GSoC 2017 Application Kanchana Ruwanpathirana: Implement Common Subexpression Elimination for SymEngine

Kanchana Ruwanpathirana edited this page on 3 Apr · 1 revision

Personal Background

Details

Name : Aravinda Kanchana Ruwanpathirana

University : University of Moratuwa, Sri Lanka

Email : kanchana.ruwanpathirana@gmail.com

GitHub : kanchanarp

A Short Bio

I am a final year student pursuing a BSc Engineering degree of Computer Science and Engineering at University of Moratuwa, Sri Lanka.

I have been involved in programming and software development aspects since early 2013 and I have a good hands on knowledge of C, C++, Python, Java and several other programming languages. I have mainly used these languages in the projects I have done at the university and also in programming competitions like IEEEXtreme. At IEEEXtreme 10.0, I was fortunate to be a member of the team that was ranked 86th globally and 2nd in Sri Lanka. I have a good knowledge of the version controlling using git and github as it is commonly used in our university projects. Apart from the main programming languages, I have good knowledge of PHP, JavaScript, JSP and several other scripting languages also.

I also have a good mathematical and research background. I participated in International Mathematics Olympiad 2012 and also in International Mathematics Competition for Undergraduates 2016. In the latter, I was able to achieve a Bronze medal.In terms of research experience, I participated in a research internship at Singapore University of Technology and Design where I worked in a team where we used machine learning techniques to learn emotions in music. In terms of development theory knowledge, I have taken courses on Data Structures and Algorithms, Object Oriented Software Development, Programming Languages, Formal Methods in Software Engineering to name a few. In terms of mathematics related courses, I have taken courses on Numerical Methods, Graph Theory, Calculus, Linear Algebra, Differential Equations and Applied Statistics at University of Moratuwa.

I was introduced to SymEngine by Isuru Fernando back in January 2016, where I worked briefly on a small modification on a few variables on SymEngine. I was unable to continue working on this later that year due to the internship and other such issues. So, I joined this year to apply for GSoC from SymEngine.

Platform Information

I use a dual boot system with following settings in two cases,

Windows

Pages 451

Find a Page...

Home

About implementing special functions

Adding Someone to AUTHORS

Algebras in SymPyCore

Allow arbitrary choice of printer to be used by init_printing()

Args Invariant

Assumptions

Assumptions history

assumptions_handlers

Automatic Simplification

Benchmark results expand2b, SymEngine

Buildbot

Canonicalization

Change Log

Closing an Issue

Show 436 more pages…

OS : Windows 10

Editor : Notepad++

Compiler : MSVC (Visual Studio 14 2015)

Bignum Library : MPIR

Ubuntu

OS : Ubuntu 16.04

Editor : gEdit

Compiler : gcc

Bignum Library : GMP

Related Technology Stack

C++ : Have been using C and C++ for over three years. C++ is the language mainly used in coding activities. Have sufficient knowledge of object oriented and related development paradigms in C++.

Git : Have been using git for many different group projects and collaborations within the university. Has a good knowledge of the basic structure of the Git commands and has sufficient experience with using Git.

Contributions to SymEngine

(Merged) Version libsymengine.so #1239; This is concerned with the issue #1226. This versions the shared library, libsymengine.so with major.minor version.

(Closed) type_id_ variable added in place of the virtual method #749;This is concerned with #387. This was a performance related issue where the earlier virtual method get_type_code() to a non-virtual method using type_code_id

GSoC Time Period Personal Schedule

In terms of my commitments during the GSoC 2017 period, as we are not closed for summer I may have few classes during the time of GSoC. But according to my current schedule my lectures are only limited to Monday and Wednesday and a few two hour lectures at that. I am mostly free on the other dates and will not be taking any other work assignments for the time being.

I may have an exam starting from 26th of June to 10th of July as per the current timetable but as I only have four modules this semester, it will be only 4 papers and would not be an issue with the GSoC tasks. Apart from that, at the moment I have no other commitments apart from the GSoC work, for the dates of GSoC competition.

Implement Common Subexpression Elimination for SymEngine

Project Overview 1

Common subexpression elimination (CSE) is a compiler optimization technique that is used to make the calculations more efficient by finding redundant expression evaluations, and replacing them with a single computation. The key goal of this project is to develop a method to implement common subexpression elimination for symbolic computations in SymEngine. When we consider the current state of CSE in SymEngine and related projects, SymPy 1.0 has a CSE implementation. In implementing, integration of CSE into LambdaVisitor and LLVMDoubleVisitor can be carried out. In case of LLVMDoubleVisitor, this is done through LLVM for the case of real numbers, but when it comes to complex numbers, LLVM is not able to recognize common-subexpressions. So, the implementation is to address these cases and also to develop benchmarks for the performance when using CSE.

SymPy and CSE 2 3

SymPy 1.0 is currently equipped with modules to carry out common subexpression elimination for symbolic computations. This is mainly done through the modules available in the sympy / simplify which hosts the modules cse_main and cse_opts and tests for the common subexpression elimination.

SymPy provides a good way of common subexpression elimination when it comes to real cases as in the sense of using something like,

Import sympy as sp sp.cse(Add(Pow(x+y,2),x+y))

Which results in, the following structure where x0 is the common statement and the expression is returned in common subexpressions as single expression.

([(x0, x + y)], [x0**2 + x0])

However, tests suggest that when it comes to the complex numbers, this case may differ based on the input expression and the way it is provided. Lets consider the case where we take (x+iy)2+ (x+iy) which is also equivalent to (x)2-(y)2+2ixy +(x+iy)

But if we run sympy cse on the two cases what we get is the following outputs. Case 1 : ([(x0, x + I*y)], [x0**2 + x0]) Case 2 : ([(x0, I*y)], [x**2 + 2*x*x0 + x + x0 - y**2])

This may be due to the complexity in computations in complex number multiplication in symbolic case.

Common Subexpression Elimination Algorithms

SymPy CSE Main Algorithm 4

We will look at the high level architecture of the CSE algorithm implemented in SymPy. We will use the code segments from SymPy to study this algorithm.

This algorithm is a tree based algorithm where the expression tree is used and reduced to develop the version with CSE carried out. The parameters used in the method are, The sympy expressions to reduce. (expr) The expressions to be substituted before any CSE action is performed. (opt_subs) The order by which Mul and Add arguments are processed. (order)

The algorithm mainly consists of two different sub processes, one to find sub-expressions and another to rebuild the tree.

The method to find sub-expressions use a simple method where there are two separate lists to maintain the subexpressions seen so far and the subexpressions that has been marked for the elimination. Given a expression this method checks if that expression is of a type that is possible to be used in common subexpression elimination. If so then if the expression has not been seen before, add it to the list of seen subexpressions and if it is already in there then it is added to the list for elimination.

```
to_eliminate = set()
seen_subexp = set()

def _find_repeated(expr):
    if not isinstance(expr, Basic):
        return
    if expr.is_Atom or expr.is_Order:
        return
    if iterable(expr):
        args = expr
    else:
        if expr in seen_subexp:
            for ign in ignore:
                if ign in expr.free_symbols:
                    break
            else:
                to_eliminate.add(expr)
                return
        seen_subexp.add(expr)
```

```
            if expr in opt_subs:
                expr = opt_subs[expr]
            args = expr.args
        list(map(_find_repeated, args))
```

When it comes to the method for rebuilding the tree, this method simply regenerates the tree with the order of the operations preserved to the possible extent. The following code provides an overview of this functionality.

```
replacements = []
subs = dict()

def _rebuild(expr):
    if not isinstance(expr, Basic):
        return expr
    if not expr.args:
        return expr
    if iterable(expr):
        new_args = [_rebuild(arg) for arg in expr]
        return expr.func(*new_args)
    if expr in subs:
        return subs[expr]
    orig_expr = expr
    if expr in opt_subs:
        expr = opt_subs[expr]
    if order != 'none':
        if isinstance(expr, (Mul, MatMul)):
            c, nc = expr.args_cnc()
            if c == [1]:
                args = nc
            else:
                args = list(ordered(c)) + nc
        elif isinstance(expr, (Add, MatAdd)):
            args = list(ordered(expr.args))
        else:
            args = expr.args
    else:
        args = expr.args
    new_args = list(map(_rebuild, args))
    if new_args != args:
        new_expr = expr.func(*new_args)
    else:
        new_expr = expr
    if orig_expr in to_eliminate:
        try:
            sym = next(symbols)
        except StopIteration:
            raise ValueError("Symbols iterator ran out of symbols.")
        if isinstance(orig_expr, MatrixExpr):
            sym = MatrixSymbol(sym.name, orig_expr.rows,
                orig_expr.cols)
            subs[orig_expr] = sym
            replacements.append((sym, new_expr))
            return sym
        else:
            return new_expr
```

The entire algorithm is reflected in the cse_main module under tree_cse which is the called algorithm in cse function. The only additional thing in cse function that it provides the pre and post processing of the cse reduced tree as well.

CSE Algorithm Based on Polynomials

The paper 5 discusses a special algorithm that uses kernels to calculate the common subexpressions. This algorithm, in its current form, is limited in the performance to the polynomials. However, it may be possible to extend this algorithm to non-polynomial cases.

This algorithm is reliant on the following terminology. In this algorithm a product of literals each raised to some non-negative integer power along with an associated positive or negative sign is known as a cube and a kernel is a cube-free expression given a polynomial P and a cube. Co-kernel is a cube utilized to get the kernel.

The algorithm has 3 sub algorithms, Kernel algorithm Distill algorithm and, Condense algorithm

The kernel algorithm is used to create two kernels in the sense of Kernel Cube Matrix and and Cube Literal Incident Matrix which are used in cases where there are respectively, multiple cube common subexpressions and single cube common subexpressions.

The kernel intersections and cube intersections are used in this algorithm to come up with the common subexpressions.kernel intersections algorithm mainly performs the calculations to find the largest rectangle available in Kernel Cube Matrix, which is determined by the '1' entries in the Kernel Cube matrix. This calculation and instructions on the pseudo code of the algorithm is available in the above mentioned research paper. While this method may be effective in working with polynomials, the possibility of use in non polynomial cases are yet to be studied.

Further study should be carried out on using this approach and how this would work out in the case of complex field operations and how this could be extended to functions such as sin(), cos(), exp() and etc.

Global Common Subexpression Elimination 6

This algorithm is the classical set of algorithms to perform global common subexpression elimination. In this algorithm, we are using basic blocks of expressions.

First we have to understand a few key terminology. Given a block b, we have three important functions in this algorithm. AVAIL(b) the set of expressions available on entry to b. NKILL(b) the set of expressions not killed in b. DEF(b) the set of expressions defined in b and not subsequently killed in b.

The algorithm is to simply calculate DEF(b) and NKILL(b) for all the blocks b.Then calculate AVAIL(b) for all blocks b and carry out a process called value numbering for the blocks using AVAIL.

This is the high level overview of the algorithm. The notes provided here gives a much more deeper explanation on how the each components in this algorithm work.

Common Subexpression Elimination Implementation

The CSE implementation for the SymEngine will mainly follow the approaches in the SymPy CSE module and will be changed as necessary for improvements. The main algorithm would be the same underlying algorithm in SymPy though few changes may be carried out to ensure the efficiency as well as to allow for CSE in complex system cases.

Prototype for CSE tree algorithm (based on SymPy)

The following provides a sample prototype function for tree cse that is based on the implementation in SymPy. This is not a complete implementation and only gives an overview of the structure in general. The methods are _find_repeated and _rebuild.

The two vectors to_elminate and seen_subexp provides vectors of the elements to eliminate as well as all seen subexpressions. This is just a prototype based on C++ and Expression is a prototype class to hold the expressions.

```
vector<Expression> to_eliminate=new vector<Expression>();
vector<Expression> seen_subexp=new vector<Expression>();
vector<Expression,Expression> replacements=new vector<Expression,Expression>();
vector<Expression> reduced_exprs=new vector<Expression>();
```

```
template <typename T>
constexpr bool is_iterable();
```

Find repeated is used to find the subexpressions that are repeated in the given set of expressions. And the function rebuild is to rebuild the expressions. The implementations are not included and the implementation is partial and does not contain all the elements. The classes Expression and Symbol are dummy classes to represent the expressions and symbols.

```
void _find_repeated(Expression expr);
```

```
Expression _rebuild(Expression expr);
```

The cse tree algorithm structure is as follows.

```
std::vector<std::vector<Expression>> tree_cse(std::vector<Expression> exprs, std::ve
for(std::vector<Expression>::iterator it = exprs.begin(); it != exprs.end(); ++it)
{
                _find_repeated(*it);
}
for(std::vector<Expression>::iterator it = exprs.begin(); it != exprs.end(); ++it)
{
                if(isinstance(e, Basic)){
        reduced_e = _rebuild(e);
}else{
            reduced_e = e;
}
      reduced_exprs.push_back(reduced_e);
}
vector<vector <Expression>> return_value=new vector<vector <Expression>>();
return_value.push_back(replacements);
return_value.push_back(reduced_exprs);
return return_value;
}
```

Improvements Possible

According to the issue #12411 of SymPy, the current performance of the SymPy algorithm for the common subexpression elimination is not efficient enough. According to the observations made by Andreas Klöckner there have been a drastic decline of performance in CSE after SymPy 1.0. Furthermore, there he suggests the improvement that can be made to SymPy CSE algorithm. He provides a sample of work done by Matt Wala in the Sumpy repository under merge request #23 where he has developed a similar algorithm to the SymPy one with many redundancies removed.

This will be studied further to improve the algorithm that is to be implemented in SymEngine. Furthermore, inclusion of elements for the better performance in regard to complex common subexpressions will be investigated in the process. This has to be further researched.

Tentative Timeline

The entire development process will be divided into several phases where the individual phases will be responsible for part of the functionality of the system. For the ease of understanding, the entire process will be divided into sections based on the modules in the SymEngine system and we will be using a test based development approach where the test cases are prepared before the coding is carried out.

Pre GSoC

I will mainly spend this time working on the community building activities, getting to know the people as well as working on the necessary preparations for the project. During this time more effort will be divulged to the study of the relevant algorithms in Common Subexpression Elimination. As the current set of algorithms may have drawbacks as the previous sections explore, I will conduct a thorough research the approaches and improvements to be made. This includes the improvements mentioned in implementation section. There will also be plans for other improvements. These will be decided based on the discussions with the mentors and other involved parties.

Week 1

Design the prototype cpp functions for the tree cse algorithm and the algorithms decided upon the discussions carried out and the necessary subroutines and create template for the test functions.

Week 2

Implement the generic tree cse equivalent function in C++ using the object constructs in SymEngine. During this period the work on the tests will also be started with the functional

prototype sections realised.

Week 3

Complete the tree cse equivalent function in C++ using the object constructs in SymEngine. During this period the work on the tests will also be started with the functional prototype sections realised.

Week 4

Design the necessary support functions for pre-processing of the expressions for cse and for the post-processing of the expressions after the cse process is carried out.

Week 5

Design the functions for the optimizing of the cse process using the optimizations in the application of add, mul, pow and etc. methods. This will be followed by a set of test cases to support the optimization activity.

Week 6

Completing the optimization functions and then carrying out the tests that can be carried out upto that point to check if the functionality is as expected.

Week 7

Design the process of the main cse function that incorporates the algorithm designed earlier and the optimization methods and support functions to calculate the common subexpression elimination and start working on the code.

Week 8

Start working on adding complex number support for LLVMDoubleVisitor. Start on developing the necessary tests to test the complex number support.

Week 9

Adding complex number support for LLVMDoubleVisitor. Developing and improving the necessary tests to test the complex number support.

Week 10

Adding complex number support for LLVMDoubleVisitor. Developing and improving the necessary tests to test the complex number support.

Week 11

Improving the c code generator and developing python wrappers for the c code generator.

Week 12

Week 12 will act as an overflow period which will be used to cover the overflows of the work we have encountered op to that point of time. Furthermore, this time will be used to prepare for the evaluation process.

Post GSoC

I will try to incorporate other algorithms that can be used in specific cases for the common subexpression elimination and check how the process would work and the possible improvements out of the algorithms implemented. Furthermore, the system performance will be checked for the efficiency against the implementation provided in SymPy.

References

[1]"sympy/sympy", GitHub, 2017. [Online]. Available: https://github.com/sympy/sympy/wiki/GSoC-2017-Ideas#symengine-projects.

[2]"sympy/sympy", GitHub, 2017. [Online]. Available: https://github.com/sympy/sympy/tree/master/sympy/simplify.

[3]"sympy/sympy", GitHub, 2017. [Online]. Available: https://github.com/sympy/sympy/wiki/Release-Notes-for-0.7.4

[4]"sympy/sympy", GitHub, 2017. [Online]. Available: https://github.com/sympy/sympy/blob/master/sympy/simplify/cse_main.py

[5]A. Hosangadi, F. Fallah and R. Kastner, "Optimizing Polynomial Expressions by Algebraic Factorization and Common Subexpression Elimination", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 25, no. 10, pp. 2012-2022, 2006

[6]Common Subexpression Elimination, 1st ed. .

[7]A. Hosangadi, Optimization techniques for arithmetic expressions, 1st ed. [Santa Barbara, Calif.]: University of California, Santa Barbara, 2006.

[8]"Faster CSE (!23) · Merge Requests · Andreas Klöckner / sumpy", GitLab, 2017. [Online]. Available: https://gitlab.tiker.net/inducer/sumpy/merge_requests/23.

[9]"Efficiency of common subexpression elimination · Issue #12411 · sympy/sympy", GitHub, 2017. [Online]. Available: https://github.com/sympy/sympy/issues/12411

[10]J. Cocke, "Global common subexpression elimination", ACM SIGPLAN Notices, vol. 5, no. 7, pp. 20-24, 1970.

[11]"LLVM's Analysis and Transform Passes", Releases.llvm.org, 2017. [Online]. Available: http://releases.llvm.org/2.6/docs/Passes.html#gcse

[12]J. Reif and H. Lewis, "Efficient symbolic analysis of programs", Journal of Computer and System Sciences, vol. 32, no. 3, pp. 280-314, 1986

[13]"sympy.simplify.cse_main — SymPy 1.0.1.dev documentation", Docs.sympy.org, 2017. [Online]. Available: http://docs.sympy.org/dev/_modules/sympy/simplify/cse_main.html

[14]J. Ullman, "Fast algorithms for the elimination of common subexpressions", Acta Informatica, vol. 2, no. 3, pp. 191-213, 1973

[15]"LLVM: EarlyCSE.cpp Source File", Llvm.org, 2017. [Online]. Available: http://llvm.org/docs/doxygen/html/EarlyCSE_8cpp_source.html

[16]"Simplify — SymPy 1.0 documentation", Docs.sympy.org, 2017. [Online]. Available: http://docs.sympy.org/latest/modules/simplify/simplify.html

[17]A New Technique for Algebraic Optimization of Arithmetic Expressions, 1st ed. .

[18]J. Ding, J. Chen and C. Chang, "A New Paradigm of Common Subexpression Elimination by Unification of Addition and Subtraction", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 35, no. 10, pp. 1605-1617, 2016.