Automatic Blocking of Nested Loops

Robert Schreiber*
Research Institute for Advanced Computer Science
Mail Stop 230-5, NASA Ames Research Center
Mountain View, CA 94035
e-mail: `schreiber@riacs.edu`

Jack J. Dongarra†
Department of Computer Science
University of Tennessee
Knoxville, TN 37996-1301
and
Mathematical Sciences Section
Oak Ridge National Laboratory
Oak Ridge, TN 37831
e-mail: `dongarra@cs.utk.edu`

May 21, 1990

Abstract

Blocked algorithms have much better properties of data locality and therefore can be much more efficient than ordinary algorithms when a memory hierarchy is

*Supported by the NAS Systems Division and/or DARPA via Cooperative Agreement NCC 2-387 between NASA and the University Space Research Association (USRA).

†Supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract DE-ACOS-84OR21400.

involved. On the other hand, they are very difficult to write and to tune for particular machines. Here we consider the reorganization of nested loops through the use of known program transformations in order to create blocked algorithms automatically. The program transformations we use are strip mining, loop interchange, and a variant of loop skewing in which we allow invertible linear transformations (with integer coordinates) of the loop indices. In this paper, we solve some problems concerning the optimal application of these transformations. We show, in a very general setting, how to choose a nearly optimal set of transformed indices. We then show, in one particular but rather frequently occurring situation, how to choose an optimal set of block sizes.

Keywords: block algorithm, parallel computing, compiler optimization, matrix computation, numerical methods for partial differential equations, program transformation, memory hierarchy.

AMS(MOS) subject classifications: ???? 05C50, 15A23, 65F05, 65F50, 68M20.

1 Introduction

An essential fact of life in very-large-scale integrated circuits is that transistors are cheap and wires are expensive. The concomitant fact in high-performance computing, especially parallel computing, is that computation is cheap and communication is expensive. The two types of communication that we are primarily concerned with here are communication between the processors in a multicomputer and communication between processors and main memory.

Both these forms of communication are characterized by long latency and low bandwidth compared to the CPU rate. For instance, in the CRAY-1 memory was able to provide only 80 Mwords per second to the vector unit, which could produce one result and take in two operands per clock at 80 MHz; thus the memory was too slow by a factor of three. This same phenomenon can be observed in the i860 RISC today, the NEC-SX supercomputers, the Alliant machines, the CM-2, and most other high-performance machines. Communication speeds are likewise slower than processor speeds in multicomputers such as the Intel iPSC/2. In that machine, processors communicate over 1-bit-wide channels but have full word-wide paths to local memory. While newer message passing machines will employ byte-wide communication channels, the evolving microprocessor already provides memory ports of 8 or 16 bytes.

The principal architectural solution to these problems is to provide a small but fast local memory at each processor. The memory may be managed by hardware on a demand basis (cache) or managed explicitly by software, either operating system or application. If the processor is B times faster than the data path to memory or to other processors, then it must make reference to that slow data path only once for every B operations in order not to be slowed down. For this to happen, it must get its data from the local memory roughly $B-1$ times out of every B. Software must organize the computation so that this "hit ratio" can be achieved.

1.1 Block algorithms: Matrix Multiplication as an Example

To achieve the necessary reuse of data in local memory, researchers have developed many new methods for computation involving matrices and other data arrays [6, 7, 16]. Typically an algorithm that refers to individual elements is replaced by one that operates on subarrays of data, which are called *blocks* in the matrix computing field. The operations on subarrays can be expressed in the usual way. The advantage of this approach is that the small blocks can be moved into the fast local memory and their elements can then be repeatedly used.

The standard example is matrix multiplication. The usual program is

for $i = 1$ **to** n **do**
    for $j = 1$ **to** n **do**
        for $k = 1$ **to** n **do**
            $c[i,j] = c[i,j] + a[i,k] * b[k,j]$; **od**
        od

```
    od
```

The entire computation involves $2n^3$ arithmetic operations (counting additions and multiplications separately), but produces and consumes only $3n^2$ data values. As a whole, the computation exhibits admirable reuse of data. In general, however, an entire matrix will not fit in a small local memory. The work must therefore be broken into small chunks of computation, each of which uses a small enough piece of the data. Note that for each iteration of the outer loop (*i.e.,* for a given value of i) n^2 operations are done and n^2 data is referred to — no reuse. For fixed values of i and j, n computation and n data referred too — again, no reuse.

Now consider a blocked matrix-multiply algorithm.

```
for i0 = 1 to n step b do
    for j0 = 1 to n step b do
        for k0 = 1 to n step b do
            for i = i0 to min(i0 + b − 1, n) do
                for j = j0 to min(j0 + b − 1, n) do
                    for k = k0 to min(k0 + b − 1, n) do
                        c[i, j] = c[i, j] + a[i, k] * b[k, j] ; od
                    od
                od
            od
        od
    od
```

First, note that in this program exactly the same operations are done on the same data; even round-off error is identical. Only the sequence in which independent operations are performed is different from the unblocked program. There is still reuse in the whole program of order n. But if we consider one iteration with fixed $i0$, $j0$, and $k0$, we see that $2b^3$ operations are performed (by the three inner loops) and $3b^2$ data are referred to. Now we can choose b small enough so that these $3b^2$ data will fit in the local memory and thus achieve b-fold reuse. (If this isn't enough — if $b < B$ in other words — then the machine is poorly designed and needs more local memory.) Put the other way, if we require B-fold reuse, we choose the block size $b = B$.

The subject of this paper is the automatic transformation of ordinary programs to blocked form.

Our motivation for seeking such a capability is as follows. Many algorithms can be blocked. Indeed, recent work by numerical analysts has shown that the most important computations for dense matrices are blockable. A major software development of blocked algorithms for linear algebra has been conducted as a result [5]. Further examples, in particular in the solution of partial differential equations by difference and variational methods, are abundant. Indeed, many such codes have also been rewritten as block methods to better use the small main memory and large solid-state disk on Cray supercomputers [14]. All experience with these techniques has shown them to be enormously effective at squeezing the best possible performance out of advanced architectures.

On the other hand, blocked code is much more difficult to write and to understand than point code. Writing it is a difficult and error-prone job. Blocking introduces block size parameters that have nothing to do with the problem being solved and which must be adjusted for each computer and each algorithm if good performance is to be achieved. Unfortunately, the alternative to having blocked code is worse: poor performance on important computations with the most powerful computers. For these reasons, Kennedy has stated that compiler management of memory hierarchies is the most important and most difficult task facing the writers of compilers for high-performance machines [12].

1.2 Program Transformation and Blocking; Previous Work

We can view the reorganized matrix-multiply program in two ways. First, we can consider the matrices A, B, and C as $\frac{n}{b} \times \frac{n}{b}$ matrices whose elements are $b \times b$ matrices. In this case, the inner three loops simply perform a multiply-add of one such block-element. This is the view taken by most numerical analysts. Second, we can derive the blocked program form the original, unblocked program by a sequence of standard program transformations. First, the individual loops are *strip mined.* For example, the loop

for $i = 1$ **to** n **do**
    $\cdots$ **od**

is replaced by

for $i0 = 1$ **to** n **step** b **do**
    for $i = i0$ **to** $\min(i0 + b - 1, n)$ **do**
        $\cdots$ **od**
    od

(Strip mining is a standard tool for dealing with vector registers. One may apply it "legally" to any loop. By legally, we mean that the transformed program computes the same result as before.) Strip mining, by itself, yields a six-loop program, but the order of the loops is not what is needed for a blocked algorithm. The second transformation we use is loop interchange. In general, this means changing the order of loops and hence the order in which computation is done. To block a program, we endeavor to move the strip loops (the $i0$, $j0$, and $k0$ loops above) to the outside and the point loops (the i, j, and k loops above) to the inside. This interchange is what causes repeated references to the elements of small blocks. In the matrix-multiply example, the interchange is legal, but there are many interesting programs for which it is not, including LU and QR decompositions and methods for partial differential equations.

This approach to automatic blocking, through loop strip mining and interchange, was first advocated by Wolfe [18]. It is derived from earlier work of Abu-Sufah, Kuck, and Lawrie on optimization in a paged virtual-memory system [1]. Wolfe introduced the term *tiling.* A *tile* is the collection of work to be done, *i.e.,* the set of values of the point loop indices, for a fixed set of values of the block or outer loop indices. We like this terminology since it allows us to distinguish what we are doing — which is to decompose the work to be done into small subtasks (the tiles) — from the quite different task of decomposing the data *a priori* into small subarrays (the blocks), even though each tile does, in fact, act on blocks. Following Wolfe, we call the problem of decomposing the work of a loop nest *index space tiling.*

Other authors have treated the issue of management of the memory hierarchy [8]. Some other treatments of the problem of automatic blocking have recently appeared [11], [4], [17], [18], [19]; none, however, gives the quantitative statments of the problem and the solution that we provide here.

1.3 Strip Mining and Loop Interchange Are Not Enough

Consider the one-dimensional, discrete diffusion process

for $t = 0$ **to** m **do**
    for $i = 1$ **to** $n - 1$ **do**
        $u[i,t] = f(u[i-1,t-1], u[i,t-1], u[i+1,t-1])$; **od**
    od

At each time step (each iteration of the t loop) at every grid point, the value of $u(i)$ is updated by using the data at the three grid points $i-1$, i, and $i+1$ from the previous time step, $t-1$. This process is typical of PDE computations. Let us apply strip mining and loop interchange to this code. The resulting program, which follows, is incorrect.

for $t0 = 0$ **to** m **step** bt **do**
    for $i0 = 1$ **to** $n-1$ **step** bi **do**
        for $t = t0$ **to** $\min(m, t0 + bt - 1)$ **do**
            for $i = i0$ **to** $\min(n-1, i0 + bi - 1)$ **do**
                $u[i,t] = f(u[i-1,t-1], u[i,t-1], u[i+1,t-1])$; **od**
            od
        od
    od

One cannot advance the computation in time for a fixed subset of the grid points without advancing it for their neighbors; to update the values at the edge of the block of grid points, we require values from neighboring grid points outside the block that have not been computed. In other words, the loop interchanges that we performed were illegal and, the transformed program produces meaningless results.

Wolfe's second paper on tiling recognizes this fact. He advocates the use of a technique called *loop skewing* [19]. (This was also discussed by Irigoin and Triolet [11].) By loop skewing, Wolfe means changing the index of the inner loop from the natural variable (i above) to the sum or difference of the old inner index and an integer multiple of the outer loop index. With this transformation, the code above can be changed as follows:

```
for t = 0 to m do
    for r = t + 1 to t + n - 1 do
        u[r - t, t] = f(u[r - t - 1, t - 1], u[r - t, t - 1], u[r - t + 1, t - 1]); od
    od
```

Here we have used $r = i + t$ as the inner loop index. Note that the inner loop now ranges over oblique lines in the (i, t) plane. We may now legally strip mine and interchange to get a tiled program:

```
for t0 = 0 to m step bt do
    for r0 = t0 + 1 to t0 + n - 1 step br do
        for t = t0 to min(m, t0 + bt - 1) do
            for r = max(r0, t + 1) to min(t + n - 1, r0 + br - 1) do
                u[r - t, t] =
                    f(u[r - t - 1, t - 1], u[r - t, t - 1], u[r - t + 1, t - 1]); od
            od
        od
    od
```

Figure 1 shows the tiles of computation in the original coordinates (i, t).

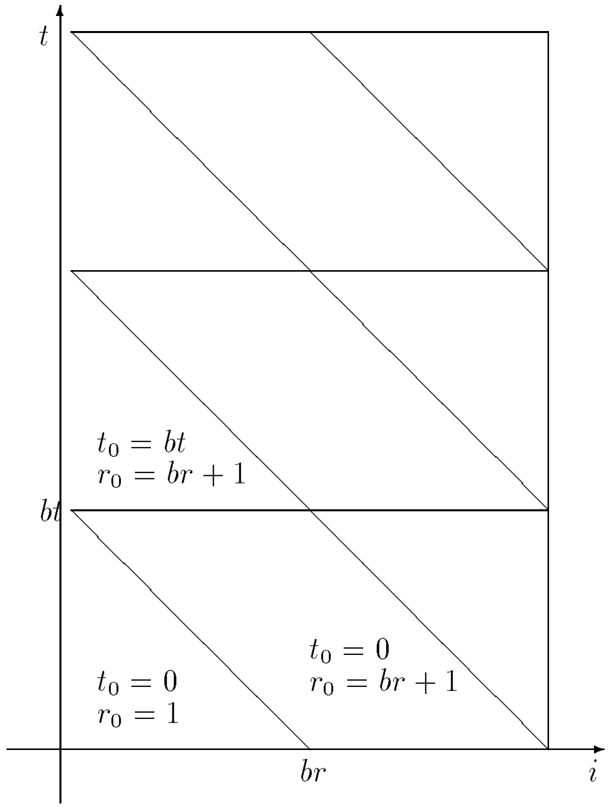

Figure 1: Tiled index space, with new inner index $r \equiv t + i$.

.

In this paper, we consider the following generalization of Wolfe's loop skewing. We allow all of the loop indices to be replaced by linear combinations of the original, natural indices. Let the computation be a loop nest of depth k. Let the natural indices be $(i_1, i_2, \ldots, i_k)$. Let A be an invertible, $k \times k$ integer matrix. We would like to use $(j_1, j_2, \ldots, j_k)$ as the indices in a transformed program, where

$$j = A^T i \, .$$

We can carry out this transformation in two steps. First, we replace every reference to any of the natural loop indices in the program by a reference to the equivalent linear combination of the transformed indices. If the rational matrix $F = [f_{pq}]_{p,q=1}^{k} = A^{-T}$ (A^{-T} denotes the inverse of A^T), then we replace a reference to i_1, for example, by the linear combination

$$f_{11} \cdot j_1 + f_{12} \cdot j_2 + \ldots + f_{1,k} \cdot j_k \, .$$

Second, we compute upper and lower bounds on the transformed indices. We call this program rewriting technique *loop index transformation.*

The first contribution of this work is a method for choosing the loop index transformation A. We start from the assumption that the computation is a nested loop of depth k in which there are some loop-carried dependences with fixed displacements in the index space. We then consider the problem of determining which loop index transformations A permit the resulting index-transformed loop nest to be successfully tiled through strip mining and interchange. (The mechanics of automating these program transformation is discussed in the compiler optimization literature [3].) We show that this problem amounts to a purely geometric one: finding a basis for real k-space consisting of vectors with integer components that are constrained to lie in a certain closed, polygonal cone defined by the dependence displacements. The basis vectors are then taken to be the columns of the loop index transformation A. We further show that the amount of reuse that can be achieved with a given amount of local memory, which is determined by the ratio of the number of iterations in a tile to the amount of data required by the tile, is dependent on A in a simple way. It is proportional to the $(k-1)^{\text{th}}$ root of $\det(\bar{A}\,)$ where $\bar{A}$ is the matrix obtained by scaling the columns of A to have euclidean length one.

We give a heuristic procedure for determining such an integer matrix A that approximately maximizes this determinant. We report on the results of some experiments to test its performance and robustness.

Finally, we consider the optimal choice of tile size and shape, once the basis A has been determined. We show that it is straightforward to derive block size parameters that maximize the ratio of computation in a tile to data required by the tile, given knowledge of the flux of data in the index space and the blocking basis A.

1.4 Notation

We use uppercase letters for matrices. The notation $X = [x_1, x_2, \ldots, x_k]$ means that X has columns $x_1, x_2, \ldots, x_k$. The notation $X = [x_{i,j}]$ means that X has elements $x_{i,j}$.

In general, we use lowercase Greek letters for scalars. Let

$$\delta_{ij} \equiv \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

The following symbols have the indicated meaning

$\mathcal{S}$	The index space — the set of values of the loop index vector
A	The matrix that transforms natural to new loop indices
$\bar{A}$	The matrix A with its columns scaled to have euclidean length one
F	A^{-T}
D	The matrix of dependences
C	The matrix of data fluxes
ρ	The ratio of the volume of a tile to its surface area
$b = [\beta_j]$	The vector of block size parameters
a_j	A normal vector to a tiling hyperplane; one of the columns of A
μ	A bound on the size of local memory.
ω	The time required to perform the computation at a point in the index space.

ϕ_j The time required to move data across one unit of area in the hyperplane normal to a_j.

We shall make considerable use of determinants. If $X = [x_1, \ldots, x_n]$ is a real, square matrix, then the real-valued function $\det(X)$ is the volume of the parallelepiped subtended by the columns of X:

$$\left\{ \sum_{j=1}^{n} \alpha_j x_j \mid 0 \le \alpha_j \le 1 \right\}.$$

Thus, $\det(I) = 1$. Also $\det(X) = \det(X^T)$. If Y is also $n \times n$, then $\det(XY) = \det(X)\det(Y)$. If $T = [t_{i,j}]$ is a triangular matrix, then $\det(T) = (t_{11} \cdot t_{22} \cdots t_{nn})$.

Let $sp\{z\}$ denote the one-dimensional subspace spanned by the vector z, and let $sp\{z\}^{\perp}$ denote its orthogonal complement.

Lemma 1 *Let x_1 have length one. Let $X_1 \equiv [x_2, \ldots, x_n]$. Let P_1 be the orthogonal projection matrix on $sp\{x_1\}^{\perp}$. Then*

$$\det(X) = \det(P_1 X_1).$$

Proof: Let $c = (c_2, \ldots, c_n)^T$ be a $k-1$-vector chosen so that for each $2 \le j \le n$, $x_j - c_j x_1$ is orthogonal to x_1. Construct the matrix

$$C = \begin{pmatrix} 1 & -c^T \\ 0 & I_{n-1} \end{pmatrix}.$$

Then, since C is triangular and has unit diagonal, $\det(C) = 1$. Since x_1 is a vector of length one, $XC = [x_1, P_1 X_1]$. Thus,

$$\begin{aligned} \det(X)^2 &= \det(X^T X) \\ &= \det(C^T X^T X C) \\ &= \det([x_1, P_1 X_1]^T [x_1, P_1 X_1]) \\ &= \det \begin{pmatrix} 1 & 0 \\ 0 & (P_1 X_1)^T (P_1 X_1) \end{pmatrix} \\ &= \det(P_1 X_1)^2. \end{aligned}$$

2 Statement of the Problem

We are given a convex set of lattice points $\mathcal{S} \in \mathbf{Z}^k$. This is the set of all values of the k dimensional natural index $i = (i_1, i_2, \ldots, i_k)$ in the loop nest. We call $\mathcal{S}$ the *index space* of the loop nest, which is the standard term, even though $\mathcal{S}$ is a finite subset of $\mathbf{Z}^k$. We are also given a set of dependence displacements $D = [d_1, \ldots, d_m]$ where each integer vector $d_j \in \mathbf{Z}^k$ is the displacement in the index space from iterations that produce values to iterations that use them. The integer m is the number of such dependences. Hence, for all points $i \in \mathcal{S}$ and for each $1 \leq j \leq m$, if $i - d_j \in \mathcal{S}$, then iteration $i - d_j$ must have been performed before we perform iteration i. We may also consider anti-dependences and output dependences and treat them in the same manner. (See [8] for the definition of the various kinds of dependences.)

We now consider the blocking problem. The problem is to partition $\mathcal{S}$

$$\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2 \cup \ldots \cup \mathcal{S}_p \tag{1}$$

where the subsets of index points $\{\mathcal{S}_j\}$ are disjoint. The j^{th} tile is the task of executing the loop body for all values of the loop index in $\mathcal{S}_j$.

Some restrictions are in order if this partition of $\mathcal{S}$ is to be of any use. The key restriction was stated by Wolfe [19]:

> "Each tile is a unit of computation to be scheduled on a processor. Once a tile is scheduled ... it runs to completion without preemption. A tile will not be initiated until all dependence constraints for that tile are satisfied, so there will never be a reason that a tile, once started, should have to relinquish the processor."

We call this the *atomicity requirement.*

The second, less fundamental but nevertheless important restriction is that the tiling should be expressible as a transformation of the original program. For this reason, we restrict our attention to partitions of $\mathcal{S}$ achieved by cutting $\mathcal{S}$ along hyperplanes. Wolfe's original tilings used planes normal to the natural coordinate axes. Here, we allow arbitrary planes with integer normals. If we want to cut up $\mathcal{S}$ along hyperplanes normal to the integer vector a, we first apply loop index transformation to one of the original loops, replacing its index with $a^T i$. We then strip mine this loop and bring the strip loop to the outermost position.

2.1 Definitions

First, we define the type of partition of $\mathcal{S}$ that we are considering. Let an integer vector $a \in \mathbf{Z}^k$ and an integer block size parameter β be given. The partition induced by a and β is given by (1) where

$$\mathcal{S}_j = \{i \in \mathcal{S} \mid (\min_{i \in \mathcal{S}} a^T i) + (j-1)\beta \le a^T i < (\min_{i \in \mathcal{S}} a^T i) + j\beta\}\,.$$

(Imagine a knife aligned so that a is normal to its flat side, cutting $\mathcal{S}$ into slices of equal thickness β.)

We associate with $\mathcal{S}$ and D the dependence graph $G = G(\mathcal{S}, D)$ with vertices $\mathcal{S}$ and edges

$$E = \{(i, i') \in \mathcal{S} \times \mathcal{S} \mid \exists \text{ a column } d_j \text{ of } D \ni i + d_j = i'\}\,.$$

We assume that G is acyclic. (If the dependence graph comes from a loop nest in an imperative language like Fortran, then G has to be acyclic.)

Definition 1 *The set*

$$\mathcal{C} = \mathcal{C}(D) \equiv \{x \in \mathbf{R}^k \mid D^T x > 0\}$$

is called the time cone *of D. (The inequality is interpreted componentwise.)*

Note that $\mathcal{C}$ is an open, convex set closed under multiplication by a positive scalar – *i.e.,* $\mathcal{C}$ is in fact a cone. It is polygonal, the intersection of the half spaces $\{d_j^T x > 0\}_{j=1}^m$. We call $\mathcal{C}$ the time cone, without mentioning D, whenever there is no ambiguity.

The closure of $\mathcal{C}$ is also important; it is defined by

$$\bar{\mathcal{C}} = \bar{\mathcal{C}}(D) \equiv \{x \in \mathbf{R}^k \mid D^T x \ge 0\}\,.$$

Two subsets of $\bar{\mathcal{C}}$ are important here. First, we must choose, as the normals to the hyperplanes used to partition $\mathcal{S}$, integer vectors in $\bar{\mathcal{C}}$. The intersection of $\bar{\mathcal{C}}$ with the surface of the unit sphere in $\mathbf{R}^k$ (with the euclidean norm) also plays a role.

Lemma 2 *If $\mathcal{C}$ is nonempty, then G($\mathcal{S}$, D) is acyclic.*

For the proof, observe that the iterations may be performed in order of increasing value of $x^T i$ where x is any vector in $\mathcal{C}$. Because all dependence displacements d_j make an acute angle with such an x, no dependence constraint is violated. We may therefore interpret $x^T i$ as the time at which iteration i is to be performed, hence the name we have given $\mathcal{C}$. Points of $\mathcal{S}$ with equal value of $x^T i$ are independent of one another and can be executed in any order – or in parallel, for that matter. This is the essence of Lamport's hyperplane method for the parallel execution of do-loops [13].

Again, if D results from a loop nest in Fortran or a language like it, we can show that $\mathcal{C}$ is not empty. In fact, it is easy to see that D has the property that the first nonzero element of every column is positive (*i.e.* it is lexicographically positive.) From this, the nonemptyness of $\mathcal{C}$ easily follows.

We can now show how to choose hyperplanes for partitioning $\mathcal{S}$ in such a manner that Wolfe's atomicity requirement is satisfied. First, we restate the requirement in terms of the quotient of the dependence graph under the partition (1).

Definition 2 *The quotient graph of $G = G(\mathcal{S}, D)$ under the partition (1) is the graph with vertices $\{\mathcal{S}_1, \ldots, \mathcal{S}_s\}$ and edges*

$$\{(\mathcal{S}_p, \mathcal{S}_q) \mid \exists\ i_p \in \mathcal{S}_p \text{ and } i_q \in \mathcal{S}_q \ni (i_p, i_q) \text{ is an edge of } G\}\,.$$

The atomicity requirement is equivalent to the requirement that the quotient graph be acyclic. A sufficient condition for this is the following.

Lemma 3 *The quotient graph of G($\mathcal{S}$, D) under the partition induced by a and β is acyclic if $a \in \bar{\mathcal{C}}$.*

For the proof, observe that, by their definition, the subsets of the partition induced by a and β may be ordered according to the values taken by $a^T i$ on them. It follows from the definition of $\bar{\mathcal{C}}$ that no point in a lower numbered subset can depend on any point in a higher numbered subset; if there were such a pair, say a point x that depends on a point y such that $x - y = d$ for some column d of D, then d makes an obtuse angle with a, *i.e.,* $a^T d < 0$, since by assumption $a^T x < a^T y$. But by definition, $a^T d \geq 0$ for all columns d of D.

Moldovan and Fortes [15] have used this technique for the synthesis of systolic arrays without cyclic data flow, which allows the array to be used to solve problems larger than the array. They gave no method for choosing the hyperplanes. The material of this section was also presented by Irigoin and Triolet [11].

2.2 Tiling with Hyperplanes

From the discussion above, we see that a valid partition of $\mathcal{S}$ may be obtained by choosing any integer vector in $\bar{\mathcal{C}}$. The tiles so obtained are slices of the index space $\mathcal{S}$. These are not sufficiently small, however, to allow for all their necessary data to fit in the local memory of a given computer.

In terms of the corresponding program, tiling by slicing with a single hyperplane can be achieved by a loop reindexing of one loop followed by strip mining of that loop (and only that loop) and interchange to make the one resulting strip-loop outermost. In the case of matrix multiply, for example, this would result in a four-loop program in which the innermost three loops do $n^2 b$ operations and use n^2 data. (For, no matter which loop we strip mine, one of the matrices is indexed by the two unchanged loop indices and so is completely accessed.)

As the matrix multiply example indicates, we need to be able to strip mine all the loops in order to be able to work with tiles whose data sets can be made arbitrarily small. In this section we investigate the problem of fully tiling loop nests.

We can state this problem as follows. Given the index space $\mathcal{S} \subset \mathbf{Z}^k$

and the dependence matrix D, choose k linearly independent vectors $A \equiv [a_1, \ldots, a_k]$ (the columns of A are a basis for k-space) such that each $a_j \in \bar{\mathcal{C}}$.

The partition induced by A and a k-vector of block size parameters b is obtained by slicing $\mathcal{S}$ into slices of thickness β_1 with a knife aligned perpendicular to a_1, then slicing again with thickness β_2 and with the knife rotated so that it is perpendicular to a_2 (making long, narrow strips rather than slices) and so on, until one has sliced k times, finally obtaining tiles that are shaped, except at the boundaries of $\mathcal{S}$, like parallelepipeds whose faces are perpendicular to the basis vectors.

Thus, in order to fully tile a loop nest with arbitrary dependences D, we must be able to choose a basis in the cone $\bar{\mathcal{C}}$.

Should we be satisfied with any such basis? What if its elements are nearly linearly dependent? Then we have tiles that are quite elongated, with some very small angles and a low ratio of volume (which measures the number of lattice points, or iterations to be performed) to surface area. The surface area is a measure of the amount of data that must be moved into local memory in order to carry out the work of a tile. In general, the data moved in is the data required because of dependences of iterations in the tile on iterations of other tiles. The iterations near the edges require this data from outside.

The $(k-1)$ dimensional volume of the tile, which grows like $\prod_{j=1}^{k} \beta_j$, is also a measure of the amount of local memory needed to carry out the work of each tile.

Let us therefore calculate how the choice of A determines the volume-to-surface ratio of the induced tiles. We first answer the question for the tiling that results when $b = (\beta, \beta, \ldots, \beta)^T$. We obtain a formula for the ratio when $\beta \equiv 1$, then we show how varying β changes both the ratio and the amount of local storage needed. In later sections we consider generalizing to tiles with non-unit aspect ratios.

2.3 Geometric Considerations

First, we note that if $a \in \bar{\mathcal{C}}$, then so is αa for any positive scalar α. The partition induced by A and b is unchanged if we scale the columns of A by arbitrary positive amount and scale the corresponding components of b by the reciprocal amounts. There is therefore no loss of generality if we replace A with $\bar{A}$, the matrix obtained by scaling the columns of A to have unit euclidean length.

We first assume that $b = (\beta, \beta, \ldots, \beta)^T$. Let $\beta \equiv 1$. Then except at the boundaries, all tiles are congruent to

$$\mathcal{T} = \{x \in \mathbf{R}^k \mid 0 \leq x^T a_j < 1\,,\ \forall\, 1 \leq j \leq k\}\,. \tag{2}$$

$\mathcal{T}$ is a parallelepiped subtended by the columns of the inverse of $\bar{A}^T$. In other words, if $F = [f_1, \ldots, f_k] \equiv \bar{A}^{-T}$, then

$$\mathcal{T} = \{x = \sum_{j=1}^{k} \alpha_j f_j \mid 0 \leq \alpha_j \leq 1\}\,. \tag{3}$$

To see this, note that $f_k^T a_j = \delta_{kj}$, so for any x that satisfies (3), $x^T a_j = \alpha_j f_j^T a_j = \alpha_j$, and since $0 \leq \alpha_j \leq 1$, equation (2) is satisfied.

Let $V(\mathcal{T})$ denote the volume of $\mathcal{T}$. Then

$$V(\mathcal{T}) = |\det(F)| = |\det(\bar{A})|^{-1}\,.$$

Let us now consider the faces of $\mathcal{T}$. Without loss of generality, consider the face $\mathcal{T}^{(1)}$subtended by $f_2, \ldots, f_k$. The face is itself a $(k-1)$ dimensional parallelepiped. We want to know its surface area, or in general its $(k-1)$ dimensional volume, which we denote $V(\mathcal{T}^{(1)})$.

Lemma 4 $V(\mathcal{T}^{(1)}) = |\det(F)| = V(\mathcal{T})$.

We give a proof, unfortunately algebraic rather than geometric in nature, in the Appendix.

What are the consequences of the lemma? we see that all the faces have the same area and that it is equal to the volume of $\mathcal{T}$. Thus, the ratio $\rho(\mathcal{T})$ of the volume to the total surface area of $\mathcal{T}$ is just the reciprocal of the number of sides, $2k$:

Theorem 5 *For any $k \times k$ matrix $\bar{A}$ with unit-length columns, the parallelepiped $\mathcal{T}$ defined by (2) has a ratio of volume to surface area of*

$$\rho = \rho(k) = \frac{1}{2k}.$$

At first this is surprising, since if $\bar{A}$ is far from having orthogonal columns we would expect a lower ratio. The explanation is that the constant ratio has been obtained because the size of $\mathcal{T}$ grows as $\bar{A}$ loses orthogonality. (Scaling up the size of any k-dimensional object by a factor ϕ increases the ratio by the factor ϕ).

The problem we have is to make the ratio ρ as big as possible subject to some limit, μ on the tile cross section. This is because, as we shall show in detail later (and it is clear intuitively), the cross section of a tile is proportional to the amount of local memory needed to execute it. The cross section of $\mathcal{T}$ is also roughly equal to $|\det(F)|$. To satisfy such a bound, we must change the size of $\mathcal{T}$. To keep the problem simple, we shall for the present consider rescaling b by a constant factor β. Let β be chosen so that the area of a face, $F(\mathcal{T})$, is exactly μ. We have that the volume and area of the rescaled tile are

$$V(\mathcal{T}) = \beta^k |\det(F)|$$

and

$$F(\mathcal{T}) = \beta^{(k-1)} |\det(F)|.$$

Thus, we must choose

$$\beta \leq (\mu \; |\det(\bar{A})|)^{1/(k-1)}.$$

We can then achieve the ratio of volume to surface area

$$\rho_{max}(\mu, \bar{A}) = \frac{(\mu \; |\det(\bar{A})|)^{1/(k-1)}}{2k}.$$

On the other hand, if we wish a ratio $\rho^\star$ of volume to surface area, we need tiles of dimension $\beta^\star = 2k\rho^\star$. Therefore, we must be able to hold tiles whose sides have area

$$\mu_{min} = \frac{(\beta^\star)^{k-1}}{|\det(\bar{A})|} \tag{4}$$

$$= \frac{(2k\rho^\star)^{k-1}}{|\det(\bar{A})|}. \tag{5}$$

We can, because of these observations, now state the optimality problem we would like to solve: Given a cone $\bar{\mathcal{C}}$, find an integer basis whose elements are in the cone. Choose them so that the matrix having the scaled basis vectors as columns has largest possible determinant. (We call the determinant of this scaled matrix the *normalized determinant*.

This problem is related to, but is not the same as, choosing $\bar{A}$ to minimize its spectral condition number under the constraints $D^T\bar{A} \geq 0$. (See [10] for the definition and properties of matrix condition numbers.) Consider the vector of singular values of $\bar{A}$. The normalization condition places it on the unit sphere in $\mathbf{R}^k$. The condition number is the ratio of the largest to the smallest component; the determinant is the product of the components. In the unconstrained case, both are optimized by the vector of equal singular values. In the constrained case, however, the optimizing matrices can differ.

Of course, for general $\bar{\mathcal{C}}$, there may be no maximizer among the integer bases in the cone. And we do not know whether there is always a maximizing choice when $\bar{\mathcal{C}}$ comes from an integer dependence matrix D.

We can view this problem as the maximization of the real valued function $|\det(\bar{A})|$ over the k^2 dimensional space of integer matrices A, subject to m linear inequality constraints $D^TA \geq 0$. It might be fruitful to use a standard method for the continuous problem and then convert the solution to integer by some rounding-off procedure; we have not pursued this approach.

In the next section, we consider a heuristic method for choosing the basis A.

3 A Procedure for Choosing the Tiling Basis

In this section, we describe a practical procedure for choosing a tiling basis A given the dependence matrix D. The procedure's complexity is a function of k, the nesting depth of the loop; m, the number of dependence directions; and p, the number of *rays* of the cone $\bar{\mathcal{C}}$. (We define what we mean by the rays of a polygonal cone below.) In these terms, the complexity of the procedure is $O(pk^2 + k^3 + m^{k-1}k^2)$. While the exponential term here may be cause for some uneasiness, the reader should keep in mind that in the practical application of these ideas k will rarely exceed four.

The procedure can be described as follows:

1. Construct the set of rays of the cone $\bar{\mathcal{C}}$. A ray is a vector $r \in \mathbf{Z}^k$ that is on the boundary of $\bar{\mathcal{C}}$ and is at the intersection of at least $k-1$ of the half spaces $\{d_j^T r = 0\}$. Thus, the rays satisfy

$$D_\Sigma r \equiv \left[d_{\sigma(1)}, d_{\sigma(2)}, \ldots, d_{\sigma(k-1)}\right]^T r = 0 \tag{6}$$

where $\Sigma = \{\sigma(1), \sigma(2), \ldots, \sigma(k-1)\}$ is a subset of the integers $\{1, 2, \ldots, m\}$. This is a necessary condition. Let us suppose that there is a unique integer solution (up to scaling) of equation (6). For the solution r to be a ray, we must check whether $D^T r \geq 0$. We also check to see whether $D^T r \leq 0$ because, if that is the case, then $-r$ is a ray of $\bar{\mathcal{C}}$. If we find that the rows of D selected by Σ are linearly dependent so that (6) has a two or more dimensional subspace of solutions, then we just ignore the set Σ.

The method we use for the construction is simply to form all of the $\begin{pmatrix} m \\ k-1 \end{pmatrix}$ subsets Σ and then solve (6) for r. Our implementation uses a QR factorization with column pivoting, which is very effective at detecting linear dependence of the columns D_Σ [10]. It is straightforward to find the integer solution to (6) by computing a solution in floating-point and then finding the smallest scalar multiple that makes the solution integer (after perturbations on the order of roundoff error). Implementations that use only integer arithmetic would also be feasible and perhaps better.

The complexity of these decompositions is $O(k^3)$. However, we may update the QR decomposition after changing one column of the matrix at cost $O(k^2)$. Bischof has recently shown how to do so and still monitor the linear independence of columns [2]. In our experiments, we do not use such an updating procedure.

We must consider the case in which D^T itself has a nontrivial null space, which in fact happens quite often. In this case, the set $\bar{\mathcal{C}}$ is a *wedge*,

$$\bar{\mathcal{C}} = \mathcal{N} \oplus \bar{\mathcal{C}}_1$$

where $\mathcal{N}$ is the null space of D^T and $\bar{\mathcal{C}}_1$ is the intersection of $\bar{\mathcal{C}}$ with the orthogonal complement of $\mathcal{N}$, the row space of D^T. To detect this case, we

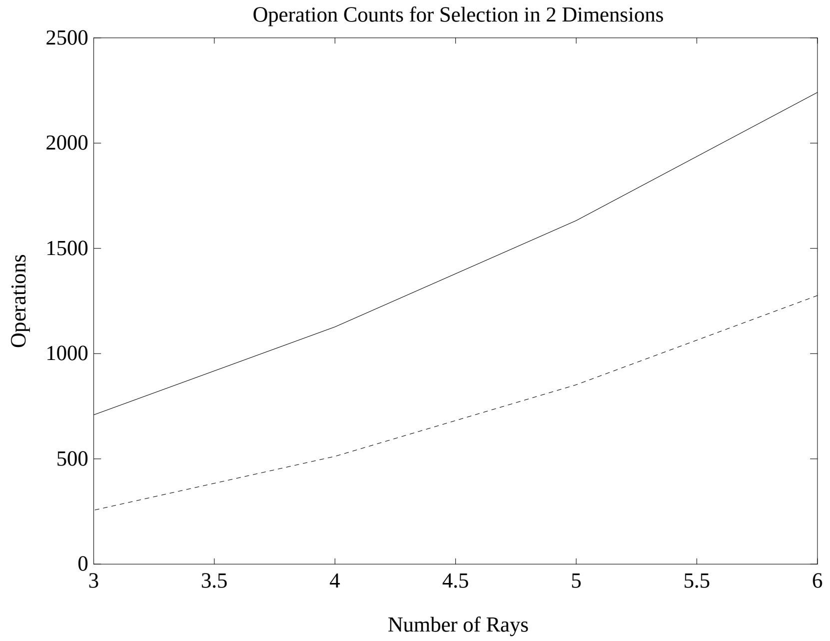

Figure 2: Operation counts versus number of rays for selection in 2 dimensions. Solid line: Subset selection; Broken line: Exhaustive search.

always start with a QR factorization of D^T itself. This allows us to find the rank of D and an integer basis for the null space of D^T in a standard manner. We then construct the rays of $\bar{\mathcal{C}}_1$ by applying a variant of the procedure above to the augmented matrix $[D, N]$, where the columns of N are the computed basis for $\mathcal{N}$. In the variant, the subsets Σ always include all of the columns of N, and enough other columns to make up a set of $k-1$. The resulting rays must therefore be members of $\bar{\mathcal{C}}_1$; together with the columns of N they are the of rays of $\bar{\mathcal{C}}$.

Having obtained the rays as the columns of a matrix $R \equiv [r_1, r_2, \ldots, r_p]$, we next choose as our first approximation to the optimal basis a subset of these rays. As the cone is invariant under scaling of these rays, we normalize them so that their length is one. Then we select a subset of k of them, chosen to approximately minimize the condition number of the subset. (We show below that this also results in a nearly maximum determinant.) This is a standard problem, called *subset selection*, in statistics. We employ the heuristic procedure of Golub, Klema, and Stewart [9], which is described in

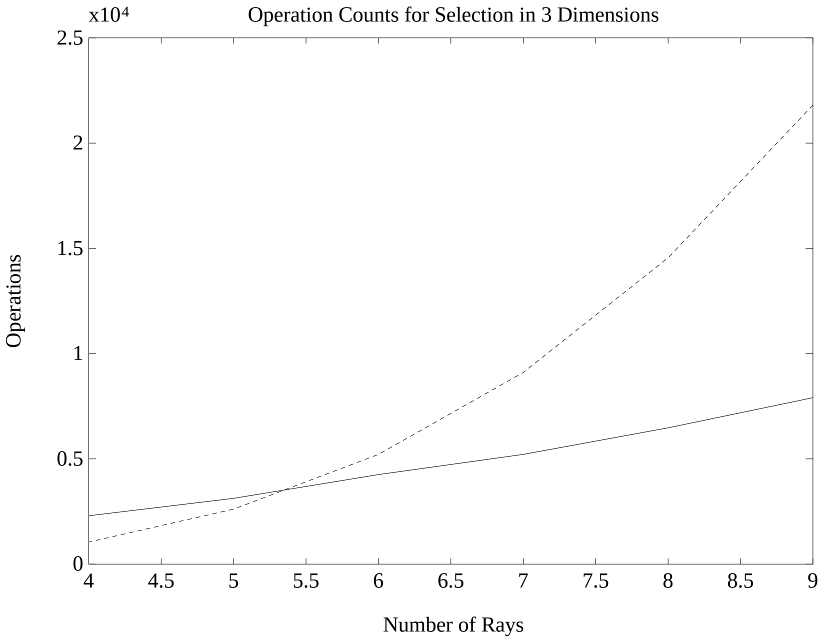

Figure 3: Operation counts versus number of rays for selection in 3 dimensions. Solid line: Subset selection; Broken line: Exhaustive search.

the text of Golub and Van Loan [10]. This procedure involves a singular value decomposition (SVD) of R and the QR decomposition with column pivoting of a matrix that is part of the SVD, with an overall cost of $rk^2 + 6k^3$ floating-point operations (and operation being a floating-point addition and a floating-point multiplication).

We know of no method for finding the optimal subset of rays other than an exhaustive search, at a cost of $\frac{1}{3}\begin{pmatrix} p \\ k \end{pmatrix} k^3$ flops. The relative costs of our implementation and exhaustive search for the optimum subset are illustrated in Figures 2 — 4. Obviously, the exhaustive procedure is prohibitively expensive for large problems, but may be used for $k = 2$, for $k = 3$ and $p < 6$, and for $k = 4$ and $p < 6$. On the other hand, subset selection does very well. In a test of 1000 randomly generated 3×6 matrices D, subset selection produced a suboptimal choice in 18 cases. In the worst of these, the determinant of the basis that it found was 17% smaller than that of the optimum basis.

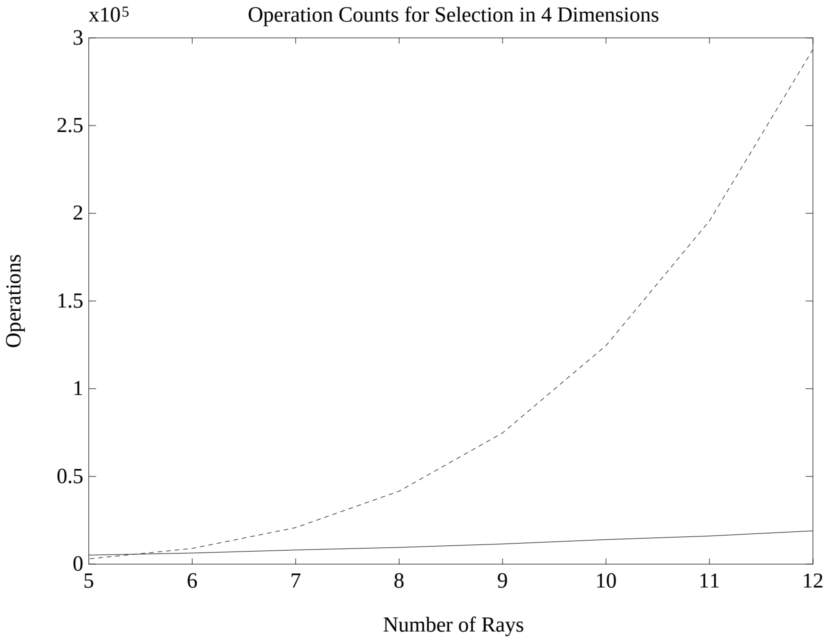

Figure 4: Operation counts versus number of rays for selection in 4 dimensions. Solid line: Subset selection; Broken line: Exhaustive search.

The basis chosen at this point may be far from optimal. Consider the case

$$D = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}.$$

The two rays of the cone are the columns of

$$R = \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix}.$$

These rays make an angle of 135°; clearly there are orthogonal bases whose elements are in $\bar{\mathcal{C}}$, but not all at the boundaries. To catch cases like this, we have implemented a generalized orthogonalization process. Let angle(x, y) denote the angle between the vectors x and y, given by

$$\text{angle}(x, y) = \arccos\left(\frac{x^T y}{(x^T x)^{1/2}(y^T y)^{1/2}}\right).$$

The procedure is

for $j = 1$ **to** k **do**
  Find $1 \le i \le k$ such that angle(a_i, a_j) is maximum;
  if angle$(a_i, a_j) > \pi/2$
    $a_j = a_j - (a_i^T a_j / a_j^T a_j) a_i$
      so that a_j is orthogonal to a_i;
    Replace a_j with an integer vector in $\bar{\mathcal{C}}$
      of approximately the same direction; **fi**
  od
if $D^T A \ge 0$ and the normalized determinant is larger than before
  improvement, accept the new A, **else** use the old one; **fi**

In a test of 1000 randomly generated 3×6 dependence matrices D, the basis selected by finding the rays of $\bar{\mathcal{C}}$ and then using the subset selection procedure above was improved by this procedure. The average determinant was improved 14%, from .63 to .71 . In comparisons with several similar procedures, this one did the best job of maximizing the normalized determinant. We also considered the following variants:

1. As above, but replace a_i rather than a_j after making it orthogonal to a_j.
2. For $j = 1$ to k, a_j is made orthogonal to each other basis vector with which it makes an obtuse angle; this continues until there are no such obtuse angles involving a_j.
3. For every pair of basis vectors a_i and a_j with $i < j$, orthogonalize a_j and a_i by adding a multiple of a_j to a_i.
4. For every pair of basis vectors a_i, and a_j with $i < j$, orthogonalize a_j and a_i by adding a multiple of a_i to a_j.

Procedures 2, 3, and 4 are more costly with little in the way of improved performance. Procedure 1 actually does worse. Thus, we recommend the use of the procedure above.

4 Other Applications

The same technique of tiling loop nests can be used in other contexts, for example:

1. The synthesis of systolic arrays. We may design an array large enough to handle a single tile of some given size; the overall computation can be performed by the small systolic array regardless of the size of the data, by tiling the index space and using the array on the individual tiles. This technique was proposed originally by Moldovan and Fortes [15], who did not specify how to choose the hyperplanes; we have filled in that gap.

2. The decomposition of work into tasks that can be executed in parallel on a shared-memory multiprocessor. This technique can find tasks of medium to large granularity that require little communication through shared memory. It is straightforward to prove that, for sufficiently large block sizes, the dependence vectors in the quotient index space are all positive. Thus, we may execute tiles simultaneously if the sum of their tile indices is equal. This approach is currently being pursued by some manufacturers of shared-

memory parallel MIMD machines. This paper enhances that technique by allowing for more effective decompositions.

5 Precise Storage and I/O Requirements

In this section, we develop formulae that give precise measures of the storage required for execution of a tile and of the number of data (input and output from local memory) required for execution of a tile. These can be used to state more precisely the optimization problem that should be solved in determining the tiling basis.

Consider I/O requirements first. Now, let E be an integer $k \times m'$ matrix whose columns represent the data required to satisfy the true dependences in an index space. Consider the loop nest

for $i = 1$ **to** n **do**
  for $j = 1$ **to** n **do**
    $a[i,j] = a[i,j-1] + b[i,j-1]$; **od**
    $b[i,j] = a[i,j-1] + b[i-1,j-1] + c[i]$; **od**
  od

In this loop nest, the dependences are

$$D = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}.$$

A given iteration requires one datum from the iteration at distance $(1\ 1)^T$ and two data from the iteration at distance $(0\ 1)^T$. Thus, the matrix E is

$$E = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}.$$

In addition to the data computed at other iterations in the index space, for which dependence directions have been established, other data may be required in order to execute a tile, for example, the c data in the example above. We express these data requirements through a second matrix, C. Each column of C corresponds to a datum (such as $c[i]$ in the example) that is used in common by a number of iterations. It gives the smallest displacement in the index space between iterations that use the datum. For the example above,

$$C = \begin{pmatrix} 0 \\ 1 \end{pmatrix},$$

since all iterations with fixed i use the value $c[i]$. If, for example, $c0[i]$ were used for $j = 0, 2, \ldots, n-1$ and $c1[i]$ were used at iterations $j = 1, 3, \ldots, n$, then we would have

$$C = \begin{pmatrix} 0 \\ 2 \end{pmatrix}.$$

We are now ready to state the I/O required to execute a tile. We assume that no data are available in local memory to begin with and that all data that may be needed later must be written back from local memory at completion of the tile's execution. Let $E = [e_r]_{r=1}^{m'}$. Let $C = [c_r]_{r=1}^{m''}$. The amount of data is given by

$$\begin{aligned} \mathrm{Data}(A, b) &= \sum_{j=1}^{k} \left[V(\mathcal{T}^{(j)}) \left(\sum_{r=1}^{m'} (2e_r^T \bar{a}_j) + \sum_{r=1}^{m''} (2c_r^T \bar{a}_j) \right) \right] \\ &= \sum_{j=1}^{k} V(\mathcal{T}^{(j)}) \left(e^T ([2E, C]^T \bar{a}_j) \right) . \end{aligned}$$

Here $V(\mathcal{T}^{(j)})$ is the volume of the face of the tile normal to the tiling basis vector a_j, $\bar{a}_j$ is a normalized tiling basis vector, and $e^T = (1, 1, \ldots, 1)$. That this is correct follows from the observation that the grid points at the face of a tile depend on values created at iteration points in a "shadow"; the shadow points are points not in the tile from which a dependence into the tile emanates. For each column d of E the corresponding shadow has as its base a face of the tile, say the face normal to a_j, and as one of its sides the vector whose direction is $-d$ and whose tail is at any corner of the face. This shadow has height $d^T \bar{a}_j$ and base area $V(\mathcal{T}^{(j)})$, so it has volume $V(\mathcal{T}^{(j)}) d^T \bar{a}_j$. The factor 2 multiplying E expresses the fact that data that are responsible for dependences must be read in and written out, while data that are used but not created are read in but not written back.

The volume of faces is explained in Section 2.3.

5.1 Choosing the Ordering of the Block Loops

A consequence of the requirement that $D^T A \geq 0$ is that the block loops may appear in any order. Suppose, without loss of generality, that

$$e^T([2E, C]^T \bar{a}_1) = \max_{j=1}^{k} e^T([2E, C]^T \bar{a}_j) .$$

Then the flux of data *per unit surface area* across the faces of the tiles normal to a_1 is greater than that across the other faces. We would choose to make the j_1 block loop innermost. This is because we would avoid storing to memory the data that flow across the faces normal to a_1 when going from one tile to the next. This has the effect, for example, of causing us to choose a "left-looking" block Gaussian elimination or block Householder QR method in preference to a "right-looking" method, which helps to reduce the memory traffic further. See the examples of Section 7 for illustration of how this technique should be applied.

5.2 Local Memory Requirements

We will make the simplifying assumption that the same computation, producing and consuming the same number of data, is done at every point of $\mathcal{S}$. The memory required to execute a tile depends on the order in which the individual points of the tile are executed. For this analysis, we assume that the points along hyperplanes normal to a given integer k-vector τ are executed simultaneously. We need to store the values produced at earlier iterations that are required by the iterations on this hyperplane. The number of such values is again given by the sum of volumes of "shadows" as

$$\mathrm{Mem}(A, b, \tau) = [\max_t V(\tau, t)] \left(e^T([2E, C]^T \bar{\tau}) \right) .$$

Here $V(\tau, t)$ is the volume of the intersection of the hyperplane $\tau^T i = t$ and the given tile, *i.e.,* of the set of iteration points computed at time t. The maximum is taken over the relevant values of t.

This largest volume is a function of the tile dimensions and of the shape of the tile as well as of the time coordinate τ. In general, it can be larger than the faces, but by no more than a constant factor (at most 2^{k-1}). It may be much smaller, as in this case: Let

$$A = \begin{pmatrix} 1 & 10 \\ 0 & 1 \end{pmatrix}$$

and let $\beta_1 = 1$ and $\beta_2 = 10$ so that the tiles are long and narrow and are nearly aligned with the i_1 axis. The faces of these tiles have lengths of 10 and about 10.5. If we take $\tau = (0\ 1)^T$, then the set of points in the tile that are simultaneously executed is small; there are at most two. On the other hand, if we choose $\tau = (1\ 0)^T$, then there are 10 such points. So our earlier assertion that face volume is a good measure of memory required is in doubt.

This is not, however, a real possibility. The example above depends on highly elongated tiles. This happens because the basis vectors (the columns of A) are close, and this in turn is due to a narrow cone $\bar{\mathcal{C}}$. But in order for τ to be used in scheduling as described above, we must have $\tau \in \mathcal{C}$. The difficulties described above are associated with a choice of τ nearly orthogonal to all of the basis vectors a, which are confined to lie in a narrow cone. Such a vector cannot also be in the cone.

6 Optimal Choice of Block Size

In this section we present solutions to two important instances of the general problem of optimal choice of the block size parameters $b = [\beta_1, \ldots, \beta_k]$. We assume that a set of tiling hyperplane normals A has been chosen and that the data fluxes E and C are known, as are the dependences D. The choice of the outermost point loop index — τ, will also play a role.

Here our viewpoint is somewhat more realistic than in Section 3. We take into account the fact that not all the data required by the execution of a tile must be read *a priori*. Instead, we consider the order in which the tiles iterations are processed and assume that the needed data are read (or written) at the time they are needed.

We need some constants to make our estimates precise. Let the amount of work per grid point be ω (the appropriate units for ω and the constants ϕ_j that follow are seconds, so that the machine characteristics are included through these constants). Let the flux of data per unit surface area across the face normal to a_j be ϕ_j. The way that ϕ_j depends on E, C, and a_j was explained in Section 5.

First we consider the case $k = 2$ with the assumption that τ is one of the two tiling vectors, say $\tau = a_1$. Then the amount of local memory required is proportional to β_2 and is independent of β_1. The total work done is $\omega\beta_1\beta_2$ and the amount of data referred to is $\phi_1\beta_1 + \phi_2\beta_2$. Thus the ratio of computation time to memory access time is

$$\rho \equiv \frac{\omega\beta_1\beta_2}{\phi_1\beta_1 + \phi_2\beta_2} \sim \frac{\omega\beta_2}{\phi_1}$$

as $\beta_1 \to \infty$. (We have redefined the dimensionless parameter ρ here). See Figure 5.

In this case, therefore, we always take β_1 as large as possible (subject only to the size of the problem being solved) and obtain the ratio shown. This ratio is the product of a ratio of work per iteration ω to data per iteration ϕ_1 and the number of grid points β_2 that fit in the local memory. Note in particular that for large problems, for which β_1 can be taken so large that the asymptote is approached, neither the data per unit surface in the direction of a_2 (that is, ϕ_2) nor the particular choice of tile length in the a_1 direction (that is, β_1) plays a role. Similar conclusions are reached if we model execution time rather than the computation to communication ratio ρ. Note also that if the ratio β_1/ϕ_2 is larger than β_2/ϕ_1, then we choose $\tau = a_2$ instead of a_1.

The discussion above is little changed if we allow arbitrary τ. What matters is that we fix all but one of the block size parameters and allow the other to grow, provided that with the given choice of τ the memory requirement is independent of this one parameter. For that to be true, all we need is that τ should not be close to a_2 rather than the much stronger requirement $\tau = a_1$.

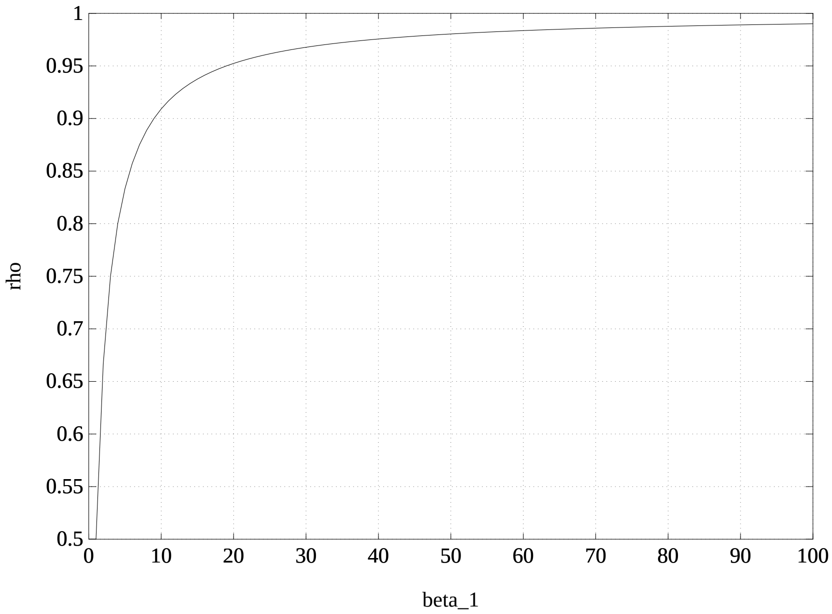

Figure 5: Reuse ratio ρ vs. tile length. Note: $\omega = \beta_2 = \phi_2 = \phi_1 = 1$

Next let us take $\tau = a_1$ and $k > 2$. Again, we fix all but one of the block size parameters, in this case $\beta_2, \ldots,$ andβ_k and allow the other one to grow, limited only by problem size.

Let $B = \prod_2^k \beta_j$. Memory size places some upper limit on B. Let the memory required per unit surface in the hyperplane normal to a_1 be M. Thus, for the given choice of the block size parameters, the local memory required is $MB/|\det(\bar{A})|$. If the available local memory has room for μ data, then B is constrained by

$$B \leq \mu \; |\det(\bar{A})|/M \, . \tag{7}$$

The amount of work per unit distance in the a_1 direction is $\omega B/|\det(\bar{A})|$.

Finally, the data required per unit distance in the a_1 direction is

$$\left(\frac{B}{|\det(\bar{A})|} \right) \sum_{j=2}^{k} \frac{\phi_j}{\beta_j} \, .$$

Thus

$$\rho = \frac{\omega}{\sum_{j=2}^{k} \phi_j / \beta_j} \, .$$

With the constraint on b given by (7), the maximizing choice of b is

$$\beta_j = \phi_j (\Phi \mu \; |\det(\bar{A})|/M)^{1/(k-1)}$$

where $\Phi \equiv \prod_{j=2}^{k} \phi_j$.

7 Blocking Examples

Our first example is an algorithm that uses plane rotations for the QR factorization of real $m \times n$ matrix X. In the description of these example algorithms we suppress all irrelevant detail. To that end, we use the notation $f(x, y, z)$ to mean a generic function of the arguments x, y, and z which may be a different function at every occurrence.

for $k = 1$ **to** n **do**
  for $i = m$ **to** $k+1$ **step** -1 **do**
(1) $(c, s) = f(x(i, k), x(i-1, k))$;
    for $j = k+1$ **to** n **do**

(2) $$\begin{bmatrix} x(i-1, j) \\ x(i, j) \end{bmatrix} = f\left(\begin{bmatrix} x(i-1, j) \\ x(i, j) \end{bmatrix} c, s\right) \text{ ; } \mathbf{od}$$

    od
  od

There are two distinct true dependences here. Statement (2) at iteration (i, j, k) depends on statement (2) at iterations $(i+1, j, k)$ (because x(i,j) is changed there) $(i-1, j, k-1)$ (because x(i-1,j) is changed there). Each iteration (i, j, k) of statment (2) depends on statement (1) at iteration (i, k), so that $(0, 1, 0)^T$ is a column of C. Furthermore, statement (1) depends on statement (2) at iterations $(i, k, k-1)$ and $(i-1, k, k-1)$. Therefore, through the uses of c and s, statement (2) depends on itself at iterations $(i, k, k-1)$ and $(i-1, k, k-1)$; this dependence is weaker that a dependence on iteration $(i, j-1, k-1)$ and $(i-1, j-1, k-1)$, so if we take these to be the actual dependences we are going to be safe. There are also antidependences and output dependences, but these can be ignored for the moment. Thus,

$$D = \begin{pmatrix} -1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}$$

and

$$C = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

In this case, there are only three rays of the cone, namely,

$$\begin{pmatrix} 0 & 0 & -1 \\ 1 & -1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

After improvement we arrive at the basis

$$\begin{pmatrix} 0 & 0 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

Thus, the new indices are j, k, and $k-i$.

After replacing the index i by $r \equiv k-i$ we have the following program:

$$
\begin{array}{l}
\textbf{for } k = 1 \textbf{ to } n \textbf{ do} \\
\quad \textbf{for } r = k-m \textbf{ to } -1 \textbf{ do} \\
\quad\quad (c(r,k), s(r,k)) = f(x(k-r,k), x(k-r-1,k)); \\
\quad\quad \textbf{for } j = k+1 \textbf{ to } n \textbf{ do} \\
\quad\quad\quad \left[\begin{array}{c} x(k-r-1,j) \\ x(k-r,j) \end{array} \right] = f\left(\left[\begin{array}{c} x(k-r-1,j) \\ x(k-r,j) \end{array} \right] c(r,k), s(r,k) \right) \text{ ; } \textbf{od} \\
\quad\quad \textbf{od} \\
\quad \textbf{od}
\end{array}
$$

Strip mining produces

$$
\begin{array}{l}
\textbf{for } k0 = 1 \textbf{ to } n \textbf{ step } b \textbf{ do} \\
\quad \textbf{for } k = k0 \textbf{ to } \min(n, k0+b-1) \textbf{ do} \\
\quad\quad \textbf{for } r0 = k0-m \textbf{ to } -1 \textbf{ step } b \textbf{ do} \\
\quad\quad\quad \textbf{for } r = \max(r0, k-m) \textbf{ to } \min(-1, r0+b-1) \textbf{ do} \\
\quad\quad\quad\quad (c(r,k), s(r,k)) = f(x(k-r,k), x(k-r-1,k)); \\
\quad\quad\quad\quad \textbf{for } j0 = k0+1 \textbf{ to } n \textbf{ step } b \textbf{ do} \\
\quad\quad\quad\quad\quad \textbf{for } j = \max(k+1, j0) \textbf{ to } \min(n, j0+b-1) \textbf{ do} \\
\quad\quad\quad\quad\quad\quad \left[\begin{array}{c} x(k-r-1,j) \\ x(k-r,j) \end{array} \right] = f\left(\left[\begin{array}{c} x(k-r-1,j) \\ x(k-r,j) \end{array} \right] c(r,k), s(r,k) \right) \text{ ; } \textbf{od} \\
\quad\quad\quad\quad\quad \textbf{od} \\
\quad\quad\quad\quad \textbf{od} \\
\quad\quad\quad \textbf{od} \\
\quad\quad \textbf{od}
\end{array}
$$

```
    od
```

Then loop interchanging produces

for $k0 = 1$ **to** n **step** b **do**
  for $r0 = k0 - m$ **to** -1 **step** b **do**
    for $j0 = k0$ **to** n **step** b **do**
      for $k = k0$ **to** $\min(n, k0 + b - 1)$ **do**
        for $r = \max(r0, k - m)$ **to** $\min(-1, r0 + b - 1)$ **do**
          if $j0 = k0$ **then** $(c(r,k), s(r,k)) = f(x(k-r,k), x(k-r-1,k))$;
          for $j = \max(k+1, j0+1)$ **to** $\min(n, j0 + b - 1)$ **do**
            $\begin{bmatrix} x(k-r-1,j) \\ x(k-r,j) \end{bmatrix} = f\left(\begin{bmatrix} x(k-r-1,j) \\ x(k-r,j) \end{bmatrix} c(r,k), s(r,k)\right)$; **od**
          od
        od
      od
    od
  od

This rather complicated blocked algorithm works as follows. We illustrate with $m = 20$, $n = 15$, $b = (5, 5, 5)$. Elements of X are eliminated by plane rotations in patches, as shown in Figure 6. The values of $k0$ and $r0$ at which elements are eliminated is shown in each patch. The rotations used to do the elimination are applied only to columns in the current patch (during the block operation with $j0 = k0$). These rotations are stored and later applied to columns to the right of the patch (when $j0 > k0$).

For another example, consider the following program for the QR factorization which uses Householder transforms rather than plane rotations. In this pseudo-code we use the notation $x(k : m, j)$ to refer to the vector of elements $[x(k,j),\ x(k+1,j)\ldots,\ x(m,j)]$. We include it as an example of a program that can be blocked without using linear loop index transformation.

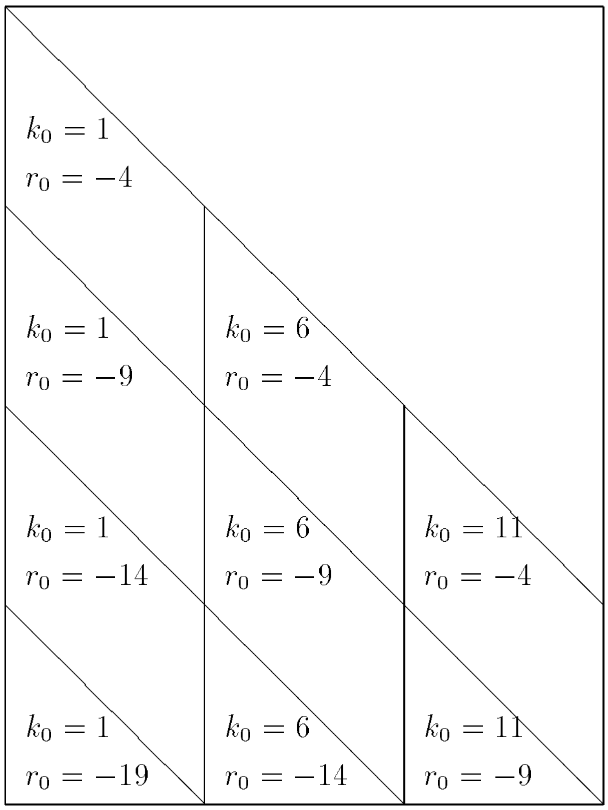

Figure 6: Blocking of the QR factorization of an 20×15 matrix with $\beta = 5$.

for $k = 1$ **to** n **do**
  $s(k) = \|x(k:m,k)\|$;
  $x(k,k) = f(x(k,k), s(k))$;
  for $j = k+1$ **to** n **step** **do**
    $s'(k,j) = f(s(k), x(k:m,k)^T x(k:m,j))$;
    $x(k:m,j) = x(k:m,j) + s'(k,j) * x(k:m,k)$; **od**
  od

In Fortran, loops would be triply nested. The compiler, on detecting a dependence of some subsequent statement on the whole of an inner loop implementing a reduction operation, such as the norm and the inner product in the example, should choose to view those loops as atomic and therefore work with an index space of reduced dimension.

The dependences in (j,k) space are

$$D = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

The basis chosen is the obvious one: $A = I$. Thus, no skewing is done.

Now, we choose the order of the block loops. The measure of data flux given in Section 5 is the same for a_2 and for a_1; so neither order is preferred.

Note, however, that the two dependences are different in character. The $(0,1)^T$ dependence is a true dependence at every point of the index space. The other, $(1,0)^T$ expresses the dependence of iteration (j,k) on "iteration" (k,k) (the task performed outside the inner loop for given k); the data that are required are used in common by all the iterations with fixed k. Thus, for the purpose of determining data flux, this dependence direction should be given weight 1 (as are columns of C), not 2. If we make this change, the flux is greater for a_2, so we make the k block loop innermost. This procedure yields a left-looking method in which all groups of Householder transforms are applied to a block of columns just before that block is triangularized. This allows the block to be held in local memory during the application of these transforms.

Acknowledgement We would like to thank Ilse Ipsen for her help at the beginning of our work on this problem and Mike Wolfe for his at the end.

Appendix. Proof of Lemma 3.

Let the $k \times k-1$ matrices $F_1 = [f_2, \ldots, f_k]$ and $A_1 = [a_2, \ldots, a_k]$. The face $\mathcal{T}^{(1)}$ is subtended by the columns of F_1. Let the matrix F_1 be factored

$$F_1 = QR \tag{8}$$

$$= [Q_1\ q_1] \begin{bmatrix} R_1 \\ 0 \end{bmatrix}. \tag{9}$$

where Q is an orthogonal matrix, R is an upper triangular matrix, Q_1 is $k \times k-1$, and R_1 is $k-1 \times k-1$; thus $F_1 = Q_1 R_1 = QR$, and q_1 is a unit vector in the direction normal to the range of F_1, which is the span of $\{a_1\}$. The matrix $P_1 \equiv Q_1 Q_1^T$ is the orthogonal projector on $\{a_1\}^\perp$. The factorization (9) always exists and is unique up to signs on the diagonal of R [10].

The columns of R_1 are the coordinates of the columns of F_1 with respect to the orthonormal basis (for the subspace of $\mathbf{R}^k$ in which $\mathcal{T}^{(1)}$ lies) consisting of the columns of Q_1. Thus

$$V(\mathcal{T}^{(1)}) = |\det(R_1)|.$$

We must therefore show that $|\det(R_1)| = |\det(F)| = |\det(A)|^{-1}$.

From the identity $I = F^T A$ it follows that $I_{k-1} = F_1{}^T A_1 = R_1^T Q_1^T A_1$; thus

$$|\det(R_1)| = |\det(Q_1^T A_1)|^{-1}.$$

The proof is complete if we can show that $\det(Q_1^T A_1) = \det(A)$. But since $Q_1^T Q_1 = I_{k-1}$,

$$\det(Q_1^T A_1)^2 = \det([A_1{}^T Q_1 Q_1^T][Q_1 Q_1^T A_1])$$

$$= \det([P_1\ A_1\ ]^T[P_1\ A_1\ ])\,.$$

The result now follows from Lemma 1. ∎

References

[1] W. A. Abu-Sufah, D. J. Kuck, and D. H. Lawrie. On the performance enhancement of paging systems throught program analysis and transformations. *IEEE Transactions on Computers*, C-30:341–356, 1981.

[2] Christian H. Bischof. *Incremental condition estimation.* Technical Report ANL-MCS-P15-1088, Argonne National Laboratory, 1989.

[3] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation of subscripted variables. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, Association for Computing Machinery, 1990.

[4] Steve Carr and Ken Kennedy. Blocking linear algebra codes for memory hierarchies. In *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing*, Society for Industrial and Applied Mathematics, 1989.

[5] James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, and Danny Sorensen. *Prospectus for the development of a linear algebra library for high-performance computers.* Technical Report, Argonne National Laboratory, 1987.

[6] J.J. Dongarra and D.C. Sorensen. Linear algebra on high-performance computers. In U. Schendel, editor, *Proceedings of Parallel Computing 85*, pages 3–32, JACK: WHAT PUBLISHER?, 1986.

[7] K.A. Gallivan, R.J. Plemmons, and A.H. Sameh. Parallel algorithms for dense linear algebra computations. *SIAM Review*, 32(1):54–135, 1990.

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

[9] G. H. Golub, V. Klema, and G. W. Stewart. *Rank degeneracy and least squares problems.* Technical Report TR-456, Department of Computer Science, University of Maryland, 1976.

[10] Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* Johns Hopkins, Baltimore, MD, Second edition, 1989.

[11] F. Irigoin and R. Triolet. Supernode partitioning. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 319–329, Association for Computing Machinery, 1988.

[12] Ken Kennedy. Talk at the fourth SIAM conference on parallel processing for scientific computing. Chicago, Illinois, 1989.

[13] Leslie Lamport. The parallel execution of do loops. *Communications of the Association for Computing Machinery*, 17:83–93, 1974.

[14] H. Lomax and T. H. Pulliam. A three-dimensional implicit code for the ILLIAC IV. In Garry Rodrigue, editor, *Computational Physics on Parallel Computers*, Academic Press, New York, NY, 1982.

[15] Dan I. Moldovan and Jose A. B. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Transactions on Computers*, C-36:1–12, 1986.

[16] Robert Schreiber. Block algorithms for parallel machines. In *Numerical Algorithms for Modern Parallel Computer Architectures*, pages 197–208, Springer-Verlag, New York, NY, 1988.

[17] Michael E. Wolf and Monica S. Lam. *An algorithm to generate sequential and parallel code with improved data locality.* Technical Report, Computer Systems Labortory, Stanford University, 1989.

[18] Michael Wolfe. Iteration space tiling for memory hierarchies. In Garry Rodrigue, editor, *Parallel Processing for Scientific Computing*, pages 357–361, Society for Industrial and Applied Mathematics, 1989.

[19] Michael Wolfe. More iteration space tiling. In *Proceedings Supercomputing '89*, pages 655–664, Association for Computing Machinery, 1989.