Near-Optimal Allocation of Local Memory Arrays

Robert Schreiber, Darren C. Cronquist
HP Laboratories Palo Alto

HPL-2004-24
February 17, 2004*

scratch pad memory, affine reference

This paper considers compiler management of fast, local memories for loop nests that contain affine array references. We show how to reduce the local memory footprint of such arrays by doing an affine reindexing of the array elements. We approximate the best reindexing by a two-step algorithm. The first step uses a generalized Hermite factorization, and the second uses a one-norm lattice basis reduction technique. We improve on earlier work in which a local memory array stores the smallest rectangular subarray of the elements referenced in the loop nest. Our techniques apply directly to nests with families of uniformly generated affine references, and we propose ways to extend them to more general situations.

Near-Optimal Allocation of Local Memory Arrays

Robert Schreiber and Darren C. Cronquist

Hewlett-Packard Laboratories
1501 Page Mill Road, M/S 1177
Palo Alto, California 94304-1126

January 16, 2004

Abstract

This paper considers compiler management of fast, local memories for loop nests that contain affine array references. We show how to reduce the local memory footprint of such arrays by doing an affine reindexing of the array elements. We approximate the best reindexing by a two-step algorithm. The first step uses a generalized Hermite factorization, and the second uses a one-norm lattice basis reduction technique. We improve on earlier work in which a local memory array stores the smallest rectangular subarray of the elements referenced in the loop nest. Our techniques apply directly to nests with families of uniformly generated affine references, and we propose ways to extend them to more general situations.

1 Introduction

In any modern high-performance computer, the bandwidth of the processor-memory channel is generally much less than what would be usable by the processor. This "memory wall" has for quite some time been understood to be a central problem in fast computation [8, 7]. Designers of general-purpose machines craft a cache hierarchy and parallel load/store units in order to overcome it. Cache is a purely hardware mechanism that presents the standard programmer's model of a single logical memory holding all data.

In designing a computer system for a particular purpose, as is often the case with computers embedded in some other device, the architect may choose to use a special memory hierarchy, designed for the specific purpose, in order to maximize performance without unduly increasing cost. He may introduce small, fast, logically separate local memories. These extra memories, controlled by software (unlike cache), allow data structures to be placed so as to avoid conflicts for cache, to minimize latency for loads in a critical path, and for increasing the number of paths to memory and hence the memory bandwidth. Digital signal processors (DSP's) and application-specific processors owe some of their performance advantages to the use of local memory. Their use, however, is a challenge for programmers and compiler writers.

In this paper, we consider some problems that arise when a compiler, presented with an architecture having a limited amount of fast local memory, must decide what data structures to move into this memory, and at what times to move them, and where to place them. These same problems come up when a design tool synthesizes an architecture for a specific application. Such tools should include the capability to craft and optimally exploit local memories, and the problems and solutions we provide can be useful in doing this.

Specifically, we consider the compilation of a loop nest in which there are one or more uniformly generated affine references to an array. Such loop nests are common in embedded applications. We consider these questions:

1. How much storage in the local memory is required to hold these elements? In other words, what subset of the elements of the array should be moved into the local memory?

2. Where should these elements be placed in local memory, and how should the code be modified in order to address them?

3. What is the best form for the code needed to move the data between the local and global memory?

An example will be useful at this point.

```
int A[200, 200], B[300], C[100, 100];
for(i = 0; i < 100; i++)
    for(j = 0; j < 100; j++)
        A[i+j, i-j+100] = B[i+j+100] + C[i, i]
```

We would like to identify the sets of elements referenced by the 10,000 iterations of the loop nest. Each of the three arrays exhibits an interesting possibility. We shall denote by $\mathcal{T}(X)$ the subset of the elements of X accessed in the loop nest.

- $\mathcal{T}(B)$ is shifted into the interior of B.

- $\mathcal{T}(C)$ is one-dimensional, even though C is two-dimensional. Even though only 100 elements of C are touched, the smallest rectangle containing $\mathcal{T}(C)$ is all of C.

- $\mathcal{T}(A)$ has *holes*. A checkerboard pattern of elements is touched. The shape of $\mathcal{T}(A)$ is a rhombus.

We would like to allocate the smallest possible amount of local memory to each of the touched arrays, but subject to the following constraint: access to the local-memory array should still be affine. For example, for C, we would allocate a local-memory array C_{loc} of size 100, move C[i, i] into C_{loc}[i], and replace the reference in the loop body to C with C_{loc}[i]. Similarly, B[k] should be moved to B_{loc}[k-100] for $100 \leq k \leq 298$, and the reference replaced by B_{loc}[i+j]. These two optimizations are obvious. Of the 40,000 elements of A, some 10,000 are in $\mathcal{T}(A)$. Our algorithm would allocate an array A_{loc}[100, 100] and would store A[i+j, i-j+100] in A_{loc}[i, j]. Clearly, in simple cases like these, simple and *ad hoc* methods may suffice. However, we present a general technique that handles all these sorts of issues in a unified and, as we shall show, a near-optimal way. For an example of a more complicated problem, consider the reference A[i+j, i+3j+4k] in a three-deep nest with loop extents [10, 100, 40]. While $\mathcal{T}(A)$ has only 9424 elements, it spreads over a rectangular subarray of size 109 by 463; we compress it into a local-memory array B of size 178 by 88, in which B[j+2k, -i+2k+9] holds A[i+j, i+3j+4k].

2 Prior work

Earlier work of Gallivan, Jalby, and Gannon [3], Eisenbeis *et al.* [2], and Anantharaman and Pande [1] addressed some of these questions.

The Gallivan paper posits the same memory hierarchy as we, and the same problems associated with it. It is concerned mainly with optimizing the process of transferring data from global to local memory. Its primary contribution is a data-transfer method that runs in time proportional to the size of $\mathcal{T}(A)$. It uses the Hermite factorization of the iteration-index to array-index map to accomplish this. We begin with this same factorization; we then extend the Gallivan work in several ways. Most notable is that we also optimize the size of the local-memory array that is used to hold $\mathcal{T}(A)$. Furthermore, we use a second step, a unimodular change of lattice basis, to make a further reduction beyond that provided by the Hermite factorization.

The paper of Eisenbeis and her colleagues looks, as we do, at ways to reduce the local memory requirement by changing the affine access function. But where we take the whole loop nest as the atomic grain of computation from the viewpoint of deciding what to move into local memory, they take the individual iteration. Hence, their concern is to characterize the *window* of elements that are live (have been touched, but not for the last time) at a given iteration and to bound the size of the window with an upper bound independent of iteration.

The Eisenbeis paper also includes an interesting application of the knapsack problem for deciding, when local memory size is fixed, which arrays to promote to the local memory.

Anantharaman and Pande try to optimize the size of the local-memory array used to hold $\mathcal{T}(A)$, but only by enclosing $\mathcal{T}(A)$ in a bounding rectangle.

3 Notation and Theoretical Background

We denote the real numbers by $\mathcal{R}$, the integers by $\mathcal{Z}$, and the positive integers by $\mathcal{Z}_+$. For a set $\mathcal{S}$, its cardinality is written $|\mathcal{S}|$. For a set $\mathcal{S} \subseteq \mathcal{R}$, we write glb $\mathcal{S}$ for its greatest lower bound.

We use upper case Roman for matrices, lower case Roman for vectors, except that the lower case letters i, j, k, m, n, p, q are used for integer indices. We write A^t for the transpose of A. For a matrix A, its k^{th} row is written A_k. For a vector x, its k^{th} element is x_k. Inequalities and absolute value apply to matrices and vectors elementwise; so for example, for n-vectors x and y, the notation $x \le y$ means $x_i \le y_i$ for all $i = 1, \dots, n$. The vector whose elements are all one is written $\mathbf{1}$. The diagonal matrix whose diagonal elements are the elements of the vector x is denoted $\mathcal{D}(x)$. The identity matrix of order n is $I_n \equiv \mathcal{D}(\mathbf{1})$.

We use n for the depth of the loop nest and consider perfectly nested rectangular loops. The loop indices are represented by the vector $z = [z_1, \dots, z_n]$ and the loop trip counts by the vector $\hat{z} \in \mathcal{Z}_+^n$. We assume the loop nest is normalized such that $0 \le z \le \hat{z}$, and we define the iteration space of the loop nest to be

$$\mathcal{I} \equiv \{z \mid 0 \le z \le \hat{z}\}\,. \tag{1}$$

We use m for the dimensionality of the array and consider affine array references. We denote elements of m-dimensional arrays using multi-indices (e.g. $A[a]$ where $a \in \mathcal{Z}^m$). An affine reference to array A is written $A[Fz + f]$ where $F \in \mathcal{Z}^{m \times n}$ and $f \in \mathcal{Z}^m$.

In the style of Matlab, we denote by [X, Y] the horizontal concatenation of two matrices with the same number of rows, by [X;Y] the vertical concatenation of matrices with the same number of columns, and by $X(p:q,:)$ the submatrix of X consisting of rows p through q.

For $\mathcal{S} \subseteq \mathcal{R}^n$, we denote by $F\mathcal{S}$ the image of $\mathcal{S}$ under F, and by $F\mathcal{S} + f$ the set $\{Fz + f \mid z \in \mathcal{S}\}$.

For a matrix X with n (not necessarily linearly independent) columns, the set $\{Xv \mid v \in \mathcal{Z}^n\}$ is a lattice that we denote $\mathcal{L}(X)$.

We define $\mathcal{T}(A)$ to be the set of elements touched by a reference $A[Fz + f]$.

Given an iteration space $\mathcal{I}$, $\mathcal{T}(A) = F\mathcal{I} + f$. $\mathcal{T}(A)$ is also the intersection of the shifted lattice $\mathcal{L}(F) + f$ with the polytope $\{Fx + f \mid x \in \mathcal{R}^n,\ 0 \leq x \leq \hat{z}\}$.

Any integer matrix F has a generalized Hermite normal form

$$F = \tilde{H}U$$

where $\tilde{H}$ is an $m \times n$ lower triangular matrix of rank r, and U is $n \times n$ and unimodular. In particular, F has a generalized Hermite factorization,

$$F = [H, 0]U = HV,$$

where H is $m \times r$ and has full column rank, 0 represents the $m \times (n - r)$ zero matrix, and $V = U(1 : r, :)$ consists of the first r rows of U. Note that $\mathcal{L}(F) = \mathcal{L}(\tilde{H}) = \mathcal{L}(H)$.

For any real number $p \geq 1$, the p-norm of the n-vector x is defined by

$$\|x\|_p \equiv \left(\sum_{k=1}^{n} |x_k|^p \right)^{(1/p)} .$$

In particular, the 1-norm, which is sometimes called the "Manhattan" norm, is

$$\|x\|_1 \equiv \sum_{k=1}^{n} |x_k| .$$

4 Local memory size minimization via array reindexing

We are given a loop nest with one or more affine references to an array A. The goal is to find the smallest array B that stores the elements of $\mathcal{T}(A)$ and has an affine reindex mapping – all manner of important compiler analyses and the dependent optimizations, not the least of which is parallelism, depend on the affinity of array references. We also need to generate the code to move the elements of $\mathcal{T}(A)$ into B prior to the loop nest and the modified elements back to A afterwards.

4.1 The size of the local-memory array with one affine reference

We first consider how to determine the size of the array required to hold the elements of a conceptually infinite array X referenced as $X[Ez + e]$ where $E \in \mathcal{Z}^{m \times n}$. Following Pande, we assume that the part of X moved to local memory will be the smallest rectangular subarray of X that contains $\mathcal{T}(X)$. Let us denote by box(S) the smallest rectangular subset of $\mathcal{Z}^n$ that contains the bounded set $S \subseteq \mathcal{Z}^n$. The set of elements of X whose array indices are in box$(\mathcal{T}(X)) =$ box$(E\mathcal{I} + e)$ are stored in local memory.

We can calculate the size of the bounding rectangle exactly. The result is stated in Corollary 4.1. To begin with, the box is a Cartesian product of intervals:

$$\text{box}(E\mathcal{I} + e) = \bigotimes_{k=1}^{m} [a_{\min}(k), a_{\max}(k)]$$

where $[x, y]$ denotes the integer interval $\{x, x+1, \ldots, y\}$, $\bigotimes$ denotes Cartesian product of sets, and

$$a_{\text{min}}(k) = \min_{0 \le z \le \hat{z}} (Ez)_k + e_k$$

and similarly for $a_{\text{max}}(k)$.

Lemma 4.1 The length of the k-th side of box$(E\mathcal{I} + e)$ is

$$|[a_{\text{min}}(k), a_{\text{max}}(k)]| = a_{\text{max}}(k) - a_{\text{min}}(k) + 1 = |E_k|\hat{z} + 1. \qquad (2)$$

Proof: Obvious.

We make an approximation and drop the added one. While this is not strictly necessary, it simplifies notation considerably. $\tilde{E} = E\mathcal{D}(\hat{z})$ is the matrix whose j^{th} column is the corresponding column of E scaled by the j^{th} loop bound $\hat{z}_j$. Note that for any matrix X and any diagonal matrix D, $|X||D| = |XD|$.

Corollary 4.1 If $\mathcal{I}$ satisfies (1), then

$$\begin{aligned} |\text{box}(E\mathcal{I} + e)| &\approx \prod_{k=1}^{m} (|E_k|\hat{z}) \\ &= \prod_{k=1}^{m} |E_k|\mathcal{D}(\hat{z})\mathbf{1} \\ &= \prod_{k=1}^{m} |E_k\mathcal{D}(\hat{z})|\mathbf{1} \\ &= \prod_{k=1}^{m} \|(E\mathcal{D}(\hat{z}))_k\|_1 \end{aligned}$$

4.2 The mathematical framework for reindexing

We develop reindexing as a two step process. The first is algebraic and the second is geometric. This section and Section 4.3 address the algebraic step, while Section 4.5 discusses the geometric step.

The purpose of the algebraic step is to deal with the fact that $\mathcal{L}(F) + f$ may leave out a lot of the elements of A which do not have to be stored. The dimension of the shifted lattice is $r = \text{rank}(F)$, which may be less than m, and we can use an array B of dimensionality r to hold the elements needed. The basic idea is to represent the element of A whose index is $a = Fz + f$ by the coordinates of $Fz = a - f$ with respect to the basis given by the columns of H, where $F = HV$ (from the generalized Hermite factorization). So we make the first reindexing, in which

$$C[Vz] \quad \text{stores} \quad A[Fz + f]\,.$$

Note that the dimensionality of C is indeed r, which may be less than and cannot be more than m.

To establish the map from indices of C to those of A we observe that

$$C[c] \quad \text{stores} \quad A[Hc + f].$$

To go from index a in A to the corresponding index in C we require that $a - f$ be in $\mathcal{L}(F)$, otherwise this entry of A does not exist in C. For any vector $w \in \mathcal{L}(F)$, $w = HH^{-1}w$ where H^{-1} is any left inverse of H (see Section A for how one can test $x \in \mathcal{L}(X)$).

Thus:

$$A[a] \quad \begin{cases} \text{is stored in } C[H^{-1}(a - f)] & \text{if } (a - f) \in \mathcal{L}(H) \\ \text{is } \textit{not} \text{ stored in } C & \text{otherwise.} \end{cases}$$

The local-memory array C corresponds to a bounding rectangle around the set of elements $V\mathcal{I}$. Not all elements of C are members of $V\mathcal{I}$. For any element $c \notin V\mathcal{I}$, the corresponding $a = Hc + f$ is not in $F\mathcal{I} + f$, and moreover it may not be within the bounds of A. Determining whether $c \in V\mathcal{I}$ is more difficult than checking whether $a = Hc + f$ is within the constant bounds of A. Thus, to load entries into C we would first determine the bounds on C then iterate over its entries c and move $A[Hc + f]$ into $C[c]$ when this element of A is within the array bounds. If there is a left-hand side occurrence in the nest, we must write back at least the set of "dirty" elements on completion. One can keep track of whether or not an entry is modified, or simply write back all entries for which the A index in is bounds.

4.3 Several uniformly generated affine references

We now discuss local memory allocation for several *uniformly generated affine* (UGA) references to A. UGA references have the same linear part F but can have different constant offsets. Following Anantharaman and Pande [1], we classify references according to whether or not they can alias, and allocate a local-memory array to each group of potentially aliasing references.

Given two such references, with constant parts f_1 and f_2, we say that the two references are *alias equivalent* if $f_1 - f_2 \in \mathcal{L}(F)$. This is indeed an equivalence relation on the set of integer m-vectors.

Lemma 4.2 Two UGA references can access the same array element only if they are alias equivalent.

Proof: If there are two iterations with loop index vectors z_1 and z_2 such that $Fz_1 + f_1 = Fz_2 + f_2$, then $F(z_1 - z_2) = f_2 - f_1$, whence $f_2 - f_1 \in \mathcal{L}(F)$ and the two references are alias equivalent. ∎

We create a separate local-memory array for each equivalence class that occurs in the loop body. If the offsets f are constant, the mapping of references to equivalence classes can be determined at compile time.

For each equivalence class, we choose one of the references in the class to be the *dominant* reference, and call all the others *subservient* references. The reindex map becomes

$$C[Vz] \quad \text{stores} \quad A[HVz + f]$$

where f is the constant vector of the dominant reference. For a subservient reference $A[Fz + f_1]$ in the same equivalence class, we want to find the r-vector u_1 such that $C[Vz + u_1] = A[Fz + f_1]$. We call u_1 the projected offset. To see how to obtain u_1 from f_1, note that $f_1 - f \in \mathcal{L}(F)$, so $f_1 - f = Fz_1 = HVz_1 = Hu_1$ where $u_1 = H^{-1}(f_1 - f)$. We therefore replace the reference $A[HVz + f_1]$ with the reference $C[Vz + H^{-1}(f_1 - f)] \equiv C[Vz + u_1]$.

The local-memory array grows when there are several affine references. For each of these, there is an offset $(f_k - f)$ from the offset of the dominant reference, and a projected offset u_k for the reindexed reference. Let $\mathcal{O}$ be the set of these projected offset vectors. Then the extents of box$(\mathcal{O})$ must be added to the extents of box$(V\mathcal{I})$ to determine the size of the smallest rectangular subarray of C that contains the union of the sets of elements touched by this equivalence class of references.

4.4 Examples

Example 1. $A[2i]$ in a one-deep loop nest. $F = [2] = [2][1] = \tilde{H}U$. $C[i]$ holds $A[\tilde{H}i] = A[2i]$. Replace the reference in the code by $C[i]$.

Example 2. $A[2z_1 + 4z_3, z_1 + 2z_2]$ in a nest indexed by (z_1, z_2, z_3).

$$F = \begin{pmatrix} 2 & 0 & 4 \\ 1 & 2 & 0 \end{pmatrix} = \tilde{H}U = \begin{pmatrix} 2 & 0 & 0 \\ 1 & 2 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 2 \\ 0 & 1 & -1 \\ 0 & 1 & 0 \end{pmatrix}$$

So that

$$V = \begin{pmatrix} 1 & 0 & 2 \\ 0 & 1 & -1 \end{pmatrix}.$$

We replace the reference to $A[Fz + f]$ with a reference to $C[Vz] = C[z_1 + 2z_3, z_2 - z_3]$. Note that the sparse set of entries of A that are referenced is compacted into C. Note also that $|\text{box}(\mathcal{T}(A))| = (2\hat{z}_1 + 4\hat{z}_3)(\hat{z}_1 + 2\hat{z}_2)$, while $|\text{box}(\mathcal{T}(C))| = (\hat{z}_1 + 2\hat{z}_3)(\hat{z}_2 + \hat{z}_3)$; usually the latter is smaller, but not always! However, the second phase of reindexing, which is described in Section 4.5, takes loop extents into account and will produce a better result.

Example 3. We have a reference $A[z_1 + 2z_2, 2z_1 + 4z_2]$. Thus,

$$\begin{aligned} F &= \begin{pmatrix} 1 & 2 \\ 2 & 4 \end{pmatrix} \\ &= \tilde{H}U \\ &= \begin{pmatrix} 1 & 0 \\ 2 & 0 \end{pmatrix}\begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}. \end{aligned}$$

Here $r = \mathrm{rank}(F) = 1$, $H = [1; 2]$, and $V = [1, 2]$. We store the accessed elements of A in the one-dimensional array C; if $f = [f_1; f_2]$ then $C[k]$ holds $A[k + f_1, 2k + f_2] = A[Hk + f]$. The reference to $A[Fz + f]$ in the code is replaced by the reference $C[Vz] = C[z_1 + 2z_2]$. If in addition the code contains the subservient reference $A[z_1 + 2z_2 + 3, 2z_1 + 4z_2 + 6]$, then we first find the projected offset u_1 (the coordinates of $(f_1 - f) = f_1 = [3; 6]$ in the basis H), $u_1 = H^{-1}(f_1 - f)$, which in this case is

$$\begin{pmatrix} 1 & 0 \end{pmatrix}\begin{pmatrix} 3 \\ 6 \end{pmatrix} = 3.$$

We replace the subservient reference with $C[z_1 + 2z_2 + 3]$.

4.5 Unimodular reindexing for reducing the bounding rectangle

This section describes the geometric step of the reindexing process. We have so far replaced references to A with references to C, where the correspondence is between $C[Vz]$ and $A[HVz + f]$. The size of C is determined by the size of the smallest rectangular region containing the image of the iteration space $\mathcal{I}$ under V, $\mathrm{box}(V\mathcal{I})$. It will often be the case that $V\mathcal{I}$ lies in a parallelepiped that is smaller than this smallest rectangle. This is because the parallelepiped is skewed with respect to the axes of C. A change of lattice basis from V can take advantage of this and reduce storage requirements.

Revisiting Example 2, let $\hat{z} = [5; 10; 15]$. The bounding rectangle of $\mathcal{T}(A)$ contains 1846 elements, even though only 456 are referenced (Figure 1a). The Hermite reindexing yields a local-memory array C having 936 entries (Figure 1b). $\mathcal{T}(A)$ is a subset of a sparse sublattice while $\mathcal{T}(C)$ is compact but skewed rather badly. To reduce this skew, we make another change of variables and introduce a new local-memory array B, with the relation

$$B[Gc] = C[c] \quad \text{and} \quad C[G^{-1}b] = B[b].$$

where G is a unimodular matrix of order r. Then we have

$$B[Gc] = C[c] = A[Hc + f]$$

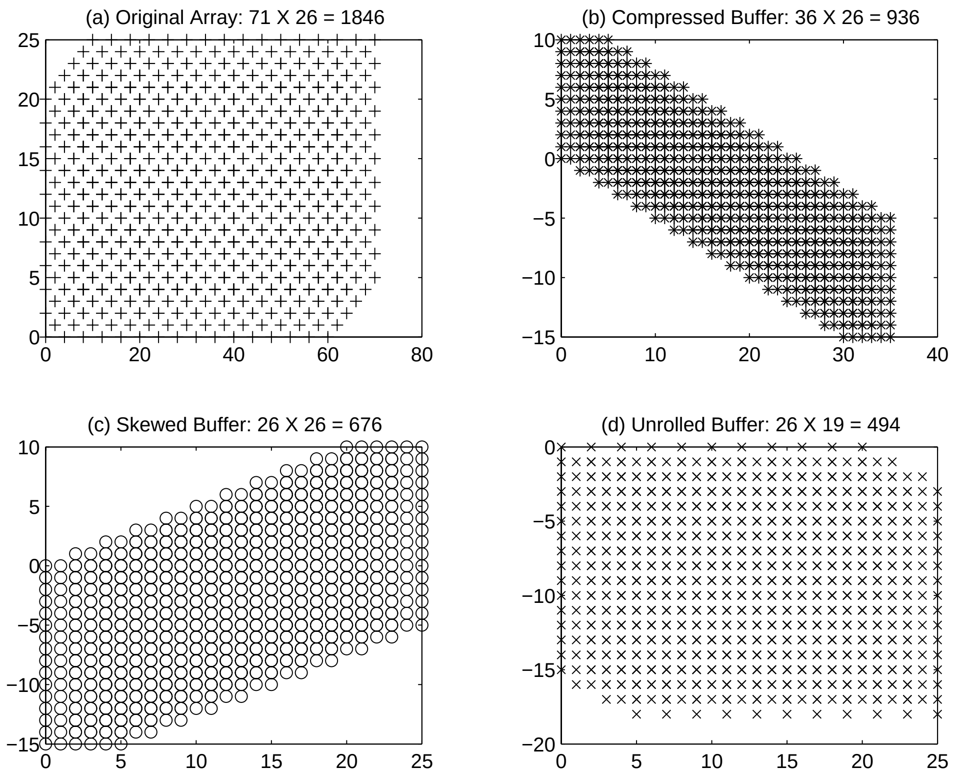

Figure 1: Four Reindexing Cases

(a) The box of the original reference is $71 \times 26 = 1846$. (b) The box after Hermite reindexing is $36 \times 36 = 936$. (c) The box after unimodular reindexing is $26 \times 26 = 676$. (d) The box after applying both Hermite and unimodular reindexing on an unrolled loop is $26 \times 19 = 494$.

and

$$B[b] = C[G^{-1}b] = A[HG^{-1}b + f]$$

and

$$A[Fz + f] = A[HVz + f] = C[Vz] = B[GVz].$$

The idea is to choose G such that the image of $\mathcal{I}$ under GV is a subset of the smallest rectangular region possible, so that $\mathcal{T}(B)$ can be contained in the smallest array possible, *i.e.*we want G to minimize $|\text{box}(GV\mathcal{I})|$. Hence, the objective function that we seek to minimize is

$$\prod_{k=1}^{r} \|(GV\mathcal{D}(\hat{z}))_k\|_1 \, .$$

We now give another interpretation of the problem in terms familiar in the theory of lattices and combinatorial optimization. The rows of $V\mathcal{D}(\hat{z})$ are linearly independent and generate an r-dimensional lattice $L \equiv \mathcal{L}((V\mathcal{D}(\hat{z}))^t)$ in $\mathcal{Z}^n$. The rows of $GV\mathcal{D}(\hat{z})$ as G varies over the unimodular matrices comprise all of the possible lattice bases for L. We seek a lattice basis for which the product of the lengths of the basis vectors, measured by the 1-norm, is minimized. Thus, our problem is the 1-norm version of the lattice basis reduction problem whose 2-norm version has been studied extensively [5, 6, 4]. We study this optimization problem carefully in Section 5.

4.6 Generalizations

We can extend our techniques to references that are not UGA. For example, if some indices of an array are nonaffine, then we can consider only the affine indices and view the array as a lower-rank affine array of nonaffine arrays. Alternatively, if the references occur only on the right-hand side and if some references have more affine indices than others, then we can create an optimized local memory surrogate for the set of maximally affine references and leave untouched any insufficiently affine references.

When there are affine references that are not UGA, we can use several alternative techniques, each of which might be best in a given situation.

1. If one affine reference generates a sublattice of another, then you may use the reindexing of the other reference. Example: for references $A[2z_i]$ and $A[6z_i]$, you can apply our reindexing technique to $A[2z_i]$ and then derive the reindexing for $A[6z_i]$. The downside is that every other location between $A[2\hat{z}_i]$ and $A[6\hat{z}_i]$ is stored when only every sixth location is actually required. A more practical example is when two references generate the same lattice but are not UGA, such as $A[i, j]$ and $A[j, i]$.

   The way you do this in general is to take two affine references with different linear parts but the same shape ($m \times n$), say $A[F_1 + f_1]$ and $A[F_2 + f_2]$. The second reference generates a sublattice of the shifted lattice of the first reference if $\forall x_2 \exists x_1 \mid F_1 x_1 + f_1 = F_2 x_2 + f_2$. This is true if and only if $f_2 - f_1 \in \mathcal{L}(F_1)$ and the equation $F_2 = F_1 E$ has an $n \times n$ integer solution matrix E: for then $F_1 x_1 = F_1 E x_2 + (f_2 - f_1)$ has the solution $x_1 = E x_2 - e$ where $F_1 e = f_2 - f_1$.

If these conditions are satisfied, then you make a reindexing using F_1 and allocate the local-memory array. If $B[Gi]$ holds $A[F_1 i + f_1]$ then $B[GEi + Ge]$ holds $A[F_1(Ei + e) + f_1] = A[F_2 i + f_2]$.

2. When several affine references are not UGA and the sublattice requirement of point 1 does not apply, we can use the lattice generated by their greatest common divisor. Example: for references $A[4i]$ and $A[6i]$, we find $2 = \gcd(4, 6)$, make a reindexing with this linear form, and proceed as in the previous case.

   Extending this to multidimensions, we are given $F_1, F_2 \in \mathcal{Z}^{m \times n}$, we form the matrix $[F_1, F_2] \in \mathcal{Z}^{m \times 2n}$, and we then find its generalized Hermite factorization HU. Assuming the rank is r, we take the first r rows of U, forming an $r \times 2n$ matrix. This is split into two $r \times n$ matrices $[V_1, V_2]$. References to $A[F_1 i]$ are then replaced with references to $B[V_1 i]$, and similarly for F_2, V_2. The size of the local-memory array is determined by bounding the set of elements of the form Hi as i varies over the iteration space.

3. Finally, in a case like $A[2i]$ and $A[3i]$, the gcd technique yields $H = I$, but there is still a potential for savings by using an lcm (least common multiple) technique. The key idea is that while the sparsest lattice that includes the lattices $\mathcal{L}(2)$ and $\mathcal{L}(3)$ is all of the integers, if we instead consider the lattice $\mathcal{L}(6)$ and its translations, then four out of the six of these translations cover $\mathcal{L}(2) \cup \mathcal{L}(3)$, which contains the set of elements we need.

   Suppose A is of size N. Unroll the loop by a factor of $\text{lcm}(2, 3) = 6$. Allocate local-memory arrays LM_0, LM_2, LM_3, and LM_4. Each has size $N/6$. $LM_k[i]$ holds $A[6i + k]$ for $k \in \{0, 2, 3, 4\}$. Thus, we have used $(2/3)N$ space in the local memory to hold the part of A that we need.

   The loop before this transformation is:

```
for (i = 0; i < N/3; i++) {
     S1:  ... A[2i]
     S2:  ... A[3i]
  }
```

After it is changed to:

```
for (i = 0; i < N/18; i++) {
     S1_0:  ... LM0[2i]
     S2_0:  ... LM0[3i]
     S1_1:  ... LM2[2i]
     S2_1:  ... LM3[3i]
     S1_2:  ... LM4[2i]
     S2_2:  ... LM0[3i+1]
     S1_3:  ... LM0[2i+1]
     S2_3:  ... LM3[3i+1]
     S1_4:  ... LM2[2i+1]
     S2_4:  ... LM0[3i+2]
     S1_5:  ... LM4[2i+1]
     S2_5:  ... LM3[3i+2]
  }
```

5 Practical approaches to the best unimodular reindexing

The lattice basis reduction algorithm of Lenstra, Lenstra, and Lovász (LLL) [4], which runs in (low degree) polynomial time, provides a basis of bounded nonoptimality w.r.t. the 2-norm. Using the equivalence of all norms on a finite dimensional space, and in particular the bounds

$$\|x\|_2 \leq \|x\|_1 \leq \sqrt{n}\|x\|_2$$

we can show that the LLL basis has bounded nonoptimality w.r.t. the 1-norm. This said, we have also looked into other algorithms, which seem to be more 1-norm friendly.

5.1 Lattice basis reduction in the 1-norm

We build G as a product of elementary unimodular transformations of the form $E(i, v)$ which is the identity except for the i-th row, in which the off-diagonal entries are the elements of the vector v.

We have found that a hill-climbing heuristic, which uses elementary unimodular transformations $E(i, v)$, is of practical value; in practice it yields better results than the LLL algorithm. It is less prone to getting stuck at a local minimum than algorithms that work with only two rows at a time. The method is organized as a sequence of sweeps; a sweep visits each of the rows in turn. When visiting a row, we consider subtracting an integer linear combination of the other rows so as to reduce the one-norm, if possible. We examine a test set of linear combinations and pick the best one.

How do we generate the test set? If x is the given row of the matrix and X is the matrix consisting of the other rows, then we are seeking the integer row vector c that solves

$$\min_{c} \|x - cX\|_1 \,. \tag{3}$$

This linear program can be solved exactly over the real vectors c in polynomial time by linear programming, or solved over the integers in exponential time. Since in our applications the dimensionality is quite small (at most the dimensionality of the program's arrays) we don't feel that *exponential* is necessarily a dirty word.

Clearly, we can find an integer solution with an ILP solver, and use this locally best integer linear combination of the other rows. Or we may get a test set faster by using an LP solver to get a real solution $c_{\text{real}} \in R^n$ to (3), generating two bounding integer vectors $c_{lo} = \lfloor c_{\text{real}} \rfloor$ and $c_{hi} = \lceil c_{\text{real}} \rceil$, and generating the 2^n integer vectors whose entries come from either c_{lo} or c_{hi}. These 2^n vectors are the test set.

The best unimodular reindexing for Example 2 (Section 4.4) is

$$G = \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix} .$$

and was found by both algorithms above. This leads to a local-memory array B having 676 elements (Figure 1c). While $\mathcal{T}(B)$ is still skewed with respect to the axes of B, it is much less so than $\mathcal{T}(C)$, and a substantial storage reduction follows.

If we iterate the ILP-based procedure, then it converges to a matrix for which no row can have its 1-norm reduced by subtracting a linear combination of other rows. If we are at such a situation, then we have locally minimized the objective function. Thus we have the algorithm in Figure 2.

```
while (changes happen)
  for each row
    minimize the row's one norm by
    subtracting an integer linear combination
    of the other rows, using ILP to find
    the minimizing integer coefficients.
  rof
elihw
```

Figure 2: Algorithm to Find Locally Optimal Basis
The cost of this algorithm is an exponential ILP solve, a polynomial number of times.

5.2 G does not have to be unimodular

Thus far we have required that G be unimodular. In some cases we can obtain better results by allowing G to be a matrix of determinant one over the rationals.

Revisiting Example 2 (Section 4.4), let $\hat{z} = [5; 10; 15]$. The matrix V that we obtained by the Hermite reindexing algorithm is

$$V = \begin{pmatrix} 1 & 0 & 2 \\ 0 & 1 & -1 \end{pmatrix}$$

The extents of box$(V\mathcal{I})$ are [36; 26]. If we take the best unimodular reindexing, we get

$$G = \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}$$

so that the new point set is the image under

$$GV = \begin{pmatrix} 1 & 2 & 0 \\ 0 & 1 & -1 \end{pmatrix}$$

which has extents [26; 26], but $\mathcal{T}(GV)$ is still not rectangular.

If we solve the linear program for the best real transform with determinant one, we obtain

$$M = \begin{pmatrix} 1 & 0 \\ -.5 & 1 \end{pmatrix};$$

this takes us to

$$MGV = \begin{pmatrix} 1 & -2 & 0 \\ -0.5 & 0 & -1 \end{pmatrix}$$

which maps $\mathcal{I}$ to a rectangle with extents [26, 19], having space for 494 elements (Figure 1d). This is very close to $|\mathcal{T}(A)|$, which in this case is 456. We can implement this non-integral transform by unrolling the first loop by a factor of two. The solution then becomes an affine map onto integers.

5.3 How good are locally optimal bases?

A natural question is whether the locally optimal unimodular reindexing generated in Section 5.1 is always best possible. For the case $r = 2$, there is a simple geometric proof that a local optimum is a global optimum. Suppose X has two rows, a and b. If X is not globally optimum, then there is some linear combination of its rows that is smaller than one of the rows that participates in this linear combination. If $ma + nb$ is smaller than the larger of a and b, then so is $a + b$, because $a + b$ is a barycentric combination of $\{a, b, ma + nb\}$. Thus, X was not locally optimum. Sadly, this proof (which is due to Alain Darte) does not generalize to more than two vectors.

For $r \geq 6$ we have a negative result: a locally optimal basis is not necessarily globally optimal; in fact, there is in general no constant factor, even one that grows with n, bounding the extent to which a locally optimal basis can have cost that exceeds the cost of the best basis for the same lattice. We show this here for the 1-norm with a counterexample in 6 dimensions. In Section C, we generalize the construction so that it works for any p-norm with integer p, and the counterexamples in general are in $5^p + 1$ dimensions.

Consider the following change of bases $B' = U \times B$:

$$\begin{pmatrix} 5 & 0 & 0 & 0 & 0 & 1 \\ 0 & 5 & 0 & 0 & 0 & 1 \\ 0 & 0 & 5 & 0 & 0 & 1 \\ 0 & 0 & 0 & 5 & 0 & 1 \\ -1 & -1 & -1 & -1 & -1 & 2 \\ 0 & 0 & 0 & 0 & 0 & 5 \end{pmatrix} =$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & -3 \\ 2 & 2 & 2 & 2 & 2 & -5 \end{pmatrix} \times \begin{pmatrix} 5 & 0 & 0 & 0 & 0 & 1 \\ 0 & 5 & 0 & 0 & 0 & 1 \\ 0 & 0 & 5 & 0 & 0 & 1 \\ 0 & 0 & 0 & 5 & 0 & 1 \\ 0 & 0 & 0 & 0 & 5 & 1 \\ 2 & 2 & 2 & 2 & 2 & 1 \end{pmatrix}.$$

B is locally optimal w.r.t. the 1-norm (see Section C for a proof), $\prod_{k=1}^{6} \|B_k\|_1 = 85536$, and U is unimodular. However, $\prod_{k=1}^{6} \|B'_k\|_1 = 45360$.

For three, four, and five dimensions, we don't know which of the following possibilities is true: locally optimal implies globally optimal, locally optimal can be worse than globally optimal by an arbitrarily large factor, or something in between. Nevertheless, we have not found any practical example of nonoptimality.

6 Conclusion

We have shown how to generate a nearly optimal reindexing of array elements to be used in generating code that explicitly loads the relevant sections of global-memory resident arrays into local memories.

This paper has not addressed other significant issues in managing local memory. Its context has been a single loop nest. It has not considered the issue of sharing a limited size local memory among contending uses when there is not enough space for all of uses to coexist at the same time. It has not addressed the problem of when to move data between local and global memory. It has not addressed the problem of moving data from one part of local memory to another when overlapping sections of an array are referenced on repeated executions of a loop nest.

7 Acknowledgments

We are very grateful to Alain Darte and Hendrik Lenstra for their contributions to this research. Hendrik gave us the blueprint for the proof of nonoptimality of our heuristic for lattice basis reduction. Alain has been a steady source of ideas and mathematical knowledge.

References

[1] S. Anantharaman and S. Pande. Compiler optimization for real time execution of loops on limited memory embedded systems. In *Proceddings of the 19th IEEE Real-Time Systems Symposium*, pages 154–164, 1998.

[2] C. Eisenbeis, W. Jalby, D. Windheiser, and c. B. Fran˙ A strategy for array management in local memory. Technical Report 1262, INRIA, Domaine de Voluceau, Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France, July 1990.

[3] K. Gallivan, W. Jalby, and D. Gannon. On the problem of optimizing data transfers for complex memory systems. In *Proceedings of the 1988 ACM International Conference on Supercomputing*, pages 238–253, 1988.

[4] A. Lenstra, H. L. Jr., and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261:515–534, 1982.

[5] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. Wiley, New York, 1988.

[6] A. Schrijver. *Linear and Integer Programming*. Wiley, New York, 1986.

[7] M. V. Wilkes. The memory gap and the future of high performance memories. *Computer Architecture News*, 29(1):2–7, Mar. 2001.

[8] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. Technical Report CS-94-48, Computer Science Department, University of Virginia, 1, 1994.

A Left inverses and integer solutions for integer matrices

We show here how to construct left inverses and how to use one to test whether a given vector is in a given lattice. In this section, m and n denote arbitrary integers.

Let $X \in \mathcal{Z}^{m,n}$ have full column rank (like H in Section 4.2). Then it has left inverses X^{-1} that satisfy $X^{-1}X = I_n$, where I_n is the identity matrix of order n. If $m > n$ then it will have infinitely many of them. Any of them will do for our purposes, because if $u \in \mathcal{L}(X)$ then for some vector $v \in \mathcal{Z}^n$, $u = Xv$ and $v = (X^{-1}X)v = X^{-1}u$ for any left inverse X^{-1}. Note that a left inverse is in general a rational matrix. (There are integer left inverses only if X has a unimodular extension.)

To test whether a given vector u is in $\mathcal{L}(X)$ we rely on the following

Lemma A.1 Let X be integer with full column rank and let X^{-1} be any left inverse of X. The vector u is in $\mathcal{L}(X)$ if and only if

(i) $X^{-1}u$ is integer, and

(ii) $XX^{-1}u = u$.

Proof: For the $\Longrightarrow$ part, let $u = Xv$ with v integer. Then $X^{-1}u = X^{-1}Xv = v$ is integer so that (i) holds. Moreover, $XX^{-1}u = XX^{-1}Xv = Xv = u$ so that (ii) holds as well. For the $\Longleftarrow$ part, (ii) yields $u = Xv$ where by (i) $v = X^{-1}u$ is integer, so that $u \in \mathcal{L}(X)$. ▮

To see that (i) alone is insufficient, consider $X = [1; 2]$ with left inverse $X^{-1} = [1, 0]$ and $u = [1; 1]$. To see that (ii) alone is also insufficient, let $X = [2; 2]$ with $X^{-1} = [.5, 0]$ and $u = [1; 1]$.

Given software for computing the Hermite normal form, finding a suitable left inverse is straightforward. The method we propose here yields an integer inverse if one exists. To begin, we compute a left-sided Hermite factorization $X = U\tilde{H}$ where U is unimodular and $\tilde{H}$ has rank n and is zero below row n. (To do so one may present the transpose of X to software for the conventional right-sided Hermite factorization.) Denote by H the nonsingular matrix of order n consisting of the first n rows of $\tilde{H}$. Since $\tilde{H} = U^{-1}X$, we have that $H = WX$ where W is the $n \times m$ integer matrix consisting of the first n rows of U^{-1}. Then $H^{-1}WX = H^{-1}H = I_n$ and the left inverse we seek is $H^{-1}W$. When X has a unimodular extension $[X, Y]$ then $U = [X, Y]$ and $H = I_n$, whence our left inverse is integer.

A.1 Code for Finding an Integer Left Inverse

We now provide Matlab code for finding an integer left inverse.

```
function Ainv = intleftinv(A)
% Left inverse of the integer matrix A; returns an
% integer result if and only if A has a unimodular
% extension. A must have full column rank
```

```
%
% METHOD
% Factor:  A = U(:, 1:n) * H
%   where U is unimodular and H is square
% Form:    Ainv = inv(H) * W
%   where W consists of the first n rows of inv(U)
%
[m, n] = size(A);   assert(rank(A) == n);
[H, U] = economy_left_hermite(A);
Uinv = inv(U);
W = Uinv(1:n,:);
Ainv = inv(H) * W;
```

B Code for Test Vector Generation

These Matlab functions implement the algorithm described in Section 5.1.

```
[r, n] = size(f);
g = eye(r);
changes = 1;

while (changes)
 changes = 0;
 for row = 1:r
  x = f(row,:);
  others = f(find( 1:r ˜= row ), :);
  coeffs = (least_onenorm(others’, x’))’;
  testvectors = (tensor([floor(coeffs);
  ceil(coeffs)]))’;
  nt = size(testvectors, 1);
  testrows = x(ones(nt,1),:) - testvectors * others;
  mintest = min(sum(abs(testrows), 2));
  if (sum(abs(x)) > min(sum(abs(testrows), 2)))
   ff = find(sum(abs(testrows), 2) == mintest);
   ff = ff(1);
   f(row, :) = testrows(ff,:);
   g = g * elemrow(r, row, testvectors(ff,:));
   changes = 1;
  end
 end  % loop over row
end
}

function e = elemrow(n, r1, mpy)
e = eye(n);
```

```
offdiag = find(1:n ~= r1);
e(r1, offdiag) = mpy;
```

C A locally optimal basis in 6 or more dimensions may be arbitrarily bad

The aim of this section is to prove that a locally optimal basis is not necessarily globally optimal. The structure of the argument is as follows. We show that if a basis is not too bad (looking at the ratio of its cost to the optimum basis cost) then its shortest vector cannot be too much bigger than the shortest vector in the lattice. This is true in any finite number of dimensions and with any norm. We then construct a 6-dimensional basis, locally optimal in the 1-norm, for which the basis vectors are all $O(1)$ in size, but for which the shortest vector in the lattice is arbitrarily small. Together with the lemma just described, this does the trick. We generalize the construction so that it works for any p-norm with integer p. The counterexamples in general are in $5^p + 1$ dimensions.

C.1 Definitions

In the following, let $\| \ \|$ be a norm over $\mathcal{R}^n$, and let $\text{cost}(X) \equiv \prod_{k=1}^{r} \|X_k\|$.

Definition C.1 A basis B is *locally optimal*, or LOPT(B), if and only if none of its rows can be made smaller with respect to some norm by adding a linear combination of the other rows. That is, B is locally optimal if and only if $\|x^{\,t}B\| \geq \|B_k\|$ for all integer vectors x whose k-th element is 1.

Definition C.2 The *length of a shortest element of a basis* B is

$$\text{shortest}(B) \equiv \min_i \|B_i\| \, .$$

Definition C.3 The *length of a shortest element of a lattice* L is

$$\text{shortest}(L) \equiv \min_{x \in L : x \neq \mathbf{0}} \|x\| \, .$$

Definition C.4 The *determinant* of a lattice L is

$$\det(L) \equiv \sqrt{\det(BB^t)} \, ,$$

where B is any basis that generates L.

Lemma C.1 The determinant of a lattice is well defined, *i.e.* it does not depend on the basis chosen. The determinant is equal to the volume of the r-dimensional parallelepiped subtended by the r rows of B.

C.2 Any good basis contains a short basis vector

Lemma C.2 (Hadamard) For any basis B, and all $p \leq 2$,

$$\det(\mathcal{L}(B)) \leq \text{cost}_p(B). \tag{4}$$

Proof: See Schrijver, Section 6.2. ∎

Corollary C.1 Let a norm be given for $\mathcal{R}^n$ for all n. There exist constants $H(n)$ such that, for any lattice basis B,

$$\det(\mathcal{L}(B)) \leq H(n)\text{cost}(B). \tag{5}$$

Proof: From the Hadamard inequality and the equivalence of all norms in $\mathcal{R}^n$. ▮

Lemma C.3 (Hermite; Minkowski) There exist constants $M(n) = O((2n/\pi e)^{n/2})$ depending only on n such that for any lattice $L \subset \mathcal{Z}^n$ there is a basis B satisfying

$$\text{cost}(B) \leq M(n)\det(L)\ . \tag{6}$$

Proof: See Schrijver, Section 6.2. ▮

Corollary C.2

$$\text{cost}(\mathcal{L}(B)) \leq M(n)\det(L)\ . \tag{7}$$

This next lemma exposes a critical property of bases – a basis that is nearly globally optimal is also near optimal as measured by shortest element.

Lemma C.4 If B is such that

$$\text{cost}(B) \leq K\ \text{cost}(\mathcal{L}(B)) \tag{8}$$

for some $K > 0$, then

$$\text{shortest}(B) \leq M(n)H(n)K\ \text{shortest}(\mathcal{L}(B))\ . \tag{9}$$

where the constants $M(n)$ and $H(n)$ are those appearing in the generalized Hadamard and Minkowski inequalities.

Proof :

From (7) and (8) we conclude that

$$\text{cost}(B) \leq M(n)K\det(\mathcal{L}(B))\ . \tag{10}$$

Let $L = \mathcal{L}(B)$, and let x be a shortest vector in L. Since B is a basis for L, there exists $w \in \mathcal{Z}^r$ such that $x = \sum_{i=1}^{r} w_i B_i$. Let w_h be a non-zero element of w. Define a new basis E in which $E_i = B_i$ for $i \neq h$ and $E_h = w_h B_h$. Clearly, $\det(\mathcal{L}(E)) = |w_h|\det(\mathcal{L}(B))$. Define a new basis F in which $F_i = B_i$ for $i \neq h$ and $F_h = x$. Then, $\mathcal{L}(F) = \mathcal{L}(E)$ since $E_h = F_h - \sum_{i \neq h} w_i F_i$. Hence, $\det(\mathcal{L}(F)) = \det(\mathcal{L}(E)) = |w_h|\det(\mathcal{L}(B))$. From this result and (10) it follows that

$$\text{cost}(B) \leq M(n)K\det(\mathcal{L}(F)), \tag{11}$$

and from (5) it follows that

$$\text{cost}(B) \leq M(n)H(n)K\text{cost}(F). \tag{12}$$

Since F differs from B only in row h, we may cancel common factors to obtain

$$\|b_h\| \leq M(n)H(n)K\|x\|, \tag{13}$$

and since $\text{shortest}(B) \leq \|b_h\|$ it follows that

$$\text{shortest}(B) \leq M(n)H(n)K\ \text{shortest}(\mathcal{L}(B)). \tag{14}$$

▮

C.3 There are p-norm locally optimal bases for $\mathcal{Z}^{5p+1}$ with no short basis vector

The following definition describes a family of bases parameterized by a positive integer p. The subsequent lemma states that each family is locally optimal in the p-norm. These families of bases will be used to prove that a locally optimal basis is not necessarily globally optimal.

Definition C.5 $\mathcal{F}(p) = \{B \mid B \in \mathcal{R}^{d\times d} : d = 5^p + 1, \epsilon \in \mathcal{R} : 0 < \epsilon \leq .2, i, j \in \mathcal{N} : 1 \leq i \leq d-1, 1 \leq j \leq d-1,\ B_{ij} = I_{ij}, B_{dj} = .4, B_{id} = \epsilon, B_{dd} = \epsilon\}$.

For example, $\mathcal{F}(1)$ consists of the bases

$$B = \begin{pmatrix} 1 & & & & & \epsilon \\ & 1 & & & & \epsilon \\ & & 1 & & & \epsilon \\ & & & 1 & & \epsilon \\ & & & & 1 & \epsilon \\ .4 & .4 & .4 & .4 & .4 & \epsilon \end{pmatrix}$$

for all $\epsilon \leq .2$.

Lemma C.5 Every basis in $\mathcal{F}(p)$ is locally optimal with respect to the p-norm.

Proof (by contradiction): Assume that for some p there is a basis $B \in \mathcal{F}(p)$ which is not locally optimal. Then, for some k and some integer vector x with x_k=1,

$$\|x^{\,t}B\|_p < \|B_k\|_p. \tag{15}$$

Case 1: $k = d$: Substituting $x_d = 1$ into (15) yields

$$\left(\sum_{i=1}^{d-1} |x_i + .4|^p + \left|\epsilon \sum_{i=1}^{d} x_i\right|^p\right)^{1/p} < (2^p + \epsilon^p)^{1/p}.$$

In addition to x_d, at least one x_i must be non-zero or else the basis doesn't change. The term $|x_i+.4|^p$ has a lower bound of $.6^p$ for a non-zero x_i and $.4^p$ otherwise. Since $0 < \epsilon \leq .2$, it follows that

$$.6^p + .4^p(5^p - 1) < 2^p + .2^p,$$

which reduces to

$$3^p - 2^p < 1,$$

which is clearly false since $p > 0$.

Case 2: $1 \leq k < d$: Since $k \neq d$, (15) becomes

$$\left(\sum_{i=1}^{d-1} |x_i + .4x_d|^p + \left|\epsilon \sum_{i=1}^{d} x_i\right|^p\right)^{1/p} < (1 + \epsilon^p)^{1/p} \tag{16}$$

First consider $x_d = 0$. Then $2 \leq \sum_{i=1}^{d-1} |x_i + .4x_d|^p$ since at least two of the x_i's must be non-zero. Since $0 < \epsilon \leq .2$, (16) reduces to $2 < 1 + .2^p$, a contradiction.

Next consider $x_d = 5n$ where n is a non-zero integer. The term $|x_k + .4x_d|$ reduces to $|1 + 2n|$, and (16) is clearly false for all $n \neq -1$. Let $S = \{1 \ldots d-1\} - \{k\}$. If $n = -1$, (16) is false if

$\exists i \in S\ x_i \neq 2$. Finally, if $\forall i \in S\ x_i = 2$, then $(\sum_{i=1}^{d} x_i) = 1 + 2(5^p - 1) - 5 \neq 0$, and hence $\epsilon^p \leq |\epsilon \sum_{i=1}^{d} x_i|^p$. From (16), it follows that $1 + \epsilon^p < 1 + \epsilon^p$, a contradiction.

Finally, consider $x_d = 5n + q$ where $n \in \mathcal{Z}$ and $q \in \{1, 2, 3, 4\}$. If $q = 1$ or $q = 4$, then $.4 \leq |x_i + .4x_d|$ and hence (16) reduces to $2 < 1 + .2^p$ which is false. If $q = 2$ or $q = 3$ then the term $|x_i + .4x_d|$ is bounded below by $.4$ when $x_i \neq -2n - 1$ and $.2$ otherwise. Hence, if $x_i \neq -2n - 1$ for at least one $i \neq d$, then (16) becomes

$$.2^p(5^p - 1) + .4^p < 1 + .2^p,$$

which reduces to

$$2^p < 2,$$

which is clearly false for $p > 0$. On the other hand, if all $x_i = -2n - 1$ where $i \neq d$, then $n = -1$ (since $x_k = 1$) and $(\sum_{i=1}^{d} x_i) = 5^p - 5 + q \neq 0$, and once again we get $\epsilon^p \leq |\epsilon \sum_{i=1}^{d} x_i|^p$ which leads to the contradiction $1 + \epsilon^p < 1 + \epsilon^p$. ∎

We can now conclude the main result of this section, which is unfortunately, but not surprisingly, negative. The optimization algorithms of the previous section can all get stuck at poor local optima. (The good news is that we only know of contrived examples in 6 or more dimensions for which this happens.)

Theorem C.1 For any positive integer p, for any $K > 0$, there exists a p-norm locally optimal basis B for $\mathcal{Z}^{5p+1}$ such that

$$\text{cost}_p(B) > K\ \text{cost}_p(\mathcal{L}(B))\,.$$

Proof : The family $\mathcal{F}(p)$ contains such a basis. By inspection, the shortest basis vector satisfies

$$\text{shortest}_p(B) > 1\,.$$

Let n be the dimensionality of $\mathcal{F}$, $n = 5^p + 1$. The shortest vector in the lattice is not longer than the lattice vector equal to twice the sum of the first $n - 1$ rows minus five times the last row, which is $(0, \ldots, 0, (25^p - 5)\epsilon)$, so that

$$\text{shortest}_p(\mathcal{L}(B)) \leq (25^p - 5)\epsilon\,.$$

Thus, by taking ϵ small enough, the ratio

$$\text{shortest}_p(B)/\text{shortest}_p(\mathcal{L}(B))$$

can be made arbitrary large, in fact larger than $M(n)H(n)K$. Taking ϵ to be rational and then multiplying by the denominator, we obtain an integer basis with the same properties. The result then follows by Lemma C.4. ∎