Formal Verification of Domain Specific Languages

Youngjae Moon

Contents

1. What is a Domain Specific Language (DSL)?
2. Why DSLs and new programming languages?
3. About Exo
    1. Background and Motivation
    2. Why developing acceleratd high performance libraries is hard?
    3. Optimizing Code by Exo
    4. When to use Exo
    5. Formal Methods for Exo
    6. What I plan to do for the final project

What is a Domain Specific Language (DSL)?

- A DSL is a computer programming langauge designed for a particular domain.
- Examples:
  - Structured Query Language (SQL)
  - HyperText Markup Language (HTML)
  - Cascading Style Sheets (CSS)
  - MATLAB
  - LaTeX
  - Verilog
  - Regular expressions
  - Yacc/Bison

Why DSLs and new programming languages?

- Based on Moore's law, computing power doubles every two years as the density of transistors in a chip doubles every two years.
- This allows business ideas that did not have commercial values in the past to be commercialised.
- Developing an existing programming language to support for new domains created make it heavier and slower.
- It becomes easier to instead develop a new lighter and faster programming language designed specifically for new domain.
- DSLs reduces the workload of a programmer to optimize its code for a particular domain
  - The built-in compiler for the DSL can do it instead.

About Exo

- Exo is a domain-specific programming language that helps low-level performance engineers transform very simple programs that specify what they want to compute into very complex programs that do the same thing as the specification, only much, much faster.
- Developed by Professor Gilbert Bernstein at University of Washington and other people

Background and Motivation

- The highest performance hardware made today (such as Google’s TPU, Apple’s Neural Engine, or Nvidia’s Tensor Cores) power key scientific computing and machine learning kernels: the Basic Linear Algebra Subroutines (BLAS) library, for example.
- However, these new chips—which take hundreds of engineers to design—are only as good (i.e. high performance) for application developers as these kernels allow.
- Unlike other programming languages and compilers, Exo is built around the concept of exocompilation.
- Traditionally, compilers are built to automatically optimize programs for running on some piece of hardware.
- This is great for most programmers, but for performance engineers the compiler gets in the way as often as it helps.
- Because the compiler’s optimizations are totally automatic, there’s no good way to fix it when it does the wrong thing and gives you 45% efficiency instead of 90%.

Why developing acceleratd high performance libraries is hard?

- First, in contrast to conventional programs on general-purpose processors, the hardware-software interfaces to accelerators are both complex
  - including specialized memories, exposed configuration state, and complex operations
  - and highly diverse, with different complexities unique to each accelerator.

Why developing acceleratd high performance libraries is hard?

- Second, the rates of change at different levels in the stack – from applications to hardware Instruction Set Architecture (ISA) – are inverted
  - accelerator architectures change more rapidly than the essential functions which run on them
    - e.g., mobile phone SoCs are rebuilt every year, with major revisions to nearly every accelerator block, while the BLAS standard changes much more slowly
  - and the implementation of these functions to most efficiently use the hardware is iterated more quickly, still.
  - This is especially acute during accelerator development, where target application workloads are often fixed, while both the hardware architecture and kernels mapping to it are iteratively co-designed to maximize performance and efficiency.

Optimizing Code by Exo

- With exocompilation, we put the performance engineer back in the driver’s seat.
- Responsibility for choosing which optimizations to apply, when, and in what order is externalized from the compiler, back to the performance engineer.
- This way they don’t have to waste time fighting the compiler on the one hand, or doing everything totally manually on the other.
- At the same time, Exo takes responsibility for ensuring that all of these optimizations are correct.
- As a result, the performance engineer can spend their time improving performance, rather than debugging the complex, optimized code.

Optimizing Code by Exo

- Another key part of exocompilation is that performance engineers can describe the new chips they want to optimize for, without having to modify the compiler.
- Traditionally, the definition of the hardware interface is maintained by the compiler developers.
- However, for most new accelerator chips, the hardware interface is proprietary.
- It also changes more frequently than for general purpose chips.
- Currently, companies have to maintain their own fork of a whole traditional compiler, modified to support their particular chip.
- This requires hiring teams of compiler developers in addition to the performance engineers.

Optimizing Code by Exo

- It has been shown that we can use Exo to quickly write code that’s as performant as Intel’s hand-optimized Math Kernel Library.
- There is also have an ongoing collaboration with UC Berkeley to create code for GEMMINI, their open-source machine learning accelerator.

When to use Exo

1. Are you optimizing numerical programs?
2. Are you targeting uncommon accelerator hardware or even developing your own?
3. Do you need to get as close as possible to the physical limits of the hardware you're targeting?

- If you answered "yes!" to all of three questions, then Exo might be right for you!
- In particular, if you just want to optimize image processing code for consumer CPUs and GPUs, then Halide might be a better fit.

Formal Methods for Exo

- Exo is written in Python. Hence PySMT is being used for three main purposes:

1. Verifying correctness
   - PySMT can be used to formally verify:
     - high-level Exo code,
     - the transformations applied by the Exocompiler
     - the generated low-level code for the target hardware accelerators
   - are correct with respect to their specifications.
2. Optimizing transformation
   - PySMT can help in determining the optimal set of transformations and optimizations to be applied by the Exocompiler to generate efficient code for the target hardware accelerators.
3. Code synthesis
   - PySMT can be utilized to synthesize code fragments that meet specific requirements,
     - such as performance constraints or hardware compatibility
     - which can then be integrated into the generated low-level code

What I plan to do for the final project

- I will try to verify the correctness of Exo programs. Note that I have not done any yet. ☹

1. Write sample Exo programs
2. Use Exocompiler to compile the code into target hardware accelerator
3. To formally verify the correctness of this transformation using PySMT, we can follow these steps:
   1. Encode the original and transformed programs as logical formulas in PySMT.
   2. Use an SMT solver to prove that the original and transformed programs are equivalent.

Any questions?

Thank you!