The document where I explain my algorithm

Krzysztof A. Drewniak

May 23, 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 $x \rightarrow \left(\frac{x_T}{\overline{x_B}}\right)$ where length$(x_T) = 0$
do until length$(X_T) = n$
  repartition $\left(\frac{x_T}{\overline{x_B}}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline\hline \chi_1 \\ \hline x_2 \end{array}\right)$

  $\vdots$] loop body

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

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

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

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

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

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

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

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

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

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

In order to allow for the automated generation of (fused) loop invariants, we first need to represent each region's function $\mathcal{F}_i^R$ as a series of *tasks*, similarly to the approach used by CLICK**TODO find that paper**. To develop this representation, we need to introduce notation for the possible states that an object can be in during a loop.

We already have $\hat{A}_R$ for the initial state of A_R and $\widetilde{A}_R$ for the final state. However, we also need a notation for partial states. We will represent partially computed states of A_R as $A_{R,n}$ for some natural number n or as $A_{R,(n,x)}$ for a number n and symbol x. These two notations allow us to indicate which partial states can and must be computed before other states.

For example, if we consider $\widetilde{A}_{BR} = CHOL(\hat{A}_{BR} - \widetilde{A}_{BL}\widetilde{A}_{BL}^T)$, we can rewrite this as $A_{BR,0} = \hat{A}_{BR} - \widetilde{A}_{BL}\widetilde{A}_{BL}^T; \widetilde{A}_{BR} = CHOL(A_{BR,0})$. This indicates that an algorithm that only computes $A_{BR,0}$ is an option we want to consider.

The extended $A_{R,(n,x)}$ notation is needed for cases such as $\widetilde{A}_R = B\hat{A}_R C$, where there are multiple possible ways to split the expression that lead to valid algorithms. We could partition it as $A_{R,0} = B\hat{A}_R; \widetilde{A}_R = A_{R,0}C$, or as $A_{R,0} = \hat{A}_R C; \widetilde{A}_R = BA_{R,0}$. To represent both of these possibilities, we will write the expression as $A_{R,(0,a)} = B(\hat{A}_R \vee A_{R,(0,b)}); A_{R,(0,b)} = (\hat{A}_R \vee A_{R,(0,a)})C$, where the ors indicate that either state can be used to begin the computation. (The final state $\widetilde{A}_R$ is implied by executing both tasks.)

This notation allows us to represent each region R in the PME as a set of tasks T_R, each of which brings R into some state and depends on a set of operands, which may be particular memory states of R or other regions. In this model, the loop invariant is the set of past tasks P_R, and the remainder is the set of future tasks F_R, such that T_R is the union of the past and future, which are disjoint. We can abstract this representation further to simplify the description and implementation of the algorithm. We can write each task as $A_{R,\sigma} \coloneqq I_{R,\sigma}$ for some set of inputs $I_{R,\sigma}$, which are (disjunctions of) symbols of the form $M_{R,\sigma}$ for some object M, region R of M, and memory state σ. For uniformity, we represent $\hat{A}_R$ as $A_{R,\perp}$ and $\widetilde{A}_R$ as $A_{R,\top}$ for uniformity.

One potential complication with this abstraction loses information on which tasks represent the main operation of the algorithm we are searching for, which is information needed to verify that a loop makes progress. Tasks with such a tag are *operation tasks*.

In this representation, a region is fully computed when $P_R = T_R$, and uncomputed if $T_R = F_R$.

2.2 Finding invariants for one PME

To produce and algorithm for finding all the loop invariants for one PME, we need to translate the conditions such an invariant must satisfy into the task-based setting.

The process of finding all invariants begins by considering all possibly partitions of each set of tasks from the PME into past and future sets. Then, we filter these partitions in order to only produce valid loop invariants. The first filter consists of ensuring that the candidate invariant has distinct regions P and Q where there is an operation task in P_T and one in F_Q.

The second condition ensures that all dependencies are satisfied. In order to express this condition, we must define what it means for a memory state A_{R_1,σ_1} to be before the state A_{R_2,σ_2}. These two states are before each other any of the following are true:

- R_1 and R_2 are different regions
- $\sigma_1 = \perp$ and σ_2 is any other state
- $\sigma_2 = \top$ and σ_1 is any other state
- $\sigma_1 = m$ or (m, x) and $\sigma_2 = n$ or (n, y), where $m < n$
- $\sigma_1 = (n, x)$ and $\sigma_2 = (n, y)$ where $x \neq y$.

Similarly, A_{R_1,σ_1} is not after A_{R_2,σ_2} if the are the same memory state or A_{R_1,σ_1} is before A_{R_2,σ_2}.

A disjunction of states is before (or not after) a state $A_{R,\sigma}$ if any of the states in the disjunction is before $A_{R,\sigma}$. Similarly, $A_{R,\sigma}$ is before (or not after) a disjunction if it is before/not after any of the elements in the disjunction.

To perform the dependency check, we define I_P to be the set of all inputs to tasks in the past (and similarly I_F the set of future inputs). We also define O_P to be the set of all memory states produced by a task in the past (and, again, O_F to be the set of memory states computed in the future). Confirming that all data dependencies are satisfied is then reduced to ensuring that, for each $s \in O_P$ and $t \in I_F$, s is not after t and that, for every $s \in I_P$ and $t \in O_F$, s is before t. These two checks ensure no past output overwrites a state that a future computation requires and that no computation in the invariant requires a stat that has not been produced, respectively.

From the theory of FLAME, we know that these conditions are sufficient to define a valid loop invariant for an operation $\widetilde{A} = \mathcal{F}(\hat{A})$.

2.3 Finding fusable loop invariants

Our method for finding fusable loop invariants rests on a corollary of the conditions needed for a sequence of loop invariants to be fusable. If we organize the fusion problem as a series of strips $S_R = [A_R^i \mid 0 \leq i < n]$ (that is, if we "transpose" the problem), we can show that, in a collection of fusable loop invariants, each such strip must begin with a (possibly empty) sequence of fully computed regions, which followed by at most one partially computed region, with the remaining regions uncomputed.

This arrangement ensures the fusion-related constraints between the same region in different loops are satisfied. Therefore, we begin by exploring all possible splits of each strip into a (possibly empty) computed strip, an "any" region P, and an uncomputed strip. We allow the "any" region to vary between possible partitionings between past and future. However, we do not investigate where P is uncomputed and not the first region, as this case would have already been checked earlier in the search.

While performing this search, we track constraints on the index of the loop in which a region is last computed, C_R, and the first index where it is uncomputed U_R. These values both start with a domain of $[-1, n]$. The additional indices allow us to represent the cases where no regions are computed and where no regions are uncomputed, respectively.

After we have chosen the past/future partitions for a given strip, we can then use the positions of the tasks to impose constraints on C_R and U_R. We know that, if a task in the invariant of the ith loop has a state of the region P as an input, then it must be the case that $C_P \geq i-1$, so that the value of A_P will be correct during the read. Similarly, inputs to tasks from the remainder imply that $U_P \leq i+1$ to prevent required memory states from being overwritten.

We also know that, once we have made past/future assignments for a strip S_R, we can determine the true values of C_R and U_R. This allows us to quickly reject partitions of a strip that would conflict with partitions already fixed for other regions.

Once we find a solution to these fusion-related constraints, we conclude by ensuring that we have produced loop invariants for each $\mathcal{F}^i$ using the method outlined in the previous subsection. It should be noted that, if there is only one operation to fuse, this algorithm reduces to the invariant-finding approach from the previous section.

2.4 Multiple output objects

The algorithm presented above can be extended to cases where operations update different output matrices, such as the program $\widetilde{y} = \hat{L}x; \widetilde{L} = \hat{L}^{-1}$. This operation updates two matrices, y and L, and the updates can be fused together. Most of the algorithm proceeds unmodified with the observation that regions of different matrices are distinct.

We do, however, need to add empty regions (regions with no tasks), which must be added to some loop invariants to ensure that all strips have the same length. Adding empty regions does not add duplicate or incorrect results, since empty regions are, by definition, uncomputed, which ensures any empty regions will be skipped by the anti-duplication filter (and an initial empty region will test the case of all actual regions being uncomputed).

Empty regions do require us to make the constraints on C_R and U_R more complex, as, in different contexts, it is necessary to consider either the index of the actual last computed/first uncomputed region or the index of any of the empty regions immediately following it. Since the empty regions do not have any tasks, allowing C_R and U_R to float into them does not affect the correctness of the search. Therefore, instead of imposing an equality constraint on C_R and U_R after the strip S_R is partitioned, we bound the values between the index of the last computed/first uncomputed region in the strip and the index end of the sequence of empty regions (if there are any) following that region. With this change, the algorithm operates correctly for problems where empty regions needed to be inserted.

The second enhancement we need to make to our algorithm concerns operations that logically update multiple regions. For example, the LU factorization splits a matrix A into lower and upper triangular matrices L and U such that $LU = A$. Our system represents such operations as a series of tasks that updates one of the regions (P), and place a "comes from" task $B^i_{Q,\top} \leftarrow A^i_{P,\top}$ as the only task in the other (Q). The dependency enforcement mechanisms prevent this task from being completed unless A^i_P is fully computed. To prevent spurious results, we impose the condition that B^i_Q is computed if only if A^i_P is.

2.5 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.

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

1.

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

2.

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

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^0_{BR} requires A^0_{TL} to be fully computed. So, we know we need A^0_{TL} to be computed. Because of this, we cannot fully compute A^0_{BR}, as this would leave no Cholesky factorizations in the remainder.

These constraints force us to compute A^1_{TL}, since the inverse in $\mathcal{F}^1_{BR}$ 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 $A^2_{TL,0}$ by performing $\hat{A}^2_{TL}(\hat{A}^2_{TL})^T$. 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^2_{TL} 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^0_{BL}. If we did not, then there would be a reference to A^0_{TL} in the remainder, which would mean that $3 \leq 1$ would need to be true. Similarly, we must compute the task $A^1_{BL,(0,a)}$ (that is, $\hat{A}^1_{BL}\widetilde{A}^1_{TL}$) for the same reason. However, we cannot fully compute A^1_{BL}, as that computation requires $C_{BR} \geq 0$. Because A^1_{BL} is partially computed, we cannot fully compute A^2_{TL} (or do any work on A^2_{BL}), 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^0_{BR}, 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^0_{BR} 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. $$\left(\begin{array}{c||c} A_{TL} = CHOL(\hat{A}_{TL}) & * \\ \hline\hline A = \hat{A}_{BL}\widetilde{A}^{-T}_{TL} & A_{BR} = \hat{A}_{BR} - \widetilde{A}_{BL}\widetilde{A}^{T}_{BL} \end{array}\right)$$

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

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

3 Implementation

Our implementation is a fairly direct translation of the algorithm presented above into SWI Prolog, using the `clpfd` to handle integer constraint. The input to our program is PMEs for each operation, written as the set of tasks that appear in any regions. We allow the description of a task to nest the operands to the task in arbitrary terms, which are ignored.

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.