Using Continous Logic Networks for Hardware Allocation

Anthony Saieva
Columbia University
ant@cs.columbia.edu

Dennis Roellke
Columbia University
Dennis.Roellke@rub.de

Gail Kaiser
Columbia University
kaiser@cs.columbia.edu

Suman Jana
Columbia University
suman@cs.columbia.edu

ABSTRACT

In recent years cloud computing has become a trend from for both enterprise and independent developers. As such cloud computing providers have become the main source of hardware for modern computing. This modern computing paradigm makes specialized hardware available at scale. Special purpose hardware or *Hardware Accelerators* have experienced a revival since they can now be shared among multiple cloud users.

While accelerators have shown great efficiency in terms of power consumption and performance, determining which functions can be accelerated remains problematic without manual selection. Static similarity analysis has traditionally been based on solving satisfiable modulo theorems (SMT), but in a parallel advancement, continuous logic networks (CLN's) have provided a faster and more efficient alternative to traditional SMT solving by replacing traditional boolean functions with smooth estimations. These smooth estimates create the opportunity to leverage gradient descent to *learn* the solution. We present AccFinder, the first CLN based code similarity solution and begin to evaluate its effectiveness across a series of accelerator benchmarks.

1 INTRODUCTION

In recent years hardware accelerator technology has evolved to make them more accessible to non specialized developers. In the past if you wanted to run your programs on an accelerator you had to build your own or pay an industry expert to build it for you. This was particularly strenuous since it required both a domain specific skill set and specialized equipment to produce the accelerator. With the advent of FPGA's and programmable micro controllers, producing accelerators became possible for hardware developers anywhere, but still required a specific skillset.

In gaming and graphics for instance, most matrix computations have been computed on graphics processing units (GPU) instead of traditional CPU computing. Since GPUs have become highly available, NVIDIA developed the CUDA framework. This framework exposes hardware level APIs at the software level, and provided the developer has a parallelized version of their algorithm they can implement any general purpose algorithm on the GPU. Consequently the GPU functions as a general purpose *accelerator* for all functions. CUDA represents a shift from hardware to software accelerator development since now the developer can implement the accelerator without dealing with the hardware explicitly. Still the developer needs to know the alternative implementation, and they need to know how to work within the CUDA framework.

Furthermore with the rise of cloud computing developers can run programs on hardware that's not available locally so any hardware that the cloud provider offers becomes easily accessible.

This rises a new problem: With all the new availability of hardware, how do you utilize it properly? Developers may have functions that can be accelerated and they may have access to the accelerated versions, but they may simply not be aware of this opportunity. Luckily regardless of how the accelerator was exposed or implemented there is some software representation of the accelerator's function. So by performing code analysis on the software representation of the accelerator and the function we wish to accelerate we can match candidates - and make use of our access to hardware acceleration.

While choosing an accelerator carries tradeoffs with respect to size, performance, and energy consumption, from the perspective of the software developer this is largely transparent. The accelerator functions as a black box that takes an input and produces the desired output. Therefore when assigning functions to accelerators only the inputs and outputs of the functions need to match. Code similarity is a well studied problem applied to code search, refactoring and other tasks. Many solutions exist both static and dynamic, but most suffer from performance problems due to heavyweight code analysis procedures []. While dynamic procedures tend to have better performance, they put constraints on the developers since the code has to run under specific circumstances for the system to work. The static code analysis solutions generally involve various forms of symbolic execution reliant on SMT solving, but unfortunately due to the computationally expensive nature of symbolic execution these solutions introduce significant overhead.

In a parallel development, continuous logic networks (CLN) have significantly increased the performance of traditional SMT solvers. By replacing the traditional boolean functions with smooth representations continuous logic networks can use gradient descent to learn the solution to an SMT formula with increased efficiency relative to state of the art SMT solvers.

This increase in efficiency makes what were previously heavyweight calculations lightweight. We present a technique to statically transform software meant for accelerators and the software modeling the accelerator into SMT formulas and then solve the SMT formulas under a variety of constraints using CLN's to develop a similarity metric between the two.

Our approach remains static without imposing the constraints on the developer that are associated with dynamic analysis, and we retain a relatively lightweight solution.

This is the first work to apply CLNs in the context of code similarity and the accelerator assignment problem. There are some

details that need to be resolved, but initial efforts indicate that with engineering effort this work is feasible.

2 BACKGROUND

2.1 SMT Formulas

Satisfiable Modulo Theories have far reaching application in computer science. Especially program analysis, tasks such as formal verification and symbolic execution benefit from a programâĂŹs abstraction as an SMT formula. In formal verification SMT formulas are used to prove the correctness and completeness of a program. This is commonly pursued by defining three inequalities. The loop pre-condition, the loop post-condition, and a the loop invariant, a consistent piece throughout each loop-iteration. These three clauses can than be chained by conjunction and their satisfiability can be decided by an SMT solver. This is especially interesting for programs containing consecutive loops or nested loops, which is when loop conditions get chained to SMT formulas of increasing size. Symbolic execution assumes knowledge about the program's branch conditions, which can be interpreted as a conjunction of inequalities as well. This technique is commonly used in bug discovery, where it helps to gain code coverage, by generating inputs and guiding fuzzers. In its raw form however, symbolic execution suffers from path explosion, namely the exponential growths of the SMT formula per discovered branch.

These agreed upon use cases for SMT formulas emphasize the expressiveness of SMT to abstract the intrinsic meaning of a computer program. As such, we have identified SMT formulas to be an intuitive fit to approach the code similarity problem by following the observation that if two programs are similar, they can be expressed by similar SMT formulas. Conversely, if two SMT formulas are similar they represent two underlying programs that are similar. We will discuss the soundness and completeness of this assumption in Section (tbd).

Note that SMT solving is NP-complete and proven not to be solved more efficiently than by using heuristics. This causes extensive runtime for SMT solvers, or even undecidability for programs of sufficient size. Even established projects like S2E and Z3 fail to overcome this performance issue, though they are under constant research and that ranges from novel heuristics [] to hardware optimizations []. A new line of research however, shows how the problem can be approached by mapping the input/ output space from binary (satisfiable or not) to floating point arithmetic. This approach enables the application of strong methods from the field of smooth optimization - a field where extensive progress was made over the last few years as it was nudged by the advent of machine learning.

2.2 Continuous Logic Networks

A major restriction to SMT, is its discrete, boolean nature, i.e. that its solutions are ∈ {0,1}. As is, there is no notion of solutions being âĂIJalmostâĂİ satisfiable or problems are âĂIJclose toâĂİ having a solution. Hajek et al. introduced basic fuzzy logic to approach this problem by defining a class of logic that uses continuous values in the range [0, 1]. This representation is differentiable almost everywhere and has thus been identified as a well-suited baseline to apply differential analysis to satisfiability problems by adapting

$$F(x) = (x = 1) \vee ((x \geq 3) \wedge (x \leq 4)) \vee (x \geq 5)$$

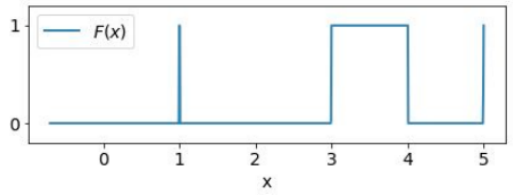

Figure 1: An SMT formula an its respective graph of truth values. Note that the graph is composed of discrete jumps and can only take y-values of 0 and 1.

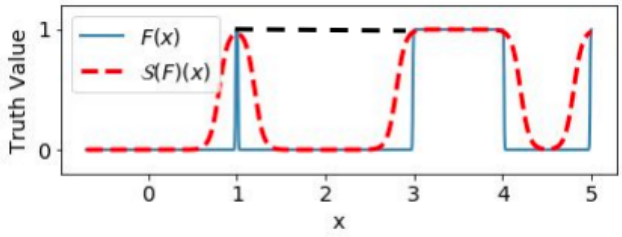

Figure 2: The SMT formula's respective smooth differentiable mapping as generated by a CLN. CLNs use sigmoid function to approximate the discrete jumps in a smoother manner and factor a smoothing value B to regularize the values to 0 and 1.

techniques commonly used to solve optimization problems. Ryan et al. define Continuous Logic Networks (CLN) to abstract SMT Formulas as *fully* differentiable optimization problems [].

To achieve a smooth continuous representation of an SMT formula, CLNs map the boolean operations of conjunctions and disjunctions to t-norms and t-conorms (denoted by ⊕ and ⊗). Similar to boolean operations, t-norms are commutative, monoton and consistent. Namely, their input order does not influence the result, larger inputs cause larger outputs, and the result of any t-norm and 1 is 1; while the t-norm of any truth value and 0 is 0. Hence, t-norms resemble boolean *and* operations. A relationship between t-conorms and disjunctions can be shown respectively.

To solve an SMT Formula, a CLN can be constructed such that every value is marked as either an input term, a constant, or a learnable parameter. Once a CLN is constructed it can be efficiently trained using gradient descent, which makes the solution finding process so much faster than traditional SMT solving. When training a CLN to approach a loss equal to 0, the resulting *continuous* SMT formula is consistent with the solution of a *boolean* SMT formula. In particular, Ryan et al. prove that continuous SMT formulas learned with CLNs are sound and complete with regard to SMT formulas on discrete logic. They further prove that a subset of SMT formulas are guaranteed to converge to a globally optimal solution [].

The expressiveness and efficiency presented by CLNs leads us to employ this novel technique to compare two programs for code

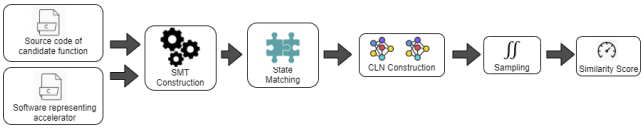

Figure 3: General Overview

similarity. Using CLNs such similarity analysis is made possible more efficiently than ever before.

Furthermore, we have motivated that SMT formulas, even if normalized to Conjunctive Normal Form (CNF) do not have to be identical to represent similar programs. More importantly, we want to quantify the number of overlapping solutions that satisfy both formulas, namely the program characteristics that both candidates have in common. As a consequence, we aim to measure the integral of the overlapping intervals of the function. The integral is well defined for discrete formulas, but we can expect better and faster results when taking under consideration the approximated solutions of a CLN. More about integration method can be found in Section 3.4.

2.3 Use Case Scenario and Problem Specification

We envision a developer writing software normally, and has a series of hardware accelerators available either physically, through a cloud provider, or in the form of a series of CUDA implementations. They have the ability to develop software in high level language like C but don't want to spend the effort to develop actual hardware. Furthermore we limit this discussion to function level comparison, we do not match function sub snippets to possible accelerators.

A more formal problem specification would be: **Given a function** f**, the set of accelerators** A**, and the set of programs** G **such that each accelerator in** A **is represented by a single program in** G **- can we determine if** f **is equivalent (as defined by input output equivalence) to some function in** G **and can be replaced by some corresponding accelerator** a ∈ A**?**

3 DESIGN

Our system is designed in 4 consecutive steps that lead to a numerical code similarity score in the range [0,1]. Figure 8 displays this general architecture. First we represent each function as an SMT formula. In the SMT formula, each variable represents a possible state of the program. We perform a variable mapping to link program states between the two code candidates. These SMT formulas then get transformed into CLNs, which we ultimately use to apply a similarity analysis and deduce the similarity score.

3.1 SMT Construction

To construct an SMT formula, each input to the candidate function - such as a parameter or global variable - gets assigned to a variable in the SMT. SImilarly, each item in an array has its own variable and each intermediary program state has another variable. Input variables can be any value, but intermediary state variables must be zero or one. If the variable representing the program state is one then the program entered that state at some point during execution,

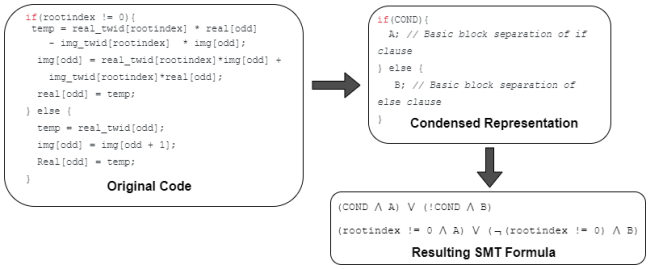

Figure 4: Branch SMT construction procedure

and if it is zero then the program never entered that state during this particular execution. To successfully model any function we need to deal with 3 main code constructs, arrays mentioned already, branches, and loops.

3.1.1 *SMT Branches.*

$$(if\ x\ then\ y\ else\ z) ==> (x \wedge y) \vee (!x \wedge z) \tag{1}$$

Formula 2 is the general rule for converting branches into an SMT representation, and Figure 4 provides an example take from a real code snippet.

The procedure starts by taking the code and condensing it to variables representing the basic blocks of the program under examination. We choose the basic block level because each basic block represents a specific program state. We also need variables representing the conditions that determine the function's control flow. After condensing the code, we follow the rules outlined in equation 2 and produce the corresponding SMT formula. Once the corresponding SMT formula has been created we expand it to include the original conditions from the source code.

Intuitively, the logic behind equation 2 is simple. Either the condition is true, and the program has entered the state corresponding with the if block, or the condition is not true and the program has entered the state corresponding with the else block. Of course, if there were more branches as in the case of an `else if` block, or if the basic blocks contained nested control flow operations, the SMT formula needs to be expanded accordingly. Any solution to this formula represents a possible path through this piece of the program and functions as a trace of the execution.

3.1.2 *SMT Loops.*

Representing loops is significantly harder than branches because by definition the code will be executed more than once, so the same code segment represents many possible states. However using loop unrolling we can assign a variable to each of the prospective states.

Figure **??** represents an overview of the procedure. Just like the branch SMT construction, first the program is condensed and variables replace each basic block. Then, for each loop we model the loop and termination condition using boolean logic, and in the diagram we refer to these as LC and T respectively. If the loop condition is true, and the loop condition is not true at the next iteration then we call that the termination condition. We can unroll the loop by representing each loop state with multiple variables, one for each iteration of the loop. In the event the loop ends early, before all the unrolled states then additional clauses simply evaluate

to false. In Figure **??** we show what this procedure looks like for the nested loops found in matrix multiplication. It's important to note that these are nested loops, so we don't include a fully expanded version of these 3 loops for space reasons, but in the actual SMT formula all of these are expanded and unique variables have to be used for each loop state. Note that loop unrolling pushes the sized of SMT formulas to blow up, making traditional SMT solving unviable and motivating our solution to use CLNs instead.

3.2 Variable and State Mapping

NOTE: This piece of the design is still in progress, but preliminary studies show that it is possible After the SMT formulas have been constructed, we need to perform some code analysis to determine which intermediary states, or combinations of intermediary states are the same. Ultimately we only care about the function outputs, but function outputs are built from the outputs of the intermediary states that are executed.

Just as in many code analysis approaches we define the inputs to a particular basic block B as the USE(B) set and the output as the DEF(B) set. The USE set of code variables is the set of variables that are used by the basic block but not defined by the basic block, and the set of USE variables is the set of variables that are defined in the basic block but are still accessible from outside the block. It should be noted that a variable can be a part of both sets.

Once variables have been substituted for basic blocks the data dependency tree can be constructed. Figure **??** shows a simple example of one such state matching procedure for a simple function. In this instance the dependencies of the DEF set of A match the dependencies of the DEF set of C and the same for B and D. Therefore if for a given input, both functions execute the matching output block, we say those functions are the same for that input.

While we are still working on the details of this algorithm in complex examples preliminary analysis indicates that this is possible.

One such analysis from multiple implementations of matrix multiplication revealed that even though the order in which operations take place, the ultimate assignment of the output variable has the same data dependence tree. As shown in Figure 9 if you create the data dependency tree for the output assignments, you get different looking trees that don't bear immediate resemblance. However if you trim the trees down to the minimum number of nodes you get the same tree. While there remains work to be done on how to finalize the details of this algorithm there seems to be a recognizable relationship between the assignment statements with additional effort.

3.3 CLN Construction

Continuous Logic Networks are based on a mapping from SMT formulas to Basic Fuzzy Logic. Most importantly, boolean operators such as $\wedge$ and $\vee$ are mapped to differentiable counterparts - t-norms ($\otimes$) and t-conorms ($\oplus$), respectively. Furthermore, the equation predicates map to smooth functions of identical characteristics. These mappings are consistent with the solutions of the inital SMT formula, but they are continuous and differentibale, which facilitates gradient descent. The functions's gradient should be increasing when approaching constraint satisfaction, and decreasing when

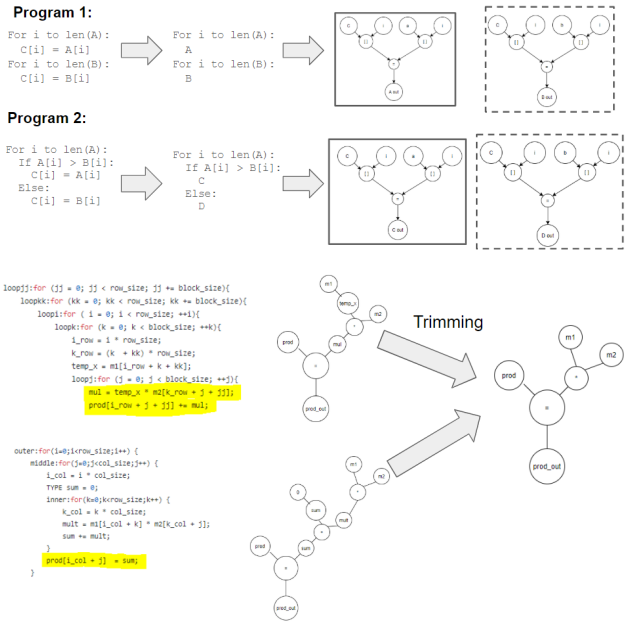

Figure 5: Program state matching for variable mapping. Conceptually and applied to Matrix Multiplication.

approaching constraint non-satisfaction. These characteristics can be achieve through the use of sigmoid functions. We conclude the following mappings:

$$x > c \rightarrow \frac{1}{1+e^{-B(x-c-\epsilon)}}$$

$$x \geq c \rightarrow \frac{1}{1+e^{-B(x-c+\epsilon)}}$$

$$x < c \rightarrow 1 - \frac{1}{1+e^{-B(x-c+\epsilon)}}$$

$$x \leq c \rightarrow 1 - \frac{1}{1+e^{-B(x-c-\epsilon)}}$$

$$x = c \rightarrow (1 - \frac{1}{1+e^{-B(x-c-\epsilon}}) \otimes (\frac{1}{1+e^{-B(x-c+\epsilon)}})$$

$$x \neq c \rightarrow 1 - (1 - \frac{1}{1+e^{-B(x-c-\epsilon}}) \otimes (\frac{1}{1+e^{-B(x-c+\epsilon)}})$$

It is easy to see that CLNs are based on a deterministic mapping. Hence, any given SMT formula can be translated into a CLN in linear time. The resulting CLN is then parameterized with a weight matrix and closely resembles what is commonly known as a neural network. In a consecutive training phase, the CLN then learns to approximate the discrete formula by using gradient descent and iteratively adjusting its weight parameters until the loss between smooth representation and discrete original converges to zero.

3.4 Monte Carlo Area and Similarity Metric

We define code similarity and apply a code similarity score based on the observation that overlapping solution space of two SMT formulas indicate program commonalities. The overlap of two smooth continuous functions, such as our formerly generated CLNs, is a

region bounded by both function's curves over a certain interval. This region is equivalent to the function's integral, often interpreted as area or volume. For SMT formulas however, this region describes the solution set of that formula. Consequentially we compute the integral of two overlapping CLN to quantify program similarity.

We observe that integrals are well defined as long as an antiderivative exists for the integrated function. Other methods to compute the integral, including the Riemann- and the Lebesque- integral, are based on enumerating infinitesimally small rectangulars under the curve. All such solutions are so called enumeration problems and cannot be solved efficiently. This issue is even more striking when integrating functions of multiple dimensions or over intervals of unknown shape, which is generally the case for neural networks - or CLNs.

Deterministic algorithms approximate such complicated integrals by defining a fixed input space and evaluating the function over a regular grid that neatly covers the defined space. The results then allow for an approximation of how many sample points fall below or above the curve. Note that the accuracy of such methods is deterministic but depends on the size of the grid.

Monte Carlo integration is a randomized technique to compute the numerical integral of any multidimensional function of unknown shape. A randomized choice of data points allows the Monte Carlo technique to outperform grid based techniques. Hence, we choose Monte Carlo as an efficient solution to our problem. It is important to understand that due to its random nature, each iteration yields slightly different results but for a large enough choice of N such results converge.

To facilitate an integration of form $I = \int_\Omega f(x)dx$, we solve the following equation in 4 steps:

$$I \approx Q_N = V\frac{1}{N}\sum_{i=1}^{N} CLN.train(x_1, x_2, ...y_1, ...y_n)$$

(1) Define input space Ω
(2) Draw N inputs $x_i \in \Omega$ uniformly at random
(3) Train the target CLN over all N inputs
(4) Compute the sum of solutions weighted by the input space volume

It is important to understand that numerical integrals of two functions may be of equal size, but bounding entirely disjoint regions. Therefore, we introduce a step-wise evaluation along the input space dimension, that helps us determine where overlaps are located. Our custom Dual-Monte Carlo function returns two lists of boolean values, indicating whether a sample point x_i lies in the integrated area (True) or outside of it (False). Correlating these two lists yields a similarity score between 0 and 1.

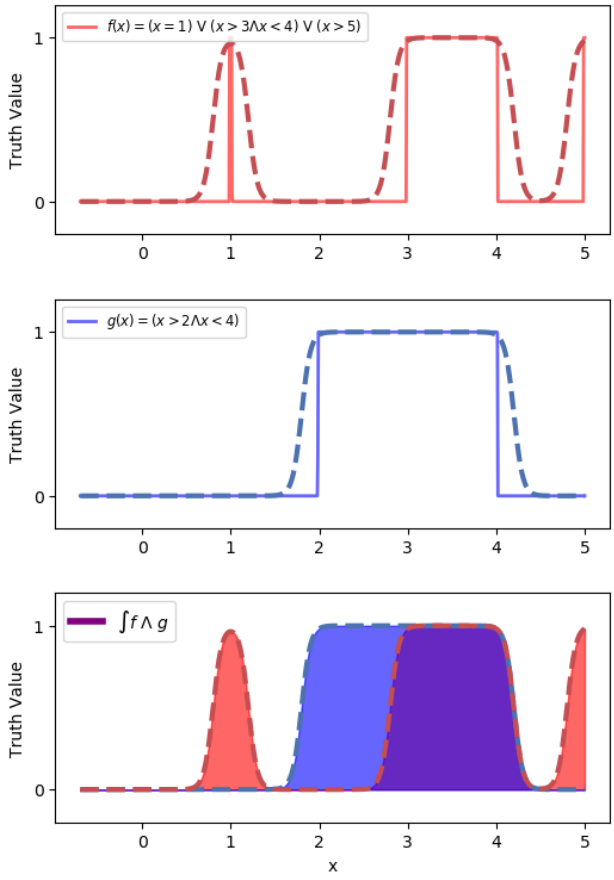

Figure 6: See how smooth? Soo smooth!

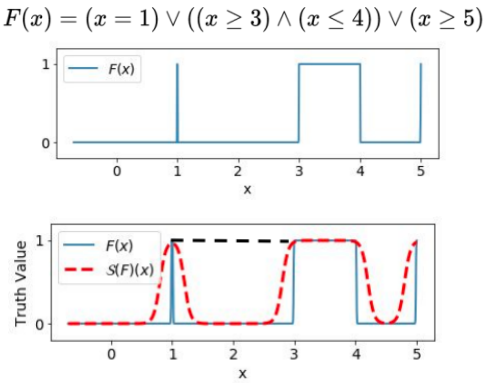

Figure 7: A traditional SMT formula and its truth values (top) and a differentiable mapping as generated by a CLN (bottom). The SMT graph (blue) is composed of discrete jumps, whereas the CLN graph (red) is smooth and continuous.

4 INTRODUCTION

Hardware accelerator technology has evolved to make them more accessible to the general computing public. While originally developers needed hardware expertise to build accelerators, the advent of FPGAs and CUDA created the opportunity for programmable implementations by skilled software developers.

This raises a new problem: With the new availability of hardware, how do you utilize it properly? Developers may have functions that can be accelerated and access to the accelerated versions, but they may not be aware of this opportunity. Luckily, regardless of how the accelerator was exposed or implemented it always exists some software representation of the accelerator's function. Therefore we can match candidates for acceleration with the appropriate hardware through code analysis.

While choosing an accelerator carries tradeoffs with respect to size, performance, and energy consumption, choosing between functionally identical accelerators is outside the scope of our problem. When assigning functions to accelerators in our context, only the inputs and outputs of the functions need to match.

Continuous logic networks (CLNs) have significantly increased the performance of traditional Satisfiable Modulo Theorems (SMT) solvers. By replacing the traditional boolean functions with smooth representations, CLNs can use gradient descent to learn the solution to an SMT formula with increased efficiency relative to state of the art SMT solvers. We present a technique to statically transform software seeking acceleration and the software modeling the accelerator into SMT formulas and then solve the SMT formulas using CLNs to implement a similarity metric.

This is the first work to apply CLNs in the context of code similarity and the accelerator assignment problem. There are some details that need to be resolved, but initial efforts indicate that with engineering effort our approach is feasible.

5 BACKGROUND

SMT Formulas have wide ranging applications from formal verification to symbolic execution. SMTs abstract the intrinsic meaning of a computer program and as such are an intuitive fit to approach the code similarity problem since if two programs are similar, they can be expressed by similar SMT formulas. Unfortunately SMT solving is NP-complete meaning long runtimes and possible timeouts. Established projects like S2E, MathSAT, and Z3 fail to overcome this performance issue, though it is an active area of research.

Traditional SMTs are restricted to boolean solutions. There is no notion of how close a possible solution is to being correct. Hajek [**?**] approached this problem by defining a class of logic that uses continuous values in the range [0, 1]. Figure 7 shows the standard 0,1 truth values associated with traditional SMT solving, and opposes it to a smooth differentiable representation. This representation permits techniques commonly used to solve optimization problems to also solve SMT formulas.

Continuous Logic Networks (CLNs) were defined by Ryan et al. [**?**] to abstract SMT Formulas as differentiable optimization problems. SMTs can be solved with CLNs by marking every variable as either an input term, a constant, or a learnable parameter. These can be efficiently trained using gradient descent, which makes the solution-finding process faster than traditional SMT solving.

6 DESIGN

Our system, AccFinder, has four consecutive steps shown in Figure 8 to calculate a numerical code similarity score (0-1).

SMT Representation: The transformation to SMT formulas is based on following standard SMT procedures in code analysis.

$$(if\ x\ then\ y\ else\ z) ==> (x \wedge y) \vee (!x \wedge z) \tag{2}$$

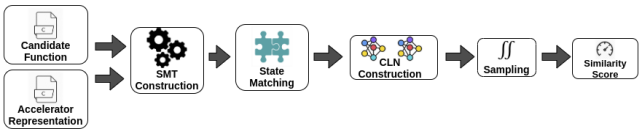

Figure 8: General Overview

Formula 2 is the general rule for converting a function's branches into an SMT representation. First, code is modeled by variables representing the possible states of the function under examination. We also need variables representing the conditions that determine the function's control flow. After condensing the code, we follow the rules outlined in equation 2 to produce the corresponding SMT formula. SMT formulas model loops through loop unrolling by assigning a new variable to the same basic block at each iteration.

State Mapping: After the SMT formulas have been constructed, we need to perform some code analysis to determine which intermediary states, or combinations of intermediary states, are the same. We define the inputs to a particular basic block `B` as the `USE(B)` set and the output as the `DEF(B)` set. Once variables have been substituted for basic blocks, the data dependency tree can be constructed as in Figure 9. While the trees don't look the same at first glance, after trimming we see that they are in fact the same.

CLN Construction: Continuous Logic Networks map boolean operators such as $\wedge$ and $\vee$ to differentiable counterparts — t-norms ($\otimes$) and t-conorms ($\oplus$), respectively. Furthermore, the equation predicates map to smooth functions of identical characteristics. Consequently, the function's gradient increases when approaching constraint satisfaction and decreases when approaching non-satisfaction. The resulting CLN is parameterized with a weight

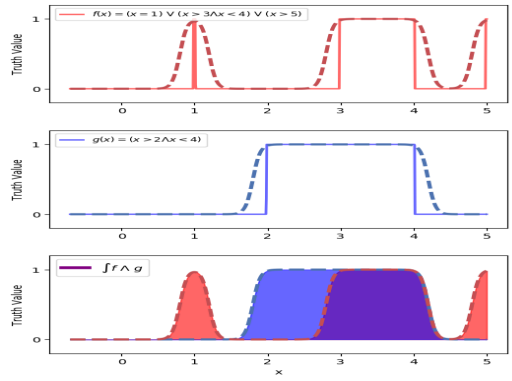

Figure 10: Overlapping integral similarity calculation

matrix and closely resembles neural networks. In a consecutive training phase, the CLN then learns to approximate the discrete formula by using gradient descent and iteratively adjusting its weight parameters until the loss between smooth representation and discrete original converges to zero.

Monte Carlo Integration and Similarity Metric: We define code similarity and calculate a score based on the overlapping solution space between SMT formulas or overlapping integrals. Monte Carlo integration is a randomized technique to compute the numerical integral of any multidimensional function of unknown shape based on random sampling as shown in Figure 10.

To facilitate an integration of form $I = \int_{\Omega} f(x)dx$, we solve the following equation in four steps:

$$I \approx Q_N = V\frac{1}{N}\sum_{i=1}^{N} CLN.train(x_1, x_2, ...y_1, ...y_n)$$

1) Define input space Ω, 2) Draw N inputs $x_i \in \Omega$ uniformly at random, 3) Train the target CLN over all N inputs, 4) Compute the sum of solutions weighted by the input space volume. Our custom Dual-Monte Carlo function returns two lists of boolean values, indicating whether a sample point x_i lies in the integrated area (True) or outside of it (False). Correlating these two lists yields a code similarity score between 0 and 1.

7 CASE STUDY

We began our investigation based on multiple implementations of algorithms in a common accelerator benchmark MachSuite [**?**]. We calculated identical similarity scores between blocked and cubed implementations of matrix multiplication when tested with matrices of sizes 4, 16, 36, and 64. These different implementations had 4, 16, 36, and 64 output states, respectively, after combinations of intermediary output states that effected the same final output state were combined. We also investigated the perceived similarity between a bulk implementation and a queued implementation of breadth-first search. However, our current state matching scheme failed to match states across those implementations since on different executions the output states will have different data dependencies. We continue to work on solving this problem.

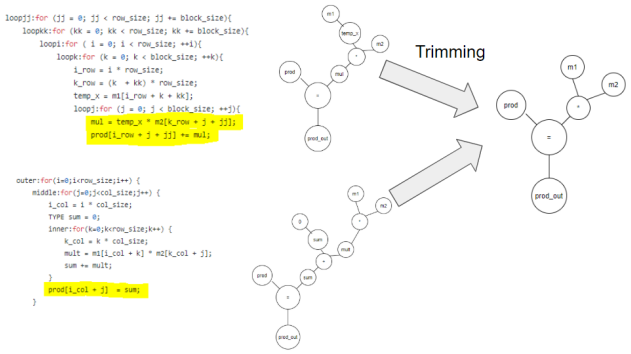

Figure 9: Function state matching for variable mapping applied to Matrix Multiplication.

8 ACKNOWLEDGEMENTS

The Programming Systems Laboratory is supported in part by NSF CNS-1563555, CCF-1815494 and CNS-1842456.