Machine Specific Symbolic Code Generation for Computational Science Applications

Robert King(student)*, Milinda Fernando(advisor)†, Hari Sundar(advisor) ‡
*u1001542@utah.edu, †milinda@cs.utah.edu, ‡hari@cs.utah.edu,

I. Introduction

New discoveries in science and engineering are primarily driven by numerical simulations of underlying governing equations specially when the physical experiments become infeasible. High performance computing (HPC) is widely used to perform these simulations efficiently. When moving towards exascale computing, HPC clusters are moving towards increased heterogeneity and frequent change in architectures. Ability to utilize modern and future HPC clusters effectively is highly depend on performance portability and adaptation to new architectures. Manually written codes to evaluate the main computational kernels lack portability, prone to human errors, ability to perform code optimizations due to the complexity of the underlying equations. In this work we present a symbolic code generation framework, which generates architecture optimized code for different platforms. As the driving application we primarily use computational relativity where computations of Einstein equations become complicated due to the presence of curvature in spacetime. But the algorithms presented in this work, is applicable to generate code for any underlying applications.
The key contributions of this work include:

- **Symbolic inferace** : The presented framework is based on `SymPy` with additional modules written to handle complicated partial differential equations (PDEs).
- **Equations → Graphs** : The symbolically written equations are converted to a computational graph, which enables to perform architecture (cache, register optimizations) and language specific (SIMD vectorization, CUDA) optimizations.
- **Common Subexpression Elimination (CSE)**: By computing common subexpressions, we can reduce the number of compute operations needed, by storing them in temory variables.

II. Background & Related Work

In this section, we present a brief introduction on the driving application, where the symbolic code generation is deployed. The recent discovery of gravitational waves (GWs) in 2015, has excited the computational relativity community. Numerically computed GWs are important to perform verification and matched filtering for the massive amount of data generated by GW detectors. Computational relativity is primaraly focused on evolving 3+1 (space + time) decomposition of 4 Einstein equations for a specified initial condition. In this work we

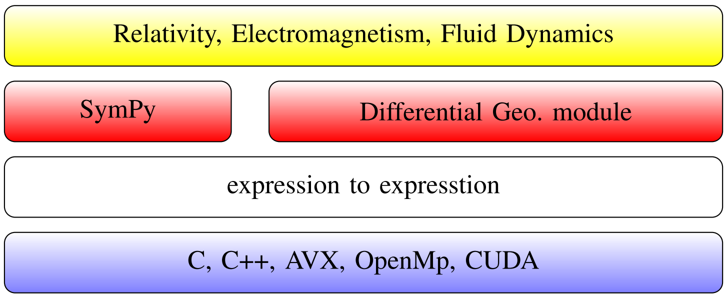
Relativity, Electromagnetism, Fluid Dynamics
SymPy
Differential Geo. module
expression to expresstion
C, C++, AVX, OpenMp, CUDA

Fig. 1: Project Architecture

use commonly used , formulation of Einstein equations.The Einstein equations are a set of non-linear, coupled, partial differential equations. On discretization, one can end up with 24 or more equations with thousands of terms. Writing, optimizing and maintaining code for this is very challenging. Sustainability and keeping it relevant for new architectural changes are additional difficulties. To address these issues, use our symbolic code generation framework, to generate architecture optimized codes to compute the BSSN equations (see Figure 2).

III. Methodology

The symbolic code generation framework is only the initial step towards connecting symbolic equations to architecture specific compute codes (see Figure 1). By representing the equations as directed acyclic graphs (DAG) enable to exploit computation to match the programming language (AVX, CUDA, OpenMp) and architecture (CPU, GPU) specifications which result in faster and efficient codes. These specifications are met by performing *expression to expression* transformations such that transformations are mathematically equivalent.

A. Staging Computations

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 derivates and constants, the sinks. The targets have a significant number of dependecies, the BSSN equation have dependecies 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

$$
\begin{aligned}
\partial_t \alpha &= \mathcal{L}_\beta \alpha - 2\alpha K, \\
\partial_t \beta^i &= \lambda_2 \beta^j \, \partial_j \beta^i + \frac{3}{4} f(\alpha) B^i \\
\partial_t B^i &= \partial_t \tilde{\Gamma}^i - \eta B^i + \lambda_3 \beta^j \, \partial_j B^i - \lambda_4 \beta^j \, \partial_j \tilde{\Gamma}^i \\
\partial_t \tilde{\gamma}_{ij} &= \mathcal{L}_\beta \tilde{\gamma}_{ij} - 2\alpha \tilde{A}_{ij}, \\
\partial_t \chi &= \mathcal{L}_\beta \chi + \frac{2}{3} \chi \left(\alpha K - \partial_a \beta^a\right) \\
\partial_t \tilde{A}_{ij} &= \mathcal{L}_\beta \tilde{A}_{ij} + \chi \left(-D_i D_j \alpha + \alpha R_{ij}\right)^{TF} + \\
&\quad \alpha \left(K \tilde{A}_{ij} - 2 \tilde{A}_{ik} \tilde{A}^k_j\right), \\
\partial_t K &= \beta^k \partial_k K - D^i D_i \alpha + \\
&\quad \alpha \left(\tilde{A}_{ij} \tilde{A}^{ij} + \frac{1}{3} K^2\right), \\
\partial_t \tilde{\Gamma}^i &= \tilde{\gamma}^{jk} \partial_j \partial_k \beta^i + \frac{1}{3} \tilde{\gamma}^{ij} \partial_j \partial_k \beta^k + \beta^j \partial_j \tilde{\Gamma}^i - \\
&\quad \tilde{\Gamma}^j \partial_j \beta^i + \frac{2}{3} \tilde{\Gamma}^i \partial_j \beta^j - 2 \tilde{A}^{ij} \partial_j \alpha + \\
&\quad 2\alpha \left(\tilde{\Gamma}^i{}_{jk} \tilde{A}^{jk} + 6 \tilde{A}^{ij} \partial_j \phi - \frac{2}{3} \tilde{\gamma}^{ij} \partial_j K\right)
\end{aligned}
$$

```
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]

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))
```

Fig. 2: 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. (See [?], [?] for more information about the equations and the differential operators.)

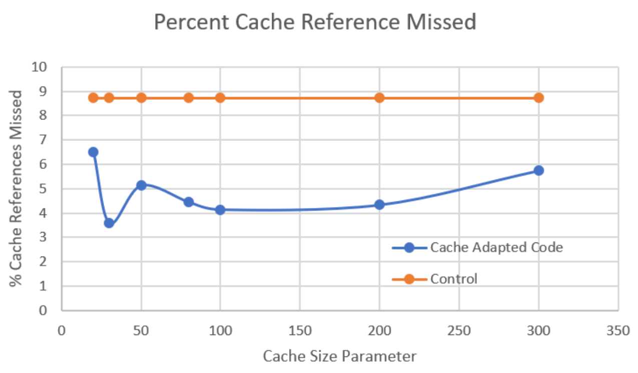

Fig. 3: Cache Utilization of Staged Code.

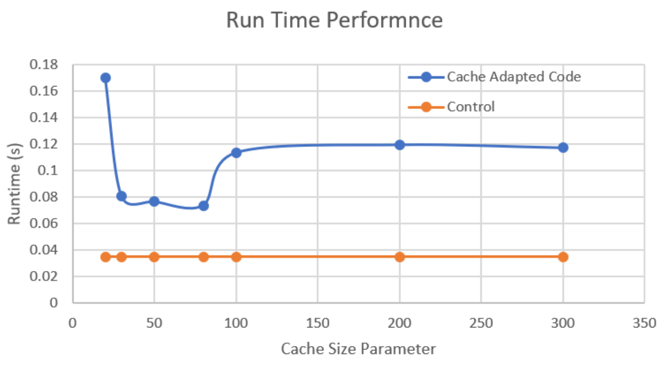

Fig. 4: Runtime Analysis of Staged Code.

of the dependencies of the original expression graph must be duplicated into multiple subgraphs. Despite computing the expression tree multiple times, the goal is to reduce runtime through increased cache efficiency.

B. Subtree Isomorphism

Once the subgraph expressions are created the goal is to order the evaluation of the sub graphs to maintain correctness and maximize cache locality. Expression subgraphs are that are a dependency of another subgraph must be computed first. If several subgraphs have no dependencies, then the subgraphs 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.

IV. RESULTS

The two areas of interest are reducing cache misses and the overall runtime of the expression graphs. Figure 3 demonstrates how the staging size affects the cache efficiency. There is a sweet spot where the staging size is large enough to leverage the entire cache while small enough to not overflow. The results presented were run on the Kingspeak Cluster at the University of Utah. The machine consits of Intel Sandy Bridge processors with 64 KB per core L1 cache, 256 KB per core L2 cache and 20 MB shared L3 cache. On the Kingspeak 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 4 shows the runtime analysis of the code. The runtimes were most effective when the cache utilization was the highest.

However, none of the stage versions of the code were able to outperform the original code. It is hypothesized that some of the dependencies are being computed a multiple time and causing the staged code to decrease. This is a current of area of interest of improvement.

V. Future Work

As mentioned in the results, the original code is faster than the staged code despite the cache improvement. Within each expression subgraph, some dependencies are calculated multiple times. To reduce this the subgraphs can be staged again such that the dependencies with the largest indegree are stored in local variables for each pass.

Another area of interest is to test the autogenerated code on GPUs. GPUs have a smaller shared memory. The goal is to decrease the staging size for the code and verify that the same cache utilization is seen and determine if runtime performance increases.