The document where I explain my algorithm

Krzysztof A. Drewniak

June 26, 2018

1 Background

1.1 FLAME

Our work is based on the FLAME [1, 2] methodology for systematically (and, from this, automatically) deriving algorithms for operations on (dense) matrices or other objects (such as graphs**TODO cite a paper here**) with a similar structure. The FLAME methodology is effective for operations $\mathcal{F}(\hat{A}, O)$ where an object A has its initial state $\hat{A}$ overwritten incrementally throughout the operation to produce its final state $\widetilde{A} = \mathcal{F}(\hat{A}, O)$ at the algorithm's termination. The computations in the algorithm may also depend on a set of read-only operands O. It should be noted that $\mathcal{F}$ does not necessarily depend on $\hat{A}$, such as in the case of matrix-vector multiply, where we have $\widetilde{y} = Ax$ for some read-only A and x.

Throughout this paper, we may omit the set of read-only operands if they clutter the presentation.

In the FLAME approach, the object being operated on is divided into regions, and the algorithm progresses by "moving" values between regions by performing computations. An example of this algorithm structure can be found in Figure 1. Each region (and therefore the whole of A) is constrained by a loop invariant, which must hold at the beginning of the algorithm and after each iteration of the algorithm's loop. At the beginning of the algorithm, all of A is assigned to some region P, which must have a loop invariant that causes A_P to be equal to $\hat{A}$ at that time. This leaves all of the other regions empty. During the algorithm, values are moved from A_P to other regions, with the goal of eventually putting them all in a region Q such that $A_Q = \widetilde{A}$ at the end of the computation. The algorithm is determined by its loop invariant, as the body of the loop is created by identifying the computations needed to allow the sizes of regions to change while respecting the loop invariant.

Loop invariants (and therefore algorithm) for some $\mathcal{F}$ can also be found systematically. This process begins by forming the *partitioned matrix expression* (PME) for $\mathcal{F}$. This consists of partitioning A (and therefore $\hat{A}$ and $\widetilde{A}$) into a series of regions A_R and finding the $\mathcal{F}_R$ that correspond to the operations needed to compute each region (which may involve some algebra). This process may also require some of the read-only operands to be partitioned, though those partitionings need not be the same as the one used for A.

As an example, if A is a general matrix, we could partition it using a 2×2 grid of regions

partition $x \rightarrow \left(\begin{array}{c} x_T \\ \hline\hline x_B \end{array}\right)$ where length$(x_T) = 0$

do until length$(x_T) = n$

repartition $\left(\begin{array}{c} x_T \\ \hline\hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline\hline \chi_1 \\ \hline x_2 \end{array}\right)$

⋮] loop body

continue with $\left(\begin{array}{c} x_T \\ \hline\hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ \hline\hline x_2 \end{array}\right)$

enddo

Figure 1: The structure of an algorithms produced by the FLAME method. This example shows the skeleton of an algoritm with a 2×1 partitioning that moves from top to bottom.

to form the following PME

$$\left(\begin{array}{c|c} \widetilde{A}_{TL} = \mathcal{F}_{TL}(\hat{A}) & \widetilde{A}_{TR} = \mathcal{F}_{TR}(\hat{A}) \\ \hline\hline \widetilde{A}_{BL} = \mathcal{F}_{BL}(\hat{A}) & \widetilde{A}_{BR} = \mathcal{F}_{BR}(\hat{A}) \end{array}\right),$$

From the PME, we can find potential loop invariants by partitioning each $\mathcal{F}_R$ into a (potential) loop invariant f_R and remainder f'_R. These are two functions, which may be the identity, such that $\mathcal{F}_R(\hat{A}_R) = f'_R(f_R(\hat{A}_R))$. In this formalism, the remainder represents computations that have not yet been performed at some particular iteration of the loop.

Not all collections of such function partitionings form a loop invariant. The first condition on these partitionings, mentioned above in different terms, is that there must be distinct regions P and Q such that P has the operation that $\mathcal{F}$ performs in its remainder and Q has that operation in the invariant. This ensures that the algorithm can make progress by changing region sizes to shrink P and expand Q.

The second condition is that loop invariants and remainders must respect data dependencies. That is, no f_R can read from a memory state that has not yet been computed, not can an f'_R read from a state that has been overwritten by previous computations. If both of these constraints are satisfied, the collection of partitionings becomes a loop invariant for $\mathcal{F}$.

As a concrete example, we can consider the Cholesky factorization $CHOL(\hat{A})$, which, given a symmetric matrix $\hat{A}$, produces a lower (or upper) triangular matrix $\widetilde{A}$ such that $\widetilde{A}\widetilde{A}^T = \hat{A}$. If we partition A in the specification, we can derive the PME (with $*$ representing

data that is not stored in memory)

$$\left(\begin{array}{c|c} \widetilde{A}_{TL} & 0 \\ \hline \widetilde{A}_{BL} & \widetilde{A}_{BR} \end{array}\right)\left(\begin{array}{c|c} \widetilde{A}_{TL}^T & \widetilde{A}_{BL}^T \\ \hline 0 & \widetilde{A}_{BR}^T \end{array}\right) = \left(\begin{array}{c|c} \hat{A}_{TL} & * \\ \hline \hat{A}_{BL} & \hat{A}_{BR} \end{array}\right)$$

$$\left(\begin{array}{c|c} \widetilde{A}_{TL}\widetilde{A}_{TL}^T = \hat{A}_{TL} & * \\ \hline \widetilde{A}_{BL}\widetilde{A}_{TL}^T = \hat{A}_{BL} & \widetilde{A}_{BL}\widetilde{A}_{BL}^T + \widetilde{A}_{BR}\widetilde{A}_{BR}^T = \hat{A}_{BR} \end{array}\right)$$

$$\left(\begin{array}{c|c} \widetilde{A}_{TL} = CHOL(\hat{A}_{TL}) & * \\ \hline \widetilde{A}_{BL} = \hat{A}_{BL}\widetilde{A}_{TL}^{-T} & \widetilde{A}_{BR} = CHOL(\hat{A}_{BR} - \widetilde{A}_{BL}\widetilde{A}_{BL}^T) \end{array}\right)$$

From this PME, we can find the following three loop invariants:

$$\left(\begin{array}{c|c} A_{TL} = CHOL(\hat{A}_{TL}) & * \\ \hline A_{BL} = \hat{A}_{BL} & A_{BR} = \hat{A}_{BR} \end{array}\right) \qquad \left(\begin{array}{c|c} A_{TL} = CHOL(\hat{A}_{TL}) & * \\ \hline A_{BL} = \hat{A}_{BL}\widetilde{A}_{TL}^{-T} & A_{BR} = \hat{A}_{BR} \end{array}\right)$$

$$\left(\begin{array}{c|c} A_{TL} = CHOL(\hat{A}_{TL}) & * \\ \hline A_{BL} = \hat{A}_{BL}\widetilde{A}_{TL}^{-T} & A_{BR} = \hat{A}_{BR} - \widetilde{A}_{BL}\widetilde{A}_{BL}^T \end{array}\right)$$

In the algorithms that arise from all of these loop invariants, the entire matrix begins in A_{BR} and moves to A_{TL} as computations are performed. It is worth noting that the third loop invariant is still valid since A_{BR} is equal to $\hat{A}_{BR}$ initially, because the A_{BL} region is empty, rendering the subtraction irrelevant at that time.

Dependency analysis is important, as

$$\left(\begin{array}{c|c} A_{TL} = CHOL(\hat{A}_{TL}) & * \\ \hline A_{BL} = \hat{A}_{BL} & A_{BR} = \hat{A}_{BR} - \widetilde{A}_{BL}\widetilde{A}_{BL}^T \end{array}\right)$$

is not a valid loop invariant for the Cholesky factorization because the computation of A_{BR} requires A_{BL}'s fully computed value to be available, even though it has not yet been written to memory.

1.2 Loop fusion

If we have a series of n operations $\widetilde{A}^0 = \mathcal{F}^0(\hat{A}_0); \widetilde{A}^1 = \mathcal{F}^1(\hat{A}^1); \ldots \widetilde{A}^{n-1} = \mathcal{F}(\hat{A}^{n-1})$ over the same object A (that is, for all i, $\hat{A}^i = \widetilde{A}^{i-1}$), *loop fusion* is the process of finding an algorithm for $\widetilde{A}^{n-1} = \mathcal{F}(\hat{A}^0)$ that only iterates through A once.

Such loop fusion is achieved by taking the loop bodies from an algorithm for each $\mathcal{F}^i$ and concatentating them to create one fused loop. However, this operation only produces a correct algorithm for $\mathcal{F}$ when the loop invariants for each loop being fused satisfy certain additional conditions, which were first set out in [2].

For the purposes of these conditions, a region $R4$ is *fully computed* in the kth loop if $f_R^{k'}$ is the identity, and it is *uncomputed* if f_R^k is the identity.

The first condition is that, if the $i+1$st loop invariant depends on the values in a region R (in a way that is not simply $A_R = \hat{A}_R$), then R must be fully computed by the ith loop

invariant, or else the assumption that $\hat{A}_R^{i+1} = \widetilde{A}_R^i$ that f^{i+1} relies on for correctness would be violated. Similarly, if the ith loop invariant depends on A_R to compute its remainder, then, for all $j > i$, R must be uncomputed by the jth invariant, or else the work that the ith loop will perform in the future will use incorrect values because A_R no longer maintains the expected state.

Finding loop invariants by hand is a process that can quickly grow tedious. For example, finding an fused algorithm for the inversion of a symmetric matrix (when computed as $A \coloneqq CHOL(A); A \coloneqq A^{-1}; A \coloneqq A^T A$) requires evaluating 72 combinations of loop invariants to find the single fused algorithm. Therefore, we have developed a tool that will automatically perform this search.

2 Algorithm

2.1 Task-based representation of PMEs

Before we give our algorithm, we need to discuss the structure of the inputs to our search procedure. Our inputs are neither the mathematical expressions for the operations we wish to fuse or their PMEs as typically written. Working at these levels would significantly reduce the generality of our system, as the process of generating a PME or determining what (if any) ways an expression can be partially computed requires domain knowledge. For example, it is not clear from the syntax that $\widetilde{A}_{BL} = -\hat{A}_{BL}\widetilde{A}_{TL}$ consists of one operation, not two, or that there ate two ways to partially compute $A - B - C$, not one. In addition, both PME creation and determining of how an expression could be partially computed are analyses that are not difficult to perform by hand and that have been automated.

Therefore, we require each region in the input PME to be split into one or more *tasks*, which represent the operations that make up all the computations in that region. This split format allows the calling user or program to explicitly indicate what form a partial update of a region can take. For example, the top left of the PME for the Cholesky factorization is simply input as $\widetilde{A}_{TL} \coloneqq_O CHOL(\hat{A}_{TL})$ (since it has no meaningful subdivisions), while the bottom-right quadrant can be written as $A_{BR,0} = \hat{A}_{BR} - \widetilde{A}_{BL}\widetilde{A}_{BL}^T$ and $\widetilde{A}_{BR} \coloneqq_O CHOL(A_{BR,0})$.

This example shows a few features of the task-based notation for PMEs. The first is that tasks are written using $\coloneqq$ to distinguish them from the equations in a PME or loop invariant. The second is that we need to explicitly mark *operation tasks*, which compute the function that the algorithm is meant to implement with $\coloneqq_O$. This type of marking is needed so that the search procedure can filter out loops that would not make progress on the underlying operation, since our method does not make any assumptions about the underlying domain.

The final important feature of the task notation is that we have introduced an explicit notation for the intermediate states a memory region R can have between its initial value $\hat{A}_R$ and final value $\widetilde{A}_R$. Specifically, we allow referring to partially computed regions as $A_{R,n}$, where n is a natural number. In this notation, memory states with higher numbers in their index need to be computed after those with lower numbers, and can use the lower-numbered states as inputs.

Unfortunately, the notation presented above does not adequately handle a rather common case: tasks that do not have an ordering relationship with respect to each other. For example,

if we want to compute $\widetilde{A}_R = \hat{A}_R - B - C$, we could denote this as either $A_{R,0} \coloneqq \hat{A} - B; \widetilde{A} \coloneqq A_{R,0} - C$ or we could use the split $A_{R,0} \coloneqq \hat{A}_R - C; \widetilde{A}_R = A_{R,0} - B$. One solution to this problem would be to call the algorithm twice, searching both possible splits, or another approach equivalent to this. However, this results in an unacceptable level of duplication in the results, as there would usually be many loop invariants where both tasks would be computed or uncomputed.

We resolve this problem by allowing disjunctions of memory states as inputs to tasks and by extending our notation for intermediate states that can be in any order with respect to each other. Our notation takes the form $A_{R,(n,x)}$, where n is a number as before and x is an arbitrary symbol. States in this form can have their computation depend on states with the same n but a different x, and act like $A_{R,n}$ otherwise. With this notation, we would write the $\widetilde{A}_R = \hat{A}_R - B - C$ example from above as the tasks $A_{R,(0,a)} \coloneqq (\hat{A}_R \vee A_{R,(0,b)}) - Be$ and $A_{R,(0,b)} \coloneqq (\hat{A} \vee A_{R,(0,a)}) - C$. The or in these expressions is interpreted by our system as representing the fact that each of these tasks could read from either the initial input or the result of the other task, depending on which one is executed first.

There are instances where a task depends on a computation that can be split in multiple ways as described above. For example, one region in the PME for the triangular Sylvester equations [1] is $\widetilde{A}_{TR} = \Omega(\hat{A} - A_{TR}\widetilde{C}_{BR} - \widetilde{C}_{TL}B_{TR})$. We can split the subtraction into $A_{TR,(0,a)}$ and $A_{TR,(0,b)}$ as shown above, but then we are faced with the question of how to notate the call to Ω, which requires the subtraction to be fully computed. Our resolution relies on the fact that our analysis does not take into account the structure of the tasks and only extracts out the (disjunctions of) memory states from the expressions. Therefore, we can write the final task as $\widetilde{C}_{TR} \coloneqq_O \Omega(C_{TR,(0,a)} \wedge C_{TR,(0,b)})$ or as $\widetilde{C}_{TR} \coloneqq_O \Omega(\text{all}(C_{TR,(0,a)}, C_{TR,(0,b)}))$ or in any other notation the user finds convenient.

An alternate method for representing such cases is enabled by the "comes from" task, which represents noops. This task, which has the form $S \leftarrow f(\ldots)$, is equivalent to $S \coloneqq \ldots$, except that if all of its dependencies are computed, it must also be computed. For example, $\widetilde{U}_{TL} \leftarrow \widetilde{L}_{TL}$ means that, if $\widetilde{L}_{TL}$ is computed in the loop invariant, $\widetilde{U}_{TL}$ must also be computed. This task is useful for encoding equations with multiple outputs, and for providing names for expressions such as $C_{TR,(0,a)} \wedge C_{TR,(0,b)}$ from above.

One final type of task, which need not appear in user input, is the constant task, $const(\hat{A}_R)$. This task has no input dependencies, and is used internally in order to ensure that the direction of iteration through pure inputs that are not shared between loops. This task is automatically created by the software in the cases where it is necessary, such as the fusion of $\widetilde{y} \coloneqq A\hat{x}$ and $\widetilde{w} \coloneqq B\hat{x}$. The additional constraint on this task is that it can be in the loop invariant if and only if $\hat{A}_R$ appears elsewhere in the invariant. This condition, combined with the constraints on loop fusion, prevents the system from generating "fusable" loops that iterate through pure inputs, such at $\hat{x}$ in different directions.

To create these tasks, we search our input for any region A_R such that there is no task in any loop that writes to A_R. If this is the case, we know that A_R is a pure input region. Once we have identified such a region, we add the $const(\hat{A}_R)$ task in any loop where A_R is referenced. This, combined with the constraints on fusability within a region, ensures that we do not generate sequences of loops that read a pure input in different directions, since having the first loop invariant read only $\hat{x}_T$ (leaving $\hat{x}_B$ in the ramainder) and the second

read only $\hat{x}_B$, for example, would result in the second loop reading from a region that was not "computed" in the first one, per the constraint above.

2.2 Constraints within a PME

Our procedure for finding loop invariants proceeds by generating the constraints that govern when a given task may or must be in the loop invariant or the remainder of its loop, and then identifying all solutions to this system of constraints.

For uniformity, we define $A_{R,\bot}$ as another notation for $\hat{A}_R$ and $A_{R,\top}$ as notation for $\widetilde{A}_R$

Within a PME, we associate each task output $A_{R,\sigma}$ with an indicator $[A_R, \sigma]$, which has domain $\{0, 1\}$, that reflects whether the task that computes $A_{R,\sigma}$ is in the loop invariant ($[R, \sigma] = 1$) or in the remainder. In the fusion problem, the indicators take the form $[A_R, \sigma, n]$, where n is the 0-based index of the loop from which a given task arises. However, since this section discusses the constraints that arise between tasks in any one given loop, we will omit the ns. These indicator variables allow us to define the constraints on our solution space in the language of constraint propagation over a finite domain.

The first constraints we impose relate to ensuring that the placement of tasks respects these dependencies. In order to impose these constraints, we need to define what it means for one memory state to precede another in memory. This notion is important because it allows us to determine that no tasks in the loop invariant will have overwritten any inputs needed to a task in the remainder. In addition, phrasing constraints in the form of predecessor states allows the user to, for example, use $\widetilde{A}_R$ in a task definition when the only tasks for that region compute $A_{R,(0,a)}$ and $A_{R,(0,b)}$, because both of those states are predecessors of $\widetilde{A}_R$.

We say that $A_{R,\sigma}$ precedes $A_{R,\sigma'}$, or $pr(A_{R,\sigma}, A_{R,\sigma'})$ if:

- $\sigma = \bot$ and $\sigma' \neq \bot$; or
- $\sigma = m$ (or $\sigma = (m, x)$), $\sigma' = n$ (or $\sigma' = (n, y)$), and $m < n$; or
- $\sigma \neq \top$ and $\sigma' = \top$

States from different regions do not precede each other.

We also define the set $\mathcal{O}$ to be the set of output states in a PME, that is, the set of $A_{R,\sigma}$s that have indicator variables which correspond to them.

If we have a task $A_{R,\sigma} \coloneqq f(A_{S_{1,1},\sigma_{1,1}}, A_{S_{2,1},\sigma 2,1} \vee A_{S_{2,2},\sigma_{2,2}} \ldots \vee A_{S_{2,k_2},\sigma_{2,i_2}} \ldots, A_{S_{k,1},\sigma_{k,1}})$, we impose the constraints

$$[A_R, \sigma] = 1 \Rightarrow \bigwedge_{i=1}^{k} \bigvee_{j=1}^{k_i} \bigwedge_{\substack{A_{S',\sigma'} \in \mathcal{O}, \\ pd(A_{S'\sigma'}, A_{S_{i,j},\sigma_{i,j}})}} [A_{S'}, \sigma'] = 1$$

and

$$[A_R, \sigma] = 0 \Rightarrow \bigwedge_{i=1}^{k} \bigvee_{j=1}^{k_i} \bigwedge_{\substack{A_{S',\sigma'} \in \mathcal{O}, \\ pr(A_{S_{i,j},\sigma_{i,j}}, A_{S'\sigma'})}} [A_{S'}, \sigma'] = 0$$

where $pd(A_{R,\sigma}, A_{R'\sigma'})$ means that $A_{R,\sigma} = A_{R'\sigma'}$ or $pr(A_{R,\sigma}, A_{R',\sigma'})$.

That is, for a task to be in the loop invariant, all of its inputs and all their predecessors must also be in the loop invariant (and if an input is an or, than only one branch of the or needs its input and predecessors). Similarly, for a task to be in the remainder, all the successors of its inputs must also be in the remainder.

The justification for these constraints is that a task cannot be in the invariant unless all of its inputs have been computed (which requires their predecessors to be computed), and that, if an input is an or, only one of its branches needs to be available. Similarly, a task cannot be in the remainder if any of its inputs have been overwritten by a task in the loop invariant (that is, a task that produces a successor of the input).

If the task producing $A_{R,\sigma}$ is a comes from task, that is, a noop, the constraint on its presence in the loop invariant becomes an equivalence instead of an implication to ensure that we do not generate spurious results in cases where a noop could be computed but isn't.

A simple example of the constraints generated by these rules comes from the Cholesky factorization, which has the task-based PME

$$\left(\begin{array}{c|c} \widetilde{A}_{TL} \coloneqq_O CHOL(\hat{A}_{TL}) & * \\ \hline \widetilde{A}_{BL} \coloneqq \hat{A}_{BL}\widetilde{A}_{TL}^{-T} & \begin{array}{l} A_{BR,0} \coloneqq \hat{A}_{BR} - \widetilde{A}_{BL}\widetilde{A}_{BL}^{T}; \\ \widetilde{A}_{BR} \coloneqq_O CHOL(A_{BR,0}) \end{array} \end{array}\right)$$

This PME generates the non-trivial constraints

$$\begin{aligned} [A_{BL}, \top] = 1 &\Rightarrow [A_{TL}, \top] = 1 \\ [A_{BR}, 0] = 1 &\Rightarrow [A_{BL}, \top] = 1 \\ [A_{BR}, 0] = 0 &\Rightarrow [A_{BR}, \top] = 0 \\ [A_{BR}, \top] = 1 &\Rightarrow [A_{BR}, 0] = 1 \end{aligned}$$

which are not sufficient to define valid loop invariants, as they will, for example, admit the solution that sets all indicators to 1, or a loop invariant where the matrix is completely factored at the beginning of each iteration.

To solve this issue, we add the constraint that ensures that the loop invariant will make progress. If there are k outputs $A_{R_1,\sigma_1}, \ldots A_{R_k,\sigma_k}$ that are the result of operation tasks, then we require that

$$0 < \sum_{i=1}^{k} [A_{R_i}\sigma_i] < n.$$

That is, not every operation task can be in the loop invariant and not all of them can be in the remainder. This ensures that there will be a region that can contain the initial input (that is, the one which has the algorithm's operation in the future) and one to contain the final output (which has the operation in the past).

A final type of constraint arises from const tasks, which are used to keep loops that iterate through pure inputs moving in the same direction. If $const(\hat{A}_R)$ is a task, and $A_{R_1,\sigma_1}, \ldots A_{R_k,\sigma_k}$ are the outputs of tasks that have A_R appear anywhere in their inputs,

then we impose the constraint

$$[R, \bot] = 1 \Leftrightarrow \vee_{i=1}^{k} [A_{R_k} \sigma_k] = 1.$$

In other words, a const task is computed if and only if one of the tasks that reads that input region is computed. This constraint is mainly necessary for cases where the input appears in an or, where, if a different branch of the or is satisfied, $[A_R, \sigma]$ is completely unconstrained (generating spurious results), since its task has no input dependencies.

2.3 Constraints for fusion

To ensure loop fusion, we need to ensure that tasks do not read data which previous loops have not fully computed for them, and that tasks to not overwrite data that future iterations of earlier loops will need to read. This condition translates to a series of additional constraints on our search space.

To describe these constraints, we will, for each region A_R, define the integer variables C_{A_R} and U_{A_R}, which represent the index of the last loop in which A_R is fully computed and the first region in which it is uncomputed, respectively. We will not directly search through values for these variables, but we will use them to constrain our solution space.

The domain of each C_{A_R} is $-1 \leq C_{A_R} < N$, where N is the number of loops being fused. The -1 is there to provide a value for C_{A_R} when no loop fully computes A_R. Similarly, $0 \leq U_{A_R} \leq N$, to allow for the case where all loops write to A_R to some degree (which causes $A_R = N$).

The first set of constraints we establish is the fusion use constraints. For each task in the nth loop, whose inputs are $A^n_{R_{1,1},\sigma_{1,1}}, A^n_{R_{2,1},\sigma_{2,1}} \vee \ldots \vee A^n_{R_{2,k_2},\sigma 2,k_2}, \ldots, A^n_{R_{k,1},\sigma_{k,1}}$ and output is $A^n_{R,\sigma}$, we impose the fusion use constraints

$$[R, \sigma, n] = 1 \Rightarrow \bigwedge_{i=1}^{k} \bigvee_{j=1}^{k_i} C_{R_{i,j}} \geq n - 1$$

and

$$[R, \sigma, n] = 0 \Rightarrow \bigwedge_{i=1}^{k} \bigvee_{j=1}^{k_i} U_{R_{i,j}} \leq n + 1$$

That is, if a task that reads from region A_R in one of its inputs, A_R must be fully computed through the loop immediately before the one containing that task. Similarly, if a task is in the remainder needs data stored in A_R, then none of the loop invariants that will be fused into the loop body after it can overwrite that data, which needs to be preserved.

The other set of constraints required to preserve fusion are the fusion entailment constraints, which define what C_{A_R} and U_{A_R}'s values imply about the state of the computation. These are, for each region A_R and loop n (where $\mathcal{O}^n$ is the set of task outputs for the nth loop):

$$C_{A_R} \geq n \Leftrightarrow \bigwedge_{A_{R,\sigma} \in \mathcal{O}^n} [R, \sigma, n] = 1$$

and

$$U_{A_R} \leq n \Leftrightarrow \bigwedge_{A_{R,\sigma} \in \mathcal{O}^n} [R, \sigma, n] = 0.$$

With these two sets of constraints, we now have enumerated all the requirements that must be satisfied to produce a fusable collection of loop invariants for a given set of N PMEs.

TODO example that doesn't need a whole bunch of space to list the constraints?

2.4 OLD Finding invariants for one PME

We can now present our method for finding all the loop invariants for one PME now that we have defined how to split a PME into tasks. This problem amounts to finding splits of each region's set of tasks T_R, into disjoint sets P_R (the "past" or loop invariant) and F_R (the "future" or remainder) such that $T_R = P_R \cup F_R$ such that the tasks in the past/future satisfy the conditions needed to create a valid loop invariant. These conditions need to be rephrased in the language of tasks and intermediate memory states.

The algorithm, which is a degenerate case of our fusion algorithm, consists of considering all possible past/future splits for each region and then checking them against the conditions for loop invariant validity. The first of these conditions is that there needs to be on operation task in the past of some region R and one in the future of a different region S. This condition ensures that loop invariants can make progress by using updates that move data A_S to A_R (by performing updates that will take data which satisfies the invariant f_S to data that satisfies f_R). If we did not have this condition, we might crate "loop invariants" where no useful computation is performed at all or ones that assert that the operation is already completed.

The second condition is that the candidate invariant must be one that can be used to synthesize updates. We ensure this by checking that all the dependencies within and between regions are satisfied. At a high level, ensuring that dependencies are satisfied means checking that no task in the loop invariant requires a memory state that cannot be available (because it is computed in the remainder) as an input, and that no task from the invariant will overwrite a memory state that will be needed by a task in the remainder.

To implement such a test, we begin by defining what it means for a memory state $A_{R,\sigma}$ to be before a state $A_{R',\sigma'}$. The intent of these conditions is to have "$A_{R,\sigma}$ is before $A_{R',\sigma'}$ encode the fact that $A_{R,\sigma}$ could appear in the expression that defines $A_{R',\sigma'}$ without violating the rules for dependencies. To improve the presentation, we will also denote $\hat{A}_R$ as $A_{R,\bot}4$ and $\widetilde{A}_R$ as $A_{R,\top}$.

We can say that $A_{R,\sigma}$ is before $A_{R',\sigma'}$ if one of the following conditions is true:

- $R \neq R'$. We allow states from different regions to be mutually before each other since there is no way to determine their relationship in time just by looking at which region they are stored in.
- $\sigma = \bot$ and $\sigma' \neq \bot$, since the input can be used to compute anything but itself.
- $\sigma' = \top$ and $\sigma \neq \top$, since any previous state can be used to compute the final output

- $\sigma = m$ (or (m, x)), $\sigma' = n$ (or (n, y)), and $m < n$, since this encodes the ordering implied by the $A_{R,n}$ notation.
- $\sigma = (n, x)$, $\sigma' = (n, y)$, and $x \neq y$, which ensures the reorderability of these states with respect to each other

To handle ors, we declare that $(A_{R_1,\sigma_1} \vee A_{R_2,\sigma_2} \ldots \vee A_{R_k,\sigma_k})$ is before $A_{R',\sigma'}$ if any of the A_{R_i,σ_i} is before $A_{R',\sigma'}$, and that $A_{R,\sigma}$ is before $(A_{R'_1,\sigma'_1} \vee A_{R'_2,\sigma'_2} \ldots \vee A_{R'_k,\sigma'_k})$ if $A_{R,\sigma}$ is before any of the $A_{R'_i,\sigma'_i}$. This definition allows the dependency analysis to operate correctly in the presence of ors.

With a definition of before, we can now state that a candidate loop invariant has satisfied dependencies if *every* input to a task in the invariant is before every output of a task in the remainder and if every input to a task in the remainder is not after (either before or equal to) every output of a task in the invariant. These conditions formalize the high-level properties around reading unavailable data and overwriting needed data discussed above.

For example, if we had the task $\widetilde{A}_R \coloneqq f_2(\widetilde{A}_S, \hat{A}_R)$ in the loop invariant and $\widetilde{A}_S \coloneqq f(\hat{A}_S)$ in the remainder, this would violate the first half of the condition because the $\widetilde{A}_S$ from the invariant's inputs would not be before the $\widetilde{A}_S$ from the output of the task in the remainder. Similarly, if $\widetilde{A}_R \coloneqq f(\hat{A}_R)$ is in the invariant, while $\widetilde{A}_S \coloneqq f(\hat{A}_S, \hat{A}_R)$ was in the remainder, this would violate the second half of the condition (past outputs must be not after future inputs) because the output $\widetilde{A}_R$ is after the $\hat{A}_R$ in the remainder.

The purpose of our choices regarding ored-together memory states was to ensure that the dependency check would not reject the pair of tasks $A_{R,(0,a)} \coloneqq f(\hat{A}_R \vee A_{R,(0,b)})$ and $A_{R,(0,b)} \coloneqq f(\hat{A} \vee A_{R,(0,a)})$ if one were in the invariant and the other were in the remainder. With our definition, the already-computed branch of the or in the invariant will make the or not come after the input of the task in the remainder, while the uncomputed branch will ensure that the whole or is before the output from the remainder.

One useful consequence of these definitions is that, when the final group of tasks in a region R consists of a group of $A_{R,(n,x)}$ tasks, other regions do not need to list out all of those tasks, but can instead refer to $\widetilde{A}_R$ (which is never explicitly computed within R) to ensure that A_R is fully computed at the time that read should take place.

With the dependency check defined, we now have defined an algorithm for generating all valid loop invariants from a task-based PME.

2.5 OLD Finding fusable loop invariants

The obvious approach to finding fusable loop invariants would be to consider all possible loop invariants for each operation and then check if each of these collections of invariants can be fused. This approach is extremely inefficient because it considers many collections of invariants which can quickly be rejected from consideration by using a different search procedure.

Instead, we implement our algorithm by searching through valid states for *strips*. Strips are a different view of the regions in each fusable loop that arise from "transposing" the fusion problem. Specifically, for each region R, we define the strip S_R as the sequence $[A_R^i \mid 0 \leq i < n]$ (where n is the number of operations being fused). These strips allow us to more

efficiently search for fusable loop invariants because looking for possible invariant/remainder splits at the strip level bypasses many unfusable collections of invariants.

To review, the conditions for loop fusion to be possible are that the invariant f^i cannot read from A_R unless all previous loops fully compute (all tasks are in the invariant) it, and cannot update A_R unless all future loops do not update it. As a corollary, we know that within a strip, all computed regions must be before all uncomputed regions, and there can be at most one partially computed region between the two groups.

This corollary gives us a more efficient initial search than checking every task split in each region of each loop. We can instead choose a partitioning of each strip into the three sections: computed, any (which contains exactly one region), and uncomputed. The search begins with the first region in the strip being the any region (and everything after it being uncomputed) and proceeds by moving the "any" region towards the end of the strip, leaving a trail of computed regions behind it.

Once the location of the "any" region has been determined in a strip, we test all possible splits of the tasks in that region between the loop invariant and remainder. However, if this region is not $S_R[0]$, we do not consider the case where all of its tasks are in the remainder/it is uncomputed. This ensures that we avoid duplicating the case where the previous region was "any" and had been made fully computed.

In our implementation, we use Prolog's non-deterministic nature to choose the states within each strip in sequence. Between each choice, we impose constraints on the splits that can occur in other regions' strips. These constraints take the form of inequalities imposed on two values that are tracked for each region: C_R, which is the index of the last fully computed region in S_R, and U_R, the index of the first uncomputed region. Tracking constraints on these two values, is, because of the conditions on strips, enough to ensure that the remaining (cross-strip) fusion conditions are respected.

Before we begin the search, for every region, we have $-1 \leq C_R, U_R \leq n$. These bounds extend to the side of the possible indices of a region so we can represent the case where there is no fully computed region in a strip ($C_R = -1$) and the case where there are no uncomputed regions ($U_R = n$).

Once we have chosen a possible set of states for the strip S_R, we set C_R and U_R to their true values as reflected by the choice. If previously-imposed constraints cause those values to not be in the domain of C_R or U_R, we will immediately rewind to a different choice of states for S_R, which results in significant savings in the number of states we need to explore and conditions we must check. Then, we walk through the inputs to all the tasks in the invariant and remainder of each region $\mathcal{F}_R^i$. If the ith loop reads from A_S in the invariant, then we impose the constraint that $C_S \geq i - 1$. Similarly, if the ith loop reads from A_S in the remainder, we add the constraint that $U_S \leq i + 1$.

These constraints are sufficient to ensure that none of the loops compute with incorrect data, and their form (which only examines the region being read from, not the precise memory state) means that we can write each algorithm's tasks as if it were not part of a fusion problem. However, even if we find choices for each strip that satisfy the fusion conditions, there is no guarantee that we have produced fusable loop invariants. For example, if every strip leaves $S_R[n-1]$ uncomputed, then loop $n-1$ cannot make progress because it has no operation tasks in its invariant. Therefore, the second phase of our algorithm is to check all task splits that pass the fusion check to ensure that they have resulted in a valid loop

invariant for each loop, a process we discussed in the section above.

2.6 OLD Multiple output objects

There are some extensions that we must make to our algorithm if not every loop writes to the same output object. For example, if are fusing the operations $\widetilde{L} = \hat{L}^{-1}$ and $\widetilde{y} = L\hat{x}$, we notice each operation updates different regions, and that there are potential issues with fusion because $y = Lx$ reads from L. The general concepts of the algorithm remains unchanged, since we can consider y_R and L_R different regions in the same way that L_R and L_S are.

However, there are some problems that we must consider. The first consideration is that, for our search procedure, each strip needs to be the same length. IF each strip was a different length, it would be difficult to keep track of constraints or to refer to objects like the L in $\widetilde{y} = Lx$. So, to solve this, as a prepossessing step, we add empty regions (which have no tasks) to loops as needed to make sure each strip has one region per loop.

This solution requires that we complicate our system of constraints. An example that shows why modifications are needed arises in a case such as this:

- $\widetilde{A}_{TL} \coloneqq f(\hat{A}_{TL})$ is in the invariant of loop 0
- $\widetilde{A}_{TR} \coloneqq g(\hat{A}_{TL}, \widetilde{A}_{TL})$ is in the remainder of loop 0
- $\widetilde{A}_{BR} \coloneqq h(\hat{A}_{TL}, \hat{A}_{BR})$ is in the invariant of loop 3 (and all the updates to A_{TL} are in the remainder)
- Loops 1 and 2 do not write to A

The constraints imposed by this scenario are that $U_{TL} \leq 1$ and that $C_{TL} \geq 2$. We want the tasks in this scenario to be fusable in our system, since there is nothing about this scenario that prevents these loops from fusing. However, there is no single value for the last computed and first uncomputed indices that allows these constraints to be satisfied.

To resolve this, instead of setting the values of C_R and U_R to concrete values, we instead impose the constraints $I_{C_R} \leq C_R \leq I_{C_R} + E_{C_R}$ and $I_{U_R} - E_{U_R} \leq U_R \leq I_{U_R}$ where I_{C_R} and I_{U_R} are the indices of the last computed and first uncomputed regions, respectively, and E_{C_R} and E_{U_R} are the number of empty regions immediately after/before the last computed/first uncomputed region, respectively. This ensures that the value of these constraints can "float" through the nearby empty regions to ensure fusion analysis works correctly.

Interestingly, empty regions do not introduce any additional duplication, since, if any empty region is anywhere but at the front of a strip, its one possible task split will be trivially uncomputed (and therefore skipped) when that region becomes the "any" region. If the empty region is at the beginning of the loop, it will serve to produce the only instance in the search where all the regions in the loop are uncomputed.

The second issue that arises with multiple output matrices is tasks that write to multiple regions. The usual example of this arises during the computation of the LU factorization, where, mathematically, we want $\{\widetilde{L}, \widetilde{U}\} = LU(\hat{A})$. However, our system as currently specified only allows for tasks to create one output. These issues must be resolved by declaring comes from tasks, such as $\widetilde{U}_{TL} \leftarrow \widetilde{L}_{TL}$ to introduce the additional regions thorough noops.

2.7 OLD An example

To demonstrate the operation of this theory, we can consider the inversion of a symmetric matrix, given by the following three task based PMEs. (We will tag operation tasks with $:=_O$). The operations needed to invert a symmetric matrix A are the Cholesky factorization, followed by a triangular inverse and the multiplication AA^T.

0.
$$\left(\begin{array}{c|c} \widetilde{A}_{TL} :=_O CHOL(\hat{A}_{TL}) & * \\ \hline \widetilde{A}_{BL} := \hat{A}_{BL}\widetilde{A}_{TL}^{-T} & \begin{array}{c} A_{BR,0} := \hat{A}_{BR} - \widetilde{A}_{BL}\widetilde{A}_{BL}^T; \\ \widetilde{A}_{BR} :=_O CHOL(A_{BR,0}) \end{array} \end{array}\right)$$

1.
$$\left(\begin{array}{c|c} \widetilde{A}_{TL} :=_O \hat{A}_{TL}^{-1} & * \\ \hline \begin{array}{c} A_{BL,(0,a)} := (\hat{A}_{BL} \vee A_{BL,(0,b)}) \cdot \widetilde{A}_{TL}; \\ A_{BL,(0,b)} := -\hat{A}_{BR}^{-1} \cdot (\hat{A}_{BL} \vee A_{BL,(0,a)}) \end{array} & \widetilde{A}_{BR} :=_O \hat{A}_{BR}^{-1} \end{array}\right)$$

2.
$$\left(\begin{array}{c|c} \begin{array}{c} A_{TL,0} :=_O \hat{A}_{TL}\hat{A}_{TL}^T; \\ \widetilde{A}_{TL} := A_{TL,0} + \hat{A}_{BL}^T\hat{A}_{BL} \end{array} & * \\ \hline \widetilde{A}_{BL} := \hat{A}_{BR}\hat{A}_{BL}^T & \widetilde{A}_{BR} :=_O \hat{A}_{BR}\hat{A}_{BR}^T \end{array}\right)$$

First, let us consider possible splits of the TL strip. Leaving all of the TL regions uncomputed cannot lead to possible loop invariants, as fully computing A_{BR}^0 requires A_{TL}^0 to be fully computed. So, we know we need A_{TL}^0 to be computed. Because of this, we cannot fully compute A_{BR}^0, as this would leave no Cholesky factorizations in the remainder.

These constraints force us to compute A_{TL}^1, since the inverse in $\mathcal{F}_{BR}^1$ cannot be computed as the data to be inverted is not available yet and there are no other matrix inversion tasks. Similarly, the only operation tasks in the PME for the multiplication are in the top left and the bottom right, and teh bottom right one cannot be computed because of the fusion conditions. Therefore, at the very least, we must compute $A_{TL,0}^2$ by performing $\hat{A}_{TL}^2(\hat{A}_{TL}^2)^T$. From our analysis of S_{TL} and the available operation tasks, we have shown that there are two potential options for S_{TL}, which only differ in whether A_{TL}^2 is partially or fully computed. In either case, we have $1 \leq C_{TL} \leq 3$ and $U_{TL} = 3$, as well as $C_{BR} = -1$ and $U_{BR} \leq 1$.

Now, if we consider S_{BL}, we know that we must compute A_{BL}^0. If we did not, then there would be a reference to A_{TL}^0 in the remainder, which would mean that $3 \leq 1$ would need to be true. Similarly, we must compute the task $A_{BL,(0,a)}^1$ (that is, $\hat{A}_{BL}^1\widetilde{A}_{TL}^1$) for the same reason. However, we cannot fully compute A_{BL}^1, as that computation requires $C_{BR} \geq 0$. Because A_{BL}^1 is partially computed, we cannot fully compute A_{TL}^2 (or do any work on A_{BL}^2), forcing $C_{TL} = 1$. This analysis shows us that C_{BL} must be 0 and U_{BL} must be 2.

These constraints on the state of the bottom left also force us to partially compute A_{BR}^0, since, if we did not, we could never perform the computation, since the bottom left would be overwritten by progress on the inverse. More formally, leaving A_{BR}^0 uncomputed would require $U_{BL} \leq 1$, which is not compatible with $U_{BL} = 2$. Therefore, $U_{BR} = 1$ and we completely specified the partitionings for each strip.

While determining that these splits are the only ones that can lead to fusable invariants, we have also shown above there is only one partitioning of the partially computed regions that leads to a valid sequence of loop invariants, which is:

1. $$\left(\begin{array}{c|c} A_{TL} = CHOL(\hat{A}_{TL}) & * \\ \hline A = \hat{A}_{BL}\widetilde{A}_{TL}^{-T} & A_{BR} = \hat{A}_{BR} - \widetilde{A}_{BL}\widetilde{A}_{BL}^{T} \end{array}\right)$$

2. $$\left(\begin{array}{c|c} A_{TL} = \hat{A}_{TL}^{-1} & * \\ \hline A_{BL} = A_{BL,(0,b)}\widetilde{A}_{TL} & A_{BR} = \hat{A}_{BR} \end{array}\right)$$

3. $$\left(\begin{array}{c|c} A_{TL} = \hat{A}_{TL}\hat{A}_{TL}^{T} & * \\ \hline A_{BL} = \hat{A}_{BL} & A_{BR} = \hat{A}_{BR} \end{array}\right)$$

References

[1] Paolo Bientinesi et al. "The science of deriving dense linear algebra algorithms". In: *ACM Transactions on Mathematical Software* 31.1 (2005), pp. 1–26. ISSN: 00983500. DOI: 10.1145/1055531.1055532.

[2] Tze Meng Low. "A Calculus of Loop Invariants for Dense Linear Algebra Optimization". PhD thesis. University of Texas at Austin, 2013.