A Formal Specification of Tensor Cores via Satisfiability Modulo Theories

Benjamin Valpey, Sreepathi Pai (advisor)
Department of Computer Science, University of Rochester

SC23 DENVER NOV 12–17

UNIVERSITY of ROCHESTER

Introduction
The need for a specification

Formal hardware specifications - useful for both developers and hardware designers
Formal specifications help:
- Test algorithms for portability
- Automatically port algorithms to new HW
- Check compiler correctness

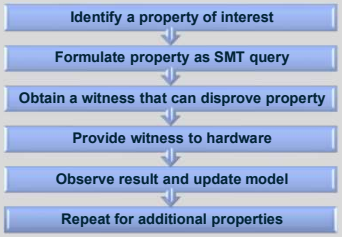
Hardware changes fast, is often undocumented or proprietary
Tools help automatically document/explore hardware are needed!

How do we replicate the behaviors of undocumented hardware units and reason about them?

- Users currently resort to manual tests and experiments to build understanding
- Results in fragmented and contradictory knowledge base

Our Method
Formalization via SMT

We propose a method to help automate the formalization of hardware implementations



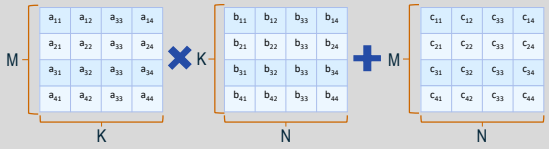
Satisfiability Modulo Theories (SMT) is an excellent way to encode and reason about hardware implementations automatically!

- Can specify precise semantics and define properties, e.g., associativity, monotonicity
- Properties can be verified with SMT solvers
- Better than a C emulation
  - Difficult to reason about C programs

Background
What are tensor cores and what do we know about them?

Tensor Cores

- Specialized units for matrix multiply-accumulate (MMA)
- Offer dramatic performance improvement, as much as 12x TFLOPS vs Pascal's floating-point units[5]
- Numerical behavior not documented by NVIDIA



Experiments and Results
What are the properties of Tensor Cores?

Automatically Generate Discriminating Tests

- Begin with design space for hardware implementation
- Create SMT queries to identify hardware behavior
- Utilize design space explored by prior work
  - Hickmann and Bradford [1]
  - Fasi et al. [2]
- Incrementally build formal SMT model as properties are tested
  - Results in a formal and executable model
  - Easily falsified against hardware
- **Found 2 discrepancies with prior work**
  - Fasi et al. concludes tensor cores use RTZ rounding
    - Tensor cores do not use guard bits, needed for IEEE-754 RTZ
    - Tensor cores instead perform truncation
  - *Due to lack of normalization*, 3 carry-out bits are needed for 5 term accumulation
    - Fasi et al. concludes tensor cores use only 2
    - Our method proves 3 are needed, and tensor cores use 3

```
from cvc5.pythonic import *  # can also import Z3
s = Solver()
a = FreshConst(Float16())
b = FreshConst(Float16())
for rm in (RNE(), RTZ(), RTN(), RTP()):
    fp16_result = fpToFP(RTZ(), fpMul(rm, a, b), Float32())

    fp32_result = fpMul(rm, fpToFP(RTZ(), a, Float32()),
     fpToFP(RTZ(), b, Float32())
    s.add(Not(fp16_result == fp32_result))
if s.check() == sat:
    m = s.get_model()
    # record values for a and b
    print(m.eval(a), m.eval(b))
else:
    print("Unsat/unknown")
```

Heed other factors that may affect result

Encode constraints for the possibilities

Have solver find values for which result differs

Property	Behavior	Query Time (cvc5)
Precision of products and sums	Performed in single-precision regardless of output	0.027s (products) 0.12s (sums)
Rounding of intermediate sums	Not equivalent to any IEEE-754 rounding mode Hardware performs truncation	0.017s
Rounding of final sums for FP16 result	IEEE-754 compliant RNE	0.008s
IEEE-754 addition is not associative, so in what order are the terms accumulated?	Not IEEE-754 compliant. Significands are aligned once based on maximal magnitude. Intermediate sums are not normalized; order does not matter	0.25s (associativity) 0.25s (normalization)
How many carry bits are used due to lack of normalization?	Three carry bits are used	72.793s
Handling of NaNs, ∞	IEEE-754 compliant	0.001s

Scan to see data supporting each property, also Ootomo vs Markidis results

Putting the Model to Use
Comparing two algorithms written for tensor cores

We study two similar algorithms that utilize tensor cores

- Markidis et al. [3]
  - Use tensor cores to perform single-precision multiplication
  - Recover loss of precision using a residual matrix
- Ootomo and Yokota [4]
  - Improve upon Markidis et al.'s method
    - Find error is due to rounding of tensor cores
  - Better accuracy for slight performance hit

Question: Is the Ootomo and Yokota method always more accurate than Markidis et al.?

- Implement each method using our model
- Assert for some input, Ootomo Error > Markidis error
- Use SMT solver to find such input
- cvc5 finds such inputs in <5 minutes!
  - Ootomo and Yokota sometimes worse
  - Less error due to lack of normalization!

Conclusion

- Knowledge of hardware implementation benefits developers
  - Allowed Ootomo and Yokota to improve upon Markidis' et al.
- SMT is great way to expose hardware implementations
  - Can help identify tests for hardware behavior
  - Provides formal model that can be reused for property checking/program analysis

References

[1] B. J. Hickmann and D. Bradford, "Experimental Analysis of Matrix Multiplication Functional Units," 2019 IEEE 26th Symposium on Computer Arithmetic (ARITH), pp. 116–119, 2019.

[2] M. Fasi, N. J. Higham, M. Mikaitis, and S. Pranesh, "Numerical Behavior of NVIDIA Tensor Cores," PeerJ Computer Science, vol. 7, p. e330, 2021.

[3] S. Markidis, S. W. Der Chien, E. Laure, I. B. Peng, and J. S. Vetter, "Nvidia Tensor Core programmability, Performance & Precision," in 2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW). IEEE, 2018, pp. 522–531.

[4] H. Ootomo and R. Yokota, "Recovering Single Precision Accuracy from Tensor Cores While Surpassing the FP32 Theoretical Peak Performance," The International Journal of High Performance Computing Applications, vol. 36, no. 4, pp. 475–491, 2022.

[5] https://www.nvidia.com/en-us/data-center/tensor-cores/