General description of the automated loop fusion algorithm

Krzysztof A. Drewniak

April 25, 2018

For simplicity, we'll start off by presenting this algorithm for loops over one matrix without additional inputs. What we want to do is to find algorithms for the sequence of operations $A^1 := \mathcal{F}^0(A^0); A^2 := \mathcal{F}^1 A^1; A^N := \mathcal{F}^{N-1}(A^{N-1})$. (All of these $\mathcal{F}^i$ owerwrite these inputs - there is one matrix A, which contains different values over time.) More specifically, we want to find (assuming they exist) fused algorithms for these operations, that is, we want an $\mathcal{F}'$ such that $A^n := \mathcal{F}'(A^0)$, with A^0 only being looped over/overwritten once.

The algorithms we're looking for are expressible in FLAME notation. That is, in order to perform our search, we're going to partition our matrix (or matrices) and into regions A_R^i. The typical partitions look like:

$$\left(\begin{array}{c|c} A_{TL}^{i+1} := \mathcal{F}_{TL}^i(A^i) & A_{TR}^{i+1} := \mathcal{F}_{TR}^i(A^i) \\ \hline A_{BL}^{i+1} := \mathcal{F}_{BL}^i(A^i) & A_{BR}^{i+1} := \mathcal{F}_{BR}^i(A^i) \end{array}\right).$$

This type of partitioned matrix expression (PME) can give rise to multiple algorithms. This is because $\mathcal{F}_R^i$ consists of "tasks", each of which can either be included in a loop invariant (which determines what updates the algorithm needs to perform) or in the remainder of that invariant (which shows what computation the algorithm needs to perform).

The general form of a FLAME algorithm is to loop over A, expanding some region(s) while shrinking others, until the whole matrix is one region that contains the final result. The loop needs to satisfy the loop invariant at each iteration.

We will write a task t that updates the region A_R as $\{(R_1, s_1), (R_2, s_2) \vee (R_3, s_3), \ldots (R_k, s_k)\} \to \{(R, t)\}$. The R_i are regions of A (or, more generally, regions of any object in the problem), while the s_i and t are specifiers of the state of the region.

These state specifiers can either be $\perp$, representing an unmodified input, $\top$, a fully computed output, a natural number d, representing intermediate states, or a pair (d, p), where d is a number and p is an arbitrary symbol, which is used to handle computations that can happen in arbitrary order.

We also allow ors ($\vee$) as task inputs, to represent cases where the input to a task could be in several possible states. For example, when performing a matrix multiply, the matrix that the result is added to can either be the input or the result of another portion of the multiplication, and we don't want to restrict which order the two parts occur in.

A partitioned matrix expression then consists of a set of regions R, each of which has an associated set of tasks T_R which update it. Finding a loop invariant means finding pasts

P_R and futures F_R for each region such that $T_R = P_R \cup F_R$, the two sets are disjoint, and certain other constraints are satisfied.

The first constraint is that some region R must have a task that represents the operation being performed by the algorithm in ins past, while a different region R' must have such a task in its future. That means that the process of moving data from R' to R will compute the operation we're trying to generate an algorithm for.

The second constraint is that dependencies between the past and future are satisfiable. That is, a task (R_i, s_i) may not be in the past of any region if the state (R_i, s_i) will only be readable after work that occurs in the future. Similarly, tasks in the future may not read from states that have been overwritten.

To ensure dependencies are satisfied, we define an ordering $<_T$ between tasks. First, $(R, s) <_T (R', s')$ and $(R', s') <_T (R, s)$ if R' and Rr are different regions. If $R' = R$, then we defer to the following ordering on states: $\bot <_S d <_S \top$ for all integers d (and pairs (d, p)). Also, for integers d and e, $d <_S e$ if $d < e$ (similarly for pairs (d, p) and (e, p')). Finally, $(d, p) <_S (d, p')$ if $p \neq p'$.

If we want to compare an or $(R_1, s_1) \vee (R_2, s_2) \ldots \vee (R_k, s_k)$, we simply need to find one of the (R_i, s_i) that makes the comparison true. That is, dependencies are satisfied for an or if there's a branch of the or that makes the loop an invariant.

These orderings allow us to characterize the constraint for satisfied dependencies as requiring all inputs to tasks in the past to be $<_O$ the outputs of all future tasks, and requiring the outputs of all past tasks to be $\leq_O$ the inputs of all tasks in the future.

There's also a constraint meant to reduce "duplicate" results that arise from task of the form $\{(R, \top) \rightarrow (R'\top)\}$ which are tagged as noops, which are needed to maintain the invariant that all tasks which update R are part of region R's task set. For example, we need such a noop tasks to represent parts of the LU factorization $A \rightarrow LU$. With these tasks, we ensure the noop cannot be in the future if the source operand $(R, \top)$ has been computed.

Returning to the loop fusion question, we know what, in the case of unfused algorithms, the operand $(R^{i+1}, \bot)$ is the exact same data as $(R^i, \top)$. In the fused case, any algorithm could leave each region in a computed ($\top$) state, an uncomputed ($\bot$) state, or a partially computed state. Since the details of exactly how a given region was partially computed are irrelevant to further discussion, the numbered intermediate states will be denoted as $\vdash$.

When finding fused algorithms, it is useful to consider the problem as a collection of *strips* instead of as a series of PMEs. The strip S_R for a region R is the sequence of tasks $[T_R^0, T_R^1, \ldots, T_R^{N-1}]$. This effectively corresponds to "transposing" the input PMEs.

The theory of loop fusion gives us two constraints on how the tasks in successive loops can relate to each other. First, the ith loop cannot read from the region R in the past unless it was fully computed by region $i - 1$ (or it is the first loop). Second, the region R cannot be read in the future unless, for all $j > i$, the jth loop does not update R, that is, the jth loop leaves R uncomputed.

One consequence of these theorems is that, within a strip, the task sets must never have outputs (the most-computed state in the past) that are more computed than the outputs of previous regions, and that there must be at most one partially-computed region per strip.

This insight allows us to begin our search by considering all the possible assignments of $\top$, $\vdash$, and $\bot$ to task sets in a strip that make the strip have the form $\top^* \vdash +?\bot^*$.

Simply performing this search for each region will generate many possible past-and-future

splits that, even if they were loop invariants for each PME, would not be fusable. To ensure fusability, we need to perform a global check for it. The only data needed for this check is that, for each strip, we need to compute the number of the last region that is fully computed C_R (if there is no such region, L_R is -1), and the first uncomptuted region U_R (if there is none such, U_R is N).

Then, if R' is an input to the past of the ith loop, we need $C_R \geq i - 1$. If it as input to the future of that loop, we need $C_R \leq i + 1$. These constraints reflect the fusion theorems from above.

So, our algorithm proceeds to search through all the valid strips (considering all possible partially-computed regions for each $\vdash$ task set). Then, it confirms that the past-future splits for each task set satisfy the fusion constraints. For efficiency, these checks are interleaved, allowing certain candidate strips to be rejected because they would violate constraints on C_R or U_R that arise from strips that have already been fixed. Once splits are found that satisfy the fusion constraints, they are checked to confirm that each loop within them satisfies the constraints on being a loop invariant.

One issue arises when there is more than one matrix or vector being updated by the operations, and not every operation updates each object. For our algorithm to operate correctly, we need each strip to be the same length. Ensuring this is the case sometimes requires adding empty task sets, which have the form $\{\} \to \{\}$. When there are empty task sets, we allow the values of C_R and U_R to live in a range between their "true" value (what it would be if the empty sets were computations) and the value that assumes the end of the surrounding empty task sets was also computed/uncomputed. We can do this since C_R and U_R are already variables in a integer constraint programming system, which allows us to assign ranges to them just like concrete values.

This expansion is needed to avoid rejecting valid fusable loop invariants by propagating the computation/non-computation of a region through the series of loops in which it isn't used while also ensuring that constraints that come from earlier loops are satisfied.