Compiler Optimization of Array Data Storage

Rajiv Gupta and Jim Kajiya

Computer Science Department
California Institute of Technology

Caltech-CS-TR-90-07

Compiler Optimization of Array Data Storage

Rajiv Gupta and Jim Kajiya
California Institute of Technology

April 25, 1990

Caltech-CS-TR-90-07

Abstract

The literature has witnessed much work aimed at improving the efficiency of memory systems. The motivation is obvious: the high cost of page faults in hierarchical stores. Most architectures, including vector processors, shared– and distributed–memory multiprocessors, and interleaved memories, similarly reward data locality and predictable patterns of access. Most current endeavors, however, prefix the storage organization of data operands and either manipulate loops in the code, tailor algorithms, or tune prefetching strategies and replacement policies.

A methodology and an algorithm for automatically organizing the storage of array data for efficient access by the executing code are described. The simple techniques presented may be incorporated in a compiler and are complementary to the other optimizations in the literature. Furthermore, examples in Fortran 8x and Fortran 77 pseudo-code support the general applicability of the work to programming languages, both scalar as well as array. Notwithstanding the column-major storage constraints, a technique for optimizing storage in Fortran 77 is outlined. The prudence of of a compiler-optimized approach over a user-optimized approach to data storage organization is discussed.

1 Introduction

Compiler optimization has traditionally been synonymous with manipulations of code: dead-code removal, code motion, common sub-expression elimination, operator strength reduction, etc. [2]. Data is considered only its effects on the optimization of code, *eg.* flow, anti–, and output dependencies between data items constrain the parallelization and vectorization of code [1, 3, 4, 16, 18, 19]. Data optimization is usually reserved for clever programmers hoping to eke higher efficiencies out of their systems. Although there is little argument that data optimization is largely responsible for the superiority of hand-optimized code over compiler-optimized code, and although the benefits of data optimization are potentially enormous, code optimization boasts of a disproportionately higher research effort and level of automation than data optimization. The disparity may result from two subconscious biases rooted in tradition: first, the physical data space is imagined as a homogeneous monolith, and second, the virtual data space is considered too unstructured for the deduction and analysis that precede optimization.

The former reasoning implies that the cost of any two accesses in the data space is constant, and thus any optimization can at best result in fewer data accesses; which again falls within the purview of code optimization. While the physical data space of some of the earlier computers may have been homogeneous and monolithic, the physical data space of most modern systems rewards data locality. Tacit in the latter bias is the belief that data accesses are essentially random and any meaningful data optimization requires complete disambiguation of all data accesses. Thus, the bias may follow, even cases that permit data optimization offer too parsimonious a return on invested effort.

This document attempts to expose this anachronism precisely. It introduces an avenue for the optimization of data – the storage organization of arrays or the physical layout of arrays in memory – and it details a simple algorithm, the "remap" algorithm, that implements the optimization. In particular, this paper strives to automate data storage optimization to the point where it is an integral part of compiler optimization.

The remap algorithm computes the patterns in which operand data arrays are accessed by the code. These patterns then direct the storage of the arrays. Conforming data storage to the use of the data by the program improves the data locality of programs executing in environments with hierarchical stores. In environments featuring interleaved memories, vector processors, and multiprocessors, the compile-time knowledge of data access patterns can be analogously used to judiciously distribute the data so that it is efficiently available in the right form, in the right place, and at the right time: the place, form, and time being determined by the use of the data.

This leads us to anticipate payoffs that include fewer page faults in hierarchical stores, and improved efficacy of architectural constructs in which the data organization drives the parallelism, *eg.* vector processors, interleaved memories, multiprocessors. The net result: faster process turnaround, better system throughput, and more efficient resource utilization. Most current computers feature more than one of the abovementioned architectural components. The cumulative advantages, and conversely the high cost of inefficiency in these computers, make the work presented in this paper particularly relevant to computers today.

Figure 1 exemplifies the different ways of storing a matrix. Most compilers today assume and expect that arrays are stored in a fixed order, "ravel". In the case of Fortran 77, ravel order for a matrix may be column-major, while for Pascal this may be row-major. For the purposes of this paper, ravel order will imply row-major. Note that the axis that moves (unravels) slowest is referred to as "major"; conversely the axis that moves fastest is referred to as "minor". The significance lies not in the difference between the interpretations of ravel, but in the precanned rigidity of a storage organization that is independent of the eventual access of the data.

1.1 Motivation

The multiplication in a paging environment of two n-dimensional square matrices, A and B, demonstrates the payoffs obtained by the storage of data that is prompted by the use of the data, versus the storage of data that is always ravel and independent of data access. Figure 2 indicates the typical nested loop that performs the matrix multiplication. The pseudo-code makes it abundantly clear that the optimal storage order for matrix A is row-major and that for matrix B is column-major.

For a page size of p, and a working space of 3 pages, Table 1 gives the expected number of page faults for three different values of n and for the following data storage organizations: ravel, optimal, and submatrix.

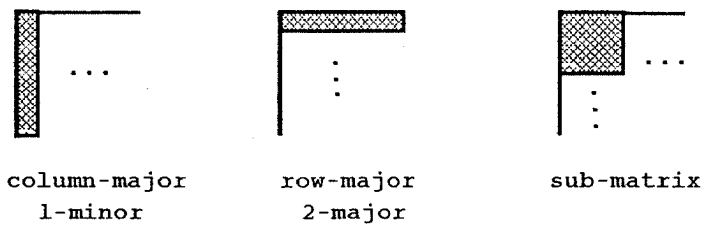


Figure 1: Storage Organizations for a Matrix

```
for i=1 to n
   for j=1 to n
      for k=1 to n
      Result[i][j] = Result[i][j] + A[i][k] * B[k][j]
```

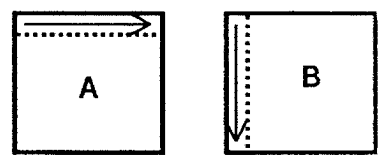


Figure 2: Typical Nested Loop for Multiplying Two $n \times n$ Matrices

The page fault computation assumes no optimization or rearrangement of the code; the sequencing in the pseudo-code is followed strictly.

From the table we can see that the improvement in page faults in going from ravel to optimal storage is approximately n, the dimensionality of the square matrices. The improvement in going from submatrix to optimal is approximately $2\sqrt{p}$, where p is the page size.

n	A row major B column major [1]	A row major B row major [2]	A submatrix B submatrix [3]	Improvement Ratio $\frac{[2]}{[1]}$	Improvement Ratio $\frac{[3]}{[1]}$
64	264	16392	16512	62	62
128	2080	262176	131584	126	63
256	16512	4194432	1050624	254	63
N	$(N+2)\frac{N^2}{p}$	$(N^2+2)\frac{N^2}{p}$	$(2N+1)\frac{N^2}{\sqrt{p}}$	$O(N)$	$O(\sqrt{p})$

Table 1: Number of page faults generated by the multiplication of two $n \times n$ matrices (p=1024, number of pages available=3)

1.2 Related Work

In the literature, most optimizations for improving the efficiency of memory hierarchies prefix the storage organization of operands and either restructure the code in the compiler-based approaches, reorganize and recode the algorithm in the algorithmic approaches [17], or tune the page prefetch and replacement policies in the operating system approaches [6, 7, 9, 20] . The few papers that discuss alternative pagination of arrays, [8, 10] for example, emphasize the costs and dynamics of transforming between alternative organizations for specific algorithms. With few exceptions, there appears no general methodology for automatically determining efficient, code-driven storage organizations of operands and intermediate results.

The Fortran 77 domain has benefited from a vigorous optimization effort aimed at improving the utilization efficiency of memory resources. The loop manipulations of [1, 3, 11, 16, 18, 19] are very effective at vectorization and parallelization or concurrentization of code, and at maximizing the reuse of data in fetched pages. The transformations, however, are handicapped by the rigid, prefixed storage organization of Fortran 77. For example, the loop interchange manipulation can sometimes mask, or ameliorate, the adverse effects of the abiding column-major storage. A loop interchange so necessitated will in many cases compromise the vectorizability of the code, or the complexity of the other manipulations. The storage restriction of Fortran 77 appears to have fettered research in manipulating the storage organizations of array data to be germane to the code.

The work described in [15] is similar in its goals and spirit with that of this paper. However, they exclusively target SIMD architectures and a data-array source language, in particular the Connection Machine and Fortran 8x respectively. In contrast, this paper presents a methodology for the general problem, and a comparison, in the context of compiler optimization, of data storage optimization in scalar languages like Fortran 77 vis-a-vis that in data-array languages like Fortran 8x or APL. This exposes the relevance of data optimization techniques to programs coded in either type of language.

Note that our goal does not preempt other compiler or algorithm optimizations for improving program attributes. This is apparent when we observe that the objects of the respective optimizations are different – in our case it is the physical distribution of data, and in the other cases it is code. Our work complements rather than outmodes other compiler optimizations. For example, some of the loop distribution manipulations often maximize data reuse in ways not targeted by our work.

The code does affect the optimal distribution of data by defining the patterns in which it accesses the data. Therefore data storage and code optimizations are inextricably linked and it is conceivable, and as we shall see also advisable, that some of the data organization considerations may be adopted in the optimization of code to produce code and data organization that are synergetically optimized. For the purposes of this document we will assume that the code is in a final, optimized form.

1.3 Organization of the Rest of the Paper

Section 2 traces the working of the remap algorithm for determining the optimal storage organization of array operands. The generalized array operators, reduction and transpose, exemplify array operator constraints on the optimal storage of operands. Section 3 discusses the applicability of data storage optimization to scalar languages. It outlines a technique for optimizing the storage of arrays in Fortran 77 in spite of its column-major constraints. Section 4 relates our work to other compiler optimizations and motivates the compiler-optimized approach over the user-optimized approach to storage organization. Section 5 concludes this document.

2 Computing Storage Order Constraints

2.1 To what level can data distribution map data access?

The previous section motivates the storage of data in the order in which it will be accessed. Furthermore, this paper advocates that the compiler should be responsible for determining data access and appropriately distributing the program operands. This naturally restricts the resolution to which the eventual data access can be determined, *eg.* elements of array operands accessed within conditional statements or within dynamically bound loops cannot be *completely* disambiguated at compile-time. But in most cases it is possible to determine at compile-time the *relative order of the axes*, or the pattern in which the data will be accessed, *eg.* the compiler may be able to deduce that axis i of operand A moves faster than axis j, which in turn moves faster than axis k, and so on. This relative speed of the indices indicates that operand A stored in i-minor, j-next-to-minor, k ... storage order maximizes the probability of addressing contiguous memory locations in consecutive accesses.

Note that this condition by itself does not guarantee that the operand storage organization is optimal over all system architectures. Even in commonplace virtual memory systems the "optimality" of data organization is dependent on the sizes and numbers, of operands and pages. In interleaved memories, the interleave factor is an architectural characteristic that affects the optimality of data organization. In vector machines it is the vector length, while in multiprocessors, relevant architectural constructs determine task distribution and include the number and type of processors, the processor connectivity, and communication bandwidth.

However, the ability of a compiler to determine and manipulate operand storage is critical to the efficient execution of programs, as was exemplified by the matrix multiplication of section 1.1. This is a good juncture to mention that in the absence of any system details, the rest of the paper will use the following definition of optimality for the organization of operand storage: consecutive accesses by the computation should address contiguous locations in the memory space. This definition coincides with the traditional implications of "locality."

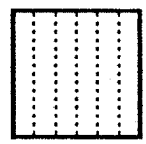
In scalar languages like Fortran 77 which require the programmer to explicitly sequence all computation in terms of scalar operations, the looping and indexing constructs readily provide the compiler with the relative ordering of operand axes. Data-array languages like APL or Fortran 8x on the other hand, provide the programmer with operators that specify the form of the final result without spelling out the computation sequencing in detail. This flexibility in computation sequencing translates to a corresponding flexibility in the optimal storage organization of the operands.

A few of the data-array operators, however, partially specify the sequencing of the constituent scalar operations over the operands. Some others do not encode any computation; instead they reorganize their operands. Reduction is typical of the former, and transpose is typical of the latter class of data-array operators. The individual effects of these two operators on optimal operand storage will be studied in the next section.

2.2 Reduction and Transpose

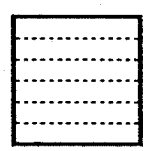
In the general form of the reduction operator, `reduce`(f, A, i), the scalar function f is applied to reduce the array operand A along the ith axis. If the compiled code will be executed in a multi-processing environment, A is optimally stored in i-minor order. For $A[m, n]$, the execution of `reduce`$(+, A, 1)$ will mirror,

```
doall j=1 to n
   B[j] = 0;
   for i=1 to m
      B[j] = B[j] + A[i][j];
```



and $B[1:n]$ will be returned as the result. If the execution environment is vector-processing, A is optimally stored with i as the major axis. The execution will now mirror,

```
B[1:n] = 0;
for i=1 to m
   B[1:n] = B[1:n] + A[i][1:n];
```



In the general form of the transpose operator, `transpose`(v, A), where v is a vector containing a permutation of integers $1 \ldots$ `rank`(A), and `rank`(A) returns the number of dimensions of A, the vector v directs the transposition of the axes of A. Although transpose does not introduce any new storage constraints, it does affect the determination of the optimal storage order by remapping constraints. Example: if rank–3 operand A is stored in 1-minor, 2-major order, the optimal storage order for `transpose`$((3\ 1\ 2), A)$ is 3-minor, 1-major; while the result of `transpose`$((1\ 3\ 2), A)$ is optimally stored in 1-minor, 3-major order. From Figure 3 it becomes apparent that as we transpose the axis at the ith position to the jth position, the relative storage order, in the major–minor spectrum, associated with the ith position is also transferred to the jth position. In other words, the transpose operator transposes the storage attributes in tandem with the shape.

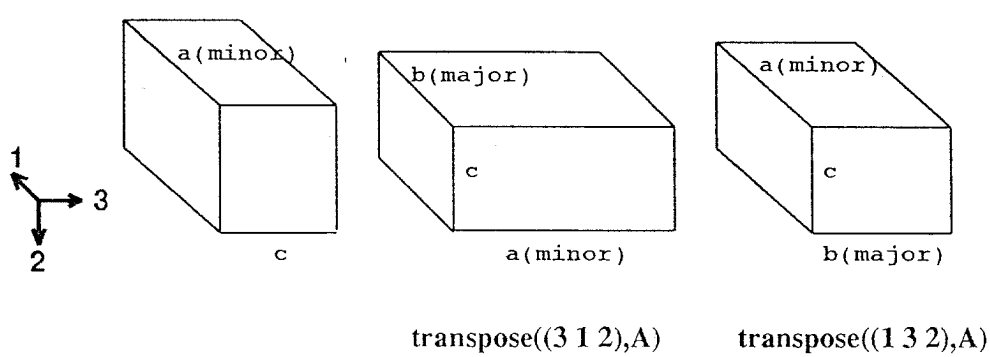


Figure 3: Storage Constraints of the Transpose Operator

A tabulation of the constraints that APL operators impose on the storage of their operands and results can be found in [14]. The relevance of APL stems from our observation that Fortran 8x [5] has adopted, and continues to adopt, many of the APL array operators.

2.3 Overview of the Remap Algorithm

The input to the remap algorithm is the output resulting from the application of all traditional optimizations on the source code. The output of the remap algorithm is an encoding, a directive perhaps to the loader, of the optimal storage organization for each data array in the source code. This storage organization is globally optimal, *ie.* it is a resolution of all the constraints that directly or indirectly affect the optimal storage organization of the array.

The previous section illustrates the constraints that the operator may introduce on the optimal storage of its operands and result. The result or a co-operand may similarly constrain the optimal storage of an operand, and vice versa. As an example, consider the arrays A, B, and C. The array sum in the expression, $A = B + C$, constrains all three arrays to have the identical storage organization. These constraints are transitive; if some other expression in the program relates the storage of B to the storage of D, then the storage constraints on D will constrain the storage of A and C through its constraints on the storage of B.

In determining the optimal storage organization for the operands in an expression, the remap algorithm maintains and manipulates a vector, the "q–vector". The q–vector accumulates constraints on storage order; remember that these constraints specify the preferred relative ordering of the operand axes. This accumulation of constraints mirrors the composition of linear transformations. The particulars of the encoding of the storage constraints in the q–vector are quite simple and are explained in the succeeding subsection. Suffice to say for now that operand storage constraints are adequately described by the contents of a vector.

The remap algorithm associates a q–vector with every operand array, atomic or intermediate, in the program computation. The starting q–vectors encode the initial constraints on the storage of the corresponding array, *eg.* the input operand to a reduction operator will have the axis of reduction as the minor-axis in a multi-processing environment, or an atomic array operand may be input to the program in a specific order. If none of the arrays have any constraints, any storage organization, including the default, is equally optimal and the algorithm terminates; however, this is rare.

Starting from the most constrained array, the remap algorithm visits all the remaining arrays in a breadth-first sequence. When two arrays are linked through an operator, the translated q–vector at the destination of the visit is an assimilation of the constraints on the array at the source of the visit, and the constraints, if any, of the operator in the path. If an array has innate constraints, and the incoming q–vector matches these initial constraints, then there is no conflict in composing the effective q–vector at that operand array. However, if the q–vectors do not conform, then the q–vector of that array is the result of resolving the conflicting constraints. This array, with the resolved q–vector as the starting constraint, is now a candidate for initiating a breadth-first visit of the program graph.

At the termination of the remap algorithm, the q–vector at each array is the resolution of constraints, from all other operators and operands, on that array. The optimal storage organization of that array is one that conforms to this q–vector. Note that in the absence of conflict, the remap algorithm has a cost proportional to the total number of arrays in the program graph. Even in cases of conflict, the resolution techniques of [13] expedite the remap algorithm to rapidly converge to a good solution.

As a preview, consider the heuristic that moderates the search of the optimal solution based on the relative costs of the operators. At one extreme, we could ignore the innate constraints of all arrays but those of the arrays associated with the most critical operator. The translated q–vector at each node is adopted without conflict. Another resolution technique, for which the cost of the remap algorithm is proportional to the total number of arrays in the program (note that this is the minimum possible cost,) simultaneously satisfies all conflicting storage constraints on an array by the use of subarray storage. For example, if a matrix has 1-minor and 2-minor as the storage constraints, the conflict might be resolved by storing the matrix as submatrices. Even when sub-optimal, this compromise is efficient, simple, and an improvement on a rigid, default storage.

2.4 The q–vector

As we have seen, every partial result has an associated q–vector that encodes the optimal storage constraints that must be satisfied by the partial result. At any point, the length of the q–vector is equal to the rank of the corresponding partial result, and the contents are such that the index of the largest element of the q–vector represents the preferred minor axis. In the same token, the index of the smallest element value represents the preferred major axis, and so on for the intermediate values.

As an example, if a rank–2 partial result, A (maybe an atomic operand,) has storage constraints described by the q–vector: 1 2, A should optimally be stored in row-major order; while the q–vector: 2 1 implies that A should optimally be stored in column-major order. Thus the ravel organization of a Pascal rank–i array is specified by q–vector: $1 \ldots i$. For Frotran 77 this might be q–vector: $i \ldots 1$. Perfect subarray storage is specified by q–vector: $1\ 1 \ldots 1$, i times. Figure 4 illustrates optimal storage corresponding to the six possible q–vectors, not including subarrays, of a rank–3 operand.

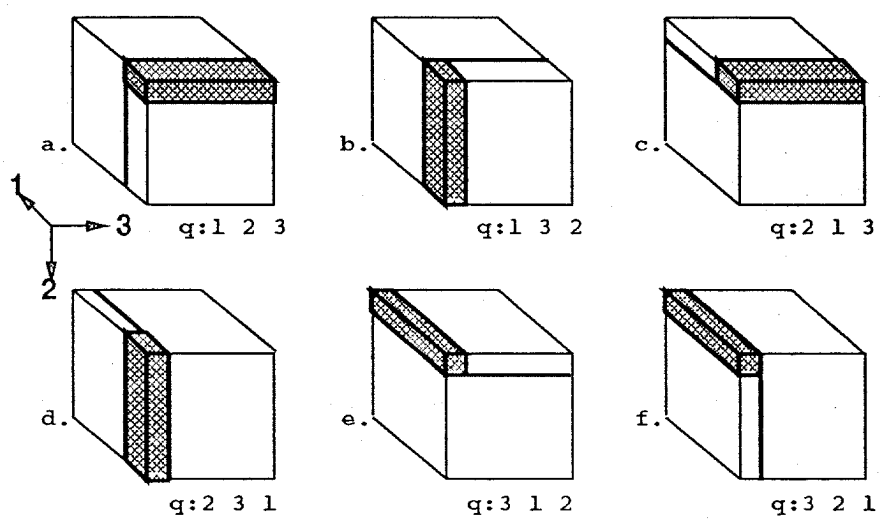


Figure 4: Optimal Storage for the Six Possible q–vectors of a rank–3 Operand

2.5 The Remap Algorithm

In incorporating the constraints of an operator, the algorithm effectively remaps the storage constraints encoded in the input q–vector into an output q–vector – hence "the *remap* algorithm." Please refer to Figure 5 during the algorithm description that follows:

1. **Create the program graph.**

   (a) The nodes of the graph represent the operators in the program computation, and the arcs represent the array operands, atomic or intermediate. Intermediate operands are also the partial results of the computation. The arrays input to, and output from, the program may be constrained to conform to a specific storage organization; hence the `IN` and `OUT` operators.

   (b) The arcs into a node represent the result of the operator at the node, with the arcs out of the node representing the operands to the operator. Monadic operators have a single arc exiting the node. The q–vectors are associated with the arcs.

2. **Initialize the q–vector of the arcs with the innate storage constraints.**
   These constraints are produced by the operators, *eg.* the reduction operator constrains its input array operand.

3. **Choose the starting q–vector (arc.)**
   The most constrained array or the array associated with the most critical opeator is usually the best choice.

4. **Push the q–vector out in a breadth-first sequence.**
   The q–vector of the arc co-incident on a node is produced by accumulating the storage constraints of the operator at the current node into the starting q–vector. Note that the operator constraint may have already been incorporated into the innate q–vector at the co-incident arc, as in the case of the reduction operator, or not, as in the case of the transpose operator.

5. **Resolve any conflicting constraints.**

We have considered only static organization of array storage. The dynamic case, where the storage of an array changes over the course of the execution of the code, is pursued as a potential resolution in [13].

2.6 Working of the remap algorithm: Example 1

In this example we compute: `reduce(+, reduce(+,`$A+B$`,2), 1)`, where A and B are rank–3 arrays. The execution environment is multi-processing. Figure 5 depicts the program graph.

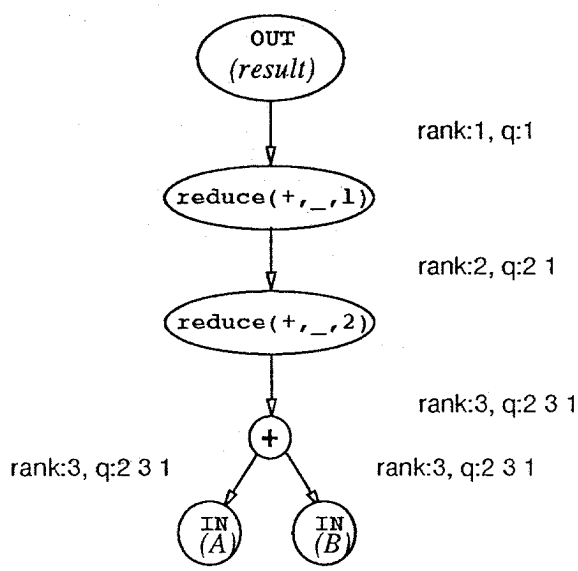


Figure 5: Example 1: `reduce(+, reduce(+,`$A+B$`,2), 1)`

The result of `reduce(+, reduce(+,`$A+B$`,2), 1)` is a vector, and is therefore labeled with q–vector: 1. From the semantics of the reduction operator in section 2.2, the operand of `reduce(+,__,`i`)` is optimally stored in i-minor order. These are the innate constraints.

We now push the `Result` q–vector: 1, through the expression graph to obtain the storage constraints for all the partial results, and atomic operands A and B. The operand to `reduce(+, reduce(+,`$A+B$`,2), 1)`, which is also the partial result of `reduce(+,`$A+B$`,2)`, accumulates the incoming q–vector: 1, and the constraint of the reduction operator into the translated q–vector: 2 1. Note that a new element was added to the incoming q–vector: 1 to conform to `length(`q–vector`)` = `rank(`corresponding partial result`)`. Also note that this q–vector conforms to the innate 1-minor constraint of `reduce(+,__,1)`.

Similarly, the operand to `reduce(+,`$A+B$`,2)`, which is also the partial result of $A+B$, acquires the q–vector: 2 3 1. Note that the index into the q–vector with the largest element value, 3, is the preferred minor axis, 2; and the order of the inherited values: 2 1, remains unchanged. The array sum operator, + of $A+B$, is equivalent to the unity transformation in that it does not introduce any additional constraint on

operand storage; therefore the q–vector: 2 3 1 is passed unaffected to the leafs, A and B. The operands A and B should optimally be stored to conform to the constraints described by q–vector: 2 3 1.

A more involved illustration in APL of the working of the remap algorithm can be found in the appendix. The APL expression there computes the inner product of two arrays. It does not use the inner product operator of APL (whose storage constraints we know from [14]) but instead uses a composition of other APL operators. Such an equivalent expression may arise because of, or in spite of, a clever programmer. Program analysis or peephole optimization for extracting the functional equivalence to the inner product and hence determining the corresponding q–vector manipulation is obviously difficult. The remap algorithm, on the other hand, operates on the semantic specification of the individual operators of the equivalent program and results in the same operand storage constraints as those resulting from the use of the inner product operator. Note that the inner product is merely an example, and any two "equivalent" programs should uniquely constrain the operands. This argument in some sense establishes a "completeness" of the q–vector transformations for the APL operators, and has significant ramifications on the equivalence of programs.

3 Applicability of our Techniques to Other languages

3.1 Are Data-Array Languages Especially Amenable to Data Optimization?

The data optimization proposed in this paper relates the storage organization of an operand to the patterns in which the operand is accessed. It is tacit that we target array operands. Independent of the programming language, every operator or programming construct in the code that accesses an array operand constrains the storage of the operand. In that sense the constraint manipulations developed in this paper span programming languages. The ease of extracting at compile time the sequence of array accesses varies with the language and its use. As an example, some languages like C encourage indirection through pointers in accessing array operands. Compile-time data optimization of programs written in such languages is naturally more complex. Fortran 77 programmers tailor algorithms to the restricted storage of the operands, and the dexterous programmer sometimes exploits such peculiarities, *eg.* assuming a linear layout of arrays in contiguous memory, making data storage optimization by the compiler counterproductive. The applicability of storage optimization is therefore subject only to the deduction of access sequences. The benefits, on the other hand, for data-array languages over scalar languages derive from one important consideration.

The advantages of data-array languages, over scalar languages, in expressing and extracting parallelism may not be relevant to data optimization that follows code optimization because,

- the same level of parallelism is acheived by code transformations that include scalar renaming and expansion, loop normalization, and loop manipulations. This follows from the observation that array operations are effectively recurrence-free `for` loops iterating over entire array extents,
- the preponderance of array operands can be easily reproduced by code optimizations in scalar languages, and,
- the looping, nesting, and indexing constructs of scalar languages manifest operand array accesses just as transparently as do the semantics of array operators in data-array languages.

The critical advantage of data-array languages, pertinent to storage optimization, rests in the flexibility in sequencing permitted by array operators. This translates to a corresponding flexibility in optimizing the storage of arrays. In contrast, scalar languages overconstrain the sequencing through arrays. Consider the following initialization fragment in a scalar language (other than Fortran 77):

```
for i=1 to n
   for j=1 to n
      A[i][j] = 0;
```

In the absence of dependency constraints, the compiler has no motivation to apply any of the loop transformations to this code. Since the remap algorithm takes the optimized code as immutable, the relative nesting of loops and indices in this code constrains the storage of A to be row-major; and that is indeed the pattern in which A will be accessed.

Consider the same initialization in a data-array language, $A = 0$. Global accesses of array A may lead the remap algorithm to conclude that A is best stored in column-major order. Since the initialization expression

does not constrain the storage of A, it can be stored in column-major order to satisfy the global constraints. The eventual execution of the initialization code will then adhere to the column-major storage. Note that the loops in an equivalent scalar encoding have been effectively interchanged. In exploiting the sequencing flexibility to optimize the storage of the arrays, the remap algorithm has tacitly optimized the sequencing without impacting the code optimization.

The same loop interchange can be effected for scalar code if the data storage constraints are allowed to motivate the code optimization. In that case, the efficacy of storage optimization in array languages can be matched by that in scalar languages other than Fortran 77. Even if the remap algorithm cannot influence code optimizations, global optimization of array storage can nevertheless improve the efficiency of the memory system and the executing code.

If the same initialization is coded in Fortran 77, the code optimizer is aware of the globally optimal storage organization: column-major. Now the storage constraints, even though independent of the code accesses, can influence the optimization of code.

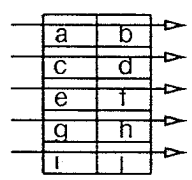
Consider the following:

- Fortran 77 imposes column-major storage on all its arrays, so the storage of arrays cannot be optimized to the accesses of the code – or can they?
- The programmer is the best judge and final arbiter on optimal data storage – or is he?

3.2 Storage Organization of Arrays in Fortran 77

All meaningful accesses of arrays are enclosed within loops in the source code – in scalar languages like Fortran 77 these loops may be explicit, while in array languages like Fortran 8x they may be mostly implicit. The loops, implied or otherwise, are schemas that concisely abstract the patterns of access in arrays; each array access indexed with the induction variable of an enclosing loop is a schema for accesses to multiple array elements. The juxtaposition of loops and the indices within these schematic array accesses represent the patterns of access which the remap algorithm then resolves to get the optimal storage organization for that array. While it is possible to program without loops, the accesses to array elements are then accesses to scalar variables and the code effectively has no arrays.

Loop manipulations therefore strongly affect the optimal storage organization of data arrays. Conversely, a prefixed storage for arrays has repercussions on the efficiency of loop manipulations. Consider the following example:




```
DIMENSION A[5][2]
for i=1 to 5
   for j=1 to 2
      A[i][j] = A[i-1][j] + 1
```

The code fragment can be directly vectorized to,

```
      for i=1 to 5
         A[i][1:2] = A[i-1][1:2] + 1;
```

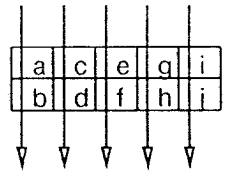
However the vectorized code accesses array A in row-major order. This conflicts with the column-major storage of A. Although the code is vectorized, a vector processor that requires its operands to be in contiguous memory locations will not execute the code as efficiently as it would if only array A had been stored in row-major order. In addition, the non-unit stride between successive vector elements might distribute the vectors in the array over more pages than available in the memory, or lines than available in the cache.

On the other hand, if the compiler interchanges the loops to conform to the localities of the column-major storage of A, the resultant code,

```
      for j=1 to 2
         for i=1 to 5
            A[i][j] = A[i-1][j] + 1;
```

cannot be vectorized. A row-major storage for A would satisfy the constraints from both, the vector processor, as well as the paged hierarchical memory of the system.

Consider the following transformed code:




```
DIMENSION A[2][5]
for i=1 to 5
   for j=1 to 2
      A[j][i] = A[j][i-1] + 1
```

This code fragment can be directly vectorized to,

```
   for i=1 to 5
      A[1:2][i] = A[1:2][i-1] + 1;
```

and this vector code satisfies both the vector processor as well as the memory system. From the accompanying figures depicting the accesses of A, we note that the transformed code preserves the dependences and hence the semantics of the original code, but it effectively transposes A.

The above example illustrates the transformation that permits the storage optimization of arrays in Fortran 77 while abiding by the column-major constraints on the storage. Whereas the remap algorithm can be directly applied to code in languages that do not restrict the storage organization, optimization of array layout in Fortran 77 need such transpositions to administer the storage recommendations of the remap algorithm. For two dimensional arrays the transposition can be implemented by:
if any array is optimally stored in row-major order,

- Transpose the definition of the array, *ie.* `DIMENSION A[i][j]` to `DIMENSION A[j][i]`
- Transpose each use of the array, *ie.* `A[i][j]` to `A[j][[i]`.

The generalization for arrays of higher dimensionality is straightforward.

Such transposition is necessary when passing array data between functions encoded in languages with differing default storage organizations, *eg.* C and Fortran 77. Typically the user explicitly programs such transpositions for the interface arrays. The remap algorithm can now motivate the automatic transposition of internal arrays to improve program efficiency while adhering to a rigid storage organization.

4 Who Should be Responsible for Data Optimization: User or Compiler?

Given a processing task and a system architecture for executing the process, the payoffs for a judicious choice of algorithm that implements the task cannot be overestimated. However, the interdependence between the encoding of the algorithm in the program and the storage organization of the operands to a large extent dictates the execution efficiency of the algorithm. The matrix multiplication example of section 1.1 bears testimony to that. There are two routes to matching code and data organization: the program may be designed to map the operand organization, or conversely, the operand storage may be organized to map the access structure of the program. Matrix multiplication again can be used as an example. A clever program can be designed to multiply two $n \times n$ matrices where both the matrices are stored in column-major order. Alternatively, the matrices may be stored as subarrays to exploit a block multiplication algorithm. Currently, compilers can manipulate loops to partially map differing operand storage organizations; they can, as we have shown, also support the automatic reorganization of operand storage to closely map differing access sequences. The user, on the other hand, may assume the responsibility of matching algorithm and data organization in either of the two routes.

Although the proficient user may optimally match his algorithm and operand storage, he needs to be alert to the optimizations that the compiler performs on his code. The storage constraints pertain to the code executed by the machine *ie.* to the object code output by the compiler. The storage constraints in the programmer's encoding of the algorithm may very well be different, and oftentimes contradictory, to those of the object code produced by the compilation and optimization of the source code. Some of the loop restructuring manipulations, most notably loop interchange, harbor such potential pitfalls.

As an example, the author of the following code,

```
for i=1 to m
   for j=1 to n
      A[i][j] = A[i][j-1] + 1;
```

stores operand A in row-major order to map the pattern of access of his code. The vectorizing compiler will however interchange the i and j loops to move the recurrence to the outermost loop. The machine will execute code that more closely resembles,

```
for j=1 to n
   A[1:m][j] = A[1:m][j-1] + 1;
```

with operand A accessed in column-major order.

The loop transformation optimizations are very effective, especially when compiling Fortran 77 programs for vector-processing and multi-processing environments. Thus the user who wishes to explicitly specify the storage organization of operands needs to inactivate these optimizations for fear of adversely affecting the efficiency and speed of his computation. This is clearly undesirable. Furthermore, such user-optimized programs suffer from low readability and maintainability.

If, on the other hand, the compiler were to optimize the storage of operands, the user would be liberated from this additional level of expertise and diligence. He would then be free to apply his ingenuity to higher levels, such as algorithm choice, which are beyond the scope of compiler optimizations, while the compiler would be responsible for all lower level code and data optimizations. Besides the ease of programmability, this specialization can lead to more efficient programs – the compiler can now freely restructure and optimize the code for the running environment and then optimally distribute the data to match the data accesses of this final executable code, or it can use data distribution constraints to drive the code optimizations, or it can apply any iterative combination of the two.

5 In Conclusion

The anachronism that leads us to assume that the physical data space is homogeneous and monolithic needs little comment. The other bias obstructing the optimization of data has been the complexity of analyzing an often unstructured data space. This paper has demonstrated that data optimization need not necessitate the complete disambiguation of all data accesses. The optimization of data storage for array operands can proceed with information only of the relative access order of the operand axes; and for this a methodology and an algorithm have been presented that are not limited to any one programming language. The payoffs of optimizing the data storage are substantial, especially for system architectures that are data driven or which reward data locality. Furthermore, the payoffs are practically achievable considering the simplicity of the remap algorithm.

There are compelling reasons for the compiler to optimize data in coordination with the optimization of code. Firstly, optimal data storage mirrors the access of the data by the code, but the code optimization by the compiler may reorganize these accesses. The option of restricting the compiler reorganization of code is self-defeating. Secondly, data storage constraints are useful evaluators of alternative code transformations by the compiler. And thirdly, the user, released from the low-level responsibility of optimizing data can now better spend his energy optimizing at higher levels which hold the promise of large improvements in execution efficiency but which are beyond the scope of the compiler.

Acknowledgement

The authors would like to thank Prof. Mani Chandy, Pieter Hazewindus, and Mike Newton for many helpful comments on this paper.

A Working of the remap algorithm: Example 2

From the multiplication algorithm presented in section 1.1 we saw that the multiplication of two n-dimensional square matrices, A and B as the left and right operands respectively, is optimal when A is stored in 2–minor and B is stored in 1–minor order. Extrapolating this observation to array operands of higher rank is relatively straightforward. For a rank–3 operand A and a rank–4 operand B, the optimal storage order for A is

3–minor and for B is 1–minor order; the relative order of the remaining axes is determined by the constraints on the result. If the rank–5 result, C, has constraints encoded by the q–vector: 5 2 1 4 3, the constraints on A are given by the q–vector: 2 1 3, and that on B by q–vector: 4 1 3 2.

From the semantics of array multiplication we know that the first two axes of C correspond to the first two axes of A, and the remaining three axes of C correspond to the last three axes of B; the third axis of A and the first axis of B are reduced by the multiplication. From the constraints on C we now proceed to develop the q–vector for A; the considerations for operand B are congruent. The q–vector for A, 2 1 3, satisfies the 3–minor order requirement of the previous paragraph. Additionally, the first axis of C is minor relative to the second axis, and since the first two axes of C are composed of the first two axes of A, the first axis of A too is relatively minor to its second axis. Thus the page accesses for operand A will cycle through the first axis before the second, and the combined page accesses of the multiplication computation will cycle through operands A and B in the following order: axis–3 of A, axis–1 of B, axis–1 of A, axis–3 of B, axis–4 of B, axis–2 of A, axis–2 of B. This order follows from the q–vector: 5 2 1 4 3 of the result.

In Example 2 we apply the remap algorithm to an equivalent program for computing the inner product of two arrays. The operands A and B, and the result C, are the same as above: C is constrained by the q–vector: 5 2 1 4 3, A has a rank of 3, B has a rank of 4, and the last axis of A and the first axis of B have the same extent. Figure 6a depicts the APL program and Figure 6b labels the program variables with information from the dimensions pass. The rank of neither operand A nor B can be deduced from the dimensions pass through the program. The remap algorithm could assume variables r_a and r_b and proceed; however for purposes of illustration let us assume that from some other program segment r_a can be deduced to be 3 and r_b to be 4. The exact shape vector remains unknown. Therefore Figure 6b assumes shape vectors consisting of variables $a \ldots f$ and conforming to the ranks r_a and r_b. Figure 6c incorporates the dimension information of Figure 6b to define the expression in terms of A and B.

```
                                  ρA = a b c; ρB = c d e f
WA ← (1 ↓ ρB),ρA                  WA = d e f a b c
VA ← ιρWA                         VA = 1 2 3 4 5 6
KA ← ρρA                          KA = 3
ZA ← KA ⌽ VA                      ZA = 4 5 6 1 2 3
TA ← ZA ⍉ WAρA                    ρTA = a b c d e f
WB ← (¯1 ↓ ρA),ρB                 WB = a b c d e f
TB ← WBρB                         ρTB = a b c d e f
      (a)                               (b)

C ↔ A +.× B ↔ +/[KA]TA × TB ↔ +/[3](4 5 6 1 2 3 ⍉ d e f a b c ρA)× (a b c d e f ρB)
                                  (c)
```

Figure 6: (a) The APL program segment for multiplying two arrays. (b) The program variables with values obtained in the dimensions pass. (c) A simplified expression for the result C

Referring to Figure 6c, the q–vector in the remap algorithm starts with the constraints on the result of the program, *ie.* q–vector: 5 2 1 4 3. From the discussions of example 1 and section 2.2, the operand of $+/[KA]$ (same as $+/[3]$), *ie.* $TA \times TB$, inherits the q–vector: 5 2 6 1 4 3. As with the array add operation before, the $\times$ operator of $TA \times TB$ does not introduce any additional constraints, and the storage organizations of both, TA and TB, are constrained by the q–vector: 5 2 6 1 4 3.

The q–vector constraining TA is now pushed down the expression that computes TA to obtain the storage constraints on operand A. The first operator we encounter is the dyadic transpose with ZA as the control argument and $WA\rho A$ as the operand; let variable X be the result of $WA\rho A$. The dimensions pass gives us (4 5 6 1 2 3) as the value of ZA. In the normal right–to–left execution of the dyadic transpose, the fourth axis of X would have become the first axis of the result. From the q–vector of the result we know that this first axis of the result has a q–vector value of 5; thus the left–to–right accumulation of the dyadic transpose into the q–vector gives 5 in the fourth position of the q–vector constraining X. By extending this reasoning, the accumulation of the dyadic transpose with control argument (4 5 6 1 2 3) transforms the q–vector: 5 2 6 1 4 3 constraining TA into the q–vector: 1 4 3 5 2 6 constraining X.

The q–vector constraining X is now pushed down the expression that computes X, *viz.* $WA\rho A$. Borrowing a terminology from Guibas & Wyatt [12], the reshape in this expression is called a "conforming" reshape because WA is of the form: $(Y, \rho A)$, where Y is a vector or a scalar. Such a reshape preserves the structure of A; it merely creates multiple copies of A along the added dimensions Y. Conforming reshapes pass on the "normalized" suffix of the q–vector to the operand. The corresponding suffix of the q–vector, Q, that will be inherited by A is: $(-\rho\rho A) \uparrow Q$. The range of the values in the q–vector suffix pertain to the parent q–vector Q, *ie.* between 1 and $\rho\rho Q$. Normalization conserves the relative order of the suffix values but makes them pertinent to operand A, *ie.* it converts the values to lie between 1 and $\rho\rho A$. If S is the unnormalized suffix, then $+/[2](S \circ . \geq S)$ normalizes S.

In our case the relevant suffix, S, of the q–vector is: 5 2 6, and the q–vector passed on to operand A is: 2 1 3. This constraint on operand A is the same as that obtained by the analysis of the multiplication of arrays A and B with the rank–5 result, C, constrained by the q–vector: 5 2 1 4 3. A similar set of manipulations shows that operand B is constrained by the q–vector: 4 1 3 2.

References

[1] W. Abu-Sufah, D. Kuck, and D. Lawrie. Automatic program transformations for virtual memory computers. *National Computer Conference*, pages 969–974, 1979.

[2] A. V. Aho and J. D. Ullman. *Principles of Compiler Design.* Addison–Wesley, Mass., 1979.

[3] John Randal Allen. *Dependence Analysis for Subscripted Variables and its Application to Program Transformation.* PhD thesis, Rice University, April 1983.

[4] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, 1987.

[5] Inc. American National Standards Institute. Fortran 8x draft. Technical Report 4, SIGPLAN Special Interest Publication on Fortran, 1989.

[6] Jean-Loup Baier and Gary R. Sager. Dynamic improvement of locality in virtual memory systems. *IEEE Trans. on Software Engineering*, SE–2(1):54–62, March 1976.

[7] L. A. Belady. A study of replacement algorithms for a virtual–storage computer. *IBM Systems Journal*, 1966.

[8] Michael W. Condry. On arbitrary array pagination strategies: Storage and reorganization. *Proc. of the Johns Hopkins Conference on Information and System Sciences*, pages 385–390, March 1978.

[9] Domenico Ferrari. The improvement of program behaviour. *IEEE Computer*, pages 39–47, Nov. 1976.

[10] Patrick C. Fischer and Robert L. Probert. Storage reorganization techniques for matrix computation in a paging environment. *Comm. of the ACM*, 22(7):405–415, July 1979.

[11] Dennis Gannon, William Jalby, and Kyle Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel & Distributed Computing*, (5):587–616, 1988.

[12] Leo J. Guibas and Douglas K. Wyatt. Compilation and delayed evaluation in APL. *Fifth Annual Symposium on Principles of Programming Languages*, pages 1–8, 1978.

[13] Rajiv Gupta. Resolving constraints on optimal data storage: Complexity and heuristics. Technical Report in preparation, California Institute of Technology, 1990.

[14] Rajiv Gupta. Storage constraints of APL operators. Technical report, California Institute of Technology, 1990.

[15] Kathleen Knobe, Joan D. Lukas, and Jr. Guy L. Steele. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, (8):102–118, 1990.

[16] D.J. Kuck, R. H. Kuhn, D. A. Padua, B. Leisure, and M. Wolfe. Dependence graphs and compiler optimizations. *Proc. of the Eigth ACM Symposium on Principles of Programming Languages*, pages 207–218, Jan. 1981.

[17] A.C. McKellar and E. G. Coffman, Jr. Organizing matrices and matrix operations for paged memory systems. *Comm. of the ACM*, 12(3):153–165, March 1969.

[18] David Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Comm. of the ACM*, 29(12):1184–1201, Dec. 1986.

[19] Constantine D. Polychronopoulos. Automatic restructuring of fortran programs for parallel execution. Technical Report 665, Univ. of Illinois, Urbana–Champaign, June 1987.

[20] Kishor S. Trivedi. Prepaging and applications to array algorithms. *IEEE Trans. on Computers*, C–25(9):915–921, Sept. 1976.