A Generalized Theory of Linear Loop Transformations

Dattatraya Kulkarni, Michael Stumm, Ron Unrau and Wei Li

Technical Report CSRI-317
December 1994

Computer Systems Research Institute,
Department of Computer Science,
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Canada
M5S 1A1

The Computer Systems Research Institute (CSRI) is an interdisciplinary group formed to conduct research and development relevant to computer systems and their application. It is an Institute within the Faculty of Applied Science and Engineering, and the Faculty of Arts and Science, at the University of Toronto, and is supported in part by the Natural Sciences and Engineering Research Council of Canada.

A Generalized Theory of Linear Loop Transformations

Dattatraya Kulkarni and Michael Stumm
kulki@cs.toronto.edu
Department of Computer Science
Department of Electrical and Computer Engineering
University of Toronto, Toronto, Canada, M5S 1A4

Ron Unrau
IBM Toronto Laboratory
Toronto, Canada, M3C 1V7

Wei Li
Department of Computer Science
University of Rochester
Rochester, NY 14627

Abstract

In this paper we present a new theory of linear loop transformations called *Computation Decomposition and Alignment* (CDA). A CDA transformation has two components: *Computation Decomposition* first decomposes the computations in the loop into computations of finer granularity, from iterations to instances of subexpressions. *Computation Alignment* subsequently, linearly transforms each of these sets of computations, possibly by using a different transformation for each set. This framework subsumes all existing linear transformation frameworks in that it reduces to a conventional linear loop transformation when the smallest granularity is an iteration, and it reduces to some of the more recently extended frameworks when the smallest granularity is a statement instance. The possibility of being able to align computations at arbitrary granularities adds a new dimensions to performance optimization on high performance computing platforms.

We describe Computation Decomposition and Alignment and provide examples of CDA transformations. We present some heuristics to derive appropriate CDA transformations, given a desired optimization objective. We present the results of experiments run on the KSR1 multiprocessor and various RS6000 and Sparc platforms that demonstrate that CDA can result in substantial performance improvements.

1 Introduction

In this paper we present a new theory of linear loop transformations called *Computation Decomposition and Alignment* (CDA). It can be used to pursue a variety of optimization objectives, both when targeting a uniprocessor system or a parallel system. For uniprocessor systems, CDA can be used to minimize *reference windows* [GJG88] in order to maximize cache reuse and to reduce register pressure. As we will show, it can also be used to move computations to make it easier for later stages of the compiler to optimize processor pipelines. For multiprocessors, besides maximizing cache locality, CDA can be used for data alignment [KN90, LC91, OH92], in order to reduce the number of remote memory accesses, for access normalization [LP92] so that the references match the data alignments, or for the elimination of ownership tests [HKKKT91, HPF93]. A CDA transformation has two components: First, *Computation Decomposition* decomposes the computations of the loop into computations of finer granularity; the granularity of the computations can vary from iterations to instances of subexpressions. Then, *Computation Alignment* subsequently linearly transforms each of these sets of computations, possibly using a different transformation for each set. Being able to align computations at arbitrary granularities adds a new dimension to performance optimization on high performance computing platforms, thus extending significantly the existing frameworks both in power and flexibility.

CDA has its roots in, and is a generalization of *linear loop transformations* [Ban90, WL90, Dow90, KKB91, LP92]. Most of the loop transformations proposed to date [PW86, Wol90, Ban93, Ban94] can be viewed as special instances of linear transformations. In this framework, a transformation or a sequence thereof is represented by a single non-singular matrix, which can be used to directly derive the transformed loop bounds, references and dependences using well known techniques [Ban90, WL90, KKB92, LP92, Ram92, AI91, CFR93]. The introduction of this algebraic framework was a major contribution in that i) it allowed a single matrix to represent a compound transformation of many existing transformations, ii) it allowed the development of a set of generic techniques that could transform loops in a systematic way, independent of the particular sequence of transformations being applied, and iii) it made possible the quantification of optimization objectives [LP92, AL93, Wol92, KKB92, KKB92]. A linear loop transformation does not change the makeup of an iteration as it maps one iteration space (IS) onto another; each iteration is mapped from a point in the original IS to a point in the new IS in its entirety (unchanged). At the code level, a linear loop transformation thus transforms the loop body as a single unit, modifying *all* array references in the loop body in the same way. In contrast, CDA alters the makeup of the iterations and in doing so may move computations across iteration boundaries or spread them across several iterations. Most of the performance advantages of CDA stem from

```
for i = 1, n
  for j = 1, n

    S1  : A(i,j)    =  A(i−1,j−1) + B(i−1,j)  + A(i−1,j)  +  A(i,j−1) + B(i,j+1)

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

  end for
end for
```

Figure 1: A loop to serve as the running example.

this ability to move computations at arbitrary granularity together with their references.

CDA is also closely related to what we call *Computational Alignment* (CA), which applies a separate linear transformation to each statement in the loop body. This notion of applying a separate transformation to each statement has origins in loop alignment [Pad79, AC87], where the transformations are simple offsets. Several researchers independently recognized the advantages of such transformations over the last two years. For example, Kelly and Pugh [KP92] proposed a framework to include separate schedules for each statement. Torres and Aygude [TALV93] integrated loop alignment with linear transformations and applied it to improve the efficiency of SPMD code. Kukarni and Stumm [KS94] presented a framework where spaces of computation are transformed relative to each other; and apply the framework for both local optimizations and global optimizations that maximize the uniformity of data accesses across loops.

CDA, as presented in this paper, goes a significant step further than CA. We show that there are specific benefits in transforming loops at a granularity finer than a statement instance. Syntactically, the statements in the loop body are decomposed into substatements and then separate linear transformations applied to each. Hence, CDA is a generalization of linear loop transformation and CA. when the granularity of computation being transformed is fixed as an entire iteration,[1] then CDA corresponds to a conventional linear loop transformation; when the computations being transformed are statement instances, then CDA corresponds to CA. CDA thus has all the advantages of linear loop transformations and CAs, and can achieve everything these transformation frameworks can, and more.

The following three sections describe Computation Decomposition and Alignment in detail. Throughout, we use the loop of Figure 1 as a running example. In Section 5 we illustrate how CDA can be applied for

[1] I.e., there is no computation decomposition.

a number of different optimization objectives, and Section 6 describes how a simple CDA application to the running example can can result in numerous benefits. Section 7 provides additional examples of a CDA applied to loop nests. There, we present the results of experiments that were run on the KSR1 multiprocessor, a RS6000 workstation cluster, and various RS6000 and Sparc platforms. We show that CDA can achieve significant performance improvements not achievable by linear loop transformations or CAs alone. In the extreme case, we observed performance improvements of up to a factor of 6 on a single processor, when a CDA could substantially alter the cache hit rate.

2 Representation of the Loop Structure

Consider a loop of dimension n, so that the iteration vector is a $n+1$ vector $\mathtt{I_n}$ in a homogeneous co-ordinate system. We assume loop bounds and array subscripts that are affine functions of the enclosing iterators. Therefore, a reference r to an m-dimensional array can be represented by an $m \times (n+1)$ reference matrix,[2] $rm(r)$, and the loop bounds can be represented by an $l \times (n+1)$ bounds matrix,[3] $\beta(L)$, where l is the number of expressions in the upper and lower bounds of the loop [WL90, LP92, Ram92].

Although CDA applies to both perfect and imperfect loop nests [KS94], we restrict ourselves in this paper to computations at the same nesting level. Moreover, for simplicity, we assume that the loop body contains only assignment statements. Although it is technically feasible to decompose a loop statement into partitions, it usually does not make sense to do so, because the loop references tend to be uniform across the iteration space, so there are no benefits to be attained by aligning two such partitions.

We represent data flow constraints in the loop with *dependence relations* [Pug91], and we keep the exact dependence information between each pair of read and write [Fea91, MHL91, Pug92]. Keeping exact dependence information is important in our case, since our transformations can be applied at any granularity of computation. Informally, a read reference in iteration $\mathtt{I}$, $r[\mathtt{I}]$, depends on a write reference in iteration $\mathtt{I}'$, $w[\mathtt{I}']$, iff:

1. $w[\mathtt{I}']$ and $r[\mathtt{I}]$ access the same element of the same array.
2. $w[\mathtt{I}']$ occurs before $r[\mathtt{I}]$ in the sequential execution order.

[2] The reference matrix for an array reference is an integer matrix such that the i^{th} row times I_n is the i^{th} subscript in the array reference.
[3] The bounds matrix represent the inequalities in loop bounds such that $\beta(L) \cdot \mathtt{I_n} \geq \mathtt{0}$. Each row of the bounds matrix represents one inequality in loop bounds. For instance, in a two dimensional loop with iterator (i, j), the bounds of j loop $max(i, 0) \leq j \leq n$ will appear as three rows, (-1 1 0), (0 1 0), and (0 -1 n).

3. $w[\mathtt{I}']$ is the temporally last write to satisfy the above conditions.

This dependence is denoted by

$$w[\mathtt{I}'] \rightarrow r[\mathtt{I}]$$

The set of all such dependences in the loop is referred to as $deprel(r)$. In affine loops, the above dependence can be represented by

$$w[d_{wr} \cdot \mathtt{I}] \rightarrow r[\mathtt{I}] \ \mid \ cond_{wr}(\mathtt{I})$$

where, d_{wr} is a $(n+1) \times (n+1)$ matrix, and $cond_{wr}(\mathtt{I})$ are conditions on $\mathtt{I}$. As will become clear later, the matrix notation above helps simplify the update of these dependence relations. The flow dependence between $\mathtt{A}(\mathtt{i}, \mathtt{j})$ and $\mathtt{A}(\mathtt{i} - \mathtt{1}, \mathtt{j} - \mathtt{1})$ in the running example can be represented by:

$$w_i[(i-1, j-1, 1)^t] \rightarrow r[(i,j,1)^t] \ \mid \ i \geq 1 \wedge j \geq 1 \ \equiv \ w_i[d_1 \cdot (i,j,1)^t] \rightarrow r[(i,j,1)^t] \ \mid \ i \geq 1 \wedge j \geq 1$$

where,

$$d_1 = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix}$$

As an example of a non-uniform dependence consider,

$$w_i[(i-j, j, 1)^t] \rightarrow r[(i,j,1)^t] \ \mid \ i \geq j \ \equiv \ w_i[d_2 \cdot (i,j,1)^t] \rightarrow r[(i,j,1)^t] \ \mid \ i \geq j$$

where

$$d_2 = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Memory related dependences or output and anti-dependences are either eliminated using some standard technique [Fea88, CF87], or are represented similar to flow dependences.

The CDA transformation, among other things, updates the dependence relations. The transformation is legal if all dependences still hold in the new loop; that is if each write occurs before the read in the new loop.

3 Computation Decomposition

Computation Decomposition is the process of dividing the computations of a loop into computations of finer granularity. It is easiest described syntactically. A loop body, $\mathtt{L}$, can, as a first step, be decomposed into its statements $\texttt{S}_1;\ldots;\texttt{S}_\texttt{m}$. CA implicitly performed this decomposition before applying any transformations. CDA, however, goes one step further and decomposes the individual statements of L into statements of finer granularity, down to statements with two operands. This is done by rewriting a statement as a sequence of smaller statements that produce the same result. Consider statement $\texttt{S}_\texttt{j}$:

$$\texttt{S}_\texttt{j} \,:\, w_j \leftarrow f_j(R_j)$$

where w_j denotes the lhs array reference, R_j the rhs references, and f_j is the function computed by the statement. This statement can be replaced by the two statements of finer granularity:

$$\texttt{S}_{\texttt{j}.1} \,:\, t_j \leftarrow f_{j.1}(R_{j.1})$$
$$\texttt{S}_{\texttt{j}.2} \,:\, w_j \leftarrow f_{j.2}(t_j, R_{j.2})$$

if

1. $R_{j.1} \subset R_j$, $R_{j.2} = R_j - R_{j.1}$, and
2. the sequence $\texttt{S}_{\texttt{j}.1}$ followed by $\texttt{S}_{\texttt{j}.2}$ results in writing the same value to w_j as $\texttt{S}_\texttt{j}$ for any R_j.

A number of points are worth noting: First, such a rewriting is clearly always legal. Second, a temporary $\texttt{t}_\texttt{j}$ is used to pass the result of the first statement to the next. Finally, the decomposition introduced a new flow dependence, namely that on $\texttt{t}_\texttt{j}$. Often, it is possible to use w_j directly instead of a temporary. In fact, a temporary is necessary only when $\texttt{S}_{\texttt{j}.2}$ contains a (0) anti-dependence, where w_j and a reference r' in the rhs of $\texttt{S}_{\texttt{j}.2}$, refer to be the same array with the same reference matrix.[4]

Figure 2 shows the result of a decomposition of the loop body of our running example. In this example, Statement $\texttt{S}_1$ is split and replaced by two statements, $\texttt{S}_{1.1}$ and $\texttt{S}_{1.2}$. The result of $\texttt{S}_{1.1}$ is stored in the temporary $\texttt{t(i,j)}$, which is then subsequently used by $\texttt{S}_{1.2}$ Although not a requirement, we have chosen to have the reference matrix of the temporary array the same as that of the lhs of the original statement $\texttt{S}_1$. Note that the bounds of the loop do not change as it is decomposed.

To see how these syntactic decompositions relate to computations, recall that the execution of a loop body is an iteration, and that the set of iterations can be represented by an iteration space (IS), where each point in the iterations space corresponds to an iteration. Similarly, we refer to the execution of a statement in the loop body as a *Statement Execution Instance* (SEI), and the set of all SEI's of a given statement is represented by a *Computation Space* (CS). Each computation space of a statement s, CS(s), is a convex polytope, just as the

[4] If there is such an anti-dependence, then replacing t_j by w_j would violate the anti-dependence, because of the flow dependence introduced with the decomposition.

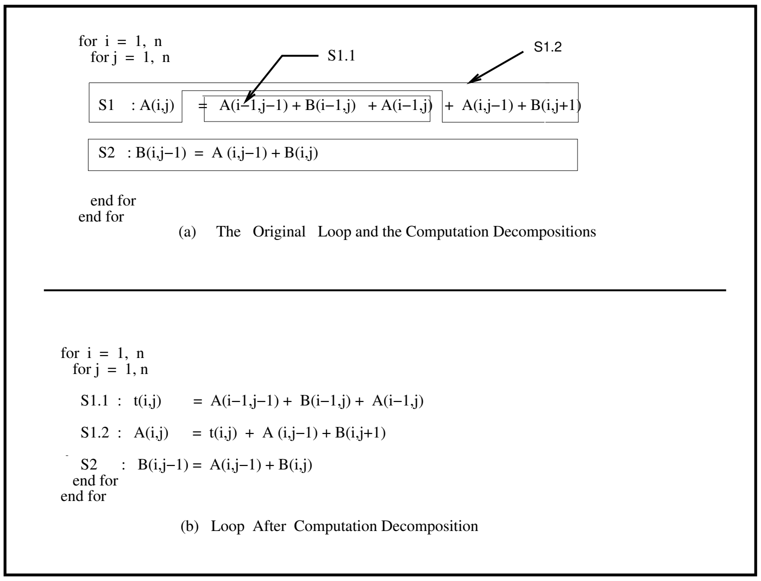


Figure 2: The new loop body after the two computation decompositions.

original iteration space, and its bounds can also be specified by a bounds matrix $\beta(s)$, which is equal to $\beta(L)$ originally.

Formally then, Computation Decomposition first decomposes the iteration space into $\mathtt{m}$ computation spaces, and then subsequently decomposes some of the computation spaces into multiple, finer ones. The decomposition of the running example is shown in Figure 3. The iteration space is first decomposed into two computation spaces, one for each statement, as shown in Figure 3b; that is:

$$IS(\mathtt{L}, \mathtt{I}) \equiv \{CS(\mathtt{S}_1, \mathtt{I}), CS(\mathtt{S}_1, \mathtt{I})\}$$

As a further step, the computation space of $\mathtt{S}_1$ is decomposed into two finer computation spaces, as shown in Figure 3c:

$$CS(\mathtt{S}_1, \mathtt{I}) \equiv \{CS(\mathtt{S}_{1.1}, \mathtt{I}), CS(\mathtt{S}_{1.2}, \mathtt{I})\}$$

Having to introduce temporary arrays may ultimately reduce the degree of locality achievable, may increase the number of references to memory, and and may add to space requirements. However, the flow dependence may add additional constraints after the transformation. Hence, the decision as to how far to decompose and the choice of subexpression to use for the decomposition depends largely on the optimization objective and the overhead introduced by the decomposition. We discuss this in more detail in Section 7.

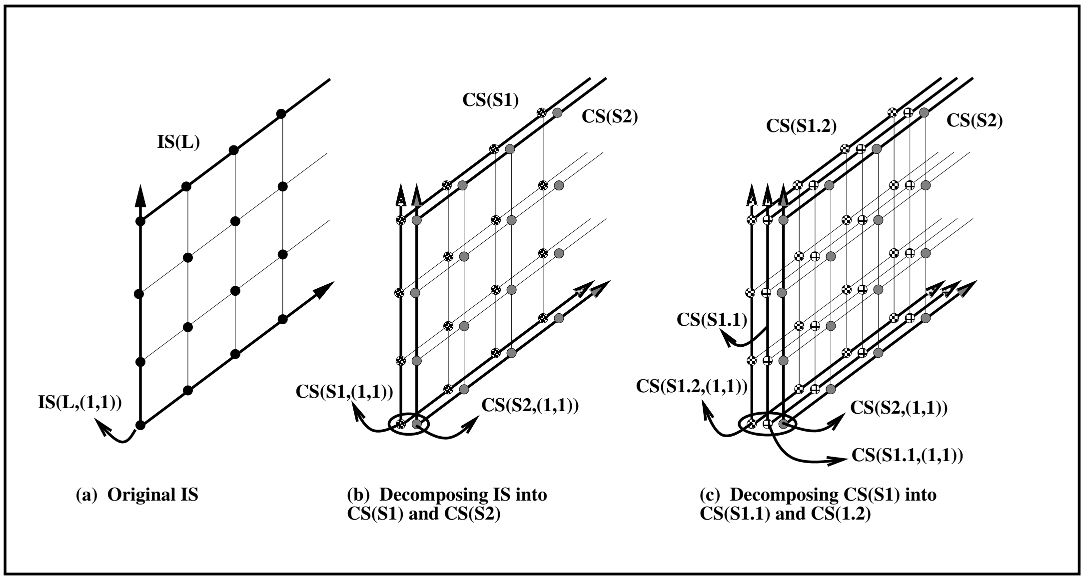


Figure 3: Computation decomposition of the computations in the running example.

4 Transforming the Decomposed Loop

A sequence of decompositions produces a new loop body that can have more statements than the original. After decomposition, it is possible to then apply a separate linear transformation to the computation spaces of each statement, thus mapping the original computation spaces to new computation spaces. Intuitively, the mapping results in a relative movement of the SEIs across iterations. As a result, the new iterations can contain computations that belonged to other iterations in the original IS. It is for this reason that CDA is fundamentally different from traditional linear loop transformations.

Consider the running example again. We can apply three separate transformations, $\mathtt{T}_1$, $\mathtt{T}_2$, and T3, to to $\mathtt{S}_{1.2}$, $\mathtt{S}_2$ and $\mathtt{S}_{1.1}$, respectively. In this example, transformation $\mathtt{T}_1$ is the identity; $\mathtt{T}_2$ moves the $(i,j)^{th}$ instance of $\mathtt{S}_2$ to iteration $(i, j-1)$; T3 moves the $(i,j)^{th}$ instance of S1.1 to iteration $(i-1, j)$ (See Figure 4.) These transformations cause computations $S_{1.1}(i{+}1, j)$, $S_{1.2}(i,j)$ and $S_2(i, j{+}1)$ of the original loop to be executed together in the same iteration. The computation movements are shown in Figure 5.

In order to transform the code to correspond to these transformations, it is necessary to update the reference matrices and dependence relations. This is done with Algorithm Apply-CDA in Figure 6. The reference matrix of r, $refmatrix(r)$ is transformed to be $refmatrix(r) \cdot \mathtt{T}_\mathtt{r}^{-1}$. A dependence relation is updated with the routines $dr\_update\_write(r)$ and $dr\_update\_read(r)$ as follows. Suppose, T_w is appplied to $\mathtt{S}_\mathtt{w}$ and $\mathtt{T}_\mathtt{r}$ is applied to $\mathtt{S}_\mathtt{r}$, and there is a dependence in $deprel(r)$ between a write reference, w, in $\mathtt{S}_\mathtt{w}$ and a read reference,

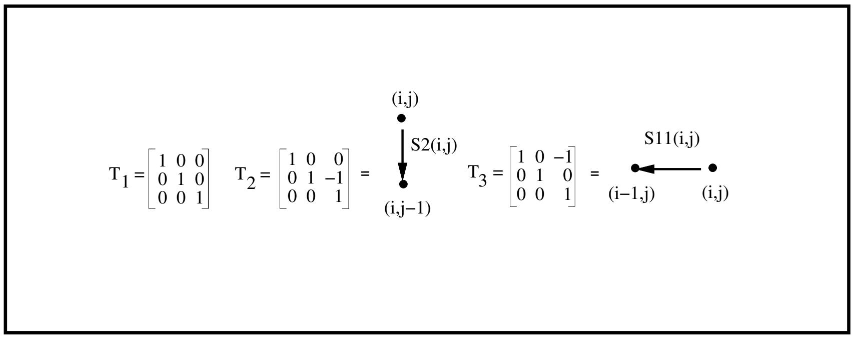


Figure 4: The simple transformations applied to the decomposed running example.

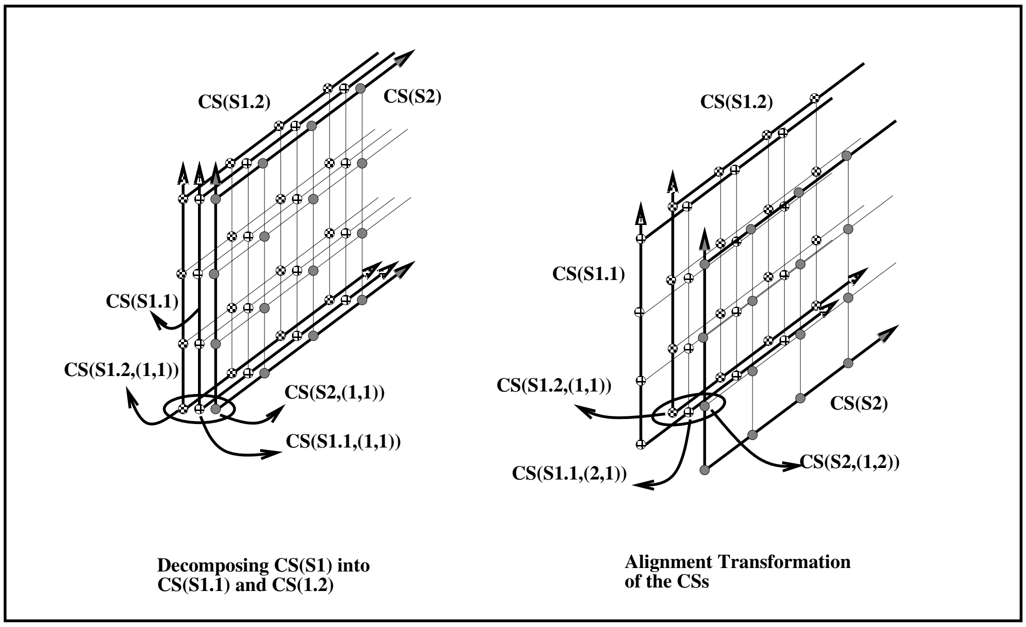


Figure 5: Computation movement in the decomposed computation spaces.

r, in $\mathtt{S_r}$:

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

This dependence is transformed to:

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

which can be rewritten as:

$$w[d_{wr} \cdot T_w \cdot T_r^{-1} \cdot \mathtt{I}] \rightarrow r[\mathtt{I}]$$

Algorithm 1 *: Apply-CDA*

```
      input:  Computation Decompositions $C_D = \langle R_1, \ldots, R_{c_d} \rangle$
      Linear Transforms $T = \{T_1, \ldots, T_{m + c_d}\}$
      output:  Transformed loop
begin
(1)   Decompose computations by $C_D$
(2)   for each $T_i \in T$ do
(3)     for each $r \in references(S_i)$ do
(4)       $refmatrix(r) = refmatrix(r) \cdot T_i^{-1}$
(5)       if $r$ is a read reference then
              dr_update_read(r)
(6)       else /* $r$ is a write reference */
              dr_update_write(r)
          end if
        end for
      end for
(7)   for each $deprel(r) \in deprel$ do
(8)     for each $dep \in deprel(r)$ do
(9)       verify positivity
        end for
      end for
(10)  gen_guard_free_code()
end
```

Figure 6: Algorithm to apply a CDA transformation.

The transformation is legal if the new dependence relation holds true. This can be easily verified if the dependences are uniform and the transformations are simple offsets. In such case, we just have to verify that the last column in the new d_{wr} is lexicographically negative (i.e. write is earlier). If the dependences are non-uniform, then sophisticated techniques that reason with symbolic affine constraints can be used [PW93, Fea91]. There are cases, when the only dependences violated are (0) flow dependences between statements, and textual interchange will be required to make the dependence legal [KS94].

Disregarding the loop bounds for the moment, consider again our running example. Figure 7a shows the transformed loop body. Note that this is an example where it was necessary to interchange statements in order to keep the CDA legal. Because there is no (0) anti-dependence in statement $S_{1.2}$, it is possible to use array A to hold the temporary value of $S_{1.1}$. This is shown in Figure 7b.

The new loop bounds have to account for the transformed computation spaces of every statement in the loop body. The basic idea is to project all computation spaces onto an integer space that becomes the IS of the transformed loop. Because the computation spaces can be transformed differently, the resulting IS can be

```
for  i  =  l1,  u1
  for j  =  l2, u2

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

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

    S1.1:     t(i+1,j)  =  A(i,j−1) + B(i,j) + A(i,j)

  end for
end for
```

(a) Loop After Applying CDA: T1 to S1.2, T2 to S2, T3 to S1.1

```
for  i  =  l1,  u1
  for j  =  l2, u2

    S1.2 :   A(i,j)      =  A(i,j) + A(i,j−1) + B(i,j+1)

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

    S1.1 :  A(i+1,j)  =  A(i,j−1) + B(i,j) + A(i,j)

  end for
end for
```

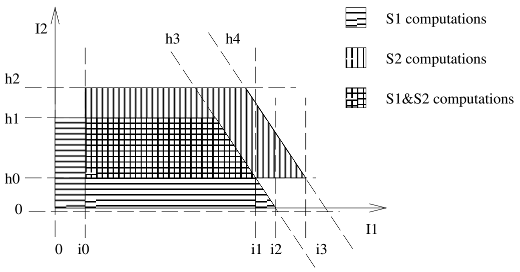
(b) Loop After Eliminating the Temporary array " t "

Figure 7: The running example after CDA transformations.

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 IS, and then generate a *perfect* nest that traverses this convex hull. However, 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 good performance, but a perfect loop may be desirable in some cases in order to avoid non-vector communications, or to avoid the loop overheads. An algorithm to generate a guard-free loop for T when all the statements require the same loop stride is described in [KS94]. Kelly et al. also developed an algorithm to generate code for general linear transformations but with conditionals [KPR94]. These algorithms reduce to the algorithm developed by Torres et al. when the transformations are simple offsets corresponding to loop alignments [TALV93].

We illustrate a typical way to generate guard_free_code with the aid of Figure 8a. The full details can be found in [KS94]. The basic idea of the algorithm is to partition the new IS into segments that contain iterations with $\mathtt{S}_1$ computation only, or those that contain $\mathtt{S}_2$ computations only, or those with both computations. 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 Further segments are defined for the ranges $[i_1 + 1, i_2]$ and $[i_2 + 1, i_3]$.



(a) Transformed Computation Spaces

```
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
```

(b) Code for I1 range [i0,i1]

Figure 8: Segmenting the union.

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 8b shows the code generated for the three segments defined in the range $[i_0, i_1]$.

5 Applications of CDA

The choice of computation decomposition and the choice of subsequent transformations are closely related. They are directly influenced by the trade-offs between the optimization objectives and the overheads introduced by the transformation.

In this section, we list a number of optimization objectives and describe how CDA can be used to achieve them. Later, in Section 7, the results of our experiments will show that the application of even simple CDA transformations can result in enormous performance gains.

The basic idea behind many of the CDA transformations described below is to align the loop references relative to one another in order to improve performance. This is something that only CDA is capable of doing, and is typically done in two steps. First, the loop is decomposed so that all references in any statement have similar indexing patterns. Then in the second step, all statements with identical or similar indexing can be aligned to statements with different indexing. For example, all references in $\mathtt{S}_1$ with $(i-1,*)$ indexing are segregated into a statement $\mathtt{S}_{1.1}$, so that it can later (possibly together with other statements) be aligned to statements having $(i,*)$ indexing.

In the following, we consider the objectives data alignment, elimination of ownership tests, stencil minimization, access normalization, and cache locality. Ofcourse, whenver one objective is pursued, other objectives may be affected due to side effects.

Data alignment

A data alignment transformation, A_t, on array `A` referenced by r causes $refmatrix(r)$ to be changed to $A_t \cdot refmatrix(r)$. A CDA T_r can be used instead of the data alignment whenever the solution, T_r, to

$$A_t \cdot refmatrix(r) = refmatrix(r) \cdot T_r^{-1}$$

is a legal CDA.[5] Hence, CDA can be used as the *dual* of data alignment in many circumstances. An algorithm to find data alignment transformations, typically identifies a data alignment "constraint", and derives a data alignment transformation that satisfies it [KN90, LC91, OH92]. We believe that by using both data alignment and CDA transformations together, one can satisfy more constraints than existing algorithms do. Data alignment and CDA have their own advatanges and drawbacks. Data alignment does not affect dependences; and satisfies a constraint between a pair of arrays without affecting others. But, it modifies the references to the array globally, possibly undoing alignments for some other loop; realignment of data is usually expensive. CDA transformations on the other hand are local to the loop; and do not require expensive data layout changes. However, CDA changes dependences, and can affect constraints due to references in the same computation decomposition.

Elimination of ownership tests

The elimination of ownership tests results in a better 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 test is to ensure that all SEIs of an iteration are to be executed by the same processor. With owner-computes rule, this is achieved by collocating the lhs references of all statements of the loop body [TALV93, KS94]. Even then, it may be more efficient to execute some subexpressions on another processor, say because it owns most of the data required for the computation. CDA can effect this modification to the owner-computes rule. A computation decomposition can segregate the subexpressions, and align them with respect to the orignal statemenet appropriately. For instance, we modified the computation rule for $\mathtt{S}_1$ by moving computations of $\mathtt{S}_{1.1}$.

In fact, CDA can be used to implement a class of flexible computation rules with the aid of owner-computes.

[5] Assuming appropriate padding of A_t and T_r when the dimension of the array is different from n.

Reducing communication for a reference stencil

A communication optimal distribution of an array depends on its reference *stencil* [AH91, IT88, AI91]. A CDA transformation 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.

Matching loop structure to data distribution

Access normalization [LP92] is a linear loop transformation that matches the references to the data distributions. For instance, when $\mathtt{A}$ is distributed by row, then it is preferable to transform the loop so that the reference patterns are $\mathtt{A}(\mathtt{i}, *)$, instead of say, $\mathtt{A}(\mathtt{i} - \mathtt{j}, *)$. In contrast to linear loop transformations, CDA can perform partial access normalization by segregating the references that need similar normalization into groups, and applying a separate transformations to each group.

Improving Cache locality

Linear loop transformations and tiling (or blocking) [LRW91] improve cache locality by maximizing the reuse of the data in the cache before it is displaced. CDA can enhance the effectiveness of these techniques. For instance, a simple loop interchange with iterators (i, j) can make the array reference $\mathtt{A}(\mathtt{j}, \mathtt{i})$ utilize all elements in a cache line. But this would not help if there is a $\mathtt{A}(\mathtt{i}, \mathtt{j})$ reference as well. With CDA, we can potentially apply interchange selectively and achieve locality for both types of references.

Blocking is an effective technique but introduces loop overheads, that in some cases negate any improvements due to data reuse. Some CDA may be able to improve locality by selectively changing the reference patterns without the overhead introduced by tiling. The mg program discussed in Section 7 illustrates this.

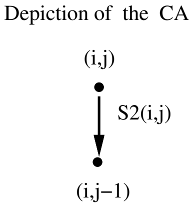
Some loops can not be blocked as they are, in which case a loop skew is required to enable blocking. As the running example illustrates CDA and CA may make dependences fully-permutable without the need for a skew. This results in simpler loop bounds, and data accesses along cache lines.

Sometimes, a given reference pattern will result in poor cache locality due to cache interference. CDA can selectively "move" references so that they no longer interfere. The calc1 program of SWM in SPEC benchmarks suite discussed in Section 7 illustrates this.

Reducing register pressure

CDA can be applied to reduce the number of distinct elements accessed in an iteration. This can reduce the number load, and potentially can reduce register pressure.

```
for i = l1, u1
  for j = l2, u2
    S1 : A(i,j)   = A(i-1,j-1) + B(i-1,j) + A(i,j-1) + A(i-1,j) B(i,j+1)
    S2 : B(i,j)   = A (i,j) + B(i,j+1)
  end for
end for
```



Loop After Applying CA only: T1 to S1 and T2 to S2

Figure 9: The running example after applying only CA.

6 The running example revisited

In this section, we identify some of the performance advantages of applying CDA on our running example. We also compare the CDA transformed loop with that obtained from a comparable CA transformation. The comparable CA aligns statements $\mathtt{S}_2$ to $\mathtt{S}_1$ by applying transformation $\mathtt{T}_2$ to $\mathtt{S}_2$, as shown in Figure 9.

The running example was carefully chosen so that one simple CDA exhibits many interesting properties. Some differences between the original loop, the CDA transformed loop and the CA transformed loop are:

- **Blocking** : Transformation $\mathtt{T}_2$ applied to CS($\mathtt{S}_2$) in both the CDA and the CA eliminates the $(1, -1)$ dependence so that the dependences become fully permutable. This makes the loop blockable as is, with indexing and loop bounds simpler than if the loop had first been skewed.

- **Dependence structure** : The new loop accesses A and B in a fundamentally different way, with the dependences:

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

  transformed to:

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

  In comparison, the CA transformation produces the following dependences:

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

  where the (1,1) dependence can result in non-vectorizable communication for some distributions.

- **Efficiency of SPMD code** : The original loop requires ownership tests, unless $\mathtt{A(i,j)}$ is aligned to $\mathtt{B(i,j-1)}$. The CDA and CA transformations both eliminated the ownership tests without changing any existing data alignments. With $\mathtt{t}(\mathtt{i}+1,\mathtt{j})$ aligned to $\mathtt{A}(\mathtt{i},\mathtt{j})$, all SEI of an iteration in both the CDA and CA versions are owned by the same processor. This allows existing alignments to contine to exist, yet still produce a loop without any ownership tests.

- **Constraints on data alignment** : CDA reduces the number of accesses that constrain alignment, thus reducing the amount of communication required. Assuming that $\mathtt{B}(\mathtt{i},\mathtt{j})$ and $\mathtt{t}(\mathtt{i}+1,\mathtt{j})$ are aligned to $\mathtt{A}(\mathtt{i},\mathtt{j})$, and assuming the use of the owner computes rule, then the original loop accesses elements `A(i-1,j-1)`, `A(i-1,j)` and `A(i,j-1)` in iteration (i, j), while it only accesses `A(i,j-1)` and `t(i,j)` in the new loop. Similarly, the original loop accesses elements `B(i,j)`, `B(i-1,j)` and `B(i,j+1)`, but only accesses `B(i,j+1)` in the new loop. In comparison, a CA transformed loop would still need to access a total of 6 elements.
- **Cache locality** : The CDA transformed loop has better cache locality. The transformation reduced the *reference window* [GJG88] of `B` along the i^{th} dimension from $2n$ to n, thus halving `B`'s cache context. This is the result of bringing the group reuse of `B` across the i loop into the same iteration.
- **Number of memory accesses** : In the sequential context, the transformed loop has 3 loads and 3 stores compared to 6 loads and 2 stores with the original loop. The savings due to reduced loads can be significant. (The savings for the running example improves its execution time on a Sparc-20 by 15%.)

7 Experimental results

In this section, we present the results of simple experiments to give an indication of the type of performance gains achievable by applying CDA. The experiments are designed primarily to illustrate individual performance aspects of CDA, and not to show absolute performance of any particular application. We conducted the experiments on a KSR1 multiprocessor with a COMA architecture [KSR91], on a cluster of RS6000 workstations, and on RS6000, Sparc-10 and Sparc-20 workstations.

7.1 NAS mg

The mg program of the NAS benchmark suite is a multigrid solver that computes a three dimensional potential field. The $psinv$ routine in mg demonstrates how a simple CDA can improve the cache hit rate substantially. The source for original and CDA transformed $psinv$ appears in the appendix.

The $psinv$ subroutine consists of a three dimensional loop, with iterators i, j and k that computes $U = U + CR$, where U and R are three dimensional array and C is a four element one dimensional array. The loop is parallelizable in all three dimensions, and has simple subscripts with ± 1 offsets in all three dimensions. It is not possible to apply CA directly, because the loop body only has a single statement. Also, tiling would not help much to improve cache hit rate, because each element of U is accessed in only one iteration, and element $R(i, j, k)$ is accessed only in a few iterations, namely iterations $(i + c, j + c, k + c)$ where c is 0,1,2.

CDA Transformation

The objective in choosing a CDA transformation for mg was to eliminate the $R(i-1, *, *)$ references in order to reduce the size of R's reference window, and thus improve the cache hit rate. Hence, we decomposed the statement into a substatement $\mathtt{S}_{1.1}$ that only had references of the form $R(i - 1, *, *)$ and into substatement $\mathtt{S}_{1.2}$ that contained the rest of the original statement. We included in $\mathtt{S}_{1.1}$ the reference $U(i, j, k)$ so that we could eliminate the need for a temporary. We then aligned $\mathtt{S}_{1.1}(\mathtt{i} + 1, \mathtt{j}, \mathtt{k})$ to $\mathtt{S}_{1.2}(\mathtt{i}, \mathtt{j}, \mathtt{k})$ so that references $R(i - 1, *, *)$ became $R(i, *, *)$. This transformation introduced a new $(1, 0, 0)$ dependence, restricting parallelization to only along the j dimension. It increased the size of the reference window for U from n^2 to $2n^2$. But it decreased the size of the reference window for R from $3n^2$ to $2n^2$, and with it the amount of self interference in the cache. The resulting savings outweigh the increase cache context of U.

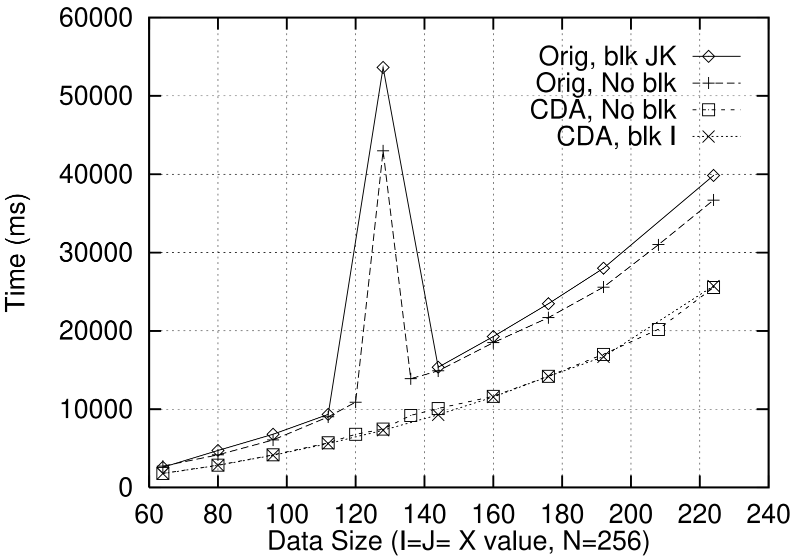


Figure 10: Execution time of the mg program on the Sparc-20 with a 1MB cache.

Sequential Performance of transformed mg

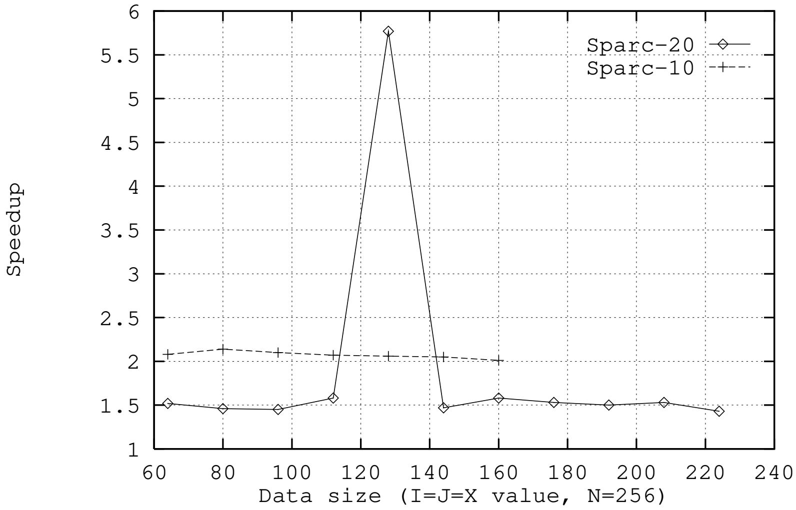


Figure 11: Improvement on Sparc-10 and Sparc-20 relative to respective original version of mg.

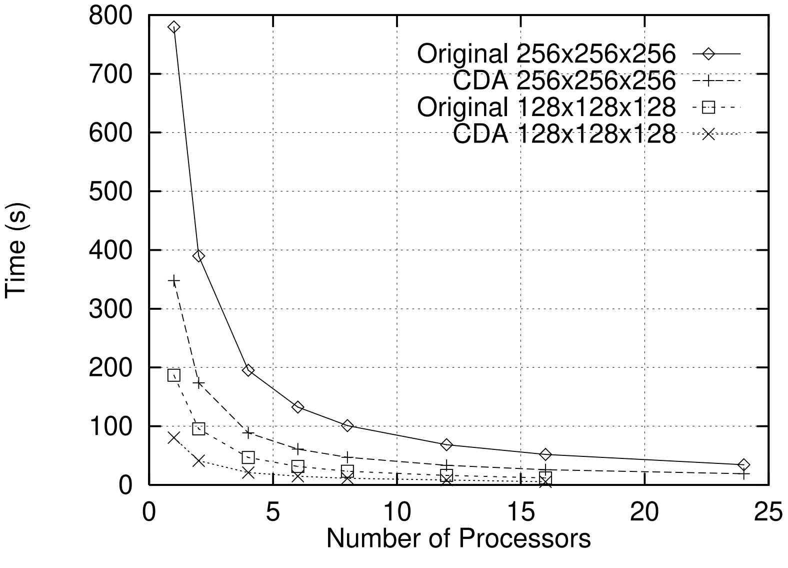


Figure 12: Execution time of the mg program on the KSR1.

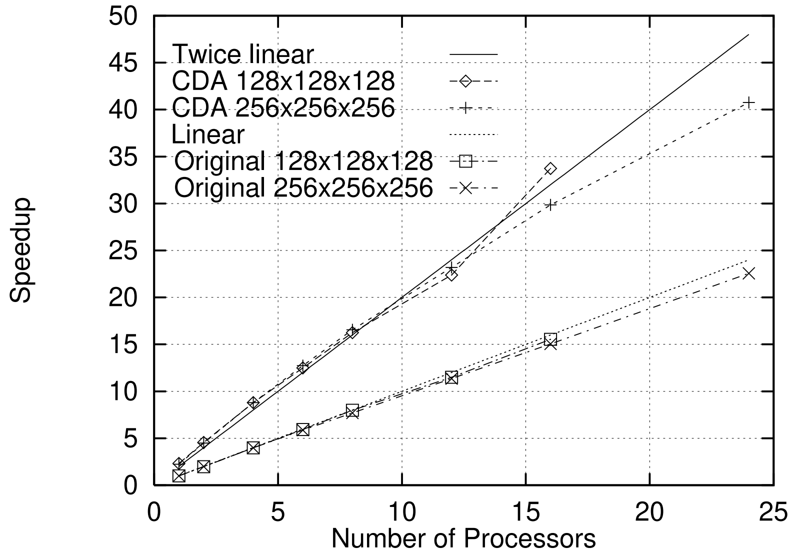


Figure 13: Speedup of the mg program, CDA relative to the original code on the KSR1.

Figure 10 shows the execution time for mg when run on a Sparc-20 system, with varying data sizes. Both the original and the CDA transformed loops were run with and without tiling. it is evident that the transformed loop runs 33% faster than the original. Hence, it is clear that the savings in the reduction of R's reference window owuweighs the overhead in the increase of U's reference window. It is evident that tiling in this introduces more overhead than the savings due to reuse.

Another interesting point is that CDA "smoothens" the peaks in the execution times that occur due to self interference on R. In the figure, such a peak is clearly visible at data size $128 \times 128 \times 256$. With this data size, CDA improved performance by a factor of close to 6.

We ran the same experiments on a Sparc-10 system, where the improvements werer even larger. Figure 11, compares the improvements obtained on both systems.

Parallel mg on the KSR1

We implemented both the original and the CDA versions of this code in C using the NUMACROS package [LS93] on the KSR1. We parallelized the CDA version of mg only along the j iterator, and we parallelized the original version along both the i and j iterators. Figure 12 compares the performance of these versions; the mapping chosen is the one that gives the best execution time for the original loop. While the original version exhibits close to linear speedup, the CDA version speeds up nearly twice as fast. This is shown more clearly in Figure 13 that depicts the speedup for the different versions.

7.2 SLIA

SLIA is a synthetic loop in Fortran which we use to demonstrate two effects. First, CDA's ability to be equivalent to data alignment. And second, CDA's ability to move code, making it easier for instruction schedules to exploit the processor pipelining. The original loop looks as follows:

```
for  j = 2, n
  for  i = 2, n
    A(i, j) = A(i - 1, j) + A(i - 1, j - 1) + B(i - 1, j) + B(i - 1, j - 1) + A(i, j - 1)
        +B(i, j - 1) + C(i - 1, j) + C(i, j - 1) + C(i - 1, j - 1)
  end for
end for
```

In order to reduce the data alignment constraints, we chose a CDA to decompose the statement into a substatement that has references with $(*, j-1)$ indexing, and another substatement with the rest of the references. We subsequently, aligned the two new statements so that $(*, j-1)$ indexing became $(*, j)$. Array `A` itself was used to pass the temporary results from one statement to the other. The CDA transformed loop became:

```
for  i = 2, n
  A(i, 2) = A(i, 1) + A(i - 1, 1) + B(i, 1) + B(i - 1, 1) + C(i, 1) + C(i - 1, 1)
end for

for  j = 2, n - 1
  for  i = 2, n
    A(i, j) = A(i, j) + A(i - 1, j) + B(i - 1, j) + C(i - 1, j)
    A(i, j + 1) = A(i, j) + A(i - 1, j) + B(i, j) + B(i - 1, j) + C(i, j) + C(i - 1, j)
  end for
end for

for  i = 2, n
  A(i, n) = A(i, n) + A(i - 1, n) + B(i - 1, n) + C(i - 1, n)
end for
```

We then used KAP on the KSR1 to tile and schedule the tiles in a wavefront, and ran the resulting code using the Presto runtime environment. Figure 14 shows that the CDA improved performance between 40% and 60%. This improvement has two sources. Firstly, the number of remote accesses is reduced, because of the elimination of the $j-1$ references. The original loop needed 6 remote accesses (3 cache line accesses) at the j axis boundary of a tile, whereas the transformed loop needs only one such access.

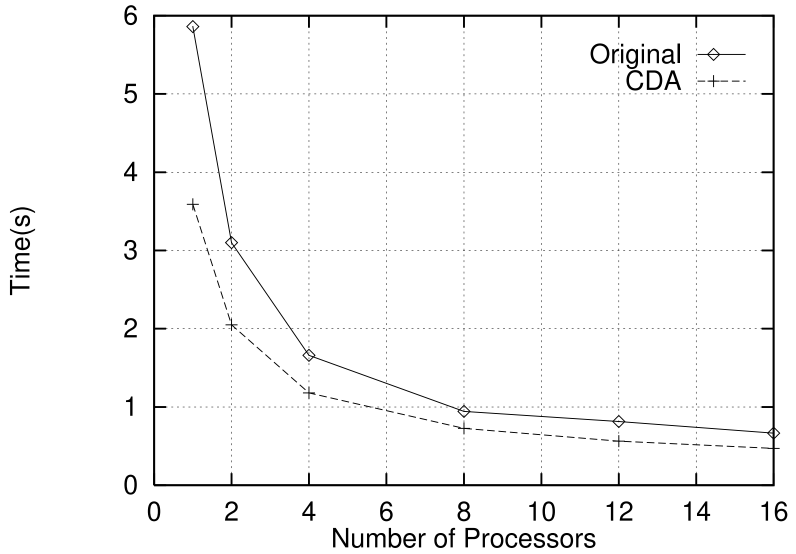


Figure 14: Execution time of SLIA loop on the KSR1.

The second source of improvement comes from improved instruction scheduling made possible by the CDA. The original loop resulted in chain of dependent instructions that could not be pipelined. The CDA broke the chain in two, allowing instructions of the two chains to be interleaved. A linear transformation could not have achieved this, since it can transform loops only at the granularity of the entire loop body. It would also not have been possible to apply a CA, since the original loop has only one statement.

7.3 SPEC 078.SWM256

SWM256 is the shallow water program of the SPEC benchmark suite. We chose its $calc1$ subroutine to demonstrate CDA's ability to eliminate ownership tests. Calc1 has 4 statements, with lhs references to $\mathtt{CU}(\mathtt{i}+1,\mathtt{j})$, $\mathtt{CV}(\mathtt{i},\mathtt{j}+1)$, $\mathtt{Z}(\mathtt{i}+1,\mathtt{j}+1)$ and $\mathtt{H}(\mathtt{i},\mathtt{j})$. The loop requires ownership tests unless $\mathtt{CU}(\mathtt{i}+1,\mathtt{j})$, $\mathtt{CV}(\mathtt{i},\mathtt{j}+1)$ and $\mathtt{Z}(\mathtt{i}+1,\mathtt{j}+1)$ are aligned to $\mathtt{H}(\mathtt{i},\mathtt{j})$. Hence, we simply aligned the statements, so that their lhs references were all of the form (i,j). The transformed version does not require ownership tests, and does not require any additional data alignments. Since the loop does not require computation decomposition, CDA here is equivalent to CA. The original and CDA version of the code are shown in the appendix.

We ran experiments on the KSR1 and a cluster of 4 RS6000 workstions. On the KSR1, we ran the C version of cacl1 parallelized using the NUMACROS package. Figure 15 shows the improvements obtained by CDA, which are about 17%. Because of the single address space and relatively low remote access costs on the KSR1 the data ownerships do not affect performance much. We expect data ownership to be more significant on distributed memory machines. It is interesting to note from shared memory point of view that the transformation eliminated the cache interference between $u(i, j)$ and $v(i + 1, j + 1)$ references.

In the second experiment, we aligned $calc2$ routine also in a similar way, and compiled all loops of SWM256 with IBM's prototype HPF compiler. The data were distributed by (BLOCK,BLOCK) because the loops are fully parallel. We ran the resulting code on a cluster of four RS6000 workstations. Due to the elimination of ownership tests, the the CDA version took 10.6s compared to 12.1s by the original loops.

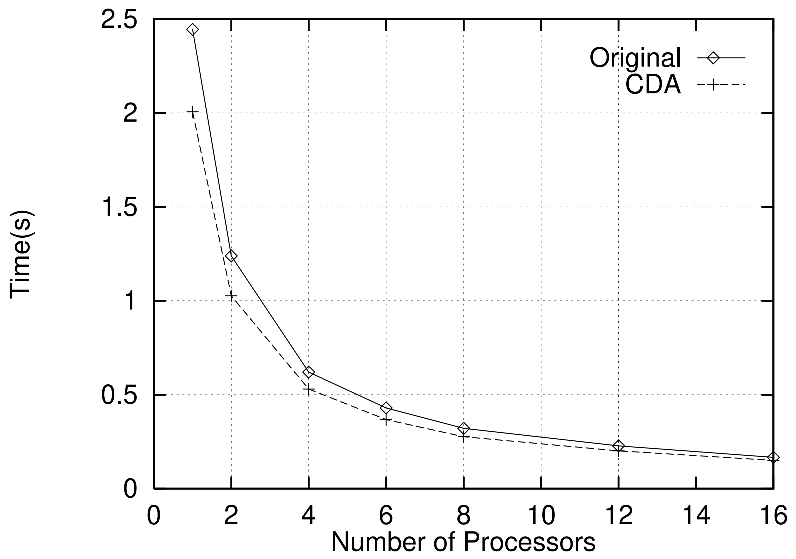


Figure 15: Execution time of the calc1 procedure of 078.swm256 of the SPEC benchmark on the KSR1.

7.4 Wanal

Wanal is a wave equation solver that is part of the Riceps benchmark suite [Li92]. We chose the following three dimensional loop from this code:

```
for  K = 0, 1
  for  J = M + 1, 0, −1
    for  I = M + 1, 0, −1
      S1 : EL(2 ∗ I − 1, 2 ∗ J, K) = (EL(2 ∗ I, 2 ∗ J, K) + EL(2 ∗ I − 2, 2 ∗ J, K))/2
      S2 : EL(2 ∗ J, 2 ∗ I − 1, K) = (EL(2 ∗ J, 2 ∗ I, K) + EL(2 ∗ J, 2 ∗ I − 2, K))/2
    end for
  end for
end for
```

To get good locality (assuming row major allocation), it is necessary to interchange J and I for statement $\mathtt{S}_2$, but not for $\mathtt{S}_1$. The standard technique would be to perform a loop distribution creating two loops, interchange I and J in the second loop, and fuse the loop back together again. CDA can directly interchange the computation space of $\mathtt{S}_2$ alone. The CDa transformed loop is:

```
for  K = 0, 1
  for  J = M + 1, 0, -1
    for  I = M + 1, 0, -1
      S1 : EL(2 * I - 1, 2 * J, K) = (EL(2 * I, 2 * J, K) + EL(2 * I - 2, 2 * J, K))/2
      S2 : EL(2 * I, 2 * J - 1, K) = (EL(2 * I, 2 * J, K) + EL(2 * I, 2 * J - 2, K))/2
    end for
  end for
end for
```

Since the loop needs to be decomposed at statement level only, CA could be used. Figure 16 shows the results obtained on the KSR1. Because, the loop from the benchmark has problem size $M = 130$, which is very small, we have increased the size artificially to 1024 and 1600, in order to be able to accurately measure the execution times. The figure shows that with $M = 1600$, improvements in the range of 80% to 100% are obtained.

With $M = 2050$, the original loop resulted in cache thrashing and an execution time of 66.8s on 12 processors.The CDA version of the code required only 2.9s for the same size.

In a separate experiment we compiled HPF versions of the original and the transformed loops with (BLOCK,*,*) distribution by the IBM prototype HPF compiler. On the RS6000 four workstation cluster CDA took 3.1s as against 5.4s by the original for $M = 1024$.

8 Concluding Remarks

We presented a generalized linear loop transformation framework, CDA, that subsumes CA and traditional linear loop transformations. Advantages of the framework stem from the fact that CDA can transform computations of arbitrary granularities independently. CDA alters the makeup of the iterations and in doing so may move computations across iteration boundaries or spread them across several iterations. This feature adds a new dimension to performance optimization on high performance computing platforms, and thus extends significantly the existing frameworks both in power and flexibility. The experimental results we presented clearly

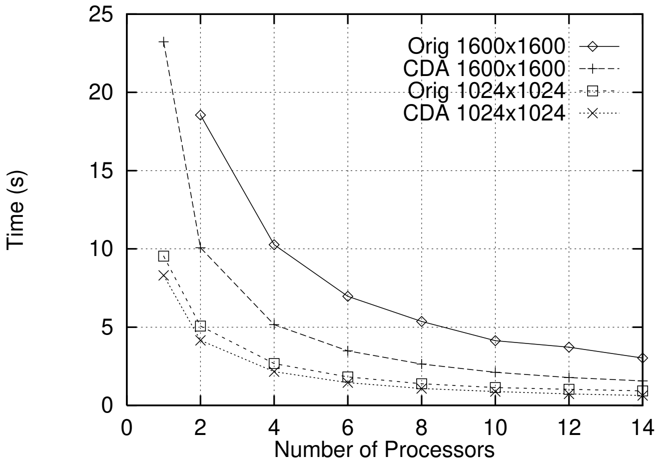


Figure 16: Execution time of the WANAL loop on the KSR1 (The sequential time for M=1600 is 122.4s which is not shown for clarity).

show the promise of being able to align computations at arbitrarily small granularities.

We are currently evaluating heuristics to derive CDA transformations at appropriate granularity given specific optimization objectives. We are working on expressing the relationship between data alignment, CDA and array padding, and devising an algorithm that uses all three together for better performance. We believe CDA can be very effective in minimizing global constraints by manipulating computation structures at the loop level. We are pursuing the integration of CDA techniques into global optimizers. A CDA transformation can be viewed as an implementation of a class of computation rules, using the simplicity of the owner computes rule; we are characterizing this class of computation rules. Instruction schedulers tend to produce better schedules with certain instruction dependencies than others; we are investigating heuristics to find CDA transformations that avoid linear chains in the graph.

References

[AH91] S. G. Abraham and D. E. Hudak. 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.

[AC87] Randy Allen, David Callahan, and Ken Kennedy. 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.

[AI91] C. Ancourt and F. Irigoin. 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.

[AL93] J. Anderson and M. Lam. 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.

[Ban90] Utpal Banerjee. Unimodular transformations of double loops. In *Proceedings of Third Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.

[Ban93] Utpal Banerjee. *Loop Transformations for Restructuring Compilers*. Kluwer Academic Publishers, 1993.

[Ban94] Utpal Banerjee. *Loop Parallelization*. Kluwer Academic Publishers, 1994.

[CFR93] J. Collard, P. Feautrier, and T. Risset. Construction of do loops from systems of affine constraints. Technical report, Research Report 93-15, Ecole Normale Superieure de Lyon, France, 1993.

[CF87] R. Cytron and J. Ferrante. What's in a name? In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 19–27, 1987.

[Dow90] M.L. Dowling. Optimum code parallelization using unimodular transformations. *Parallel Computing*, 16:155–171, 1990.

[Fea88] P. Feautrier. Array expansion. In *Proceedings of the 1988 International Conference on Supercomputing*, 1988.

[Fea91] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, volume 20(1), 1991.

[HPF93] H. Forum. Hpf: High performance fortran language specification. Technical report, HPF Forum, 1993.

[GJG88] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5:587–616, 1988.

[HKKKT91] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the fortran d programming system. Technical Report CRPC-TR91121, Dept of computer Science, Rice University, 1991.

[IT88] F. Irigoin and R. Triolet. Supernode partitioning. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 319–329, San Diego, CA, 1988.

[KKB91] K.G. Kumar, D. Kulkarni, and A. Basu. Generalized unimodular loop transformations for distributed memory multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, Chicago, MI, July 1991.

[KKB92] K.G. Kumar, D. Kulkarni, and A. Basu. 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.

[KKBP91] D. Kulkarni, K.G. Kumar, A. Basu, and A. Paulraj. Loop partitioning for distributed memory multiprocessors as unimodular transformations. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[KKB92] K. Kumar, D. Kulkarni, and A. Basu. Mapping nested loops on hierarchical parallel machines using unimodular transformations. *Journal of Parallel and Distributed Computing*, (under revision).

[KN90] K. Knobe and V. Natarajan. Data optimization: Minimizing residual interprocessor data motion on simd machines. In *Proceedings of the Symposium on frontiers of massively parallel computation*, pages 416–423, 1990.

[KP92] W. Kelly and W. Pugh. A framework for unifying reordering transformations. Technical Report UMIACS-TR-92-126, University of Maryland, 1992.

[KPR94] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. Technical Report UMIACS-TR-94-87, University of Maryland, 1994.

[KS94] D. Kulkarni and M. Stumm. 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.

[KS95] D. Kulkarni and M. Stumm. Computational decomposition and alignment and its applications. Technical Report In preparation, Computer Systems Research Institute, University of Toronto, January 1995.

[KSR91] Kendall Square Research (KSR). *KSR1 Principles of Operation*. Waltham, MA, 1991.

[LC91] J. Li and M. Chen. The data alignment phase in compiling programs for distributed memory machines. *Journal of parallel and distributed computing*, 13:213–221, 1991.

[Li92] C.H. Li. Program wanal1. ftp ftp.cs.rice.edu, Rice University, 1992.

[LP92] W. Li and K. Pingali. 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.

[LRW91] M.S. Lam, E.E. Rothberg, and M.E. Wolf. The cache performance and optimizations of block algorithms. In *4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Santa Clara, CA, April 1991.

[LS93] Hui Li and Kenneth C. Sevcik. NUMACROS: Data Parallel Programming on NUMA Multiprocessors. In *Proceedings of Fourth Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, pages 247–263, September 1993.

[MHL91] D.E. Maydan, J.L. Hennessy, and M.S. Lam. Efficient and exact data dependence analysis. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, volume 26, pages 1–14, Toronto, Ontario, Canada, 1991.

[OH92] M.F.P O'Boyle and G.A. Hedayat. Data alignment: Transformations to reduce communication on distributed memory architectures. In *Proceedings of the Scalable High Performance Computing Conference, IEE Press*, Williamsburg, 1992.

[Pad79] D. Padua. Multiprocessors: Discussion of some theoretical and practical problems. Phd thesis, University of Illinois, Urbana-Champaign, 1979.

[Pug91] W. Pugh. Uniform techniques for loop optimization. In *International Conference on Supercomputing*, Cologne, Germany, 1991.

[Pug92] W. Pugh. A practical algorithm for exact array dependence analysis. In *Communications of the ACM*, volume 35, pages 102–114, 1992.

[PW86] D. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[PW93] W. Pugh and D. Wonnacott. An exact method for analysis of value-based array data dependences. Technical Report CS-TR-3196, University of Maryland, 1993.

[Ram92] J. Ramanujam. Non-singular transformations of nested loops. In *Supercomputing 92*, pages 214–223, 1992.

[Sim94] B. Simmons. Personal communication. 1994.

[TALV93] J. Torres, E. Ayguade, J. Labarta, and M. Valero. Align and distribute-based linear loop transformations. In *Proceedings of Sixth Workshop on Programming Languages and Compilers for Parallel Computing*, 1993.

[WL90] M.E. Wolf and M.S. Lam. An algorithmic approach to compound loop transformation. In *Proceedings of Third Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.

[Wol90] Michael Wolfe. *Optimizing supercompilers for supercomputer*. The MIT Press, 1990.

[Wol92] M.E. Wolf. Improving locality and parallelism in nested loops. Phd thesis, Stanford University, 1992.

A Original and Transformed Programs

```
for(i = 1; i < I - 1; i++)
  for(j = 1; j < J - 1; j++)
    for(k = 1; k < N - 1; k++){
      U[i][j][k] = U[i][j][k]
        +c[0] * (R[i][j][k])
        +c[1] * (R[i - 1][j][k] + R[i + 1][j][k]
        +R[i][j - 1][k] + R[i][j + 1][k]
        +R[i][j][k - 1] + R[i][j][k + 1])
        +c[2] * (R[i - 1][j - 1][k] + R[i + 1][j - 1][k]
        +R[i - 1][j + 1][k] + R[i + 1][j + 1][k]
        +R[i][j - 1][k - 1] + R[i][j + 1][k - 1]
        +R[i][j - 1][k + 1] + R[i][j + 1][k + 1]
        +R[i - 1][j][k - 1] + R[i - 1][j][k + 1]
        +R[i + 1][j][k - 1] + R[i + 1][j][k + 1])
        +c[3] * (R[i - 1][j - 1][k - 1] + R[i + 1][j - 1][k - 1]
        +R[i - 1][j + 1][k - 1] + R[i + 1][j + 1][k - 1]
        +R[i - 1][j - 1][k + 1] + R[i + 1][j - 1][k + 1]
        +R[i - 1][j + 1][k + 1] + R[i + 1][j + 1][k + 1]);

      }
```

Figure 17: Original MG

```
for(j = 1; j < J - 1; j++)
  for(k = 1; k < N - 1; k++){
    U[1][j][k] = U[1][j][k]+
      c[1] * (R[0][j][k])
      +c[2] * (R[0][j - 1][k] + R[0][j + 1][k]
      +R[0][j][k - 1] + R[0][j][k + 1])
      +c[3] * (R[0][j - 1][k - 1] + R[0][j + 1][k - 1]
      +R[0][j - 1][k + 1] + R[0][j + 1][k + 1]);
      }

for(i = 1; i < I - 2; i++)
  for(j = 1; j < J - 1; j++)
    for(k = 1; k < N - 1; k++){
      U[i + 1][j][k] = U[i + 1][j][k]+
        c[1] * (R[i][j][k])
        +c[2] * (R[i][j - 1][k] + R[i][j + 1][k]
        +R[i][j][k - 1] + R[i][j][k + 1])
        +c[3] * (R[i][j - 1][k - 1] + R[i][j + 1][k - 1]
        +R[i][j - 1][k + 1] + R[i][j + 1][k + 1]);
        +R[i][j - 1][k] + R[i][j + 1][k]
        +R[i][j][k - 1] + R[i][j][k + 1])

      U[i][j][k] = U[i][j][k] + c[0] * (R[i][j][k])
        +c[1] * (R[i + 1][j][k] + R[i][j - 1][k]
        +R[i][j + 1][k] + R[i][j][k - 1] + R[i][j][k + 1])
        +c[2] * (R[i + 1][j - 1][k]
        +R[i + 1][j + 1][k]
        +R[i][j - 1][k - 1] + R[i][j + 1][k - 1]
        +R[i][j - 1][k + 1] + R[i][j + 1][k + 1]
        +R[i + 1][j][k - 1] + R[i + 1][j][k + 1])
        +c[3] * (R[i + 1][j - 1][k - 1]
        +R[i + 1][j + 1][k - 1]
        +R[i + 1][j - 1][k + 1]
        +R[i + 1][j + 1][k + 1]);
    }

for(j = 1; j < J - 1; j++)
  for(k = 1; k < N - 1; k++){
    U[I - 2][j][k] = U[I - 2][j][k] + c[0] * (R[I - 2][j][k])
      +c[1] * (R[I - 1][j][k]
      +R[I - 2][j - 1][k] + R[I - 2][j + 1][k]
      +R[I - 2][j][k - 1] + R[I - 2][j][k + 1])
      +c[2] * (R[I - 1][j - 1][k]
      +R[I - 1][j + 1][k]
      +R[I - 2][j - 1][k - 1] + R[I - 2][j + 1][k - 1]
      +R[I - 2][j - 1][k + 1] + R[I - 2][j + 1][k + 1]
      +R[I - 1][j][k - 1] + R[I - 1][j][k + 1])
      +c[3] * (R[I - 1][j - 1][k - 1]
      +R[I - 1][j + 1][k - 1]
      +R[I - 1][j - 1][k + 1]
      +R[I - 1][j + 1][k + 1]);

      }
```

Figure 18: Transformed MG

```
for(i = 1; i < n; i++)
  for(j = 1; j < n; j++){
      cu[i + 1][j] = 0.5 * (p[i + 1][j] + p[i][j]) * u[i + 1][j];
      cv[i][j + 1] = 0.5 * (p[i][j + 1] + p[i][j]) * v[i][j + 1];
      z[i + 1][j + 1] = fsdx * (v[i + 1][j + 1] - v[i][j + 1])-
          fsdy * (u[i + 1][j + 1] - u[i + 1][j])/p[i][j]+
          p[i + 1][j] + p[i + 1][j + 1] + p[i][j + 1];
      h[i][j] = p[i][j] + 0.25 * (u[i + 1][j] * u[i + 1][j]+
          u[i][j] * u[i][j] + v[i][j + 1] * v[i][j + 1]+
          v[i][j] + v[i][j]);

      }
```

Figure 19: Original C version of Calc1.f from SWM256

```
h[1][1] = p[1][1] + 0.25 * (u[2][1] * u[2][1]+
        u[1][1] * u[1][1] + v[1][2] * v[1][2]+
        v[1][1] + v[1][1]);
for(j = 2; j < n; j + +){
    cv[1][j] = 0.5 * (p[1][j] + p[1][j - 1]) * v[1][j];
    h[1][j] = p[1][j] + 0.25 * (u[2][j] * u[2][j]+
            u[1][j] * u[1][j] + v[1][j + 1] * v[1][j + 1]+
            v[1][j] + v[1][j]);
}

cv[1][n] = 0.5 * (p[1][n] + p[1][n - 1]) * v[1][n];
for(i = 2; i < n; i + +){
    cu[i][1] = 0.5 * (p[i][1] + p[i - 1][1]) * u[i][1];
    h[i][1] = p[i][1] + 0.25 * (u[i + 1][1] * u[i + 1][1]+
            u[i][1] * u[i][1] + v[i][2] * v[i][2]+
            v[i][1] + v[i][1]);
    for(j = 2; j < n; j + +){
            cu[i][j] = 0.5 * (p[i][j] + p[i - 1][j]) * u[i][j];
            cv[i][j] = 0.5 * (p[i][j] + p[i][j - 1]) * v[i][j];
            z[i][j] = fsdx * (v[i][j] - v[i - 1][j])-
                    fsdy * (u[i][j] - u[i][j - 1])/p[i - 1][j - 1]+
                    p[i][j - 1] + p[i][j] + p[i - 1][j];
            h[i][j] = p[i][j] + 0.25 * (u[i + 1][j] * u[i + 1][j]+
                    u[i][j] * u[i][j] + v[i][j + 1] * v[i][j + 1]+
                    v[i][j] + v[i][j]);
    }
    cv[i][n] = 0.5 * (p[i][n] + p[i][n - 1]) * v[i][n];
    z[i][j] = fsdx * (v[i][n] - v[i - 1][n])-
            fsdy * (u[i][n] - u[i][n - 1])/p[i - 1][n - 1]+
            p[i][n - 1] + p[i][n] + p[i - 1][n];
}

cu[n][1] = 0.5 * (p[n][1] + p[n - 1][1]) * u[n][1];
for(j = 2; j < n; j + +){
    cu[n][j] = 0.5 * (p[n][j] + p[n - 1][j]) * u[n][j];
    z[n][j] = fsdx * (v[n][j] - v[n - 1][j])-
            fsdy * (u[n][j] - u[n][j - 1])/p[n - 1][j - 1]+
            p[n][j - 1] + p[n][j] + p[n - 1][j];
}

z[n][n] = fsdx * (v[n][n] - v[n - 1][n])-
        fsdy * (u[n][n] - u[n][n - 1])/p[n - 1][n - 1]
        p[n][n - 1] + p[n][n] + p[n - 1][n];
```

Figure 20: CDA C version of Calc1.f from SWM256