Formal Verification on Deep Learning Instructions of GPU

Jian (Jeffrey) Wang Jia Zhu

Advanced Micro Devices, Inc.
{Jeffrey.Wang, Jimmy.Zhu}@amd.com

***Abstract*- Formal verification on deep learning instructions of a GPU is a great challenge, due to the very high complexity of the DUV. We adopt formal C-RTL equivalence checking to prove that the functional behavior of the implementation of the deep learning instructions is identical to that of its reference C model. Several optimization techniques are adopted to aid the formal verification tool to finish the formal equivalence checking task in reasonable time.**

I. Introduction

Deep learning has been one of the leading drivers of growth in the semiconductor industry recently. In particular, *convolution/deep neural networks* (CNN/DNN) have led to breakthroughs such as beating the best human players in a Go game [1] and reducing error rates since 2011 from 26% to 3.5% in an image recognition competition [2]. GPUs have become the accelerators of the de facto standard for deep learning algorithms due to the massive computation threads available in GPUs and the computationally intensive nature of deep learning algorithms [3][4].

There are two phases of *neural networks* (NN), which are *training* (or learning) and *inference* (or prediction), and they refer to development versus production. Though there are many customized NN accelerators, such as TPU [5], proposed in academia and industry, most of them are focused on inference, and training still heavily relies on GPUs. As NN is based on statistical learning theory, it has the inherent nature of error resiliency. Along with the inevitable noise in the data set of NN, it is well appreciated that high-precision computation, such as Floating-Point 32, is not necessary in training nor inference. Half precision tanning (Floating-Point-16) is well supported in state-of-the-art NN libraries, and there is even research [6] showing that 16-bit fixed-point precision computation works as well. In inference, 8-bit computation, such as in [5], is very favored, and there are works, such as in [7], using computation of even lower precision.

The primary computing patterns for NN algorithms are matrix operations. Traditional GPUs use many fused multiply and add (FMA) operations in SIMDs to conduct parallel multiply and add operations on matrix elements. As low precision computation is sufficient for NN, using traditional Floating-point-32 FMA operations of GPU would waste both computing resources and memory bandwidth. New Deep Learning (DL) instructions can be integrated in GPU ISA to better support various kinds of computation patterns for deep learning applications. Since low precision data have shorter bit width, several FMA operations can be packed in a single 32-bit instruction to make the best use of computing resources and bandwidth, and improve the efficiency of matrix operations as well. For example, a 32-bit instruction can pack two Floating-point-16 FMAs or eight Integer-4 FMAs.

It is a great challenge to conduct the functional verification of the implementation of the DL instructions. The DL instruction contains much more operands and all the instructions contain many multiply and add operations. This situation makes it challenging for simulation-based verification. It took several hours of a simulation run for each instruction with only very low coverage. Unfortunately, it was not even practical to make a reasonable simulation-based verification plan, since there could be numerous possible cases of the product of corner cases in each operand.

We adopt formal verification techniques to beat the challenge in verification of the DL instructions. However, due to the large scale and complexity of the DL instructions, the formal problems are beyond the power of the formal verification tool. Consequently, we have to use several advanced techniques to guide the formal tool to solve the problems. Finally, we can formally prove that the implementation of

AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc.
Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.
© 2017 Advanced Micro Devices, Inc. All rights reserved.

the DL instructions in RTL have identical behaviors with the spec of them in C at the transaction level. The formal proofs take much less run time than simulation-based verification, and the proofs achieve full verification confidence.

The rest of this paper is organized as follows. Section II gives a brief introduction of related works. The overview of formal verification on DL instructions is presented in Section III. Section IV describes the optimization techniques adopted to conquer the convergence challenge, and a case study is given to demonstrate how the optimizations are used. Run times are show in Section V. A final summary is provided.

II. Related Works

Formal verification is becoming more and more popular in industry [8] with the increasing scalability which formal verification tools can support. There are many formal verification applications embedded in these formal verification tools, making formal verification a much easier task than before. Formal verification has become available to almost all verification engineers, rather than just those with formal expertise. The formal verification applications are based on one or a hybrid of three formal verification techniques, which are: 1) theorem proving; 2) model checking; and 3) equivalence checking.

Theorem proving is a subfield of automated reasoning and mathematical logic dealing with proving mathematical theorems by computer programs. Commercial use of automated theorem proving is mostly concentrated in integrated circuit design and verification. Since the Pentium FDIV bug [9], the complicated floating-point units of modern microprocessors have been designed with extra scrutiny. AMD and Intel use automated theorem proving to verify that division and other operations are correctly implemented in their processors [10] [11]. However, theorem proving is not as well adopted as the other two techniques in industry and commercial industry tools are rarely seen.

Model checking checks whether the implementation of the specification satisfies a set of properties (usually in form of SVA, PSL, CTL, etc.) implied by the specification. It is powerful in verification of control-intensive logics, such as arbiter and protocol. Model checking has attracted significant interest from both academia and industry for several decades [12] [13] [14] [15]. The most popular industry application of model checking is called formal property checking, such as FPV applications provided by EDA vendors in VC Formal [16], Jasper Gold [17], and Questa Formal [18]. There are successful industry works on formal property checking on GPUs, such as a recent work in [19].

Equivalence checking determines whether designs in different abstraction levels are consistent or not in function. Combinational equivalence checking (like what Formality does) is the most successful equivalence checking application. It checks the equality between the RT-Level and the gate-level implementation of the design, and it is very well adopted in industry for decades and it is very easy to use, making it a standard flow in IC design. With the development of formal equivalence checking techniques in recent years [20], now it is possible to check the functional equivalence at higher abstraction levels, such as C to RTL, RTL to RTL, and C to C, both on combinational and sequential logics. Commercial EDA tools are available in applications such as sequential equivalence checking and transaction-level equivalence checking [21] [20]. There are very successful works in formal equivalence checking of ALU floating-point operators between C-level and RTL recently, such as those presented in [22], and our previous work in [23].

This paper presents our work on C to RTL formal equivalence checking between deep learning instructions (Spec, in C++ language) and its corresponding ALU implementation in RTL. The implementation of DL instructions is much more complex than that of general ALU opcodes, which makes the formal verification much more challenging. It is beyond the power of the EDA tool to prove the correctness of the implementation. So, we adopt several optimization techniques and recursively use them to solve the formal verification problems.

III. Formal Verification on Deep Learning Instructions

In this work, we use formal verification to prove that the implementation (RTL of ALU) of the DL instructions have the identical transaction-level behaviors with their C models. The C-RTL formal equivalence checking tool is used. The C-RTL equivalence checking problem is demonstrated in Fig. 1. The C model is reconstructed as a *data flow graph* (DFG) and the DFG is composed with the RTL. Fig. 2 shows an example of C to DFG transformation. Then, the formal tool proves that the two models have identical output, assuming they have the same input with some additional constraints defining the legal input space. The principle of C to RTL equivalence checking is shown in Fig. 3.

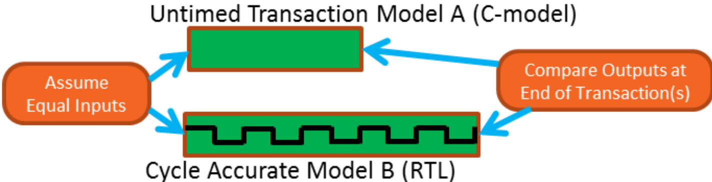

Figure 1. C to RTL equivalence checking problem.

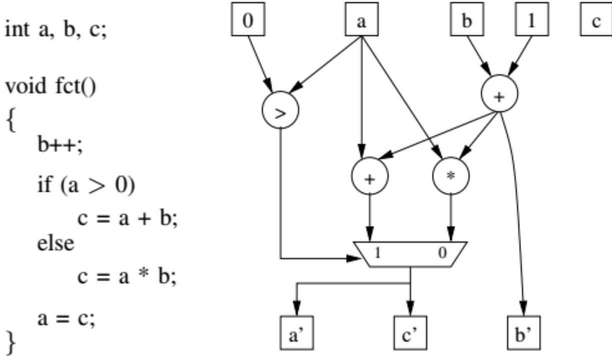

Figure 2. C to DFG transformation [20].

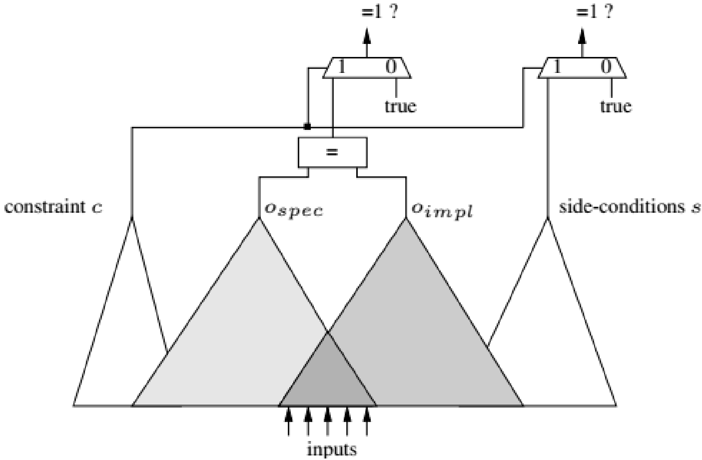

Figure 3. Principle of C to RTL equivalence checking [20].

The DL instructions conduct multiple complex floating-point and fixed-point multiply and add computations. Also, the instructions should make the best reuse of computing resources mutually and with other instructions in the ISA. The DL instructions support different computing patterns with different

precisions. Several calculations are packed to make the best use of computing resources as well as improve the efficiency of matrix operation. Consequently, the DUV of formal verification is very large and the logics of it are very complex.

The formal tool meets convergence issues with the DL instructions due to the large scale and complexity as described above. In other words, the formal tool is not able to prove the correctness of the design nor detect bugs. So, we adopt some optimizations to aid the formal tools to finish the proof. The optimization techniques are detailed described in Section IV.

IV. Optimizations

The basic theory behind the optimizations is to explore and prove the relations between the logics of RTL and C model, and then use the relations as hints for the formal tool. This process recursively makes the solving space collapse, so that in the end, the complexity is within the power of the formal tool. The optimization techniques include: 1) compositional reasoning, such as case-split and assume-guarantee; and 2) rewriting.

A. Compositional reasoning [24]

Why do formal methods meet problems with large scale designs? The answer is because the complexity of formal verification problems has at least exponential complexity with the number of property related variables. C-RTL equivalence checking is essentially to prove a formal property of equality, so it meets the complexity problem, too.

Compositional reasoning is to decompose the original complex properties to a set of easier ones. If all the easier properties stand, the original one stands too. The logic behind compositional reasoning is simple. One easier property usually has less variables than a harder one. The sum of exponential is supposed to be much less than the exponential of the sum, so the complexity is reduced. This can be formally analyzed as follows:

$\mathrm{Comp}(\boldsymbol{p}) = \mathbf{EXP}(\boldsymbol{wp}*\boldsymbol{np})$

$\boldsymbol{p}$: formal property, $\boldsymbol{np}$: number of variables in COI of $\boldsymbol{p}$, $\boldsymbol{wp}$: coefficient

$\{\boldsymbol{p_1}, \boldsymbol{p_2}, \ldots, \boldsymbol{p_i}, \ldots \boldsymbol{p_m}\} \rightarrow \boldsymbol{p}$ (decompose the original complex properties to a set of easier ones)

$\sum \mathrm{Comp}(\boldsymbol{p_i}) = \sum \mathrm{EXP}(\boldsymbol{wp_i}*\boldsymbol{np_i})$

$\boldsymbol{np_i} < \boldsymbol{np} \rightarrow \sum \mathrm{EXP}(\boldsymbol{wp_i}*\boldsymbol{np_i}) \ll \mathrm{EXP}(\boldsymbol{wp}*\boldsymbol{np})$

To make sure the set of easier properties is sufficient to deduce the original property, a proof is generated, which uses the easier properties as assumption to prove the original properties, so as to make sure there is no flaw in the decomposition.

The optimizations essentially decompose the difficult formal verification problem to a set of easier ones, and the relations between the impl and spec provide hints of partition for the decomposing. Since the problem is so complex, several layers of decomposition are needed to solve it.

In the analysis above, we assume that a formal property with less variables in its COI is easier. It is true in most cases. However, strictly speaking, this is not necessarily always true. Formal complexity has very complicated mechanisms with many factors, among which the variable number is one of them. For most formal verification practices, the variable number is a very dominate factor to estimate complexity and it is the only easily measurable factor. In addition, the root cause of why compositional reasoning works is that the problem space is properly partitioned by compositional reasoning. The partition is not conducted randomly but with the domain knowledge of design and problem. The verification engineers usually have deep insight of the design and they usually know how to make good partitions (i.e., parti-

tions with light mutual interactions). So, in formal verification practice, this assumption stands. Case-split and assume-guarantee are two simple methods to make "good partitions".

A.1. Case-split

In case-split, shown in Fig. 4, the proof of equivalence is spited into different cases, and then combine all the cases to deduce the original proof. Each case only a subset of logics are used, so the complexity is reduced.

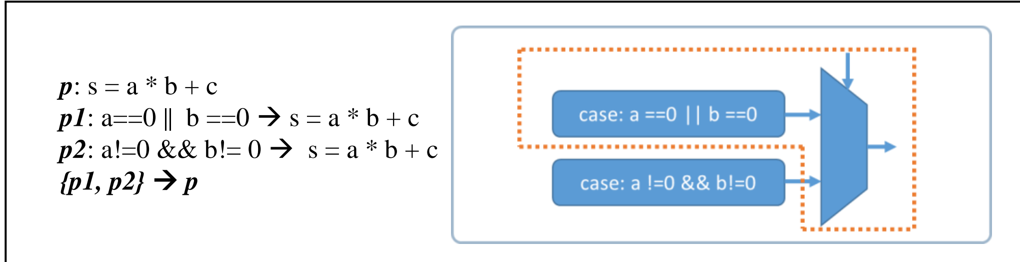

Figure 4. Case-split example.

A.2. Assume-guarantee

In assume-guarantee, shown in Fig. 5, an easier property is proved first, and then the easier property is used as an assumption to the final property. Also, each proof only uses a subset of the logics, so the complexity is reduced, too.

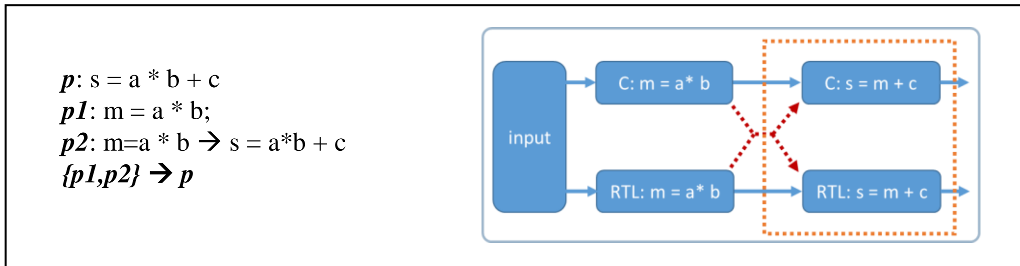

Figure 5. Assume-guarantee example.

For very difficult equivalence proofs, one step of compositional reasoning is not sufficient to solve the problem. Solving some "easier properties" may be still beyond the power of the formal verification tool. In this case, compositional reasoning can be recursively applied on the unsolved "easier properties". The recursive algorithm is shown as pseudo codes in Algorithm 1. First, we send the original equivalence check proof to the formal tools with a time threshold. The formal tool will return a status either of pass, fail, or time overflow. We are happy if it is pass, and we are less happy if it is fail, which indicates there are bugs or it needs additional debugging. We are unhappy if the status is time overflow, which indicates the proof is beyond the power of formal tools. Then, we decompose the proof to a set of easier ones using compositional reasoning, and run the algorithm for each proof until at last all the proofs are solved.

Algorithm 1. Recursive Computational Reasoning

---

p: property***, th:*** time threshold, ***M:*** the DUV
status: enum {succ(0), fail(1), time_overflow(2)}
Recurs_C_R(***p***, ***th***, ***M***)
{
    status = **brute_force_solve_p(*p*, *th*, *M*)**;
    if (***status*** == time_overflow)   {
        // assume-guarantee or case-split
            {***p1,p2,…pi,…pm***} = **Compositional_Reasoning(*p*** , ***M***);
            foreach(***pi***)   status |= **Recurs_C_R(** ***pi, th*** , ***M***);
    }
    return status;
}
brute_force_solve_p() :
    try to solve a property using formal tools with built-in optimizations
Compositional_Reasoning() :
    find a good decomposition of ***p*** to a set of sub-properties,
    this relies on manual efforts.

B. Rewriting

As we mentioned in the example of assume-guarantee in *A.2*, the complexity of formal proof can be largely reduced by proving the equivalence of the mapped multiply results first, then using it as an assumption for the following proof. In some cases, the mappings between C and RTL are not so simple and they cannot be easily found, since C and RTL do not use identical algorithm. For example, a uint32 multiply in C model could be simply written as,

```
uint64 C = uint32 A * uint32 B
```

In the RTL, there might not be a variable which can be mapped to C. The RTL implementation could be,

```
bit[31:0] c1 = a[15:0] * b[15:0]
bit[31:0] c2 = a[15:0] * b[31:16]
bit[31:0] c3 = a[31:16] * b[15:0]
bit[31:0] c4 = a[31:16] * b[31:16]
```

and c1, c2, c3, c4 could be directly used in successor logics, rather than generating the complete multiply result. In this case, the multiply results of C and RTL cannot be directly mapped. However, we want to map some complex logic such as multiplier in assume-guarantee, otherwise it will introduce too much complexity to the following proofs. In this case, we can locally rewrite the logics in C or RTL to make it easy for mapping. For the example above, the multiply in C model can be rewritten as,

```
uint64 C1 = (A & 0xffff) * (B & 0xffff)
uint64 C2 = (A & 0xffff) * (B >> 16)
uint64 C3 = (A >> 16) * (B & 0xffff)
uint64 C4 = (A >> 16) * (B >> 16)
uint64 C  = C1 + C2 + C3 + C4
```

Then C1, C2, C3 and C4 in C can be easily mapped with c1, c2, c3 and c4 in RTL.

C. A case study

A case study is shown here for better understanding of how to use the optimization techniques. The C-RTL formal equivalence checking process on an example of pack2_fp16 instruction is demonstrated as follows. It should be noted here that none of the examples in this paper are real industry designs, nor are

the instructions. Too many details in a real industry design would be misleading, and make it too complicated to demonstrate the principles of our work.

The spec for pack2_fp16 could be:

```
Result = Fused_multiply (A1, B1) + Fused_multiply (A2, B2) + C
Result: fp32; A1, B1, A2, B2: fp16; C: fp32
```

A typical C reference of this instruction could be like this:

```
fp32 a1 = fp16tofp32(A1); fp32 a2 = fp16tofp32(A2);
fp32 b1 = fp16tofp32(B1); fp32 b2 = fp16tofp32(B2);
fp32 mult_result1 = a1*b1; fp32 mul_result2 = a2*b2;
Result = mult_reslt1 + mult_result2 + C;
```

A typical RTL implementation of this instruction could be like what is shown in Fig. 6.

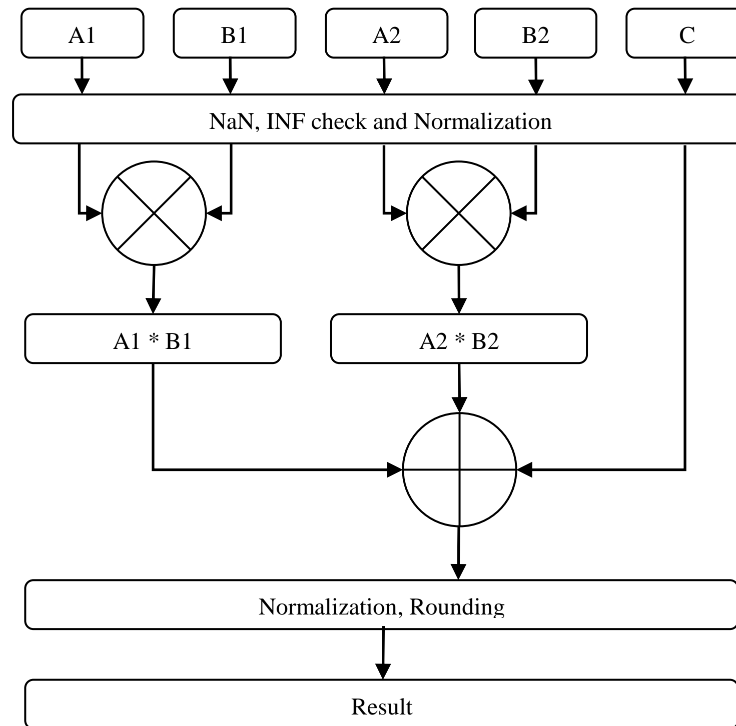

Figure 6. Diagram of RTL implementation of pack2_fp16.

The formal equivalence checking on this instruction cannot be solved directly by the formal tool. It takes five levels of recursive using compositional reasoning. Detailed structure of the proofs is shown as a tree in Fig. 7.

Root proof: Cmodel. Result == RTL. Result. This cannot be solved by the formal tool.

L1: Use case-split to deal with cases with or without NaN and Inf separately. The case with no Nan or Inf in operands cannot be solved.

L2: Use assume-guarantee to prove the multiply result in mantissa field of C and RTL are equivalent first. Then use it as an assumption to prove the final result. The later proof cannot be solved.

L3: This step uses assume-guarantee further, trying to prove that C and RTL have the same ADD results, and uses this as an assumption to prove the final result. The former one cannot be solved.

L4: The equivalence on ADD results of C and RTL is difficult to prove because of the difference in C and RTL implementation. C model uses FP32 multiply to calculate fused FP16 multiply, so there

are bit-shifts on multiplicands introduced by the transformation from FP16 to FP32, along with changes in exponent field accordingly. In consequence, there is difference between addends of C and RTL. In this step, different cases are created from the finite difference value of the exponent of C and RTL. Only 1 out of 22 proofs cannot be proved.

L5: Try to solve the unproved case in L4. Case-split is used again, splitting is based on which addend has the largest exponent. Finally, all the sub proofs can be solved in this step. Consequently, the root proof is finished.

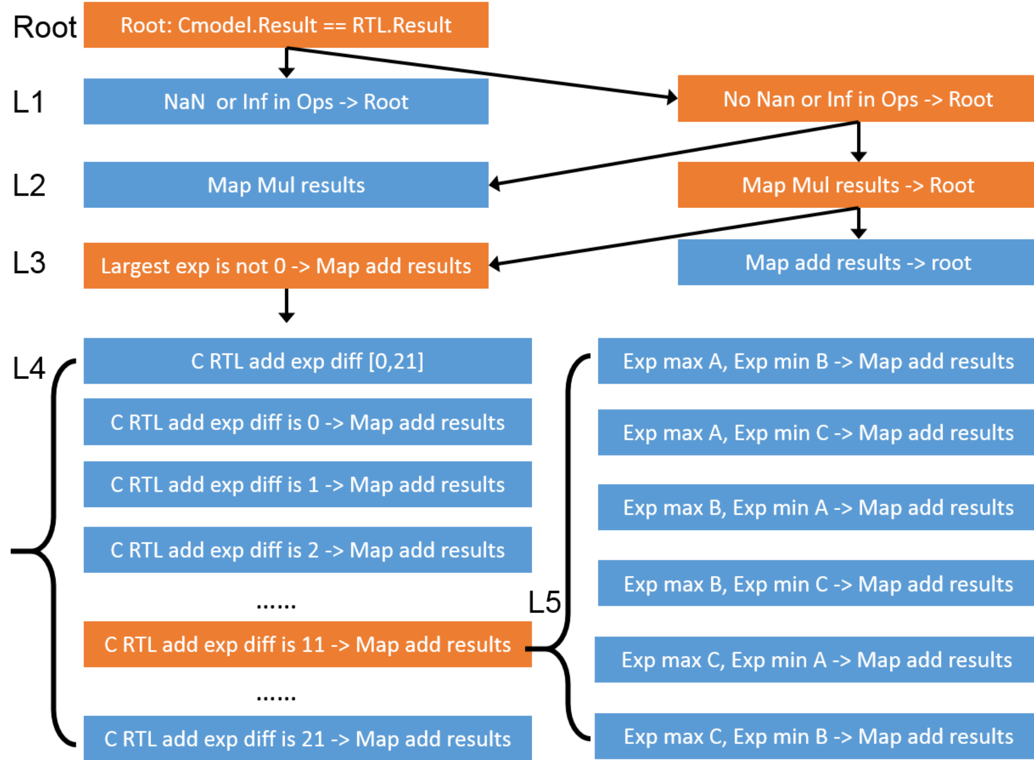

Figure 7. An example of decomposing and solving tree.
Nodes in orange color represent proofs beyond the power of the formal tool.

V. Results

With our optimizations, C-RTL formal equivalence checking can prove the correctness of the implementation of the DL instructions in reasonable time. In addition, the nature of decomposition of the optimization techniques makes it possible for the formal verification proofs to run in parallel with multi-threads. The run time and thread number for the DL instructions are shown in Table 1.

Table 1. Run time and thread number for DL instructions

Instruction name *	Floating-point-1	Fixed-point-1	Fixed-point-2	Fixed-point-3	Fixed-point-4	Fixed-point-5	Fixed-point-6
Num. mult	2	2	2	4	4	8	8
Num. add	4	2	2	3	3	5	5
Num. thread	30	1	1	1	1	1	1
Max thread time (s)	4938	53	52	3652	1164	57	75

* The instruction names are omitted here due to confidentiality.

VI. Conclusion

We present a work on formal verification for GPU deep learning instructions in this paper. Due to the large scale and complexity, the formal verification tasks are beyond the power of a formal verification tool. We adopt several optimization techniques and recursively use them to conquer this problem. Finally, the correctness of the implementation of deep learning instructions can be proved in a reasonable time (less than 1.5 hours each).

References:

[1] D. Silver, et al. Mastering the game of Go with deep neural networks and tree search. Nature, 529(7587), 2016.

[2] K. He, X. Zhang, S. Ren and J. Sun. Deep residual learning for image recognition. IEEE conf. on Computer Vision and Pattern Recognition (CVPR), pp. 770-778, 2016.

[3] A. Angelova, A. Krizhevsky, V. Vanhoucke. Pedestrian detection with a Large-Field-Of-View deep network. IEEE Conf. on Robotics and Automation (ICRA), pp. 704-711, 2015.

[4] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and Andrew, Ng. Deep learning with COTS HPC systems. The 30th International Conference on Machine Learning, pp.1337-1345, 2013.

[5] N.P. Jouppi, et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. The 44th International Symposium on Computer Architecture (ISCA), 2017.

[6] G. Suyog, A. Ankur and G. Kailash. Deep learning with limited numerical precision. arXiv: 1502.02551v1 [cs.LG],2015.

[7] R. Doshi, K.W. Hung, L. Liang and K.H. Chiu. Deep learning neural networks optimization using hardware cost penalty, IEEE International Symposium on Circuits and Systems (ISCAS), pp. 1954-1957, 2016.

[8] H. Foster, Trends in Functional Verification-A 2014 Industry Study, Design Automatic Conference (DAC) 2015.

[9] D. Price, Pentium FDIV flaw-lessons learned, IEEE Micro, vol15, issue 2, 1995.

[10] S.F.Oberman, Floating point division and square root algorithms and implementation in the AMD-K7TM microprocessor, Proc. IEEE Symposium in Computer Arithmetic, 1999.

[11] R. Kaivola and N. NarasimhanFormal, verification of the Pentium4 floating-point multiplier, Proc. Design Automation Conference (DAC), 2012.

[12] E.M. Clarke, and E.A. Emerson, “Synthesis of synchronization skeletons for branching time temporal logic,” In Logic of Programs: Workshop, Yorktown Heights, Lecture Notes in Computer Science, NY, Springer-Verlag, 1981.

[13] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, “Symbolic model checking: 1020 states and beyond,” Information and Computation, vol. 98 no.2 , pp. 142–170, 1992.

[14] K. L. McMillan, “Interpolation and SAT-based model checking”, in Proc. of Computer-Aided Verification (CAV 03), 2003, pp.1-13.

[15] A.R. Bradley , “SAT-based model checking without unrolling” , Verification, Model Checking, and Abstract Interpretation, Springer Berlin Heidelberg, 2011, pp.70-87.

[16] VC Formal, https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html.

[17] Jasper Gold, http://www.jasper-da.com/products/jaspergold_apps.

[18] Questa Formal, http://www.mentor.com/products/fv/questa-formal-verification-apps.

[19] J. Zhu, C. Yan and N. Wang, Formal Proof for GPU Resource Management, Design and Verification Conference and Exhibition (DVCON), 2017.

[20] A. Koelbl, R. Jacoby, H. Jain and C. Pixley, Solver Technology for System-level to RTL Equivalence Checking, Proc. Design Automation and Test Conference of Europe (DATE) 2009.

[21] SLEC, https://www.mentor.com/products/fv/questa-slec.

[22] T. W. Pouarz, and V. Agrawal, Efficient and Exhaustive Floating Point Verification Using Sequential Equivalence Checking, Design and Verification Conference and Exhibition (DVCON), 2017.

[23] J. Wang and J. Zhu, Conquer difficult C-RTL formal verification problems using recursive compositional reasoning, design/IP track, Design Automation Conference (DAC) 2017. http://www2.dac.com/54th/proceedings/slides/40_5.pdf

[24] S. Berezin, S. Campos and EM. Clarke, Compositional Reasoning in Model Checking, COMPOS'97, LNCS, Springer.