Redundancy-Avoiding Fusion for Tensor Algebra

Tong Zhou[1,2], Ruiqin Tian[1], Gokcen Kestor[1], Roberto Gioiosa[1], Vivek Sarkar[2]

[1] Pacific Northwest National Laboratory, [2] Georgia Institute of Technology

Problem Statement

Input: Dense or sparse tensor expressions in index notation, such as $A_{ij} = B_{ik} * C_{kj}$. Each dimension of a tensor can be specified as sparse or dense.

Output: Efficient C code that corresponds to the semantic of the expression.

Challenge: Always generating maximumly fused code or totally unfused code can both introduce redundancies. Sparse expressions make fusion decision more complicated.

Solution: We propose a fusion algorithm that fuses to the right amount without introducing redundancies.

Redundancy Category

We identify the following types of redundancies with maximum fusion or no fusion.

Type 1 (reduction redundancy): When multiplication happens together with a summation due to maximum fusion.

Type 2 (loop invariant redundancy): When a loop invariant expression is introduced (could be invariant to a middle loop) due to maximum fusion.

Type 3 (dead code redundancy): When some values are computed but not used (multiplying with 0s of some sparse tensor) due to no fusion.

Type 4 (load store redundancy): When some values are computed and require memory loads later when it's used due to no fusion.

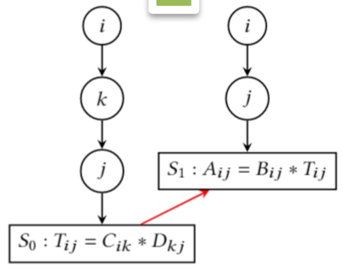
Approach and Example

Transformation steps:

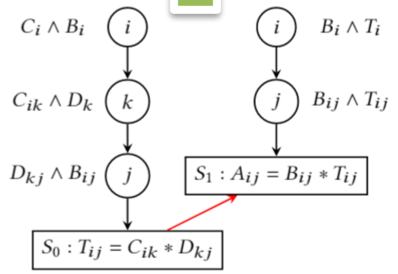
1. Convert tensor expressions to a loop nest tree with dependence edges
2. Compute iteration domains for each index node and each tensor
3. Perform maximal outer loop fusion of the trees
4. Intermediate tensor dimension reduction
5. Code generation

$$A_{ij} = B_{ij} * (C_{ik} * D_{kj})$$

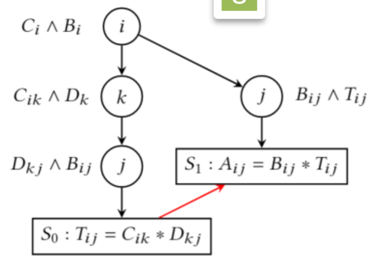
1


i
k
j
$S_0 : T_{ij} = C_{ik} * D_{kj}$
i
j
$S_1 : A_{ij} = B_{ij} * T_{ij}$

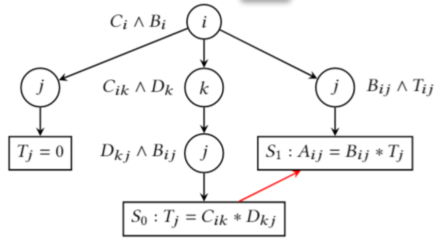
2



3



4



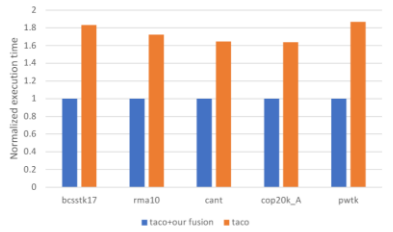
5

```
double* T = new double[NJ]();
int jT = 0;
for (i = 0; i < NI; ++i) {
  for (k = 0; k < NK; ++k) {
    jT = 0;
    for (j = B.rowptrs[i]; j < B.rowptrs[i+1]; j++) {
      T[jT] += C[i,k] * D[k, B.cols[ji]];
      jT++;
    }
  }
  jT = 0;
  for (j = B.rowptrs[i]; j < B.rowptrs[i+1]; j++) {
    A.vals[j] += B.vals[j] * T[jT];
    T[jT] = 0;
    jT++;
  }
}
```

Implementation is work in progress on top of the COMET[2] compiler based on MLIR.

Preliminary Results for SDDMM

- Machine: 2.5 GHz AMD 32-core (results are single-core for now)
- Benchmarks: real-world sparse matrices from SuiteSparse[3]



Incorporating our fusion scheme to state-of-the-art (TACO[1]) can bring **1.7x** speedup.

Conclusion and Future Work

We show four common types of redundancies that can be introduced, and a fusion algorithm that eliminates all of them for both single and multiple expressions. Empirical evaluation on real-world compound kernels (such as SDDMM) shows that our fusion algorithm and associated redundancy elimination can bring 1.4x to 40x speedup over state-of-the-art tensor algebra compiler (TACO) and > 100x speedup over library approaches such as Eigen. Future work includes parallel code generation, interaction with tiling etc,

Major References

1. Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and SamanAmarasinghe. 2017. The Tensor Algebra Compiler. OOPLSA'17.
2. Ruiqin Tian, Luanzheng Guo, Jiajia Li, Bin Ren, and Gokcen Kestor. 2021.A High-Performance Sparse Tensor Algebra Compiler in Multi-Level IR. . The 7th Annual Workshop on the LLVM Compiler Infrastructure in HPC, November 2021.
3. Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. ACM Trans. Math. Softw. 38, 1, Article 1 (Dec. 2011).