Strip Mining on SIMD Architectures

Michael Weiss
Compass, Inc.
550 Edgewater Dr.
Wakefield, MA

Abstract

A compiler for vector processors must strip mine statements to fit the vector register length. The same compiler technology can be applied to SIMD machines, removing the need for a virtual processor mechanism and offering significant advantages over that approach.

The different memory organizations of SIMD and vector machines give rise to strip mining differences. SIMD interprocessor communication must be stripped. Array allocation gives SIMD strip mining a global aspect. SIMD loads and stores may be uniform or ragged. These all contribute to the complexity of the SIMD strip mining problem, and provide opportunities for a compiler to make use of its wider view of the code.

1 Introduction

Strip mining may be defined as *folding array-based parallelism to fit the available hardware parallelism.* The problem of strip mining appears in different guises on vector, SIMD, and MIMD machines. The vector case is probably the best known [2]. In the MIMD case, strip mining is usually called *loop blocking* [8]; on a SIMD architecture, the term *virtualization* is commonly used [9]. This term derives from the approach taken in Thinking Machine Corporation's Connection Machine, which supports (at the instruction level) multiple virtual processors for each physical processor. For vector machines, the compiler usually performs strip mining.

A simple example illustrates the relationship between vector strip mining and the virtual processor approach. Consider the statement:

```
A(1:102400) = B(1:102400)
```

Suppose this is to be run on a machine with 1K processors. The data parallel programming paradigm dictates one virtual processor per array element. We let virtual processor v contain $A(v+1)$ and $B(v+1)$. Suppose there are 100 virtual processors per physical processor. Suppose virtual processors 0 to 1023 are mapped to physical processors 0 to 1023, virtual processors 1024 to 2047 are mapped to physical processors 0 to 1023, and in general virtual processor $1024p+q$ is mapped to physical processor q, where p ranges from 0 to 99 and q ranges from 0 to 1023. (The Connection Machine actually maps consecutive ranges of virtual processors to each physical processor. We will consider this mapping later.)

The memory of each physical processor is partitioned among its virtual processors. Under our mapping, memory partition p of physical processor q belongs to virtual processor $1024p+q$. The machine performs an operation by looping through the partitions so that on the p-th iteration, 1024 physical processors collectively perform the operation on the data items in the p-th partition— here, $A((1024p+1):1024(p+1))$ and $B((1024p+1):1024(p+1))$.

Contrast this with the vector strip mining approach. The parallelism bandwidth is now the vector register length, say 1024. The statement is strip mined into the following form:

```
DO p=0,99
  A((1024p+1):1024(p+1))
          = B((1024p+1):1024(p+1))
ENDDO
```

The compiler generates the loops instead of hardware (or firmware), but otherwise the effect is the same.

Strip mining uses dependence analysis to check that the stripped version of a vector assignment has the same semantics as the original. If it does not, then a temporary may be required. Global knowledge of this sort is unavailable at run-time, so a machine architecture providing virtual processors is forced to introduce temporaries more frequently.

This paper describes the application of the vector strip mining approach to SIMD machines. The concept of virtual processors remains useful, but we assume the hardware does not provide them. Instead, the compiler manages the mapping from virtual to physical processors.

The compiler's global knowledge permits a variety of optimizations. The classic vector strip mining optimizations— loop reversal, loop interchange, loop fusion— offer as much benefit on a SIMD as a vector machine. Loop reversal and loop interchange can eliminate the need for a temporary or reduce its size. Loop fusion reduces loop overhead, and preserves common subexpressions that would be split into different strip mining loops, allowing more extensive use of registers (assuming the hardware provides them).

Certain SIMD-specific problems must be solved. Interprocessor communication must be stripped. Uniform versus non-uniform memory depth becomes an issue.

Arrays must be allocated across the combined memory of the all the processors— this is the foremost difference between SIMD and vector strip mining. In the vector case, arrays live in main memory between strip mining loops. In the SIMD case, strip mining becomes tied up with data allocation, and the global aspects of the problem become even more decisive.

In the SIMD case, it sometimes pays to strip mine multi-dimensionally. For example, the statement:

```
A(1:M,1:N) = B(1:M,1:N)
```

could be transformed into:

```
DO p1 = start1, end1
  DO p2 = start2, end2
    (p1,p2)-strip of A =
         (p1,p2)-strip of B
  ENDDO
ENDDO
```

where the $(p1, p2)$-strip of A would take the form $A((p1 \cdot L_1 + 1) : (p1 + 1) \cdot L_1, (p2 \cdot L_2 + 1) : (p2 + 1) \cdot L_2)$, and likewise for B. This can prove beneficial even if M exceeds the total number of processors. In contrast, multi-dimensional strip mining pays off in the vector case only when the separate array dimensions are too short to fill out the vector registers. Multi-dimensional stripping incurs the penalty of non-unit stride, not a concern on the SIMD machine.

The ideas in this paper are an outgrowth of the experience Compass has gained implementing a number of compilers, among them the Fortran compiler for the Connection Machine [1] (a non-strip mining compiler for a SIMD machine), the Suprenum Fortran compiler (a strip mining compiler for a vector machine), and the MasPar MP-1 Fortran compiler (a strip mining compiler for a SIMD machine).

In the current Compass SIMD compiler architecture, strip mining is combined with *data optimization*, a powerful technique for reducing interprocessor communication [4, 5]. (See Li and Chen [6] for related ideas.)

The paper is structured as follows: Section 2 describes the SIMD architecture assumed in the rest of the paper. Section 3 outlines the method of data optimization. The output of data optimization is the input to the strip miner. Section 4 defines the class of mappings contemplated, and depicts the various kinds of strips that arise in the SIMD case. Section 5 explores some of the problems encountered in strip mining statements with more than one stride or strip length. Sections 6 and 7 deal with the stripping of loads, stores, and spreads. (Strip mining grid motion is discussed in the full version of this paper [11].) Section 8 discusses the allocation problem. Section 9 is a summary, and refers to related work.

2 Hardware Architecture

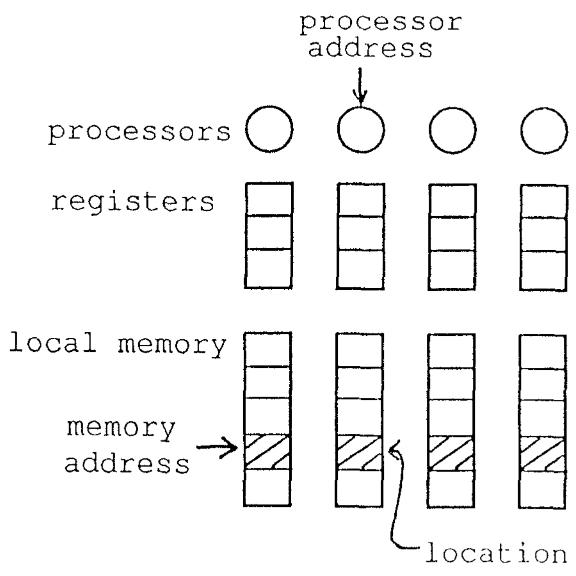
We describe a SIMD machine architecture for which compiler strip mining is appropriate.

Our SIMD machine consists of a set of identical processors. As usual with SIMD systems, a certain subset of the processors is enabled at any given time (the *context*); as implied by the term SIMD, all enabled processors execute the same instruction. Each processor has a unique *processor address.* For simplicity, we assume that the number of processors is 2^N, so the processor address can be regarded as a string of N bits.

Each processor has its own local storage, divided into *registers* and *memory.* Since the processors are identical, each processor's memory is addressed identically (as are the registers). While real SIMD systems have an attached scalar host or controller with its own memory, this is irrelevant to the topic of this paper, so the term *memory* for us always refers to the memory of the SIMD processors. A (memory) *location* is an ordered pair (processor address, memory address)— see figure 1. Thus, a memory address specifies where data is within one processor, while a location specifies where it is within the combined memory of the machine.

A *slice* is a set of locations, one per processor, all with the same memory address. A *cross-cut* is a set of locations, one per processor, but not necessarily at a uniform memory address.

The processors are assumed to have a conventional load/store architecture, where data must be loaded



The cross-hatched locations form a slice.

Figure 1: Processor address, Memory address, and Location

from memory into registers before operations can be performed. Results are ultimately stored back to memory, but it is desirable, as usual, to minimize traffic between registers and memory by avoiding the storage of intermediate results (temporaries).

There is a close analogy with a conventional vector machine. At an abstract level, a vector machine consists of main memory and vector registers and a processor; the processor can perform vector operations on the vector registers, and data can be transferred between main memory and the vector registers. Main memory on the vector machine corresponds to the combined memory of all the SIMD processors. A vector register corresponds to a "slice" of SIMD registers, i.e. a set of registers, one per processor, all with the same register address. To emphasise the analogy, we use the term *vector register* for such a set, and reserve the term *slice* for memory, as defined above. The SIMD processors collectively perform vector operations on the vector registers.

There are also significant differences. In the SIMD case, we distinguish between loading a slice (a *uniform* load) and loading a cross-cut (a *ragged* load). (Likewise, storing to a slice is a uniform store, and storing to a cross-cut is a ragged store.) One expects the cost of uniform operations to be less than ragged ones.[1] This distinction has no counterpart in the vector case. (Vector gather and scatter do not correspond to ragged loads and stores. We will examine the counterpart of gather and scatter shortly.)

More fundamentally, a processor can manipulate data only in its own local registers and memory. A SIMD machine has a need for some kind of communication network. In keeping with our general assumption of a load/store architecture, we assume that interprocessor communication is between the registers.

Certain patterns of communication usually enjoy special hardware support. On a mesh machine, translation a given distance along the grid is favored (we call this *grid communication*.) Arbitrary (non-regular) communication must also be provided in some fashion to support constructs such as vector-valued subscripts or parallel pointers. Regular communication patterns that are not translation along a grid also arise. (Example: sending from processor i to processor $2i$, which could come from the Fortran statement $A(2:100:2) = B(1:50)$.) Some machines support arbitrary interprocessor communication directly in the hardware (the Connection Machine, the MasPar MP-1), others through algorithms on top of grid hardware (the Princeton Engine, the Illiac IV, the ICL DAP).

So far as strip mining is concerned, what matters is the ability to perform certain operations involving interprocessor communication. Two such operations play a special role: REMOTE_STORE and REMOTE_LOAD. In REMOTE_STORE, each enabled processor has a value in a register, and a location it wishes to store the value to (perhaps in a different processor). We assume that there are no conflicts among the locations, i.e., no two enabled processors wish to store to the same location. Even so, two processors may want to store to two different locations that share the same processor address. In REMOTE_LOAD, each enabled processor has a location it wishes to fetch a value from.

Finally, we consider scatter and gather. A scatter operation at the source level looks like this in Fortran 90: $A(V(1:N)) = B(1:N)$. The stripped version of this scatters strips of B to A. Thus, the strip mining loop loads a strip of B and REMOTE_STOREs it to A. A gather (such as $A(1:N) = B(V(1:N))$) is stripped into REMOTE_LOADs.

3 Data optimization

In the Compass SIMD compiler architecture, data optimization poses the allocation problem that strip mining must solve. This section provides a condensed treatment.

Data optimization allocates array occurrences on a

[1]For example, on the MasPar MP-1 the difference seems to be a factor of 3 or more [9].

virtual machine with an unlimited number of ***virtual processors.*** Each virtual processor can be identified by its ***virtual processor coordinates***; data optimization determines a linear mapping from source-level subscripts to virtual processor coordinates. Data optimization analyzes usage patterns in the source, striving to find a virtual allocation that minimizes the cost of interprocessor communication. In principle, each occurrence of every array is allocated independently, though there is a preference to allocate a definition and a use of an array the same way.

Data optimization associates a *cell* with each subscript position. One should think of cells as holding virtual processor coordinates. A cell may be viewed as a placeholder for a set of bits in the processor address. We use the term *field* for such a set of bits.

We summarize the task presented by data optimization to the strip miner. Each array occurrence $A(\ldots)$ has cells assigned to its subscript positions. A cell can be used in more than one array occurrence. Each cell must be capable of holding a certain range of virtual processor coordinates, based on all the places where the cell is used.

Suppose the field allocated to a cell has k bits. The values $0 : 2^k - 1$ can be stored in the field. (Data optimization will choose the mapping from subscripts to virtual processor coordinates to make sure the latter are non-negative.) If this is inadequate for the range of values the cell must hold, then the cell must be stripped. Once the allocation has been chosen, the strip miner must strip the vector statements.

The *effective stride* along a given dimension is the amount one must increment the virtual processor coordinate to go from one array element to the next. The effective stride may differ from the source-level stride because of data optimization. For simplicity of exposition, all our examples will assume the virtual processor coordinates are identical to the source-level subscripts, so the effective stride will be the same as the source-level stride.

4 Strips and Slices

Suppose we have an array element with virtual processor coordinates $(v_1, \ldots, v_n)$. We use $\bar{v}$ for the processor address and $\hat{v}$ for the memory address, and v as shorthand for the n-tuple of virtual processor coordinates. We need to describe the mappings:

$$\begin{aligned} v &\mapsto \bar{v} \\ v &\mapsto \hat{v} \end{aligned}$$

We simplify the problem by demanding separate mappings for each virtual dimension:

$$\begin{aligned} v_k &\mapsto \overline{v_k} \\ v_k &\mapsto \widehat{v_k} \end{aligned}$$

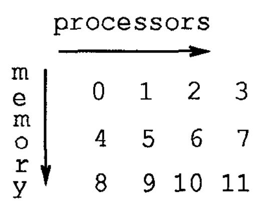
We call $\overline{v_k}$ the k-th ***physical processor coordinate***, and $\widehat{v_k}$ the k-th *memory depth.*

Before going into the details of these component mappings, we describe how they determine the mappings $v \mapsto (\bar{v}, \hat{v})$. The processor address is partitioned in fields corresponding to the cells that hold $v_1, \ldots, v_n$, so we specify $\bar{v}$ simply by saying that the k-th field holds the binary representation of $\overline{v_k}$. As for $\hat{v}$, we are mapping n coordinates $(\widehat{v_1}, \ldots, \widehat{v_n})$ into a linear memory space. We use the familiar row-major method:

$$\hat{v} = \text{Base} + \widehat{v_1} + \text{Extent}_1(\widehat{v_2} + \ldots)$$

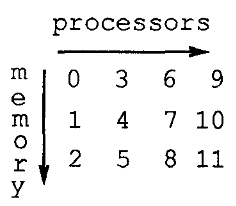
(Column-major would work just as well.)

We consider two ways of mapping a virtual processor coordinate to a physical processor coordinate and memory depth. First, the ***horizontal*** method (also called *cyclic* or *cut-and-stack*) may be pictured as follows:



Numbering starts at 0. For example, virtual processor coordinate 9 maps to physical coordinate 1 and memory depth 2. In general, suppose N_k is the size of the k-th field, and let $L_k = 2^{N_k}$. Coordinate v_k maps to $(v_k \bmod L_k, \lfloor v_k / L_k \rfloor)$.

The *vertical* method (also called *blocked*) interchanges the roles of memory depth and the processor address. Thus:



Here, virtual processor coordinate 9 maps to (3,0). To give formulas for the mapping, one must decide what range of virtual processor coordinate values are to be mapped. Suppose we wish to map $[0 : E_k]$. Let $L_k = 2^{N_k}$ as before, and let $M_k = \lceil (E_k + 1)/L_k \rceil$. Now v_k maps to $(\lfloor v_k / M_k \rfloor, v_k \bmod M_k)$.

We call L_k the *strip length* for both horizontal and vertical stripping, and call M_k the *strip depth* in the vertical case.

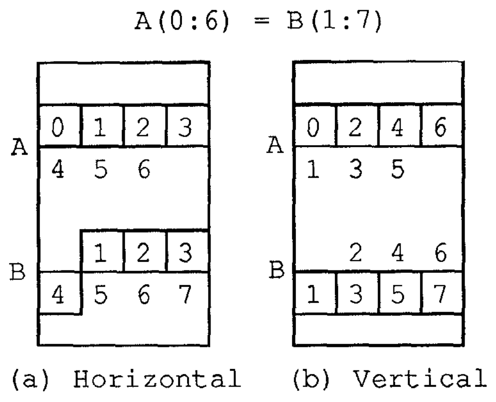


Figure 2: One-dimensional Strips

With horizontal stripping, the p-th slice of virtual coordinates is $[pL : (p+1)L-1]$. With vertical stripping, the p-th slice is $[p : E : M]$, i.e., all virtual coordinates $v \equiv p \pmod{M}$.

We turn to the stripping of statements. Let $A(\ldots)$ be an array section occurring in the statement; we have to partition the set of virtual processors containing $A(\ldots)$ into ***strips*** that fit in vector registers. Slices would certainly do; however, if $B(\ldots)$ is another section in the same statement, the strips of A must conform to the strips B, whereas the slices of A will not always conform to the slices of B.

We look at some simple examples.

Example 1: $A(0:6) = B(1:7)$, with horizontal stripping and $L = 4$ (see figure 2(a).) The initial strip of A is $A(0:3)$, a slice; this corresponds to $B(1:4)$, which is *not* a slice.

This example suggests a SIMD strip mining optimization with no vector counterpart. For each subscript position in a vector assignment statement, the compiler chooses an array occurrence whose slices determine the strips. A suitable choice may reduce the number of ragged loads or stores required. For example, in $A(0:6) = B(1:7) + C(1:7)$, it is best to use B (or C) to determine the strips.

Example 2: $A(0:6) = B(1:7)$, with vertical stripping and $L = 4$ (see figure 2(b).) The initial strip of A is $A(0:6:2)$, corresponding to $B(1:7:2)$, both slices.

This is a nice feature of vertical stripping in general. The p-th slice consists of all $v \equiv p \pmod{M}$. If this slice is translated a distance b, we obtain all $v \equiv p + b \pmod{M}$, which is also a slice.

Example 3: $A(0:3, 0:3) = B(1:4, 2:5)$, with horizontal stripping and strip lengths of 4 in each dimension (see figure 3.) All of A fits into one slice (so this example is a model for what happens to a slice in the multi-dimensional case.) The corresponding strip for B consists of parts of 4 slices. In the n-dimensional case, if m dimensions are stripped horizontally and the rest vertically, then a slice of A corresponds to parts of 2^m slices of B.

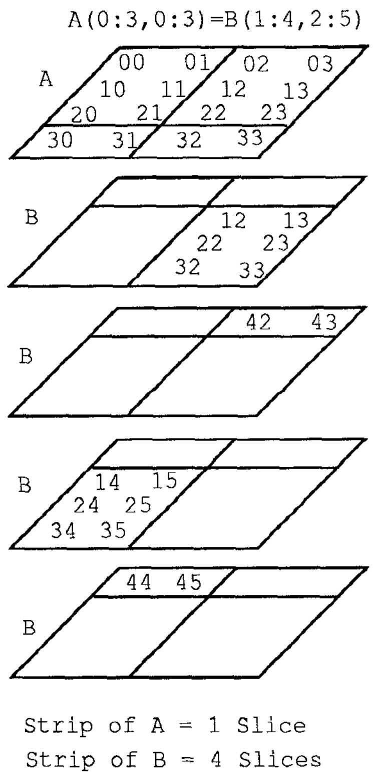


Figure 3: Two-dimensional Strips

Example 4: $A(0:3) = B(0:6:2)$, with horizontal stripping and strip length 4 (see figure 4(a).) We can use slices of A as strips, but the corresponding strips of B are not slices. In fact, a strip of B isn't even strictly speaking a strip, since processor 0 contains two elements, $B(0)$ and $B(4)$. We discuss what to do about this problem in Section 5.

Example 5: $A(0:3) = B(0:6:2)$, stripped vertically with strip length 4 and strip depth 2 (see figure 4(b).) A consists of two slices that together correspond to one slice of B. We can use slices of A to define the strips without violating our definition of a strip (at most element per processor). If we strip vertically to a depth of 3, then both sections decompose into 3 slices that correspond to one another, so we obtain the same strips regardless of which array section is used to define them.

With vertical stripping, we have to pay attention

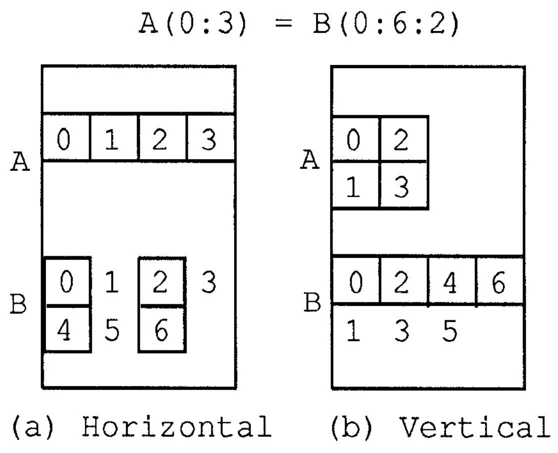


Figure 4: Different Strides

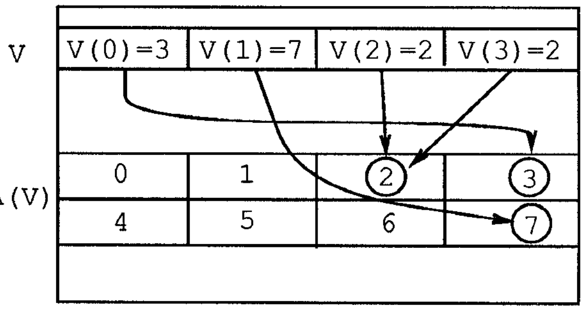


Figure 5: Vector Subscripts

to divisibility issues. Suppose the strip depth is M, and a triple $b : c : a$ conforms to a triple $b' : c' : a'$. We have a mapping from virtual coordinates to virtual coordinates: coordinate v of the first triple maps to $v' = \frac{a'}{a}v + (b' - \frac{a'b}{a})$ of the second triple. We ask now how a slice is mapped. Essentially we are asking how a linear mapping on the integers factors modulo M. Elementary number theory tells us that the mapping is well-defined (i.e., a slice maps into a slice) if a is relatively prime to M.

Example 6: $A(V(0:3))$, horizontally stripped with strip length 4, where $V(0:3) = [3,7,2,2]$ (see figure 5.) This is similar to the previous example. A slice of V corresponds to three elements of the array A. We have two kinds of conflicts. First, $V(2)$ and $V(3)$ point at the same element of A. Second, $V(0)$ and $V(1)$ point at the same processor, even though they point at different locations. Section 6 discusses how to deal with this. Vertical stripping of vector-valued subscripts leads to the same kind of problem.

5 Strip Mining Regions

The general method for strip mining a statement is to pick one of its array occurrences, and use the slices of this occurrence to define strips for the other (conformant) array occurrences.

Each dimension that is stripped has a strip mining loop around the entire statement. Thus we have a nest of strip mining loops around the strip mined statement.

As we have seen, strips are not always slices. In some cases, what correspond to slices are not strictly speaking strips— that is, they have more than one array element assigned to the same physical processor (see Example 4 of Section 4.)

In general, each statement tree could require internal "mini-loops". For example, consider $A(0:7) = B(0:7) + C(0:14:2) * D(0:14:2)$, stripped horizontally with strip length 4. There is one "outer" strip mining loop with two iterations. However, the strip $B(0:3)$ is to be added to $C(0:6:2) * D(0:6:2)$, and the computation of this product requires another "mini-loop" with two iterations. Differing strip lengths lead to the same kind of problem.

Such mini-loops are still supportable, but this approach quickly becomes unmanageable if one has to deal with expressions like $B(0:25:5, 1:15:2) + C(2:20:3, 0:49:7)$.

A simpler solution that still provides acceptable performance is to divide the statement tree into *strip mining regions*. A strip mining region is a group of nodes which all have the same strip length and same effective stride for all dimensions.

Crossing a strip mining region boundary means computing all strips of the subtree and reshaping it into a temporary that matches the shape of the parent. Example: $A(0:7) = B(0:7) + C(0:14:2) * D(0:14:2)$ would be transformed into two statements

```
temp(0:7) = C(0:14:2)*D(0:14:2)
A(0:7) = B(0:7) + temp(0:7)
```

The second statement can now be stripped without difficulty. The first statement is stripped according to the right hand side, and as each strip of the product is computed, it is stored into the temporary via a REMOTE_STORE. (We discuss an alternative to the REMOTE_STORE in Section 6.) Note that these two statements are stripped separately— the strip loops cannot be fused.

This method is simple and general. In special cases, some optimizations are desirable. First, suppose an array section B doesn't match its parent (different strip length or effective stride). Instead of reshaping B into a temporary that matches its parent (forcing a separate statement and the need for a temporary) one can just strip the statement according to the characteristics of the parent, and get a strip of B via a REMOTE_LOAD.

Certain kinds of mismatches can be handled by inserting a statement just before an inner loop. The key requirement is that one strip of a subexpression corresponds exactly to several strips of the parent node. For example, the subexpression may have a longer strip length in an index, or the parent's effective stride may be an exact multiple of the child's. One can write:

```
DO p =.....
  temp(p) = p-th strip of child
  DO q =....
    lhs(p,q) = ...temp(p,q)...
  ENDDO
ENDDO
```

where $temp(p)$ consists of $temp(p,1),\ldots,temp(p,r)$. A full-sized temp for the child is not necessary.

This optimization introduces an ordering constraint on the strip mining loops. Hence it cannot be performed on two dimensions at once. For the same reason, it can interfere with classic strip mining optimizations that interchange strip mining loops.

6 Stripping Loads and Stores

To make the language a little smoother, we sometimes discuss only loads, but the discussion applies to stores as well.

Slices are loaded via the basic uniform load mechanism of the SIMD machine, as discussed in Section 2.

Consider next cross-cuts that come from different offsets but identical strides. Examples 1 and 3 of Section 4 fall in this category. Two methods may be used:

- Each processor computes the memory address it needs to load values from. A ragged load is then performed.
- A series of uniform loads is performed, loading the various slices which make up the strip. In Example 3, there would be four loads. An n-dimensional case could require 2^n loads.

The two methods differ less than one might think at first, since the memory address must be assembled with the first method. However, it takes $2n$ different contexts to assemble the memory address for n dimensions. We have a tradeoff between one ragged load (and $2n$ context calculations) versus 2^n uniform loads (each with its own context calculation).

Consider next vector-valued subscripts, e.g., $A(W(LB : UB))$. We can use a REMOTE_LOAD, where processor v first loads $W(v)$, then computes the location of $A(W(v))$ and obtains this array element from the processor possessing it. The REMOTE_LOAD involves a loop with ragged loads inside. Another technique substitutes uniform loads for the ragged loads. Suppose one knows that the memory addresses of the section $A(W(LB : UB))$ all lie in the range $[g : h]$. One can strip the load on the range of values, thus:

```
Processor v loads W(v) and
computes the location of
A(W(v));

DO p = g,h
  Set context to those
  processors v such that
  A(W(v)) lies at memory
  address p;

  Change context to the
  processors containing
  these A(W(v)) and have
  them load the array
  elements at memory address
  p;

  Move A(W(v)) to processor v
 ENDDO
```

Changing the context and moving $A(W(v))$ both require interprocessor communication with this method (changing the context requires sending one bit). The same interprocessor communication is needed for the REMOTE_LOAD solution, however there the memory address must be sent instead of just a bit. Which method is superior depends on how tightly one can bound the range $[g : h]$, and the relative expense of uniform and ragged loads.

The REMOTE_LOADs and REMOTE_STOREs referred to in Section 5 can be eliminated by stripping on the range of values. The analysis is similar to the vector-valued subscript case; the chief differences are that one can always compute an expression for the range of memory addresses, and the context of "receiving" processors can be computed directly, instead of sending a bit.

7 Stripping Spreads

Stripping a spread, say $A(1 : M, 1 : N) =$ SPREAD($B(1 : M)$, DIM $= 2$, NCOPIES $= N$), depends on how the spread dimension is stripped. For example, suppose the second dimension of A is completely stripped and the first dimension is not stripped at all. Then the desired stripping is:

```
DO p=1,N
  A(1:M,p)=B(1:M)
ENDDO
```

If the second dimension is partially stripped, then one should load $B(1 : M)$, spread it along the second dimension so far as the strip length permits (call it L), and store the result in the strip mining loop:

```
temp(1:M,1:L)=SPREAD(B(1:M),
               DIM=2, NCOPIES=L)
DO p=1,N,L
  A(1:M,p:p+L-1)=temp(1:M,1:L)
ENDDO
```

Finally, if the first dimension is also stripped, then one has to put a strip mining loop for this dimension around all the above code. The order of the strip mining loops is constrained.

If the argument being spread is an expression, and if the SPREAD is part of a more complicated right hand side, then one computes one strip of the argument at a time and spreads it.

The reuse of a piece of the spread object is an optimization that is not always possible. Example: $A(1 : M, 1 : N) = \text{SPREAD}(B(1 : M), 2, N) + \text{SPREAD}(C(1 : N), 1, M)$. If both dimensions are stripped, then we have a conflict on the desired order of the strip mining loops.

As with grid motion, only certain fields may "support spreading" in the hardware. A binary tree broadcast algorithm is the best general method around this problem, though in special cases other algorithms may prove superior. We omit details, which are highly hardware specific.

By a *reduction*, we mean a transformational function that reduces rank (like a SUM). Stripping reductions is the "dual" of stripping spreads. More details are provided in the full version of this paper [11].

8 Allocation

Having surveyed the effects of various allocations on the strip mining of statements, we turn to the question of choosing the allocation in the first place.

Data optimization determines a set of cells. A cell may be used several times; each cell use is associated with a particular subscript position of an array occurrence. Each cell must be allocated a field in the processor address. The fields corresponding to all the cells for an array occurrence must partition the processor address.

A cell must accommodate a range of virtual coordinate values, determined by the collection of all its uses. If the associated field has too few bits to represent all these values, the cell must be stripped, either horizontally or vertically. For vertical stripping, a strip depth must be chosen, subject to the constraint that the product of the strip depth and the strip length exceed or equal the range of values to be represented.

Data optimization assumes that two array elements with the same virtual processor coordinates are aligned, i.e., belong to the same physical processor. This implies that all uses of the same cell must be stripped identically.

Cell allocation thus consists of allocating fields, choosing horizontal or vertical stripping, and choosing a strip depth (if necessary). Once this is done, the notion of slice is well-defined for each array occurrence. The strip miner still has the task of deciding how each statement is to be stripped, since strips are not always slices.

We list some of the important factors in allocating fields for cells.

Parallel usage: If a cell is never used in parallel, there is no reason to allocate it any bits. Each parallel use is a reason for allocating bits. Dynamic frequency of parallel use, in so far as it can be estimated at compile-time, favors giving it many bits.

Known extent: If the range of values the cell must accommodate is known at compile-time, then we know how many bits it needs. A cell use like $A(1 : N)$ can at best be given a default number of bits, and stripping code must always be generated, even though the array may fit in one strip at run-time.

Grid motion: If the cell is used in a place where grid motion is necessary, it is preferable to give it a field that has hardware support for grid motion. Hardware wraparound should also be considered, as well as the distance of shifts.

Reductions, Spreads: Likewise, hardware support for spreading and reductions must be considered.

Strip Mining Regions: Giving two cells used in the same expression different strip lengths may cause different strip mining regions, which costs a temporary, extra statements, and is generally less efficient. As noted in Section 5, there are times it is helpful to make sure a cell used in a child is longer or equal to the corresponding cell in the parent node in an expression tree.

Powers of Two: A cell use like $A(1:128)$ wastes no processors. In general, if a cell needs to represent E values, $\lceil \log_2 E \rceil - \log_2 E$ measures the degree of waste.

An extent of the form $k2^l$ for small integer k also is good for preventing waste. If the extent is close to but less than such a number, it is a good idea to allocate l bits to the cell. Example: consider $A(1:150, 1:150)$ on a machine with 1024 processors. If the first dimension is allocated 8 bits and the second dimension 2 (a 256×4 allocation), then only the second dimension requires a strip mining loop, with 38 iterations. Another allocation is suggested by the fact that 150 is close to $160 = 32 \times 5$: 5 bits each dimension (a 32×32 allocation) resulting in two strip mining loops with 5 iterations each, hence 25 iterations total— a savings of 44%.

Cell allocation points up the advantages of compile-time global knowledge. Consider the following code fragment:

```
S1:  A(1:32,1:32) = ....
     DO I=1,100
S2:        ....=....A(I,1:512)
     ENDDO
S3:  ....=....A(1:1024,1)
```

Suppose a 1K machine is the target. If data optimization decides to align all three occurrences of A, then the cost of executing this fragment is:

$$\begin{aligned} &\text{unit\_cost}(S1) \cdot \text{num\_strips}(S1) \\ +100 \cdot &\text{unit\_cost}(S2) \cdot \text{num\_strips}(S2) \\ +&\text{unit\_cost}(S3) \cdot \text{num\_strips}(S3) \end{aligned}$$

where unit_cost($S1$) is the cost of one iteration of the strip loop for $S1$ (likewise for $S2$ and $S3$). The unit cost may depend on how cells are allocated (because of grid motion, for example). Ignoring this aspect, and assuming the unit costs for $S1$, $S2$, and $S3$ are equal, the best allocation is 4×256 (2 bits for the first dimension and 8 for second). With this allocation, the execution cost is 464 times the unit cost, in comparison to 628 times unit cost with the allocation 2×512.

In general, one must balance the cost of motion against the gain in parallelism obtained by allocating different occurrences differently.

The choice between horizontal and vertical stripping is another aspect of cell allocation. Each method has its own benefits. In favor of vertical stripping, grid motion sometimes disappears (see the full version of this paper). Translated slices remain slices, causing loads and stores to be uniform instead of ragged (see Section 4).

Vertical stripping suffers the disadvantage of sometimes requiring more strip loop iterations. For example, suppose a cell is associated with occurrences $A(1:1000)$ and $A(1:2000)$ (in separate statements), and the target is a 1K machine. With horizontal stripping, the cell can be allocated a strip length of 1024, resulting in no stripping for the first occurrence and two strips for the second. With vertical stripping, the second occurrence demands a strip depth of 2, forcing two strips for both uses of the cell. The pair of occurrences $A(1:1000)$ and $A(1000:2000)$ cause the same problem (if they are assigned the same cell). In general, the total extent of virtual processor coordinates that the cell must cover determines the strip depth for vertical stripping, and hence the number of strip loop iterations for all uses of the cell.

In special cases, vertical stripping can use fewer iterations. If the strip depth is M and the effective stride of a use of a cell is S, then a strip loop with $M/\gcd(M,S)$ iterations is needed. For example, if $A(0:100:2)$ is stripped to depth 6, then only three iterations are needed:

```
DO p=0,4,2
  .... A(p:100:6) ....
ENDDO
```

While this kind of phenomenon may seem rather specialized, it occurs in Red-Black Jacobi relaxation, as was noticed by McBryan for the Connection Machine [7].

Vertical stripping destroys the loop reversal optimization [2]. For example, if $A(2:256) = 2*A(1:255)$ is stripped horizontally with strip length 64, then running the loop backwards avoids a temporary:

```
DO p=256,2,-64
  A(p:p-63:-1) = 2*A(p+1:p-62:-1)
ENDDO
```

(Dependence analysis shows that the strip mining loop cannot be run forward without introducing a temporary.) With vertical stripping, the temporary cannot be avoided:

```
DO p=0,3
  temp(p+2:256:4) = 2*A(p+1:255:4)
  A(p+2:256:4)    = temp(p+2:256:4)
ENDDO
```

9 Conclusions and Related Work

Allen and Kennedy [2] is the standard reference on vector strip mining. Prins and Smith [9] and Tomboulian and Pappas [10] both perform by-hand strip mining on the MasPar MP-1, for the bitonic sort and the Mandelbrot set respectively. These papers offer an interesting perspective on strip mining at the assembly code level. They also highlight the importance of ragged loads and stores (there termed *indirect addressing*). Christy [3] discusses SIMD virtual processors from a hardware perspective.

Comparing strip mining on vector and SIMD machines, the SIMD case presents greater complexity—or viewed positively, offers scope for new optimizations. This stems from a fundamental difference in the two architectures. In a vector machine, the data moves to the processor; in the SIMD case, the data is spread across processors. The notion of aligning operands does not apply in the vector case because the required motion is absorbed into the loading and storing of registers. We may draw an analogy with SIMD versus MIMD: synchronization disappears as an issue for SIMD machines because it is present at every step.

Completing the list of differences, vertical strip mining offers no advantages on vector hardware.

Compiler-managed strip mining (or virtualization) offers many benefits over the hardware approach. The classic vector strip mining optimizations apply. Intelligent cell allocation can reduce motion, increase parallelism, and make better use of communication hardware (e.g., grid instead of router motion). Ragged loads and stores can often be replaced with their uniform counterparts, either by vertical strip mining, or by choosing the right slice to define strips. The only significant drawbacks are increased compiler complexity, and increased code size.

10 Acknowledgments

I would like to thank Kathy Knobe, Venkat Natarajan, and Carl Offner for many helpful conversations and ideas.

References

[1] Eugene Albert, Kathleen Knobe, Joan D. Lukas, and Guy L. Steele Jr. Compiling Fortran 8x array features for the Connection Machine computer system. In *Symposium on Parallel Programming: Experience with Applications, Languages, and Systems*, New Haven, Connecticut, July 1988. ACM SIGPLAN.

[2] Randy Allen and Ken Kennedy. Vector register allocation. Report COMP TR86-45, Department of Computer Science, Rice University, Houston, Texas, April 1986.

[3] Peter Christy. Virtual processors considered harmful, April 1991. To appear in Proceedings of the 6th Distributed Memory Computer Conference.

[4] Kathleen Knobe, Joan D. Lukas, and Guy L. Steele, Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, February 1990.

[5] Kathleen Knobe and Venkataraman Natarajan. Data optimization: Minimizing residual interprocessor data motion on SIMD machines. In *Frontiers '90: 3rd Symposium on the Frontiers of Massively Parallel Computation*, pages 416–423, College Park, Maryland, October 1990. University of Maryland.

[6] Jingke Li and Marina Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. Technical Report YALEU/DCS/TR-725, Yale University, November 1989.

[7] Oliver McBryan. Numerical computation on massively parallel hypercubes. In *2nd Conference on Hypercube Multiprocessors*, October 1986.

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

[9] Jan F. Prins and J.A.Smith. Parallel sorting of large arrays on the MasPar MP-1. In *Frontiers '90: 3rd Symposium on the Frontiers of Massively Parallel Computation*, pages 59–64, College Park, Maryland, October 1990. University of Maryland.

[10] Sherryl Tomboulian and Matthew Pappas. Indirect addressing and load balancing for faster solution to Mandelbrot set on SIMD architectures. In *Frontiers '90: 3rd Symposium on the Frontiers of Massively Parallel Computation*, pages 443–450, College Park, Maryland, October 1990. University of Maryland.

[11] Michael Weiss. Strip mining on SIMD architectures (extended version). Technical Report CA-9011-2901, Compass, Inc., November 1990.