Common Subexpression Elimination with Subtree Isomporphisms

Robert King(student)*, Milinda Fernando(mentor)†, Hari Sundar (mentor)‡,
*giskard.king@gmail.com, †milinda@cs.utah.edu, ‡hari@cs.utah.edu,

I. Introduction

The purpose of this work is to improve the run time of the simulation of black hole collisions and their corresponding gravitational waves. Black hole collisions can be modeled using the BSSN equations. The BSSN equations consist of several complex partial differential equations and to model these equations the value of each variable is computed once per a time step in the model. The overall model is constructed by calculating millions of timesteps to see how black holes interact with each other. Each timestep must solve the partial differential equations. This process is automated using the python package SymPy. SymPy takes mathematical expressions and generates python code to solve each expression. However due to the complexity of the BSSN differential equations, the auto generated code consists of thousands of temporary variables. Due to the number of temporary variables, modern compilers are unable to effectively optimize the code causing the code to become incredibly inefficient. This thesis illustrates a technique to use Subtree Isomorphisms and Common Subexpression Elimination to improve the run time. The focus of this work is to use a bottom up approach to find an efficient way to solve for the values of the partial differential equations for each timestep. The strategy is to convert all temporary variable computations into expression trees. Once the expression trees are created, they are rebuilt to take advantage of caching for the targeted memory architecture. In addition to outlining an algorithm to improve calculation, this work proves a lower bound for the registers necessary to calculate each temporary variable using graph partitioning.
The key contributions of this work are:

Automatic symbolic code-generation. Given the complexity of the Einstein equations, we have developed an automatic code generation framework for GR using `SymPy` that automatically generates architecture-optimized codes. This greatly improves code portability, use by domain scientists and the ability to add additional constraints and checks to validate the code.

II. Methodology

The Figure 3 on the poster presents an overview of our approach.
The research for this thesis consists of two main projects. The first is the subtree isomorphism problem that will focus on the common subexpression elimination and the second is a lower bound analysis for the number of temporary variables needed to solve the partial differential equations. The goal is to create an algorithm that will be able to analyze the different partial differential equations and reorder the temporary variable calculations to maximize cache effectiveness and variable reuse.

A. Staging

Staging is focused on finding variable reuse within the partial differential equations. The first problem is to create an expression tree from the partial differential equations generated from the SymPy auto generated code. Once the expression tree is created, subtree isomorphism analysis can begin. This will be a bottom up approach that considers the values used in each leaf node in addition to finding similar tree structure. Each node within the tree will keep track of all the leaf values that it depends on. Set similarity will be used as a precondition before calculating the more expensive tree isomorphism. By the end of the Staging Process the most common subexpressions will be identified. Each expression will be valued depending on the number of leaf node dependents, to mitigate the total number of temporary variables, and the number of times each expression appears, to maximize data reuse.

B. Rebuilding

The rebuilding phase takes the results from the staging process and rebuilds the expression tree. In order of importance the rebuilding phase must preserve the correct final answers, maximize cache effectiveness, and minimize the total number of temporary variables used. The current strategy is to use a dynamic programming approach. Dynamic programming will also allow for the rebuilding process to scale effectively. Once complete, these results will be measured experimentally. Once a proof of concept has been completed the rebuilding process will take in parameters such as memory architecture, and L1 cache size to optimize the calculations for each machine. Once the rebuilding process is completed a parser will be created to transform the original SymPy autogenerated code to reflect the updated expression tree.

III. Results

As a proof of concept, an initial algorithm was developed to parse the SymPy autogenerated code into an encompassing expression tree and reduce the tree using a user specified cache size. The greedy strategy aims to create expression that with the number of dependencies equivalent to the specified cache size. This process is applied recursively to calculate all the target variables from the SymPy autogenerated code.

The preliminary results demonstrate that lower cache sizes cause parts of expressions tree to be duplicated. This similarity suggests there is an opportunity to effectively utilize memory locality to reduce run time. At larger cache sizes it is important to stage the expression tree to reduce the number of cache misses and improve performance.

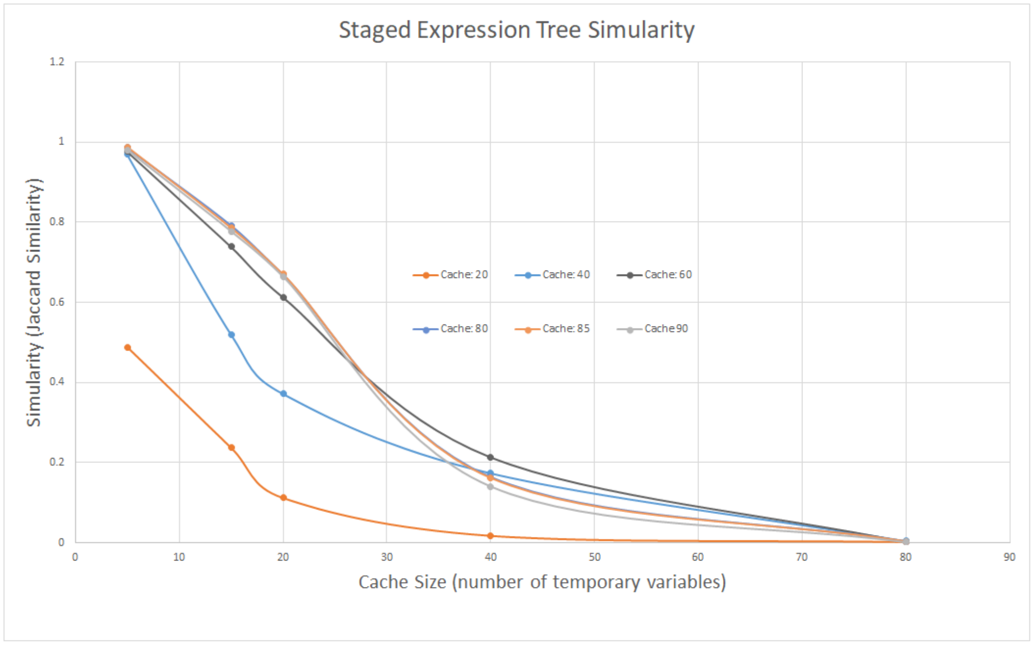

Fig. 1: Preliminary Results