Array Layouts for Parallel Processing

February 19, 2010

Byline

Paul Feautrier, Emeritus Professor of Computer Science
Ecole Normale Supérieure de Lyon
46, Allée d'Italie
69364 LYON CEDEX, France
Paul.Feautrier@ens-lyon.fr

Synonyms

Related Entries

SIMD architecture
permutation networks

Definitions

A high-performance architecture needs a fast processor, but the fastest processor is useless if a memory subsystem does not provide data at the rate of several words per clock cycle. Run-of-the-mill memory chips in today technology have a latency of the order of ten to a hundred processor cycles, far more than the necessary performance. The usual method for increasing the memory bandwith as seen by the processor is to implement a cache, i.e. a small but fast memory which is geared to hold frequently used data. Caches work best when used by programs with almost random but non-uniform addressing patterns. However, high-performance applications, like linear algebra or signal processing have a tendency to use very regular adressing patterns, which

degrade cache performance. In linear algebra codes, and also in stream processing, one find long sequences of accesses to regularly increasing adresses. In image processing, a template moves regularly across a pixel array.

To take advantage of these regularities, in fine-grain parallel architectures, like SIMD or vector processors, the memory is divided into several independent banks. Adresses are evenly scattered among the banks. If B is the number of banks, then word x is located in bank $x \bmod B$ at displacement $x \div B$. The low order bits of x (the byte displacement) do not participate in the computation. In this way, words in consecutive addresses are located in distinct banks, and can, with the proper interface hardware, be accessed in parallel. A problem arise when the requested words are not at consecutive addresses. *Array layouts* were invented to allow efficient access to various *templates*.

Discussion

Parallel Memory Access There are many ways of taking advantage of such a memory architecture. One possibility is for the processors to access in one memory cycle the B words which are necessary for a piece of computation, and then to cooperate in computing the result. Consider for instance the problem of summing the elements of a vector. One reads B words, adds them to an accumulator, and proceeds to the next B words.

Another organization is possible if the program operations can be executed in parallel, and if the data for each operation are regularly located in memory. Consider for instance the problem of computing the weighted sum of three rows in a TV image (downsampling). One possibility is to read three words in the same column and do the computation. But since it is likely that B is much larger than 3, the memory subsystem will be underutilized. The other possibility is to read B words in the first row, B words in the second row, and so on, do the summation, and store B words of the result. This method is more efficient, but places more constraints on the target application.

Templates and Template Size In both situations, the memory is addressed throught *templates*, i.e. finite sets of cells which can move accross an array. Consider for instance a vector:

```
float A[100];
```

The template $(0, 1, 2, 3)$ represents four consecutive words. If located (or *anchored*) at position i, it covers elements `A[i]`, `A[i+1]`, `A[i+2]` and `A[i+3]`

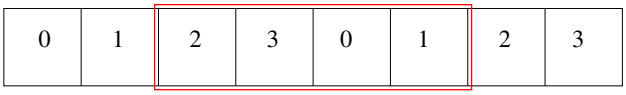

Figure 1: A vector layout

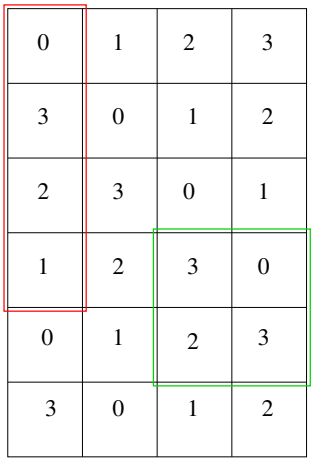

Figure 2: A matrix layout

of vector `A`. But the most interesting case is that of a two dimensional array, which may be a matrix or an image. The template $((0,0),(0,1),(1,0),(1,1))$ represents a two by two square block. If anchored at position (i,j) in a matrix `M`, it covers elements `M[i][j], M[i][j+1], M[i+1][j]` and `M[i+1][j+1]`. The first template is one-dimensional, and the second one is two-dimensional. One may consider templates of arbitrary dimensions, but these two are the most important cases.

The basic problem is to distribute arrays among the available memory banks in such a way that access to all elements of the selected template(s) takes only one memory cycle. It follows that the size of a template is at most equal to the number of banks, B. As has been seen before, memory usage is optimized when templates have exactly B cells; this is the only choice that is discussed here.

Observe also that, as the previous example has shown, there may be many template selections for a given program. The problem of selecting the best one will not be discussed here.

Layouts A layout is an assignment of bank numbers to array cells. A template can be viewed as a stencil that moves accross an array and shows

bank numbers throught its openings. A layout is *valid* for a template position if all shown bank numbers are distinct. In Fig. 1, the layout is valid for all positions of the red template. In Fig. 2, the layout is valid for all positions of the red template, and valid for no position of the green template.

The reader may have noticed that the information given by array layouts such as Fig. 1 or 2 is not complete. One must also know at which address each cell is located in its bank. This information can always be retrieved from the layout. Simply order the array cells, and allocate each cell to the first free word in its bank. For Fig. 2, the bank address is found equal to the row number. This scheme imposes two constraints on a layout:

- array cells must be evenly distributed among banks,
- the correspondance between cells and bank addresses cannot be too complex, since each processor must know and use it.

Depending on the application, one may want that a layout be valid at some positions in the array, or at every possible position. For instance, in linear algebra, it is enough to move the template in such a way that each entry is covered once and only once. One says that the template tesselates or *tiles* the array domain. Once a tiling has been found one simply number each square of the template from 0 to $B-1$, and reproduce this assignment (perhaps after a permutation) at all positions of the tile. This insures that the array cells are evenly distributed among the banks. In other applications, like image processing, the template must be moved at every position in the array. Imagine for instance that a smoothing operator must be applied at all pixels of an image. In fact, the two situations are equivalent: if a template tiles an array and if block numbers are assigned as explained above, then the layout is valid for every position. For a prof, see [4].

Skewing Schemes: Vector Processing Consider the following example:

```
for(i=0; i< n; i=i+1)
  X[i] = Y[i] + Z[i];
```

to be run on a SIMD architecture with B processing elements. If the layout of `Y` is such that `Y[i]` is stored in bank i mod B at address $i \div B$, B elements of `Y` can be fetched in one memory cycle. After another fetch of B words of Z, the SIMD processor can execute B addition in parallel. Another memory cycle is needed to store the B results. Note that this scheme has the added advantage that all groups of B words are at the same address in each bank, thus fitting nicely in the SIMD paradigm.

Suppose now that the above loop is altered to read:

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3

(a) $B = 4, S = 0$: four cycles are needed to access column 0

0,0	0,1	0,2	0,3
1,3	**1,0**	1,1	1,2
2,2	2,3	**2,0**	2,1
3,1	3,2	3,3	**3,0**

(b) $B = 4, S = 1$: column 0 can be accessed in one cycle

Figure 3: The effect of skewing

```
for(i=0; i< n; i=i+1)
  X[d*i] = Y[d*i] + Z[d*i];
```

where d is a constant (the step). The block number $d.i \bmod B$ is equal to $d.i - k.B$ for some k, and hence is a multiple of the greatest common divisor (gcd) g of d and B. It follows that only B/d words can be fetched in one cycle, and the performance is divided by g. Since B is usually a power of 2 in order to simplify the computation of $x \bmod B$ and $x \div B$, a loss of performance will be incurred whenever d is even. One can always invent layouts that support non-unit step accesses. For instance, if $d = 2$, assign cell i to bank $(i \div 2) \bmod B$. However, in many algorithms that use non-unit steps, like the Fast Fourier Transform, d is a variable. It is not possible to accomodate all its possible values without costly remapping.

Skewing Schemes: Matrix Processing In C, the canonical method for storing a matrix is to concatenate its rows in the order of their subscripts. If the matrix is of size $M \times N$, the displacement of element (i, j) is $N.i + j$. In Fortran, the convention is reversed, but the conclusions are the same, *mutatis mutandis*. One may assume that the row length N is a multiple of B, by padding if necessary; hence, all matrices may be assumed to have size $M \times B$. As a consequence, element (i, j) is allocated to bank $j \bmod B$.

If the algorithm accesses a matrix by rows, as in:

```
for(j=0; j<N, j++)
  ... X[i][j]...
```

performance will be maximal. However, when accessing columns, consecutive items will be separated by B rows, and be located all in the same bank. All parallelism will be lost.

A better solution is *skewing*. The cell (i, j) is now allocated to bank $(S.i + j) \bmod B$. S is the skewing factor. Here, two consecutive cells in a row are still at a distance of 1; hence, a row can still be accessed in one cycle.

But two consecutive cells in a column, (i, j) and $(i + 1, j)$ are at a distance of S. If S is selected to be relatively prime to B (for instance, $S = 1$ if B is even) access to a column will also take one cycle.

This raise the question if other patterns can be accessed in parallel. Consider the problem of accessing the main diagonal of a matrix. In the above skewing scheme, the distance from cell (i, i) to $(i + 1, j + 1)$ is $S + 1$, and S and $S + 1$ cannot be both odd. Hence, accessing both columns and diagonals (or both rows and diagonals) in parallel with the same skewing scheme is impossible when B is even.

Skewing Schemes: More Templates It is clear that a more general theory is needed when the subject algorithm needs to use several templates at the same time. As seen above, a template has a valid layout if and only if it tiles the two dimensional plane. This results seems natural if the template is required to cover once and only once each cell of the underlying array. If banks numbers are assigned in the same order in each tile ([4]) or are regularly permuted from tile to tile ([3]) then the layout will obviously be valid for each tile. The striking fact is that the layout will also be valid when the template is offset from a tile position. Observe that the previous results on matrix layouts corresponds to the many ways of tiling the plane with B-sized linear templates.

A tiling is a defined by a set of vectors $v_1, \ldots, v_n$ (translations). If a template instance has been anchored at position (i, j), then there are other instances at positions $(i, j) + v_1, \ldots, (i, j) + v_n$. The translations $v_1, \ldots, v_n$ must be selected in such a way that the translates do not overlap.

If an algorithm needs several templates, the first condition is that each of them tiles the plane. But each tiling defines a layout, and these layouts must be compatible. One can show that the compatibility condition is that the translation vectors are the same for all templates. Since a template may tile the plane in several different ways, finding the right tiling may prove a difficult combinatorial problem.

Pipelined Processors and GPU Consider the loop:

```
for(i=0; i<n; i=i+1)
  s = s + A[i];
```

to be run on a pipelined processor which executes one addition per cycle. If the memory latency is equal to B processor cycles, the memory banks can be activated in succession, in such a way that, after a prelude of B cycles, the processor receives an element of `A` at each cycle. As for parallel processors,

	0	1	2	3
0	**0**	1	2	3
1	2	**3**	0	1
2	0	1	**2**	3
3	2	3	0	**1**

Figure 4: The need for reordering

if the loop has a step of d, the performance will be reduced by a factor of $\gcd(d, B)$.

While today vector processors are confined to niche applications, the same problem is re-emerging for Graphical Processing Units (GPU), which can access B words in parallel under the name "Global Memory Access Coalescing".

Reordering With non standard layouts, the results of a parallel access may not be returned in the natural order. Consider for instance the problem of having parallel access both for rows and for the main diagonal of a 4×4 matrix on a 4 banks memory. The skewing factor S must be such that $S + 1$ is relatively prime to 4: $S = 2$ is a valid choice. The corresponding layout is shown on Fig. 4. The integer in row i and column j is the number of the bank which holds element (i, j) of the matrix. One can see that bank 0 holds element $(0, 0)$, bank 1 holds element $(3, 3)$, and so on. If the banks are connected to processors in the natural order the elements of the main diagonal are returned in the order $(0, 0), (3, 3), (2, 2), (1, 1)$. This might be unimportant in some cases. For instance if the elements of the main diagonal are to be summed (the *trace* computation), since addition is associative and commutative.

In other cases, returned values must be reordered. This can be done by inserting a B words fast memory between the main memory subsystem and the processors. One can also use a *permutation network* which may route a datum from any bank to any processor.

The Number of Banks Remember that in vector processing, when the step d is equal to one, performance is degraded by a factor $\gcd(d, B)$. It is tempting to use a prime number for B, since then the gcd will always be one except when d is a multiple of B. However, having B a power of two considerably simplifies the addressing hardware, since computing $x \bmod B$ and $x \div B$ is done just by routing wires. This has lead to the search for prime numbers with easy division, which are of one of the forms $2^p \pm 1$. See [2] for

a discussion of this point.

References

[1] P. Budnik and D.J. Kuck. The organisation and use of parallel memories. *IEEE Trans. on Computers*, C-20:1566–1569, December 1971.

[2] Benoît Dupont de Dinechin. A ultra fast euclidean division algorithm for prime memory systems. In *Supercomputing'91*, pages 56–65, 1991.

[3] W. Jalby, J.-M. Frailong, and J. Lenfant. Diamond schemes: An organization of parallel memories for efficient array processing. Technical Report RR-342, INRIA, 1984.

[4] H. D. Shapiro. Theoretical limitations on the efficient use of parallel memories. *IEEE Trans. on Computers*, C-27:421–428, May 1978.