1

CDA LOOP TRANSFORMATIONS

Dattatraya Kulkarni and Michael Stumm

Department of Computer Science and

Department of Electrical and Computer Engineering

University of Toronto, Toronto, Canada, M5S 1A4

ABSTRACT

In this paper we present a new loop transformation technique called *Computation Decomposition and Alignment* (CDA). *Computation Decomposition* first decomposes the iteration space into finer computation spaces. *Computation Alignment* subsequently, linearly transforms each computation space independently. CDA is a general framework in that linear transformations and its recent extensions are just special cases of CDA. CDA's fine grained loop restructuring can incur considerable computational effort, but can exploit optimization opportunities that earlier frameworks cannot. We present four optimization contexts in which CDA can be useful. Our initial experiments demonstrate that CDA adds a new dimension to performance optimization.

1 INTRODUCTION

The introduction of linear transformations in 1990 as an algebraic framework for loop optimization [5, 22] was a major contribution for three reasons: First, the framework provides a unified approach to loop restructuring since most existing loop transformations [19, 23] and arbitrary sequences of these transformations can be represented by a single transformation matrix. Second, the framework allowed the development of a set of generic techniques to transform loops in a systematic way, independent of the nature of transformations in the compound transformation. Finally, it made possible semi-quantitative evaluation of candidate transformations [4, 13, 15, 22].

A linear transformation changes the structure of a loop so as to change the execution order of the iterations. But note that it does not change the constitution of the iterations themselves: a given iteration in the new iteration space performs the same computations as the corresponding iteration in the original iteration space, only at a different time.

In the last three years Computational Alignment (CA) frameworks have been proposed that extend linear transformations [9, 11, 21]. CA applies a separate linear transformation to each statement in the loop body. Since the new execution order of a statement can be different from that of another statement, CA transformations can alter the constitution of the iterations. A statement is, however, always mapped in its entirety. The origins of the basic idea in CA can be traced to loop alignment [2, 18] which is a special case of CA. The statement level transformation retains the advantages of linear transformations while enabling additional code optimizations. For example, a CA can be used to align the lhs references in statements so that all lhs data elements accessed in an iteration are located on the same processor. This eliminates ownership tests and thus improves the efficiency of SPMD code.

CDA is a generalization of CA and goes a step further in that it can move computations of granularity smaller than a statement. Instead of transforming the statements as written by the programmer, CDA first partitions the original statements into finer statements. This creates additional opportunities for optimization. Thus, a CDA transformation has two components. First, *Computation Decomposition* divides the statements in the loop body into smaller statements. Then, *Computation Alignment* linearly transforms each of these statements, possibly using a different transformation for each.

We intend to show in this paper that there are benefits in transforming a loop at subexpression granularity. Because CDA is a generalization of linear loop transformations and CA, it has all their advantages and can achieve everything they can and more. CDA can potentially exploit much of the flexibility available in loop structures to extend existing local optimizations, or to minimize constraints that are otherwise treated global. CDA does, however, also have serious drawbacks. The derivation of a suitable CDA requires considerable computational effort. The search space for CDA's is so much larger than that for linear transformations that good heuristics are even more important. Other drawbacks are that CDA may increase memory requirements, may introduce additional dependences, and may produce complex transformed code. However, we believe there are many situations where the benefits of CDA outweigh its drawbacks.

We present an overview of Computation Decomposition and Computation Alignment in Sections 2 and 3, respectively. A simple example of how CDA is applied is given in Section 4. The fine grain manipulation of the loop computation and memory access structures enables the application of CDA to several optimization contexts; four of them are listed in Section 5. We present the results of some representative experiments that demonstrate the promise of CDA as a loop restructuring technique om Section 7.

2 COMPUTATION DECOMPOSITION

Computation Decomposition first decomposes the loop body into its individual statements and then may additionally decompose individual statements into statements of finer granularity. Because of this, CDA has more alignment opportunities than does CA alone. The choice of subexpressions that are elevated to the status of statements is a key decision in CDA optimization. As we see later in Section 5, the optimization objective influences this decision.

A statement is decomposed by rewriting it as a sequence of smaller statements that accumulate the intermediate results and produce the same final result. Consider a statement $\mathtt{S}_{\mathrm{j}}$ in a loop body :

$$\mathtt{S}_{\mathrm{j}}: \ w_j \leftarrow f_{j.1}(R_{j.1}) \ op \ f_{j.2}(R_{j.2})$$

where w_j denotes the lhs array reference. $R_{j.1}$ and $R_{j.2}$ are the sets of references in subexpressions $f_{j.1}(R_{j.1})$ and $f_{j.2}(R_{j.2})$ respectively. The above statement can be decomposed into the following two statements to produce the same result :

$$\begin{aligned} \mathtt{S}_{\mathrm{j.1}}: \ & t_j \leftarrow f_{j.1}(R_{j.1}) \\ \mathtt{S}_{\mathrm{j.2}}: \ & w_j \leftarrow t_j \ op \ f_{j.2}(R_{j.2}) \end{aligned}$$

where t_j is a temporary variable introduced to accumulate the intermediate result. We can repeatedly decompose a statement into possibly many statements, with the result of each new statement held in a different temporary variable. The loop bounds remain unchanged after a computation decomposition. The temporary variables are typically chosen as arrays in order to reduce the number of dependences introduced by the decomposition and this allows for more freedom in the subsequent search for alignments.

The decomposition of statements adds two main complications. First, the temporary arrays may reduce the degree of cache locality achievable, may increase the number of references to memory, and may add to space requirements. However, there are also several optimizations that can reduce some of these overheads. In the best case, a temporary can be eliminated altogether, for example if the lhs array or a dead variable can be used in place of the temporary. Otherwise it may be possible to reduce its dimension and size.[1] Also, sometimes temporary arrays can be reused in loops that follow.

Second, the loop independent flow dependence on a temporary array can later become a loop carried dependence because of alignments that follow. In practice, this often does not introduce additional constraints, for example if it is identical to an already existing dependence.

[1] Note that, for some optimizations it is desirable to use the temporary array, for example to reduce data alignment constraints (see Section 5).

3 COMPUTATION ALIGNMENT

A sequence of decompositions produces a new loop body that can have more statements than the original. We can now employ CA to transform each statement of the new loop body [9, 11, 21]. Analogous to the iteration space, the computation space of a statement S, $CS(S)$, is an integer space representing all execution instances of S in the loop. A separate linear transformation is applied to each computation space. That is, if the decomposition produces a loop body with statements $\mathtt{S}_1, \ldots, \mathtt{S}_{\mathrm{K}}$, which have computation spaces $CS(S_1), \ldots, CS(S_K)$, then we can separately transform these computation spaces by linear transforms $T_1, \ldots, T_K$, respectively. The transformed computation spaces together define the new iteration space as follows. Suppose $(i_1, \ldots, i_n; S_j)$ denotes the execution instance of statement S_j in iteration $(i_1, \ldots, i_n)$. An iteration $(i_1, \ldots, i_n)$ in the original iteration space then consisted of computations:

$$(i_1, \ldots, i_n) \equiv \{(i_1, \ldots, i_n; S_1), \ldots, (i_1, \ldots, i_n; S_K)\}$$

The corresponding iteration in the new iteration space consists of computations:

$$(i_1, \ldots, i_n) \equiv \{(T_1^{-1} \cdot (i_1, \ldots, i_n); S_1), \ldots, (T_K^{-1} \cdot (i_1, \ldots, i_n); S_K)\}$$

Intuitively, the mapping results in a relative movement of the individual computations across iterations. As a result, a new iteration may consist of computations that originally belonged to different iterations. This computation movement is explicitly reflected in the text of the new loop structure. It is for this reason that CDA (and CA) is fundamentally different from traditional linear loop transformations.

If computation space $CS(S)$ is transformed by T, and r is a reference in S with reference matrix R, then r has a new reference matrix, $R \cdot T^{-1}$, after the transformation. The dependence relations change as well. Consider statements $\mathtt{S}_{\mathrm{w}}$ and $\mathtt{S}_{\mathrm{r}}$ in the original code, where $\mathtt{S}_{\mathrm{r}}$ is flow dependent on $\mathtt{S}_{\mathrm{w}}$. Let w be the write reference in $\mathtt{S}_{\mathrm{w}}$ and r be the corresponding read reference.

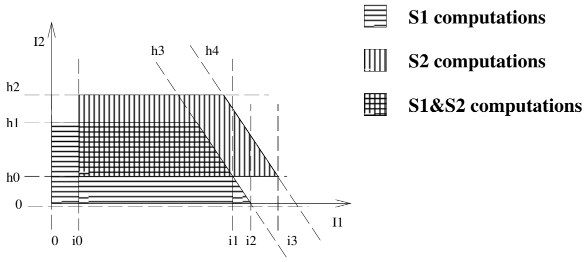
$$w[d_{wr} \cdot \mathtt{I}] \rightarrow r[\mathtt{I}]$$

If T_w is applied to $CS(S_w)$ and T_r is applied to $CS(S_r)$, then the dependence is then transformed to:

$$w[d_{wr} \cdot T_w \cdot \mathtt{I}] \rightarrow r[T_r \cdot \mathtt{I}]$$

which can be rewritten as:

$$w[d'_{wr} \cdot \mathtt{I}] \rightarrow r[\mathtt{I}]$$




```
for I1 = i0, i1
  for I2 = 0, h0−1
    S1: ...
  end for
  for I2 = h0, min(h1,h3)
    S1: ...
    S2: ...
  end for
  for I2 = min(h1,h3)+1, min(h2,h4)
    S2: ...
  end for
end for
```

(a) Transformed Computation spaces **(b) Code for range [i0,i1]**

Figure 1 Segmenting the union.

with $d'_{wr} = T_w \cdot T_r^{-1}$. The transformation is legal if the new dependence relations are positive. This can be easily verified if the dependences are uniform and the transformations are simple offsets: we just have to verify that the last column in d'_{wr} is lexicographically negative (i.e. the write is earlier). If the dependences are non-uniform, then more sophisticated techniques are necessary, such as those that reason with symbolic affine constraints [6, 20]. There are cases, when the only violated dependences are (0) flow dependences between statements and textual interchange will then suffice to make these positive again.

The new loop bounds have to account for each of the transformed computation spaces. The basic idea is to project all computation spaces onto an integer space that becomes the iteration space of the transformed loop. Because transformations $T_1, \ldots, T_K$ can be different, the resulting iteration space can be non-convex. There are two basic strategies that can be pursued to generate code. First, it is possible to take the convex hull of the new iteration space and then generate a *perfect* nest that traverses this hull, but this requires the insertion of *guards* that disable the execution of statements where necessary.

A second, alternative strategy is to generate an *imperfect* nest that has no guards. Guard-free code is usually desirable for better performance, but a perfect loop may be desirable in some cases, for instance to avoid non-vector communications or to avoid loop overheads. An algorithm to generate a guard-free loop for T when all statements require the same loop stride is described in Kelly et al [9]. They also developed an algorithm to generate code for general linear transformations but with conditionals [10]. These algorithms reduce to the algorithm developed by Torres et al. when the transformations are simple offsets corresponding to loop alignments [21].

For completeness, we illustrate a typical way to generate guard free code in Figure 1. The full details can be found in the literature [9, 10, 11, 21]. Assume a loop with two statements $\mathtt{S}_1$ and $\mathtt{S}_2$. The basic idea of the algorithm is to partition the new iteration space into segments that contain iterations with $\mathtt{S}_1$

```
for i = 1, n
  for j = 1, n
                        S1.2                    S1.1
      S1 : A(i,j)   =  A(i,j-1) + B(i,j+1) + A(i-1,j-1) + B(i-1,j) + A(i-1,j)

      S2 : B(i,j-1) =  A (i,j-1) + B(i,j)

  end for
end for
```

(a) Original loop

$$T_{1.1} = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad T_{1.2} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad T_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix}$$

(b) Transformations

Figure 2 An example loop and a CDA transformation.

computation only, or those that contain $\mathtt{S}_2$ computations only, or those with both computations. First split the $\mathtt{I}_1$ axis into ranges that demarcate these segments. For the $\mathtt{I}_1$ range $[0, i_0 - 1]$, the lower bound of the segment is given by 0, and the upper bound by h_1; this segment has only $\mathtt{S}_1$ computations. For the range $[i_0, i_1]$, three segments must be defined. The first is delineated by 0 and h_0 and has only $\mathtt{S}_1$ computations. The second is delineated by h_0 and $min(h_1, h_3)$ and has both $\mathtt{S}_1$ and $\mathtt{S}_2$ computations. The third segment is delineated by $min(h_1, h_3) + 1$ and $min(h_2, h_4)$ and has only $\mathtt{S}_2$ computations. Further segments are defined for the ranges $[i_1 + 1, i_2]$ and $[i_2 + 1, i_3]$.

Given these segments, we can construct a program with a sequence of four loops, one for each defined $\mathtt{I}_1$ range. Each of these loops consists of a sequence of inner $\mathtt{I}_2$ loops, one for each segment defined in that particular $\mathtt{I}_1$ range. The individual bound expressions are chosen so as to delineate the segment. Figure 1b shows the code generated for the three segments defined in the range $[i_0, i_1]$.

4 ILLUSTRATION OF A SIMPLE CDA TRANSFORMATION

We will use the loop of Figure 2a to illustrate the application of a simple CDA transformation (Figure 2b) and its effect on the computation and memory access structures. We assume that there are no pre-existing data alignments. Our goal here is to minimize the data alignment constraints and eliminate the ownership tests.

The loop body of Figure 2a has two statements, $\mathtt{L} = (\mathtt{S}_1; \mathtt{S}_2)$, each with its own computation space: $CS(S_1)$ and $CS(S_2)$. We partition $\mathtt{S}_1$ into two smaller statements $\mathtt{S}_{1.1}$ and $\mathtt{S}_{1.2}$, using a temporary array $\mathtt{t}$ to pass the result of $\mathtt{S}_{1.1}$ on to $\mathtt{S}_{1.2}$. This effectively partitions $CS(S_1)$ into $CS(S_{1.1})$ and $CS(S_{1.2})$ for a total of three computation spaces as shown in Figure 3b. We chose this

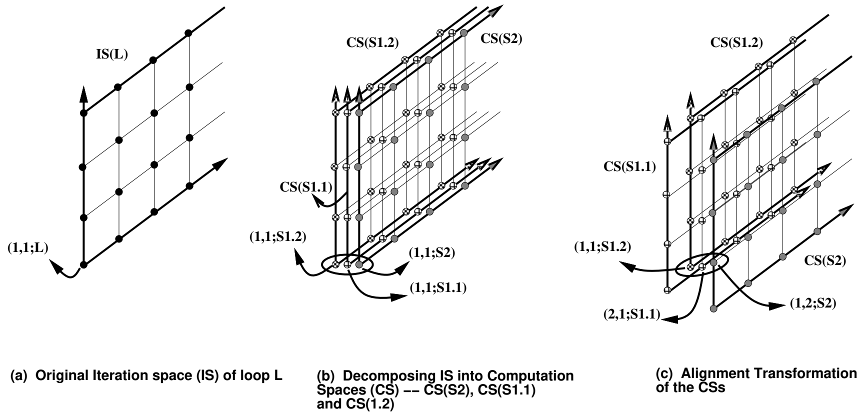


Figure 3 Illustration of a simple CDA transform on the example loop.

particular decomposition for $\mathtt{S}_1$, because all references in $\mathtt{S}_{1.1}$ now have the same data alignment constraint with respect to $\mathtt{A(i,j)}$ along the i-dimension; that is, the first index in the references are all $i-1$. The remaining references in $\mathtt{S}_1$ and $\mathtt{S}_2$ align to $\mathtt{A(i,j)}$ along the i-dimension as they are. This partitioning allows us to align $CS(S_{1.1})$ to $CS(S_{1.2})$ to eliminate the data alignment constraints along the i-dimension for all three references in $\mathtt{S}_{1.1}$, without affecting the references in $\mathtt{S}_{1.2}$ and $\mathtt{S}_2$.

After this decomposition, iteration (i,j) has the following computations :

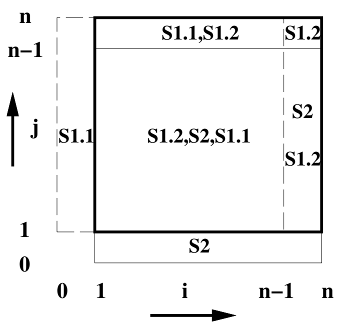
$$(i,j;L) \equiv (i,j;S_{1.1}) \ < \ (i,j;S_{1.2}) \ < \ (i,j;S_2)$$

We can now transform the three computation spaces separately by applying transformations $T_{1.1}$, $T_{1.2}$ and T_2 of Figure 2b to $CS(S_{1.1})$, $CS(S_{1.2})$ and $CS(S_2)$, respectively. The transformations required in this case turn out to be simple offsets. Computation spaces $CS(S_{1.1})$ and $CS(S_2)$ move relative to $CS(S_{1.2})$, which was applied an identity transformation. $CS(S_{1.1})$ moves one stride in direction i in order to change the $(i-1,*)$ references in $\mathtt{S}_{1.1}$ to $(i,*)$ references. and $CS(S_2)$ moves one stride in direction j in order to align $\mathtt{B(i,j-1)}$ to $\mathtt{A(i,j)}$ to remove the need for ownership tests. Figure 3c shows the transformed computation spaces and highlights three computations that are now executed in one iteration.

The new iteration space is defined by the projection of the transformed computation spaces onto a plane which becomes the new iteration space (Figure 4a). Iteration (i,j) in the new iteration space now has the following computations:

$$(i,j;L') \equiv (i,j;S_{1.2}) \ < \ (i,j+1;S_2) \ < \ (i+1,j;S_{1.1})$$

Notice that it was necessary to change the order of the statements in the loop so that $\mathtt{S}_{1.1}$ is executed after the other two statements. Before the transformation,



(a) New iteration space

```
for i = 0, n
  for j = 0, n

   S1.2 :  (i>0;j>0)   A(i,j)     = t(i,j) + A(i,j−1) + B(i,j+1)

   S2   :  (i>0;j<n)   B(i,j)     = A (i,j) + B(i,j+1)

   S1.1 :  (i<n;j>0)   t(i+1,j)   = A(i,j−1) + B(i,j) + A(i,j)

  end for
end for
```

(b) CDA transformed loop

Figure 4 The CDA transformed loop.

$\mathtt{S}_{1.1}$ had a loop carried flow dependence from both $\mathtt{S}_{1.2}$ and $\mathtt{S}_2$. These dependences become loop independent after the alignment, thereby necessitating the reordering.

Figure 4a shows which iterations need to execute which computations. The new iteration space is non-convex. We choose loop bounds to correspond to the convex hull and hence require guards to step around computations that should not be executed. The resulting code is listed in Figure 4b. Code without guards could have been produced as described in Section 3.

The above CDA, although simple, can be very effective, whether targeting a parallel system or a uniprocessor. The transformation has achieved our two main objectives:

(i) The CDA reduced the number of data alignment constraints, thus reducing the amount of communication required. Assuming both $\mathtt{B(i,j)}$ and $\mathtt{t(i+1,j)}$ are aligned to $\mathtt{A(i,j)}$, and assuming the use of the owner computes rule, then the original loop accesses elements $\mathtt{A(i-1,j-1)}$, $\mathtt{A(i-1,j)}$ and $\mathtt{A(i,j-1)}$ in iteration (i,j), while it only accesses $\mathtt{A(i,j-1)}$ and $\mathtt{t(i,j)}$ in the new loop. Similarly, the original loop accesses, elements $\mathtt{B(i,j)}$, $\mathtt{B(i-1,j)}$ and $\mathtt{B(i,j+1)}$, but only accesses $\mathtt{B(i,j+1)}$ in the new loop. In comparison, a CA transformed loop would still need to access a total of 6 elements.

(ii) The original loop needed ownership tests, unless $\mathtt{A(i,j)}$ was aligned to $\mathtt{B(i,j-1)}$. The CDA transformation eliminated the need for these tests, without the need data alignment of $\mathtt{B}$ to $\mathtt{A}$. Moreover, the execution of statement $\mathtt{S}_1$ is now spread over two processors, effectively implementing a modified computation rule.

This transformation has a number of other side effects on the loop that were not specifically part of our goals, but that cause improvements in perforamance nevertheless.

(*iii*) The new loop accesses `A` and `B` in a fundamentally different way, with the dependences changed from:

$$flow : \{(1,1),(0,1),(1,-1),(1,0)\}, anti : \{(0,1),(0,2)\}\}$$

to:

$$flow : \{(0,1),(1,0)\}, anti : \{(0,1)\}, output : \{(1,0)\}\}$$

In comparison, a CA (without first decomposing S) would change the dependences to:

$$flow : \{(1,1),(0,1),(1,0)\}, anti : \{(0,1)\}\}$$

The (1,1) dependence results in non-vectorizable communication with some distributions. CDA (so could a CA) eliminated the (1,-1) dependence. Thus the loop is blockable as is, with indexing and loop bounds simpler than if the loop had first been skewed.

(*iv*) If the the array sizes and the cache geometry are such that $\mathtt{A}(\mathtt{i}-\mathtt{1},\mathtt{j})$ and $\mathtt{B}(\mathtt{i},\mathtt{j})$ conflict in iteration $(\mathtt{i},\mathtt{j})$ of the original loop, then the transformation eliminates these conflicts without any changes to the data layout.

(*v*) The CDA transformation reduces the cache context of `B` from 2n to n elements. This is as a result of bringing the two accesses to each element of `B` in i and $i+1$ outer loop iterations to the same iteration (i). Similar effect on array `A` is negated by the cache context required for the intermediate storage.

(*vi*) The transformation modifies the overall number of loads and stores per iteration from 8 distinct array element accesses to 6 (5 if `t` is replaced by `A`). This can have an impact on register pressure.

5 APPLICATIONS OF CDA

In this section, we list a number of optimization objectives and describe how they can be targetted by CDA. However, it should be noted that CDA is often used to augment existing techniques and is not necessarily intended to replace them.

Removing data alignment constraints

Data alignment transformations are a popular way of removing data alignment constraints. For example, a data alignment transformation T_{da} maps array `B` onto `A` so that references r_B to `B` and r_A to `A` go to the same (hopefully local) processor if

$$T_{da} r_B = r_A$$

Such a data alignment is a global change, since every reference r to `B` in the program is changed to $T_{da} r$.

In some cases, a CDA transformation can have exactly the same effect without having to (data) align array `B`. First, the CDA would have to decompose the loop body so that references to `A` and references to `B` occur in different statements.[2] Second, the linear transformation T_{cda} that satisfies

$$r_B T_{cda}^{-1} = r_A$$

would have to be applied *legally* to each statement with a reference to `B`.[3] (The identity transformation is implicitly applied to all other statements, particularly those with references to `A`.)

However, the power of CDA allows more localized optimization. Instead of applying T_{cda} to each statement with a reference to `B`, it is applied to only those statements with references r_B, leaving the other references to `B` in other statements undisturbed.[4] This is illustrated with the example of Figure 2, assuming that $\mathtt{B(i,j)}$ is aligned to $\mathtt{A(i,j)}$. The CDA transformation $T_{1.1}$ changes the reference $\mathtt{B(i-1,j)}$ to $\mathtt{B(i,j)}$. The same transformation also illustrates what we call *self-alignment*, where reference $\mathtt{A(i-1,j)}$ is aligned to $\mathtt{A(i,j)}$ without affecting accesses to `A` in other statements.

Data alignment and CDA each have their own advantages and drawbacks. Data alignment does not affect dependences and satisfies a constraint between a pair of arrays without affecting other arrays. But, it only satisfies a single constraint, and it modifies the references to the array globally, possibly undoing alignments in some other loop.[5] CDA transformations on the other hand are local to the loop, can potentially remove several data alignment constraints, and do not require data layout changes. However, CDA changes dependences (so legality checking is necessary) and changes all references in a statement. An integrated algorithm might attempt to exploit the advantages of both data alignment and CDA.

Optimizing SPMD code

The elimination of ownership tests results in better performing SPMD code, because a processor does not have to execute every iteration just to check whether it has work to do or not. One way to eliminate ownership tests is to ensure that all statement instances in an iteration are to be executed by the same processor. This can be achieved by transforming statements so as to *collocate* all the lhs references of the loop body if this can be legally done [21]. To achieve this in the CDA transformed loop, we first choose a lhs reference, say $\mathtt{A(i,j)}$ that serves as a basis. Each temporary array is data aligned so that its lhs reference is collocated with $\mathtt{A(i,j)}$. Then, each statement with a lhs reference r is applied a linear transformation, T, such that rT^{-1} and $\mathtt{A(i,j)}$ are collocated. We eliminated the need for ownership tests in the example of Figure 2

[2] This is not always beneficial in practice, as in the following statement: $r_A = r_B + c$.

[3] T_{cda} will change *all* references in the statement `s` being transformed.

[4] It is sufficient to decompose the loop body so as to separate only those references to `A` and `B` with indexing as in r_A and r_B into different statements.

[5] Realignment of data at run-time is usually expensive.

by data aligning $\mathtt{t(i+1,j)}$ to $\mathtt{A(i,j)}$ and (computationally) aligning $\mathtt{S}_2$ to $\mathtt{S}_{1.2}$ so that all three lhs accesses in an iteration become collocated, allowing the entire iteration to be executed by the same processor.

The example of Figure 2 also shows that in general CDA can be viewed to be implementing a class of flexible computation [7] rules with the aid of a fixed computation rule such as owner-computes. The original statement $\mathtt{S}_1$ is executed in parts by two processors instead of the owner of $\mathtt{A(i,j)}$ alone.

Reducing cache conflicts
Array padding is a simple and popular technique that changes the data layout in memory to eliminate cache conflicts. However, array padding is a global change and requires that the size of the arrays be known a priori. More seriously, array padding can be illegal without proper inter-procedural analysis.

CDA can also be used to eliminate cache conflicts. The loop is decomposed into statements such that all (most) references in a statement do not conflict so that conflicts are (mostly) between references in different statements. Computation alignment then moves each statement with respect to the other statements until there are no conflicts. This spreads the conflicting data accesses in an iteration across different iterations. CDA is an attractive alternative to array padding since it does not change data layouts and is therefore always legal. Even when the array sizes are unknown, a simple conditional on the size and the cache geometry can dynamically select between the original code or a CDA transformed code at run-time. However, CDA is constrained by dependences and therefore may not be able to eliminate all conflicts. Moreover, CDA may introduce extra loop overhead compared to array padding.

Reducing communication for a reference stencil
A communication optimal distribution of an array depends on its reference *stencil* in the loop [1, 3, 8]. A CDA can modify the reference stencil, thus providing an additional dimension of optimization in the choice of distribution. Conversely, if an array distribution is given, then it is possible to change the reference stencil to suit the given distribution better.

6 EXPERIMENTAL RESULTS

We summarize the results of five experiments run on the SUN Sparc 20 and the KSR1 platforms to demonstrate the flexibility loops have at fine granularity and how this flexibility can be exploited both on parallel and uniprocessor systems. It should be noted that the KSR1 has a COMA architecture, where the data automatically moves to the processor accessing it.

We chose *mg*, *rtmg*, *slia*, *swm256*, and *wanal*, so as to show improvements over existing transformation frameworks. The loops do not benefit from any linear transformation as such. Three loops, namely, mg, rtmg, and slia have a single statement in their loop body, so CA cannot be applied directly, but only transformations that can be applied at subexpression granularity. Listings of the original and transformed loops, details of the applied CDA, and all of the measurement data can be found in [12].

SLIA

Objective: Removal of data alignment constraints.
SLIA is a synthetic two dimensional loop with $i \pm c$ references to three arrays A, B, and C [12]. The original loop needs data alignment of both $(*, \mathtt{j}-1)$ and $(*, \mathtt{j})$ references of arrays to $\mathtt{A(i,j)}$. Clearly, a data alignment can satisfy only one reference pattern to $\mathtt{A(i,j)}$, not both. We applied CDA to remove the $(*, \mathtt{j}-1)$ data alignment constraints. We decomposed the statement in the loop body to have a statement with $(*, \mathtt{j}-1)$ references and another statement with other references. We then aligned the first statement to obtain $(*, \mathtt{j})$ references. The execution time on the KSR1 improved by 30%-38% when using upto 16 processors.

Swm256

Objective: Elimination of ownership tests without data alignment.
The calc1 subroutine of the SPEC benchmark, SWM256, has 4 statements, with lhs references to $\mathtt{CU(i+1,j)}$, $\mathtt{CV(i,j+1)}$, $\mathtt{Z(i+1,j+1)}$ and $\mathtt{H(i,j)}$. The loop requires ownership tests unless $\mathtt{CU(i+1,j)}$, $\mathtt{CV(i,j+1)}$ and $\mathtt{Z(i+1,j+1)}$ are aligned to $\mathtt{H(i,j)}$. Hence, we aligned the statements so that their lhs references are all of the form $\mathtt{(i,j)}$. The transformed version does not require ownership tests and does not require any data alignments. Because of the shared address space and relatively low cost of remote accesses on the KSR1, the execution time of the transformed code improved by only 17%.

Wanal

Objective: Improving cache locality.
Wanal is a wave equation solver that is part of the Riceps benchmark suite [14]. The three dimensional loop we CDA transformed has two statements in its loop body. A linear transformation cannot improve cache locality here, because only one statement requires a loop interchange, while the other does not. A CDA, which is equivalent to a CA in this case, can be applied to the statement to achieve locality. On the KSR1, the parallel execution time improved by 45-50%.

Rtmg

Objective: Elimination of cache conflicts on the SUN Sparc 20.
The rtmg loop from the Arco Seismic benchmarks suite is a two a dimensional loop with a single statement in the loop body which accesses two dimensional arrays $\mathtt{p1}$ and $\mathtt{p2}$ [16]. There are cache conflicts between the lhs reference $\mathtt{p1(i,k)}$ and the rhs reference $\mathtt{p2(i,k)}$ on a Sparc 20 (with a 1MB, direct-mapped cache). We decomposed the statement into a statement $\mathtt{S}_2$ with references $\mathtt{p1(i,k)}$ and $\mathtt{p2(i-1,k)}$ and statement $\mathtt{S}_1$ contaning all other references . We then aligned $\mathtt{S}_1$ to $\mathtt{S}_2$ such that the $\mathtt{p2(i,k)}$ reference became $\mathtt{p2(i+1,k)}$, leaving the $\mathtt{S}_2$ references unchanged. The indexing of the temporary was chosen as to conflict with $\mathtt{p}_1$ and $\mathtt{p}_2$. The data accesses in an iteration now map to different

cache lines and therefore do not conflict, reducing the execution time by about 50-55% of the original.

Mg
Objective: Elimination of cache conflicts on the KSR1.
NAS mg is a multigrid solver in the NAS benchmarks suite [17]. We CDA transformed the psinv subroutine and were able to improve the speed up by a factor of 2 over the original by eliminating cache conflicts. However, the dependence introduced by CDA reduced the available degree of parallelism.

The loop has iterators i,j, and k, and accesses three dimensional arrays `U` and `R` with $(\mathtt{i} \pm \mathtt{c}_1, \mathtt{j} \pm \mathtt{c}_2, \mathtt{k} \pm \mathtt{c}_3)$ reference patterns, where the `c`'s are either 0,1, or -1. For a given `i` iteration, the loop accesses elements in i^{th} plane of `U`, and elements in $(i-1)$, i and $(i+1)^{th}$ planes of `R`. The references with similar `j` indexing conflict in cache. We decomposed the (only) statement into a statement with references to the $(i-1)^{th}$ plane of `R` (i.e. all $\mathtt{R}(\mathtt{i}-\mathtt{1}, *, *)$ references) and another statement with the other references. We then aligned the two statements so that the $\mathtt{R}(\mathtt{i}-\mathtt{1}, *, *)$ references become $\mathtt{R}(\mathtt{i}, *, *)$ references, effectively eliminating the references to the $(i-1)^{th}$ plane of `R`.[6] The transformed loop, therefore, has fewer planes with similar j indexing, and hence, fewer conflicts.

7 CONCLUDING REMARKS

With respect to optimization, loop structures have considerable flexibility at the subexpression level. Computational Decomposition and Alignment (CDA), which we introduced in this paper, provides a framework to linearly transform loops at this relatively fine granularity. It can be applied to target a number of different optimization objectives. However, heuristics are a key to applying CDAs successfully, since the derivation of a suitable CDA is more complex than say the derivation of a linear transformation. Nevertheless, we are hopeful that it will be possible to find good heristic algorthms that find near optimal CDAs, similar to the way linear transformations are found today.

We believe that CDA will be particularly effective in the context of global optimization, because it can help reduce constraints that are otherwise treated global. It is also interesting to observe that CDA can be applied both to extend current control optimization techniques, as well as to optimizations that are traditionally handled by data layout changes.

In our current work, we are focusing on deriving CDAs that improve cache performance on both uniprocessors and multiprocessors. For example, we are comparing CDA and array padding in reducing the number of cache conflicts

[6] In this case, we can eliminate the need for the temporary by using the lhs array, `U`, to hold the intermediate results.

on numerous benchmark codes, and intend to develop algorithms capable of automatically deriving suitable CDA transformations for this purpose.

Acknowledgements

We thank Ron Unrau at IBM, Toronto Laboratory and Wei Li at the University of Rochester for their contribution to certain aspects of the CDA framework. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada and the Information Technology Research Center of Ontario.

REFERENCES

[1] Abraham, S.G., and Hudak, D.E., "Compile-time partitioning of iterative parallel loops to reduce cache coherency traffic," *IEEE Transactions on Parallel and Distributed Systems*, 2(3):318–328, July 91.

[2] Allen, R., Callahan, D., and Kennedy, K., "Automatic decomposition of scientific programs for parallel execution," In *Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 63–76, Munich, West Germany, January 1987.

[3] Ancourt, C. and Irigoin, F., "Scanning polyhedra with DO loops," In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume 26, pages 39–50, Williamsburg, VA, April 1991.

[4] Anderson, J. and Lam, M., "Global optimizations for parallelism and locality on scalable parallel machines," In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, volume 28, June 1993.

[5] Banerjee, U., "Unimodular transformations of double loops," In *Proceedings of Third Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.

[6] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20, 1991.

[7] Gilbert, J. and Schreiber, R., "Optimal expression evaluation for data parallel architectures," *Journal of Parallel and Distributed Computing*, 13:58–64, 1991.

[8] Irigoin, F. and Triolet, R., "Supernode partitioning," In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 319–329, San Diego, CA, 1988.

[9] Kelly, W. and Pugh, W., "A framework for unifying reordering transformations," Technical Report UMIACS-TR-92-126, University of Maryland, 1992.

[10] Kelly, W., Pugh, W., and Rosser, E., "Code generation for multiple mappings," Technical Report UMIACS-TR-94-87, University of Maryland, 1994.

[11] Kulkarni, D. and Stumm, M., "Computational alignment: A new, unified program transformation for local and global optimization," Technical Report CSRI-292, Computer Systems Research Institute, University of Toronto, January 1994. *http://www.eecg.toronto.edu/EECG/RESEARCH/ParallelSys.*

[12] Kulkarni, D., Stumm, M., Unrau, R., and Li, W., "A generalized theory of linear loop transformations," Technical Report CSRI-317, Computer Systems Research Institute, University of Toronto, December 1994. *http://www.eecg.toronto.edu/EECG/RESEARCH/ParallelSys.*

[13] Kumar, K.G., Kulkarni, D., and Basu, A., "Deriving good transformations for mapping nested loops on hierarchical parallel machines in polynomial time," In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, July 1992.

[14] C.H. Li. Program wanal1. ftp ftp.cs.rice.edu, Rice University, 1992.

[15] Li, W. and Pingali, K., "A singular loop transformation framework based on non-singular matrices," In *Proceedings of the Fifth Workshop on Programming Languages and Compilers for Parallel Computing*, August 1992.

[16] Mosher, C., "Arco Seismic Benchmarks," ARCO E&PT.

[17] NASA, Ames Research Center "NAS Parallel Benchmarks"

[18] Padua, D., "Multiprocessors: Discussion of some theoretical and practical problems," Phd thesis, University of Illinois, Urbana-Champaign, 1979.

[19] Padua, D. and Wolfe, M., "Advanced compiler optimizations for supercomputers," *Communications of the ACM*, 29(12):1184–1201, December 1986.

[20] Pugh, W. and Wonnacott, D., "An exact method for analysis of value-based array data dependences," Technical Report CS-TR-3196, University of Maryland, 1993.

[21] Torres, J., Ayguade, E., Labarta, J., and Valero, M., "Align and distribute-based linear loop transformations," In *Proceedings of Sixth Workshop on Programming Languages and Compilers for Parallel Computing*, 1993.

[22] Wolf, M. and Lam, M., "An algorithmic approach to compound loop transformation," In *Proceedings of Third Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.

[23] Wolfe, M., *Optimizing supercompilers for supercomputers.* The MIT Press, 1990.