Machine Specific Symbolic Code Generation

Robert King, Mentors: Hari Sundar, Milinda Fernando
University of Utah

Motivation

The purpose of this work is to improve the run time of the simulation of black hole collisions as seen in the Dendro-Gr framework. The BSSN equations and several other simulation codes 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 project presented aims to provide a method to increase cache utilization to increase runtime performance.

Contributions

- **Automatic 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 which helps to improve code portability.
- **Performance**. Developed a tree isomorphism algorithm for common sub expression elimination. The algorithm aims to improve performance by maximizing the efficiency of the cache. This is done by reducing expression trees into smaller sub trees such the each sub tree can be computed within the limits of the cache.
- **Visualization**. Code is able to represent expression trees with Graphiz and visualize expression tree staging
- **Flexibility** Code provides flexibility for any type of expression tree. Code can also be staged for any type of cache architecture. This allows easy adaptability between CPU and GPU codes architectures. By providing the target size of the cache the code is able to generate high performance code that can utilize the cache effectively. The results presented were applied to the `DENDRO_sym` project `https://arxiv.org/abs/1807.06128`.

Symbolic code generation for BSSNKO equations

$$
\begin{aligned}
\partial_t \alpha &= \mathcal{L}_\beta \alpha - 2\alpha K, \\
\partial_t \beta^i &= \lambda_2 \beta^j \partial_j \beta^i + \frac{3}{4} f(\alpha) B^i \\
\partial_t B^i &= \partial_t \tilde{\Gamma}^i - \eta B^i + \lambda_3 \beta^j \partial_j B^i - \lambda_4 \beta^j \partial_j \tilde{\Gamma}^i \\
\partial_t \tilde{\gamma}_{ij} &= \mathcal{L}_\beta \tilde{\gamma}_{ij} - 2\alpha \tilde{A}_{ij}, \\
\partial_t \chi &= \mathcal{L}_\beta \chi + \frac{2}{3}\chi\left(\alpha K - \partial_a \beta^a\right) \\
\partial_t \tilde{A}_{ij} &= \mathcal{L}_\beta \tilde{A}_{ij} + \chi\left(-D_i D_j \alpha + \alpha R_{ij}\right)^{TF} + \\
&\quad \alpha\left(K\tilde{A}_{ij} - 2\tilde{A}_{ik}\tilde{A}^k_j\right), \\
\partial_t K &= \beta^k \partial_k K - D^i D_i \alpha + \\
&\quad \alpha\left(\tilde{A}_{ij}\tilde{A}^{ij} + \frac{1}{3}K^2\right), \\
\partial_t \tilde{\Gamma}^i &= \tilde{\gamma}^{jk}\partial_j\partial_k\beta^i + \frac{1}{3}\tilde{\gamma}^{ij}\partial_j\partial_k\beta^k + \beta^j\partial_j\tilde{\Gamma}^i - \\
&\quad \tilde{\Gamma}^j\partial_j\beta^i + \frac{2}{3}\tilde{\Gamma}^i\partial_j\beta^j - 2\tilde{A}^{ij}\partial_j\alpha + \\
&\quad 2\alpha\left(\tilde{\Gamma}^i_{jk}\tilde{A}^{jk} - \frac{2}{3\chi}\tilde{A}^{ij}\partial_j\chi - \frac{2}{3}\tilde{\gamma}^{ij}\partial_j K\right)
\end{aligned}
$$

```
from DENDRO_sym import *
a_rhs = Dendro.Lie(b, a) - 2*a*K
b_rhs = [3/4 * f(a) * B[i] +
l2*vec_j_del_j(b, b[i]) for i in e_i]
l2*vec_j_del_j(b, b[i])
for i in e_i]
B_rhs = [Gt_rhs[i] - eta * B[i] +
l3 * vec_j_del_j(b, B[i]) -
l4 * vec_j_del_j(b, Gt[i])
for i in e_i]
gt_rhs =  Dendro.Lie(b, gt) - 2*a*At
chi_rhs = Dendro.Lie(b, chi) +
2/3*chi*(a*K - del_j(b))
At_rhs = Dendro.Lie(b, At) + chi *
Dendro.TF(-DiDj(a) +
a*Dendro.Ricci) +
a*(K*At -2*At_ikAtKj)
K_rhs = vec_k_del_k(K) - DIDi(a) +
a*(1/3*K*K + A_ij_A_IJ(At))
```

The left panel shows the BSSNKO formulation of the Einstein equations. These are tensor equations, with indices $i, j, \ldots$ taking the values $1, 2, 3$. On the right we show the `DENDRO_sym` code for these equations. `DENDRO_sym` uses `SymPy` and other tools to generate optimized C++ code to evaluate the equations. Note that $\mathcal{L}_\beta$, D, ∂ denote Lie derivative, covariant derivative and partial derivative respectively, and we have excluded $\partial_t \Gamma^i$ from `DENDRO_sym` to save space.

Methods

This research consists of two main projects. The first is the sub tree 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.

Staging

The expression DAG contains the desired target variables that solve the Einstein equations, the sources, their corresponding dependencies, the internal nodes, which are calculate from derivatives and constants, the sinks. The targets have a significant number of dependencies, the BSSN equation have dependencies on the order of 100s, such that the cache misses occur while calculating the target at each grid point in the mesh. This approach mitigates cache misses by reducing the original expression DAG into smaller sub graphs such that the number of dependencies does not exceed the cache size of the specified machine. In order to maintain correctness some of the dependencies of the original expression graph must be duplicated into multiple sub graphs. Despite computing the expression tree multiple times, the goal is to reduce runtime through increased cache efficiency.

Rebuilding

Once the sub graph expressions are created the goal is to order the evaluation of the sub graphs to maintain correctness and maximize cache locality. Expression sub graphs are that are a dependency of another sub graph must be computed first. If several sub graphs have no dependencies, then the sub graphs are order such that graphs with the largest Jaccard Similarity are computed one after another. In doing so variables within the cache can increase usability.

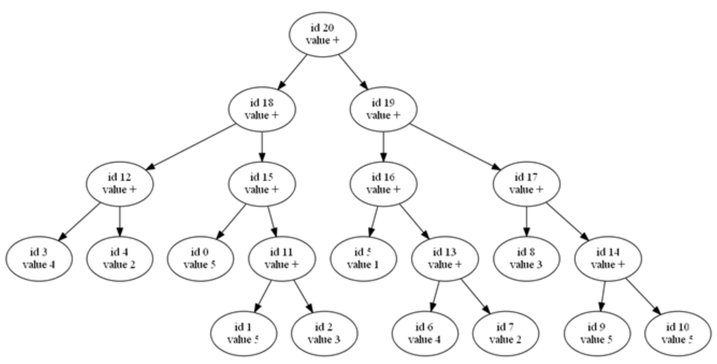

Figure 1: Initial Expression Tree

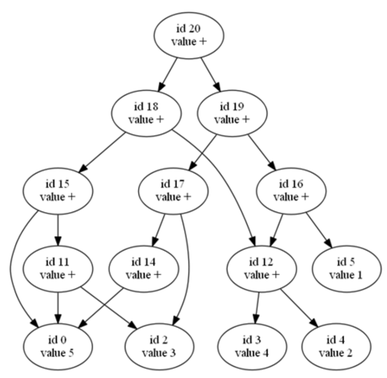

Figure 2: Staged Expression Tree

Results

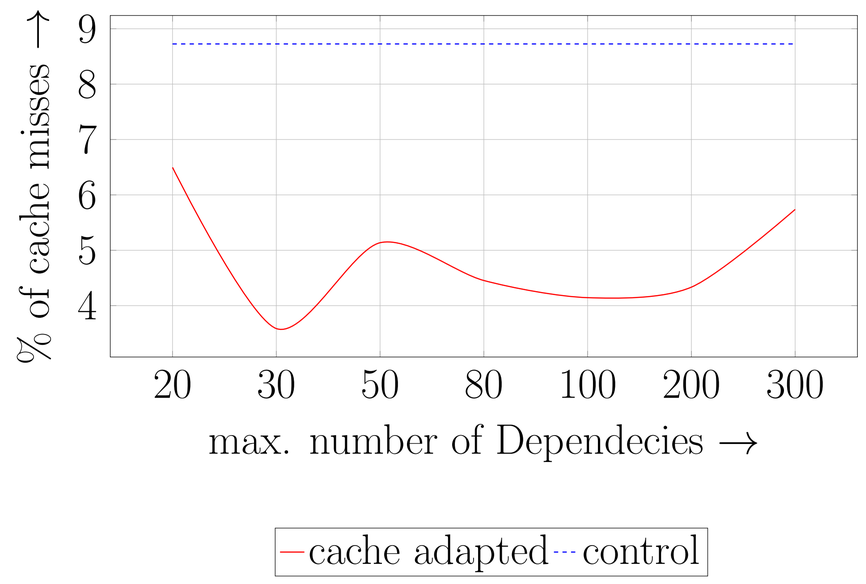

Figure 4: Number of cache misses based on cache requests for cache adapted code (staged) code vs. the control (unstaged) version of the code in TACC's Frontera supercomputer node

The main area of interest is reducing cache misses when computing the expression trees. Figure 4 demonstrates how the staging size affects the cache efficiency. There is a sweetspot where the staging size is large enough to leverage the entire cache while small enough to not overflow. The goal is adapt the expression trees to sit in the middle of the cache size sweetspot to improve performance. The results presented were run on the Kingspeak Cluster at the University of Utah. The machine consist of Intel Sandybridge processor with 64 KB per core L1 cache, 256 KB per core L2 cache and 20 MB shared L3 cache. On the Kingspeak chpc machines the performance was most effective with cache size of 30. Notice how all staged versions of the code were more cache efficient than the original code.

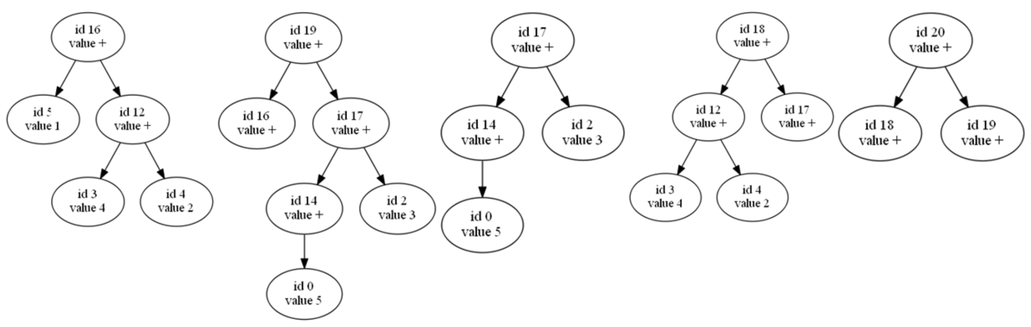

Figure 3: Rebuilt Expression Tree