Revesz, Lambda-calculus, Combinators and Functional Programming: 4. Cambridge Tracts in Theoretical Computer Science, Series Number 4, 1998.

[10] I. Watson, V. Woods, P. Watson, R. Banach, M. Greenberg, and J. Sargeant, “Flagship: A parallel architecture for declarative programming,” ACM SIGARCH Computer Architecture News, vol. 16, no. 2, 1988.

[11] S. Peyton Jones, “Grip - a high-performance architecture,” in Functional Programming Languages and Computer Architecture, Springer, 1987.

[12] P. G. Harrison and M. J. Reeve, “The parallel graph reduction machine, ALICE,” in Graph Reduction (J. H. Fasel and R. M. Keller, eds.), pp. 181–202, Springer, 1987.

[13] M. H. Kesseler, The implementation of functional languages on parallel machines with distributed memory. PhD dissertation, Radboud University, NL, 1996.